Stride manual

These pages contain information about how to use Stride, an open-source C# game engine.

Stride community toolkit

Check out our Stride community toolkit for additional helpers and extensions.

Improve this documentation

The Stride documentation is open source, so anyone can edit it. If you find a mistake, you can correct it or comment in GitHub.

To edit any page of this manual, click the **Edit this page** link at the bottom. Please make sure to follow the writing guidelines.

Latest documentation

Recent updates

- Manual
  - **Updated** Scripts - Best Practices - More content added
  - **Important** Physics - Script a trigger - Example updated
  - **Important** Physics - Triggers - Example updated
  - **Updated** Graphics - Custom shaders - Troubleshooting section added

- New Glossary - New glossary section added

Previous updates

- Manual
  - **Updated** Graphics - Materials - Materials for developers - Modifying parameters at runtime added
  - New Scripts - Best Practices docs added
  - New Physics - Bepu Physics docs added
  - **Updated** Bullet Physics - Bullet Physics docs moved
  - **Updated** Files and Folders - Game distribution steps updated
  - **Updated** Scripts - Types of script - Asynchronous script example improved
  - New Scripts - Gizmos - Description and example of the Flexible Processing system
  - New ECS - Flexible Processing - Description and example of the Flexible Processing system
  - **Updated** Linux - Setup and requirements - Fedora OS option added
  - New Scripts - Serialization - Explanation of serialization
  - **Updated** Scripts - Public properties and fields - Content improvements and additions
  - New Engine - Entity Component model - Usage - Explanation of ECS usage
  - **Updated** Engine - Entity Component model - Content improvements
  - **Updated** Stride for Unity® developers - Content improvements
- Tutorials
  - **Updated** Tutorials - Quick Tutorials section added
  - **Updated** Tutorials - Added lesson counts and total length
- Contributing
  - **Updated** Contributing - Roadmap - GitHub Project - Roadmap link added
  - New Contributing - Core team - The Stride core team
  - **Updated** Contributing - Roadmap - Status added

Development Requirements

General requirements

To develop projects with Stride, you need:

Requirement	Specifications
Hard drive space	5GB
Operating system	Windows 10, 11 [see (1)]
CPU	x64
GPU	Direct3D 10+ compatible GPU
RAM	4GB (minimum), 8GB (recommended) [see (2)]
.NET SDK	8+ [see (3)]

(1) Earlier versions of Windows *may* work but are untested.

(2) RAM requirements vary depending on your project:

- Developing simple 2D applications doesn't require much RAM.
- Developing 3D games with lots of assets requires larger amounts of RAM.

(3) .NET SDK is being downloaded with the Stride installer.

Mobile development requirements

To develop for mobile platforms, you also need:

Platform	Requirements
Android	Xamarin [see (4)]
iOS	Mac computer, Xamarin [see (4)]

(4) Xamarin is included with Visual Studio installations. For instructions on installing Xamarin with Visual Studio, see this MSDN page.

Running Stride Games

To run games made with Stride, you need:

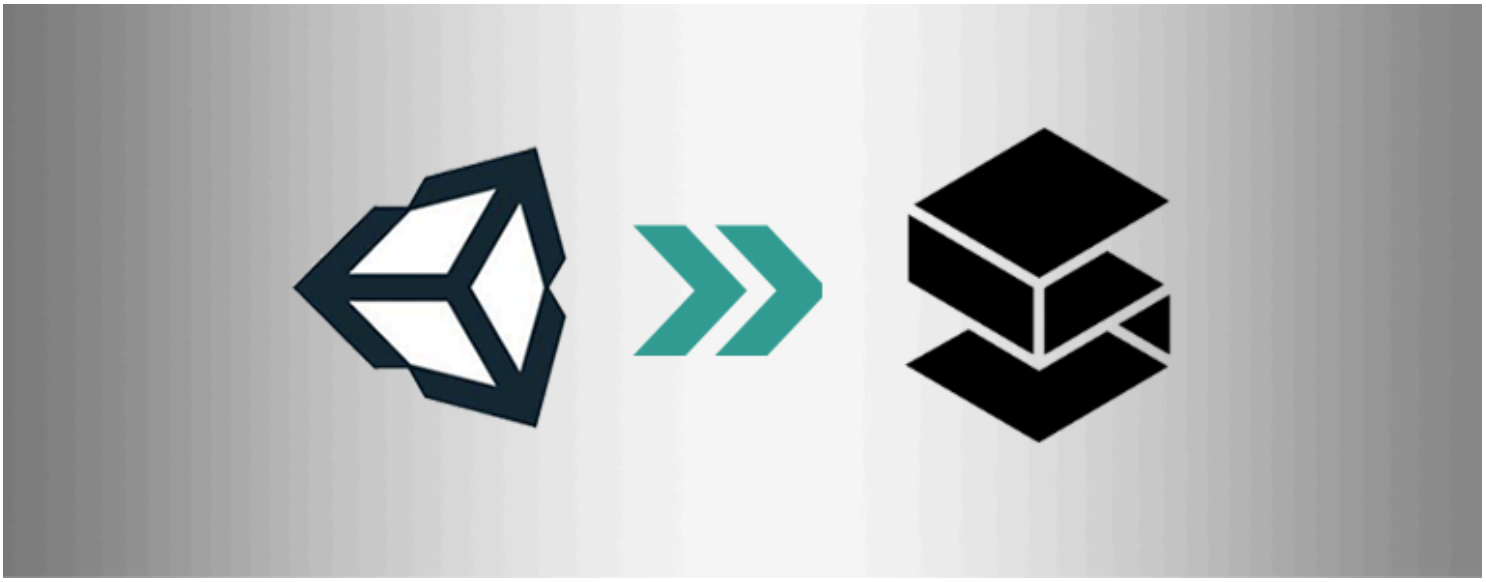
- .NET 10 if your application is not self-contained
- DirectX11 (included with Windows 10 and later), OpenGL, or Vulkan depending on the platform, and the graphics API override set in your `.csproj`
- Visual C++ 2015 runtimes (x86 and/or x64, depending on what you set in your project properties in Visual Studio)

Supported Platforms

For information about platforms Stride supports, see Platforms.

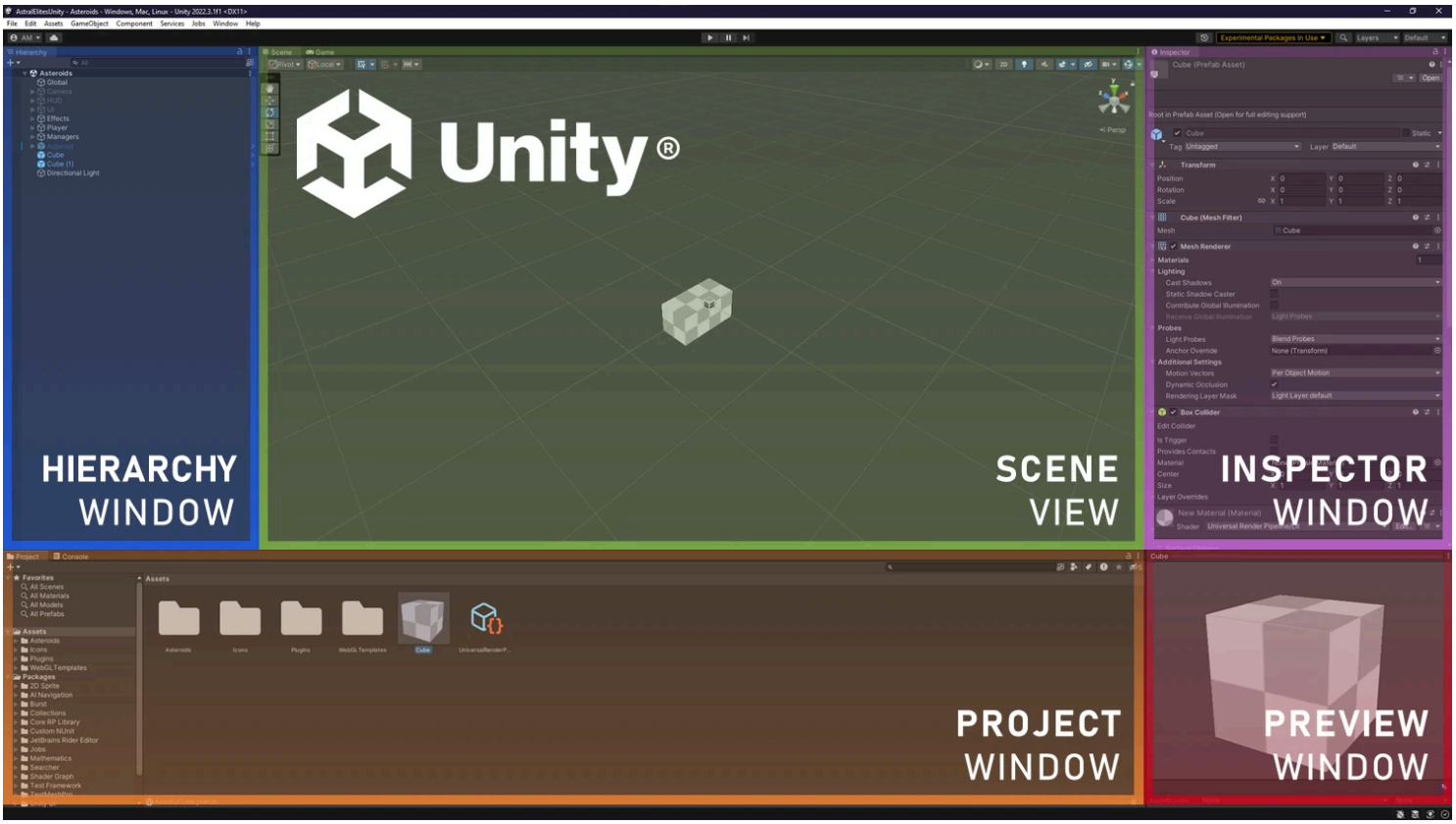
Stride for Unity® developers

Stride and Unity® both use C# and share many concepts, with a few major differences.



Editor

The Stride editor is **Game Studio**. This is the equivalent of the Unity® Editor.



You can customize the Game Studio layout by dragging tabs, similar to Visual Studio.

For more information about Game Studio, see the Game Studio page.

Terminology

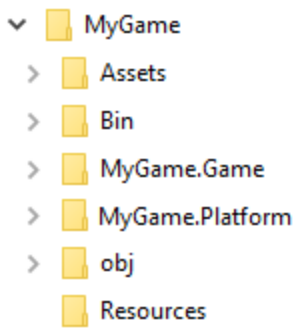
Unity® and Stride use mostly common terms, with a few differences:

Unity®	**Stride**
Hierarchy Window	Entity Tree
Inspector Window	Property Grid
Project Window	Asset View
Scene View	Scene Editor
`GameObject`	`Entity`
`MonoBehaviour`	`SyncScript`, `AsyncScript`, `StartupScript`

Folders and files

Like Unity®, Stride projects are stored in a directory that contains:

- The project `.sln` solution file, which you can open with Game Studio or any IDE such as Visual Studio
- A **MyGame.Game** folder with project source files, dependencies, resources, configurations, and binaries



- **Assets** contains asset configuration files.
- **Bin** contains the compiled binaries and data. Stride creates the folder when you build the project, with a subdirectory for each platform.
- **MyPackage.Game** contains your source code.
    - **MyPackage.Platform** contains additional code for the platforms your project supports. Game Studio creates folders for each platform (e.g. *MyPackage.Windows*, *MyPackage.Linux*, etc.). These folders are usually small and only contain the entry point of the program.
- **obj** contains cached files. Game Studio creates this folder when you build your project. To force a complete asset and code rebuild, delete this folder and build the project again.
- **Resources** is the recommended location for storing source files for your project, such as textures, models, and audio files.

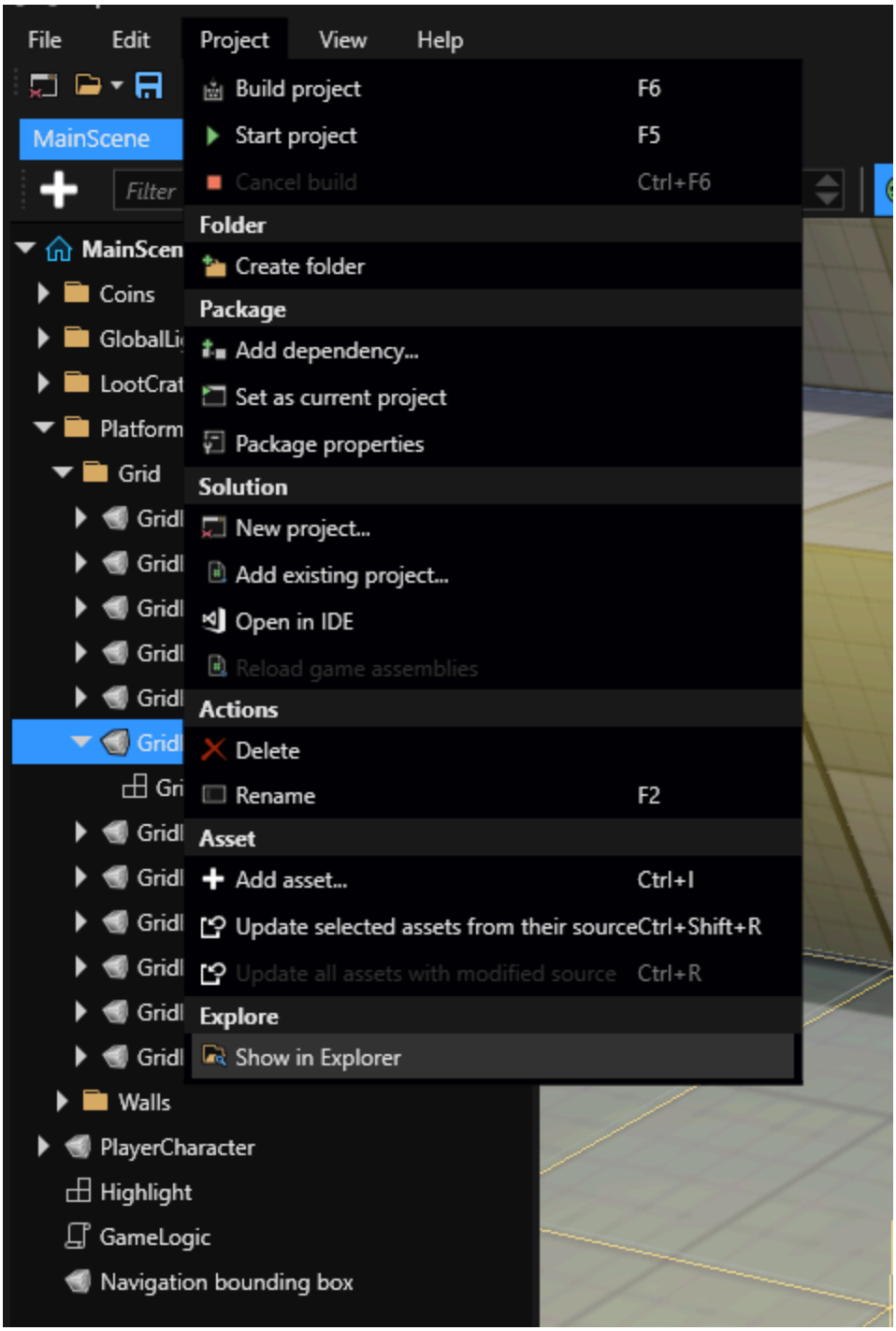
Stride and Unity® differ in the following ways:

- Stride doesn't automatically copy resource files to your project folder when you import them into assets. You have to do this yourself. We recommend you save them in the **Resources** folder.
- Stride doesn't require resource files and asset files to be in the same folder. You can save resource files in the Assets folder if you want, but instead, we recommend you save them in the **Resources** folder. This makes sharing your project via version control easier.

For more information about project structure in Stride, including advice about how to organize and share your files, see the Project structure page.

Open the project directory from Game Studio

You can open the project directory from **Project > Show in explorer** in Game Studio.



Game settings

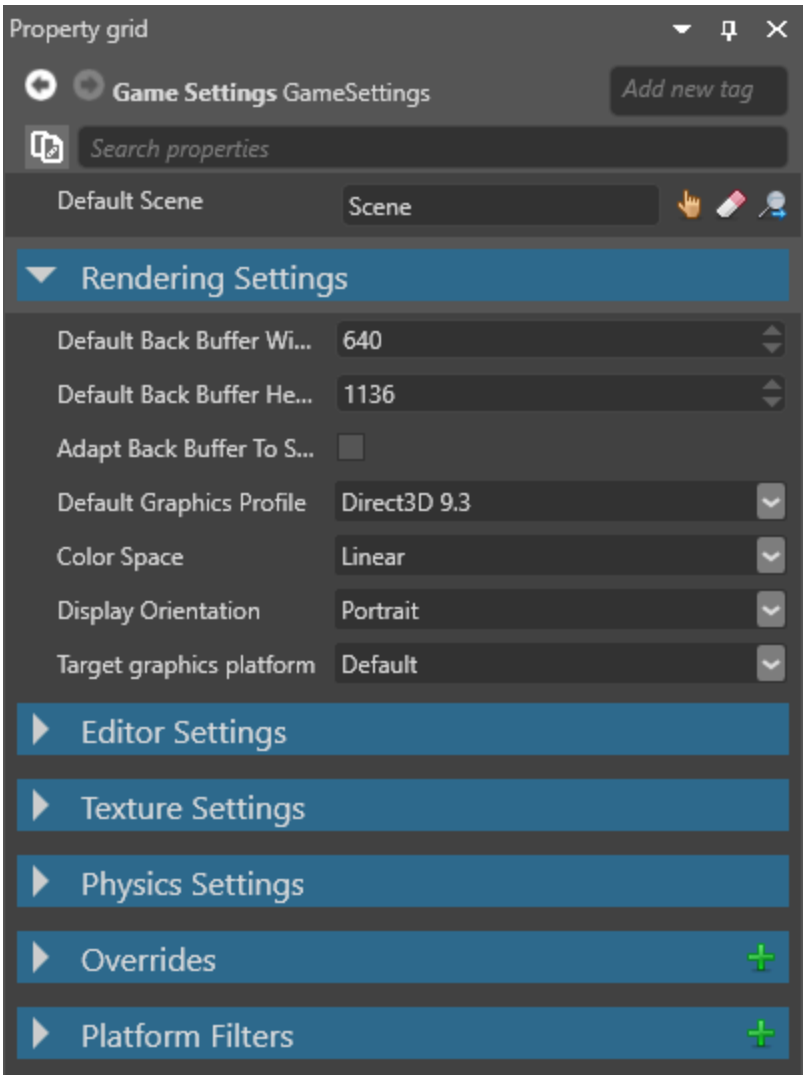
Unity® saves global settings in separate assets (i.e. Graphics Settings, Quality Settings, Audio Manager, and so on).

Stride saves global settings in a single asset, the **Game Settings** asset. You can configure:

- The **default scene**
- **Rendering settings**
- **Editor settings**
- **Texture settings**

- **Physics settings**
- **Overrides**

To use the Game Settings asset, in the **Asset View**, select **GameSettings** and view its properties in the **Property Grid**.



Scenes

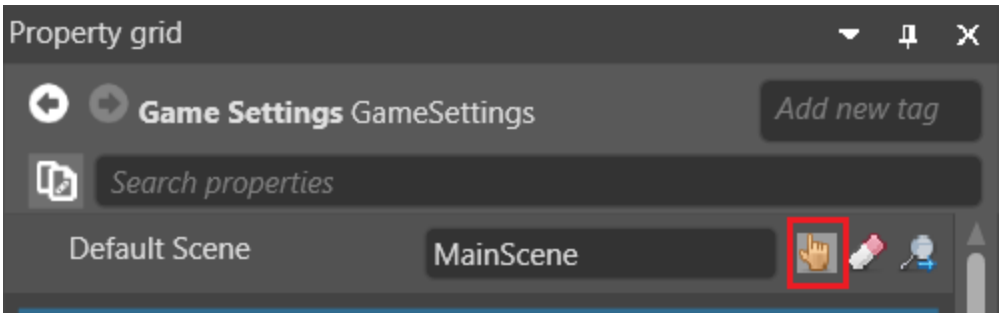
Like Unity®, in Stride, you place all objects in a scene. Game Studio stores scenes as separate `.sdscene` assets in your project directory.

Set the default scene

You can have multiple scenes in your project. The scene that loads up as soon as your game starts is called the *Default Scene*.

To set the default scene:

1. In the **GameSettings** properties, next to **Default Scene**, click (**Select an asset**).



The **Select an asset** window opens.

2. Select the default scene and click **OK**.

For more information about scenes, see Scenes.

Entities vs GameObjects

In Unity®, objects in the scene are called **GameObjects**. In Stride, they're called **entities**.

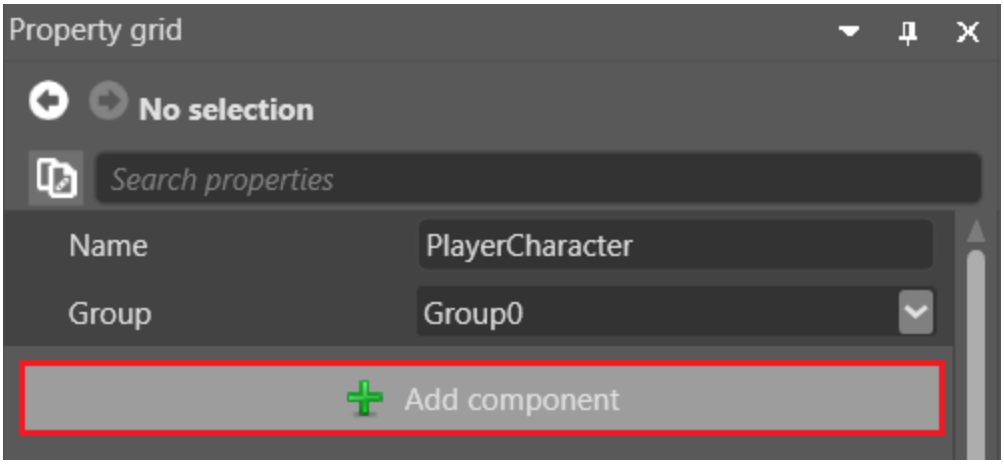
Like GameObjects, entities are carriers for components such as transform components, model components, audio components, and so on. If you're used to working with GameObjects in Unity®, you should have no problem using entities in Game Studio.

Entity components

In Stride, you add components to entities just like you add components to GameObjects in Unity®.

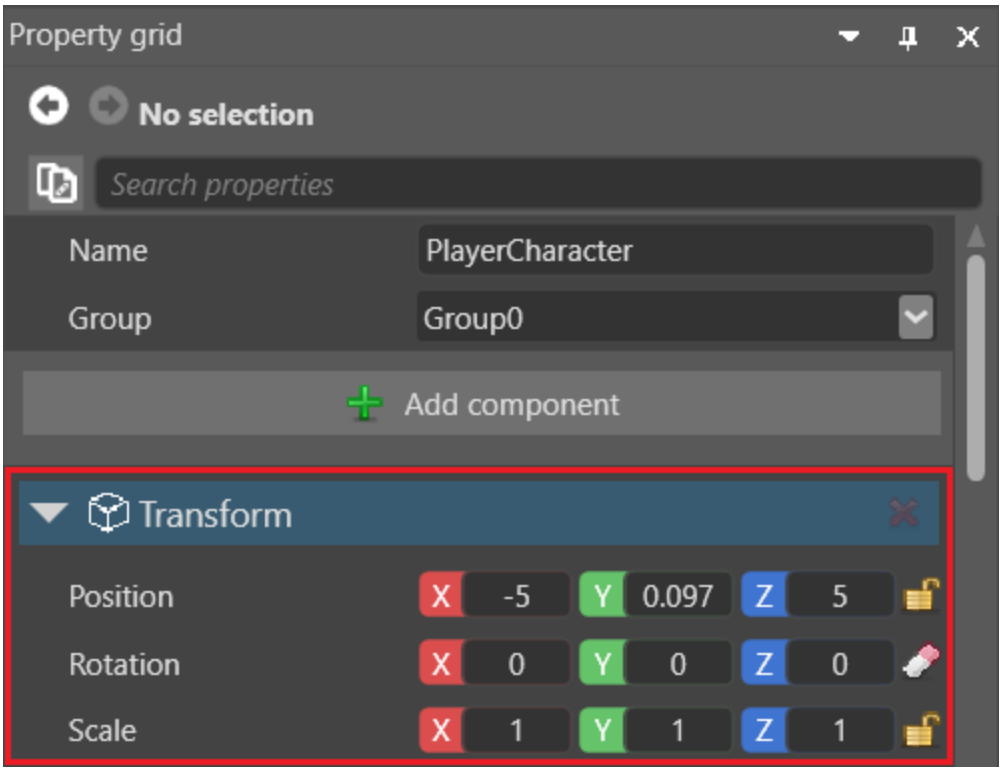
To add a component to an entity in Game Studio:

1. Select the entity you want to add the component to.
2. In the **Property Grid** (on the right by default), click **Add component** and select the component from the drop-down list.



Transform component

Like GameObjects in Unity®, each entity in Stride has a Transform component which sets its position, rotation, and scale in the world.



All entities are created with a Transform component by default.

In Stride, Transform components contain a LocalMatrix and a WorldMatrix that are updated in every Update frame. If you need to force an update sooner than that you can use

`TranformComponent.UpdateLocalMatrix()`, `Transform.UpdateWorldMatrix()`, or `Transform.UpdateLocalFromWorld()` to do so, depending on how you need to update the matrix.

Local Position/Rotation/Scale

Stride uses position, rotation, and scale to refer to the local position, rotation, and scale.

Unity®	Stride
`transform.localPosition`	`Transform.Position`
`transform.localRotation`	`Transform.Rotation`
`transform.localScale`	`Transform.Scale`
`transform.localEulerAngles`	`Transform.RotationEulerXYZ`

World Position/Rotation/Scale

In comparison to Unity, many of the Transform component's properties related to its location in the world have been moved to the WorldMatrix.

Unity®	Stride
`transform.position`	`Transform.WorldMatrix.TranslationVector`
`transform.rotation`	N/A
`transform.scale`	N/A
`transform.eulerAngles`	`Transform.WorldMatrix.DecomposeXYZ(out Vector3 rotation)`
`transform.scale` and `transform.position`	`Transform.WorldMatrix.Decompose(out Vector3 scale, out Vector3 translation)`
`transform.scale`, `transform.rotation`, and `transform.position`	`Transform.WorldMatrix.Decompose(out Vector3 scale, out Quaternion rotation, out Vector3 translation)`

> **NOTE**
>
> `WorldMatrix` is only updated after the entire Update loop runs, which means that you may be reading outdated data if that object's or its parent's position changed between the previous frame and now. To ensure you're reading the latest position and rotation, you should force the matrix to update by calling `Transform.UpdateWorldMatrix()` before reading from it.

Transform Directions

Unlike Unity, Stride provides a Backward, Left, and Down property. Note that those are matrix properties, so setting one of those is not enough to properly rotate the matrix.

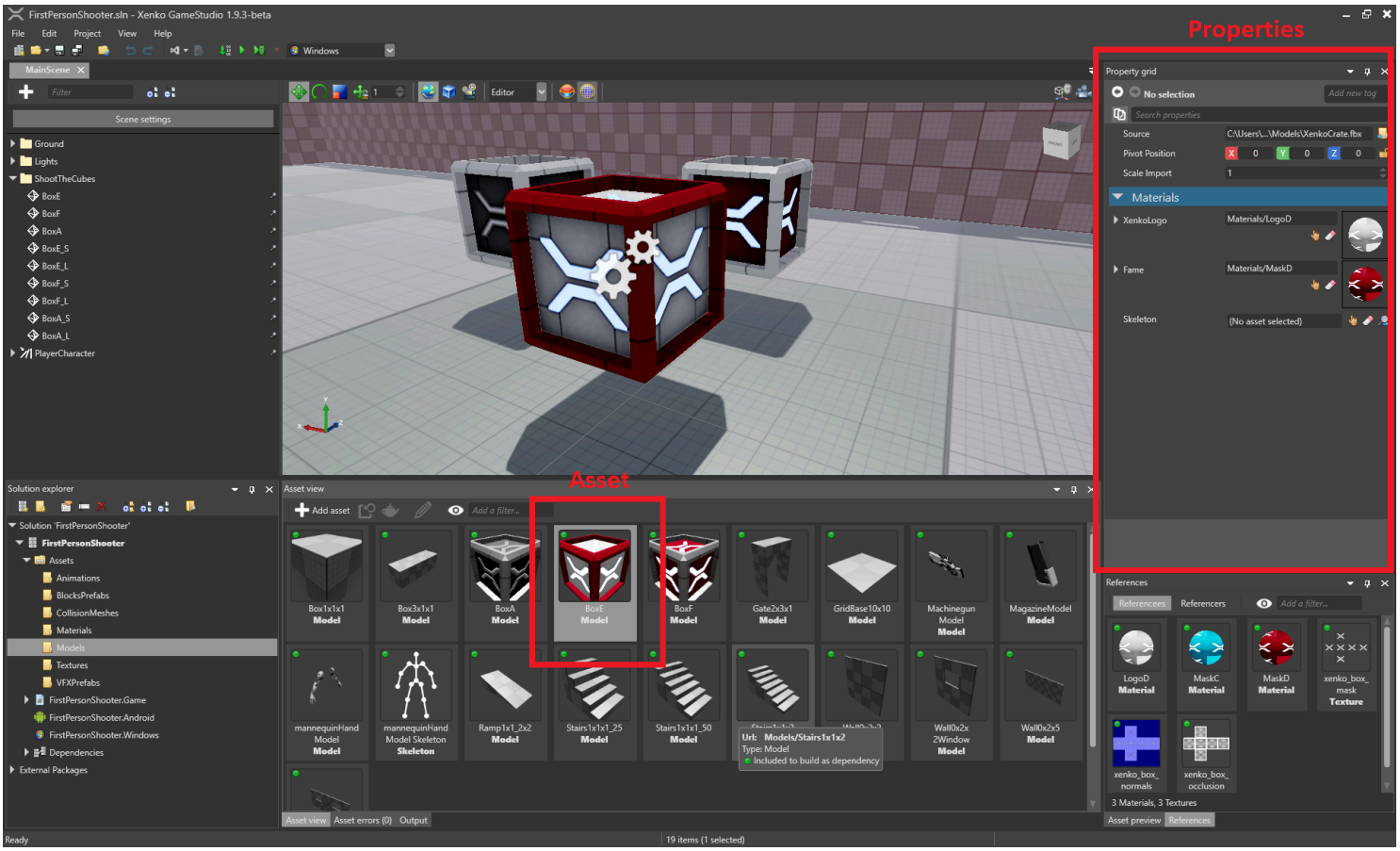
Unity®	Stride
`transform.forward`	`Transform.WorldMatrix.Forward`
`transform.forward * -1`	`Transform.WorldMatrix.Backward`
`transform.right`	`Transform.WorldMatrix.Right`
`transform.right * -1`	`Transform.WorldMatrix.Left`
`transform.up`	`Transform.WorldMatrix.Up`
`transform.up * -1`	`Transform.WorldMatrix.Down`

> **NOTE**
>
> See note in World Position/Rotation/Scale

Assets

In Unity®, you select an asset in the **project browser** and edit its properties in the **Inspector** tab.

Stride is similar. You select an asset in the **Asset View** and edit its properties in the **Property Grid**.



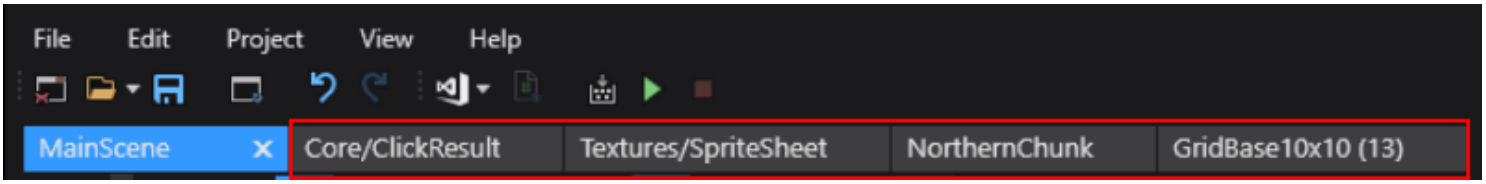
For certain types of assets, Game Studio also has dedicated editors:

- prefabs
- scenes
- sprite sheets
- UI pages
- UI libraries
- scripts

To open the dedicated editor for these types of assets:

- double-click the asset, or
- right-click the asset and select Edit asset, or
- select the asset and type Ctrl + Enter

The editor opens in a new tab. You can arrange the tabs how you like, or float them as separate windows, just like tabs in web browsers.



> **NOTE**
>
> When you modify resource files outside Game Studio, the corresponding assets update automatically in Game Studio.

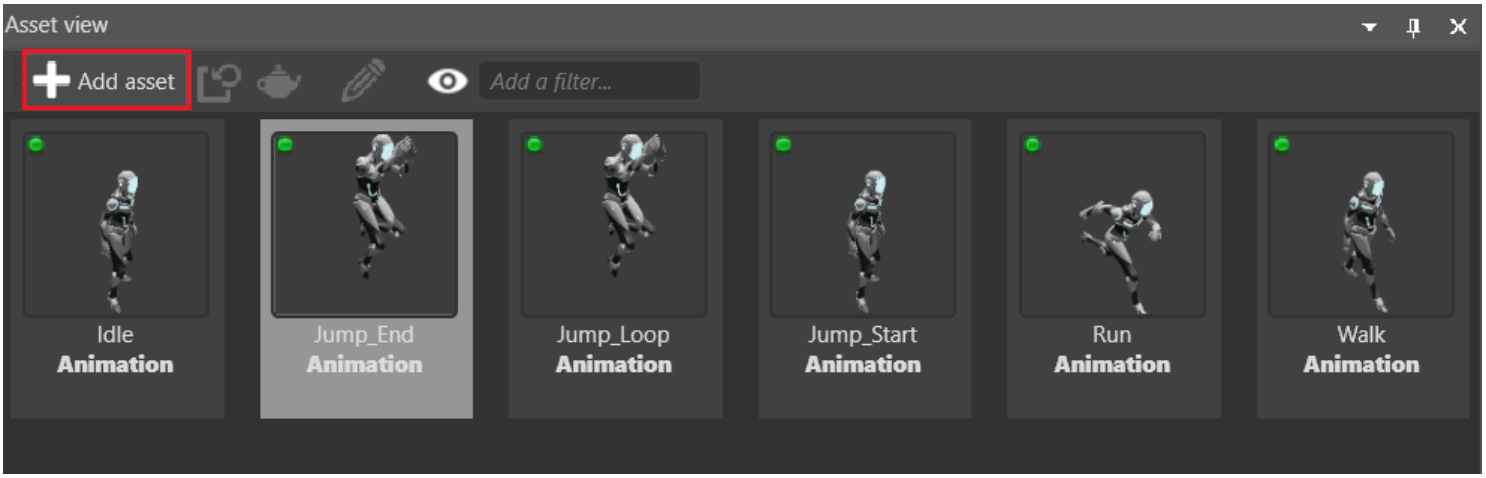
Scriptable Objects

See the Custom Assets page.

Import assets

To import an asset, drag it from Explorer to the **Asset View**. You can also click an **Add asset** button, navigate to the desired file, and specify the type of asset you want to import.

As soon as you add an asset to your project, you can edit its properties in the **Property Grid**.



> **NOTE**
>
> Unlike Unity®, Stride doesn't automatically copy resource files to the project directory when you import them to projects.

Supported file formats

Like Unity®, Stride supports file formats including:

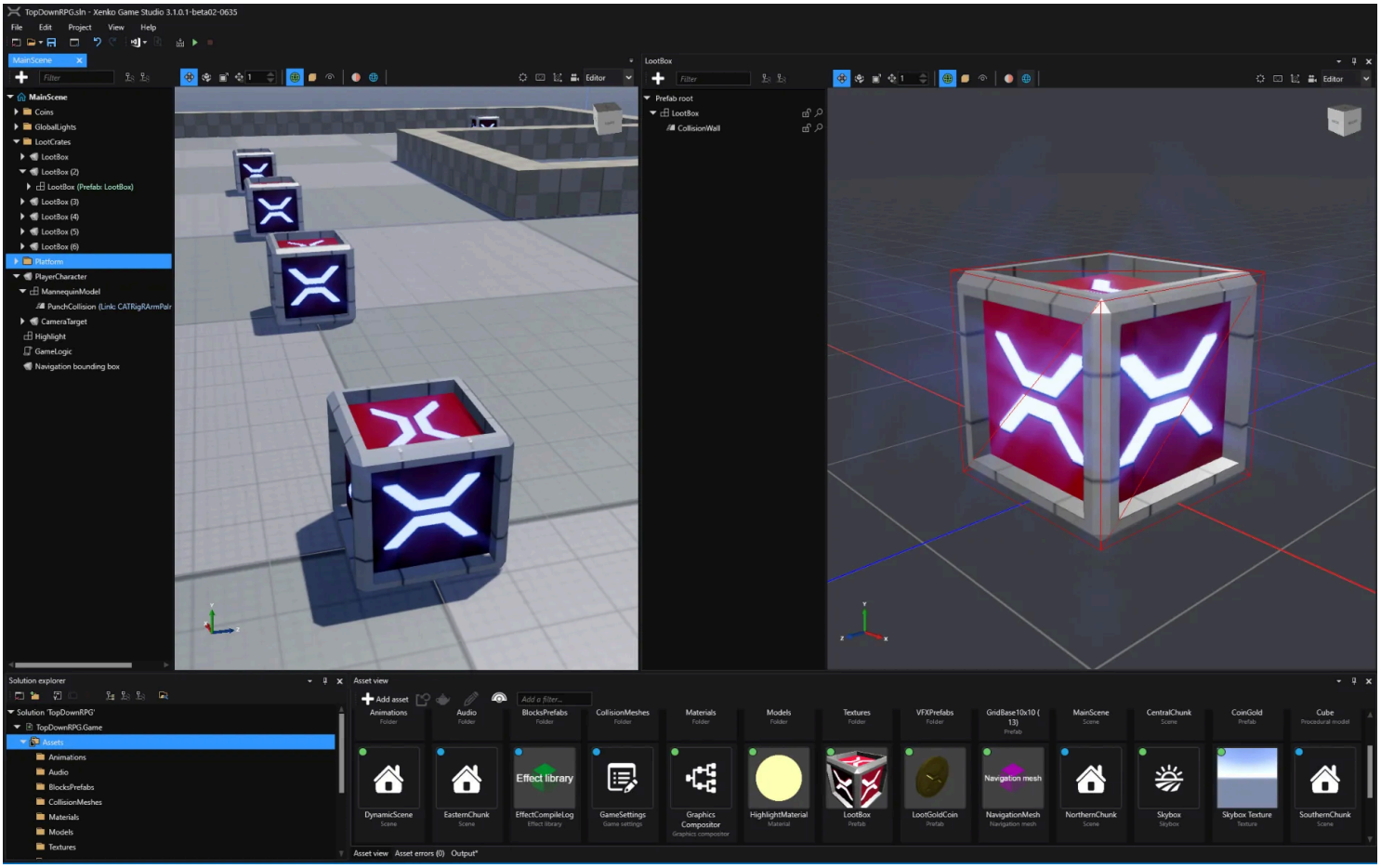
Asset type	Supported formats
Models, animations, skeletons	`.fbx`, `.dae`, `.3ds`, `.obj`, `.blend`, `.x`, `.md2`, `.md3`, `.dxf`
Sprites, textures, skyboxes	`.dds`, `.jpg`, `.jpeg`, `.png`, `.gif`, `.bmp`, `.tga`, `.psd`, `.tif`, `.tiff`

Asset type	Supported formats
Audio	`.wav`, `.mp3`, `.ogg`, `.aac`, `.aiff`, `.flac`, `.m4a`, `.wma`, `.mpc`
Fonts	`.ttf`, `.otf`
Video	`.mp4`

For more information about assets, see Assets.

Prefabs

Like Unity®, Stride uses prefabs. Prefabs are "master" versions of objects that you can reuse wherever you need. When you change a prefab, every instance of the prefab changes too.



Just like with Unity®, in Stride, you can add prefabs to other prefabs. These are called **nested prefabs**. If you modify a nested prefab, all the dependent prefabs inherit the change automatically.

For example, imagine you create a *Vehicle* prefab with acceleration, braking, steering, and so on. Then you nest the *Vehicle* prefab inside prefabs of different types of vehicles: a taxi, bus, truck, etc. If you adjust a property in the *Vehicle* prefab, the changes are inherited by all other prefabs. For example, if you

increase the Acceleration property in the *Vehicle* prefab, the acceleration property in the taxi, bus, and truck prefabs also increase.

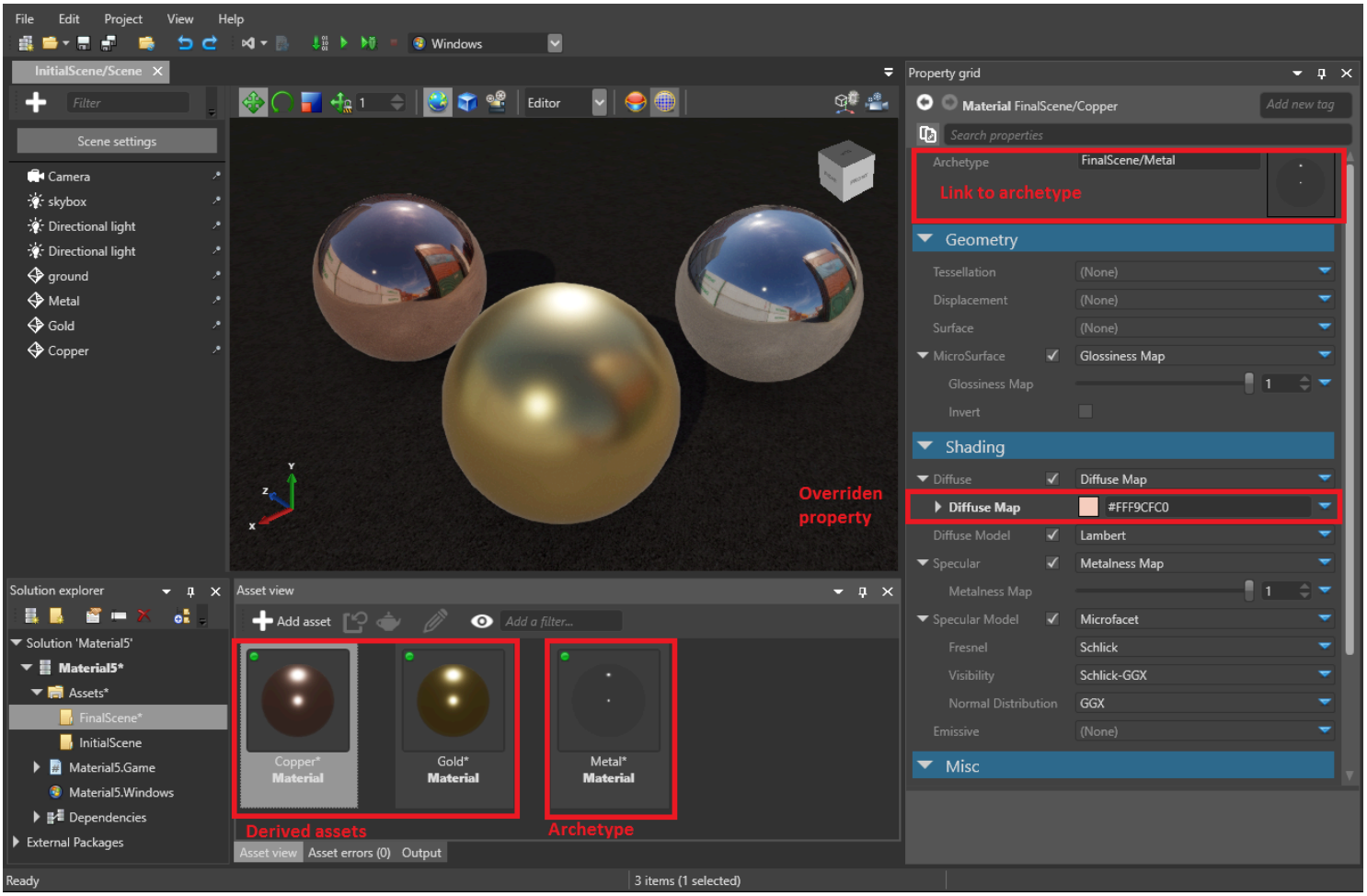
For more information about using prefabs in Stride, see Prefabs.

Archetypes

Archetypes are master assets that control the properties of assets you **derive** from them. Derived assets are useful when you want to create a "remixed" version of an asset. This is similar to prefabs.

For example, imagine we have three sphere entities that share a material asset named *Metal*. Now imagine we want to change the color of only *one* sphere, but keep its other properties the same. We could duplicate the material asset, change its color, and then apply the new asset to only one sphere. But if we later want to change a different property across *all* the spheres, we have to modify both assets. This is time-consuming and leaves room for mistakes.

The better approach is to derive a new asset from the archetype. The derived asset inherits properties from the archetype and lets you override individual properties where you need them. For example, we can derive the sphere's material asset and override its color. Then, if we change the gloss of the archetype, the gloss of all three spheres changes.



You can derive an asset from an archetype, then in turn derive another asset from that derived asset. This way you can create different layers of assets to keep your project organized:

```
Archetype
    Derived asset
        Derived asset
```

For more information about archetypes, see Archetypes.

Object Life Time

Entities and components are not destroyed in Stride, they are removed from the scene they exist in and then freed by the Garbage Collector.

This seemingly small difference significantly changes how objects are managed within the engine. For example, entities can be removed from a scene, kept as a reference in a component, and added back into another scene later on. Components can be removed from an entity and added onto another without losing its internal state.

Input

Stride supports a variety of inputs. The code samples below demonstrate the difference in input code between Stride and Unity®.

For more information about Input in Stride, see Input.

Unity®

```
void Update()
{
    // true for one frame in which the space bar was pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        // Do something.
    }

    // true while this joystick button is down
    if (Input.GetButton("joystick button 0"))
    {
        // Do something.
    }

    float horiz = Input.GetAxis("Horizontal");
    float vert = Input.GetAxis("Vertical");
```

```
    // Do something else.
}
```

Stride

```
public override void Update()
{
    // true for one frame in which the space bar was pressed
    if (Input.IsKeyDown(Keys.Space))
    {
        // Do something.
    }

    // true while this joystick button is down
    if (Input.GameControllers[0].IsButtonDown(0))
    {
        // Do something.
    }

    float horiz = (Input.IsKeyDown(Keys.Left) ? -1f : 0) + (Input.IsKeyDown(Keys.Right) ? 1f
: 0);
    float vert = (Input.IsKeyDown(Keys.Down) ? -1f : 0) + (Input.IsKeyDown(Keys.Up) ? 1f
: 0);
    // Do something else.
}
```

Time

Unity®	Stride
`Time.deltaTime`	`Game.UpdateTime.WarpElapsed.TotalSeconds`
`Time.unscaledDeltaTime`	`Game.UpdateTime.Elapsed.TotalSeconds`
`Time.realtimeSinceStartup`	`Game.UpdateTime.Total.TotalSeconds`
`Time.timeScale`	`Game.UpdateTime.Factor`
`Time.fixedDeltaTime`	`myRigidbodyComponent.Simulation.FixedTimeStep`

Physics

Just like Unity®, Stride has three types of colliders:

- static colliders
- rigidbodies
- characters

They're controlled by scripts in slightly different ways.

Kinematic rigidbodies

Unity®

```
public class KinematicX : MonoBehaviour
{
    public Rigidbody rigidBody;

    void Start()
    {
        // Initialization of the component.
        rigidBody = GetComponent<Rigidbody>();
    }

    void EnableRagdoll()
    {
        rigidBody.isKinematic = false;
        rigidBody.detectCollisions = true;
    }

    void DisableRagdoll()
    {
        rigidBody.isKinematic = true;
        rigidBody.detectCollisions = false;
    }
}
```

Stride

```
public class KinematicX : SyncScript
{
    public RigidbodyComponent rigidBody;

    public override void Start()
    {
        // Initialization of the component.
        rigidBody = Entity.Get<RigidbodyComponent>();
    }
```

```
    public override void Update()
    {
        // Perform an update every frame.
    }

    void EnableRagdoll()
    {
        rigidBody.IsKinematic = false;
        rigidBody.ProcessCollisions = true;
    }

    void DisableRagdoll()
    {
        rigidBody.IsKinematic = true;
        rigidBody.ProcessCollisions = false;
    }
}
```

For more information about rigidbodies in Stride, see Rigidbodies.

Triggers

Unity®

```
// Occurs when game objects go through this trigger.
void OnTriggerEnter(Collider Other)
{
    Other.transform.localScale = new Vector3(2.0f, 2.0f, 2.0f);
}

// Occurs when game objects move out of this trigger.
void OnTriggerExit(Collider Other)
{
    Other.transform.localScale = new Vector3(1.0f, 1.0f, 1.0f);
}
```

Stride

```
var trigger = Entity.Get<PhysicsComponent>();
trigger.ProcessCollisions = true;

// Start state machine.
while (Game.IsRunning)
{
    // 1. Wait for an entity to collide with the trigger.
```

```
    Collision firstCollision = await trigger.NewCollision();

    PhysicsComponent otherCollider = trigger == firstCollision.ColliderA
        ? firstCollision.ColliderB
        : firstCollision.ColliderA;
    otherCollider.Entity.Transform.Scale = new Vector3(2.0f, 2.0f, 2.0f);

    // 2. Wait for the entity to exit the trigger.
    Collision collision;

    do
    {
        collision = await trigger.CollisionEnded();
    }
    while (collision != firstCollision);

    otherCollider.Entity.Transform.Scale = new Vector3(1.0f, 1.0f, 1.0f);
}
```

For more information about triggers in Stride, see Triggers

Raycasting

Unity®

```
public static Collider FindGOCameraIsLookingAt()
{
    int distance = 50;

    // Cast a ray and set it to the mouse cursor position in the game
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit, distance))
    {
        // Draw invisible ray cast/vector
        Debug.DrawLine(ray.origin, hit.point);
        // Log hit area to the console
        Debug.Log(hit.point);
        return hit.collider;
    }
    return null;
}
```

Stride

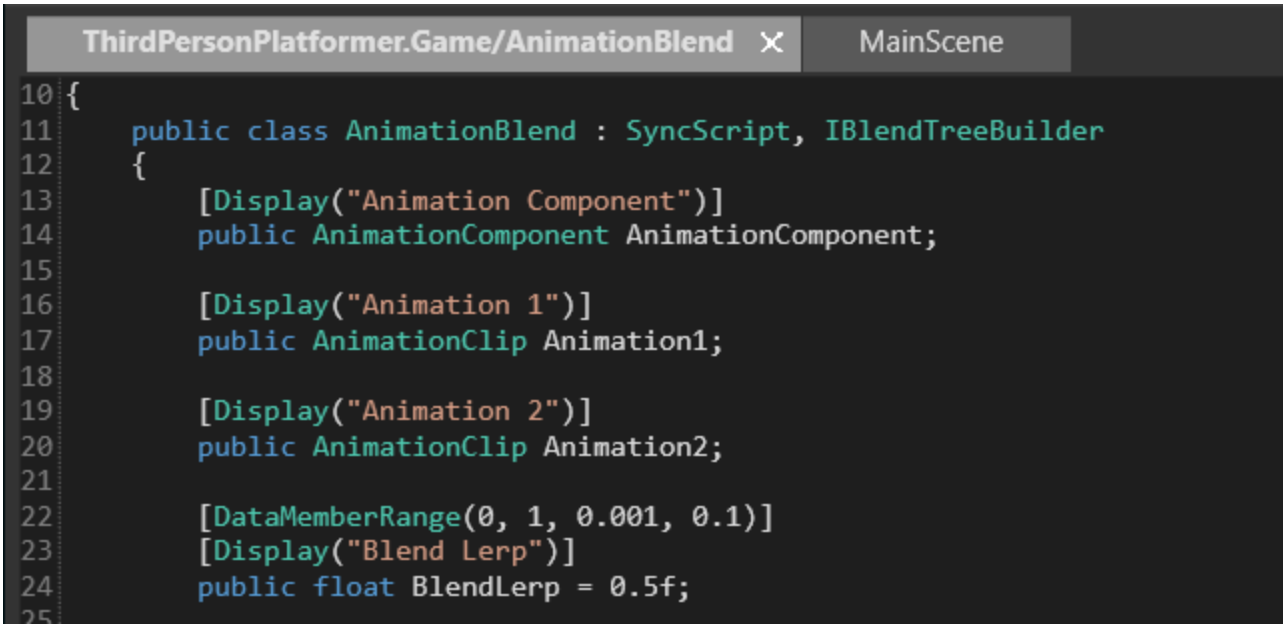
```
public static bool ScreenPositionToWorldPositionRaycast(Vector2 screenPos, CameraComponent
camera, Simulation simulation)
{
    Matrix invViewProj = Matrix.Invert(camera.ViewProjectionMatrix);

    Vector3 sPos;
    sPos.X = screenPos.X * 2f - 1f;
    sPos.Y = 1f - screenPos.Y * 2f;

    sPos.Z = 0f;
    Vector4 vectorNear = Vector3.Transform(sPos, invViewProj);
    vectorNear /= vectorNear.W;

    sPos.Z = 1f;
    Vector4 vectorFar = Vector3.Transform(sPos, invViewProj);
    vectorFar /= vectorFar.W;

    HitResult result = simulation.Raycast(vectorNear.XYZ(), vectorFar.XYZ());
    return result.Succeeded;
}
```

For more information about Raycasting in Stride, see Raycasting.

Scripts

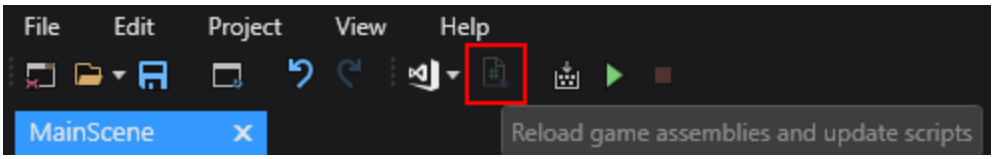
Stride saves scripts in a subfolder in the **MyGame.Game** folder in the project directory.

To open a script in the Game Studio script editor, double-click it in the **Asset View**. The script editor has syntax highlighting, auto-completion, and live diagnostics.

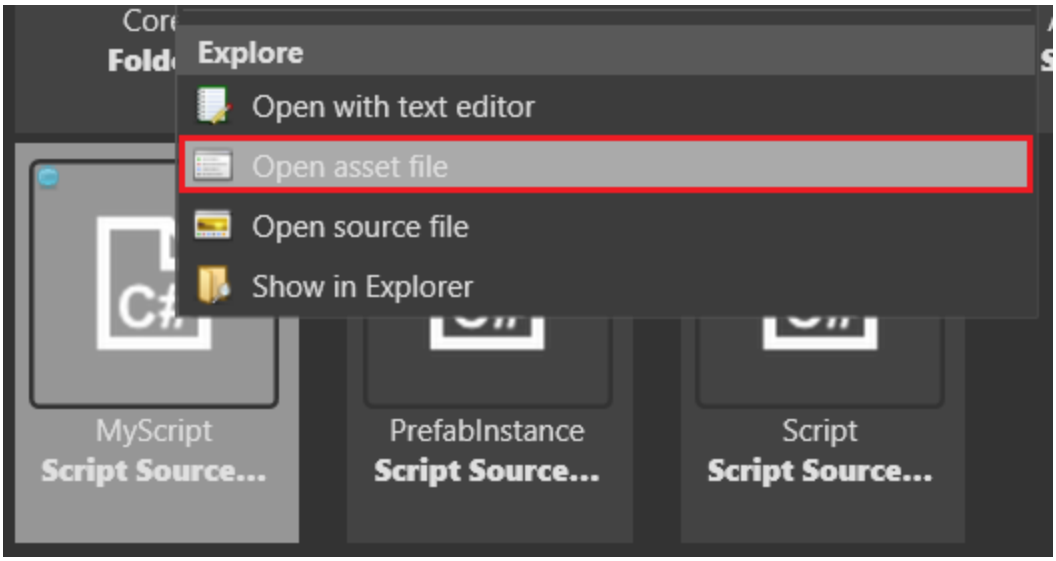


You can also edit scripts in other IDEs, such as Visual Studio. When you edit a script in an external IDE, Stride reloads it automatically.

If you install the Visual Studio plug-in during the Stride installation, you can open your project in Visual Studio from Game Studio. To do this, in the Game Studio toolbar, click **Open in IDE**.



Alternatively, right-click the script in the **Asset View** and click **Open asset file**:



Event functions (Start, Update, Execute, etc)

In Unity®, you work with MonoBehaviours with Start(), Update(), and other methods.

Instead of MonoBehaviours, Stride has three types of scripts: SyncScript, AsyncScript, and StartupScript. For more information, see Types of script.

Unity® MonoBehaviour

```
public class BasicMethods : MonoBehaviour
{
    void Start() { }
    void OnDestroy() { }
    void Update() { }
}
```

Stride SyncScript

```
public class BasicMethods : SyncScript
{
    public override void Start() { }
    public override void Cancel() { }
    public override void Update() { }
}
```

Stride AsyncScript

```
public class BasicMethods : AsyncScript
{
    // Declared public member fields and properties that will appear in the game studio
    public override async Task Execute()
    {
        while (Game.IsRunning)
        {
            // Do stuff every new frame
            await Script.NextFrame();
        }
    }

    public override void Cancel()
    {
        // Cleanup of the script
    }
}
```

Stride StartupScript

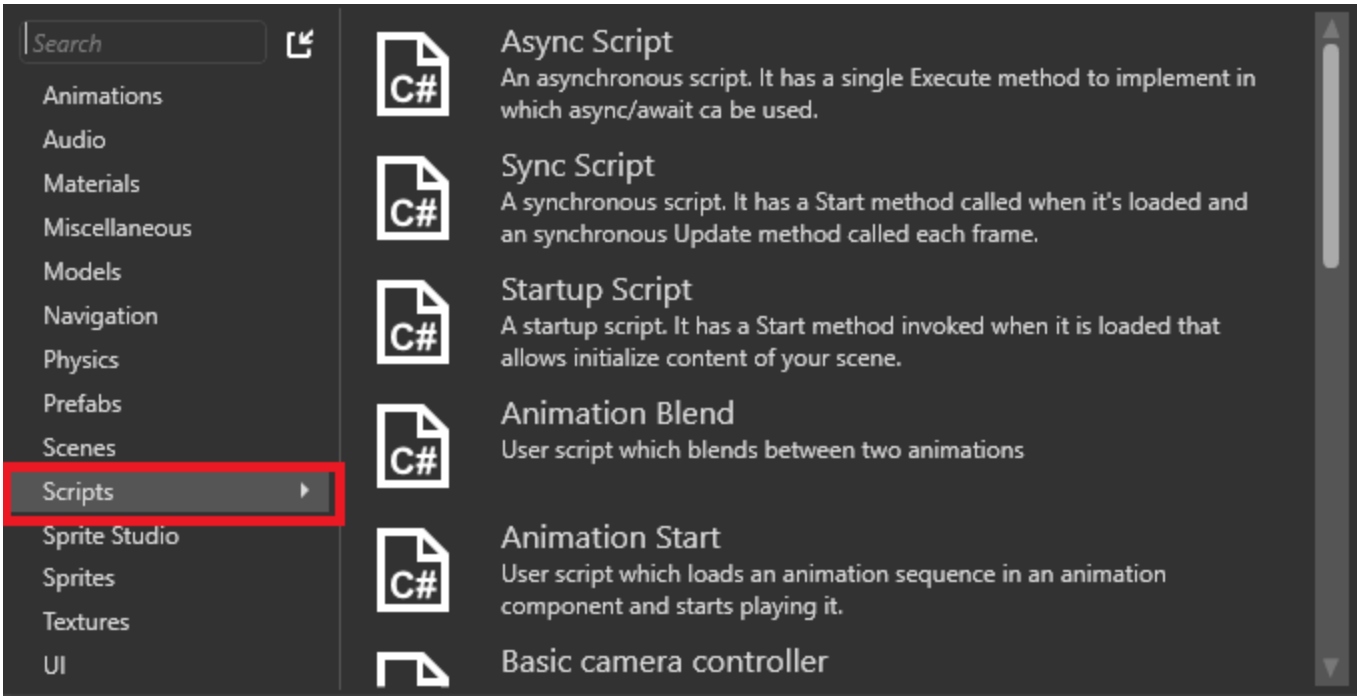
```
public class BasicMethods : StartupScript
{
    // Declared public member fields and properties that will appear in the game studio
    public override void Start()
    {
        // Initialization of the script
    }

    public override void Cancel()
    {
        // Cleanup of the script
    }
}
```

Script components

Like Unity®, in Stride, you attach scripts to entities by adding them as script components.

Create a script

To create a script, click the **Add asset** button and select **Scripts**.



In Unity®, when you create a `MonoBehaviour` script, it has two base functions: `MonoBehaviour.Start()` and `MonoBehaviour.Update()`. Stride has a `SyncScript` that works similarly. Like `MonoBehaviour`, `SyncScript` has two methods:

- `SyncScript.Start()` is called when it the script is loaded.
- `SyncScript.Update()` is called every update.

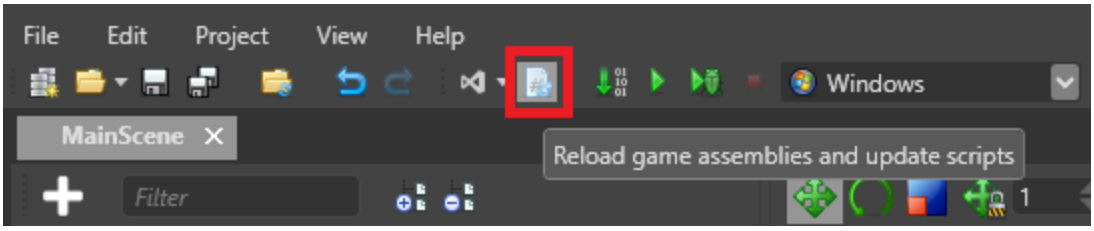
Unlike `MonoBehaviour`, implementating the `SyncScript.Update()` method is not optional, and as such, must be implemented in every `SyncScript`.

If you want your script to be a startup or asynchronous, use the corresponding script types:

- `StartupScript`: this script has a single `StartupScript.Start()` method. It initializes the scene and its content at startup.
- `AsyncScript`: an asynchronous script with a single method `AsyncScript.Execute()` and you can use async/await inside that method. Asynchronous scripts aren't loaded one by one like synchronous scripts. Instead, they're all loaded in parallel.

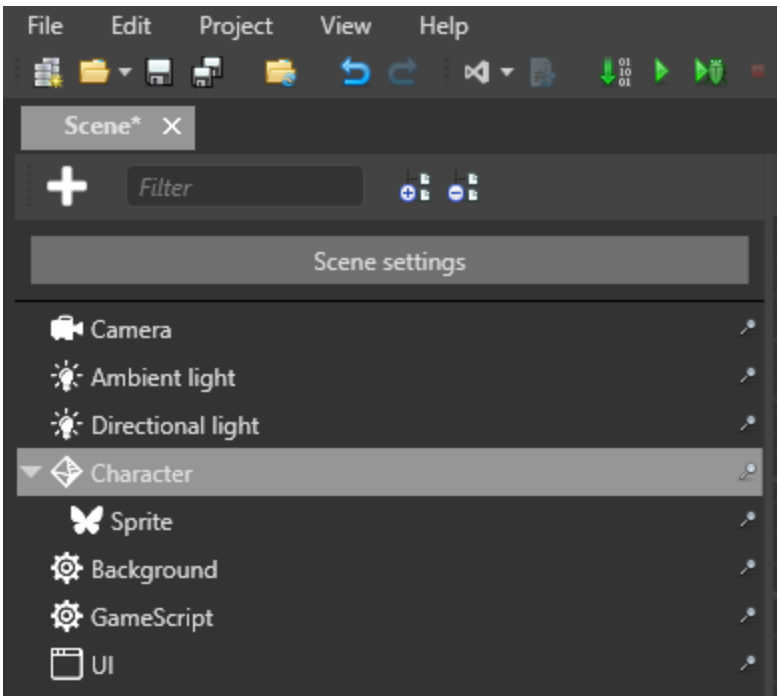
Reload assemblies

After you create a script, you may have to reload the assemblies manually. To do this, click **Reload assemblies** in the Game Studio toolbar.

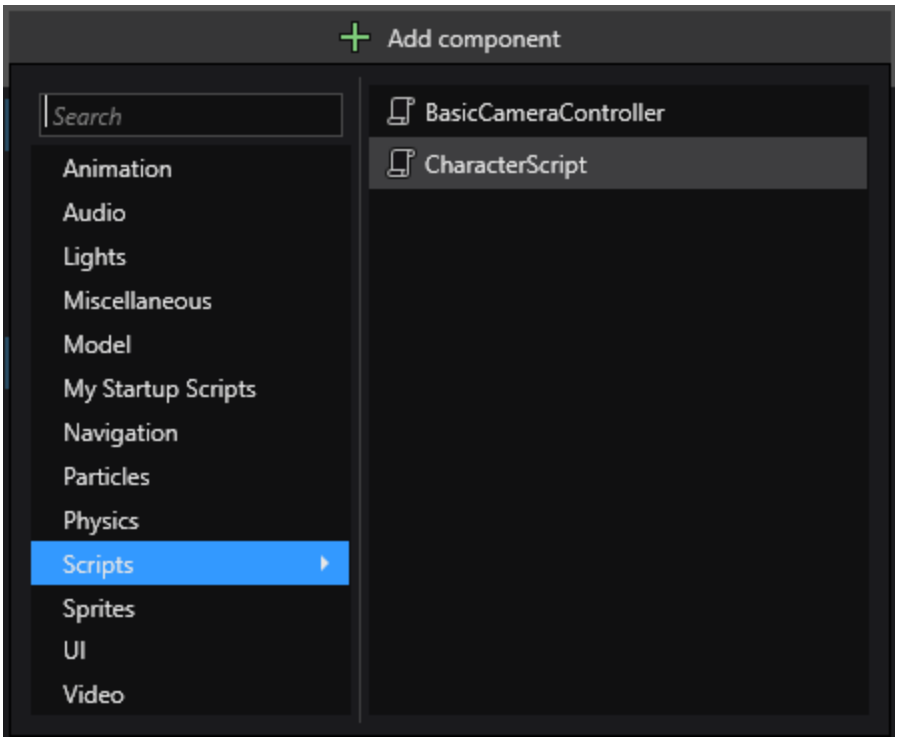


Add scripts to entities

1. In the **Entity Tree** (on the left by default), or in the scene, select the entity you want to add the script to.



2. In the **Property Grid** (on the right by default), click **Add component** and select the script you want to add.



In Unity®, script components are grouped under **Components > Scripts**. In Stride, scripts are not grouped. Instead, Game Studio lists them alphabetically with other components.

For more information about adding scripts in Stride, see Use a script.

Scripting gameplay

Unity® and Stride both use C#. However, scripting gameplay in Stride is a little different from Unity®.

Instantiate Entity / GameObject

In Unity®, you use `Instantiate` to create new object instances. This function makes a copy of `UnityEngine.Object` and spawns it to the scene.

Unity®

```
public GameObject CarPrefab;
public Vector3 SpawnPosition;
public Quaternion SpawnRotation;

void Start()
{
    GameObject newGameObject = (GameObject)Instantiate(CarPrefab,
SpawnPosition, SpawnRotation);
    newGameObject.name = "NewGameObject1";
}
```

Stride

In Stride, you can instantiate **Entities** similarly to Unity® GameObjects:

```
// Declared public member fields and properties displayed in the Game Studio Property Grid.
public Prefab CarPrefab;
public Vector3 SpawnPosition;
public Quaternion SpawnRotation;

public override void Start()
{
    // Initialization of the script.
    List<Entity> car = CarPrefab.Instantiate();
    SceneSystem.SceneInstance.RootScene.Entities.AddRange(car);
    car[0].Transform.Position = SpawnPosition;
    car[0].Transform.Rotation = SpawnRotation;
    car[0].Name = "MyNewEntity";
}
```

Use default values

Each class in Unity® has certain default values. If you don't override these properties in the script, the default values will be used. This works the same in Stride:

Unity®

```
public int NewProp = 30;
public Light MyLightComponent = null;

void Start()
{
    // Create the light component if we don't already have one.
    if (MyLightComponent == null)
    {
        MyLightComponent = gameObject.AddComponent<Light>();
        MyLightComponent.intensity = 3;
    }
}
```

Stride

```
// Declared public member fields and properties displayed in the Game Studio Property Grid.
public int NewProp = 30;
public LightComponent MyLightComponent = null;
```

```
public override void Start()
{
    // Create the light component if we don't already have one.
    if (MyLightComponent == null)
    {
        MyLightComponent = new LightComponent();
        MyLightComponent.Intensity = 3;
        Entity.Add(MyLightComponent);
    }
}
```

Disable GameObject/entity

Unity®

```
MyGameObject.SetActive(false);
```

Stride

```
Entity.EnableAll(false, true);
```

Access component from GameObject/entity

Unity®

```
Light lightComponent = GetComponent<Light>();
```

Stride

```
LightComponent lightComponent = Entity.Get<LightComponent>();
```

Access GameObject/entity from component

Unity®

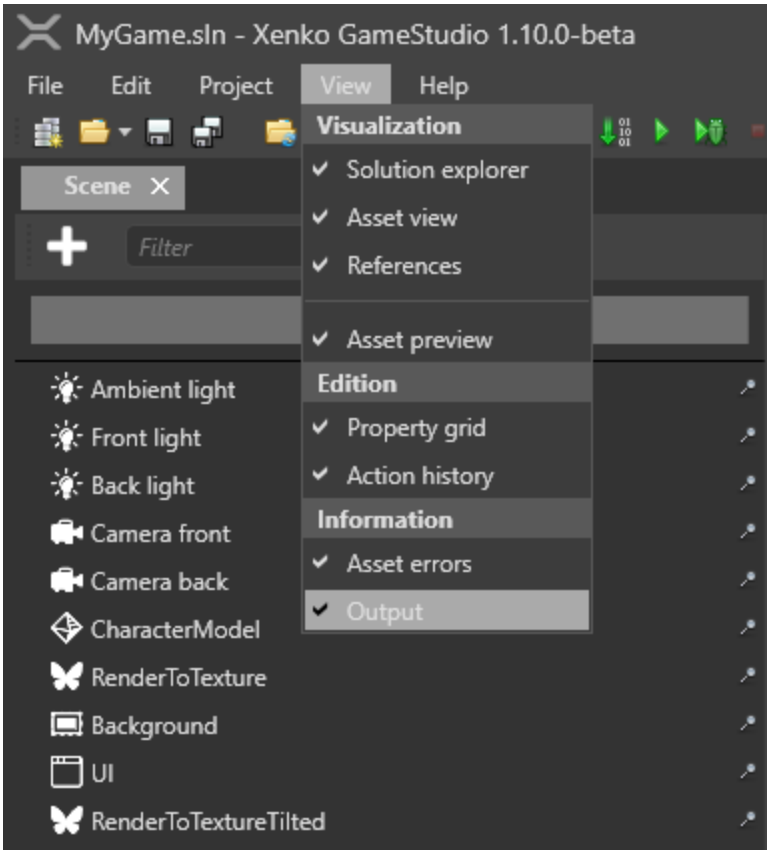
```
GameObject componentGameObject = lightComponent.gameObject;
```

Stride

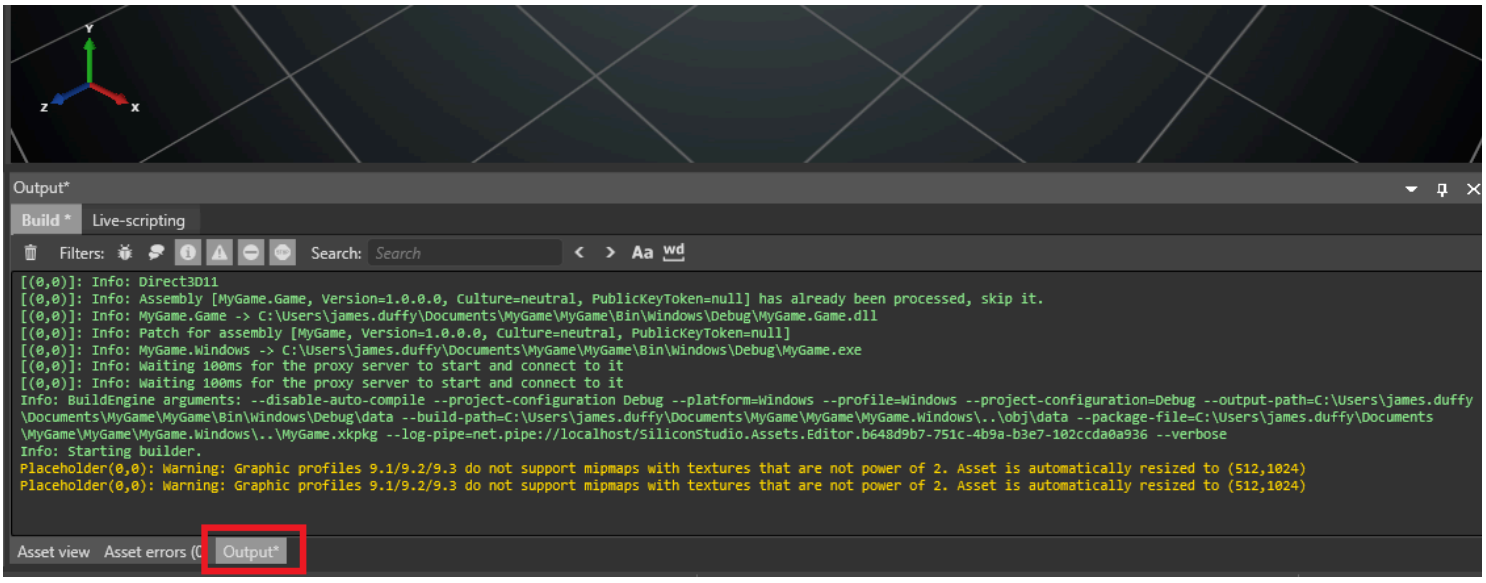
```
Entity componentEntity = lightComponent.Entity;
```

Log output

To see the output, in the Game Studio toolbar, under **View**, enable **Output**.



Game Studio displays in the **Output** tab (at the bottom of Game Studio by default).



Print debug messages

Logging from a ScriptComponent:

```
public override void Start()
{
    // Enables logging. It will also spawn a console window if no debuggers are attached.
    // The argument dictates the kinds of message that will be filtered out, in this case,
anything with less priority than warning won't show up
    Log.ActivateLog(LogMessageType.Warning);
    // Log this message to your console or IDE output window
    Log.Warning("hello");
}


System.Diagnostics.Debug.WriteLine("hello");
```

> **NOTE**
>
> To print debug messages, you have to run the game from your IDE, not Game Studio. Running games cannot print to the Game Studio output window.

Attributes

Unity®	Stride
`[Serializable]`	`[DataContract]`
`[SerializeField]`	`[DataMember]`
`[HideInInspector]`	`[DataMemberIgnore]`
`[Range]`	`[DataMemberRange]`
`[Header("My Header")]`	`[Display(category: "My Header")]`
`[Tooltip("My tooltip")]`	`/// <userdoc>My tooltip</userdoc>`

> **NOTE**
>
> You cannot serialize `private` fields in Stride, if you want to set a field in editor but prevent other scripts from writing to that field, you should use a init property

```
public float MyProperty { get; init; }
```

Unity® is a trademark of Unity Technologies.

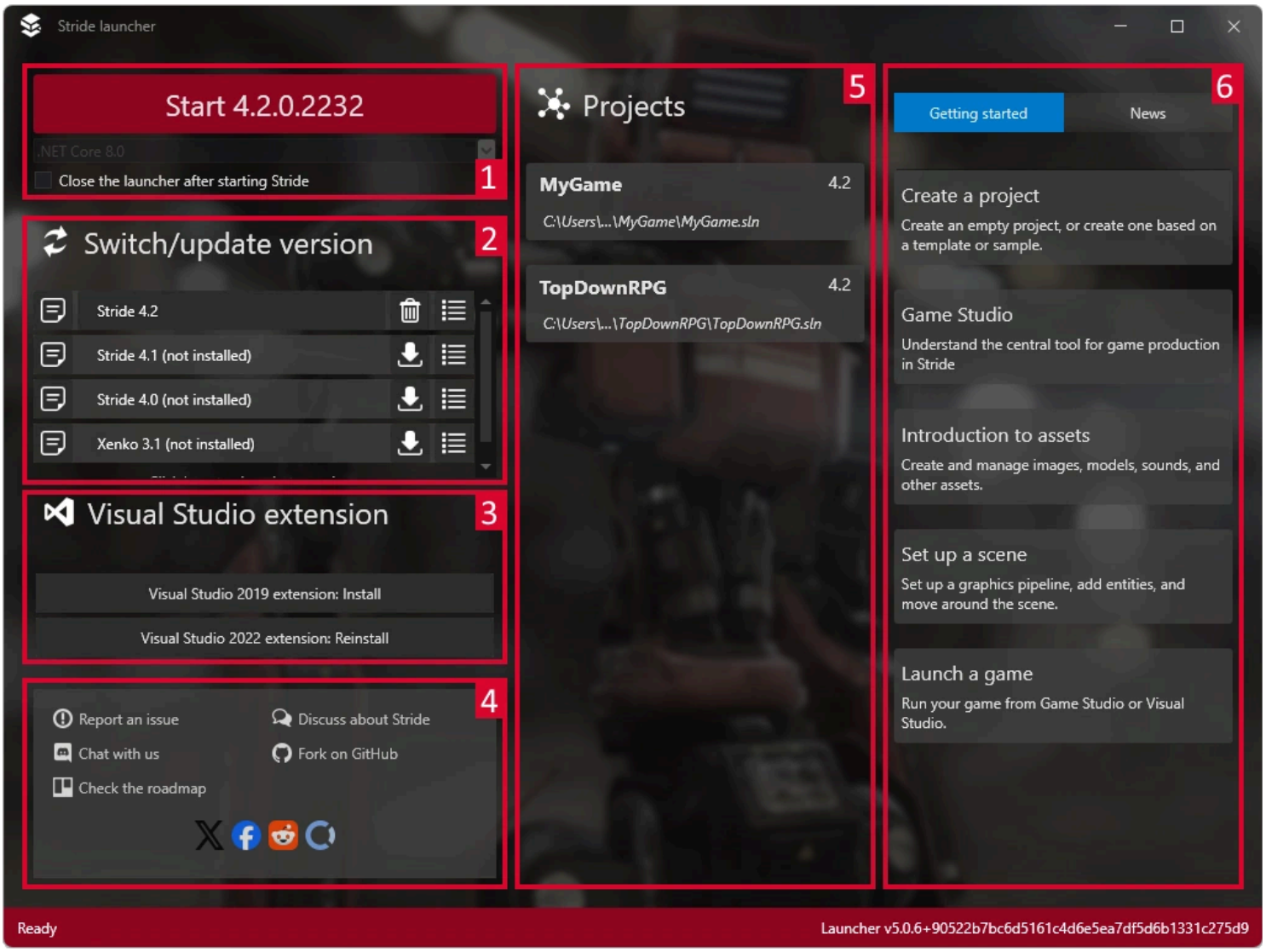
See also

- Best Practice

Stride Launcher

Beginner

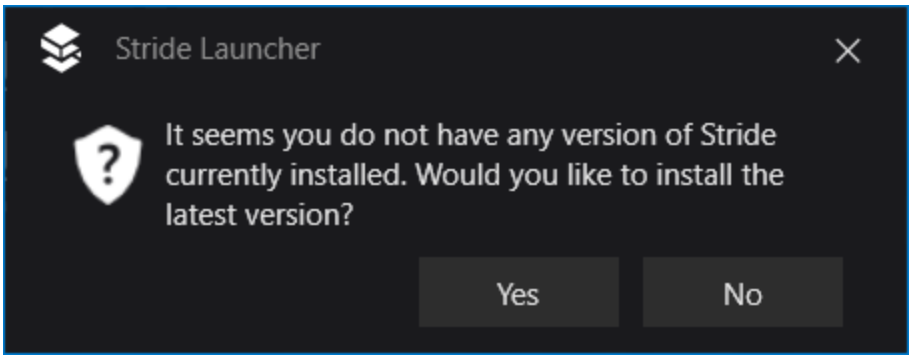
With the **Stride launcher**, you can install, manage and run different versions of Stride.



1 Start Stride Game Studio
2 Stride Different Versions
3 Visual Studio Plugins
4 Interact with Community
5 Open Recent Projects
6 Documentation and News

Install the latest Stride version

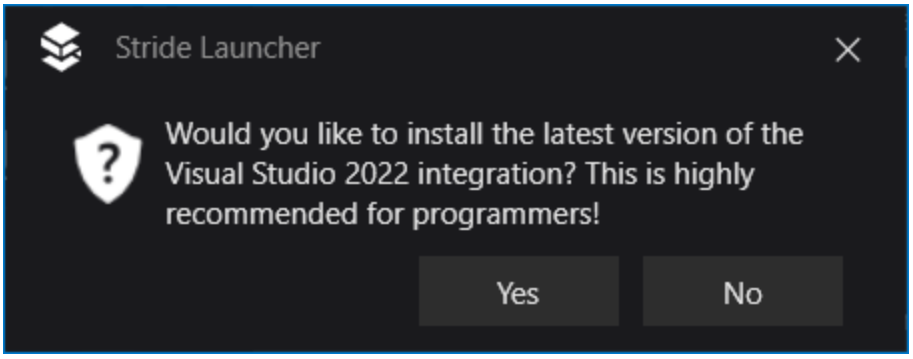
If you don't have Stride installed, the Stride Launcher prompts you to install the latest version.



You can install other versions of Stride in the **Switch/update version** section (**2**). To do this, click the **install** icon next to the version in the list.

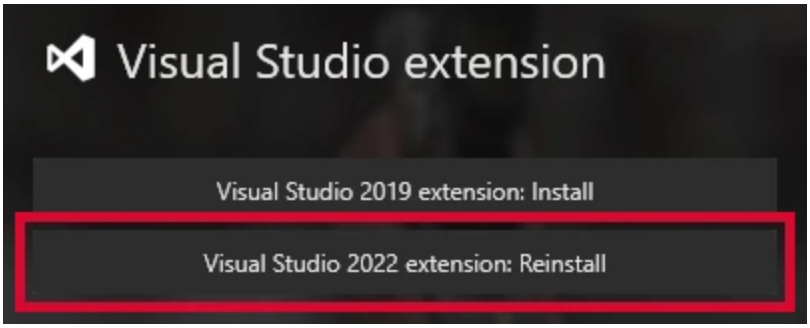
Install the Stride Visual Studio extension

If you choose to install the latest version of Stride, the Stride Launcher asks if you want to install the Visual Studio extension.



The Visual Studio extension lets you edit shaders directly from Visual Studio, and provides syntax highlighting, live code analysis with validation, error checking, and navigation (jump to definition). Installing the extension isn't mandatory, but we recommend it.

To install or reinstall the Visual Studio extension at any time, click the **Reinstall** button in the Stride Launcher.



Switch the Stride version

To choose which version of Stride runs, select it in the list under **Switch/update version**.

Manage different versions of Stride

You can install and uninstall multiple versions of Stride from the **Switch/update version** section.

1 Release Notes
2 Version
3 Uninstall
4 Download and Install
5 Install Minor Versions

You might need to use an older version of Stride to work with old projects. Newer versions of Stride might contain changes that require old projects to be upgraded.

The version number consists of two numbers. The first number refers to the **major** version, and the second number refers to the **minor** version.

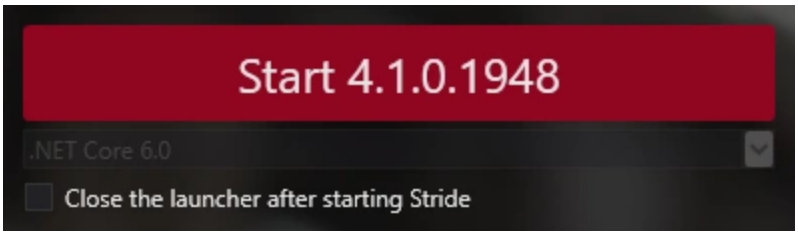
Major updates add significant changes, and you might need to update your projects to use them. Minor updates don't contain breaking changes, so they're safe use with your existing projects.

- To see the release notes for a particular version, click the **note icon** next to the version name (**1**).
- To install a particular version, click the **Download and install** icon next to the version name (**4**).
- To install a particular minor version, click the **List** icon (**5**).

Start Game Studio

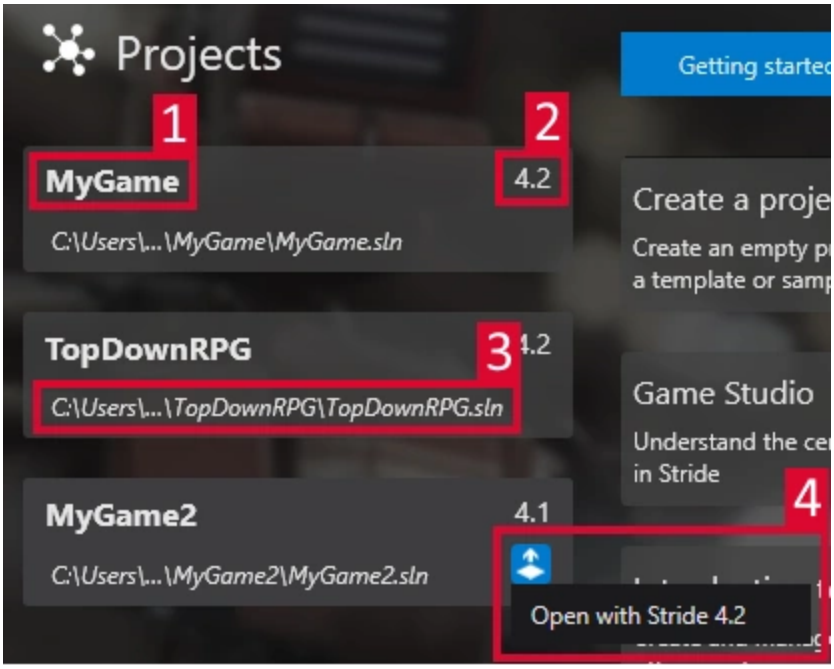
1. Under **Switch/update version**, select the version of Stride you want to use.

   The version number is updated on the **Start** button.



2. Click **Start** to launch Game Studio.

Recent projects



1 Project Name
2 Project Stride Version
3 Project Path
4 Update Project and Open with more Current Version of Stride

The **Projects** section displays your recent projects. To open a project, click it.

Open a project with a newer version of Stride

The top right of each project button (**2**) shows which version of Stride the project was made with.

To open a project with a more recent version of Stride:

1. On the project button, click the **upgrade** icon in the bottom right (**4**).

2. Select the Stride version you want to open the project with. Game Studio prompts you to upgrade the project when it opens.

NOTE

After you update a project to use a newer version of Stride, you might need to make manual changes to get it to work. **Make sure you back up the project and all its related files before you upgrade it.**

Get started with Stride

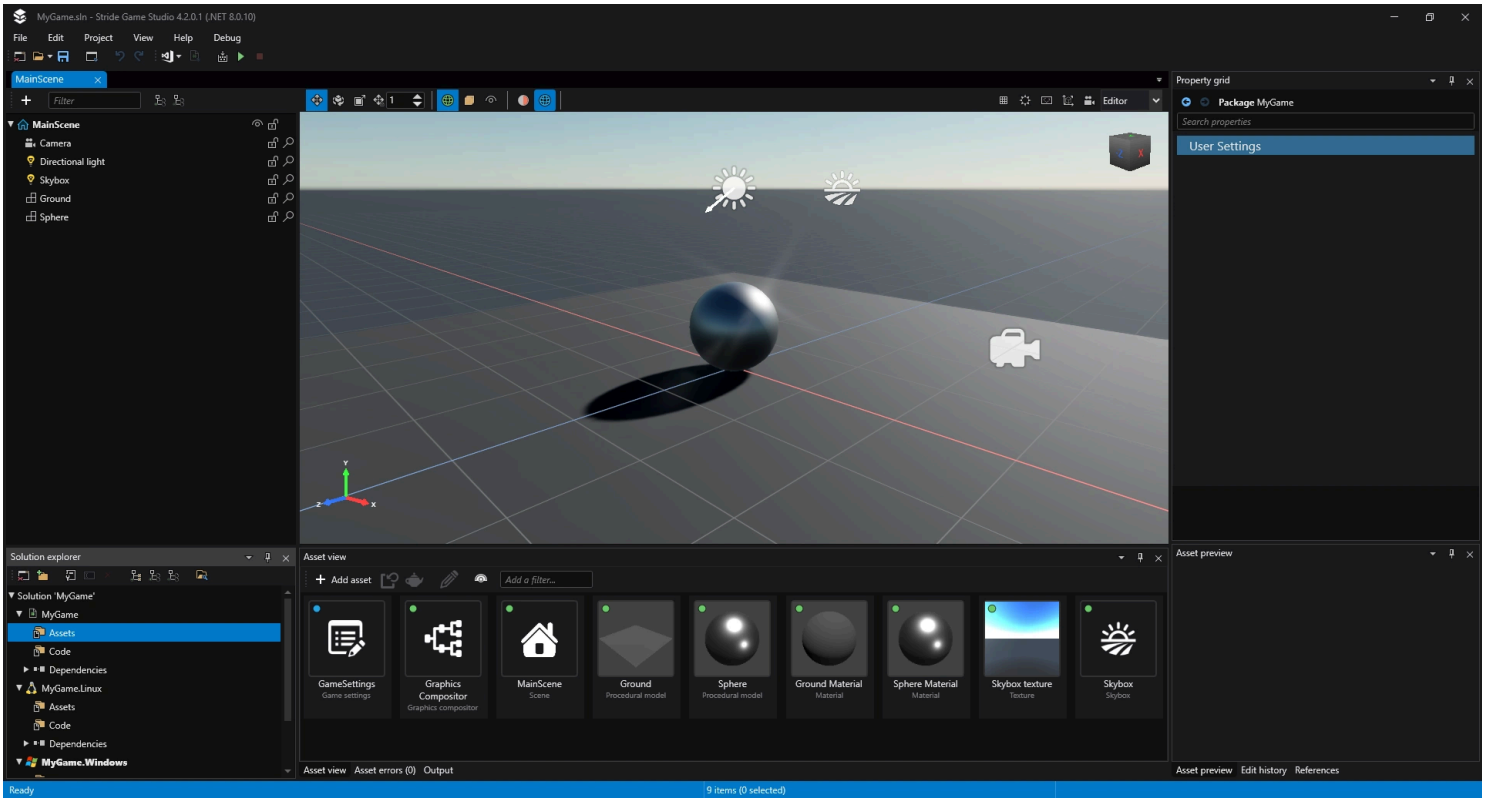
Beginner

Welcome to Stride! This chapter provides everything you need to start creating games using the Stride game engine. If you're new to Stride, we recommend starting with the Install Stride guide, which will help you set up the engine and get you ready for development.

Stride is designed for game developers who want a powerful, flexible, and open-source engine for their projects. Whether you're an experienced developer or just starting out, these guides will walk you through the basics and help you get up and running quickly.

For video tutorials, have a look at the Tutorials.

If you're interested in building the Stride engine from source or contributing to its development, please refer to the instructions on our GitHub repository.



In this section

- Install Stride
- Launch Stride
- Visual Studio extension
- Create a project
- Game Studio
- Assets

- Introduction to scenes
- Launch your game

Install Stride

Beginner

Introduction

If you want to **create games using Stride**, this guide provides the installation steps you'll need to follow. You'll need to install the Stride installer and launcher. The Stride installer is approximately **55 MB** and is downloaded directly from our main GitHub repository.

The installer will automatically download and install the prerequisites if they are not detected, including the Stride Launcher, which is essential for downloading and installing the latest version of Stride for game development.

If you're interested in **building the Stride engine from source** or **contributing to its development**, please visit the Stride GitHub repository for instructions on how to build from source and contribute to the project.

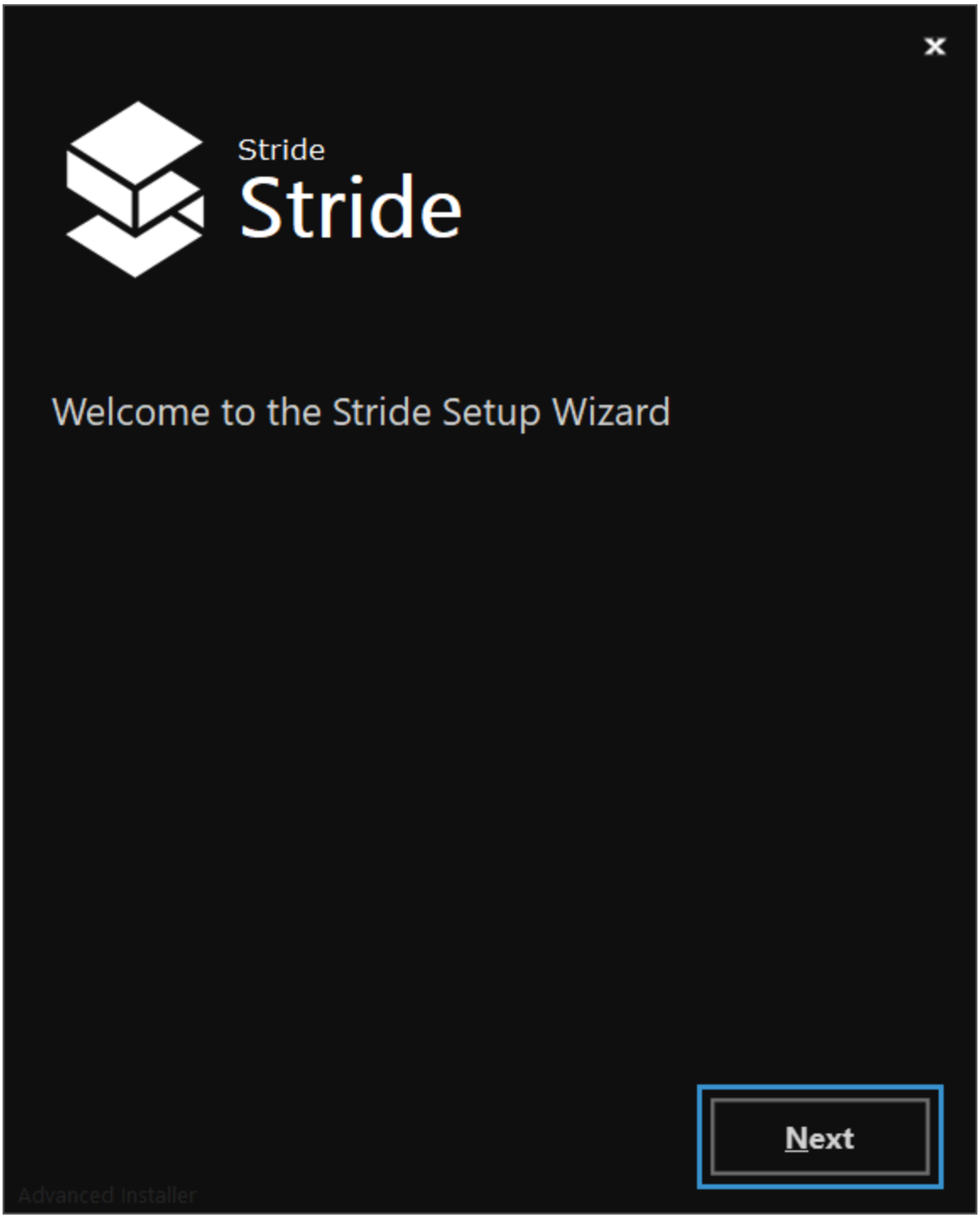
Prerequisites (automatically installed if not present):

- Latest **.NET SDK** supported by Stride
- Microsoft Visual C++ Redistributable

The **Stride Launcher** will download and install the latest version of Stride.

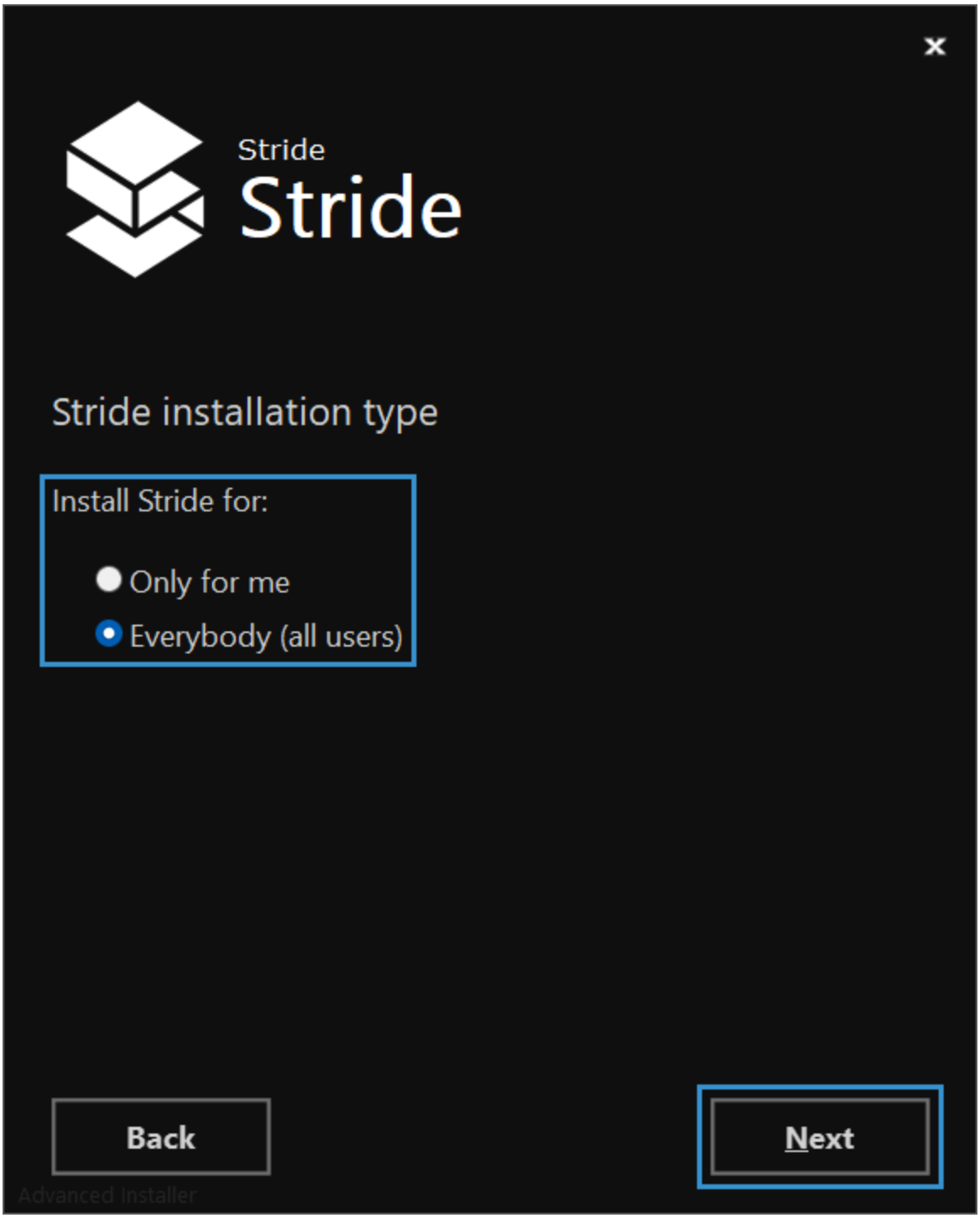
Installation Steps

1. Download the Stride installer (**StrideSetup.exe**) from the Stride website.
2. Run the installer by double-clicking the **StrideSetup.exe** file.
3. The **Stride Setup Wizard** opens.



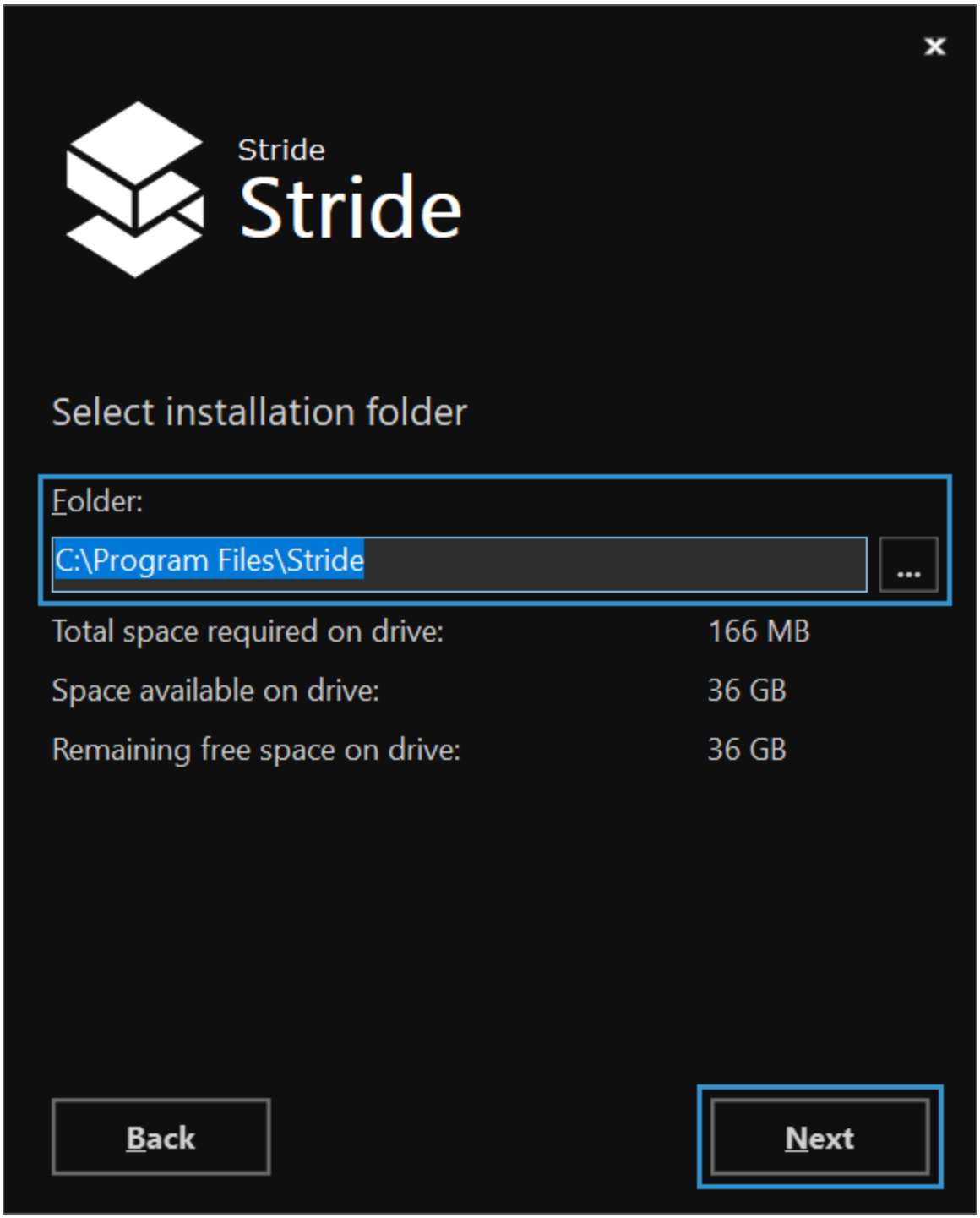
Click **Next**.

4. The **Stride installation type** window opens.



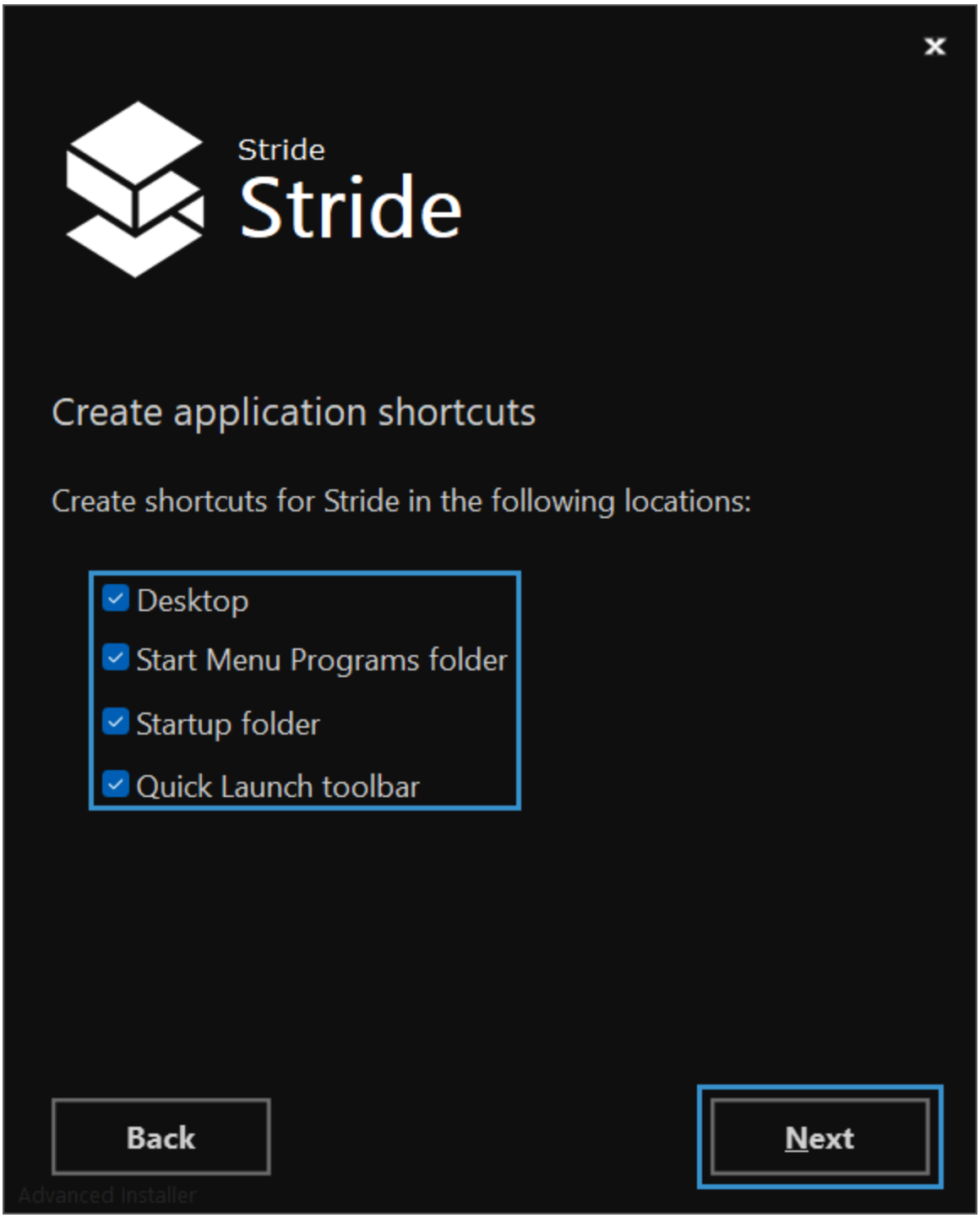
Select an installation type and click **Next**.

5. The **Select installation folder** window opens.



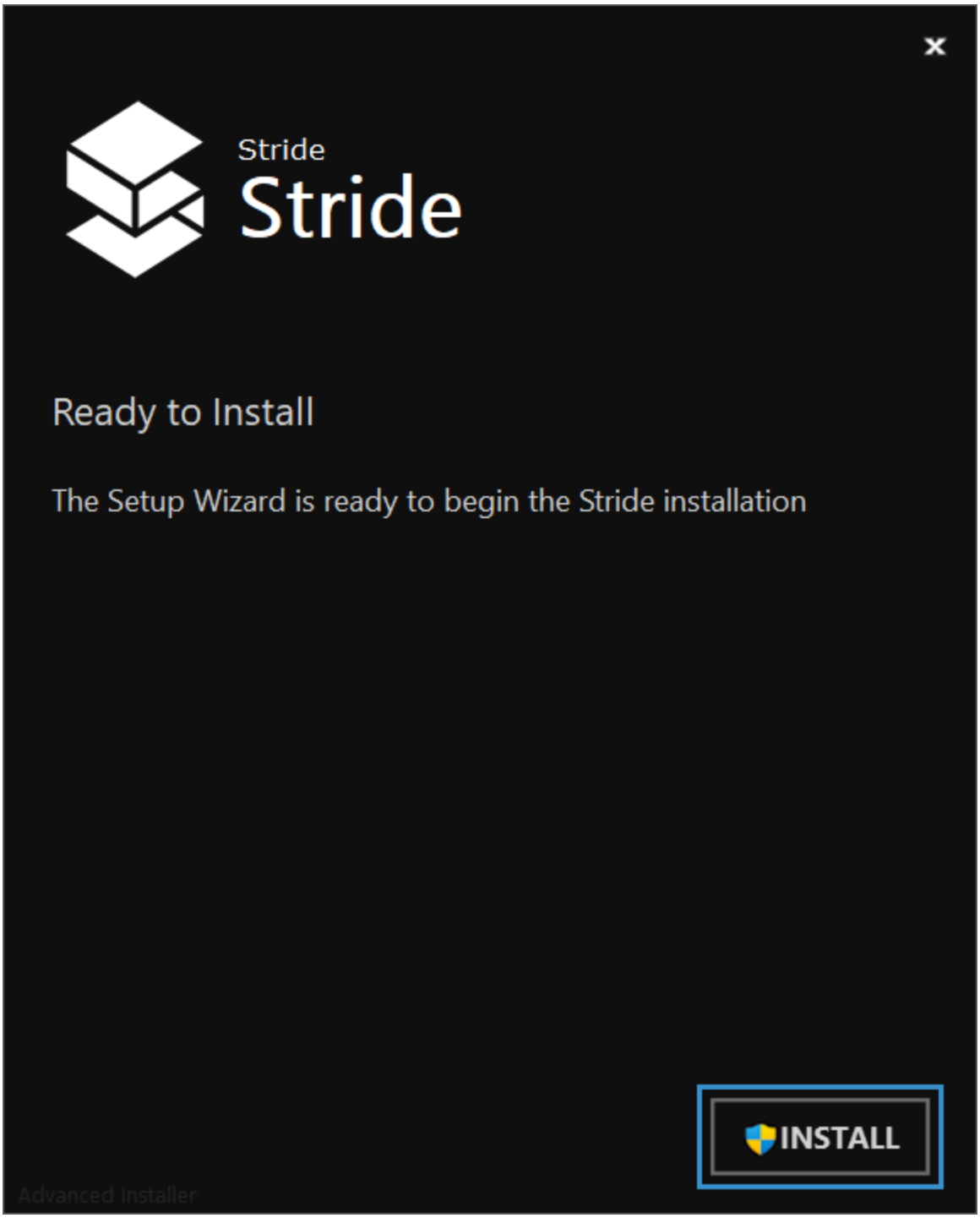
Choose a folder where you want to install Stride, then click **Next**.

6. The **Create application shortcuts** window opens.



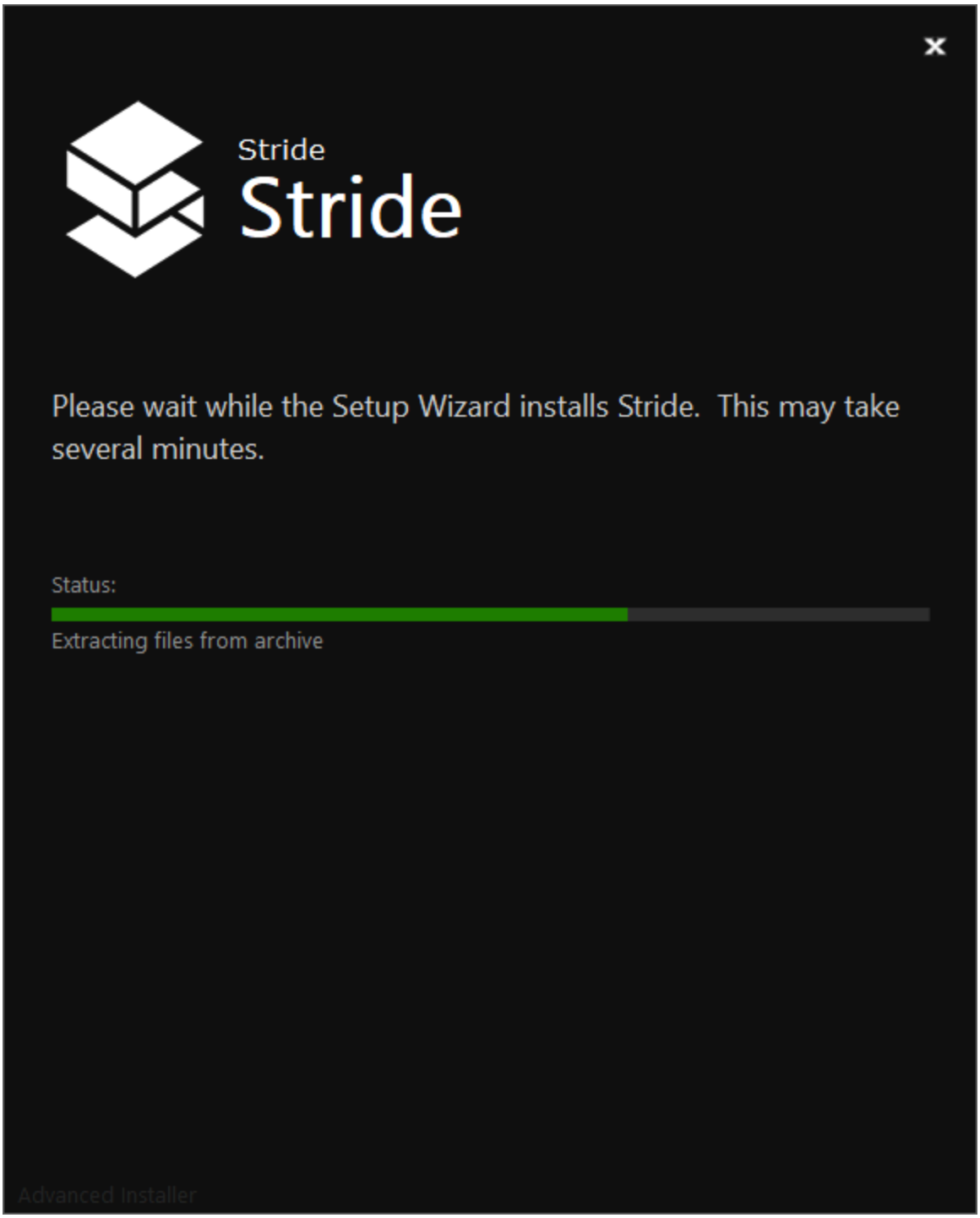
Choose which shortcuts you want Stride to create, then click **Next**.

7. The **Ready to Install** window opens.



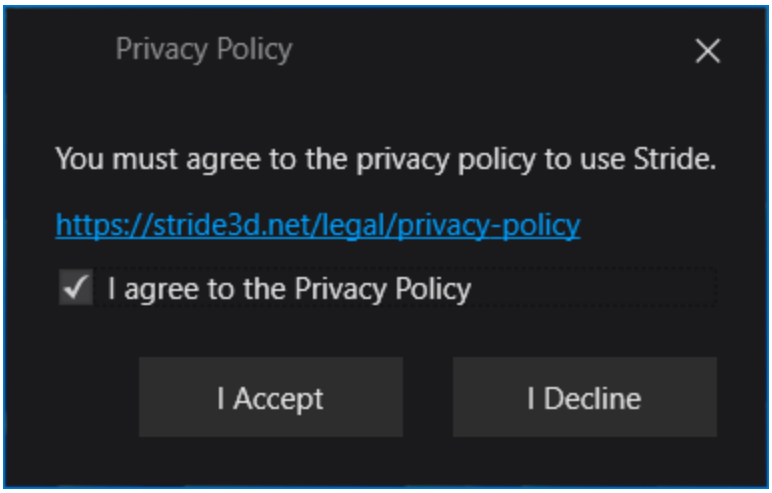
Click **Install** to begin the installation.

8. Installation begins.



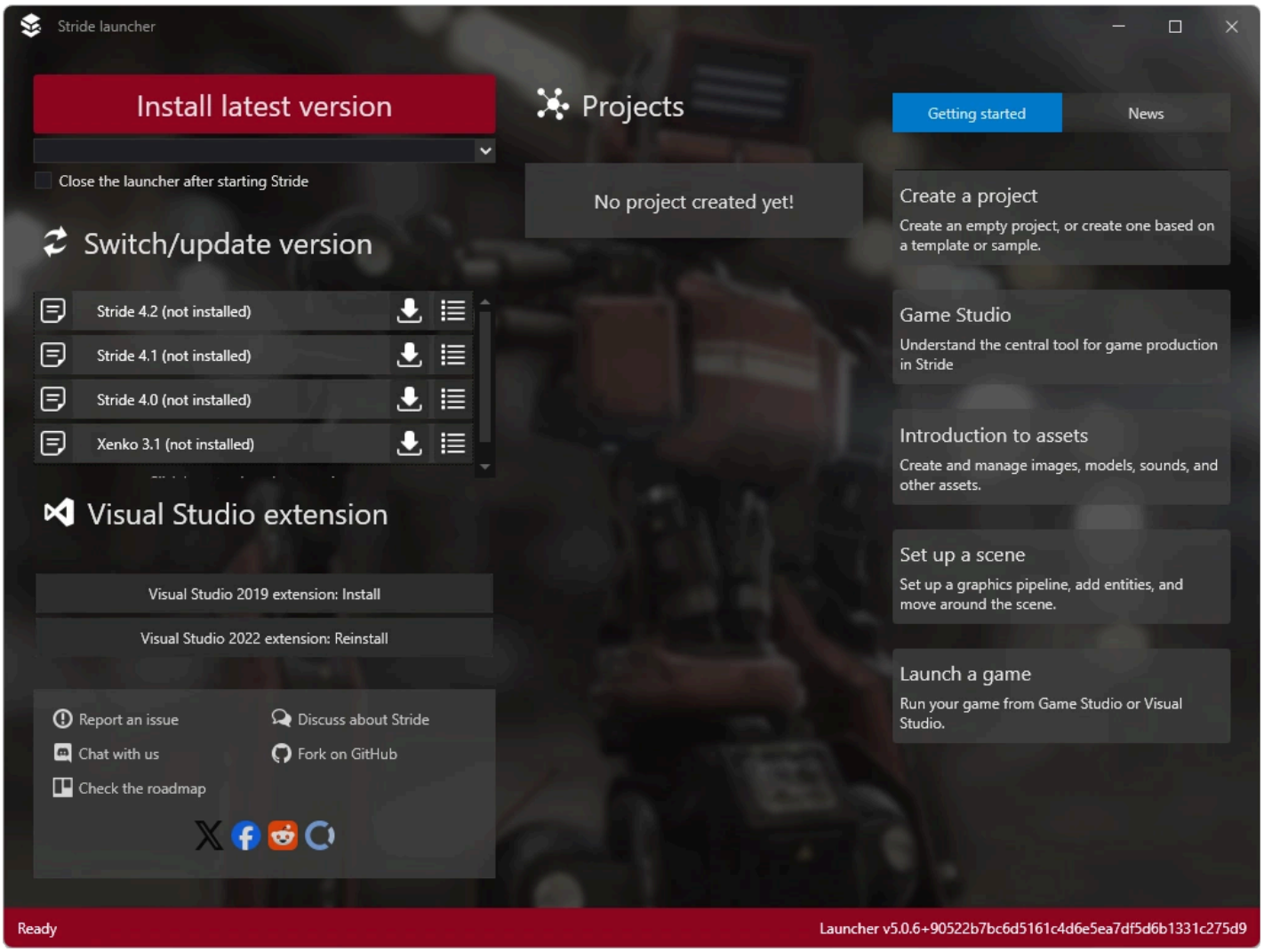
The installer will proceed with the installation. After it completes, Stride creates shortcuts in the locations you selected, and the **Stride Launcher** starts automatically.

9. The first time you run the Stride Launcher, you will be asked to accept the privacy policy.

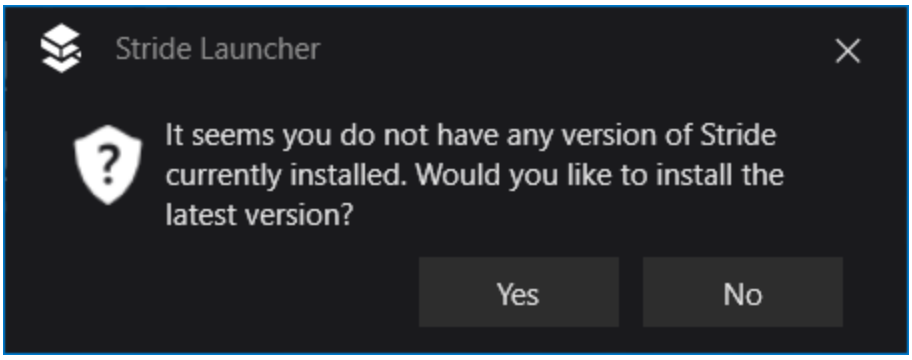


Check *I agree to the Privacy Policy*, then click **I Accept**.

10. The **Stride Launcher** window opens.

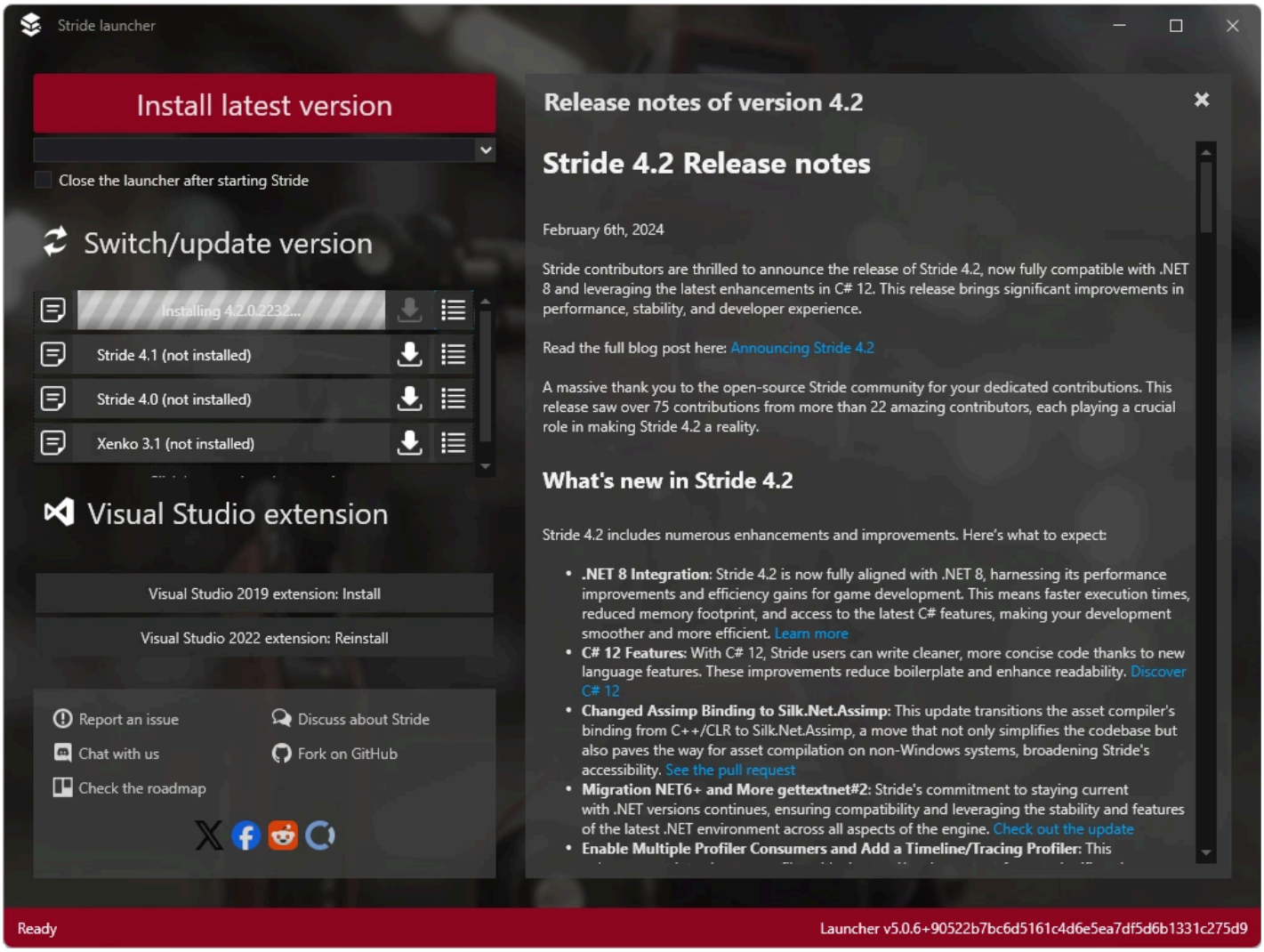


The Stride Launcher prompts you to install the latest version of Stride.



Click **Yes** to install the latest version.

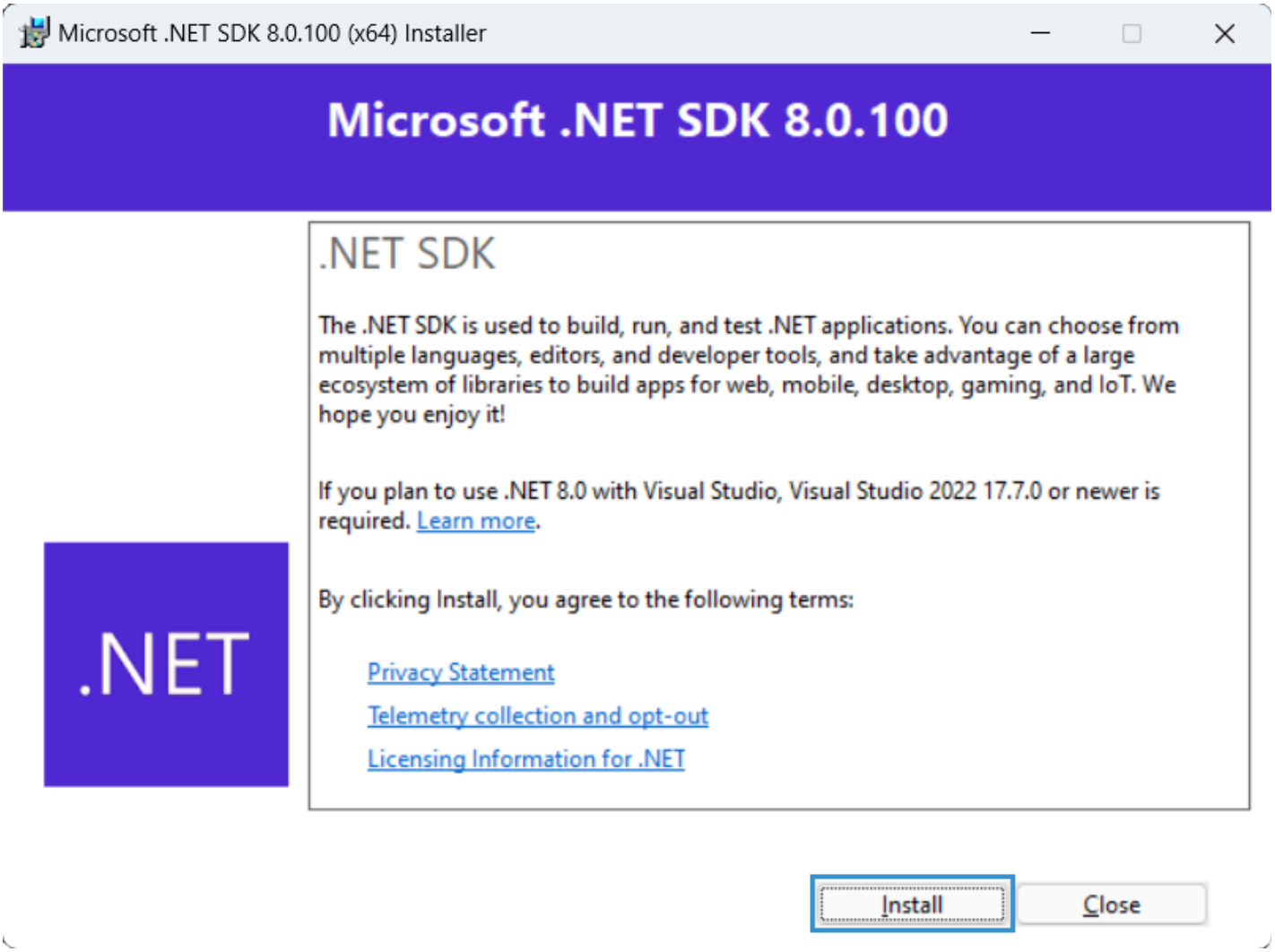
11. Installation of the latest version of Stride begins.



While the installation is in progress, the release notes are displayed.

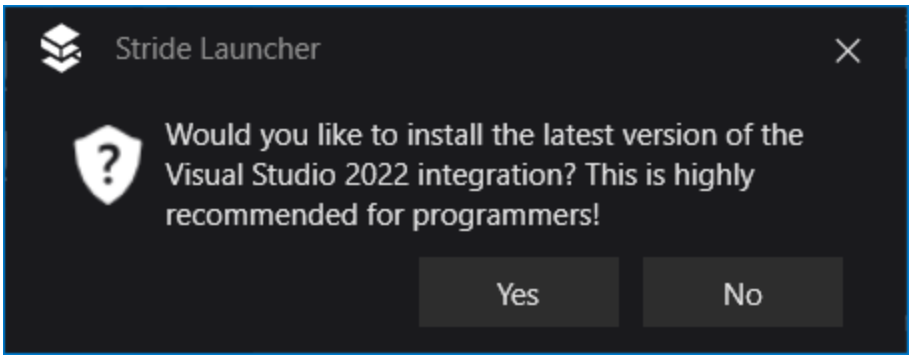
> **WARNING**
>
> If the .NET SDK has never been installed on your machine, the .NET SDK installation window might appear below the Stride installation window. Please check step 12 for details and be prepared to manually continue the .NET SDK installation.

12. During the installation, you might be asked to install the .NET SDK if it's not already on your machine.



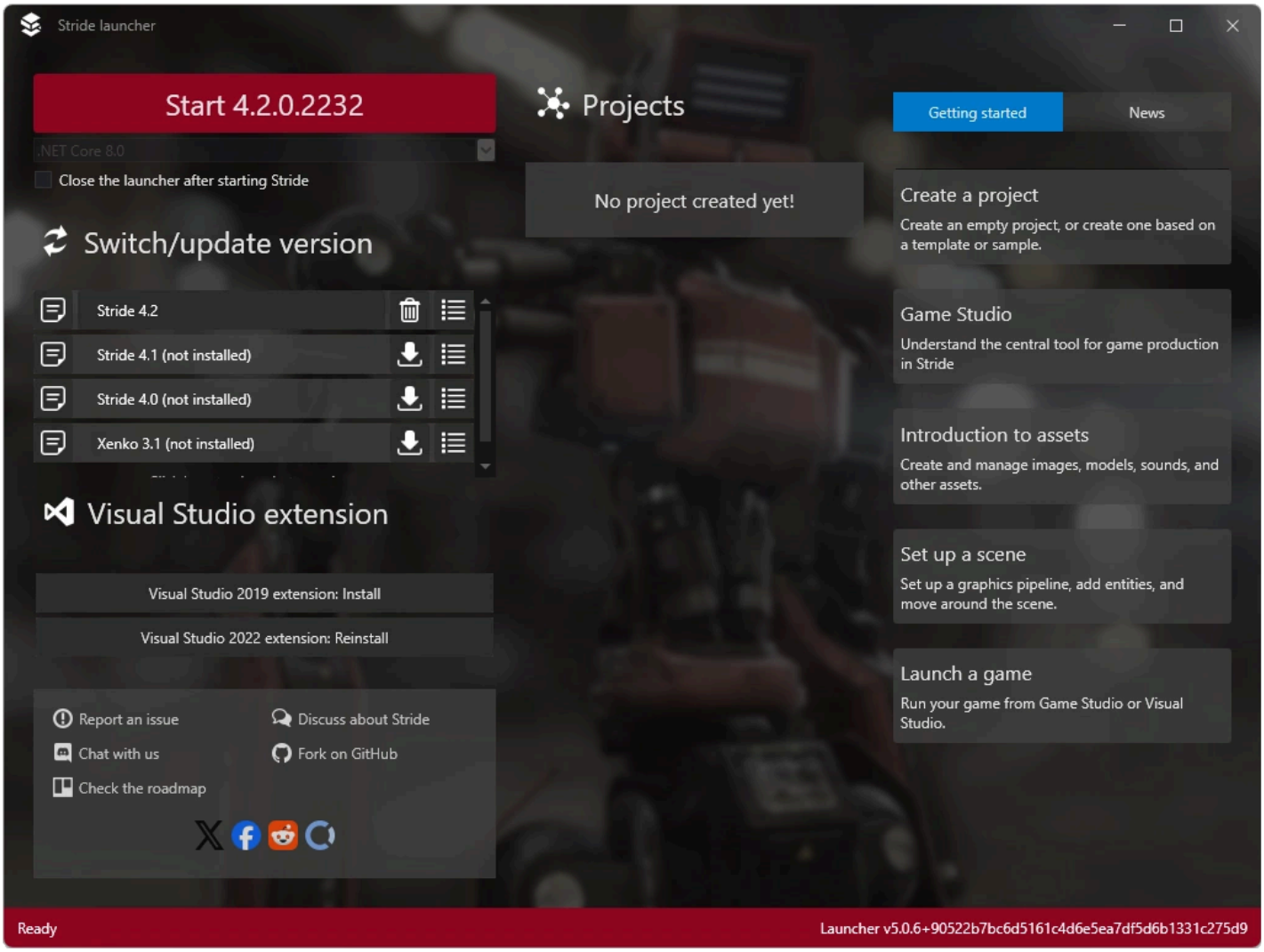
Click **Install**.

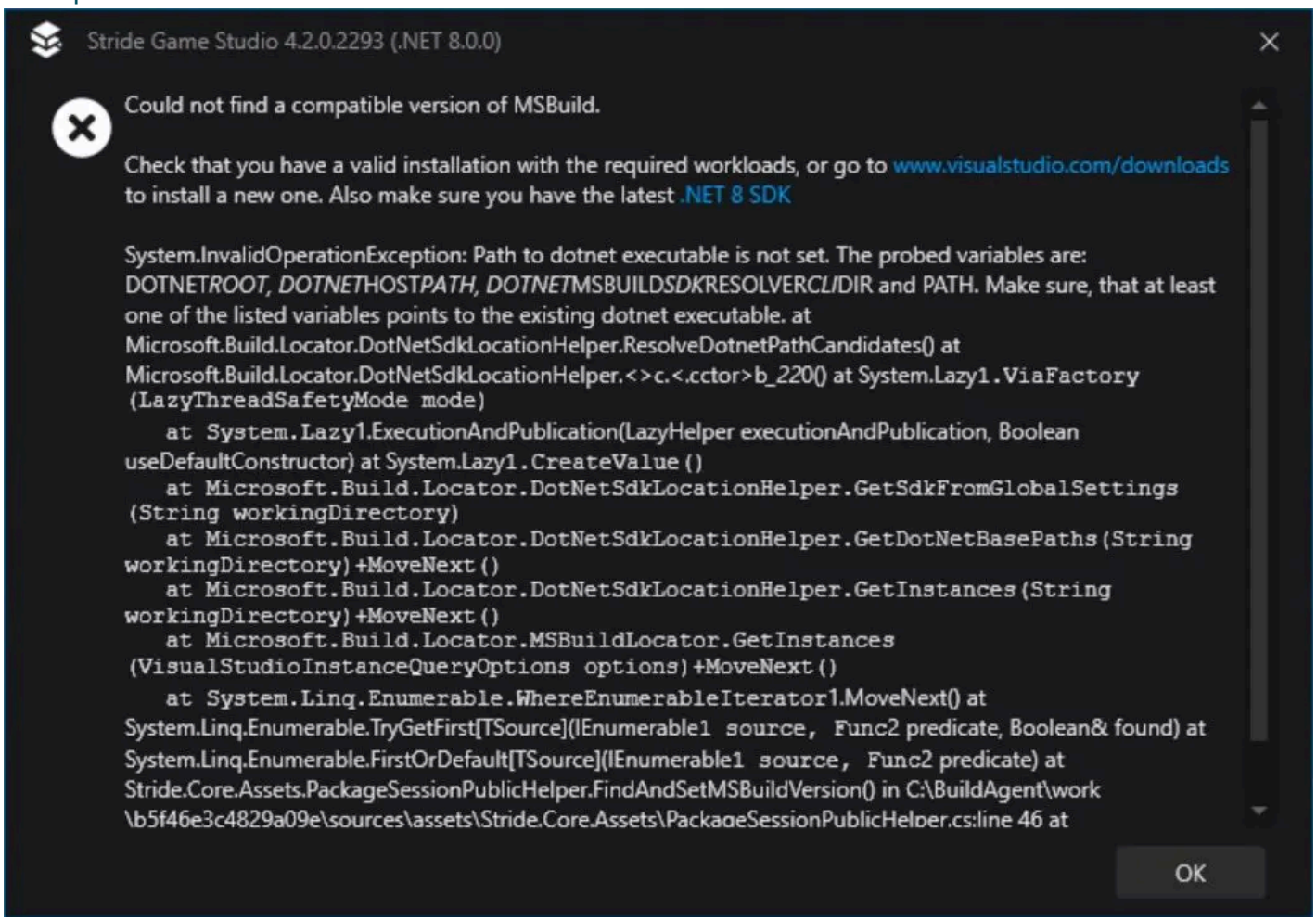
13. The Stride Launcher asks if you want to install the Visual Studio integration. This allows you to edit shaders directly from Visual Studio, providing syntax highlighting, live code analysis with validation, error-checking, and navigation. Installing the integration isn't mandatory, but we recommend it.



Click **Yes** to install the integration, or **No** to skip.

14. Stride is now installed and ready to use.



> **NOTE**
>
> **Stride Launcher:** If you click **Start** and see an error message such as `Could not find a compatible version of MSBuild.` or `Path to dotnet executable is not set.`, close the Stride Launcher and restart it. This issue is caused by the Stride Launcher not detecting the .NET SDK installation. Restarting the Stride Launcher should resolve the issue. Alternatively, restart your computer.
>
> Example error:
>
> 
>

> **NOTE**
>
> If you don't install the prerequisites, Stride won't run. In this case, you can download and install the prerequisites separately. For instructions, see Troubleshooting — Stride doesn't run.
>
> Alternatively, uninstall Stride, restart the Stride installer, and install the prerequisites when prompted.

What's next?

- Launch Stride

Visual Studio extension

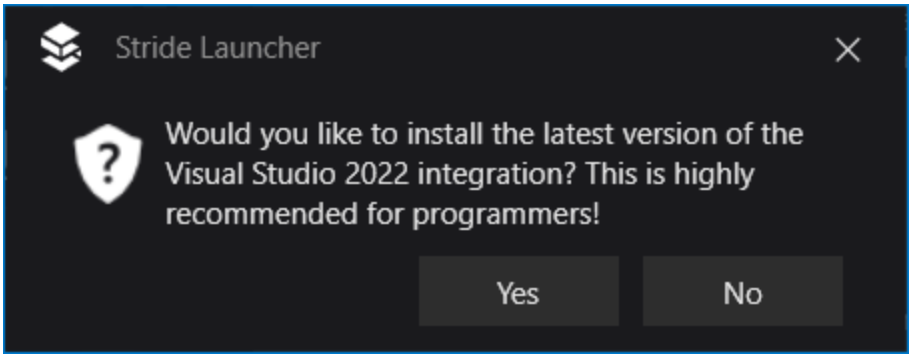
Beginner

The **Stride Visual Studio extension** lets you edit shaders directly from Visual Studio.

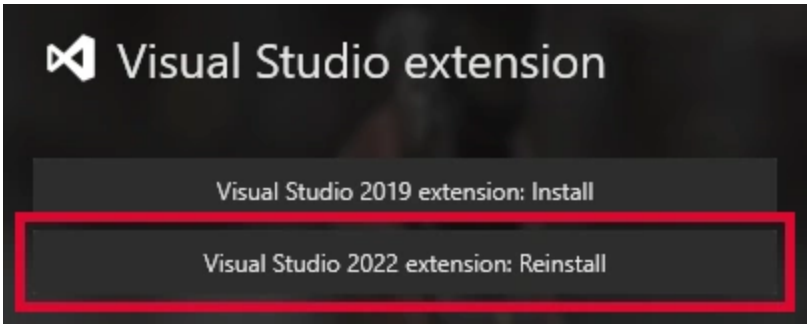
You don't need to install the extension to use Stride, but we recommend it, especially for programmers.

Install the Stride Visual Studio Extension

When you install Stride, Stride asks if you want to install the Visual Studio extension.



Alternatively, you can install or reinstall the extension at any time in the Stride Launcher under **Plug-in**.



See also

- Custom shaders

Update Stride

Beginner

Updating Stride is a straightforward process, but it's important to follow the steps carefully to ensure a seamless transition. Below are the guidelines for updating both the Stride engine and your existing projects.

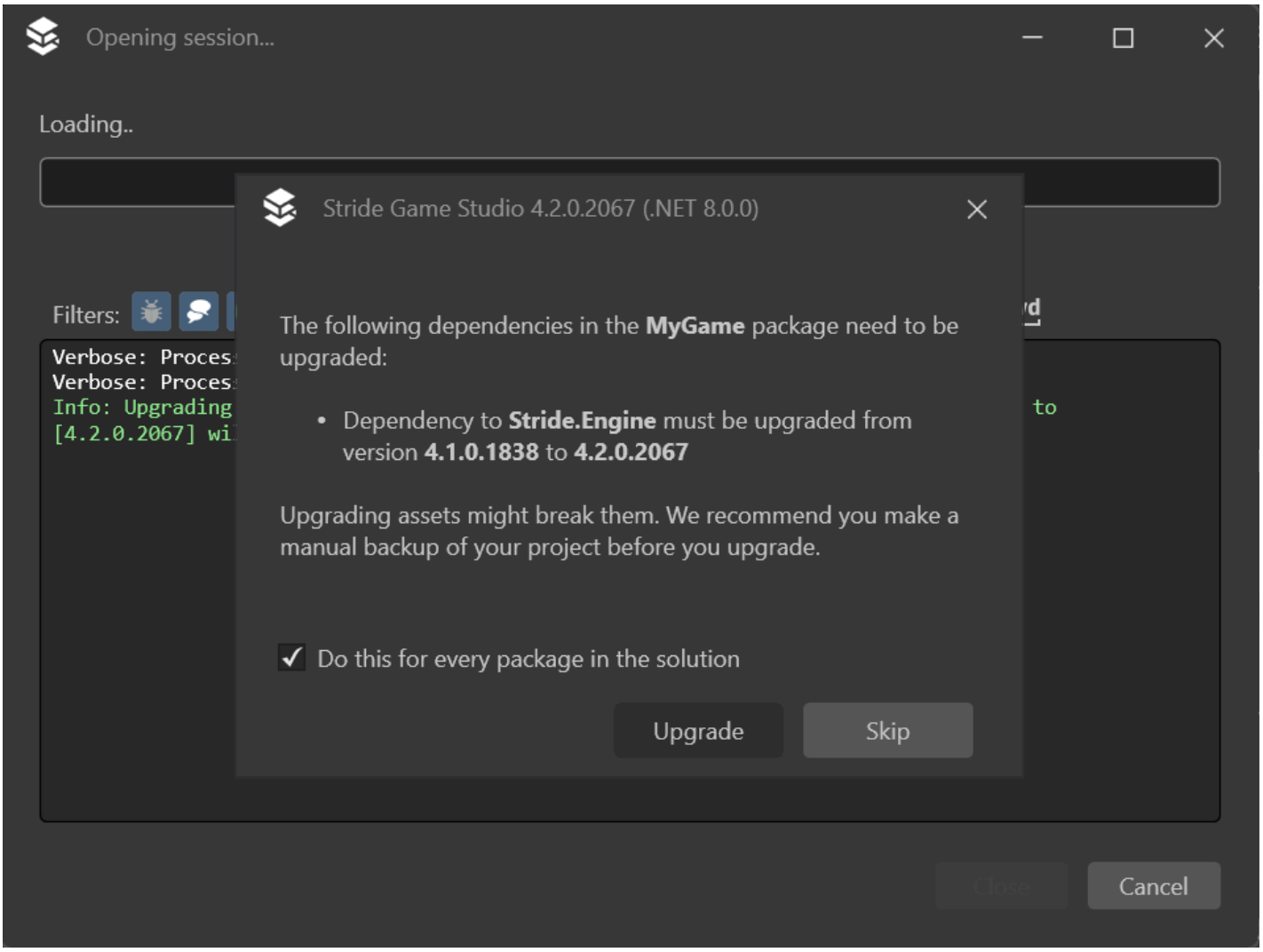
> **NOTE**
>
> The instructions provided here can be used as a general guide for updating to any new version of Stride.

Updating Stride

1. **Update Visual Studio 2026:** Ensure that you have the latest version of Visual Studio 2026. This is crucial for compatibility with the latest Stride version. After updating Visual Studio, restart your computer to apply the changes fully.
2. **Stride Launcher Instructions:** Open the Stride Launcher. Follow the instructions provided to update or install the Visual Studio plugin for Stride. This step is essential for integrating the latest version of Stride with your development environment.
3. **Restart Again:** After completing the installation or update of the Visual Studio plugin, restart your computer once more. This helps to ensure that all components are correctly loaded and ready for use.

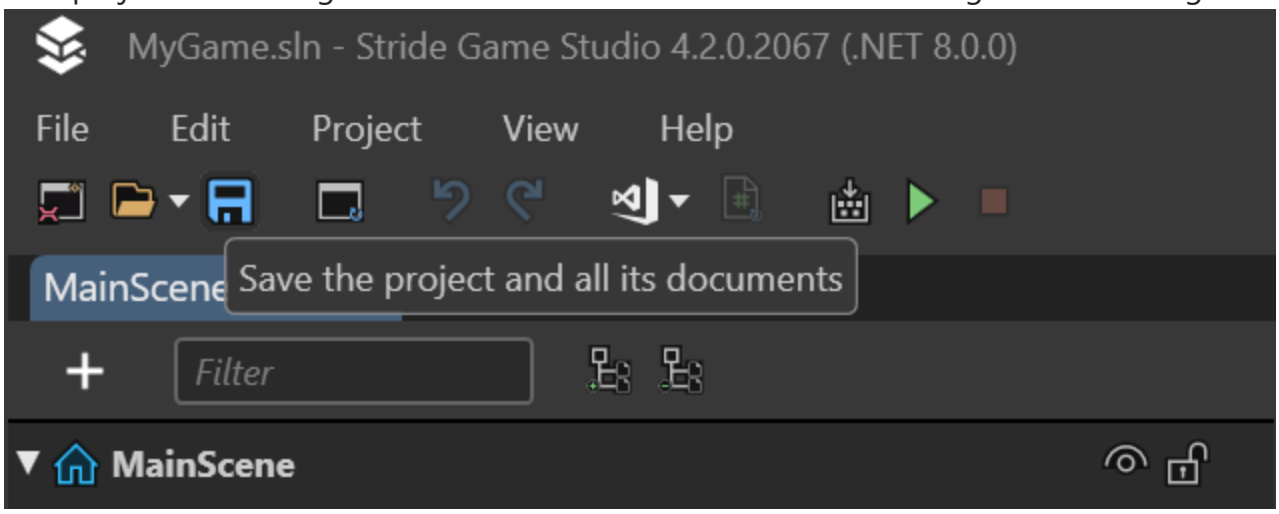
Updating Your Project

1. **Version Control:** Before proceeding with the update, confirm that your project is under version control with all current changes committed. This provides a safety net, allowing you to revert to the previous state if needed. If you're not using version control, ensure you have a backup of your project.
2. **Opening the Project:** When you open a project created with an older version of Stride, a dialogue will appear, prompting you to update the project. Make sure to check the option to apply the update to all packages in the solution. Additionally, you can verify later whether all packages have been updated by checking your project files, specifically the `.csproj` files.



Dialog prompting for project update in Stride.

3. **Saving the Project:** After Stride updates the project, it's crucial to save it immediately. This step prevents the project from being in an undefined state and solidifies the changes made during the update.



4. **Rebuild and Reload:** Finally, rebuild the project and reload assemblies. This ensures that all components are up-to-date and properly synchronized with the new version of Stride.

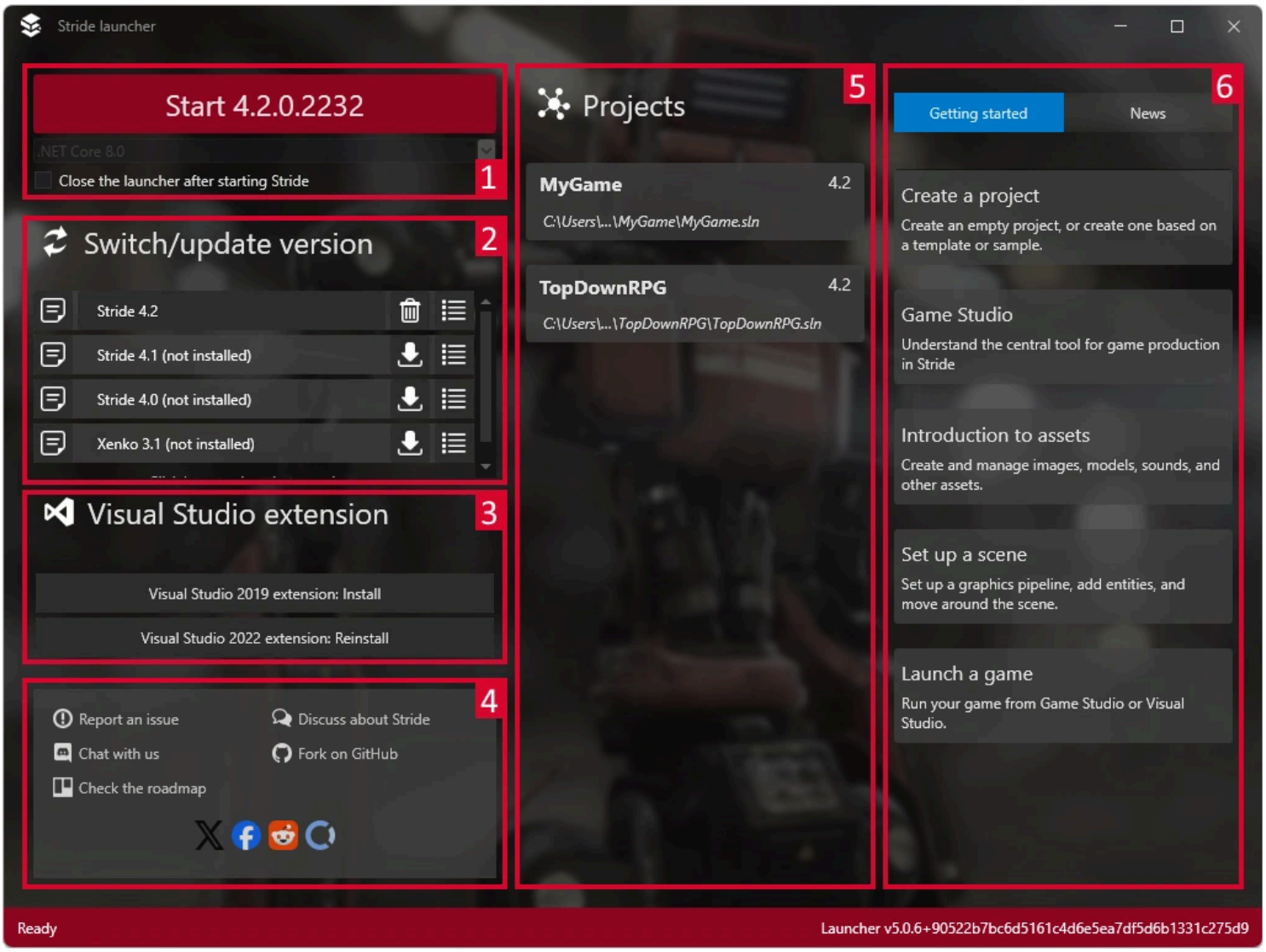
By following these steps, you can smoothly transition to the latest version of Stride, taking full advantage of the new features and improvements it offers. Remember, these procedures are designed to provide a

hassle-free update experience and safeguard your project against potential issues.

Launch Stride

Beginner

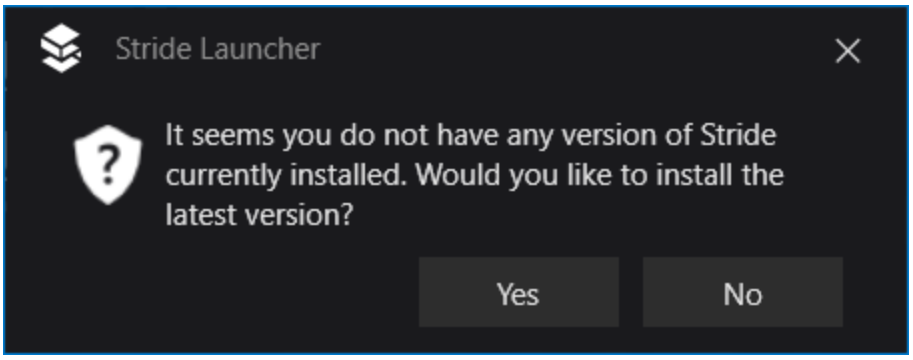
With the **Stride launcher**, you can install, manage and run different versions of Stride.



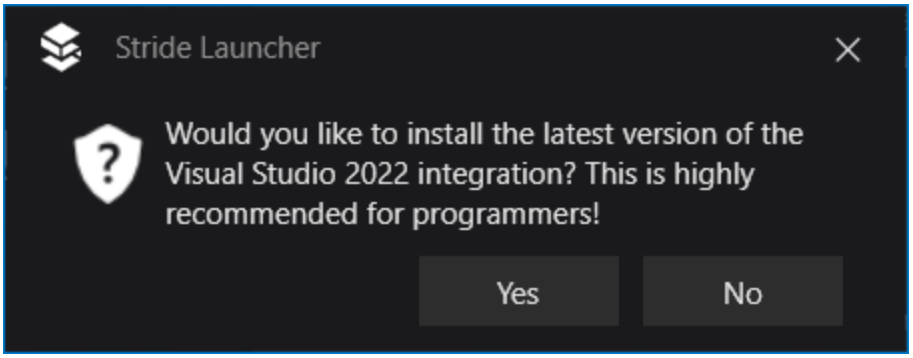
1 Start Stride Game Studio
2 Stride Different Versions
3 Visual Studio Plugins
4 Interact with Community
5 Open Recent Projects
6 Documentation and News

Install the latest version of Stride

If you don't have Stride installed, the Stride Launcher prompts you to install the latest version.



If you choose to install the latest version, the Stride Launcher asks if you want to install the Stride Visual Studio extension.



The Stride Visual Studio extension lets you you edit shaders directly from Visual Studio. You don't need to install the extension to use Stride, but we recommend it, especially for programmers.

Manage different versions of Stride

You can install multiple versions of Stride and launch them from the Stride Launcher.

1 Release Notes
2 Version
3 Uninstall
4 Download and Install
5 Install Minor Versions

You might need to use an older version of Stride to work with old projects. Newer versions of Stride might contain changes that require old projects to be upgraded.

For minor versions, only the last number of the version number changes (1.9.0, 1.9.1, 1.9.2, etc). Minor versions don't contain breaking changes, so they're safe to install and use with your existing projects.

> **NOTE**
> You can't revert to earlier minor versions. For example, you can install both Stride 1.9 and 1.8 side by side, but you can't revert from Stride 1.9.2 to Stride 1.9.1.

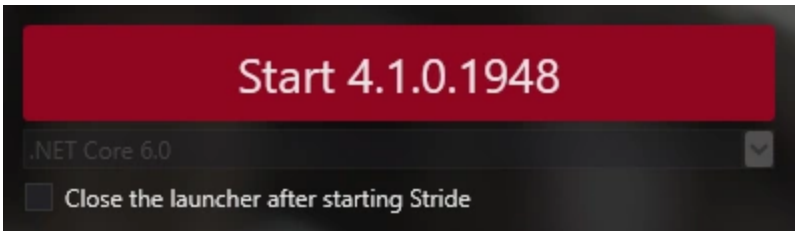
- To see the release notes for a particular version, click the **note icon** next to the version name.
- To install a particular version, click the **Download and install** icon next to the version name.

Start Game Studio

Now you've installed Stride, it's time to start Game Studio and build a project.

1. Under **Switch/update version**, select the version of Stride you want to use.

   The version number is updated on the **Start** button.



2. Click **Start** to launch Game Studio.

What's next?

- Create your first project in Game Studio

See also

For more details about the Stride launcher, see the Stride launcher page.

Create a project

Beginner

This page explains how to:

- create a new empty project
- create a project based on a template or sample

Templates are projects that contain just the necessary elements to start working on a game.

Samples are complete games, which you can learn from or base a new game on.

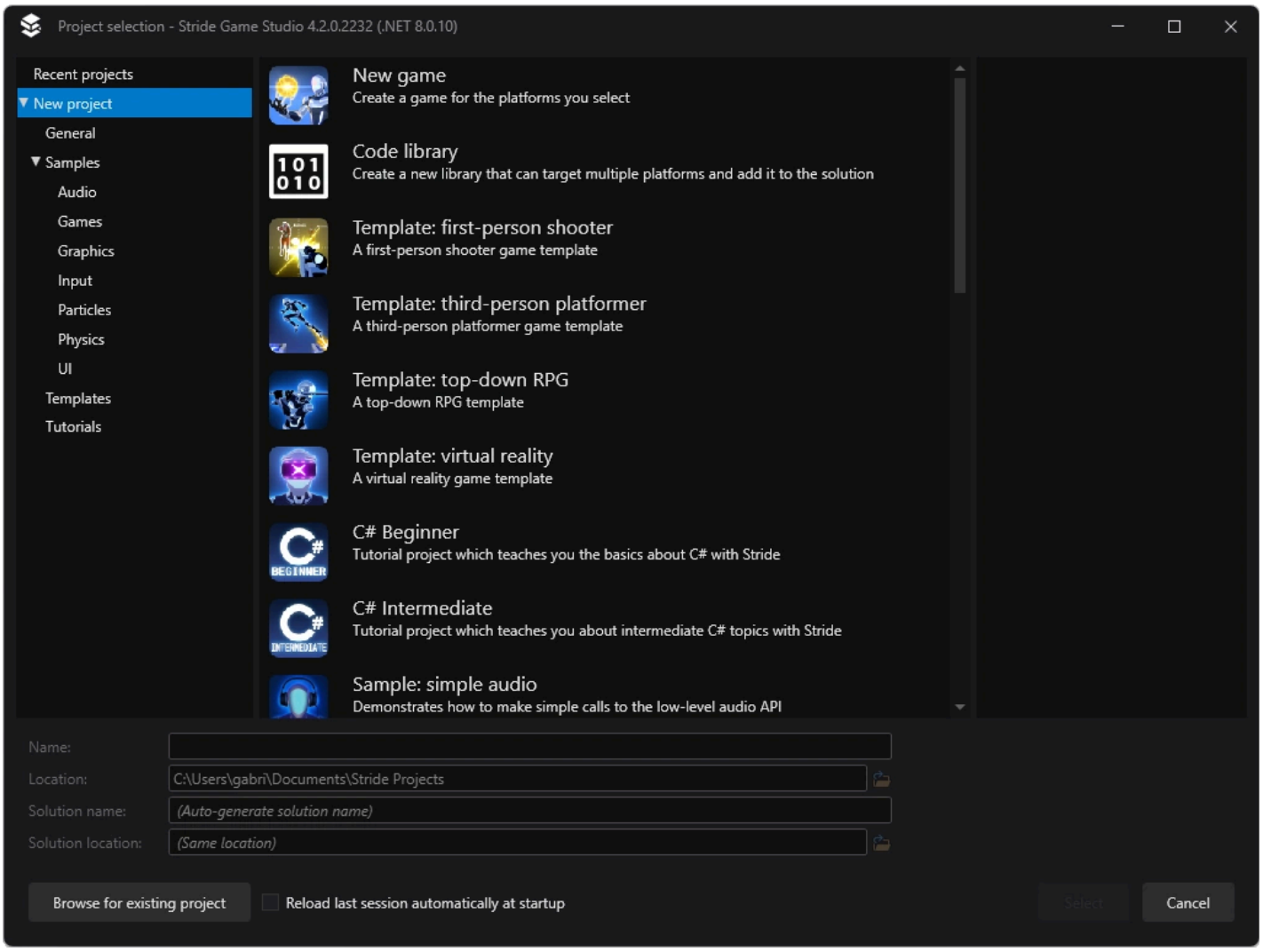
Create an empty project

An **empty project** is project that contains only the bare minimum to make a game: a simple scene with a light, camera, and script to move the camera, plus a preconfigured rendering pipeline. This is good when you want to start your game from scratch without elements you don't need.

To create an empty project:

1. In the **Stride Launcher**, click **Start** to start Game Studio.

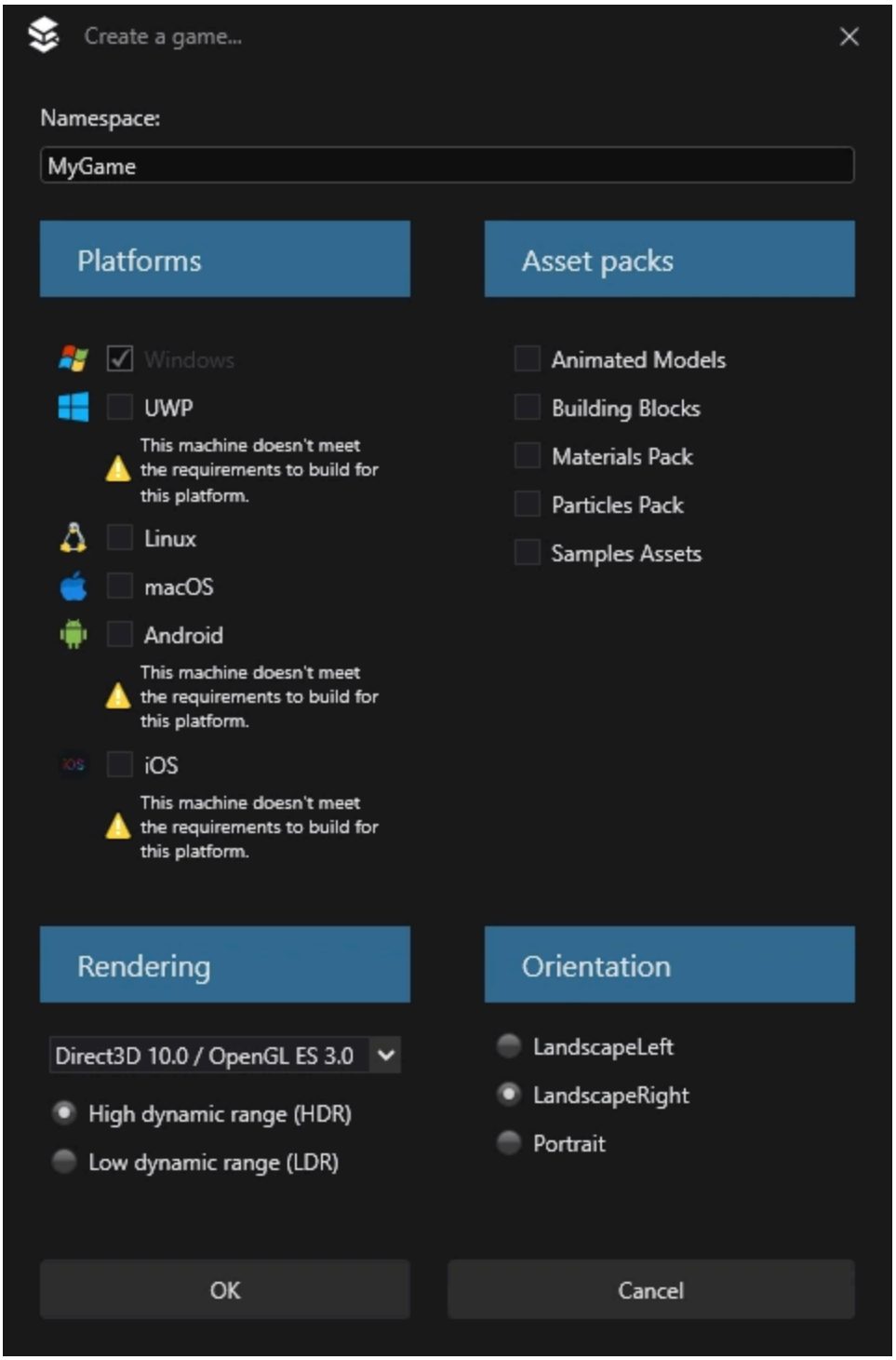
   The **New/open project** dialog opens.



You can also open a new project in Game Studio from **File > New**.

2. Select **New Game**.
3. In the **Name** and **Location** fields, specify a name for the project and the folder to save it in.
4. Click **Select**.

   The **Create a new game** dialog opens.



5. In the **Namespace** field, specify the namespace you want to use. If you don't know what your namespace should be, leave it as default.

6. Under **Platforms**, select the platforms you want your game to support.

> **NOTE**
>
> To support iOS and Android, you need to install Xamarin (free if you have Visual Studio). If your development system is missing prerequisites for any of the platforms you select, Stride displays a warning.

7. Under **Asset Packs**, you can select additional assets to include in your project. These include assets such as animations and materials. The asset packs are fun to play with when you're learning how to use Stride, but they're not necessary.

8. Under **Rendering**, select the options you want.

   Graphics API: The graphics features you can use in your project depend on the API you select. For advanced graphics features, select the latest version of the graphics APIs.

   > **WARNING**
   >
   > Some graphics cards don't support the latest APIs. For some mobile devices, only Direct3D 9.3 / OpenGL ES 2.0 and Direct3D 10.0 / OpenGL ES 3.0 are available.

   High or Low Dynamic Range (HDR / LDR): This defines how color is computed in your project. In LDR mode, colors range from 0 to 1. In HDR mode colors can take any float value. HDR provides more advanced and realistic rendering but requires more processing power and profile Direct3D 10.0 / OpenGL ES 3.0 or later.

9. Under **Orientation**, choose the orientation for your project. For PC games, use landscape. Portrait should usually only be used for mobile games.

10. Click **OK**.

Stride creates the project and opens it in Game Studio. For more information, see Game Studio.

Create a project from a sample or template

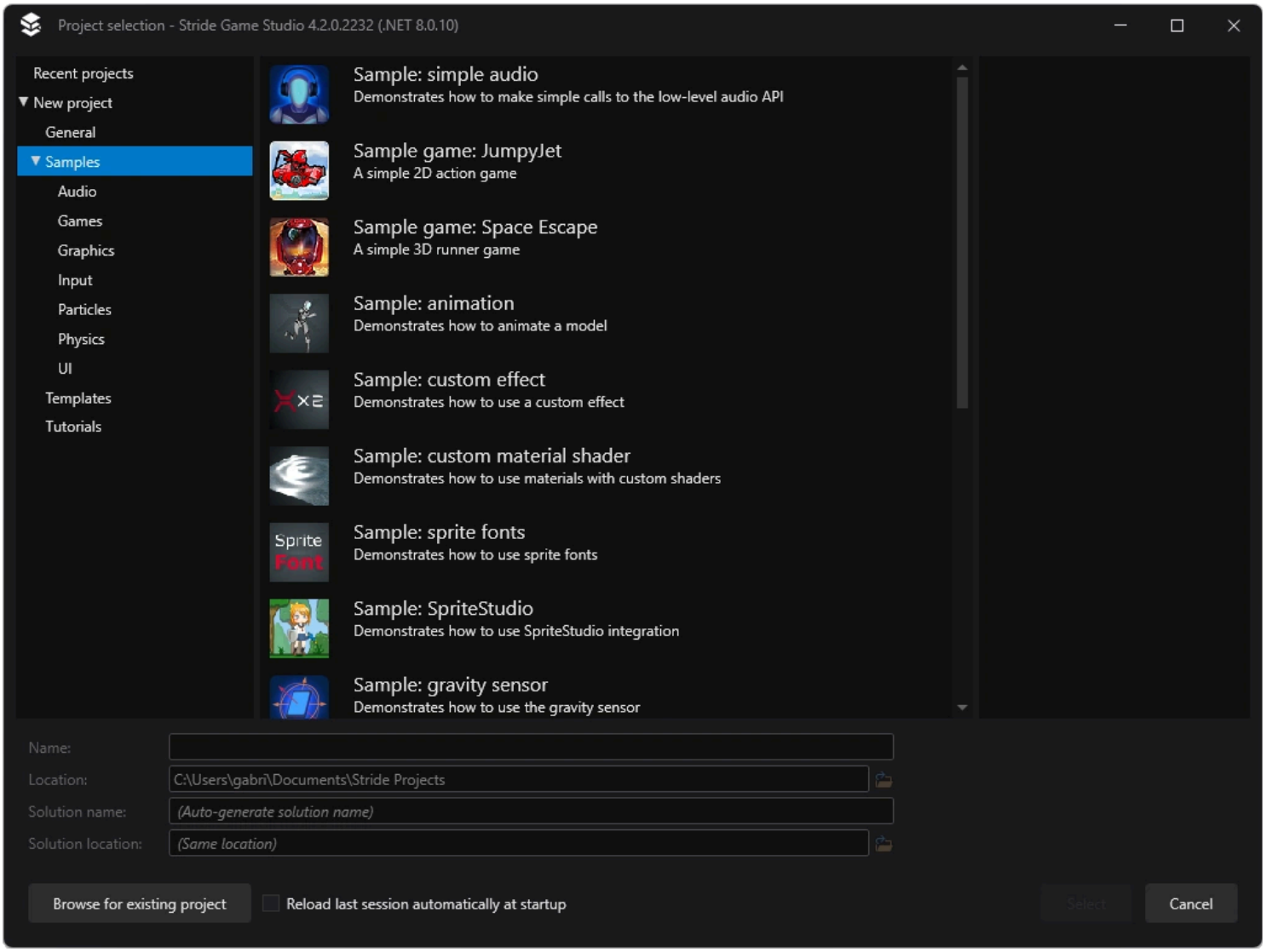
Stride includes several sample projects demonstrating each part of the engine (2D, 3D, sprites, fonts, UI, audio, input, etc). It also includes template games to help you make your own game.

To create a project from a sample or template:

1. Open the **New Project** dialog.

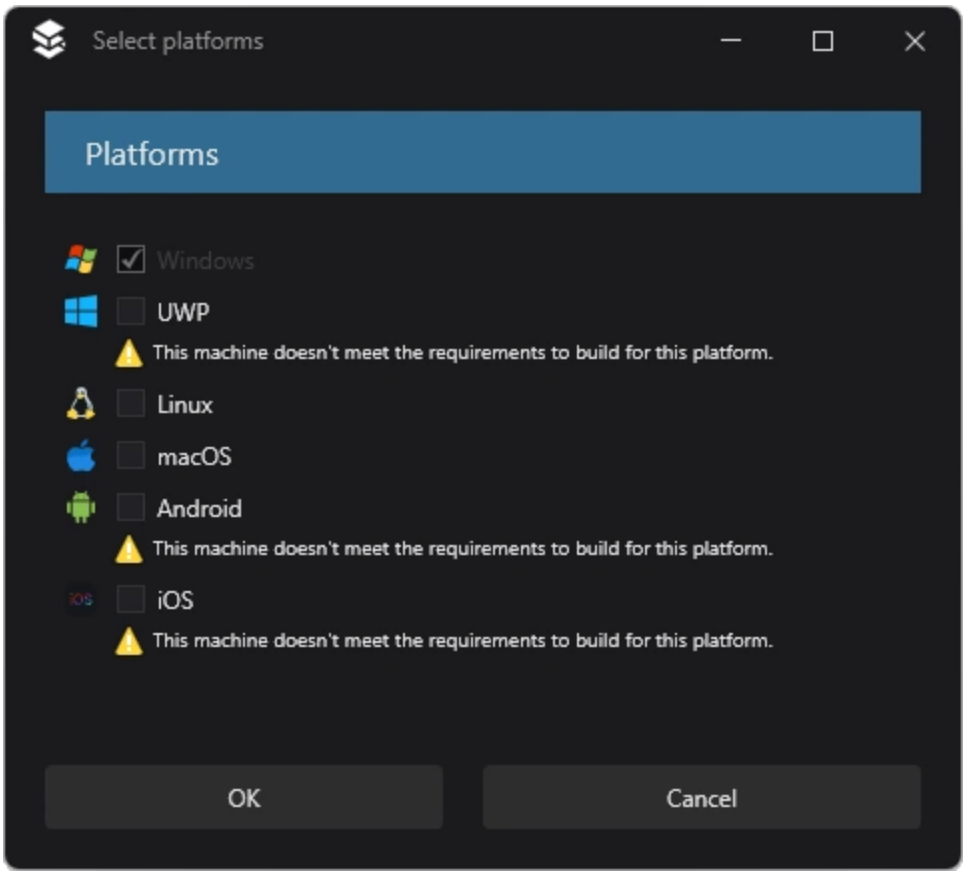
2. On the left, navigate to **New project > Samples**.

3. **Select the sample** you want to create a project from.



4. Click **Select**.

   The **Select Platforms** window opens.



5. Select the platforms you want your game to support and click **OK**.

Stride creates the project and opens it in Game Studio.

What's next?

- Get familiar with Game Studio

Game Studio

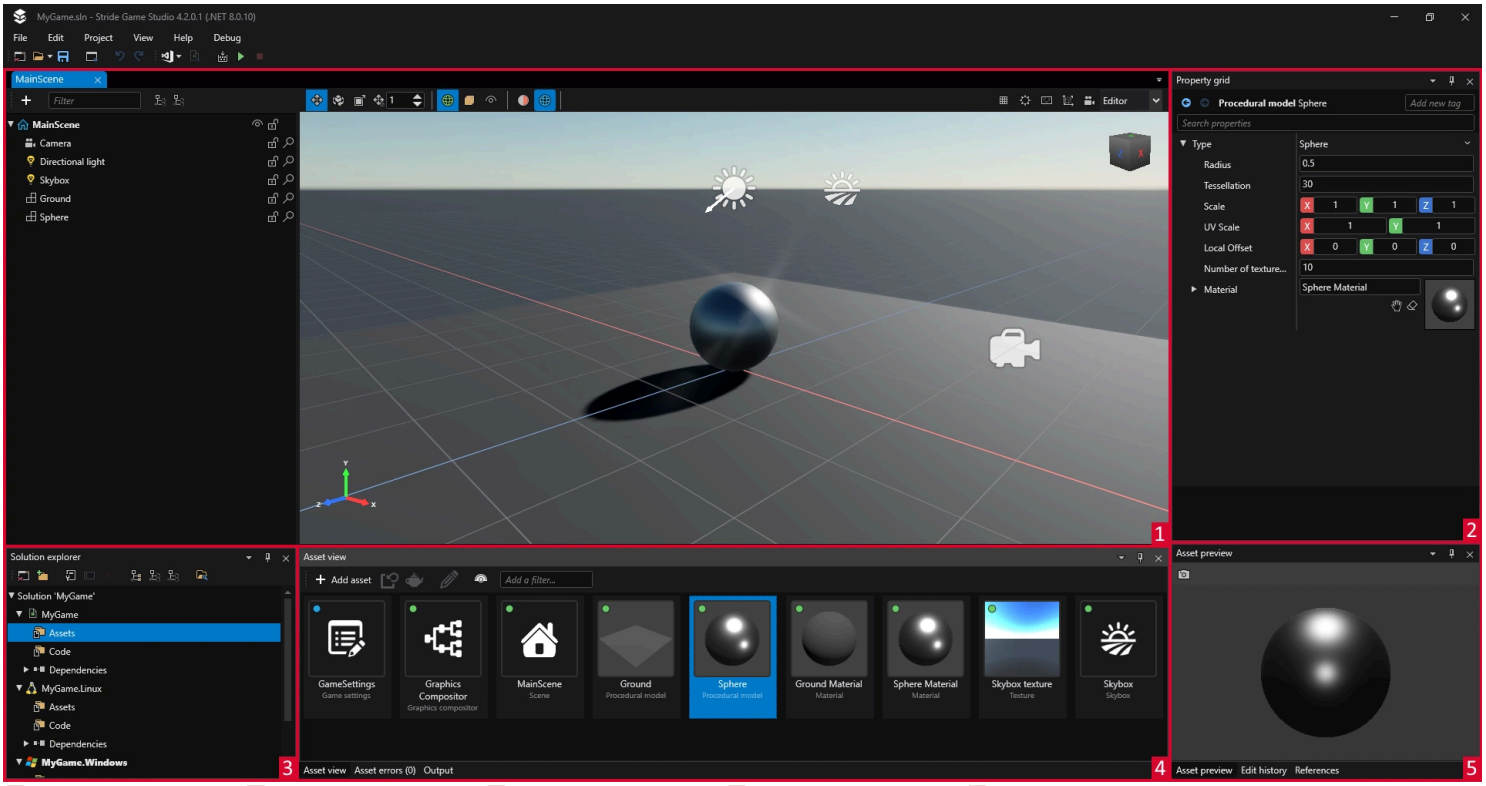
Beginner

Game Studio is the central tool for game and application production in Stride. In Game Studio, you can:

- create and arrange scenes
- import assets, modify their parameters and see changes in real time in the preview window
- organize assets by folder, attach tags and get notifications from modified assets on the disk
- build a game executable and run it directly

Game Studio is also integrated with your Visual Studio projects, so you can seamlessly sync and switch between them.

Interface



1 Asset Editor 2 Property Grid 3 Solution Explorer 4 Asset View 5 Asset Preview

The **asset editor** (1) is used to edit assets and scenes. Some asset types, such as scenes, have dedicated editors where you can make complex changes to the asset. To open a dedicated editor (when available), double-click the asset or right-click it and select **Edit asset**.

The **Property Grid** (2) displays the properties of the asset or entity you select. You can edit the properties here.

The **Solution Explorer** (3) displays the hierarchy of the elements of your project, such as assets, code files, packages and dependencies. You can create folders and objects, rename them, and move them.

The **Asset View** (4) displays the project assets. You can create new assets using the **New Asset** button or by dragging and dropping resource files into the Asset View. You can also drag and drop assets from the Asset View to the different editors or the Property Grid to Create an instance of the asset or add a reference to it. By default, the Asset View is in the bottom center.

The **Asset Preview** tab (5) displays a preview of the selected asset. The preview changes based on the type of the asset you have selected. For example, you can play animations and sounds. This is a quick way to check changes to an asset when editing it in the Property Grid. By default, the Asset Preview is in the bottom right.

You can show and hide different parts of the Game Studio in the View menu. You can also resize and move parts of the UI.

In this section

- Scenes
  - Create a scene
  - Navigate in the Scene Editor
  - Manage scenes
  - Load scenes
  - Add entities
  - Manage entities
- Assets
  - Create assets
  - Use assets
  - Archetypes
  - Game settings
- Prefabs
  - Create a prefab
  - Use prefabs
  - Edit prefabs
  - Nested prefabs
  - Override prefab properties
- World units

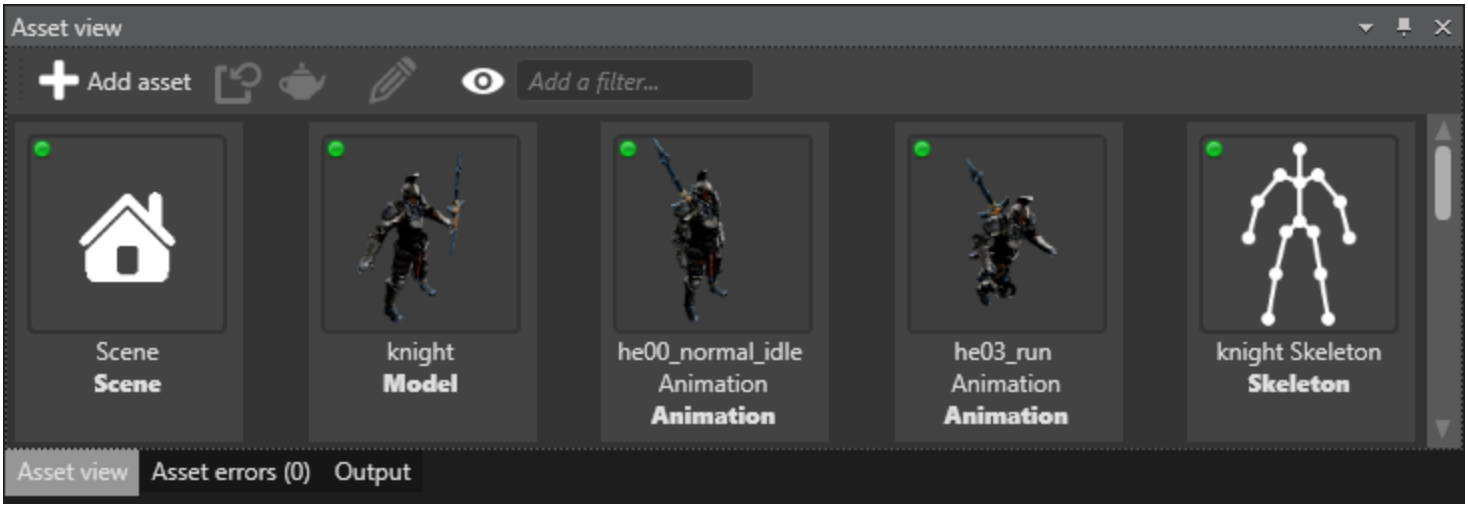
Assets

Beginner

An **asset** is a representation of an element of your game inside Game Studio, such as a texture, animation, or model.

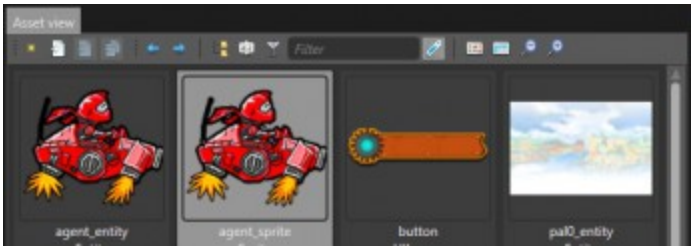
Some assets require **resource files**. For example, texture assets need image files and audio assets need audio files. Other types of assets (such as scenes, physics colliders, and game settings) don't use resource files, and can be created entirely in Game Studio.

You can compile and optimize assets with a special compiler provided by Stride. Compiled assets are packed together as reusable bundles.

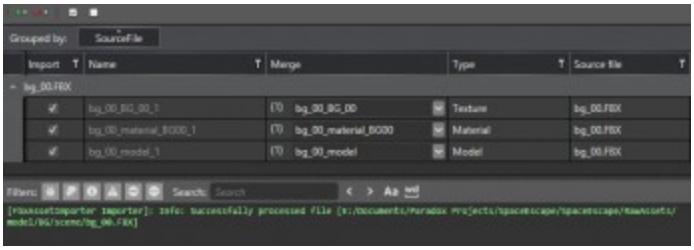


You can:

- create and browse assets in the **Asset View**

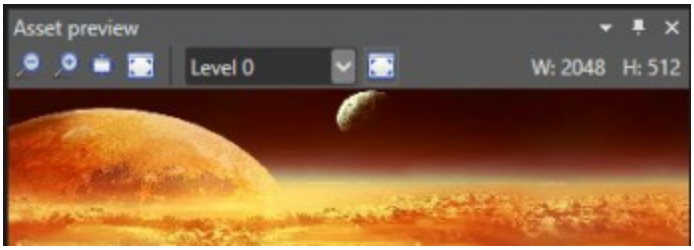


- import existing assets, such as FBX files



- edit assets in the **property editor**

- see a live preview in the **Asset Preview**



In this section

- Create assets
- Manage assets
- Use assets

Create assets

Beginner

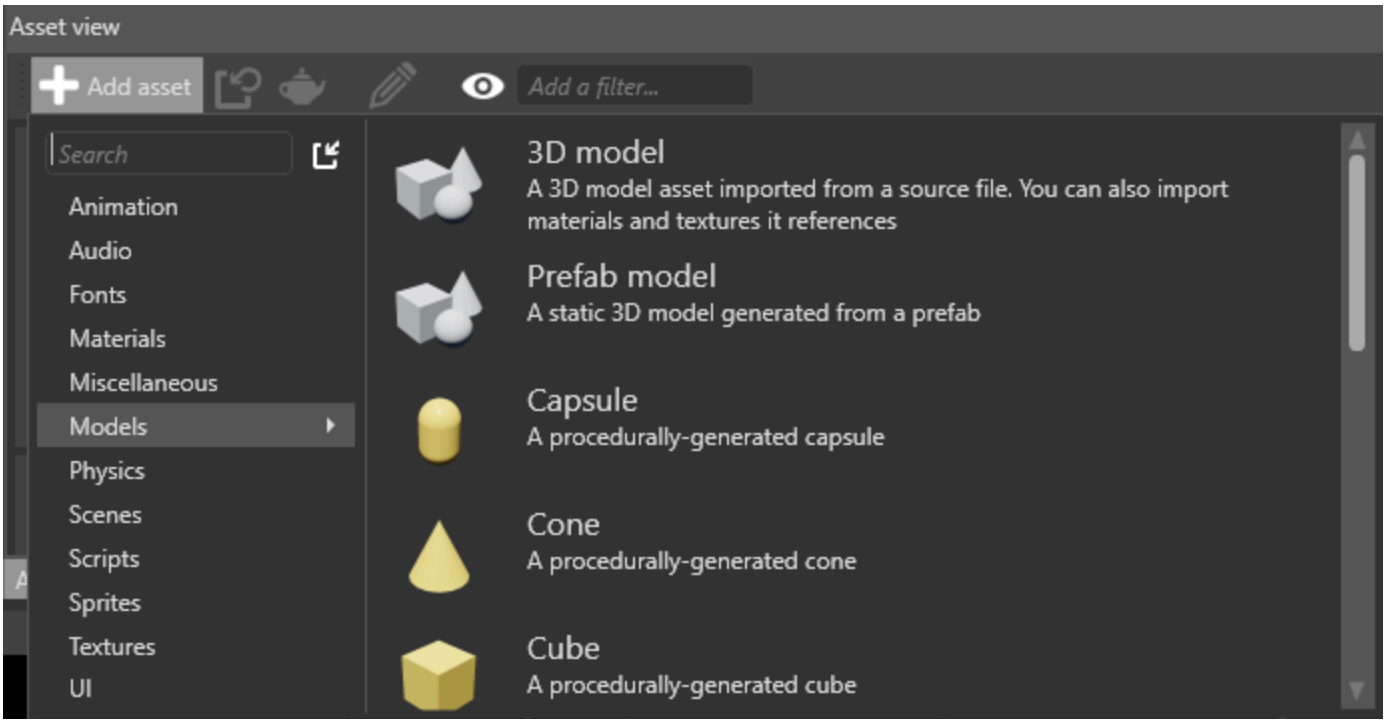
There are two ways to create assets:

- Use the **Add asset** button in the **Asset View**
- Drag and drop **resource files** (such as image or audio files) to the **Asset View** tab

Use the **Add asset** button

1. In the *Asset View*, click Add asset

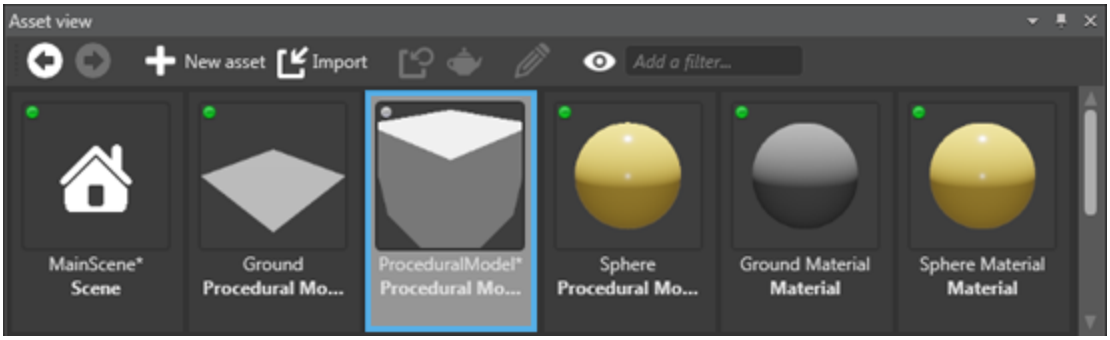
2. Select the type of asset you want to create.



   Game Studio displays a list of asset templates. These are assets configured for a specific use.

3. Select the right template for your asset.

   Game Studio adds the asset to the Asset View:



> **NOTE**
>
> Some assets, such as textures, require a resource file. When you add these assets, Game Studio prompts you for a resource file.

Drag and drop resource files

You can drag compatible resource files directly into Game Studio to create assets from them. Game Studio is compatible with common file formats.

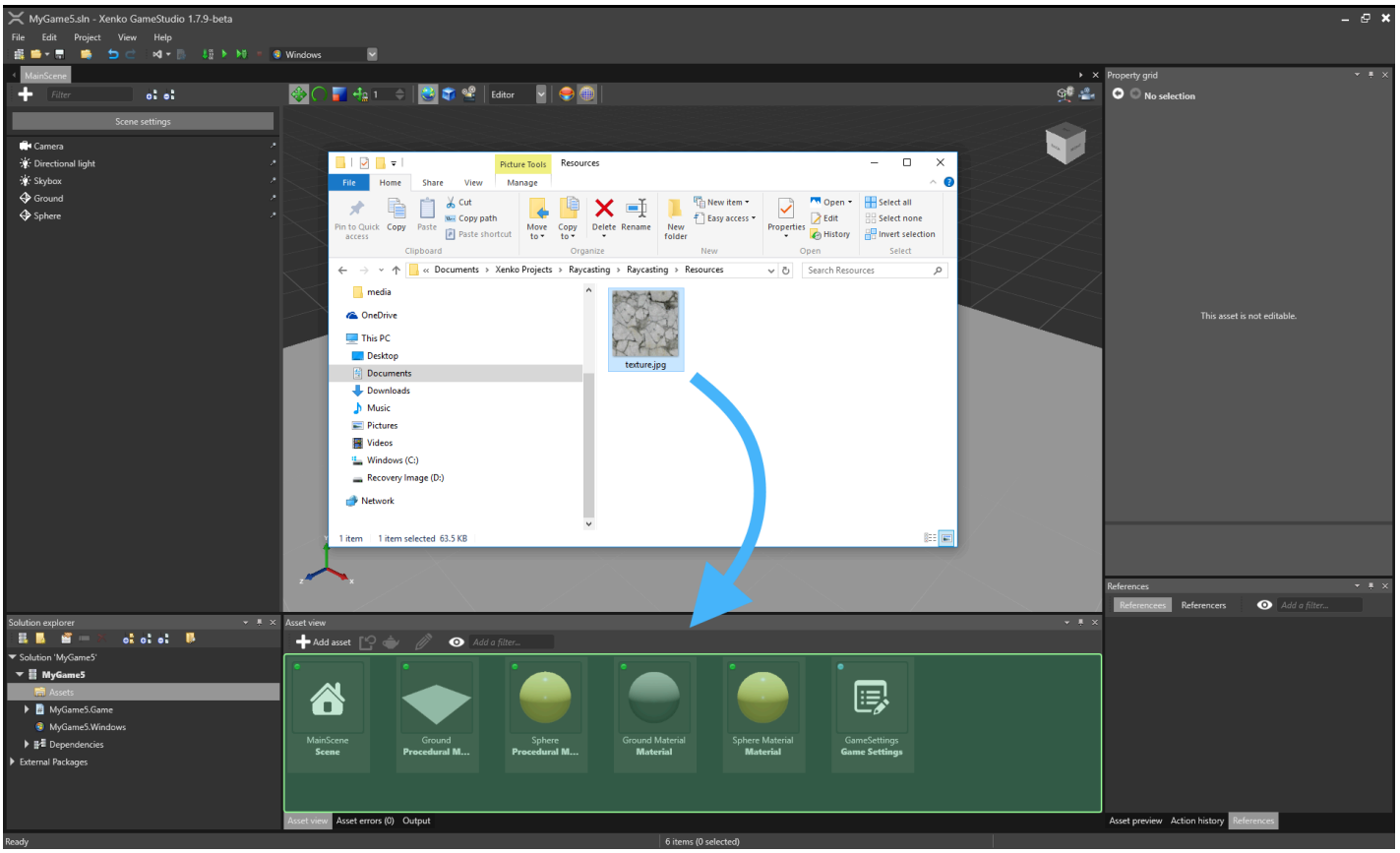
> **NOTE**
>
> - You can't use this method to create assets that don't use resource files (eg prefabs, materials, or scenes).

Asset type	Compatible resource file formats
Models, animations, skeletons	.dae, .3ds, obj, .blend, .x, .md2, .md3, .dxf, .fbx
Sprites, textures, skyboxes	.dds, .jpg, .jpeg, .png, .gif, .bmp, .tga, .psd, .tif, .tiff
Audio	.wav, .mp3, .ogg, .aac, .aiff, .flac, .m4a, .wma, .mpc

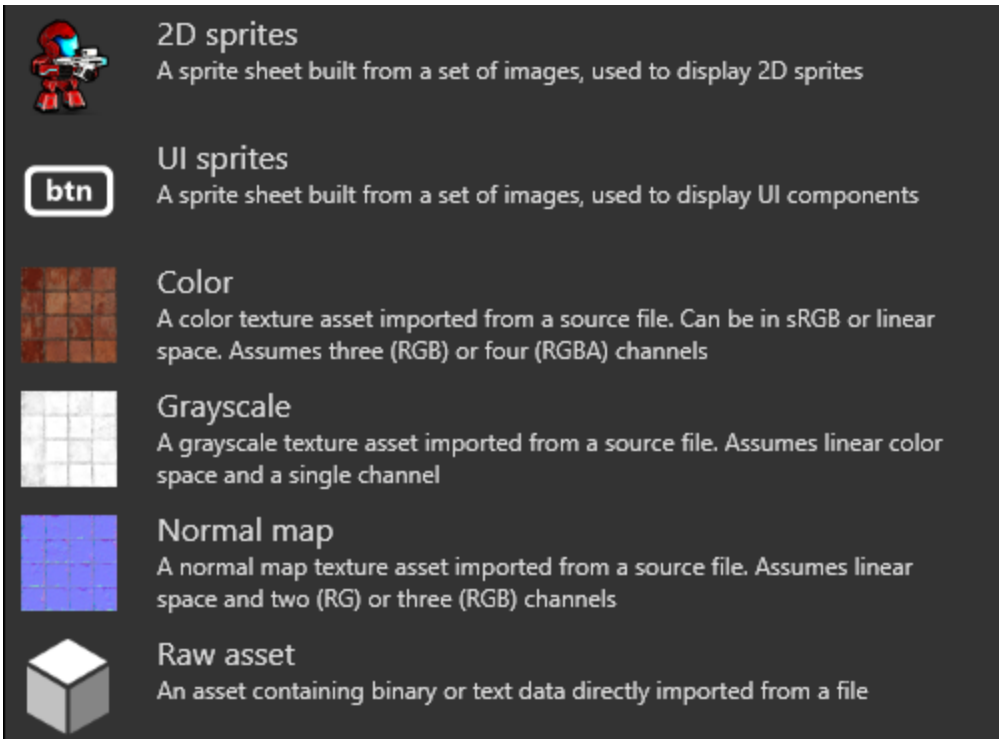
To create an asset by dragging and dropping a resource file:

1. (Optional) If it isn't there already, move the resource file you want to use in the **Resources** folder of your project. You don't have to do this, but it's good practice to keep resource files organized and makes projects easier to share. For more information, see Project structure.

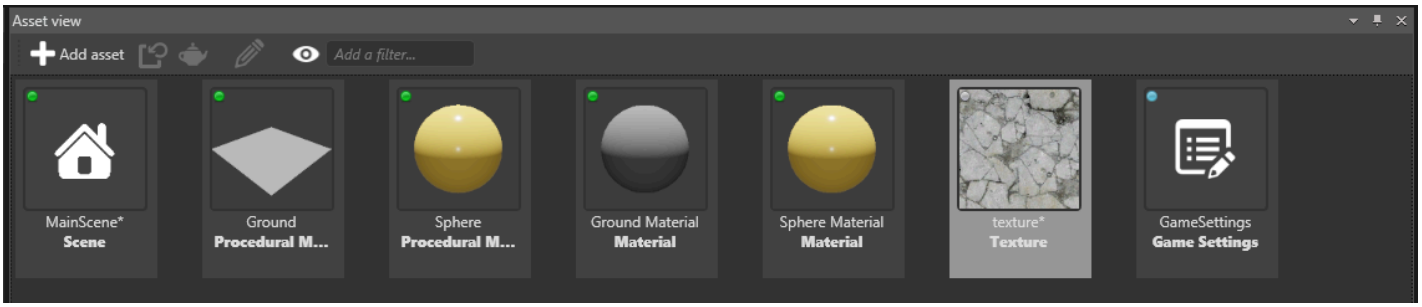
2. Drag the resource file from Explorer to the Asset View:



3. Select the kind of asset you want to create:



Game Studio adds the asset to the Asset View:



Game Studio automatically imports all dependencies in the resource files and creates corresponding assets. For example, you can add a model or animation resource file and Game Studio handles everything else.

> **TIP**
>
> You can drag multiple files simultaneously. If you drop multiple files of different types at the same time, Game Studio only adds only files that match your template selection. For example, if you add an image file and a sound file, then select the audio asset template, only the sound file is added.

See also

- Manage assets
- Use assets

Use assets

Beginner

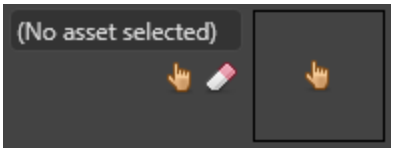
There are four ways to use assets:

- reference them in entity components
- reference them in other assets
- load them from code as content
- load them from code as content using `UrlReference`

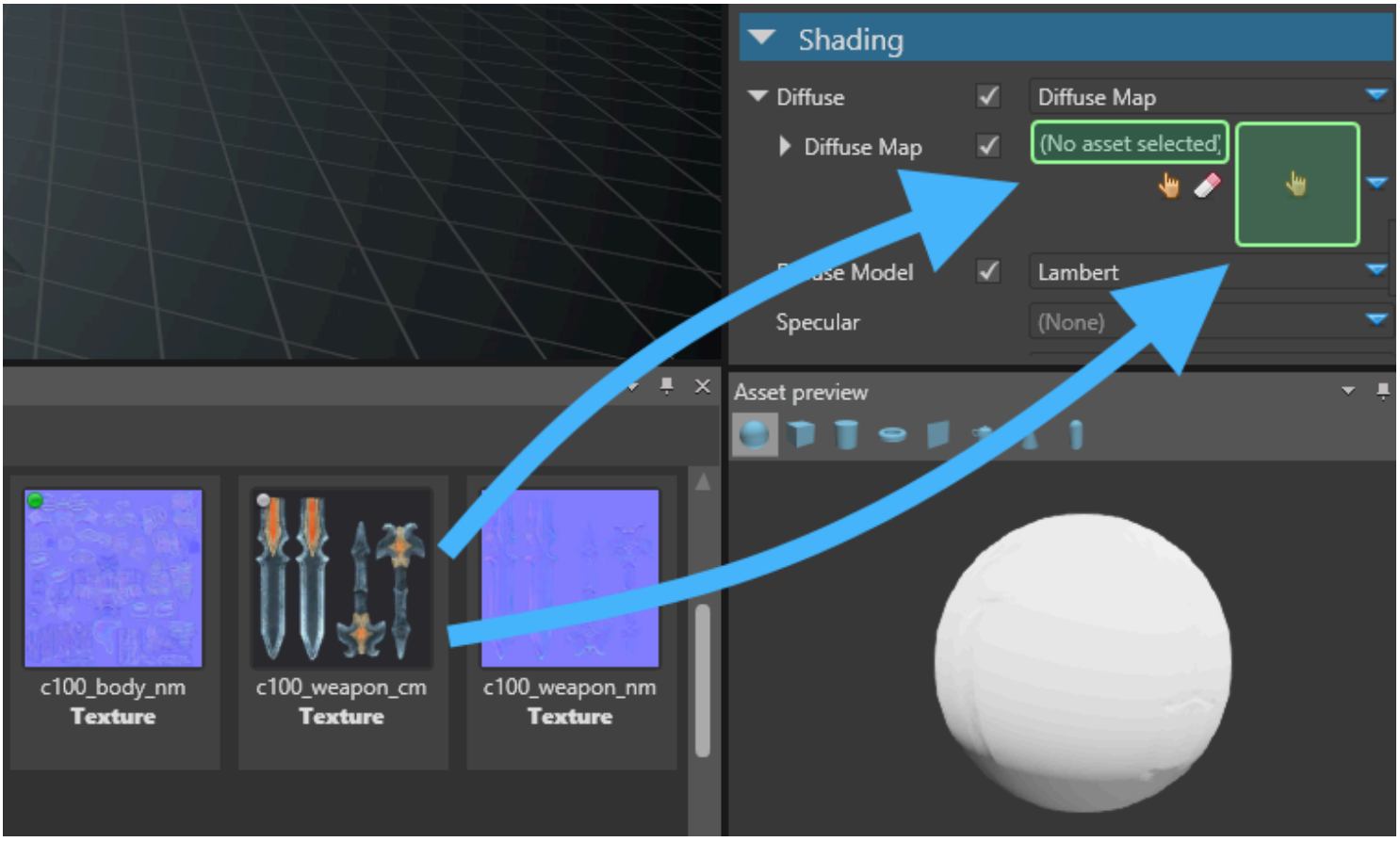
Reference assets in components

Many kinds of component use assets. For example, model components use model assets.

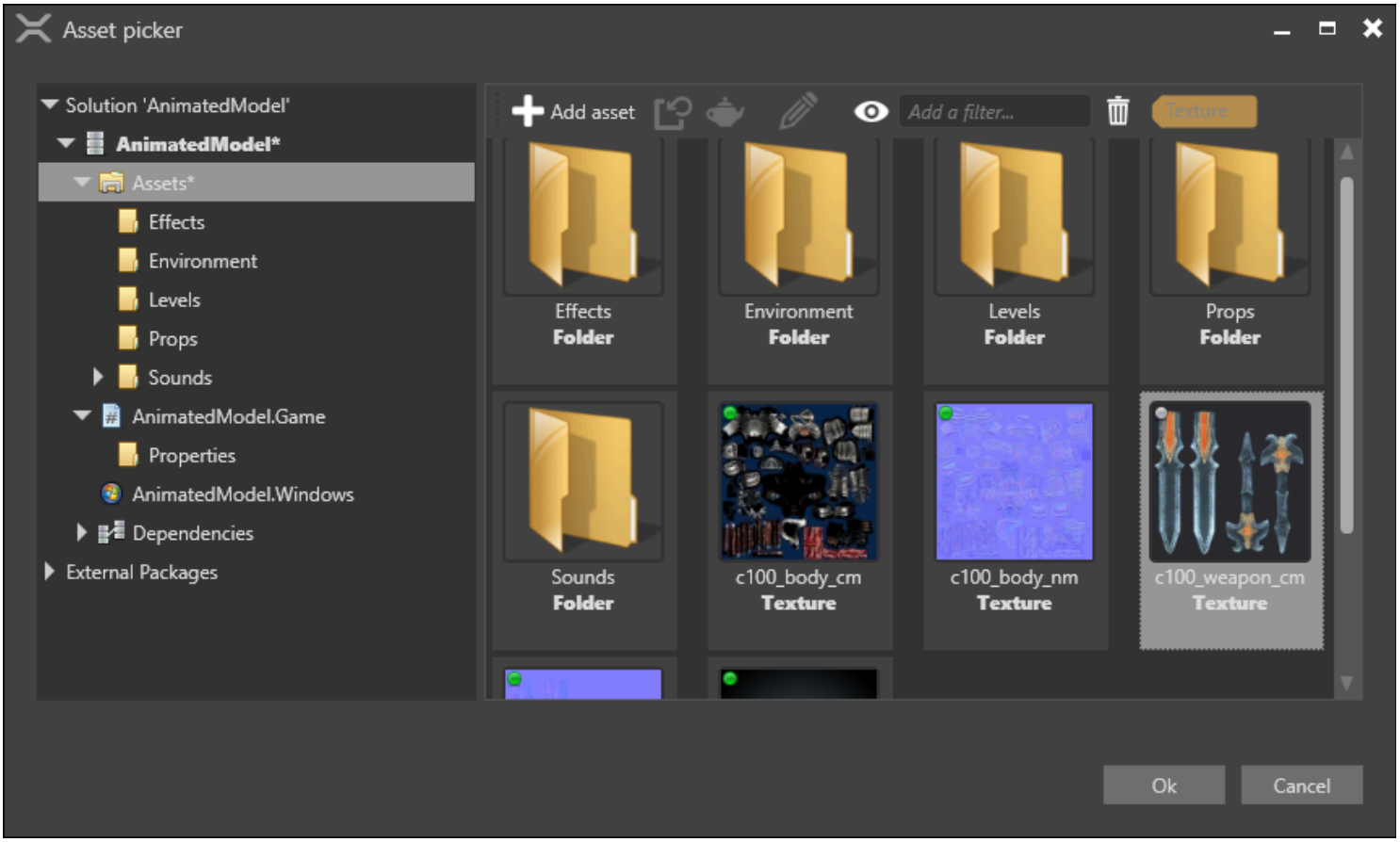
Components that use assets have **asset docks** in the **property grid**.



To add an asset to an entity component, drag the asset to the asset dock in the component properties (in the **property grid**). You can drop assets in the text field or the empty thumbnail.



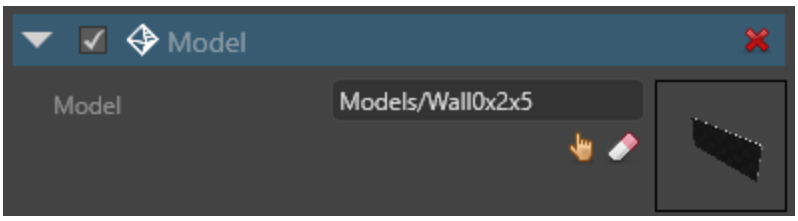
Alternatively, click (**Select an asset**).



The **Select an asset** window opens.

> **NOTE**
>
> The **Select an asset** window only displays assets of types expected by the component. For example, if the component is an audio listener, the window only displays audio assets.

After you add an asset to a component, the asset dock displays its name and a thumbnail image.



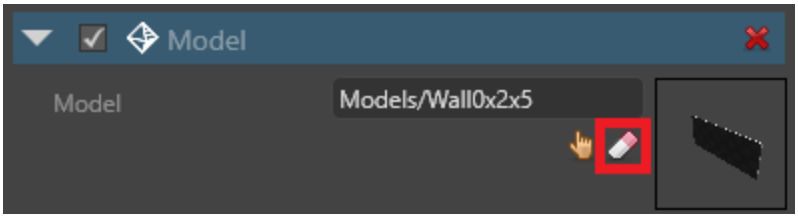
Reference assets in other assets

Assets can reference other assets. For example, a model asset might use material assets.

You can add asset references to assets the same way you add them to entity components (see above).

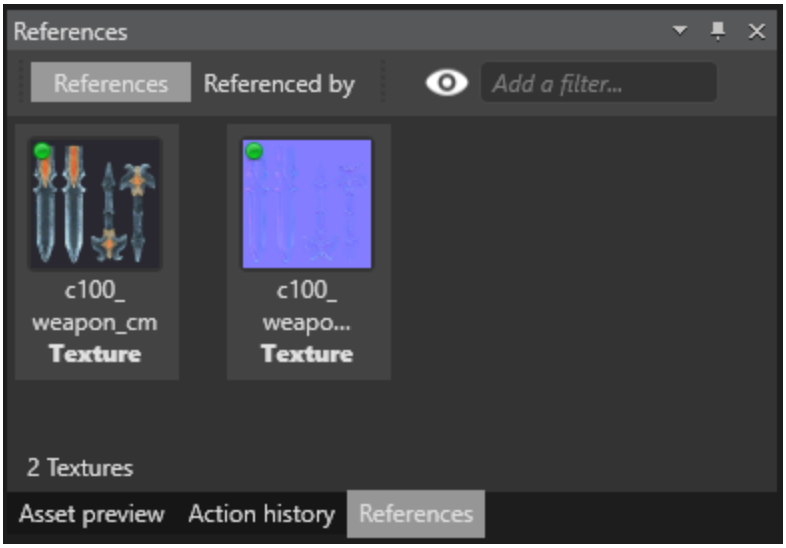
Clear a reference

To clear a reference to an asset, in the **asset dock**, click (**Clear reference**).



Examine references

You can see the references in a selected asset in the **References** tab. By default, this is in the bottom right of Game Studio.



- The **References** tab displays the assets referenced by the selected asset.
- The **Referenced by** tab displays the assets that reference the selected asset.

> **TIP**
>
> If you can't see the References tab, make sure it's displayed under **View > References**.

Load assets from code

When loading in assets at runtime we speak of "Content" rather than assets. The loaded content refers to the asset and can then be used in your script.

```
// Load a model (replace URL with valid URL)
var model = Content.Load<Model>("AssetFolder/MyModel");
```

```
// Create a new entity to add to the scene
Entity entity = new Entity(position, "Entity Added by Script") { new ModelComponent { Model
= model } };

// Add a new entity to the scene
SceneSystem.SceneInstance.RootScene.Entities.Add(entity);
```

> **TIP**
>
> To find the asset URL, in Game Studio, move the mouse over the asset. Game Studio displays the asset URL in a tooltip. URLs typically have the format *AssetFolder/AssetName*.

> **WARNING**
>
> When loading assets from scripts, make sure you:
>
> - include the asset in the build as described in Manage assets
> - make sure you add the script as a component to an entity in the scene

Unload unneeded assets

When loading content from code, you should unload content when you don't need them any more. If you don't, content stays in memory, wasting GPU.

To unload an asset, use `Content.Unload(myAsset)`.

Load assets from code using UrlReference

`UrlReference` allows you to reference assets in your scripts the same way you would with normal assets but they are loaded dynamically in code. Referencing an asset with a `UrlReference` causes the asset to be included in the build.

You can reference assets in your scripts using properties/fields of type `UrlReference` or `UrlReference<T>`:

- `UrlReference` can be used to reference any asset. This is most useful for the "Raw asset".
- `UrlReference<T>` can be used to specify the desired type. i.e. `UrlReference<Scene>`. This gives Game Studio a hint about what type of asset this `UrlReference` can be used for.

Examples

Loading a Scene

Using `UrlReference<Scene>` to load the next scene.

```
using System.Threading.Tasks;
//Include the Stride.Core.Serialization namespace to use UrlReference
using Stride.Core.Serialization;
using Stride.Engine;

namespace Examples
{
    public class UrlReferenceExample : AsyncScript
    {
        public UrlReference<Scene> NextSceneUrl { get; set; }

        public override async Task Execute()
        {
            //...
        }

        private async Task LoadNextScene()
        {
            //Dynamically load next scene asynchronously
            var nextScene = await Content.LoadAsync(NextSceneUrl);
            SceneSystem.SceneInstance.RootScene = nextScene;
        }
    }
}
```

Load data from a Raw asset JSON file

Use a Raw asset to store data in a JSON file and load using Newtonsoft.Json. To use `Newtonsoft.Json` you also need to add the `Newtonsoft.Json` NuGet package to the project.

```
//Include the Newtonsoft.Json namespace.
using Newtonsoft.Json;
using System.IO;
using System.Threading.Tasks;
//Include the Stride.Core.Serialization namespace to use UrlReference
using Stride.Core.Serialization;
using Stride.Engine;

namespace Examples
{
    public class UrlReferenceExample : AsyncScript
    {
        public UrlReference RawAssetUrl { get; set; }
```

```
        public override async Task Execute()
        {
            //...
        }

        private async Task<MyDataClass> LoadMyData()
        {
            //Open a StreamReader to read the content
            using (var stream = Content.OpenAsStream(RawAssetUrl))
            using (var streamReader = new StreamReader(stream))
            {
                //read the raw asset content
                string json = await streamReader.ReadToEndAsync();
                //Deserialize the JSON to your custom MyDataClass Type.
                return JsonConvert.DeserializeObject<MyDataClass>(json);
            }
        }
    }
}
```

See also

- Create assets
- Manage assets

Scenes

Beginner Level designer

Scenes are the levels in your game. A scene is composed of **entities**, the objects in your project.

The screenshot below shows a scene with a knight, a light, a background, and a camera entity:

Scenes are a type of asset. As they are complex assets, they have a dedicated editor, the **Scene Editor**.

In this section

- Create and open a scene
- Navigate in the Scene Editor
- Manage scenes
- Load scenes
- Add entities
- Manage entities

Create and open a scene

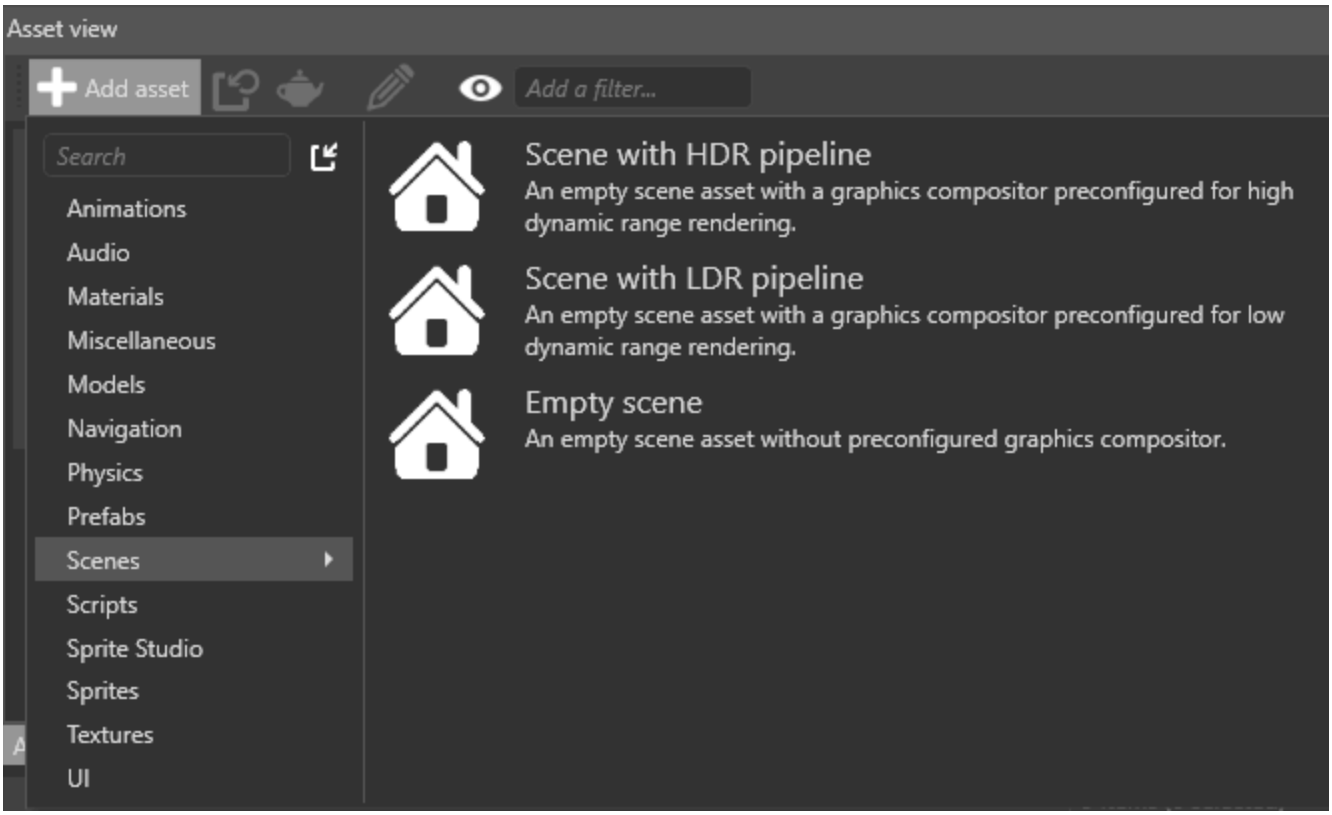
Beginner Level Designer

When you create a new project, Game Studio creates an initial scene and populates it with basic entities such as a light, a camera, and a skybox.

You can create scenes like any other asset. As they are complex assets, they have a dedicated editor, the **Scene Editor**.

Create a scene

1. In the **Asset View** (by default in the bottom pane), click **Add asset** and select **Scenes**.

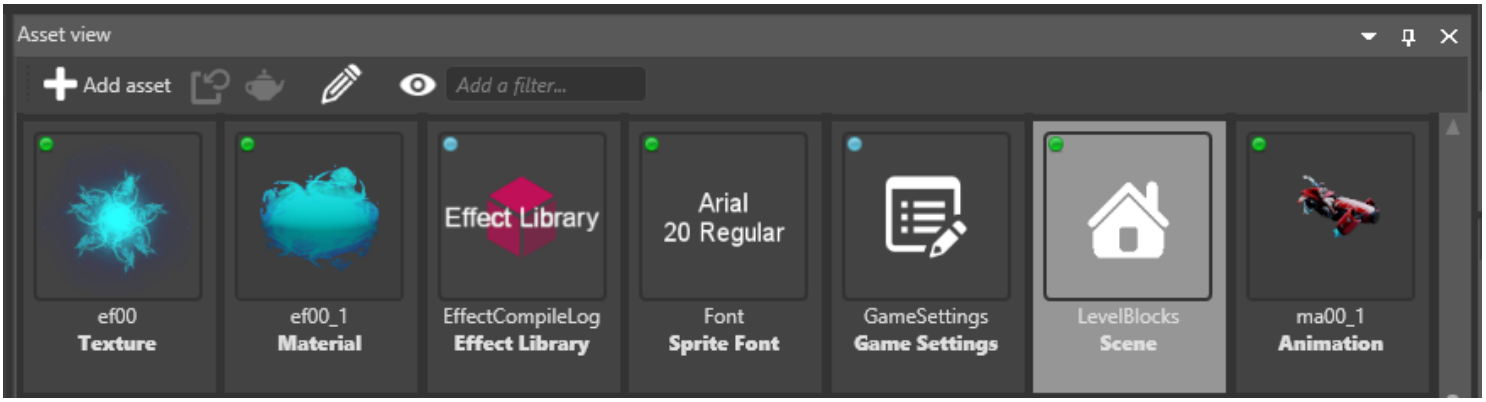


2. Select the appropriate **scene template**.

Template	Result
Empty scene	An empty scene with no entities or preconfigured rendering pipeline
Scene with HDR pipeline	A scene containing basic entities and preconfigured for HDR rendering
Scene with LDR pipeline	A scene containing basic entities and preconfigured for LDR rendering

Open a scene in the Scene Editor

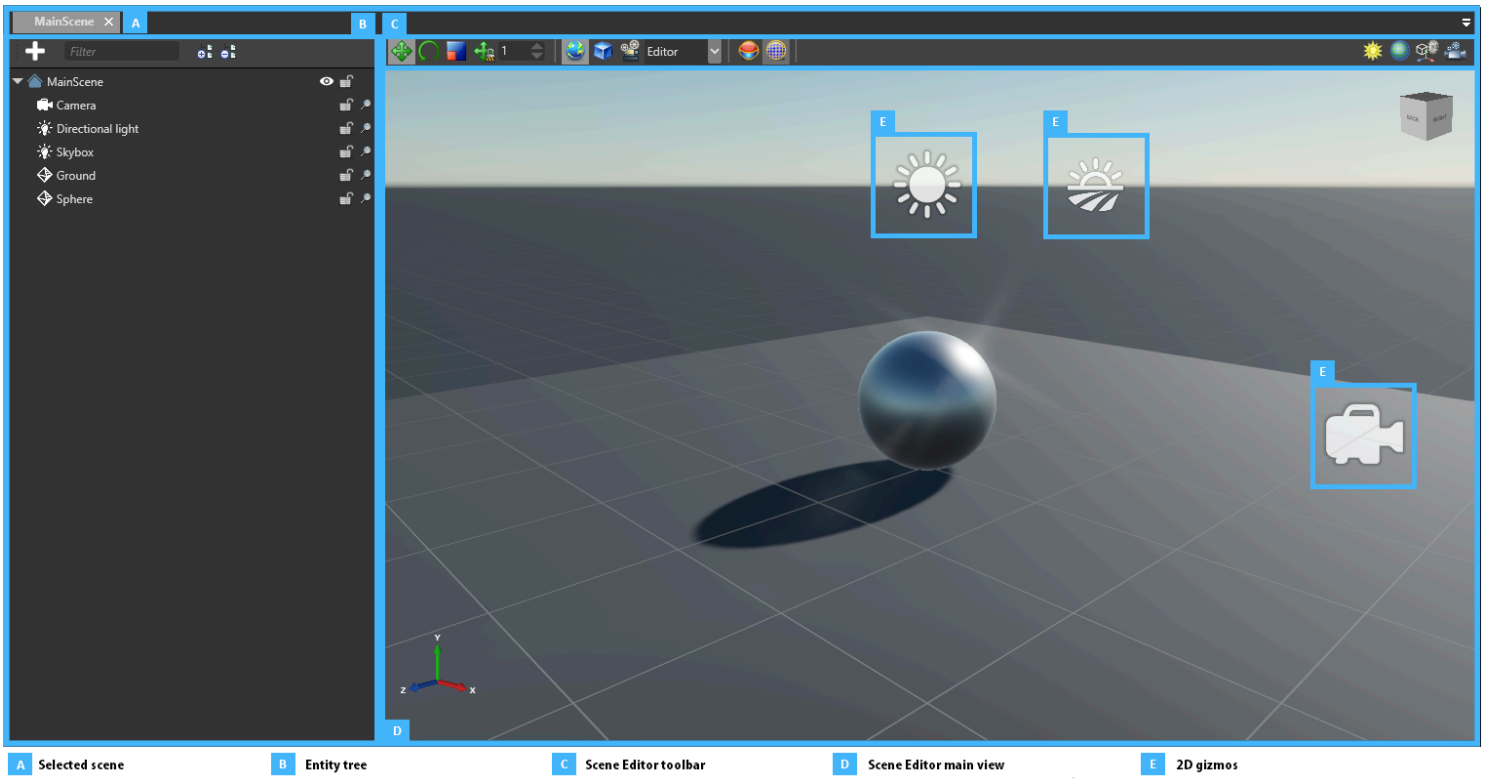
In the **Asset View**:



- double-click the scene asset, or
- right-click the asset and select **Edit asset**, or
- select the asset and type **Ctrl + Enter**

> **TIP**
>
> You can have several scenes open simultaneously.

Use the Scene Editor



The **Scene Editor tabs** (A) display the open scenes. You can switch between open scenes using the tabs.

The **Entity Tree** (B) shows the hierarchy of the entities included in the scene. The same entity hierarchy is applied at runtime. You can use the Entity Tree to browse, select, rename, and reorganize your entities.

You can use the **tool bar** (C) to modify entities and change the Scene Editor display.

The **main window** (D) shows a simplified representation of your scene, with your entities positioned inside it. For entities that have no shape (E), Game Studio represents them with **2D gizmos**; for example, cameras are represented with camera icons.

See also

- Navigate in the Scene Editor
- Manage scenes
- Load scenes
- Add entities
- Manage entities

Add entities

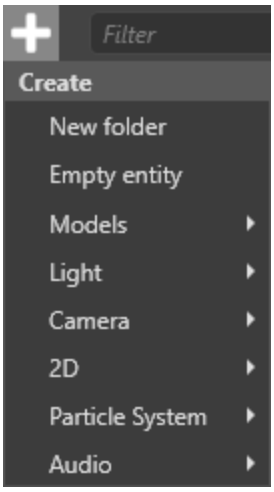
Beginner Level Designer

After you create a scene, you need to add entities to your scene to build your level.

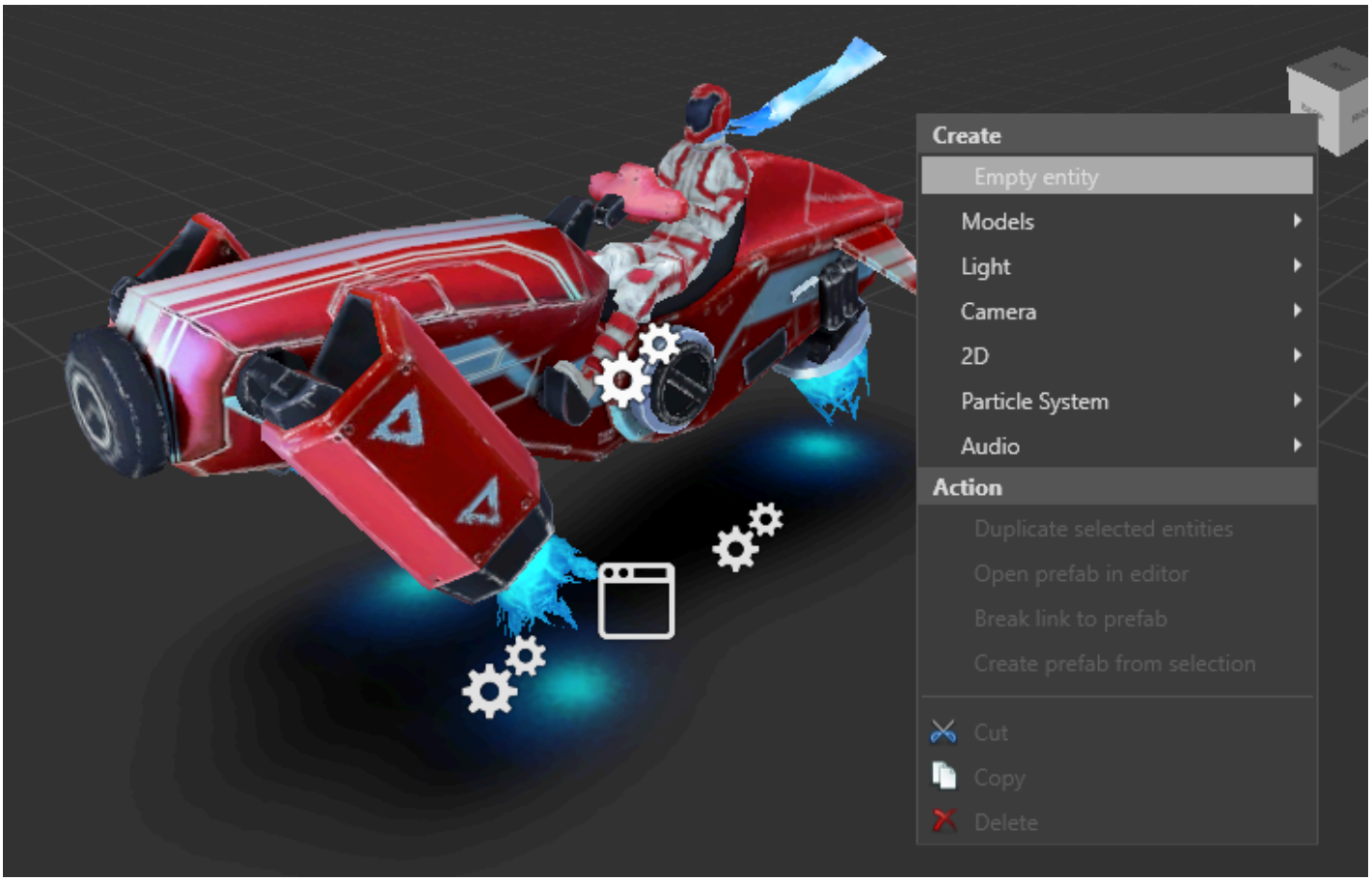
Create an entity from the Scene Editor

1. Above the **Entity Tree**, click the icon.

   The **Create** menu opens:

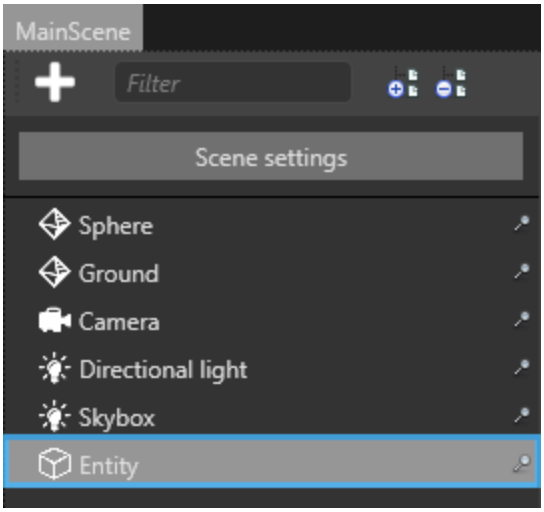
   

   Alternatively, right-click the **Entity Tree** or anywhere in the **scene**. If you create an entity in the scene, Game Studio adds an entity in the location you click.

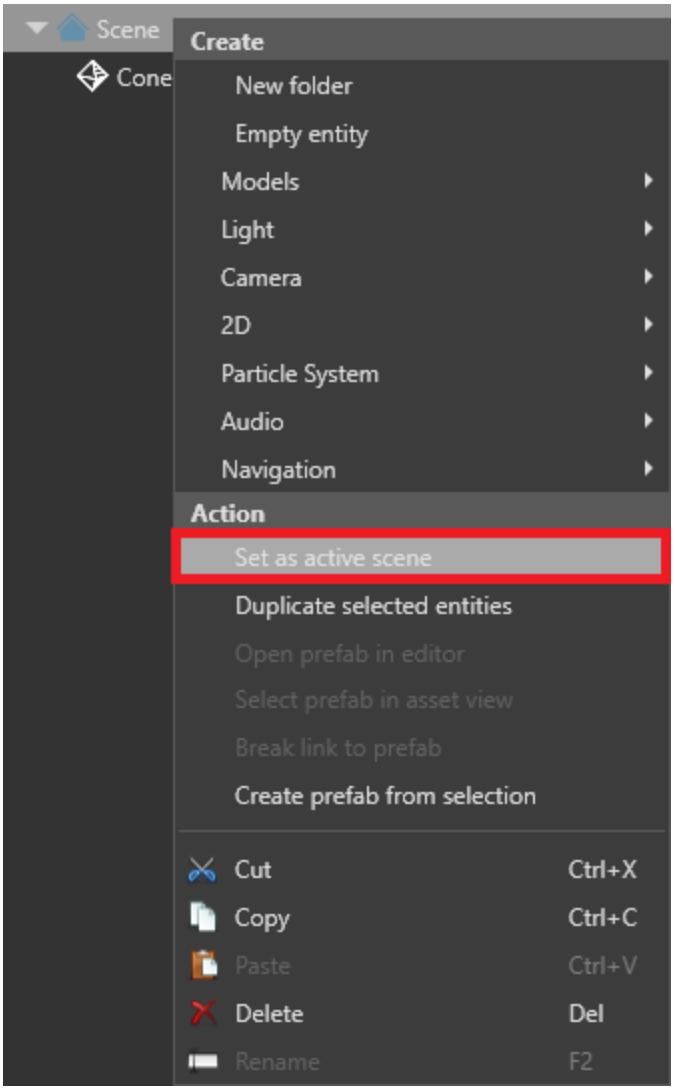


2. Select **Empty entity**, or select an entity template.

   Game Studio adds an entity to the active scene and displays it in the Entity Tree:



> **TIP**
>
> The **active scene** is the scene entities are added to. To set the active scene, **Entity Tree** (left by default), right-click the scene and select **active scene**.



The active scene has no effect on runtime.

Create an entity from an asset

You can add an entity by dragging and dropping an asset from the **Asset View** to the scene.

0:00

Game Studio automatically creates an entity and adds the required component and reference based on the asset you used. For example, if you drag a model asset to the scene, Game Studio creates an entity with a model component with the model asset as its reference.

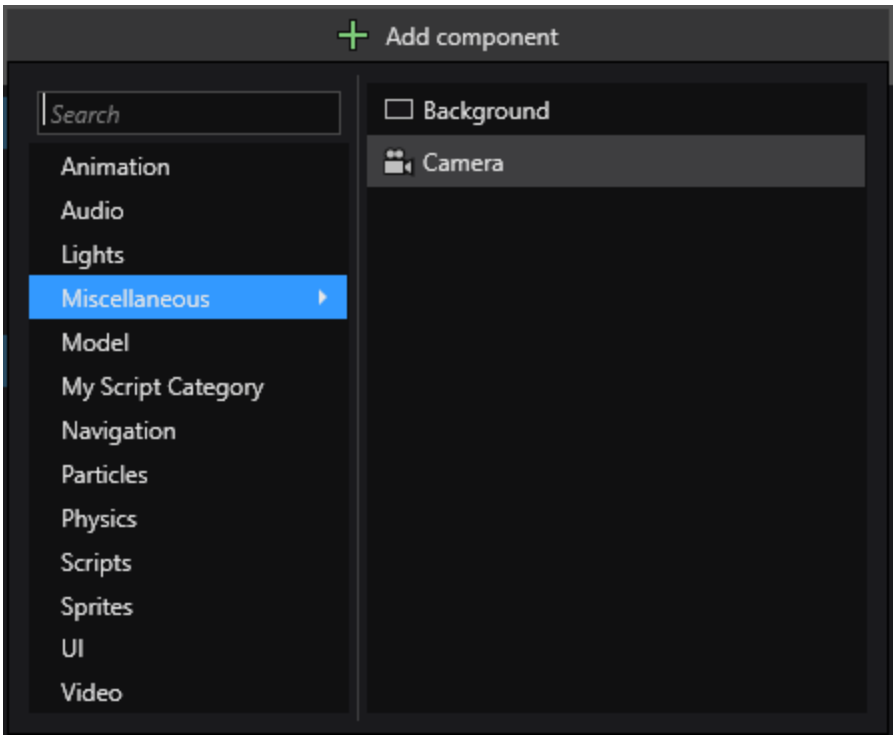
> **NOTE**
> You can only create entities by dragging assets with corresponding components. For example, model components use model assets, so can be dragged; animations have no corresponding component, so can't be dragged.

Set up a component

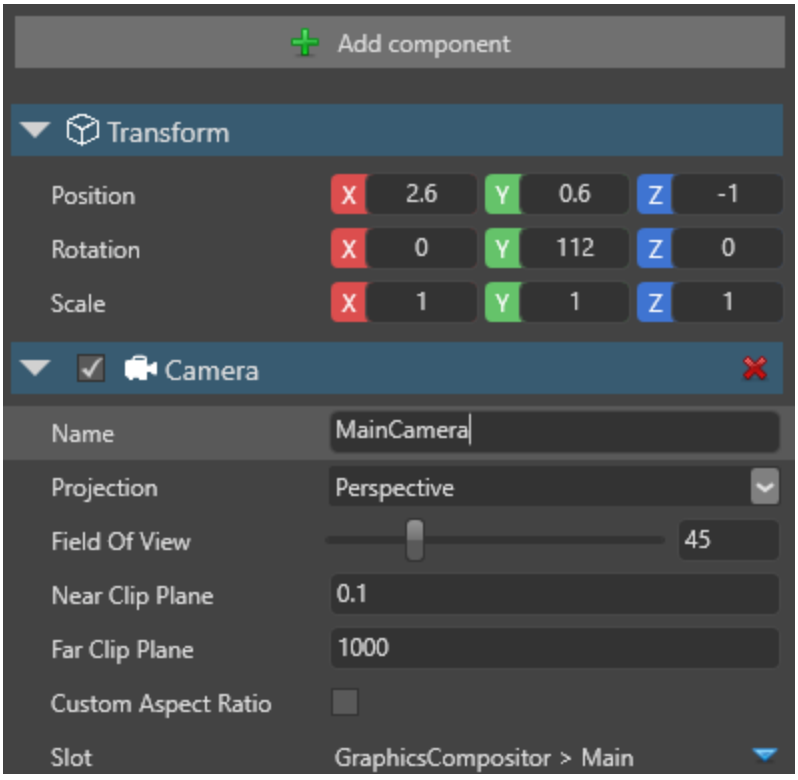
Components add special properties to entities that define their purpose in your project. For example, you add lights to your scene by adding Light components to entities, add models by adding model components, and so on. An entity with no component has no purpose.

To add a component to an entity:

1. Select the entity.
2. In the Property Grid, click **Add component**, and add component you want.



Game Studio adds the component.



3. **Set the properties** of your new component.

Duplicate an entity

You can duplicate an entity along with all its properties. Duplicating an entity and then modifying the properties of the new entity is often faster than creating an entity from scratch.

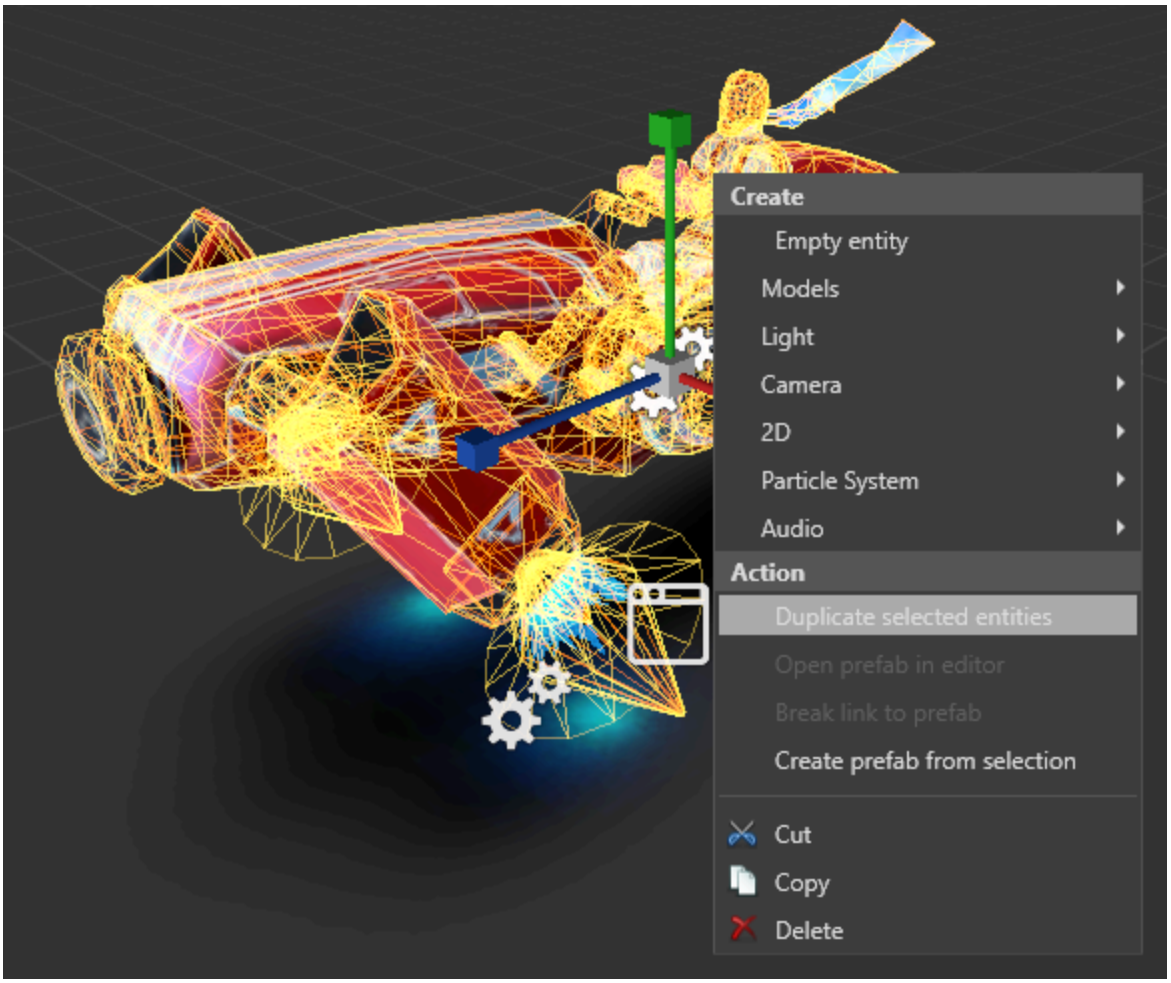
1. Select the entity you want to duplicate.

2. Hold **Ctrl** and move the entity with the mouse.

   The entity and all its properties are duplicated.

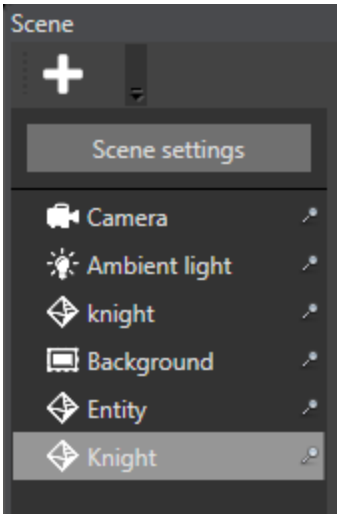
0:00

Alternatively, right-click the entity and select **Duplicate selected entities**.



Rename an entity

1. Select the entity and press **F2**.
2. Type a name for the entity, and then press **Enter**.



See also

- Manage scenes

Manage entities

Beginner Level designer

To build the levels of your game, you need to translate (move), rotate, and resize entities in your scene. These are known as **transformations**.

Transformation gizmos

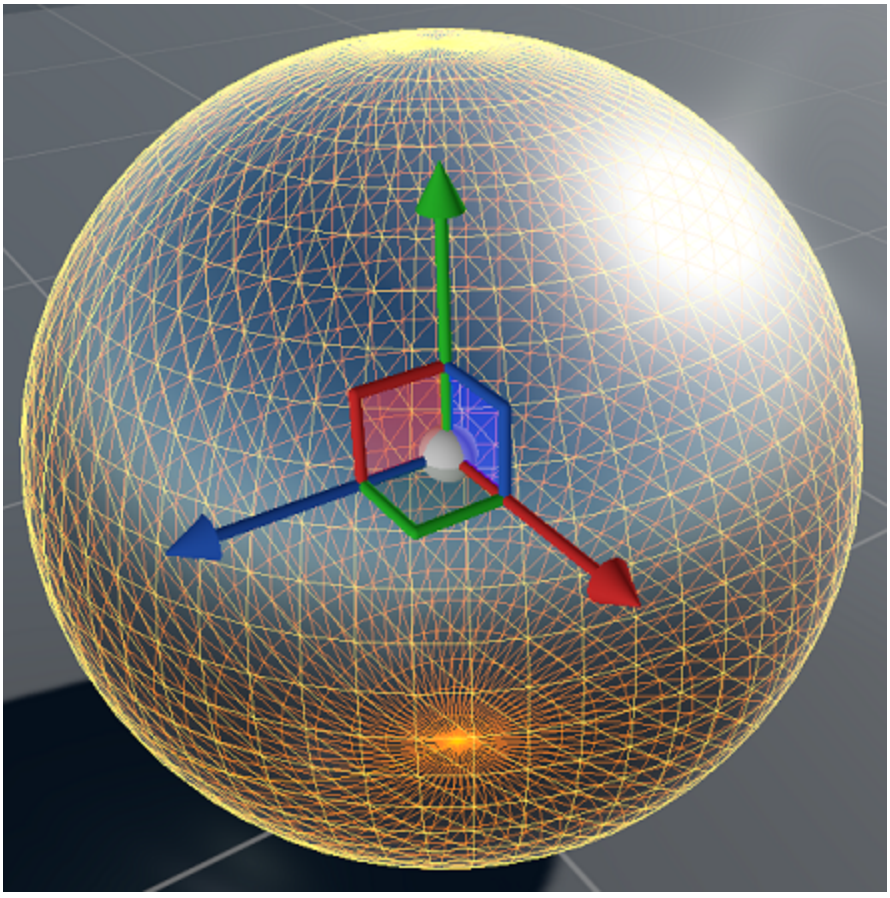
You can select the transformation gizmos from **Scene Editor toolbar**.

Alternatively, press **Space** to switch between gizmos.

There are three types of transformation gizmo:

- The **translation gizmo** moves entities
- The **rotation gizmo** rotates entities
- The **scale gizmo** resizes entities

Game Studio displays the selected transformation gizmo at the origin of the entity.



Translation gizmo

To select the translation gizmo, click the icon in the Scene Editor toolbar or press **W**.

The translation gizmo moves (**translates**) entities in the scene along the axis you select.

- To move an entity along the X axis, drag it by the **red** arrow.
- To move an entity along the Y axis (up and down), drag it by the **green** arrow.
- To move the entity along the Z axis, drag it by the **blue** arrow.
- To move the entity in free 3D, drag it by the central sphere.

0:00

Rotation gizmo

To select the rotation gizmo, click the icon in the Scene Editor toolbar or press **E**.

The rotation gizmo rotates entities in the scene along the axis you select.

- To rotate an entity along the X axis (pitch), drag it by the **red** ring.
- To rotate an entity along the Y axis (yaw), drag it by the **green** ring.
- To rotate the entity along the Z axis (roll), drag it by the **blue** ring.

0:00

Scale gizmo

To select the scale gizmo, click the icon in the Scene Editor toolbar or press **R**.

The scale gizmo resizes entities along a single axis ("stretching" or "squashing" them) or all axes (making them larger or smaller without changing their proportions).

- To resize an entity along the X axis, drag it by the **red** ring.
- To resize an entity along the Y axis, drag it by the **green** ring.
- To resize the entity along the Z axis, drag it by the **blue** ring.
- To resize the entity in all axes, drag it by the central sphere.

0:00

> **NOTE**
>
> The scale gizmo only works with the **local** coordinate system (see below). When you select the scale gizmo, Game Studio switches to local coordinates.

Change gizmo coordinate system

You can change how the gizmo coordinates work.

1. Select the entity whose gizmo coordinates you want to change.
2. In the Scene Editor toolbar, select the coordinate system you want.

Coordinate system	**Function**
World coordinates	Uses world coordinates for transformations. The X, Y, and Z axes are the same for every entity.
Local coordinates	Uses local coordinates for transformations. The axes are oriented in the same direction as the selected entity.

Coordinate system	Function
Camera coordinates	Uses the current camera coordinates for transformations. The axes are oriented in the same direction as the editor camera.

Snap transformations to grid

You can "snap" transformations to the grid. This means that the degree of transformation you apply to entities is rounded to the closest multiple of the number you specify. For example, if you set the rotation snap value to 10, entities rotate in multiples of 10 (0, 10, 20, 30, etc).

You can change the snap values for each gizmo in the scene view toolbar. Snap values apply to all entities in the scene. For example:

Icon	Function
	Snap translation to multiple of 1
	Snap rotation to multiple of 22.5
	Snap scale to multiple of 1.1

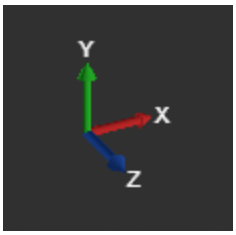
See also

- Create and open a scene
- Navigate in the Scene Editor
- Load scenes
- Add entities

Navigate in the Scene Editor

Beginner Level designer

You can move around the scene and change the perspective of the editor camera. The XYZ axes in the bottom left show your orientation in 3D space.



Move around in the scene

There are several ways to move the editor camera around the Scene Editor.

> **TIP**
> Holding the **Shift** key speeds up movement.

Fly

0:00

Hold the **right mouse button** and **move the mouse** to change the camera direction. Hold the **right mouse button** and use the **WASD keys** to move. This is similar to the controls of many action games.

Pan

Hold the **right mouse button** and the **center mouse button** and move the mouse.

Dolly

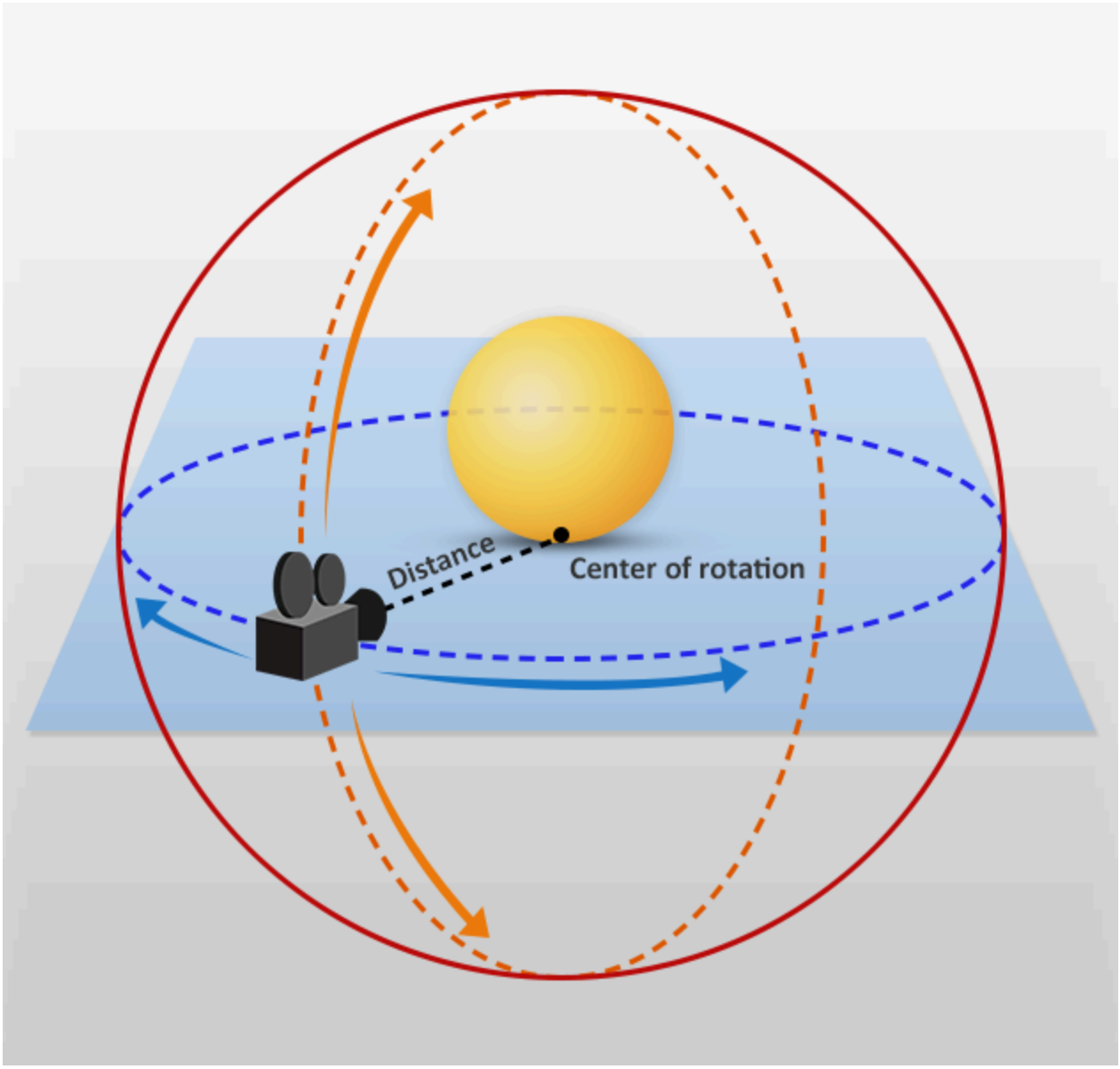
0:00

To dolly (move the camera forward and backward), use the **mouse wheel**.

Orbit

Hold **Alt** and the **left mouse button** and move the **mouse**.

The point of rotation is always the center of the screen. To adjust the distance to the center, use the **mouse wheel**.



0:00

Focus on an entity

0:00

After you select an entity, press the **F** key. This zooms in on the entity and centers it in the camera editor.

You can also focus by clicking the **magnifying glass icon** next to the entity in the Entity Tree.

> **TIP**
> Focusing and then orbiting with **Alt + left mouse button** is useful for inspecting entities.

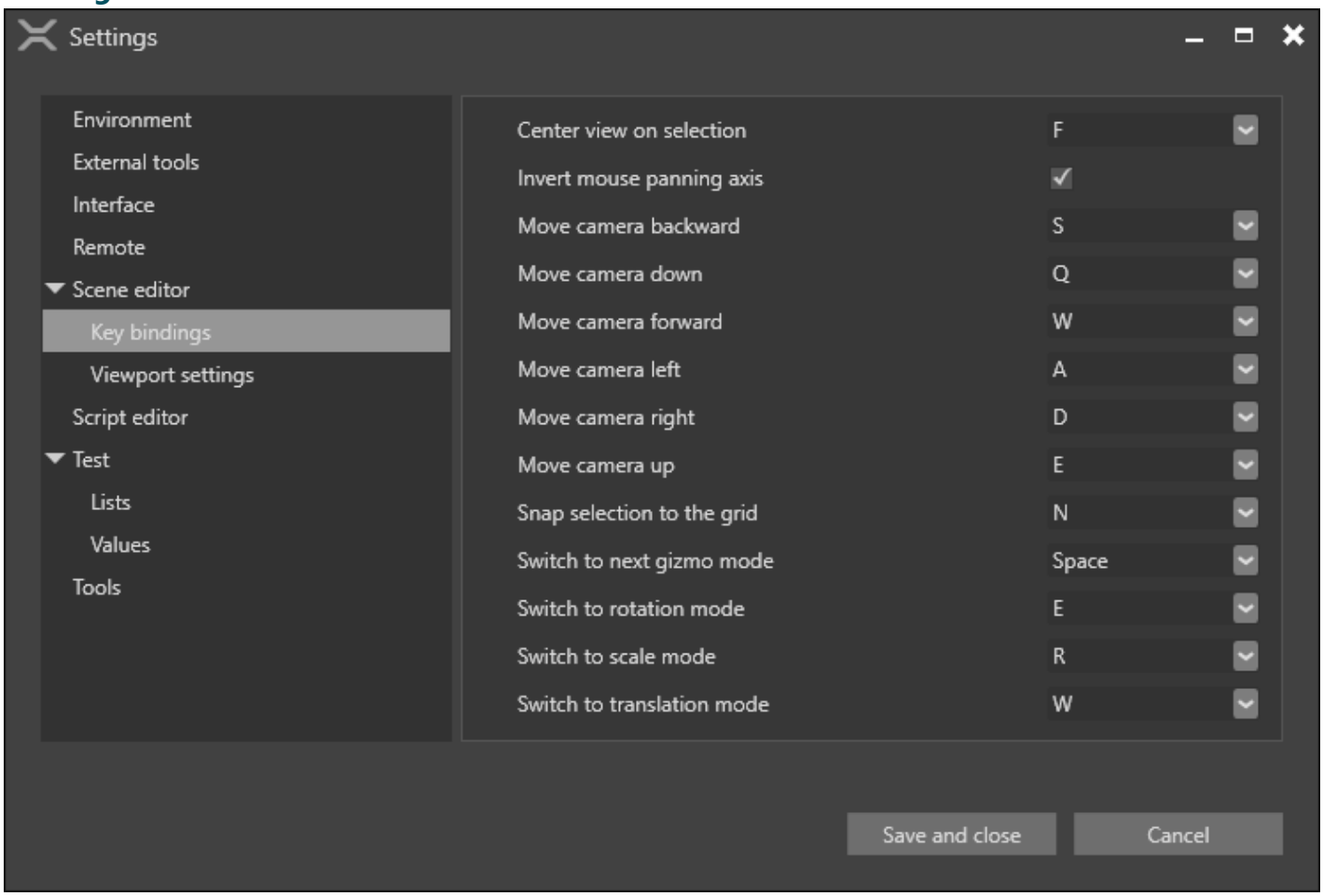
Controls

Action	Control
Move	Arrow keys + right mouse button WASDQE keys + right mouse button
Look around	Hold right mouse button + move mouse
Dolly	Middle mouse button + right mouse button + move mouse
Orbit	Alt key + left mouse button
Zoom	Mouse wheel Alt + Right mouse button + move mouse

Action	Control
Pan	Middle mouse button + move mouse
Focus	F (with entity selected)

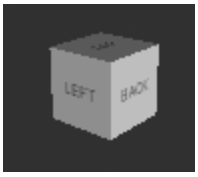
TIP

You can change the scene navigator controls in **Edit > Settings** under **Scene Editor > Key bindings**.



Change camera editor perspective

You can change the camera editor perspective using the **view camera gizmo** in the top-right of the Scene Editor.



Snap camera to position

To change the angle of the editor camera, click the corresponding face, edge, or corner of the **view camera gizmo**.

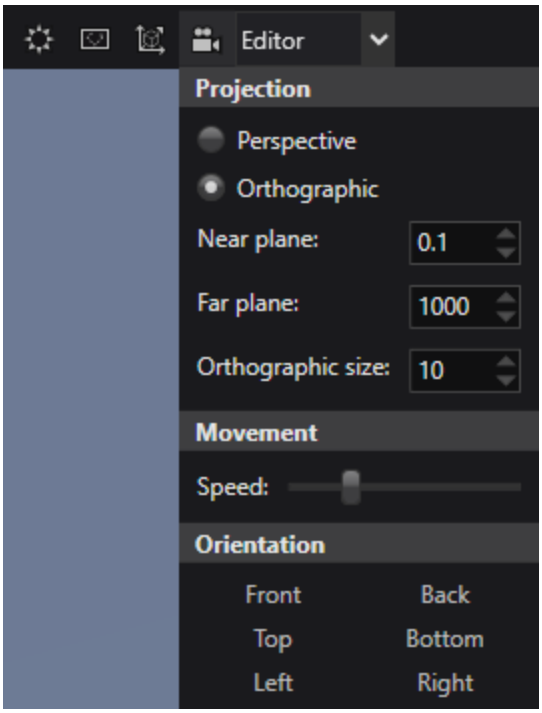
Click	Camera position
Face	Faces the selected face
Edge	Faces the two adjacent faces at a 45° angle
Corner	Faces the three adjacent faces at a 45° angle

0:00

Camera options

> **NOTE**
>
> This page explains how to use the Scene Editor camera. For information about how to use cameras in your game, see Graphics — Cameras.

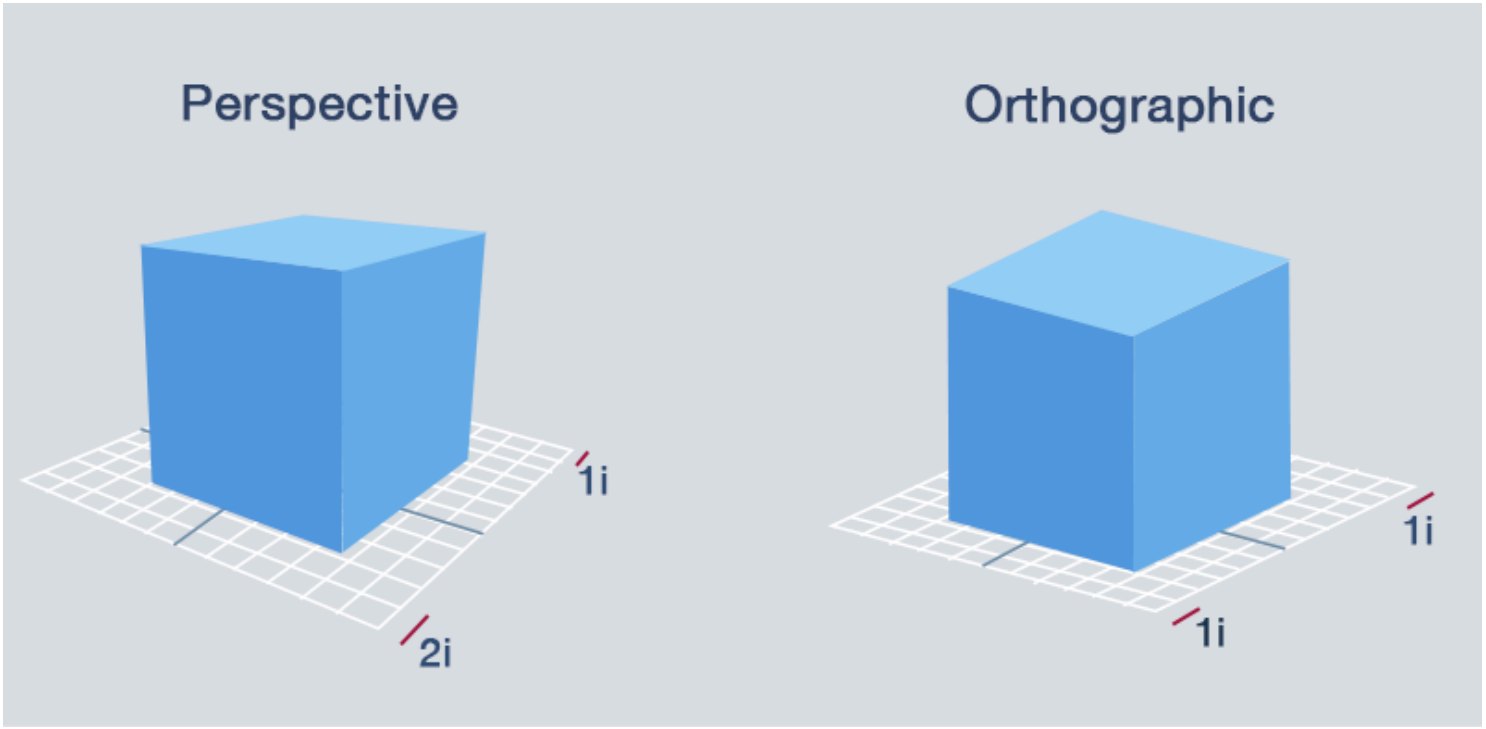
To display the Scene Editor camera options, click the **camera icon** in the top-right of the Scene Editor.



Perspective and orthographic views

Perspective view is a "real-world" perspective of the objects in your scene. In this view, objects close to the camera appear larger, and lines of identical lengths appear different due to foreshortening, as in reality.

In **orthographic view**, objects are always the same size, no matter how far their distance from the camera. Parallel lines never touch, and there's no vanishing point. It's easy to tell if objects are lined up exactly in orthographic view.



Perspective Orthographic

You can also switch between perspective and orthographic views by clicking the **view camera gizmo** as it faces you.

0:00

Field of view

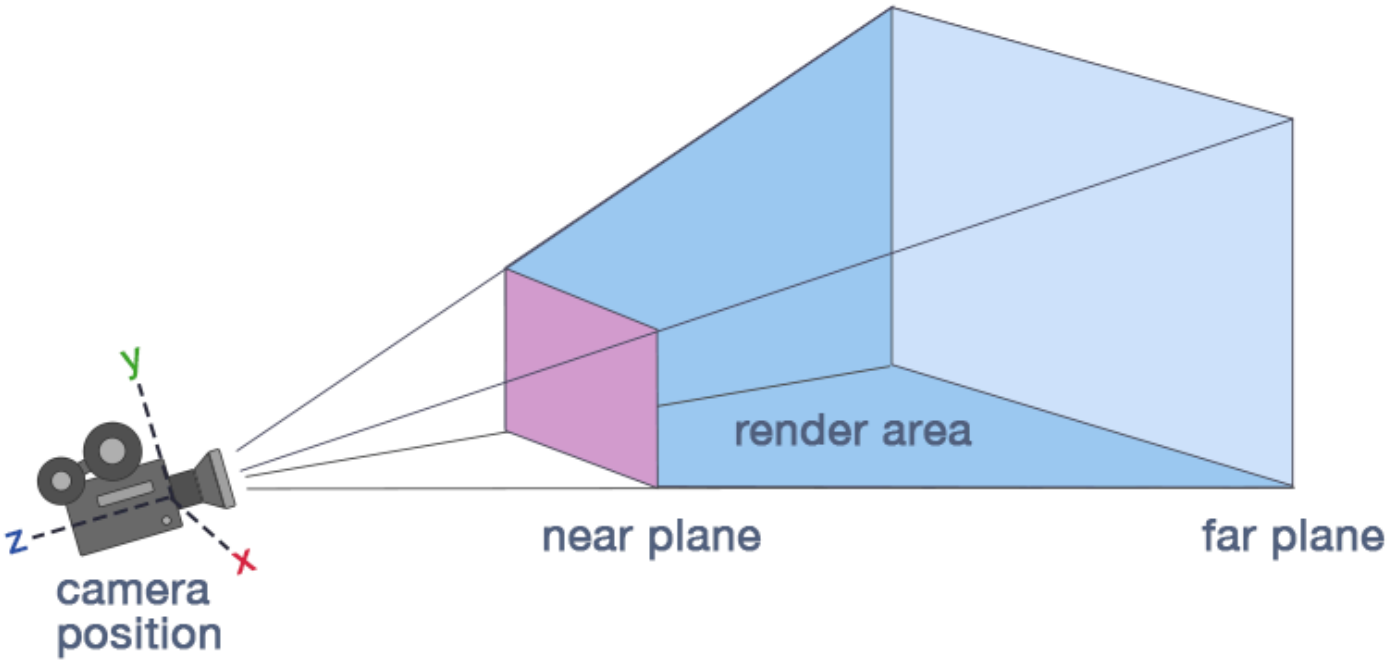
You can change the camera field of view. This changes the camera frustum, and has the effect of zooming in and out of the scene. At high settings (90 and above), the field of view creates stretched "fish-eye lens" views. The default setting is 45.

Near and far planes

The near and far planes determine where the camera's view begins and ends.

- The **near plane** is the closest point the camera can see. The default setting is 0.1. Objects before this point aren't drawn.
- The **far plane**, also known as the draw distance, is the furthest point the camera can see. Objects beyond this point aren't drawn. The default setting is 1000.

Game Studio renders the area between the near and far planes.



Camera speed

The **camera speed** setting changes how quickly the camera moves in the editor.

See also

- Create and open a scene
- Load scenes
- Add entities
- Manage entities

Launch a game

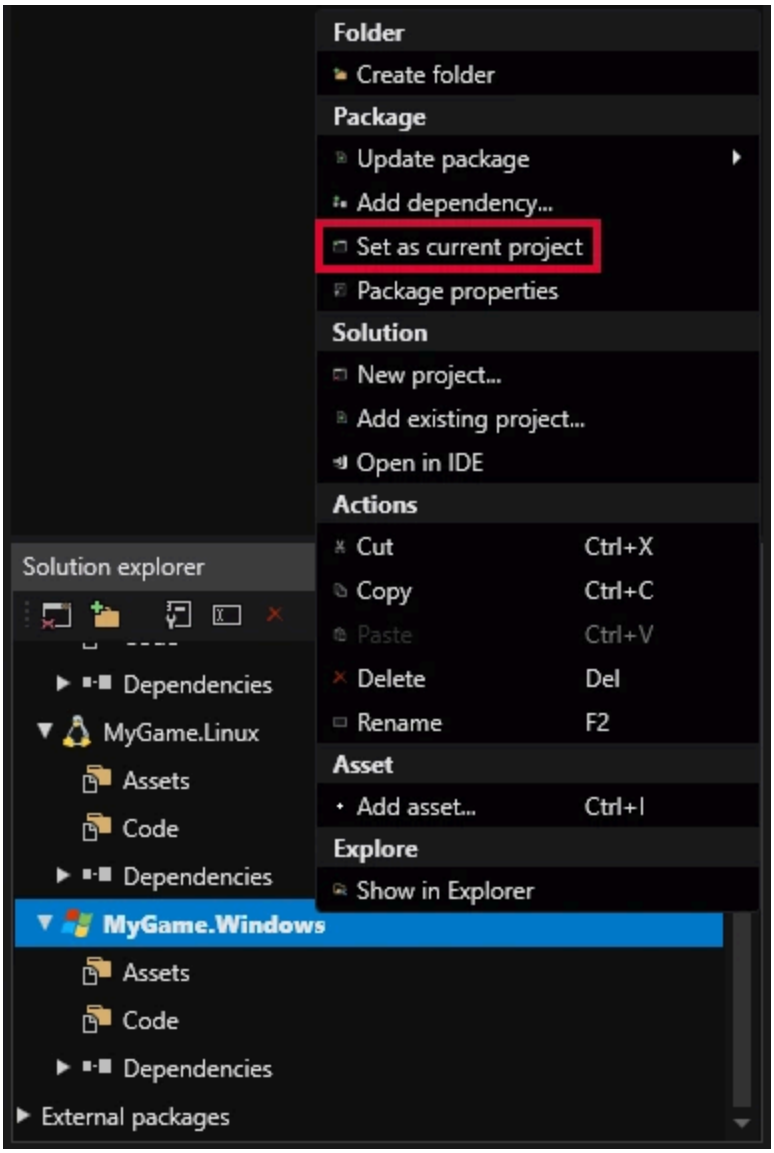
Beginner

This page explains how to launch your game using Game Studio or Visual Studio.

Launch a game from Game Studio

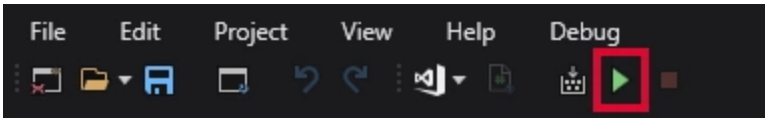
> **NOTE**
> Game Studio can't launch games for the Windows Store or UWP (Universal Windows Platform) platforms. To launch a game for those platforms, use Visual Studio (see below).

1. In the **Solution Explorer**, right-click in your target platform, then choice **Set as current project**.

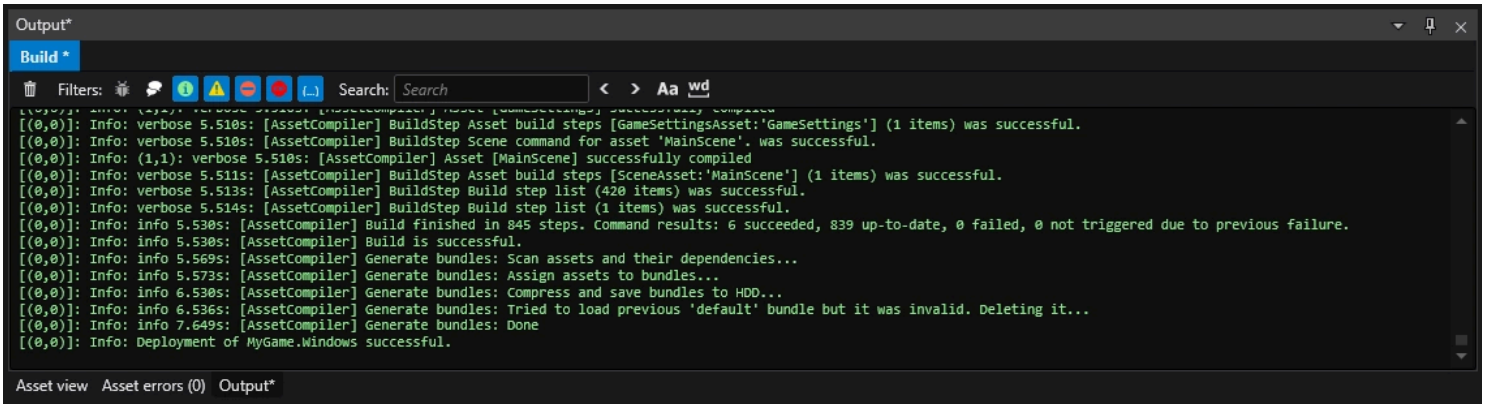


> **NOTE**
>
> You can only select platforms you selected in the **Create a new game** dialog when you created the project. To add additional platforms to the project, see Add or remove a platform.

2. To run the game, click ▶ in the toolbar or press **F5**.



The **Output window** shows the build progress.



When the build is complete, your game starts on the selected platform.

Launch a game from Visual Studio

1. In Game Studio, in the toolbar, click (**Open in IDE**) to launch Visual Studio.

2. In the Visual Studio toolbar, set the appropriate project as the startup project.

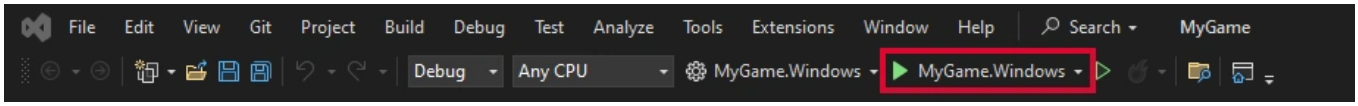
The startup project configuration is updated automatically.

> **TIP**
>
> You can see your projects in the Solution Explorer on the right. The project filename extensions identify the platform (eg *.Android*, *.iOS*, etc).

3. Check that the configuration and platform properly matches what you are expected.

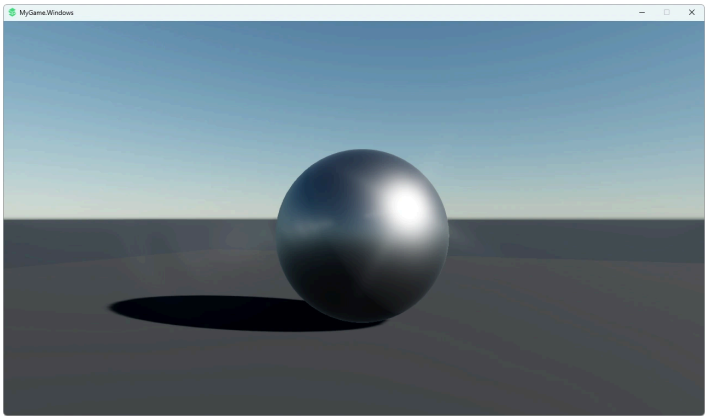
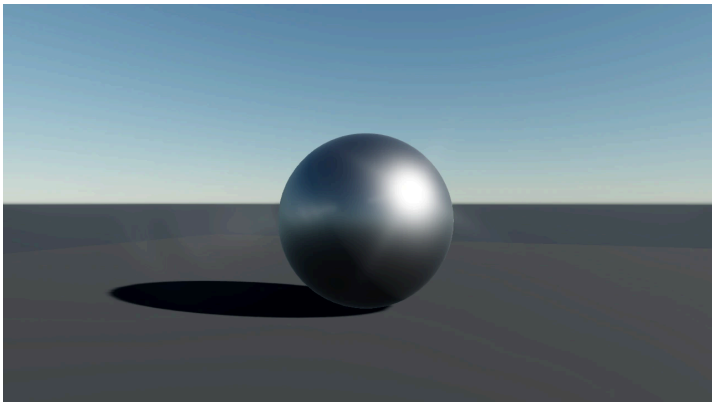
4. ○ To start the game without debugging, press **Ctrl + F5**.

- To start the game with debugging, click **Start** or press **F5**.



Remove borders

By default, the game runs with window borders.

With borders	Without borders
	

To run the game without borders, use:

```
Game.Window.IsBorderLess = true;
```

For example:

```
using Stride.Engine;

namespace MyGame
{
    public class MyScript : StartupScript
    {
        public override void Start()
        {
            base.Start();
            Game.Window.IsBorderLess = true;
        }
    }
}
```

Animation

Designer Programmer

3D models are animated by adding three kinds of asset:

- a skeleton
- a skinned model
- an animation clip

> **NOTE**
> For information about 2D animation, see Sprites.

Skeletons

Skeletons are digital structures that describe deformation patterns of 3D models. Skeletons are made of bones that form a hierarchy. When parent bones change their position, they also affect the positions of child bones. For example, a hand bone might have five child bones (the fingers and thumb); when the hand moves up and down, the fingers and thumb move with it.

Skeletons don't have to resemble the skeletons of real humans or animals. You can make skeletons to animate any 3D model.

> **NOTE**
> There's currently no way to visualize skeletons in Game Studio.

Skinned models

Skinning is the process of assigning weights to vertices and bones they depend on. Each vertex usually depends on one to four bones.

Skinned models are models that have been skinned to match a skeleton. The **skin** describes how vertices of the mesh transform when bones move.

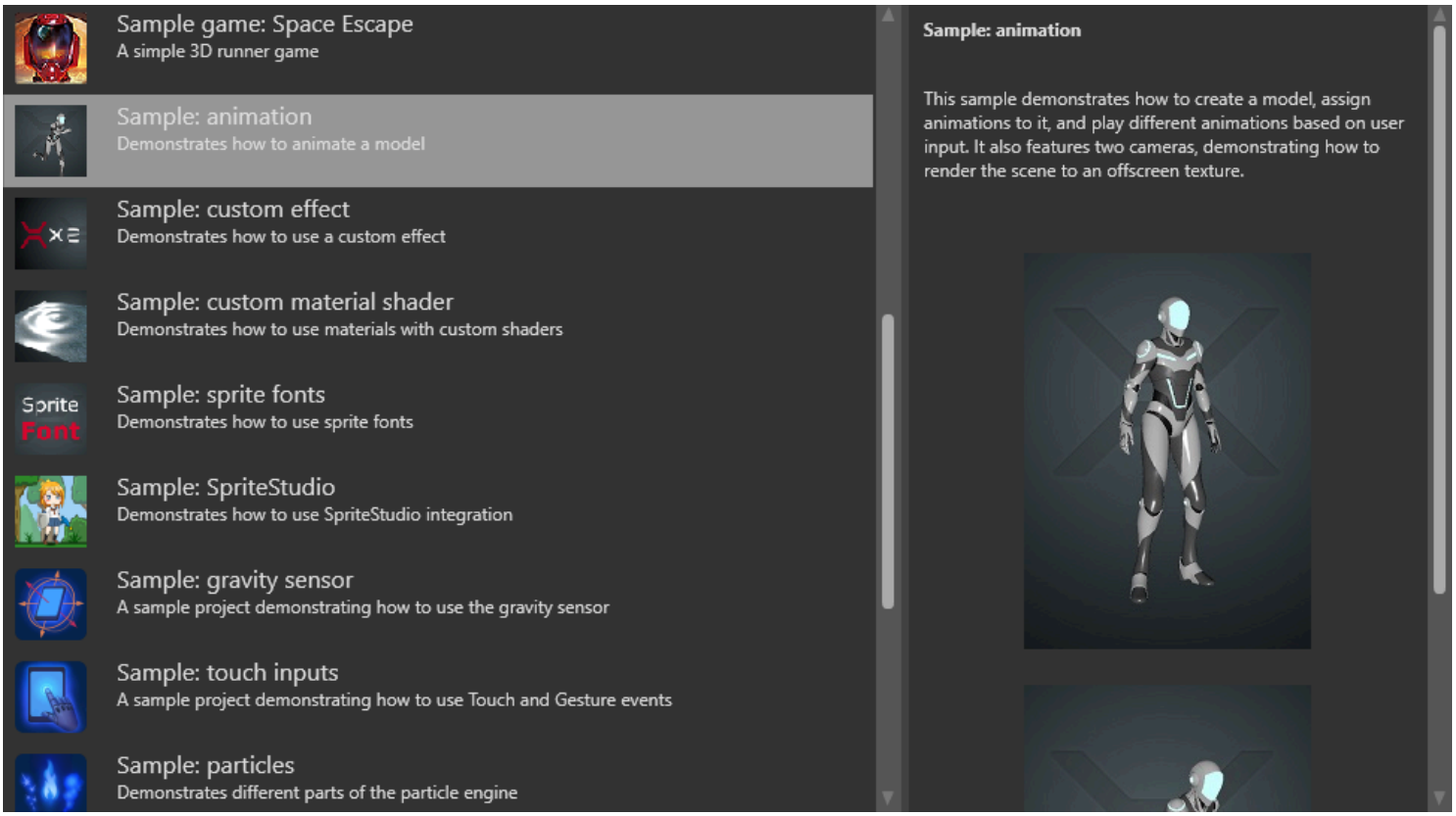
> **NOTE**
>
> In Game Studio, you can only create simple 3D models such as spheres and cubes. For information about how to do this, see Create assets. To create more complex models, use dedicated software like 3DS Max, Maya, or Blender, then import the model into Game Studio.

Animation clips

Animation clips describe the pose of a **skeleton** at a particular moment. The skeleton moves according to the animation. The mesh vertices transform (skin) to match the current pose.

Animation samples

For an example of how animations work in Stride, load the **Sample: animation** sample project.



The templates **First-person shooter**, **Third-person platformer** and **Top-down RPG** also include some advanced animation techniques.

In this section

- Import animations
- Animation properties
- Set up animations

- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links
- Custom attributes

Import animations

Beginner Designer

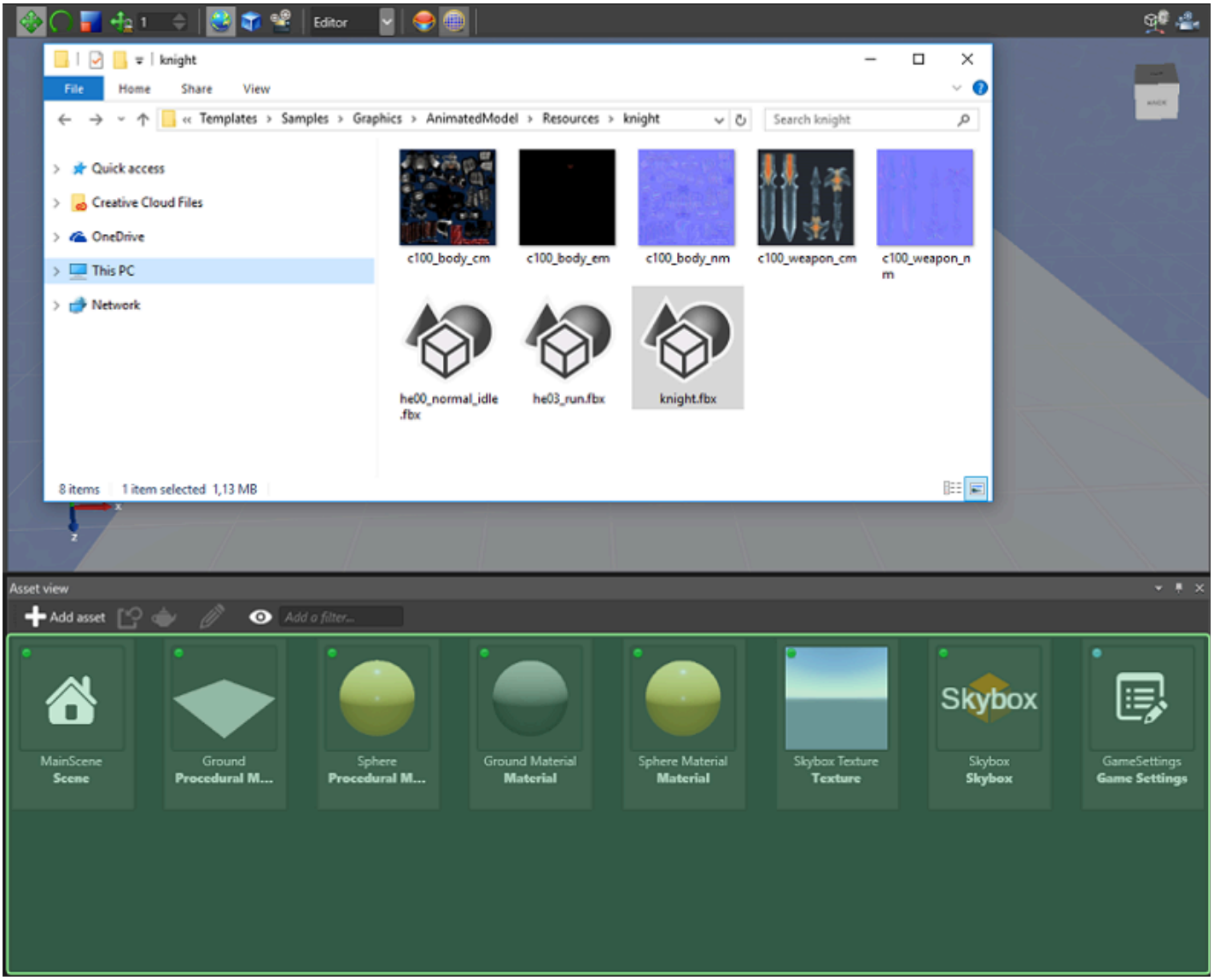
To animate a model, you need to use three kinds of assets together:

- models
- skeletons
- animations

Stride supports the following model file types: `.3ds`, `.blend`, `.dae`,`dxf`, `.fbx`, `.glb`, `.gltf`, `.md2`, `.md3`, `.obj`, `.ply`, `.stl`,`.stp`, `.x`.

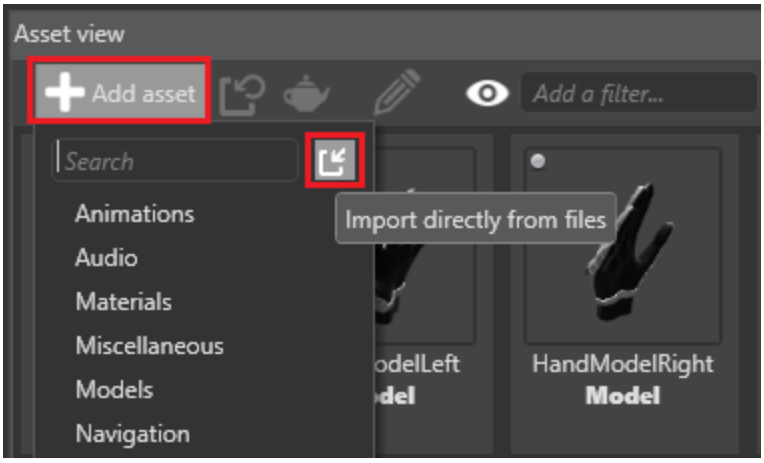
Import a model, skeleton, or animation from a model file

1. Drag the model file from Explorer to the **Asset View** (in the bottom pane by default).



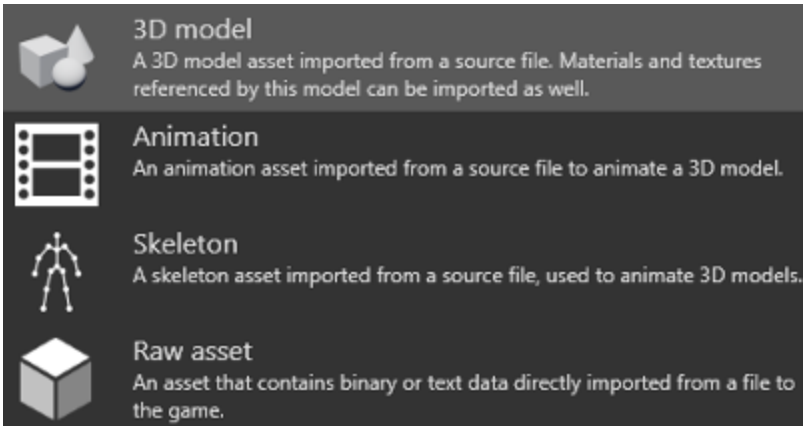
Alternatively, in the **Asset View**:

1a. Click Add asset and select **Import directly from files**.

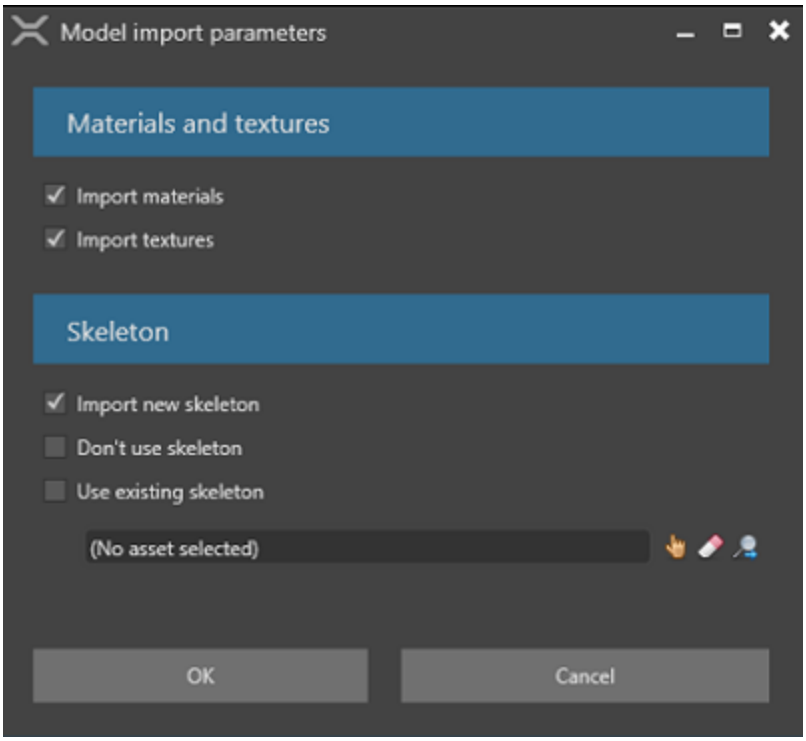


2b. Browse to the file and click **Open**.

2. Specify whether you want to import the **3D model**, **animation**, or **skeleton** from the model file.

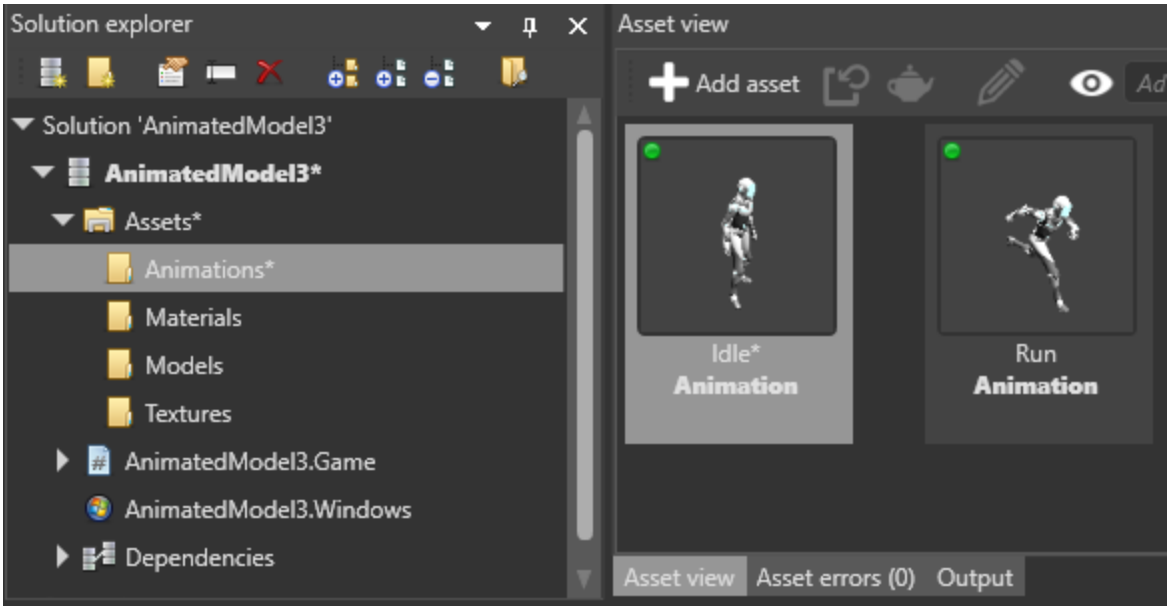


   - If you choose **3D model**, Stride can import any additional materials, textures and skeletons it finds in the model file. You can also import the skeleton from the model (**Import new skeleton**), import no skeleton (**Don't use skeleton**), or specify a different skeleton (**Use existing skeleton**) in the lower field.

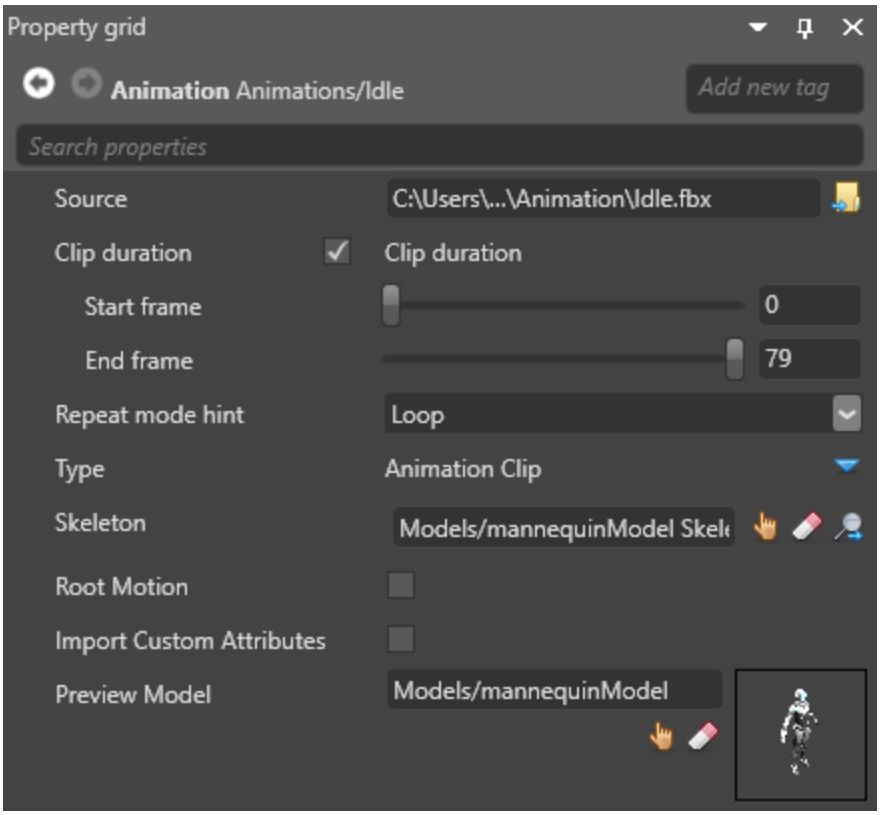


- If you choose **Skeleton**, Stride imports only the skeleton from the model file. You might want to do this, for example, if you want to use it for a new skeleton that uses a subset of its nodes.
- If you choose **Animation**, Stride imports only the animation from the model file. This is sufficient for regular animations; for additive information, there are some extra steps. For details, see Additive animation.

After you import the assets, Game Studio adds them to the **Asset View**.



You can view and edit their properties in the **Property Grid** (on the right by default). For more information, see Animation properties.



Use an animation asset

To use an animation asset, add an AnimationComponent to an entity, then add the animation asset to the animation component. For more information, see Set up animations.

> **NOTE**
>
> Make sure you correctly skin your mesh to the skeleton. If you don't, you won't be able to animate your model correctly.

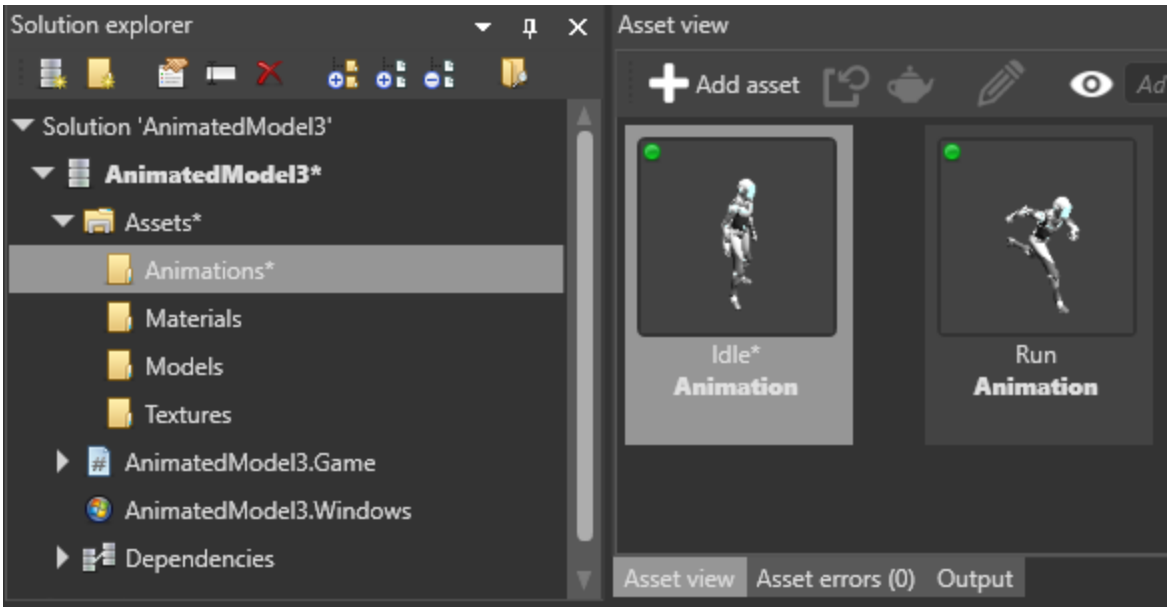
See also

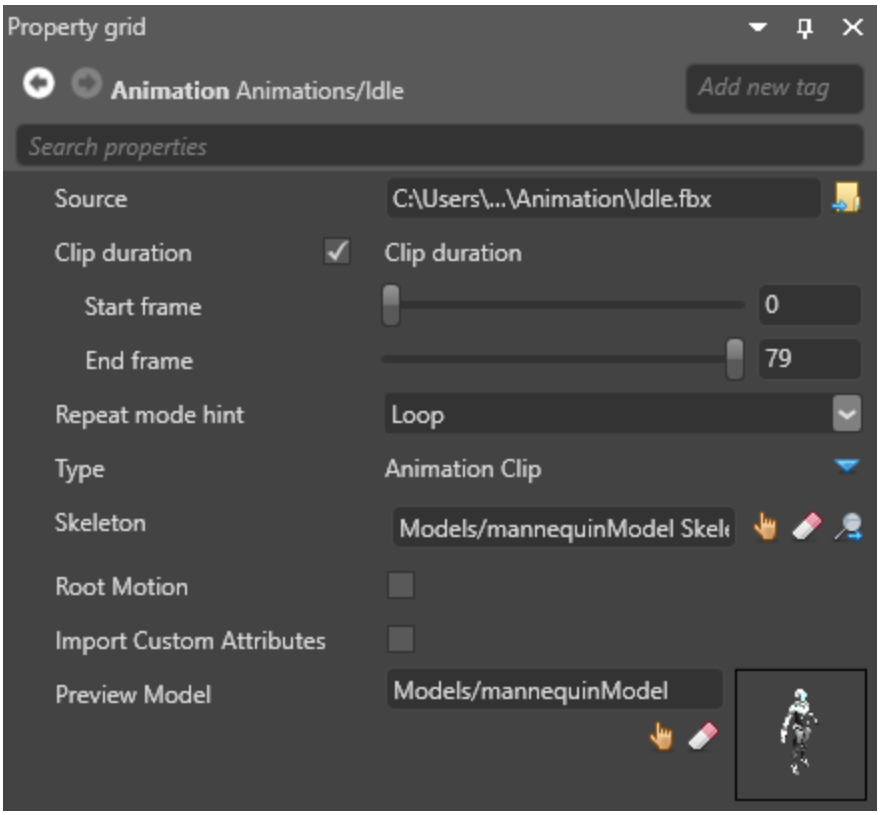
- Animation index
- Animation properties
- Set up animations
- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links
- Custom attributes

Animation properties

Beginner Designer

After you import an animation, you can select it in the **Asset View** (in the bottom pane by default) and view and edit its properties in the **Property Grid** (on the right by default).





Source

The source file used by the animation asset. If you change this, Game Studio re-imports the animation.

Clip duration

By default, clip duration is disabled. This means the animation starts at frame 0 and runs to the last written keyframe in the file.

However, single animation tracks sometimes include several animations. In this case, you have to split the track. To do this, enable **Clip duration** and adjust the **start** and **end** frames to match the duration of each animation.

The start and end frames are still limited by the keyframes exported in the file. For example, if you originally exported frames 20 to 40 from the animation tool, the start frame cannot be lower than 20 and the end frame cannot be higher than 40.

By default, Game Studio assumes the frame rate is 30. You can change this in the **Game settings** asset properties under **Editor settings > Animation frame rate**.

Pivot position

Game Studio assumes the pivot is the origin of the coordinate system local to the animation. It should be set to `(0, 0, 0)`. If your animation was shifted from the origin when exported, you can use this property to re-adjust it.

Scale import

The scale import should be set to `1`. Stride detects the units in which your data was exported and adjusts it automatically. If there are no export settings in your animation file and the scale appears incorrect, you can use the scale import property to re-adjust it.

Repeat mode

You can choose **PlayOnce**, **LoopInfinite** or **PlayOnce&Hold**. This is just a *hint* for the engine. When you assign an animation asset to the model, you can specify differently. If you don't specify the mode later, Stride uses the attribute you set here by default.

Type

Stride supports two types of animation clip. Regular animations default to **Animation clip** and are used with linear blending if mixed. For **Difference clip**, there are few more settings. For more information, see Additive animation.

Skeleton

If you want to animate bones/joints, the animation needs a skeleton.

Skeletons are made of bones that form a hierarchy. When parent bones change their position, they also affect the positions of child bones. For example, a hand bone might have five child bones (the fingers and thumb); when the hand moves up and down, the fingers and thumb move with it.

Make sure you reference the same skeleton used by the model you want to animate. If there are missing bones or other differences between the bone/joint hierarchy of the skeleton in your animation file and the target skeleton, Stride retargets the animation as closely as possible.

> **NOTE**
>
> There's currently no way to visualize skeletons in Game Studio.

Root motion

When root motion is enabled, Stride applies the **root node animation** to the TransformComponent of the entity you add the animation to, instead of applying it to the skeleton.

This is useful, for example, to animate entities that don't require skeletons, such as a spot light moving back and forth.

> **NOTE**
>
> If the animation has no skeleton specified in **Skeleton**, Stride always applies the animation to TransformComponent, even if **root motion** is disabled.

> **NOTE**
>
> The TransformComponent applies an offset to the model node position. If you don't want to add an offset, make sure the TransformComponent is set to `0,0,0`.

Import custom attributes

If you have custom attribute in the animation file...

See also

- Animation index
- Import animations
- Set up animations

- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links

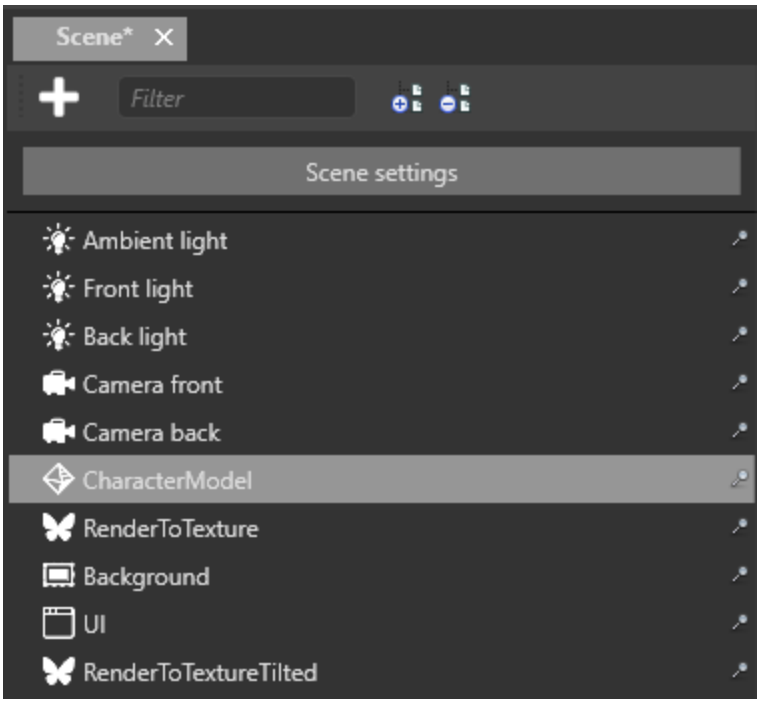
Set up animations

Beginner Designer Programmer

After you import animation assets, you need add them to an entity and play them with a script.

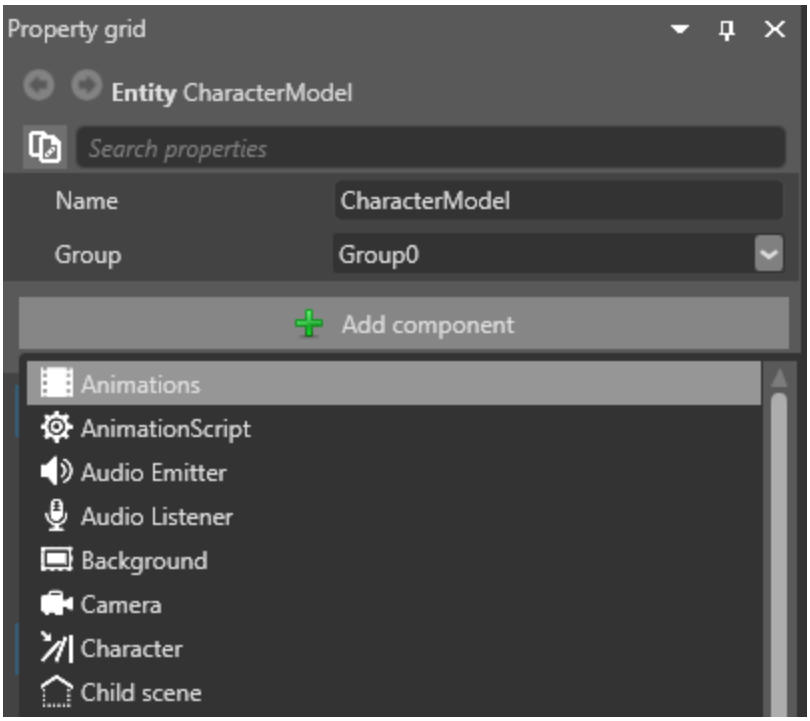
1. Add animation assets to an entity

1. In the **Scene Editor**, select the entity you want to animate.



> **NOTE**
>
> To animate an entity, the entity must have a model component.

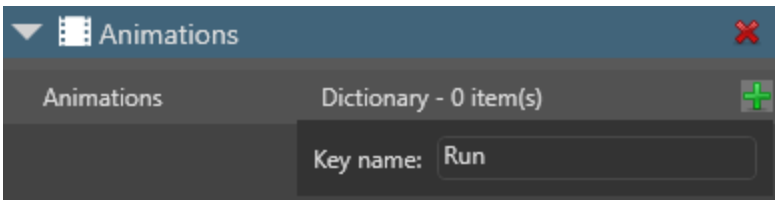
2. In the **Property Grid**, click **Add component** and choose **Animations**.



Game Studio adds an animation component to the entity.

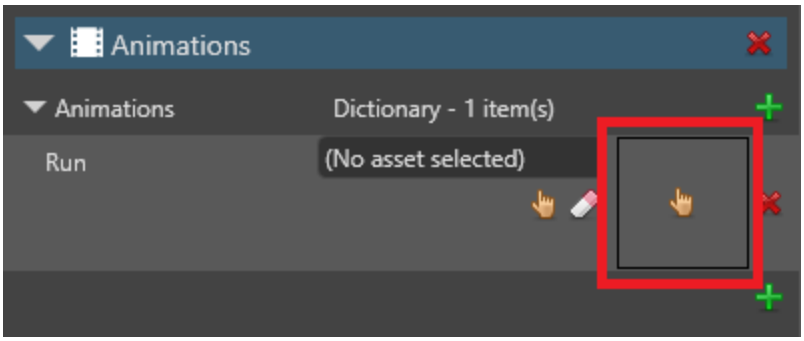
3. In the animation component properties, next to **Animations**, click (**Add**) to add a new animation to the library.

4. Type a name for the animation and press Enter.



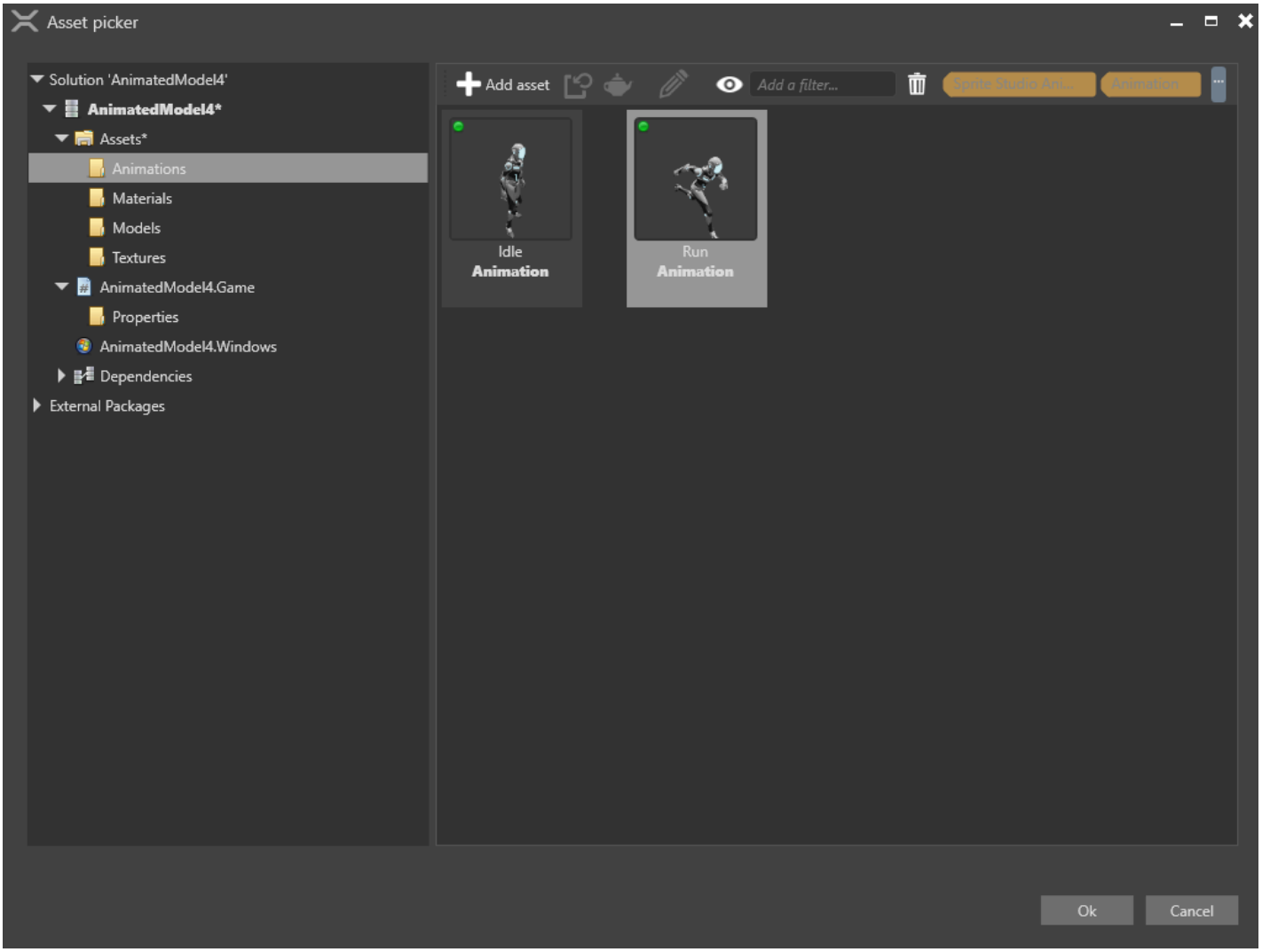
> **TIP**
>
> When you play animations using scripts later, you use this name, **not** the name of the animation asset. To make identification easy, we recommend you give your animation the same name as the animation asset.

5. Click (**Select an asset**).

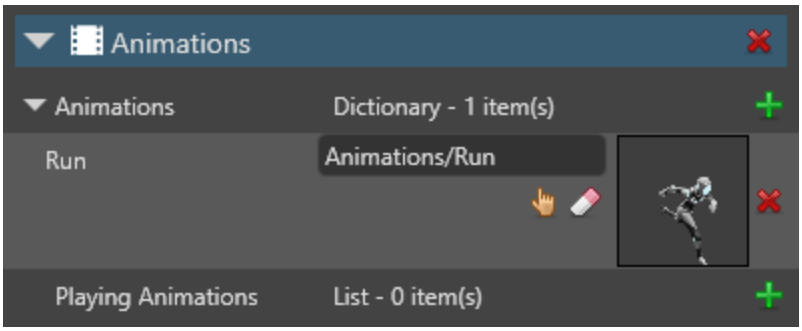


The **Select an asset** window opens.

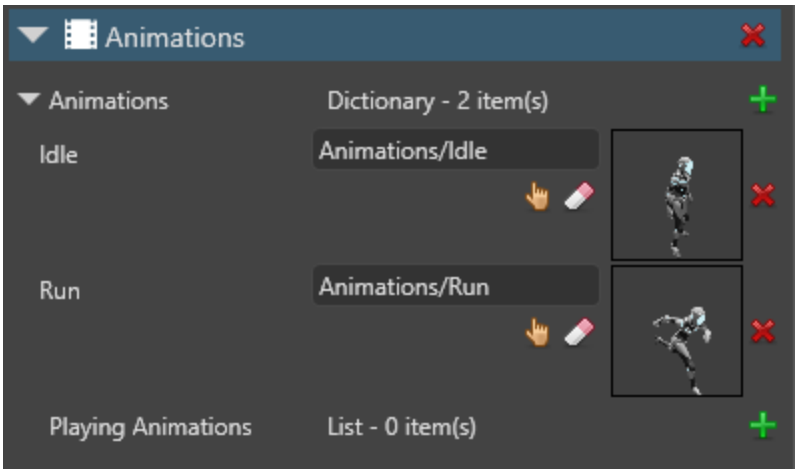
6. Browse to the animation asset you want to add and click **OK**.



Game Studio adds the animation asset to the entity.



You can add as many animations to the animation component as you need. The Property Grid lists them in alphabetical order.



2. Create a script to play the animations

After you add animations to an entity, you need to play them with a script.

Example script

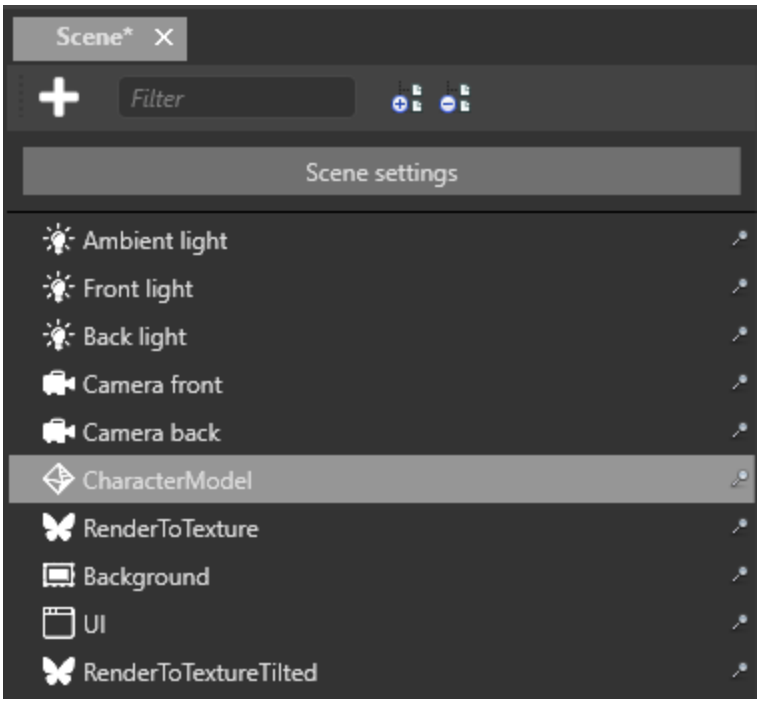
```
public class SimpleAnimationScript : StartupScript
{
    public override void Start()
    {
        Entity.Get<AnimationComponent>().Play("Walk");
    }
}
```

This script looks for an animation with the name *Walk* under the animation component on the entity.

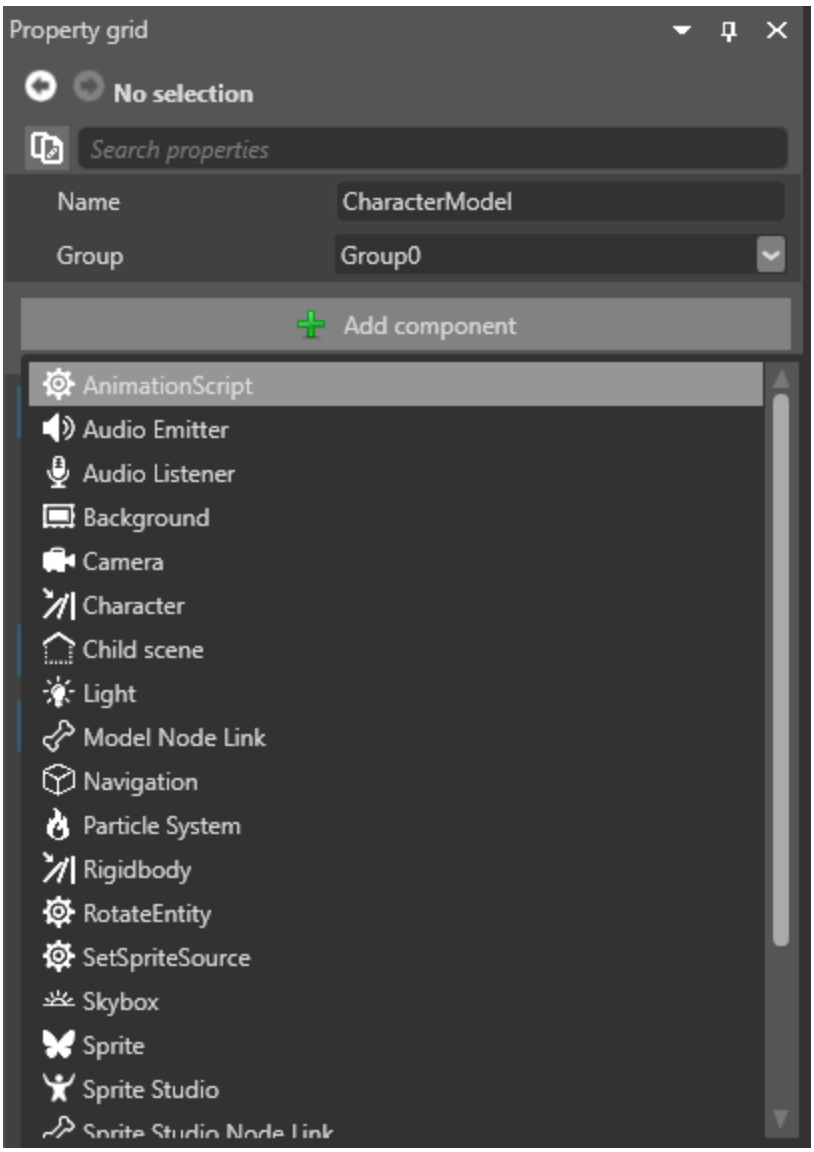
For more information about creating animation scripts, see animation scripts.

3. Add the script to the entity

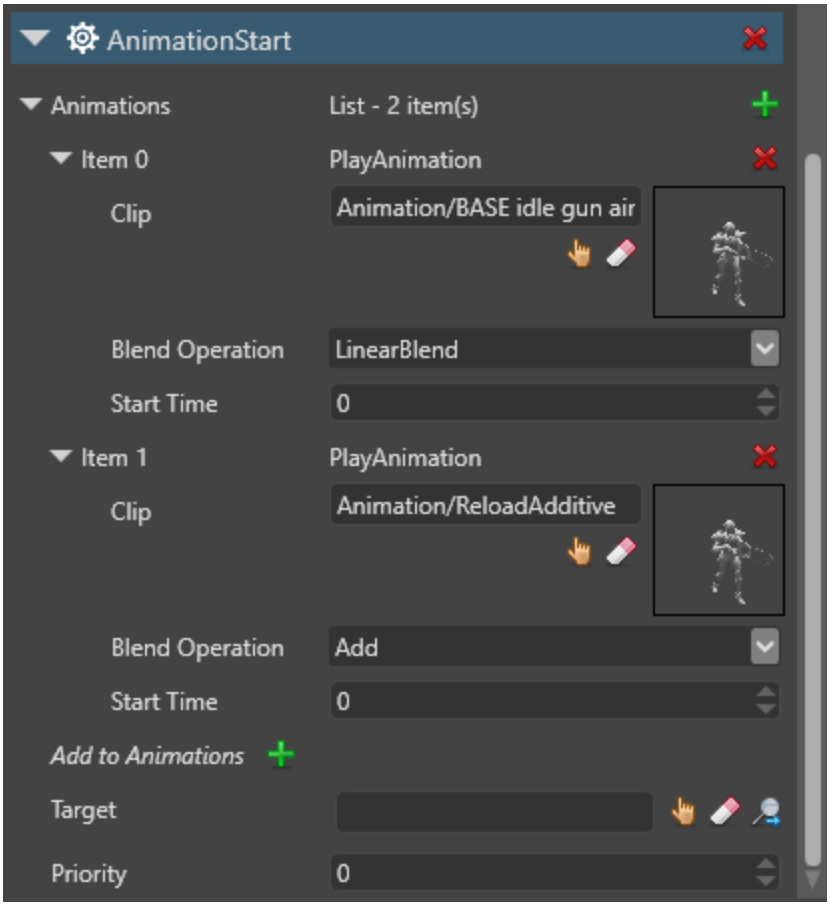
1. In the **Scene Editor**, select the entity you want to animate.



2. In the **Property Grid**, click **Add component** and choose the animation script you want to add.



Game Studio adds the script as a component. You can adjust public variables you define in the script in the **Property Grid** under the script component properties.



See also

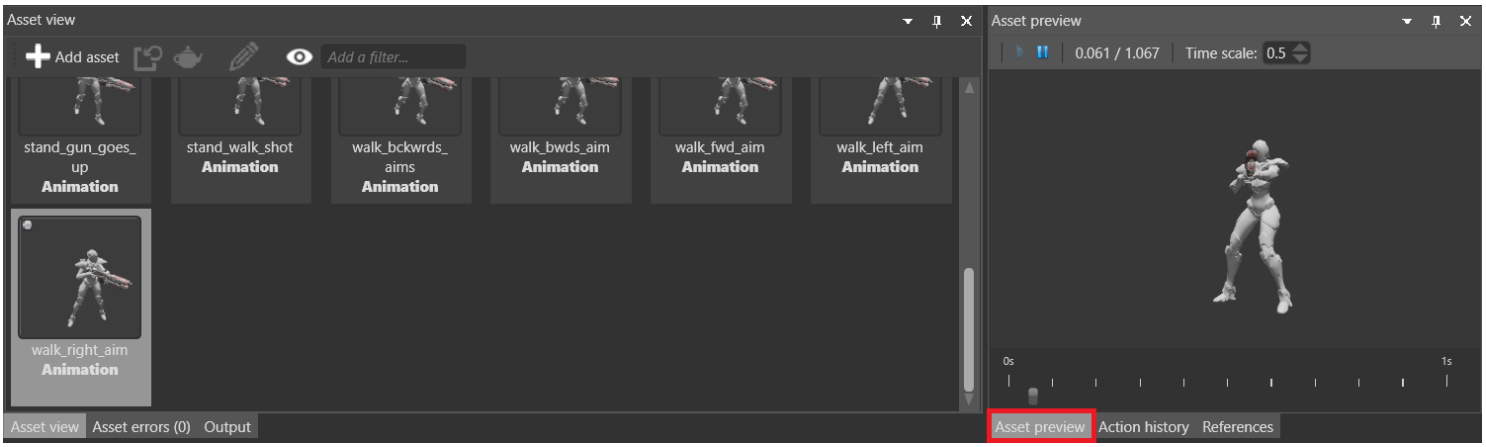
- Animation index
- Import animations
- Animation properties
- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links
- Custom attributes

Preview animations

Intermediate Designer

After you import an animation, you can preview it in the **Asset Preview**.

By default, the Asset Preview is in the bottom-right under the **Asset Preview** tab.

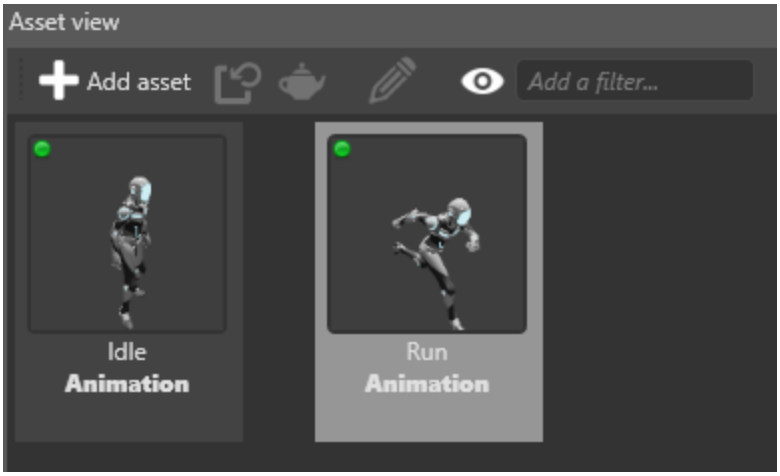


> **TIP**
>
> To rotate the animation, click and drag the mouse.

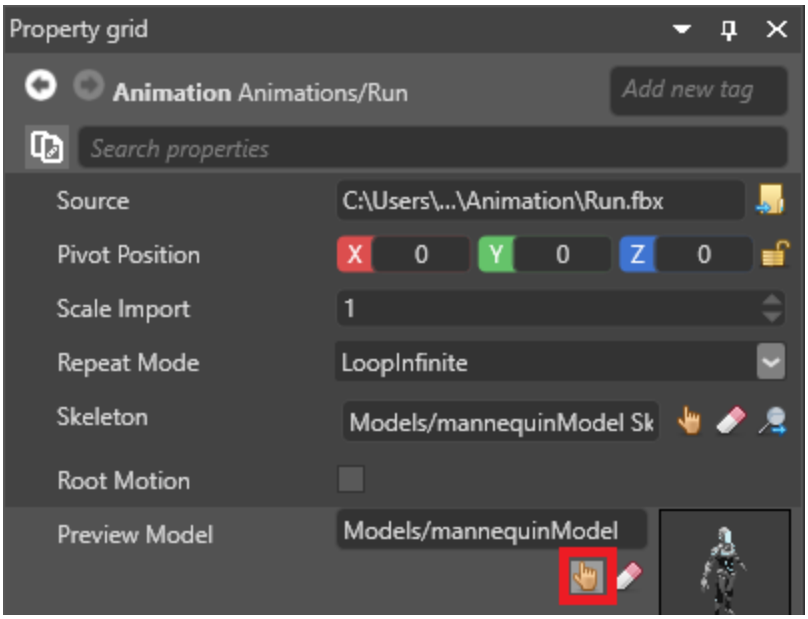
The animation preview uses the model selected in the **preview model** in the **animation asset properties**.

Set the preview model

1. In the **Asset View** (at the bottom by default), select the animation asset.



2. In the **Property Grid** (on the right by default), under **Preview model**, click (**Select an asset**).



   The **Select an asset** window opens.

3. Select the model you want to use to preview the animation.

> **NOTE**
>
> Make sure the model and the animation share identical skeletons.

See also

- Animation index

- Import animations
- Animation properties
- Set up animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links
- Custom attributes

Animation scripts

Intermediate Programmer

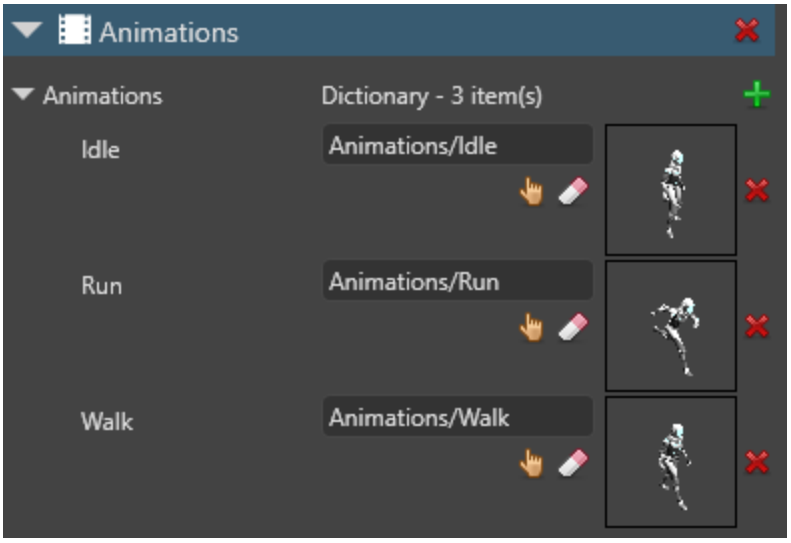
Animations are controlled using scripts.

You can add an AnimationComponent to an entity and set up its parameters in Game Studio. The AnimationComponent class is designed to be used mainly from a script.

The more useful properties include:

Property	Description
Animations	Gets the animation clips associated with this AnimationComponent
BlendTree Builder	Gets or sets animation blend tree builder. Note you can create custom blend trees; for more information, see Custom blend tree
Playing Animations	Gets the list of active animations. Use it to customize your startup animations. The playing animations are updated automatically by the animation processor, so be careful when changing the list or keeping a reference to a playing animation

> **NOTE**
>
> Animation clips you reference in scripts must be added to the same entity under the Animation Component.



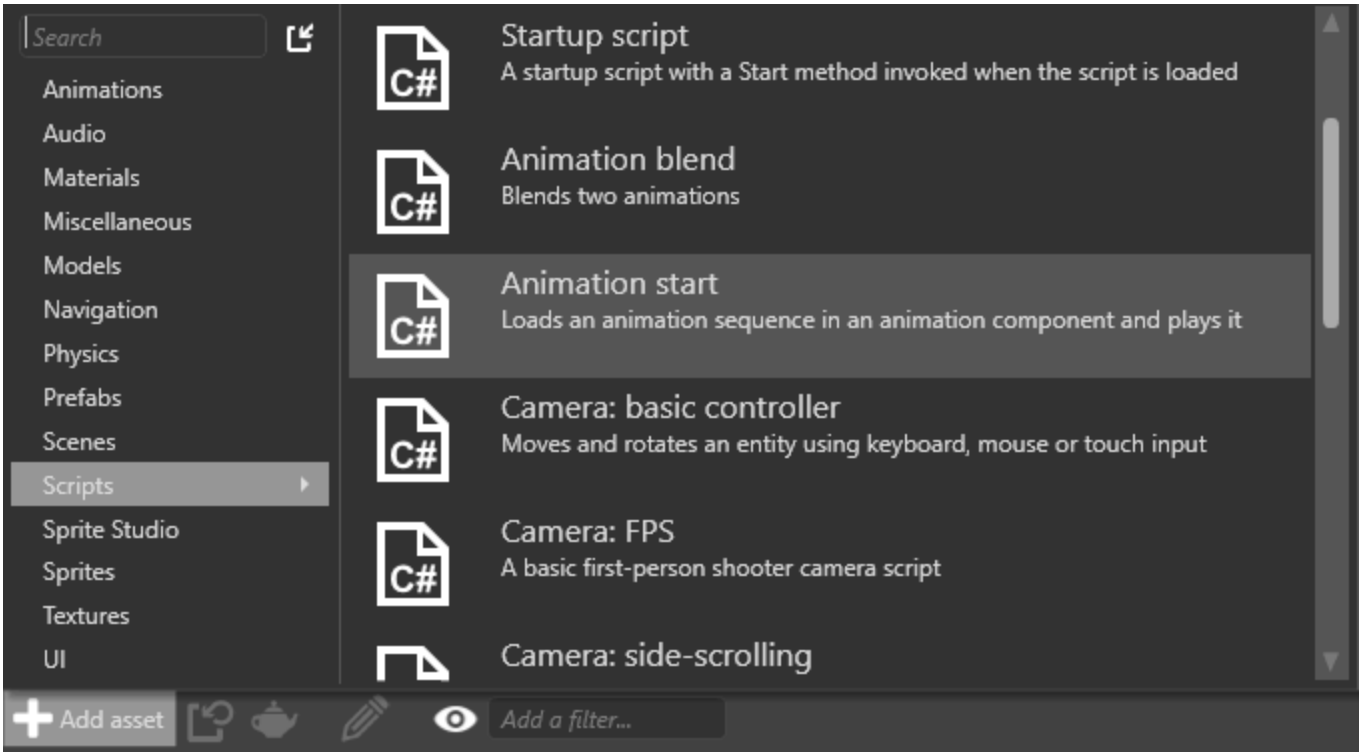
> For more information, see Set up animations.

Use the pre-built **AnimationStart** script

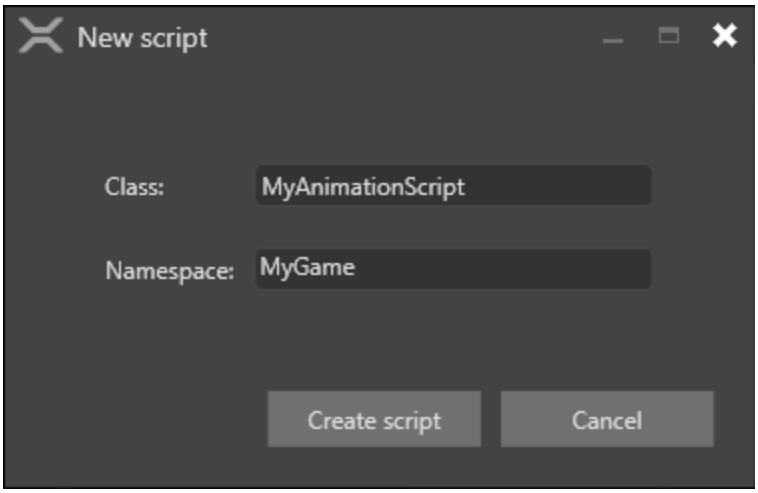
Stride includes a pre-built **AnimationStart** script. You can use this script as a template to write your own animation scripts.

To use the **AnimationStart** script:

1. In the **Asset View** (bottom pane by default), click **Add asset**.
2. Choose **Add asset > Scripts > Animation start**.



3. Specify a name for the script and click **Create script**.



3a. If Game Studio asks if you want to save your script, click **Save script**.

3b. If Game Studio asks if you want to reload the assemblies, click **Reload assemblies**.

4. Edit the script as necessary and save it.

Example animation script

This sample script assigns a simple animation to a character based on its walking speed.

```
using Stride.Engine;

namespace AdditiveAnimation
{
    public class AnimationClipExample : SyncScript
    {
        public float MovementSpeed { get; set; } = 0f;

        private float walkingSpeedLimit = 1.0f;

        // Assuming the script is attached to an entity which has an animation component
        private AnimationComponent animationComponent;

        public override void Start()
        {
            // Cache some variables we'll need later
            animationComponent = Entity.Get<AnimationComponent>();
            animationComponent.Play("Idle");
        }

        protected void PlayAnimation(string name)
        {
            if (!animationComponent.IsPlaying(name))
                animationComponent.Play(name);
        }

        public override void Update()
        {
            if (MovementSpeed <= 0)
            {
                PlayAnimation("Idle");
            }
            else if (MovementSpeed <= walkingSpeedLimit)
            {
                PlayAnimation("Walk");
            }
            else
```

```
            {
                PlayAnimation("Run");
            }
        }
    }
}
```

Override the animation blend tree

You can also override the animation blend tree and do all animation blending in the script. The templates *First-person shooter*, *Third-person platformer* and *Top-down RPG*, which use some advanced techniques, are examples of how to do this. For more information, see custom blend trees.

See also

- Scripts
- Animation index
- Import animations
- Animation properties
- Set up animations
- Preview animations
- Additive animation
- Procedural animation
- Custom blend trees
- Model node links
- Custom attributes

Additive animation

Intermediate Designer

Additive animation is the process of combining animations using **difference clips** (also known as **additive animation clips**).

In the example above, the leftmost animation is the *Walk* animation. The rightmost animation is the *Idle* animation. The two animations in the center are the *Walk* and *Idle* animations respectively, but have the *Reload* animation added to them.

This means we only had to create three animations: *Walk*, *Idle*, and *Reload*. Additionally, we can add the *Reload* animation to other suitable animations (eg *Crouch*, *Strafe* or *Run*). This helps keep the memory budget and number of animations low.

Difference clips

A **difference clip** describes the difference between two animation clips: a **source** and a **reference**.

Take the *Reload* animation above, which we want to add to other animation clips. This is our **source** clip (S). Because the *Reload* animation mainly involves the arms, it will blend well with animations that don't involve the arms (such as idling and crouching). We can use one of these animations — let's say the *Idle* animation — as our **reference** clip (R).

Stride calculates the difference between the source and reference clips to create the **difference clip** (D). The difference clip encodes the difference between the source and reference clips. We can express it as D = S - R.

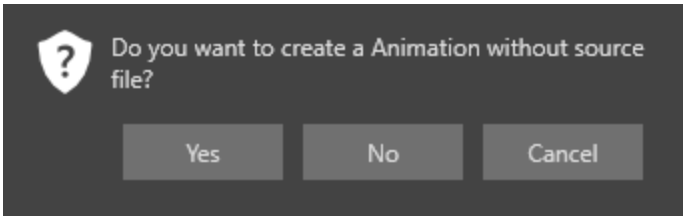
We can use use the difference clip to blend the source and reference animations. We can also use the same difference clip to blend the source animation with **other** animations. If the animation you add it to is sufficiently similar to the original reference clip, then the animations blend effectively. For example, you could use it to add the reload animation to any animation that doesn't use the arms, such as crouching.

> **NOTE**
> Additive animations should use the same skinned mesh and skeleton.

Create a difference clip

1. In the **Asset View** (at the bottom by default), click **Add asset** and select **Animations > Animation**. A browser dialog opens.
2. As we don't need a source for this animation, click **Cancel**.

   Game Studio asks if you want to create an animation without a source file.

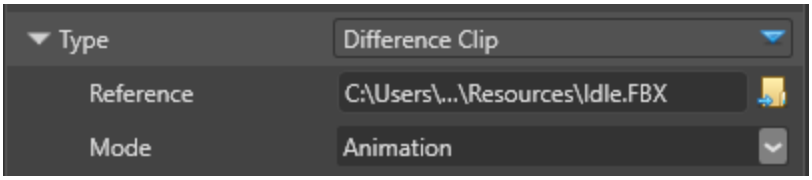
   

3. Click **Yes**. Game Studio adds a new empty animation asset to the Asset View.
4. Give the asset a name that makes it easy to identify. For example, if you want to make a reload animation that can be used with other animations, you could name the asset *ReloadAdditive*.
5. In the **Asset View** (bottom pane by default), select the animation asset you created.
6. In the **Property Grid** (on the right by default), add the **Source** animation clip. This is the animation you want to apply to other animations.

   > **NOTE**
   > Make sure you add the file that contains the animation itself (eg a model file such as .fbx), **not** the animation asset that references it. Animation files are usually saved in the **Resources** folder.

7. Under **Type**, choose **Difference Clip**.

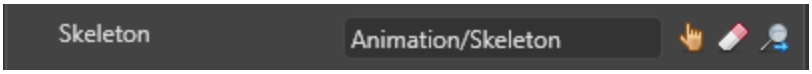
8. Under **Reference**, specify the animation you want to use as your **reference clip**. This is the animation Stride references to create a difference clip.



9. Choose the **Mode** from the drop-down menu.

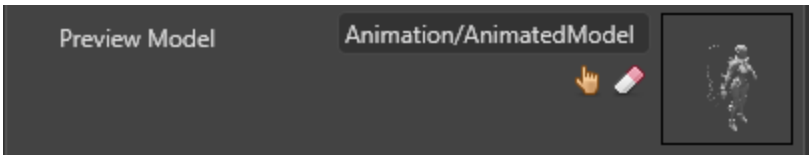
    - **Animation** creates a difference clip from the entire source animation, referencing it frame by frame.
    - **FirstFrame** creates a difference clip from only the first frame of the source animation, as a still pose.

10. Next to **Skeleton**, specify a skeleton for the difference clip.



    This should be a skeleton that works for all the animations you want to blend with the difference clip. In most cases, you should use the same skeleton you used for the source and reference animations.

11. If you want to Preview the animation in the Asset Preview, specify a **Preview model** suitable for the animation.



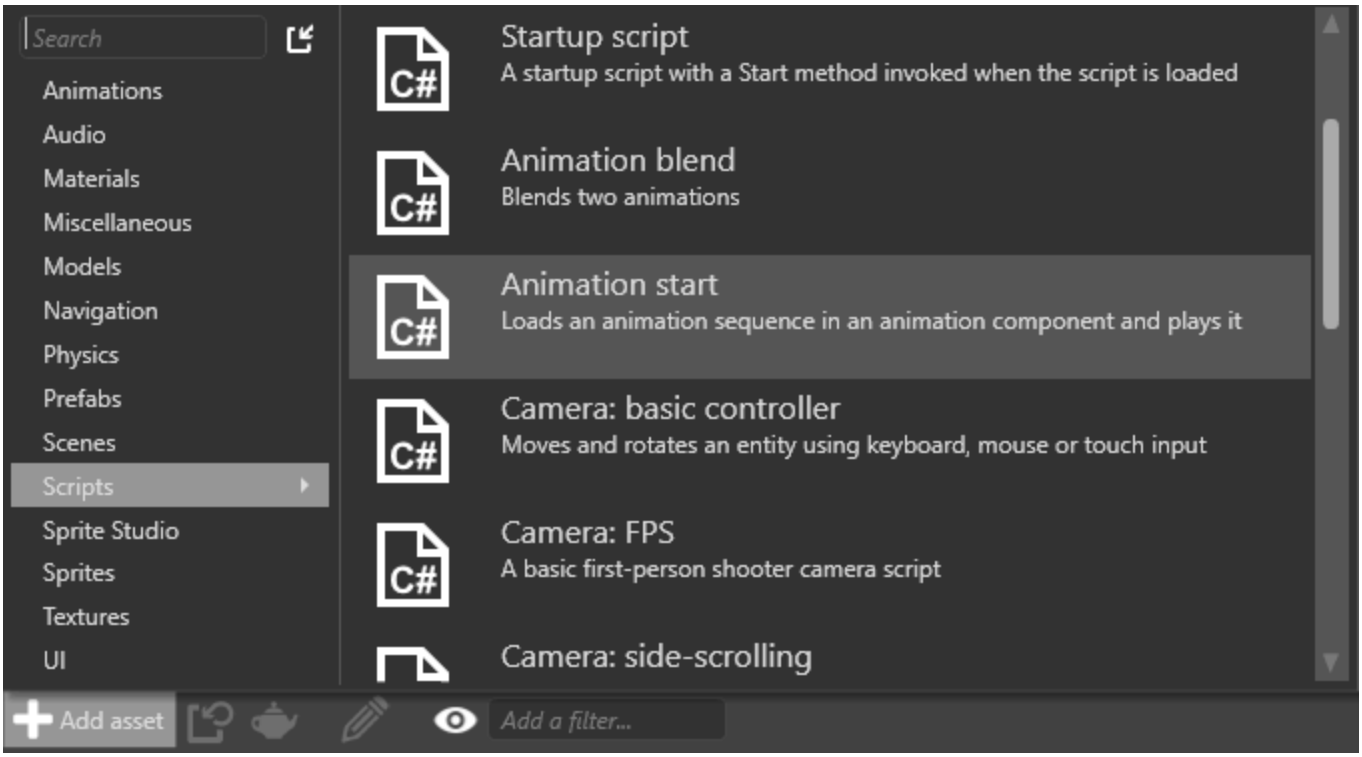
> **NOTE**
> The Asset Preview shows only the source animation you specify in the difference clip.

Use an additive animation

You can use additive animations with animations that use the same skeleton and skinned mesh.

1. In the **Asset View** (in the bottom pane by default), click **Add asset**.

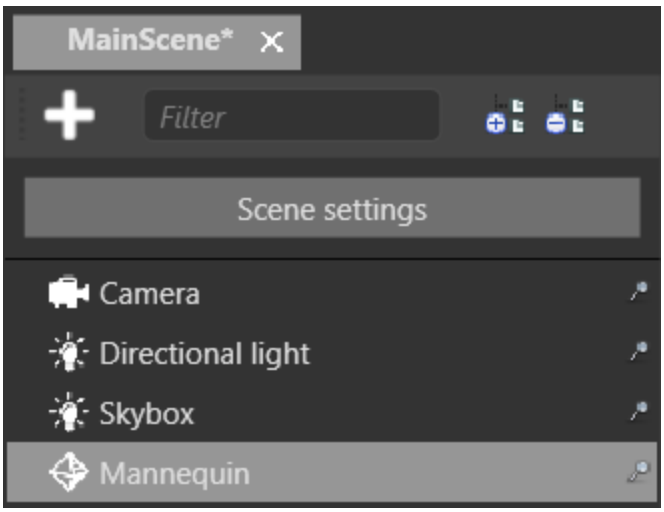
2. Select **Scripts > Animation Start**.



AnimationStart is a startup script you can use to load animations into your model, including additive animations. For more information, see Animation scripts.

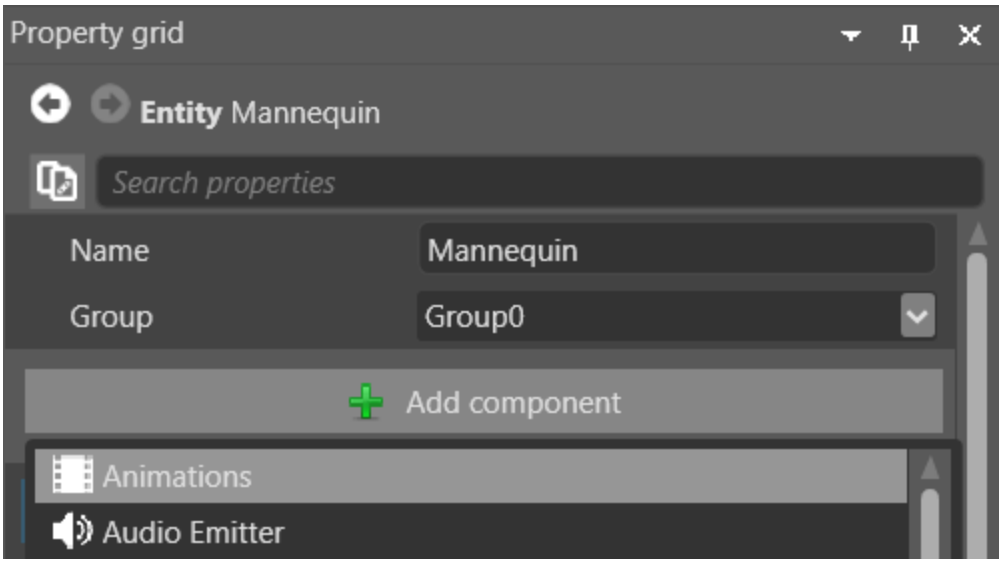
3. Recompile your project to apply the changes.

4. In the **scene view**, select the entity you want to animate.



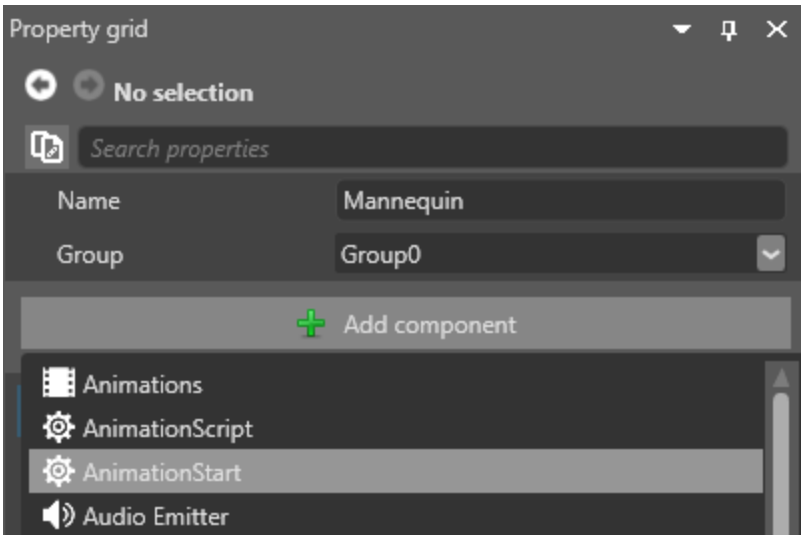
> **NOTE**
>
> To animate an entity, the entity must have a model component.

5. In the **Property Grid** (on the right by default), click **Add component** and choose **Animations**.



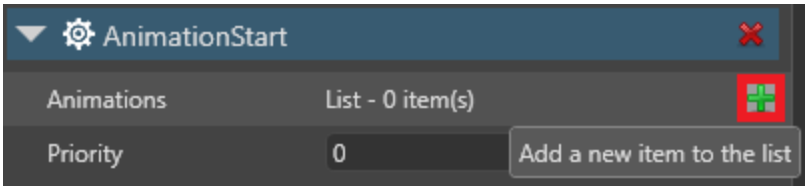
Game Studio adds an animation component to the entity.

6. Click **Add component** and choose the **Animation Start** script.

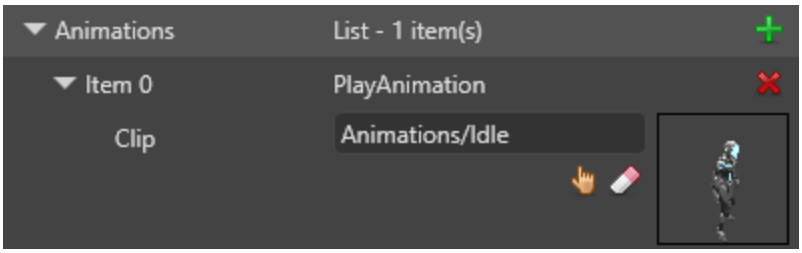


The script lets you customize a list of animations to be loaded into your entity.

7. In the **Animation Start** properties, next to **Animations**, click (**Add**).

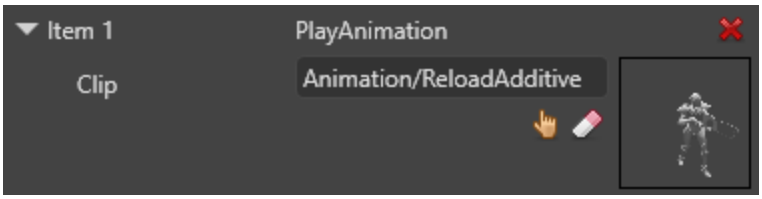


8. Next to **Clip**, specify the **source** animation you set in the difference clip.



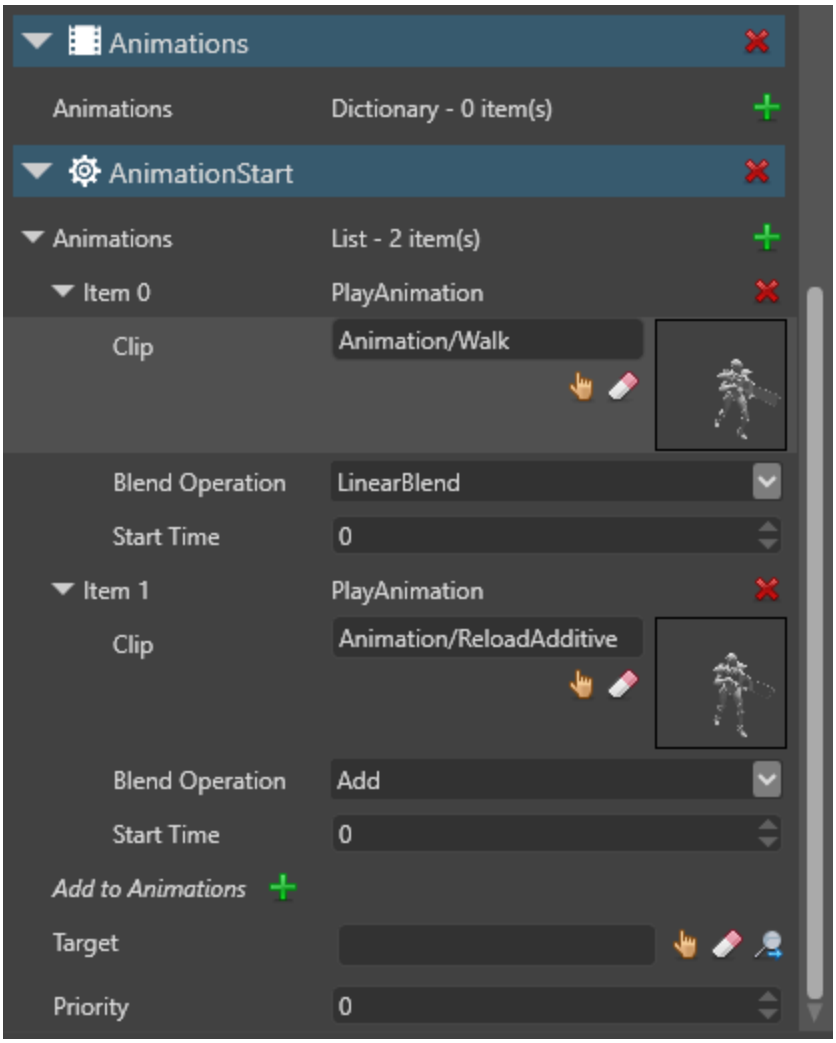
9. Next to **Add to Animations**, click (**Add**).

10. Expand the animation properties. Next to **Clip**, specify the **reference** animation you set in the difference clip.



11. Under **Blend Operation**, select **Additive**.

12. Repeat the steps to add as many animations as you need.



See also

- Animation index
- Import animations
- Animation properties
- Set up animations
- Preview animations
- Animation scripts
- Procedural animation
- Custom blend trees
- Custom attributes

Procedural animation

Intermediate Programmer

Procedural animation is an alternative method of animation. Instead of creating animations yourself, you can use engine components to animate 3D models at runtime.

In some cases, this creates more effective and efficient animations. For example, imagine a shrink effect that happens when the player shoots a monster with a shrink weapon. Instead of creating a complex shrinking animation, you can access the entity TransformComponent and simply scale the enemy down to the required size.

The animation can animate a wide variety of components besides Skeleton bones, including:

- TransformComponent
- LightComponent
- RigidBodyComponent
- Custom components

Stride's animation system works just like Blender or Maya's curve animation editor. Each bone/value is assigned a curve composed of several points that are interpolated either in linear, cubic or constant fashion.

Code samples

Transform component

```
public class AnimationScript : StartupScript
{
    public override void Start()
    {
        // Create an AnimationClip. Make sure you set its duration properly.
        var animationClip = new AnimationClip { Duration = TimeSpan.FromSeconds(1) };

        // Add a curves specifying the path to the transformation property.
        // - You can index components using a special syntax to their key.
        // - Properties can be qualified with a type name in parenthesis.
        // - If a type isn't serializable, its fully qualified name must be used.

        animationClip.AddCurve("[TransformComponent.Key].Rotation", CreateRotationCurve());

        // Optional: pack all animation channels into an optimized interleaved format.
        animationClip.Optimize();

        // Add an AnimationComponent to the current entity and register our custom clip.
```

```
        const string animationName = "MyCustomAnimation";
        var animationComponent = Entity.GetOrCreate<AnimationComponent>();
        animationComponent.Animations.Add(animationName, animationClip);

        // Play the animation right away and loop it.
        var playingAnimation = animationComponent.Play(animationName);
        playingAnimation.RepeatMode = AnimationRepeatMode.LoopInfinite;
        playingAnimation.TimeFactor = 0.1f; // slow down
        playingAnimation.CurrentTime = TimeSpan.FromSeconds(0.6f); // start at
different time
    }

    // Set custom linear rotation curve.
    private AnimationCurve CreateRotationCurve()
    {
        return new AnimationCurve<Quaternion>
        {
            InterpolationType = AnimationCurveInterpolationType.Linear,
            KeyFrames =
            {
                CreateKeyFrame(0.00f, Quaternion.RotationX(0)),
                CreateKeyFrame(0.25f, Quaternion.RotationX(MathUtil.PiOverTwo)),
                CreateKeyFrame(0.50f, Quaternion.RotationX(MathUtil.Pi)),
                CreateKeyFrame(0.75f, Quaternion.RotationX(-MathUtil.PiOverTwo)),
                CreateKeyFrame(1.00f, Quaternion.RotationX(MathUtil.TwoPi))
            }
        };
    }

    private static KeyFrameData<T> CreateKeyFrame<T>(float keyTime, T value)
    {
        return new KeyFrameData<T>((CompressedTimeSpan)TimeSpan.FromSeconds(keyTime),
value);
    }
}
```

Light component's color

```
public class AnimationLight : StartupScript
{
    public override void Start()
    {
        // Our entity should have a light component
        var lightC = Entity.Get<LightComponent>();
```

```
        // Create an AnimationClip and store unserializable types. Make sure you set its
duration properly.
        var clip = new AnimationClip { Duration = TimeSpan.FromSeconds(1) };
        var colorLightBaseName = typeof(ColorLightBase).AssemblyQualifiedName;
        var colorRgbProviderName = typeof(ColorRgbProvider).AssemblyQualifiedName;

        // Point to the path of the color property of the light component
        clip.AddCurve(
            $"[LightComponent.Key].Type.({colorLightBaseName})Color.
({colorRgbProviderName})Value",
            CreateLightColorCurve()
        );

        // Play the animation right away and loop it.
        clip.RepeatMode = AnimationRepeatMode.LoopInfinite;
        var animC = Entity.GetOrCreate<AnimationComponent>();
        animC.Animations.Add("LightCurve",clip);
        animC.Play("LightCurve");
    }
    private AnimationCurve CreateLightColorCurve()
    {
        return new AnimationCurve<Vector3>
        {
            InterpolationType = AnimationCurveInterpolationType.Linear,
            KeyFrames =
            {
                CreateKeyFrame(0.00f, Vector3.UnitX), // Make the first keyframe a red color

                CreateKeyFrame(0.50f, Vector3.UnitZ), // then blue

                CreateKeyFrame(1.00f, Vector3.UnitX), // then red again
            }
        };
    }

    private static KeyFrameData<T> CreateKeyFrame<T>(float keyTime, T value)
    {
        return new KeyFrameData<T>((CompressedTimeSpan)TimeSpan.FromSeconds(keyTime),
value);
    }
}
```

NOTE

If you need to animate a bone procedurally you must use the `NodeTransformations` field of the `Skeleton`.

See also

- Animation index
- Import animations
- Animation properties
- Set up animations
- Preview animations
- Animation scripts
- Additive animation
- Custom blend trees
- Model node links
- Custom attributes

Custom blend trees

Advanced Programmer

The AnimationComponent has the property AnimationComponent.BlendTreeBuilder. If you want absolute control over which animations are played, how are they blended and what weights they have, you can create a script which implements from `IBlendTreeBuilder` and assign it to the BlendTreeBuilder under your animation component.

When the animation component is updated, it calls `void BuildBlendTree(FastList<AnimationOperation> animationList)` on your script instead of updating the animations itself. This allows you to choose any combination of animation clips, speeds and blends, but is also more difficult, as all the heavy lifting is now on the script side.

The templates *First-person shooter*, *Third-person platformer* and *Top-down RPG*, included with Stride, are examples of how to use custom blend trees.

Code sample

```
public class AnimationBlendTree : SyncScript, IBlendTreeBuilder
{
    /// <summary>
    /// The animation component is required
    /// </summary>
    [Display("Animation Component")]
    public AnimationComponent AnimationComponent { get; set; }

    [Display("Walk")]
    public AnimationClip AnimationWalk { get; set; }

    [Display("Run")]
    public AnimationClip AnimationRun { get; set; }

    [Display("Lerp Factor")]
    public float LerpFactor = 0.5f;

    private AnimationClipEvaluator animEvaluatorWalk;
    private AnimationClipEvaluator animEvaluatorRun;
    private double currentTime = 0;

    public override void Start()
    {
        base.Start();

        // IMPORTANT STEP
```

```
        // By setting a custom blend tree builder we can override the default behavior of
the animation system.
        // Instead, BuildBlendTree(FastList<AnimationOperation> blendStack) will be called
each frame.
        // We need to update the animation state in Update() and then
        // pass the new animation state (stack = blend tree) to the animation system.
        AnimationComponent.BlendTreeBuilder = this;

        // As we override the animation system, we need to create an AnimationClipEvaluator
for each clip we want to use.
        animEvaluatorWalk = AnimationComponent.Blender.CreateEvaluator(AnimationWalk);
        animEvaluatorRun = AnimationComponent.Blender.CreateEvaluator(AnimationRun);
    }

    public override void Cancel()
    {
        // When the script is cancelled, don't forget to release all animation resources
created in Start() - AnimationClipEvaluators
        AnimationComponent.Blender.ReleaseEvaluator(animEvaluatorWalk);
        AnimationComponent.Blender.ReleaseEvaluator(animEvaluatorRun);
    }

    public override void Update()
    {
        // Use DrawTime rather than UpdateTime because the animations are updated only when
they are drawn.
        var time = Game.DrawTime;

        // This update function accounts for animation with different durations,
        // keeping a current time relative to the blended maximum duration.
        long blendedMaxDuration = (long)MathUtil.Lerp(AnimationWalk.Duration.Ticks,
AnimationRun.Duration.Ticks, LerpFactor);

        var currentTicks = TimeSpan.FromTicks((long)(currentTime * blendedMaxDuration));

        currentTicks = blendedMaxDuration == 0
            ? TimeSpan.Zero
            : TimeSpan.FromTicks((currentTicks.Ticks + (long)(time.Elapsed.Ticks))
% blendedMaxDuration);

        currentTime = ((double)currentTicks.Ticks / (double)blendedMaxDuration);
    }

    /// BuildBlendTree is called every frame from the animation system when the
AnimationComponent needs to be evaluated.
    /// It overrides the default behavior of the AnimationComponent by setting a custom
```

```
blend tree.
    public void BuildBlendTree(FastList<AnimationOperation> blendStack)
    {
        var timeWalk = TimeSpan.FromTicks((long) (currentTime *
AnimationWalk.Duration.Ticks));
        var timeRun  = TimeSpan.FromTicks((long) (currentTime
* AnimationRun.Duration.Ticks));

        // Build the animation blend tree (stack)
        blendStack.Add(AnimationOperation.NewPush(animEvaluatorWalk, timeWalk));    // Will
PUSH animation state to be evaluated at the specified Time.
        blendStack.Add(AnimationOperation.NewPush(animEvaluatorRun, timeRun));      // Will
PUSH another animation state to be evaluated at the specified Time.
        blendStack.Add(AnimationOperation.NewBlend(CoreAnimationOperation.Blend,
LerpFactor));   // Will POP the last two states, blend them with the factor and PUSH back
the result.

        // NOTE
        // Because the blending operations are laid out in a stack you have to pack the
operations in this manner.
        // In general, traversing a binary tree depth-first and adding operations as you
*leave* precessed nodes should be sufficient.
        // For non-binary trees, you have to properly weight the blending factors as well

        // DONE
        // The top of the stack now contains the final state used for the animated model
    }
}
```

See also

- Animation index
- Import animations
- Animation properties
- Set up animations
- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Model node links
- Custom attributes

Model node links

Beginner Artist

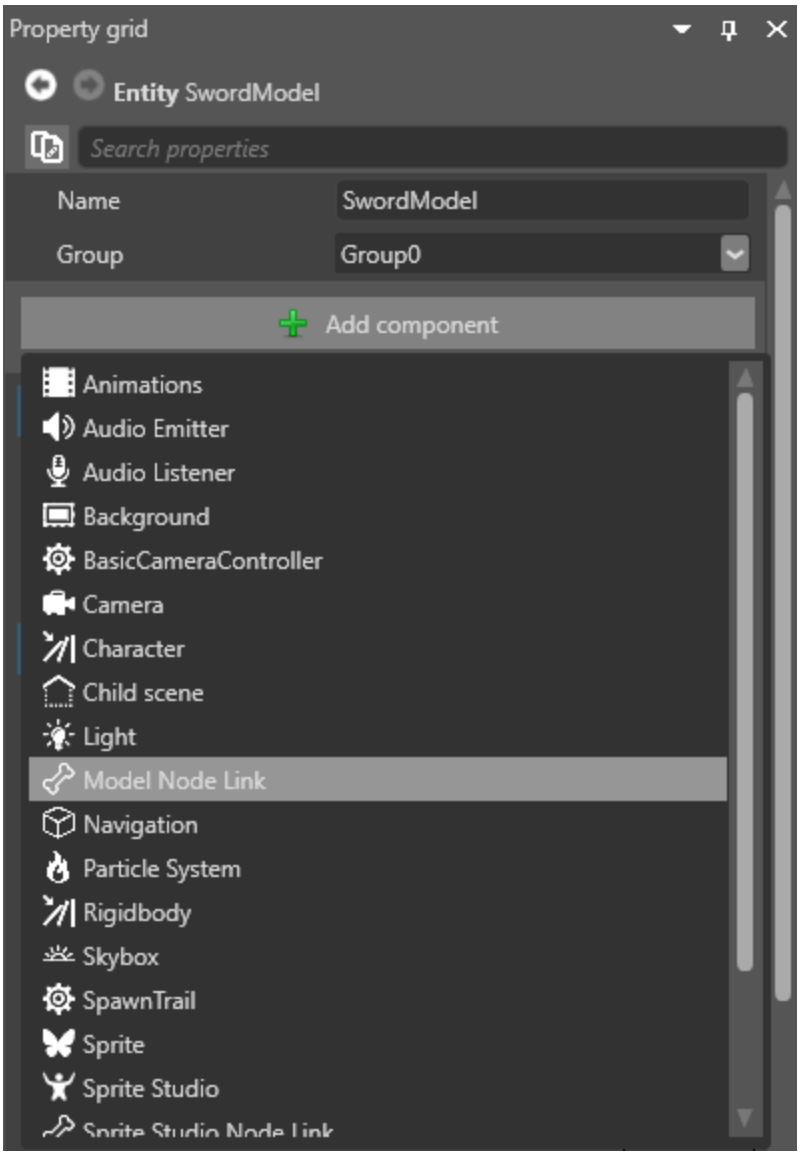
> **NOTE**
>
> In some versions of Stride, **Model node links** are called **Bone links**.

The **model node link** component attaches an entity to a node of a skeleton on another entity.

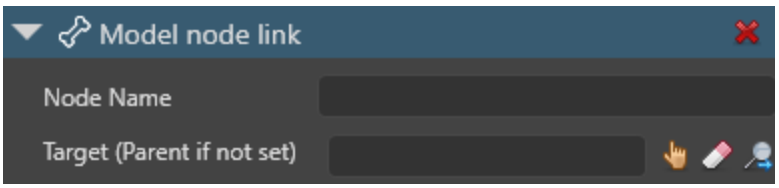
For example, imagine you have two models: a knight, and a sword. The character has a sword swinging animation. You can use a model link node to place the sword in the knight's hand and attach it to the correct node in the knight skeleton, so the sword swings with the knight animation.

Set up a model node link component

1. In the **Scene Editor**, select the entity you want to link to a node in another entity.
2. In the **Property Grid**, click **Add component** and select **Model node link**.



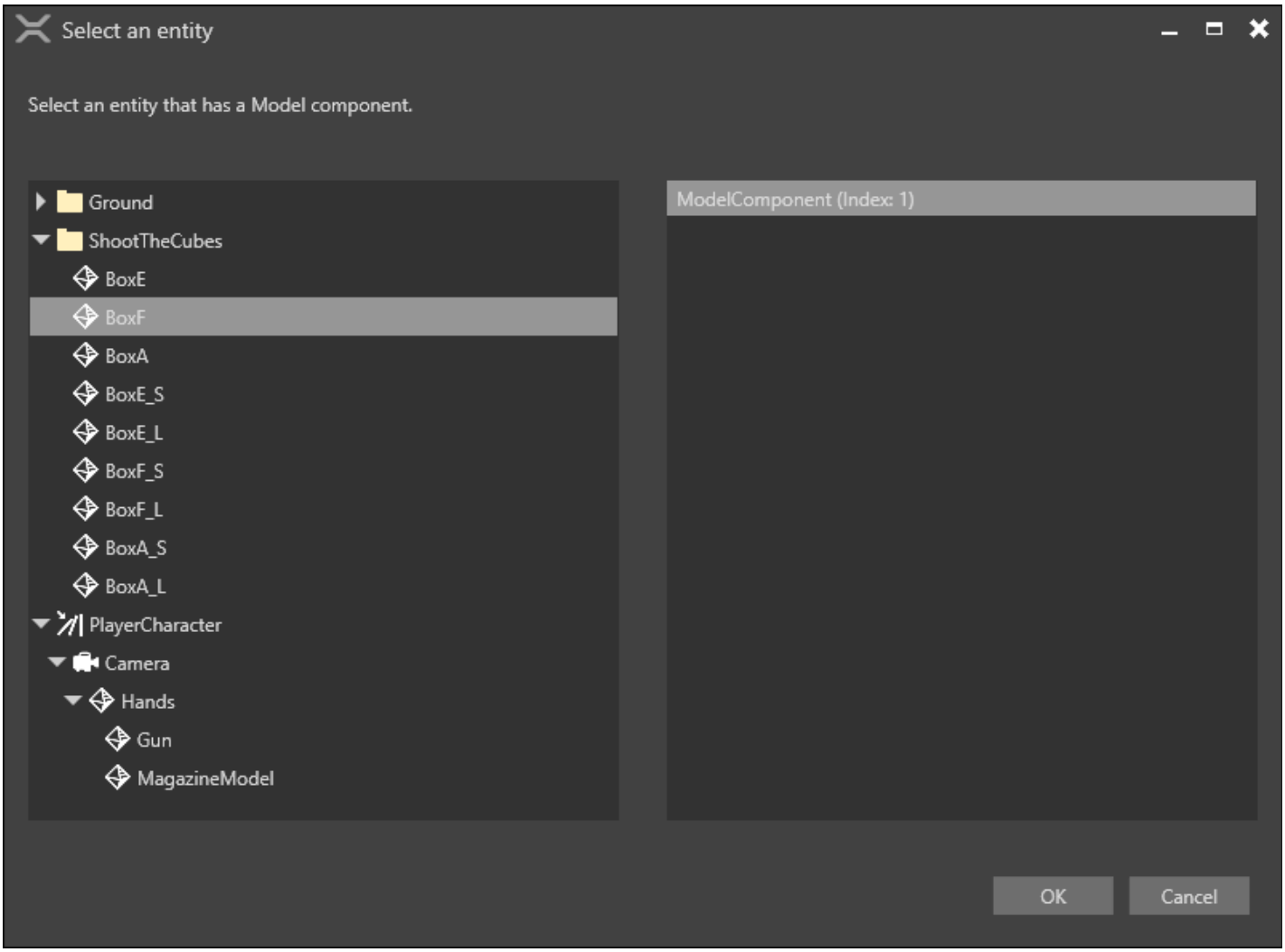
Game Studio adds a model node link component to the entity.



The component only has two properties: **Node name** and **Target**.

3. Next to **Target**, click .

The **Select an entity** window opens.

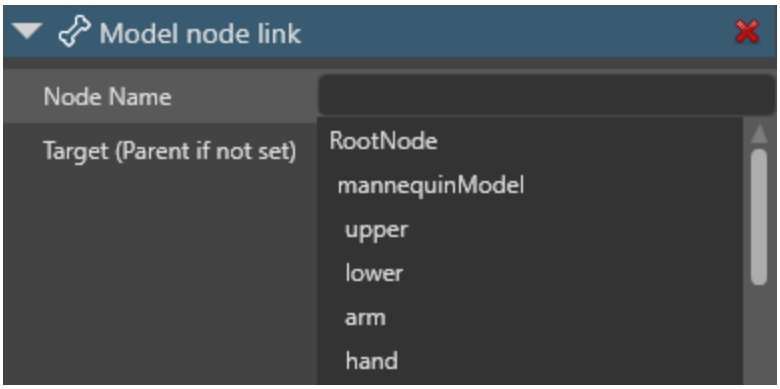


4. Select the model you want to link the entity to and click **OK**.

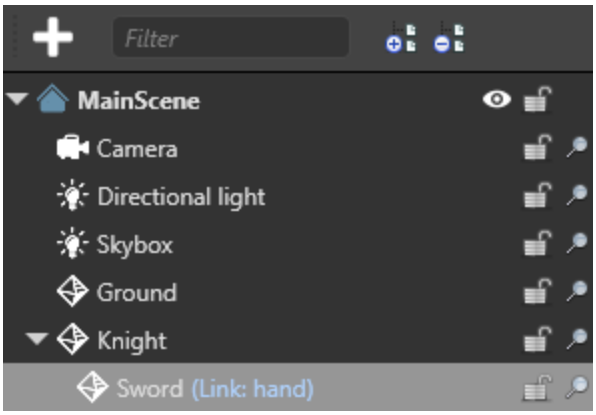
> **NOTE**
>
> The entity you link to must have a model with a skeleton, even if the model isn't visible at runtime.

> **TIP**
>
> If you don't specify a model, Stride links the entity to the model on the parent entity.

5. In **Node name**, select the node in the model you want to attach this entity to.

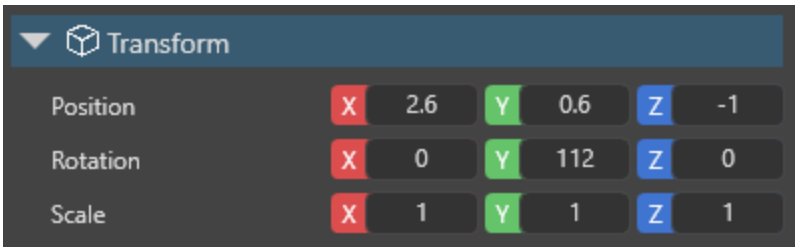


After you link the node, the Entity Tree shows the link in blue next to the entity name.



Offset

To add an offset to the linked entity, use the entity's TransformComponent.



> **NOTE**
>
> If you don't want to add an offset, make sure the values are all set to `0,0,0`.

Example script

This script demonstrates how to link one entity (such as a `SwordModel`) to a specific bone (`weapon_bone_R`) in another entity's skeleton hierarchy (in this case, the `mannequinModel`) using Stride's `ModelNodeLinkComponent`.

```
public class BoneLink : StartupScript
{
    // This example assumes you've created a project with the default Stride models
    // "mannequinModel" and "SwordModel." Add them from the "Assets/Models" folder to
your scene,
    // and then attach this script to the "SwordModel" entity

    ModelNodeLinkComponent boneLink;

    public override void Start()
    {
        // Initialize the script
        // Here we locate the entity named "mannequinModel" by searching the root
scene's entities
        Entity owner = SceneSystem.SceneInstance.RootScene.Entities.Where(e => e.Name
== "mannequinModel").Single();

        boneLink = new ModelNodeLinkComponent
        {
             // This is the ModelComponent on the target entity (mannequinModel)
            Target = owner.Get<ModelComponent>(),

            // We set a "hard link" to Nodes[70], which corresponds to "weapon_bone_R"
            // in the target's skeleton hierarchy
            NodeName = owner.Get<ModelComponent>().Model.Skeleton.Nodes[70].Name
        };

        // Finally, add this link component to our current (SwordModel) entity
        base.Entity.Components.Add(boneLink);
    }
}
```

See also

- Import animations
- Animation properties
- Set up animations
- Preview animations
- Animation scripts
- Additive animation
- Procedural animation
- Custom blend trees
- Custom attributes

For examples of how model node links are used, see:

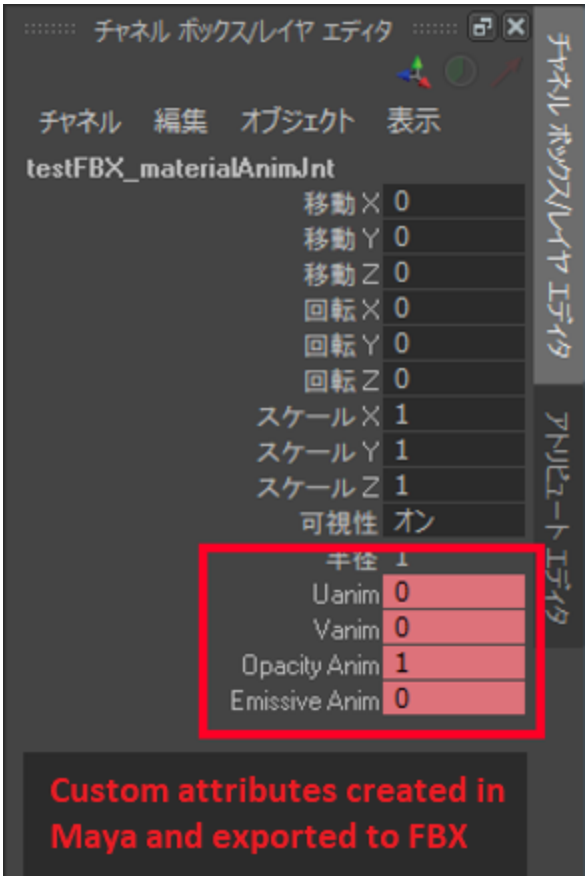
- Particles — Create a trail
- Cameras — Animate a camera with a model file

Custom attributes

Intermediate

You can import custom attributes created in modeling tools such as Maya.

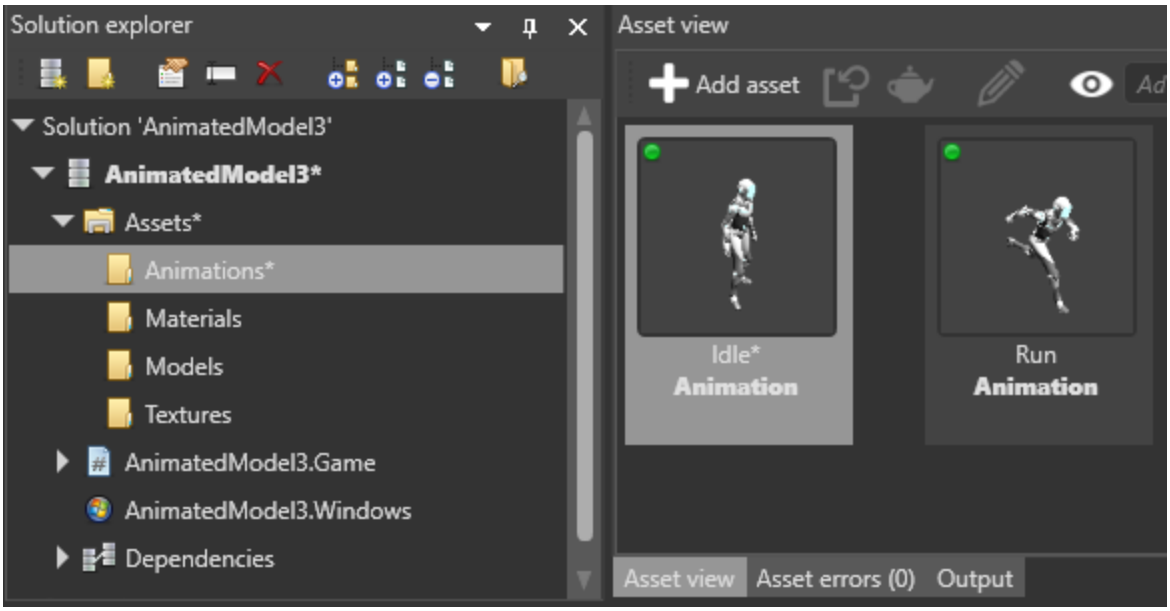
Currently, you can only import custom **animated** attributes. Attributes that aren't animated can't be imported.



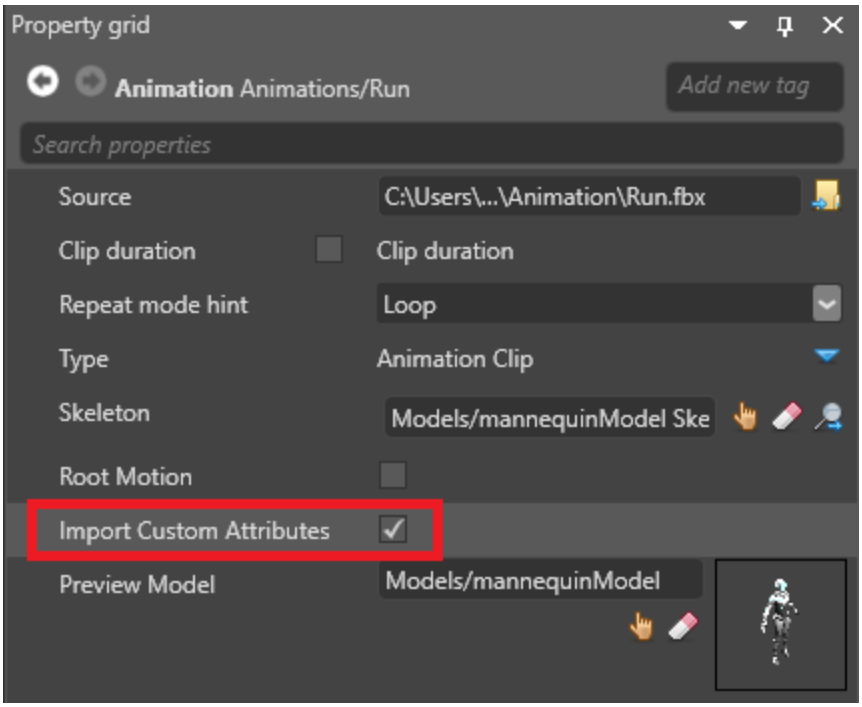
1. Import custom attributes

1. Import the animation. For instructions, see Import animations.

2. In the **Asset View**, select the animation asset.



3. In the **Property Grid**, select **Import custom attributes**.



When the assets are built, Stride imports the custom attributes from the FBX file.

2. Control custom attributes with a script

Add a script to read the custom attributes and copy their value to another property. This can be a separate script, or part of another animation script.

To look up an attribute, use `NodeName_AttributeName`. For example, if you have the node `myNode` with the custom attribute `myAttribute`, use `myNode_myAttribute`.

Example script

```
using Stride.Animations;
using Stride.Engine;
using Stride.Rendering;
using Stride.Audio;
using Stride.Rendering.Materials;
using System.Linq;

namespace Sample
{
    public class HologramScript : SyncScript
    {
        public Material MyMaterial;

        private AnimationComponent animationComponent;
        private AnimationProcessor animationProcessor;

        public override void Start()
        {
            base.Start();

            animationComponent = Entity.GetOrCreate<AnimationComponent>();
            animationProcessor =
SceneSystem.SceneInstance.Processors.OfType<AnimationProcessor>().FirstOrDefault();
        }

        public override void Update()
        {
            if (animationProcessor == null || MyMaterial == null)
                return;

            // Animation result may be Null if animation hasn't been played yet.
            var animResult = animationProcessor.GetAnimationClipResult(animationComponent);
            if (animResult == null)
                return;

            // Read the value of the animated custom attribute:
            float emissiveIntensity = 0;
            unsafe
            {
                fixed (byte* structures = animResult.Data)
                {
                    foreach (var channel in animResult.Channels)
                    {
                        if (!channel.IsUserCustomProperty)
                            continue;
```

```
                        var structureData = (float*)(structures + channel.Offset);
                        var factor = *structureData++;
                        if (factor == 0.0f)
                            continue;

                        var value = *structureData;
                        if (channel.PropertyName == "myNode_myProperty")
                            emissiveIntensity = value;
                    }
                }
            }

            // Bind the material parameter:

            MyMaterial.Passes[0].Parameters.Set(MaterialKeys.EmissiveIntensity,
emissiveIntensity);
        }
    }
}
```

Audio

You can import sound files and use them in your games. Stride supports audio features including 3D spatialized audio, streaming, and low-latency playback.

In this section

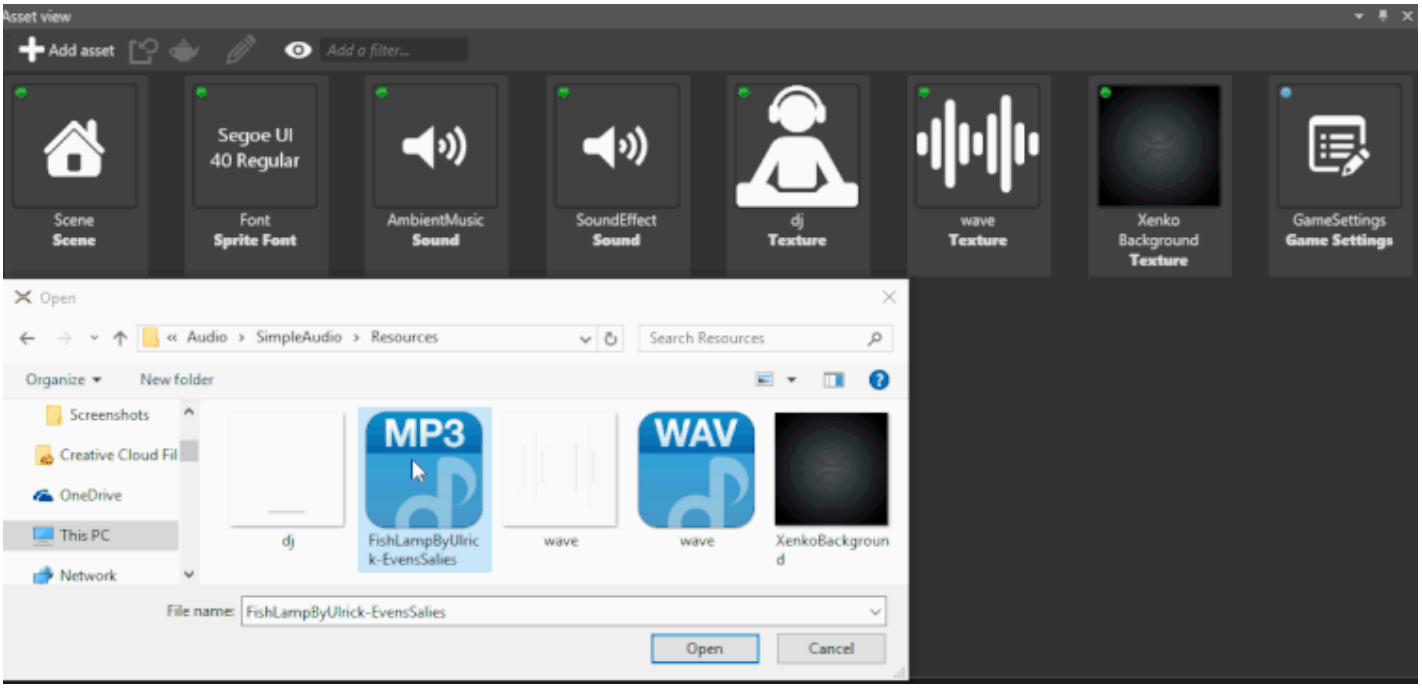
- Import audio
- Audio asset properties
- Non-spatialized audio
- Spatialized audio
  - Audio emitters
  - Audio listeners
  - HRTF
- Stream audio
- Global audio settings
- Play a range within an audio file
- Custom audio data
- Set an audio device

Import audio

Beginner Designer

You can import audio files to use as **audio assets** in your project. You can import file types including .wav, .mp3, .ogg, .aac, .aiff, .flac, .m4a, .wma, and .mpc.

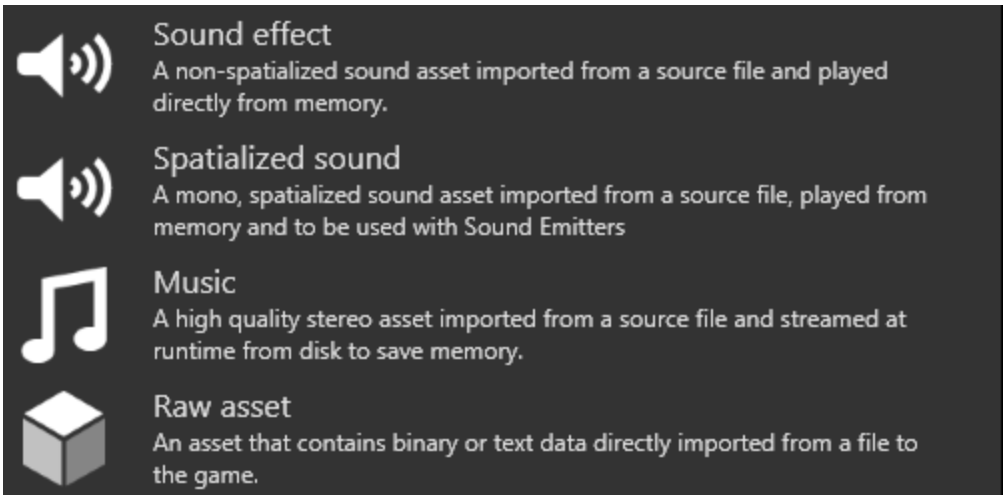
1. Drag and drop the audio file from Windows Explorer to the **Asset View**:



   Alternatively, in the **Asset View**:

   1. Click Add asset
   2. Click (**Import audio directly from file**) and select the audio file.

2. To give the audio asset some default properties, choose a preset. (You can always change the properties in the Property Grid later.)



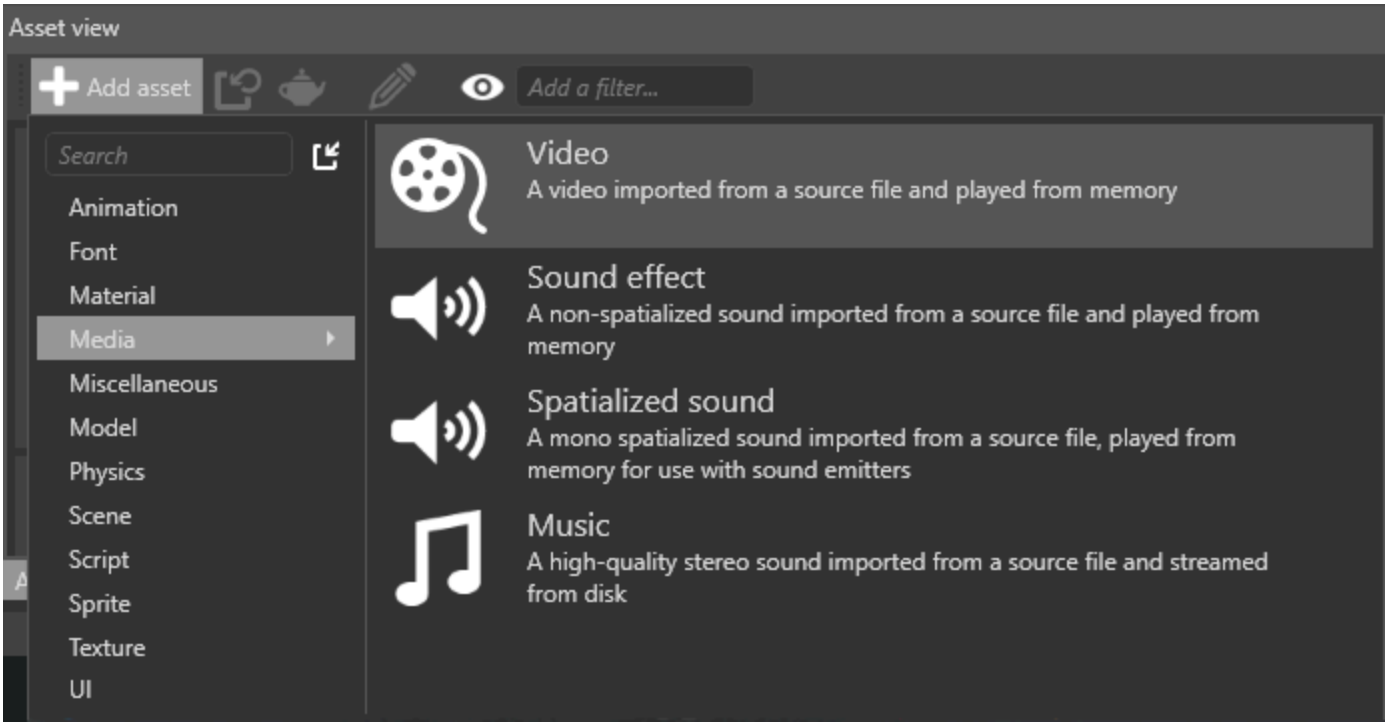
- **Sound effect**: Recommended for smaller files that you want to play directly from memory.
- *_Spatialized audio_: Process the audio asset as spatialized audio. Note that Stride processes audio files as mono (single-channel) audio. The source file is unaffected.
- **Music**: Recommended for larger files that you want to stream from disk to save memory.

After you import an audio file, you can select it as an asset in the **Asset View**.

Import audio from a video file

You can also import a video file and choose to import only the audio tracks from it.

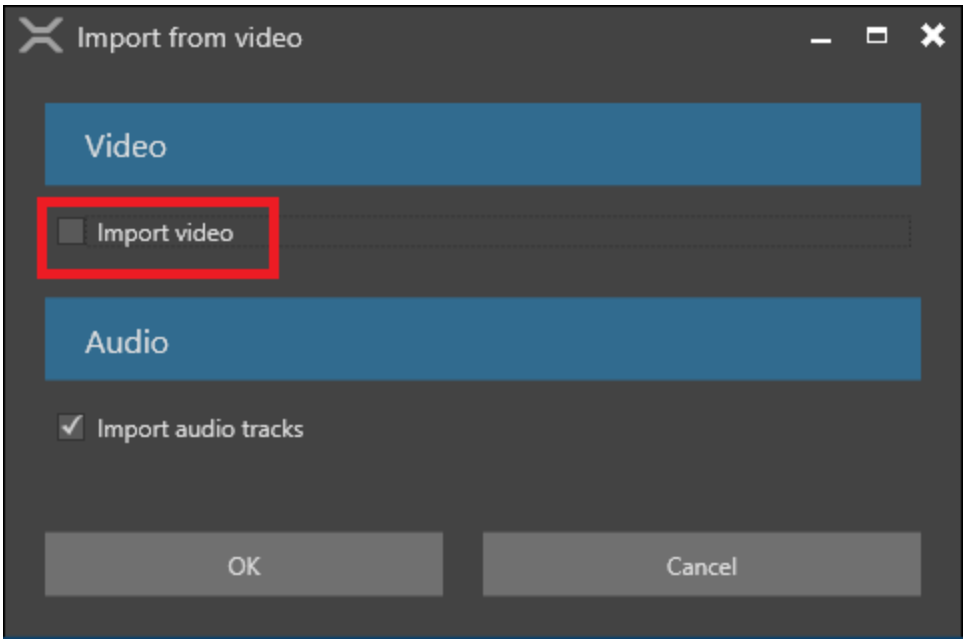
1. In the **Asset View**, click **Add asset** and select **Media > Video**.



2. Browse to the video you want to import audio from and click **Open**.

   Alternatively, drag the file from **Explorer** into the **Asset View**.

3. Clear **Import video** and click **OK**.



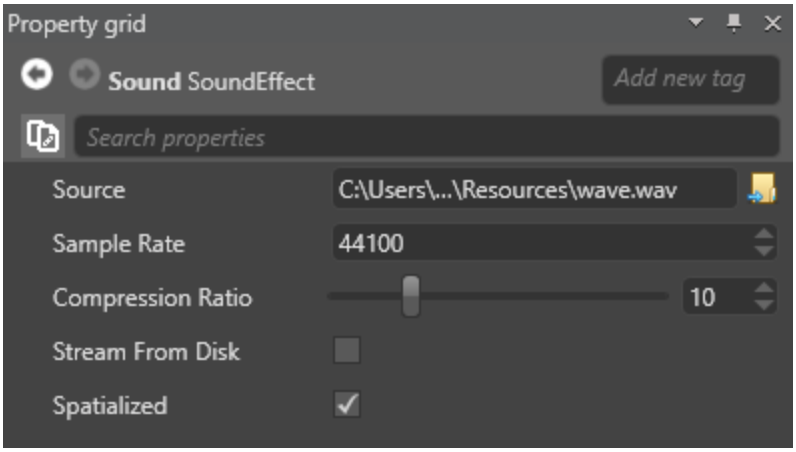
Stride adds the audio tracks from the video to the **Asset View**.

See also

- Spatialized audio
- Non-spatialized audio
- Global audio settings
- Video

Audio asset properties

After you select an audio asset in the **Asset View**, you can configure its properties in the **Property Grid**.



Property	Description
Source	The source audio file (note that Stride never alters source files)
Compression ratio	Set the compression rate from `1` (no compression) to `40` (maximum). Greater compression optimizes memory use, but decreases audio quality. Stride compresses audio files with the open-source Opus/Celt codec.
Sample rate	The rate at which Stride resamples the source file. The higher the sample rate, the higher the audio quality. Typical sample rates are 44.1 kHz (44,100 Hz), 48 kHz, 88.2 kHz, and 96 kHz. Note that high sampling rates doesn't improve the quality of low-quality audio files.
Spatialized	Simulate 3D audio (see spatialized audio)
Stream from disk	Streaming is useful for larger audio files, as it saves memory. For more information, see Stream audio.

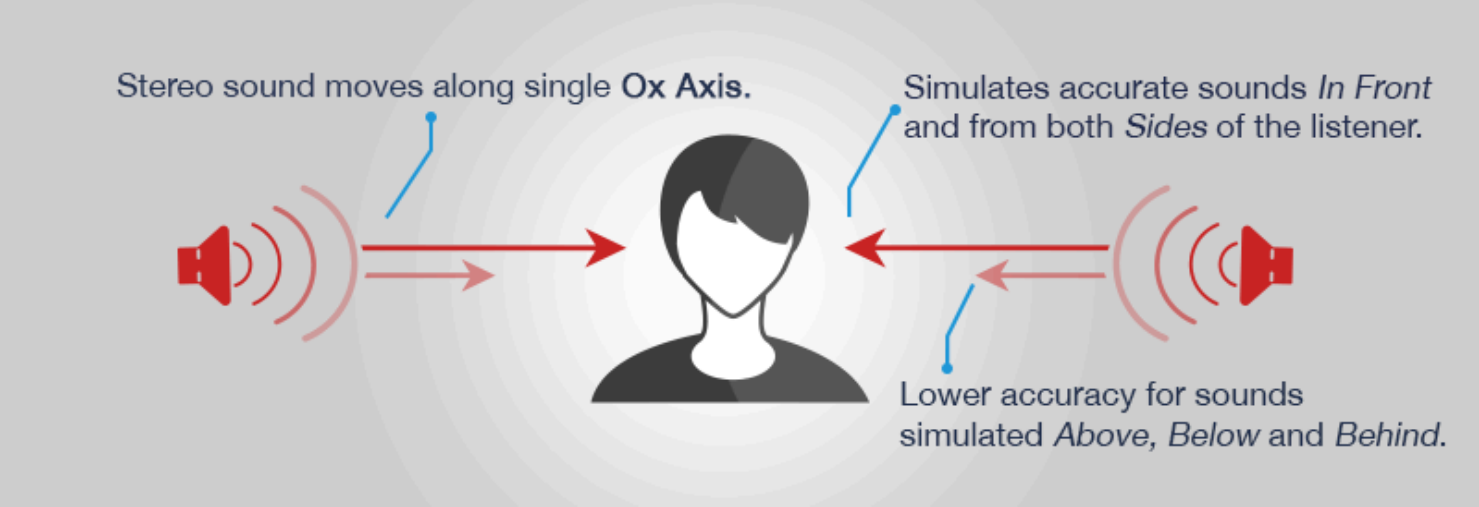
See also

- Import audio
- Global audio settings
- Spatialized audio
- Non-spatialized audio

Non-spatialized audio

Beginner Programmer

Non-spatialized audio sounds the same throughout the scene, regardless of the position of entities (such as the player camera). It's stereo and moves along a single axis (usually the X-axis). Unlike spatialized audio, the *volume*, *pitch* (*frequency*), and other parameters of spatialized audio don't change. This is useful, for example, for background music and menu sound effects.



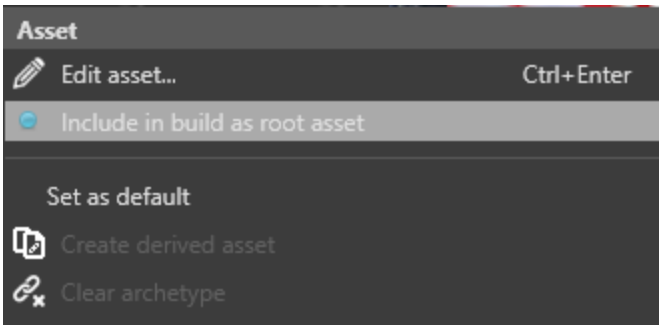
Non-spatialized audio requires no audio emitters or audio listeners.

1. Import audio and include it in the build

1. Import the audio as a audio asset.

2. Make sure the audio asset is a **root asset**. Root assets are assets that Stride includes in the build so they can be used at runtime.

   In the **Asset View**, right-click the asset and select **Include in build as root asset**:

   If the menu option reads **Do not include in build as root asset**, the option is already selected and you don't need to change it.

2. Create a script to play audio

To play non-spatialized audio at runtime, create an instance of it and define its behavior in the code.

The SoundInstance controls audio at runtime with the following properties:

Property	Function
IsLooping	Gets or sets looping of the audio.
Pan	Sets the balance between left and right speakers. By default, each speaker a value of 0.
Pitch	Gets or sets the audio pitch (frequency).
PlayState	Gets the state of the SoundInstance.
Position	Gets the current play position of the audio.
Volume	Sets the audio volume.

For more details, see the SoundInstance API documentation.

> **NOTE**

If the sound is already playing, Stride ignores all additional calls to SoundInstance.Play. The same goes for SoundInstance.Pause (when a sound is already paused) and SoundInstance.Stop (when a sound is already stopped).

For example, the following code:

- instantiates non-spatialized audio
- sets the audio to loop
- sets the volume
- plays the audio

```
public override async Task Execute()
{
    // Load the sound
    Sound musicSound = Content.Load<Sound>("MySound");

    // Create a sound instance
    SoundInstance music = musicSound.CreateInstance();

    // Loop
    music.IsLooping = true;
```

```
    // Set the volume
    music.Volume = 0.25f;

    // Play the music
    music.Play();
}
```

Alternative: create a script with public variables

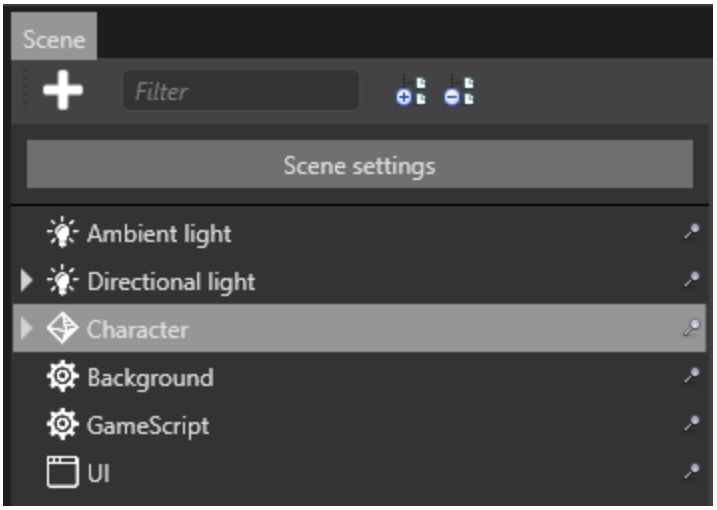
Create a public variable for each audio asset you want to use. You can use the same properties listed above.

For example:

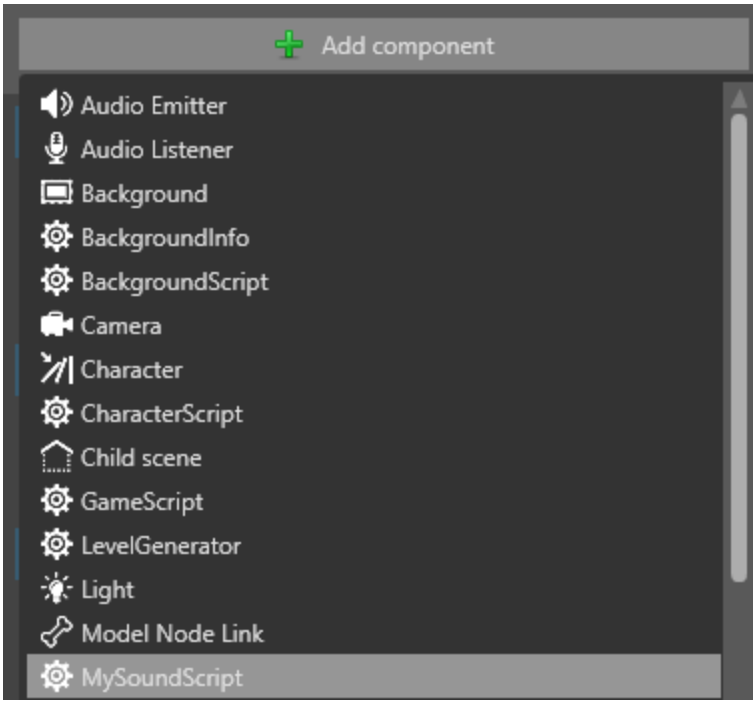
```
public class MySoundScript : SyncScript
{
    public Sound MyMusic;

    private SoundInstance musicInstance;
    public bool PlayMusic;

    public override void Start()
    {
        musicInstance = MyMusic.CreateInstance();
    }

    public override void Update()
    {
        // If music isn't playing but should be, play the music.
        if (PlayMusic & musicInstance.PlayState != PlayState.Playing)
        {
            musicInstance.Play();
        }

        // If music is playing but shouldn't be, stop the music.
        else if (!PlayMusic)
        {
            musicInstance.Stop();
        }
    }
}
```

Add the script to the entity

1. In the **Scene view**, select the entity you want to add the script to:



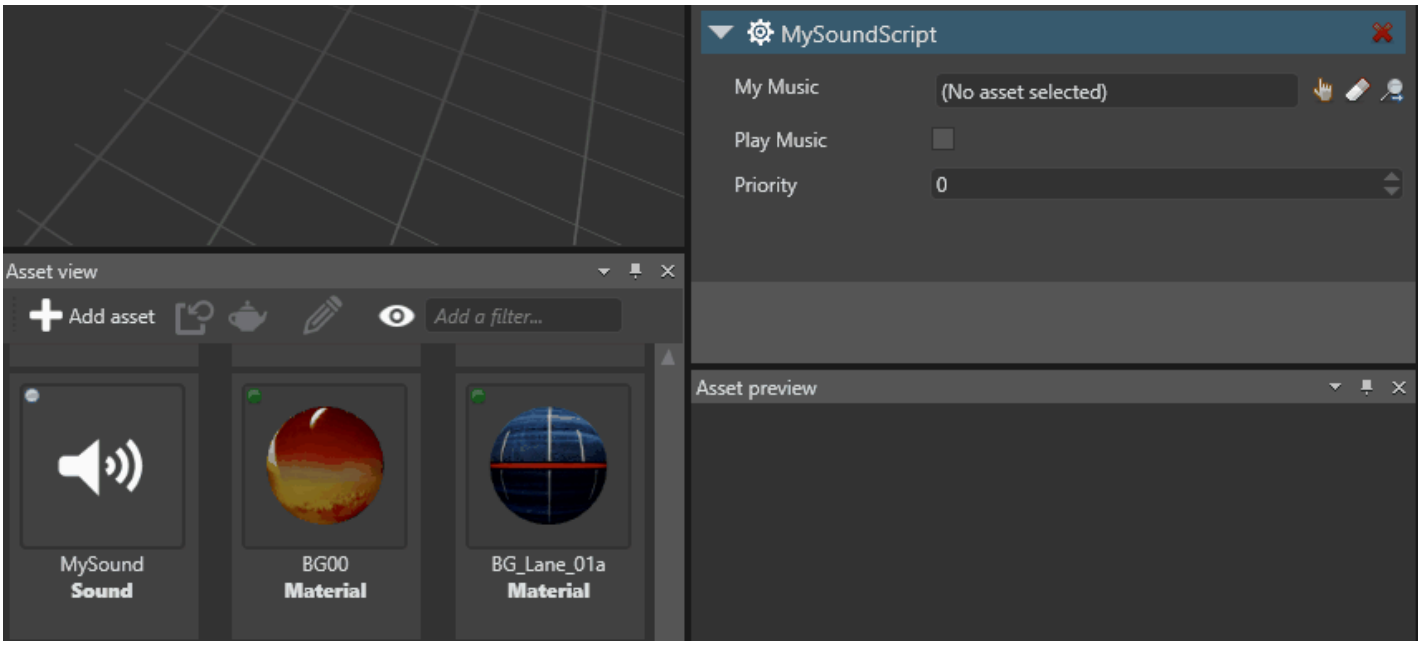
2. In the **Property Grid**, click **Add component** and select your script:



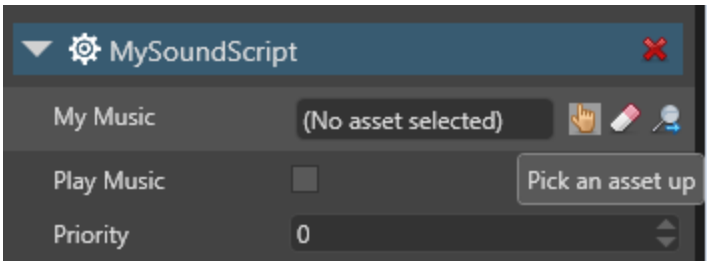
   The script is added to the entity.

3. If you added **public variables** to the script, you need to tie them to audio assets.

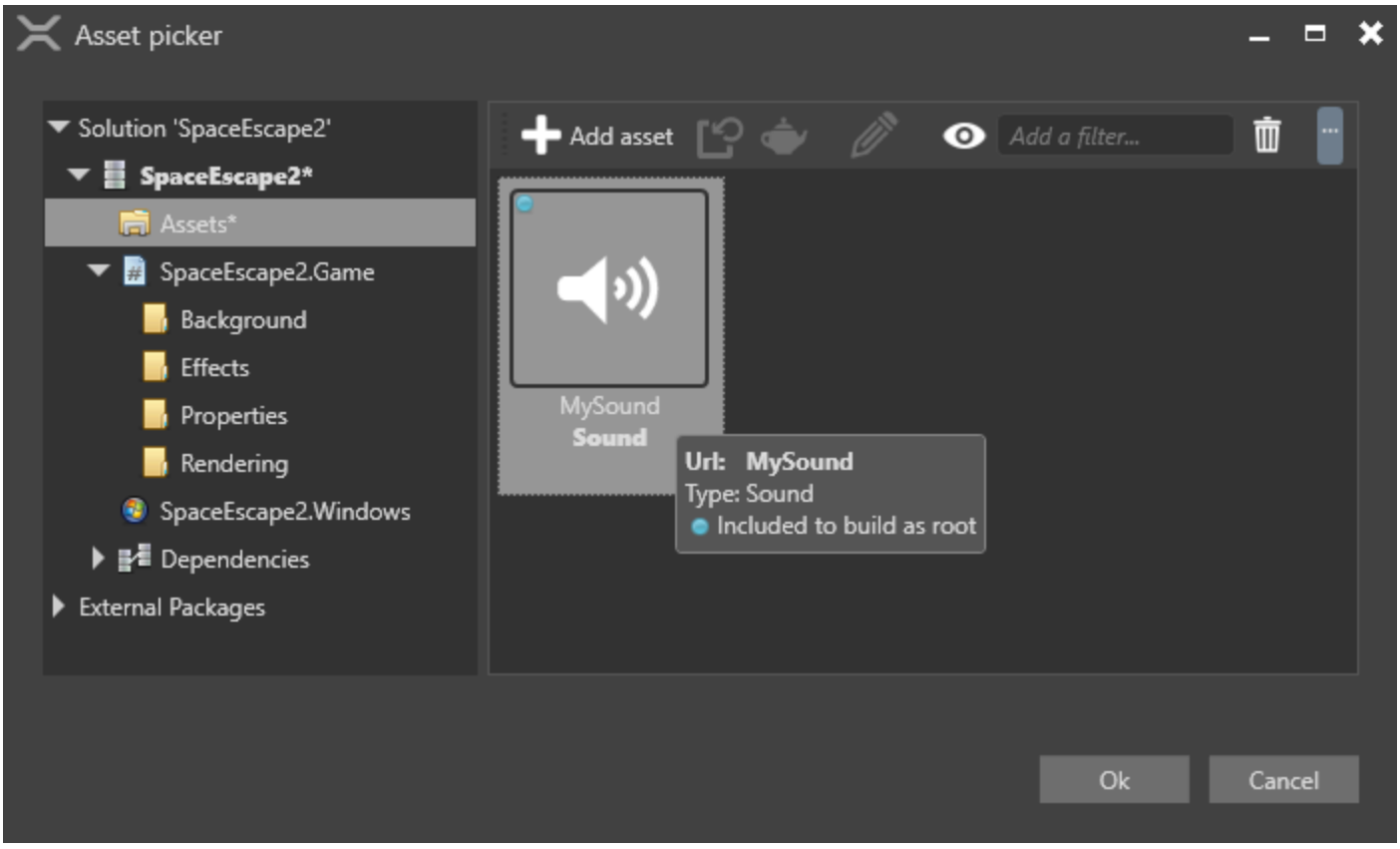
   Drag and drop an asset from the **Asset View** to each variable:



Alternatively, click (**Select an asset**):



Then choose the audio asset you want to use:



See also

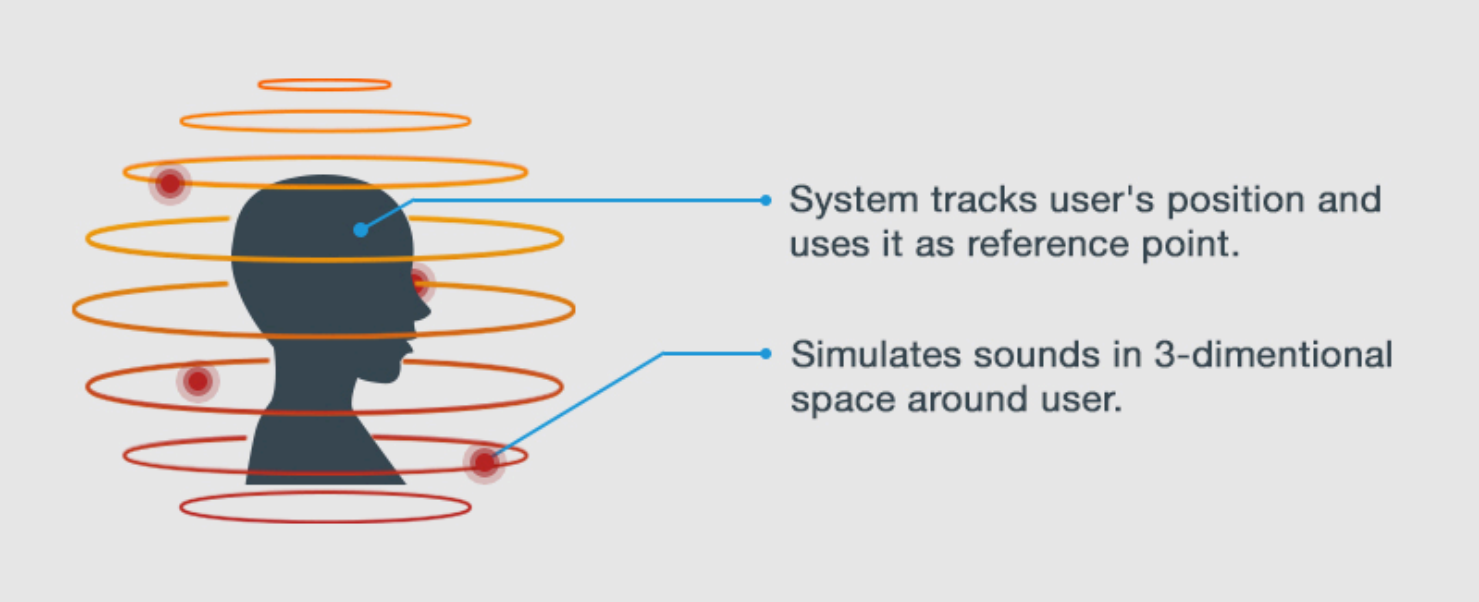
- Import audio
- Global audio settings
- Spatialized audio

Spatialized audio

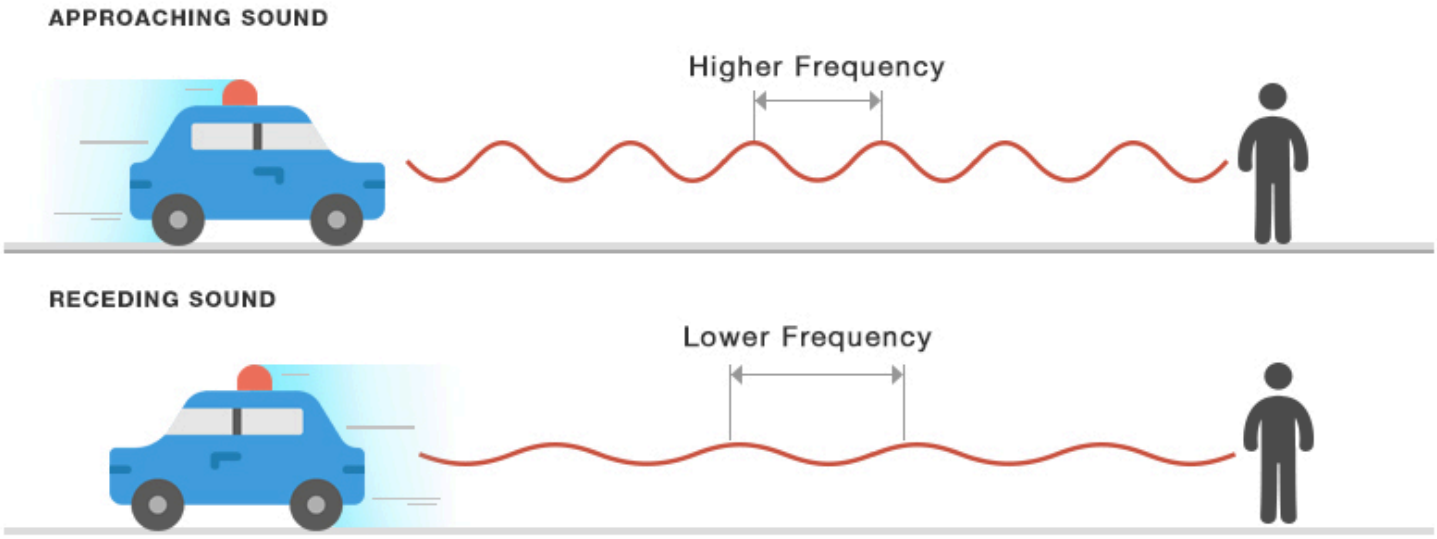
Beginner Designer Programmer

Spatialized audio, also called **3D audio**, simulates three-dimensional sound. This creates more realistic audio than non-spatialized audio.

In real life, our experience of sound is affected by factors including its volume, the surrounding area (such as a cave or small room), and the position and movement of the sound source. We can usually tell approximately where a sound is coming from and whether it's moving.



For example, the frequency (pitch) of the sound coming from a moving object varies depending on the observer's position (the Doppler effect). Sound from an approaching source has a higher frequency than sound from a receding source:



To simulate realistic 3D audio, Stride tracks the positions of two entities in the scene:

- **audio emitters**, which emit audio
- **audio listeners**, which hear the sound emitted by audio emitters

You must have both audio emitters and audio listeners to hear spatialized sound in a scene.

Spatialized audio is widely used for sound effects in platform, desktop, and VR games. For example, a gun might make a gunshot sound when fired, or a character might make a footstep sound when they take a step.

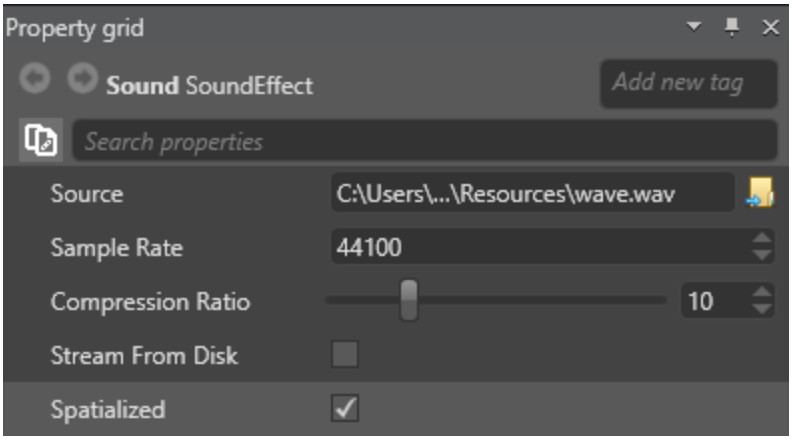
> **NOTE**
> Spatialized audio uses more CPU than non-spatialized audio.

Enable spatialized audio

When you import your audio, select *Spatialized Sound* as the asset type.

You can also set audio to spatialized in the asset's **Property Grid**:

1. In **Asset View**, select *Audio Asset*.
2. In the **Property Grid**, select the **Spatialized** checkbox:



> **NOTE**
> Stride processes spatialized audio as mono (single-channel) audio. It doesn't alter the source file.

See also

- Audio emitters

- Audio listeners
- HRTF
- Global audio settings

Audio emitters

Beginner Programmer Designer

Audio emitter components emit audio used to create spatialized audio. You can add them to any entity.

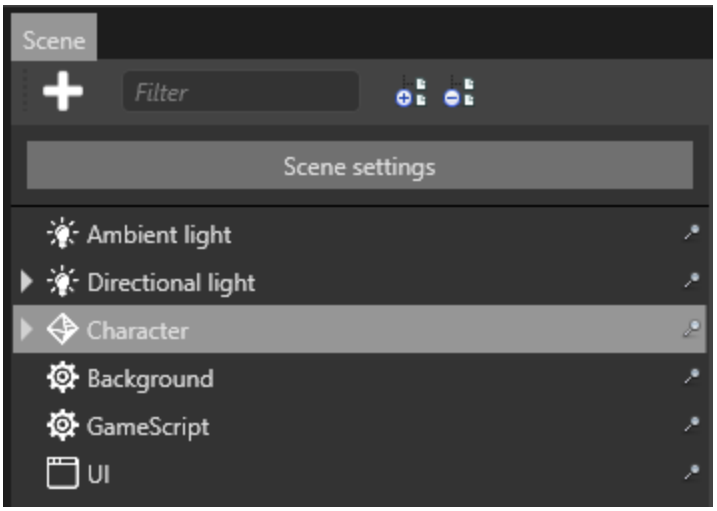
The pitch and volume of the sound changes as the audio listener moves closer to and away from the audio emitter.

> **NOTE**

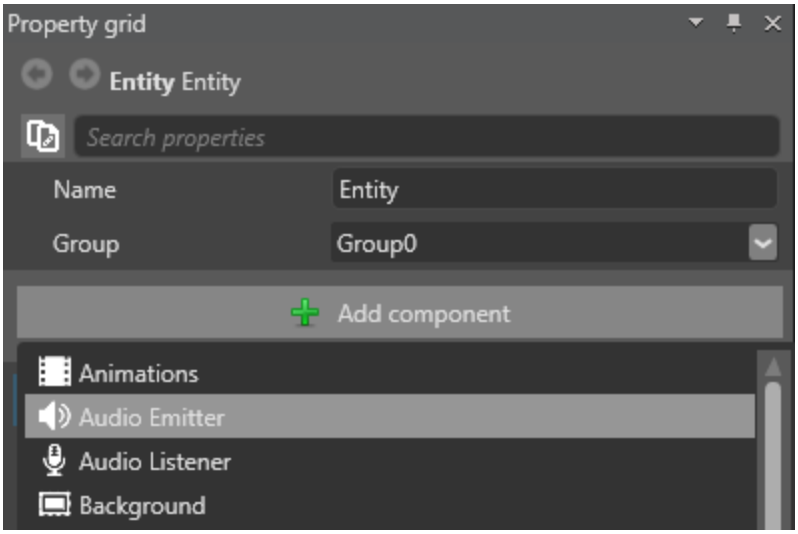
You need at least one AudioListenerComponent in the scene to hear audio from audio emitters.

1. Set up an audio emitter asset

1. In the **Scene view**, select an entity you want to be an audio emitter.

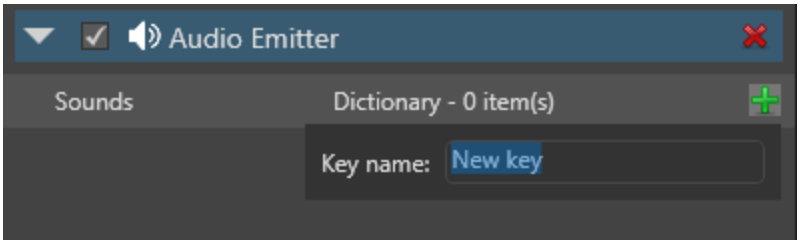


2. In the **Property Grid**, click **Add component** and select **Audio Emitter**.

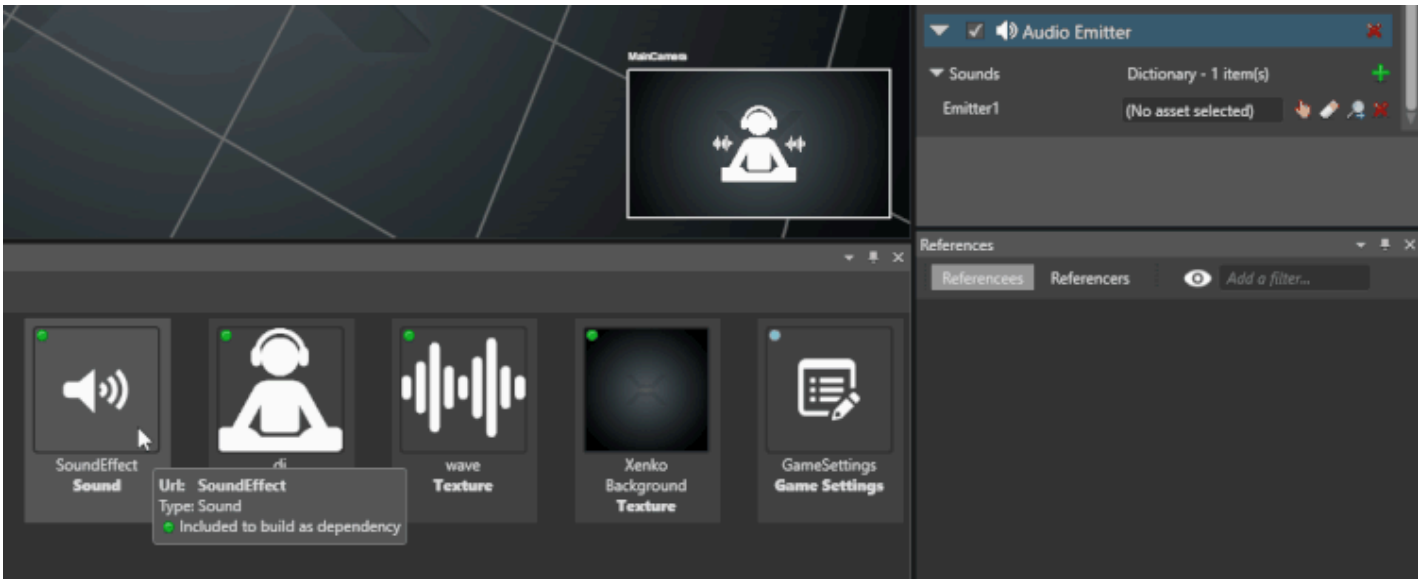


Now we need to add audio to the emitter.

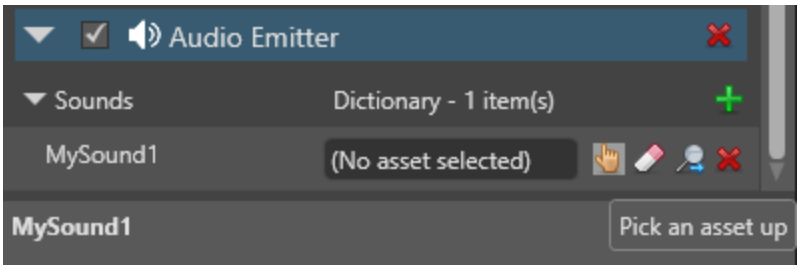
3. Under **Audio Emitter**, click (**Add**) and specify a name for the audio.



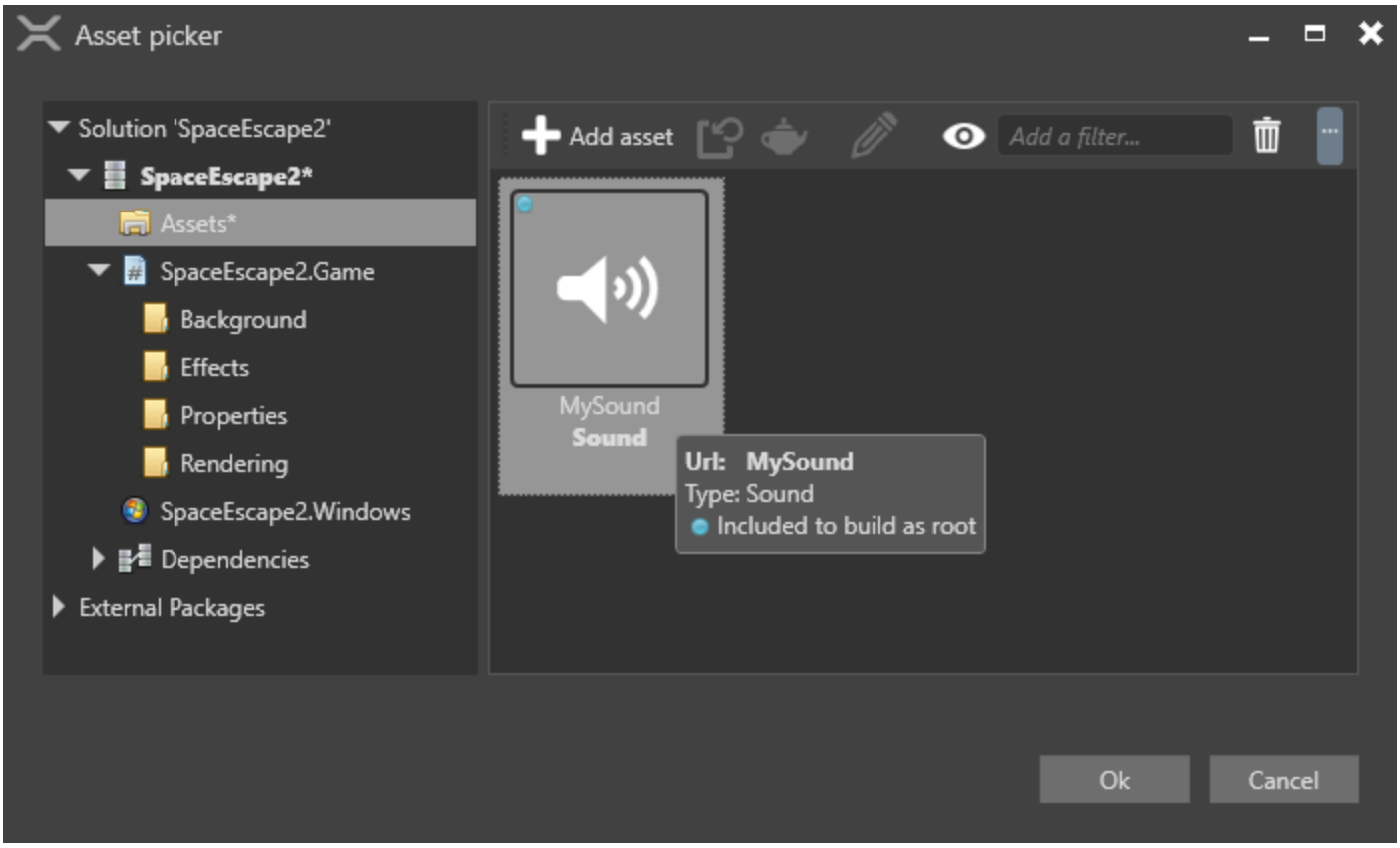
4. From the **Asset View**, drag and drop an audio asset to the audio you just added:



Alternatively, click (**Select an asset**).

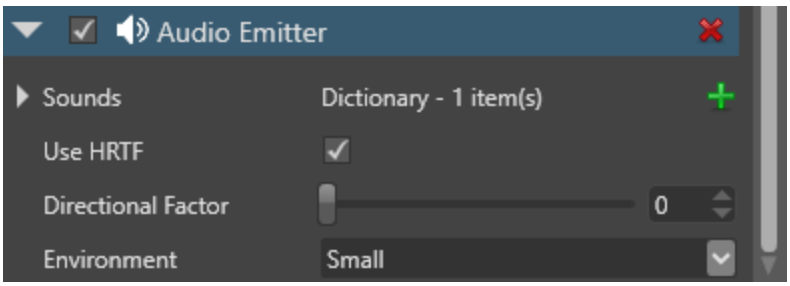


Then choose an audio asset:



5. Repeat steps 3 and 4 to add as many audio assets as you need.

6. Configure the properties for this audio emitter.



Property	Description
Use HRTF	Enable head-related transfer function (HRTF). With this enabled, sounds appear to come from a specific point in 3D space, synthesizing binaural audio. For more information, see HRTF.
Directional factor	How directional the audio is, from 0 (min) to 1 (max). If set to 0, the audio is emitted from all directions. You can control this with a slider or number value.
Environment	The reverb type for the audio, simulating reverberation of real environments (small, medium, large, or outdoors).

2. Create a script to play the audio

Now we need to create a script to play and configure the audio asset.

1. In your script, instantiate AudioEmitterSoundController for each sound you want to use in the script.

   For example, say we have two sounds, **MySound1** and **MySound2**:

```
AudioEmitterComponent audioEmitterComponent = Entity.Get<AudioEmitterComponent>();
AudioEmitterSoundController mySound1Controller = audioEmitterComponent["MySound1"];
AudioEmitterSoundController mySound2Controller = audioEmitterComponent["MySound2"];
```

2. Use the following AudioEmitterSoundController properties and methods to play and configure the audio:

Property / method	**Description**
IsLooping	Loops audio. Has no effect if PlayAndForget() is set to true.
Pitch	Gets or sets sound pitch (frequency). Use with caution for spatialized audio.
PlayState	Gets the current state of the audio emitter sound controller.
Volume	Volume of the audio.
Pause()	Pauses audio.
Play()	Plays audio.
PlayAndForget()	Plays audio once, then clears the memory. Useful for short sounds such as gunshots. Overrides IsLooping.
Stop()	Stops audio.

For example:

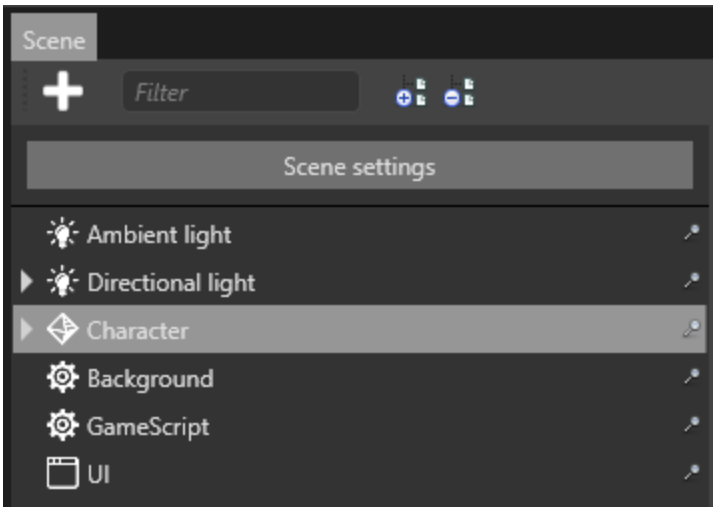
```
mySound1Controller.IsLooping = true;
mySound1Controller.Pitch = 2.0f;
mySound1Controller.Volume = 0.5f;
mySound1Controller.Play();
```

This sound will loop at double the original pitch and half the original volume. For more information, see the AudioEmitterSoundController Stride API documentation.

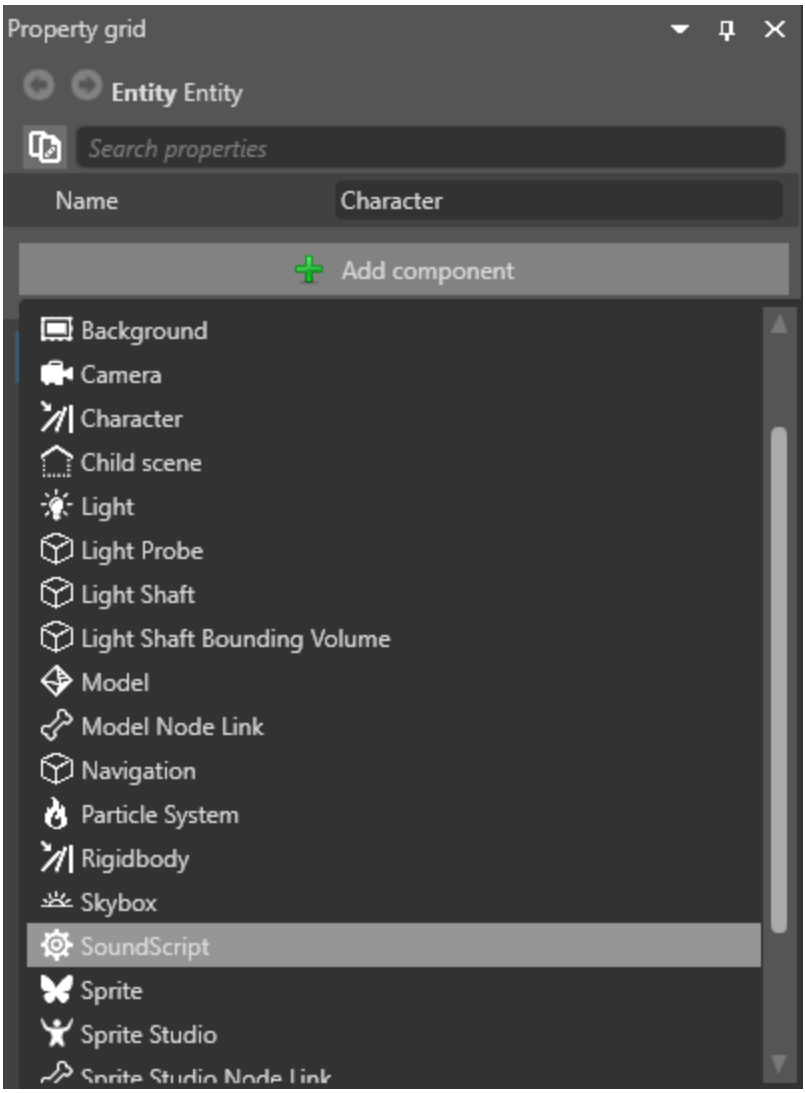
3. Add the script to the audio emitter entity

Game Studio lists the script as a component under **Add component**. Add the script to the audio emitter entity.

1. In the **Scene view**, select an entity you want to be an audio emitter.



2. Click **Add component** and select the script.



See also

- Spatialized audio
- Audio listeners
- Global audio settings

Audio listeners

Beginner Designer

An **audio listener** is an entity that listens for audio emitted by audio emitters to create spatialized audio. There can be multiple audio listeners in a scene. This is common, for example, in multiplayer games, where each player camera is an audio listener.

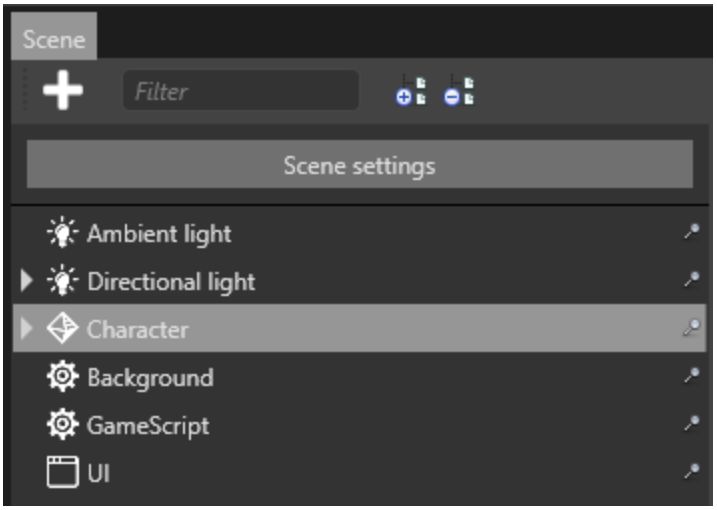
You don't need to configure audio listeners. All settings for sound effects, including *Volume* and *Pitch* (*Frequency*), are configured on the audio emitter.

If there's no audio listener in the scene, you won't hear audio from audio emitters.

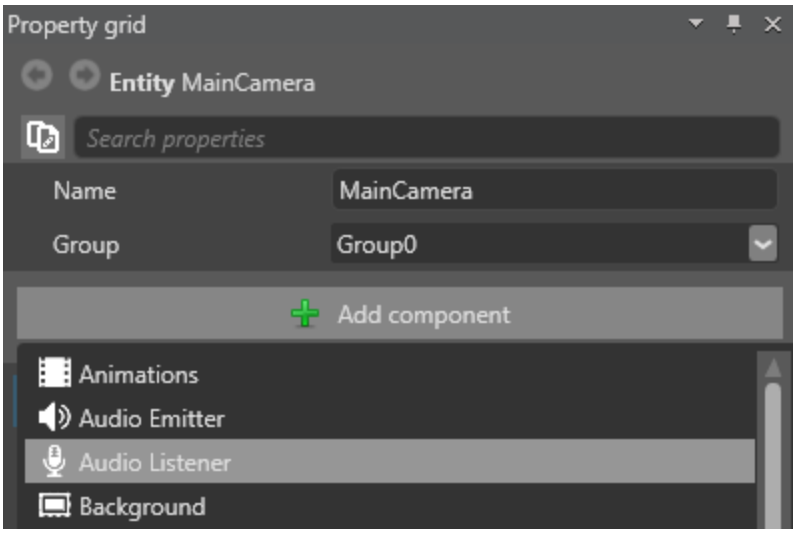
Add an audio listener component to an entity

To create an audio listener, attach an **audio listener component** to an entity. You can attach this component to any entity.

1. In **Scene view**, select the entity you want to be an audio listener:



2. In the **Property Grid**, click *Add Component* and select Audio listener component:



The entity is now an audio listener.

> **WARNING**

On iOS, you can create multiple objects with Audio listener component in a scene, but only one is used at runtime.

See also

- Spatialized audio
- Audio emitters
- Global audio settings

Head-related transfer function (HRTF) audio

Head-related transfer function (HRTF) is an advanced way of rendering audio so that sounds appear to come from a specific point in 3D space, synthesizing binaural audio. It provides more realistic audio than standard spatialized audio. For example, with HRTF, the player can hear whether a character is above or below them. This is particularly useful for VR applications, as it increases immersion.

Players don't need special hardware to use HRTF. However, the effect works much better with headphones than speakers.

This video demonstrates the effect of HRTF audio:

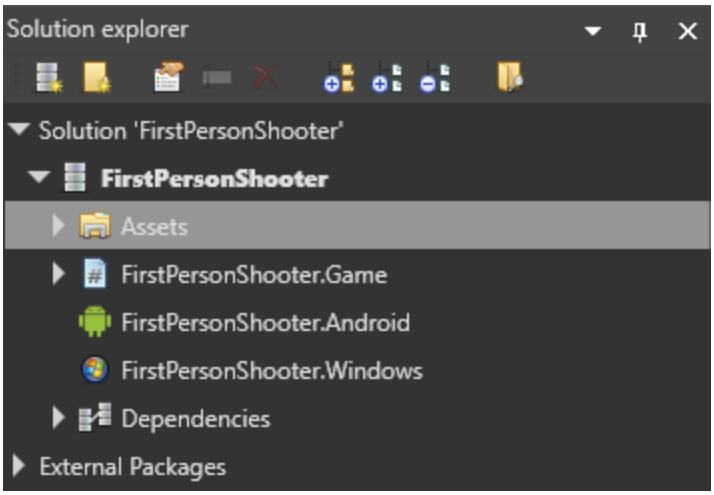
> **NOTE**
> For now, you can only use HRTF on Windows 10.

Enable HRTF

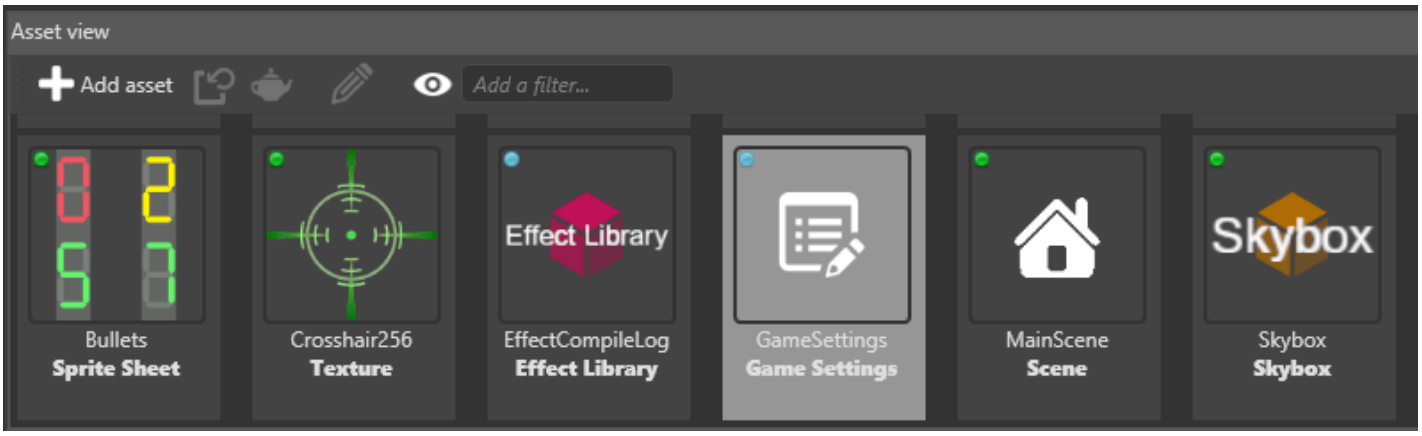
To use HRTF, first enable it globally in the **Game Settings** asset, then enable HRTF on the entities you want to use it with.

1. Enable HRTF globally

1. In **Solution explorer** (the bottom-left pane by default), select the **Assets folder**.



2. In the **Asset View** (the bottom pane by default), select the **GameSettings** asset.



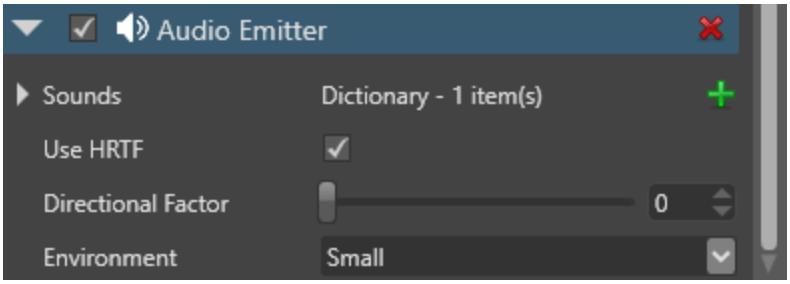
3. In the **Property Grid** (the right-hand pane by default), under **Audio settings**, select **HRTF support**.

For more information about the Game Settings asset, see Game settings.

2. Enable HRTF on the entities

1. Select the entity with the audio emitter that contains the sound you want to enable for HRTF.

2. In the **Property Grid** (on the right by default), under **Audio emitter**, select **Use HRTF**.



Sounds emitted from this entity will use HRTF.

> **NOTE**
>
> The HRTF option applies to every sound emitted from this audio emitter.

For more information about audio emitters, including the properties you can change, see Audio emitters.

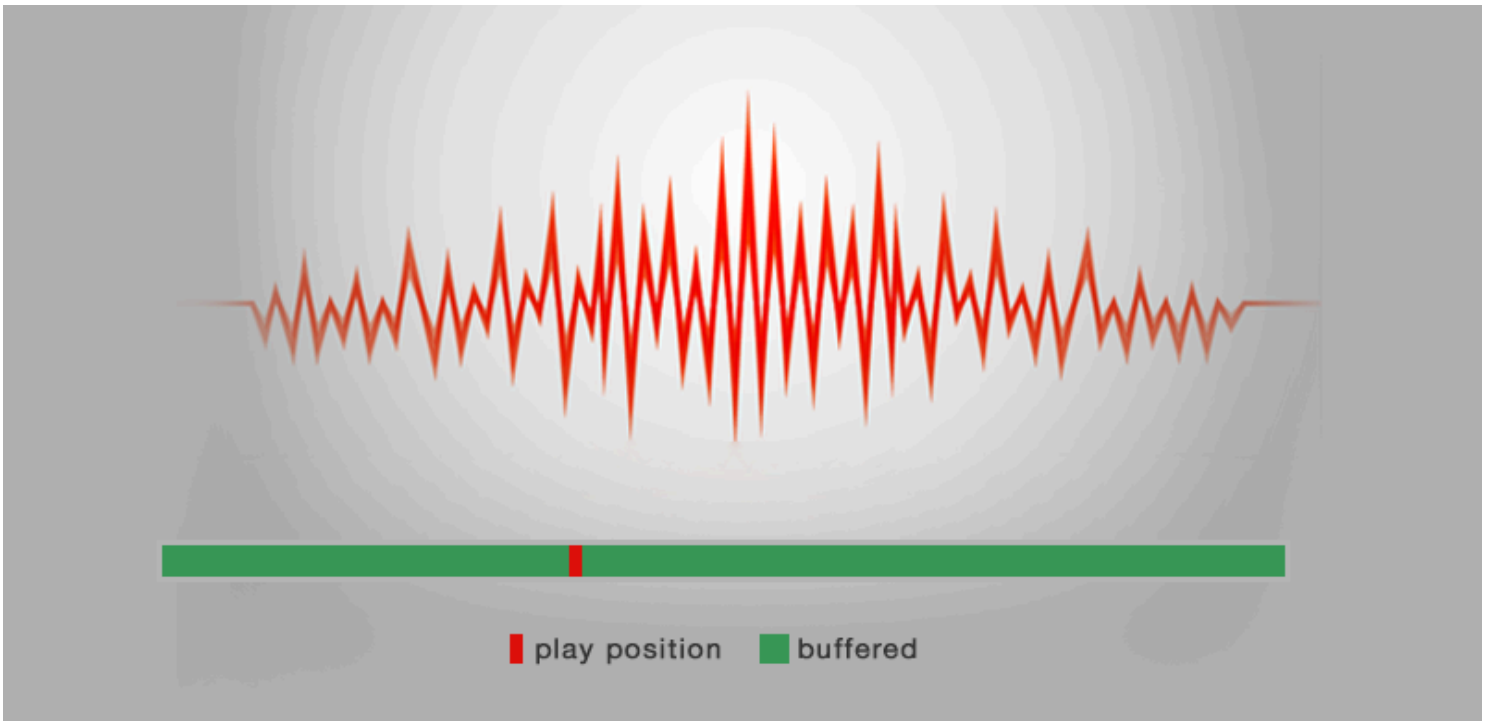
See also

- Head-related transfer function (Wikipedia)
- Spatialized audio
- Audio emitters
- Audio listeners
- Game settings

Stream audio

Beginner Designer Programmer

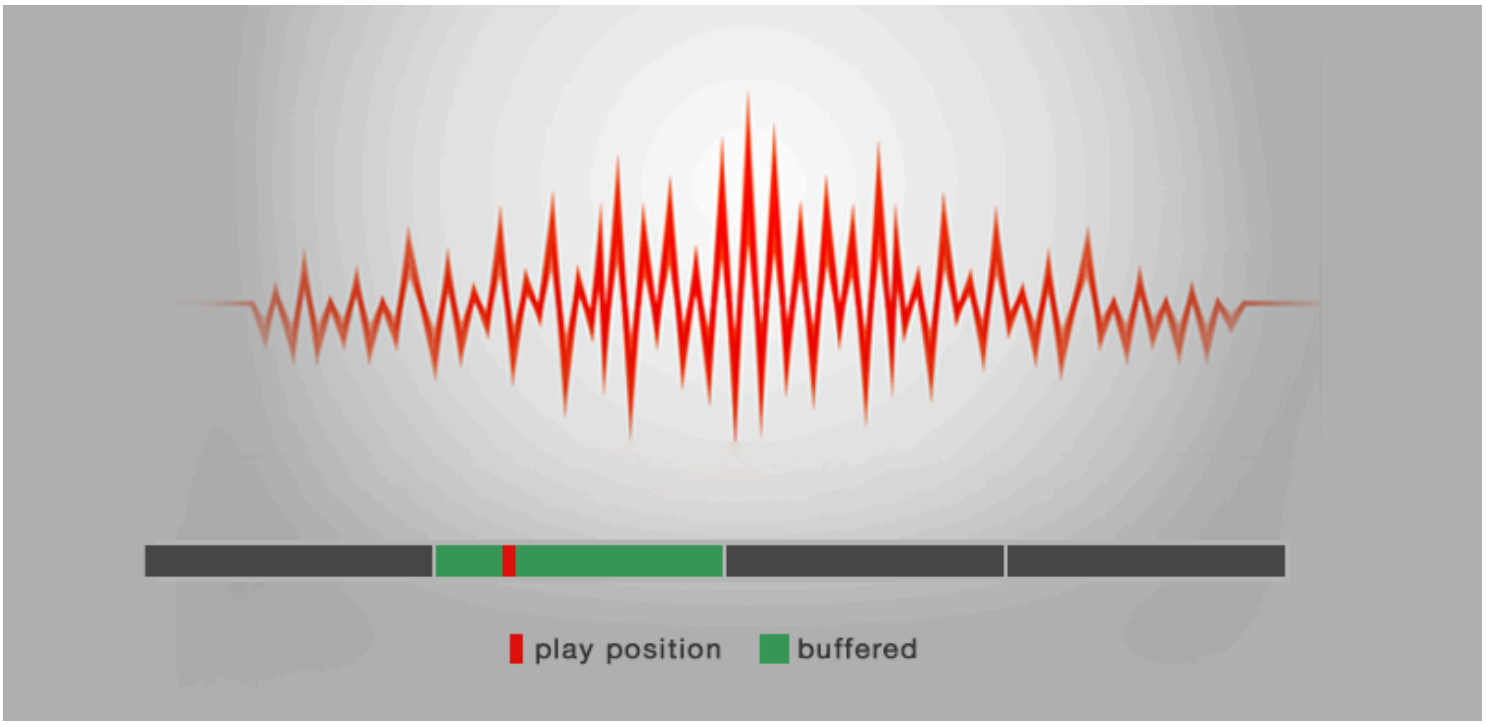
By default, Stride plays audio directly from memory. This is useful for short sound effects such as gunshots or footsteps.



Alternatively, Stride can buffer audio and stream it in sequences. As soon as the first sequence is buffered, Stride plays it while buffering the following sequences in parallel. This saves a lot of memory when used for larger audio files such as background music and character dialogue.

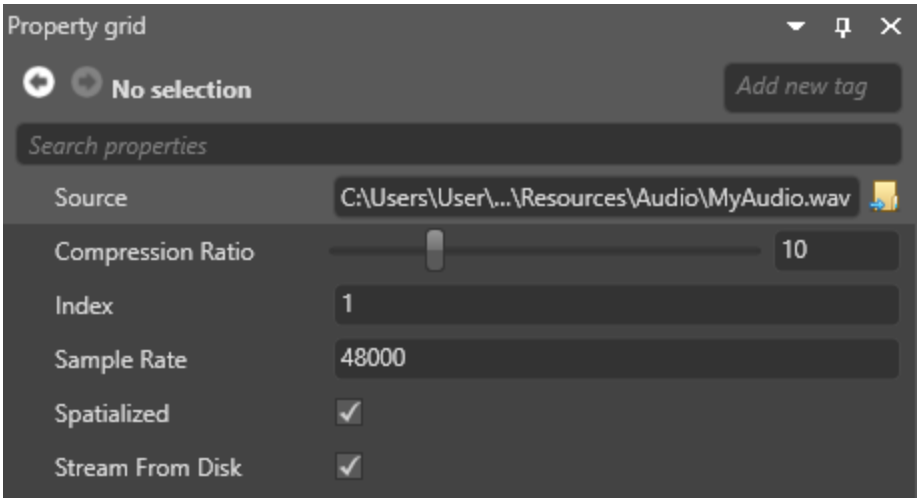
> **NOTE**

Streaming audio increases latency unless you preload it with the ReadyToPlay task (see below).



To stream an audio asset:

1. In the **Asset View**, select the audio asset.
2. In the **Property Grid**, select **Stream From Disk**:



In the script for the asset, you can configure an audio file to play once the audio engine buffers enough audio samples. To do this, use this task:

```
SoundInstance music = musicSound.CreateInstance();
await music.ReadyToPlay();
music.Play();
```

See also

- Global audio settings
- Audio asset properties
- Spatialized audio
- Non-spatialized audio

Global audio settings

Beginner Programmer

Global audio settings apply to all the audio in your project.

You can control the global audio settings by accessing the AudioEngine **properties** class:

Property	Function
MasterVolume	Sets the master volume.
PauseAudio	Pauses all audio.
ResumeAudio	Resumes all audio.

You can also control sounds individually using the SoundInstance API.

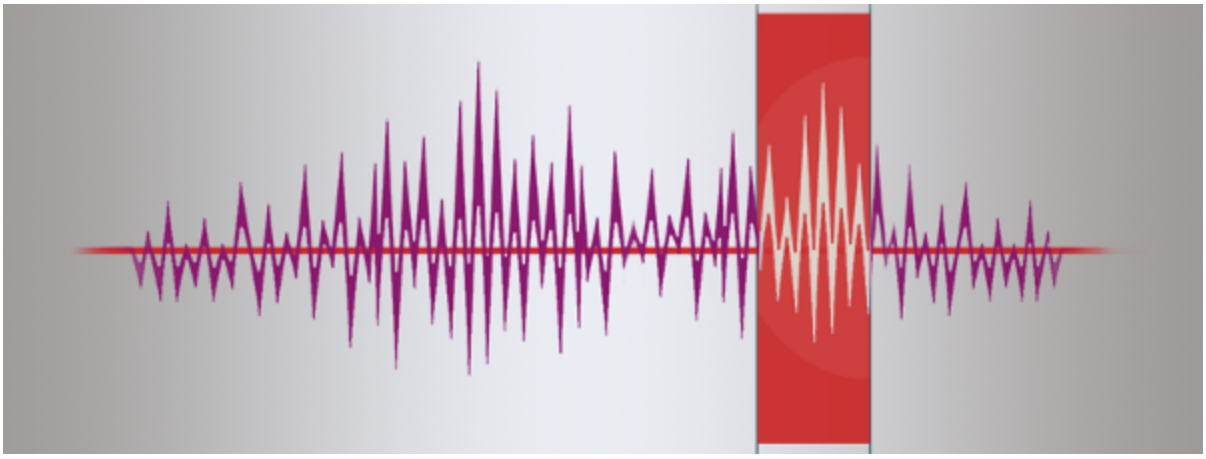
See also

- Spatialized audio
- Non-spatialized audio
- SoundInstance API documentation

Play a range within an audio asset

Intermediate Programmer

You can have Stride play only certain portions of an audio asset. This means, for example, that you can create multiple samples from a single audio asset by specifying different ranges in different Sound Instance objects.



You can use the following properties, methods, and structures:

Property, method, or structure	Function
TotalLength	The total length of the sound.
SoundInstance.SetRange(PlayRange)	Sets the time range to play within the audio asset.
PlayRange	Time information, including the range's starting point and length.
SoundInstance.Position	Gets the current play position as **TimeSpan**.

For example:

```
//Assume sample length is 4 seconds.
var length = mySound.TotalLength;
var begin = TimeSpan.FromSeconds(2);
var duration = TimeSpan.FromSeconds(2);
mySoundInstance.SetRange(new PlayRange(begin, duration));
```

See also

- Global audio settings
- Spatialized audio
- Non-spatialized audio

Custom audio data

Advanced Programmer

You can generate audio using your own mechanism. To do this, create a subclass of DynamicSoundSource. For an example of how to implement this, see the CompressedSoundSource` source code.

Example code

To play a custom DynamicSoundSource at runtime, use:

```
int sampleRate = 48000;
bool mono = false;
bool spatialized = false;
DynamicSoundSource myCustomSource = new MyCustomSource(...);
AudioListener listener = Audio.AudioEngine.DefaultListener;
AudioEngine audioEngine = Audio.AudioEngine;
SoundInstance myCustomInstance = new SoundInstance(audioEngine, listener, myCustomSource,
sampleRate, mono, spatialized);
await myCustomInstance.ReadyToPlay();
myCustomInstance.Play();
```

See also

- Global audio settings

Set an audio device

Advanced Programmer

You can set which audio device Stride uses. For example, you can access the *Oculus Rift* audio device from your custom game constructor.

If you don't specify a device, Stride uses the default audio advice.

Example code

This code sets the Oculus Rift device at runtime:

```
namespace OculusRenderer
{
    public class OculusGame : Game
    {
        public OculusGame()
        {
            var deviceName = OculusOvr.GetAudioDeviceFullName();
            var deviceUuid = new AudioDevice { Name = deviceName };
            Audio.RequestedAudioDevice = deviceUuid;
        }
    }
}
```

See also

- Global audio settings

Engine

> **WARNING**
>
> This section is out of date. For now, you should only use it for reference.

- Asset
- Entity-component system
- File system
- Build pipeline
- Asset introspection

See also

- Introduction to assets
- Scripts

Asset manager

> **WARNING**
>
> This section is out of date. For now, you should only use it for reference.

Assets

After creating your assets in Game Studio, @'Stride.Core.Serialization.Assets.AssetManager' is the class responsible for loading, unloading and saving assets.

Creating

You usually create assets directly in Game Studio.

Their URL will match the name (including folder) in Game Studio.

Examples of URLs:

- knight (user imports knight.fbx directly in main asset folder)
- level1/room1 (user creates level1 and import room1.fbx inside)

For more information, see Assets for more details.

Loading

Loading an asset should be done with the help of @'Stride.Core.Serialization.Assets.AssetManager' class:

```
// Load an asset directly from a file:
var texture = Content.Load<Texture>("texture1");

// Load a Scene asset
var scene = Content.Load<Scene>("scenes/scene1");

// Load an Entity asset
var entity = Content.Load<Entity>("entity1");
```

Note that loading an asset that has already been loaded only increment the reference counter and do not reload the asset.

Unloading

Unloading is also done using the AssetManager class:

```
Asset.Unload(asset);
```

Asset life time

Asset load and unload are working in pairs. For each call to 'load', a corresponding call to 'unload' is expected.

An asset is actually loaded only during the first call to 'load'. All subsequent calls only result to an asset reference increment.

An asset is actually unload only when the number of call to unload match the number of call the load.

The @'Stride.Core.Serialization.Assets.AssetManager.Get' method returns the reference to a loaded asset but does not increment the asset reference counter.

```
 var firstReference = Content.Load<Texture>("MyTexture"); // load the asset and increase the reference counter (ref count = 1)

// the texture can be used here

var secondReference = Content.Load<Texture>("MyTexture"); // only increase the reference counter (ref count = 2)

// the texture can still be used here

Asset.Unload(firstReference); // decrease the reference counter (ref count = 1)

// the texture can still be used here

Asset.Get<Texture>("MyTexture"); // return the loaded asset without increasing the reference counter (ref count = 1)

// the texture can still be used here
Asset.Unload(secondReference); // decrease the reference counter and unload the asset (ref count = 0)

// The texture has been unloaded, it cannot be used here any more.
```

Asset bundles

> **WARNING**
> This section is out of date. For now, you should only use it for reference.

A bundle of assets allows to package assets into a single archive that can be downloaded into the game at a specific time.

It allows creation of **Downloadable Content (DLC)**.

Basic rules:

- A project can generate several bundle.
- A bundle is created from several **assets selectors** (Currently, only the `PathSelector` and `TagSelector` are supported)
- A bundle can have dependencies to others bundles
- Every bundle implicitly references `default` bundle, where every asset which shouldn't go in a specific bundle will be packaged
- Once a bundle is deployed into the game, all assets from this bundle and all its dependencies are accessible
- Bundle resolution is done through an asynchronous callback that allows you to download bundle, and will be called once per dependency (similar to AssemblyResolve event).

Create a bundle

> **NOTE**
> Creating currently requires some manual steps (i.e. editing `sdpkg` by hand).

Open the `sdprj` file of the game executable and add the following configuration:

Example:

- A bundle named `MyBundleName` will embed assets with tags `MyTag1` and `MyTag2`
- A bundle named `MyBundleName2` will embed assets with tags `MyTag3` and `MyTag4`. This bundle has a dependency to `MyBundleName`
- There is also a `PathSelector` which follow the `.gitignore` filtering convention.

```
Bundles:
 - Name: MyBundleName
   Selectors:
    - !TagSelector
      Tags:
        - MyTag1
        - MyTag2
 - Name: MyBundleName2
   Dependencies:
    - MyBundleName
   Selectors:
    - !TagSelector
      Tags:
        - MyTag3
        - MyTag4
    - !PathSelector
      Paths:
        - folder1/
        - /folder2/
        - *.bin
        - folder3/*.xml
```

NOTE

Asset dependencies are automatically placed in the most appropriate bundle.

Current process works that way:

- Find assets that matches specific Tag Selectors ("roots" of bundle assets).
- Enumerate assets that are dependent on those "roots" bundle assets and put them in the same bundle than their "roots" asset.
  - Except if already accessible through one of package dependencies (i.e. a shared dependent package or default package).
- Place everything else in default bundle.

Note that:

- Shared assets might be duplicated if not specifically placed in common or default package, but that is intended (i.e. if user wishes to distribute 2 separate DLC that need common assets but need to be self-contained).
- Every bundle implicitly depends on default bundle.

Load a bundle at runtime

Loading bundle is done through `ObjectDatabase.LoadBundle(string bundleName) (ref: {Stride.Core.Storage.ObjectDatabase.LoadBundle})`:

```
// Load bundle
Assets.DatabaseFileProvider.ObjectDatabase.LoadBundle("MyBundleName2");

// Load specified asset
var texture = Assets.Load<Texture2D>("AssetContainedInMyBundleName2");
```

Selectors

Selectors help deciding which assets are stored in a specific bundle.

Tag selector

Select assets based on a list of tag attached on each asset.

Properties:

- Tags: List of Tags. Any asset that contains at least one of the tag will be included.

Path selector

Select assets based on their path.

Standard .gitignore patterns are supported (except ! (negate), # (comments) and [0-9] (groups)).

Properties:

- Paths: List of filters. Any asset whose URL matches one of the filter will be included.

Asset control

> **WARNING**
> This section is out of date. For now, you should only use it for reference.

Until now, all assets of a game package, and its dependencies, were compiled as part of your game.

Starting with 1.3, we compile only the assets required by your game.

Don’t worry, most of it is done automatically for you! We do that by starting to collect dependencies from the new Game Setting asset: it references the Default Scene, and we can easily detect all the required asset references (Models, Materials, Asset referenced by your scripts and so on).

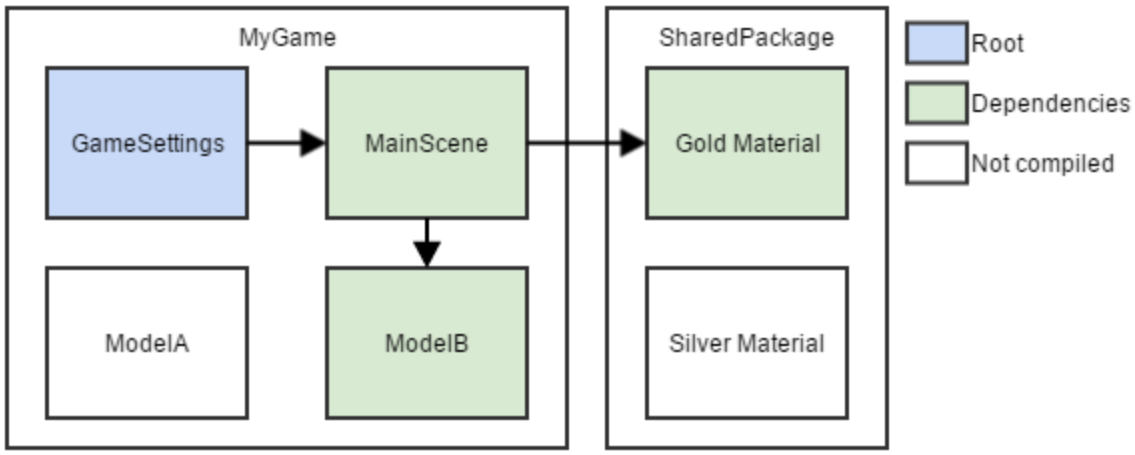
In case you were loading anything in your script using Content.Load, you can still tag those assets specifically with “Mark as Root” in the editor.

However, we now recommend to instead create a field in your script and fill it directly in the editor. All the samples have been updated to this new practice, so please check them out.

Which assets are compiled?

Assets that will be compiled and packaged in your project are:

- **Root assets (blue)**
  - **Automatic** for a few asset types (i.e. Game Settings, Shaders)
  - Explicit (using "**Mark as Root**" on the asset)
- **Dependencies of root assets (green)**
  - Since Game Settings is collected, that means that Default Scene and all its dependencies will be compiled as well (includes Model, Script field members pointing to other assets, etc...)
  - Also, we encourage our users to switch your script from Content.Load (which require "Mark as Root") to a field member that you can set within the editor using drag and drop. That will create an implicit dependency that will force that asset to be compiled as well.
- **Everything else (white)** (objects not marked as root and not referenced directly or indirectly by a root) **won't be packaged**



"Mark as root"

One important thing to understand is that "Mark as root" is not part of the asset, it is stored in the "current" package (the one that is in bold in the Solution Explorer).

It means that if "MyGame" is current package, if you check "Mark as Root" on Silver Material (part of SharedPackage), this information will be stored in MyGame.sdpkg as part of the reference to SharedPackage.

As a result, you can use a shared package from multiple games even if you have different explicit roots.

See also

For additional information about asset management, see Manage Assets

ECS (Entity Component System) Introduction

Problem

> `Dog` is a subclass of `Animal`.

This example is often used as an example of inheritance in introductions to programming. However, when things get more complex, we get problems:

- `Dog` and `Fish` can swim, so we create `SwimmingAnimal` as a class in between
- `Bee` and `Bird` can fly, so we create `FlyingAnimal`
- What do we now do with the `Duck`, who can do both?

We have the exact same problem in video games. Enemies can walk, shoot, fly - but not all of them can do everything. Even something basic like hitpoints is not universal, as some enemies are indestructible.

Solution

> Entity component system (ECS) is a software architectural pattern mostly used in video game development for the representation of game world objects. An ECS comprises entities composed from components of data, with systems which operate on entities' components.
> -*Wikipedia*

The general idea of an ECS is that an *entity* - an "object" in your virtual world - does not really do anything. It is mostly just a "bag of components".

The selection of components on an entity decides what it does. An entity with a collider component can collide, an entity with a sound component can make a noise, etc.

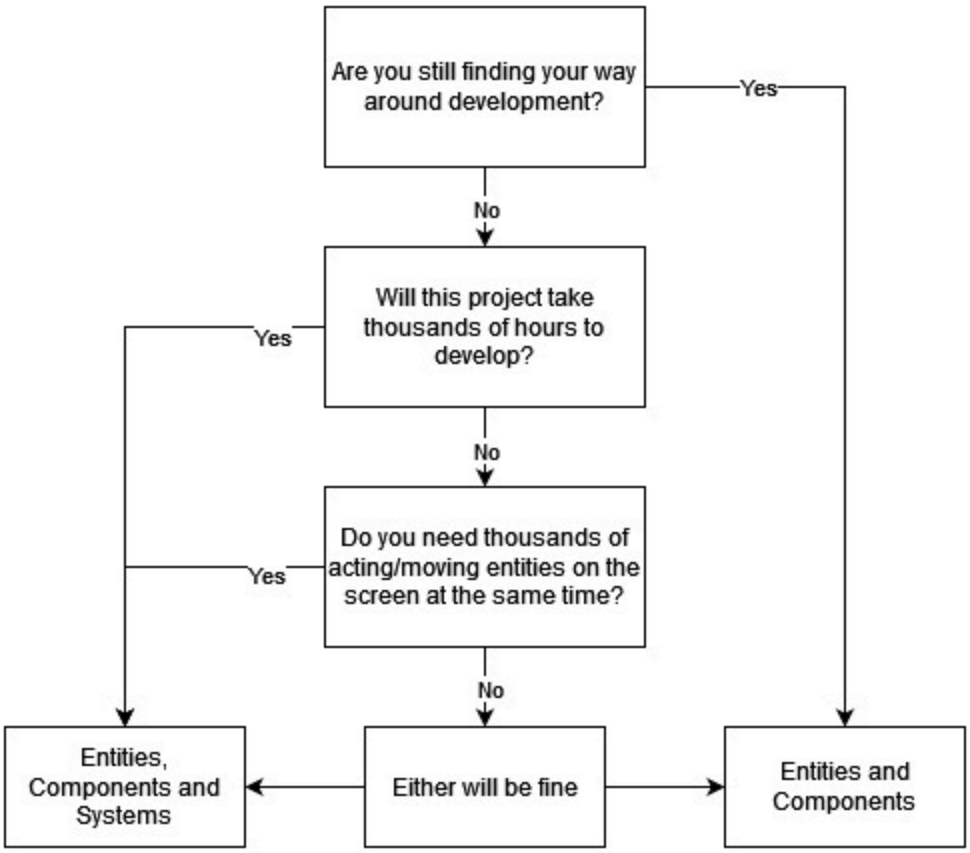
Differing opinions

For the **"System"** part of the term, there are two interpretations:

1. **Entity-and-Component System:** In this setup, the components contain both the data they need and the functionality that works with that data.
2. **Entity, Component, System:** In this arrangement, a component only contains data, while a third part - the system - contains the functionality.

Stride allows for working in both ways. **1)** can be achieved by using scripts while the usage of **2)** is described in this section of the manual.

Which one to choose?



ECS Usage

Classes

The three parts of **Entity Component System** map to the following classes:

- Entity - `Stride.Engine.Entity`
- Component - `Stride.Engine.EntityComponent`
- System - `Stride.Engine.EntityProcessor`

Minimal Setup

A component can be defined by deriving a class from `EntityComponent`. By adding the attribute `DefaultEntityComponentProcessor` to an `EntityComponent`, an `EntityProcessor` can be assigned to it. This will automatically set up and run the `EntityProcessor` if the `EntityComponent` is in the scene. An `EntityComponent` also needs to indicate that it can be serialized by adding the attribute `DataContract` to it. A system can be defined by deriving a class from `EntityProcessor`.

Code

Component

```
[DataContract(nameof(MyComponent))]
[DefaultEntityComponentProcessor(typeof(MyProcessor))]
public class MyComponent : EntityComponent
{
    public int MyValue { get; set; }
}
```

System

```
public class MyProcessor : EntityProcessor<MyComponent>
{
    public override void Update(GameTime time)
    {
        foreach (var myComponent in ComponentDatas.Values)
        {
            Console.WriteLine($"myComponent with value {myComponent.MyValue}
at {time.Total.TotalSeconds}");
        }
    }
}
```

Additional Note

An `EntityComponent` can currently not be drag-dropped onto an entity in Game Studio. It has to be added by selecting an entity, and then clicking the **Add component** button in the **Property grid**. Alternatively, this can also be done in code via `entity.Add(entityComponent)`.

Advanced Features

More Component Attributes

Display

By adding the `Display` attribute, a nicer name can be shown in Game Studio.

```
[Display("My better name")]
```

ComponentCategory

By default, your components will be listed in the category "Miscellaneous". By adding the `ComponentCategory` attribute, a different category can be chosen. If the chosen name does not exist yet, it will be added to the list in Game Studio.

```
[ComponentCategory("My own components")]
```

ComponentOrder

By adding the `ComponentOrder` attribute, the order in which components are listed in Game Studio can be changed.

```
[ComponentOrder(2001)]
```

Component Combinations

By passing the types of other components to the `EntityProcessor` constructor, it will only include entities *that also have those other components*. For example, the following `EntityProcessor` is for `MyComponent`, but will skip any entity that does not also have both `TransformComponent` and `AnimationComponent` on it.

```
public class MyProcessor : EntityProcessor<MyComponent>
{
    public MyProcessor() : base(typeof(TransformComponent), typeof(AnimationComponent))
    {
    }
}
```

Non-default Processors

Adding processors for a type of component via the attribute `DefaultEntityComponentProcessor` has been explained above. However, as the name implies, this is for the *default* processor. Non-default processors can also be added via

```
SceneSystem.SceneInstance.Processors.Add(entityProcessor);
```

Separation of EntityComponent and Data

`EntityProcessor<TComponent>` is a shortcut for `EntityProcessor<TComponent, TComponent>`. By explicitly using `EntityProcessor<TComponent, TData>` instead, a different type can be chosen for the actual data. This way, the `EntityComponent` can e.g. have "heavier" startup data and references, while the data object that needs to be processed every frame can be kept small. This will require overriding a method `GenerateComponentData`, which produces a `TData` instance from a `TComponent` instance.

Overrides

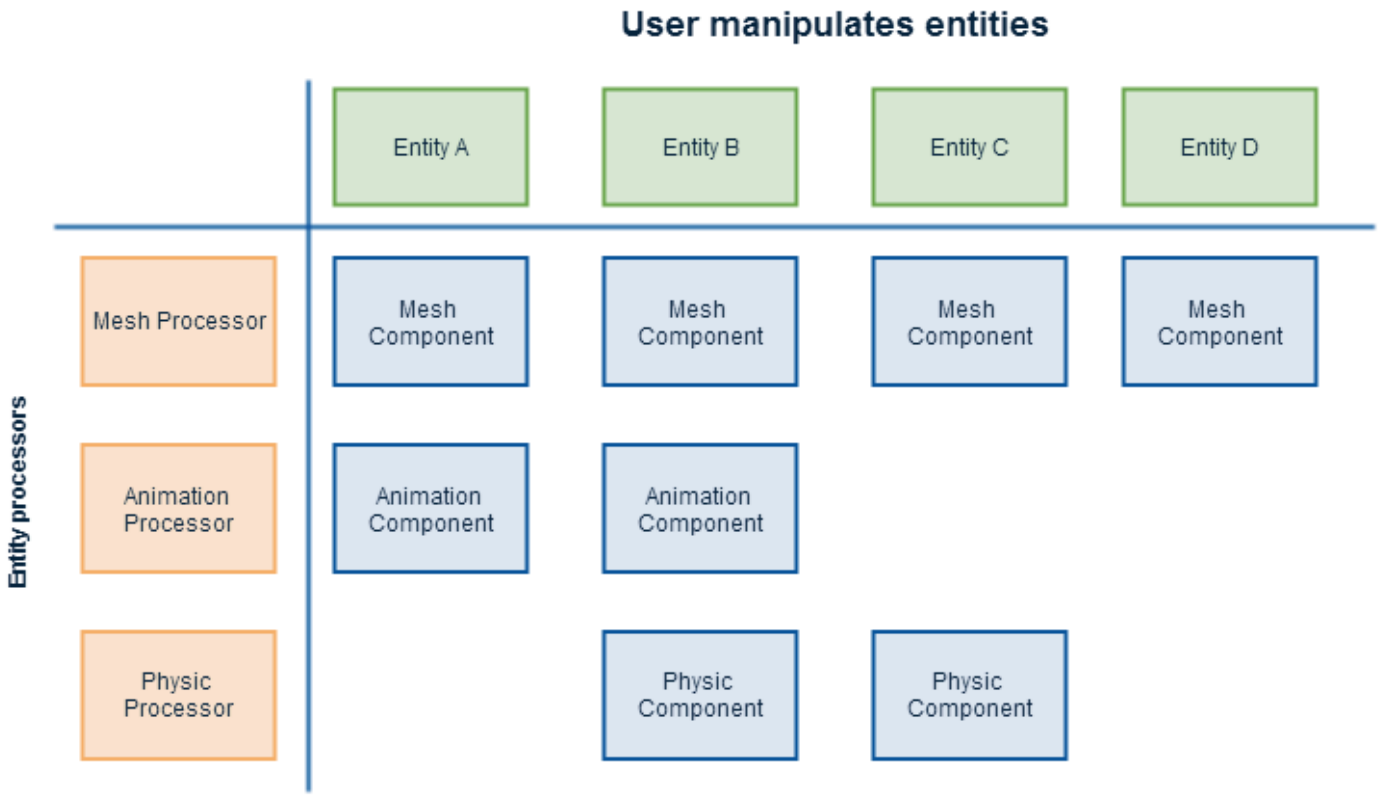
`EntityProcessor` also provides several methods which can be overridden in order to react to certain events. They are not overly complicated, so that their usage should be clear from their doc comments.

Manage entities

> **WARNING**
> This documentation is under construction.

Overview

User usually want to manipulate Component contained in a specific entity, while engine wants to access component by types (i.e. all Mesh Component while drawing, all animation components while updating animations, etc...):



User will add component-based entities into an entity manager.

Engine or user registers entity processors that can process specific entities and/or components.

Entity Processor

To solve this problem, the concept of Entity Processor has been added. An Entity Processor will access Entities that matches specific requirements (i.e. process all entities with MeshComponent) and process all of them in a single update function. This allows for greater efficiency and cache-friendliness, as there is no need to check every entity/components many times per frame.

This approach also solves many update order dependencies issues (just need to order the entity processors updates properly).

Here is some examples of entity processors:

- TransformProcessor: Compute transformation matrices from hierarchy and local transformation stored in TransformComponent.
  - As a result, EntityManager can be used as a hierarchical scenegraph instead of a simple entity list.
- ModelTransformProcessor: Add Model to rendering.
- LightProcessor: Collects and update lights, and transfer it to rendering system. It can hides implementation details (deferred or forward rendering, etc...)

Entity System

Entity are grouped together in an EntityManager. It will also contains the list of entity processors. When an entity is added or an entity components changes, it will ask entity processors if they should be included.

```
// Add an entity:
var myEntity = new Entity();
engine.EntityManager.AddEntity(myEntity);

// Iterate through added entities:
foreach (var entity in engine.EntityManager.Entities)
{
        Console.WriteLine(entity.Name);
}
```

EntityManager can be used to enumerate its `Entities (ref:{Stride.Engine.Entity})`. Note that children of a given entities will also be in this list.

To manipulate entities as a scenegraph, refer to TransformComponent class.

Flexible Processing

This document expects the reader to be familiar with ECS, please take a look at usage first.

Handling components through `EntityProcessor` may be too rigid in some cases, when the components you're trying to process cannot share the same base implementation for example.

`Stride.Engine.FlexibleProcessing.IComponent<TProcessor, TThis>` provides similar features to `EntityProcessor` while being more flexible on the component type, this document covers some of the usage of this particular interface.

The `IComponent` interface requires to type parameters,

- `TProcessor` which is your processor's type.
- And `TThis` which is your component's type.

While that last type may seem redundant, it is required to ensure your processor and your implemented type are compatible.

A summarised example satisfying those type constraint would look like so:

```
public class MyComponent : StartupScript, IComponent<MyComponent.MyProcessor, MyComponent>
{
    public class MyProcessor : IProcessor
    {
        public List<MyComponent> Components = new();

        public void SystemAdded(IServiceRegistry registryParam) { }
        public void SystemRemoved() { }

        public void OnComponentAdded(MyComponent item) => Components.Add(item);
        public void OnComponentRemoved(MyComponent item) => Components.Remove(item);
    }
}
```

The main difference compared to `EntityProcessor` is that `IComponent` is not limited to concrete types, your processor may operate on interfaces as well;

```
// Here, declaring the interface, which will be the type received by the processor
public interface IInteractable : IComponent<IInteractable.InteractableProcessor,
IInteractable>
{
    void Interact();
    public class InteractableProcessor : IProcessor
```

```
    {
        // Process each IInteractable here
        // Omitted method implementation for brievety
    }
}

// Now any component implementing IInteractable will be processed by
the InteractableProcessor
public class Button : StartupScript, IInteractable
{
    public void Interact(){}
}
public class Character : SyncScript, IInteractable
{
    public void Interact(){}
    public override void Update(){}
}
```

Updating Processors

Processors do not receive any updates by default, you have to implement the IUpdateProcessor or IDrawProcessor interface to receive them:

```
public interface ISpecialTick : IComponent<ISpecialTick.Processor, ISpecialTick>
{
    void Tick();

    public class Processor : IProcessor, IUpdateProcessor
    {
        public List<ISpecialTick> Components = new();

        public void SystemAdded(IServiceRegistry registryParam) { }
        public void SystemRemoved() { }

        public void OnComponentAdded(ISpecialTick item) => Components.Add(item);
        public void OnComponentRemoved(ISpecialTick item) => Components.Remove(item);

        // The execution order of this Update, smaller values execute first compared to
other IComponent Processors
        public int Order => 0;

        public void Update(GameTime gameTime)
        {
            foreach (var comp in Components)
                comp.Tick();
```

```
            }
        }
    }
```

Performance

While it is more flexible, processing components as interfaces instead of concrete class may introduce some overhead. If the system you're writing is performance critical you should look into strategies to elide or reduce virtual calls in your hot path.

File system

> **WARNING**
>
> This documentation is under construction.

We recommend you use the static class VirtualFileSystem to access files across platforms. It offers all basic operations such as reading, writing, copying, checking existence and deleting files.

> **NOTE**
>
> The path separator is / (Unix/Linux convention).

Code example

```
// Open a file through VirtualFileSystem
var gamesave1 = VirtualFileSystem.OpenStream("/roaming/gamesave001.dat",
VirtualFileMode.Open, VirtualFileAccess.Read);

// Alternatively, directly access the same file through its file system provider
(mount point)
var gamesave2 = VirtualFileSystem.ApplicationRoaming.OpenStream("gamesave001.dat",
VirtualFileMode.Open, VirtualFileAccess.Read);
```

Default mount points

Mount point	Description	Writable	Cloud	Notes	PC	Android
data	Application data, deployed by package	✗	✗		Output directory/data	APK itself
binary	Application binaries, deployed by package	✗	✗	Usually the same as *app_data* (except	Assembly directory	Assembly directory

Mount point	Description	Writable	Cloud	Notes	PC	Android
				on Android)		
roaming	User specific data (roaming)	✓	✓	Backup	Output directory/roaming, *%APPDATA%*	*$(Context.getFilesDi*
local	User application data	✓	✓	Backup	Output directory/local	$(Context.getFilesDi
cache	Application cache	✓	✗	DLC, etc. Might be deleted manually by user (restore, clear data, etc...)	Output directory/cache, with do-not-back-up flags	*$(Context.getFilesDi*
tmp	Application temporary data	✓	✗	Might be deleted without notice by OS	Output directory/temp, *%TEMP%/%APPNAME%*	*$(Context.getCacheL*

Build pipeline

This document describes the Build pipeline in Stride, its current implementation (and legacy), and the work that should be done to improve it.

Terminology

- An **Asset** is a design-time object containing information to generate **Content** that can be loaded at runtime. For example, a **Model asset** contains the path to a source FBX file, and additional information such as an offset for the pivot point of the model, a scale factor, a list of materials to use for this model. A **Sprite font asset** contains a path to a source font, multiple parameters such as the size, kerning, etc. and information describing in which form it should be compiled (such as pre-rasterized, or using distance field...). **Asset** are serialized on disk using the YAML format, and are part of the data that a team developing a game should be sharing on a source control system.

- **Content** is the name given to compiled data (usually generated from **Asset**s) that can be loaded at runtime. This means that in term of format, **Content** is optimized for performance and size (using binary serialization, and data structured in a way so that the runtime can consume it without re-transforming it). Therefore **Content** is the platform-specific optimized version of your game data.

Design

Stride uses *Content-addressable storage* to store the data generated by the compilation. The main concept is that the actual name of each generated file is the hash of the file. So if, after a change, the resulting content built from the asset is different, then the file name will be different. An index map file contains the mapping between the content *URL* and the actual hash of the corresponding file. Parameters of each compilation commands are also hashed and stored in this database, so if a command is ran again with the same parameters, the build engine can easily recover the hashes of the corresponding generated files.

Build Engine

The build engine is the part of the infrastructure that transforms data from the **assets** into actual **content** and save it to the database. It was originally designed to build content from input similar to a makefile. (eg. "compile all files in `MyModels/*.fbx` into Stride models). It has then been changed to work with individual assets when the asset layer has been implemented. Due to this legacy, this library is still not perfectly suited or optimal to build assets in an efficient way (dependencies of build steps, management of a queue for live-compiling in the Game Studio, etc.).

Builder

The `Builder` class is the entry point of the build engine. A `Builder` will spawn a given number of threads, each one running a `Microthread` scheduler (see `RunUntilEnd` method).

Build Steps

The `Builder` takes a root `BuildStep` as input. We currently have two types of `BuildStep`s:

- A `ListBuildStep` contains a sequence of `BuildStep` (Formerly we had an additional parent class called `EnumerableBuildStep`, but it has been merged into `ListBuildStep`). A `ListBuildStep` will schedule all the build steps it contains at the same time, to be run in parallel. Formerly we had a synchronization mechanism using a special `WaitBuildStep` but it has been removed. We now use `PrerequisiteSteps` with `LinkBuildSteps` to manage dependencies.
- A `CommandBuildStep` contains a single `Command` to run, which does actual work to compile asset.

> **TODO:** Currently, when compiling a graph of build steps, we need to have all steps to compile in the root `ListBuildStep`. More especially, if we have a `ListBuildStep` container in which we want to put a step A that depends on a step B and C, we need to put A, B, C in the `ListBuildStep` container. This is cumbersome and error-prone. What we would like to do is to rely only on the `PrerequisiteSteps` of a given step to find what we have to compile. If we do so, we wouldn't need to return a `ListBuildStep` in `AssetCompilerResult`, but just the final build step for the asset, the graph of dependent build steps being described by recursive `PrerequisiteSteps`. The `ListBuildStep` container could be removed. We would still need to have lists of build steps when we compile multiple asset (eg. when compiling the full game), but it would be nothing that the build engine should be aware of.

Commands

Most command inherits from `IndexFileCommand`, which automatically register the output of the command into the command context.

Basically, at the beginning of the command (in the `PreCommand` method), a `BuildTransaction` object is created. This transaction contains a subset of the database of objects that have been already compiled, provided by the `ICommandContext.GetOutputObjectsGroups()`. In term of implementation, this method returns all the objects that where written by prerequisite build steps, and all the objects that are already written in any of the parent `ListBuildStep`s, recursively. The objects coming from the parent `ListBuildStep` are a legacy of when we were using `WaitBuildStep` to synchronize the build steps. This hopefully should be implemented differently, relying only on prerequisite (since no synchronization can happen in the `ListBuildStep itself, everything is run in parallel).

> **TODO:** Rewrite how OutputObjects are transfered from `BuildStep`s to other `BuildStep`s. Only the output from prerequisite `BuildStep` should be transfered. A lot of legacy makes this code very convoluted and hard to maintain.

The `BuildTransaction` created during this step is mounted as a *Microthread-local database*, which is accessible only from the current microthread (which is basically the current command).

At the end of the command (in the `PostCommand` method), every object that has been written in the database by the command are extracted from the `BuildTransaction` and registered to the current `ICommandContext` (which is how the `ICommandContext` can "flow" objects from one command to the other.

It's important to keep in mind that objects accessible in a given command (in the `DoCommandOverride`) using a `ContentManager` are those provided during the `PreCommand` step, and therefore it is important that dependencies between commands (what other commmands a command needs to be completed to start) are properly set.

Compilers

Compilers are classes that generate a set of `BuildStep`s to compile a given `Asset` in a specific context. This list could grow in the future if we have other needs, but the current different contexts are:

- compiling the asset for the game
- compiling the asset for the scene editor
- compiling the asset to display in the preview
- compiling the asset to generate a thumbnail

IAssetCompiler

This is the base interface for compiler. The entry point is the `Prepare` method, which takes an `AssetItem` and returns a `AssetCompilerResult`, which is a mix of a `LoggerResult` and a `ListBuildStep`. Usually there are two implementations per asset types, one to compile asset for the game and one to compile asset for its thumbnails. Some asset types such as animations might have an additional implementation for the preview.

Each implementation of `IAssetCompiler` must have the `AssetCompilerAttribute` attached to the class, in order to be registered (compilers are registered via the `AssetCompilerRegistry`.

> **TODO:** The `AssetCompilerRegistry` could be merged into the `AssetRegistry` to have a single location where asset-related types and meta-information are registered.

Each compiler provides a set of methods to help discover the dependencies between assets and compilers. They will be covered later in this document.

ICompilationContext

> Not to be mistaken with `CompilerContext` and `AssetCompilerContext`.

Contexts of compilation are defined by *types*, which allow to use inheritance mechanism to fallback on a default compiler when there is no specific compiler for a given context. Each compilation context type must implement `ICompilationContext`. Currently we have:

- `AssetCompilationContext` is the context used when we compile an asset for the runtime (ie. the game).
- `EditorGameCompilationContext` is the context used when we compile an asset for the scene editor, which is a specific runtime. Therefore, it inherits from `AssetCompilationContext`.
- `PreviewCompilationContext` is the context used when we compile an asset for the preview, which is a specific runtime. Therefore, it inherits from `AssetCompilationContext`.
- `ThumbnailCompilationContext` is the context used when we compile an asset to generate a thumbnail. Generally, for thumbnails, we compile one or several assets for the runtime, and use additional steps to generate the thumbnail with the `ThumbnailCompilationContext` (see below).

> **TODO:** Currently thumbnail compilation is in a poor state. In `ThumbnailListCompiler.Compile`, we first generate the steps to compile the asset in `PreviewCompilationContext`, then generate the steps to compile the asset in `ThumbnailCompilationContext`, and finally we like the first with the latter. Dependencies from thumbnail compilers (which load a scene and take screenshots) to the runtime compiler (which compile the asset) is **not** expressed at all. It just works now because in all current cases, the `PreviewCompilationContext` does what we need for thumbnails (for example, the `AnimationAssetPreviewCompiler` adds the preview model to the normal compilation of the animation, which is needed for both preview and thumbnail).

Dependency managers

We currently have two mechanisms that handle dependencies.

> **TODO:** Merge the `AssetDependencyManager` and the `BuildDependencyManager` together into a single dependency manager object. There is a lot of redundancy between both, one rely on the other, some code is duplicated. See `XK-4862`

AssetDependencyManager

The `AssetDependencyManager` was the first implementation of an mechanism to manage dependencies between assets. It works independently of the build, which is one of the main issue it had and the reason why we started to develop a new infrastructure.

It is based essentially on visiting assets with a `DataVisitorBase` to find references to other assets. There are two ways of referencing an asset:

- Having a property whose type is an implementation of `IReference`. More explicitely the only case we have currently is `AssetReference`. This type contains an `AssetId` and a `Location` corresponding to the referenced asset.
- Having a property whose type correspond to a *Content* type, ie. a type registered as being the compiled version of an asset type (for example, `Texture` is the Content type of `TextureAsset`).

The problem of that design was that once all the references are collected, there is no way to know of the referenced assets are actually consumed, which could be one of the three following way:

- the referenced asset is not needed to compile this asset, but it's needed at runtime to use the compiled content (eg. Models need Materials, who need Textures. But you can compile Models, Materials and Textures independently).
- the referenced asset needs to be compiled before this asset, and the compiler of this asset needs to load the corresponding content generated from the referenced asset (eg. A prefab model, which aggregates multiple models together, needs the compiled version of each model it's referencing to be able to merge them).
- the referenced asset is read when compiling this asset because it depends on some of its parameter, but the referenced asset itself doesn't need to be compiled first (eg. Navigation Meshes need to read the scene asset they are related to in order to gather static colliders it contains, but they don't need to compile the scene itself).

BuildDependencyManager

The `BuildDependencyManager` has been introduced recently to solve the problems of the `AssetDependencyManager`. It is currently not complete, and the ultimate goal is to merge it totally with the `AssetDependencyManager`.

The approach is a bit different. Rather than extracting dependencies from the asset itself, we extract them from the compilers of the assets, which are better suited to know what they exactly need to compile the asset and what will be needed to load the asset at runtime.

But one asset type can have multiple compilers associated to it (for the game, for the thumbnail, for the preview...). So the `BuildDependencyManager` works in the context of a specific compiler.

Currently there is one `BuildDependencyManager` for each type of compiler.

> **TODO:** Have a single global instance of `BuildDependencyManager` that contains all types of dependencies for all context of compilers. For example, we have thumbnail compilers that requires *game* version of assets, which means that the `BuildDependencyManager` for thumbnails will also contain a large part of the `BuildDependencyManager` to build the game. Merging everything into a

> single graph would reduce redundancy and risk to trigger the same operation multiple times simultaneously.

AssetDependenciesCompiler

The `AssetDependenciesCompiler` is the object that computes the dependencies with the `BuildDependencyManager`, and then generates the build steps for a given asset, including the runtime dependencies. It's the main entry point of compilation for the CompilerApp, the scene editor, and the preview. Thumbnails also use it, via the `ThumbnailListCompiler` class.

> **TODO:** This class should be removed, and its content moved into the `BuildDependencyManager` class. By doing so, it should be possible to make `BuildAssetNode` and `BuildAssetLink` internal - those classes are just the data of the dependency graph, they should not be exposed publicly. To do that, a method to retrieve the dependencies in a given context must be implemented in `BuildDependencyManager` in order to fix the usage of `BuildAssetNode` in `EditorContentLoader`.

In the Game Studio

The Game Studio compiles assets in various versions all the time. It has some specific way of managing database and content depending on the context.

Remark: the Game Studio never saves index file on the disk, it keeps the url -> hash mapping in memory, always.

Databases

Before accessing content to load, a Microthread-local database must be mounted. Depending on the context, it can be a database containing a scene and its dependencies (scene editor), the assets needed to create a thumbnail, an asset to display in the preview...

For the scene editor, this is handled by the `GameStudioDatabase` class. Thumbnails and preview also handle database mounting internally (in `ThumbnailGenerator` for example).

> **TODO:** See if it could be possible/useful to wrap all database-mounting in the Game Studio into the GameStudioDatabase class.

Builder service

All compilations that occur in the Game Studio is done through the `GameStudioBuilderService`. This class creates an instance of `Builder`, a `DynamicBuilder` which allows to feed the Builder with build steps at any

time. Having a single builder for the whole Game Studio allows to control the number of threads and concurrent tasks more easily.

The `DynamicBuilder` class simply creates a thread to run the Builder on, and set a special build step, `DynamicBuildStep`, as root step of this builder. This step is permanently waiting for other child build step to be posted, and execute them.

> **TODO:** Currently the dynamic build step waits arbitratly with the `CompleteOneBuildStep` method when more than 8 assets compiling. This is a poor design because if the 8 assets are for example prefabs who contains a lot of models, materials, textures, it will block until all are done, although we could complete the thumbnails of these models/materials/textures individually. Ideally, this `await` should be removed, and a way to make sure thumbnails of assets which are compiled are created as soon as possible should be implemented.

The builder service uses `AssetBuildUnit`s as unit of compilation. A build unit corresponds to a single asset, and encapsulates the compiler and the generated build step of this asset.

EditorContentLoader

The scene editor needs a special behavior in term of asset loading. The main issue is that any type of asset can be modified by the user (for example a texture), and then need to be reloaded. Stride use the `ContentManager` to handle reference counting of loaded assets. With a few exception (Materials, maybe Textures), it does not support hot-swapping an asset. Therefore, when an asset needs to be reloaded, we actually need to unload and reload the *first-referencer* of this asset.

The *first-referencer* is the first asset referenced by an entity, that contains a way (in term of reference) to the asset to reload. For example, in case of a texture, we will have to reload all models that use materials that use the texture to reload.

This is done by the `EditorContentLoader` class. At initialization, this class collects all *first-referencer* assets and build them. Each time an asset is built, it is then loaded into the scene editor game, and the references (from the entity to the asset) are updated. This means that this class needs to track all first-referencers on its own and update them. This is done specifically by the `LoaderReferenceManager` object. The reference are collected from the `GameEditorChangePropagator`, an object that takes the responsibility to push synchronization of changes between the assets and the game (for all properties, including non-references). There is one instance of it per entity. When a property of an entity that contains a reference to an asset (a *first-referencer*) is modified, the propagator will trigger the work to compile and update the entity. In case of a referenced asset modified by the user, `EditorContentLoader.AssetPropertiesChanged` takes the responsibility to gather, build, unload and reload what needs to be reloaded.

Additional Todos

TODO: `GetInputFiles` exists both in `Command` and in `IAssetCompiler`. It has the same signature in both case, so it's returning information using `ObjectUrl` and `UrlType` in the compiler, where we are trying to describe dependency. That signature should be changed, so it returns information using `BuildDependencyType` and `AssetCompilationContext`, just like the GetInputTypes method. Also, the method is passed to the command via the `InputFilesGetter` which is not very nice and has to be done manually (super error-prone, we had multiple commands that were missing it!). An automated way should be provided.

TODO: The current design of the build steps and list build steps is a *tree*. For this reason, same build steps are often generated multiple times and appears in multiple trees. It could be possible to cache and share the build step if the structure was a *graph* rather than a *tree*. Do to that, the `Parent` property of build steps should be removed. The main difficulty is that the way output objects of build steps flow between steps has to be rewritten.

Asset, introspection and prefab

NOTE: Please read the Terminology section of the Build Pipeline documentation first

Design notes

Assets contains various properties describing how a given **Content** should be generated. Some constraints are defined by design:

- All types that can be referenced directly or indirectly by an asset must be serializable. This means that it should have the `[DataContract]` attribute, and the type of all its members must have it too.
- Members that cannot or should not be serialized can have the `[DataMemberIgnore]` attributes
- Other members can have additional metadata regarding serialization by using the `[DataMember]` attributes. There is also a large list of other attributes that can be used to customize serialization and presentation of those members.
- Arrays are not properly supported
- Any type of ordered collection is supported, but unordered collection (sets, bags) are not.
- Dictionaries are supported as long as the type of the key is a primitive type (see below for the definition of primitive type)
- When an asset references another asset, the member or item shouldn't use the type of the target asset, but the corresponding **Content**. For example, the `MaterialAsset` needs to reference a texture, it will have a `Texture` member and not a `TextureAsset`.
- It is possible to use the `AssetReference` type to represent a reference to any type of asset.
- Nullable value types are not properly supported
- An asset can reference multiple times the same objects through various members/items, but one of the member/item must be the "real instance", and the others must be defined as "object references", see below for more details.

Yaml metadata

When assets are serialized to/deserialized from Yaml files, dictionaries of metadata is created or consumed in the process. There is one dictionary per type of metadata. The dictionary maps a property path (using `YamlAssetPath`) to a value, and is stored in a instance of `YamlAssetMetadata`. These dictionary are exchanged between the low-level Yaml serialization layer and the asset-aware layer via the `AssetItem.Metadata` property. This property is not synchronized all the time, it is just consumed after deserialization, to apply metadata to the asset, and generated just before serialization, to allow the metadata to be consumed during serialization.

Overrides

The prefab and archetype system introduces the possibility to override properties of an asset. Some nodes of the property tree of an asset might have a *base*. (usually all of them in case of archetype, and some specific entities that are prefab instances in case of scene). How nodes are connected together is

explained later on this documentation, but from a serialization point of view, any property that is overridden will have associated yaml metadata. Then we usa a custom serializer backend, `AssetObjectSerializerBackend`, that will append a star symbol `*` at the end of the property name in Yaml.

Collections

Collections need special handling to properly support override. An item of a collection that is inherited from a base can be either modified (have another value) or deleted. Also, new items that are not present in the base can have been added. This is problematic in the case of ordered collection such as `List` because adding/deleting items changes the indices of item.

To solve all these issues, we introduce an object called `CollectionItemIdentifiers`. There is one instance of this object per collection that supports override. This instance is created or retrieved using the `CollectionItemIdHelper`. They are stored using `ShadowObject`, which maintain weak references from the collection to the `CollectionItemIdentifiers`. This means that it is currently not possible to have overridable items in collection that are `struct`.

A collection that can't or shouldn't have overridable items should have the `NonIdentifiableCollectionItemsAttribute`.

The `CollectionItemIdentifiers` associates an item of the collection to a unique id. It also keep track of deleted items, to be able to tell, when an item in an instance collection is missing comparing to the base collection, if it's because it has been removed purposely from the instance collection, or if it's because it has been added after the instance collection creation to the base collection.

Items, in the `CollectionItemIdentifiers`, are represented by their key (for dictionaries) or index (list). This means that any collection operation (add, remove...) must call the proper method of this class to properly update this collection. This is automatically done as long as the collection is updated through Quantum (see below).

In term of inheritance and override, the item id is what connect a given item of the base to a given item of the instance. This means that items can be re-ordered, and other items can be inserted, without loosing or messing the connection between base and instances. Also, for dictionary, keys can be renamed in the instance.

At serialization, the item id is written in front of each item (so collections are transformed to dictionaries of [`ItemId`, `TValue`] and dictionary are transformed to dictionaries of [`KeyWithId<TKey>`, `TValue`], with `KeyWithId` being equivalent to a Tuple). Here is an example of Yaml for a base collection and an instance collection:

Base collection, with one id per item:

```
Strings:
    309e0b5643c5a94caa799a5ea1480617: Hello
    e09ec493d05e0446b75358f0e1c0fbdd: World
    9550f04dcee1d24fa8a30e41eea71a94: Example
    1da8adce3f0ce9449a9ed0e48cd32f20: BaseClass
```

Derived collection. The first item is overridden, the 4th is a new item (added), and the last one express that the `BaseClass` entry has been deleted in the derived instance.

```
Strings:
    309e0b5643c5a94caa799a5ea1480617*: Hi
    e09ec493d05e0446b75358f0e1c0fbdd: World
    9550f04dcee1d24fa8a30e41eea71a94: Example
    cfce75d38d66e24fae426d1f40aa4f8a*: Override
    1da8adce3f0ce9449a9ed0e48cd32f20: ~(Deleted)
```

When two assets that are connected with a base relationship are loaded, it is then possible to reconcile them:

- any item missing in the derived collection is re-added (so the `~(Deleted)` is need to purposely delete items)
- any item existing in the derived collection that doesn't exist in the base collection and doesn't have the star `*` is removed
- any item that exists in both collection but have a different value is overwritten with the value of the base collection
- overridden items (with the star `*`) are untouched

Quantum

In Stride, we use an introspection framework called *Quantum*.

Type descriptors

The first layer used to introspect object is in `Stride.Core.Reflection`. This assembly contains type descriptors, which are basically objects abstracting the reflection infrastructure. It is currently using .NET reflection (`System.Reflection`) but could later be implemented in a more efficient way (using `Expression`, or IL code).

The `TypeDescriptorFactory` allows to retrieve introspection information on any type. `ObjectDescriptor`s contains descriptor for members which allow to access them. Collections, dictionaries and arrays are also handled (NOTE: arrays are not fully supported in Quantum itself).

This assembly also provides an `AttributeRegistry` which allows to attach `Attribute`s to any class or member externally.

> **TODO:** make sure all locations where we read `Attribute`s are using the `AttributeRegistry` and not the default .NET methods, so we properly support externally attached attributes.

Node graphs

In order to introspect object, we build graphs on top of each object, representing their members, and referencing the graphs of other objects they reference through members or collection. The classes handling theses graphs are in the `Stride.Core.Quantum` assembly.

Node containers

Nodes of the graphs are created into an instance of `NodeContainer`. Usually a single instance of `NodeContainer` is enough, but we have some scenarios where we use multiple ones: for example each instance of scene editor contains its own `NodeContainer` instance to build graphs of game-side objects, which are different from asset-side (ie. UI-side) objects, have a different lifespan, and require different metadata.

In the GameStudio, the `NodeContainer` class has two derivations: the `AssetNodeContainer` class, which expands the primitive types to add Stride-specific types (such as `Vector3`, `Matrix`, `Guid`...). This class is inherited to a `SessionNodeContainer`, which additionally allows plugin to register their own primitive types and metadata.

Node builders

The `NodeContainer` contains an `INodeBuilder` member and provides a default implementation for it. So far we didn't had the need to make a custom implementation, since the structure of the graphs themselves is pretty stable.

However, the `INodeBuilder` interface presents an `INodeFactory` member which we override. This factory allows to customize the nodes to be constructed.

The `INodeBuilder` also contains a list of types to be considered as *primitive types*, which means that even if the type contains members or is a reference type, it will be, in term of graph, considered as a primitive value and won't be expanded.

Nodes

There are 3 types of nodes in Quantum:

- `ObjectNode` are node corresponding to an object that is a reference type. They can contain members (properties, fields...), and items (collection).
- `BoxedNode` are a special case of `ObjectNode` that handles `struct`. They are able to write back the value of the struct in other nodes that reference them
- `MemberNode` are node corresponding to the members of an object. If the value of the member is a class or a struct, the member will also contain a reference to the corresponding `ObjectNode`.
- `ObjectNode` that are representing a collection of class/struct items will also have a collection of reference to target nodes via the `ItemReferences` property.

Each node has some methods that allow to manipulate the value it's wrapping. `Retrieve` returns the current value, `Update` changes it. Collections can be manipulated with the `Add` and `Remove` methods (and a single item can be modified also with `Update`).

Events

Each node presents events that can be registered to:

- `PrepareChange` and `FinalizeChange` are raised at the very beginning and the very end of a change of the node value. These events are internal to Quantum.
- `MemberNode`s have the `ValueChanging` and `ValueChanged` events that are raised when the value is being modified.
- `ObjectNode` have `ItemChanging` and `ItemChanged` events that are raised when the wrapped object is a collection, and this collection is modified.

The arguments of these events all inherits from `INodeChangeEventArgs`, which allows to share the handlers between collection changes and member changes.

Finally, Quantum nodes are specialized for assets, where the implementation of the support of override and base is. These specialized classes also present `OverrideChanging` and `OverrideChanged` event to handle changes in the override state.

AssetPropertyGraph

Concept

We use Quantum nodes mainly to represent and save the properties of an asset. The AssetPropertyGraph is a container of all the nodes related to an asset, and describes certain rules such as which node is an object reference, etc.

Asset references

When an asset needs to reference another asset, it should never contains a member that is of the type of the referenced asset. Rather, the type of the member should be the type of the *Content* corresponding to the referenced asset.

Node listener

A node listener is an object that can listen to changes in a graph of node (rather than an individual nodes). The base class is `GraphNodeChangeListener`, and this class must define a visitor that can visit the graph of nodes to register, and stop at the boundaries of that graph.

Object references

In many scenarios of serialization (in YAML, but also in the property grid where objects are represented by a tree rather than a graph), we need a way to represent multiple referencers of the same object such a way that the object is actually expanded at one unique location, and shown/serialized as a reference to all other locations. We introduce the concept of **Object references** to solve this issue.

By design, only objects implementing the `IIdentifiable` interface can be referenced from multiple locations from the same root object. But right now they can only be referenced from the same unique root object (usually an `Asset`). Later on we might support *cross-asset references* but this would require to change how we serialize them.

There are two methods to implement to define if a node must be considered as an object reference or not:

- one for members of an object: `IsMemberTargetObjectReference`
- one for items of a collection: `IsTargetItemObjectReference`

Node presenters

Node presenters are objects used to present the properties of an object to a view system, such as a property grid. They transform a graph of nodes to a tree of nodes, and contains metadata to be consumed by the view. The resulting tree is slightly different from the graph. When an object A contains a member that is an object B that contains a property C, the graph will look like this:

```
ObjectNode A --(members)--> MemberNode B --(target)--> ObjectNode B --(members)--> MemberNode C
```

the corresponding tree of node presenters will be:

```
RootNodePresenter A --> MemberNodePresenter B --> MemberNodePresenter C
```

There is also a `ItemNodePresenter` for collection. On the example above, if B is instead a collection that contains a single item C, the graph would be:

```
ObjectNode A --(members)--> MemberNode B --(target)--> ObjectNode B --(items)--> ObjectNode C
```

the corresponding tree of node presenters will be:

```
RootNodePresenter A --> ItemNodePresenter B --> MemberNodePresenter C
```

Node presenter are constructed by a `INodePresenterFactory` in which `INodePresenterUpdater` can be registered. A `INodePresenterUpdater` allows to attach metadata to nodes, and re-organize the hierarchy in case it want to be presented differently from the actual structures (by inserting nodes to create category, bypassing a class object to inline its members, etc.). `INodePresenterUpdater` have two methods to update node:

- `void UpdateNode(INodePresenter node)` is called on **each** node, after its children have been created. But it's not guaranteed that its siblings, or the siblings of its parents, will be constructed.
- `void FinalizeTree(INodePresenter root)` is called once, at the end of the creation of the tree, and only on the root. Here it's guaranteed that every node is constructed, but you have to visit manually the tree to find the node that you want to customize.

Node presenters listens to changes in the graph node they are wrapping. In case of an update, the children of the modified node are discarded and reconstructed. `UpdateNode` is called again on all new children, and `FinalizeTree` is also called again at the end on the root of the tree. Therefore, you have to be aware that an updater can run multiple time on the same nodes/trees.

Metadata can be attached to node presenters via the `NodePresenterBase.AttachedProperties` property containers. These metadata are exposed to the view models as described in the section below.

Commands can also be attached to node presenters. A command does special actions on a node, in order to update it. Node presenter commands implements the `INodePresenterCommand` interface. A command is divided in three steps, in order to handle multi-selection:

- `PreExecute` and `PostExecute` are run only once, for a selection of similar node presenters, before and after `Execute` respectively.
- `Execute` is run once per selected node presenter.

Node view models

The view models are created on top of node presenters. Each node presenter has a corresponding `NodeViewModel`. In case of multi-selection, a `NodeViewModel` can actually wrap a collection of node presenters, rather than a single one.

Metadata (ie. attached properties) are also exposed from the node presenter to the view via the view model, assuming they are common to all wrapped node presenter, if not, it is possible to add a `PropertyCombinerMetadata` to the property key to define the rule to combine the metadata. The default behavior for combining is to set the value to `DifferentValues` (a special object representing different values) if the values are not equals.

Commands are also exposed. They are added to the view model, combined depending on their `CombineMode` property. They are transformed into WPF commands by being wrapped into a `NodePresenterCommandWrapper`.

All members, attached properties, and commands of node view models are exposed as `dynamic` properties, and can therefore be used in databinding.

All node view models are contained in an instance of `GraphViewModel`. A `GraphViewModelService` is passed in this object that acts as a registry for the node presenter commands and updaters that are available during the construction of the tree.

Template selector

In order to be presented to the property grid, a proper template must be selected for each NodeViewModel. The `TemplateProviderSelector` object picks the proper template by finding the first registered one that accept the given node. Templates are defined in various XAML resource dictionaries, the base one being `DefaultPropertyTemplateProviders.xaml`. There is a priority mechanism that uses an `OverrideRule` enum with four values: `All`, `Most`, `Some`, `None`. One template can also explicitly override the other with the `OverriddenProviderNames` collection. The algorithm that picks the best match is in the `CompareTo` method of `TemplateProviderBase`.

There is actually 3 levels of templates for each property. `PropertyHeader` and `PropertyFooter` represent the section above and the section below the expander that contains the children properties. In the default implementation (`DefaultPropertyHeaderTemplate` and most of its specializations), the header presents the left part of the property (the name, sometimes a checkbox...), and use the third template category, `PropertyEditor`, for the right side of the property grid.

Bases

The base-derived concept and the override are stored in specialized Quantum nodes that implements `IAssetNode`. Properties (as well are items of collections) are automatically overridden when `Update`/`Add`/`Remove` methods are called. Some methods are also provided to manually interact with overrides, but it should not be used directly by users of Quantum.

Node linker

`GraphNodeLinker` is an object that link a given node to another node. It has two main usages: it links objects that are game-side in the scene editor to their counterpart asset-side, and they also link a node to its base if it has one.

The `AssetToBaseNodeLinker` is used to do that. It is invoked at initialization, as well as each time a property changes. It has a `FindTarget` method and `FindTargetReference`, which basically resolve, when visiting the derived graph, which equivalent node of the base graph corresponds to it.

This linker is run from the `AssetPropertyGraph` that can then call `SetBaseNode` to actually link the nodes together.

Reconciliation with base

Each time a change occurs in an asset, all nodes that have the modified nodes as base will call `ReconcileWithBase`. This method visits the graph, starting from the modified properties, and "reconcile" the change. The method is a bit long but well commented. The principle is, for each node, to detect first if something should be reconciled, and if yes, find the proper value (either cloning the value from the base, or find a corresponding existing object in the derived) and set it.

`ReconcileWithBase` is also called at initialization to make sure that any desynchronization that could happen offline is fixed.

Future

Undo/redo

The undo/redo system currently records only the change on the modified object, and rely on `ReconcileWithBase` to undo/redo the changes on the derived object. This is not an ideal design because there are a lot of consideration to take, and a lot of special cases.

What we would like to do is:

- record everything that changes, both in derived and in base nodes
- disbranch totally automatic propagation during an undo/redo

This design was not possible initially, and I'm not sure it is possible to do now - it's possible to hit a blocker when implementing it, or that it requires a lot of refactoring here and there before being doable.

Dynamic nodes

Currently we still expose the real asset object in `AssetViewModel`, which it should never, in the editor, be modified out of Quantum node. Also, manipulating Quantum node is quite difficult sometimes due to indirection with target nodes, and access to members.

```
var partsNode = RootNode[nameof(AssetCompositeHierarchy<TAssetPartDesign,
TAssetPart>.Hierarchy)].Target[nameof(AssetCompositeHierarchyData<IAssetPartDesign<IIdentifi
able>, IIdentifiable>.Parts)].Target;
partsNode.Add(newPart);
```

Ideally, we would like to use the `DynamicNode` objects (currently broken) to manipulate quantum nodes:

```
dynamic root = DynamicNode.Get(RootNode);
root.Hierarchy.Parts.Add(newPart)
```

If this is done properly, `AssetViewModel.Asset` could be turned private, and `AssetViewModel` could just expose the root dynamic node, which would allow to seemlessly manipulate the asset through a `dynamic`

object.

Files and folders

This section explains Stride's files and folders and the best way to organize them in development.

In this section

- Project structure
- Cached files
- Version control
- Distribute a game

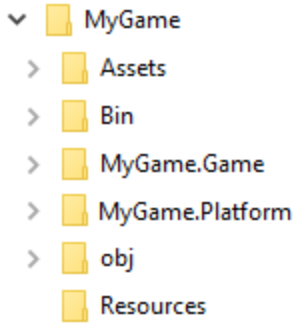
Project structure

Stride saves your projects as Visual Studio solution files. You can open the projects with Stride Game Studio or any IDE such as Visual Studio.

Stride organizes project files into **packages**. Each package comprises several folders and an *.sdpkg file which describes the package.

A project can contain one package or several. You can share packages between projects.

Package folder structure



- **Assets** contains the asset files which represent elements in your game.
- **Bin** contains the compiled binaries and data. Stride creates the folder when you build the project, with a subdirectory for each platform.
- **MyGame.Game** contains the source code of your game as a cross-platform Visual Studio project (.csproj). You can add multiple projects to the same game.
- **MyGame.Platform** contains additional code for the platforms your project supports. Game Studio creates folders for each platform (eg *MyPackage.Windows*, *MyPackage.Linux*, etc). These folders are usually small, and only contain the entry point of the program.
- **obj** contains cached files. Game Studio creates this folder when you build your project.
- **Resources** is a suggested location for files such as images and audio files used by your assets.

Recommended project structure

For advice about the best way to organize your project, see the Version control page.

See also

- Version control
- Distribute a game

Cached files

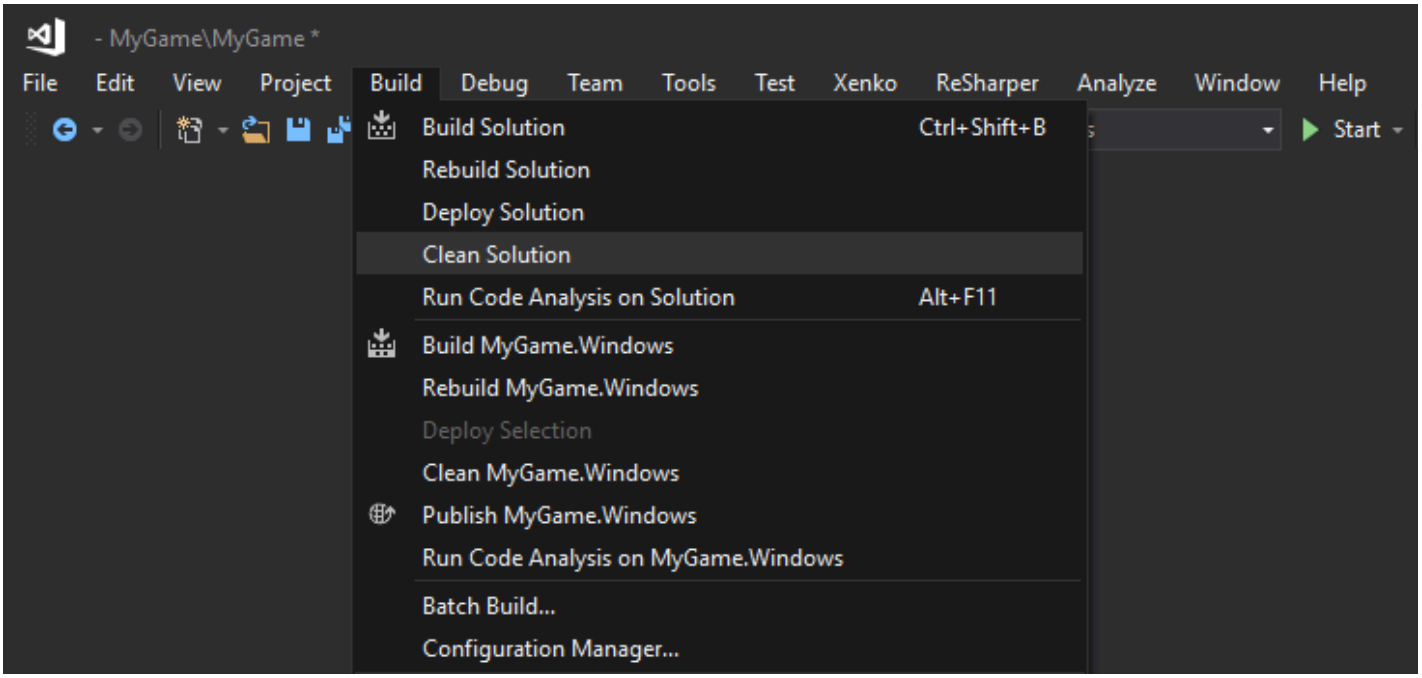
When you build your project, Stride caches the assets and code in folders inside the project.

You might want to clean the cache if:

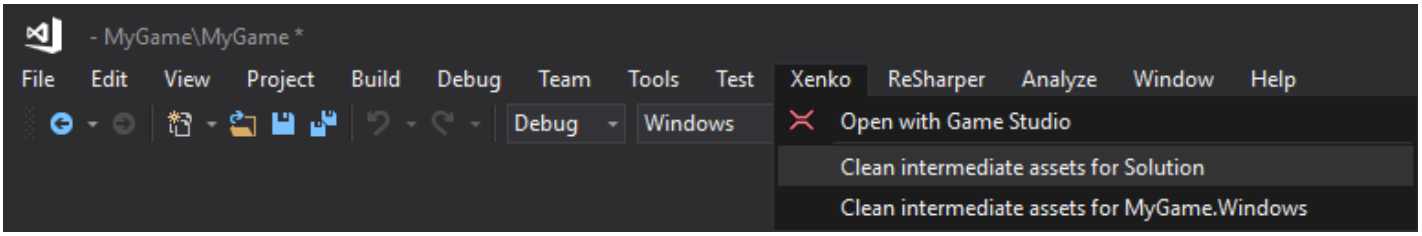
- the cache is taking up too much space on disk
- assets don't update in-game after you edit or delete them

Clean the cache from Visual Studio

1. To clean the code cache, under **Build**, select **Clean Solution**.



2. If you have the Stride Visual Studio extension installed, you can also clean the asset cache. **Using Visual Studio**: To do this, under Extensions > **Stride**, select **Clean intermediate assets for Solution**.



3. Rebuild the project to rebuild the cache from scratch.

Clean the cache manually

If cleaning the cache from Visual Studio doesn't work, try deleting the files manually.

1. Delete the following folders:
    - the binary cache: *~/MyGame/MyGame/Bin*
    - the asset cache: *~/MyGame/MyGame/Cache*
    - the **obj** folders in the platform folders for your game (eg *~/MyGame.iOS/obj*)
2. If you're developing for iOS, on your Mac, also delete:
   ~/Library/Caches/Xamarin/mtbs/builds/MyGame
3. Rebuild the project to rebuild the cache from scratch.

Clear the Game Studio caches

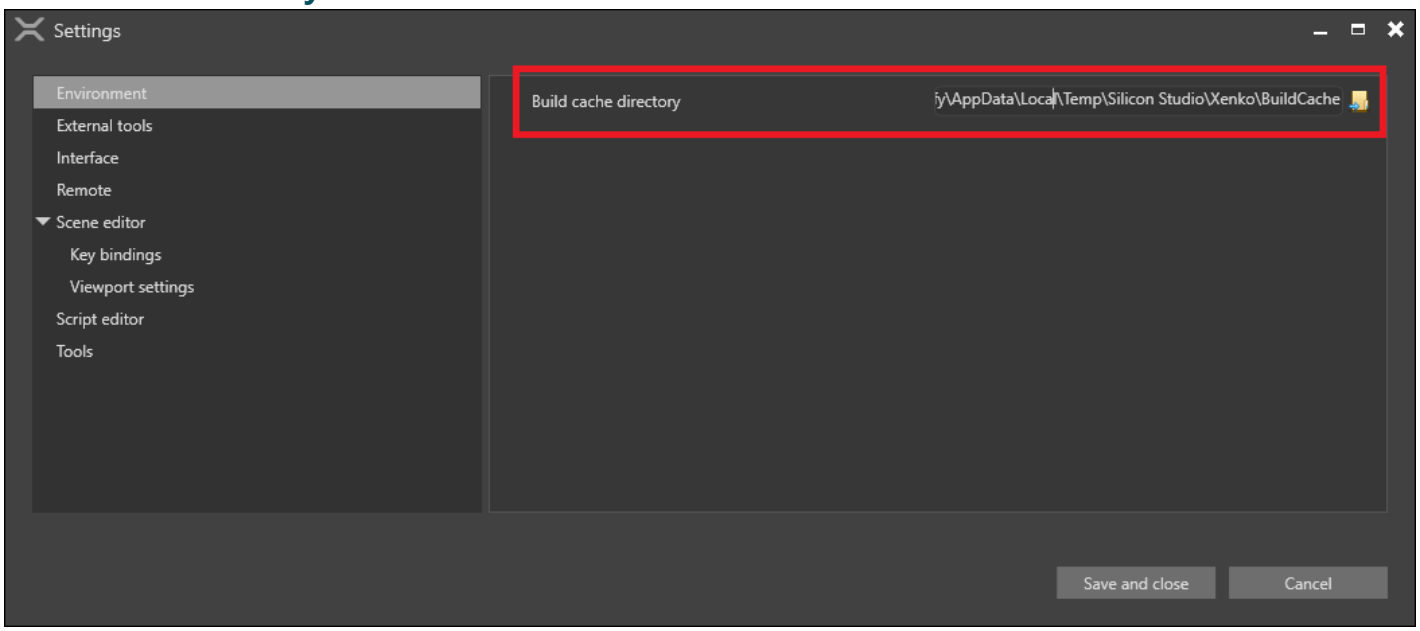
In addition to the caches Stride creates for your project, Game Studio keeps caches for the editor.

Asset cache

To speed up asset loading in the editor, Game Studio saves a cache of asset references. It contains data about every asset ever loaded in every project. This means it can grow very large over time.

By default, the folder is in: *%temp%/Stride*

> **TIP**
>
> To check or change where Game Studio saves the cache, see **Edit > Settings > Environment > Build cache directory.**



To clean the cache, delete the folder and run Game Studio again.

Settings cache

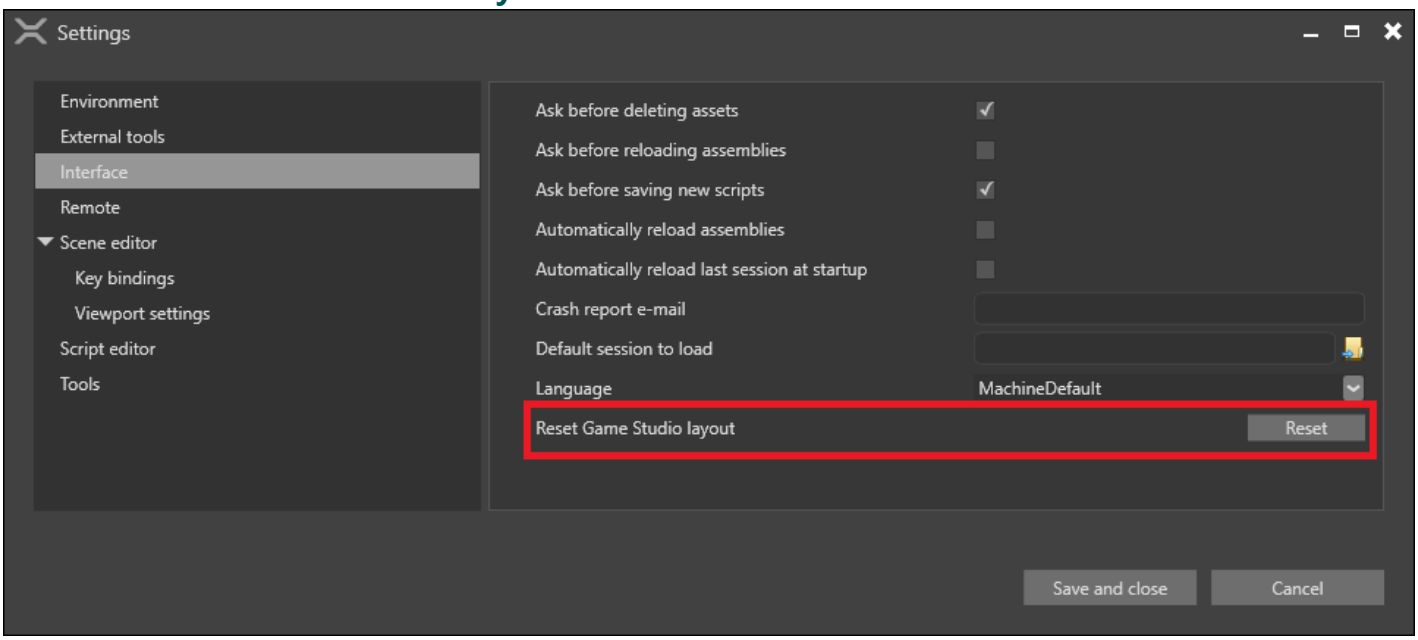
Game Studio saves editor information (such as window positions and recently-opened projects) in: *%AppData%/Stride*

Game Studio also saves information about open tabs and the editor camera position in the `.sdpkg.user` file in the project folder (eg *~/MyGame/MyGame/MyGame.sdpkg.user*).

These files are small, but you might want to delete them if you get Game Studio into a bad state. Deleting them doesn't affect anything in your project.

After you delete cache files, when you start Game Studio, it builds a new cache using the default settings.

> **TIP**
>
> You can also reset the Game Studio layout without clearing the cache in **Edit > Settings > Interface > Reset Game Studio layout**.



See also

- Project structure
- Version control

Organize your files in version control

We recommend you use a version control system such as Git, SVN, or Perforce Helix to save a history of changes to your project.

How you organize and share your files is up to you, but there are some things to keep in mind.

Files you shouldn't add to version control

Bin and **obj** folders

We don't recommend you add the **Bin** or **obj** folders to version control. This is because:

- Game Studio builds these folders every time you run the game, so you don't need to keep a history of them.
- You can't see if they match the source files they were generated from in a given commit.
- They take up space and slow down version control synchronization.

Visual Studio also puts **.obj** folders inside each code folder. For the same reasons, we don't recommend you add these to version control.

Resource files

Resource files are files imported into Game Studio and used by assets. They include image files (eg `.png`, `.jpg`), audio files (eg `.mp3`, `.wav`), and models (eg `.fbx`). We recommend you save these files in the **Resources** folder in your project folder.

We don't recommend you save resource files in the Assets folder. You might be used to organizing files this way if you use Unity®, as Unity® requires resource files and asset files to be in the same folder. Stride doesn't require this, and doing so has downsides.

For example, imagine an artist has edited 10GB of textures and committed them to source control. At the same time, a designer needs to edit an asset quickly. To do this, the designer gets the latest version of the asset from source control. However, because the assets and resource files are in the same folder, the designer is forced to get the 10gb of files at the same time. If the files are in a separate folder, however, the designer only has to get the folder they need. Additionally, as asset files are much smaller than resource files, it's much faster to navigate the asset history in a dedicated asset folder.

Content creation files

Content creation files are created with external content creation tools, such as `.psd` files (Photoshop) or `.max` files (3D Studio Max).

We don't recommend you save content creation files in your project folder. This is because the files are often large and aren't used in the project directly. Instead, we recommend you save the files in a

different version control repository - or, if your version control system supports partial checkouts (such as SVN or Perforce), a different root folder. This means team members only get the files they need.

Suggested directory structure

Following these suggestions, an example folder structure might look like this:

```
- MyGame
    - Assets
        - texture.sdtex
    - Bin
    - MyGame.Game
    - MyGame.Platform
    - obj
    - Resources
        - texture.png
- ContentCreationFiles
    - texture.psd
```

You could even create separate folders for different kinds of content creation file:

```
- MyGame
    - Assets
        - texture.sdtex
        - model.sdtex
    - Bin
    - MyGame.Game
    - MyGame.Platform
    - obj
    - Resources
        - texture.png
        - model.fbx
- PhotoshopProjects
    - texture.psd
- MayaProjects
     - model.mb
```

Example

Imagine a team with two programmers, two 2D artists, and two 3D artists.

- The programmers check out the *MyGame* project folder containing code, assets, and resources.
- The 2D artists check out the game project and the *PhotoshopProjects* folder containing `.psd` files.

- The 3D artists check out the game project and the *MayaProjects* folder containing `.mb` (Maya project) files.

Now imagine one of the 2D artists changes several `.psd` files and commits 2GB of changes to version control. Because only the 2D artists have the *PhotoshopProjects* folder checked out, only the other 2D artist gets this change. The other team members don't need it. This is an efficient way to share files across projects.

See also

- Project structure
- Distribute a game

Distribute a game

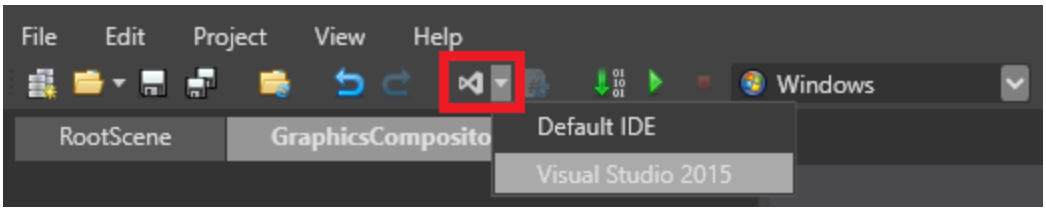
When you're ready to publish your game, create a release build from Visual Studio, then distribute it.

1. Create a release build

1. If you've built your game in Release mode before, in your project folder (eg *MyGame/Bin/MyPlatform/Release/*), delete the *Data* folder. This folder might contain unnecessary files, such as old versions of assets, so it's simplest to build it again from scratch.

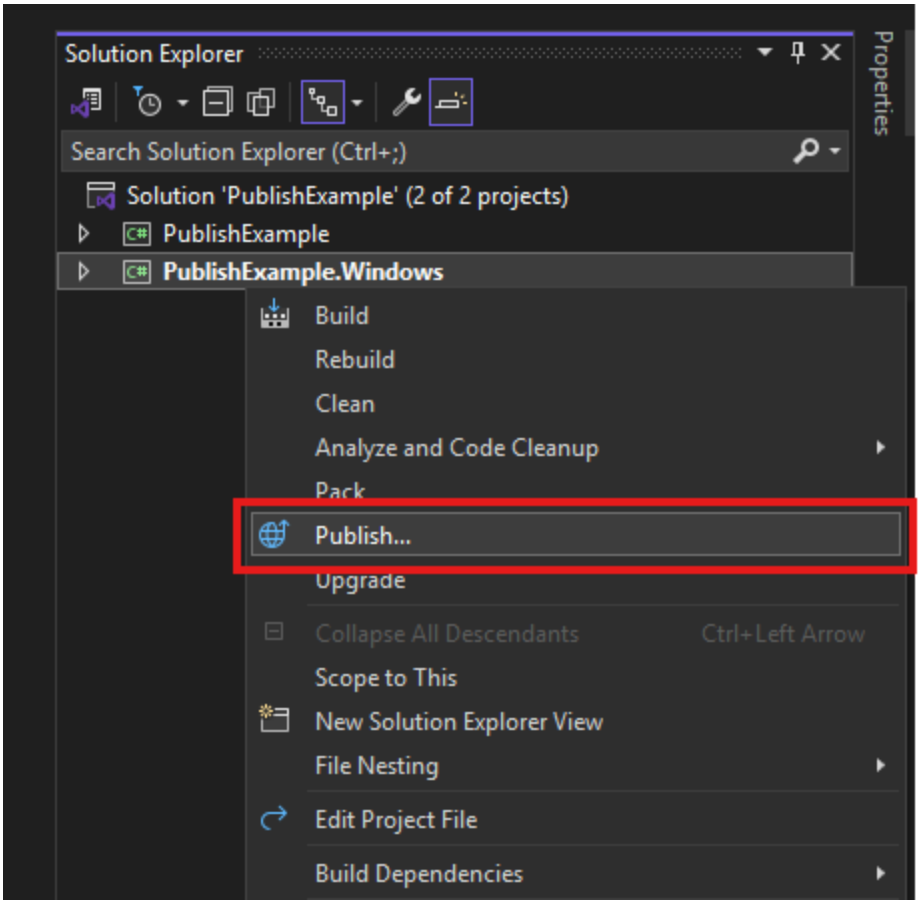
2. Open your project in Game Studio.

3. In the toolbar, click the drop-down menu and select **Visual Studio**.



   Your project opens in Visual Studio.

4. In Visual Studio, from the **Solution Explorer** right click your Windows project and select **Publish**

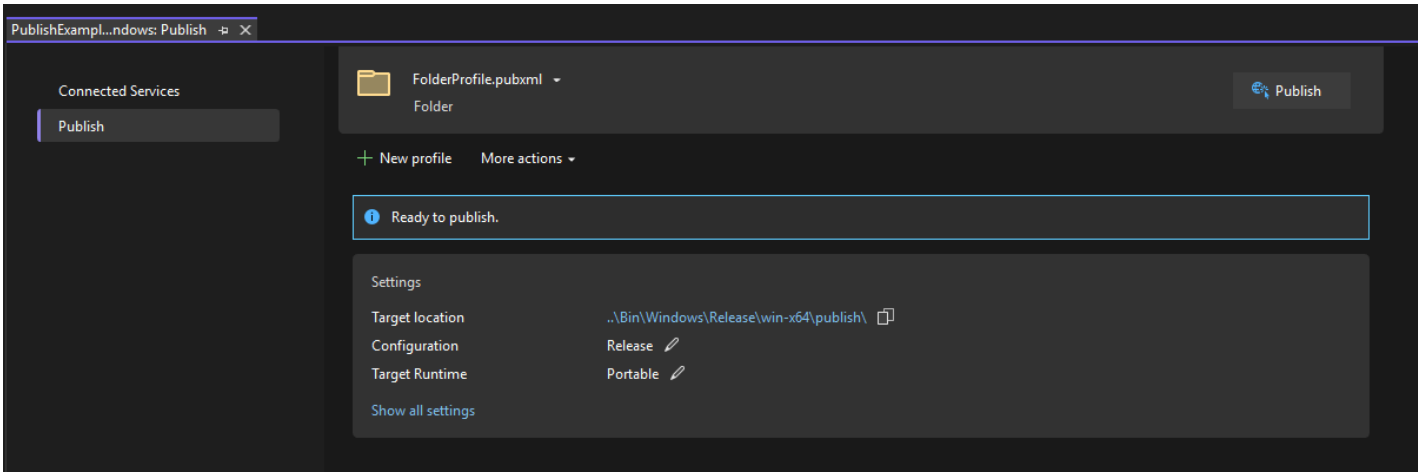


5. Select the **Target Folder** in the publish window.

6. Select the **Specified target Folder** again.

7. Confirm the output folder and Click **Finish**.

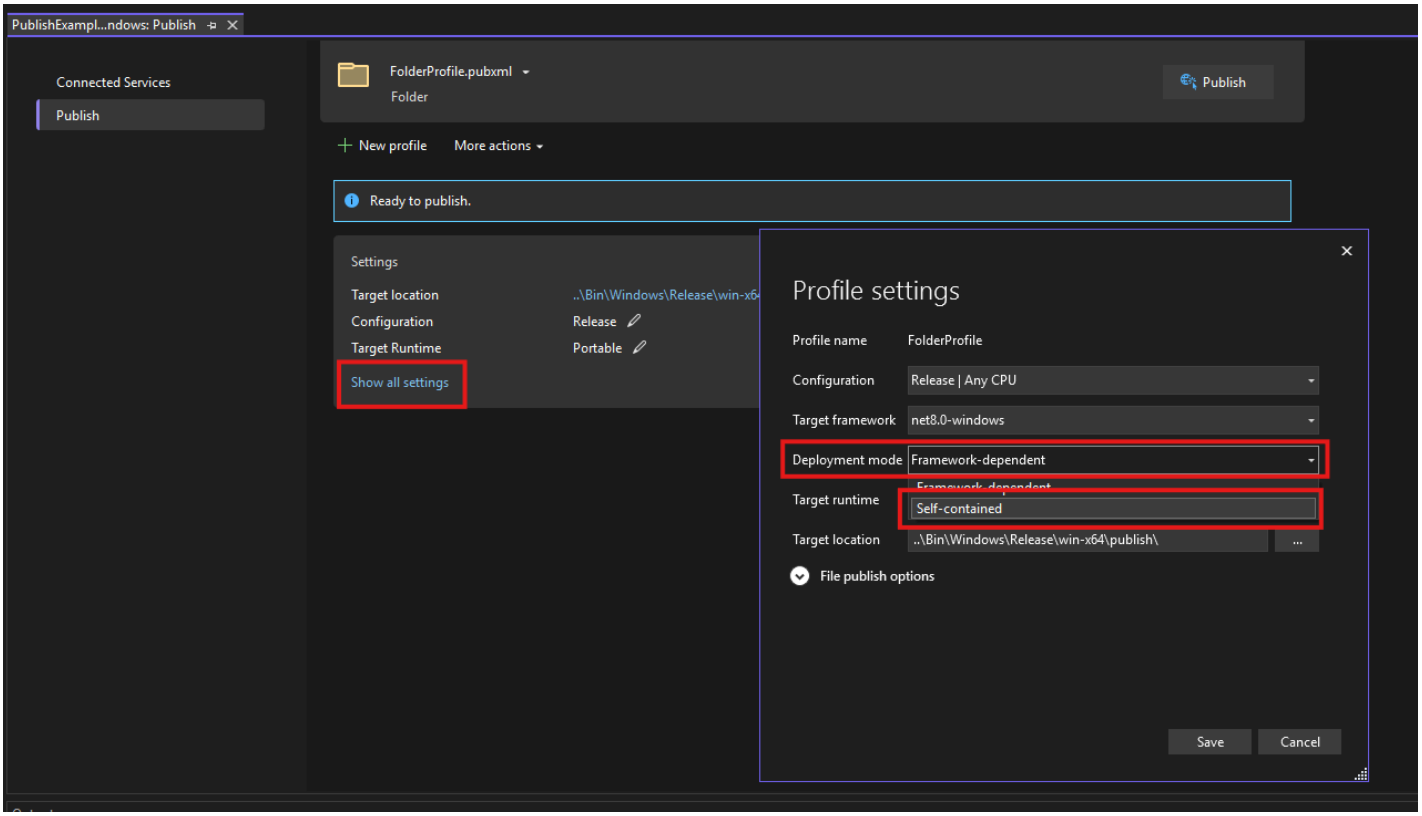
8. You should now see the **Publish** view where you can manage the project export settings.



9. Finally you can click publish and see your project in the output folder you selected at **step 6**

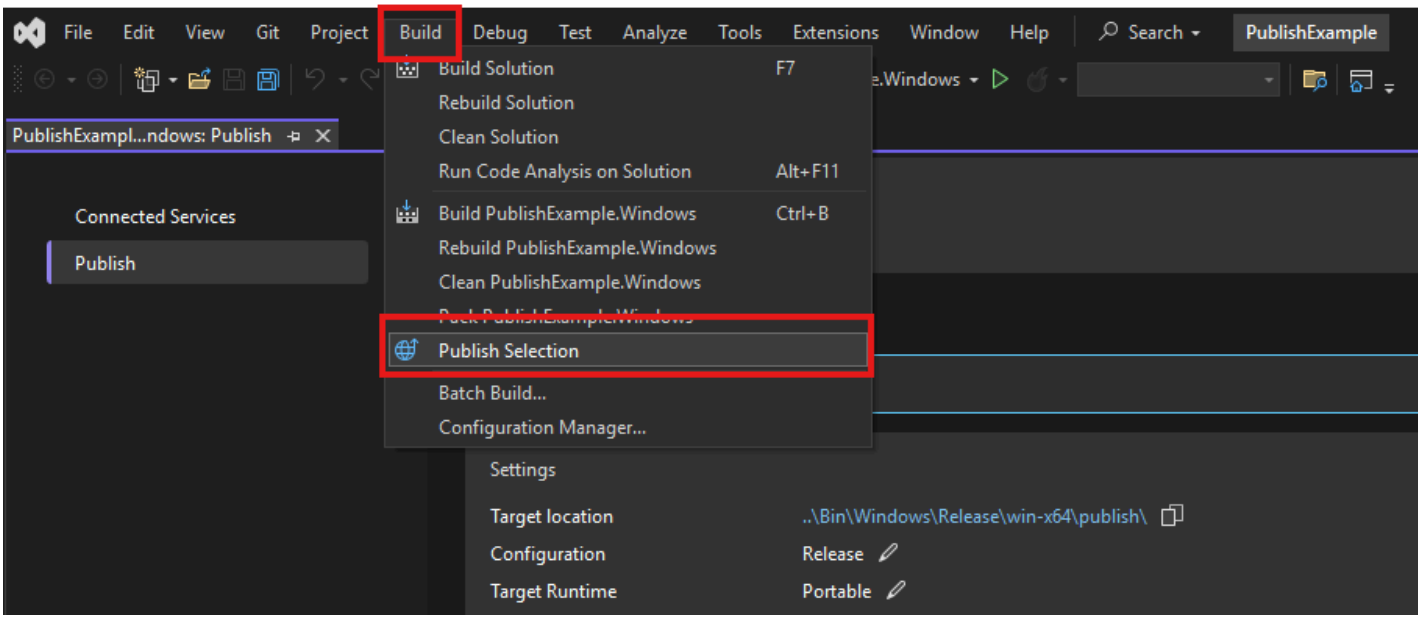
   Optionally you can also include the .NET runtime in your exported game to reduce a dependancy on the user.

   Select **Show all settings** -> **Deployment mode** -> **Self-contained** -> **Save**



> **NOTE**
>
> You can only build for platforms you've added to your Stride project. For instructions about how to do this, see Add or remove a platform.
>
> To build for Android or iOS, you need Xamarin, which is included with Visual Studio licenses. For instructions about how to install Xamarin with Visual Studio 2017, see this MSDN page.

10. Under **Build**, select **Publish Selection** and click the **Publish** button.



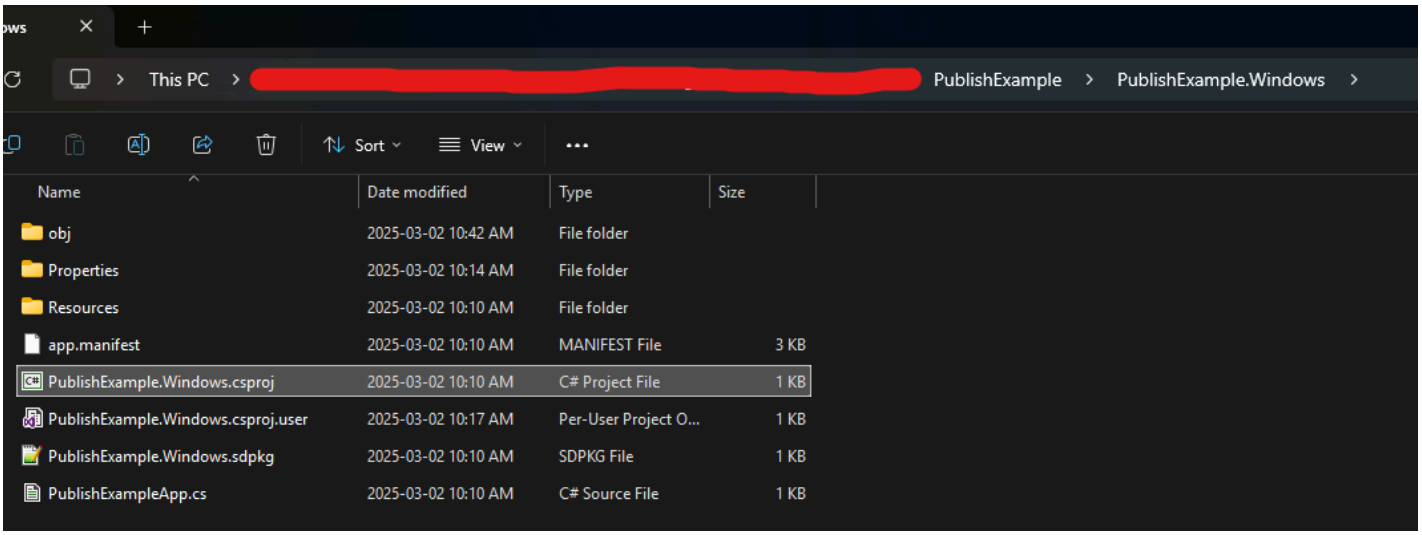
Visual Studio creates a release build in your selected output folder.

> **TIP**
>
> You might want to rename the **Release** folder to something more descriptive (such as the title of your game).

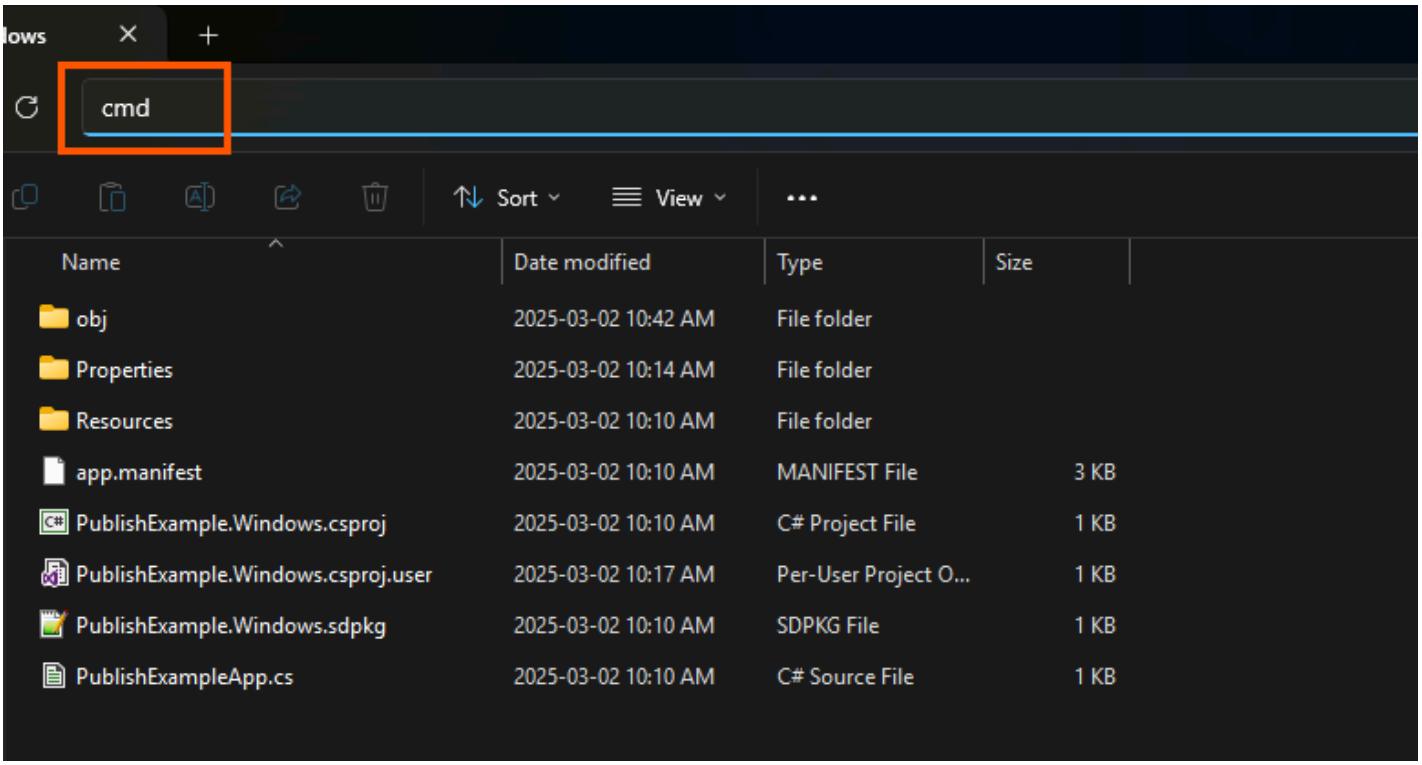
To build using terminal instead of Visual Studio

1. Ensure the relevant .NET SDK is installed (Stride 4.3 is on .NET 10)

2. Open the folder of your project where the `*.Windows.csproj` file sits.



3. Type `cmd` in the search bar to open the folder easily in terminal.



4. Finally publish with the command

```
dotnet publish
```

or the below to include the .NET runtime with your game

```
dotnet publish -r win-x64 --self-contained  true --  framework net10.0-windows
```

You can also append `--output <YOUR_EXPORT_FOLDER>` to specify where to export to.

2. Delete unnecessary files

In the release folder in your project bin folder (eg *MyGame/Bin/MyPlatform/Release*), you can delete the following unnecessary files:

- `.pdb` files (debug information)
- `.xml` files (API documentation)
- files that contain `vshost` in their filenames (eg `MyGame5.vshost.exe` and `MyGame5.vshost.exe.manifest`)
- folders other than the `x64`, `x86`, or `data` folders
- other unnecessary files, such as custom configuration files (ie files not created with Stride)

3. Distribute your game

After you create a release build, how you distribute it is up to you.

To run games made with Stride on Windows, users need:

- .NET 10 Runtime (Unless you published with **self-contained**)
- DirectX11 (included with Windows 10 and later), OpenGL, or Vulkan
- Visual C++ 2015 runtimes (x86 and/or x64, depending on what you set in your project properties in Visual Studio)

See also

- Add or remove a platform
- Version control
- Project structure
- Microsoft documentation

Game Studio

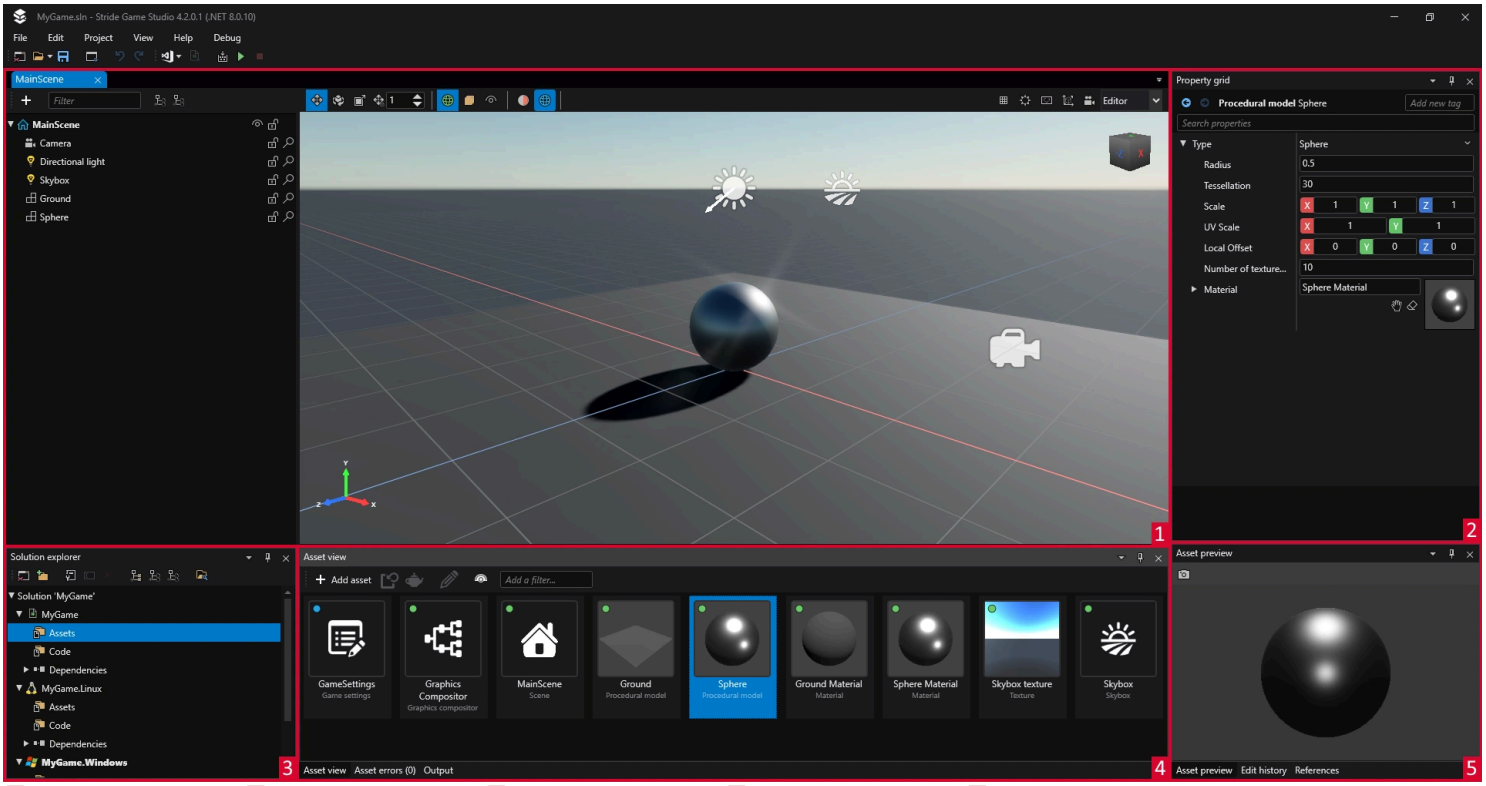
Beginner

Game Studio is the central tool for game and application production in Stride. In Game Studio, you can:

- create and arrange scenes
- import assets, modify their parameters and see changes in real time in the preview window
- organize assets by folder, attach tags and get notifications from modified assets on the disk
- build a game executable and run it directly

Game Studio is also integrated with your Visual Studio projects, so you can seamlessly sync and switch between them.

Interface



1 Asset Editor 2 Property Grid 3 Solution Explorer 4 Asset View 5 Asset Preview

The **asset editor** (1) is used to edit assets and scenes. Some asset types, such as scenes, have dedicated editors where you can make complex changes to the asset. To open a dedicated editor (when available), double-click the asset or right-click it and select **Edit asset**.

The **Property Grid** (2) displays the properties of the asset or entity you select. You can edit the properties here.

The **Solution Explorer** (3) displays the hierarchy of the elements of your project, such as assets, code files, packages and dependencies. You can create folders and objects, rename them, and move them.

The **Asset View** (4) displays the project assets. You can create new assets using the **New Asset** button or by dragging and dropping resource files into the Asset View. You can also drag and drop assets from the Asset View to the different editors or the Property Grid to Create an instance of the asset or add a reference to it. By default, the Asset View is in the bottom center.

The **Asset Preview** tab (5) displays a preview of the selected asset. The preview changes based on the type of the asset you have selected. For example, you can play animations and sounds. This is a quick way to check changes to an asset when editing it in the Property Grid. By default, the Asset Preview is in the bottom right.

You can show and hide different parts of the Game Studio in the View menu. You can also resize and move parts of the UI.

In this section

- Scenes
  - Create a scene
  - Navigate in the Scene Editor
  - Manage scenes
  - Load scenes
  - Add entities
  - Manage entities
- Assets
  - Create assets
  - Use assets
  - Archetypes
  - Game settings
- Prefabs
  - Create a prefab
  - Use prefabs
  - Edit prefabs
  - Nested prefabs
  - Override prefab properties
- World units

Scenes

Beginner Level designer

Scenes are the levels in your game. A scene is composed of **entities**, the objects in your project.

The screenshot below shows a scene with a knight, a light, a background, and a camera entity:

Scenes are a type of asset. As they are complex assets, they have a dedicated editor, the **Scene Editor**.

In this section

- Create and open a scene
- Navigate in the Scene Editor
- Manage scenes
- Load scenes
- Add entities
- Manage entities

Create and open a scene

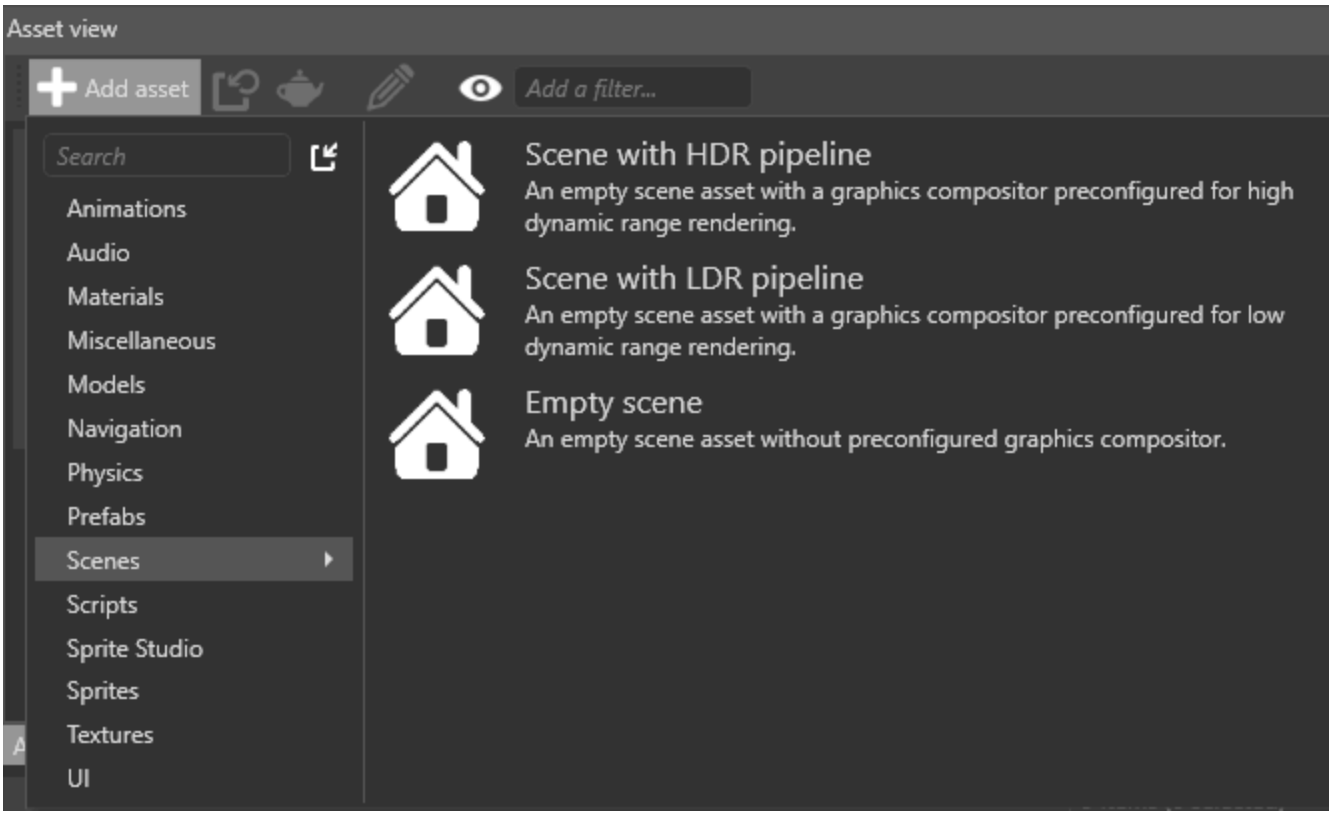
Beginner Level Designer

When you create a new project, Game Studio creates an initial scene and populates it with basic entities such as a light, a camera, and a skybox.

You can create scenes like any other asset. As they are complex assets, they have a dedicated editor, the **Scene Editor**.

Create a scene

1. In the **Asset View** (by default in the bottom pane), click **Add asset** and select **Scenes**.

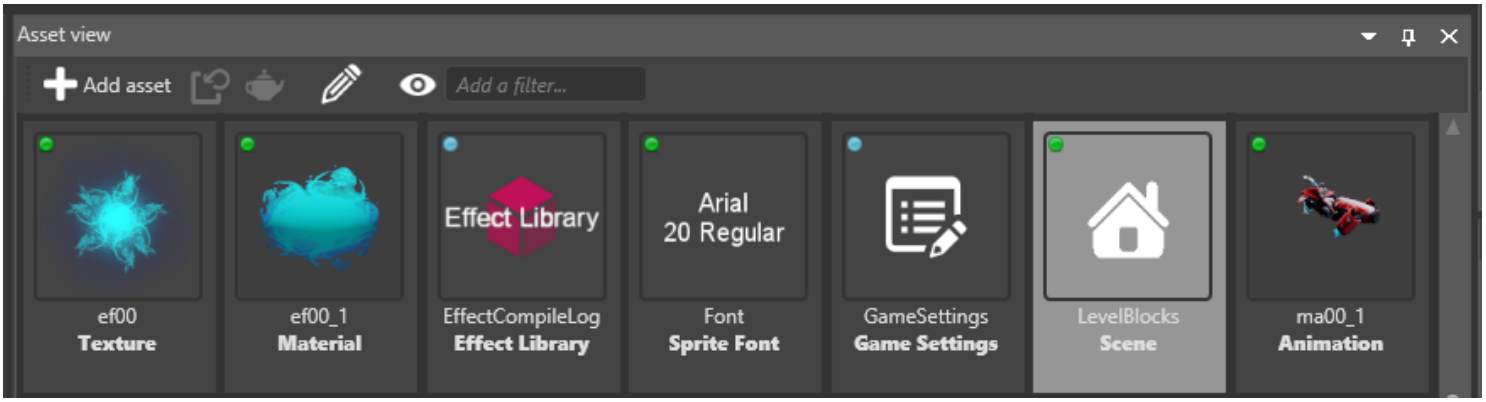


2. Select the appropriate **scene template**.

Template	Result
Empty scene	An empty scene with no entities or preconfigured rendering pipeline
Scene with HDR pipeline	A scene containing basic entities and preconfigured for HDR rendering
Scene with LDR pipeline	A scene containing basic entities and preconfigured for LDR rendering

Open a scene in the Scene Editor

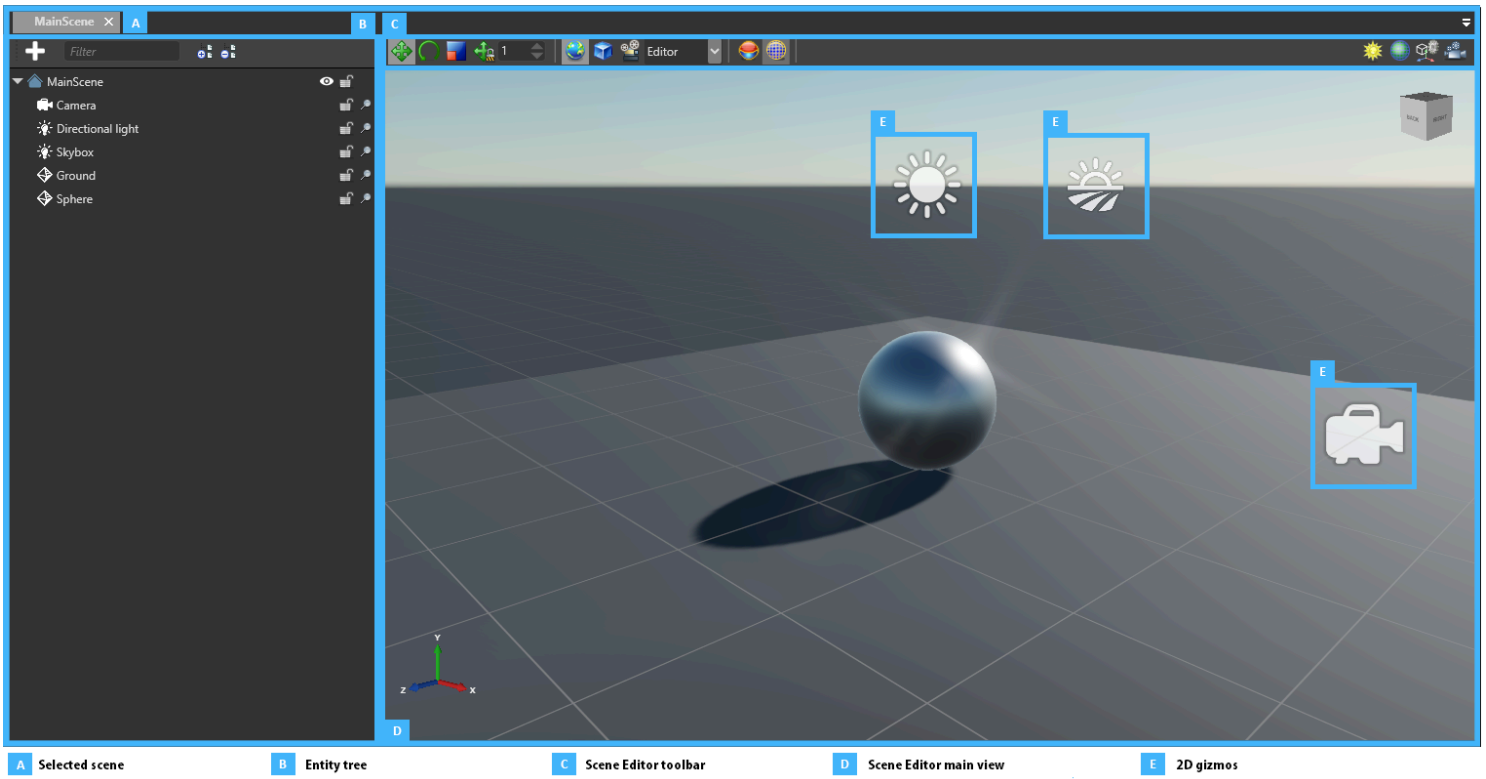
In the **Asset View**:



- double-click the scene asset, or
- right-click the asset and select **Edit asset**, or
- select the asset and type **Ctrl + Enter**

> **TIP**
>
> You can have several scenes open simultaneously.

Use the Scene Editor



The **Scene Editor tabs** (A) display the open scenes. You can switch between open scenes using the tabs.

The **Entity Tree** (B) shows the hierarchy of the entities included in the scene. The same entity hierarchy is applied at runtime. You can use the Entity Tree to browse, select, rename, and reorganize your entities.

You can use the **tool bar** (C) to modify entities and change the Scene Editor display.

The **main window** (D) shows a simplified representation of your scene, with your entities positioned inside it. For entities that have no shape (E), Game Studio represents them with **2D gizmos**; for example, cameras are represented with camera icons.

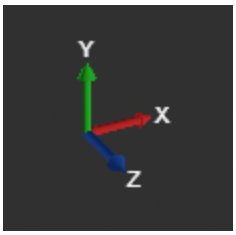
See also

- Navigate in the Scene Editor
- Manage scenes
- Load scenes
- Add entities
- Manage entities

Navigate in the Scene Editor

Beginner Level designer

You can move around the scene and change the perspective of the editor camera. The XYZ axes in the bottom left show your orientation in 3D space.



Move around in the scene

There are several ways to move the editor camera around the Scene Editor.

> **TIP**
> Holding the **Shift** key speeds up movement.

Fly

0:00

Hold the **right mouse button** and **move the mouse** to change the camera direction. Hold the **right mouse button** and use the **WASD keys** to move. This is similar to the controls of many action games.

Pan

Hold the **right mouse button** and the **center mouse button** and move the mouse.

Dolly

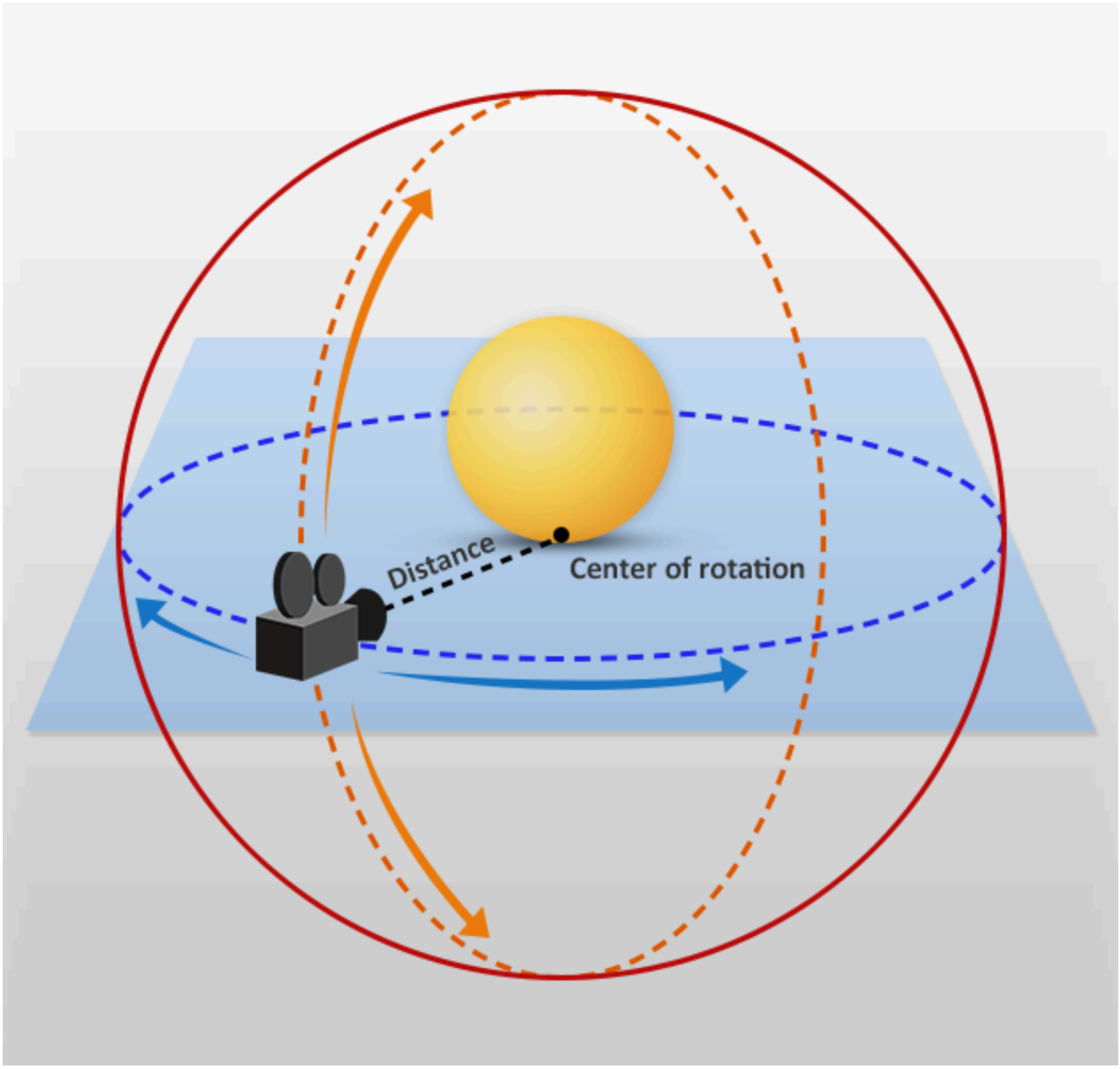
0:00

To dolly (move the camera forward and backward), use the **mouse wheel**.

Orbit

Hold **Alt** and the **left mouse button** and move the **mouse**.

The point of rotation is always the center of the screen. To adjust the distance to the center, use the **mouse wheel**.



0:00

Focus on an entity

0:00

After you select an entity, press the **F** key. This zooms in on the entity and centers it in the camera editor.

You can also focus by clicking the **magnifying glass icon** next to the entity in the Entity Tree.

> **TIP**
> Focusing and then orbiting with **Alt + left mouse button** is useful for inspecting entities.

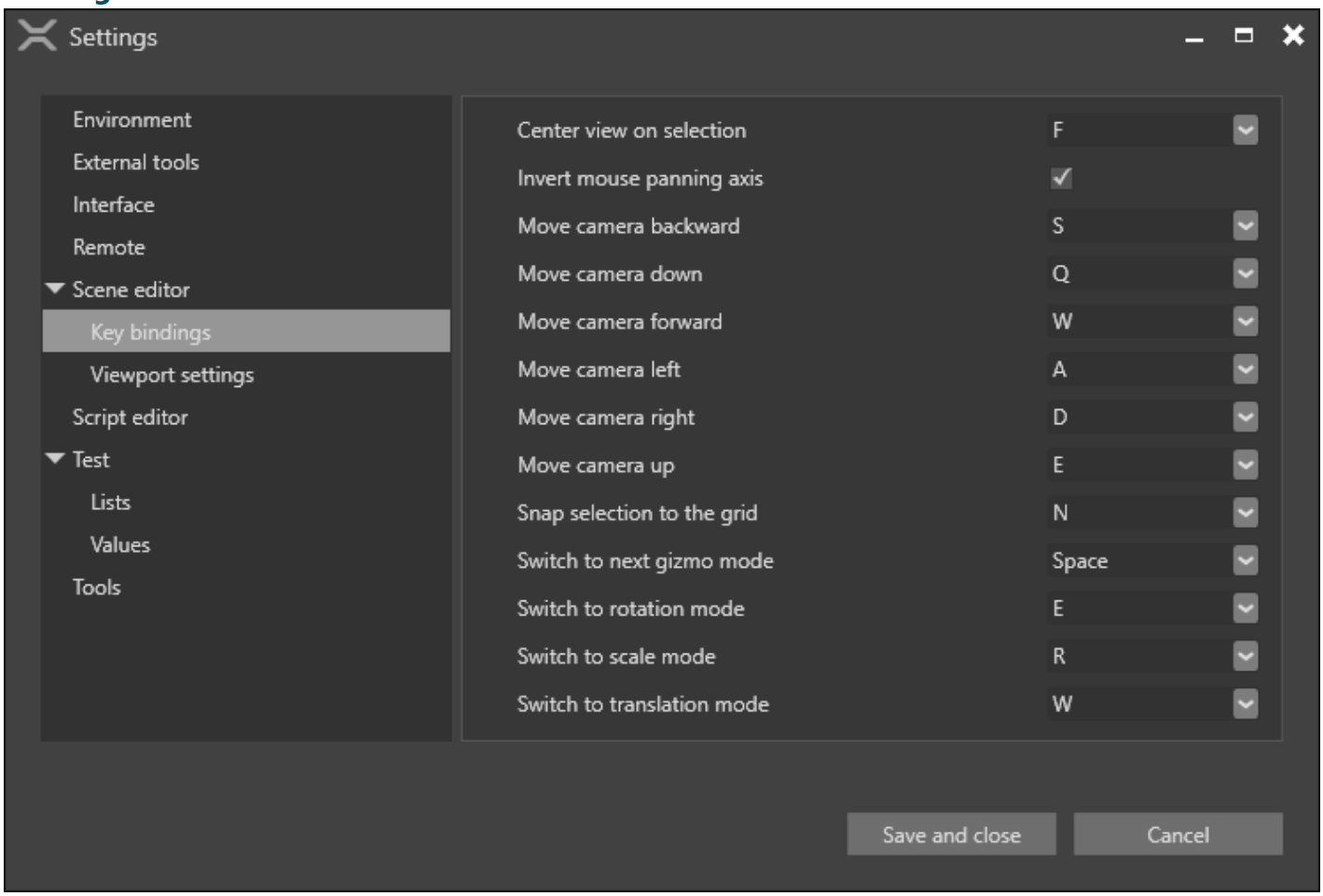
Controls

Action	Control
Move	Arrow keys + right mouse button WASDQE keys + right mouse button
Look around	Hold right mouse button + move mouse
Dolly	Middle mouse button + right mouse button + move mouse
Orbit	Alt key + left mouse button
Zoom	Mouse wheel Alt + Right mouse button + move mouse

Action	Control
Pan	Middle mouse button + move mouse
Focus	F (with entity selected)

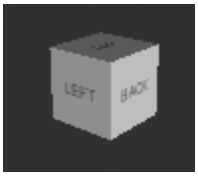
TIP

You can change the scene navigator controls in **Edit > Settings** under **Scene Editor > Key bindings**.



Change camera editor perspective

You can change the camera editor perspective using the **view camera gizmo** in the top-right of the Scene Editor.



Snap camera to position

To change the angle of the editor camera, click the corresponding face, edge, or corner of the **view camera gizmo**.

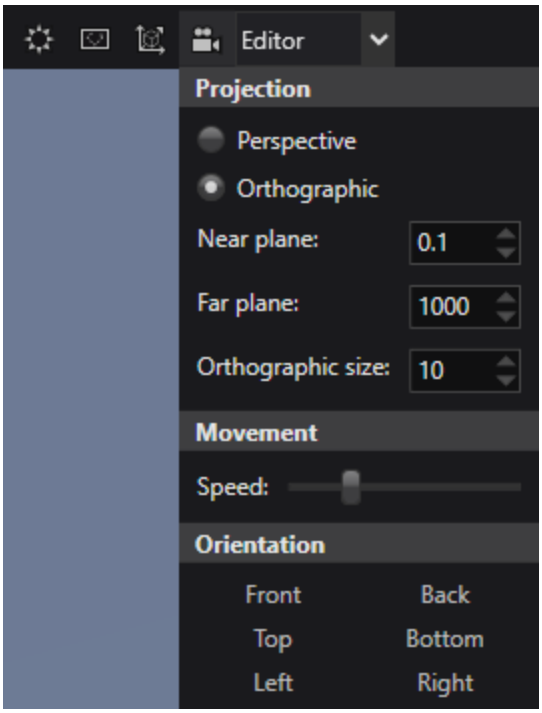
Click	Camera position
Face	Faces the selected face
Edge	Faces the two adjacent faces at a 45° angle
Corner	Faces the three adjacent faces at a 45° angle

0:00

Camera options

> **NOTE**
>
> This page explains how to use the Scene Editor camera. For information about how to use cameras in your game, see Graphics — Cameras.

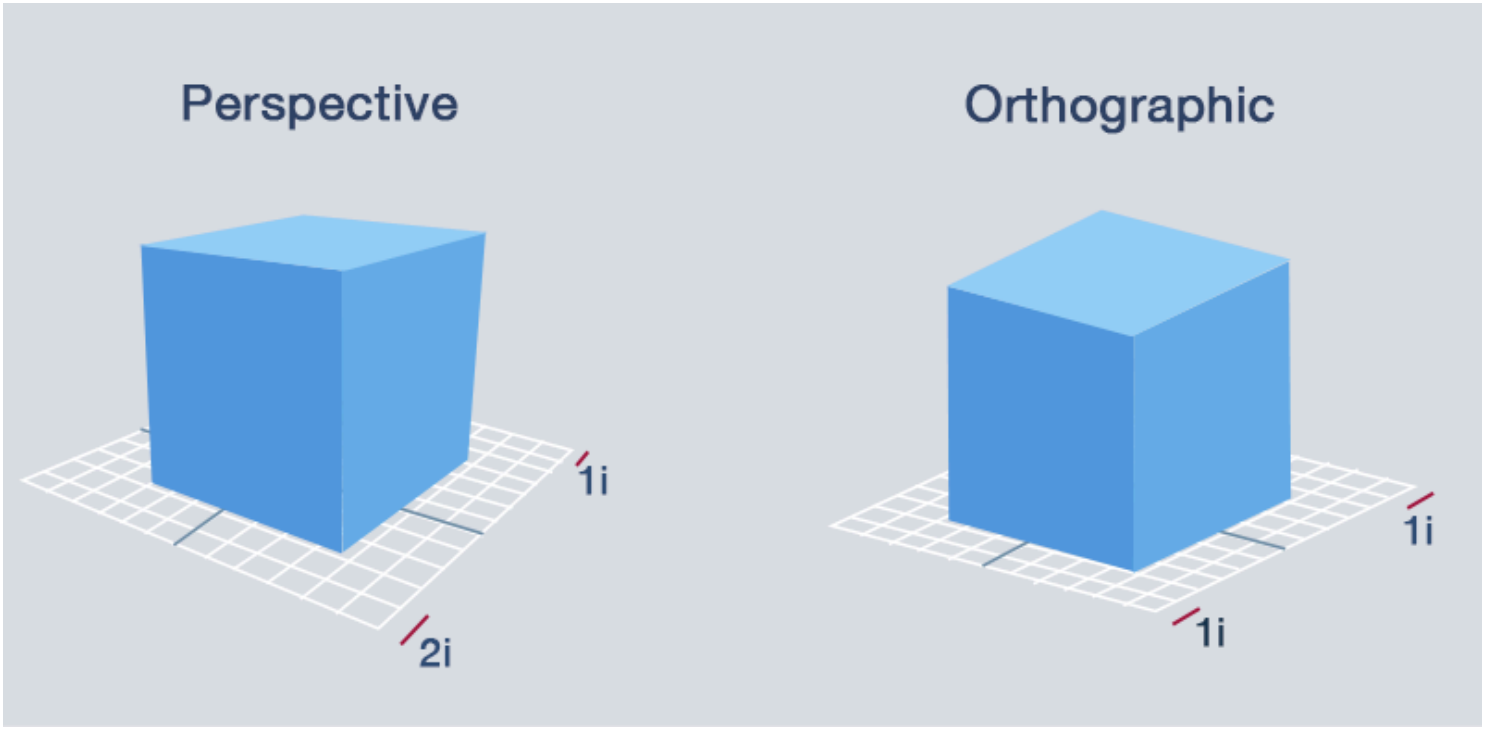
To display the Scene Editor camera options, click the **camera icon** in the top-right of the Scene Editor.



Perspective and orthographic views

Perspective view is a "real-world" perspective of the objects in your scene. In this view, objects close to the camera appear larger, and lines of identical lengths appear different due to foreshortening, as in reality.

In **orthographic view**, objects are always the same size, no matter how far their distance from the camera. Parallel lines never touch, and there's no vanishing point. It's easy to tell if objects are lined up exactly in orthographic view.



Perspective Orthographic

You can also switch between perspective and orthographic views by clicking the **view camera gizmo** as it faces you.

0:00

Field of view

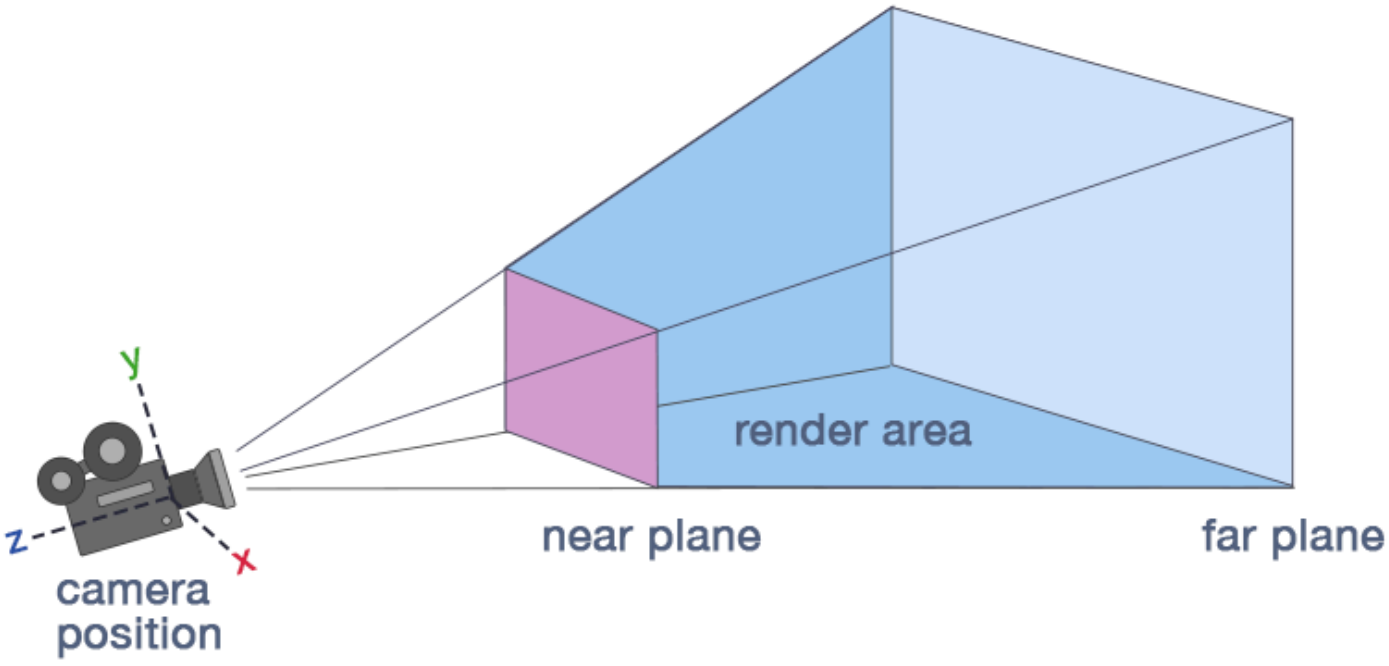
You can change the camera field of view. This changes the camera frustum, and has the effect of zooming in and out of the scene. At high settings (90 and above), the field of view creates stretched "fish-eye lens" views. The default setting is 45.

Near and far planes

The near and far planes determine where the camera's view begins and ends.

- The **near plane** is the closest point the camera can see. The default setting is 0.1. Objects before this point aren't drawn.
- The **far plane**, also known as the draw distance, is the furthest point the camera can see. Objects beyond this point aren't drawn. The default setting is 1000.

Game Studio renders the area between the near and far planes.



Camera speed

The **camera speed** setting changes how quickly the camera moves in the editor.

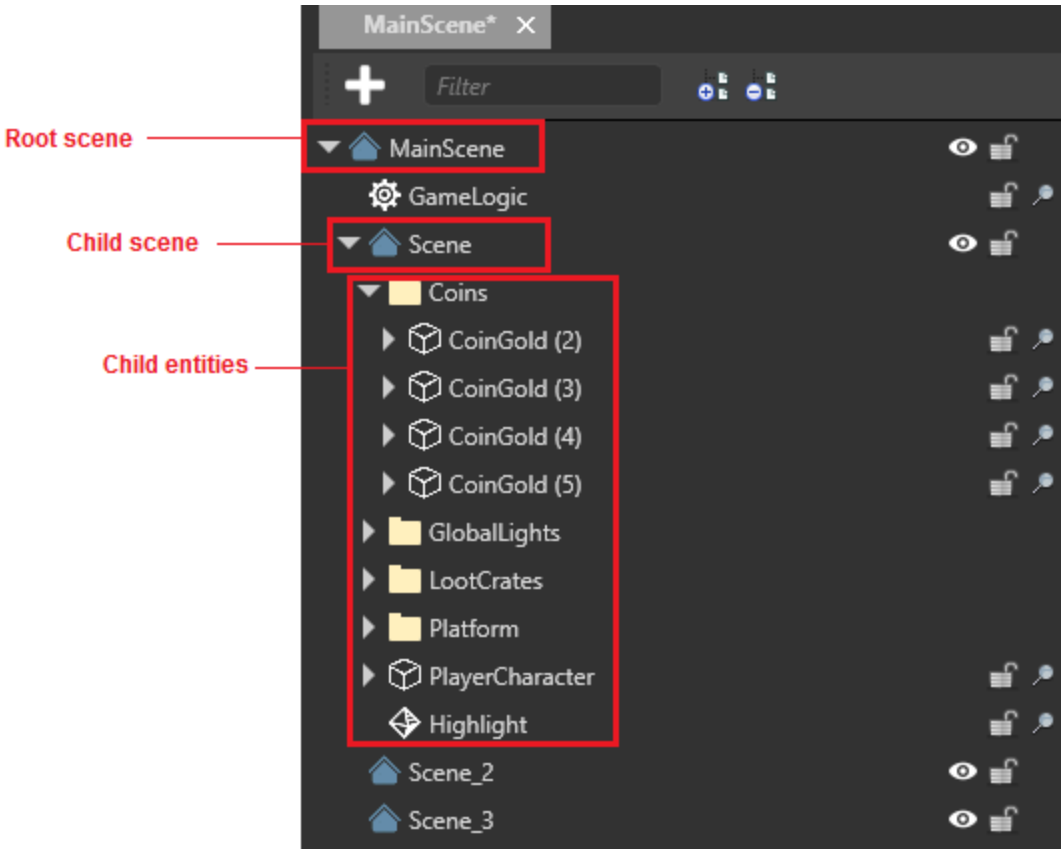
See also

- Create and open a scene
- Load scenes
- Add entities
- Manage entities

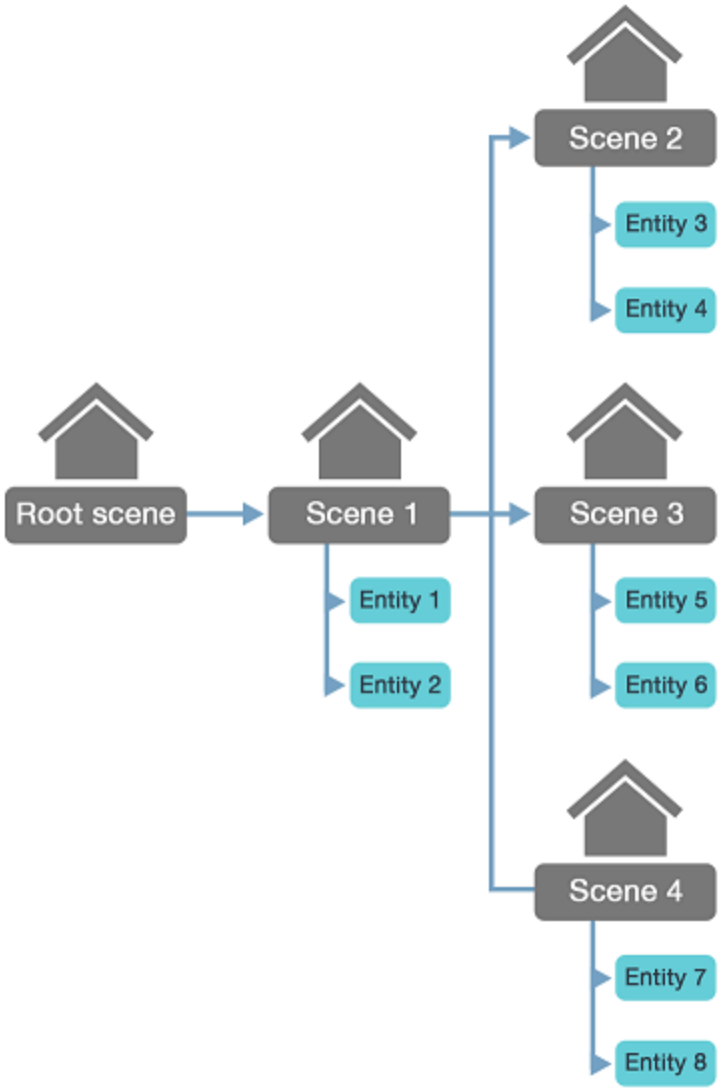
Manage scenes

Beginner Programmer Designer

Scenes and entities are arranged in a hierarchy, with the **root scene** at the top. This hierarchy is displayed in the **Entity Tree** in the Scene Editor on the left.



The root scene contains all the scenes and entities in your game. It should contain common entities that the other scenes and entities use, such as game logic scripts.



Scenes are kept in different folders. This means that different people can work on them without overwriting each other's work.

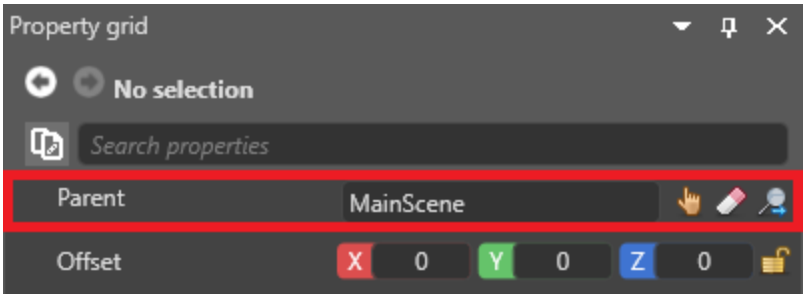
> **NOTE**
>
> When scenes load at runtime, their **child scenes aren't automatically loaded too**. You have to load child scenes in code. For more information, see Load scenes.

Set parent and child scenes

The relationship between parent and child scenes is set on the child, not the parent. In other words, child scenes know about their parent scenes, but parent scenes don't know about their child scenes.

There are several ways to make a scene a child of another scene:

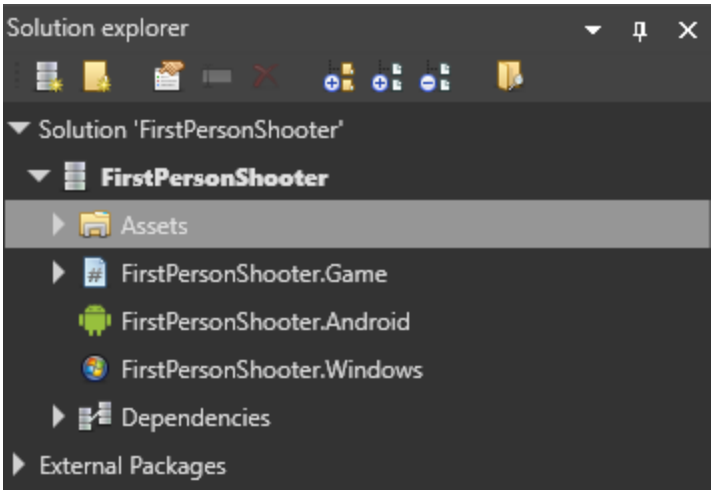
- In the Scene Editor **Entity Tree** (left by default), drag the scene onto the scene you want to make its parent.
- Drag the scene from the **Asset View** (bottom by default) onto the scene you want to make its parent in the **Entity Tree**.
- In the scene **Property Grid** (on the right by default), next to **Parent**, specify the scene you want to be the scene's parent.



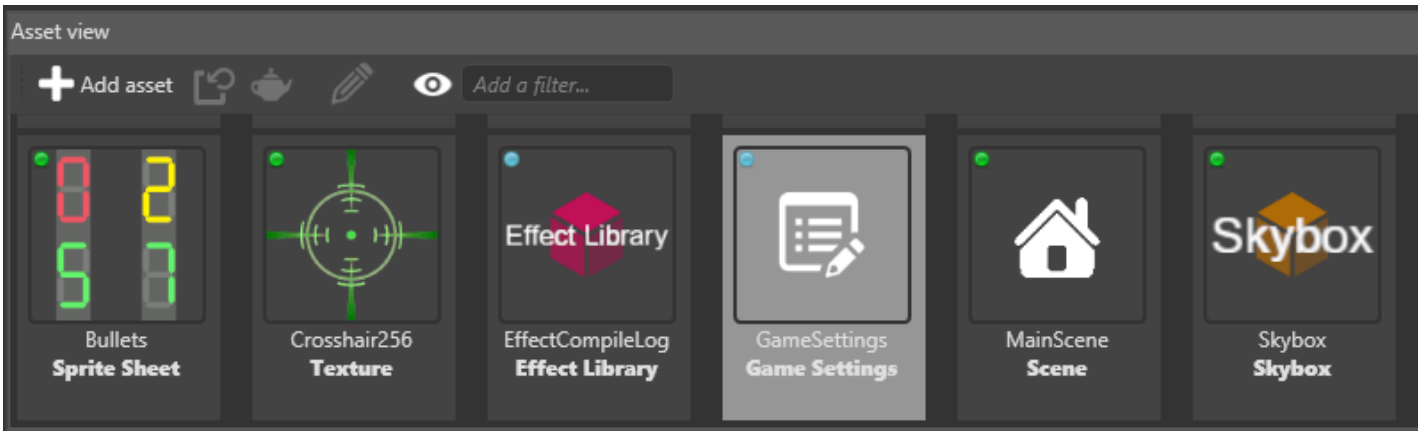
Set the default scene

The **default scene** is the scene Stride loads at runtime. You can set this in the Game Settings asset.

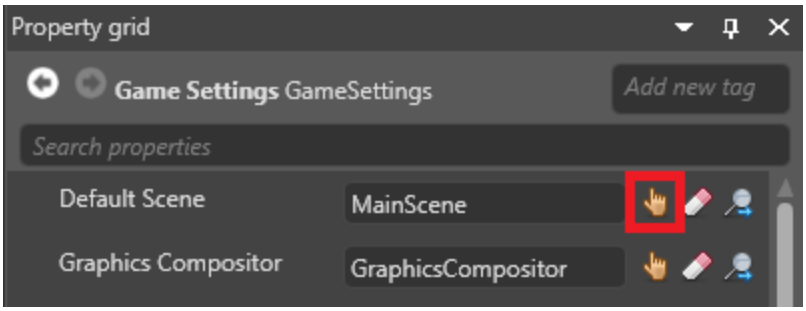
1. In the **Solution Explorer** (the bottom-left pane by default), select the **Assets folder**.



2. In the **Asset View** (the bottom pane by default), select the **GameSettings** asset.



3. In the **Property Grid** (the right-hand pane by default), next to **Default Scene**, click (**Select an asset**).



   The **Select an asset** window opens.

4. Select the default scene and click **OK**.

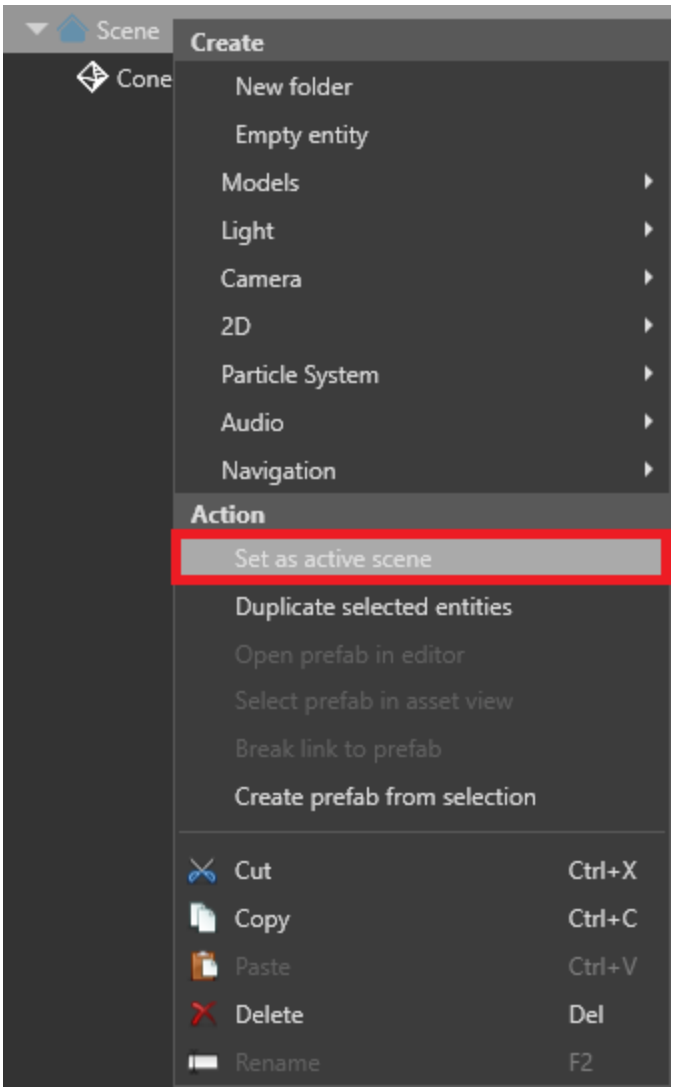
Stride loads this scene at runtime.

For more information about the Game Settings asset, see Game Settings.

Set the active scene

The **active scene** is the scene entities are added to when you drop them in the Scene Editor. Game Studio adds the entities as children to the active scene.

To set the active scene, **Entity Tree** (left by default), right-click the scene and select **active scene**.

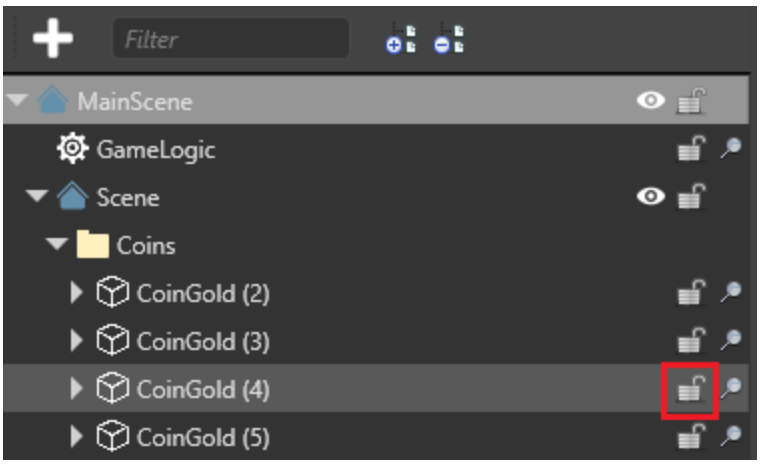


The active scene has no effect on runtime.

Lock scenes and entities

You can lock scenes and entities so they can't be selected in the main window. This is useful when you have lots of things in your scene. You can still select scenes and entities in the Entity Tree.

To lock or unlock a scene or entity, in the Entity Tree, click the **padlock** icon.



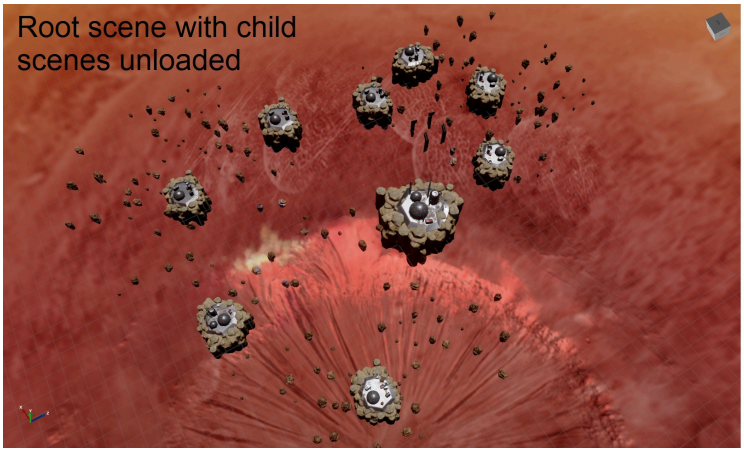
> **TIP**
>
> When you lock a scene, all its child scenes and entities are locked too. To lock an entity along with its child entities, hold **Ctrl** and click the padlock icon.

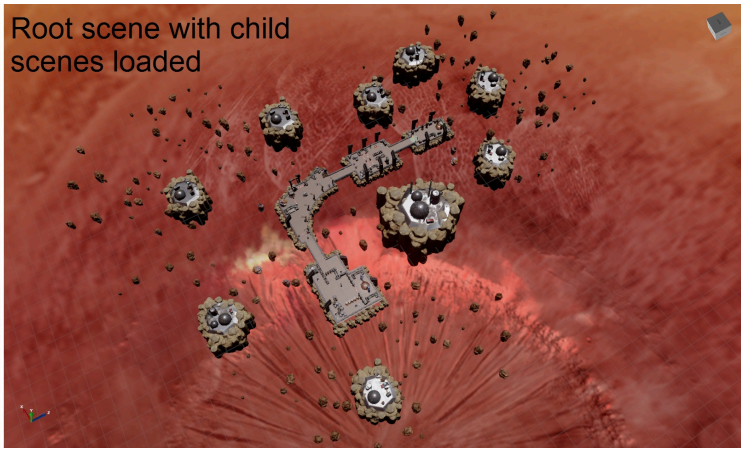
Locked items have a **gold locked padlock** icon in the Entity Tree.

Load and unload scenes in the Scene Editor

You can load and unload scenes (with all their child scenes and entities) in the Scene Editor. Unloading scenes in the editor is useful if, for example, you want to remove clutter from your editing view, or improve editor performance.

The screenshots below show a root scene with child scenes loaded and unloaded. The root scene contains entities that all the scenes use, including the skybox, scripts, asteroids, and player character. The child scenes are sections of level.

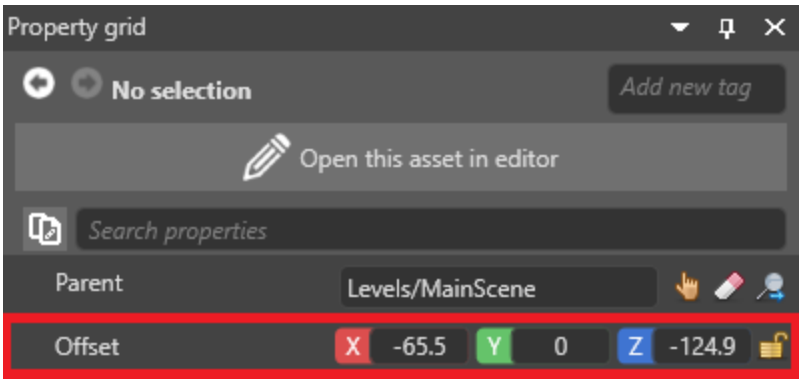




To load or unload a scene, in the **Scene Editor**, in the **Entity Tree** on the left, next to the scene you want to load or unload, click the **eye icon**.

Move a scene

As scenes aren't entities, they don't have transform components. However, you can move a scene using its **offset** property.



To move a scene at runtime, use:

```
myScene.Offset = new Vector3(x, y, z);
```

Replace `myScene` with the name of the scene, and `x,y,z` with the XYZ coordinates you want to move the scene to.

See also

- Create and open a scene
- Navigate in the Scene Editor
- Load scenes
- Add entities
- Manage entities

Load and unload scenes at runtime

Loading scenes

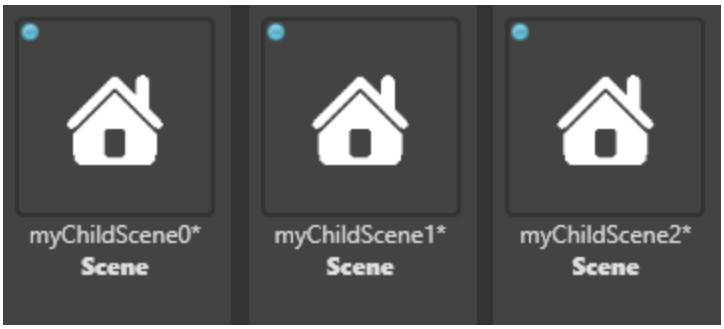
You can use `UrlReference<Scene>` properties on your scripts to assign **Scene** assets then load the via code:

```
public UrlReference<Scene> ChildSceneUrl { get; set; }

//...
var childScene = Content.Load(ChildSceneUrl);

parentScene.Children.Add(childScene);
```

Alternatively you can load scenes by name. The following code loads three scenes and adds them as children:

```
var myChildScene0 = Content.Load<Scene>(url0);
var myChildScene1 = Content.Load<Scene>(url1);
var myChildScene2 = Content.Load<Scene>(url2);

myParentScene.Children.Add(myChildScene0);
myParentScene.Children.Add(myChildScene1);
myChildScene1.Add(myChildScene2);
```

> **NOTE**
>
> If you are not using `UrlReference` make sure all the scenes you want to load are included in the build as **root assets** (indicated with blue icons in the **Asset View**).



> To include a scene in the build, in the **Asset View**, right-click the scene asset and select **Include in build as root asset**.

> For more information about including assets in the build, see Manage assets.

For more information about scene hierarchies, see Manage scenes.

Unloading scenes

Before a scene is unloaded remove it from the scene hierarchy:

```
parentScene.Children.Remove(childScene);

//Alternatively
childScene.Parent = null;
```

Once the scene asset is no longer required make sure to unload it:

```
Content.Unload(childScene);
```

Scene streaming script

Stride also includes a scene streaming script that uses a trigger to load scenes.

> **NOTE**
> The scene streaming script is included as an example. It isn't always the most appropriate way to load scenes. Feel free to modify it as much as you need.

Add a scene streaming script

To add a scene streaming script, in the **Asset View** (bottom pane by default), click **Add asset** and select **Scripts > Scene streaming**.

Game Studio adds a scene streaming script to your project assets.

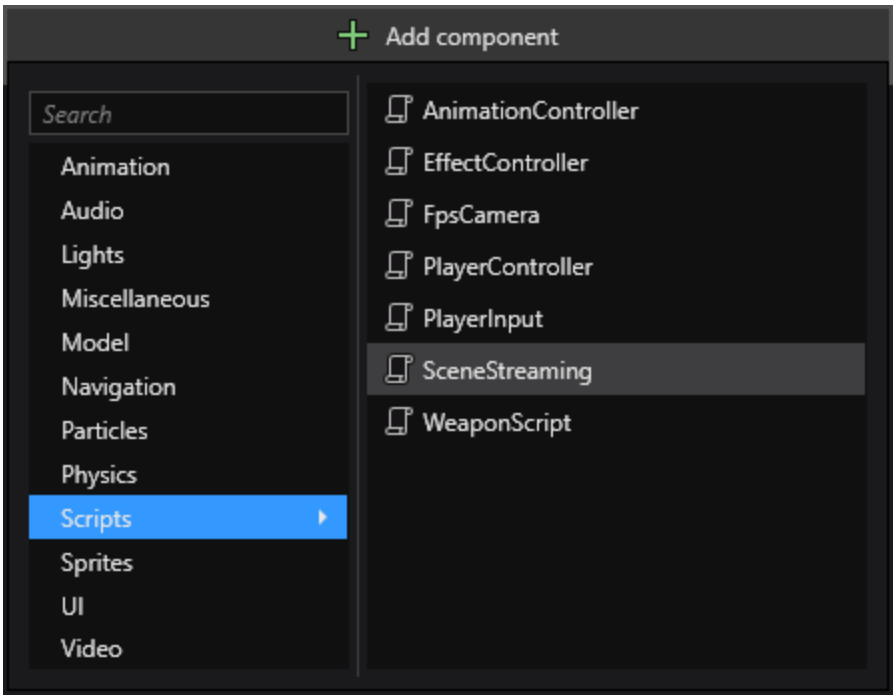
Use the scene streaming script

1. Create a trigger entity. When this is triggered at runtime, Stride loads the scene. For more information about creating triggers, see Triggers.

   How the entity is triggered is defined in the collider properties. For more information, see Colliders.

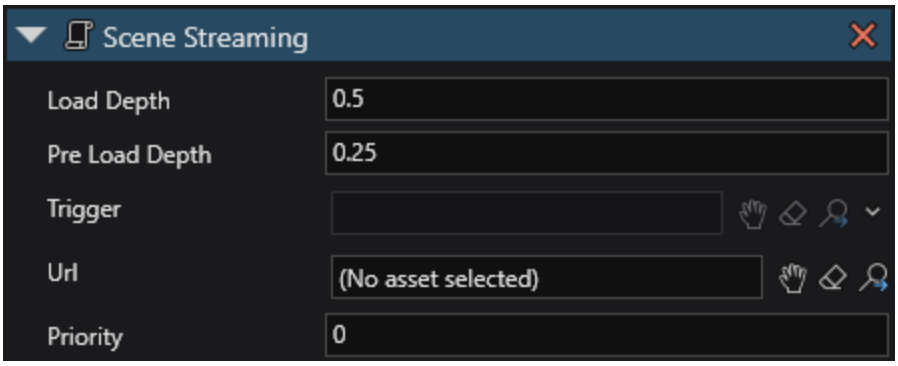
2. Create an entity and position it where you want the scene to load.

3. With the entity selected, in the **Property Grid** (on the righy by default), click **Add component** and select the **scene streaming** script.



> **NOTE**
> If the scene streaming script doesn't appear in the list of components, reload the assemblies.

Game Studio adds the script to the entity as a component.

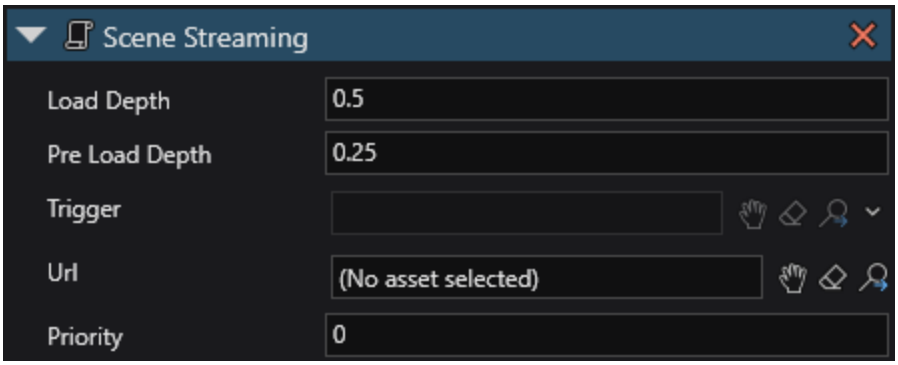


4. Under **Url**, specify the scene asset you want to load.

5. Under **Trigger**, specify the entity you created in step 1.

At runtime, when the trigger you created in step 1 is triggered, Stride loads the scene you specified in step 4.

Scene streaming script properties



Property	Description
Pre Load Depth	The point (in world units) at which the scene begins to load. For example, if *2.5*, the scene begins to load when the player is 2.5 units into the trigger area
Load Depth	The point (in world units) at which the game freezes to finish loading the scene if it hasn't loaded already. For example, if *5*, the game freezes when the player is 5 units into the trigger area
Priority	The script priority. For more information, see Scheduling and priorities

See also

- Colliders
- Triggers
- Create and open a scene
- Navigate in the Scene Editor
- Manage scenes
- Add entities
- Manage entities

Add entities

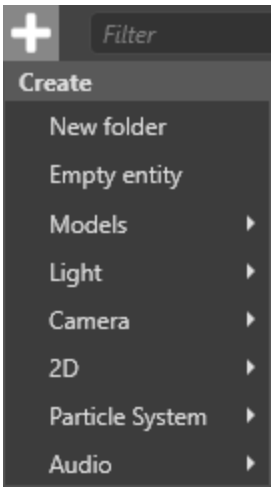
Beginner Level Designer

After you create a scene, you need to add entities to your scene to build your level.

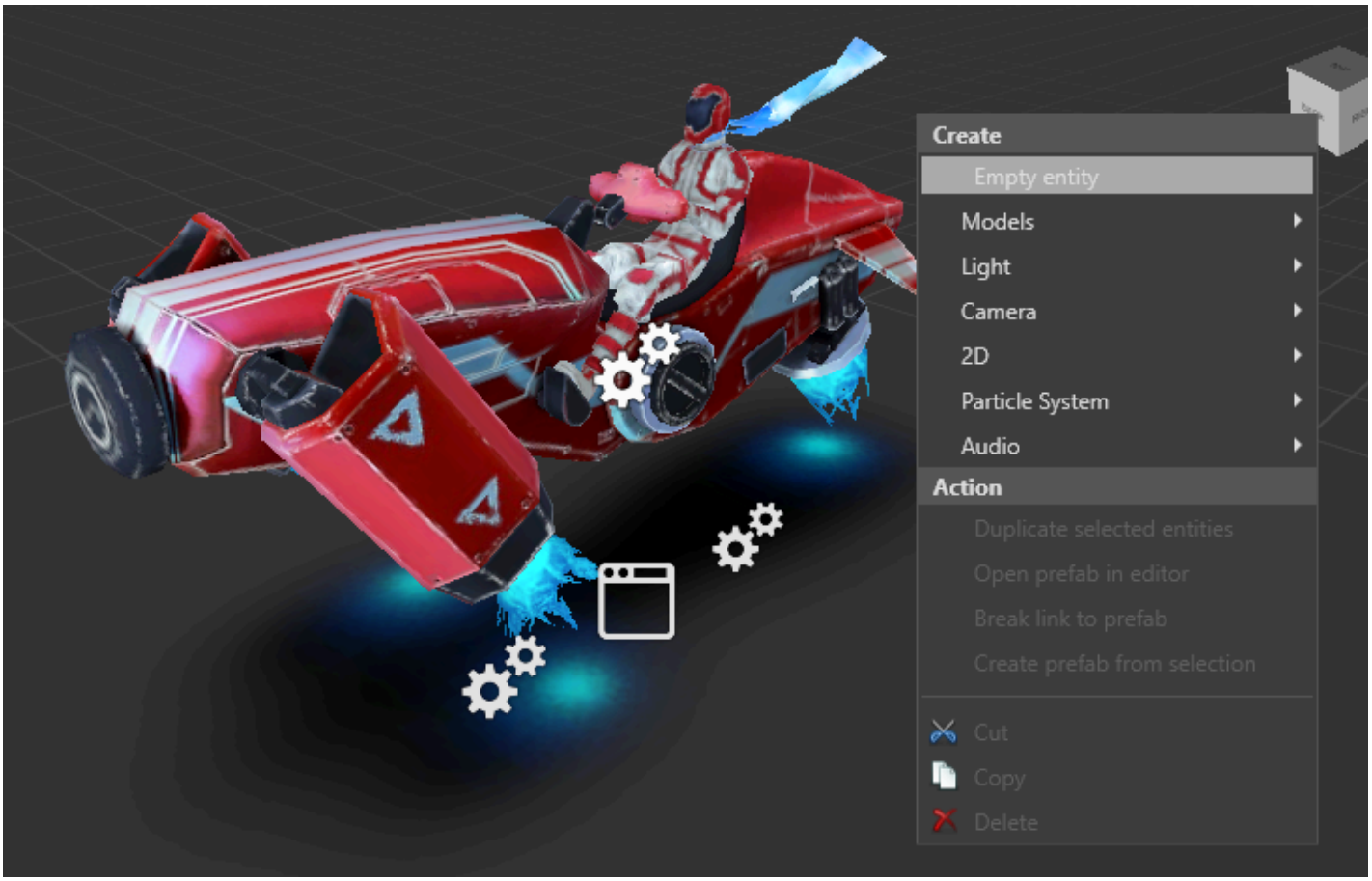
Create an entity from the Scene Editor

1. Above the **Entity Tree**, click the icon.

   The **Create** menu opens:

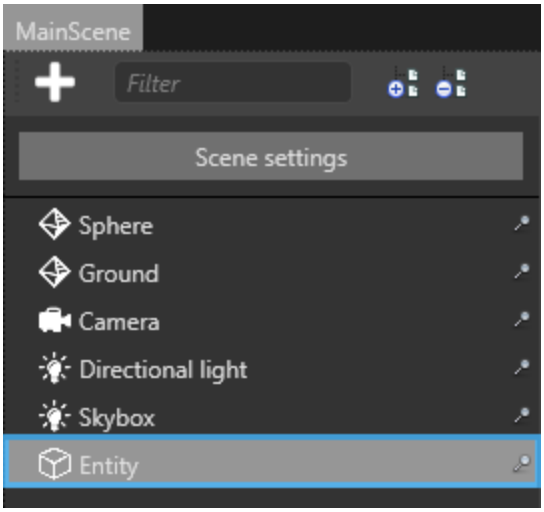
   

   Alternatively, right-click the **Entity Tree** or anywhere in the **scene**. If you create an entity in the scene, Game Studio adds an entity in the location you click.

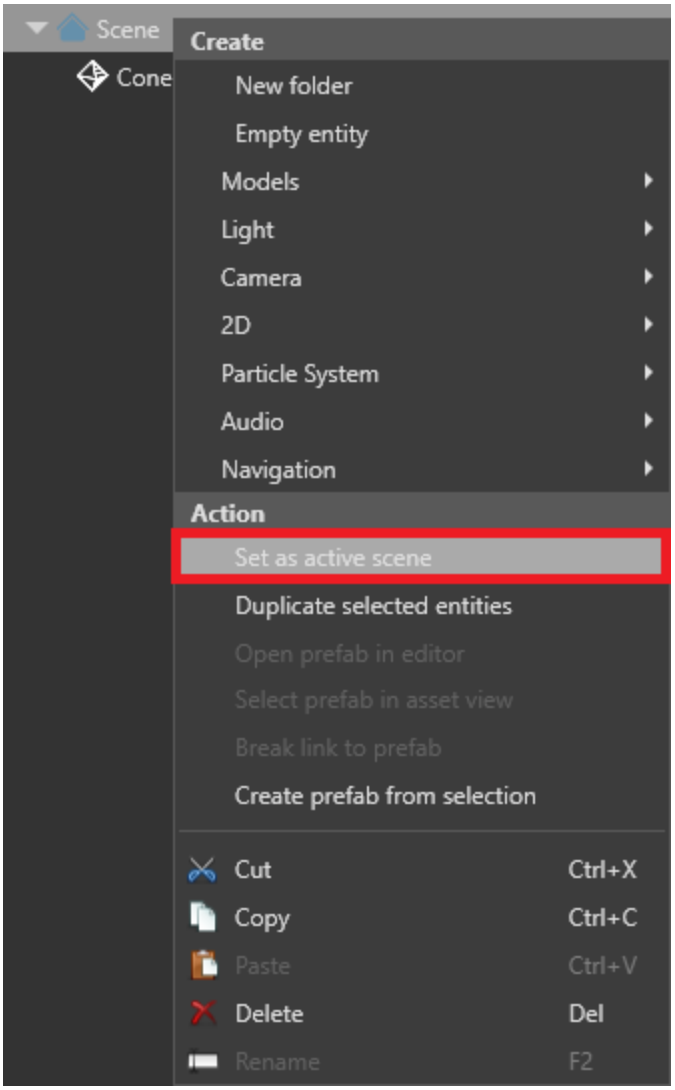


2. Select **Empty entity**, or select an entity template.

   Game Studio adds an entity to the active scene and displays it in the Entity Tree:



> **TIP**
>
> The **active scene** is the scene entities are added to. To set the active scene, **Entity Tree** (left by default), right-click the scene and select **active scene**.



The active scene has no effect on runtime.

Create an entity from an asset

You can add an entity by dragging and dropping an asset from the **Asset View** to the scene.

0:00

Game Studio automatically creates an entity and adds the required component and reference based on the asset you used. For example, if you drag a model asset to the scene, Game Studio creates an entity with a model component with the model asset as its reference.

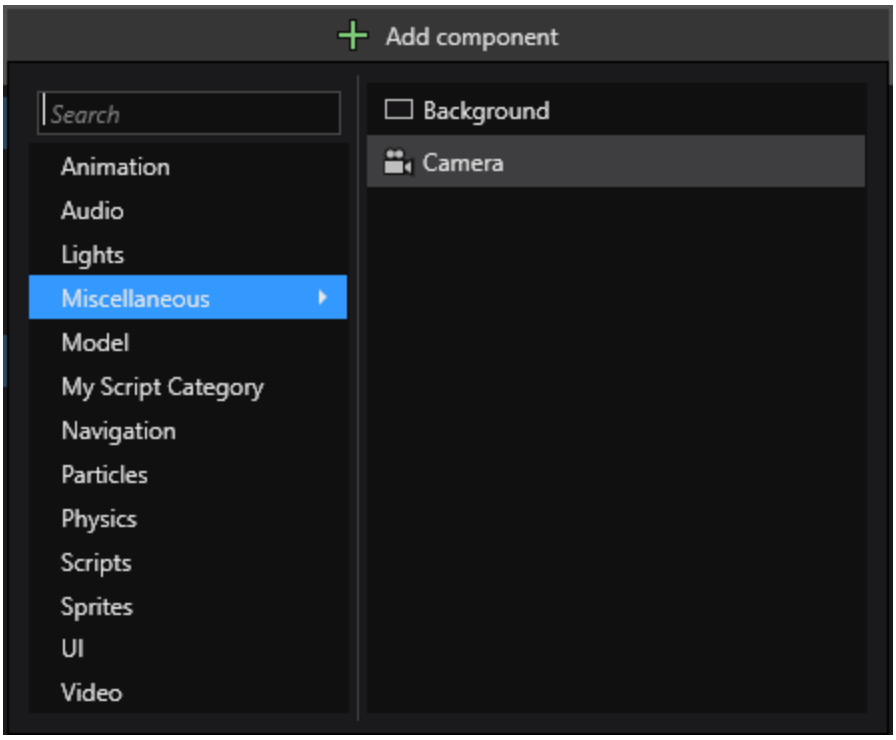
> **NOTE**
> You can only create entities by dragging assets with corresponding components. For example, model components use model assets, so can be dragged; animations have no corresponding component, so can't be dragged.

Set up a component

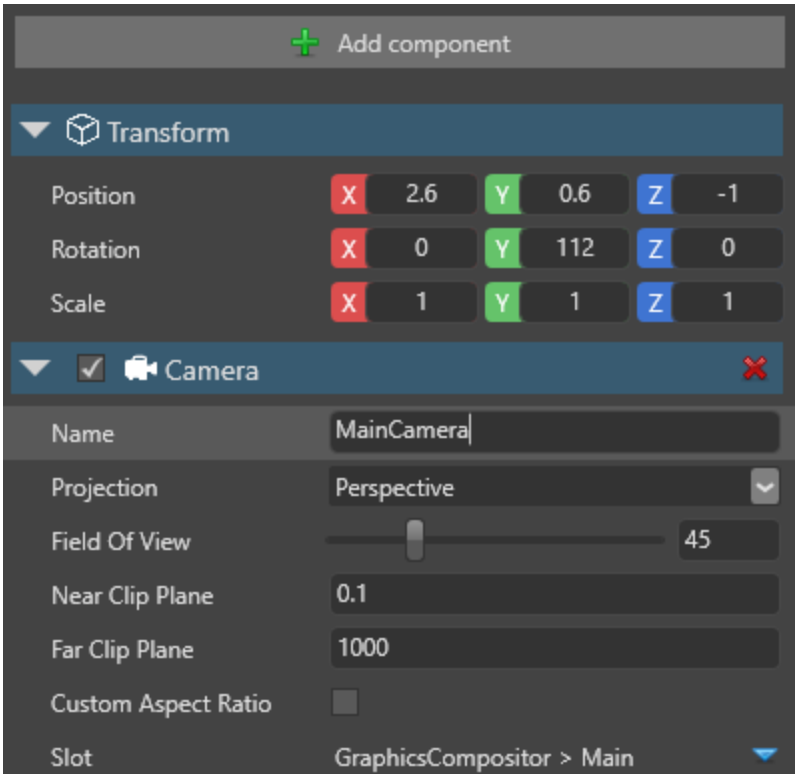
Components add special properties to entities that define their purpose in your project. For example, you add lights to your scene by adding Light components to entities, add models by adding model components, and so on. An entity with no component has no purpose.

To add a component to an entity:

1. Select the entity.
2. In the Property Grid, click **Add component**, and add component you want.



Game Studio adds the component.



3. **Set the properties** of your new component.

Duplicate an entity

You can duplicate an entity along with all its properties. Duplicating an entity and then modifying the properties of the new entity is often faster than creating an entity from scratch.

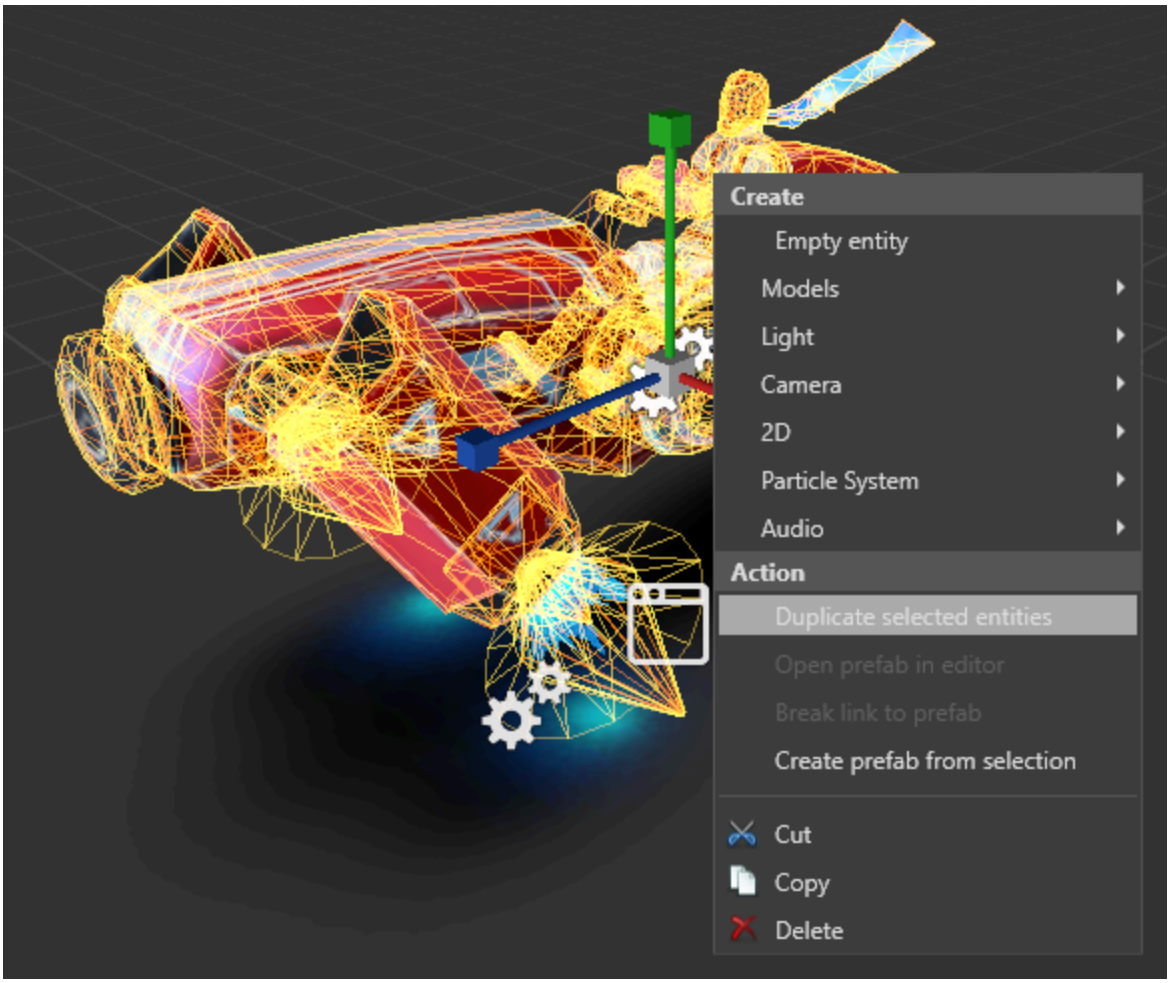
1. Select the entity you want to duplicate.

2. Hold **Ctrl** and move the entity with the mouse.

   The entity and all its properties are duplicated.

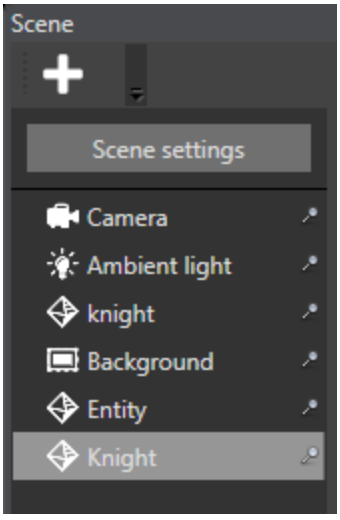
0:00

Alternatively, right-click the entity and select **Duplicate selected entities**.



Rename an entity

1. Select the entity and press **F2**.
2. Type a name for the entity, and then press **Enter**.



See also

- Manage scenes

Manage entities

Beginner Level designer

To build the levels of your game, you need to translate (move), rotate, and resize entities in your scene. These are known as **transformations**.

Transformation gizmos

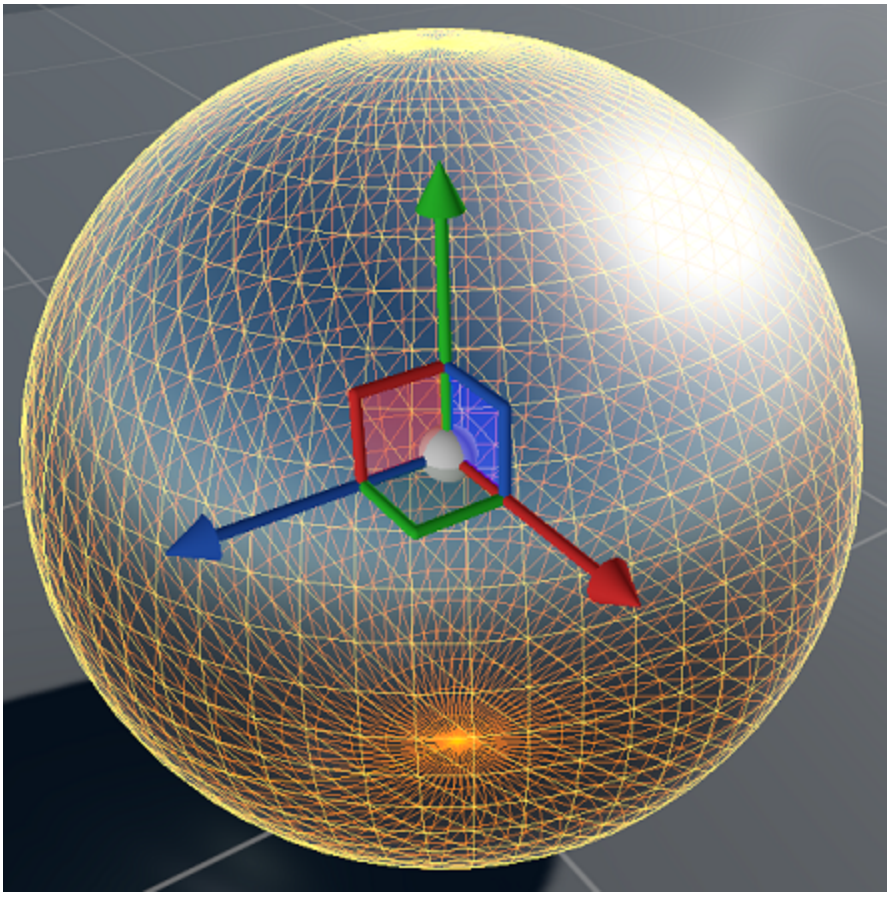
You can select the transformation gizmos from **Scene Editor toolbar**.

Alternatively, press **Space** to switch between gizmos.

There are three types of transformation gizmo:

- The **translation gizmo** moves entities
- The **rotation gizmo** rotates entities
- The **scale gizmo** resizes entities

Game Studio displays the selected transformation gizmo at the origin of the entity.



Translation gizmo

To select the translation gizmo, click the icon in the Scene Editor toolbar or press **W**.

The translation gizmo moves (**translates**) entities in the scene along the axis you select.

- To move an entity along the X axis, drag it by the **red** arrow.
- To move an entity along the Y axis (up and down), drag it by the **green** arrow.
- To move the entity along the Z axis, drag it by the **blue** arrow.
- To move the entity in free 3D, drag it by the central sphere.

0:00

Rotation gizmo

To select the rotation gizmo, click the icon in the Scene Editor toolbar or press **E**.

The rotation gizmo rotates entities in the scene along the axis you select.

- To rotate an entity along the X axis (pitch), drag it by the **red** ring.
- To rotate an entity along the Y axis (yaw), drag it by the **green** ring.
- To rotate the entity along the Z axis (roll), drag it by the **blue** ring.

0:00

Scale gizmo

To select the scale gizmo, click the icon in the Scene Editor toolbar or press **R**.

The scale gizmo resizes entities along a single axis ("stretching" or "squashing" them) or all axes (making them larger or smaller without changing their proportions).

- To resize an entity along the X axis, drag it by the **red** ring.
- To resize an entity along the Y axis, drag it by the **green** ring.
- To resize the entity along the Z axis, drag it by the **blue** ring.
- To resize the entity in all axes, drag it by the central sphere.

0:00

> **NOTE**
>
> The scale gizmo only works with the **local** coordinate system (see below). When you select the scale gizmo, Game Studio switches to local coordinates.

Change gizmo coordinate system

You can change how the gizmo coordinates work.

1. Select the entity whose gizmo coordinates you want to change.
2. In the Scene Editor toolbar, select the coordinate system you want.

Coordinate system	**Function**
World coordinates	Uses world coordinates for transformations. The X, Y, and Z axes are the same for every entity.
Local coordinates	Uses local coordinates for transformations. The axes are oriented in the same direction as the selected entity.

Coordinate system	Function
Camera coordinates	Uses the current camera coordinates for transformations. The axes are oriented in the same direction as the editor camera.

Snap transformations to grid

You can "snap" transformations to the grid. This means that the degree of transformation you apply to entities is rounded to the closest multiple of the number you specify. For example, if you set the rotation snap value to 10, entities rotate in multiples of 10 (0, 10, 20, 30, etc).

You can change the snap values for each gizmo in the scene view toolbar. Snap values apply to all entities in the scene. For example:

Icon	Function
	Snap translation to multiple of 1
	Snap rotation to multiple of 22.5
	Snap scale to multiple of 1.1

See also

- Create and open a scene
- Navigate in the Scene Editor
- Load scenes
- Add entities

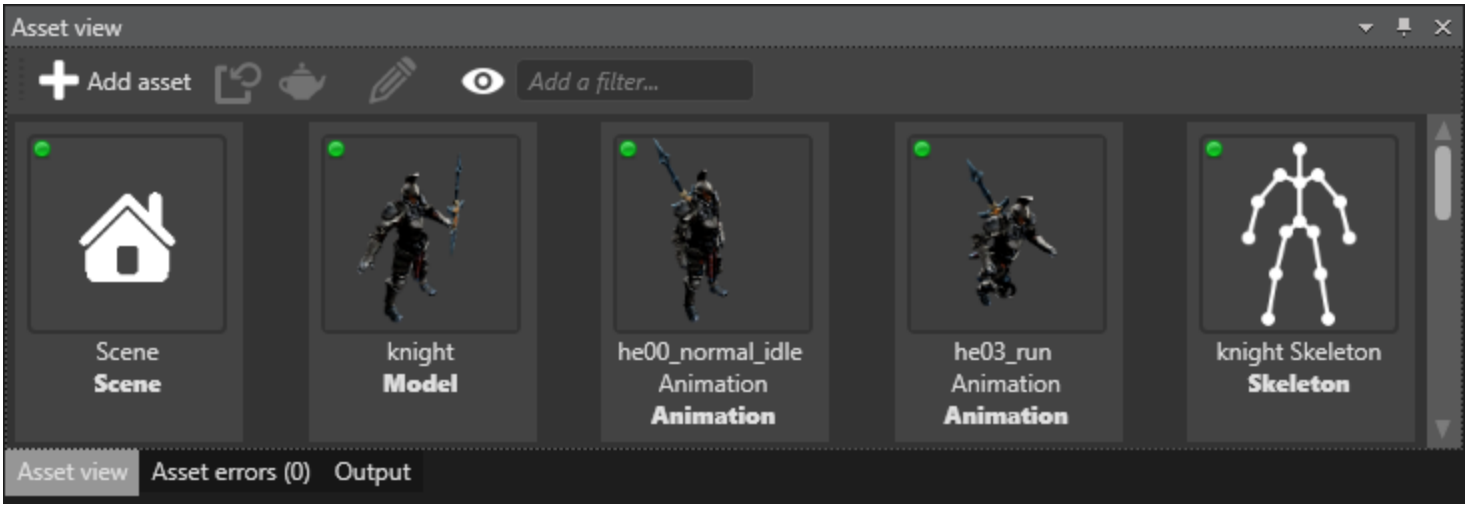
Assets

Beginner

An **asset** is a representation of an element of your game inside Game Studio, such as a texture, animation, or model.

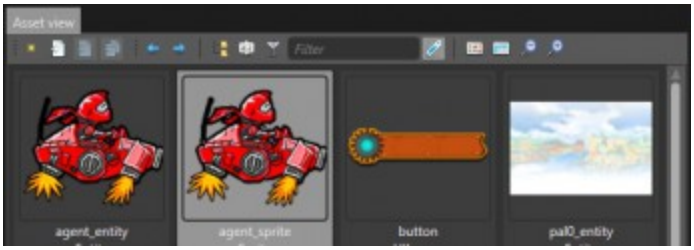
Some assets require **resource files**. For example, texture assets need image files and audio assets need audio files. Other types of assets (such as scenes, physics colliders, and game settings) don't use resource files, and can be created entirely in Game Studio.

You can compile and optimize assets with a special compiler provided by Stride. Compiled assets are packed together as reusable bundles.

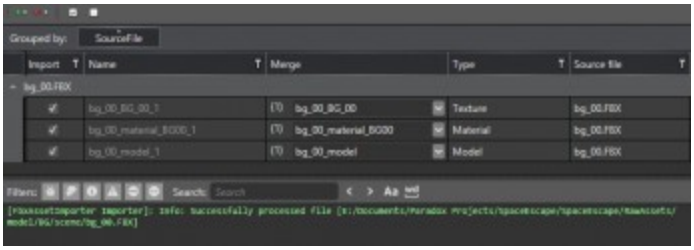


You can:

- create and browse assets in the **Asset View**

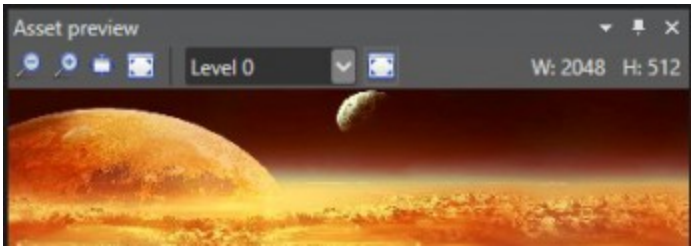


- import existing assets, such as FBX files



- edit assets in the **property editor**

- see a live preview in the **Asset Preview**



In this section

- Create assets
- Manage assets
- Use assets

Create assets

Beginner

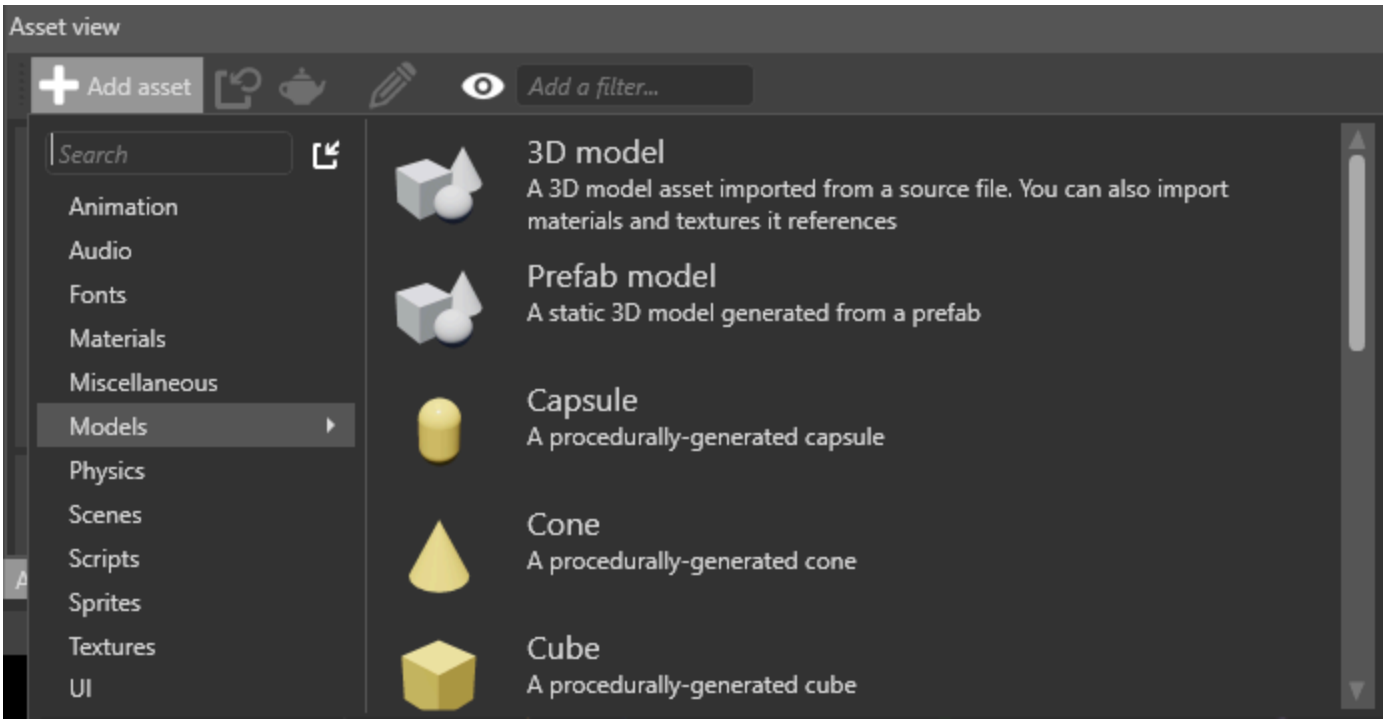
There are two ways to create assets:

- Use the **Add asset** button in the **Asset View**
- Drag and drop **resource files** (such as image or audio files) to the **Asset View** tab

Use the **Add asset** button

1. In the *Asset View*, click Add asset

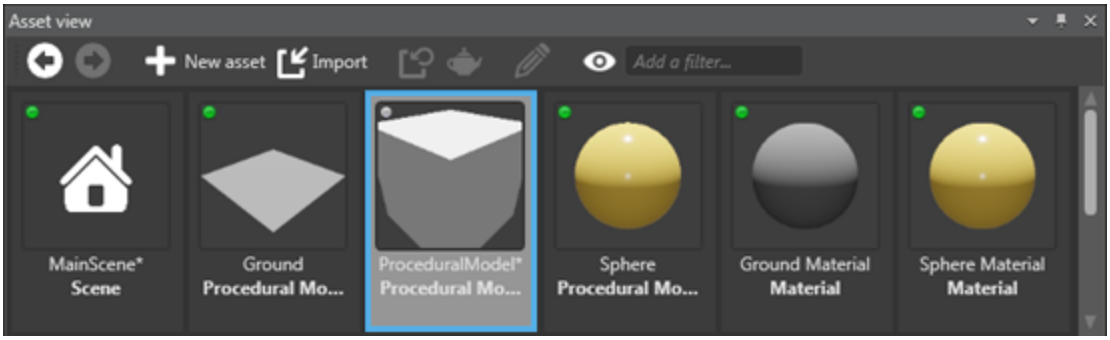
2. Select the type of asset you want to create.



   Game Studio displays a list of asset templates. These are assets configured for a specific use.

3. Select the right template for your asset.

   Game Studio adds the asset to the Asset View:



> **NOTE**
>
> Some assets, such as textures, require a resource file. When you add these assets, Game Studio prompts you for a resource file.

Drag and drop resource files

You can drag compatible resource files directly into Game Studio to create assets from them. Game Studio is compatible with common file formats.

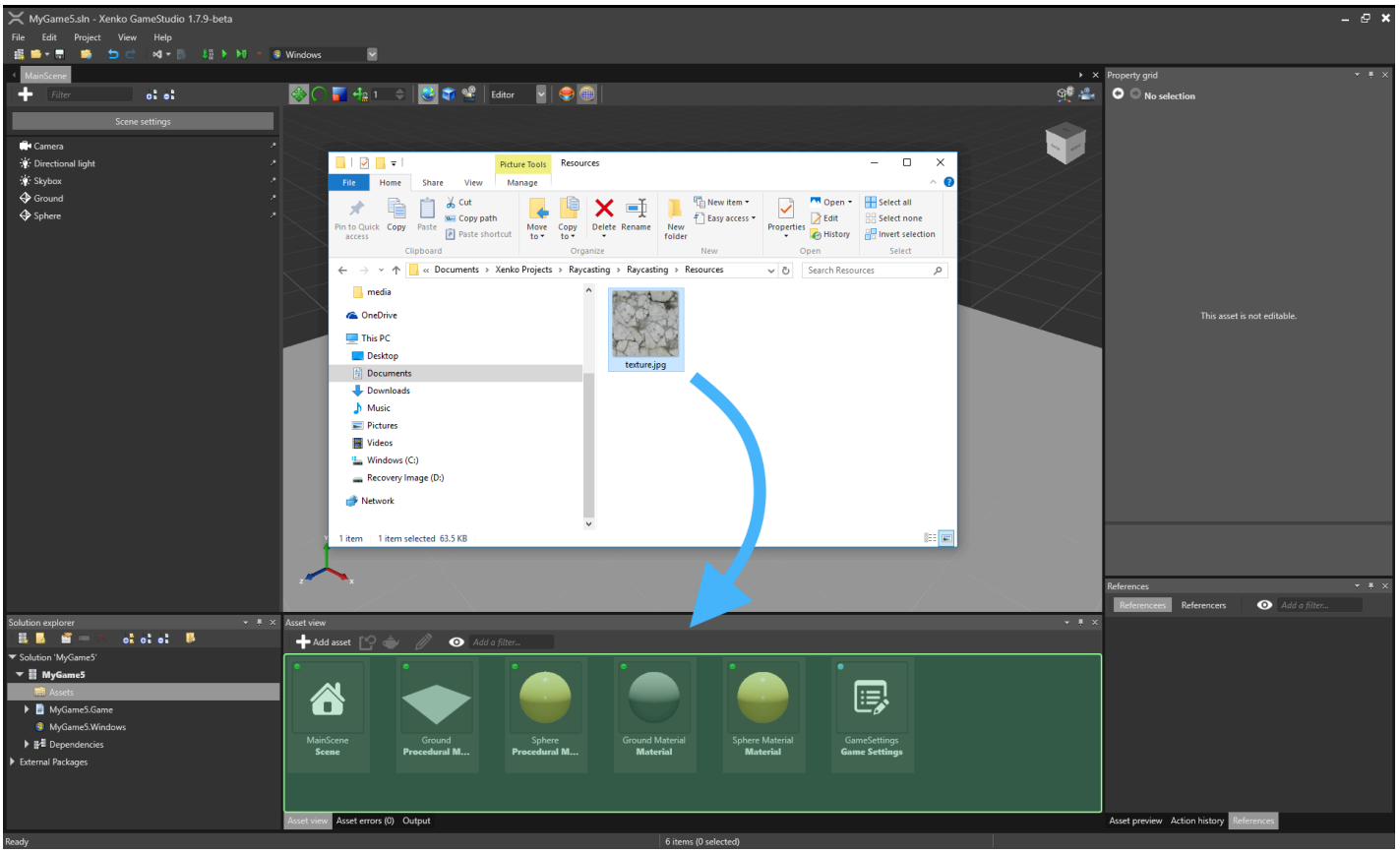
> **NOTE**
>
> - You can't use this method to create assets that don't use resource files (eg prefabs, materials, or scenes).

Asset type	Compatible resource file formats
Models, animations, skeletons	.dae, .3ds, obj, .blend, .x, .md2, .md3, .dxf, .fbx
Sprites, textures, skyboxes	.dds, .jpg, .jpeg, .png, .gif, .bmp, .tga, .psd, .tif, .tiff
Audio	.wav, .mp3, .ogg, .aac, .aiff, .flac, .m4a, .wma, .mpc

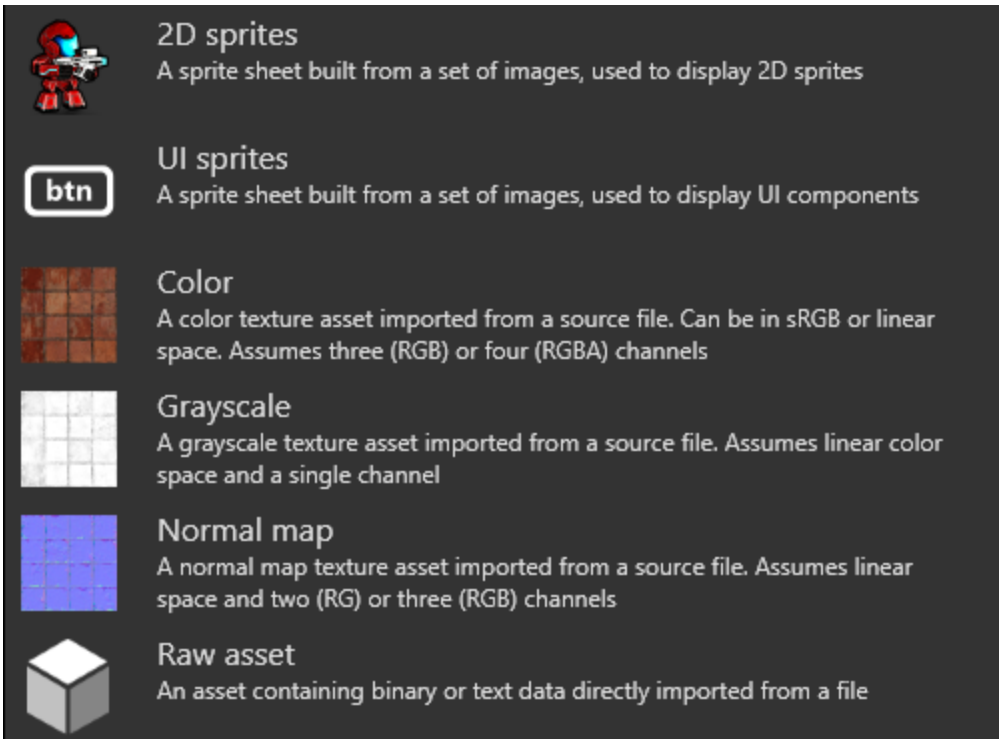
To create an asset by dragging and dropping a resource file:

1. (Optional) If it isn't there already, move the resource file you want to use in the **Resources** folder of your project. You don't have to do this, but it's good practice to keep resource files organized and makes projects easier to share. For more information, see Project structure.

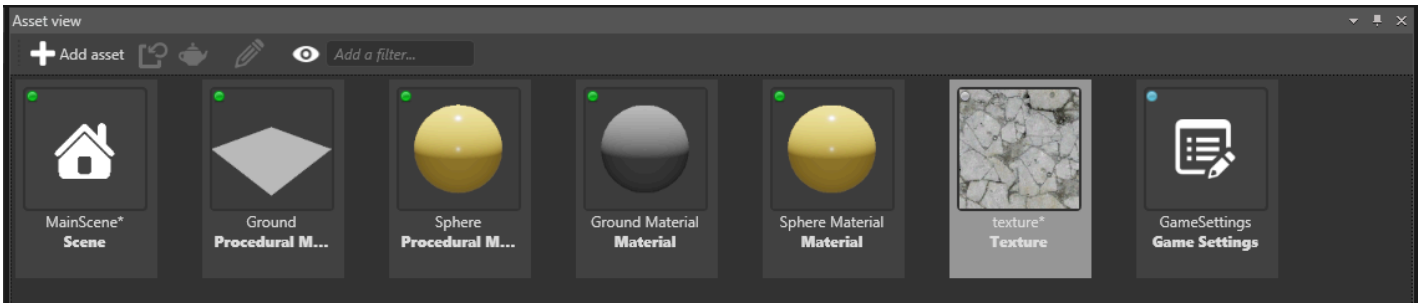
2. Drag the resource file from Explorer to the Asset View:



3. Select the kind of asset you want to create:



Game Studio adds the asset to the Asset View:



Game Studio automatically imports all dependencies in the resource files and creates corresponding assets. For example, you can add a model or animation resource file and Game Studio handles everything else.

> **TIP**
>
> You can drag multiple files simultaneously. If you drop multiple files of different types at the same time, Game Studio only adds only files that match your template selection. For example, if you add an image file and a sound file, then select the audio asset template, only the sound file is added.

See also

- Manage assets
- Use assets

Manage assets

Beginner

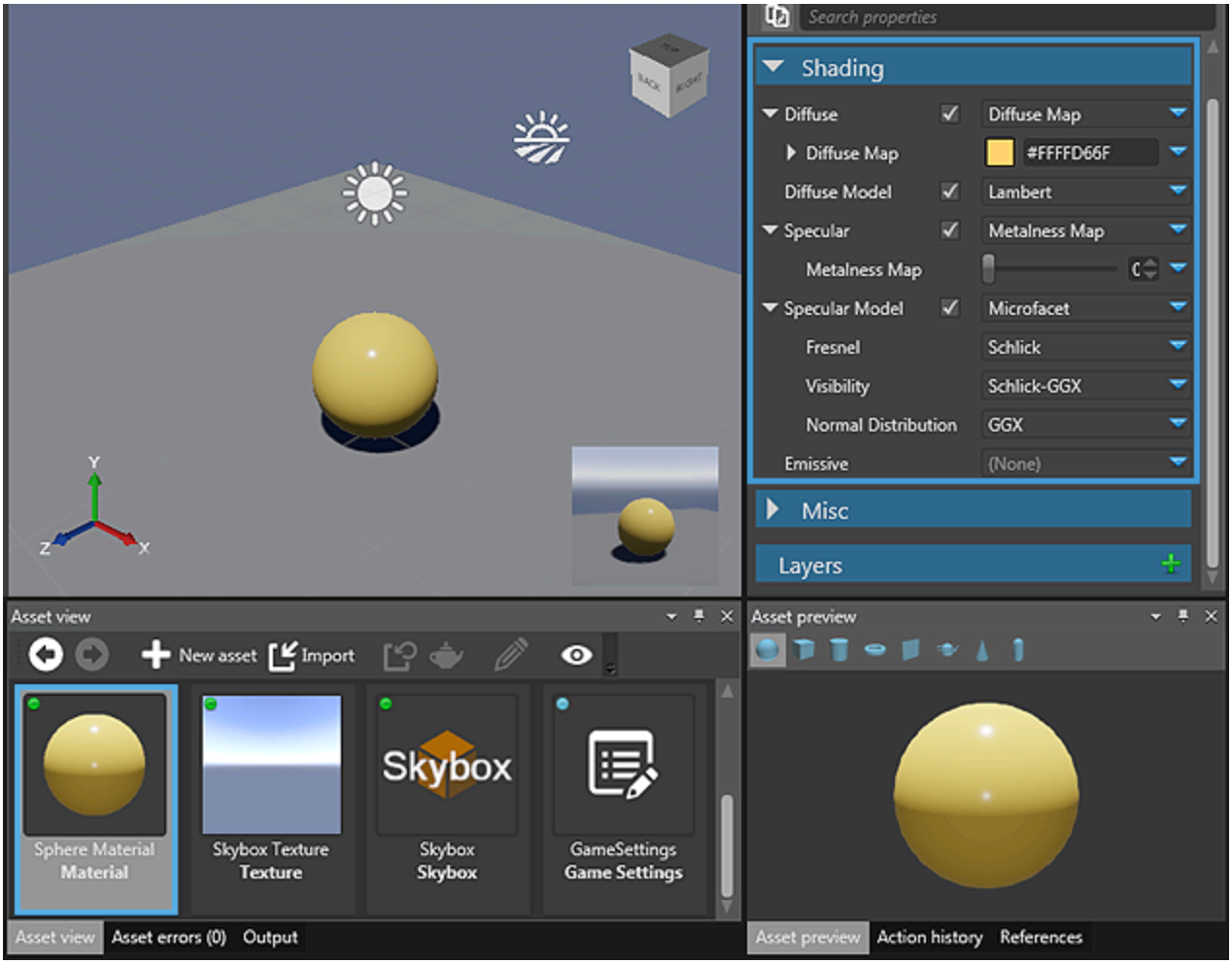
This page explains how to edit and manage your assets.

Edit assets in the Property Grid

You can edit most assets using the **Property Grid**. By default, this is in the top-right of Game Studio.

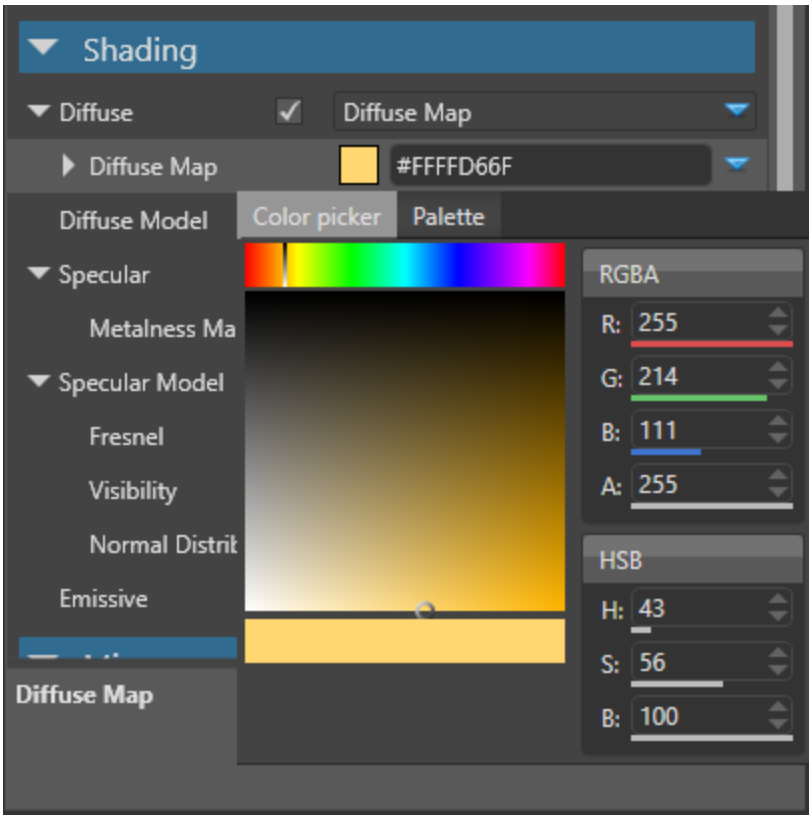
For example, to change the color of a material asset::

1. In the **Asset View** (in the bottom by default), select the material.

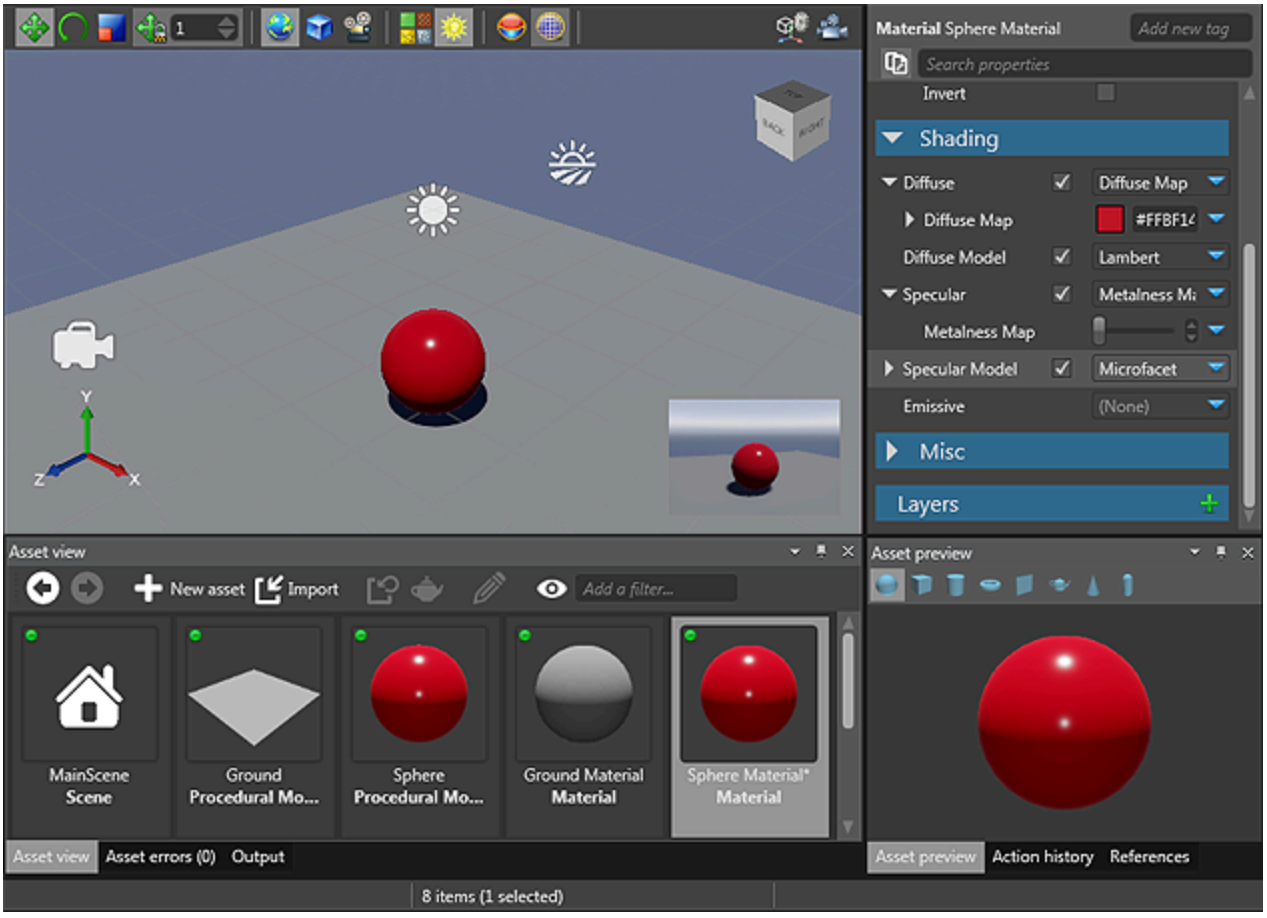


2. In the Property Grid, under **Shading > Diffuse**, next to **Diffuse Map**, click the **colored box**, which displays the asset color (yellow in this example).

   The color picker opens.

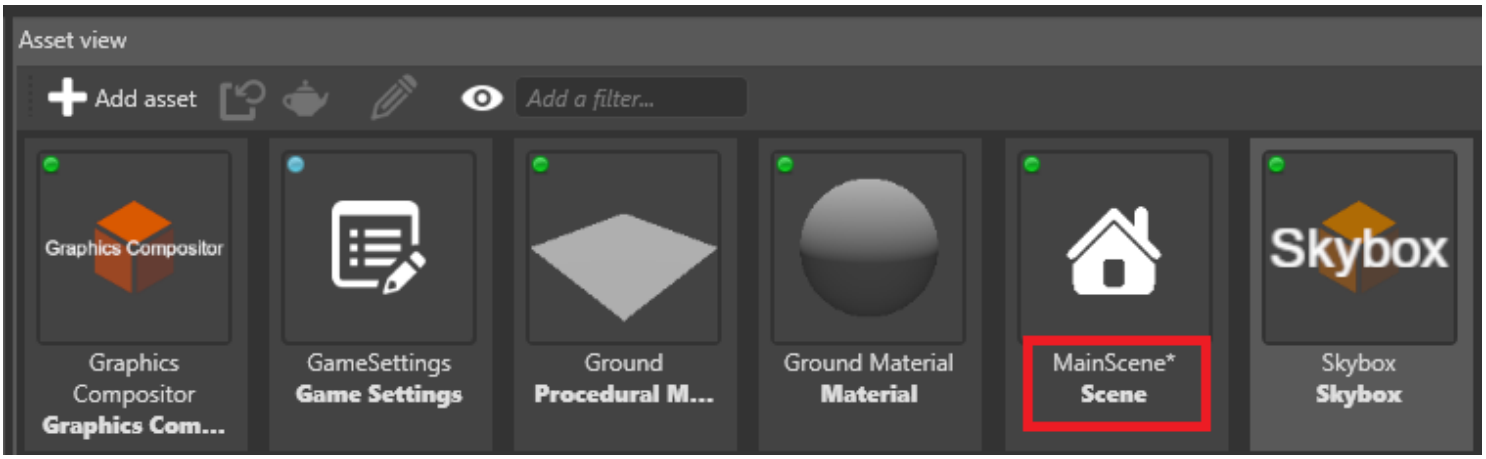


3. Select a new color for the asset.



The **Asset Preview** (bottom right by default) displays asset changes in real time.

The **Asset View** indicates assets with unsaved changes with asterisks (*).

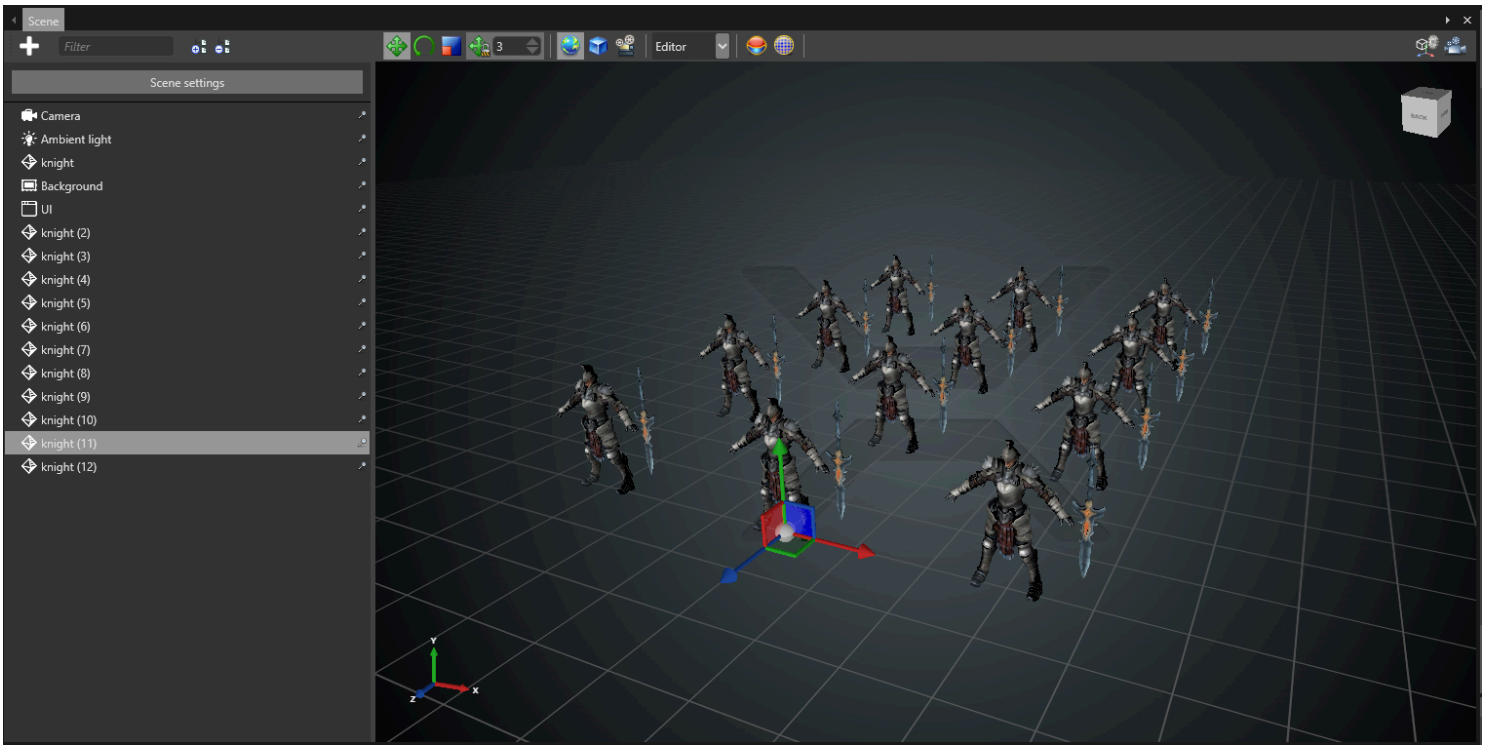


Edit assets using dedicated editors

Game Studio has dedicated editors for the following asset types:

- prefabs
- scenes
- sprite sheets
- UI pages
- UI libraries
- scripts

For example, you edit scenes in the **Scene Editor**.



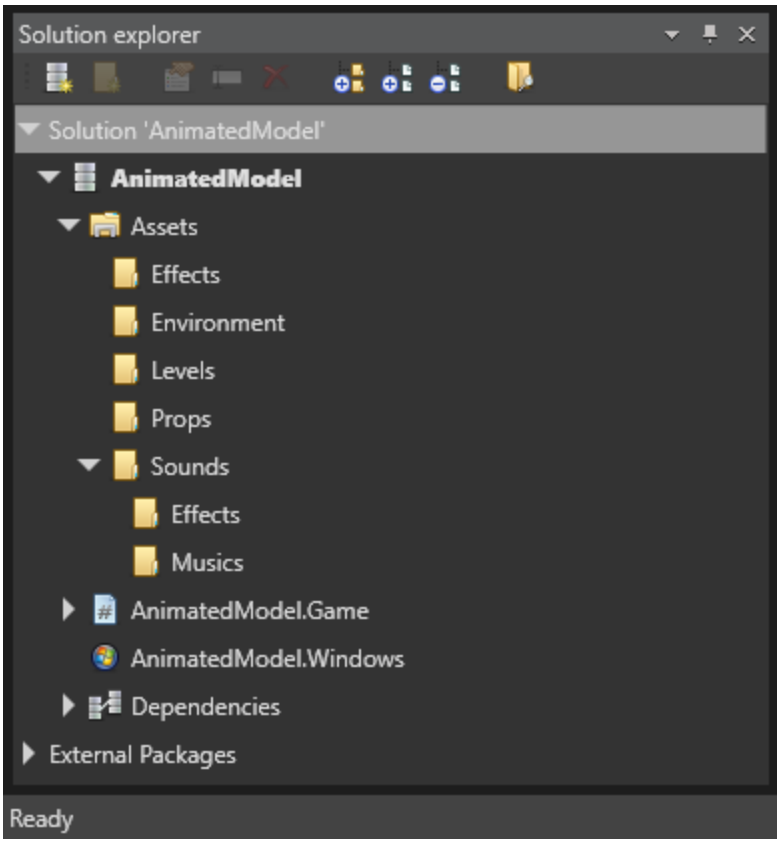
To open the dedicated editor for these types of asset:

- double-click the asset, or

- right-click the asset and select **Edit asset**, or
- select the asset and type **Ctrl + Enter**

Organize assets

We recommend you organize your assets into subfolders by type. This makes projects much easier to manage, especially as they become large.



Assets are contained in the **Assets** folder of your project package. You can see the project in the **Solution Explorer** (by default shown in the bottom left).

- To create a subfolder, right-click the parent folder and select **Create subfolder**.
- To move an asset, select one or more assets in the **Asset View** and drag and drop them to the folder.

> **NOTE**
>
> When you move an asset, Game Studio updates all references to other assets inside the asset.

TIP

To see the URL and type of an asset, move the mouse over the asset thumbnail.

Include assets in the build

By default, Stride doesn't include every asset when you build the game. This is because you might not need every asset at runtime — for example, if the asset is incomplete.

Stride only includes assets which:

- you've specifically marked for inclusion (**root assets**), or
- are **referenced** by a root asset

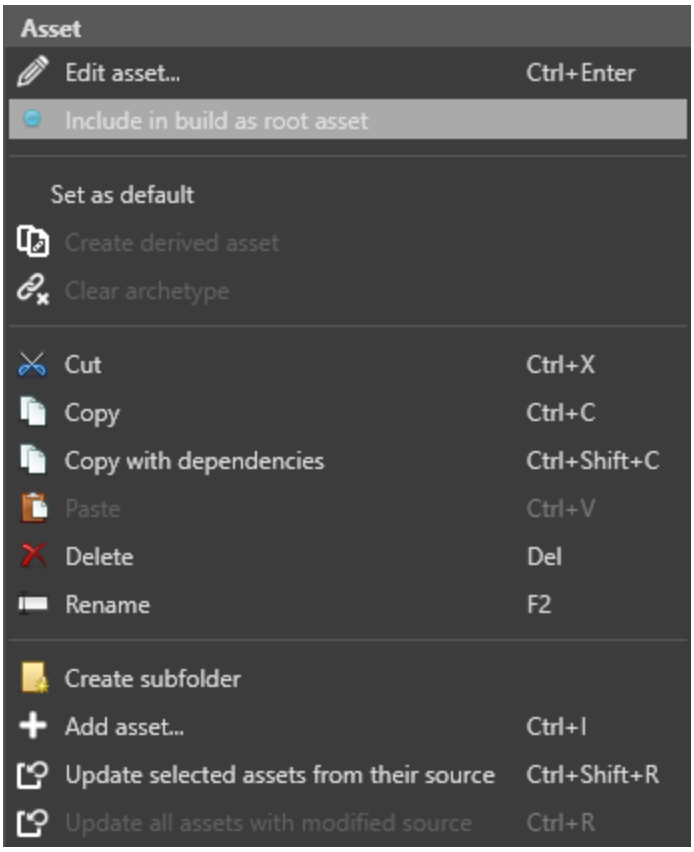
Game Studio indicates whether an asset is included with a colored icon in the top-left of the asset thumbnail.

Color	Status
Blue	The asset is a root asset and included in the build.
Green	The asset is referenced by a root asset and included in the build.

Color	Status
Gray 	The asset isn't included in the build.

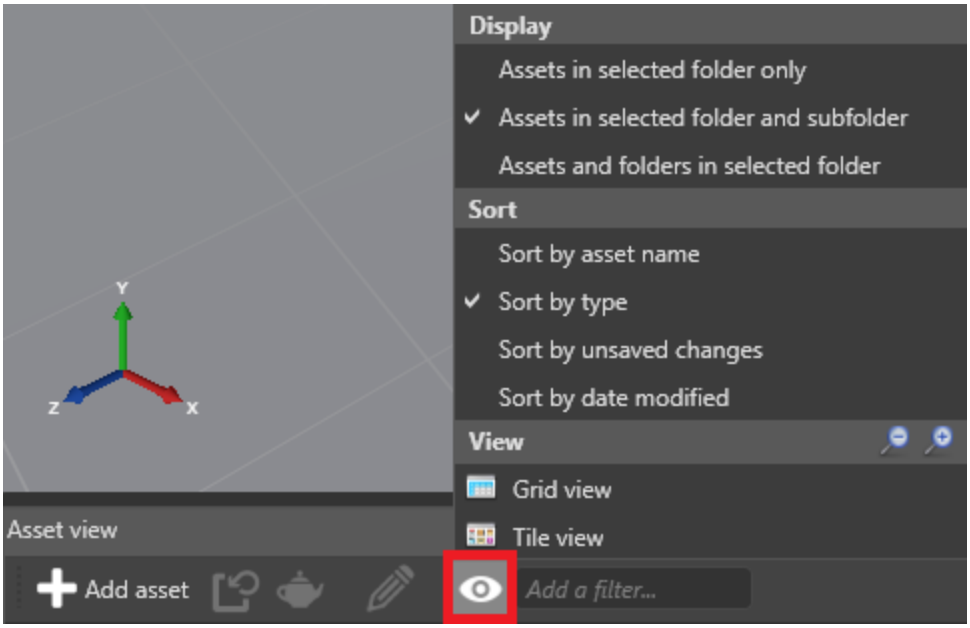
If you plan to load an asset at runtime using scripts, make it a root asset. To do this:

- click the **gray dot** in the top-left of the thumbnail, or
- right-click the asset and select **Include in build as root asset**



Asset View options

To change the Asset View options, click the eye icon in the Asset View toolbar.



You can:

- display assets in the selected folder only, the selected folder and subfolder
- sort assets by name, type, unsaved changes, and modification date
- switch between tile view (default) and grid view

Filter assets

When browsing assets in the **Asset View** (in the bottom by default), you can filter by name, tag, type, or a combination of all three.

The tag and name filters are "and" filters. For example, if you filter by *tag:level* and *name:knight*, the Asset View only displays assets with the tag "level" **and** the name "knight".

Type filters are "or" filters. For example, if you filter by *type:animation* and *type:texture*, the Asset View only displays assets that are animations **or** textures.

Add a filter

1. In the Asset View, type in the filter bar.

   Game Studio displays a list of matching filters (name, type, or tag).

2. To filter by name, press **Enter**.

   To filter by a tag or type, select **tag** or **type** filters in the drop-down list.

   Game Studio applies the filter and shows matching assets in the Asset View.

You can add multiple filters. Name filters are green, tag filters are blue, and type filters are orange.

Toggle filters on and off

To toggle a filter on and off without removing it, click it. Disabled filters have darker colors.

Remove a filter

To remove a filter, click the X icon in the filter tag.

See also

- Create assets
- Manage assets
- Use assets

Use assets

Beginner

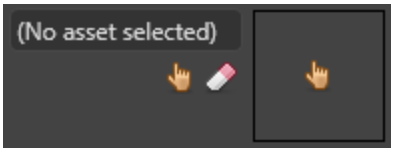
There are four ways to use assets:

- reference them in entity components
- reference them in other assets
- load them from code as content
- load them from code as content using `UrlReference`

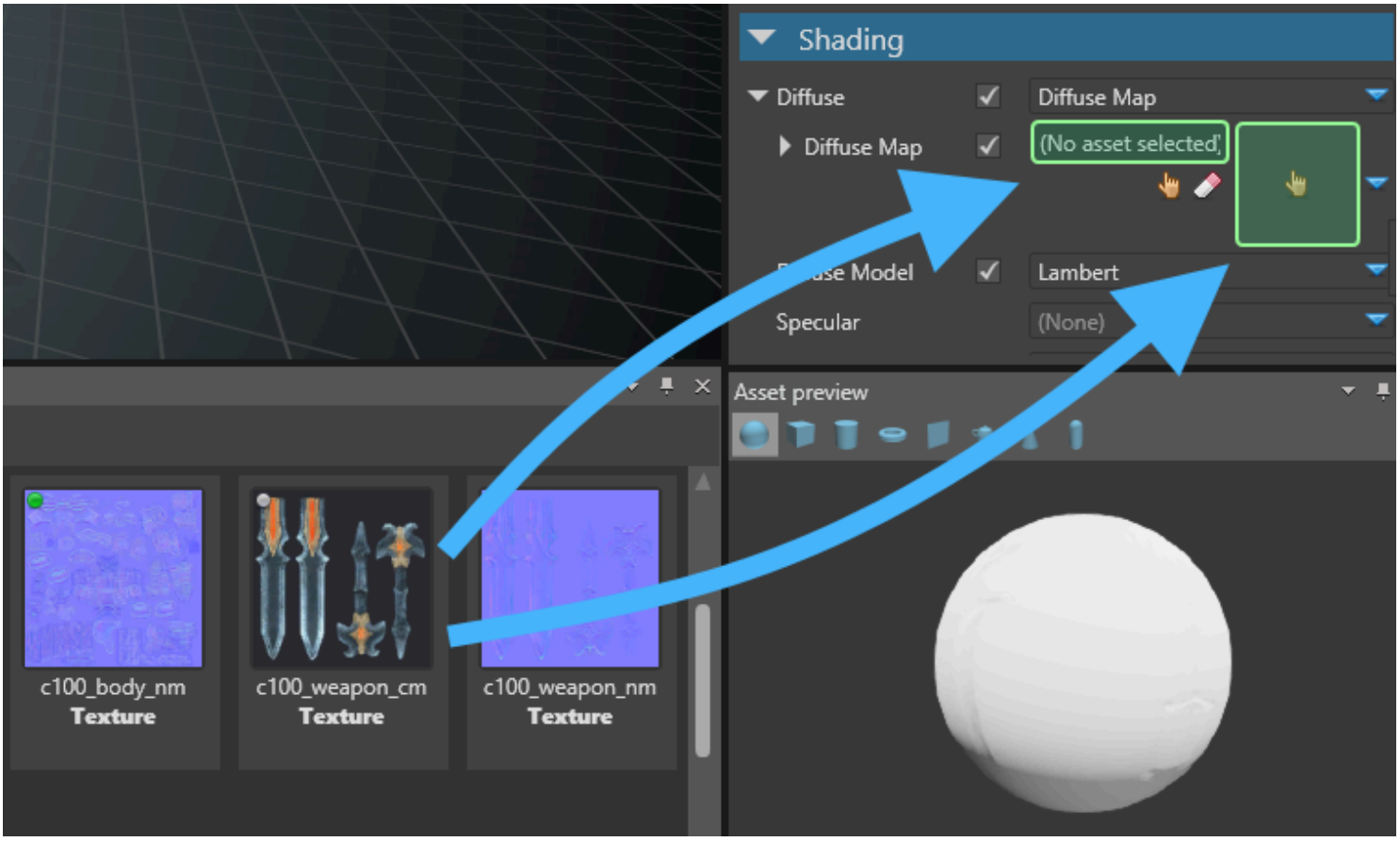
Reference assets in components

Many kinds of component use assets. For example, model components use model assets.

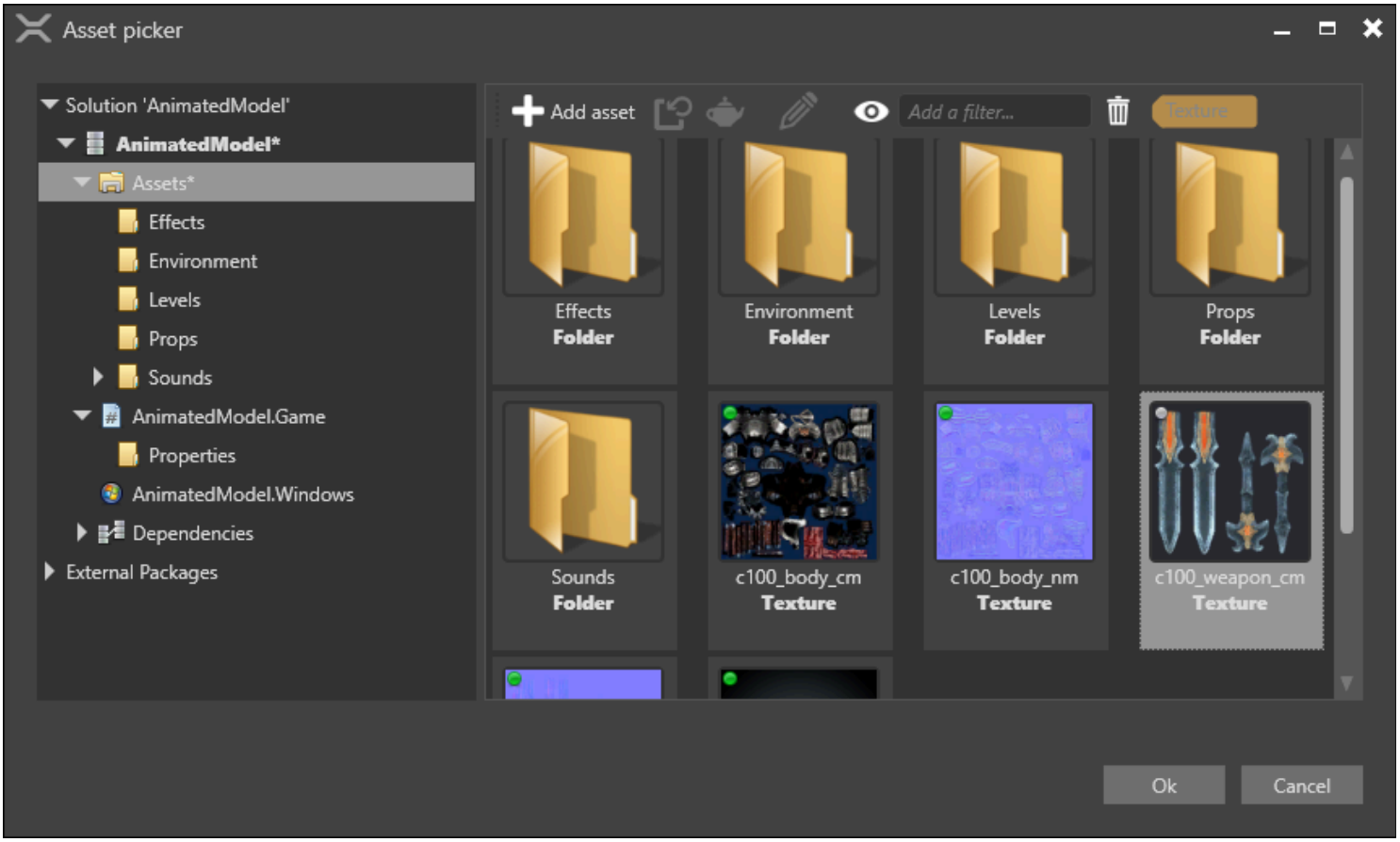
Components that use assets have **asset docks** in the **property grid**.



To add an asset to an entity component, drag the asset to the asset dock in the component properties (in the **property grid**). You can drop assets in the text field or the empty thumbnail.



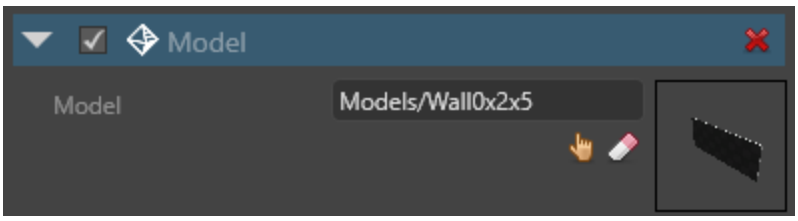
Alternatively, click (**Select an asset**).



The **Select an asset** window opens.

> **NOTE**
>
> The **Select an asset** window only displays assets of types expected by the component. For example, if the component is an audio listener, the window only displays audio assets.

After you add an asset to a component, the asset dock displays its name and a thumbnail image.



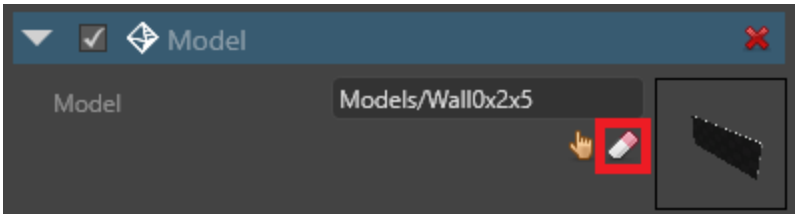
Reference assets in other assets

Assets can reference other assets. For example, a model asset might use material assets.

You can add asset references to assets the same way you add them to entity components (see above).

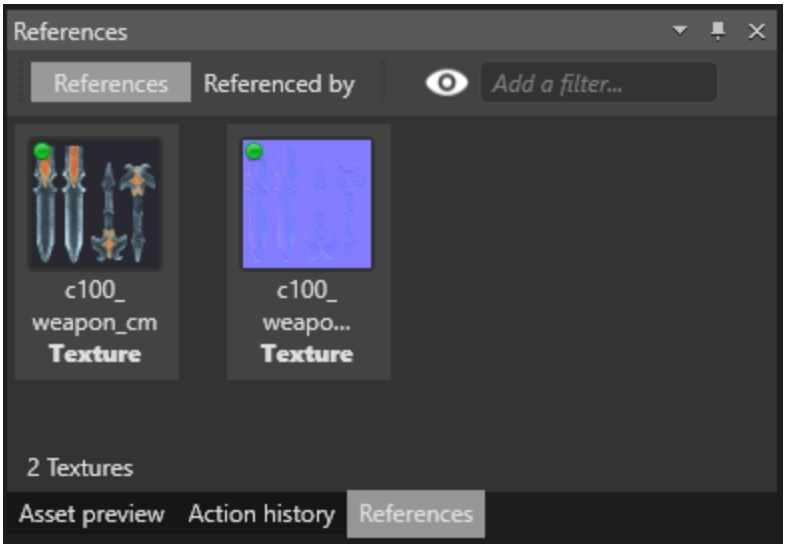
Clear a reference

To clear a reference to an asset, in the **asset dock**, click (**Clear reference**).



Examine references

You can see the references in a selected asset in the **References** tab. By default, this is in the bottom right of Game Studio.



- The **References** tab displays the assets referenced by the selected asset.
- The **Referenced by** tab displays the assets that reference the selected asset.

> **TIP**
>
> If you can't see the References tab, make sure it's displayed under **View > References**.

Load assets from code

When loading in assets at runtime we speak of "Content" rather than assets. The loaded content refers to the asset and can then be used in your script.

```
// Load a model (replace URL with valid URL)
var model = Content.Load<Model>("AssetFolder/MyModel");
```

```
// Create a new entity to add to the scene
Entity entity = new Entity(position, "Entity Added by Script") { new ModelComponent { Model
= model } };

// Add a new entity to the scene
SceneSystem.SceneInstance.RootScene.Entities.Add(entity);
```

> **TIP**
>
> To find the asset URL, in Game Studio, move the mouse over the asset. Game Studio displays the asset URL in a tooltip. URLs typically have the format *AssetFolder/AssetName*.

> **WARNING**
>
> When loading assets from scripts, make sure you:
>
> - include the asset in the build as described in Manage assets
> - make sure you add the script as a component to an entity in the scene

Unload unneeded assets

When loading content from code, you should unload content when you don't need them any more. If you don't, content stays in memory, wasting GPU.

To unload an asset, use `Content.Unload(myAsset)`.

Load assets from code using UrlReference

`UrlReference` allows you to reference assets in your scripts the same way you would with normal assets but they are loaded dynamically in code. Referencing an asset with a `UrlReference` causes the asset to be included in the build.

You can reference assets in your scripts using properties/fields of type `UrlReference` or `UrlReference<T>`:

- `UrlReference` can be used to reference any asset. This is most useful for the "Raw asset".
- `UrlReference<T>` can be used to specify the desired type. i.e. `UrlReference<Scene>`. This gives Game Studio a hint about what type of asset this `UrlReference` can be used for.

Examples

Loading a Scene

Using `UrlReference<Scene>` to load the next scene.

```
using System.Threading.Tasks;
//Include the Stride.Core.Serialization namespace to use UrlReference
using Stride.Core.Serialization;
using Stride.Engine;

namespace Examples
{
    public class UrlReferenceExample : AsyncScript
    {
        public UrlReference<Scene> NextSceneUrl { get; set; }

        public override async Task Execute()
        {
            //...
        }

        private async Task LoadNextScene()
        {
            //Dynamically load next scene asynchronously
            var nextScene = await Content.LoadAsync(NextSceneUrl);
            SceneSystem.SceneInstance.RootScene = nextScene;
        }
    }
}
```

Load data from a Raw asset JSON file

Use a Raw asset to store data in a JSON file and load using Newtonsoft.Json. To use `Newtonsoft.Json` you also need to add the `Newtonsoft.Json` NuGet package to the project.

```
//Include the Newtonsoft.Json namespace.
using Newtonsoft.Json;
using System.IO;
using System.Threading.Tasks;
//Include the Stride.Core.Serialization namespace to use UrlReference
using Stride.Core.Serialization;
using Stride.Engine;

namespace Examples
{
    public class UrlReferenceExample : AsyncScript
    {
        public UrlReference RawAssetUrl { get; set; }
```

```
        public override async Task Execute()
        {
            //...
        }

        private async Task<MyDataClass> LoadMyData()
        {
            //Open a StreamReader to read the content
            using (var stream = Content.OpenAsStream(RawAssetUrl))
            using (var streamReader = new StreamReader(stream))
            {
                //read the raw asset content
                string json = await streamReader.ReadToEndAsync();
                //Deserialize the JSON to your custom MyDataClass Type.
                return JsonConvert.DeserializeObject<MyDataClass>(json);
            }
        }
    }
}
```

See also

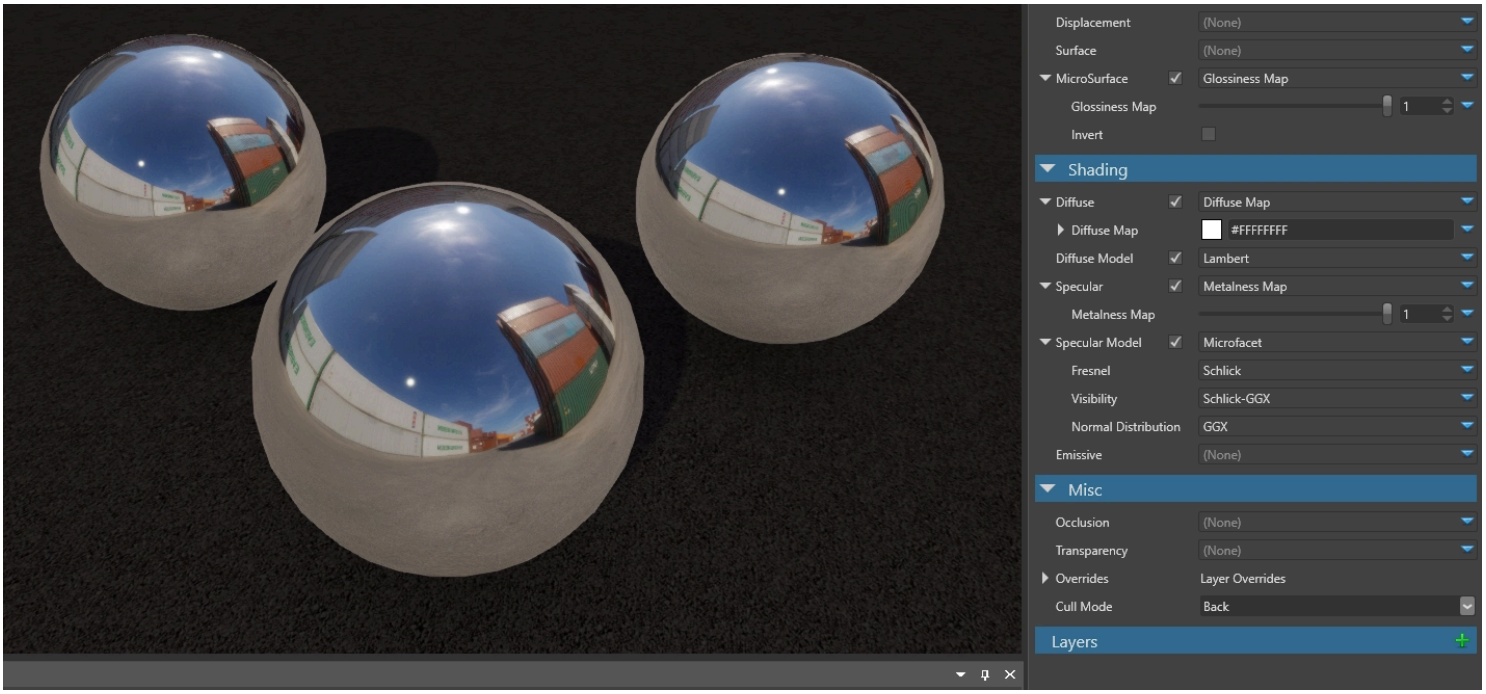
- Create assets
- Manage assets

Archetypes

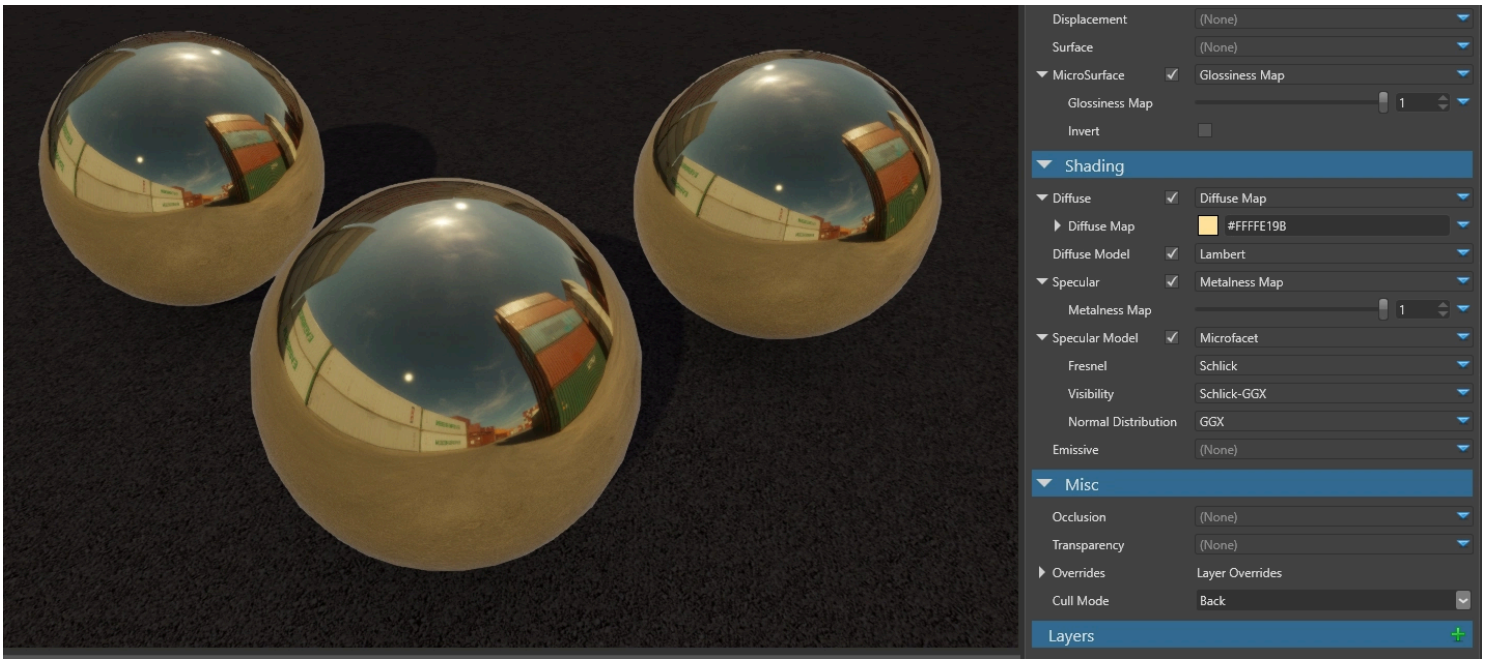
Intermediate Designer

An **archetype** is a master asset that controls the properties of assets you **derive** from it. Derived assets are useful when you want to create a "remixed" version of an asset.

For example, imagine we have three sphere entities that share a material asset named *Metal*. The Metal asset has properties including color, gloss, and so on.

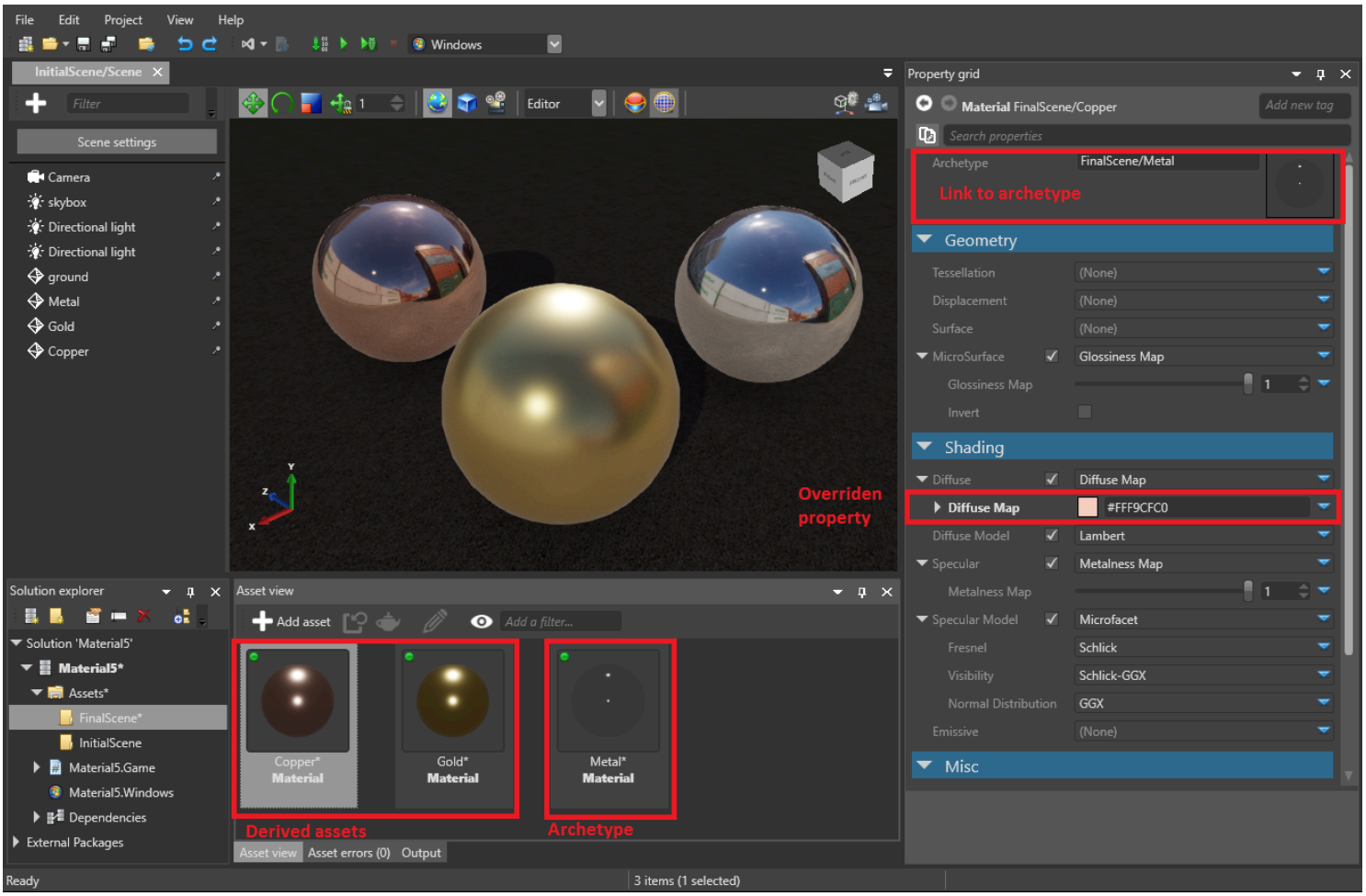


If we change a property in the **Metal** asset, it applies to all three spheres. So, for example, if we change the color property, all three spheres change color.



Now imagine we want to change the color of only *one* sphere, but keep its other properties the same. We could duplicate the material asset, change its color, and then apply the new asset to only one sphere. But if we later want to change a different property across *all* the spheres, we have to modify both assets. This is time-consuming and leaves room for mistakes.

The better approach is to derive a new asset from the archetype. The derived asset inherits properties from the archetype and lets you override individual properties where you need them. For example, we can derive the sphere's material asset and override its color. Then, if we change the gloss of the archetype, the gloss of all three spheres changes.

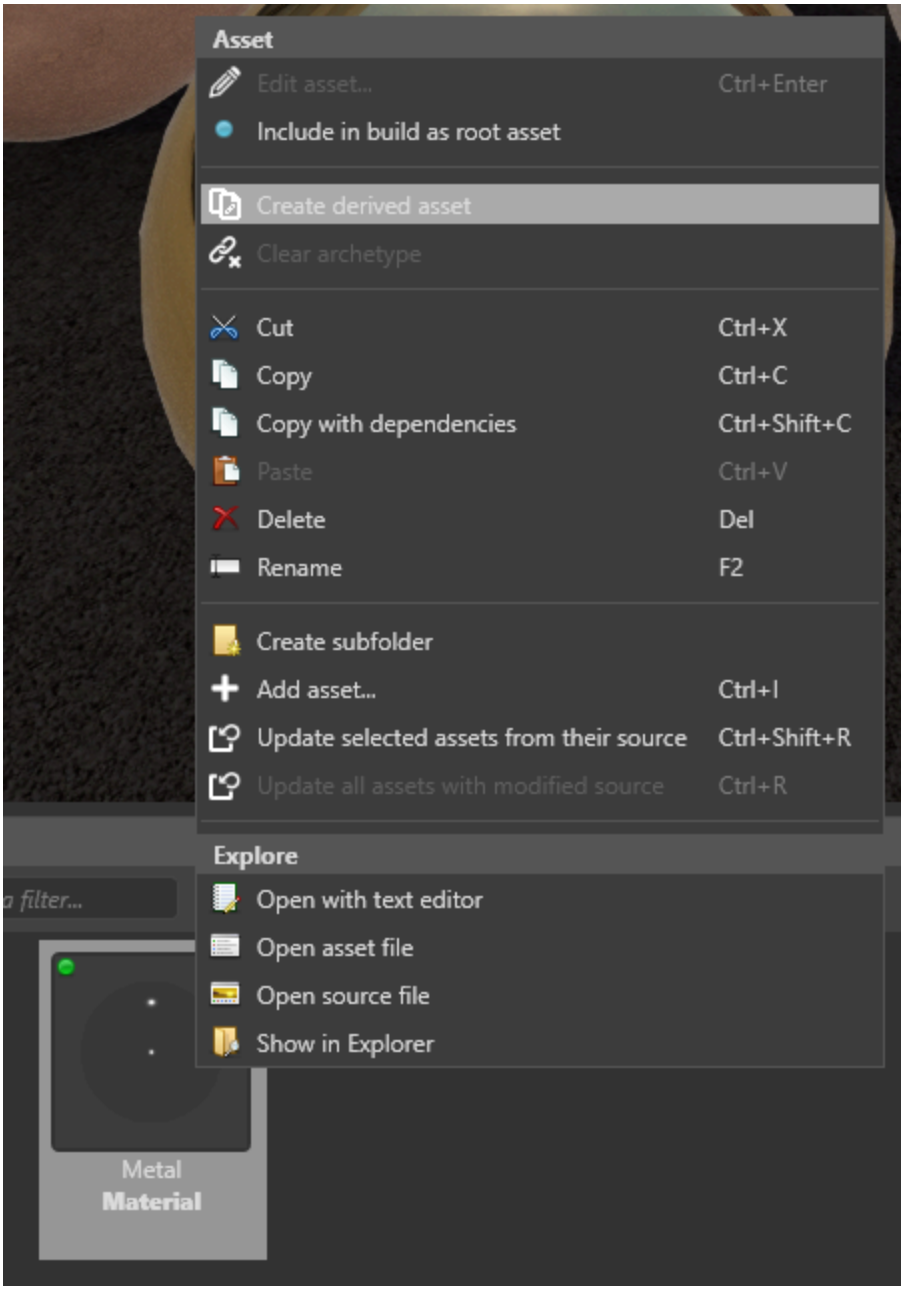


You can derive an asset from an archetype, then in turn derive another asset from that derived asset. This way you can create different layers of assets to keep your project organized:

```
Archetype
    Derived asset
        Derived asset
```

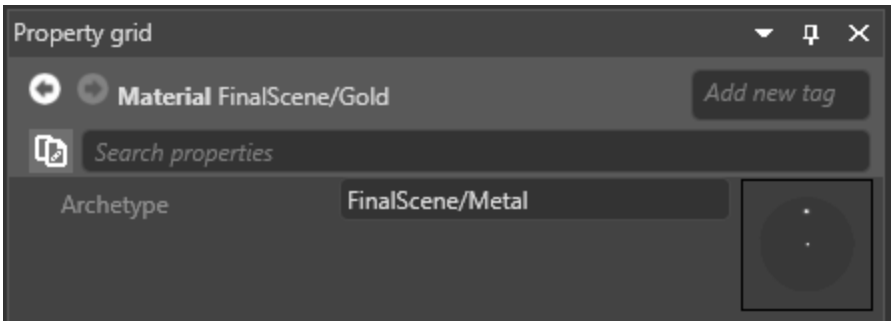
Derive an asset from an archetype

In the **Asset View**, right-click the asset you want to derive an asset from and select **Create derived asset**:

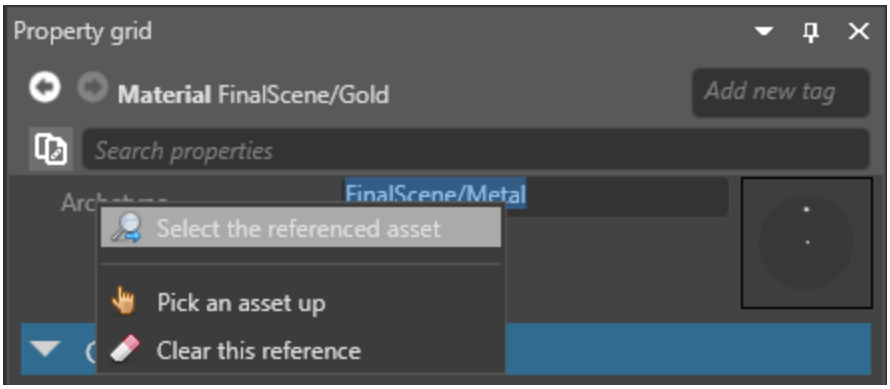


Game Studio adds a new **derived asset** to the project. This asset derives its properties from the **archetype** asset.

The derived asset properties display the archetype asset under **Archetype**:



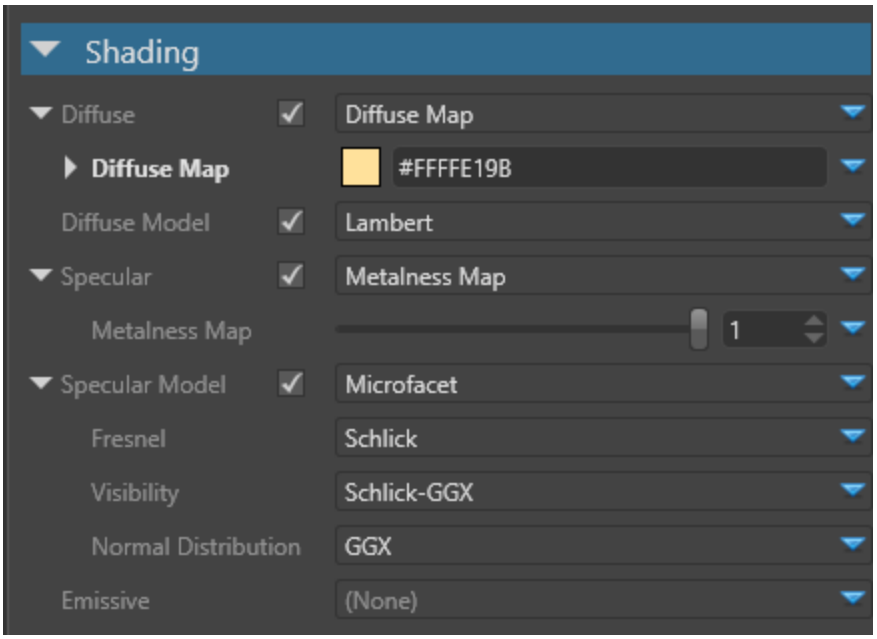
You can right-click the archetype asset in the Property Grid and select **Select the referenced asset** to quickly select the archetype asset:



Overridden properties

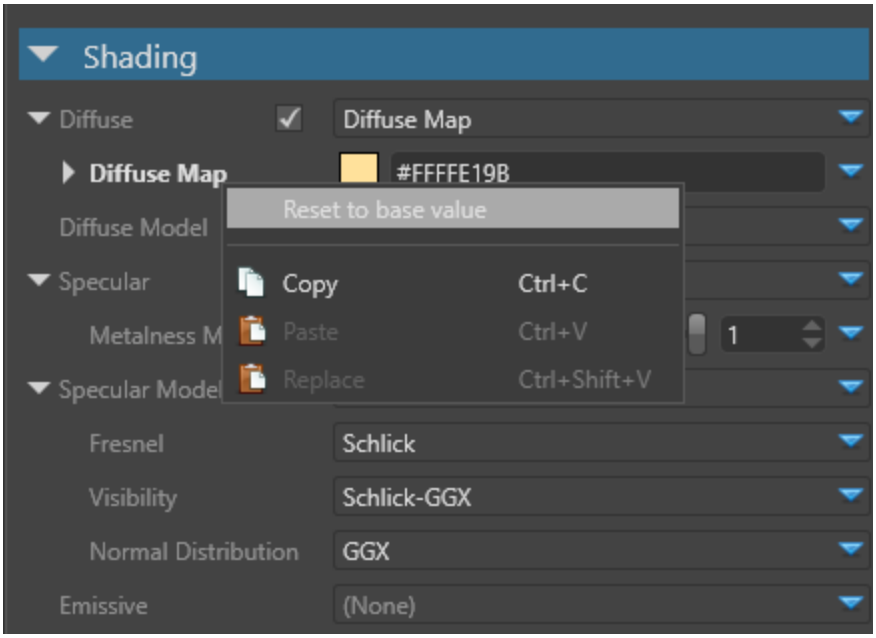
The **Property Grid** shows which properties of the derived asset differ from the archetype. **Overridden** and **unique** properties are **white**, and **inherited** (identical) properties are **gray**.

In this screenshot, the **Diffuse Map** property is overridden. The other properties are inherited:



Reset a property to archetype value

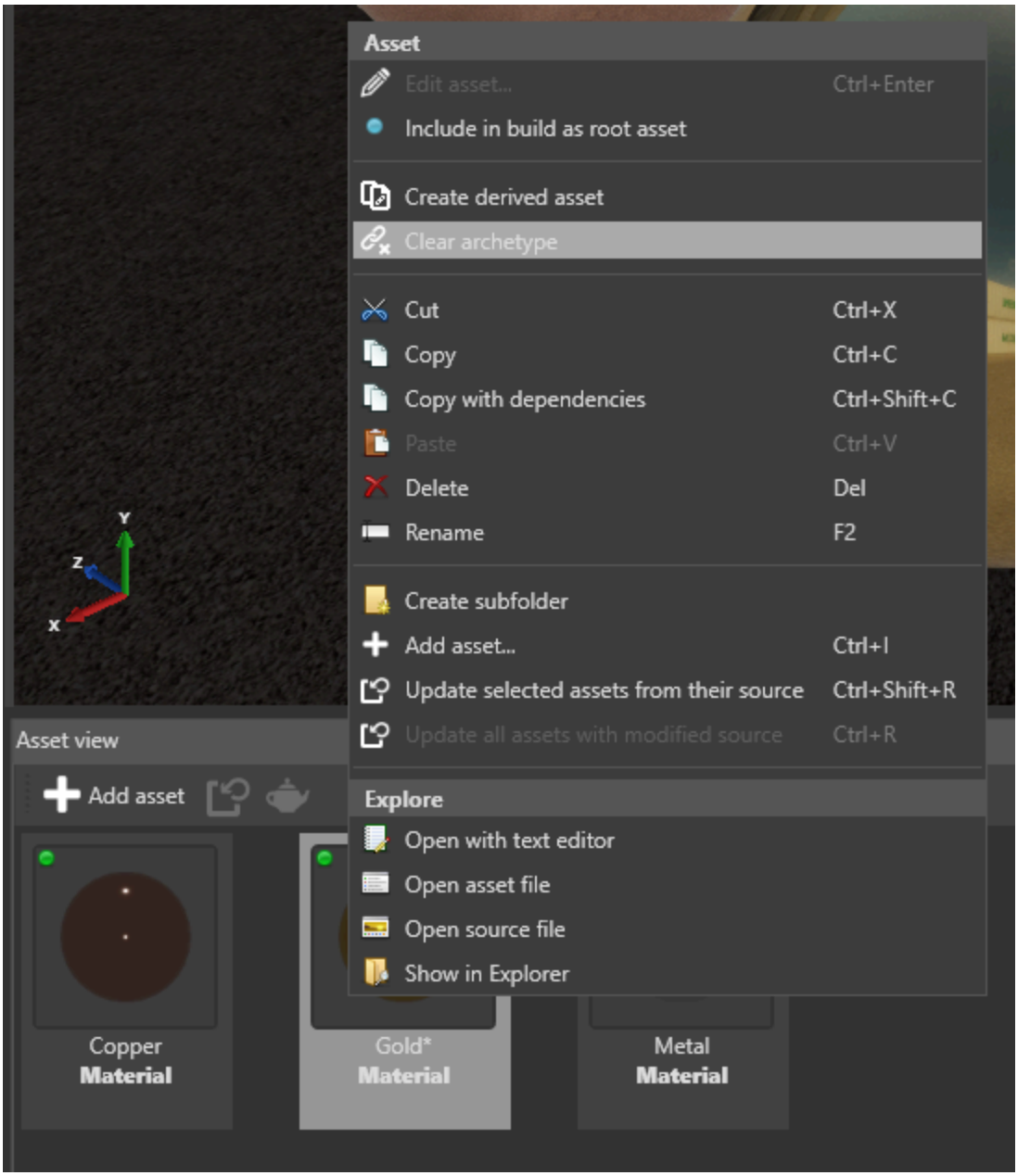
You can reset overridden or unique properties of a derived asset to the values in the archetype. To do this, right-click the overridden property and select **Reset to base value**.



Clear an archetype

You can remove the link between the archetype and the derived asset. This means the derived asset no longer inherits changes to the archetype; it becomes a completely independent.

To do this, in the **Asset View**, right-click the derived asset and select **Clear archetype**.



See also

- Assets
- Prefabs

Prefabs

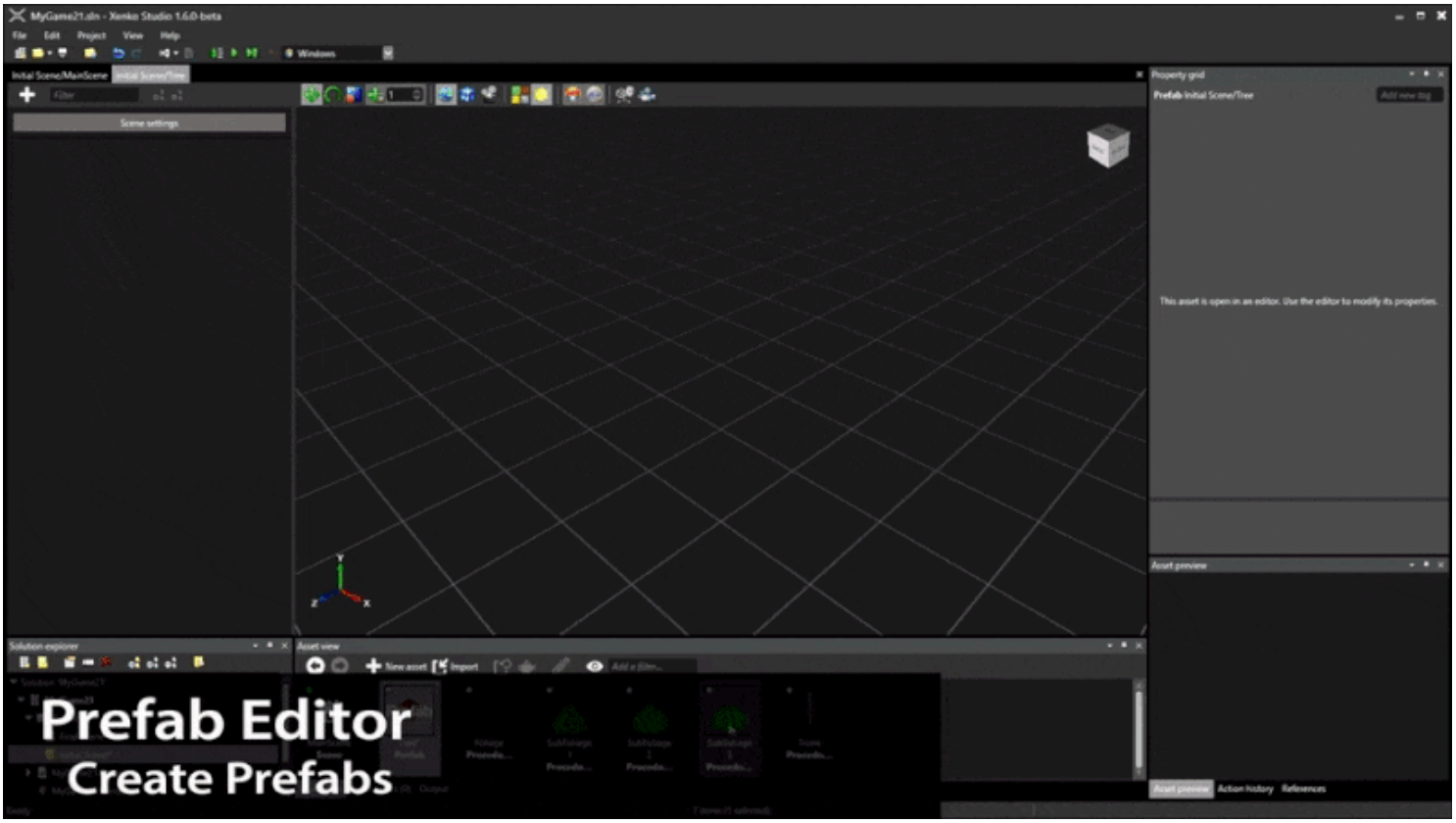
Beginner Programmer Designer

A **prefab** is a "master" version of an object that you can reuse wherever you need. When you change the prefab, every instance of the prefab changes too.

For example, imagine we make a simple tree object by assembling several entities. The entities contain components such as models, materials, physics colliders, and so on, which in turn reference assets.

Now imagine we want to place several trees around the scene. We could simply duplicate the tree, but if we want to modify it later, we have to edit each one individually. This is time-consuming and leaves room for mistakes.

The better approach is to make the a tree prefab. Then we can place as many trees as we like, and when we modify the prefab, every tree is instantly updated to match. This saves lots of time.



The most common use for prefabs is to create a small piece of your scene — like a car, NPC, or item of furniture — and duplicate it as many times as you need. When you need to modify it — for example, if you want to change its model — you can change it in one place and apply the change everywhere at once.

You can make any entity or combination of entities of a prefab; they can be as simple or as complex as you need. Prefabs can even contain other prefabs (known as nested prefabs).

You can override specific properties in each prefab instance.

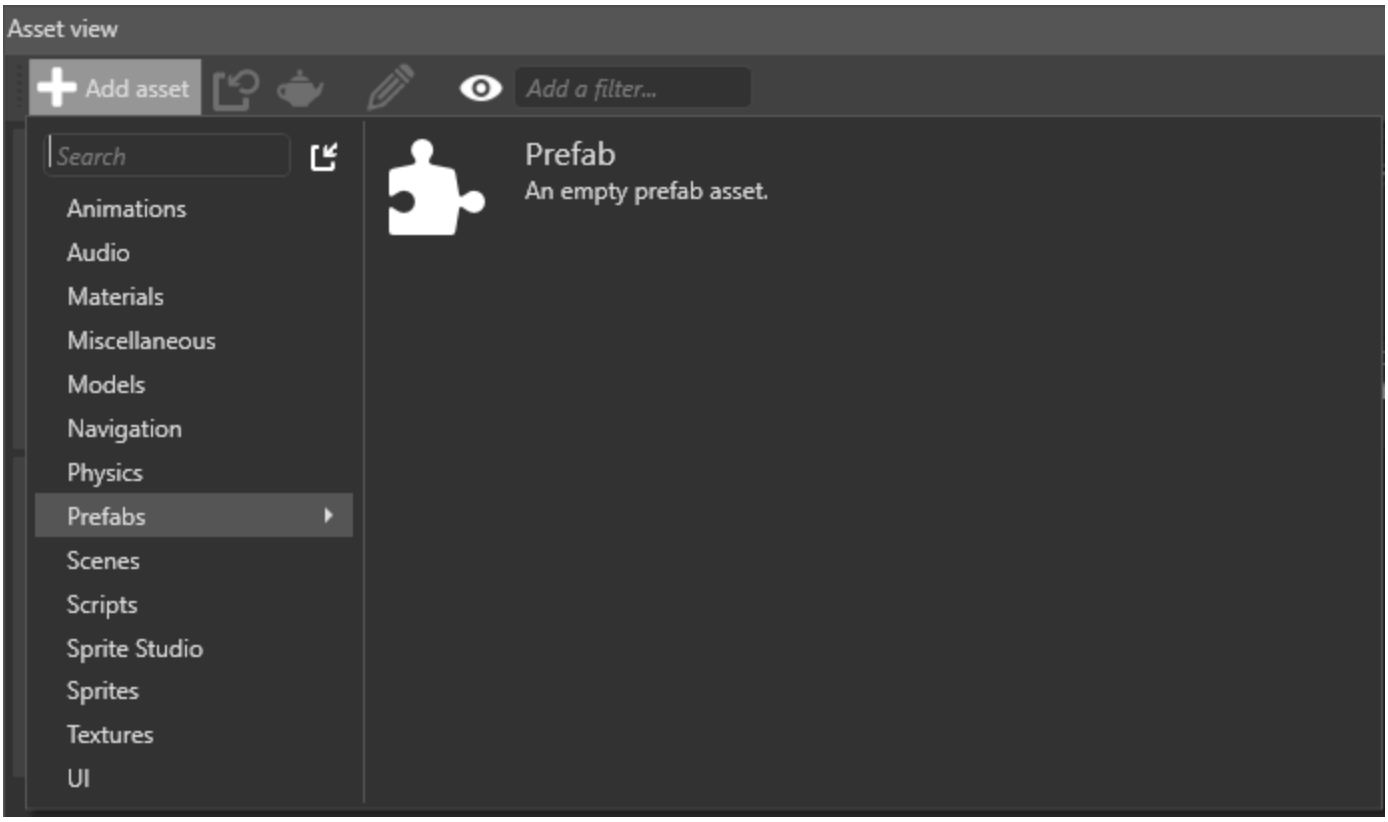
See also

- Create a prefab
- Use prefabs
- Edit prefabs
- Nested prefabs
- Override prefab properties
- Prefab models
- Archetypes

Create a prefab

Beginner Designer

In the **Asset View**, click **Add asset** and select **Prefabs** > **Prefab**.



Game Studio creates an empty prefab asset with the default name *Prefab*. Double-click the asset to open the **Prefab Editor** and add entities.

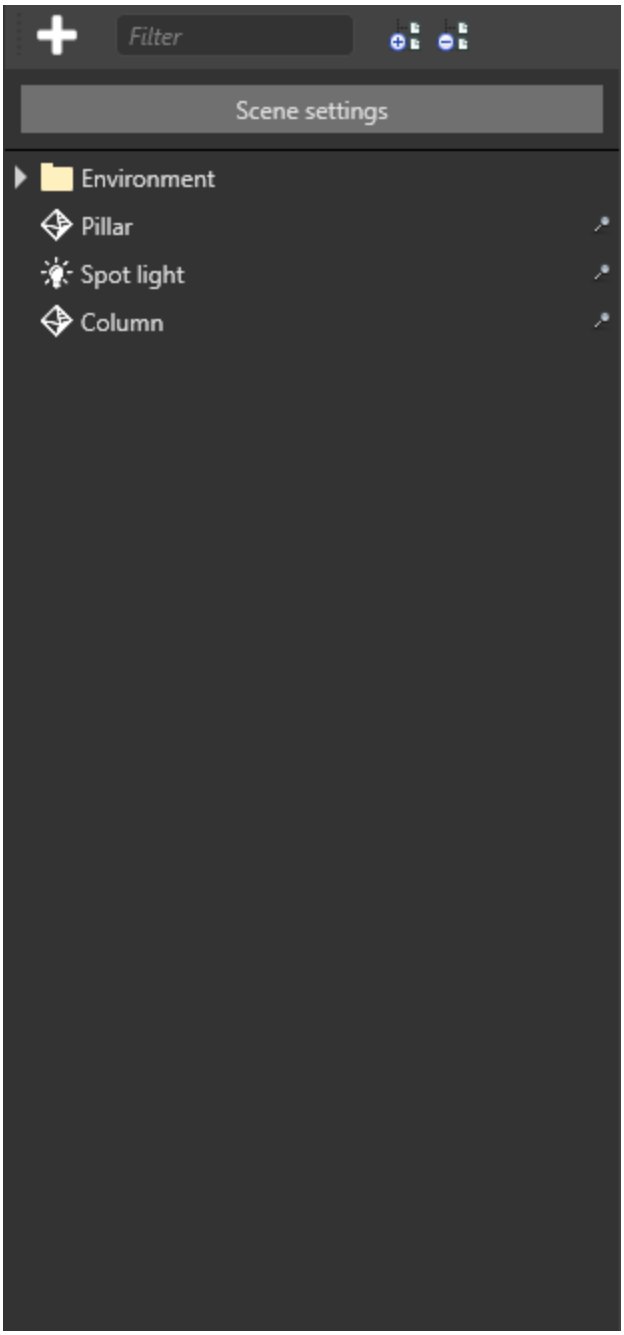
Create a prefab from an entity

You can also create a prefab from an existing entity or entities.

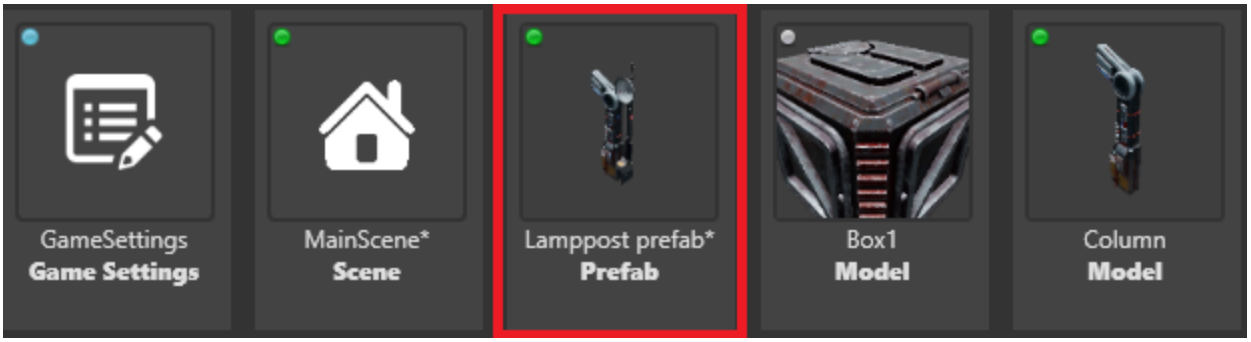
1. In the **Scene Editor**, select the entity or entities you want to create a prefab from.

> **TIP**
>
> Hold Ctrl to select multiple items.

2. Right-click the selection and select **Create prefab from selection**:



Game Studio creates a prefab asset from the entity or entities you selected. You can access the new prefab from the **Asset View**.



> **NOTE**
>
> After you create a prefab from a selection, the original selection itself **becomes a prefab**.

Create a prefab from an existing modified prefab

You can create new prefabs from modified prefabs. For example, you can instantiate a prefab, override one of its properties, then use this modified prefab instance to create a new prefab.

See also

- Prefab index
- Use prefabs
- Edit prefabs
- Nested prefabs
- Override prefab properties
- Prefab models

Use prefabs

Intermediate Programmer Designer

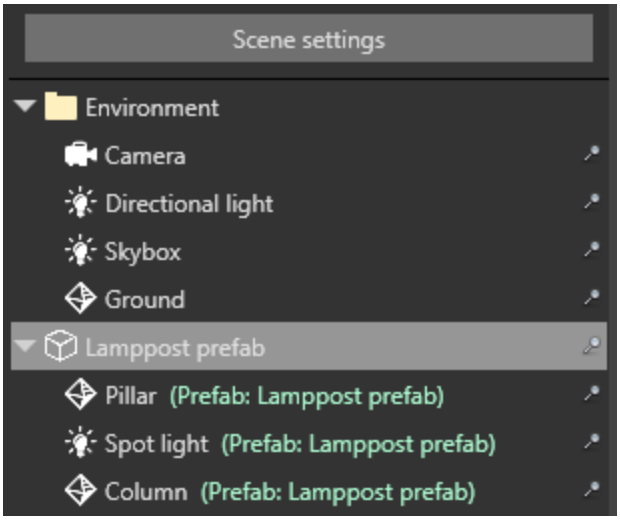
To instantiate a prefab, drag and drop it from the **Asset View** to the scene.

You can re-arrange entities in the prefab instance just like you do with other entities:

- create child and parent entities
- drag entities to add them to the prefab instance
- drag entities from the prefab instance to make them independent entities

Manage prefab parent entities

By default, Game Studio creates an empty parent entity with the prefab's entities as its children. The Entity Tree displays the prefab parent name in green next to the child entities.



This is useful because you can manage the prefab entities as a group and maintain their relative positions. For example, imagine you have a car prefab assembled from several entities (a body, seats, four wheels, etc). You want its component entities to stay grouped together as you move the car around the scene. You can do this by moving the prefab parent entity.

If you don't want to create a parent entity with the prefab, hold **Alt** when you drop the prefab into the scene. This is useful if you don't care about the relative positions of the prefab's entities and don't need to move them together as a group. For example, imagine you have a prefab composed of several crate entities arranged in a random fashion. It's not important that the crates maintain their relative position after you place them; in fact, several identical stacks of "randomly" arranged crates looks artificial.

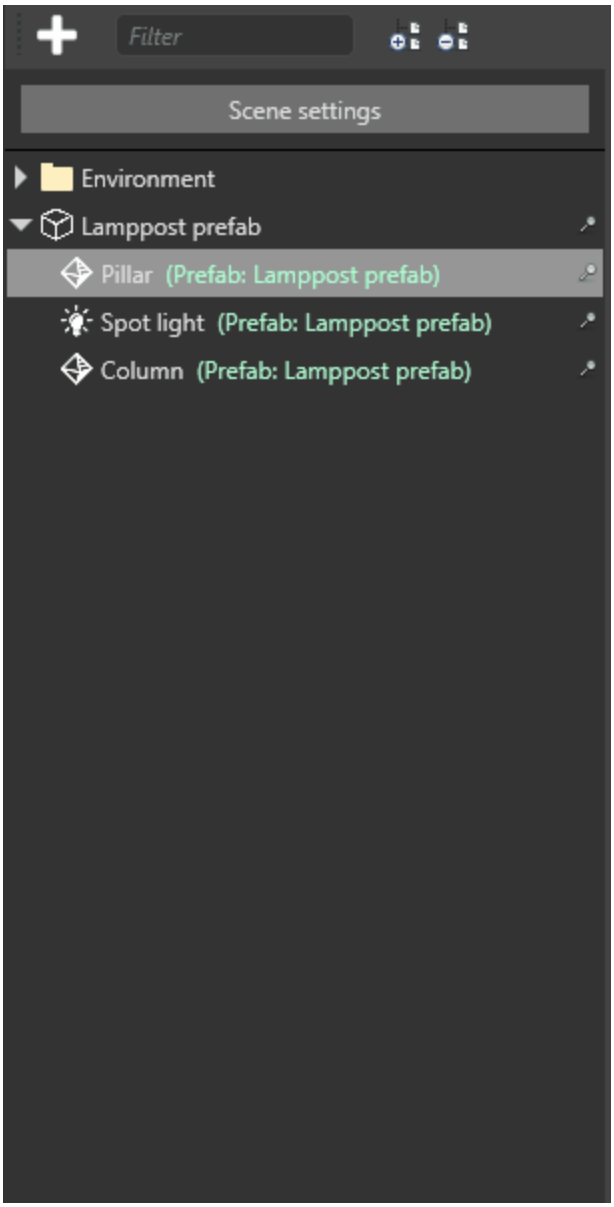
In this case, a parent entity is unnecessary. Instead, you can create several instances of the prefab, then re-arrange their individual crate entities to create the effect you need.

Relative positions maintained	Relative positions ignored

Break link to prefab

After you add a prefab instance, you can break the link between the prefab and any of its child entities. This means the child entity is no longer affected by changes you make to the prefab.

To do this, in the **Scene Editor**, right-click a child entity or entities and select **Break link to prefab**.



Instantiate and add prefabs at runtime

To use prefabs at runtime, you need to instantiate them and then add them to the scene in code.

```
public class SpawnPrefabOnStart : StartupScript
{
    public Prefab MyPrefab { get; init; } // init here prevents other scripts from changing
this property

    public override void Start()
    {
        // A prefab may contain multiple entities
        var entities = MyPrefab.Instantiate();
        // Adding them to the scene this entity is on
        Entity.Scene.Entities.AddRange(entities);
```

```
    }
}
```

> **NOTE**
>
> `Instantiate()` by itself isn't enough to add a prefab instance to the scene. You also need to `Add()` or `AddRange()` them to a scene . For example, if your prefab contains a model, the model is invisible until you add the prefab instance. Likewise, if your prefab contains a script, the script won't work until you add the prefab instance.

If you have a prefab named *MyBulletPrefab* in the root folder of your project, you can instantiate and add it with the following code:

```
private void InstantiateBulletPrefab()
{
    // Note that "MyBulletPrefab" refers to the name and location of your prefab asset
    var myBulletPrefab = Content.Load<Prefab>("MyBulletPrefab");

    // Instantiate a prefab
    var instance = myBulletPrefab.Instantiate();
    var bullet = instance[0];

    // Change the X coordinate
    bullet.Transform.Position.X = 20.0f;

    // Adding just the bullet to the root scene
    SceneSystem.SceneInstance.RootScene.Entities.Add(bullet);
}
```

> **NOTE**
>
> At runtime, changes made to prefabs (*myBulletPrefab* in the above example) don't affect existing prefab instances (*bullet* in the above example). Subsequent calls to `Instantiate(Prefab)` include new changes. For example, imagine you have a tree prefab that contains a script to change the tree color from green to red at runtime. The script won't affect existing instances of the prefab; it can only change the color of **future** instances. This means prefabs instantiated after the code runs will have the new color, but existing prefabs won't.

See also

- Prefab index
- Create a prefab
- Edit prefabs
- Nested prefabs
- Override prefab properties
- Prefab models

Edit prefabs

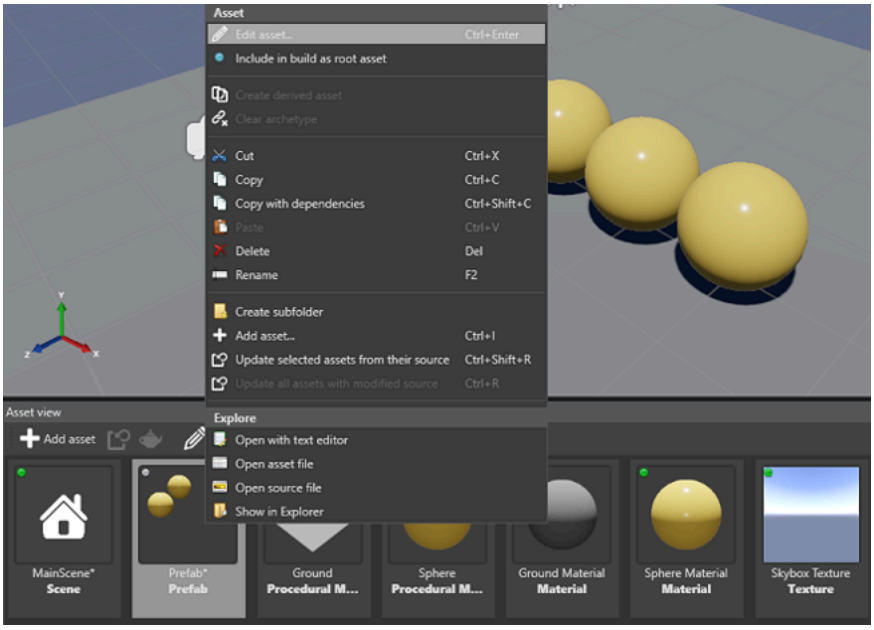
Beginner Designer

You can edit prefabs in the **Prefab Editor**.

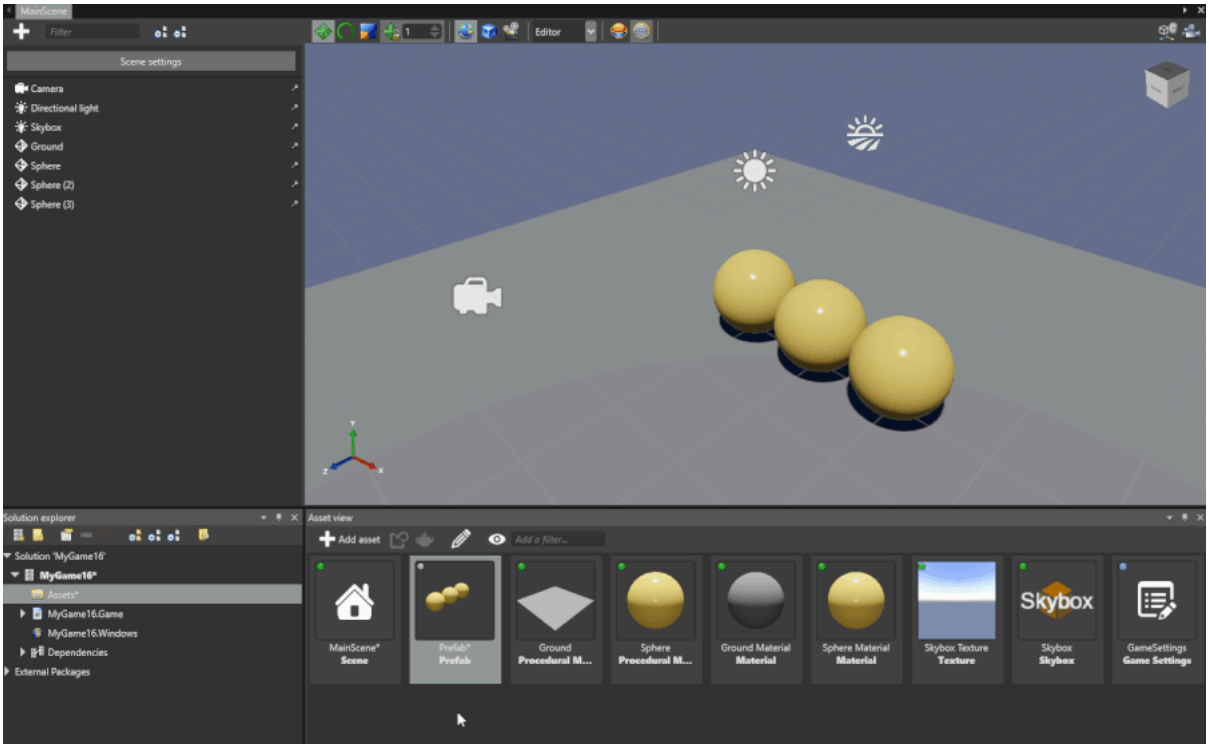
Open the Prefab Editor

To open the Prefab Editor from the **Asset View**:

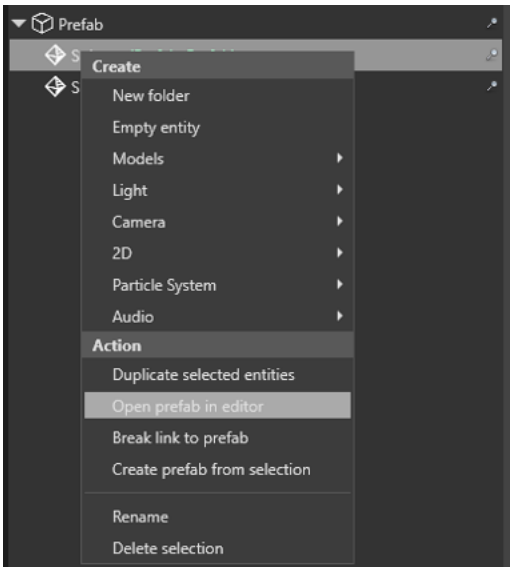
- Right-click the prefab you want to edit and select **Edit asset**:



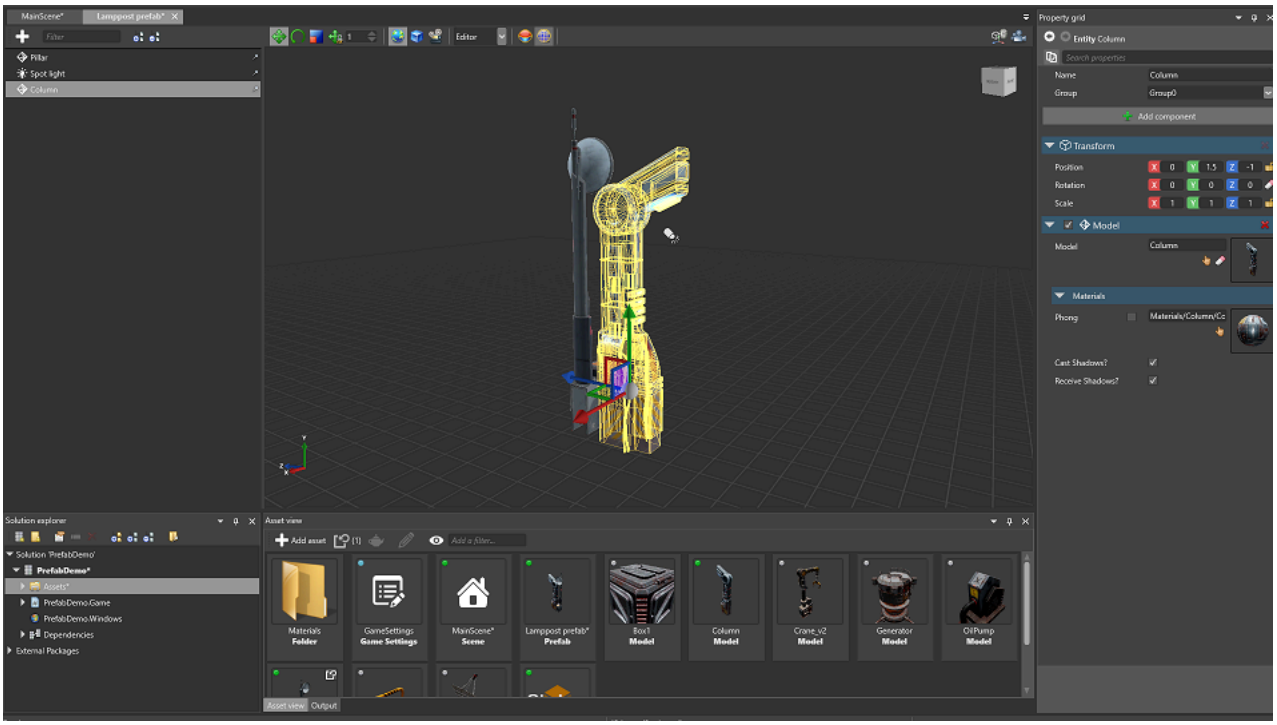
- Alternatively, double-click the prefab you want to edit:



To open the Prefab Editor from the **Scene Editor**, right-click any child of a prefab instance and select **Open prefab in editor**.



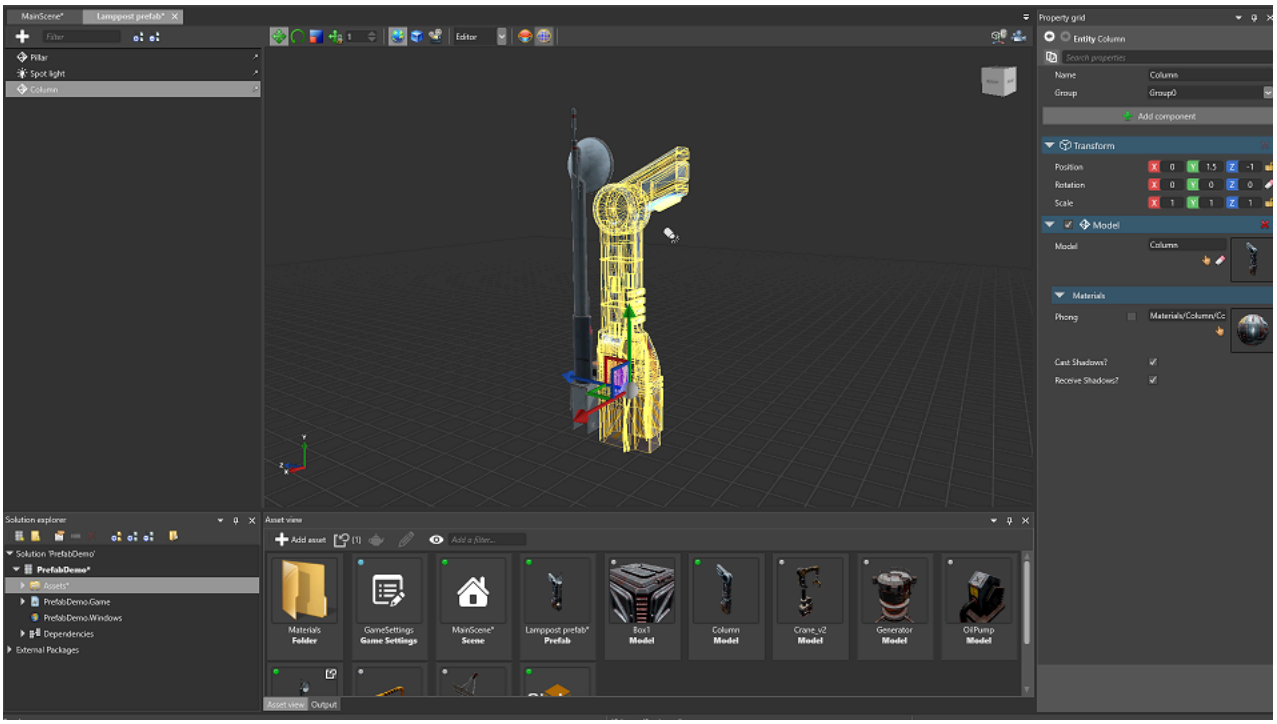
Use the Prefab Editor



The Prefab Editor works similarly to the Scene Editor. For example, you can:

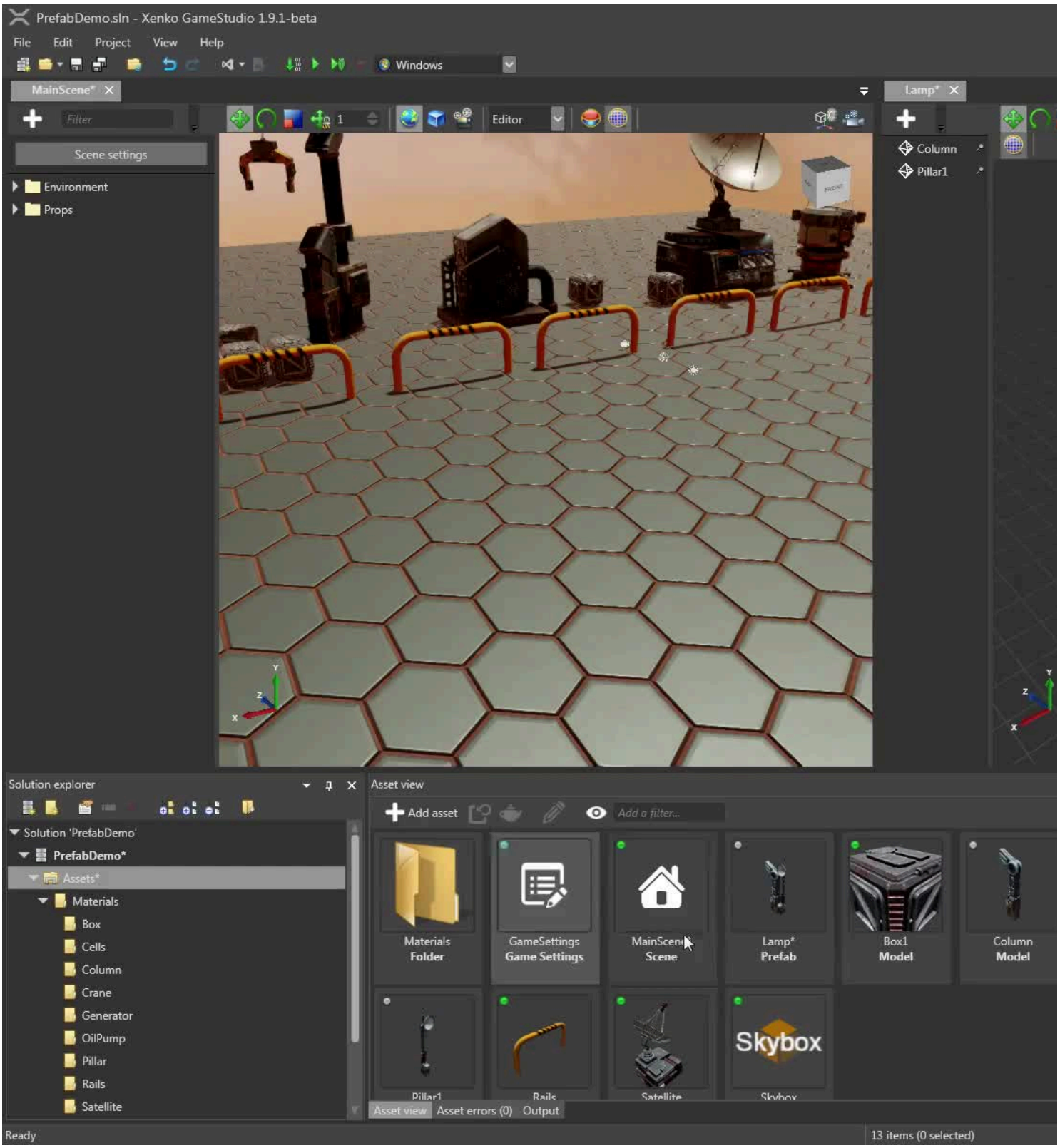
- add and delete entities
- use transformation gizmos to translate, rotate and scale entities
- create parent-child relations between entities
- add and modify entity components (scripts, materials, models, animations, etc)

For more information about managing entities, see Populate a scene.



When you edit a prefab in the Prefab Editor, the changes are applied to the instances of the prefab in the scene in **real time**.

This video demonstrates what happens when we make changes to the prefab. The Scene Editor is on the left, and the Prefab Editor on the right:



See also

- Prefab index
- Create a prefab
- Use prefabs
- Nested prefabs
- Override prefab properties
- Prefab models

Nested prefabs

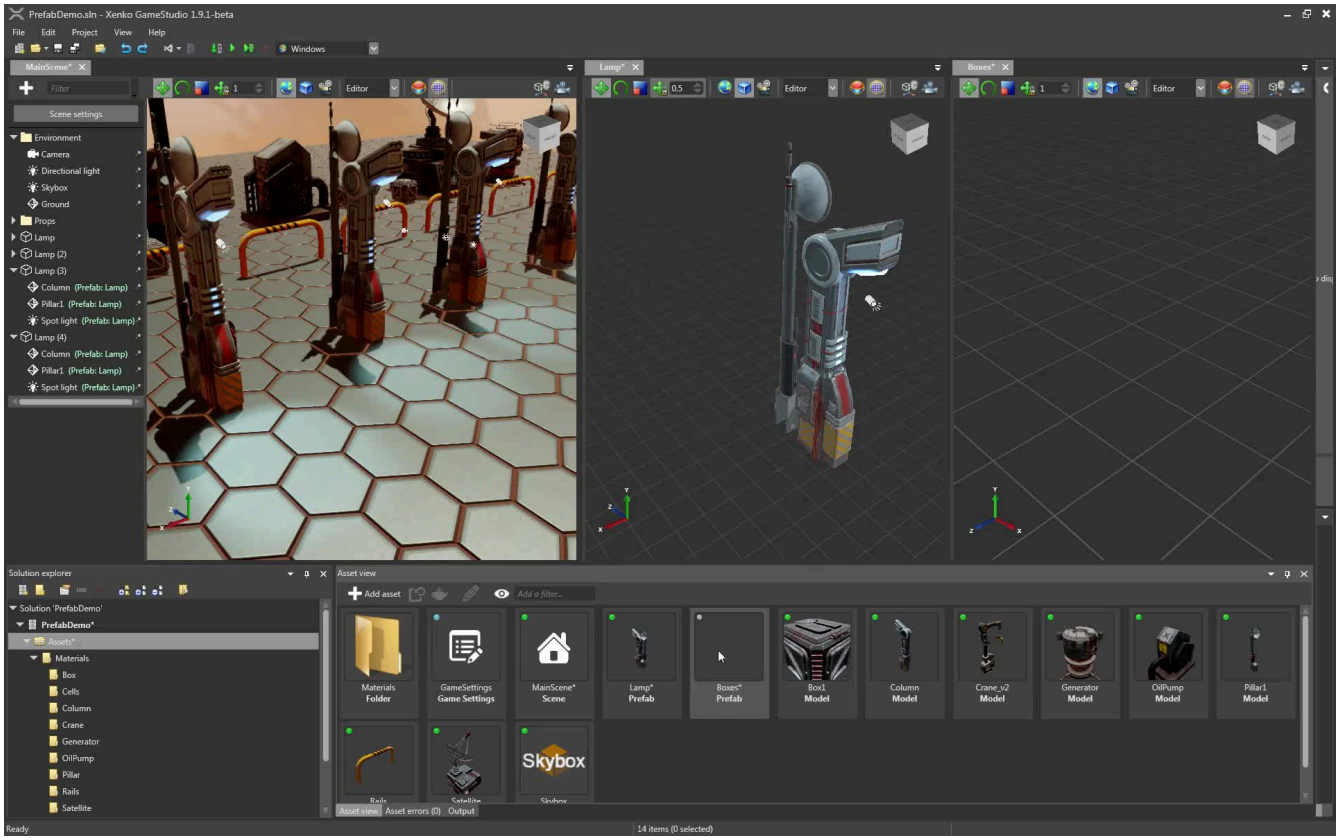
Beginner Designer

You can add prefabs to other prefabs. These are called **nested prefabs**.

For example, imagine you have a table prefab, a chair prefab, and a television prefab. Then you create a living room prefab, which in turn contains the table, chair, and television prefabs. You might then create a house prefab, which in turn contains the living room prefab, which in turn contains the table, chair, and television prefabs. There's no limit to how many prefabs you can nest.

If you modify a nested prefab, all the dependent prefabs inherit the change automatically. For example, if you change the shape of the table prefab, it changes in the living room and house prefabs too.

This video demonstrates an example of nested prefabs:



In the center pane, we already have a prefab named **Lamp**. In the right pane, we create a new prefab named **Boxes**, comprising several box entities positioned together. We add the Boxes prefab to the Lamp prefab. Changes made to the Boxes prefab are reflected in the Lamp prefab. These are in turn reflected in the instances of the Lamp prefab in the scene (left pane).

See also

- Prefab index
- Create a prefab

- Use prefabs
- Edit prefabs
- Override prefab properties
- Prefab models

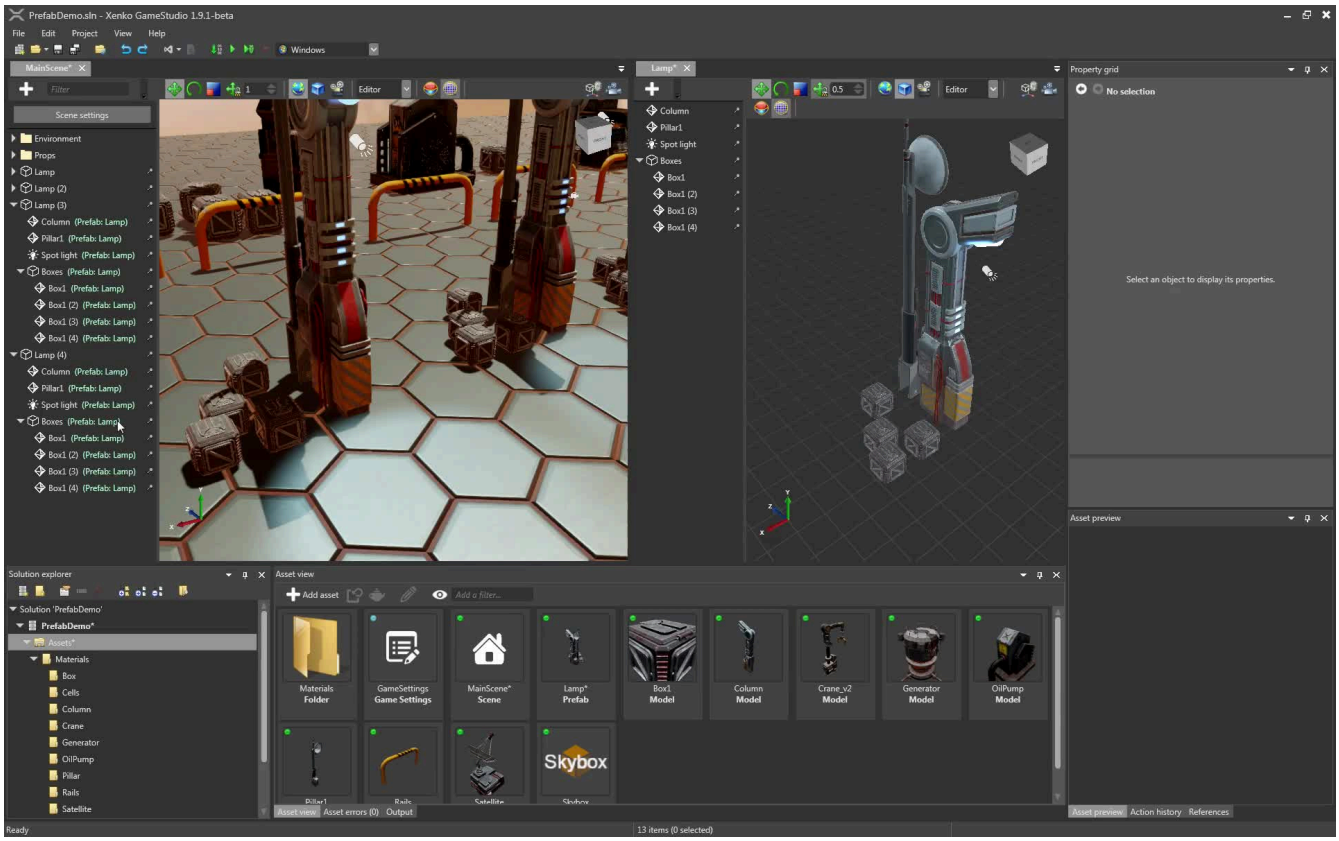
Override prefab properties

Intermediate Programmer Designer

If you modify a property in a prefab instance, the instance no longer inherits changes from the prefab for that property. This is called an **override**.

In the following video, the **Lamp** prefab contains several box entities that belong to the **Boxes** parent. When we delete the boxes from the instance, only that instance is affected. The prefab (shown on the right) is unchanged.

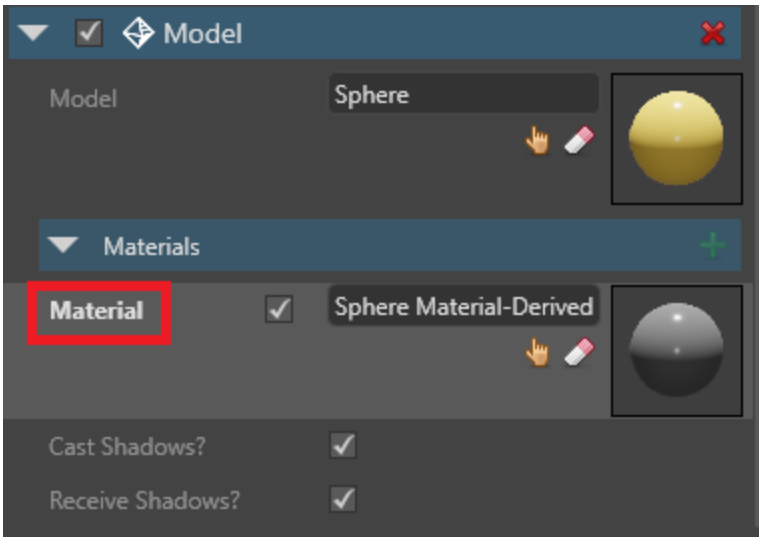
If we add another box to the **Boxes** parent in the prefab, it doesn't appear in the overridden instance. That's because we deleted the **Boxes** parent from that instance.



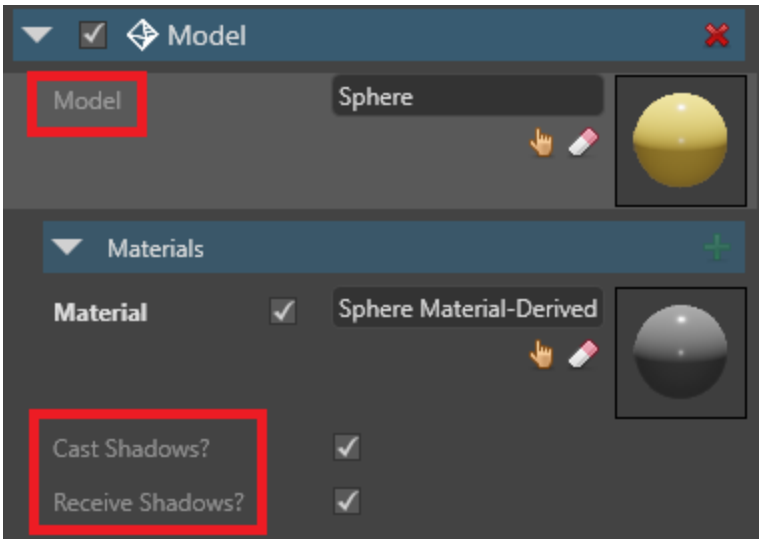
View overridden properties

In the **Property Grid**, you can see which properties of the prefab instance differ from the base values in the prefab.

- **Overridden** and **unique** properties are **white and bold**:

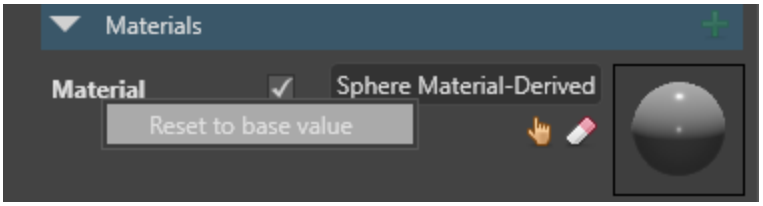


- **Identical** properties are **gray**:



Reset a property to the prefab value

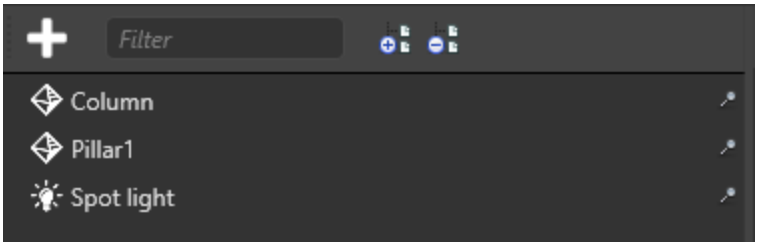
To reset an overridden property to the value in the parent prefab, right-click the property and click **Reset to base value**.



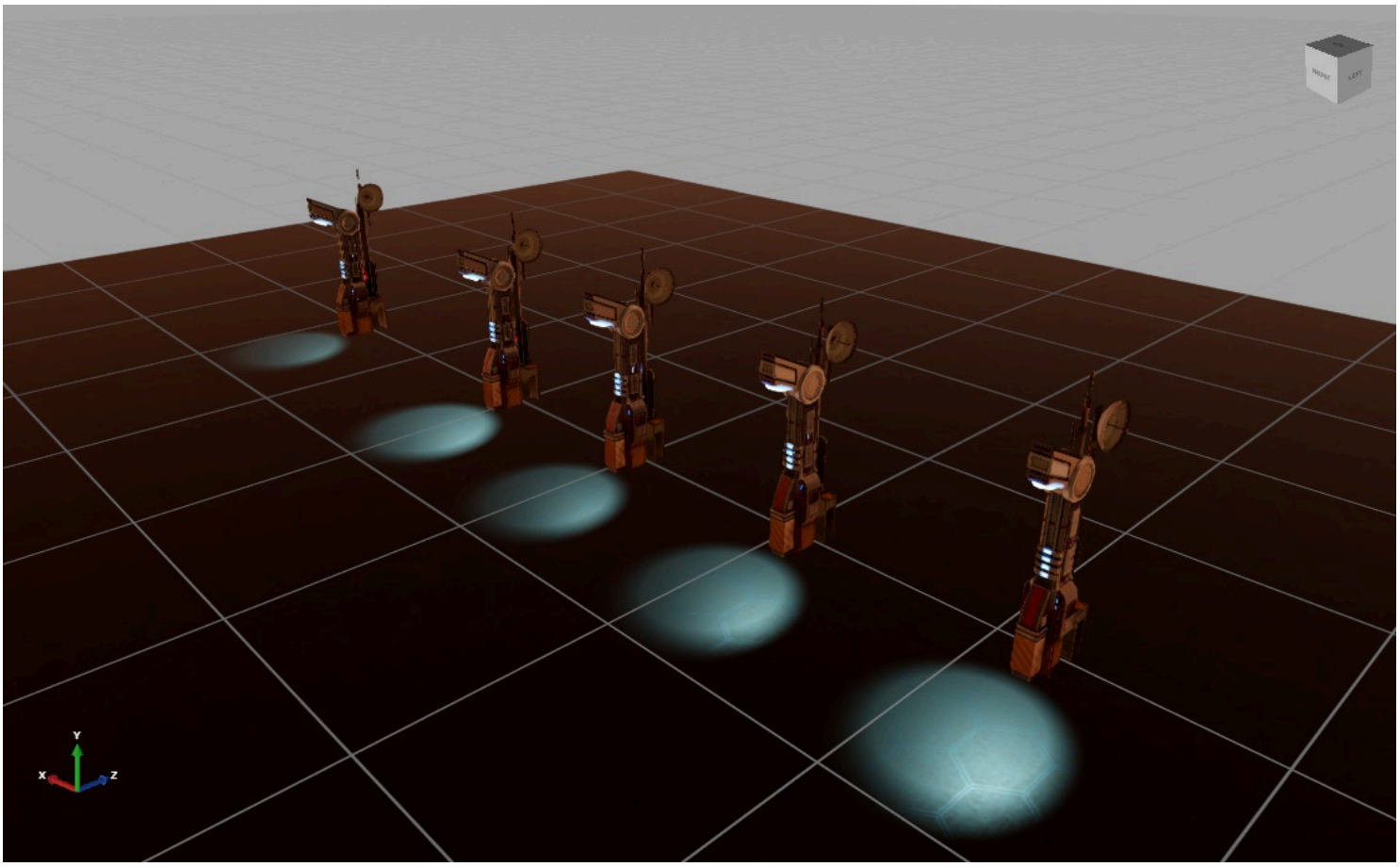
Example

In this example, we have a prefab of a futuristic lamppost.

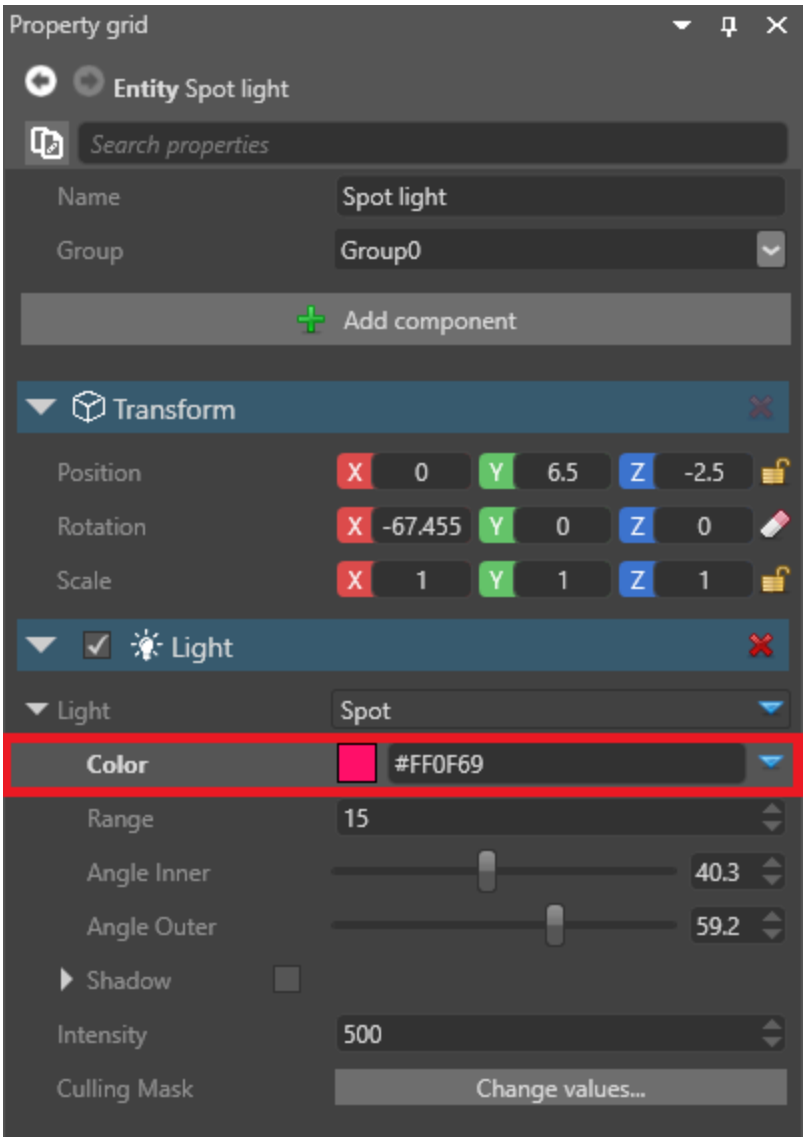
The lamppost prefab is composed of three entities: a column, a pillar, and a spot light. These are listed in the Entity Tree in the Prefab Editor.



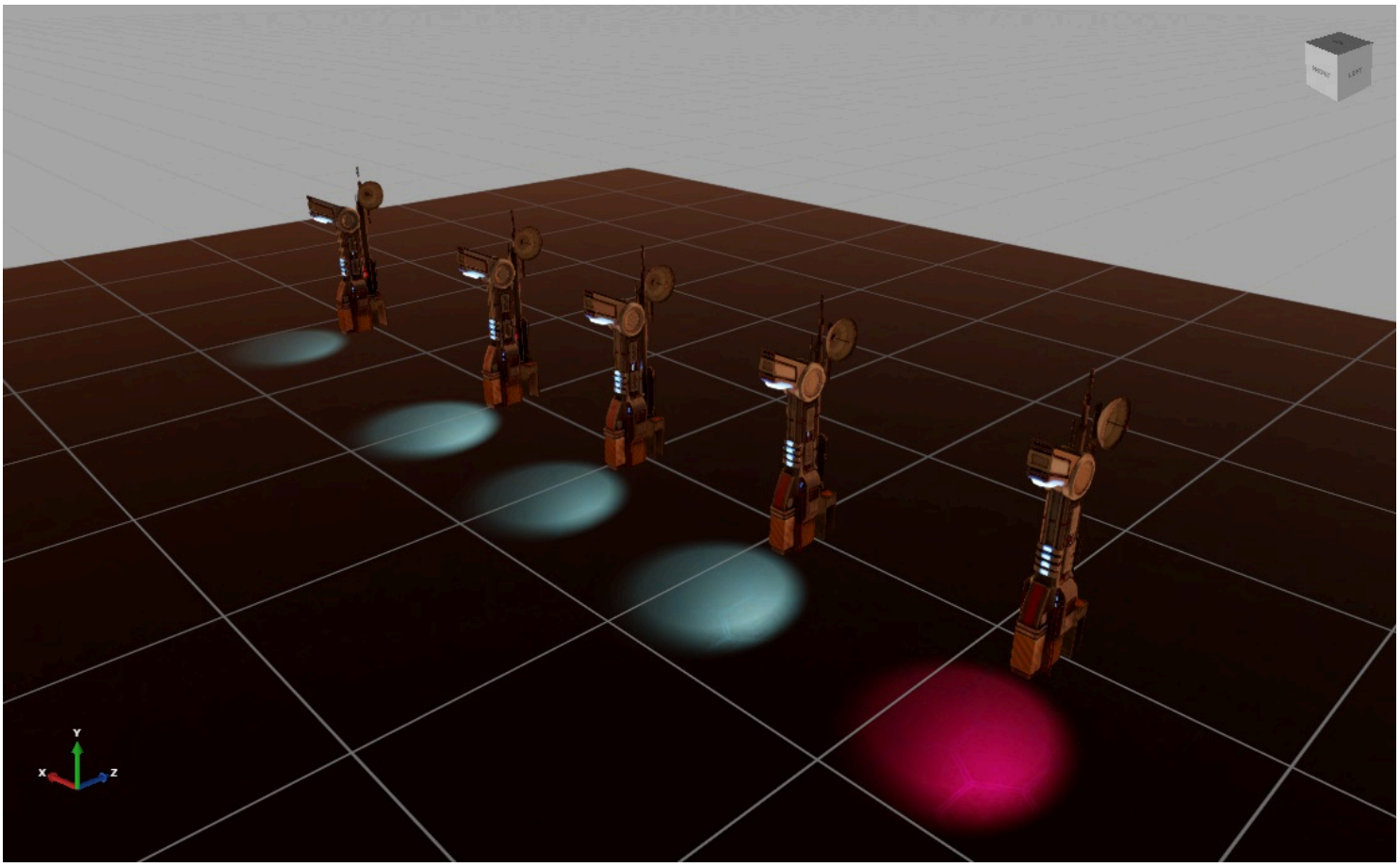
Let's add five instances of the lamppost prefab to our scene.



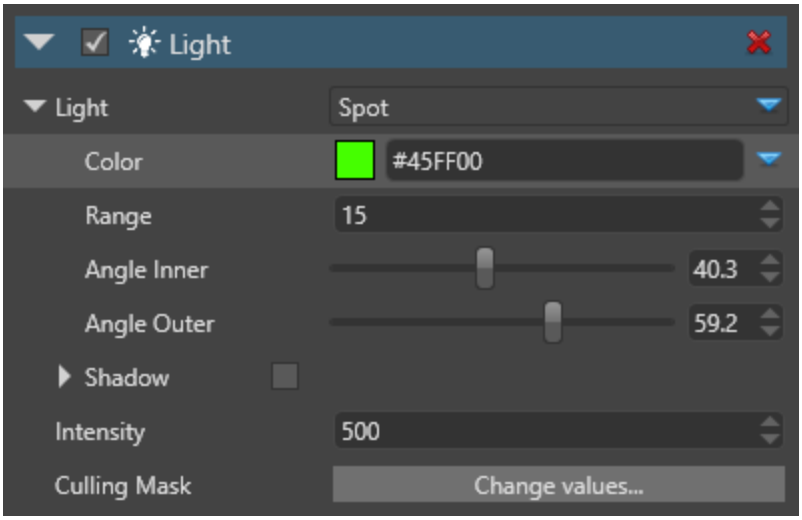
Now we'll modify one of the instances. In the Scene Editor, we select one **spot light** entity and, in the spot light component properties, change its color to red. The Property Grid displays the modified **Color** property in **bold white**. This means it's overriding the prefab property.



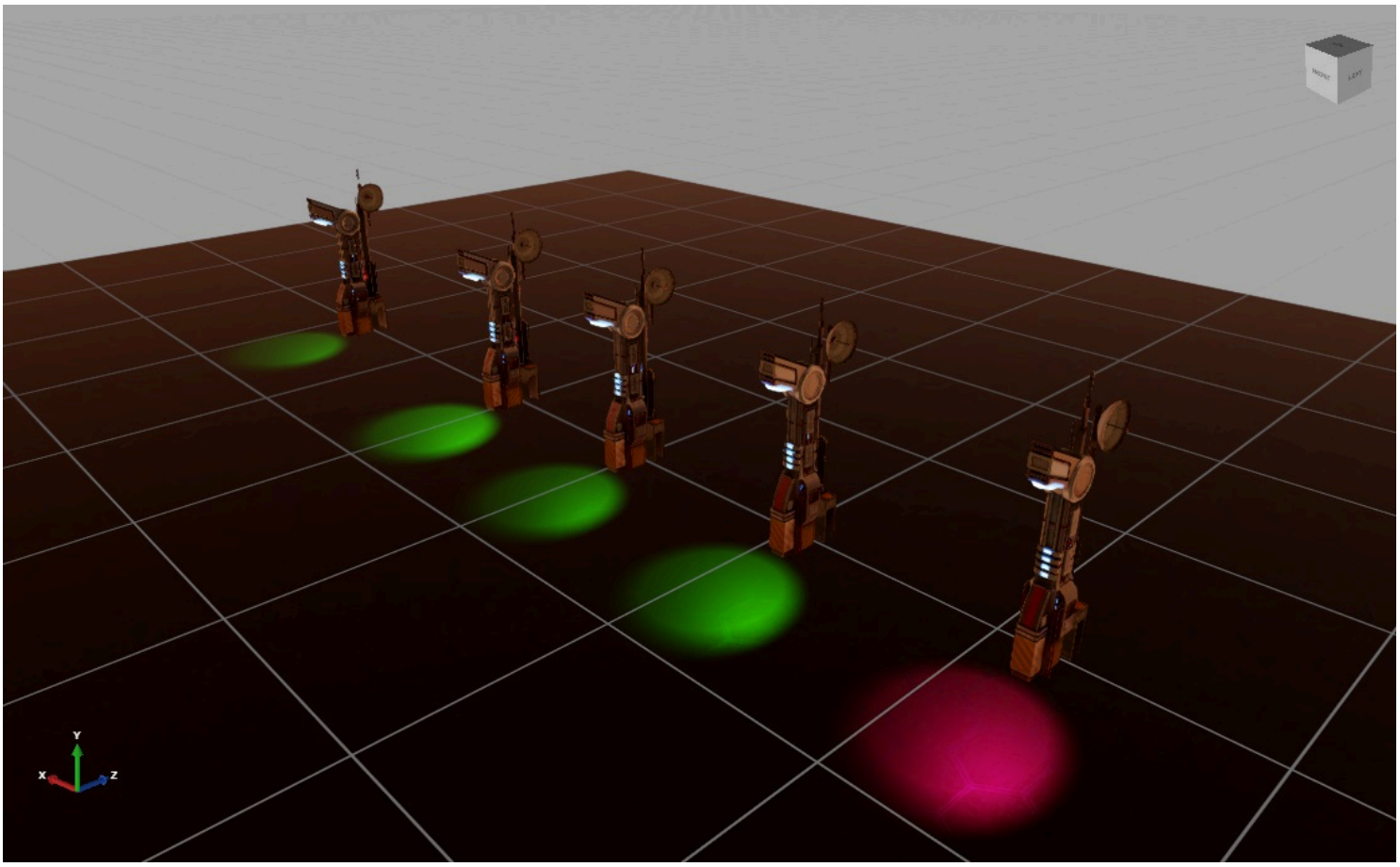
We can see this in the scene view.



Now let's see what happens when we go back to the Prefab Editor and change the color of the spot light in the prefab to green.



Four of the lampposts now have a green light. The fifth is still red, as overridden properties don't change when you modify the prefab.



See also

- Prefab index
- Create a prefab
- Use prefabs
- Edit prefabs
- Nested prefabs
- Prefab models

Prefab models

Beginner Designer

Prefab models convert prefabs to single drawcalls. This is useful for optimization, as Stride only renders the final model instead of the separate entities in the prefab. When you make changes to the prefab, Game Studio regenerates the prefab model.

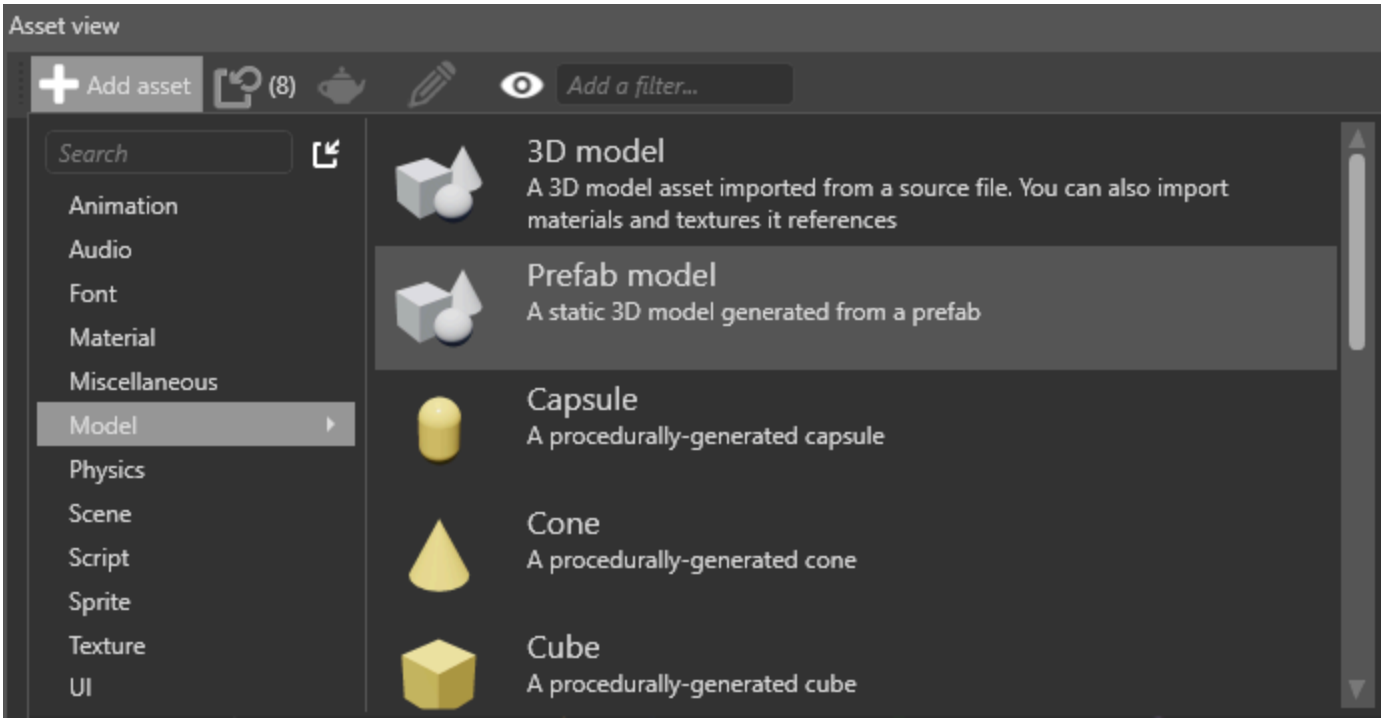
Drawbacks

Prefab models don't inherit elements such as lights, colliders, or other components — they're only models, and have to be used just like other models. For example, if you have a prefab comprising two models with physics components, the prefab model creates a single model from the two models and ignores the physics components. If you need to add components to a prefab model, add them to the prefab model itself.

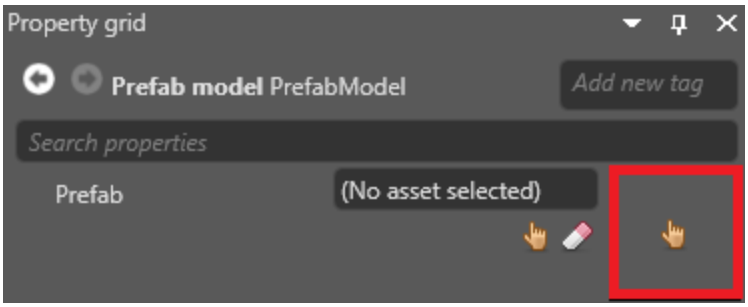
Prefab models don't expose materials. This means you can't view or edit them in the prefab model asset, or in model components that use the prefab model.

Create a prefab model

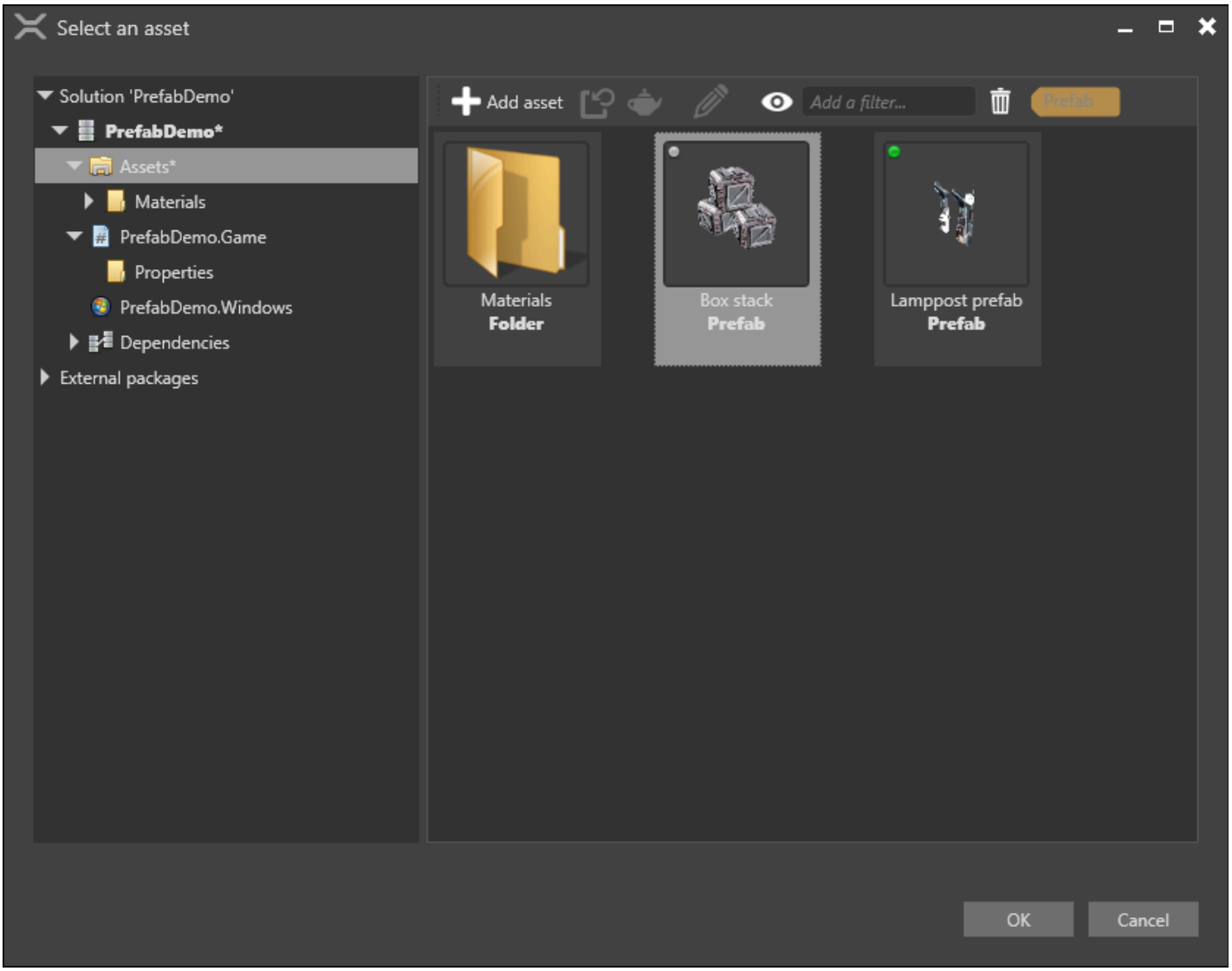
1. In the **Asset View**, select **Add asset** > **Model** > **Prefab model**.



2. In the Property Grid (on the right by default), next to **Prefab**, click (**Select asset**).

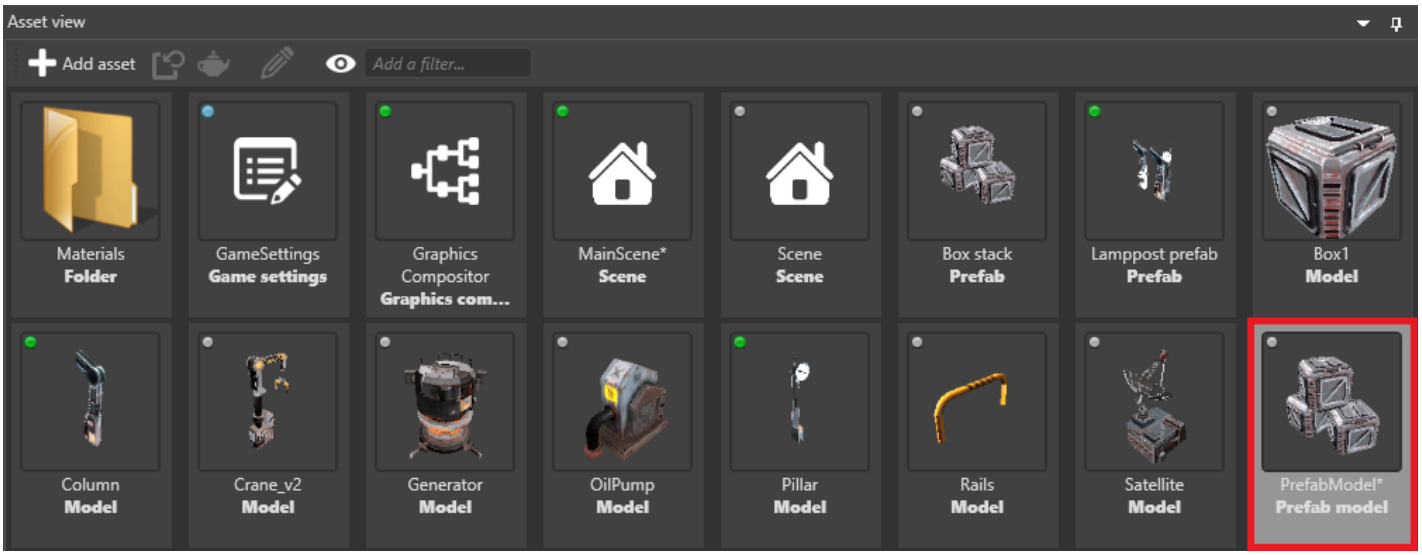


The **Select an asset** window opens.



3. Select the prefab you want to create a model from and click **OK**.

   Game Studio adds the prefab model to the Asset View.



See also

- Create a prefab
- Use prefabs
- Edit prefabs
- Nested prefabs
- Override prefab properties
- Archetypes

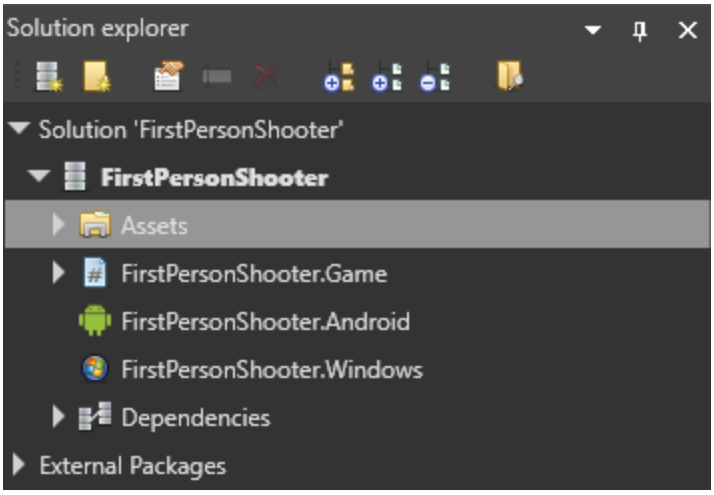
Game settings

Beginner Programmer Designer

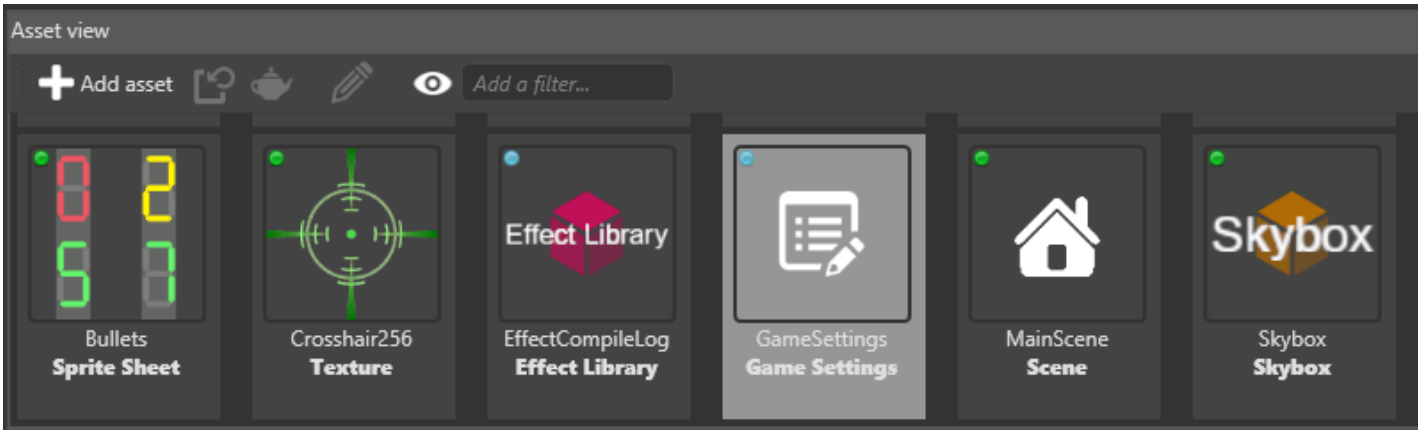
You can configure the global settings of your game in the **Game Settings** asset. By default, the Game Settings asset is stored in your project's **Assets** folder.

Edit game settings

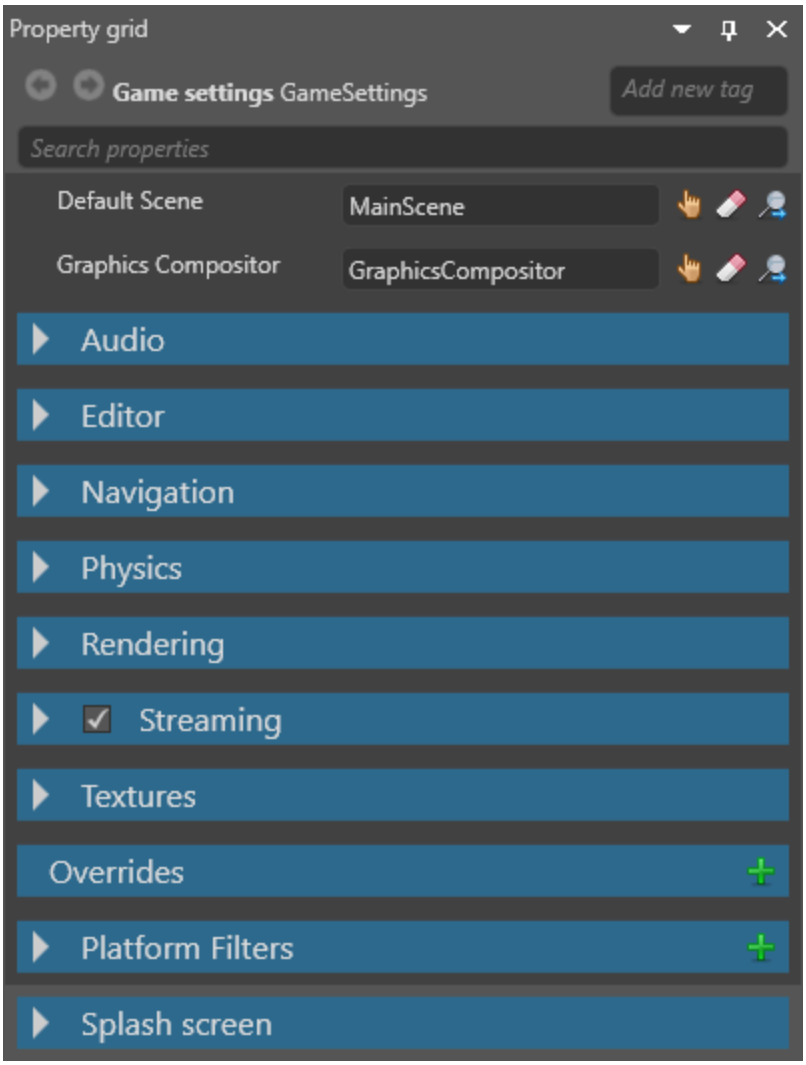
1. In the **Solution Explorer** (the bottom-left pane by default), select the **Assets** folder.



2. In the **Asset View** (the bottom pane by default), select the **GameSettings** asset.



3. In the **Property Grid** (the right-hand pane by default), edit the Game Settings properties.

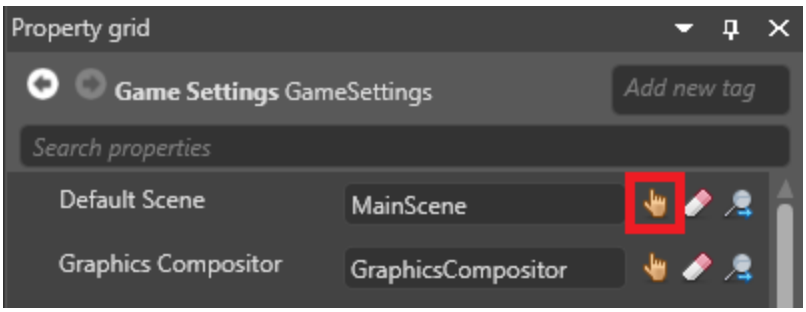


Default scene

You can have multiple scenes in your project. The **default scene** is the scene Stride loads at runtime.

To set the default scene:

1. In the **GameSettings** properties, next to **Default Scene**, click (**Select an asset**).



   The **Select an asset** window opens.

2. Select the default scene and click **OK**.

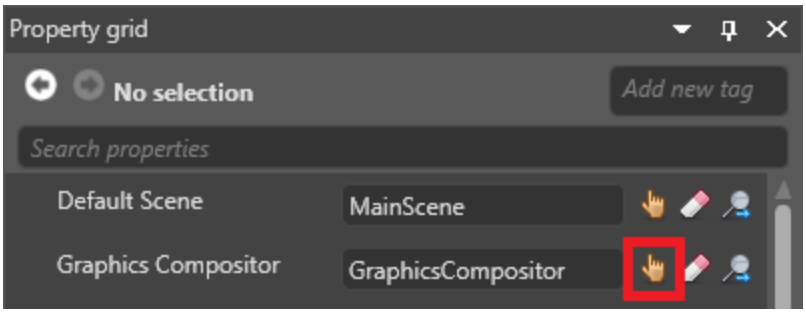
For more information about scenes, see Manage scenes.

Graphics compositor

You can have multiple graphics compositors in your project, but you can only use one at a time.

To set the graphics compositor:

1. In the **GameSettings** properties, next to **Graphics compositor**, click (**Select an asset**).



   The **Select an asset** window opens.

2. Select the graphics compositor and click **OK**.

For more information, see Graphics compositor.

Audio

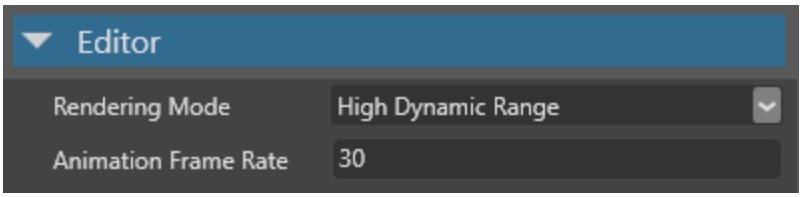
Property	Description
HRTF support	Enable HRTF audio. Note that only audio emitters with HRTF enabled will produce HRTF audio. For more details, see HRTF.

For more details about audio, see Audio.

Editor

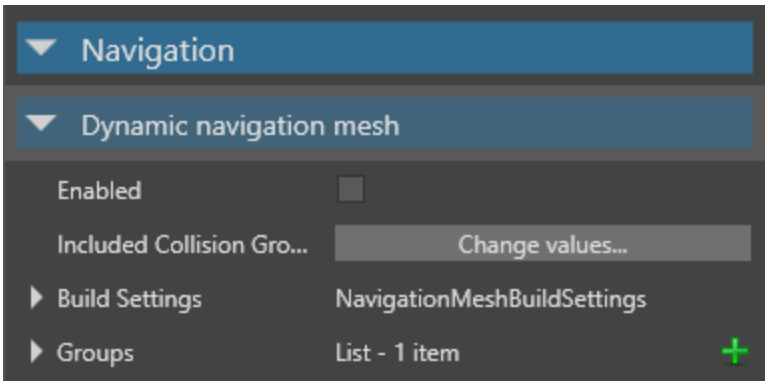
The **editor** settings control how Game Studio displays entities in the Scene Editor. These settings have no effect on your game.

> **NOTE**
>
> How Game Studio displays entities is also affected by the **Color space** setting under **Rendering**.



Property	Description
Rendering mode	How Game Studio renders thumbnails and Asset Previews
Animation framerate	The framerate of animations shown in Game Studio. This doesn't affect animation data.

Navigation



Dynamic navigation mesh properties

Property	Description
Enabled	Enable dynamic navigation on navigation components that have no assigned navigation mesh
Included collision groups	Set which collision groups dynamically-generated navigation meshes use. By default, meshes use all collision groups
Build settings	Advanced settings for dynamically-generated navigation meshes

For more details, see Dynamic navigation.

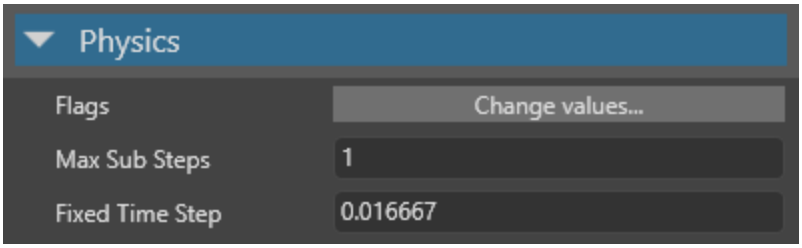
Navigation group properties

Property	Description
Item	The name of the group.

Property	Description
Height	The height of the entities in this group. Entities can't enter areas with ceilings lower than this value.
Maximum climb height	The maximum height that entities in this group can climb.
Maximum slope	The maximum incline (in degrees) that entities in this group can climb. Entities can't go up or down slopes higher than this value.
Radius	The larger this value, the larger the area of the navigation mesh entities use. Entities can't pass through gaps of less than twice the radius.

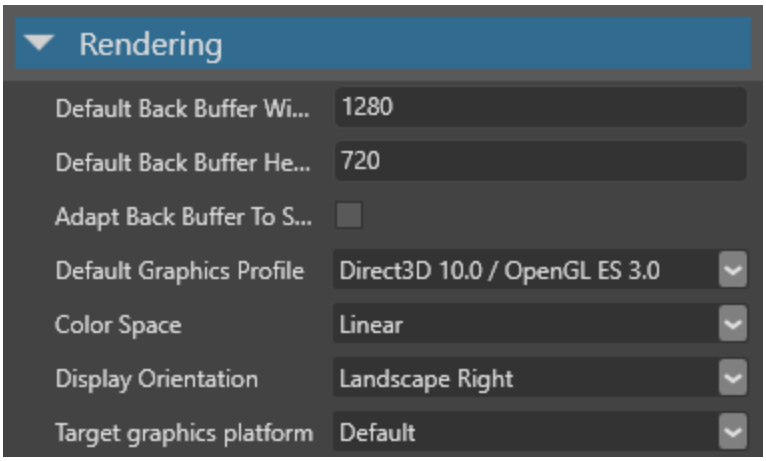
For more details, see Navigation.

Physics



Property	Description
Flags	**CollisionsOnly** disables physics except for collisions. For example, if this is enabled, objects aren't moved by gravity, but will still collide if you move them manually. **ContinuousCollisionDetection** prevents fast-moving entities erroneously moving through other entities. Note: other flags listed here currently aren't enabled in Stride.
Max sub steps	The maximum number of simulations the physics engine can run in a frame to compensate for slowdown.
Fixed time step	The length in seconds of a physics simulation frame. The default is 0.016667 (one sixtieth of a second).

Rendering

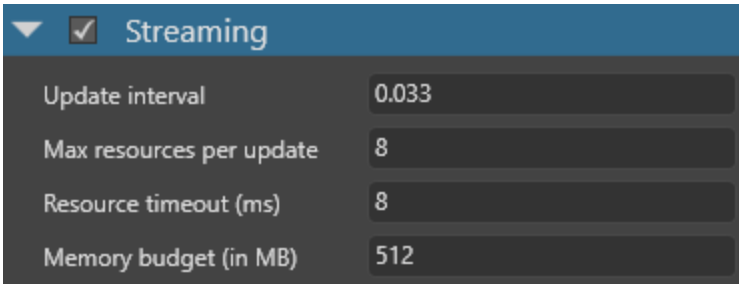


Property	Description
Default back buffer width	This might be overridden depending on the ratio and/or resolution of the device. On Windows, this is the window size. On Android/iOS, this is the off-screen target resolution.
Default back buffer height	This might be overridden depending on the ratio and/or resolution of the device. On Windows, this is the window size. On Android/iOS, this is the off-screen target resolution.
Adapt back buffer to screen	Adapt the ratio of the back buffer to fit the screen ratio
Default graphics profile	The graphics feature level required by the project
Color space	The color space (gamma or linear) used for rendering. This affects the game at runtime and how elements are displayed in Game Studio.
Display orientation	The display orientation of the game (default, portrait, left landscape, or right landscape).
Target graphics platform	The target platform Stride builds the project for. If you set this to **Default**, Stride chooses the most appropriate platform. For more information, see Set the graphics platform.

TIP

To check which default platform your project uses, add a break point to your code (eg in a script), run the project, and check the value of the GraphicsDevice.Platform variable.

Streaming

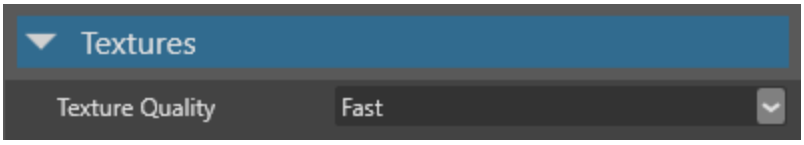


Property	Description
Streaming	Enable streaming
Update interval	How frequently Stride updates the streaming. Smaller intervals mean the streaming system reacts faster, but use more CPU and cause more memory fluctuations.
Max resources per update	The maximum number of textures loaded or unloaded per streaming update. Higher numbers reduce pop-in but might slow down the framerate.
Resource timeout (ms)	How long resources stay loaded after they're no longer used (when the **memory budget** is exceeded)
Memory budget (in MB)	When the memory used by streaming exceeds this budget, Stride unloads unused textures. You can increase this to keep more textures loaded when you have memory to spare, and vice versa.

> **NOTE**
>
> Currently, only textures can be streamed.

For more details, see Streaming.

Textures

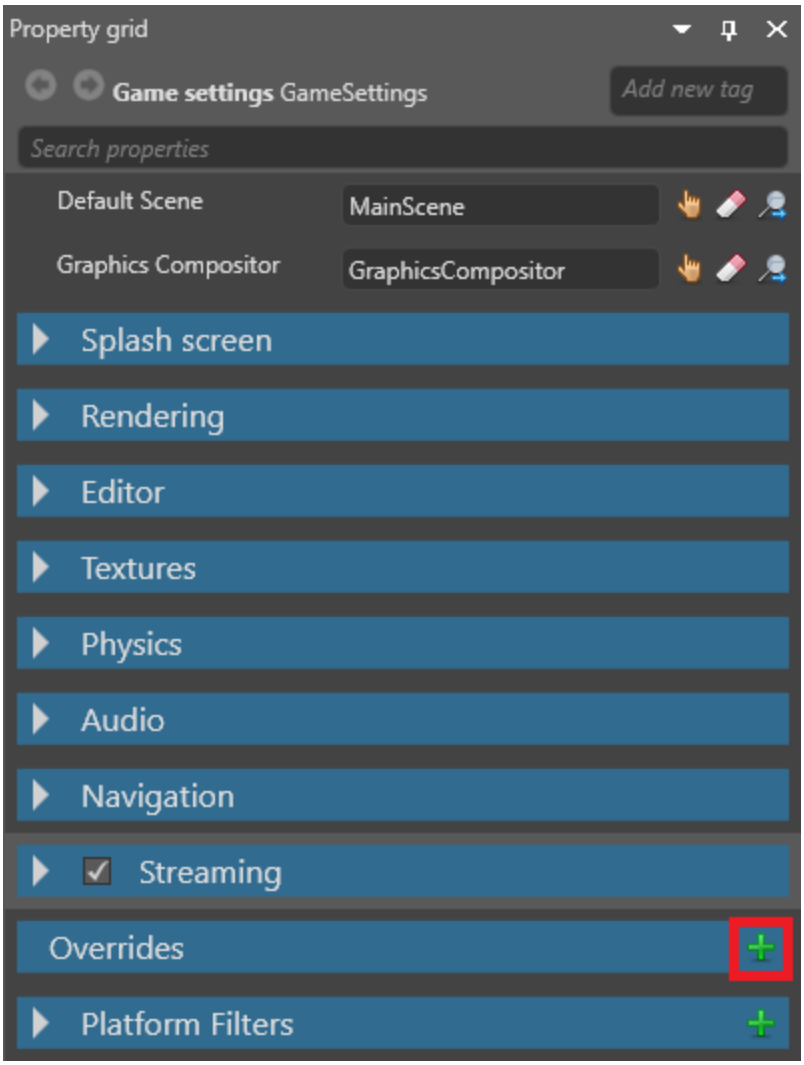


Property	Description
Texture quality	The texture quality when encoding textures. **Fast** uses the least CPU, but has the lowest quality. Higher settings might result in slower builds, depending on the target platform.

Overrides

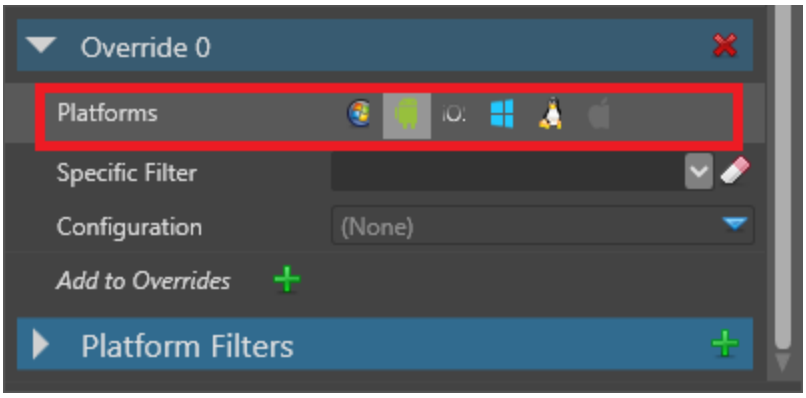
You can override settings for particular platforms, graphics APIs, and so on. For example, you can set different texture qualities for different platforms.

1. With the **GameSettings** asset selected, in the **Property Grid**, under **Overrides**, click (**Add**).

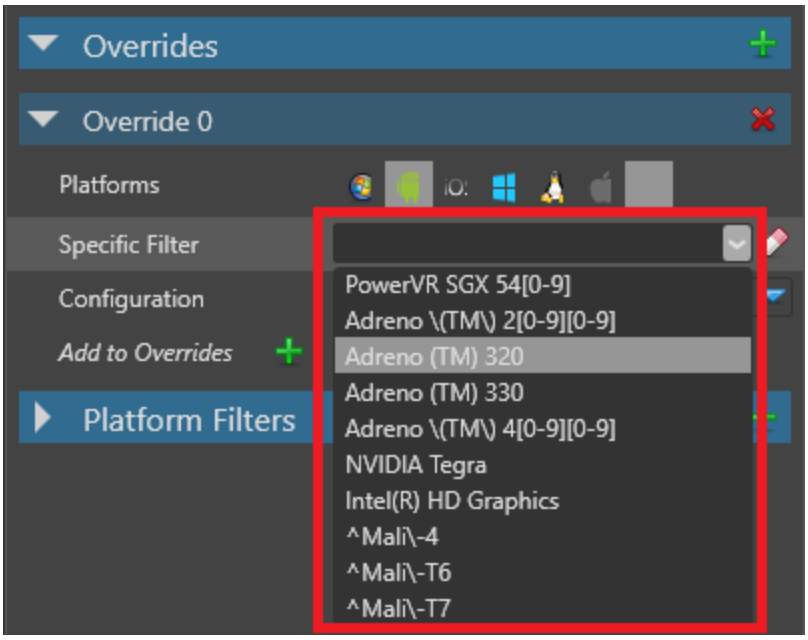


   Game Studio adds an override.

2. In the new override, next to **Platforms**, select the platforms you want the override to apply to. You can select as many as you need.

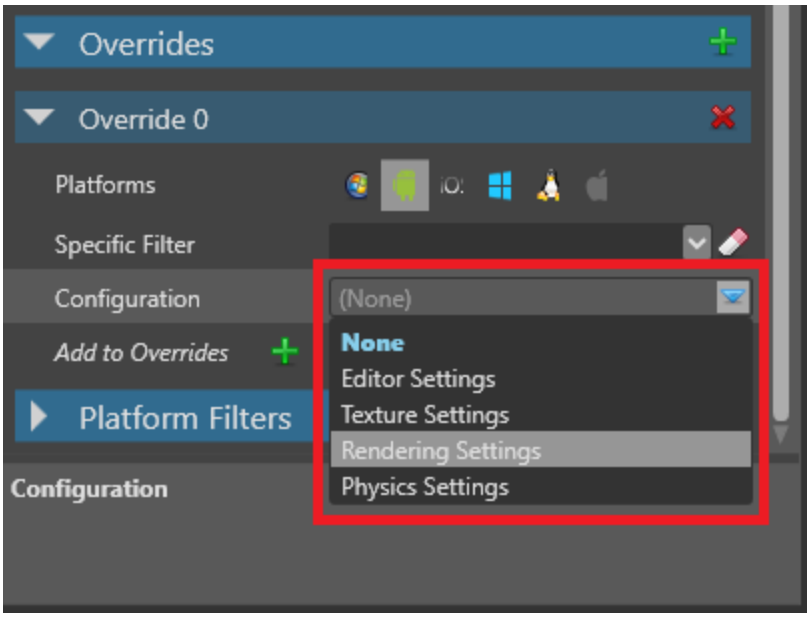


3. **Optional**: If you want this override to apply only to a specific GPU platform, choose it from the **Specific filter** drop-down list.



You can add GPU platforms to this list under **Platform filters** (see **Add a platform filter** below).

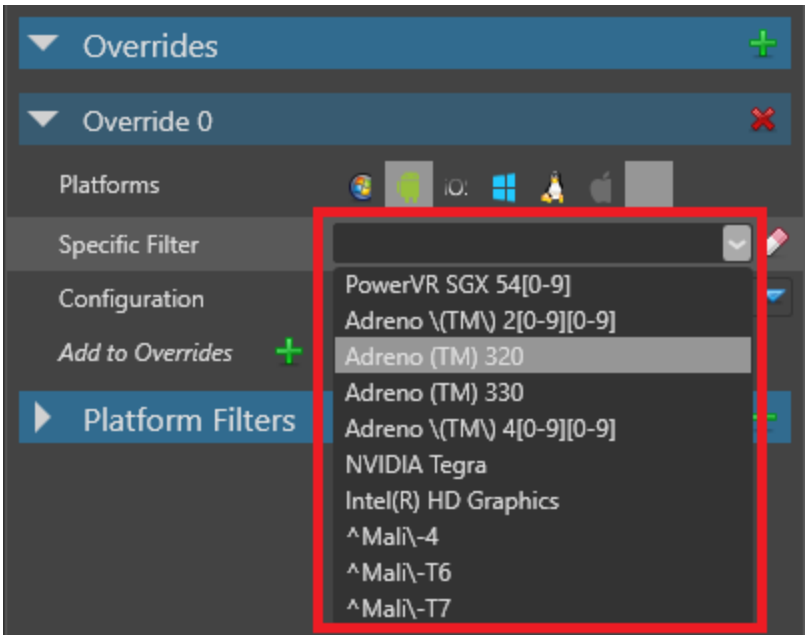
4. In the **Configuration** drop-down menu, select the kind of setting you want to override (**Editor**, **Texture**, **Rendering** or **Physics**).



5. Set the options you want to override.

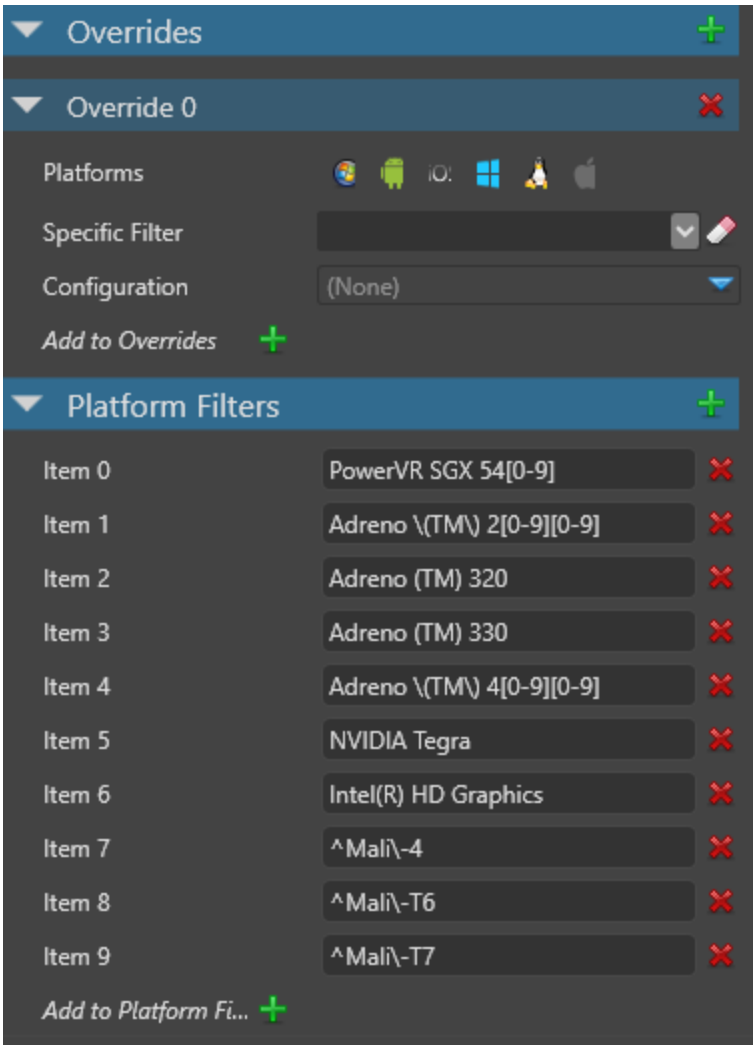
Add a platform filter

You can choose items in the **Platform Filters** list as a specific platform filter when you set an override (see above).



1. With the **GameSettings** asset selected, in the **Property Grid**, expand **Platform Filters**.

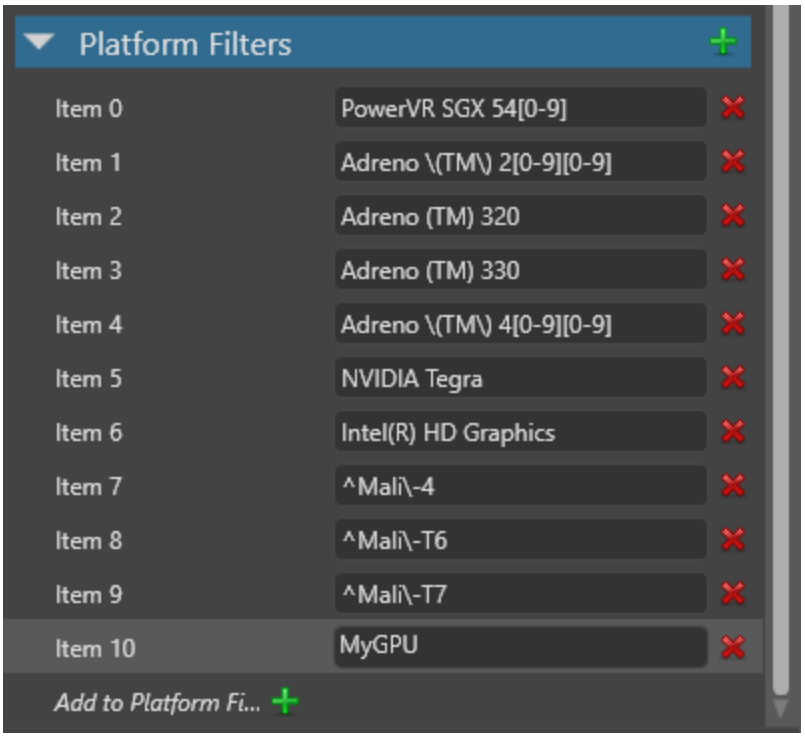
   The Property Grid displays a list of platform filters you can use.



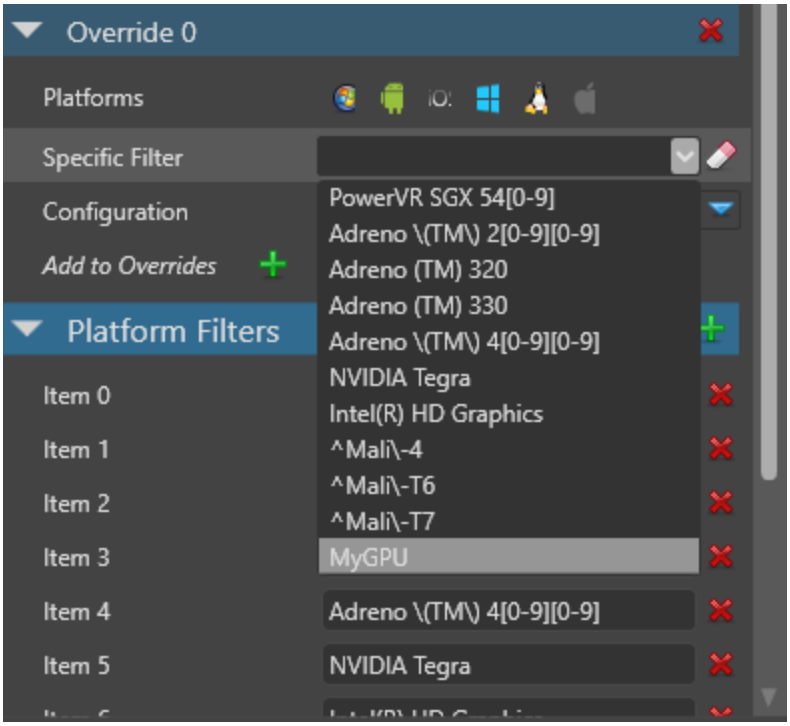
2. At the bottom of the list, click **Add to Platform Filters**.

   Game Studio adds a new empty item.

3. In the item field, type the GPU filter you want to add.



After you add a platform filter, you can select it under **Override > Specific filter**.

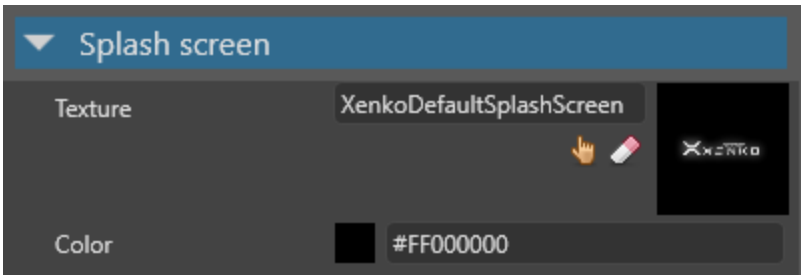


> **NOTE**
>
> If the new filter isn't listed, remove the override and re-add it.

Splash screen

The **splash screen** is displayed when your game starts. The default is the Stride splash screen.

> **NOTE**
> The splash screen is only displayed when the game is built in release mode.



Property	Description
Texture	The image (eg company logo) displayed as the splash screen. By default, this is *StrideDefaultSplashScreen*.
Color	The color the splash screen fades in on top of. By default, this is black (*#FF000000*).

For more information, see Splash screen.

See also

- Assets

Splash screen

Beginner

The **splash screen** is the image (usually a logo) displayed when your game starts. It fades in over the color you specify, then fades out.

> **NOTE**
> The splash screen is only displayed when the game is built in release mode.

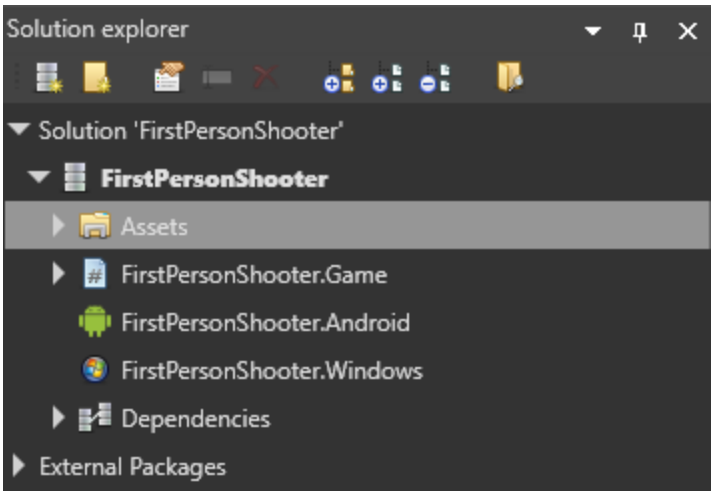
The default splash screen is the Stride logo.

You can only specify one splash screen in Game Settings. If you want to add more, you need to implement them manually.

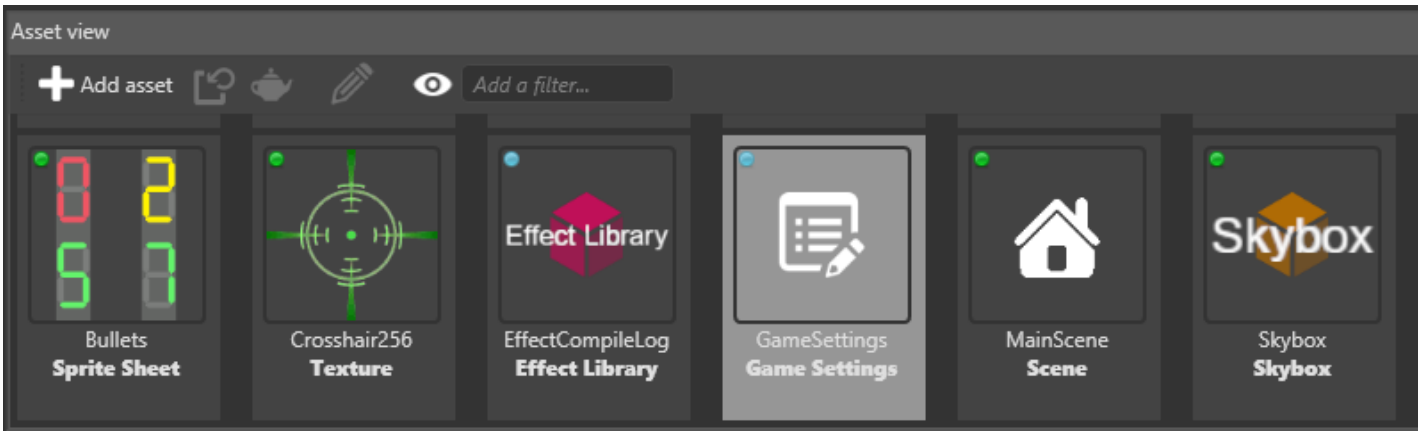
Edit the splash screen

The splash screen settings are part of the **Game settings** asset.

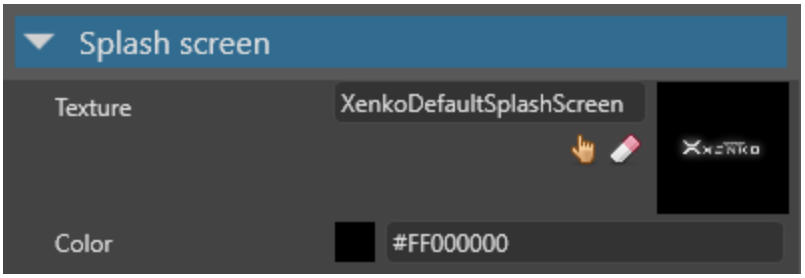
1. In the **solution explorer** (the bottom-left pane by default), select the **Assets folder**.



2. In the **asset view** (the bottom pane by default), select the **GameSettings** asset.



3. In the **property grid** (the right-hand pane by default), expand **Splash screen**.



Splash screen properties

Property	Description
Texture	The image (eg company logo) displayed as the splash screen. By default, this is *StrideDefaultSplashScreen*.
Color	The color the splash screen fades in on top of. By default, this is black (*#FF000000*).

> **TIP**
>
> Additionally, you might want to **disable streaming** on the properties of the splash screen texture itself. This makes sure the texture is always loaded and displayed at the highest quality. For more information, see Textures > Streaming.

See also

- Assets
- Textures

World units

In Stride, one unit is one meter. This is used by the physics and rendering engines.

Game Studio displays units as a grid.

See also

- Physics

Graphics

This section explains how to use Game Studio and the Stride API for graphics and rendering.

Shaders

Shaders are authored in the Stride's shading language, an extension of `HLSL`. They provide true composition of modular shaders through the use of inheritance, shader mixins, and automatic weaving of shader in-out attributes.

Effects

Effects in Stride use C#-like syntax to combine shaders. They provide conditional composition of shaders to generate effect permutations.

Target everything

Stride shaders are converted automatically to the target graphics platform, either plain HLSL for Direct3D, `GLSL` for OpenGL, or `SPIR-V` for Vulkan platforms.

Advanced graphics

The graphics module provides a set of methods to display the game. Although Stride is available on multiple platforms, the whole system behaves like Direct3D 11 from the user perspective. You need a basic knowledge of the rendering pipeline to use it.

In this section

- Cameras
- Materials
- Textures
- Lights and shadows
- Post effects
- Graphics compositor
- Effects and shaders
- Low-level API
- Rendering pipeline
- Sprite fonts
- Voxel Cone Tracing GI
- Graphics API

Cameras

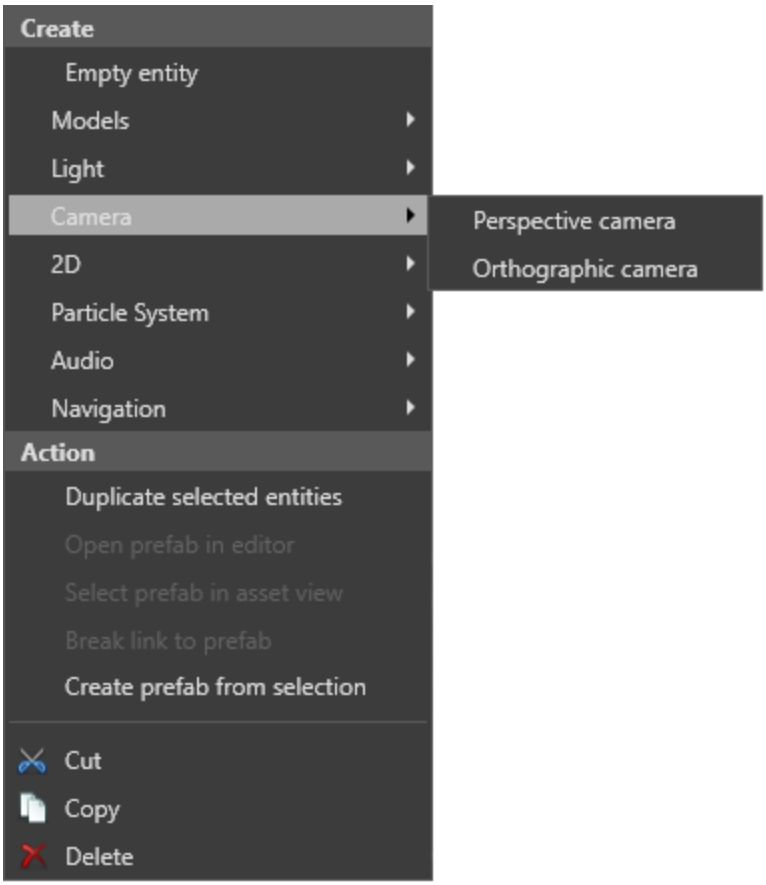
Beginner Designer

Cameras capture your scene and display it to the player. Without cameras, you can't see anything in your game.

You can have an unlimited number of cameras in your scene.

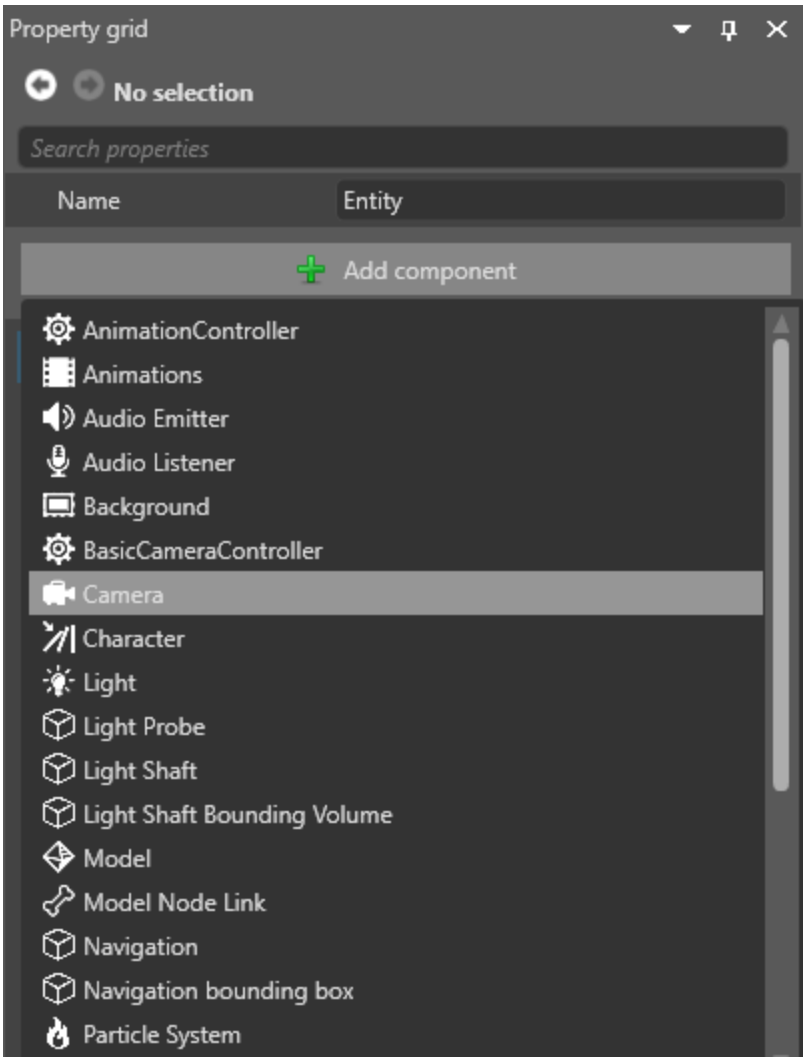
Create a camera in Game Studio

In the Scene Editor, right-click and select **Camera**, then choose the kind of camera you want to create (**perspective** or **orthographic**).

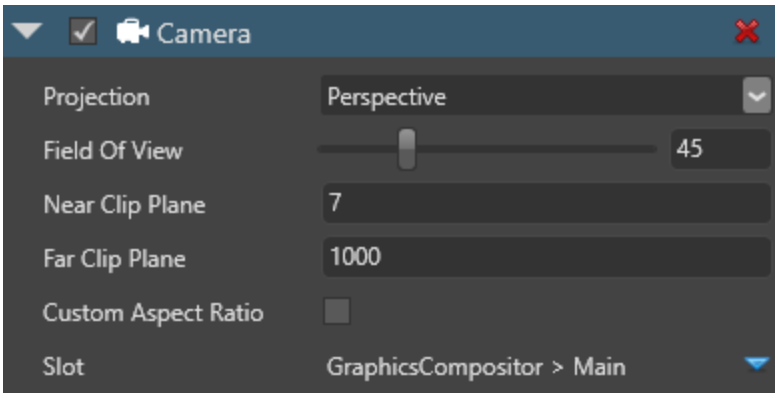


Game Studio creates an entity with a camera component attached.

Alternatively, select the entity you want to be a camera, and in the **Property Grid**, click **Add component** and select **Camera**.



Camera properties



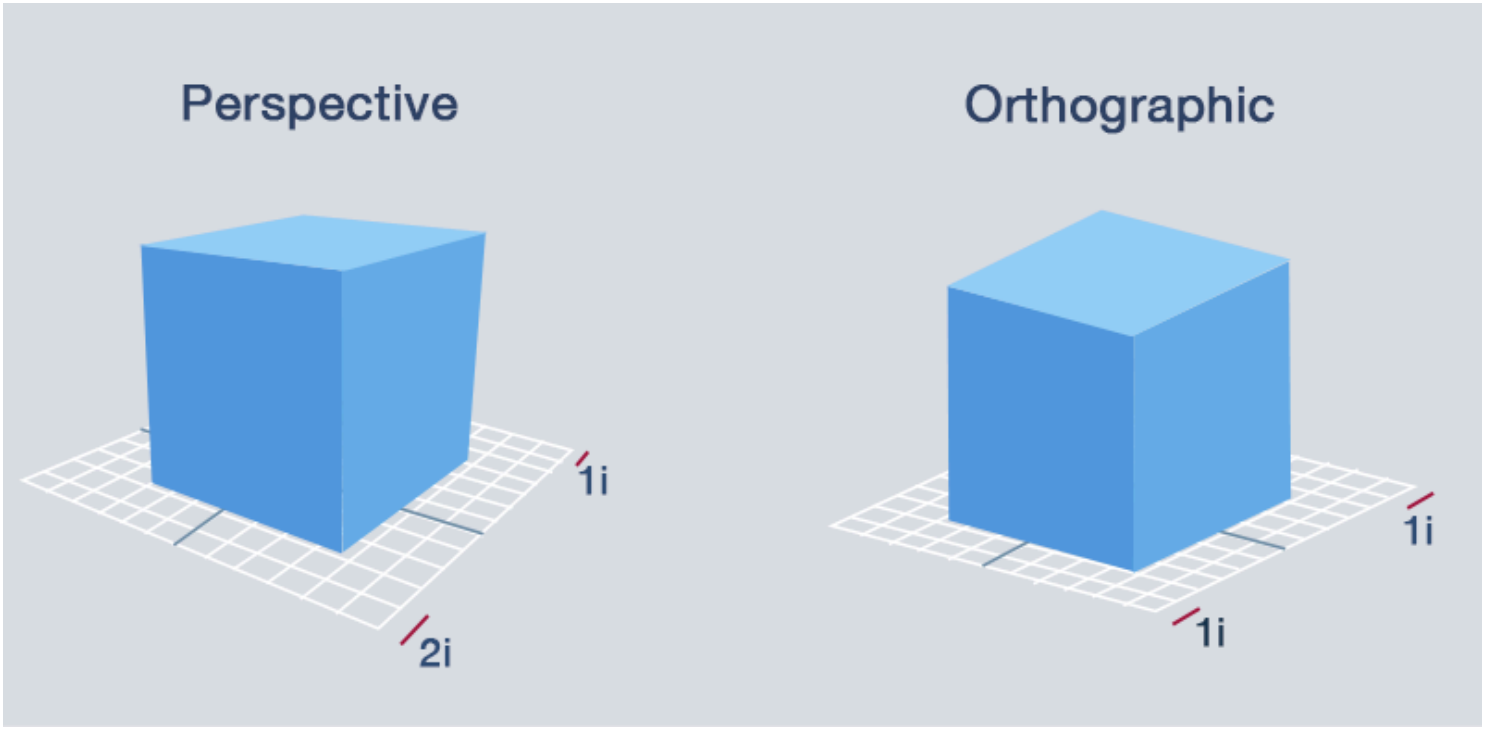
Property	Description
Projection	The type of projection used by the camera (perspective or orthographic)
Field of view (degrees)	The vertical field of view used for perspective projection

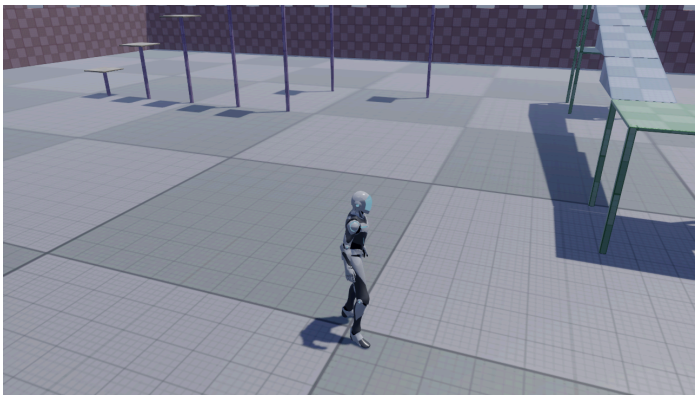
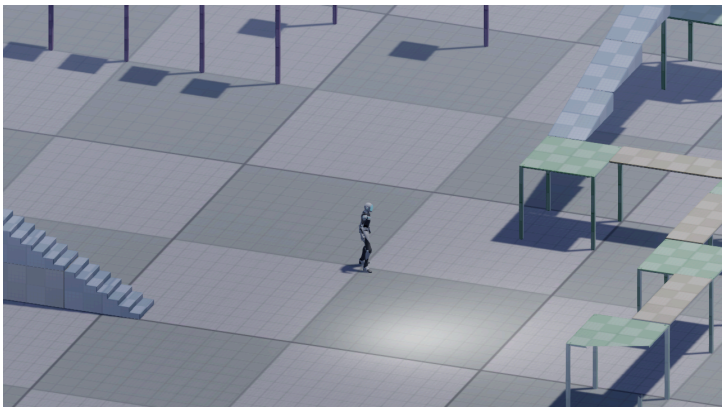
Property	Description
Orthographic size	The height of the orthographic projection (the orthographic width is automatically calculated based on the target ratio). This has the effect of zooming in and out
Near clip plane	The nearest point the camera can see
Far clip plane	The furthest point the camera can see
Custom aspect ratio	Use a custom aspect ratio you specify. Otherwise, automatically adjust the aspect ratio to the render target ratio
Custom aspect ratio	The aspect ratio for the camera (when the **Custom aspect ratio** option is selected)
Slot	The camera slot used in the graphics compositor. For more information, see Camera slots

Perspective and orthographic cameras

Perspective cameras provide a "real-world" perspective of the objects in your scene. In this view, objects close to the camera appear larger, and lines of identical lengths appear different due to foreshortening, as in reality. Perspective cameras are most used for games that require a realistic perspective, such as third-person and first-person games.

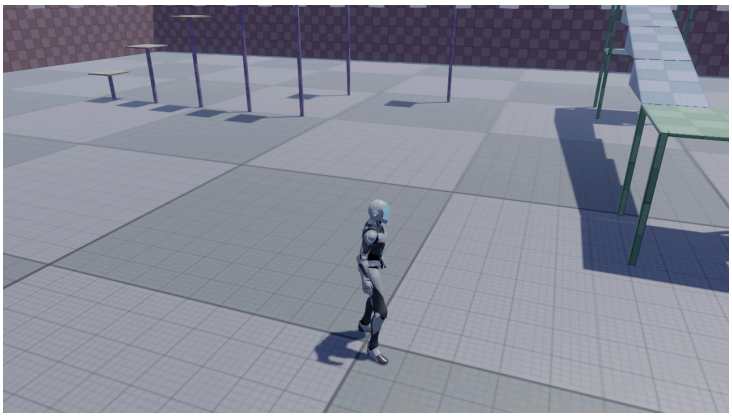
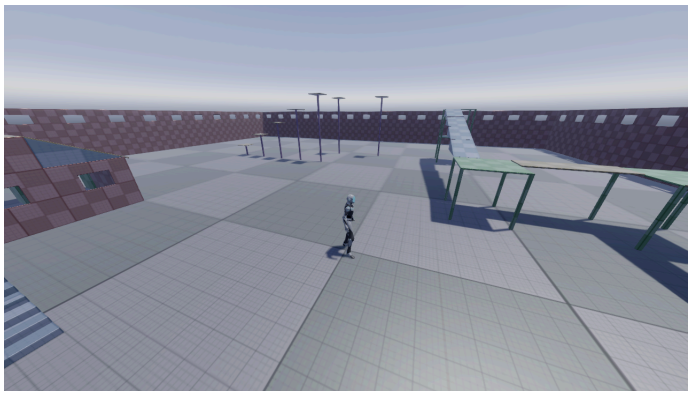
With **orthographic cameras**, objects are always the same size, no matter their distance from the camera. Parallel lines never touch, and there's no vanishing point. Orthographic cameras are most used for games with isometric perspectives, such as some strategy, 4X, or role-playing games.



Perspective	Orthographic
	

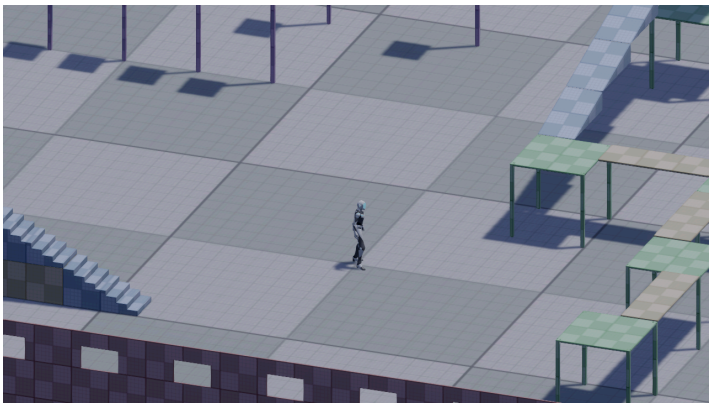
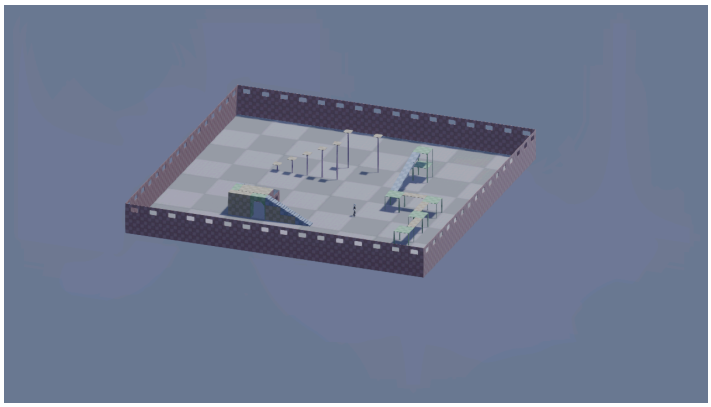
Field of view (perspective mode only)

When the camera is set to **perspective** mode, the **field of view** changes the camera frustum, and has the effect of zooming in and out of the scene. At high settings (90 and above), the field of view creates stretched "fish-eye lens" views. The default setting is 45.

Field of view: 45 (default)	Field of view: 90
	

Orthographic size (orthographic mode only)

When the camera is set to **orthographic** mode, the **orthographic size** has the effect of zooming in and out.

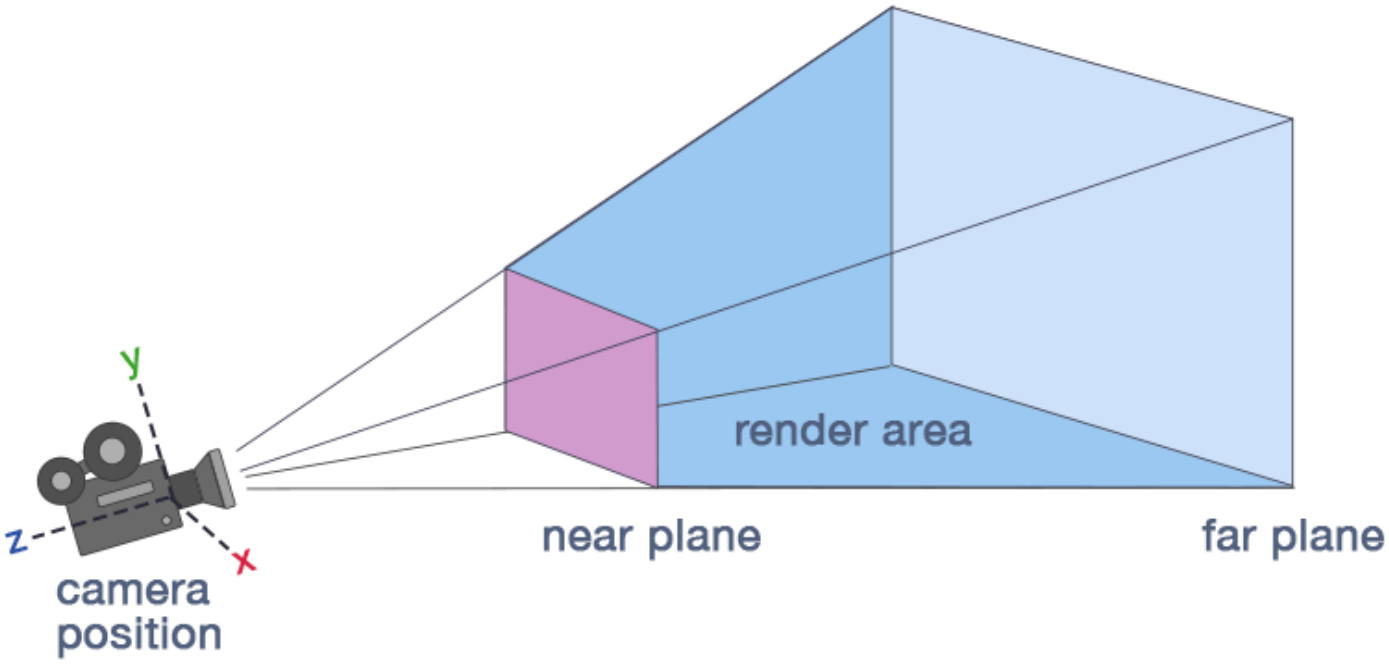
Orthographic size: 10 (default)	Orthographic size: 50
	

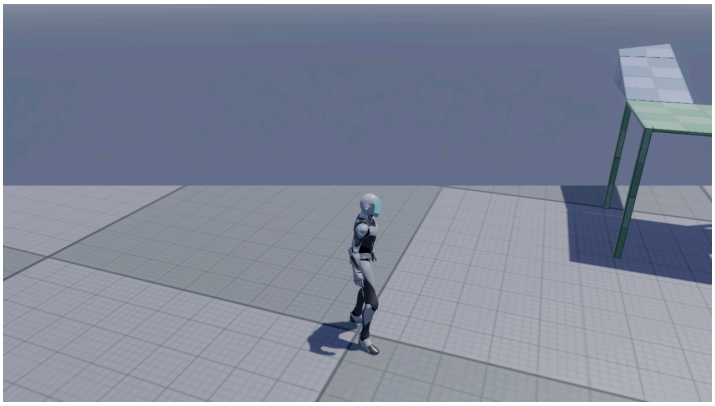
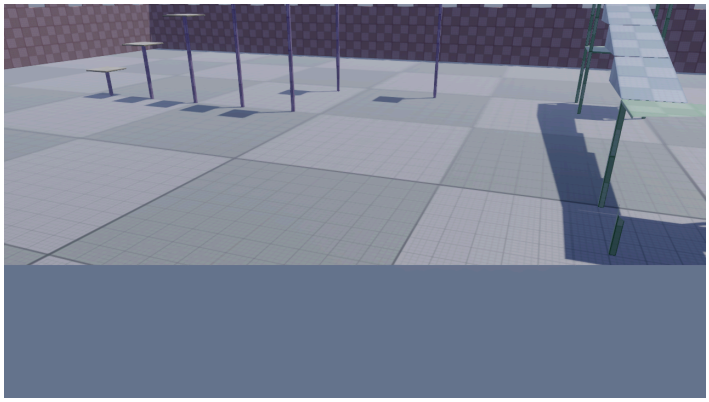
Near and far planes

The near and far planes determine where the camera's view begins and ends.

- The **near plane** is the closest point the camera can see. The default setting is 0.1. Objects before this point aren't drawn.
- The **far plane**, also known as the draw distance, is the furthest point the camera can see. Objects beyond this point aren't drawn. The default setting is 1000.

Stride renders the area between the near and far planes.



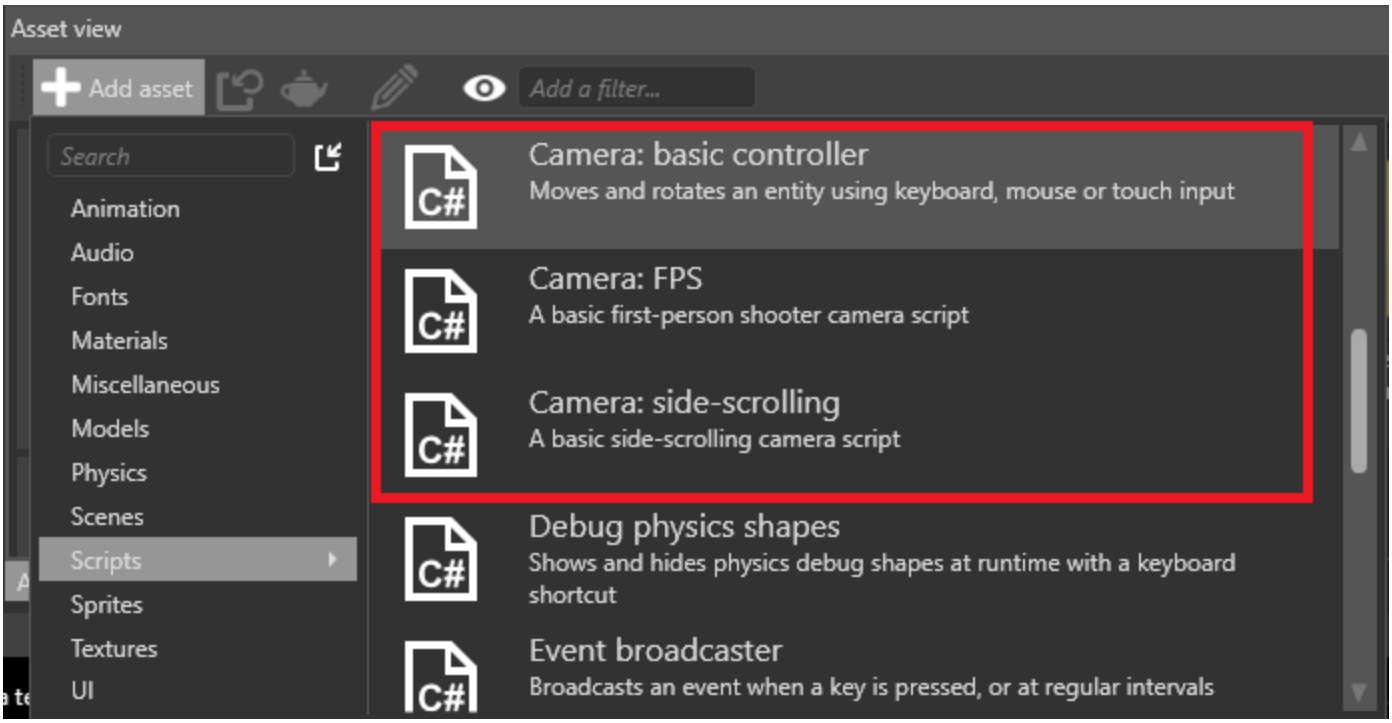
Near plane 0.1 (default); far plane: 50	**Near plane: 7; far plane 1000 (default)**
	
With a low **far plane** value, objects in the near distance aren't drawn.	With a high **near plane** value, objects close to the camera aren't drawn.

Camera scripts

You can control cameras using **camera scripts**. Stride includes three camera script templates: an FPS camera script, a side-scrolling camera script, and a basic camera controller script.

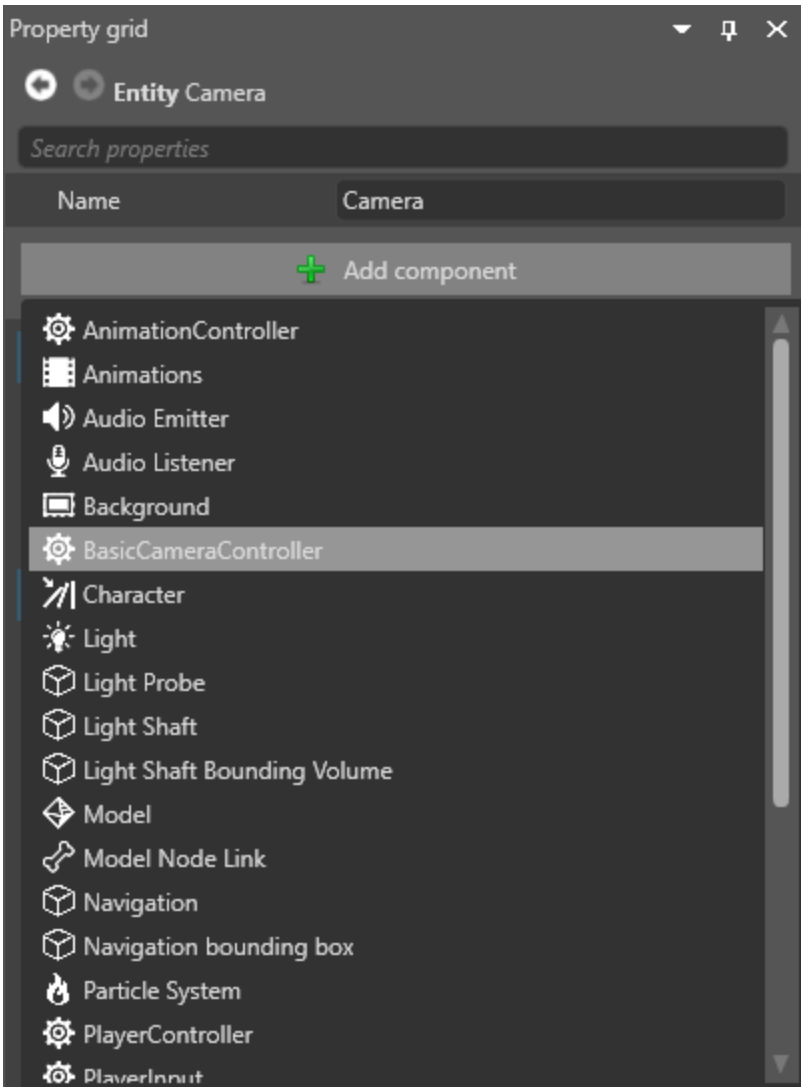
Add a camera script in Game Studio

1. In the **Asset View** (in the bottom by default), click **Add asset > Scripts** and choose the camera script you want to add.



2. In the **Scene Editor**, select the entity with the camera you want to control.

3. In the **Property Grid** (on the right by default), click **Add component** and select the camera script you want to use.



Game Studio adds the camera script to the entity.

For more information about how to create and use scripts, see Scripts.

Camera slots

Camera slots link the graphics compositor to the cameras in your scene. You bind each camera to a slot, then define which slot the compositor uses. This means you can change the root scene or graphics compositor without having to assign new cameras each time.

For more information, see Camera slots.

Render a camera to a texture

You can send a camera's view to a texture and use the texture on objects in your scene. For example, you can use this to display part of your scene on a TV screen in the same scene, such as security camera footage. For more information, see Render textures.

See also

- Camera slots
- Animate a camera
- Graphics compositor

Camera slots

Camera slots link the graphics compositor to the cameras in your scene. You bind each camera to a slot, then define which slot the compositor uses. This means you can change the root scene or graphics compositor without having to assign new cameras each time.

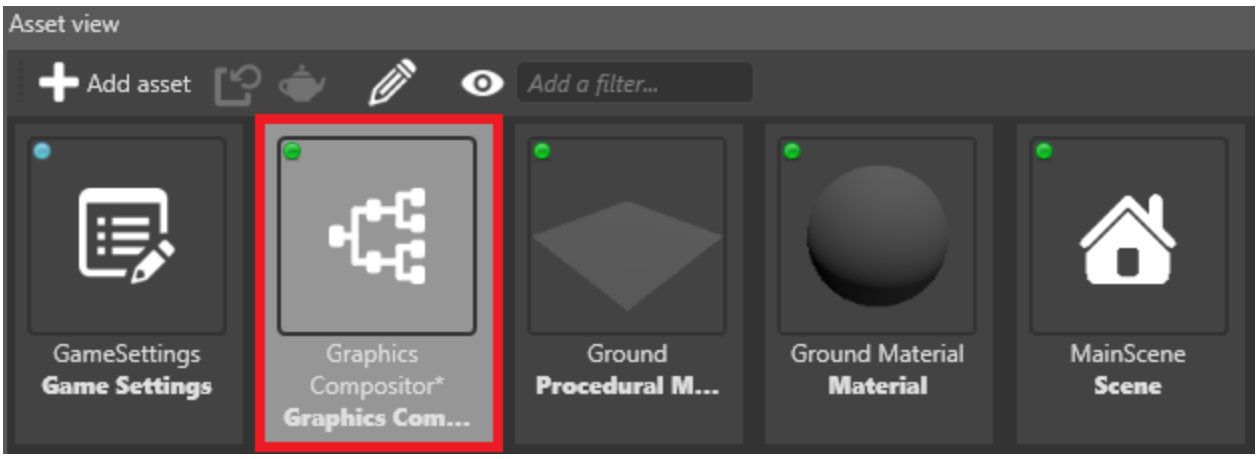
You don't have to create a different camera slot for each camera. Instead, you can just change which cameras use each slot. The best practice is to disable the camera components on cameras you don't need.

> **NOTE**
>
> Each camera slot must have a camera assigned to it. If you have an unused camera slot, delete it.
>
> You can't assign a single camera to more than one slot. If you need to do this, duplicate the camera entity and assign it to a different slot.

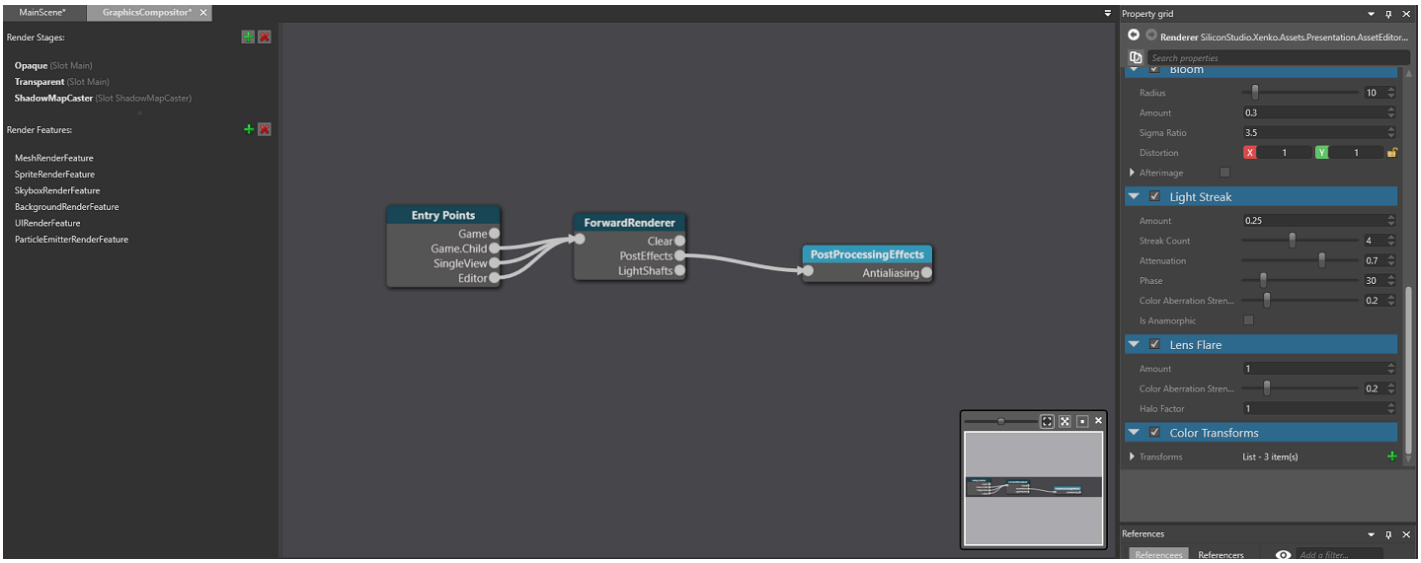
> If multiple enabled cameras in your scene use the same camera slot, the result is undefined.

Create a camera slot

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

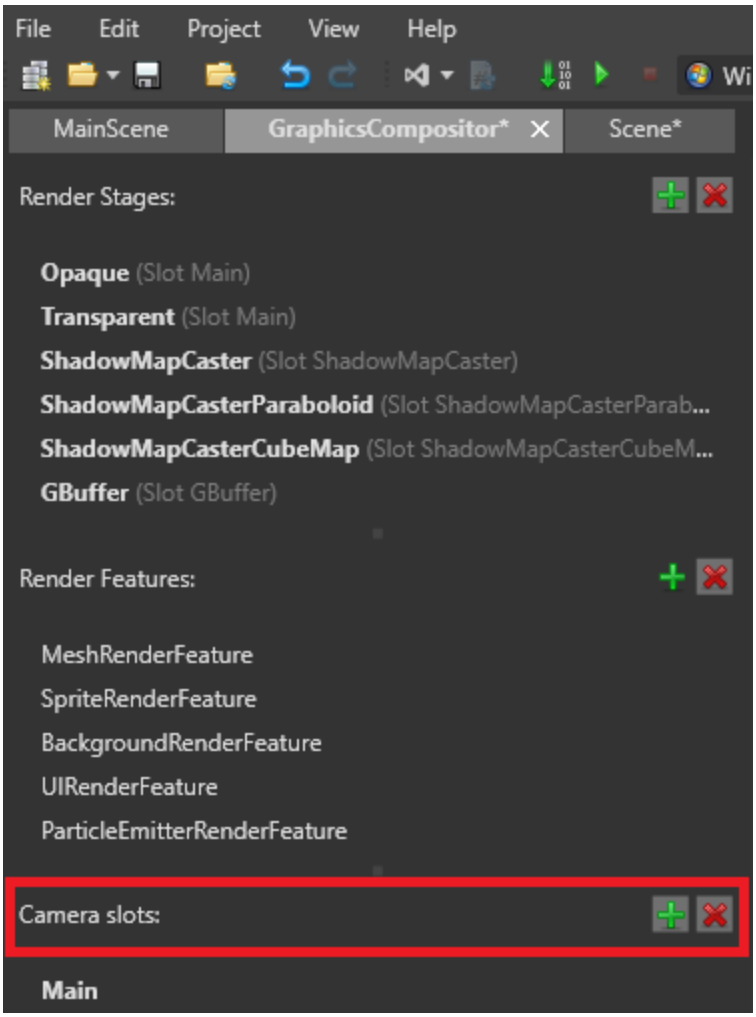


   The graphics compositor editor opens.

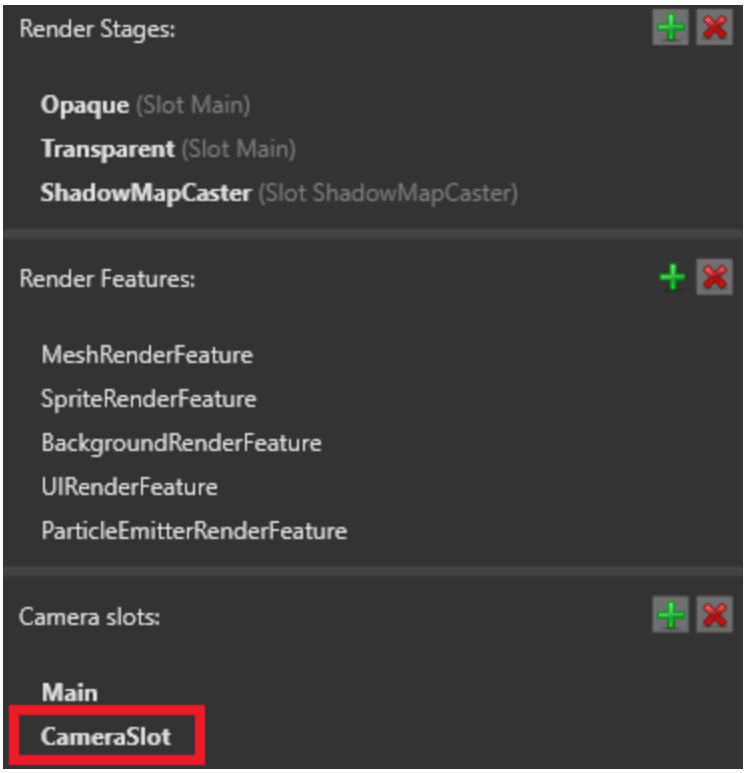


For more information about the graphics compositor, see the Graphics compositor page.

2. In the graphics compositor editor, on the left, under **Camera slots**, click (**Add**).



Game Studio adds a new camera slot to the list:

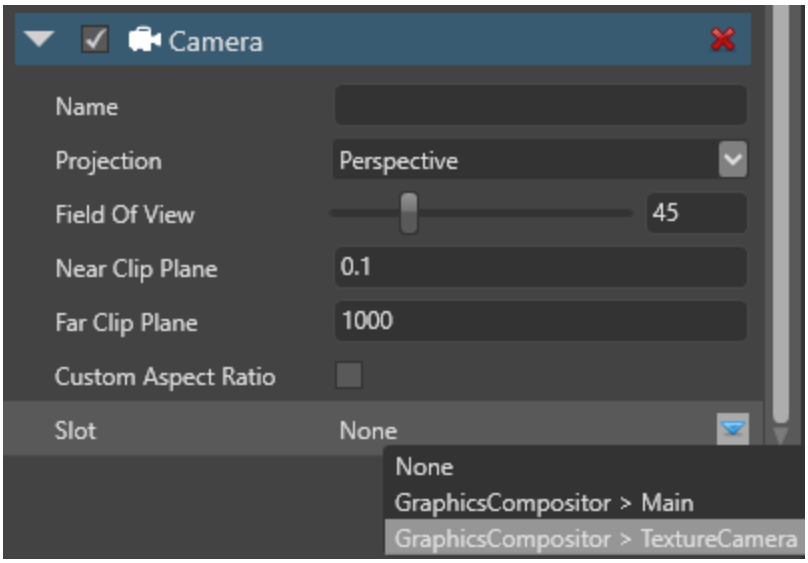


> **TIP**
>
> To name a camera slot, double-click it in the list and type a new name.

Bind a camera to a camera slot

1. In your scene, select the **entity** with the camera component you want to bind.
2. In the **Property Grid** (on the right by default), in the **Camera** component properties, under **Slot**, select the slot you want to bind the camera to.

> **NOTE**
>
> The drop-down menu lists camera slots from the graphics compositor selected in the game settings.



The graphics compositor matches enabled cameras to their appropriate slots each frame.

Create a camera and assign a camera slot from a script

Use:

```
var camera = new CameraComponent();
 camera.Slot = SceneSystem.GraphicsCompositor.Cameras[0].ToSlotId();
```

To change the camera at runtime, toggle the `Enabled` property.

> **NOTE**
>
> Make sure you:
>
> - always have at least one enabled camera
> - don't have multiple cameras enabled and assigned to the same slot at the same time

See also

- Cameras
- Graphics compositor
- Game Studio — Game settings
- Game Studio — Manage scenes

Animate a camera with a model file

Beginner Artist

Like other entities, you can animate cameras using animations imported from 3D model files such as `.3ds`, `.fbx`, and `.obj`.

> **NOTE**
> To animate a camera using a model file, you first need to bake the animation using your modeling tool (eg Maya, 3ds Max or Blender). Stride doesn't support cameras animated using target cameras.

If the camera moves independently, the simplest method is to export the camera animation as a separate file, enable the **root motion** option on the animation, then add the camera, animation, and animation script to the same entity. If the animations include FOV or near or far plane animations, the Stride camera updates accordingly. With this method, you don't need a model or a skeleton.

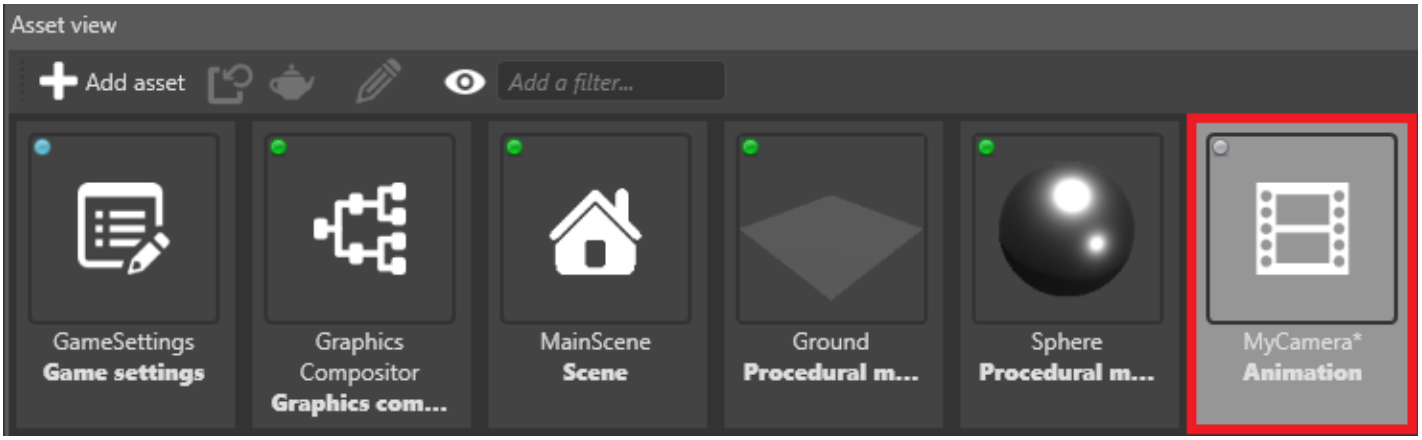
If you want the camera to move in tandem with another animation — for example, if the camera is held by a cameraman character with its own model, skeleton and animation — use a model node link component to link the camera entity to the cameraman's movements.

Animate a camera independently

To do this, you need the following assets in your project:

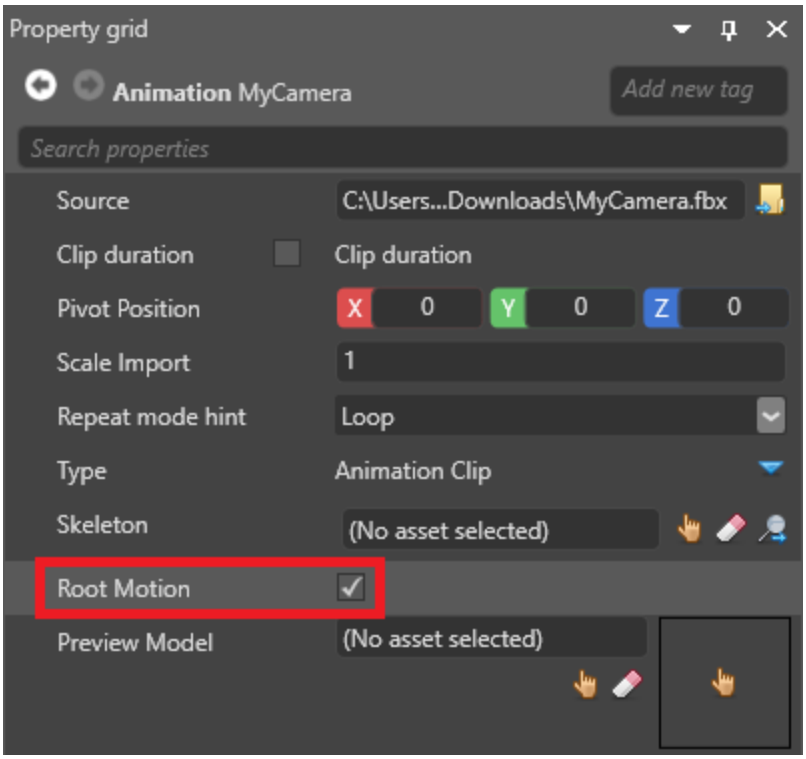
- a camera entity, the camera to be animated
- an animation, to animate the camera (exported separately in your modeling tool)
- an animation script, to play the animation

1. In the **Asset View**, select the animation asset you want to use to animate the camera.



> **NOTE**
>
> For instructions about how import animations, see Import animations.

2. In the **Property Grid**, enable **Root motion**.



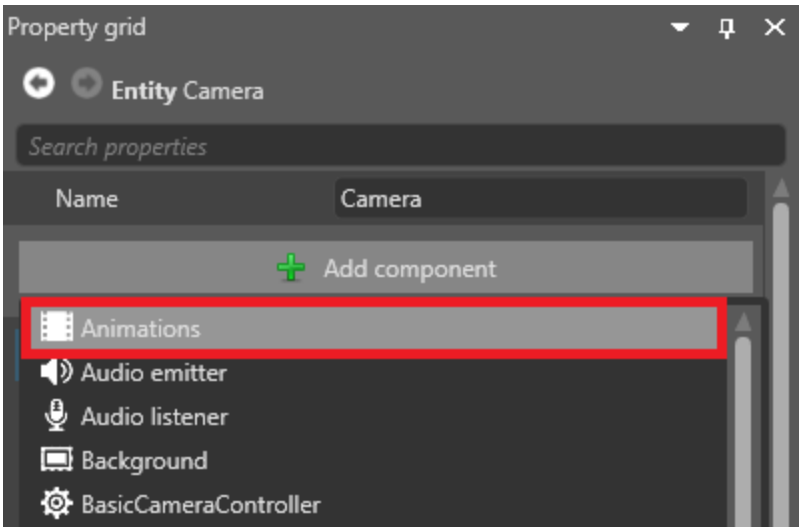
When root motion is enabled, Stride applies the **root node animation** to the TransformComponent of the entity you add the animation to, instead of applying it to the skeleton.

> **NOTE**
>
> If there is no skeleton specified in **Skeleton**, Stride always applies the animation to Transform Component, even if **root motion** is disabled.

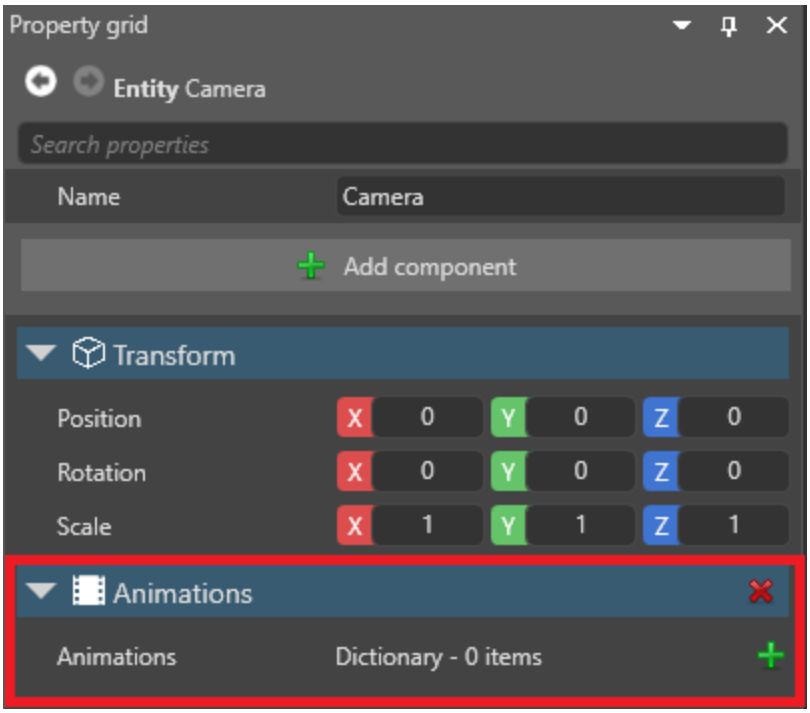
3. In the **Scene Editor**, select the entity that contains the camera you want to animate.

> **NOTE**
>
> For instructions about how add cameras, see Cameras.

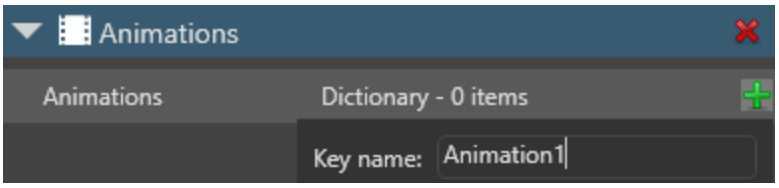
4. In the **Property Grid**, click **Add component** and select **Animations**.



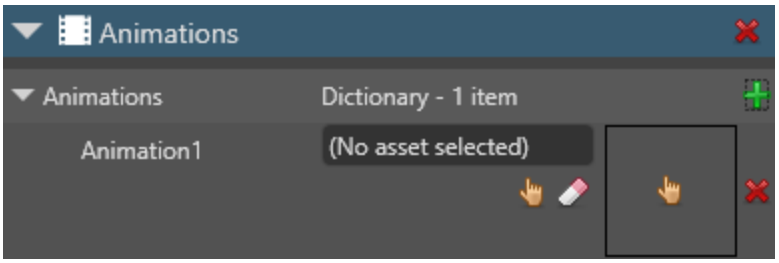
Game Studio adds an animation component to the entity.



5. Next to **Animations**, click (**Add**) and type a name.

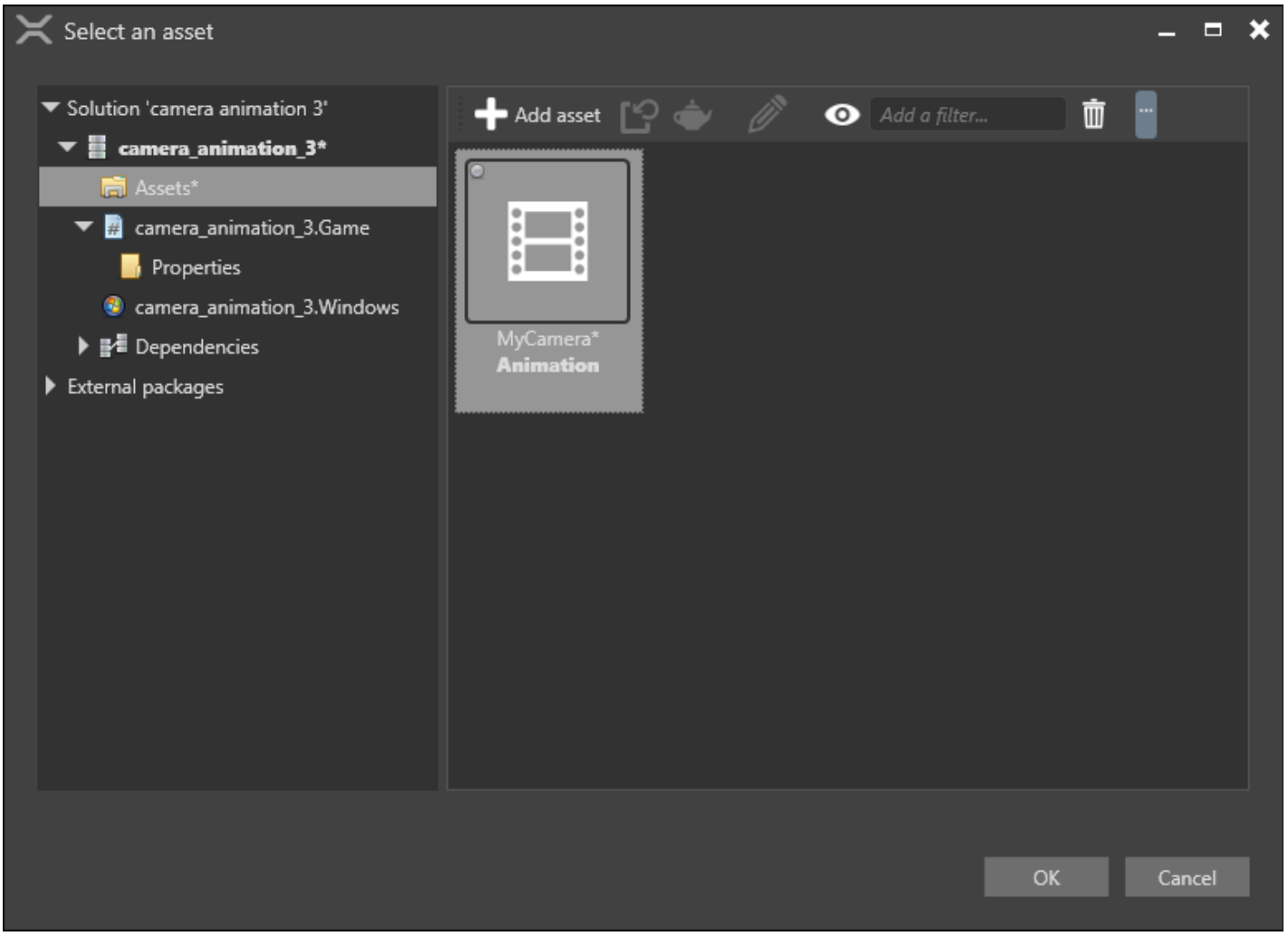


Game Studio adds an animation to the list.



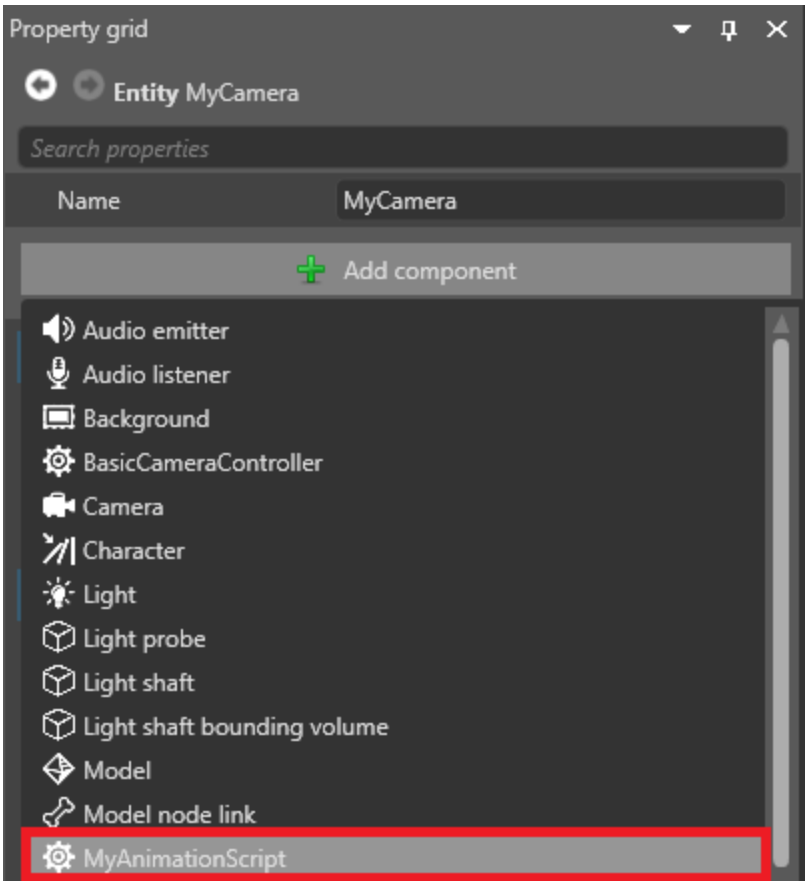
6. Next to the animation you added, click (**Select an asset**).

   The **Select an asset** window opens.



7. Select the animation you want to use to animate the camera and click **OK**.

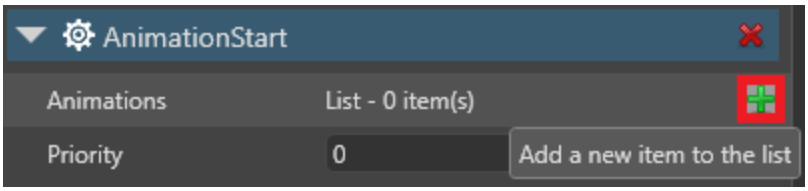
8. Click **Add component** and select the animation script you want to use to animate the camera.



Game Studio adds the script to the entity as a component.

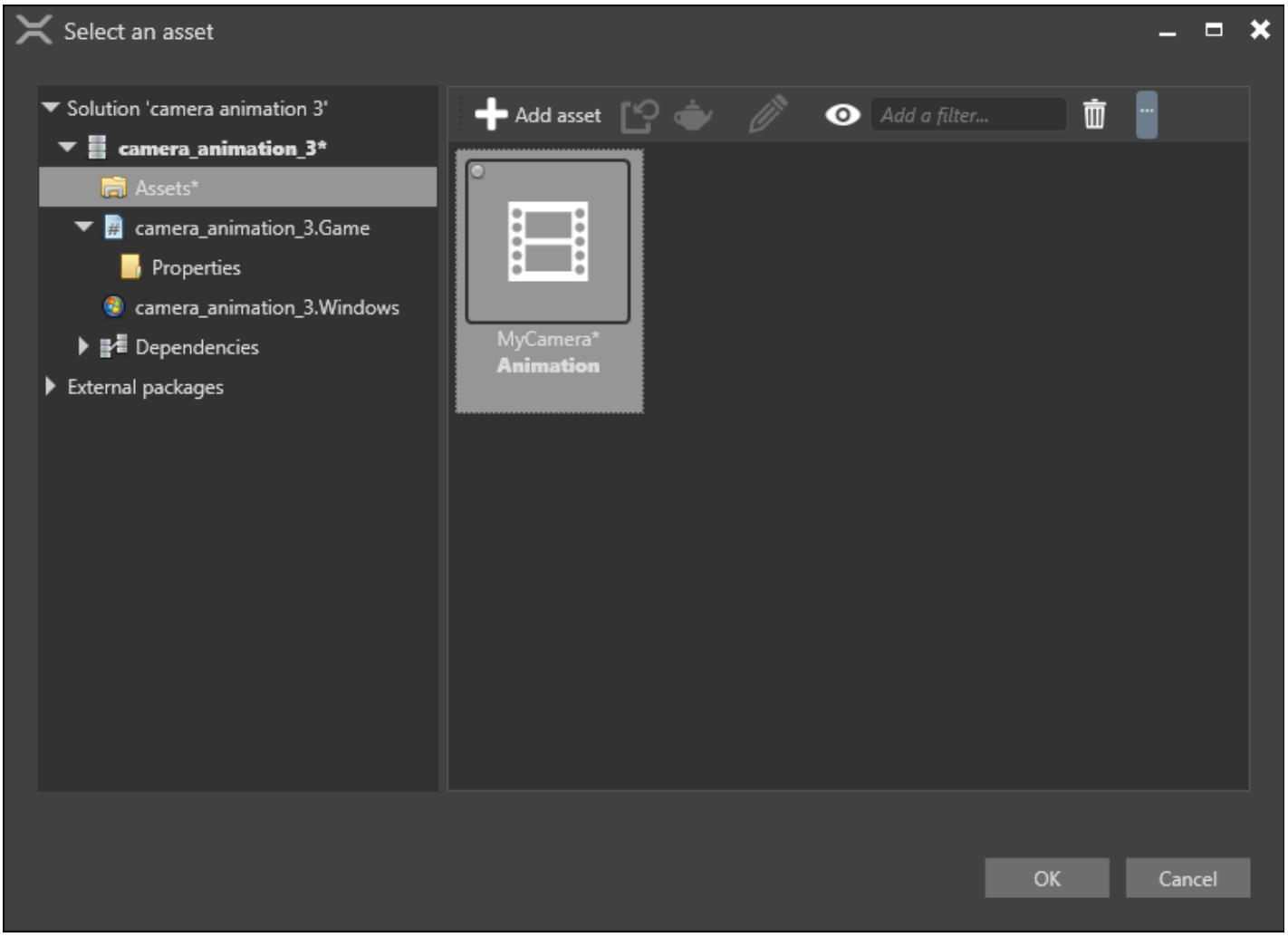
> **NOTE**
>
> For instructions about how to add animation scripts, see Animation scripts.

9. Under the script component, next to **Animations**, click (**Add**).



10. Next to **Clip**, click (**Select an asset**).

    The **Select an asset** window opens.



11. Select the animation asset you want to use to animate the camera and click **OK**.

At runtime, the camera uses the animation. If the animation includes FOV or near or far plane animations, the Stride camera updates accordingly.

Attach the camera to a node on another model

To move a camera in tandem with another model, create a separate entity for the camera, then use a **model node link** component to link the entity to the correct node.

To do this, you need the following assets in your project:

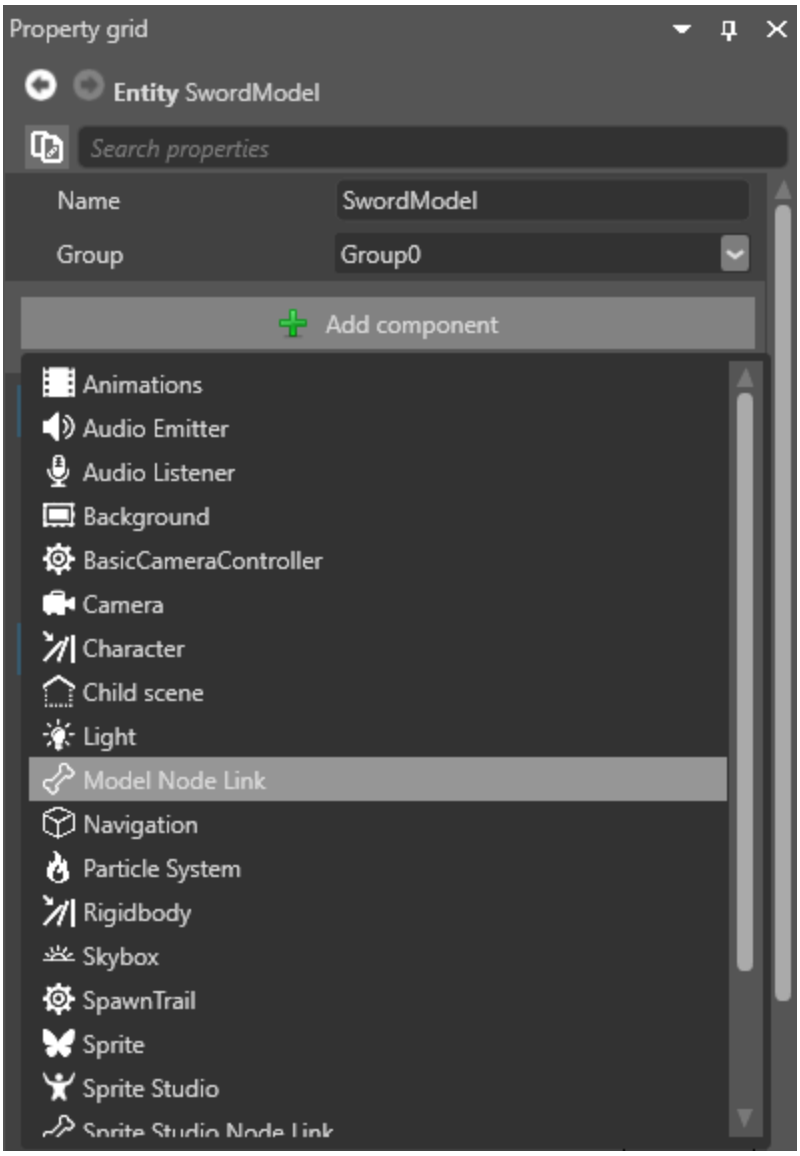
- a camera entity, the camera you want to animate
- a model, to attach the camera to
- a skeleton that matches the model
- an animation, to animate the model
- an animation script, to play the animation

> **NOTE**
>
> FOV and near or far plane animations are ignored if you use this method.

1. In the **Asset View**, select the model you want to link the camera to. Next to **Skeleton**, make sure a skeleton is specified that matches the model.
2. Make sure the entity you want to attach the camera to has the model, animation clip, and animation script components needed to animate it.

   > **NOTE**
   >
   > For instructions about how to add these, see Animation.

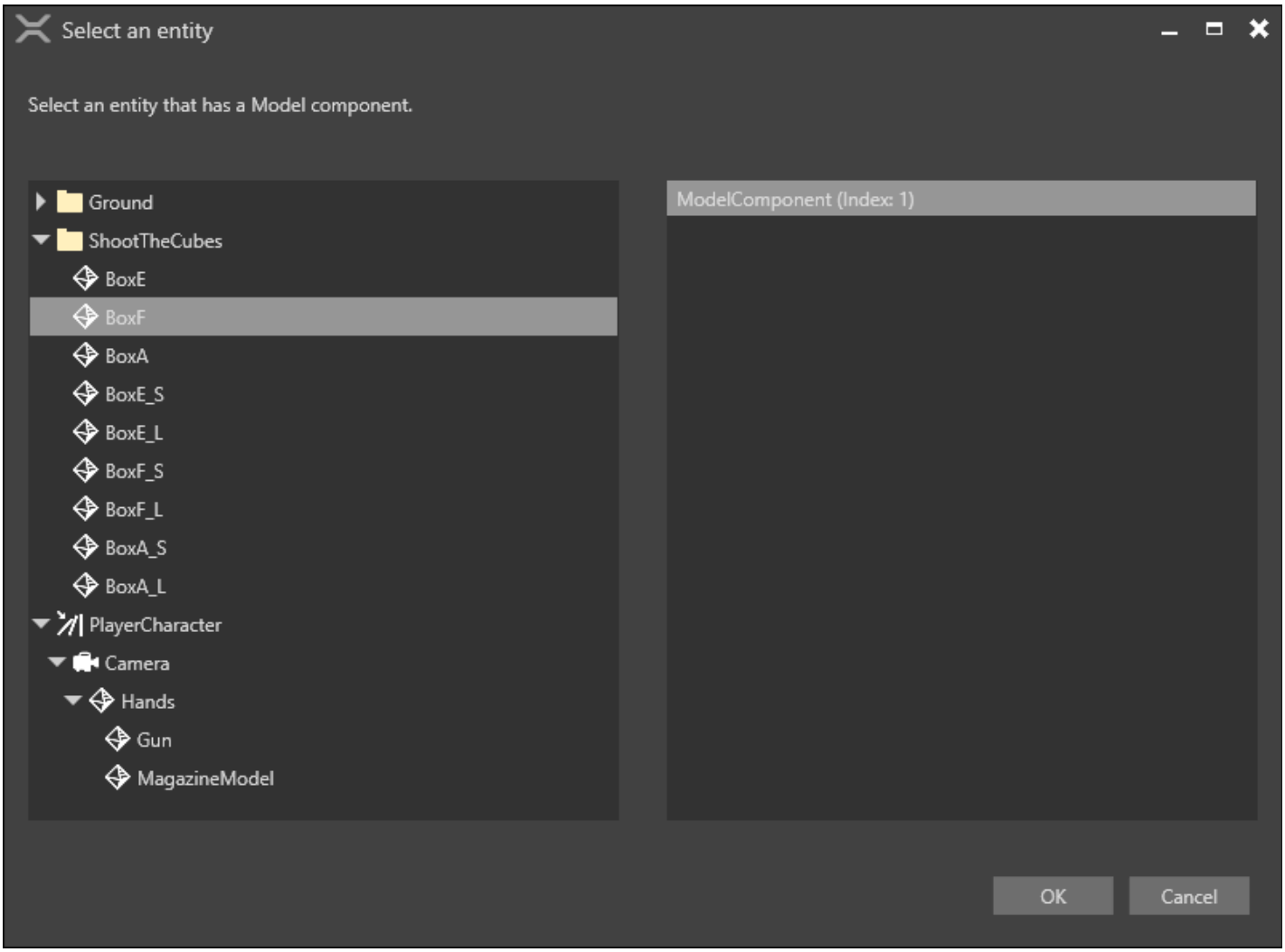
3. With the camera entity selected, in the **Property Grid**, click **Add component** and select **Model node link**.



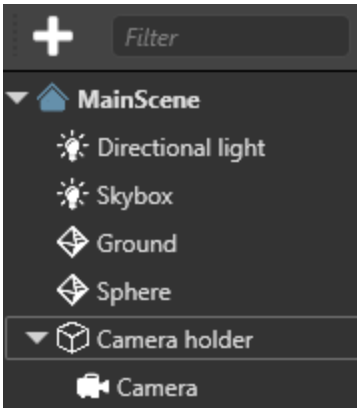
> **NOTE**
>
> The TransformComponent applies an offset to the model node position. If you don't want to add an offset, make sure the TransformComponent is set to `0,0,0`.

Game Studio adds a model link component to the entity.

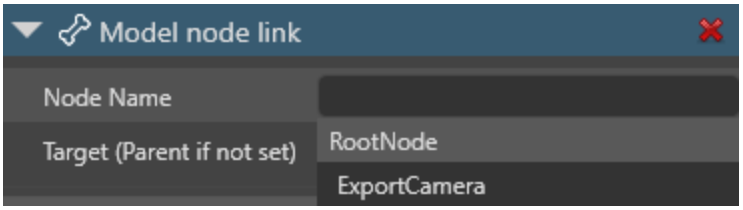
4. Next to **Target**, click and select the entity that has the model you want to link the camera to.



Alternatively, leave the **Target** field blank. In the **Entity Tree**, drag the **camera entity** you want to animate to the entity that contains the model. Stride links the entity to the model on the parent entity.



5. In **Node name**, select the node you want to link the camera to.



> **NOTE**
>
> The entity you link to must have a model with a skeleton, even if the model isn't visible at runtime.

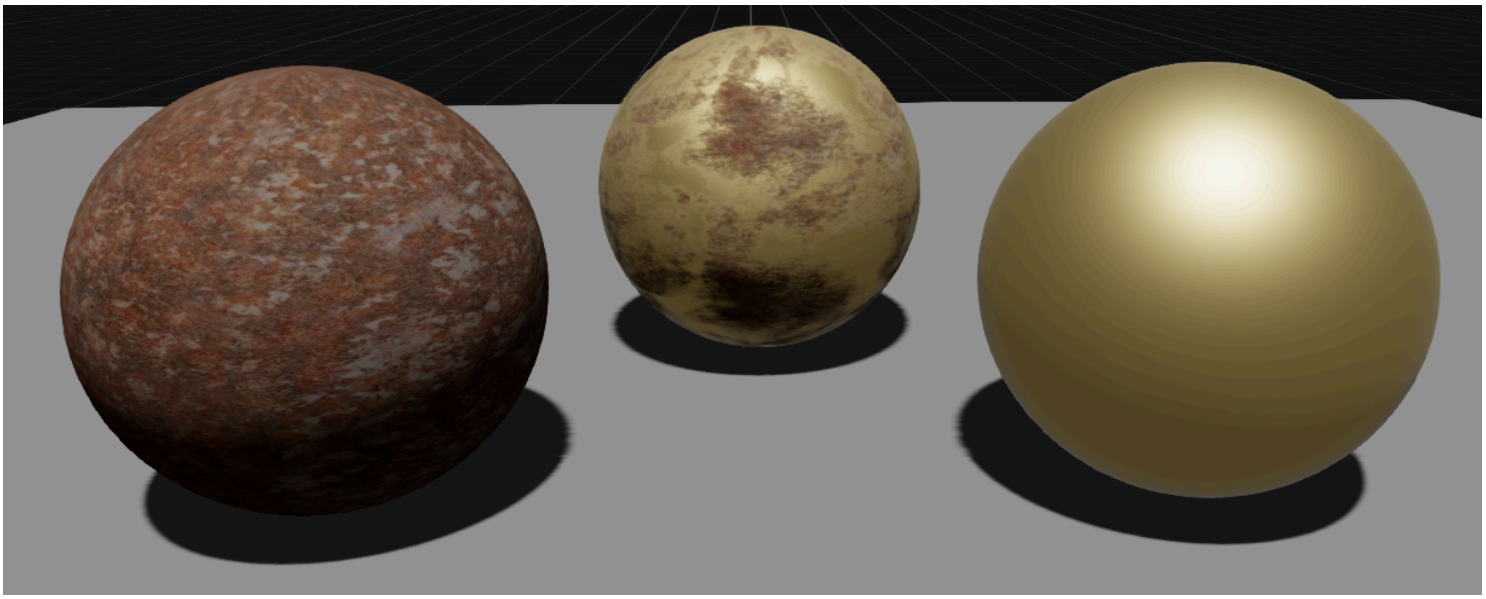
At runtime, the camera uses the animation.

See also

- Cameras
- Model node links
- Animation
- Animation scripts

Materials

Materials define the appearance of 3D model surfaces and how they react to light. Without materials, models are simply shapes, blank canvases.



Materials can affect both the geometry of a model (vertex shading) and its colors (pixel shading).

You can use multiple material layers to build more complex materials.

In practice, materials generate partial definitions of shaders integrated as part of the shading of models (lights and shadows).

In this section

- Material maps
- Material attributes
  - Geometry attributes
  - Shading attributes
  - Misc attributes
    - Clear-coating shading
- Material layers
- Material slots
- Materials for developers

Material maps

Intermediate Artist Programmer

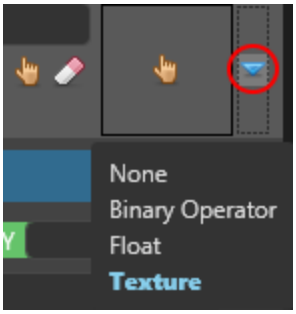
Material maps calculate how materials are rendered. They can use two kinds of values: color (RGB) values or scalar (single float) values.

You can use material maps for several purposes, including gloss maps, diffuse maps, or blend maps (for combining material layers)

Material maps can fetch values using one of several providers:

- **Vertex stream**: a value taken from mesh attributes
- **Binary operator**: a combination of any other two providers
- **Float4 / Float**: a constant value
- **Color**: a hex color value
- **Shader**: a value provided by a ComputeColor shader. This lets you use procedural values
- **Texture**: a value sampled from a texture

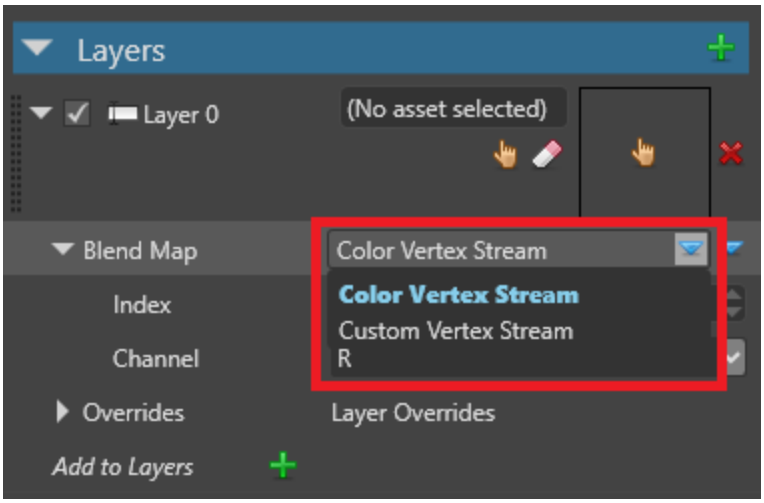
To choose the provider, click (**Replace**) and select it from the drop-down menu:



Vertex stream

This provider takes a value from an attribute of the mesh of the model you apply the material to.

It has two modes: **Color Vertex Stream** and **Custom Vertex Stream**. To switch between them, with **Vertex Stream** selected as the provider, click (**Replace**) and choose the mode you want to use.



Color vertex stream

Takes a color value from the mesh.

Property	Description
Index	The index in the named stream
Channel	The channel (RGBA) to sample from the stream

Custom vertex stream

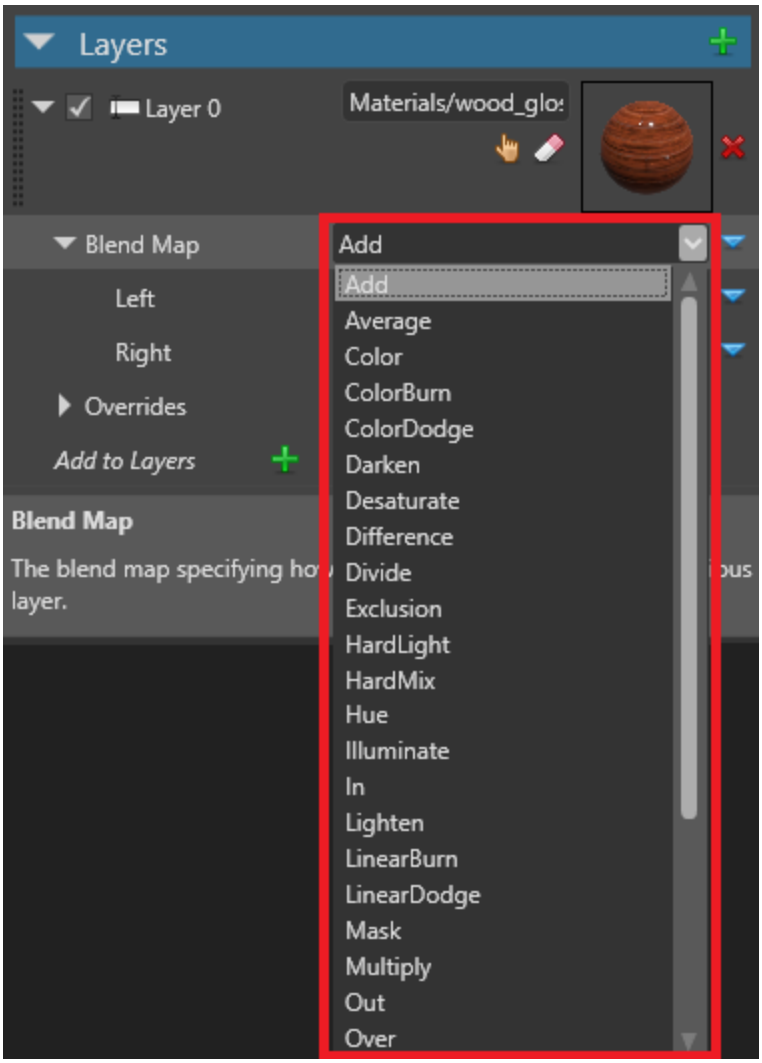
Takes a value from the mesh channel you specify.

Property	Description
Name	Semantic name of the channel to read data from
Channel	The channel (RGBA) to sample from the stream

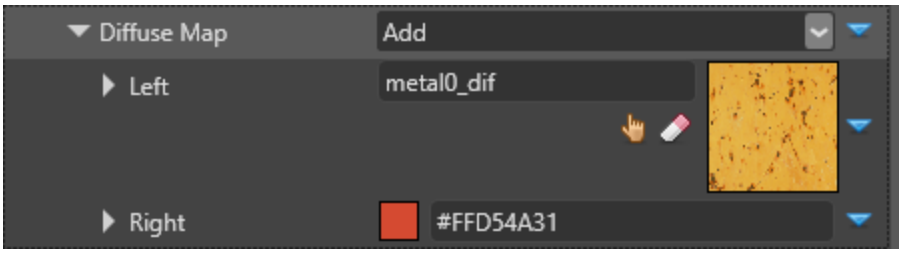
Binary operator

Perform a binary operation from two color/scalar value providers. You can nest as many material maps inside binary operators as you need (including further binary operators).

To choose how the operation works, click (**Replace**) and select from the drop-down menu. The operations are similar to options when blending layers in Photoshop.




```
Result = LeftColor <operator> RightColor
```

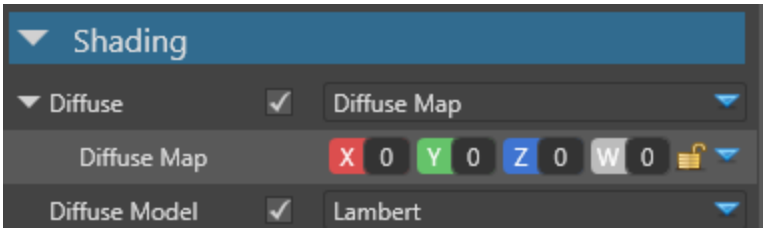


Property	Description
Operator	A binary operator (eg add, multiply, etc)
Left	The left color/scalar used in the operation
Right	The right color/scalar used in the operation

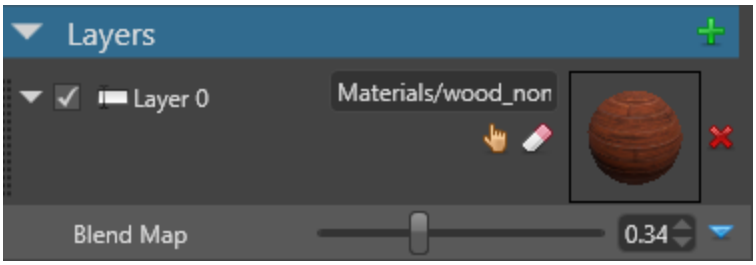
Float4 / Float

Provided directly as a constant value over the whole material.

In the case of RGB values, you control the RGBA value with the X, Y, Z and W values (*Float4*).

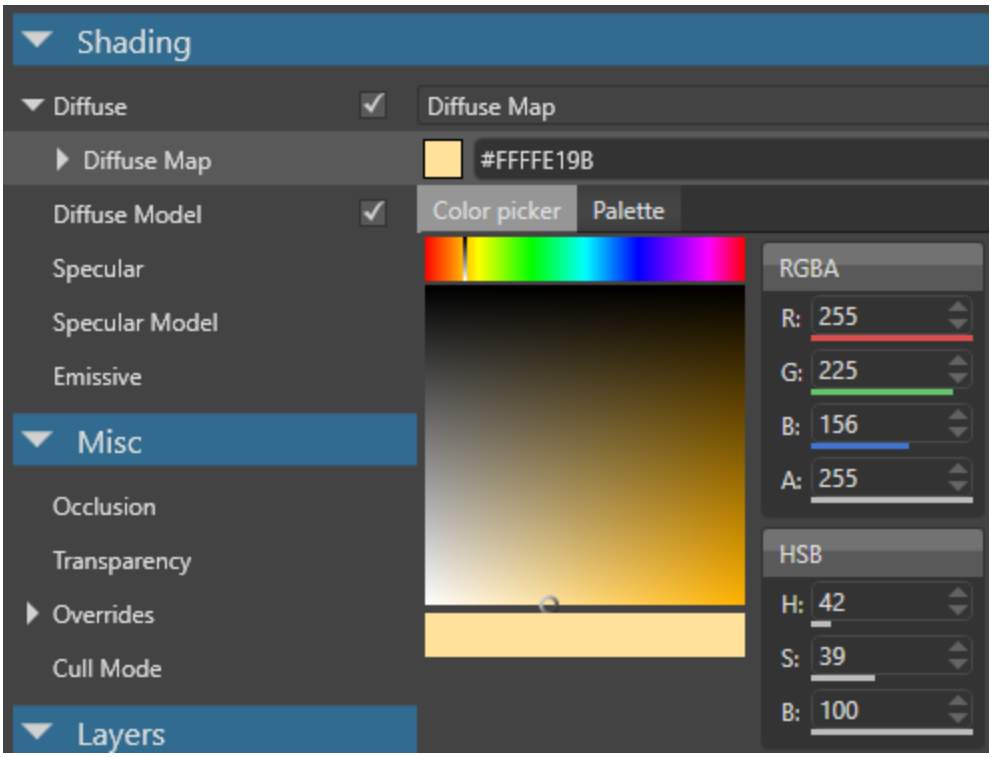


In the case of scalar values, you control the value with a slider (*Float*).



Color

A value provided from a color hex value. This is only available for material maps that use RGB values.



Shader

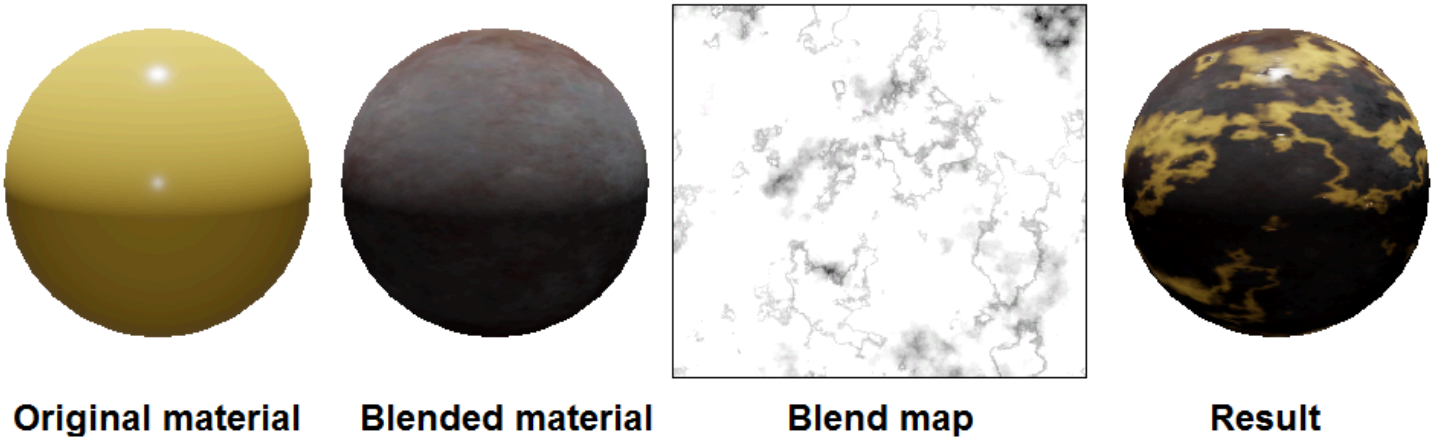
A value provided by a ComputeColor shader. This lets you use procedural values.

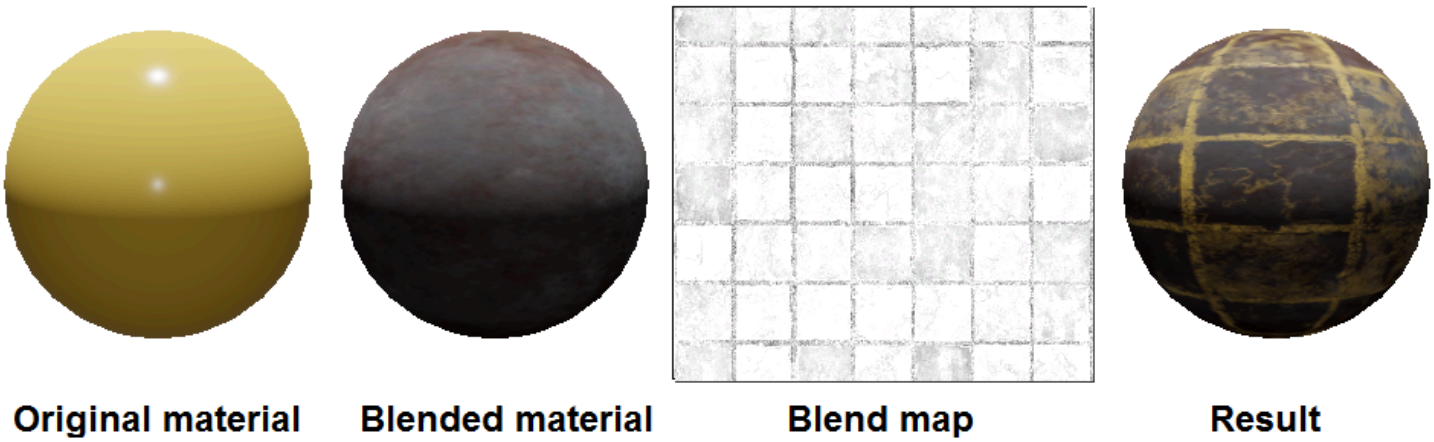
For an example of a ComputeColor shader, see the Particle materials tutorial.

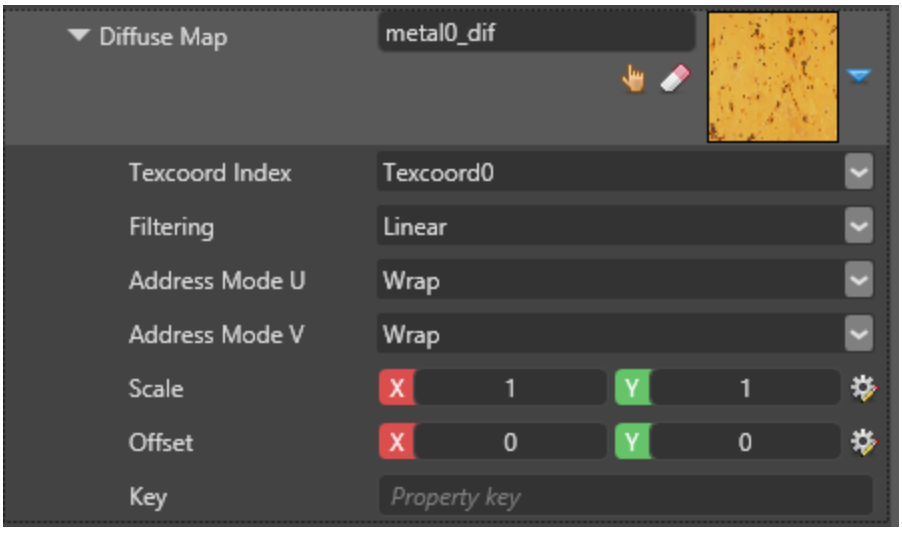
Texture

Sample the color/scalar from a texture.

For example, the images below demonstrate how the texture changes the way Stride blends materials.







Property	Description
Texture	A reference to a texture

Property	Description
Channel	The channel (R, G, B, A) used to extract the scalar value. Only valid for scalar textures
Texcoord Index	The texture coordinates (u,v) to use from the mesh with this texture
Filtering	The sampling method (eg Linear, Point, Anisotropic, etc)
Address Mode U / V	Defines how (u,v) coordinates are addressed **Wrap**: Tiles (u,v) at integer junctions. For example, if u ranges from 0.0 to 3.0, the texture repeats three times on the U axis **Mirror**: Flips (u,v) at integer junctions. For example, if u ranges from 0.0 to 1.0, the texture is displayed as expected; but from 1.0 to 2.0, the texture is mirrored **Clamp**: Clamps (u,v) to the range (0.0, 1.0)
Scale	A scale applied to (u,v)
Offset	An offset applied to (u,v)

See also

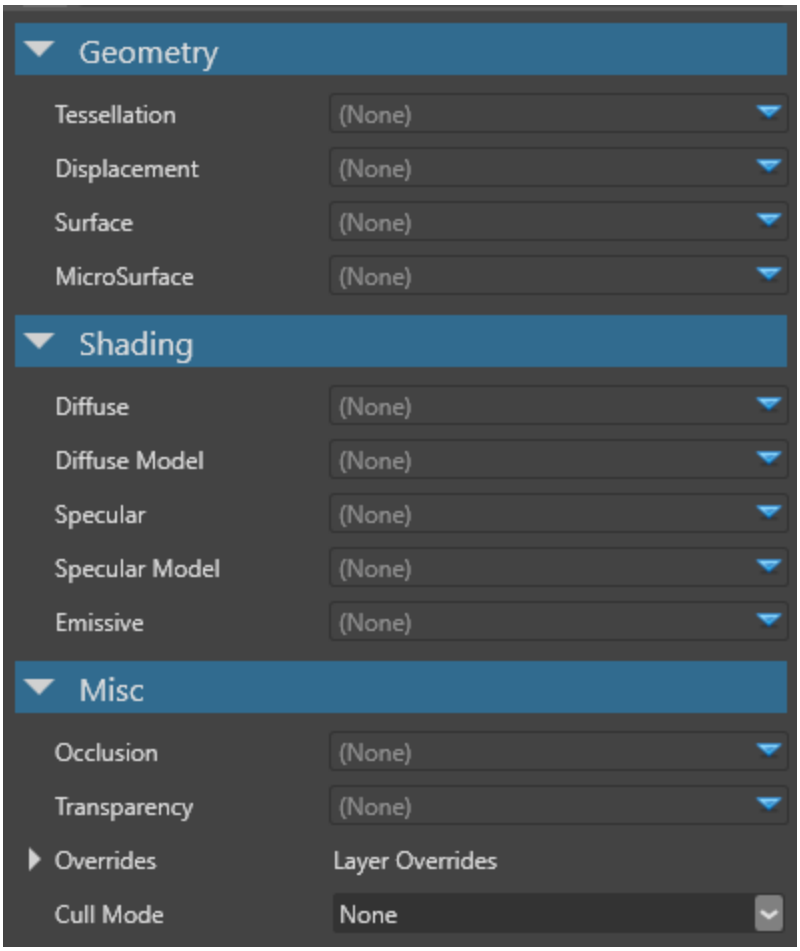
- Material attributes
- Material layers

- Material slots
- Materials for developers

Material attributes

Intermediate Artist Programmer

Material attributes define the core characteristics of a material, such as its diffuse color, diffuse shading model, and so on. Attributes are organized into **geometry**, **shading**, and **misc**.



There are two types of attribute:

- attributes used as input values for a shading model (for example, the **Diffuse** attribute provides only color used by the diffuse shading model)
- attributes that can change the shading model (for example, diffuse shading models, such as the lambert model, interprets the diffuse attribute color)

Attributes contribute to a layer of a material. If a material is directly used as a model material, all its root attributes are considered part of the first layer.

You can also write custom shaders to use in material attributes.

In this section

- Geometry attributes

- Shading attributes
- Misc attributes
    - Clear coat shading

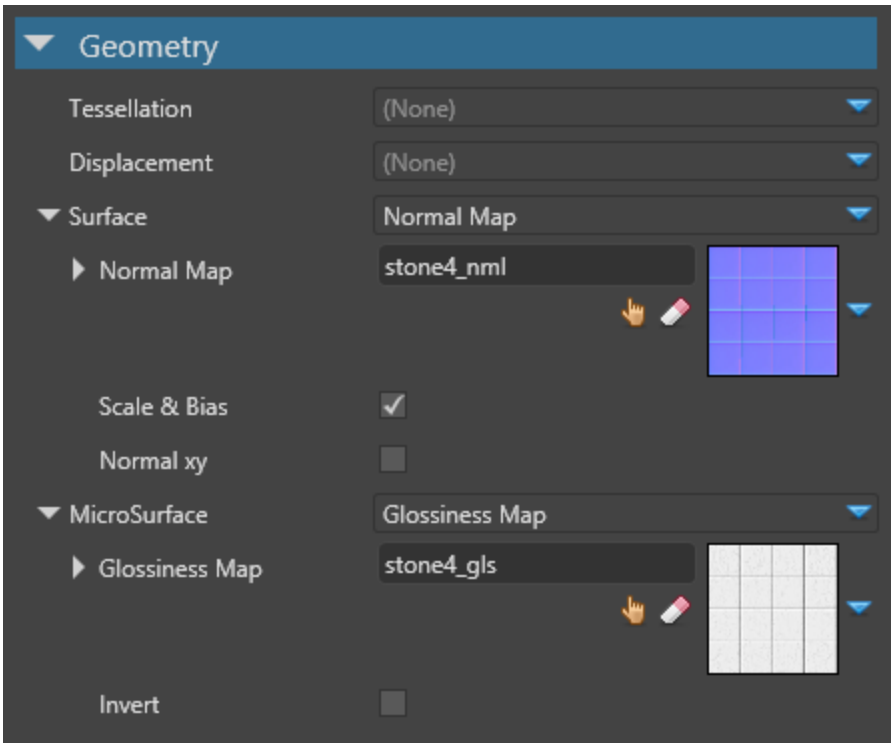
See also

- Material maps
- Material layers
- Material slots
- Materials for developers
- Custom shaders

Geometry attributes

Intermediate Artist Programmer

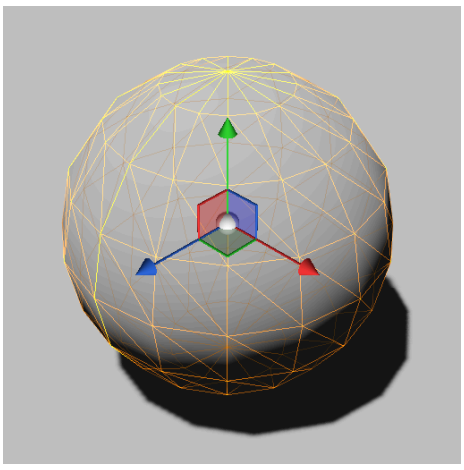
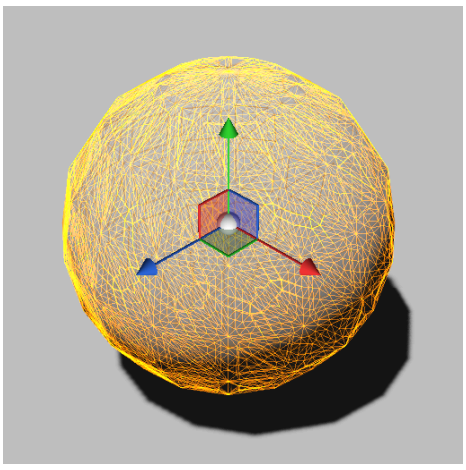
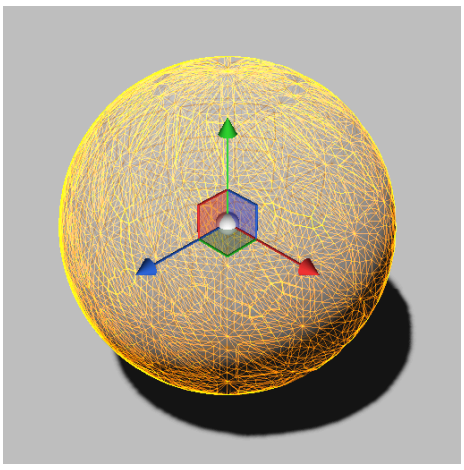
The material **geometry** attributes define the shape of a material.



Tessellation

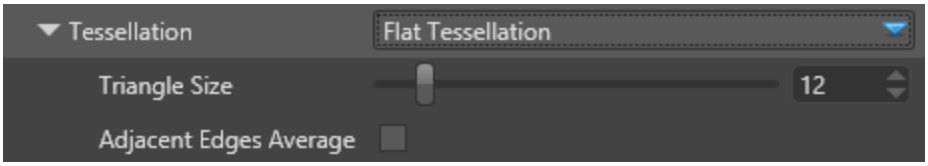
Real-time tessellation uses a HW feature of the GPU to massively subdivide triangles. This increases the realism and potential of deformations of the surface geometry.

You can choose **none**, **flat tessellation**, or **point normal tessellation**.

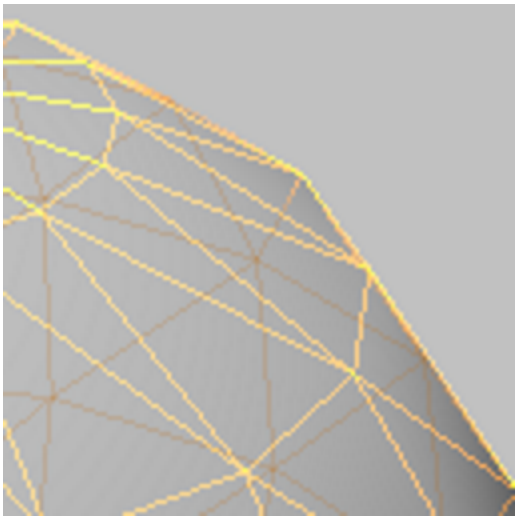
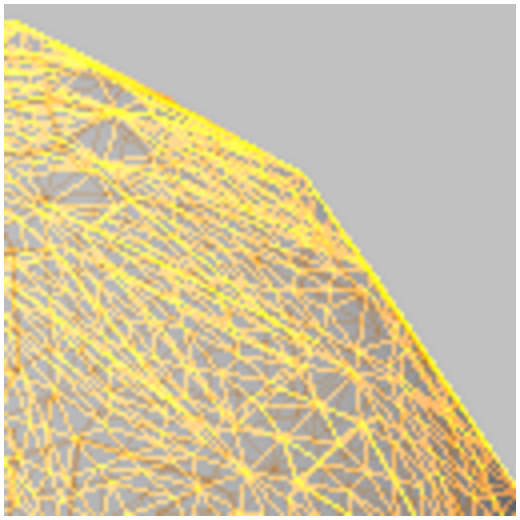
No tessellation	Flat tessellation	Point normal tessellation
		

Flat tessellation

This option tessellates the mesh uniformly.



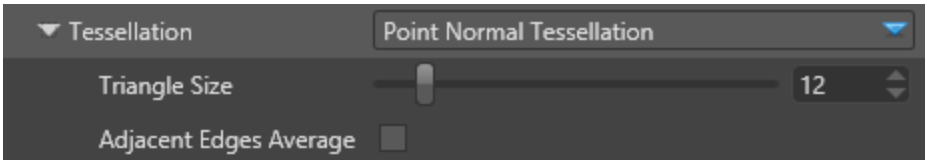
In the images below, notice how the flat tessellation adds extra triangles, but doesn't take the curve into account:

No tessellation	**Flat tessellation**
	

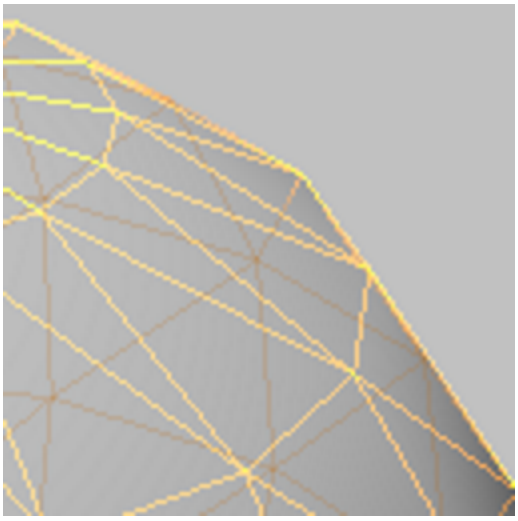
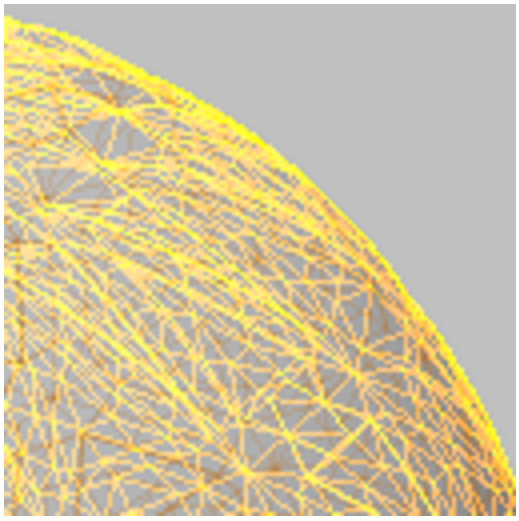
Property	**Description**
Triangle size	The size of a tessellated triangle in screen-space units
Adjacent edges average	Adjust the triangle size values from the average of adjacent edges values

Point normal tessellation

This option tessellates the mesh using the curvature provided by the mesh normals.



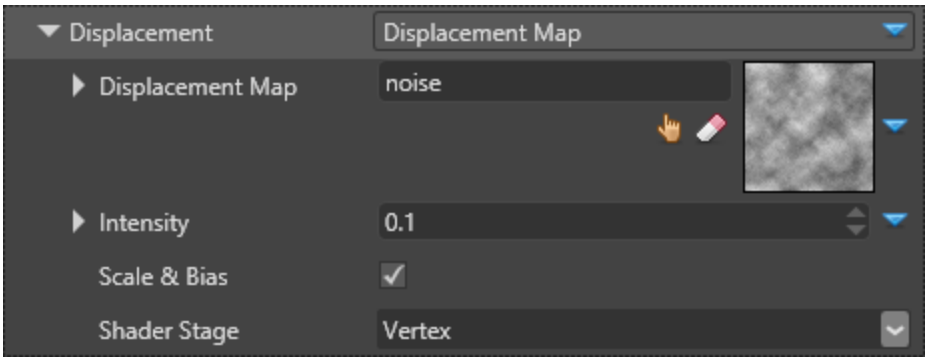
The images below show how point normal tessellation adds extra triangles while taking the curvature of the mesh into account:

No tessellation	Point normal tessellation
	

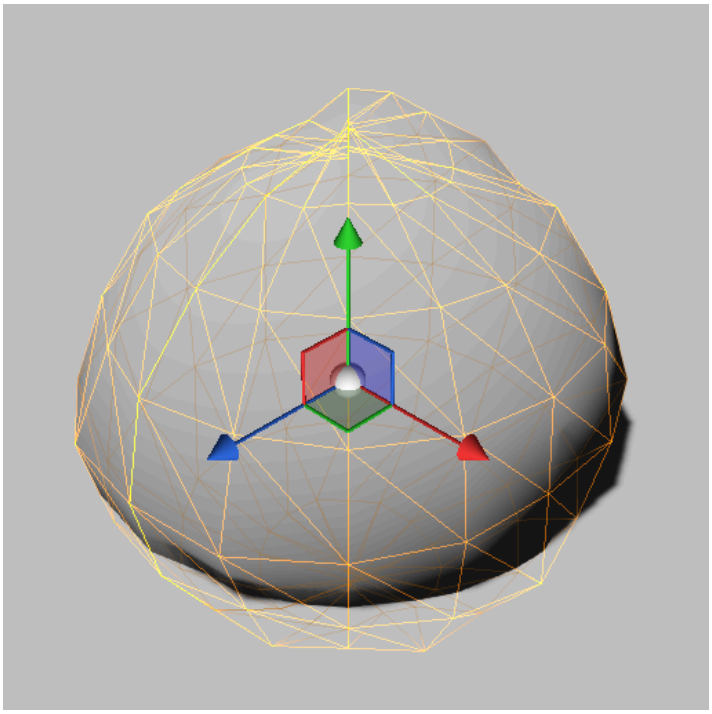
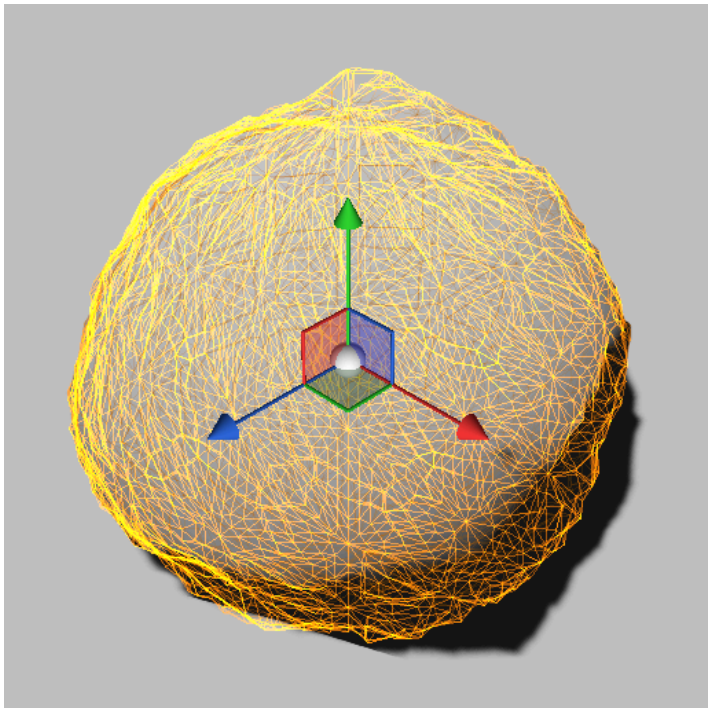
Property	Description
Triangle size	The size of a tessellated triangle in screen-space units
Adjacent edge average	Adjust the **triangle size** and **normal curvature** values from the average of adjacent edge values

Displacement

Under the **Displacement** properties, you can specify **displacement map**. This displaces the geometry of the mesh.

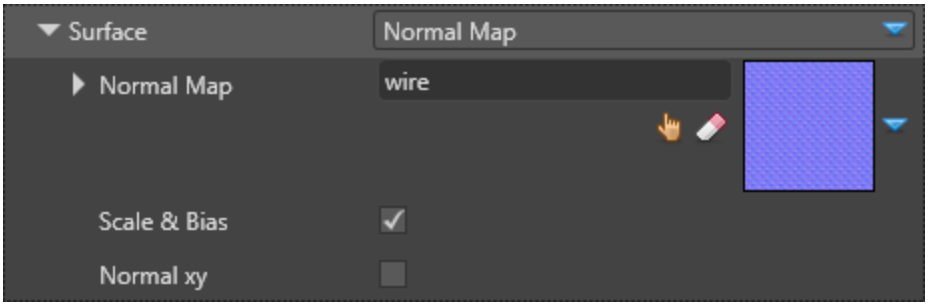


Depending on the stage at which the displacement is applied, the results can be very different:

Displacement with vertex shader	Tessellation with displacement
	

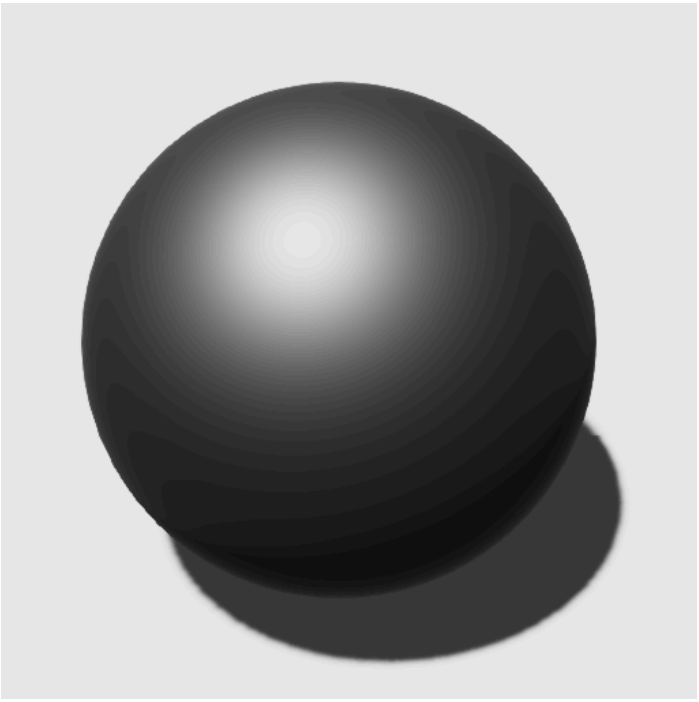
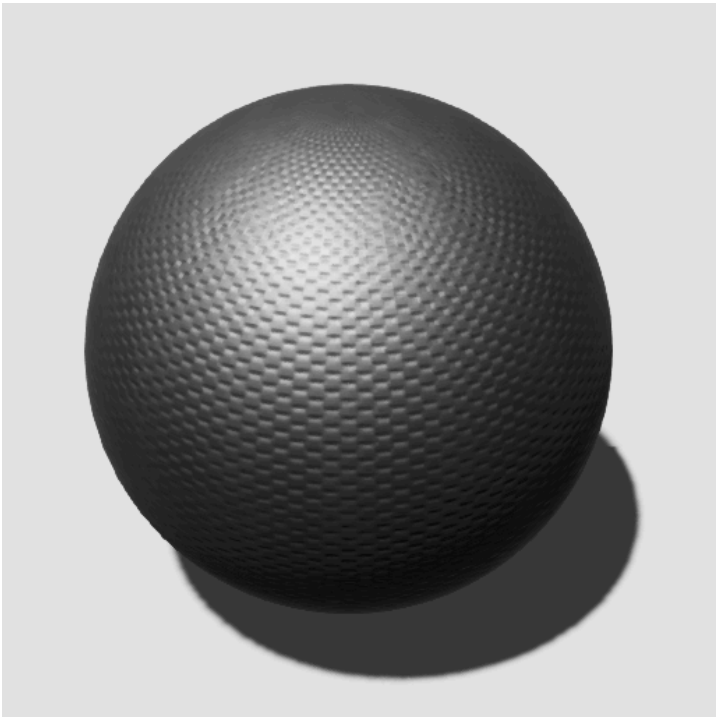
Property	Description
Displacement Map	The displacement texture as a material color provider
Intensity	The amount of displacement
Scale & Bias	When enabled, the value coming from the texture is considered a positive value ranging from `0.0` to `1.0` and the shader applies a scale to get the range -1.0 to `1.0`
Shader Stage	Specify which shader stage the displacement map should be applied to: vertex shader or domain shader (used with tessellation)

Surface



Under the **Surface** properties, you can define a Normal maps to define **macro** surface normals. The **normal map** provides per-pixel normal perturbation of the normal of the mesh. Normal maps create the

appearance of bumps and indents in the mesh:

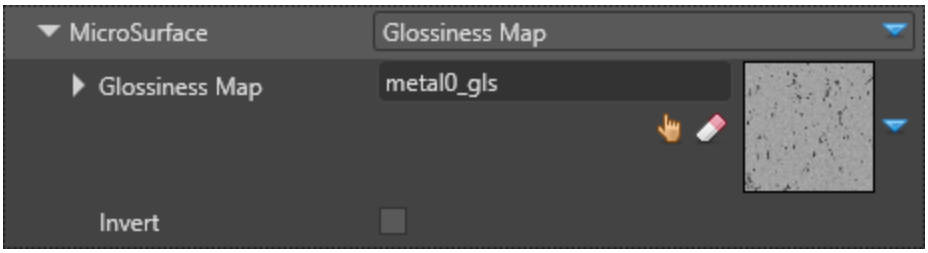
Flat	Using a normal map
	

Property	Description
Normal map	The normal map color provider
Scale and offset	Interpret values from the texture as positive values ranging from `0.0` to `1.0`. The shader applies a scale to get the range `-1.0` to `1.0`.
Reconstruct Z	If there's no Z component in the texture, reconstruct it from the X and Y components. This assumes that $X^2 + Y^2 + Z^2 = 1$ and that Z is always positive, so no normal vector can point to the back side of the surface. We recommend you enable this option, as Stride might remove the Z component when you compress normal maps.

For more information about normal maps, see the normal maps page.

Micro surface

Under the **Micro surface** setting, you can provide a **gloss map** to provide per-pixel information for gloss.

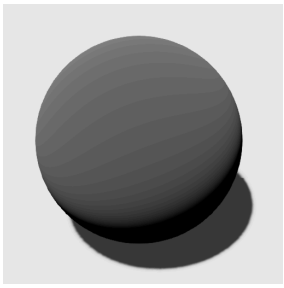
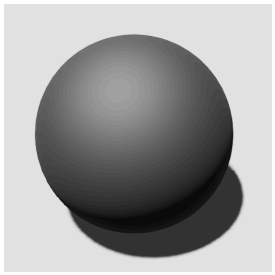
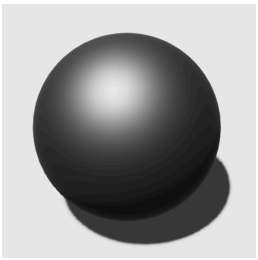
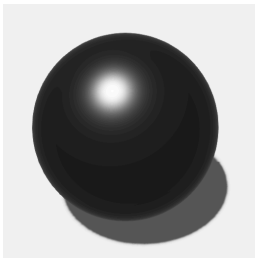
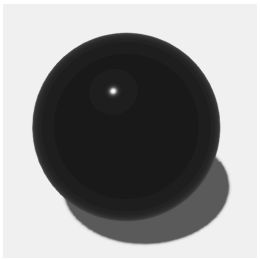


If you select **Float**:

- a value of `1.0` means the surface is highly glossy (the coarse normal isn't perturbed)
- a value of `0.0` means the surface is very rough (the coarse normal is highly perturbed in several directions)

The screenshots below show different levels of gloss on a material:

- Diffuse = #848484, Lambert
- Specular Metalness = 1.0, GGX

Gloss = 0.0	**0.25**	**0.5**	**0.8**	**1.0**
				

Property	**Description**
Gloss map	The gloss map color provider
Invert	Inverts the gloss value (eg a value of `1.0` produces zero gloss instead of maximum). This effectively turns the gloss value into a **roughness** value, as used in other game engines

If you have local reflections enabled, the scene is reflected in materials with a gloss map value higher than the threshold you specify in the local reflections properties. For more information, see Local reflections.

See also

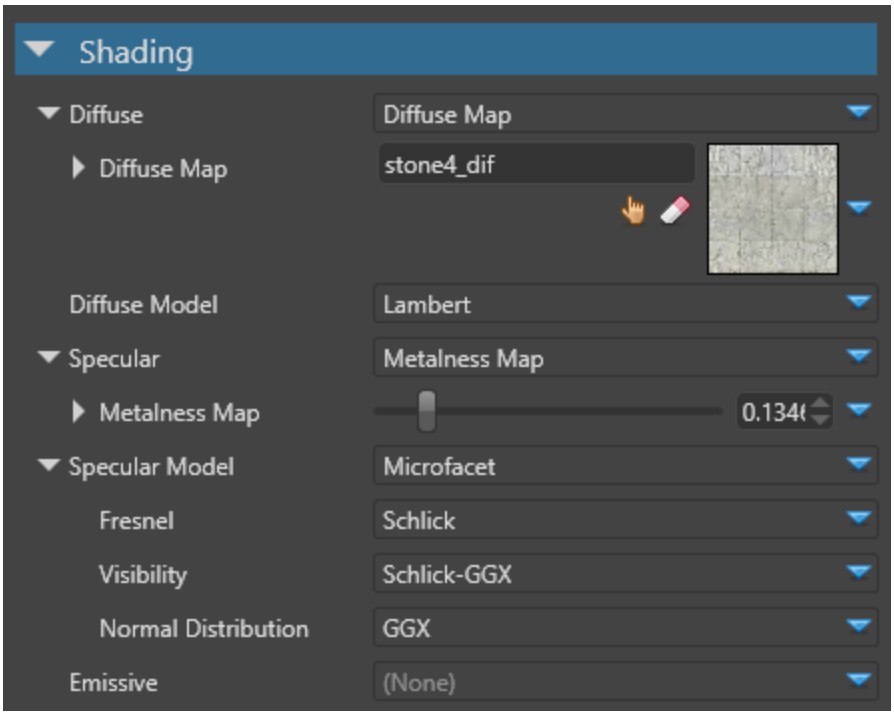
- Material maps
- Material attributes
  - Shading attributes

    - Misc attributes
    - Clear-coat shading
- Clear-coating shading
- Material layers
- Material slots
- Materials for developers
- Custom shaders

Shading attributes

Intermediate Artist Programmer

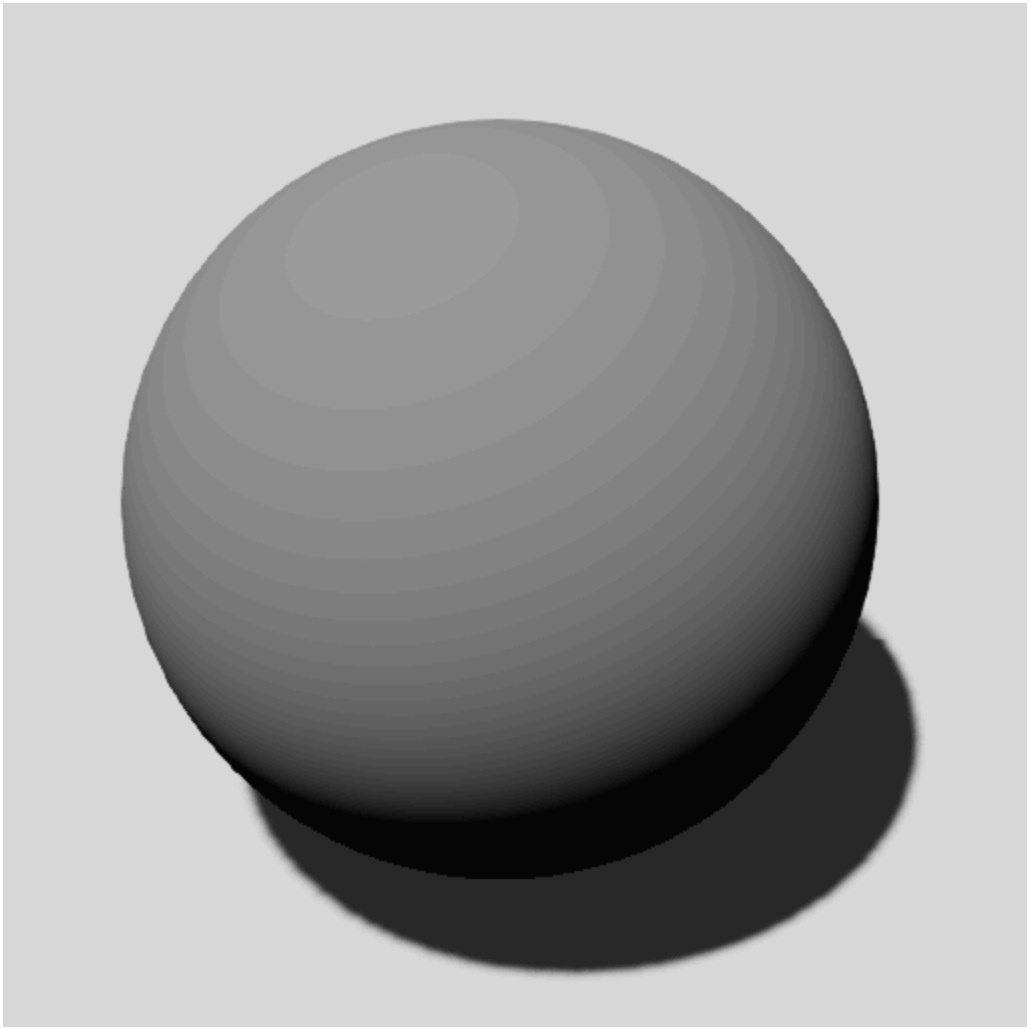
The material **shading attributes** define the color characteristics of the material and how it reacts to light.



> **NOTE**
> To display a material, you need to select at least one shading model (diffuse, specular or emissive model) in the model attributes.

Diffuse

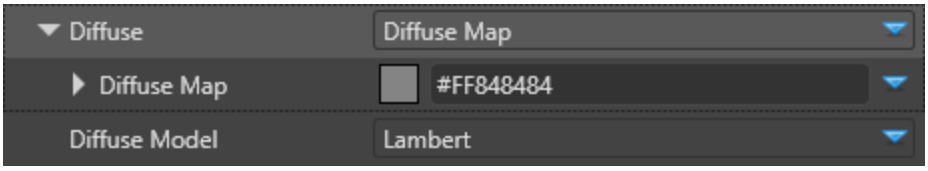
The **diffuse** is the basic color of the material. A pure diffuse material is completely non-reflective and "flat" in appearance.



The final diffuse contribution is calculated like this:

- the **diffuse** defines the color used by the diffuse model
- the **diffuse model** defines which shading model is used for rendering the diffuse component (see below)

Currently, the diffuse attribute supports only a **diffuse map**.

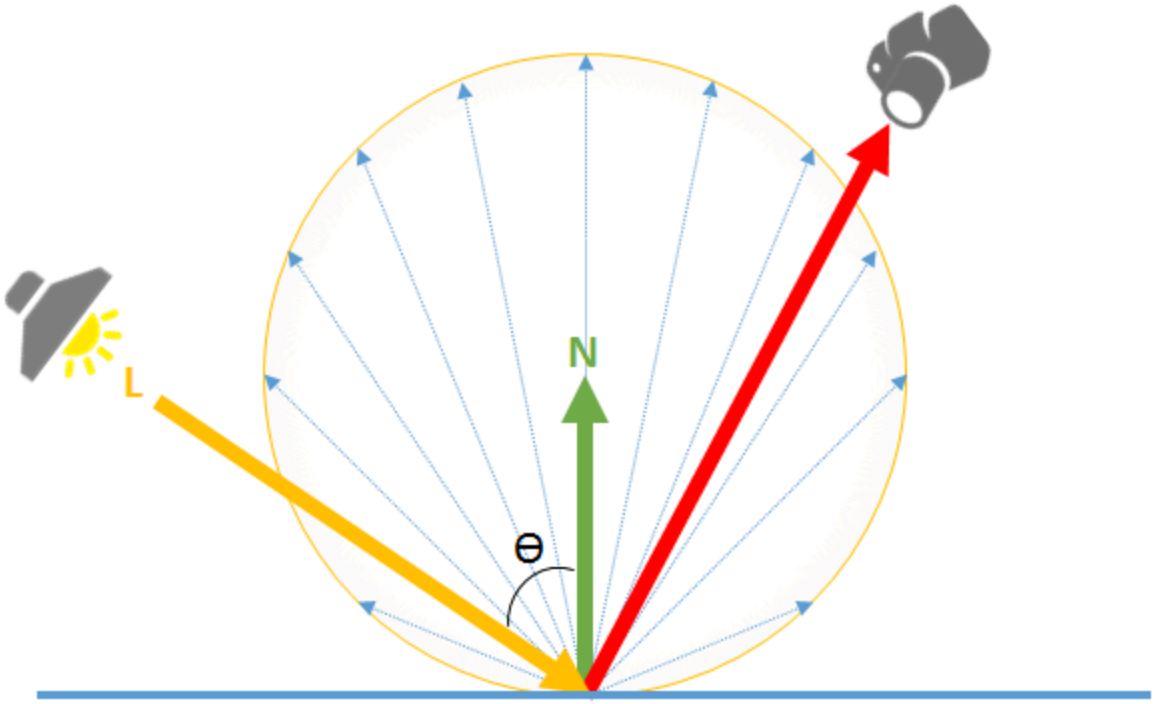


Diffuse model

The **diffuse model** determines how the diffuse material reacts to light. You can use the **Lambert** or **cel-shading**.

Lambert model

Under the Lambert model, light is reflected equally in all directions with an intensity following a cosine angular distribution (angle between the normal and the light):

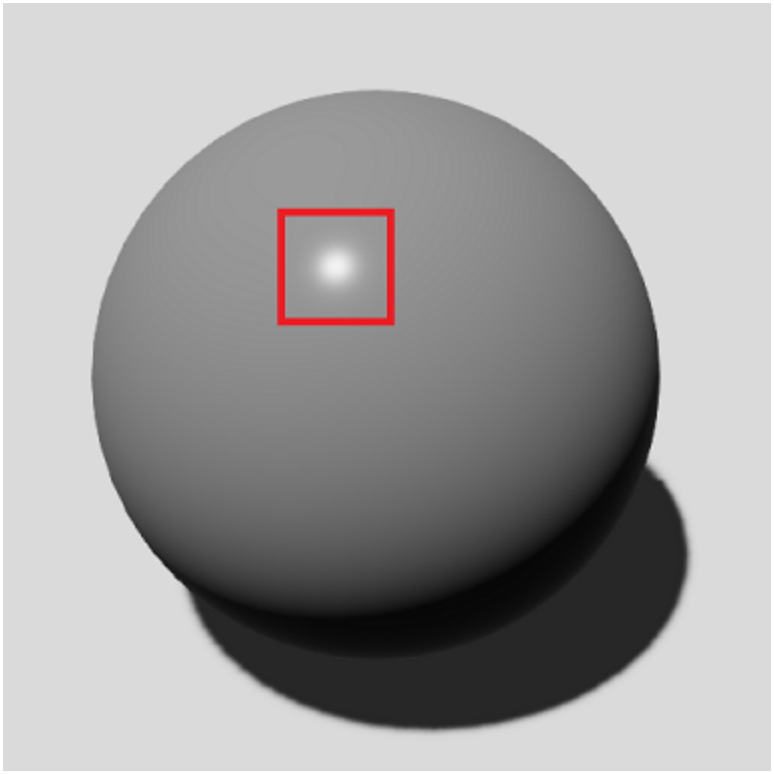


> **NOTE**
>
> A pure Lambertian material doesn't exist in reality. A material always has a little specular reflection. This effect is more visible at grazing angles (a mostly diffuse surface becomes shiny at grazing angle).

Property	Description
Diffuse map	The diffuse map color provider
Diffuse model	The shading model for diffuse lighting

Specular

A **specular** is a point of light reflected in a material.



The specular color can be defined using a metalness map (which uses the diffuse color as a base color), or a specular map (the specular color is defined separately from the diffuse color).

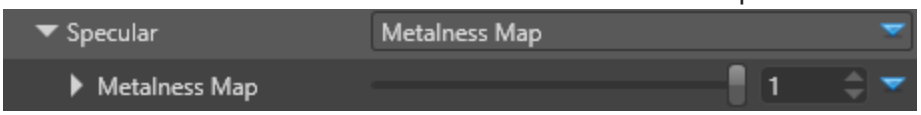
Metalness map

The **metalness map** simplifies parametrization between the diffuse and specular color.

By taking into into account the fact that almost all materials always have some "metalness"/reflectance in them, using the metalness map provides realistic materials with minimal parametrization.

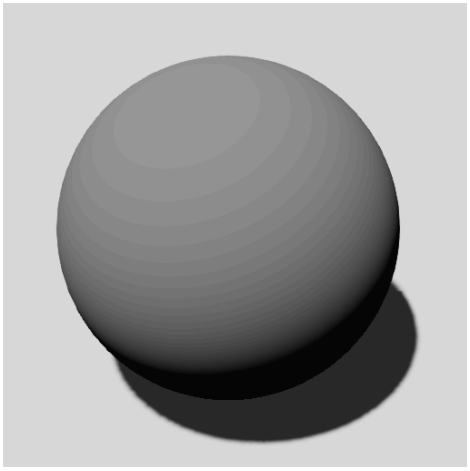
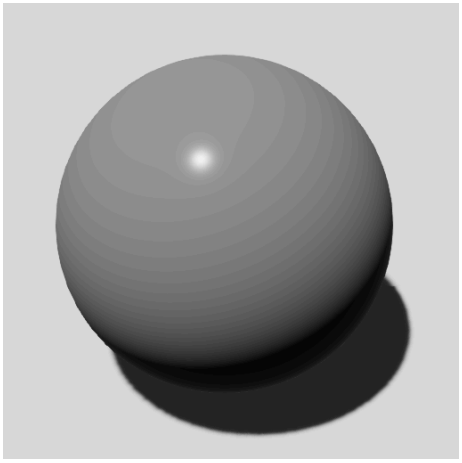
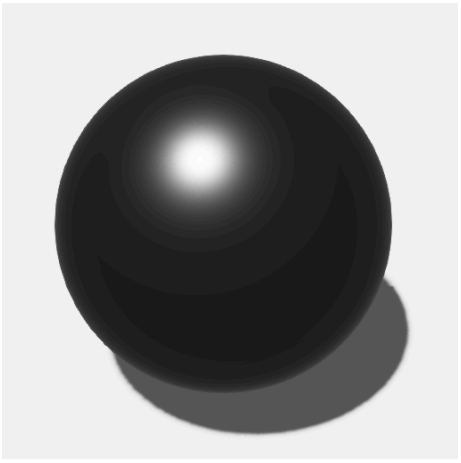
The final specular color is calculated by mixing a fixed low-reflection color and the diffuse color.

- With the metalness color at `0.0`, the effective specular color is equal to `0.02`, while the diffuse color is unchanged. The material is not metal but exhibits some reflectance and is sensitive to the Fresnel effect.
- With the metalness color at `1.0`, the effective specular color is equal to the diffuse color, and the diffuse color is set to `0`. The material is considered a pure metal.



The screenshots below show the result of the metalness factor on a material with the following attributes:

- Gloss = `0.8`
- Diffuse = `#848484`, Lambert
- Specular GGX

Pure diffuse (no metalness)	**Metalness = `0.0`**	**Metalness = `1.0`**
		
- The diffuse color is dominant	- The diffuse color is dominant	- The diffuse color isn't visible
- The specular color isn't visible	- The specular color is visible (`0.02`)	- The specular color is visible

Specular map

The specular map provides more control over the actual specular color, but requires you to modify the diffuse color accordingly.

Unlike the metalness workflow, this lets you have a different specular color from the diffuse color even in low-reflection scenarios, allowing for materials with special behavior.

> **NOTE**
> You can combine metalness and specular workflows in the same material by adding separate layers.

Specular model

A pure specular surface produces a highlight of a light in a mirror direction. In practice, a broad range of specular materials, not entirely smooth, can reflect light in multiple directions. Stride simulates this using the **microfacet** model, also known as Cook-Torrance (academic paper).

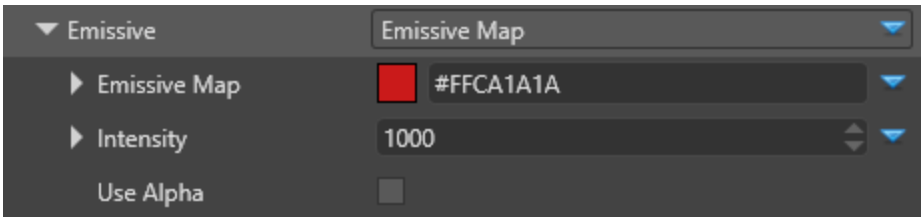
The microfacet is defined by the following formula, where Rs is the resulting specular reflectance:

$$Rs = \frac{(F \times D \times G)}{\pi(n \cdot l)(n \cdot v)}$$

Property	Description
Fresnel	Defines the amount of light that is reflected and transmitted. The models supported are: **Schlick**: An approximation of the Fresnel effect (default) **Thin glass**: A simulation of light passing through glass **None**: The material as-is with no Fresnel effect
Visibility	Defines the visibility between of the microfacets between (0, 1). Also known as the geometry attenuation - Shadowing and Masking - in the original Cook-Torrance. Stride simplifies the formula to use the visibility term instead: $$V = \frac{G}{(n \cdot l)(n \cdot v)}$$ and $$Rs = \frac{(F \times D \times V)}{\pi}$$ **Schlick GGX** (default) **Implicit**: The microsurface is always visible and generates no shadowing or masking **Cook-Torrance** **Kelemen** **Neumann** **Smith-Beckmann** **Smith-GGX correlated** **Schlick-Beckmann**
Normal Distribution	Defines how the normal is distributed. The gloss attribute is used by this part of the function to modify the distribution of the normal. **GGX** (default) **Beckmann** **Blinn-Phong**

Emissive

An **emissive** material is a surface that emits light.



With HDR, a Bloom and a Bright filter post-processing effects, we can see the influence of an emissive material:

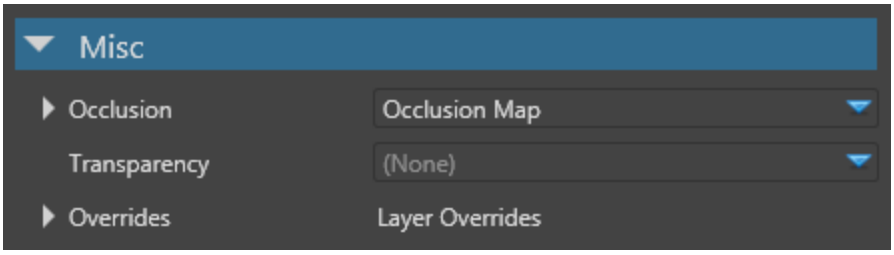
Property	Description
Emissive map	The emissive map color provider
Intensity	The factor to multiply by the color of the color provider
Use alpha	Use the alpha of the emissive map as the main alpha color of the material (instead of using the alpha of the diffuse map by default)

See also

- Geometry attributes
- Misc attributes
- Material maps
- Material layers
- Materials for developers
- Custom shaders

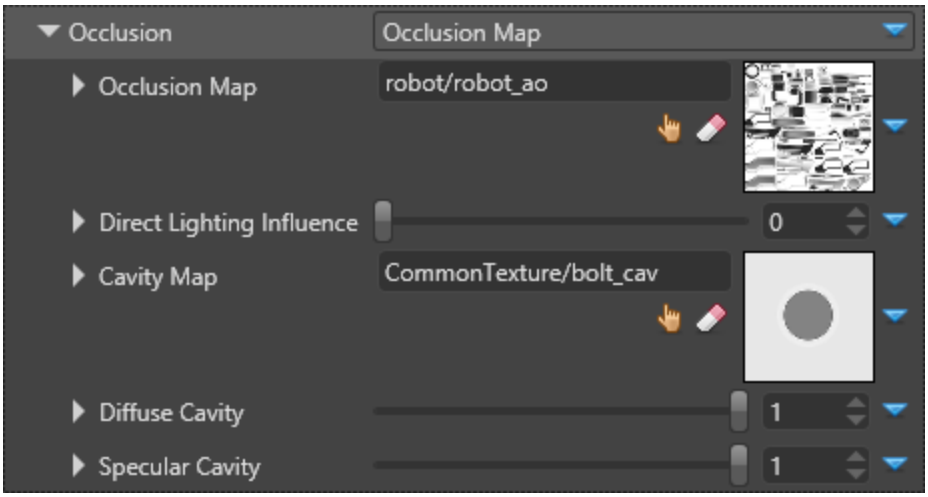
Misc attributes

Intermediate Artist Programmer

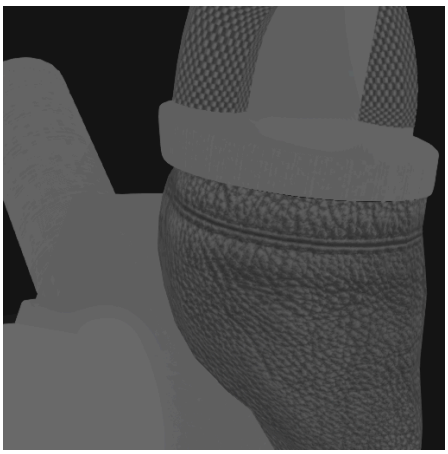


Occlusion

Under the **Occlusion** properties, you can set an **occlusion map**. This is the default occlusion attribute. The occlusion map use geometry occlusion information baked into a texture to modulate the ambient and direct lighting.



The screenshots below demonstrate the use of occlusion maps and cavity maps:

Occlusion Map	Cavity Map	Final Composition
		

Occlusion Map	Cavity Map	Final Composition
Coarse occlusion of the ambient light	Fine-grained occlusion of direct light	Result

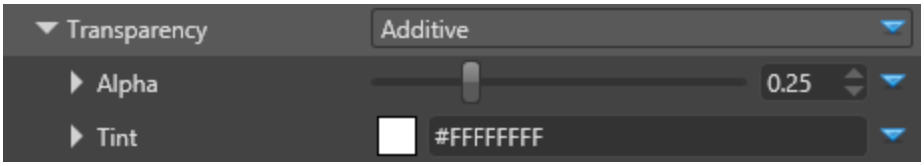
Property	Description
Occlusion Map	The occlusion map scalar provider that determines how much ambient light is accessible on the material. A value of 1.0 means that the material is fully lit by ambient lighting. A value of 0.0 means that the material is not lighted by the ambient lighting
Direct Lighting Influence	Applies to Occlusion Map and influences direct lighting
Cavity Map	The cavity map scalar provider is multiplied with direct lighting. It lets you define very fine grained cavity where direct light can't enter. The cavity map is usually defined for thin concave cavity
Diffuse Cavity	A factor for diffuse lighting influence of the cavity map. A value of 1.0 means the cavity map fully influences the diffuse lighting
Specular Cavity	A factor for specular lighting influence of the cavity map. A value of 1.0 means the cavity map fully influences the specular lighting

Transparency

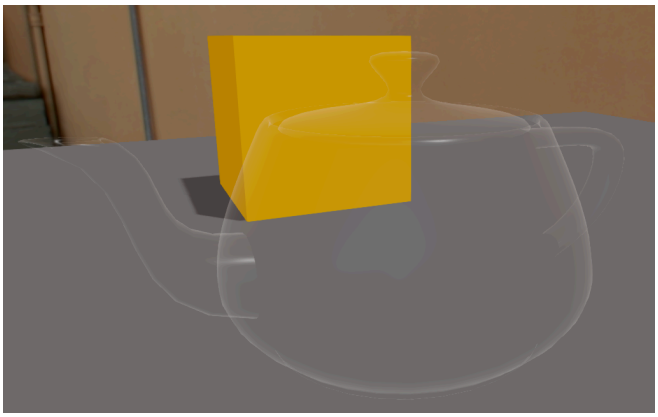
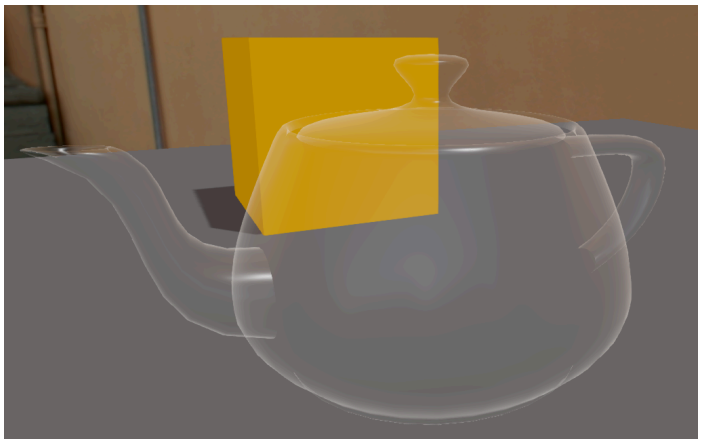
Under the **Transparency** properties, you can specify values that change the transparency of the material. You can coose **Blend**, **Additive**, or **Cutoff**.

Additive

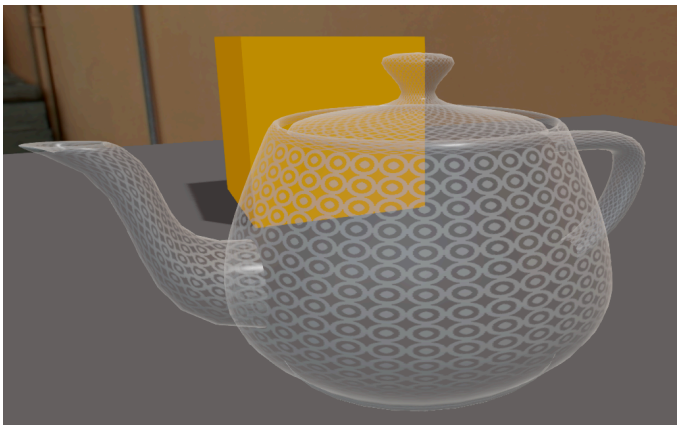
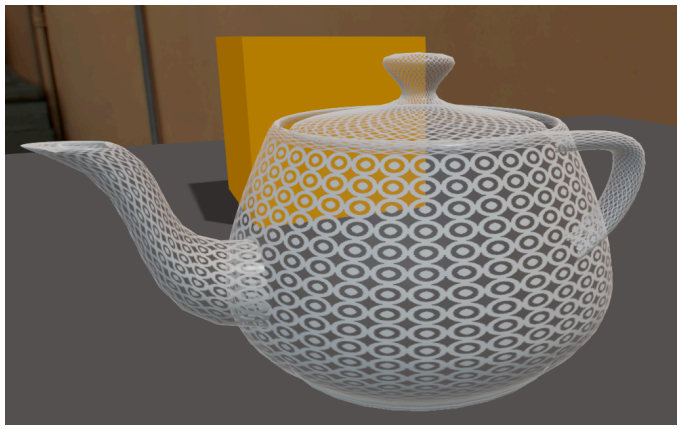
The additive transparency takes into account the diffuse and diffuse/emissive alpha.



- If the **Alpha** property is less than 0.5, only the specular highlights are visible. The material itself is completely invisible.

Alpha = 0.25	Alpha = 0.5
	
We only see the specular highlight in additive mode	Transparency is fully additive. Specular highlights at maximum

- If the **Alpha** <= 1.0, the material is semi-opaque with the diffuse/emissive component. If the diffuse component has an alpha, it's transparent.

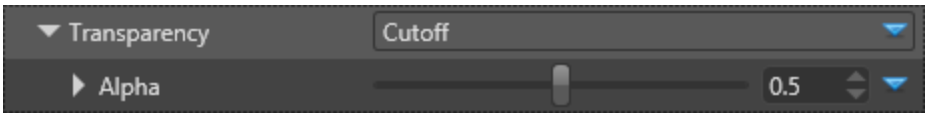
Alpha = 0.75	Alpha = 1.0
	
Specular highlights, diffuse with alpha and semi-opaque diffuse	Specular highlights, diffuse with alpha and opaque diffuse

Property	Description
Alpha	The alpha value is interpreted like this: Alpha <= 0.5, the material is rendered in additive mode without the diffuse component (only specular highlights)

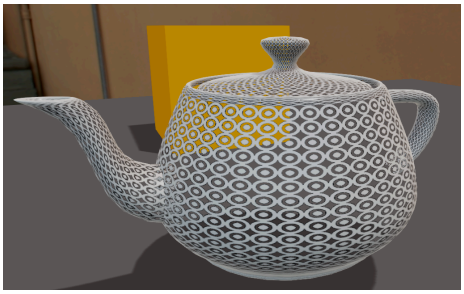
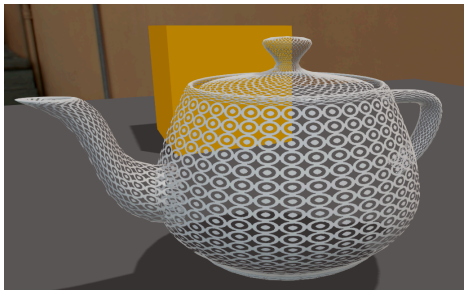
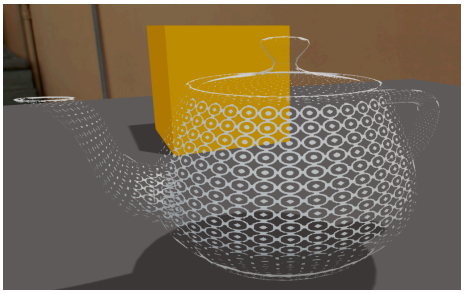
Property	Description
	Alpha <= 1.0, the material is rendered in semi-opaque mode with the diffuse/emissive component. If the diffuse component has an alpha, it's displayed as transparent
Tint	Apply a color tint to the transparency layer

Cuttoff

Renders a material when the current alpha color is above the threshold you specify with the **Alpha** slider.



The following screenshots show the influence of the cutoff Alpha value.

Alpha = 0.01	Alpha = 0.5	Alpha = 1.0
		

Clear coat

Clear-coat shading uses physically-based rendering to simulate vehicle paint.

For details, see clear-coat shading.

See also

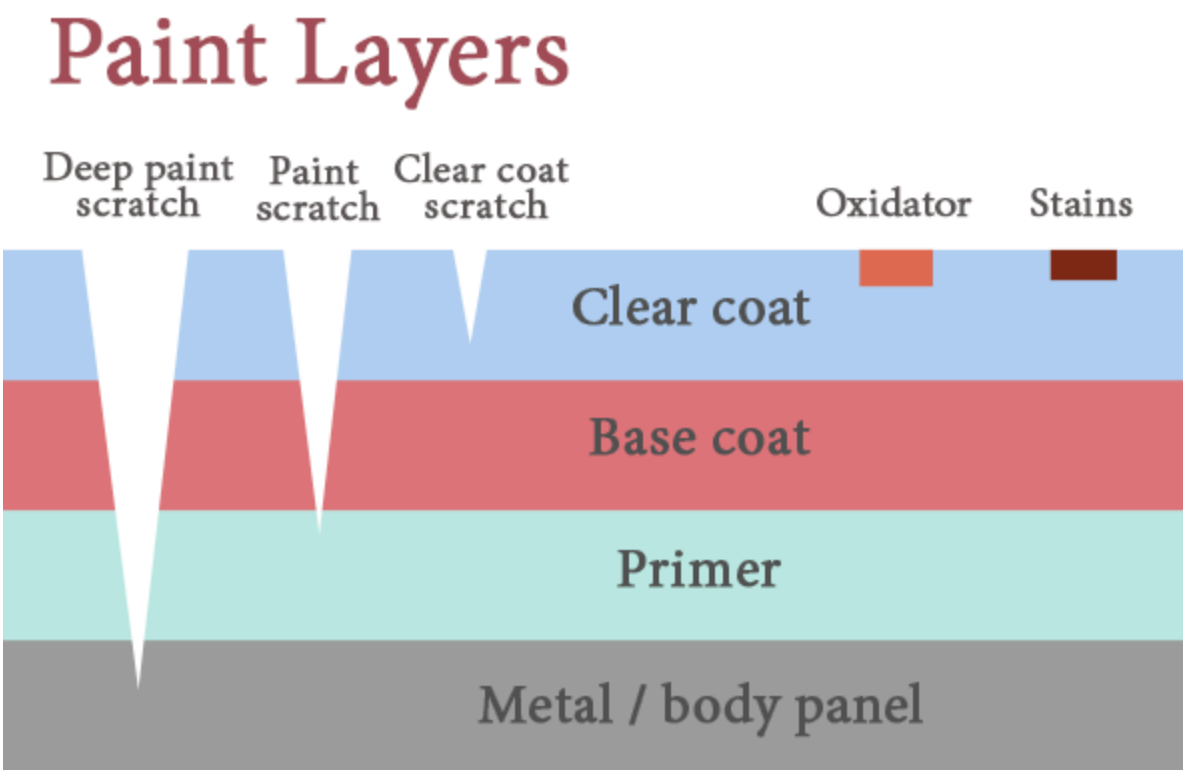
- Geometry attributes
- Shading attributes
- Clear-coat shading
- Material maps
- Material layers
- Material slots
- Materials for developers
- Custom shaders

Clear-coat shading

Intermediate Artist Programmer

Clear-coat shading uses physically-based rendering to simulate vehicle paint.

Real vehicles typically have three layers of paint applied to the body, as in the diagram below:



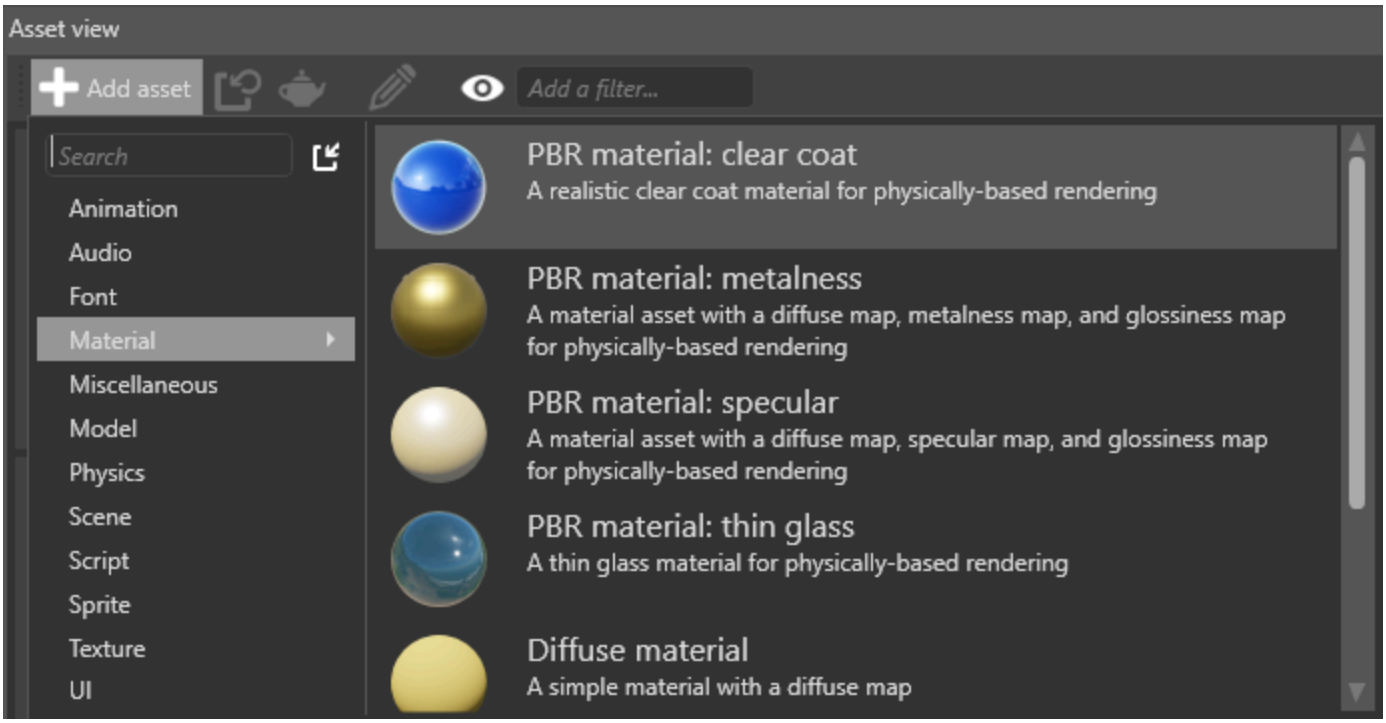
To keep the shading simple, Stride only simulates the **base coat** (including optional metal flakes) and **clear coat** layers. Stride blends the layers depending on how far the camera is from the material. This reduces visual artifacts caused by the metal flake normal map (which becomes more visible as the camera moves away from the material).

Clear-coat shading has several advantages over creating the effect manually with material layers:

- layers are blended based on distance
- increased performance
- improved visualization

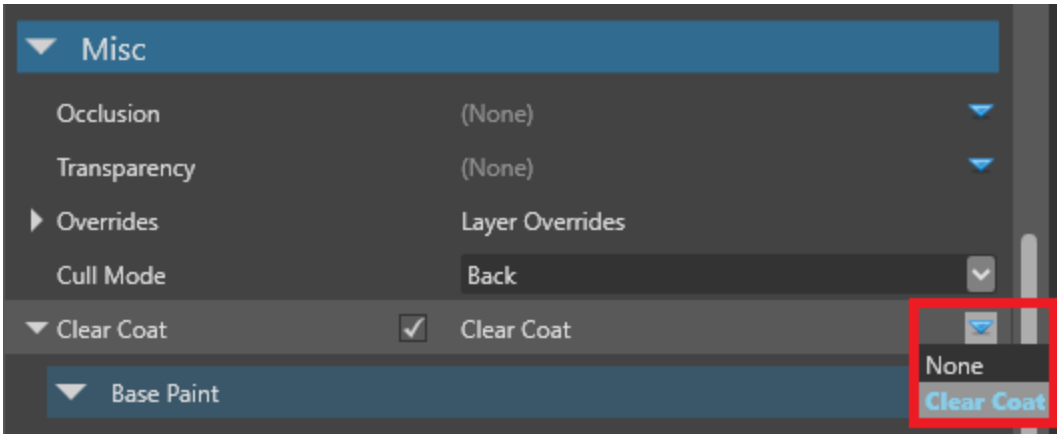
Add a clear-coat material

Stride includes a clear-coat material template. To add it, in the **Asset View**, click **Add asset** and select **Material > PBR material: clear coat**.

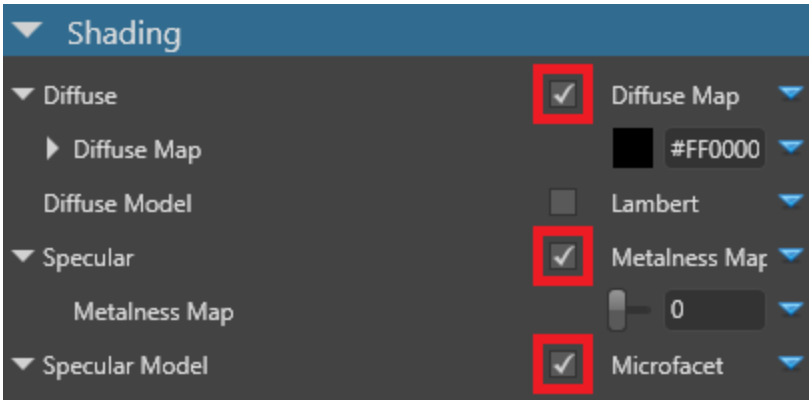


Alternatively, to set clear-coat properties yourself:

1. Select the material you want to use clear-coat shading.
2. In the Property Grid, under the **Misc** properties, next to **Clear coat**, click (**Replace**) and choose **Clear coat**.



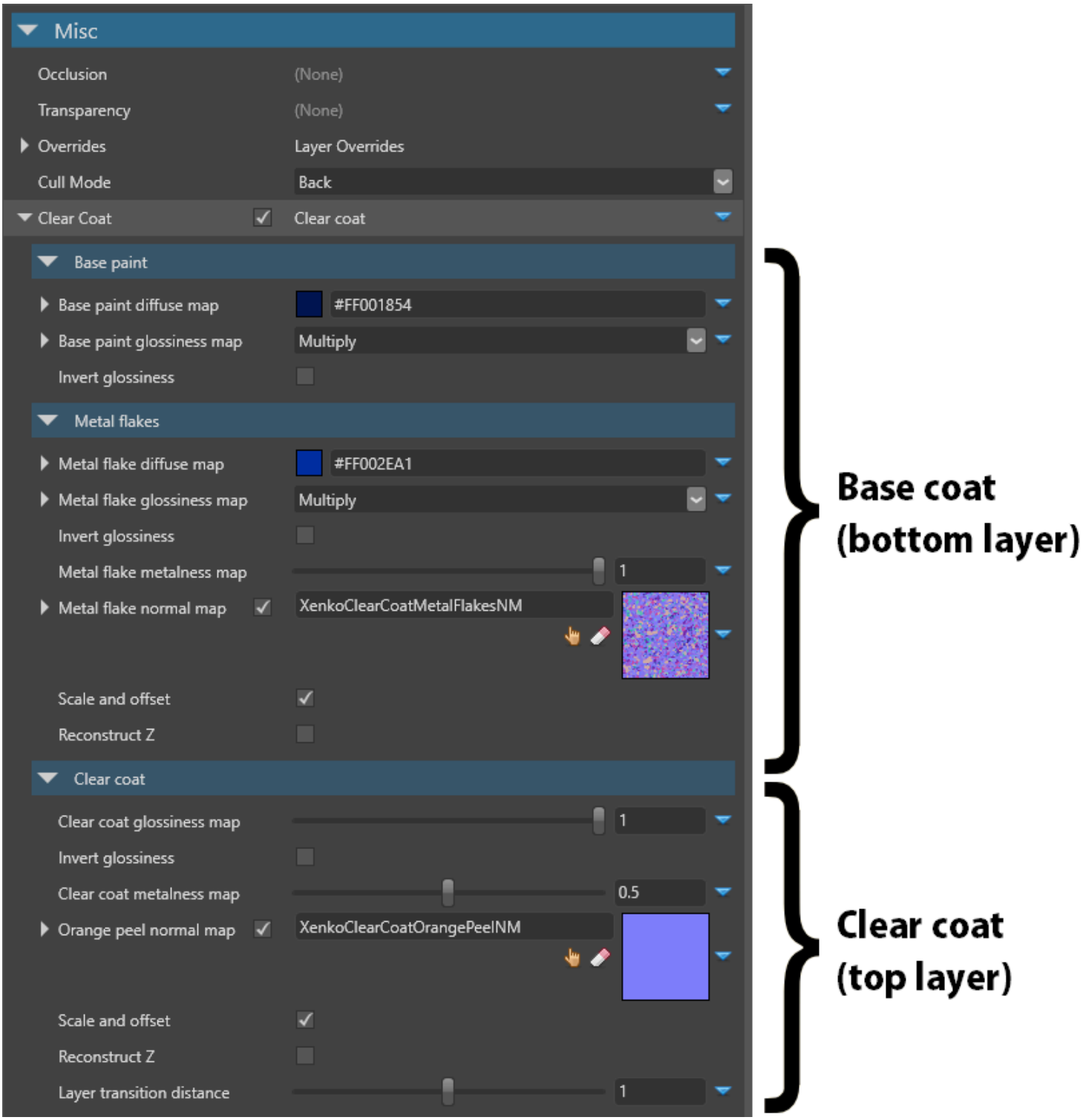
> **NOTE**
>
> For clear-coat shading to work correctly, make sure you enable **Diffuse**, **Specular** and **Specular model** under the material **Shading** properties.



Properties

You can access the clear-coat shader properties under **Misc > Clear coat**. They're split into three parts: the **base paint** and optional **metal flake** properties simulate the base coat, and the **clear coat** properties simulate the clear coat.

The metal flake properties simulate a metallic paint effect. To disable the effect, remove the metal flake normal map.

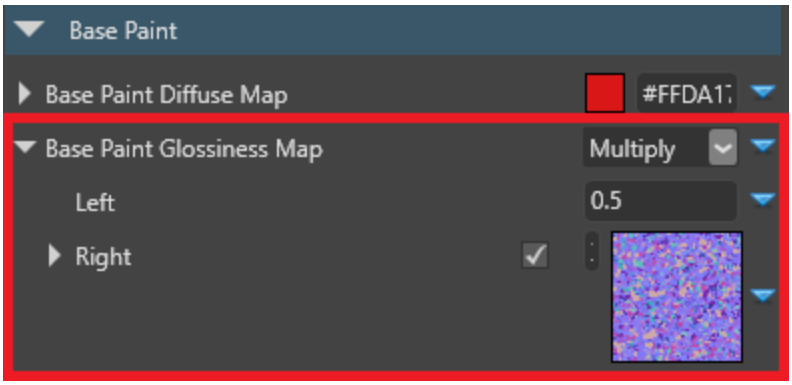


Property	Description
Base paint diffuse map	The diffuse map used by the base paint layer (the lowest layer). This determines the color of the layer.
Base paint gloss map	The gloss map used by the base paint layer. For a coherent result, use the **metal flake normal map** as a mask.

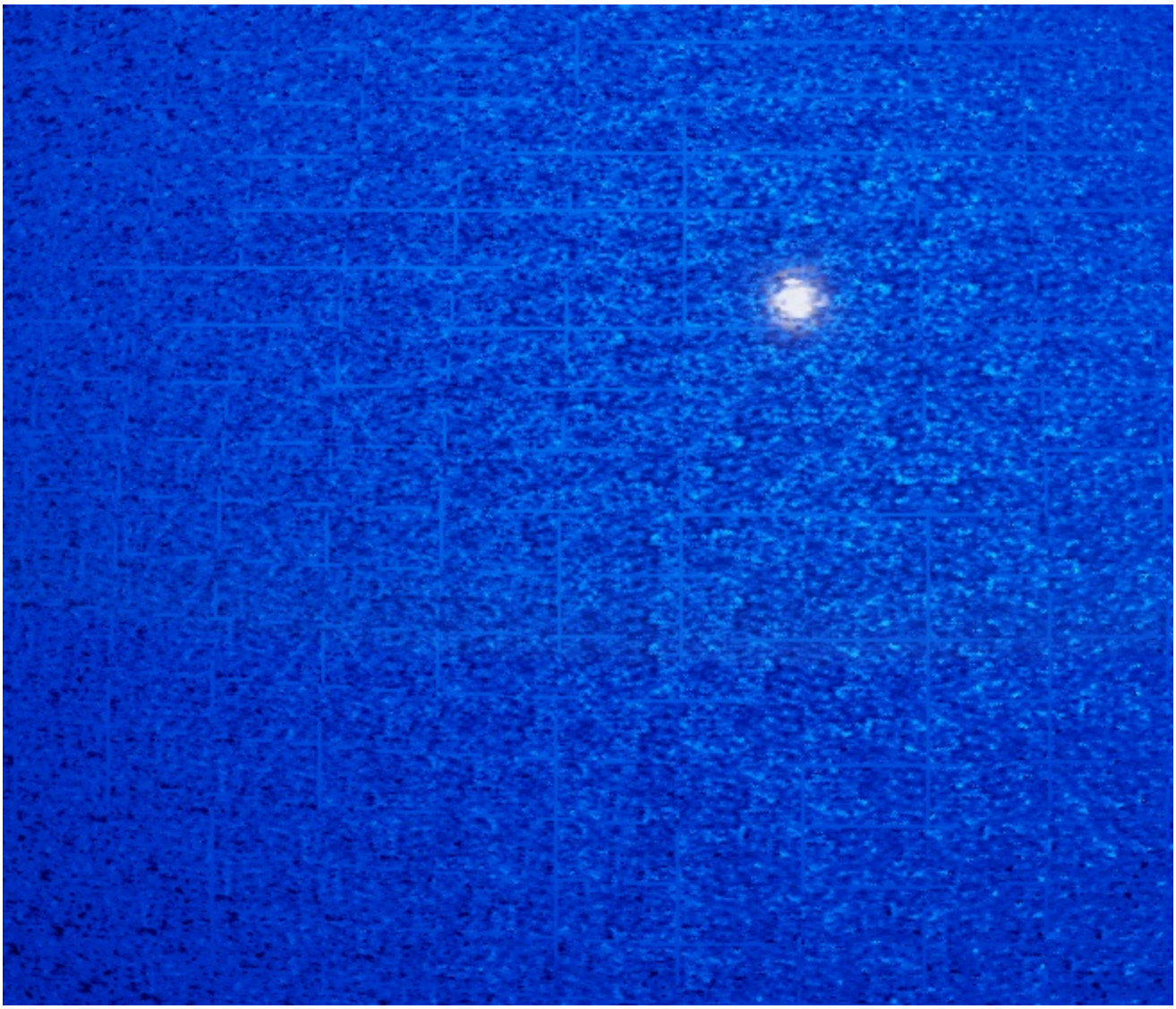
Property	Description
Metal flakes diffuse map	The diffuse map used by the metal flake layer (the layer above the base paint). For a coherent result, use a value close to the base paint value.
Metal flakes gloss map	The gloss map used by the metal flake layer. For a coherent result, use the **metal flake normal map** as a mask.
Metal flakes metalness map	The metalness map used by the metal flake layer. For best results, use high values.
Metal flake normal map	The normal map used by the metal flake layer. This shapes the flake geometry. A metal flake normal map (**StrideClearCoatMetalFlakesNM**) is included in the Stride assets package. If the texture has a high UV scale, enable **Use random texture coordinates** below to reduce tiling effects. To disable the metal flakes effect, remove the normal map.
Coat gloss map	The gloss map used by the clear coat layer. Change this value to simulate different kinds of paint (eg matte).
Clear coat metalness map	The metalness map used by the clear coat layer
Orange peel normal map	The normal map used by the clear coat layer to create an "orange peel" effect. This reflects light in different angles, simulating paint imperfections whereby the texture appears bumpy, like the skin of an orange. An orange peel normal map (**StrideClearCoatOrangePeelNM**) is included in the Stride assets package.
Layer transition distance	The distance (in meters) at which the base paint layer transitions to the metal flake layer. This helps fight visual artifacts caused by the metal flake normal map (which becomes more visible as the camera moves away from the material).

Reduce tiling and artifacts

Properties that use binary operators should use **normalized values** (ie between `0.0` and `1.0`). For example, in the screenshot below, the **left** operator uses a value of `0.5`.



Values over `1.0` might produce tiling artifacts, as in the image below (note the grid pattern):

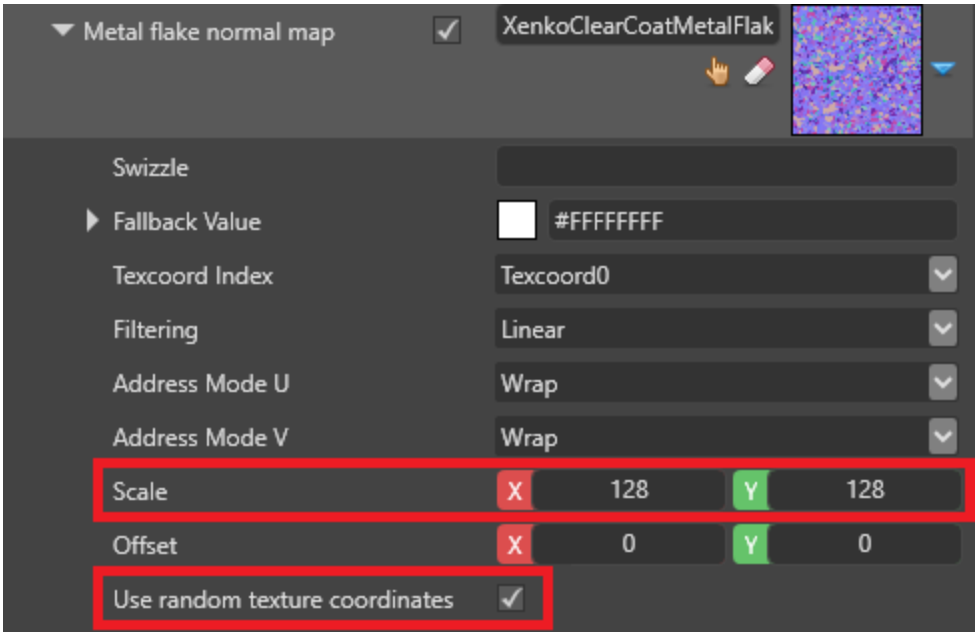


StrideClearCoatMetalFlakesNM

The metal flakes in the metal flake normal map included in the Stride assets package (**StrideClearCoatMetalFlakesNM**) are quite large. For this reason, we recommend you:

- use a high **UV scale factor** which tiles the texture (thereby shrinking the flakes)

- enable **Use random texture coordinates**, preventing an obvious tiling effect



> **NOTE**
>
> The **Use random texture coordinates** option is costly, so we don't recommend you use it for mobile platforms.

Alternatively, use a normal map with a higher density of smaller metal flakes.

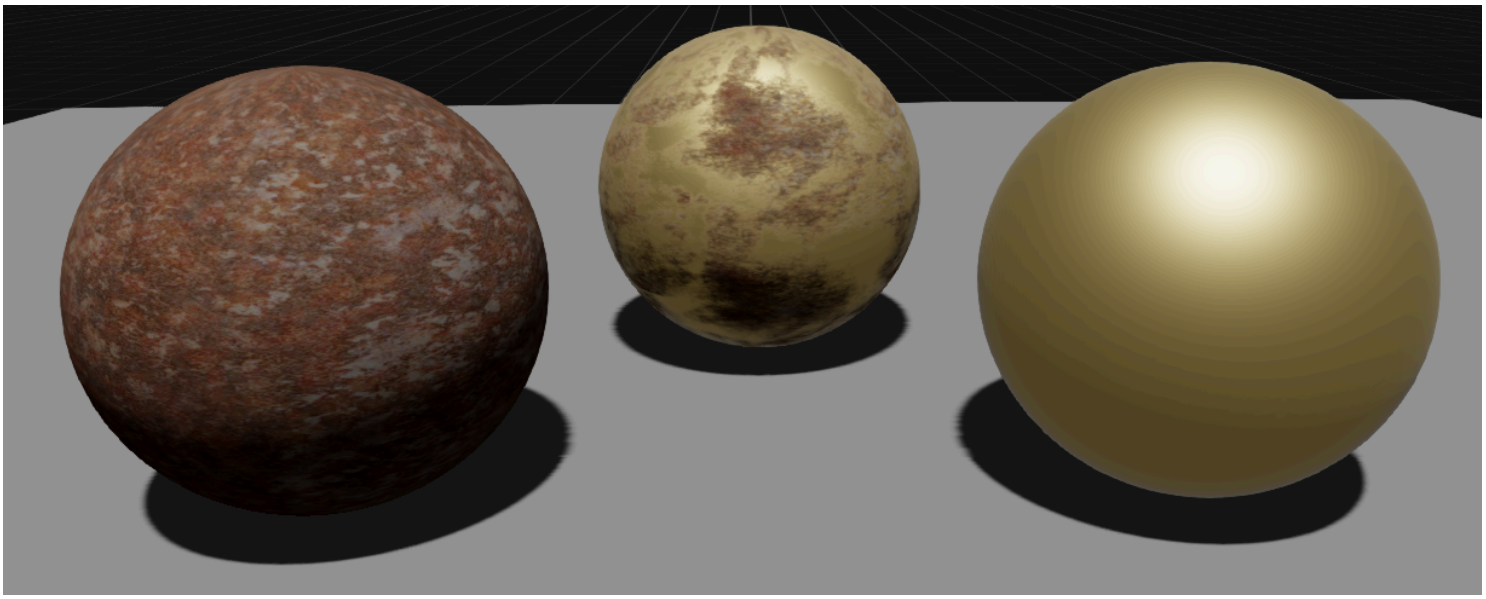
See also

- Material maps
- Material attributes
  - Geometry attributes
  - Shading attributes
  - Misc attributes
- Material layers
- Material slots
- Materials for developers

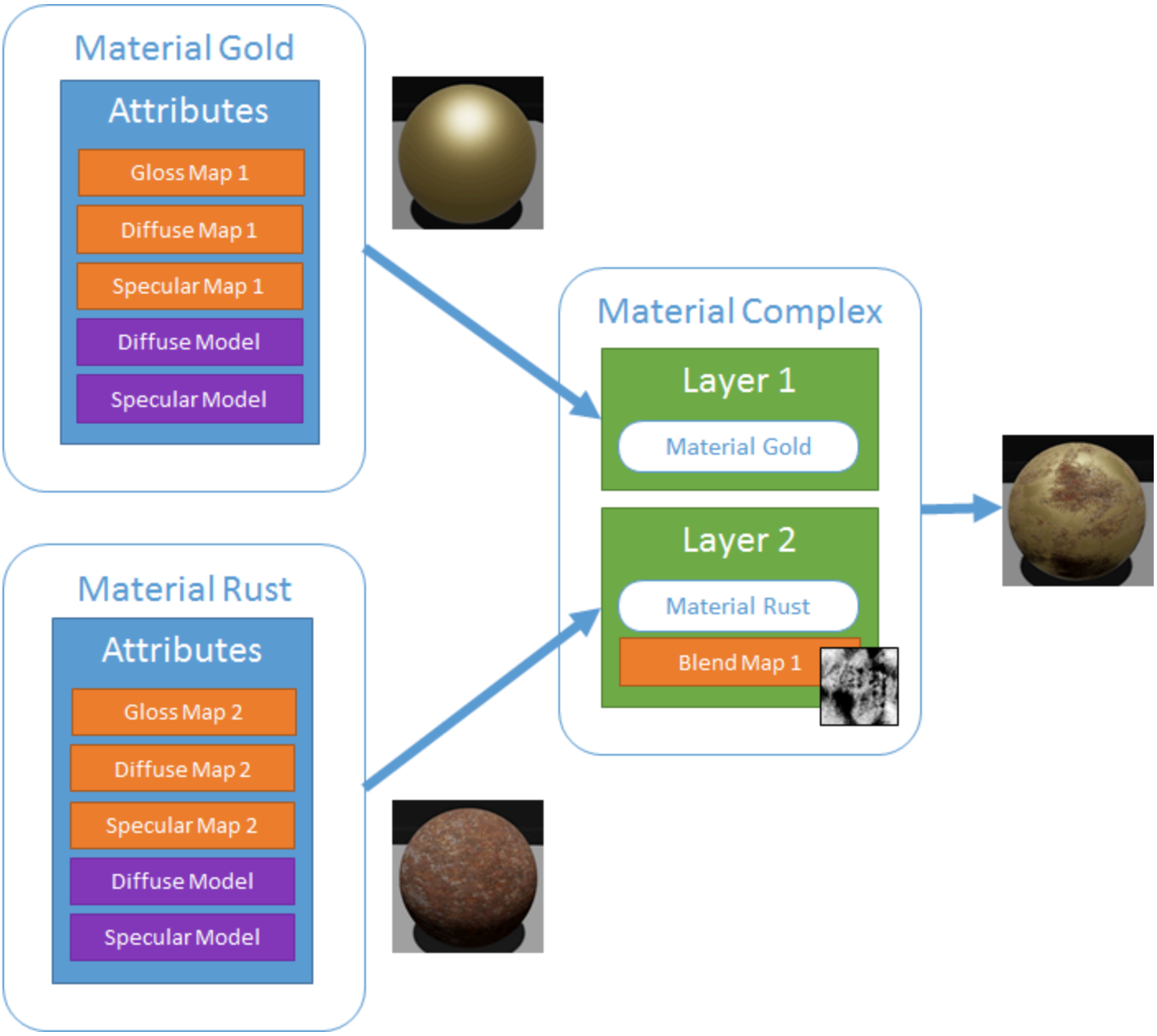
Material layers

Intermediate Artist Programmer

You can combine layers of materials to build more complex materials. For example, this screenshot shows the blending of a rust material (left) with a gold material (right):

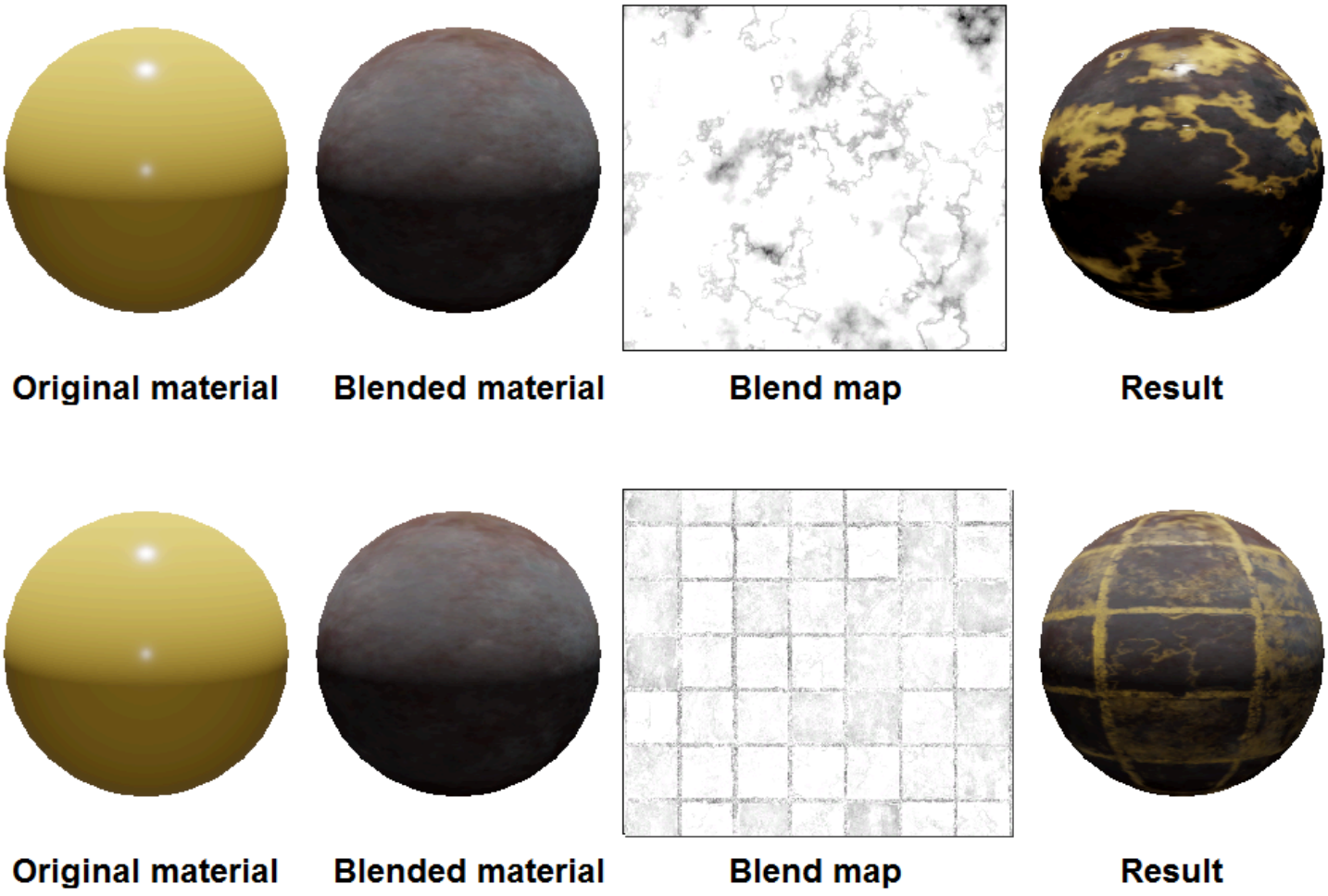


This diagram shows the definition of the materials blended in the screenshot above:



Blend maps

Blend maps are material maps that determine how Game Studio blends layers. For example, you can use a texture as a blend map:



Note how the blend map texture corresponds to the patterning on the result.

Blend maps work the same way as any other kind of material map. For more information, see Material maps.

Shading models

Stride blends materials differently depending on whether their shading models (eg diffuse models, specular models, etc) are different.

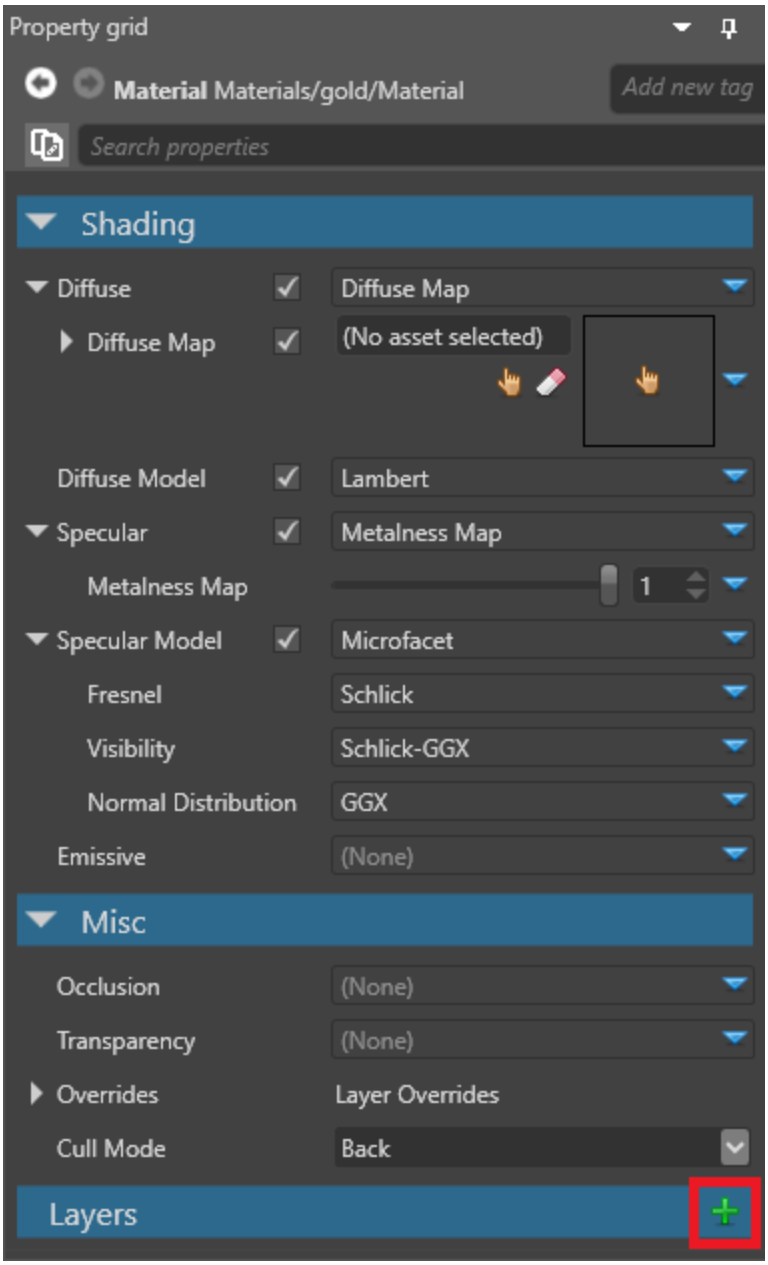
If you blend materials that have **identical** shading models, Stride collects the **attributes** of the materials, then applies the shading models to all of them. This saves GPU.

If the materials have **different** shading models, Stride applies each material's shading models to that material's attributes, then blends the results. This uses more GPU.

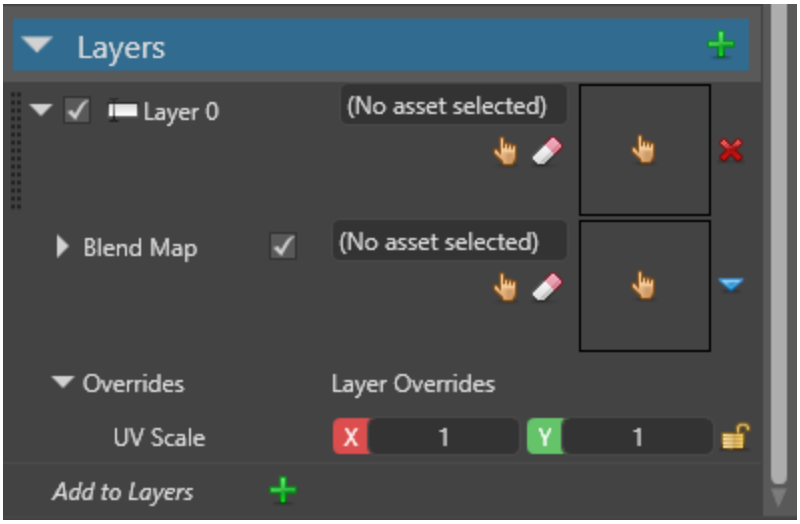
Add a layer

1. Select the material you want to add a layer to.

2. In the **Property Grid** (on the right by default), next to **Layers**, click (**Add**).

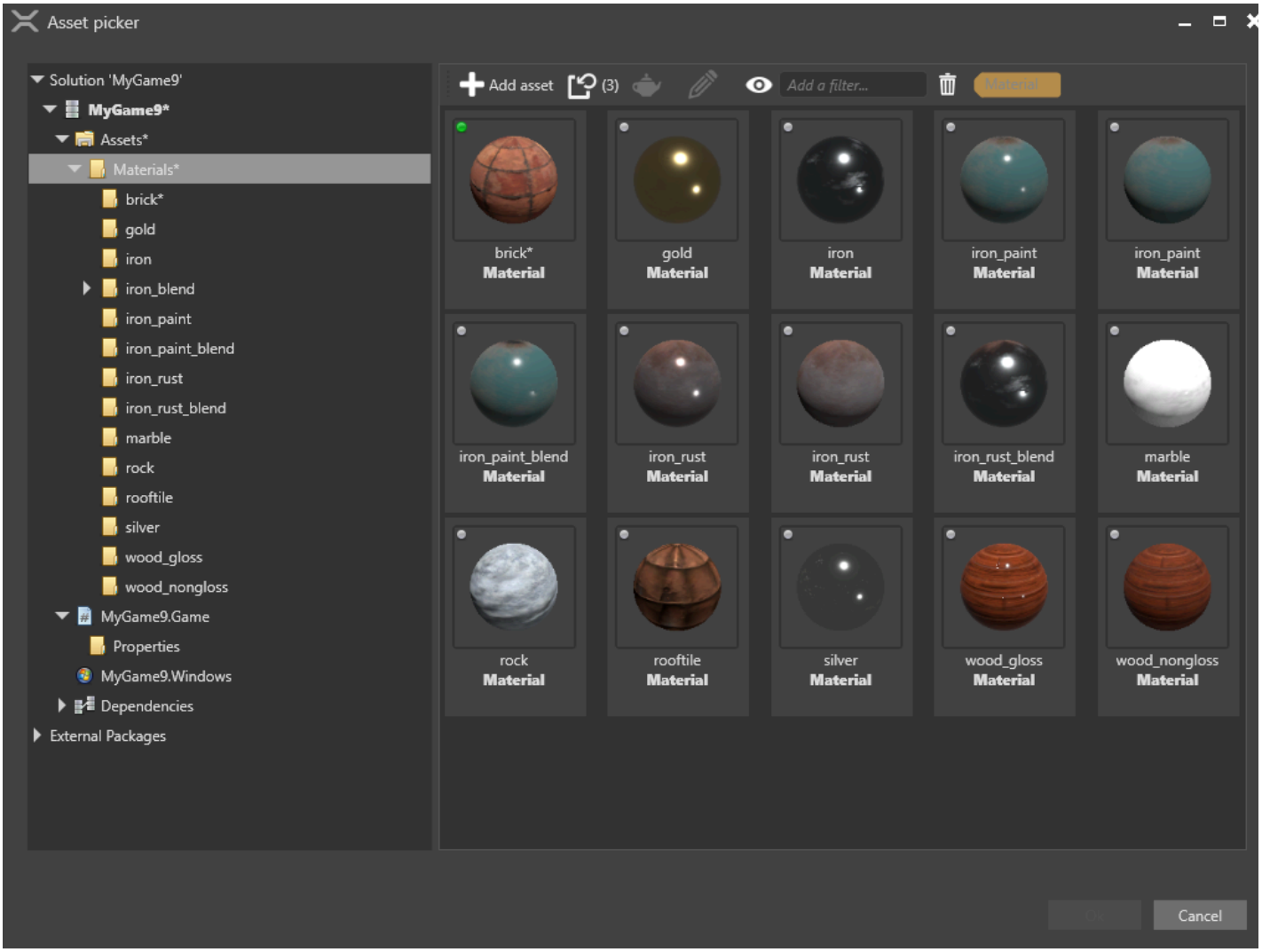


Game Studio adds a layer to the material.



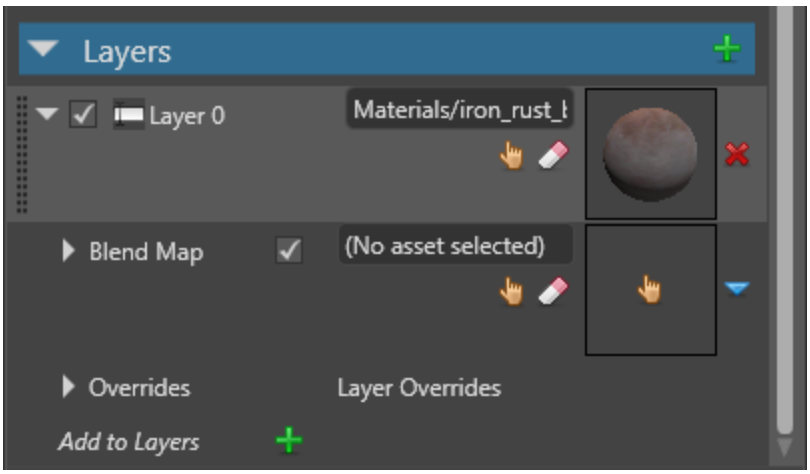
3. Next to the layer, click (**Select an asset**).

The **Select an asset** window opens.

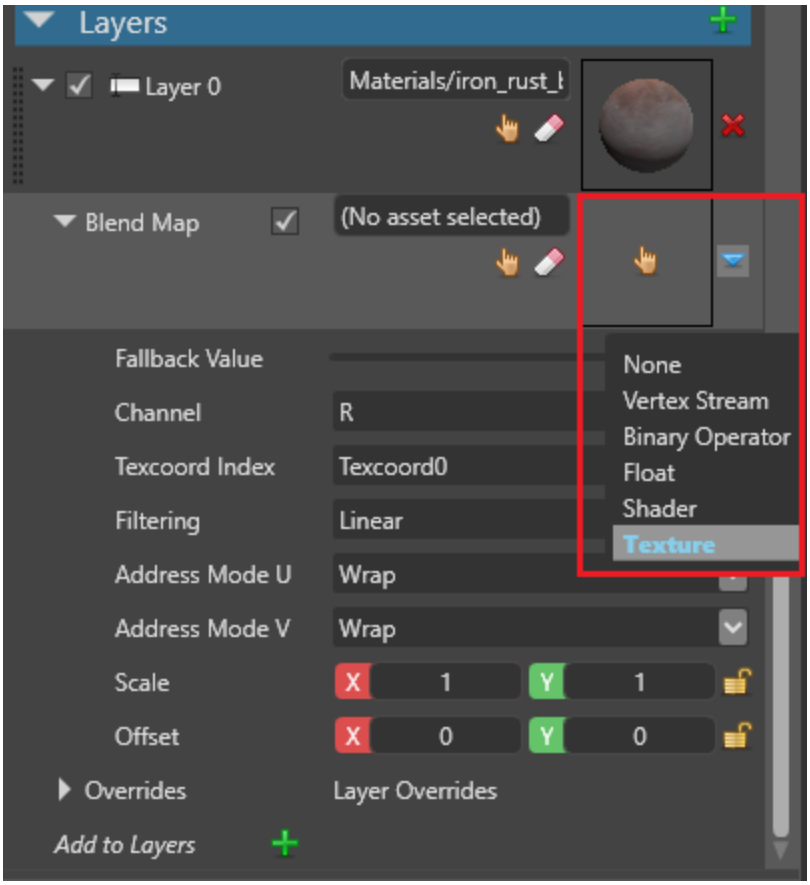


4. Specify a material you want to add as a layer and click **OK**.

   Game Studio adds the material as a layer.



5. Next to **Blend Map**, click (**Replace**) and select the type of blend map you want to use to blend the layers. For more information about blend maps, see Material maps.



Game Studio blends the material layers using the blend map you specified. You can add as many layers as you need.

Layer properties

Property	Description
Material	The material blended in this layer
Blend Map	The blend map used to blend this layer with the layer above
Layer Overrides	**UV Scale**: A UV scale applied to all textures UV of the material of the layer (excluding the occlusion map)

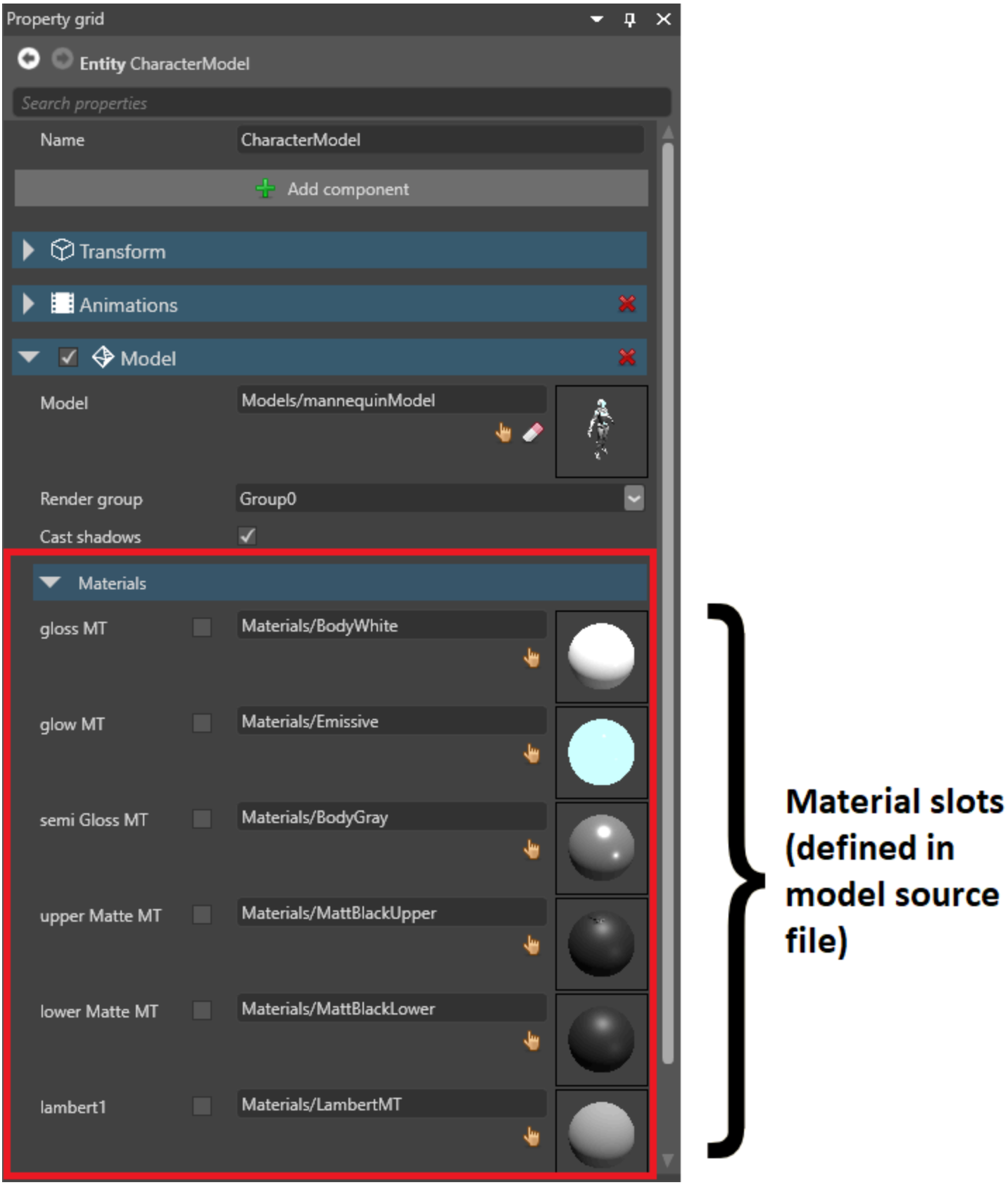
See also

- Material maps
- Material attributes
- Material slots
- Materials for developers

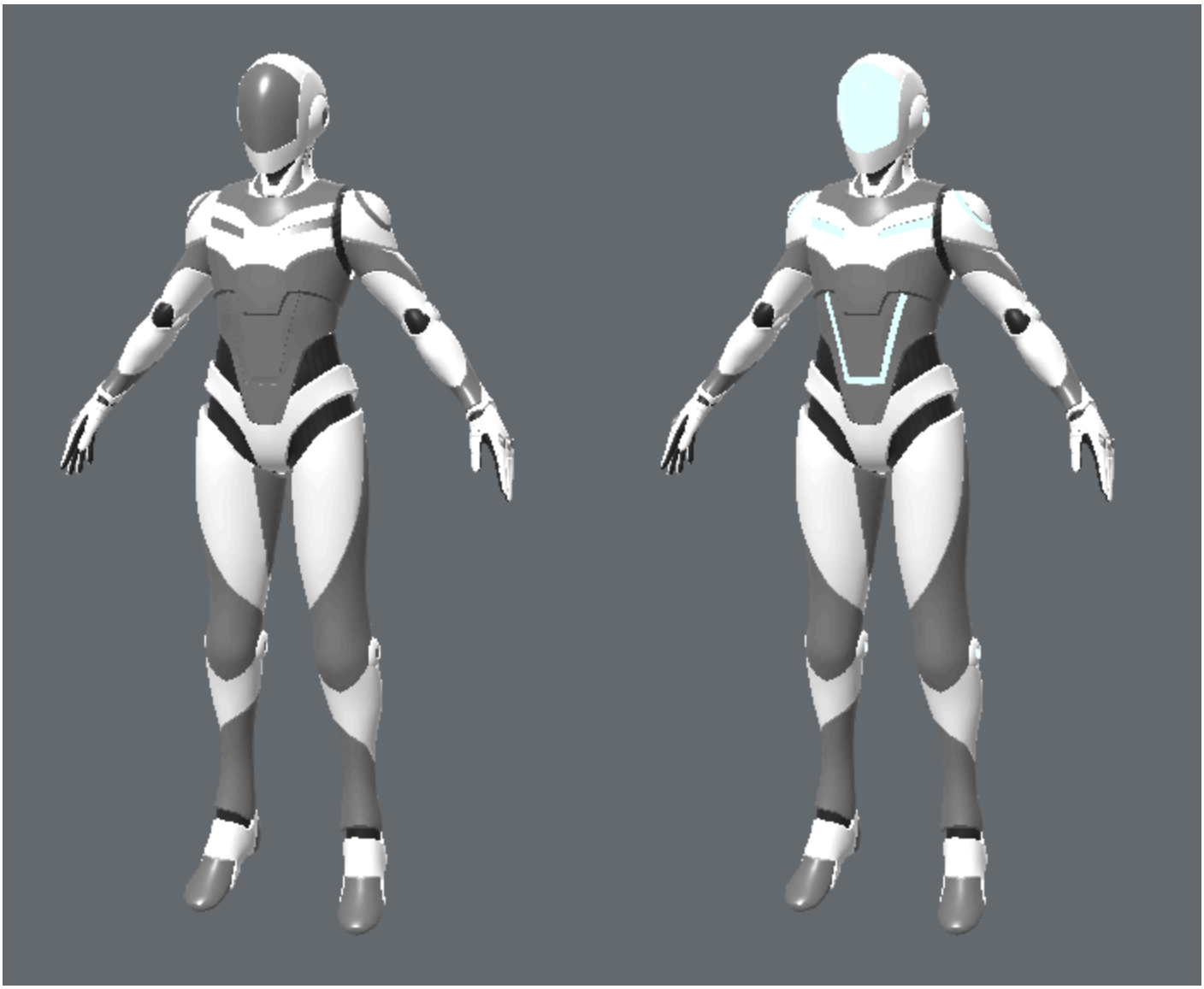
Material slots

Intermediate Artist Programmer

Models can use multiple materials. You can set the materials in the model's **material slots**.



For example, the second material slot in this model specifies the material for the visor and the shoulder and chest plate stripes. By changing the material in this slot, we change the material used in these parts of the model.

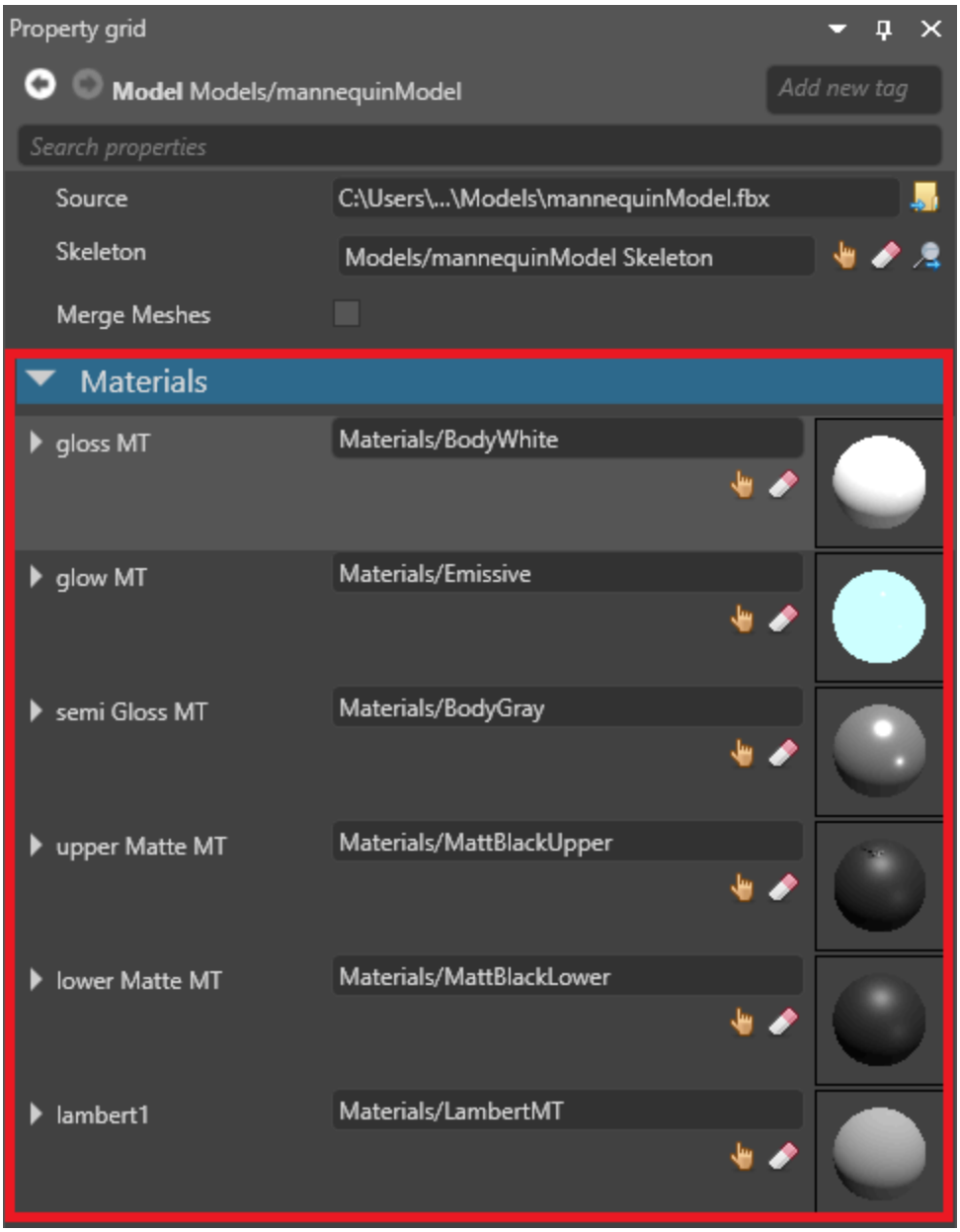


The material slots themselves — their number and position — are defined in the model source file (eg `.fbx`, `.obj`, etc). You can't edit material slots in Game Studio; you can only change which materials are used in each slot.

Set materials on a model

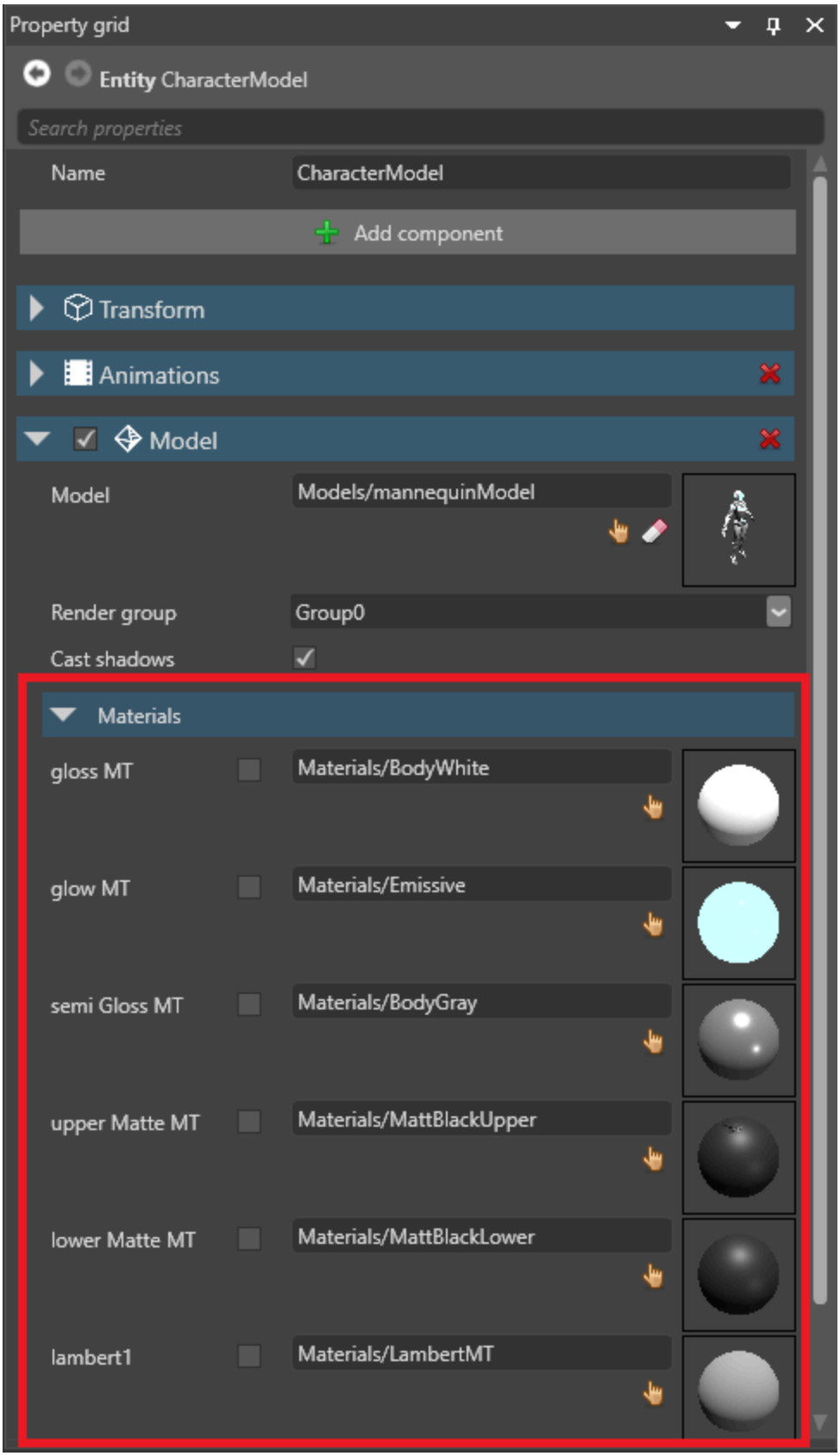
You can change the materials a model uses in two places:

- Under the **Materials** properties of the model itself:



> **NOTE**
>
> This affects every instance of this model.

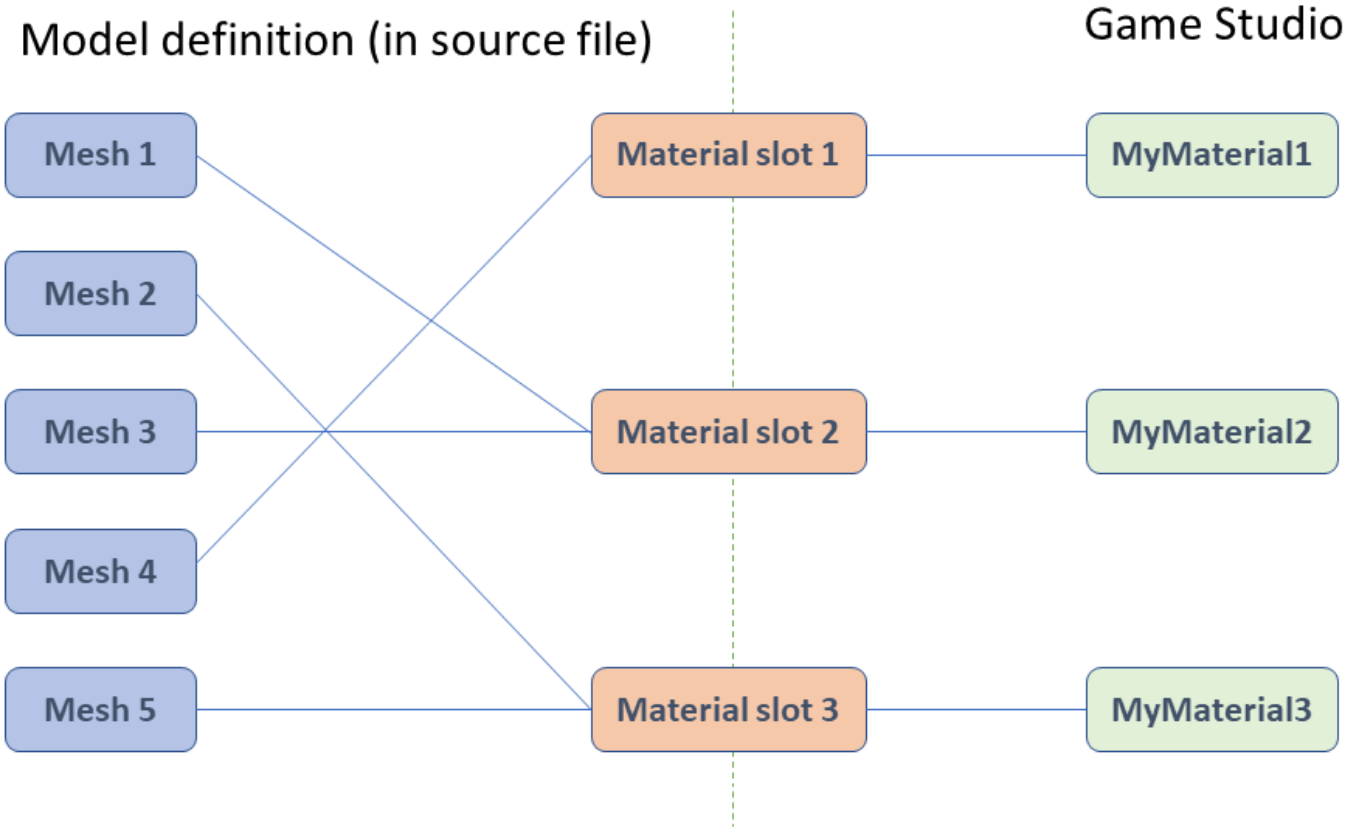
- In the **model component** of an entity or prefab:



This only affects **this** instance or prefab.

Meshes and material slots

Models imported from modeling software can contain meshes. Meshes can share materials via material slots.



The association between a mesh and a material slot is defined in the model source file. You can't change these associations in Game Studio, but you can change them in code at runtime.

To change the association between a mesh and a material, use:

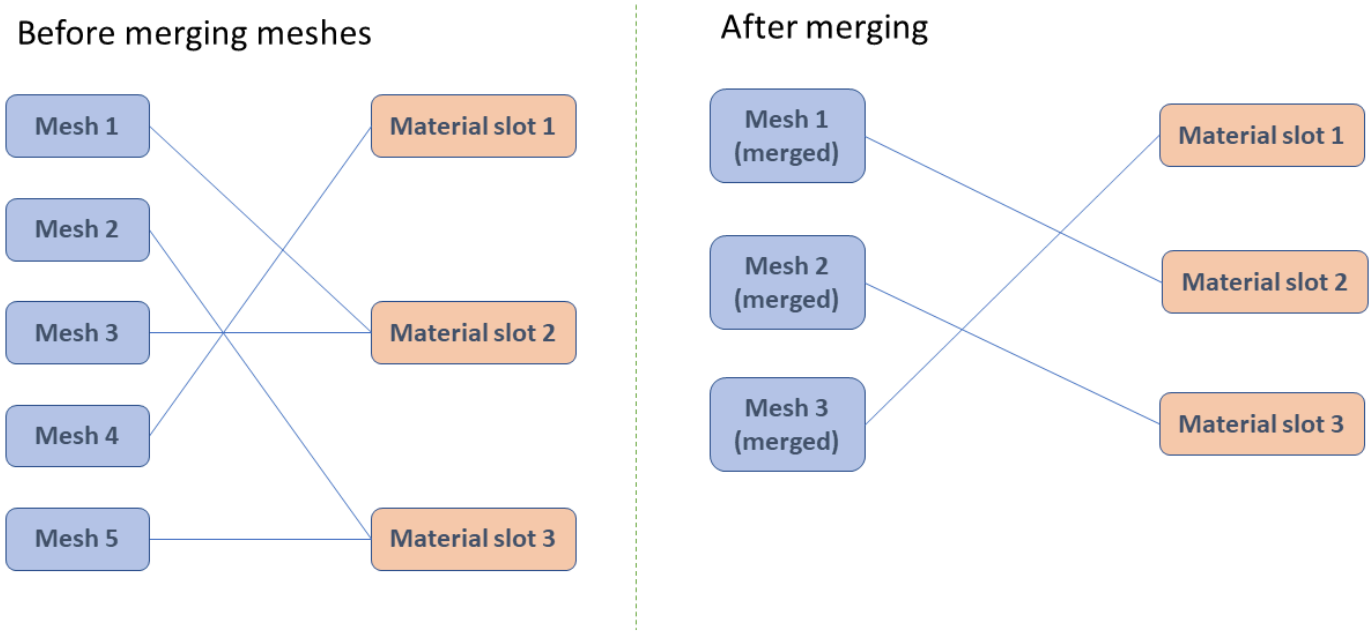
```
MyModelComponent.Model.Meshes[submeshIndex].MaterialIndex = materialIndex;
```

To change or add a material to the list of materials:

```
MyModelComponent.Materials[ExistingOrNewMaterialIndex] = myMaterial;
```

Merging meshes

When Stride draws a model with meshes, it performs one GPU draw call for each mesh. By default, to improve performance, at build time, Stride merges meshes that share materials.



In the example above, there are five meshes and five draw calls. After merging, there are three meshes and three draw calls.

> **NOTE**
>
> When Stride merges meshes, it merges the vertex and index buffers. This means you can't draw the meshes separately at runtime, and you can't change the original mesh position (transformation matrix). The meshes become a single mesh with a single material and a single transformation matrix (relative to the model).

> **NOTE**
>
> When Stride merges meshes, it changes the draw order of elements. In the case of transparent materials, this can produce different results.

> **NOTE**
>
> When you create a physics collider from a model, Stride builds separate convex hulls for each mesh in the model. If the meshes are merged, only one mesh remains per material, so convex hulls are also built from merged meshes.

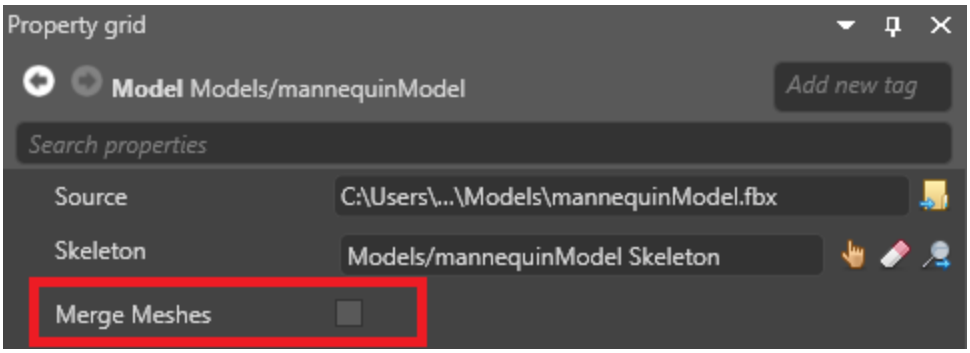
Disable mesh merging

You might want to disable mesh merging if you want to:

- animate a mesh
- change the material of a mesh at runtime

To disable mesh merging on a model:

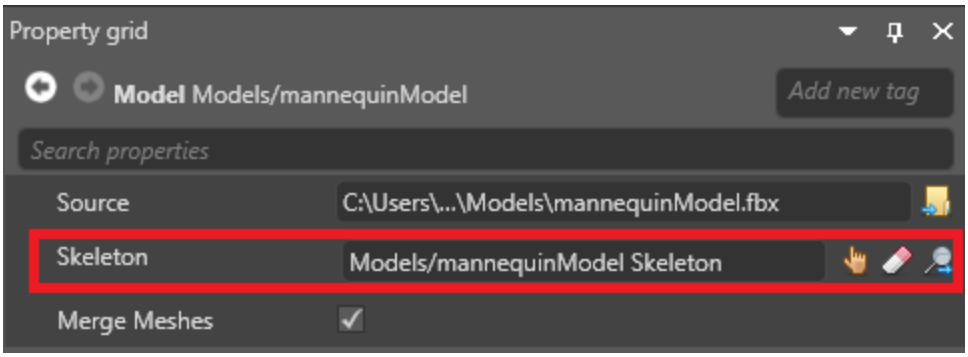
1. Select the model you want to disable mesh merging for.
2. In the **Property Grid**, disable **Merge meshes**.



Disable merging for specific meshes

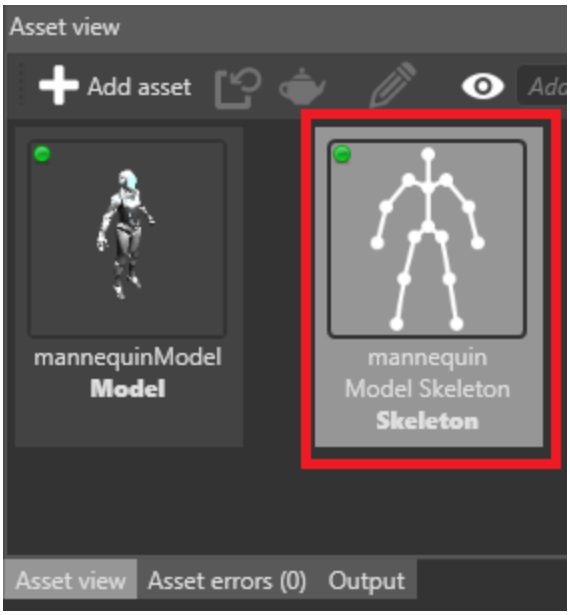
To disable merging only for specific meshes, enable their corresponding **nodes**.

1. Select the model that contains the meshes.
2. In the **Property Grid**, under **Skeleton**, make sure the model has a skeleton associated with it.

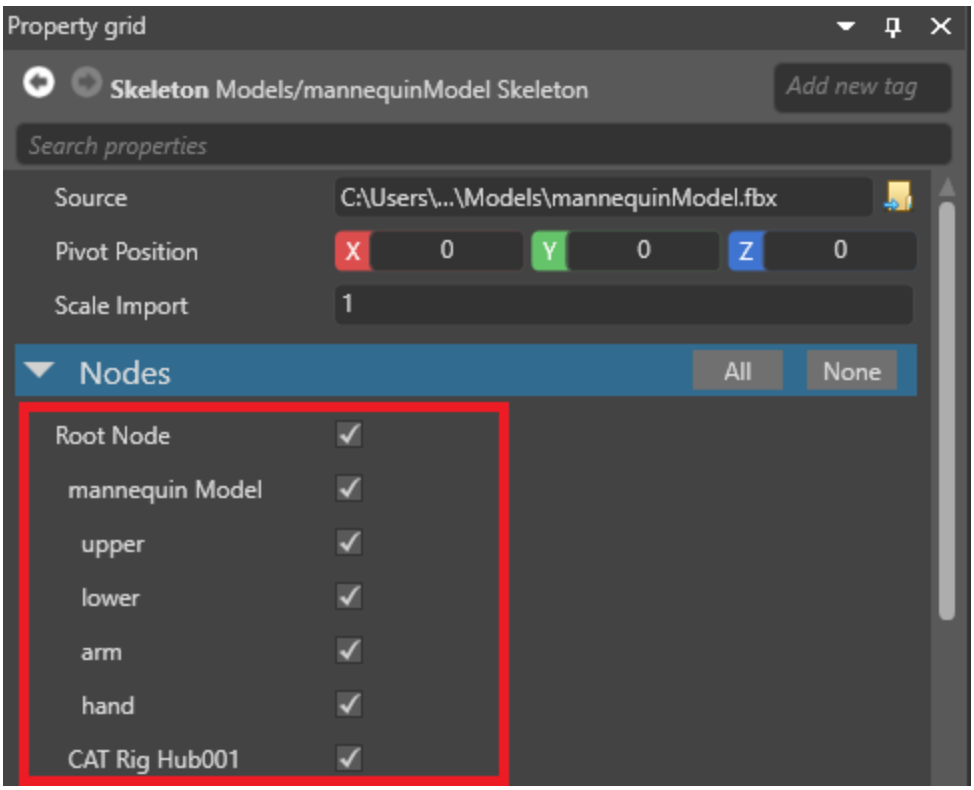


   For more information about skeletons, see Animation.

3. In the **Asset View**, select the skeleton.



4. In the **Property Grid**, under **Nodes**, select the nodes that correspond to the meshes you don't want to merge.



> **TIP**
>
> To see which nodes correspond to which mesh, open the model source file in a modeling application such as Maya.

> **NOTE**
>
> Make sure you don't disable nodes that are animated at runtime.

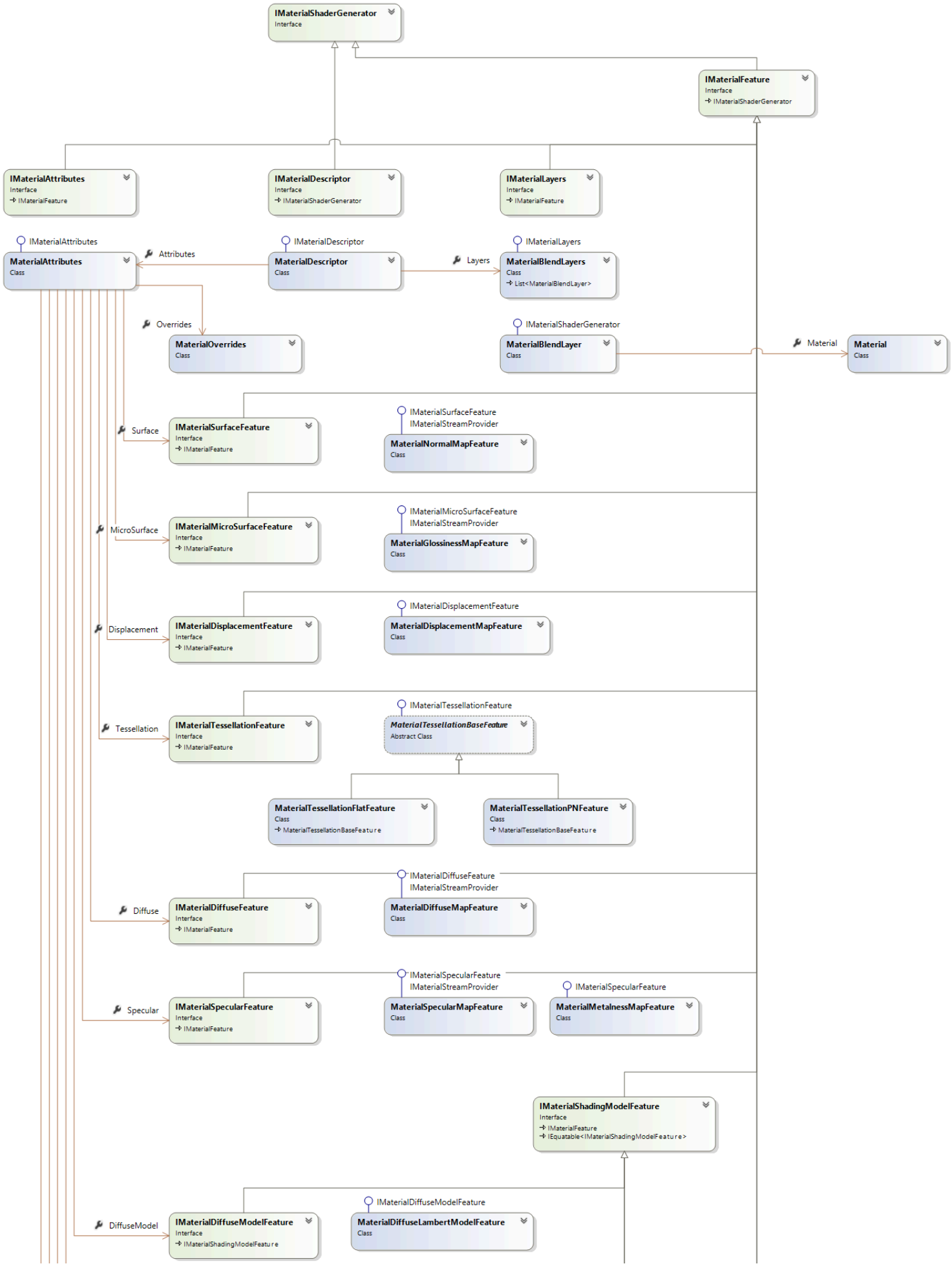
See also

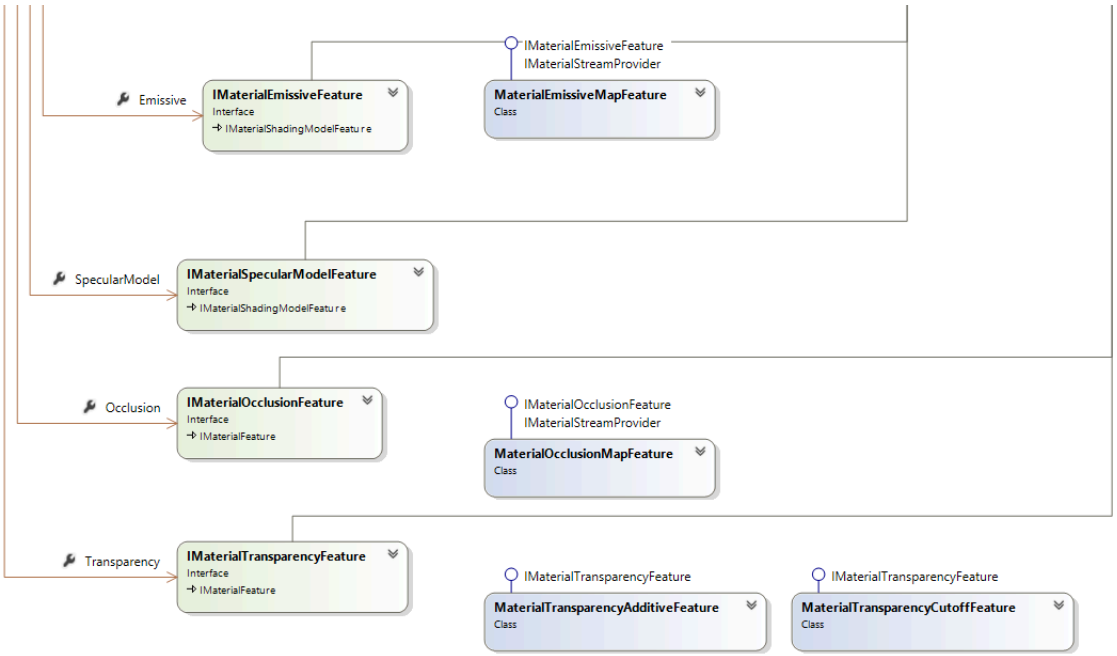
- Material maps
- Material attributes
- Material slots

Materials for developers

Advanced Programmer

This diagram shows the Material interfaces and implementation classes:


IMaterialShaderGenerator
Interface
IMaterialFeature
Interface
IMaterialShaderGenerator
IMaterialAttributes
Interface
IMaterialFeature
IMaterialDescriptor
Interface
IMaterialShaderGenerator
IMaterialLayers
Interface
IMaterialFeature
IMaterialAttributes
MaterialAttributes
Class
Attributes
IMaterialDescriptor
MaterialDescriptor
Class
Layers
IMaterialLayers
MaterialBlendLayers
Class
List<MaterialBlendLayer>
Overrides
MaterialOverrides
Class
IMaterialShaderGenerator
MaterialBlendLayer
Class
Material
Material
Class
Surface
IMaterialSurfaceFeature
Interface
IMaterialFeature
IMaterialSurfaceFeature
IMaterialStreamProvider
MaterialNormalMapFeature
Class
MicroSurface
IMaterialMicroSurfaceFeature
Interface
IMaterialFeature
IMaterialMicroSurfaceFeature
IMaterialStreamProvider
MaterialGlossinessMapFeature
Class
Displacement
IMaterialDisplacementFeature
Interface
IMaterialFeature
IMaterialDisplacementFeature
MaterialDisplacementMapFeature
Class
Tessellation
IMaterialTessellationFeature
Interface
IMaterialFeature
IMaterialTessellationFeature
MaterialTessellationBaseFeature
Abstract Class
MaterialTessellationFlatFeature
Class
MaterialTessellationBaseFeature
MaterialTessellationPNFeature
Class
MaterialTessellationBaseFeature
Diffuse
IMaterialDiffuseFeature
Interface
IMaterialFeature
IMaterialDiffuseFeature
IMaterialStreamProvider
MaterialDiffuseMapFeature
Class
Specular
IMaterialSpecularFeature
Interface
IMaterialFeature
IMaterialSpecularFeature
IMaterialStreamProvider
MaterialSpecularMapFeature
Class
IMaterialSpecularFeature
MaterialMetalnessMapFeature
Class
IMaterialShadingModelFeature
Interface
IMaterialFeature
IEquatable<IMaterialShadingModelFeature>
DiffuseModel
IMaterialDiffuseModelFeature
Interface
IMaterialShadingModelFeature
IMaterialDiffuseModelFeature
MaterialDiffuseLambertModelFeature
Class

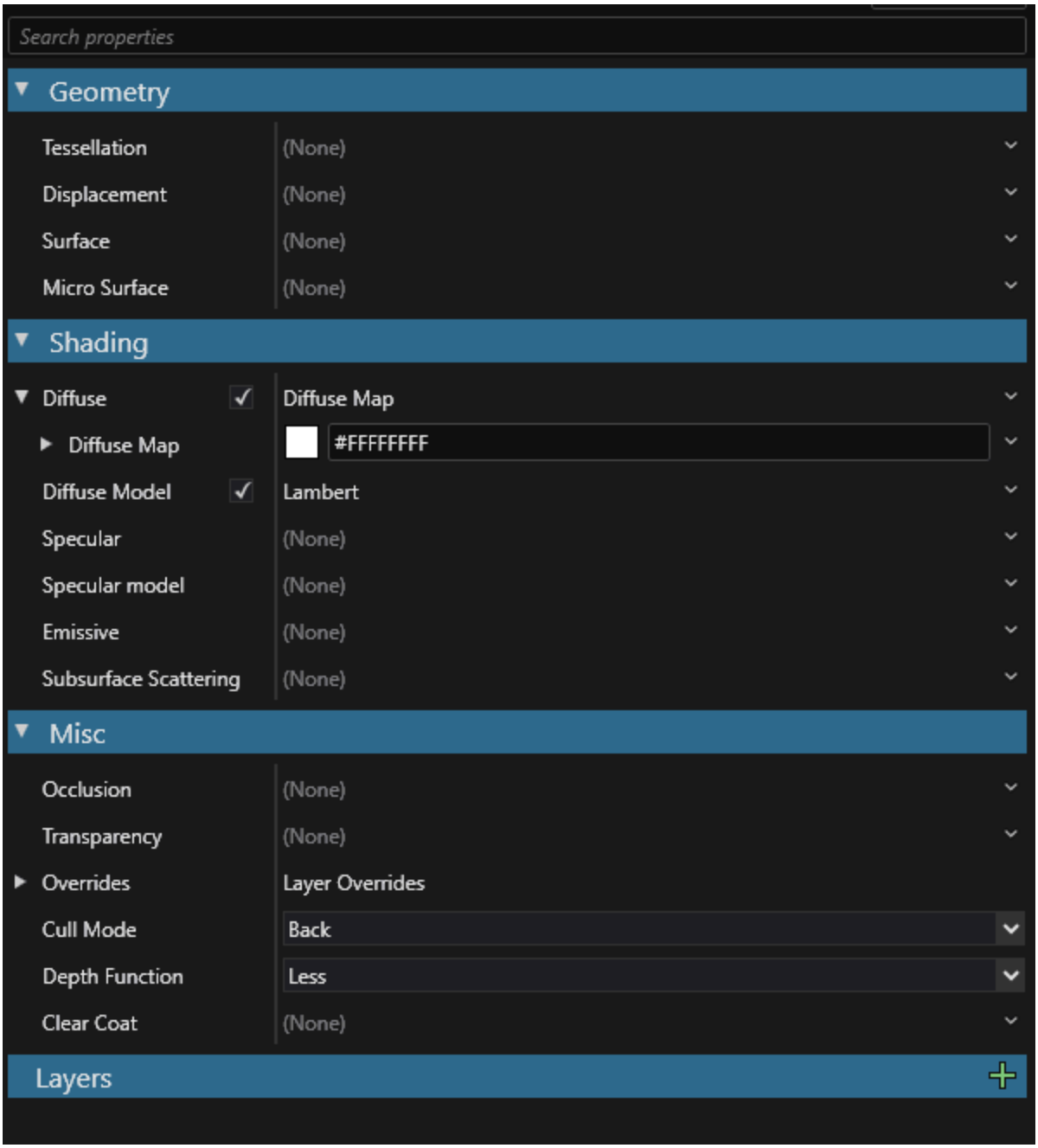


- The interface IMaterialDescriptor is the root interface for a material description.
- The IMaterialShaderGenerator is the main interface used to generate a material shader of the material.
- Each attribute and layer implements this interface to modify the final material shader.
- The MaterialDescriptor is the editor-time description of the material before being compiled into a material shader.
- The Material class is the runtime material shader generated from the MaterialDescriptor

Modifying parameters at runtime

The file MaterialKeys contains most material keys you might need to use, have a look through it to figure out which one you might need to get to modify the parameter you are interested in.

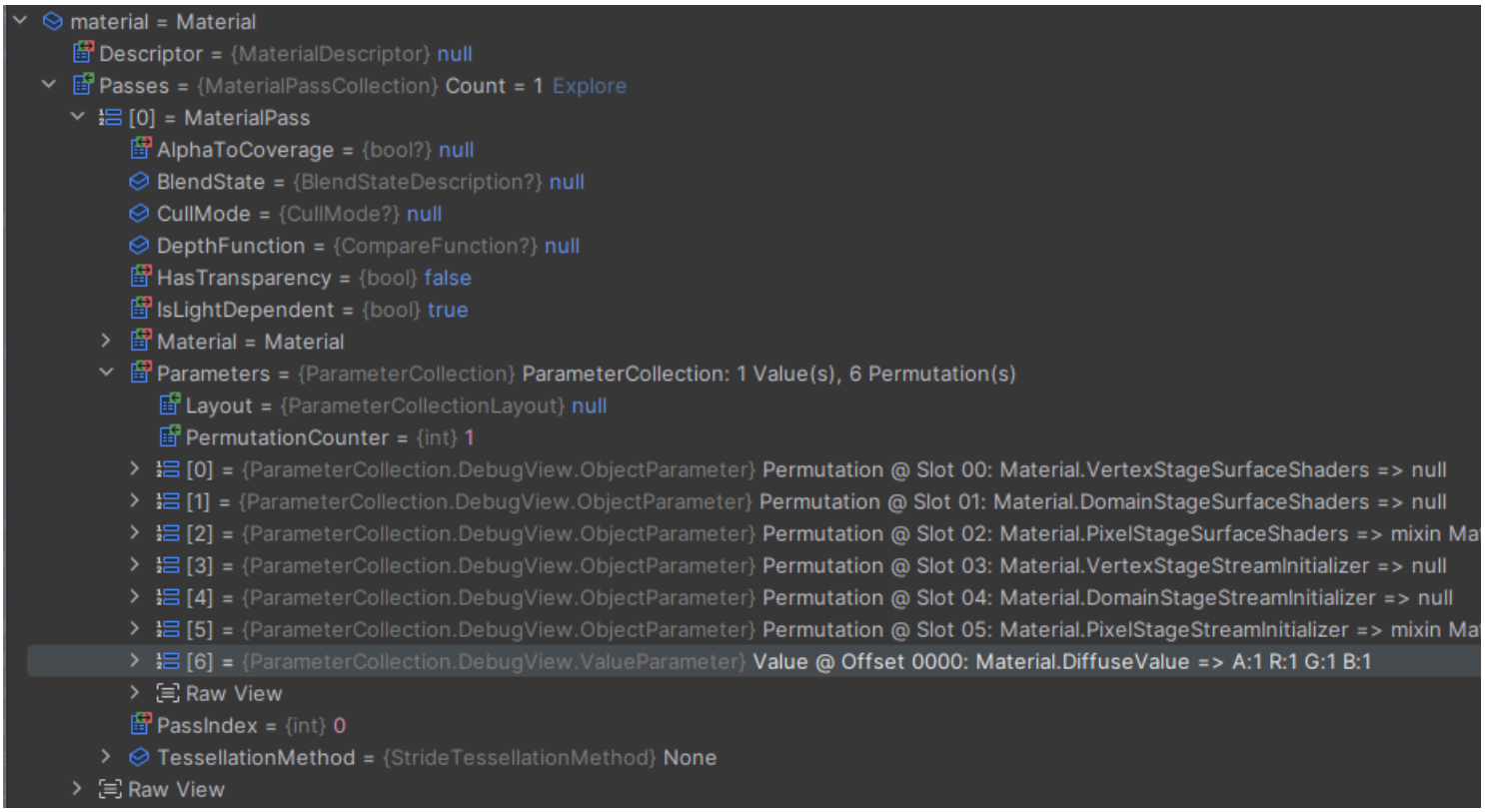
Let's say you have this fairly simple material:



And you want to clone that material, but change its color to red at runtime. Searching through the different keys contained in MaterialKeys you would find `MaterialKeys.DiffuseValue` and use it as the key to set the new color value you want:

```
var clone = SerializerExtensions.Clone(MyMaterial);
clone.Passes[0].Parameters.Set(MaterialKeys.DiffuseValue, new Color4(1, 0, 0));
```

If you aren't too sure which parameter keys your material uses, the best way to figure it out would be to inspect the material's variables with a debugger. Here's an example of that through Rider's *Threads & Variables* window:



See also

- Material maps
- Material attributes
- Material layers
- Material slots

Textures

Beginner Artist Programmer

Textures are images mainly used in materials. Stride maps textures to the surfaces the material covers.

Textures can add color information to a material — for example, to add a brick pattern to a wall or a wood pattern to a table. The values of the pixels in a texture (**texels**) can also be used for other calculations, such as in specular maps, metalness maps, or normal maps.

Materials typically contain multiple textures; for example, a material might contain a color texture, a normal map texture, and a roughness texture.

Textures can also be used outside materials; for example, you can draw them directly to the UI, or use them in sprites.

Supported file types

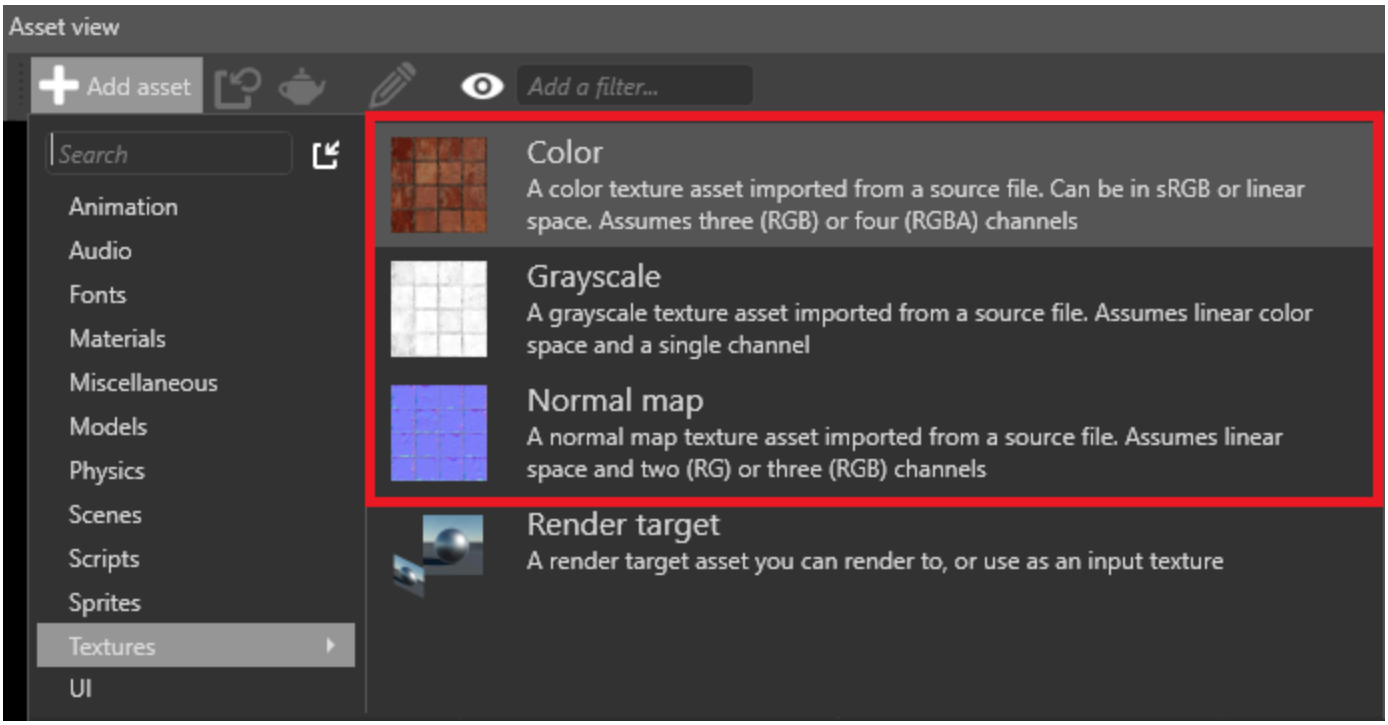
You can use the following file types as textures:

- `.dds`
- `.jpg`
- `.jpeg`
- `.png`
- `.gif`
- `.bmp`
- `.tga`
- `.psd`
- `.tif`
- `.tiff`

> **NOTE**
>
> - Stride only imports the first frame of animated image files, such as animated gifs or PNGs. They don't animate in Stride; they appear as static images.
> - Stride currently doesn't support movie files.

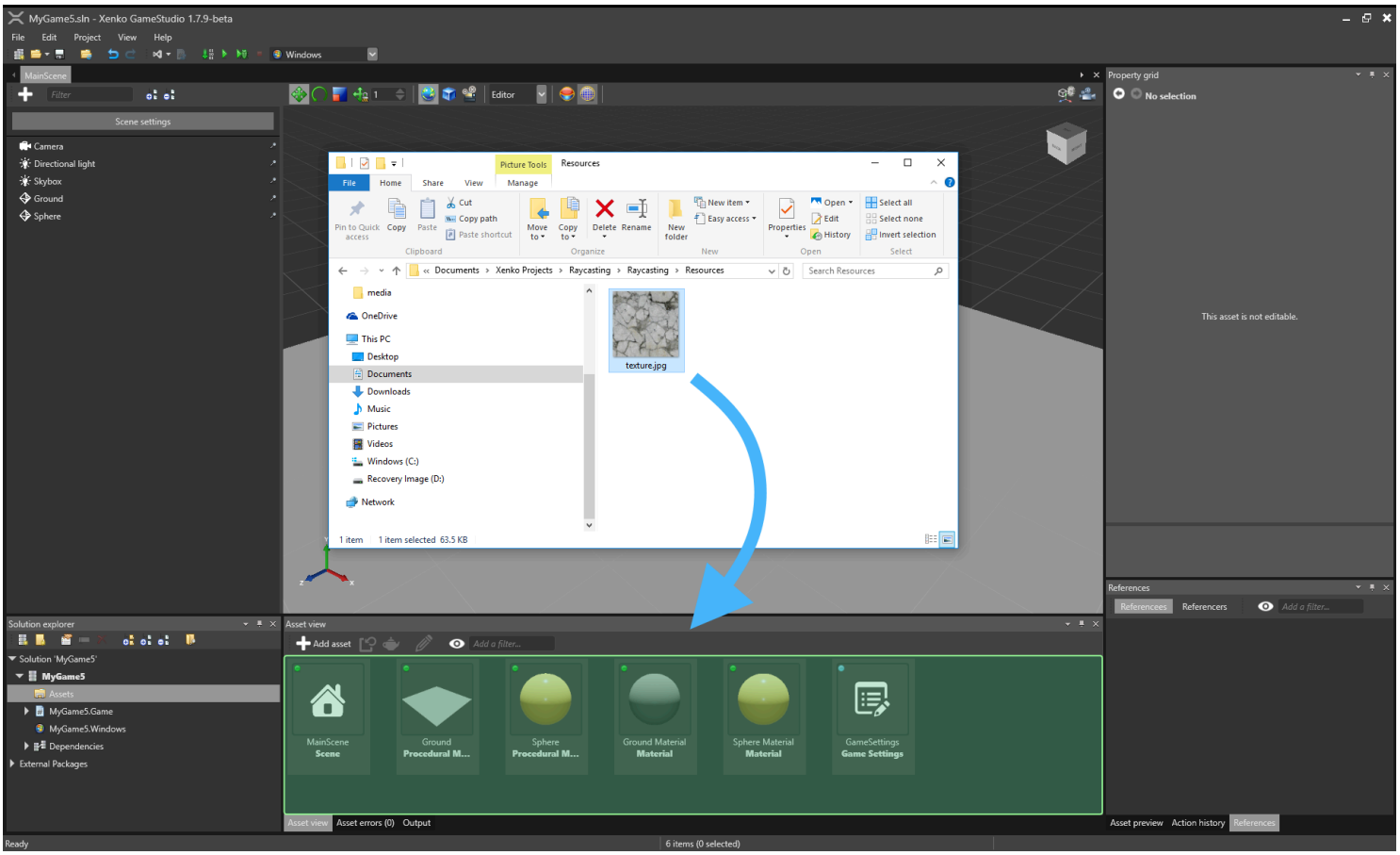
Add a texture

In the **Asset View**, click **Add asset** > **Texture**, then select a template for the texture (**color**, **grayscale** or **normal map**):

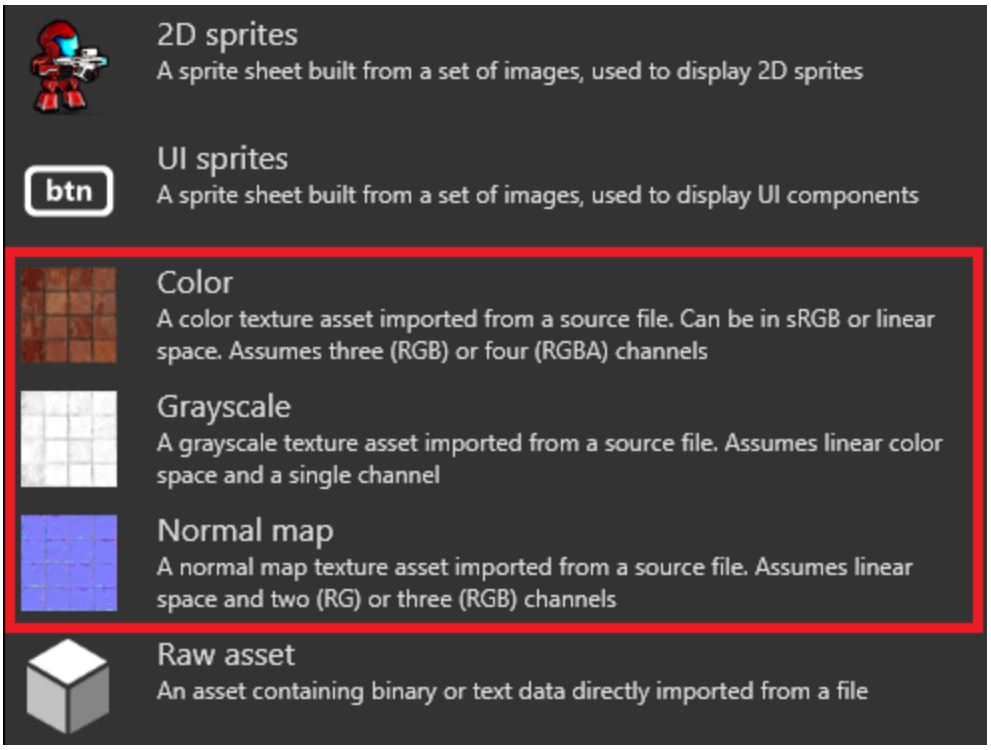


> **NOTE**
>
> Render targets are a different kind of texture, and don't use images. Instead, they render the output from a camera. For more information, see Render targets.

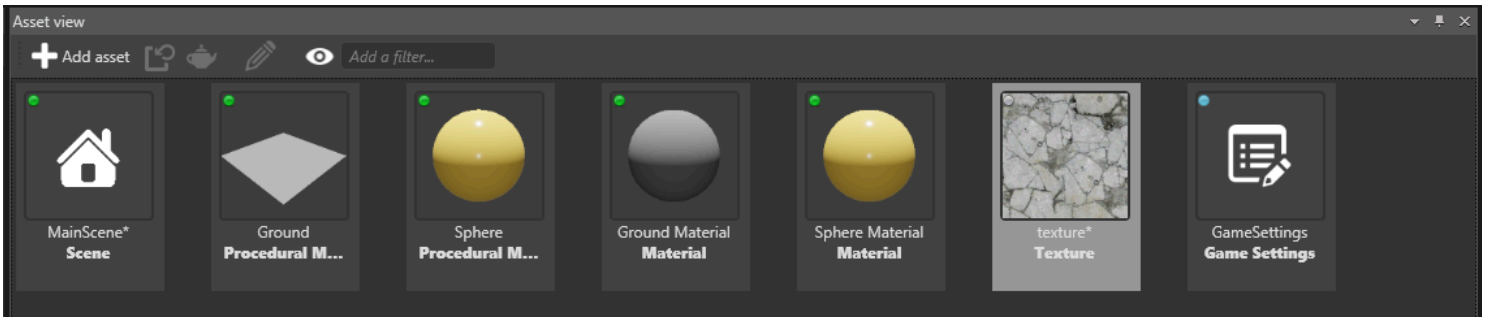
Alternatively, drag the texture file from Explorer to the Asset View:



Then select a texture template (**color**, **grayscale** or **normal map**):

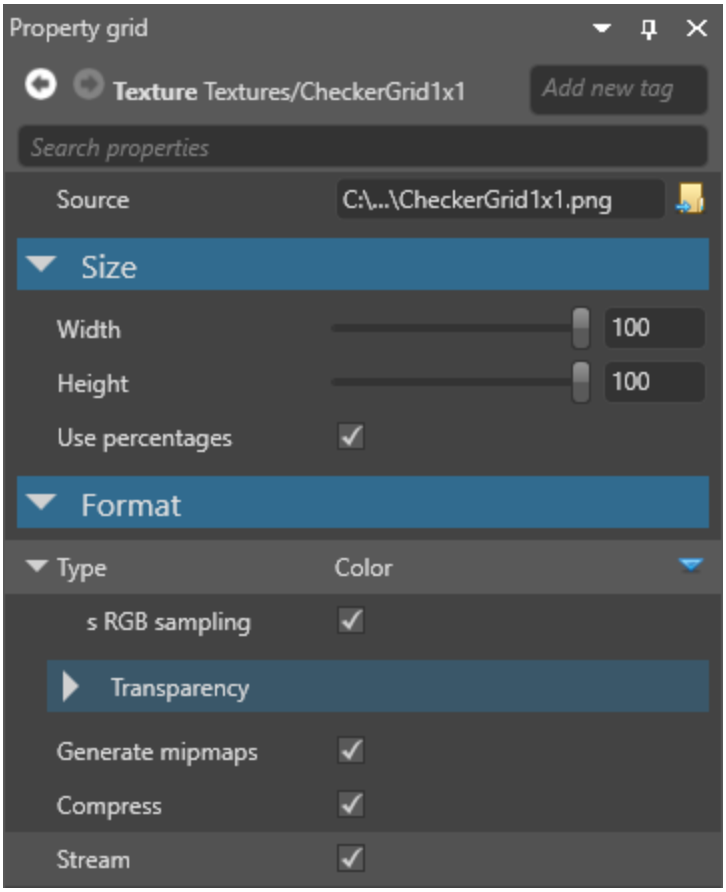


Game Studio adds the texture to the Asset View:



Texture properties

The following properties are common to all textures.

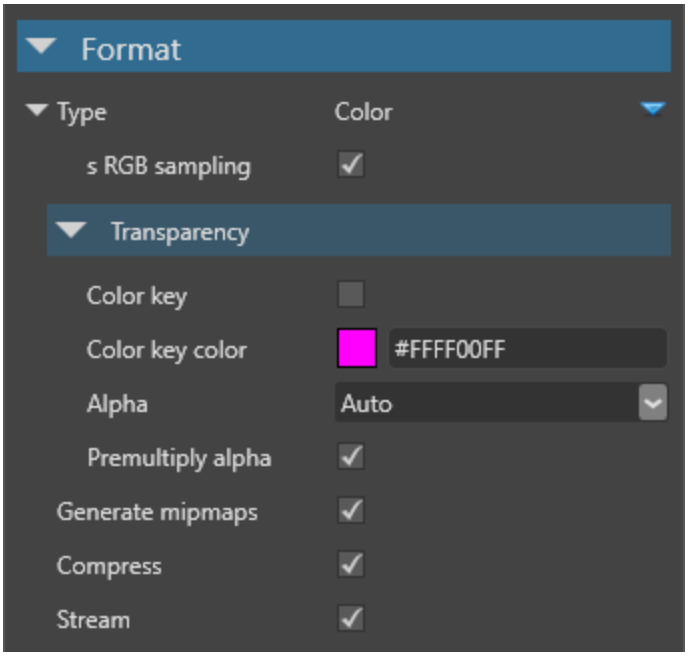


Property	Description
Width	The width of the texture in-game
Height	The height of the texture in-game
Use percentages	Use percentages for width and height instead of actual pixel size
Type	Use **Color** for textures you want to display as images, **Normal map** for normal maps, and **Grayscale** to provide values for other things (eg specular maps, metalness maps,

Property	Description
	roughness maps). Color textures and normal maps have additional properties (see below).
Generate mipmaps	Generate different versions of the texture at different resolutions to be displayed at different distances. Improves performance, removes visual artifacts, and reduces pop-in when using **streaming**, but uses more memory. Unnecessary for textures always at the same distance from the camera (such as UIs).
Compress	Compress the final texture to a format based on the target platform and usage. The final texture is a multiple of 4. For more information, see Texture compression.
Stream	Stream the texture dynamically at runtime. This improves performance and scene loading times. Not recommended for important textures you always want to be loaded, such as splash screens. For more information, see Streaming.

Color texture properties

The following properties apply if you set the texture **type** to **color**.

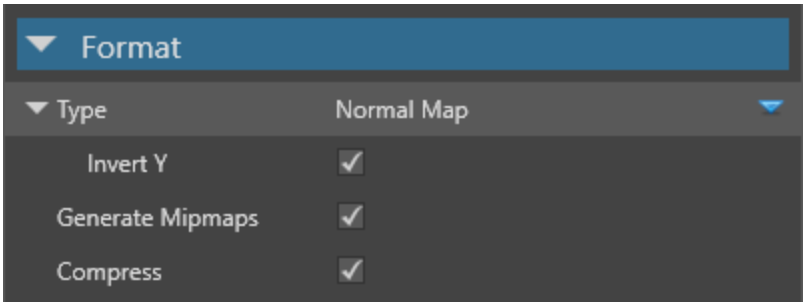


Property	Description
sRGB sampling	Store the texture in sRGB format and convert to linear space when sampled. Recommended for all color textures, unless they're explicitly in linear space.
Color key enabled	Use the color set in the **Color key color** property for transparency at runtime. If disabled, the project uses transparent areas of the texture instead

Property	Description
Color key color	The color used for transparency at runtime. Only applied if **Color key enabled** is selected.
Alpha	The texture alpha format (**None**, **Mask**, **Explicit**, **Interpolated**, or **Auto**)
Premultiply alpha	Premultiply all color components of the images by their alpha component

Normal map properties

The following property applies if you set the texture **type** to **normal map**.



Property	Description
Invert Y	Have positive Y-component (green) face up in tangent space. This depends on the tools you use to create normal maps.

For more information about normal maps, see the Normal maps page.

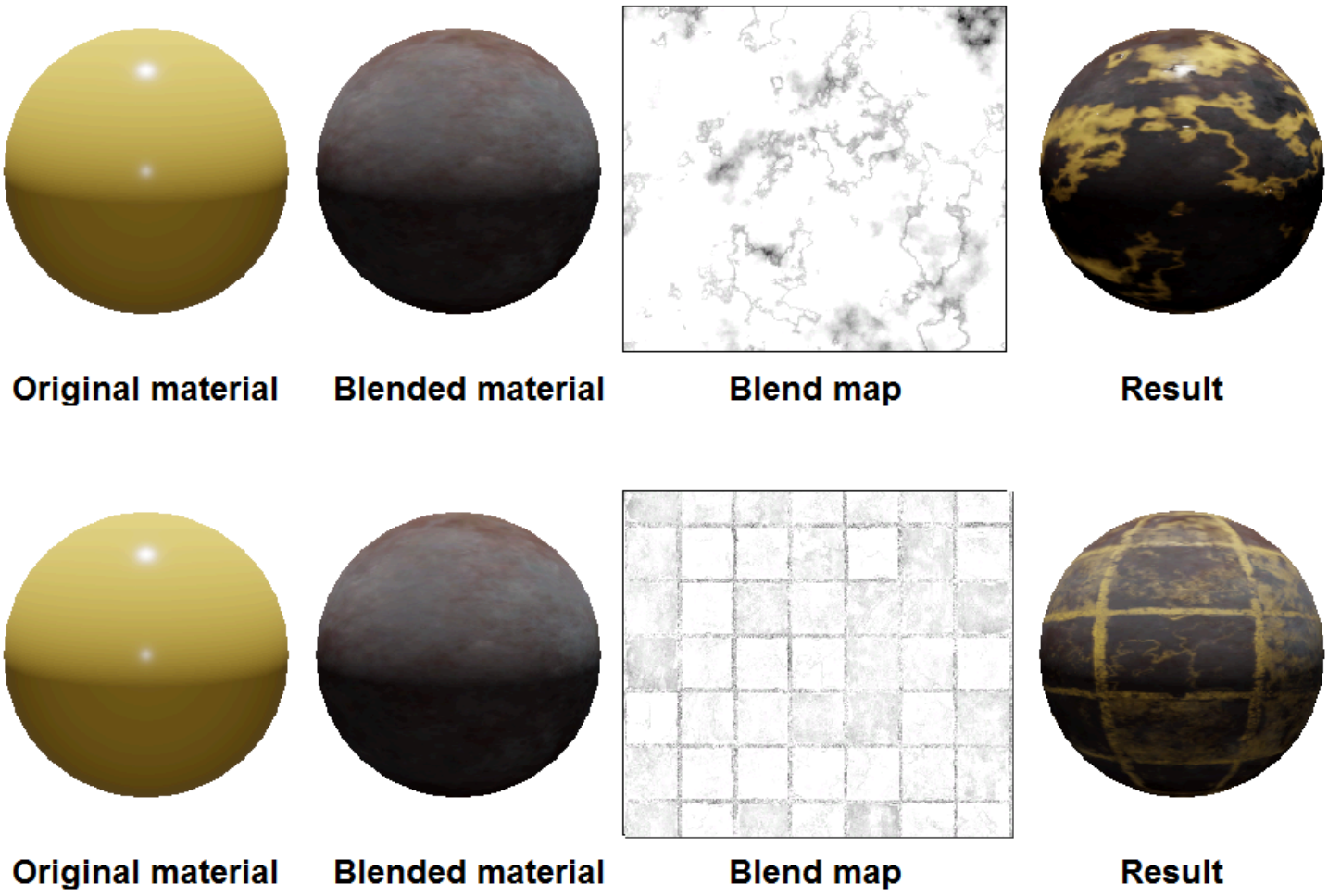
Grayscale textures

Grayscale texture use only the R channel of the image (finalRGBA = originalRRRR).

> **NOTE**
> If you add a texture to a scene (as a sprite component), and set the texture type to grayscale, it appears red, not monochrome. This is because the image uses the R (red) channel.

To make the channel monochrome, in the **Sprite** component properties, set the **Type** as **Grayscale**. For more information about the sprite component properties, see Use sprites.

You can use grayscale textures to provide values in material maps. For example, you can use a texture as a **blend map** to blend two material layers:

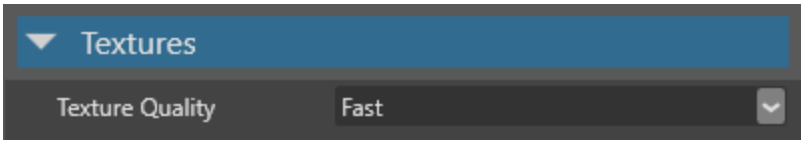


Note how the blend map texture corresponds to the patterning on the result.

For more information, see Material maps.

Global texture settings

For instructions about how to access the global texture settings, see the Game Settings page.



Property	Description
Texture quality	The texture quality when encoding textures. **Fast** uses the least CPU, but has the lowest quality. Higher settings might result in slower builds, depending on the target platform.

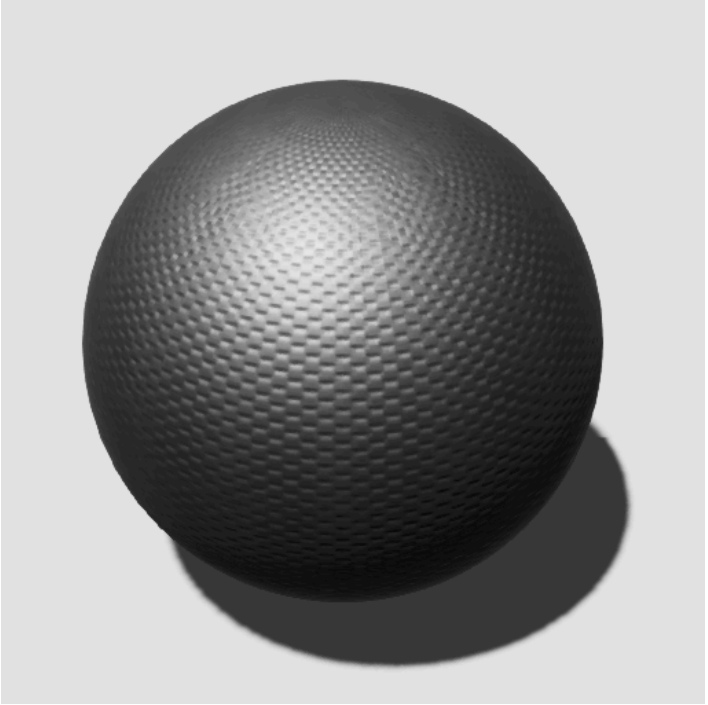
See also

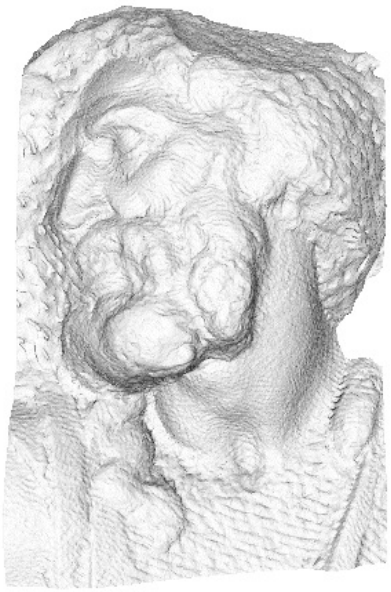
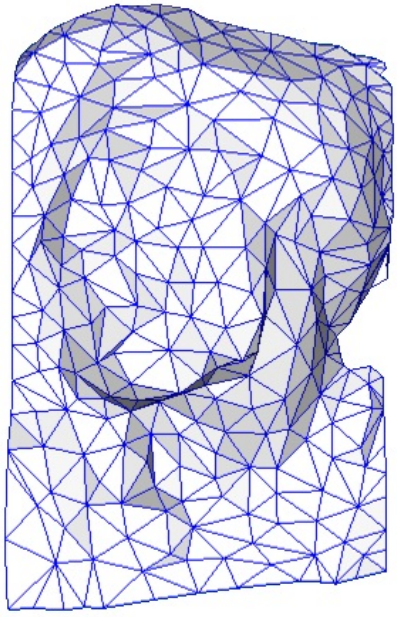
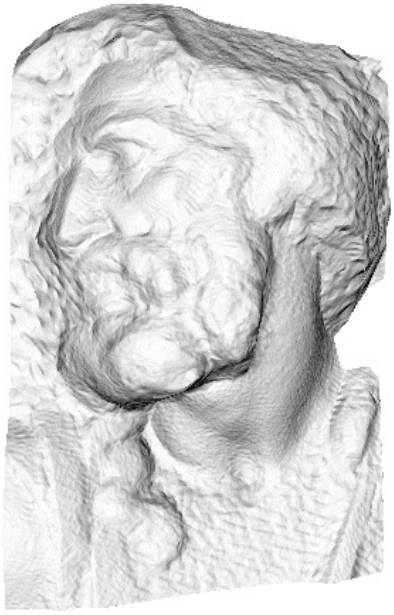
- Normal maps
- Texture compression
- Texture streaming
- Materials
- Sprites
- Render textures
- Skyboxes and backgrounds

Normal maps

Intermediate Artist Programmer

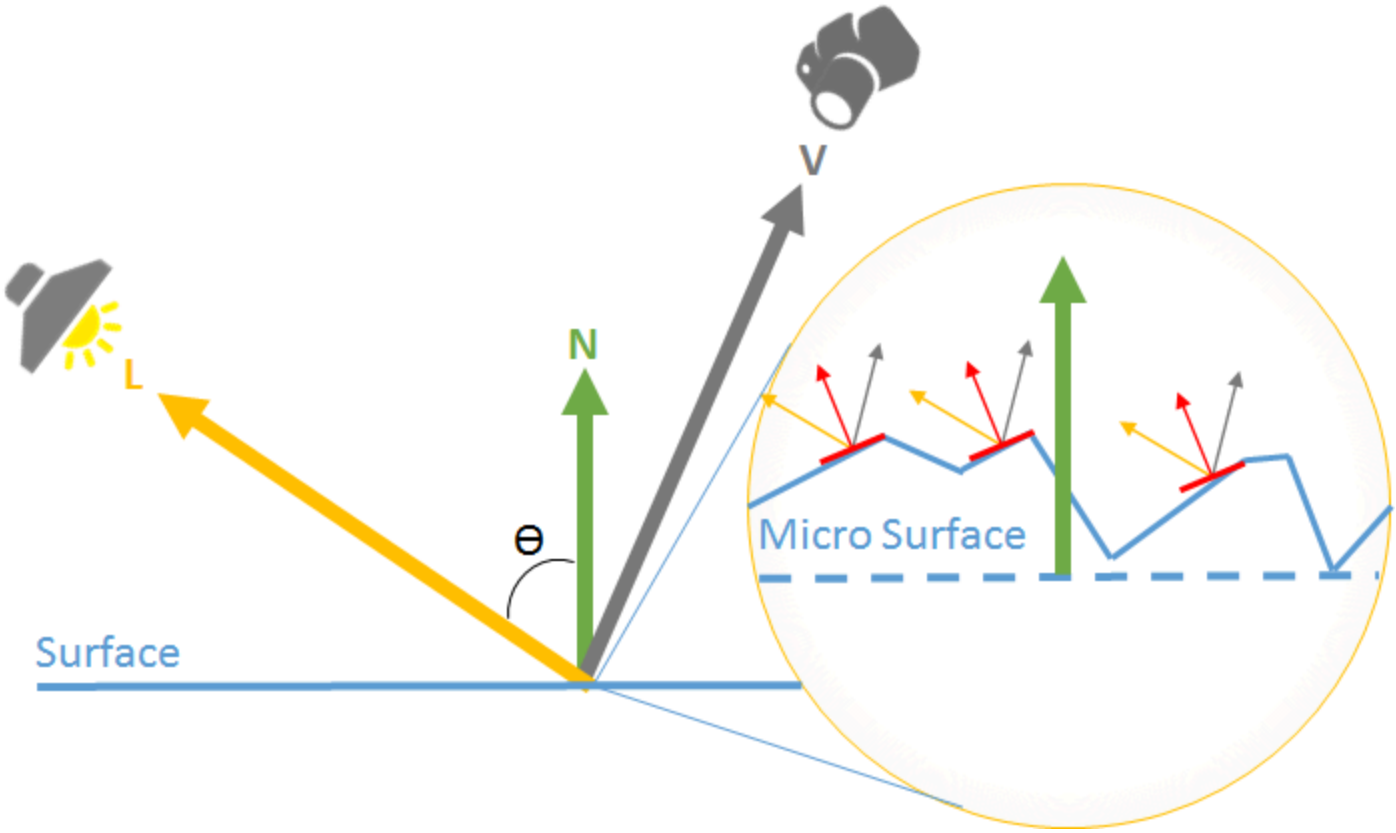
Normal maps are textures that add the appearance of surface detail, such as cracks and bumps, without changing the actual geometry of a model. They contain information about how meshes should reflect light, creating the illusion of much more complex geometry. This saves lots of processing power.

No normal map	With a normal map
	

Original mesh	Simplified mesh	Simplified mesh and normal map
		
4m triangles	500 triangles	500 triangles

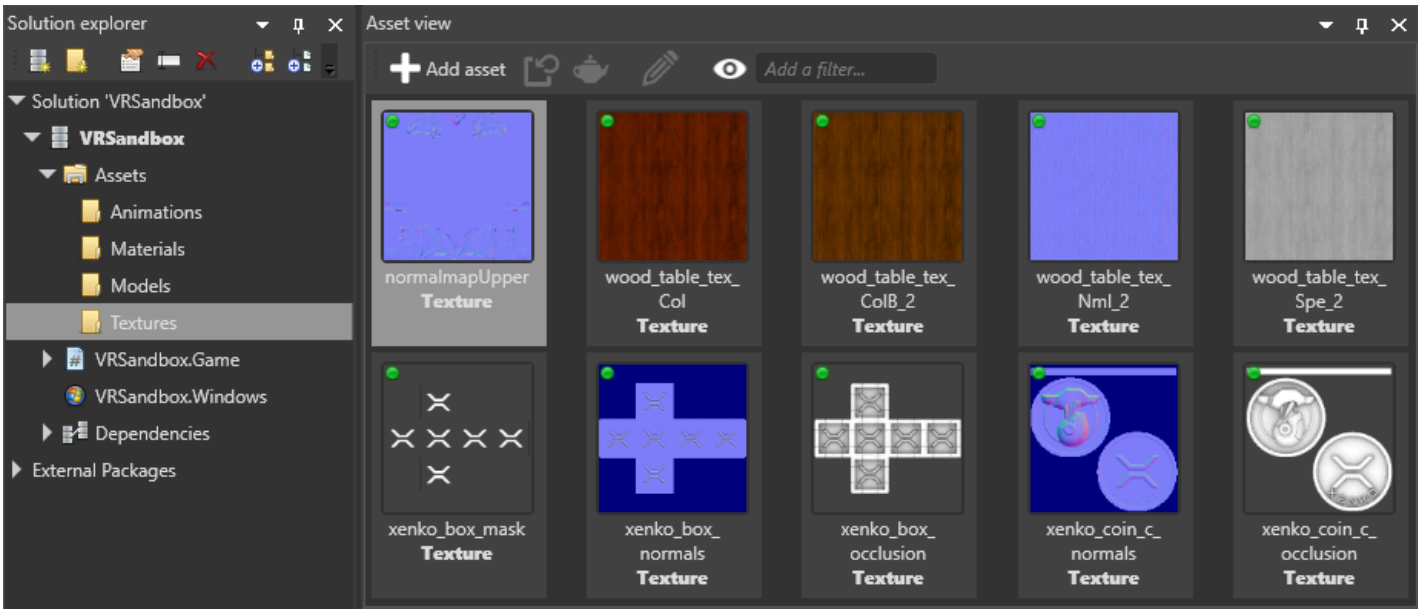
(Images courtesy of Paolo Cignoni, shared under Attribution-ShareAlike 1.0 Generic (CC BY-SA 1.0)

Normal maps usually represent small changes of the normal vector (the vector which points away from the surface). Stride uses the most common convention: the X and Y components follow the tangent and the bitangent of the surface, and the Z component follows the normal vector of the surface. This means that a value of `(0, 0, 1)` coincides with the normal vector and represents no change, while a value of `(-1, 0, 0)` tilts to the "left" (ie negative X value in the tangent (local) space).

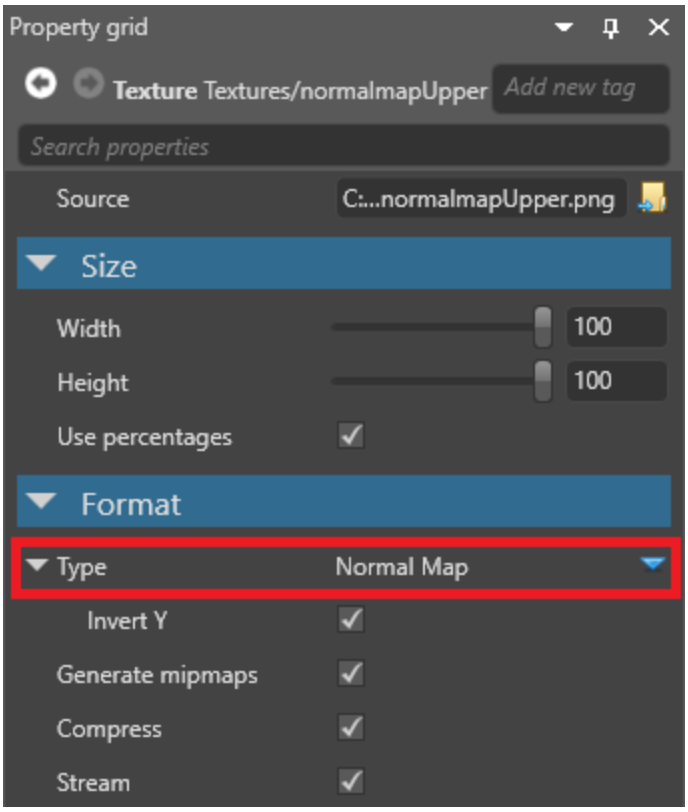


Use a normal map

1. In the **Asset View**, select the texture you want to use as a normal map.

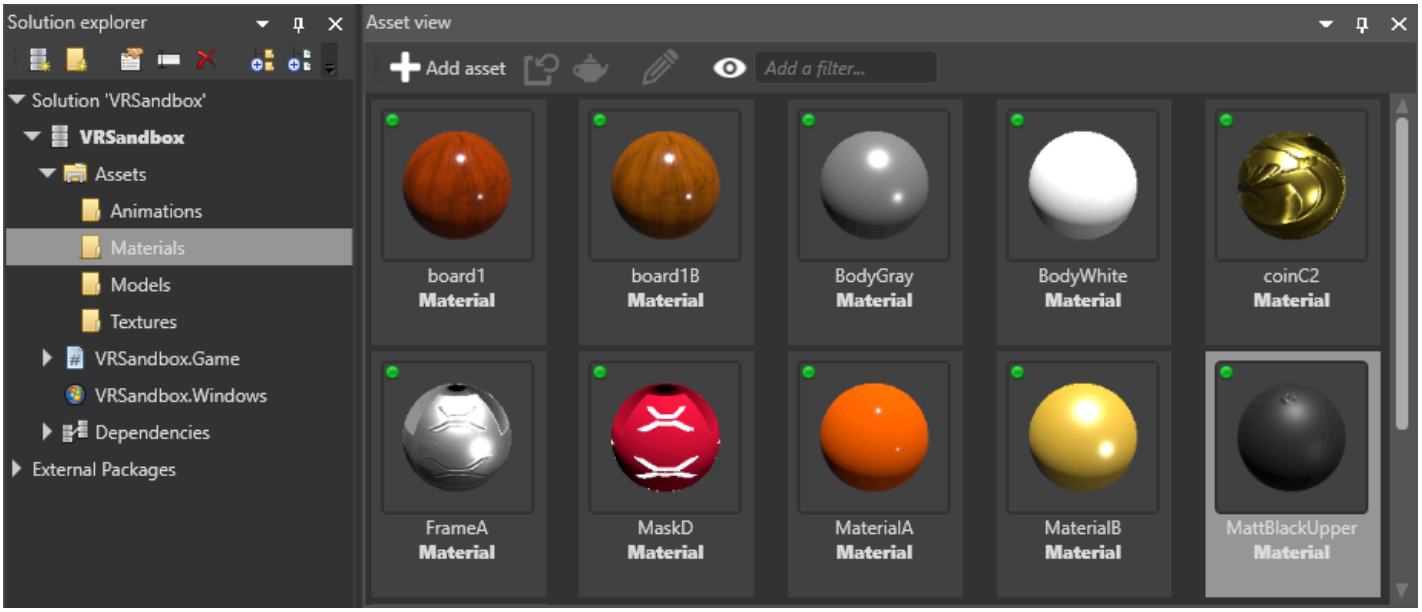


2. In the **Property Grid**, make sure the **type** is set to **normal map**.

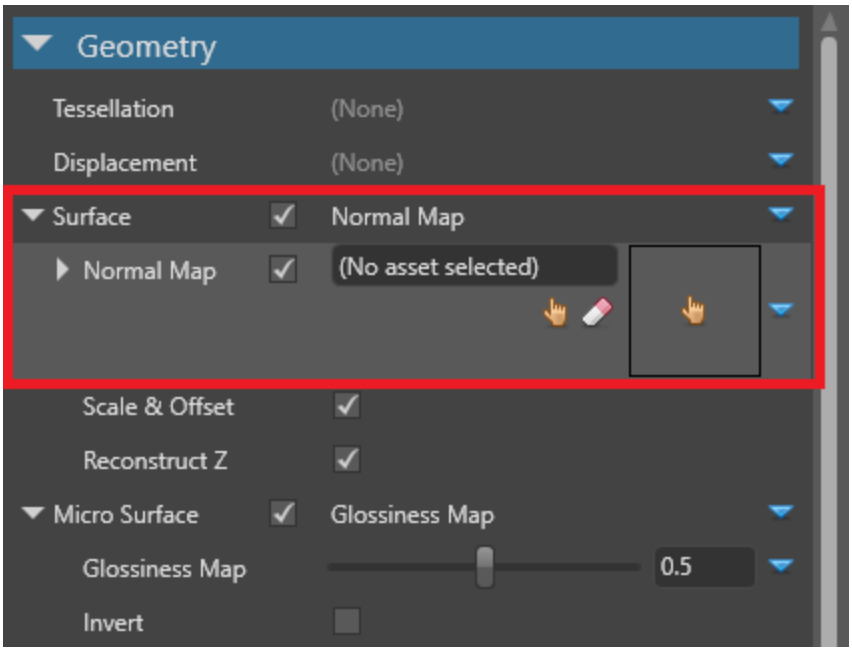


This means Stride assumes the texture is in linear color space and converts it to a format suited for normal maps.

3. In the **Asset View**, select the material you want to use the normal map.

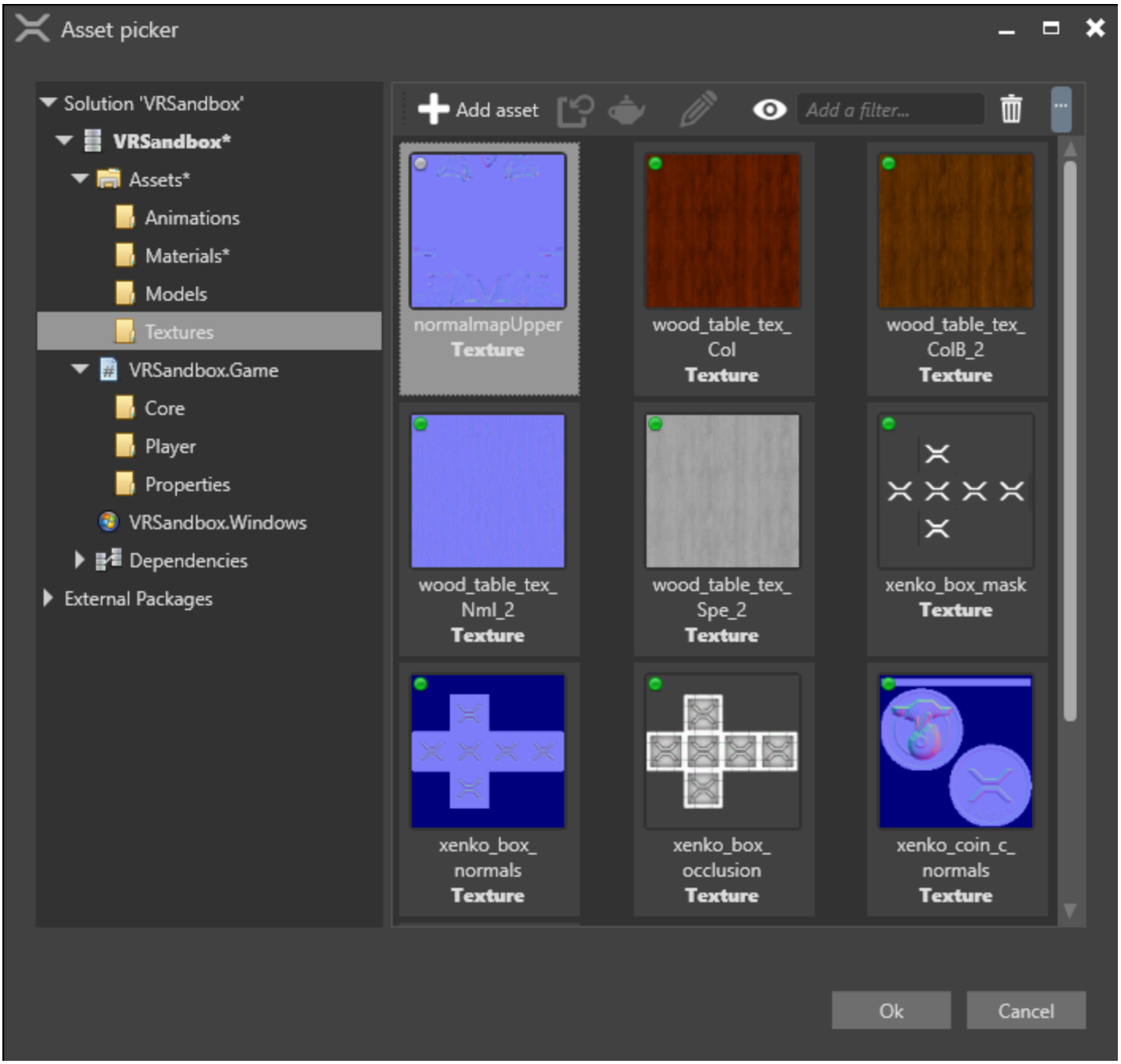


4. In the **Property Grid**, under the material **Geometry** properties, expand **Surface**.



5. Next to **Normal map**, click (**Replace**) and make sure **Texture** is selected.

6. Next to **Normal map**, click (**Select an asset**).

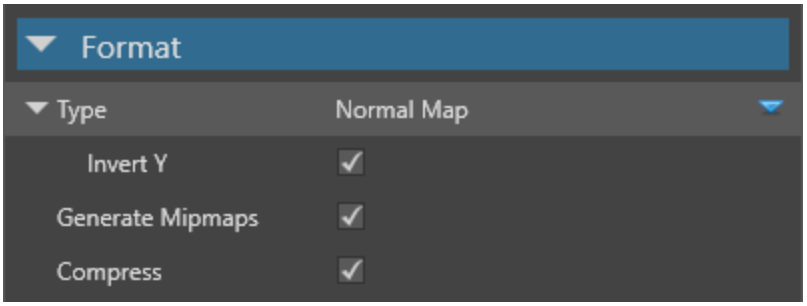


7. Select the normal map texture and click **OK**.

For more information about materials, see Materials.

Normal map properties

Normal map textures have two properties in addition to the common texture properties.



Property	Description
Invert Y	Have positive Y components (green pixels) face up in tangent space. This option depends on the tools you use to create normal maps.

For information about normal map properties in materials, see Materials — Geometry attributes.

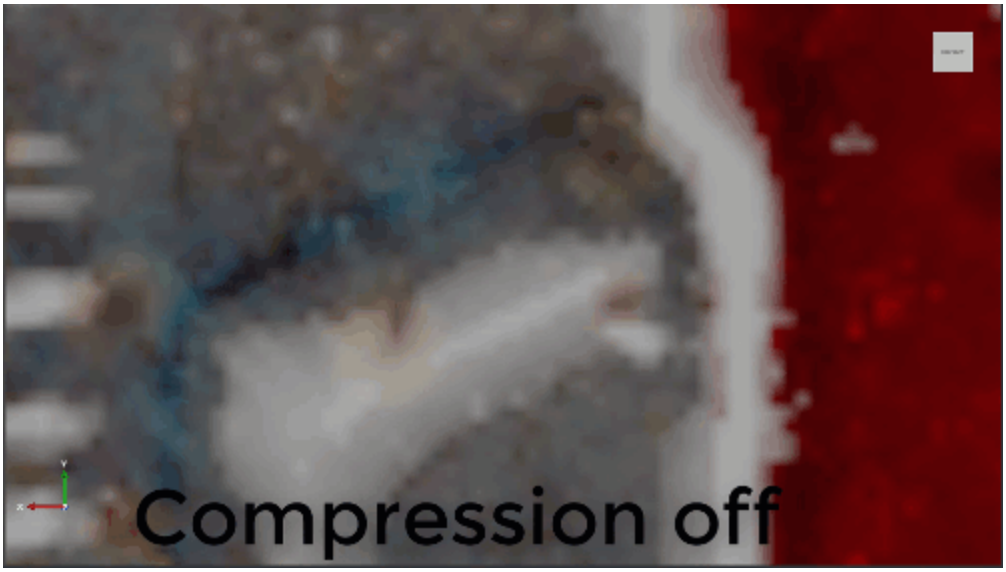
See also

- Textures
- Materials
- Normal mapping on Wikipedia

Texture compression

Beginner Artist Programmer

Compressed textures use up to 50% less space and are faster to load. Compression produces results similar to JPEG compression. The animation below was recorded with the camera positioned extremely close to the texture; at normal distances, the difference isn't noticable.



For color textures, compression is best used for visually busy images, where the effects are less noticeable. You probably don't want to compress textures with fine edges, such as logos used in splash screens.

Compression converts the texture to a multiple of 4. If the texture isn't already a multiple of 4, Stride expands it.

Compression removes data from the image based on the texture type:

Texture type	Compression
Color	Compresses all RGBA channels. If the `Alpha` property is set to `None` in the texture properties, the alpha channel is removed
Grayscale	Removes all RGBA channels except red
Normal map	Removes the blue and alpha channels (alpha isn't used in normal maps anyway). The blue channel is reconstructed from the red and green channels (assuming a pixel has a vector length of 1)

- Textures index

- Normal maps
- Materials
- Sprites
- Render textures

Streaming

Beginner Artist Programmer

When you **stream** textures, Stride only loads them when they're needed. This significantly decreases the time it takes to load a game or scene, uses less memory, and makes your game easier to scale.

> **NOTE**
> Currently, only textures can be streamed.

How Stride streams textures

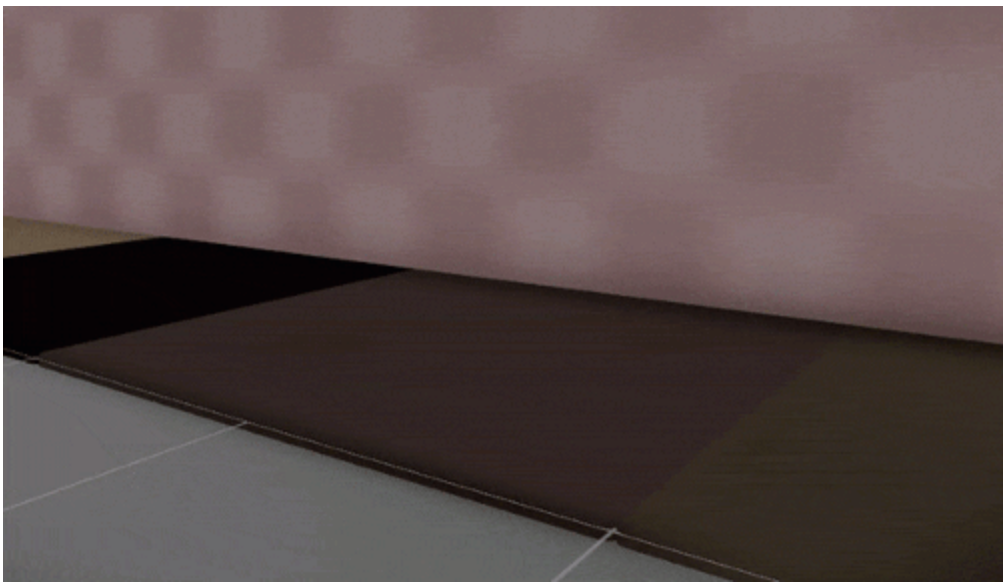
Instead of loading a texture when Stride loads the scene (with all its mipmaps), Stride only loads it when it's used (eg a model using the texture is onscreen).

When the texture is no longer needed (ie no objects that use the texture are onscreen), Stride unloads it.

Currently, there's no loading priority for textures. For example, Stride doesn't load textures based on distance; instead, Stride loads them all in sequence.

Using streaming with mipmaps

If mipmaps (different-resolution versions of textures displayed at different distances) are enabled in the texture properties, the lower-resolution mipmaps load first, as they're smaller in size. The gif below shows this process happening in slow motion.



In most situations, the process is very quick. We recommend you enable mipmaps for streaming as it means lower-resolution versions of textures act as placeholders until the higher-quality versions can

load, reducing pop-in.

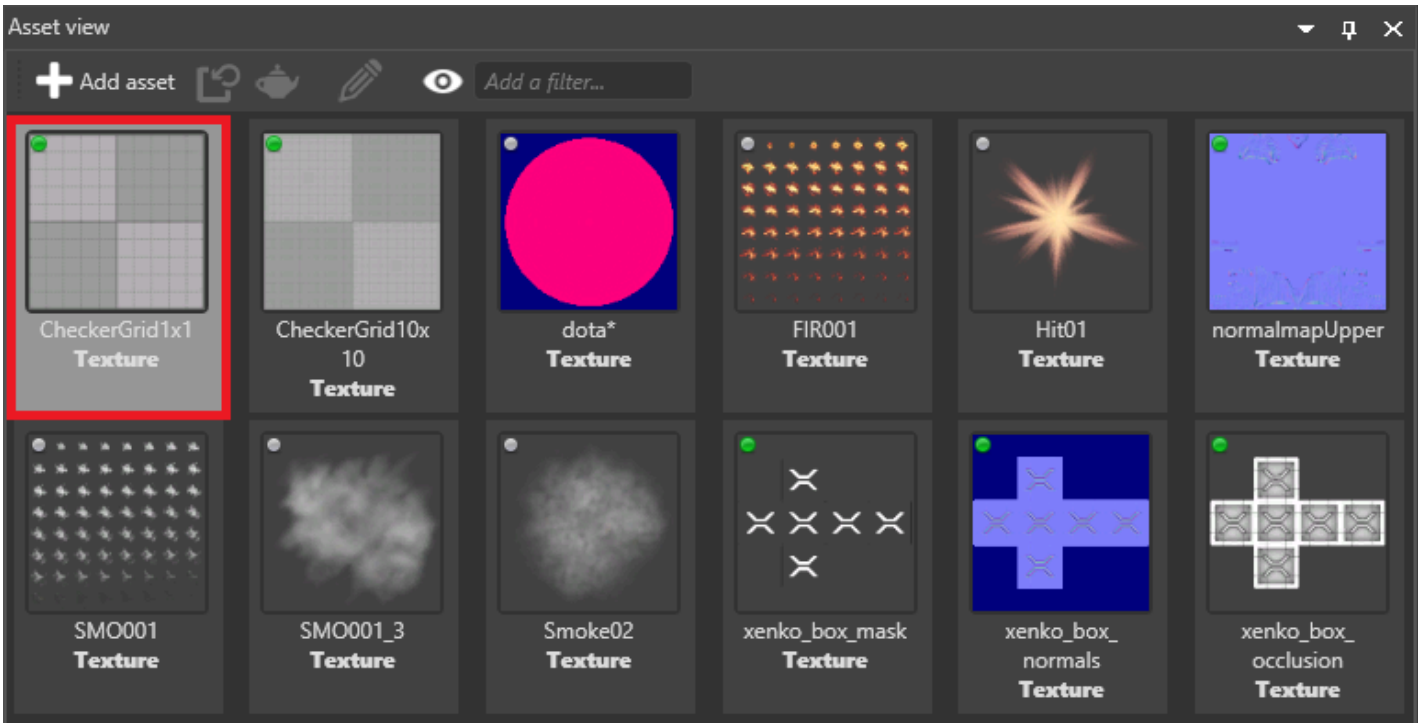
When **not** to use streaming

Streaming is enabled by default for all textures. You might want to disable streaming on important textures you always want to display immediately and in high quality, such as:

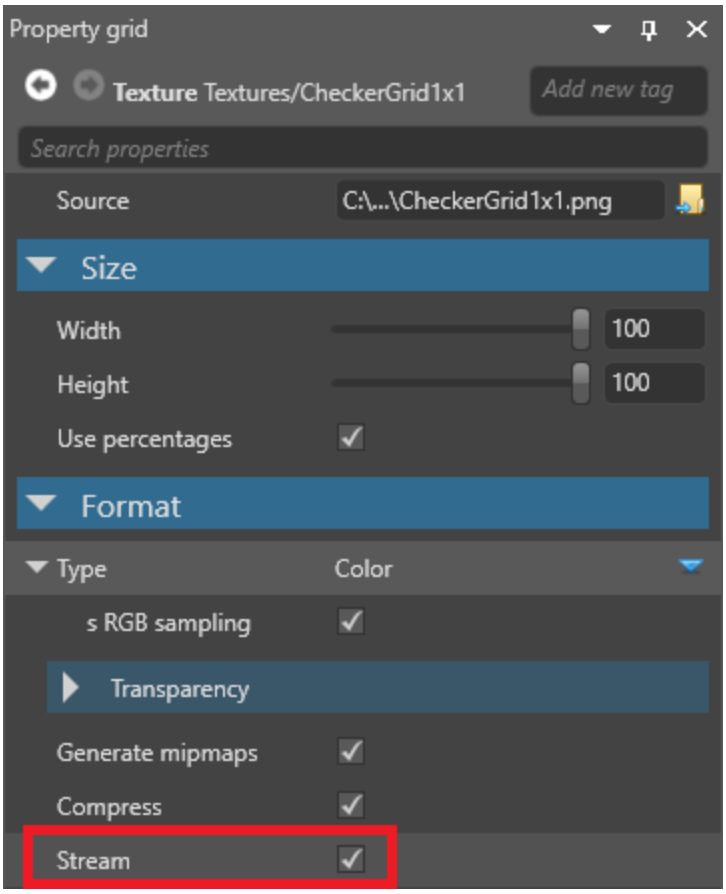
- splash screens
- textures on player models
- textures used in particles (particles often have a short lifespan, so might disappear before the texture loads)

Enable or disable streaming on a texture

1. In the **Asset View**, select the texture.



2. In the **Property Grid**, under **Format**, use the **Stream** check box.

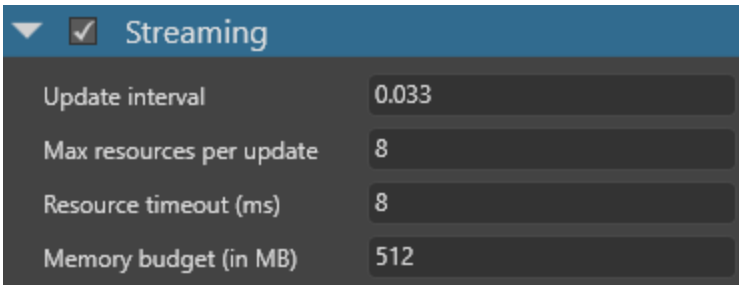


Global streaming settings

You can access the global streaming settings in the Game Settings asset. These settings apply to all textures that have streaming enabled.

For instructions about how to access the global streaming settings, see the Game Settings page.

Properties



Property	Description
Streaming	Enable streaming
Update interval	How frequently Stride updates the streaming. Smaller intervals mean the streaming system reacts faster, but use more CPU and cause more memory fluctuations.

Property	Description
Max resources per update	The maximum number of textures loaded or unloaded per streaming update. Higher numbers reduce pop-in but might slow down the framerate.
Resource timeout (ms)	How long resources stay loaded after they're no longer used (when the **memory budget** is exceeded)
Memory budget (in MB)	When the memory used by streaming exceeds this budget, Stride unloads unused textures. You can increase this to keep more textures loaded when you have memory to spare, and vice versa.

Access the streaming manager in code

Use Streaming.

For example, to disable streaming globally, use:

```
Streaming.EnableStreaming = false;
```

To start streaming a texture:

```
Streaming.StreamResources(myTexture);
```

To disable streaming at load time:

```
var texture = Content.Load<Texture>("myTexture",
ContentManagerLoaderSettings.StreamingDisabled);
```

Options

There are three StreamingOptions:

- The `KeepLoaded` option keeps the texture in memory even when the memory budget is exceeded.
- If mipmaps are enabled, the `ForceHighestQuality` option loads only the highest-quality version of the texture.
- The `KeepLoaded` option keeps the texture in memory even when it's not used.

For example:

```
var myOptions = new StreamingOptions() { KeepLoaded = true };
Streaming.StreamResources(myTexture, myOptions);
```

To change the `StreamingOptions` at runtime, use `SetResourceStreamingOptions`. For example:

```
var myNewOptions = new StreamingOptions() { KeepLoaded = false };
Streaming.SetResourceStreamingOptions(myTexture, myNewOptions);
```

See also

- StreamingManager API
- Textures index
- Texture compression
- Game Settings

Skyboxes and backgrounds

Beginner Designer Programmer

Skyboxes are backgrounds that create the illusion of space and distance. Typical skybox backgrounds include skies, clouds, mountains, and other scenery. As skyboxes are prerendered, they require little GPU and CPU.

You can use **cubemaps** or **360° panoramic textures** as skyboxes. You can also use them to light the scene.

> **NOTE**
> Currently, Stride doesn't support skydomes or local skyboxes.

Alternatively, you can display a **2D background**, which is often useful for 2D games.

Cubemaps

A **cubemap** is a six-sided texture. When these textures are assembled in a cube around the scene, the cubemap simulates spacious 3D surroundings.

Currently, Game Studio can't convert image files to cubemaps (`.dds` files). Use another application to create a cubemap from separate image files, such as:

- Nvidia conversion tool
- ATI conversion tool

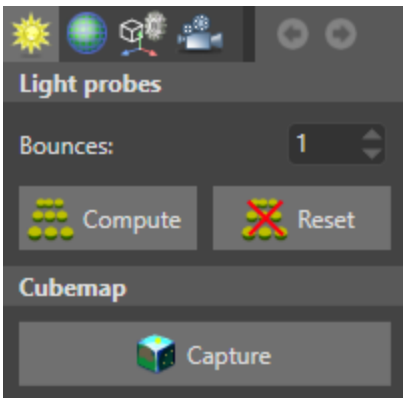
Create a cubemap in Game Studio

You can capture a cubemap from a position in your scene.

1. In the **scene editor**, position the camera at the point where you want to capture the cubemap. The direction the camera faces doesn't matter, only the position.

   Typically, you should capture cubemaps at the center of your scene to create the best all-round view.

2. In the scene editor toolbar, open the **Lighting options** menu.



3. Under **Cubemap**, click **Generate**.

4. Browse to the location on disk you want to save the cubemap, specify a name, and click **Save**.

> **TIP**
>
> We recommend you save the cubemap in your project **Resources** folder. For more information, see Organize your files in version control.

Game Studio creates a cubemap `.dds` file in the location you specified.

360° panoramic textures

Instead of using a cubemap, you can use a **360° panoramic texture** as a 3D background.

360° panorama	Appearance in game

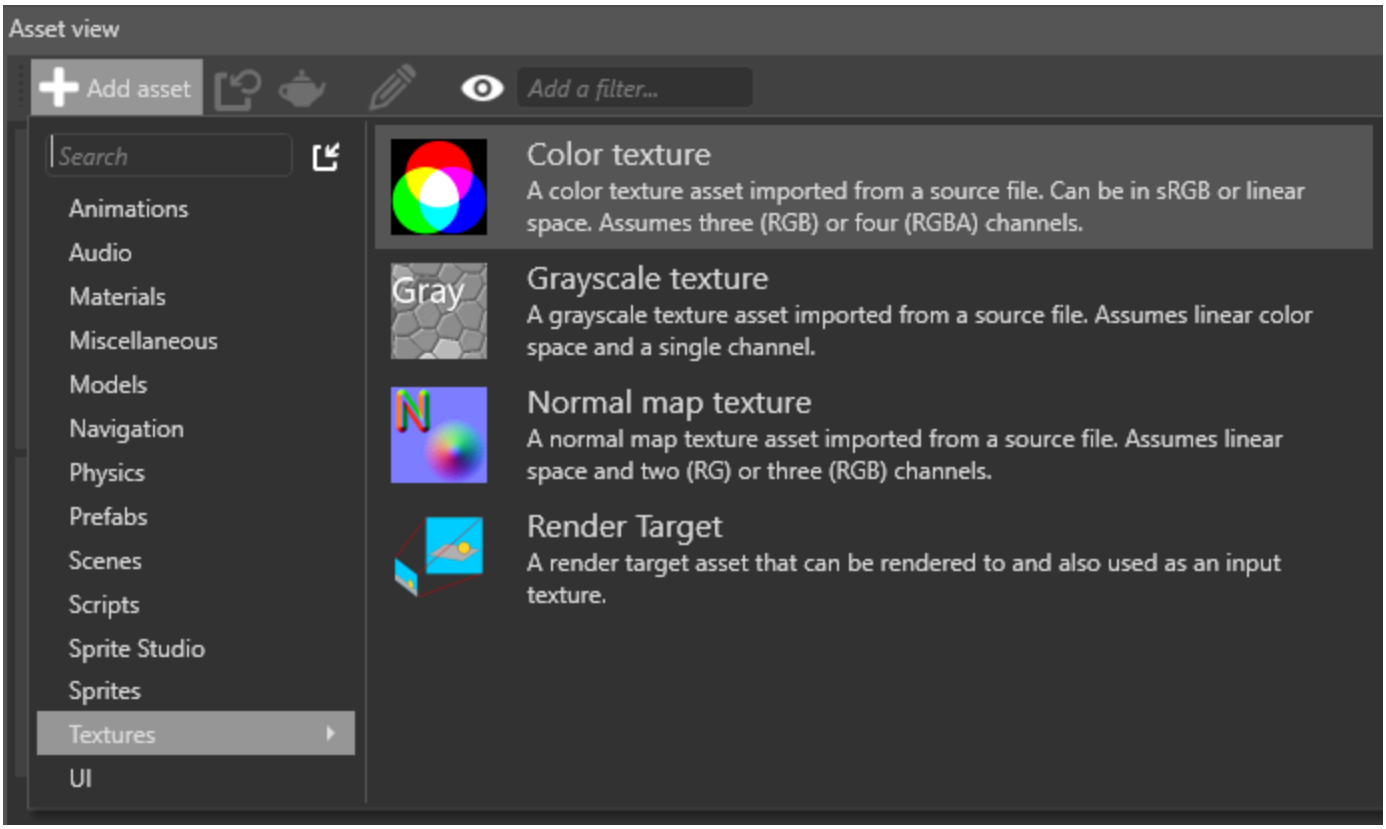
Image courtesy of Texturify

> **NOTE**
>
> Remember that post effects affect the appearance of your skybox. If it doesn't look how you expect, try changing your post effect settings.

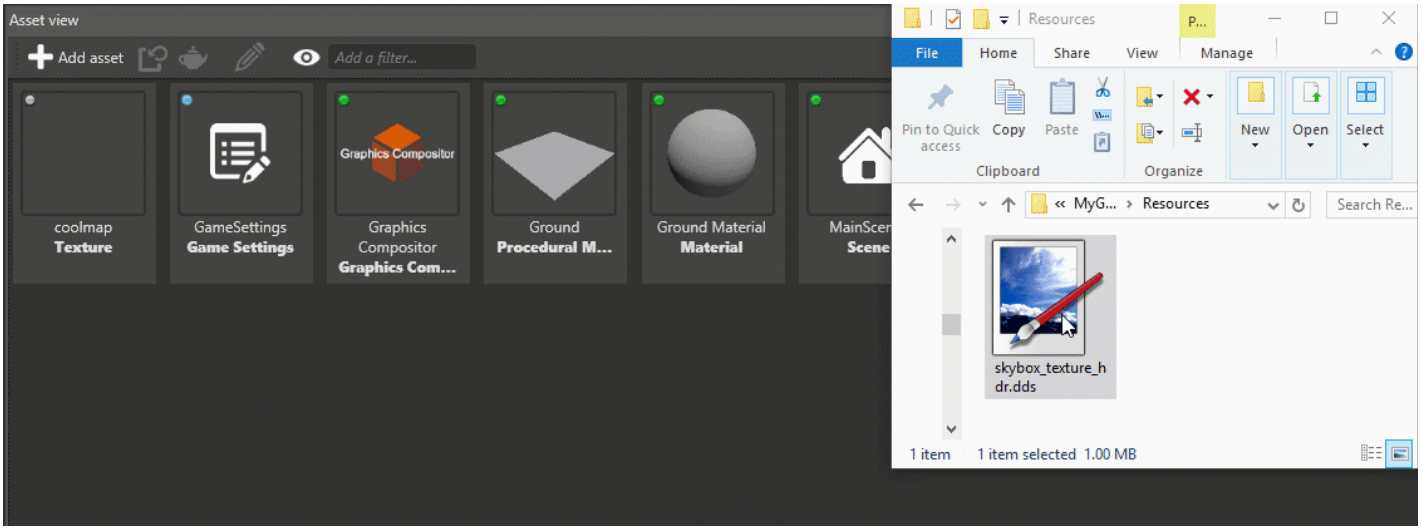
Add a cubemap or 360° panoramic texture to the project

You add these like other color textures.

- In the **Asset View**, click Add asset, select **Textures** > **Color texture**, and browse to the file.



- Alternatively, drag and drop the file from **Windows Explorer** to the **Asset View**, then select **Color texture**.



Create a skybox

To create a skybox, add a cubemap or 360° panoramic texture to a **background component**.

Stride includes an entity with a background component in the project by default. Only one background can be active in a scene at a time. If there are multiple backgrounds, Stride only loads the first.

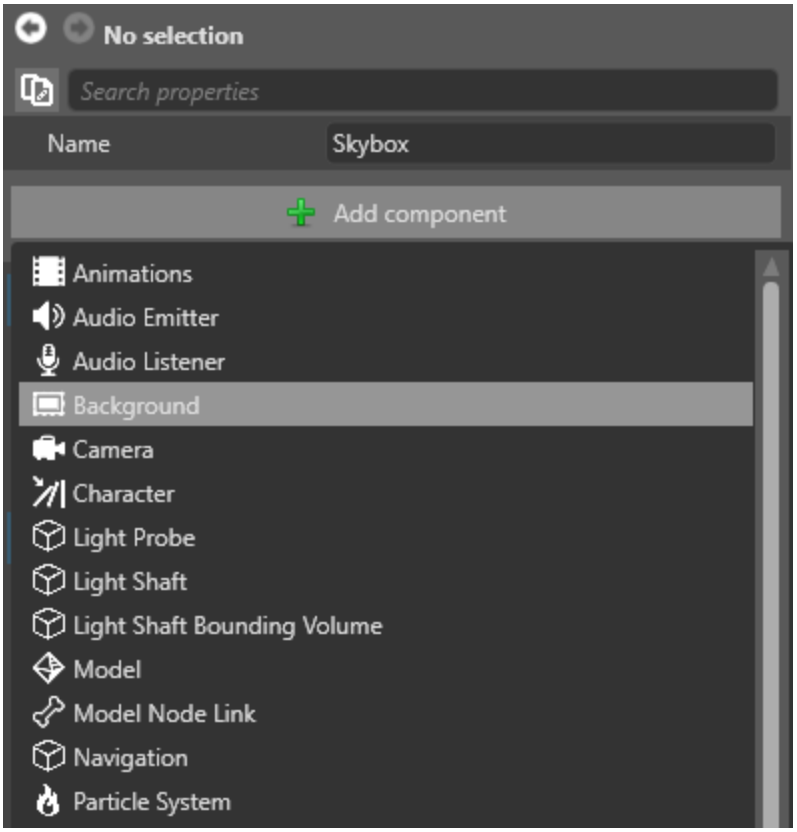
You can add background components to as many entities as you need. You might want to include more than one background, for example, if you want to switch skyboxes at runtime.

Add a background entity

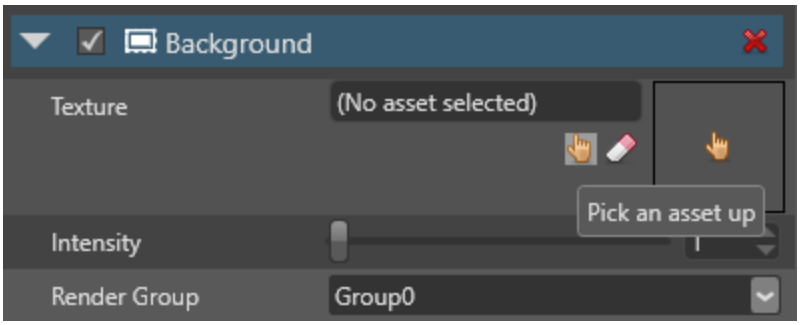
1. In the **Scene view**, select the entity you want to add the component to.

   This can be an empty entity. Its position in the scene doesn't matter.

2. In the **Property Grid** (on the right by default), click **Add component** and select **Background**.



3. Under **Texture**, select the cubemap or 360° panoramic texture you want to use in the skybox.



Use a skybox as a light source

You can use a skybox to light the scene. Stride analyzes the skybox texture and generates lighting using image-based lighting (Wikipedia). For more information, see Skybox lights.

Change the skybox at runtime

The following code changes the cubemap in a background:

```
public Texture cubemapTexture;
public void ChangeBackgroundParameters()
{
    // Get the background component from an entity
        var background = directionalLight.Get<BackgroundComponent>();

        // Replace the existing background
        background.Texture = cubemapTexture;

        // Change the background intensity
        background.Intensity = 1.5f;
}
```

Convert cubemaps to panoramas and vice versa

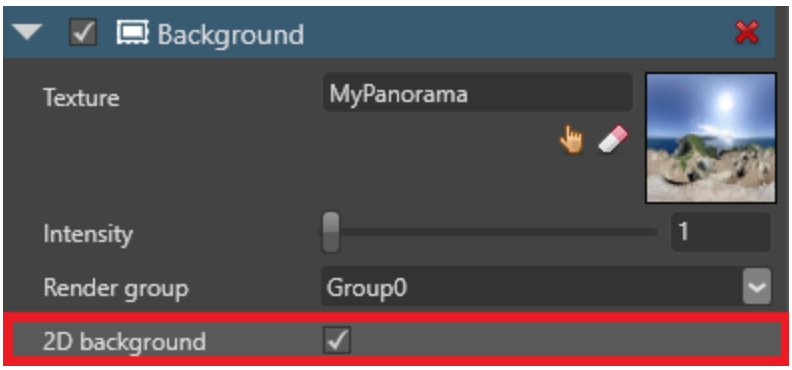
Various tools exist to convert a panoramas to cubemaps and vice versa, including:

- Panorama Converter
- Panorama to Cubemap
- Convert Cubemap to Equirectangular

Set a 2D background

Instead of using a 3D skybox, you can display the texture as a static background. This displays the texture as a flat image that stays static no matter how you move the camera. This is often useful for 2D games.

To do this, in the **Background** component properties, select **2D background**.



If you enable this with a cubemap, Stride uses the first face of the cubemap as the background.

Use a video as a skybox

For details, see Videos - Use a video as a skybox.

See also

- Skybox lights
- Lights and shadows

Lights and shadows

Beginner Designer Artist

Lights in Stride are provided by light components. There are several kinds of light.

In this section

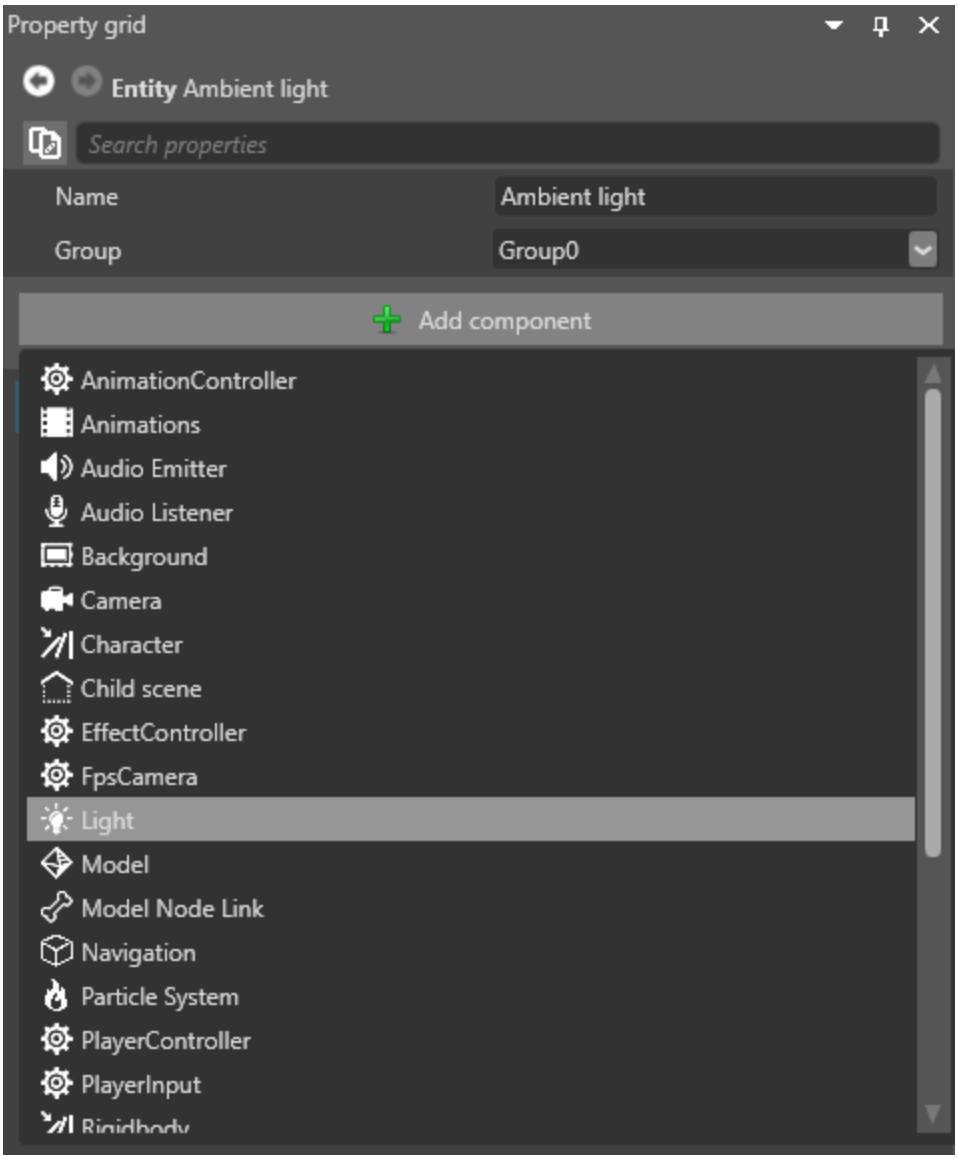
- Add a light
- Point lights
- Ambient lights
- Directional lights
- Skybox lights
- Spot lights
- Light probes
- Light shafts
- Shadows

Add a light

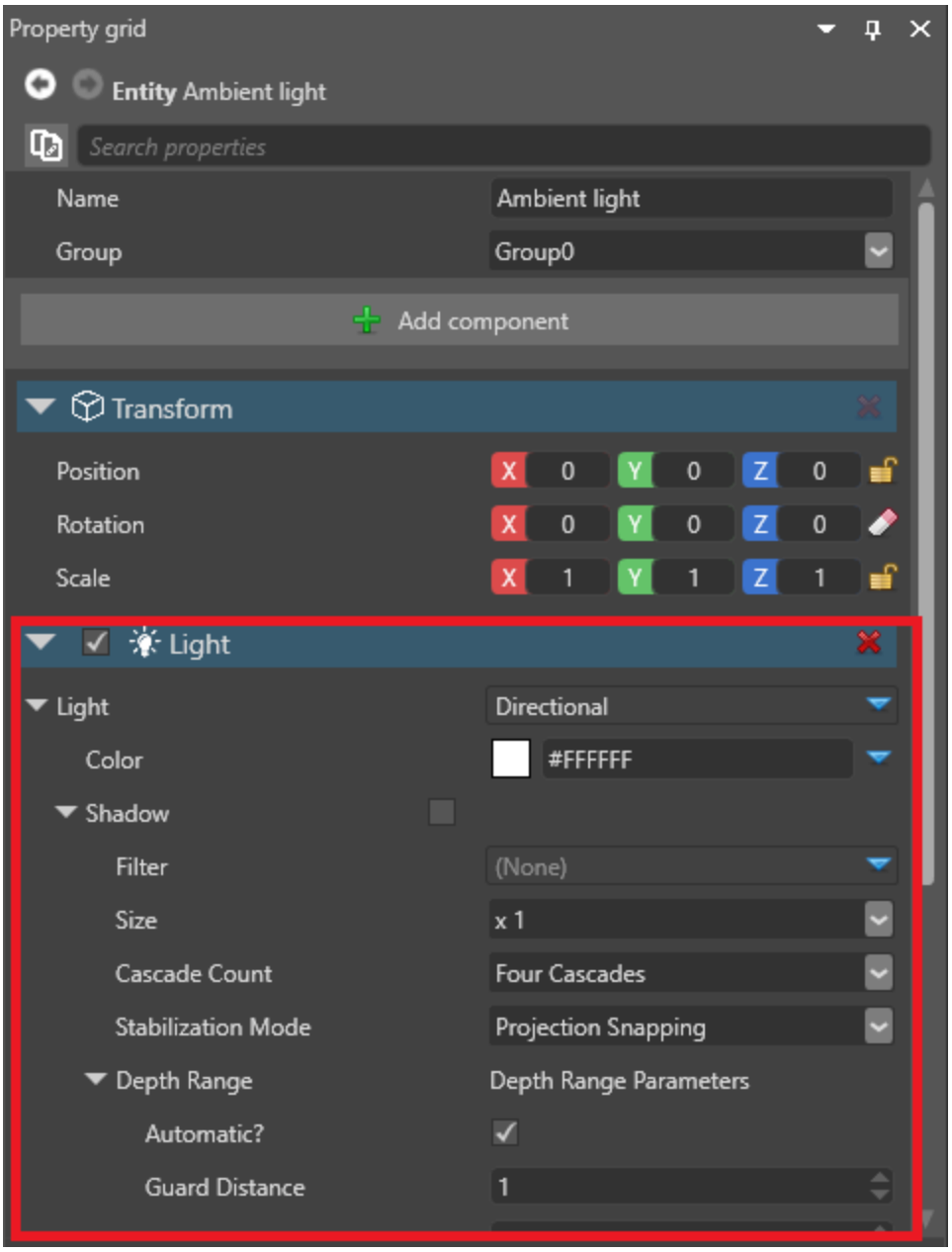
Beginner Designer Artist

To add a light to a scene, add a light component to an entity.

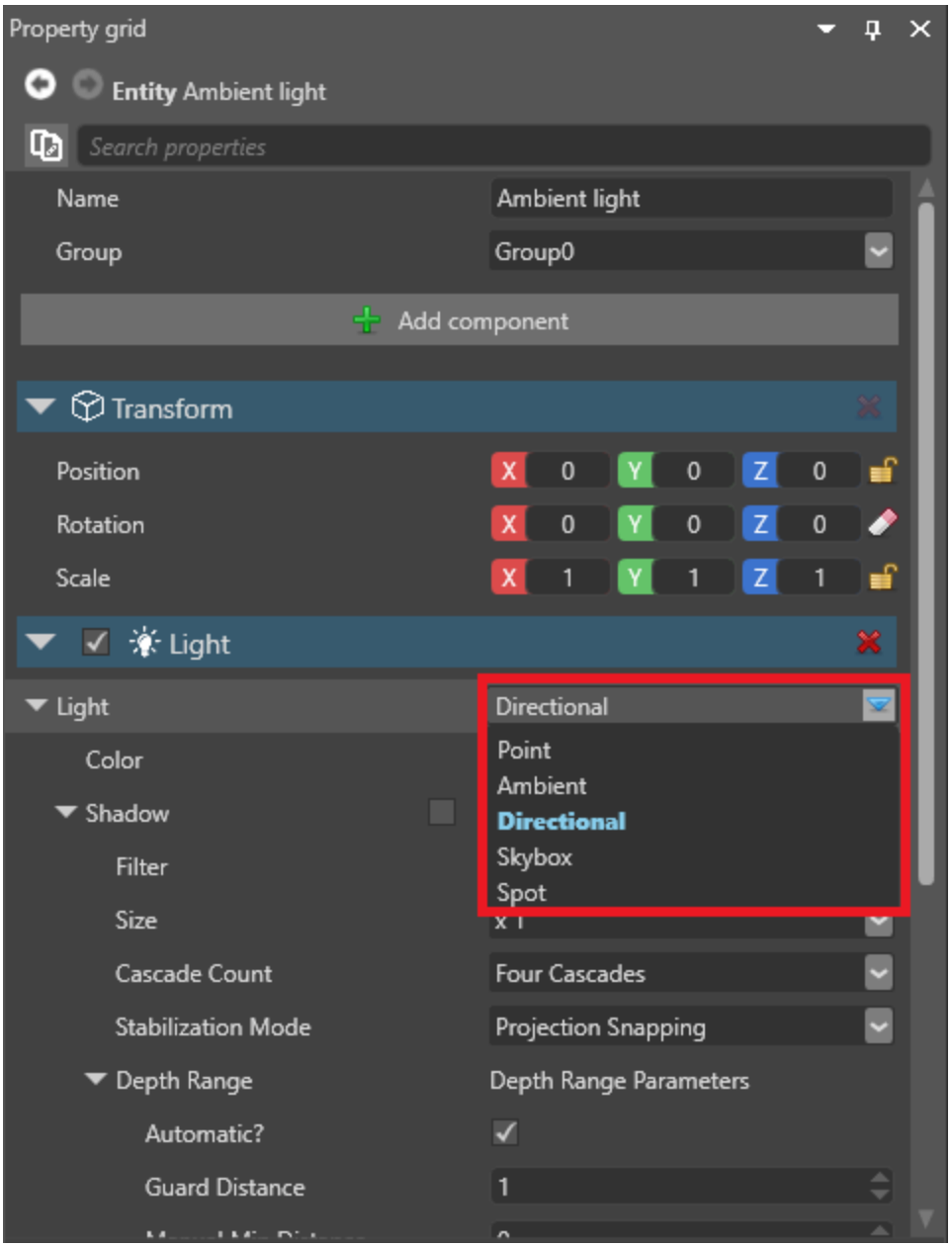
1. Select the entity you want to be a light.
2. In the Property Grid (on the right by default), click **Add component** and select **Light**.



Game Studio adds a light component to the entity.



3. Under the **Light** component properties, next to **Light**, from the drop-down menu, select the kind of light you want this entity to use.



You can choose:

- Point light
- Ambient light
- Directional light
- Skybox light
- Spot light

For information about each type of light, see its respective page.

See also

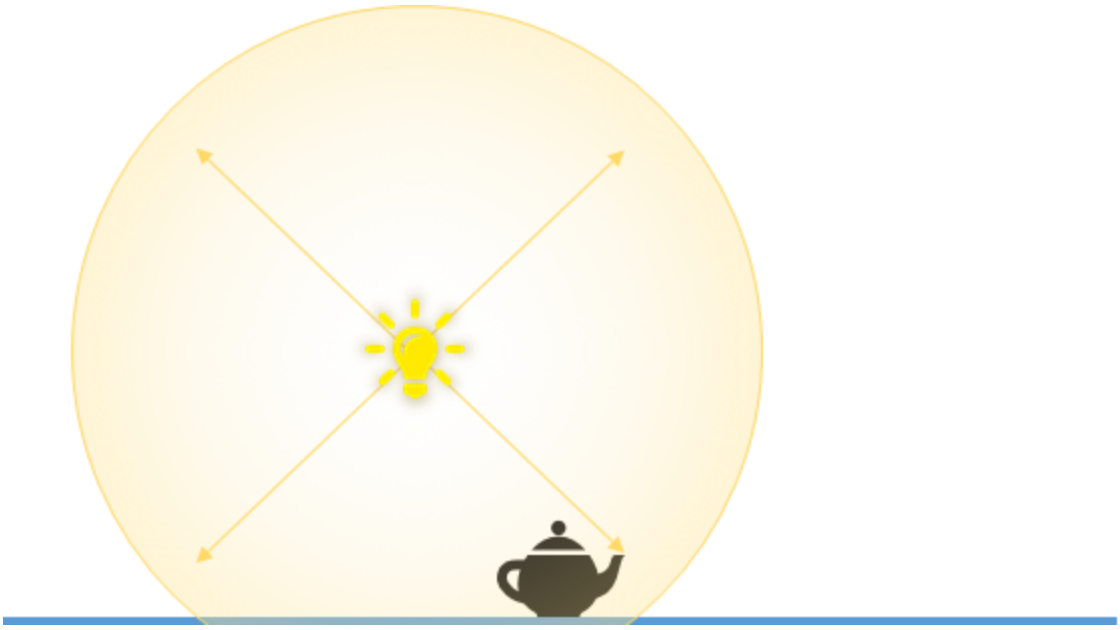
- Point lights
- Ambient lights
- Directional lights
- Skybox lights

- Spot lights
- Light probes
- Shadows

Point lights

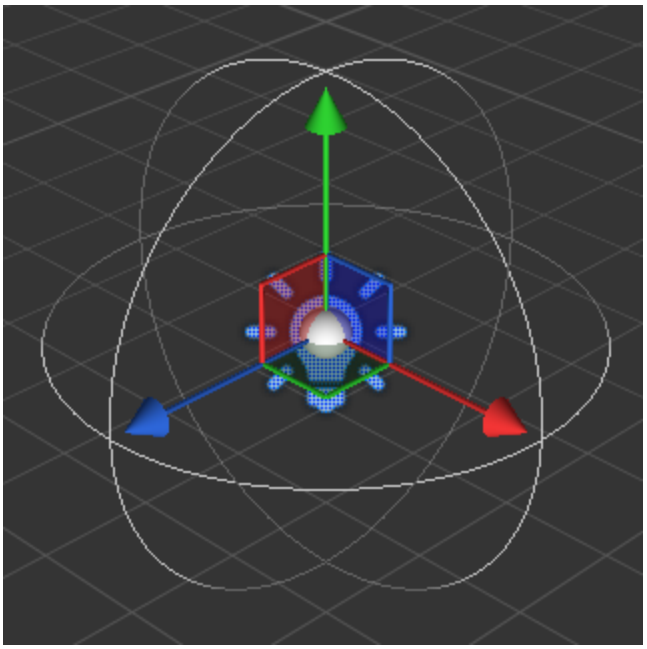
Beginner Designer Artist

Point lights emit light in all directions within a sphere. They're useful for simulating sources of local light, such as lamps and lightbulbs. They cast shadows.

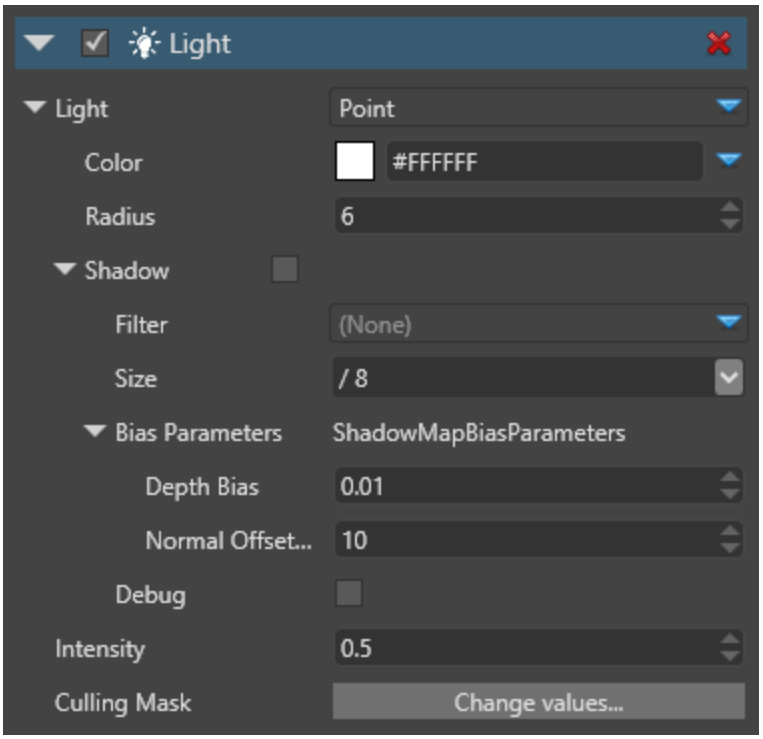


The Scene Editor shows the position of point lights with the following icon:

Once selected, the point light gizmo displays the sphere in which it projects light:



Properties



Property	Description
Color	The color of the light (RGB)
Radius	The sphere influence radius in world units. Beyond this range, the light doesn't affect models
Shadow	If shadows are enabled, the light casts shadows.

Property	Description
	Filter: Produces soft shadows instead of hard shadows via PCF (Percentage Closer Filtering) **Size:** The size of texture to use for shadowing mapping. Larger textures produce better shadows edges, but are much more costly. For more information, see Shadows
Bias Parameters	These parameters are used to avoid some artifacts of the shadow map technique. **Depth Bias:** The amount of depth to add to the sampling depth to avoid shadow acne **Normal Offset Scale**: A factor multiplied by the depth bias toward the normal
Intensity	The intensity of the light. The color is multiplied by this value before being sent to the shader. Note: negative values produce darkness and have unpredictable effects
Culling Mask	Which entity groups are affected by this light. By default, all groups are affected

See also

- Add a light
- Point lights
- Ambient lights
- Skybox lights
- Spot lights
- Light shafts
- Light probes
- Shadows

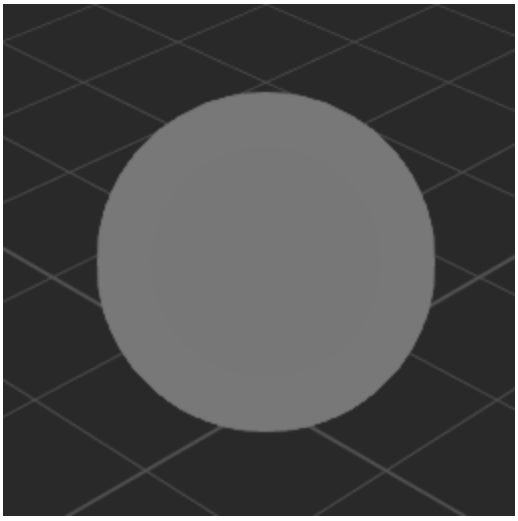
Ambient lights

Beginner Designer Artist

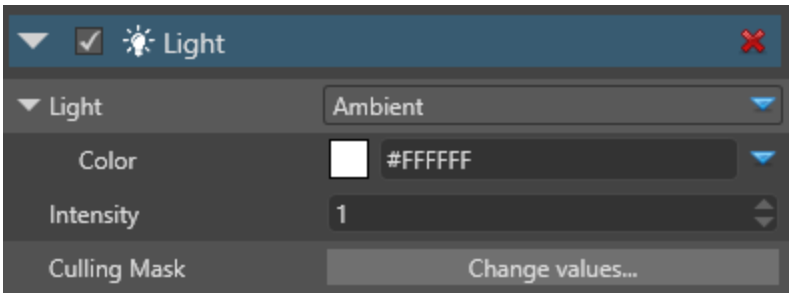
Ambient lights are uniform lights that illuminate the entire scene. Because they don't come from any specific direction or source, ambient lights illuminate everything equally, even objects in shadow or obscured by other objects. They don't cast shadows.

Ambient lights aren't realistic light sources. Instead, they contribute to the overall brightness and aesthetic of a scene.

An example of an object lit uniformly with ambient lighting (with a pure diffuse material):



Properties



Property	Description
Color	The color of the light (RGB)
Intensity	The intensity of the light. The color is multiplied by this value before being sent to the shader. Note: negative values produce darkness and have unpredictable effects
Culling Mask	Which entity groups are affected by the light. By default, all groups are affected

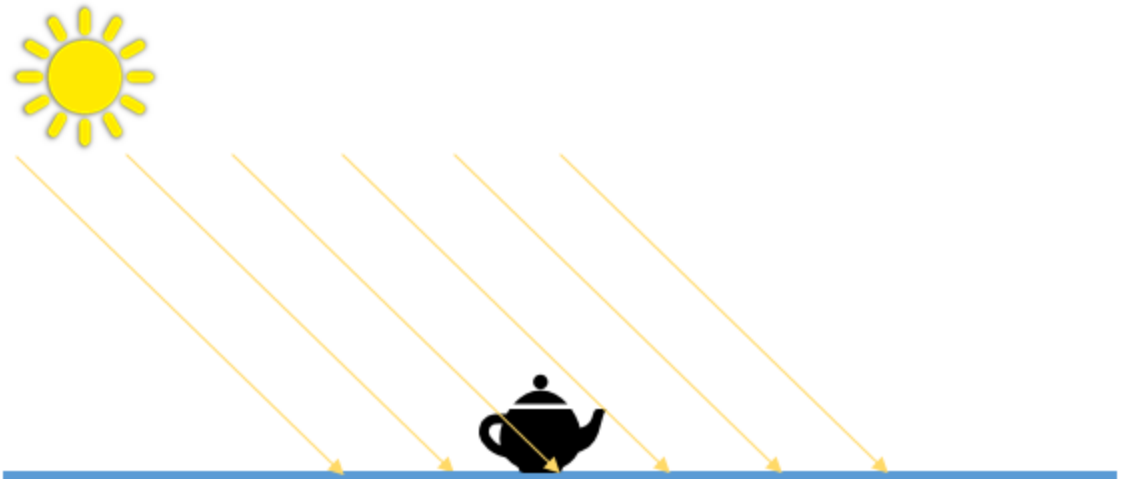
See also

- Add a light
- Point lights
- Directional lights
- Skybox lights
- Spot lights
- Light probes
- Shadows

Directional lights

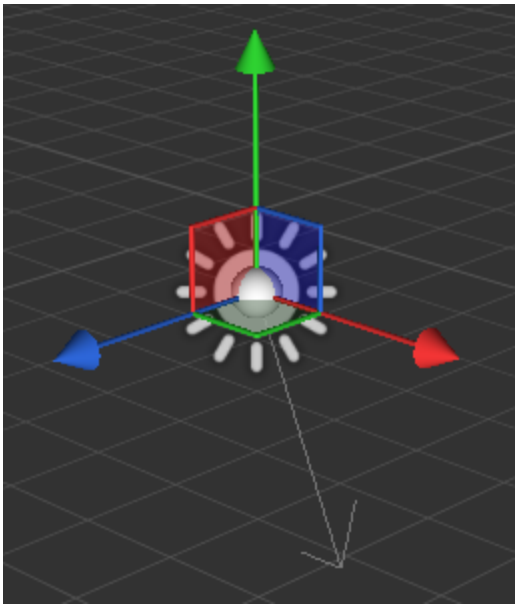
Beginner Designer Artist

Directional lights come uniformly from one direction. They're often used for simulating large, distant light sources such as the sun, and cast shadows. By default, new scenes you create in Stride contain a directional light.

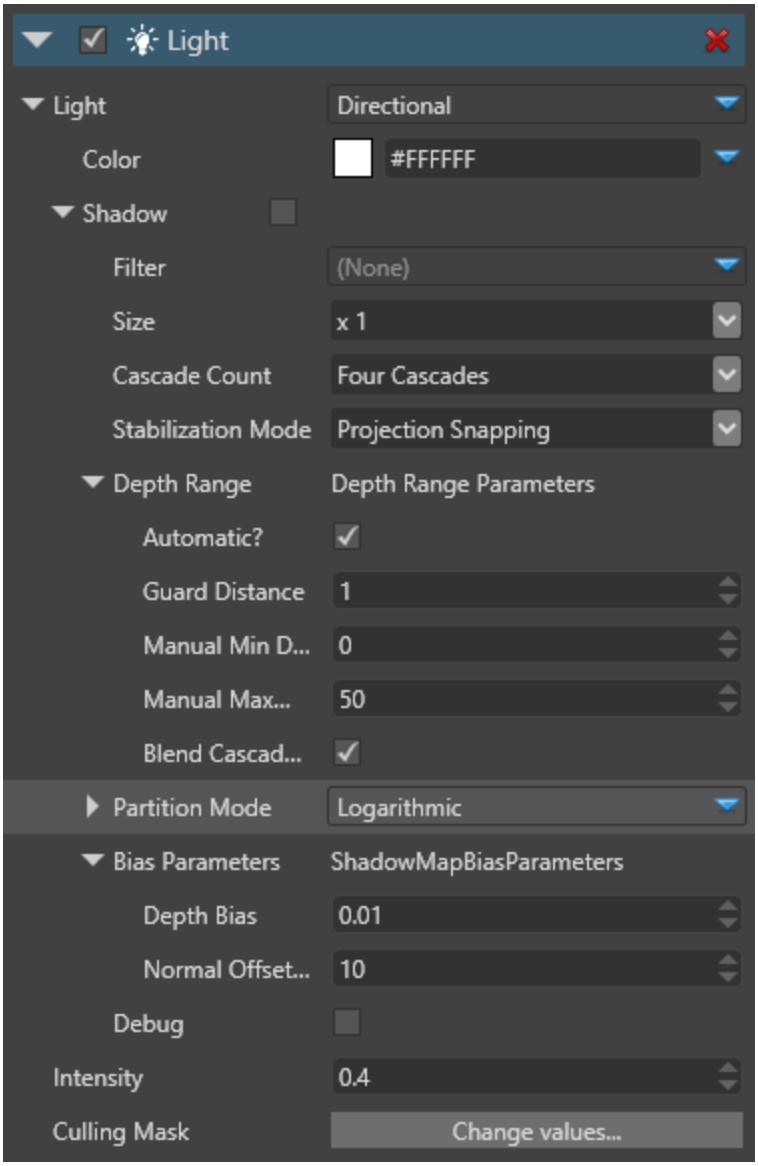


The Scene Editor shows the position of directional lights with the following icon:

When you select a directional light, the gizmo displays the light's main direction:



Properties



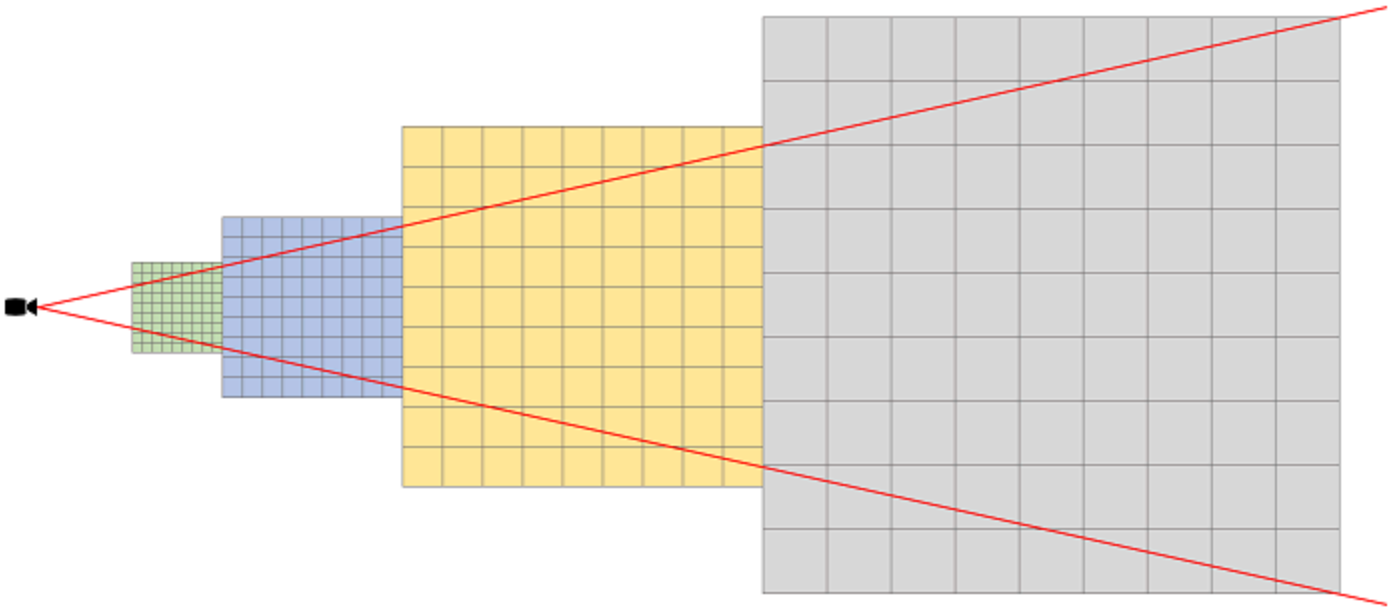
Property	Description
Color	The color of the light (RGB)
Shadow	See **Shadow properties** below
Intensity	The intensity of the light. The color is multiplied by this value before being sent to the shader. Note: negative values produce darkness and have unpredictable effects
Culling Mask	Defines which entity groups are affected by this light. By default, all groups are affected

Shadows cast by directional lights

Like point lights and spot lights, directional lights cast shadows. However, shadows cast by directional lights can spawn across a large view range, so they require special treatment to improve their realism.

Directional lights use an additional technique, **cascaded shadow mapping**. This consists of rendering the depth of occluding objects from the point of view of the light to a texture, then rendering the scene taking the occluder information into account.

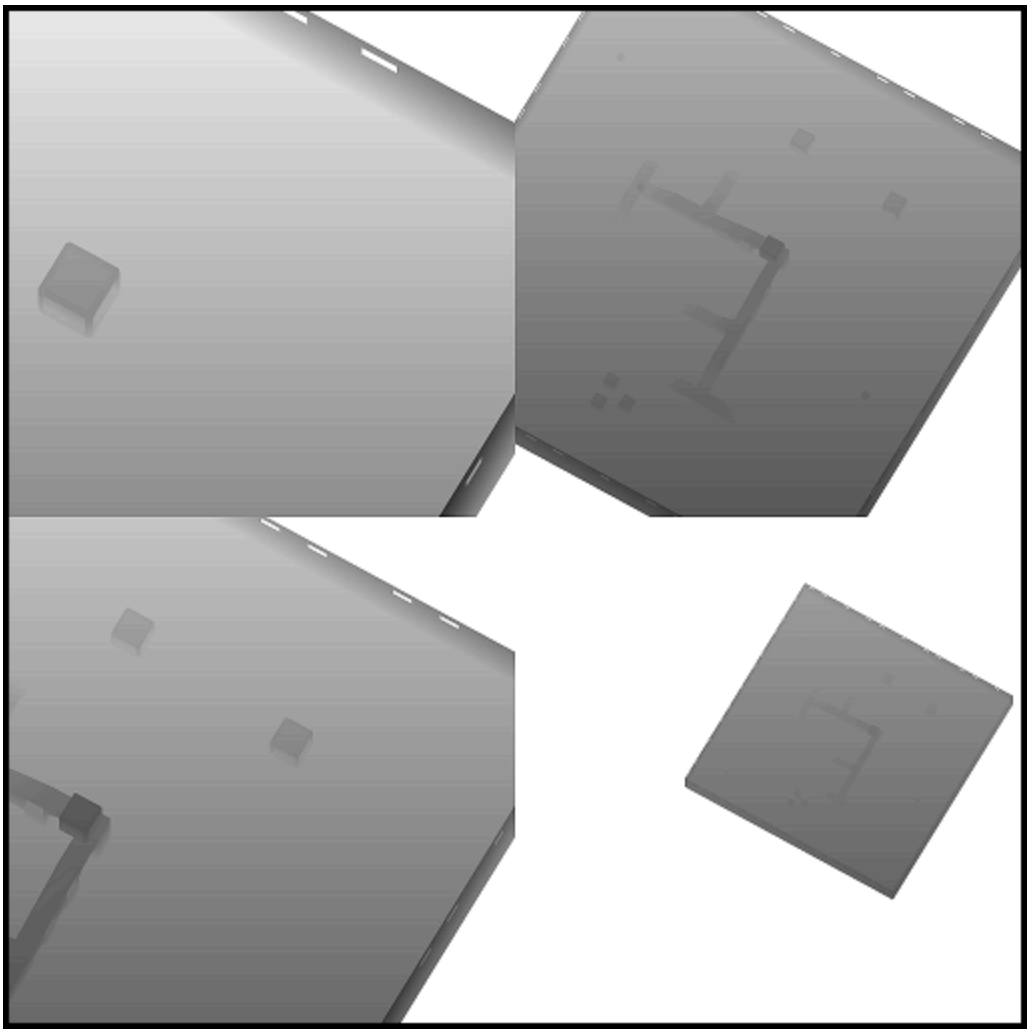
This method slices the depth range from the camera's point of view into different sections or "cascades" of different resolutions. The nearer each cascade is to the camera, the higher resolution it has, and the higher-resolution its shadows are.



Put simply, the closer shadows are to the camera, the better quality they are. This means you can spend more memory on shadows closer to the camera, where you can see them, and less on distant shadows.

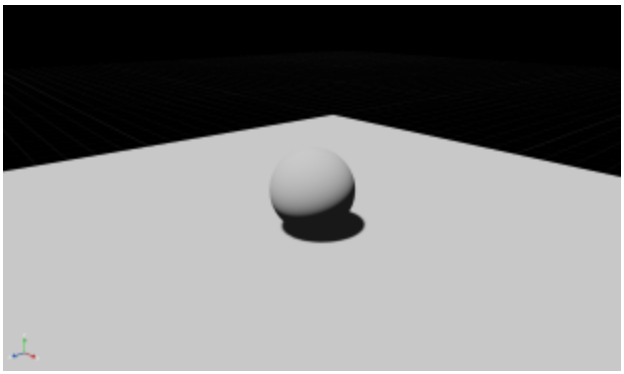
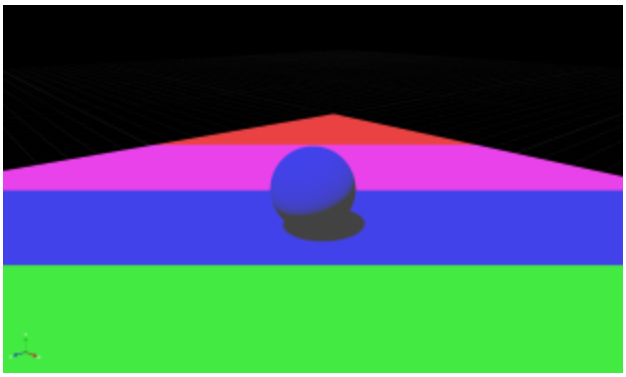
You can have one, two, or four cascades. The more cascades you use, the more memory you save, but the lower resolution your shadows become over distance.

This an example of a shadow map generated from a directional light, using four cascades:



See shadow cascades in the editor

In the **Property Grid**, under the **Shadow** properties, enable the **Debug** option.

Cascades debug off	Cascades debug on
	

The different colors indicate the cascade for each distance range (Green: 0, Blue: 1, Purple: 2, Red: 3).

Directional light shadow properties

Property	Description
Filter	Filtering produces **soft shadows** instead of **hard shadows**. Currently, the implemented technique is PCF (Percentage Closer Filtering)
Size	The size of the shadow map texture. For the directional light, this value is **x1** by default, as a directional light has more visual impact than lights with shorter ranges
Cascade Count	The number of cascades used for slicing the range of depth covered by the light. Values are 1, 2 or 4 cascades; a typical scene uses 4 cascades
Stabilization mode	The technique used to reduce shadow map flickering. Flickering is a result of the potential aliasing introduced by the shadow map when a texel from the perspective of the light covers more space than a texel from the camera's perspective. **Projection snapping** tries to snap the projection matrix of the light to a texel dependent on the resolution of the shadow map texture **View snapping** tries to snap the target of the view matrix of the light (center of the camera view cascade frustum) Both projection and view snapping force the shadow matrix to cover a larger region, increasing the aliasing of the shadow map texture. Note that when using depth range camera is set to automatic, the stabilization mode is ignored
Depth Range	How the visible depth range from the camera's perspective is calculated. This directly affects how near and how far cascades splits occur
Blend Cascades	Smooths the transition between cascades
Partition mode	How the cascade split distance is determined.

Property	Description
	Manual: the split is defined manually for each cascade, in percentage of the visible depth range. A value of 0.1 for a cascade means that the cascade is rendered on the distance 0.1 * (VisibleDepthMax - VisibleDepthMin) **Logarithmic**: the split is automatically calculated using a logarithmic scale The PSSM factor lets you blend from a pure logarithmic scale (0.0f) to a pure uniform scale (1.0f)
Depth Bias	The amount of depth to add to the sampling depth to avoid the phenomenon of shadow acne
Normal Offset Scale	A factor multiplied by the depth bias toward the normal
Debug	Displays the shadow map cascades in the Scene Editor

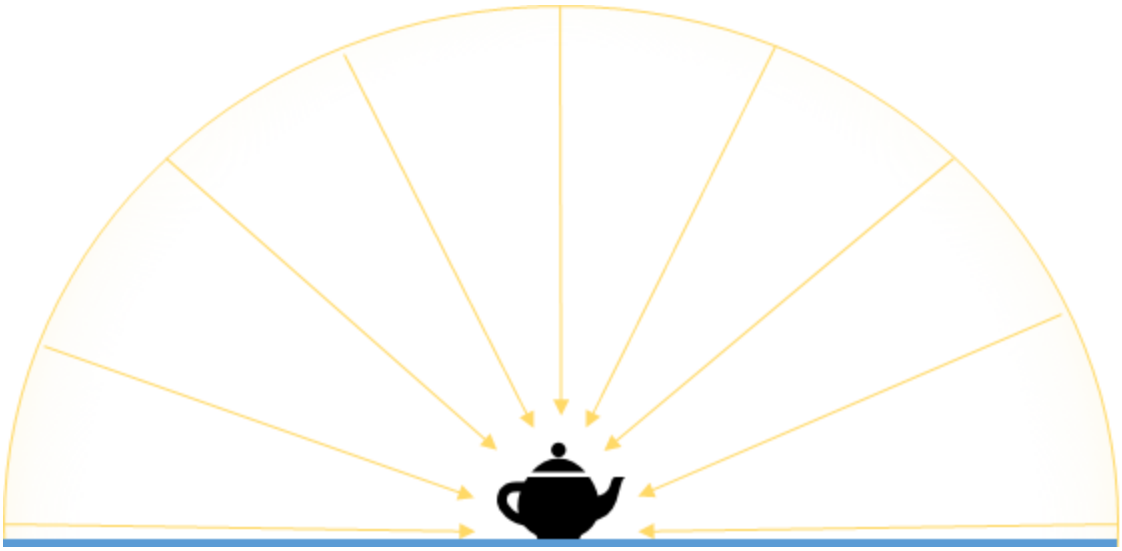
See also

- Add a light
- Point lights
- Ambient lights
- Skybox lights
- Spot lights
- Light probes
- Light shafts
- Shadows

Skybox lights

Beginner Designer Programmer

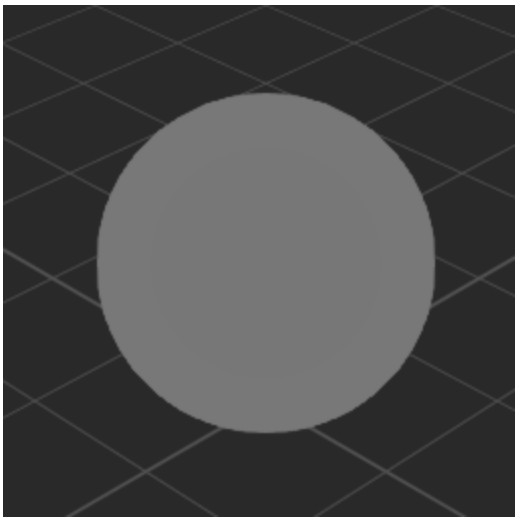
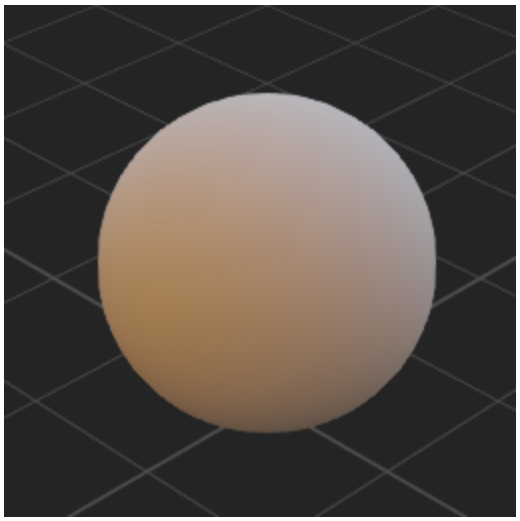
A **skybox light** is an ambient light emitted by a skybox. Stride analyzes the skybox texture and generates lighting using image-based lighting (Wikipedia).



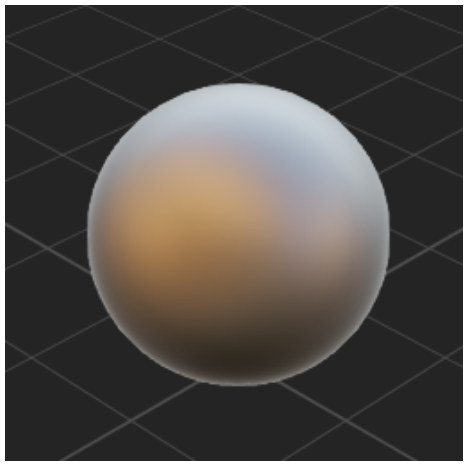
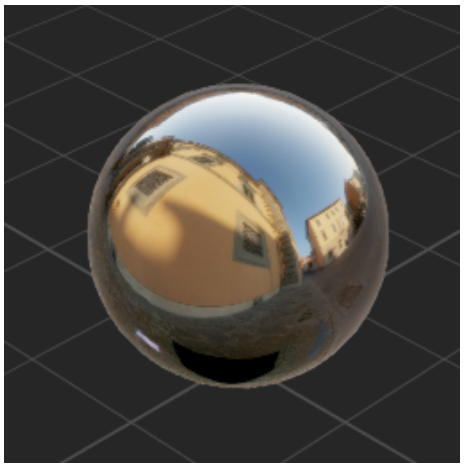
Skybox lights are good for exterior scenes, where the skybox is visible. They're less useful for interior scenes, such as in rooms where the skybox is only visible through windows; as the skybox light nonetheless lights the entire room, this creates an unnatural effect.

How skyboxes light the scene

These images show the difference between ambient and skybox lighting on two pure diffuse materials:

Ambient lighting	Skybox lighting
	

These images show the effect of skybox lighting on a material with different metal and gloss properties:

Material Plastic	Metal 100% Gloss 50%	Metal 100% Gloss 100%
		

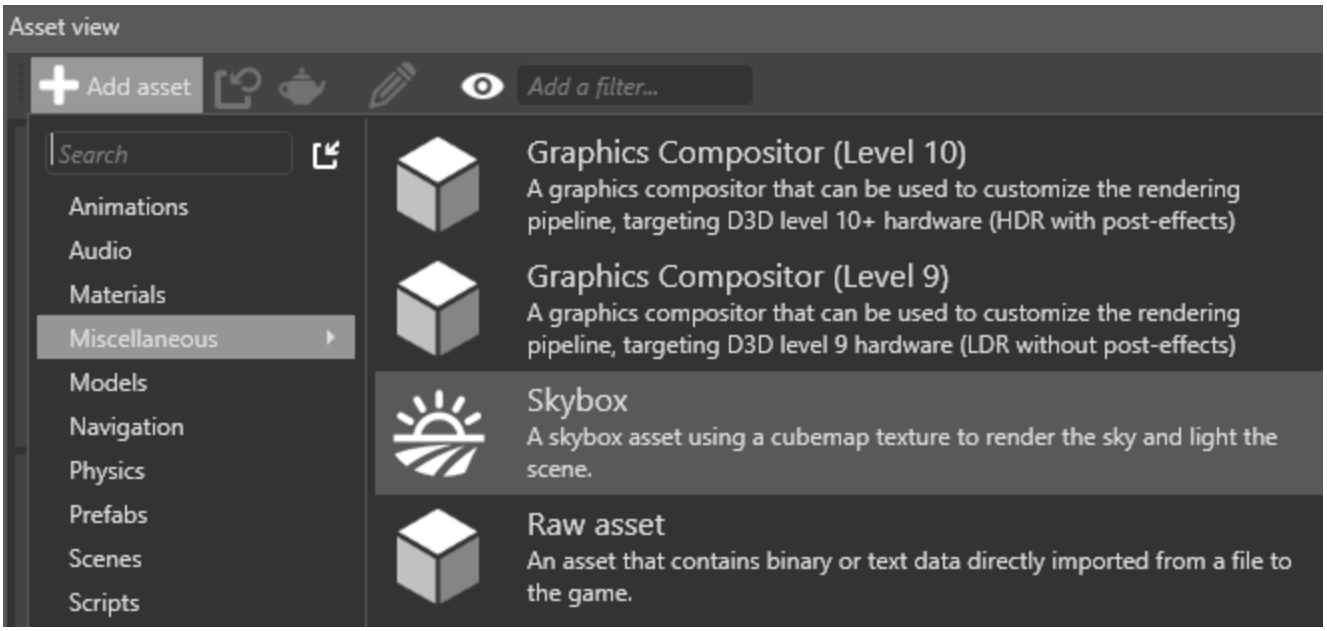
Notice how the skybox texture colors are reflected.

Set up a skybox light

To use a skybox as a light, you need to add a skybox asset, then select it in a Light component.

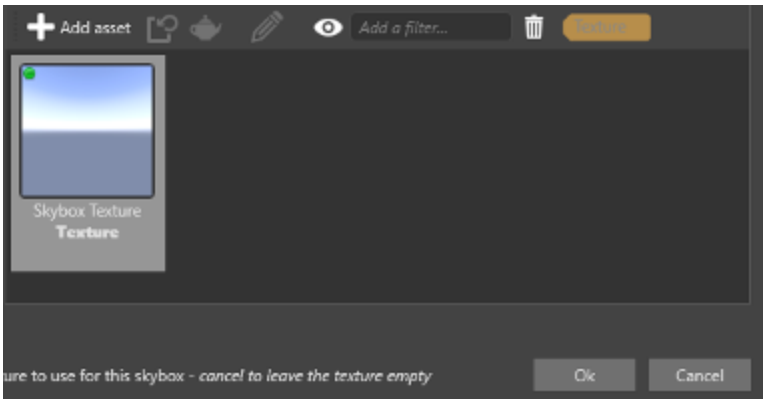
1. In the **Asset View**, click Add asset

2. Select **Miscellaneous** > **Skybox**.



   The **Select an asset** window opens.

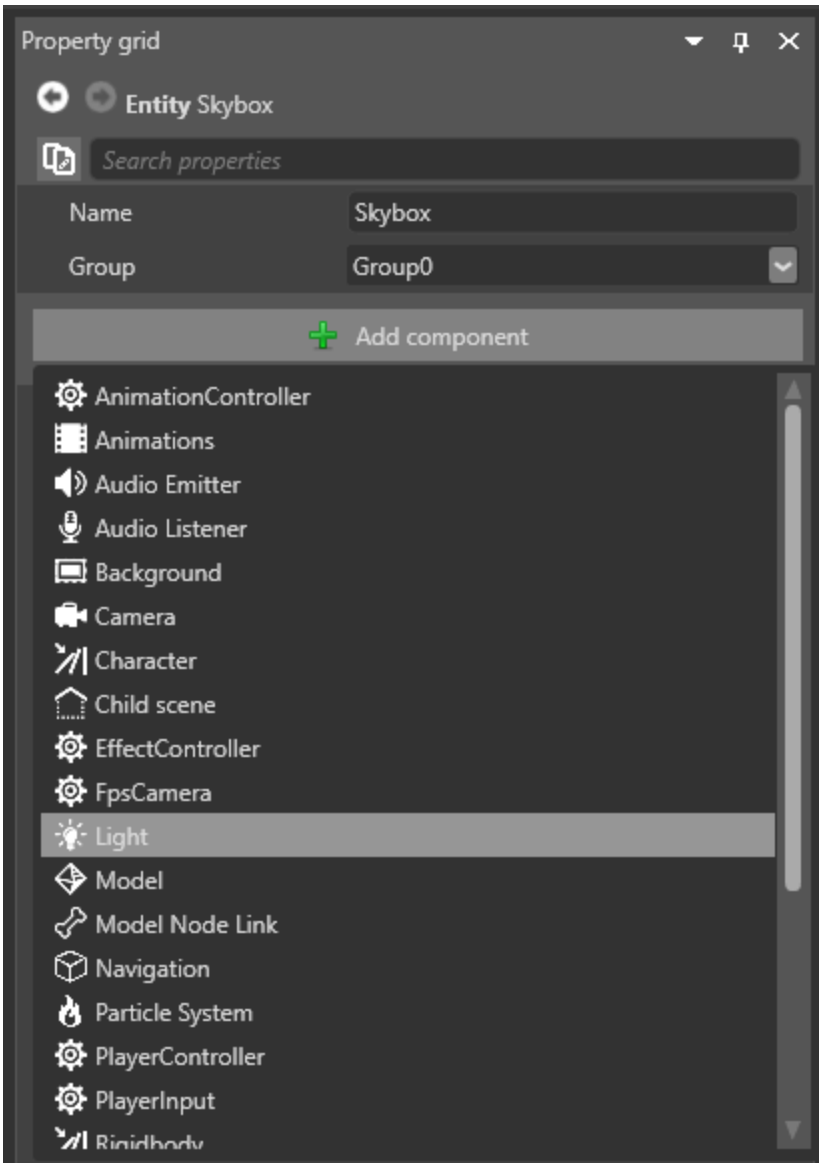
3. Choose a skybox texture from the project assets and click **OK**.



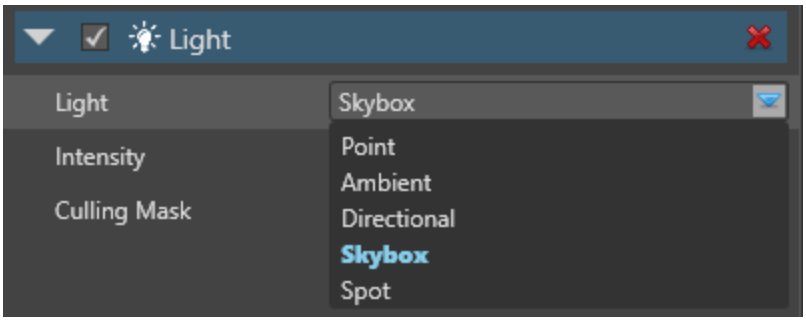
Game Studio adds the skybox asset with the texture you specified.

4. Select the entity you want to be the skybox light.

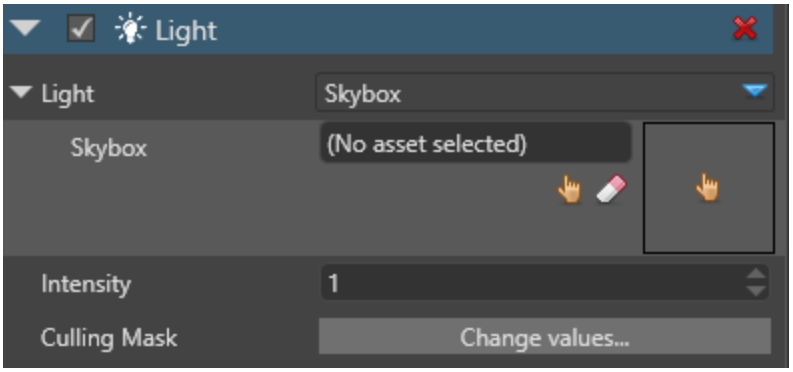
5. In the **Property Grid** (on the right by default), click **Add component** and select Light.



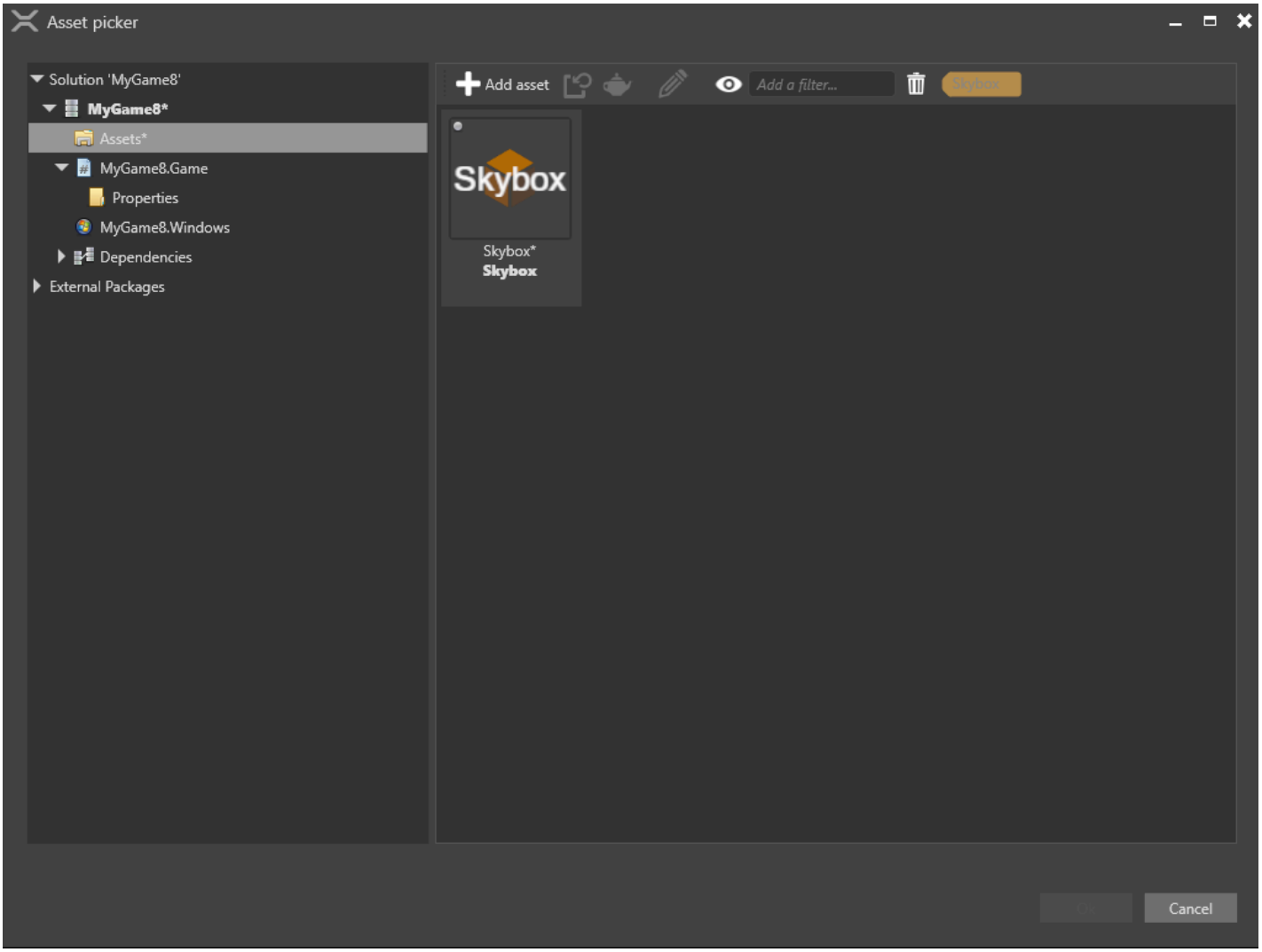
6. In the **Light** component properties, under **Light**, select **Skybox**.



7. Click (**Select an asset**):



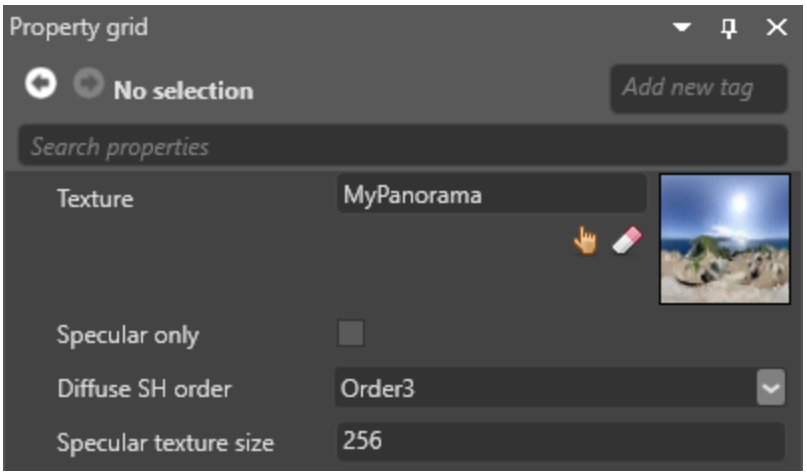
8. Select the skybox asset you want to use as a light source and click **OK**.



The Light component uses the skybox asset to light the scene.

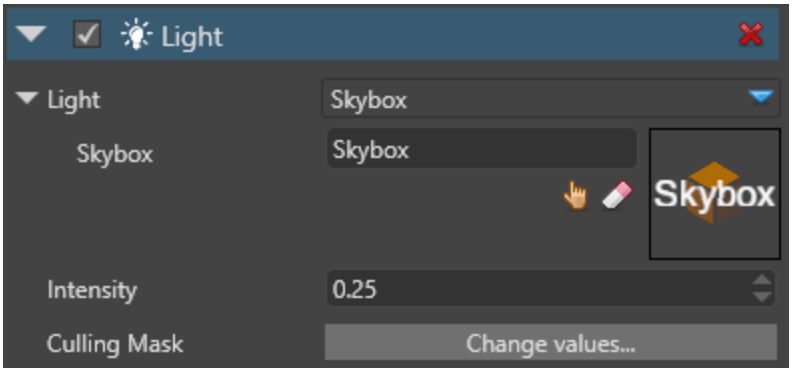
Skybox asset properties

When you use a skybox as a light, Stride uses it both in compressed form (spherical harmonics (Wikipedia)) and as a texture to light different kinds of material. You can control the detail of both in the skybox asset properties.



Property	Description
Texture	The texture to use as skybox (eg a cubemap or panoramic texture)
Specular Only	Use the skybox only for specular lighting
Diffuse SH order	The level of detail of the compressed skybox, used for diffuse lighting (dull materials). `Order5` is more detailed than `Order3`.
Specular Cubemap Size	The texture size used for specular lighting. Larger textures have more detail.

Skybox light properties



Property	Description
Intensity	The light intensity
Culling Mask	Which entity groups are affected by the light. By default, all groups are affected

Example code

The following code changes the skybox light and its intensity:

```
public Skybox skybox;
public void ChangeSkyboxParameters()
{
    // Get the light component from an entity
        var light = Entity.Get<LightComponent>();

        // Get the Skybox Light settings from the light component
        var skyboxLight = light.Type as LightSkybox;

        // Replace the existing skybox
        skyboxLight.Skybox = skybox;

        // Change the skybox light intensity
        light.Intensity = 1.5f;
}
```

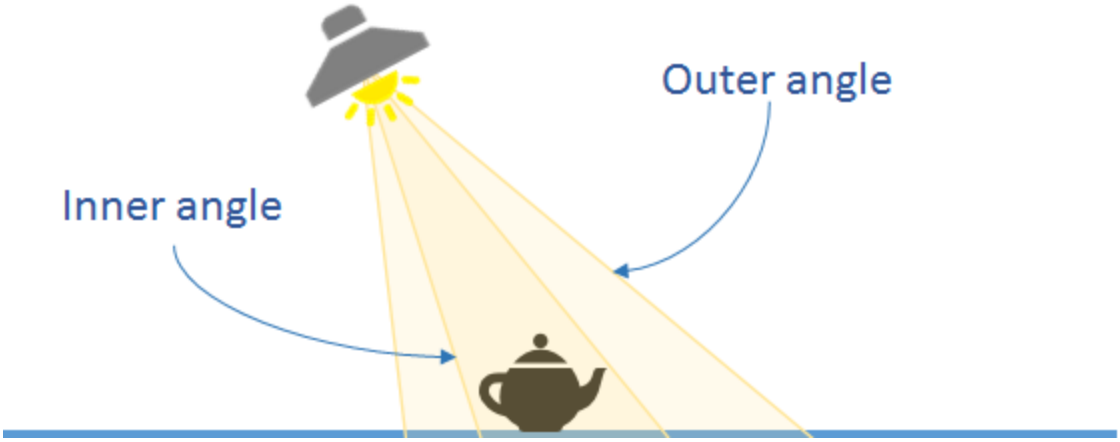
See also

- Skyboxes and backgrounds

Spot lights

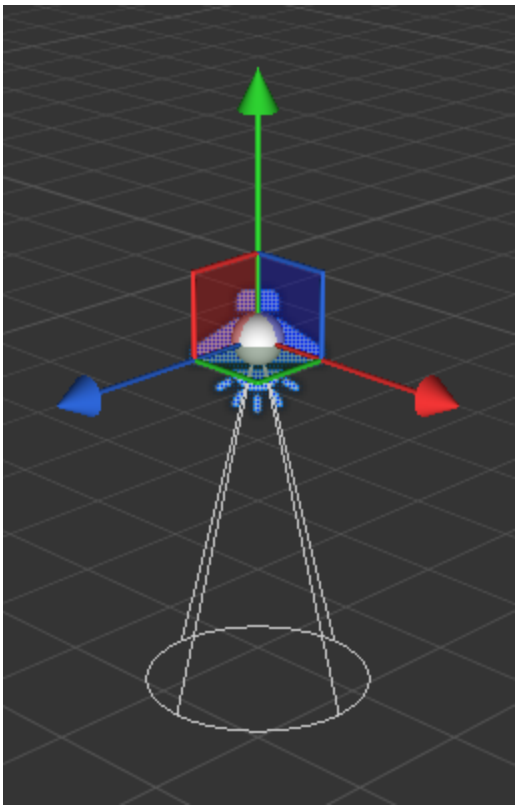
Beginner Designer Artist

Spot lights produce a cone of light in a specific direction. They're useful for simulating light from objects such as lampposts and flashlights. They cast shadows. You can control them with scripts or animation to create dramatic lighting effects.

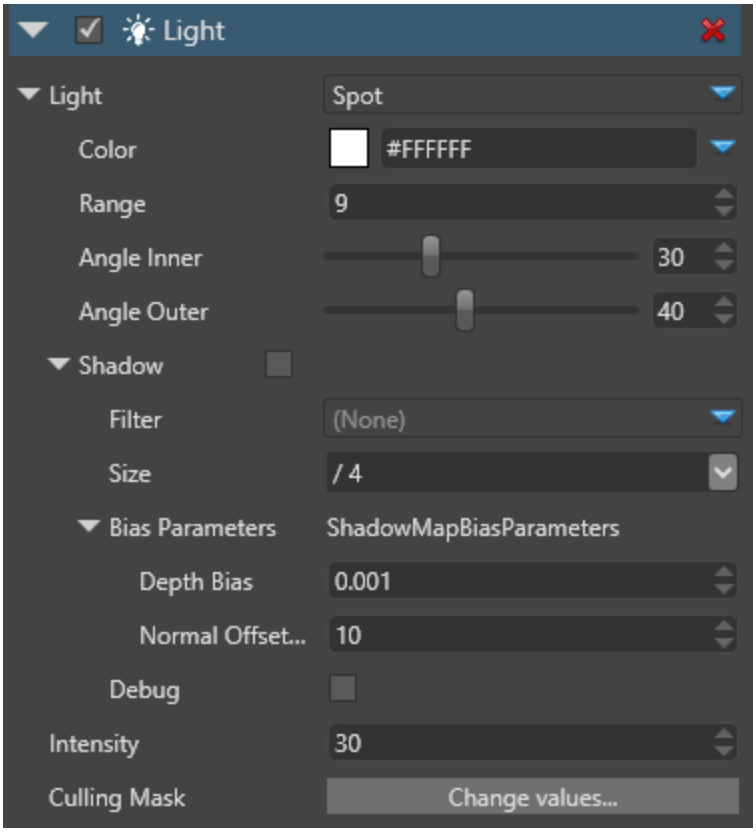


The Scene Editor shows the position of the spot light with the following icon:

Once selected, the gizmo of the spot light displays its main direction, range and the outer cone:



Properties



Property	Description
Color	The color of the light (RGB)

Property	Description
Range	The range in world units. Beyond the this range, the light doesn't affect models.
Angle Inner	The inner angle of the spot cone where the light intensity influence is at one
Angle Outer	The outer angle of the spot cone where the light intensity influence is zero
Shadows	Cast shadows **Filter**: Produces soft shadows instead of hard shadows via PCF (Percentage Closer Filtering) **Size**: The size of texture to use for shadowing mapping. Larger textures produce better shadows edges, but are much more costly. For more information, see Shadows For spot lights, the default value is **medium**, as a spot light has usually a medium visual impact
Bias Parameters	These parameters are used to avoid some artifacts of the shadow map technique. **Depth Bias:** The amount of depth to add to the sampling depth to avoid shadow acne **Normal Offset Scale**: A factor multiplied by the depth bias toward the normal
Intensity	The intensity of the light. The color is multiplied by this value before being sent to the shader. Note: negative values produce darkness and have unpredictable effects

Property	Description
Culling Mask	Defines which entity groups are affected by this light. By default, all groups are affected

See also

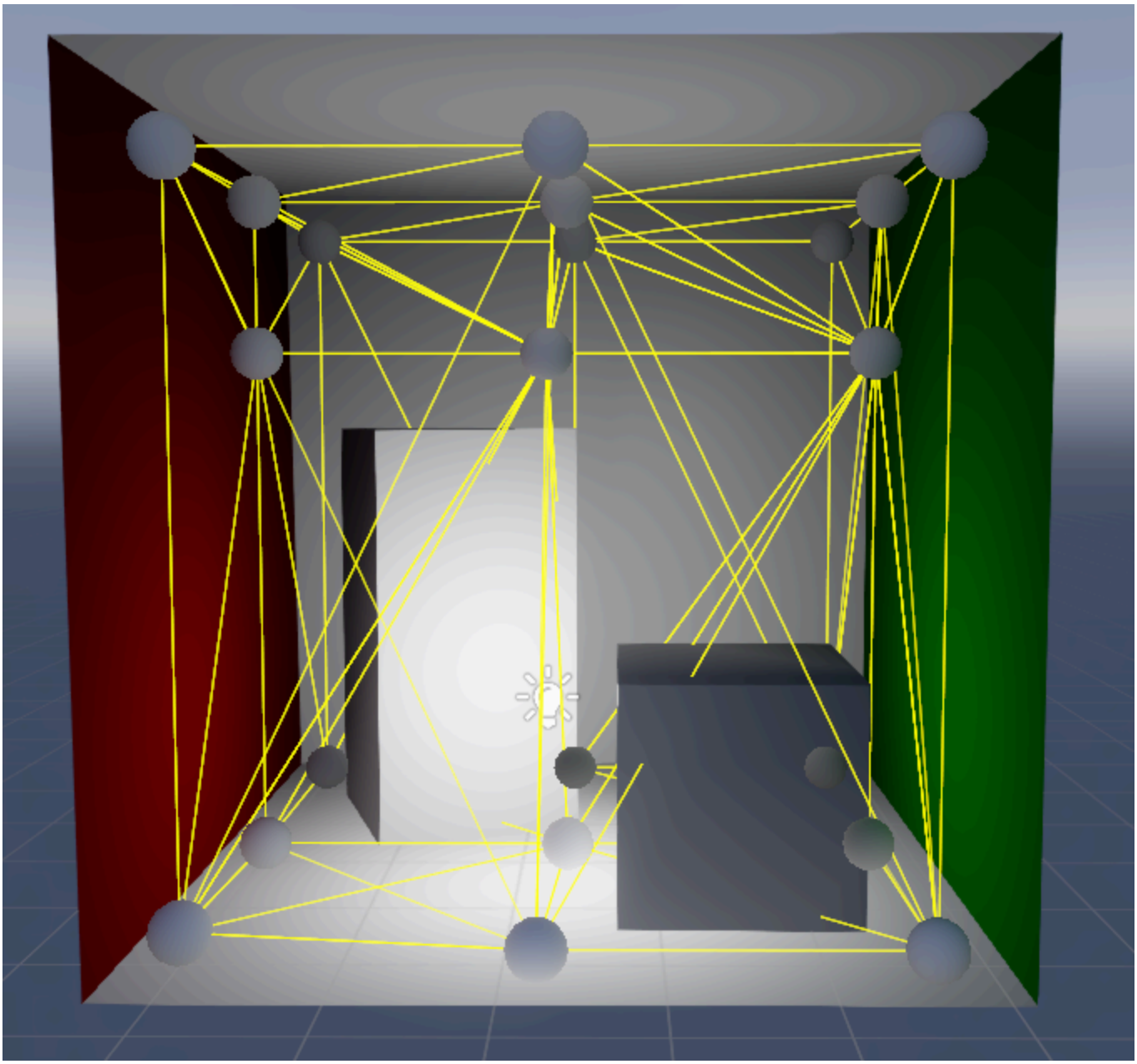
- Add a light
- Point lights
- Ambient lights
- Directional lights
- Light shafts
- Skybox lights
- Light probes
- Shadows

Light probes

Beginner Designer Artist

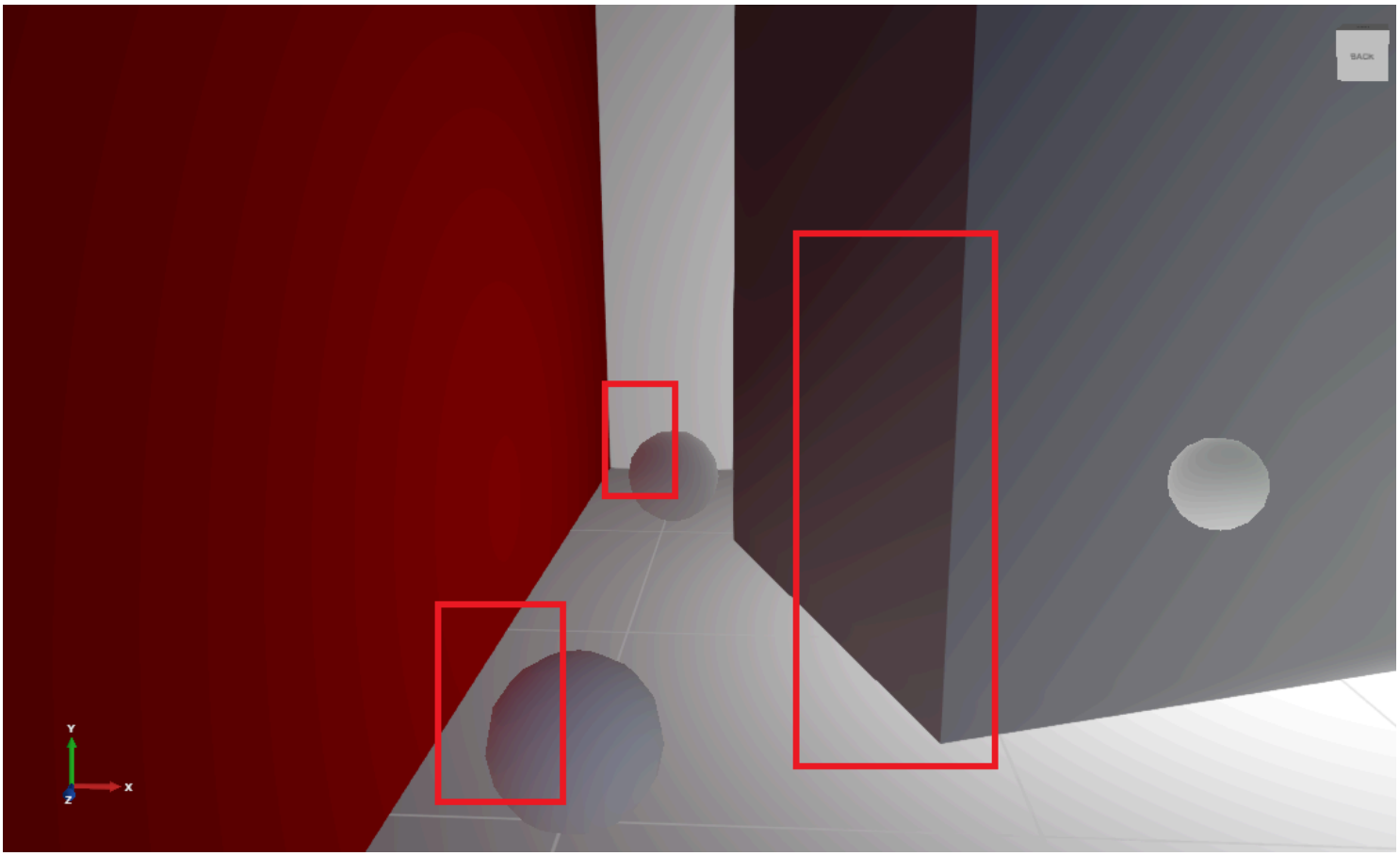
Light probes capture the lighting at the position you place them. They simulate **indirect light**, the effect of light bouncing off surfaces and illuminating other surfaces. They can make a dramatic difference to the mood and appearance of your scene.

The screenshot below shows a point light surrounded by light probes in the Scene Editor. The probes form 3D areas (shown in the Scene Editor by the yellow wireframe connecting the probes).



Stride colors pixels within a light probe area to simulate the effect of light bouncing from nearby surfaces. To find a color for a given pixel, Stride interpolates from the lighting captured by the four surrounding light probes.

For example, in the screenshot below, notice how the red of the wall is reflected on the other objects. In the Scene Editor, this is also visible on the surface of the light probes themselves.

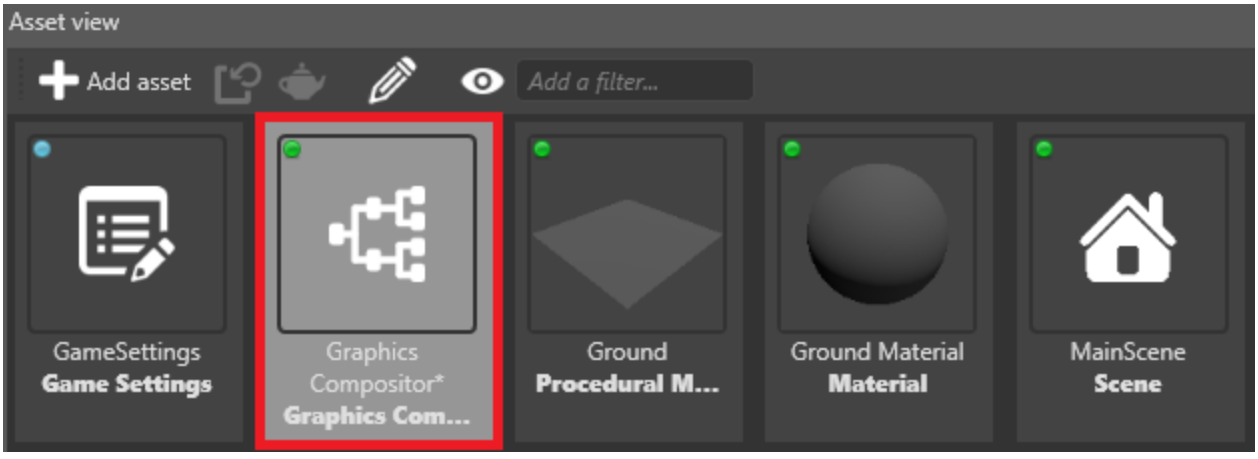


Light probes affect all objects in the area they cover, including static and dynamic objects. You don't need to enable any extra options on the entities that light probes affect.

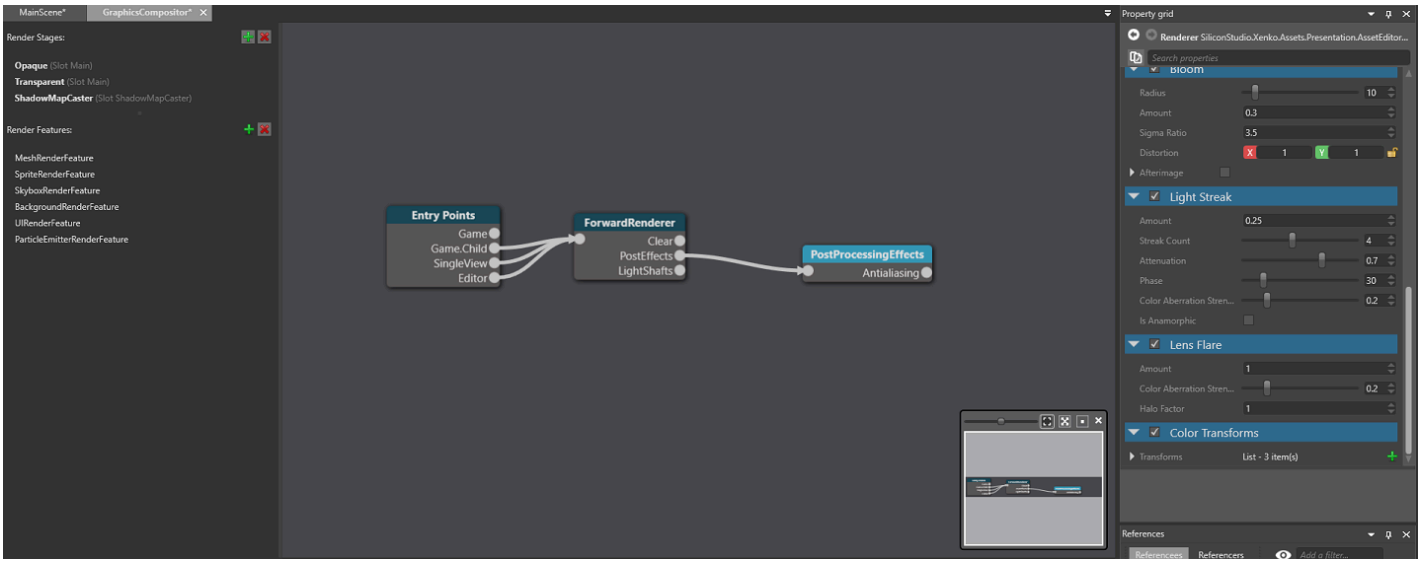
1. Enable light probes in the graphics compositor

Stride enables light probes by default in new projects. To make sure light probes are enabled:

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

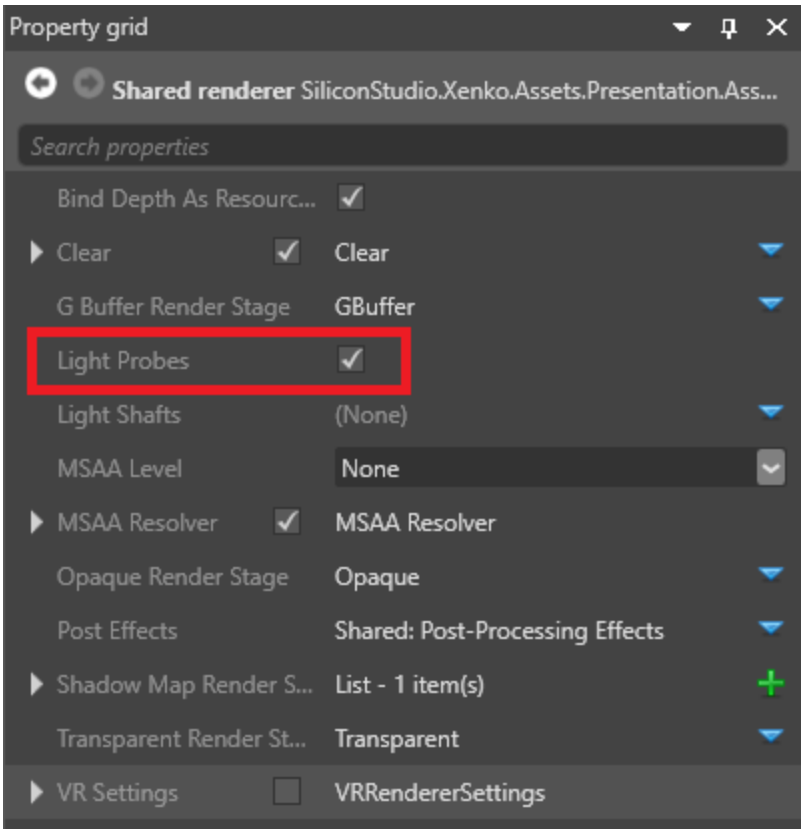


The graphics compositor editor opens.



2. Select the **forward renderer** node.

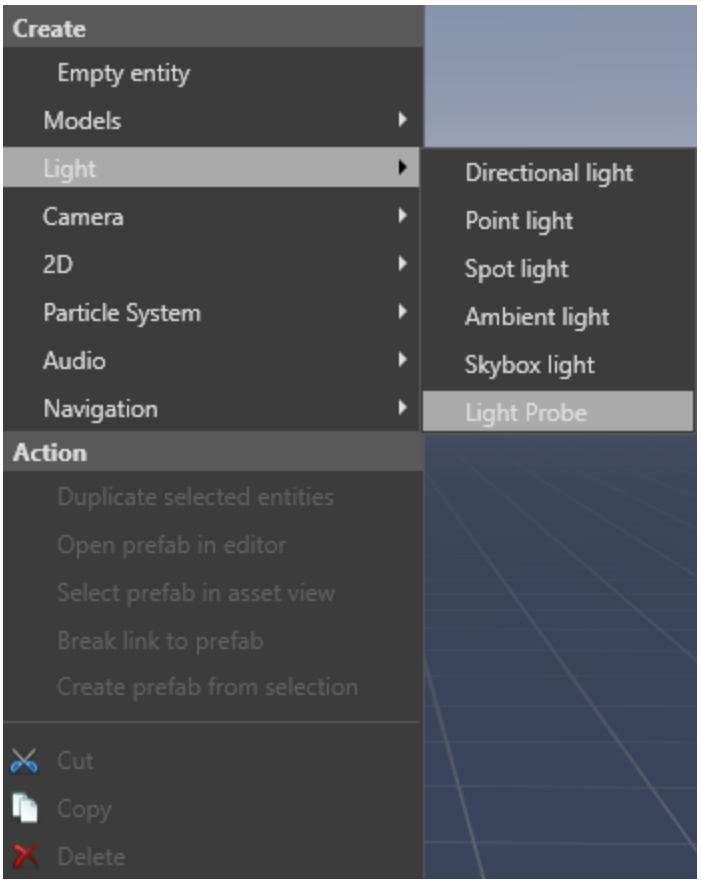
3. In the **Property Grid** (on the right by default), make sure the **Light probes** checkbox is selected.



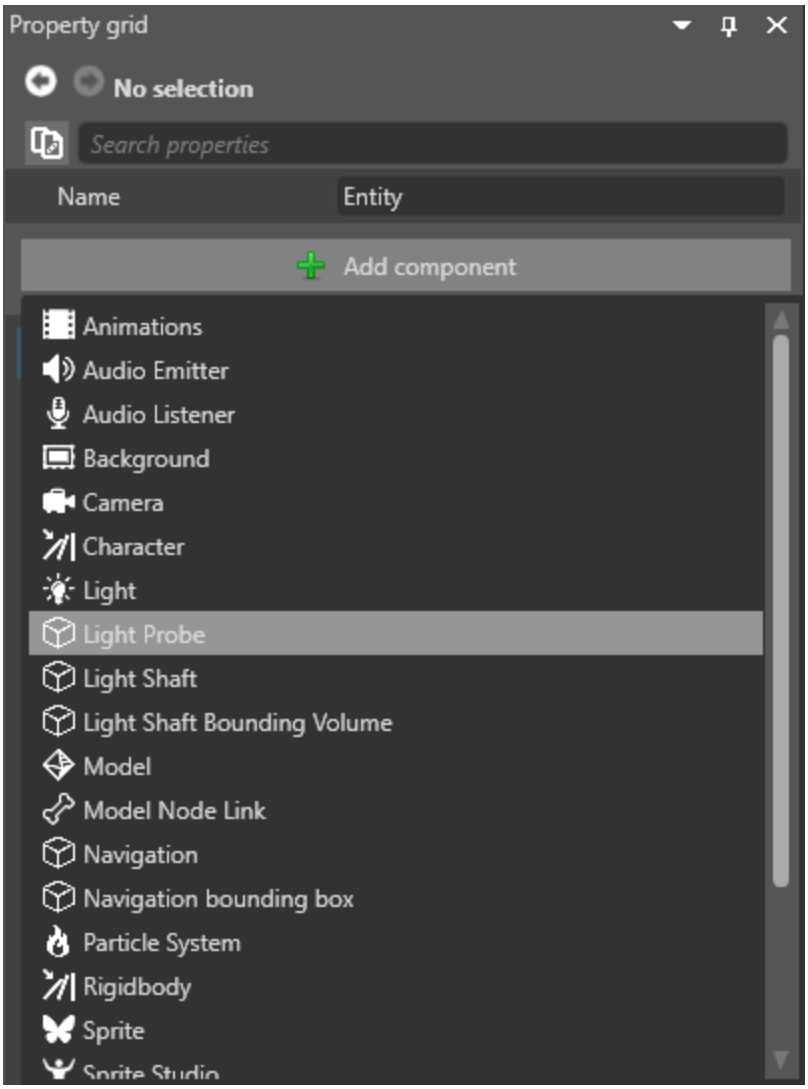
For more information about the graphics compositor, see the Graphics compositor page.

2. Create a light probe

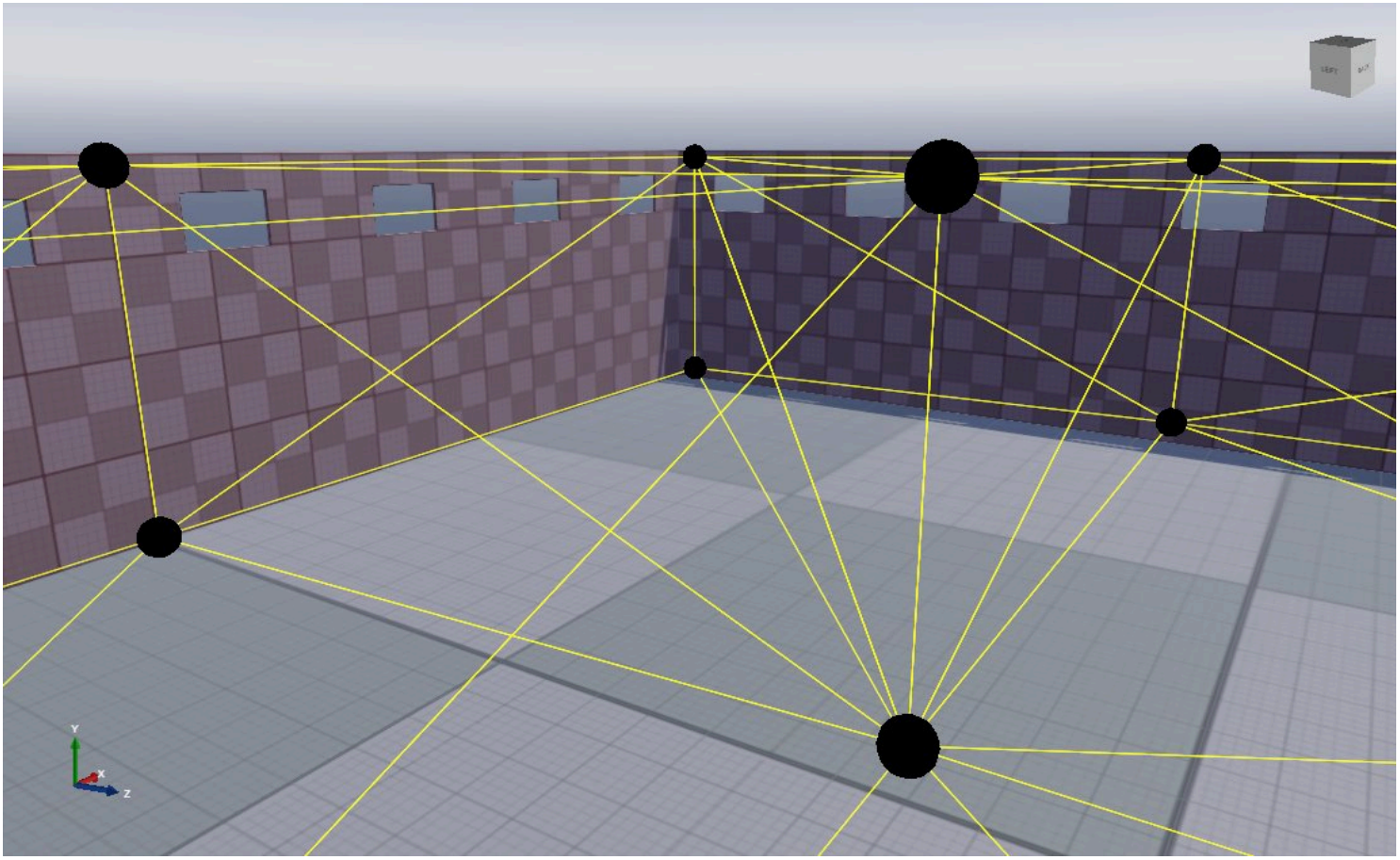
Right-click the scene or Entity Tree and select **Light > Light probe**.



Alternatively, create an empty entity and add a **Light probe component** in the Property Grid.



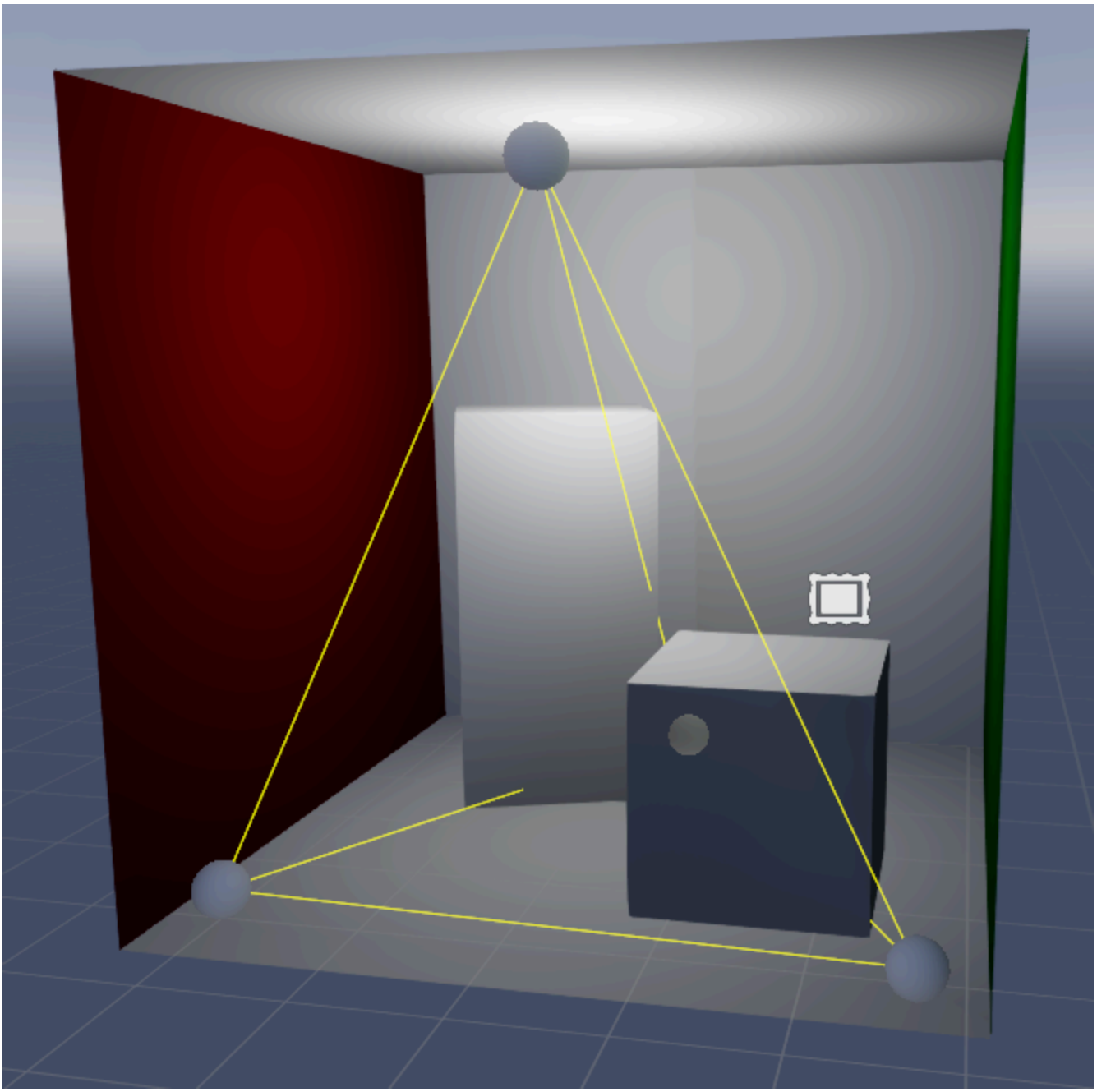
Light probes appear as spheres in the Scene Editor. Before you capture a light bounce for the first time, they have a completely black surface.



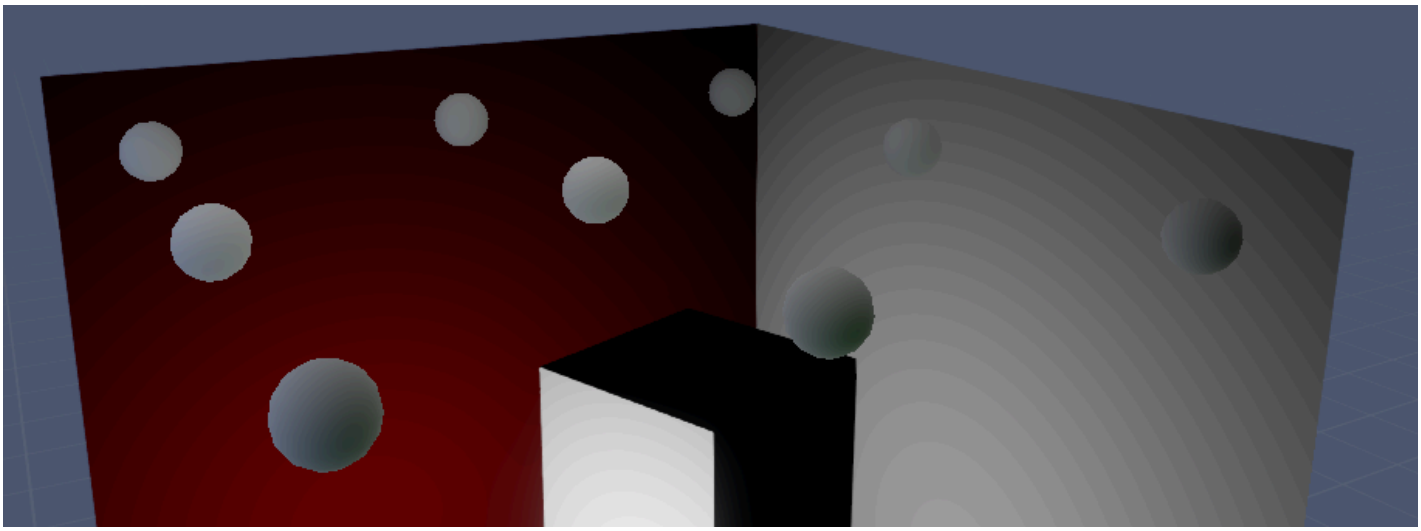
3. Place light probes

Light probes must be placed in a way that creates a **3D volume**. This means:

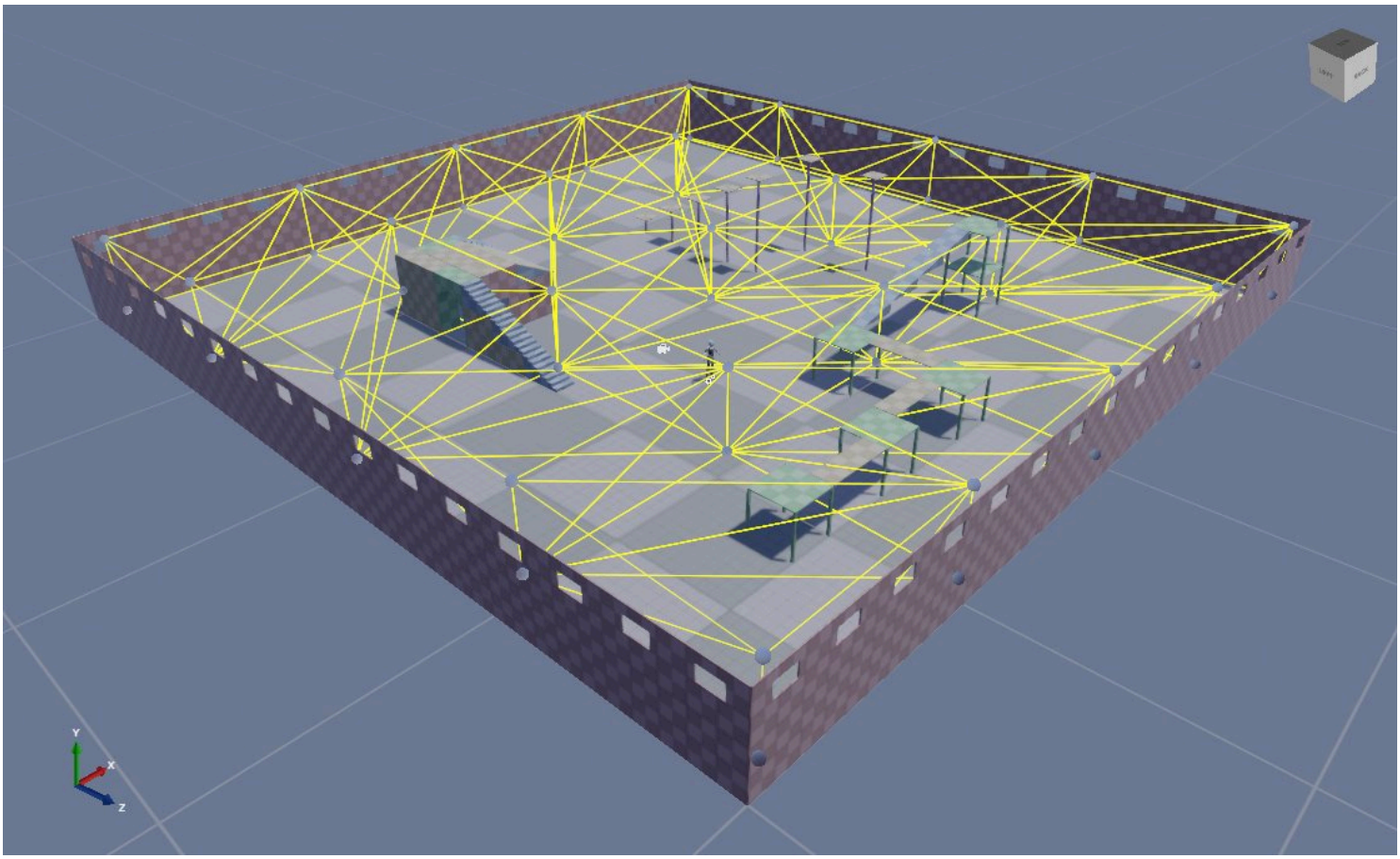
- You need **at least four light probes** in the scene — enough to create the four points of a tetrahedron, as below:

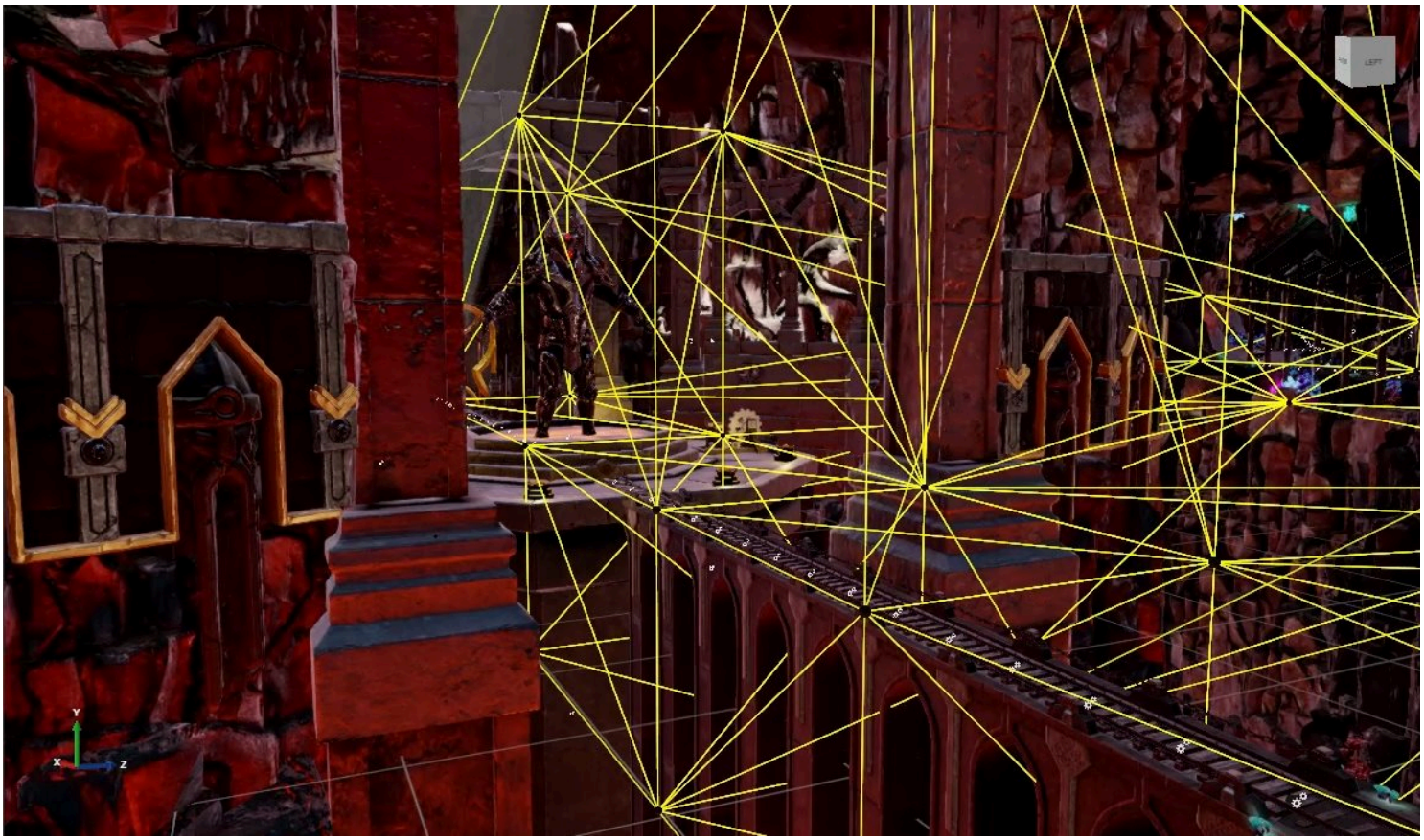


- At least one light probe must be on a different plane from the rest. For example, the probes in the screenshot below won't work, as they are on a flat plane and create no volume:



A typical method is to place light probes in a grid across your scene covering a general area, as in the screenshots below:

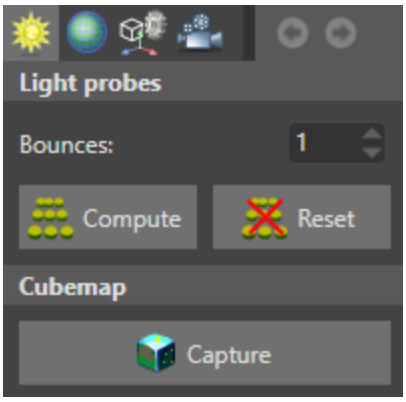




> **TIP**
>
> You can quickly duplicate light probes just like other entities. To do this, select a light probe, hold **Ctrl**, and move it with the mouse.

4. Capture lighting

1. In the Scene Editor toolbar, click the **lighting options** button to open the lighting options menu.



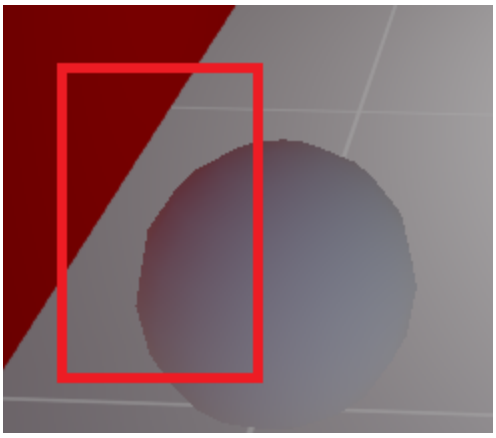
2. Next to **Bounces**, set the number of light bounces you want to capture.

   Multiple bounces simulate the effect of light bouncing between surfaces multiple times. This generally has the effect of brightening the lighting. Three or four bounces should be enough;

beyond this, changes are almost unnoticeable. The number of bounces has no impact on runtime performance.

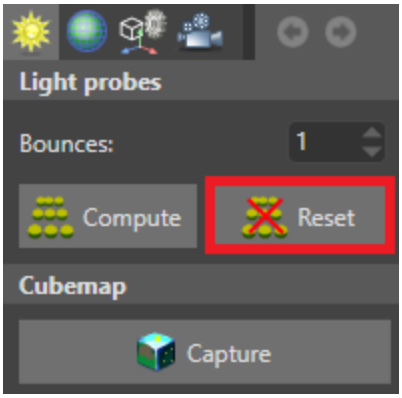
3. To capture the lighting, click **Compute**.

   You can see the lighting on the surface of the light probes in the Scene Editor.



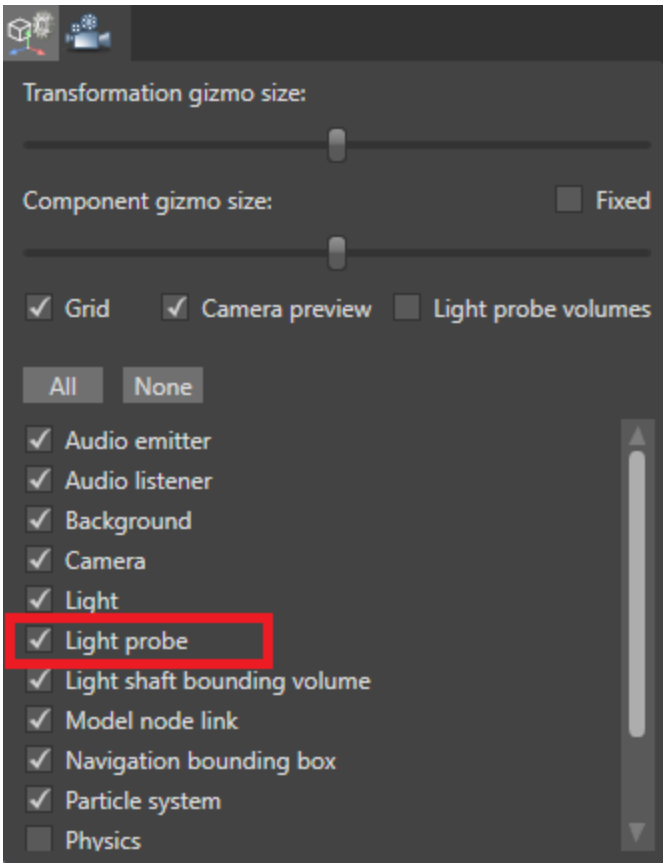
Reset light probes

To reset the light probes, in the **lighting options** menu, click **Reset**. This is useful after you change the lights in your scene and need to capture the lighting from scratch.



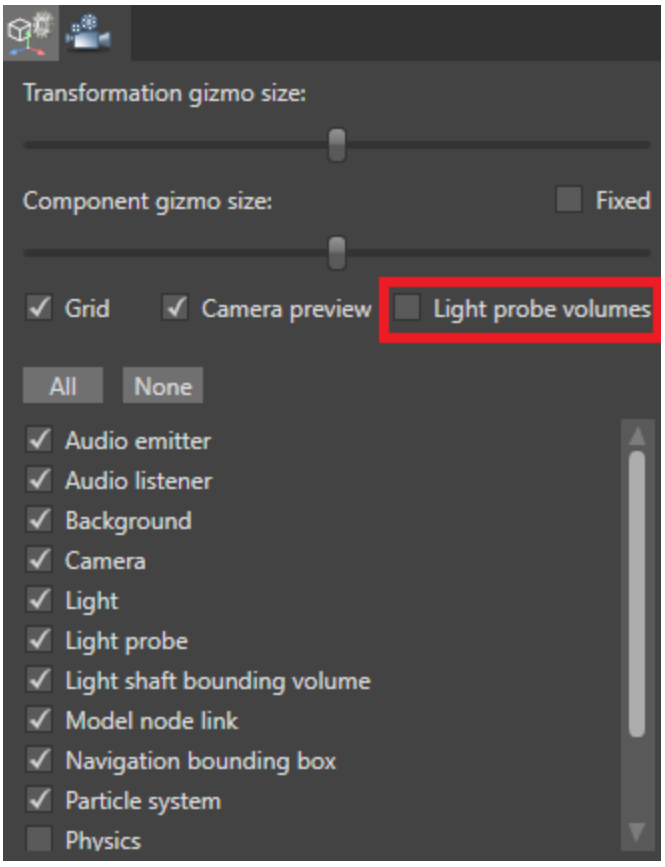
Show and hide light probes in the Scene Editor

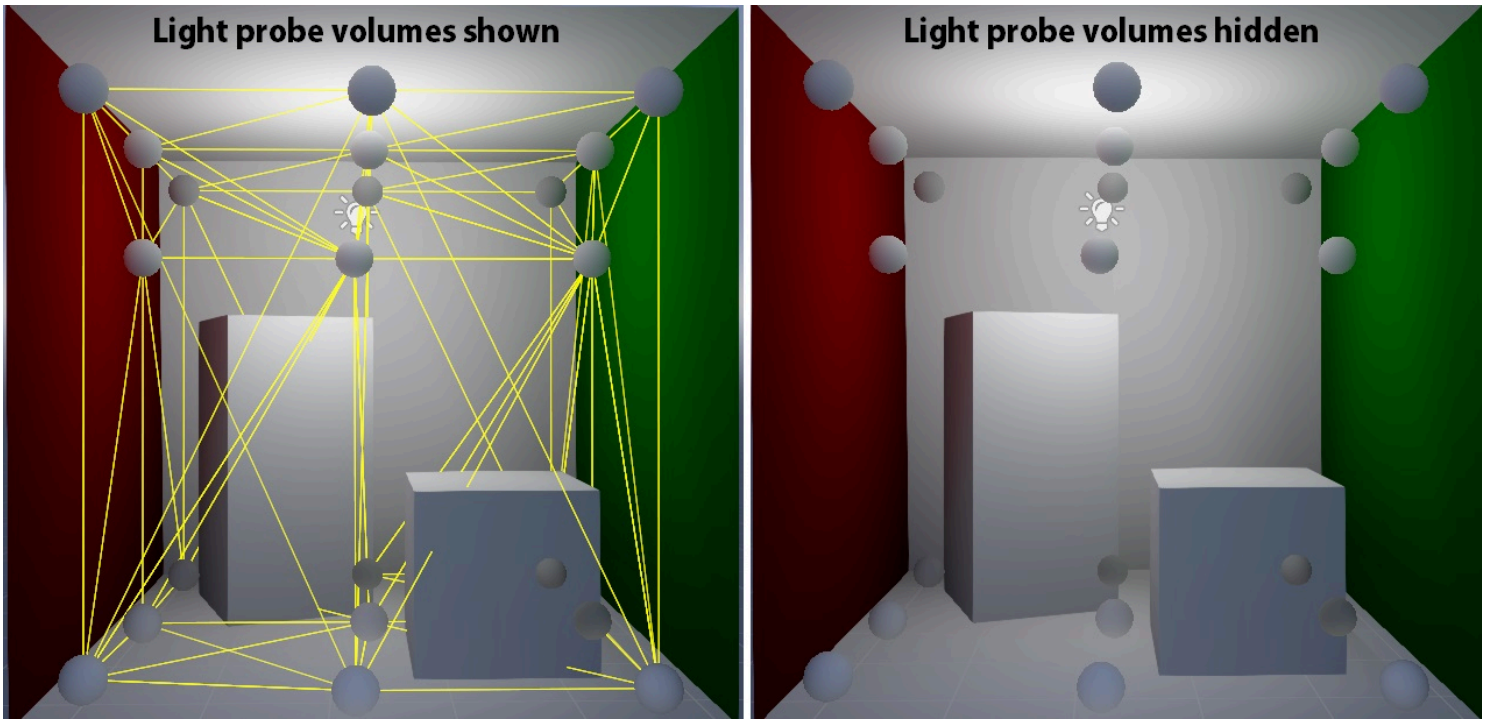
Under the **gizmo options** in the Scene Editor toolbar, use the **Light probes** checkbox.



Show and hide light probe volumes in the Scene Editor

Under the **gizmo options** in the Scene Editor toolbar, use the **Light probe volumes** checkbox.





See also

- Add a light
- Point lights
- Ambient lights
- Directional lights
- Skybox lights
- Spot lights
- Shadows

Light shafts

Beginner Designer Artist

Light shafts, also called **god rays**, are visible rays of light.

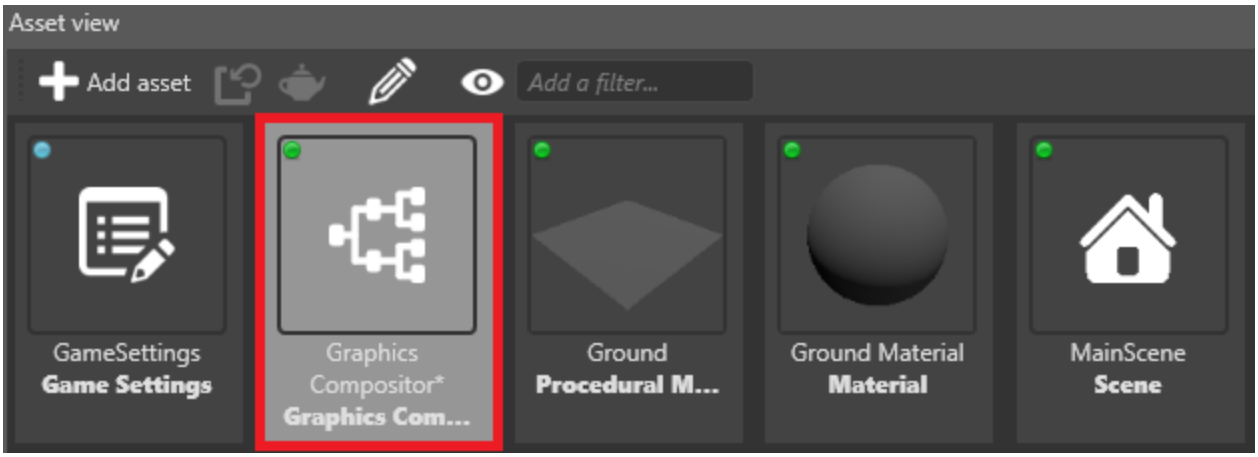
Stride light shafts are based on shadow maps and use raymarching rather than post effects, so they're visible even when the light source isn't. Any light source that casts shadows (ie point lights, directional lights and spot lights) can cast light shafts.

To create light shafts, use three components together: **lights**, **light shafts**, and **light shaft bounding volumes**.

1. Enable light shafts in the graphics compositor

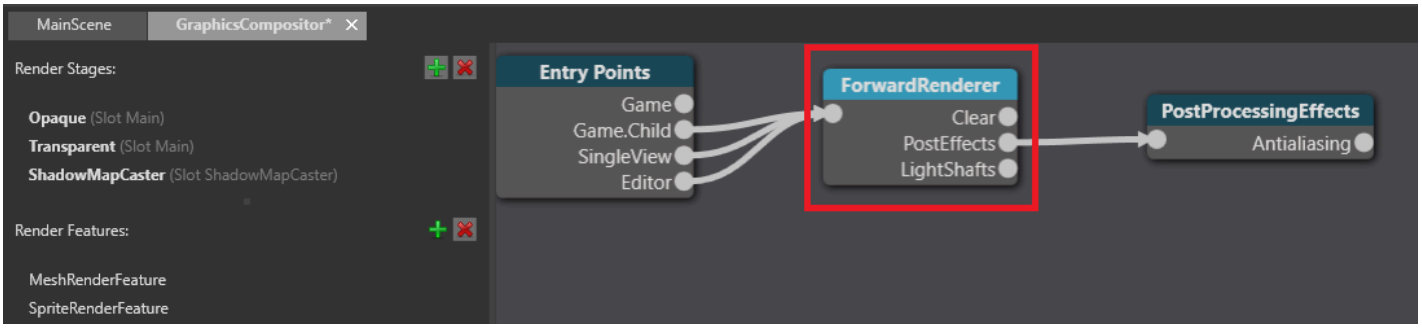
By default, Stride disables light shafts in new projects. To enable them:

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

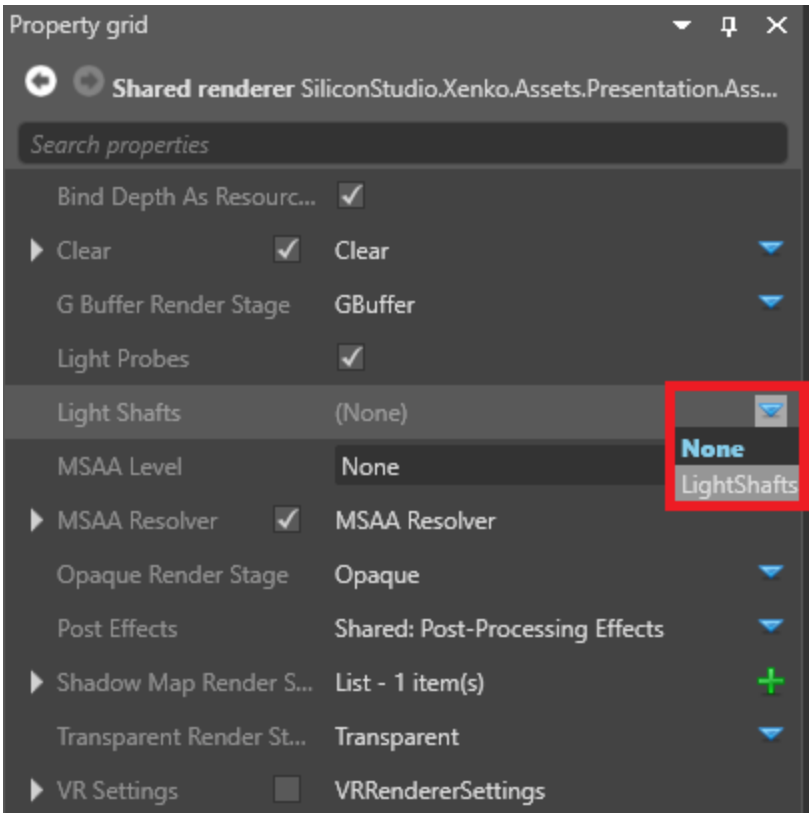


The graphics compositor editor opens.

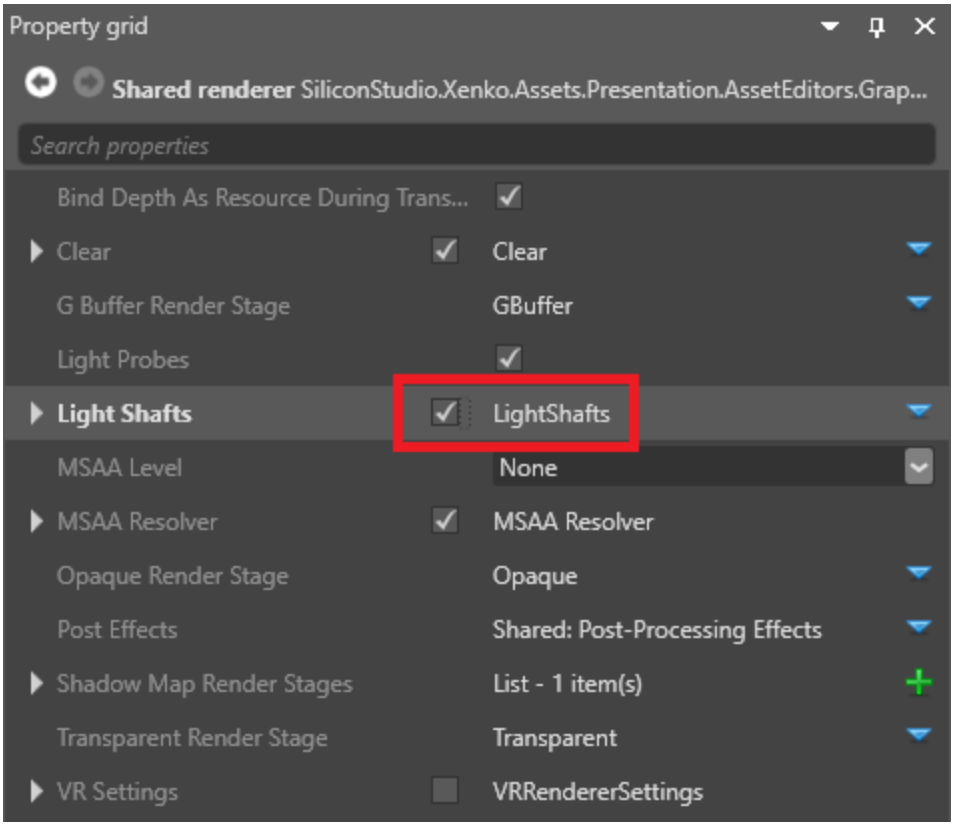
2. Select the **forward renderer** node.



3. In the **Property Grid** (on the right by default), next to **Light shafts**, click (**Replace**) and select **LightShafts**.



4. Make sure the **light shafts** checkbox is selected.

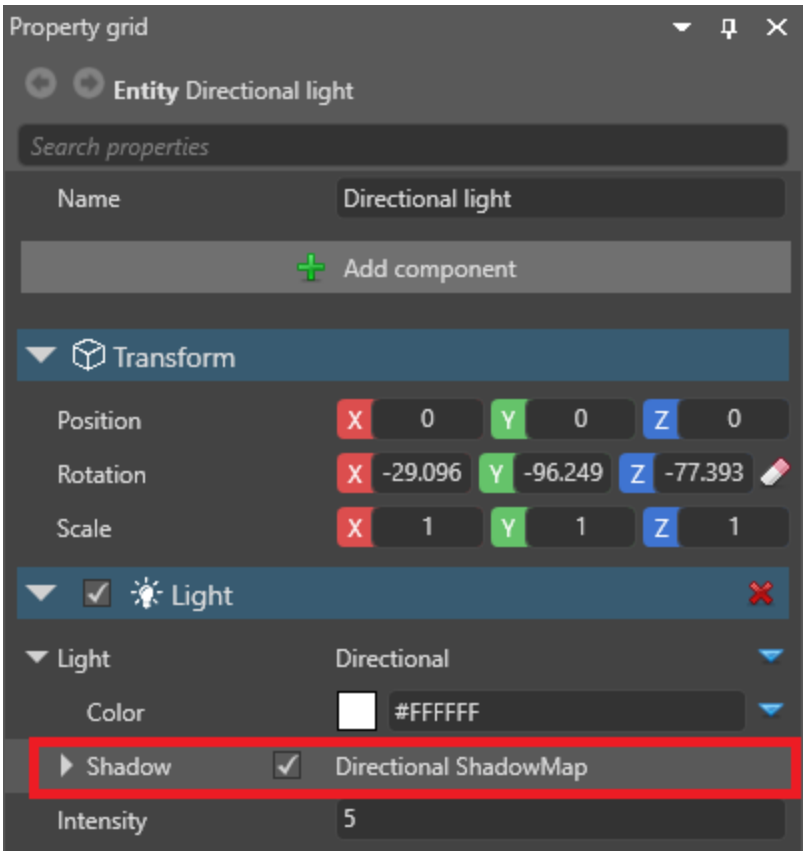


For more information about the graphics compositor, see the Graphics compositor page.

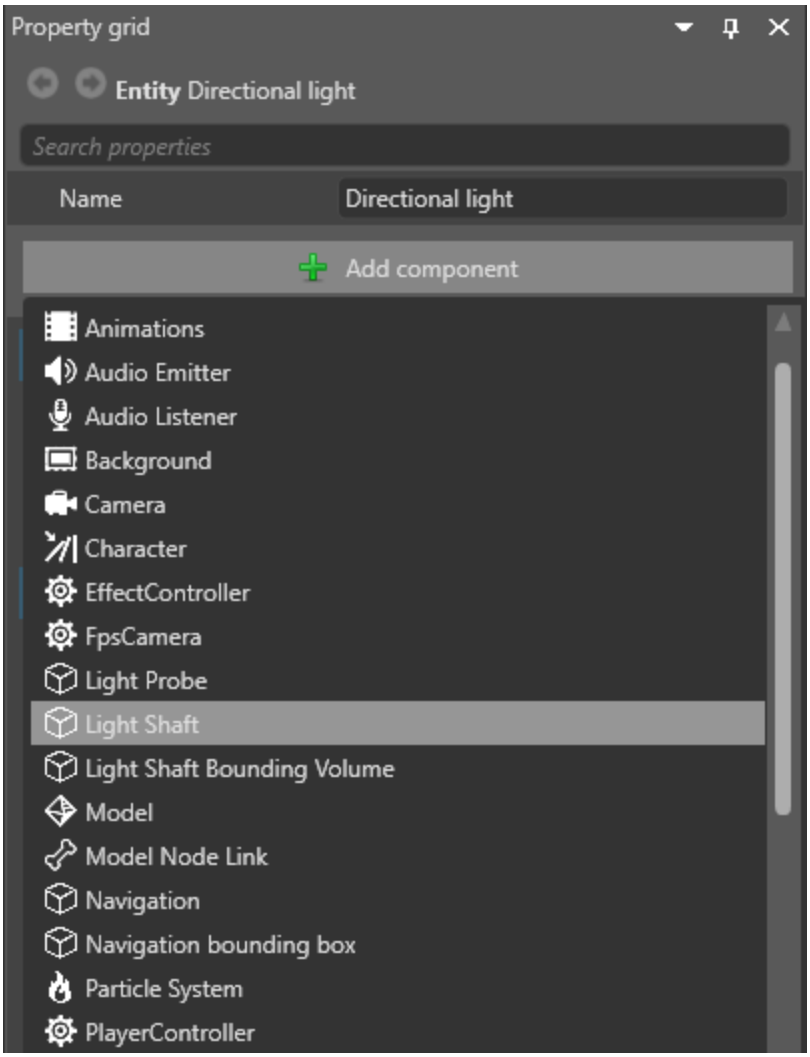
2. Add a light shaft component

1. In your scene, select the entity with the **light** you want to create light shafts. This must be a light that casts shadows (ie a point light, directional light, or spot light).

2. In the **Property Grid**, in the **Light** component properties, make sure the **Shadow** checkbox is selected.



3. Click **Add component** and select **Light shaft**.

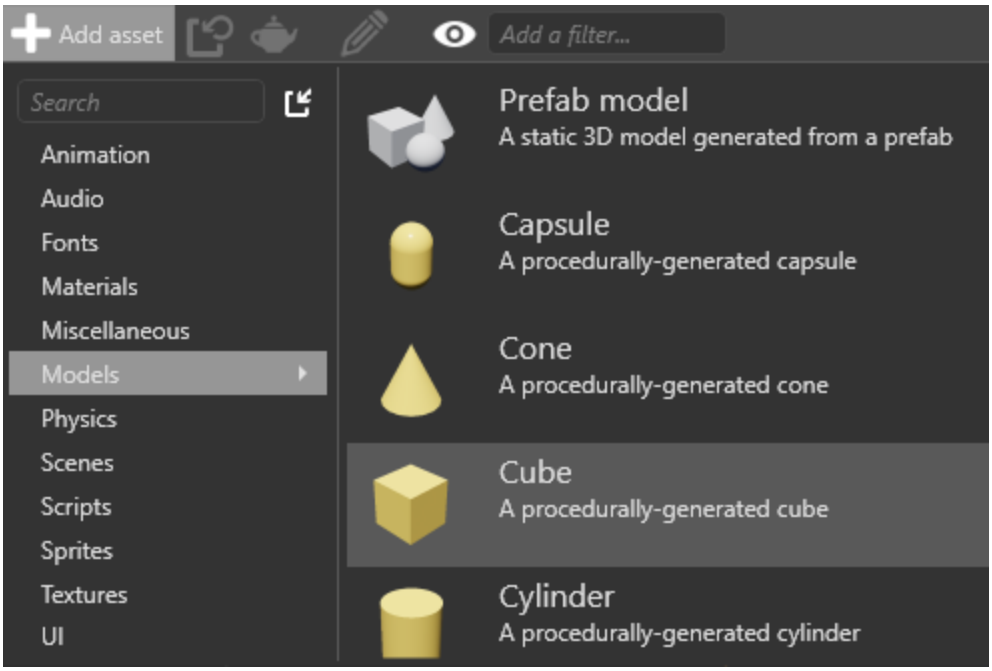


Game Studio adds a light shaft component to the entity.

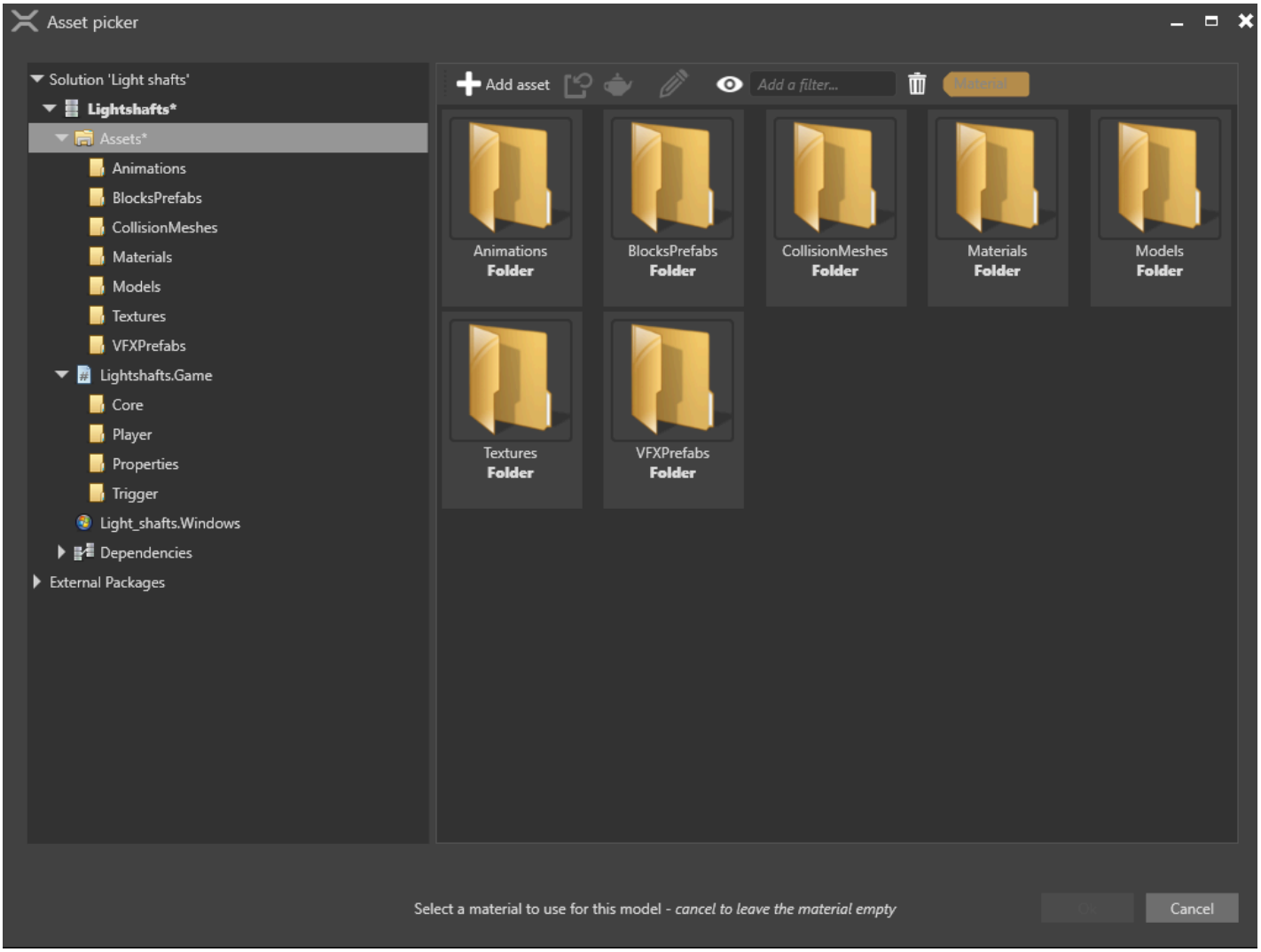
3. Add a bounding volume

The **light shaft bounding volume** defines the area in which light shafts are created. You can add the bounding volume to the same entity that has the directional light, but it's usually simpler to add it to a separate entity.

1. In the **Asset View**, click **Add asset**.
2. Under **Models**, select a model in the shape you want the volume to be. For example, if you use a cube, light shafts will be created in a cube-shaped area.



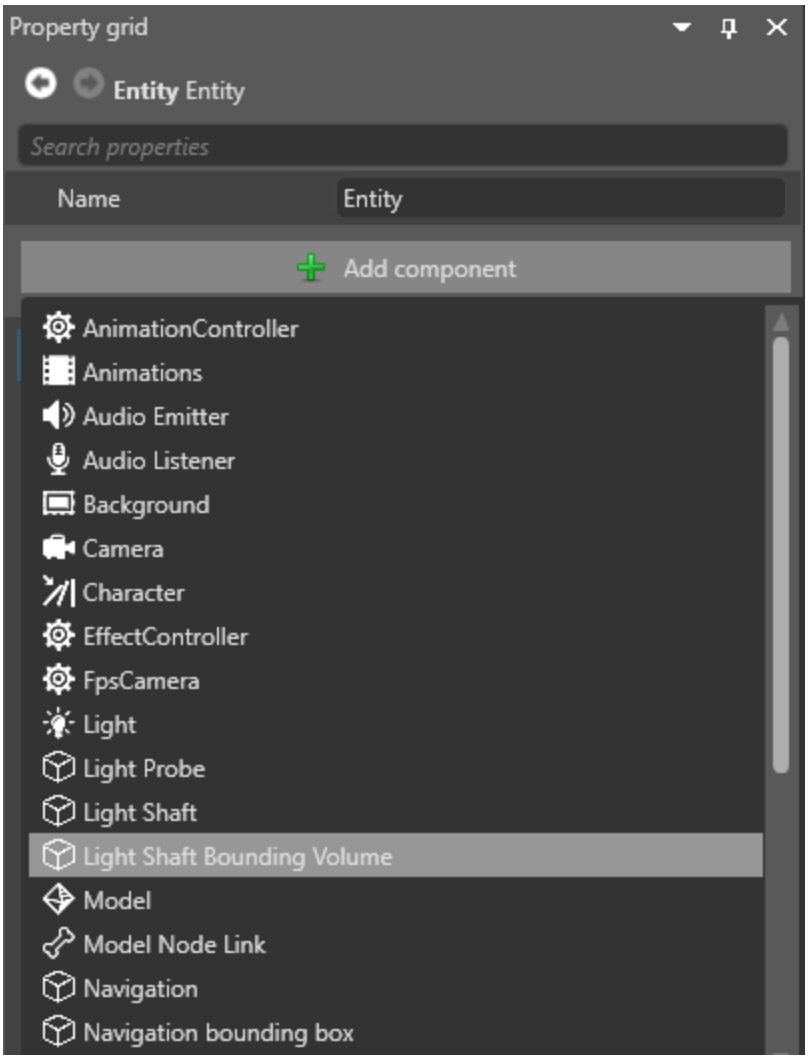
The **Select an asset** window opens.



3. You don't need a material for the model, so click **Cancel** to create a model without a material.

4. In the scene, create an empty **entity**. For now, it doesn't matter where you put it; you can reposition it later.

5. With the entity selected, in the **Property Grid**, click **Add component** and select **light shaft bounding volume**.

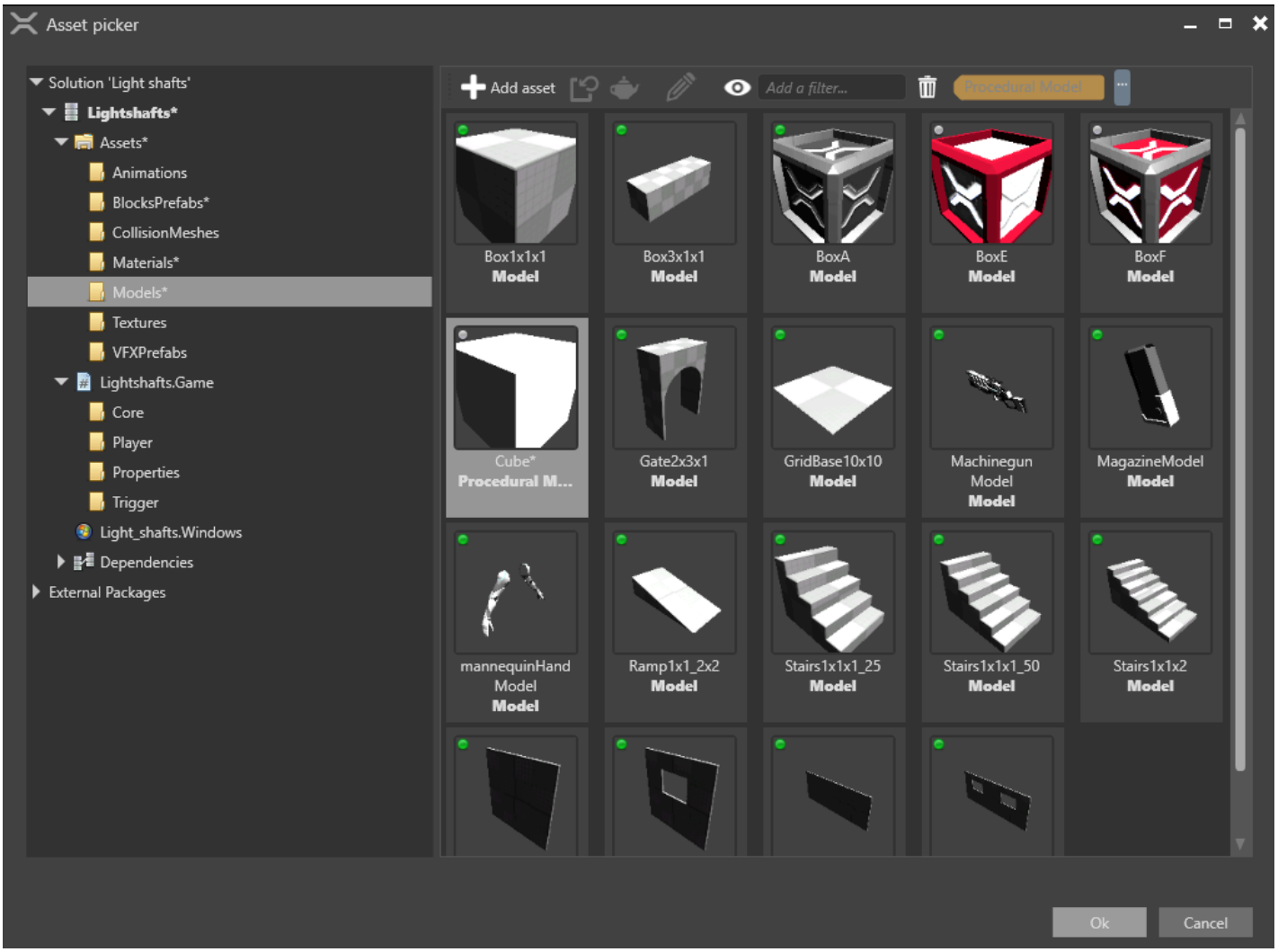


6. In the **light shaft bounding volume** component properties, next to **light shaft**, click (**Select an asset**).

7. In the **entity picker**, select the entity with the directional light you want to create light shafts and click **OK**.

8. In the **light shaft bounding volume** component properties, next to **Model**, click (**Select an asset**).

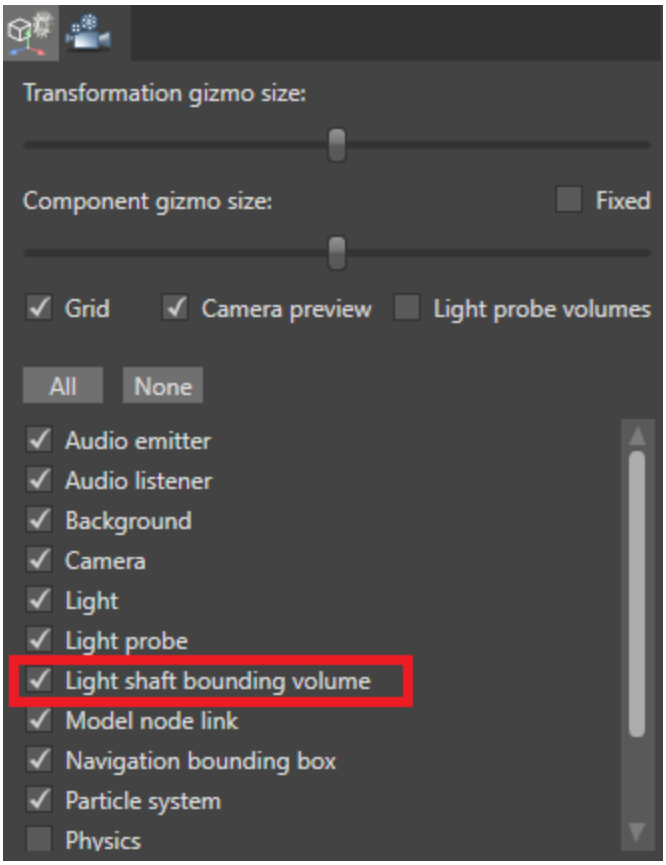
9. In the **Select an asset** window, select the model you created and click **OK**.



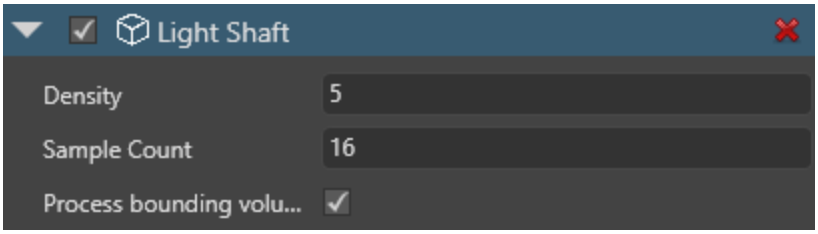
This model defines the shape of the light shaft bounding volume.

10. Using the **transform** component, position and scale the entity to cover the area where you want to create light shafts.

> **TIP**
>
> To show or hide navigation light shaft bounding volumes in the Scene Editor, in the **Scene Editor toolbar**, open the **gizmo options** menu and use the **Light shaft bounding volumes** checkbox.



Light shaft properties

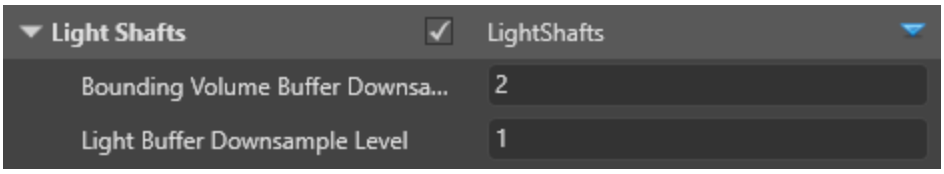


Property	Description
Density	Higher values produce brighter light shafts
Sample count	Higher values produce better light shafts but use more GPU
Process bounding volumes separately	Preserves light shaft quality when seen through separate bounding boxes, but uses more GPU

Light shaft graphics compositor properties

To access these properties, in the **graphics compositor editor**, select the **forward renderer** node and expand **Light Shafts**.

These properties apply globally to all the light shafts in the scene.

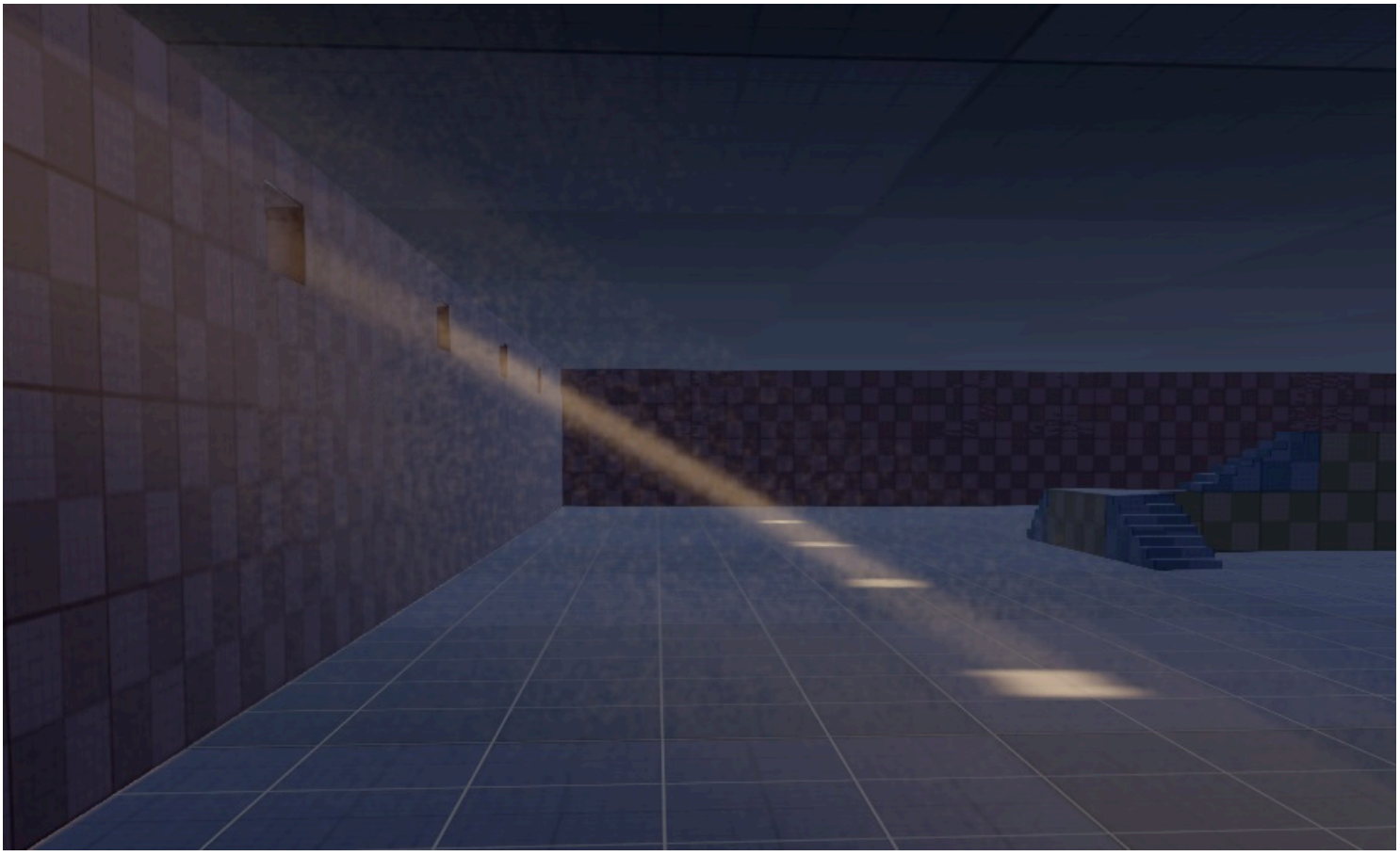


Property	Description
Bounding volume buffer downsample level	Lower values produce more precise volume buffer areas, but use more GPU
Light buffer downsample level	Lower values produce sharper light shafts, but use more GPU

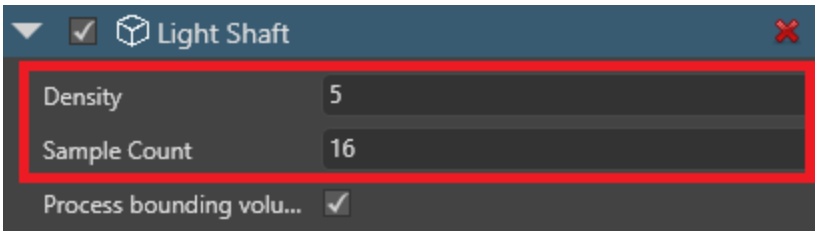
Optimize light shafts

Light shafts work best in dark environments. You can adjust the light and light shaft component properties to achieve different results — for example, by changing the light color (in the **light component properties**) or the light shaft density (in the **light shaft component properties**).

Multiple light shafts viewed through one another can become visually noisy, as in the screenshot below:

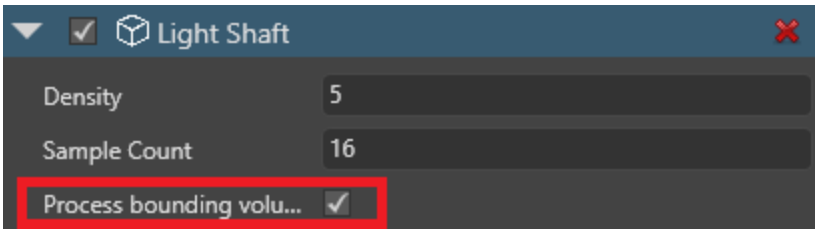


To reduce this effect, in the **light shaft component properties**, reduce the **density** and increase the **sample count**.



Alternatively, add additional bounding volumes and process them separately. To do this:

1. Create additional bounding volumes and position them to cover the areas where you want to create light shafts. Make sure the bounding volumes don't overlap, as this makes light shafts extra-bright.
2. In the **light shaft component properties**, make sure **Process bounding volumes separately** is selected.



> **NOTE**
> Processing bounding volumes separately uses more GPU.

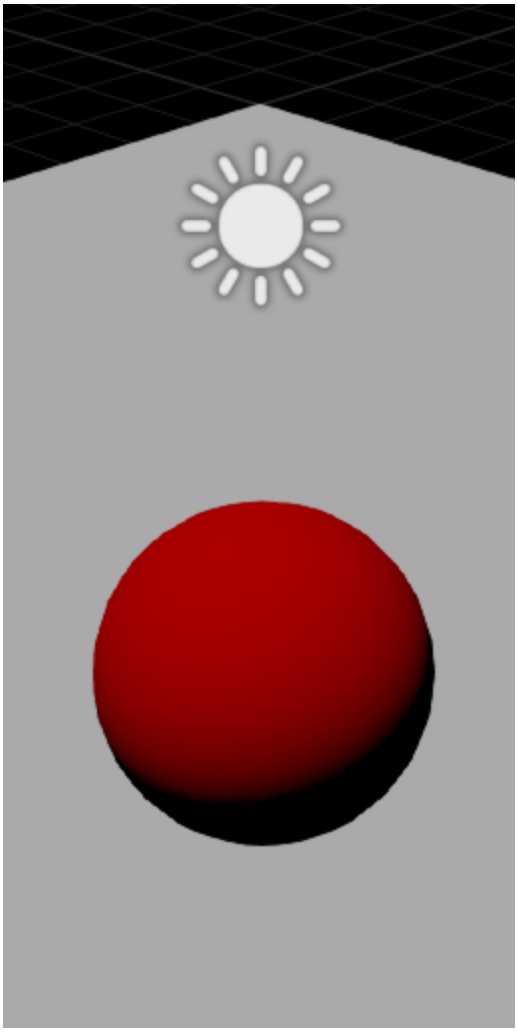
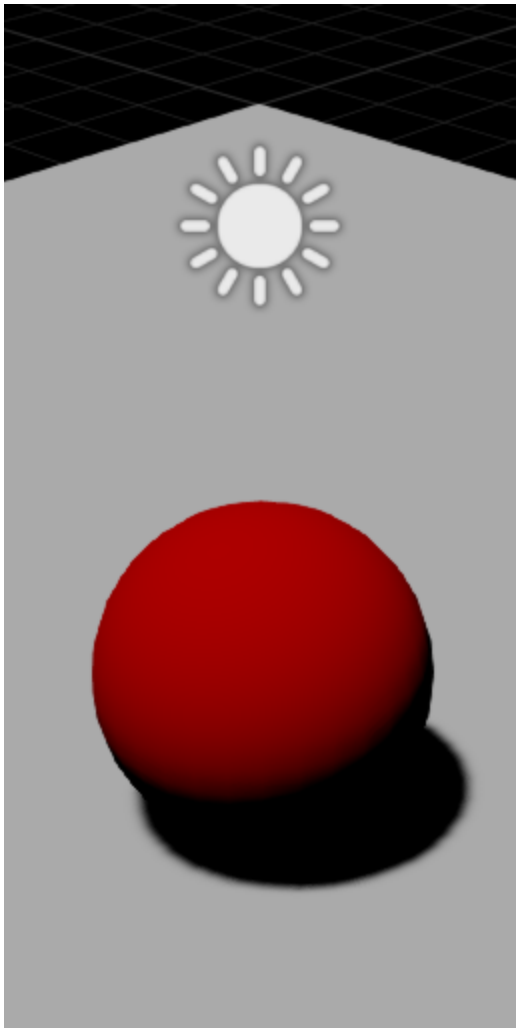
See also

- Directional lights
- Shadows
- Graphics compositor

Shadows

Beginner Designer Artist

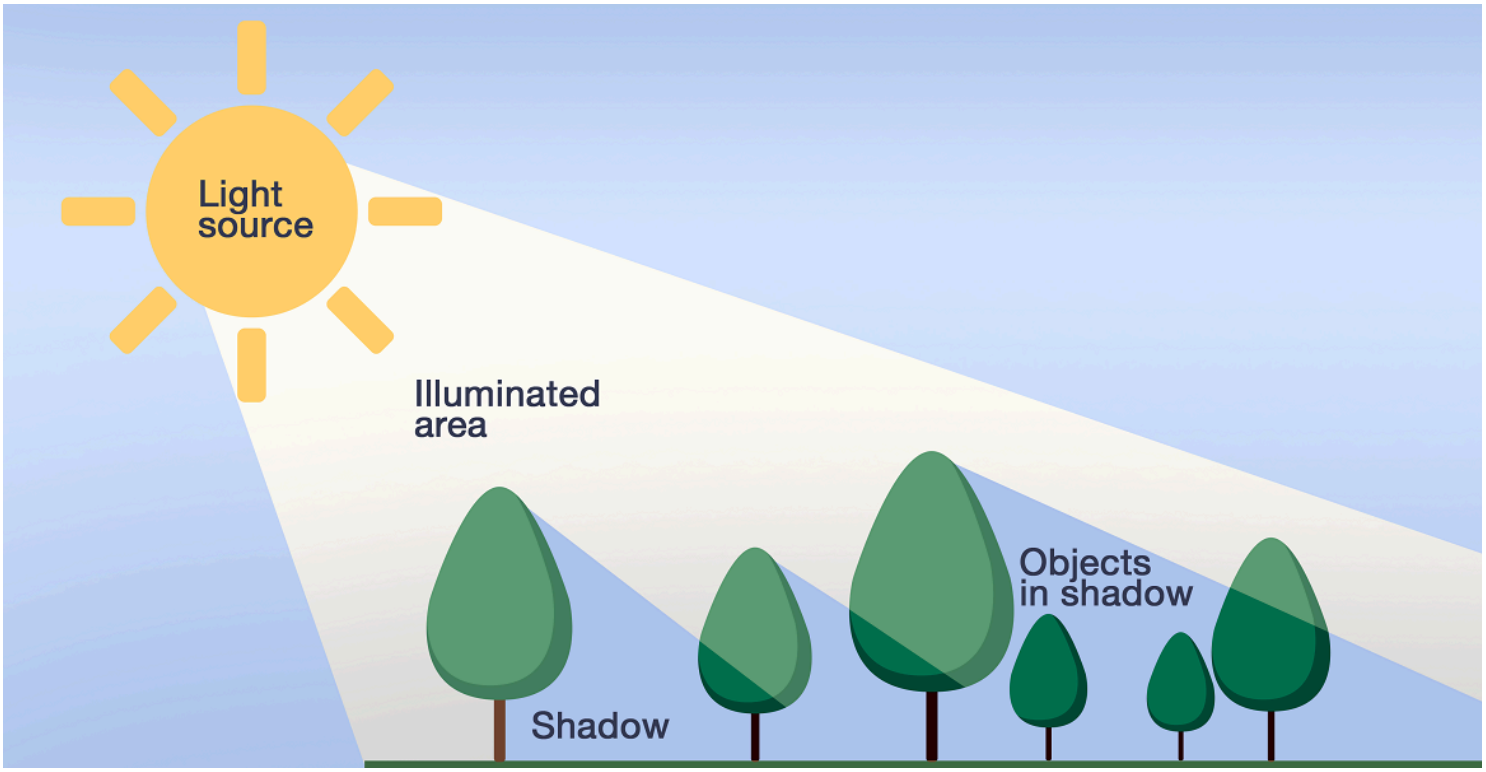
Shadows bring significant information and realism to a scene.

Shadows off	**Shadows on**
	

Only directional lights, point lights, and spot lights can cast shadows.

Shadow maps

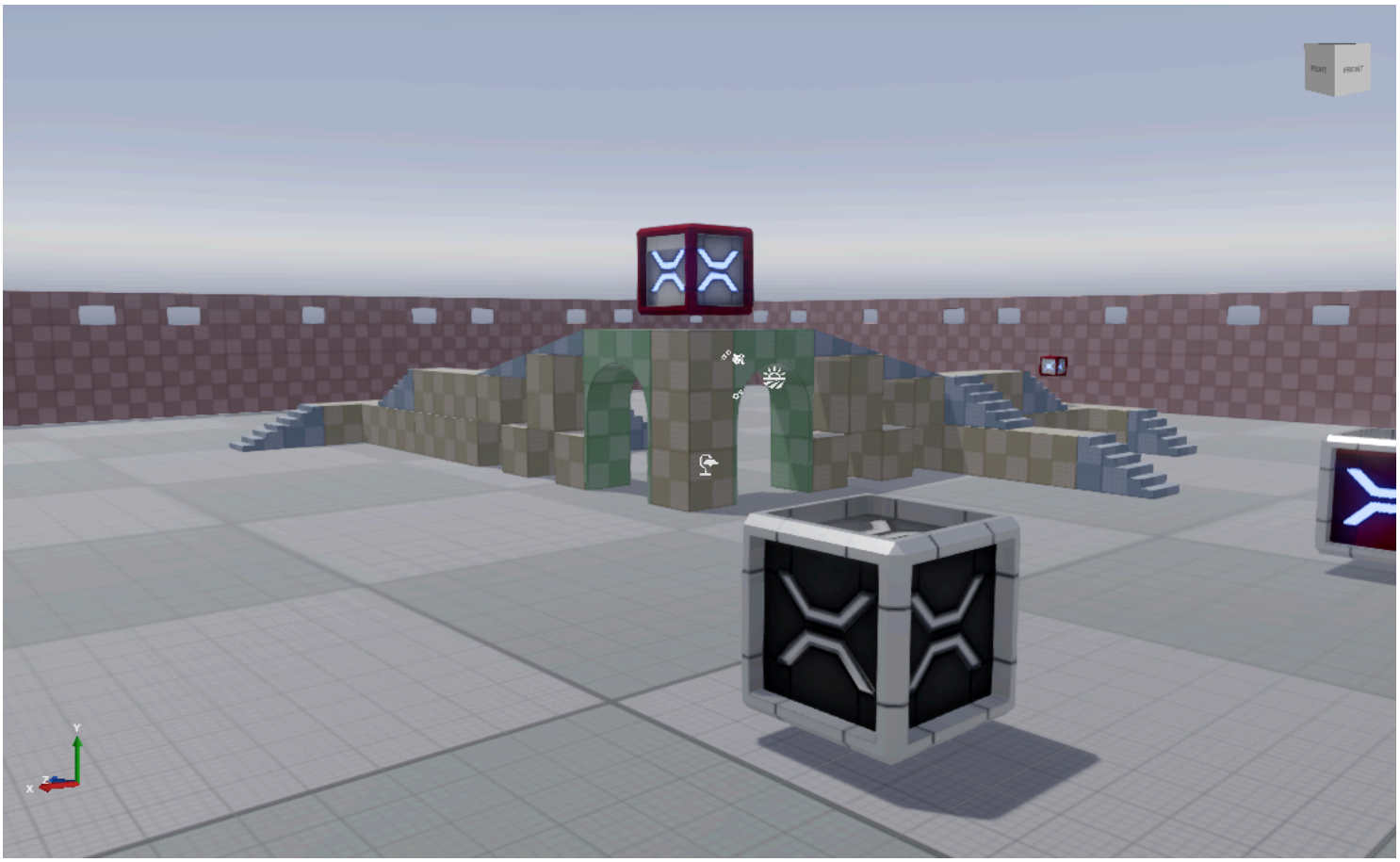
Stride uses **shadow mapping** to render shadows. To understand shadow maps, imagine a camera in the center of the sun, so you're looking down from the sun's perspective.

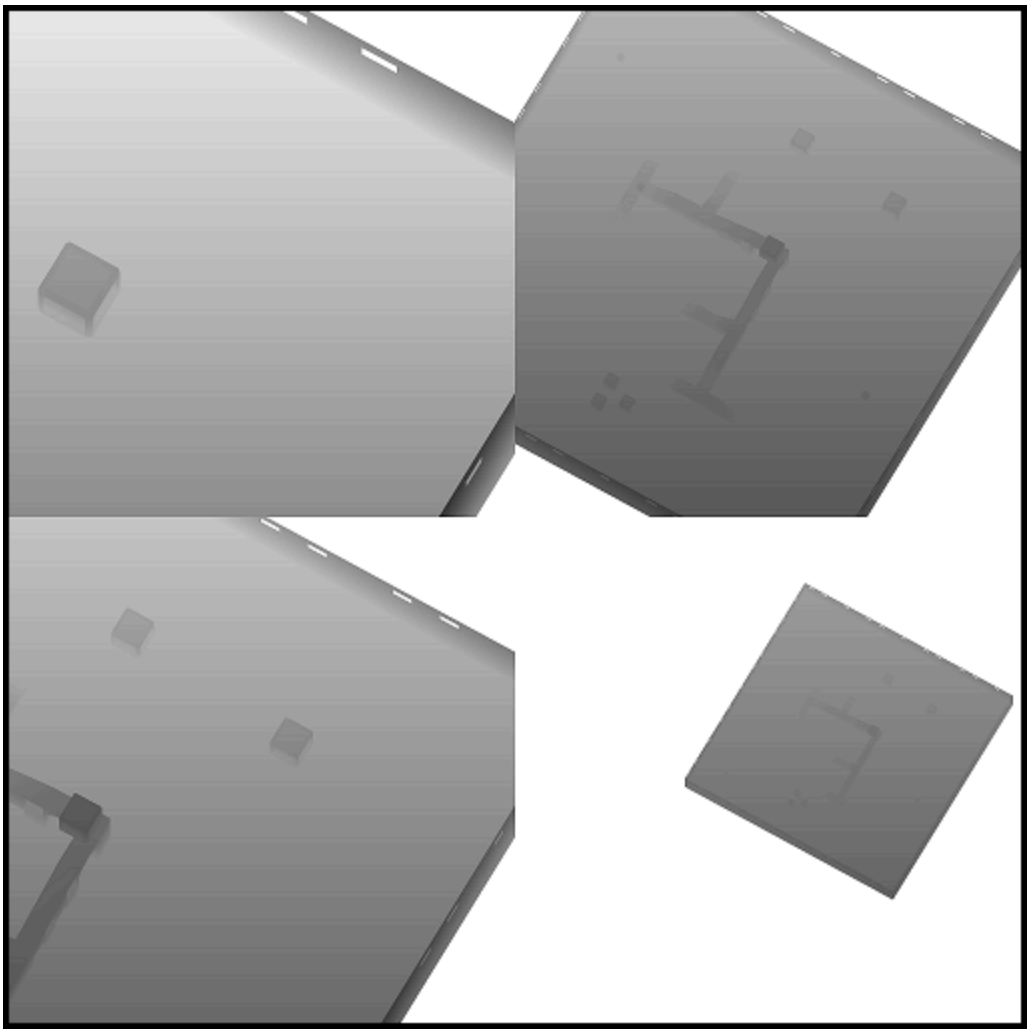


Everything the sun sees is in light. Everything hidden from the sun (ie behind **occluders**) is in shadow.

From this perspective, Stride creates a **shadow map** for each light that casts shadows. This tells us how far each visible pixel is from the light. When Stride renders the scene, it checks the position of each pixel in the shadow map to learn if it can be "seen" by the light. If the light can see the pixel, the light is illuminated. If it can't, the pixel is in shadow.

For example, these are shadow maps from the first-person shooter sample included in Stride, generated by a directional light.

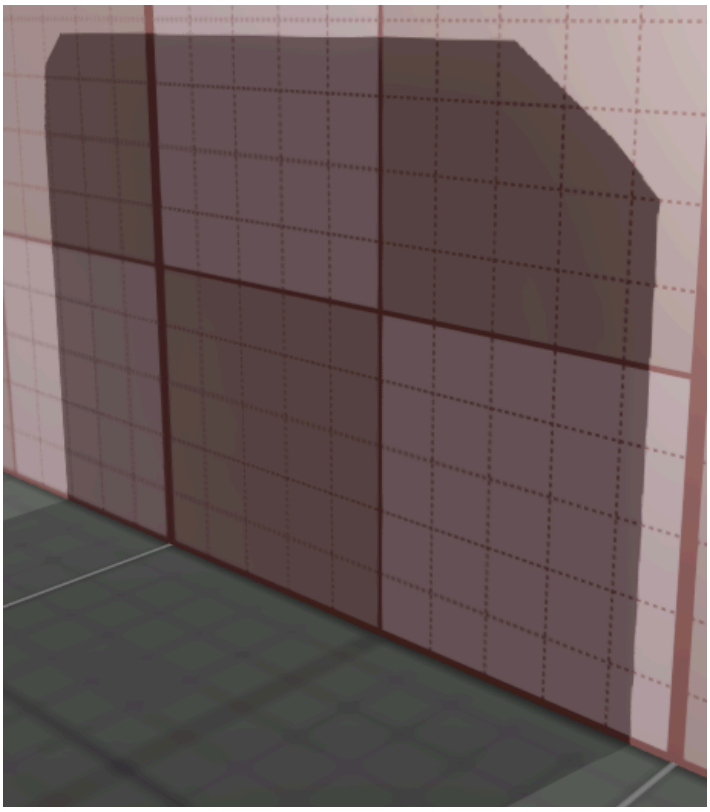
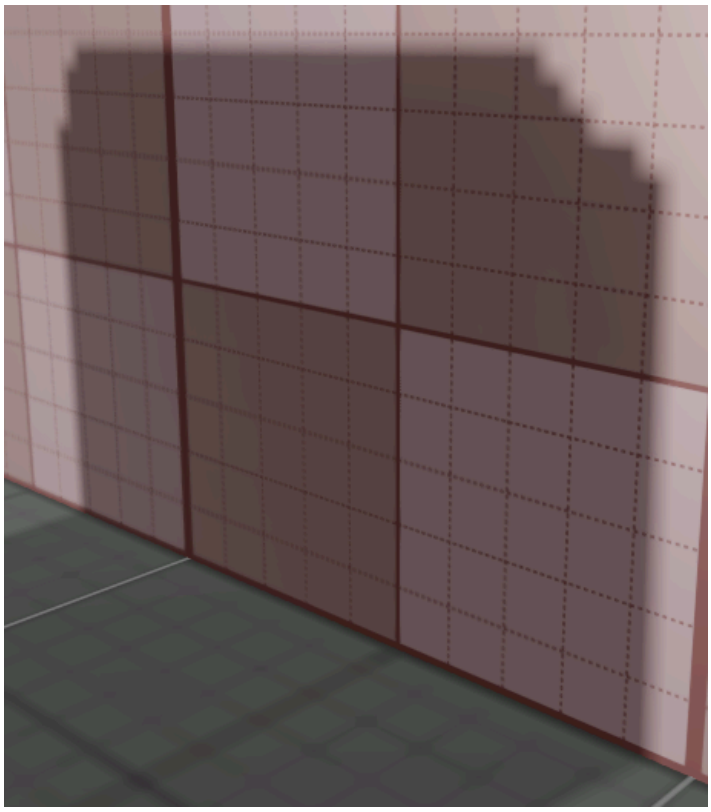
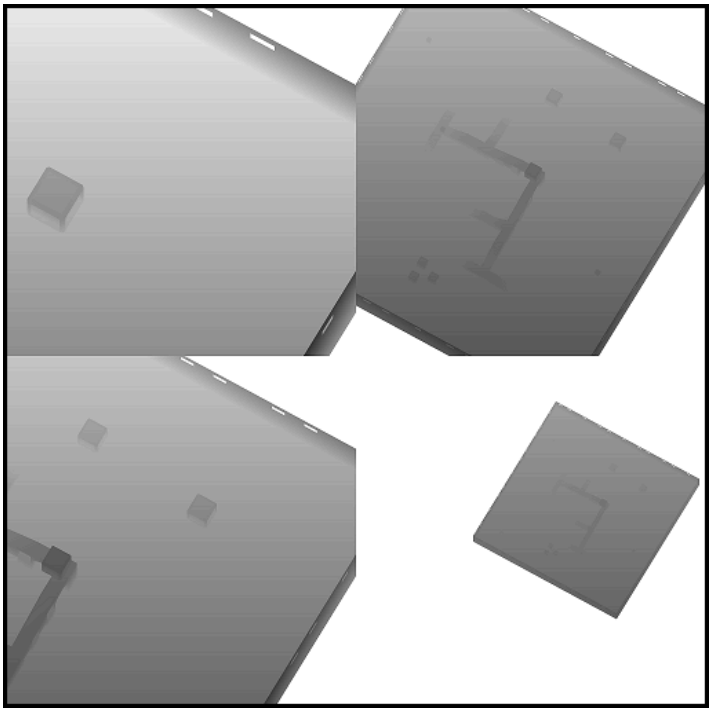
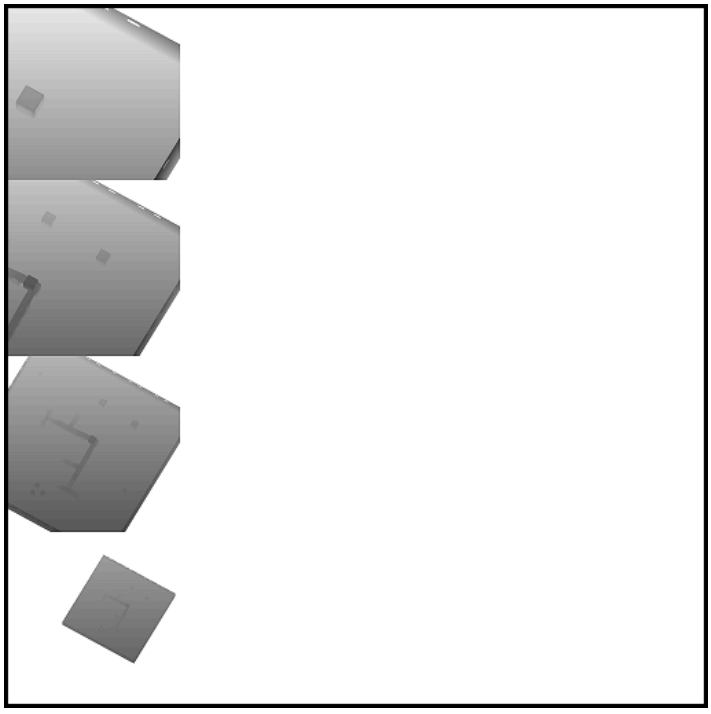




> **NOTE**
>
> Note that the directional light in the example above creates four shadow maps, one for each cascade. For more information, see the Directional lights page.

The shadow atlas

Shadow maps for each light that casts a shadow are saved in a region of the **shadow atlas** texture. You can choose how much of the shadow atlas each light uses. The larger the shadow map, the better the shadow quality, but the less space you have for shadow maps from other light sources.

Higher-quality shadow (uses a large area of the shadow atlas)	Lower-quality shadow (uses a smaller area of the shadow atlas)
	
	

Generally, you should give more space to light sources that cast the most visible shadows.

The size of each area in the shadow map depends on several factors:

- the `shadowMapSizeFactor` based on the `LightShadowMap.Size` property (/8, /4, /2, x1, or x2)

- the projected size of the light in screenspace (`lightSize`)
  - for directional lights, the lightSize is equal to the max (screenWidth, screenHeight)
  - for spot lights, the lightSize is equal to the projection of the projected sphere at the target spot light cone
- the `ShadowMapBaseSize` equals `1024`

The final size of the shadow map is calculated like this:

```
// Calculate the size factor
var shadowMapSizeFinalFactor = shadowImportanceFactor * shadowMapSizeFactor;
// Multiply the light projected size by the size factor
var shadowMapSize = NextPowerOfTwo(lightSize * shadowSizeFinalFactor);
// Clamp to a maximum size
shadowMapSize = min(shadowMapSize, ShadowMapBaseSize  * shadowSizeFinalFactor);
```

If you've enabled shadows on a light in your scene, but it isn't casting shadows, make sure there's enough space in the shadow atlas to create a shadow map for the light. For more information, see Troubleshooting — Lights don't cast shadows.

See also

- Point lights
- Directional lights
- Spot lights
- Troubleshooting — Lights don't cast shadows

Voxel Cone Tracing Global Illumination

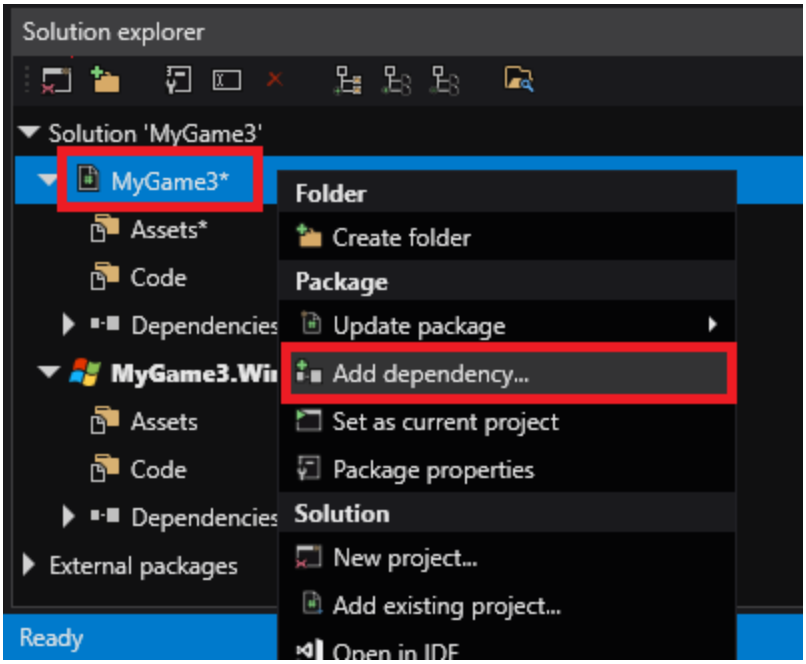
How to set up an existing project with Voxel Cone Tracing GI

Prerequesites:

VoxelGI requires Graphics Profile Level 11 or Higher (Direct3D 11.0 / OpenGL ES 3.1). This can be set in the Game Settings asset under the Rendering category.

Since Stride is modular, we need to add a reference to the `Stride.Voxels` plugin:

1. In the **Solution Explorer**, right-click on the user project
2. Select `Add Dependency`



3. Select `Stride.Voxels` in the list and press `OK`
4. Close and re-open the project.

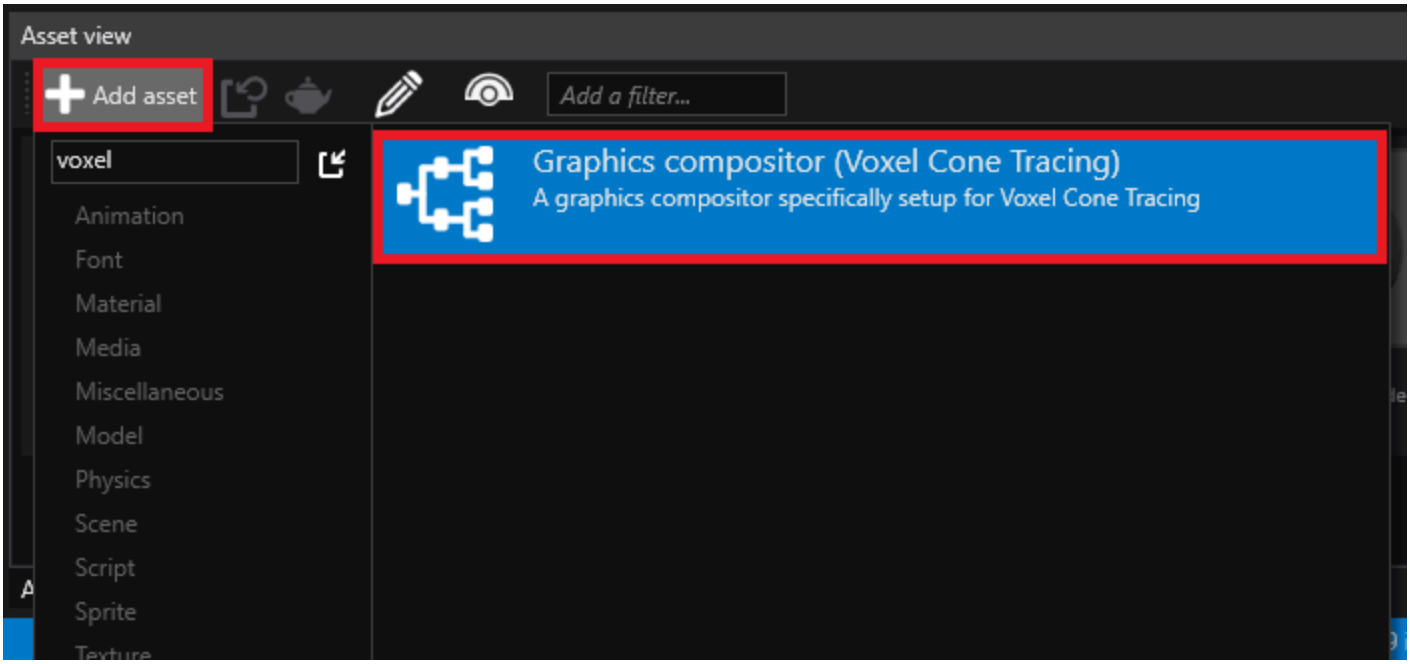
Graphics Compositor

Voxel Cone Tracing requires several changes to the graphics compositor.

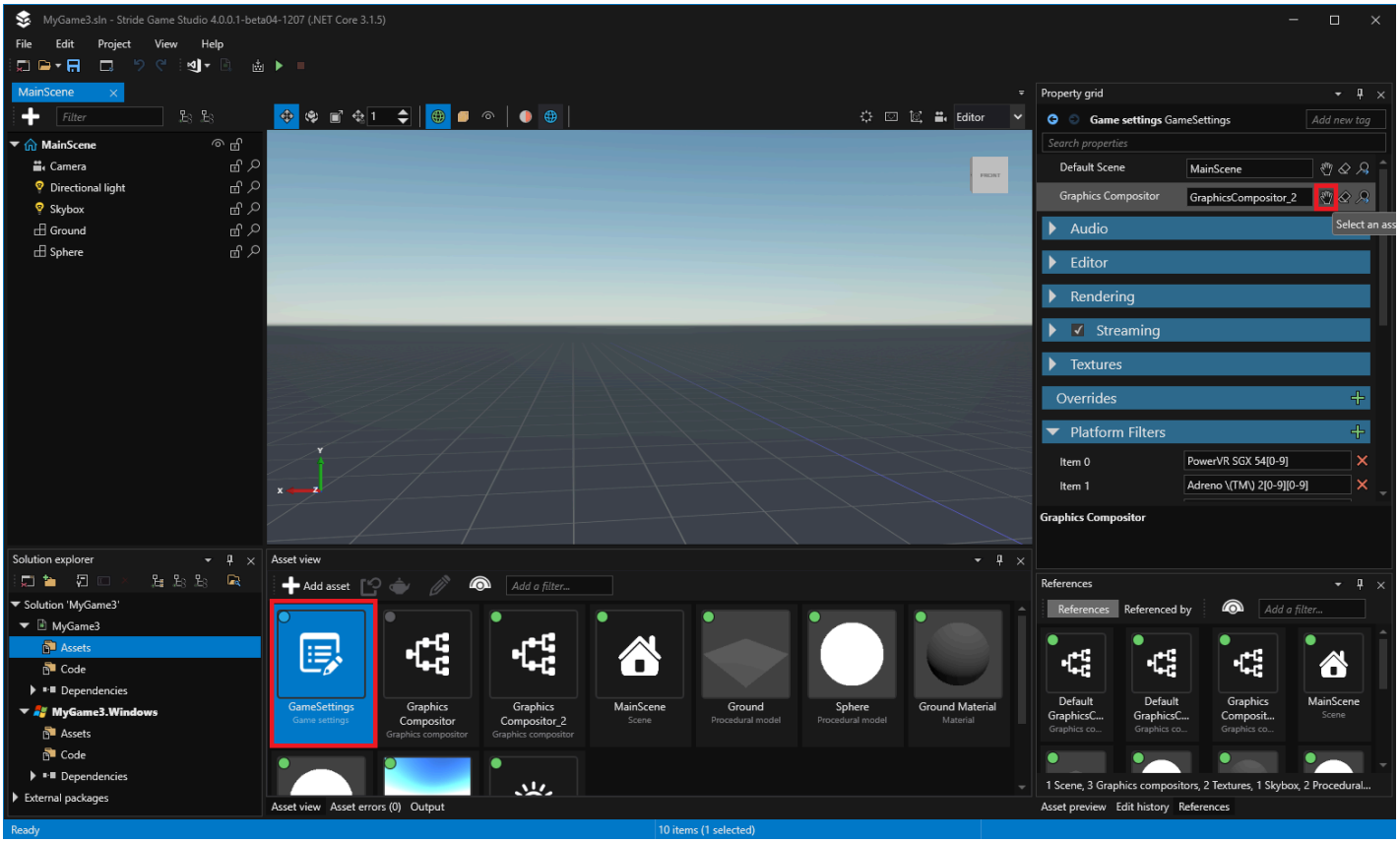
To make this easier, we prepared a graphics compositor ready to use with Voxel GI in the asset templates:

1. In the **Asset View**, click Add asset
2. Start to type `Voxel` in the search bar

3. Select `Graphics Compositor (Voxel Cone Tracing)`



4. In your `Game Settings` asset, change the graphics compositor to the newly created one:



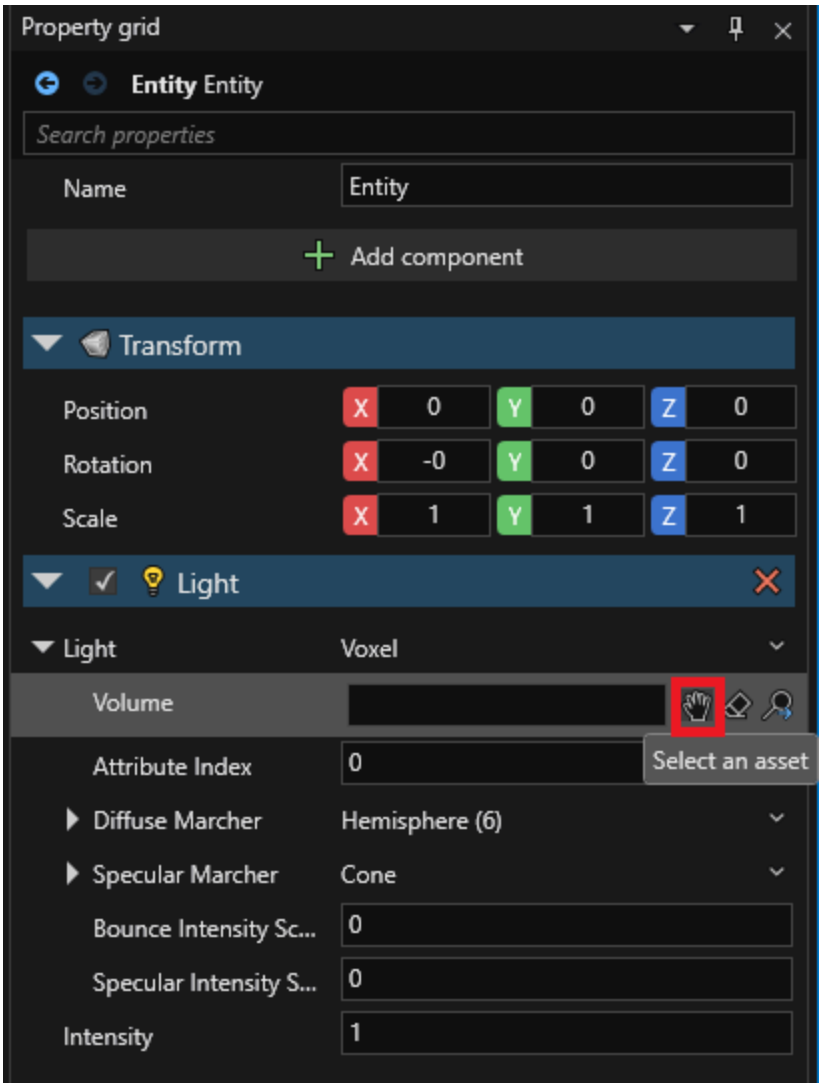
Setup scene: Volume and Light

1. In the scene explorer, above the **Entity Tree**, click the icon and select `Lights` then `Voxel volume`

2. Click the icon again and select `Lights` then `Voxel light`

   At that point, the scene rendering will likely crash due to the light not being setup correctly (with error `No Voxel Volume Component selected for voxel light.`), but that's expected.

3. In the property grid, change the Light Volume to the previously created entity:



4. At that point, you can click the `Resume` button in scene renderer, and everything should be setup!

Play with it

To do a quick test, you can disable Skybox light (keep only directional light), go in shadow area and see if some ambient light spread there. You can also play with emissive materials.

Video tutorial

Here's a youtube alternative tutorial made by David Jeske on how to set it up:

Howto setup Real Time Voxel Global Illumination in Stride3...

Post effects

Post effects are usually applied after your game has completed the rendering of a frame, but before the UI is drawn. You can use post effects to tune or embellish an image — for example, by producing a more natural, realistic look, or creating stylized cinematic effects.

Post effects are usually applied to an image. This means they have no connection with vertices or meshes. They only work with the color values of each pixel (and sometimes their depth).

Typically, you set up a post effect by specifying:

- input buffers (eg color, depth)
- one or several output buffers
- parameters to customize the behavior of the post effect during its rendering pass

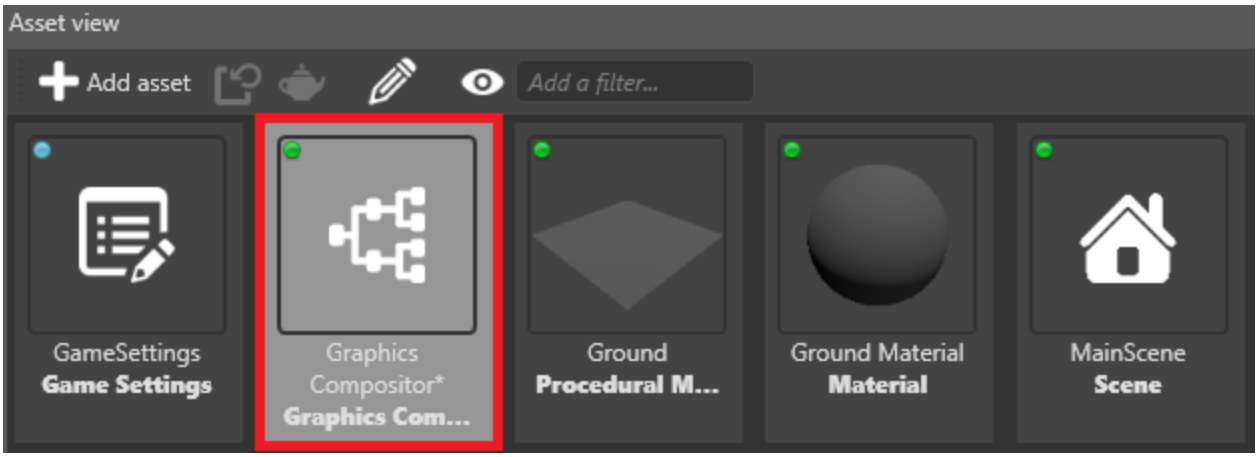
Stride provides several predefined post effects. You can also extend the system to create your own color transform effects.

> **NOTE**
>
> Depth-aware post effects_ie depth of field, ambient occlusion, and local reflections_nullify MSAA (multisample anti-aliasing).

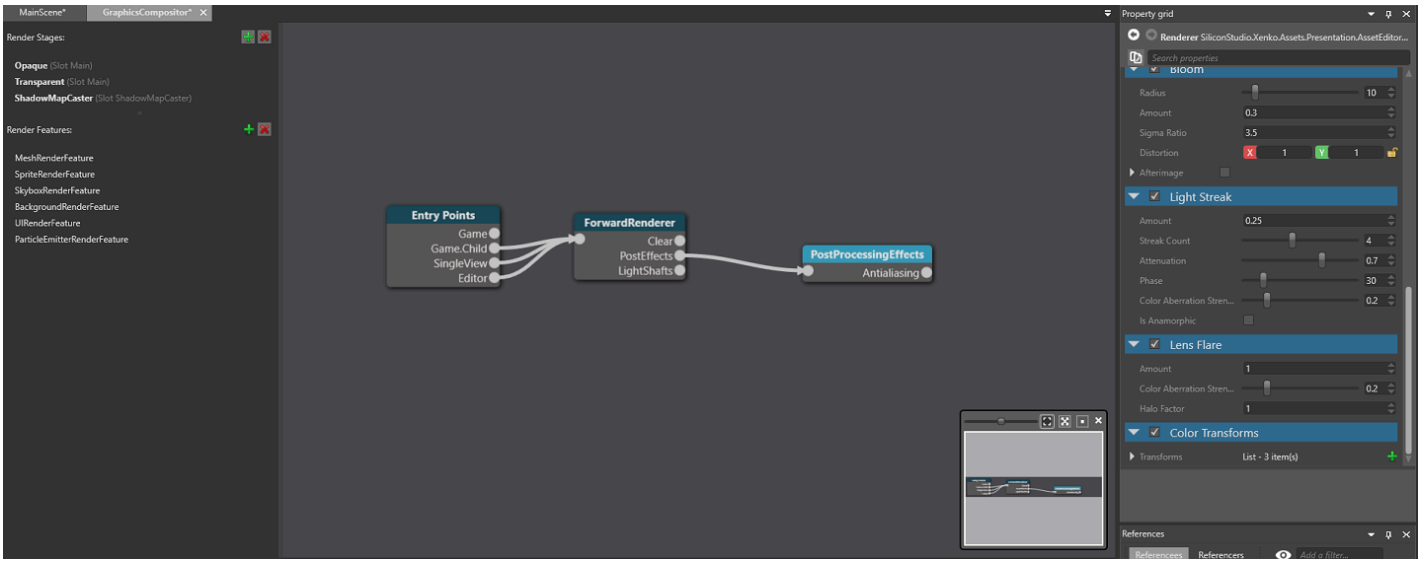
Add or edit a post effect

You add and edit post effects in the graphics compositor.

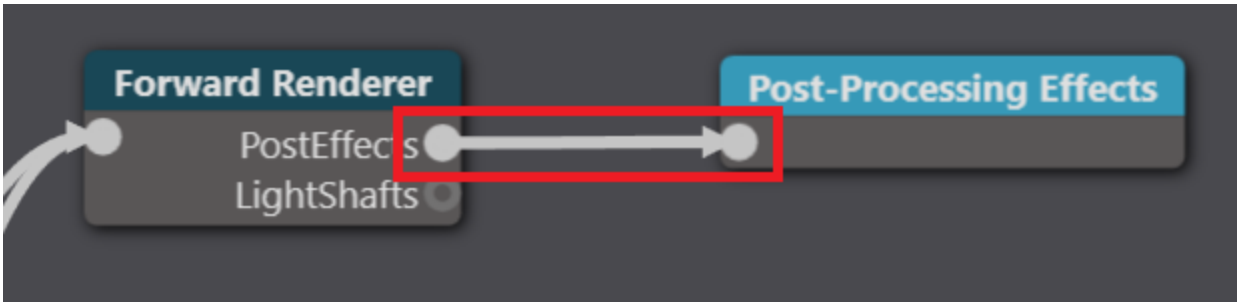
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



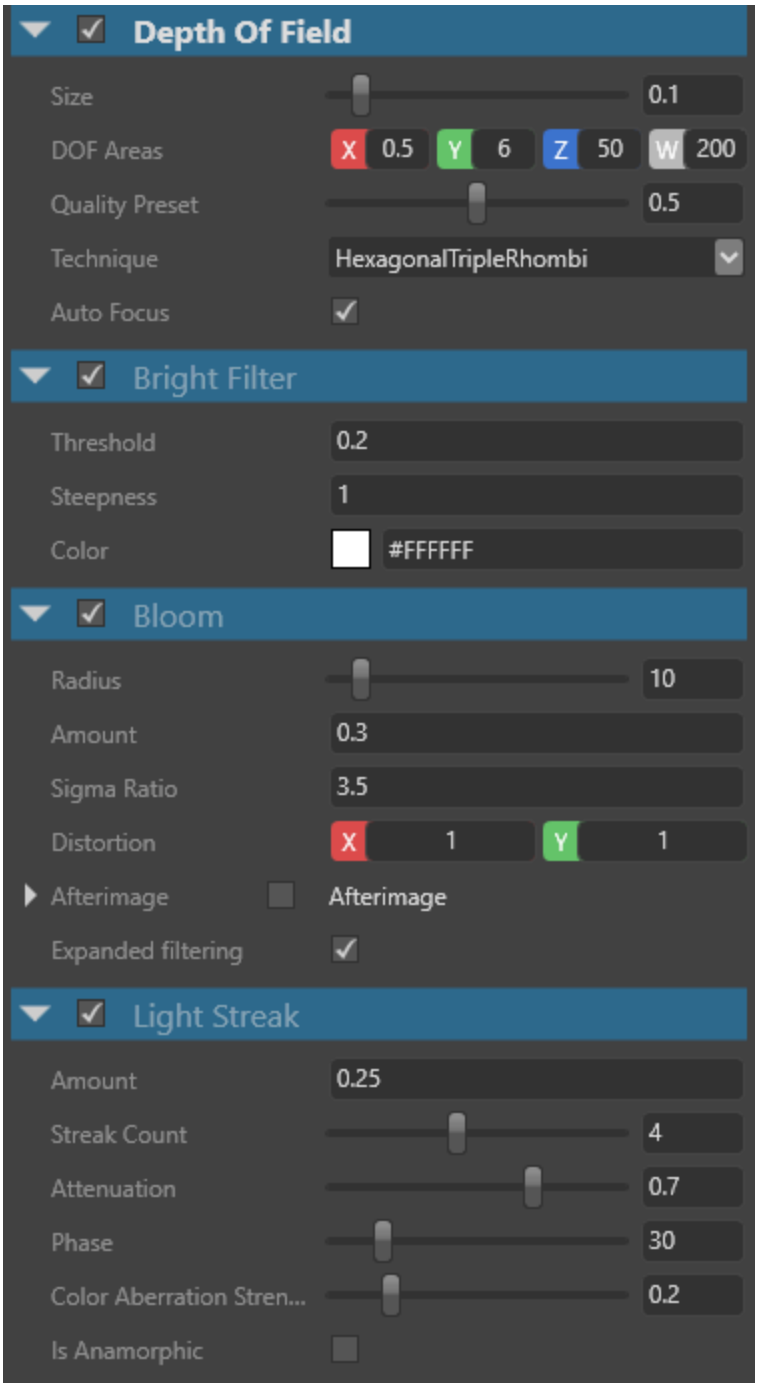
The graphics compositor editor opens.



2. Select the **Post-processing effects** node.

> **TIP**
>
> If there's no post-process effects node, right-click and select **Create > post-processing effects** to create one. On the new **forward renderer** node, on the **PostEffects** slot, click and drag a link to the **post-processing effects** node.
>
> 
>

3. In the **Property Grid** (on the right by default), enable the post effects you want to use and configure their properties.



For details about each post effect and its properties, see the pages below.

In this section

- Anti-aliasing
- Fog
- Outline
- Ambient occlusion
- Bloom
- Bright filter
- Color transforms
  - Film grain

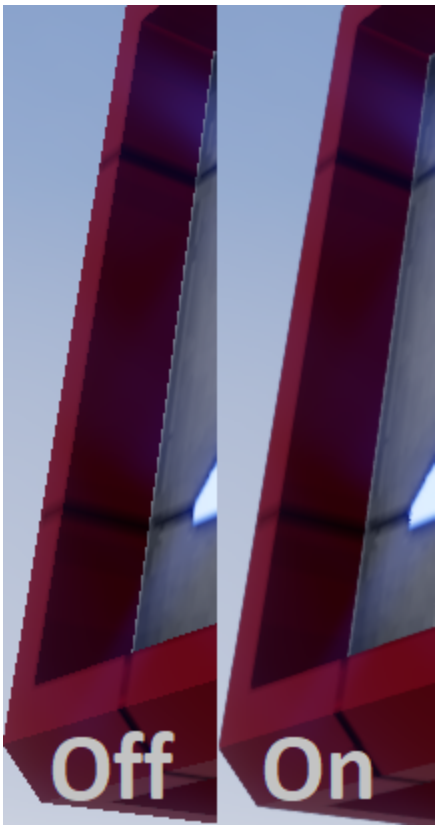
    - Gamma correction
    - ToneMap
    - Vignetting
    - Custom color transforms
- Depth of field
- Lens flare
- Light streaks
- Local reflections

See also

- Graphics compositor

Anti-aliasing

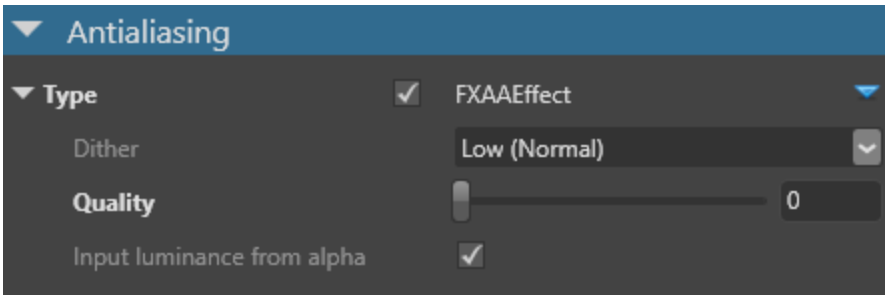
Anti-aliasing smooths jagged edges. For post-processing, Stride uses fast-approximate anti-aliasing (FXAA), a single-pass screen-space technique with low performance impact.



> **NOTE**
> Currently, the anti-aliasing post-effect doesn't work correctly on Android devices.

Stride also includes **MSAA** (multisample anti-aliasing), but this isn't a post effect. You can enable MSAA in the **forward renderer** properties.

Properties



Property	Description
Dither	The amount of dither. Less dither produces better results, but is slower.
Quality	The quality of the effect. This directly affects performance.
Input luminance from alpha	Retrieve the luminance from the alpha channel of the input color. This is slower but more accurate. If disabled, the effect uses the green component of the input color as an approximation for the luminance.

See also

- Ambient occlusion
- Fog
- Outline
- Bloom
- Bright filter
- Color transforms
- Depth of field
- Lens flare
- Light streaks

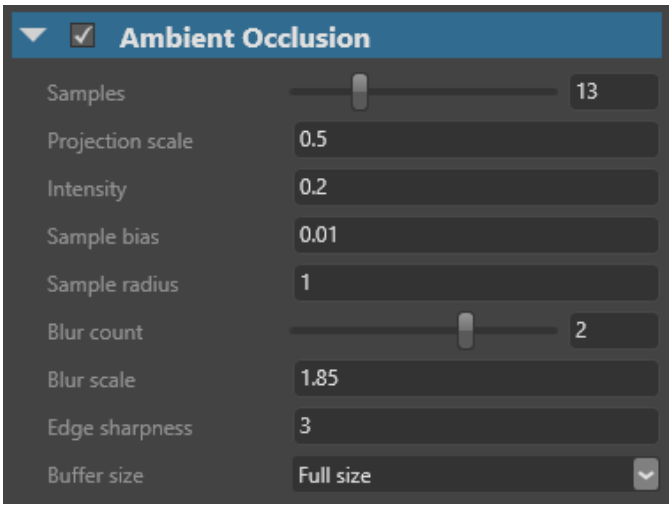
Ambient occlusion

Intermediate Artist

> **NOTE**
> As with other depth-aware post effects, enabling ambient occlusion nullifies MSAA (multisample anti-aliasing).

Ambient occlusion darkens areas where light is occluded by opaque objects, such as corners and crevices. You can use it to add subtle realism to scenes.

Properties



Property	Function
Samples	The number of pixels sampled to determine how occluded a point is. Higher values reduce noise, but affect performance. Use with **Blur count** to find a balance between results and performance.
Projection scale	Scales the sample radius. In most cases, `1` (no scaling) produces the most accurate result.
Intensity	The strength of the darkening effect in occluded areas
Sample bias	The angle at which Stride considers an area of geometry an occluder. At high values, only narrow joins and crevices are considered occluders.
Sample radius	Use with **projection scale** to control the radius of the occlusion effect
Blur count	The number of times the ambient occlusion image is blurred. Higher numbers reduce noise, but can produce artifacts.
Blur scale	The blur radius in pixels
Edge sharpness	How much the blur respects the depth differences of occluded areas. Lower numbers create more blur, but might blur unwanted areas (ie beyond occluded areas).
Buffer size	The resolution the ambient occlusion is calculated at. The result is upscaled to the game resolution. Larger sizes produce better results but use more memory and affect performance.

See also

- Anti-aliasing
- Fog
- Outline
- Bloom
- Bright filter
- Color transforms
- Depth of field

- Lens flare
- Light streaks
- Local reflections

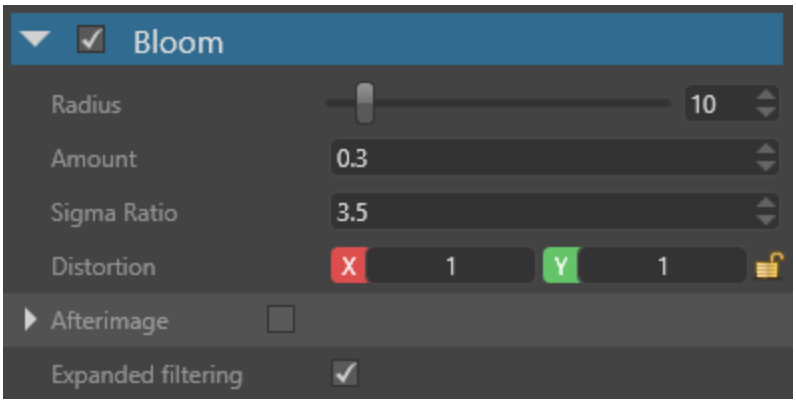
Bloom

Intermediate Artist

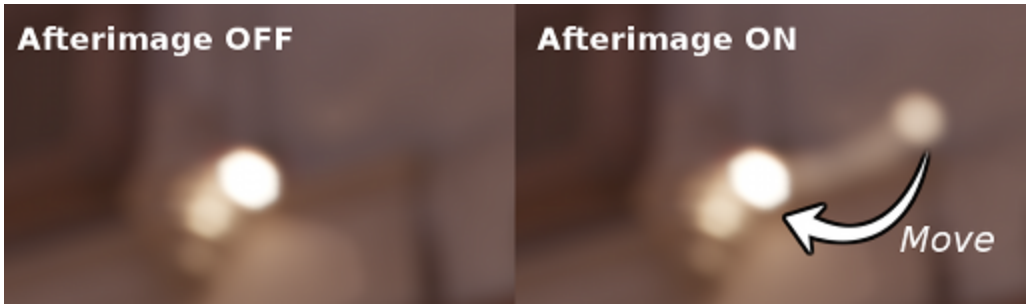
The **bloom** effect takes the brightest areas of an image, extends them, and bleeds them into the surrounding areas to simulate bright light overwhelming the camera.

It uses the result of the bright filter effect as input.

Properties



Property	Description
Radius	Radius of the bloom. Note that high values can affect performance.
Amount	Amount/strength of bloom.
Sigma Ratio	This affects the fall-off of the bloom. It's the standard deviation (sigma) used in the Gaussian blur formula when calculating the kernel of the bloom.
Distortion	Stretches the image horizontally or vertically.
Afterimage	Simulates afterimage (Wikipedia) — the effect of bright spots "burning" into the retina the longer you look at them, before fading away.

Property	Description
	
Fade Out Speed	The factor by which the afterimage (if enabled) decreases at each frame (1 means infinite persistence, while 0 means no persistence at all)
Sensitivity	How sensitive the afterimage (if enabled) is to light. The higher this value is, the faster the effect is created when the camera focuses on a light.
Expanded filtering	Reverses FXAA and bloom, and uses a richer convolution kernel during blurring. This helps reduce temporal shimmering.

See also

- Anti-aliasing
- Fog
- Outline
- Ambient occlusion
- Bright filter
- Color transforms
- Depth of field
- Lens flare
- Light streaks

Bright filter

Intermediate Artist

The **bright filter** extracts the brightest areas of an image. The bright filter itself isn't a post effect, but its result is used later by other effects such as bloom, light streaks, and lens flare.

Properties

Property	Description
Threshold	The threshold used to determine if a color passes or fails the bright filter.
Steepness	Increasing the steepness has a similar effect to increasing the threshold, but causes less aliasing risk. However, the effect is more washed out. For better temporal stability, if your scene has HDR spreads, setting the steepness to a value somewhere in the middle of the expected maximum allows for smooth filtering of bright spots. For sharpness, we recommend you keep a threshold.
Color	The result of the bright filter is modulated by this color value, then affects the color of other post effects. If set to white, the color isn't modified.

In this section

- Anti-aliasing
- Fog
- Outline
- Ambient occlusion
- Bloom
- Color transforms
- Depth of field
- Lens flare
- Light streaks

Color transforms

Intermediate Artist Programmer

Color transforms are special effects designed to be combined in a chain at runtime. You can define a series of color transforms to apply to an image. Each transform uses the previous transform's output as its own input. At runtime, the series of transforms is squashed into one shader and rendered in a single draw call for maximum performance.

You can also write your own custom color transforms to create unique effects.

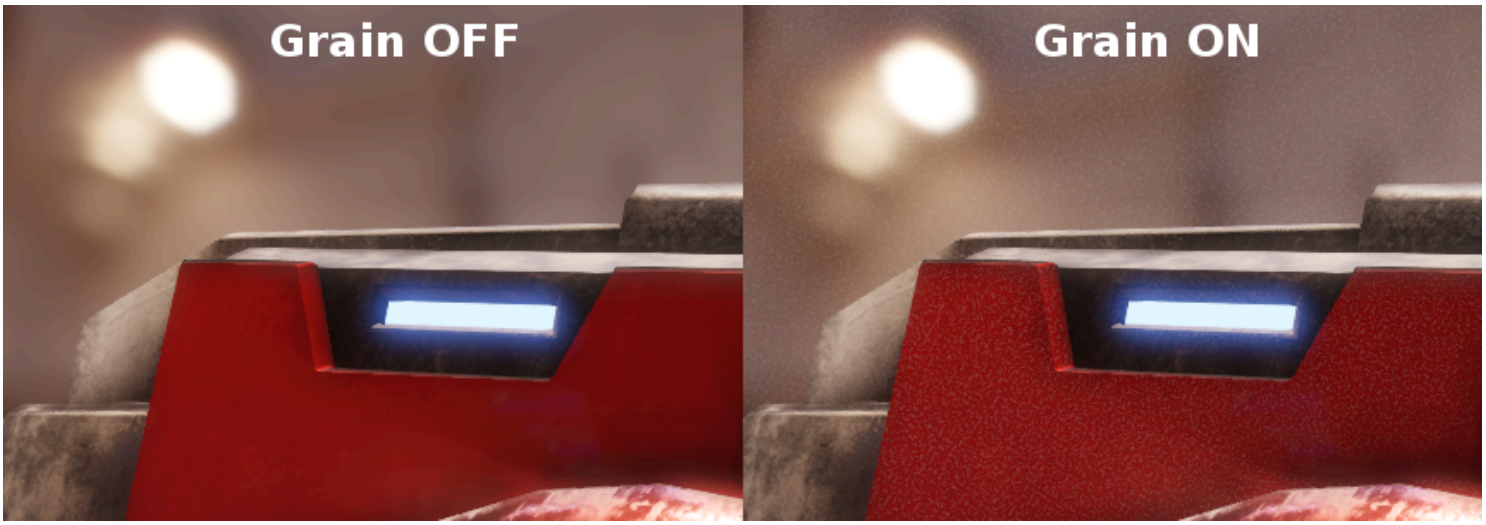
In this section

- Film grain
- Gamma correction
- ToneMap

Film grain

Beginner Artist Programmer

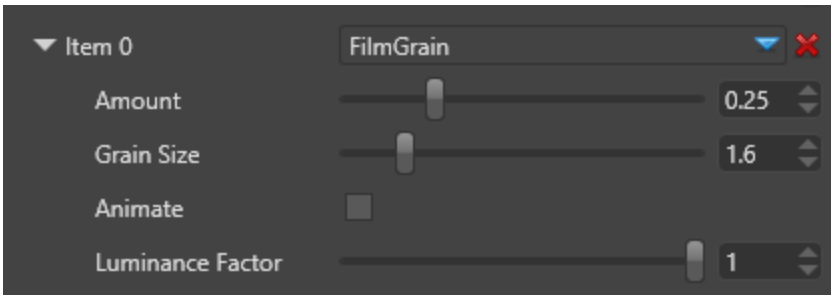
The **film grain** adds noise at each frame to simulate the grain of films used in real cameras.



The pattern is procedurally generated and changes at each frame.

To simulate real film grain, the noise should be more visible in areas of medium light intensity, and less visible in bright or dark areas.

The pattern locally modifies the luminance of the pixels affected.



Properties

Property	Description
Amount	Amount/strength of the effect
Grain Size	Size of the grain
Animate	When enabled, the procedural pattern changes at each frame
Luminance Factor	How strongly the original pixel luminance is affected by the grain pattern

See also

- Gamma correction
- ToneMap
- Vignetting

Gamma correction

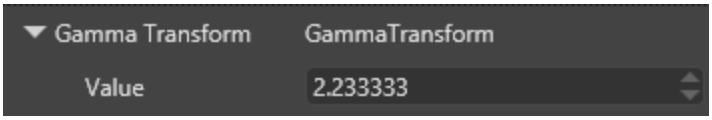
Beginner Artist Programmer

All post effect calculations are made in a linear space (ie RGB space). This means doubling the color value of a pixel doubles the light it emits. This guarantees correct lighting calculations.

However, real-world computer monitors don't behave this way: for dark color values they tend to emit much less light than they should. For this reason, after our other post effects have been applied, we apply **gamma correction** to transform our image from a linear space to a sRGB space (or gamma space).

A buffer in the sRGB space displays correctly on a monitor or a TV screen.

Non-gamma-corrected images have dark areas appear darker than they're supposed to.



Properties

Property	Description
Value	Gamma value. A typical value is around 2.2.

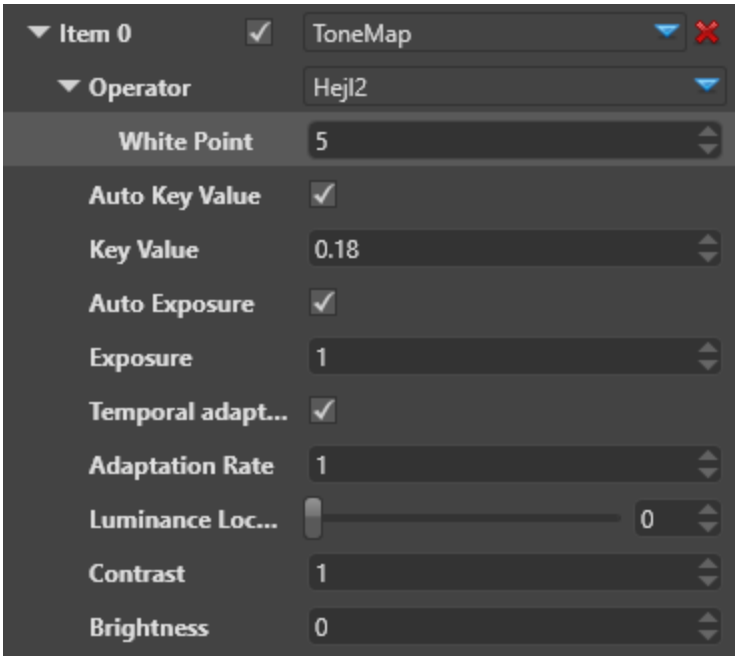
See also

- Gamma correction (Wikipedia)
- Film grain
- ToneMap
- Vignetting
- Custom color transforms

ToneMap

Tone-mapping takes an HDR buffer as input, and remaps its color to a [0, 255] range so we can display it on a screen.

There are many ways to remap colors from an HDR space to an LDR, depending on the formula you choose.



Stride supports several tone-mapping operators out of the box:

- Reinhard (the classic operator)
- Exponential
- Logarithmic
- Drago
- Hejl-Dawson
- Mike-Day
- U2-Filmic

See also

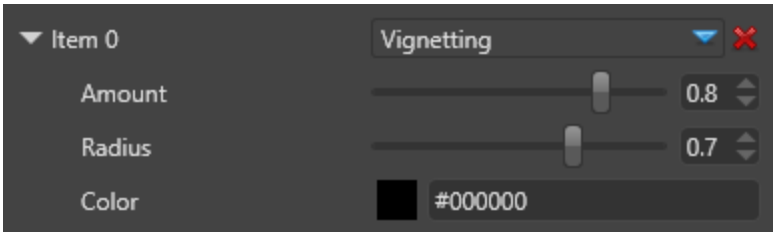
- Film grain
- Gamma correction
- Vignetting

Vignetting

Beginner Artist Programmer

The **vignetting** effect darkens the angles or the borders of an image.

This is an artifact appears with real-world cameras. You can use it in your game to change the mood of the scene or focus on the center of the image.



Properties

Property	Description
Amount	Amount/strength of the effect
Radius	Radius of the vignette from the center of the screen. A low value thickens the makes border and narrows the central space
Color	The vignette color

See also

- Film grain
- Gamma correction
- ToneMap
- Custom color transforms

Custom color transforms

Advanced Programmer

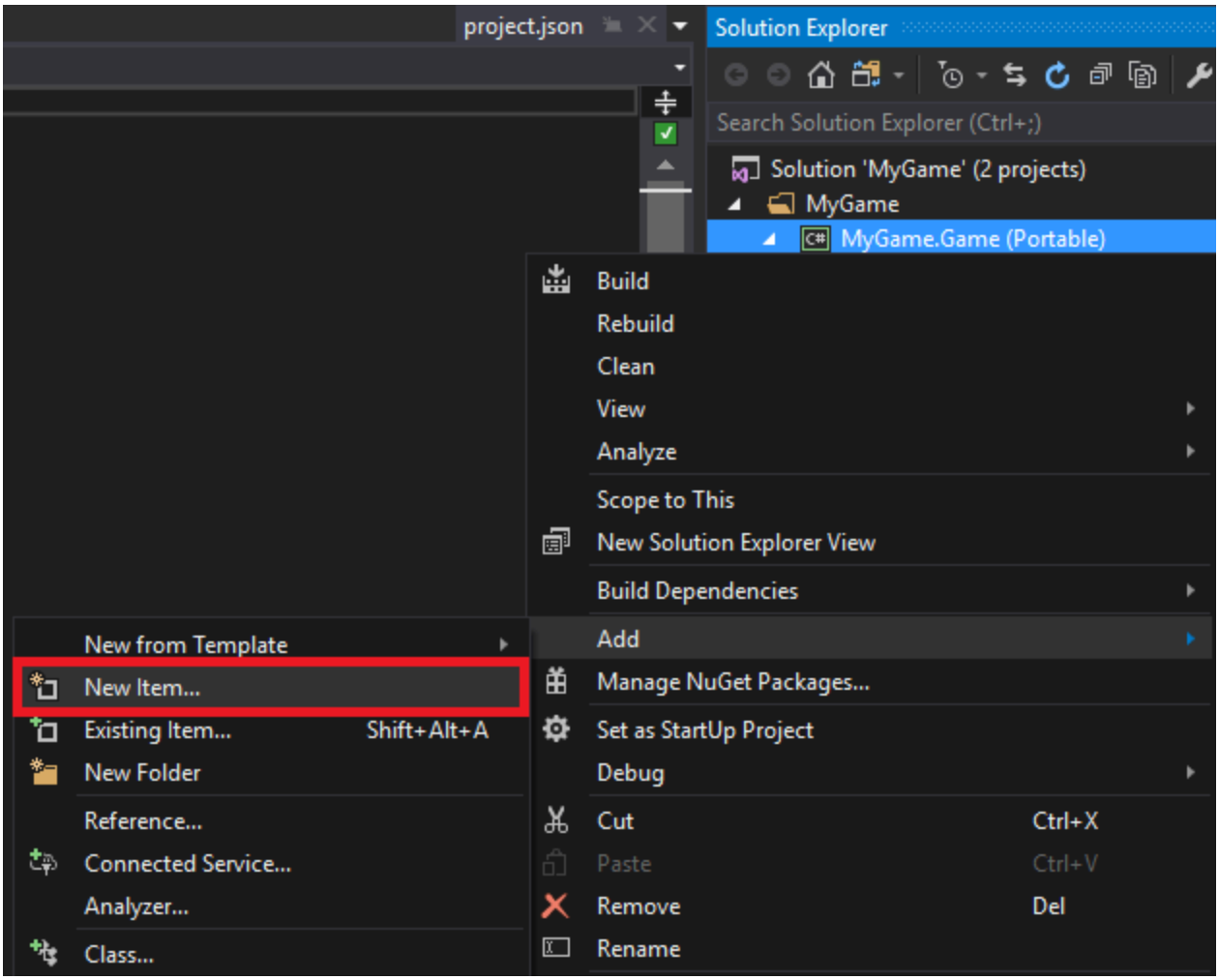
You can write your own **custom color transform** effects. For example, you can create:

- water droplets on the camera
- screen transitions (such as fade-ins and fade-outs)
- effects simulating pain or intoxication (eg by applying tints or other effects)
- object outlines

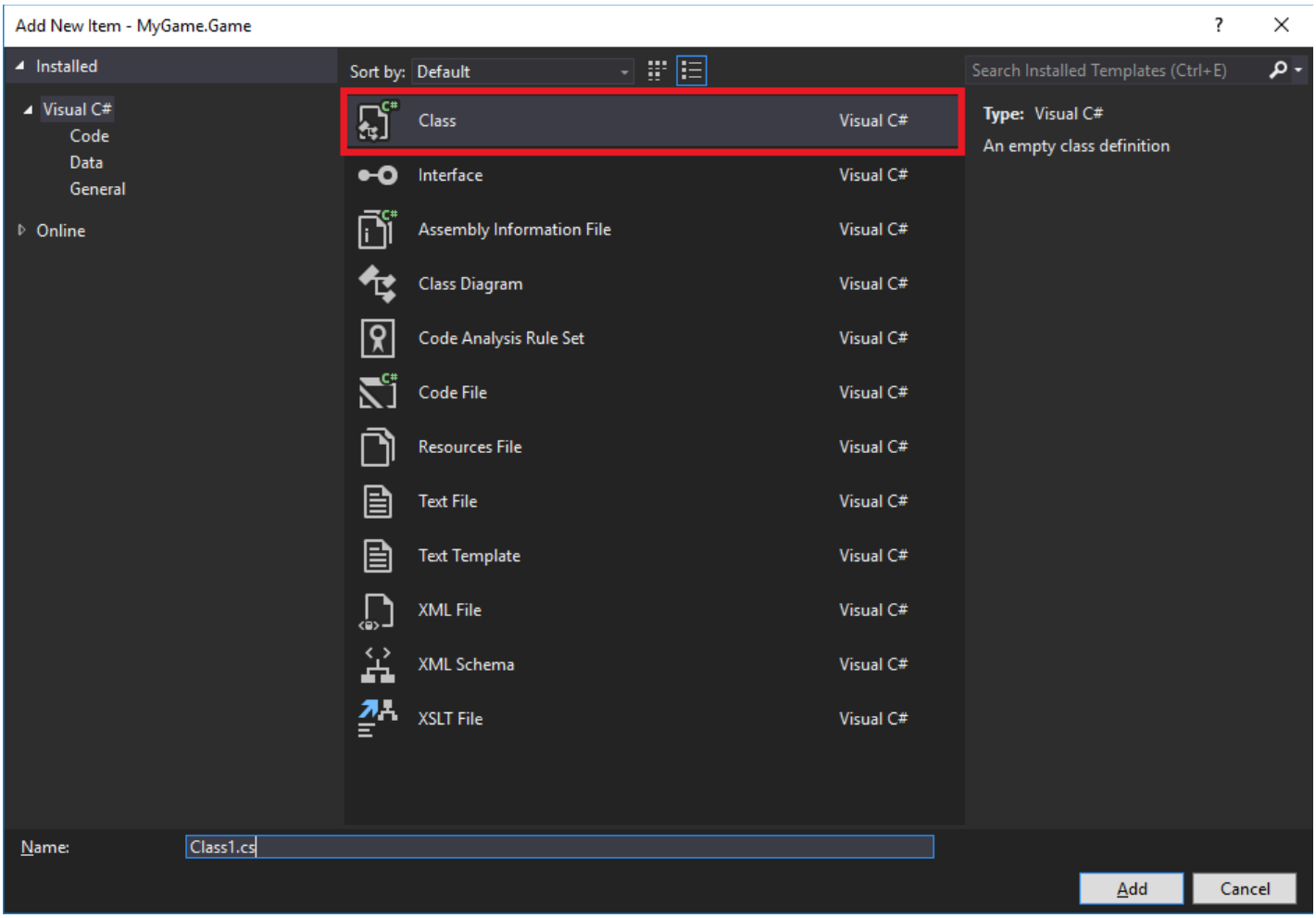
To create a custom color transform, you need to write two files: an effect shader (containing the effect itself), and a C# class (to make the effect accessible in Game Studio).

1. Create a shader

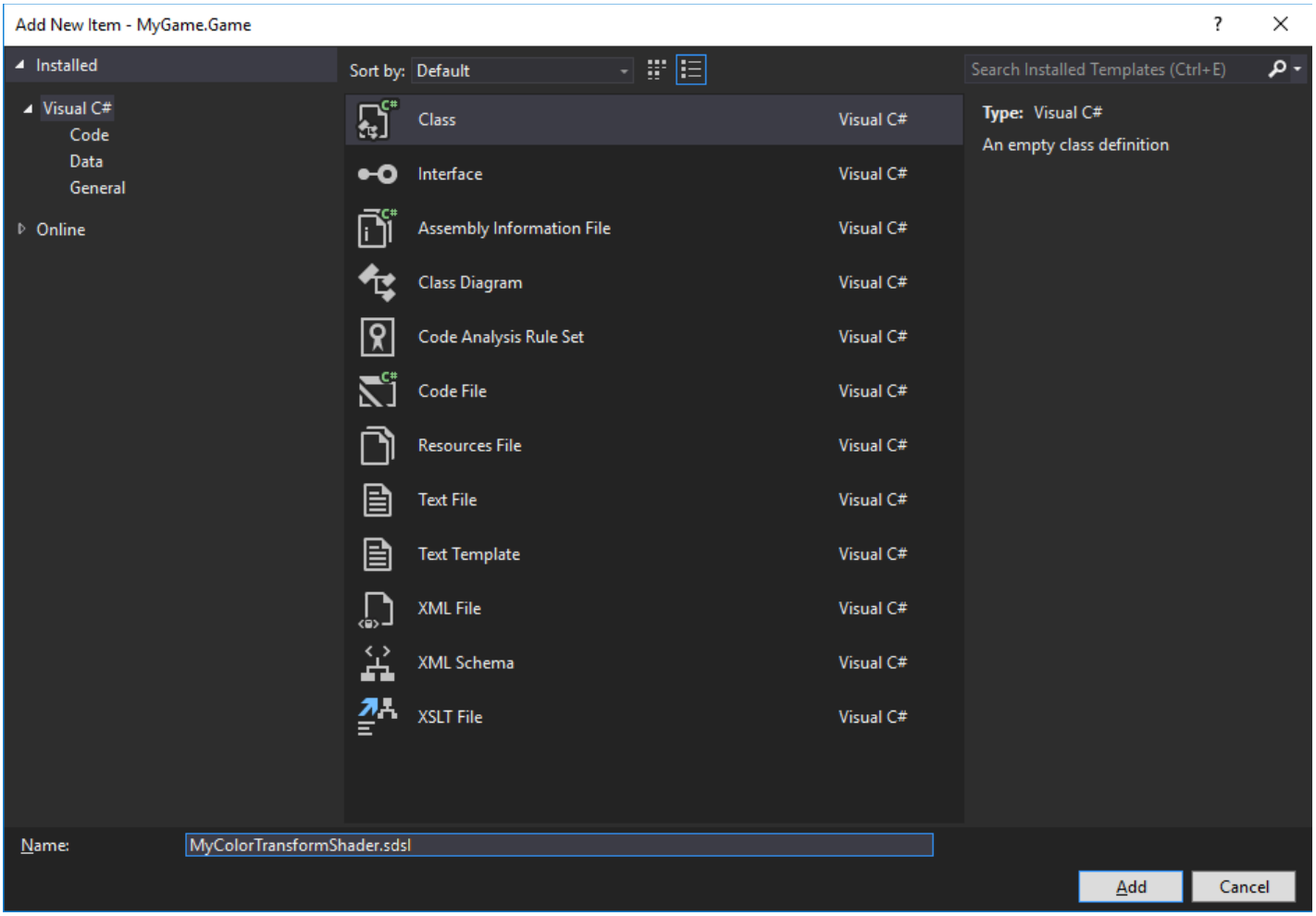
1. Make sure you have the Stride Visual Studio extension installed. This is necessary to convert the shader files from SDSL (Stride shading language) to `.cs` files.
2. In Game Studio, in the toolbar, click (**Open in IDE**) to open your project in Visual Studio.
3. In the Visual Studio **Solution Explorer**, right-click the project (eg *MyGame.Game*) and select **New item**.



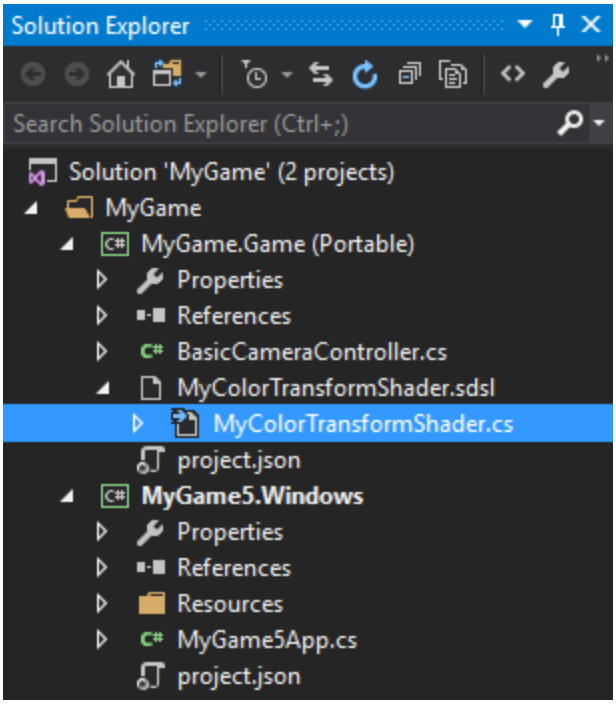
4. Select **Class**.



5. In the **Name** field, specify a name with the extension **.sdsl** (eg *MyColorTransformShader.sdsl*), and click **Add**.



The Stride Visual Studio extension automatically generates a `.cs` file from the `.sdsl` file. The Solution Explorer lists it as a child of the `.sdsl` file.



6. Open the `.sdsl` file, remove the existing lines, and write your shader.

Shaders are written in Stride Shading Language (SDSL), which is based on HLSL. For more information, see Shading language.

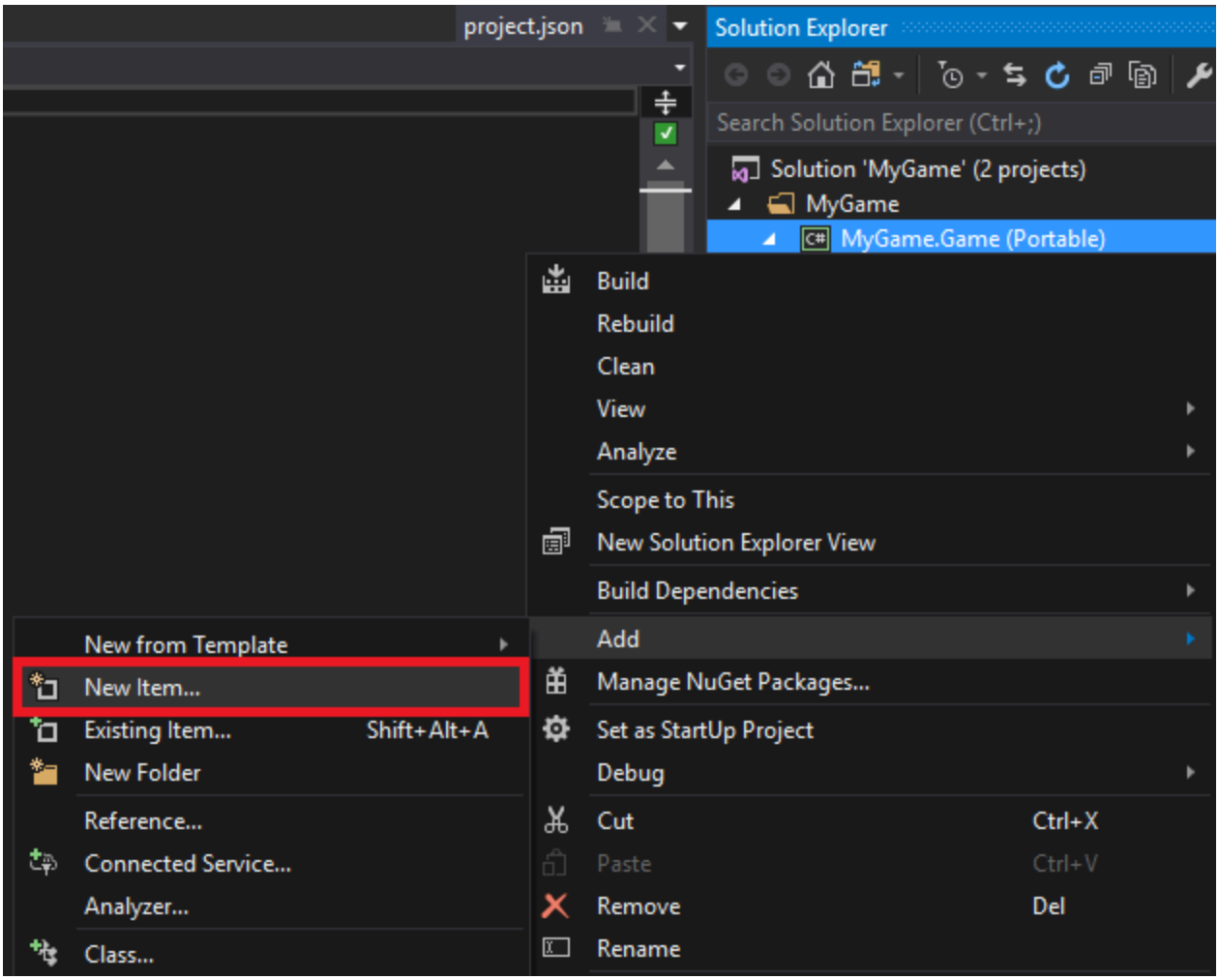
For example, the shader below multiplies the image color by the `MyColor` parameter:

```
shader MyColorTransformShader : ColorTransformShader
{
    [Color]
    float4 MyColor;

    override float4 Compute(float4 color)
    {
        return color * MyColor;
    }
};
```

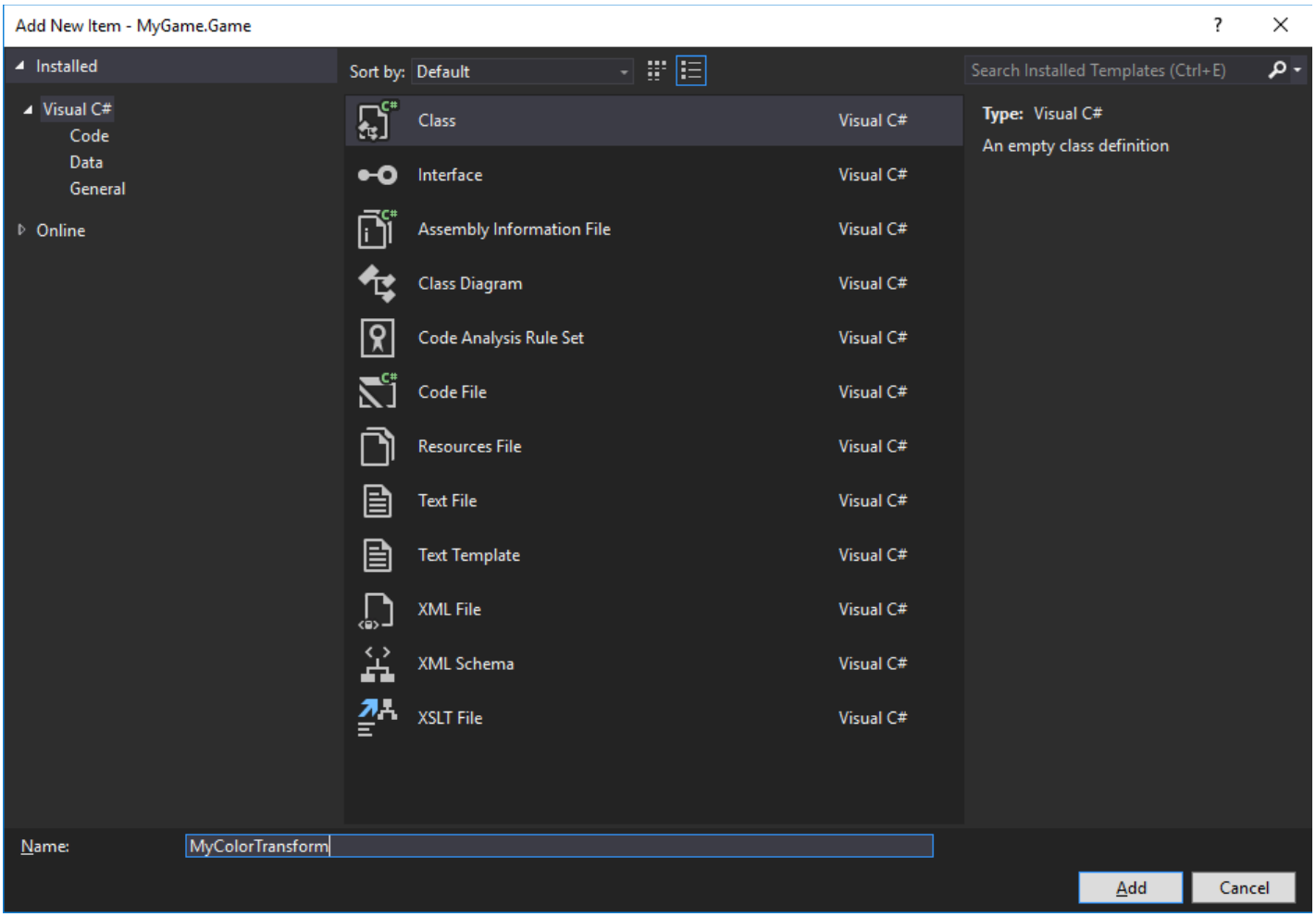
> **NOTE**
> Make sure the shader name in the file (eg `MyColorTransformShader` in the code above) is the same as the filename (eg *MyColorTransformShader.sdsl*).

2. Create a C# class

1. In the Visual Studio **Solution Explorer**, right-click the project (eg *MyGame.Game*) and select **Add > New item**.



2. Select **Class**, specify a **name** (eg *MyColorTransform.cs*), and click **Add**.



Open the file and write the class.

For example, the code below creates the class `MyColorTransform`, which uses the shader and supplies a value for the color `MyColor` (defined in the shader).

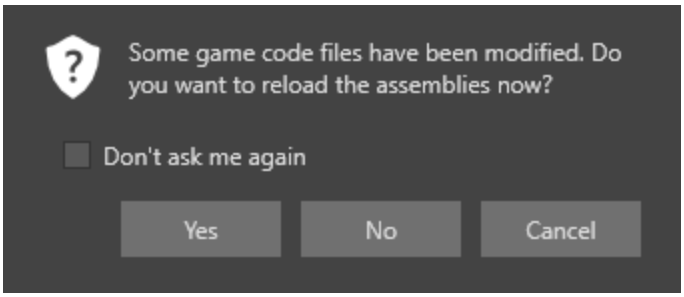
```
using Stride.Core;
using Stride.Core.Mathematics;
using Stride.Rendering;
using Stride.Rendering.Images;

namespace MyGame
{
    [DataContract("MyColorTransform")]
    public class MyColorTransform : ColorTransform
    {
        /// <inheritdoc />
        public MyColorTransform()
            : base("MyColorTransformShader")
        {
        }
```

```
        public Color4 MyColor { get; set; }

        public override void UpdateParameters(ColorTransformContext context)
        {
            Parameters.Set(MyColorTransformShaderKeys.MyColor, MyColor);

            // Copy parameters to parent
            base.UpdateParameters(context);
        }
    }
}
```

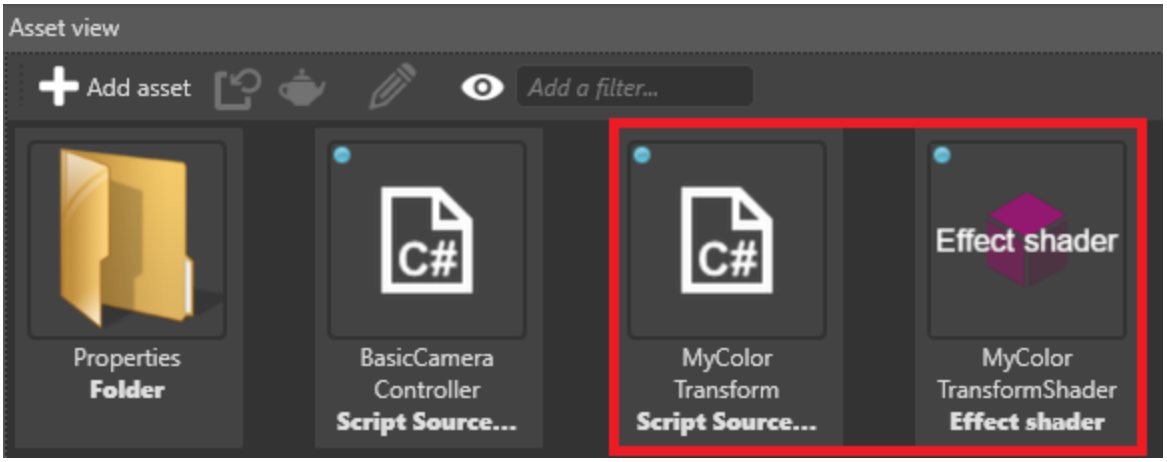
> **NOTE**
>
> Make sure the class name in the file (eg `MyColorTransform` in the class above) is the same as the filename (eg *MyColorTransform.cs*).

3. Save all the files in the solution (**File > Save All**).

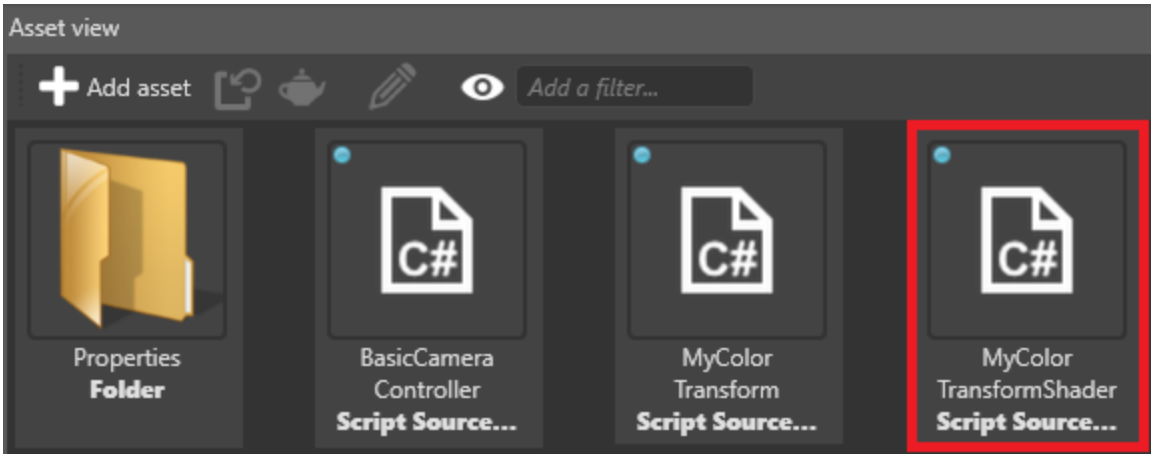
4. In Game Studio, reload the assemblies.



The **Asset View** lists the class and effect shader in the same directory as your scripts (eg **MyGame.Game**).



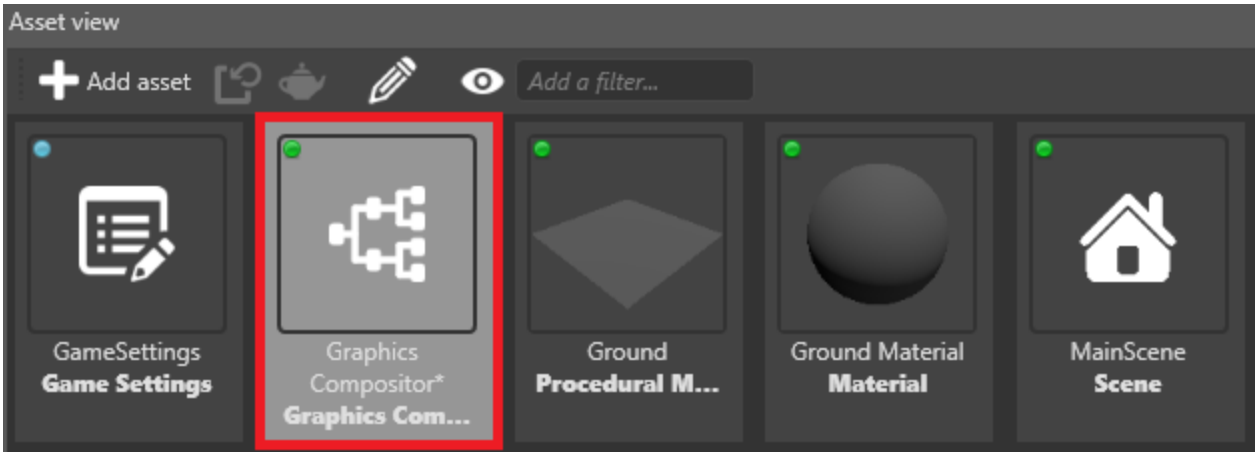
> **NOTE**
>
> In some situations, Game Studio incorrectly detects the shader as a script, as in the screenshot below:



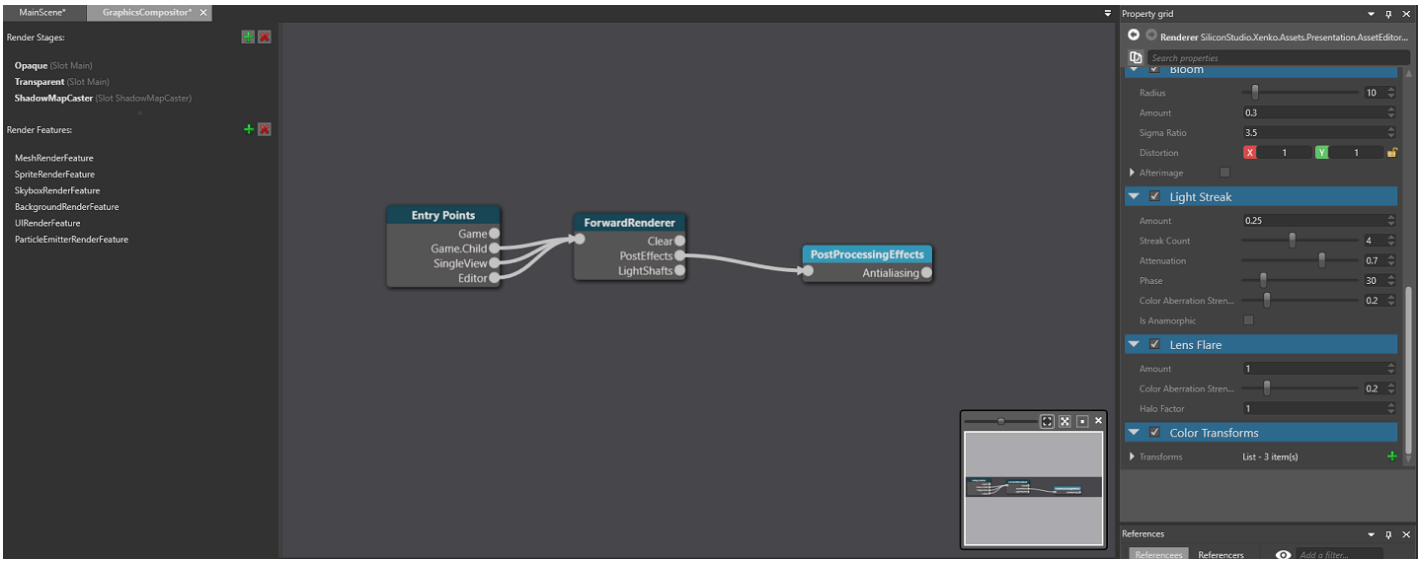
If this happens, restart Game Studio (**File > Reload project**).

3. Use a custom color transform

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

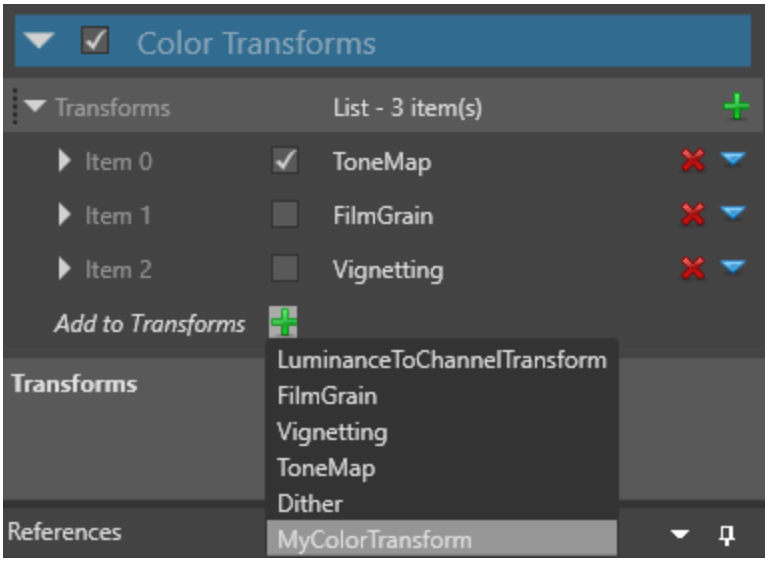


The **graphics compositor editor** opens.

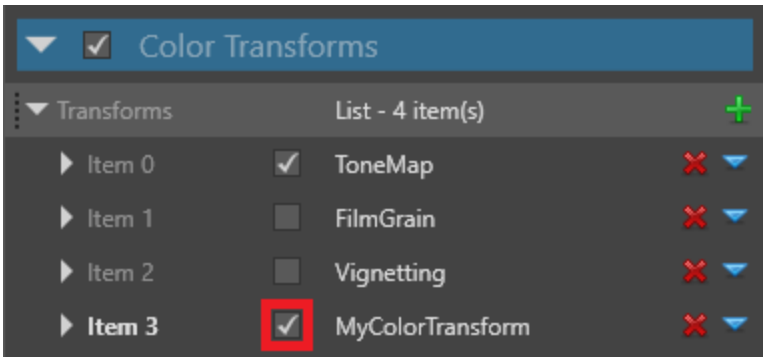


2. Select the **Post-processing effects** node.

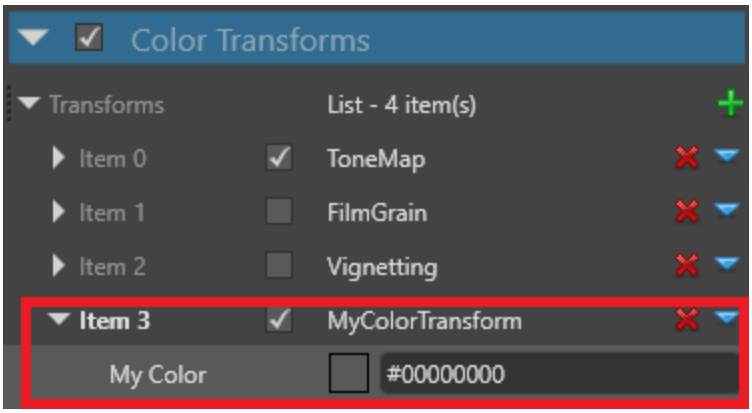
3. In the **Property Grid**, under **Color transforms**, click (**Change**) and select your color transform (eg **MyColorTransform**).



- To enable and disable the effect, use the check box next to the item.



- To edit the public properties you specified in the class, expand the item.



When you adjust the properties, Game Studio updates the effect automatically.

> **WARNING**
>
> Unfortunately, this part of Game Studio has a memory leak problem. Every time you change a value in the graphics compositor, it uses 60MB of memory. To prevent Game Studio using too much memory, we recommend you restart it after you change a property a few times. This is a known issue.

See also

- Shading language
- Custom shaders
- Graphics compositor
- Post effects
- Color transforms
- Stride Visual Studio extension

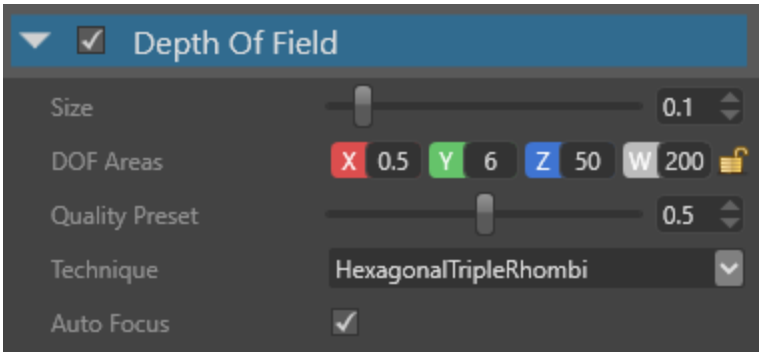
Depth of field

Intermediate Artist

By default, rendering produces a very sharp image, which can look artificial. **Depth of field** effects simulate the behavior of a real camera lens focusing an object, leaving background and foreground objects out of focus.

To create the effect, Stride creates several versions of the original image with different intensities of blur, and interpolates between them. The more layers used, the better the quality, but at performance cost.

Properties



Property	Description
Size	Size of the bokeh (Wikipedia), expressed as a factor of the image width so it's resolution-independent. The bigger the size, the worse the performance
DOF Areas	Areas of the depth of field. There are three main zones defined by their distance from the camera: near out-of-focus area (from X to Y), in-focus area (from Y to Z), and far out-of-focus area (from Z to W)
Technique	The technique affects both the performance and the shape of the bokeh. **Circular Gaussian** is fast but unrealistic. **Hexagonal Triple Rhombi** is heavier than Gaussian.

Property	Description
	Hexagonal McIntosh is the heaviest.
Auto Focus	Automatically updates the DOF areas so the camera focuses on the object at the center of the screen

See also

- Anti-aliasing
- Fog
- Outline
- Ambient occlusion
- Bloom
- Bright filter
- Color transforms
- Lens flare
- Light streaks

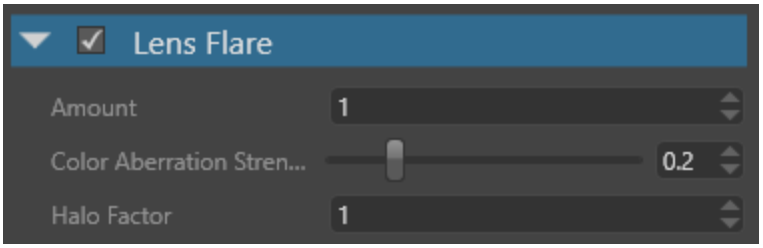
Lens flare

Intermediate Artist

The **lens flare** effect simulates the artifacts produced by the internal reflection or scattering of the light within a real-world lens.

The artifacts are generally aligned along the line defined by the original bright spot and the center of the screen. The most noticeable artifact is often exactly symmetrical to the real spot light with respect to the center of the screen.

Properties



Property	Description
Amount	Strength of the light streak
Color Aberration Strength	Strength of the color aberration artifacts
Halo Factor	Strength of the main artifact

See also

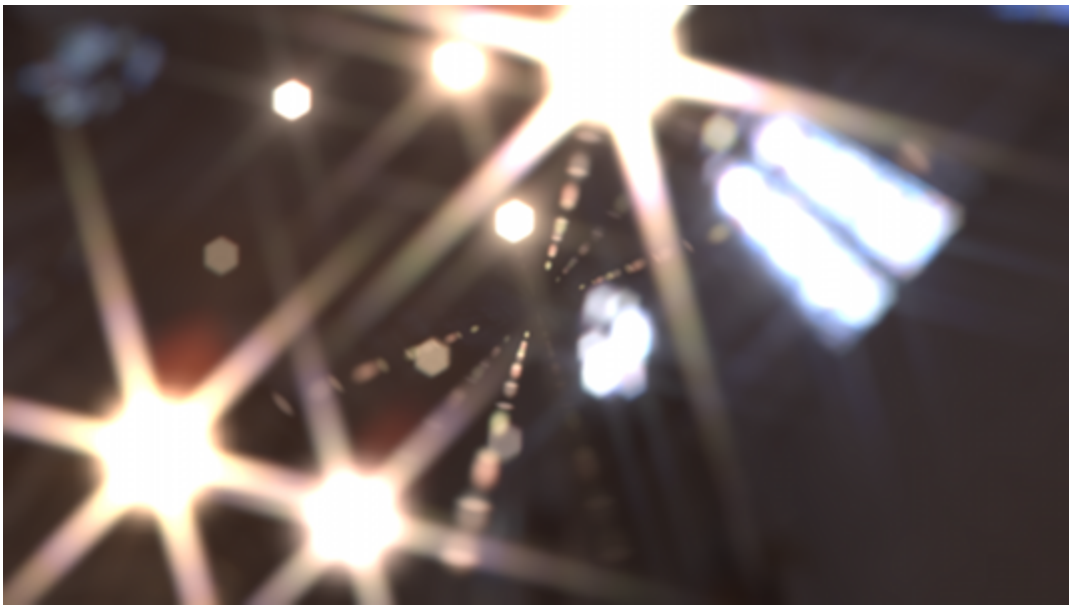
- Anti-aliasing

- Fog
- Outline
- Ambient occlusion
- Bloom
- Bright filter
- Color transforms
- Depth of field
- Light streaks

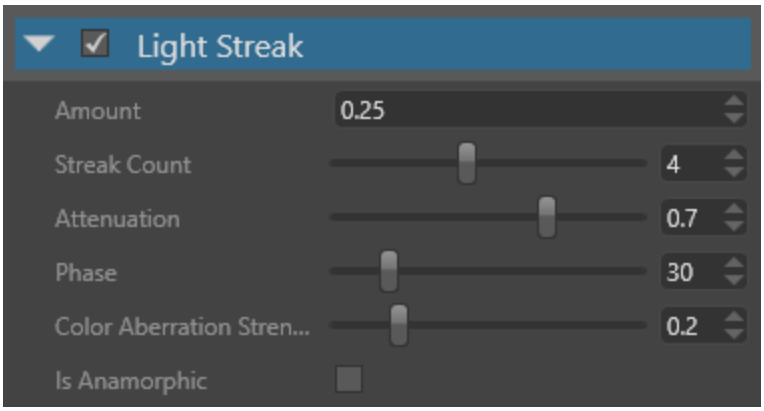
Light streaks

Intermediate Artist

Similar to the bloom effect, the **light streak** effect uses the result of the bright filter to make the bright areas bleed along a direction. It creates star-pattern beams from the light point.



Properties



Property	Description
Amount	Strength of the light streak
Streak Count	Number of beams emitted by a bright point. The more streaks, the higher the performance cost.
Attenuation	How fast the light attenuates along a streak (0 for immediate attenuation, 1 for no attenuation)
Phase	Phase (angle) of the star-like pattern

Property	Description
Color Aberration Strength	Strength of the color aberration along the streaks. Notice the streaks involve multiple colors (yellow, purple, green, pink).
Is Anamorphic	Simulates the behavior of anamorphic lenses, widely used in Hollywood productions. The effect above is achieved by using two light streaks with a phase of 0, enabling anamorphic mode, and slightly distorting the bright pass result horizontally.

See also

- Anti-aliasing
- Fog
- Outline
- Ambient occlusion
- Bloom
- Bright filter
- Color transforms
- Depth of field
- Lens flare

Local reflections

Intermediate Artist Programmer

> **WARNING**
> Currently, local reflections aren't compatible with mobile platforms and cause crashes.

> **NOTE**
> As with other depth-aware post effects, enabling local reflections nullifies MSAA (multisample anti-aliasing).

When **local reflections** are enabled, the scene is reflected in glossy materials.

Local reflections dramatically increase the realism of scenes. They're most obvious when they reflect bright spots onto other surfaces. The effect is especially striking in dark scenes, which have high contrast, and in conditions with lots of reflective surfaces and highlights.

Where to use local reflections

Local reflections are a **screenspace effect**, which means they only reflect objects that are already on the screen; they don't reflect objects that are offscreen or obscured by other objects. Put simply, if the camera can't see an object at that moment, then that object isn't reflected.

This means local reflections work well in enclosed areas such as corridors and rooms, but less well in open spaces, where you'd expect more of the world to be reflected. They also work best on bumpy surfaces, which hide imperfections in reflections, and less well on very glossy, mirror-like surfaces. Missing reflections are noticeable in mirrors, for example.

Algorithm

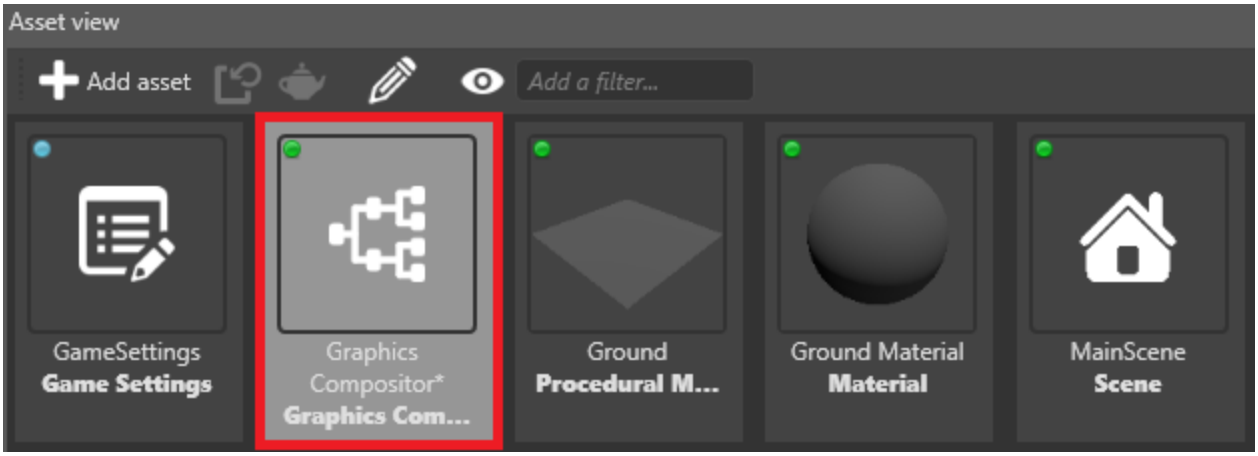
Stride processes local reflections in four passes:

1. The **raycast** pass performs screenspace ray tracing over the depth buffer to find intersections.
2. The **resolve** pass resolves the rays and calculates the reflection color.
3. The **temporal** pass uses the history buffer to blur constantly between the current and previous frames. This reduces noise in the reflection, but produces an animated "jittering" effect that is sometimes noticeable. You can adjust or disable this step to create the effect you want.
4. The **combine** pass mixes the results of the effect with the rendered image.

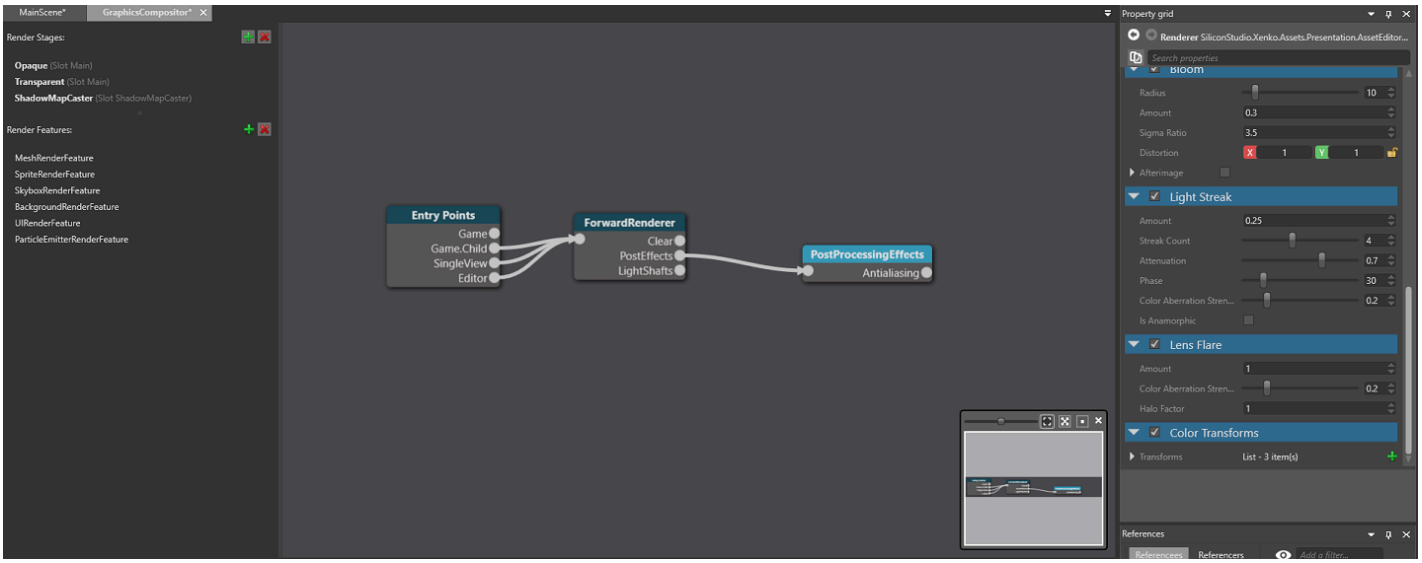
Enable local reflections

To use local reflections, enable the effect in the **graphics compositor**.

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

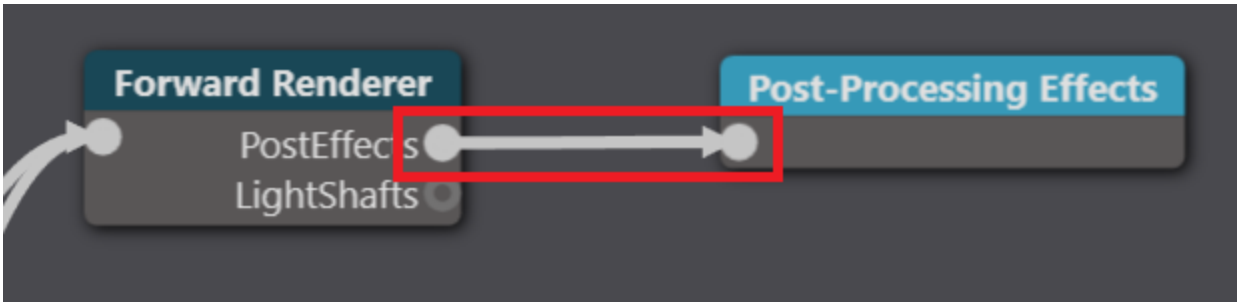


The graphics compositor editor opens.

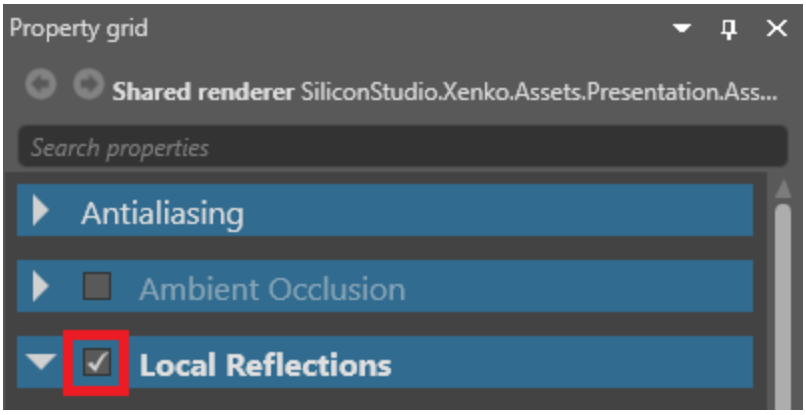


2. Select the **post-processing effects** node.

> **TIP**
>
> If there's no post-process effects node, right-click and select **Create > post-processing effects** to create one. On the new **forward renderer** node, on the **PostEffects** slot, click and drag a link to the **post-processing effects** node.



3. In the **Property Grid** (on the right by default), enable **Local reflections**.



After you enable local reflections, the scene is reflected in glossy materials. You can use the **gloss threshold** (see below) to set how glossy materials should be to reflect the scene.

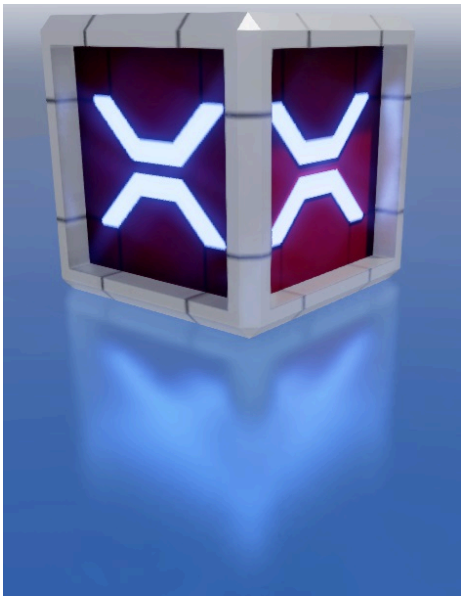
Properties

The local reflections properties affect all reflections in the scene.

Raycast properties

BRDF bias

The reflection spread. Higher values provide finer, more mirror-like reflections. This setting has no effect on performance. The default value is `0.82`.

BRDF: `0.6`	**BRDF:** `0.8`	**BRDF:** `1.0`
		

Depth resolution

Downscales the depth buffer to optimize raycast performance. Full gives better quality, but half improves performance. The default is half.

Gloss threshold

The amount of gloss a material must have to reflect the scene. For example, if this value is set to `0.4`, only materials with a **gloss map** value of `0.4` or above reflect the scene. The default value is `0.55`.

> **NOTE**
>
> If the **Invert** check box is selected in the material **micro surface** properties, the opposite is true. For example, if the reflections gloss value is set to `0.4`, only materials with a **gloss map** value of less than `0.4` reflect the scene.

For more information about gloss, see Materials — geometry attributes.

Max steps

The maximum number of raycast steps allowed per pixel. Higher values produce better results, but worse performance. The default value is `60`.

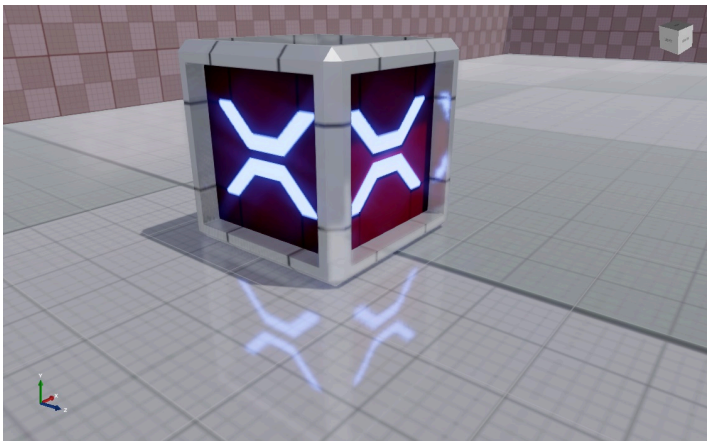
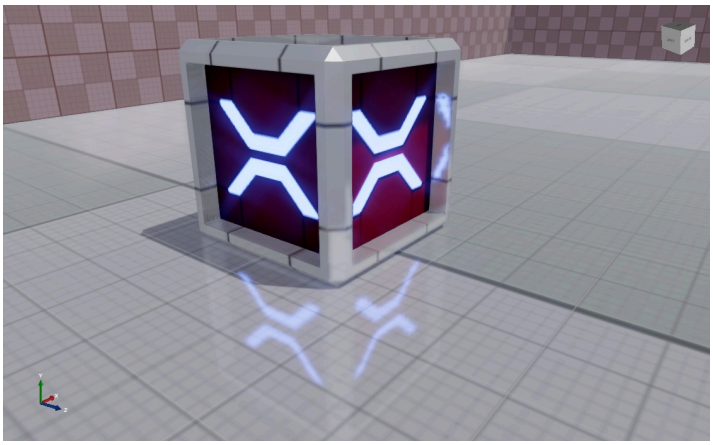
> **NOTE**
>
> This is the most important property for controlling performance.

Resolution

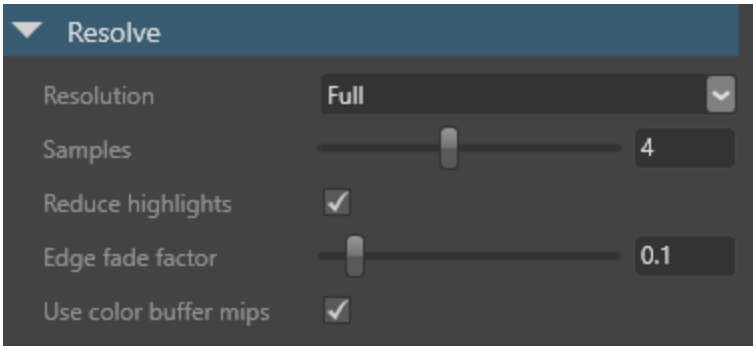
The raycast resolution. There are two options: **full** and **half**. Full gives better quality, but half improves performance. The default value is half.

Ray start bias

The offset of the raycast origin. Lower values produce more correct reflection placement, but produce more artifacts. We recommend values of `0.03` or lower. The default value is `0.01`.

Start bias: `0.01`	**Start bias:** `0.1`
	
Larger gap between reflection and box (more correct)	Narrower gap between reflection and box (less correct)

Resolve properties



Resolution

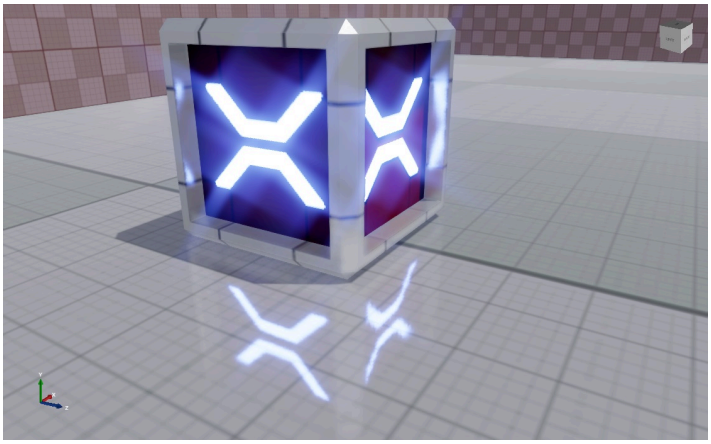
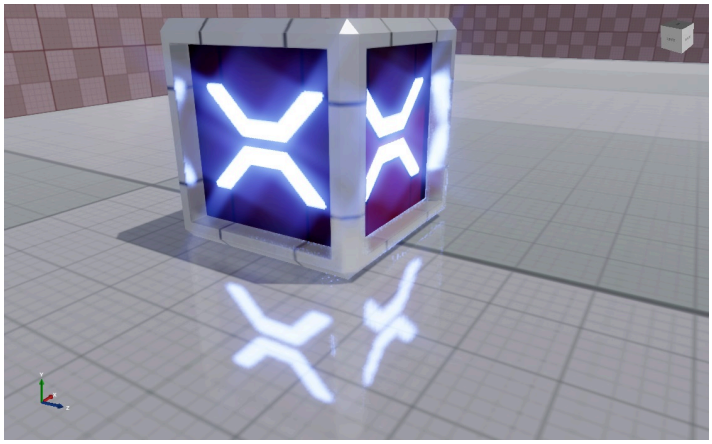
Calculates reflection color using raycast results. There are two options: **full** and **half**. Full gives the best results, but half improves performance. The default value is **full**.

Samples

The number of rays used to resolve the reflection color. Higher values produce less noise, but worse performance. The default value is `4`.

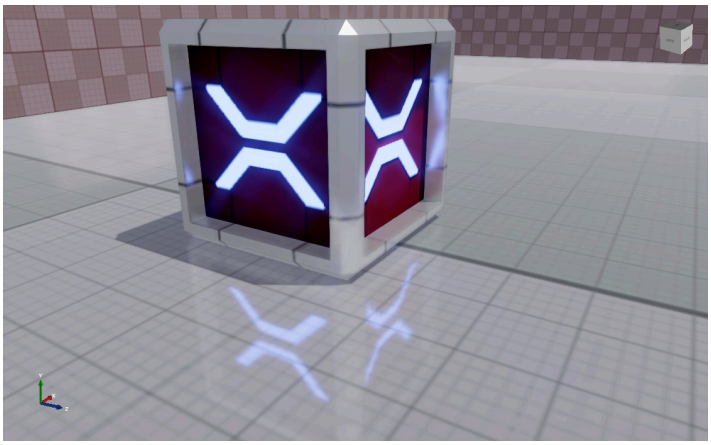
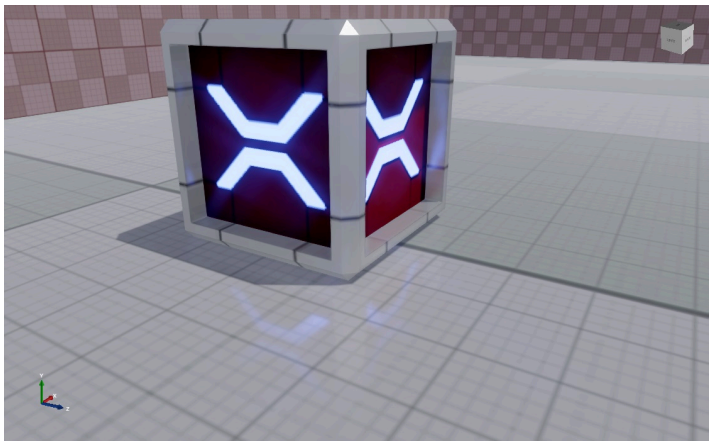
Reduce highlights

Reduces the brightness of particularly bright areas of reflections. This has no effect on performance.

Reduce highlights: on	**Reduce highlights: off**
	

Edge fade factor

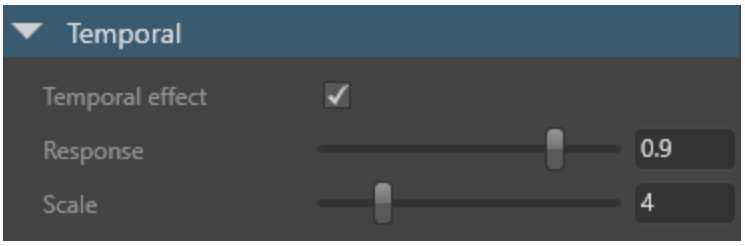
The point at which the far edges of the reflection begin to fade. This has no effect on performance. The default value is `0.1`.

Edge fade factor: `0`	**Edge fade factor:** `0.5`
	

Use color buffer mips

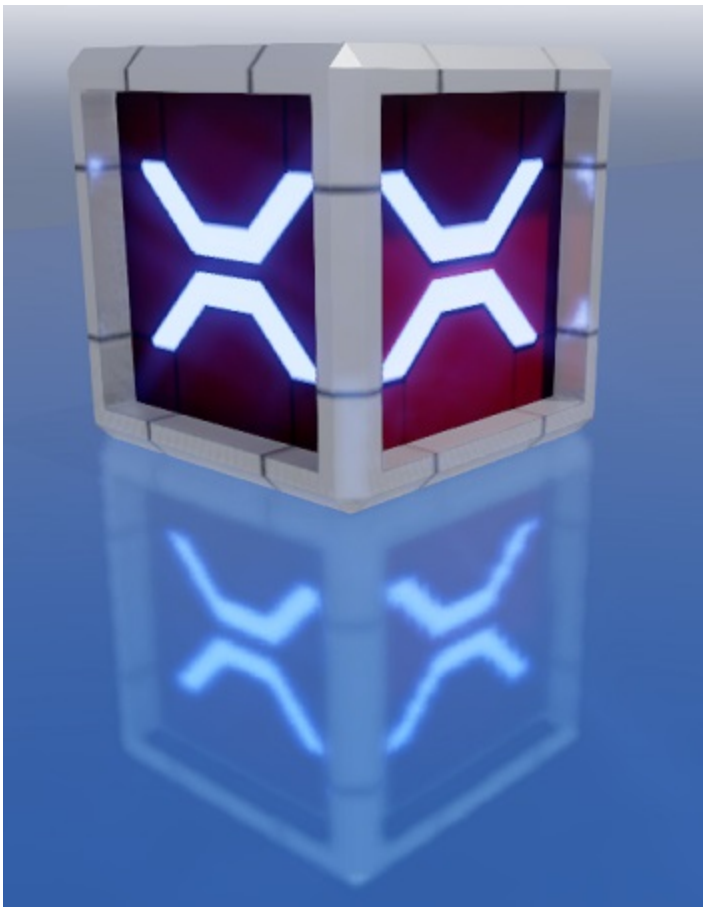
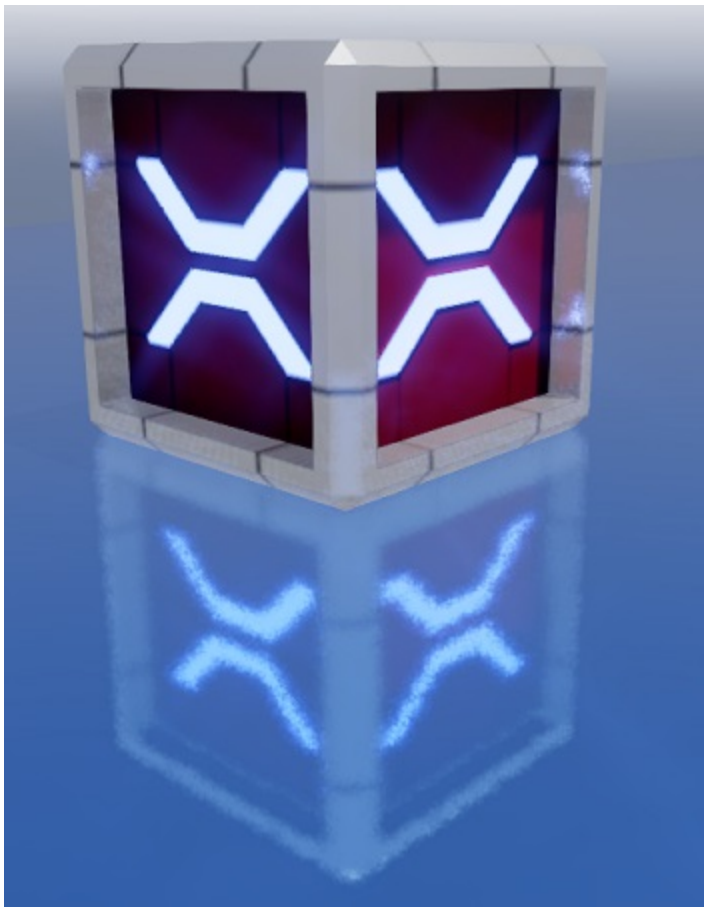
Downscales the input color buffer and uses blurred mipmaps when resolving the reflection color. This produces more realistic results by blurring distant parts of reflections in rough (low-gloss) materials. It also improves performance on most platforms. However, it uses more memory, so you might want to disable it on (for example) mobile platforms.

Temporal properties



Temporal effect

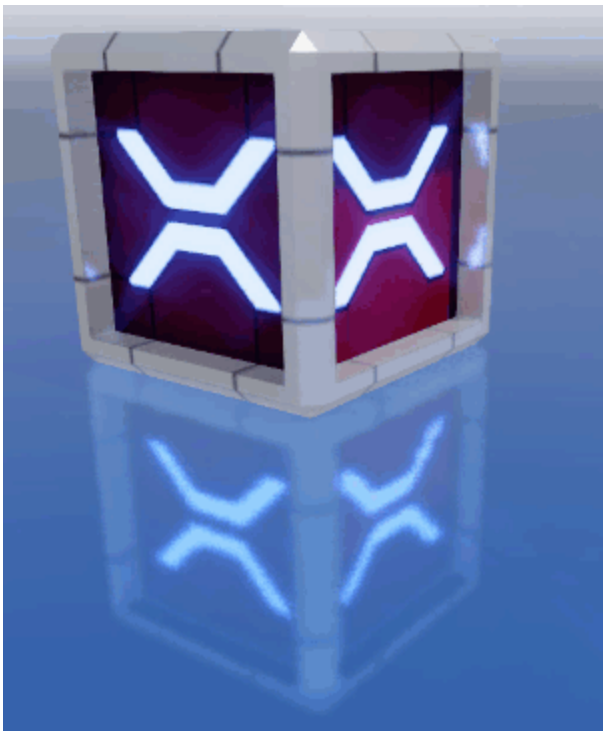
Enables the temporal pass. This reduces noise, but produces an animated "jittering" effect that is sometimes noticeable. The temporal effect is enabled by default.

Temporal effect: on	**Temporal effect: off**
	

> **NOTE**
>
> If the temporal effect is disabled, the other temporal properties have no effect.

Response

How quickly reflections blend between the reflection in the current frame and the history buffer. Lower values produce reflections faster, but with more jittering. Note the jittering in the reflection below:



If the camera in your game doesn't move much, we recommend values closer to `1`. The default value is `0.9`.

Scale

The intensity of the temporal effect. Lower values produce reflections faster, but with more noise. The default value is `4`.

See also

- Materials
- Materials — geometry attributes
- Post effects
- Graphics compositor

Graphics compositor

Advanced Programmer

> **NOTE**
> This page requires a basic understanding of graphics pipelines.

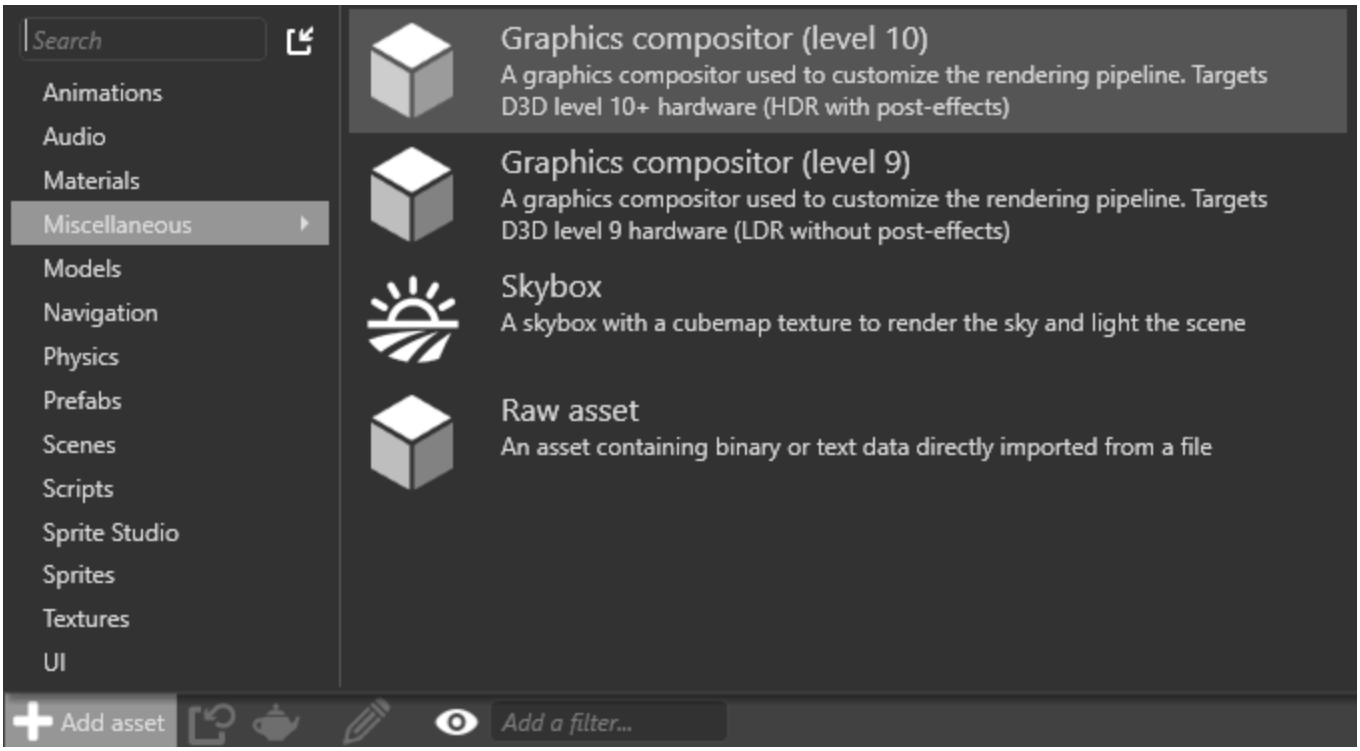
The **graphics compositor** organizes how scenes are rendered. You can use it to customize almost every part of the rendering pipeline. For example, you can:

- use one or multiple cameras
- filter entities
- render to one or more render textures, with different viewports
- set HDR or LDR rendering
- apply post effects to a render target, selected before or after rendering a camera
- clear a render target or clear only the depth buffer (eg to always render on top of a render target in a FPS game, or render the UI)
- modify the compositor from scripts (or any animation system), for example to modify post effects

Create a graphics compositor

Stride includes a graphics compositor when you create a project.

If you need to create another graphics compositor, in the **Asset View**, click **Add asset** and select **Misc > Graphics compositor**.

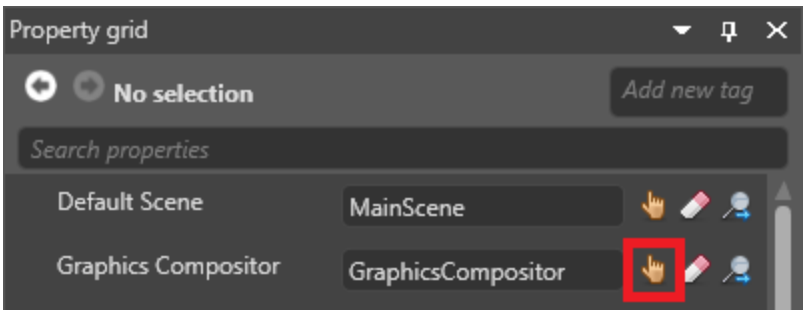


You can choose one of two presets:

- Level 10 (HDR with post effects)
- Level 9 (LDR with no post effects)

Set the graphics compositor

You can have multiple graphics compositors in your project, but you can only use one compositor at a time. At runtime, Stride uses the graphics compositor you specify in Game Settings.



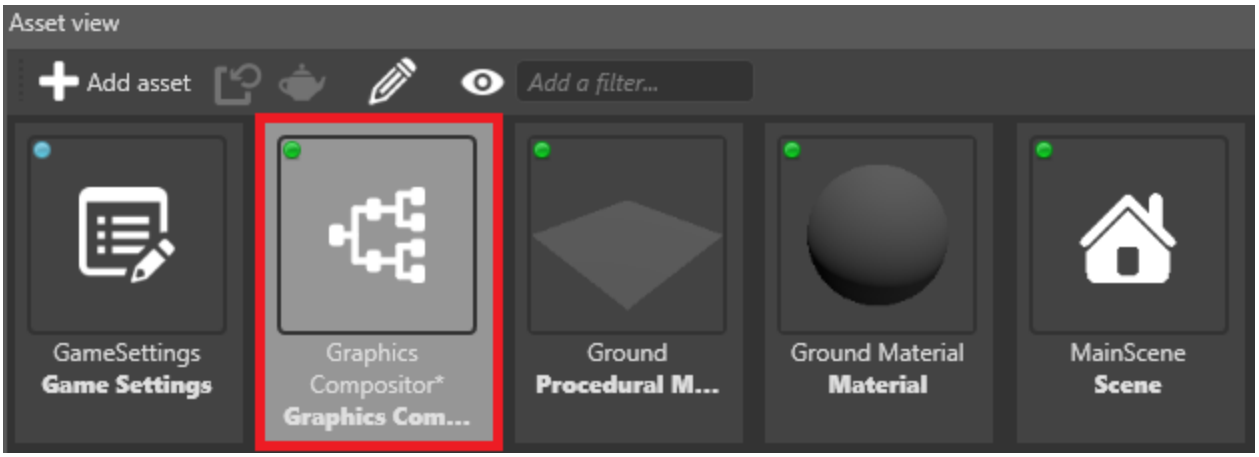
You can also change the graphics compositor at runtime in a script.

Open the graphics compositor editor

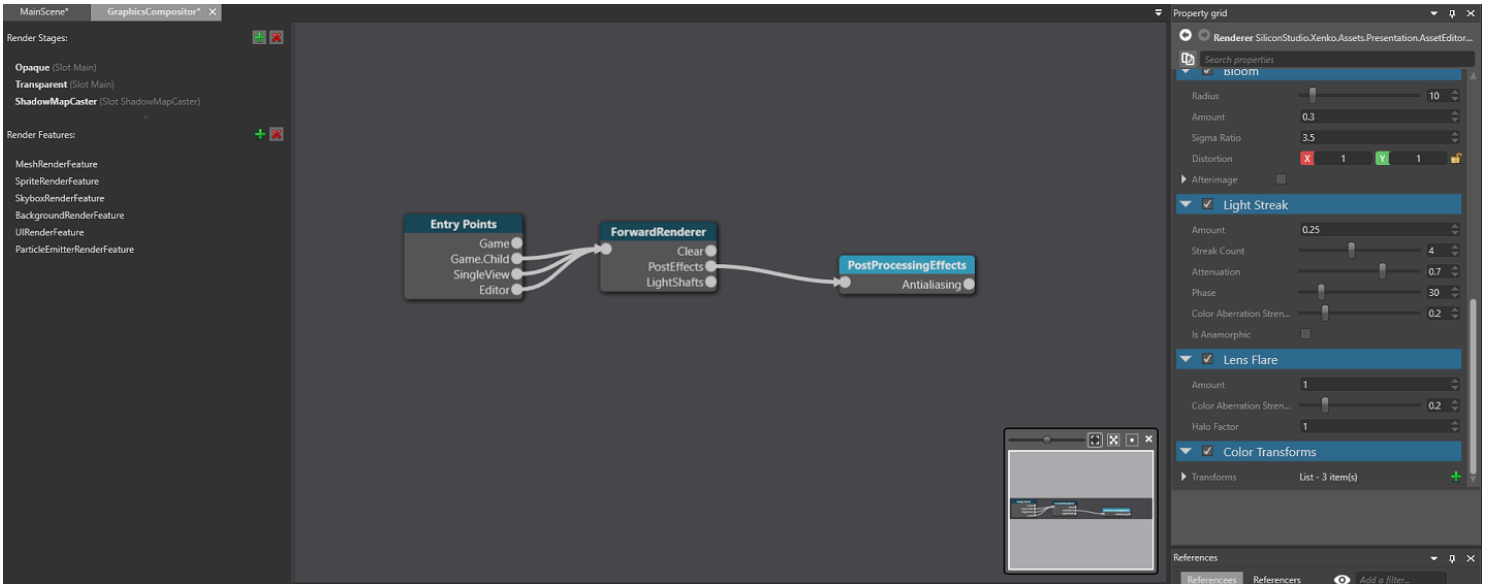
You customize the graphics compositor in the **graphics compositor editor**.

> **NOTE**
>
> The graphics compositor editor is an experimental feature.

In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



The **graphics compositor editor** opens.

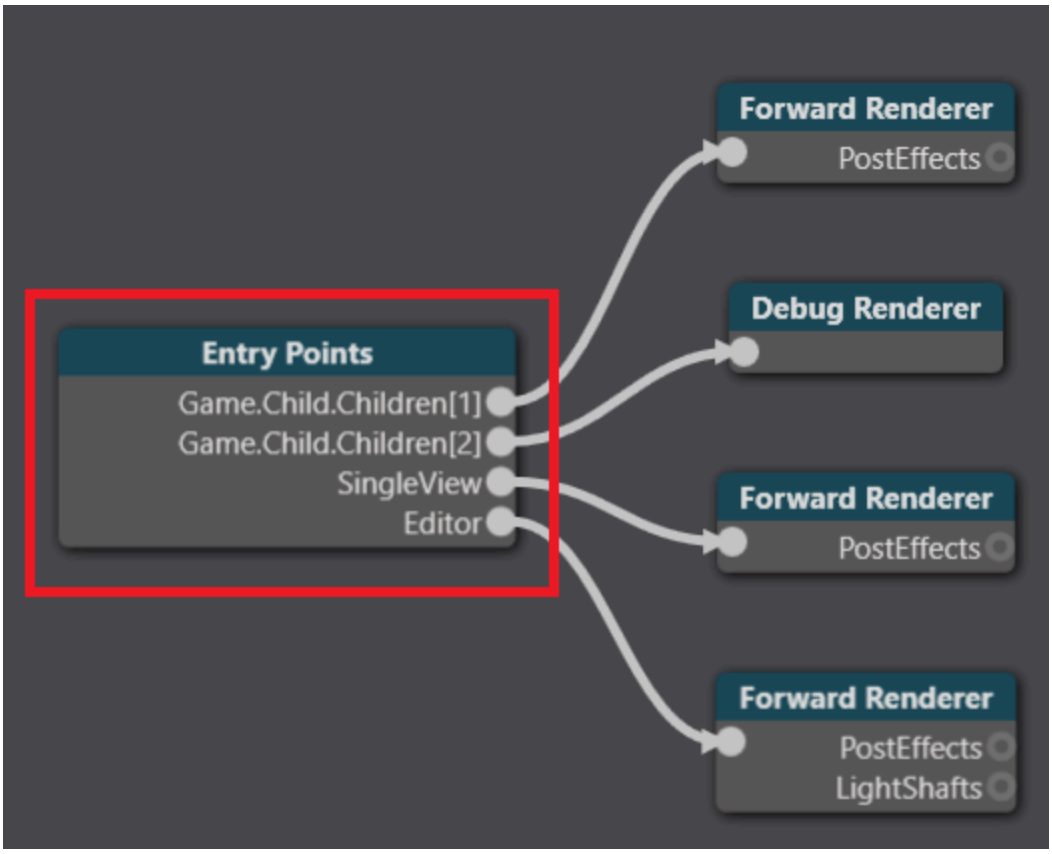


Nodes

The graphics compositor editor is divided into **nodes**. You can set the properties of each node in the **Property Grid** on the right.

Entry points

In the **Entry Points** node, you configure the pipeline for each entry point.



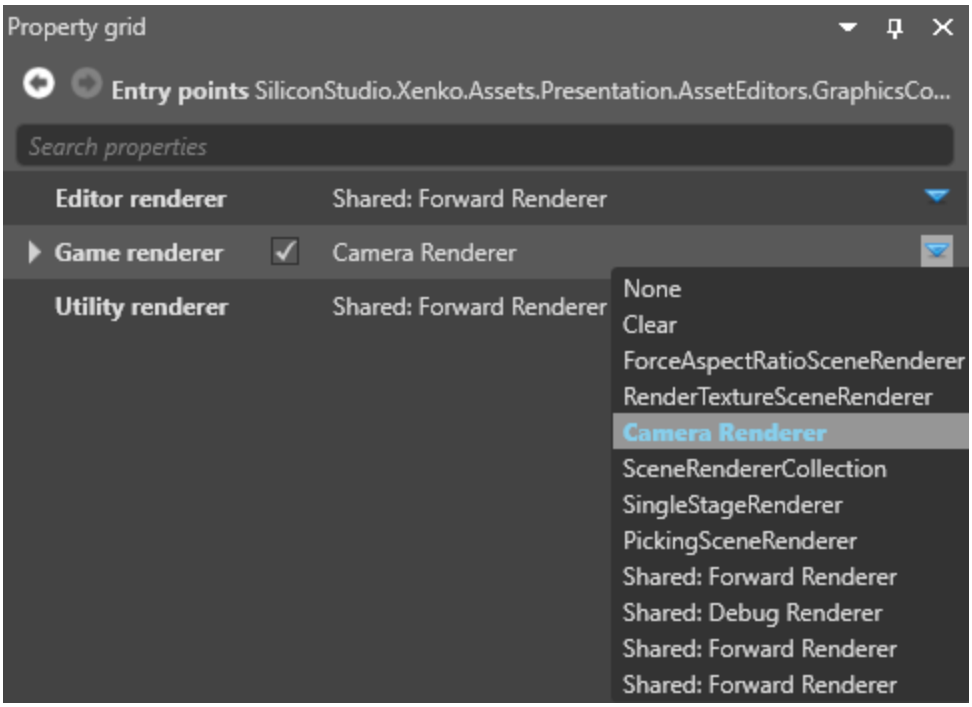
There are three entry points:

- **Game**, to render your game
- **Editor**, to render the Game Studio editor
- **Single view** (referred to as **Utility** in the Property Grid), to render other things, such as light probes and cubemaps

Each entry point can use a separate rendering pipeline. For example, the game and editor might share the same forward renderer and post-processing effects while your single view uses a separate forward renderer.

Connect an entry point to a renderer

1. Select the **Entry point** node.
2. In the **Property Grid**, next to the entry point you want to connect (**Editor**, **Game** or **Utility**), select the renderer you want to connect to.



For information about the different renderers, see Scene renderers.

Forward renderer

In a typical setup, the **forward renderer** renders almost everything in your scene. It renders, in order:

1. opaque objects
2. transparent objects
3. post effects

The forward renderer is also where you set virtual reality options. You configure the forward renderer properties in the **forward entry node**.

Debug renderer

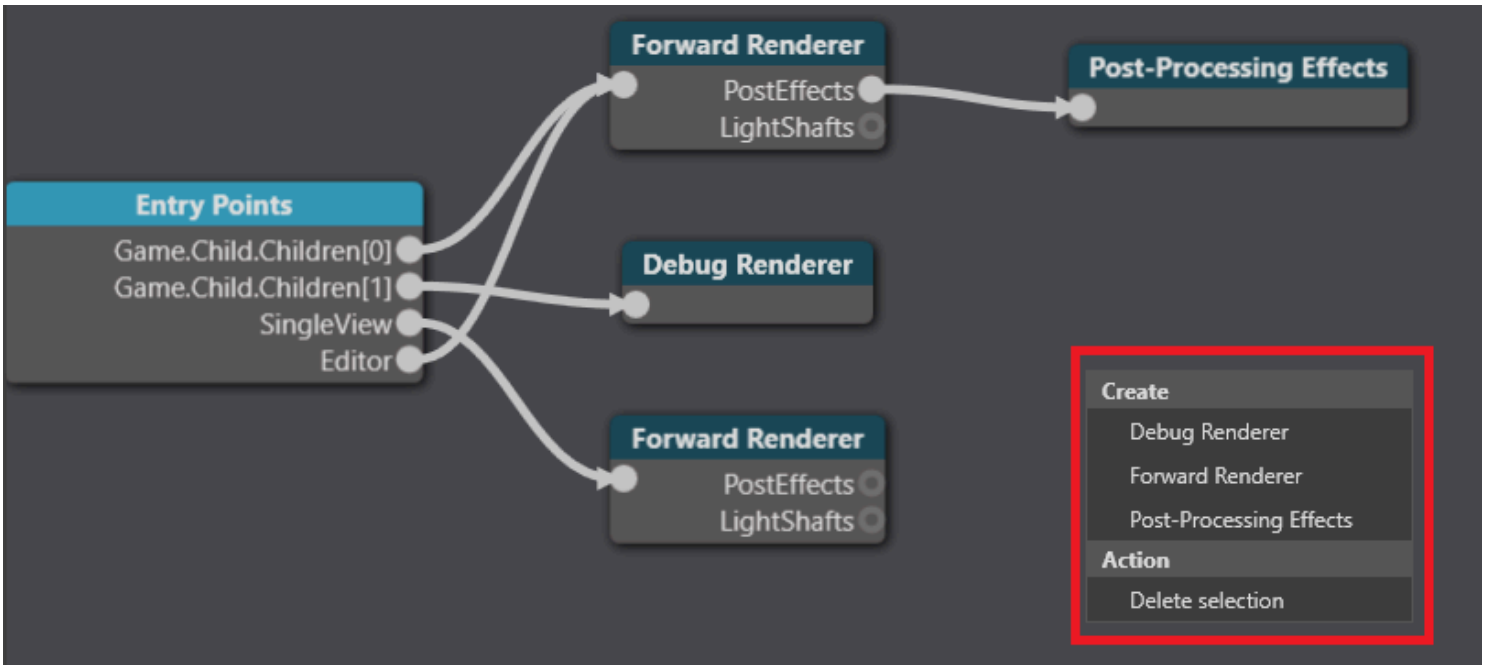
The **debug renderer** is used by scripts to print debug information. For more information, see Debug renderers.

Post-processing effects

The **post-processing effects** node comes after the forward renderer and controls the post effects in your game. For more information, see post-processing effects.

Create a node

To create a node, right-click the graphics compositor editor and select the type of node you want to create:



See also

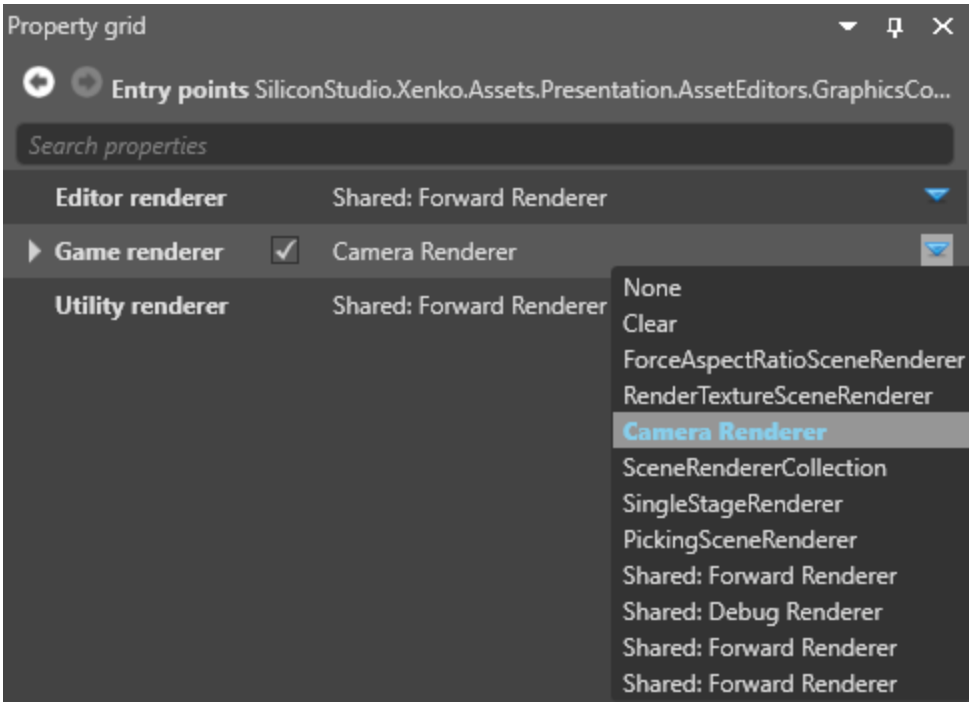
- Camera slots
- Scene renderers
  - Custom scene renders
- Debug renderers

Scene renderers

Intermediate Designer

Scene renderers let you customize the **collect** and **draw** phases of the rendering. For more information about these stages, see Render features.

You select scene renderers in the **entry points** node properties.

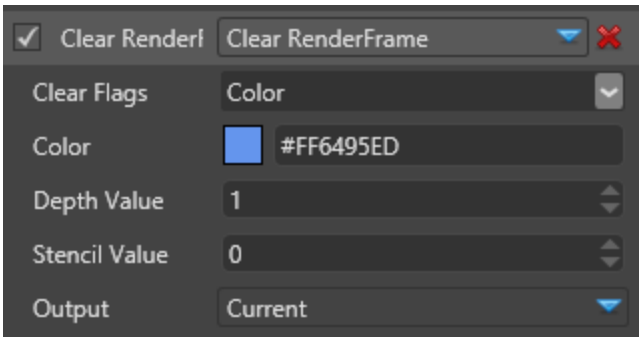


For more information about selecting renderers, see the Graphics compositor page.

> **NOTE**
>
> Currently, **all** renderers must have a camera, or be a child of a renderer that has a camera. This applies even to renderers that don't necessarily use cameras, such as the single stage renderer (eg to render a UI).

Clear

Clears a frame, with a solid color.



Properties

Property	Description
Clear flags	What to clear in the render frame (**Color only**, **Depth only**, or **Color and depth**)
Color	The color used to clear the color texture of the render frame. Only valid when **Clear Flags** is set to **Color** or **Color and depth**
Depth value	The depth value used to clear the depth texture of the render frame
Stencil value	The stencil value used to clear the stencil texture of the render frame

Camera renderer

Uses Child to render a view from a camera slot. The **render camera** renderer takes the input from a camera in the scene so it can be displayed somewhere.

Properties

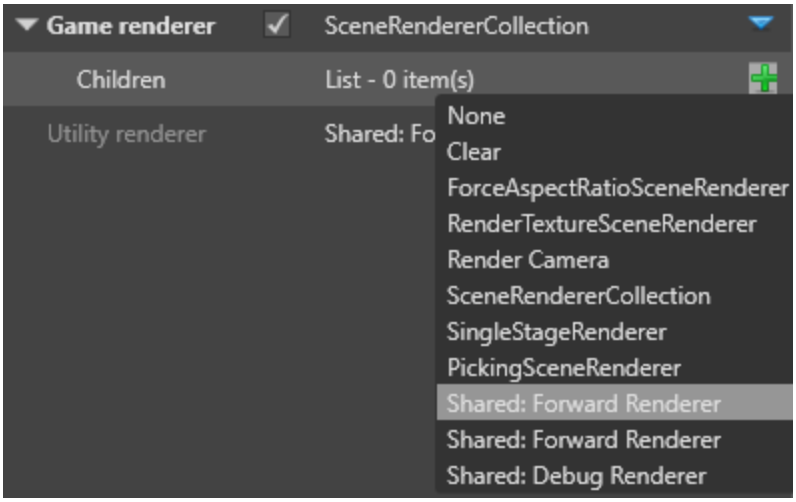
Property	Description
Camera	Specify a camera slot to render from
Child	Specify a renderer for the camera (eg a forward renderer or a custom renderer)

Scene renderer collection

The **scene renderer collection** executes multiple renderers (eg camera renderer, render texture, etc) in sequence. This lets you set multiple renderers for an entry point. You can add as many renderers to the collection as you need.

> **NOTE**
>
> Stride executes the renderers in list order.

To add a renderer to the collection, next to **Children**, click (**Add**) and select the renderer you want to add.



Forward renderer

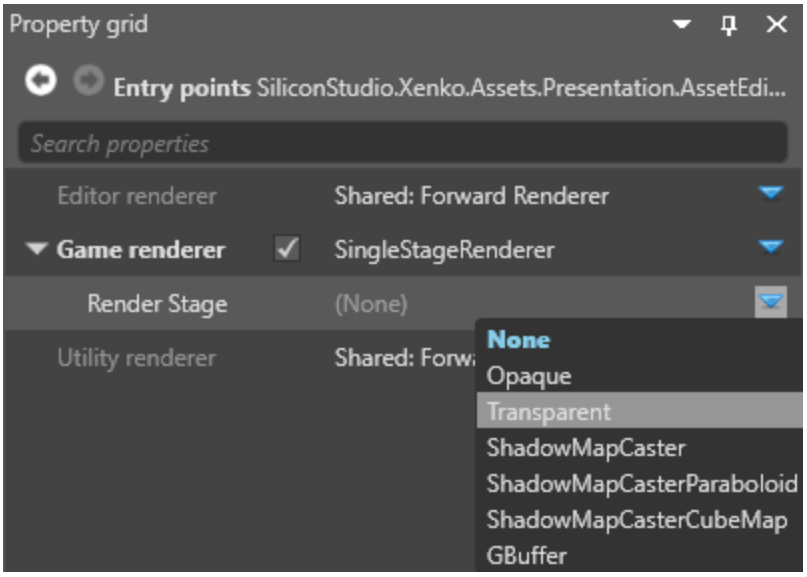
In a typical setup, the **forward renderer** renders almost everything in your scene. It renders, in order:

1. opaque objects
2. transparent objects
3. post effects

The forward renderer is also where you set VR options. For more information, see Virtual reality.

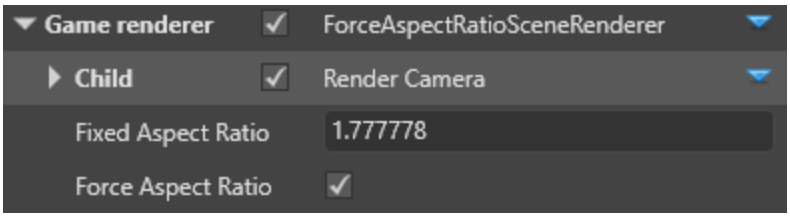
You configure the forward renderer properties in the **forward entry node**.

Single stage renderer



Force aspect ratio scene renderer

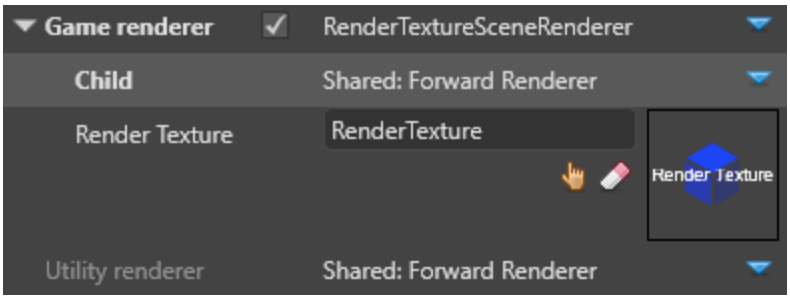
Uses ForceAspectRatioSceneRenderer to force an aspect ratio and applies a letterbox if the ratio is different from the screen. Use this before the **render camera**.



Property	Description
Child	Specify a renderer for the camera (eg a forward renderer or a custom renderer)
Fixed aspect ratio	The aspect ratio to force the view to
Force aspect ratio	Enable forced aspect ratio

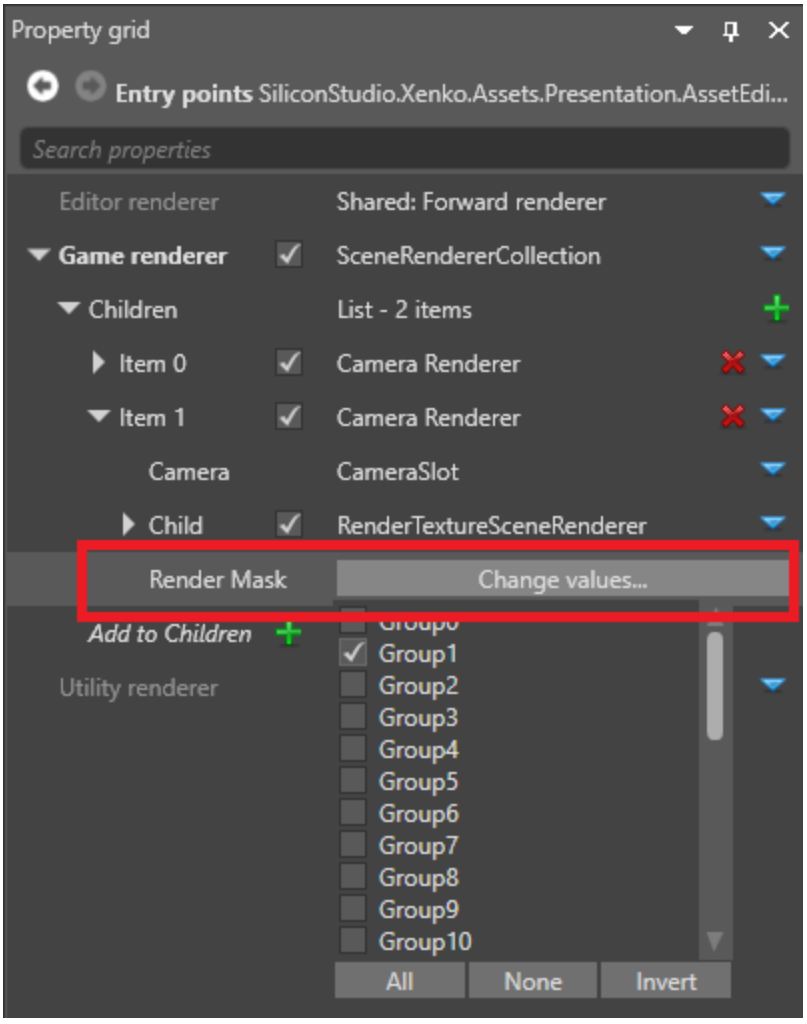
Render texture

Renders to a render texture, which you can display in your scene (eg to display security camera footage on a screen). For more information, see Render textures.



Property	Description
Child	Specify a renderer for the camera (eg a forward renderer or a custom renderer)
Render texture	Specify a texture to render to

Render mask



The **render mask** filters which groups are rendered. You can use it to only render particular models. For more information, see Render groups and render masks

See also

- Graphics compositor
- Camera slots
- Custom scene renders
- Debug renderers
- Render groups and render masks

Custom scene renderers

To create a custom renderer, directly implement the ISceneRenderer or use a delegate through the DelegateSceneRenderer.

Implement an ISceneRenderer

The SceneRendererBase provides a default implementation of ISceneRenderer. It automatically binds the output defines on the renderer to the GraphicsDevice before calling the `DrawCore` method.

```
[DataContract("MyCustomRenderer")]
[Display("My Custom Renderer")]
public sealed class MyCustomRenderer : SceneRendererBase
{
    // Implements the DrawCore method
    protected override void DrawCore(RenderContext context, RenderDrawContext drawContext)
    {
        // Access to the graphics device
        var graphicsDevice = drawContext.GraphicsDevice;
        var commandList = drawContext.CommandList;
        // Clears the current render target
        commandList.Clear(commandList.RenderTargets[0], Color.CornflowerBlue);
        // [...]
    }
}
```

Use a delegate

To develop a renderer and attach it to a method directly, use DelegateSceneRenderer:

```
var sceneRenderer = new DelegateSceneRenderer(
    (drawContext) =>
    {
        // Access to the graphics device
        var graphicsDevice = drawContext.GraphicsDevice;
        var commandList = drawContext.CommandList;
        // Clears the current render target
        commandList.Clear(commandList.RenderTargets[0], Color.CornflowerBlue);
        // [...]
   });
```

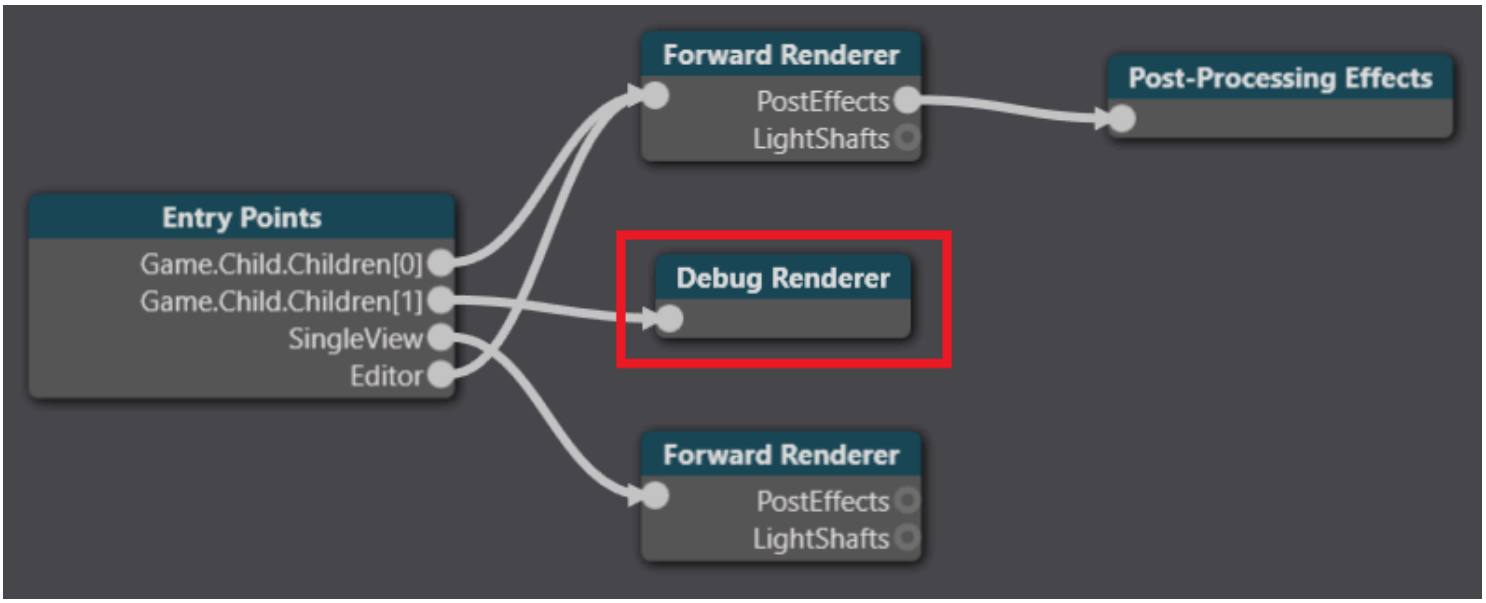
See also

- Scene renderers

- Debug renderers

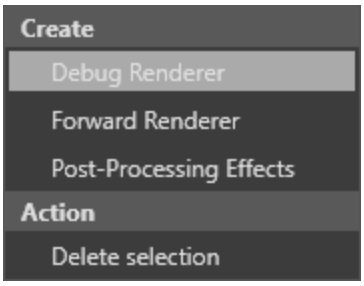
Debug renderer

The **debug renderer** is a placeholder renderer you can use with scripts to print debug information. By default, the debug renderer is included in the graphics compositor as a child of the game entry point.



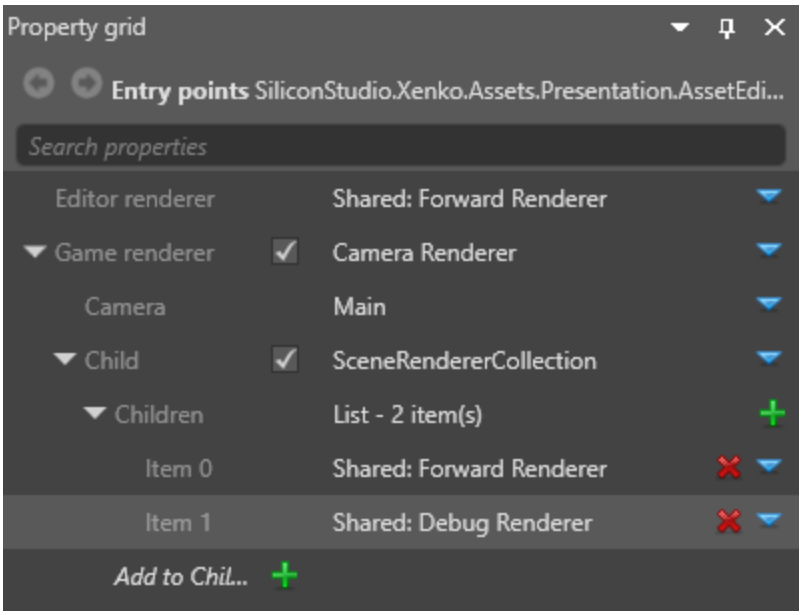
Create a debug renderer

To create a debug renderer, right-click the graphics compositor editor and select **Debug renderer**.



Connect a debug renderer to an entry point

In most cases, you want the debug renderer to share an entry point with one or more forward renderers. To do this, use a **scene renderer collection** and select the debug renderer and forward renderer(s) as children, as in the screenshot below:



Use a debug renderer

To use the debug renderer, reference it in your script and add debug render stages.

For example, the **Debug physics shapes** script included in Stride uses the debug renderer to display collider shapes at runtime.

```
using System.Linq;
using System.Threading.Tasks;
using Stride.Input;
using Stride.Engine;
using Stride.Physics;
using Stride.Rendering;
using Stride.Rendering.Compositing;

namespace MyGame
{
    public class DebugPhysicsShapes : AsyncScript
    {
        public RenderGroup RenderGroup = RenderGroup.Group7;

        public override async Task Execute()
        {
        //set up rendering in the debug entry point if we have it
        var compositor = SceneSystem.GraphicsCompositor;
        var debugRenderer =
            ((compositor.Game as SceneCameraRenderer)?.Child as
SceneRendererCollection)?.Children.Where(
                x => x is DebugRenderer).Cast<DebugRenderer>().FirstOrDefault();
        if (debugRenderer == null)
```

```
            return;

        var shapesRenderState = new RenderStage("PhysicsDebugShapes", "Main");
            compositor.RenderStages.Add(shapesRenderState);
            var meshRenderFeature = compositor.RenderFeatures.OfType<MeshRenderFeature>
().First();
            meshRenderFeature.RenderStageSelectors.Add(new SimpleGroupToRenderStageSelector
            {
                EffectName = "StrideForwardShadingEffect",
                RenderGroup = (RenderGroupMask)(1 << (int)RenderGroup),
                RenderStage = shapesRenderState,
            });
            meshRenderFeature.PipelineProcessors.Add(new WireframePipelineProcessor {
RenderStage = shapesRenderState });
            debugRenderer.DebugRenderStages.Add(shapesRenderState);

            var simulation = this.GetSimulation();
            if (simulation != null)
                simulation.ColliderShapesRenderGroup = RenderGroup;

            var enabled = false;
            while (Game.IsRunning)
            {
                if (Input.IsKeyDown(Keys.LeftShift) && Input.IsKeyDown(Keys.LeftCtrl)
&& Input.IsKeyReleased(Keys.P))
                {
                    if (simulation != null)
                    {
                        if (enabled)
                        {
                            simulation.ColliderShapesRendering = false;
                            enabled = false;
                        }
                        else
                        {
                            simulation.ColliderShapesRendering = true;
                            enabled = true;
                        }
                    }
                }

                await Script.NextFrame();
            }
        }
    }
}
```

For information about how to use this script, see Colliders.

See also

- Scene renderers
  - Custom scene renders
- Physics — Colliders

Render textures

Intermediate Designer Programmer

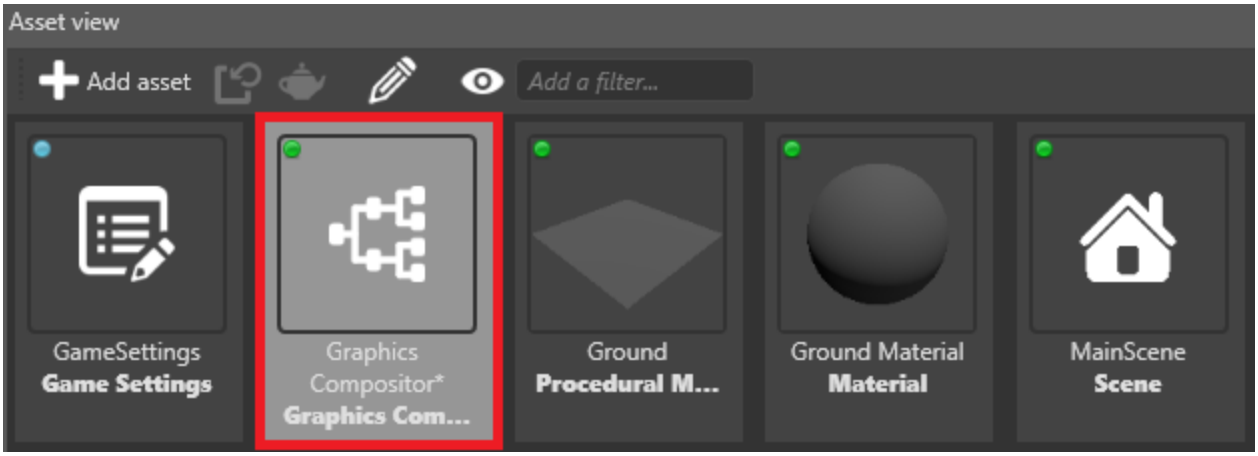
With **render textures**, you can send a camera's view to a texture and use the texture on objects in your scene. For example, you can use this to display part of your scene on a TV screen in the same scene, such as security camera footage.

For API details, see Textures and render textures.

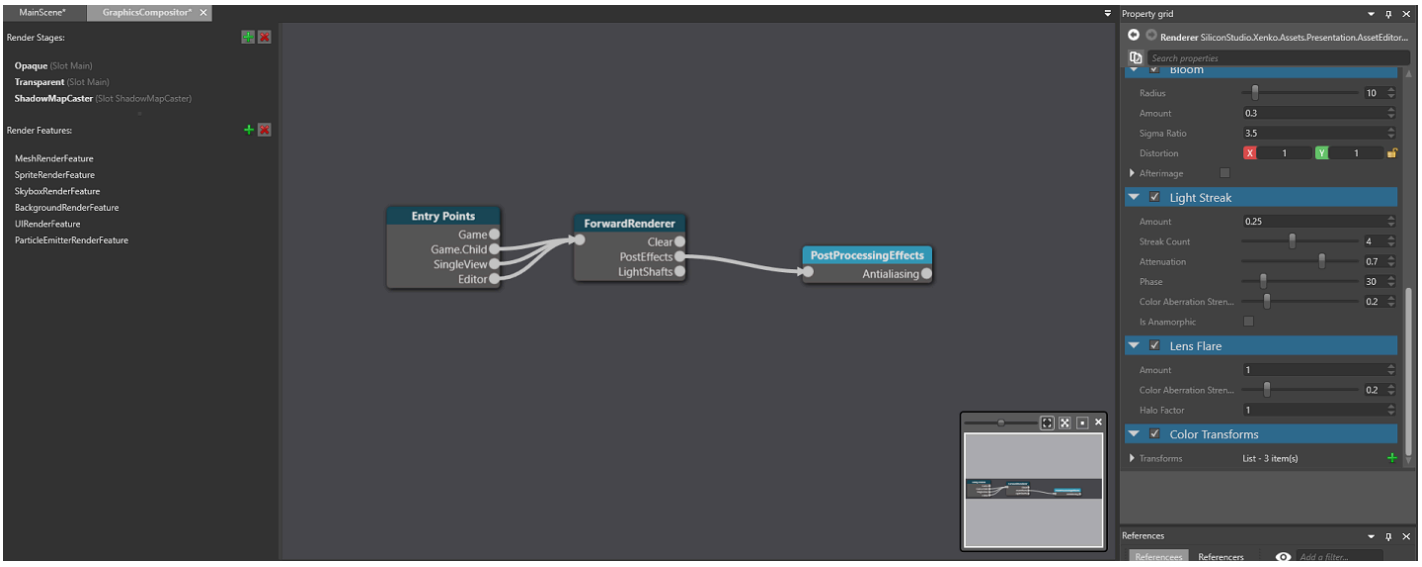
1. Create an extra camera slot

Camera slots link the graphics compositor to the cameras in your scene. You need to add a camera slot for a new camera to use. For more information about camera slots, see Camera slots.

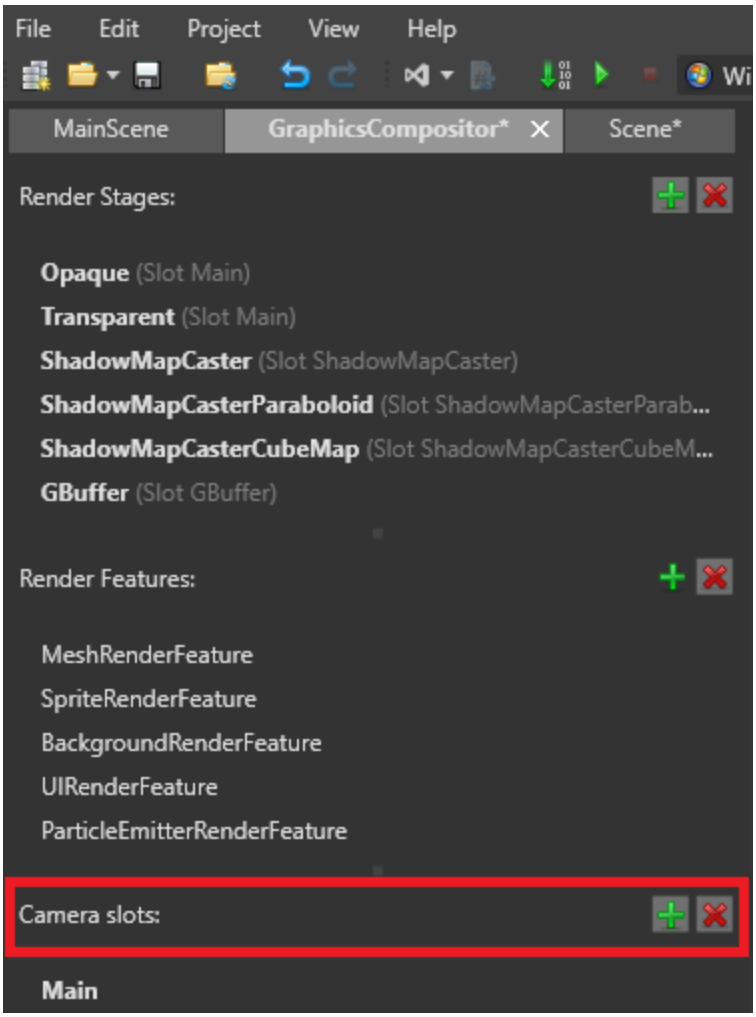
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



   The graphics compositor editor opens.



2. On the left, under **Camera slots**, click (**Add**).

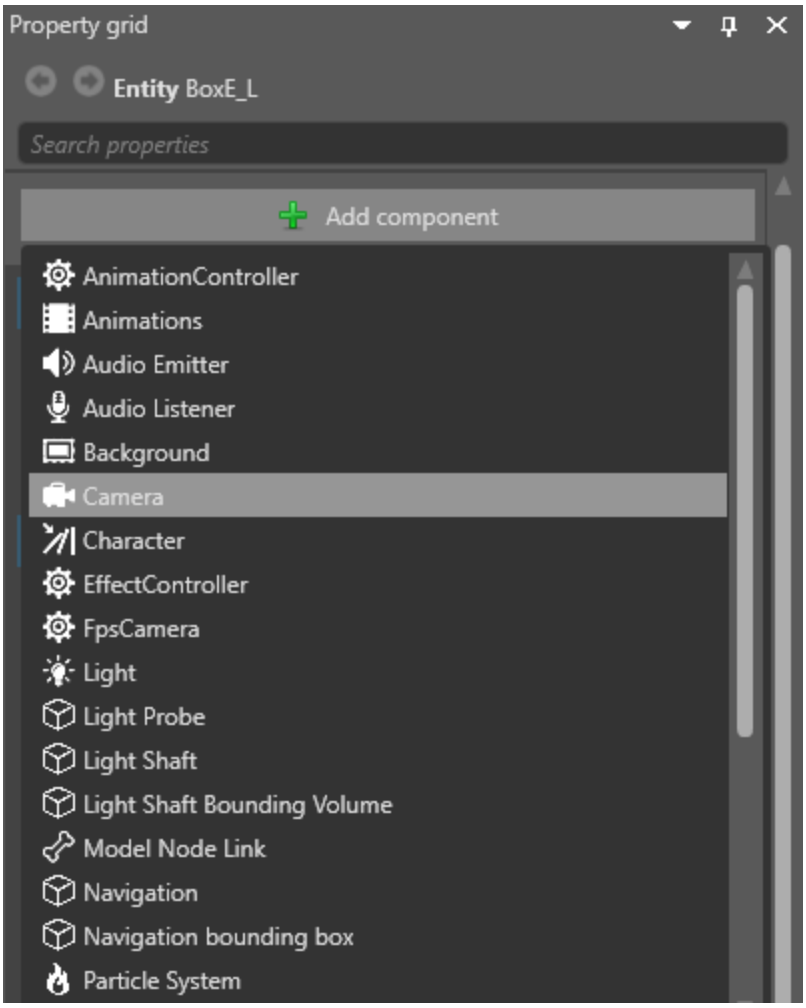


Game Studio adds a new camera slot.

> **TIP**
>
> To rename a camera slot, double-click it in the list and type a new name.
>
>
>

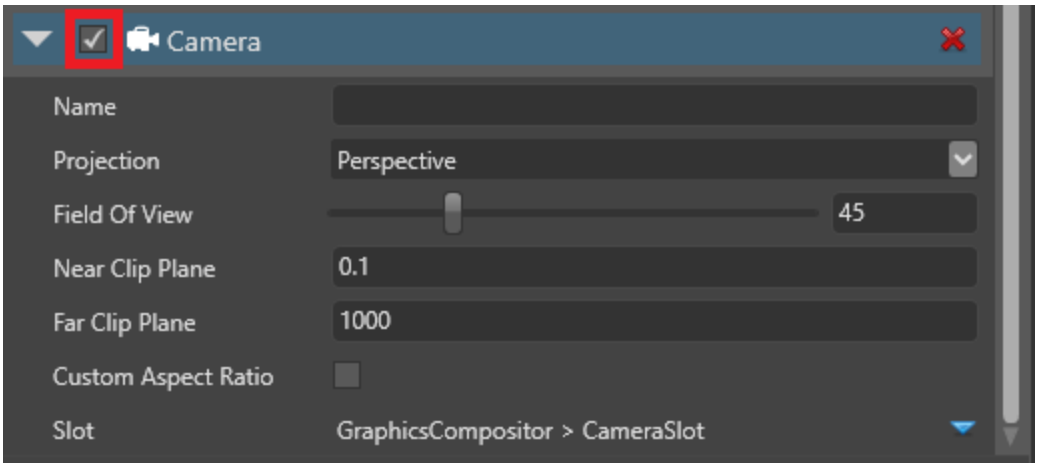
2. Create a camera and bind it to the slot

1. In your scene, add a **camera component** to the entity you want to be your camera.

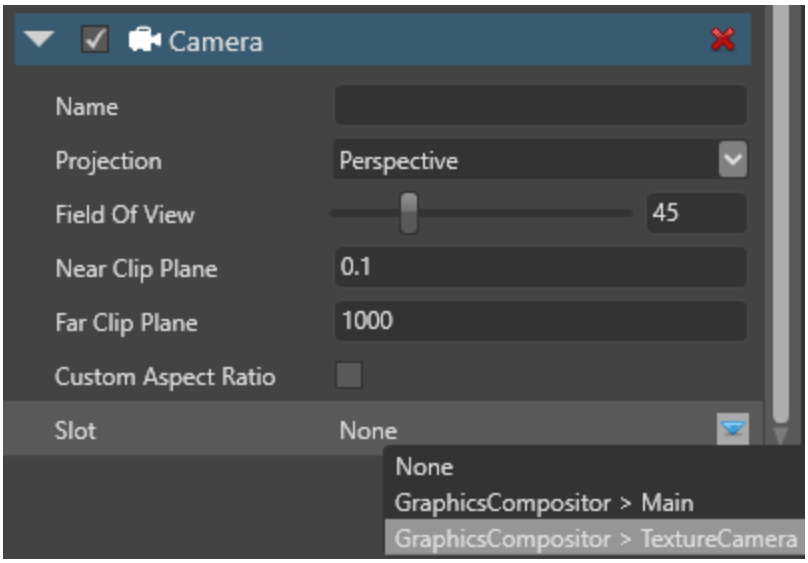


2. Position the entity so the camera captures the area of the scene you want to render to a texture.

3. In the entity **Property Grid**, enable the **Camera** component using the checkbox.

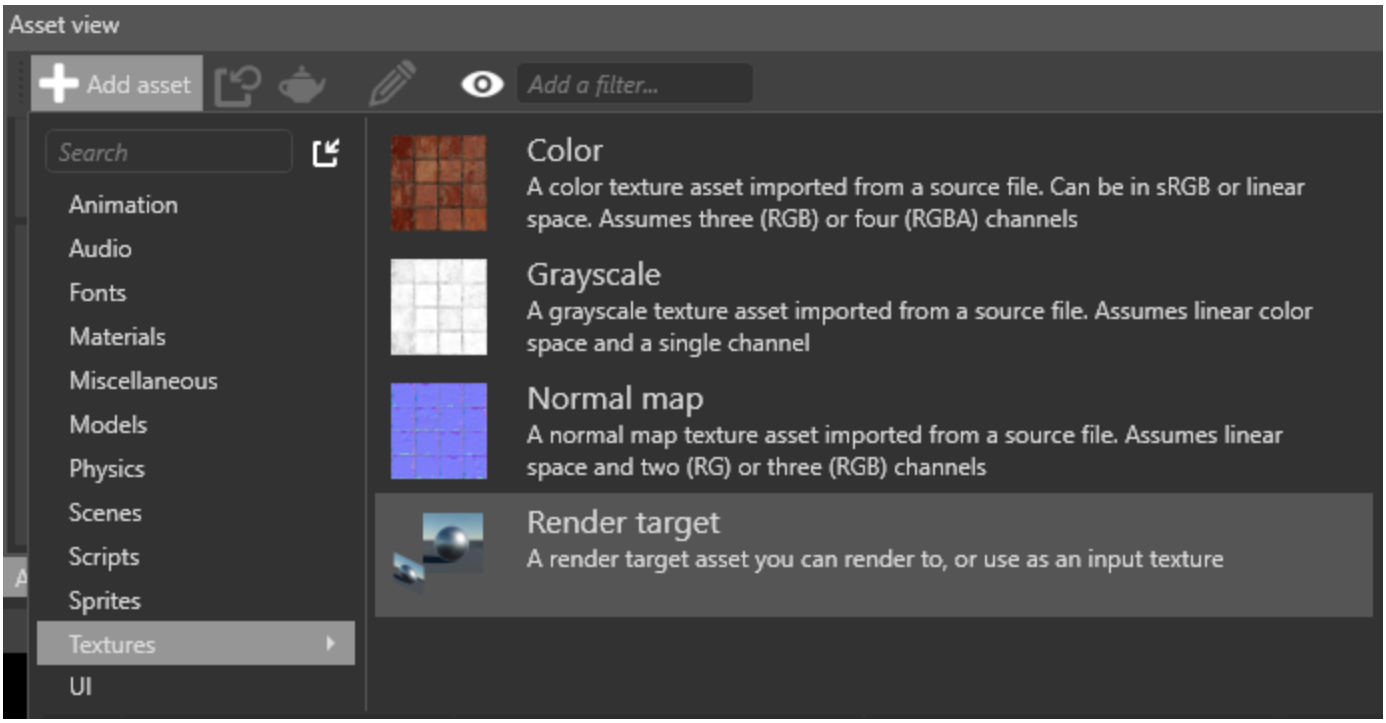


4. in the **Camera** component properties, under **Slot**, select the slot you created in the previous step.

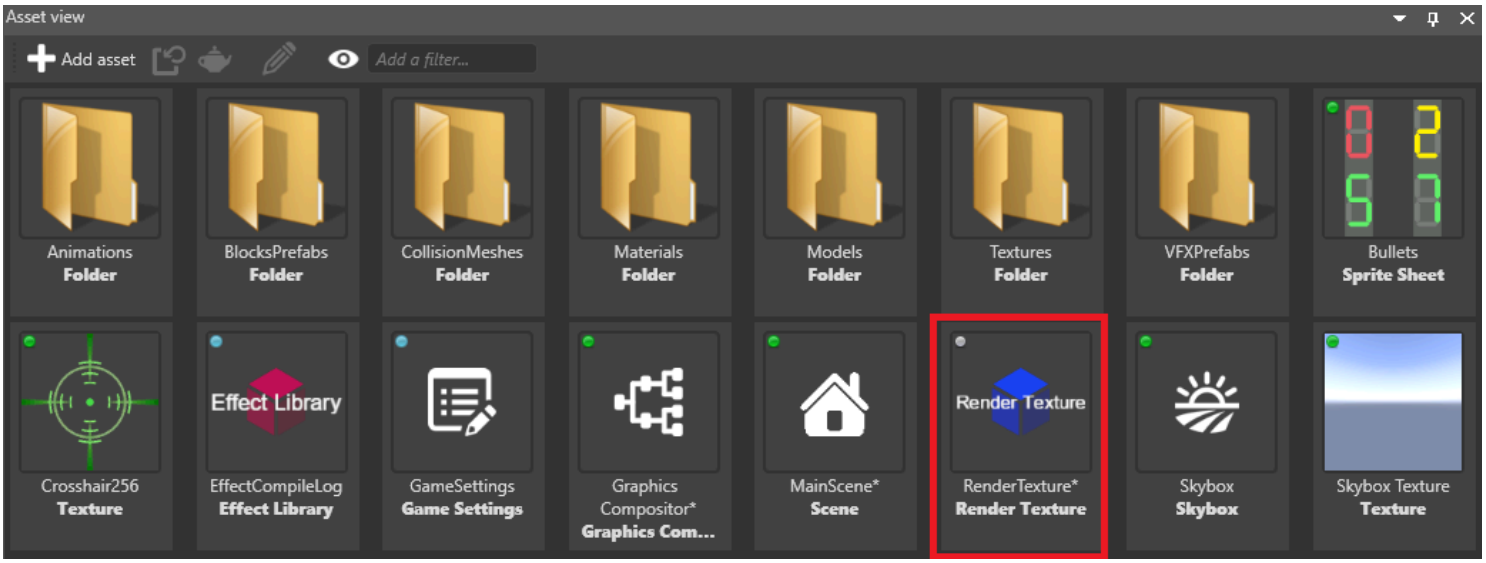


3. Create a render target texture

In the **Asset View**, click **Add asset** and select **Texture** > **Render target**.



Game Studio adds a **render target** texture to your project assets.

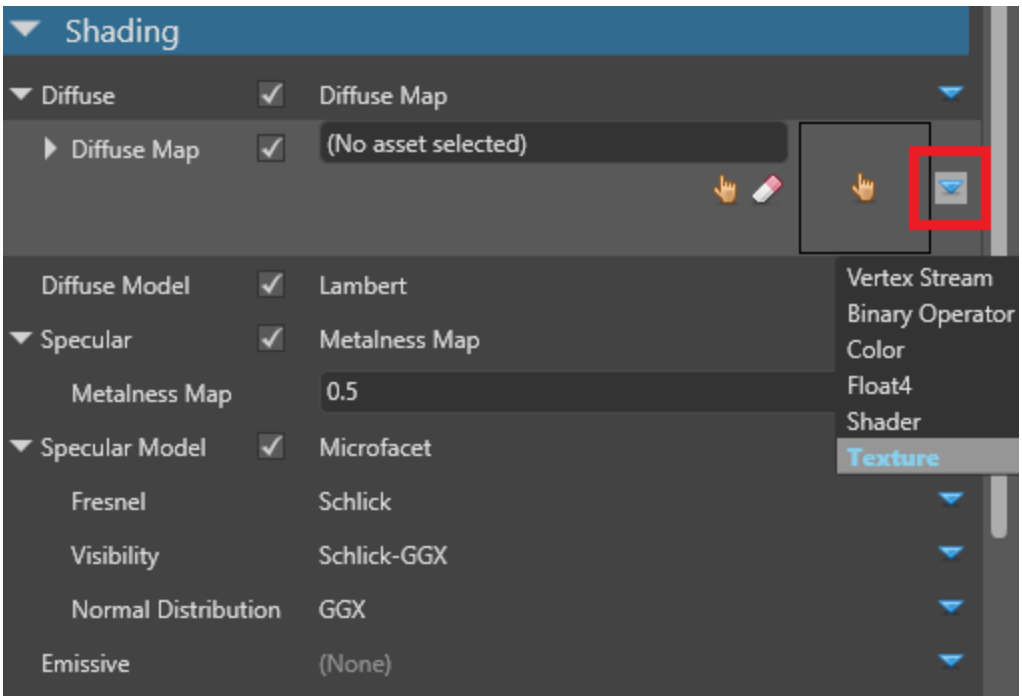


4. Place the render target texture in the scene

There are various ways you can use the render target texture.

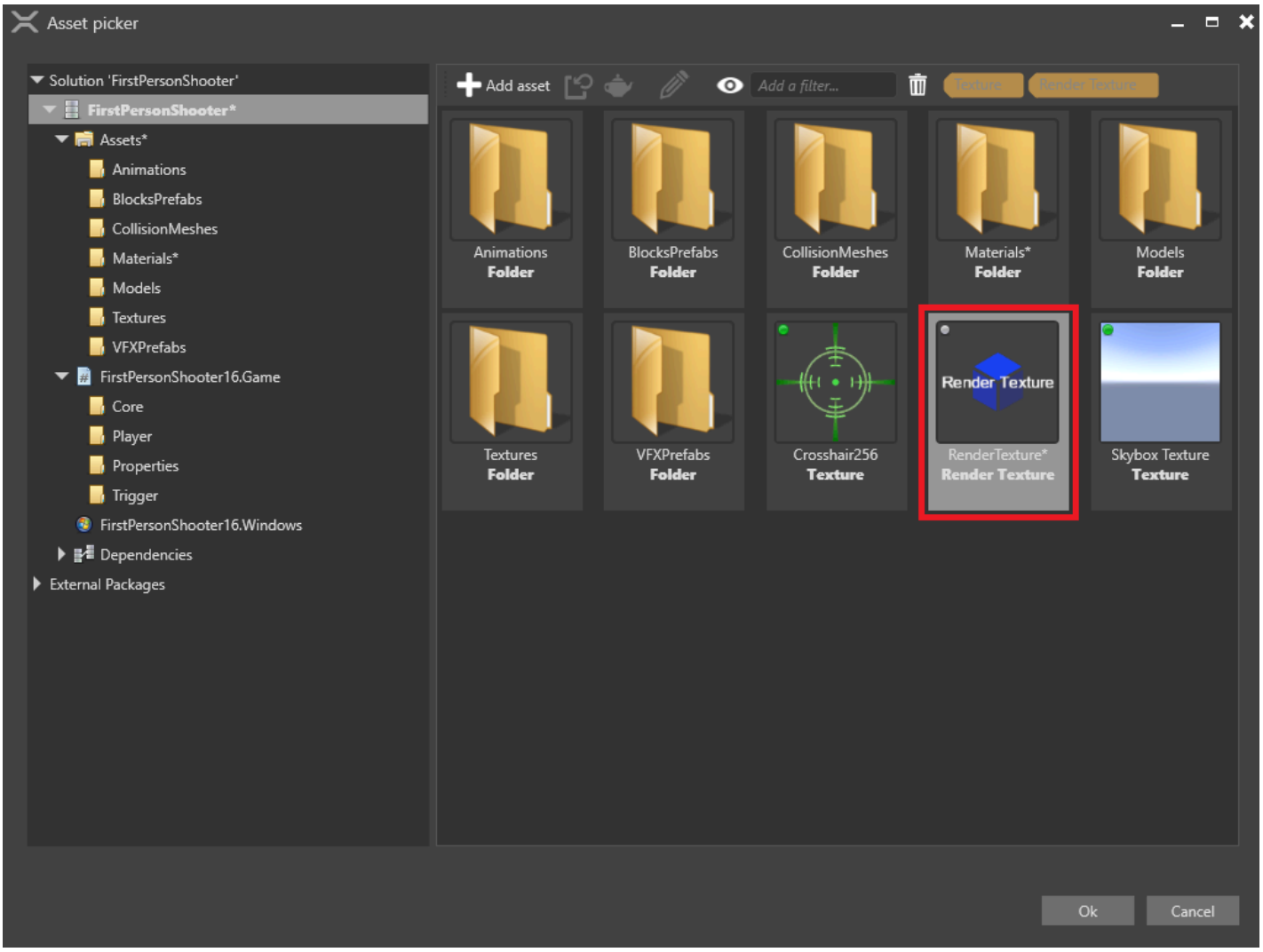
Example 1: Use the render target texture in a material

1. In the material properties, under **Shading**, next to **Diffuse map**, click (**Replace**) and select **Texture**.



2. Click (**Select an asset**).

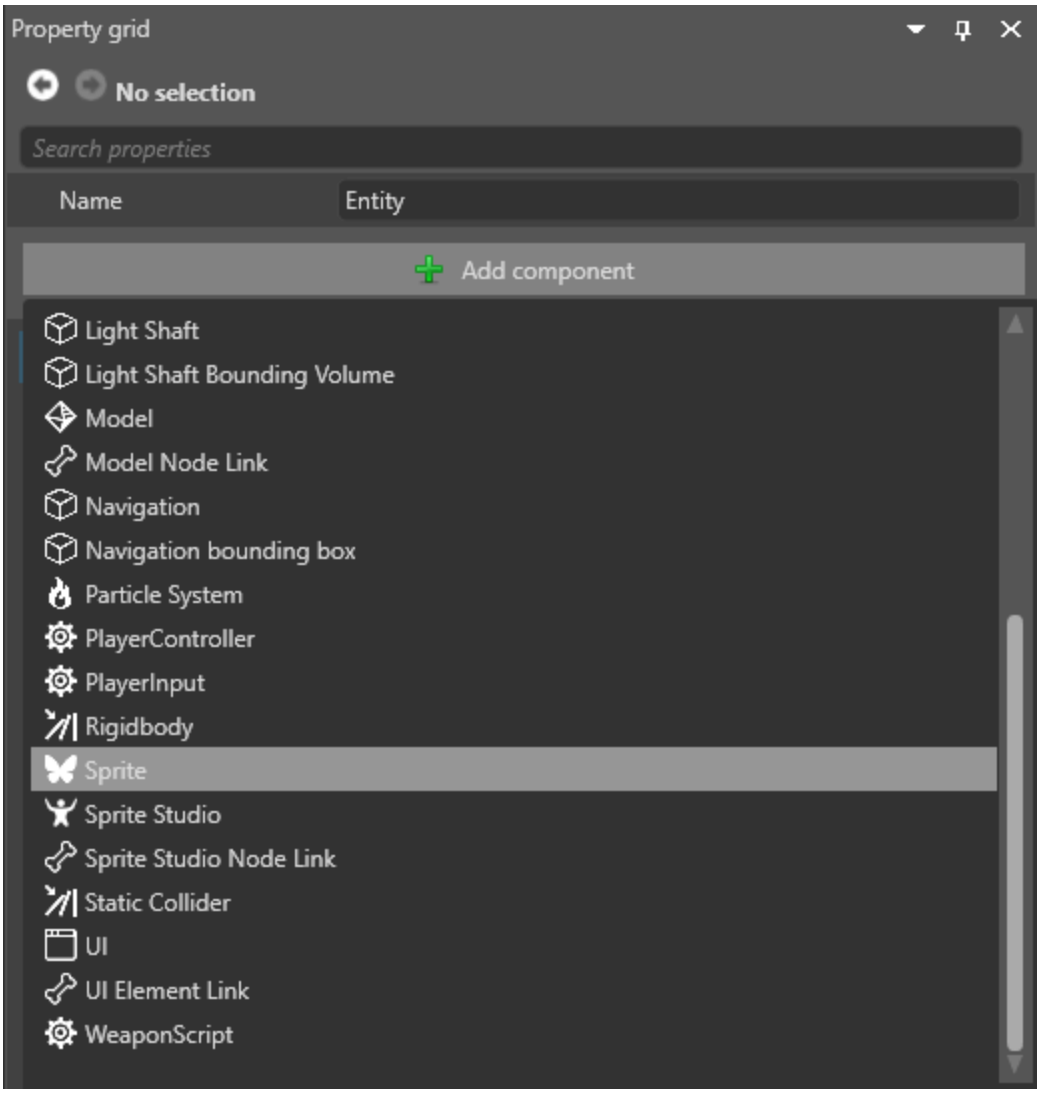
3. Select the **Render texture** asset and click **OK**.



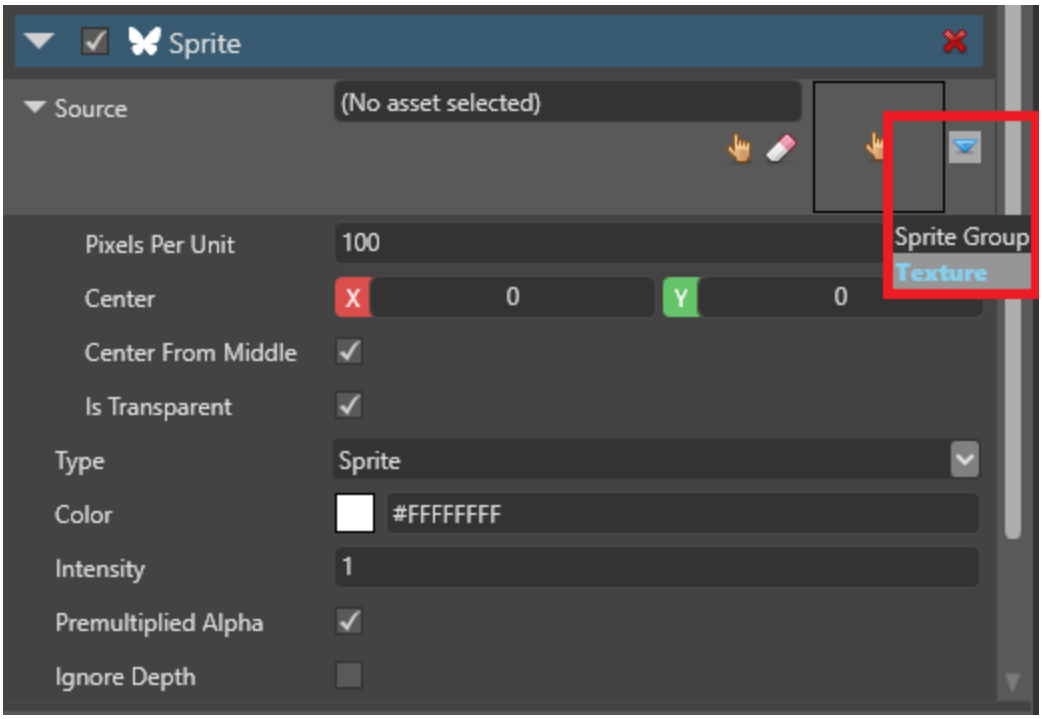
Example 2: Use the render target texture in a sprite component

1. Create an entity and position it where you want to display the texture.

2. With the entity selected, in the **Property Grid**, click **Add component** and add a **sprite** component.



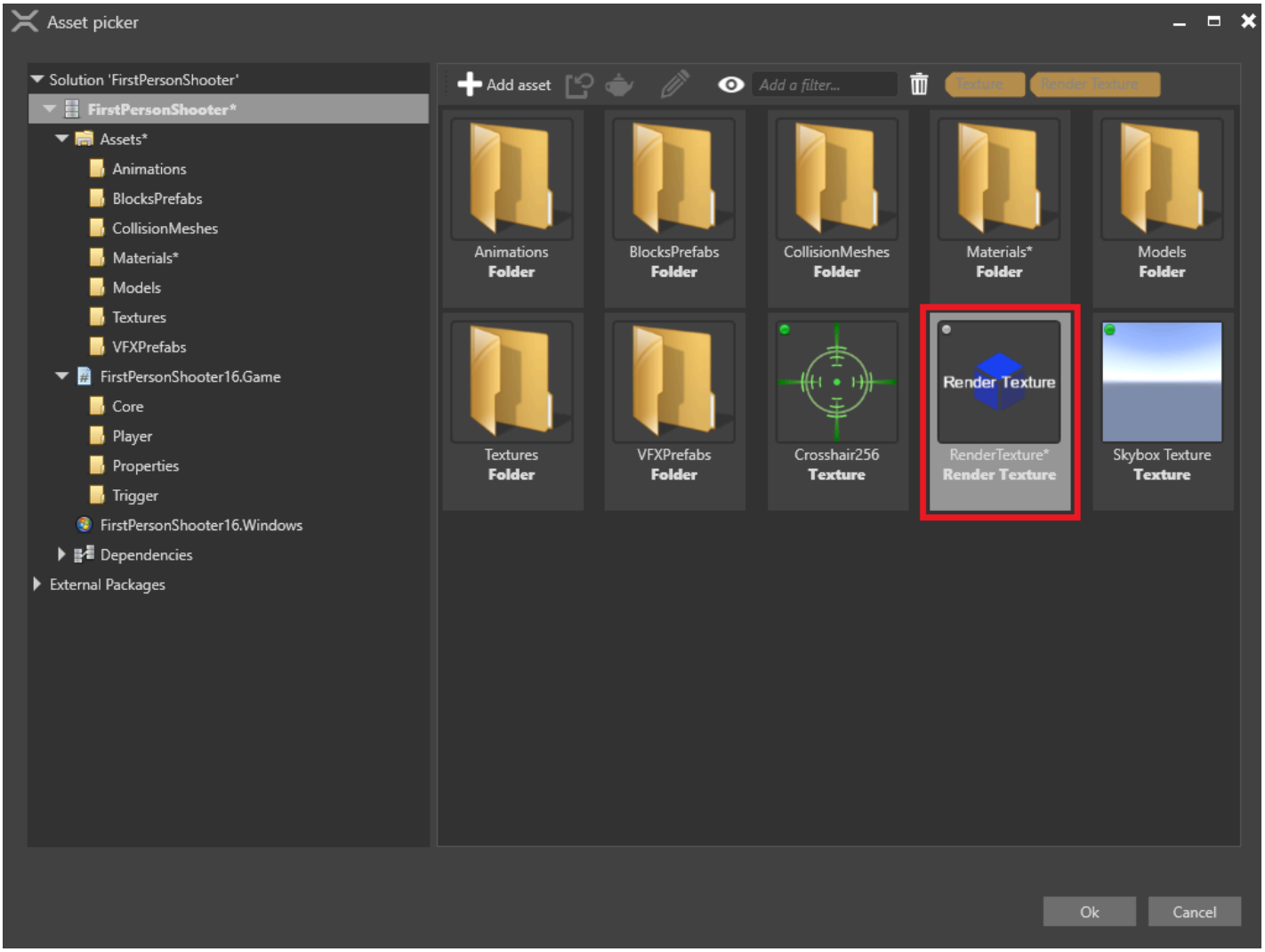
3. In the sprite component properties, next to **Source**, click (**Replace**) and select **Texture**.



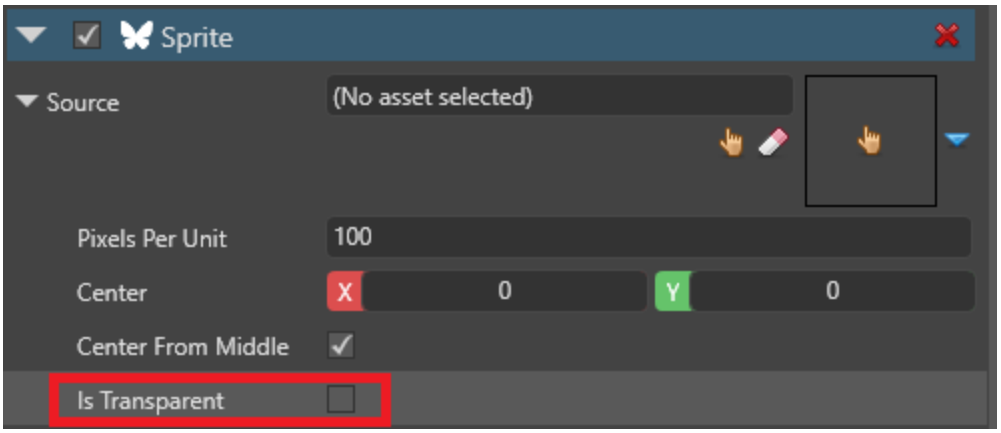
4. Click (**Select an asset**).

The **Select an asset** window opens.

5. Select the **Render texture** asset and click **OK**.



6. If you don't want the texture to be semi-transparent, under the **Source** properties, clear the **Is transparent** checkbox.



5. Set up the graphics compositor

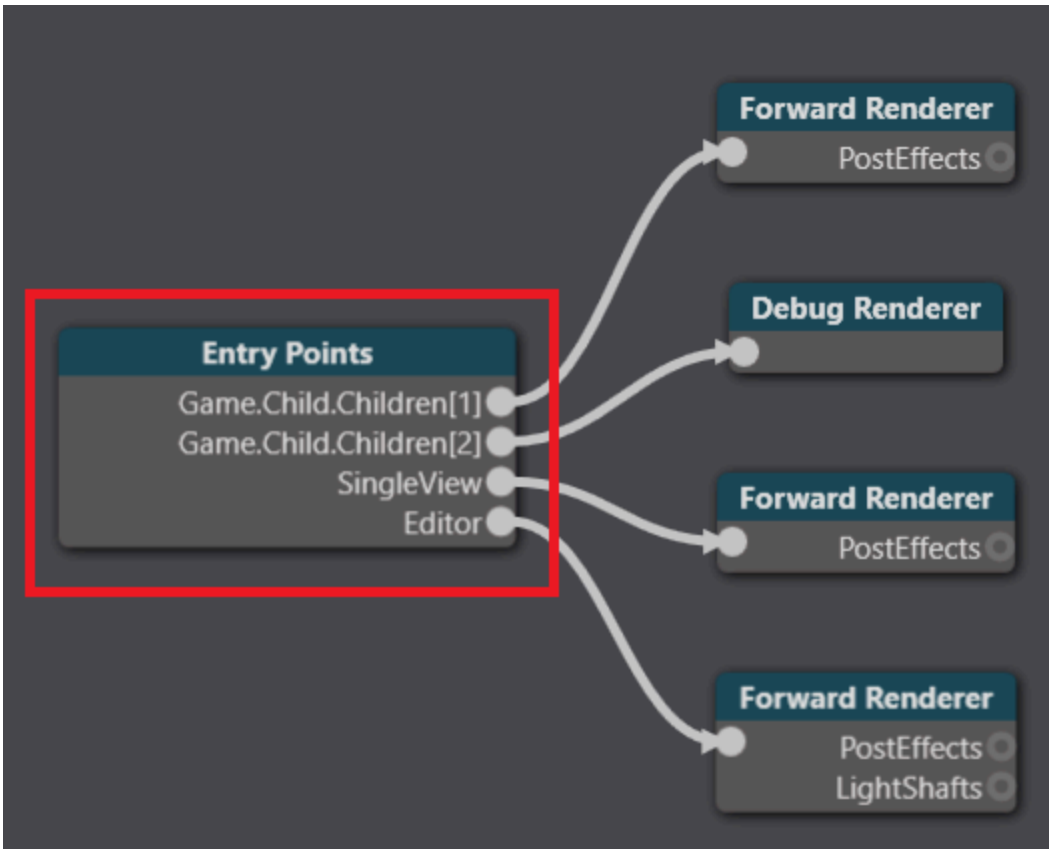
To display a render texture in your scene, you need at least two renderers:

- one to render your main camera
- one to render the second camera to the render texture

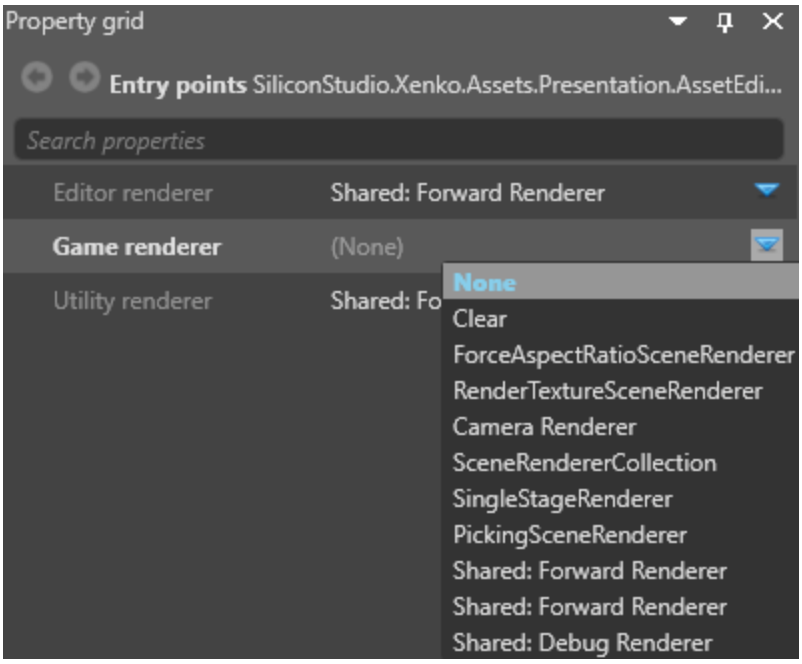
This page describes the simplest way to do this from scratch, using two cameras and two renderers. Depending on your pipeline, you might need to create a different setup.

> **WARNING**
> These instructions involve deleting your existing renderers for the game entry point. You might want to make a backup of your project in case you want to restore your pipeline afterwards.

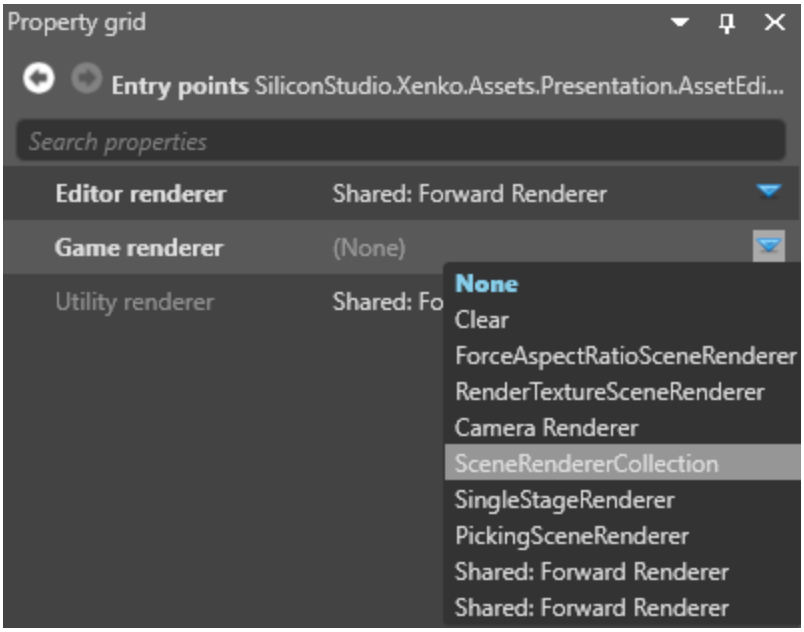
1. In the graphics compositor editor, select the **Entry points** node.



2. In the **Property Grid** on the right, next to **Game renderer**, click (**Replace**) and select **None** to delete your existing renderers.



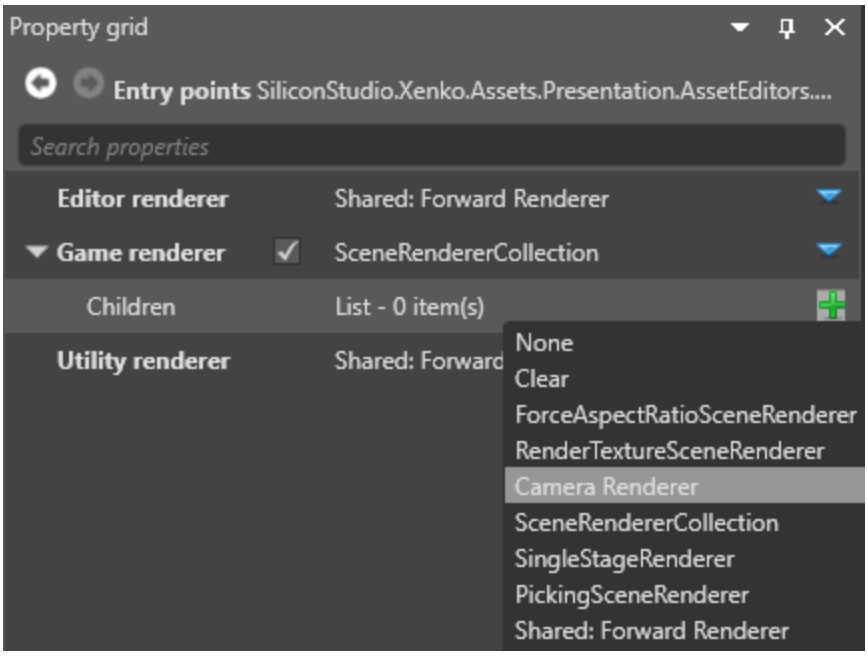
3. Click (**Replace**) and select **Scene renderer collection**.



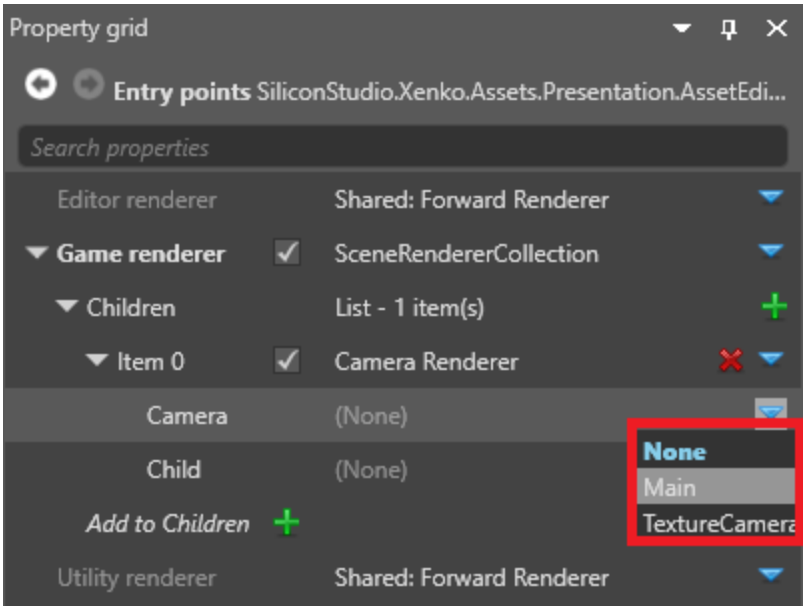
This lets you set multiple renderers for the game entry point.

1. Render the main camera

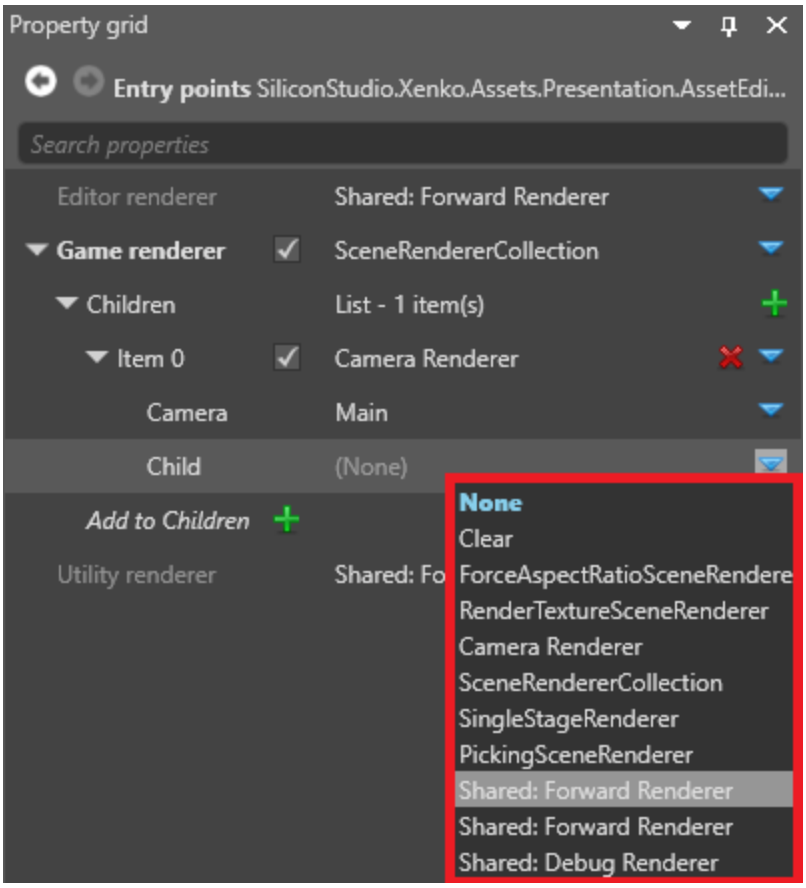
1. Under **Game renderer**, next to **Children**, click (**Add**) and select **Camera renderer**.



2. Next to **Camera**, click (**Replace**) and select your main game camera.

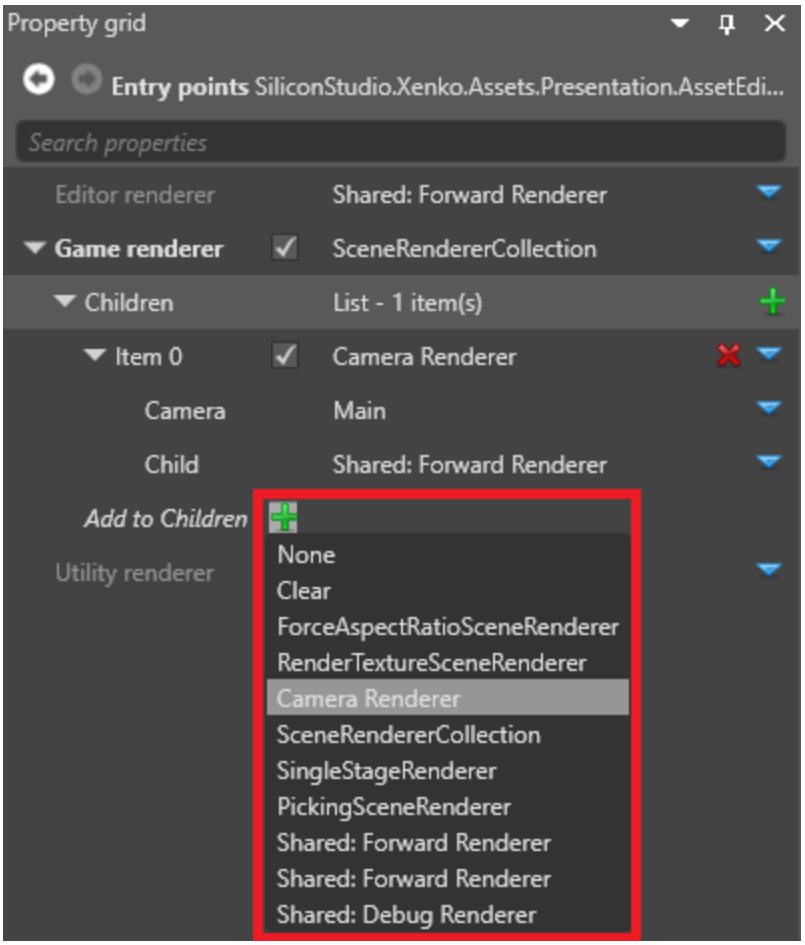


3. Next to **Child**, select the renderer for your main game camera (eg the **forward renderer**).

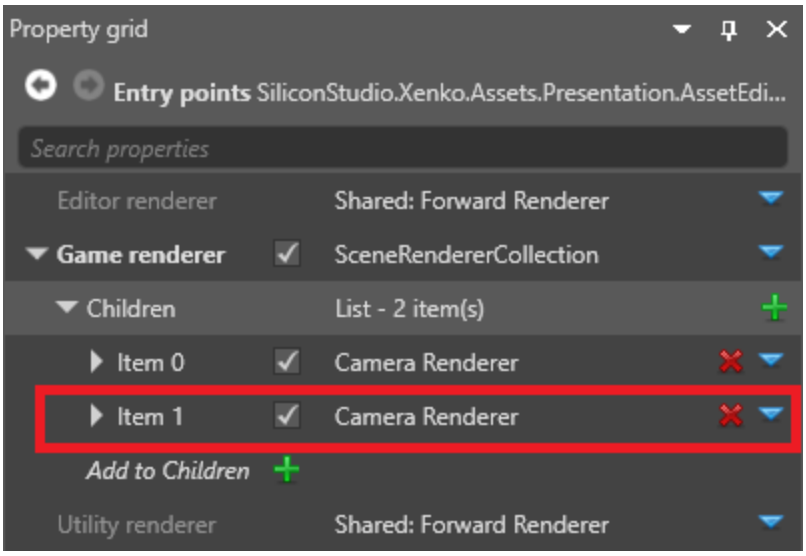


2. Render the texture

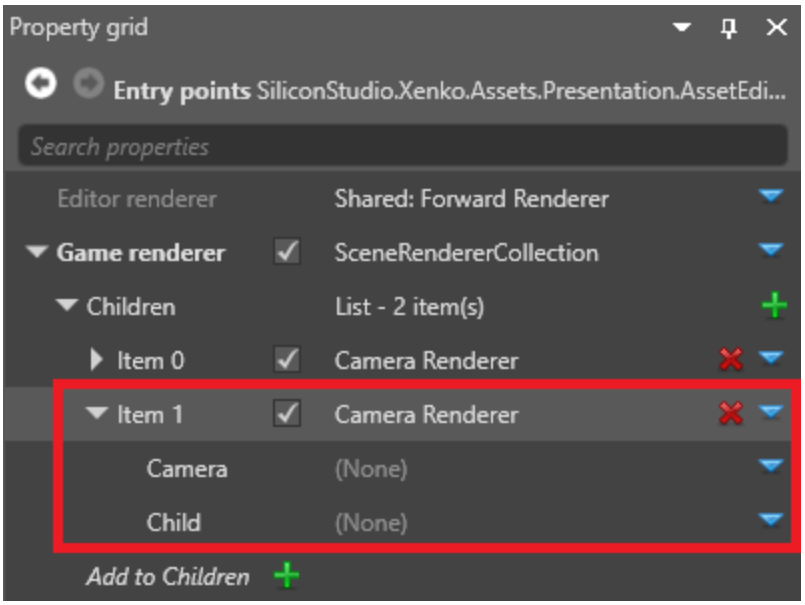
1. Under **Game renderer**, next to **Add to Children**, click (**Add**) and select **Camera renderer**.



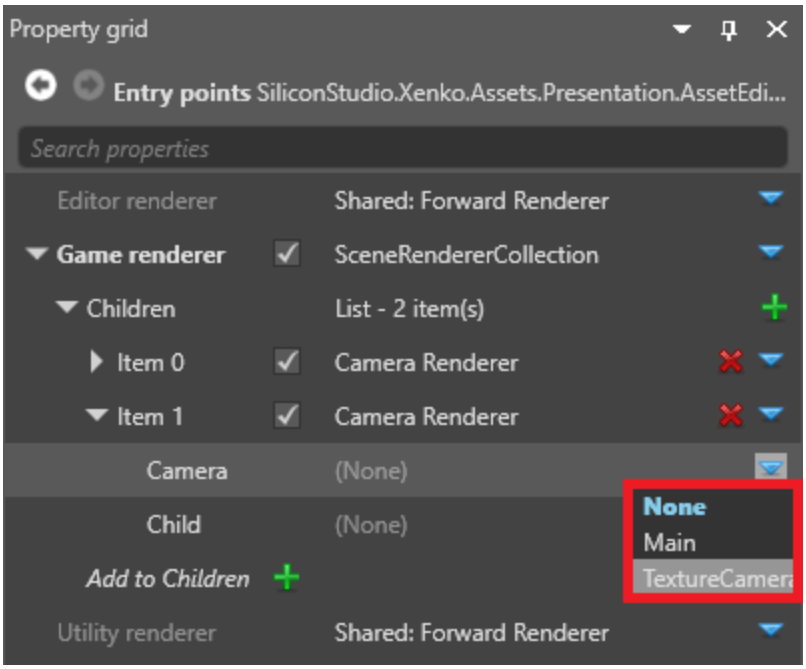
Game Studio adds a camera renderer to the list of children.



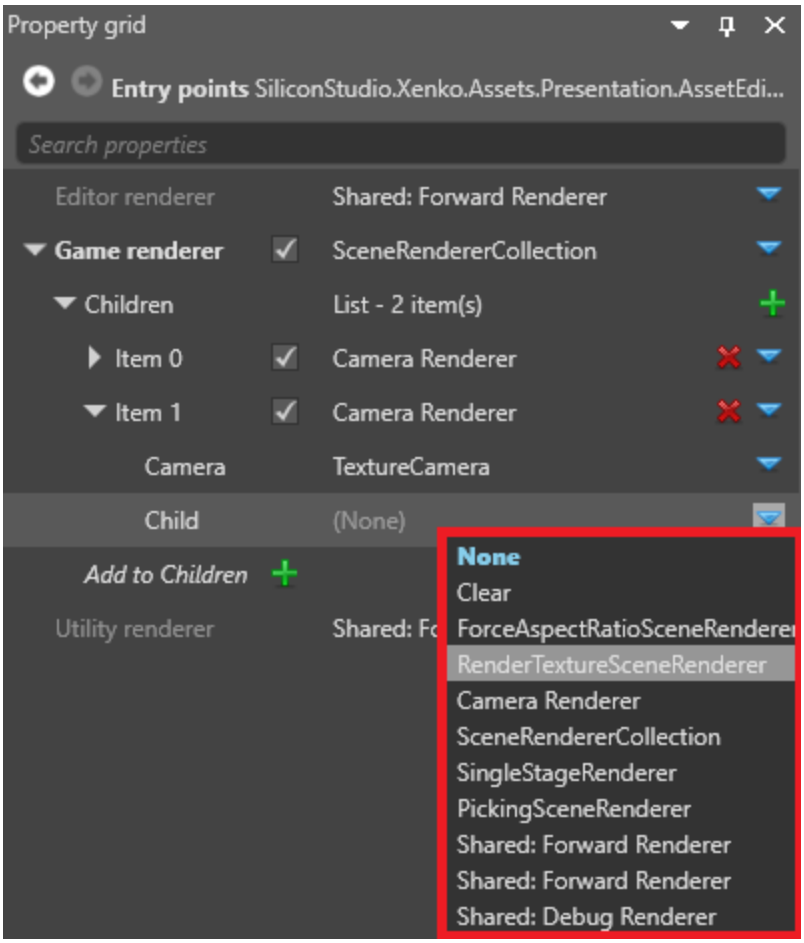
2. Expand the second **camera renderer**.



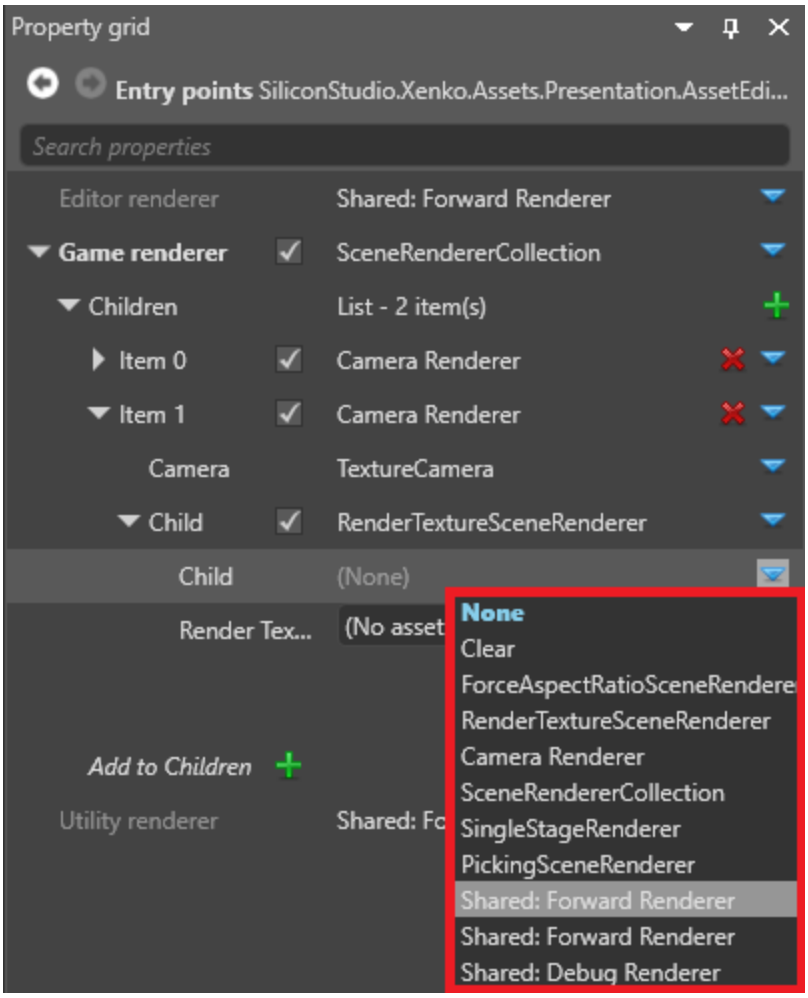
3. Next to **Camera**, click (**Replace**) and select the camera you want to render to a texture.



4. Next to **Child**, click (**Replace**) and select **RenderTextureSceneRenderer**.



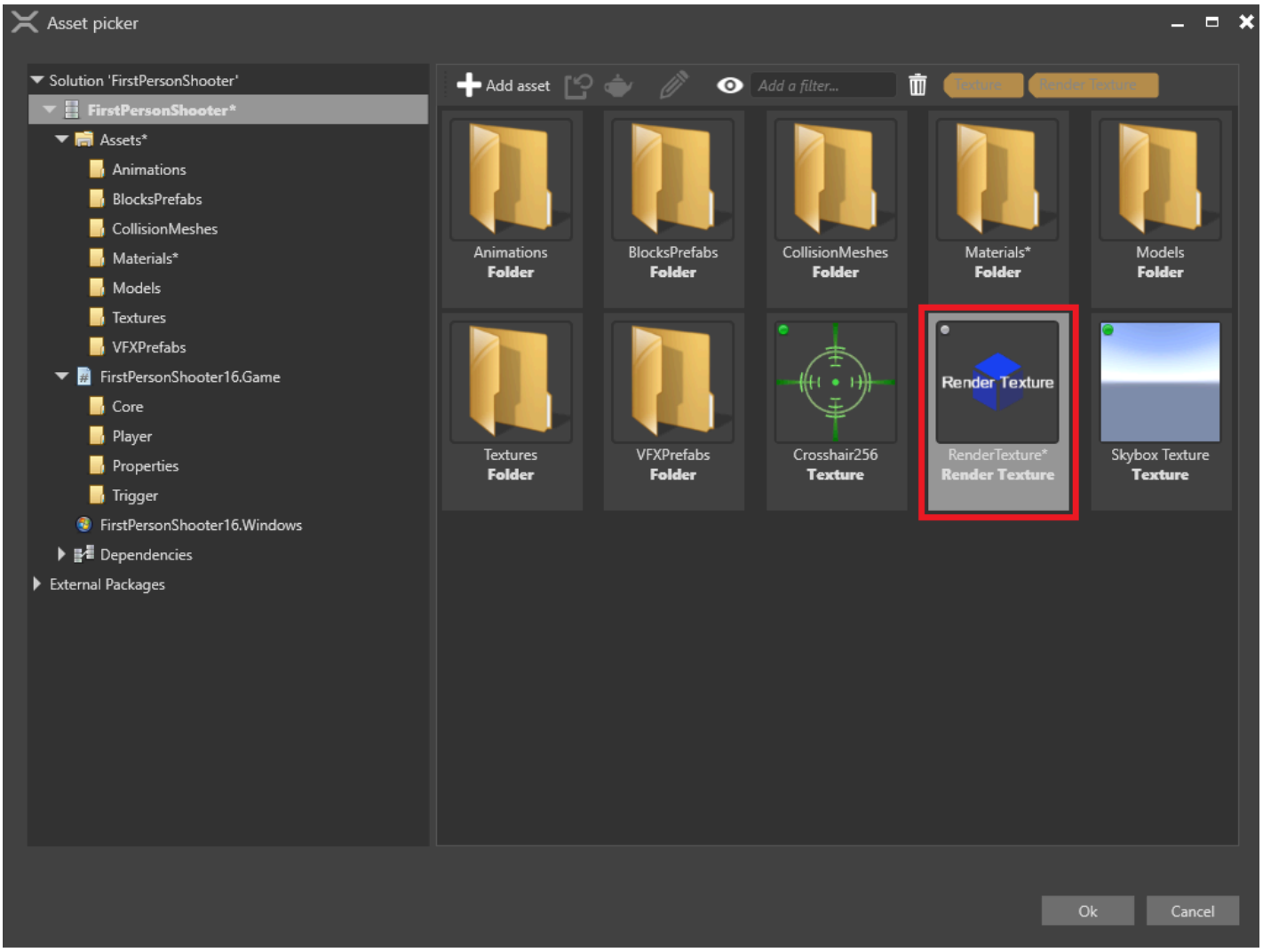
5. Under the **RenderTextureSceneRenderer**, next to **Child**, click (**Replace**) and select the renderer for your main game camera (eg the **forward renderer**).



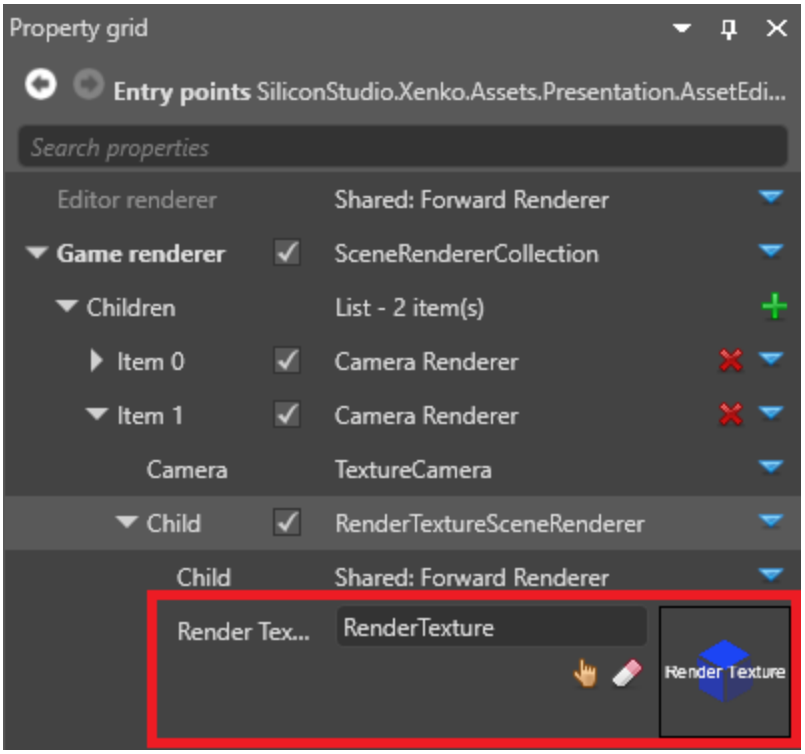
6. Next to **Render texture**, click (**Select an asset**).

   The **Select an asset** window opens.

7. Select the **render texture** and click **OK**.



Game Studio adds the render texture to the renderer.

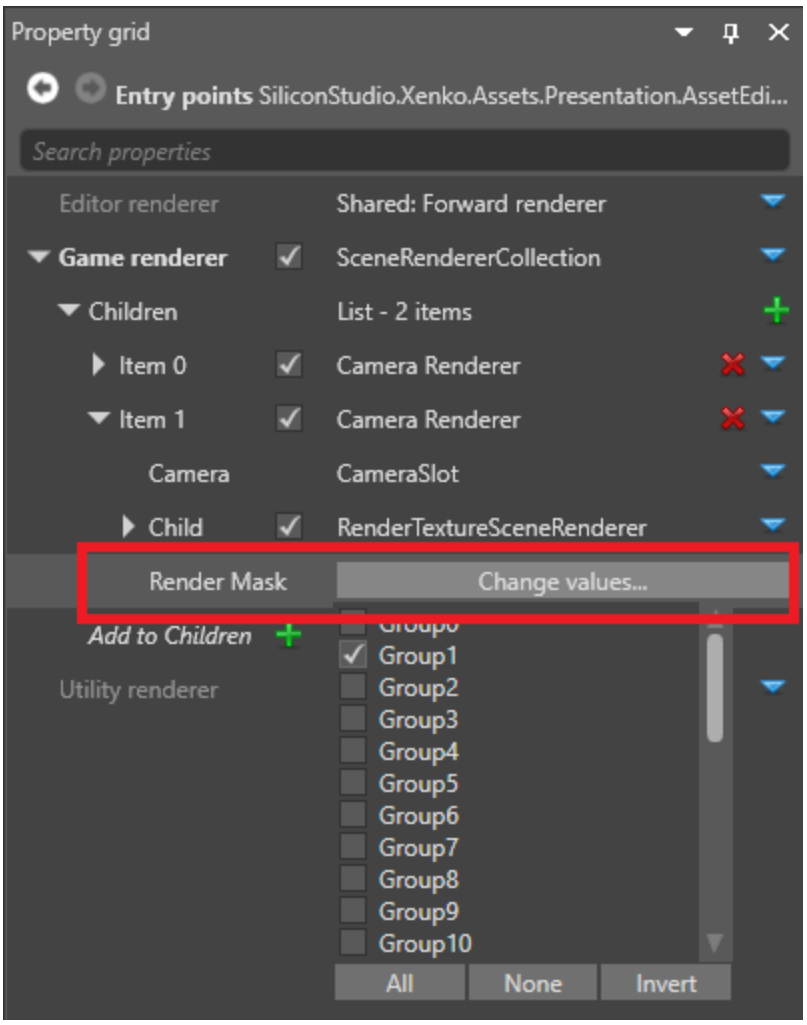


Your game is now ready to render the camera to the texture in the scene.

Set a render mask

You can use the **render mask** to filter which groups are rendered in the render texture.

Next to **Render mask**, click **Change values** and select the render groups you want the camera to render.



For more information, see Render groups and masks.

Sample

For an example of rendering to a texture in a project, see the **Animation** sample included with Stride.

See also

- Cameras
- Camera slots
- Low-level API – Textures and render textures
- Render groups and masks
- Graphics compositor

- Scene renderers

Render groups and masks

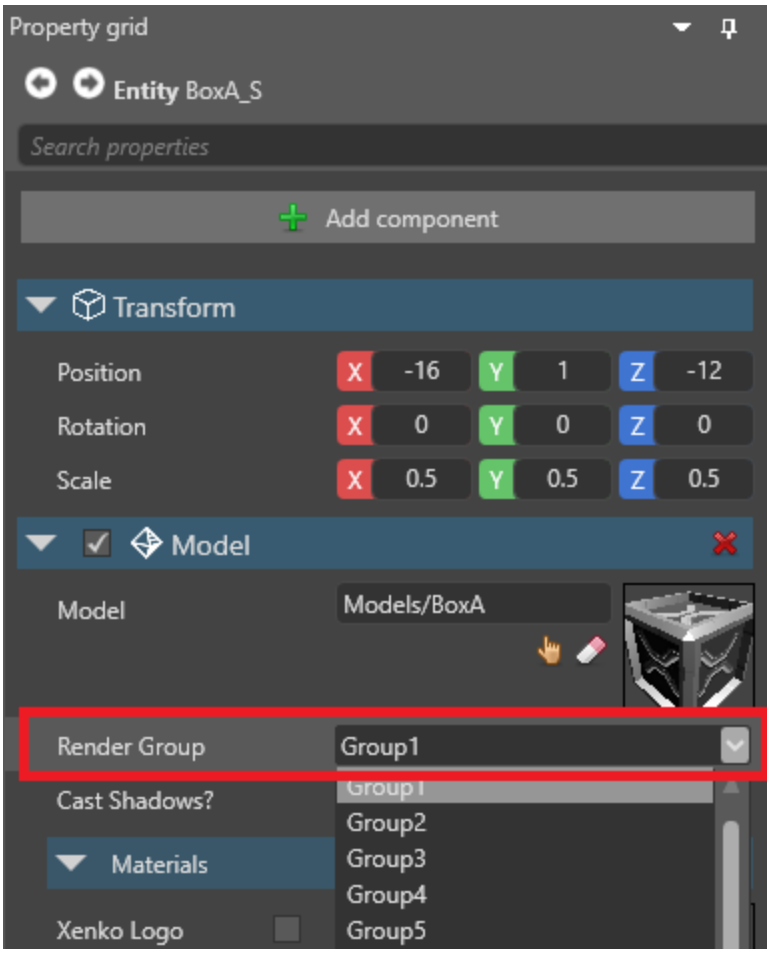
Intermediate Designer

With **render groups** and **render masks**, you can choose which parts of your scene are rendered by different cameras. For example, you can have a model be visible to Camera A but invisible to Camera B.

Set a render group

1. In the scene, select the entity with the component (such as a model or UI component) you want to add to a render group.

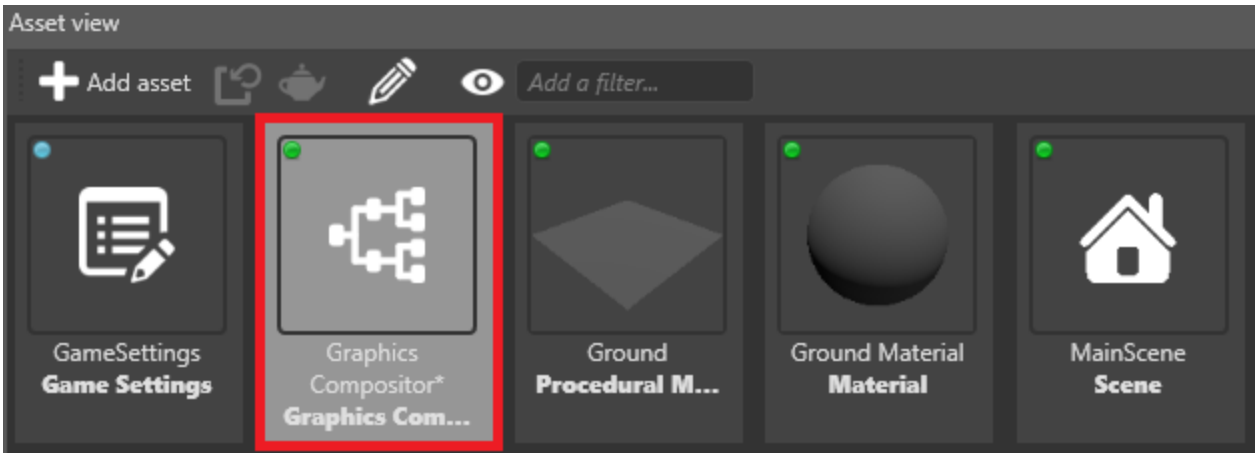
2. In the **Property Grid**, next to **Render group**, select the group you want the entity to belong to.



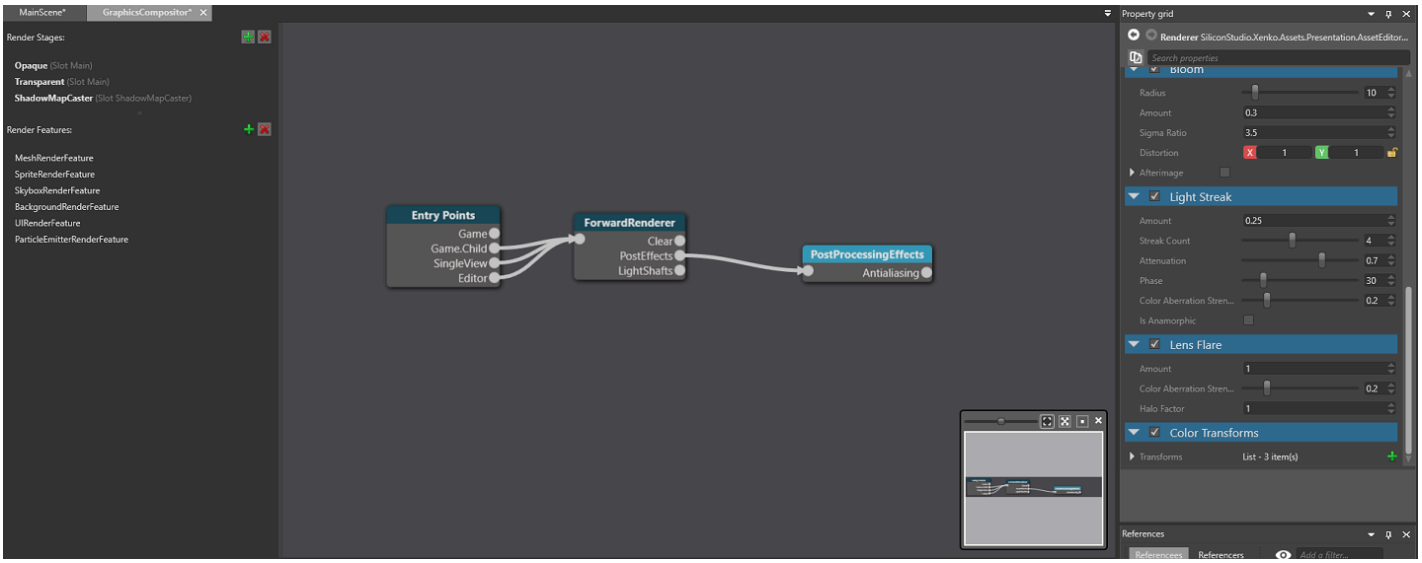
Set a render mask

The **render mask** filters which groups are rendered.

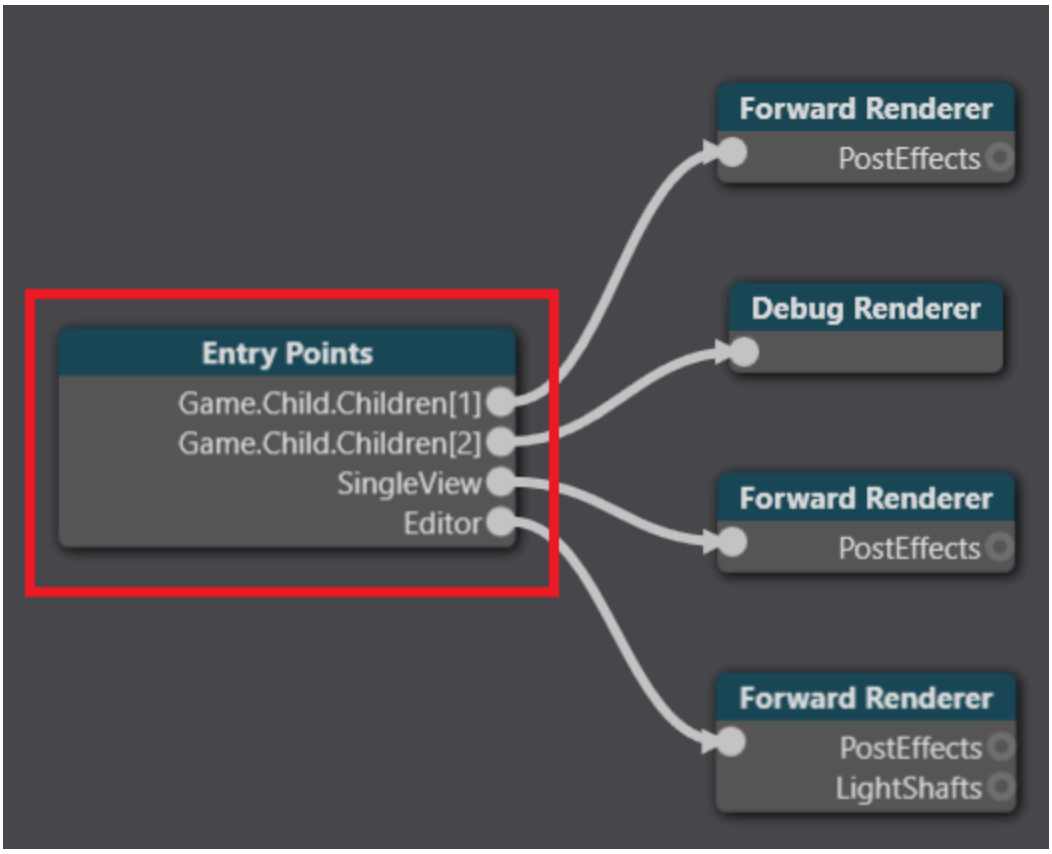
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



The Graphics Compositor Editor opens.

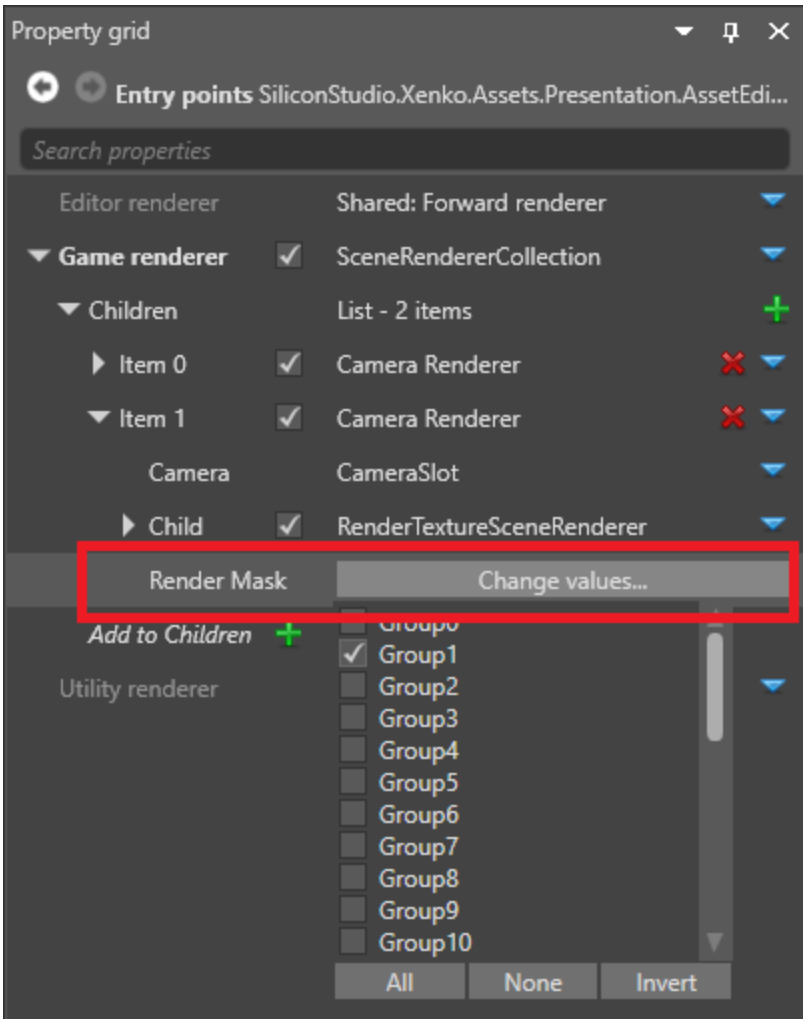


2. Select the **Entry points** node.



3. In the **Property Grid**, expand the renderer you want to render the model.

4. Next to **Render mask**, click **Change values** and select the render groups you want the camera to render.



See also

- Cameras
- Camera slots
- Graphics compositor
- Scene renderers
- Render textures

Effects and shaders

Stride uses a programmable shading pipeline. You can write custom shaders, create Effects from them, and use them for drawing. The EffectSystem class provides an easy way to load an effect.

Load an effect

Use:

```
var myEffect = EffectSystem.LoadEffect("MyEffect").WaitForResult();
```

You can then bind the effect as pipeline state.

An effect also often defines a set of parameters. To set these, you need to bind resources before drawing.

Shaders

Shaders are authored in the Stride's shading language, which is an extension of `HLSL`. They provide true composition of modular shaders via inheritance, shader mixins and automatic weaving of shader in-out attributes.

Effects

Effects in Stride use C#-like syntax to further combine shaders. They provide conditional composition of shaders to generate effect permutations.

As some platforms can't compile shaders at runtime (eg iOS, Android, etc), effect permutation files (`.sdeffectlog`) are used to enumerate all permutations ahead of time.

Target everything

Stride shaders are converted automatically to the target graphics platform — either plain HLSL for Direct3D, `GLSL` for OpenGL, or `SPIR-V` for Vulkan platforms.

For mobile platforms, shaders are optimized by a GLSL optimizer to improve performance.

> **NOTE**
>
> Converting to OpenGL Compute shaders are not supported in SDSL yet.

In this section

- Effect language
- Shading language
  - Shader classes, mixins and inheritance
  - Composition
  - Compile shaders
  - Templates
  - Shader stage input/output automatic management
- Custom shaders

Effect language

Create shaders in C#

You can create a shader at runtime with ShaderSource objects. Shaders come in three types:

- ShaderClassSource correspond to a unique shader class
- ShaderMixinSource mix several ShaderSource, set preprocessor values, define compositions
- ShaderArraySource are used for arrays of compositions

This method produces shaders at runtime. However, many platforms don't support HLSL and have no ability to compile shaders at runtime. Additionally, the approach doesn't benefit from the reusability of mixins.

Stride Effects (SDFX)

Many shaders are variations or combinations of pre-existing shaders. For example, some meshes cast shadows, others receive them, and others need skinning. To reuse code, it's a good idea to select which parts to use through conditions (eg "Skinning required"). This is often solved by "uber shaders": monolithic shaders configured by a set of preprocessor parameters.

Stride offers the same kind of control, keeping extensibility and reusability in mind. The simple code blocks defined by shader classes can be mixed together by a shader mixer. This process can use more complex logic, described in Stride Effect (*.sdfx) files.

General syntax

An .sdfx file is a small program used to generate shader permutations. It takes a set of parameters (key and value in a collection) and produces a `ShaderMixinSource` ready to be compiled.

An example .sdfx file:

```
using Stride.Effects.Data;

namespace StrideEffects
{
        params MyParameters
        {
                bool EnableSpecular = true;
        };

        effect BasicEffect
        {
                using params MaterialParameters;
                using params MyParameters;
```

```
                mixin ShaderBase;
                mixin TransformationWAndVP;
                mixin NormalVSStream;
                mixin PositionVSStream;
                mixin BRDFDiffuseBase;
                mixin BRDFSpecularBase;
                mixin LightMultiDirectionalShadingPerPixel<2>;
                mixin TransparentShading;
                mixin DiscardTransparent;

                if (MaterialParameters.AlbedoDiffuse != null)
                {
                        mixin compose DiffuseColor = ComputeBRDFDiffuseLambert;
                        mixin compose albedoDiffuse = MaterialParameters.AlbedoDiffuse;
                }

                if (MaterialParameters.AlbedoSpecular != null)
                {
                        mixin compose SpecularColor = ComputeBRDFColorSpecularBlinnPhong;
                        mixin compose albedoSpecular = MaterialParameters.AlbedoSpecular;
                }
        };
}
```

Add a mixin

To add a mixin, use `mixin <mixin_name>`.

Use parameters

The syntax is similar to C#. The following rules are added:

- When you use parameter keys, add them using `params <shader_name>`. If you don't, keys are treated as variables.
- You don't need to tell the program where to check the values behind the keys. Just use the key.

```
using params MaterialParameters;

if (MaterialParameters.AlbedoDiffuse != null)
{
        mixin MaterialParameters.AlbedoDiffuse;
}
```

The parameters behave like any variable. You can read and write their value, compare their values, and set template parameters. Since some parameters store mixins, they can be used for composition and inheritance, too.

Custom parameters

You can create your own set of parameters using a structure definition syntax.

> **NOTE**
>
> Even if they're defined in the SDFX file, don't forget the `using` statement when you want to use them.

```
params MyParameters
{
        bool EnableSpecular = true; // true is the default value
}
```

Compositions

To add a composition, assign the composition variable to your mixin with the syntax below.

```
// albedoSpecular is the name of the composition variable in the mixin
mixin compose albedoSpecular = ComputeColorTexture;

or

mixin compose albedoSpecular = MaterialParameters.AlbedoSpecular;
```

Partial effects

You can also break the code into sub-mixins to reuse elsewhere with the syntax below.

```
partial effect MyPartialEffect
{
        mixin ComputeColorMultiply;
        mixin compose color1 = ComputeColorStream;
        mixin compose color2 = ComputeColorFixed;
}

// to use it
```

```
mixin MyPartialEffect;
mixin compose myComposition = MyPartialEffect;
```

You can use the `MyPartialEffect` mixin like any other mixin in the code.

See also

- Shading language

Shading language

Stride provides a superset of the HLSL Shading language, bringing advanced and higher level language constructions, with:

- **extensibility** to allow shaders to be extended easily using object-oriented programming concepts such as classes, inheritance, and composition
- **modularity** to provide a set modular shaders each focusing on a single rendering technique, more easily manageable
- **reusability** to maximize code reuse between shaders

Stride Shading Language (SDSL) is automatically transformed to an existing shading language (HLSL, GLSL, GLSL ES).

In this section

- Shader classes, mixins, and inheritance
- Composition
- Templates
- Shader stage input/output automatic management

Shader classes, mixins and inheritance

Stride Shading Language (SDSL) is an extension of HLSL, which makes it closer to C# syntax and concepts. The language is object-oriented:

- shader classes are the foundation of the code
- shader classes contain methods and members
- shader classes can be inherited, methods can be overridden
- member types can be shader classes

SDSL uses an original way to handle multiple inheritance. Inheritance is performed through mixins, so the order of inheritance is crucial:

- the order of inheritance defines the actual implementation of a method (the last override)
- if a mixin appears several times in the inheritance, only the first occurrence is taken into account (as well as its members and methods)
- to can call the previous implementation of a method, use `base.<method name>(<arguments>)`

Keywords

SDSL uses the keywords as HLSL, and adds new ones:

- `stage`: method and member keyword. This keyword makes sure the method or member is only defined once and is the same in the compositions.
- `stream`: member keyword. The member is accessible at every stage of the shader. For more information, see Automatic shader stage input/out.
- `streams`: sort of global structure storing variables needed across several stages of the shader. For more information, see Automatic shader stage input/out.
- `override`: method keyword. If this keyword is missing, the compilation returns an error.
- `abstract`: used in front of a method declaration (without a body).
- `clone`: method keyword. When a method appears several times in the inheritance tree of a shader class, this keyword forces the creation of multiple instances of the method at each level of the inheritance instead of one. For more information, see Composition.
- `Input`: for geometry and tessellation shaders. For more information, see Shader stages.
- `Output`: for geometry and tessellation shaders. For more information, see Shader stages.
- `Input2`: for tessellation shaders. For more information, see Shader stages.
- `Constants`: for tessellation shaders. For more information, see Shader stages.

Abstract methods

Abstract methods are available in SDSL. They should be prefixed with the `abstract` keyword. You can inherit from a shader class with abstract methods without having to implement them; the compiler will

simply produce a harmless warning. However, you should implement it in your final shader to prevent a compilation error.

Annotations

Like HLSL, annotations are available in SDSL. Some of the most useful ones are:

- `[Color]` for float4 variables. The ParameterKey will have the type `Color4` instead of `Vector4`. It also specifies to Game Studio that this variable should be treated as a color, so you can edit it in Game Studio.
- `[Link(...)]` specifies which ParameterKey to use to set this value. However, an independent default key is still created.
- `[Map(...)]` specifies which ParameterKey to use to set this value. No new ParameterKey is created.
- `[RenameLink]` prevents the creation of a ParameterKey. It should be used with `[Link()]`.

Example code: annotations

```
shader BaseShader
{
        [Color] float4 myColor;

        [Link("ProjectKeys.MyTextureKey")]
        [RenameLink]
        Texture2D texture;

        [Map("Texturing.Texture0")] Texture2D defaultTexture;
};
```

Example code: inheritance

```
shader BaseInterface
{
        abstract float Compute();
};

shader BaseShader : BaseInterface
{
        float Compute()
        {
                return 1.0f;
        }
};

shader ShaderA : BaseShader
```

```
{
        override void Compute()
        {
                return 2.0f;
        }
};

shader ShaderB : BaseShader
{
        override void Compute()
        {
                float prevValue = base.Compute();
                return (5.0f + prevValue);
        }
};
```

Example code: the importance of inheritance order

Notice what happens when we change the inheritance order between `ShaderA` and `ShaderB`.

```
shader MixAB : ShaderA, ShaderB
{
};

shader MixBA : ShaderB, ShaderA
{
};

// Resulting code (representation)

shader MixAB : BaseInterface, BaseShader, ShaderA, ShaderB
{
        float Compute()
        {
                // code from BaseShader
                float v0 = 1.0f;

                // code from ShaderA
                float v1 = 2.0f;

                // code from ShaderB
                float prevValue = v1;
                float v2 = 5.0f + prevValue;

                return v2; // = 7.0f
```

```
        }
};

shader MixBA : BaseInterface, BaseShader, ShaderA, ShaderB
{
        float Compute()
        {
                // code from BaseShader
                float v0 = 1.0f;

                // code from ShaderB
                float prevValue = v0;
                float v1 = 5.0f + prevValue;

                // code from ShaderA
                float v2 = 2.0f;

                return v2; // = 2.0f
        }
};
```

Static calls

You can also use a variable or call a method from a shader without having to inherit from it. To do this, use `<shader_name>.<variable or method_name>`. It behaves the same way as a static call.

Note that if you statically call a method that uses shader class variables, the shader won't compile. This is a convenient way to only use a part of a shader, but this isn't an optimization. The shader compiler already automatically removes any unnecessary variables.

Code example: static calls

```
shader StaticClass
{
        float StaticValue;
        float StaticMethod(float a)
        {
                return 2.0f * a;
        }

        // this method uses a
        float NonStaticMethod()
        {
                return 2.0f * StaticValue;
        }
```

```
};

// this shader class is fine
shader CorrectStaticCallClass
{
        float Compute()
        {
                return StaticClass.StaticValue * StaticMethod(5.0f);
        }
};

// this shader class won't compile since the call is not static
shader IncorrectStaticCallClass
{
        float Compute()
        {
                return StaticClass.NonStaticMethod();
        }
};

// one way to fix this
shader IncorrectStaticCallClassFixed : StaticClass
{
        float Compute()
        {
                return NonStaticMethod();
        }
};
```

See also

- Effect language
- Shading language index
  - Composition
  - Templates
  - Shader stage input/output automatic management
  - Shader stages

Composition

Beginner Programmer

In addition to the inheritance system, SDSL introduces the concept of **composition**. A composition is a member whose type is another shader class. It's defined the same way as variables.

You can compose with an instance of the desired shader class or an instance of a shader class that inherits from the desired one.

Example code

```
shader CompositionBase
{
        float4 Compute()
        {
                return float4(0.0);
        }
};

shader CompositionShaderA : CompositionBase
{
        float4 myColor;

        override float4 Compute()
        {
                return myColor;
        }
};

shader CompositionShaderB : CompositionBase
{
        float4 myColor;

        override float4 Compute()
        {
                return 0.5 * myColor;
        }
};

shader BaseShader
{
        CompositionBase Comp0;
        CompositionBase Comp1;
```

```
        float4 GetColor()
        {
                return Comp0.Compute() + Comp1.Compute();
        }
};
```

The compositions are compiled in their own context, meaning that the non-stage variables are only accessible within the composition. It's also possible to have compositions inside compositions.

Example code: access root context

If you want to access the root compilation context, you can use the following format:

```
shader CompositionShaderC : CompositionBase
{
        BaseShader rootShader = stage;

        float4 GetColor()
        {
                return rootShader.GetColor();
        }
};
```

This is error-prone, since `CompositionShaderC` expects `BaseShader` to be available in the root context.

Example code: array of compositions

You can also create an array of compositions the same way you use an array of values. Since there's no way to know beforehand how many compositions there are, you should iterate using a `foreach` statement.

```
shader BaseShaderArray
{
        CompositionBase Comps[];

        float4 GetColor()
        {
                float4 resultColor = float4(0.0);

                foreach (var comp in Comps)
                {
                        resultColor += comp.Compute();
                }
```

```
                return resultColor;
        }
};
```

Example code: stage behavior

The behavior of the `stage` keyword is straightforward: only one instance of the variable or method is produced.

```
shader BaseShader
{
        stage float BaseStageValue;
        float NonStageValue;
};

shader TestShader : BaseShader
{
        BaseShader comp0;
        BaseShader comp1;
};

// resulting shader (representation)
shader TestShader
{
        float BaseStageValue;
        float NonStageValue;
        float comp0_NonStageValue;
        float comp1_NonStageValue;
};
```

Example code: stage member behavior

```
shader BaseShader
{
        stage float BaseStageMethod()
        {
                return 1.0;
        }

        float NonStageMethod()
        {
                return 2.0;
        }
};
```

```
shader TestShader : BaseShader
{
        BaseShader comp0;
        BaseShader comp1;
};

// resulting shader (representation)
shader TestClass
{
        float BaseStageMethod()
        {
                return 1.0;
        }

        float NonStageMethod()
        {
                return 2.0;
        }
        float comp0_NonStageMethod()
        {
                return 2.0;
        }
        float comp1_NonStageMethod()
        {
                return 2.0;
        }
};
```

Keep in mind that even in composition, you can call for base methods, override them, and so on. Overriding happens in the same order as the compositions.

This behavior is useful when you need a value in multiple composition but you only need to compute it once (eg the normal in view space).

Clone behavior

The `clone` keyword has a less trivial behavior. It prevents the `stage` keyword to produce a unique method.

```
shader BaseShader
{
        stage float BaseStageMethod()
        {
                return 1.0;
```

```
        }

        stage float BaseStageMethodNotCloned()
        {
                return 1.0;
        }
};

shader CompShader : BaseShader
{
        override clone float BaseStageMethod()
        {
                return 1.0 + base.BaseStageMethod();
        }

        override float BaseStageMethodNotCloned()
        {
                return 1.0f + base.BaseStageMethodNotCloned();
        }
};

shader TestShader : BaseShader
{
        CompShader comp0;
        CompShadercomp1;
};

// resulting shader (representation)
shader TestShader
{
        // cloned method
        float base_BaseStageMethod()
        {
                return 1.0;
        }

        float comp0_BaseStageMethod()
        {
                return 1.0 + base_BaseStageMethod();
        }

        float BaseStageMethod() // in fact comp1_BaseStageMethod
        {
                return 1.0 + comp0_BaseStageMethod; // 3.0f
        }
```

```
        // not cloned method
        float base_BaseStageMethodNotCloned()
        {
                return 1.0f;
        }

        float BaseStageMethodNotCloned()
        {
                return 1.0f + base_BaseStageMethodNotCloned(); // 2.0f
        }
};
```

This behavior is useful when you want to repeat a simple function but with different parameters (eg adding color on top of another).

See also

- Effect language
- Shading language index
  - Shader classes, mixins, and inheritance
  - Templates
  - Shader stage input/output automatic management
  - Shader stages

Templates

Shader templating is available in SDSL. Unlike many templating systems, sdsl requires strong typed templates. The available types are:

- value types from HLSL (float, int, float2, float3, float4)
- 2D textures
- sampler states
- semantics (used to replace semantics on variables)
- link types (used to replace link annotations)

An instantiated shader behaves the same way as any other shader. The value, texture and sampler template parameters are accessible like any other variable. However, it's impossible to modify their value; attempting to do so results in a compilation error. If a template variable is incorrectly used (eg using a sampler as a semantic), it should result in a compilation error. However, the behavior is officially unknown.

Code: Templating

```
shader TemplateShader<float speed, Texture2D myTexture, SamplerState mySampler, Semantic
mySemantic, LinkType myLink>
{
        [Color]
        [Link("myLink")]
        float4 myColor;

        stream float2 texcoord : mySemantic;

        float4 GetValue()
        {
                return speed * myColor * myTexture.Sample(mySampler, streams.texcoord);
        }
};

// To instantiate the shader, use:
TemplateShader<1.0f, Texturing.Texture0, Texturing.Sampler0, TEXCOORD0, MyColorLink>
```

See also

- Effect language
- Shading language index
  - Shader classes, mixins, and inheritance
  - Composition

- Shader stage input/output automatic management

Automatic shader stage input/output

Advanced Programmer

When you write a HLSL shader, you have to precisely define your vertex attributes and carefully pass them across the stage of your final shader.

Here's an example of a simple HLSL shader that uses the color from the vertex.

```
struct VS_IN
{
        float4 pos : POSITION;
        float4 col : COLOR;
};

struct PS_IN
{
        float4 pos : SV_POSITION;
        float4 col : COLOR;
};

PS_IN VS( VS_IN input )
{
        PS_IN output = (PS_IN)0;

        output.pos = input.pos;
        output.col = input.col;

        return output;
}

float4 PS( PS_IN input ) : SV_Target
{
        return input.col;
}

technique10 Render
{
        pass P0
        {
                SetGeometryShader( 0 );
                SetVertexShader( CompileShader( vs_4_0, VS() ) );
                SetPixelShader( CompileShader( ps_4_0, PS() ) );
        }
}
```

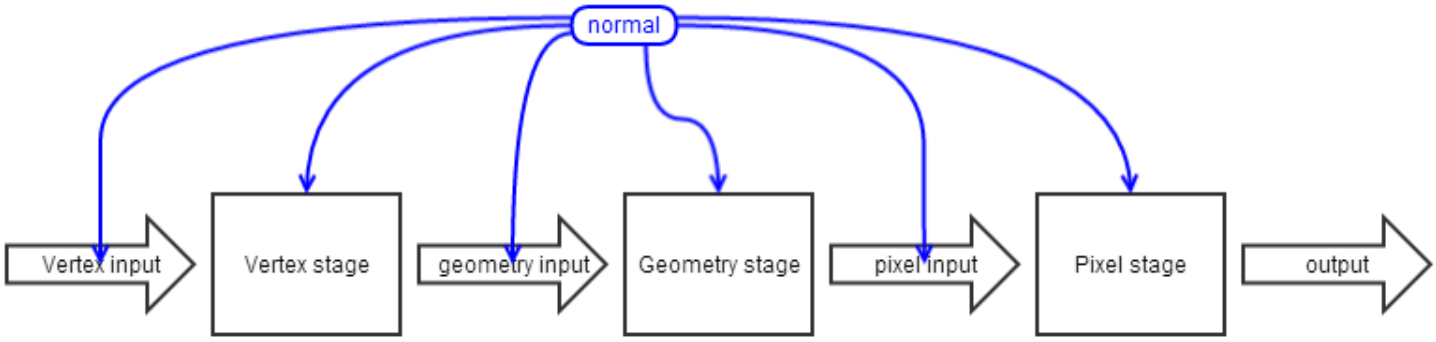
Imagine you want to add a normal to your model and modify the resulting color accordingly. You have to modify the code that computes the color and adjust the intermediate structures to pass the attribute from the vertex to the pixel shader. You also have to be careful of the semantics you use.

Code: Modified HLSL shader

```
struct VS_IN
{
        float4 pos : POSITION;
        float4 col : COLOR;
        float3 normal : NORMAL;
};

struct PS_IN
{
        float4 pos : SV_POSITION;
        float4 col : COLOR;
        float3 normal : TEXCOORD0;
};

PS_IN VS( VS_IN input )
{
        PS_IN output = (PS_IN)0;

        output.pos = input.pos;
        output.col = input.col;
        output.normal = input.normal;

        return output;
}

float4 PS( PS_IN input ) : SV_Target
{
        return input.col * max(input.normal.z, 0.0);
}

technique10 Render
{
        pass P0
        {
                SetGeometryShader( 0 );
                SetVertexShader( CompileShader( vs_4_0, VS() ) );
                SetPixelShader( CompileShader( ps_4_0, PS() ) );
        }
}
```

This example is simple. Real projects have many more shaders, so a single change might mean rewriting lots of shaders, structures, and so on.

Schematically, adding a new attribute requires you to update all the stages and intermediate structures from the vertex input to the last stage you want to use the attribute in.



SDSL

SDSL has a convenient way to pass parameters across the different stages of your shader. The stream variables are:

- variables
- defined like any shader member, with the stream keyword
- used with the stream prefix (omitting it results in a compilation error). When the stream is ambiguous (same name), you should provide the shader name in front of the variable (ie `streams.<my_shader>.<my_variable>`)

Streams regroup the concepts of attributes, varyings and outputs in a single concept.

- An attribute is a stream read in a vertex shader before being written to.
- A varying is a stream present across shader stages.
- An output is a stream assigned before being read.

Think of streams as global objects that you can access everywhere without specifying as a parameter of your functions.

> **NOTE**
>
> You don't have to create a semantic for these variables; the compiler creates them automatically. However, keep in mind that **the variables sharing the same semantic will be merged in the final shader**. This behavior can be useful when you want to use a stream variable locally without inheriting from the shader where it was declared.

After you declare a stream, you can access it at any stage of your shader. The shader compiler takes care of everything. The variables just have to be visible from the calling code (ie in the inheritance hierarchy) like any other variable.

Code: Stream definition and use:

```
shader BaseShader
{
        stream float3 myVar;

        float3 Compute()
        {
                return streams.myVar;
        }
};
```

Code: Stream specification

```
shader StreamShader
{
        stream float3 myVar;
};

shader ShaderA : BaseShader, StreamShader
{
        float3 Test()
        {
                return streams.BaseShader.myVar + streams.StreamShader.myVar;
        }
}
```

Example of SDSL shader

Let's look at the same HLSL shader as the first example but in SDSL.

Code: Same shader in SDSL

```
shader MyShader : ShaderBase
{
        stream float4 pos : POSITION;
        stream float4 col : COLOR;

        override void VSMain()
        {
```

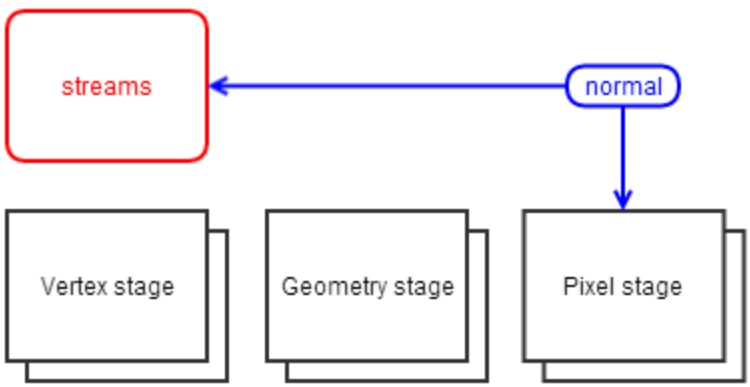
```
                streams.ShadingPosition = streams.pos;
        }

        override void PSMain()
        {
                streams.ColorTarget = streams.col;
        }
};
```

Now let's add the normal computation.

Code: Modified shader in SDSL

```
shader MyShader : ShaderBase
{
        stream float4 pos : POSITION;
        stream float4 col : COLOR;
        stream float3 normal : NORMAL;

        override void VSMain()
        {
                streams.ShadingPosition = streams.pos;
        }

        override void PSMain()
        {
                streams.ColorTarget = streams.col * max(streams.normal.z, 0.0);
        }
};
```

In SDSL, adding a new attribute is as simple as adding it to the pool of streams and using it where you want.



See also

- Effect language
- Shading language index
  - Shader classes, mixins and inheritance
  - Composition
  - Templates
  - Shader stages

Shader stages

The function for each stage has a predefined name, so we recommend you don't change it.

- `HSMain` for hull shader
- `HSConstantMain` for patch constant function
- `DSMain` for domain shader
- `VSMain` for vertex shader (takes no arguments)
- `GSMain` for geometry shader
- `PSMain` for pixel shader (takes no arguments)
- `CSMain` for compute shader (takes no arguments)

These are all void methods.

The geometry and tessellation shaders need some kind of predefined structure as input and output. However, since Stride shaders are generic, it's impossible to know beforehand what the structure will be. As a result, these shaders use `Input` and `Output` structures that are automatically generated to fit the final shader.

Vertex shader

A vertex shader has to set the variable with the semantic `SV_Position`. In `ShaderBase`, it's `ShadingPosition`.

For example:

```
override stage void VSMain()
{
        ...
        streams.ShadingPosition = ...;
        ...
}
```

Pixel shader

A pixel shader has to set the variable with the semantic `SV_Target`. In `ShaderBase`, it is `ColorTarget`.

For example:

```
override stage void PSMain()
{
        ...
        streams.ColorTarget = ...;
```

```
        ...
}
```

Geometry shader

An example of geometry shader:

```
[maxvertexcount(1)]
void GSMain(triangle Input input[3], inout PointStream<Output> pointStream)
{
        ...
        // fill the streams object
        streams = input[0];
        ...

        // always append streams
        pointStream.Append(streams);
        ...
}
```

`Input` can be used in the method body. It behaves like the streams object and contains the same members.

`Output` is only used in the declaration of the method. You should append the streams object to your geometry shader output stream.

Tessellation shader

An example of a tessellation shader:

```
[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("HSConstantMain")]
[maxtessfactor(48.0)]
void HSMain(InputPatch<Input, 3> input, out Output output, uint uCPID :
SV_OutputControlPointID)
{
        ...
        output = streams;
}

void HSConstantMain(InputPatch<Input, 3> input, const OutputPatch<Input2, 3> output, out
```

```
Constants constants)
{
        ...
        output = streams;
        ...
}

[domain("tri")]
void DSMain(const OutputPatch<Input, 3> input, out Output output, in Constants constants,
float3 f3BarycentricCoords : SV_DomainLocation)
{
        ...
        output = streams;
        ...
}
```

`Input` and `Input2` both behave like streams.

> **NOTE**
>
> Don't forget to assign `output` to `streams` at the end of your stage.

Compute shader

An example of a compute shader:

```
[numthreads(2, 3, 5)]
void CSMain()
{
        ...
}
```

You can inherit from `ComputeShaderBase` and override the `Compute` method.

See also

- Effect language
- Shading language index
  - Shader classes, mixins, and inheritance
  - Composition
  - Templates
  - Shader stage input/output automatic management

Custom shaders

Intermediate Programmer

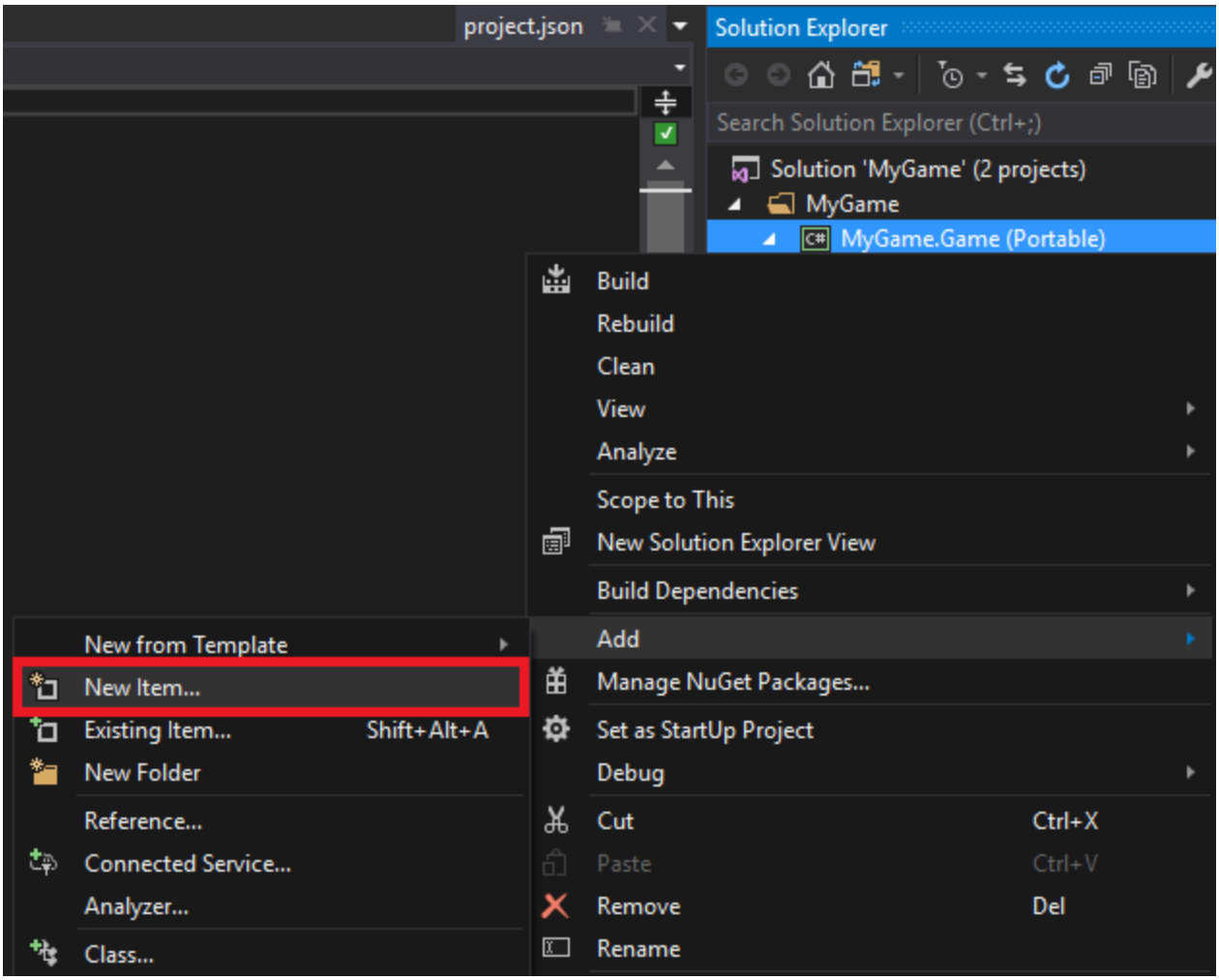
You can write your own shaders in Visual Studio and use them in material attributes. For example, you can write a shader to add textures to materials based on the objects' world positions, or generate noise and use it to randomize material properties.

As shaders are text files, you can add comments, enable and disable lines of code, and edit them like any other code file. This makes them easy to maintain and iterate.

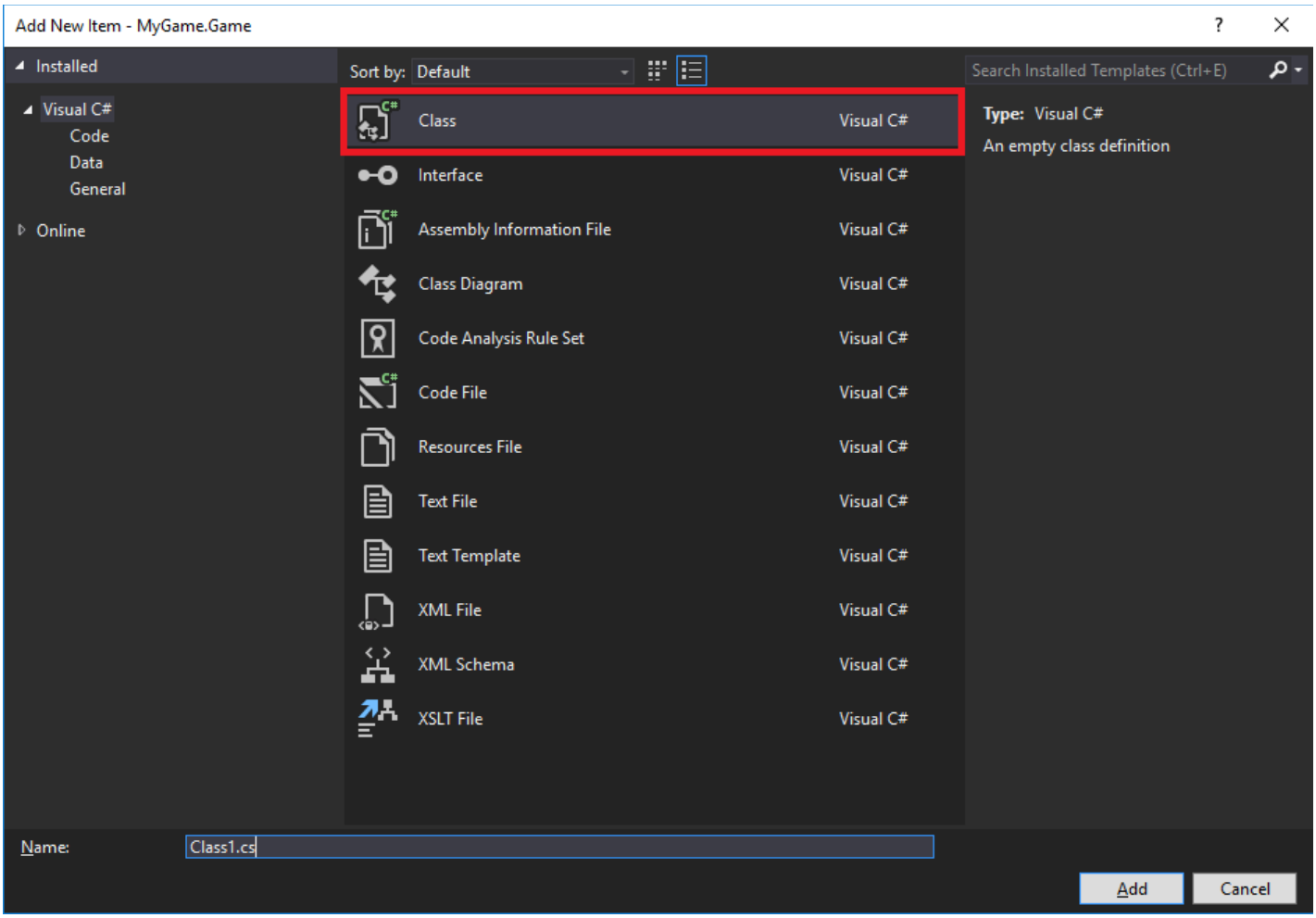
You can also use custom shaders to create custom post effects. For more information, see Custom color transforms.

Create a shader

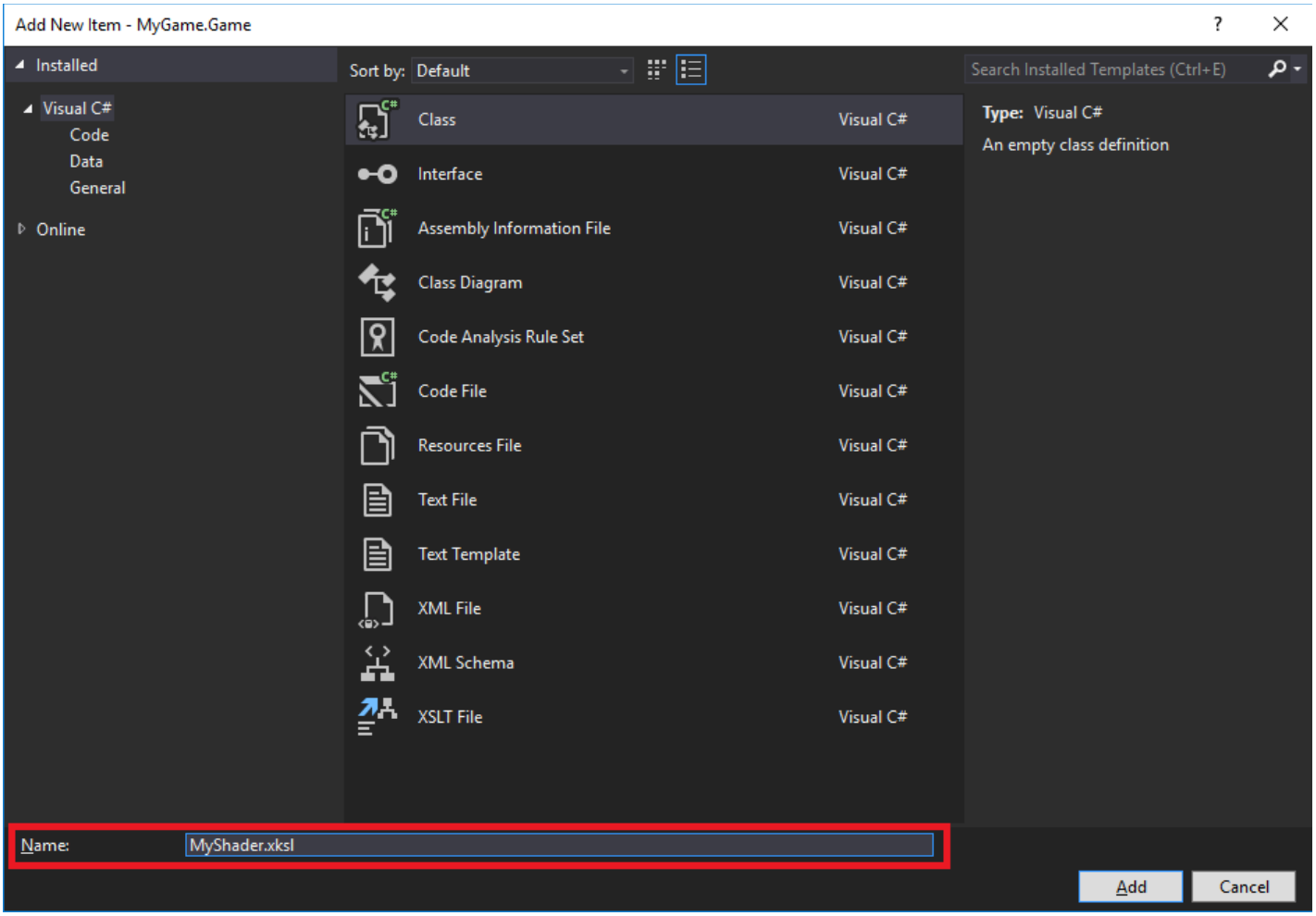
1. Make sure you have the Stride Visual Studio extension installed. This is necessary to convert the shader files from SDSL (Stride shading language) to `.cs` files.
2. In Game Studio, in the toolbar, click (**Open in IDE**) to open your project in Visual Studio.
3. In the Visual Studio **Solution Explorer**, right-click the project (eg *MyGame.Game*) and select **Add > New item**.



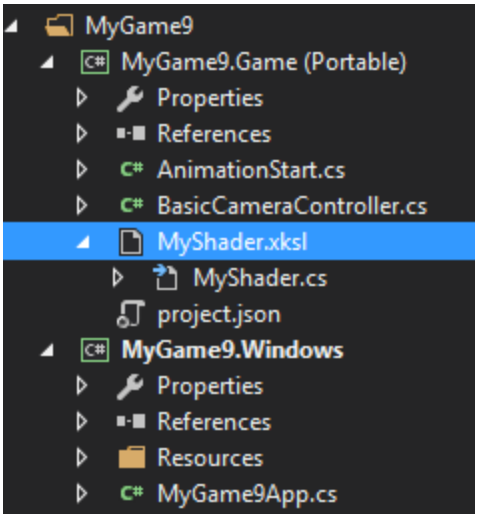
4. Select **Class**.



5. In the **Name** field, specify a name with the extension **.sdsl** (eg *MyShader.sdsl*), and click **Add**.



The Stride Visual Studio extension automatically generates a `.cs` file from the `.sdsl` file. The Solution Explorer lists it as a child of the `.sdsl` file.



6. Open the `.sdsl` file, remove the existing lines, and write your shader.

   Shaders are written in Stride Shading Language (SDSL), which is based on HLSL. For more information, see Shading language.

   For example, this shader creates a green color (`RGBA 0;1;0;1`):

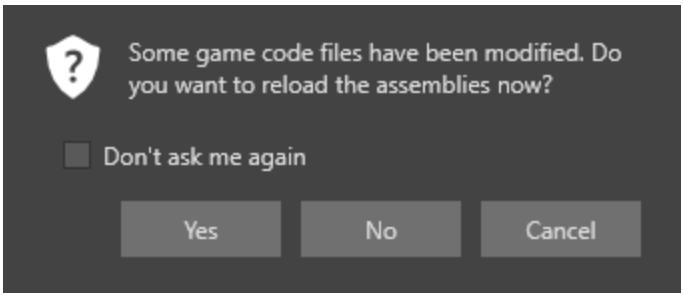
```
namespace MyGame
{
    shader MyShader : ComputeColor
    {
        override float4 Compute()
        {
            return float4(0, 1, 0, 1);
        }
    };
}
```

> **NOTE**
>
> Make sure the shader name in the file (eg `MyShader` above) is the same as the filename.

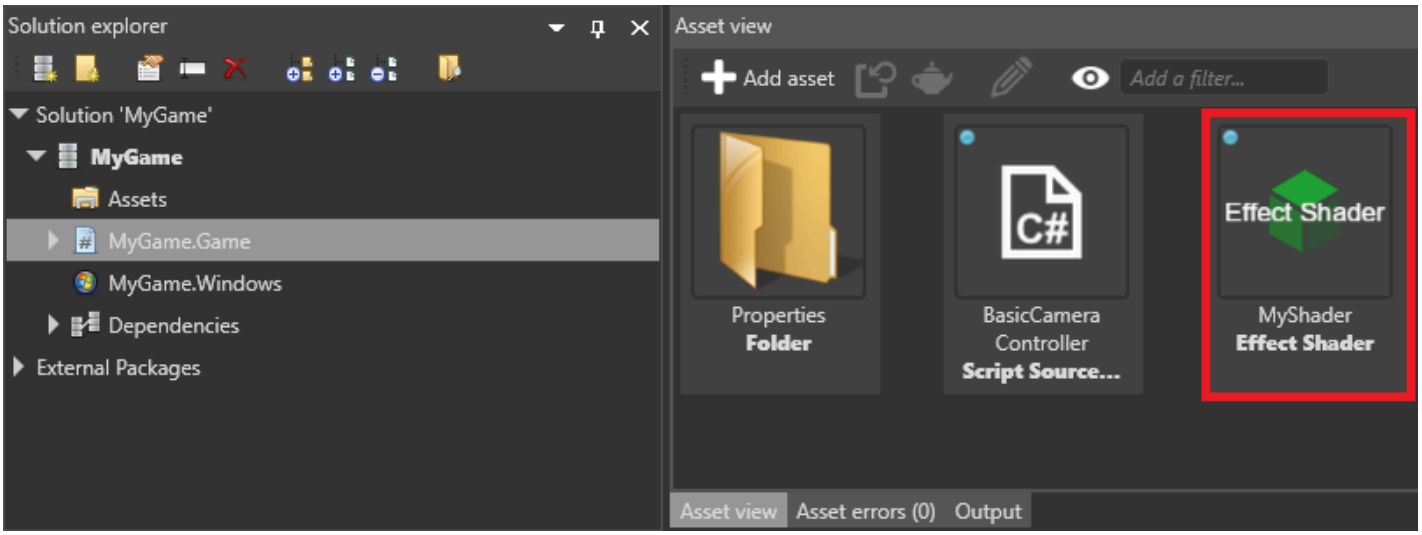
> **NOTE**
>
> To be accessible from the Game Studio Property Grid, the shader must inherit from `ComputeColor`. As `ComputeColor` always returns a float4 value, properties that take float values (eg metalness and gloss maps) use the first component (the red component) of the value returned by `ComputeColor`.

7. Save all the files in the solution (**File > Save All**).

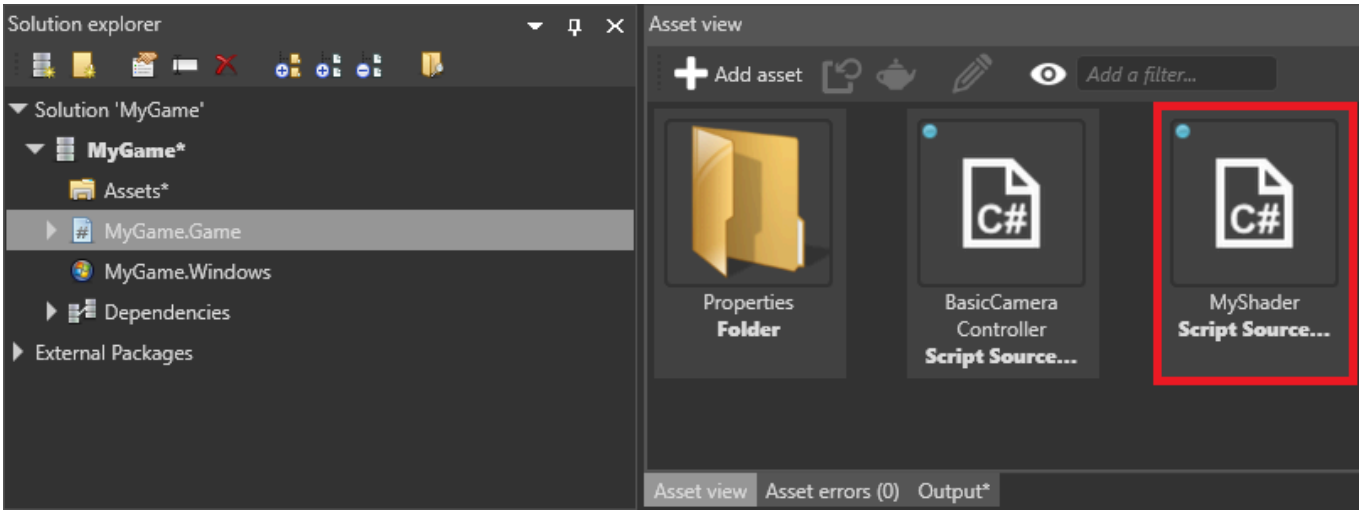
8. In Game Studio, reload the assemblies.



The **Asset View** lists the shader in the same directory as your scripts (eg **MyGame.Game**).



> **NOTE**
>
> In some situations, Game Studio incorrectly identifies the shader as a script, as in the screenshot below:

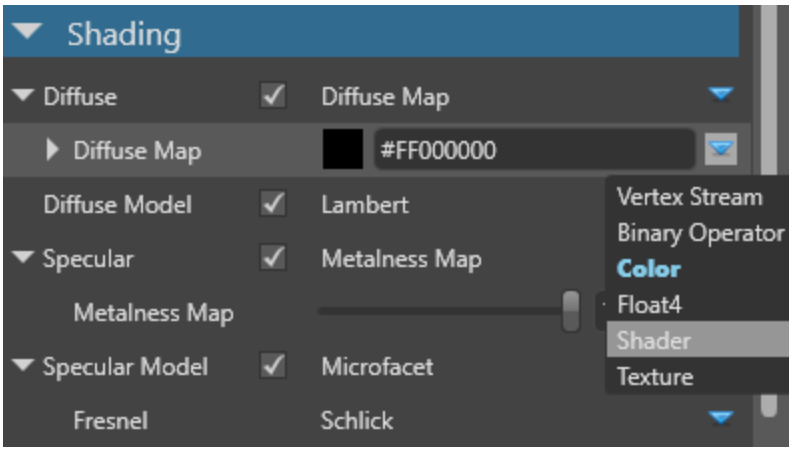


If this happens, restart Game Studio (**File > Reload project**).

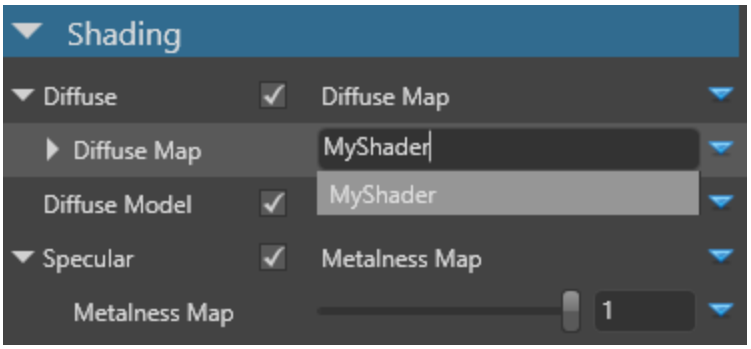
Use a custom shader

You can use custom shaders in any material attribute.

1. In the **Asset View**, select the material you want to use the shader in.
2. In the **Property Grid**, next to the property you want to control with the shader, click (**Change**) and select **Shader**.



3. In the field, type the name of your shader (eg *MyShader*).



The property uses the shader you specify.

> **TIP**
>
> When you make a change to the `.sdsl` file in Visual Studio and save it, Game Studio automatically updates the project with your changes. If this doesn't happen, restart Game Studio (**File > Reload project**).

> **NOTE**
>
> If you delete a shader from the project assets, to prevent errors, make sure you also remove it from the properties of materials that use it.

Arguments and parameters

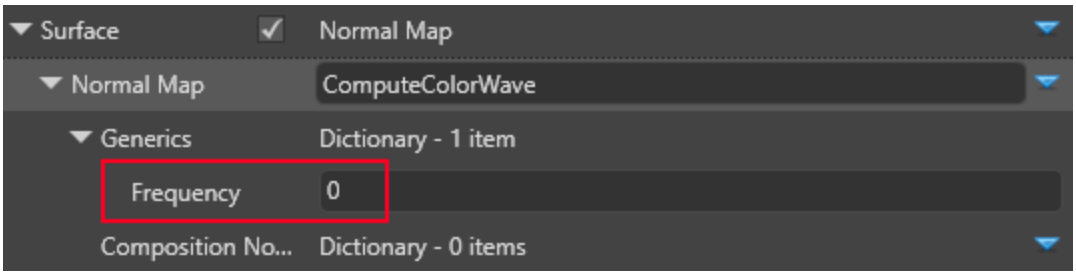
Template arguments

Template arguments (generics) don't change at runtime. However, different materials can use different instances of the shader with different values.

When the shaders are compiled, Stride substitutes template arguments with the value you set in the Property Grid.

For example, the code below defines and uses the template argument `Frequency`:

```
shader ComputeColorWave<float Frequency> : ComputeColor, Texturing
{
    override float4 Compute()
   {
        return sin((Global.Time) * 2 * 3.14 * Frequency);
    }
};
```



Parameters

Parameters can be changed at runtime.

For example, the code below defines and uses the dynamic parameter `Frequency`:

```
shader ComputeColorWave: ComputeColor, Texturing
{
    cbuffer PerMaterial
    {
        stage float Frequency = 1.0f;
    }

    override float4 Compute()
    {
        return sin(( Global.Time ) * 2 * 3.14 * Frequency);
    }
};
```

To modify the value at runtime, access and set it in the material parameter collection. For example, to change the `Frequency`, use:

```
myMaterial.Passes[myPassIndex].Parameters.Set(ComputeColorWaveKeys.Frequency, MyFrequency);
```

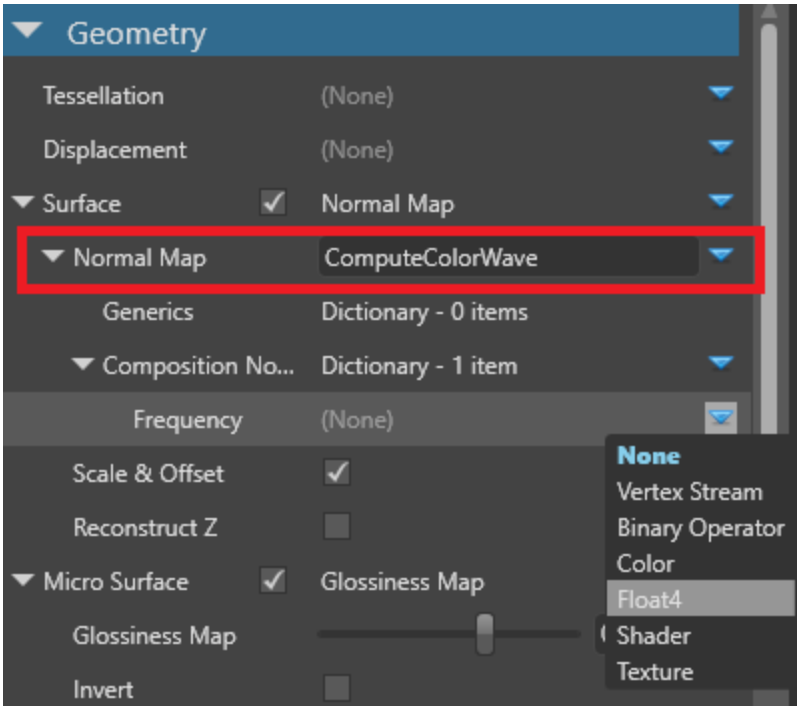
> **NOTE**
>
> `ComputeColorWaveKeys.Frequency` is generated by the Stride Visual Studio extension from the shader file.

Compositions

This composition lets you set the `Frequency` in the Game Studio Property Grid:

```
shader ComputeColorWave : ComputeColor, Texturing
{
    compose ComputeColor Frequency;

    override float4 Compute()
    {
        return sin(( Global.Time ) * 2 * 3.14 * Frequency.Compute().r);
    }
};
```

Then you can set the value in the material properties:



Troubleshooting

You can access the compilation logs of your shaders through the Game Studio's debug window's EffectCompilerCache tab (`Ctrl+Shift+D` or `Help>Show debug window`).

The launcher hosts a Visual Studio extension specifically for syntax highlighting and errors you may want to use.

The engine's source also provides a good reference point for how one would write shaders. To that end Stride.ShaderExplorer facilitates the act of finding and browsing through the different shaders the source contains.

A common issue you may stumble onto is the following:

> E1202: The mixin X in Y dependency is not in the module

This error often comes about as a result of missing or malformed Generics and Composition nodes. When that is the case you have two options:

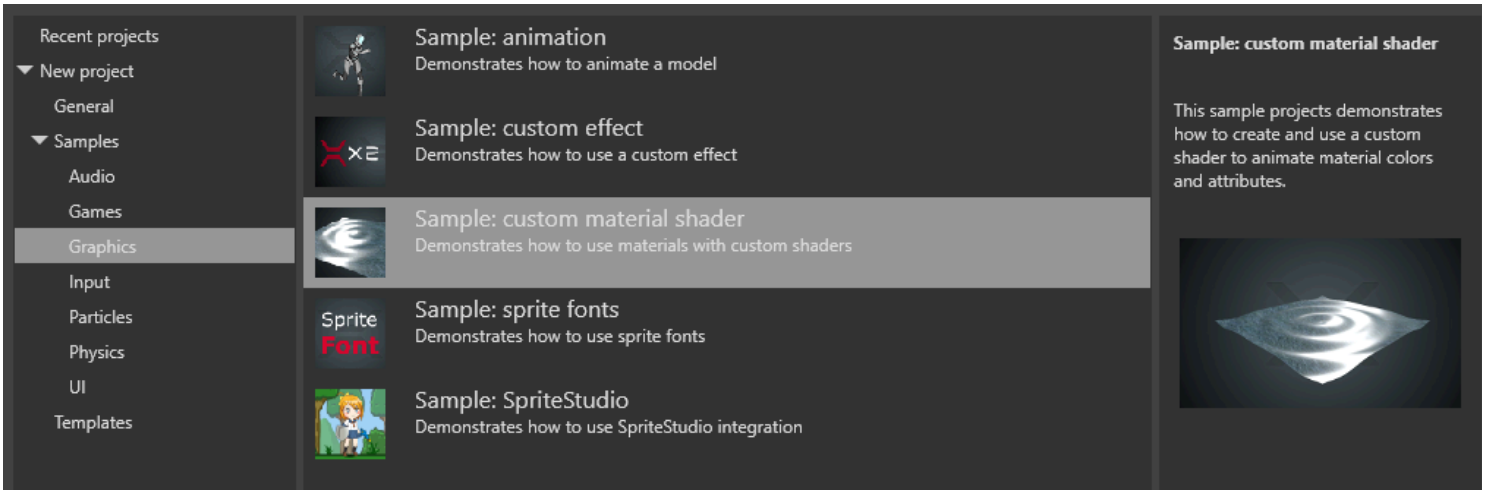
- Either manually fill these in through the material's Yaml file.
- Or remove and add the shader back in.

In both cases, you will have to close the Game Studio before doing those operation, for the former it's to ensure that the asset is reloaded from the disk. And for the latter, it's because the cache used by the property grid needs to be rebuilt.

We're currently in the process of rebuilding the shader system's backend, which should hopefully get rid of these friction.

Custom shader sample

For an example of a custom shader, see the **custom material shader** sample project included with Stride.



In the project, the **ComputeColorWaveNormal** shader is used in the **displacement map** and **surface** material properties.

See also

- Shading language
- Custom color transforms
- Material attributes
- Stride Visual Studio extension

Compile shaders

Beginner Programmer

Stride converts Stride shaders (`sdsl` and `.sdfx` files) into the shader language used by the graphics platform.

Platform	Shader language
Direct3D	HLSL
OpenGL	GLSL
Vulkan	SPIR-V
iOS	OpenGL ES

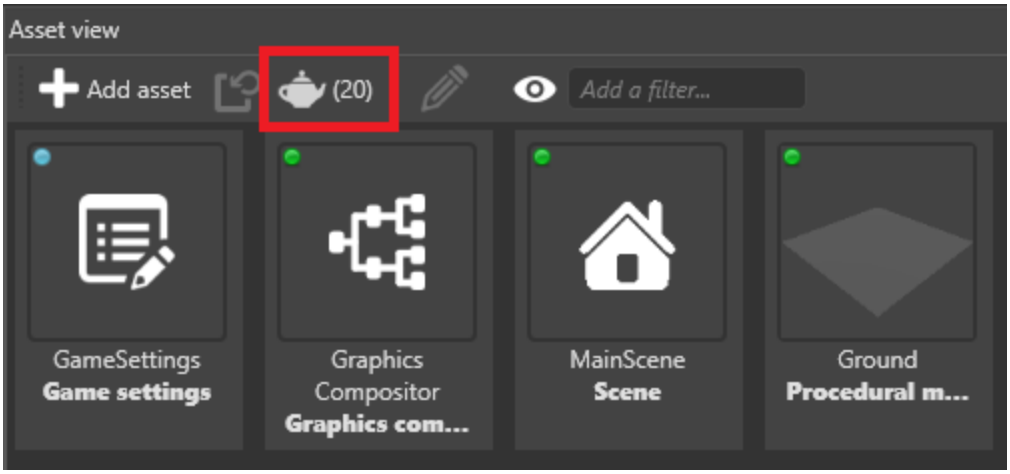
Stride can convert the shaders at runtime (when the game is running) or at build time (when the editor builds the game assets). When Stride generates shaders at runtime, rendering stops until the shader is compiled. This is usually something you want to avoid in your release build — especially on mobile platforms, which have less CPU, so the pause can be more noticable.

How Stride converts shaders at runtime

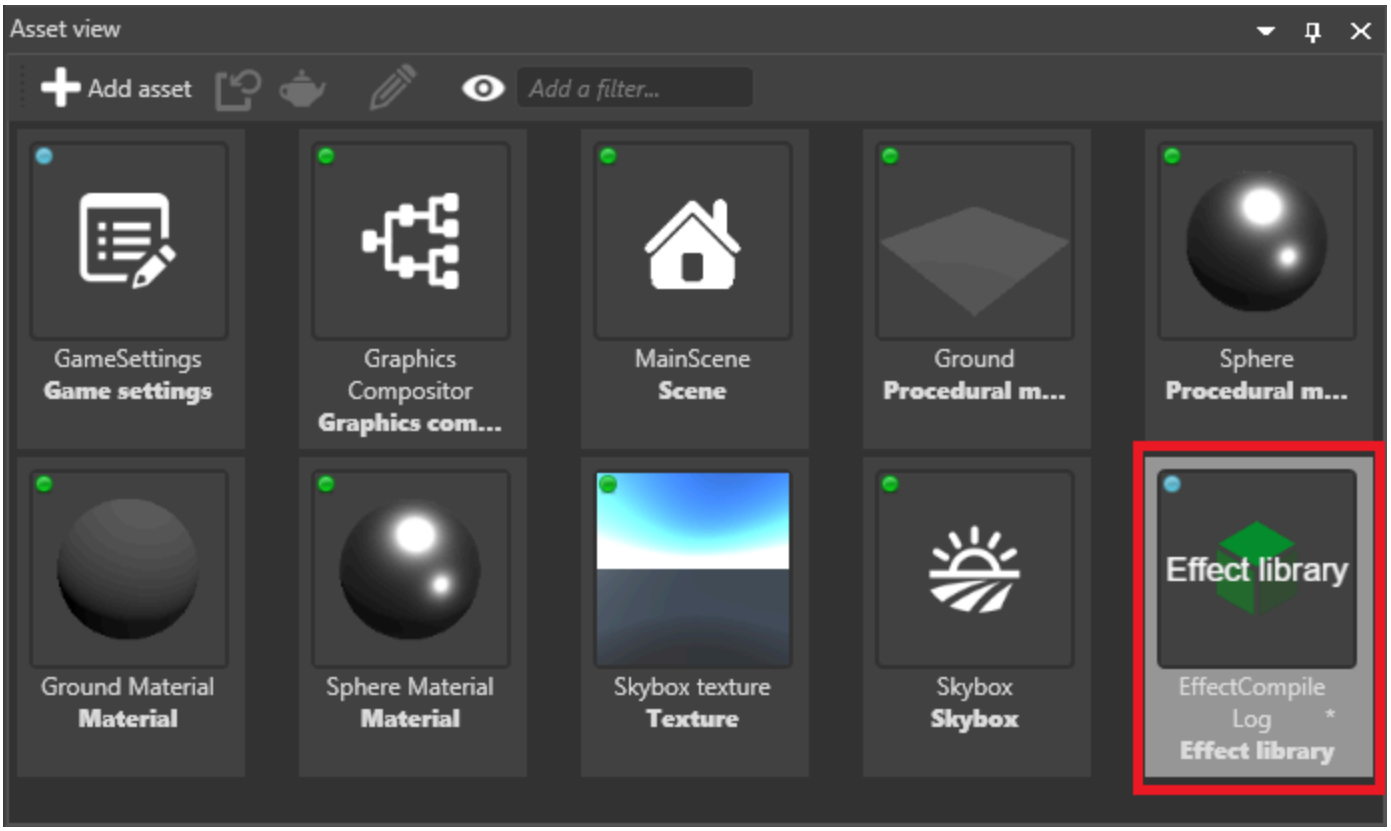
Stride can't know in advance which shaders will be used at runtime. This is because users might generate new shader permutations by, for example, changing material parameters or modifying post-effect parameters from scripts. Additionally, the final shaders depend on the graphics features on the execution platform.

1. When Stride needs a new shader at runtime, it checks its database to see if the shader has already been converted. If the shader is in the database, Stride uses it.
2. If the shader hasn't already been converted, Stride compiles it — either locally (on the device) or remotely (in Game Studio), depending on the package **User Settings** (see below).
3. If **Record used effects** is enabled in the package **User Settings** (see below), Stride notifies Game Studio that it needs a new shader.
4. Game Studio notifies you that there are new shaders to import.

In the **Asset View**, the **Import effects** button becomes available.

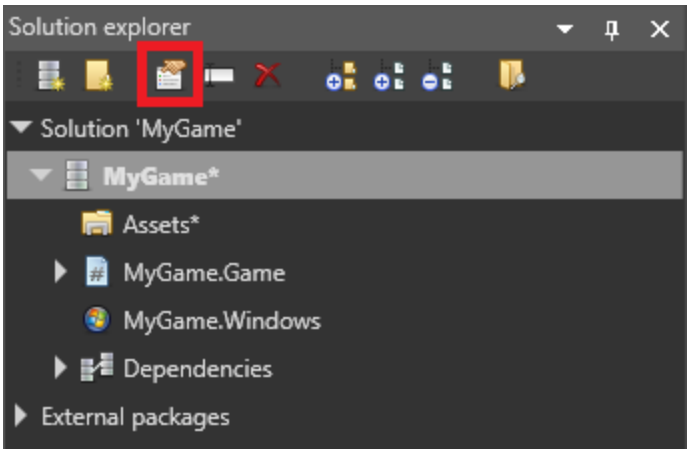


5. If you click **Import effects**, Game Studio updates the **Effect Log** (or creates it if it doesn't exist) and adds them to the game database to be used the next time you build (see step 1).

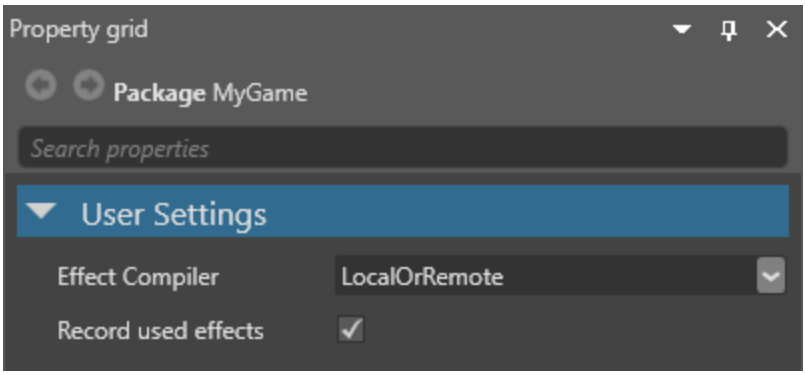


Change how Stride compiles shaders

1. In Game Studio, in the **Solution Explorer**, select the package and click **Package properties**.



2. In the **Property Grid**, set the properties.



The **Effect compiler** property specifies how to compile the shader.

- **Local**: Convert the shader on the device. This is recommended for release versions of your game.
- **Remote**: Convert the shader on the developer machine. There's no reason to use this for release versions of your game, as it won't be able to connect to the developer machine.
- **LocalOrRemote**: Convert the shader on the developer machine; if this fails, try to convert it on the device. Like the **Remote** setting, this has no use for release versions of your game.
- **None**: Don't convert shaders. Note that the application will crash if it requires a shader that isn't in the database. Currently, using this feature doesn't save any space your application, so there's no advantage in using it. However, it might be useful for making sure you have every shader in the database; if the game crashes, you know the database is missing at least one shader.

If you enable **Record used effects**, Game Studio adds new shaders to the Effect Log as soon as they're needed. We recommend you disable this for release versions of your game, as it can't connect to the developer machine.

Compile shaders remotely when developing for iOS

As iOS devices can't connect directly to PC, to convert Stride shaders remotely when developing for iOS, you need to use a Python script as a relay between the device, a Mac, and the developer PC.

1. Make sure your PC and Mac are connected to the same network.
2. On your Mac, install Python. You can download Python from the Python site.
3. Download and unzip ios-tcreplay.zip.
4. Open **Terminal**, navigate to the folder where you unzipped the file, and execute **stride-ios-relay.py MyPcName**, where **MyPcName** is the name of your developer PC.

   > **TIP**
   >
   > To find the name of your developer PC, on the PC, press the Windows key, type **About**, and press **Enter**. The PC name is listed under **PC name**.

The iOS device should now be able to communicate with the PC via your Mac to build shaders remotely.

Error messages

You may see the following error in Visual Studio's output pane while running your game:

> [RouterClient]: Error: Could not connect to connection router using mode Connect. System.AggregateException: One or more errors occurred. ---> System.Net.Sockets.SocketException: No connection could be made because the target machine actively refused it 127.0.0.1:31254

Low-level API

Advanced Programmer

The GraphicsDevice class is the central class for displaying your game. It's used to create resources and present images on the screen. You can access it as a member of the Game and ScriptComponent classes.

Actions such as drawing, setting graphics states and using resources are recorded using CommandList objects for later execution by the device.

Many command lists can be filled at the same time (eg one per thread). A default command list is available as member of the GraphicsContext of your Game.

In methods, these objects are typically provided through contexts such as RenderContext and RenderDrawContext.

Performing any drawing requires multiple steps, including:

- setting textures as render textures, clearing them, and setting viewports and scissors
- setting up the graphics pipeline state, including input description, shaders, depth-stencil, blending, rasterizer, etc
- binding resources, such as constant buffers and textures
- drawing vertices using built-in primitives or custom vertex buffers

In this section

- Draw vertices
- Pipeline state
- Resources
- SpriteBatch
- SpriteFont
- Textures and render textures

Textures and render textures

Advanced Programmer

Stride uses the Texture class to interact with texture objects in code.

For more information about rendering to a texture, see Render textures.

Load a texture

To load a texture from an asset in Stride, call this function:

```
// loads the texture called duck.dds (or .png etc.)
var myTexture = Content.Load<Texture>("duck");
```

This automatically generates a texture object with all its fields correctly filled.

Create a texture

You can also create textures without any assets (eg to be used as render target). To do this, call the constructor of the Texture class. See the Texture class reference to get the full list of available options and parameters. Some texture formats might not be available on all platforms.

Code: Create a texture

```
var myTexture = Texture.New2D(GraphicsDevice, 512, 512, false, PixelFormat.R8G8B8A8_UNorm,
TextureFlags.ShaderResource);
```

Render textures

Create a render target

The GraphicsPresenter class always provides a default render target and a depth buffer. They are accessible through the BackBuffer and DepthStencilBuffer properties. The presenter is exposed by the Presenter property of the GraphicsDevice. However, you might want to use your own buffer to perform off-screen rendering or post-processes. As a result, Stride offers a simple way to create textures that can act as render textures and a depth buffers.

Code: Create a custom render target and depth buffer

```
// render target
var myRenderTarget = Texture.New2D(GraphicsDevice, 512, 512, false,
PixelFormat.R8G8B8A8_UNorm, TextureFlags.ShaderResource | TextureFlags.RenderTarget);
```

```
// writable depth buffer
var myDepthBuffer = Texture.New2D(GraphicsDevice, 512, 512, false,
PixelFormat.D16_UNorm, TextureFlags.DepthStencil);
```

> **NOTE**
>
> Don't forget the flag RenderTarget to enable the render target behavior.
>
> Make sure the PixelFormat is correct, especially for the depth buffer. Be careful of the available formats on the target platform.

Use a render target

Once these buffers are created, you can easily set them as current render textures.

Code: Use a render target

```
// settings the render textures
CommandList.SetRenderTargetAndViewport(myDepthBuffer, myRenderTarget);

// setting the default render target
CommandList.SetRenderTargetAndViewport(GraphicsDevice.Presenter.DepthStencilBuffer,
GraphicsDevice.Presenter.BackBuffer);
```

> **NOTE**
>
> Make sure both the render target and the depth buffer have the same size. Otherwise, the depth buffer isn't used.

You can set multiple render textures at the same time. See the overloads of SetRenderTargets(Texture, Texture[]) and SetRenderTargetsAndViewport(Texture, Texture[]) method.

> **NOTE**
>
> Only the BackBuffer is displayed on screen, so you need to render it to display something.

Clear a render target

To clear render textures, call the Clear(Texture, Color4) and Clear(Texture, DepthStencilClearOptions, float, byte) methods.

Code: Clear the targets

```
CommandList.Clear(GraphicsDevice.Presenter.BackBuffer, Color.Black);
CommandList.Clear(GraphicsDevice.Presenter.DepthStencilBuffer,
DepthStencilClearOptions.DepthBuffer); // only clear the depth buffer
```

> **NOTE**
>
> Don't forget to clear the BackBuffer and the DepthStencilBuffer each frame. If you don't, you might get unexpected behavior depending on the device. If you want to keep the contents of a frame, use an intermediate render target.

Viewport

SetRenderTargetAndViewport(Texture, Texture) adjusts the current Viewport to the full size of the render target.

If you only want to render to a subset of the texture, set the render target and viewport separately using SetRenderTarget(Texture, Texture) and SetViewport(Viewport).

You can bind multiple viewports using SetViewports(Viewport[]) and SetViewports(int, Viewport[]) overloads for use with a geometry shader.

Code: Set the viewports

```
// example of a full HD buffer
CommandList.SetRenderTarget(GraphicsDevice.Presenter.DepthStencilBuffer,
GraphicsDevice.Presenter.BackBuffer); // no viewport set

// example of setting the viewport to have a 10-pixel border around the image in a full HD
buffer (1920x1080)
var viewport = new Viewport(10, 10, 1900, 1060);
CommandList.SetViewport(viewport);
```

Scissor

The SetScissorRectangle(Rectangle) and SetScissorRectangles(Rectangle[]) methods set the scissors. Unlike the viewport, you need to provide the coordinates of the location of the vertices defining the scissor instead of its size.

Code: Set the scissor

```
// example of setting the scissor to crop the image by 10 pixel around it in a full hd
buffer (1920x1080)
var rectangle = new Rectangle(10, 10, 1910, 1070);
CommandList.SetScissorRectangles(rectangle);

var rectangles = new[] { new Rectangle(10, 10, 1900, 1060), new Rectangle(0, 0, 256, 256) };
CommandList.SetScissorRectangles(rectangles);
```

See also

- Render textures

Pipeline states

Stride gives you total control over the graphics pipeline state. This includes:

- rasterizer state
- depth and stencil state
- blend state
- effects
- input layout
- output description

State is compiled into immutable PipelineState objects, which describe the whole pipeline. They are then bound using a CommandList.

Code: Create state objects

```
// Creating pipeline state object
var pipelineStateDescription = new PipelineStateDescription();
var pipelineState = PipelineState.New(GraphicsDevice, ref pipelineStateDescription);

// Applying the state to the pipeline
CommandList.SetPipelineState(pipelineState);
```

The MutablePipelineState class let you set states independently, while caching the underlying pipeline states.

Code: Mutable pipeline state

```
// Creating the pipeline state object
var mutablePipelineState = new MutablePipelineState();

// Setting values and rebuilding
mutablePipelineState.State.BlendState = BlendStates.AlphaBlend;
mutablePipelineState.Update();

// Applying the state to the pipeline
CommandList.SetPipelineState(mutablePipelineState.CurrentState);
```

Rasterizer state

The rasterizer can be set using the RasterizerState property. A set of predefined descriptions is held by the RasterizerStates class. They deal with the cull mode, and should be enough for most use cases:

- CullNone: no culling
- CullFront: front-face culling
- CullBack: back-face culling

Code: Set the cull mode

```
pipelineStateDescription.RasterizerState = RasterizerStates.CullNone;
pipelineStateDescription.RasterizerState = RasterizerStates.CullFront;
pipelineStateDescription.RasterizerState = RasterizerStates.CullBack;
```

You can create your own custom state. For the list of options and default values, see the RasterizerStateDescription API documentation.

Code: Custom rasterizer states

```
var rasterizerStateDescription = new RasterizerStateDescription(CullMode.Front);
rasterizerStateDescription.ScissorTestEnable = true; // enables the scissor test
pipelineStateDescription.RasterizerState = rasterizerStateDescription;
```

Depth and stencil states

The DepthStencilState property contains the depth and stencil states. A set of commonly used states is defined by the DepthStencilStates class:

- Default: depth read and write with a less-than comparison
- DefaultInverse: read and write with a greater-or-equals comparison
- DepthRead: read only with a less-than comparison
- None: neither read nor write

Code: Setting the depth state

```
pipelineStateDescription.DepthStencilState = DepthStencilStates.Default;
pipelineStateDescription.DepthStencilState = DepthStencilStates.DefaultInverse;
pipelineStateDescription.DepthStencilState = DepthStencilStates.DepthRead;
pipelineStateDescription.DepthStencilState = DepthStencilStates.None;
```

You can also set custom depth and stencil states. For the list of options and default values, see the DepthStencilStateDescription API documentation.

Code: Custom depth and stencil state

```
// depth test is enabled but it doesn't write
var depthStencilStateDescription = new DepthStencilStateDescription(true, false);
pipelineStateDescription.DepthStencilState = depthStencilStateDescription;
```

The stencil reference isn't part of the PipelineState. It's set using SetStencilReference(int).

Code: Set the stencil reference

```
CommandList.SetStencilReference(2);
```

Blend state

The BlendState and SampleMask properties control blending. The BlendStates class defines a set of commonly used blend states:

- Additive: sums the colors
- AlphaBlend: sums the colors using the alpha of the source on the destination color
- NonPremultiplied: sums using the alpha of the source on both colors
- Opaque: replaces the color

Code: Set the blend state

```
// Set common blend states
pipelineStateDescription.BlendState = BlendStates.Additive;
pipelineStateDescription.BlendState = BlendStates.AlphaBlend;
pipelineStateDescription.BlendState = BlendStates.NonPremultiplied;
pipelineStateDescription.BlendState = BlendStates.Opaque;

// Set the sample mask
pipelineStateDescription.SampleMask = 0xFFFFFFFF;
```

You can set custom depth and blend states. For a list of options and default values, see the BlendState Description API documentation.

Code: Custom blend state

```
// create the object describing the blend state per target
var blendStateRenderTargetDescription = new BlendStateRenderTargetDescription();
blendStateRenderTargetDescription.BlendEnable = true;
blendStateRenderTargetDescription.ColorSourceBlend = Blend.SourceColor;
// etc.
```

```
// create the blendStateDescription object
var blendStateDescription = new BlendStateDescription(Blend.SourceColor,
Blend.InverseSourceColor);
blendStateDescription.AlphaToCoverageEnable = true; // enable alpha to coverage
blendStateDescription.RenderTargets[0] = blendStateRenderTargetDescription;
pipelineStateDescription.BlendState = blendStateDescription;
```

The blend factor isn't part of the PipelineState. It's set using SetBlendFactor(Color4).

Code: Set the blend factor

```
CommandList.SetBlendFactor(Color4.White);
```

Effects

The pipeline state also includes the shaders you want to use for drawing. To bind an Effect to the pipeline, set the EffectBytecode and RootSignature properties of the PipelineStateDescription to the matching values of the effect.

An EffectBytecode contains the actual shader programs. For more information, see Effects and Shaders.

The RootSignature describes the number and kind of resources expected by the effect. The next chapter covers how to bind resources to the pipeline.

Code: Bind an effect

```
var effectInstance = new
EffectInstance(EffectSystem.LoadEffect("MyEffect").WaitForResult());
pipelineStateDescription.EffectBytecode = effectInstance.Effect.Bytecode;
pipelineStateDescription.RootSignature = effectInstance.RootSignature;
```

Input layout

The pipeline state describes the layout in which vertices are sent to the device through the Input Elements and PrimitiveType properties.

The Draw vertices page describes how to create custom vertex buffers and their VertexDeclaration in more detail.

Code: Set an input layout

```
VertexDeclaration vertexDeclaration = new VertexDeclaration();
pipelineStateDescription.InputElements = vertexDeclaration.CreateInputElements();
```

```
pipelineStateDescription.PrimitiveType = PrimitiveType.TriangleStrip;
```

Output description

The Output property of the PipelineStateDescription defines the number and PixelFormat of all bound render textures.

For information on how to bind render textures to the pipeline, see Textures and render textures.

Code: Create an output description

```
var renderOutputDescription = new
RenderOutputDescription(GraphicsDevice.Presenter.BackBuffer.Format,
GraphicsDevice.Presenter.DepthStencilBuffer.Format);
pipelineStateDescription.Output = renderOutputDescription;
```

You can use the CaptureState(CommandList) to retrieve the output description last set on a Command List. This is especially useful in combination with MutablePipelineState, when the render target might not be known up front.

Code: Capture output description

```
mutablePipelineState.State.Output.CaptureState(CommandList);
mutablePipelineState.Update();
```

Resource binding

Advanced Programmer

When drawing vertices using an effect, the shaders expect certain resources to be available, including:

- textures and buffers
- samplers
- constant buffers

Automatic resource binding

The EffectInstance class handles the details of enumerating these resources from a loaded effect as well as binding them.

It exposes the RootSignature, which has to be set as pipeline state, and allows to fill constant buffers and bind resources based on a ParameterCollection.

Code: Using an EffectInstance

```
// Create a EffectInstance and use it to set up the pipeline
var effectInstance = new
EffectInstance(EffectSystem.LoadEffect("MyEffect").WaitForResult());
pipelineStateDescription.EffectBytecode = effectInstance.Effect.Bytecode;
pipelineStateDescription.RootSignature = effectInstance.RootSignature;

// Update constant buffers and bind resources
effectInstance.Apply(context.GraphicsContext);
```

Manual resource binding

When more optimized code is required (eg in the rendering pipeline), constant buffer updates and resource binding can be done manually.

Draw vertices

Advanced Programmer

When loading a scene, Stride automatically handles the draw calls to display the scene throughout the entity system. This page introduces manual drawing.

Primitives

Stride provides the following set of built-in primitives:

- Plane
- Cube
- Sphere
- Geosphere
- Cylinder
- Torus
- Teapot

They aren't automatically created along with the GraphicsDevice, so you have to instantiate them. You can do this through the GeometricPrimitive class.

Code: Creating and using a primitive

```
// creation
var myCube = GeometricPrimitive.Cube.New(GraphicsDevice);
var myTorus = GeometricPrimitive.Torus.New(GraphicsDevice);

// ...

// draw one on screen
myCube.Draw(CommandList, EffectInstance);
```

They have no effect associated with them, so the user has to provide an EffectInstance when drawing. For information on loading effects, please see Effects and shaders.

Custom drawing

Outside of built-in primitives, any geometry can be drawn by creating custom vertex buffers. To create a vertex buffer, first a VertexDeclaration has to be defined. A vertex declaration describes the elements of each vertex and their layout. For details, see the VertexElement reference page.

Next, a vertex buffer can be created from an array of vertices. The vertex data type has to match the VertexDeclaration.

Given vertex buffer and declaration, a VertexBufferBinding can be created.

Code: Creating a vertex buffer

```
// Create a vertex layout with position and texture coordinate
var layout = new VertexDeclaration(VertexElement.Position<Vector3>(),
VertexElement.TextureCoordinate<Vector2>());

// Create the vertex buffer from an array of vertices
var vertices = new VertexPositionTexture[vertexCount];
var vertexBuffer = Buffer.Vertex.New(GraphicsDevice, vertices);

// Create a vertex buffer binding
var vertexBufferBinding = new VertexBufferBinding(vertexBuffer, layout, vertexCount);
```

To draw the newly created vertex buffer, it has to be bound to the pipeline. The vertex layout and the PrimitiveType to draw have to be included in the pipeline state object. The buffer itself can be set dynamically.

Afterwards, the vertices are ready to be rendered using Draw(int, int).

Code: Binding and drawing vertex buffers

```
// Set the pipeline state
pipelineStateDescription.InputElements = vertexBufferBinding.Layout.CreateInputElements();
pipelineStateDescription.PrimitiveType = PrimitiveType.TriangleStrip;

// Create and set a PipelineState object
// ...

// Bind the vertex buffer to the pipeline
commandList.SetVertexBuffers(0, vertexBuffer, 0, vertexBufferBinding.Stride);

// Draw the vertices
commandList.Draw(vertexCount);
```

It is also possible to draw indexed geometry. To use an index buffer, first create it similarly to the vertex buffer and bind it to the pipeline. It can then be used for drawing using DrawIndexed(int, int, int).

Code: Drawing indexed vertices

```
// Create the index buffer
var indices = new short[indexCount];
var is32Bits = false;
```

```
var indexBuffer = Buffer.Index.New(GraphicsDevice, indices);

// set the VAO
commandList.SetIndexBuffer(indexBuffer, 0, is32Bits);

// Draw indexed vertices
commandList.DrawIndexed(indexBuffer.ElementCount);
```

SpriteBatch

Advanced Programmer

A sprite batch is a collection of sprites (2D textured planes).

> **NOTE**
> Remember that you need to put all custom code in a custom scene renderer to include it in the composition.

Create a sprite batch

Stride offers a easy way to deal with batches of sprites through the SpriteBatch class. You can use this class to regroup, update, and display sprites efficiently.

Code: Creating a sprite batch

```
var spriteBatch = new SpriteBatch(GraphicsDevice);
```

You can specify the size of your batch size. This isn't the maximum number of sprites the SpriteBatch is able to display, but the maximum number of sprites it can store before drawing.

Code: Setting the batch size

```
var spriteBatch = new SpriteBatch(GraphicsDevice, 2000);
```

You can also set states like the ones discussed on the Pipeline state page.

Draw a sprite batch

The SpriteBatch class has multiple draw methods to set various parameters. For a list of features, see the SpriteBatch API documentation.

Code: Drawing a sprite batch

```
// begin the sprite batch operations
spriteBatch.Begin(Game.GraphicsContext, SpriteSortMode.Immediate);

// draw the sprite immediately
spriteBatch.Draw(myTexture, new Vector2(10, 20));
```

```
// end the sprite batch operations
spriteBatch.End();
```

There are five modes to draw a sprite batch. They are enumerated in the SpriteSortMode enum:

- Deferred (default mode): the sprites are drawn at the same time at the end to reduce the drawcall overhead
- Immediate: the sprites are drawn after each Draw call
- Texture: Deferred mode but sprites are sorted based on their texture to reduce effect parameters update
- BackToFront: Deferred mode with a sort based on the z-order of the sprites
- FrontToBack: Deferred mode with a sort based on the z-order of the sprites

To set the mode, specify it in the Begin method.

Code: Deferred drawing of the sprite batch

```
// begin the sprite batch operations
spriteBatch.Begin(Game.GraphicsContext); // same as spriteBatch.Begin(GraphicsContext,
SpriteSortMode.Deferred);

// store the modification of the sprite
spriteBatch.Draw(myTexture, new Vector2(10, 20));

// end the sprite batch operations, draw all the sprites
spriteBatch.End();
```

You can set several parameters on the sprite. For example:

- position
- rotation
- scale
- depth
- center offset
- color tint

For a full list, see the SpriteBatch API documentation, especially the **Draw** methods.

Code: More complex sprite batch drawing

```
// begin the sprite batch operations
spriteBatch.Begin(GraphicsContext);
const int gridCount = 10;
```

```
var textureOffset = new Vector2((float)graphicsDevice.BackBuffer.Width/gridCount,
(float)graphicsDevice.BackBuffer.Height/gridCount);
var textureOrigin = new Vector2(myTexture.Width/2.0f, myTexture.Height/2.0f);

// draw 100 sprites on a 10x10 grid with a rotation of 1.2 rad and a scale of 0.5 for each
of them
for (int y = 0; y < gridCount; y++)
{
    for (int x = 0; x < gridCount; x++)
    {
        spriteBatch.Draw(UVTexture, new Vector2(x * textureOffset.X + textureOffset.X
/ 2.0f, y * textureOffset.Y + textureOffset.Y / 2.0f), Color.White, 1.2f,
textureOrigin, 0.5f);
    }
}

// end the sprite batch operations, draw all the sprites
spriteBatch.End();
```

See also

- SpriteFont

SpriteFont

Advanced Programmer

The SpriteFont class is a convenient way to draw text. It works with the SpriteBatch class.

> **NOTE**
> You need to put all custom code in a Custom scene renderer to include it in the composition.

Load a spriteFont

After a font asset is compiled it can be loaded as a SpriteFont instance using the ContentManager. It contains all the options to display a text (bitmaps, kerning, line spacing etc).

Code: Load a SpriteFont

```
var myFont = Content.Load<SpriteFont>("MyFont");
```

Write text on screen

Once the font is loaded, you can display any text with a SpriteBatch. The DrawString method performs the draw. For more information about the SpriteBatch, see the SpriteBatch page.

Code: Write text

```
// create the SpriteBatch
var spriteBatch = new SpriteBatch(GraphicsDevice);

// don't forget the begin
spriteBatch.Begin(GraphicsContext);

// draw the text "Helloworld!" in red from the center of the screen
spriteBatch.DrawString(myFont, "Helloworld!", new Vector2(0.5, 0.5), Color.Red);

// don't forget the end
spriteBatch.End();
```

The various overloads let you specify the text's orientation, scale, depth, origin, etc. You can also apply some SpriteEffects to the text:

- None

- FlipHorizontally
- FlipVertically
- FlipBoth

Code: Advanced text drawing

```
// draw the text "Hello world!" upside-down in red from the center of the screen
spriteBatch.DrawString(myFont, "Hello world!", new Vector2(0.5, 0.5), Color.Red, 0, new
Vector2(0, 0), new Vector2(1,1), SpriteEffects.FlipVertically, 0);
```

See also

- SpriteBatch

Rendering pipeline

Render features

Rendering logic is divided into RenderFeatures. Each render feature processes one type of RenderObject (eg meshes, sprites, particles, etc).

Stride executes features in phases: **collect**, **extract**, **prepare** and **draw**. This means each step of the pipeline can be isolated, parallelized and optimized separately.

For more information, see Render features.

Render views

You can render scenes from multiple points of view, represented as RenderViews – eg player views in a splitscreen game, or separate shadow views for cascades in a shadow map cascade.

Views are a first-class concept available to all rendering phases, allowing batching across multiple views.

Render stages

RenderStages select the effect and pipeline state per object, and define the output of a render pass.

For more information, see Render stages.

Visibility

RenderObjects are registered with a VisibilityGroup. During the **collect** phase, the visibility group culls and filters them based on the RenderView and RenderStage.

In this section

- Render features
- Render stages

Render features

A RenderFeature is responsible for drawing a given type of RenderObject.

Render phases

Render features have several phases.

Collect

The **collect** phase determines what needs to be processed and rendered. It's usually driven by the graphics compositor.

The collect phase:

- creates render views, and updating them with the most recent data such as view and projection matrices
- creates and setting up render stages
- performes visibility culling and sorting

Extract

The **extract** phase copies data from game states of previously collected objects to short-lived render-specific structures. It's usually driven by the RenderSystem and RenderFeatures.

This should be as fast as possible and avoid heavy computations since game update and scripts are blocked. Heavy computations should be deferred to Prepare.

> **NOTE**
> Currently, Stride doesn't parallelize game updates and scripts, so they won't be resumed until the **prepare** and **draw** phases are finished.

Example tasks:

- copying object matrices
- copying material parameters

Prepare

The **prepare** phase prepares GPU resources and performs heavy computations. This is usually driven by the RenderSystem and RenderFeatures.

Example tasks:

- computing lighting data and structures
- filling constant buffers and resource tables

Draw

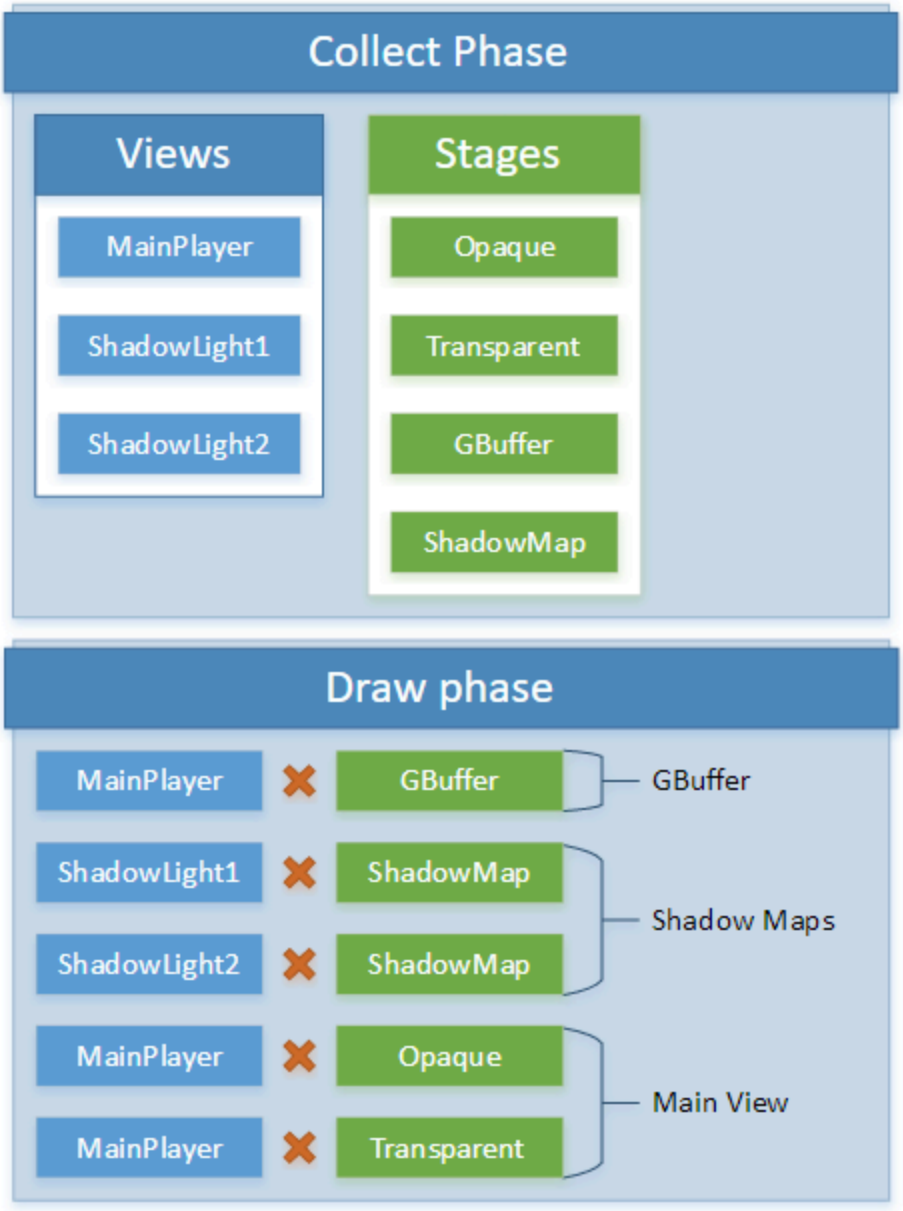
The **draw** phase fills the GPU command list.

Example tasks:

- setting up render textures
- drawing combinations of render stage with render view.

Example

A typical example of views and stages created during **collect** phase, used during the **draw** phase:



Pipeline processors

Pipeline processors are classes called when creating the pipeline state. This lets you do things such as enable alpha blending or wireframe rendering in a specific render stage.

Stride includes several predefined pipeline processors. You can also create your own.

See also

- Rendering pipeline
- Render stages
- Effects and shaders
- Graphics compositor

Render stages

Render stages define how given objects are rendered (usually with their associated effect/shader). They also let you control advanced properties such as sorting and filtering objects.

Objects can subscribe to multiple render stages. For example, a mesh typically subscribes to both the `Opaque` and `ShadowCaster` stages, or the `Transparent` stage.

> **NOTE**
>
> Render stages don't define the rendering order. The rendering order is controlled by the graphics compositor.

Effect slots

Effect slots determine which effect/shader a render stage uses. You choose the effect slot with EffectSlot Name.

If multiple render stages exclusively render different objects, the stages can share the same effect slot. For example, as the opaque stage only renders opaque objects and the transparent stage only renders transparent objects, both stages can use the main effect slot.

If they render any of the same objects, they can't share effect slots. This is why, for example, we typically render opaque meshes with the main effect slot, then render opaque meshes again with the shadow caster effect slot to create shadows.

A typical setup of render stages:

Render stage	Effect slot
Opaque	Main
Transparent	Main
UI	Main
Shadow caster	Shadow caster

Sort objects in a render stage

SortMode defines how Stride sorts objects in that render stage. Stride comes with several SortMode implementations, such as:

- FrontToBackSortMode, which renders objects from front to back with limited precision, and tries to avoid state changes in the same depth range of objects (useful for opaque objects and shadows)
- BackToFrontSortMode, which renders objects strictly from back to front (useful for transparent objects)
- StateChangeSortMode, which tries to reduce state changes

Of course, you're free to implement your own, too.

Filter objects in a render stage

To create your own filter for objects in a render stage, inherit from RenderStageFilter.

Render stage selectors

Render stage selectors define which objects in your scene are sent to which render stage, and choose which effect to use when rendering a given object.

For example, this is the typical setup for meshes:

- MeshTransparentRenderStageSelector chooses either the `Main` or `Transparent` render stage, depending on the material properties. The default effect is `StrideForwardShadingEffect` defined by Stride (you can create your own if you want).
- ShadowMapRenderStageSelector selects opaque meshes that cast shadows and adds them to the `ShadowMapCaster` render stage. The default effect is `StrideForwardShadingEffect.ShadowMapCaster`, defined by Stride.

Either can filter by render group.

You can customize everything, so you can add other predefined render stage selectors (eg to add UI to a later full-screen pass), or create your own selector specific to your game.

See also

- Rendering pipeline
- Render features
- Effects and shaders
- Graphics compositor

Sprite fonts

Intermediate

Sprite fonts take a TrueType font as an input (either a system font or a file you assign) and then create all the images (sprites) of characters (glyphs) for your game.

It's often inefficient to render fonts directly. We usually want to create (rasterize) them just once, then only render the image of a letter character (eg A, a, B, C etc) every time we need it. This involves creating a sprite (billboarded rectangular image) of the character, which is displayed on the screen as a regular image. A text block would be a collection of sprites rendered as quads so all the characters are aligned and spaced properly.

Offline-rasterized sprite fonts

Offline-rasterized sprite fonts create (rasterize) a fixed number of characters (glyphs) of a certain size, and bake them into an atlas texture when building the assets for your game.

In the game, they can only be drawn with this size. Only the characters you specify can be displayed.

When to use offline-rasterized fonts

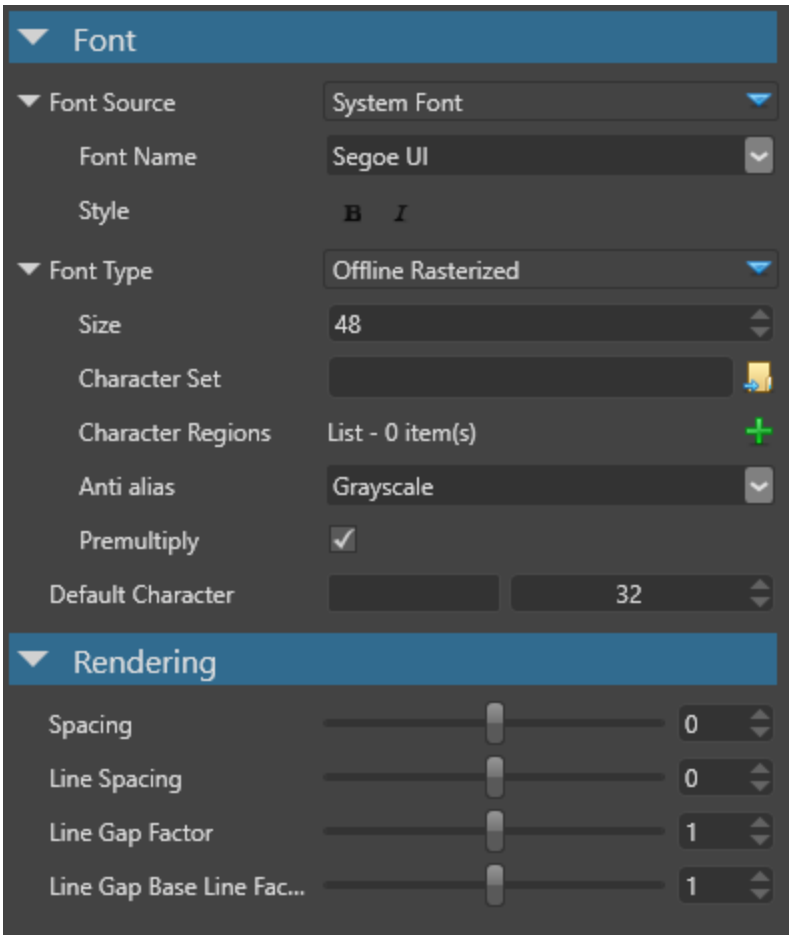
Use offline-rasterized fonts when:

- you use a font of known size with a known character set in your game
- you need anti-aliasing on your fonts
- your UI is only used in fullscreen mode

Do **not** use offline-rasterized fonts when:

- your UI is rendered as part of the 3D world scene
- you have a varied or unknown number of font sizes and character sets

Offline-rasterized sprite font properties



Property	Description
Font Source	System (installed on this machine) or from file. The system fonts can also take **Bold** and *Italic* options.
Font Type	Offline Rasterized
Size (in pixels)	The font is baked with this size. No other font size can be displayed.
Character set	(Optional) A text file containing all characters which need to be baked.
Character regions	Code for regions of characters which need to be baked. For example, (32 - 127) is a region sufficient for ASCII character sets.
Anti alias	None, Grayscale or ClearType
Premultiply	If the alpha should be premultiplied. Default is yes to match the rest of the engine pipeline.
Default character	Missing characters default to this when rendered. The default code is 32 which is space.

Runtime-rasterized sprite fonts

Runtime-rasterized sprite fonts create (rasterize) a varied number of characters (glyphs) of any size and bake them into an atlas texture **on demand**.

This function is invoked at runtime when you change the font size or request characters that haven't been drawn before.

Under the hood, the runtime-rasterized fonts use similar atlas textures to the offline-rasterized fonts. This means that if you have three different font sizes, they take about three times more memory than a single font size. The font sizes are also taken into account.

When to use runtime-rasterized fonts

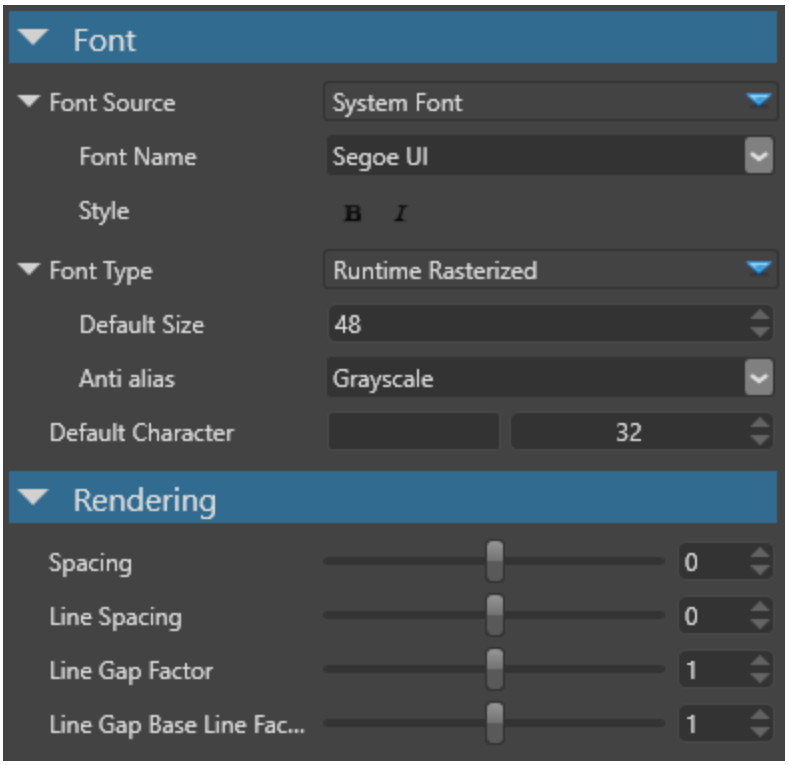
Use runtime-rasterized fonts when:

- you need multiple sizes for your font or don't know which characters you need
- the number of possible characters in the font greatly outnumbers the number of characters you need to display at runtime (eg Japanese or Chinese, which use thousands of characters)
- you need anti-aliasing on your fonts
- your UI is only used in fullscreen mode

Do **not** use runtime-rasterized fonts when:

- your UI is rendered as part of the 3D world scene
- you only need one or two known sizes for a small character set
- you have a scaling text (as runtime-rasterized fonts will recreate every single font size)

Runtime-rasterized sprite font properties



Property	Description
Font Source	System (installed on this machine) or from file. The system fonts can also take **Bold** and *Italic* options.
Font Type	Runtime Rasterized
Default Size (in pixels)	If size isn't specified the text is rendered with this one.
Anti alias	None, Grayscale or ClearType
Default character	Missing characters will default to this one when rendered. The default code is 32, which is space.

Signed distance field sprite fonts

Signed distance field (SDF) fonts use an entirely different technique to render fonts. Rather than rasterize the color of the character on the sprite, they output the distance of the current pixel to the closest edge of the glyph.

The distance is positive if the pixel is **inside** the glyph boundaries, and negative if the pixel is **outside** the glyph (hence the name signed).

When rendering, check the distance and output a white pixel if it's positive or `0`, and a black pixel if it's negative. This allows very sharp and clean edges to be rendered even under magnification (which

otherwise makes traditional sprites look pixelated).

The image below compares SDF fonts and the offline-rasterized fonts under magnification:

When to use SDF fonts

Use SDF fonts when:

- your UI is rendered as part of the 3D world scene or fullscreen (SDF works well for both cases)
- you have a scaling text or expect the user to be able to zoom in
- you require multiple sizes for your font
- you have very large font sizes (SDF consumes less memory than runtime-rasterized fonts)

Do **not** use SDF fonts when:

- you need anti-aliasing on your fonts (SDF fonts currently don't support it)
- you only require one or two known sizes for a small character set (better use offline-rasterized font)
- the number of possible characters in the font greatly outnumbers the number of characters you need to display at runtime (eg Japanese or Chinese, which use thousands of characters). If a runtime-rasterized font is not an option (eg because of scaling), make sure you bake every character you might need, or they won't be displayed.

SDF properties

Property	Description
Font Source	System (installed on this machine) or from file. The system fonts can also choose **Bold** and *Italic* options.
Font Type	Signed Distance Field
Size (in pixels)	The font will be baked with this size. All font sizes can still be displayed. Bigger size usually results in better quality, and generally you want to keep this at 20 or more to avoid visual glitches.
Character set	(Optional) A text file containing all characters which need to be baked.
Character regions	Code for regions of characters which need to be baked. For example (32 - 127) is a region sufficient for ASCII character sets.
Default character	Missing characters will default to this one when rendered. The default code is 32 which is space.

Texture atlases for different sprite fonts

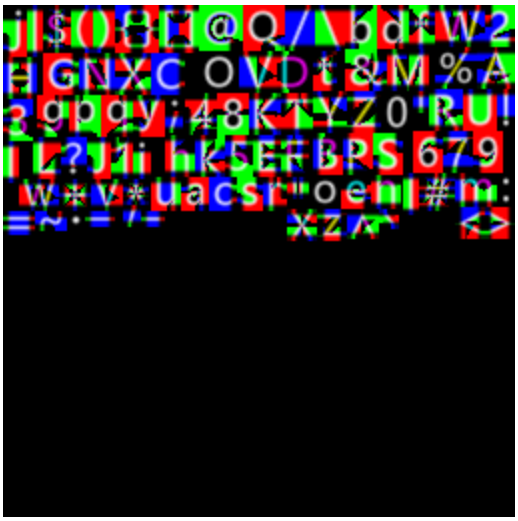
Offline rasterized

The offline-rasterized sprite font bakes all requested characters once in a grayscale texture. If you zoom in, you'll see that they are pixelated. The font has a fixed size and doesn't work well for scaling text.

Runtime rasterized

The runtime-rasterized sprite font only bakes (rasterizes) the characters that are drawn in the game. The initial atlas texture is intentionally bigger so it can hold more characters of potentially different sizes before it needs resizing.

Signed distance field



Like the offline-rasterized sprite font, the signed distance field sprite font bakes all requested characters once. The major difference is that it encodes distances from the character lines rather than actual color, and it uses all three channels' RGB. You can still recognize each character, but a special shader is needed to render them properly. The upside is that the edges remain sharp, even under magnification.

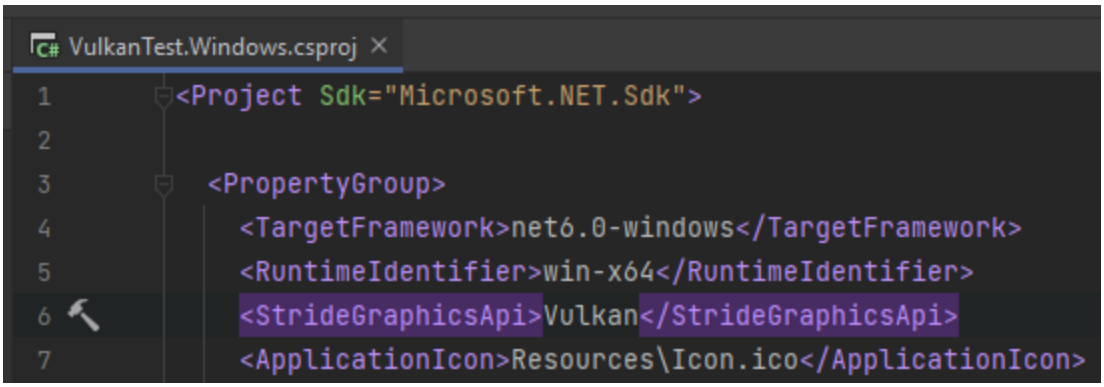
Further reading

- Paper on how distance fields and multi-channel distance fields in particular work
- Stack Exchange thread outlining the differences between single-channel SDF and multi-channel SDF fonts

Graphics API

To run your projects through a different API than the default one, add the following line to the `PropertyGroup` of your executable's `.csproj` file:

```
<StrideGraphicsApi>Vulkan</StrideGraphicsApi>
```



Supported values are as follows:

```
Null
Direct3D11
Direct3D12
OpenGL
OpenGLES
Vulkan
```

You *may* also have to add `<PackageReference Include="Stride.Shaders.Compiler" Version="x.x.x.x" />` to your *main* `.csproj`, and don't forget to replace `Version` appropriately.

```
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Stride.Engine" Version="4.1.0.1" />
  <PackageReference Include="Stride.Shaders.Compiler" Version="4.1.0.1" />

  <PackageReference Include="Stride.Video" Version="4.1.0.1" />
  <PackageReference Include="Stride.Physics" Version="4.1.0.1" />
  <PackageReference Include="Stride.Navigation" Version="4.1.0.1" />
  <PackageReference Include="Stride.Particles" Version="4.1.0.1" />
  <PackageReference Include="Stride.UI" Version="4.1.0.1" />
```

Engine

If you are using a local build of the engine you should run the build again with the following command:

```
msbuild /t:Build /p:StrideGraphicsApiDependentBuildAll=true Stride.sln
```

Input

Beginner Programmer

Users interact with games and applications using **input devices** such as gamepads, mice, and keyboards. Every interactive application must support at least one input device.

Stride handles input entirely via scripts. There are low-level and high-level APIs to handle different input types:

- **Low-level** APIs are close to hardware, so they have lower latency. These allow fast processing of the input from pointers, keyboards, mouse, gamepads, and some sensors.
- **High-level** APIs interpret input for you, so they have higher latency. These APIs are used for gestures and some sensors.
- There are also **special APIs** for some sensors and virtual buttons.

Handle input

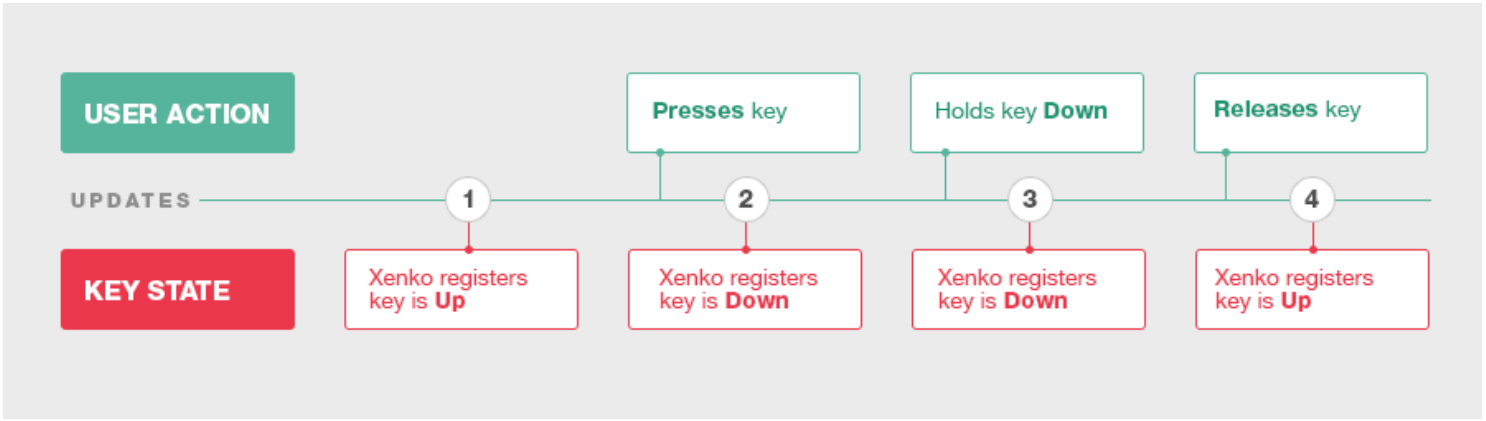
Handle input with the InputManager class. You can access this class from a script with its properties and methods.

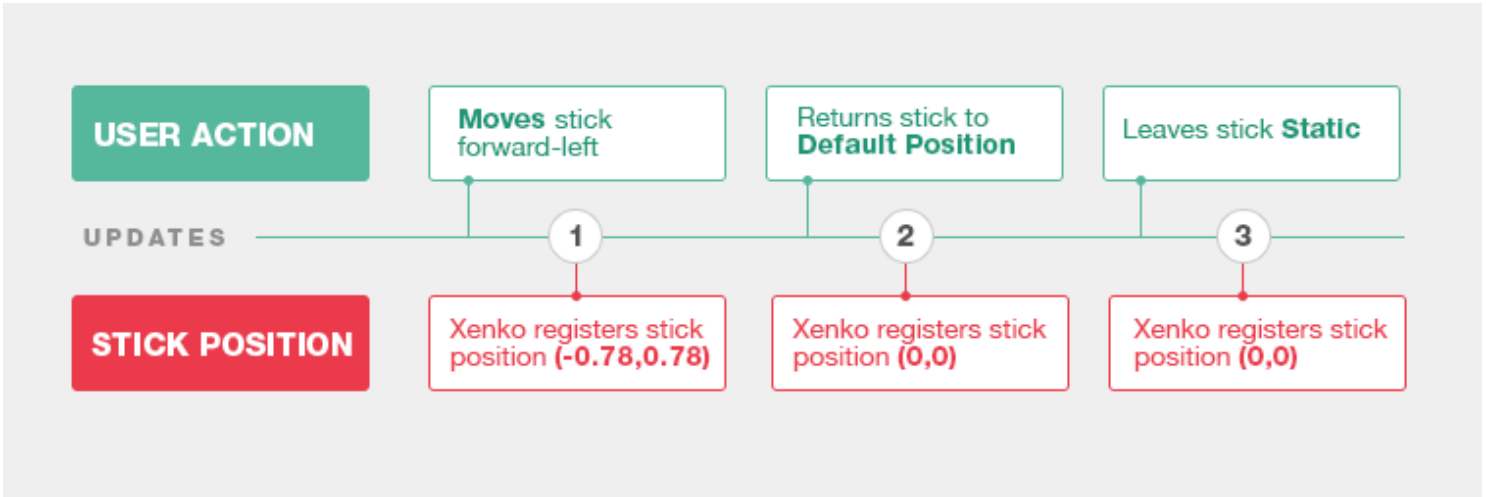
To check whether a particular input device is available, use the corresponding InputManager property. For example, to check if a mouse is connected, use Input.HasMouse.

After you check the device availability, there are four ways to handle input in Stride.

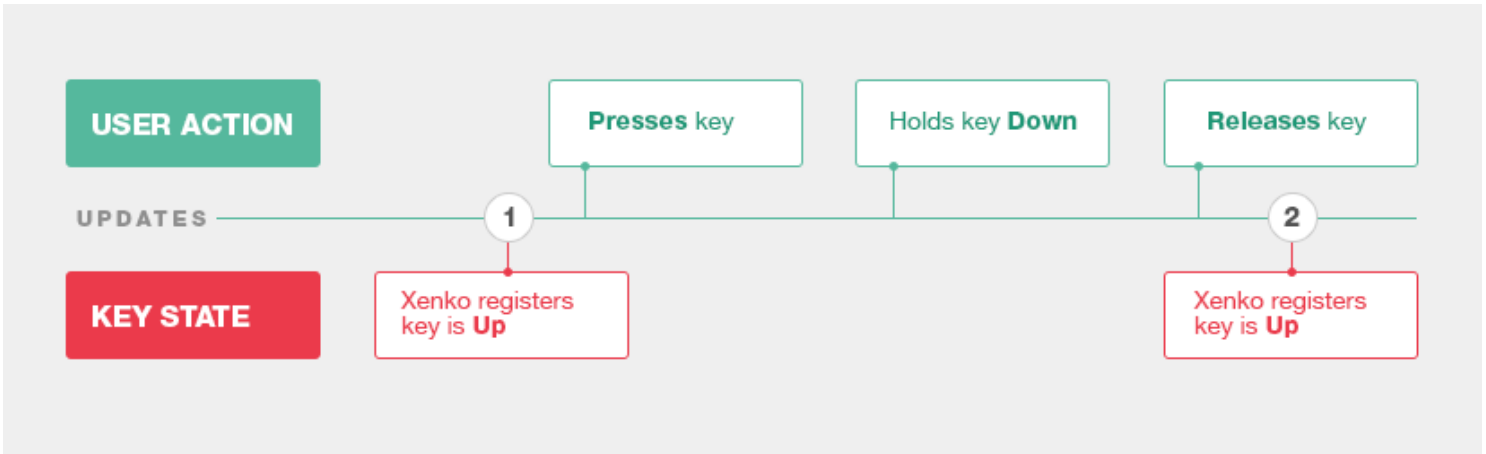
Query state

You can query the state of digital keys and buttons (ie *Up* or *Down*) and the numeric values of analog buttons and sensors. For example, DownKeys gets a list of the keys that were in the state *Down* in the last update.





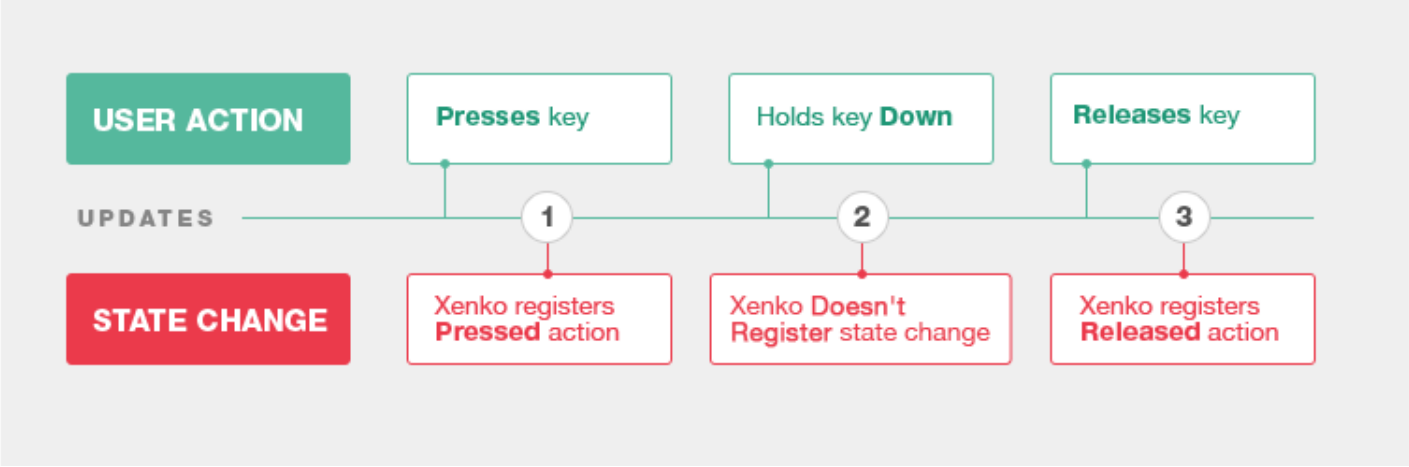
Sometimes a user performs more than one action between updates. If there's no state change between the updates (the end result is the same), Stride registers no action:



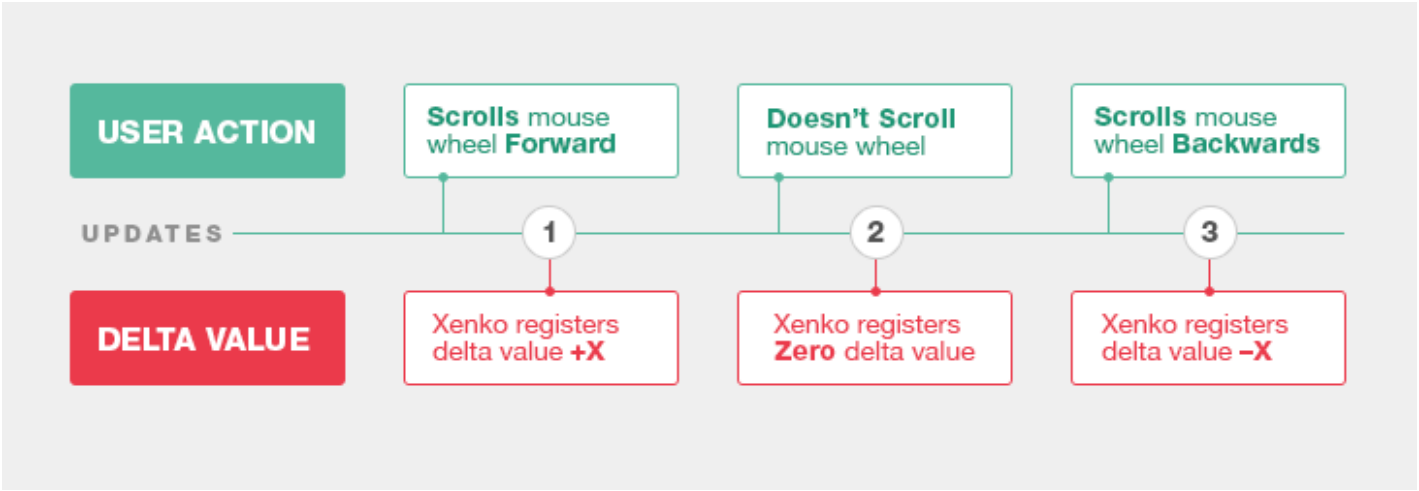
Query a state change

You can query the change of state of buttons and keys since the previous update. In this case, you don't get the list of all buttons and keys, but have to query each button and key separately.

- For digital buttons and keys, query if the button or key was *Pressed*, *Down* or *Released* in the last update.



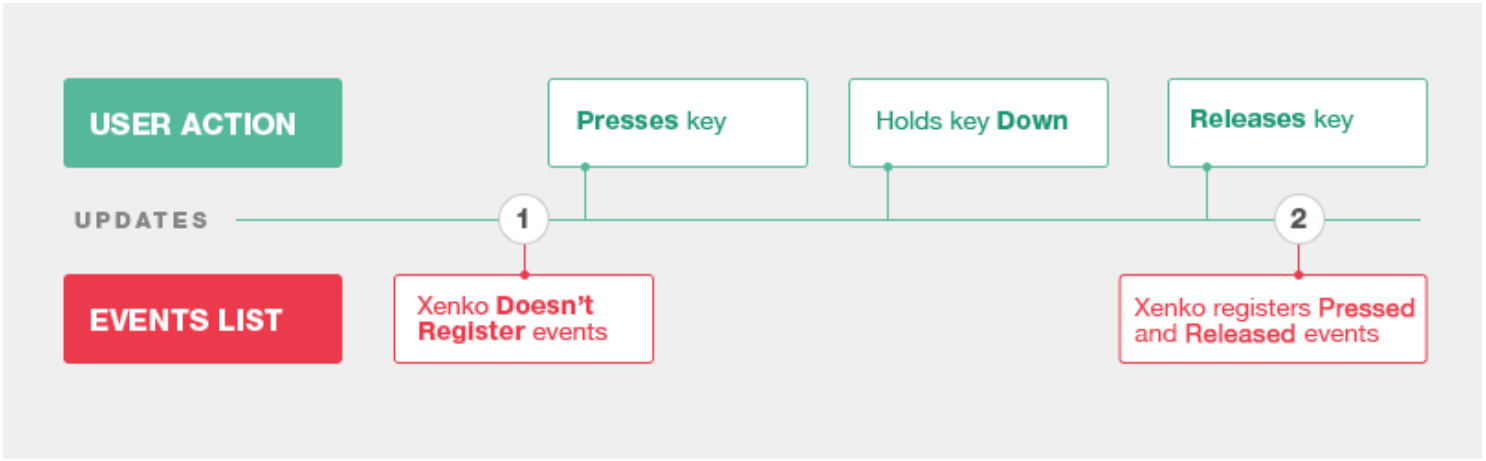
- For mouse positions and mouse wheel scrolling, query *Delta Values* since the previous update:



Sometimes a user performs several actions between two updates. If there's no state change between two updates (the end result is the same), Stride registers no action.

Query the list of events

For pointers, gestures, and keyboards, you can query all the events that happened in the last update.



> **NOTE**
>
> Even if a user performs several actions between two updates, Stride registers all these events.

Use virtual buttons

You can use **virtual buttons** to associate input to actions rather than physical keys, then let the user define their own keys. For more information, see virtual buttons.

In this section

- Gamepads
- Gestures
- Keyboards
- Mouse
- Pointers
- Sensors
- Virtual buttons

Gamepads

Beginner Programmer

Gamepads, such as the Xbox Elite Wireless Controller and the PS4 DualShock, are popular input devices for consoles and desktop.

> **NOTE**
> Stride is currently optimized for the Xbox Elite gamepad. Other controllers work, but might have unexpected button mappings. Gamepad-specific features like the PS4 DualShock touchpad aren't supported.

Digital and analog buttons

- **Digital** buttons have two states: **up** and **down**. The D-pad, Start, Back, Thumbstick (press), A, B, X and Y buttons are digital buttons.
- **Analog** buttons return a value depending on how hard the user presses. The triggers are analog buttons, and return a value between 0 and 1. The thumbsticks are also analog, and return values between -1 and 1 on the X and Y axes.

The Xbox Elite controller buttons have the following names in Stride:

Handle gamepad input

Check that gamepads are connected

Before handling gamepad input:

- To check if any gamepads are connected, use InputManager.HasGamePad.
- To check how many gamepads are connected, use InputManager.GamePadCount.
- To check if the current device has been disconnected, use the InputManager.DeviceRemoved event.
- To check if a device has been connected, use the InputManager.DeviceAdded event.

Digital buttons

To query the states and state changes of digital gamepad buttons, on the `GamePad` object, call:

Method	Functionality
IsButtonDown(IGamePadDevice, GamePadButton)	Checks whether the button is in the *down* state.
IsButtonPressed(IGamePadDevice, GamePadButton)	Checks whether the user has *pressed* the button since the previous update.
IsButtonReleased(IGamePadDevice, GamePadButton)	Checks whether the user has *released* the button since the previous update.

Button (GamePadButton) is the gamepad button you want to check.

You can also get the state of digital buttons using GamePadState.Buttons.

> **NOTE**
>
> The GamePadState.Buttons field is a bitmask that uses binary system. Depending on the bitmask value, you can determine which buttons are *up* or *down*.

To get the gamepad state, use IGamePadDevice.State.

Analog buttons

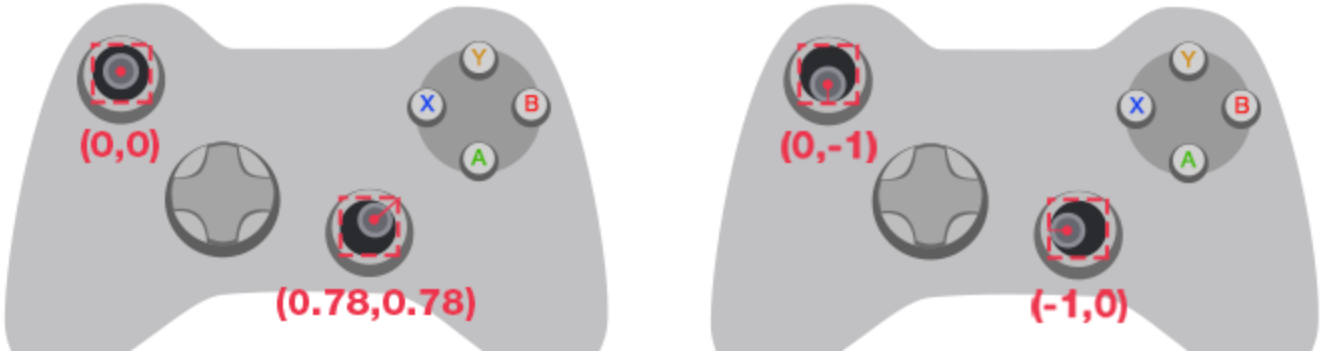
To query values of analog buttons, first get the current state of gamepad using GetGamePadByIndex(index), where *index (Integer)* is the index of the gamepad you want to check.

> **WARNING**
>
> The value returned by IGamePadDevice.State is the state of the gamepad at the **current** update. You can't reuse this value for the next updates. You have to query it again in every update.

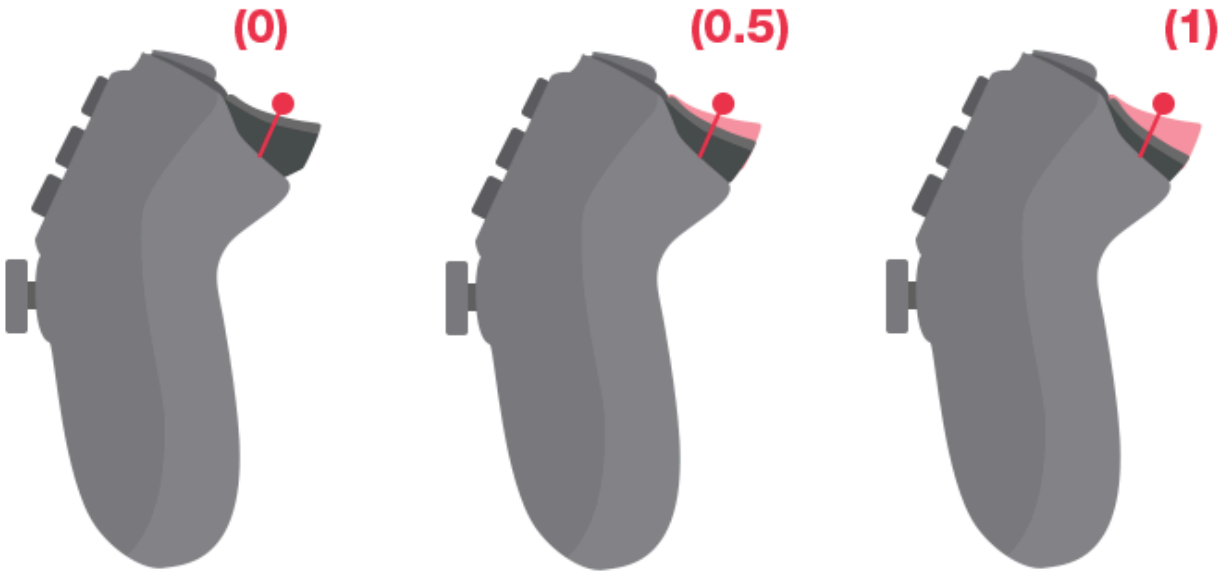
To get trigger and thumbstick positions, use these GamePadState fields:

Field	Description
GamePadState.LeftThumb	Left thumbstick X-axis/Y-axis value in the range [-1.0f, 1.0f] for both axes.
GamePadState.RightThumb	Right thumbstick X-axis/Y-axis value in the range [-1.0f, 1.0f] for both axes.
GamePadState.LeftTrigger	Left trigger analog control value in the range [0, 1.0f] for a single axes.
GamePadState.RightTrigger	Right trigger analog control value in the range [0, 1.0f] for a single axis.

Thumbsticks move along the X and Y axes. Their positions read as follows:



Triggers move along the X axis. Their positions read as follows:



Vibration

To set the gamepad vibration level, use IGamePadDevice.SetVibration.

> **NOTE**
>
> Stride currently only supports vibration for Xbox gamepads.

Example code

```
using Stride.Core.Mathematics;
using Stride.Engine;

public class TestScript : SyncScript
{
    public override void Update()
    {
        //Check if a gamepad is connected
        if (Input.HasGamePad)
        {
            //Get the number of connected gamepads
            int gamepadCount = Input.GamePadCount;

            // Check each gamepad's status
            foreach(var gamepad in Input.GamePads)
            {
```

```
                // Get the analog thumbstick positions
                Vector2 speed = gamepad.State.LeftThumb;
                Vector2 direction = gamepad.State.RightThumb;

                // Get the digital buttons' status
                if (gamepad.IsButtonDown(GamePadButton.X))
                {
                    // The action repeats for as long as the user holds the button down.
                    // This is useful for continuous actions such as firing a machine gun.
                }
                if (gamepad.IsButtonPressed(GamePadButton.A))
                {
                    // The action is triggered only once, even if the user holds the
button down.
                    // This is useful for one-time actions such as jumping.
                }
            }
        }
    }
}
```

See also

- Keyboards
- Virtual buttons
- Input overview

Gestures

Intermediate Programmer

Gestures are predefined pointer patterns. Stride can recognize gestures and trigger corresponding events. For example, in a strategy game, the player can drag and drop a unit to the battlefield with a **drag** gesture. Gestures can use one or several fingers.

> **NOTE**
> All lengths, speeds and error margins of configuration files must use normalized values.

Turn on gesture recognition

By default, the input system doesn't recognize gestures, as this requires CPU time.

To turn on gesture recognition:

1. Create an instance of the configuration class for the gesture you want to recognize. For example, for the drag gesture, create an instance of GestureConfigDrag.
2. Configure the class parameters.
3. Add the gesture configuration to the Gestures collection.

> **WARNING**
> After you activate recognition for a gesture, you can't modify the gesture's parameters. If you need to do this, delete the gesture from the Gestures collection and create a new entry with new parameters.

Turn off gesture recognition

Delete the gesture from the InputManager.Gestures collection.

Gesture recognition

When the input system detects a gesture, it adds a GestureEvent to the list of InputManager.Gesture Events. The event contains information about the gesture and its state, such as its location and the number of fingers used.

> **NOTE**
>
> Each gesture has its own associated gesture event class (see below).

The GestureEvent.Type field indicates which gesture has been recognized. You can then cast the base gesture event into the gesture-specific event type to have gesture-type-specific information about the event.

Stride can detect several gestures simultaneously, so the event list can contain more than one item in an update.

The list is cleared with every update, so you don't need to clear it manually.

Configure gestures

In the GestureConfig classes, you can configure parameters including:

- the number of fingers the gesture uses
- the number and duration of taps the gesture uses
- the gesture direction

> **NOTE**
>
> Each gesture has its own configuration class with specific configuration parameters (see below).

Types of gesture

Stride supports two main types of gesture:

- **Discrete** gestures (tap, flick, long press) trigger a single event.
  - Tap
  - Flick
  - Long press
- **Continuous** gestures (drag and composite) trigger a series of events when the user changes the direction of the gesture.
  - Drag

- Composite

Discrete gestures

Tap

The user touches the screen and quickly removes their finger.

Configuration class: GestureConfigTap

Event class: GestureEventTap

The number of fingers on the screen can't vary during the gesture. To set the number of fingers required for a tap, modify RequiredNumberOfFingers.

> **TIP**
>
> To distinguish single taps from multi-taps, the system uses latency in tap events. To disable this, set the GestureConfigTap.MaximumTimeBetweenTaps field to **0**.

Flick

The user touches the screen, performs a quick straight translation, and withdraws their finger(s).

Configuration class: GestureConfigFlick

Event class: GestureEventFlick

The number of fingers on the screen can't during the gesture.

To set a minimum length for the flick gesture, use GestureConfigFlick.MinimumFlickLength.

To restrict the direction of the flick to **vertical** or **horizontal**, use GestureConfigFlick.FlickShape.

Long press

The user touches the screen and maintains pressure without removing their finger for a certain period of time (the default time is one second).

Configuration class: GestureConfigLongPress

Event class: GestureEventLongPress

The number of fingers on the screen can't vary during the gesture.

To change the minimum press length for the long press gesture, modify GestureConfigLongPress.RequiredPressTime.

Continuous gestures

Drag

The user touches the screen, performs a translation, and withdraws their finger(s).

Configuration class: GestureConfigDrag

Event class: GestureEventDrag

The number of fingers on the screen can't vary during the gesture.

To detect smaller drags, decrease GestureConfigDrag.MinimumDragDistance.

To restrict the direction of the drag to **vertical** or **horizontal**, use GestureConfigDrag.DragShape.

Composite

The user touches the screen with two fingers and moves them independently.

Configuration class: GestureConfigComposite

Event class: GestureEventComposite

The composite gesture requires exactly two fingers on the screen. It's triggered when the system detects one of the three basic actions:

- *Translation*: the user translates two fingers together in the same direction.
- *Scale*: the user moves two fingers closer together or further apart.
- *Rotation*: the user rotates two fingers around a center point.

Gesture states

A gesture always has one of four states:

- Began
- Changed
- Ended
- Occurred

Discrete gestures (tap, flick, long press) always have the state *occurred*. **Continuous** gestures (drag and composite) always begin with the state *began*, followed by any *changed* states, and end with the *ended* state.

To query the current state of a gesture, use the GestureEvent.State field of the triggered gesture event.

Example code

Activate or deactivate gesture recognition

To create the configuration of a gesture you want to recognize:

```
// Create the configuration of a gesture you want to recognize.
var singleTapConfig = new GestureConfigTap();
```

```
// Start tap gesture recognition.
Input.Gestures.Add(singleTapConfig);

// Create the configuration of the gesture you want to recognize.
var doubleTapConfig = new GestureConfigTap(2, 1);

// Start double tap gesture recognition.
Input.Gestures.Add(doubleTapConfig);

// Stop tap gesture recognition.
Input.Gestures.Remove(singleTapConfig);

// Stop all gesture recognitions.
Input.Gestures.Clear();
```

Configure the gesture

Each configuration class has a parameterless constructor that corresponds to the default gesture configuration. You can use special constructors for frequently-modified parameters.

> **WARNING**
>
> We don't recommend you modify other fields as this might break the input system. But if you need to, you can modify them using the corresponding properties.

```
// Default gesture config.
var singleTapConfig = new GestureConfigTap();

// Personalize gesture config using the dedicated constructor.
var doubleTapConfig = new GestureConfigTap(2, 2);

// Personalize gesture config by directly accessing the desired property.
// Make sure you know what you're doing! Modifying this might break the input system.
var noLatencyTap = new GestureConfigTap() { MaximumTimeBetweenTaps= TimeSpan.Zero };
```

Access gesture events

You can access the list of events triggered by recognized gestures using the InputManager.Gesture Events collection. The collection is automatically cleared at every update.

```
var currentFrameGestureEvents = Input.GestureEvents;
```

Identify the gesture type

Use the GestureEvent.Type field to identity the gesture type, then cast it to the appropriate event type to get extra information about the event.

```
foreach( var gestureEvent in Input.GestureEvents)
{
        // Determine if the event is from a tap gesture
        if (gestureEvent.Type != GestureType.Tap)
                continue;

        // Cast a specific tap event class.
        GestureEventTap  tapEvent = (GestureEventTap) gestureEvent;

    // Access tap-event-specific field.
    log.Info("Tap position: {0}.", tapEvent.TapPosition);
}
```

Identify the gesture state

Use the GestureEvent.State field to get gesture event state.

```
switch(compositeGestureEvent.State)
{
case GestureState.Began:
        image.ComputePreview();
        break;
case GestureState.Changed:
        image.TransformPreview(compositeGestureEvent.TotalScale,
compositionGestureEvent.TotalRotation);
        break;
case GestureState.Ended:
        image.TransformRealImage(compositeGestureEvent.TotalScale,
compositionGestureEvent.TotalRotation);
        break;
default:
        break;
}
```

See also

- Pointers
- Virtual buttons
- Input overview

Keyboards

Beginner Programmer

The **keyboard** is the most common input device for desktop games. There are two ways to handle keyboard input in Stride:

- query **key states**
- use KeyEvent lists

You can access both from the input base class. For more information about these options, see the input index

Check keyboard availability

Before handling keyboard input, check whether a keyboard is connected using Input.HasKeyboard.

Get key states

You can query **key states** and **state changes** with the following methods:

Method	Description
IsKeyDown(Keys)	Checks if a specified key is in the **down** state.
IsKeyPressed(Keys)	Checks if a specified key has been **pressed** since the last update.
IsKeyReleased(Keys)	Checks if a specified key has been **released** since the last update.

> **NOTE**
> Stride doesn't support retrieving interpreted keys, such as special characters and capital letters.

Get key events

In some cases, you want to know all the keys that are currently *Down*, or all the keys that have been *Pressed* since the last update. The key state API isn't good for this situation, as you have to query each available key separately.

Instead, use the **key events** collections available in the Input base class.

Public List	Description I
InputManager.DownKeys	Gets a list of the keys that were down in the last update.
InputManager.PressedKeys	Gets a list of the keys pressed in the last update.
InputManager.Released Keys	Gets a list of the keys released in the last update.
InputManager.KeyEvents	Gets a list of the key events in the last update (keys pressed or released).

Every KeyEvent has two properties: Key (the affected key) and IsDown (the new state of the key).

Example code

```
public class KeyboardEventsScript : SyncScript
{
        //Declared public member variables and properties show in Game Studio.

        public override void Update()
        {
                //Perform an action in every update.
                if (Game.IsRunning)
                {
                        if (Input.IsKeyDown(Keys.Left))
                        {
                                this.Entity.Transform.Position.X -= 0.1f;
                        }
                        if (Input.IsKeyDown(Keys.Right))
                        {
                                this.Entity.Transform.Position.X += 0.1f;
                        }
                }
        }
}
```

See also

- Gamepads
- Mouse
- Virtual buttons
- Input overview

Mouse

Beginner Programmer

The **mouse** is a common input device for desktop games.

There are two ways to handle mouse input in Stride:

- Query **mouse button states**.
- For cross-platform games that target mobile devices, you can use PointerEvent lists. For more information, see Pointers.

You can access **mouse button states** and **pointer events list** from the Input manager.

Class	Project type	When to use
Input Manager	Desktop only	For desktop games, you usually handle input with multiple mouse buttons. This means you should use **mouse button states**.
Pointer Event	Cross-platform	For mobile games, you usually simulate pointers with just the left mouse button. This means you can treat the mouse input like pointers. There's no need to create separate mouse-specific controls. For more information, see Pointers.

For more information about these options, see the Input index.

Check mouse availability

Before handling mouse input, use Input.HasMouse to check if a mouse is connected.

Get the mouse position

You can get the mouse position in normalized or absolute coordinates.

Normalized coordinates

MousePosition returns the mouse pointer position in **normalized** X, Y coordinates instead of actual screen sizes in pixels. This means the pointer position adjusts to any resolution and you don't have to write separate code for different resolutions.

- (0,0): the pointer is in the top-left corner of the screen
- (1,1): the pointer is in the bottom-right corner of the screen

Absolute coordinates

InputManager.AbsoluteMousePosition returns the mouse pointer position in absolute X and Y coordinates (the actual screen size in pixels). For example, if the pointer is in the top-left corner of the screen, the values are (0,0). If the pointer is in the bottom-right corner, the values depends on the screen resolution (eg 1280, 720).

> **TIP**
>
> To get the actual size of the screen, access IPointerDevice.SurfaceSize. For example:

```
var surfaceSize = Input.Mouse.SurfaceSize;
```

Query mouse button state changes

You can use the mouse buttons to trigger actions in a project. For example, in first-person shooter games, the left mouse button is commonly used to shoot.

The Input manager has several methods that check mouse button states (*Pressed*, *Down*, or *Released*):

Method	Description
HasDownMouseButtons	Checks if one or more mouse buttons are currently pressed down.
HasPressedMouseButtons	Checks if one or more mouse buttons were pressed in the last update.
HasReleasedMouseButtons	Checks if one or more mouse buttons were released in the last update.
IsMouseButtonDown (Mouse Button)	Checks if a specified mouse button is currently pressed down.
IsMouseButtonPressed (Mouse Button)	Checks if a specified mouse button was pressed in the last update.
IsMouseButtonReleased (Mouse Button)	Checks if a specified mouse button was released in the last update.

Mouse delta

Use InputManager.MouseDelta to get the change in mouse position in normalized coordinates since the last update. You can use this to analyze mouse movement speed and direction.

Mouse wheel delta

You can use the mouse wheel to trigger actions in a project. For example, in a first-person shooter game, moving the mouse wheel might switch weapons or zoom a camera.

The InputManager.MouseWheelDelta returns a positive value when the user scrolls forwards and a negative value when the user scrolls backwards. A value of `0` indicates no movement.

Lock the mouse position

For some projects, the user needs to move the mouse cursor beyond the borders of the screen. For example, first-person shooter games usually need 360-degree camera rotation. In these cases, you also probably want the mouse cursor to be hidden.

You can lock the mouse position and hide the cursor with the following properties and methods:

Method or property	Description
LockMousePosition(Boolean)	Locks the mouse position until the next call to the UnlockMousePosition() event.
UnlockMousePosition()	Unlocks the mouse position locked by the LockMousePosition(Boolean) event.
IsMousePositionLocked	Checks if the mouse position is locked.

> **TIP**
>
> You can get or set mouse visibility with GameWindow.IsMouseVisible.

Example code

```
public class MouseInputScript : SyncScript
{
        public override void Update()
        {
                //If the left mouse button is pressed in this update, do something.
                if (Input.IsMouseButtonDown(MouseButton.Left))
                {
                }
                //If the middle mouse button has been pressed since the last update, do
something.
                if (Input.IsMouseButtonPressed(MouseButton.Middle))
```

```
                {
                }

                //If the mouse moved more than 0.2 units of the screen size in X direction,
do something.
                if (Input.MouseDelta.X > 0.2f)
                {
                }

        }
}
```

See also

- Pointers
- Virtual buttons
- Keyboard
- Gamepads
- Input overview

Pointers

Beginner Programmer

Pointers are points on the device screen corresponding to **finger touches**. Devices with multi-touch functionality support multiple simultaneous pointers.

On desktop platforms, the left mouse button can be used to simulate pointers. For more information about mouse input, see Mouse.

How Stride processes pointer input

1. The user touches the screen or clicks the left mouse button.
2. Stride creates a pointer.
3. Stride assigns **pointer ID** to that pointer corresponding to a given finger.
4. Every time the pointer is modified, Stride creates a new **pointer event** with that pointer.
5. For each new finger, Stride creates a new pointer with a new pointer ID.

> **NOTE**
> Each pointer event contains information about only one pointer. If several pointers are modified simultaneously in the same update, Stride creates a separate event for each pointer.

> **WARNING**
> Each OS handles pointer modifications differently. This means the same finger gesture can generate slightly different pointer event sequences across different platforms. For example, Android doesn't create a new pointer event when a finger touches the screen but doesn't move. For more information, check your OS documentation.

You can enable gesture recognition to detect gestures such as long presses and taps. For more information, see Gestures.

The PointerEvent class

PointerEvent reports pointer events. It contains the current **pointer status** and time information. It is thrown every time the **pointer** is modified.

You can access the list of **pointer events** since the last update using InputManager.PointerEvents. Stride lists pointer events in chronological order. The list is cleared at every update, so you don't need to clear it manually.

Get pointer information

You can use the following properties to get information about the pointer that triggered the event:

Property	Description
PointerEvent.PointerId	Indicates the ID of the pointer which triggered the event.

> **WARNING**
>
> The ID of a pointer is valid only during a single *Pressed->Moved->Released* sequence of pointer events. A finger can have different IDs each time it touches the screen (even if this happens very quickly).

> **WARNING**
>
> Each OS has its own way of assigning IDs to pointers. There's no relation between the pointer ID values and corresponding fingers.

To check if a pointer event was triggered by a mouse or touch, use:

```
bool isTriggeredByMouse = event.Pointer is IMouseDevice
```

Get the pointer position

You can get the pointer position in normalized or absolute coordinates.

Normalized coordinates

Position returns the pointer position in **normalized** X and Y coordinates instead of actual screen sizes in pixels. This means the pointer position adjusts to any resolution and you don't have to write separate code for different resolutions.

- (0,0): the pointer is in the top-left corner of the screen
- (1,1): the pointer is in the bottom-right corner of the screen

Absolute coordinates

PointerEvent.AbsolutePosition returns the pointer position in absolute X and Y coordinates (the actual screen size in pixels). For example, if the pointer is in the top-left corner of the screen, the values are (0,0). If the pointer is in the bottom-right corner, the values depends on the screen resolution (eg 1280, 720).

> **TIP**
>
> To get the actual size of the screen, access IPointerDevice.SurfaceSize. For example:

```
var surfaceSize = Input.Pointer.SurfaceSize;
```

Get pointer events

Use the PointerEvent.EventType to check the pointer events.

There are five types of pointer event:

- **Pressed**: The finger touched the screen.
- **Moved**: The finger moved along the screen.
- **Released**: The finger left the screen.
- **Canceled**: The pointer sequence was canceled. This can happen when the application is interrupted; for example, a phone app might be interrupted by an incoming phone call.

> **NOTE**
>
> A sequence of pointer events for one pointer always starts with a **Pressed** event. This might be followed by one or more **Moved** events, and always ends with a **Released** or **Canceled** event.

Get delta values

PointerEvent.DeltaTime gets the time elapsed from the previous PointerEvent.

You can get the delta position in normalized or absolute coordinates.

Normalized delta values

PointerEvent.DeltaPosition gets the change in position since the previous PointerEvent in **normalized** X,Y coordinates.

> **NOTE**
>
> Delta values are always nulls at the beginning of the sequence of pointer events (ie when the **pointer state** is **down**).

Absolute delta values

PointerEvent.DeltaPosition gets the change in position since the previous PointerEvent in **absolute** (X,Y) coordinates.

Example code

This script tracks the pointer movement and prints its positions:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Stride.Core.Mathematics;
using Stride.Engine;

namespace Stride.Input.Tests
{
    public class PointerTestScript : AsyncScript
    {
        public override async Task Execute()
        {
            var pointerPositions = new Dictionary<int, Vector2>();
            while (true)
            {
                await Script.NextFrame();
                foreach (var pointerEvent in Input.PointerEvents)
                {
                    switch (pointerEvent.EventType)
                    {
                    case PointerEventType.Pressed:
                        pointerPositions[pointerEvent.PointerId] = pointerEvent.Position;
                        break;
                    case PointerEventType.Moved:
                        pointerPositions[pointerEvent.PointerId] = pointerEvent.Position;
                        break;
                    case PointerEventType.Released:
                    case PointerEventType.Canceled:
                        pointerPositions.Remove(pointerEvent.PointerId);
```

```
                        break;
                    default:
                        throw new ArgumentOutOfRangeException();
                    }
                }
                var positionsStr = pointerPositions.Values.Aggregate("", (current, pointer)
=> current + (pointer.ToString() + ", "));
                Log.Info("There are currently {0} pointers on the screen located at {1}",
pointerPositions.Count, positionsStr);
            }
        }
    }
}
```

See also

- Gestures
- Mouse
- Virtual buttons
- Input overview

Sensors

Intermediate Programmer

You can use various **sensors**, such as gyroscopes and accelerometers, as input devices in your project. Sensors are often used in mobile games.

Use InputManager to access sensors and:

- check if a sensor is supported by Stride
- disable a sensor
- retrieve sensor data

Stride can receive data from six types of sensor:

- Orientation
- Accelerometer
- UserAcceleration
- Gravity
- Compass
- Gyroscope

They inherit from ISensorDevice.

Stride creates a default instance for each sensor type. You can access each instance from the Input Manager.

Sensors are state-based. Each sensor instance is automatically updated every frame, and contains the value of the sensor just before the update.

For example, to access the accelerometer, use:

```
var accelerometer = Input.Accelerometer;
```

Check if a sensor is available

Before you get the value of a sensor, check that the sensor is available in the device (ie not null). For example, to check if the compass is available:

```
var hasCompass = Input.Compass != null;
```

> **NOTE**
>
> If a sensor isn't natively supported by the device, Stride tries to emulate it using the device's other sensors.

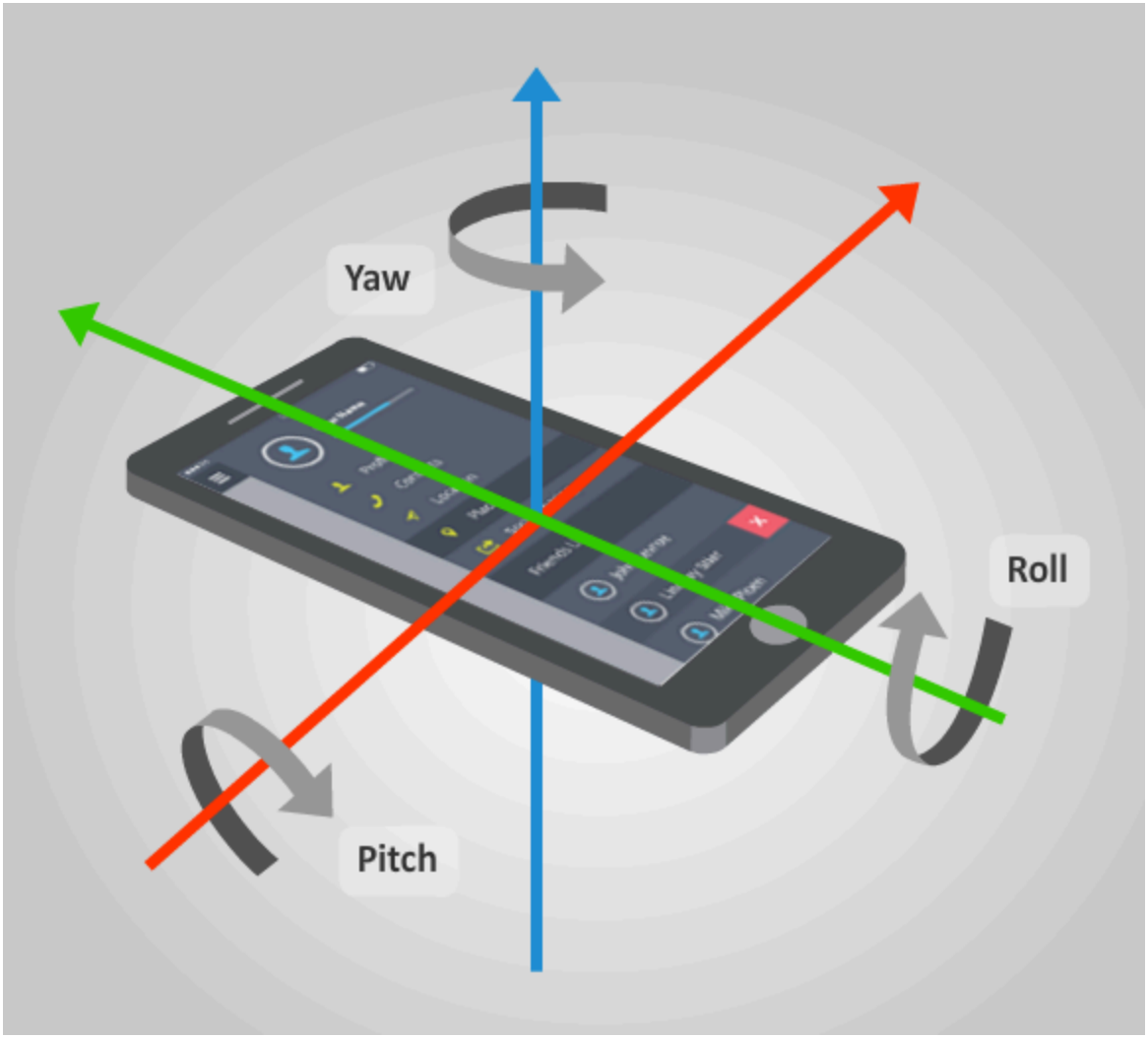
Enable a sensor

By default, Stride disables all available sensors, as retrieving and updating sensor data takes significant CPU time.

To enable a sensor, set IsEnabled to `true`. When you don't need the sensor, disable it by setting the property to `false`.

Use the orientation sensor

The **orientation sensor** indicates the **orientation of the device** with respect to gravity and the Earth's north pole. The orientation is null when the device's Y-axis is aligned with the magnetic north pole and the Z-axis with the gravity. You can use orientation data to control various actions in a game.



Use Input.Orientation to get the current orientation of the device.

Property	Description	Declaration
Roll	The rotation around the X-axis	`public float Roll { get; }`
Pitch	The rotation around the Y-axis	`public float Pitch { get; }`
Yaw	The rotation around the Z-axis	`public float Yaw { get; }`
Rotation Matrix	The device rotation	`public Matrix RotationMatrix { get; }`
Quaternion	The device orientation and rotation	`public Quaternion Quaternion { get; }`

For example:

```
var orientation = Input.Orientation.Quaternion;
```

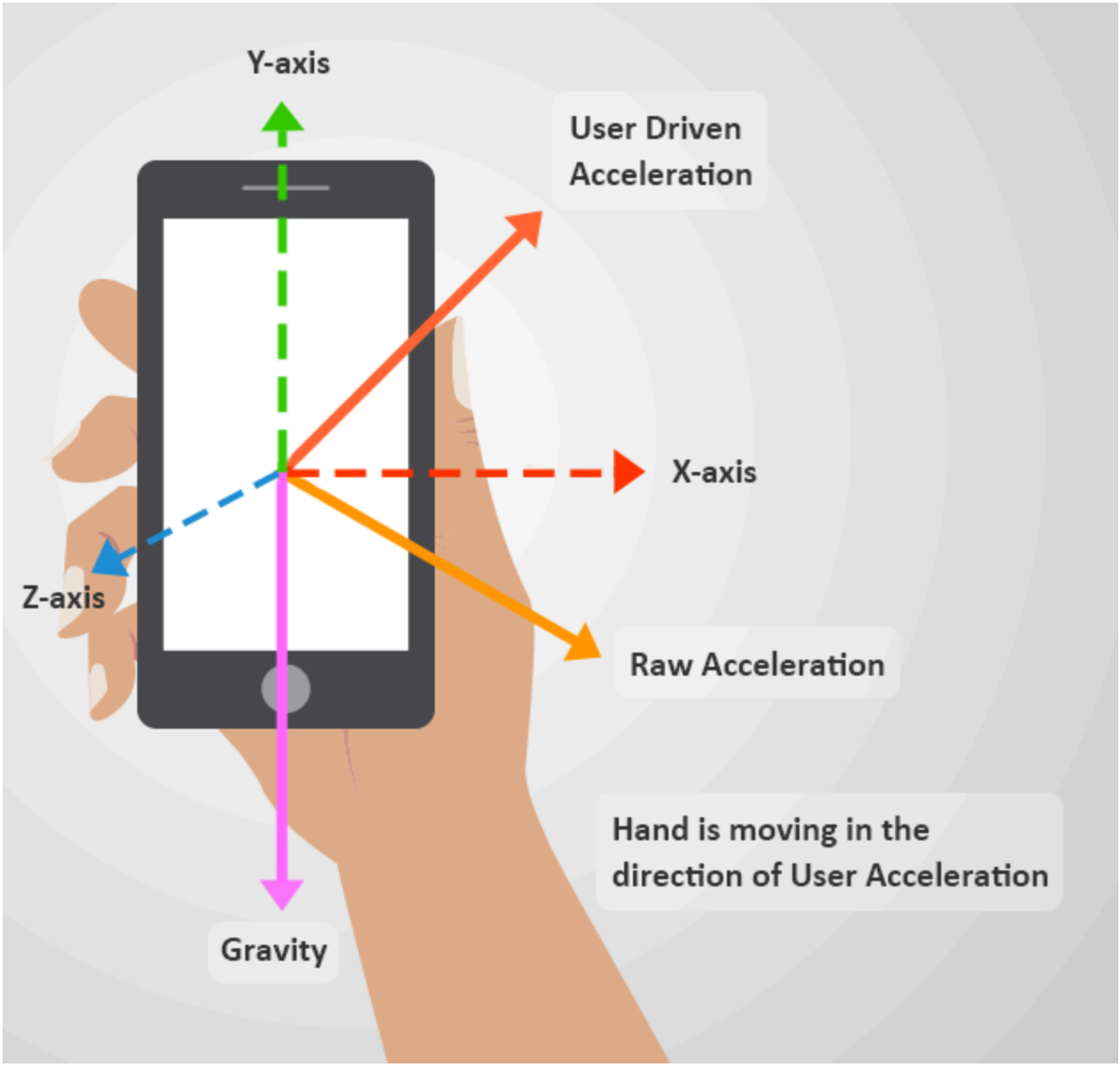
> **NOTE**
> Stride provides the orientation under the pitch/yaw/roll, rotation matrix, or quaternion forms. We recommend the quaternion form as it doesn't suffer from gimbal lock.

Motion sensors

Motion sensors measure **acceleration forces** such as tilts, shakes, and swing. Stride supports three types of motion sensor:

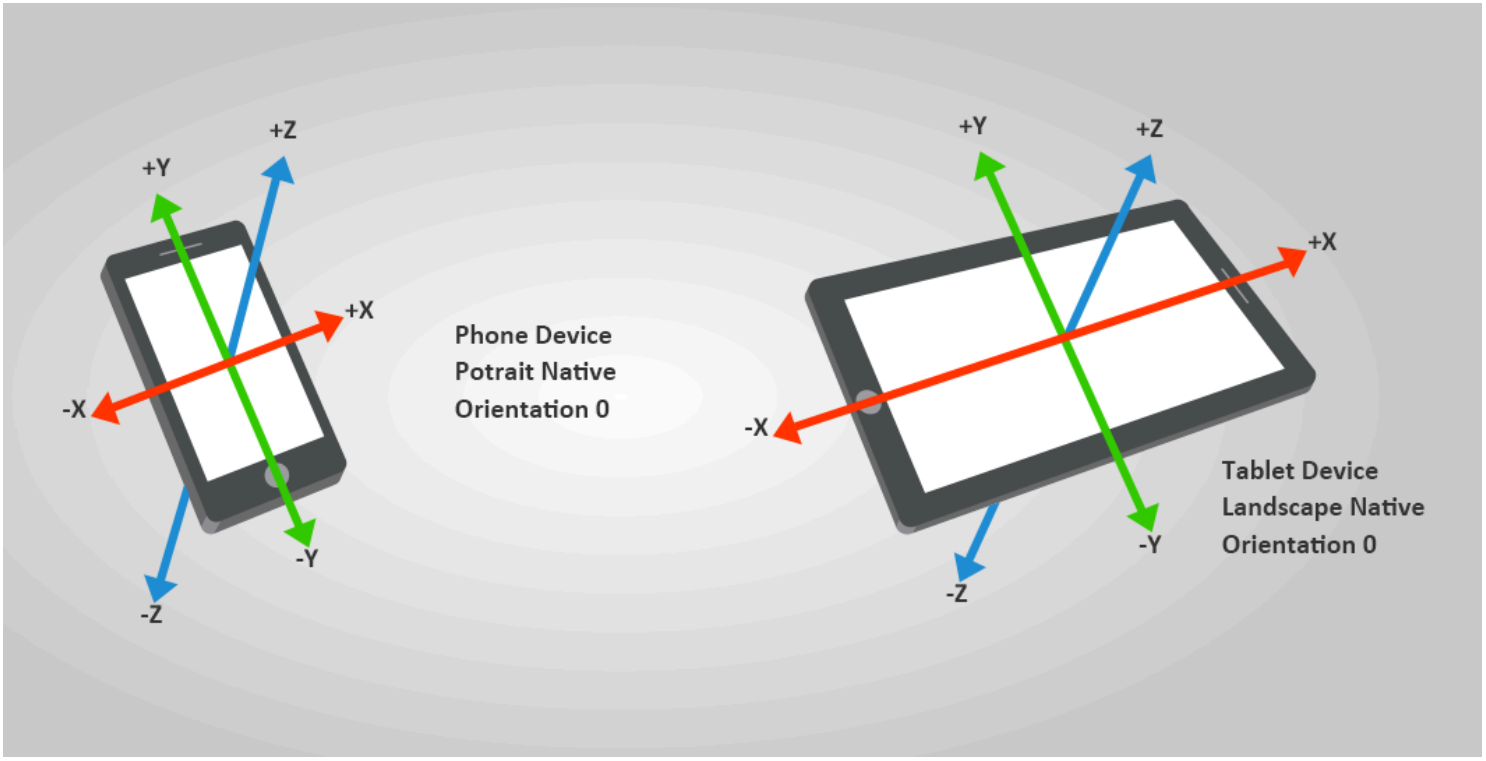
- **Accelerometer**: measures the **raw acceleration**
- **Gravity**: measures gravity only
- **UserAcceleration**: measures only the acceleration applied by the user

The sensors use the physic relation `Accelerometer = Gravity + UserAcceleration`.



Motion sensors have a single field that specifies the current **acceleration vector** on the device. Stride measures the acceleration in **meters per second squared**.

This image shows the **coordinate basis** Stride uses to measure acceleration on smartphones and tablets:



Use the accelerometer

The **accelerometer** measures the raw acceleration applied to the device. This includes **gravity** and **user acceleration**.

> **NOTE**
> When the user isn't applying force, the **device acceleration** is equal to its **gravity**.

To get the raw acceleration, use Accelerometer.Acceleration. For example:

```
var acceleration = Input.Accelerometer.Acceleration;
```

Use the user acceleration sensor

The **user acceleration sensor** is similar to the accelerometer, but measures the acceleration applied **only** by a user (without gravitational acceleration).

To get the user acceleration, use UserAcceleration.Acceleration. For example:

```
var userAcceleration = Input.UserAcceleration.Acceleration;
```

Use the gravity sensor

The gravity sensor gives a 3D vector indicating the direction and magnitude of gravity (meters per second squared) acting on the device.

To get the gravity vector, use GravitySensor. For example:

```
var gravityVector = Input.Gravity.Vector;
```

Use the compass sensor

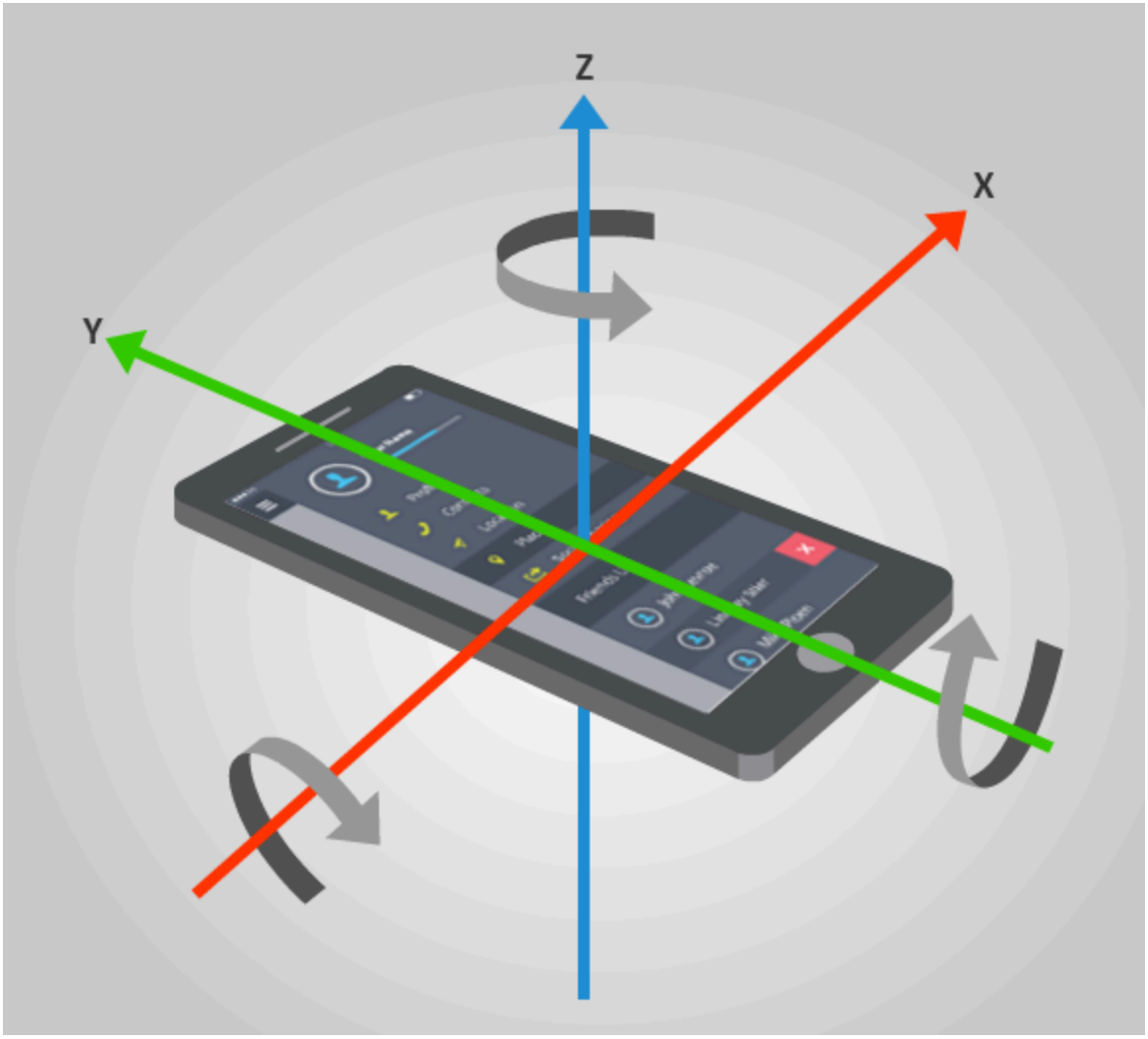
The **compass** indicates measures the angle between the top of the device and the **North Pole**. This is useful, for example, to rotate and align digital maps.

To get this angle, use CompassSensor.Heading. For example:

```
var heading = Input.Compass.Heading;
```

Use the gyroscope

The gyroscope measures the **rotation speed** of the device (**radians per second**).



To get the rotation speed, use GyroscopeSensor.RotationRate. For example:

```
var rotationRate = Input.Gyroscope.RotationRate;
var rotationSpeedX =  rotationRate.X;
var rotationSpeedY =  rotationRate.Y;
var rotationSpeedZ =  rotationRate.Z;
```

Example code

```
public class SensorScript : AsyncScript
{
        public override async Task Execute()
        {
                // Check availability of the sensor
                if(Input.Accelerometer != null)
                        return;

                // Activate the sensor
                Input.Accelerometer.IsEnabled = true;
```

```
                while (Game.IsRunning)
                {
                        // read current acceleration
                        var accel = Input.Accelerometer.Acceleration;

                        // perform require works...
                        await Script.NextFrame();
                }
                // Disable the sensor after use
                Input.Accelerometer.IsEnabled = false;
        }
}
```

See also

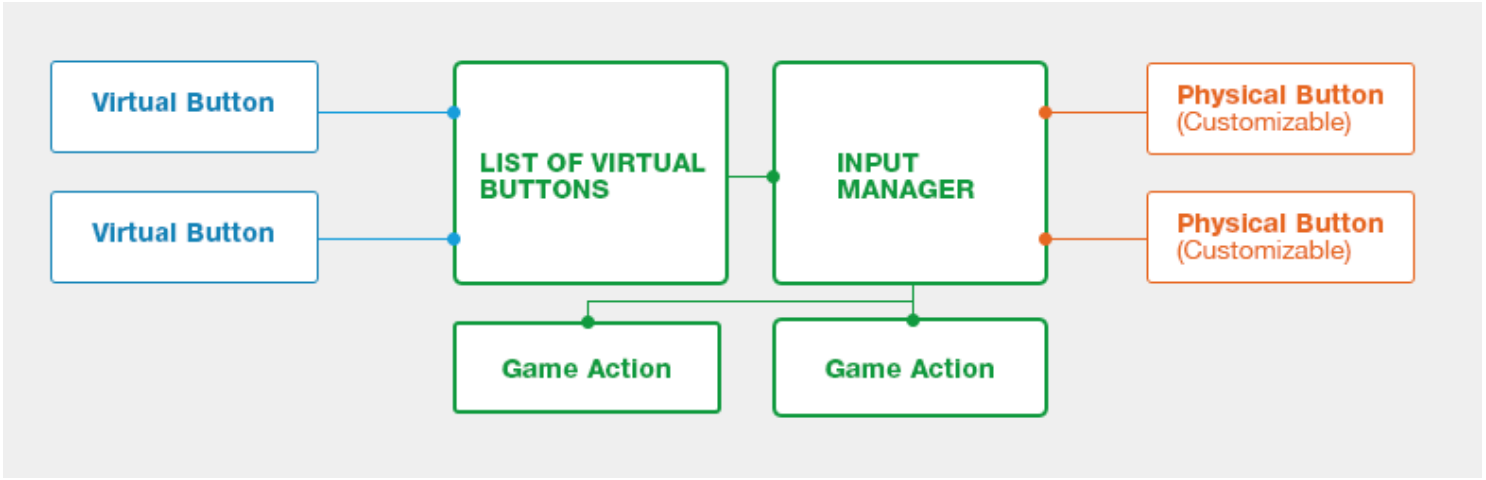
- Gestures
- Pointers
- Input overview

Virtual buttons

Intermediate Programmer

Rather than bind controls to physical keys and buttons, you can bind them to **virtual buttons**. Players can then assign physical buttons to the virtual buttons, allowing them to create their own control schemes.

For example, imagine you develop a first-person shooter game and need to assign a key for the *UseMedkit* function. Rather than bind the function to a particular key, you can create a **virtual button** called *UseMedkit*, then bind the virtual button to, say, the **F** key. If they want to, the player can then bind the virtual key to a different key at runtime.



Use virtual buttons

1. Bind a key, button, or pointer to a virtual button (eg *MyButton*).
2. Create a list of virtual buttons.
3. Add *MyButton* to the list of virtual buttons.
4. Assign a function to *MyButton*.
5. Create additional virtual buttons.
6. Add the additional buttons to the same list, or create additional lists.

Example code

```
public override void Start()
{
    base.Start();

    // Create a new VirtualButtonConfigSet if none exists.
    Input.VirtualButtonConfigSet = Input.VirtualButtonConfigSet ??
new VirtualButtonConfigSet();
```

```
    //Bind "M" key, GamePad "Start" button and left mouse button to a virtual
button "MyButton".
    VirtualButtonBinding b1 = new VirtualButtonBinding("MyButton",
VirtualButton.Keyboard.M);
    VirtualButtonBinding b2 = new VirtualButtonBinding("MyButton",
VirtualButton.GamePad.Start);
    VirtualButtonBinding b3 = new VirtualButtonBinding("MyButton",
VirtualButton.Mouse.Left);

    VirtualButtonConfig c = [b1, b2, b3];

    Input.VirtualButtonConfigSet.Add(c);
}

public override void Update() {
    float button = Input.GetVirtualButton(0, "MyButton");
}
```

See also

- Gamepads
- Keyboard
- Mouse
- Pointers
- Input overview

Navigation

Beginner Level designer Programmer

You can use the **navigation** system to control how characters and other objects navigate scenes.

Set up navigation

1. Create a navigation group
2. Add a navigation mesh
3. Add a navigation bounding box
4. Add a navigation component

Sample project

For an example of how to implement navigation, including enabling and disabling dynamic navigation at runtime, see the **top-down RPG** sample project included with Stride.

See also

- Dynamic navigation

Navigation groups

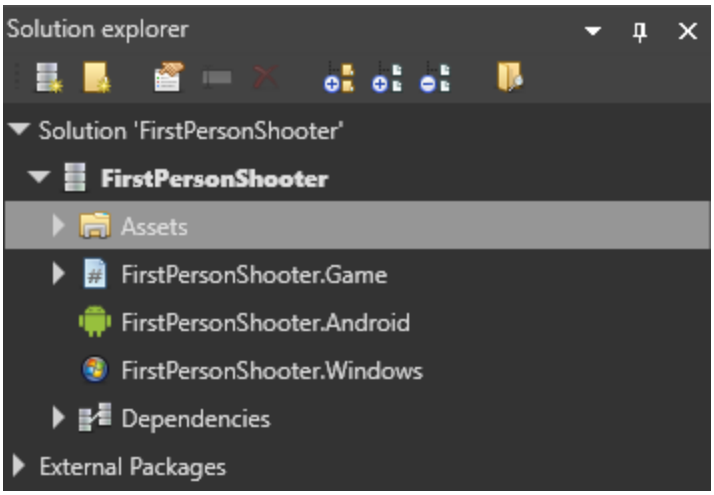
Beginner Level designer Programmer

Navigation groups define different navigation properties for the entities you add to them. You define navigation groups in the project **game settings**.

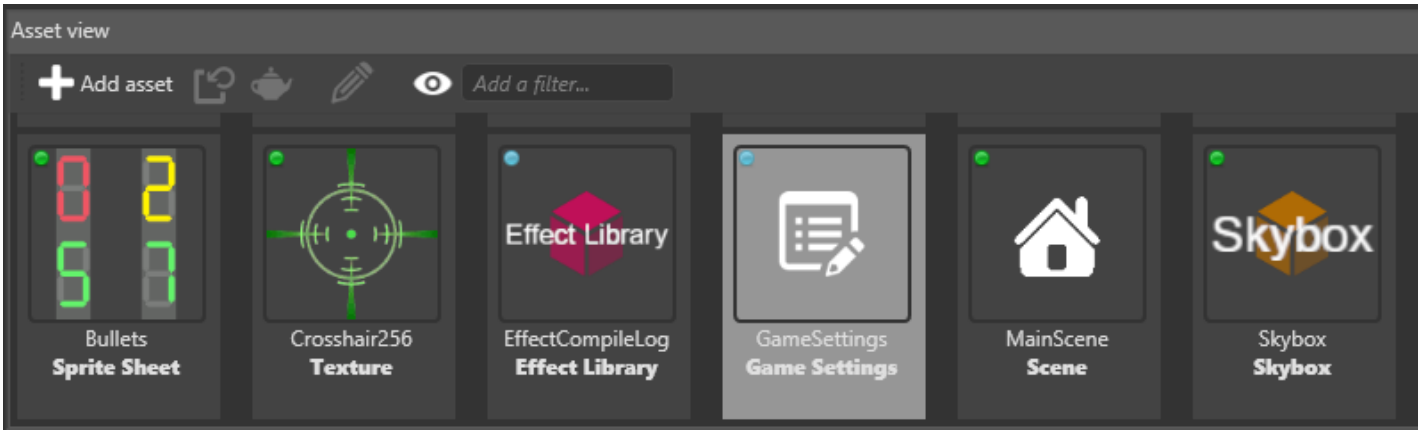
You can create different navigation groups for different kinds of entity. For example, if your game features vehicles controlled by scripts, you might create different navigation groups for different sizes of vehicle, each with different properties: a car group, a bus group, a motorcycle group, and so on.

Create a navigation group

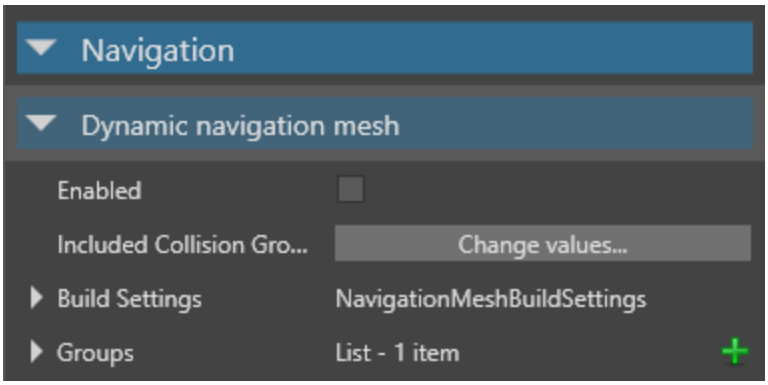
1. In the **Solution Explorer** (the bottom-left pane by default), select the **Assets folder**.



2. In the **Asset View** (the bottom pane by default), select the **Game Settings** asset.

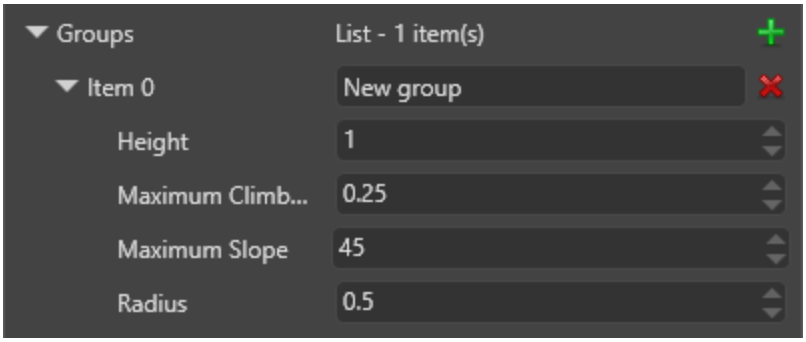


3. In the **Property Grid** (the right-hand pane by default), expand **Navigation Settings**.



4. Next to **Groups**, click (**Add**).

   Game Studio adds a new item to the list of navigation groups.



5. Set the properties for the navigation group. Entities you add to this group use these properties.

Navigation group properties

In most cases, the navigation group properties should approximately match the properties in the character component of the entities in the group, if they have one.

Property	Description
Item	The name of the group
Height	The height of the entities in this group. Entities can't enter areas with ceilings lower than this value
Maximum climb height	The maximum height that entities in this group can climb
Maximum slope	The maximum incline (in degrees) that entities in this group can climb. Entities can't go up or down slopes higher than this value. In most cases, this should be approximately the same value as the **max slope** property in the character component of the entities in this group, if they have one.

Property	Description
Radius	The larger this value, the larger the area of the navigation mesh entities use. Entities can't pass through gaps of less than twice the radius.

See also

- Navigation meshes
- Navigation bounding boxes
- Navigation components
- Dynamic navigation
- Physics — Characters

Navigation meshes

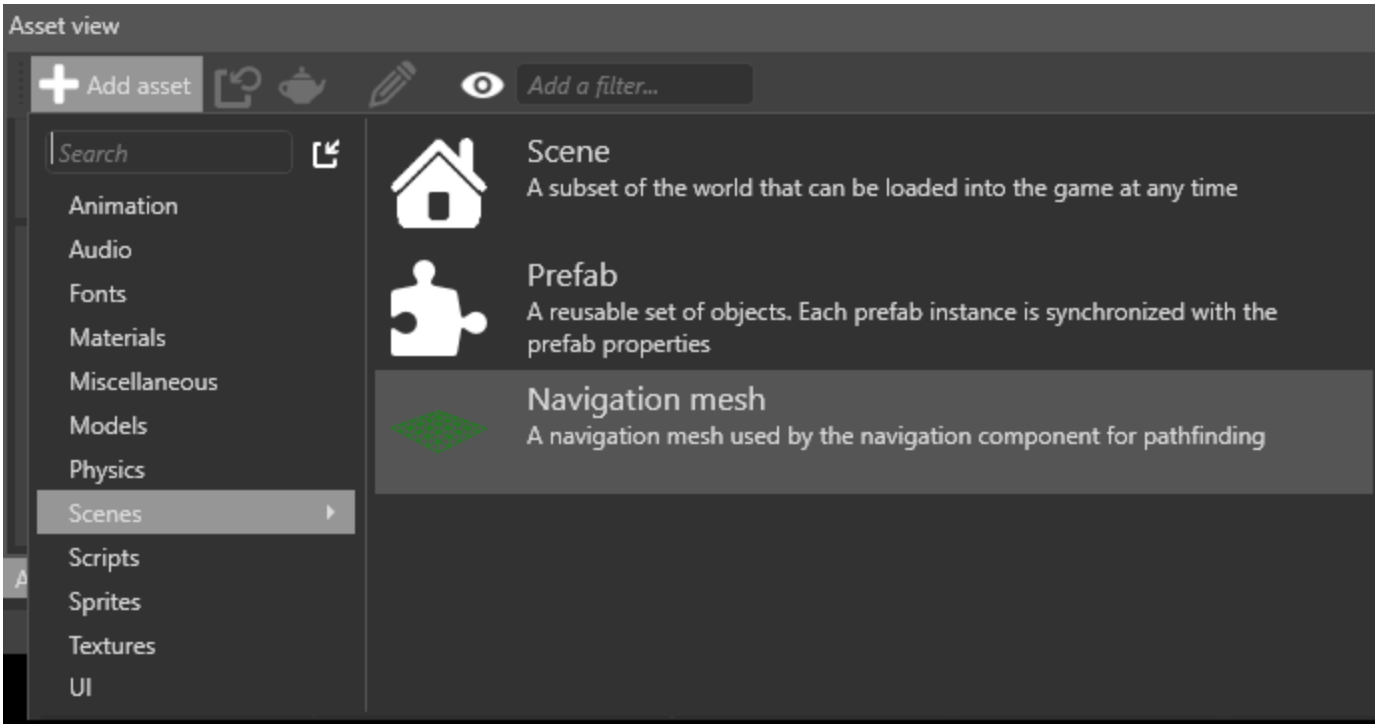
Beginner Level designer Programmer

Navigation meshes form the area that entities with navigation components can navigate. Stride creates a layer in the navigation mesh for each navigation group you create.

Game Studio displays navigation meshes as colored overlays in your scene. The overlay shows where entities in the navigation group for that layer can move. The mesh updates in real time as you edit your scene.

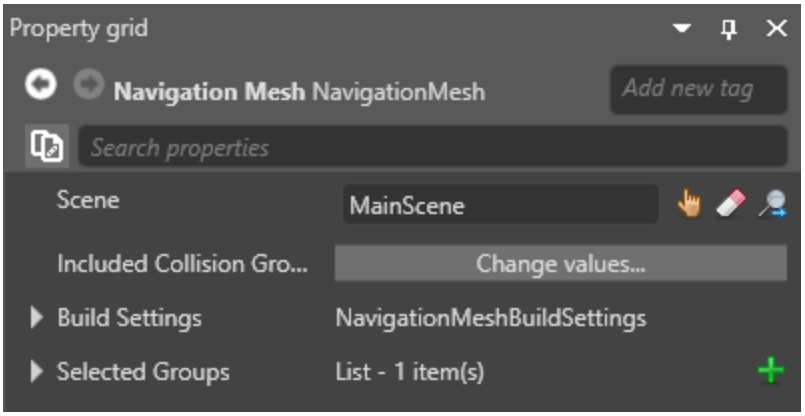
Create a navigation mesh

1. In the **Asset View** (bottom by default), click **Add asset > Scenes > Navigation mesh**.



Game Studio adds a **navigation mesh asset** to your project.

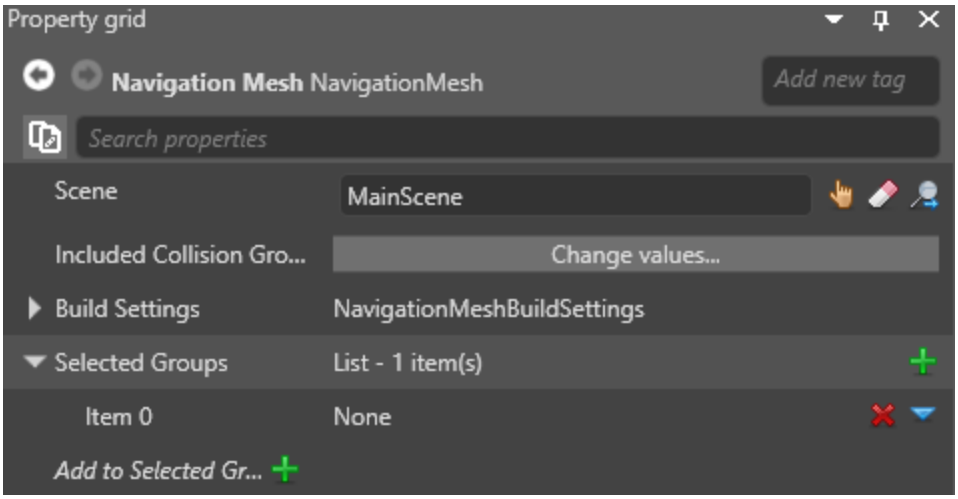
2. With the navigation mesh selected in the **Asset View**, in the **Property Grid**, set the **scene** the navigation meshes in this asset apply to.



   For more information about scenes, see Scenes.

3. Under **Selected groups**, click (**Add**).

   Game Studio adds a new item to the list of groups.



4. Click (**Replace**) and choose a group from the drop-down menu.

   Stride builds a layer in the navigation mesh for this group. For more information about groups, including how to create them, see Navigation groups.

5. Repeat steps 3 and 4 for as many groups as you want to use the navigation mesh.

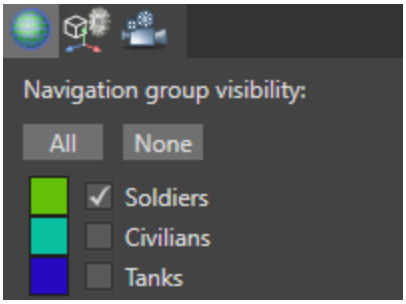
> **NOTE**
> If you want to create a navigation mesh for a different scene, create another navigation mesh asset and select the scene in the asset properties.

Navigation mesh properties

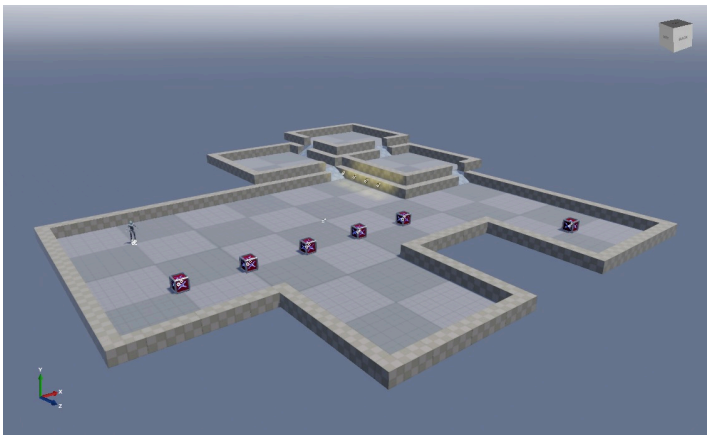
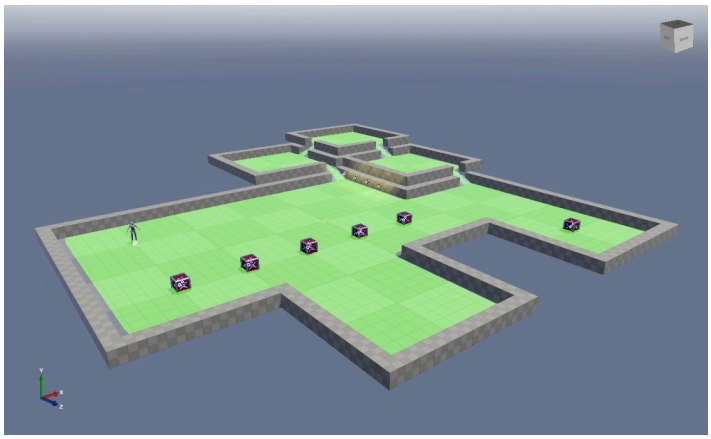
Property	Description
Scene	The scene this navigation mesh applies to
Included collision groups	Set which collision groups the navigation mesh uses. By default, meshes use all collision groups
Build settings	Advanced settings for the navigation mesh
Groups	The groups that use this navigation mesh

Show or hide a navigation mesh in the Scene Editor

Use the **navigation visibility** menu in the Scene Editor toolbar.



To show or hide layers belonging to different groups, use the checkboxes. The colored boxes indicate the color of the groups displayed in the Scene Editor.

Navigation mesh hidden	Navigation mesh shown
	

These options have no effect on runtime behavior.

See also

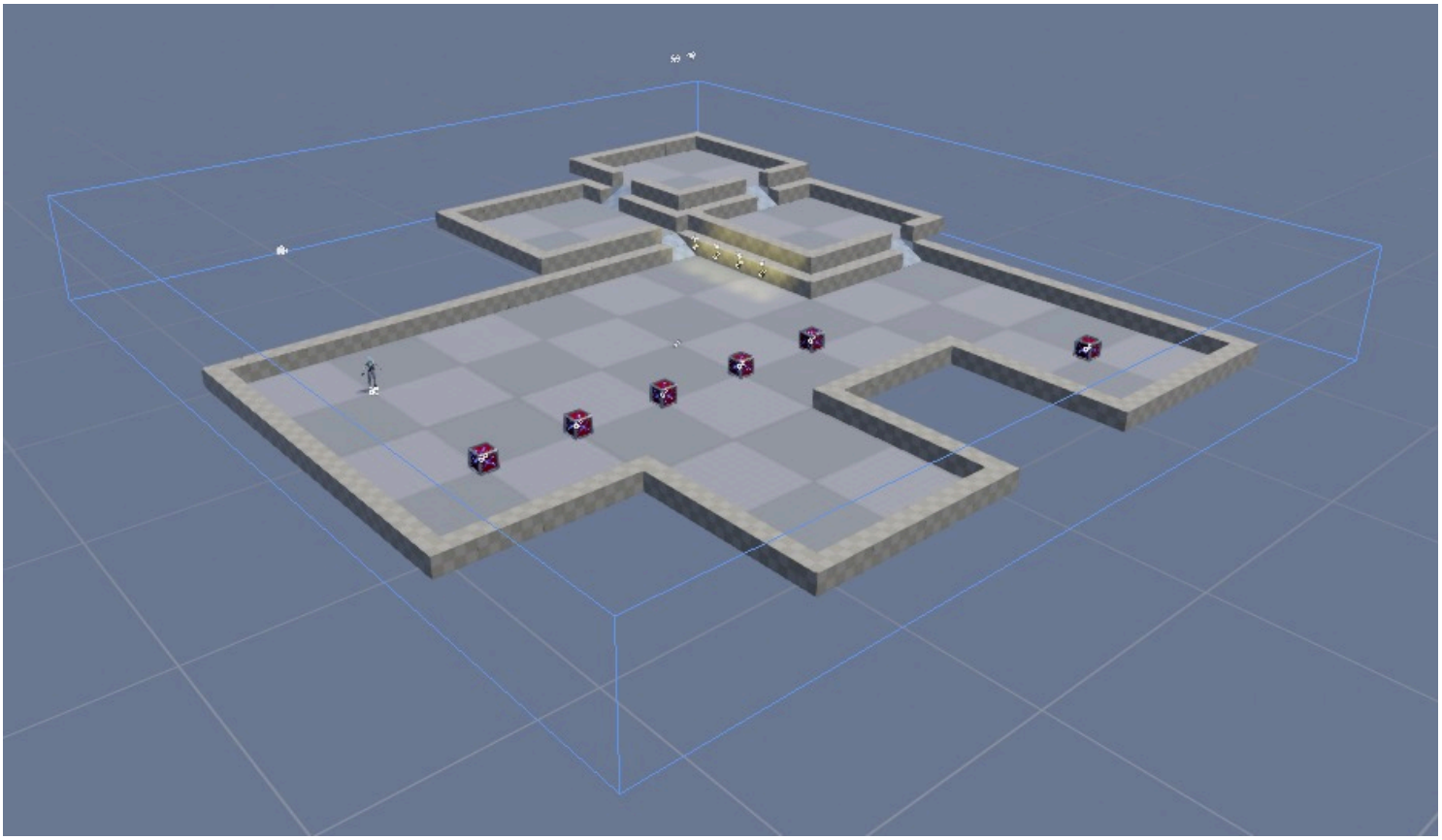
- Navigation groups
- Navigation bounding boxes
- Navigation components
- Dynamic navigation
- Scenes

Navigation bounding boxes

Beginner Level designer Programmer

Navigation bounding boxes define the area that navigation meshes cover. You can use them to create smaller navigation areas in your scene, rather than having a mesh cover the entire scene.

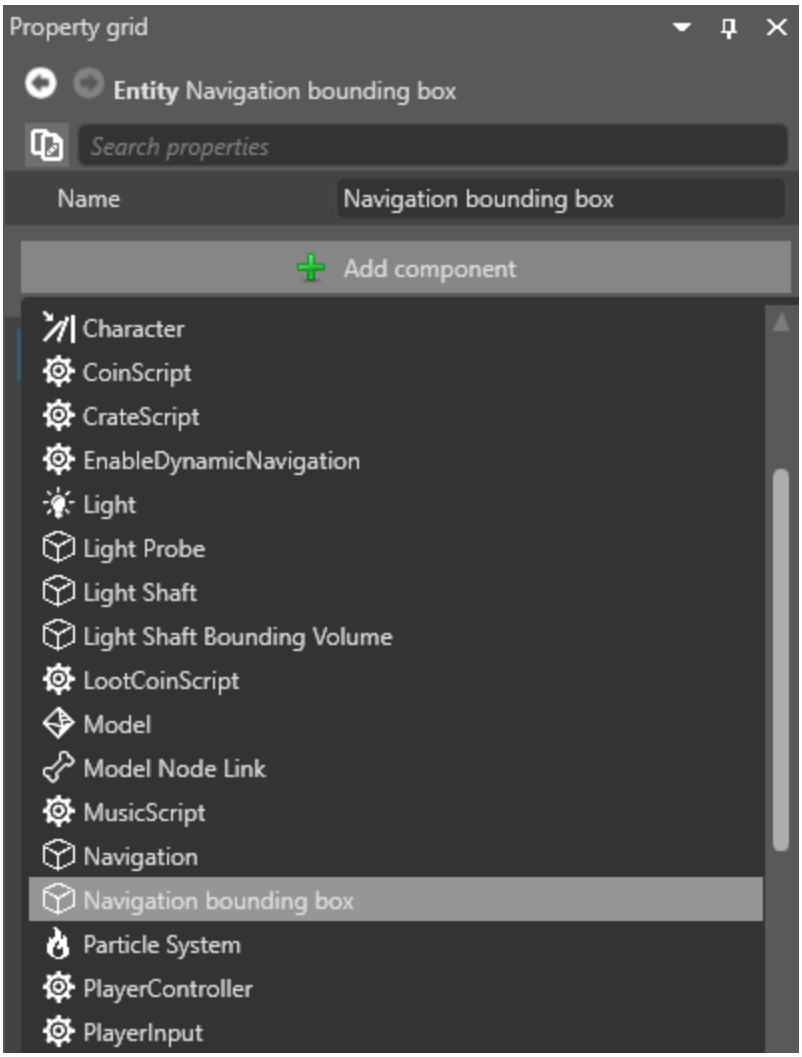
The Scene Editor displays the bounding box as a blue outline.



Create a navigation bounding box

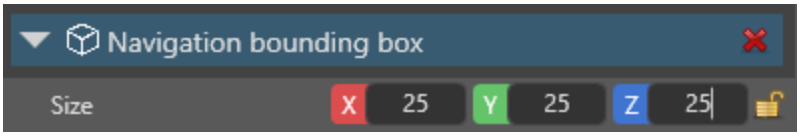
To create a navigation bounding box, add a **navigation bounding box component** to an entity.

1. In the scene, select the entity you want to contain the bounding box, or create a new entity.
2. With the entity selected, in the **Property Grid**, click **Add component** and select **Navigation bounding box**.



Game Studio adds a navigation bounding box to the entity.

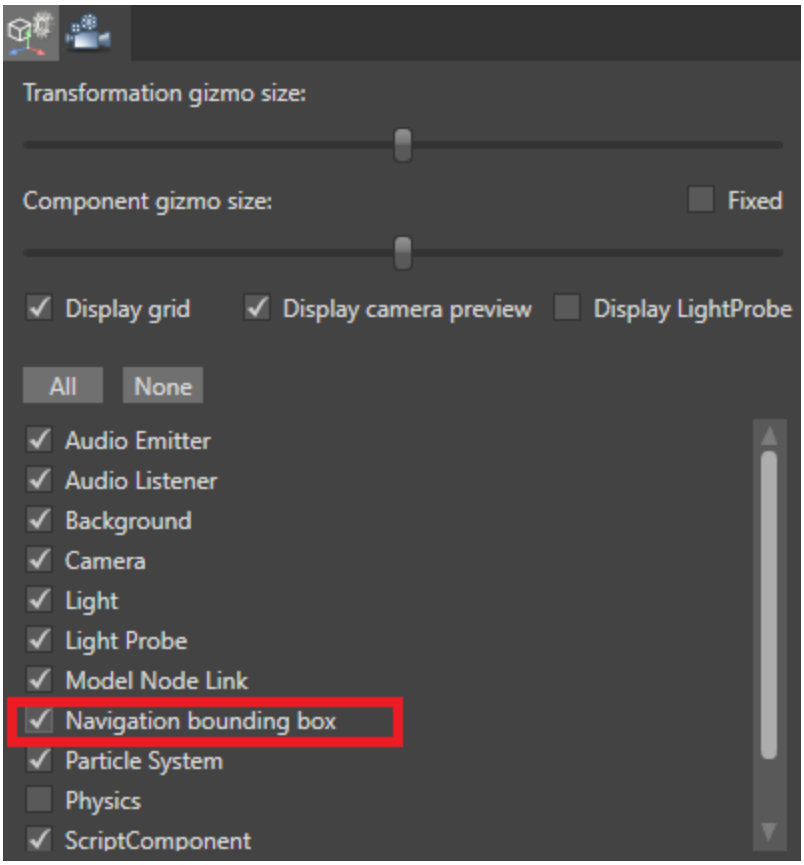
3. Under the **Navigation bounding box** component properties, use the **XYZ** values to set the size of the bounding box.

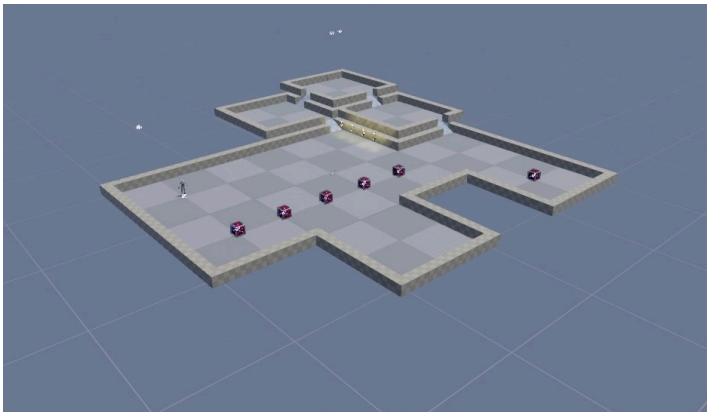
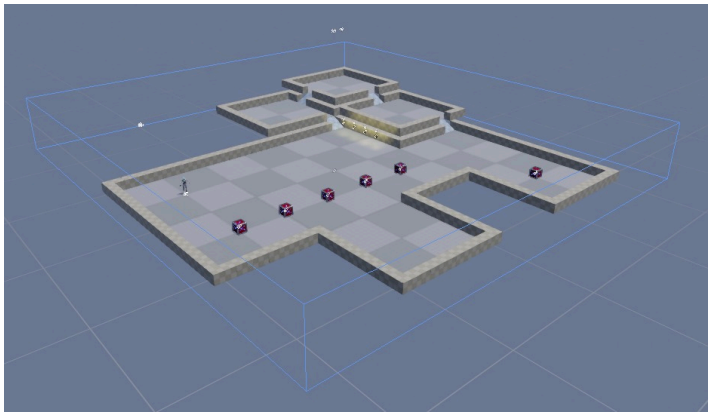


4. Use the entity's **transform component** to position the bounding box in your scene.

Show or hide the bounding box in the Scene Editor

In the Scene Editor toolbar, open the **gizmo options** menu and use the **Navigation bounding box** checkbox.



Bounding box hidden	Bounding box shown (note blue box outline)
	

See also

- Navigation groups
- Navigation meshes
- Navigation components
- Dynamic navigation

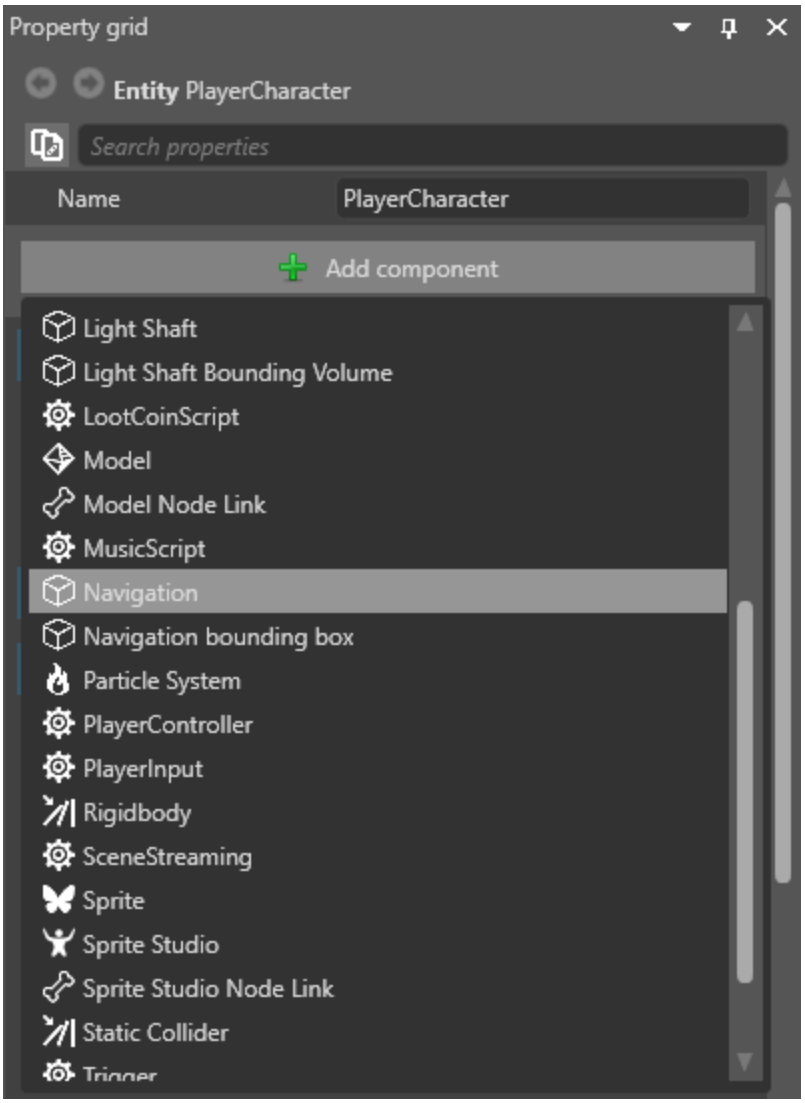
Navigation components

Beginner Level designer Programmer

Navigation components allow entities to use navigation meshes to find paths through the scene. Alternatively, if you enable dynamic navigation in Game Settings, entities can generate their own navigation meshes.

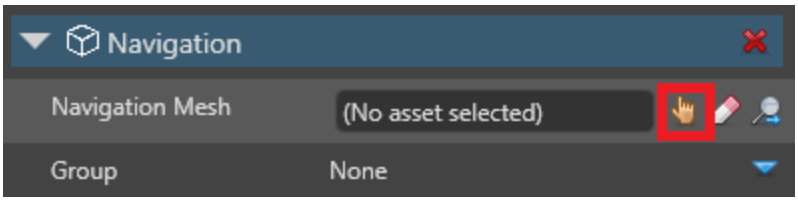
Add a navigation component

1. Select an entity you want to use navigation.
2. In the **Property Grid**, click **Add component** and select **Navigation**.



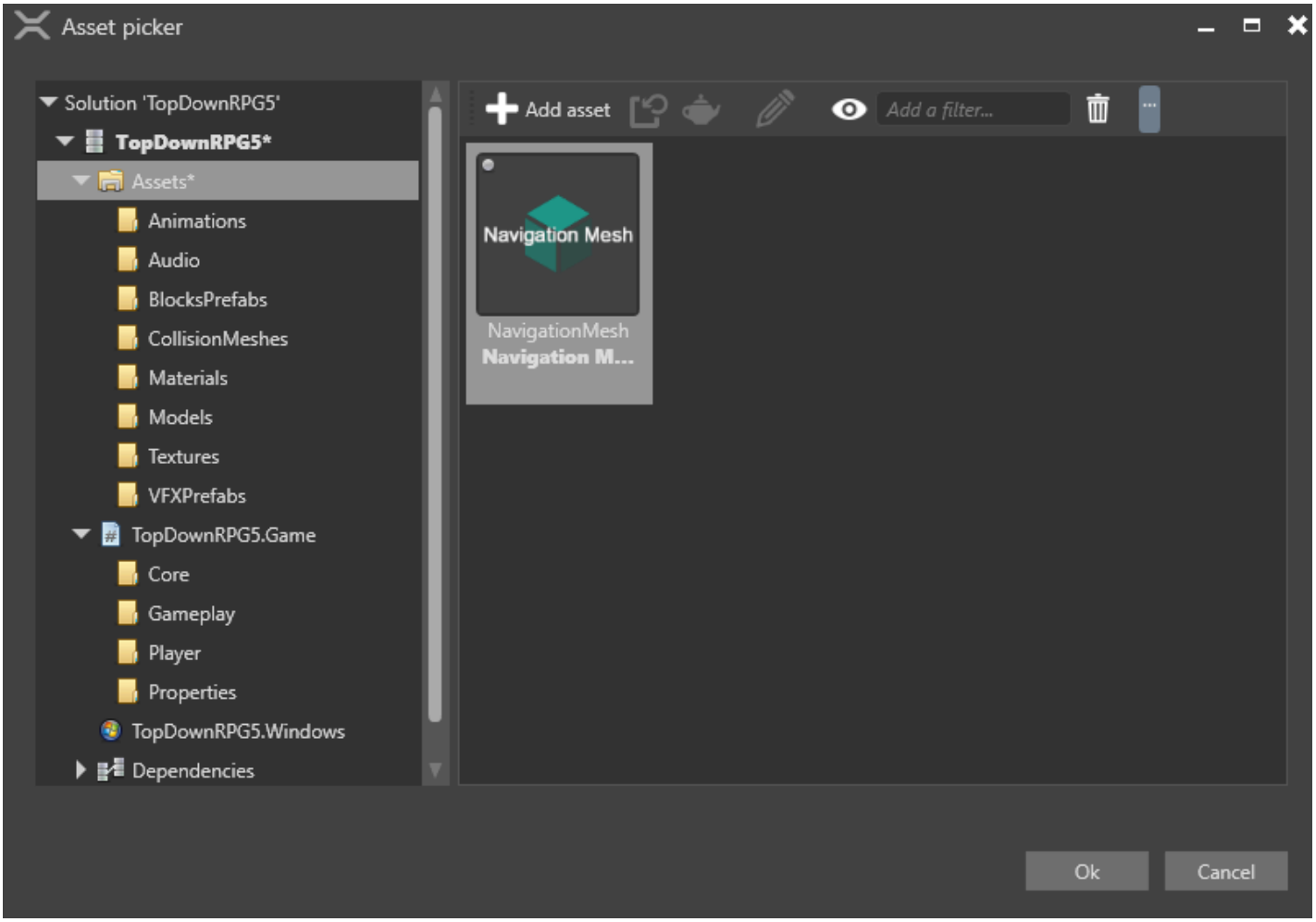
   Game Studio adds a navigation component to the entity.

3. Under the **Navigation** component properties, next to **Navigation mesh**, click (**Select an asset**):



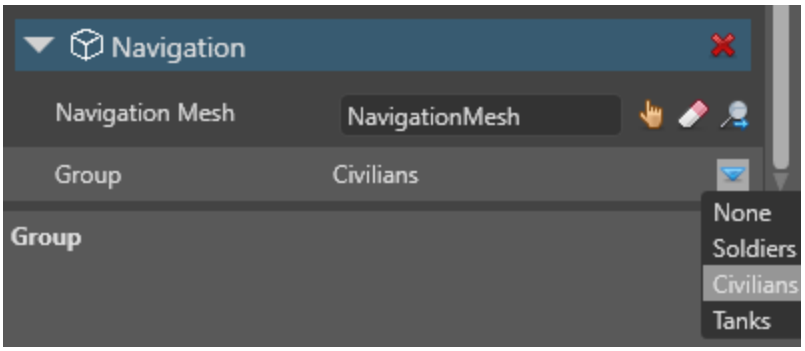
The **Select an asset** window opens.

4. Select the navigation mesh you want the entity to use and click **OK**.



Alternatively, if you want this entity to navigate dynamically by generating its own navigation mesh, leave the **Navigation mesh** field empty. For more information, see Dynamic navigation.

5. Under **Group**, select the navigation group the entity should belong to. The entity uses the navigation properties you set in this group.



Use navigation components in scripts

For example:

```
void Move(Vector3 from, Vector3 to)
{
        var navigationComponent = Entity.Get<NavigationComponent>();
        List<Vector3> path = new List<Vector3>();
        if(navigationComponent.TryFindPath(from, to, path))
        {
                // Follow the points in path
        }
        else
        {
                // A path couldn't be found using this navigation mesh
        }
}
```

For more information, see the NavigationComponent API documentation.

See also

- Navigation groups
- Navigation meshes
- Navigation bounding boxes
- Dynamic navigation

Dynamic navigation

Beginner Level designer Programmer

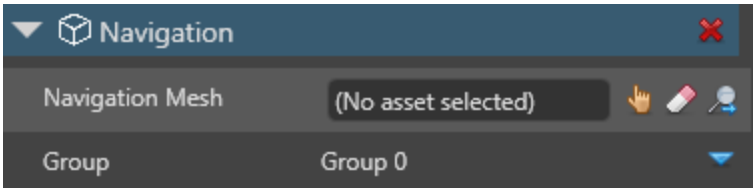
If you enable **dynamic navigation**, entities with navigation components don't need a navigation mesh asset. Instead, the entities generate navigation meshes dynamically.

> **NOTE**
>
> Make sure that the scenes you want the entities to navigate dynamically have navigation bounding boxes.

Enable dynamic navigation

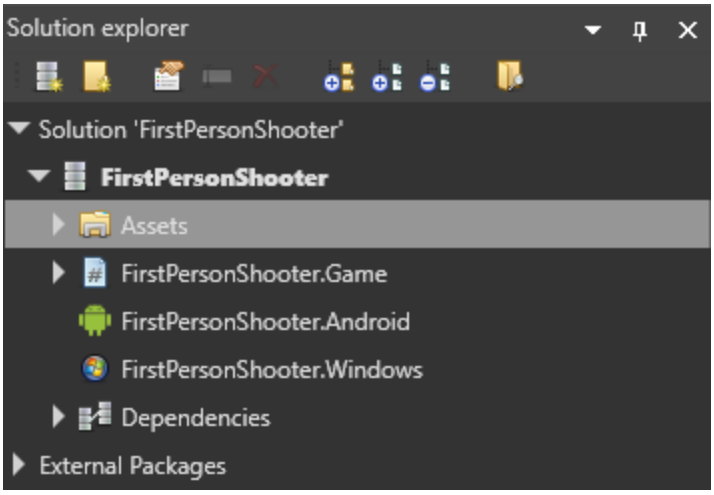
You can enable and disable dynamic navigation in the global game settings asset.

1. On the entities you want to navigate dynamically, under the navigation component properties, next to **Navigation mesh**, make sure no navigation mesh is selected.

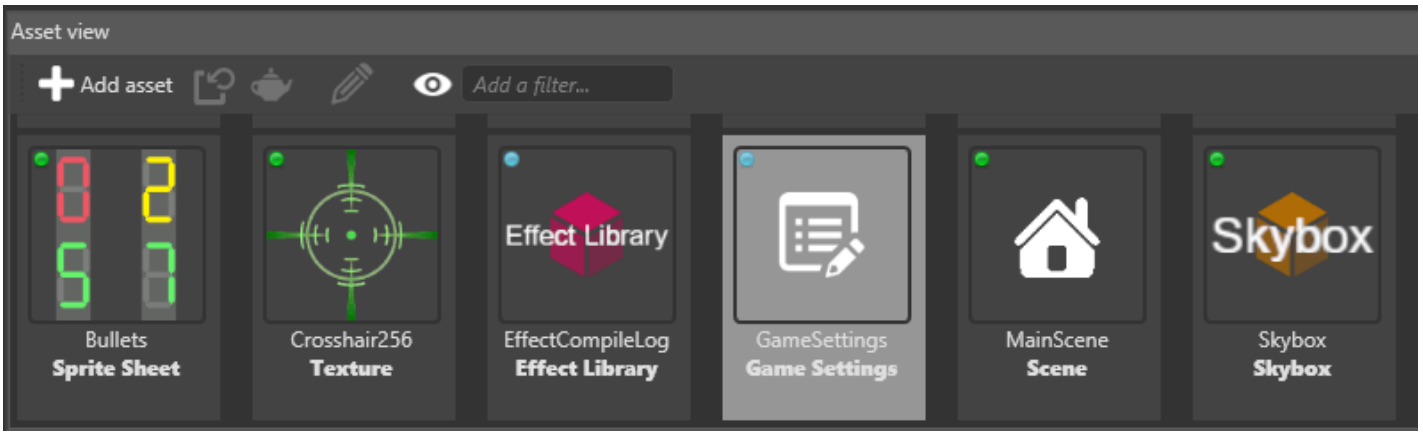
   

   For more information about the navigation component, see Navigation components.

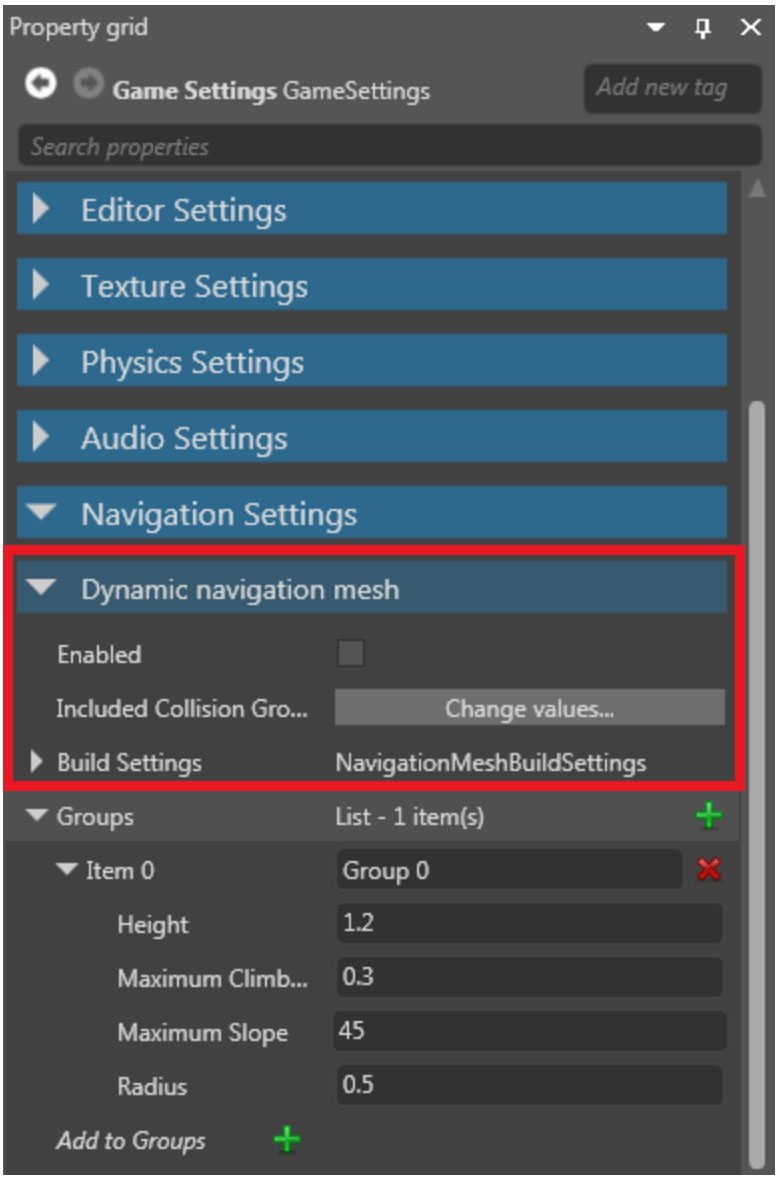
2. In the **Solution Explorer** (the bottom-left pane by default), select the **Assets folder**.

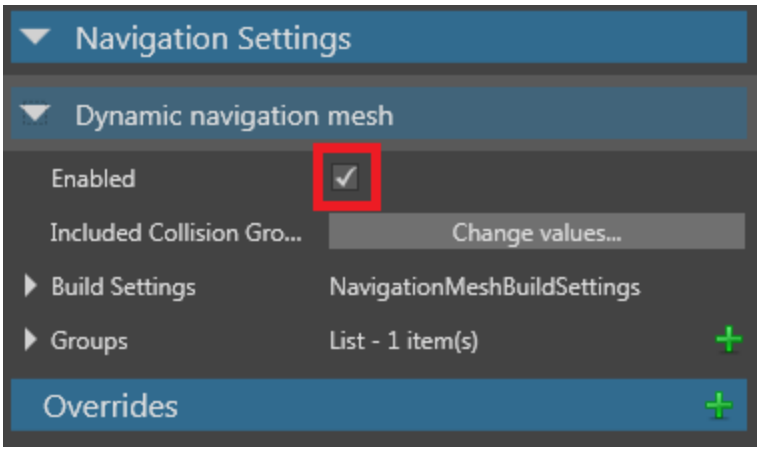
3. In the **Asset View** (the bottom pane by default), select the **Game Settings** asset.



4. In the **Property Grid** (the right-hand pane by default), under **Navigation Settings**, expand **Dynamic navigation mesh**.



5. Select the **Enable dynamic navigation** checkbox.



Dynamic navigation mesh properties

Property	Description
Enabled	Enable dynamic navigation on navigation components that have no assigned navigation mesh
Included collision groups	The collision groups dynamically-generated navigation meshes use. By default, meshes use all collision groups
Build settings	Advanced settings for dynamically-generated navigation meshes

Enable and disable dynamic navigation from a script

Example code:

```
// Find and enable the dynamic navigation mesh system
dynamicNavigationMeshSystem = Game.GameSystems.OfType<DynamicNavigationMeshSystem>
().FirstOrDefault();
dynamicNavigationMeshSystem.Enabled = true;

// This stops the dynamic navigation mesh system from automatically rebuilding in the
folowing cases:
//  - loading/Unloading scenes
//  - adding/removing static collider components
//  - adding/removing navigation mesh bounding boxes
dynamicNavigationMeshSystem.AutomaticRebuild = false;

// ...

if (/* any condition that should cause the navigation mesh to update (eg open/close door)
*/)
{
```

```
        // Start an asynchronous rebuild of the navigation mesh
        var rebuildTask = dynamicNavigationMeshSystem.Rebuild();
        rebuildTask.ContinueWith((x) =>
        {
                if (x.Result.Success)
                {
                        // The navigation mesh is successfully rebuilt
                }
        });
}
```

See also

- Navigation groups
- Navigation meshes
- Navigation bounding boxes
- Navigation components

Particles

Particles are shapes that, used in large numbers, create pseudo-3D effects. You can use particles to create effects such as liquid, fire, explosions, smoke, lightning, motion trails, magic effects, and so on.

In this section

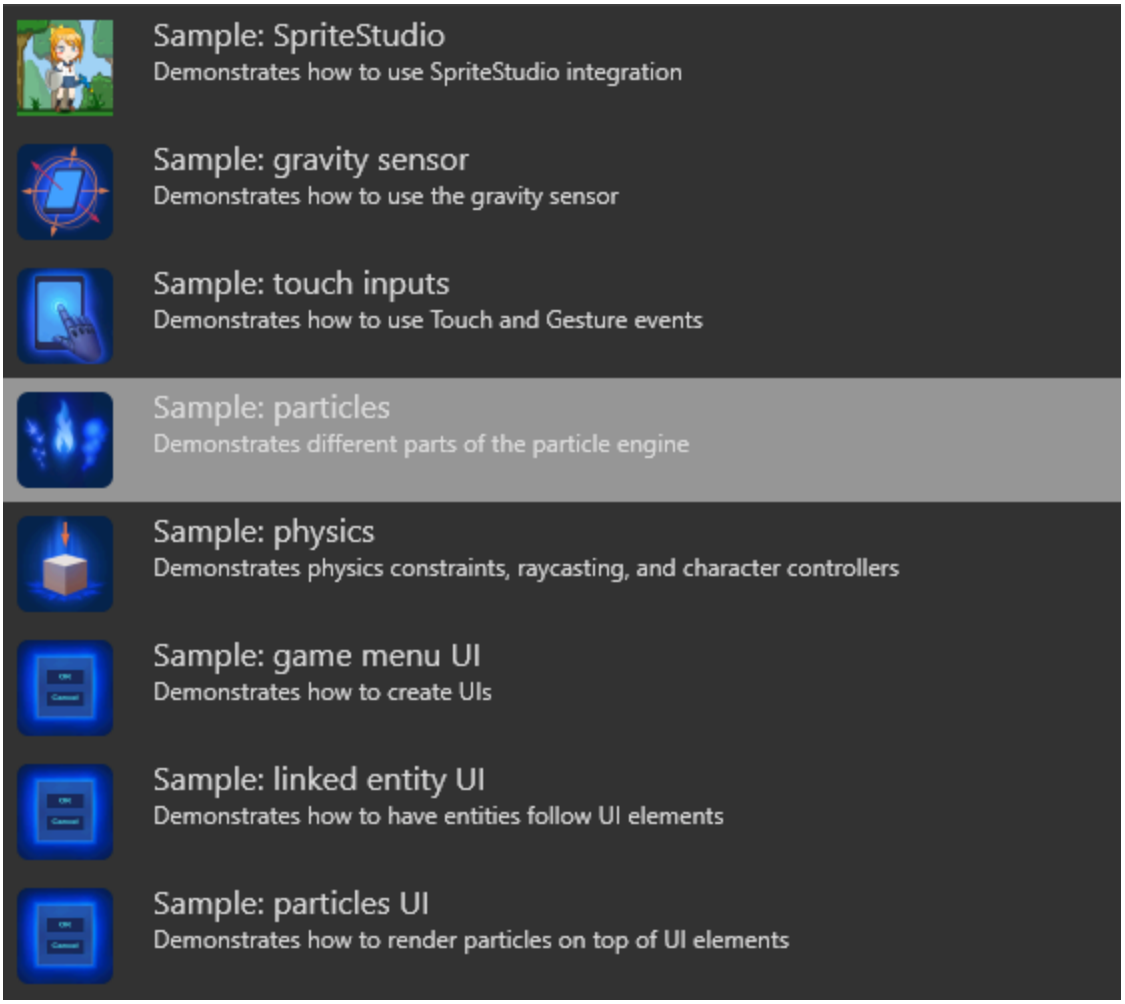
- Create particles
  - Emitters
    - Shapes
      - Ribbons and trails
    - Materials
    - Spawners
    - Initializers
    - Updaters

Tutorials

- Create a trail — Create a motion trail for a sword slash animation
- Lasers and lightning — Create laser and lightning effects using particles and custom materials
- Inheritance — Create particles that inherit attributes from other particles
- Particle materials
- Custom particles

Sample project

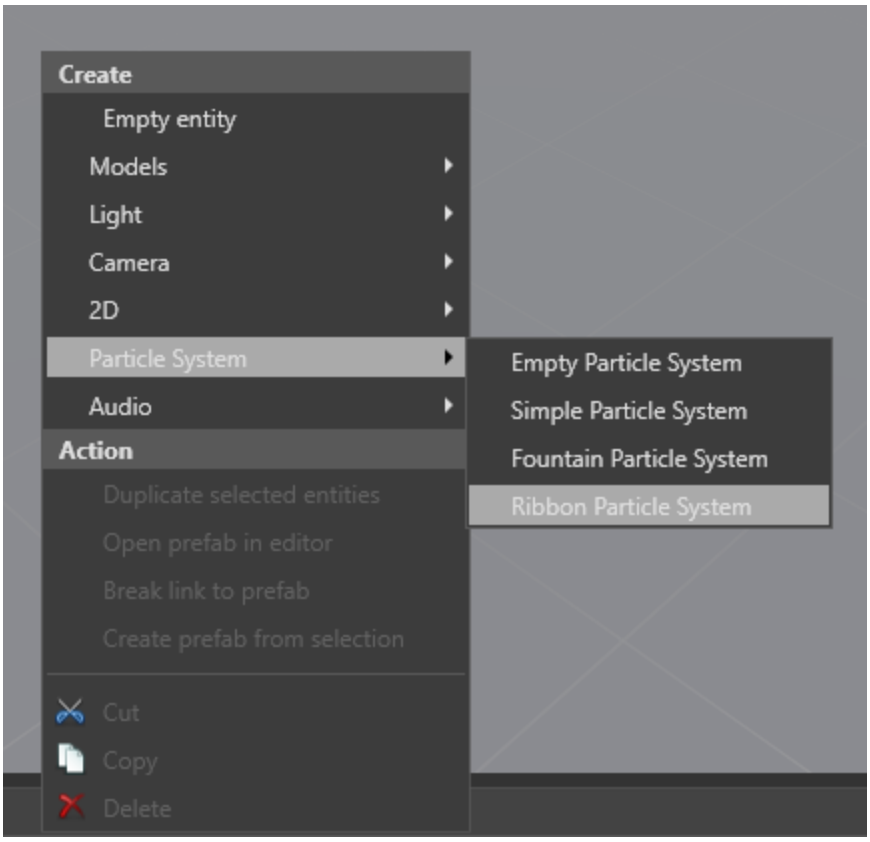
To see some examples of particles implemented in a project, create a new **Sample: Particles** project and check out the different scenes.



Create particles

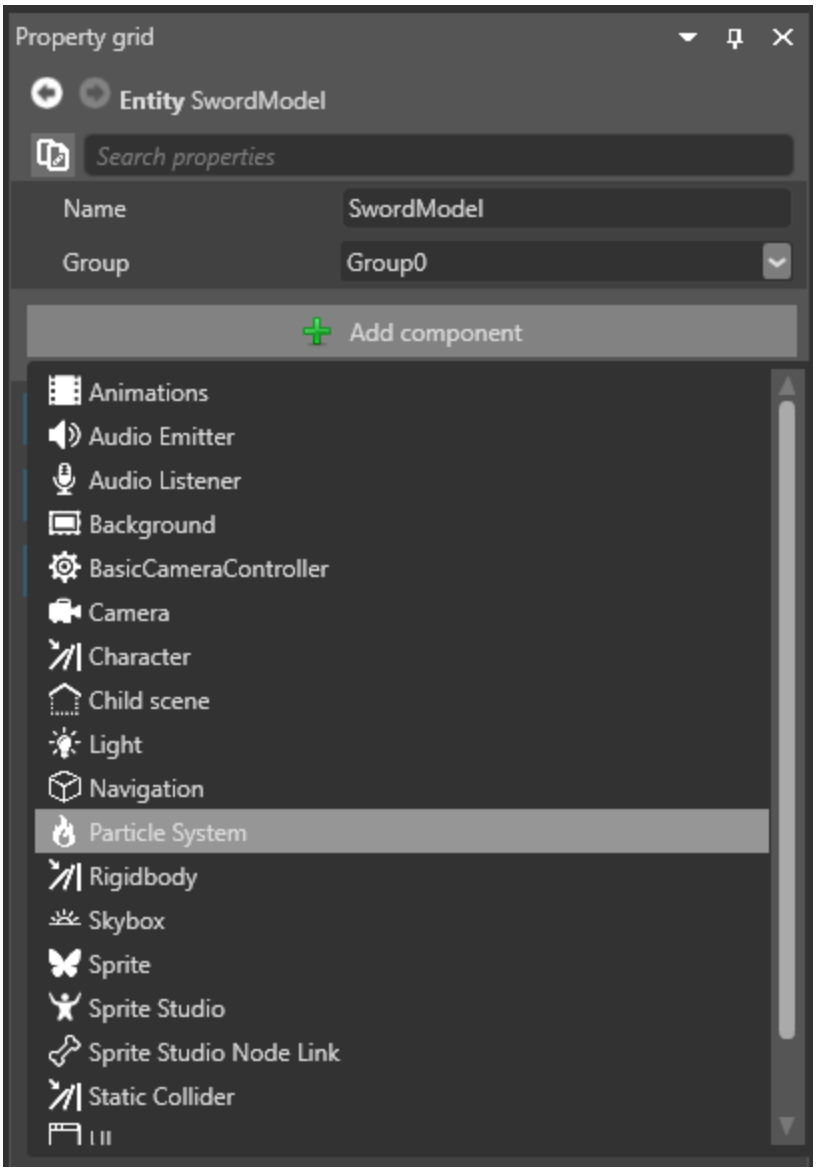
Beginner Artist Programmer

To create a particle system, right-click the scene or Entity Tree, select **Particle System**, and choose a preset (**Empty**, **Simple**, **Fountain**, or **Ribbon**).



Game Studio creates an entity with a **Transform** component and a **Particle System** component with your chosen preset. Particle entities are represented with a flame icon.

Alternatively, you can add a particle component to an existing entity. With the entity selected, in the **Property Grid**, click **Add component** and select **Particle System**.



Game Studio adds an empty particle system to the entity.

Transform component

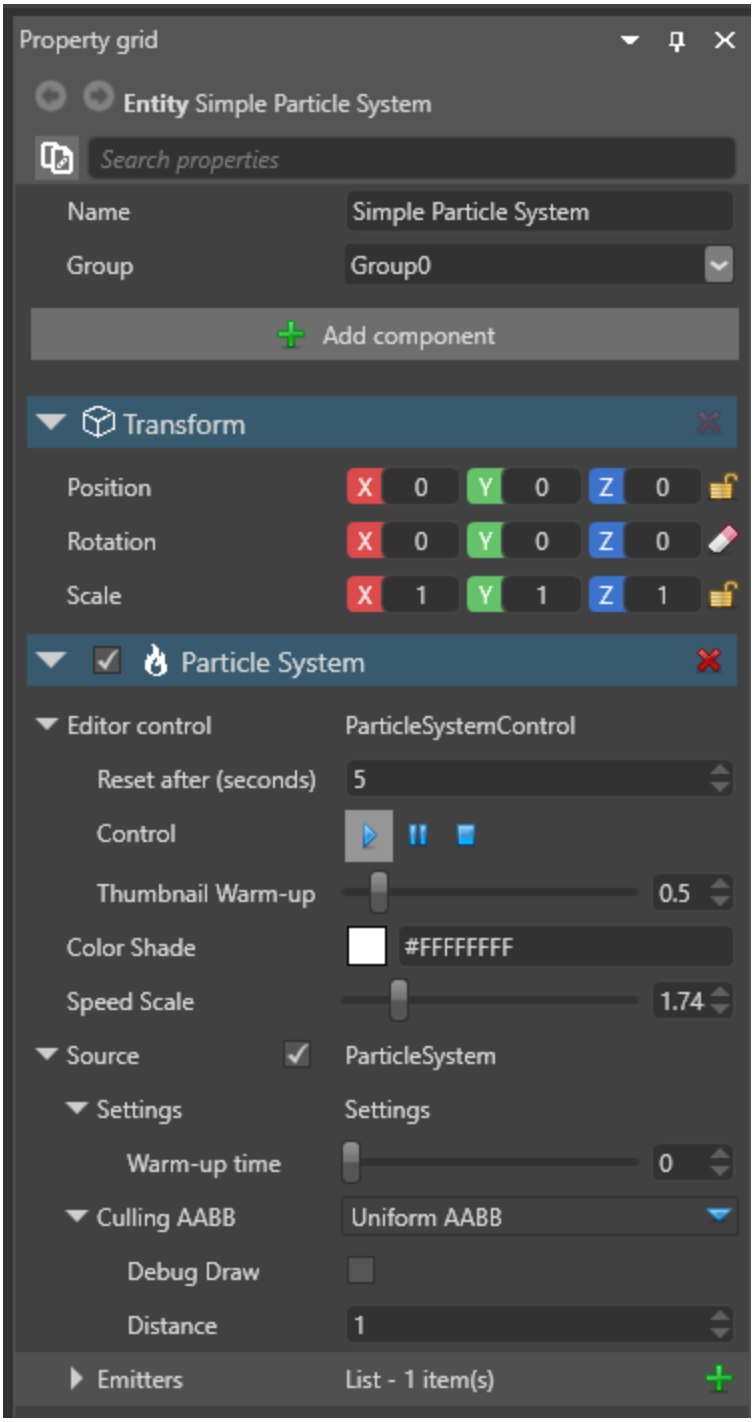
All entities have a transform component. Some particle elements ignore some elements of the transform component, such as rotation or scaling. For example, the gravity force shouldn't depend on the rotation of the particle system, and always ignores rotation; however, fountain particle systems inherit the location for the purposes of initial particle velocity.

Only uniform scaling is supported. If you have a non-uniform scale on the transform component, only the X axis is used.

If you want two particle systems to share a transform component, create two particle system entities and make one a child of the other.

Particle component properties

With a particle system entity selected, you can edit its properties in the **Property Grid**, just like any other entity.



Property	Description
Editor control	This changes how Game Studio displays particles while you work on the scene. You can play, pause, and stop the particle system. You can also reset the particle effect at set intervals, which is useful for previewing one-shot effects. The editor controls don't affect how particles are displayed at runtime.

Property	Description
Warm-up time	If you set the warm-up time to a value greater than 0, the particle appears as if it's already active when it appears. This value is in seconds. For example, if you set the warm-up time to 1, the particle effect appears as if it has already been active for 1 second when it appears. This is useful, for example, if you set a fire effect warm-up time to 0, the fire appears to ignite as soon as it's rendered. If you want the fire to appear as if it's already ignited when it's rendered, increase the warm-up time.
Speed scale	Controls the speed of the particle effect.
Culling AABB	This creates an axis-aligned bounding box (AABB) around the particle effect. If the bounding box isn't in the camera view, Stride doesn't render the particle effect. This is useful for culling and optimization. **Rotated AABB** sets box shape in XYZ co-ordinates. **Uniform AABB** creates a cube of the scale you specify (in world units). To view the AABB in the Scene Editor, select **Debug Draw**.
Emitters	The emitters the particle system contains. The emitters are updated and drawn in the order they appear in the list, and can be re-ordered. For more information, see Emitters.

See also

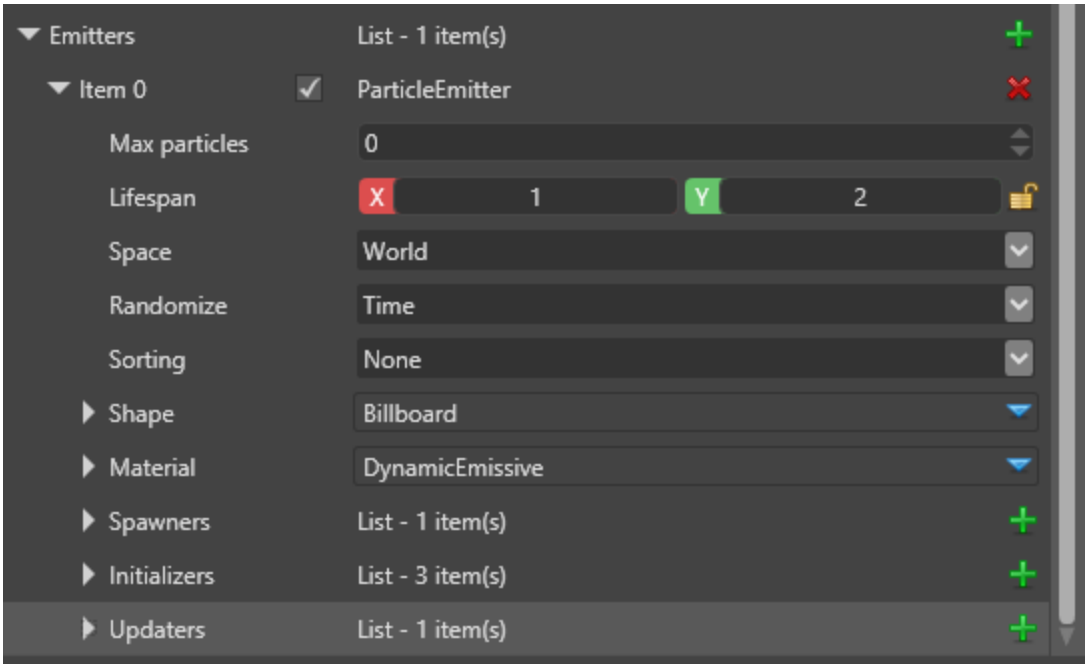
- Emitters
- Shapes
- Materials
- Spawners
- Initializers
- Updaters

Emitters

Beginner Artist Programmer

Particle emitters manage how many particles are in an effect, how they appear, move, and disappear, and how they are drawn. For example, a fire effect might be composed of three separate particle effects: flames, embers, and smoke. Each of these effects is managed by a separate particle emitter.

Emitters contain further controls such as spawners, initializers, and updaters.



Property	Description
Emitter name	A unique identifier for the particle emitter
Max particles	The maximum number of active particles the emitter can manage at a given time, based on the particles' spawn rate and lifespan. If you leave this at 0, Stride uses its own estimate.
Lifespan	New particles have a lifespan between these two values
Space	Particles in **world** space remain in the world space when the emitter moves away from them. Particles in **local** space always exist in the emitter's local coordinate system; if the emitter moves, rotates, or scales, the particles move with it.
Randomize	Particles use pseudo-random values for everything which requires randomness. If you set this to **Time**, different emitters generate different random numbers. If you set it to

Property	Description
	Fixed, different instances of the same effect behave identically. **Position** acts as Fixed but is different for different positions.
Draw priority	This controls the order in which particles are drawn. Higher numbers have higher priority. For example, if this particle effect has a draw priority of 2, it will be drawn after a particle effect with a draw priority of 1.
Sorting	Choose if the articles should be drawn by **depth** (away from the camera), **age** (particles spawned first are drawn on top), **order**, or in no order **none** (good for additive particles, which need no sorting).
Shape	Specifies the shape used to draw the particles
Material	Specifies the material used to render the particles
Spawners	Spawners control how quickly new particles are emitted. To emit particles, emitters must have at least one spawner.
Initializers	Initializers set the initial values of new particles
Updaters	Updaters update living particles every frame, changing their attributes. Updaters execute in the order in which they appear on the list.

See also

- Create particles
- Shapes
- Materials
- Spawners
- Initializers
- Updaters

Particle shapes

Beginner Artist Programmer

Because particles are essentially only points in space, they have no defined shape. Instead, Stride draws shapes **between** the points.

The major difference between particle shapes is whether they always face the camera, or if they can rotate freely in 3D space.

Currently, emitters can only emit one type of shape at a time.

Billboards

Billboards always face the camera. They appear **fixed in 3D space**, so they don't change with the camera position.

Because they always face the camera, billboards support angular rotation only. This means they only rotate clockwise or counter-clockwise.

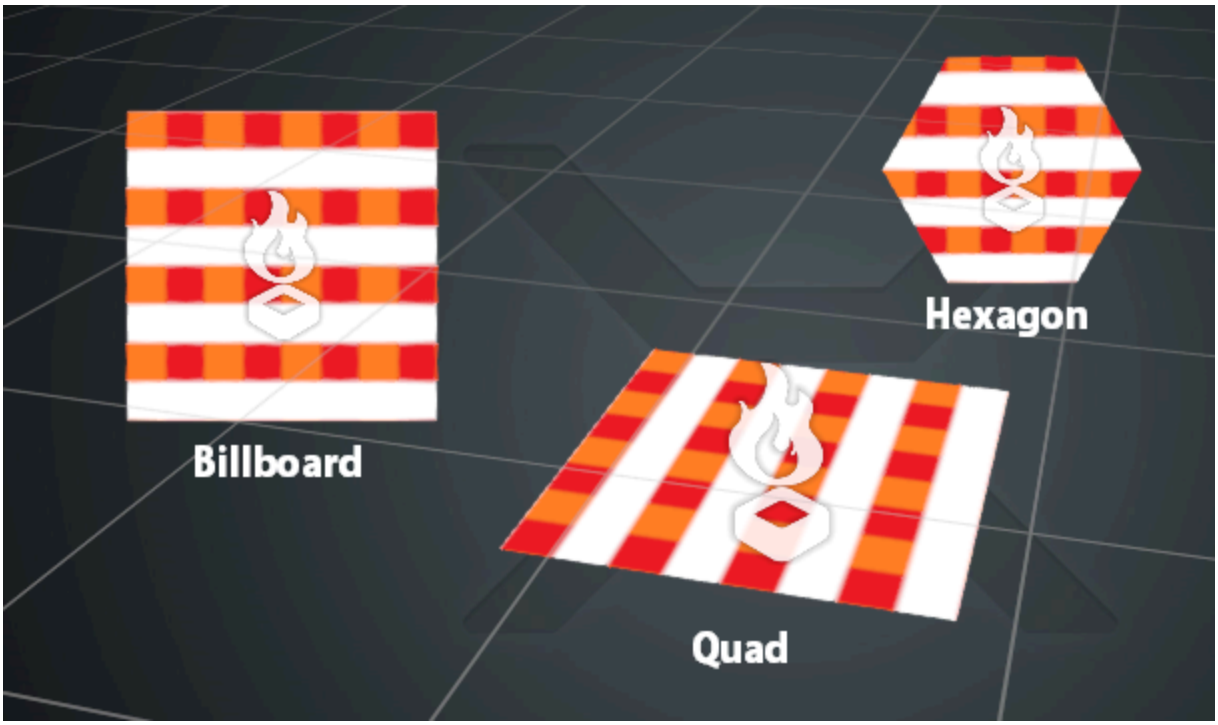
Hexagon

Hexagons are identical to billboards, but are hexagonal in shape. Like billboards, they always face the camera and support angular rotation only.

Quads

Quads are identical to billboards, but don't rotate to face the camera, and so support 3D orientation and rotation.

Stride draws billboard particles to the **Size** value in the particle effect properties. If you don't specify a size, Stride expands the quads to 1m x 1m.



Direction-aligned sprite

This sprite is billboard-aligned and stretched in the direction of the particle. You can set an initial direction for the particles with an initializer, or add an updater which writes particle speed as direction.

Ribbons and trails

See Ribbons and trails.

See also

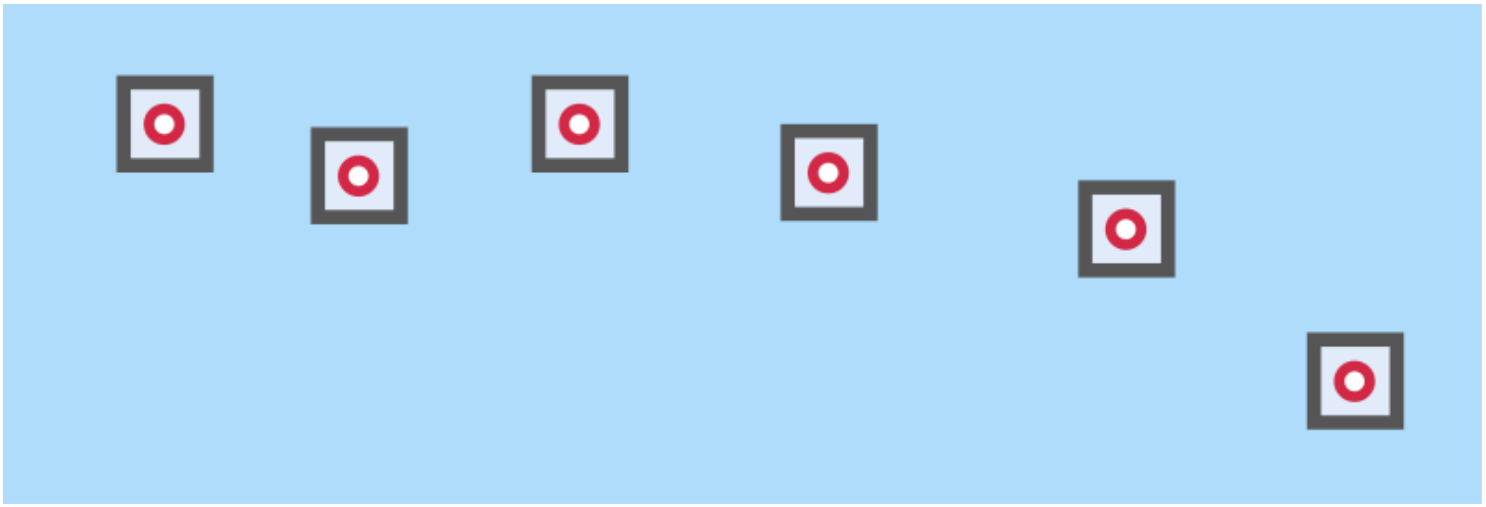
- Create particles
- Emitters
- Materials
- Spawners
- Initializers
- Updaters

Ribbons and trails

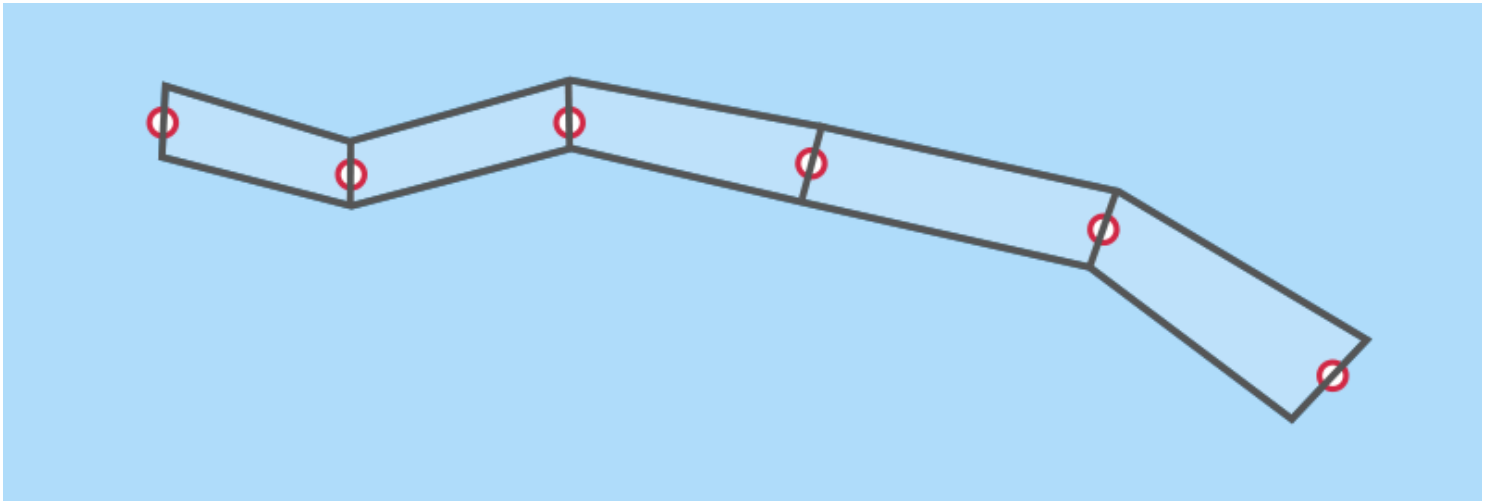
Intermediate Artist Programmer

To create **ribbons** and **trails**, Stride builds the mesh data as a strip connecting the particles, rather than individual quads. Ribbons and trails are often used to create visual effects such as sword slashes.

In the diagram below, several particles (represented as red dots) are rendered as individual quads (blue squares):



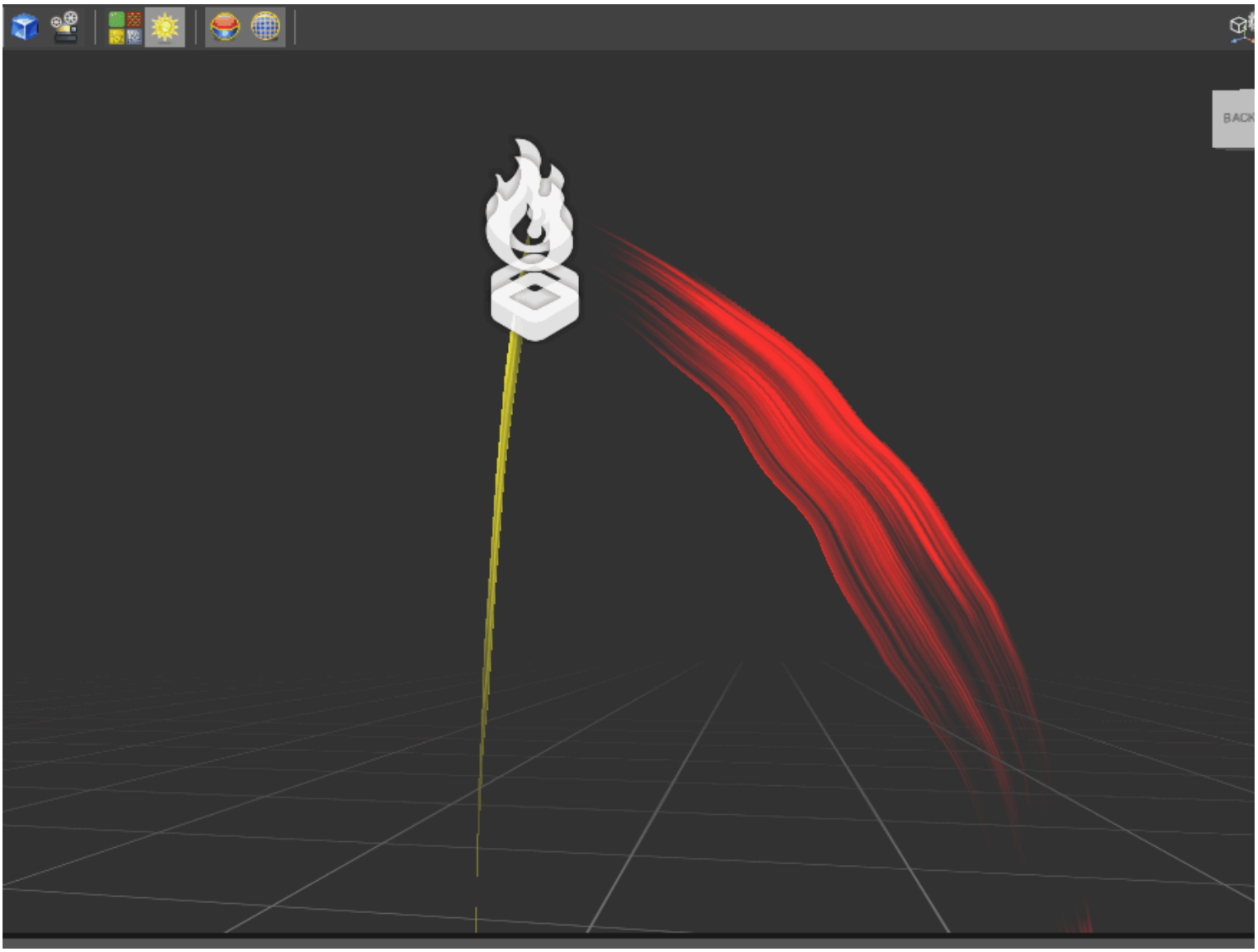
In the diagram below, a strip is created by connecting the particles and rendering quads between the adjacent particles:



Ribbons vs trails

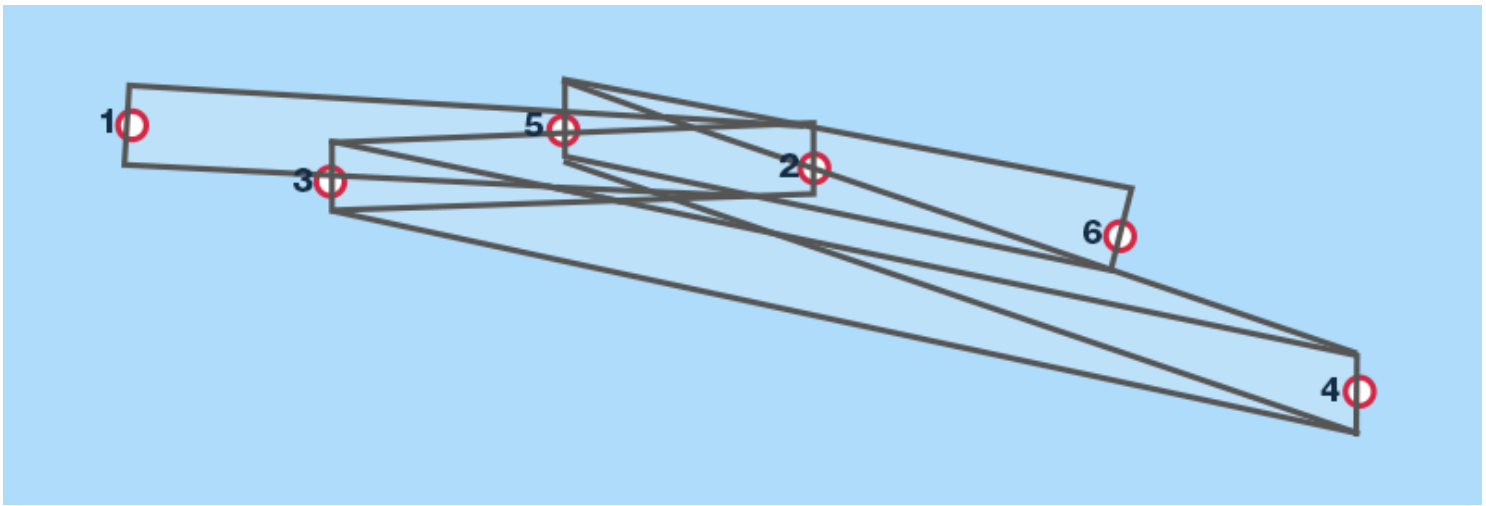
Both ribbons and trails generate a flat surface which follows an axis connecting adjacent particles in a line. This line defines one of the axes of the surface. The difference is that ribbons always face the camera, and trails don't.

The gif below shows the different behavior of ribbons (red) and trails (yellow) when viewed from different camera angles. Note how the ribbon doesn't change as the camera moves; it's fixed in space.

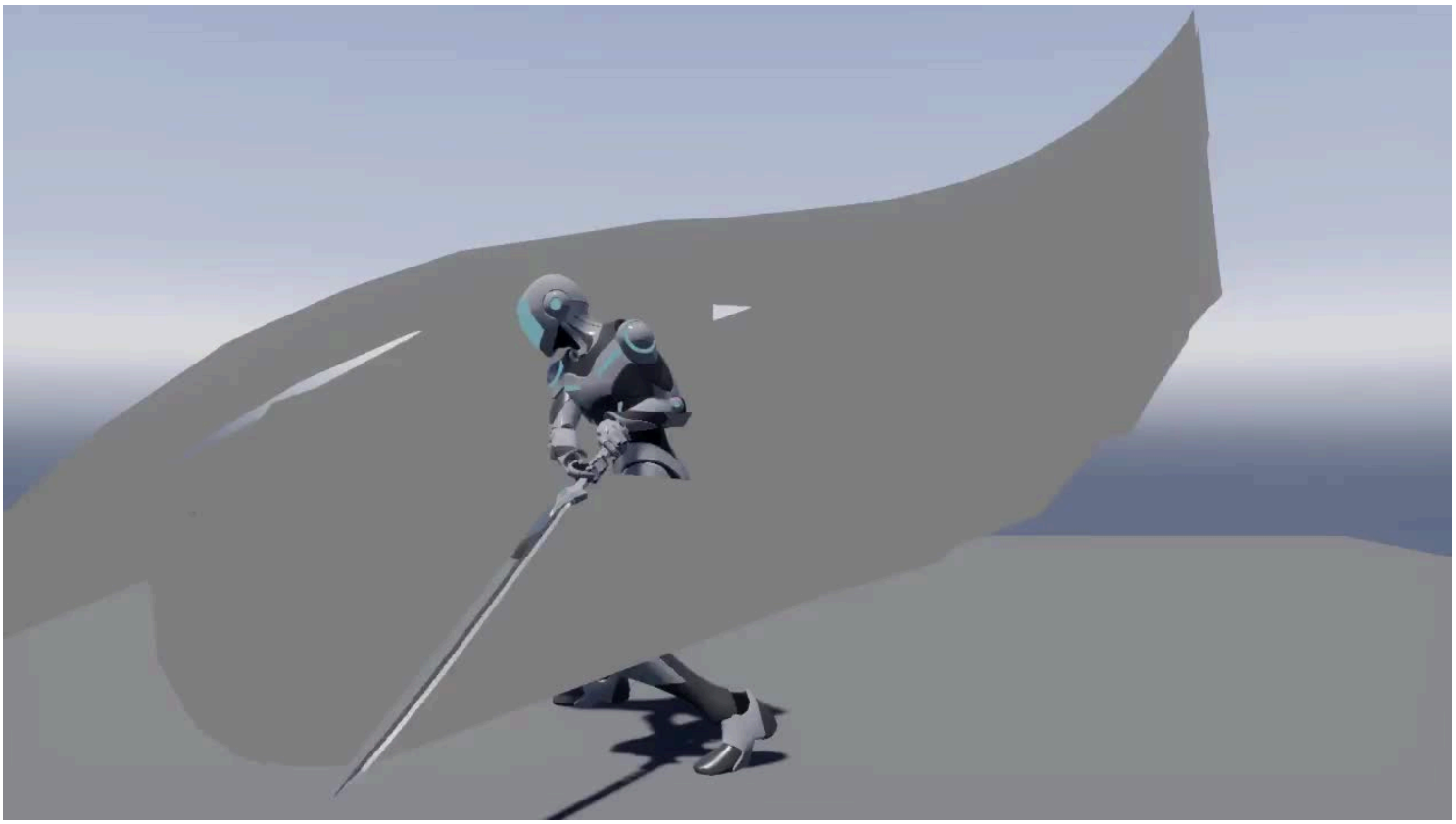


Sort particles

To create ribbons and trails, you usually need to sort the particles into an order. If you don't sort the particles, they connect erratically, as in this diagram:



Here's an example of how unsorted particles look at runtime:

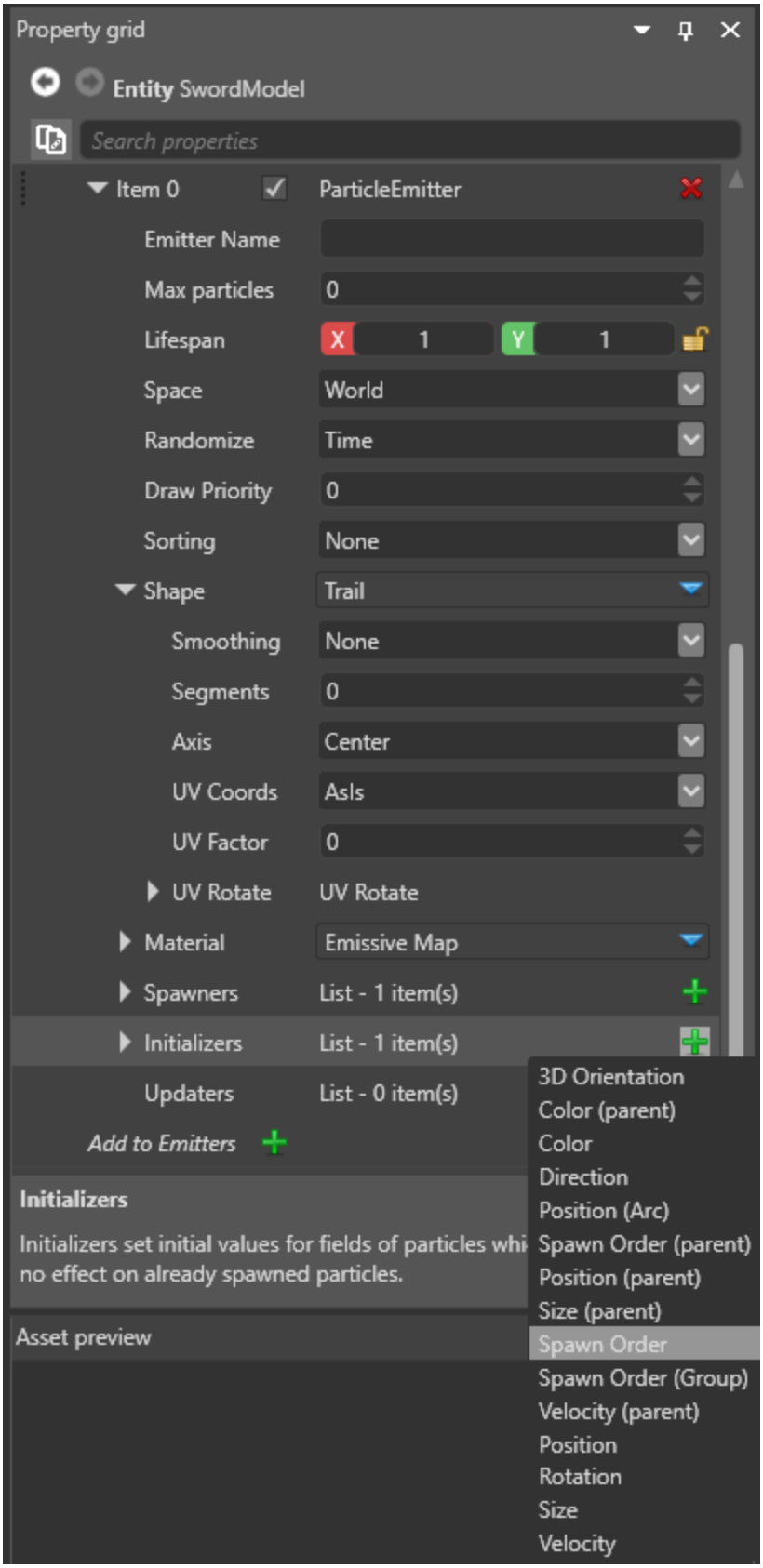


Rather than the particles connecting in order, the strip erratically jumps between particles. (This is the same problem alpha-blended quads have when they're not properly sorted.)

To sort the particles, under **Particle System > Source > Emitters**, change the **Sorting** property.

If your particles have the same **lifespan** property, and are emitted no more than once per frame (usually the case at 30 particles per second or fewer), you can sort them by age.

However, if you spawn several particles per second or your particles vary in lifespan, sorting by age doesn't provide a consistent order, as the sorting parameter changes between frames. In this case, you should sort the particles by order. To do this, you need to add a **spawn order initializer**. To do this, in the entity properties, under **Particle System > Source > Emitters**, next to **Initializers**, click (**Add**) and select **Spawn Order**.

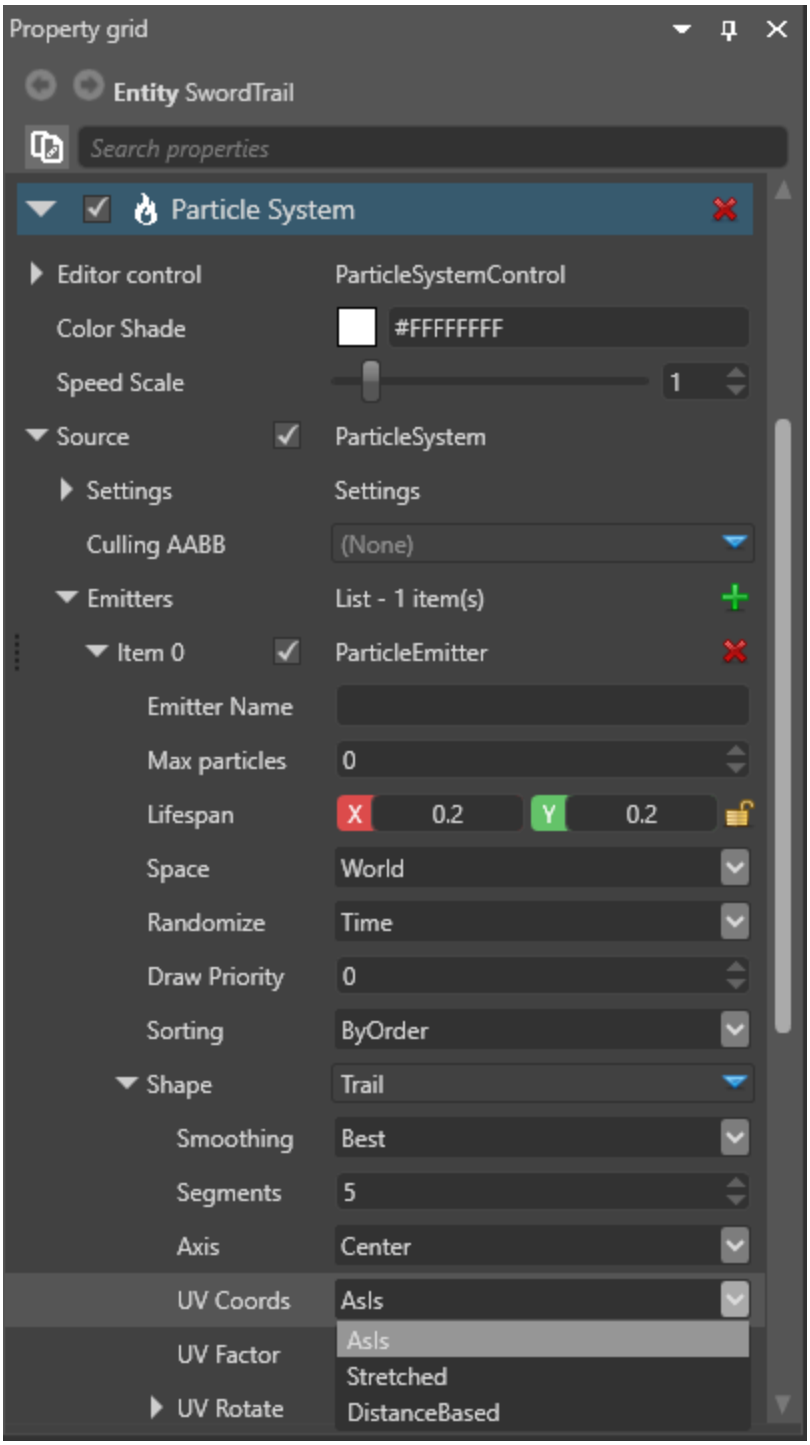


This adds a spawn order initializer to the emitter. It doesn't have any properties, but it gives the particles a SpawnID we can sort them by.

> **NOTE**
>
> Sorting by depth might work in niche cases, but this doesn't preserve the order between different frames. We don't recommend it for most situations.

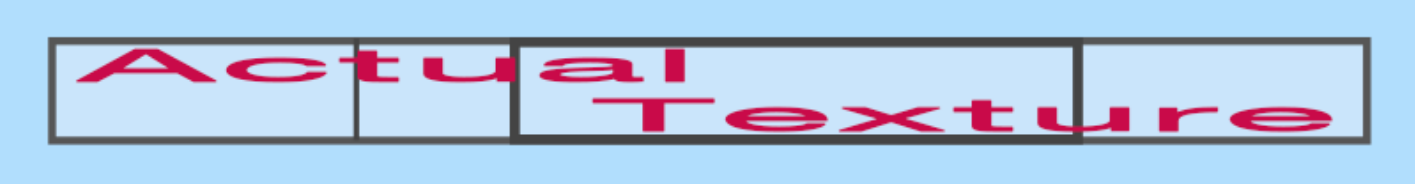
Texture coordinates

Unlike billboards, which are individual quads, ribbons and trails have a single surface across all particles. To define how textures are mapped across the surface, under **Particle System > Source > Emitters > Shape**, change the **UV Coords** property.



- **AsIs**: The texture is mapped per segment, copying the same quad stretched between every two particles. This is sometimes useful with flipbook animations (in the Material settings).

- **Stretched**: The texture is stretched between the first and last particle of the trail or ribbon. The **UV Factor** defines how many times the texture appears across the entire trail or ribbon (1 = once).

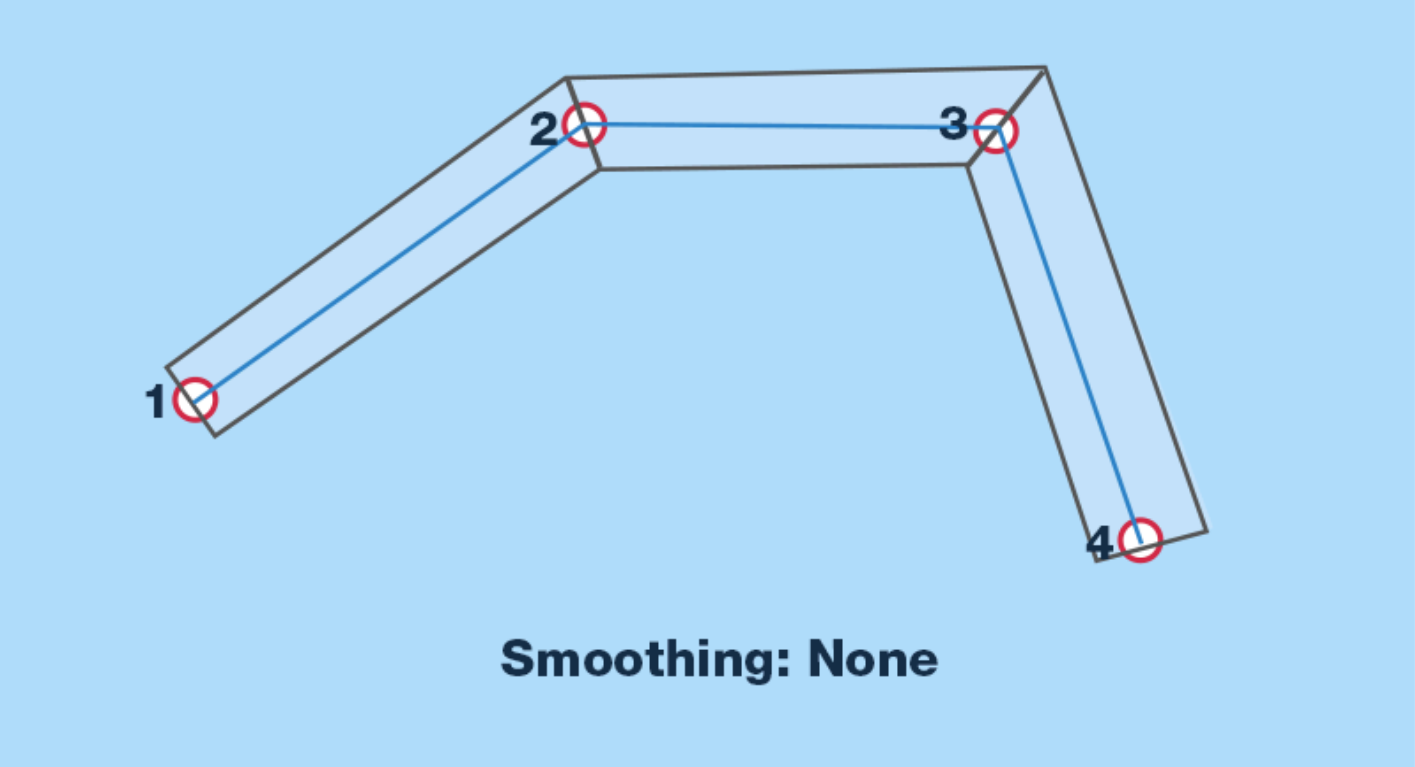


- **DistanceBased**: The texture is repeated based on the actual world length of the ribbon or trail rather than the number of particles. The **UV Factor** defines the distance in world units after which the texture repeats

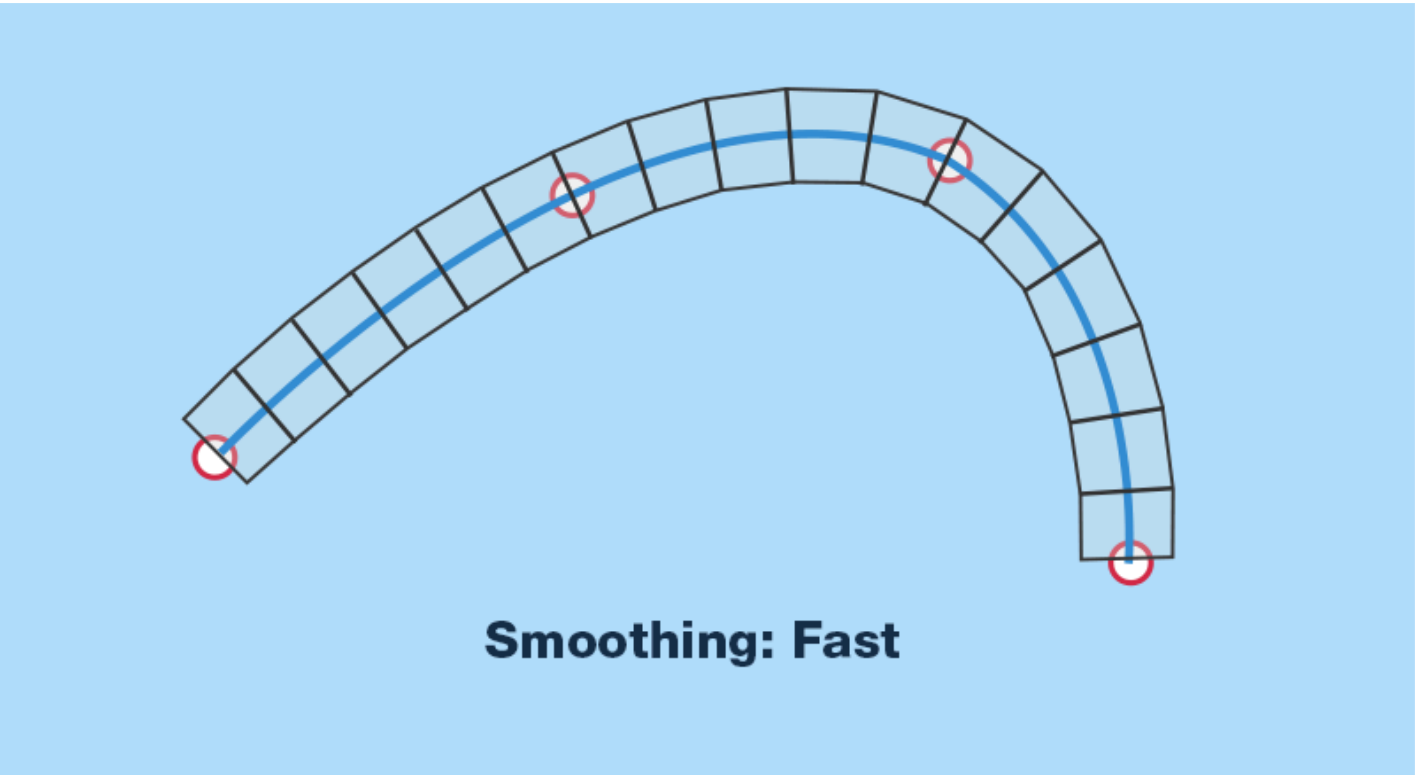
Smooth ribbons and trails

You can add extra segments between adjacent particles to smooth the lines between particles. To do this, under **Particle System > Source > Emitters > Shape**, change the **Smoothing** property.

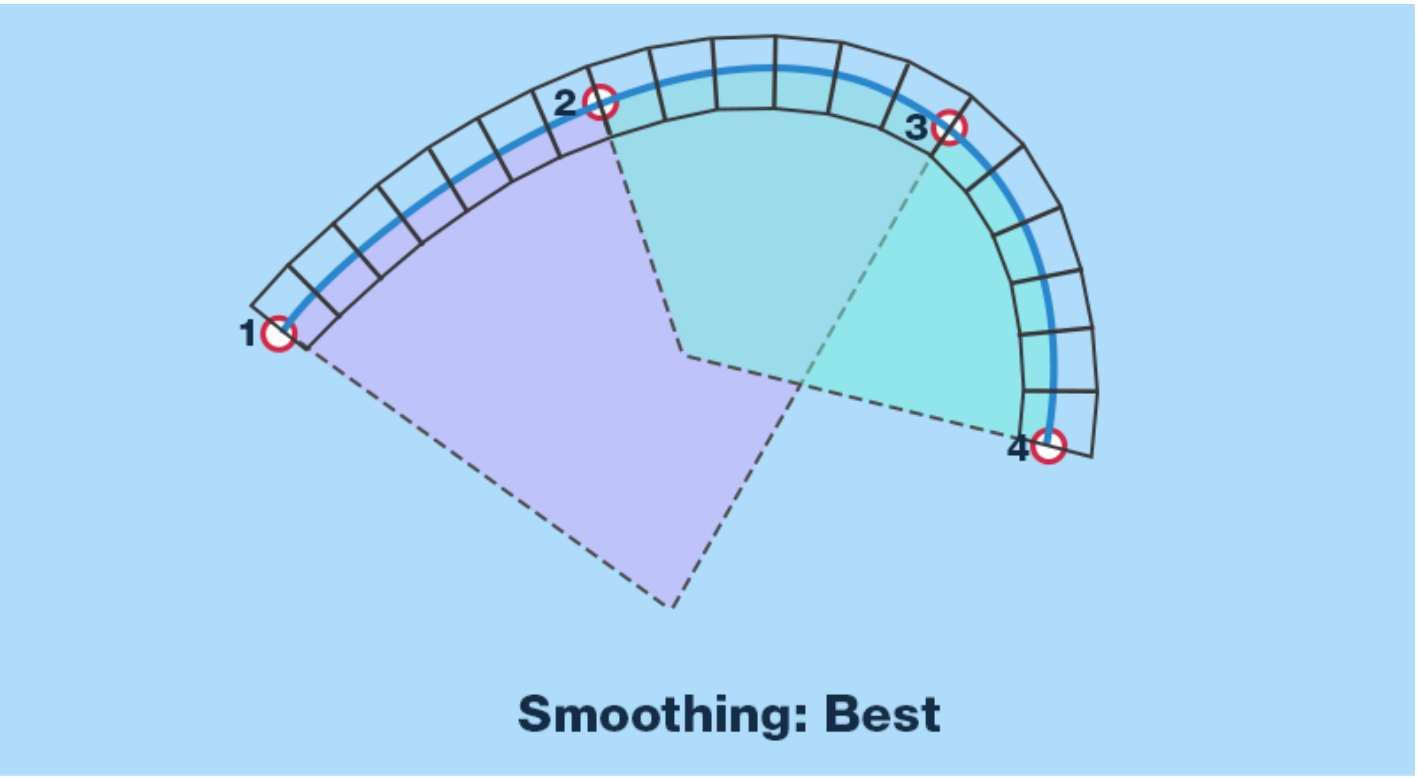
- **None** — No smoothing creates only one segment joining two particles. This creates trails and ribbons with sharp angles.



- **Fast** — This uses Catmull-Rom interpolation (Wikipedia) to add extra segments between particles, creating a smoother effect. You can set the number of segments with the **Segments** property.



- **Best** — This generally creates the smoothest effect, but requires more CPU. It calculates a circumcircle around every three sequential particles along the control axis, then adds extra control points on the circle, keeping the segments in an arc. For the first and the last segment, there is only one arc to be followed, but for mid-sections, two different arcs from two different circles overlap; Stride interpolates the control points from the first arc and the second as the point approaches the second particle. You can set the number of segments between every two particles with the **Segments** property.



This video shows the difference between the three smoothing methods. Note that the rightmost trail (using the **Best** method) is slightly more circular, closer to the actual path of the sword swing.

Sample project

For an example of a project that uses ribbons and trails, try the **Ribbon Particles Sample** included with Stride.

See also

- Shapes
- Tutorial: Create a trail
- Tutorial: Lasers and lightning

Particle materials

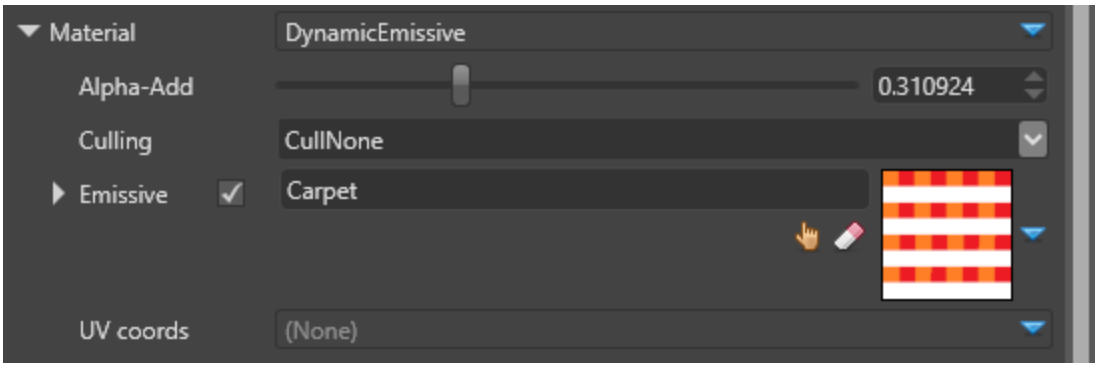
Beginner Artist Programmer

Materials define how the expanded shape should be rendered. They defines color, textures, and other parameters.

Particle materials are simplified versions of materials used for meshes. There is only one type of material currently, the Dynamic Emissive material.

Dynamic emissive

This material uses a translucent emissive color RGBA for the pixel shading. In HDR rendering mode, the values are used as intensity, and can be higher than 1.

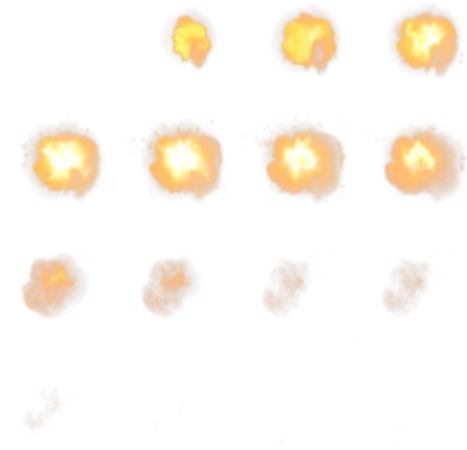


Property	Description
Alpha-Add	Translucent rendering supports alpha-blending, additive blending or anything in-between. With this parameter you can control how much alpha-blended (0) or additive (1) the particles should be.
Culling	There are options for no culling, front face culling and back face culling. Camera-facing particles always have their front face towards the camera.
Emissive	The emissive RGBA color for the particle. See Material maps for a full description.
UV coords	For particles which use texture sampling uv coordinates animation can be specified. The two currently existing types are specified below.

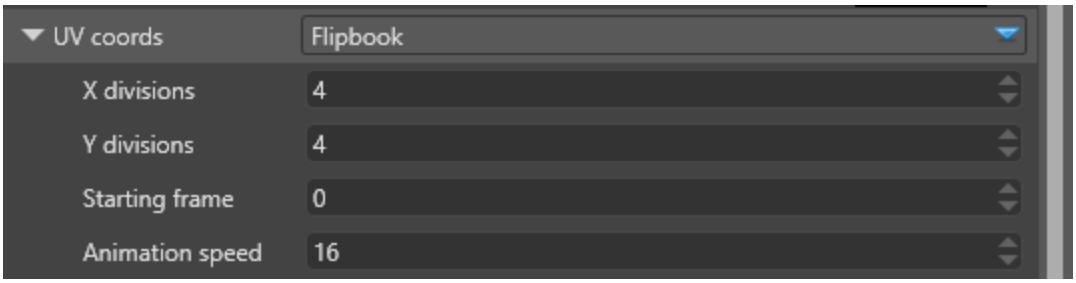
UV Coords — Flipbook

The flipbook animation considers a texture a sequence of frames and displays it one frame at a time, like a flipbook.

This image is an example of a 4x4 flipbook animation texture of an explosion:



The flipbook animation has the following properties:



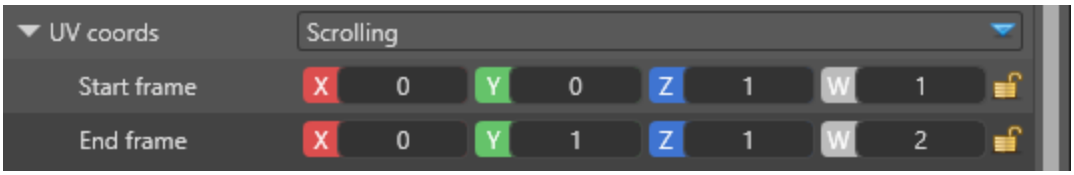
Property	Description
X divisions	The number of columns to split the texture into
Y divisions	The number of rows to split the texture into
Starting frame	The frame to start the animation at. The top-left frame is 0 and increases by 1 from left to right before moving down.
Animation speed	The total number of frames to show over the particle lifetime. If Speed = X x Y, then the animation shows **all** frames over the particle lifetime. The speed is relative; particles with longer lifespans have slower animation.

UV Coords — Scrolling

The scrolling animation defines a starting rectangle for the billboard or quad, which moves across the texture to its end position. This creates a scrolling or a scaling effect of the texture across the quad's surface.

The texture coordinates can go below 0 or above 1. How the texture is sampled depends on the addressing mode defined in the material maps. For more information, see the MSDN documentation.

The scrolling animation has the following properties:



Property	Description
Start frame	The initial rectangle for texture sampling when the particle first spawns
End frame	The last rectangle for texture sampling when the particle disappears

See also

- Create particles
- Emitters
- Shapes
- Spawners
- Initializers
- Updaters

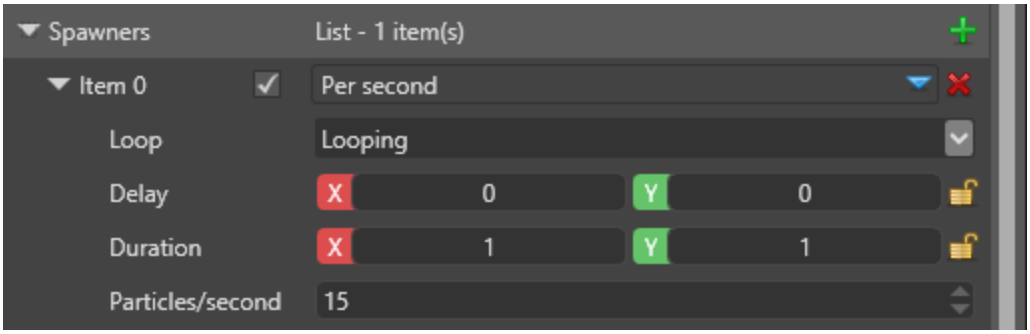
Particle spawners

Beginner Artist Programmer

Particle spawners control when, how many, and how quickly particles are emitted. Emitters need at least one spawner, but can contain multiple spawners with different settings.

Per second

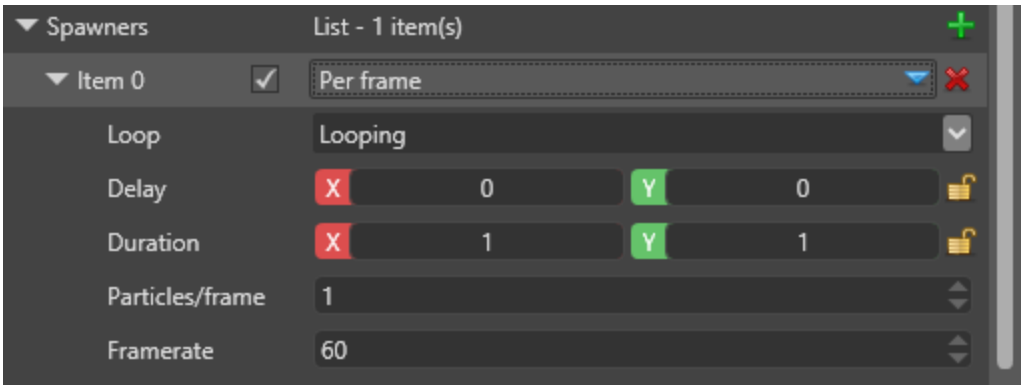
Emits a fixed number of particles per second. It balances and interpolates them and is stable even if the framerate changes or drops. For example, at a rate of 20 particles per second, the spawner spawns one particle every three frames for 60fps games and two particles for every three frames (skipping every third frame) for 30fps games.



Property	Description
Loop	To have the spawner loop when it reaches the end of its duration, select **Looping** (default). To have the spawner loop with no wait between each loop, select **Looping, no delay**. To have the spawner spawn once and then stop, select **One shot**.
Delay	How long (in seconds) the spawner waits before spawning. This is a random value between the X (minimum) and Y (maximum) values.
Duration	How long (in seconds) the spawner spawns particles for. At the end of the duration, the spawner either starts again or stops, depending on the Loop property.
Particles	The number of particles the spawned per second. This can be a floating value (eg 36.875).

Per frame

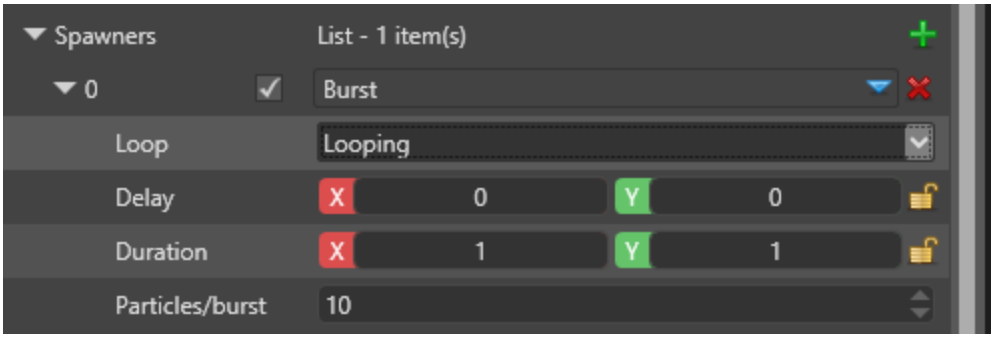
Emits a fixed number of particles per frame, regardless of framerate. This can be useful if you require a fixed number of particles - for example, exactly one every frame, which is useful for trails and ribbons.



Property	Description
Loop	To have the spawner loop when it reaches the end of its duration, select **Looping** (default). To have the spawner loop with no wait between each loop, select **Looping, no delay**. To have the spawner spawn once and then stop, select **One shot**.
Delay	How long (in seconds) the spawner waits before spawning. This is a random value between the X (minimum) and Y (maximum) values.
Duration	How long (in seconds) the spawner spawns particles for.
Particles	The number of particles spawned per frame. The value can be a floating value, including values less than 1, in which case a new particle is spawned every few frames.
Framerate	This is for estimation purposes only when the engine calculates the maximum number of particles.

Burst

Emits all particles in one burst.



Property	Description
Loop	To have the spawner loop when it reaches the end of its duration, select **Looping** (default). To have the spawner loop with no wait between each loop, select **Looping, no delay**. To have the spawner spawn once and then stop, select **One shot**.
Delay	How long (in seconds) the spawner waits before spawning. This is a random value between the X (minimum) and Y (maximum) values.
Duration	How long (in seconds) the spawner spawns particles for.
Particles/burst	The number of particles spawned in each burst.

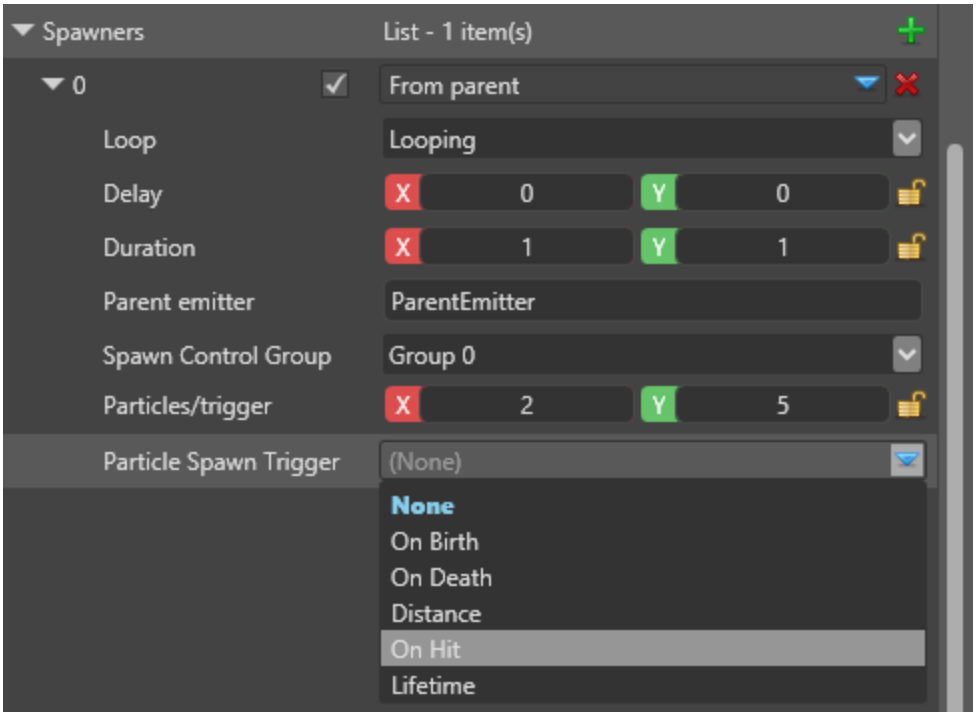
Distance

Emits particles based on the distance traveled by the emitter. If the emitter doesn't move, it spawns no particles.

Property	Description
Loop	To have the spawner loop when it reaches the end of its duration, select **Looping** (default). To have the spawner loop with no wait between each loop, select **Looping, no delay**. To have the spawner spawn once and then stop, select **One shot**.
Delay	How long (in seconds) the spawner waits before spawning. This is a random value between the X (minimum) and Y (maximum) values.
Duration	How long (in seconds) the spawner spawns particles for.
Particles/unit	The number of particles spawned for every distance unit the spawner moves. You can use fractions if you need fewer than one particle per distance unit. The rate adjusts with scaling.

From parent

Emits particles based on other particles (parents). When certain conditions in a parent particle are met, the spawner spawns particles.



Property	Description
Loop	To have the spawner loop when it reaches the end of its duration, select **Looping** (default). To have the spawner loop with no wait between each loop, select **Looping, no delay**. To have the spawner spawn once and then stop, select **One shot**.
Delay	How long (in seconds) the spawner waits before spawning. This is a random value between the X (minimum) and Y (maximum) values.
Duration	How long (in seconds) the spawner spawns particles for.
Parent emitter	The parent emitter, which should match the emitter's name set on that emitter.
Spawn Control Group	This field will be added to the parent particles for more precise control over which parent particle spawns how many children. There are 4 groups you can choose from and they should match the initializers' groups, if initializers require control.
Particles/trigger	How many particles (min and max) are spawned from a parent each time the triggering condition is met.
Particle Spawn Trigger	What condition triggers child particles spawning (detailed below)

Particle Spawn Trigger

- On Birth - Child particles are spawned once when a parent particle is born (once per parent)
- On Death - Child particles are spawned once when a parent particle dies (once per parent)
- Distance - Child particles are spawned per distance traveled as the parent particle moves
- On Hit - Parent particles need to implement *Collision Updater*. Child particles are spawned when a parent particle hits the surface.
- Lifetime - Child particles are spawned when the parent particle's lifetime is between two limits, A and B, expressed as normalized values (0 to 1) over the particle's lifetime. If A < B, the period is 0..(A..B)..1, if B > A the period is reversed to (0..B)..(A..1). This method is less precise than the On Birth/On Death conditions.

See also

- Create particles
- Emitters
- Shapes
- Materials
- Initializers
- Updaters

Particle initializers

Intermediate Artist Programmer

Initializers control the states of particles such as position, velocity, size, and so on when the particles are first spawned. They have no effect on particles spawned on previous frames.

> **NOTE**
>
> Some updaters act change the particle's value at the *end* of the frame. They effectively overwrite any initial values set by a similar initializer. Such is the case with all animations. They operate on the particle's lifetime and a color animation updater will overwrite any initial values from a color initializer.

Similarly, initializers which operate on the same field are exclusive and only the bottom one will have any effect, since they are executed in order. For example if you assign two color initializer, only the second one will have any effect.]

Common properties

Several properties are common across many initializers.

Property	Description
Debug draw	Draws a debug wireframe in the scene to show the boundaries of the initializer. This is only visible in the Scene Editor, not at runtime.
Position inheritance	Inherit the particle system component position, as defined in the particle entity's Transform component

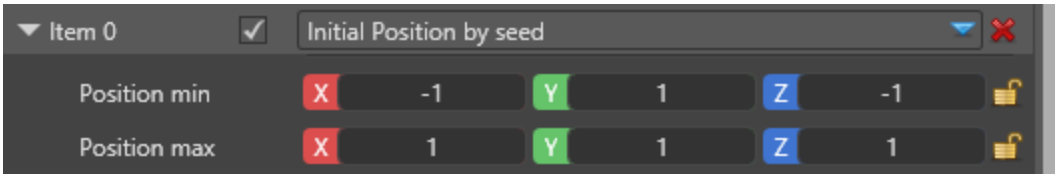
Property	Description
Position offset	Additional translation of the module. If it inherits the parent position, this is applied on top of the inherited position.
Rotation inheritance	Inherit the particle system component's rotation, as defined in the Transform component
Rotation offset	Additional rotation of the module. If it inherits the parent's rotation, this is applied on top of the inherited rotation.
Scale inheritance	Inherit the particle system component's uniform scale, as defined in the Transform component
Scale offset	Additional scaling of the module. If it inherits the parent's scale, this is applied on top of the inherited scale.

For example, a velocity initializer can change its direction depending on the parent's rotation or decide to ignore it and always shoot particles in a fixed direction.

On the other hand, size initializers don't change based on the parent's rotation, so the rotation fields won't appear at all.

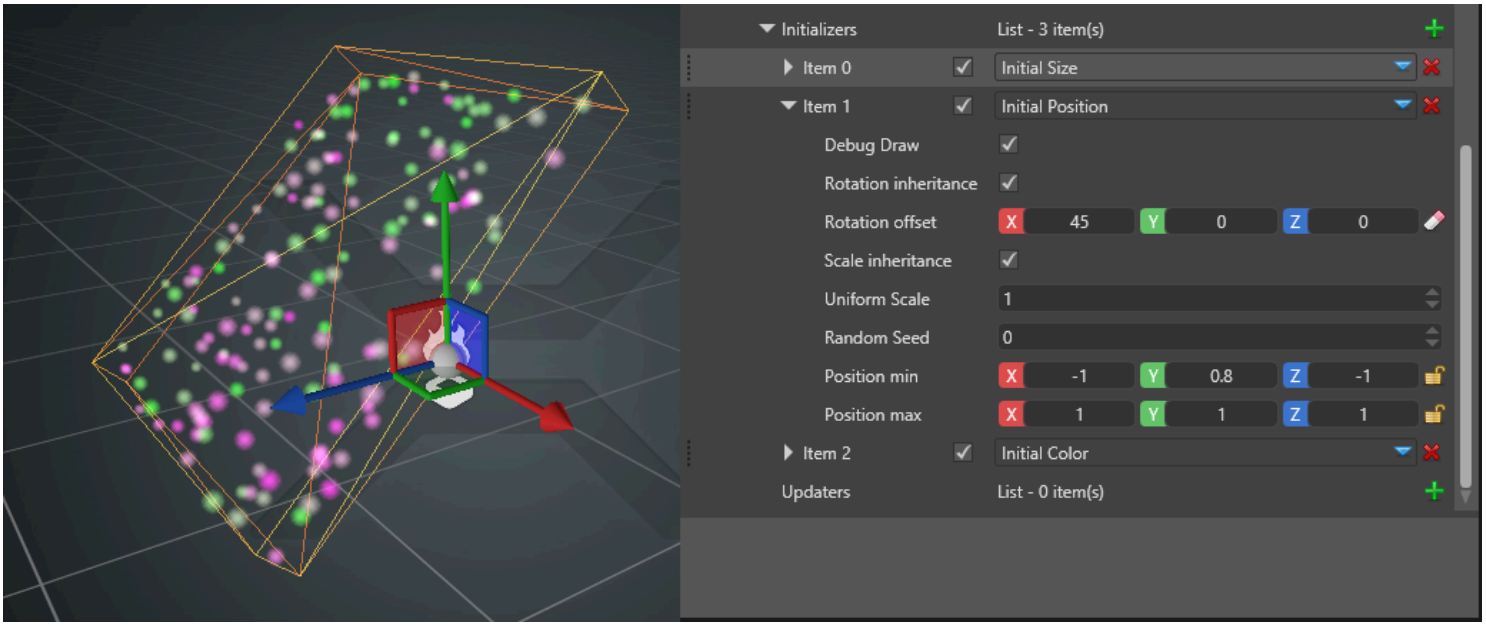
Position

Particles are spawned in an axis-aligned bounding box, defined by its left lower back corner and its right upper front corner.



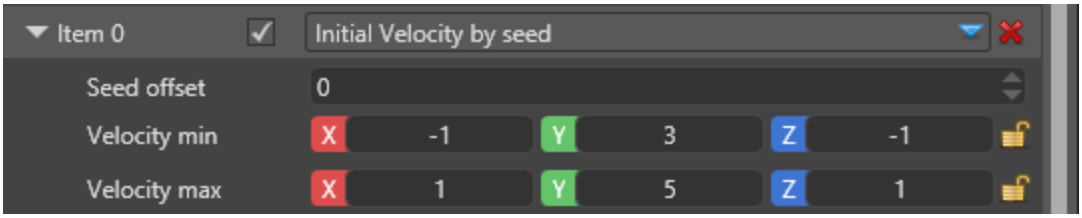
Property	Description
Seed offset	Used for random numbers. Set it to the same value to force the position to be coupled with other particle fields which have three properties (X, Y, Z), eg velocity. Make them different to force the position to be unique and independent from other fields
Position min	Left lower back corner for the box
Position max	Right upper front corner for the box

This image shows the bounding box where particles initially appear for this emitter. In addition to the corners (-1, 0.8, -1) ~ (1, 1, 1), the box is further rotated by 45 degrees as seen from the offset rotation.



Velocity

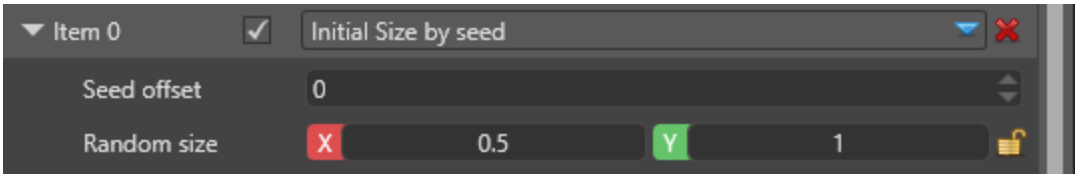
Particles spawn with initial velocity which ranges between the defined values. The velocity is independent in all three directions between X, Y and Z.



Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the velocity to be coupled with other particle fields which have 3 properties (X, Y, Z), like position for example. Make them different to force the velocity to be unique and independent from other fields.
Velocity min	Left lower back corner for the box
Velocity max	Right upper front corner for the box

Size

Initial size sets the particle's uniform size when it's spawned for the first time. A size of 1 will result in a 1 meter by 1 meter billboard or quad when rendered.



Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the size to be coupled with other particle fields which have 1 property, like color for example. Make them different to force the size to be unique and independent from other fields
Random size	Shows the minimum and maximum size a particle can have at spawn time

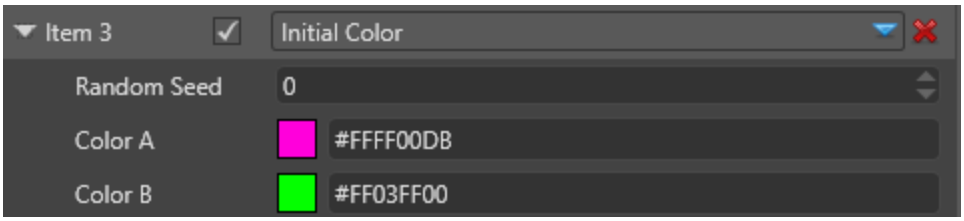
Rotation

Initial rotation sets the particle's angular rotation when facing the camera. Positive values are clockwise rotations. The field only has meaning for camera-facing particles, such as billboards. It has no effect on oriented quads and models.

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the angle to be coupled with other particle fields which have 1 property, like color for example. Make them different to force the angle to be unique and independent from other fields
Angle (degrees)	The minimum and maximum value, in degrees, for the initial rotation

Color

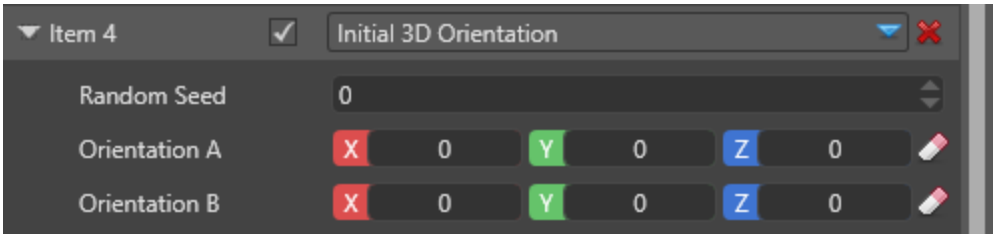
Initial color sets the particle's initial color at spawn time. It goes into the vertex buffer when building the particles and can be used by the material, but might not if the option is not set in the material itself. If setting the color has no effect please refer to the Material page for further discussion.



Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the color to be coupled with other particle fields which have 1 property, like size for example. Make them different to force the color to be unique and independent from other fields
Color A	The first value, in hexadecimal code. The color will be a random tint between this and the second color.
Color B	The second value, in hexadecimal code. The color will be a random tint between this and the first color.

3D Orientation

Initial 3D orientation sets the orientation for 3D-aware particles when they first spawn. The editable fields use euclidean rotation which is packed into a quaternion orientation by the engine. The interpolated value is on the shortest path between the two orientations, rather than interpolating each value separately.



Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the orientation to be coupled with other particle fields which have 1 property, like size for example. Make them different to force the orientation to be unique and independent from other fields.
Orientation A	The first oriented position
Orientation B	The second oriented position

Direction

This initializer creates the **Direction** field in the particle properties and sets its initial value. Some shape builders, like the Trail shape or the Direction Aligned Sprite shape use the particle's direction to properly display it.

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the direction to be coupled with other particle fields which have 3 properties (X, Y, Z), like position for example. Make them different to force the velocity to be unique and independent from other fields.
Direction min	Left lower back corner for the box
Direction max	Right upper front corner for the box

Spawn Order

This initializer has no properties. It simply sets an increasing number to each particle spawned from this emitter, starting from 0. The spawn order can be used for sorting or some custom calculations.

Position (Arc)

The arc position initializer positions the particles in an arc (or a straight line if the arc's height is 0) between two point, the emitter's position and a target transform component. With random position offset you can cause the particles to deviate a little from their original location on the arc.

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the position to be coupled with other particle fields which have 3 properties (X, Y, Z), like velocity for example. Make them different to force the position to be unique and independent from other fields.
Position min	Left lower back corner for the box
Position max	Right upper front corner for the box
Target	Allows you to pick up an Entity for the end of the arc. If no Entity is set, Fallback Target will be used, which is an offset from the emitter's location.
Fallback Target	Offset from the emitter's location used as the end point in case Target is not set
Arc Height	The height of the arc at its highest point (middle of the distance between the two points). By default it's the Y-up vector, but can be rotated with rotation offset and rotation inheritance
Ordered	If checked, new particles will appear in order from the emitter towards the target. If unchecked, new particles will appear randomly on the arc anywhere between the emitter and the target. If you plan to visualize the particles as a ribbon or a trail you should set this box to checked.
Fixed count	By default particles will appear on the arc at distances enough for the maximum number of particles to fit exactly on the line. If you want to control spawn rate and distance, you can set how many fixed "positions" are there on the arc. For example, with a fixed count of 10 and Ordered spawning, the first 10 particles will appear in order, then the 11th particle will appear from the beginning, at the same position as the first, and so on.
Seed offset	This is used for random numbers. Set it to the same value to force the position to be coupled with other particle fields which have 3 properties (X, Y, Z), like velocity for example. Make them different to force the position to be unique and independent from other fields.
Position min	Left lower back corner for the box. This is an offset in addition to the arc position.

Property	Description
Position max	Right upper front corner for the box. This is an offset in addition to the arc position.

Position (parent)

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the position to be coupled with other particle fields which have 3 properties (X, Y, Z), like velocity for example. Make them different to force the position to be unique and independent from other fields.
Position min	Left lower back corner for the box
Position max	Right upper front corner for the box
Parent emitter	You have to type the name of the parent emitter. Child particles' positions will match the parent emitter's particles' positions.
Parent Offset	Random seed used to couple or decouple the way a parent particle is chosen. For example, if you want to pick position *and* color from seemingly random particles, you can use the same offset. If you want to avoid such connection, you can use different offsets for position and color initializers.
Spawn Control Group	When None, parents will be picked randomly. When set to one of the four groups, only particles from a specific parent will be initialized. It should match a control group from the **Spawn from Parent** spawner to work properly.

Velocity (parent)

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the velocity to be coupled with other particle fields which have 3 properties (X, Y, Z), like position for example. Make them different to force the velocity to be unique and independent from other fields.
Velocity min	Left lower back corner for the box

Property	Description
Velocity max	Right upper front corner for the box
Parent emitter	You have to type the name of the parent emitter. Child particles' positions will match the parent emitter's particles' positions.
Parent Offset	Random seed used to couple or decouple the way a parent particle is chosen. For example, if you want to pick position *and* color from seemingly random particles, you can use the same offset. If you want to avoid such connection, you can use different offsets for position and color initializers.
Spawn Control Group	When None, parents will be picked randomly. When set to one of the four groups, only particles from a specific parent will be initialized. It should match a control group from the *Spawn from Parent* spawner to work properly.

Size (parent)

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the size to be coupled with other particle fields which have 1 property, like color for example. Make them different to force the size to be unique and independent from other fields.
Random size	Shows the minimum and maximum size a particle can have at spawn time
Parent emitter	You have to type the name of the parent emitter. Child particles' positions will match the parent emitter's particles' positions.
Parent Offset	Random seed used to couple or decouple the way a parent particle is chosen. For example, if you want to pick position *and* color from seemingly random particles, you can use the same offset. If you want to avoid such connection, you can use different offsets for position and color initializers.
Spawn Control Group	When None, parents will be picked randomly. When set to one of the four groups, only particles from a specific parent will be initialized. It should match a control group from the *Spawn from Parent* spawner to work properly.

Color (parent)

Property	Description
Seed offset	This is used for random numbers. Set it to the same value to force the color to be coupled with other particle fields which have 1 property, like size for example. Make them different to force the color to be unique and independent from other fields.
Color A	The first value, in hexadecimal code. The color will be a random tint between this and the second color.
Color B	The second value, in hexadecimal code. The color will be a random tint between this and the first color.
Parent emitter	You have to type the name of the parent emitter. Child particles' positions will match the parent emitter's particles' positions.
Parent Offset	Random seed used to couple or decouple the way a parent particle is chosen. For example, if you want to pick position *and* color from seemingly random particles, you can use the same offset. If you want to avoid such connection, you can use different offsets for position and color initializers.
Spawn Control Group	When None, parents will be picked randomly. When set to one of the four groups, only particles from a specific parent will be initialized. It should match a control group from the *Spawn from Parent* spawner to work properly.

Spawn Order (parent)

This initializer requires the parent emitter to also have a Spawn Order initializer. It combines the parent's spawn number with its own, effectively creating groups of particles among the children. This initializer is required to properly sort and render child ribbon particles.

Property	Description
Parent emitter	You have to type the name of the parent emitter. Child particles' positions will match the parent emitter's particles' positions.
Parent Offset	Random seed used to couple or decouple the way a parent particle is chosen. For example, if you want to pick position *and* color from seemingly random particles, you can use the same offset. If you want to avoid such connection, you can use different offsets for position and color initializers.

Property	Description
Spawn Control Group	When None, parents will be picked randomly. When set to one of the four groups, only particles from a specific parent will be initialized. It should match a control group from the *Spawn from Parent* spawner to work properly.

See also

- Create particles
- Emitters
- Shapes
- Materials
- Spawners
- Updaters

Particle updaters

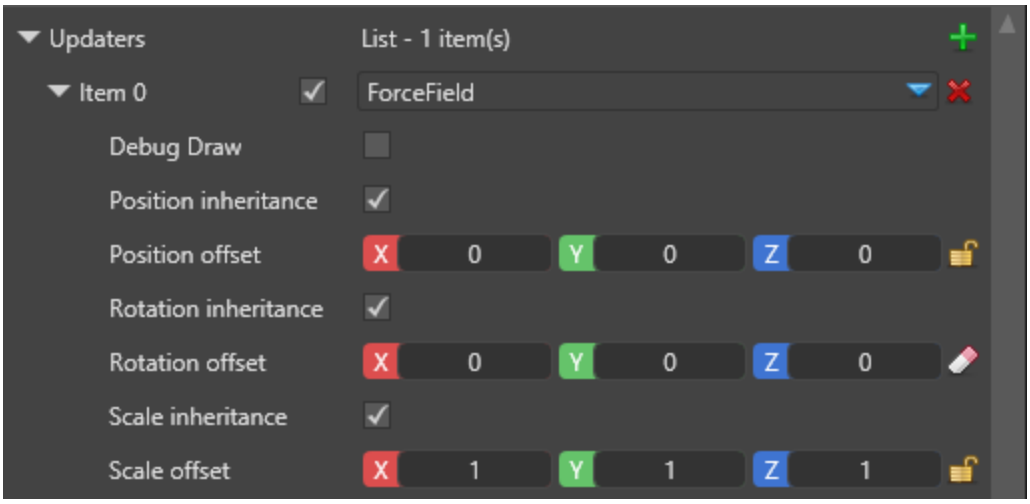
Intermediate Artist Programmer

After a particle spawns, it can change over time before it disappears. **Updaters** act on all living particles over time, changing attributes such as position, velocity, color, and so on. For example, a gravity force updates the particle's velocity at a constant rate, accelerating it toward the ground.

Stride features several built-in updaters. The custom particles sample demonstrates how you can add updaters to the engine.

Common properties

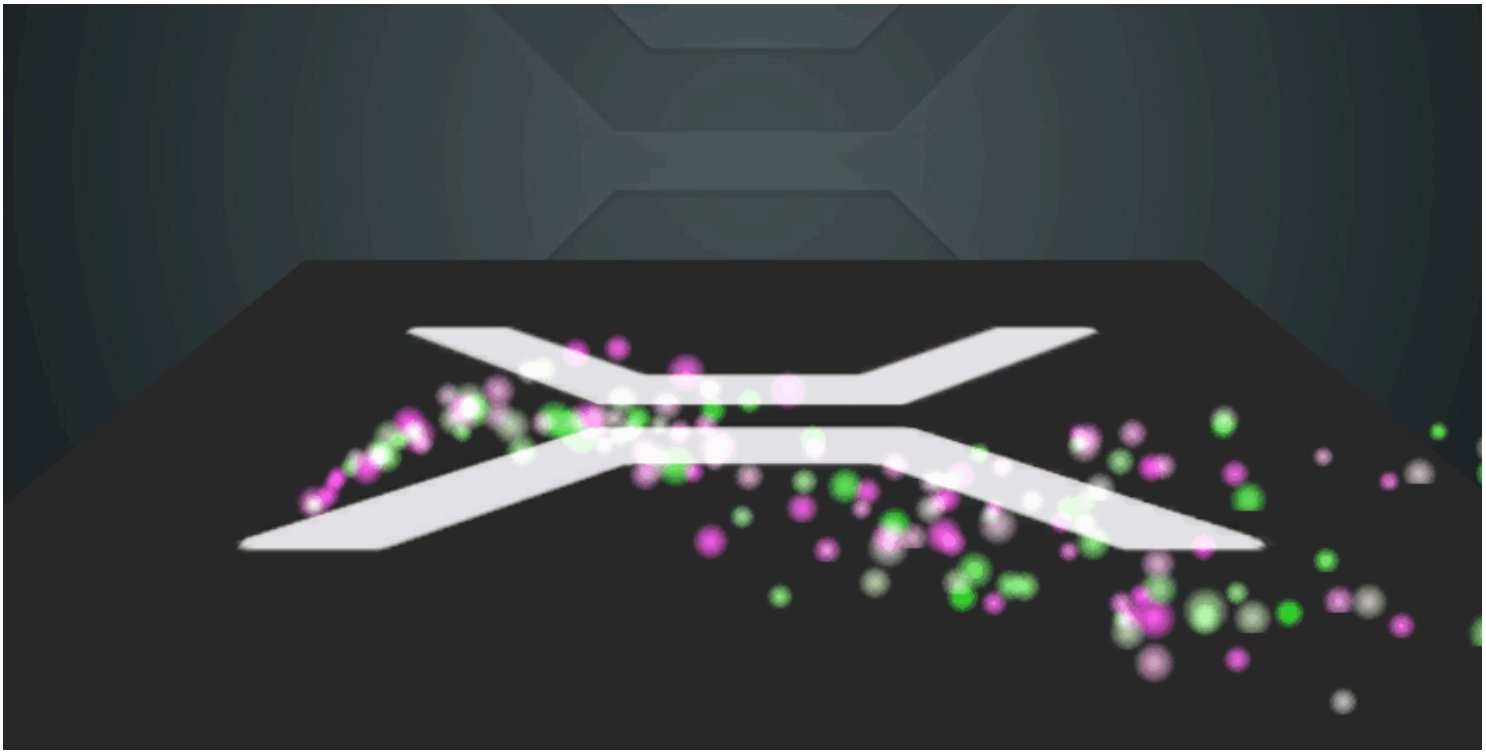
Several properties are common across many updaters.



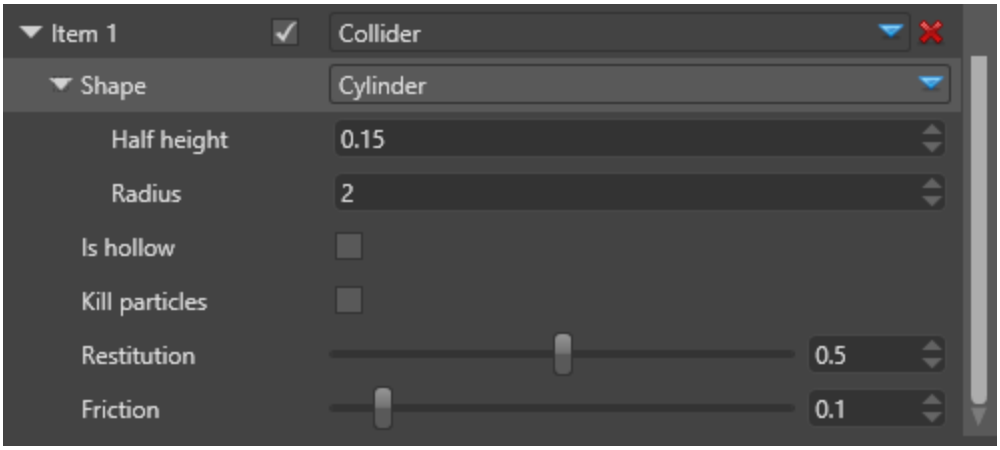
Property	Description
Debug draw	Draws a debug wireframe shape to show the boundaries for the updater. This feature only works for the editor and is ignored when you run your game.
Position inheritance	Inherit the particle system component position, as defined in the Transform field
Position offset	Additional translation of the module. If it inherits the parent's position, this is applied on top of the inherited one.
Rotation inheritance	Inherits the particle system component rotation, as defined in the Transform field
Rotation offset	Additional rotation of the module. If it inherits the parent's rotation, this is applied on top of the inherited one.

Property	Description
Scale inheritance	Inherits the particle system component's uniform scale, as defined in the Transform field.
Scale offset	Additional module scaling. If it inherits the parent scale, this is applied on top of the inherited one.

Collider

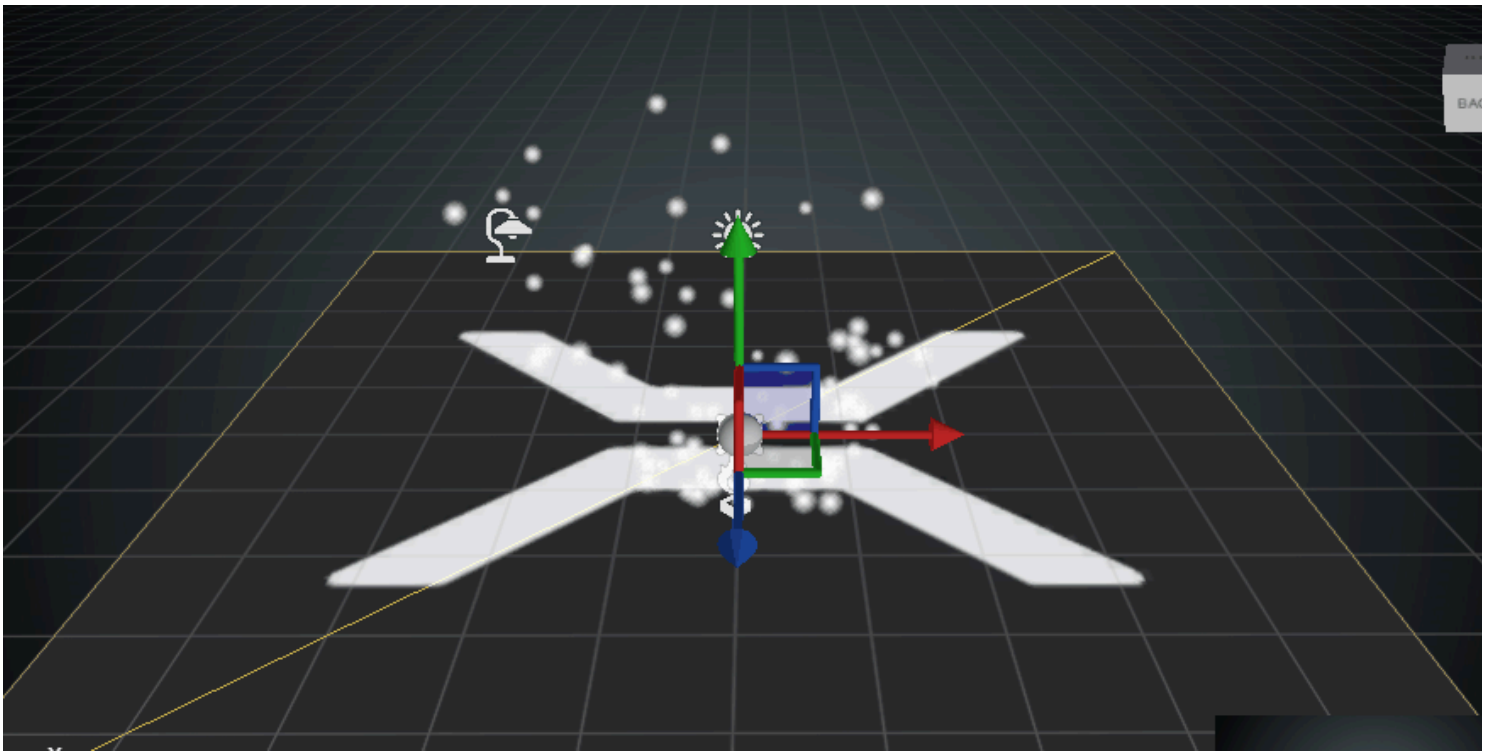


A **collider** is an updater that changes the particle position and velocity when it collides with a predefined shape.

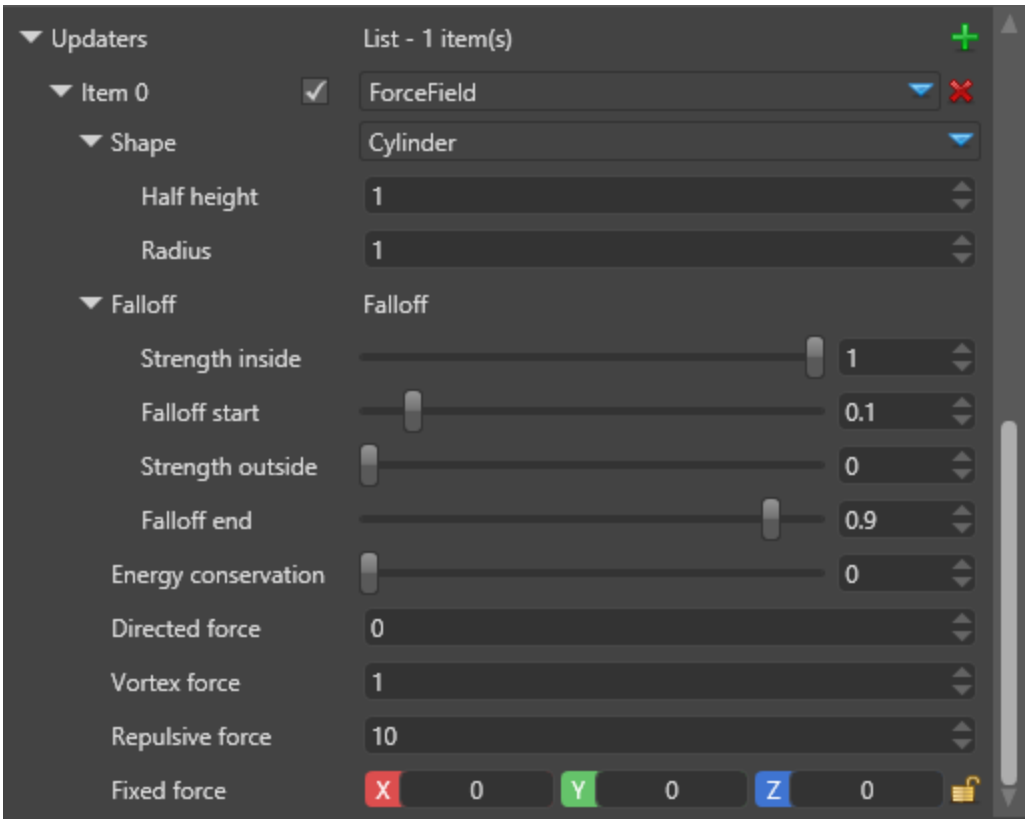


Property	Description
Shape	The shape the particles collide with (sphere, cylinder, box, or torus)
Is hollow	If disabled, the shape is solid and the particles bounce off it. If enabled, the shape is hollow like a container, and the particles stay inside the volume.
Kill particles	If enabled, the particles are killed immediately when they collide with the shape.
Restitution	The coefficient of restitution is the speed the particle retains in comparison to its speed before the collision. In this updater we use restitution as a *vertical only* speed. It doesn't affect the speed along the surface.
Friction	The amount of horizontal speed the particle loses on collision with the shape. It only affects the speed along the surface, and doesn't change the height at which the particle bounces.

Force field



The **force field** is defined by a bounding shape and several force vectors that operate on the particles based on their relative position to the bounding shape.



Property	Description
Shape	The bounding shape (sphere, cylinder, box or torus)
Falloff	The falloff is a simple linear function which dictates the intensity of the force applied on particles. It is based on the particle's distance from its center. Strength inside is how much of the magnitude applies when the particle is within *falloff start* distance from the center. Strength outside is how much of the magnitude applies when the particle is more than *falloff end* away from the center. Both values are relative to the bounding shape size; values inbetween are interpolated between the two magnitudes. Values in the center can still be 0, making the force only work *outside* the bounding shape.
Energy conservation	Which part of the force energy conserved by the particles. Conserved energy is stored as particle velocity and results in gradually increasing speed. Energy not conserved directly applies to the particle's position and is lost when the force vanishes.
Directed force	The vector force that moves the particle along the field's central axis (normally upwards)
Vortex force	The force that moves the particle around the field's central axis using the right-hand rule for rotation

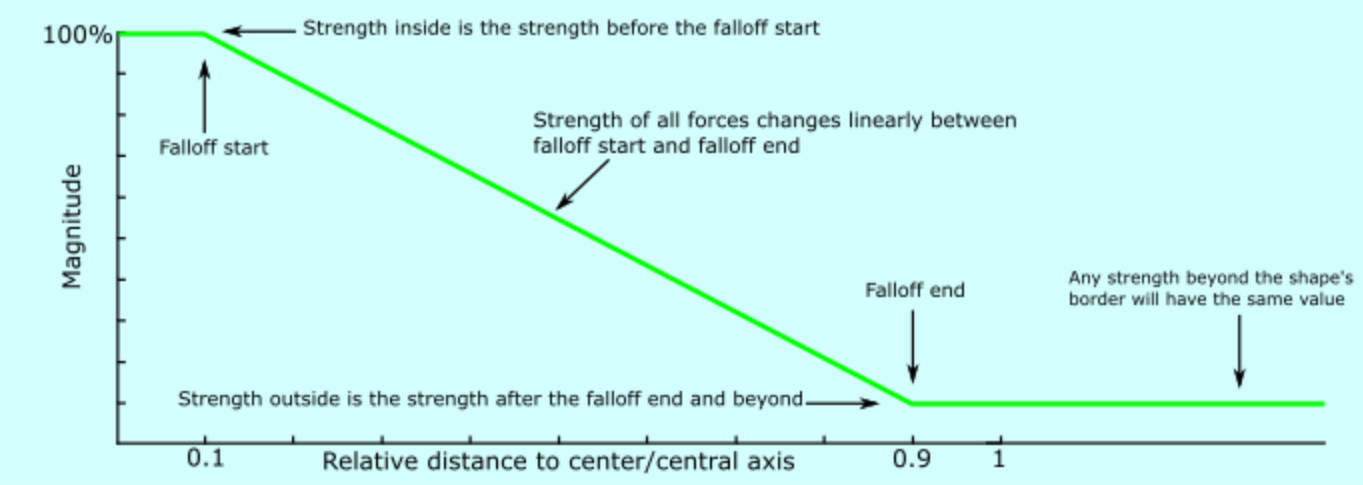
Property	Description
Repulsive force	The force that moves the particle away from the field's center or towards it, if negative
Fixed force	The force that moves the particle along a fixed non-rotating and non-scaling axis

Falloff

The **falloff** is the change in the forces' strength based on the distance of the particle from the shape's center. The falloff is a function of the relative distance, where distance of `0` is the center, 1 is the shape's boundaries, and more than 1 means the particle is outside the shape.

Particles closer than the falloff start are always affected with the coefficient Strength Inside. Particles farther than falloff end are always affected with the coefficient Strength Outside.

Coefficient for particles in between changes linearly:



The strength coefficient for all forces is a function of the relative distance of the particle from the shape's center, with 1 being the shape's boundaries, and more than 1 being outside the shape.

For example, if the bounding shape is a sphere with a radius of 10m, particles within 1m from its center (0.1 x 10m) will be moved with full strength. After the 1m distance the strength linearly decreases until it reaches zero at 9m distance (0.9 x 10m). After that point, the forces don't affect the particle.

Bounding shapes

Sphere

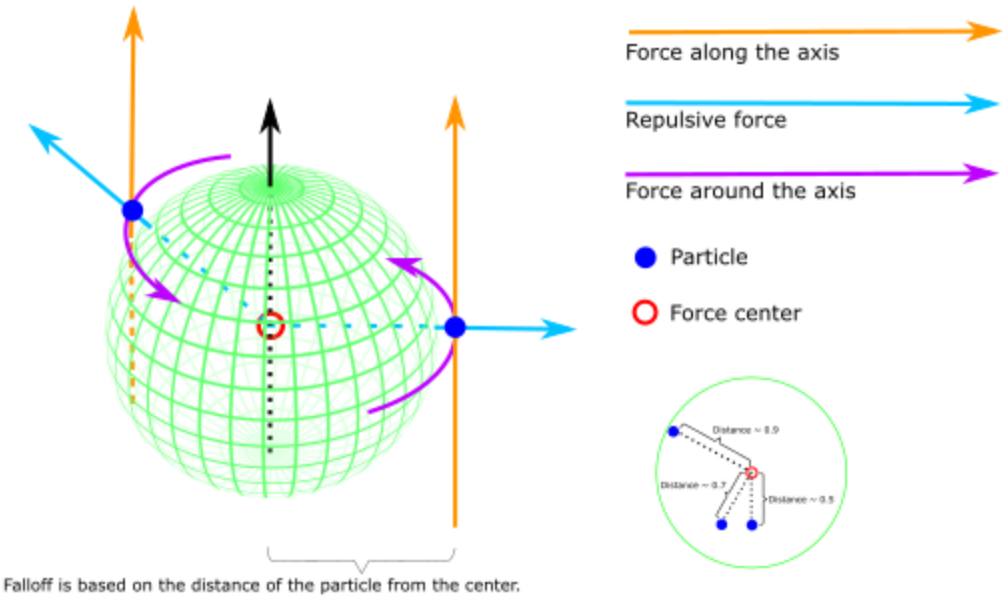
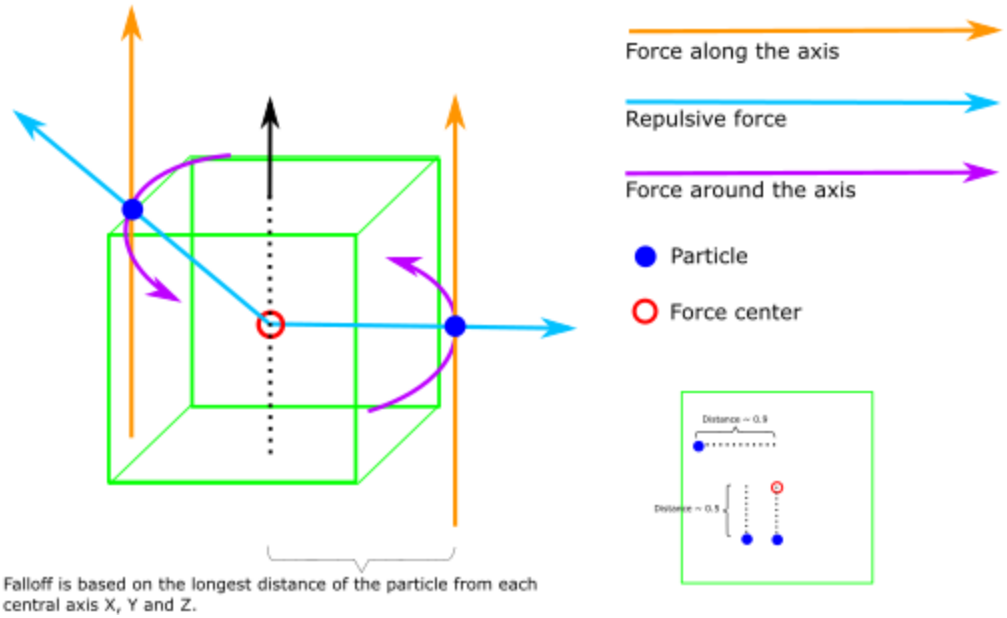


Image license: CC-BY-SA 4.0, sphere image from the "Sphere wireframe" work by Geek3 under CC-BY-SA 3.0

When the bounding shape is a sphere, the falloff distance is based on the radial distance of the particle from the sphere's center. If the sphere is scaled to an ellipsoid, this distance is also scaled. The distance is relative to the radius, with 1.0 being the sphere's surface.

The directed force vector is parallel to the sphere's local Y axis. The repulsive force vector points from the center to the particle. The vortex force vector goes around the sphere's Y axis at the particle's position (using the right-hand rule for rotation).

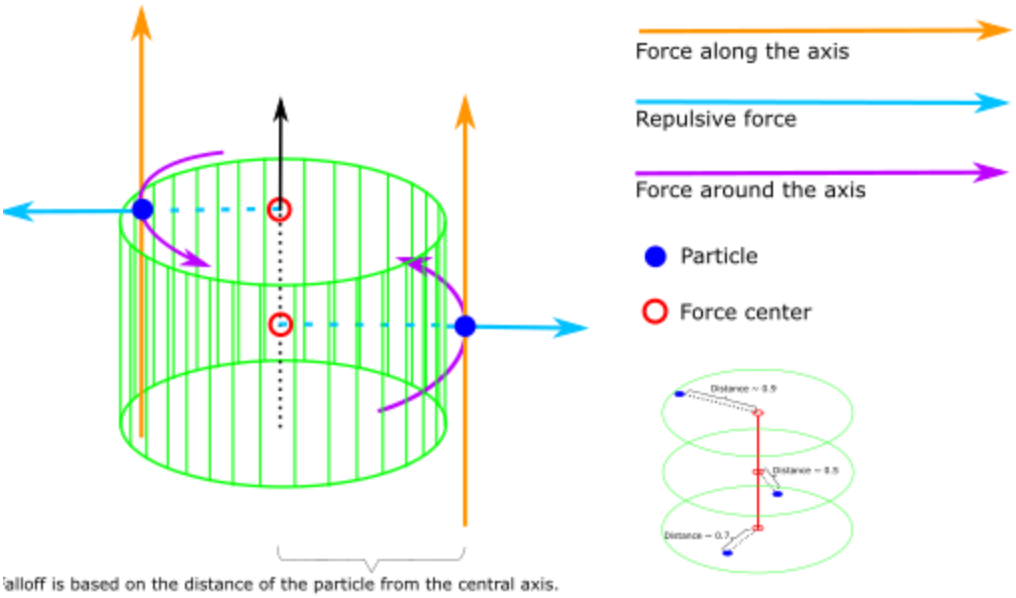
Box



When the bounding shape is a box, the falloff distance is the longest of the three distances on the X, Y and Z axes. The distance is relative to the box's sizes, with 1.0 being the box's surface.

The directed force vector is parallel to the box's local Y axis. The repulsive force vector points from the center to the particle. The vortex force vector goes around the box's Y axis at the particle's position (using the right-hand rule for rotation).

Cylinder



When the bounding shape is a cylinder, the falloff distance is based on the radial distance of the particle from the cylinder's local Y axis. The particle height (position on the Y axis) is ignored unless the particle is outside the cylinder, in which case the distance is always 1.

The directed force vector is parallel to the cylinder's local Y axis. The repulsive force vector points from the cylinder's local Y axis to the particle, so the repulsive force is always horizontal. The vortex force vector goes around the cylinder's Y axis at the particle position (using the right-hand rule for rotation).

Torus

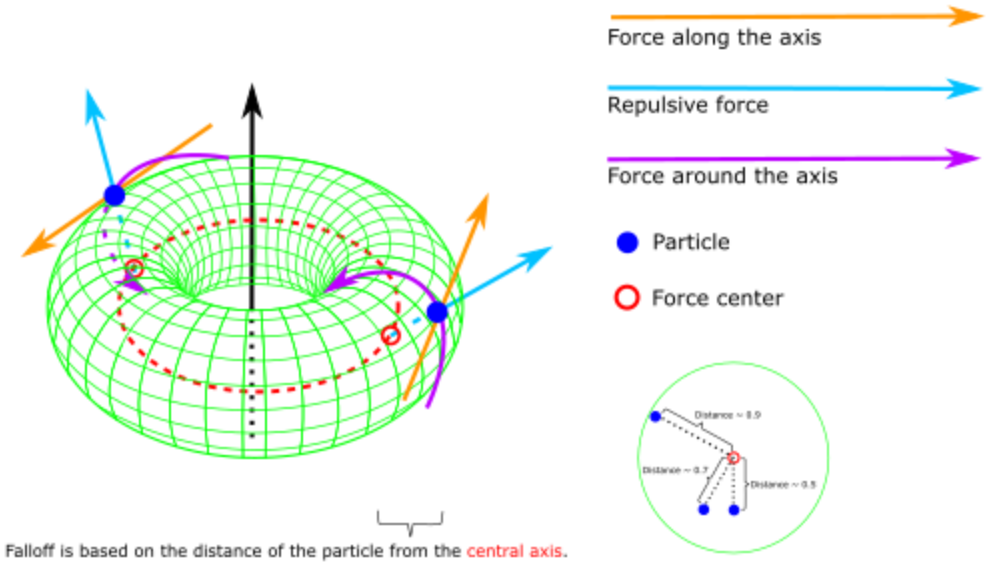


Image license: GFDL, CC-BY-SA 4.0, torus image from the "A simple Torus" work by Yassine Mrabet under GFDL, CC-BY-SA 3.0

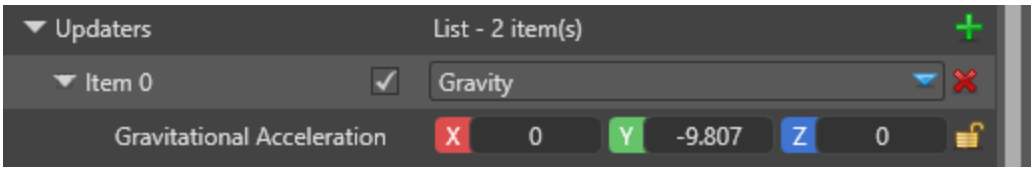
When the bounding shape is a torus, the field's nature changes completely. The falloff distance is based on the radial distance of the particle from the torus's inner circle (axis of revolution, shown in red), choosing a point on the circle closest to the particle.

The directed force vector is tangent to the axis of revolution at the point closest to the particle. The repulsive force vector points from the axis to the particle. The vortex force vector goes around the directed force vector using the particle's position relative to the axis (using the right-hand rule for rotation).

While the math is a little complicated, using the torus force field isn't. Try it out!

Gravity

The **gravity updater** is a simplified force which affects all particles regardless of their position, with a constant force vector which doesn't scale or rotate. It's editable, so you can use it in projects with different scales and behavior.



The gravity force ignores most properties such as offset and inheritance, and only uses the following attributes:

Property	Description
Gravitational acceleration	The gravity force vector that defines the acceleration for all affected particles. The default value matches the average gravity on Earth.

Direction from speed

Direction from speed is a post-updater, meaning it resolves after updaters which are not post-updaters, even if they appear later in the list.

It has no properties and simply updates the particle's direction to match its speed. It uses the difference between the positions of the particle from the last frame and isn't directly dependent on velocity. This means even if the particle's own velocity is 0 and it's only moved by external forces, direction from speed resolves correctly.

Direction isn't a normalized vector and changes its magnitude to match the delta distance. It overwrites any previous direction parameters, such as from an initializer.

Color animation

Color animation is a post-updater, meaning it resolves after updaters which aren't post-updaters, even if they appear later in the list.

Color animation updates the particle Color field by sampling a curve over the particle's normalized lifetime (0 to 1). You can set a secondary curve in which case the particles will have slightly varied colors. Color animation overwrites any previous Color parameters, such as Initial Color.

The curve values are given as Vector4, corresponding to RGBA with standard values between 0 and 1. Values above 1 are valid for RGB only (not Alpha) and can be used for HDR rendering.

Rotation animation

Rotation animation is a post-updater, meaning it resolves after updaters which are not post-updaters, even if they appear later in the list. It's strictly a single axis rotation, used for billboarded particles.

Rotation animation updates the particle's Rotation field by sampling a curve over the particle's normalized lifetime (0 to 1). You can set a secondary curve in which case the particles will have slightly varied rotations.

Rotation animation overwrites any previous Rotation parameters, such as Initial Rotation. If you need additive kind of animation check if the Shape Builder supports it (found in the Shape Builder's properties). Additive animations are not preserved in particle fields and do not persist, but can be applied in addition to any fields the particles already have.

Size animation

Size animation is a post-updater, meaning it resolves after updaters which aren't post-updaters, even if they appear later in the list.

This is strictly a uniform size. Size animation updates the particle's Size field by sampling a curve over the particle's normalized lifetime (0 to 1). You can set a secondary curve, in which case the particles have slightly varied sizes.

Size animation overwrites any previous Size parameters, such as Initial Size. If you need additive kind of animation, check if the Shape Builder supports it (in the Shape Builder properties). Additive animations aren't preserved in particle fields and don't persist, but can be applied in addition to any fields the particles already have.

- Create particles
- Emitters
- Shapes
- Materials

- Spawners
- Initializers

Tutorials

- Inheritance
- Lasers and lightning
- Create a trail
- Custom particles
- Particle materials

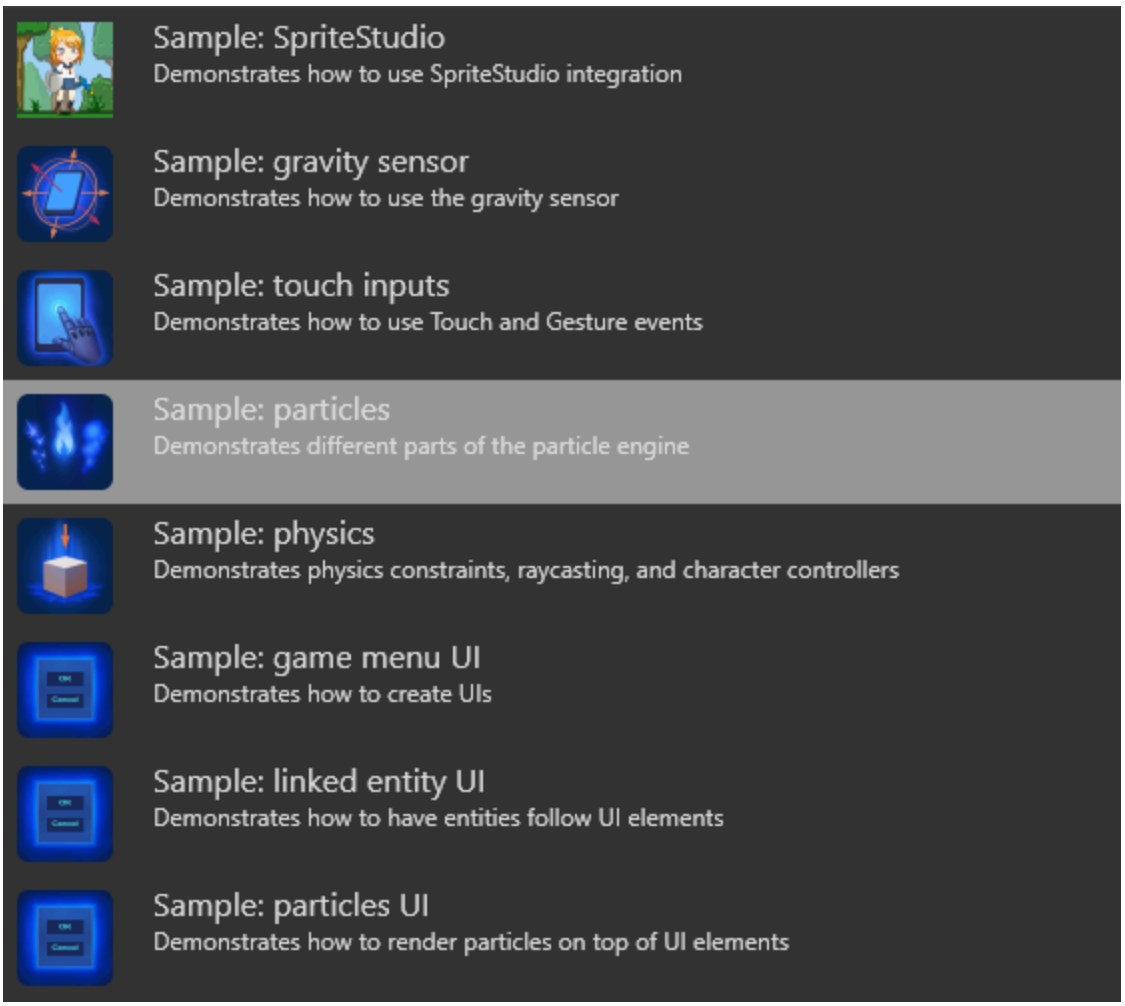
Tutorial: Particle materials

Intermediate Artist Programmer

This tutorial demonstrates how to create custom shaders and materials for a particle system, providing functionality not available in the core engine. It focuses on shaders and rendering. For simulation, see the custom particles tutorial.

If you're not familiar with editing particles, see Create particles.

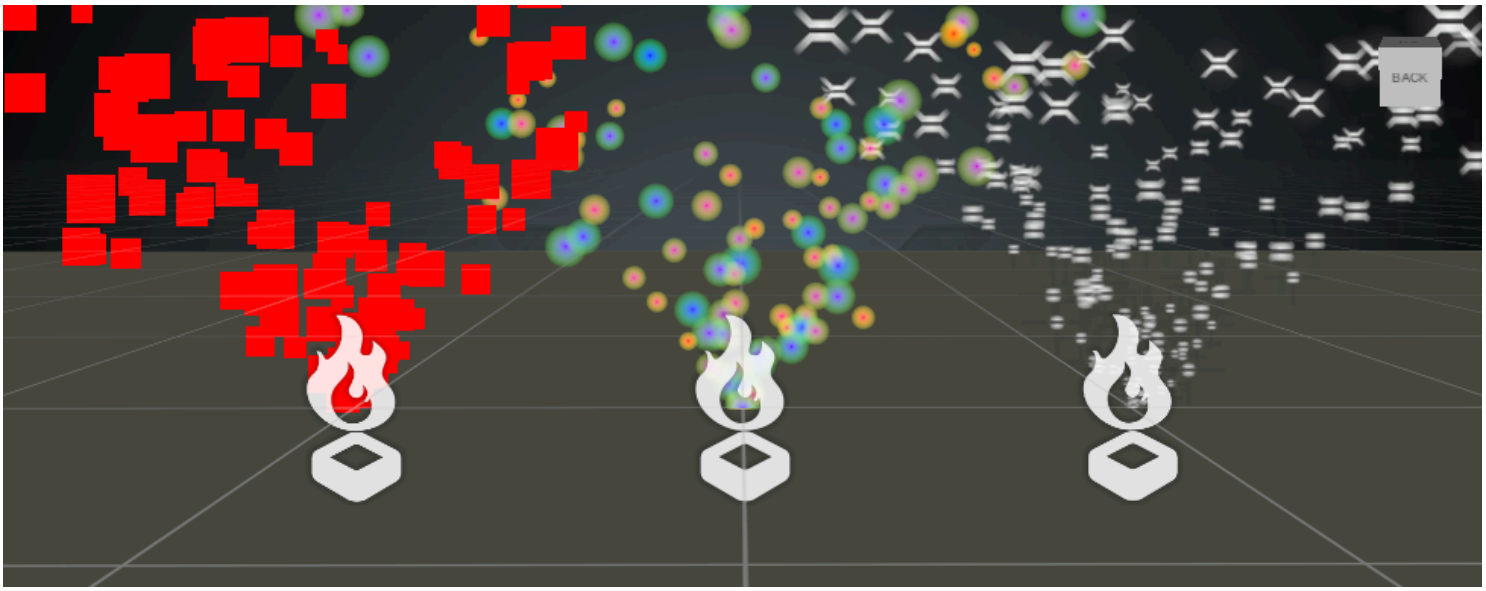
Start by creating a new **Sample: Particles** project.



This project contains four scenes, each demonstrating a different way to use particles: **AnimatedParticles**, **ChildParticles**, **CustomMaterials**, and **CustomParticles**.

Open the **CustomMaterials** scene.

There are three particle entities in the scene: **Rad Particle System**, **Radial Particle System**, and **Two Textures Particle System**.



Select one of the particle entities and navigate to its source particle system, expanding the emitter in it and its material.

Red particle system

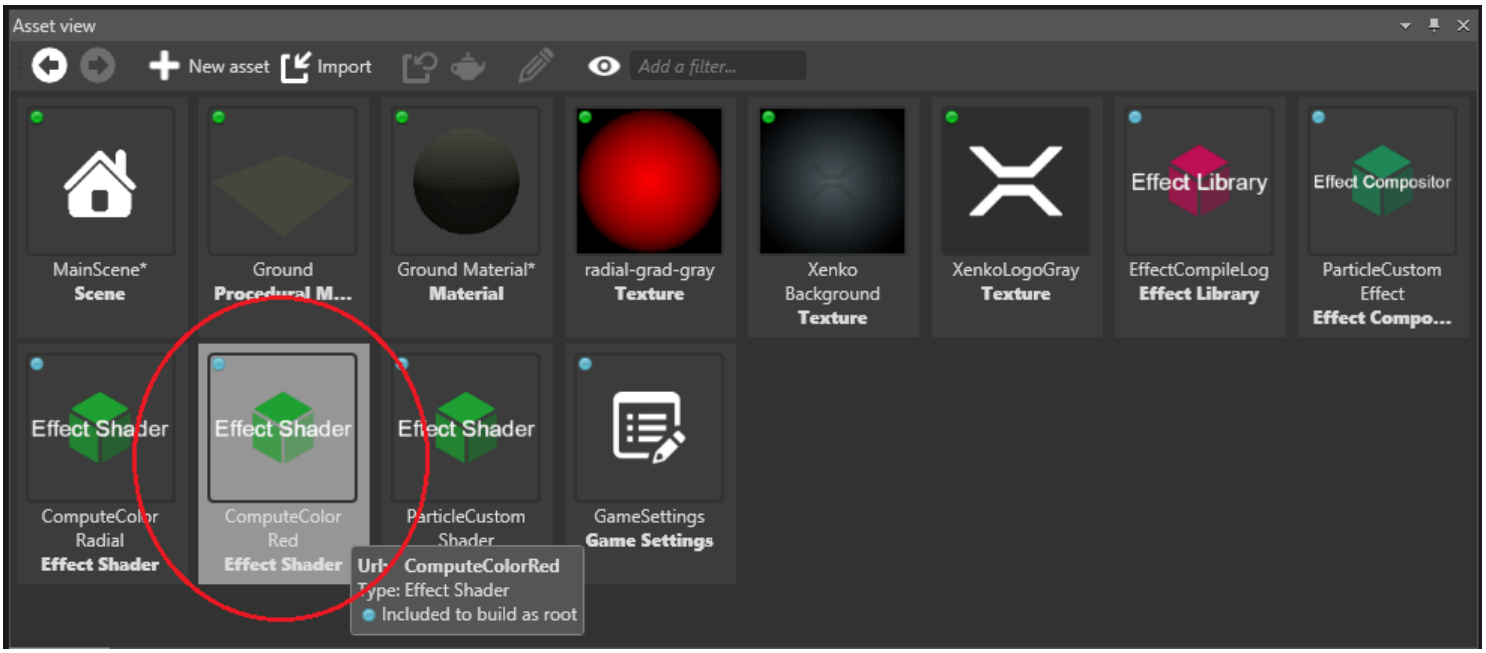
The **red particle system** has a very simple customization. Since the material maps already provide an option to use shaders as a leaf node input, we can create a custom shader and assign it to that node.

First, create a shader (`ComputeColorRed.sdsl`) with a derived class for `ComputeColor`:

```
class ComputeColorRed : ComputeColor
{
  override float4 Compute()
  {
      return float4(1, 0, 0, 1);
  }
};
```

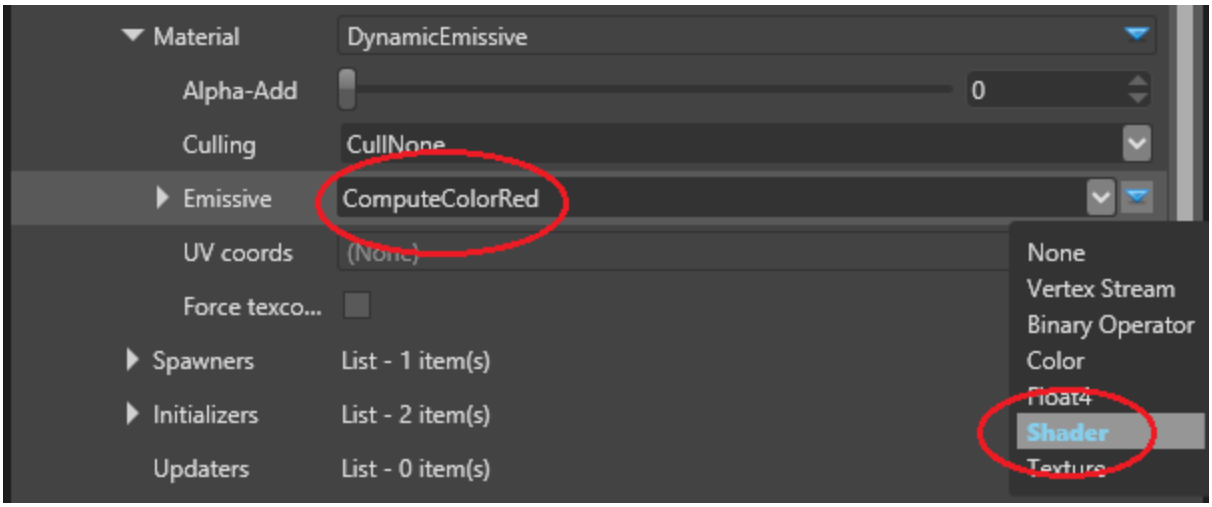
The only thing this shader does is return the red color for pixel shading every time `Compute` is called. We'll try something more difficult later, but for now let's keep it simple.

Save the file and reload the scripts in Game Studio. You should see the new shader in **Asset View**.



If the shader isn't there, reload the project.

Once the shader is loaded, you can access it in the **Property Grid** under the **dynamic emissive material** for the particles. Choose a type of shader and, from the drop-down menu, select the shader you just added to the scene.



The particles are red. With Game Studio running, edit and save `ComputeColorRed.sdsl` to make them yellow.

```
class ComputeColorRed : ComputeColor
{
  override float4 Compute()
  {
      return float4(1, 1, 0, 1);
  }
};
```

Because Stride supports dynamic shader compilation, the particles immediately turn yellow.

Radial particle system

For the next shader we'll use texture coordinates expose arbitrary values to the editor.

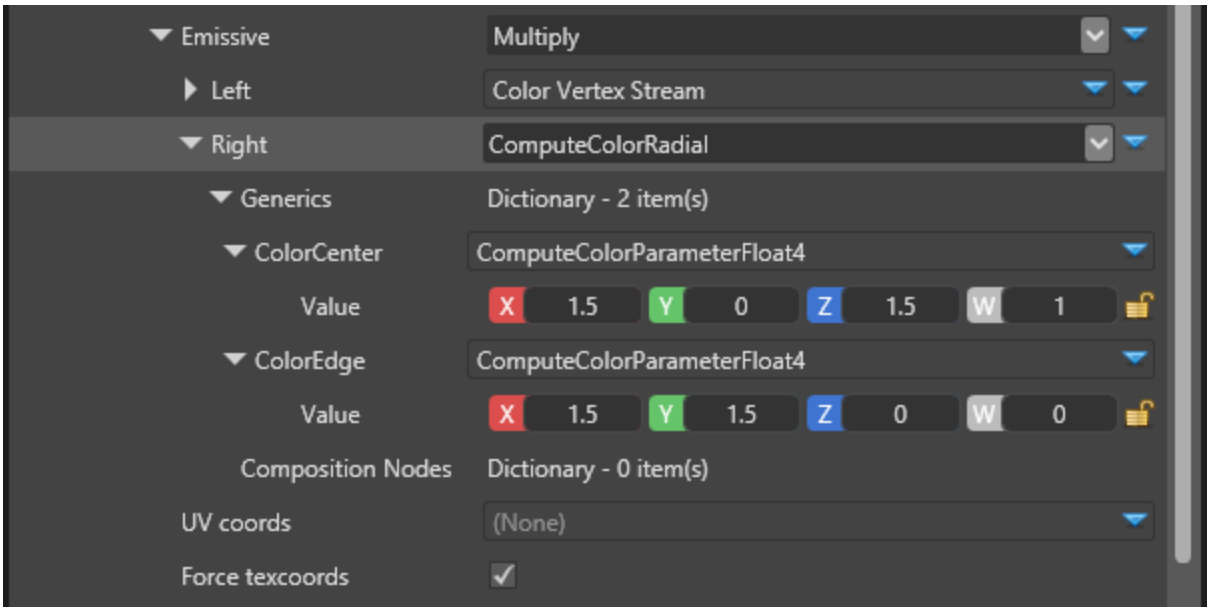
Check `ComputeColorRadial.sdsl`.

```
class ComputeColorRadial<float4 ColorCenter, float4 ColorEdge> : ComputeColor, Texturing
{
  override float4 Compute()
  {
      float radialDistance = length(streams.TexCoord - float2(0.5, 0.5)) * 2;

      float4 unclamped = lerp(ColorCenter, ColorEdge, radialDistance);

      // We want to allow the intensity to grow a lot, but cap the alpha to 1
      float4 clamped = clamp(unclamped, float4(0, 0, 0, 0), float4(1000, 1000, 1000, 1));

      // Remember that we use a premultiplied alpha pipeline so all color values should
be premultiplied
      clamped.rgb *= clamped.a;

      return clamped;
  }
};
```

This is similar to `ComputeColorRed` and can be compiled and loaded the same way.

There are several key differences. The shader now inherits from the `Texturing` shader base class as well. This allows it to use texture coordinates in from the streams. On the material side in Game Studio, we can force the texture coordinates to be streamed in case we don't use texture animation.

The input values `float4 ColorCenter` and `float4 ColorEdge` in our shader are permutations. When we load the shader the Property Grid displays them under the **Generics** dictionary.



The values we set here will be used by the `ComputeColorRadial` shader for the particles. The rest of the shader simply calculates a gradient color based on the distance of the shaded pixel from the center of the billboard.

Two-texture particle system

This demonstrates how to create custom materials and effects for the particles. The `DynamicColor` material supports one RGBA channel. For our sample, we'll separate the RGB and A channels, allowing them to use different texture coordinate animations and different textures and binary trees to compute the color.

Parameter keys

Parameter keys are used to map data and pass it to the shader. Some of them are generated, and we can define our own too.

If we define more streams in our shader (`ParticleCustomShader`), they're exported to an automatically generated class. Try adding the following to `ParticleCustomShader.sdsl`:

```
    // ------------------------------------
    // streams
    // ------------------------------------
    stage float4 SomeRandomKey;
```

The generated .cs file should now contain:

```
  namespace Stride.Rendering
  {
    public static partial class ParticleCustomShaderKeys
```

```
  {
      public static readonly ParameterKey<Vector4> SomeRandomKey =
ParameterKeys.New<Vector4>();
  }
}
```

We don't need this stream for now, so we can delete it.

We'll define some extra keys in `ParticleCustomMaterialKeys.cs` to use in our material and effects.

```
namespace Stride.Rendering
{
  public partial class ParticleCustomShaderKeys
  {
      static ParticleCustomShaderKeys()
      {

      }

      public static readonly ParameterKey<ShaderSource> BaseColor     =
ParameterKeys.New<ShaderSource>();

      public static readonly ParameterKey<Texture> EmissiveMap =
ParameterKeys.New<Texture>();
      public static readonly ParameterKey<Color4> EmissiveValue =
ParameterKeys.New<Color4>();



      public static readonly ParameterKey<ShaderSource> BaseIntensity =
ParameterKeys.New<ShaderSource>();

      public static readonly ParameterKey<Texture> IntensityMap =
ParameterKeys.New<Texture>();
      public static readonly ParameterKey<float> IntensityValue =
ParameterKeys.New<float>();
  }
}
```

As we saw above, the generated class has the same name and the namespace is `Stride.Rendering`, so we have to make our class partial and match the namespace. This has no effect on this specific sample, but will result in compilation error if your shader code auto-generates some keys.

The rest of the code is self-explanatory. We'll need the map and value keys for shader generation later, and we'll set our generated code to the `BaseColor` and `BaseIntensity` keys respectively so the shader can use it.

Custom Shader

Let's look at `ParticleCustomShader.sdsl`:

```
class ParticleCustomShader : ParticleBase
{
  // This shader can be set by the user, and it's a binary tree made up from smaller shaders
  compose ComputeColor  baseColor;

  // This shader can be set by the user, and it's a binary tree made up from smaller shaders
  compose ComputeColor  baseIntensity;

  // Shading of the sprite – we override the base class's Shading(), which only
returns ColorScale
  stage override float4 Shading()
  {
      // ------------------------------------------------
      // Base particle color RGB
      // ------------------------------------------------
      float4 finalColor = base.Shading() * baseColor.Compute();

      // ------------------------------------------------
      // Base particle alpha
      // ------------------------------------------------
      finalColor.a    = baseIntensity.Compute();

      //  Don't forget to premultiply the alpha
      finalColor.rgb *= finalColor.aaa;

      return finalColor;
  }
};
```

It defines two composed shaders, `baseColor` and `baseIntensity`, where we'll plug our generated shaders for RGB and A respectively. It inherits `ParticleBase` which already defines `VSMain`, `PSMain` and texturing, and uses very simple `Shading()` method.

By overriding the `Shading()` method we can define our custom behavior. Because the composed shaders we use are derived from `ComputeColor`, we can easily evaluate them using `Compute()`, which gives us the root of the compute tree for color and intensity.

Custom effect

Our effect describes how to mix and compose the shaders. It's in `ParticleCustomEffect.sdfx`:

```
namespace Stride.Rendering
{
  partial shader ParticleCustomEffect
  {
      // Use the ParticleBaseKeys for constant attributes, defined in the game engine
      using params ParticleBaseKeys;

      // Use the ParticleCustomShaderKeys for constant attributes, defined in this project
      using params ParticleCustomShaderKeys;

      // Inherit from the ParticleBaseEffect.sdfx, defined in the game engine
      mixin ParticleBaseEffect;

      // Use the ParticleCustomShader.sdsl, defined in this project
      mixin ParticleCustomShader;

      // If the user-defined shader for the baseColor is not null use it
      if (ParticleCustomShaderKeys.BaseColor != null)
      {
          mixin compose baseColor = ParticleCustomShaderKeys.BaseColor;
      }

      // If the user-defined shader for the baseIntensity (alpha) is not null use it
      if (ParticleCustomShaderKeys.BaseIntensity != null)
      {
          mixin compose baseIntensity = ParticleCustomShaderKeys.BaseIntensity;
      }

  };

}
```

`ParticleBaseKeys` and `ParticleBaseEffect` are required by the base shader which we inherit.

`ParticleCustomShaderKeys` provides the keys we defined earlier, where we'll plug our shaders.

Finally, for both shaders we only need to check if there is user-defined code for it and plug it. The `baseColor` and `baseIntensity` parameters are from the shader we created earlier.

Last, we need a material which sets all the keys and uses the newly created effect.

Custom particle material

We'll copy ParticleMaterialComputeColor into `ParticleCustomMaterial.cs` in our project and customize it to use two shaders for color binary trees.

```
        [DataMemberIgnore]
        protected override string EffectName { get; set; } = "ParticleCustomEffect";
```

The base class automatically tries to load the effect specified with `EffectName`. We give it the name of the effect we crated earlier.

```
        [DataMember(300)]
        [Display("Alpha")]
        public IComputeScalar ComputeScalar { get; set; } = new ComputeTextureScalar();

        [DataMember(400)]
        [Display("TexCoord1")]
        public UVBuilder UVBuilder1;
        private AttributeDescription texCoord1 = new AttributeDescription("TEXCOORD1");
```

In addition to the already existing IComputeColor, we'll use IComputeScalar for intensity, which returns a float, rather than a float4. We will also add another UVBuilder for a second texture coordinates animation.

```
    var shaderBaseColor = ComputeColor.GenerateShaderSource(shaderGeneratorContext, new
MaterialComputeColorKeys(ParticleCustomShaderKeys.EmissiveMap,
ParticleCustomShaderKeys.EmissiveValue, Color.White));
    shaderGeneratorContext.Parameters.Set(ParticleCustomShaderKeys.BaseColor,
shaderBaseColor);

    var shaderBaseScalar = ComputeScalar.GenerateShaderSource(shaderGeneratorContext, new
MaterialComputeColorKeys(ParticleCustomShaderKeys.IntensityMap,
ParticleCustomShaderKeys.IntensityValue, Color.White));
    shaderGeneratorContext.Parameters.Set(ParticleCustomShaderKeys.BaseIntensity,
shaderBaseScalar);
```

We load the two shaders: one for the main color and one for the intensity. These are similar to the shaders we wrote manually in the last two examples, except we generate them on the fly directly from the `ComputeColor` and `ComputeScalar` properties, which you can edit in the Property Grid. The generated code is similar to the shader code we wrote in the way that it calls `Compute()` and it returns the final result of our color or scalar compute tree.

After we generate the shader code, we set it to the respective key we need. Check how `ParticleCustomShaderKeys.BaseColor` is defined in `ParticleCustomShaderKeys.cs`. In the effect file we check if this key is set, and if yes, we pass it to the stream defined in our shader code.

See also

- Tutorial: Create a trail
- Tutorial: Custom particles
- Tutorial: Inheritance
- Tutorial: Lasers and lightning
- Particles
- Create particles

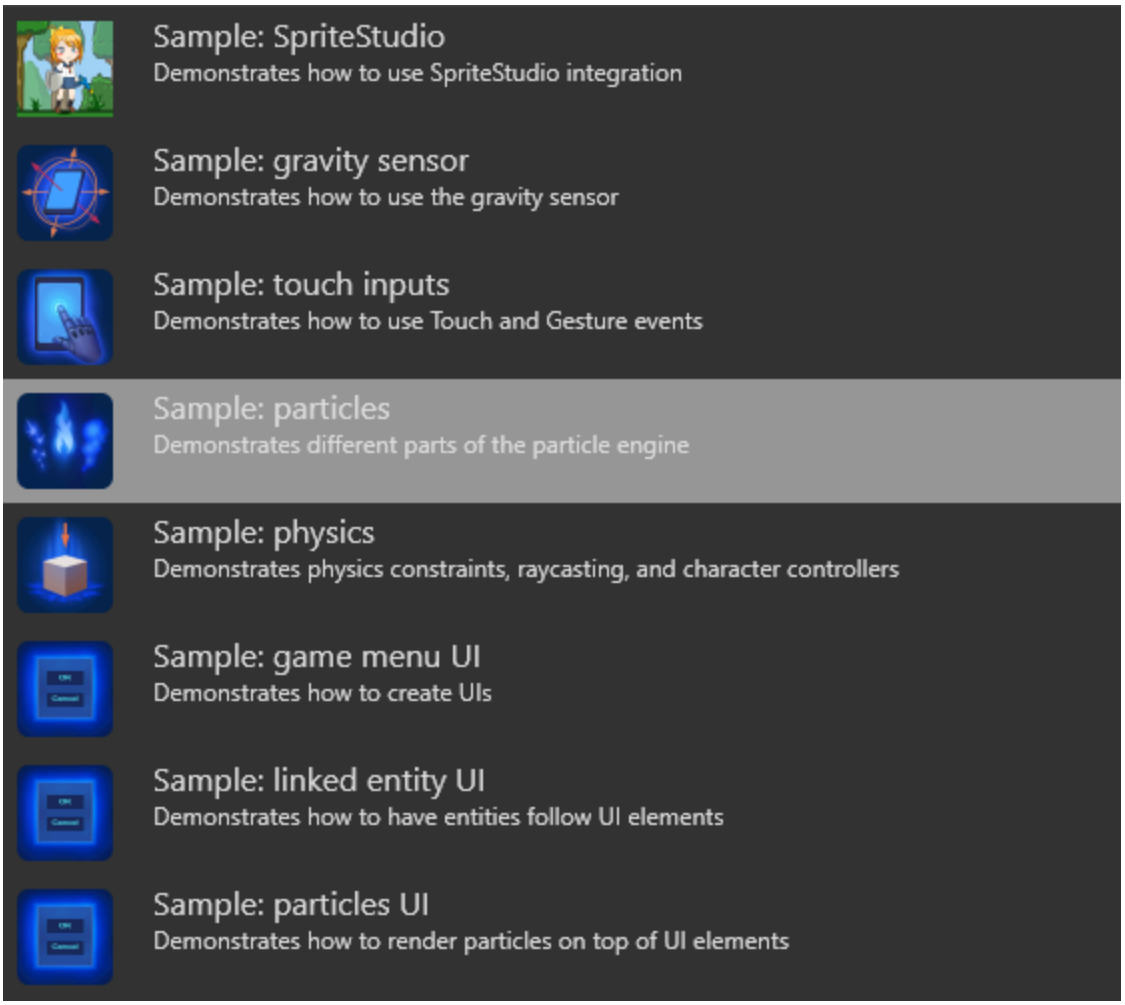
Tutorial: Inheritance

Intermediate Artist Programmer

This tutorial explains how to create particles which inherit one or more attributes, such as position or color, from other particles.

Sample

To see some of the techniques described on this page implemented in a project, create a new **Sample: Particles** project and open the **ChildParticles** scene.



Inheriting position

It helps if you think about inheritance in terms of parent and child particles.

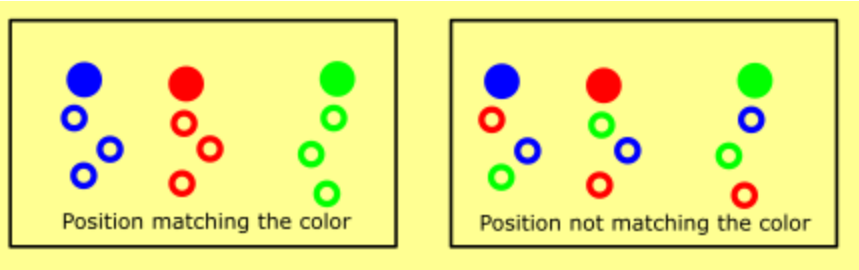
For example, in the **ChildParticles** scene in the **Sample: Particles** project, check out the **Fireworks** entity.

It contains two emitters. Particles reference parent emitters by name, so in the first emitter you can see we've set the **Emitter Name** property. It's an optional name, but it's required if you want other emitters

to be able to reference this emitter's particles.

In the second emitter we create a new initializer, **Position from parent**. This lets us reference the first emitter's particles and use their position to initialize the child particles. In the **Parent emitter** attribute we put the first emitter's name (*Parent*). This randomly assigns a parent particle for each child particle spawned and copy its position to the child particle.

The **Parent Offset** seed matches fields when more than one attributes are inherited. For example, if you want to inherit both **Position** and **Color** from the same parent particle (chosen at random) you should make the *Parent Offset* seed the same. Alternatively, you can make the *Parent Offset* seed for both initializers different, in which case particles spawning from one parent's position can inherit their color from a different random particle. Usually, you want to keep them the same, but in some cases you might want to mix them.



As you can see, this kind of inheritance doesn't control spawn count, maximum particles, or any other parameters, and is very random. For most effects it's sufficient, but sometimes you want more direct control over the particles.

Controlled inheritance

On occasion you will want to spawn a certain number of particles from a specific parent and have those particles only inherit attributes from the parent particle that spawned them.

To do this, choose a spawner for the child emitter from type **From parent**. Fill in the parent emitter's name in the **Parent emitter** field.

The **Spawn Control Group** determines how the particles save their control information. You need to assign the same control group on all initializers later in order to retrieve the spawning information.

There can be up to 4 control groups. If you spawn particles based on different conditions, or spawn more than two different child particles from the same parent, assign them different control groups so they don't get mixed up.

The **Particle Spawn Trigger** is the triggering condition on the parent side, which determines if particles should be spawned. If you leave it as **None**, no particles are spawned, so set it to **On Hit** or **Lifetime**.

On hit works for parent particles with a collider assigned, and triggers every time they hit the surface.

Lifetime is based on the parent particle's relative lifetime, and triggers every frame the lifetime is within the limits. There are two sliders to control from which point to which point particles should be spawned. Alternatively, you can reverse them to reverse the spawning condition. For example, a particle with lifetime condition (0.9 - 1.0) only spawns child particles in the last 10% of its lifetime.

Finally, the **Particles/trigger** determines how many particles are spawned each time the condition is met.

For child emitters, it's good practice to control the maximum number of particles the emitter can have, especially for non-deterministic cases, such as the collision hit.

Determinism

On the initializers, choose a **Spawn Control Group** corresponding to the spawner's control group. This forces the initializers to only work for particles spawned with the triggering condition, skipping the rest (if more than one spawner is assigned).

Ribbons and trails

Ribbon and trail renderers are a little more difficult to set up in the beginning, as they are dependent on spawn order. In case of parents, they also become dependent on the parent's spawn order.

1. Add a **Spawn Order** initializer to the parent. It will be used in the children particles.

2. On the child emitter, remove all spawners and add only one, **From parent**. You want to control the spawning of the children particles so all particles can be properly grouped in a ribbon behind the parent particle. If you add another spawner that adds random behavior to the system, the ribbons will connect in the wrong way. Set the triggering condition to **Lifetime**.

3. On the **child emitter** side again, add an **Order from parent** initializer. This assigns a spawning order to the particles, but also groups them by parent. If you set the **Sort** to use this order and assign a ribbon shape builder, you'll see how each trail is properly grouped behind the parent particle that spawned it.

Circular behavior

Particle emitters can inherit attributes circularly from each other, or even inherit attributes from particles in the same emitter. This can produce random or "swingy" effects, but can be interesting.

In the **Colliding Particles** particle entity (in the **MainScene** of the **Sample: Particles** project), you can see that particles are spawned on hit, but the parent emitter is the same. This means that each time a particle hits the surface, it produces more of the same kind. There are two important elements which allow this to happen.

First, we have two spawners. One spawns a small number of particles per second, which give us the initial elements to populate the system. The other spawner spawns more particles on hit and uses a control group.

Second, we have two **Position** initializers. The first assigns a position where we want the particles to appear. It works over all particles (even those spawned from parents), so if you leave it like this, it will fire more particles from the initial position every time they hit the surface.

The second initializer is **Position from parent** and initializes the particle positions using the same control group as the **On hit** spawner. The **Position from parent** overwrites the positions for the particles with control group, leaving the particles spawned from the **Per second** spawner untouched. This creates a small number of particles constantly coming from a single entry point and multiplying like an avalanche every time they hit the surface.

See also

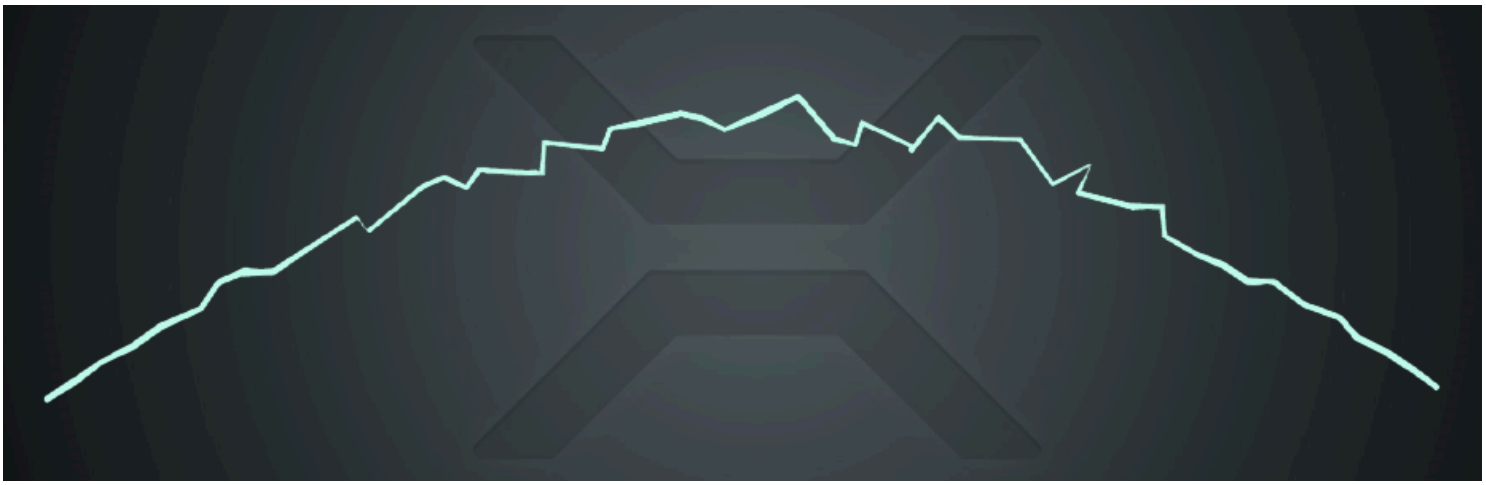
- Tutorial: Create a trail
- Tutorial: Custom particles
- Tutorial: Lasers and lightning
- Particles
- Create particles

Tutorial: Lasers and lightning

Intermediate Artist Programmer

This tutorial explains how to create lasers and lightnings using particles and custom materials.

Imagine we want to create a lightning arc like this one:



This effect is a strip which:

- connects two fixed points
- changes positions very quickly
- can be rendered as a single strip

Because the lightning is a single-line strip, we can render it using the ribbon shape builder, but with a few major differences. The particles:

- spawn at the same time, rather than in sequence
- appear on a single line or arc, but with semi-randomized positions to give the illusion of lightning
- should reappear very quickly

Simultaneous spawning

We can create a looping Spawner by frame which spawns a certain number of particles (lets say 50) **every** frame.

Because we only need one set visible at a time we limit the Maxmimum Particles on the emitter to 50 and give them the same lifespan (for example 0.2 seconds).

This means the Spawner will try to emit 50 particles every frame, but because we have limited them it will only spawn 50 particles the first frame.

They all have the same lifespan, so when they die at the same time a new batch of 50 particles will be spawned.

Connect two points

We are going to use the **Position (Arc)** initializer. It picks a second point from another Entity and sets the particles' positions to lie on an arc between the Emitter and the target Entity.

By clicking the Ordered checkbox we can force the particles to be placed at equal distances starting from the emitter and moving towards the target Entity. This is important when we render them using a Ribbon shape builder because if they appear at random (unordered) positions along the arc it will be a mess. We also have to add Spawn Order initializer and sort the particles by Order (this is true for all ribbons, not just lightning.)

The arc position initialzier also allows for a random offset which we set to some small number.

Change positions fast

We can set the particles' lifespan to a small number (eg 0.2 seconds). With the Time scale parameter, we can additionally control the speed of the entire particle system.

To illustrate better what's going on here is the same effect with Billboard shape builder instead of Ribbon, and slowed down 30 times:

Moving lightning

There is a way to make the lightning arc move from point A to point B instead of being static.

There are a few adjustments we need to make:

- Change the spawn rate to a lower one. The example above uses 600/second and is played at 0.1 time scale, which means around 1 particle per frame.
- Set a fixed count on the arc positioner (50). Because it interpolates the distances based on the number of particles spawned *each* frame, if we spawn them sequentially they'll all stay in the beginning of the arc. By setting the count to 50 we tell the arc positioner to expect 50 particles in total.
- Set a delay to the spawner to allow the old arc to completely disappear before starting again. Otherwise the Ribbon will wrongly connect the old and the new particles, as it can't know how to split them.

Lasers using particles

Creating lasers with particles is very similar to making lightning. We actually need less particles, because the lasers are straight and do not deviate. By setting the arc positioner's arc height to 0 and random offset to (0, 0, 0) we can spawn the particles in a straight line. If you want you can give them slightly different sizes to make the laser beam appear shimmering.

One thing to be mindful about lasers is that usually when the target moves you want the laser to move with it. Because the arc positioner is an initializer and not an updater, it has no effect on particles already spawned, which and stay behind. There are three ways to counter this.

- Spawn the particles very fast. If they only live for 1-2 frames the laser will be recreated too fast for the user to notice any visual differences.
- Spawn particles in Local space. This means they will move together with the emitter, but then you will have to rotate and scale the emitter to always point to the target Entity.
- Create a custom Updater. If you create a custom post-updater similar (or simpler) to the arc positioner you can force it to update the particle positions every frame, correctly placing them

between the two points even if they move.

Depending on the type of game you want to make each of these options can have benefits or drawbacks. Spawning the particles every frame is the easiest and simplest way to do it and will be sufficient for most needs.

Lasers using custom materials

Creating lasers using custom materials is similar to using particles in Local space. We need to manually rotate the scale the emitter to always face a target entity.

We can designate one axis which points towards the target to be our length, leaving the other two axes for width of the laser.

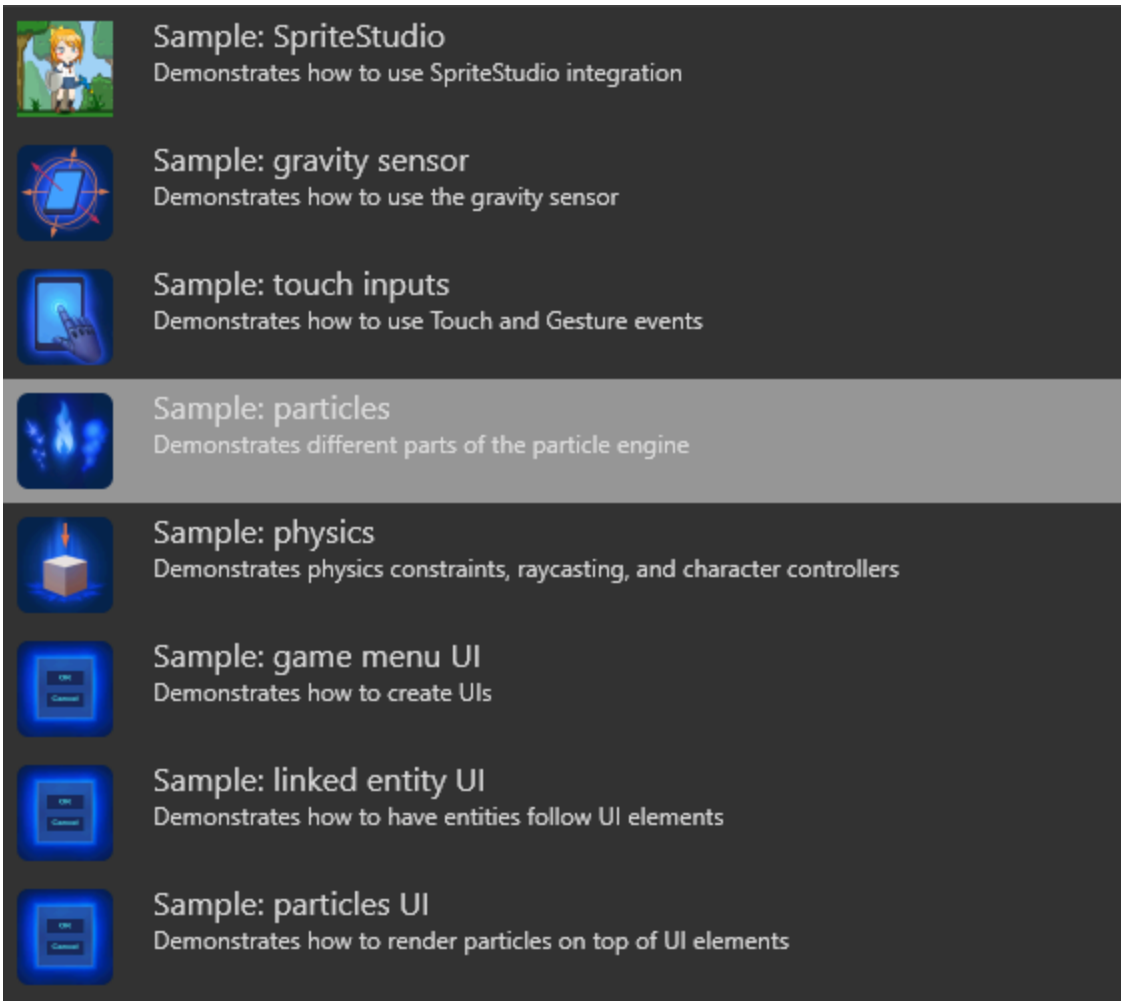
Rendering a cylinder with height of 1 which is placed under the rotated entity will cause it to stretch and reach the target point.

The custom material is required to place a scrolling texture on the cylinder. Or you can use a regular Emissive map with no scrolling in which case you won't need a custom material.

The particles sample already contains an example of how to create lasers this way. The LaserOrientationScript rotates and scales the entity towards a target point and the ComputeColorTextureScroll shader samples a scrolling texture.

Sample project

To see some of the techniques described on this page implemented in a project, create a new **Sample: Particles** project and open the **Lasers** scene.



See also

- Tutorial: Create a trail
- Tutorial: Custom particles
- Tutorial: Inheritance
- Particles
- Create particles

Tutorial: Create a trail

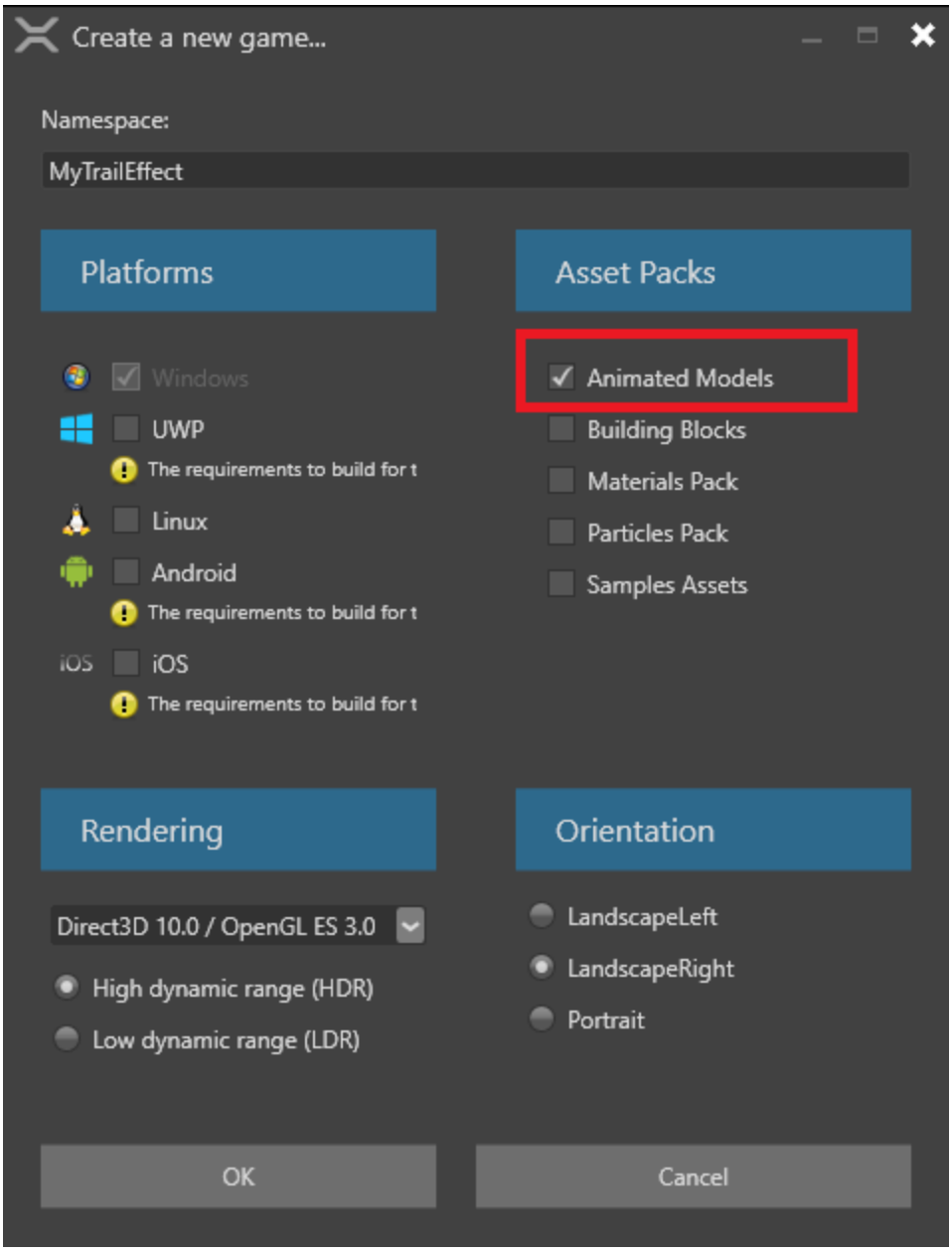
Intermediate Artist Programmer

This tutorial demonstrates how to use particles to create a trail effect for a sword swing.

0:00

1. Create a project

1. In the Stride Launcher, click **Start** and select **New Game**.
2. In the **Create a new game** dialog, under Asset Packs, select **Animated Models**. The Animated Models pack contains assets we'll be using in this example. (Note that we'll make our particle effect from scratch.)



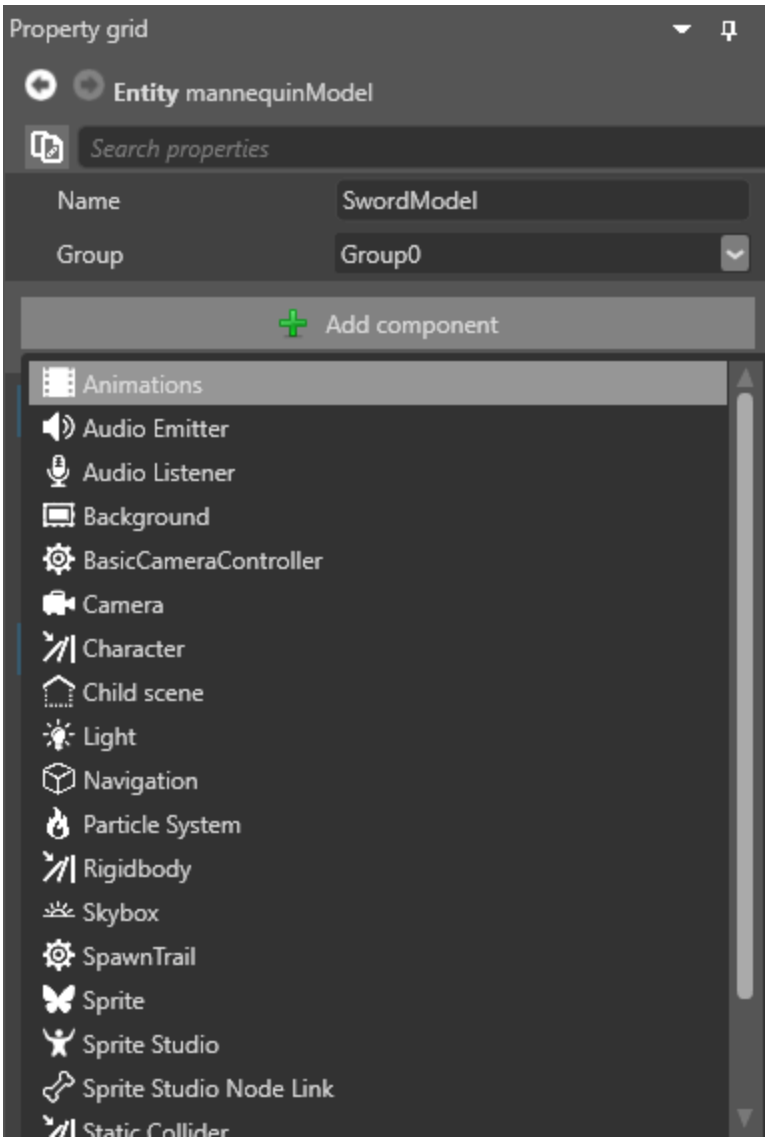
3. Give the project a name (eg *MyTrailEffect*) and click **OK**. Game Studio loads a simple scene with a few entities.

4. We don't need the **Sphere** entity for this tutorial, so go ahead and delete it (select it and press **Delete**).

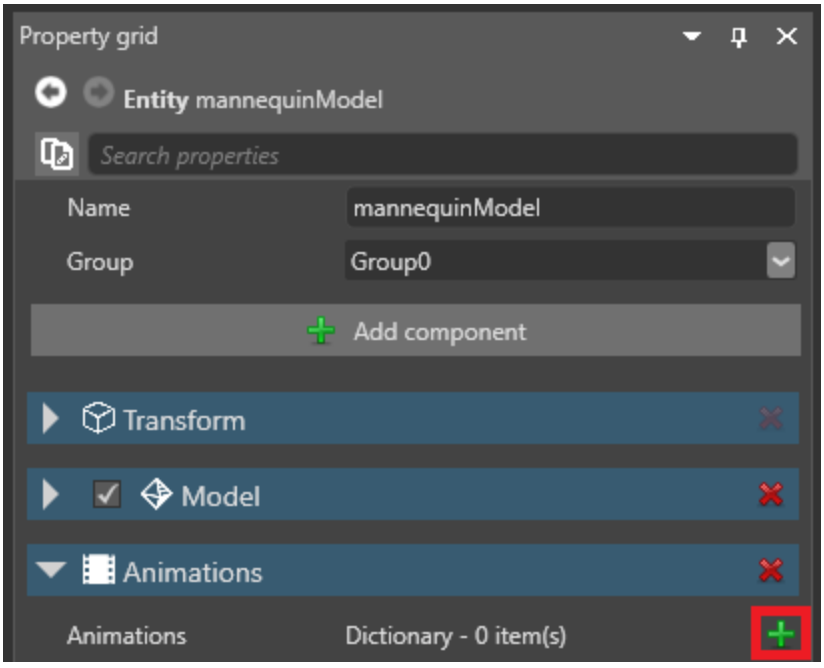
2. Set up the models and animation

1. In the **Asset View**, open the **Models** folder and drag and drop the **mannequinModel** into the scene. The mannequinModel contains a skeleton asset that we'll use for our sword slash animation.

2. With the **mannequinModel** selected, in the **Property Grid**, select **Add component > Animations**. This adds an Animation component to the model.

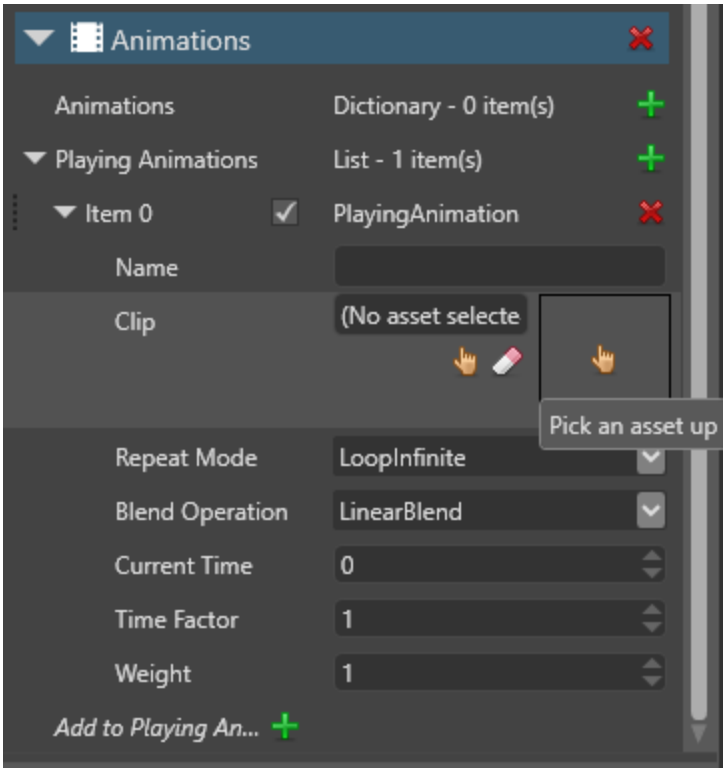


3. Under the **Animations** component, click (**Add**).

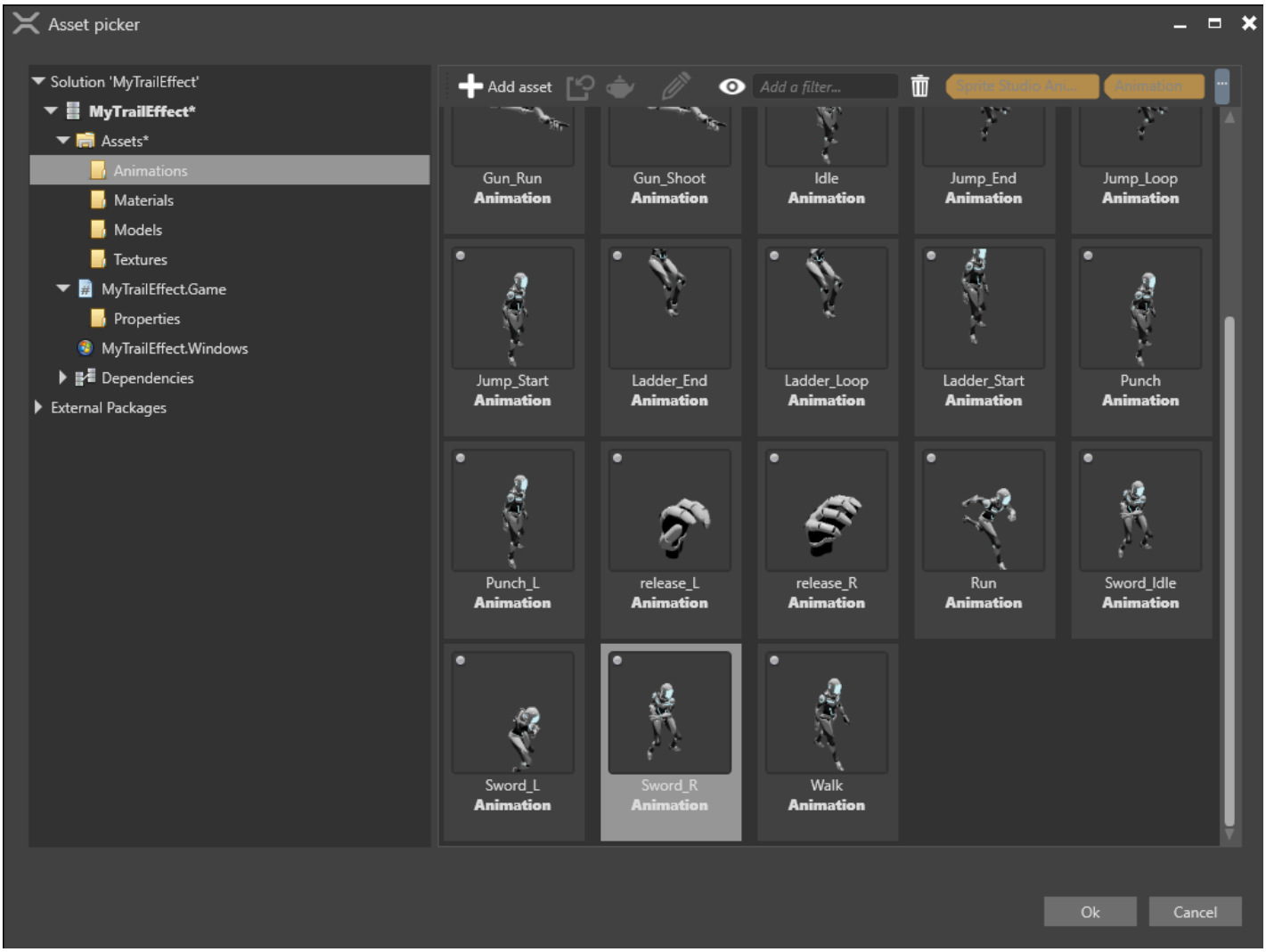


4. Type a name for the animation.

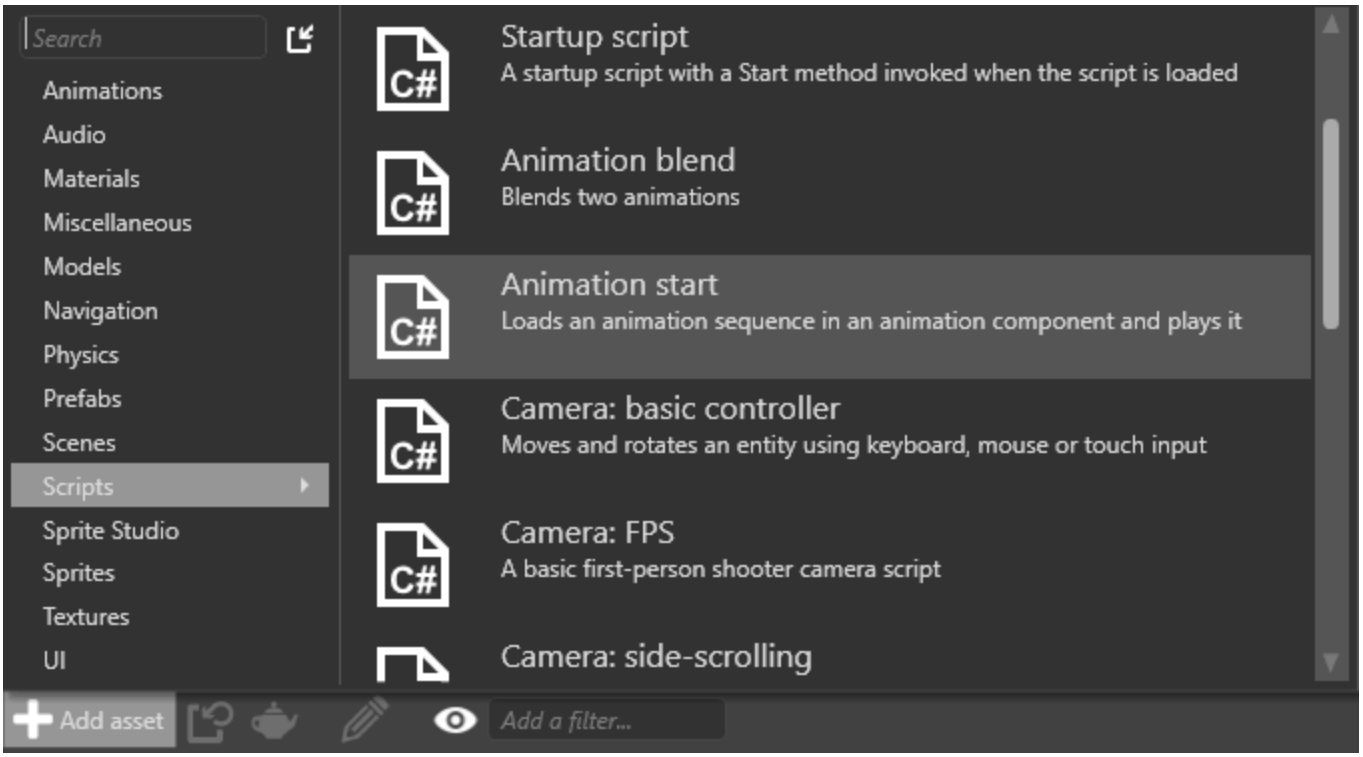
5. Next to **Clip**, click (**Select an asset**).



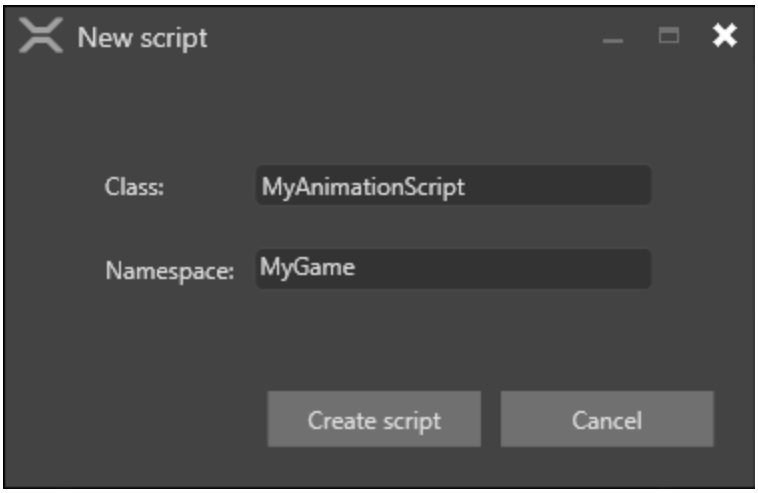
6. Browse to the **Animations** folder, select the **Sword_R animation**, and click **OK**. This is our right-to-left slash animation.



7. To play the animation at runtime, we need to add an animation script. We can use the pre-built **AnimationStart** script. In the **Asset View** (bottom pane by default), click **Add asset** and choose **Script > Animation start**.



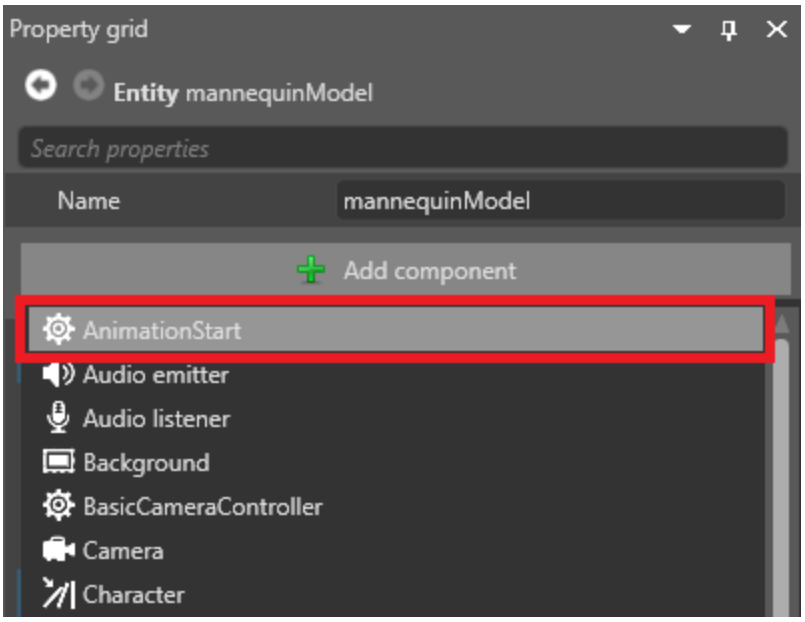
8. Specify a name for the script and click **Create script**.



9a. If Game Studio asks if you want to save your script, click **Save script**.

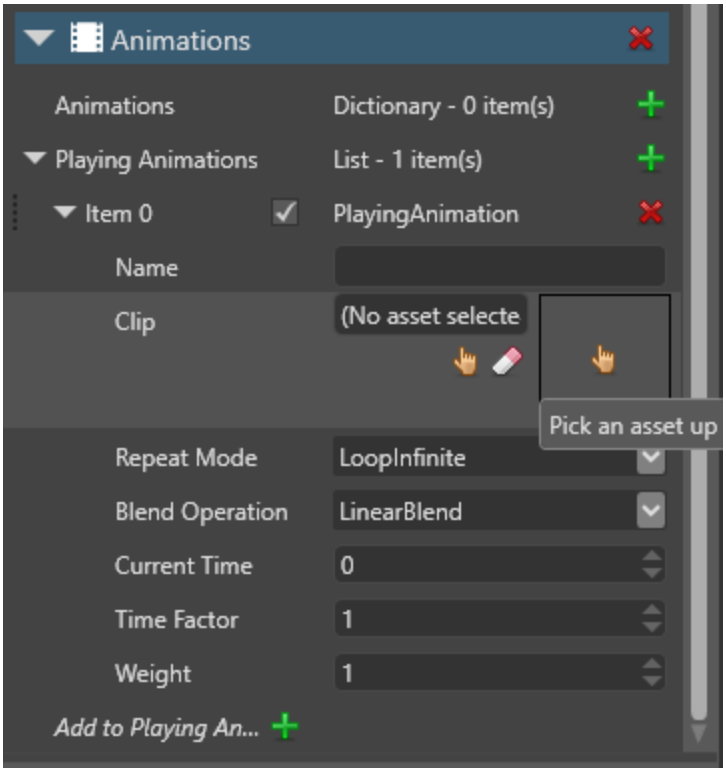
9b. If Game Studio asks if you want to reload the assemblies, click **Reload assemblies**.

9. With the **mannequinModel** selected, in the **Property Grid**, click **Add component** and select the **AnimationStart** script.

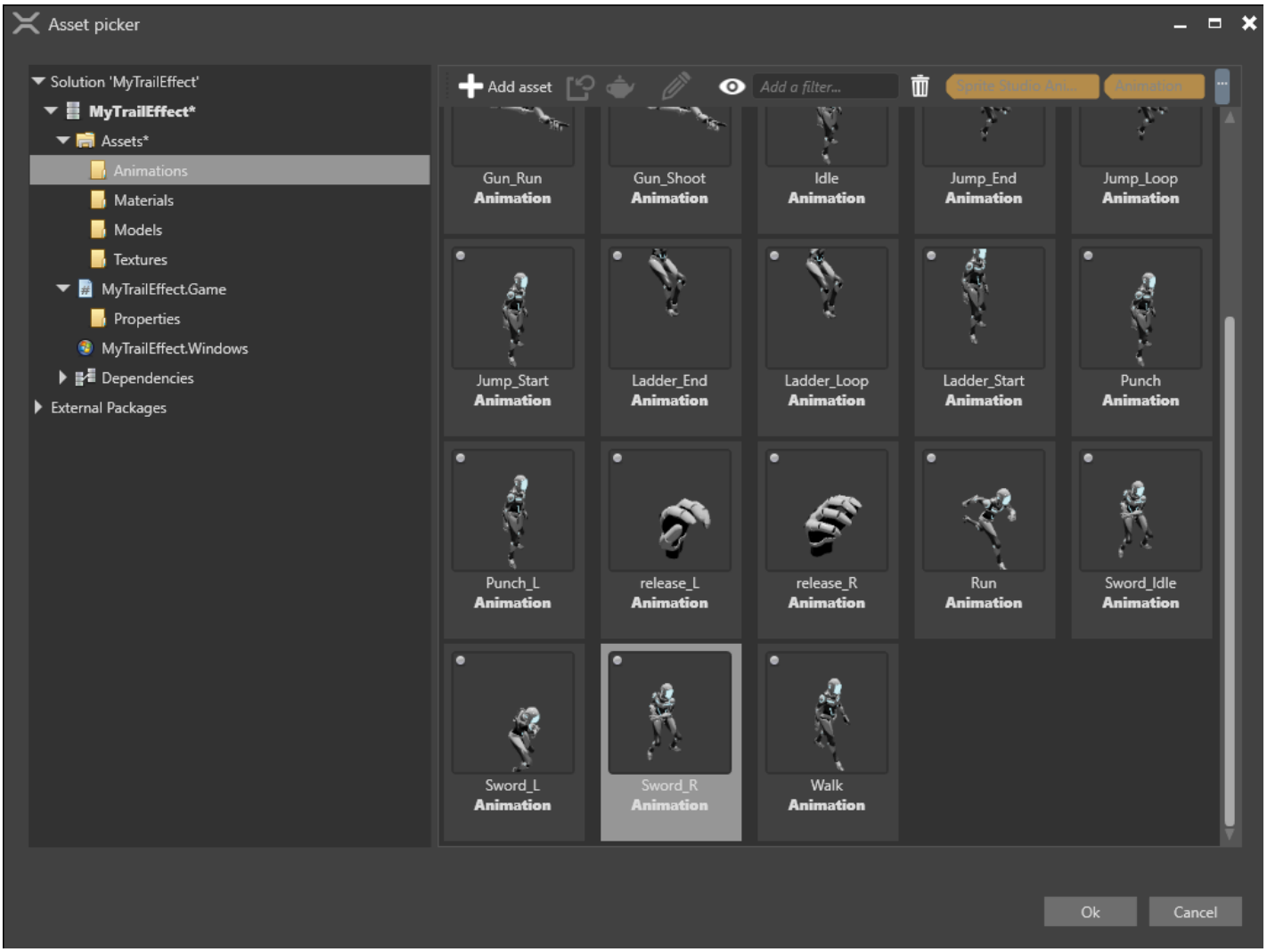


> **NOTE**
>
> If the animation script isn't in the list of components, in the taskbar, save your project and click **Reload game assemblies and update scripts**.

10. Under the **Animation** component, under **Item 0**, click (**Select an asset**).

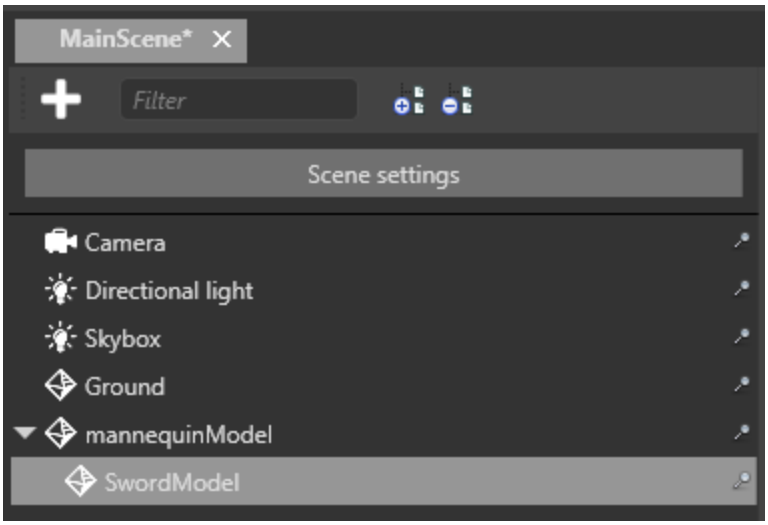


11. Browse to the **Animations** folder and select the **Sword_R animation** again.

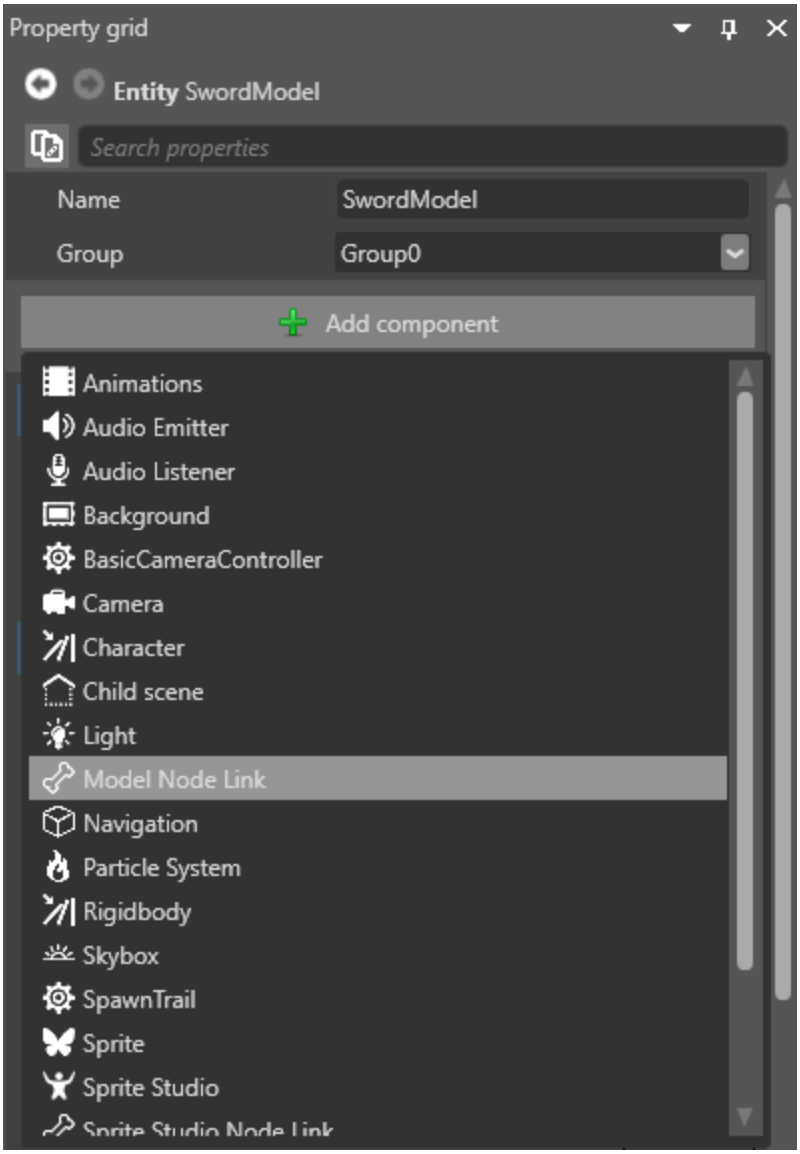


12. Now let's give the mannequin a weapon. In the **Asset View**, browse to the **Models** folder and drag the **SwordModel** to the mannequinModel in the Entity Tree. This makes the SwordModel a child entity of the **mannequinModel**.

13. In the Entity Tree, expand **mannequinModel** to see its child entities, and select **SwordModel**.



14. With the in the **Property Grid**, click **Add component** and select **Model Node Link**. This is called a **Bone Link** in some versions of Stride.



We can use this to link the SwordModel to a point in the mannequinModel skeleton. There's no need to specify a target, as the entity uses its parent entity (**mannequinModel**) by default.

For more information, see the Model node links page.

15. Under **Model Node Link**, in the **Node Name** (or **Bone**) field, select **weapon_bone_R**. This attaches the model to the point in the skeleton that uses a weapon in the right hand.

16. Let's see how everything looks so far. Click **Play** to run the game and check it out. Remember you can use the mouse and WASD keys to move the camera and see the animation from different perspectives.

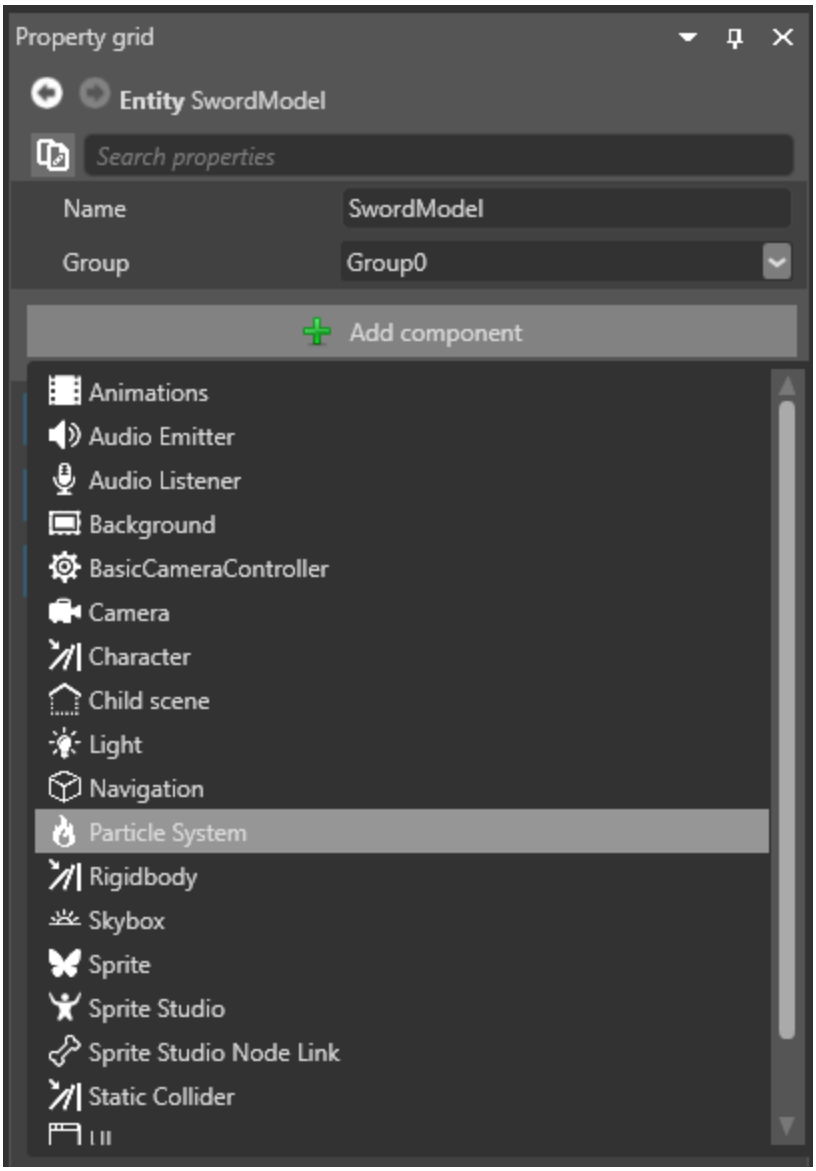
0:00

We have a swinging sword animation. Next, let's add a trail effect.

3. Create a basic trail

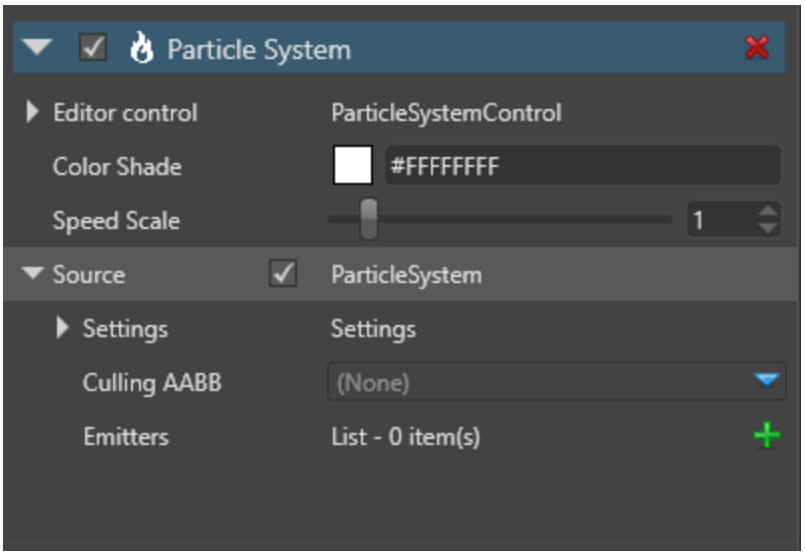
First we'll build a basic trail, just to see how it looks.

1. In Game Studio, select the **SwordModel**. In the **Property Grid**, click **Add component** and select **Particle System**.



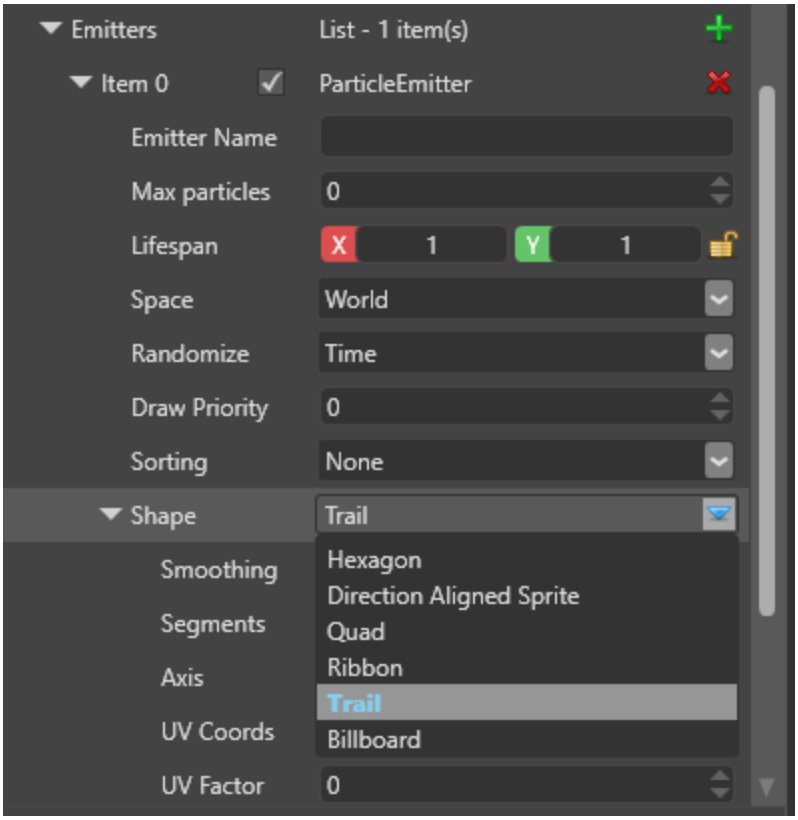
This adds a particle system component to the model, which we'll use to build a trail effect.

2. Click **Source** to expand its properties.

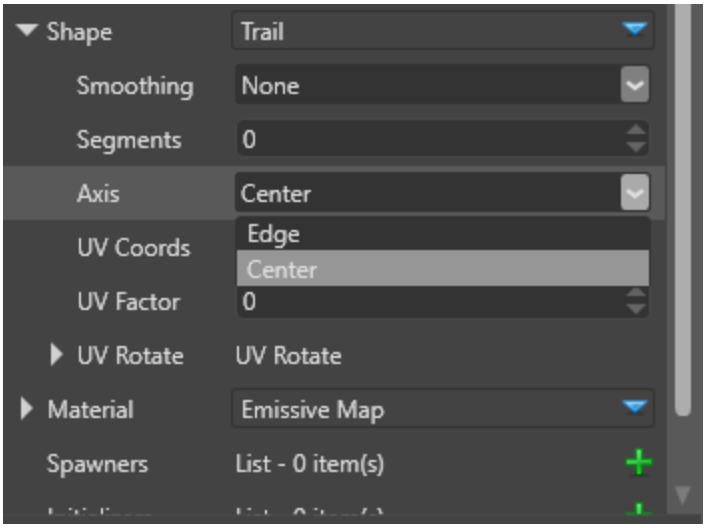


3. Next to **Emitters**, click (**Add**). This adds a new particle emitter.

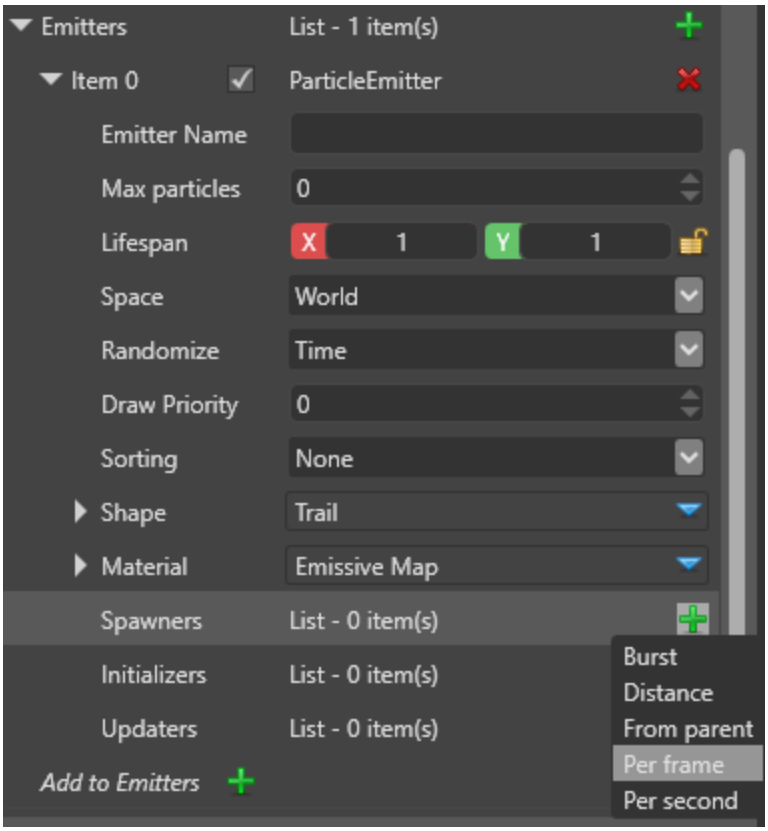
4. Under the emitter properties, set the **Shape** to **Trail**.



5. Unfortunately, we need to make a brief detour due to a bug in Stride. Under the **Shape** properties, set the **Axis** to **Center**. (The shape should really be set to Edge, but the Edge and Center settings are reversed in the UI. This will be fixed in Stride 1.9.3.)

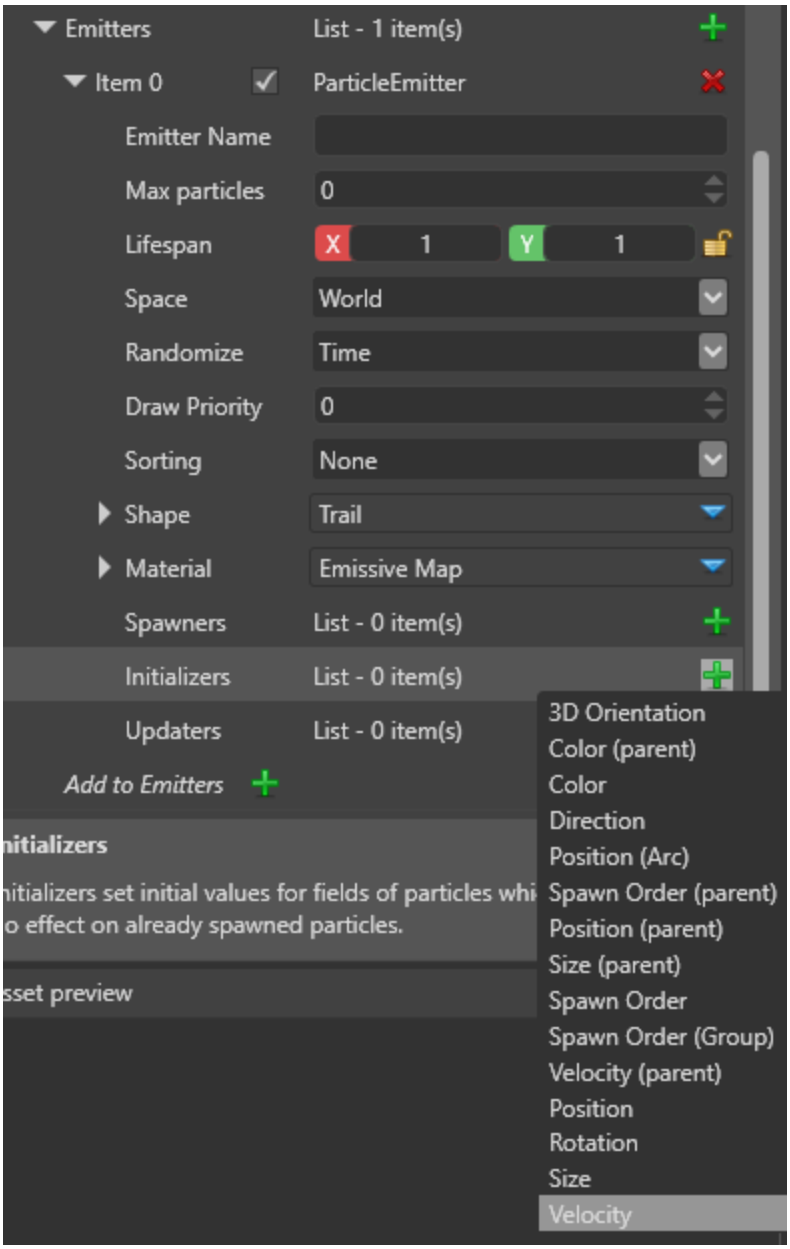


6. Next to **Spawners**, click (**Add**) and select **Per frame**.



This adds a per-frame spawner to the emitter, which spawns X number of particles per frame (as opposed to, say, per second).

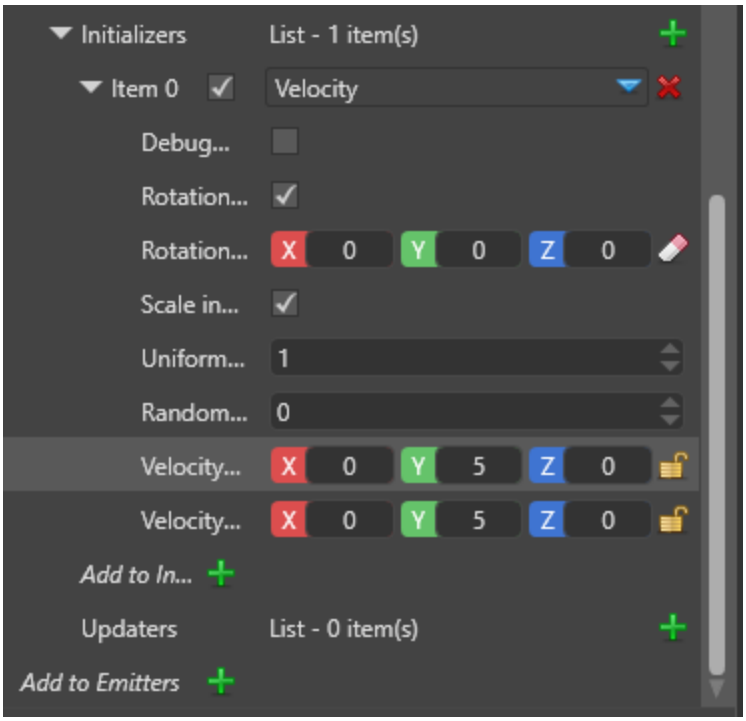
7. Next to **Initializers**, click (**Add**) and select **Velocity**.



This adds a velocity initializer to the emitter.

At this point, you can grab the mannequin and move it around the scene to see how the particles behave. They look like a cloud of blocky smoke.

8. Under the velocity initializer, set both the **Velocity min** and **Velocity max** values to **0, 5, 0**.



This restricts the particles to the Y axis, like an infinitely thin sheet of paper.

9. Next to **Initializers**, click (**Add**) and select **Direction**.

This adds a direction initializer to the emitter.

10. Expand the direction initializer to view the properties. Set both the **Direction min** and **Direction max** to **0, 0, -1**. This aligns the trail with the direction of the swing animation.

11. Run the game to see how the particles look with the sword-swinging animation.

0:00

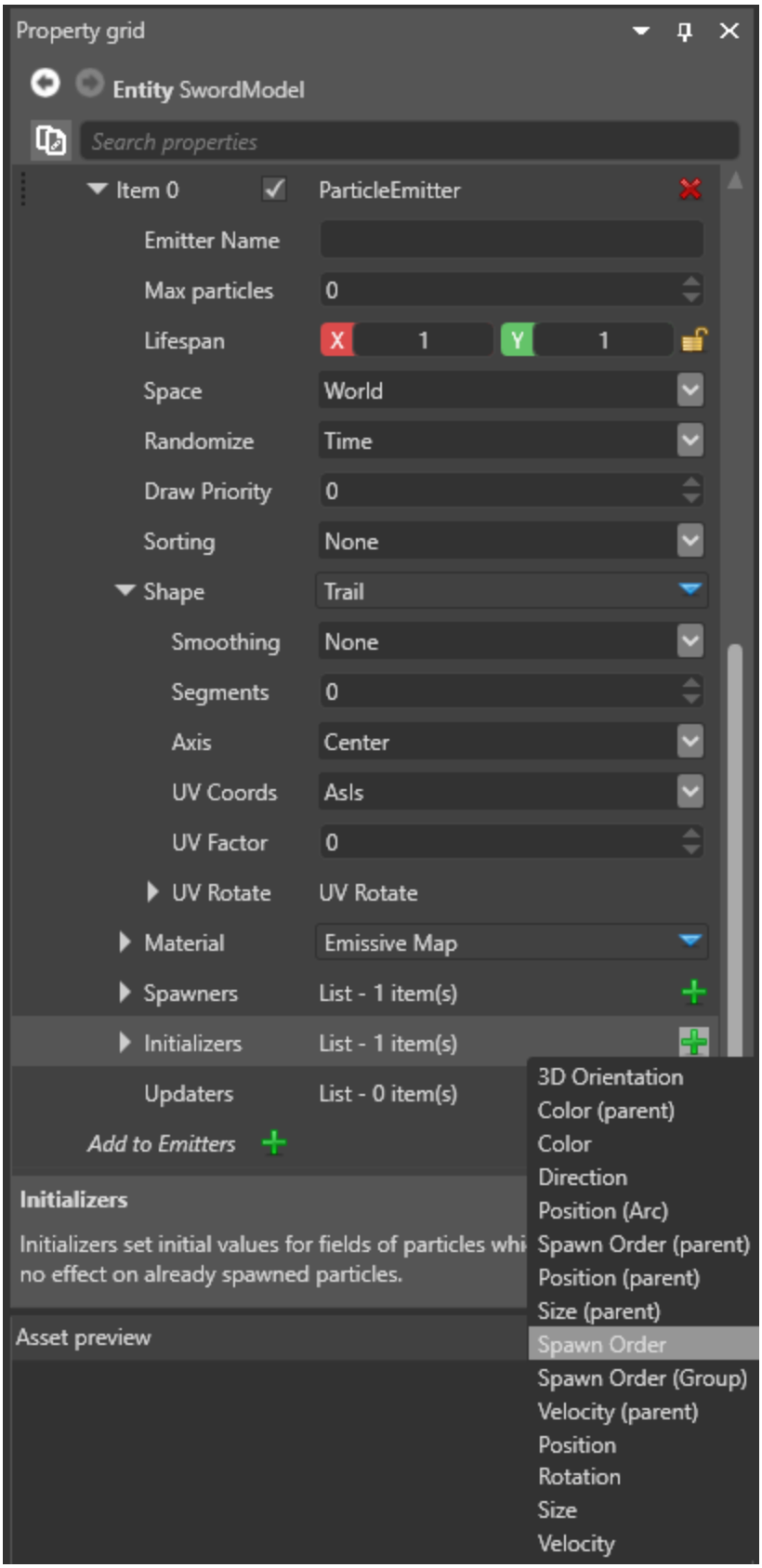
We have a trail, but it doesn't look too good yet. It's too long, it's a single block of color, its particles interconnect strangely, and it never disappears.

4. Sort the particles

Because the particles are rendered as billboards, the segments of the trail interconnect strangely. To create a proper trail effect, we need to sort the particles into an order by adding a **spawn order initializer**.

1. In the SwordModel properties, under **Particle System > Source > Emitters**, next to **Initializers**, click ➕ (**Add**) and select **Spawn Order**.

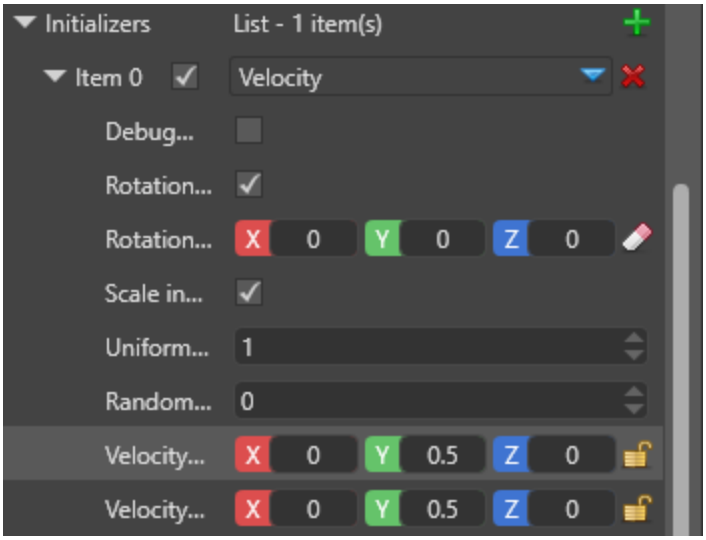
> **NOTE**
> Make sure you don't select **Spawn Order (Parent)** or **Spawn Order (Group)**.



This adds a spawn order initializer to the emitter. It doesn't have any properties, but it gives the particles a SpawnID we can sort them by.

2. Under **Emitters**, under **Sorting**, choose **ByOrder**.

3. Under **Initalizers**, under the **Velocity** initializer, change both the **Velocity min** and **Velocity max** values to **0,0.5,0**.



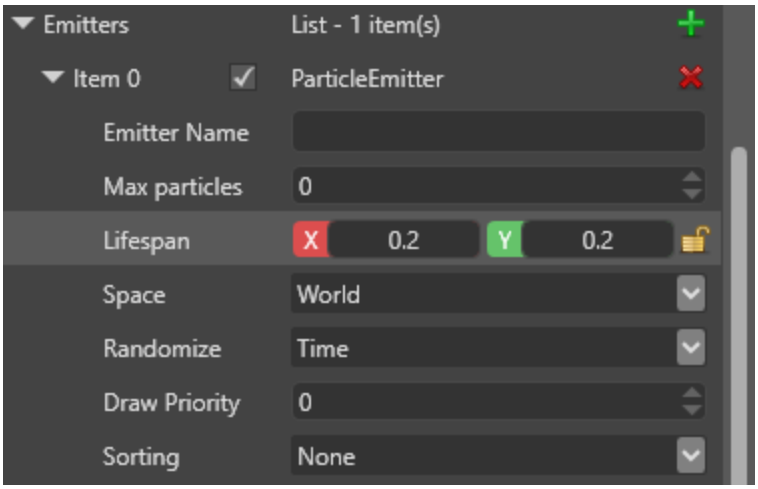
4. Run the game.

0:00

Now the particles move cohesively.

5. Change the length

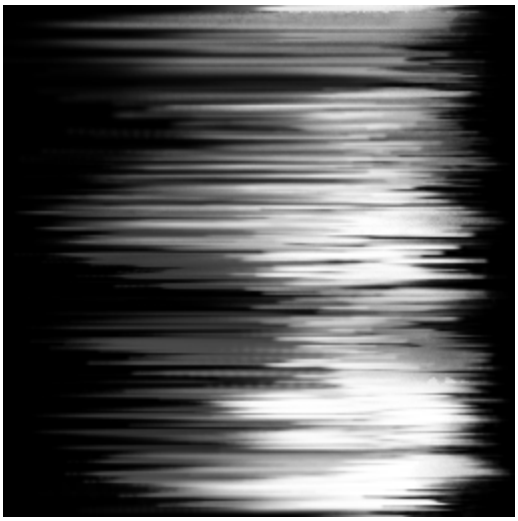
In the SwordModel properties, under **Particle System > Source > Emitters**, change the **Lifespan** to **0.2, 0.2**.



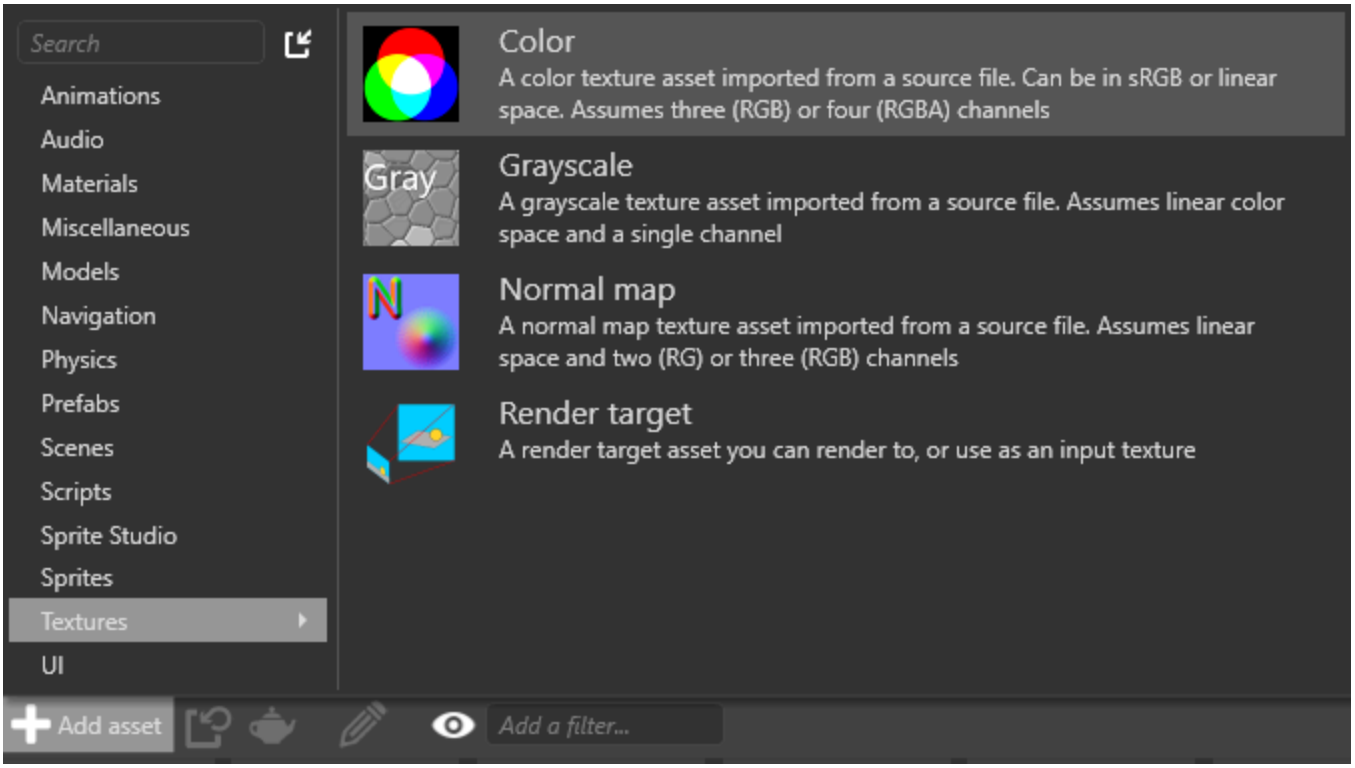
Move the mannequin around the scene and notice how the trails extinguish more quickly.

6. Add a texture

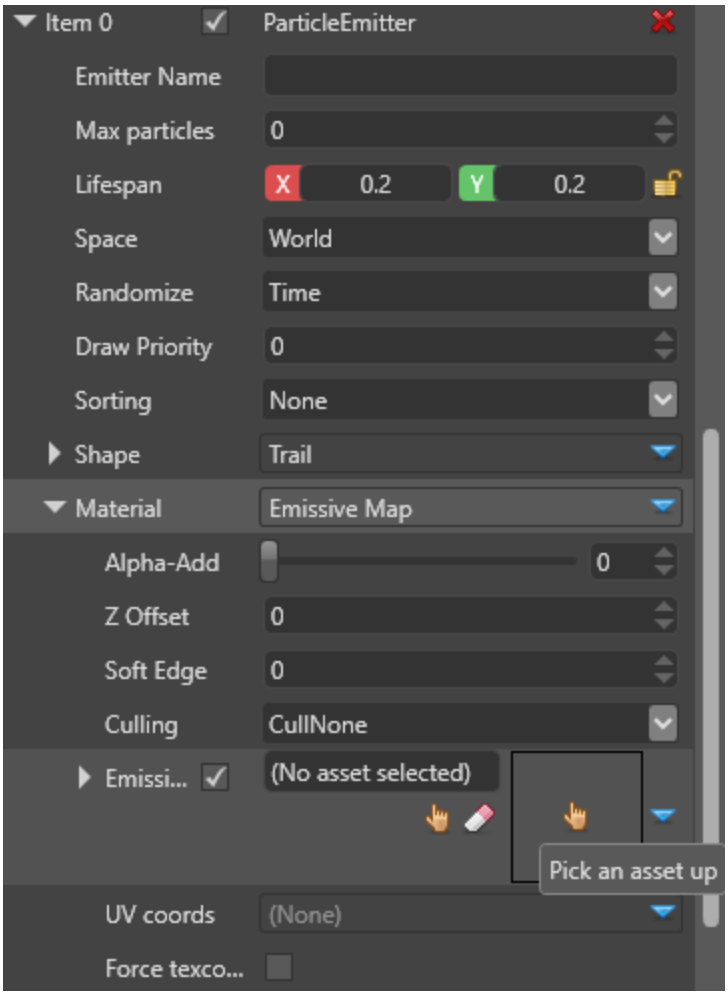
To fix the color, we'll give the particles a "swoosh" texture:



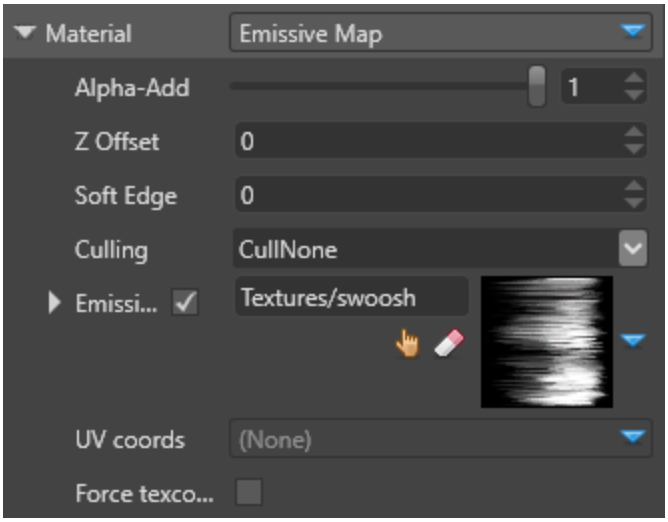
1. Save the texture image above (**swoosh.png**) to disk.
2. Import it into the project. To do this, in the **Asset View**, click **Add asset > Textures > Color** and select **swoosh.png**.



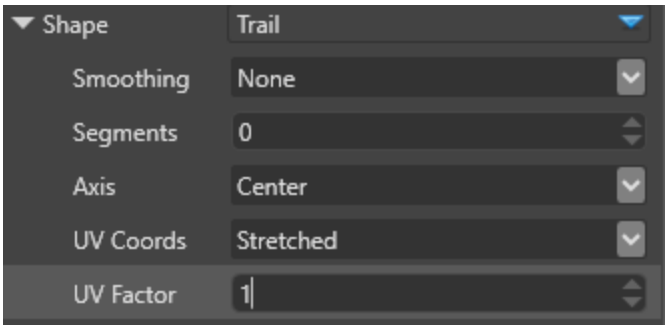
3. In the **SwordModel** properties, expand **Emitters > Material**. Click (**Select an asset**). Browse to the **Textures** folder and select **swoosh.png**.



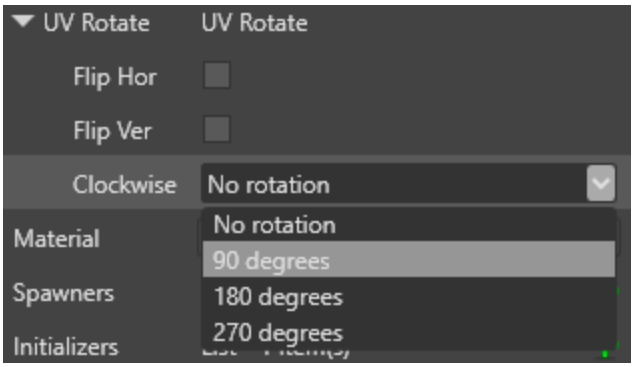
4. Set the **Alpha-Add** bar to **1**, so it's 100% emissive.



5. Under the **Particle emitter** properties, expand **Shape** and set **UV Coords** to **Stretched** and **UV Factor** to **1**.



6. Expand **UV Rotate**. Under **Clockwise**, select 90 degrees. This rotates the texture 90 degrees clockwise, so the "swoosh" lines point in the right direction.



7. Run the game.

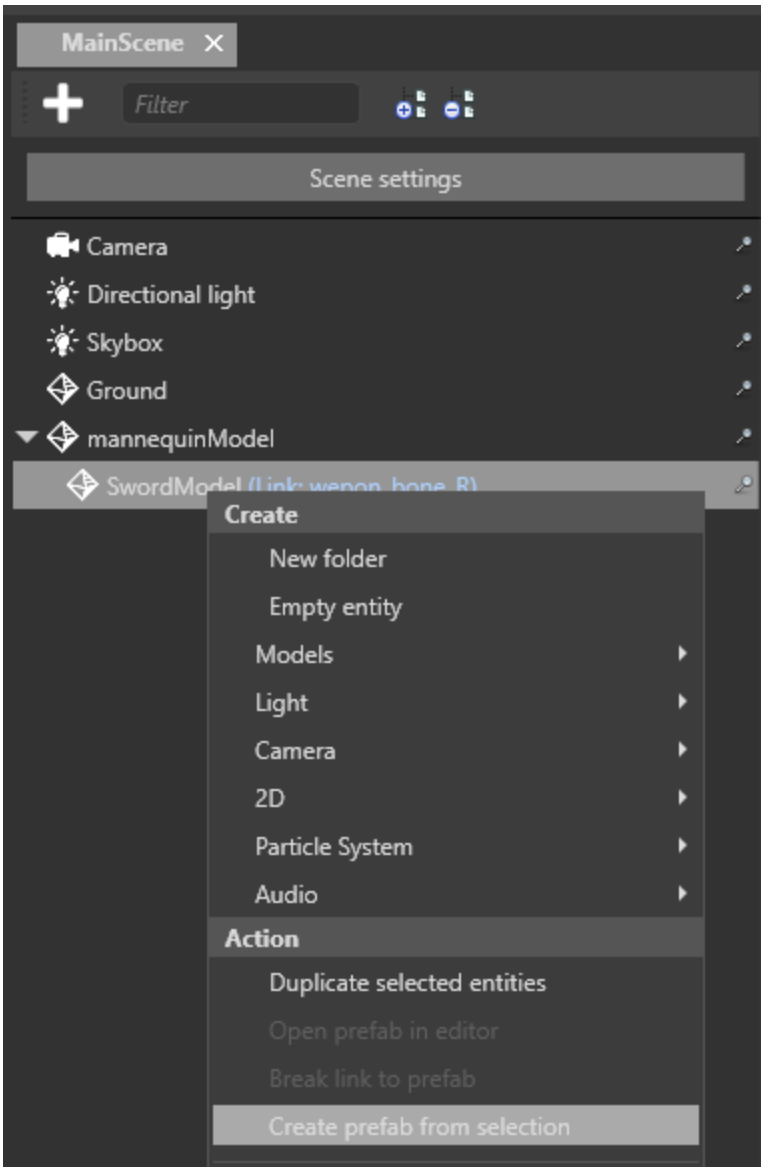
0:00

We're getting closer. But the trail doesn't disappear, so it looks like it's attached to the sword. We need to make the effect appear when the mannequin swings, then disappear at the end of the swing.

7. Make the particle effect a prefab

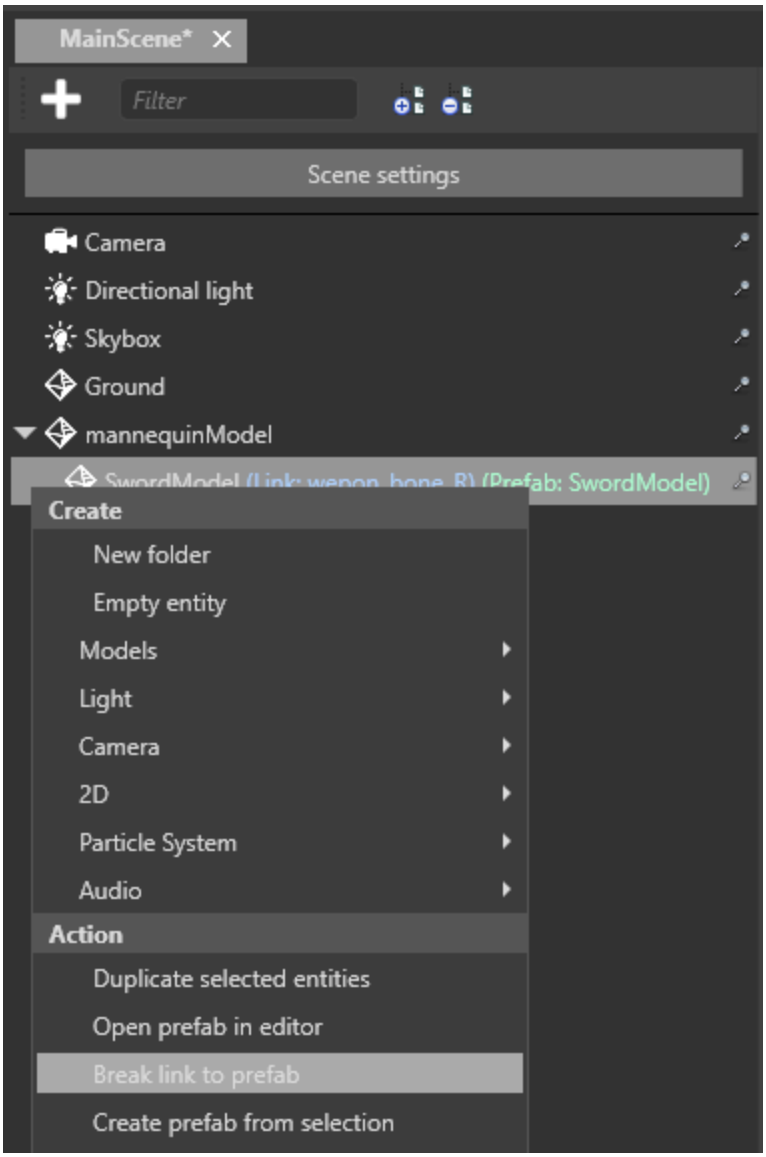
So far, we've created a particle effect by attaching it as a component to the sword. Now we're going to separate the effect from the sword and make it an independent entity we can turn on and off when we like. To do this, we'll create a prefab. For more information about prefabs, see the prefab documentation.

1. Right-click the **SwordModel** and select **Create prefab from selection**.

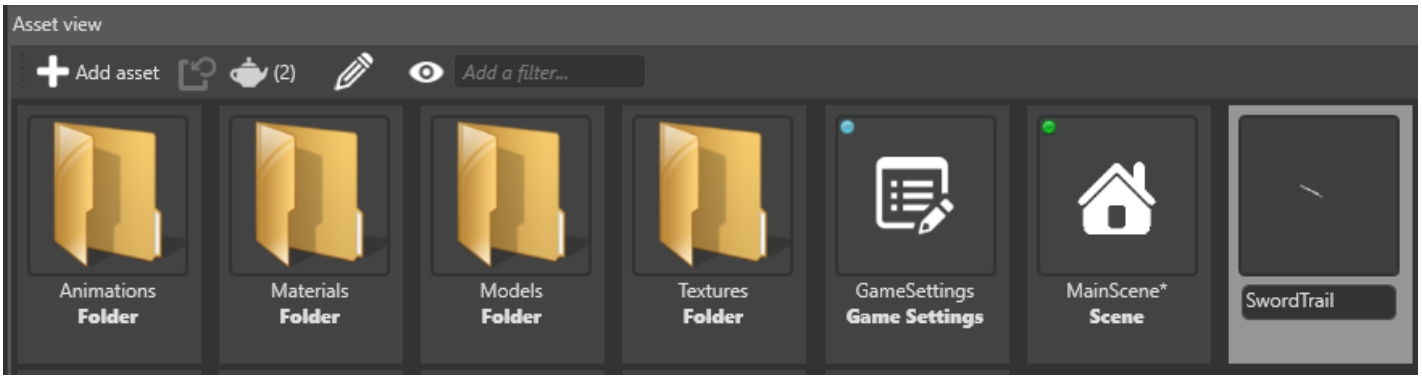


   Game Studio creates a prefab from the SwordModel and adds it to the Asset View. By creating a prefab from the selection, we can quickly copy over the options we've set up so far.

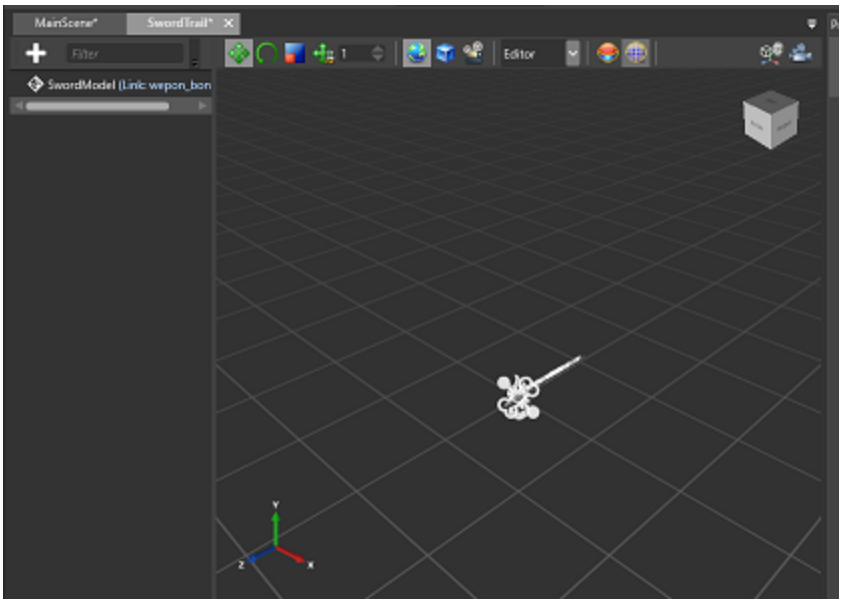
2. We don't want the SwordModel itself to be a prefab — we just used it as a template to create the prefab from. It should be separate from our new particle effect prefab, so right-click it and select **Break link to prefab**.



3. Because naming things properly makes everything easier, rename the prefab *SwordTrail*. To do this, in the **Asset View**, right-click the **SwordModel** prefab, select **Rename**, and type *SwordTrail*.



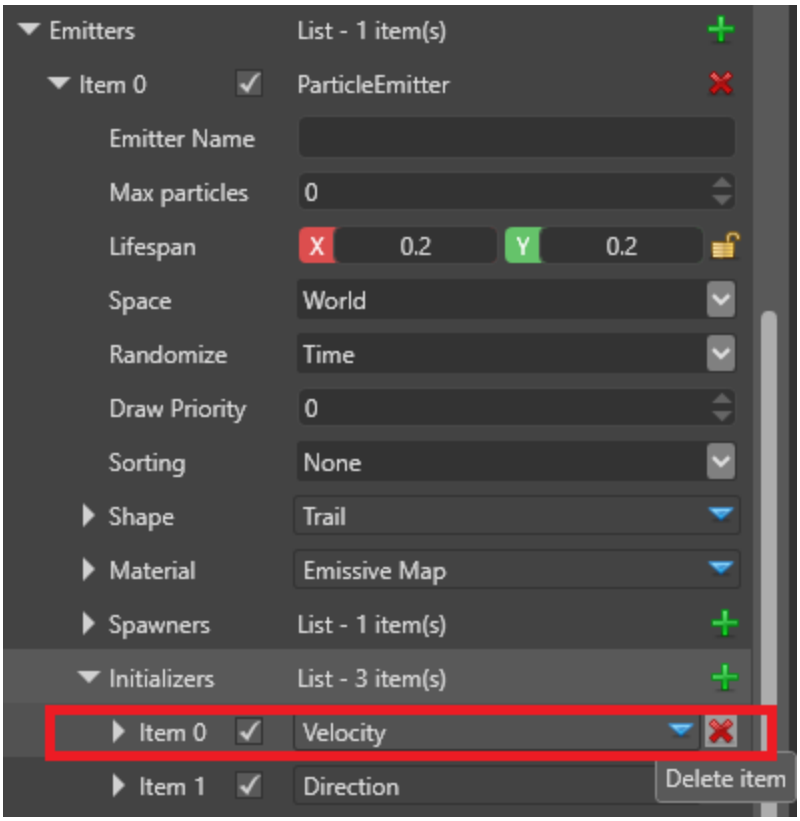
4. Double-click the **SwordTrail** prefab to open it in the Prefab Editor. This is where we'll customize the prefab.



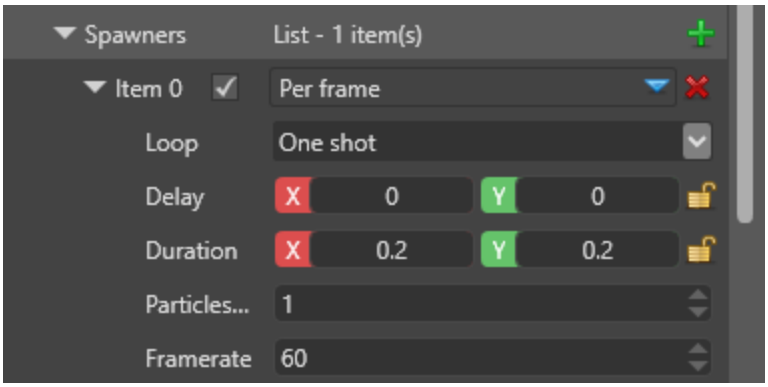
5. The prefab contains just one entity, SwordModel. It's not going to be a model for much longer, so let's rename this entity *SwordTrail* (the same as the prefab it belongs to).

6. Remove the **Model** and the **Model Node Link** (or **Bone Link**) components from the **SwordTrail** entity. We don't need them any more — this prefab will just be a particle effect.

7. Likewise, under **Particle System > Source > Emitters > Initializers**, delete the **Velocity** initializer. For now, we want the prefab effect to be static.



8. In the **SwordTrail** properties, under **Particle System > Source > Emitters > Spawners**, set **Loop** to **One shot** and change **Duration** to **0.2, 0.2**.

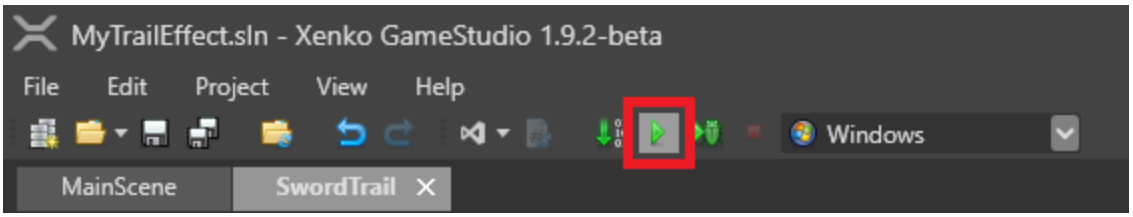


9. Now we've created a separate prefab for the particle effect, we don't need to keep a particle effect on the sword model. In the main scene, select **SwordModel** and delete the **Particle System** component.

8. Control the effect prefab with a script

We've created a sword trail effect prefab. Next we'll use a script to spawn the effect every time the mannequin swings and delete the effect a few frames later.

1. Open the project in Visual Studio. To do this, in Game Studio, click the Visual Studio icon (**Open in IDE**).



2. In Visual Studio, right-click the game project and select **Add > New item**. In the **Name** field, give your script the name *SpawnTrail*, and click **Add**.

3. Replace the script content with the code in this script: SpawnTrail.cs

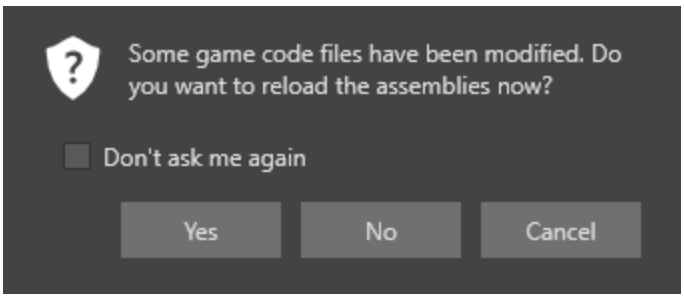
   This is a modified version of the Prefab Instance script included in Stride. Instead of listening to events or key presses, it listens to animation changes — such as our sword swing animation.

4. In the script, make sure the `namespace` is correct. This usually matches your Stride project name (eg *MyTrailEffect*).

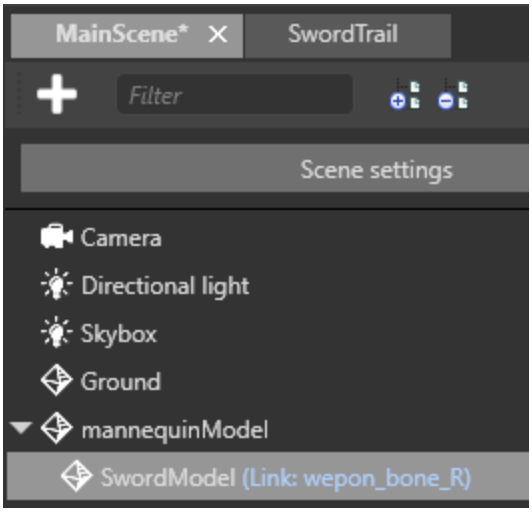
```
MyTrailEffect.Game
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using SiliconStudio.Core;
7  using SiliconStudio.Xenko.Engine;
8
9  namespace MyTrailEffect
10 {
11     /// <summary>
12     /// A script which spawns a timed instance f
13     /// </summary>
       0 references
14     public class SpawnTrail : AsyncScript
15     {
16
17         private float timeIntervalCountdown = 0f
```

5. Save the script and the Visual Studio project (**Ctrl + Shift + S**).

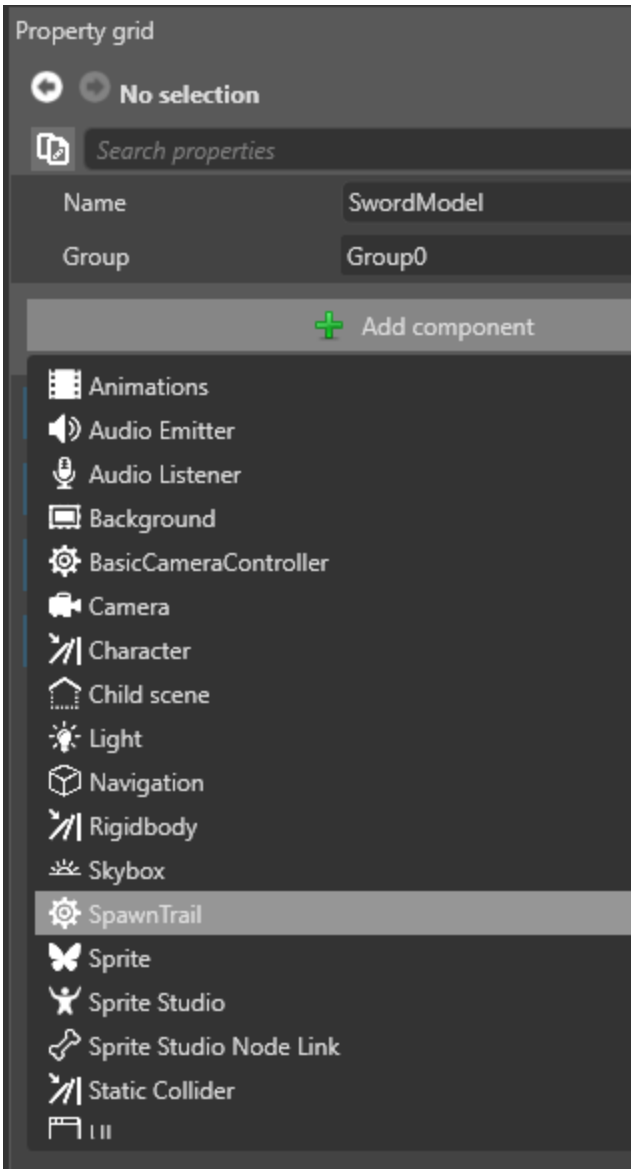
6. In Game Studio, reload the assemblies.



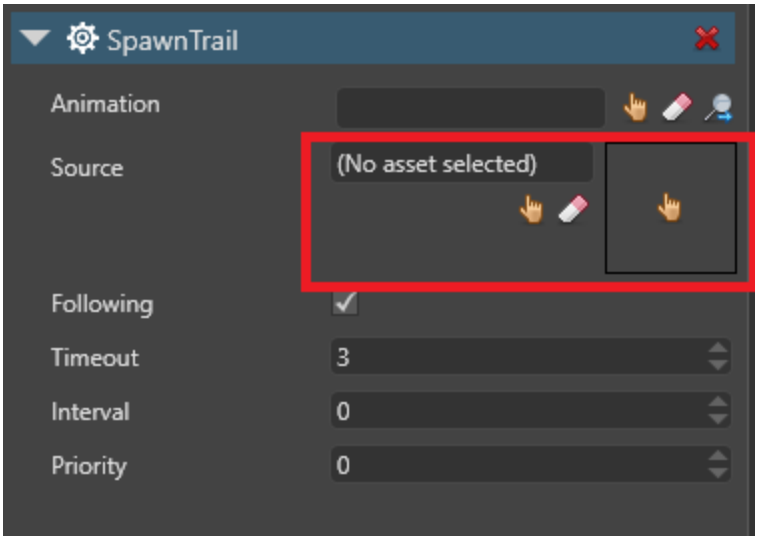
7. In the **MainScene**, select the **SwordModel**.



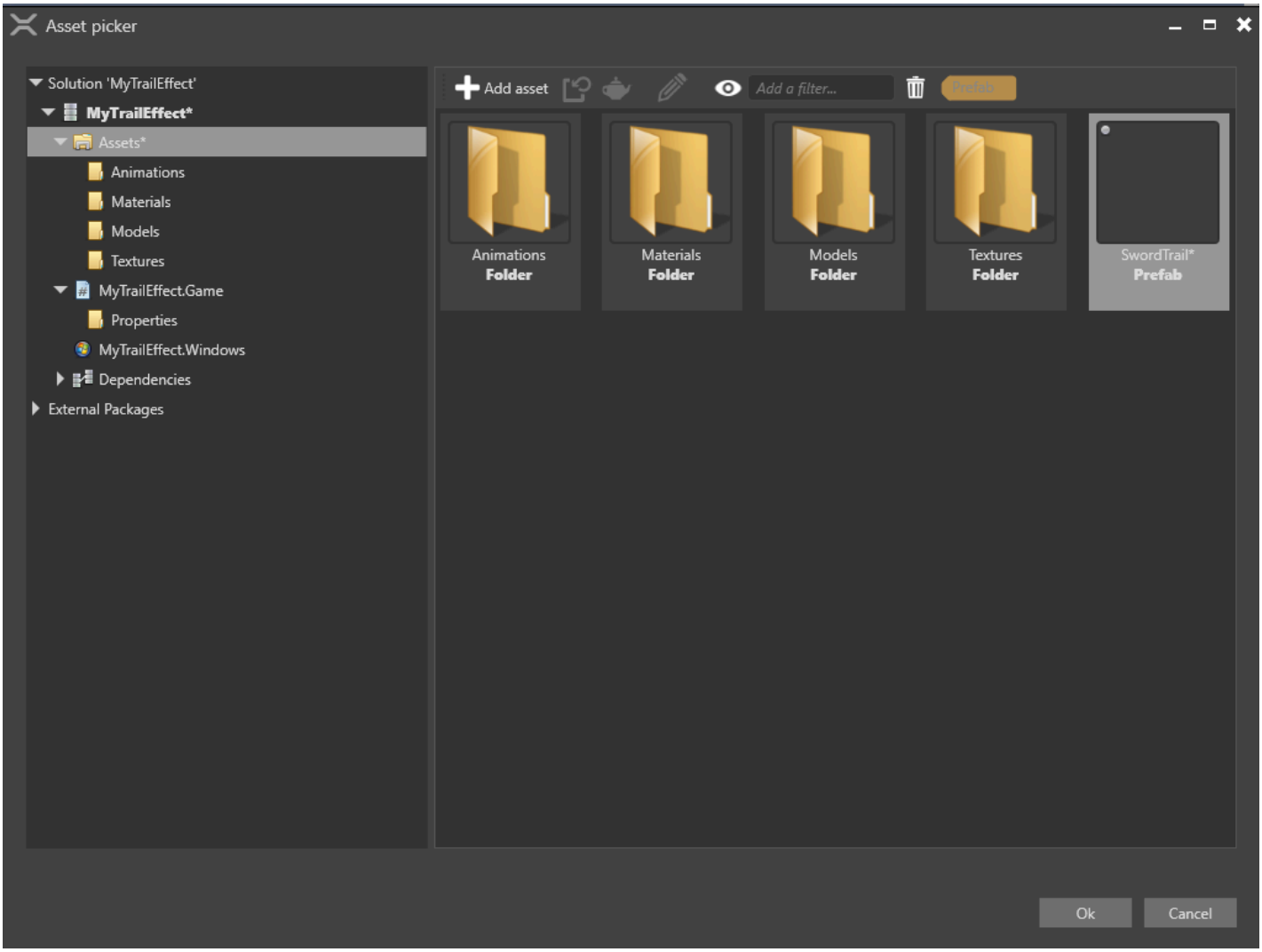
8. In the SwordModel properties, click **Add component** and select the **SpawnTrail** script. This adds the script as a component.



9. Under the **SpawnTrail** component properties, next to **Source**, click (**Select an asset**).

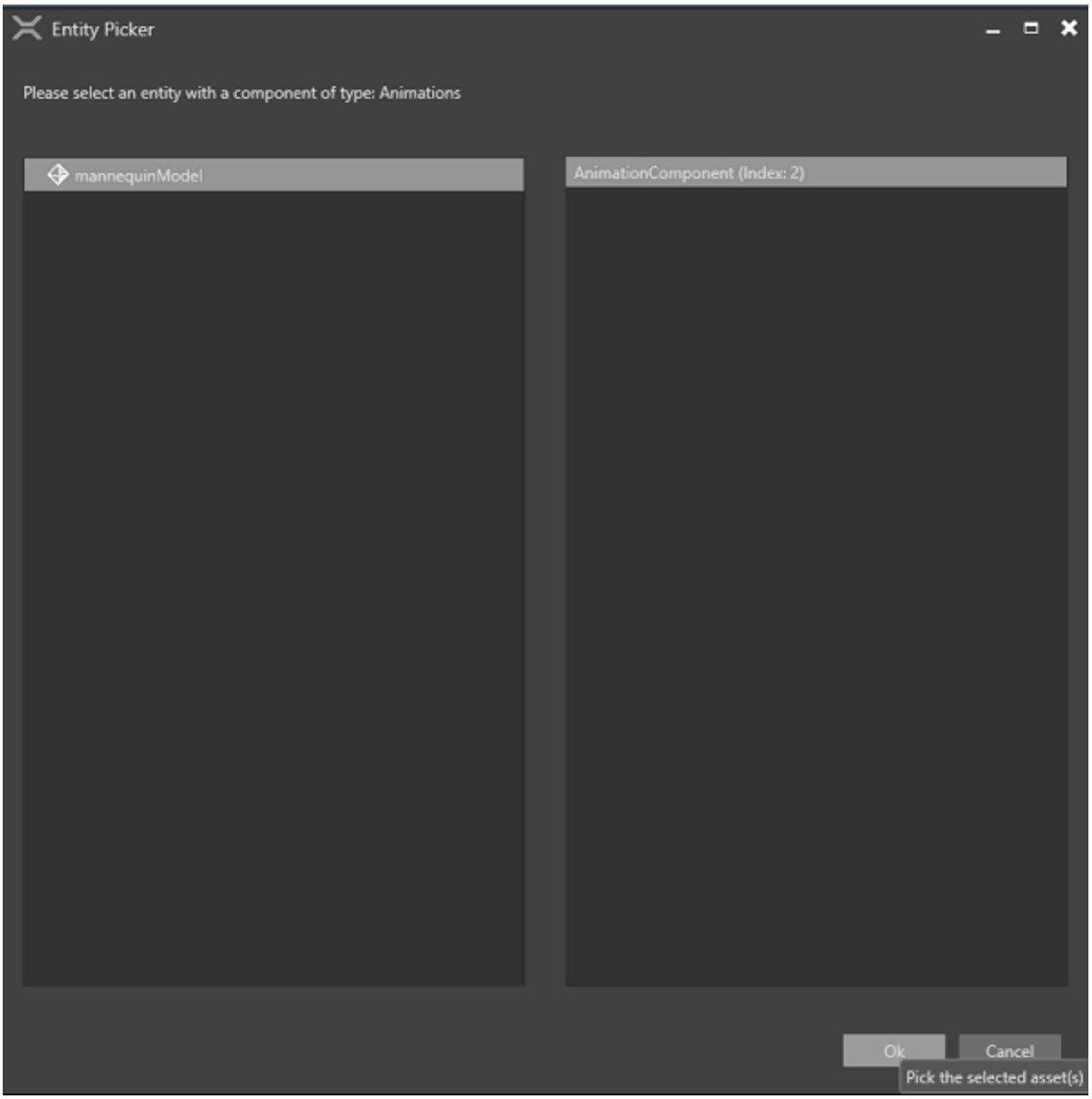


10. In the Entity Picker, select the **SwordTrail** prefab.



11. In the SpawnTrail component, in the **Animation** field, click the hand icon (**Select an asset**). The **Select an asset** window opens.

    In the left pane, select the **mannequinModel** and click **OK**.



12. Run the game.

Thanks to our script, the particle effect appears at the start of the sword swing animation and disappears at the end.

9. Adjust the trail start time

1. With the **Sword_R** animation asset selected, check the swing animation in the **Asset Preview** in the bottom-right. (If the Asset Preview isn't displayed, check **View > Asset Preview**.)

   The Asset Preview shows the animation length in seconds. If you look closely, you can see the mannequin doesn't begin to swing the sword down until about 0.1 seconds into the animation. Let's set the trail effect to spawn just when the mannequin swings.

2. Select the **SwordModel**.

3. In the **SpawnTrail** properties, set the **Start time** to 0.06. This means the trail effect won't spawn until 0.06 seconds into the swing animation, which looks a little more natural. Feel free to tweak this to your liking.

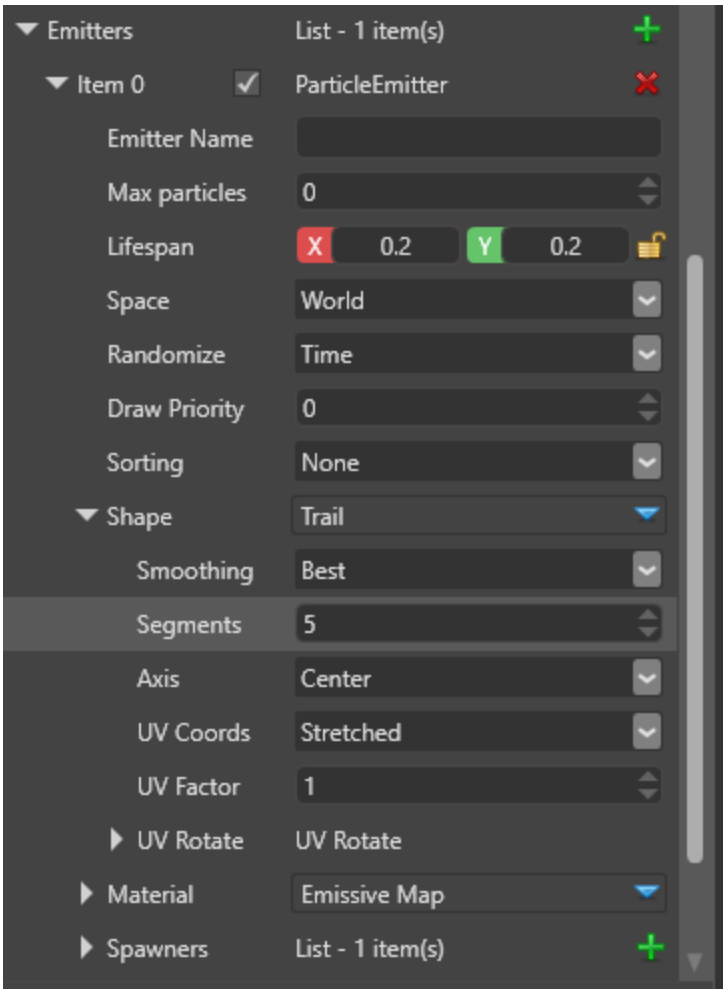
4. Run the game to see how it looks.

0:00

You might notice our trail effect looks a little jagged, creating a "spiderweb" effect. Let's make it more curved.

10. Curve the trail

1. In the **SwordTrail** prefab, on the **SwordTrail** entity, under **Particle System > Source > Emitters > Shape**, set **Smoothing** to **Best** and **Segments** to **5**.

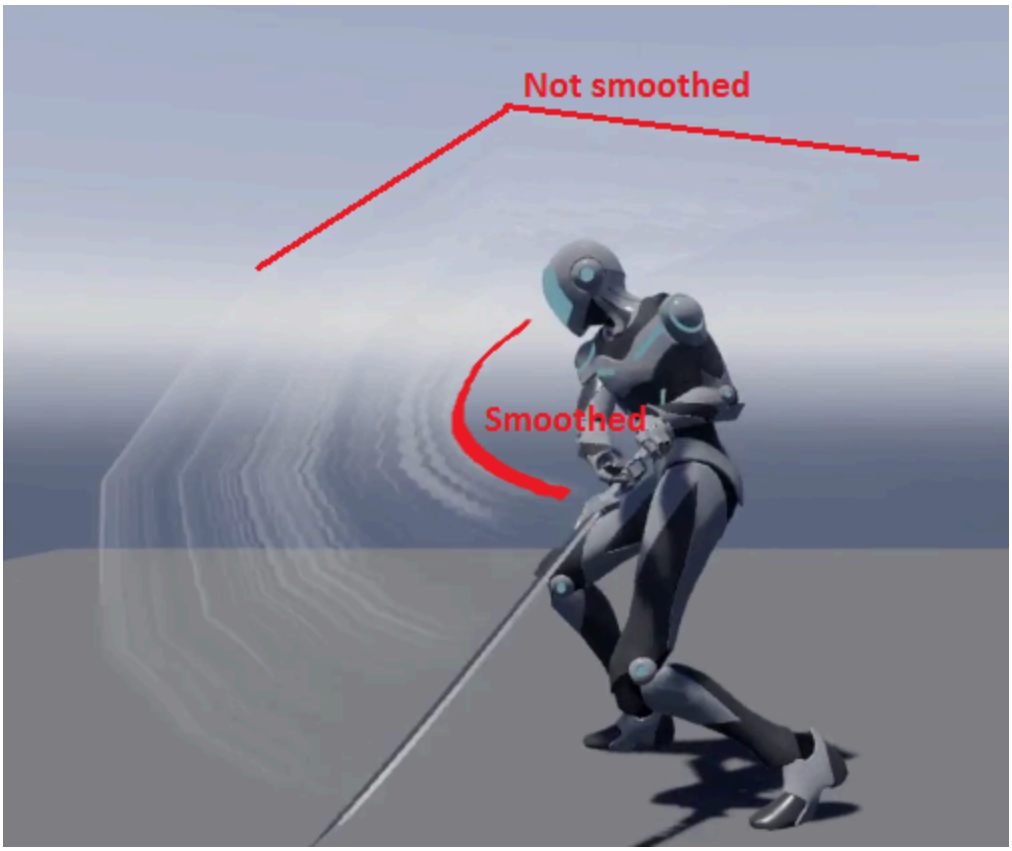


This adds three vertices between the particles of our trail, which should be enough to create a noticeably smoother effect.

2. Run the game.

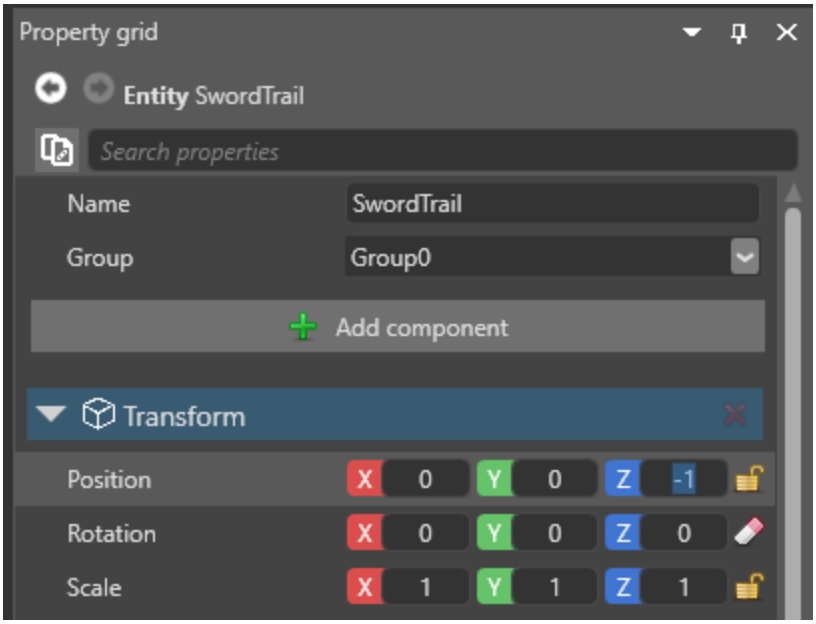
0:00

The inner curve, at the sword hilt, is smoother. But the curve at the sword's edge is still jagged.



We want to smooth the effect at the sword's edge, where it's more noticeable. To do that, we'll flip the particle direction.

1. Still in the **SwordTrail** prefab, in the **Transform** component properties, change the **Position** to **0, 0, -1**.



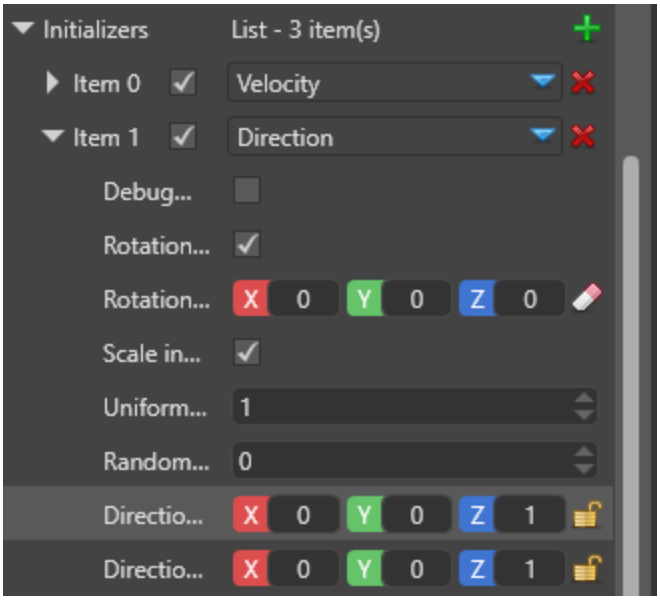
   This moves the starting point of the particle effect to the tip of the sword.

2. Run the game.

0:00

Now we have a new problem. Because we moved the particle effect to the tip of the sword, the particles are flying from the tip. We need to reverse their direction, so they move down along the sword blade to the hilt.

6. Under **Particle System > Source > Emitters > Initializers**, under the **Direction** initializer, change both the **Direction min** and **Direction max** to **0, 0, 1**. This inverts the trail direction.



7. Run the game.

0:00

Congratulations! You created a trail effect from scratch. How you tweak it now is up to you.

Sample project

Here's a more elaborate trail that combines multiple particle effects:

0:00

If you'd like to see how it works, download the project file and take a look.

See also

- Tutorial: Custom particles
- Tutorial: Inheritance
- Tutorial: Lasers and lightning
- Particles
- Create particles
- Model node links

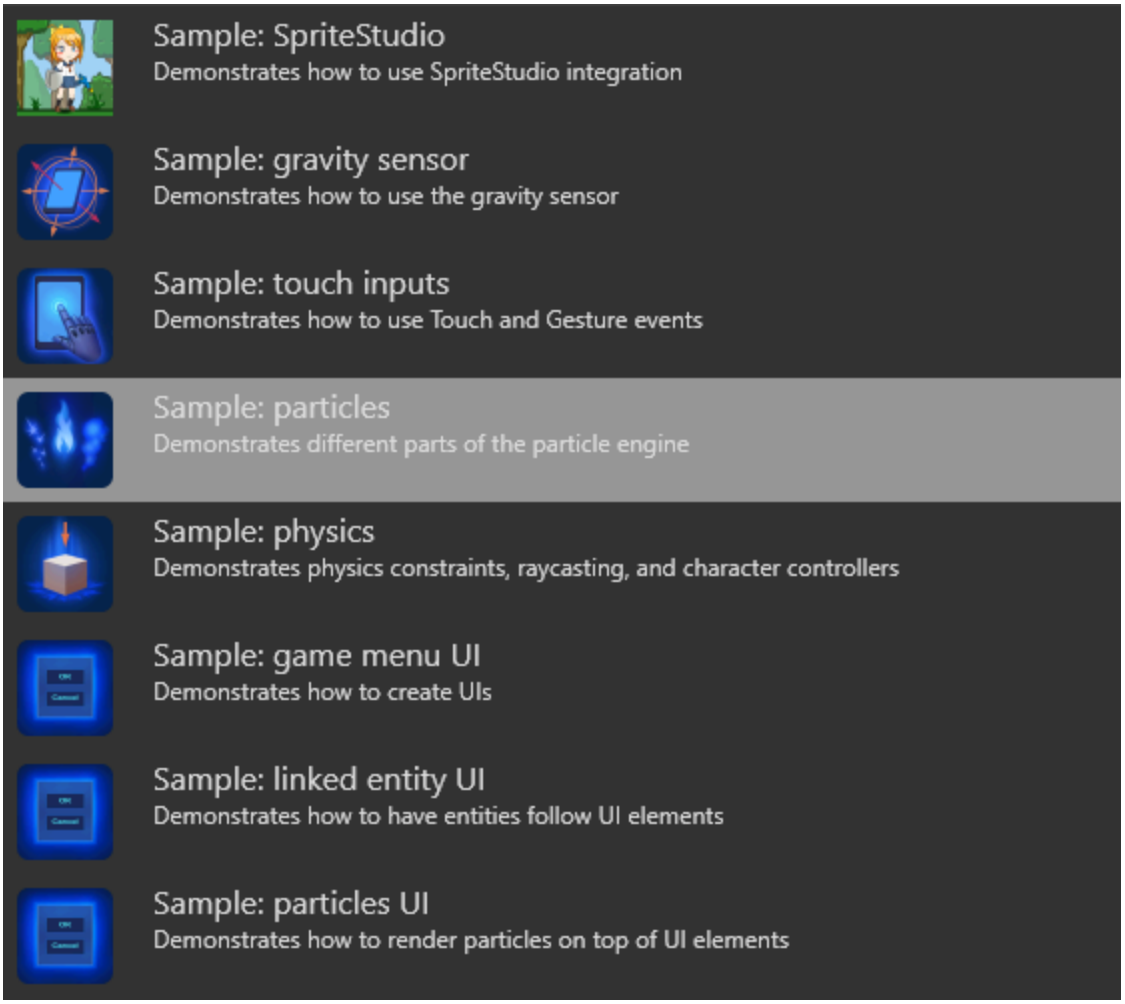
Tutorial: Custom particles

Intermediate Artist Programmer

This walkthrough shows how you can create custom extensions for the particle system, providing functionality not available in the core engine.

If you're not familiar with editing particles, see Create particles.

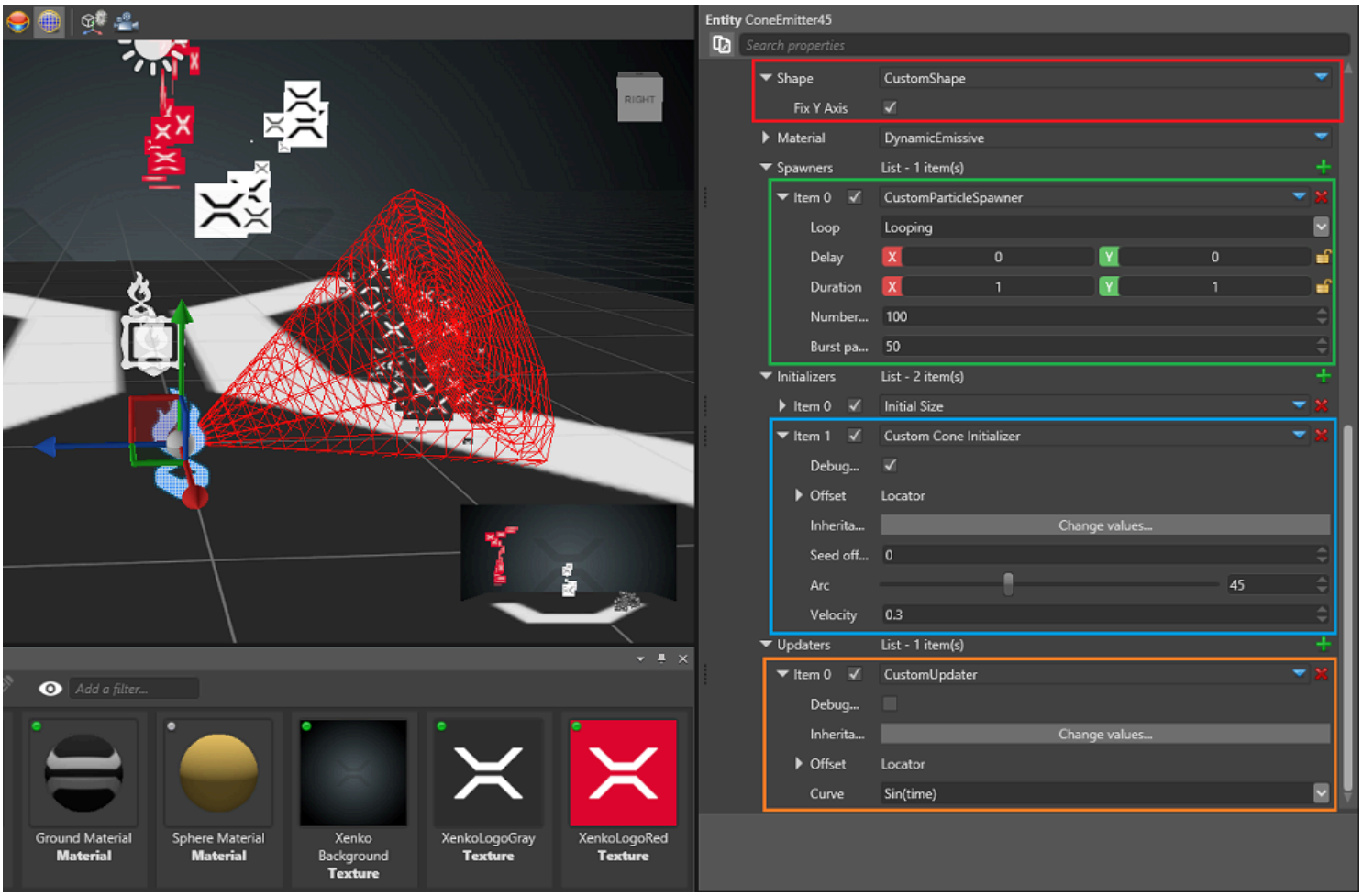
Start by creating a new **Sample: Particles** project.



This project contains different scenes that demonstrate different ways to use particles. Open the **CustomParticles** scene.

There are three particle entities in the scene: **ConeEmitter15**, **ConeEmitter30**, and **ConeEmitter45**.

Select one of the particle entities. In the Property Grid, navigate to its source particle system and expand the emitter.



There are four custom elements in this emitter:

- The custom spawner is similar to the spawn-per-second spawner, but also emits a burst of particles every time it loops.
- The custom initializer initially positions the particles in a cone shape and sets their velocity accordingly.
- The custom updater operates on a new particle field named **RactangleXY**, allowing the shape builder to use non-uniform sizes when building the billboards.
- The custom shape builder is similar to the billboard with two additions. It can create non-uniform rectangles, rather than the standard squares, and it can align (fix) the rectangle's Y axis to the world's Y axis rather than the camera space.

Spawner

We'll create a spawner which emits particles per second **and** in bursts every few seconds. We could do this by adding two different spawners, but for this sample we'll combine them.

```
  [DataContract("CustomParticleSpawner")] // Used for serialization, a good practice is to
have the data contract have the same name as the class
  [Display("CustomParticleSpawner")]
  public sealed class CustomParticleSpawner : ParticleSpawner
  {
      [DataMemberIgnore]
      private float carryOver;    // Private members do not appear on the Property Grid

      [DataMember(100)]               // When data is serialized, this attribute decides
its priority
      [Display("Number of particles")] // This is the name which will be displayed on the
Property Grid
      public float SpawnCount { get; set; }

      [DataMemberIgnore]
      private float burstTimer;    // Private members do not appear on the Property Grid

      [DataMember(200)]               // When data is serialized, this attribute decides
its priority
      [Display("Burst particles")]     // This is the name which will be displayed on the
Property Grid
      public float BurstCount {get;set;}

                ...

      public override int GetMaxParticlesPerSecond()
      {
          return (int)Math.Ceiling(SpawnCount) + (int)Math.Ceiling(BurstCount);
      }

      public override void SpawnNew(float dt, ParticleEmitter emitter)
      {
          // State is handled by the base class. Generally you only want to spawn particle
when in active state
          var spawnerState = GetUpdatedState(dt, emitter);
          if (spawnerState != SpawnerState.Active)
              return;

          // Calculate particles per second
          var toSpawn = spawnCount * dt + carryOver;
          var integerPart = (int)Math.Floor(toSpawn);
          carryOver = toSpawn - integerPart;

          // Calculate burst particles
          burstTimer -= dt;
          if (burstTimer < 0)
```

```
            {
                burstTimer += 1f;
                integerPart += (int)Math.Floor(BurstCount);
            }

            // Lastly, tell the emitter how many new particles do we want to spawn this frame
            emitter.EmitParticles(integerPart);
        }
    }
```

This class mimics the ParticleSpawner, with the addition of a `BurstCount` and a `burstTimer` to control how often and how many particles are spawned in bursts.

The `SpawnNew` method is called every frame to allow the spawner to calculate how many new particles should be emitted in the emitter based on the elapsed time.

As an exercise, try implementing the following changes:

- Rather than one-second bursts, create a property and have the user control the timing.
- Remove the spawn-per-second fields and make it a pure burst spawner.

Our spawner only emits particles, but doesn't set any fields. This is done by the initializer.

Initializer

We want to place the particles in a cone and shoot them outwards when they spawn.

```
    [DataContract("CustomParticleInitializer")]
    [Display("Cone Initializer")]
    public class CustomParticleInitializer : ParticleInitializer
    {
        [DataMember(100)]
        [DataMemberRange(0, 120, 0.01, 0.1)]
        [Display("Arc")]
        public float Angle = 20f;

        [DataMember(200)]
        [Display("Velocity")]
        public float Strength = 1f;

        public CustomParticleInitializer()
        {
            RequiredFields.Add(ParticleFields.Position);
            RequiredFields.Add(ParticleFields.Velocity);
            RequiredFields.Add(ParticleFields.RandomSeed);
```

```
        }

        public unsafe override void Initialize(ParticlePool pool, int startIdx, int endIdx,
int maxCapacity)
        {
                        ...
        }
    }
```

Our initializer simply defines an angle for the cone and strength for the velocity. Any scaling and rotation of the cone come from the location inheritance and offset, which are common for all initializers and updaters and are ready to use. For more information, see the ParticleInitializer.

The constructor for the initializer is important, as it sets the list of required fields we'll use. The initializer sets the particle's position and velocity, so we add those, and needs to generate some randomness, so we also add the random seed which we are going to use. All particles have `Life` and `RandomSeed` fields when they spawn.

```
// This method is called for all new particles once the initializer is added to an emitter.
Rather than updating all of them, we are given a starting and end indices and must only use
particles in the defined range.
public unsafe override void Initialize(ParticlePool pool, int startIdx, int endIdx, int
maxCapacity)
{
  // Make sure the fields exist and avoid illegal memory access
        if (!pool.FieldExists(ParticleFields.Position) ||
!pool.FieldExists(ParticleFields.Velocity) || !pool.FieldExists(ParticleFields.RandomSeed))
                return;

        var posField = pool.GetField(ParticleFields.Position);
        var velField = pool.GetField(ParticleFields.Velocity);
        var rndField = pool.GetField(ParticleFields.RandomSeed);

        var range = (float) (Angle*Math.PI/180f);
        var magnitude = WorldScale.X;

        var i = startIdx;
        while (i != endIdx)
        {
                var particle = pool.FromIndex(i);
                var randSeed = particle.Get(rndField);

                var x = (randSeed.GetFloat(RandomOffset.Offset2A + SeedOffset) -
0.5f)*range;
                var z = (randSeed.GetFloat(RandomOffset.Offset2B + SeedOffset) - 0.5f) *
```

```
range;

                var u = (randSeed.GetFloat(RandomOffset.Offset2A + SeedOffset) - 0.5f) *
range;
                var v = (randSeed.GetFloat(RandomOffset.Offset2B + SeedOffset) - 0.5f) *
Math.PI;

                var xz = (float) Math.Sin(u);
                var particleRandPos = new Vector3((float) Math.Cos(v) * xz,
(float)Math.Sqrt(1 - u*u), (float)Math.Sin(v) * xz);
                particleRandPos.Normalize();

                particleRandPos *= magnitude;
                WorldRotation.Rotate(ref particleRandPos); // WorldRotation is the current
rotation of our initializer. We can use it as it is, since inheritance and offset are
already taken in account.

                (*((Vector3*) particle[posField])) = particleRandPos + WorldPosition; //
WorldPosition is the current position of our initializer. We can use it as it is, since
inheritance and offset are already taken in account.

                (*((Vector3*) particle[velField])) = particleRandPos * Strength;

                i = (i + 1) % maxCapacity;
        }
}
```

Updater

We want our updater to change a particle's width and height every frame based on a simple sine function over the particle's life.

Because there's no such field yet, start by creating a new particle field. Let's name it `RactangleXY`:

```
  public static class CustomParticleFields
  {
      public static readonly ParticleFieldDescription<Vector2> RectangleXY = new
ParticleFieldDescription<Vector2>("RectangleXY", new Vector2(1, 1));
  }
```

The field has type Vector2, since we only need two values for the width and the height. No fields are added automatically to the particles, so even if you have many declarations, the particle size won't change. Fields are only added when we plug a module which requires them, such as the custom updater below.

For API reference, see ParticleUpdater.

```
  [DataContract("CustomParticleUpdater")] // Used for serialization so that our custom
object can be saved. A good practice is to have the data contract have the same name as the
class name.
  [Display("CustomUpdater")]                              // Unless a display name is
specified, the name of the data contract will be used. Sometimes we want to hide it and
display something simpler instead.
  public class CustomParticleUpdater : ParticleUpdater
  {
      [DataMemberIgnore]        // Public fields and properties are serialized. We want to
avoid this in some cases and can use the DataMemberIgnore attribute.
      public override bool IsPostUpdater => true; // By making this updater a post-updater
we can ensure it will be called for both newly spawned and old particles (1 frame or older)

      [DataMember(10)]  // This public field will be serialized. With the DataMember
attribute we can specify the serialization and display order.
      public AnimatedCurveEnum Curve; // Refer to the actual sample code for
AnimatedCurveEnum

                // In the constructor we have to specify all the fields we need for this
updater.
                // It calculates our newly created field by using the particle's lifetime so
we need "RectangleXY" and "Life"
      public CustomParticleUpdater()
      {
          // This is going to be our "input" field
          RequiredFields.Add(ParticleFields.Life);

          // This is the field we want to update
          // It is not part of the basic fields - we created it just for this updater
          RequiredFields.Add(CustomParticleFields.RectangleXY);
      }

                // The update method is called once every frame and requires the updater to
iterate over all particles in the pool and update their fields.
                // If the updater is a post-updater it will get called **after** spawning
new particles for this frame and might overwrite their initial values on the same frame
                // If the updater is not a post-updater it will get called **before**
spawning new particles for this frame and can't overwrite their initial values for the first
frame
      public override void Update(float dt, ParticlePool pool)
      {
                        ...
```

```
        }
    }
```

Let's take a look at the `Update` method. The sample code is longer, but here we've trimmed it for the sake of simplicity.

```
public override void Update(float dt, ParticlePool pool)
{
  // Make sure the fields exist and avoid illegal memory access
  if (!pool.FieldExists(ParticleFields.Life) ||
!pool.FieldExists(CustomParticleFields.RectangleXY))
      return;

  var lifeField = pool.GetField(ParticleFields.Life);
  var rectangleField = pool.GetField(CustomParticleFields.RectangleXY);

  // X and Y sides depend on sin(time) and cos(time)
  foreach (var particle in pool)
  {
      // Get the particle's remaining life. It's already normalized between 0 and 1
      var lifePi = particle.Get(lifeField) * MathUtil.Pi;

      // Set the rectangle as a simple function over time
      particle.Set(rectangleField, new Vector2((float)Math.Sin(lifePi),
(float)Math.Cos(lifePi)));
  }
}
```

The updater will animate all particles' RectangleXY fields with a simple sine and cosine functions over their life.

In the next step we'll demonstrate how to display the created values.

Shape builder

The `shape builder` is the class which takes all particle fields and creates the actual shape we are going to render. It's a little long, so let's break it down.

```
        public override int QuadsPerParticle { get; protected set; } = 1;
```

The engine draws quads using 1 quad = 4 vertices = 6 indices, but we can only specify the number of quads we need. For a rectangle we need only one.

> **NOTE**
>
> The number of quads is important because the vertex buffer is allocated and mapped prior to writing out the vertex data. If we allocate smaller buffer it might result in illegal memory access and corruption.

```
public unsafe override int BuildVertexBuffer(ParticleVertexBuilder vtxBuilder, Vector3
inverseViewX, Vector3 inverseViewY,
  ref Vector3 spaceTranslation, ref Quaternion spaceRotation, float spaceScale,
ParticleSorter sorter)
```

This method is called when it needs our shape builder to iterate over all particles and build the shape. The ParticleVertexBuilder is the wrapper around our vertex stream. We'll use it to write out the vertex data for the particles.

`inverseViewX` and `inverseViewY` are unit vectors in camera space passed down to the shape builder if we need to generate camera-facing shapes.

```
  foreach (var particle in sorter)
  {
      var centralPos = particle.Get(positionField);

      var particleSize = sizeField.IsValid() ? particle.Get(sizeField) : 1f;
      var rectangleSize = rectangleField.IsValid() ? particle.Get(rectangleField) : new
Vector2(1, 1);
      var unitX = invViewX * (particleSize * 0.5f) * rectangleSize.X;
      var unitY = invViewY * (particleSize * 0.5f) * rectangleSize.Y;

      // Particle rotation. Positive value means clockwise rotation.
      if (hasAngle) { ... }

      var particlePos = centralPos - unitX + unitY;
      var uvCoord = new Vector2(0, 0);

      // 0f 0f
      vtxBuilder.SetAttribute(posAttribute, (IntPtr)(&particlePos));
      vtxBuilder.SetAttribute(texAttribute, (IntPtr)(&uvCoord));
      vtxBuilder.NextVertex();

      // 1f 0f
      particlePos += unitX * 2;
      uvCoord.X = 1;
```

```
            vtxBuilder.SetAttribute(posAttribute, (IntPtr)(&particlePos));
            vtxBuilder.SetAttribute(texAttribute, (IntPtr)(&uvCoord));
            vtxBuilder.NextVertex();

            // 1f 1f
            particlePos -= unitY * 2;
            uvCoord.Y = 1;
            vtxBuilder.SetAttribute(posAttribute, (IntPtr)(&particlePos));
            vtxBuilder.SetAttribute(texAttribute, (IntPtr)(&uvCoord));
            vtxBuilder.NextVertex();

            // 0f 1f
            particlePos -= unitX * 2;
            uvCoord.X = 0;
            vtxBuilder.SetAttribute(posAttribute, (IntPtr)(&particlePos));
            vtxBuilder.SetAttribute(texAttribute, (IntPtr)(&uvCoord));
            vtxBuilder.NextVertex();

            renderedParticles++;
        }
```

Our particles' width and height depend both on the uniform size field `Size` and the field we created earlier in this walkthrough, `RectangleXY`. From there, we need to set the positions and texture coordinates for the four corner vertices of our quad. The number of vertices we have to set is per particle four times the number of quads we requested.

You can add more complicated shapes or attributes here if your game requires them.

Conclusion

With these 4 custom modules you can add a lot of functionality to the particle engine and tailor behavior to your needs. Because they're all serialized and loaded in Game Studio, once you create them, you can use them directly from Game Studio, together with the core modules.

If you want to experiment with the modules, try adding a new `.cs` file to the `CustomParticles.Game` project. You can duplicate one of the existing classes, but don't forget to change the class name and the data contract to avoid collisions.

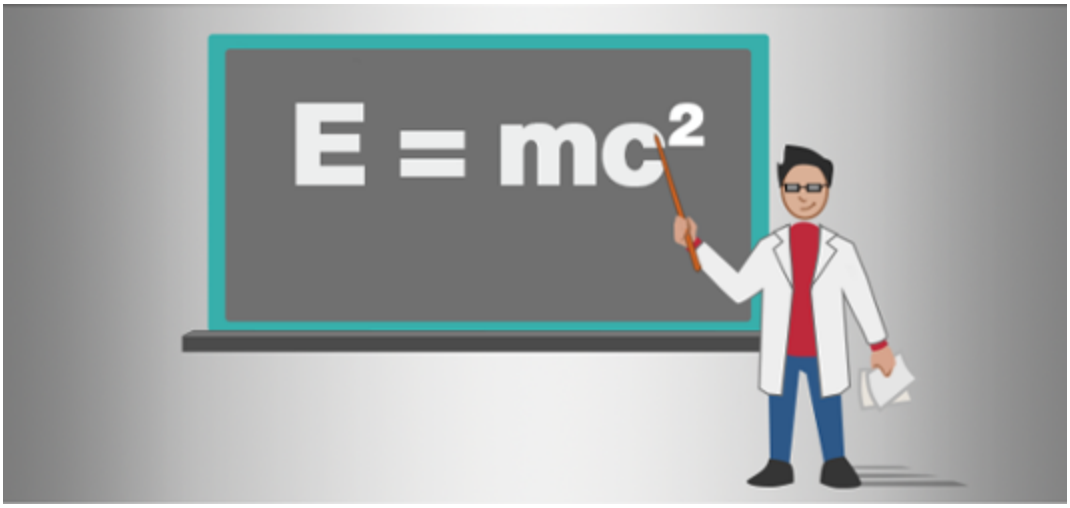
You can then reload the scripts in Game Studio. If they don't load, relaunch your project. If there are no compilation errors in your code you should see the new modules in the spawners, initializers, updaters and shape builders lists.

See also

- Tutorial: Create a trail

- Tutorial: Particle materials
- Tutorial: Inheritance
- Tutorial: Lasers and lightning
- Particles
- Create particles

Bepu Physics



> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

Stride simulates real-world physics such as gravity and collisions using "Bepu physics". This section explains how physics components work, how to add them to your project, and how to use them with scripts.

In this section

- Configuration: Setting up Bepu
- Simulation: Managing the simulations parameters
- Collidables: Physics objects in your game world
  - Statics: Strong immovable objects such as terrain, walls, floors, or large rocks
  - Bodies: Movable objects that can be knocked around, like cans, balls, or boxes
  - Kinematic Bodies: Entities moved programmatically, such as moving platforms or doors
  - Characters: Creatures moved programmatically, such as the player character, animals, or NPCs (non-player characters)
- Collider Shapes: Define the geometric shape of yours collidable components
- Triggers: Use triggers to detect passing objects
- Constraints: Join physics objects together, constrain them around points
- Physics Queries: Operations to find objects in the scene
- Physics Update: Updating logic alongside physics

Tutorials

- Physics tutorials
- Create a bouncing ball: Use the static collider and rigidbody components to create a ball bouncing on a floor
- Script a trigger: Create a trigger that doubles the size of a ball when the ball passes through it

Additional physics resources

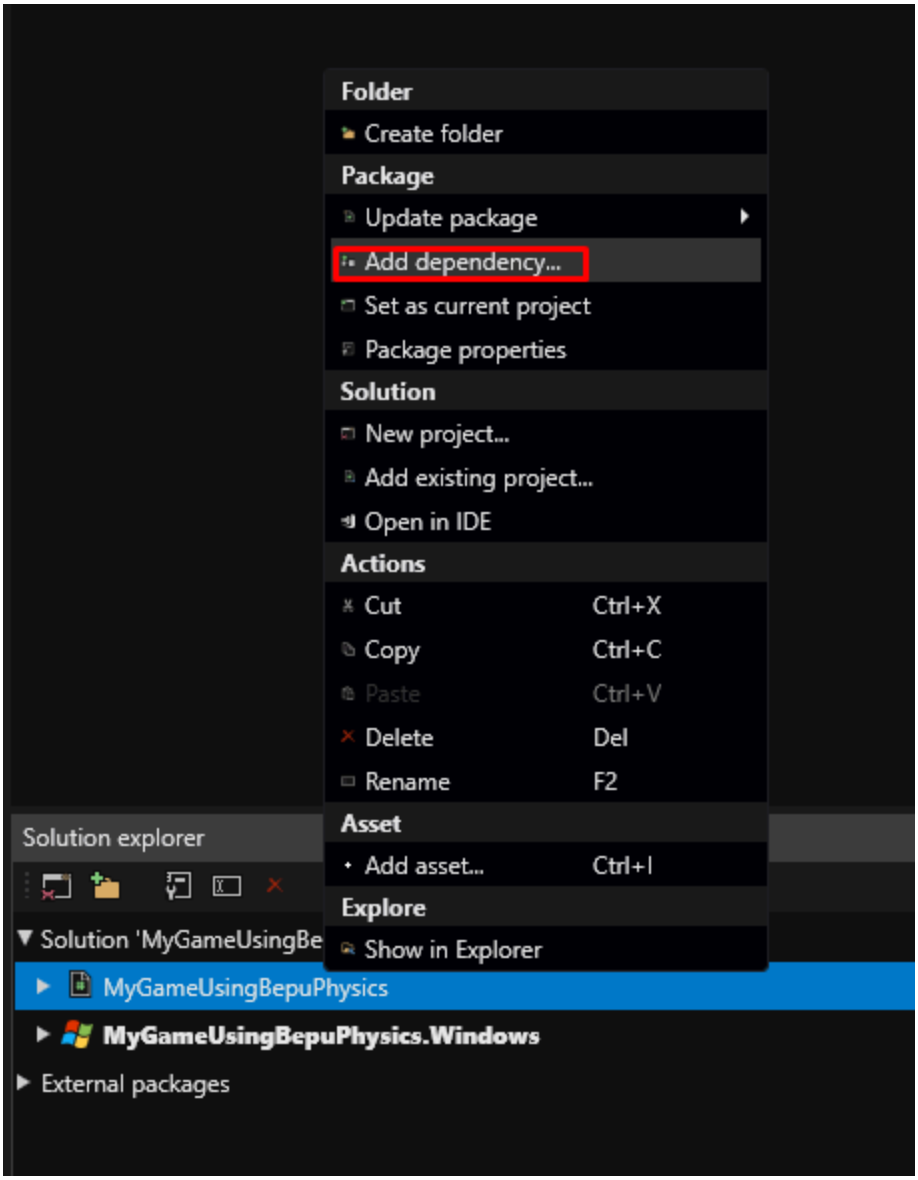
- Stride integrates the open-source Bepu Physics engine.
  - Explore the official Bepu website
  - Watch demos on the Bepu YouTube
- For solutions on mitigating physics jitter, refer to our guide on Fixing Physics Jitter

Bepu Physics - Configuration

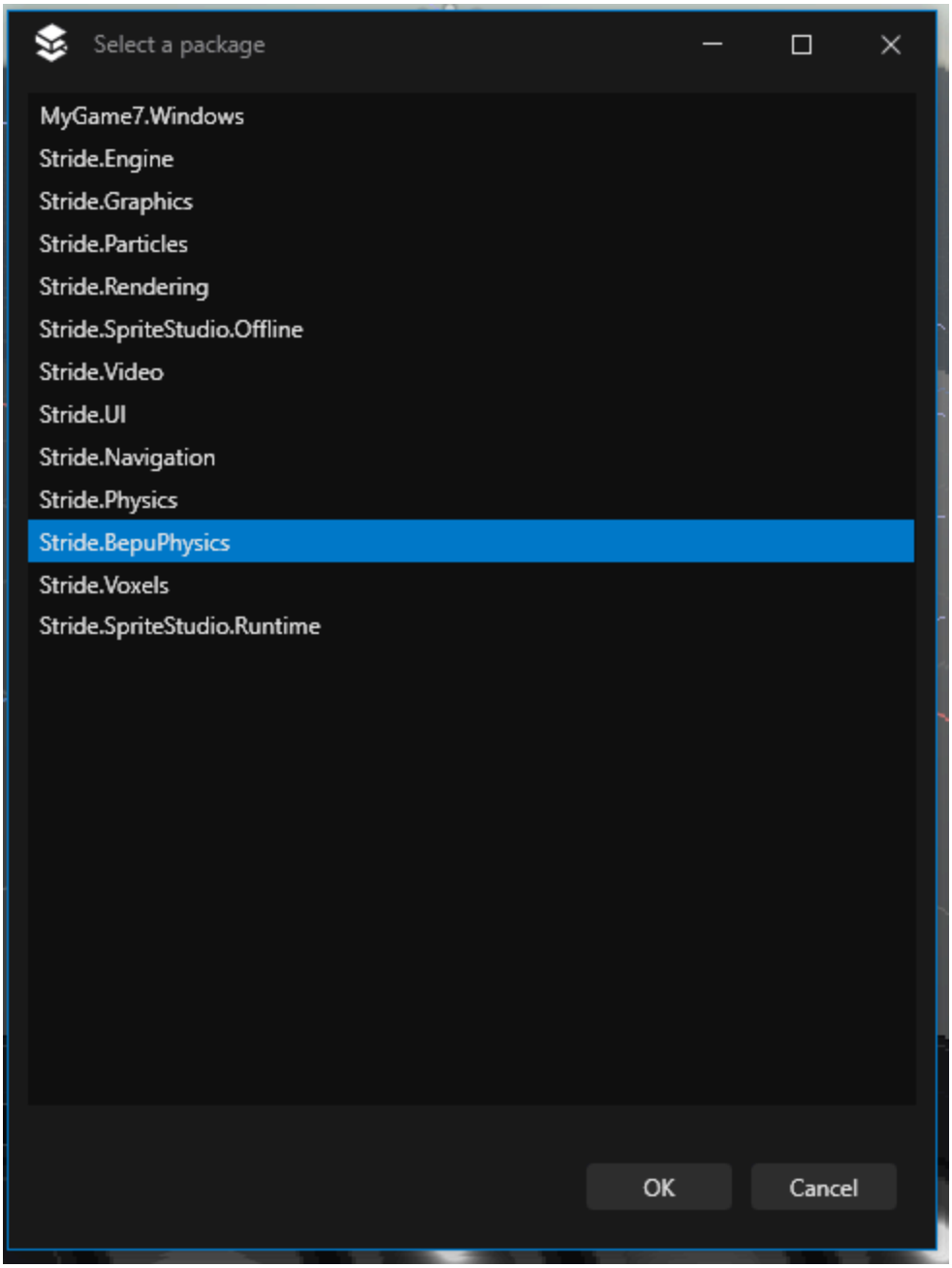
Beginner Designer

Adding Bepu physics to your game

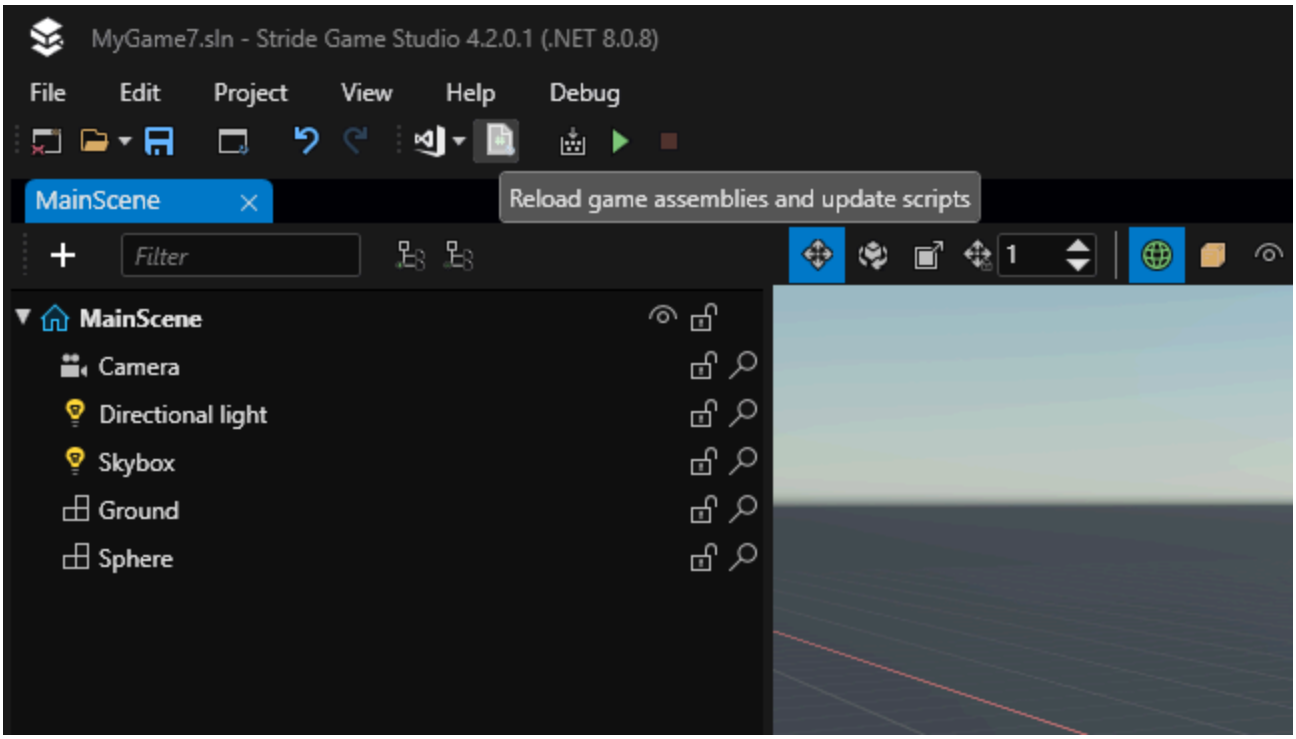
Some projects may not come with Bepu Physics pre-installed, to add it to your project right-click on the **game** project and select `Add dependency...`



A new window will open, select `Stride.BepuPhysics` and click `Ok`.



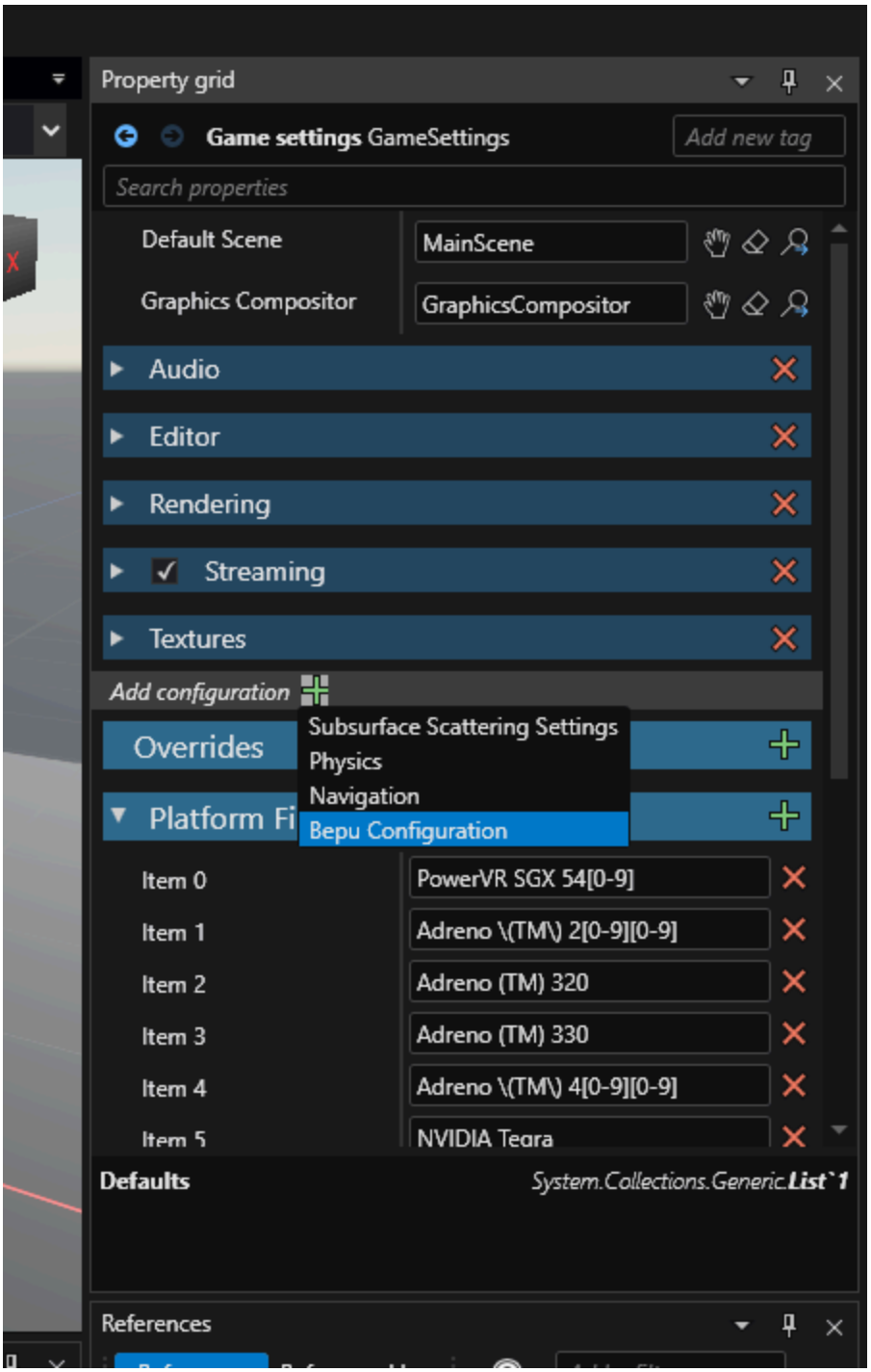
The editor may or may not reload the assemblies automatically depending on your settings, if it does not, press the `Assembly Reload` button.



You can now add Bepu's own physics component, but before playing around with that, you may want to add the physics configuration section to your game's settings.

Configuring Physics

To manage Bepu's simulation configuration, in the `Asset View` pane, select your `GameSettings` asset which is at the root of your asset folder, look at the property grid, press on the `+` sign next to `Add configuration` and select `Bepu Configuration`.



You can set up and control the different simulations your game hosts from this section. Add your first Simulation by pressing on the green `+` sign next to `Bepu Simulations`.

You should now have something like that:

See also

- Simulation
- Index

Physics simulation

Intermediate Programmer

Stride's physics are controlled by the Simulation class. You can change how Stride initializes the Simulation by going to the Configuration section of your game's settings.

These settings control the rules and parameters your physics simulation run under, enabling customization and fine-tuning of the physics environment to suit your game's requirements. You may also create multiple simulations and distribute your physics object between them depending on your needs.

Have a look at the API for more detail on what each property does.

Collision Layers and the Collision Matrix

You can assign your physics object to specific collision layers, those layers can then be set to ignore objects assigned to other layers.

Those rules can be set by modifying the collision matrix at runtime, or through the `Layer[...]` fields in the editor.

For example, pressing `Change values...` next to `Layer0` and un-ticking `Layer1` would cause all objects on `Layer0` to pass through objects on `Layer1`.

Retrieving the Simulation

There are multiple ways to retrieve a reference to this `BepuSimulation` from inside one of your `ScriptComponent`:

- The recommended way is through a reference to a physics component, something like `myBody.Simulation` as it is the fastest.
- Or through the `Entity.GetSimulation()` extension method.

Performance Considerations

The following are relevant excerpts from Bepu's documentation.

If physics causes your game to hang for a while when simulating for the first time, it may be related to just-in-time compilation. If it becomes a problem, consider running a small simulation that hits all the relevant codepaths (a bunch of objects colliding with constraints applied) behind a loading screen. Using an ahead-of-time compilation toolchain would also work.

See also

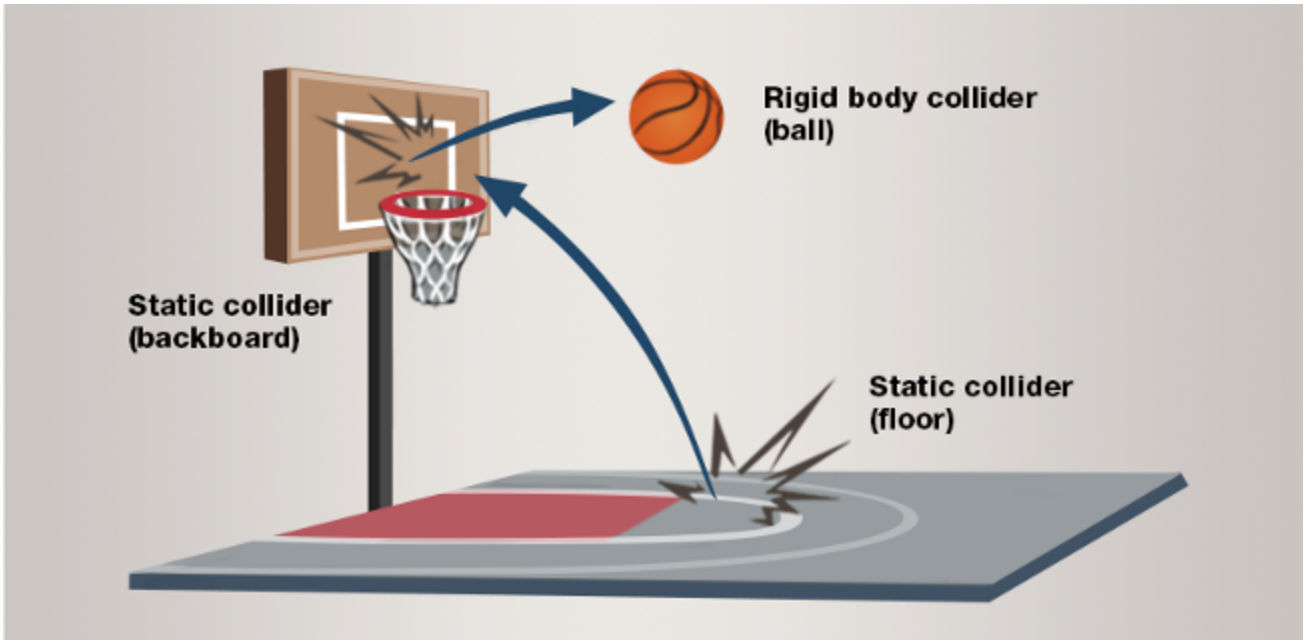
- Collidables
- Collider shapes

Collidables

Beginner Designer

Collidables are the base entity components for physics objects. There are three main types:

- Statics: Objects that don't move (terrain, walls, floors, large rocks)
- Bodies: Moving objects, affected by gravity and collisions (cans, balls, boxes) or Kinematics (moving platforms, doors)
- Characters: Colliders for basic characters (player character, animals, NPCs)



You can also:

- Define the shape of collidables components
- Make triggers, and detect when other physics objects pass through them
- Constrain collider movement with constraints

Collisions

Collidables interact according to the table below.

	Characters	**Bodies**	**Statics**
Characters	Collides	Collides and bounces	Collides
Bodies	Collides and bounces	Collides and bounces	Collides and bounces
Statics	Collides	Collides and bounces	Pass through

Characters do not have any inertia, and so cannot bounce off of bodies or statics when colliding with them.

Three other factor control whether two collidables would collide with each other, their `Collision Layer`, `Collision Group` and their `Contact Event Handler`

Collision Layers

The collision layer controls whether that object would collide with object on other layers.

This relationship is controlled through the Simulation's Collision Matrix.

Collision Group

This property is used to filter collisions inside a group of object, when two or more objects must share the same `Collision Layer`, but should not collide between each other.

It allows objects sharing the same `CollisionGroup.Id` to pass through each other when the absolute difference between their `IndexA`, `IndexB`, and `IndexC` is less than two.

Its utility is best shown through concrete examples.

- You have multiple characters `A, B, C, D` all set to the same `CollisionLayer`, they are split in two teams `A, B` and `C, D`. Members of the same team must not collide between each other, you can set `A, B`'s Id to 1 and `C, D`'s Id to 2.
- You have a chain of three colliders attached to each other `A, B, C`, you don't want A and C to collide with B, but A and C should collide together. Set A, B and C's Ids to 1 to start filtering, leave A's `IndexA` at 0, B's to 1 and C to 2. A and C will collide since the difference between their `IndexA` is equal to two, but neither of them will collide with B since they are both only one away from B's `IndexA` value.

Contact Event Handler

The contact event handler is a class that receives collision data whenever the object it is associated with collides with the world.

It is most often used to transform physics object into 'trigger boxes', areas that run events whenever objects, like the player character, passes through them. See Triggers.

If the contact event handler you bind to an object is set to `NoContactResponse`, the object will never prevent anything from passing through it, it will only collect collision events.

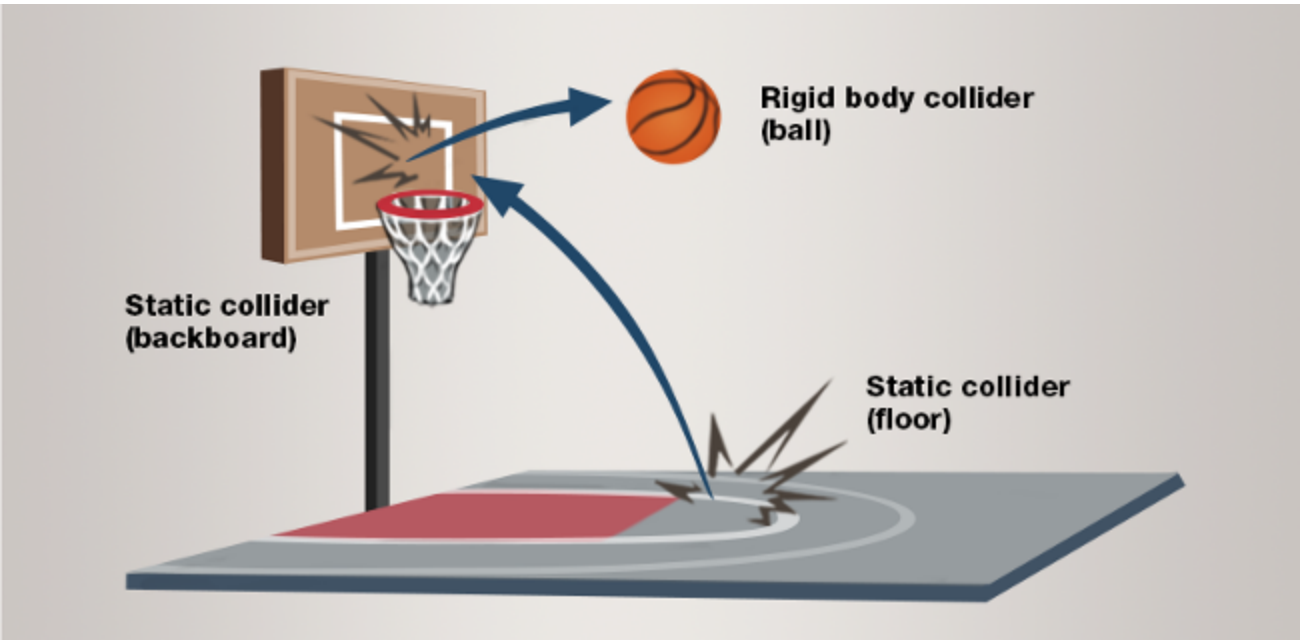
See also

- Configuration
- Static
- Body
- Character
- Colliders
- Physics tutorials

Static

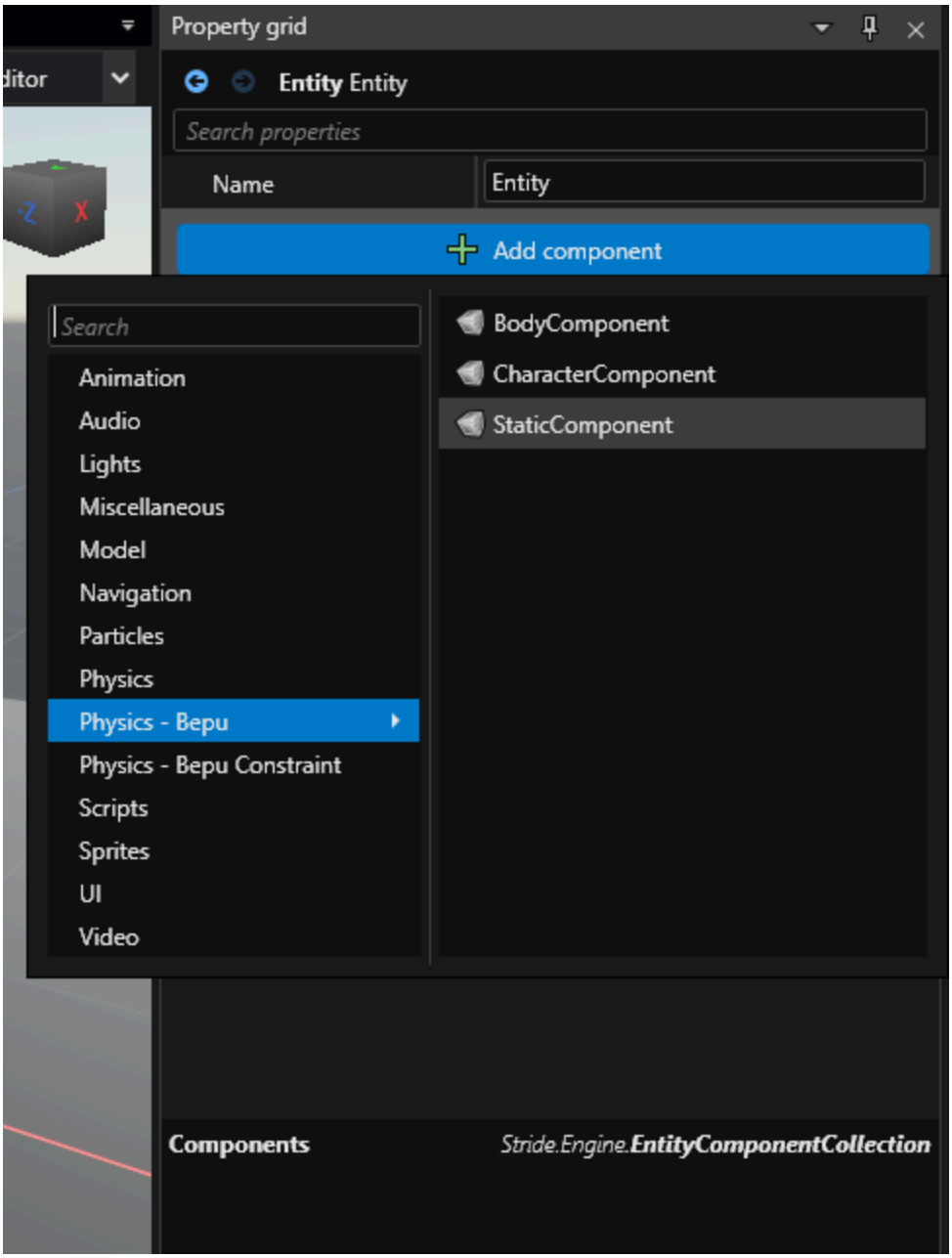
Beginner Designer

Static colliders aren't moved by forces such as gravity and collisions, but other physics objects can bump into them. Typical static colliders are strong immovable objects like walls, floors, large rocks, and so on.



Add a Static Component to an Entity

1. In the **Scene Editor**, select the entity you want to add the component to.
2. In the **Property Grid**, click `Add component`, hover `Physics - Bepu` and select `StaticComponent`.



> **NOTE**
>
> You will need to set a collider for this newly created static component, you can do so through the Collider property, see collider shapes.

Have a look at the API for more detail on the properties of this component.

Moving Static Components at Runtime

Moving a static collider while your game runs is generally not recommended as it will lead to unexpected collision issues and performance degradation. Moving it every couple of seconds should be fine, but if you have to move it more frequently, use a Body instead.

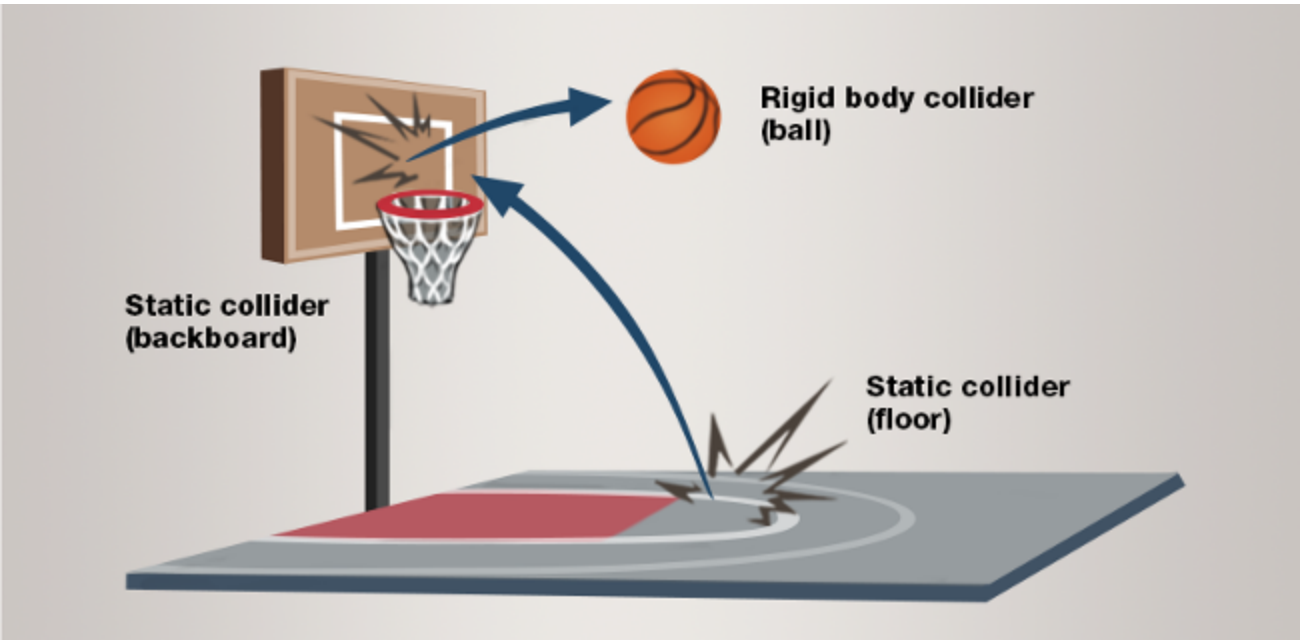
See also

- Bodies
- Characters
- Collider shapes
- Triggers

Body

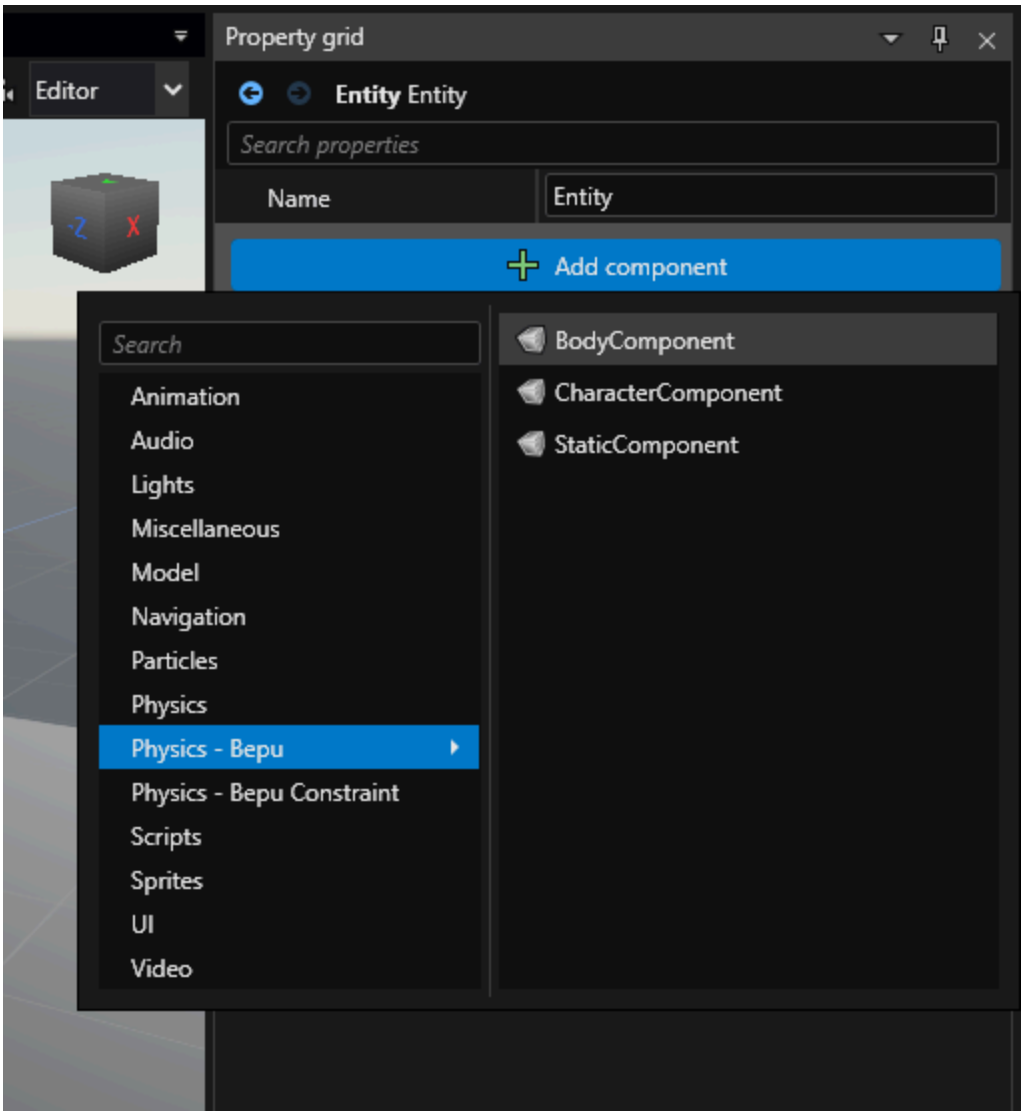
Beginner Designer

Body move based on physical forces applied to them, such as gravity and collisions. Typical (rigid)bodies are boxes, balls, furniture, and so on — objects that can be pushed, pulled, and knocked around, and also have effects on other bodies they collide with.



Add a Body Component to an entity

1. In the **Scene Editor**, select the entity you want to add the component to.
2. In the **Property Grid**, click `Add component`, hover `Physics - Bepu` and select `BodyComponent`.



NOTE

You will need to set a collider for this newly created body, you can do so through the Collider property, see collider shapes.

WARNING

Never use mesh colliders for bodies, use them only for statics, they are far too slow to be used as bodies. If you absolutely need a more complex shape than the primitive ones, use a convex hull instead, see collider shapes.

Have a look at the API for more detail on the properties of this component.

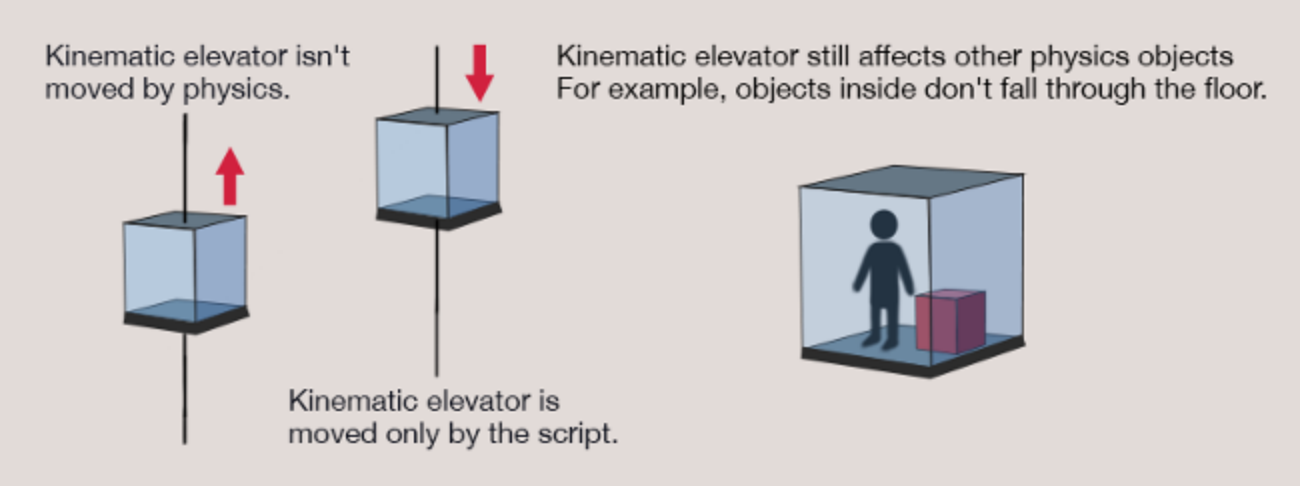
See also

- Kinematic rigidbodies
- Statics
- Characters
- Collider shapes
- Triggers

Kinematic Bodies

Sometimes you want to move bodies in a specific way rather than have physics apply outside forces, like drag, inertia, etc. For example, you might control an elevator directly through its `LinearVelocity` property, rather than have other objects push and pull it. This is a **kinematic** body.

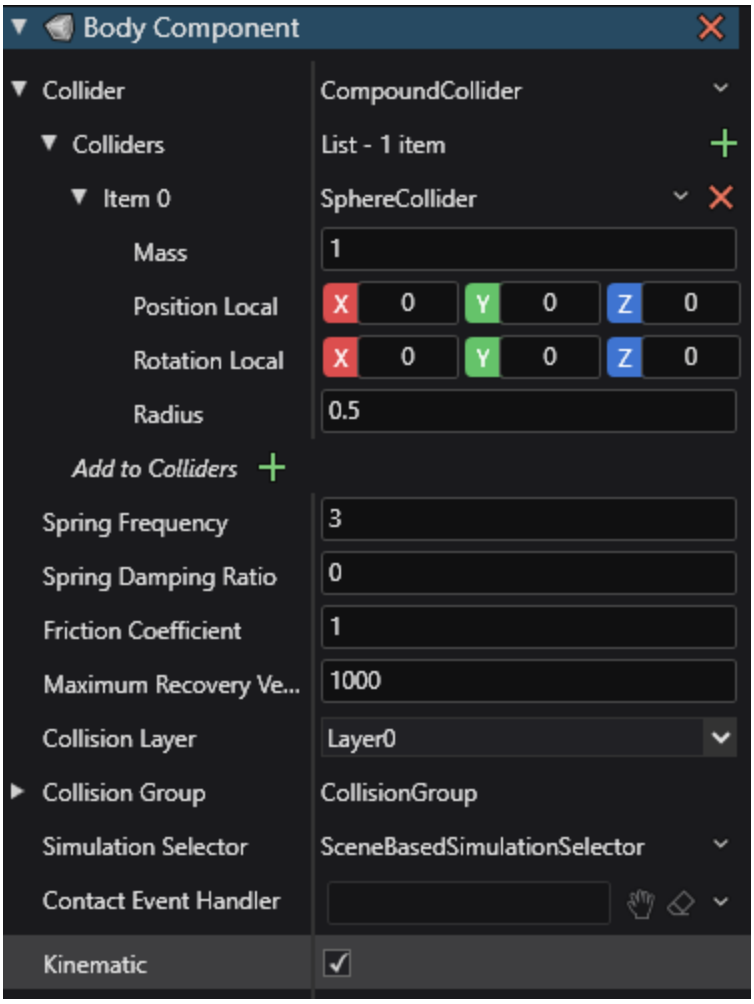
Although kinematic bodies aren't bound by physics, other objects can still collide with them. For example, in the case of the elevator, objects placed inside won't fall through the elevator floor.



Make a Kinematic Body

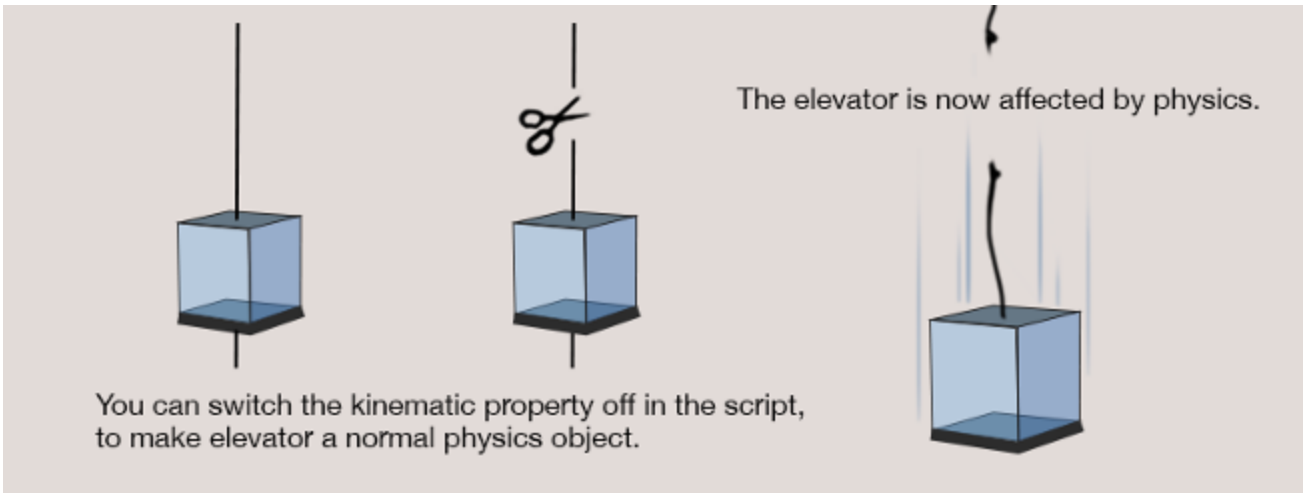
1. Select the entity you want to be a kinematic body.

2. In the **Property Grid**, under the **Body** component properties, select `Kinematic`.



Scripting Kinematic Bodies

You can script the `Kinematic` property to turn on and off on certain events. For example, imagine our kinematic elevator's suspension cables are cut. You can script the **Kinematic** property to change to *false* when this happens. The elevator becomes subject to the usual forces of physics, and falls.



See also

- bodies

- Statics
- Characters
- Collider shapes
- Triggers

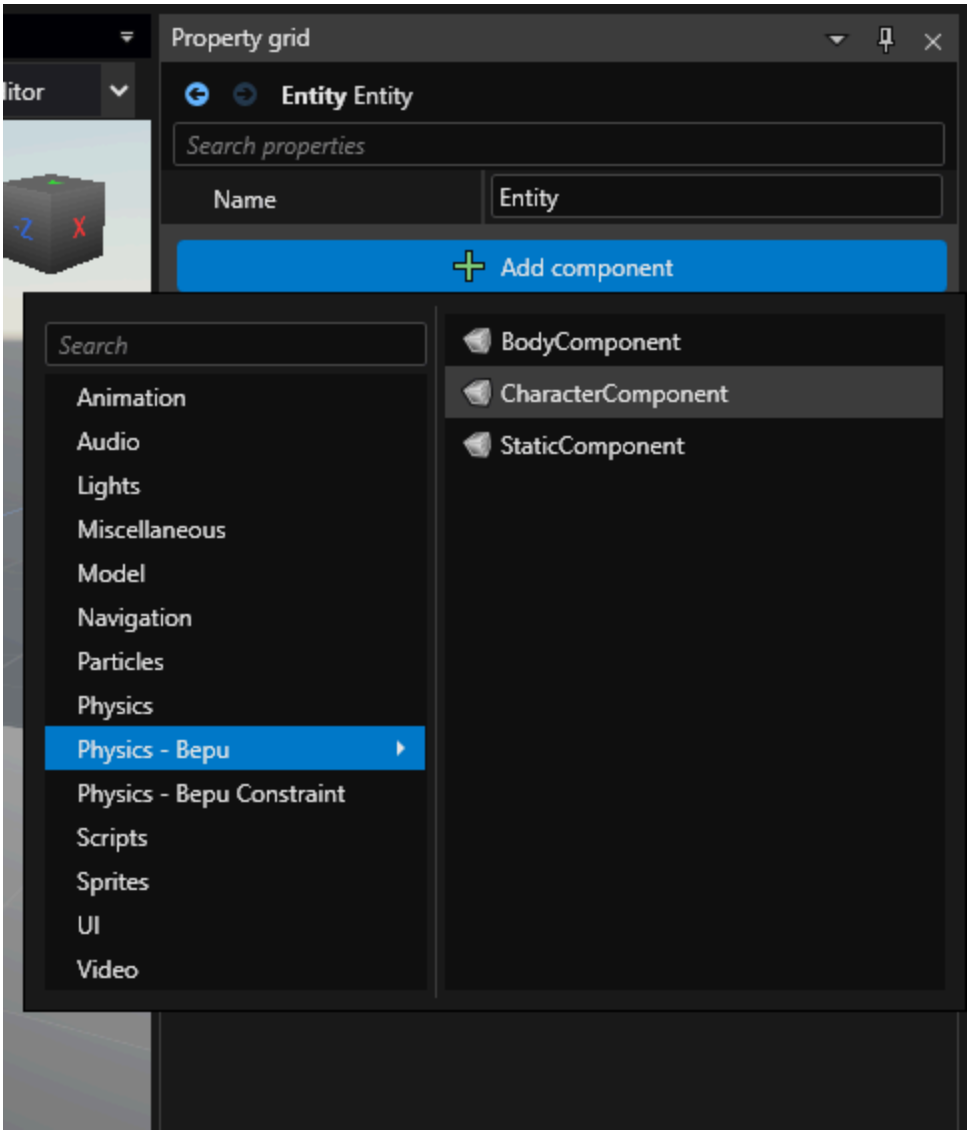
Character

Beginner Designer

Characters are used for player and NPC movement. Entities with character components can be moved with Move, TryJump, and teleported by setting its Position property.

Add a Character Component to an Entity

1. In the **Scene Editor**, select the entity you want to add the component to.
2. In the **Property Grid**, click `Add component`, hover `Physics - Bepu` and select `CharacterComponent`.



> **NOTE**
>
> You will need to set a collider for this newly created character, you can do so through the Collider property. The capsule shape is appropriate for most character colliders. For more information, see collider shapes.

> **WARNING**
>
> Never use mesh colliders for characters, use them only for statics, they are far too slow to be used as characters. If you absolutely need a more complex shape than the primitive ones, use a convex hull instead, see collider shapes.

Have a look at the API for more detail on the properties of this component.

Custom Character Controllers

> **IMPORTANT**
>
> The `CharacterComponent`'s default `IsGrounded` handling temporarily overrides the `Gravity` flag: when the character is grounded and idle, `Gravity` is set to `false` to prevent sliding down slopes. If you need to disable or customize gravity (for flying characters or special movement modes), derive from `CharacterComponent` and adjust the update logic.

When creating a controller, it is recommended to create a new class inheriting from this component instead of using it as is, you can access its internal state and override it with your own logic to better fit your game.

```
using Stride.BepuPhysics;
using Stride.Core.Mathematics;

public class MyCharacterController : CharacterComponent
{
    public override void Move(Vector3 direction)
    {
        base.Move(direction);
    }

    public override void TryJump()
    {
```

```
            base.TryJump();
        }

        public override void SimulationUpdate(BepuSimulation sim, float simTimeStep)
        {
            base.SimulationUpdate(sim, simTimeStep);
        }

        public override void AfterSimulationUpdate(BepuSimulation sim, float simTimeStep)
        {
            base.AfterSimulationUpdate(sim, simTimeStep);
        }
    }
```

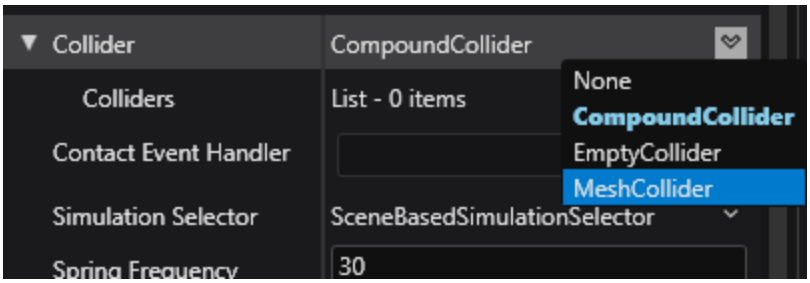
See also

- Statics
- Bodies
- Collider shapes

Collider Shapes

Beginner Designer

Each collidables should have a collider defining its shape. You can set them through the **Property Grid**.

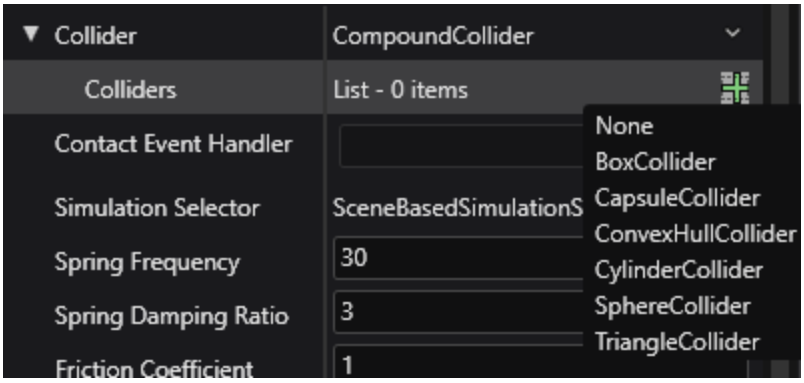


Empty

Empty do not collide with other objects, they are mostly used to anchor bodies with constraints.

Compound

A compound collider is a shape made up of a bunch of more primitive shapes, most of these are self-explanatory, while `Convex Hull` has a section describing it in more detail below.



Those individual primitives can intersect between each other, and don't necessarily have to match the model they are attached to. Each shape has additional properties including size, orientation, offset, and so on.

Meshes

Mesh colliders use 3D models as the collision shape itself. They are significantly slower than compounds, use them only when building a compound collider would be counter-productive.

> **WARNING**
>
> Never use mesh colliders for body collidables, use them only for statics, they are far too slow to be used as bodies. If you absolutely need a more complex shape than the primitive ones, use a convex hull instead.

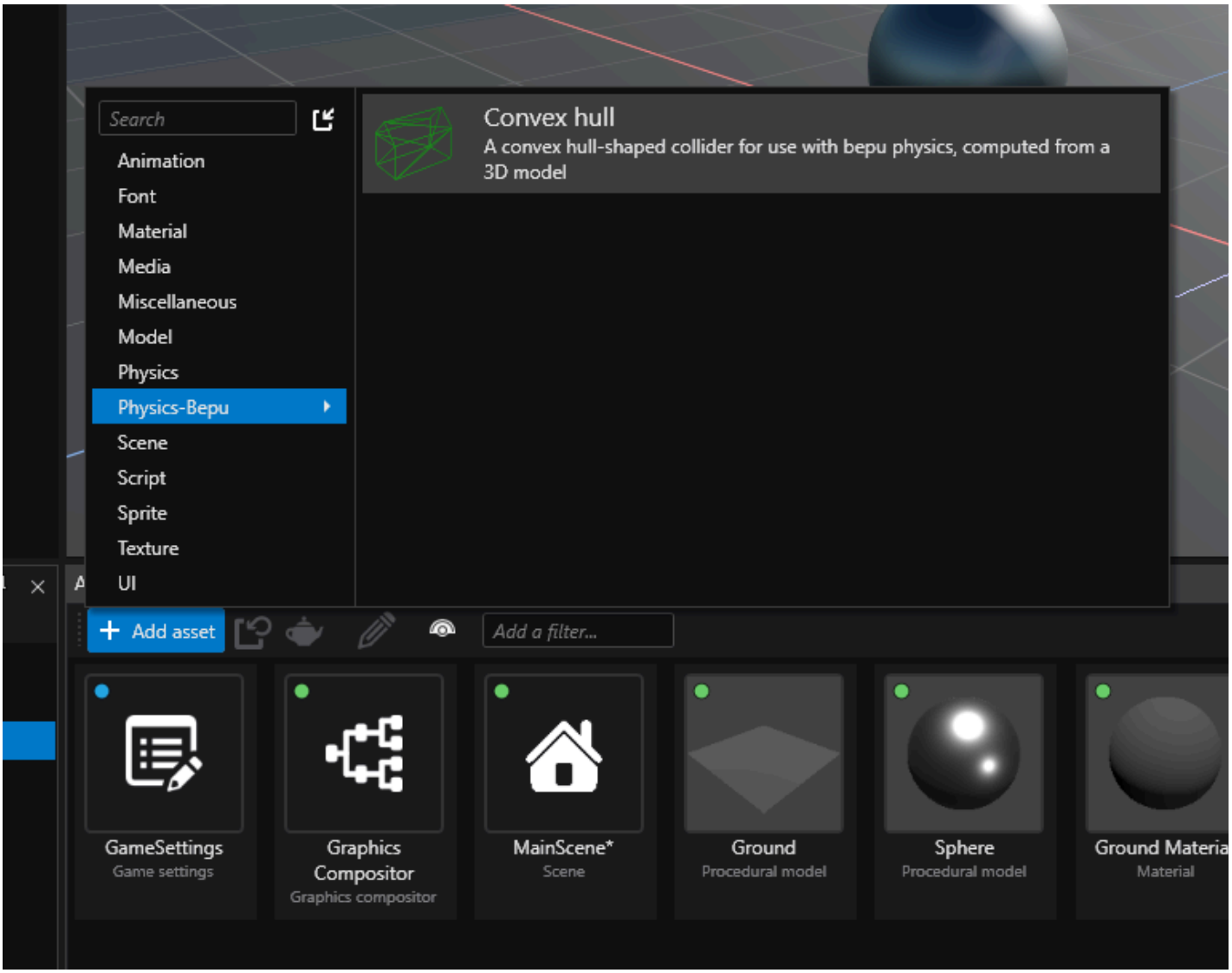
Convex Hulls

A convex hull is a convex shape that envelopes another. For example, the convex hull of the Eiffel Tower would be a pyramid large enough to contain the entire tower without any bits poking through.

Convex shapes are easier to test for collision, simulate and find intersections with, reducing the compute load physics engine have to deal with compared to their mesh counterpart.

Creating a Convex Hull

1. In the Asset View pane, press the Add asset button, hover on the Physics-Bepu option and select Convex hull

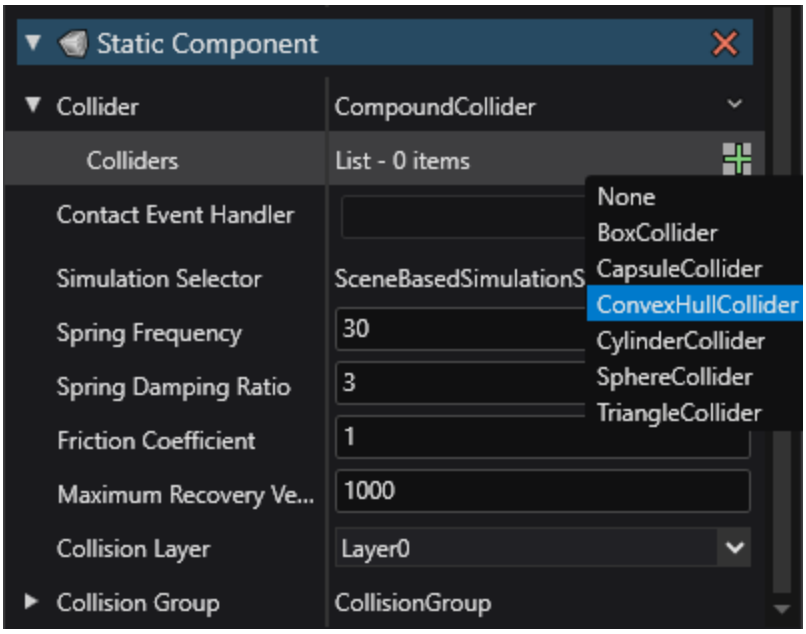


A new window will open prompting you to select a model asset, select the asset you want to create this hull from and press Ok.

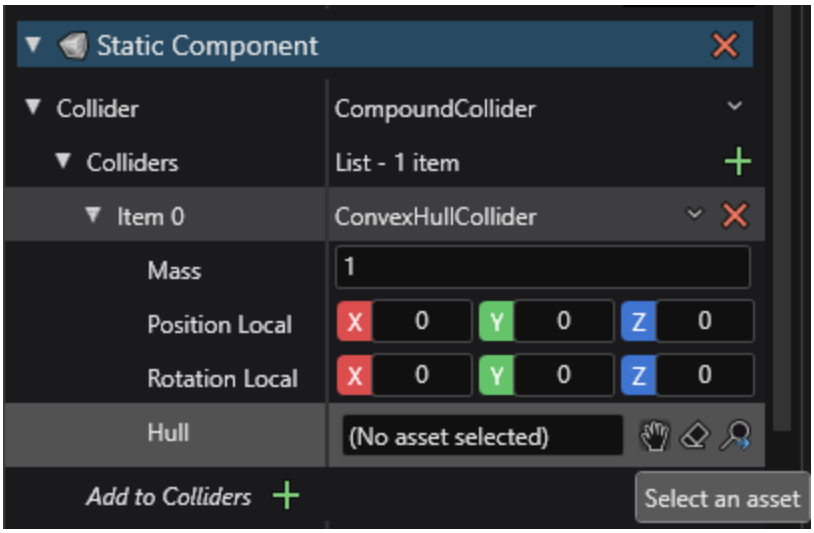
You can now add this new hull to one of your collidable.

2. Select the entity you want to add this Convex Hull to, add in a collidable component as is described in the static or body section.

3. Next to `Colliders`, click (**Add**) and select `ConvexHullCollider`



4. Set the `Hull` property to your newly created hull by pressing on the hand icon



Performance Consideration

The following are relevant excerpts from Bepu's documentation

Use simple shapes whenever possible. Spheres and capsules are fastest followed by boxes, cylinders, and finally convex hulls. While cylinders and convex hulls are not slow in an absolute sense, they can be an order of magnitude slower than spheres and capsules.

While you shouldn't be too afraid of cylinders and convex hulls (they're still pretty fast), it's hard to beat the simpler shapes.

If you need to use a convex hull, use the minimum number of vertices needed to approximate the shape. The cost of hull collision detection is proportional to their complexity.

If you *really*, *definitely* need a mobile mesh, especially one that needs to collide with other meshes, spend a while confirming that you *really*, **definitely**, ***seriously*** need it and there is no other option, and

then use a compound of simple shapes instead.

Okay, so maybe you actually truly really seriously need an actual mobile mesh. Keep the number of triangles to the minimum necessary to approximate the desired shape, and try to keep the triangles fairly uniform in size. Long sliver-like triangles can end up with large and inefficient bounding boxes. Static meshes follow the same optimization guidelines. Don't be surprised when you run into behavioral issues associated with infinitely thin one-sided triangles not colliding with each other and relatively crappy performance.

See also

- Colliders
- Tutorial: Create a bouncing ball
- Tutorial: Script a trigger

Triggers

Beginner Designer

A trigger is a collision shape which detects when colliders enter it, it can be used to run an event when a player character enters a room for example.

Triggers in Stride's Bepu implementation fall into what is known as a special kind of Contact Event Handler.

The contact event handler is a class that receives collision data whenever the object it is associated with collides with the world.

If the contact event handler you bind to an object is set to `NoContactResponse`, it is a Trigger, the object will never prevent anything from passing through it, it will only receive collision events.

Here's a basic example of a component which acts as a trigger to display a message in the console:

```
using Stride.BepuPhysics;
using Stride.BepuPhysics.Definitions.Contacts;
using Stride.Engine;

public class Test : StartupScript, IContactHandler
{
    public bool NoContactResponse => true;

    void IContactHandler.OnStartedTouching<TManifold>(Contacts<TManifold> contacts)
    {
        Log.Warning("Entered!");
    }

    void IContactHandler.OnStoppedTouching<TManifold>(Contacts<TManifold> contacts)
    {
        Log.Warning("Exited!");
    }
}
```

For an example of how to use triggers, see the Script a trigger tutorial.

See also

- Tutorial: Script a trigger
- Collidables
- Collider shapes

Constraints

> **WARNING**
> This page is WIP

> **WARNING**
> This documentation is under construction.

Advanced Programmer

Constraints restrict bodies to certain movement patterns. For example, a realistic knee joint can only move along one axis and can't bend forwards.

Constraints can either link two collidables together. They allow for interaction and dependency between bodies.

Performance And Stability

The following are relevant excerpts from Bepu's documentation

Try using the minimum number of solver iterations sufficient to retain stability. The cost of the solver stage is linear with the number of iterations, and some simulations can get by with very few.

For some simulations with very complex constraint configurations, there may be no practical number of solver iterations that can stabilize the simulation. In these cases, you may need to instead use substepping or a shorter time step duration for the entire simulation. More frequent solver execution can massively improve simulation quality, allowing you to drop velocity iteration counts massively (even to just 1 per substep). See the Substepping documentation for more details.

> **WARNING**
> Probably need to adapt and extend some parts of the above to our implementation, not sure how relevant everything is. Take stuff from https://github.com/bepu/bepuphysics2/blob/master/Documentation/StabilityTips.md and https://github.com/bepu/bepuphysics2/blob/master/Documentation/Substepping.md
>
> - Eideren

See also

- Colliders

Physics Queries

Intermediate Programmer

Physics Queries are a set of operation to test and retrieve physics object in a given simulation, the most well known form of those queries is the `RayCast` which traces an invisible line through the scene to find intersecting Collidables. This is useful, for example, to check which objects are in a gun's line of fire, or are under the mouse cursor when the user clicks.

> **NOTE**
>
> Physics queries uses **collider shapes** to find objects. It ignores entities that have no Collidable Component attached or no collider shape set, see Collidables.

To raycast in the current Simulation, see the Simulation API.

For an example of raycasting, see the **Physics Sample** project included with Stride.

Examples

On left click, if there is a 'HostileUnit' in front of 'MyGun', deal damage to it

```
public override void Update()
{
    var simulation = Entity.GetSimulation();
    MyGun.Transform.GetWorldTransformation(out var rayStart, out var worldRot, out _);
    var rayDir = worldRot * Vector3.UnitZ;
    float range = 10;
    // All layers except 0 & 1
    var layers = CollisionMask.Everything & ~(CollisionMask.Layer0 | CollisionMask.Layer1);

    if (Input.IsMouseButtonPressed(MouseButton.Left) && simulation.RayCast(rayStart, rayDir,
range, out HitInfo hitResult, layers))
    {
        var hostile = hitResult.Collidable.Entity.Get<HostileUnit>();
        if (hostile is not null)
            hostile.DealDamage(10);
    }
}
```

Using the non-alloc version of a penetrating query to find multiple objects intersecting with the ray

```
var simulation = Entity.GetSimulation();
Entity.Transform.GetWorldTransformation(out var rayStart, out var worldRot, out _);
var rayDir = worldRot * Vector3.UnitZ;

// Allocates a working buffer on the stack for the penetrating test,
// The 16 specified here would be the maximum item this test would keep track of,
// if the test finds more than 16 objects, it will yield the 16 closest ones
System.Span<HitInfoStack> buffer = stackalloc HitInfoStack[16];

// The order in which elements are yielded throughout this foreach is not guaranteed
// Some hitInfo with a smaller distance may be returned after one with a larger one,
// and the opposite is also true.
foreach (var hitInfo in simulation.RayCastPenetrating(rayStart, rayDir, 100f, buffer))
{
    Log.Warning($"T: {hitInfo.Distance}\nNormal: {hitInfo.Normal}\nEntity
: {hitInfo.Collidable.Entity}");
}
```

Moves a box through the scene and return what it collided with

```
var simulation = Entity.GetSimulation();
Entity.Transform.GetWorldTransformation(out var center, out var worldRot, out _);
var rayDir = worldRot * Vector3.UnitZ;
var initialPose = new RigidPose(center, worldRot);
var displacement = new BodyVelocity(rayDir, default);
var shape = new Box(0.25f, 0.25f, 0.25f);

if (simulation.SweepCast(shape, initialPose, displacement, 10, out HitInfo hitInfo))
{
    Log.Warning($"T: {hitInfo.Distance}\nNormal: {hitInfo.Normal}\nEntity
: {hitInfo.Collidable.Entity}");
}
```

Find up to 16 physics object a box placed at that position would overlap with

```
var simulation = Entity.GetSimulation();
Entity.Transform.GetWorldTransformation(out var center, out var worldRot, out _);
var initialPose = new RigidPose(center, worldRot);
var shape = new Box(0.25f, 0.25f, 0.25f);

System.Span<CollidableStack> buffer = stackalloc CollidableStack[16];
foreach (var hit in simulation.Overlap(shape, initialPose, buffer))
{
```

```
    Log.Warning($"Entity : {hit.Entity}");
}
```

Send a raycast from the mouse's screen position:

```
public static bool ScreenPositionToWorldPositionRaycast(Vector2 screenPos,
CameraComponent camera)
{
    Matrix invViewProj = Matrix.Invert(camera.ViewProjectionMatrix);

    // Reconstruct the projection-space position in the (-1, +1) range.
    //    Don't forget that Y is down in screen coordinates, but up in projection space
    Vector3 sPos;
    sPos.X = screenPos.X * 2f - 1f;
    sPos.Y = 1f - screenPos.Y * 2f;

    // Compute the near (start) point for the raycast
    // It's assumed to have the same projection space (x,y) coordinates and z = 0 (lying on
the near plane)
    // We need to unproject it to world space
    sPos.Z = 0f;
    var vectorNear = Vector3.Transform(sPos, invViewProj);
    vectorNear /= vectorNear.W;

    // Compute the far (end) point for the raycast
    // It's assumed to have the same projection space (x,y) coordinates and z = 1 (lying on
the far plane)
    // We need to unproject it to world space
    sPos.Z = 1f;
    var vectorFar = Vector3.Transform(sPos, invViewProj);
    vectorFar /= vectorFar.W;

    var delta = (vectorFar - vectorNear).XYZ();
    var maxDistance = delta.Length();
    var dir = delta / maxDistance; // normalize delta

    // Raycast from the point on the near plane to the point on the far plane and get the
collision result
    return camera.Entity.GetSimulation().RayCast(vectorNear.XYZ(), dir, maxDistance, out
HitInfo hit);
}
```

> **NOTE**
>
> There are multiple ways to retrieve a reference to this `Simulation` from inside one of your `ScriptComponent`:
>
> - The recommended way is through a reference to a physics component, something like `myBody.Simulation` as it is the fastest.
> - Or through the `Entity.GetSimulation()` extension method.

See also

- Colliders

Physics Update

Beginner Programmer

When you need your logic to interact, query or otherwise modify the state of a simulation, you may need to do so right before or after physics updates. You can implement the `ISimulationUpdate` interface in your EntityComponent or Scripts to be notified of when simulation updates.

```
using System;
using Stride.BepuPhysics;
using Stride.BepuPhysics.Components;
using Stride.Engine;

public class MyPhysicsComponent : SyncScript, ISimulationUpdate
{
    public override void Update()
    {
        // Here you would call stuff you need to do every frame
    }

    public void SimulationUpdate(BepuSimulation simulation, float simTimeStep)
    {
        // Here is for stuff you need to do right before physics runs,
        // sample player input, setting body velocities, etc.
    }

    public void AfterSimulationUpdate(BepuSimulation simulation, float simTimeStep)
    {
        // Here is for stuff you need to check right after physics ran,
        // check if an object collided with anything, if it's on the ground,
        // if it failed to increase in height when jumping, etc.
    }
}
```

See also

- Character
- Physics Jitter
- Body
- Index

Tutorials

- Create a bouncing ball: Use the static and body components to create a ball bouncing on a floor.
- Script a trigger: Create a trigger that doubles the size of a ball when the ball passes through it.

See also

- Index

Create a bouncing ball

Beginner Designer

In this tutorial, we'll use the static and body components to create a ball bouncing on a floor.

1. Create a new project

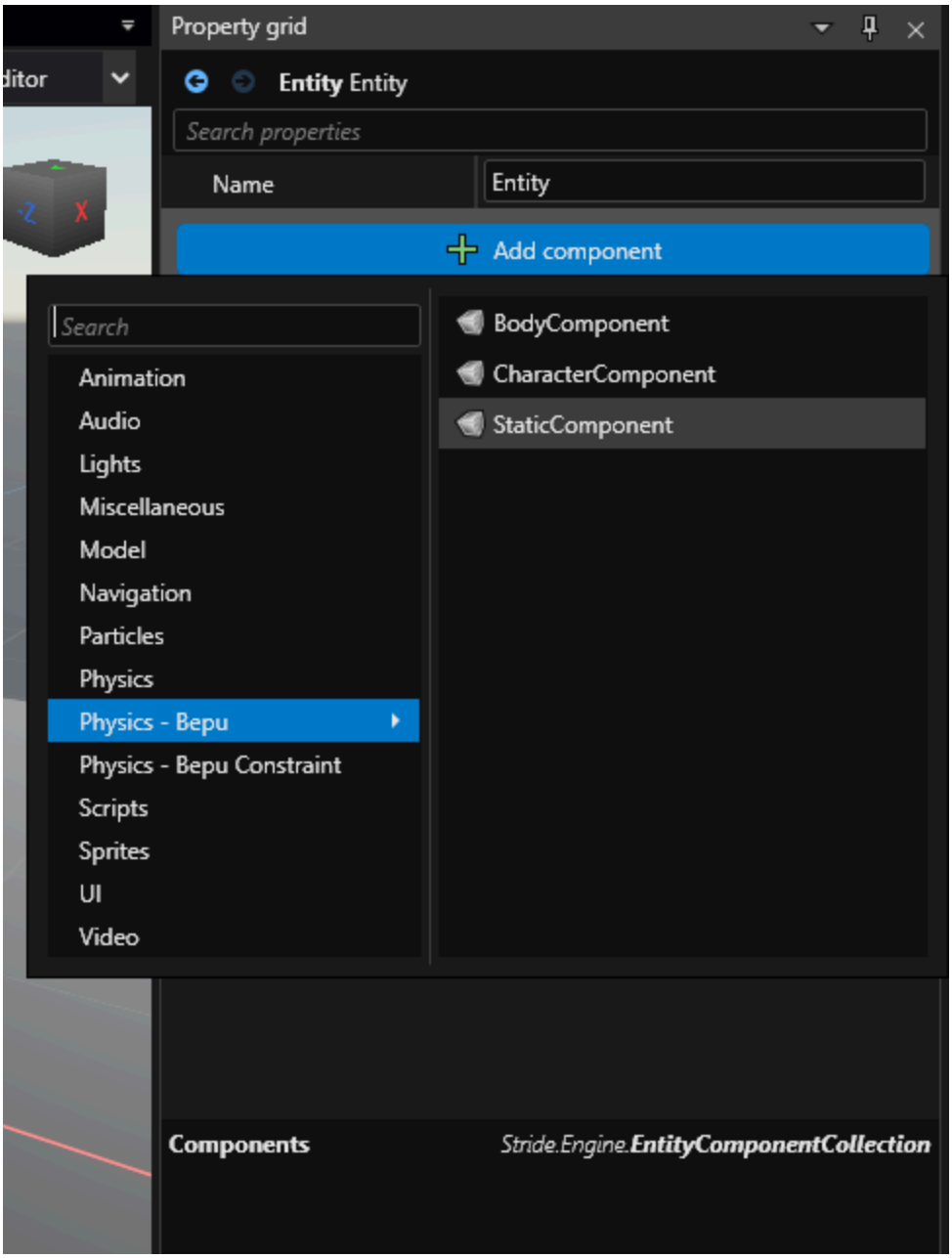
Start a **New Game** project.

The default scene comes pre-loaded with five entities: Camera, Directional light, Skybox, Ground, and Sphere. We're going to add physics components to the **Ground** and **Sphere** entities.

2. Add a Static Component

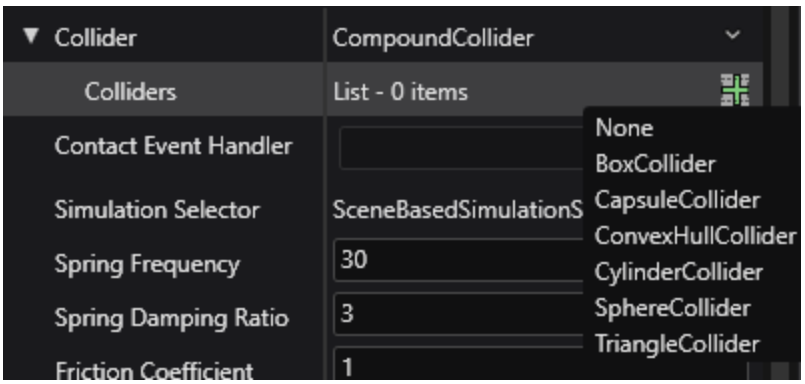
Let's begin by adding a static collider component to the Ground entity. A static component is a physics object that doesn't move. Typical static component are walls, floors, large rocks, and so on. In this case, the static component will give the ball something to bounce on.

1. Select the **Ground** entity.
2. In the **Property Grid**, click **Add component** and select **Static Collider**.



3. Set the collider shape to match the shape of the entity. To do this, in the **Property Grid**, expand the **Static Component** to view its properties.

4. Next to **Colliders**, click (**Add**) and select **Box Collider**.



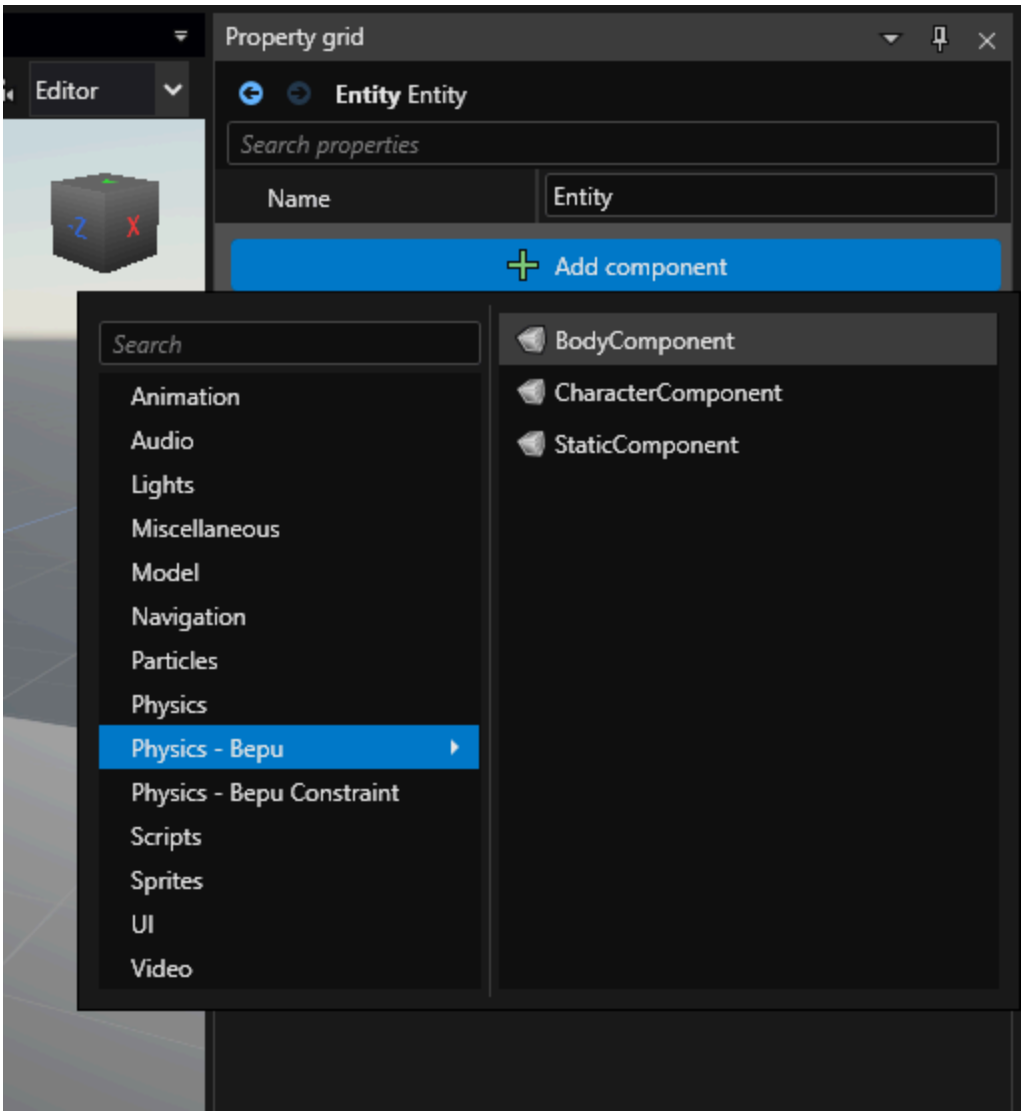
This adds a static collider to the ground, so the ball has something to bounce off.

5. Set the `Size` property to `X: 10, Y: 0.1, Z: 10` to create a flat plane from this box

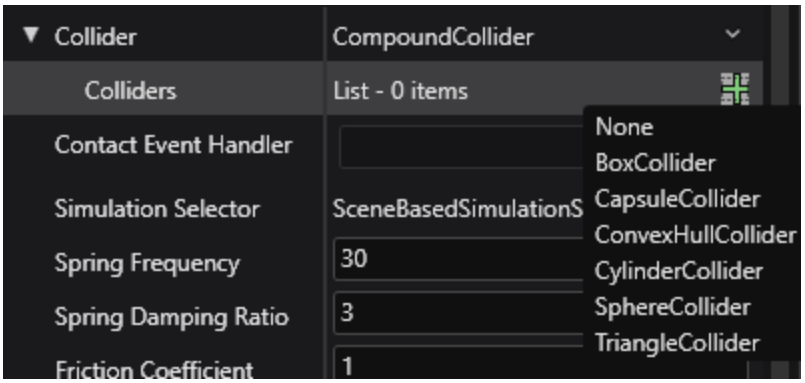
3. Add a Body Component

Next, we'll add a body component to the sphere. A body is a physics object that moves — perfect for our bouncing ball.

1. In the **Scene Editor**, select the **Sphere** entity.
2. In the **Property Grid**, click **Add component** and select **Rigidbody**.



3. Just like we did for the Ground entity, set the collider shape to match the entity. To do this, in the **Property Grid**, expand the **Body Component** to view its properties.
4. Next to **Colliders**, click (**Add**) and select **Sphere**.



4. Position the ball

Let's position the sphere so it starts in midair and falls to the ground.

1. Select the **Sphere** entity.
2. In the **Property Grid**, under **Transform**, set the **Position** to: `X: 0, Y: 6, Z: 0`

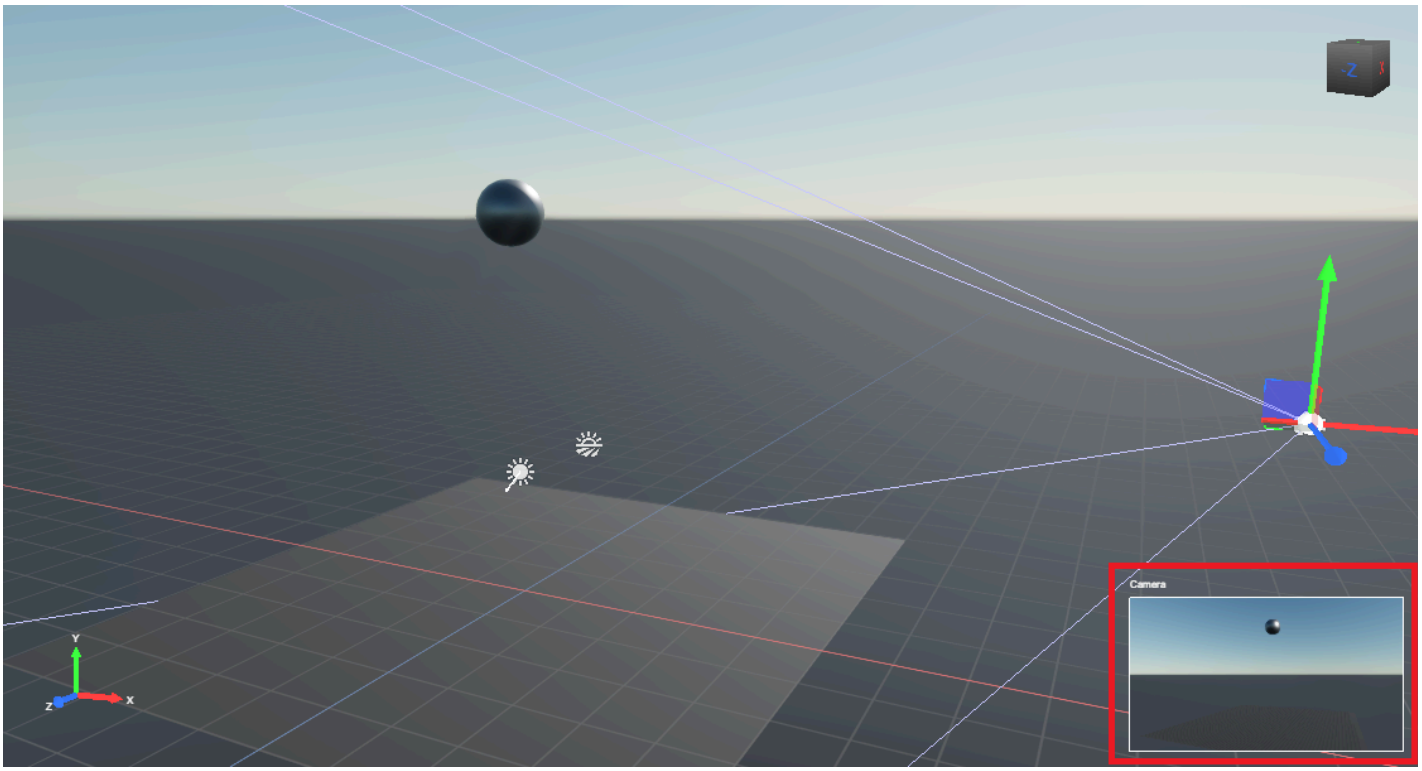
   This places the ball in midair above the ground.

5. Position the camera

Now we'll move the camera to give us a good view of the scene.

1. Select the **Camera** entity.
2. In the **Property Grid**, under **Transform**, set the **Position** to: `X: 11, Y: 3, Z: -4`

   You can see preview the camera view in the **Camera preview** in the bottom-right of the Scene Editor.



6. Set the restitution

Let's see what the scene looks like so far. To run the project, press **F5**.

The Sphere (body) responds to gravity and falls. The Ground (static collider) breaks its fall. But there's no bounce effect yet.

To create a bounce effect, we need to change the `Spring Frequency` and `Spring Daming Ratio` of the Sphere.

> **NOTE**
>
> Bepu internally uses speculative contacts which does not play well with traditional coefficient of restitution. Instead, bounces can be implemented through contact constraint springiness, this specificity mean that bounces actually take place over multiple simulation steps instead of on contact. This may be counter-intuitive for users accustomed to other engine's restitution-based bounciness.

- Bounciness is dominated by `Spring Damping Ratio`; setting it to zero minimizes energy loss on impact.
- Increasing `Spring Frequency` can make impacts less bouncy when. This happens because the integration rate becomes too slow to represent the motion, and it gets damped away. Increasing the substepping rate or using more timesteps preserves bounciness with higher frequencies.

Let's set the `Spring Frequency` and `Spring Damping Ratio` of the Sphere.

1. Select the **Sphere** entity.
2. In the **Property Grid**, under **Body**, set the `Spring Frequency` to 3 and `Spring Damping Ratio` to 0.

To see how this changes the physics, run the project again (**F5**). This time, the ball bounces on the ground before coming to a stop:

Now that we've created a bouncing ball, we can use it to learn to Script a trigger.

See also

- Colliders
- Collider shape
- Tutorial: Script a trigger

Tutorial: Script a trigger

Beginner Designer

In this tutorial, we'll create a trigger that doubles the size of a ball when the ball passes through it.

> **NOTE**
> The screenshots and videos in this tutorial were made using an earlier version of Stride, so some parts of the UI, and the default skybox and sphere, might look different from your version.

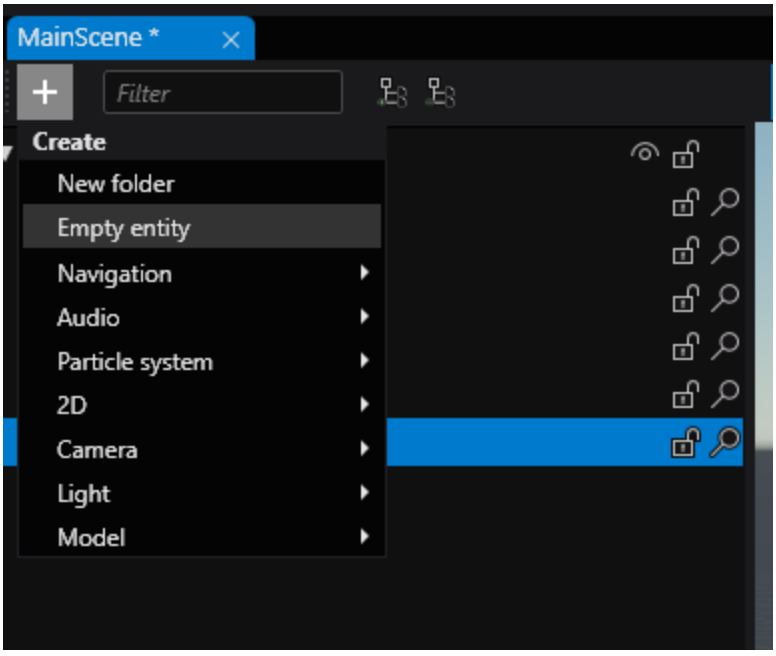
1. Create a bouncing ball

Follow the instructions in the Create a bouncing ball tutorial. This creates a simple scene in which a ball falls from midair, hits the ground, and bounces.

3. Add a trigger

Now we'll add a trigger between the ball and the ground, so the ball passes through it.

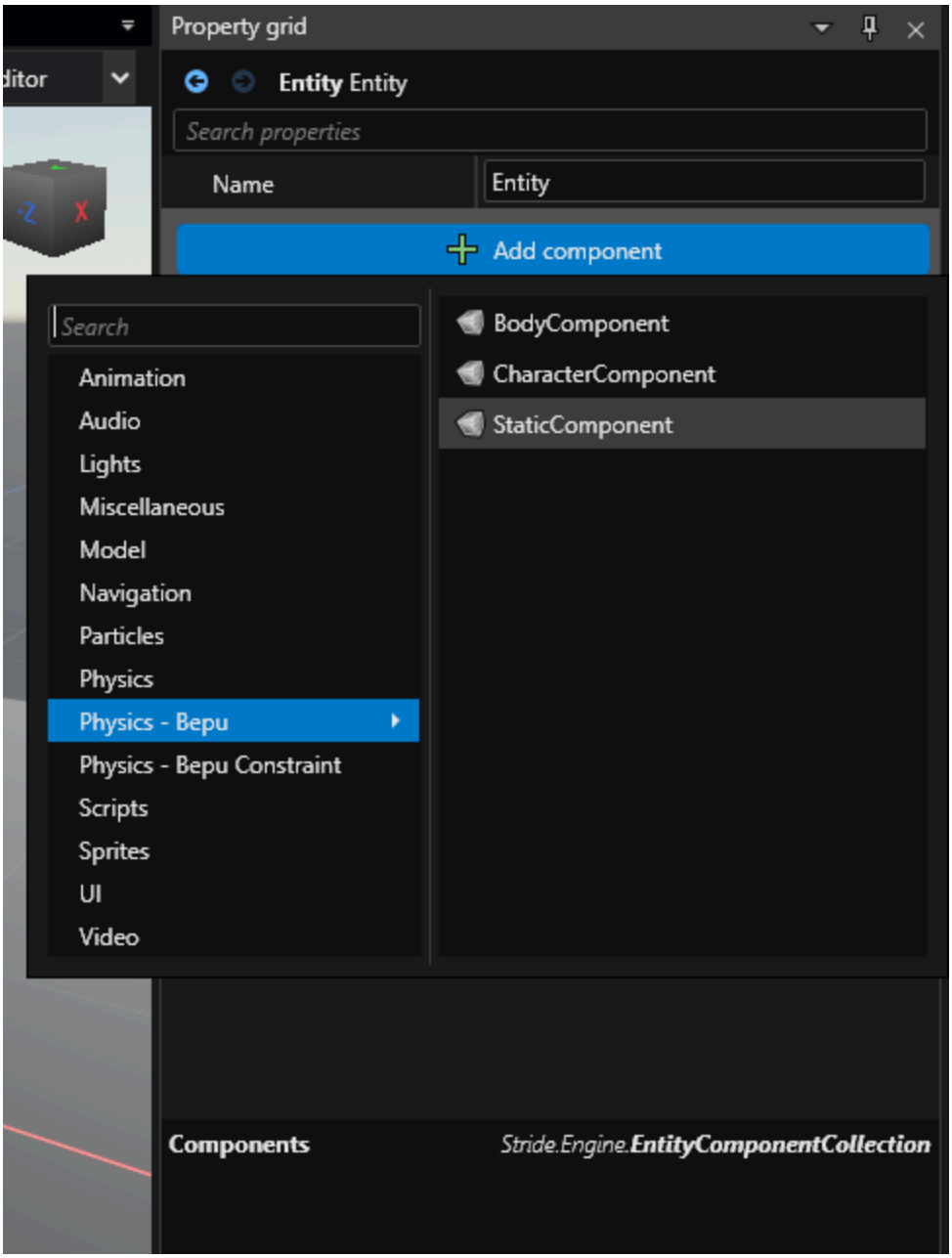
1. In the **Scene Editor**, click the white plus button (**Create new entity**) and select **Empty entity**.



   Game Studio adds an entity to the scene with the default name **Entity**.

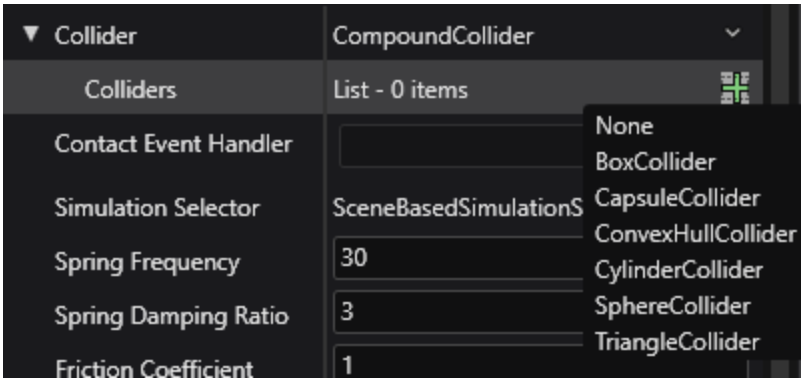
2. This entity will be our trigger, so rename it *Trigger* to make it easy to identify.

3. Since we don't need the trigger to move, we'll make it a Static Component. In the **Property Grid**, click **Add component** and select **Static component**.



4. In the **Property Grid**, expand the **Static Component** to view its properties.

5. We need to give the trigger a shape. Next to **Colliders**, click (**Add**) and select **Box**.

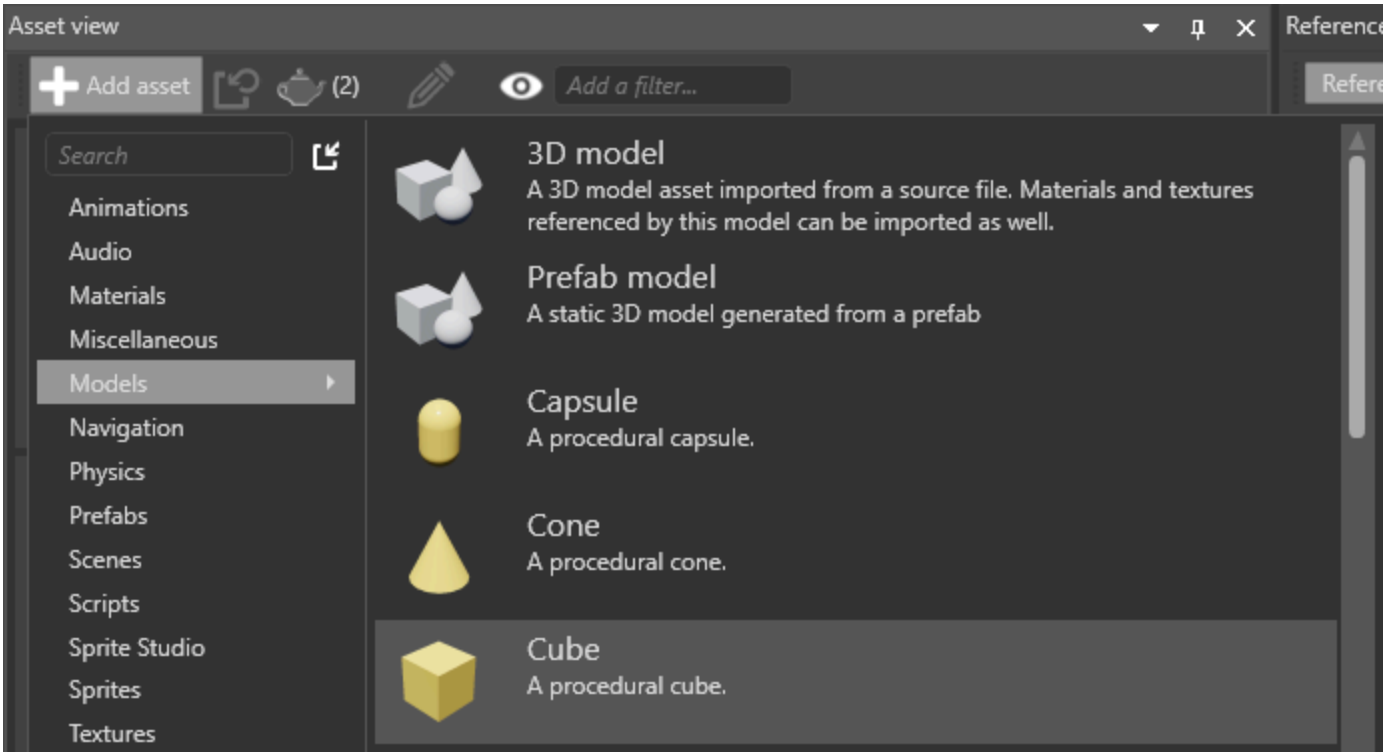


This gives the trigger a box shape.

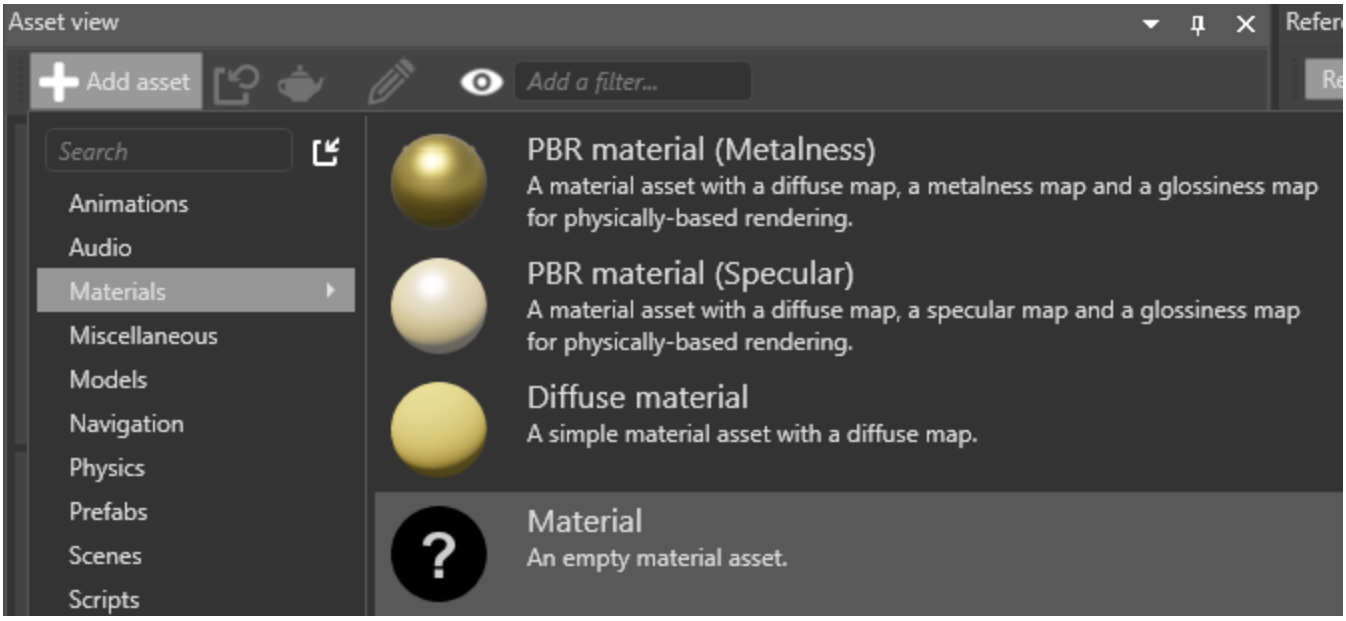
4. Give the trigger a model

Right now, the trigger is invisible at runtime. To better show how the trigger works, we'll make it a transparent box. This has no effect on how the trigger works; it just means we can easily see where it is at runtime.

1. Create a new procedural model asset. To do this, in the **Asset View**, click **Add asset**, and select **Models > Cube**.

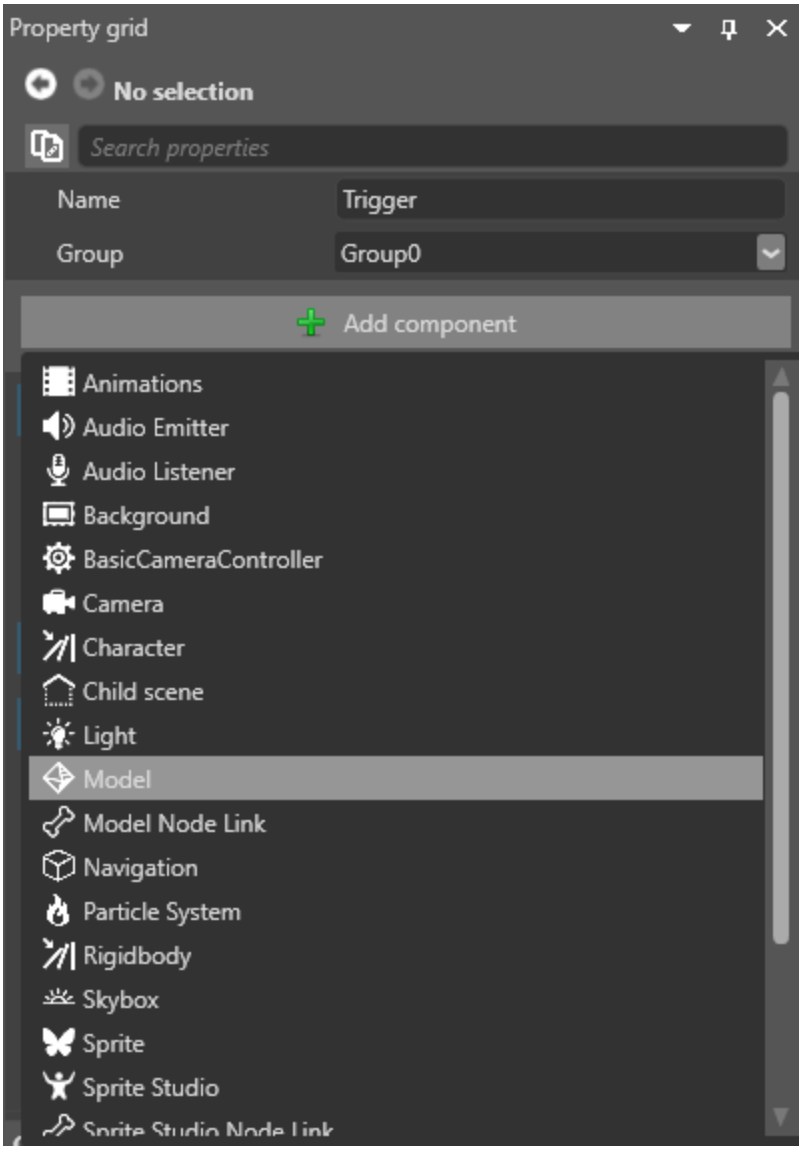


2. Create a new empty material asset. To do this, in the **Asset View**, click **Add asset**, and select **Materials > Material**.



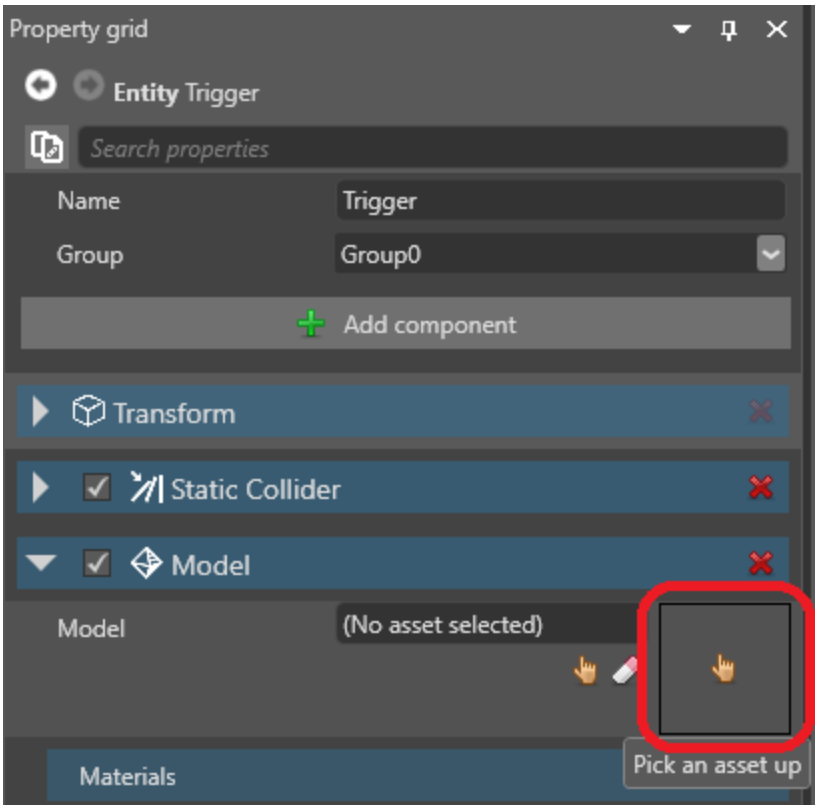
3. Let's rename the material to make it easy to identify. To do this, right-click, select **Rename**, and type a new name (eg *Transparent*).

4. Select the **Trigger** entity. In the **Property Grid**, click **Add component** and select **Model**.

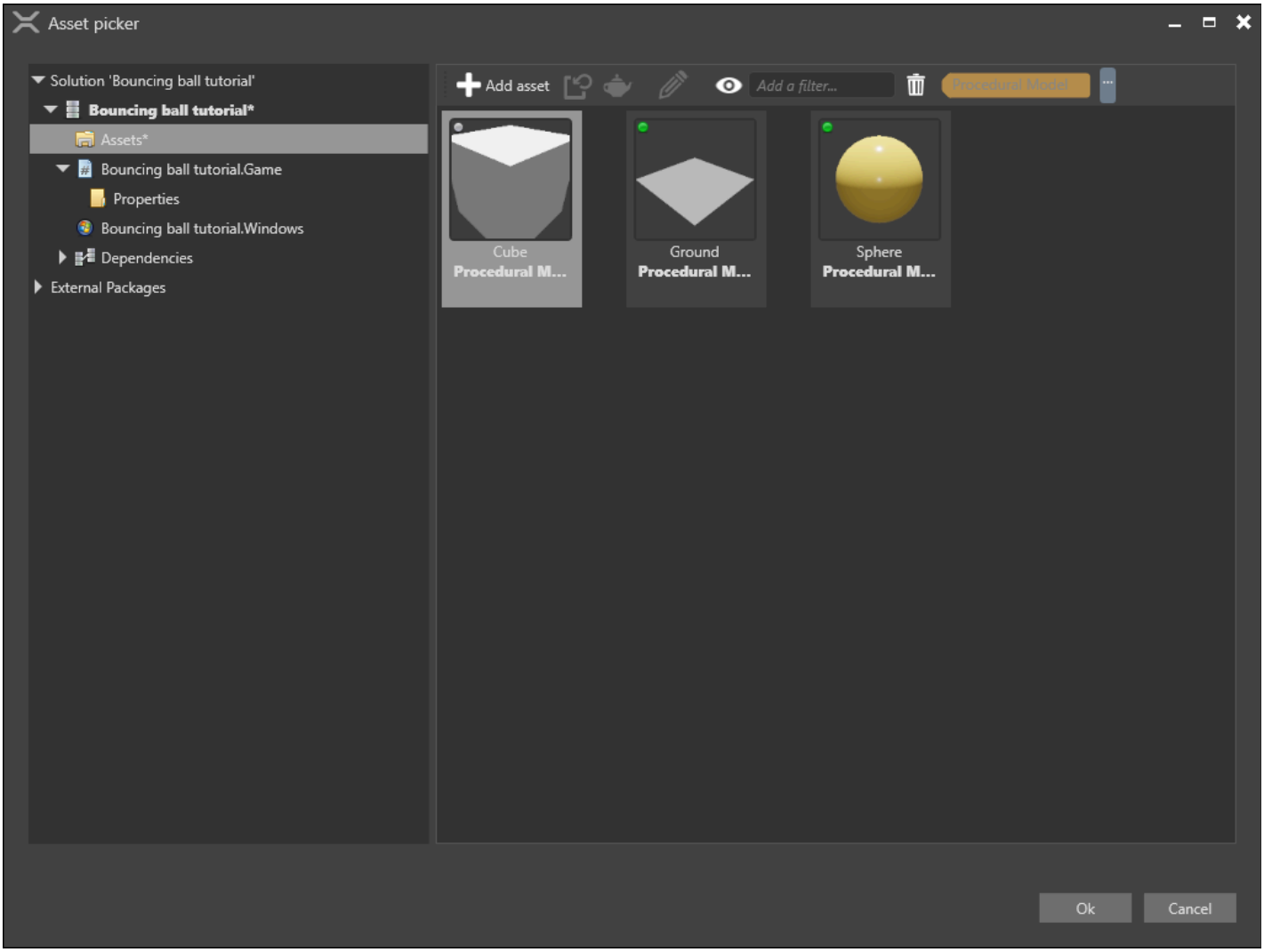


Game Studio adds a model component to the entity.

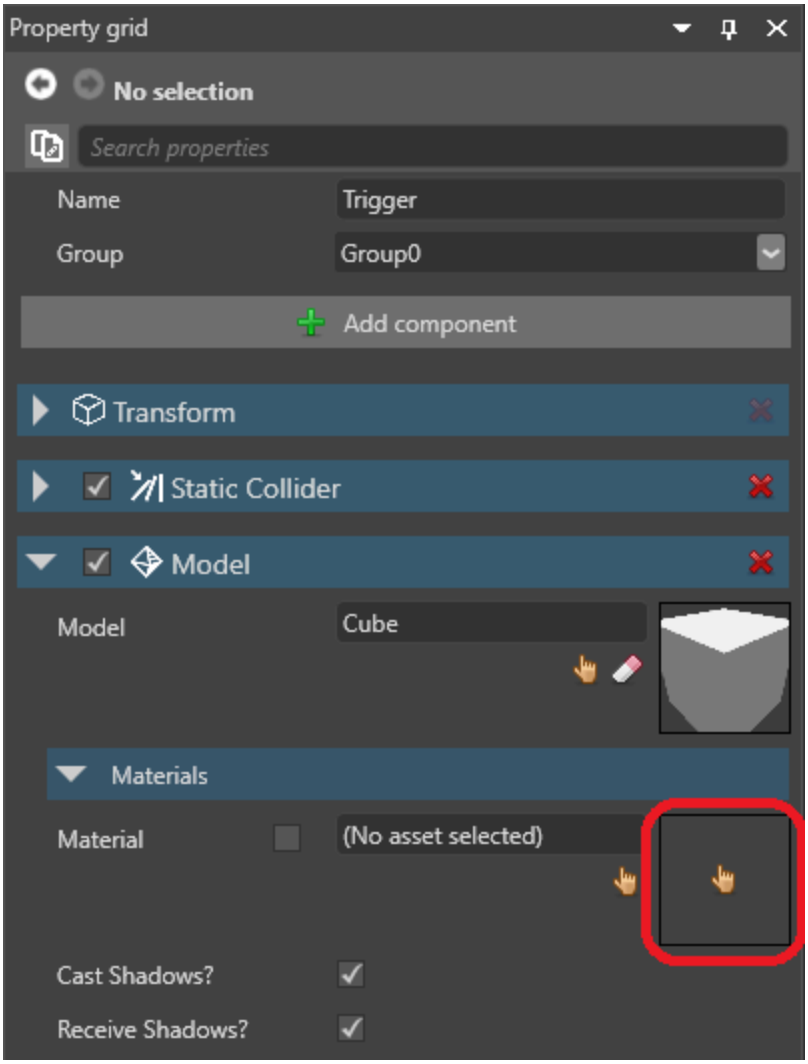
5. Under **Model**, click (**Select an asset**).



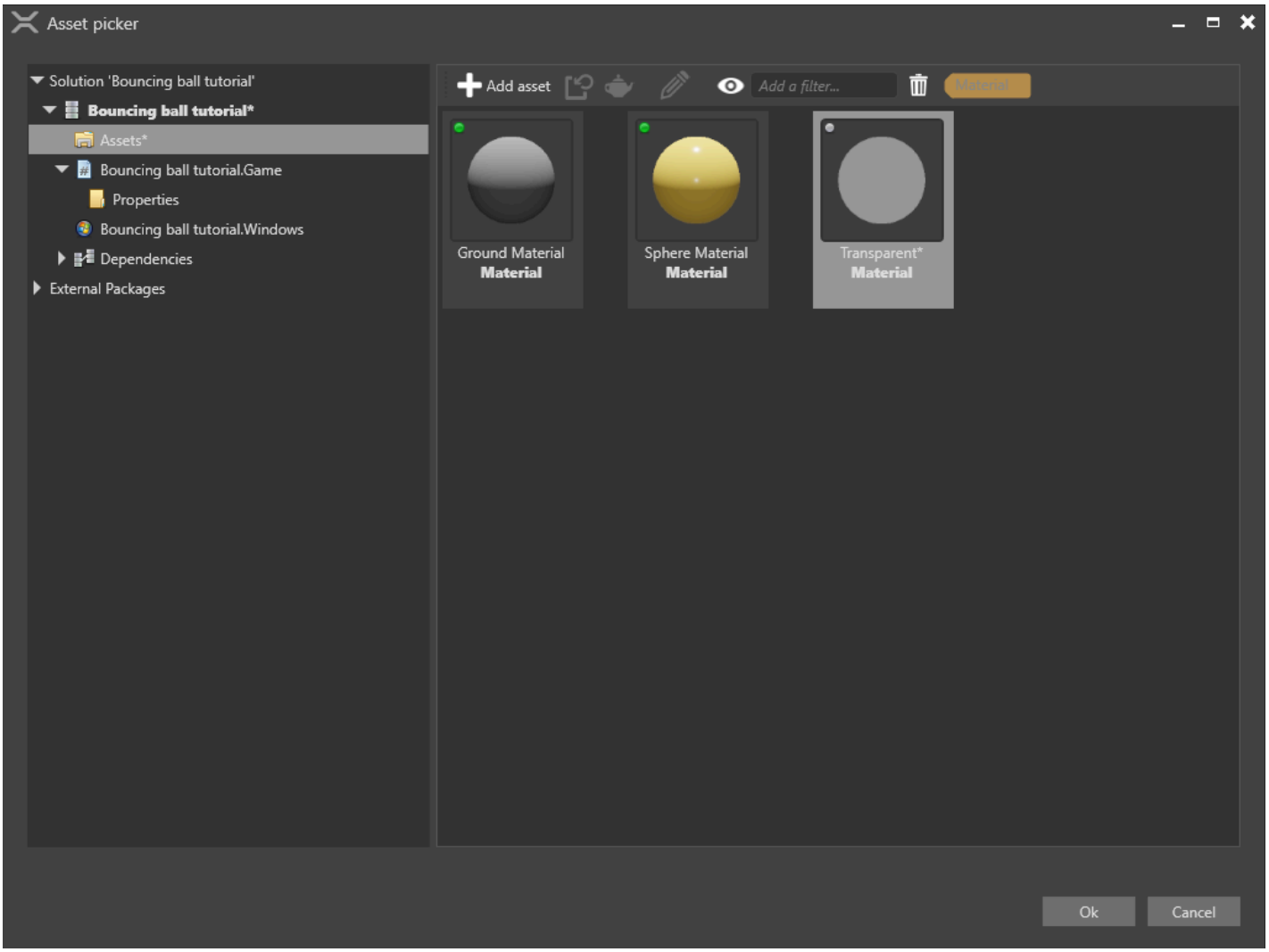
6. Select the **Cube** model we created in step 1 and click **OK**.



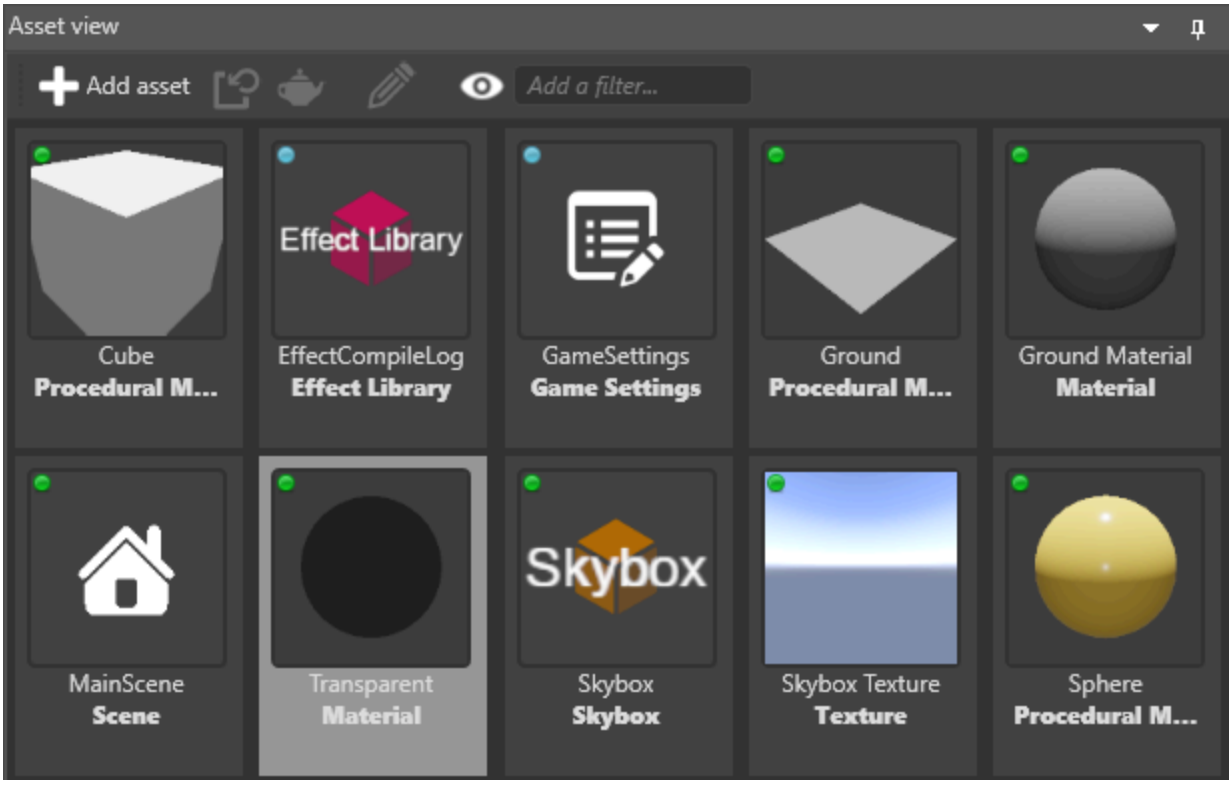
7. In the **Property Grid**, under **Model > Materials**, click (**Select an asset**).



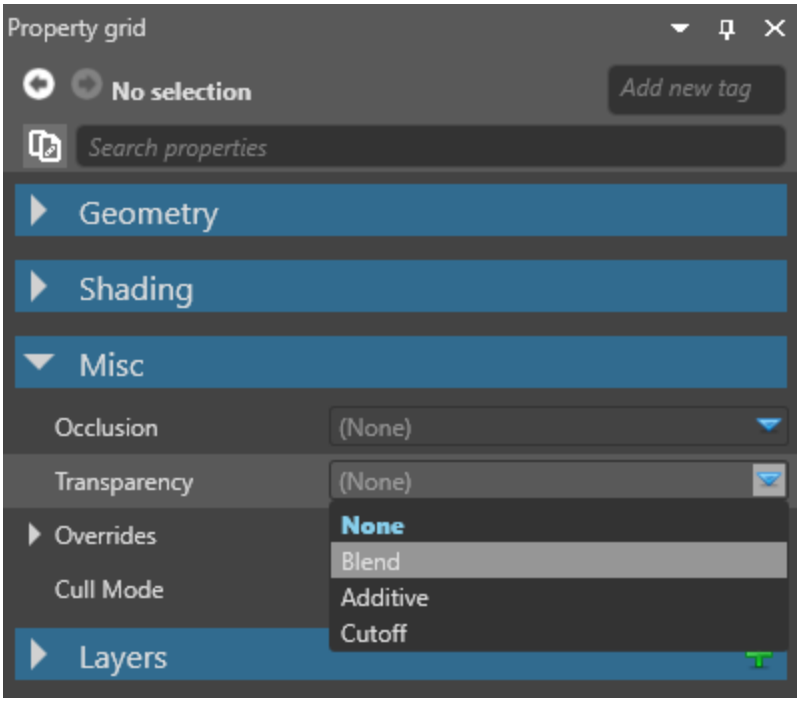
8. Select the **Transparent** material we created in step 2 and click **OK**.



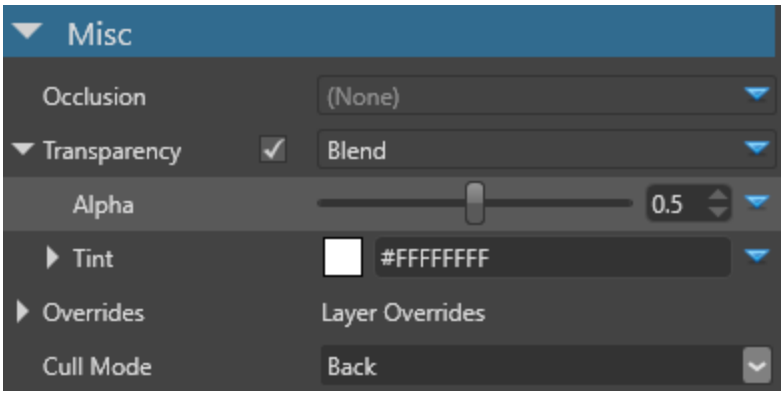
9. In the **Asset View**, select the **Transparent** material asset.



10. In the **Property Grid**, under **Misc > Transparency**, select **Blend**.



11. By default, the Alpha is set to 1. This makes the material completely opaque. To set it to 50% opacity, set the **Alpha** to 0.5.



Now the trigger area will be visible at runtime.

5. Position the trigger

We need to position the trigger between the ground and the sphere, so the ball falls through it.

In the **Property Grid**, under **Transform**, set the **Position** to: *X:0, Y:3, Z:0*

Now the trigger entity is between the ground and the sphere.

6. Change the sphere size with script

If we run the project now (**F5**), the ball bounces off the trigger, but nothing happens.

Let's write a script to change the size of the ball when it enters the trigger.

> **NOTE**
>
> For more information about scripts, see Scripts.

1. In the **Asset View**, click **Add asset** and select **Scripts** > **Sync Script**.
2. In the **Create a script** dialog, name your script *Trigger* and click **Create script**.

   2a. If Game Studio asks if you want to save your script, click **Save**.

   2b. If Game Studio asks if you want to reload the assemblies, click **Reload**.
3. Open the script, replace its content with the following code, and save the file:

```
using Stride.BepuPhysics;
using Stride.BepuPhysics.Definitions.Contacts;
using Stride.Core.Mathematics;
using Stride.Engine;

namespace TransformTrigger
{
    // Adding IContactHandler to listen to contact events
```

```
    public class Trigger : SyncScript, IContactHandler
    {
        public override void Start()
        {
            // Initialization of the script.
        }

        public override void Update()
        {
            // Do stuff every new frame
        }

        // Let objects pass through this trigger, false would make objects bounce off it
        public bool NoContactResponse => true;

        void IContactHandler.OnStartedTouching<TManifold>(Contacts<TManifold> contacts)
        {
            // When something enters inside this object
            contacts.Other.Entity.Transform.Scale = new Vector3(2.0f);
        }

        void IContactHandler.OnStoppedTouching<TManifold>(Contacts<TManifold> contacts)
        {
            // When something exits this object
            contacts.Other.Entity.Transform.Scale = new Vector3(1.0f);
        }
    }
}
```

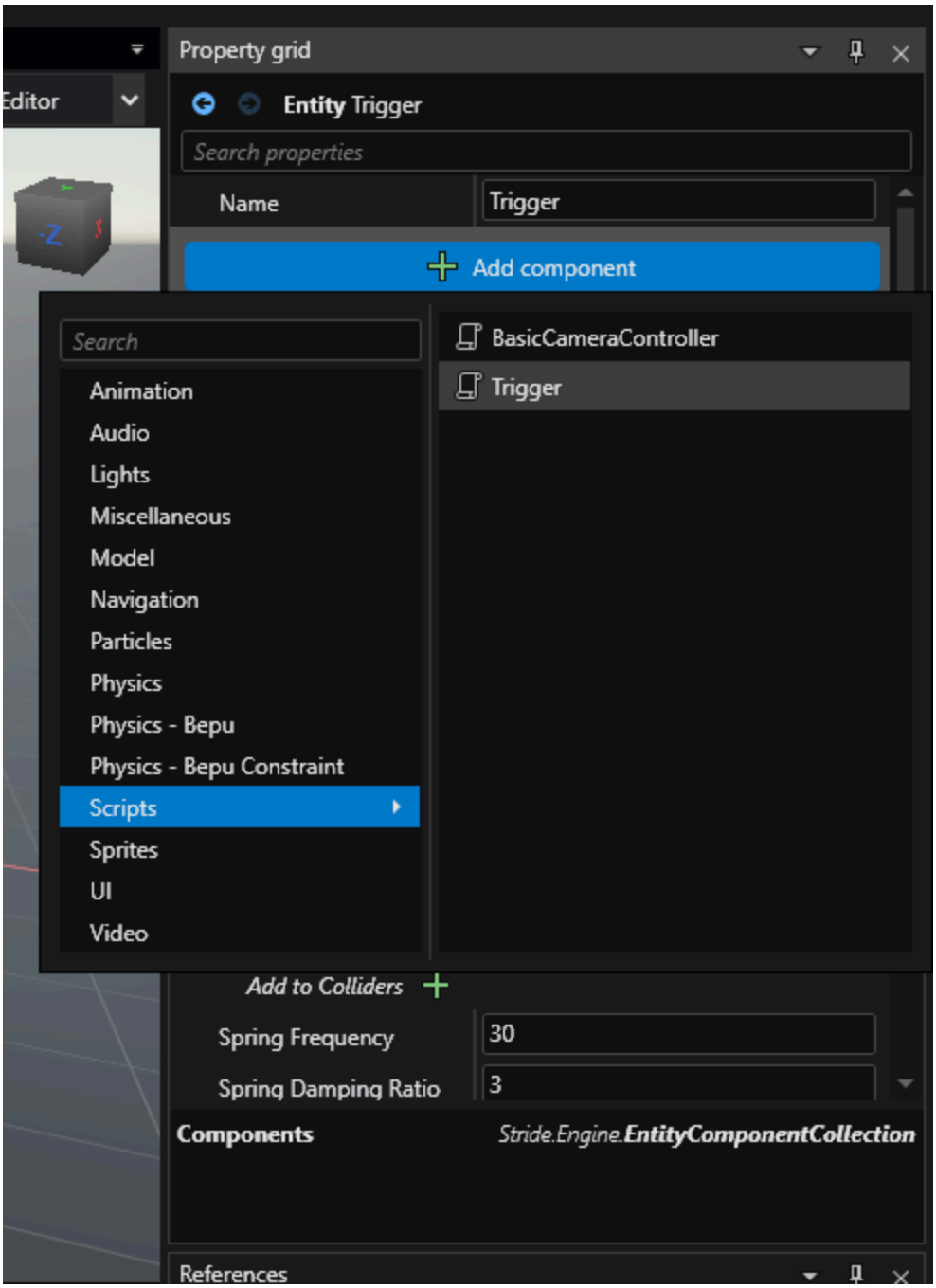
This code doubles the size (scale) of any entity that enters the trigger. When the entity exits the trigger, it is set to a unit size.

4. Reload the assemblies.

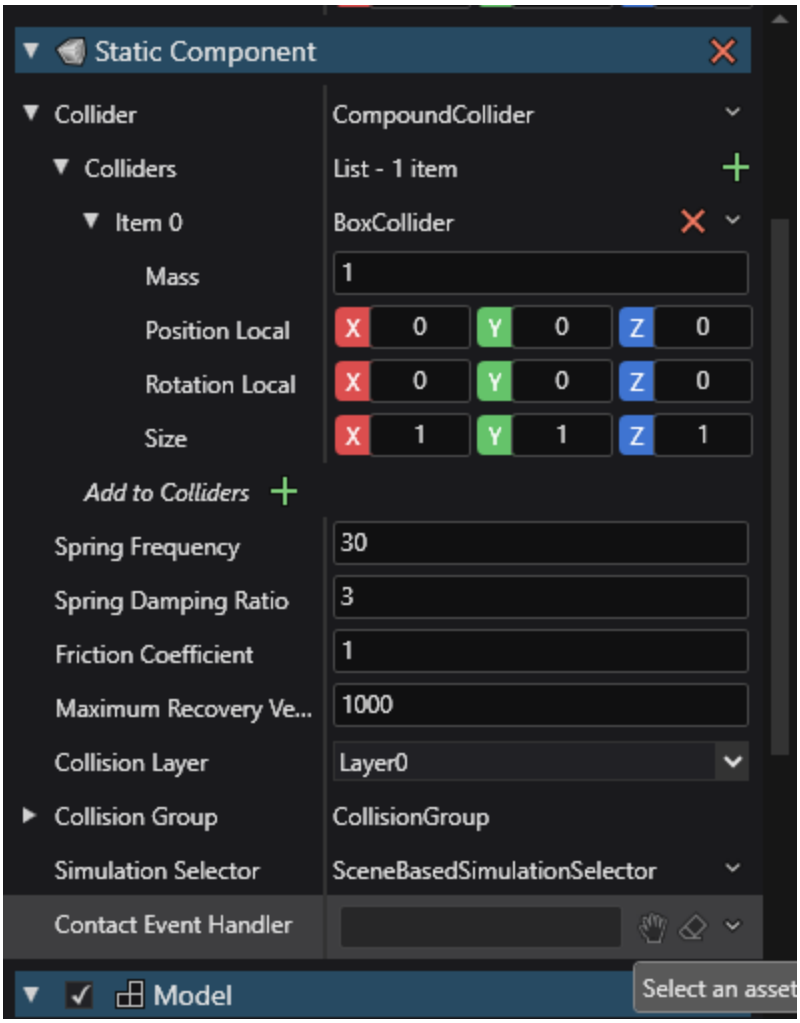
7. Add the script

Finally, let's add this script to the trigger entity as a component.

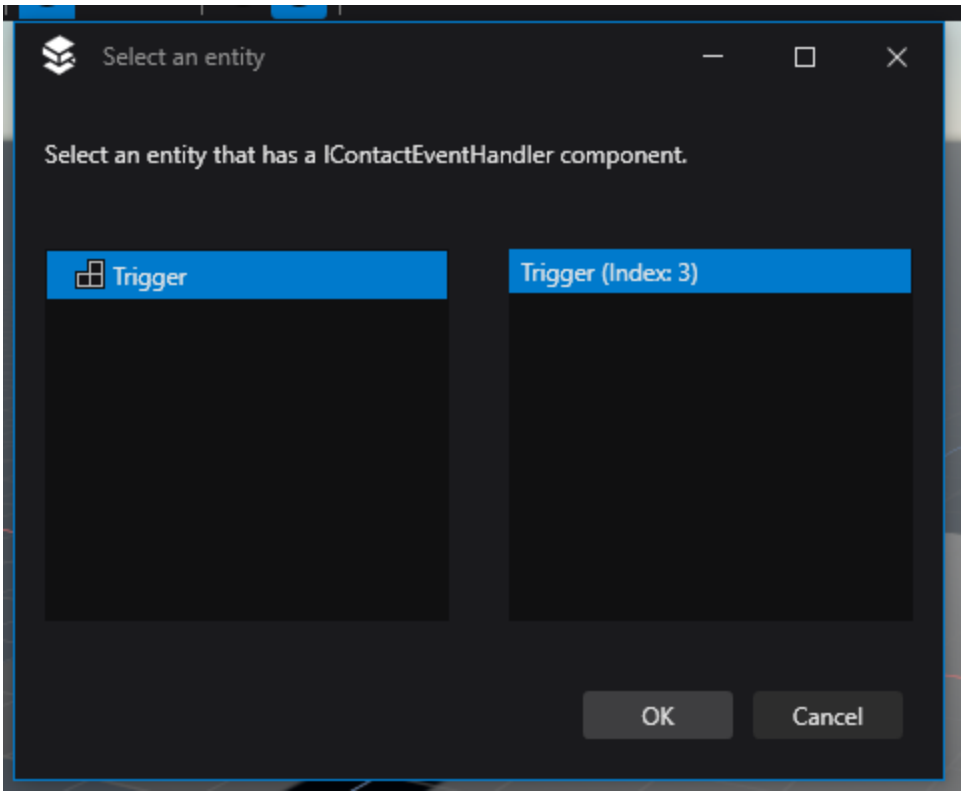
1. In **Game Studio**, select the **Trigger** entity.
2. In the **Property Grid**, click **Add component** and select the **Trigger** script.



3. Inside your **Static Component** click on the hand icon next to the `Contact Event Handler` property



4. Select your newly created `Trigger` component and press `Ok`



8. Run the project

Run the project (**F5**) to see the trigger in action.

The ball falls through the trigger, doubles in size, exits the trigger, and returns to its normal size.

See also

- Tutorial: Create a bouncing ball
- Colliders
- Collider shapes
- Scripts

Physics Jitter

Beginner Programmer

Physics engine operate on a fixed time step; instead of updating physics every frame, physics engine update at a fixed frequency, 60 times per second for example.

> **NOTE**
> This is controlled through the `FixedTimeStep` property in your Simulation Settings

The discrepancy between when your game updates and when physics updates means that you may see physics objects, or things that are attached to physics objects, move in a jittery manner. They jitter because they stay static after every physics update while the whole rest of the world moves every single frame.

It becomes less noticeable when your frame rate approaches the physics update rate, but it will always occasionally jitter as there is no feasible way to reach exact parity between frame rate and physics.

There are ways to combat this issue though.

Body Component Interpolation Mode

Bodies have an `Interpolation Mode` property, this property smooth out the jitter by interpolating the bodies' position and rotation between every physics update.

- `Interpolated` takes the two last position and rotation the physics engine set for this body and smoothly moves the object from the oldest to the youngest position. This means that the object is always 'in the past' compared to its physics representation since it is always between the two last position instead of exactly at the last one.
- `Extrapolated` also takes the two last position and rotation, but instead of being between the two, it starts from the last position and continues moving in the same direction. If the velocity of the object is constant, this works perfectly, but if it suddenly collides along that path, it will 'correct' itself leading to some jerky motion.

Jitter when a physics object follows another

There is a fair amount of jitter issues that may come up when setting up objects to move relative to each other. Here's everything that may prevent those objects from moving smoothly:

- The objects are not set to interpolate
- Your logic does not run in Physics Update, it overrides interpolation

- You are setting the entity's position/rotation instead of the physics object's `LinearVelocity`/`AngularVelocity`
- Your objects are not `Awake`

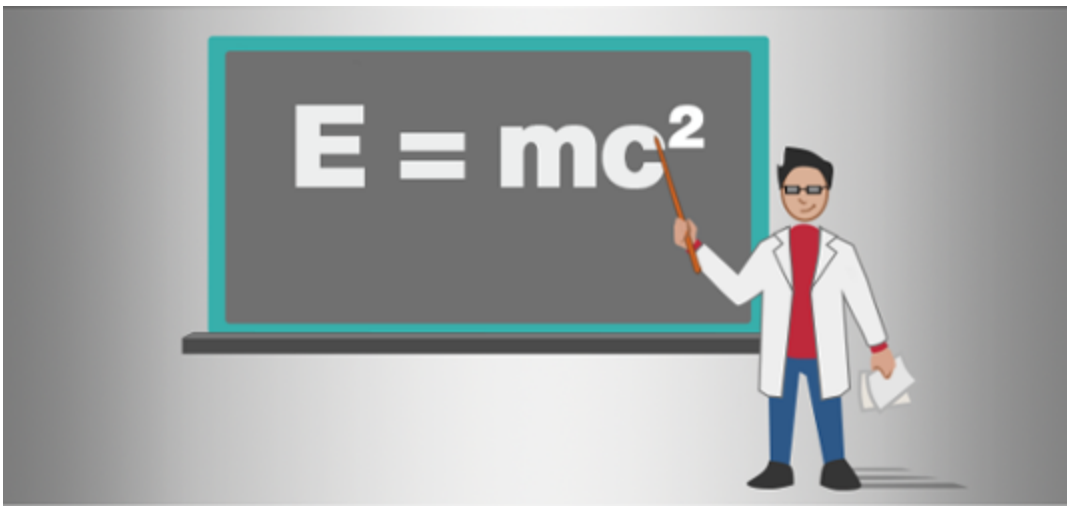
Lastly, maybe consider using a Constraint instead.

See also

- Simulation
- Index

Bullet Physics

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.



Stride simulates real-world physics such as gravity and collisions. This section explains how physics components work, how to add them to your project, and how to use them with scripts.

In this section

- Colliders: Create physics by adding collider components to entities
  - Static colliders: Colliders that don't move
  - Rigidbodies: Moving objects, affected by gravity and collisions
  - Kinematic rigidbodies: Physics objects controlled by scripts
  - Characters: Colliders for characters (such as player characters and NPCs)
  - Collider shapes: Define the shape of collider components
  - Triggers: Use triggers to detect passing objects
  - Constraints: Create appealing and realistic physics
- Raycasting: Trace intersecting objects
- Simulation: How Stride controls physics

Tutorials

- Create a bouncing ball: Use the static collider and rigidbody components to create a ball bouncing on a floor
- Script a trigger: Create a trigger that doubles the size of a ball when the ball passes through it

Additional physics resources

- Stride integrates the open-source Bullet Physics engine. For comprehensive details, consult the Bullet User Manual
- For solutions on mitigating physics jitter, refer to our guide on Fixing Physics Jitter

Colliders

Beginner Designer

> **WARNING**
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

To use physics in your project, add a **collider** component to an entity.

Colliders define the shapes and rules of physics objects. There are three types:

- static colliders don't move (eg walls, floors, heavy objects, etc)
- rigidbodies are moved around by forces such as collision and gravity (eg balls, barrels, etc)
- characters are controlled by user input (ie player characters)

You can also:

- set the shape of collider components
- make triggers, and detect when objects pass through them
- constrict collider movement with constraints

How colliders interact

Colliders interact according to the table below.

	Kinematic objects	**Kinematic triggers**	**Rigidbody colliders**	**Rigidbody triggers**	**Static colliders**	**Static triggers**
Kinematic objects	Collisions	Collisions	Collisions and dynamic	Collisions	Collisions	Collisions
Kinematic triggers	Collisions	Collisions	Collisions	Collisions	Collisions	Collisions
Rigidbody colliders	Collisions and dynamic	Collisions	Collisions and dynamic	Collisions	Collisions and dynamic	Collisions

	Kinematic objects	Kinematic triggers	Rigidbody colliders	Rigidbody triggers	Static colliders	Static triggers
Rigidbody triggers	Collisions	Collisions	Collisions	Collisions	Collisions	Collisions
Static colliders	Collisions	Collisions	Collisions and dynamic	Collisions	Nothing	Nothing
Static triggers	Collisions	Collisions	Collisions	Collisions	Nothing	Nothing

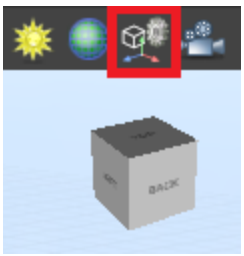
- "Collisions" refers to collision information and events only. This means the collision is detected in the code, but the objects don't bump into each other (no dynamic response).
- "Dynamic" means both collision information and events, plus dynamic response (ie the colliders bump into each other instead of passing through).

For example, rigidbody colliders dynamically collide with static colliders (ie bump into them). However, no objects dynamically collide with triggers; collisions are detected in the code, but objects simply pass through.

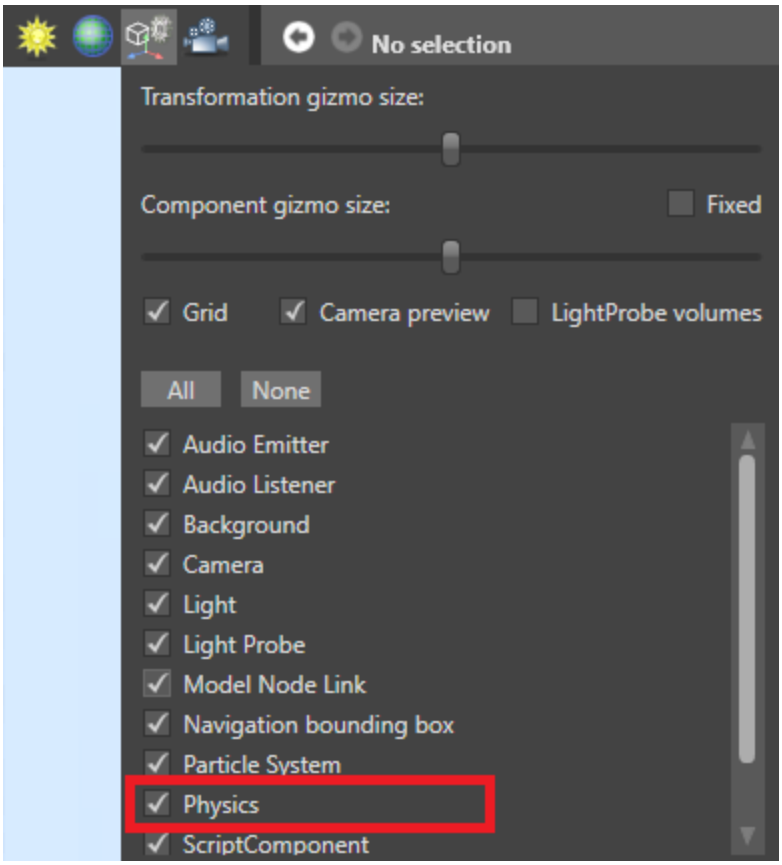
Show colliders in the Scene Editor

By default, colliders are invisible in the Scene Editor. To show them:

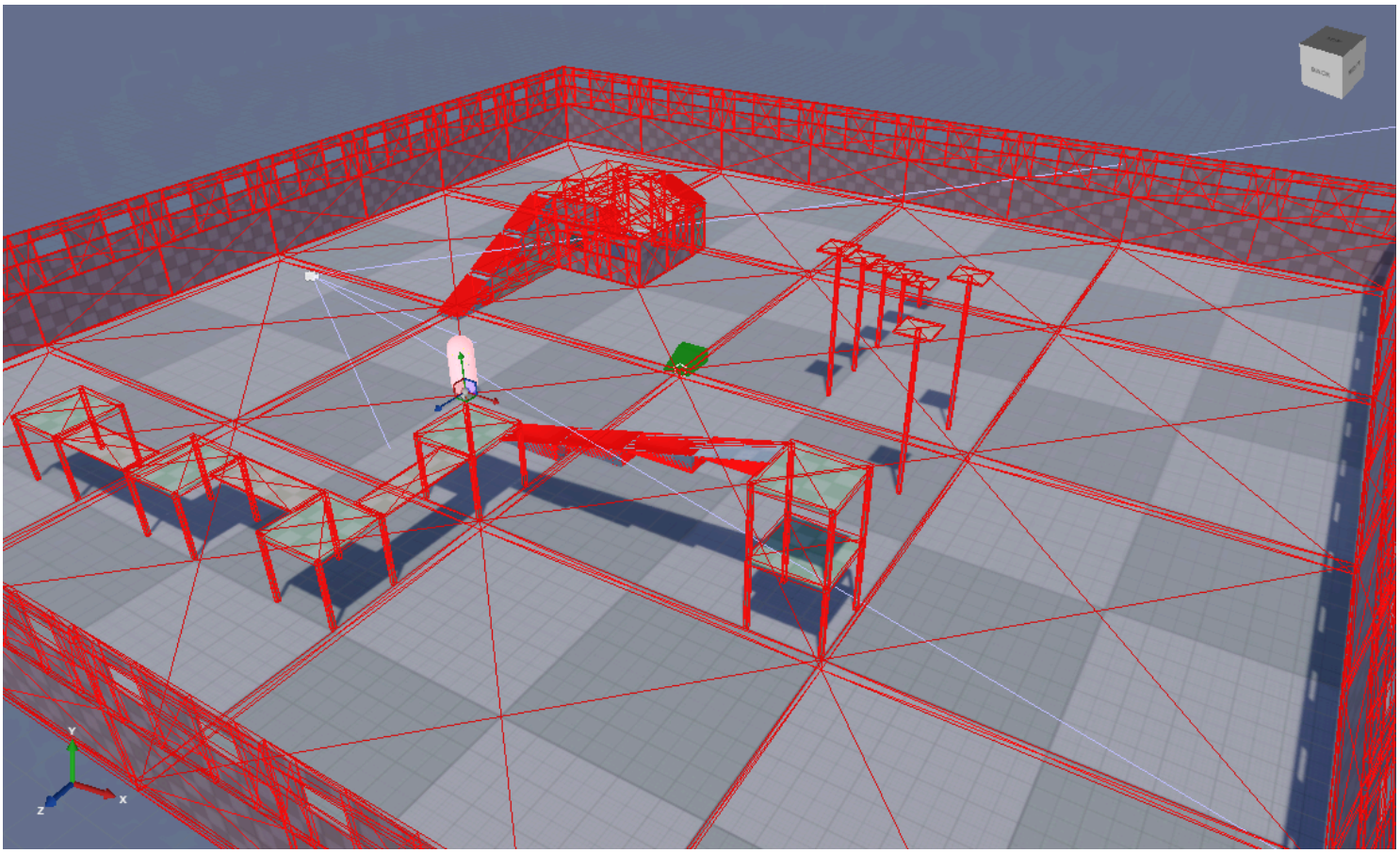
1. In the Game Studio toolbar, in the top right, click the **Display gizmo options** icon.



2. Select **Physics**.



The Scene Editor displays collider shapes.



Show colliders at runtime

You can make colliders visible at runtime, which is useful for debugging problems with physics. To do this, use:

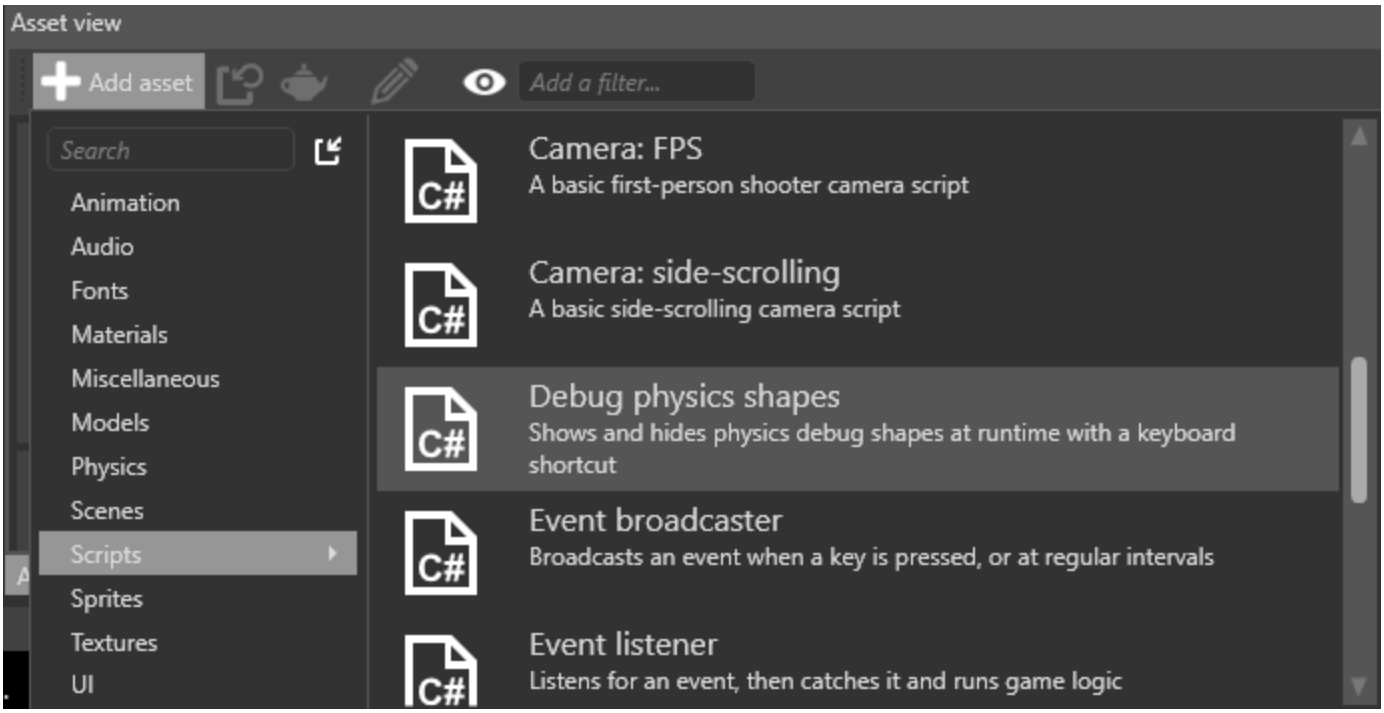
```
this.GetSimulation().ColliderShapesRendering = true;
```

> **NOTE**
>
> Collider shapes for infinite planes are always invisible.

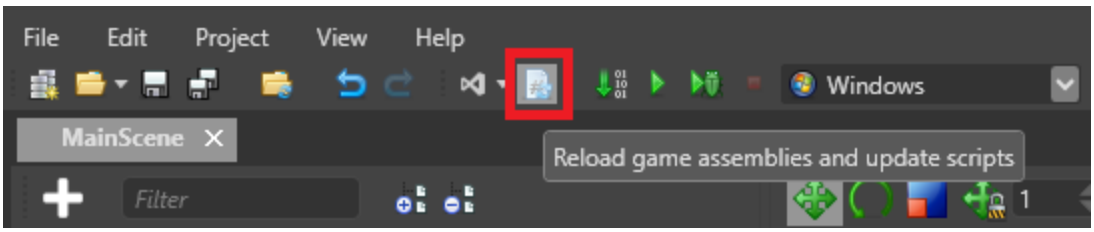
Keyboard shortcut

To show or hide collider shapes at runtime with a keyboard shortcut, use the **Debug physics shapes** script.

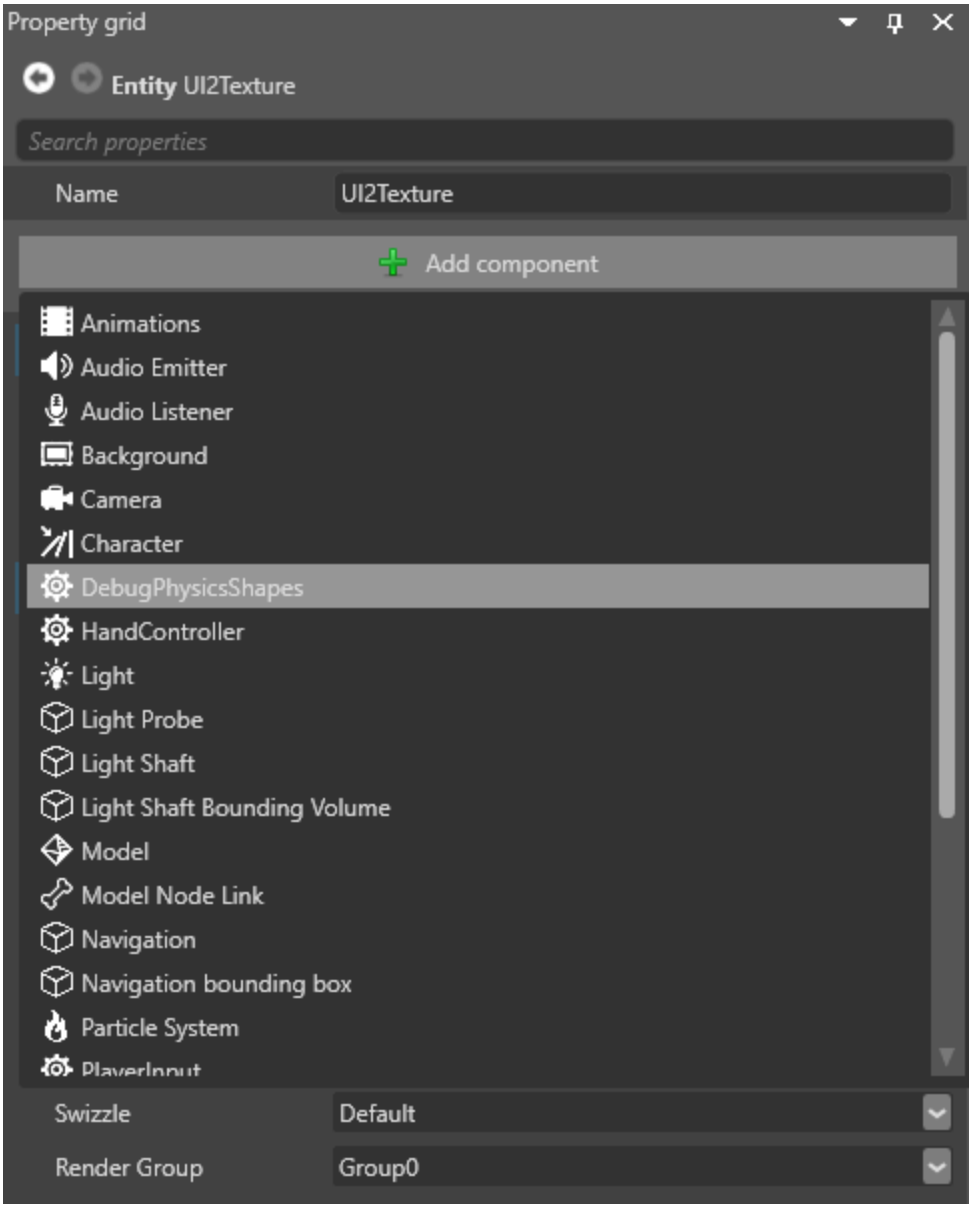
1. In the **Asset View**, click **Add asset**.
2. Select **Scripts** > **Debug physics shapes**.



3. In the Game Studio toolbar, click **Reload assemblies and update scripts**.



4. Add the **Debug physics shapes** script as a component to an entity in the scene. It doesn't matter which entity.



The script binds the collider shape visibility to **Left Shift + Left Ctrl + P**, so you can turn it on and off at runtime. You can edit the script to bind a different key combination.

See also

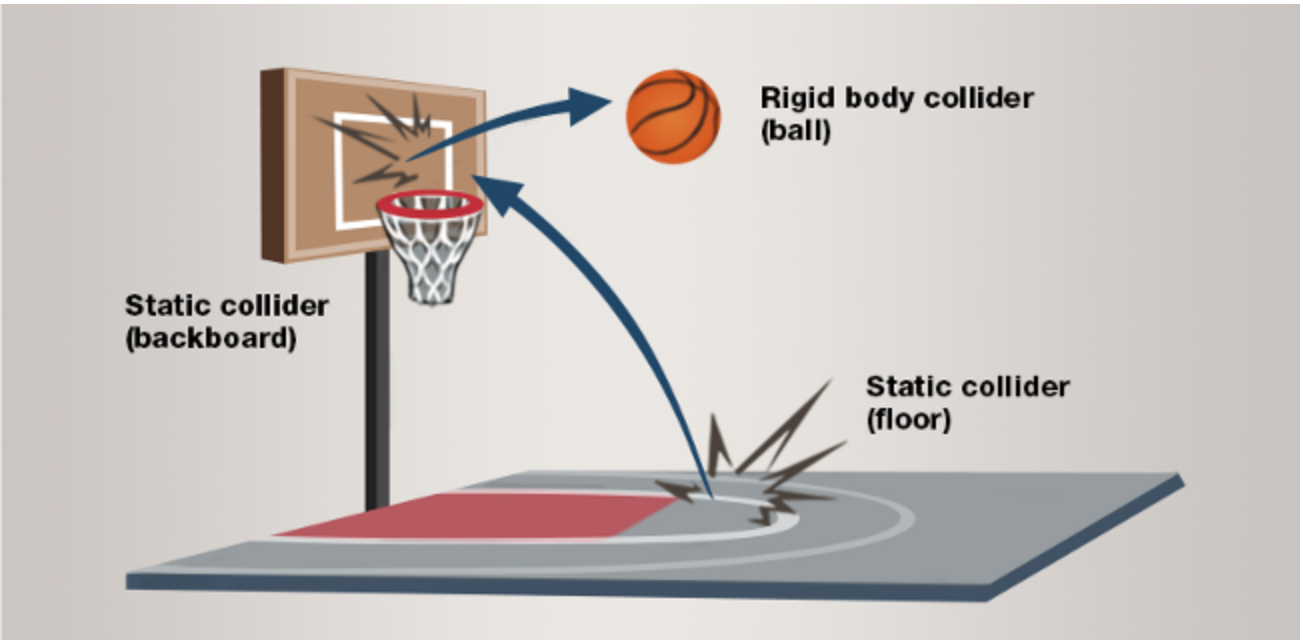
- Collider shapes
- Static colliders
- Rigidbodies
- Kinematic rigidbodies
- Simulation
- Physics tutorials

Static colliders

Beginner Designer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

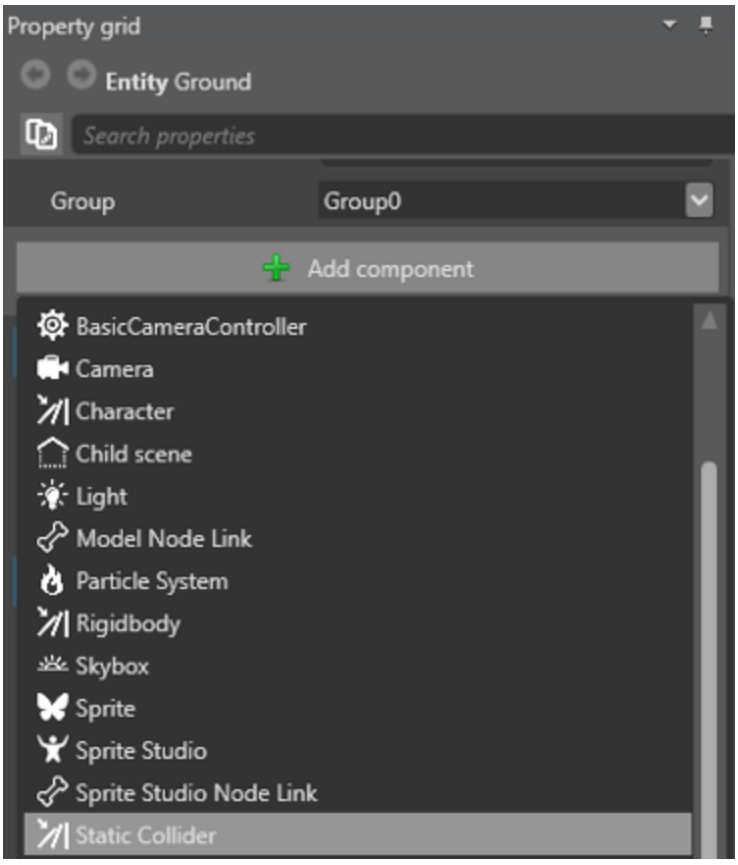
Static colliders aren't moved by forces such as gravity and collisions, but other physics objects can bump into them. Typical static colliders are strong immovable objects like walls, floors, large rocks, and so on.



Add a static collider

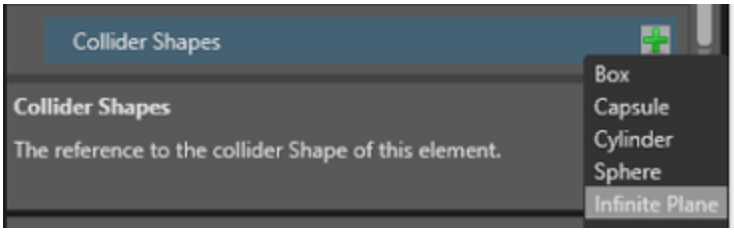
1. Select the entity you want to make a static collider.

2. In the **Property Grid**, click **Add component** and select **Static Collider**.



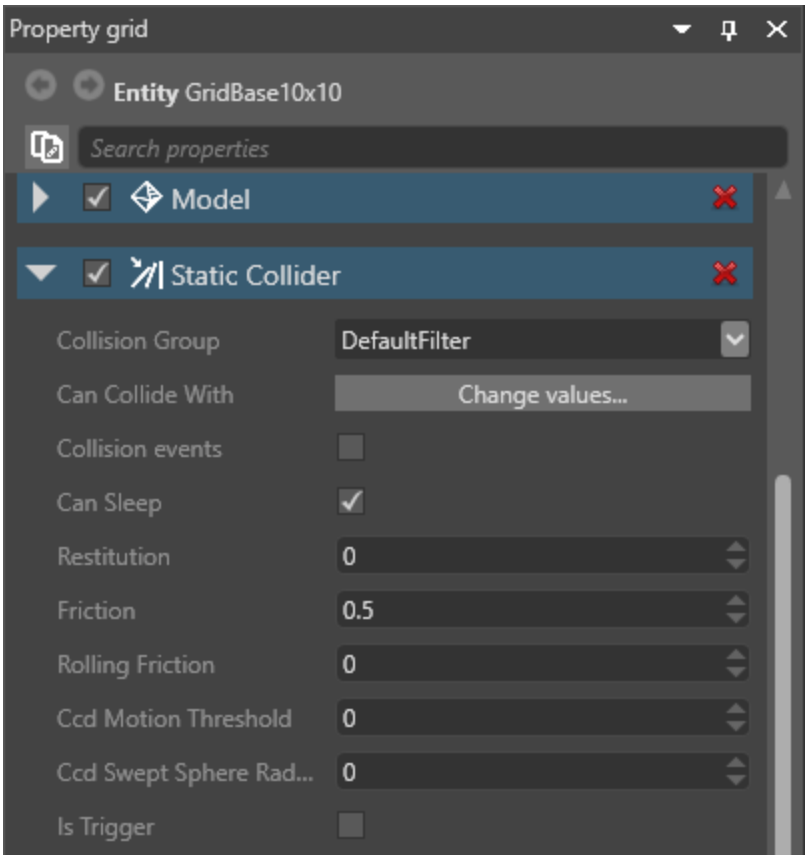
3. Set the collider shape to match the shape of the entity. To do this, in the **Property Grid**, expand the **Static Collider component** to view its properties.

4. Next to **Collider Shapes**, click (**Add**) and select the shape you want.



Static collider properties

You can adjust the static collider properties in the **Property Grid**.



Property	Description
Collision Group	Sets which collision group the object belongs to.
Can Collide With	Sets which groups the object collides with.
Collision Events	If this is enabled, the object reports collision events, which you can use in scripts. It has no effect on physics. If you have no scripts using collision events for the object, disable this option to save CPU.
Can Sleep	If this is enabled, the physics engine doesn't process physics objects when they're not moving. This saves CPU.
Restitution	Sets the amount of kinetic energy lost or gained after a collision. A typical value is between 0 and 1. If the restitution property of colliding entities is 0, the entities lose all energy and stop moving immediately on impact. If the restitution is 1, they lose no energy and rebound with the same velocity they collided at. Use this to change the "bounciness" of rigidbodies.
Friction	Sets the surface friction.

Property	Description
Rolling Friction	Sets the rolling friction.
CCD Motion Threshold	Sets the velocity at which continuous collision detection (CCD) takes over. CCD prevents fast-moving entities (such as bullets) erroneously passing through other entities.
CCD Swept Sphere Radius	Sets the radius of the bounding sphere containing the position between two physics frames during continuous collision detection.
Is Trigger	Toggles whether the static collider is a trigger.

Move a static collider at runtime

If you need to move a static collider at runtime, you can do it with a script:

```
PhysicsComponent.Entity.Transform.Position += PhysicsComponent.Entity.Transform.Position
+ Vector3.UnitX;
PhysicsComponent.Entity.Transform.UpdateWorldMatrix();
PhysicsComponent.UpdatePhysicsTransformation();
```

See also

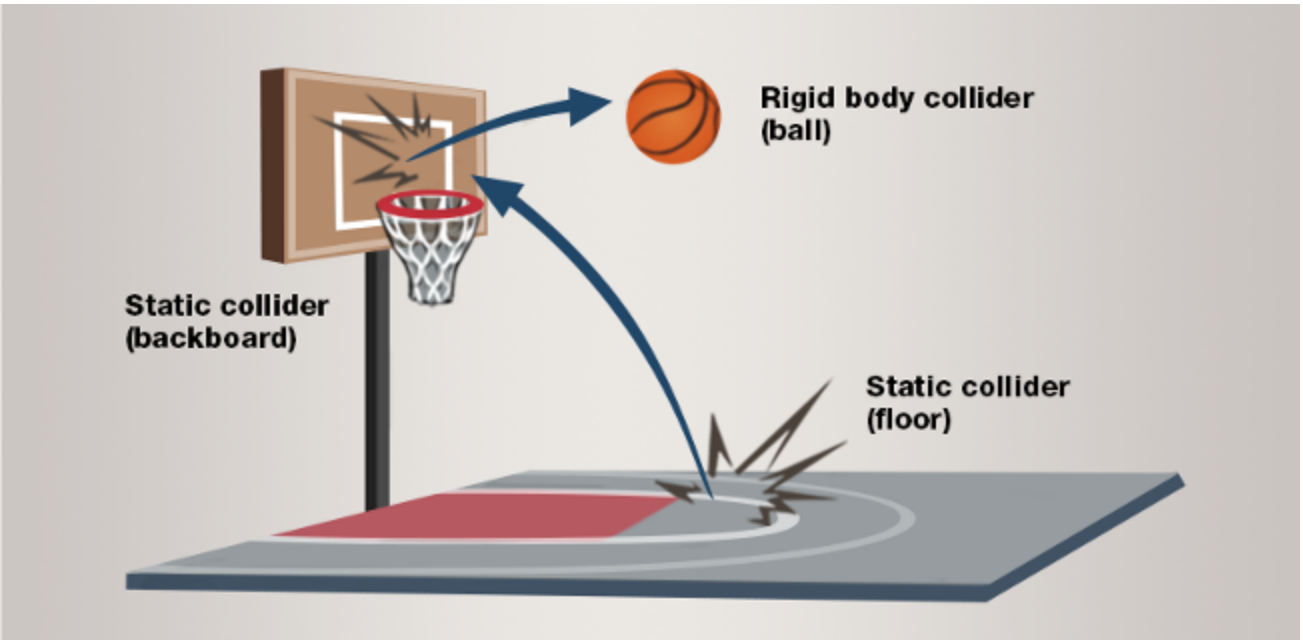
- Rigidbodies
- Characters
- Collider shapes
- Triggers

Rigidbodies

Beginner Designer

> **WARNING**
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

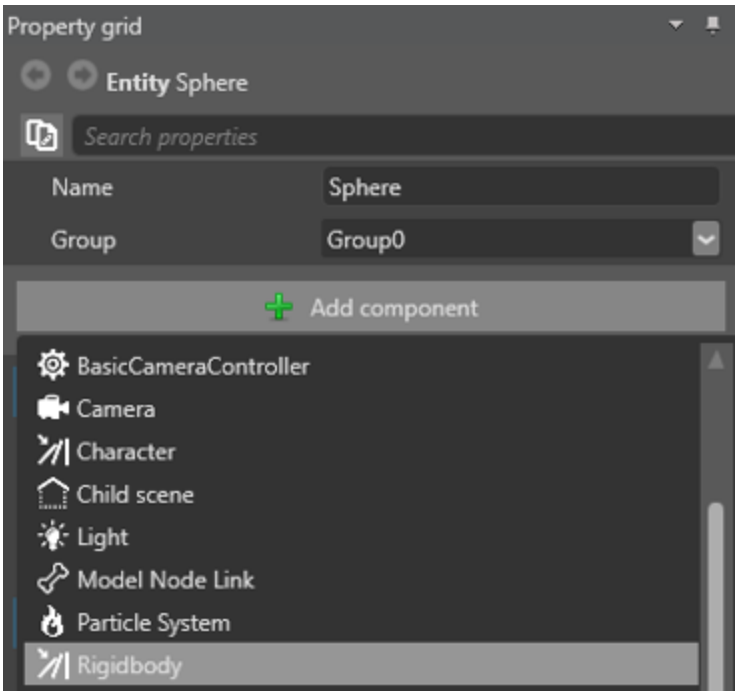
Rigidbodies move based on physical forces applied to them, such as gravity and collisions. Typical rigidbodies are boxes, balls, furniture, and so on — objects that are pushed, pulled, and knocked around, and also have effects on other rigidbodies they collide with.



Add a rigidbody collider

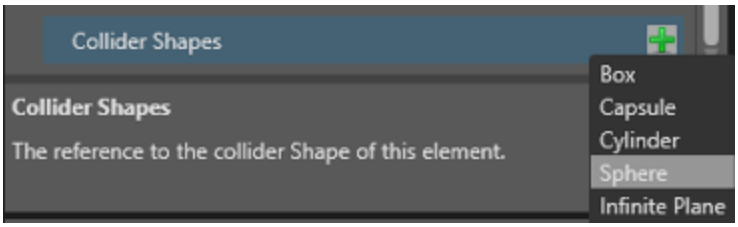
1. Select the entity you want to be a rigidbody collider.

2. In the **Property Grid**, click **Add component** and select **Rigidbody**.



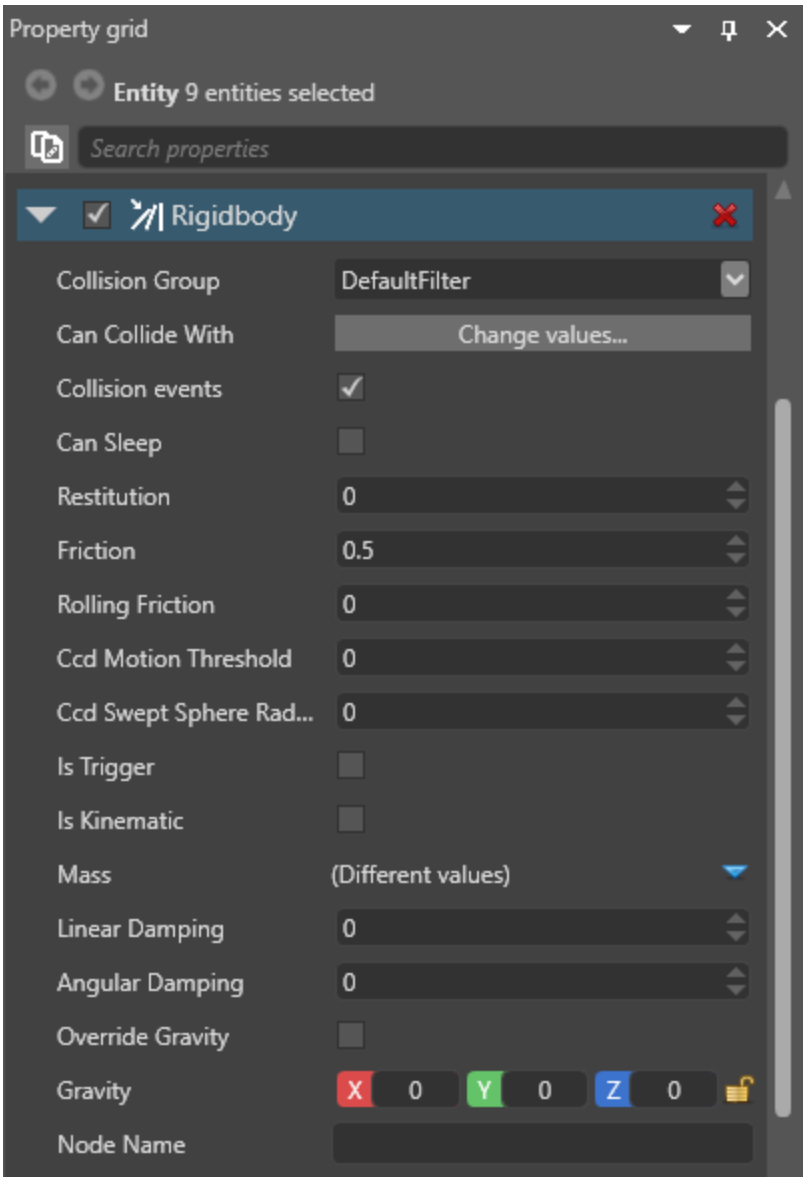
3. Set the collider shape to match the entity. To do this, in the **Property Grid**, expand the **Rigidbody component** to view its properties.

4. Next to **Collider Shapes**, click (**Add**) and select the shape you want.



Component properties

You can adjust the rigidbody properties in the **Property Grid**.



Property	Description
Collision Group	Sets which collision group the object belongs to.
Can Collide With	Sets which groups the object collides with.
Collision Events	If this is enabled, the object reports collision events, which you can use in scripts. It has no effect on physics. If you have no scripts using collision events for the object, disable this option to save CPU.
Can Sleep	If this is enabled, the physics engine doesn't process physics objects when they're not moving. This saves CPU.

Property	Description
Restitution	Sets the amount of kinetic energy lost or gained after a collision. A typical value is between 0 and 1. If the restitution property of colliding entities is 0, the entities lose all energy and stop moving immediately on impact. If the restitution is 1, they lose no energy and rebound with the same velocity they collided at. Use this to change the "bounciness" of rigidbodies.
Friction	Sets the surface friction.
Rolling Friction	Sets the rolling friction.
CCD Motion Threshold	Sets the velocity at which continuous collision detection (CCD) takes over. CCD prevents fast-moving entities (such as bullets) erroneously passing through other entities.
CCD Swept Sphere Radius	Sets the radius of the bounding sphere containing the position between two physics frames during continuous collision detection.
Is Trigger	Toggles whether the rigidbody is a trigger.
Is Kinematic	Toggles whether the rigidbody is kinematic and therefore moved only by its Transform property.
Mass	Sets the collider mass. For large differences, use a point value; for example, write *0.1* or *10*, not *1* or *100000*.
Linear damping	The amount of damping for directional forces.
Angular damping	The amount of damping for rotational forces.
Override Gravity	Overrides gravity with the vector specified in Gravity.
Gravity	Sets a custom gravity vector applied if Override Gravity is selected.
Node Name	If the collider entity contains a bone structure, the node name can refer to a bones node name to be linked to that specific bone.

Property	Description
Collider Shapes	Adds a collider shape.

See also

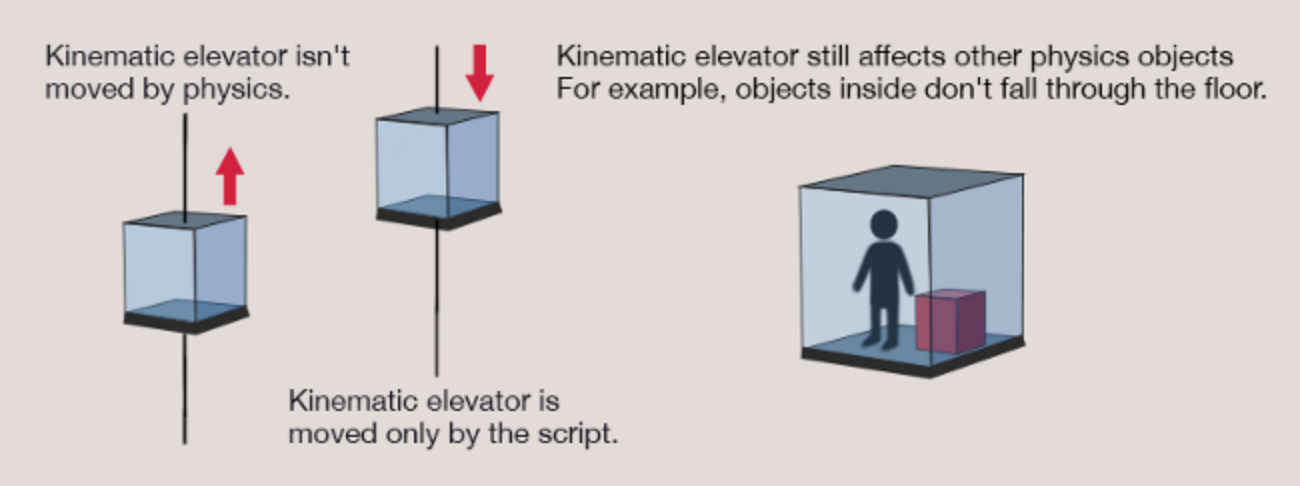
- Kinematic rigidbodies
- Static colliders
- Characters
- Collider shapes
- Triggers

Kinematic rigidbodies

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

Sometimes you want to move rigidbodies in a specific way rather than have other objects move them. For example, you might control an elevator with a script, via its `Transform` property, rather than have other objects push and pull it. This is a **kinematic** rigidbody.

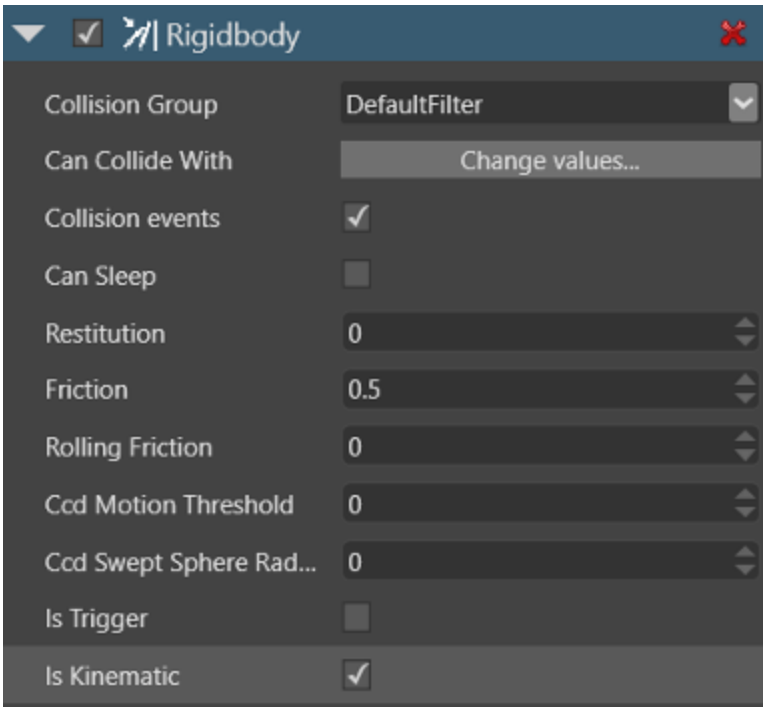
Although kinematic rigidbodies aren't moved by physics, other objects can still collide with them. For example, in the case of the elevator, objects placed inside won't fall through the elevator floor.



Make a kinematic rigidbody

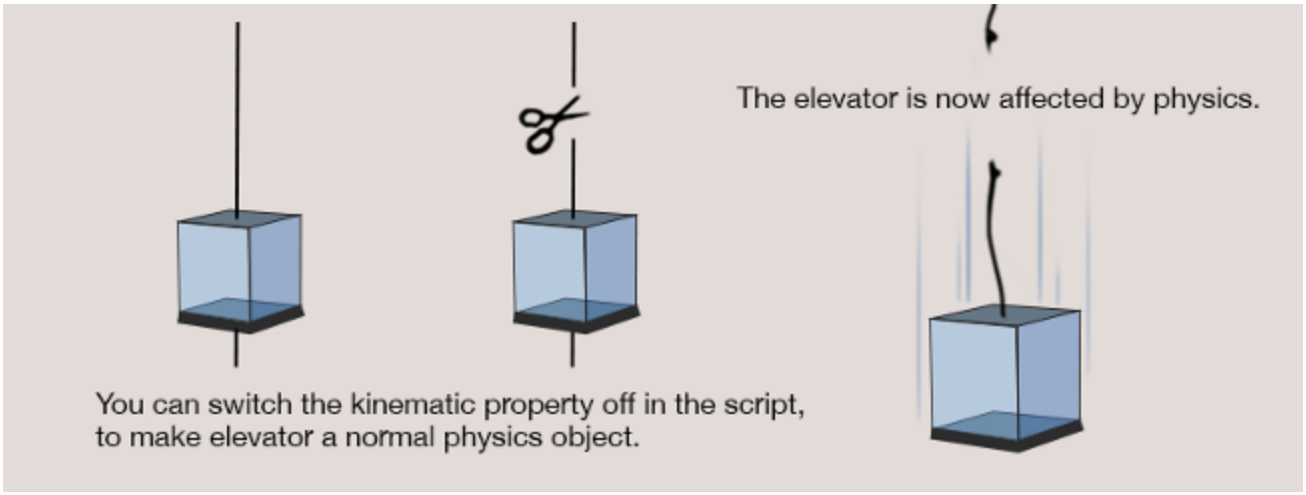
1. Select the entity you want to be a kinematic rigidbody.

2. In the **Property Grid**, under the **Rigidbody** component properties, select **Is kinematic**.



Scripting kinematic rigidbodies

You can script the **Is kinematic** property to turn on and off on certain events. For example, imagine our kinematic elevator's suspension cables are cut. You can script the **Is kinematic** property to change to *false* when this happens. The elevator becomes subject to the usual forces of physics, and falls.



See also

- Rigidbodies
- Static colliders
- Characters
- Collider shapes
- Triggers

Characters

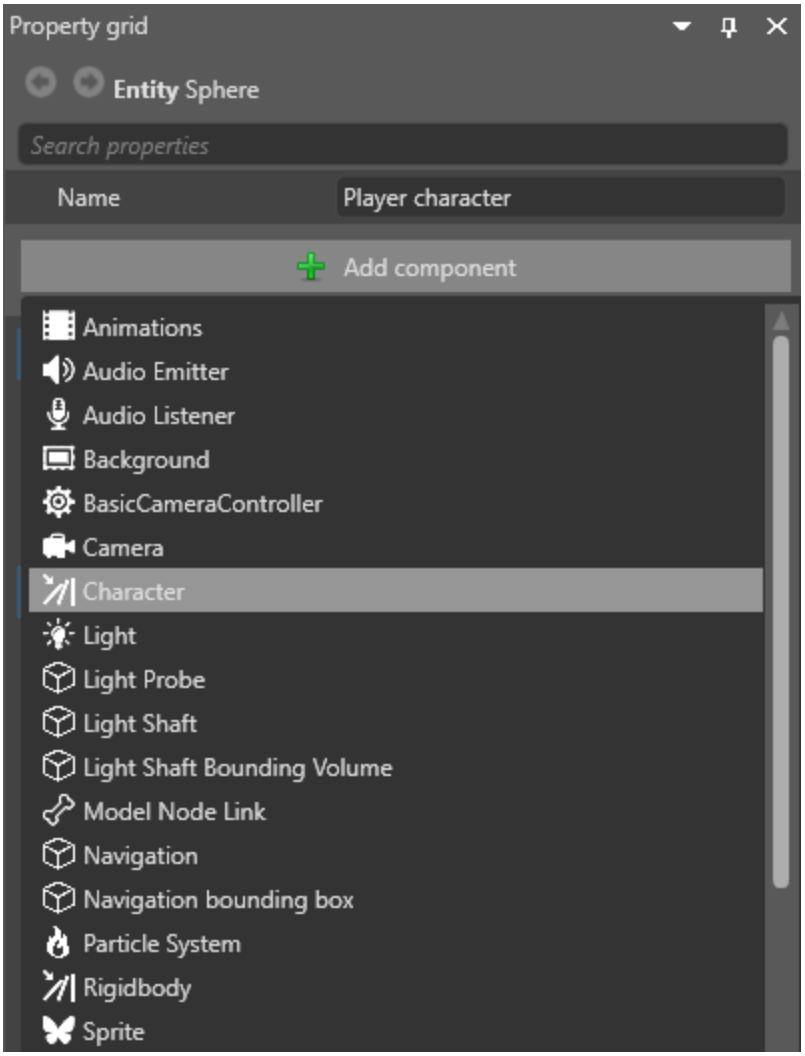
Beginner Designer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

Character colliders are used for player and script-controlled characters such as NPCs. Entities with character components can only be moved with SetVelocity, Jump, and Teleport.

Add a character component to an entity

1. In the **Scene Editor**, select the entity you want to add the component to.
2. In the **Property Grid**, click **Add component** and select **Character**.



> **NOTE**
>
> For the character collider to interact with other physics objects, you also need to set a collider shape in the collider component properties. The capsule shape is appropriate for most character colliders. For more information, see collider shapes.

Component properties

You can adjust the character component properties in the **Property Grid**.

Property	Description
Collision Group	Sets which collision group the object belongs to.
Can Collide With	Sets which groups the object collides with.
Collision Events	If this is enabled, the object reports collision events, which you can use in scripts. It has no effect on physics. If you have no scripts using collision events for the object, disable this option to save CPU.
Can Sleep	If this is enabled, the physics engine doesn't process physics objects when they're not moving. This saves CPU.
Restitution	Sets the amount of kinetic energy lost or gained after a collision. A typical value is between 0 and 1. If the restitution property of colliding entities is 0, the entities lose all energy and stop moving immediately on impact. If the restitution is 1, they lose no energy and rebound with the same velocity they collided at. Use this to change the "bounciness" of rigidbodies.
Friction	Sets the surface friction.
Rolling Friction	Sets the rolling friction.
CCD Motion Threshold	Sets the velocity at which continuous collision detection (CCD) takes over. CCD prevents fast-moving entities (such as bullets) erroneously passing through other entities.
CCD Swept Sphere	Sets the radius of the bounding sphere containing the position between two physics frames during continuous collision detection.

Property	Description
Radius	
Gravity	For rigidbodies, sets a custom gravity vector applied if Override Gravity is selected. For characters, specifies how much gravity affects the character.
Step Height	The maximum height the character can step onto.
Fall Speed	The maximum fall speed.
Max Slope	The maximum slope the character can climb, in degrees.
Jump Speed	The amount of jump force.

See also

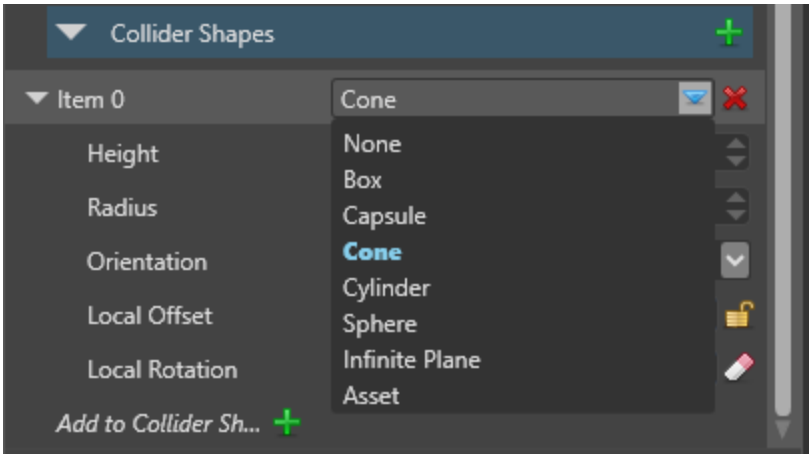
- Static colliders
- Rigidbodies
- Collider shapes

Collider shapes

Beginner Designer

> **WARNING**
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

For colliders to interact, you need to set their shape in the **Property Grid**. You can specify a geometric shape, or use a collider shape asset.



Components can have multiple intersecting shapes, and don't have to match the entity model, if it has one. Each shape has additional properties including size, orientation, offset, and so on.

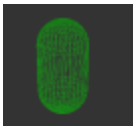
Types of collider shape

Box

Property	Description
Is 2D	Makes the box infinitely flat in one dimension.
Size	The box size in XYZ values.
Local offset	The box position relative its entity.

Property	Description
Local rotation	The box rotation in XYZ values.

Capsule



The capsule shape is especially useful for character components, as its curved base lets the entity move to higher planes (eg when climbing staircases).

Property	Description
Is 2D	Makes the capsule infinitely flat in one dimension.
Length	The length of the capsule.
Radius	The radius of the capsule.
Orientation	The axis along which the shape is stretched (X, Y, or Z).
Local offset	The capsule position relative to its entity.
Local rotation	The capsule rotation in XYZ values.

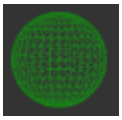
Cone

Property	Description
Height	The height of the cone.
Radius	The radius of the cone at the bottom end.
Orientation	The axis along which the shape is stretched (X, Y, or Z).
Local offset	The cone position relative to its entity.
Local rotation	The cone rotation in XYZ values.

Cylinder

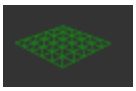
Property	Description
Height	The length of the cylinder.
Radius	The radius of the cylinder.
Orientation	Sets the axis along which the shape is stretched (X, Y, or Z).
Local offset	The cylinder position relative to its entity.
Local rotation	The cylinder rotation in XYZ values.

Sphere



Property	Description
Is 2D	Makes the sphere infinitely flat in one dimension.
Radius	The radius of the sphere.
Local offset	The sphere position relative to its entity.

Infinite plane



The infinite plane covers an infinite distance across one dimension. Think of it like a wall or floor stretching into the distance for ever. You can use several infinite planes together to box users in and stop them "tunneling" outside the level.

Property	Description
Normal	Which vector (X, Y, or Z) is perpendicular to the plane. For example, to make an infinite floor, set the normal property to: *X:0, Y:1, Z:0*.

Property	Description
Offset	The plane position relative to its entity.

Asset

Assigns a collider shape from a collider shape asset (see **Collider shape assets** below).

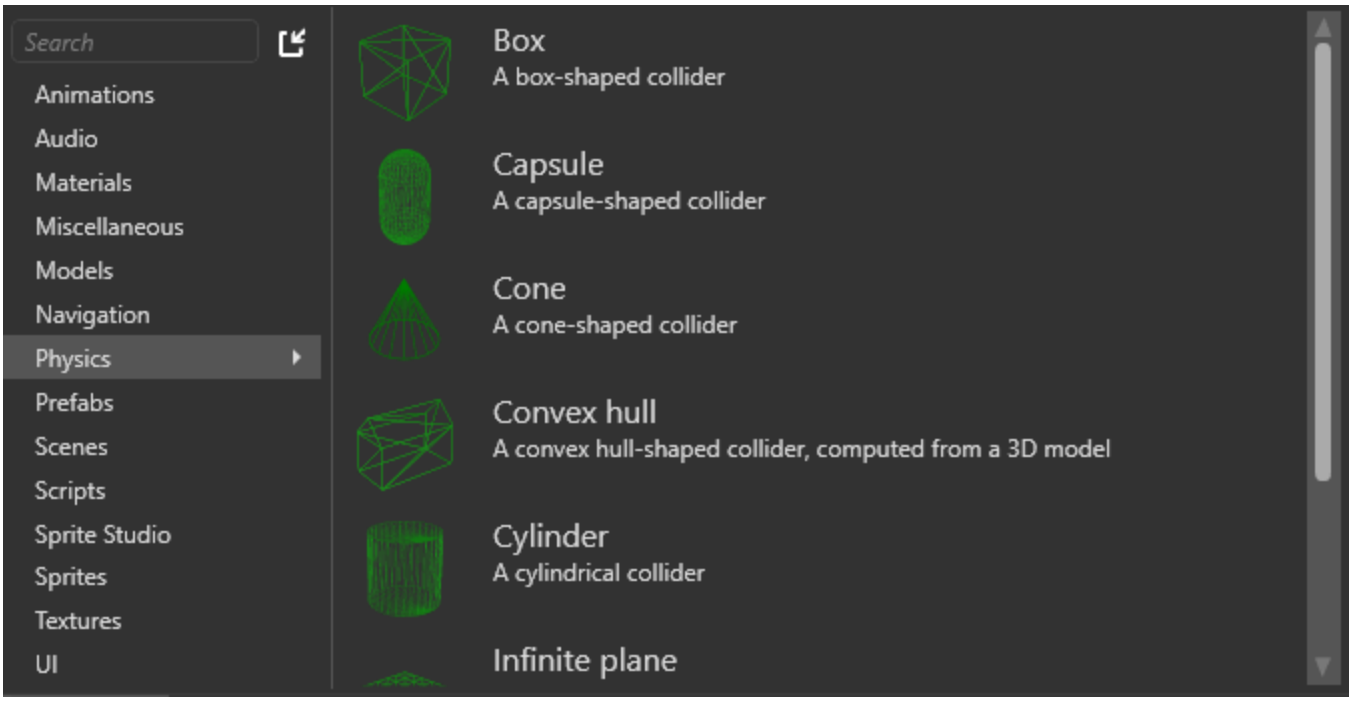
Property	Description
Shape	The collider shape asset used to generate the collider shape.

Collider shape assets

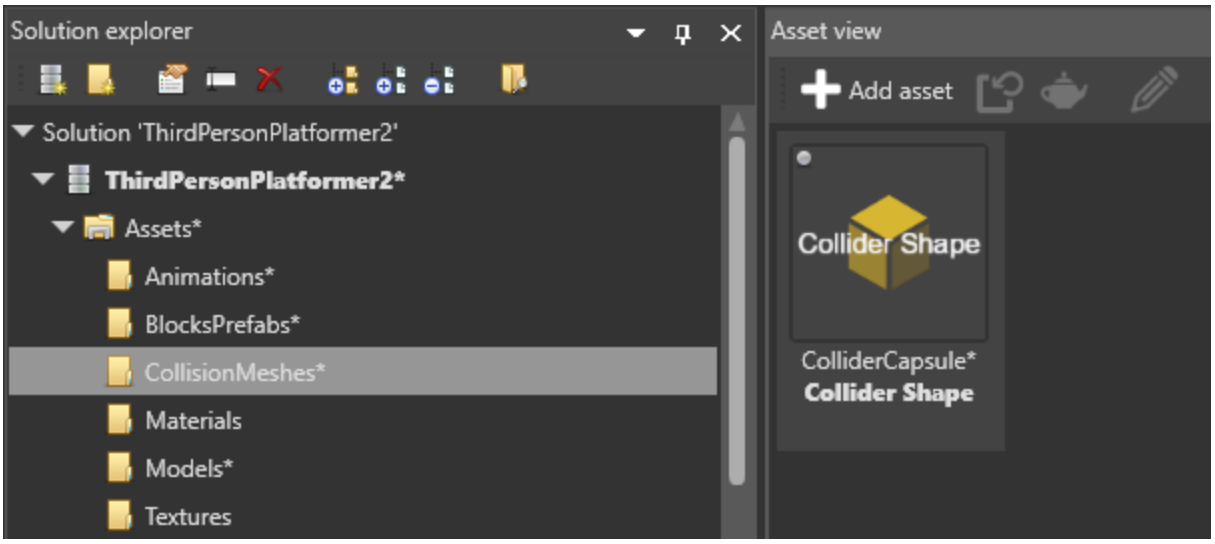
You can also create **collider shape assets** and use them as your collider shape. This means you can edit the collider shape asset and automatically update it in every entity that uses it.

Create a collider shape asset

1. In the **Asset View** (bottom by default), click **Add asset**.
2. Select **Physics**, then select the shape you want to create.



Game Studio creates the new collider shape asset in the **CollisionMeshes** folder.

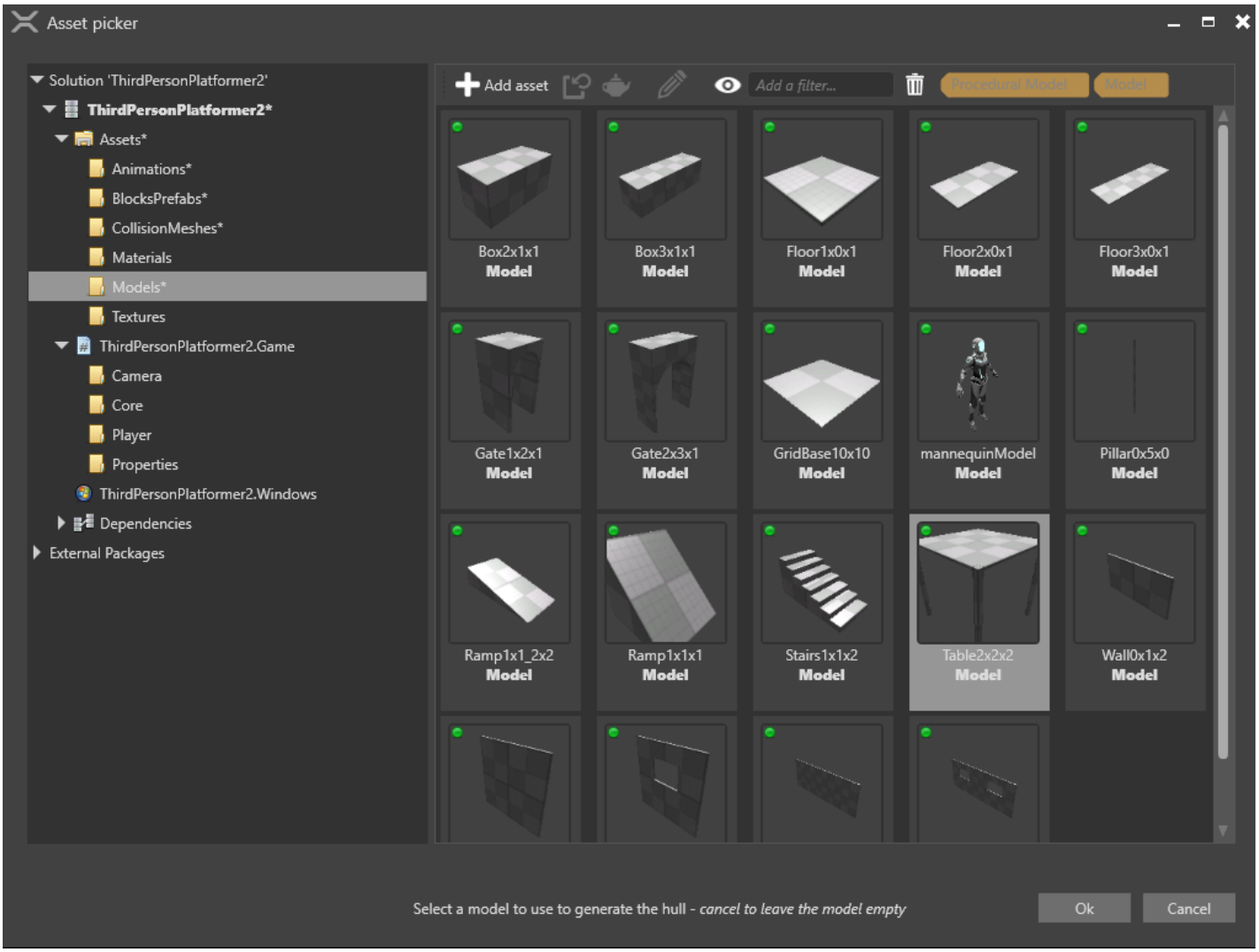


Create a collider shape asset from a model

This is useful to quickly create a collider shape that matches a model.

1. In the **Asset View** (bottom by default), click **Add asset**.
2. Select **Physics** > **Convex hull**.

   The **Select an asset** window opens.

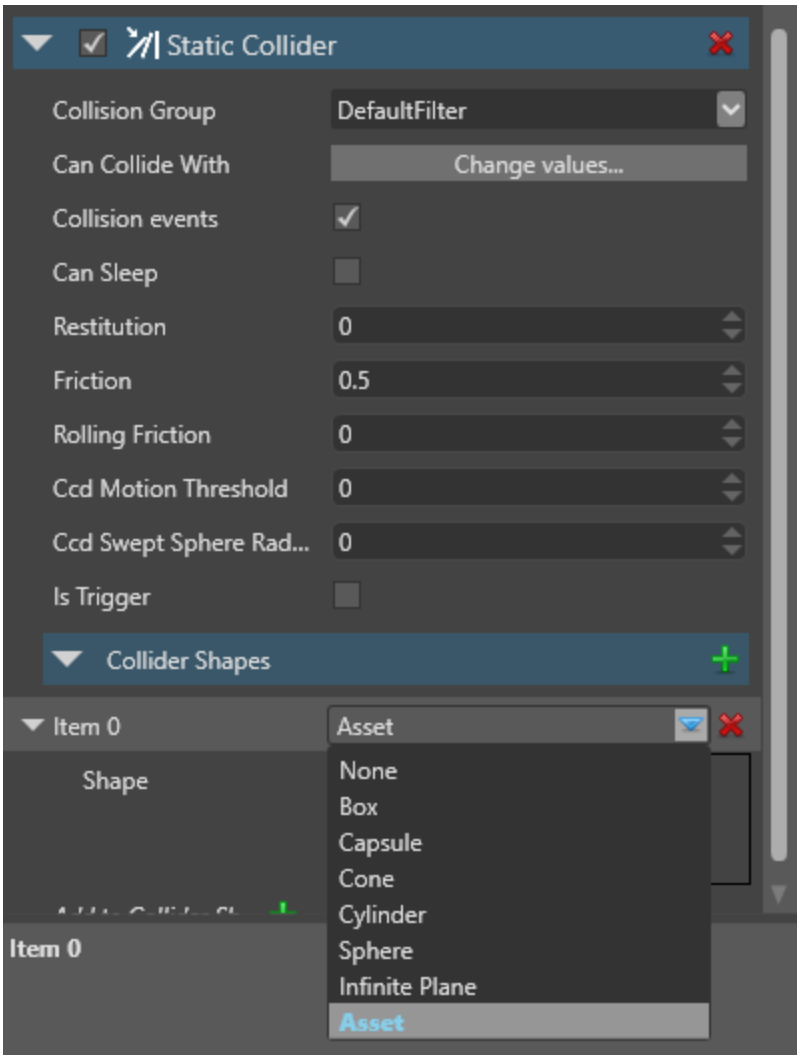


3. Browse to the model asset you want to create a collider shape asset from and click **OK**.

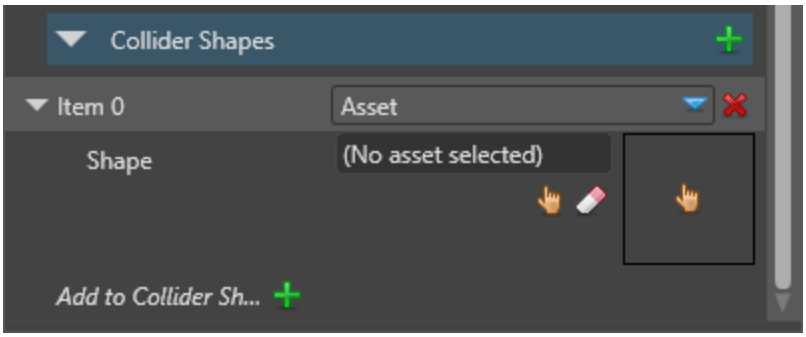
Game Studio creates a collider shape asset from the model.

Use a collider shape asset

1. Under the **static collider** or **rigidbody** properties, under **Collider Shapes**, select **Asset**.



2. Next to **Shape**, specify the collider shape asset you want to use.



   To do this, drag the asset from the **Asset View** to the **Shape** field in the Property Grid. Alternatively, click (**Select an asset**) and browse to the asset.

See also

- Colliders
- Tutorial: Create a bouncing ball
- Tutorial: Script a trigger

Triggers

Beginner Designer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

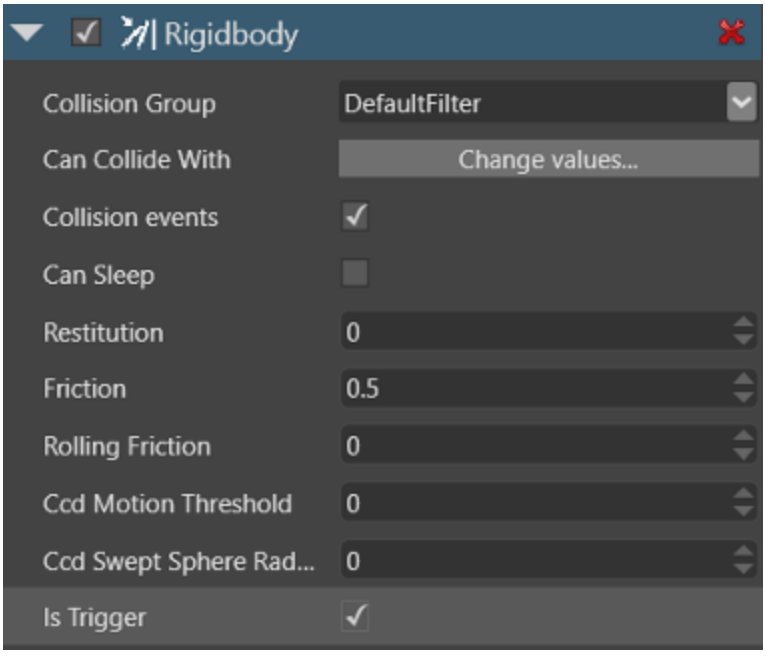
If you set a collider to be a **trigger**, other colliders no longer bump into it. Instead, they pass through.

The trigger detects when colliders enter it, which you can use to script events. For example, you can detect when a player character enters a room, and use this in your script to trigger an event. For more information, see Events.

> **NOTE**
>
> Character colliders can't be used as triggers.

Create a trigger

1. Create a collider.
2. In the **Property Grid**, under the collider component properties, select **Is Trigger**.



Detect trigger collisions

You can see when something enters the trigger using the following code:

```
// Wait for an entity to collide with the trigger
var firstCollision = await trigger.NewCollision();

var otherCollider = trigger == firstCollision.ColliderA
    ? firstCollision.ColliderB
    : firstCollision.ColliderA;
```

Alternatively, directly access the `TrackingHashSet`:

```
var trigger = Entity.Get<PhysicsComponent>();
foreach (var collision in trigger.Collisions)
{
    //do something with the collision
}
```

Or use the `TrackingHashSet` events:

```
var trigger = Entity.Get<PhysicsComponent>();
trigger.Collisions.CollectionChanged += (sender, args) =>
{
    if (args.Action == NotifyCollectionChangedAction.Add)
    {
        //new collision
        var collision = (Collision) args.Item;
        //do something
    }
    else if (args.Action == NotifyCollectionChangedAction.Remove)
    {
        //old collision
        var collision = (Collision)args.Item;
        //do something
    }
};
```

For an example of how to use triggers, see the Script a trigger tutorial.

See also

- Tutorial: Script a trigger

- Colliders
- Collider shapes
- Events

Constraints

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

> **WARNING**
>
> This documentation is under construction.

Advanced Programmer

Constraints restrict rigidbodies to certain movement patterns. For example, a realistic knee joint can only move along one axis and can't bend forwards.

Constraints can either link two rigidbodies together, or link a single rigidbody to a point in the world. They allow for interaction and dependency among rigidbodies.

There are six types of constraints:

- hinges
- gears
- sliders
- cones (twist and turn)
- point to point (fixed distance between two colliders)
- six degrees of freedom

For a demonstration of the different constraints, load the **PhysicsSample** sample project.

Create a constraint

> **NOTE**
>
> Currently, you can only use constraints from scripts.

To create a constraint, use the Simulation static method CreateConstraint:

```
CreateConstraint(ConstraintTypes type, RigidbodyComponent rigidBodyA, Matrix frameA,
bool useReferenceFrameA);
```

This links RigidBodyA to the world at its current location. The boolean useReferenceFrameA specifies which coordinate system the limit is applied to (either RigidBodyA or the world).

> **NOTE**
>
> - In the case of ConstraintTypes.Point2Point, the frame represents a pivot in A. Only the translation vector is considered. useReferenceFrameA is ignored.
> - In the case of ConstraintTypes.Hinge, the frame represents a pivot in A and Axis in A. This is because the hinge allows only a limited angle of rotation between the rigidbody and the world.
> - In the case of ConstraintTypes.ConeTwist, useReferenceFrameA is ignored.
> - ConstraintTypes.Gear needs two rigidbodies to be created. This function will throw an exception.

```
CreateConstraint(ConstraintTypes type, RigidbodyComponent rigidBodyA, RigidbodyComponent
rigidBodyB, Matrix frameA, Matrix frameB, bool useReferenceFrameA)
```

This method links RigidBodyA to RigidBodyB.

> **NOTE**
>
> - In the case of ConstraintTypes.Point2Point, the frame represents a pivot in A or B. Only the translation vector is considered. useReferenceFrameA is ignored.
> - In the case of ConstraintTypes.Hinge the frame represents pivot in A/B and Axis in A/B. This is because the hinge allows only a limited angle of rotation between the rigidbody and the world in this case.
> - In the case of ConstraintTypes.ConeTwist, useReferenceFrameA is ignored.
> - In the case of ConstraintTypes.Gear, useReferenceFrameA is ignored. The frame just represents the axis either in A or B; only the translation vector (which should contain the axis) is used.

The boolean useReferenceFrameA determines which coordinate system (RigidBodyA or RigidBodyB) the limits are applied to.

Add constraints to the simulation

After you create a constraint, add it to the simulation from a script by calling:

```
this.GetSimulation().AddConstraint(constraint);
```

or:

```
var disableCollisionsBetweenLinkedBodies = true;
this.GetSimulation().AddConstraint(constraint, disableCollisionsBetweenLinkedBodies);
```

The parameter disableCollisionsBetweenLinkedBodies stops linked bodies colliding with each other.

Likewise, to remove a constraint from the simulation, use:

```
this.GetSimulation().RemoveConstraint(constraint);
```

See also

- Colliders

Raycasting

Intermediate Programmer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

Raycasting traces an invisible line through the scene to find intersecting colliders. This is useful, for example, to check which objects are in a gun's line of fire, or are under the mouse cursor when the user clicks.

> **NOTE**
>
> Raycasting uses **colliders** to calculate intersections. It ignores entities that have no collider component. For more information, see Colliders.

To use a raycast, in the current Simulation, use Simulation.Raycast.

For an example of raycasting, see the **Physics Sample** project included with Stride.

Example code

This code sends a raycast from the mouse's screen position:

```
public static bool ScreenPositionToWorldPositionRaycast(Vector2 screenPos, CameraComponent
camera, Simulation simulation)
{
    Matrix invViewProj = Matrix.Invert(camera.ViewProjectionMatrix);

    // Reconstruct the projection-space position in the (-1, +1) range.
    //    Don't forget that Y is down in screen coordinates, but up in projection space
    Vector3 sPos;
    sPos.X = screenPos.X * 2f - 1f;
    sPos.Y = 1f - screenPos.Y * 2f;

    // Compute the near (start) point for the raycast
    // It's assumed to have the same projection space (x,y) coordinates and z = 0 (lying on
the near plane)
```

```
    // We need to unproject it to world space
    sPos.Z = 0f;
    var vectorNear = Vector3.Transform(sPos, invViewProj);
    vectorNear /= vectorNear.W;

    // Compute the far (end) point for the raycast
    // It's assumed to have the same projection space (x,y) coordinates and z = 1 (lying on
the far plane)
    // We need to unproject it to world space
    sPos.Z = 1f;
    var vectorFar = Vector3.Transform(sPos, invViewProj);
    vectorFar /= vectorFar.W;

    // Raycast from the point on the near plane to the point on the far plane and get the
collision result
    var result = simulation.Raycast(vectorNear.XYZ(), vectorFar.XYZ());
    return result.Succeeded;
}
```

> **NOTE**
>
> There are multiple ways to retrieve a reference to this `Simulation` from inside one of your `ScriptComponent`:
>
> - The recommended way is through a reference to a physics component, something like `myRigidBody.Simulation` or `myCollision.Simulation` as it is the fastest.
> - Then through `SceneSystem` by calling `SceneSystem.SceneInstance.GetProcessor<PhysicsProcessor>()?.Simulation`.
> - Or through `this.GetSimulation()`, note that the `this` is required as it is an extension method.

See also

- Colliders

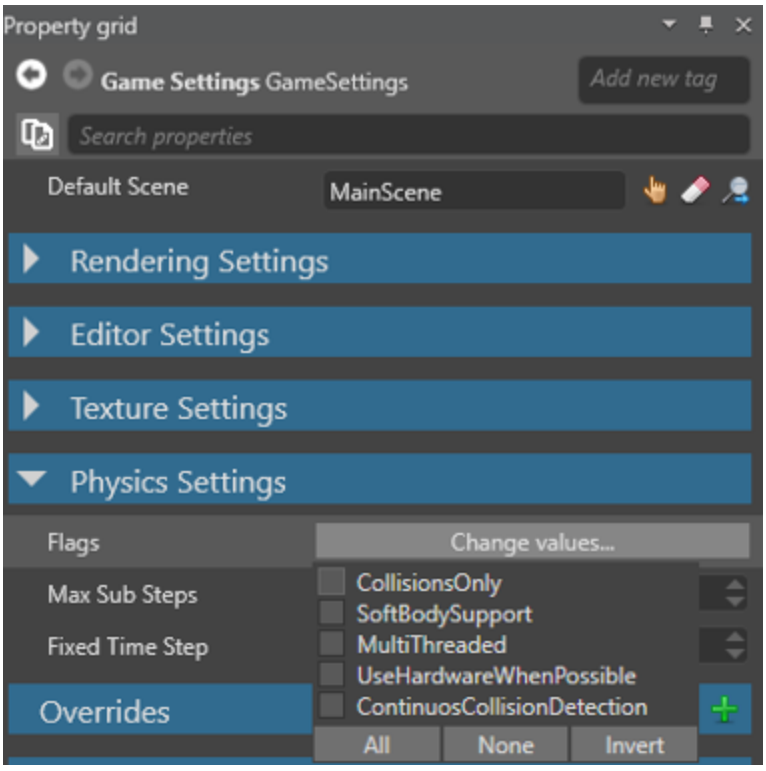
Physics simulation

Intermediate Programmer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

Stride's physics are controlled by the Simulation class. You can change how Stride initializes the simulation by modifying flags in PhysicsSettings, accessed in the **GameSettings** asset properties.

> **NOTE**
>
> Your scene must have at least one Collider in order for Stride to initialize the Simulation instance.



- `CollisionsOnly` initializes the Simulation with collision detection turned on, but no other physics. Objects won't react to physical forces.
- `ContinuousCollisionDetection` initializes the Simulation with continuous collision detection (CCD). CCD prevents fast-moving entities (such as bullets) erroneously passing through other entities.

> **NOTE**
>
> The `SoftBodySupport`, `MultiThreaded`, and `UseHardwareWhenPossible` flags are currently disabled.

At runtime, you can change some Simulation parameters:

- `Gravity` — the global gravity, in world units per second squared
- `FixedTimeStep` — the length of a simulation timestep, in seconds
- `MaxSubSteps` — the maximum number of fixed timesteps the engine takes per update

See also

- Colliders
- Collider shapes

Tutorials

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

- Create a bouncing ball: Use the static collider and rigidbody components to create a ball bouncing on a floor.
- Script a trigger: Create a trigger that doubles the size of a ball when the ball passes through it.

Create a bouncing ball

Beginner Designer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

In this tutorial, we'll use the static collider and rigidbody components to create a ball bouncing on a floor.

> **NOTE**
>
> The screenshots and videos in this tutorial were made using an earlier version of Stride, so some parts of the UI, and the default skybox and sphere, might look different from your version.

1. Create a new project

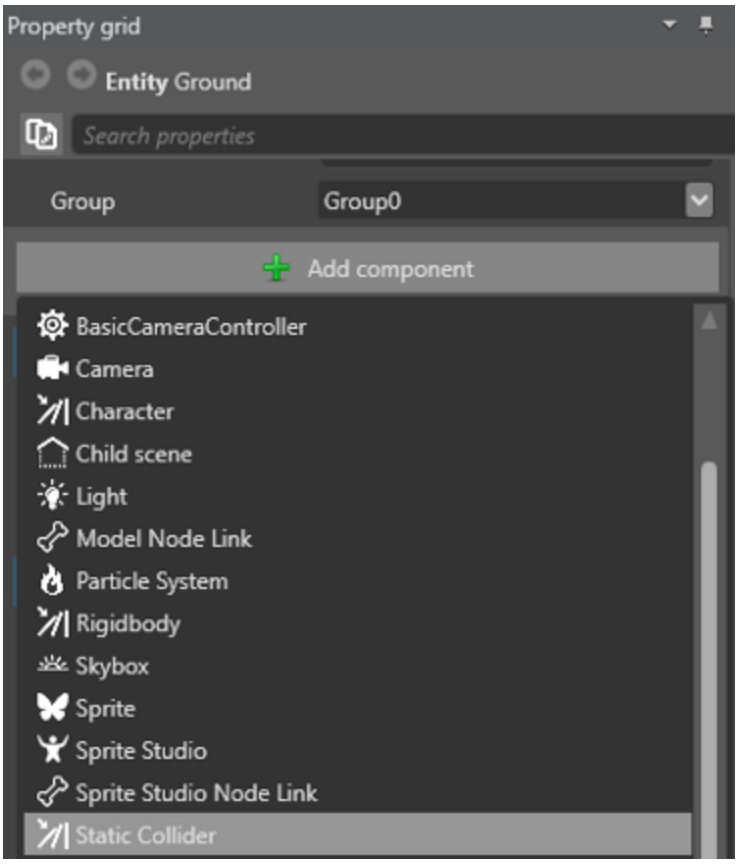
Start a **New Game** project.

The default scene comes pre-loaded with five entities: Camera, Directional light, Skybox, Ground, and Sphere. We're going to add physics components to the **Ground** and **Sphere** entities.

2. Add a static collider

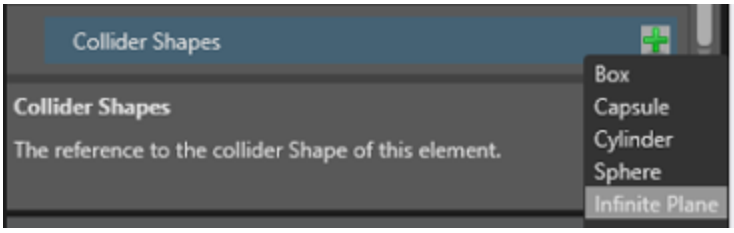
Let's begin by adding a static collider component to the Ground entity. A static collider is a physics object that doesn't move. Typical static colliders are walls, floors, large rocks, and so on. In this case, the static collider will give the ball something to bounce on.

1. Select the **Ground** entity.
2. In the **Property Grid**, click **Add component** and select **Static Collider**.



3. Set the collider shape to match the shape of the entity. To do this, in the **Property Grid**, expand the **Static Collider component** to view its properties.

4. Next to **Collider Shapes**, click (**Add**) and select **Infinite Plane**.



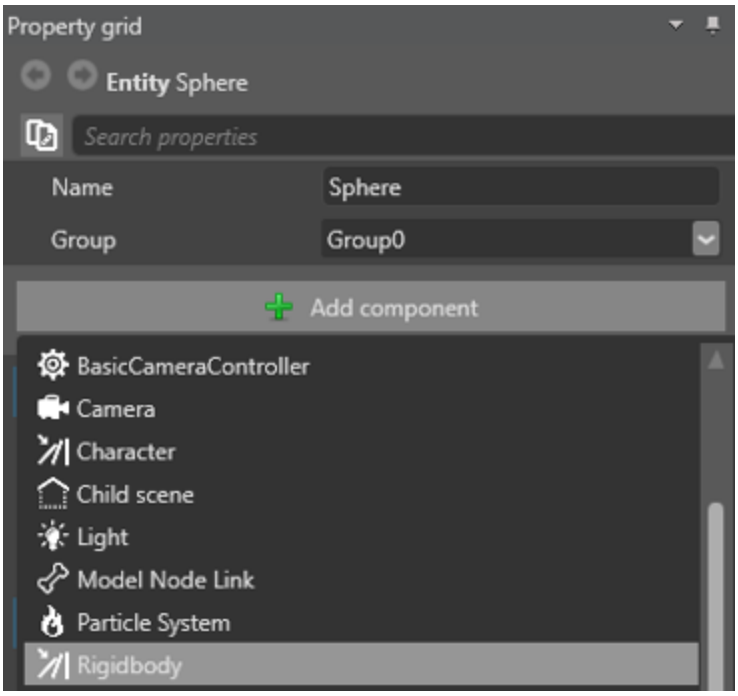
This adds a static collider to the ground, so the ball has something to bounce off.

3. Add a rigidbody collider

Next, we'll add a rigidbody component to the sphere. A rigidbody is a physics object that moves — perfect for our bouncing ball.

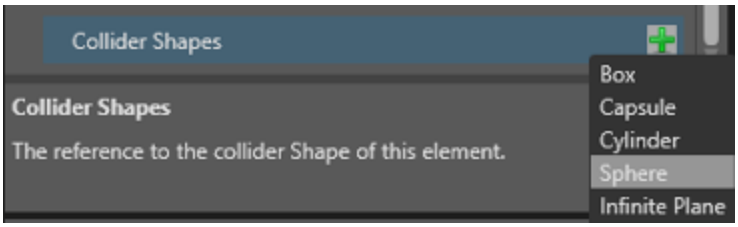
1. In the **Scene Editor**, select the **Sphere** entity.

2. In the **Property Grid**, click **Add component** and select **Rigidbody**.



3. Just like we did for the Ground entity, set the collider shape to match the entity. To do this, in the **Property Grid**, expand the **Rigidbody component** to view its properties.

4. Next to **Collider Shapes**, click (**Add**) and select **Sphere**.

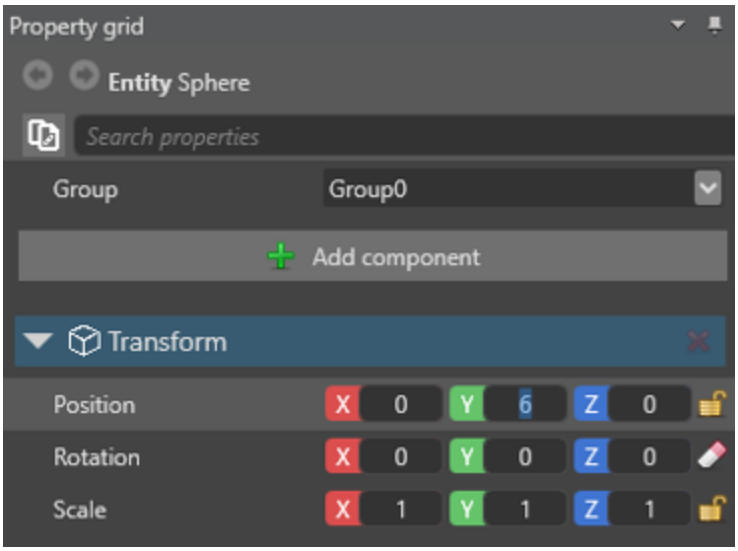


4. Position the ball

Let's position the sphere so it starts in mid-air and falls to the ground.

1. Select the **Sphere** entity.

2. In the **Property Grid**, under **Transform**, set the **Position** to: *X: 0, Y: 6, Z: 0*

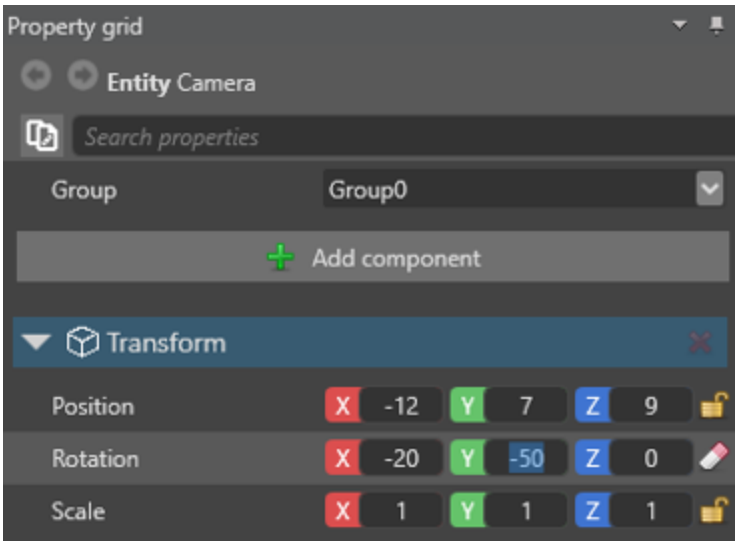


This places the ball in mid-air above the ground.

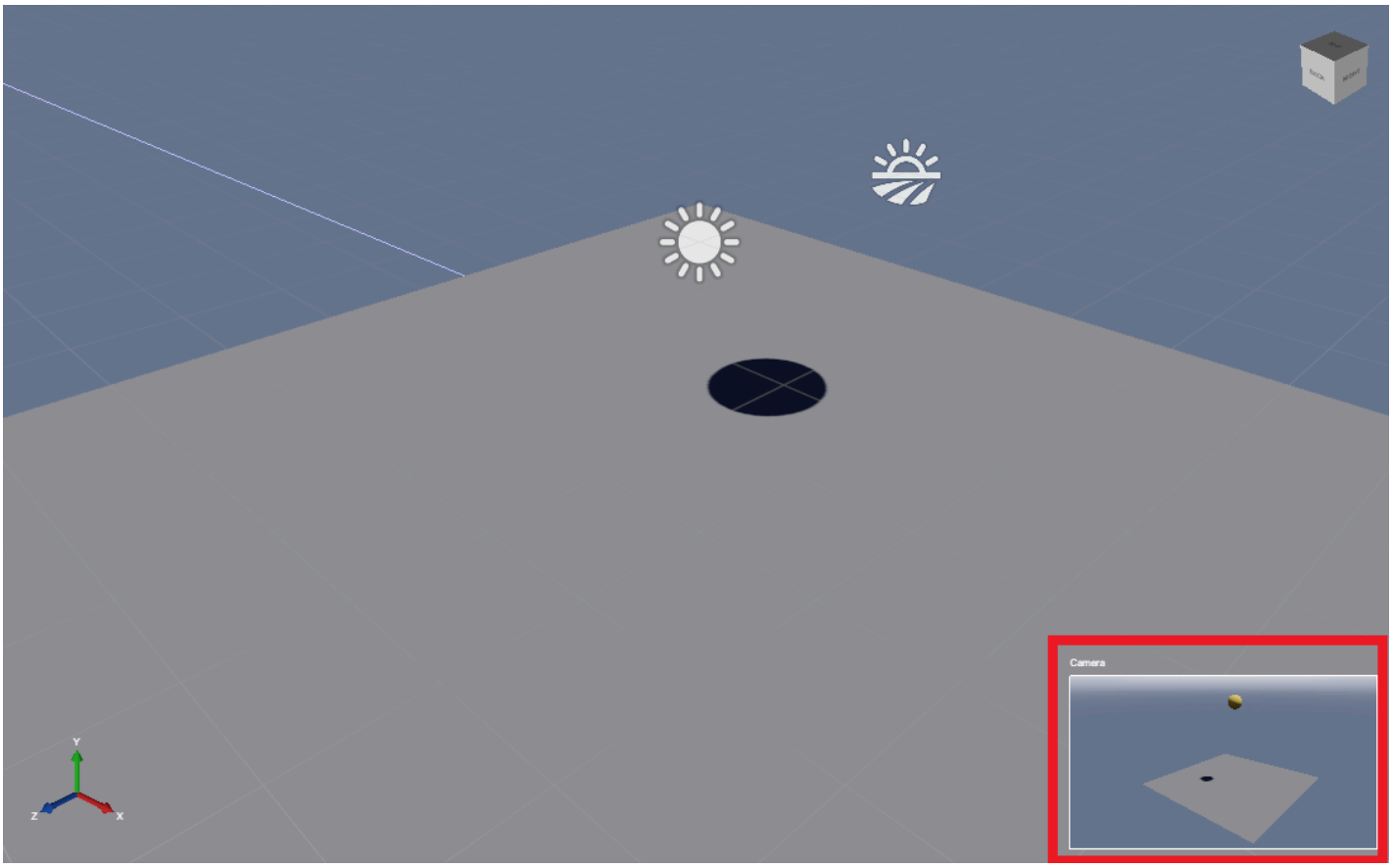
5. Position the camera

Now we'll move the camera to give us a good view of the scene.

1. Select the **Camera** entity.
2. In the **Property Grid**, under **Transform**, set the **Position** to: *X: -12, Y: 7, Z: 9*
3. Set the **Rotation** to: *X: -20, Y: -50, Z: 0*

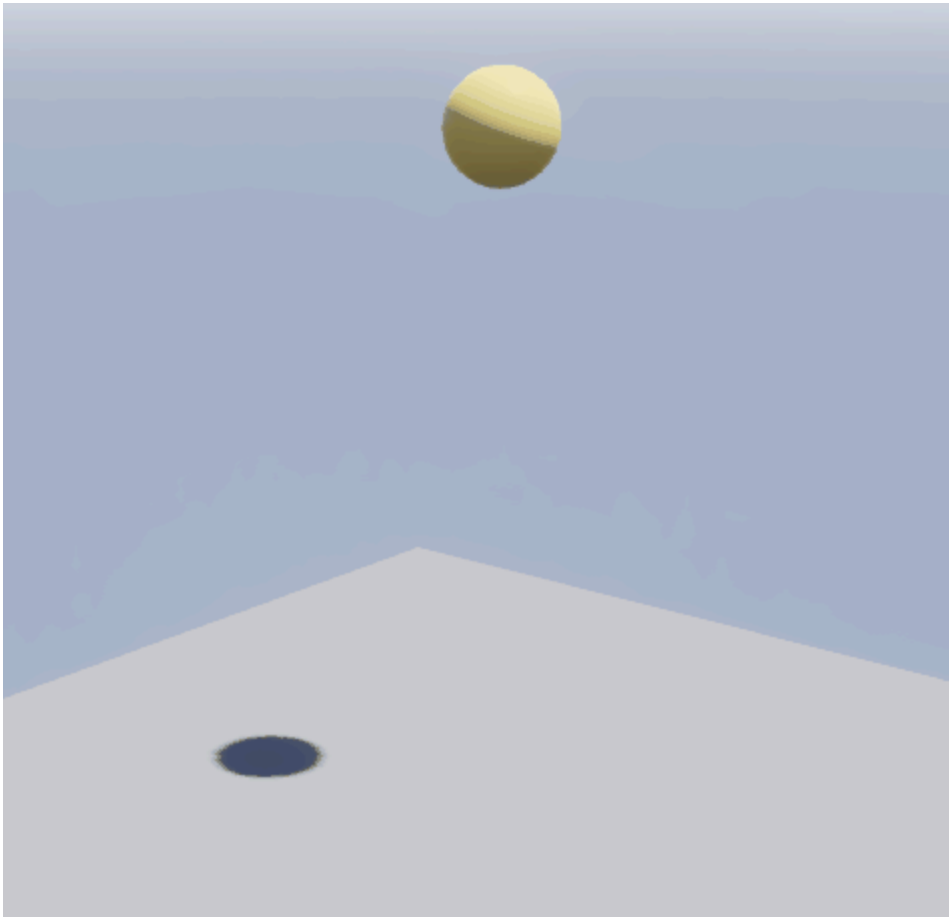


You can see preview the camera view in the **Camera preview** in the bottom-right of the Scene Editor.



6. Set the restitution

Let's see what the scene looks like so far. To run the project, press **F5**.



The Sphere (rigidbody) responds to gravity and falls. The Ground (static collider) breaks its fall. But there's no bounce effect yet.

To create a bounce effect, we need to change the **restitution** of the Sphere and the Ground. This simulates the coefficient of restitution (Wikipedia) of real-world collisions.

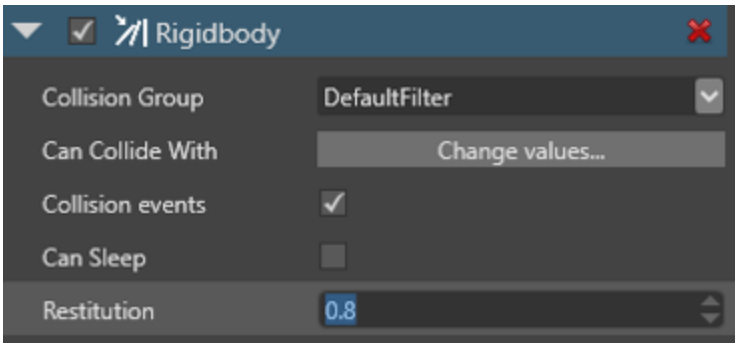
- If the restitution property of colliding entities is 0, the entities lose all energy and stop moving immediately on impact.
- If the restitution is 1, they lose no energy and rebound with the same velocity at which they collided.
- If the restitution is higher than 1, they gain energy and rebound with *more* velocity.

As a rule, to create realistic collisions, set the restitution between 0 and 1.

Let's set the restitution of our Sphere and Ground entities.

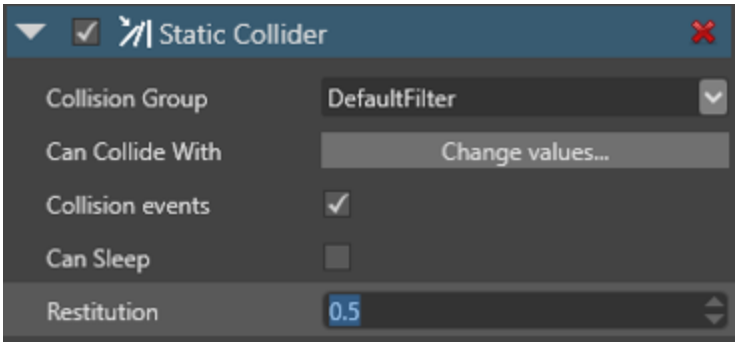
1. Select the **Sphere** entity.

2. In the **Property Grid**, under **Rigidbody**, set the **Restitution** to 0.8.

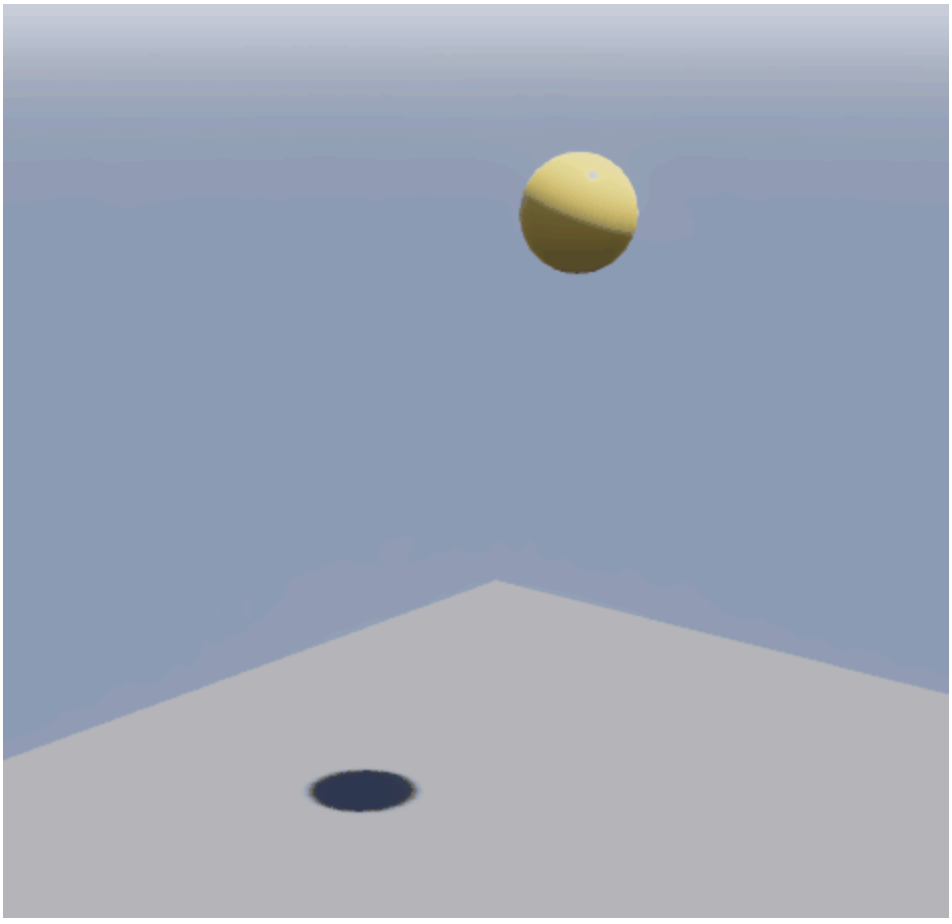


3. Select the **Ground** entity.

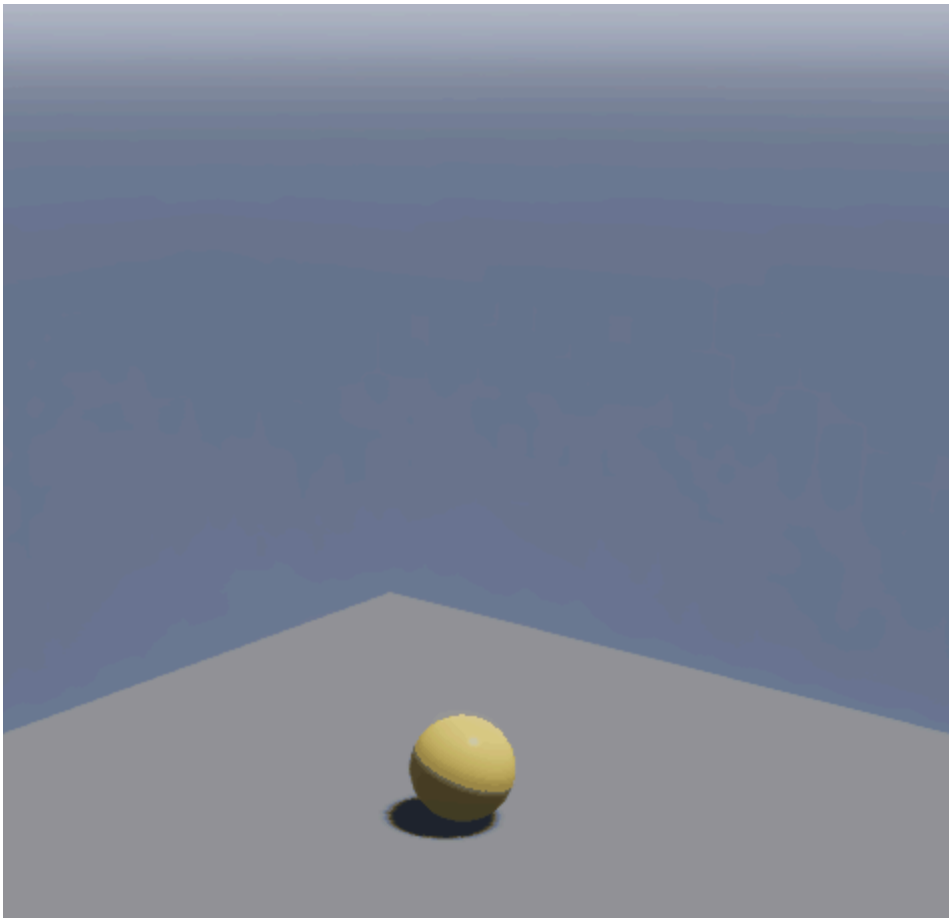
4. In the **Property Grid**, under **Static Collider**, set the **Restitution** to 0.5.



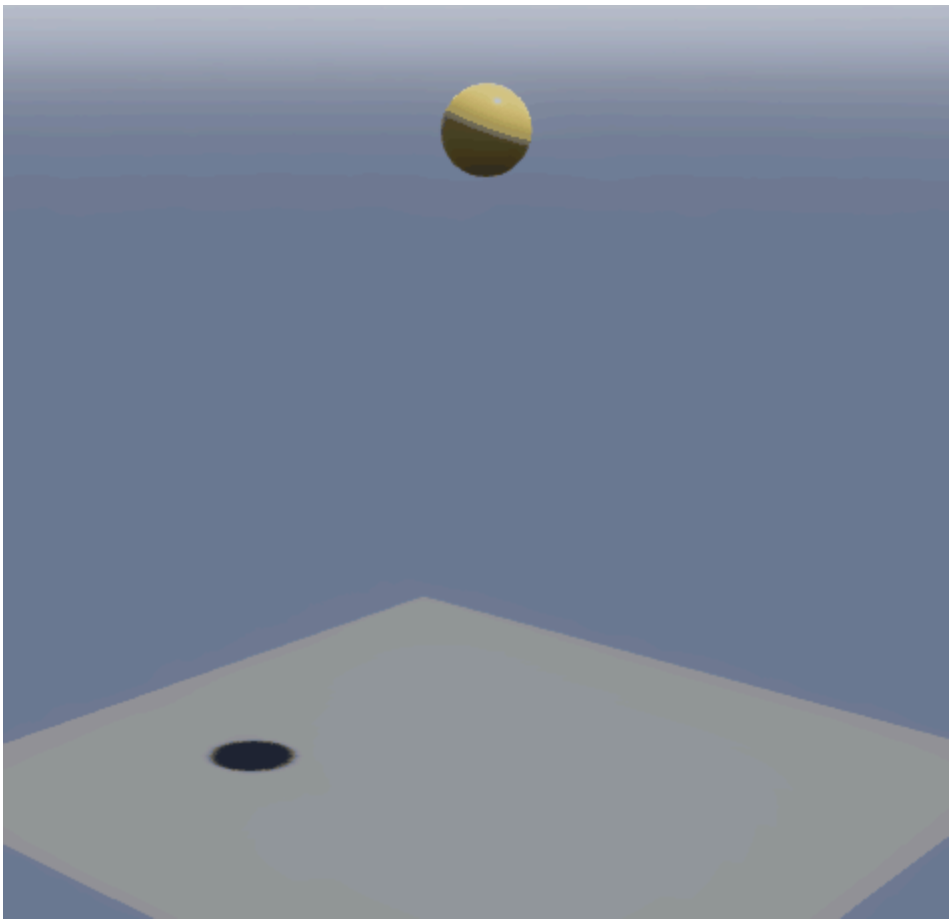
To see how this changes the physics, run the project again (**F5**). This time, the ball bounces on the ground before coming to a stop:



Try changing the restitution of both entities to 1. This creates a ball that bounces indefinitely, losing no energy:



Set the restitution to 1.1 and the ball bounces a little higher each time:



Now we've created a bouncing ball, we can use it to learn about triggers. For more information, see the Script a trigger tutorial.

See also

- Colliders
- Collider shape
- Tutorial: Script a trigger

Tutorial: Script a trigger

Beginner Designer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

In this tutorial, we'll create a trigger that doubles the size of a ball when the ball passes through it.

> **NOTE**
>
> The screenshots and videos in this tutorial were made using an earlier version of Stride, so some parts of the UI, and the default skybox and sphere, might look different from your version.

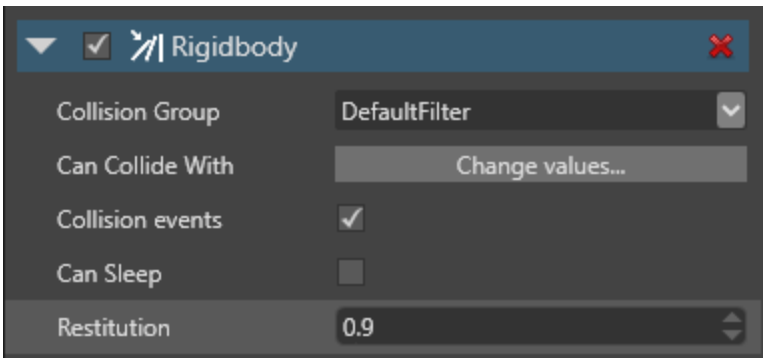
1. Create a bouncing ball

Follow the instructions in the Create a bouncing ball tutorial. This creates a simple scene in which a ball falls from mid-air, hits the ground, and bounces.

2. Set the restitution

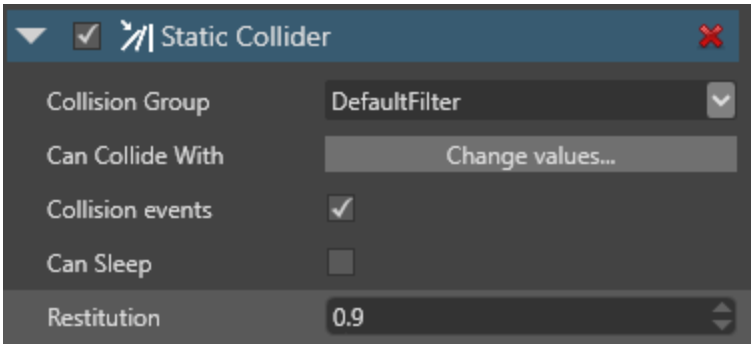
For this tutorial, we'll set the restitution of both the ground and the sphere to 0.9, which makes the ball very bouncy. This makes it easier to see the effect of the trigger later, as the ball will bounce in and out of the trigger area repeatedly.

1. Select the **Sphere** entity.
2. In the **Property Grid**, under **Rigidbody**, set the **Restitution** to *0.9*.



3. Select the **Ground** entity.

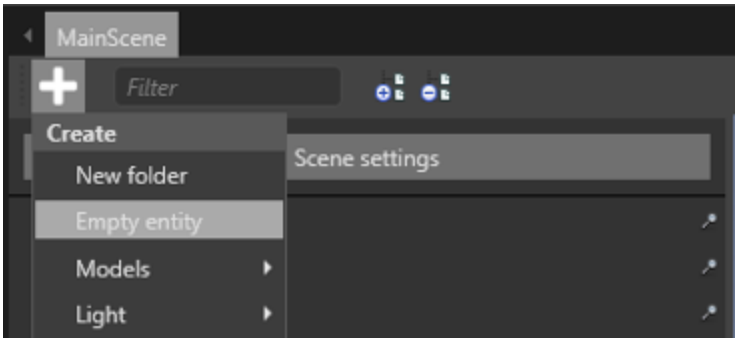
4. In the **Property Grid**, under **Static Collider**, set the **Restitution** to *0.9*.



3. Add a trigger

Now we'll add a trigger between the ball and the ground, so the ball passes through it.

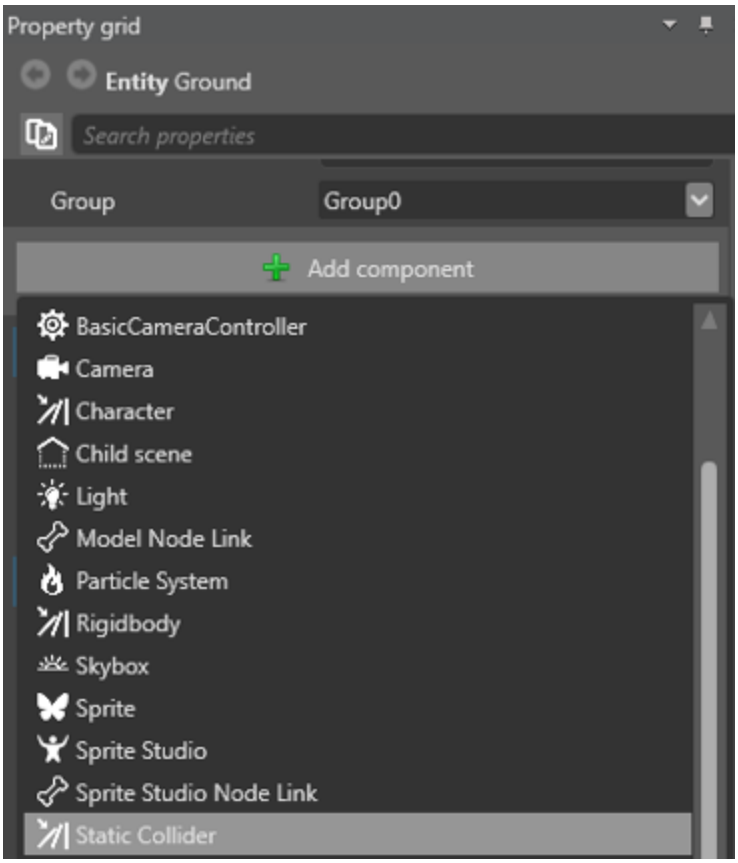
1. In the **Scene Editor**, click the white plus button (**Create new entity**) and select **Empty entity**.



   Game Studio adds an entity to the scene with the default name **Entity**.

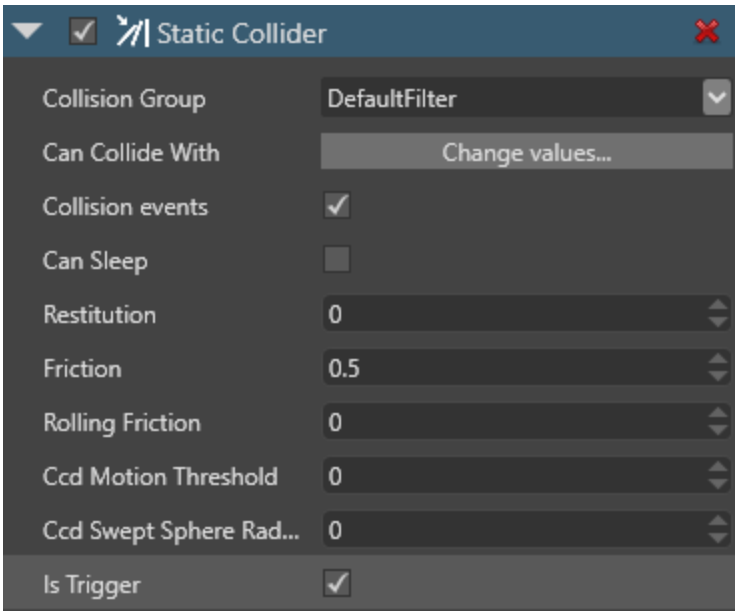
2. This entity will be our trigger, so rename it *Trigger* to make it easy to identify.

3. Since we don't need the trigger to move, we'll make it a static collider. In the **Property Grid**, click **Add component** and select **Static Collider**.



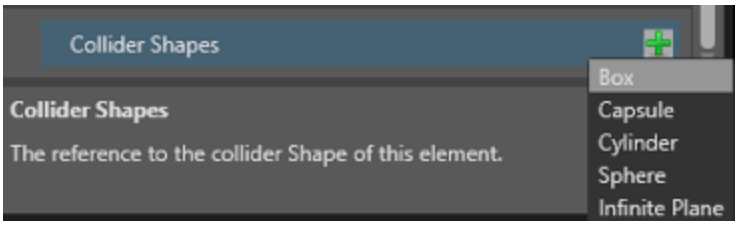
4. In the **Property Grid**, expand the **Static Collider component** to view its properties.

5. Select the **Is Trigger** checkbox.

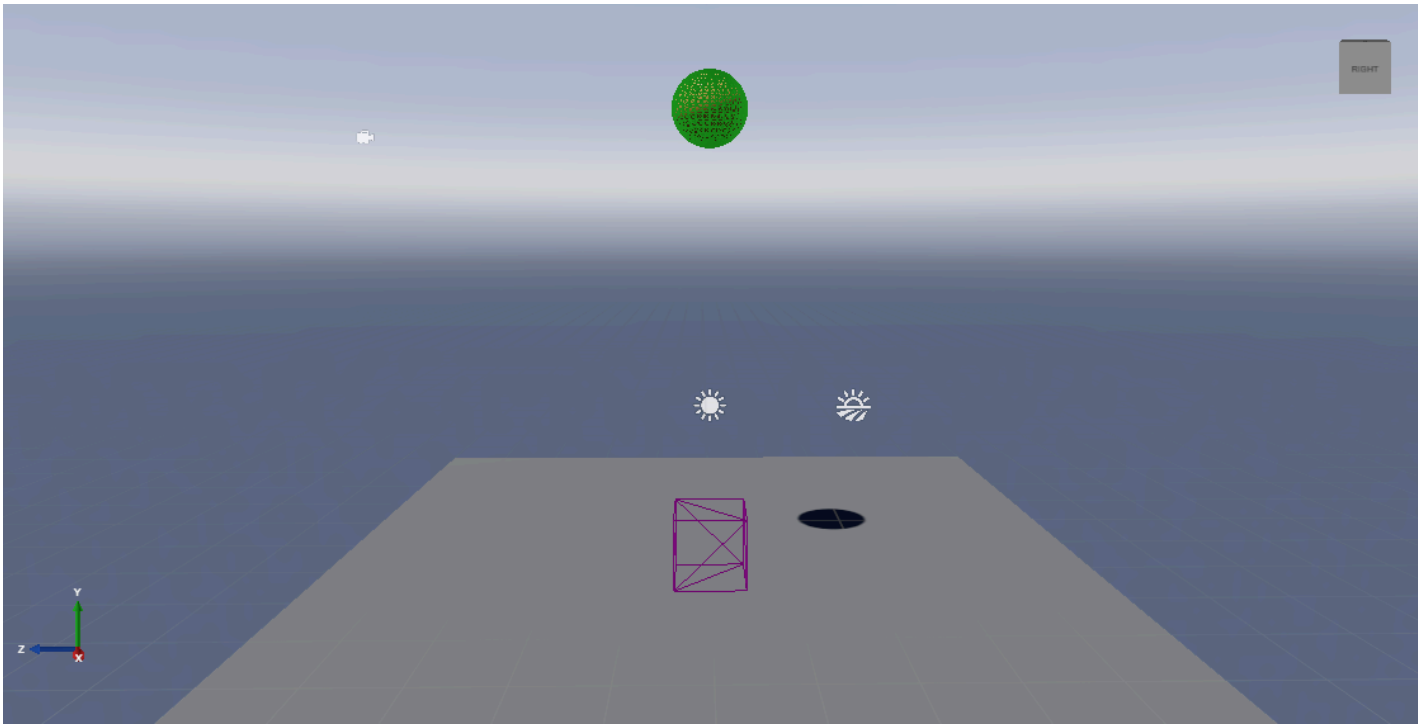


   This makes the collider a trigger. This means objects can pass through it, but are still detected in the code.

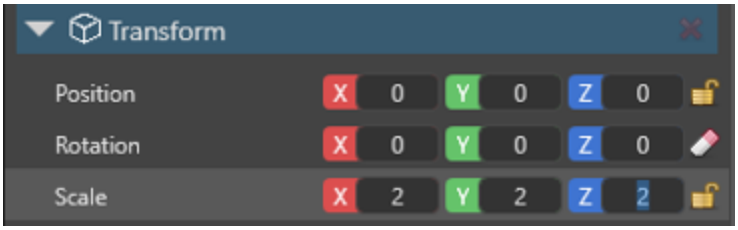
6. We need to give the trigger a shape. Next to **Collider Shapes**, click (**Add**) and select **Box**.



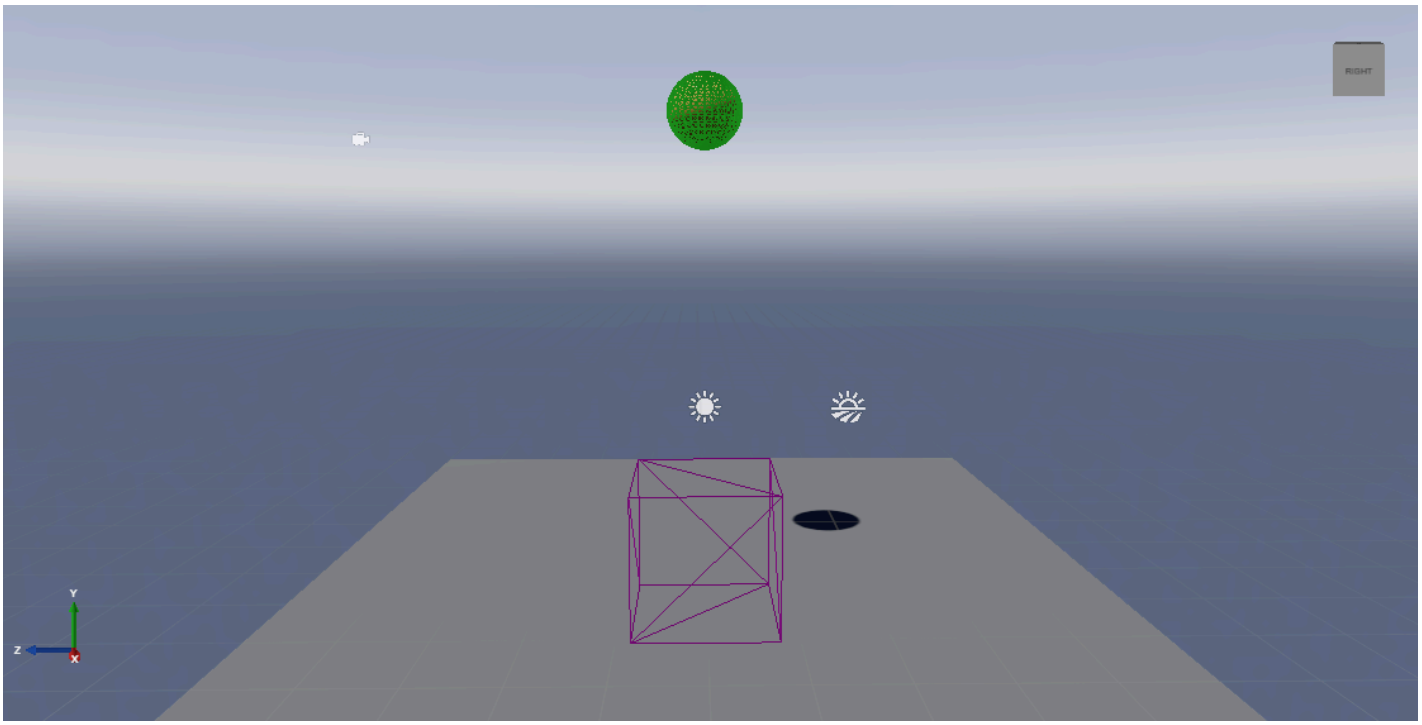
This gives the trigger a box shape.



7. Let's make the trigger a larger area. In the **Property Grid**, under the **Transform** component properties, set the **scale** to: *X:2, Y:2, Z:2*



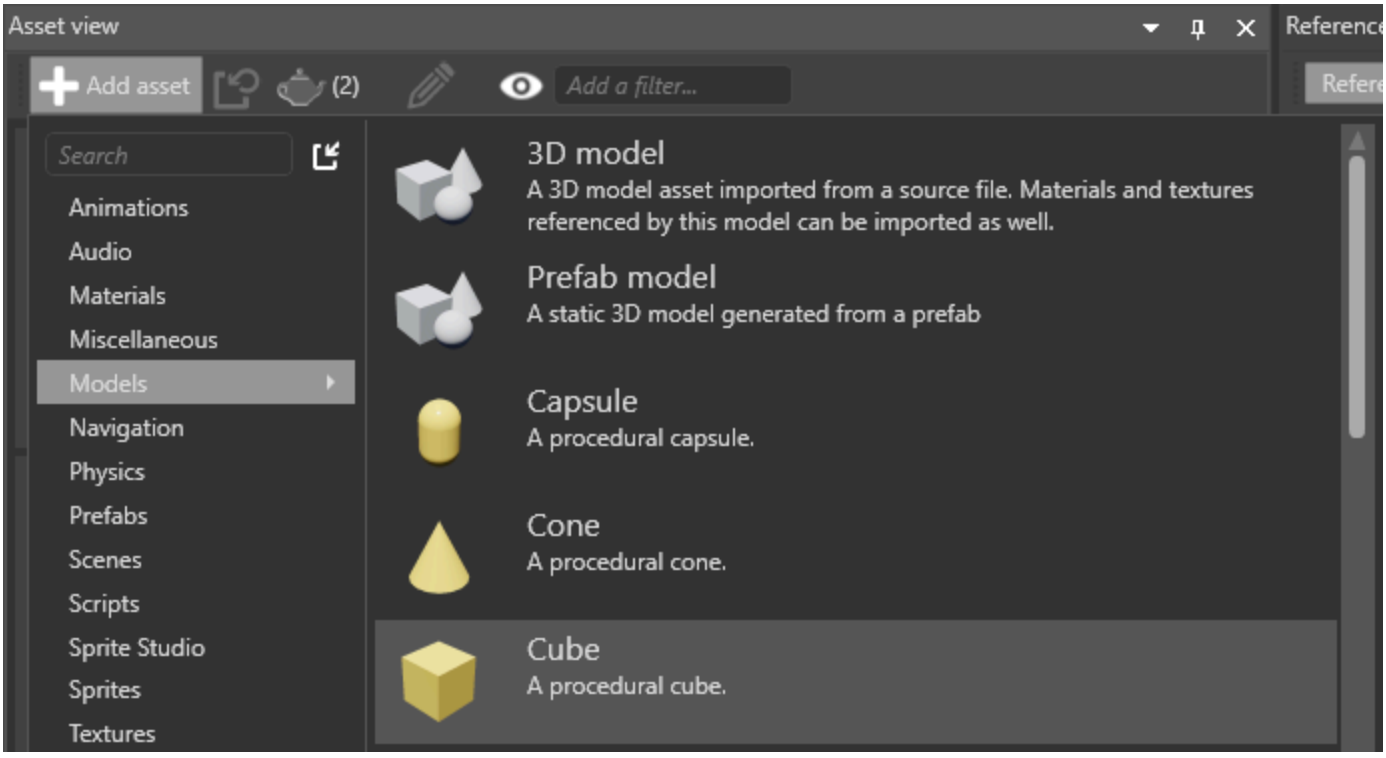
This doubles the size of the trigger.



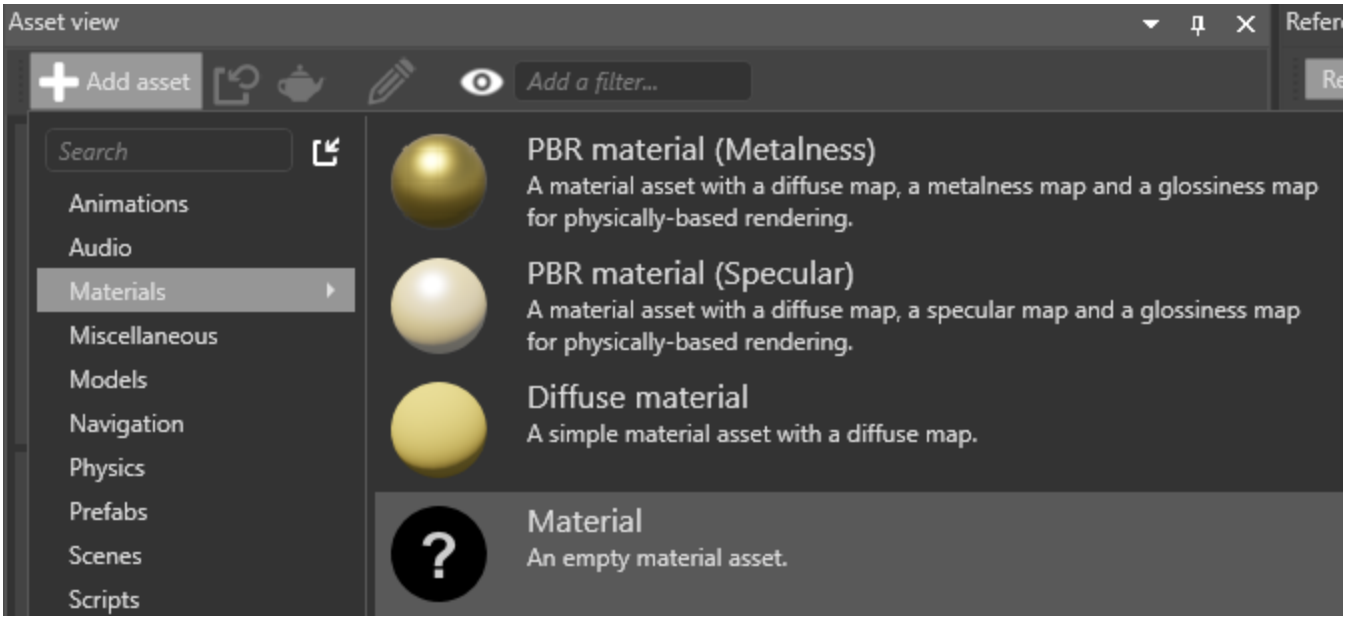
4. Give the trigger a model

Right now, the trigger is invisible at runtime. To better show how the trigger works, we'll make it a transparent box. This has no effect on how the trigger works; it just means we can easily see where it is at runtime.

1. Create a new procedural model asset. To do this, in the **Asset View**, click **Add asset**, and select **Models > Cube**.

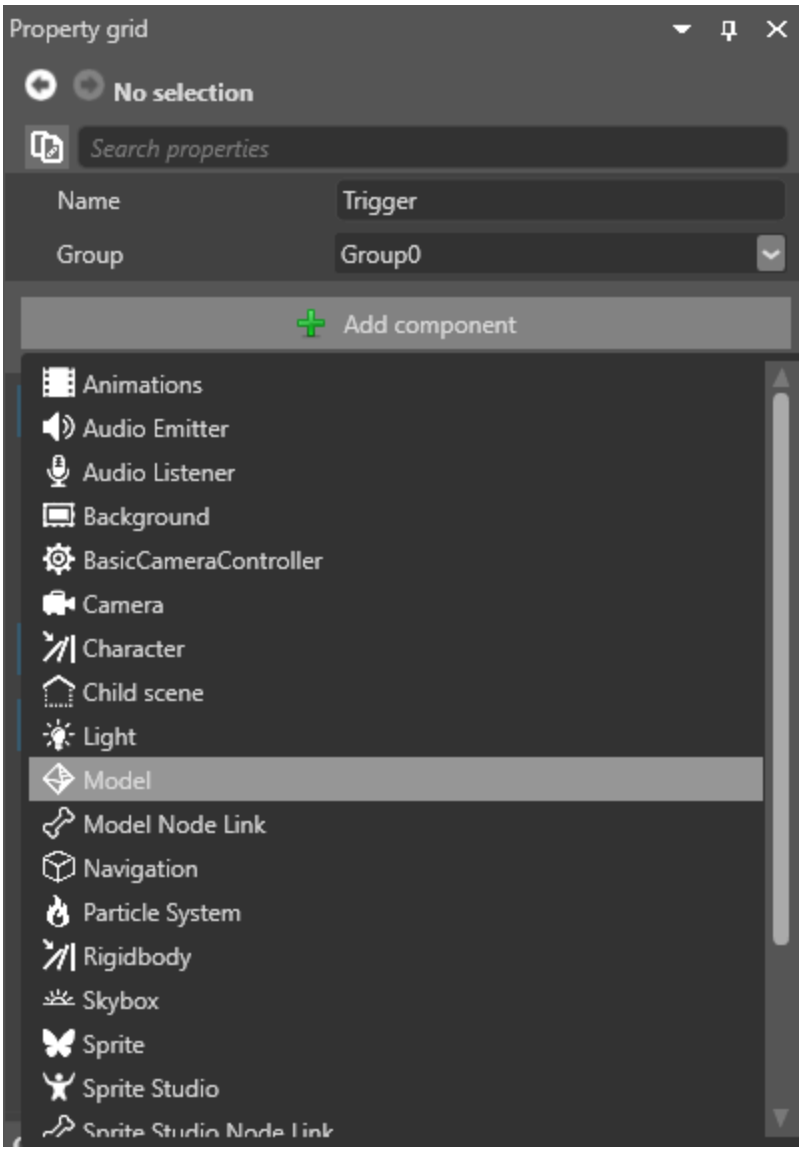


2. Create a new empty material asset. To do this, in the **Asset View**, click **Add asset**, and select **Materials > Material**.



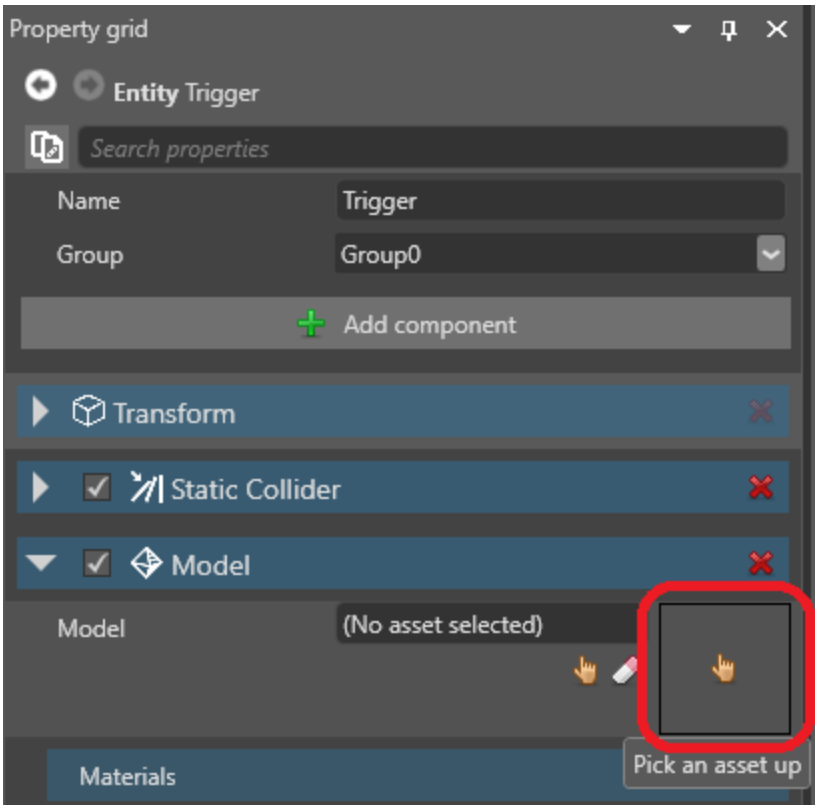
3. Let's rename the material to make it easy to identify. To do this, right-click, select **Rename**, and type a new name (eg *Transparent*).

4. Select the **Trigger** entity. In the **Property Grid**, click **Add component** and select **Model**.

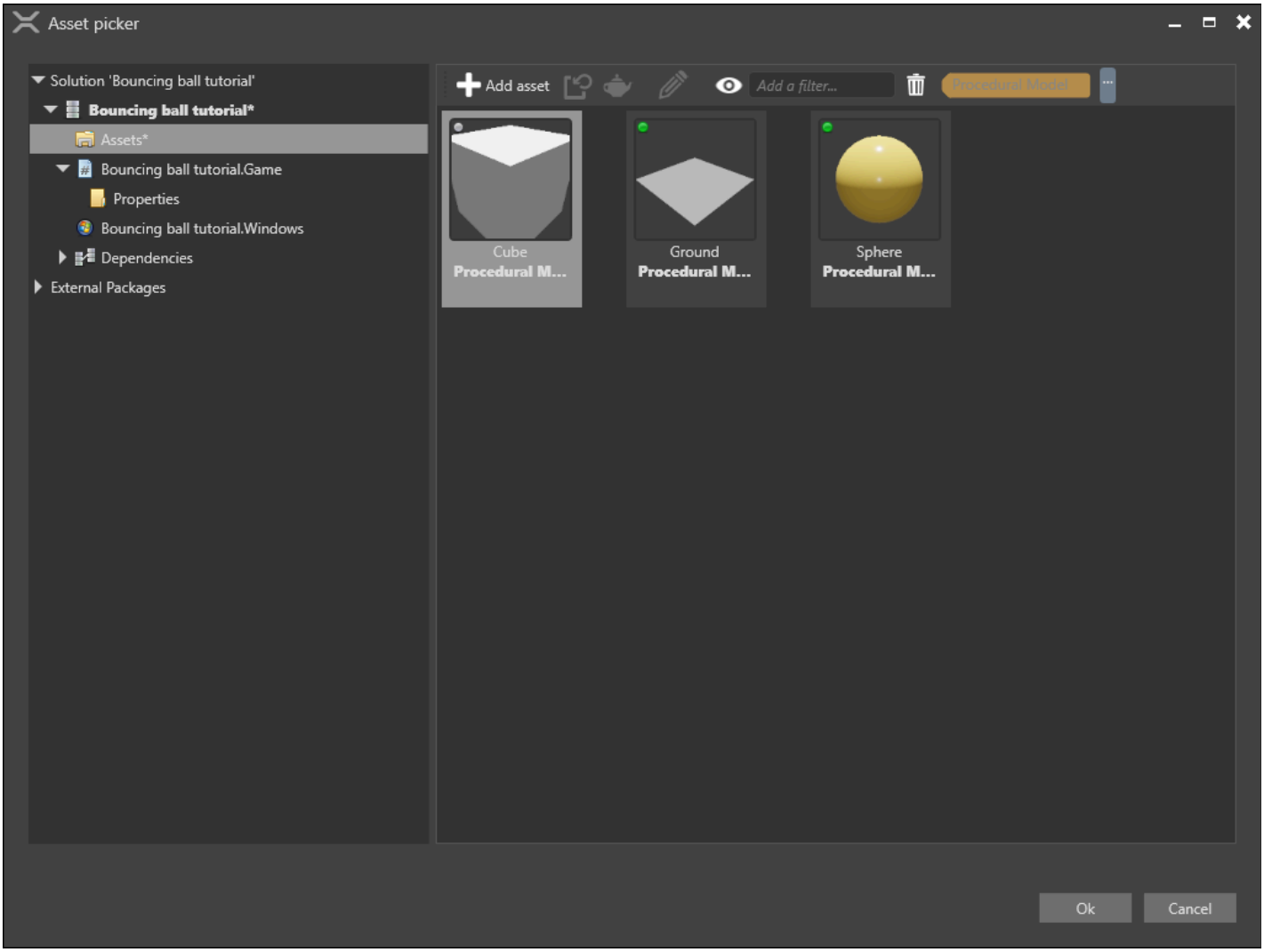


Game Studio adds a model component to the entity.

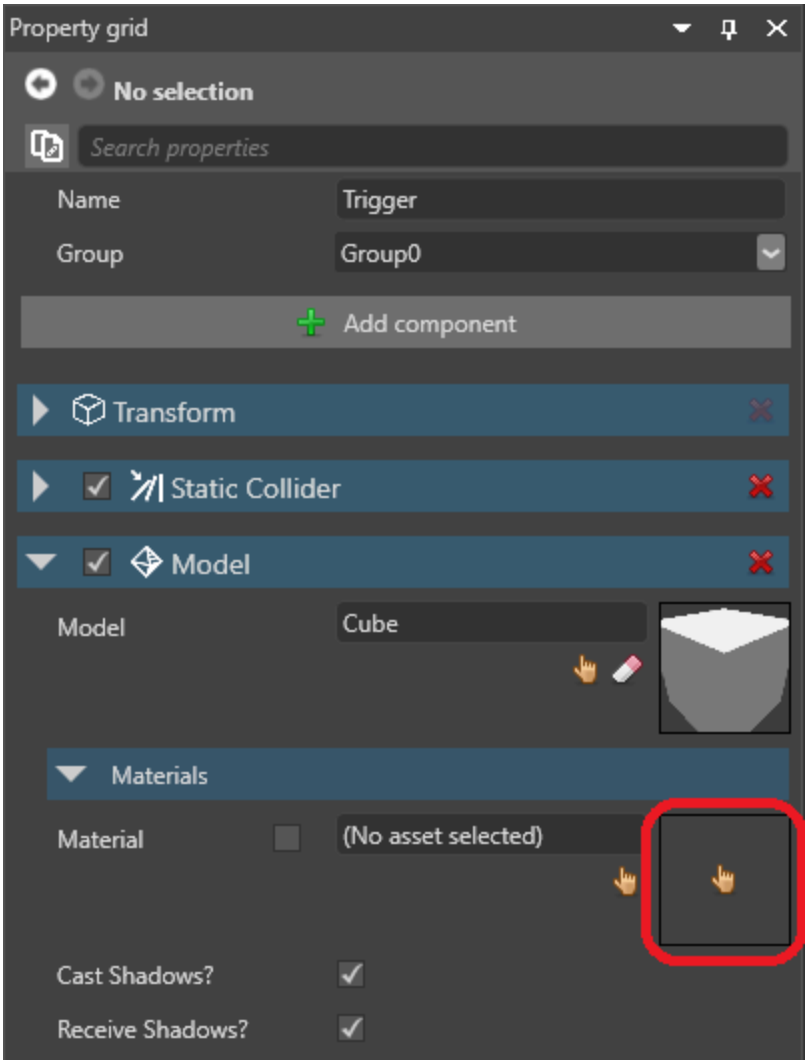
5. Under **Model**, click (**Select an asset**).



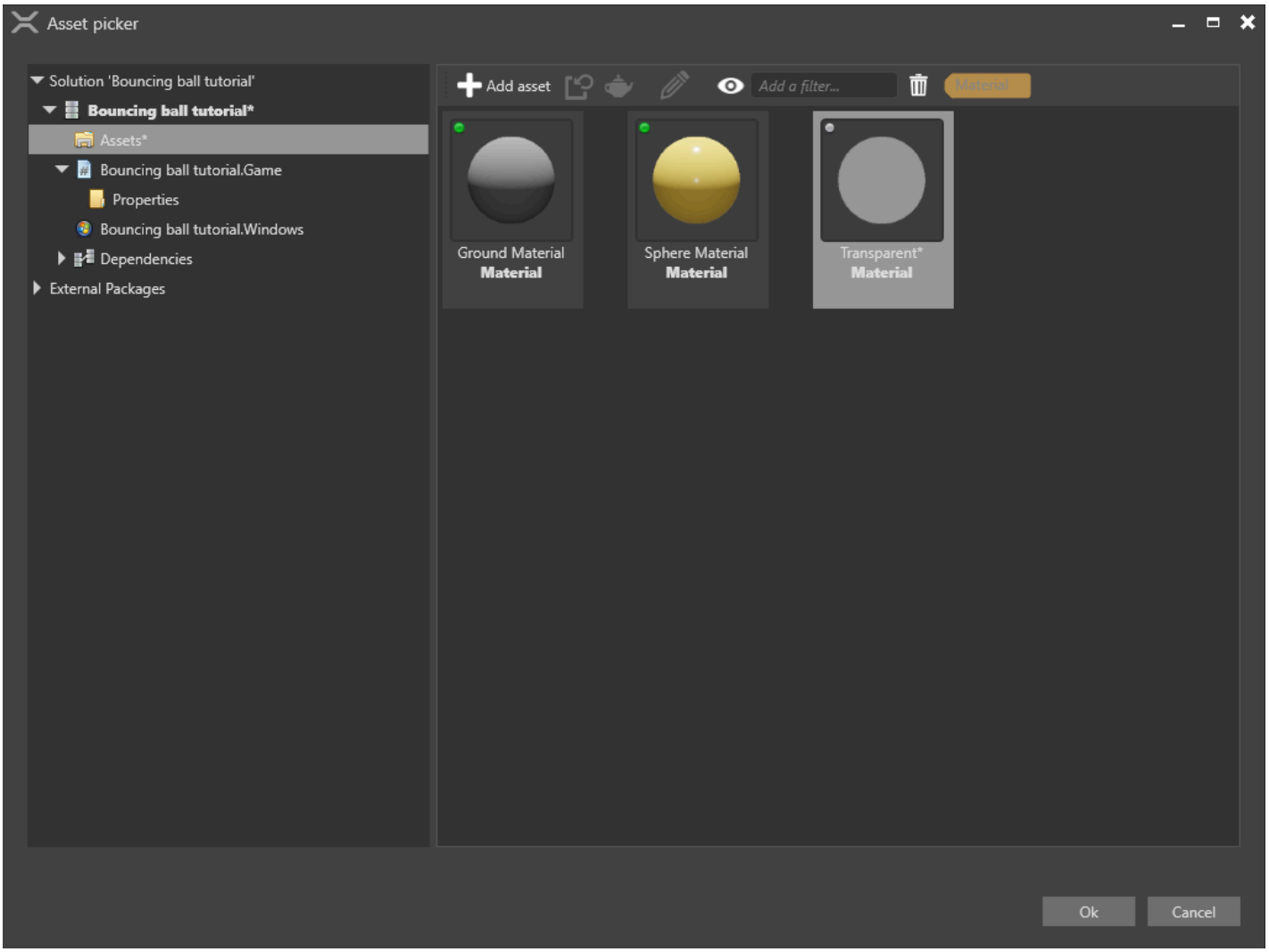
6. Select the **Cube** model we created in step 1 and click **OK**.



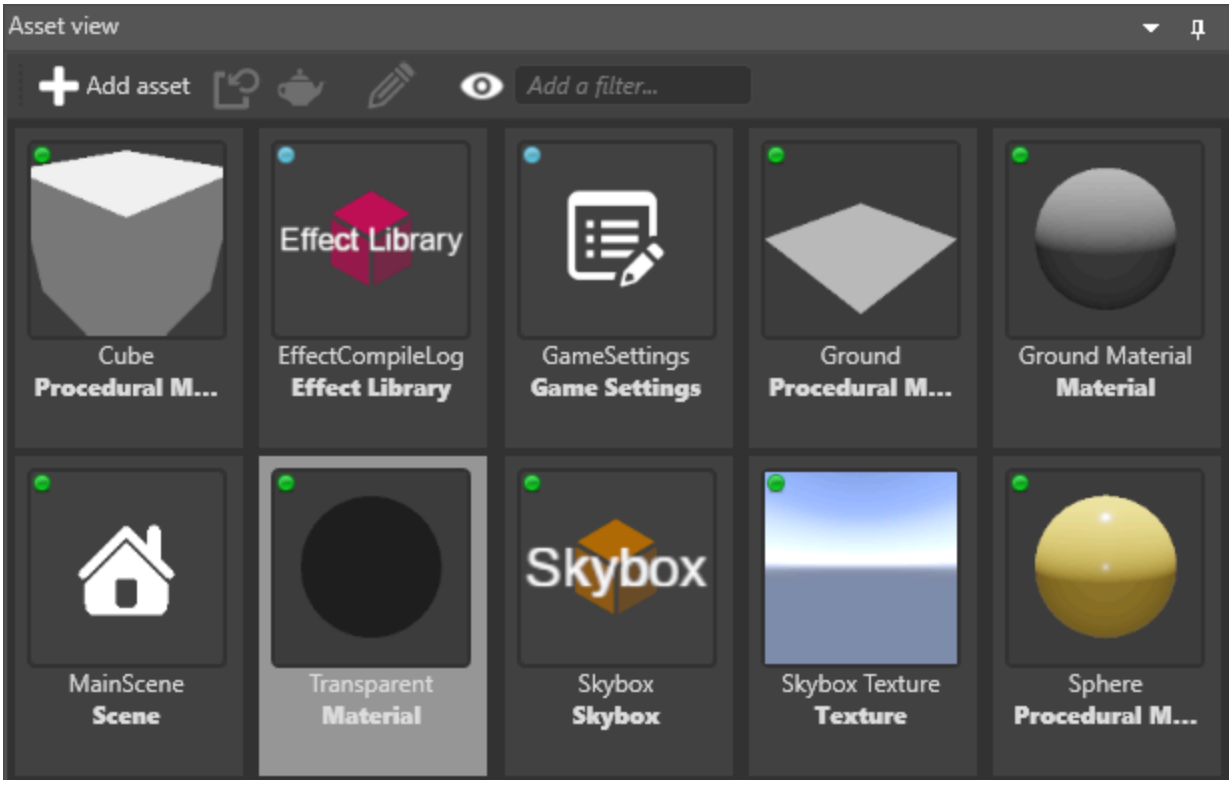
7. In the **Property Grid**, under **Model > Materials**, click (**Select an asset**).



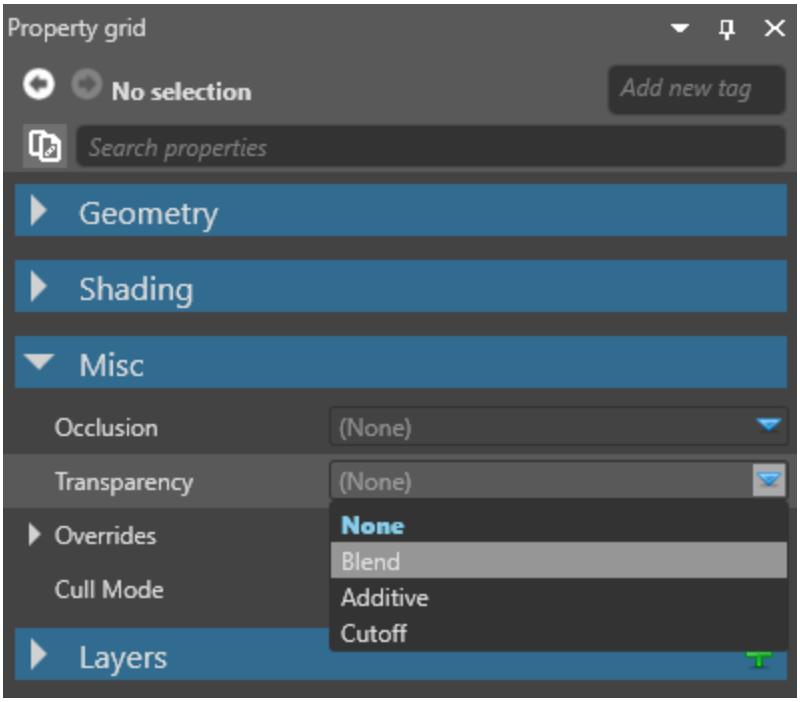
8. Select the **Transparent** material we created in step 2 and click **OK**.



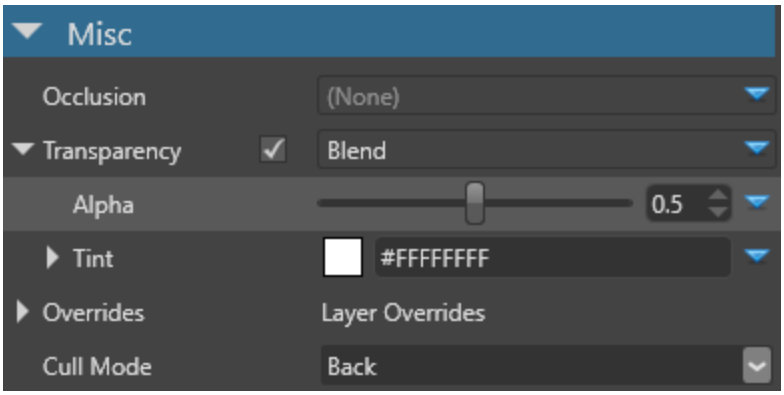
9. In the **Asset View**, select the **Transparent** material asset.



10. In the **Property Grid**, under **Misc > Transparency**, select **Blend**.



11. By default, the Alpha is set to 1. This makes the material completely opaque. To set it to 50% opacity, set the **Alpha** to 0.5.



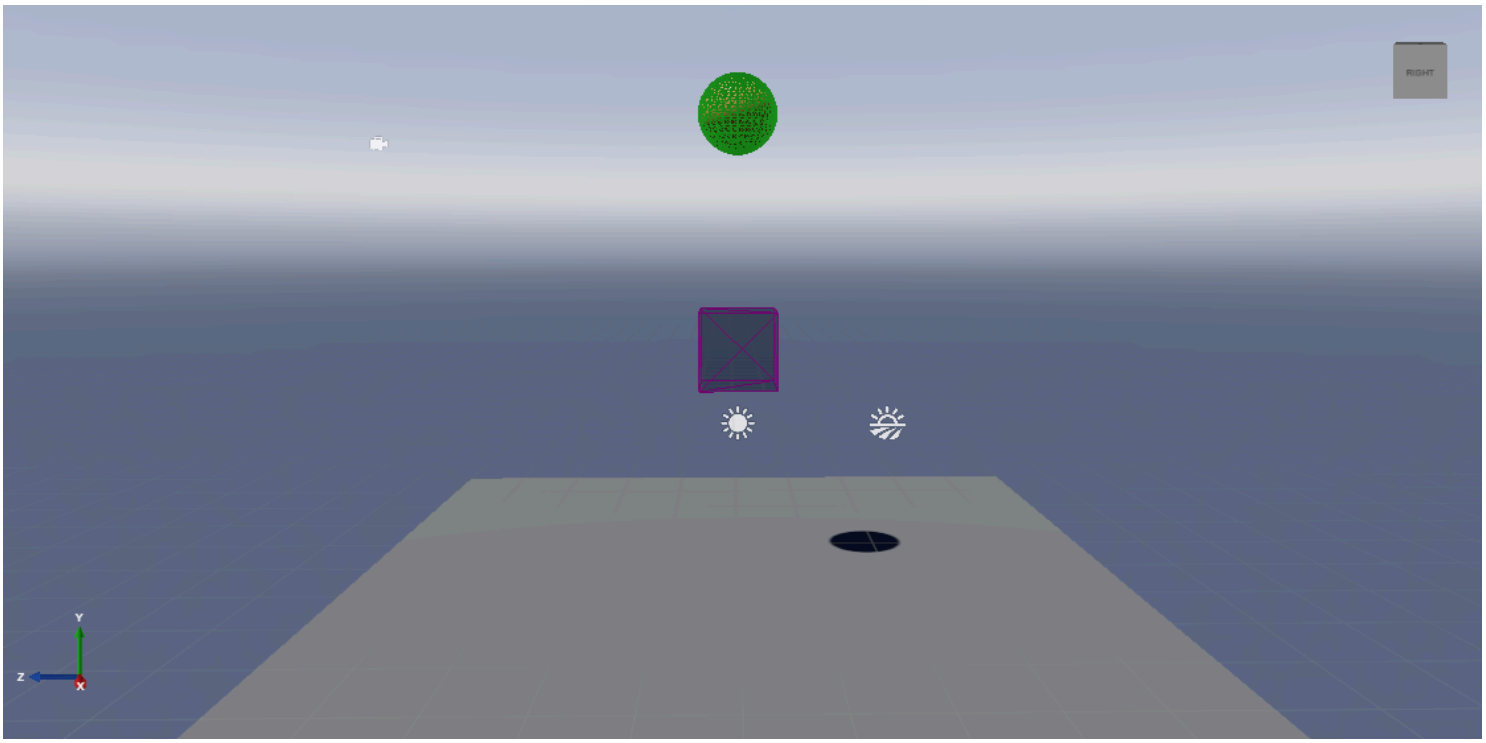
Now the trigger area will be visible at runtime.

5. Position the trigger

We need to position the trigger between the ground and the sphere, so the ball falls through it.

In the **Property Grid**, under **Transform**, set the **Position** to: *X:0, Y:3, Z:0*

Now the trigger entity is between the ground and the sphere:



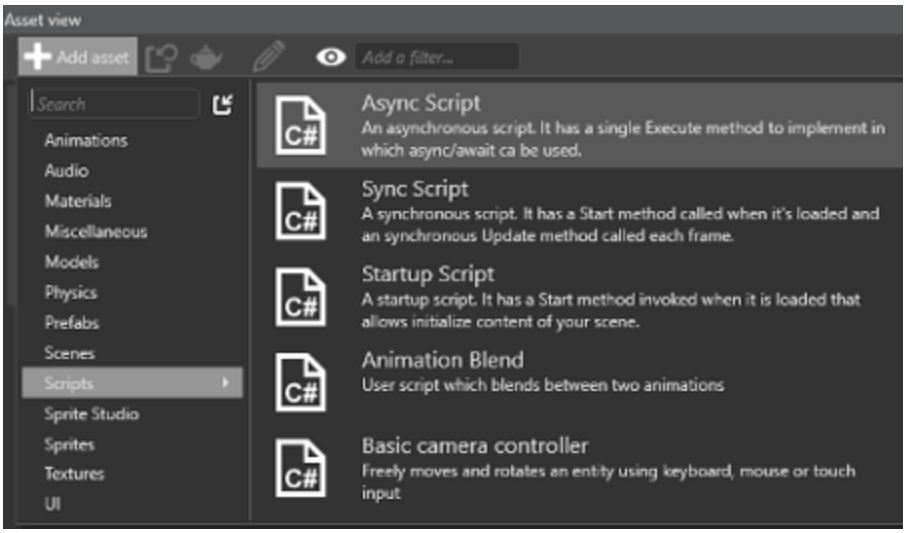
6. Change the sphere size with script

If we run the project now (**F5**), the ball falls through the trigger, but nothing happens.

Let's write a script to change the size of the ball when it enters the trigger.

> **NOTE**
> For more information about scripts, see Scripts.

1. In the **Asset View**, click **Add asset** and select **Scripts** > **Async Script**.



2. In the **Create a script** dialog, name your script *Trigger* and click **Create script**.

   2a. If Game Studio asks if you want to save your script, click **Save**.

   2b. If Game Studio asks if you want to reload the assemblies, click **Reload**.

3. Open the script, replace its content with the following code, and save the file:

```
using Stride.Engine;
using Stride.Physics;
using System.Threading.Tasks;
using Stride.Core.Mathematics;

namespace TransformTrigger
// You can use any namespace you like for this script.
{
    public class Trigger : AsyncScript
    {
        public override async Task Execute()
        {
            var trigger = Entity.Get<PhysicsComponent>();
            trigger.ProcessCollisions = true;

            // Start state machine
            while (Game.IsRunning)
            {
                // 1. Wait for an entity to collide with the trigger
                var firstCollision = await trigger.NewCollision();

                var otherCollider = trigger == firstCollision.ColliderA
                    ? firstCollision.ColliderB
                    : firstCollision.ColliderA;
                otherCollider.Entity.Transform.Scale = new Vector3(2.0f, 2.0f, 2.0f);
```

```
                    // 2. Wait for the entity to exit the trigger
                    await firstCollision.Ended();

                    otherCollider.Entity.Transform.Scale= new Vector3(1.0f, 1.0f, 1.0f);
                }
            }
        }
    }
```

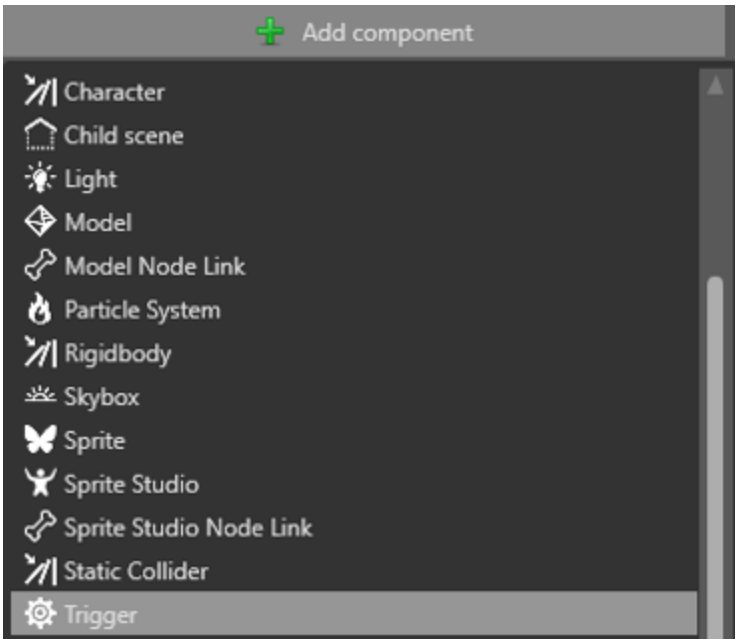
This code doubles the size (scale) of any entity that enters the trigger. When the entity exits the trigger, it returns to its original size.

4. Reload the assemblies.

7. Add the script

Finally, let's add this script to the trigger entity as a component.

1. In **Game Studio**, select the **Trigger** entity.
2. In the **Property Grid**, click **Add component** and select the **Trigger** script.



8. Run the project

Run the project (**F5**) to see the trigger in action.

The ball falls through the trigger, doubles in size, exits the trigger, and returns to its normal size.

More ideas

You can alter the script to make other changes when the sphere enters the trigger.

For example, you can switch the material on the sphere entity. This script switches the material on the Sphere entity from the **Sphere Material** to the **Ground Material** and back again:

```
using Stride.Engine;
using Stride.Physics;
using System.Threading.Tasks;
using Stride.Core.Mathematics;
using Stride.Rendering;

namespace TransformTrigger
// You can use any namespace you like for this script.
{
    public class Trigger : AsyncScript
    {
        private Material material1;
        private Material material2;

        public override async Task Execute()
        {
            var trigger = Entity.Get<PhysicsComponent>();
            trigger.ProcessCollisions = true;

            // Make sure the materials are loaded
            material1 = Content.Load<Material>("Sphere Material");
            material2 = Content.Load<Material>("Ground Material");

            // Start state machine
            while (Game.IsRunning)
            {
                // 1. Wait for an entity to collide with the trigger
                var firstCollision = await trigger.NewCollision();

                var otherCollider = trigger == firstCollision.ColliderA
```

```
                    ? firstCollision.ColliderB
                    : firstCollision.ColliderA;

                // 2. Change the material on the entity
                otherCollider.Entity.Get<ModelComponent>().Materials[0] = material2;

                // 3. Wait for the entity to exit the trigger
                await firstCollision.Ended();

                // 4. Change the material back to the original one
                otherCollider.Entity.Get<ModelComponent>().Materials[0] = material1;
            }
        }

        public override void Cancel()
        {
            Content.Unload(material1);
            Content.Unload(material2);
        }
    }
}
```

See also

- Tutorial: Create a bouncing ball
- Colliders
- Collider shapes
- Scripts

Fix physics jitter

Beginner Programmer

> **WARNING**
>
> Bullet Physics is being phased out. We no longer plan to support or expand its features as our focus shifts to Bepu Physics. We recommend transitioning to Bepu Physics for access to the latest updates and ongoing improvements.

In Stride, there is no default smoothing applied to entities that are attached to physics entities. This can cause noticeable jitter, especially if the camera is attached to a character component.

In this tutorial, we will explore how to add smoothing to an entity using a SyncScript.

> **NOTE**
>
> You can also decrease the `FixedTimeStep` in the physics settings configuration to achieve more accurate physics simulations. For example, changing it from `0.016667` to `0.008` will increase accuracy but at the cost of higher CPU usage.

Code to handle smoothing between two entities

The following code is all that's needed to smoothly attach two entities. Ensure that you unparent the entity you are trying to smooth, otherwise the transform processor will override this script.

```
[ComponentCategory("Utils")]
[DataContract("SmoothFollowAndRotate")]
public class SmoothFollowAndRotate : SyncScript
{
    public Entity EntityToFollow { get; set; }
    public float Speed { get; set; } = 1;

    public override void Update()
    {
        var deltaTime = (float)Game.UpdateTime.Elapsed.TotalSeconds;
        var currentPosition = Entity.Transform.Position;
        var currentRotation = Entity.Transform.Rotation;

        var lerpSpeed = 1f - MathF.Exp(-Speed * deltaTime);
```

```
        EntityToFollow.Transform.GetWorldTransformation(out var otherPosition, out var
otherRotation, out var _);

        var newPosition = Vector3.Lerp(currentPosition, otherPosition, lerpSpeed);
        Entity.Transform.Position = newPosition;

        Quaternion.Slerp(ref currentRotation, ref otherRotation, lerpSpeed, out
var newRotation);
        Entity.Transform.Rotation = newRotation;
    }
}
```

Example Usage

This example demonstrates modifications to the **First Person Shooter** template to integrate smooth camera movement.

1. Detach the camera from the physics entity.
2. Remove the FPS camera script from the camera.
3. Create a new entity as a child of the character body.
4. Add the FPS script to the new entity.
5. Adjust any code that directly references the `CameraComponent` to reflect these changes.

PlayerInput.cs

Change

```
public CameraComponent Camera { get; set; }
```

to

```
public Entity Camera { get; set; }
```

Utils.cs

Change

```
CameraComponent camera
```

to

```
Entity camera,
```

and change

```
camera.Update();
var inverseView = Matrix.Invert(camera.ViewMatrix);
```

to

```
var inverseView = camera.Transform.WorldMatrix;
```

FpsCamera.cs

Remove

```
/// <summary>
/// Gets the camera component used to visualized the scene.
/// </summary>
private Entity Component;
```

and change

```
private void UpdateViewMatrix()
{
    var camera = Component;
    if (camera == null) return;
    var rotation = Quaternion.RotationYawPitchRoll(Yaw, Pitch0);

    Entity.Transform.Rotation = rotation;
}
```

to

```
private void UpdateViewMatrix()
{
    var rotation = Quaternion.RotationYawPitchRoll(Yaw, Pitch, 0);

    Entity.Transform.Rotation = rotation;
}
```

That should be all that is needed to see the smoothing in action as a before and after. You can see the original issue in the Stride GitHub here if you need to find more info on the problem.

Platforms

Stride is cross-platform game engine. This means you can create your game once, then compile and deploy it on all the platforms Stride supports. The engine converts C# and shaders to the different native languages, and abstracts the concepts that differ between platforms, so you don't have to adapt your code for each platform.

Supported platforms

- Windows 7, 8, 10
- Windows Universal Platform (UWP)
- Linux
- Android 2.3 and later
- iOS 8.0 and later

> **TIP**
>
> To check which platform your project uses, add a break point to your code (eg in a script), run the project, and check the Platform.Type variable.

Supported graphics platforms

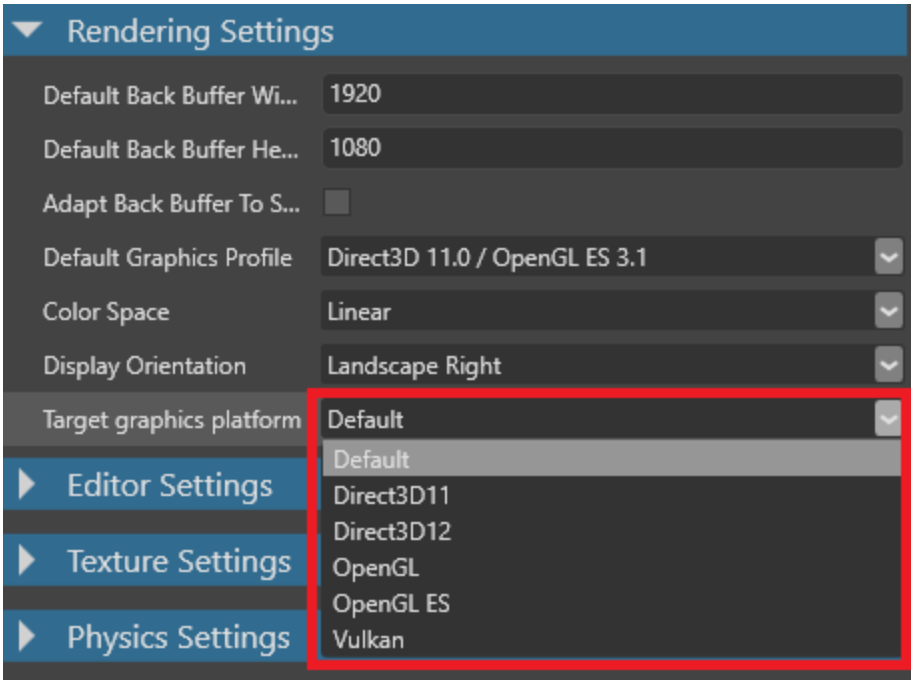
- Direct3D 9 (limited support), 10, 11, 12
- OpenGL 3, 4
- OpenGL ES 2 (limited support), 3
- Vulkan

> **NOTE**
>
> Stride only supports MSAA (multisample anti-aliasing) for Direct3D 11 and later. Depending on your device's OpenGL shader compiler, Stride might not run with OpenGL ES2.

> **WARNING**
>
> Direct3D 9 doesn't support HDR textures. Using HDR textures with DirextX 9 will crash your game.

Set the graphics platform

You set the graphics platform in the **Game settings** asset under **Rendering settings > Target graphics platform**.



For more information, see Set the graphics platform.

Preprocessor variables

Stride defines preprocessor variables if you want to write code that compiles only under a specific platform. For more information, see Preprocessor variables.

In this section

- Linux
- UWP
  - Xbox Live
- iOS
- Add or remove a platform
- Set the graphics platform
- Game settings

Linux

- Setup and requirements
- Create a Linux game

Setup and requirements

To develop for Linux using Stride, you need a Linux PC with a graphics card that supports at least OpenGL 4.2 or Vulkan 1.0. The preferred Linux distribution for Stride is Debian 12 or later, as this was the setup we used to develop the Linux version of Stride.

The instructions below assume you have Debian 12 installed.

You will also need a Windows PC to build your projects for Linux using Game Studio.

Setup

You need the following packages:

- FreeType
- OpenAL
- SDL2
- FreeImage

FreeType

To render fonts, we use the FreeType library. The minimum required version is 2.6 and can be installed via:

Debian / Ubuntu | Fedora | Arch

```
sudo apt install libfreetype6-dev
```

OpenAL

To play sounds and music, we use the OpenAL library. It can be installed via:

Debian / Ubuntu | Fedora | Arch

```
sudo apt install libopenal-dev
```

SDL2

To run games on Linux, we use the SDL2 library which provides the ability to create windows, handle mouse, keyboard and joystick events. The minimum required version is 2.0.4 and can be installed via:

Debian / Ubuntu | Fedora | Arch

```
sudo apt install libsdl2-dev
```

FreeImage

FreeImage is battle-tested library for loading and saving popular image file formats like BMP, PNG, JPEG etc. The minimum required version is 3.18 and can be installed via:

Debian / Ubuntu | Fedora | Arch

```
sudo apt install libfreeimage-dev
```

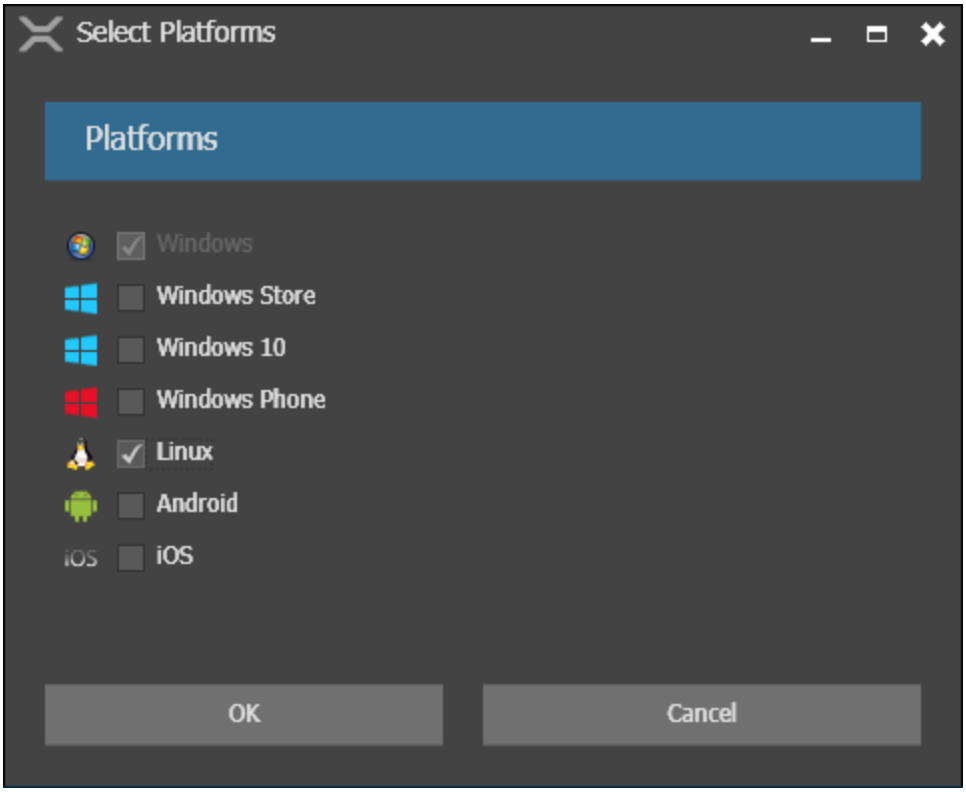
See also

- Create a Linux game

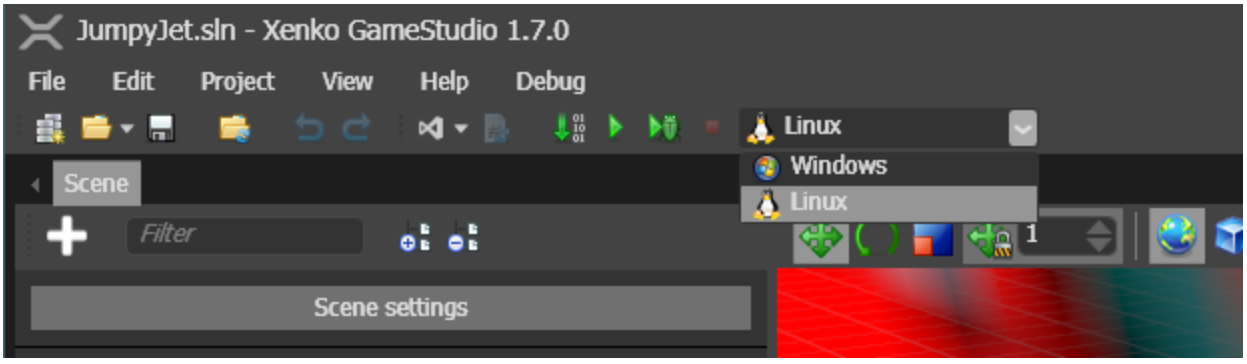
Create a Linux game

> **NOTE**
>
> Before following these instructions, make sure you've followed the instructions in Linux - Setup and requirements.

1. In the Stride launcher, create a new game and select Linux as a target platform.

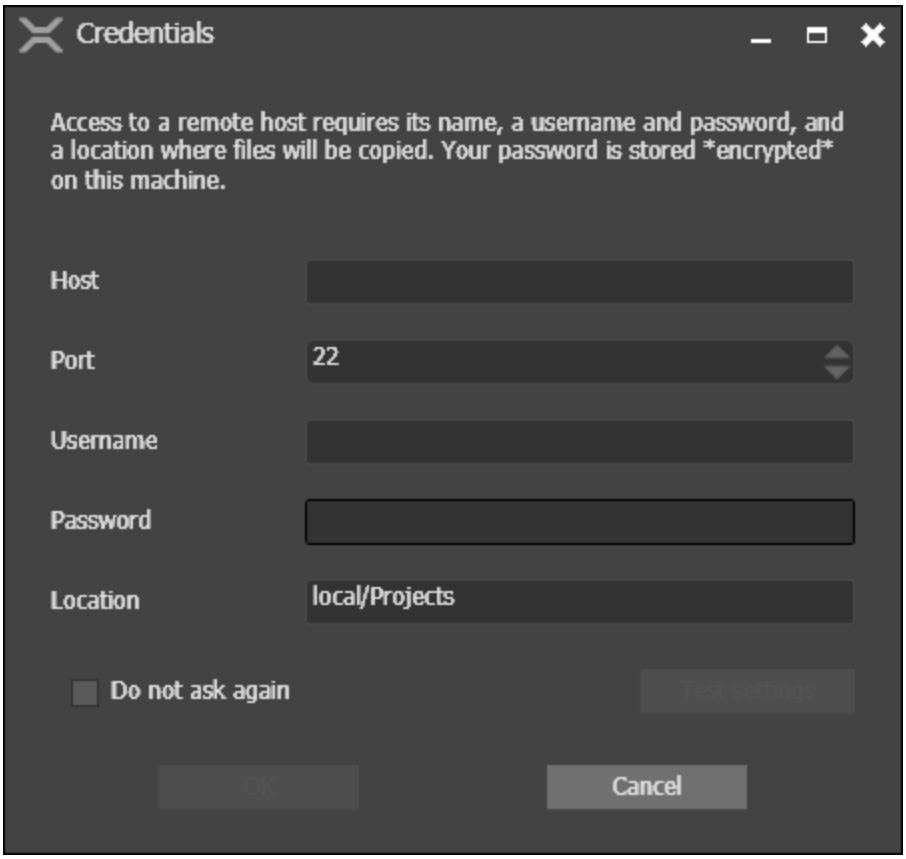


2. In Game Studio, in the platforms menu, select **Linux**.

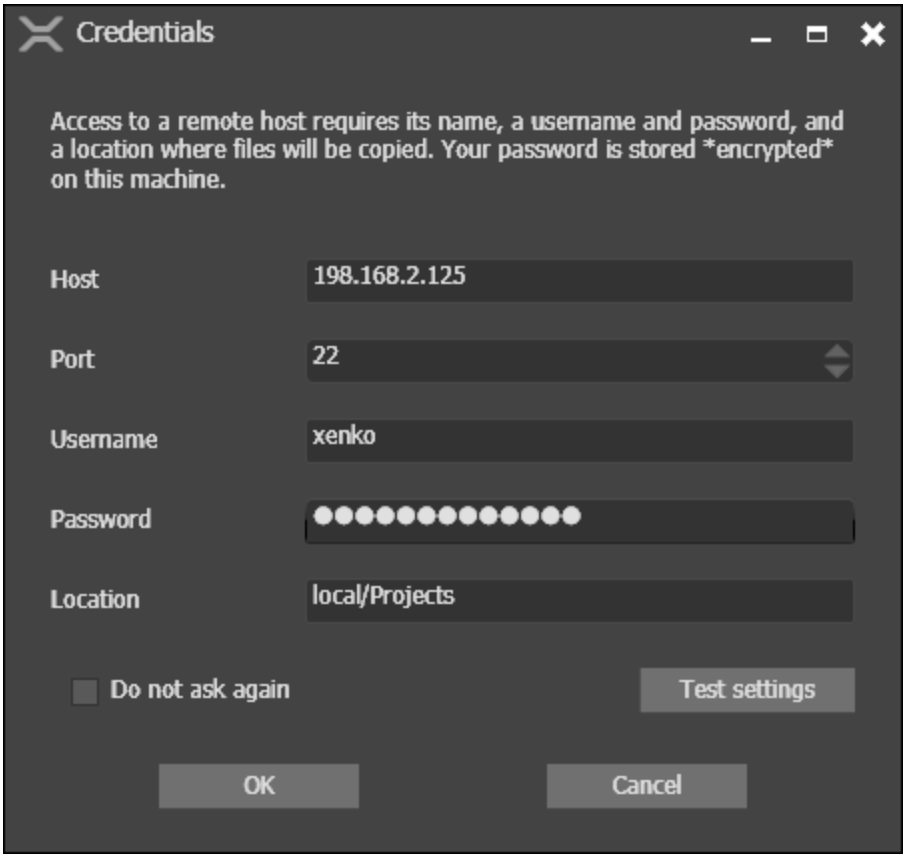


3. Press **F5** to build and run the project.

4. The first time you run the project, enter information about your Linux host:

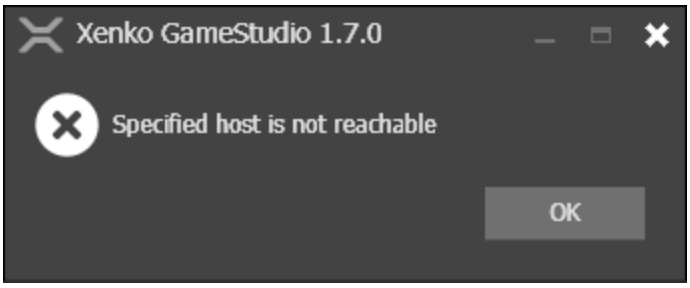


Enter your information as below:

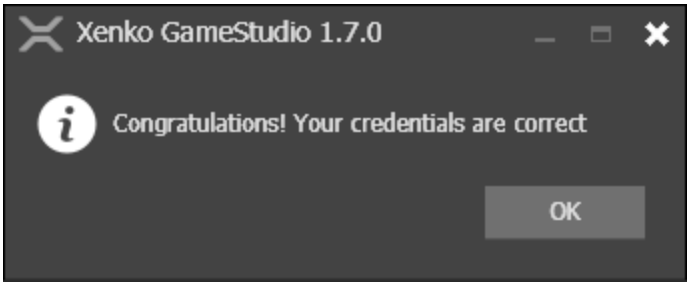


5. Click **Test settings** to test the credentials.

   If you made an error, Game Studio displays:



If the credentials are correct, Game Studio displays:



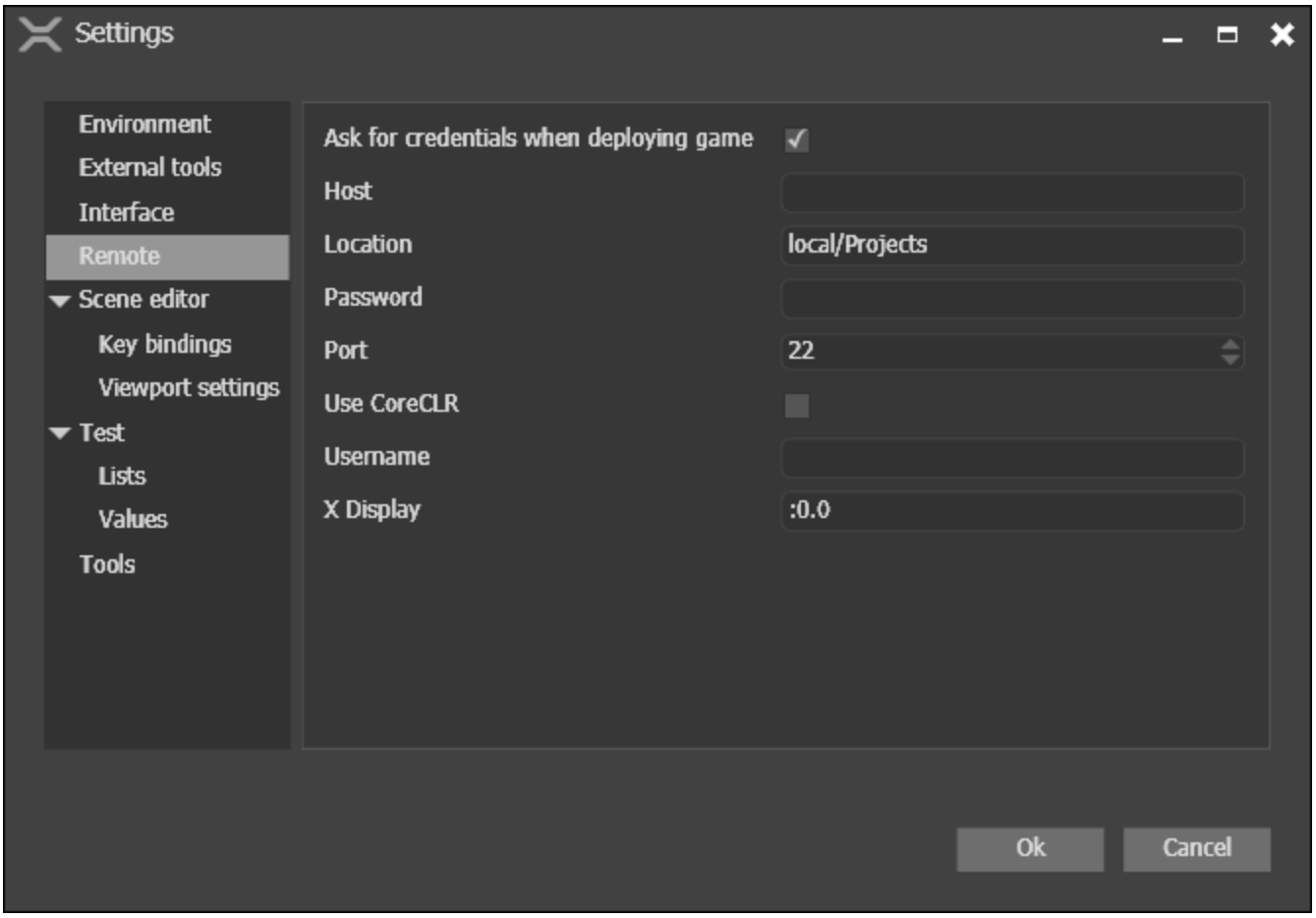
Click the **OK** button to continue.

Game Studio copies the files over your Linux host in a subdirectory of the location you have provided. The name of the subdirectory is the name of your game.

If something goes wrong, check the **Output** pane for details.

Settings

Your credentials are saved in the **Settings** dialog:



The password is encrypted using the Micrsoft *System.Security.Cryptograph.ProtectedData.Protect* method for the current user, and saved in Base64, displayed in the Settings. You can't change the password from the Settings dialog.

There are two additional settings that control the execution of a game:

- Use CoreCLR: forces execution using .NET Core
- X Display: forces execution on a specific X display of your Linux host

Compile outside Game Studio

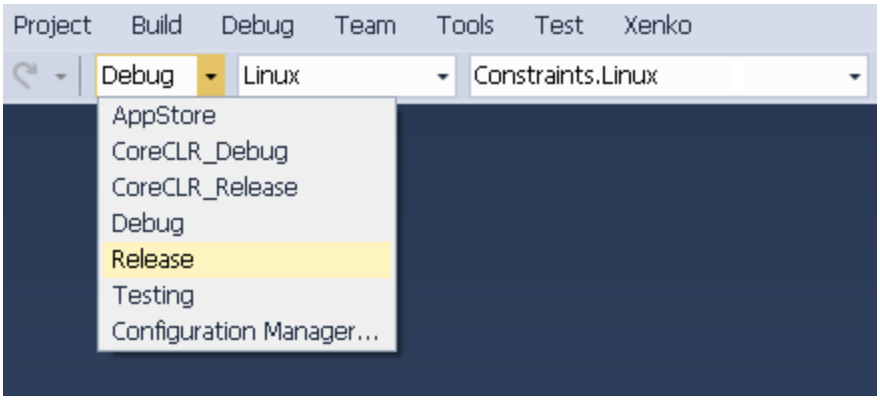
Like any Stride project, you can also compile the project directly from Visual Studio or from the command line. In both cases, you need to select a valid configuration:

- Debug
- Release
- CoreCLR_Debug
- CoreCLR_Release

Debug and Release target Mono. The others target .NET Core.

Visual Studio

Once your project is loaded in Visual Studio, select the Linux project. In the **Solution Configurations** drop-down menu, you select a valid Linux configuration:



MSBuild

To compile for Linux, from a command line, use:

```
msbuild /p:Platform=Linux /p:Configuration=CONFIG YourGame.sln
```

Where **CONFIG** is a valid Linux configuration.

Limitations

- No debugging facility yet
- Switching the rendering graphics platform might cause the game to hang on startup. As a workaround, on the Linux host, in the directory where the game is deployed, delete the following directories:
  - `cache`
  - `local`
  - `roaming`

See also

- Linux — Setup and requirements

UWP

Windows 10 introduces the Universal Windows Platform (UWP), which provides a common app platform available on every device that runs Windows 10. For more information about UWP, see Intro to the Universal Windows Platform on the MSDN documentation.

In this section

- Xbox Live

Xbox Live

This page explains how to configure your project to work with Xbox Live.

1. Before you start

1. Make sure your project uses UWP as a platform. To do this, you can either:

   - create a project and select **UWP** as a platform, or
   - add **UWP** as a platform to an existing project

   > **TIP**
   > For this tutorial, you might find it useful to create a new project to test the process, then apply the knowledge to your existing projects.

2. Make sure you can run the project from UWP. To do this, in Visual Studio, select the platform you want (UWP-64, UWP-32, or UWP-ARM) from the **Solution Platform** drop-down list, and run the project.

3. Download the Xbox Live SDK.

   To write this page, we used XboxLiveSDK-1612-20170114-002. The sample is loosely based on the Achievements sample in the Xbox Live SDK.

4. Change your Xbox Live environment. In the **SDK** folder, under **Tools**, run:

   ```
   SwitchSandbox.cmd XDKS.1
   ```

   XDKS.1 is the sandbox used for the Microsoft samples.

   > **WARNING**
   > This blocks regular Xbox accounts and only permits developer accounts. To switch back, run:

   ```
   SwitchSandbox.cmd RETAIL
   ```

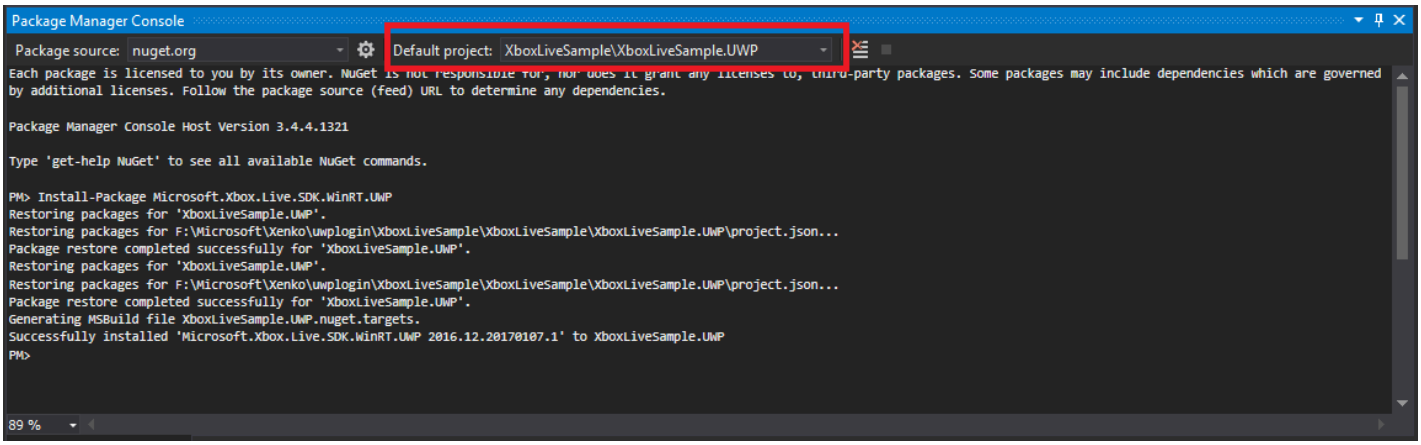
5. Make sure you can run the Achievements sample with your developer account.

2. Add the Xbox Live SDK to your solution

1. In Visual Studio, open your game solution.

2. Open the Package Manager Console (**Tools > NuGet Package Manager > Package Manager Console**).

3. In the **Default project** field, select your UWP project (eg *MyGame.UWP*).

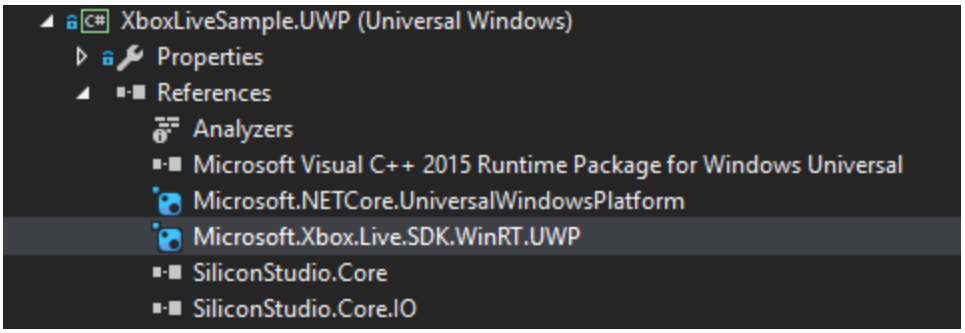


4. In the console, type:

```
PM > Install-Package Microsoft.Xbox.Live.SDK.WinRT.UWP
```

   Visual Studio adds the NuGet package to your project.

5. Make sure the package appears in the **References** list.



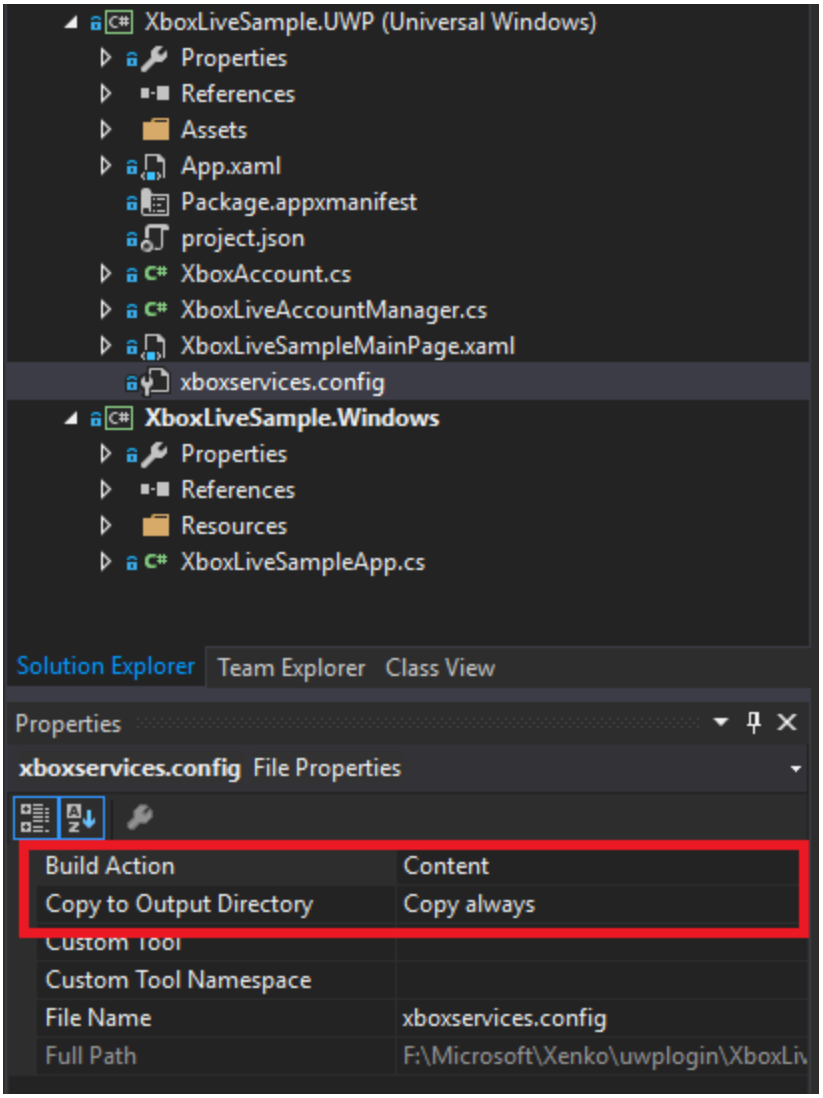
3. Configure the UWP project

1. Delete *MyGame.UWP.TemporaryKey.pfx*.

2. Add *xboxservices.config* to your project.

   You can get this file from any Xbox Live SDK sample (eg the Achievements sample) for test purposes. When you want to publish the game, change the **TitleId** and **Service config Id** with the ones provided for your project. For details, see the Xbox Live documentation.

3. In the *xboxservices.config* properties, under **Build Action**, select **Content**, and under **Copy to Output Directory**, select **Always**.



4. Edit *Package.appxmanifest* with details relevant to your project.

   Again, you can use the manifest file from any Xbox Live SDK sample (eg the Achievements sample) for test purposes. If you associate your game with a store app, use the generated manifest file. For details, see the Xbox Live documentation.

5. Make sure the capability *InternetClientServer* is enabled.

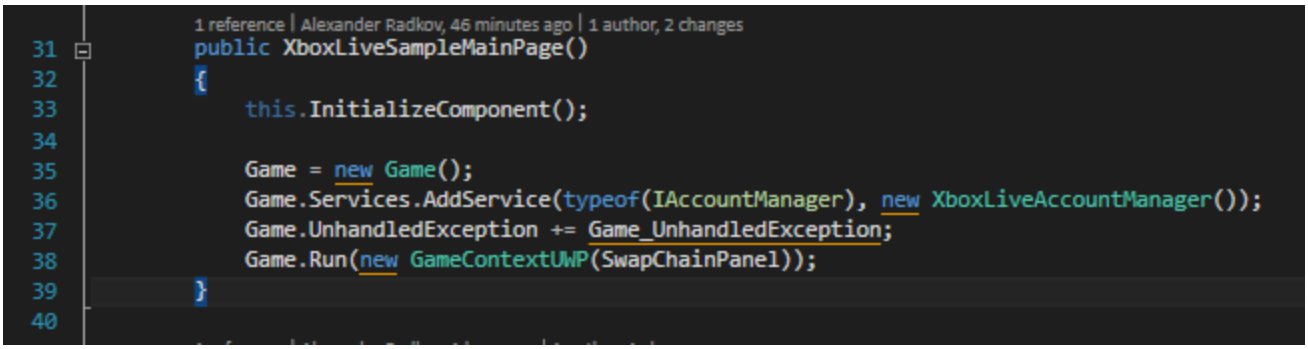
4. Add user account scripts to your game

You need to enable Xbox Live capability in your game project without exposing the Xbox Live SDK. As *MyGame.UWP* already references *MyGame.Game*, we can't reference it. Instead, we can create an interface and implement it from the UWP project side.

> **NOTE**
>
> There are several ways to do this. This page explains one method.

1. Add two interfaces to your game, `IAccountManager` and `IConnectedAccount`.
2. On your UWP project (eg *MyGame.UWP*), implement the interfaces `public class XboxAccount : IConnectedAccount` and `public class XboxLiveAccountManager : IAccountManager`.
3. Add the account factory to your game so you can access it later from a game script. In the `MyGameMainPage.xaml.cs`, add the following line:

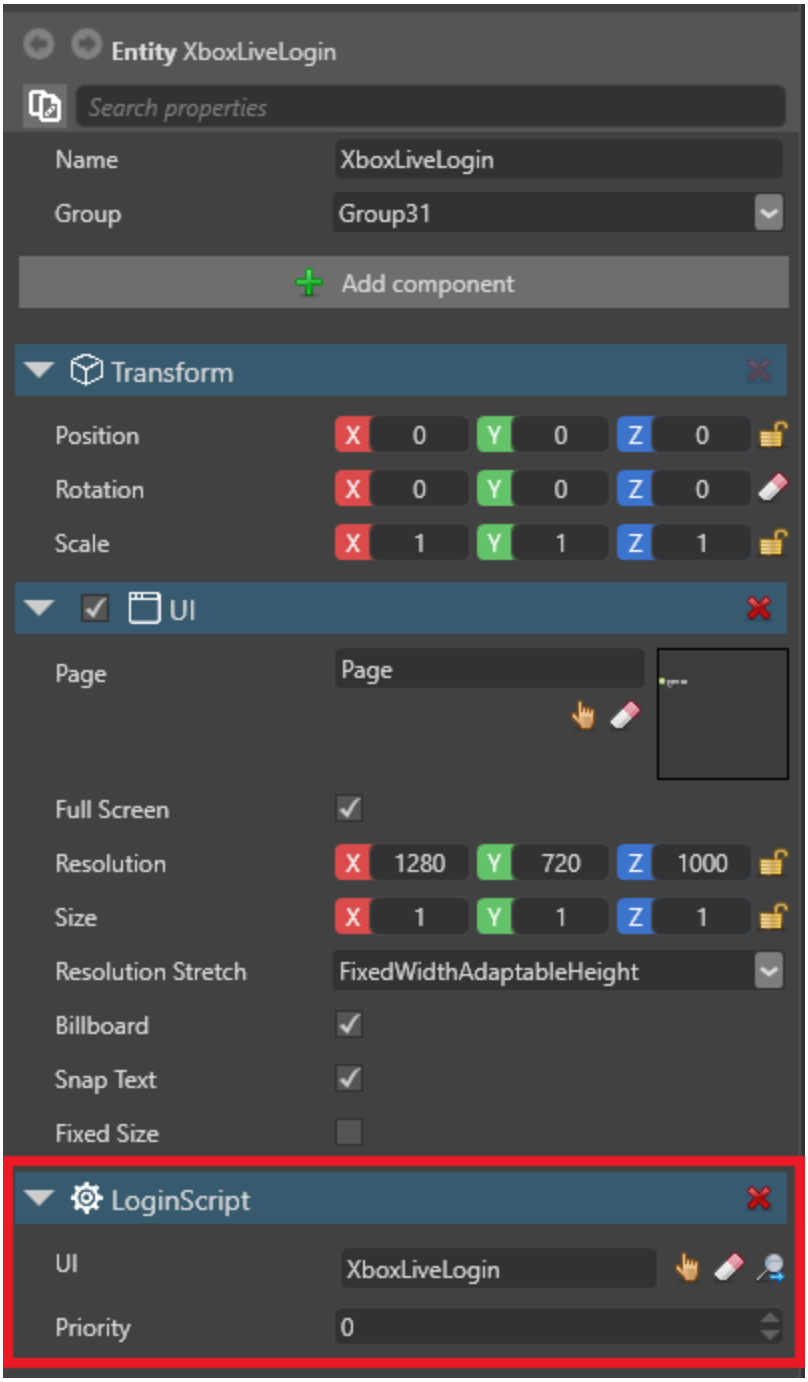
```
Game.Services.AddService(typeof(IAccountManager), new XboxLiveAccountManager());
```



The final script should look like this at minimum:

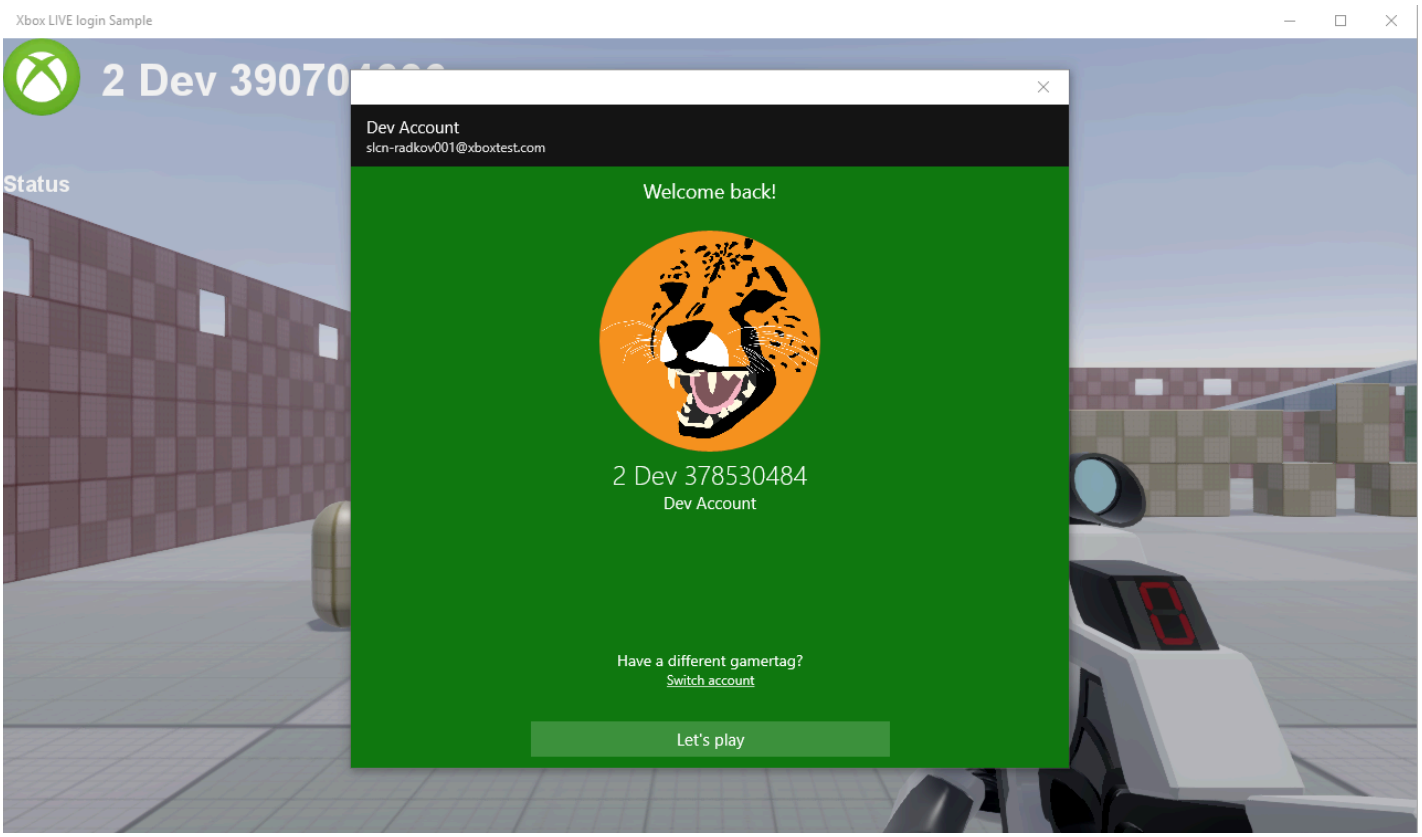
```
public class LoginScript : AsyncScript
{
    private IConnectedAccount account;

    public override async Task Execute()
    {
        var accountMgr = Services.GetServiceAs<IAccountManager>();
        account = accountMgr?.CreateConnectedAccount();
                if (account == null)
                        return;

        var result = account.LoginAsync();

            // TODO Add your code here!
    }
}
```

Now you can expose the `xbox_live_user` functionality and other classes in your game.



Sample project

- Download a sample project with Xbox Live login functionality



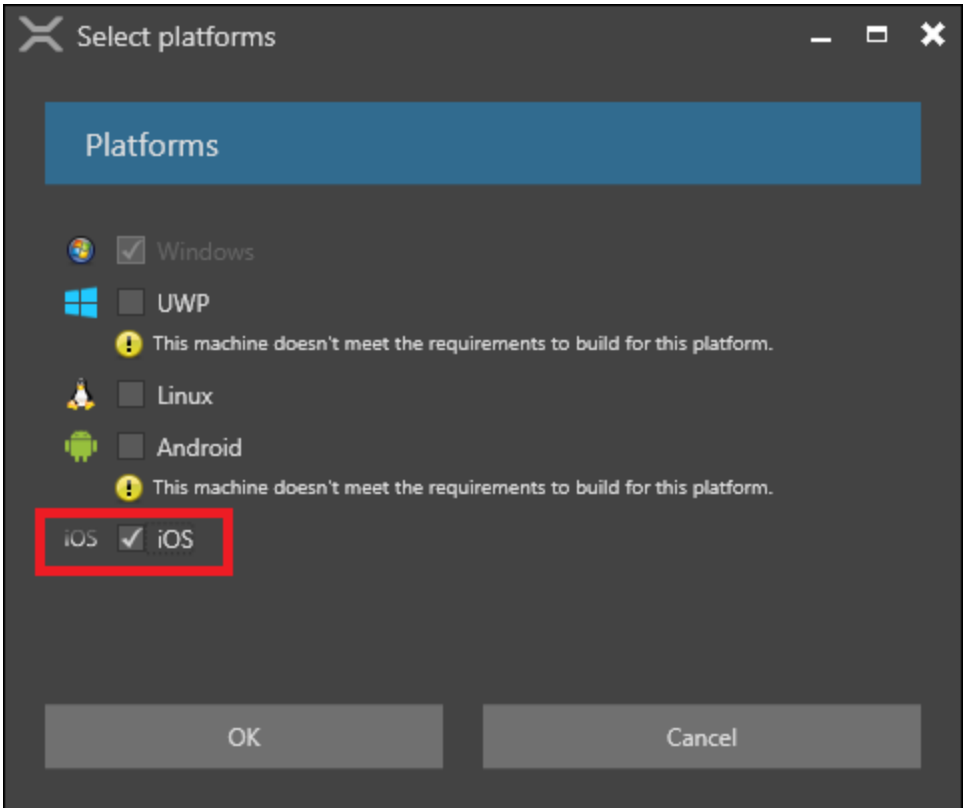
See also

- Platforms

iOS

To deploy your game on iOS devices, you need to connect the device to a Mac with Xamarin.

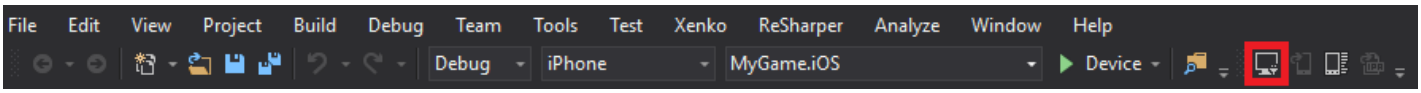
1. Make sure Xamarin is installed on the PC and the Mac. For instructions about how to install and set up Xamarin, see the Xamarin documentation:
   - Installing Xamarin in Visual Studio on Windows
   - Connecting to Mac
2. Make sure your iOS device is provisioned. For instructions, see Device provisioning in the Xamarin documentation.
3. Make sure the iOS platform is added to your Stride project. To do this, in Game Studio, right-click the solution, select **Update package > Update Platforms**, and make sure **iOS** is selected.



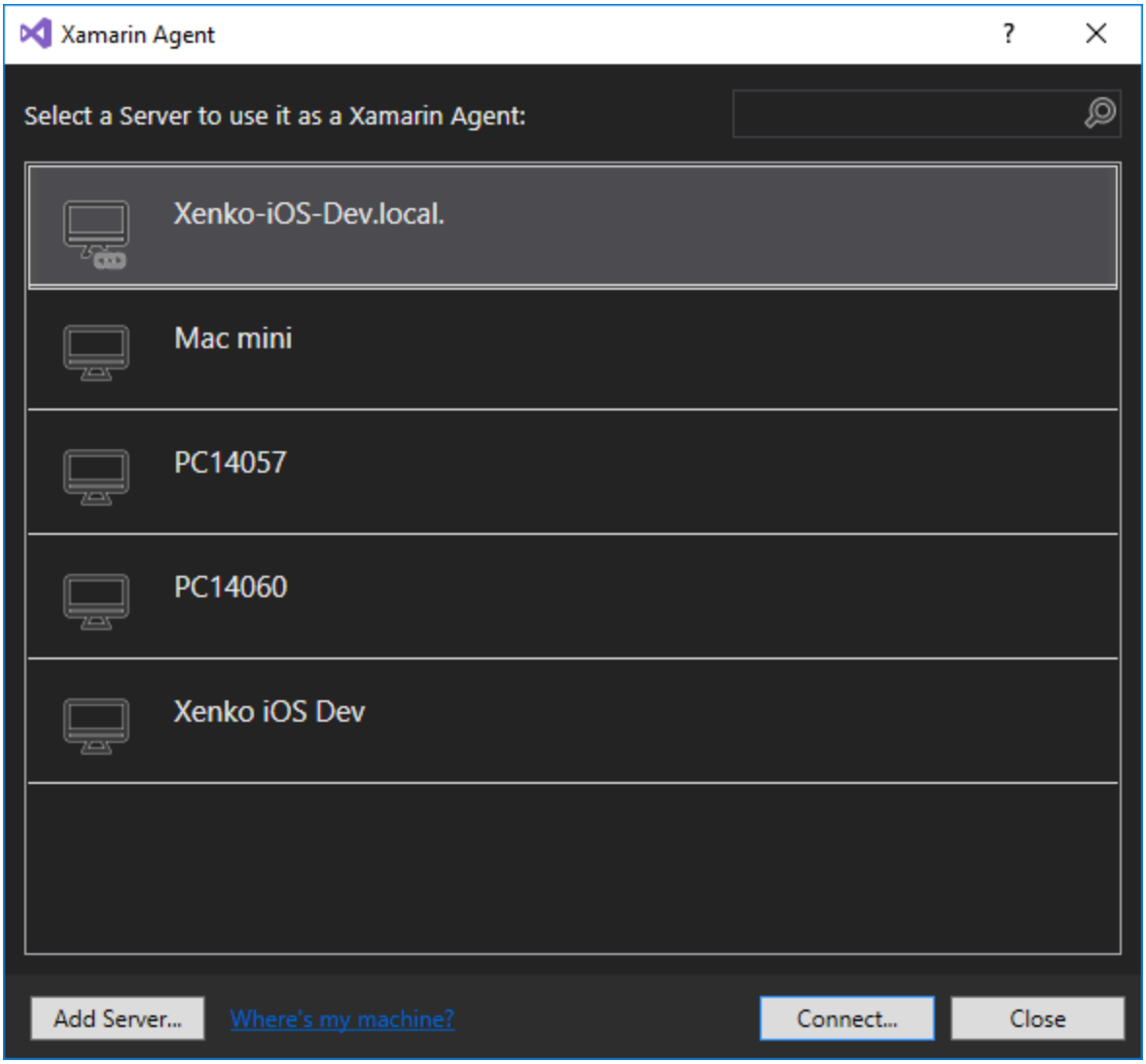
   For more information about adding platforms in Game Studio, see Add or remove a platform.
4. Open your solution in Visual Studio.

> **TIP**
>
> To open your project in Visual Studio from Game Studio, in the Game Studio toolbar, click (**Open in IDE**).

5. In the Visual Studio toolbar, click .

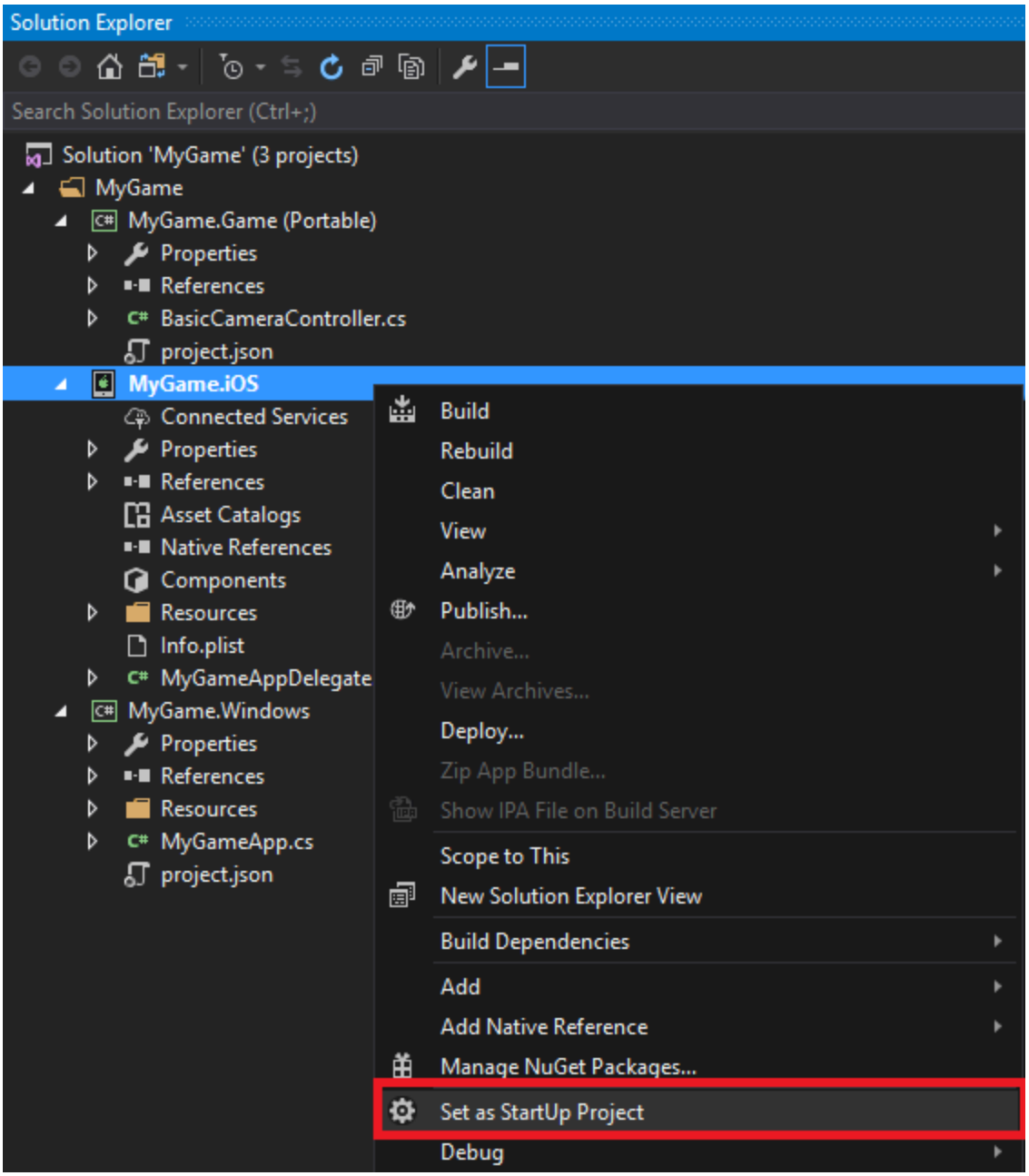


Xamarin Agent opens.

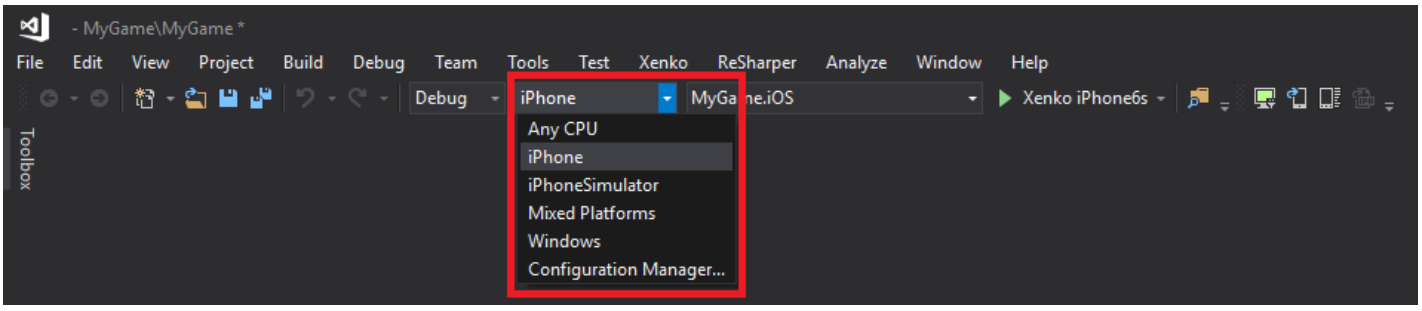


6. Connect to the Mac via Xamarin. For instructions, see Introduction to Xamarin iOS for Visual Studio in the Xamarin documentation.

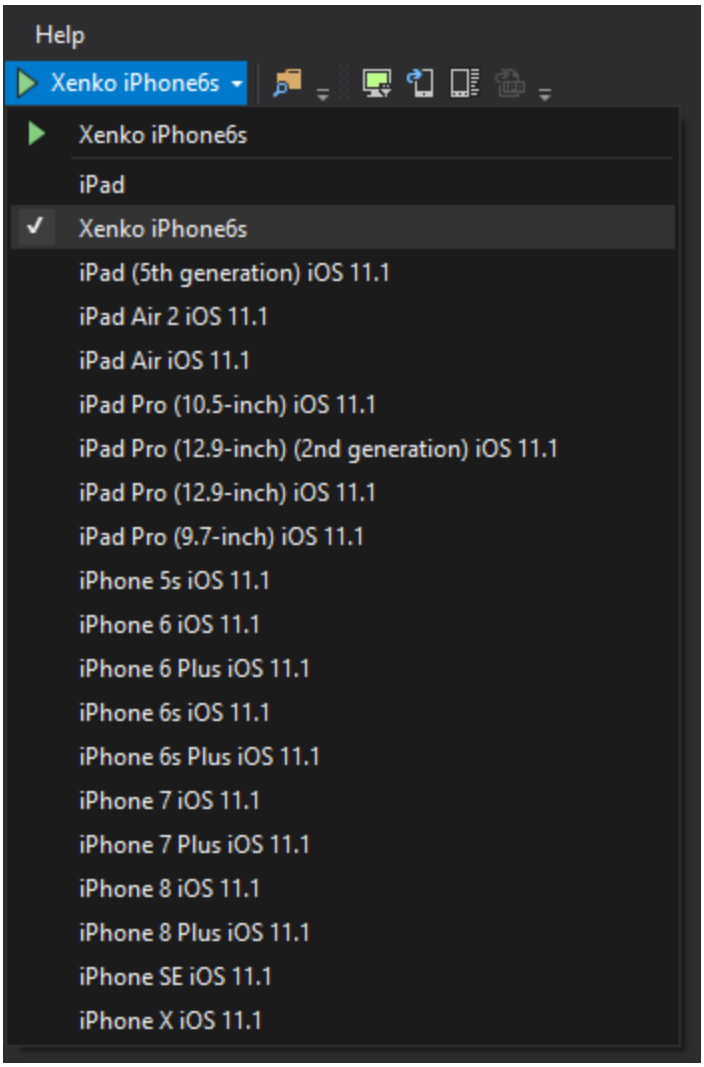
7. In the **Solution Explorer**, right-click the project and select **Set as StartUp Project**.



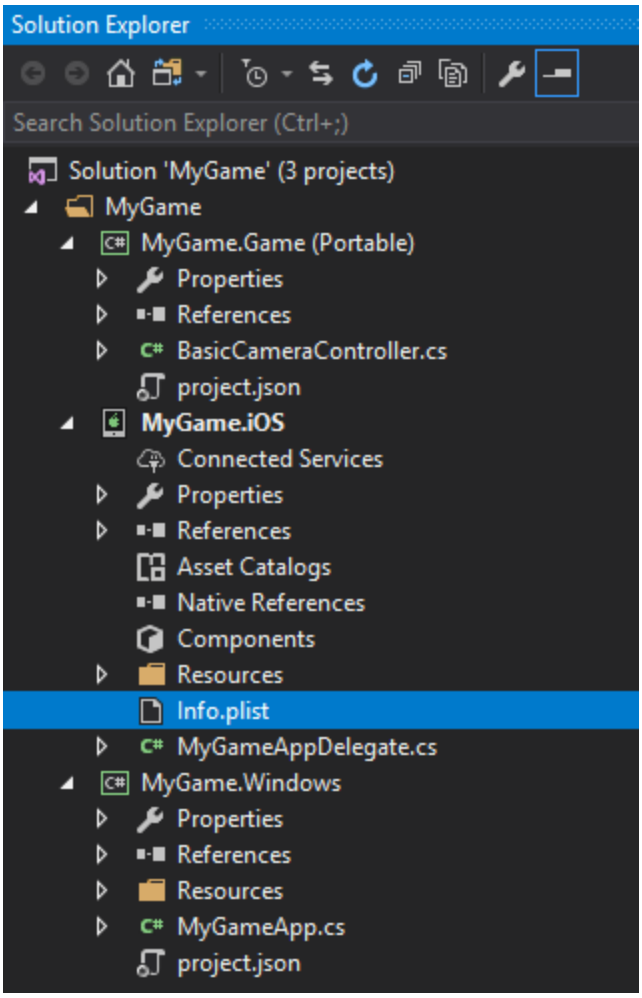
8. In the **Solution Platforms** menu, select **iPhone** to build on physical iOS devices (including iPad), or **iPhoneSimulator** to build for the simulator. The simulator emulates iOS devices on your machine, but has some drawbacks (see below).



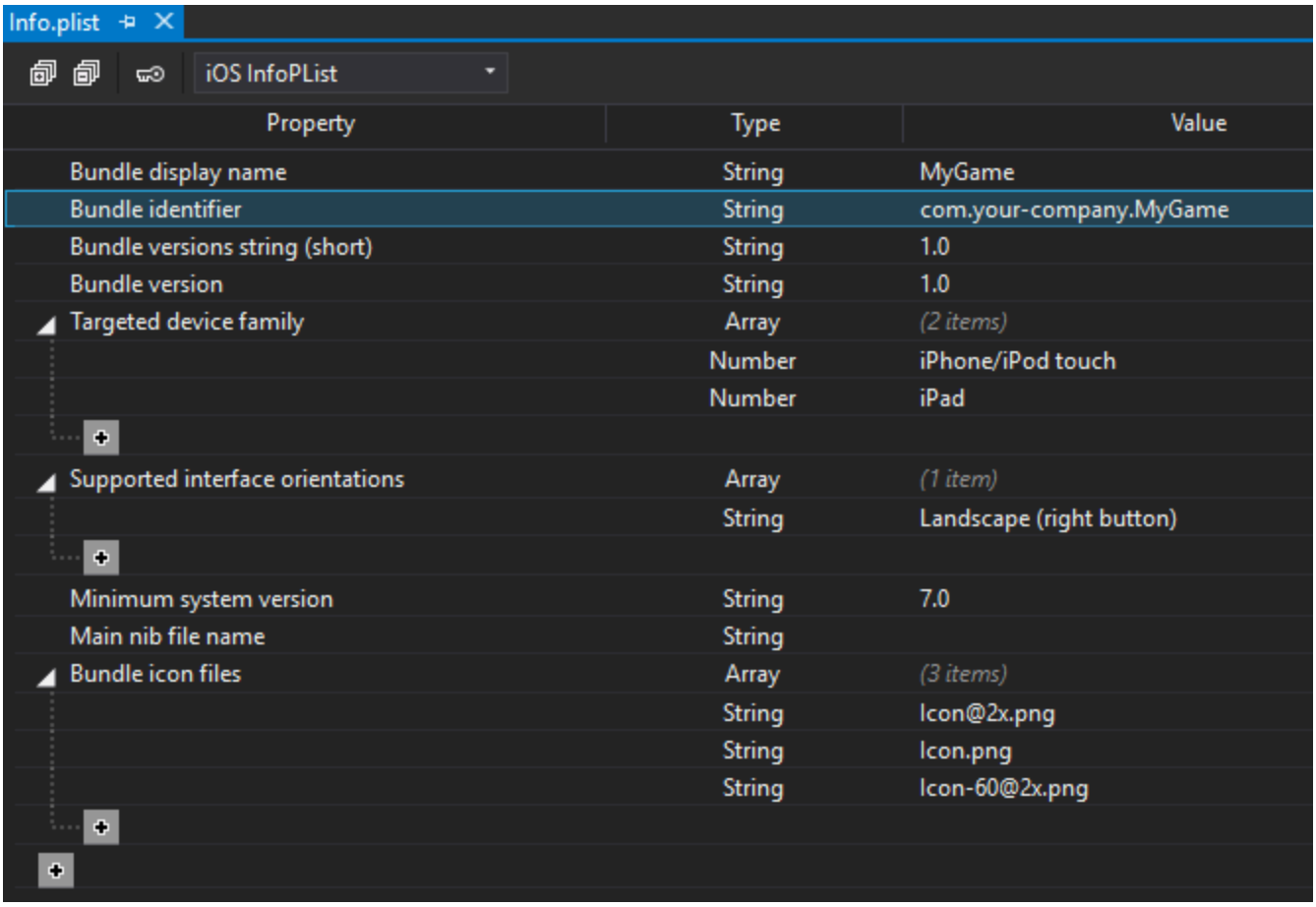
9. In the Visual Studio toolbar, select the iOs device you want to build for.



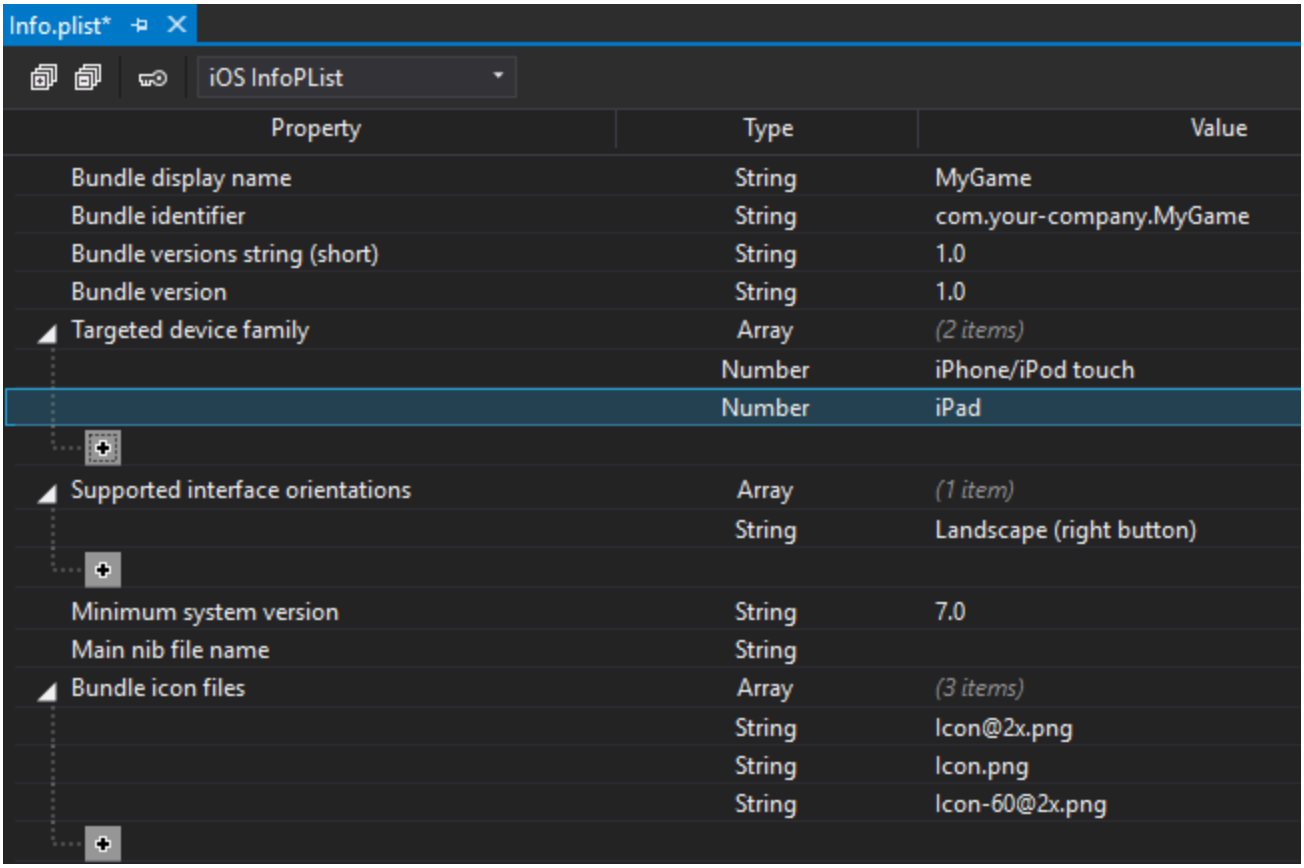
10. From the **Solution Explorer**, open `info.plist`.



11. If you want to create a release build, set the **bundle identifier**. This is a unique ID for your application.



12. If you want to deploy on iPad, under **Targeted device family**, click .

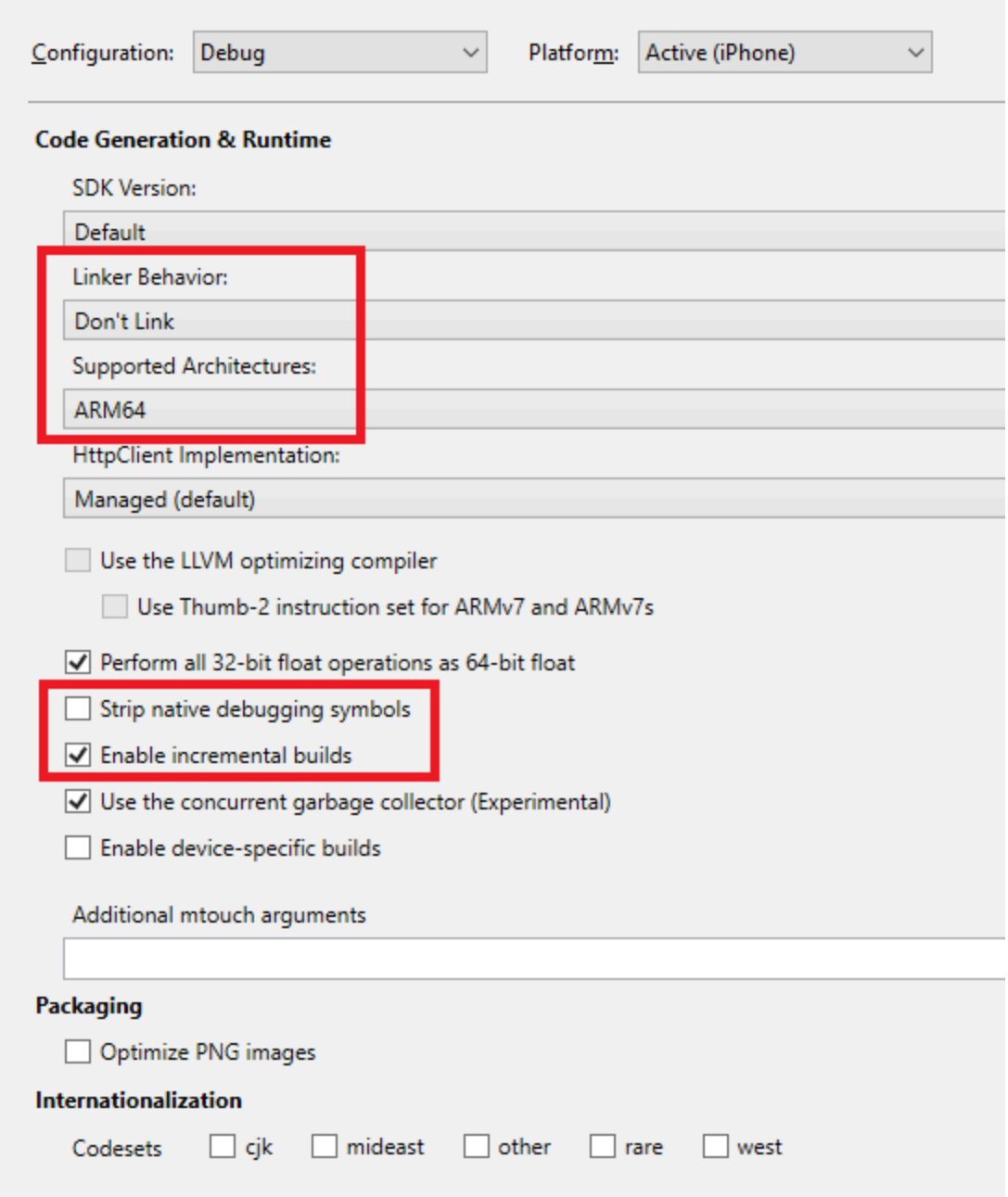


Speed up builds on iOS devices

It takes a long time to build on iOS devices. This is because:

- the Mac needs to build code ahead of time (AOT) for the different devices
- the Apple sandbox system doesn't let you update packages incrementally, so the Mac needs to completely redeploy the application on the device for every change

To compile code more quickly, in the Solution Explorer, right-click the iOS project and select **Properties**.



- Under **Linker Behavior**, select **Don't link**.
- Under **Supported Architectures**, select only the architecture of the debug device.
- Disable **Strip native debugging symbols**.
- Enable **incremental builds** (only code that changes from one execution to another is AOT)

For more information, see iOS Build Mechanics in the Xamarin documentation. For information about profiling, see Using instruments to detect native leaks using markheap.

To make redeploying each time faster, make your debug packages as small as possible.

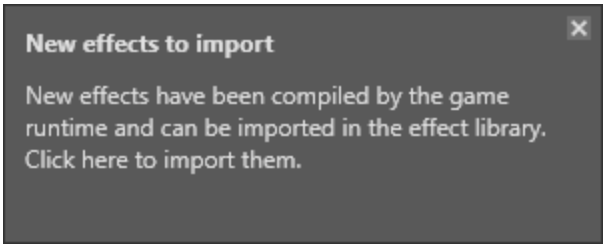
- In Game Studio, reduce the **Size** of the textures in your project.
- Remove unused assets.
- Test your scenes one by one rather than loading them simultaneously.
- Debug your application on the **iPhone simulator** instead of a real device. However, execution is slow on the simulator and it produces some rendering artifacts, so we don't recommend using it to debug real-time graphics.

Compile shaders on iOS

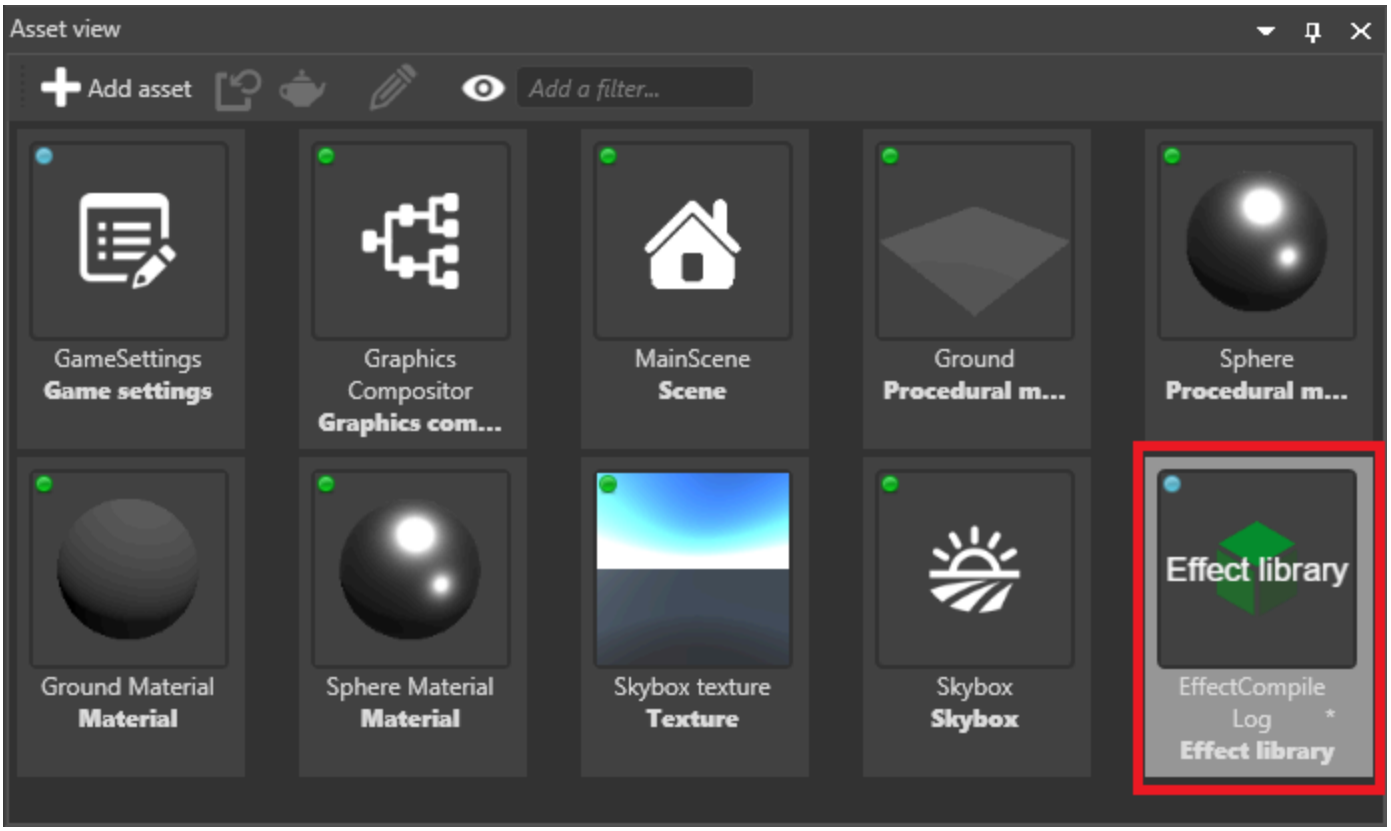
As converting Stride shaders to OpenGL shaders on iPhone devices is slow, we recommend you convert them remotely (ie in Game Studio).

Our recommended workflow is:

1. Execute the app on Windows. This creates the shader permutations.



2. Import the new shaders in Game Studio. This generates an effect log.



3. Save and run the game on iOS.

Ideally, this creates all the shader permutations remotely, so you don't need to convert them on the device. However, new permutations might still occur due to differences such as supported screen resolutions. For more information, including information about how to compile shaders remotely on iOS, see Compile shaders.

See also

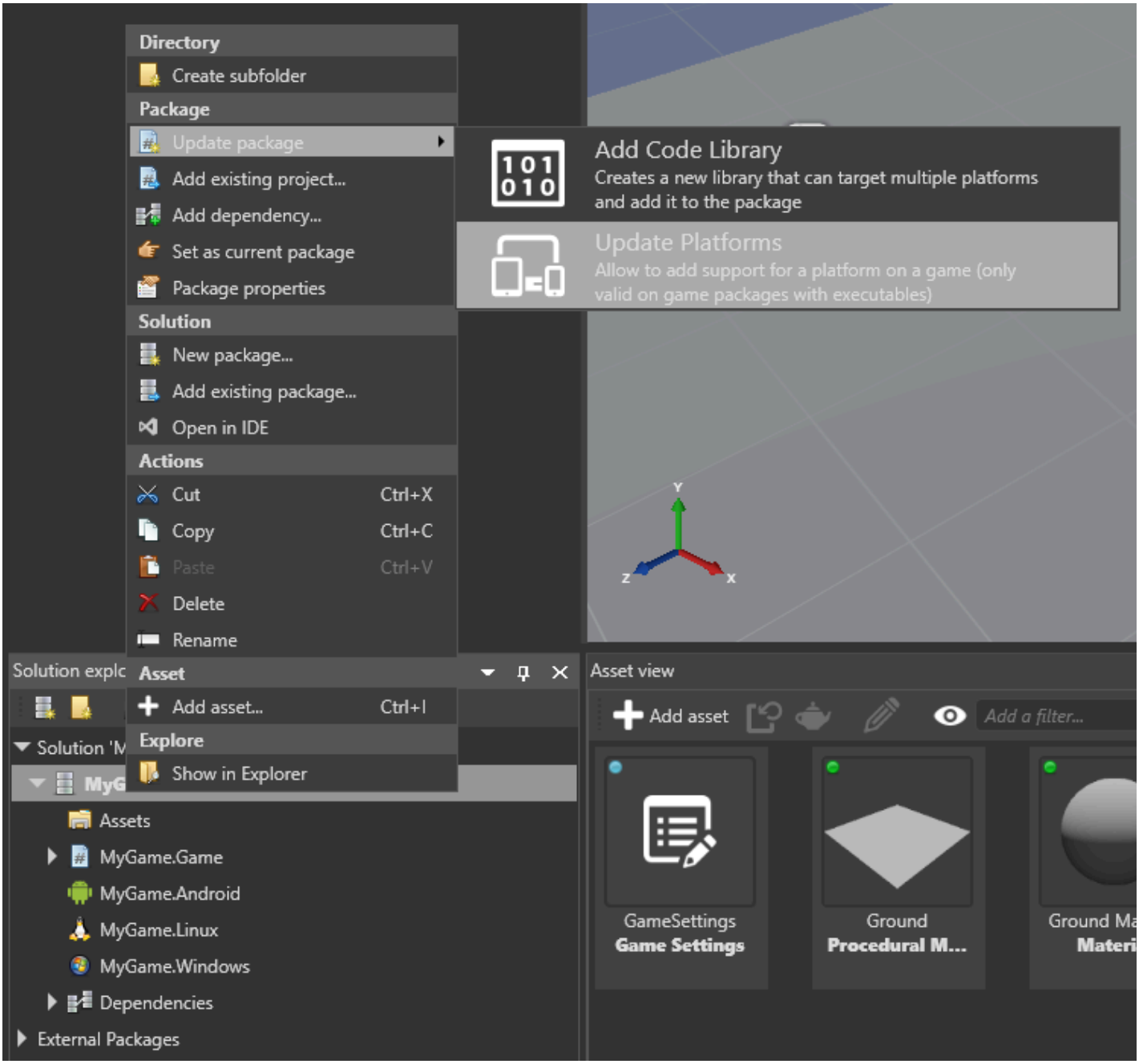
- iOs in the Xamarin documentation
- Compile shaders

Add or remove a platform

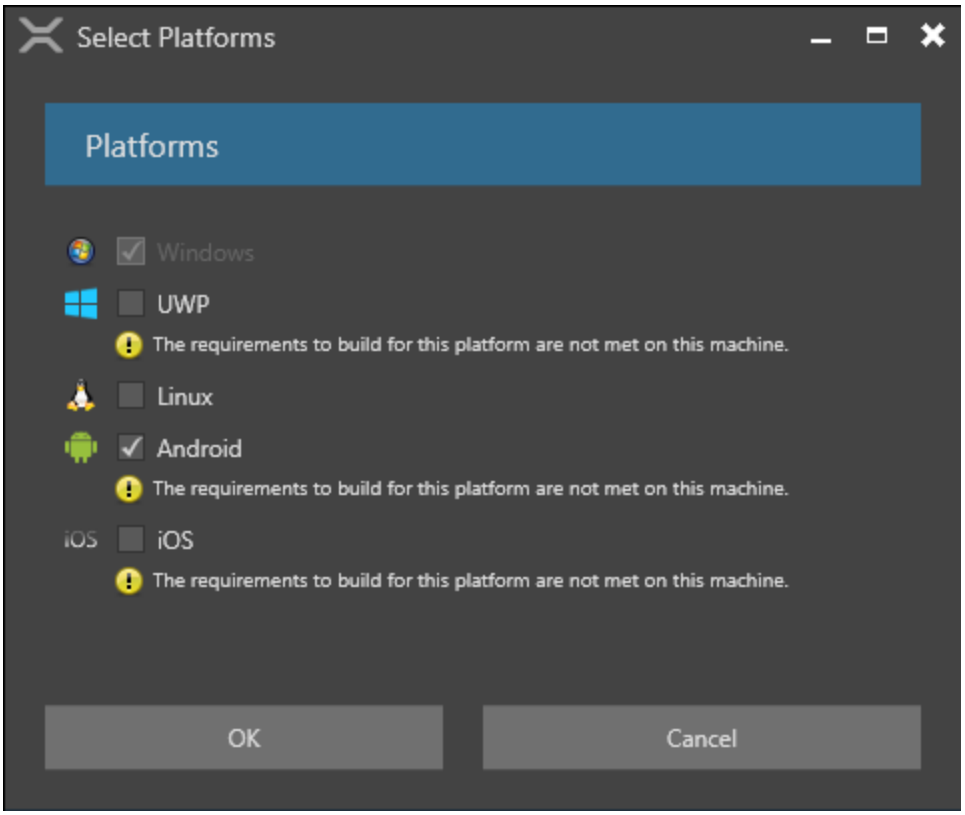
Beginner

You can add and remove platforms to and from projects.

1. In the **Solution Explorer** (default bottom left), right-click the project and select **Update package** > **Update platforms**.

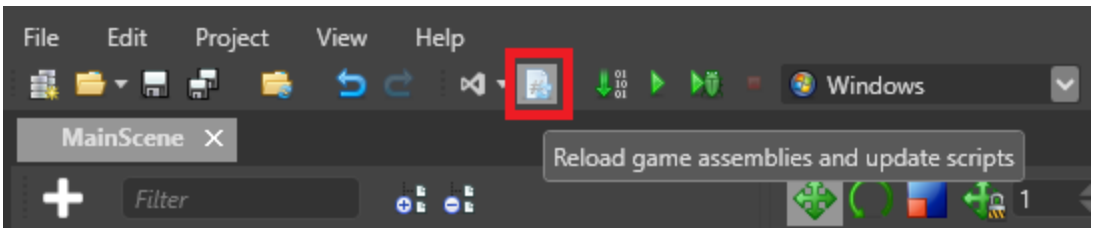


The **Select Platforms** dialog opens.



2. Select the platforms you want to support and click **OK**.

3. Reload the assemblies by clicking the **Reload game assembles** button in the toolbar.



The supported platforms are updated. To refresh the platforms list in the toolbar, restart Game Studio.

See also

- Platforms

Set the graphics platform

Beginner

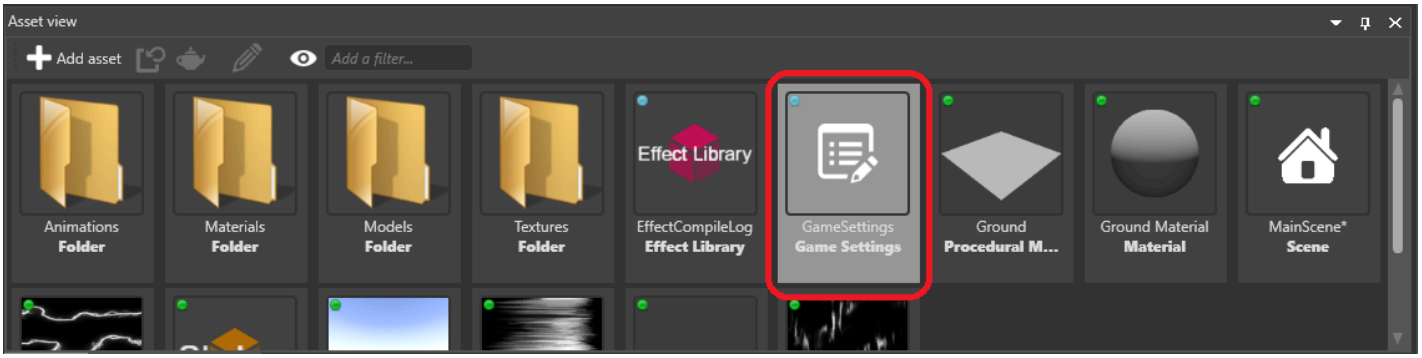
The **graphics platform** controls the graphics hardware in the device you run your project on. Different devices support different graphics platforms; for example, iOS supports the OpenGL ES graphics platform. You can select which graphics platform your game uses, and add overrides for different platforms (eg Windows, Android, etc).

> **WARNING**
>
> Moving from Direct3D to an earlier Direct3D version can create problems. For example, if your game contains HDR textures, it will crash, as Direct3D 9 doesn't support them.

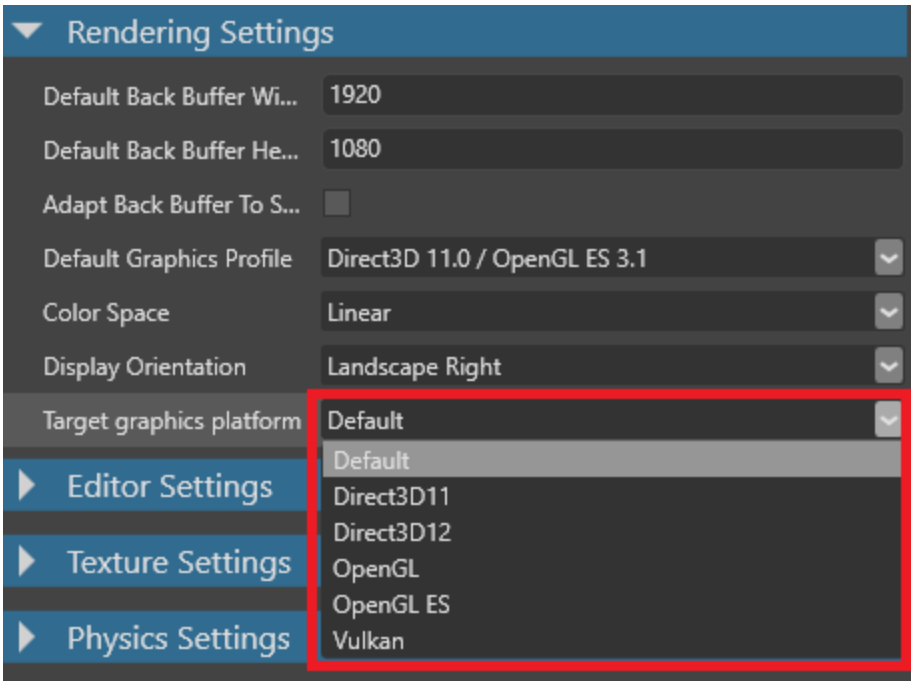
You set the graphics platform in the game settings asset.

> **NOTE**
>
> Make sure you have the latest drivers for the graphics platforms you want to use.

1. In the **Asset View**, select the **Game Settings** asset.



2. In the Property Grid, under **Rendering Settings > Target graphics platform**, select the graphics platform you want to use.



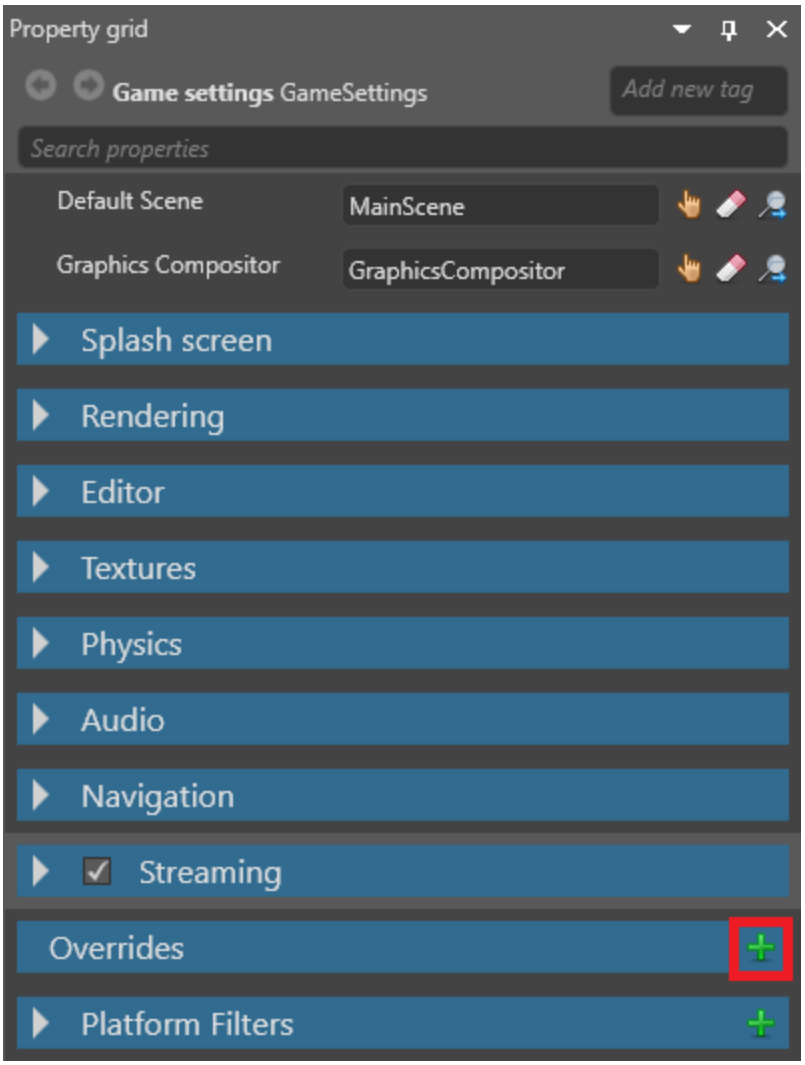
If you select **Default**, Stride uses the graphics platform appropriate for your platform (eg Windows, Android) when you build.

Platform	**Default graphics platform**
Windows, UWP	Direct3D11
Linux, Mac OS	OpenGL
Other	OpenGL ES

Override the graphics platform

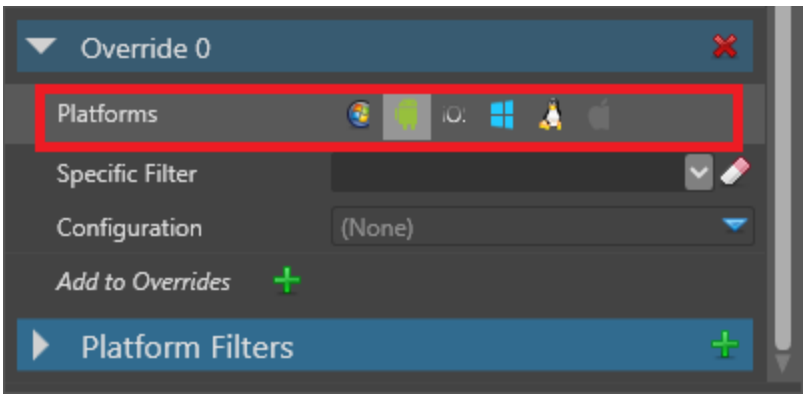
You can override the graphics platform Stride uses for specific platforms. For example, you can have Linux use Vulkan while other platforms use the default.

1. With the **GameSettings** asset selected, in the Property Grid, under **Overrides**, click ➕ (**Add**).

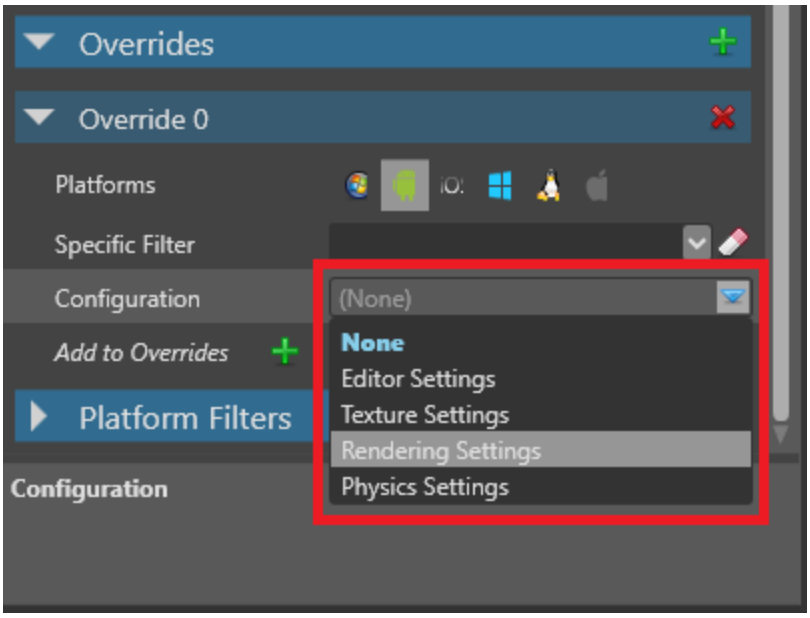


Game Studio adds an override.

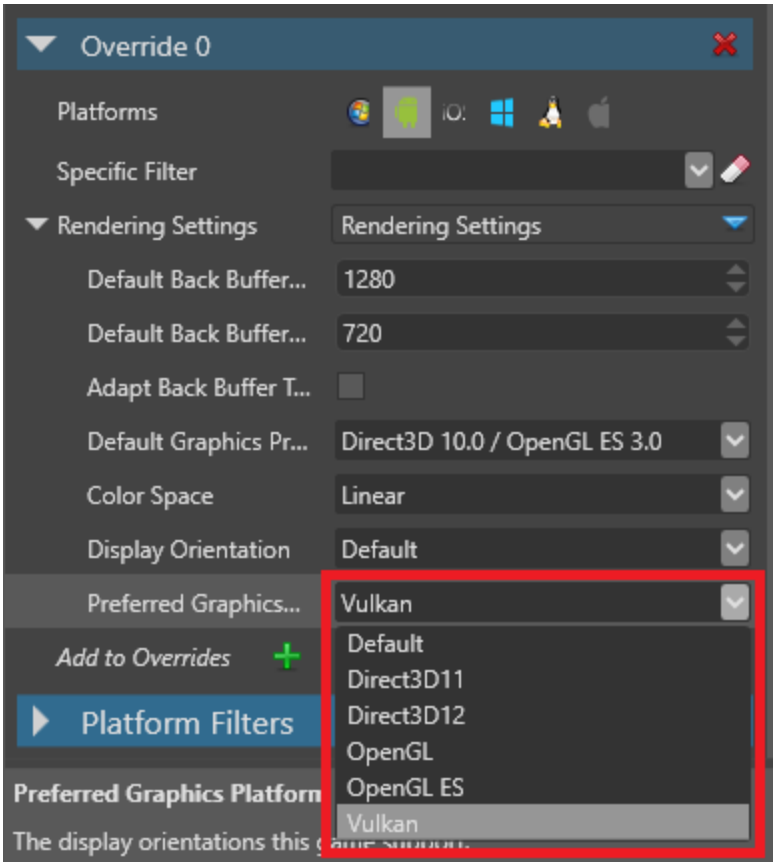
2. In the new override, next to **Platforms**, select the platforms you want this override to apply to.



3. In the **Configuration** drop-down menu, select **Rendering Settings**.



4. Under **Rendering Settings**, in the **Preferred Graphics Platform** drop-down menu, select the graphics platform you want to use.



Stride overrides the graphics platform for the platforms you selected.

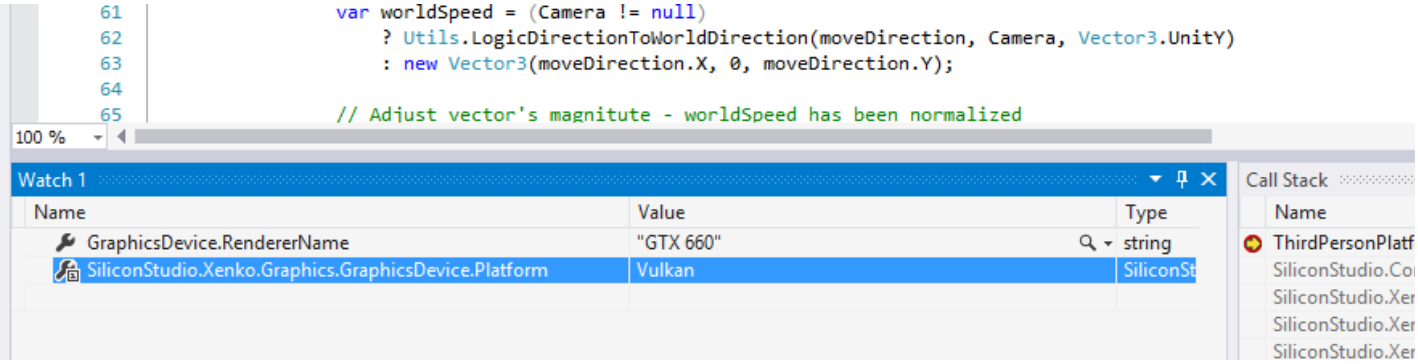
Check which graphics platform your project uses

1. Add a break point to your game code (eg in a script).

2. Run the project.

3. Check the value of the GraphicsDevice.Platform variable.

   For example, this project is using Vulkan:



See also

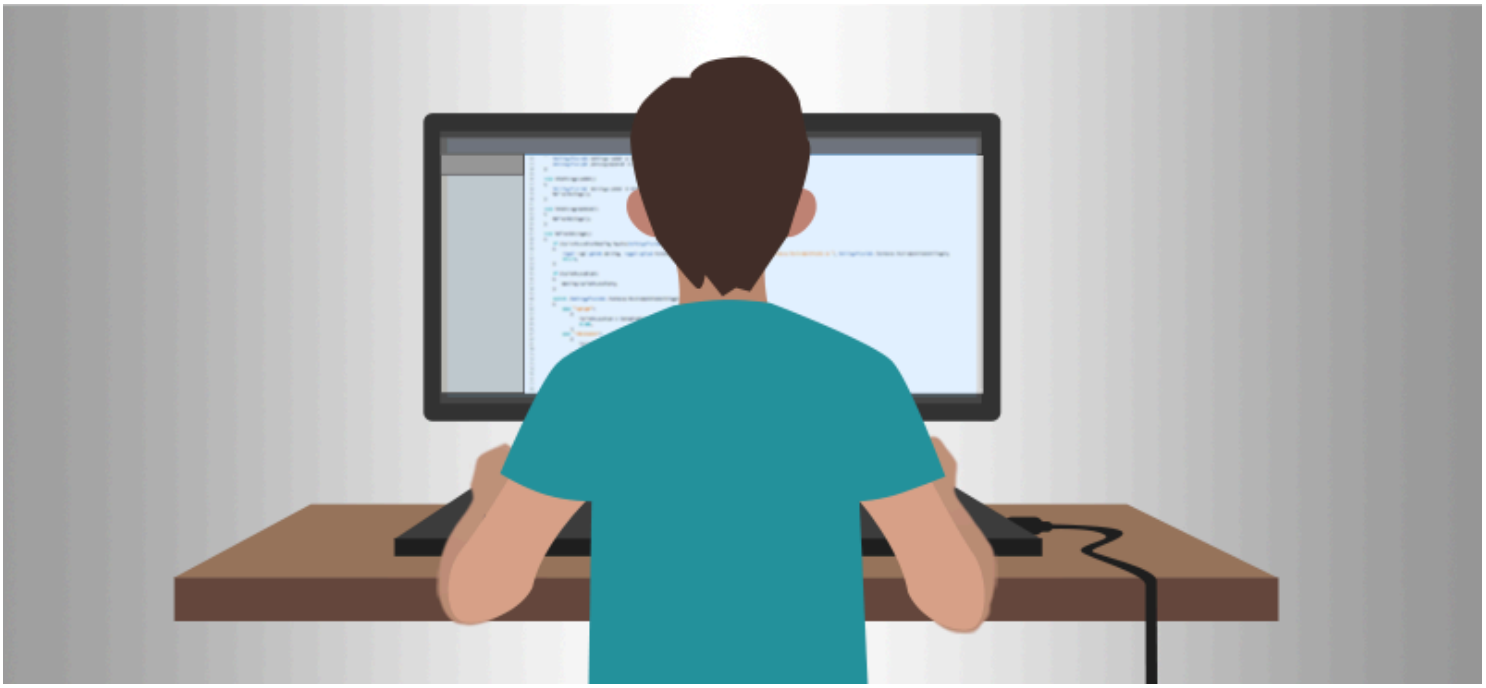
- Platforms index
- Game settings

Scripts

Scripts are units of code that handle game events, respond to user input, and control entities. In short, scripts make games interactive by adding gameplay.

You use scripts by adding them to entities in the scene as components. Stride loads a script when the entity it is added to is loaded in the scene.

Stride scripts are written in **C#**. You can edit scripts in Game Studio or another IDE (such as Visual Studio). Scripts are debugged in Visual Studio.



> **NOTE**
>
> Explaining C# is out of the scope of this documentation.

Scripts have access to the main modules of the Stride engine:

- Audio: the audio system
- Content: loads and saves content from assets
- DebugText: prints debug text
- EffectSystem: loads and compiles effects and shaders
- Game: accesses all information related to your game
- GraphicsDevice: low-level graphics device to create GPU resources
- Input: keyboard, mouse and gamepad states and events
- Log: logs messages and errors from scripts
- SceneSystem: the currently displayed scene

- Script: accesses the script manager to schedule or wait for the termination of scripts
- Services: a registry of services you can use to register your own services
- SpriteAnimation: animates sprites
- Streaming: streams content

You can still use standard C# classes in Stride, but these aren't called scripts and you can't attach them to entities in Game Studio.

In this section

- Types of script
- Create a script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Events
- Debugging
- Preprocessor variables
- Create a model from code
- Create Gizmos for your components
- Create Custom Assets
- Best Practice

Types of script

Beginner Programmer

There are three main types of script in Stride: **startup scripts**, **synchronous scripts**, and **asynchronous scripts**.

When you write your script, inherit from the type of script with the behavior that best fits your needs.

Startup scripts

Startup scripts only run when they are added or removed at runtime. They're mostly used to initialize game elements (eg spawning characters) and destroy them when the scene is unloaded. They have a Start method for initialization and a Cancel method. You can override either method if you need to.

Example:

```
public class StartUpScriptExample : StartupScript
{
  public override void Start()
  {
    // Do some stuff during initialization
  }
}
```

Synchronous scripts

Synchronous scripts are initialized, then updated every frame, and finally canceled (when the script is removed).

- The initialization code, if any, goes in the Start method.
- The code performing the update goes in the Update method.
- The code performing the cancellation goes in the Cancel method.

The following script performs updates every frame, no matter what:

```
public class SampleSyncScript : SyncScript
{
  public override void Update()
  {
    // Performs the update on the entity – this code is executed every frame
  }
}
```

Asynchronous scripts

Asynchronous scripts are initialized only once, then canceled when removed from the scene.

- Asynchronous code goes in the Execute function.
- Code performing the cancellation goes in the Cancel method.

The following script performs actions that depend on events and triggers:

```
public class SampleAsyncScript : AsyncScript
{
  public override async Task Execute()
  {
    // The initialization code should come here, if necessary
    // This method starts running on the main thread

    while (Game.IsRunning) // loop until the game ends (optionalpendingon the script)
    {
      // We're still on the main thread

      // Task.Run will pause the execution of this method until the task is completed,
      // while that's going on, the game will continue running, it will display new frames
and process inputs appropriately
      var lobbies = await Task.Run(() => GetMultiplayerLobbies());

      // After awaiting a task, the thread the method runs on will have changed,
      // this method now runs on a thread pool thread instead of the main thread
      // You can manipulate the data returned by the task here if needed
      // But if you want to interact with the engine safely, you have to make sure the
method runs on the main thread

      // await Script.NextFrame() yields execution of this method to the main thread,
      // meaning that this method is paused, and once the main thread processes the
next frame,
      // it will pick that method up and run it
      await Script.NextFrame();
      // So after this call, this method is back on the main thread

      // You can now safely interact with the engine's systems by displaying the lobbies
retrieved above in a UI for example
    }
  }
}
```

Check out an example from our Async scripts tutorial.

See also

- Create a script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Events
- Debugging
- Preprocessor variables

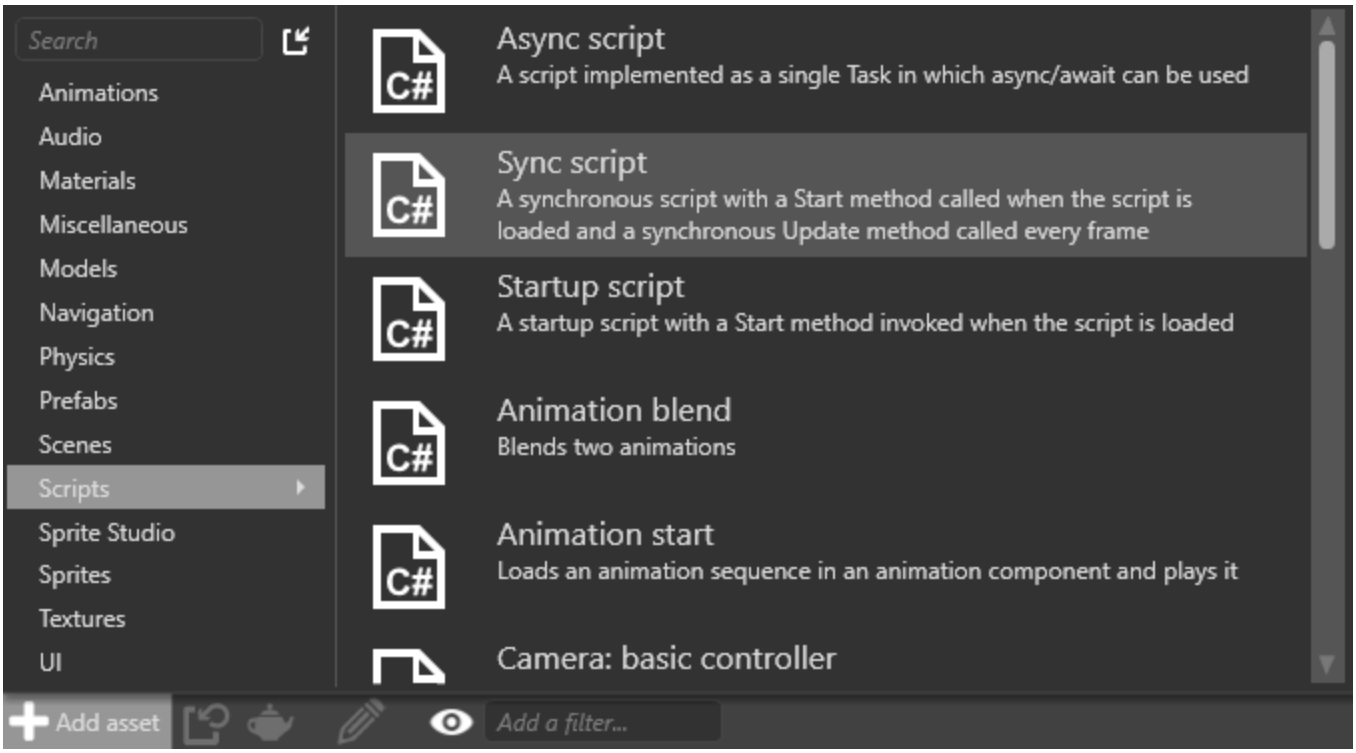
Create a script

Beginner Programmer

You can create scripts using Game Studio or an IDE such as Visual Studio.

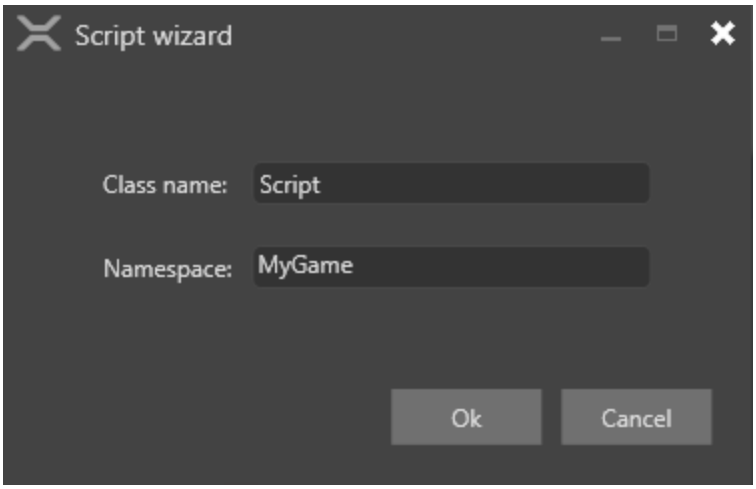
Create a script in Game Studio

1. In the **Asset View**, click **Add asset** > **Scripts** and select a script type.



> **NOTE**
>
> For information about different types of script, see Types of script.

The **New script** dialog opens.



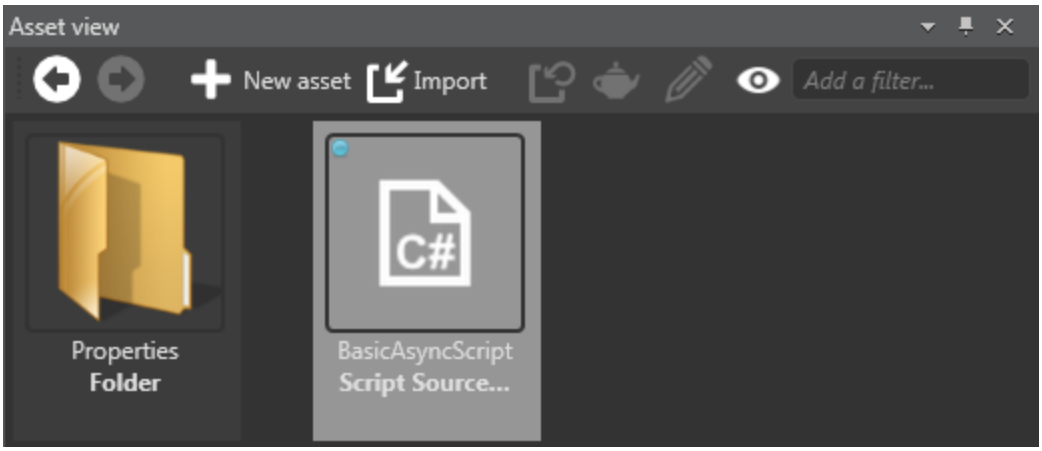
2. Specify a class and namespace for the script and click **Create script**.

3. To use the script, you need to save it. By default, Game Studio prompts you to save the script now.

After you save the script, you can see it in the **Asset View**.

> **NOTE**
>
> Although scripts are a kind of asset, they're not saved in the Assets folder. Instead, they're saved in the relevant assembly folder. For more information, see Project structure.

You can also see the new script in Visual Studio.



> **TIP**
>
> To open your solution in Visual Studio from Game Studio, click the (**Open in IDE**) icon in the Game Studio toolbar.

```
using System;
using System.Text;
using System.Threading.Tasks;
using Stride.Core.Mathematics;
using Stride.Input;
using Stride.Engine;

namespace MyGame
{
        public class BasicAsyncScript : AsyncScript
        {
                public override async Task Execute()
                {
                        while(Game.IsRunning)
                        {
                                // Do some stuff every frame
                                await Script.NextFrame();
                        }
                }
        }
}
```

Create a script in Visual Studio

1. Open Visual Studio.

   > **TIP**
   >
   > To open your solution in Visual Studio from Game Studio, click the (**Open in IDE**) icon in the Game Studio toolbar.

   The game solution is composed of several projects:

   - The project ending *.Game* is the main project, and should contain all your game logic and scripts.
   - Other projects (eg *MyGame.Windows*, *MyGame.Android* etc) contain platform-specific code.

   For more information, see Project structure.

2. Add a new class file to the `.Game` project. To do this, right-click the project and select **Add > New Item**.

The **Add New Item** dialog opens.

3. Select **Class**, type a name for your script, and click **Add**.

   Visual Studio adds a new class to your project.

4. In the file you created, make sure the script is public and derives from either **AsyncScript** or **SyncScript**.

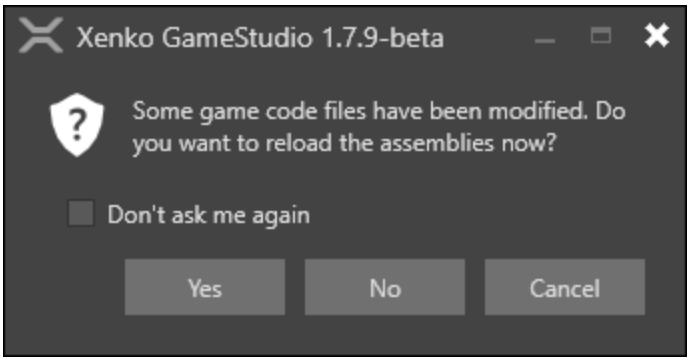
5. Implement the necessary abstract methods.

   For example:

```
using System;
using System.Text;
using System.Threading.Tasks;
using Stride.Core.Mathematics;
using Stride.Input;
using Stride.Engine;

namespace MyGame
{
        public class SampleSyncScript : SyncScript
        {
                public override void Update()
                {
                        if (Game.IsRunning)
                        {
                                // Do something every frame
                        }
                }
        }
}
```

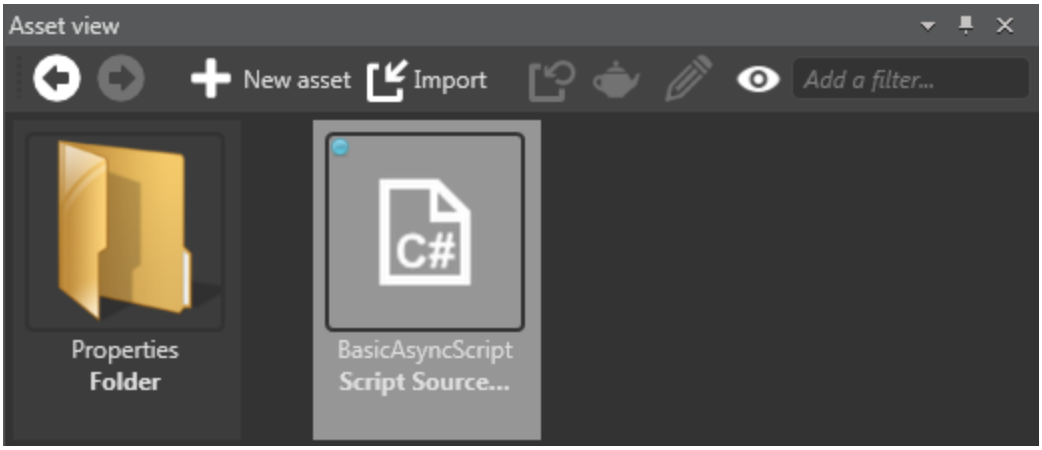
6. Save the project and script files.

7. Because you modified the script, Game Studio needs to reload the assembly to show the changes.



Click **Yes**.

You can see the script in the **Asset View**.



> **NOTE**
>
> Although scripts are a kind of asset, they're not saved in the Assets folder. Instead, they're saved in the relevant assembly folder. For more information, see Project structure.

See also

- Best Practice
- Types of script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Events
- Debugging
- Preprocessor variables

Use a script

Beginner Programmer

To use a script, add it to an entity as a component. You can do this in Game Studio or in code. Stride runs scripts when the entity they are attached to loads.

You can add a single script to as many entities as you need. You can also add multiple scripts to single entities; in this case, Stride creates multiple instances of the script. This means the same script can have different values in its public properties and fields.

Add a script in Game Studio

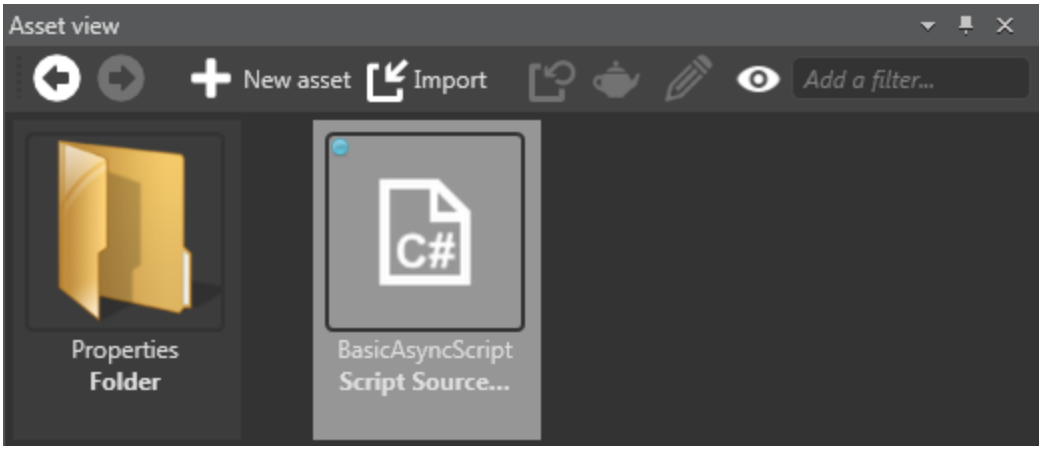
There are three ways to add scripts to entities in Game Studio:

- drag the script from the asset view to the **entity tree**
- drag the script from the asset view to the **entity properties**
- add the script in the **property grid**

Drag to the entity tree

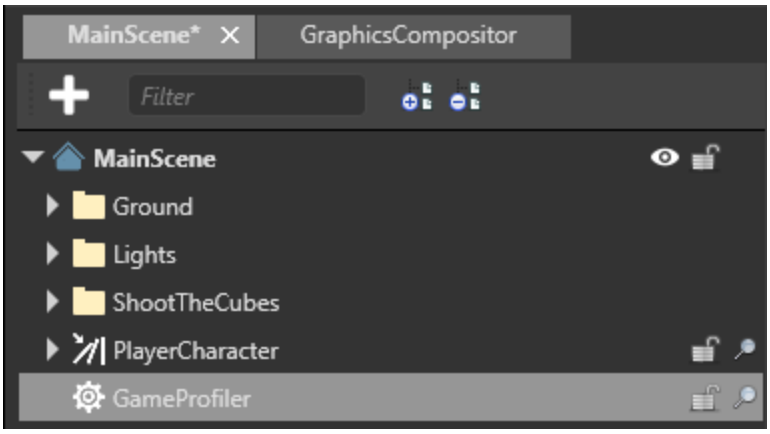
This method automatically creates a new entity that contains the script.

1. In the **solution explorer** (in the bottom left by default), select the assembly that contains your script. Game Studio shows your script in the **asset view**.



2. Drag the script from the asset view to the **entity tree**.

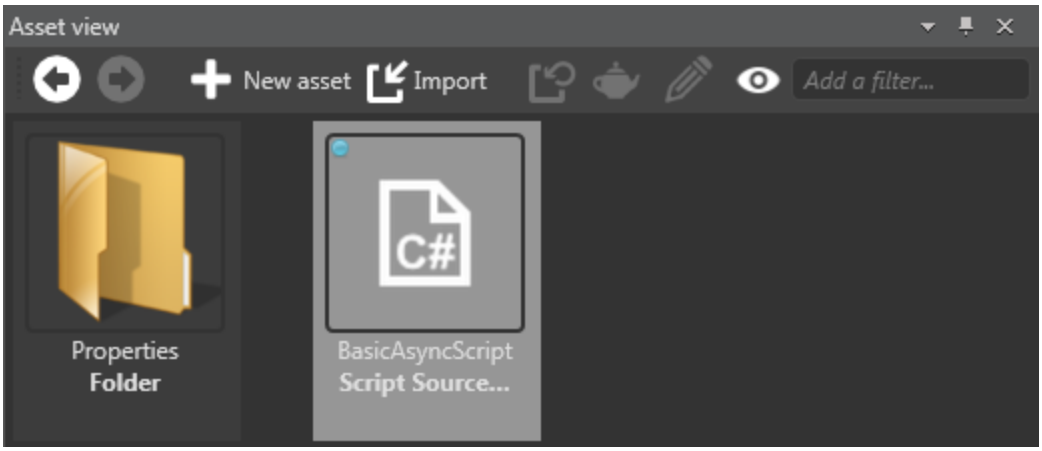
Game Studio adds an entity to your scene, with the script as a component on the entity.



Drag to the property grid

1. In the **entity tree** (on the left by default), or in the scene, select the entity you want to add the script to.

2. In the **solution explorer** (in the bottom left by default), select the assembly that contains your script. Game Studio shows your script in the **asset view**.

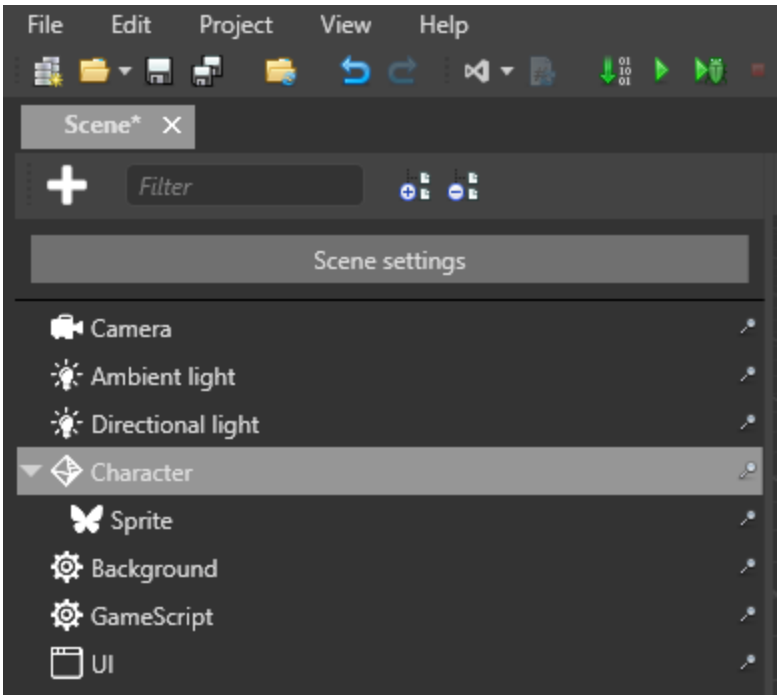


3. Drag the script from the **asset view** to the **property grid**.

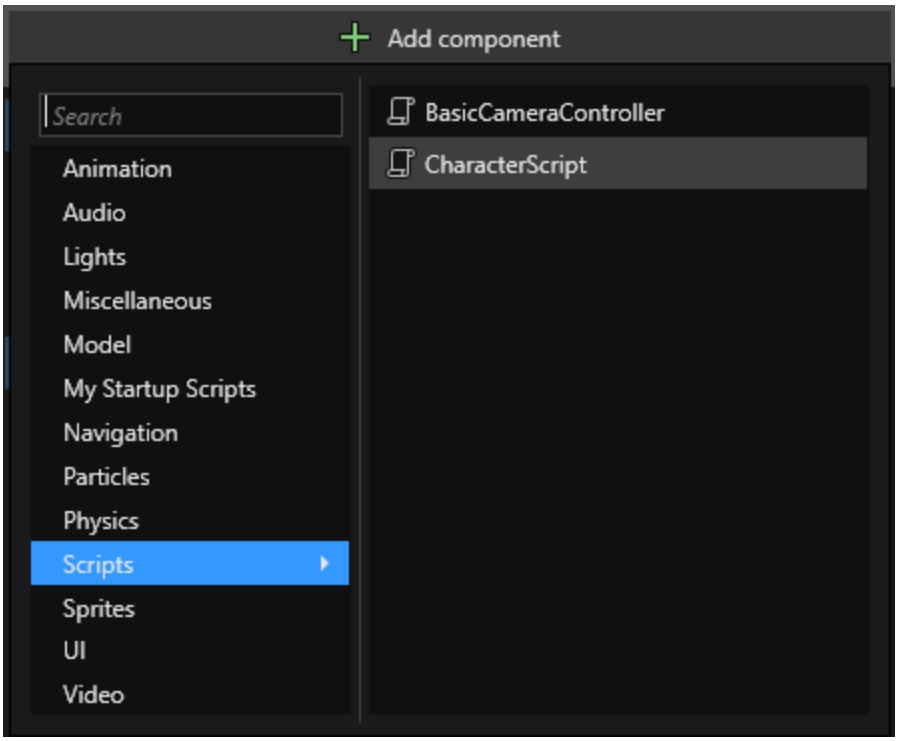
   Game Studio adds the script to the entity.

Add the script in the property grid

1. In the **scene editor**, select the entity you want to add the script to.



2. In the **property grid** (on the right by default), click **Add component** and select the script you want to add.



Game Studio adds the script to the entity.

> **NOTE**
>
> You can customize where scripts appear in the dropdown using the `ComponentCategoryAttribute`:

```
[ComponentCategory("My Startup Scripts")]
public class SampleStartupScript : StartupScript
{
    public override void Start()
    {
        // Do some stuff during initialization
    }
}
```

Add a script from code

The code below adds a script to an entity.

```
// myEntity is an existing entity in the scene; myAsyncScript is the script you want to add to the entity
myEntity.Add(new myAsyncScript());
```

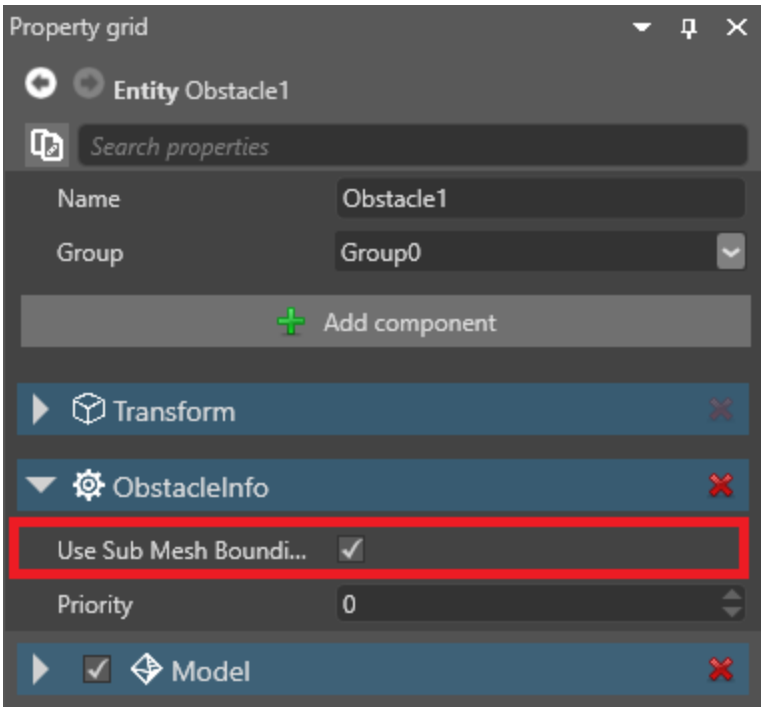
See also

- Types of script
- Create a script
- Public properties and fields
- Scheduling and priorities
- Events
- Debugging
- Preprocessor variables

Public properties and fields

Beginner Programmer

When you declare a public property or field in a script, the property becomes accessible in Game Studio from the script component properties.



You can attach the same script to multiple entities and set different property values on each entity.

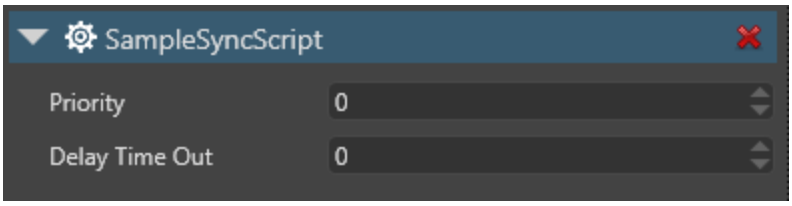
> **NOTE**
>
> Properties and fields must be serializable to be used in Game Studio.

Add a public property or field

This script has a public property (`DelayTimeOut`):

```
public class SampleSyncScript : StartupScript
{
        // This public member will appear in Game Studio
        public float DelayTimeOut { get; set; }
}
```

Game Studio shows the `DelayTimeOut` property in the script component properties:



> **NOTE**
>
> As a general rule, if you want to display the property or field in Game Studio, getters and setters should do as little as possible. For example, they shouldn't try to call methods or access Stride runtime API.

For example, the following code will create problems, as it tries to access `Entity.Components`, which is only available at runtime:

```
public class SampleSyncScript : StartupScript
{
        private float delayTimeOut;
        // This public member will appear in Game Studio
        public float DelayTimeOut
        {
                get { return delayTimeOut; }
                set
                {
                        delayTimeOut = value;
                        Entity.Components.Add(new SkyboxComponent());
                }
        }
}
```

If you want to include code like this in a property or field, hide it so Game Studio doesn't display it (see below).

Hide properties or fields in the Property Grid

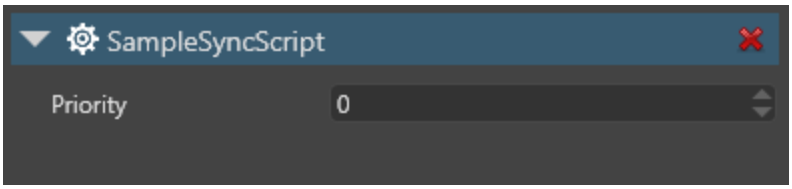
If you don't want Game Studio to show a property in the Property Grid, you can:

- declare your member internal or private, or
- use the DataMemberIgnore attribute like this:

```
// This public property isn't available in Game Studio
[DataMemberIgnore]
```

```
public float DelayTimeOut { get; set; }
```

Game Studio no longer shows the property:



Adding property descriptions

When you add a `<userdoc>` comment block above your public property in code, Game Studio will display it in the description field.

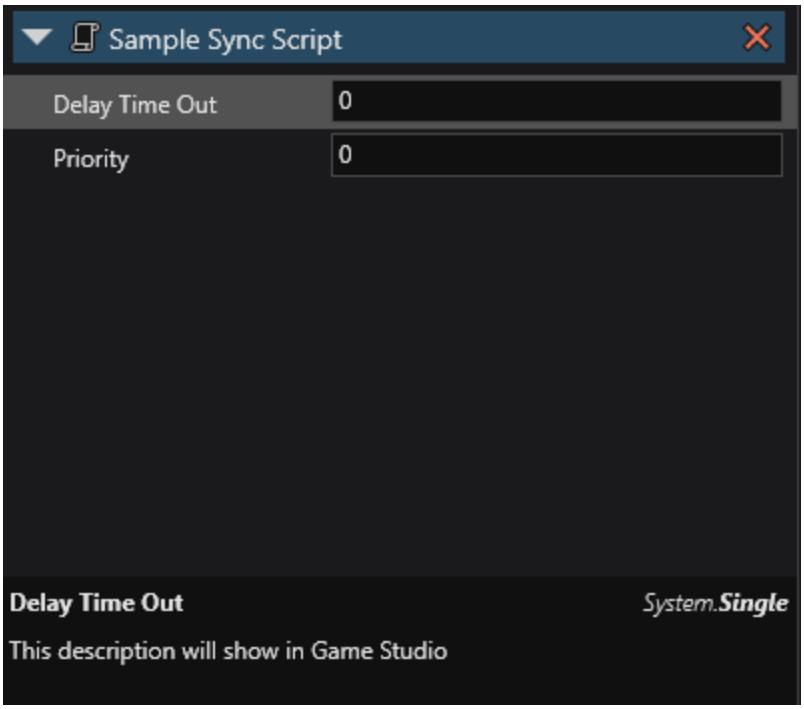
```
///<summary>
/// This summary won't show in Game Studio
///</summary>
///<userdoc>
/// This description will show in Game Studio
///</userdoc>
public float DelayTimeOut { get; set; }
```

Enable documentation file generation:

```
<PropertyGroup>
  <TargetFrameworks>net6.0</TargetFrameworks>

<DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</Documentatio
nFile>
</PropertyGroup>
```

> **NOTE**
>
> Game Studio will only look in your build output directory for each assembly. Using the above path is recommended.

On next reload, the Game Studio should display the documentation:



MemberRequiredAttribute

The `MemberRequiredAttribute` is used to specify if a field or property should not be left null in the editor. If no values are set for this member, a warning or error will be logged when building your game.

```
[Stride.Core.Annotations.MemberRequired(MemberRequiredReportType.Error)] public
CharacterComponent MyCharacter;
```

Additional Serialization Attributes

- `DataMemberRangeAttribute`
- `InlinePropertyAttribute`
- `ItemCanBeNullAttribute`
- `ItemNotNullAttribute`
- `MemberCollectionAttribute`
- `DataStyleAttribute`
- `DisplayAttribute`

See also

- Serialization
- Types of script
- Create a script
- Use a script
- Scheduling and priorities
- Events

- Debugging
- Preprocessor variables

Serialization

Beginner Programmer

The editor and serialization system uses four attributes to determine what is serialized and visible in the editor.

DataContractAttribute

Adding the `DataContractAttribute` to your `class` or `struct` notifies the serializer that it should serialize the data it contains, and the editor that it should display fields and properties of that type, along with the scenes or assets that might include it.

```
[Stride.Core.DataContract(Inherited = true)]
public class MySerializedClass
{
    public float MyValue;
}

// 'DataContract' is inherited above. You don't need to specify it for a derived class.
public class MyDerivedSerializedClass : MySerializedClass
{
    public string MyName;
}
```

> **NOTE**
>
> Your IDE may wrongfully add `System.Runtime.Serialization` to your list of `using` when adding `DataContract`. They are not interchangeable. Make sure that your `DataContract` comes from `Stride.Core`, specifying the namespace explicitly like shown above if necessary.

DataMemberAttribute

The `DataMemberAttribute` notifies the editor and serializer that the property or field on this `DataContract`'ed `class` or `struct` should be serialized. Note that you can omit this attribute for most public fields and properties, as they will be included by default. See Fields and Properties for specifics.

```
[Stride.Core.DataContract]
public class MySerializedClass
{
    [Stride.Core.DataMember]
```

```
    internal float MyValue;
}
```

DataAliasAttribute

The `DataAliasAttribute` can be used to ensure you do not break previously serialized data whenever you have to change how that member is named in your source.

```
[Stride.Core.DataAlias("PreviousNameOfProp")]
public string MyRenamedProp { get; set; }
```

> **NOTE**
>
> Alias remaps values only while in the editor; this feature is specific to the YAML serialization system. Alias will be ignored during builds and at runtime.

DataMemberIgnoreAttribute

The `DataMemberIgnoreAttribute` notifies the editor and serializer that the property or field on this `DataContract`'ed `class` or `struct` should ***NOT*** be serialized.

```
[Stride.Core.DataContract]
public class MySerializedClass
{
    [Stride.Core.DataMemberIgnore]
    public float MyValue { get; set; } // This public property will NOT show up in
the editor
}
```

TODO

- DataMemberCustomSerializerAttribute
- DataMemberUpdatableAttribute

Rule of Thumb

Serialization and the editor's access and view of your properties mirrors how access modifiers work in C#.

Think of the serializer/editor as being a class external to your codebase, if you want the serializer to read and write your properties you have to ensure that the access modifiers for its getter and setter allows the serializer to access them.

If you're hiding that property behind an `internal` access modifier, you have to annotate it with the attribute to ensure it is visible to the serializer.

Fields

```
// Read and set in the editor by default
public object obj;

// Read and set in editor with attribute
[DataMember] public internal object obj;

// Read only fields cannot be modified to point at another object, but the currently set
object may be modified
public readonly object obj;
[DataMember] internal readonly object obj;

// Never
private protected/private/protected object obj;
```

Properties

```
// Read and set in the editor ...

// By default
public object obj { get; set; }
public object obj { get => x; set => x = value; }

// Forced with the attribute for 'internal' modifiers
[DataMember] public object obj { internal get; public/internal set; }
[DataMember] public object obj { internal get => x; public/internal set => x; }

// Read only
public object obj { get; }

// Read only for any non-public access modifier
public object obj { get; internal/private protected/private/protected set; }
public object obj { get => x; internal/private protected/private/protected set => x =
value; }

// Read only for internal properties must be enforced through the attribute
[DataMember] internal object obj { get; }
[DataMember] internal object obj { get => x; }

// Read only, special case for get-only public properties without backing field,
```

```
//They must use the attribute to be deserialized, see the comment below
[DataMember] public object obj { get => x; }

// Read only for access modifiers more restrictive than internal, even with the attribute
[DataMember] public object obj { internal get; private protected/private/protected set; }
[DataMember] public object obj { internal get => x; private protected/private/protected set
=> x; }

// Never
private protected/private/protected object obj { get; set; }
private protected/private/protected object obj { get => x; set => x; }
```

> **NOTE**
>
> Get-only public properties without backing field (`public object obj { get => x; }`) are not serialized by default as they are, more often than not, shortcuts to values of another object or used purely as a function. It might make more sense to change it to `{ get; } = new MyObject();` or `{ get; init; }` if you want to serialize it, and if that doesn't work for you, feel free to add the attribute to enforce serialization.

What about init ?

The `init` access modifier is seen as a `public set` by the editor and serialization, it will be settable in the editor.

See also

- Public properties and fields

Scheduling and priorities

Beginner Programmer

Stride doesn't run scripts simultaneously; they run one at a time. Where scripts depend on each other, you should make sure they run in the correct order by giving them priorities.

Priorities apply to all kinds of scripts. This means that, for example, synchronous and asynchronous scripts don't have separate priority lists. They both join the same queue.

Scripts with lower priority numbers have higher priorities. For example, a script with priority 1 runs before a script with priority 2, and a script with priority -1 runs before a script with priority 1. By default, scripts have a priority of 0.

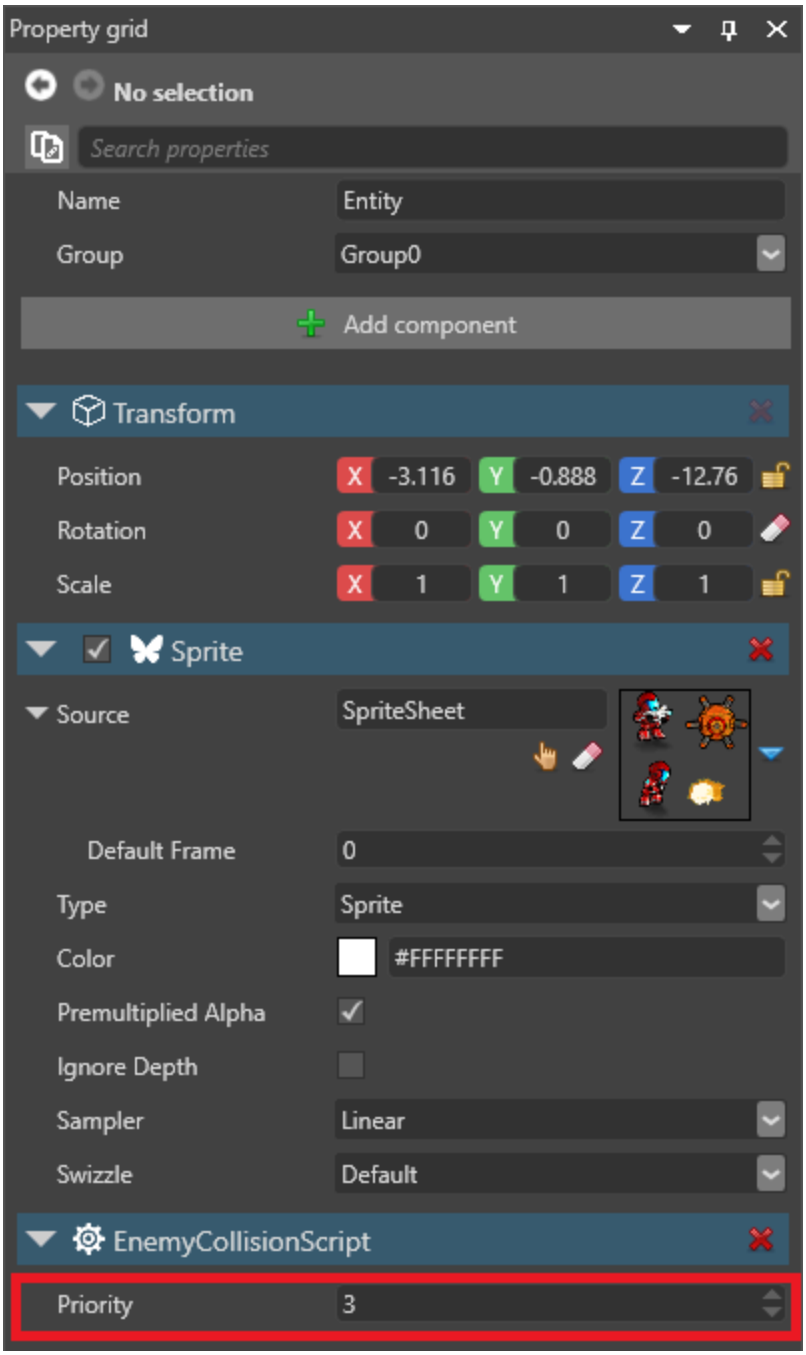
If scripts have the same priority, the order in which Stride runs them isn't deterministic. You might give scripts the same priority if you don't care which order they run in.

> **NOTE**
> Currently, there's no way to see a list of priorities in one place. You have to set each priority of each script individually in the script component properties.

Set a script priority

Priorities aren't set in the scripts themselves. Instead, they're set in the script component properties on the entity the script is attached to.

1. Attach the script to an entity. For information about how to do this, see Use a script.
2. With the entity selected, in the **Property Grid**, under the **script component properties**, set the **Priority** you want the script to have.



See also

- Types of script
- Create a script
- Use a script
- Public properties and fields
- Events
- Debugging
- Preprocessor variables

Events

> **NOTE**
>
> Events are not recommended anymore, see Best Practice

Intermediate Programmer

Events facilitate communication between scripts. They work one-way, broadcast from **broadcasters** to **receivers**. Events come in two flavors, a non-generic version for basic event broadcasting, and a generic version for when data needs to be passed to receivers.

For example, imagine your game has a "Game Over" state that occurs when the player dies. To handle this, you can create a "Game Over" event, which is broadcast to all scripts with receivers listening for the event. When the event is broadcast, the receivers run appropriate scripts to handle the Game Over event (eg reset enemies, replace level objects, start a new timer, etc). You can also send information related to the "Game Over" state (eg game stats, who won, etc).

> **NOTE**
>
> Events are handled entirely in scripts. You can't configure them in Game Studio.

Create and broadcast an event

Broadcasters in the Stride API are of type EventKey or EventKey<T>. They use the method Broadcast or Broadcast(T) to broadcast events to receivers.

For example, this code creates two "Game Over" events. One with a non-generic and the other with a generic version of EventKey:

```
public static class GlobalEvents
{
    public static EventKey GameOverEventKey = new EventKey("Global", "Game Over");
    public static EventKey<string> GameOverWithDataEventKey = new EventKey<string>("Global",
"Game Over With Data");

    public static void SendGameOverEvent()
    {
        GameOverEventKey.Broadcast();
        GameOverWithDataEventKey.Broadcast("Player 1");
```

```
    }
}
```

Create a receiver

Receivers in the Stride API are of type EventReceiver or EventReceiver<T>.

To receive the "Game Over" events described above, use:

```
var gameOverListener = new EventReceiver(GlobalEvents.GameOverEventKey);
var gameIsOver = gameOverListener.TryReceive();

var gameOverListenerWithData = new EventReceiver<string>
(GlobalEvents.GameOverWithDataEventKey);
if(gameOverListenerWithData.TryReceive(out string data))
{
        //data == "Player 1"
}

//Or in Async
await gameOverListener.ReceiveAsync();
string asyncData = await gameOverListenerWithData.ReceiveAsync();
//asyncData == "Player 1"
```

See also

- Best Practice
- Types of script
- Create a script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Debugging
- Preprocessor variables

Debugging

Beginner Programmer

If your script isn't producing the expected result at runtime, you can debug it in an IDE such as Visual Studio.

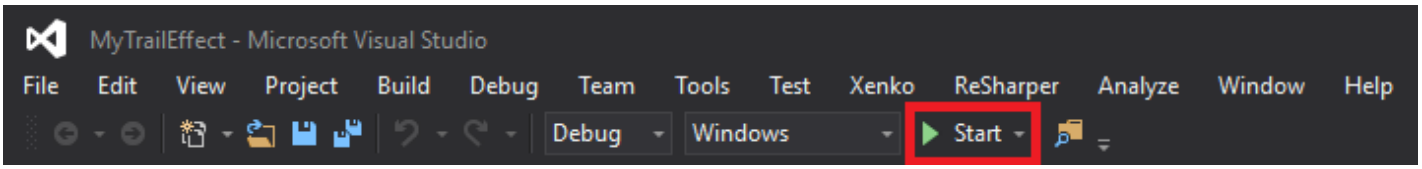
> **NOTE**
> There are lots of ways to debug code. This page suggests one method, using Visual Studio.

1. Open the script in Visual Studio.

   > **TIP**
   > To open your project in Visual Studio from Game Studio, in the Game Studio toolbar, click (**Open in IDE**).

2. Press **F9** to add a break point at the required places.

3. In Visual Studio, press **F5** or click **Start** in the toolbar to run the game in debug mode.



   The game starts in a new window. In Visual Studio, on the script page, the first break point highlights and stops the execution.

4. Verify the state of your variables.

5. Press **F10** (step over) if you want to execute the code line by line, or press **F5** to continue code execution.

> **NOTE**
> If Visual Studio doesn't stop at the break point, make sure you attached the script to an entity in the active scene.

For more information about debugging in Visual Studio, see the MSDN documentation.

See also

- Debugging in Visual Studio (MSDN documentation)
- Types of script
- Create a script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Events
- Preprocessor variables

Preprocessor variables

Advanced Programmer

If you're developing for multiple platforms, you often need to write custom code for each platform. In most cases, the best way to do this is to use Platform.Type and GraphicsDevice.Platform. Alternatively, you can use **preprocessor variables**.

> **WARNING**
> We recommend you avoid using preprocessor variables where possible, and instead use Platform.Type and GraphicsDevice.Platform. This is because you might miss errors in your code, as only code for your target platform is checked at compile time.

Platforms

Variable	Value
STRIDE_PLATFORM_WINDOWS	Windows (standard and RT)
STRIDE_PLATFORM_WINDOWS_DESKTOP	Windows (non-RT)
STRIDE_PLATFORM_MONO_MOBILE	Xamarin.iOS or Xamarin.Android
STRIDE_PLATFORM_ANDROID	Xamarin.Android
STRIDE_PLATFORM_IOS	Xamarin.iOS

Graphics APIs

Variable	Value
STRIDE_GRAPHICS_API_DIRECT3D	Direct3D 11
STRIDE_GRAPHICS_API_OPENGL	OpenGL (Core and ES)
STRIDE_GRAPHICS_API_OPENGLCORE	OpenGL Core (Desktop)
STRIDE_GRAPHICS_API_OPENGLES	OpenGL ES
STRIDE_GRAPHICS_API_VULKAN	Vulkan

Example

```
#if STRIDE_PLATFORM_WINDOWS
    // Windows-specific code goes here...

#elif STRIDE_PLATFORM_MONO_MOBILE
    // iOS and Android-specific code goes here...

#else
    // Other platform code goes here...

#endif
```

See also

- Platforms
- Types of script
- Create a script
- Use a script
- Public properties and fields
- Scheduling and priorities
- Events
- Debugging

Create a model from code

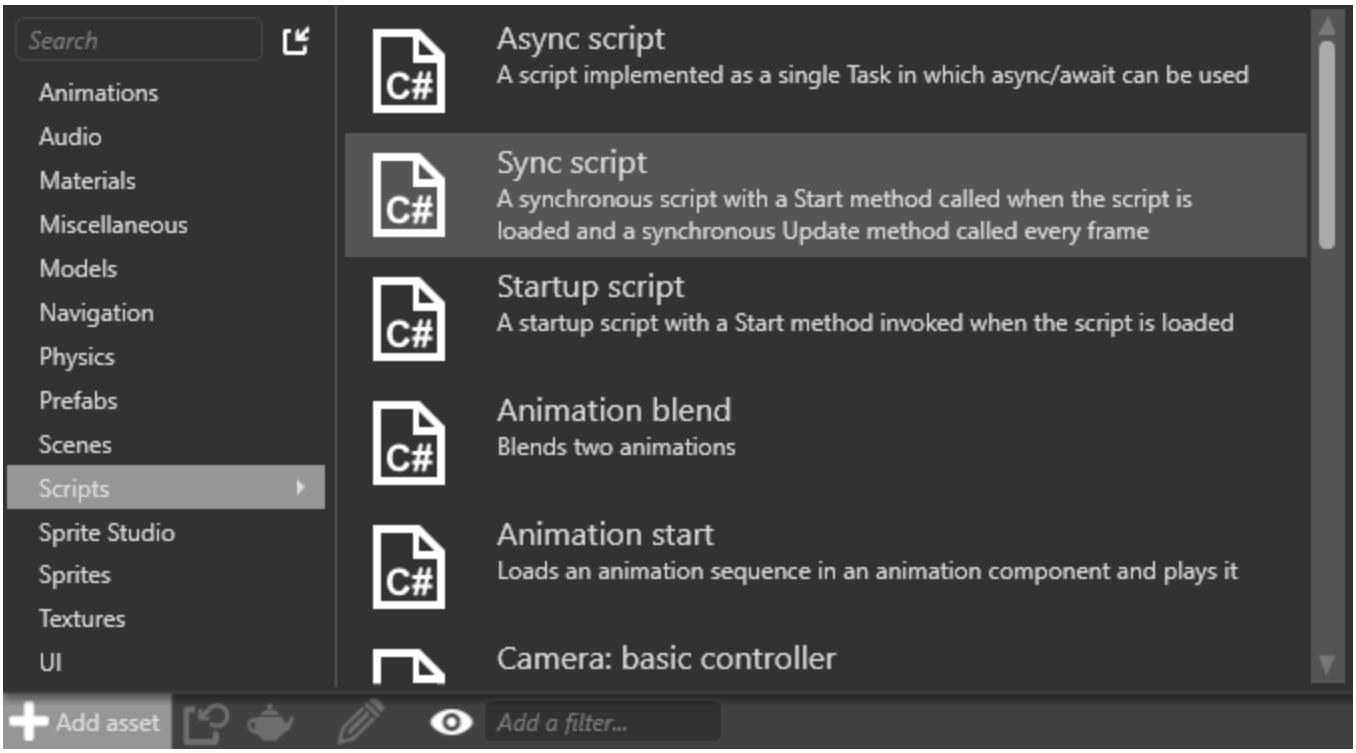
Beginner Programmer

You can create models in scripts at runtime. You can do this in several different ways, including:

- creating a model from an asset
- creating a procedural model using built-in geometric primitives (eg a sphere or cube)
- instantiating a prefab that contains a model (see Use prefabs)

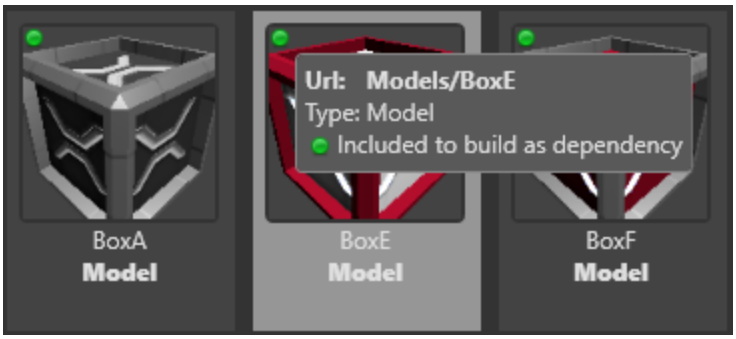
Create a model from an asset

1. Create a new, empty synchronous script. For full instructions, see Create a script.

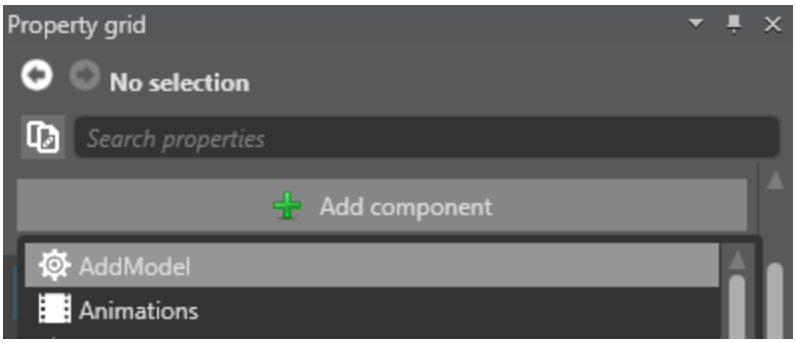


2. In the script, load the model using its asset URL. For example:

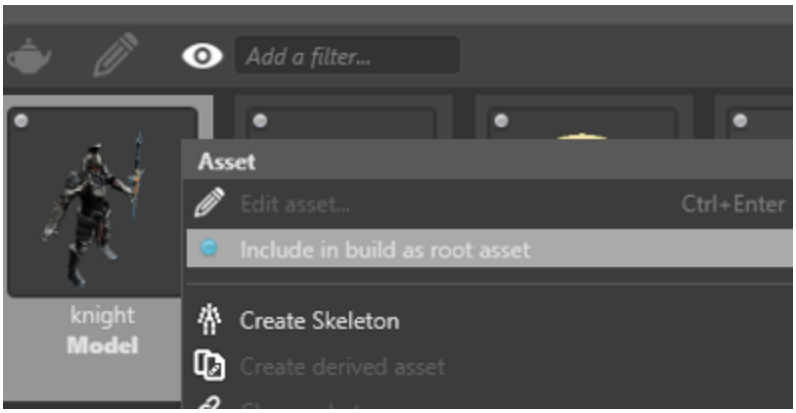
```
// Create a new entity and add it to the scene.
var entity = new Entity();
SceneSystem.SceneInstance.RootScene.Entities.Add(entity);

// Add a model included in the game files.
var modelComponent = entity.GetOrCreate<ModelComponent>();
modelComponent.Model = Content.Load<Model>("MyFolder/MyModel");
```

> **TIP**
>
> To find the model's asset URL, in the **Asset View**, move the mouse over the model.
>
> 
>

3. Add the script as a **script component** to any entity in the scene. It doesn't matter which entity you use. For instructions, see Use a script.



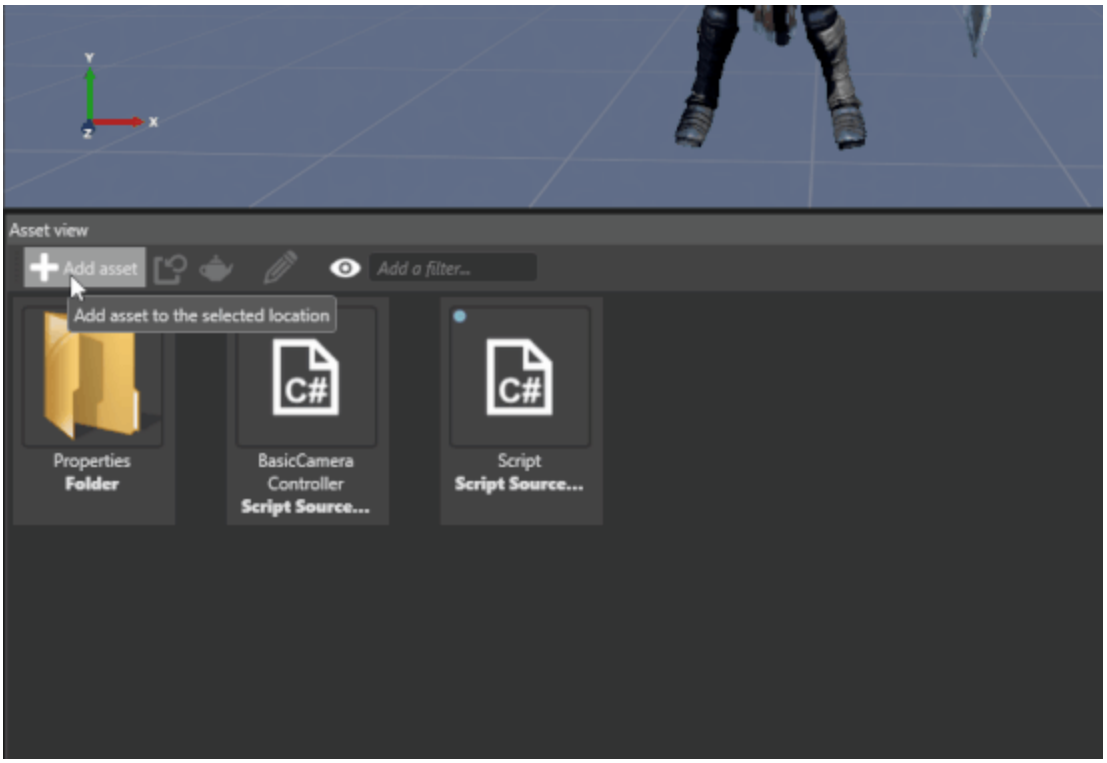
4. In the **Asset View**, right-click the model you want to create at runtime and select **Include in build as root asset**.



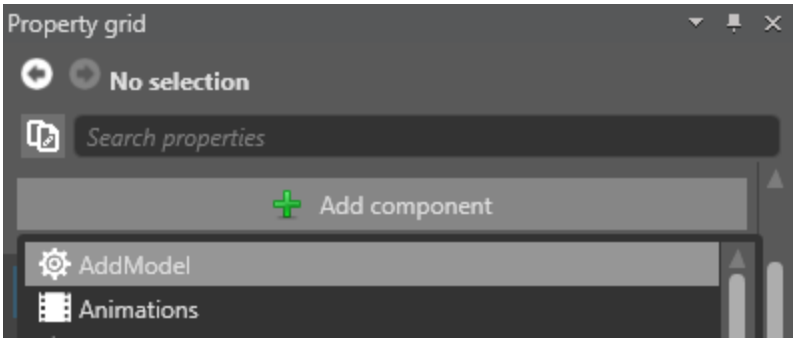
This makes sure the asset is available for the script to use at runtime. For more information, see Manage assets.

Create a procedural model

1. Create a new, empty synchronous script. For full instructions, see Create a script.



2. Add the script as a **script component** to any entity in the scene. It doesn't matter which entity you use. For instructions, see Use a script.



3. In your script, instantiate an empty entity and an empty model. For example:

```
// Create an entity and add it to the scene.
var entity = new Entity();
SceneSystem.SceneInstance.RootScene.Entities.Add(entity);

// Create a model and assign it to the model component.
var model = new Model();
entity.GetOrCreate<ModelComponent>().Model = model;
```

4. In your script, create a procedural model using built-in geometric primitives (eg a sphere or cube). For example:

```
// Add one or more meshes using geometric primitives (eg spheres or cubes).
var meshDraw = GeometricPrimitive.Sphere.New(GraphicsDevice).ToMeshDraw();
```

```
var mesh = new Mesh { Draw = meshDraw };
model.Meshes.Add(mesh);
```

> **NOTE**
>
> To use the code above, make sure you add `using Stride.Extensions` to the top of your script.

Alternatively, create a mesh using your own vertex and index buffers. For example:

```
// Create a mesh using your own vertex and index buffers.

mesh = new Mesh { Draw = new MeshDraw { /* Vertex buffer and index buffer setup */ } };
model.Meshes.Add(mesh);
```

5. Here is a more complete example that draws a custom triangle..

```
var vertices = new VertexPositionTexture[3];
vertices[0].Position = new Vector3(0f,0f,1f);
vertices[1].Position = new Vector3(0f,1f,0f);
vertices[2].Position = new Vector3(0f,1f,1f);
var vertexBuffer = Stride.Graphics.Buffer.Vertex.New(GraphicsDevice, vertices,
                                                     GraphicsResourceUsage.Dynamic);
int[] indices = { 0, 2, 1 };
var indexBuffer = Stride.Graphics.Buffer.Index.New(GraphicsDevice, indices);

var customMesh = new Stride.Rendering.Mesh
{
    Draw = new Stride.Rendering.MeshDraw
    {
        /* Vertex buffer and index buffer setup */
        PrimitiveType = Stride.Graphics.PrimitiveType.TriangleList,
        DrawCount = indices.Length,
        IndexBuffer = new IndexBufferBinding(indexBuffer, true, indices.Length),
        VertexBuffers = new[] { new VertexBufferBinding(vertexBuffer,
                                    VertexPositionTexture.Layout,
vertexBuffer.ElementCount) },
    }
};
// add the mesh to the model
model.Meshes.Add(customMesh);
```

> **NOTE**
>
> For more information about how to set up vertex and index buffers, see Drawing vertices.

Finally, you need to give the model one or more materials. There are two ways to do this.

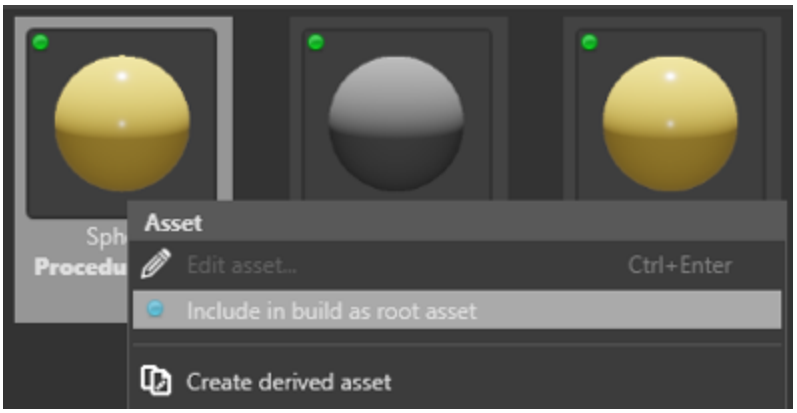
Option 1: load a material in code

1. In your code, load one or more materials and add them to the model. Because models can use multiple materials (one for each mesh in the model), use Mesh.MaterialIndex to specify which materials in the list are used for which mesh.

   For example:

   ```
   // Add one or more materials. Because models might expect multiple materials (one per mesh), Mesh.MaterialIndex specifies which material in the list is used for which mesh.

   Material material = Content.Load<Material>("MyFolder/MyMaterial");
   model.Materials.Add(material);
   ```

2. In the **Asset View**, right-click every material asset your script uses and select `Include in build as root asset`.

   This makes sure the asset is available for the script to use at runtime. For more information, see Manage assets.

Option 2: Create new materials in code

For example:

```
// Create a material (eg with red diffuse color).
var materialDescription = new MaterialDescriptor
{
```

```
    Attributes =
    {
        DiffuseModel = new MaterialDiffuseLambertModelFeature(),
        Diffuse = new MaterialDiffuseMapFeature(new ComputeColor { Key =
MaterialKeys.DiffuseValue })
    }
};
var material = Material.New(GraphicsDevice, materialDescription);
material.Passes[0].Parameters.Set(MaterialKeys.DiffuseValue, Color.Red);
model.Materials.Add(0, material);
```

See also

- Create a script
- Use a script
- Use prefabs

Gizmos

Intermediate Programmer

Gizmos are a tool which you can implement over your components to provide visual assistance for designers when manipulating component values.

Here's an exhaustive example one could implement:

```
using Stride.Core;
using Stride.Core.Mathematics;
using Stride.Engine;
using Stride.Engine.Gizmos;
using Stride.Graphics;
using Stride.Graphics.GeometricPrimitives;
using Stride.Rendering;
using Stride.Rendering.Materials;
using Stride.Rendering.Materials.ComputeColors;

// We will be implementing a Gizmo for the following class
public class MyScript : StartupScript
{

}

// This attribute specifies to the engine that the following gizmo class is bound
to 'MyScript',
// the game studio will pick that up and spawn an instance of that class for each 'MyScript'
in the scene
[GizmoComponent(typeof(MyScript), isMainGizmo:false/*When true, only the first gizmo on an
entity with true is visible, false means that it is always visible*/)]
public class Gizmo : IEntityGizmo
{
    private bool _selected, _enabled;
    private MyScript _component;
    private ModelComponent _model;
    private Material _material, _materialOnSelect;

    // This property is set based on whether the gizmo is globally turned on or off in the
editor's view settings
    public bool IsEnabled
    {
        get
        {
            return _enabled;
```

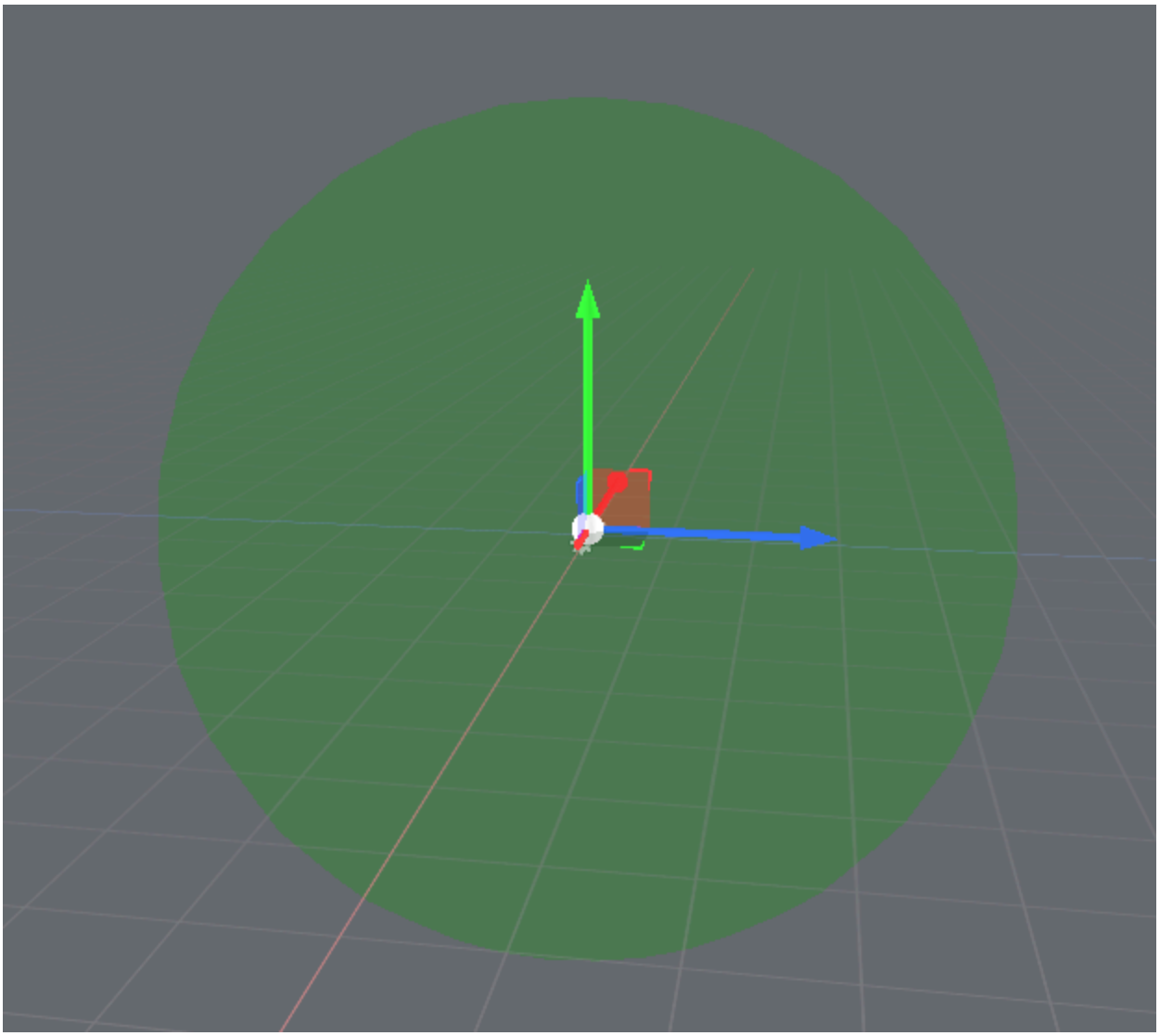
```
        }
        set
        {
            _enabled = value;
            _model.Enabled = _enabled;
        }
    }

    // The size slider value in the view settings pane
    public float SizeFactor { get; set; }

    // The editor will set this property whenever the entity the component is on is selected
    public bool IsSelected
    {
        get
        {
            return _selected;
        }
        set
        {
            _selected = value;
            _model.Materials[0] = _selected ? _materialOnSelect : _material;
            // The logic below shows gizmos for all components when they are on in the gizmo
settings, and when off, only shows the one from the selected entity
            // Removing the line hides gizmos even when selected when the gizmo settings
is off
            _model.Enabled = _selected || _enabled;
        }
    }

    // This constructor is called by the editor,
    // A gizmo class MUST contain a constructor with a single parameter of the
component's type
    public Gizmo(MyScript component)
    {
        _component = component;
    }

    public bool HandlesComponentId(OpaqueComponentId pickedComponentId, out
Entity? selection)
    {
        // This function is called when scene picking/mouse clicking in the scene on a gizmo
        // The engine calls this function on each gizmos, gizmos in term notify the engine
        // when the given component comes from them by returning true, and provide the
editor with the corresponding entity this gizmo represents
        if (pickedComponentId.Match(_model))
```

```
        {
            selection = _component.Entity;
            return true;
        }
        selection = null;
        return false;
    }

    public void Initialize(IServiceRegistry services, Scene editorScene)
    {
        // As part of initialization, we create a model in the editor scene to visualize
the gizmo
        var graphicsDevice = services.GetSafeServiceAs<IGraphicsDeviceService>
().GraphicsDevice;

        // In our case we'll just rely on the GeometricPrimitive API to create a sphere
for us
        // You don't have to create one right away, you can delay it until the component is
in the appropriate state
        // You can also dynamically create and update one in the Update() function
further below
        var sphere = GeometricPrimitive.Sphere.New(graphicsDevice);

        var vertexBuffer = sphere.VertexBuffer;
        var indexBuffer = sphere.IndexBuffer;
        var vertexBufferBinding = new VertexBufferBinding(vertexBuffer, new
VertexPositionNormalTexture().GetLayout(), vertexBuffer.ElementCount);
        var indexBufferBinding = new IndexBufferBinding(indexBuffer,
sphere.IsIndex32Bits, indexBuffer.ElementCount);

        _material = Material.New(graphicsDevice, new MaterialDescriptor
        {
            Attributes =
            {
                Emissive = new MaterialEmissiveMapFeature(new ComputeColor(new
Color4(0.25f,0.75f,0.25f,0.05f).ToColorSpace(graphicsDevice.ColorSpace))) { UseAlpha =
true },
                Transparency = new MaterialTransparencyBlendFeature()
            },
        });
        _materialOnSelect = Material.New(graphicsDevice, new MaterialDescriptor
        {
            Attributes =
            {
                Emissive = new MaterialEmissiveMapFeature(new ComputeColor(new
Color4(0.25f,0.75f,0.25f,0.5f).ToColorSpace(graphicsDevice.ColorSpace))) { UseAlpha =
```

```
true },
                Transparency = new MaterialTransparencyBlendFeature()
            },
        });

        _model = new ModelComponent
        {
            Model = new Model
            {
                (_selected ? _materialOnSelect : _material),
                new Mesh
                {
                    Draw = new MeshDraw
                    {
                        StartLocation = 0,
                        // You can swap to LineList or LineStrip to show the model in
wireframe mode, you'll have to adapt your index buffer to that new type though
                        PrimitiveType = PrimitiveType.TriangleList,
                        VertexBuffers = new[] { vertexBufferBinding },
                        IndexBuffer = indexBufferBinding,
                        DrawCount = indexBuffer.ElementCount,
                    }
                }
            },
            RenderGroup = IEntityGizmo.PickingRenderGroup, // This RenderGroup allows scene
picking/selection, use a different one if you don't want selection
            Enabled = _selected || _enabled
        };

        var entity = new Entity($"{nameof(Gizmo)} for {_component.Entity.Name}"){ _model };
        entity.Transform.UseTRS = false; // We're controlling the matrix directly in
this case
        entity.Scene = editorScene;

        vertexBuffer.DisposeBy(entity);
        indexBuffer.DisposeBy(entity); // Attach buffers to the entity for manual
disposal later
    }

    public void Dispose()
    {
        _model.Entity.Scene = null;
        _model.Entity.Dispose(); // Clear the two buffers we attached above
    }

    public void Update()
```

```
    {
        // This is where you'll update how the gizmo looks based on MyScript's state
        // Here we'll just ensure the gizmo follows the entity it is representing whenever
that entity moves,
        // note that UseTRS is disabled above to improve performance and ensure that there
are no world space issues
        _model.Entity.Transform.LocalMatrix = _component.Entity.Transform.WorldMatrix;
    }
}
```

And the result:



Do note that you may have to restart the editor if it was open while you introduced this new gizmo.

Creating Custom Assets

Stride supports the creation of custom asset types that can be referenced in your scenes as well as reference other assets. To do so, you must add a reference to the `Stride.Core.Assets` package in your game's `.csproj`: Here's how it looks like in a default game project:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>net8.0-windows</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Stride.Engine" Version="4.2.0.1" />
    <PackageReference Include="Stride.Video" Version="4.2.0.1" />
    <PackageReference Include="Stride.Physics" Version="4.2.0.1" />
    <PackageReference Include="Stride.Navigation" Version="4.2.0.1" />
    <PackageReference Include="Stride.Particles" Version="4.2.0.1" />
    <PackageReference Include="Stride.UI" Version="4.2.0.1" />
    <PackageReference Include="Stride.Core.Assets.CompilerApp" Version="4.2.0.1"
IncludeAssets="build;buildTransitive" />

    <PackageReference Include="Stride.Core.Assets" Version="4.2.0.1" />
  </ItemGroup>
</Project>
```

> **WARNING**
>
> Make sure that the version specified for `Stride.Core.Assets` matches the other package's versions.

Inside the same project, create a new csharp file and replace its content with the following:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Stride.Core;
using Stride.Core.Assets;
using Stride.Core.Assets.Compiler;
using Stride.Core.BuildEngine;
using Stride.Core.Serialization;
using Stride.Core.Serialization.Contents;
using Stride.Engine;
using Stride.Engine.Design;
```

```
namespace YOUR_GAME_NAMESPACE;

/// <summary>
/// Runtime representation of the asset, this is the actual class you would use in your
scripts
/// </summary>
[DataContract]
[ContentSerializer(typeof(DataContentSerializerWithReuse<YOUR_CLASS>))]
// The ReferenceSerializer attribute is required when specifying a DataSerializerGlobal with
ReferenceSerializer<>,
// The two of them specifies to the serializers that this class is a runtime Asset
[ReferenceSerializer, DataSerializerGlobal(typeof(ReferenceSerializer<YOUR_CLASS>), Profile
= "Content")]
// The line below ensures that the asset's reference is re-used instead of cloned for all
new instances when instantiating prefabs
[DataSerializerGlobal(typeof(CloneSerializer<YOUR_CLASS>), Profile = "Clone")]
public class YOUR_CLASS
{
        // Replace this with whatever you would want this asset to hold at runtime
    public List<Prefab> PrefabCollection { get; set; } = new();
}

/// <summary>
/// Design time and file representation of <see cref="YOUR_CLASS"/>, you can add content
here that won't be included in the build
/// </summary>
[AssetDescription(FileExtension, AllowArchetype = false)]
[AssetContentType(typeof(YOUR_CLASS))]
[AssetFormatVersion(nameof(YOUR_GAME_NAMESPACE), CurrentVersion, "1.0.0.0")]
public sealed class YOUR_CLASS_ASSET : Asset
{
    private const string CurrentVersion = "1.0.0.0";
    public const string FileExtension = ".blks";

        // Replace this with whatever you would want this asset to have while inside the
gamestudio
    public List<Prefab> PrefabCollection { get; set; } = new();
}

/// <summary> Compiler which transforms your <see cref="YOUR_CLASS_ASSET"/> into <see
cref="YOUR_CLASS"/> when building your game </summary>
[AssetCompiler(typeof(YOUR_CLASS_ASSET), typeof(AssetCompilationContext))]
public sealed class YOUR_CLASS_COMPILER : AssetCompilerBase
{
    protected override void Prepare(AssetCompilerContext context, AssetItem assetItem,
```

```
string targetUrlInStorage, AssetCompilerResult result)
    {
        var asset = (YOUR_CLASS_ASSET)assetItem.Asset;

        // you can have many build steps, each one is running an AssetCommand
        result.BuildSteps = new AssetBuildStep(assetItem);
        result.BuildSteps.Add(new DESIGN_TO_RUNTIME_COMMAND(targetUrlInStorage, asset,
assetItem.Package));
    }

    public override IEnumerable<ObjectUrl> GetInputFiles(AssetItem assetItem)
    {
        // Yield assets that must be built before this one is built,
        // this is for cases were you need to read from a compiled version of an asset to
build this one.
        // Note that creating cyclic references through this method will cause a deadlock
when building
        // (e.g.: A is input of B while B is input of A)

        // below only for reference purposes, useless in this context
        var asset = (YOUR_CLASS_ASSET)assetItem.Asset;
        foreach (var block in asset.PrefabCollection)
        {
            var url = AttachedReferenceManager.GetUrl(block);

            if (!string.IsNullOrEmpty(url))
            {
                yield return new ObjectUrl(UrlType.Content, url);
            }
        }
    }

    /// <summary>
    /// An <see cref="AssetCommand"/> that converts design time asset into runtime asset.
    /// </summary>
    public class DESIGN_TO_RUNTIME_COMMAND(string url, YOUR_CLASS_ASSET parameters,
IAssetFinder assetFinder)
        : AssetCommand<YOUR_CLASS_ASSET>(url, parameters, assetFinder)
    {
        protected override Task<ResultStatus> DoCommandOverride(ICommandContext
commandContext)
        {
            var assetManager = new
ContentManager(MicrothreadLocalDatabases.ProviderService);

            var runtimeObject = new YOUR_CLASS{ PrefabCollection =
```

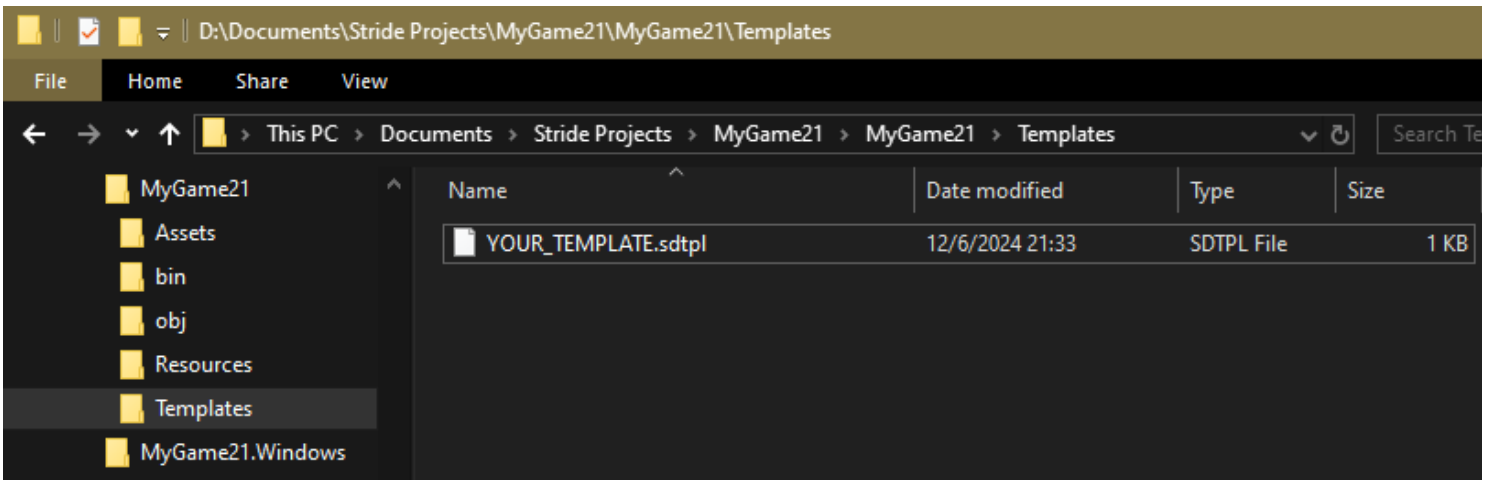
```
Parameters.PrefabCollection };
            assetManager.Save(Url, runtimeObject);

            commandContext.Logger.Info($"Saving {nameof(YOUR_CLASS)}:
{runtimeObject.PrefabCollection}");

            return Task.FromResult(ResultStatus.Successful);
        }
    }
}
```

> **WARNING**
>
> Every changes made to the runtime asset will break previously built asset databases, make sure to clean the solution or manually delete the build artifacts stride generates for assets (`YOUR_PROJECT.Windows/obj/stride` and `bin/db`) after changing the class to make sure the asset database is fully rebuilt on the next build.

This takes care of the support for this asset, you could create a `*.blks` file inside your `Assets` directory and fill in the content manually, but might as well do it through the editor ...

Adding a section for the Add asset menu inside the editor

Create a new directory named `Templates` within your Game's directory, this directory will be used to store your templates. Inside of that directory, create a new file named after your new asset with the `.sdtpl` extension.



Open the file and paste the following into it

```
!TemplateAssetFactory
Id: 21CC3354-9F0B-4D1F-8242-62D56454B27C
```

```
AssetTypeName: YOUR_CLASS_ASSET
Name: THE NAME IN THE EDITOR
Scope: Asset
Description: A DESCRIPTIVE DESCRIPTION OF YOUR ASSET
Group: WHERE THIS WOULD BE CLASSIFIED UNDER IN THE EDITOR
DefaultOutputName: THE DEFAULT FILE NAME
```

Edit the different fields appropriately,

- `Id` must be unique ! There are a couple of services online to generate one if you need to, search for `generate guid online`
- `AssetTypeName` must match the name of the class that inherits from `Asset` (the namespace can be omitted)

Now you have to edit your `*.sdpkg` to include this new template, to do so you just have to add the following lines below your `TemplateFolders:`

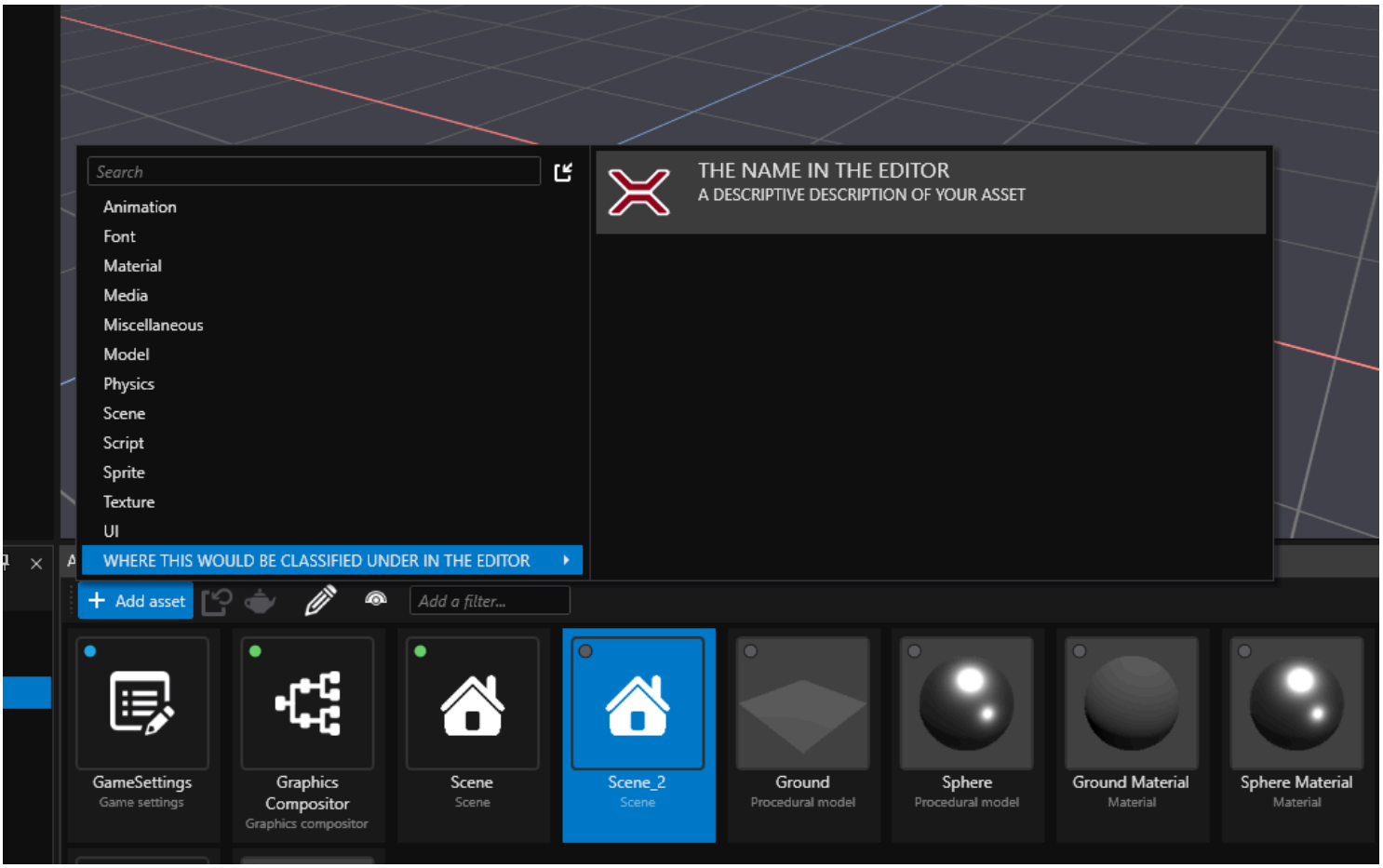
```
TemplateFolders:
    - Path: !dir Templates
      Group: Assets
      Files:
        - !file Templates/YOUR_TEMPLATE.sdtpl
```

Here's how it looks like when included into a default game `*.sdpkg`:

```
!Package
SerializedVersion: {Assets: 3.1.0.0}
Meta:
    Name: MyGame21
    Version: 1.0.0
    Authors: []
    Owners: []
    Dependencies: null
AssetFolders:
    -   Path: !dir Assets
    -   Path: !dir Effects
ResourceFolders:
    - !dir Resources
OutputGroupDirectories: {}
ExplicitFolders: []
Bundles: []
TemplateFolders:
    - Path: !dir Templates
      Group: Assets
```

```
      Files:
        - !file Templates/YOUR_TEMPLATE.sdtpl
RootAssets: []
```

And you're finally done, have fun !



See also

- Best Practice

Best Practices

Tips to build a robust and maintainable codebase for your Stride project.

Planning Ahead and Technical Debt

Before writing major systems, consider how it would fit with the rest of the existing systems. Many systems become interdependent once implemented, for example:

- How should this new system behave when saving and reloading a game session.
- When the game is paused.
- When loading in a new scene.
- When changing game mode.
- Which player input should it read.
- How should it synchronize across clients in a multiplayer context.
- etc.

With that said, one will stumble on issues along the way that will require a change in design, a rewrite of those systems, for example: an uninteresting or unfeasible feature, a re-evaluation of your game's scope, shortcuts made to get a feature out the door sooner, an ill-defined initial design, and so on.

The design of a game is not set in stone; it is refined as Macklin and Sharp, Game Designers at Parsons School of Design, observed:

> *Designing games is challenging [...] Compounding all this is the fact that game designers can't really "see" their designs until they are played, and their games can't be played until they are made. This is where the iterative design process comes in handy.*
>
> **Macklin, C., & Sharp, J.** (2016). Games, design and play: A detailed approach to iterative game design. Addison-Wesley Professional.

You might feel like leaving those rewrites on the back burner as they don't contribute directly to the game, but doing so will cause you to slowly write yourself into a corner, wasting time stacking quick patches around problematic systems to keep them going, accruing bugs and performance issues instead of dealing with the problem head-on.

Frederick Brooks, one of the most influential software designers, on planning and rewriting software:

> *[...] is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers. Seen this way, the answer is much clearer. Delivering that throwaway to customers buys time, but it*

> *does so only at the cost of agony for the user, distraction for the builders while they do the redesign, and a bad reputation for the product that the best redesign will find hard to live down.*
> *Hence plan to throw one away; you will, anyhow.*
>
> **Frederick P. Brooks Jr.** (1975). The Mythical Man-Month. Addison-Wesley.

Premature Generalization

Related to the above, we may fall into the extreme opposite, over designing systems. Misleading ourselves into adding features covering hypothetical cases, expanding it to accommodate future games, or encapsulating it to share it for others to use.

Here's Richard Fabian, Senior Programmer on Max Payne 3, on systems re-use:

> *It is regarded as one of the holy grails of development to be able to consistently reduce development overhead by reusing old code. [...] The unfortunate truth is any interesting new features you want to add will probably be incompatible with your old code and old way of laying out your data, and you will need to either rewrite the old code to allow for the new feature, or rewrite the old code to allow for the new data layout.*
>
> **Richard Fabian**. Data-Oriented Design 11.7 Reusable generic code.

Continuing on the same subject, here's a survey by William Agresti:

> *One survey result had implications for a role that reuse may play in large software projects exceeding their budgets: 40% of developers reported that in 25% or more of the cases, they ended up extensively modifying or completely rewriting a module that had planned to be reused.*
>
> **Agresti, W**. (2011) Software Reuse: Developers' Experiences and Perceptions. Journal of Software Engineering and Applications, 4, 48-58. doi: 10.4236/jsea.2011.41006.

When starting a new project, it's also likely that we would prefer rewriting those systems. Our experience on previous projects may provide us with insight to design them in a more concise, maintainable, or robust way.

There definitely are systems that could very well be re-used, but we're not building middleware. We should ask ourselves if now is the right time to work on generalizing the system, or whether our game should take precedence.

AI assistants and LLMs

Large language models and AI assistants have become ubiquitous in most programmers' toolboxes. However, like all tools, they have their drawbacks. Consider the well-known idiom:

> *If all you have is a hammer, everything looks like a nail.*

As this phrase implies, we should carefully consider when to apply a given tool, lest we become overzealous or over-reliant on them, like the following study infers:

> *AI tools were perceived as helpful for understanding code and increasing confidence, particularly during initial development. However, students reported difficulties transferring knowledge to unaided tasks, revealing possible overreliance and gaps in conceptual understanding.*
>
> **Rojas-Galeano et al**. (2025). Between Tool and Trouble: Student Attitudes Toward AI in Programming Education. doi: 10.48550/arXiv.2508.05999.

Relying on LLMs for design or programming tasks will stunt our growth in those areas, those are skills we must maintain and improve through continuous practice. The following excerpt conveys how this reliance affects us:

> *[...] over-reliance on AI tools has led to a decline in confidence and skill retention, as students became less attentive to grammar rules and idea formulation, relying instead on AI to complete those tasks.*
>
> **Budiyono, Herman**. (2025). Exploring long-term impact of AI writing tools on independent writing skills: A case study of Indonesian language education students. International Journal of Information and Education Technology, 15(5), 1003-1013. 10.18178/ijiet.2025.15.5.2306.

We must use LLMs to teach us new concepts or skills, but troubleshooting complex issues, designing groundbreaking systems, and creating innovative solutions to our design problems, that is our own prerogative. Given enough experience, those are areas where we perform better than an LLM could.

Figuring Out Your System's Lifetime

What is the scope of the system you're writing:

- Should the same instance be used throughout the entire lifetime of the application?
- Only while playing a game session?
- Only for the duration of the currently loaded game session?

- Within a single map?
- For a specific game mode?
- ...

This will set some expectation as to where the system you're building should reside, how it interacts with other systems, and when it should be started and disposed ...

Some entry and exit points to manage your systems' lifetime:

- On demand by getting and setting it in the Services
- System as a singleton script with the `Start()` and `Cancel()`
- Through your game's `BeginRun()` and `EndRun()`
- As a component processor when associated component types are added to the scene

Statics, Singletons and Other Globals

We strongly advise you to make sure that the entirety of your game's state is implemented as instance properties on components inside the root scene's hierarchy. Avoid static properties and static objects.

This is essential to reduce bugs that come in when implementing systems that manage the game's state, like the saving system, or the multiplayer layer. Those systems can then be implemented with the expectation that everything they may care about is within the root scene, they can replace this root scene and expect the game state to be completely reset, they can serialize or monitor those entities for changes, etc. without the risk of your game's state leaking between play sessions and creating issues that are really hard to reproduce.

Some systems may not make sense as part of the scene when:

- The functionality and variables saved by such systems persist for the duration of the program, or across all sessions
  - Saving, loading systems, meta-progression trackers or achievements
- The system is read-only
  - Multiplayer server browser or matchmaking back-end. Once connected to a session it's a different story though, now you must hold a bunch of states that are only valid to this session, it should not leak to the rest of the program, and so is best left as a component on an entity in the scene.

Those restrictions do not prevent you from using the singleton pattern, you can use the `ServiceRegistry` which can be accessed from any `ScriptComponent`.

Show Example

Implement Custom Assets

Some of the systems you will build make far more sense as assets rather than entities, consider making them an asset when any of the following is true:

- It survives between multiple scenes
- It is read only
- It is not part of the definition of an entity, doesn't exist within your game world
- It should be editable within the editor, by a designer

Examples of such are:

- Player Input Configuration, defining actions in the source, assigning buttons in editor, saving and loading to disk when initializing the game
- Balance settings, tweaking constants and formulas from the editor to improve iteration when testing your game
- Mission/quest, referencing quests inside of components to unlock spots in game when they are completed, giving the ability for your designer to set those up
- Loot tables, having a list of `UrlReference<Prefab>` with a probability of drop to easily re-use across multiple mobs
- As an all-purpose robust 'key' or 'identifier' type, see this section

Do not Mutate Assets

To that point, make sure to only mutate assets when it makes sense to do so. Remember that a single asset may be referenced by hundreds of components and systems, those may not expect them to change at runtime. Adhering strictly to this idea also ensures that your game's state does not leak through them when loading a new session, game mode, or whatever else. For example, let's say you have an Axe Asset which has a list of modifier, you save the game, progress for a bit then add a modifier, but end up reloading ot the previous save, the modifier will carry over to that previous game state.

Scene Quirks

The default scene the game spawns for you is the instance stored in the content manager, when running the game you mutate that very instance, meaning that if you want to retrieve the scene in its initial state, you must force the content manager to unload it, and then reload it. This makes it a bit counterintuitive when you just want to re-spawn the current scene to roll back your changes.

Strings as Keys or Identifiers

This is a very popular anti-pattern, strings that are used as keys or identifiers shows up all over the place, here's a short example describing such a usage: The quest you're implementing requires the player to gather 10 bear arses, your check for that is to loop through the list of items the player has and check that the item's name match `bear arses`.

Here's a couple of reasons why this is a bad idea:

- Hard to maintain; if your item ends up changing names because bears are banned in Freedonia, your checks will silently fail
- Fragile; your string isn't checked against anything, typos - `bear ass` wouldn't work, careful with leading or trailing whitespaces ...
- Obtuse; designers not aware of how your system work may not understand what they should input there
- Non-explicit uniqueness requirement, if your system expects those to be unique, i.e.: you are looking for a very specific "bear arse"; but you could have multiple different items all named `bear arse`
- Hard to keep track of; strings are too ubiquitous, hard to quickly retrieve all instances/usages of those to build a database for different purposes, like translations, validations ...

All of this will inevitably lead to bugs, or additional work to avoid them - time you could definitely use to take care of other, more fun parts of your game.

There are a couple ways to avoid this, one of the more robust ones is to rely on assets themselves; see custom assets.

```
public class Item { }

// Now in your component ...
public Item ItemToCheck; // You would assign this reference in the editor
public int AmountRequired = 10;

public bool HasTheItem()
{
    if (Inventory.TryGetAmount(ItemToCheck, out var amount) && amount >= AmountRequired)
        return true;
    return false;
}
```

- Robust; you can change its name and its path, as long as you do not delete it or change its internal id, all components referencing it will keep that reference.
- Easy to use and understand; If a component requires a specific asset, you don't have an infinite amount of possibilities, you can either set it to an existing one or create a new one. It's far more foolproof too now that typos are out of the equation.
- Easy to keep track of; each type has a unique extension which you can search for in your file explorer, they exist on disk, and so can be organized into the same directory.
    - The editor's reference window lists all assets that use the selection, this greatly helps when you need to swap the identifier for another or remove it altogether, just go through all the assets that refer to it.

    - you won't need to keep a document going over each identifier you might have in game, one just has to look at the directory were they are stored in the editor.
- Easy to extend; your identifier can now be more than just that, you can attach properties to it, perhaps a description to keep more information about this key.

Avoid Patterns with High Levels of Indirection

Particularly when mutating the game state, Event Keys and async methods carry a lot of implicit complexity as they may not complete when signaled/called. When the async resumed/the event key is received, the game may not be in a state where the logic you run is still valid. Some entities might have been removed from the scene, the inventory might no longer hold the item, the player character may be incapacitated ...

This quirk also means that their execution are not part of their callers' stack, making debugging issues with them far harder to figure out.

Their lifetime is also far harder to reason about as EventKeys will carry the signal even if the scene was replaced in the meantime, while async will continue running when running outside your AsyncScript's `Execute()`.

Alternatives to EventKeys:

- C# events, although this requires the receivers to have a direct reference to the sender
- Components with an interface bound to a Flexible processors. Add the processor to the service registry, call some method which goes through and call each one of the components implementing the interface of that processor

Alternatives to async:

- Restructure your async into a synchronous one ... obviously!
- If you can't avoid using async:
    - Don't touch the game state, just take some input, spit out an output that gets read by a `SyncScript`
    - Ensure you always leave the game state in a valid state before awaiting, and after awaiting check that it is still in a state were continuing the async method makes sense. I.e.: are we suddenly back on the main menu?!

You may notice that those two last ones could require a ton of additional logic to support properly, this is an indication that your logic should be rethought - you're writing yourself into a corner.

Avoid Writing Shortcut Extension Methods

This is specifically referring to methods of this kind:

```
static Entity GetFirstChild(this Entity Entity) => Entity.Transform.Children[0].Entity;
// Or
static void AddAsFirstChild(this Entity Entity, Entity entity) =>
Entity.Transform.Children.Insert(0, entity);
```

It's a double-edged sword:

- You are reducing the skill floor required for users not accustomed to the API, but you're also hindering their growth as they now rely on your shortcut instead of discovering the API for themselves, making them aware of concepts and objects neighboring that one, giving them a clearer view of how all the objects fit together. What if they need to access all children, from this extension method they would not be aware that the transform component stores them, that they could access it directly for that.
- Make sure that accessing what you are hiding is never error-prone, even more so if the name of the method does not make that obvious. You may be reducing the time wasted from typing, but you could very well increase the time you would take to debug it when it does create issues.
- It may very well be a slippery slope to introduce even more shortcuts to other properties or methods of the object you are presenting, how about creating an extension for the second child of the entity, the third ...
- Polluting intellisense; in most cases this is a non-issue, but collection types are a prime example of this. Discoverability for extension methods through intellisense is nigh-on-impossible, there are just far too many extension methods introduced by linq.
- It might imply to the user that your shortcut is somehow different from the source.

Entity and Components' Lifetime

One unexpected quirk of Stride is that components and entities are expected to survive across any number of removal and re-insertion into the scene. Those objects are never truly 'destroyed', they are treated like any other c# object, they either exist or are out of scope.

Make sure that your components adhere to this rule by rolling back any effects introduced in `Start()` through `Cancel()` This quirk provides a couple of nice benefits, a major one is that you can temporarily remove components, entities and even scenes from your game and re-introduce them whenever you need without any loss of data or complex serialization steps.

This also means that you should avoid writing any custom 'destroy' function to ensure that any part of the engine at any time can simply remove the entity from the scene and rely on your implementation of `Cancel()` to take care of anything that should occur when 'destroyed'.

Usage of Get

When using `Get<MyComponent>` ask yourself whether the function would fail to operate if that call were to return null, if that is the case, then your function is dependent on that component existing on that entity. This is a hard dependency, you should do everything you can to notify the rest of your codebase and designers using the editor that in this component is a requirement to avoid wasting time debugging issues related to it.

There are a couple of ways to do so, here we simply add the component directly as a parameter to the function:

```
// From
public void MyFunction(Entity entity)
{
    entity.Get<MyComponent>().DoSomething();
}
// To
public void MyFunction(MyComponent component)
{
    component.DoSomething();
}
```

And here we add this component as a property to set in the editor:

```
// From
public void MyFunction()
{
    Entity.Get<MyComponent>().DoSomething();
}

// To
// The 'required' keyword will generate a warning on build when the value is not set in
the editor
public required MyComponent MyRequiredComponent { get; set; }

public void MyFunction()
{
    MyRequiredComponent.DoSomething();
}
```

A trap you may fall into after reading this is to write defensively, checking if it is null and returning in such cases even if the rest of the logic expects some sort of change. This will more often than not force you to write far more boilerplate logic than you would have if you ensured you had a valid one in the first place.

One thing you may also consider is whether to simply merge the dependant object together, if either one of the objects are used only for the other's purpose, it may make far more sense to simply merge them instead of having two different components.

See also

- Scheduling and priorities
- Flexible processors
- Custom Assets

Sprites

2D applications are made of **sprites**.

The most efficient way to render sprites is to add them all to a **sprite sheet**, a single image. You can then define regions of the sprite sheet as different sprites in Game Studio's Sprite Editor. After you define sprites, you can add them to entities using sprite components and render them with scripts.

In this section

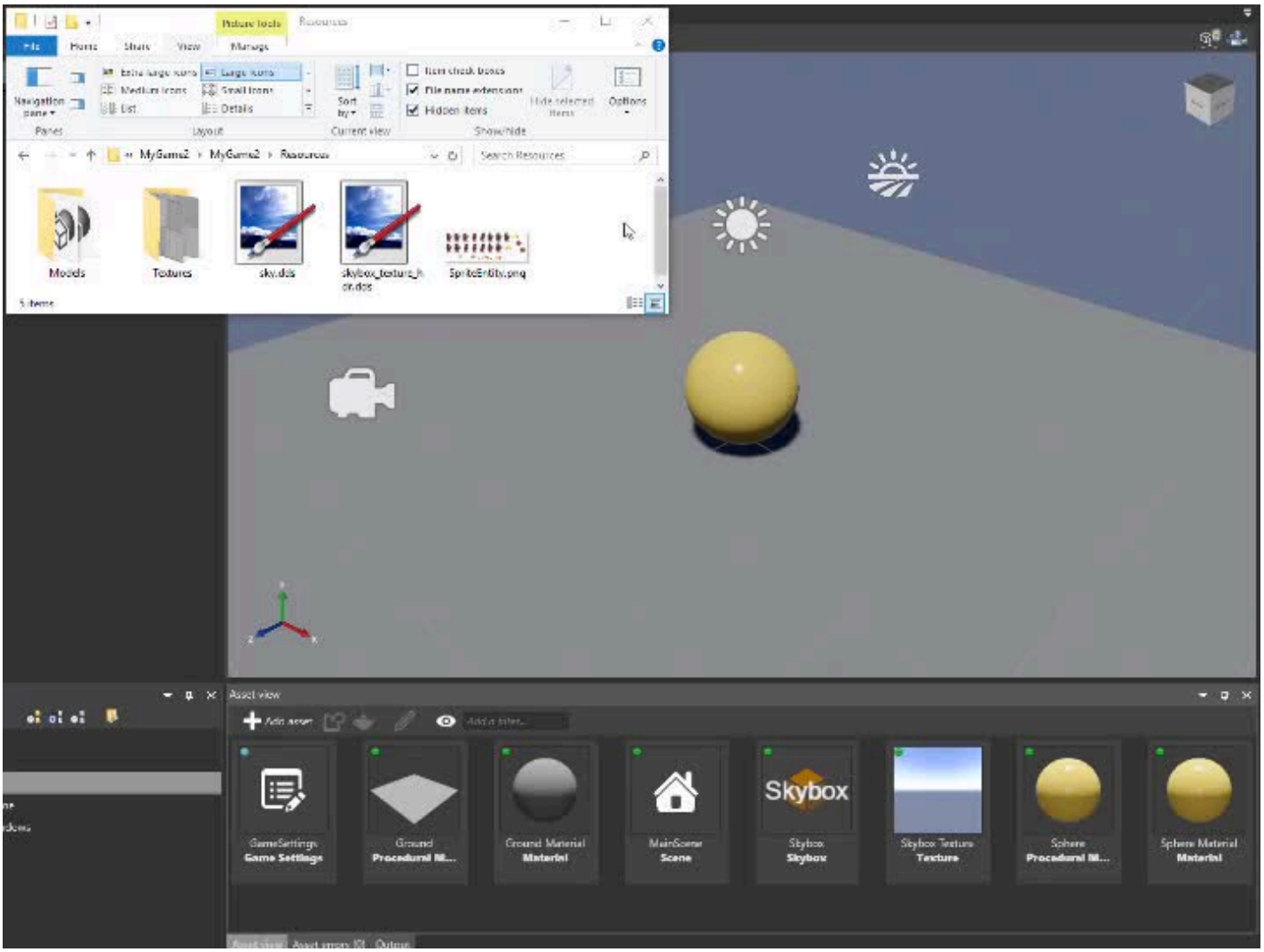
- Import sprite sheets
- Edit sprites
  - Set sprite borders
- Use sprites

Import sprite sheets

Beginner Designer

You can import sprite sheets (image files containing sprites) just like any other kind of asset.

1. Drag the sprite sheet file from Explorer to the Game Studio **Asset View**.



   Alternatively, in the **Asset View**, click **Add asset**.

2. Choose a preset for the sprite sheet.

   If you want to use the sprite sheet for UI elements such as menu buttons, select **Sprite sheet - UI sprites**. This lets you set borders for the sprite (see Set sprite borders). Otherwise, select **Sprite sheet - 2D sprites**.

> **NOTE**
>
> You can change this any time. For more information, see Edit sprites.

Game Studio adds a sprite sheet asset.

See also

- Edit sprites
- Use sprites
- Assets

Edit sprites

Beginner Designer

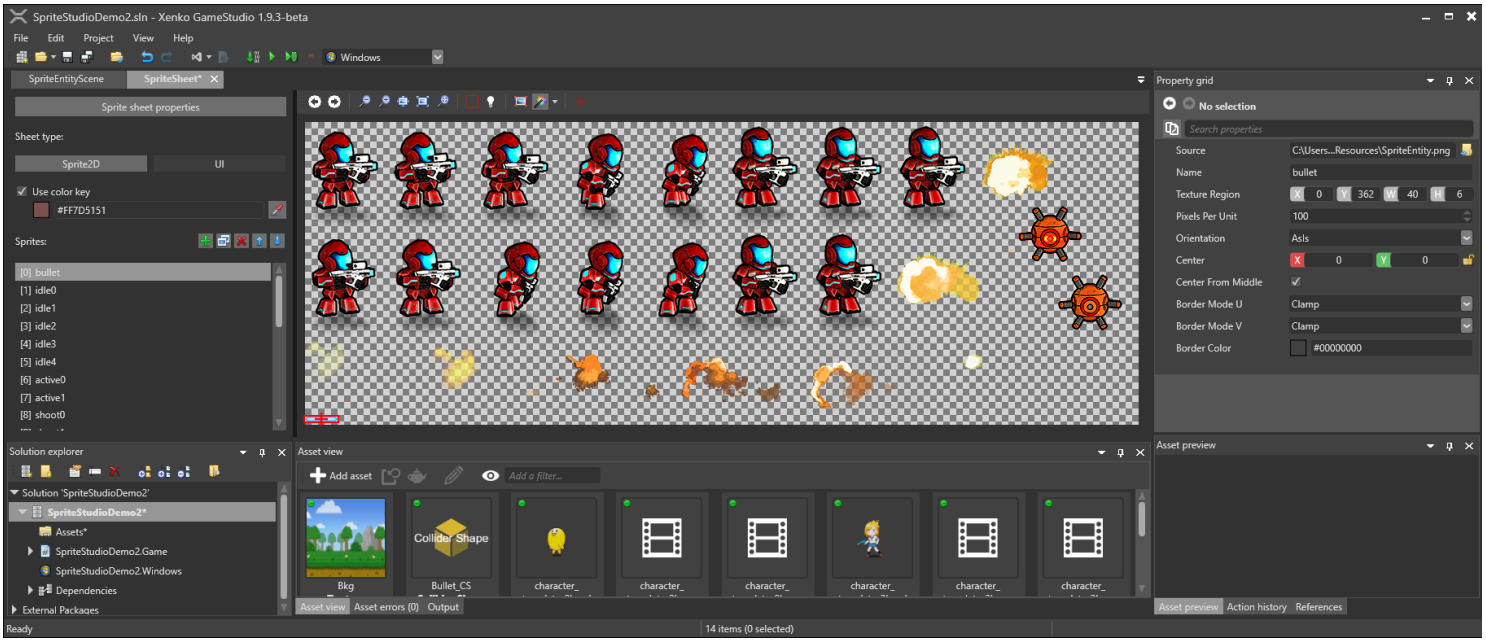
After you import a sprite sheet, you can use the dedicated **Sprite Editor** to select sprites within the sprite sheet.

You can also edit sprite properties in the **Property Grid** like any other asset.

Open the Sprite Editor

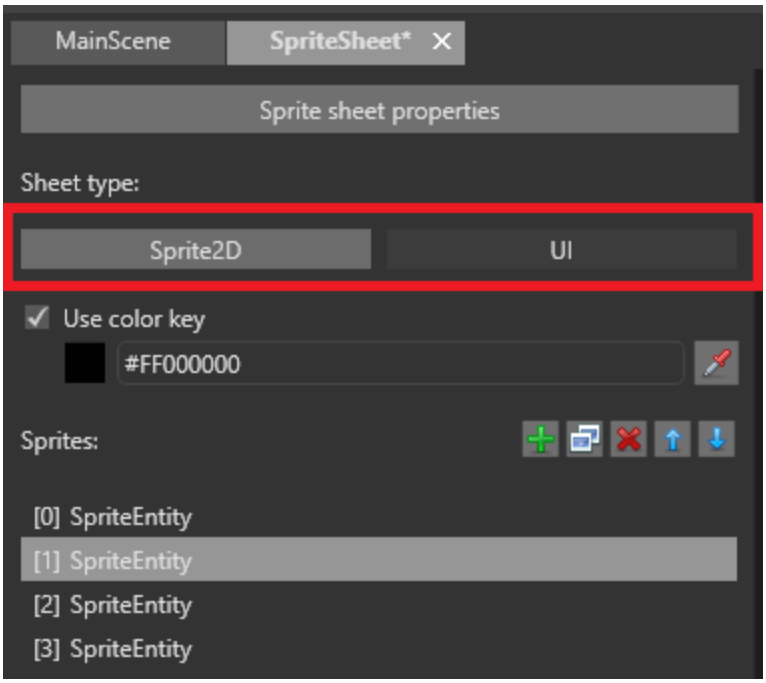
To open the Sprite Editor, in the **Asset View**, double-click the sprite sheet asset.

The sprite sheet opens in the Sprite Editor.



Set sprite sheet type

You can set whether the sprite sheet contains gameplay sprites (**Sprite2D**) or UI sprites (**UI**). This has no effect on how the sprite is rendered at runtime, but lets you set slightly different properties, described under **Sprite properties** below. You can change the sprite sheet type any time.



Add a sprite

1. Click the **Add empty sprite** button.

   Game Studio adds a empty sprite to the list.

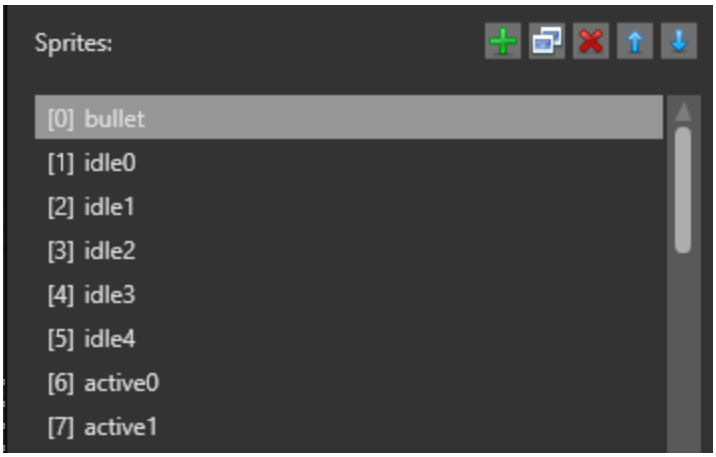
2. In the **Property Grid** on the right, in the **Source** field, specify the sprite sheet that contains the sprite.

TIP

If you want to create a new sprite from the same sprite sheet as other sprites in the list, it's often faster to duplicate an existing sprite. To duplicate a sprite, select it and click **Duplicate selected sprites** or press **Ctrl + D**.

Sprite list

The Sprite Editor lists the sprites in your project on the left. Each sprite has an index number; the first has the index *[0]*, second has index *[1]*, and so on. You can use these indexes in your scripts (see Use sprites).

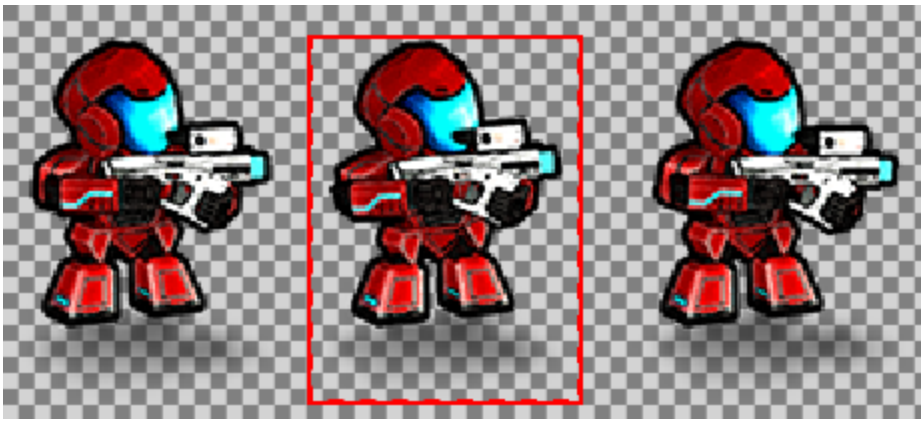


To change the order (and index number) of sprites, use the (**Move selected sprite up/down**) buttons. For example, if you move *[1] Sprite* down, it becomes *[2] Sprite*.

To rename a sprite in the list, double-click it and type a new name.

Set the texture region

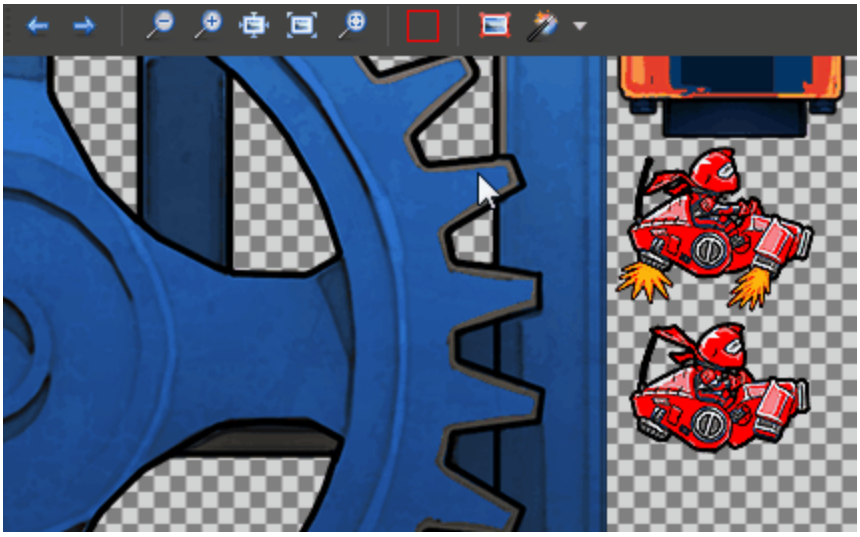
You create sprites by defining rectangular **texture regions** in the sprite sheet.



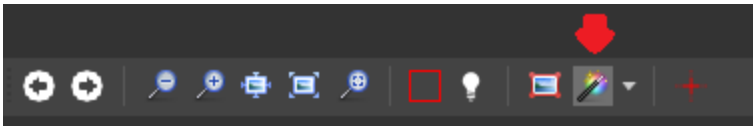
There are three ways to do this: by using the Magic Wand tool, by setting the edges of the region manually, or by specifying the pixel coordinates in the sprite properties.

Use the Magic Wand

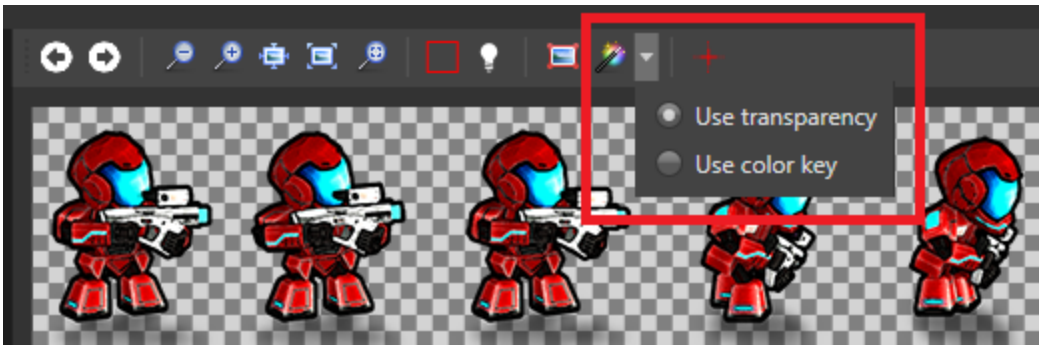
The **Magic Wand** selects the texture region around a sprite automatically. This is usually the fastest way to select sprites.



To select or deselect the Magic Wand, click the icon in the Sprite Editor toolbar, or press the **M** key.



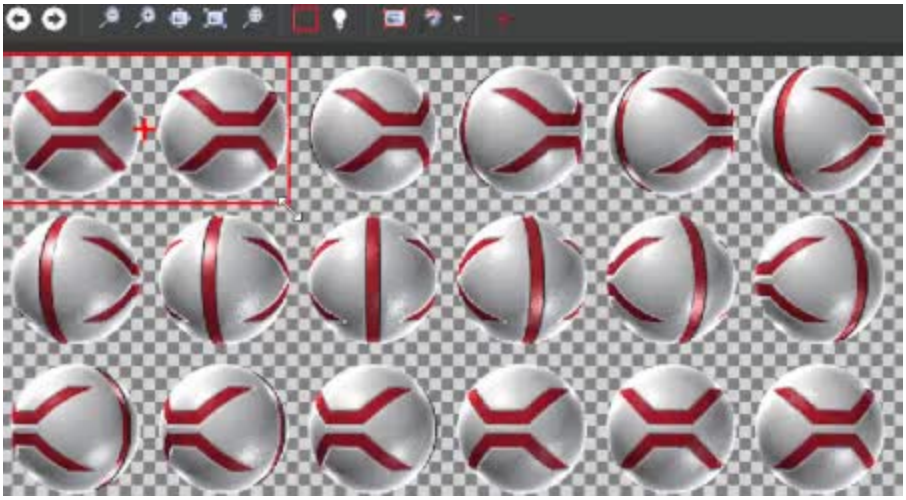
To choose how the Magic Wand identifies texture regions, use the **drop-down list** in the toolbar.



- **Transparency**: The Magic Wand treats the edges of the non-transparent regions as the edges of the texture region. For example, if the sprite is surrounded by transparent space, it sets the texture region at the edge of the transparent space.
- **Color key**: The Magic Wand identifies texture regions using the color set under **Color key** in the Sprite Editor. For example, if the sprite is surrounded by absolute black (#FF000000), and you set absolute black as the color key, the Magic Wand sets the texture region at the edge of the absolute black space.

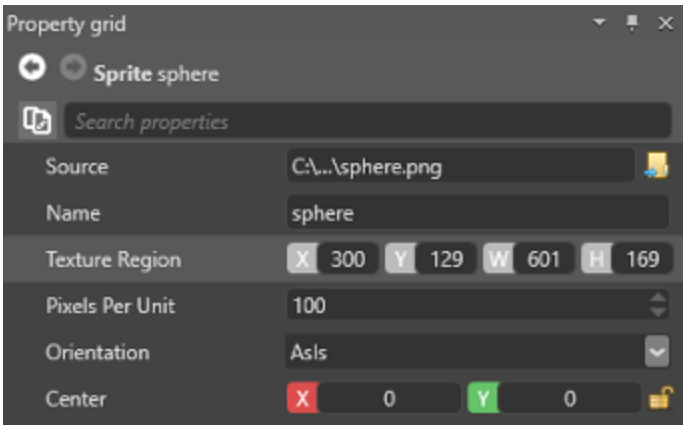
Set texture region manually

You can drag the edges of the texture region and reposition the region manually.



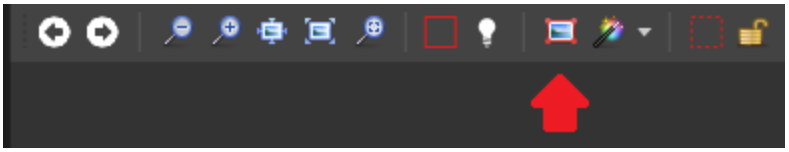
Set the texture region in the Property Grid

You can define the pixel coordinates of the texture region in **Property Grid** under **Texture Region**. X is the left edge, Y is the top, Z is the right, and W is the bottom.



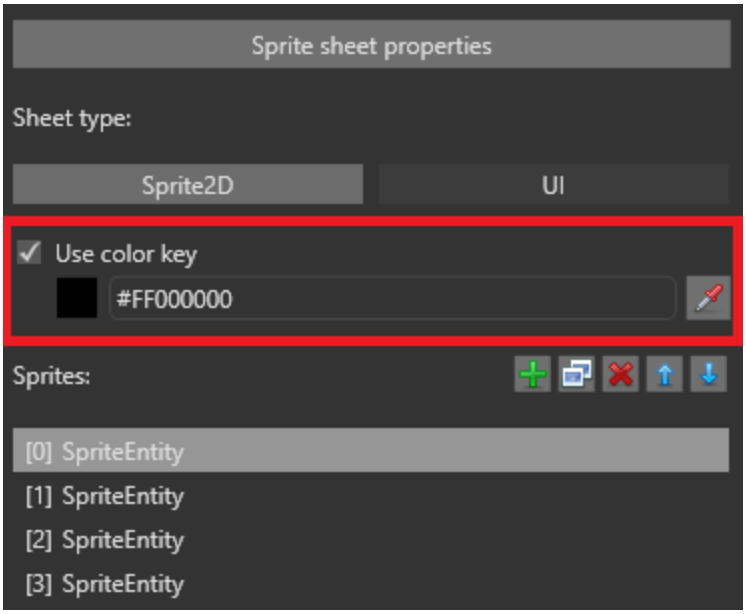
Use entire sprite sheet for the sprite

If you want to use the entire sprite sheet image for the sprite, you can do this quickly by clicking **Use whole image for this sprite** in the toolbar. This is useful when you have only one sprite in a sprite sheet.

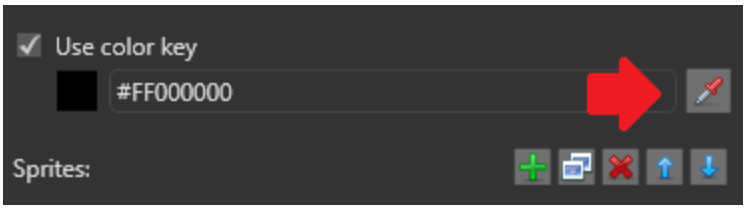


Set transparency

By default, Stride treats transparent areas of the sprite sheet as transparent at runtime. Alternatively, you can set a key color as transparent. To do this, select **Use color key** and define a color. For example, if you set absolute black (#FF000000), areas of absolute black are transparent at runtime.

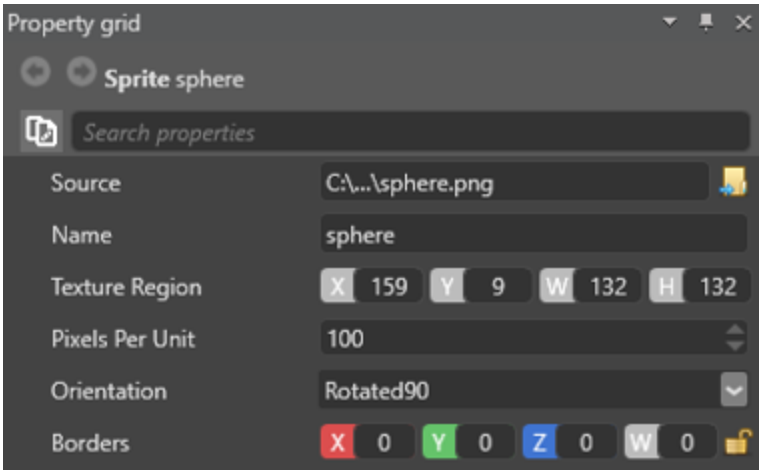


You can also use the **color picker** tool to select a color from the sprite sheet.



Sprite properties

You can set the properties of individual sprites in the **Property Grid**.



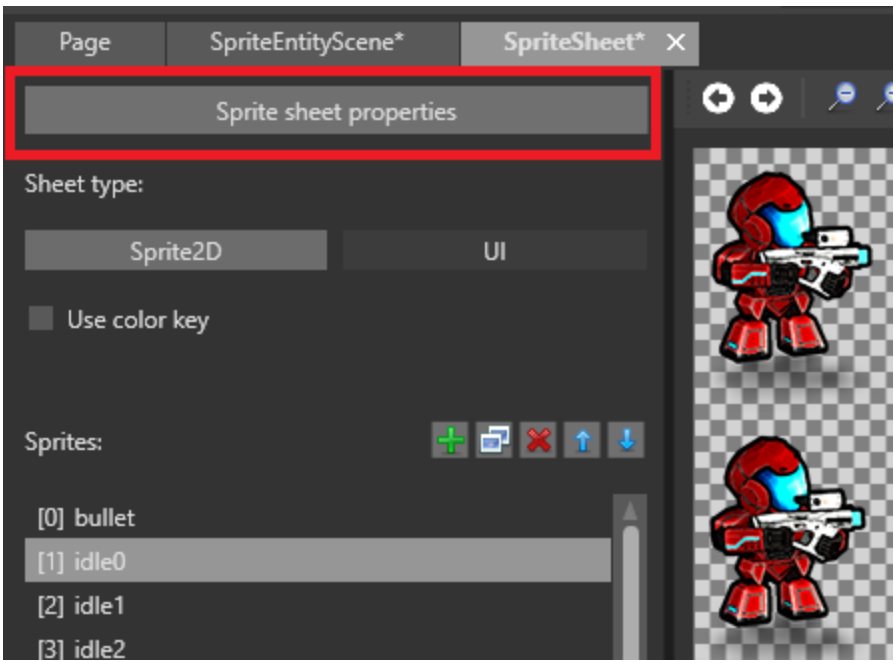
Property	Description
Source	The path to the sprite sheet
Name	The name of this sprite. You can also edit this by double-clicking a sprite in the sprite list in the Sprite Editor
Texture region	The region of the sprite sheet used for this sprite
Pixels per unit	The number of pixels representing a unit in the scene. The higher this number, the smaller the sprite is rendered in the scene
Orientation	If you select **Rotated90**, Stride rotates the sprite 90 degrees at runtime
Center	The position of the center of the sprite, in pixels. By default, the center is **0, 0**. Note: this property is only available if the sprite sheet type is set to **Sprite2D** in the Sprite Editor.
Center from middle	Have the value in the Center property represent the offset of the sprite center from the middle of the sprite. Note: this property is only available if the sprite sheet type is set to **Sprite2D** in the Sprite Editor.

Property	Description
Borders	The size in pixels of the sprite borders (areas that don't deform when stretched). X is the left border, Y is the top, Z is the right, and W is the bottom. For more information, see Set sprite borders. Note: this property is only available if the sprite sheet is set to **UI** on the left.

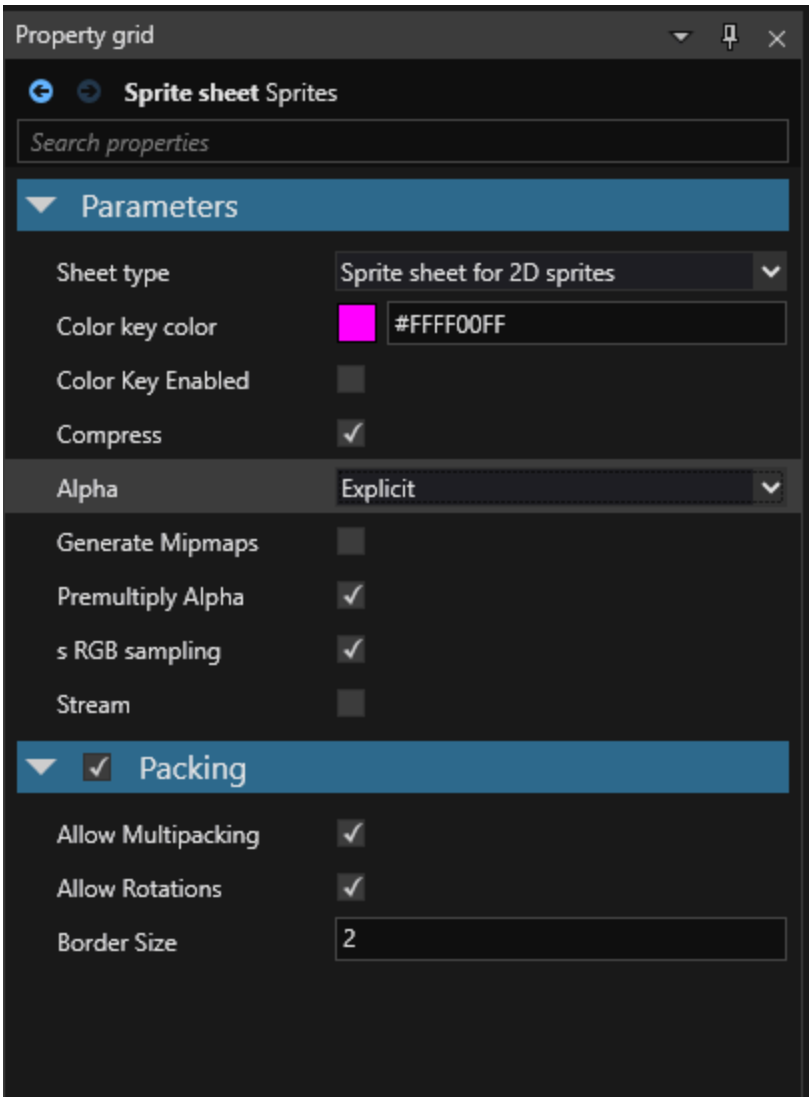
Sprite sheet properties

You can also set the properties for the entire sprite sheet asset. To access the properties:

- select the sprite sheet asset in the **Asset View** and set the properties in the **Property Grid**, or
- in the Sprite Editor, click **Sprite sheet properties**.



Many of the properties are the same as texture properties.



Property	**Description**
Sheet Type	Specify whether this sprite sheet is used for 2D sprites or UI elements. If you select **Sprite sheet for UI**, you can define sprite borders in the sprites.
Color Key Color	The color used for transparency at runtime. This is only applied if **Color Key Enabled** is selected below
Color Key Enabled	Use the color set in the **Color Key Color** property for transparency at runtime. If this isn't selected, the project uses transparent areas of the sprite sheet instead
Compress	Compress the texture to a format based on the target platform. The final texture size will be a multiple of 4.
ColorSpace	The color space for the sprites in the sprite sheet (Auto, Linear, or Gamma)
Alpha	The texture alpha format which all the sprites in the sprite sheet are converted to (None, Mask, Explicit, Interpolated, or Auto)

Property	Description
Generate Mipmaps	Generates mipmaps for all sprites in the sprite sheet
Premultiply Alpha	Premultiply all color components of the images by their alpha component
Allow Multipacking	Generate multiple atlas textures if the sprites can't fit into a single atlas
Allow rotations	Rotate sprites inside the sprite sheet to optimize space. This doesn't affect how sprites are displayed at runtime.
Border size	The size in pixels of the border around the sprites. This prevents side effects in the sprite sheet.

See also

- Import sprite sheets
- Set sprite borders
- Use sprites

Set sprite borders

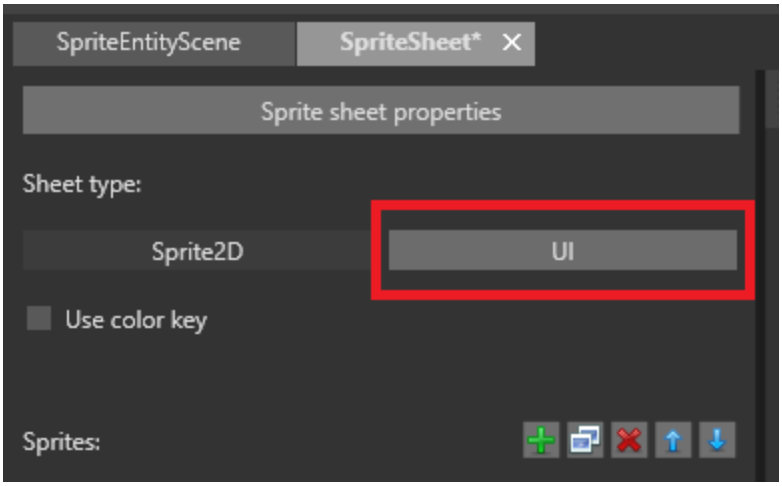
Beginner Designer

Sprite borders are areas that don't deform when you scale the sprite. These are often useful for sprites used for UI elements such as menu buttons. You can only set sprite borders for sprites set to the **UI** sheet type.

Original sprite	Scaled without borders	Scaled with borders
	Edges are deformed	Edges not deformed

Set sprite borders

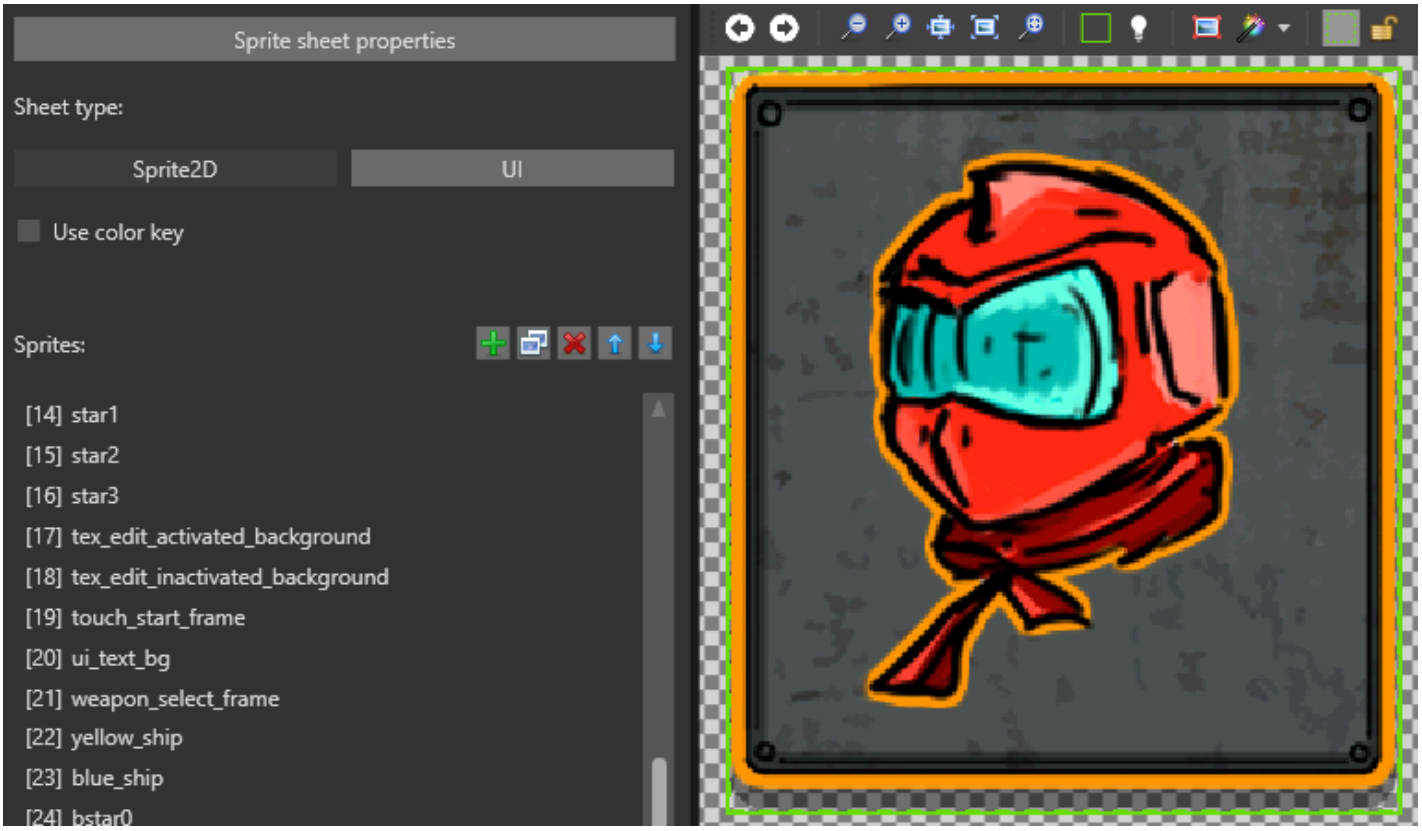
1. In the Sprite Editor, make sure the **sheet type** is set to **UI**.



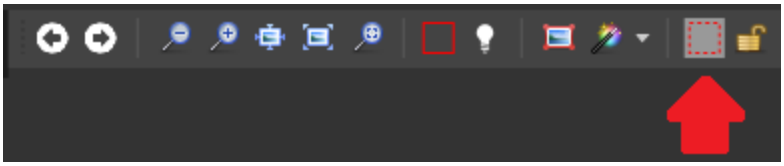
> **NOTE**
> This has no effect on how the sprite is rendered at runtime, but lets you set slightly different properties, including sprite borders.

2. From the **Sprites** list, select the sprite you want to add sprite borders to.

3. Make sure the texture region for the sprite is correct. For information about how to do this, see Edit sprites.



4. In the Sprite Editor toolbar, select **Sprite border resize** tool.



5. Drag the sprite borders into position.

NOTE

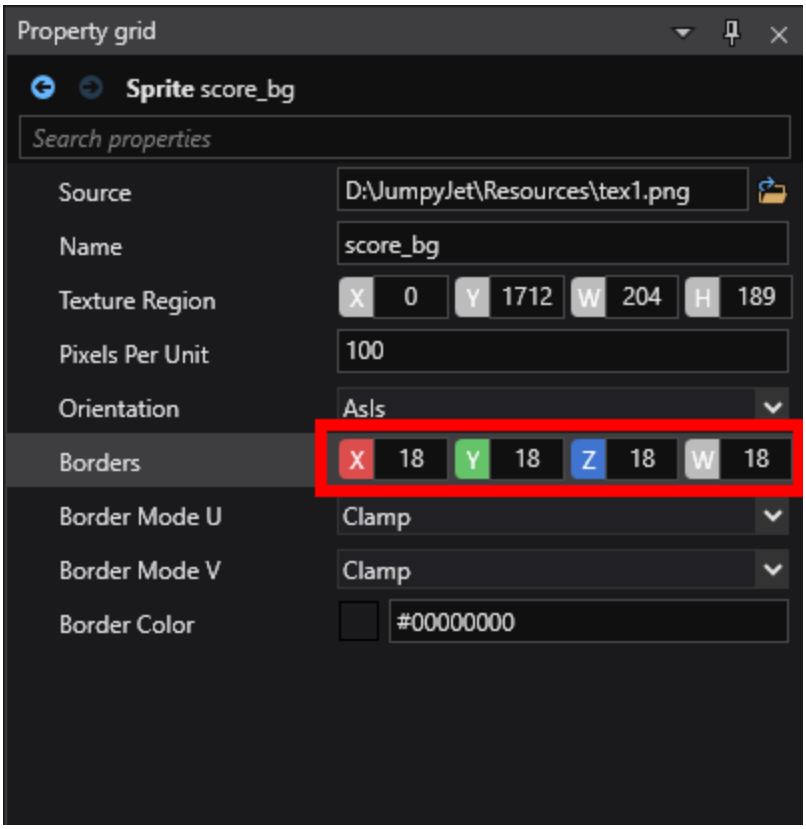
By default, the sprite borders match the sprite texture region.

TIP

You can zoom in and out using **Ctrl + mousewheel** to make precise selections.

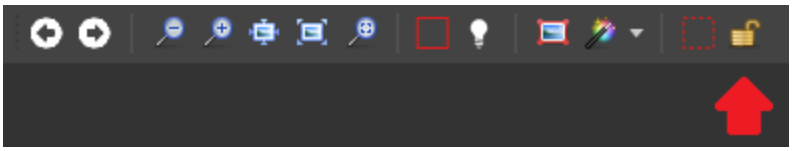
TIP

For fine-tune control over the sprite borders, adjusting one-by-one in the **Property Grid**



Lock the sprite borders

By default, sprite borders move as you resize the texture region. To stop this from happening, click **Lock the sprite borders** in the toolbar.



NOTE

Sprite borders always stay inside the texture region.

See also

- Import sprite sheets
- Edit sprites
- Use sprites

- UI

Use sprites

Intermediate Programmer

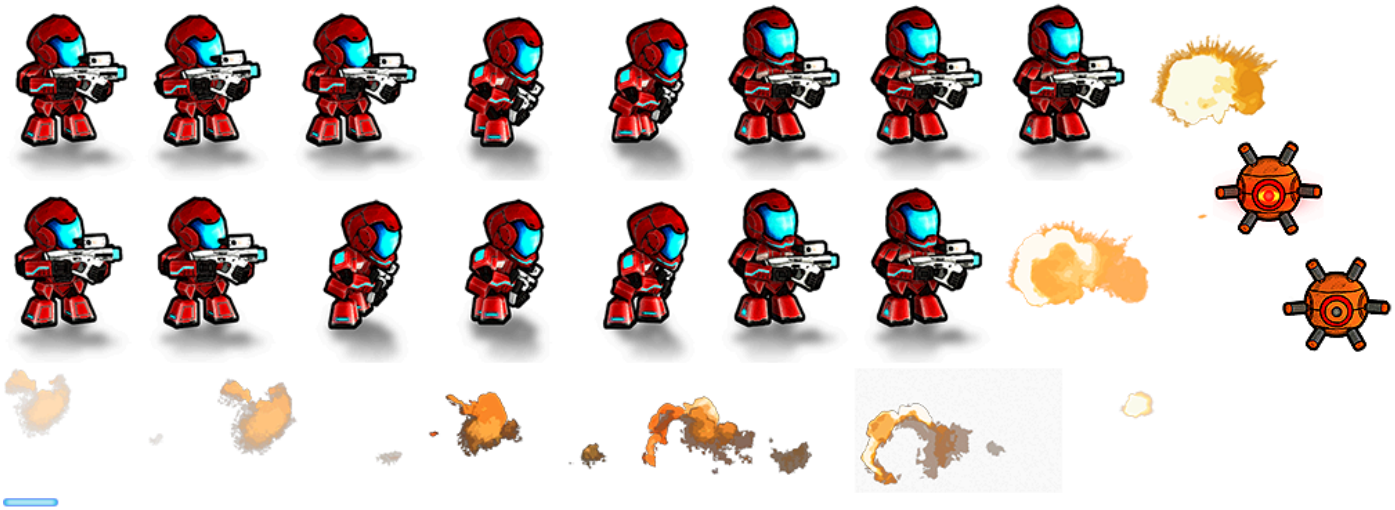
To add a sprite to a scene, add a **sprite component** to an entity. Afterwards, you can control the sprite with a script.

Add a sprite component

1. In the **Scene Editor**, select the entity you want to add a sprite to.

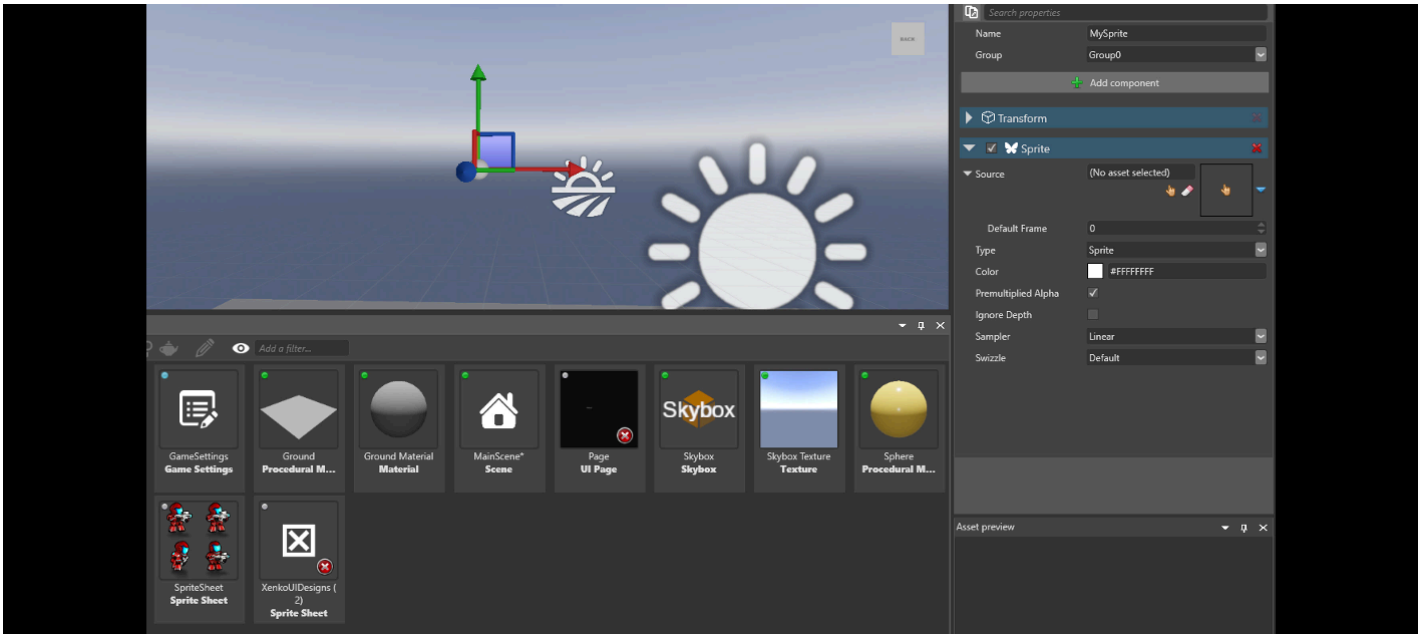
   > **TIP**
   > To create an entity, right-click the scene or Entity Tree and select **Empty entity**.

2. In the **Property Grid**, click **Add component** and select **Sprite**.

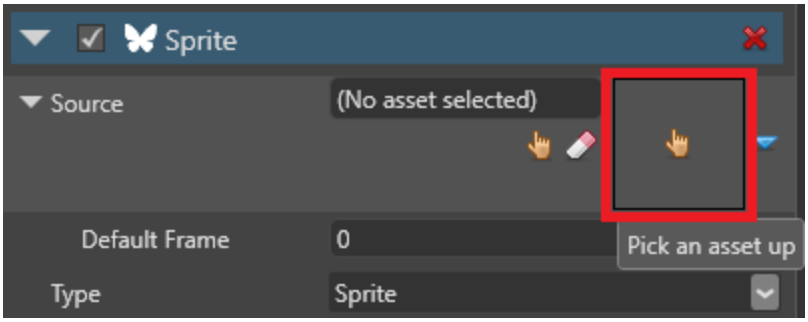


   Game Studio adds a Sprite component to the entity.

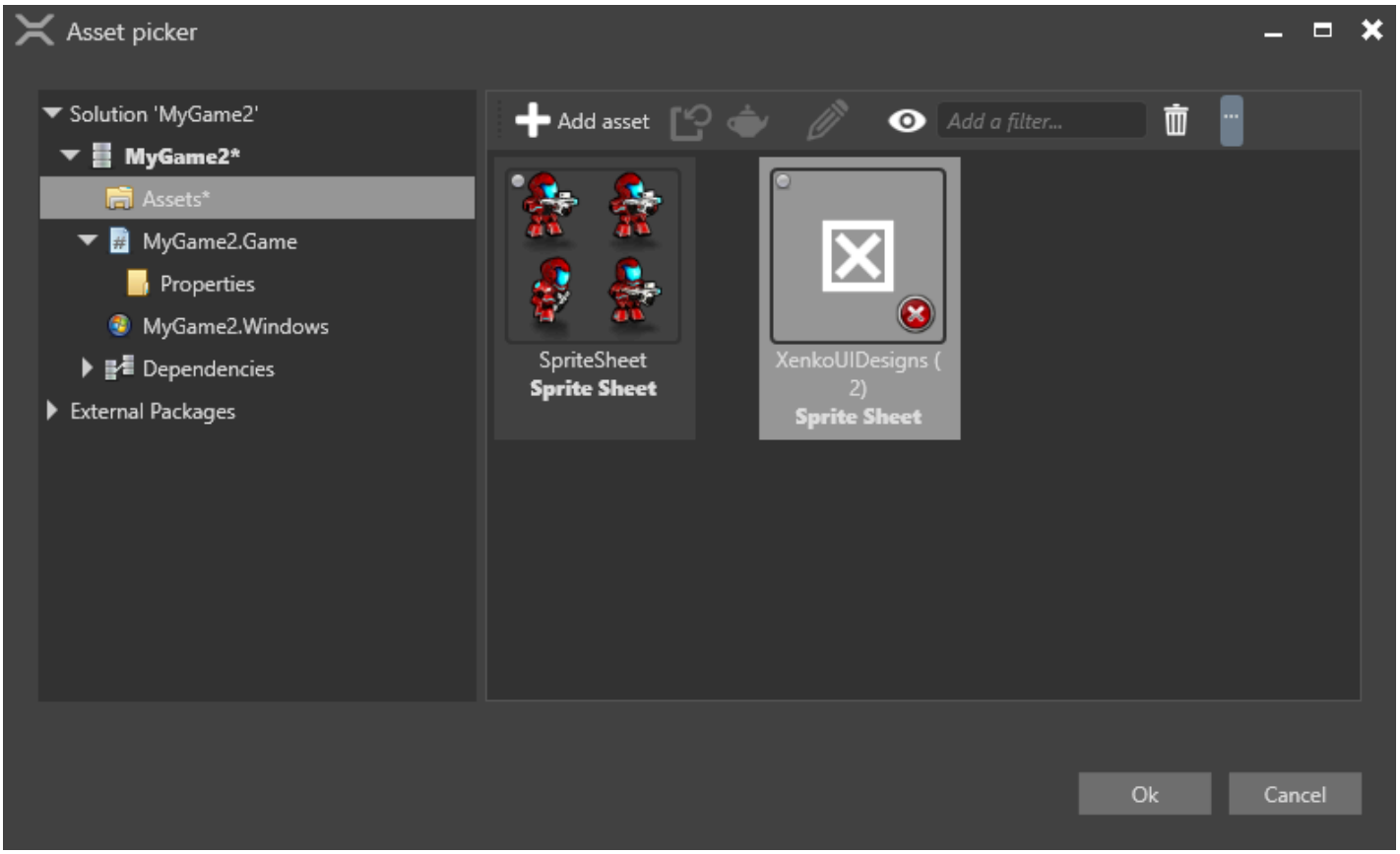
3. From the **Asset View**, drag the sprite sheet to the **Source** field in the Sprite component:



Alternatively, click (**Select an asset**):



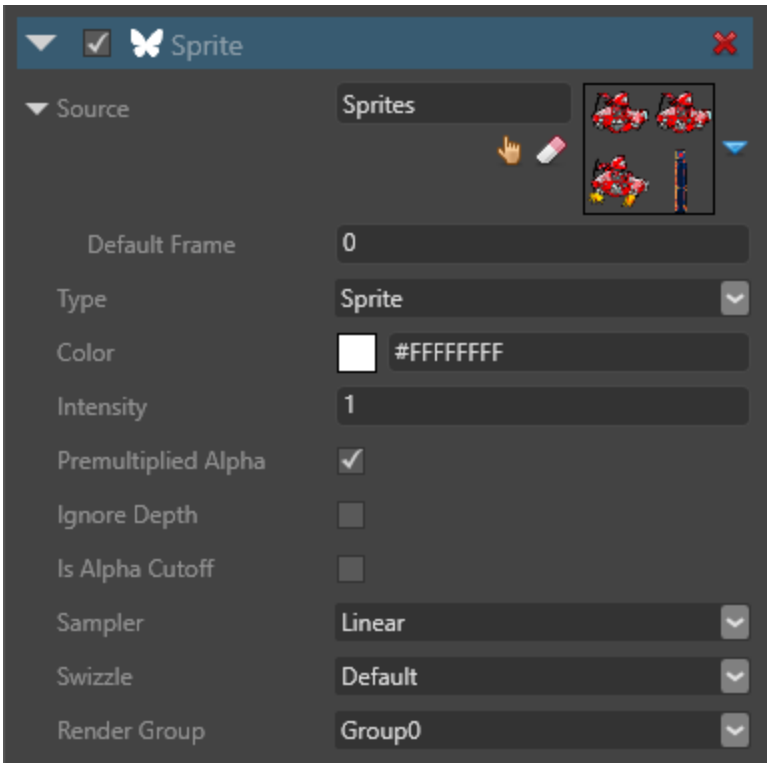
Then choose a sprite sheet:



Game Studio adds the sprite to the entity.

Sprite component properties

You can access the sprite component properties in the **Property Grid**.



Property	Function
Source	The source image file for the sprite
Type	**Sprites** have 3D space in the scene. **Billboards** always face the camera and appear fixed in 3D space.
Color	Applies a color to the sprite
Intensity	The intensity by which the color is scaled (mainly used for rendering LDR sprites in HDR scenes)
Premultiply alpha	Premultiply color components by their alpha component
Ignore depth	Ignore the depth of other elements in the scene when rendering the sprite. This always places the sprite on top of previous elements.
Alpha cutoff	Ignore pixels with low alpha values when rendering the sprite
Sampler	The texture sampling method used for the sprite: Point (nearest), Linear, or Anisotropic
Swizzle	How the color channels are accessed. **Default** leaves the image unchanged (finalRGB = originalRGB) **Normal map** uses the color channels as a normal map **Grayscale (alpha)** uses only the R channel (finalRGBA = originalRRRR), so the sprite is red **Grayscale (opaque)** is the same as **Grayscale (alpha)**, but uses a value of `1` for the alpha channel, so the sprite is opaque
Render group	Which render group the sprite belongs to. Cameras can render different groups. For more information, see Render groups and render masks.

Use sprites in a script

You can use scripts to render sprites at runtime. To do this, attach the script to an entity with a sprite component.

For information about how to add scripts to entities, see Use a script.

Code sample

This script displays a sprite that advances to the next sprite in the index every second. After it reaches the end of the sprite index, it loops.

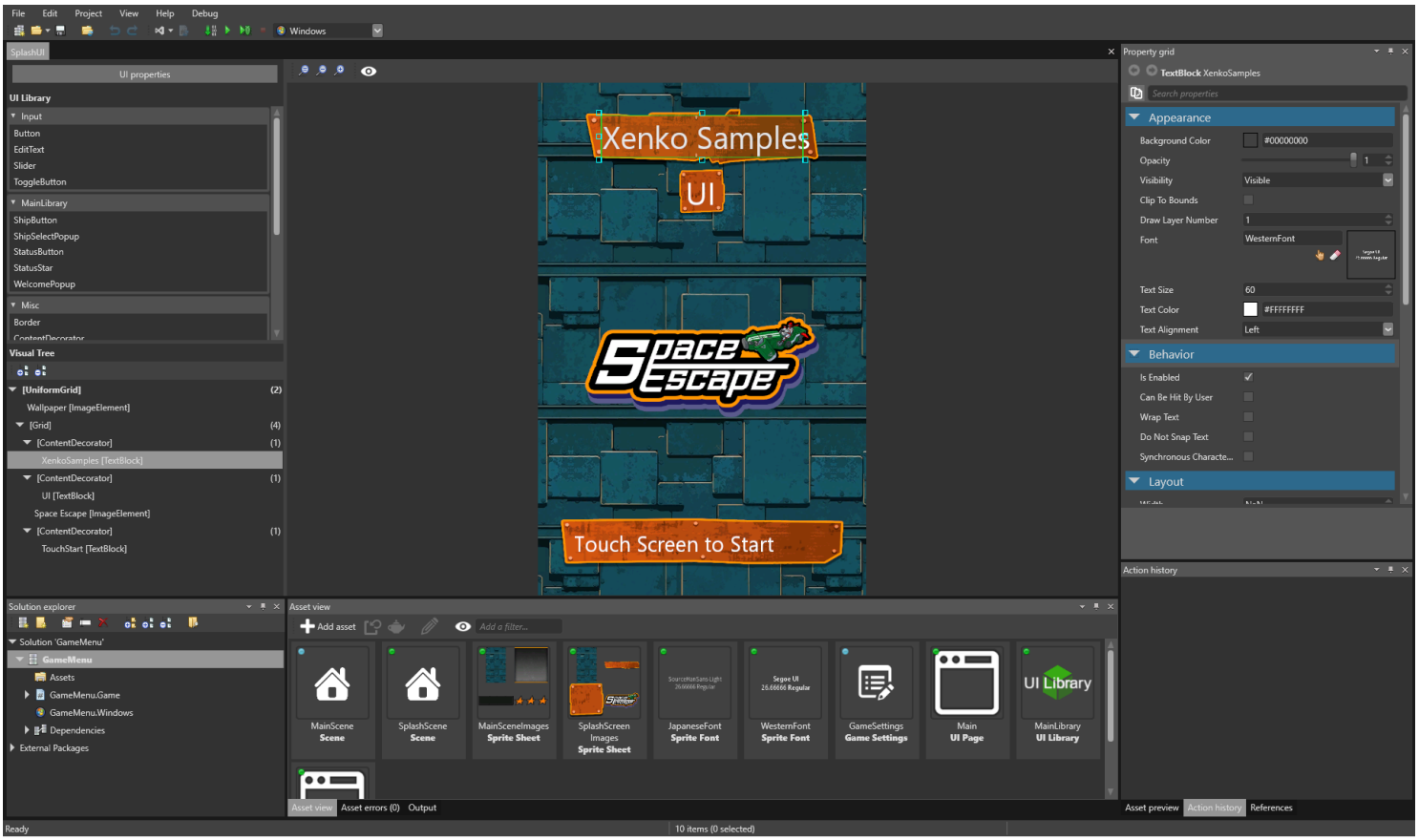
```
using Stride.Rendering.Sprites;

public class Animation : SyncScript
{
   // Declared public member fields and properties are displayed in Game Studio.
   private SpriteFromSheet sprite;
   private DateTime lastFrame;

   public override void Start()
   {
       // Initialize the script.
       sprite = Entity.Get<SpriteComponent>().SpriteProvider as SpriteFromSheet;
       lastFrame = DateTime.Now;
   }

   public override void Update()
   {
      // Do something every new frame.
      if ((DateTime.Now - lastFrame) > new TimeSpan(0, 0, 1))
      {
         sprite.CurrentFrame += 1;
         lastFrame = DateTime.Now;
      }
   }
}
```

See also

- Import sprite sheets
- Edit sprites

UI

Stride features a UI editor and layout system you can use to build sophisticated user interfaces. It supports 2D and 3D independently of resolution.



Stride uses two types of UI asset: `UIPageAsset` and `UILibraryAsset`. Their runtime counterparts are `UIPage` and `UILibrary` respectively.

To reduce the number of draw calls, Stride draws multiple elements using a sprite batch renderer.

Controls

Stride features many UI control components, including:

- ImageElement
- ContentControl
  - ScrollViewer
  - Button
    - ToggleButton
  - ContentDecorator
- TextBlock
  - ScrollingText
- EditText (displays soft keyboard on mobile devices)

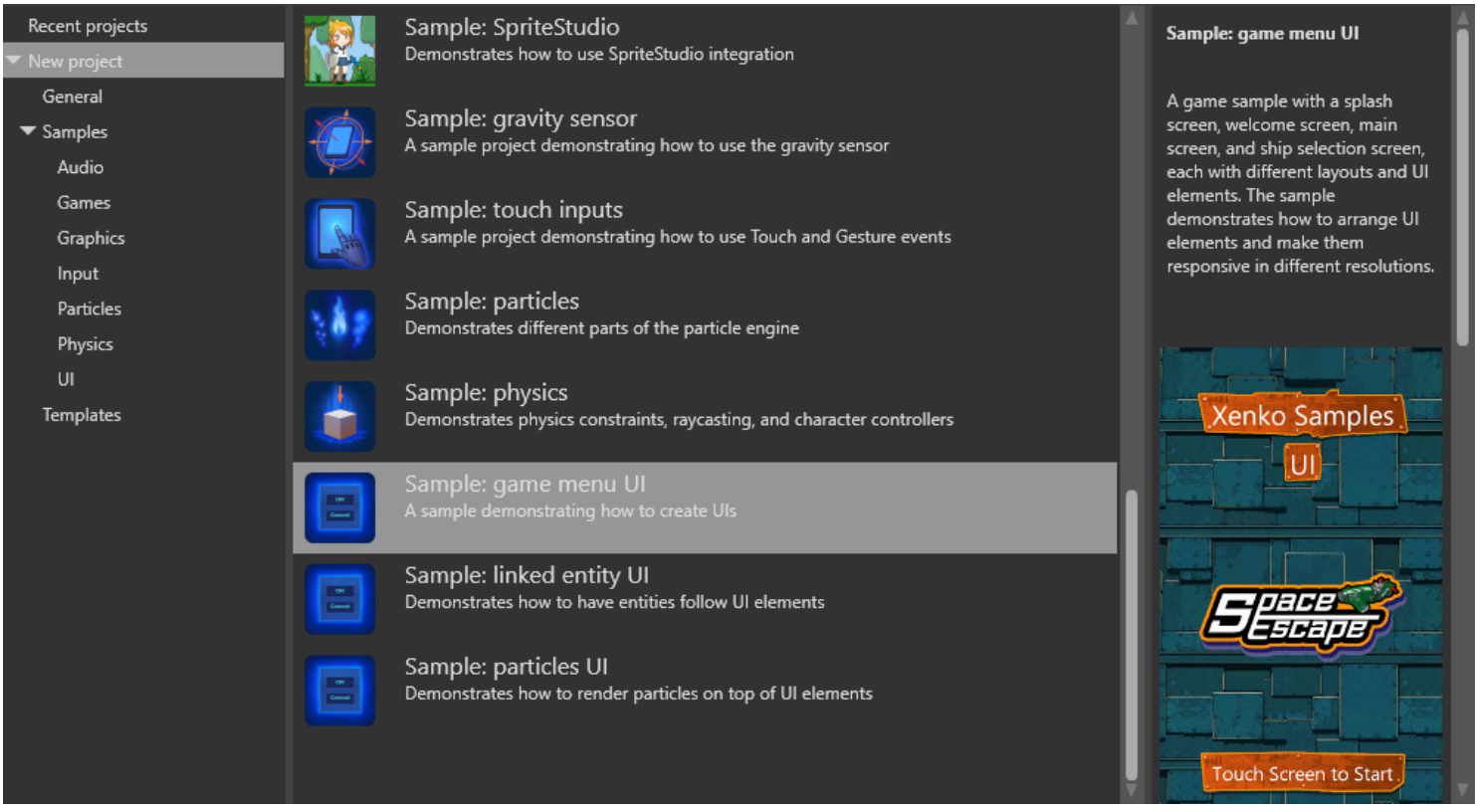
- Panel
  - StackPanel (supports virtualization)
  - Grid
  - UniformGrid
  - Canvas
- ScrollBar
- ModalElement

You can also create your own.

Sample project

Without scripts, UIs are simply non-interactive images. To make them interactive, add a script.

For an example of a UI implemented in Stride, see the **game menu UI** sample included with Stride.



In this section

- UI pages
- UI libraries
- UI editor
- Add a UI to a scene
- Layout system

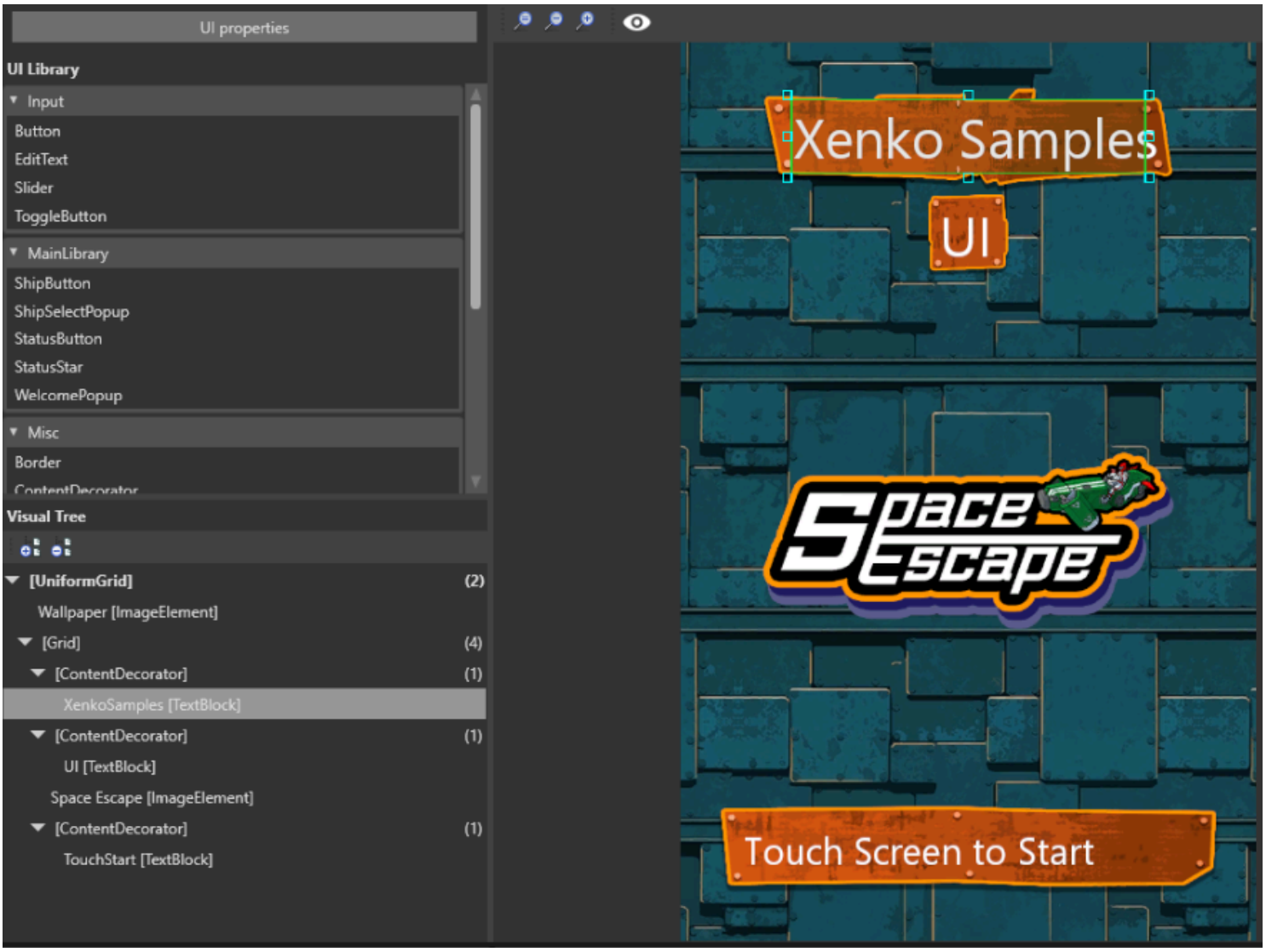
See also

- VR — Display a UI in an overlay

UI pages

Beginner Artist Designer

A **UI page** is a collection of UI elements, such as buttons and grids, that form a piece of UI in your game, such as a menu or title screen.

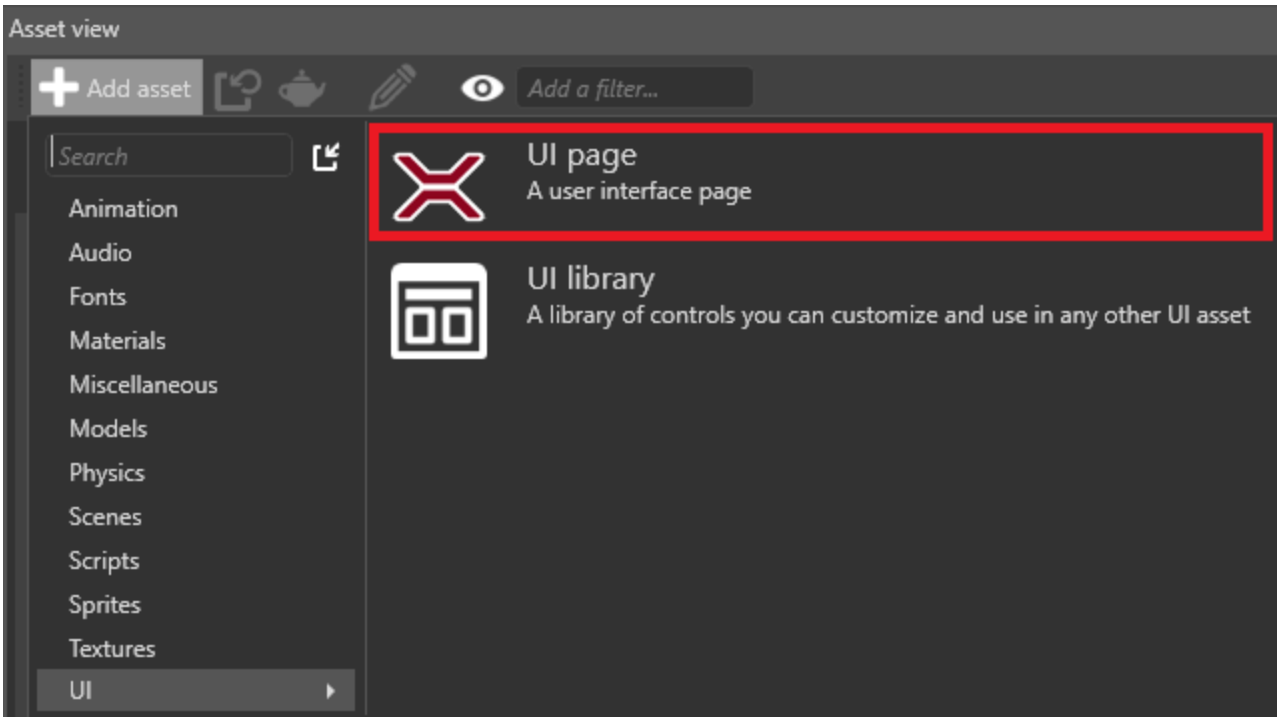


In terms of organization, a UI page is equivalent to a scene in the Scene Editor, and UI elements are equivalent to entities in a scene. Just like entities, elements can have parents and children.

Each UI scene opens in its own tab in the UI editor. For information about how to edit UI pages, see the UI editor page.

Create a UI page

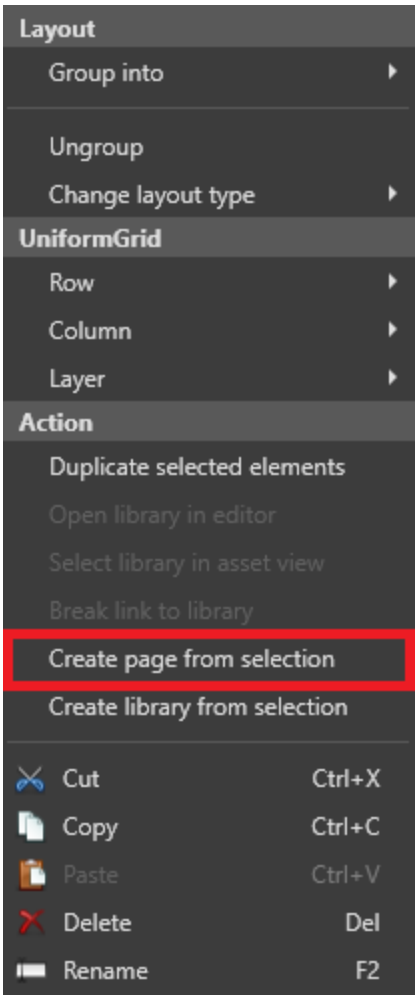
In the **Asset View**, click **Add asset > UI > UI page**.



Game Studio adds the UI page to the Asset View.

Create a UI page from a UI element

1. In the UI editor, select the element or elements you want to create a page from.
2. Right-click and select **Create page from selection**.

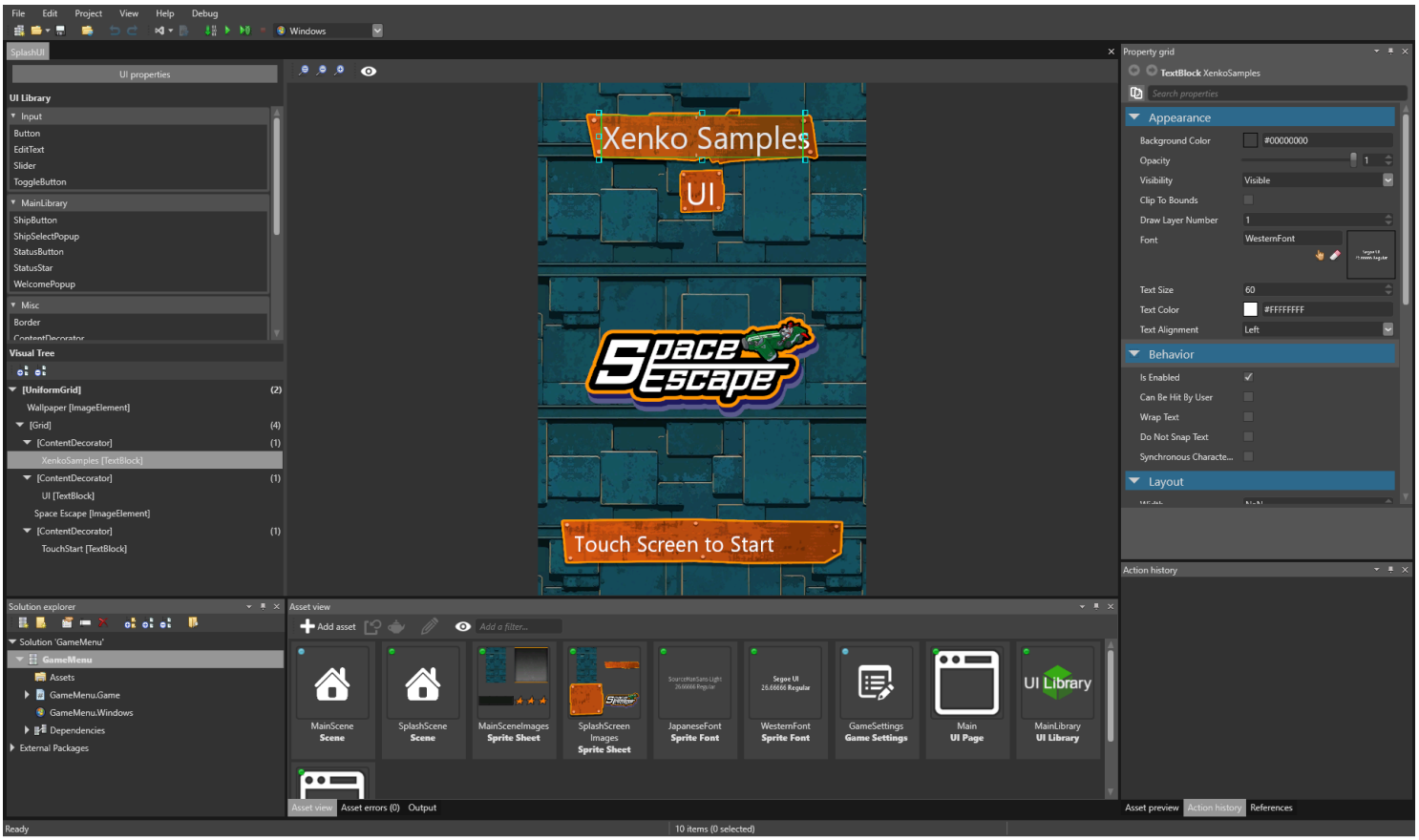


Game Studio creates a page with a copy of the elements you selected.

Open a UI page

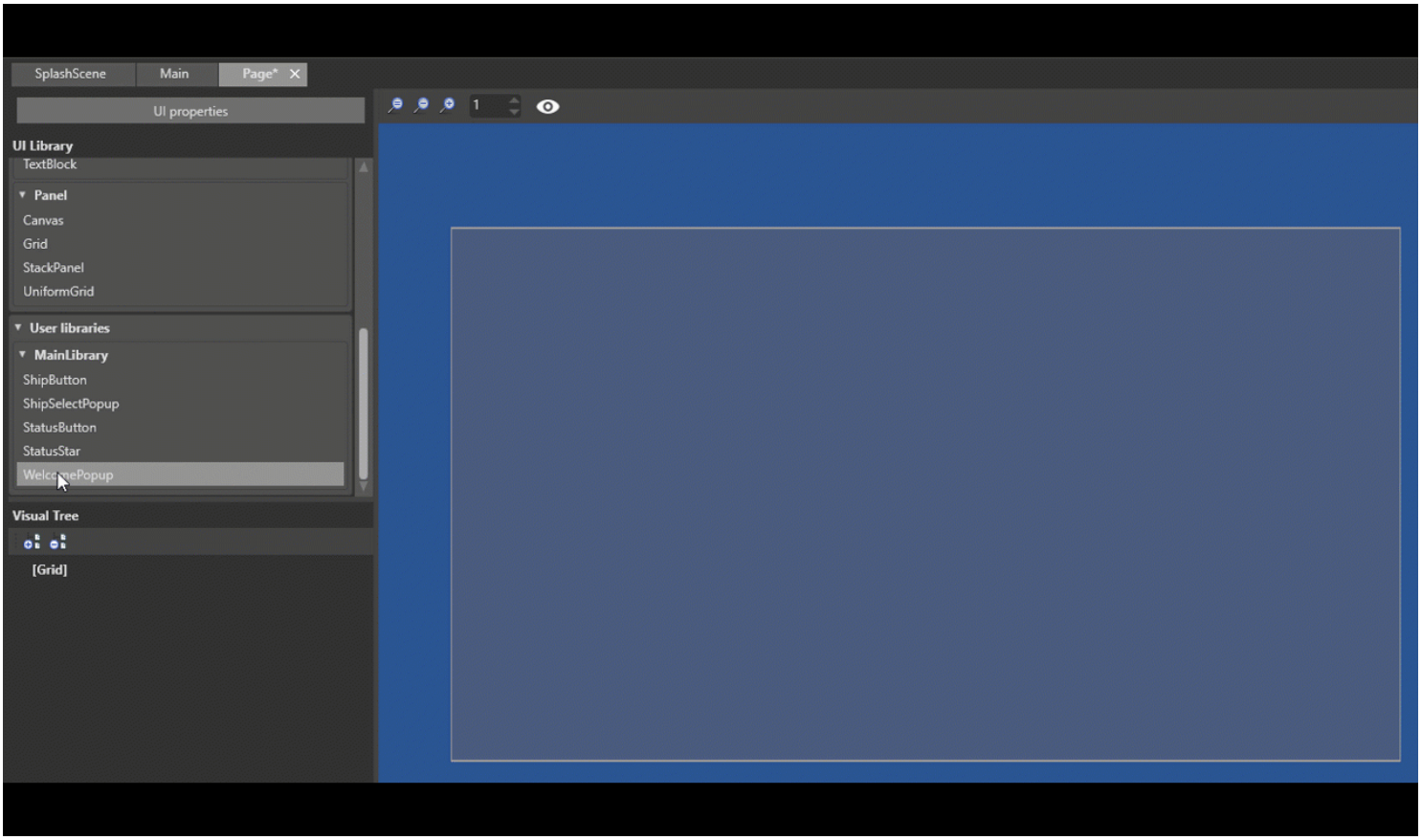
In the **Asset View**, double-click the **UI page**.

The UI page opens in the UI editor.



Add a UI element to a UI page

To add an element, such as a grid or button, drag it from the **UI library** to the UI page or the **visual tree**.



For more information about how to use the UI editor, see the UI editor page.

See also

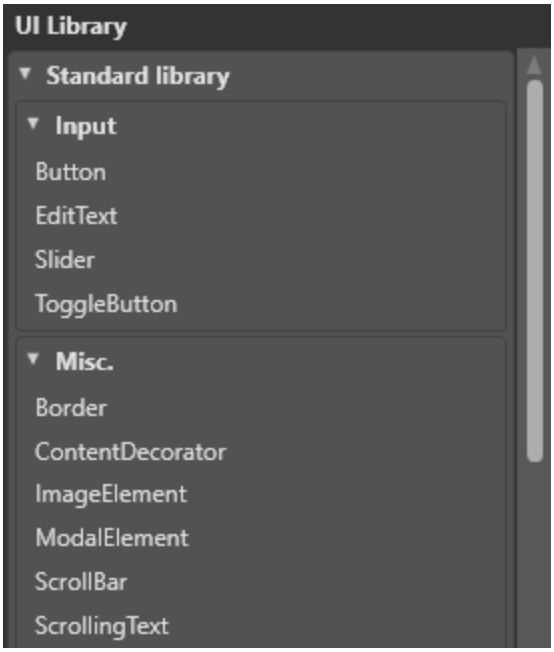
- UI libraries
- UI editor
- Add a UI to a scene
- Layout system

UI libraries

Beginner Artist Designer

UI libraries contain **UI elements** such as grids, buttons, sliders and so on that you can use and re-use in your UI pages.

Stride projects include a **standard library** of UI elements. You can create your own libraries of custom elements too.

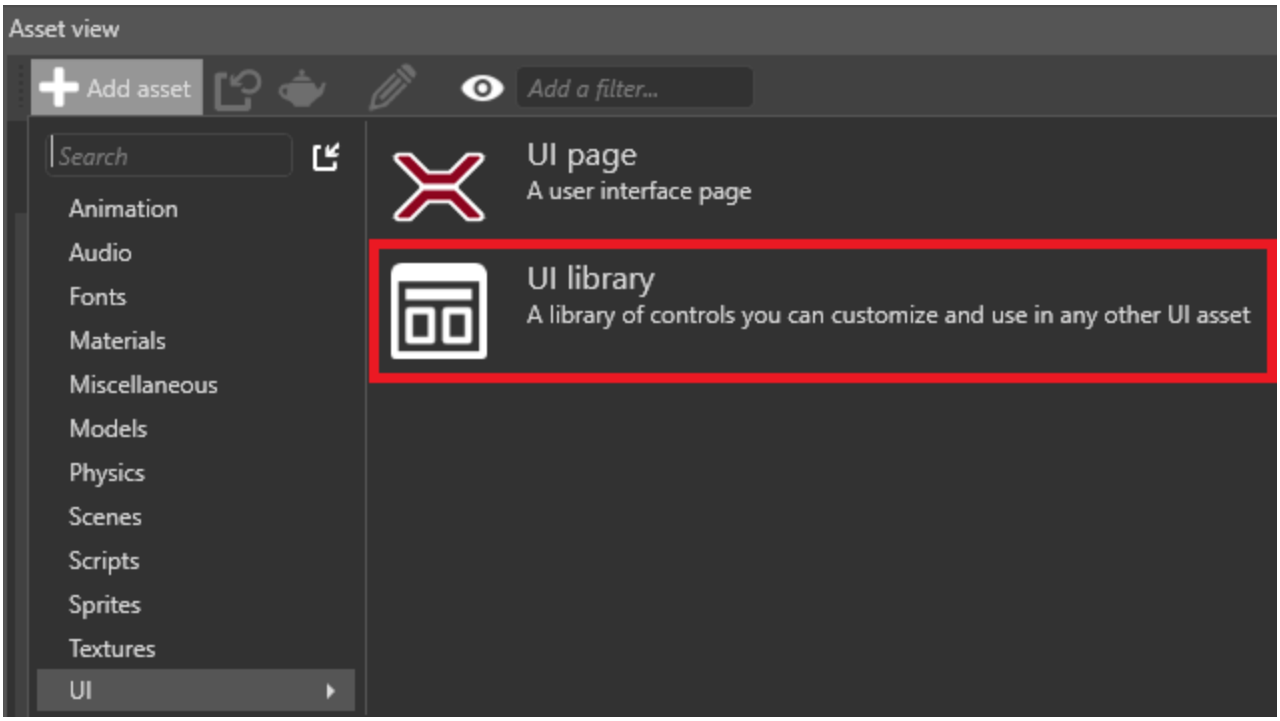


UI libraries are similar to prefabs in the Scene Editor; you can create your own elements, save them in a custom UI library, and then use them wherever you need across multiple UI pages. You can also nest libraries inside other libraries, just like nested prefabs.

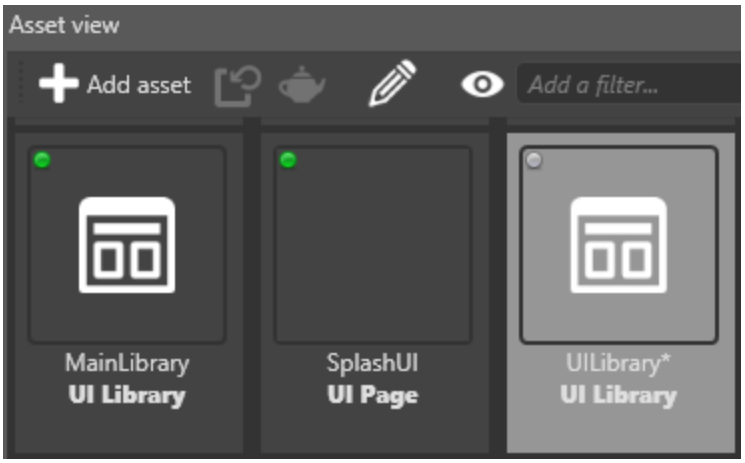
At runtime, you can re-instantiate UI library roots and insert them into an existing UI tree.

Create a UI library

In the **Asset View**, click **Add asset > UI > UI library**.

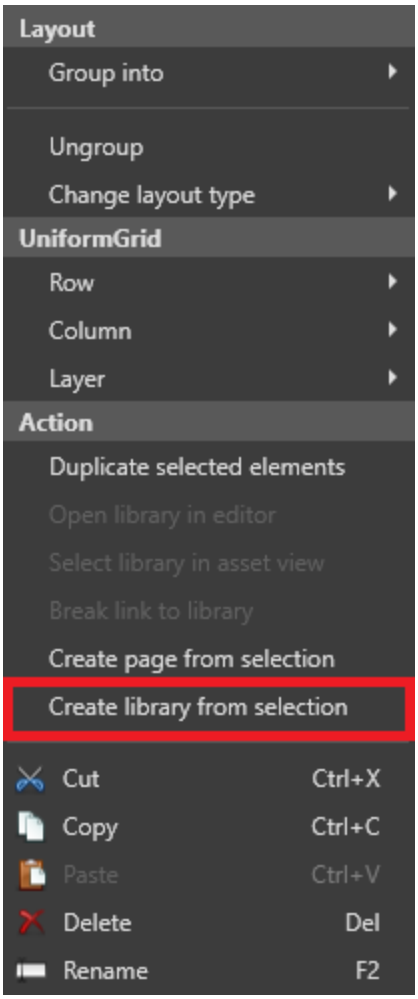


Game Studio adds the UI library to the **Asset View**.



Alternatively, to create a UI library from an existing UI element:

1. Select the elements you want to create a UI library from.
2. Right-click and select **Create library from selection**.



Game Studio creates a library with a copy of the elements you selected.

Assign a UI library in code

```
// This property can be assigned from a UI library asset in Game Studio
public UILibrary MyLibrary { get; set; }

public Button CreateButton()
{
    // assuming there is a root element named "MyButton" of type (or derived from) Button
    var button = MyLibrary.InstantiateElement<Button>("MyButton");

    // if there is no root named "MyButton" in the library or the type does not match,
    // the previous method will return null
    if (button != null)
    {
        // attach a delegate to the Click event
        button.Click += delegate
        {
            // do something here...
        };
```

```
    }

    return button;
}
```

UI pages have only one root element. UI libraries can have multiple root elements.

See also

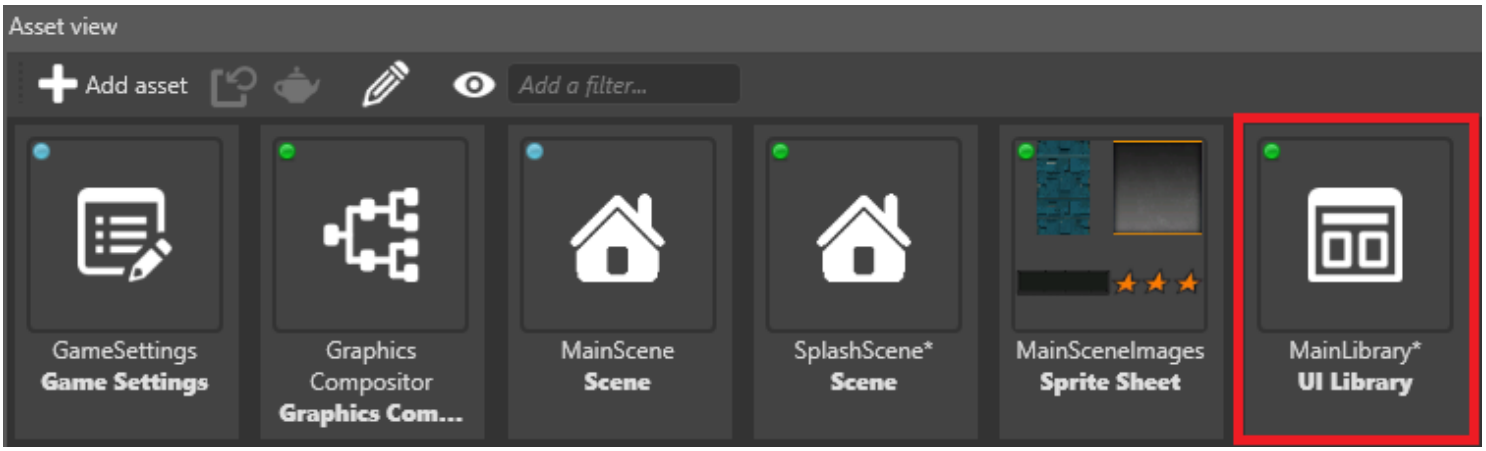
- UI pages
- UI editor
- Add a UI to a scene
- Layout system

UI editor

Beginner Artist Designer

You can edit UI pages and UI libraries with the **UI editor**.

To open the editor, in the **Asset View**, double-click a **UI page** or **UI library**.



The UI editor opens.

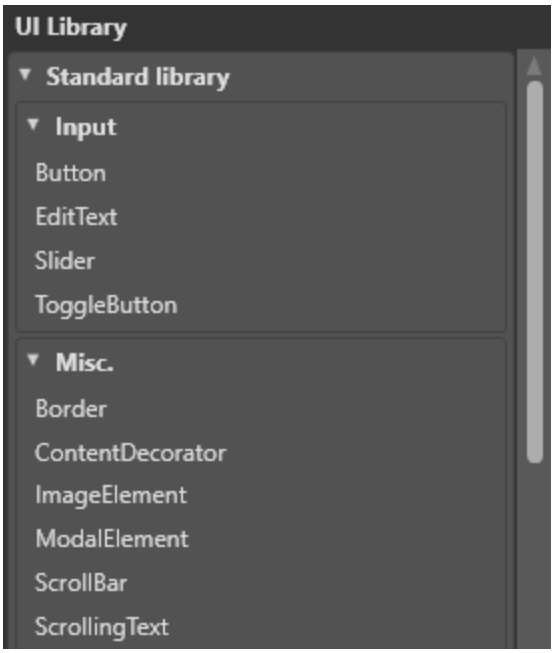
The UI editor comprises:

- the list of UI libraries (**A**), which contain the elements (such as buttons and grids) you can add to your UI
- a visual tree of the elements in the UI page (**B**)

- a preview of the UI page as it appears in the game (**C**)
- a Property Grid (**D**) to edit the properties of your UI elements

UI libraries

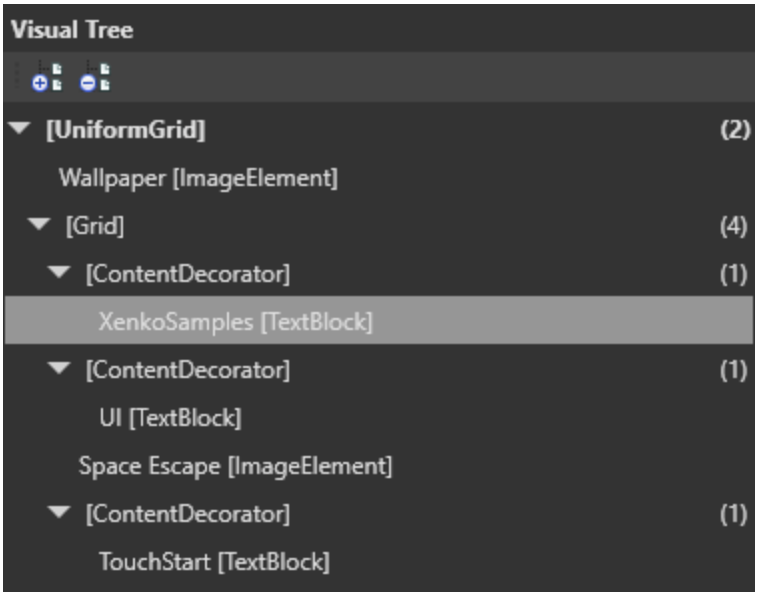
A **UI library** contains **UI elements** (such as grids, buttons, sliders and so on) that you can use and re-use in your UI pages. They work similarly to prefabs in the Scene Editor.



For more information, see UI libraries.

Visual tree

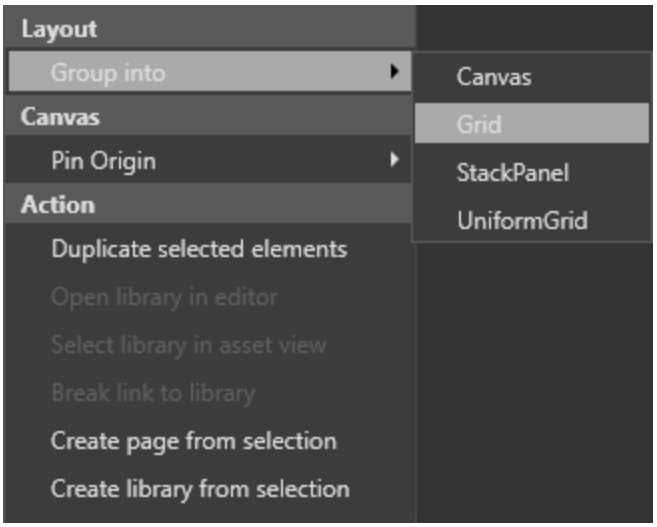
The **visual tree** shows the elements in the UI page and their hierarchy. This is similar to the **Entity Tree** in the Scene Editor.



The number in parentheses indicates the number of children an element has. Some elements, such as buttons, can only have one child.

To re-order elements in the visual tree, drag and drop them.

To move an element to a new group, right-click the element and select **Group into**. For example, to create a new grid and move an element into it, right-click the element and select **Group into > Grid**.



UI preview

The **UI preview** displays a preview of the UI as it appears at runtime. The rendering is equivalent to the rendering in the game, assuming the design resolution is the same as the UI component that uses the edited asset.

By default, the UI is a **billboard**, meaning it always faces the camera. The UI view camera is **orthographic** (see Cameras).

You can select, move, and resize elements in the preview as you do in image editing applications.

Space
Escape

Space
Escape

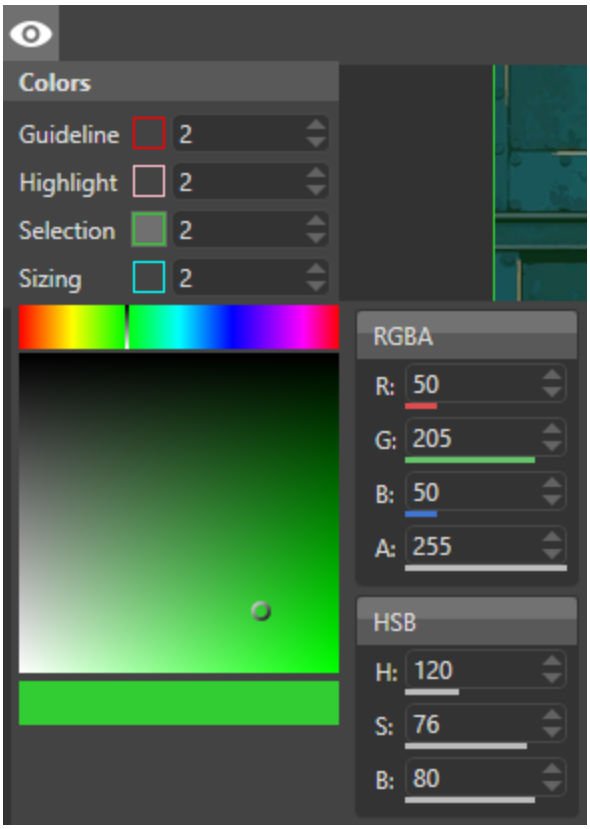
Controls

Action	Control
Pan	Hold middle mouse button + move mouse
Zoom	Mouse wheel
Speed up pan/zoom	Hold shift while panning or zooming

Tool options

To change the color and size of the selection tools, in the **UI editor toolbar**, click

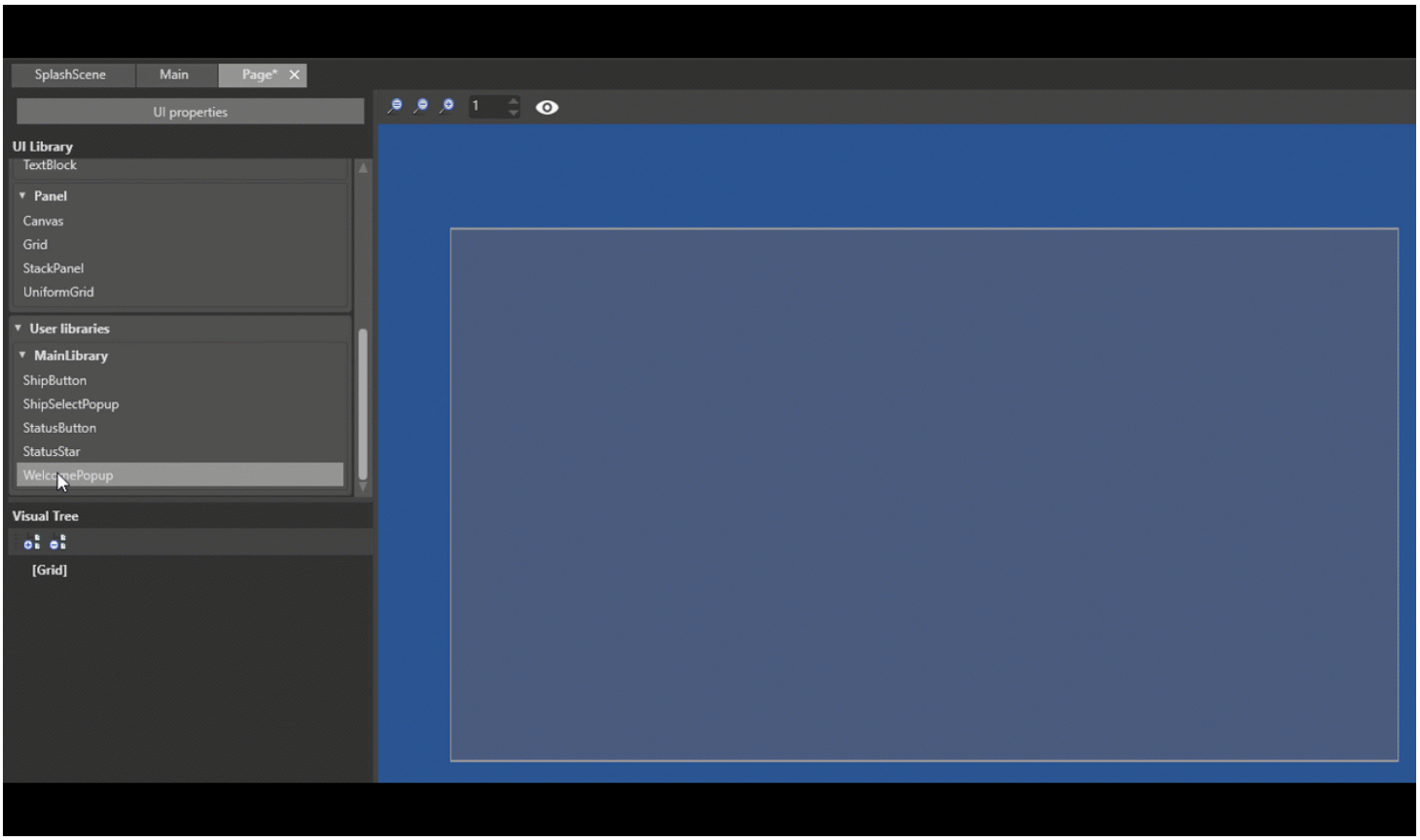
> **NOTE**
> These options have no effect on how the UI is displayed at runtime.



- **Guideline**: changes the width of the lines that indicate the distance to the margins (in pixels)
- **Highlight**: changes the width of the highlight that appears when you move your mouse over an element
- **Selection**: changes the width of the selection highlight
- **Sizing**: changes the size of the boxes at the edges of selections used to resize elements

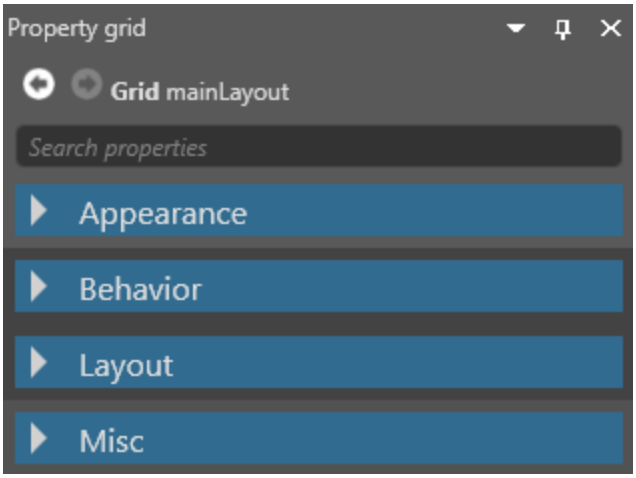
Add a UI element to a UI page

To add an element (such as a grid or button), drag it from the **UI library** to the UI page or the **visual tree**.



Properties

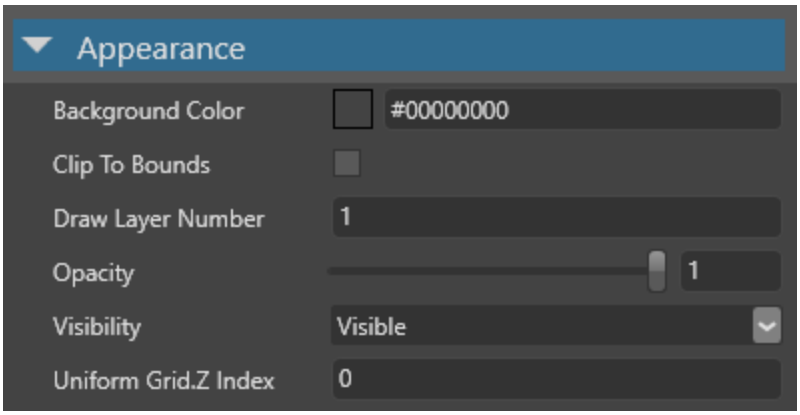
You can view and edit element properties in the **Property Grid**.



Properties are sorted by **Appearance**, **Behavior**, **Layout** and **Misc**.

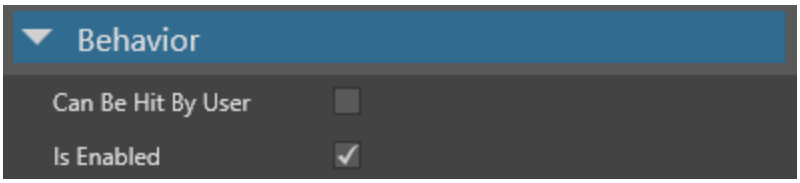
Appearance

Commonly used properties include `BackgroundColor`, `Opacity`, `Visibility` and `ClipToBounds`.



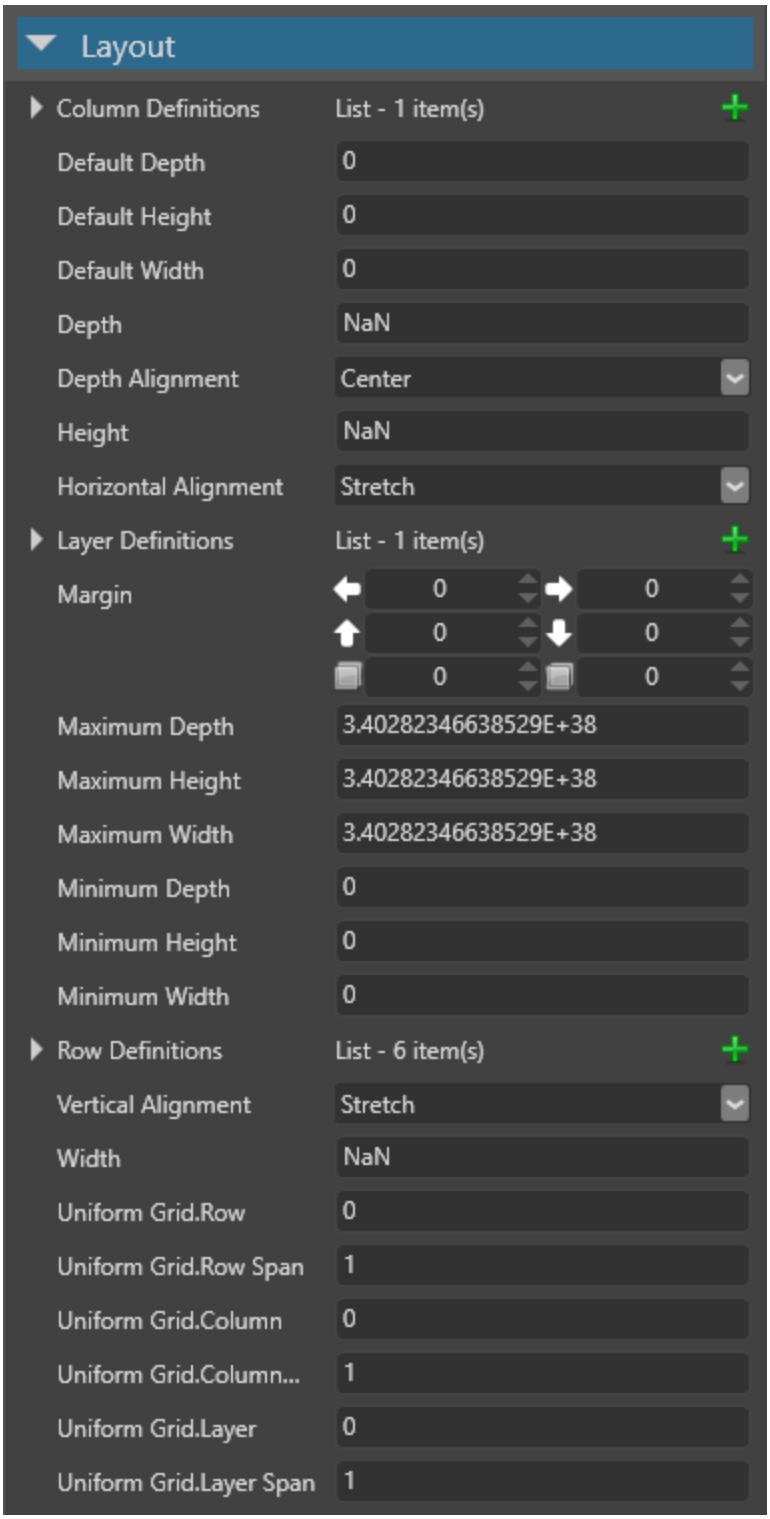
Behavior

Commonly used properties include whether the element responds to touch events(`CanBeHitByUser`).



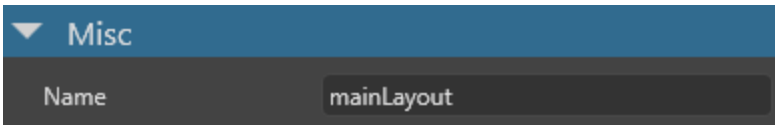
Layout

Commonly used properties include the size of the element (`Height`, `Width` and `Depth`), its alignment (`HorizontalAlignment`, `VerticalAlignment`, `DepthAlignement`) and its `Margin`.



Misc

This category contains only the `Name` of the element.



See also

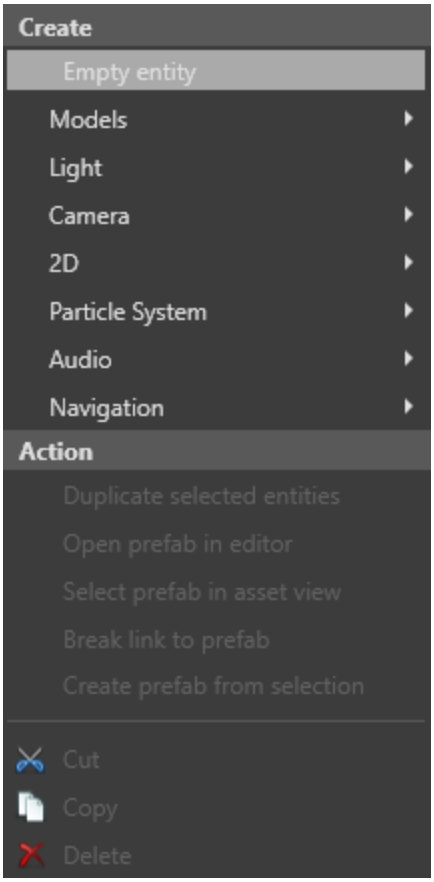
- UI pages
- UI libraries
- Add a UI to a scene
- Layout system

Add a UI to a scene

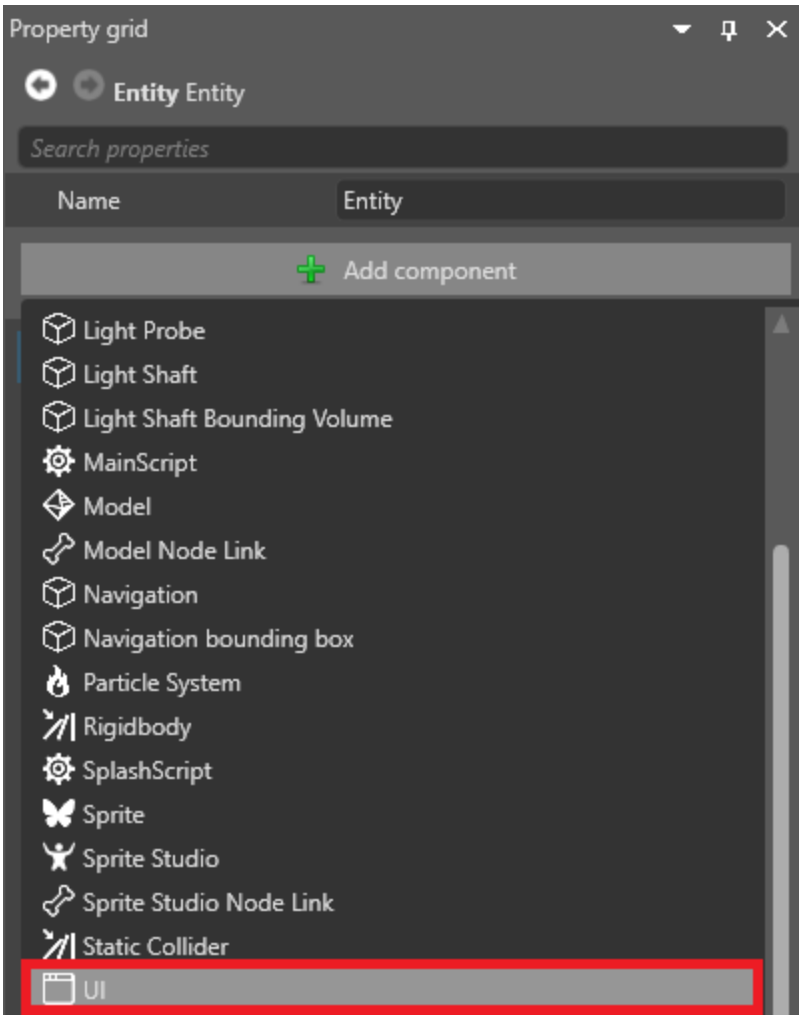
Beginner Artist Designer

After you create a UI page, add it to the scene as a component on an entity.

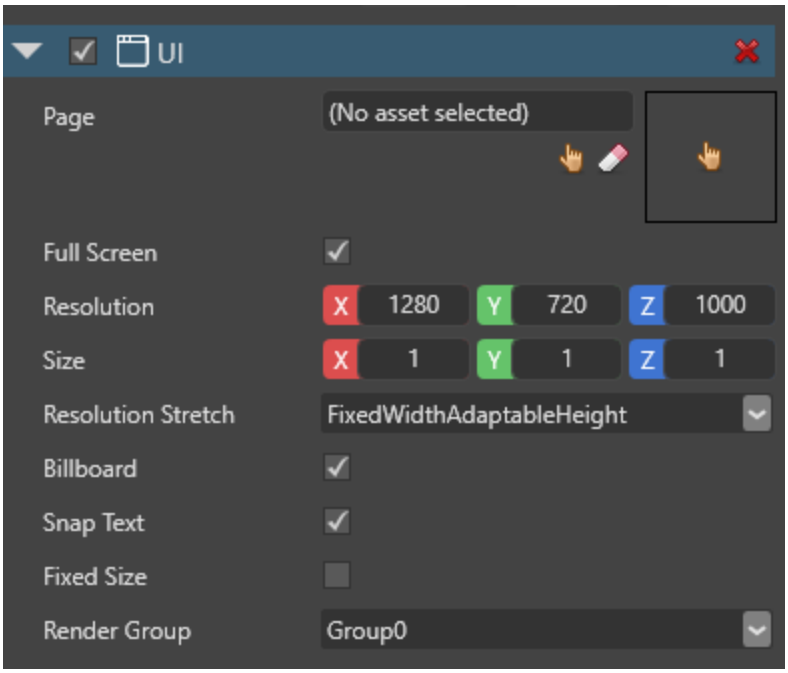
1. In the **Scene Editor**, create an empty entity. To do this, right-click the scene and select **Empty entity**.



2. In the Property Grid (on the right by default), click **Add component** and select **UI**.

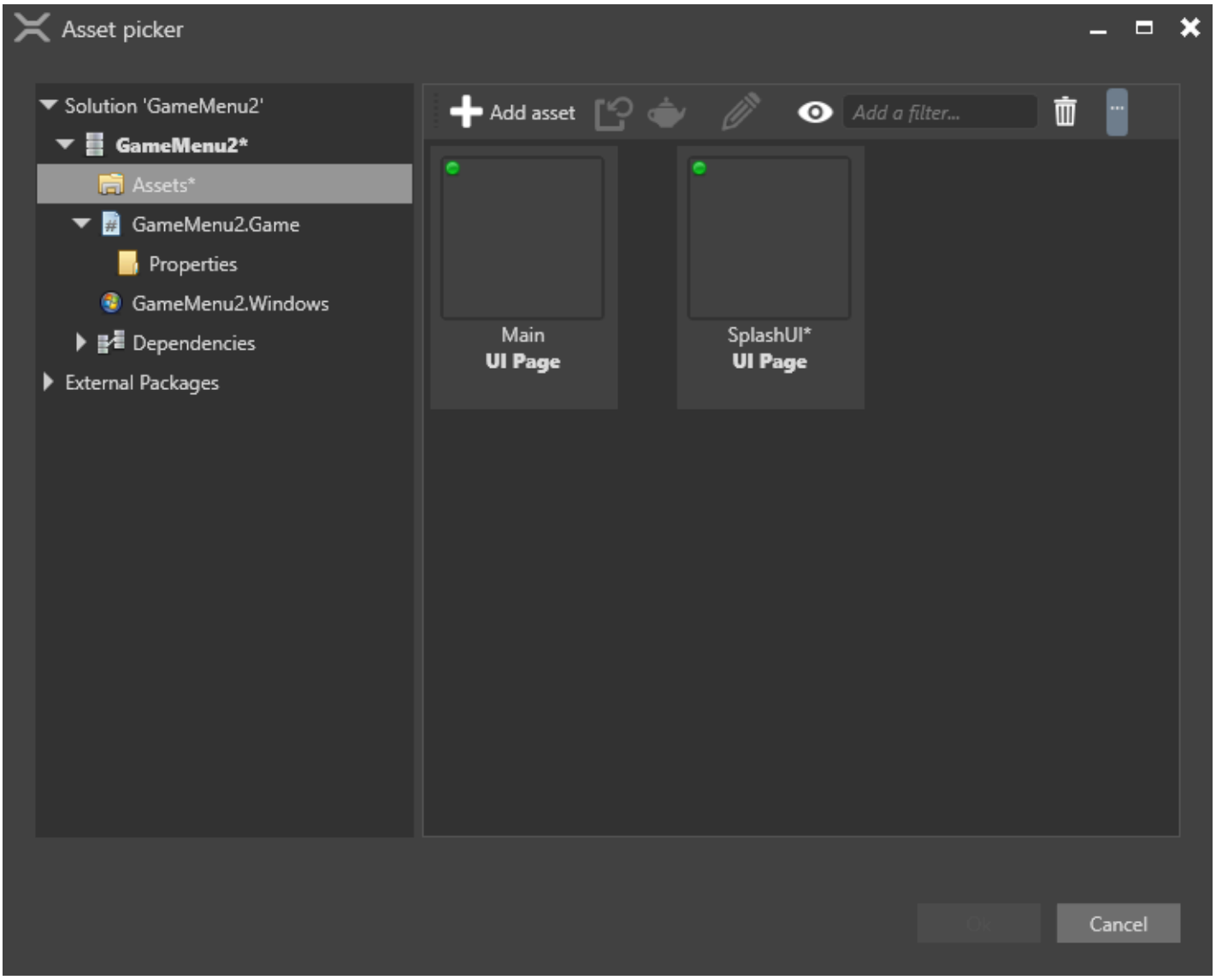


Game Studio adds a **UI component** to the entity.



3. Next to **Page**, click (**Select an asset**).

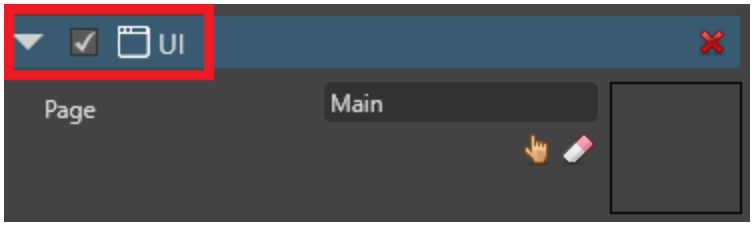
The **Select an asset** window opens.



4. Select the UI page you want to add and click **OK**.

   For information about how to create and edit UI pages, see the UI editor page.

> **TIP**
>
> To stop the UI obscuring the rest of the scene in the editor, disable the **UI component** in the Property Grid.



> Remember to enable the component again before you run the game. If you don't, Stride doesn't display the UI.

Assign a UI page to a UI page component in code

You can assign a UI page to the `Page` property of a UI component.

```
// This property can be assigned from a UI page asset in Game Studio
public UIPage MyPage { get; set; }

protected override void LoadScene()
{
    InitializeUI();
}

public void InitializeUI()
{
    var rootElement = MyPage.RootElement;
    // to look for a specific element in the UI page, extension methods can be used
    var button = rootElement.FindVisualChildOfType<Button>("buttonOk");

    // if there's no element named "buttonOk" in the UI tree or the type doesn't match,
    // the previous method returns null
    if (button != null)
    {
        // attach a delegate to the Click event
        button.Click += delegate
        {
            // do something here...
        };
    }

    // assign the page to the UI component
    var uiComponent = Entity.Get<UIComponent>();
    uiComponent.Page = MyPage;
}
```

UI component properties

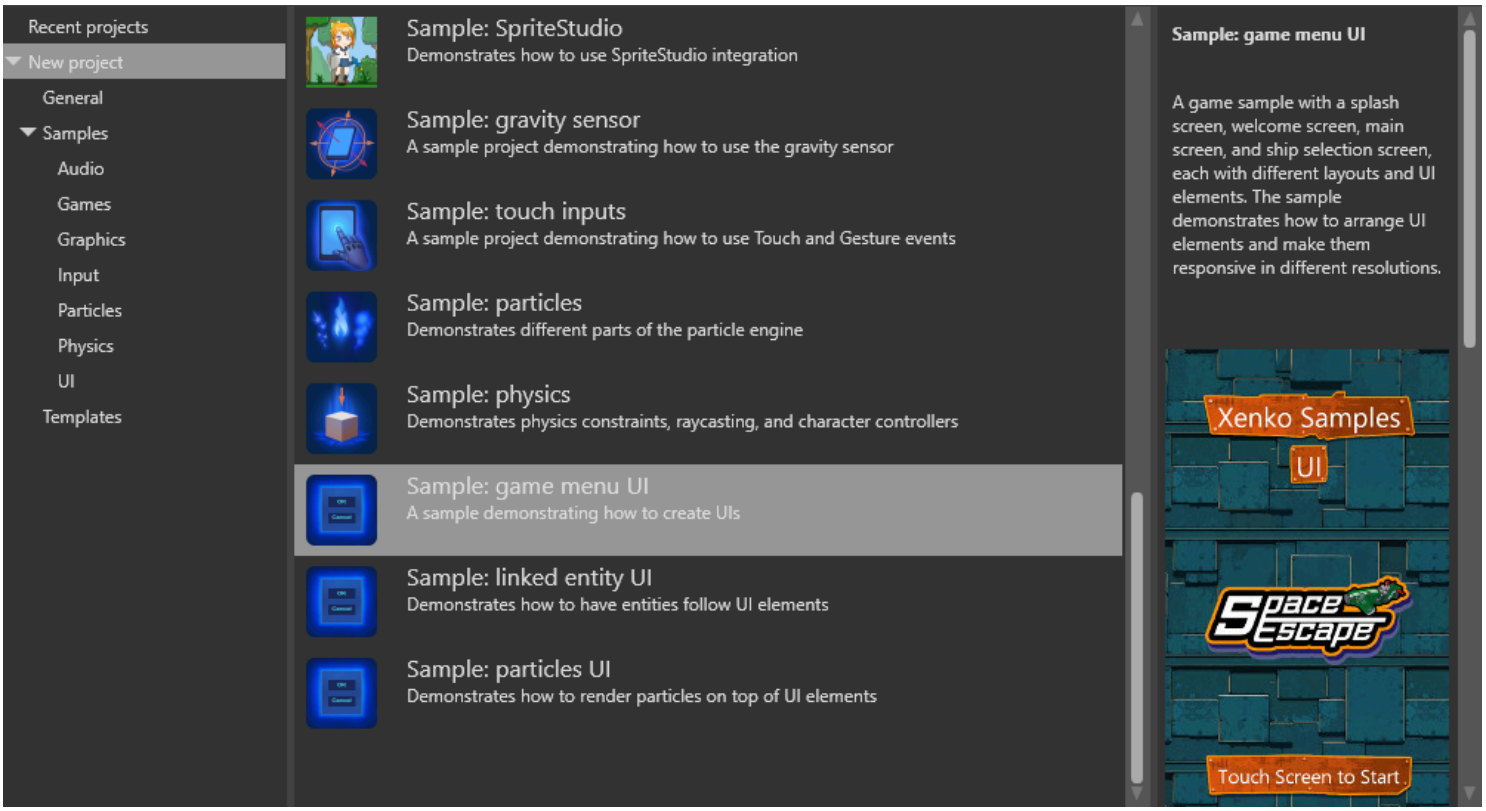
Property	Description
Page	The UI page displayed by the component

Property	Description
Sampler	Texture sampling method: Point (Nearest), Linear (**Default option**), or Anisotropic
Full screen	**Note:** We recommend you use this as other stuff is broken
Resolution	The UI resolution in pixels
Size	Gets or sets the actual size of the UI component in world units
Resolution stretch	How the virtual resolution value should be used (`FixedWithFixedHeight`, `FixedWithAdaptableHeight`, or `FixedHeightAdaptableWidth`)
Billboard	If selected, the UI always faces the camera. **Note:** Disabling billboard mode causes UI text errors in the current version of Stride
Snap text	If selected, the UI text is snapped to the closest pixel
Fixed size	Gets or sets the value indicating whether the UI should always be a fixed size on screen (eg a component with a height of 1 will use 0.1 of the screen). **Note:** This feature doesn't work in the current version of Stride
Render group	The render group the UI uses

UI scripts

To make UIs interactive, you need to add a script. Without scripts, UIs are simply non-interactive images.

For an example of a UI implemented in Stride, see the **game menu UI** sample included with Stride.



See also

- UI pages
- UI libraries
- UI editor
- Layout system
- VR — Display a UI in an overlay

Layout system

Intermediate Programmer Designer

The Stride UI layout system is similar to Windows Presentation Foundation (WPF). For more information about the WPF layout system, see the MSDN documentation. Much of the WPF documentation also applies to the Stride layout system.

Every UIElement in the Stride UI system has a surrounding rectangle used in layouts. Stride computes layouts according to the UIElement requirement, available screen space, constraints, margins, padding, and the special behavior of Panel elements (which arrange children in specific ways).

Processing this data recursively, the layout system computes a position and size for every UIElement in the UI system.

Measure and arrange

Stride performs the layout process recursively in two passes: Measure and Arrange.

Measure

In the Measure pass, each element recursively computes its DesiredSize according to the properties you set, such as Width, Height, and Margin.

Some Panel elements call Measure recursively to determine the DesiredSize of their children, and act accordingly.

Arrange

The Arrange pass arranges the elements, taking into account:

- Margin
- Width
- Height
- HorizontalAlignment
- VerticalAlignment
- Panel
- specific Arrange rules

See also

- MSDN WPF layout documentation
- UI pages
- UI libraries
- UI editor

- Add a UI to a scene

Video

Beginner Designer

You can import **video files** and use them in your scenes.

> **NOTE**
> Currently, Stride doesn't support video on iOS platforms.

In this section

- Set up a video
- Video properties
- Use a video as a skybox

Set up a video

Beginner Programmer Designer

> **NOTE**
>
> Stride supports most major video formats, but converts them to `.mp4`. To reduce compilation time, we recommend you use `.mp4` files so Stride doesn't have to convert them.

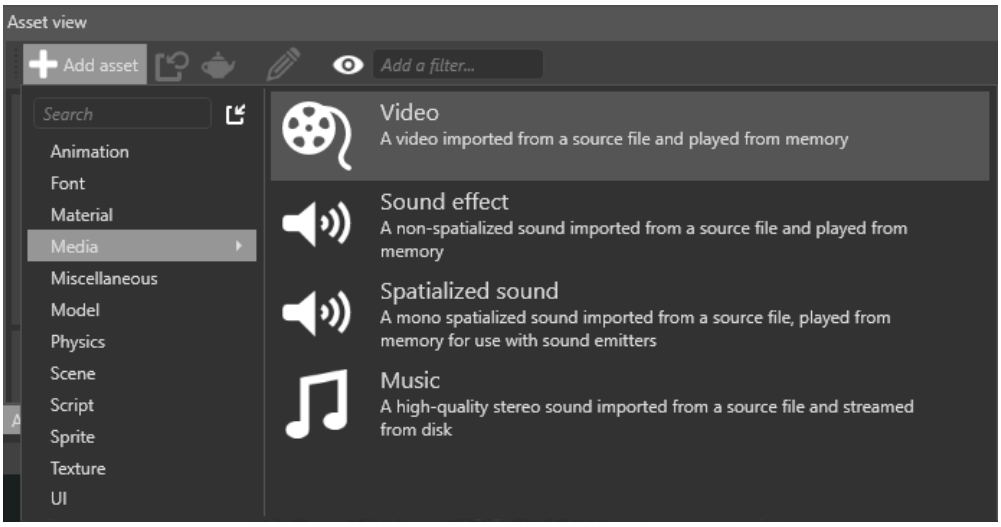
> **NOTE**
>
> Currently, Stride doesn't support video on iOS platforms.

1. Add a video asset

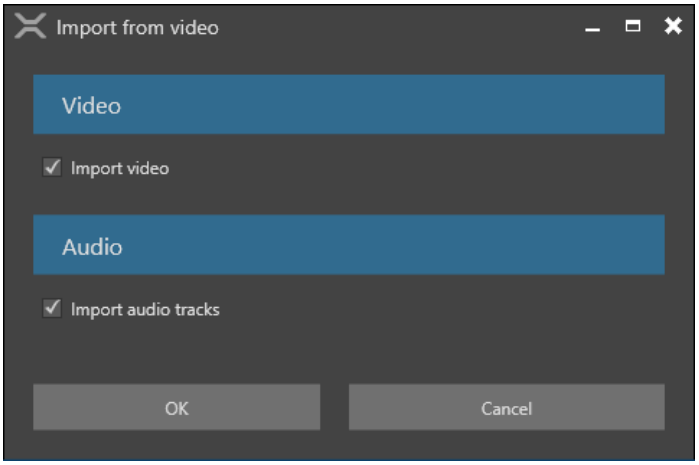
Before you can use a video in your game, you need to import it as an asset.

1. Drag the video file from **Explorer** into the **Asset View**.

   Alternatively, in the **Asset View**, click **Add asset** and select **Media > Video**, then browse to the video you want to add and click **Open**.

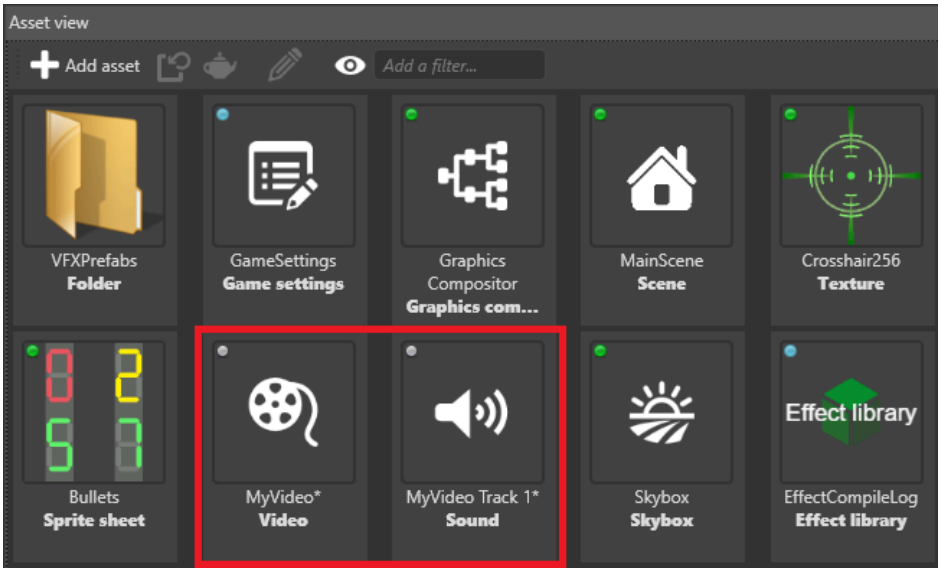


2. If the video has audio tracks, you can import these at the same time, or import just the audio from the video.



3. Click **OK**.

   Stride adds the video as an asset in the **Asset View**. If you imported audio tracks from the video file, Stride adds them as separate audio assets.



> **NOTE**
> Currently, you can't preview videos in the Asset Preview.

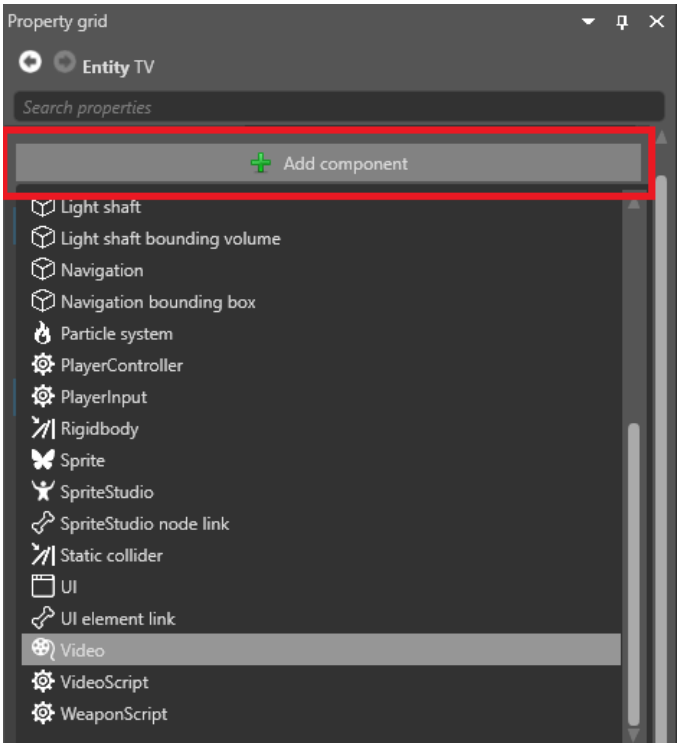
For information about video asset properties, see Video properties.

2. Add a video component

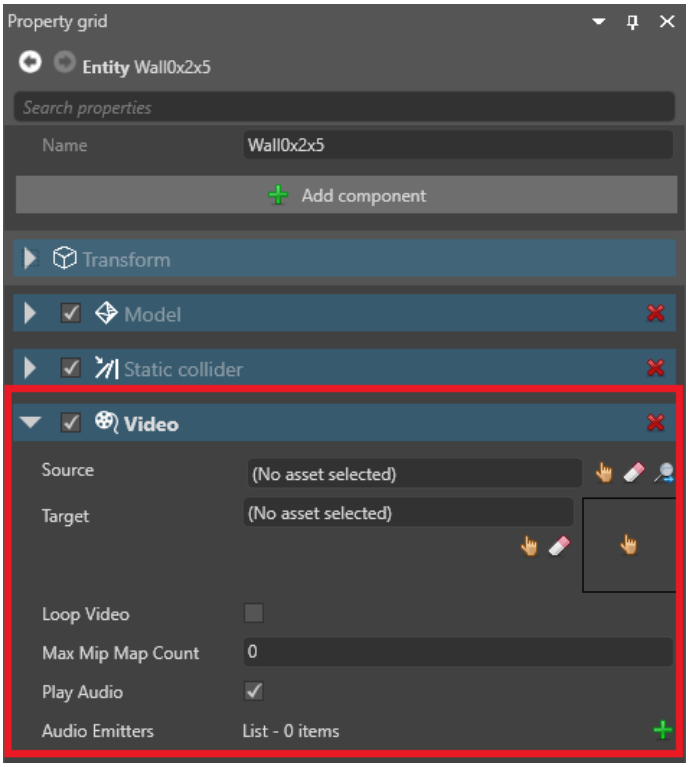
1. In the **Scene Editor**, select or create an entity to add a video component to.

> **TIP**
> It's usually simplest to add the component to the same entity that has the texture plays the video. This just makes it easier to organize your scene.

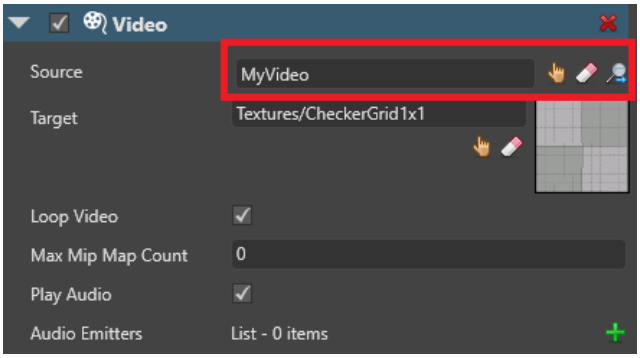
2. In the **Property Grid**, click **Add component** and select **Video**.



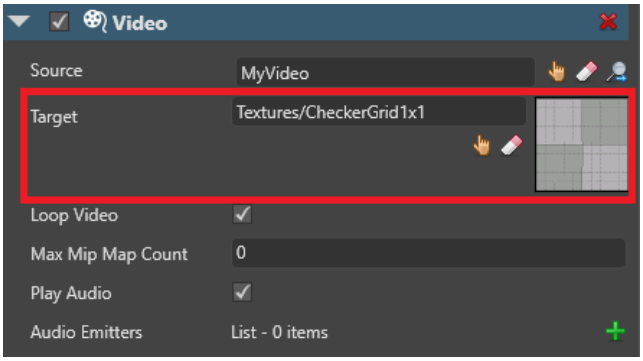
Stride adds a **video component** to the entity.



3. In the **Video** properties, under **Source**, select the video asset.



4. Under **Target**, select the texture you want to display the video from.



Models that use this texture will display the video.

When the video isn't playing in your scene, Stride displays the texture instead.

3. Create a script to play the video

After you set up the video component, play it from a script using:

```
myVideoComponent.Instance.Play();
```

Other functions

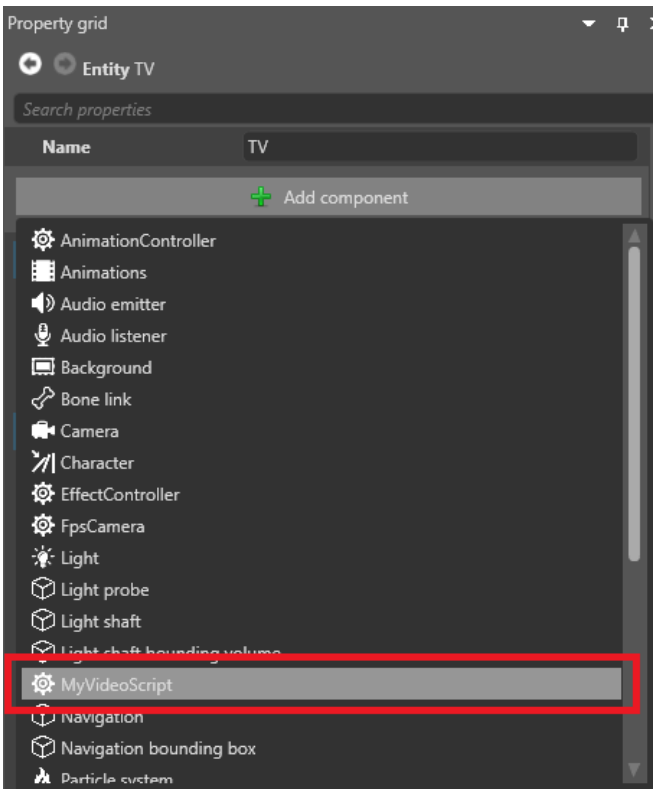
- `LoopRange`: The looping range (must be an area in the video in `PlayRange`)
- `IsLooping`: Loop the video loop infinitely
- `SpeedFactor`: Set the video play speed. `1` is normal speed.
- `PlayState`: The current video play state (`playing`, `paused` or `stopped`)
- `Duration`: The duration of the video
- `CurrentTime`: The current play time in the video
- `Volume`: The audio volume
- `PlayRange`: The video start and end time
- `Play/Pause/Stop`: Play, pause, or stop the video
- `Seek`: Seek to a given time

Example script

```
{
    public class VideoScript : StartupScript
    {
        // Game Studio displays the public member fields and properties you declare in this script

        public override void Start()
        {
            // Initialization of the script.
            Entity.Get<VideoComponent>().Instance.Play();
        }
    }
}
```

4. Add the script to the entity

1. In the **Scene Editor**, select the entity that has the video component.
2. In the **Property Grid**, click **Add component** and select the video script.



Stride adds the script as a component.

You can adjust public variables you define in the script in the **Property Grid** under the script component properties.

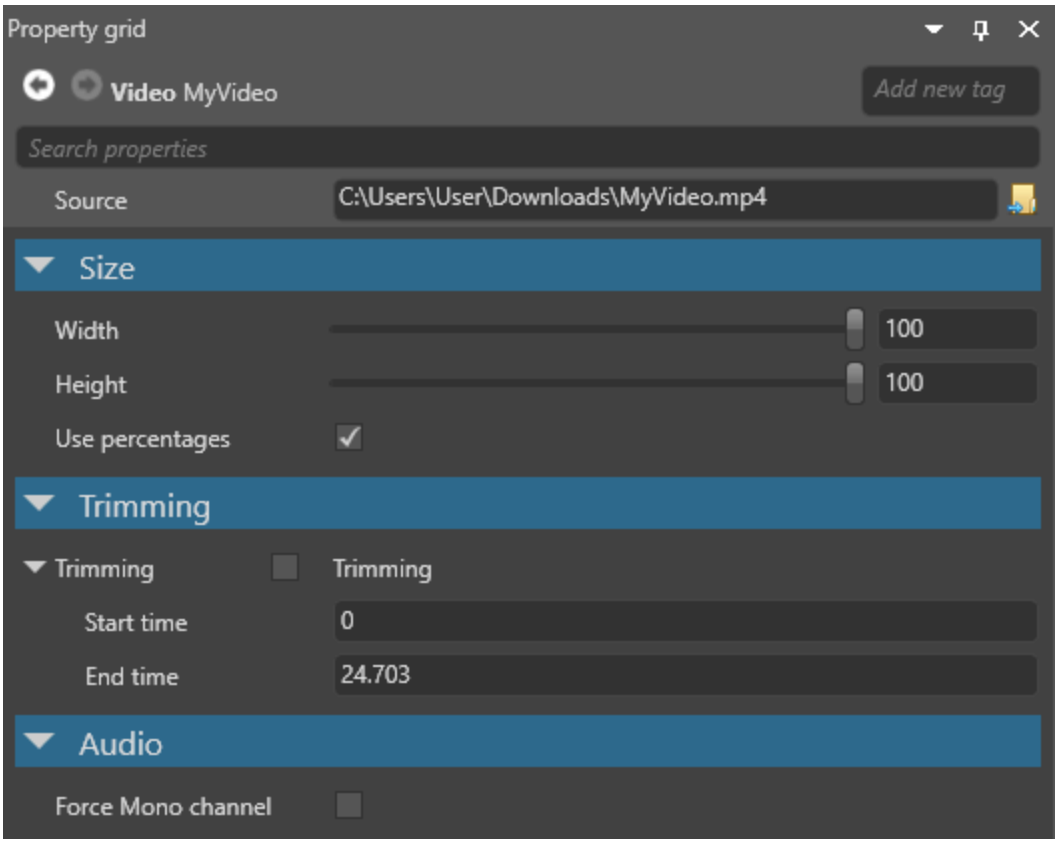
See also

- Video properties

Video properties

Beginner Designer

To view the properties of a video asset, select it in the **Asset View** and use the **Property Grid**.



Property	Description
Width	Resize the video width. The value is in a percentage or actual pixel size depending on whether you select **Use percentages**.
Height	Resize the video height. The value is in a percentage or actual pixel size depending on whether you select **Use percentages**.
Use percentages	Use percentages for the video height and width. If enabled, and the height is set to 100%, Stride displays 100% of the video's actual width. If disabled, the height and width values use pixels, so you can stretch them beyond the video's actual size.
Trimming	Display only the part of the video you define using the **Start** and **End** times
Start time	The time to start playing the video from (in seconds, eg `100.500`)
End time	The time to stop playing the video (in seconds, eg `100.500`)

Property	Description
Force mono channel	Convert video audio to mono. This is useful when you want the video to use spatialized audio.

> **NOTE**
>
> Currently, you can't preview videos in the Asset Preview.

See also

- Set up a video
- Use a video as a skybox

Use a video as a 3D skybox

1. Create a panorama movie.
2. Add a background component.
3. Add a video component to the scene and use the panorama as the video source and the background component as a texture target.

Skybox lights with videos

Currently, you can't use videos for skybox lights.

To create matching lighting for a video background, use a screenshot from the video, or blend between several screenshots.

See also

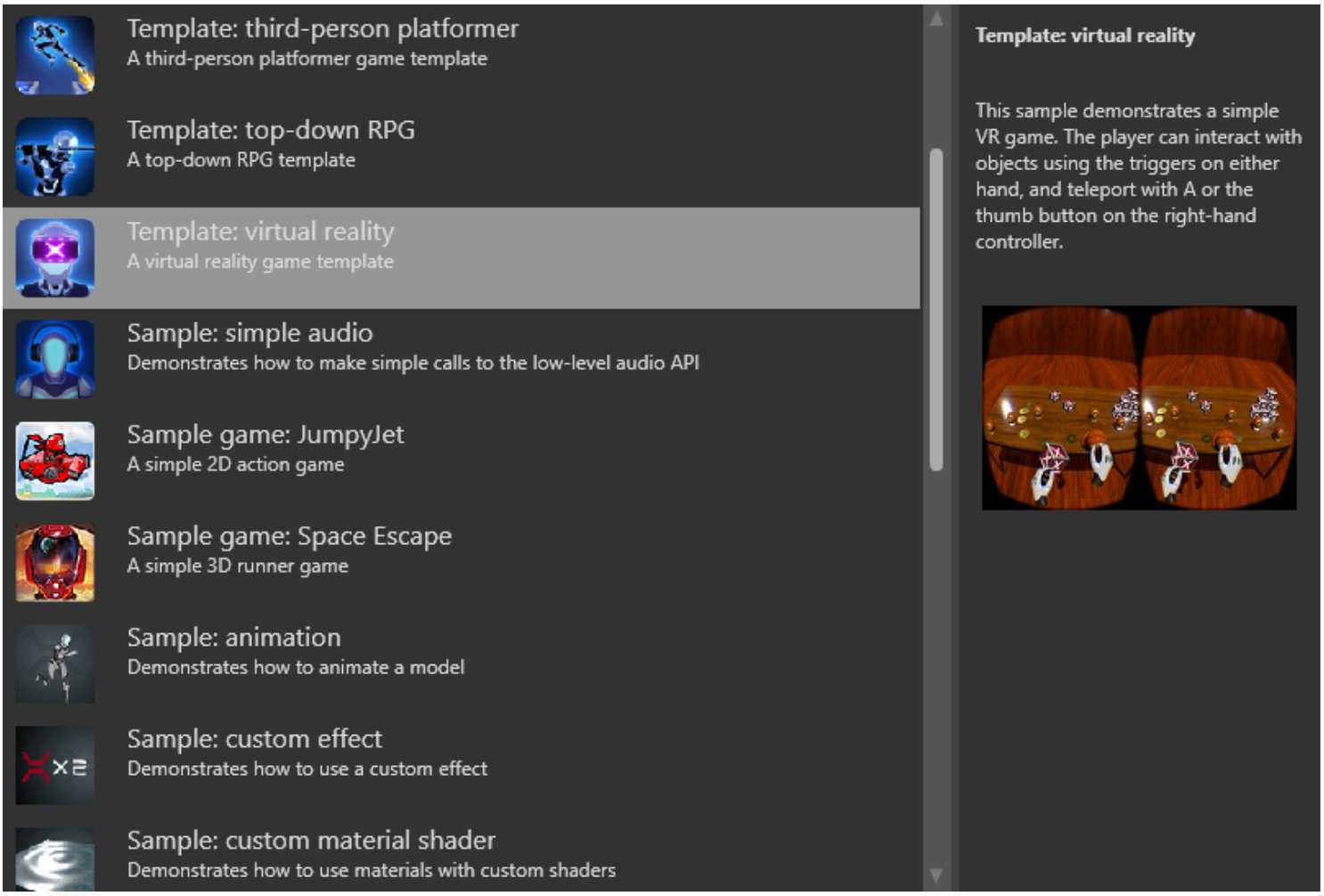
- Skyboxes and backgrounds
- Skybox lights.
- Set up a video
- Video properties

Virtual reality (VR)

Stride currently supports the Oculus Rift and Vive virtual reality (VR) devices.

VR template

Stride includes a VR template you can use to check out VR implementation.



In this section

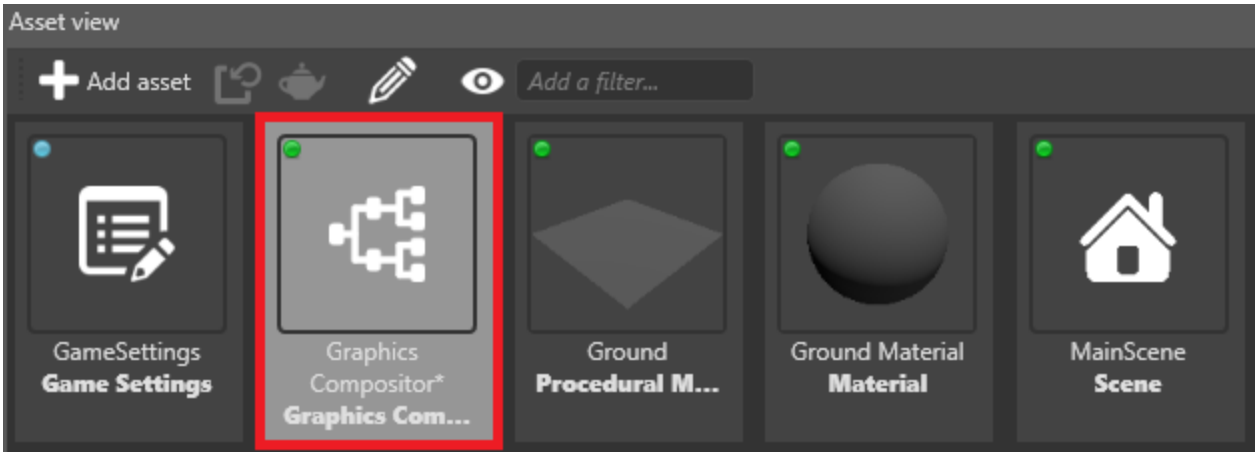
- Enable VR
- Preview a scene in VR
- Overlays
  - Display a UI in an overlay
- VR sickness

Enable VR

Beginner Programmer

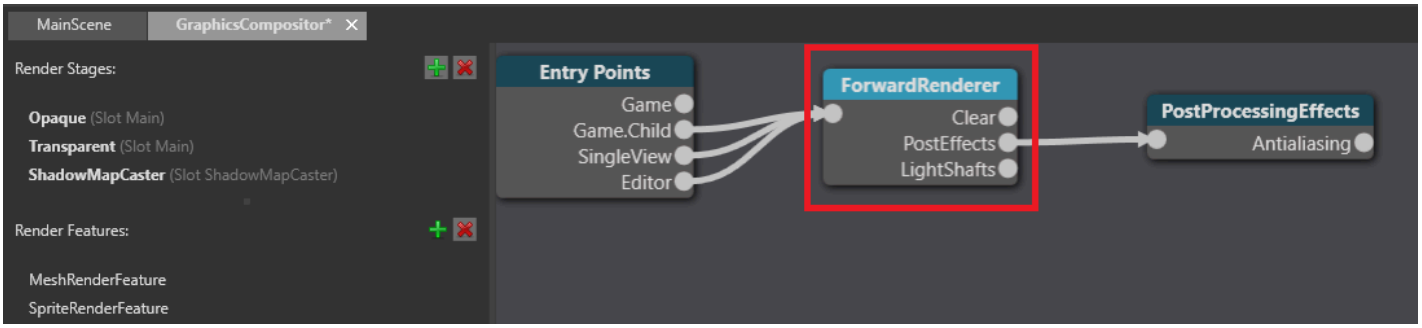
This page explains how to add support for the Oculus Rift and Vive devices to your game. Stride doesn't support other VR devices yet.

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

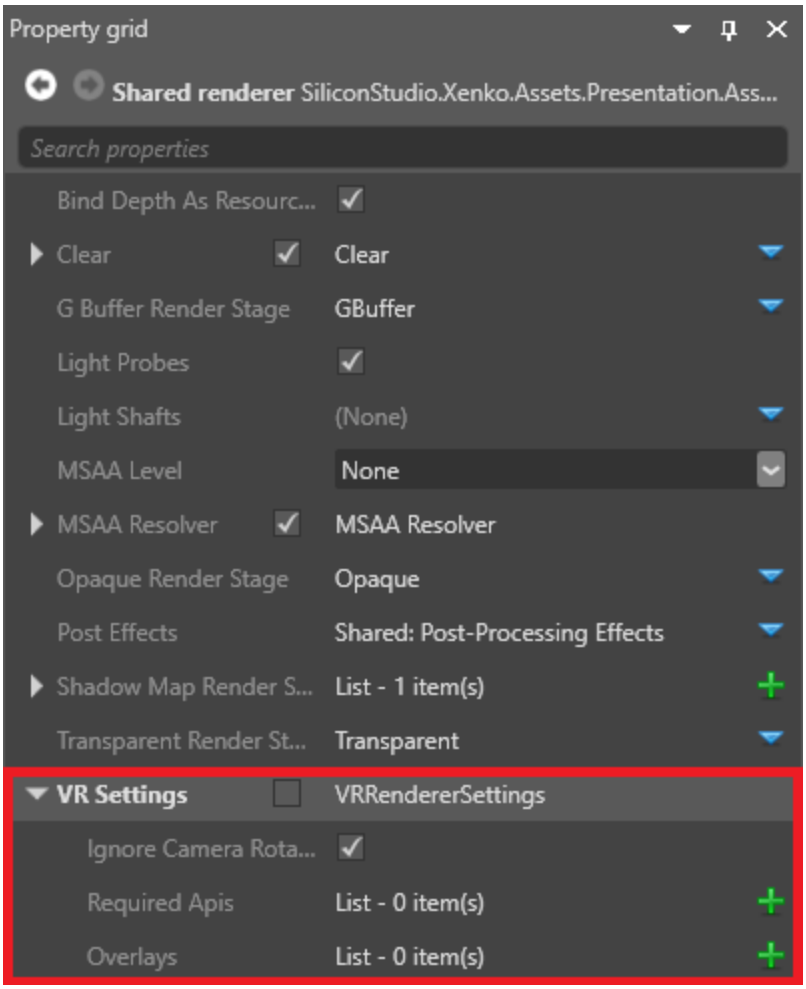


   The graphics compositor editor opens.

2. In the graphics compositor editor, select the **forward renderer** node.

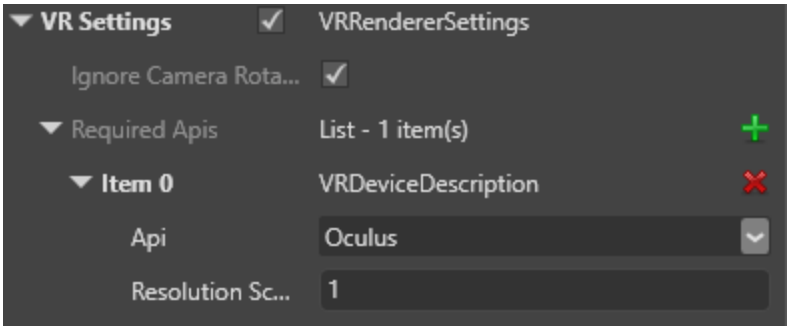


3. In the **Property Grid** (on the right by default), expand **VR Settings**.

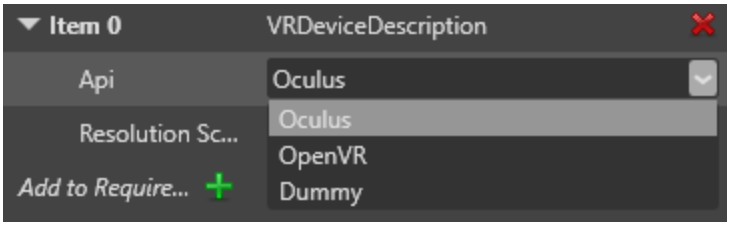


4. Next to **Required APIs**, click (**Add**).

   Game Studio adds a new API to the list.



5. From the **Item** drop-down menu, select a VR API you want your game to support.



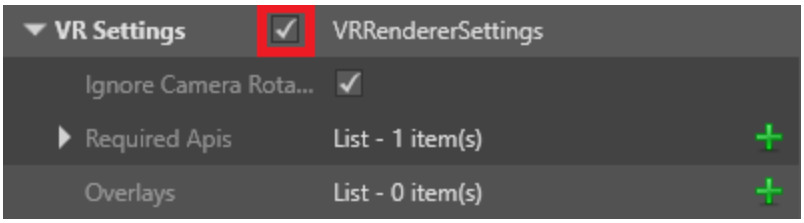
API	Description
Oculus	Supports Oculus Rift devices (best support for Oculus Rift)
OpenVR	Supports Vive and Oculus Rift devices (best support for Vive)
Dummy	Displays the game on the screen with two cameras (one per eye), instead of in the VR device. This is mainly useful for development. To display the dummy view in the Game Studio Scene Editor, make sure the editor is connected to the forward renderer.

6. Repeat steps 4 and 5 to add as many APIs as you need.
7. Make sure the list order is correct. When your game runs, it attempts to use the devices in the list order. For example, if the first item is Dummy, the game uses no VR device. If the last item is Dummy, the game only uses it if there is no VR device available.

   To change the order, change the selected VR device in each item.

   > **TIP**
   > Although the OpenVR API supports both Vive and Oculus Rift devices, the Oculus API provides better support for Oculus Rift. For this reason, we recommend the following list order for most situations:
   >
   > - Item 0: Oculus
   > - Item 1: OpenVR
   >
   > This means your game uses the Oculus API if an Oculus Rift device is connected, and the OpenVR API if another device (eg a Vive) is connected.

8. Enable **VRRendererSettings**.



Your game is now ready to use VR.

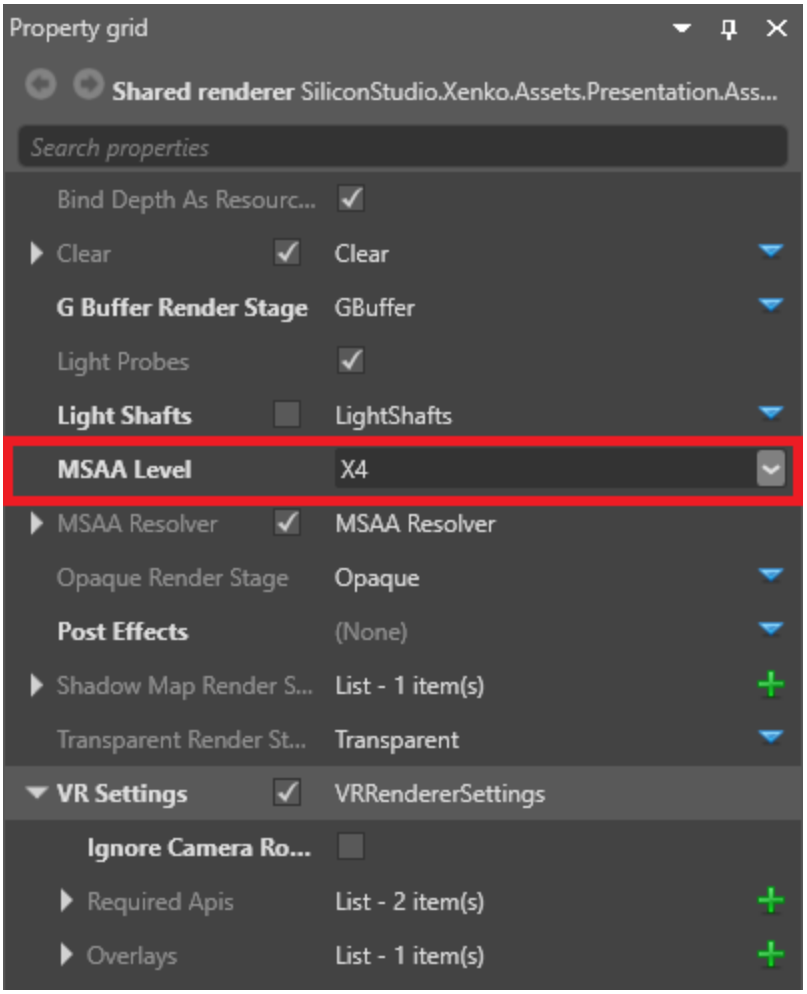
> **NOTE**
>
> After you change APIs, you need to reload the project (**File > Reload project**) for the change to take effect at runtime.

VR properties

Property	Description
Ignore camera rotation	Disable camera movement from inputs other than VR devices, helping to reduce VR sickness
Resolution scale	The resolution of the image displayed in the VR device. Higher resolutions produce better images, but require more GPU.

Multisample anti-aliasing

As aliasing artifacts are more obvious in VR, we recommend you enable **MSAA** (multisample anti-aliasing) in the forward renderer properties (above the VR settings).



> **NOTE**
>
> MSAA isn't supported for Direct3D 11 or lower.

Disable screen synchronization

For best performance, VR games need to run at 90FPS. This means you have to turn off synchronization with your monitor.

For now, this is done in a script. We recommend you use `IsDrawDesynchronized` in `IsFixedTimeStep`.

```
using System;
using Stride.Engine;

namespace VRSandbox
{
    class VRSandboxApp
    {
        static void Main(string[] args)
        {
            using (var game = new Game())
            {
                //VR needs to run at 90 fps, vsync must be disabled, draw must be
not synchronized
                //You might want to set physics time step to 90 fps as well if you use
character controller with unregular movements, but please avoid that! use Kinematic
rigidbodies when possible.
                game.IsFixedTimeStep = true;
                game.IsDrawDesynchronized = true;
                game.GraphicsDeviceManager.SynchronizeWithVerticalRetrace = false;
                game.TargetElapsedTime = TimeSpan.FromSeconds(1 / 90.0f);
                game.Run();
            }
        }
    }
}
```

See also

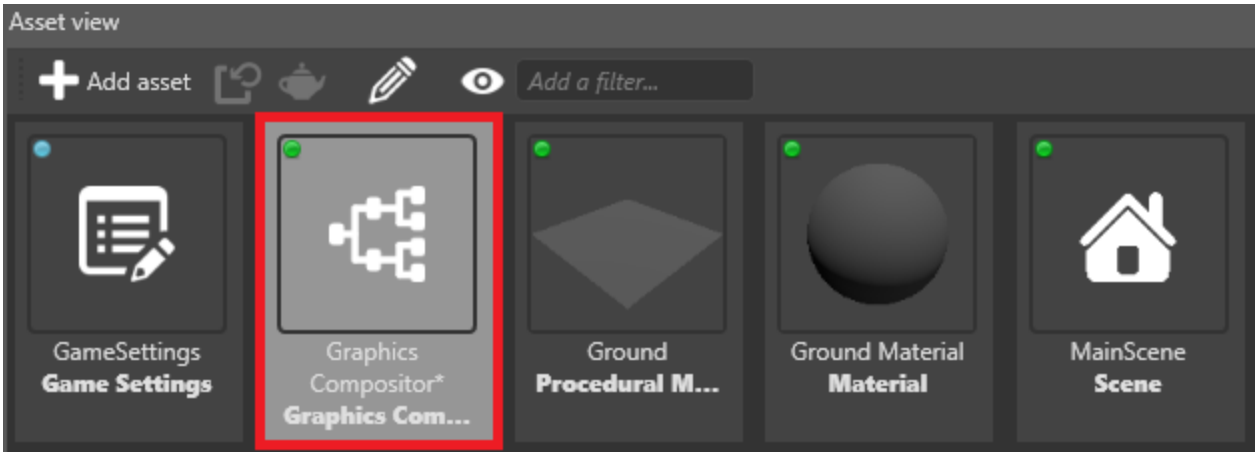
- VR sickness
- Graphics compositor

Preview a scene in VR

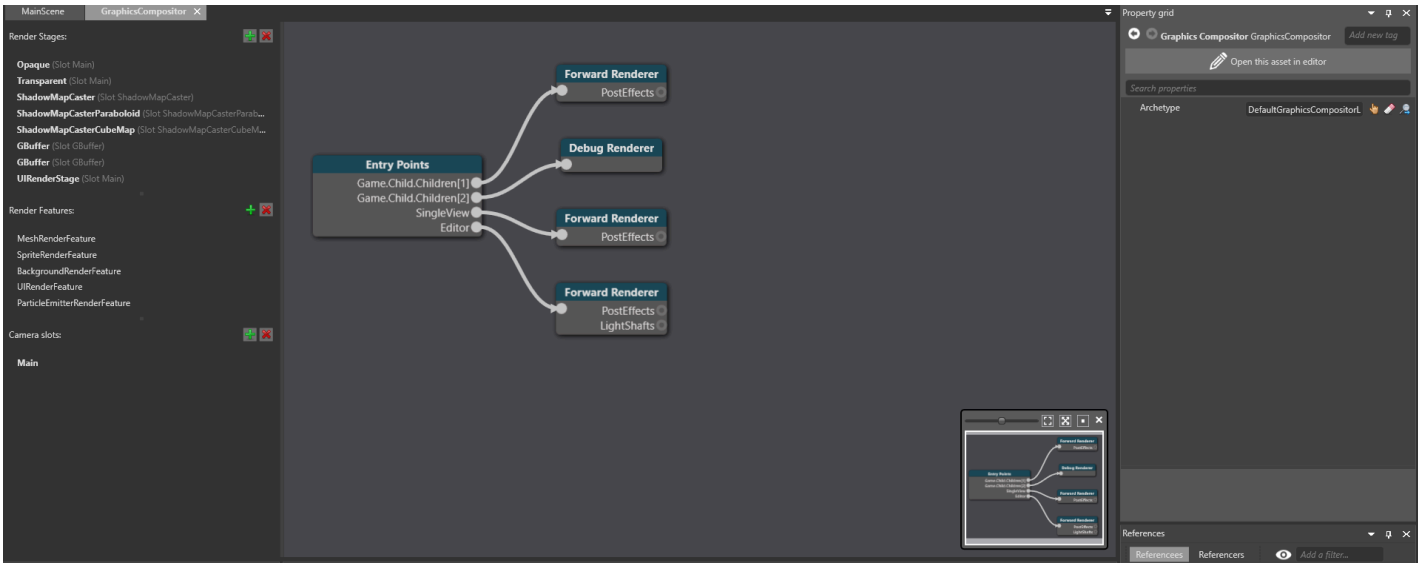
To preview your scene in your VR device, connect the editor to a VR-enabled renderer.

To do this:

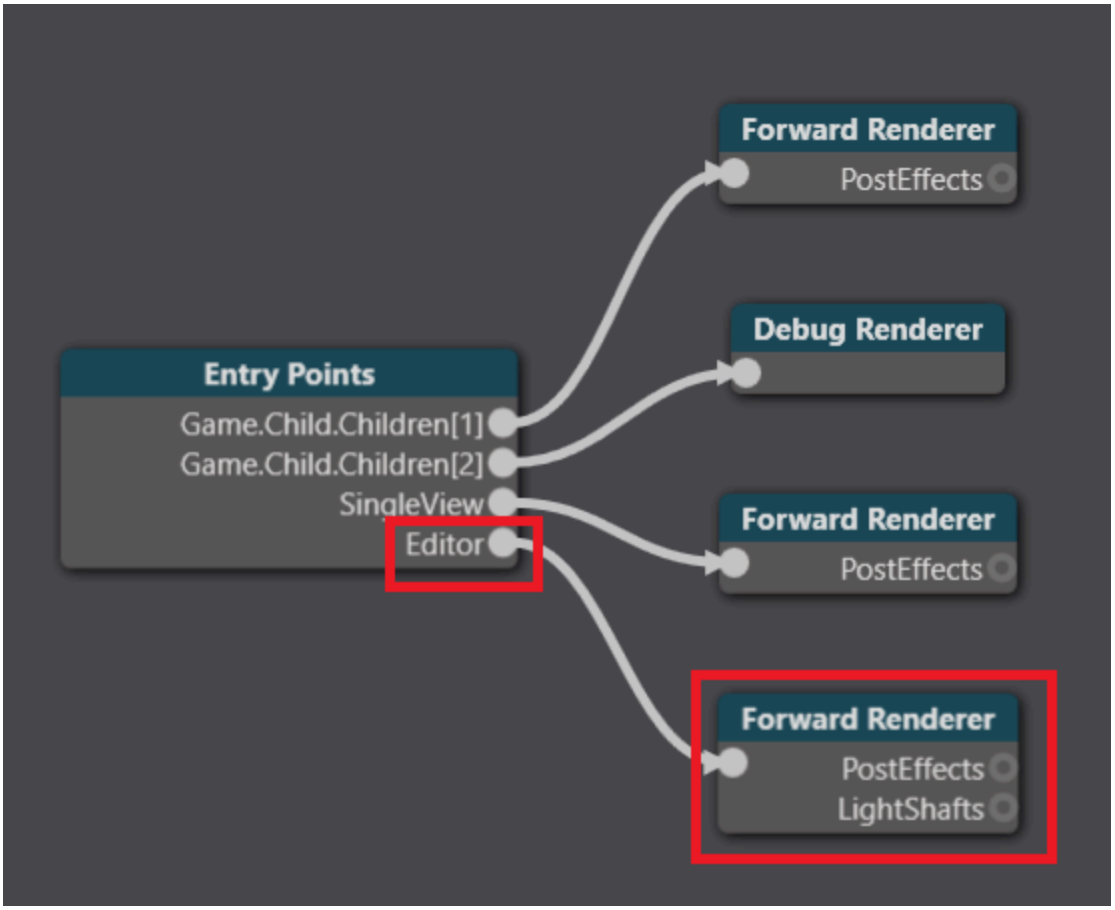
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



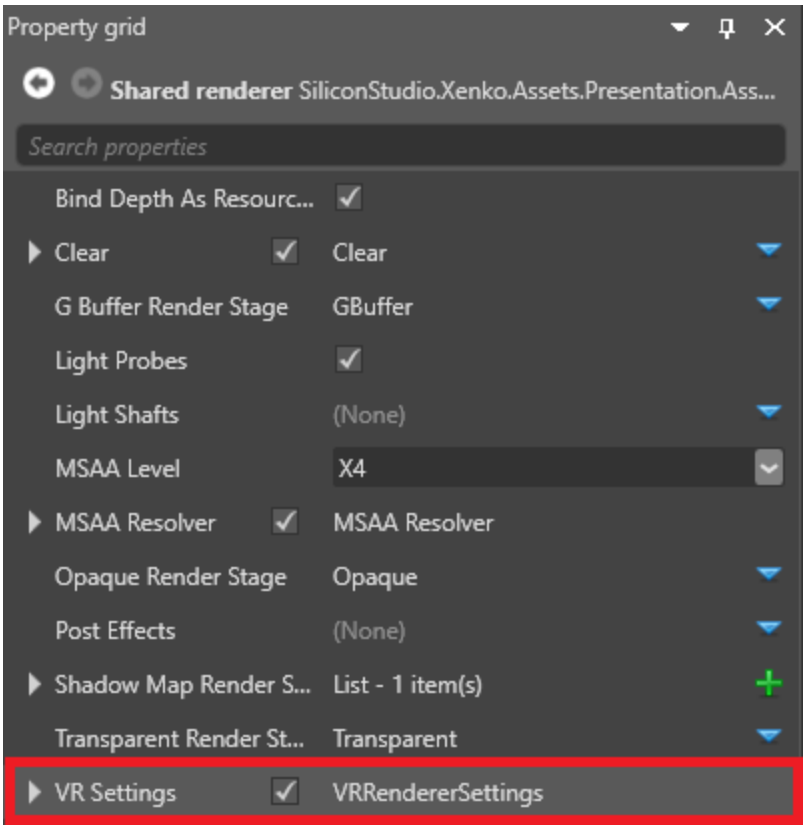
   The graphics compositor editor opens.



2. Select the **forward render node** connected to the editor node. For example, in the screenshot below (taken from the Stride VR sample project), the editor is connected to the lower forward renderer node.



3. With the forward renderer node selected, in the **Property Grid**, enable **VRRendererSettings**.



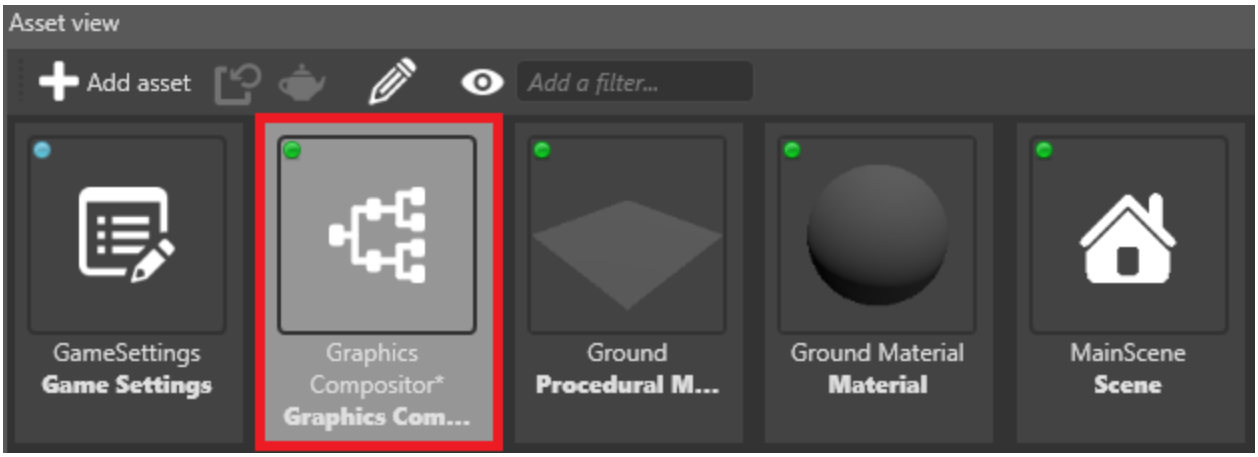
Your VR device displays the scene preview. To display the scene on your monitor instead, disable **VRRendererSettings**.

Create a separate renderer to preview scenes in VR

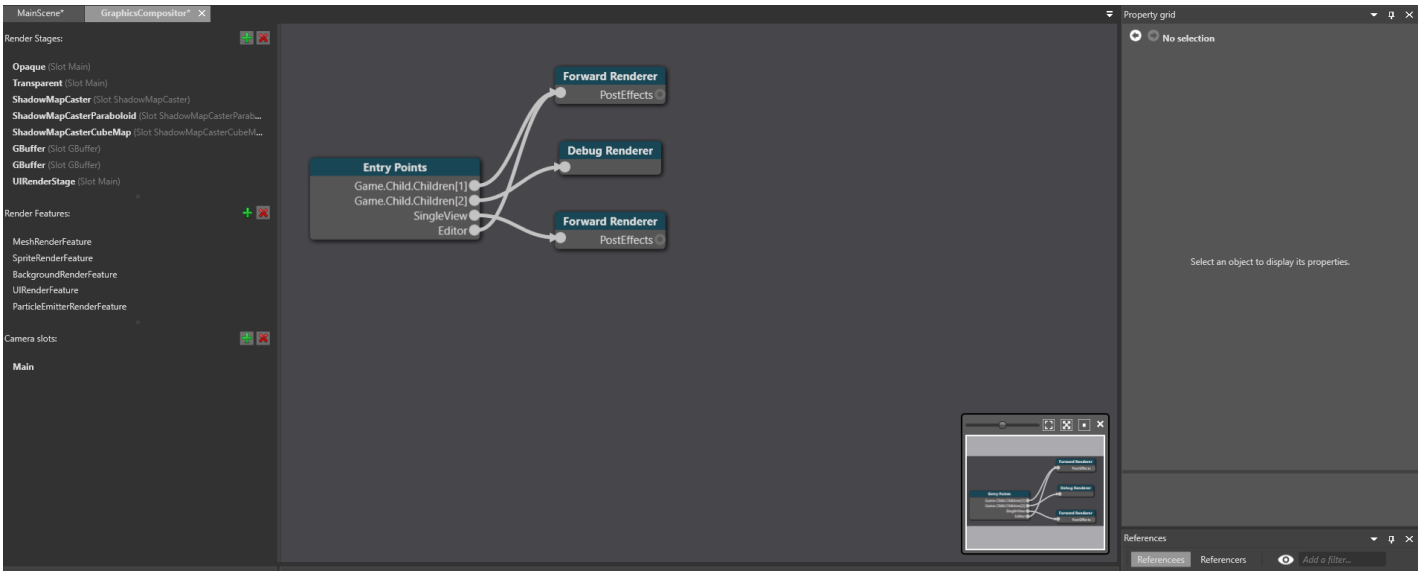
If your editor and game nodes are connected to the same forward renderer, you might want to create a separate renderer dedicated to the editor. This lets you easily switch between previewing the scene in your VR device and on your monitor.

> **NOTE**
>
> If your editor and game nodes already use separate renderers (as in the VR sample project), you don't need to follow these instructions.

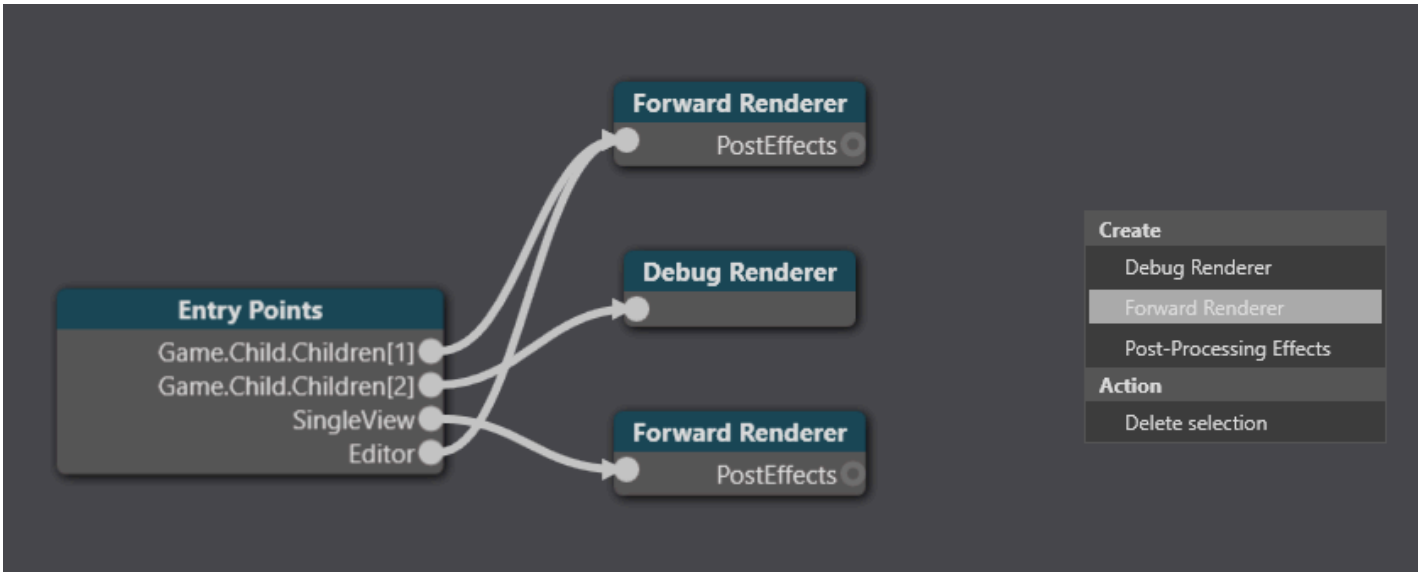
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



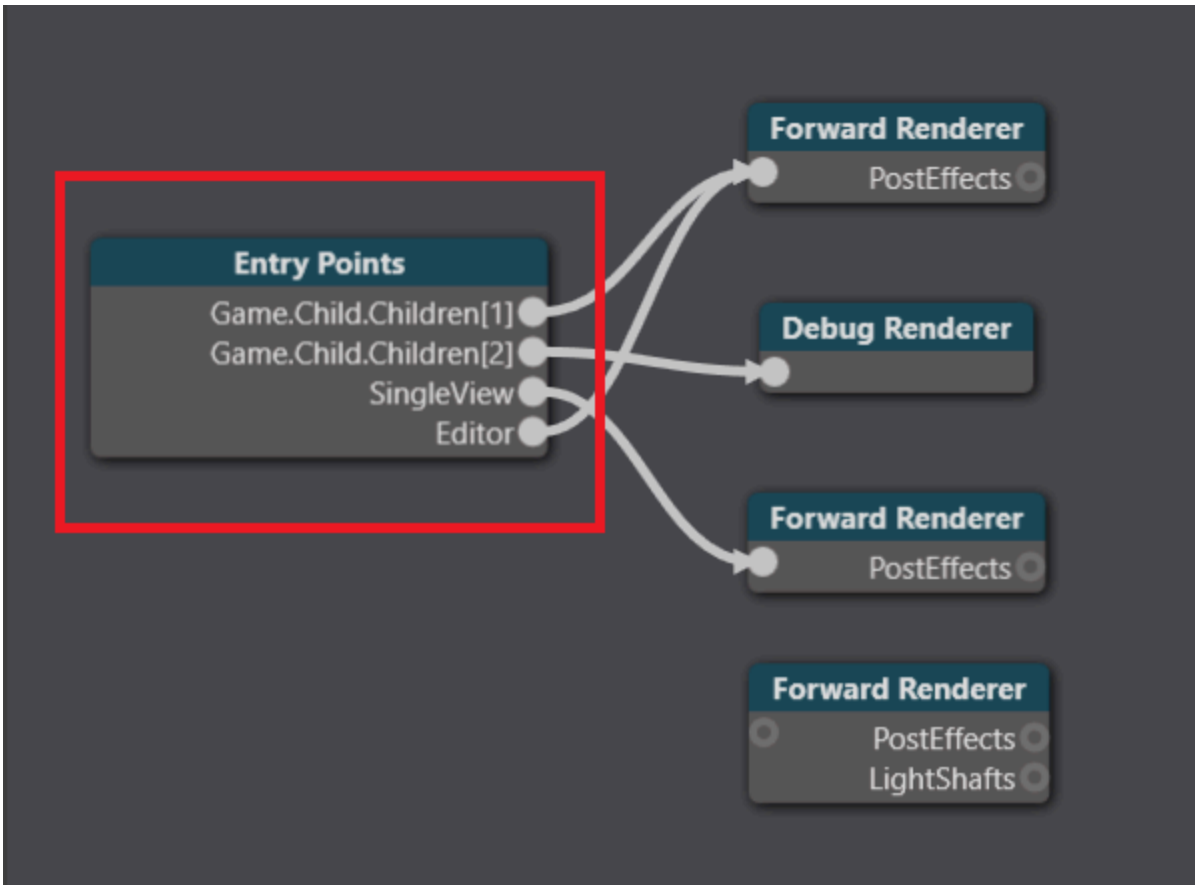
The graphics compositor editor opens.



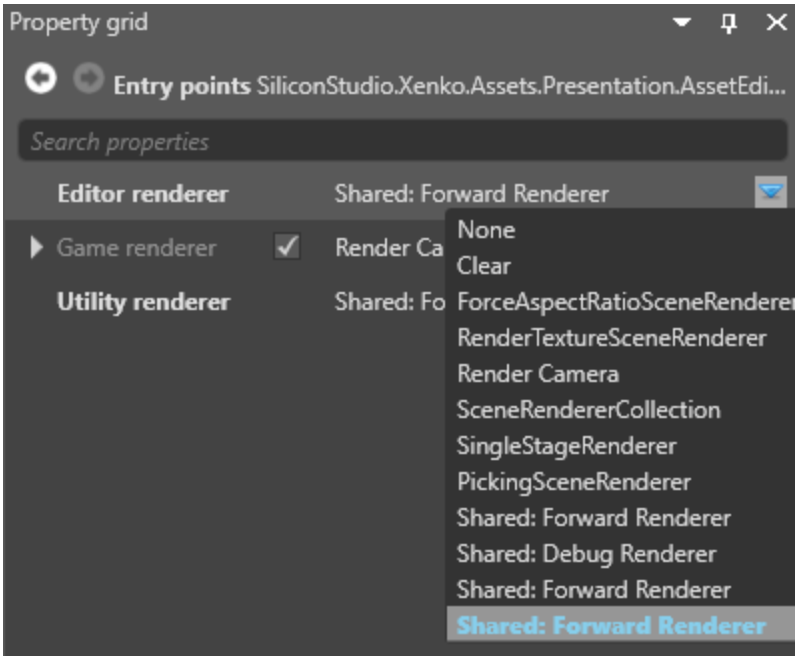
2. Create a new forward renderer node. To do this, right-click the game compositor editor and select **Create > Forward renderer**.



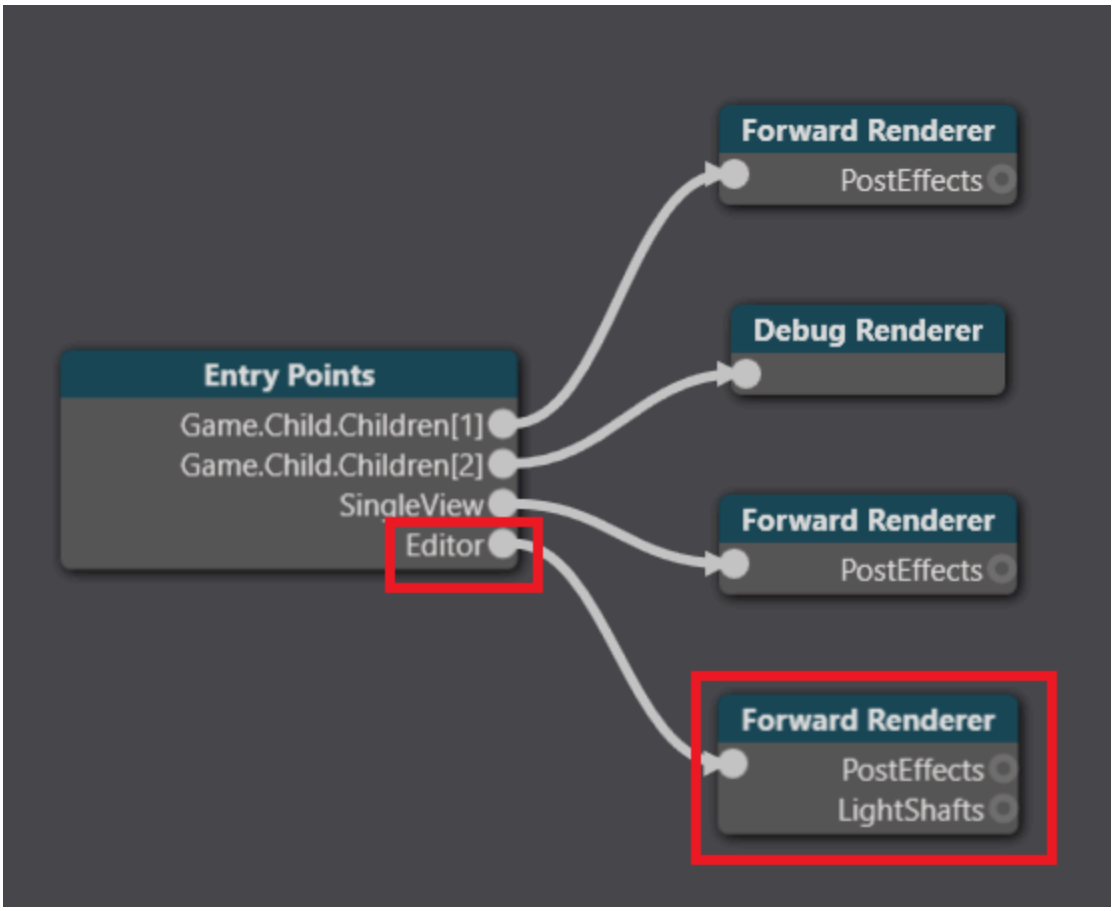
3. Select the **Entry points** node.



4. In the **Property Grid**, next to **Editor renderer**, select the forward renderer you created.

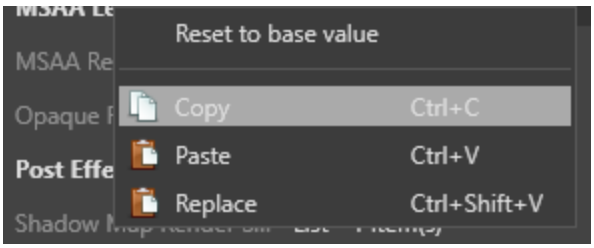


Stride links the editor to the forward renderer node.



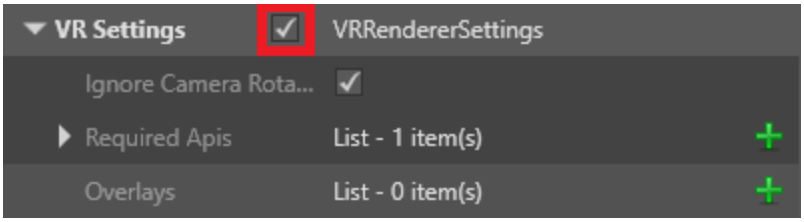
5. Set the properties of the new forward renderer so they're identical to the forward renderer you use to run the game in VR, including the VR settings.

> **TIP**
>
> You can right-click a property to copy or paste it.



> **NOTE**
>
> Make sure the forward renderer has VR enabled. For instructions, see Enable VR.

Stride displays the scene preview in your VR device. To display the scene on your monitor instead, disable **VRRendererSettings** in the properties of the new forward renderer.



See also

- Enable VR
- Graphics compositor

Overlays

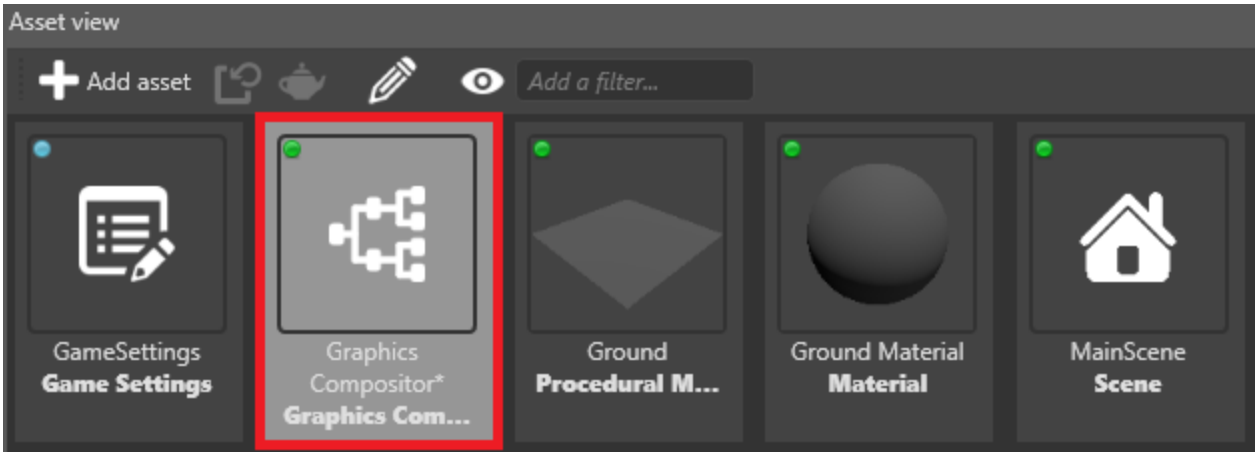
In VR games, you can display textures (including render textures) as overlays that appear to float in front of the player. This is especially useful for UIs.

> **NOTE**
>
> You can't see overlays when you don't run your game in your VR device. This is because the VR device itself creates the overlay.

This page explains how to add an overlay. To display a **UI** in an overlay, you need to render the UI to a render texture, and display the render texture in the overlay. For instructions, see Display a UI in an overlay.

Add an overlay

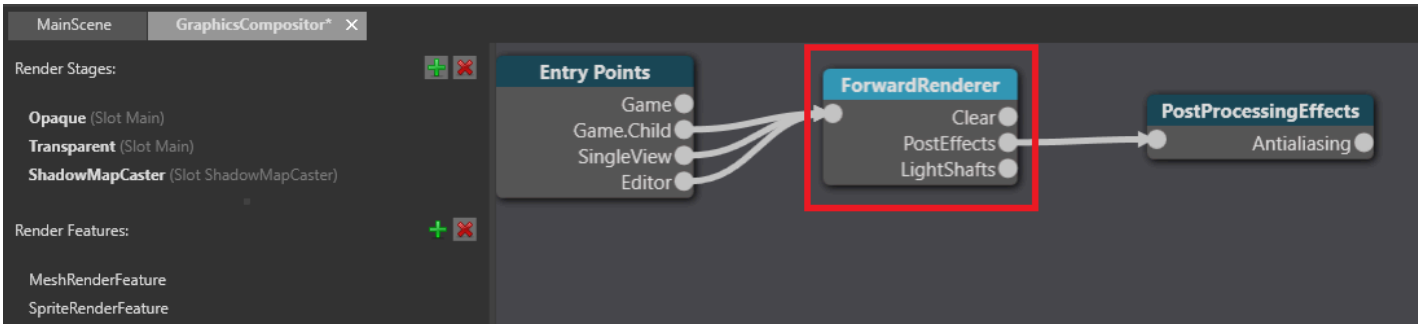
1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.



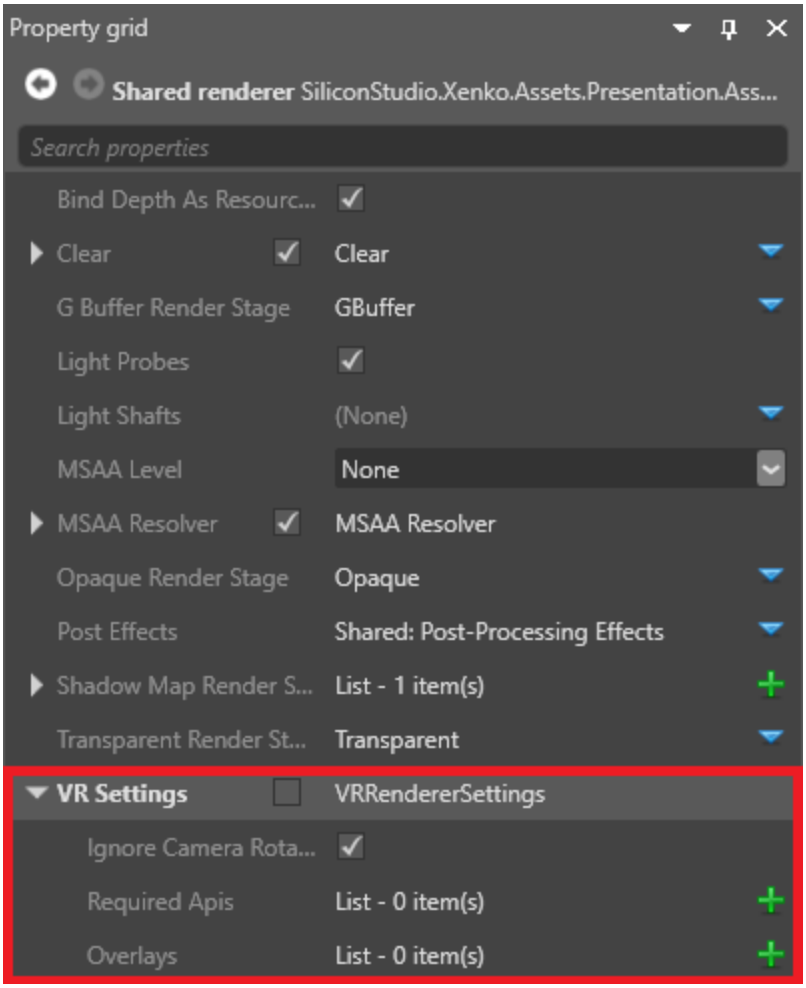
   The graphics compositor editor opens.

   For more information about the graphics compositor, see the Graphics compositor page.

2. In the graphics compositor editor, select the **forward renderer** node.

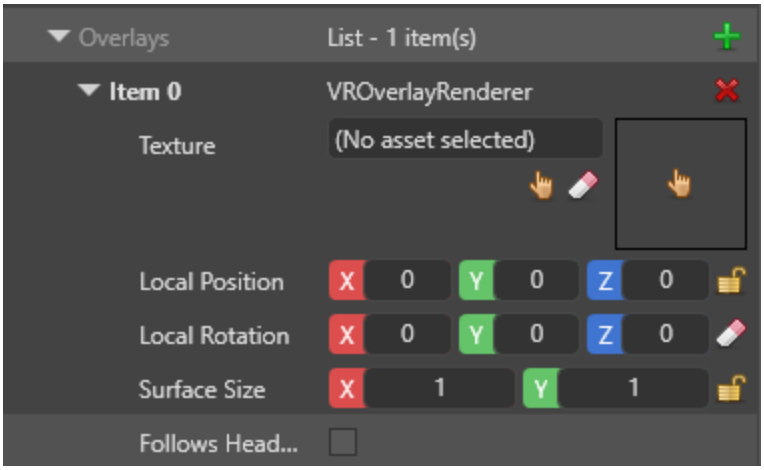


3. In the **Property Grid** (on the right by default), expand **VR Settings**.



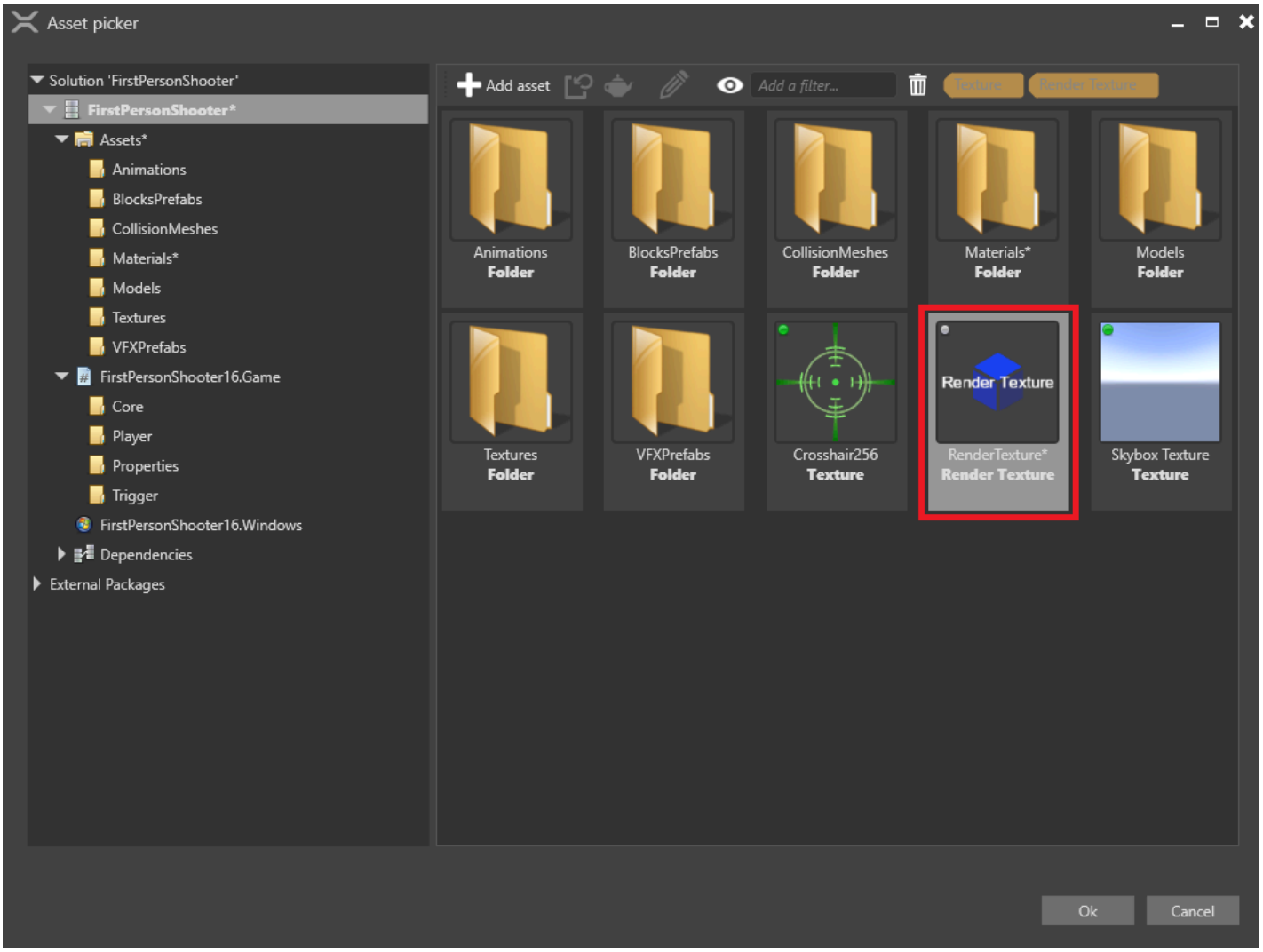
4. Next to **Overlays**, click (**Add**).

   Game Studio adds a new overlay to the list.



5. Next to **Texture**, click (**Select an asset**).

   The **Select an asset** window opens.



6. Select the texture you want to display in the overlay and click **OK**.

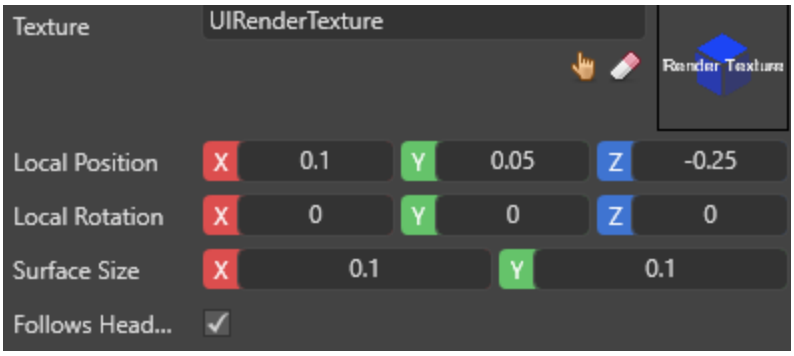
Your game is now ready to render the UI to an overlay in your VR device.

Multiple overlays

You can add as many overlays as you need. To add another overlay, click **Add to overlays** and follow the instructions above from step 4.

> **NOTE**
>
> If overlays overlap in the user view, overlays first in the list appear on top.

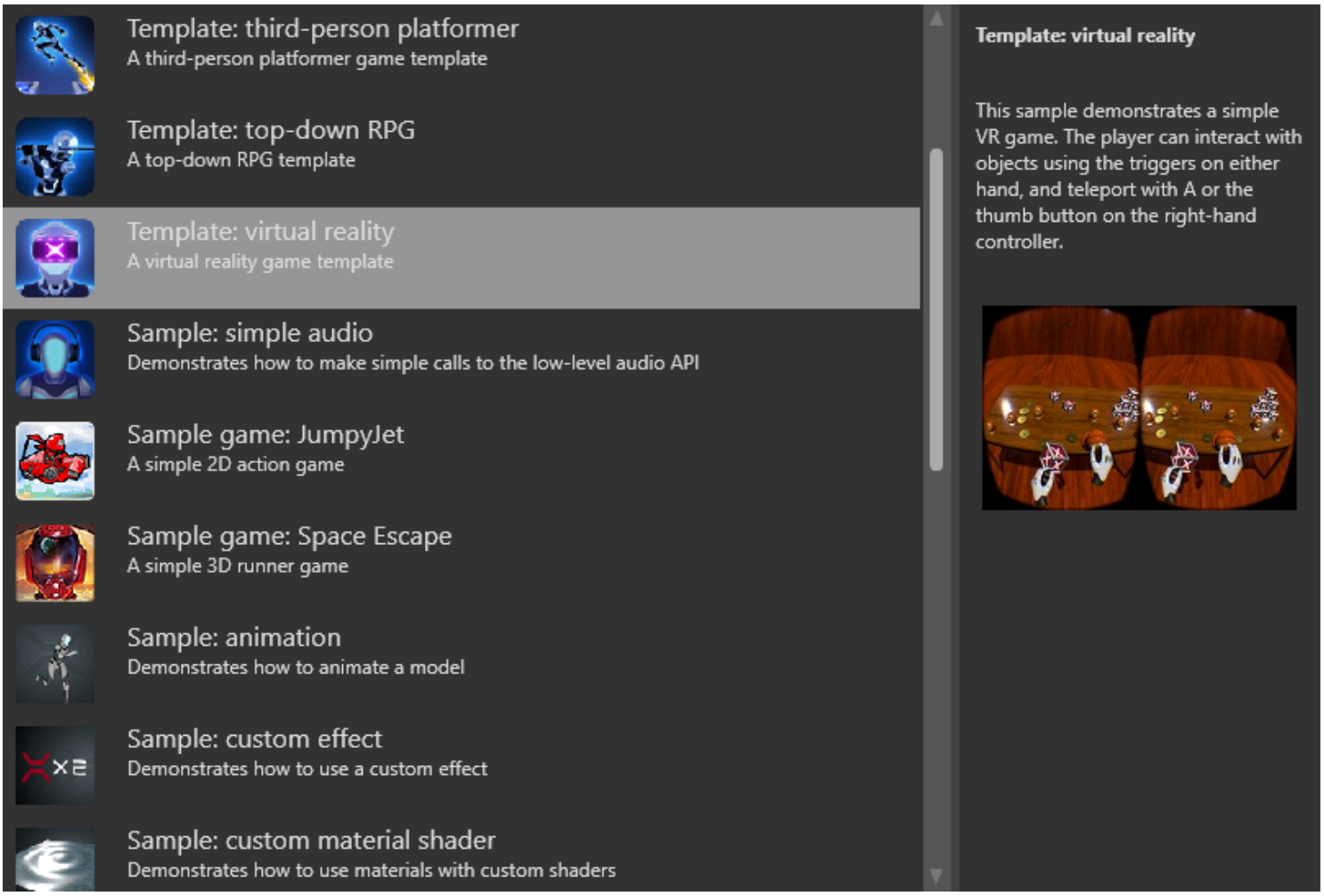
Overlay properties



Property	Description
Texture	The texture displayed in the overlay
Local position	The position of the overlay relative to the user
Local rotation	The rotation of the overlay relative to the user
Surface size	The size of the overlay in world units
Follows head	Follow the user's head so the overlay is always in front of their view

VR template

For an example of a UI overlay implemented in a VR game, see the VR template included with Stride.



See also

- Display a UI in an overlay
- Render textures
- Graphics compositor

Display a UI in an overlay

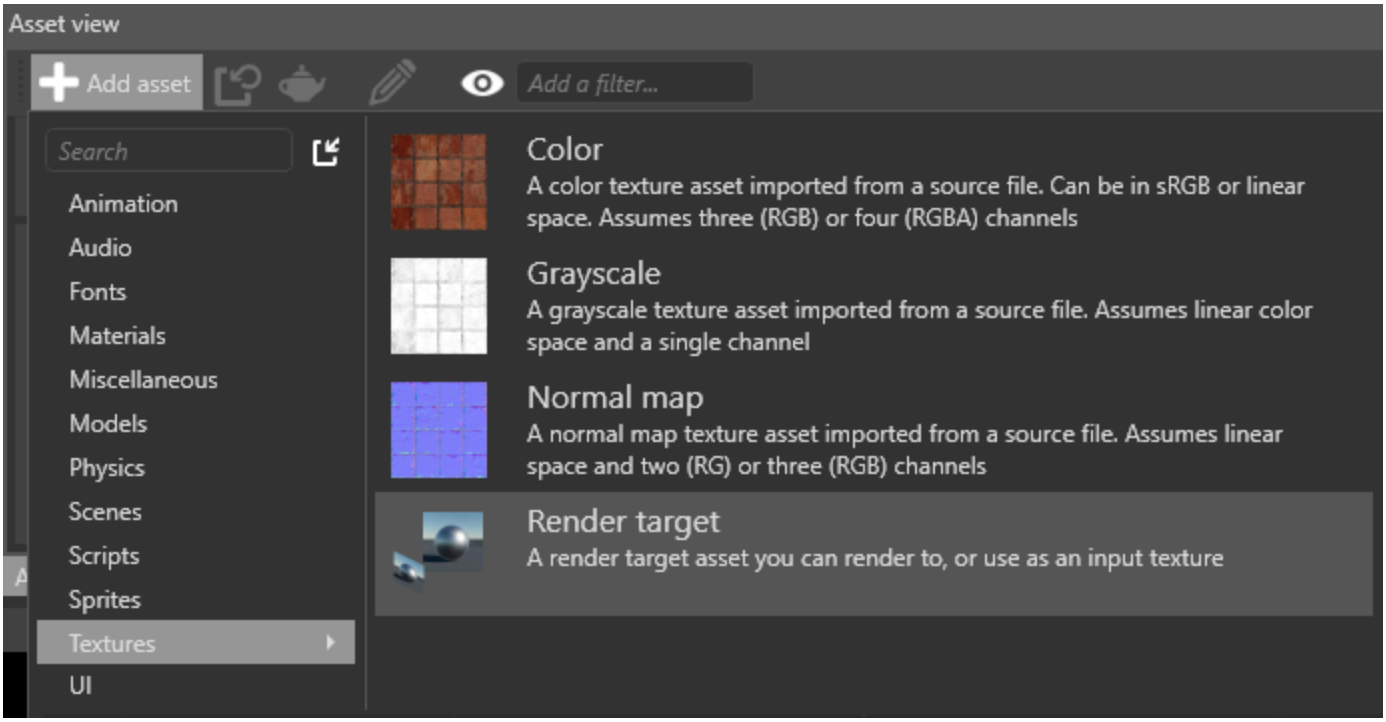
This page explains how to render a UI to a texture, then display it as an overlay.

These instructions assume you already have a UI that you want to display in the overlay. For information about creating UIs, see the UI section.

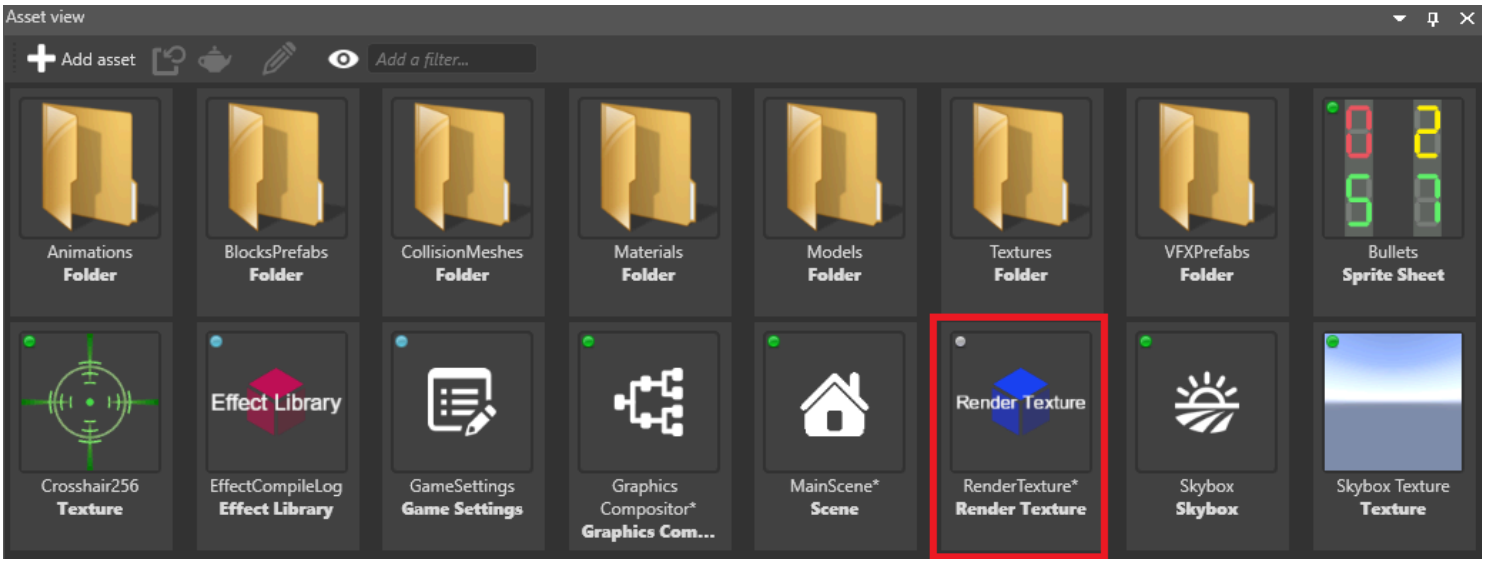
> **NOTE**
>
> You can't see overlays when you don't run your game in your VR device. This is because the VR device itself creates the overlay, not other hardware.

1. Create a render target texture

In the **Asset View**, click **Add asset** and select **Texture** > **Render target**.



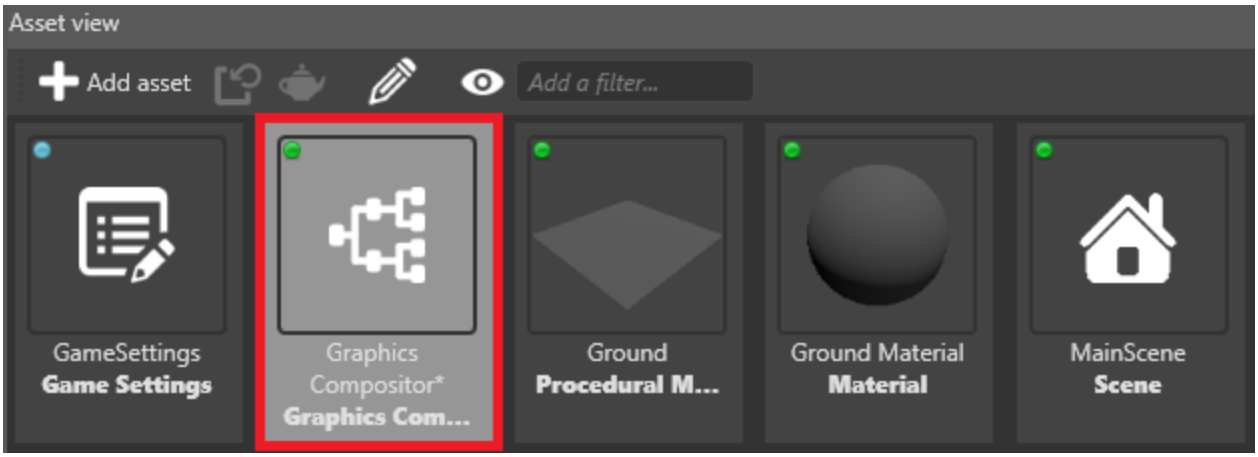
Game Studio adds a **render target** texture to your project assets.



In the following steps, we'll render the UI to this texture, then display it in the overlay.

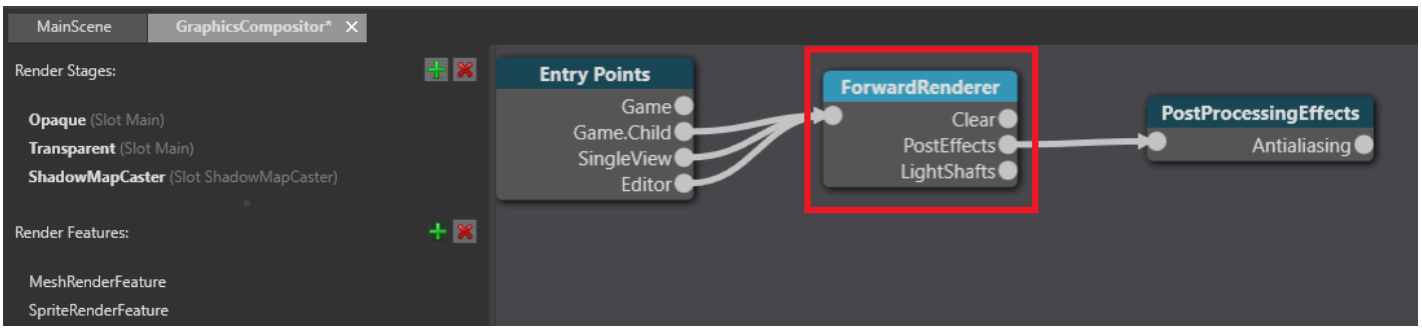
2. Add a VR overlay

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

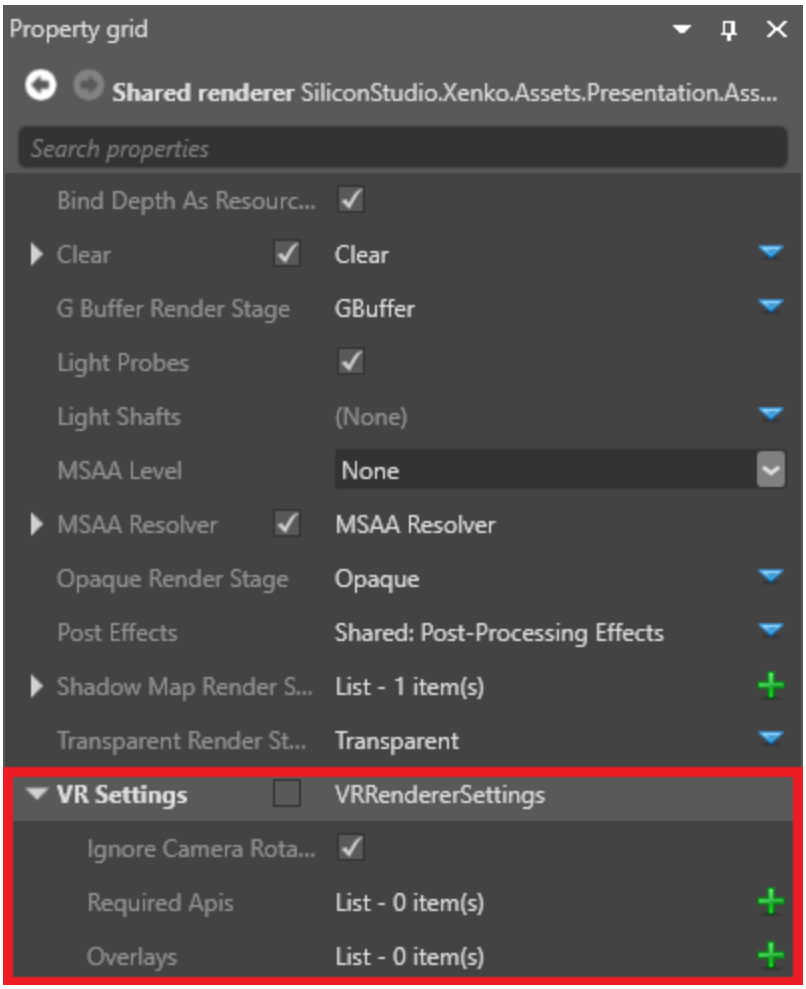


   The graphics compositor editor opens. For more information about the graphics compositor, see the Graphics compositor page.

2. In the graphics compositor editor, select the **forward renderer** node.

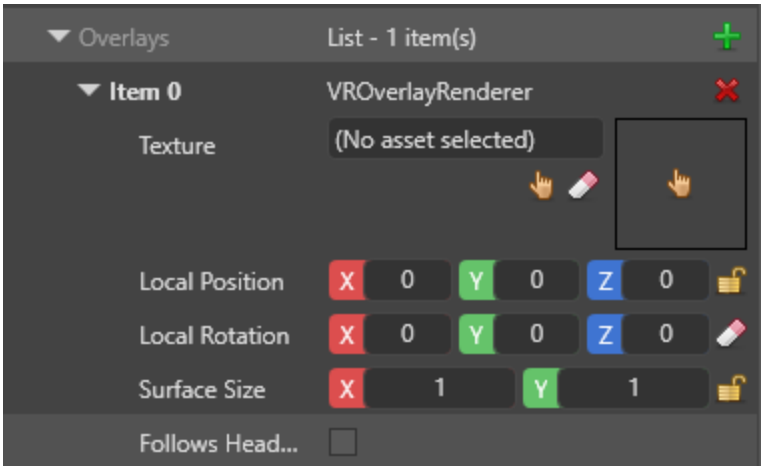


3. In the **Property Grid** (on the right by default), expand **VR Settings**.



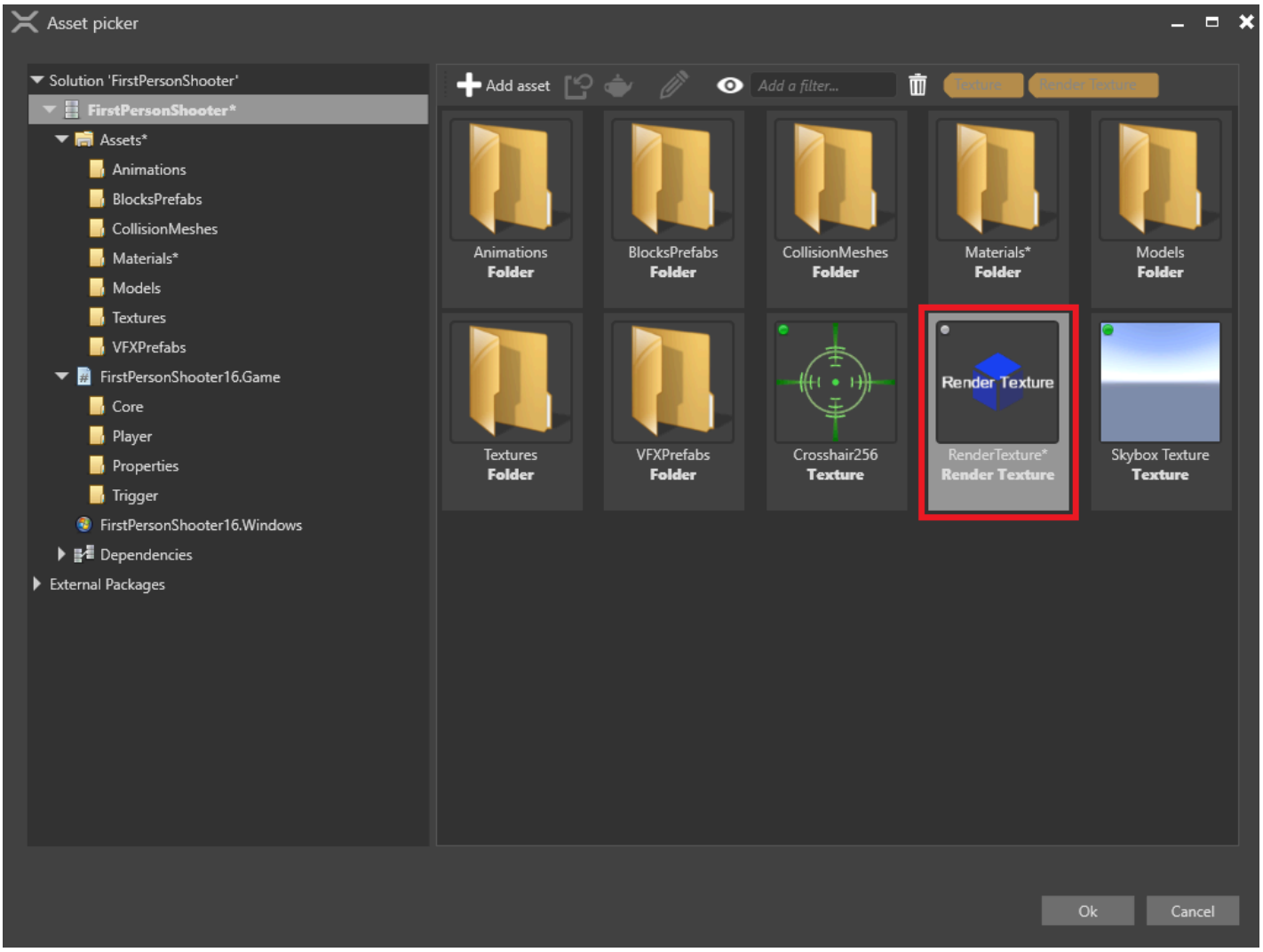
4. Next to **Overlays**, click (**Add**).

   Game Studio adds a new overlay to the list.



5. Next to **Texture**, click (**Select an asset**).

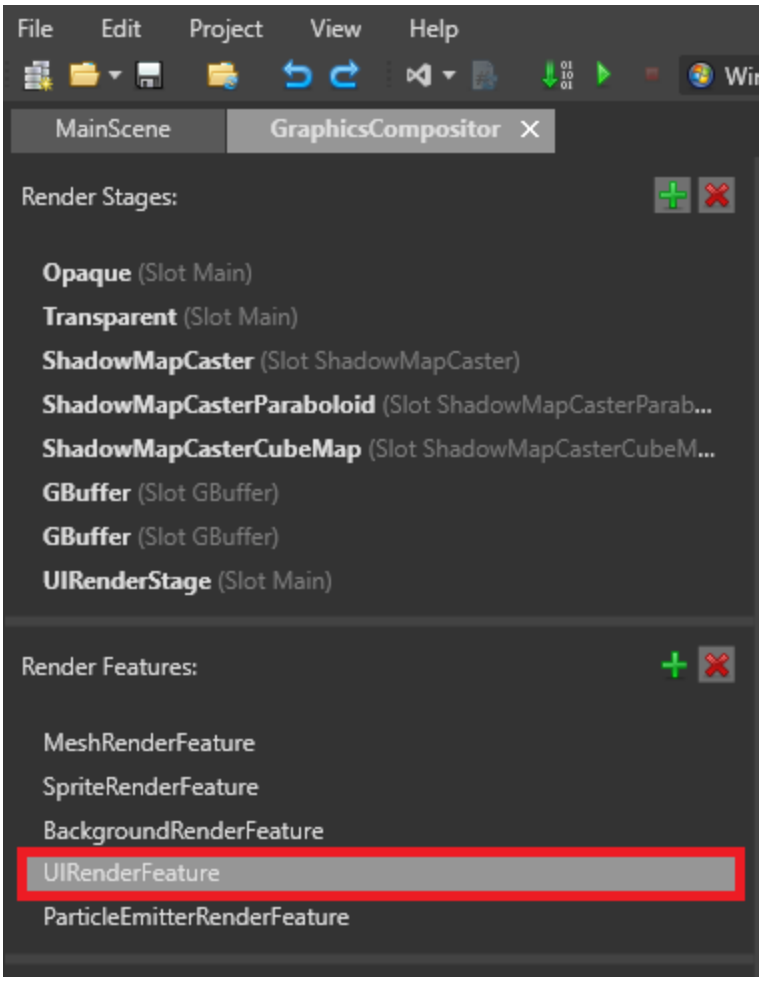
   The **Select an asset** window opens.



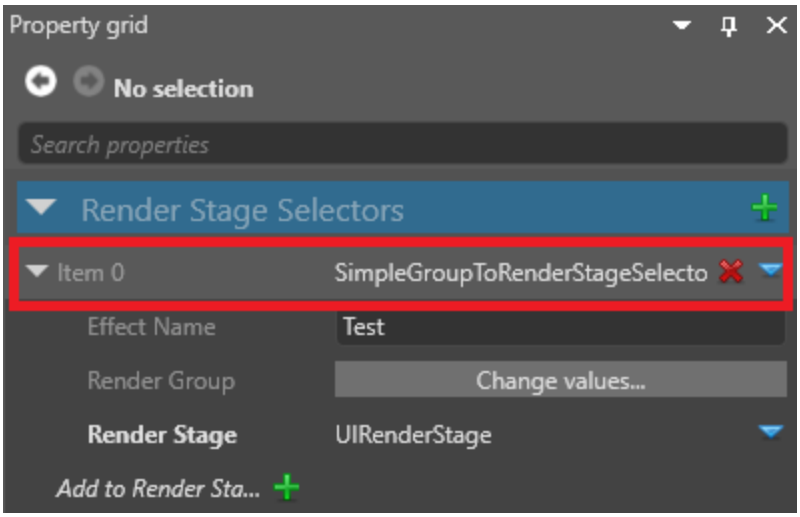
6. Select the **render texture** you created and click **OK**.

3. Set up the UI render feature

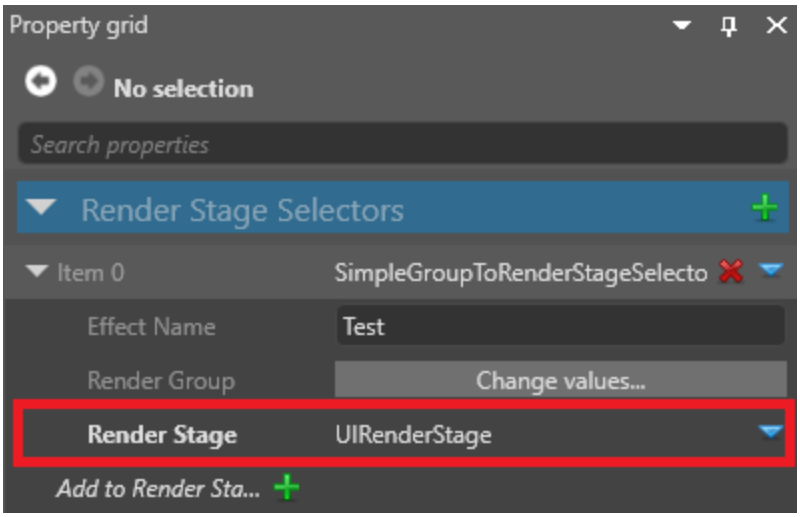
1. In the graphics compositor editor, on the left, under **Render Features**, select the **UIRenderFeature**.



2. In the Property Grid, make sure **SimpleGroupToRenderStageSelector** is selected.



3. Under **Render Stage**, make sure **UIRenderStage** is selected.



This makes sure the UI is rendered in the UI render stage, which we'll use in the next step.

4. Set up the renderers

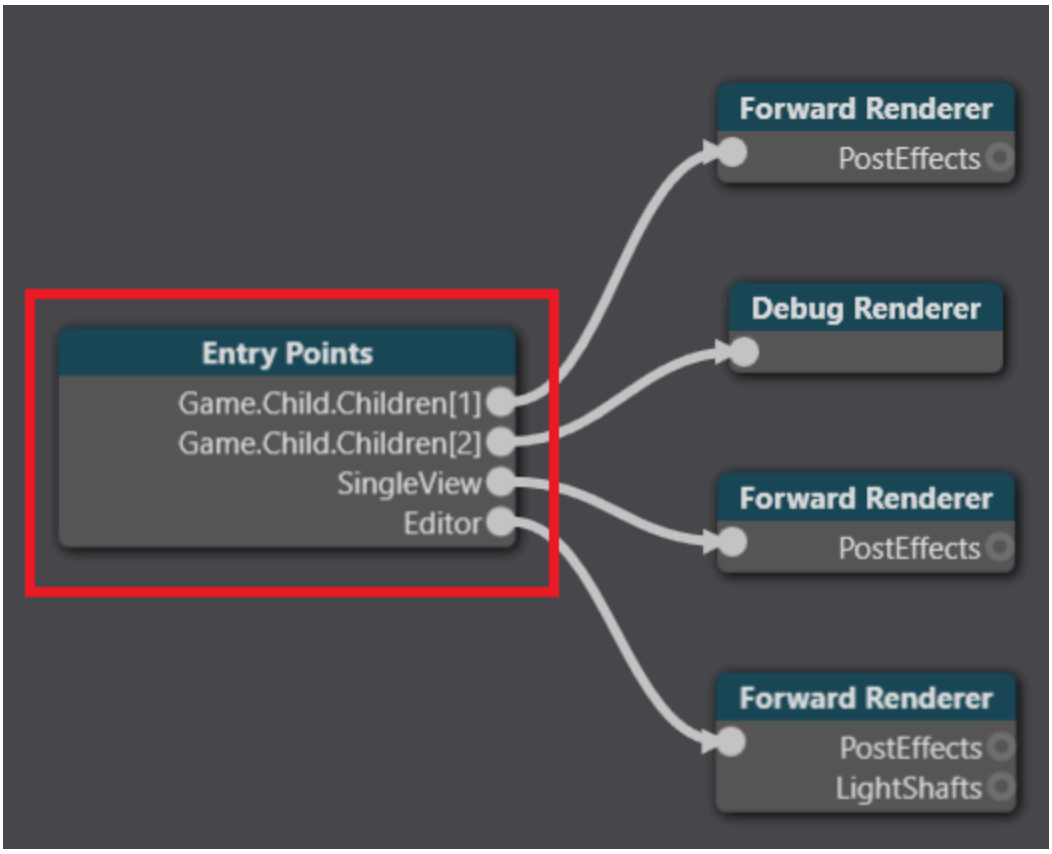
To display an overlay, you need at least two renderers:

- one to render your main camera
- one to render the UI to the overlay

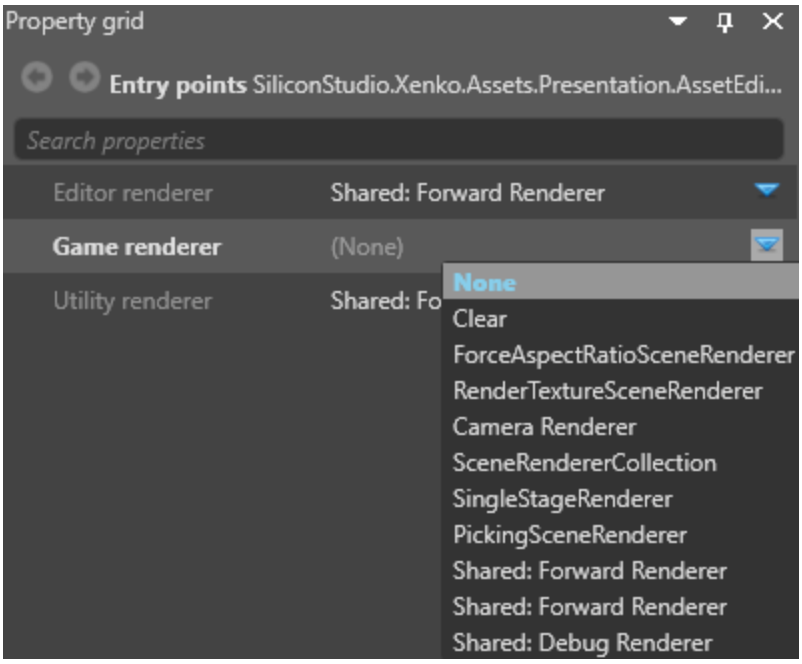
This page describes the simplest way to do this from scratch, using two cameras and two renderers. Depending on your pipeline, you might need to create a different setup.

> **WARNING**
>
> These instructions involve deleting your existing renderers for the game entry point. You might want to make a backup of your project in case you want to restore your pipeline afterwards.

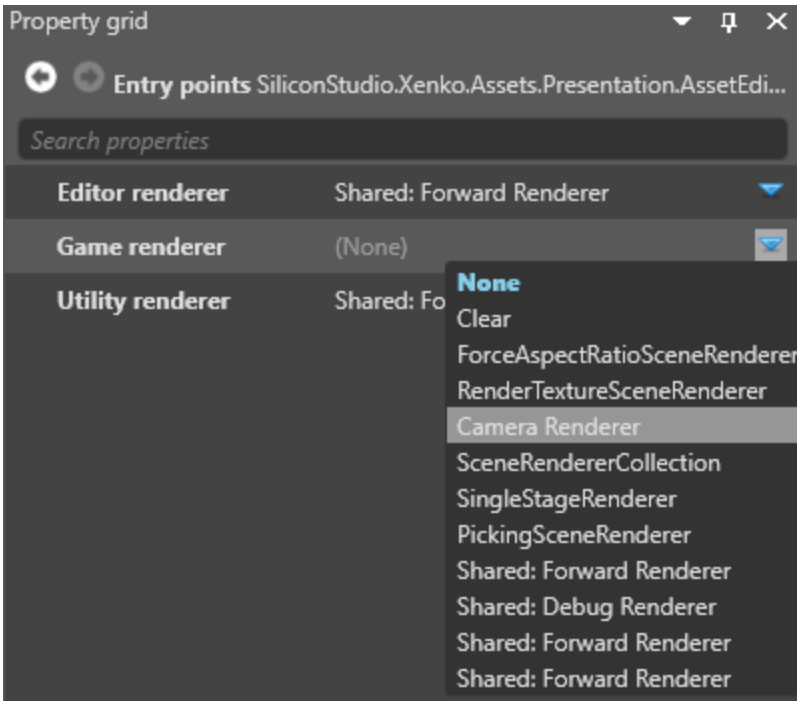
1. In the graphics compositor editor, select the **Entry points** node.



2. In the **Property Grid** on the right, next to **Game renderer**, click (**Replace**) and select **None** to delete your existing renderers.



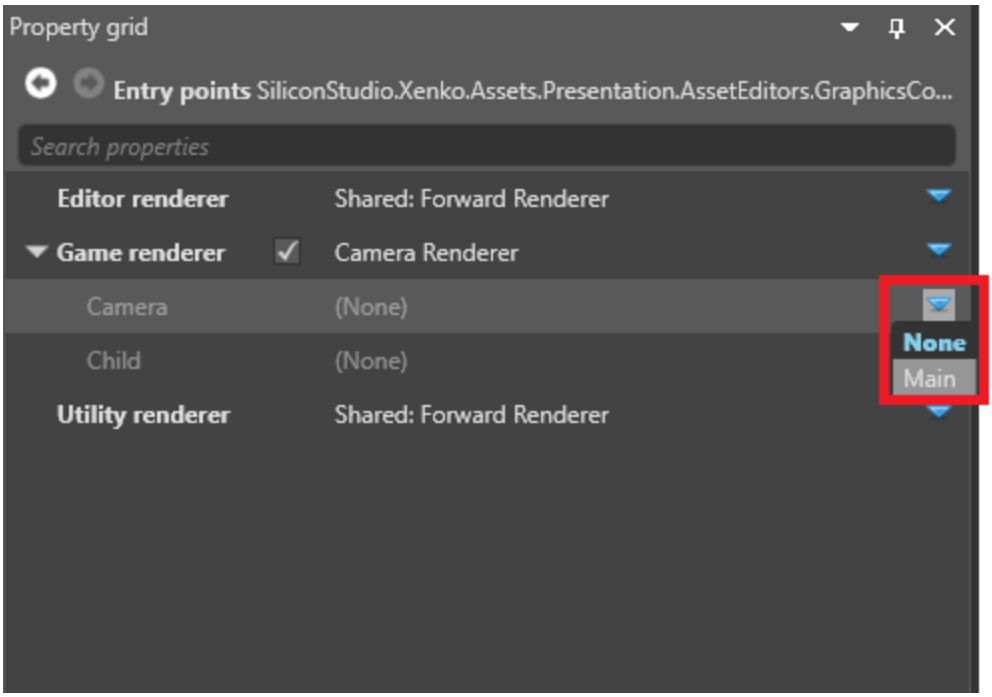
3. Next to **Game rendererer**, click (**Replace**) and select **Camera Renderer**.



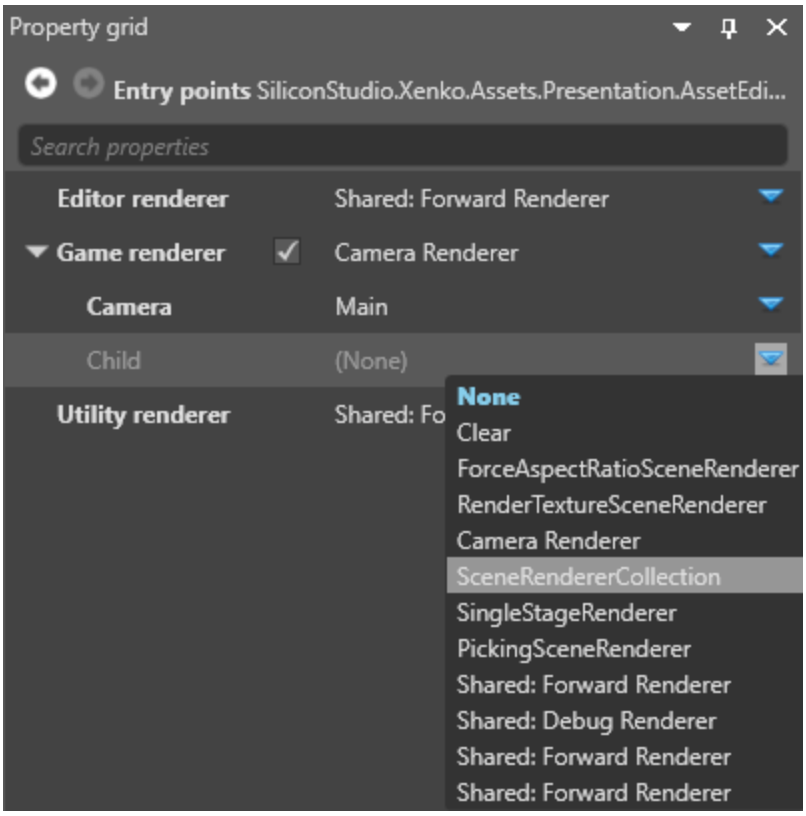
Currently, **all** renderers must have a camera, or be a child of a renderer that has a camera. This applies even to renderers that don't necessarily use cameras, such as the single stage renderer, which renders the UI.

For this reason, in these instructions, we'll add a game renderer with a camera, then make the two renderers children of that renderer. This makes sure both renderers have a parent with a camera.

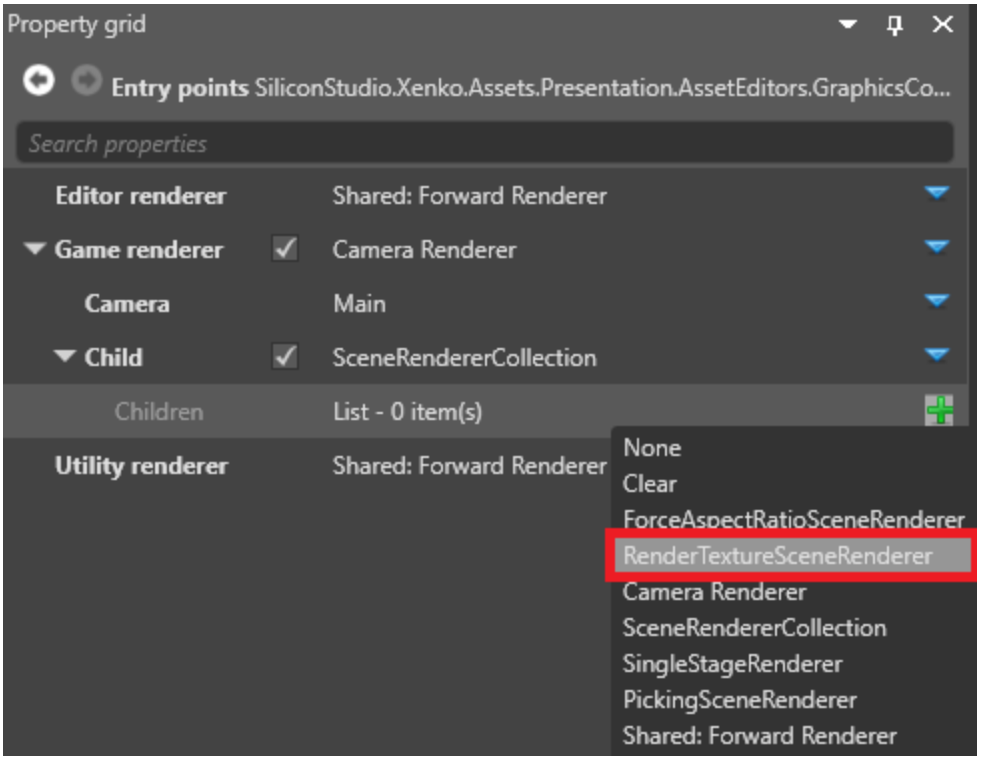
4. Next to **Camera**, click (**Replace**) and select your main game camera.



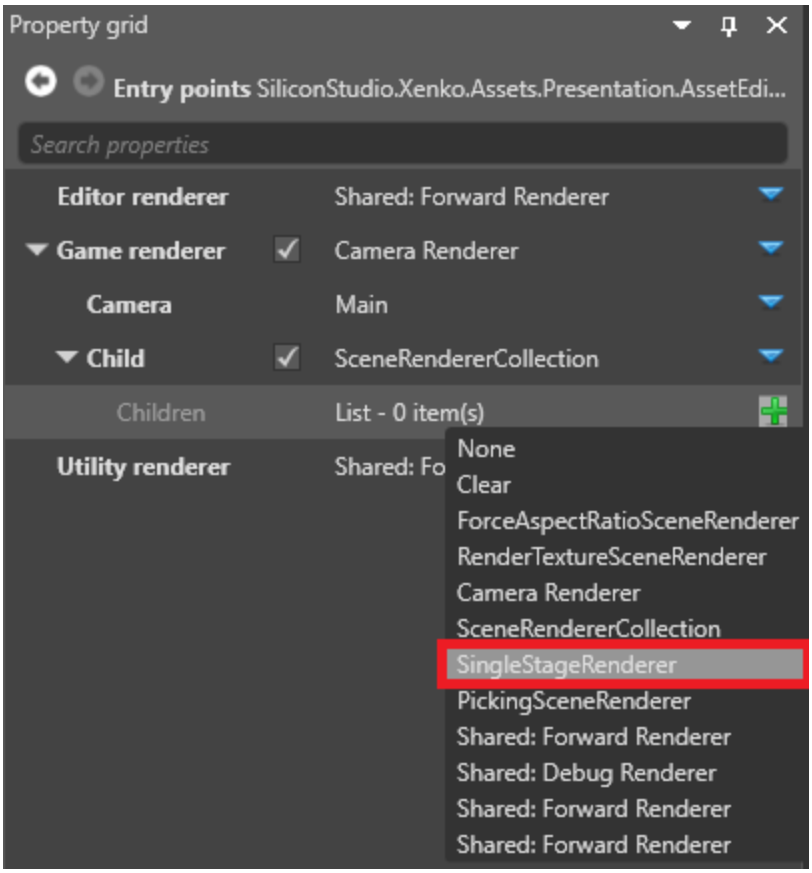
5. Next to **Child**, click (**Replace**) and select **SceneRendererCollection**.



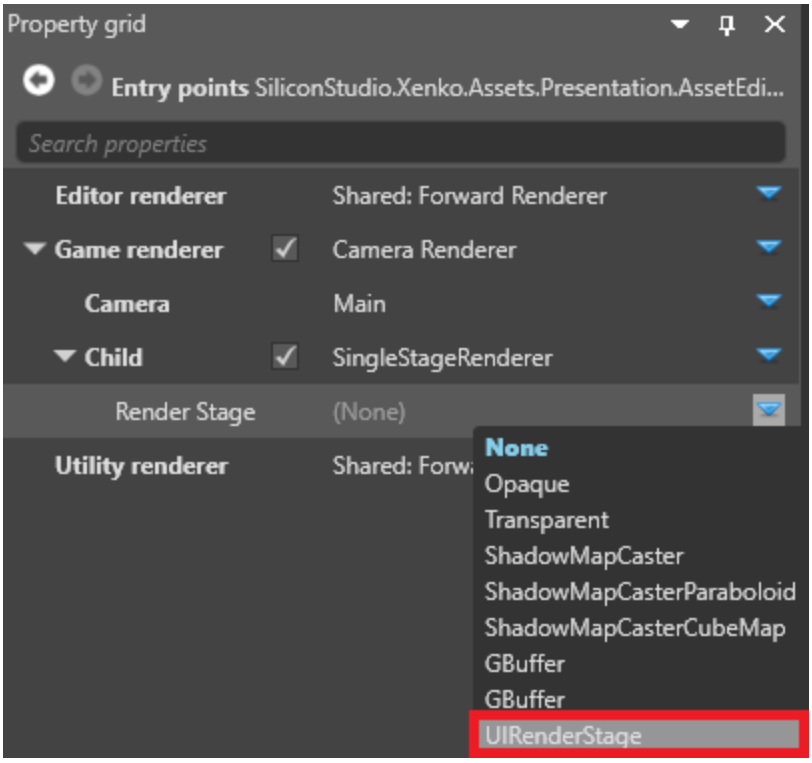
6. Next to **Children**, Click (**Add**) and select **RenderTextureSceneRenderer**.



7. Next to **Child**, click (**Replace**) and select **SingleStageRenderer**.



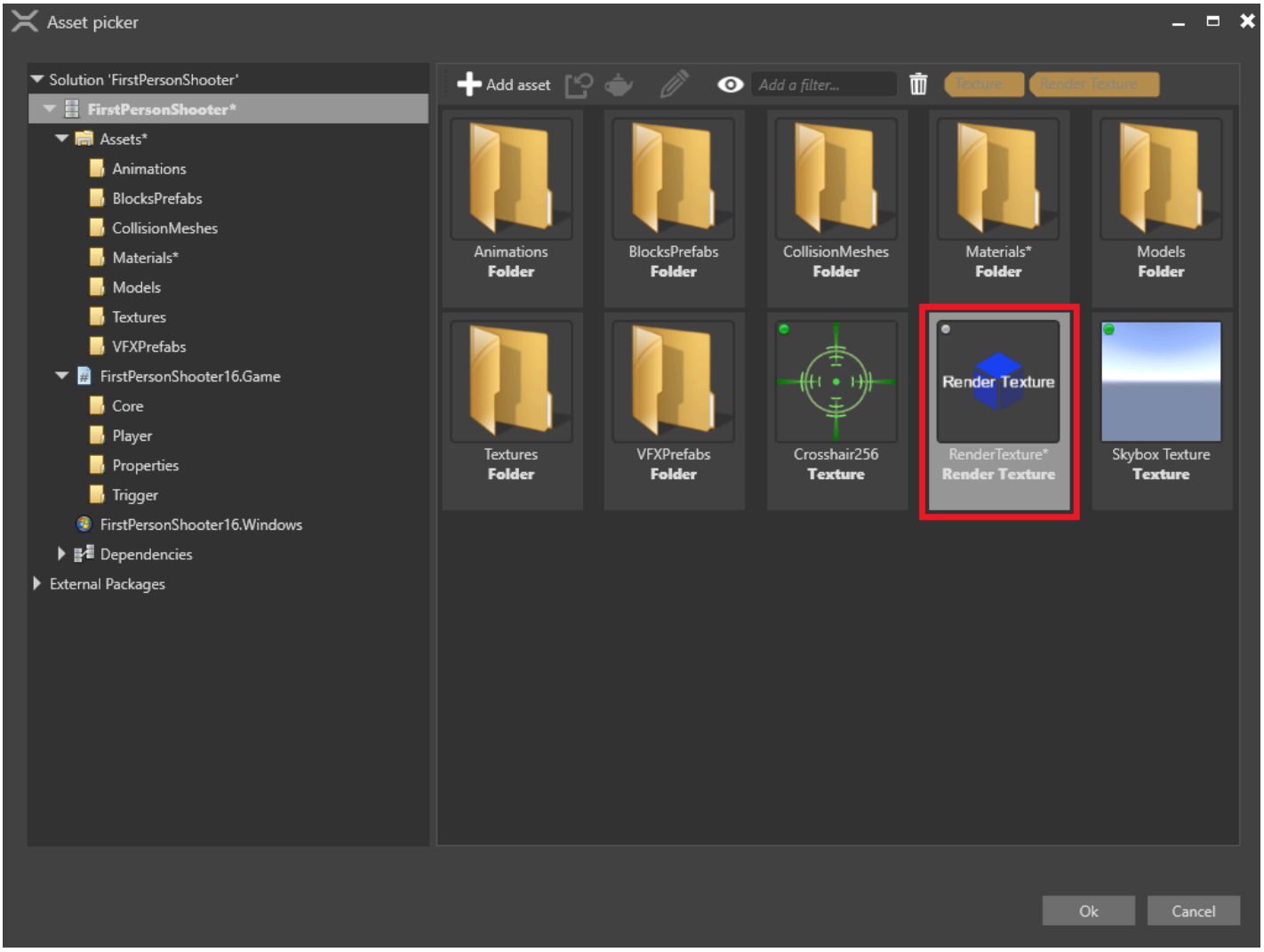
8. Next to **Render Stage**, click (**Replace**) and select **UIRenderStage**. This is the renderer that renders the UI.



9. Next to **Render Texture**, click (**Select an asset**).

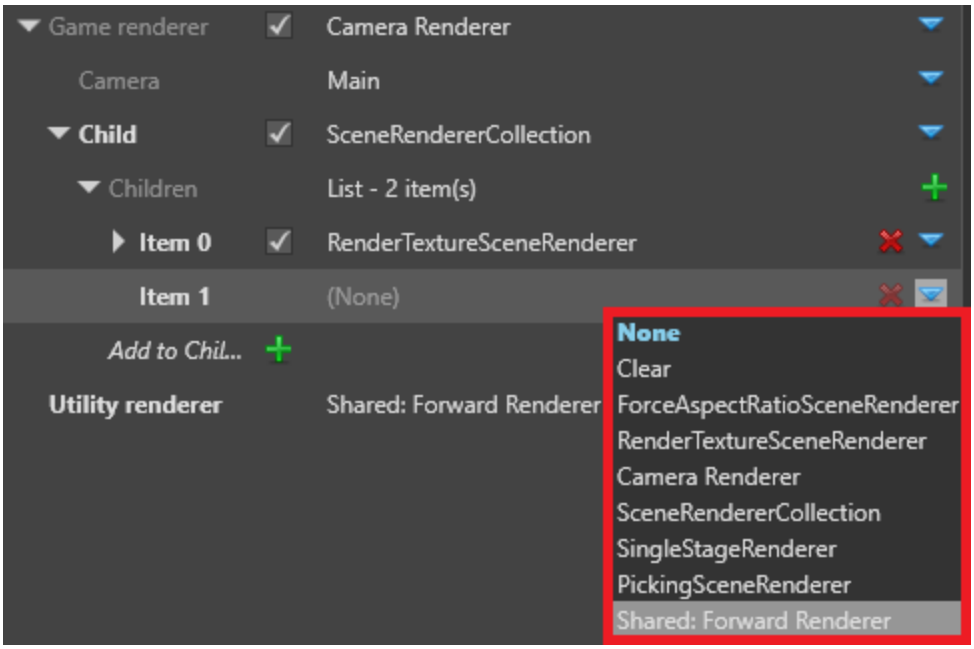
   The **Select an asset** window opens.

10. Select the **render texture** and click **OK**.



Game Studio adds the render texture to the renderer.

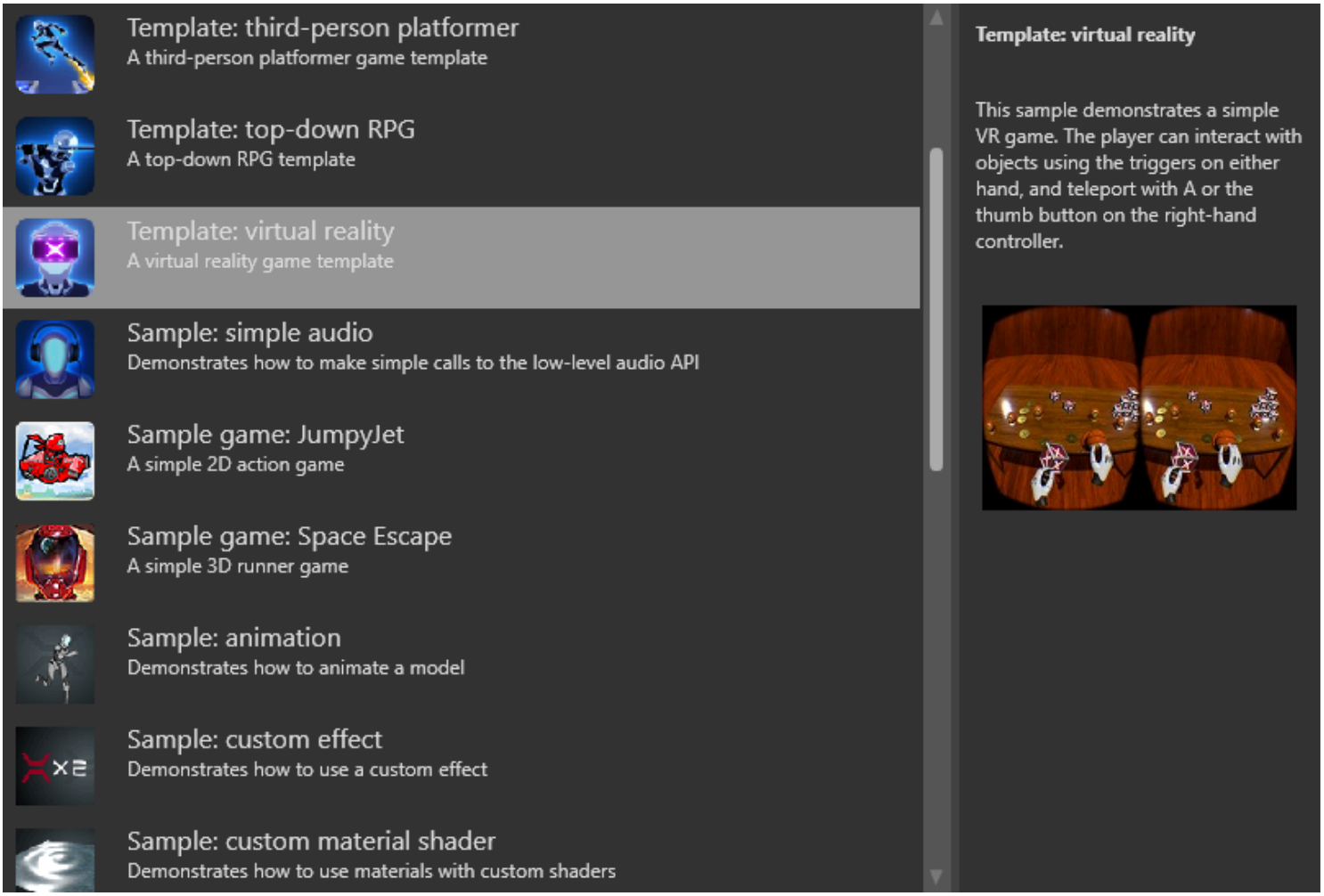
11. Under **Game renderer**, next to **Children**, click (**Add**) and select **Forward renderer**.



Your game is now ready to render the UI to an overlay in your VR device.

VR template

For an example of a UI overlay implemented in a VR game, see the VR template included with Stride.



See also

- Overlays
- UI
- Render textures
- Graphics compositor

Virtual reality sickness

Some players experience nausea and discomfort when playing VR games. Though the causes aren't completely understood, it seems to be mainly caused by the player moving around a virtual environment while their real-world body remains still.

There may be no way to completely prevent VR sickness in every player. However, there are a few things to keep in mind to minimize it in your game. We recommend you test your game with as wide a range of players as possible.

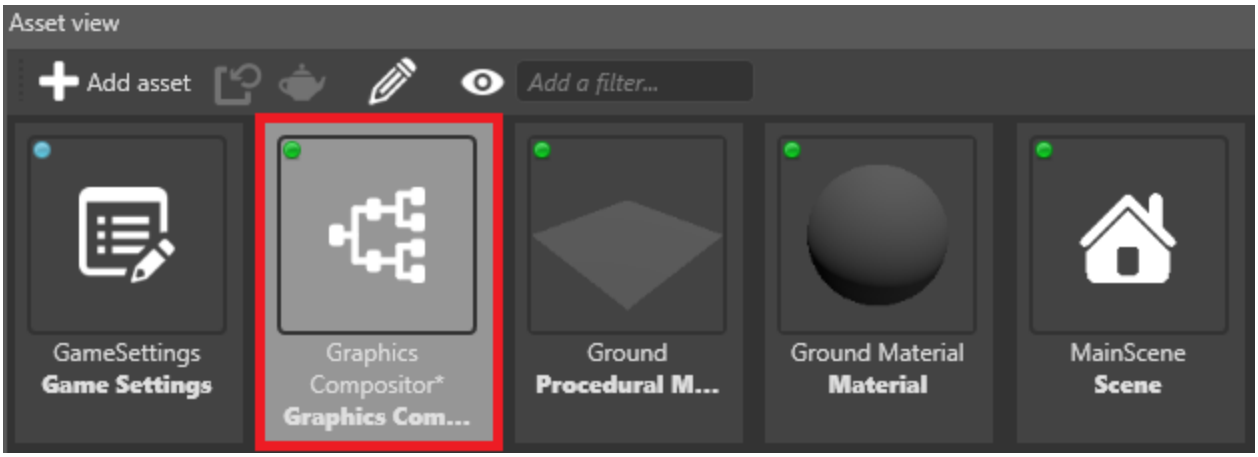
Camera movement

In general, players should control the camera by moving their head. Moving the camera by other methods, such as gamepads or keyboards, seems to be the biggest cause of VR sickness, especially with horizontal (yaw) movement.

Disable camera movement

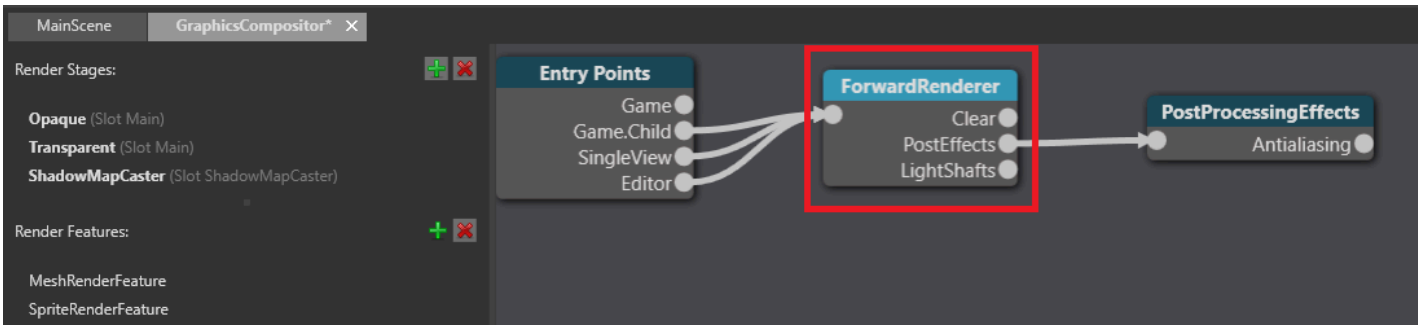
To disable camera movement from inputs other than VR devices:

1. In the **Asset View** (in the bottom pane by default), double-click the **Graphics Compositor** asset.

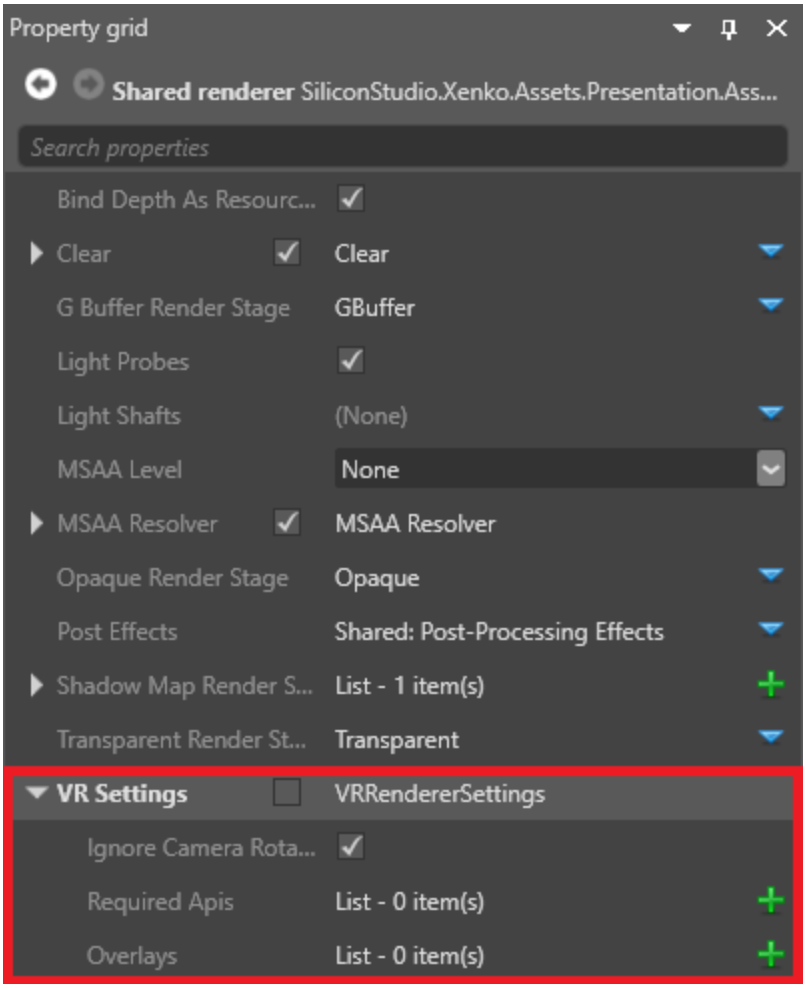


   The graphics compositor editor opens.

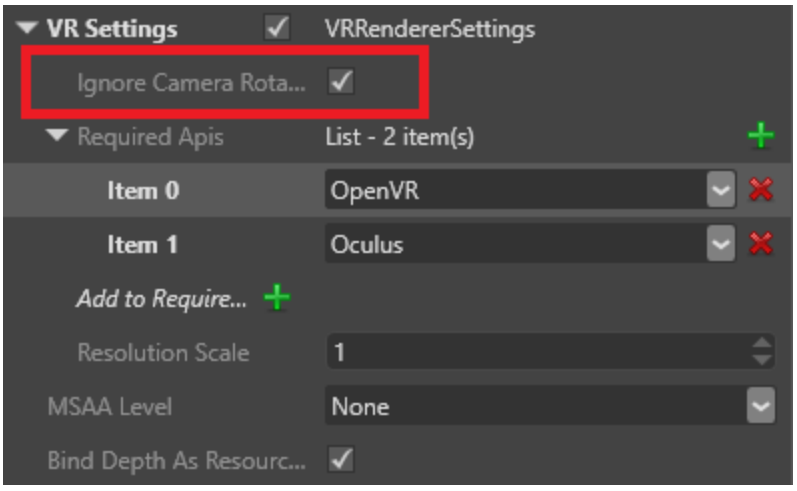
2. Select the **ForwardRenderer**.



3. In the **Property Grid** (on the right by default), expand **VR Settings**.



4. Select **Ignore camera rotation**.



For more information about the graphics compositor, see the Graphics compositor page.

Framerate

In general, the higher the framerate, the less likely players are to become sick. Framerates below 60fps seem especially likely to cause sickness.

Vection

Vection is the sensation of movement caused by the environment changing. You might have experienced this in the real world; for example, if you've been on a stationary train and a nearby train moves, creating the sensation that your own train is moving in the opposite direction. This can cause sickness in VR.

To reduce vection in your game, use simple textures and reduce the player movement speed.

Acceleration

Acceleration can cause VR sickness. For example, if the player moves on a train that speeds up and slows down, this causes more sickness than if the train moves at a constant speed.

Static point of reference

Adding a static point of reference to the player view, such as a HUD or virtual "helmet", may help reduce sickness.

See also

- Virtual reality sickness (Wikipedia)

Packaging

Introduction

Since 3.1, Stride is using NuGet format to pack and reference not only the code libraries, but also Stride assets.

As a result, you can:

- Create a NuGet package out of your project to share your Stride assets.
- Consume a NuGet package by simply referencing it. You can then use its Stride assets.

Topics

- Consume packages
- Create packages

Consume packages

Beginner Programmer

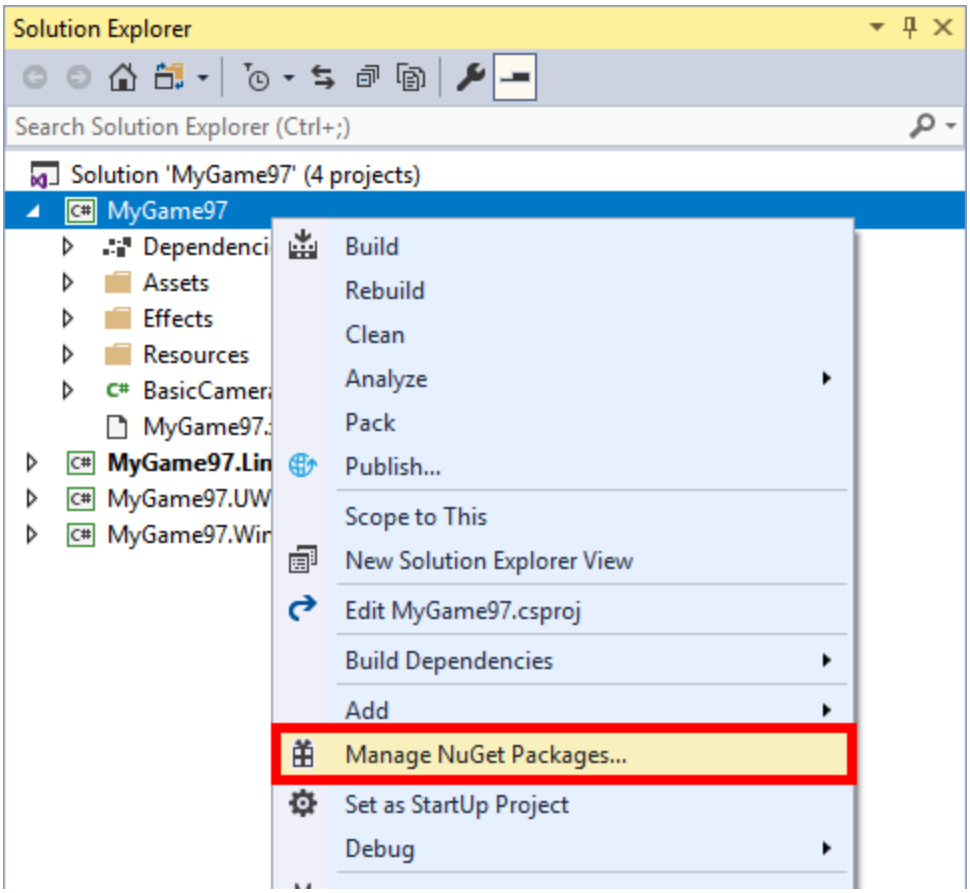
Open your project in Visual Studio

> **NOTE**
>
> Game Studio will later support adding NuGet packages directly.

First of all, after saving all your changes, open your project with Visual Studio. You can easily do this by clicking the appropriate button on the toolbar:

Add a reference

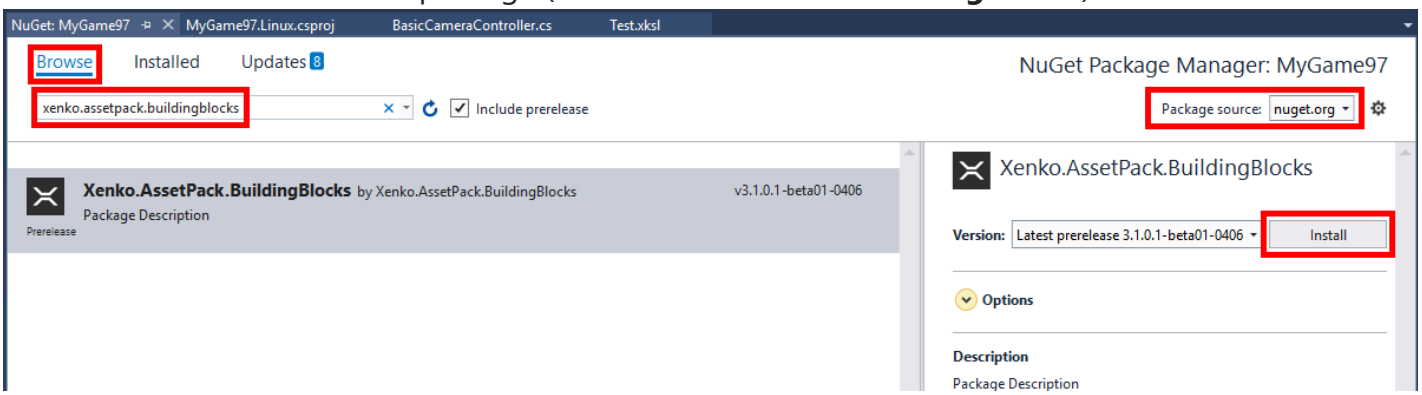
1. In the **Solution Explorer**, right-click on the project and click on **Manage NuGet Packages...**



2. For our example, let's use `Stride.AssetPack.BuildingBlocks` package:

    - Choose "nuget.org" or "All" as the **Package source**

- Make sure **Include prerelease** is checked (if necessary)
- Go to the **Browse** tab
- **Search** for a Stride asset package (i.e. **Stride.AssetPack.BuildingBlocks**) and select **Install**

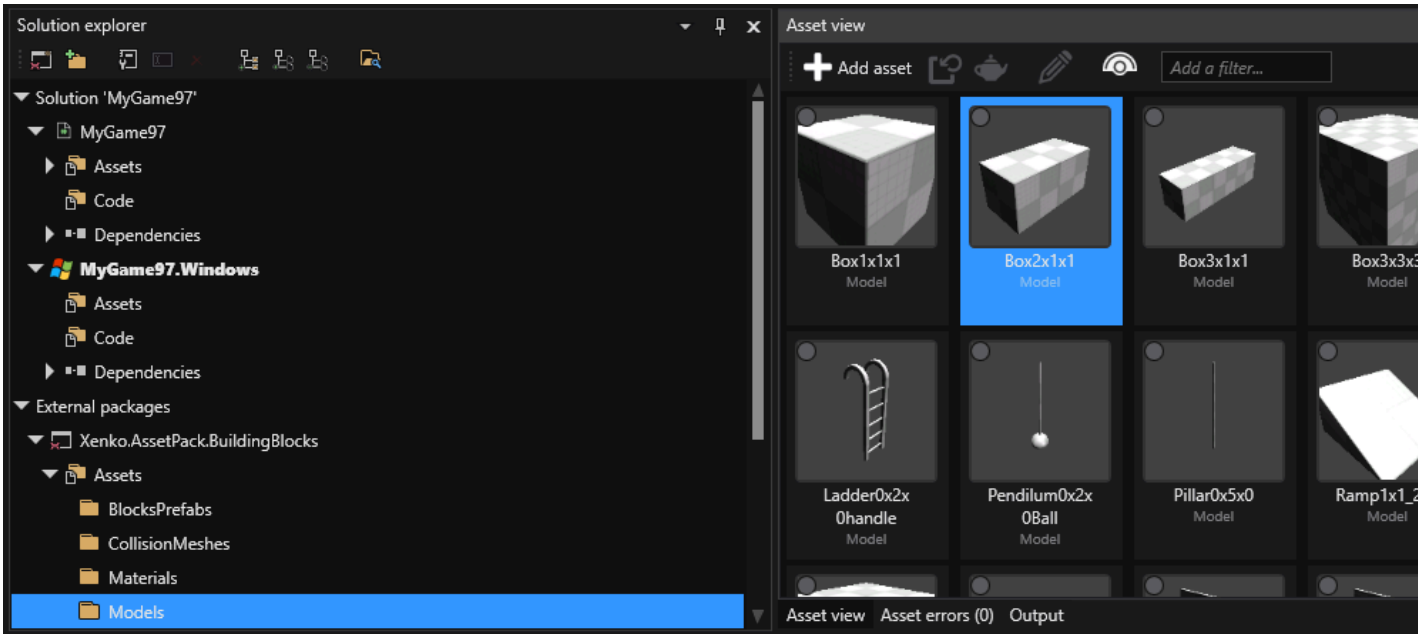


3. Save the Visual Studio project.

Use assets in Game Studio

1. In **Game Studio**, go to the **File** menu and select **Reload project**

2. You should now be able to see the referenced project and its assets in **Solution explorer**



> **NOTE**
>
> Those assets are readonly and as such can't be dragged and dropped into the scene. This will be fixed soon. In the meantime, you can still use the asset selector to change an existing model or material reference to one from the asset pack.

Create packages

Intermediate Programmer

Open your project in Visual Studio

First of all, after saving all your changes, open your project with Visual Studio. You can easily do this by clicking the appropriate button on the toolbar:

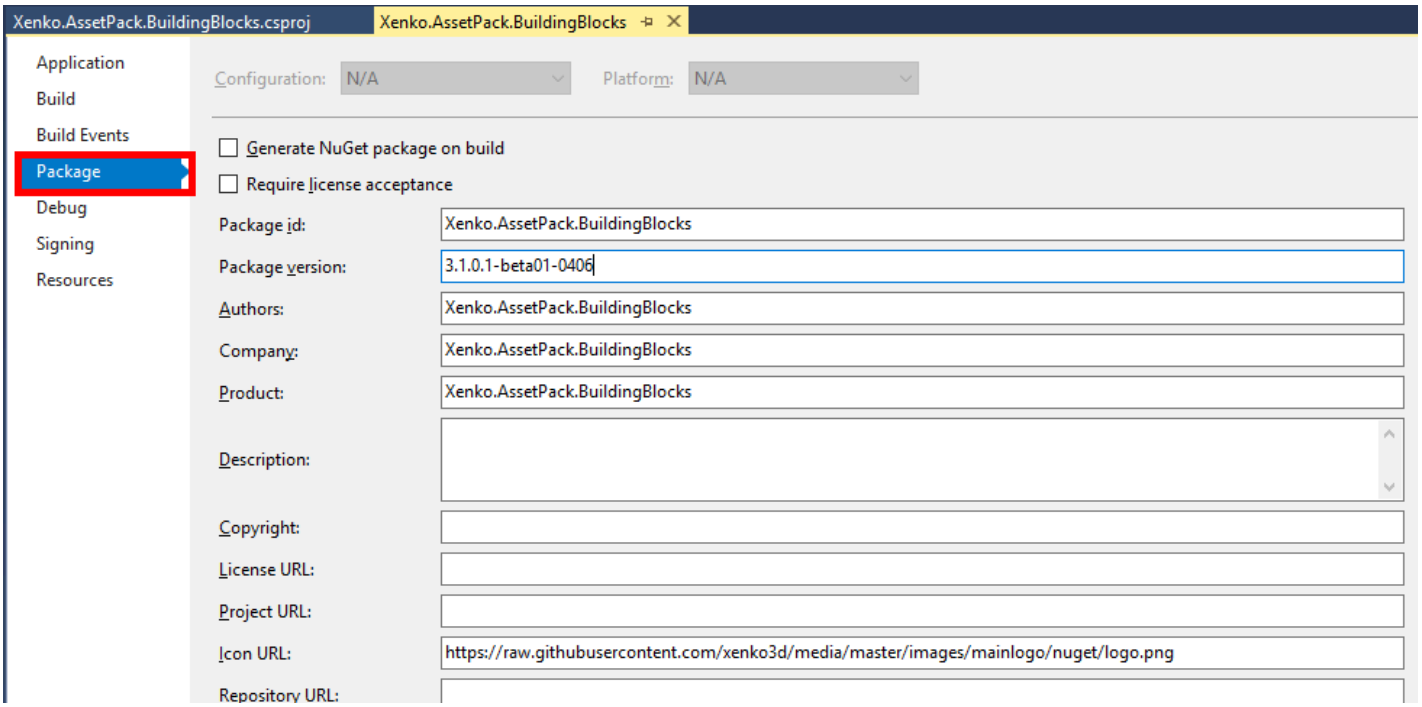
A few things to look out for:

- Delete unnecessary assets (i.e. GameSettings, etc...)
- Delete unnecessary `PackageReference`

Optional: Setup Package properties

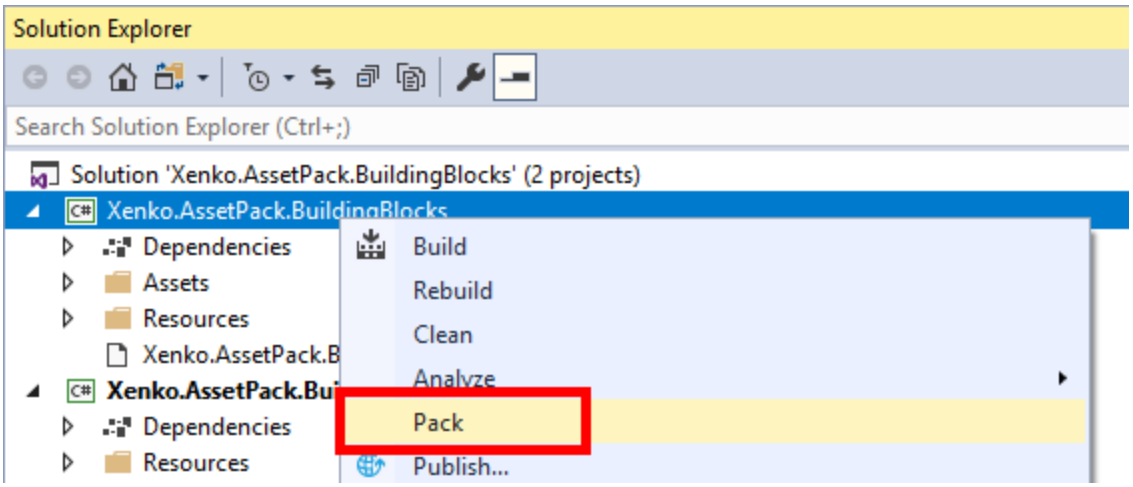
1. In the **Solution Explorer**, right-click on the project and click on **Properties**.

2. Go to the **Package** tab and edit Package version, description, URL, etc.



Pack

1. In the **Solution Explorer**, right-click on the project and click on **Pack**.



2. Visual Studio will build and pack the project. The resulting `.nupkg` should be in `bin\Debug` or `bin\Release` folder, depending on your configuration.

Publish

You can now publish the `.nupkg` file on a NuGet repository such as nuget.org.

There is several ways to do that: `nuget.exe` client, `dotnet.exe` client or nuget.org Upload Package

For additional information, please reference to Publishing packages in NuGet documentation.

Once your package is properly listed, it can now be consumed by other Stride users!

Extensibility

Introduction

Stride game project is a regular .NET project, and as such, it can be extended by a regular C# library. This is a great way to share code between multiple projects, or to create reusable components.

Read more about this subject in C# Libraries.

C# Libraries

Advanced Programmer

If you want to share code between multiple projects or create reusable components, you can create a C# library and reference it in your Stride project.

If your library uses the DataContractAttribute and you want to reference it through a **NuGet** package, there are additional steps required to make it compatible with Stride.

Adding a Module Initializer

First, add a module initializer to your library. This ensures your library is properly registered with Stride's serialization system.

Example `Module.cs`:

```
using Stride.Core.Reflection;
using System.Reflection;

namespace MyProjectName;

internal class Module
{
    [ModuleInitializer]
    public static void Initialize()
    {
        AssemblyRegistry.Register(typeof(Module).GetTypeInfo().Assembly,
AssemblyCommonCategories.Assets);
    }
}
```

Updating to the Latest Stride NuGet Packages

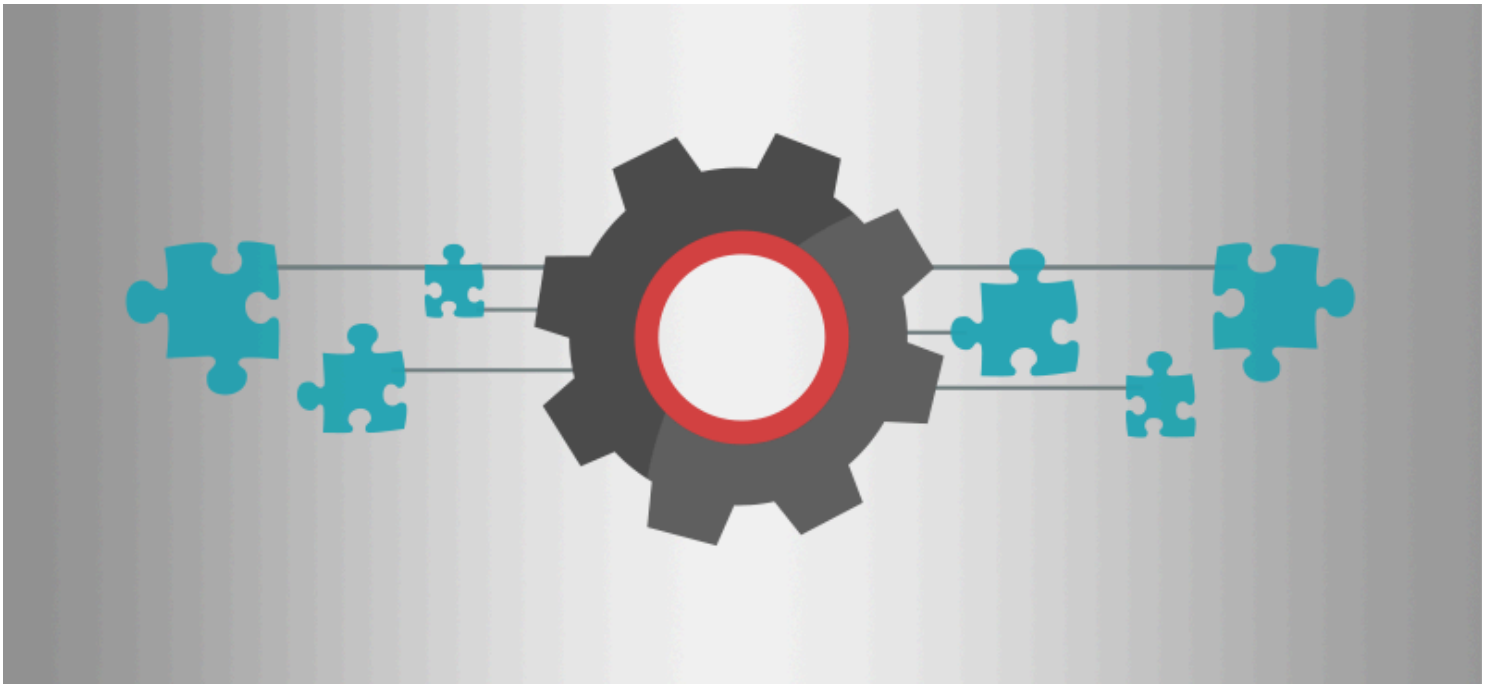
If your library references any Stride NuGet packages, you must recompile it with the latest version of those packages. This ensures compatibility with the current Stride ecosystem.

About the Module Initializer Attribute

The `ModuleInitializer` attribute is now generated using a Roslyn source generator. This means the file `sources/core/Stride.Core.CompilerServices/Generators/ModuleInitializerGenerator.cs` must run during your code's compilation. Otherwise, the module initializer and potentially other source generators added in the future will not function correctly.

Troubleshooting

These pages describe how to fix problems with Stride.



- Logging
- Debug text
- Profiling
- Unit Tests
- Stride doesn't run
- Default value changes ignored at runtime
- Lights don't cast shadows
- Full call stack not available
- SceneCameraRenderer error
- Unable to resolve Stride.GameStudio
- The Launcher is showing only 3.x version

Logging

Intermediate Programmer

You can **log** information about your game while it runs using Log.

Unlike profiling, which retrieves information automatically, it's up to you to create your own log messages and define when they're triggered. For example, you can create a log message that triggers when a character performs a certain action. This is useful to investigate how your game is performing.

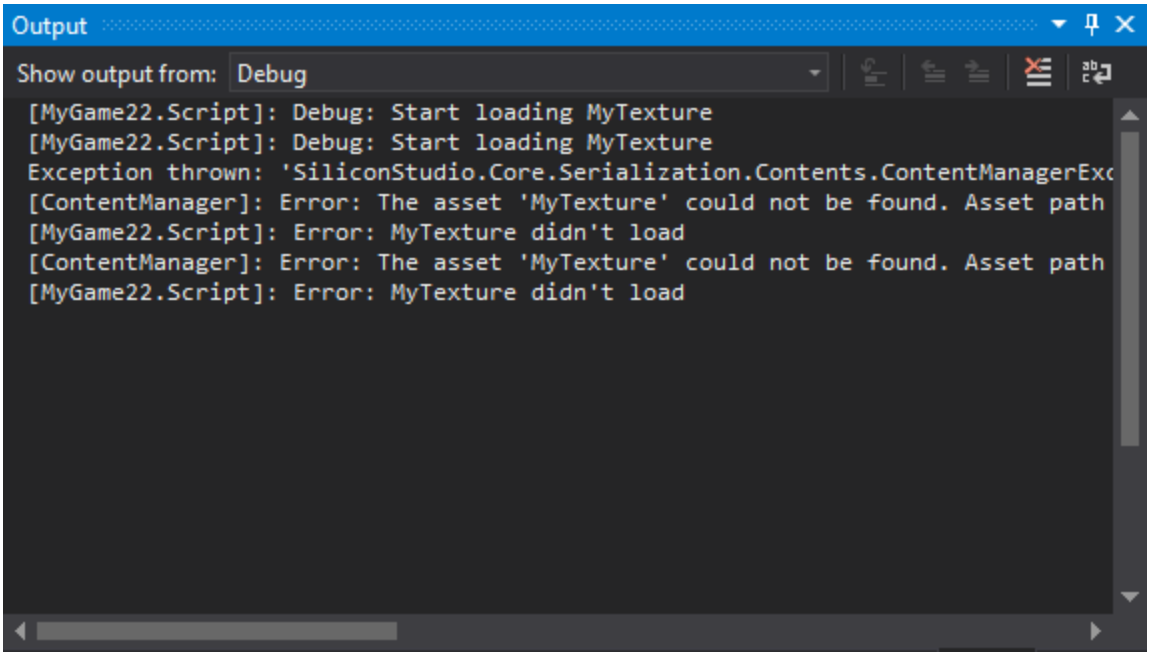
> **NOTE**
>
> Logging is disabled when you build the game in release mode.

When you use logging and run your game in debug mode, Stride opens a console in a second window to display logging information. The messages are color-coded by level. The name of the module (such as the script containing the log message) is displayed in brackets. This is followed by the log level (eg **Warning**, **Error**, etc), then the log message.

```
[MyGame.Script]: Debug: Start loading MyTexture
[ContentManager]: Error: The asset 'MyTexture' could not be found. Asset path should be 'MyFolder/MyAssetName'.
Check that the path is correct and that the asset has been included into the build.
[MyGame.Script]: Error: MyTexture didn't load
```

The console displays log messages from all modules, not just your own scripts. For example, it also displays messages from the ContentManager.

If you run your game from Visual Studio, log messages are shown in the Visual Studio **Output** window instead.



Log levels

There are six levels of log message, used for different levels of severity.

Log level	Color	Description
Debug	Gray	Step-by-step information for advanced debugging purposes
Verbose	White	Detailed information
Info	Green	General information
Warning	Yellow	Minor errors that might cause problems
Error	Red	Errors
Fatal	Red	Serious errors that crash the game

By default, the log displays messages for the level **Info** and higher. This means it doesn't display **Debug** or **Verbose** messages. To change this, see **Set the minimum level** below.

Write a log message

In the script containing code you want to log, write:

```
Log.Debug("My log message");
```

You can replace `Debug` with the level you want to use for the log message (see **Log levels** above).

You can combine this with `if` statements to log this message under certain conditions (see **Example script** below).

Set the log level

You can set a minimum log level to display. For example, if you only want to see messages as severe as **Warning** or higher, use:

```
Log.ActivateLog(LogMessageType.Warning);
```

> **NOTE**
>
> This isn't a global setting. The log level you set only applies to the script you set it in.

Change the log level at runtime

```
((Game)Game).ConsoleLogLevel = LogMessageType.myLogLevel;
```

Disable a specific log

```
GlobalLogger.GetLogger("RouterClient").ActivateLog(LogMessageType.Debug,
LogMessageType.Fatal, false);
// Disables logging of the RouterClient module
```

Disable logging in the console

```
((Game)Game).ConsoleLogMode = ConsoleLogMode.None;
```

Create a log file

To save the log output to a text file, add this code to the `Start` method:

```
var fileWriter = new TextWriterLogListener(new FileStream("myLogFile.txt",
FileMode.Create));
GlobalLogger.GlobalMessageLogged += fileWriter;
```

This creates a file in the Debug folder of your project (eg *MyGame\MyGame\Bin\Windows\Debug\myLogFile.txt*).

Example script

The following script checks that the texture `MyTexture` is loaded. When the texture loads, the log displays a debug message (`Log.Debug`). If it doesn't load, the log records an error message (`Log.Error`).

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Stride.Core.Diagnostics;
using Stride.Core.Mathematics;
using Stride.Input;
using Stride.Engine;
using Stride.Graphics;

namespace MyGame
{
    public class Script : SyncScript
```

```
    {
            public Texture myTexture;

        public override void Start()
        {
            // Initialization of the script.
            Log.ActivateLog(LogMessageType.Debug);
            Log.Debug("Start loading MyTexture");

            myTexture = Content.Load<Texture>("MyTexture");
            if (myTexture == null)
            {
                Log.Error("MyTexture not loaded");
            }
            else
            {
                Log.Debug("MyTexture loaded successfully");
            }
        }
    }
}
```

See also

- Debug text
- Profiling
- Scripts

Debug text

Beginner Programmer

You can print debug text at runtime with DebugText. For example, you can use this to display a message when a problem occurs.

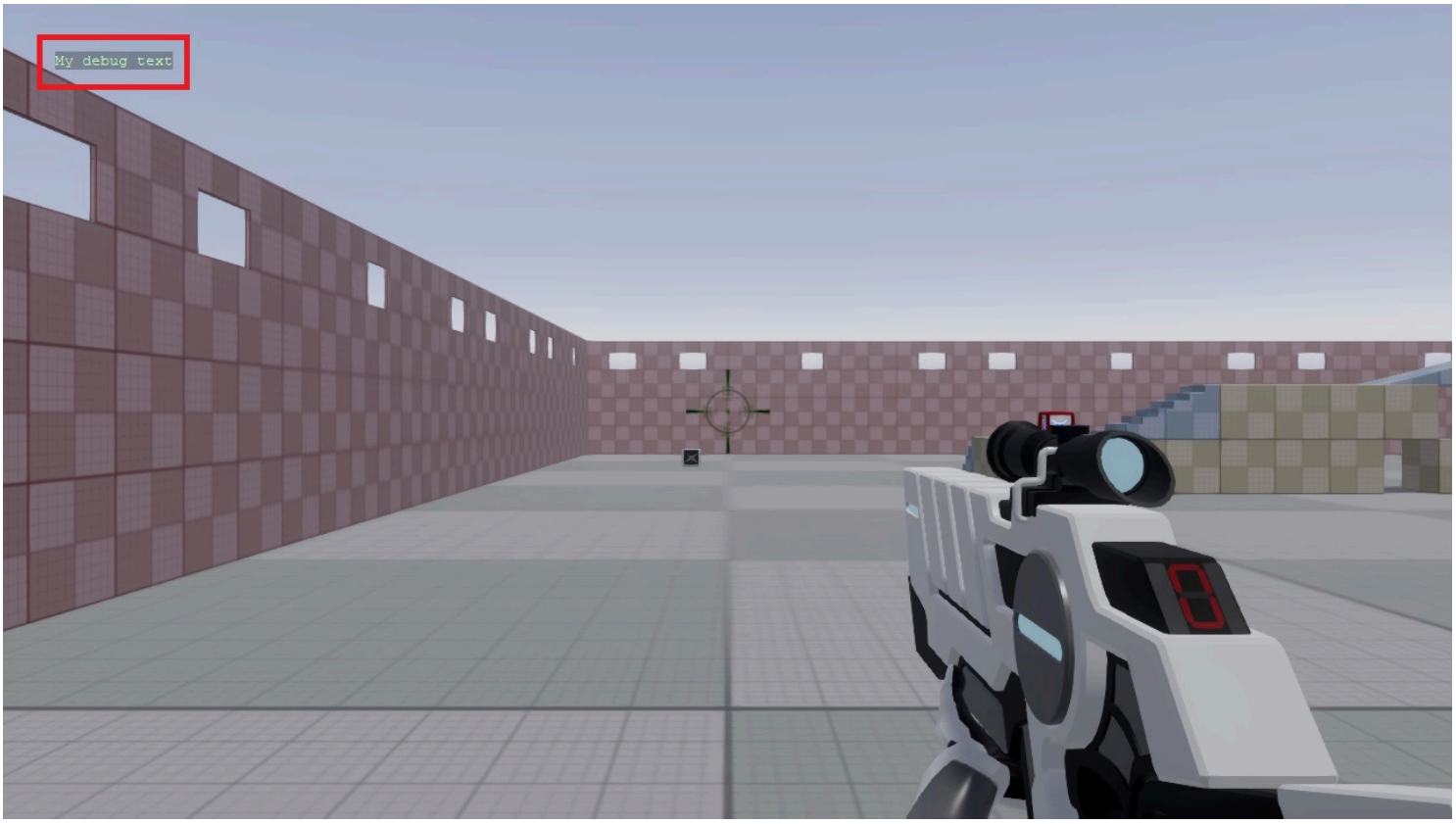
> **NOTE**
> Debug text is automatically disabled when you build the game in release mode.

In the `Update` method of your script, add:

```
DebugText.Print("My debug text",new Int2(x: 50, y: 50));
```

Where `x` and `y` are the pixel coordinates to display the text at.

The debug message is displayed when you run the game.



To hide debug text, use:

```
DebugText.Visible = false;
```

Example script

The following script checks that the texture `MyTexture` is loaded. If it isn't loaded, the game displays the debug text "MyTexture not loaded".

```
using Stride.Core.Mathematics;
using Stride.Engine;
using Stride.Graphics;

namespace MyGame
{
    public class Script : SyncScript
    {
                public Texture myTexture;

        public override void Start()
        {
            // Initialization of the script.
            myTexture = Content.Load<Texture>("MyTexture");
        }

        public override void Update()
        {
                        if (myTexture == null)
                DebugText.Print("MyTexture not loaded", new Int2(x: 50, y: 50));
        }
    }
}
```

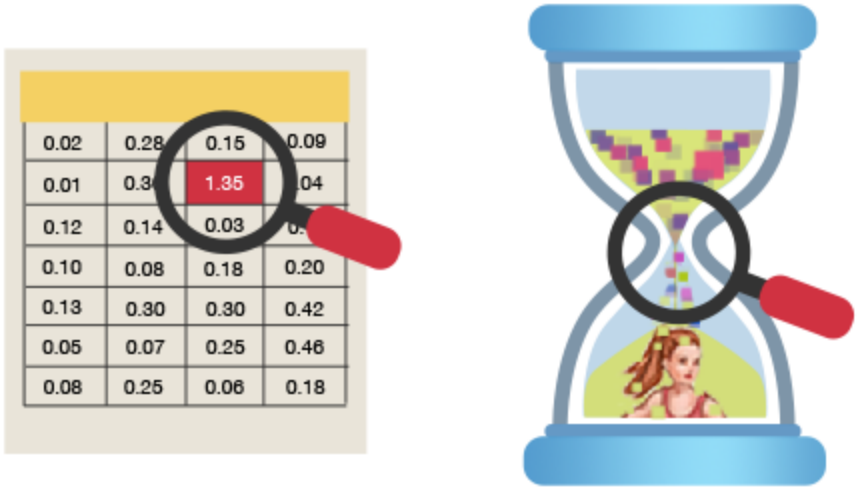
See also

- Logging
- Scripts

Profiling

Beginner Programmer

You can **profile** your project to check its runtime performance and find problems. Use the Stride **Game Profiler** script or an external profiling tool such as the Performance Profiler in Visual Studio.

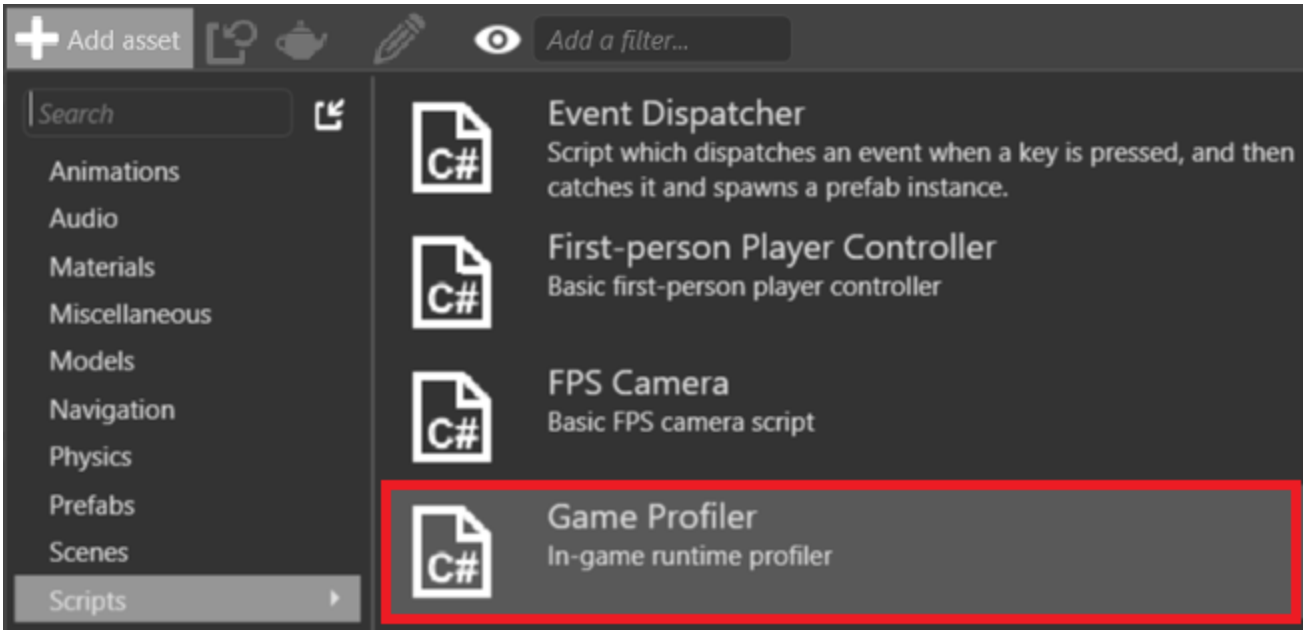


Profile with the Stride **Game Profiler** script

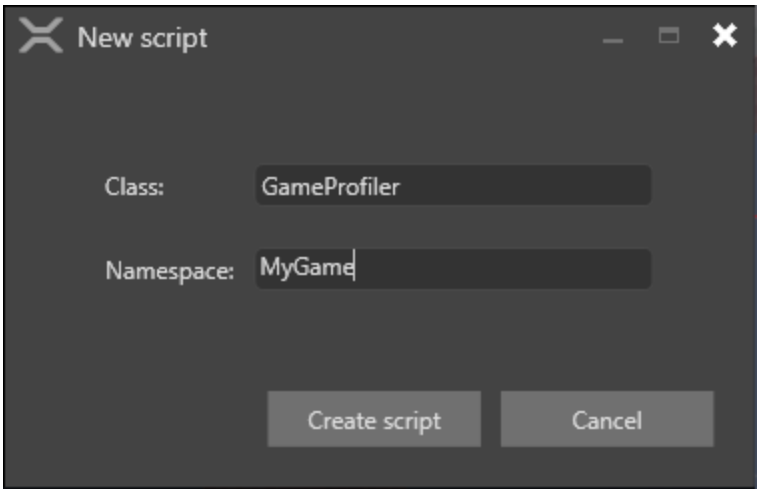
The **Game Profiler** script shows how performance costs change at runtime. This helps isolate bottlenecks and find their cause.

To use the script:

1. In the **Asset View**, click Add asset and select **Scripts > Game Profiler**.



2. The **New script dialog** opens. Leave the default information and click **Create script**.

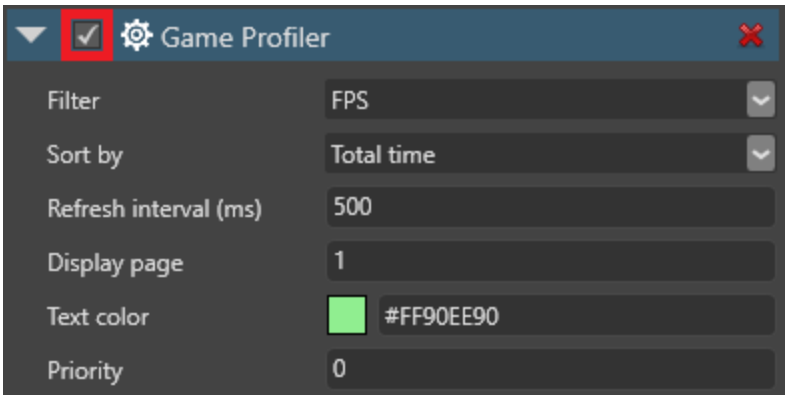


   Game Studio adds the GameProfiler script to your project.

3. Add the script to an entity. For instructions, see Use scripts.

4. Select the entity that contains the **GameProfiler**.

5. In the **Property Grid** (on the right by default), enable the **Game Profiler** component.



> **TIP**
> You can also enable and disable the profiler at runtime with **Left Ctrl + Left Shift + P**.

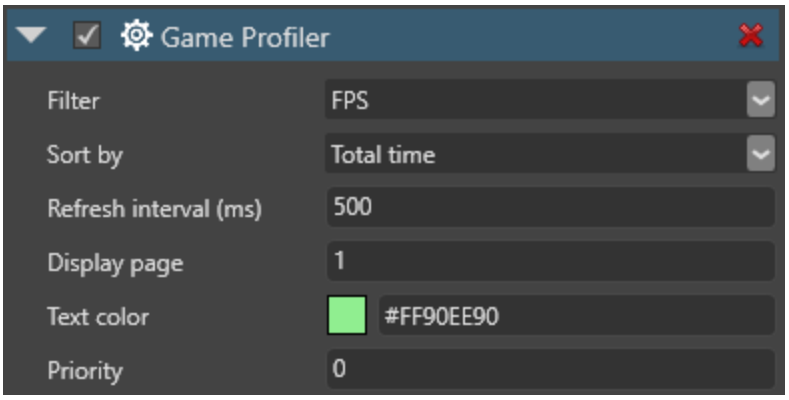
6. Run the game.

   The Game Profiler shows profiling results as your game runs.

> **NOTE**
> Game Profiler disables VSync. This gives you the true profiling values, ignoring sync time.

Game Profiler properties

To change the Game Profiler properties, select the **GameProfiler** entity and use the **Property Grid**.



Property	Description
Filter	The kind of information the profiler displays (FPS only, CPU, or GPU). At runtime, change with **F1**.
Sort by	Sort the result pages by: **Name**: the profile key (the thing being profiled)

Property	Description
	Time: the key that uses the most time At runtime, toggle with **F2**.
Refresh interval (ms)	How frequently the profiler gets and displays new results. At runtime, control with **-** / **+**.
Display page	The results page displayed. At runtime, jump to a page with the **number keys**, or move forward and backwards with **F3** and **F4**.
Text color	The color of the profiler text
Priority	See Scheduling and priorities

Understanding the Game Profiler results

The top row displays information about basic performance.

```
Displaying: CpuEvents, Frame: 180910, Update: 4.17ms, Draw: 4.17ms, FPS: 240.09
```

- `Displaying`: the kind of information the profiler displays (FPS only, CPU, or GPU)
- `Frame`: the current frame
- `Update`: the average time (ms) taken to update the game since the profiler last refreshed
- `Draw`: the average time (ms) taken to render the frame since the profiler last refreshed
- `FPS`: the average number of frames rendered per second

If you select **CPU** as the display mode, the profiler displays:

```
Allocated memory> Total: 109.28MB Peak: 144.38MB Allocations: 183.96KB
Garbage collections> Gen0: 313, Gen1: 304, Gen2: 12
```

- `Total`: the amount of memory currently used
- `Peak`: the peak memory use since the game started
- `Allocations`: the amount of memory allocated or freed since the profiler last refreshed
- `Gen0`, `Gen1`, `Gen1`: the number of garbage collections per each generation of object (`Gen0` is the most recent generation)

If you select **GPU** as the display mode, the profiler displays:

```
Device: NVIDIA GeForce GTX 760 (192-bit), Platform: Direct3D11, Profile: Level_10_0, Resolution: (1280,720)
Drawn triangles: 1093.3k, Draw calls: 741, Buffer memory: 7.16[MB], Texture memory: 106.21[MB]
```

- `Device`: the graphics device (manufacturer's description)

- `Platform`: the currently used backend (eg DirectX, OpenGL, Vulkan, etc)
- `Profile`: the feature level for your game, set in **Game Settings > Rendering** (see Game settings)
- `Resolution`: the game resolution
- `Drawn triangles`: the number of triangles drawn per frame
- `Draw calls`: the number of draw calls per frame
- `Buffer memory`: the amount of memory allocated to buffers
- `Texture memory`: the amount of memory allocated to textures

In the **GPU** and **CPU** modes, the profiler displays information about the parts of the code being profiled, including active scripts.

```
TOTAL     | AVG/CALL  | MIN/CALL  | MAX/CALL  | CALLS | MARKS | PROFILING KEY / EXTRA INFO
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.AnimationProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.ParticleSystemRenderProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.PhysicsProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.BackgroundRenderProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.ModelNodeLinkProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Draw.ScriptProcessor
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.ModelRenderProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.SpriteRenderProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.TransformProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.CameraProcessor / Entities: 305
000.000ms | 000.000ms | 000.000ms | 000.001ms | 01.01 | 00.00 | Game.Update.LightProcessor / Entities: 305
```

> **NOTE**
> Each value describes the events per frame since the last profiler refresh.

Column	Description
`TOTAL`	The total time taken to execute the code in one frame
`AVG/CALL`	Average time taken to execute a single call of the code
`MIN/CALL`	The shortest amount of time taken to execute a single call of the code
`MAX/CALL`	The longest amount of time taken to execute a single call of the code
`CALLS`	The number of times the code was executed in one frame
`MARKS`	The number of times per frame marked code is executed. This column is only displayed if marked code is executed
`PROFILE KEY / EXTRA INFO`	The part of the code (such as a function or script) being profiled. This column also displays additional information, such as the number of entities affected.

Game Profiler runtime controls

You can change the Game Profiler settings at runtime using keyboard shortcuts.

Action	Control
Left Ctrl + Left Shift + P	Enable/disable the profiler
F1	Toggle between CPU, GPU, and FPS-only results
F2	Toggle between sorting by profile key and time

- / + | Slow down / speed up the refresh time F3 / F4 | Page back / page forward Number keys | Jump to a page

Use the Game Profiler in code

- Enable profiling:

```
GameProfiler.EnableProfiling();
```

- Enable profiling only for the profiler keys you specify:

```
GameProfiler.EnableProfiling(true, {mykey1,mykey2});
```

- Enable the profiling except for the profiler keys you specify:

```
GameProfiler.EnableProfiling(false, {mykey1,mykey2});
```

- To access the prolifing key of a script, use ProfilingKey.

Use external profiling tools

Instead of using the Stride Game Profiler, you can use external profiling tools to profile your project.

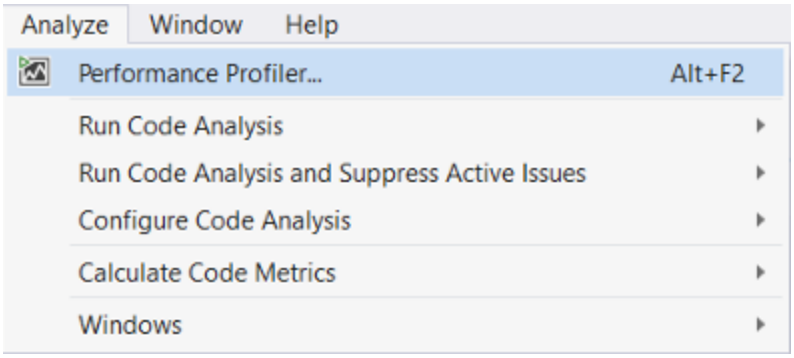
Profiler	Type	Platforms
Visual Studio profiler	Visual Studio feature	Desktop and mobile
Xamarin Profiler	Standalone tool distributed with Xamarin Studio	Mobile
RenderDoc	Standalone	Desktop and mobile

Use the Visual Studio profiler

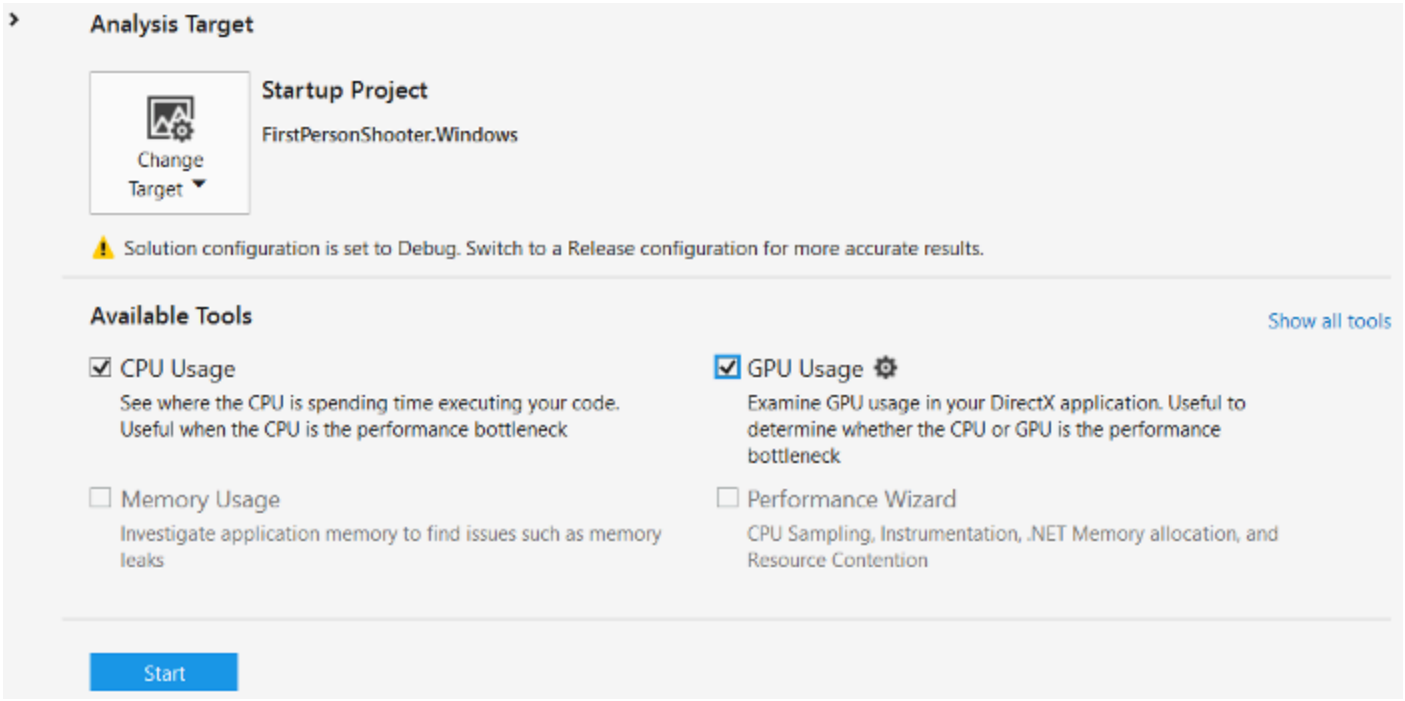
Visual Studio has powerful in-built profiling tools that can identify common performance issues.

1. In Visual Studio, open your project solution (`.sln`) file.

2. To open the profiler, press **Alt + F2**, or in the task bar click **Analyze > Performance Profiler**.

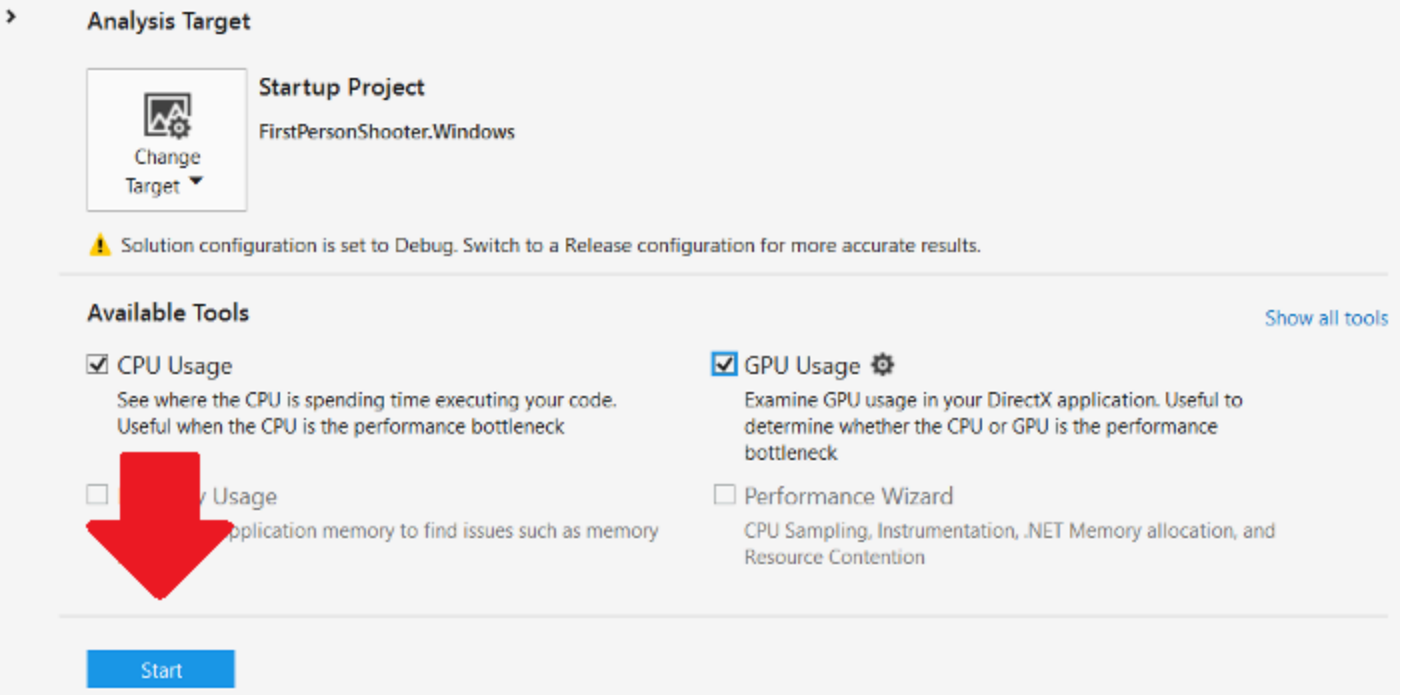


3. In the **Profiler** window, select the profiling tools you want to run.



   You can run multiple profiling tools at once.

4. To launch the profiler, in the Performance Profiler tab, at the bottom, click **Start**.



Visual Studio runs your application and begins profiling.

For more information about the Visual Studio profiler, see the MSDN documentation.

Use RenderDoc

RenderDoc is a free MIT licensed stand-alone graphics debugger that allows quick and easy single-frame capture and detailed introspection of any application using Vulkan, D3D11, OpenGL & OpenGL ES or D3D12 across Windows, Linux, Android, or Nintendo Switch™.

1. Download RenderDoc.

2. Optional: This step is optional and only necessary if you want to have render pass markers with name following the Graphics Compositor:

   2.1. In your executable project (Windows), locate `game.Run();` and insert the following code just before:

```
game.GraphicsDeviceManager.DeviceCreationFlags |= DeviceCreationFlags.Debug;
```

> **NOTE**
>
> If you have a `SharpDXException` of type `DXGI_ERROR_SDK_COMPONENT_MISSING`, please follow the instructions from https://docs.microsoft.com/en-us/windows/uwp/gaming/use-the-directx-runtime-and-visual-studio-graphics-diagnostic-features

2.2. Also, make sure profiler is enabled by calling this code from any of your game script:

```
GameProfiler.EnableProfiling();
```

3. Optional: Add a package reference to `Stride.Graphics.RenderDocPlugin`.

   You can then use the @'Stride.Graphics.RenderDocManager' class to trigger captures:

```
var renderDocManager = new RenderDocManager();
renderDocManager.StartFrameCapture(GraphicsDevice, IntPtr.Zero);
// Some rendering code...
renderDocManager.EndFrameCapture(GraphicsDevice, IntPtr.Zero);
```

Common bottlenecks

As CPU and GPU process different types of data, it's usually easy to identify which part is causing a bottleneck.

Most GPU problems arise when the application uses expensive rendering techniques, such as post effects, lighting, shadows, and tessellation. To identify the problem, disable rendering features.

If instead there seems to be a CPU bottleneck, reduce the complexity of the scene.

For graphics:

- decrease the resolution of your game
- reduce the quality of your post effects
- reduce the number of lights and size of shadow maps
- reduce shadow map sizes
- use culling techniques to reduce the number of objects and vertices rendered

For textures:

- use compressed textures on slower devices
- use sprite sheets, not individual images
- use texture atlases, not separate textures

See also

- Profiling

Unit Tests

Unit tests in Stride are set up like any other unit tests in dotnet, you create a new project specifically for unit tests, then write your tests in different C# files.

Here's a bare-bone project to get you started: `YOUR_PROJECT_NAME.Windows.Tests.csproj`

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <!-- Change this framework to match the one specified in your *.Windows.csproj -->
        <TargetFramework>net8.0-windows</TargetFramework>
        <RuntimeIdentifier>win-x64</RuntimeIdentifier>
        <OutputType>WinExe</OutputType>

        <OutputPath>..\Bin\Tests\Windows\$(Configuration)\</OutputPath>
        <AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>

        <!-- Force msbuild to check to rebuild this assembly instead of letting VS IDE guess
-->
        <DisableFastUpToDateCheck>true</DisableFastUpToDateCheck>

        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>

        <IsPackable>false</IsPackable>
        <IsTestProject>true</IsTestProject>
    </PropertyGroup>
    <ItemGroup>
        <!-- Add a reference to your game project here, do not reference the windows project
here -->
        <PackageReference Include="coverlet.collector" Version="6.0.0"/>
        <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0"/>
        <PackageReference Include="xunit" Version="2.5.3"/>
        <PackageReference Include="xunit.runner.visualstudio" Version="2.5.3"/>
    </ItemGroup>
    <ItemGroup>
        <Using Include="Xunit"/>
    </ItemGroup>
</Project>
```

And an example C# file:

```
using Stride.Engine;
```

```
public class Tests
{
    [Fact]
    public void MyBareboneTest()
    {
        Assert.NotEqual(1, 2);
    }

    [Fact]
    public void MyGameTest()
    {
        RunGameTest(async (game, scene) =>
        {
            var myEntity = new Entity();
            scene.Entities.Add(myEntity);

            Assert.NotEmpty(scene.Entities);

            await game.Script.NextFrame(); // Wait one frame if you need to

            myEntity.Scene = null;

            Assert.Empty(scene.Entities);
        });
    }

    /// <summary>
    /// Run the given function within a game, providing support for tests requiring ECS,
physics simulation, graphics and others.
    /// </summary>
    private static void RunGameTest(Func<Game, Scene, Task> asyncFunction)
    {
        using var game = new Game();

        // Fixed time step to reduce framerate discrepancies
        game.IsFixedTimeStep = true;
        game.IsDrawDesynchronized = false;
        game.TargetElapsedTime = TimeSpan.FromTicks(10000000 / 60); // 60hz, 60fps

        game.Script.AddTask(async () =>
        {
            await asyncFunction(game, game.SceneSystem.SceneInstance.RootScene);
            game.Exit();
        });
        game.Run();
```

```
    }
}
```

Stride doesn't run

Prerequisites

If you're having trouble running Stride, make sure you've installed all the prerequisites:

- .NET SDK 10.0
- Visual C++ Redistributable 2022 (or later)
- .NET Framework 4.7.2 (required for the Visual Studio plugin)
- Visual Studio or Build Tools (optional but recommended)

Alternatively, uninstall Stride, restart the Stride installer, and install the prerequisites when prompted.

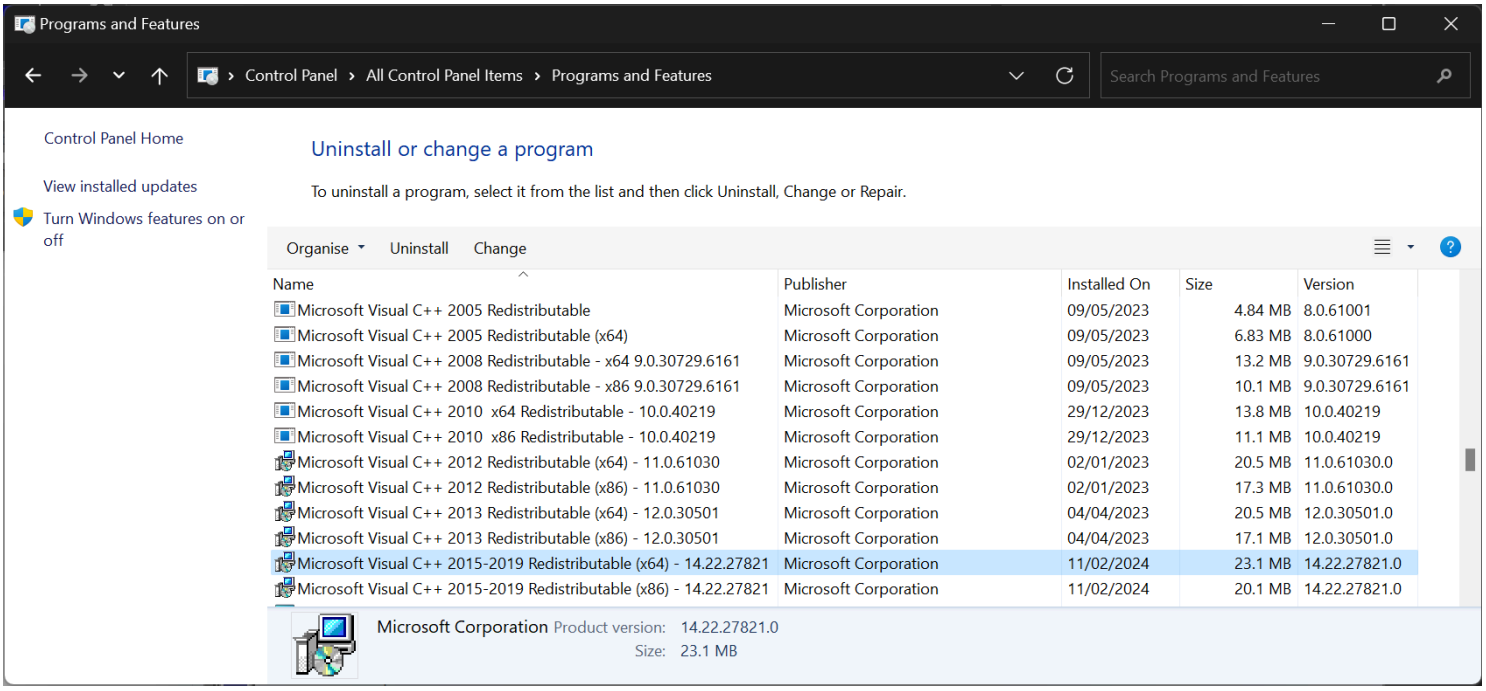
.NET SDK 10.0

.NET SDK 10.0 should have been installed by Stride prerequisite installer, if Visual Studio 2026 didn't do it previously.

If for some reason you need to install it manually, you can use this link and select the latest .NET SDK 10.0 for Windows.

Visual C++ Redistributable 2019 (or later)

To check if this is installed, see **Control Panel > Programs > Programs and Features** and look for **2015-2019 Redistributable**.



If it's not installed, you can download the Redistributable from Visual Studio Downloads (under **Other Tools, Frameworks, and Redistributables**). Make sure to install both **x86** and **x64** versions.

> **NOTE**
>
> If you see **2015-2022 Redistributable** instead, it's ok. Since 2015, they are cumulative. Just make sure the last year is at least 2019.

.NET Framework 4.7.2 (or later)

To check if this is installed, follow the instructions on this page.

If it's not installed, you can download it from the Microsoft Download Center.

> **NOTE**
>
> If you have .NET 4.8 installed, you don't need to install .NET 4.7.2. Each 4.x version is cumulative.

Visual Studio 2026 (optional)

If you have Visual Studio 2026 (or later) installed, you need to have the following workloads and/or components installed:

- **.NET desktop development** with **Development tools for .NET** optional component enabled.

> **NOTE**
>
> Earlier versions might work with older version of Stride. However, for Stride 4.3 and later you only need to have .NET SDK 10.0 installed.

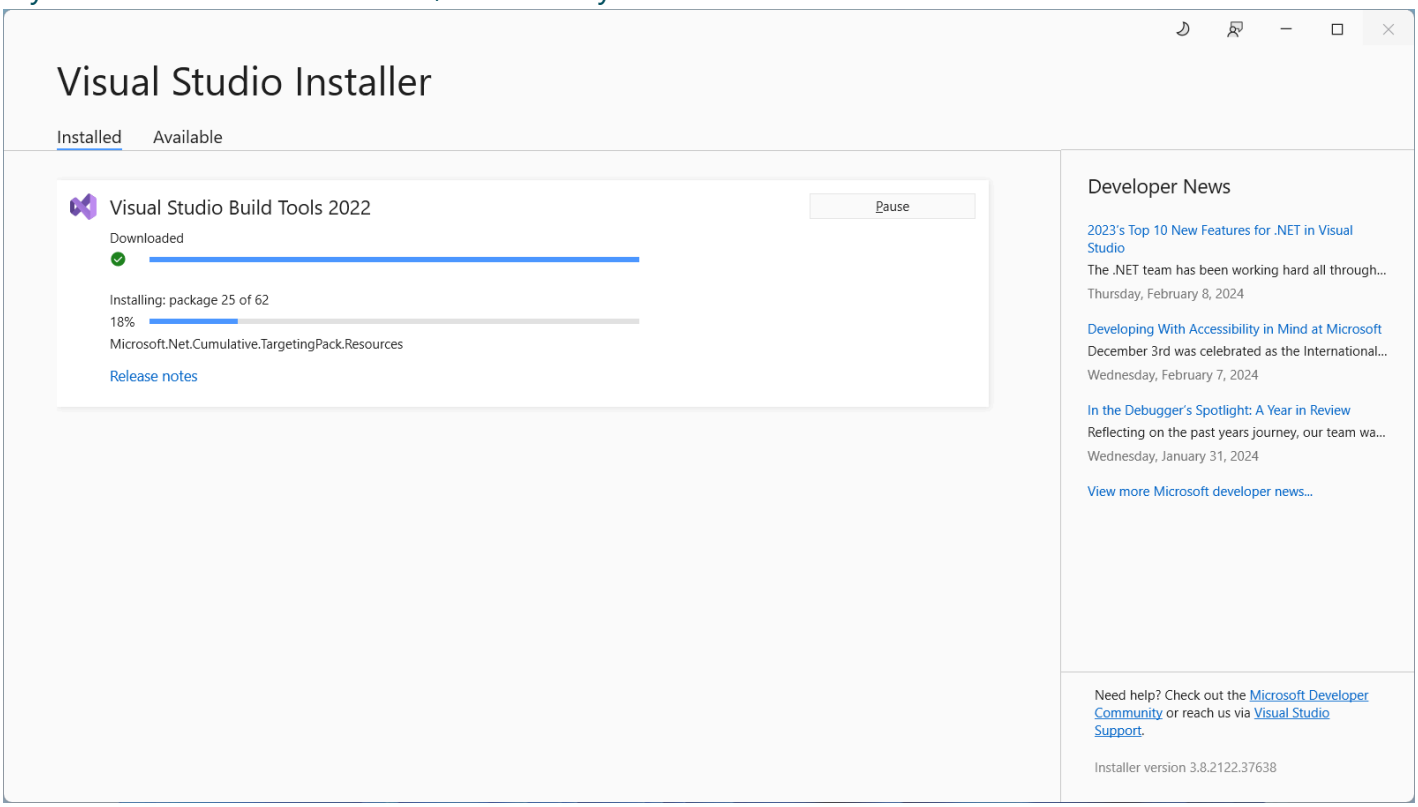
Build Tools for Visual Studio 2026 (optional)

If you **don't** have Visual Studio installed and don't want to install it, you can install **Build Tools for Visual Studio** instead. You can download this from Visual Studio Downloads (under **Tools for Visual Studio**).

You need to have the following workloads and/or components installed:

- **.NET desktop build tools** with **.NET SDK** optional component enabled.

> **NOTE**
>
> If you don't need Visual Studio, don't worry – it doesn't install it.



See also

- Install Stride

Default value changes ignored at runtime

When you add a script to your project as a component, Game Studio lists its public variables in the Property Grid. These are the values used at runtime.

However, if you then change the default value in the script, Game Studio doesn't update the component with the new value.

For example, imagine you have a script with the variable `SpeedFactor` with a default value of `5.0f`. You add the script to the project as a component. Now, in the script, you change the default value of the `SpeedFactor` variable to `6.0f`, save the script, and run the project. Game Studio doesn't update the component with the script changes, so the speed `SpeedFactor` value is still `5.0f`.

Fix

In your project, delete and re-add the script component.

Alternatively, if you want Game Studio to update the values in the component properties after you change them in the script, you can do this with additional code. You need to add a new line of code for every property you want this to apply to.

1. Add `using System.ComponentModel` at the top of the script.
2. Above the variable you want to update, add `[DefaultValue()]`. For example, if the variable is `SpeedFactor`, use:

```
[DefaultValue(6.0f)]
public float SpeedFactor { get; set; } = 6.0f;
```

When you change the value, update both the `SpeedFactor` and the `DefaultValue` to the same value.

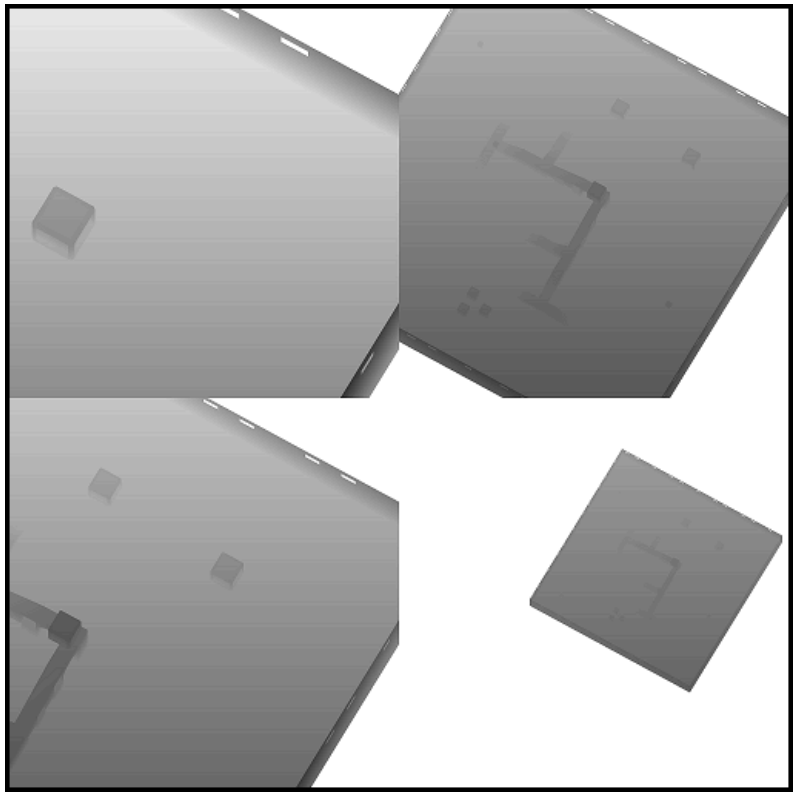
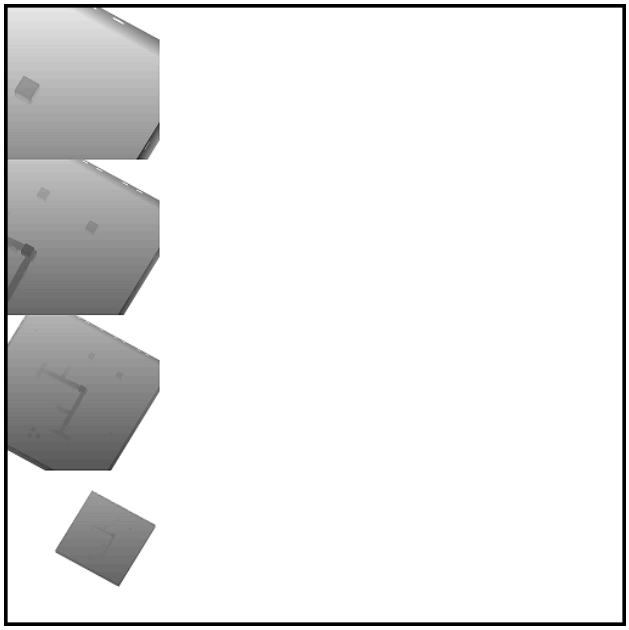
> **NOTE**
>
> This doesn't work in both directions. If you set a value other than the `DefaultValue` in the Property Grid, Game Studio saves the value in the asset and overrides the default value at runtime.

Lights don't cast shadows

If you've enabled shadows on a light in your scene, but it isn't casting shadows, make sure you have enough space in the shadow atlas. You might need to reduce the size of the shadows in the properties of your light components to create room.

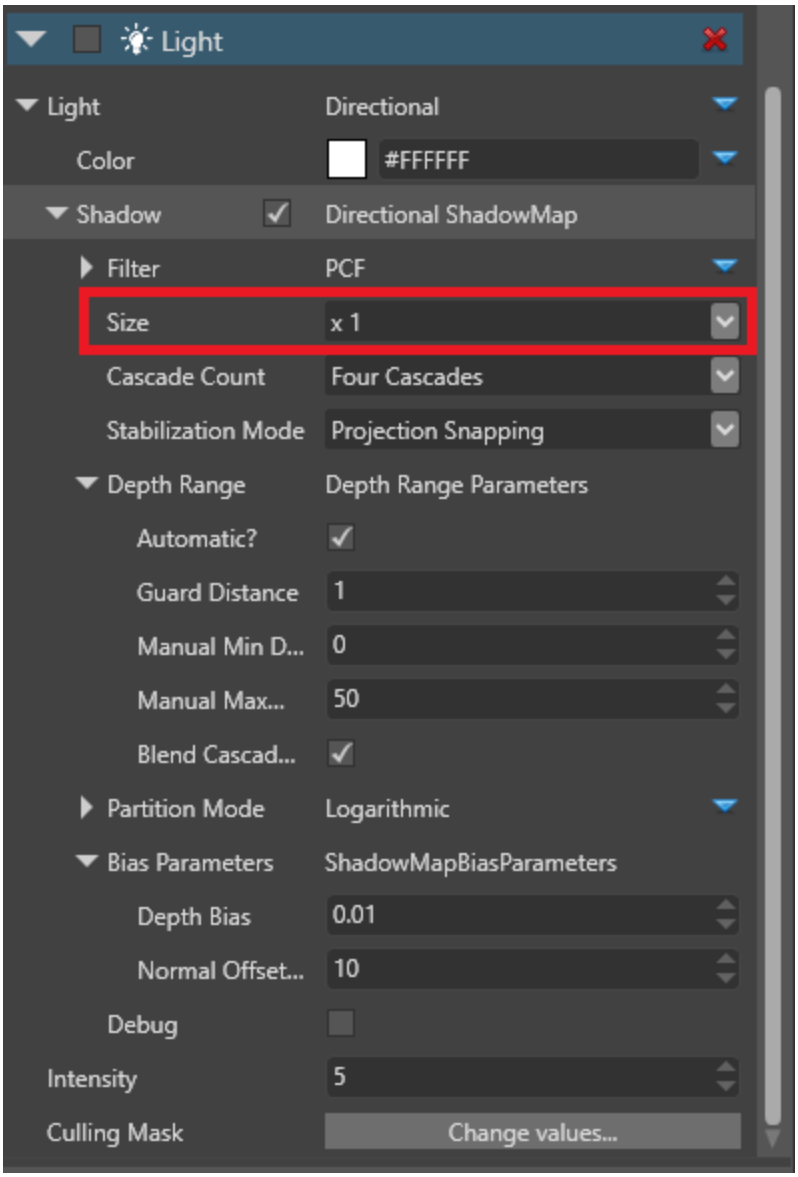
For more information about shadows and the shadow atlas, see Graphics - Shadows.

Shadow atlas comparison

Size: 2x	**Size: 1x**
	
This light source uses the entirety of the shadow atlas. This means other lights won't cast shadows, as there's no room left in the atlas.	This light source uses one quarter of the shadow atlas. The rest can be allocated to other light sources.

Reduce the shadow size

1. In the Scene Editor, select an entity with a light that casts a shadow.
2. In the **Light** component properties, under **Shadow > Size**, reduce the size of the shadow using the drop-down menu.



Alternatively, disable shadows on the light entirely by clearing the **Shadows** checkbox.

Repeat these steps for as many light entities as you need to create space in the shadow atlas.

See also

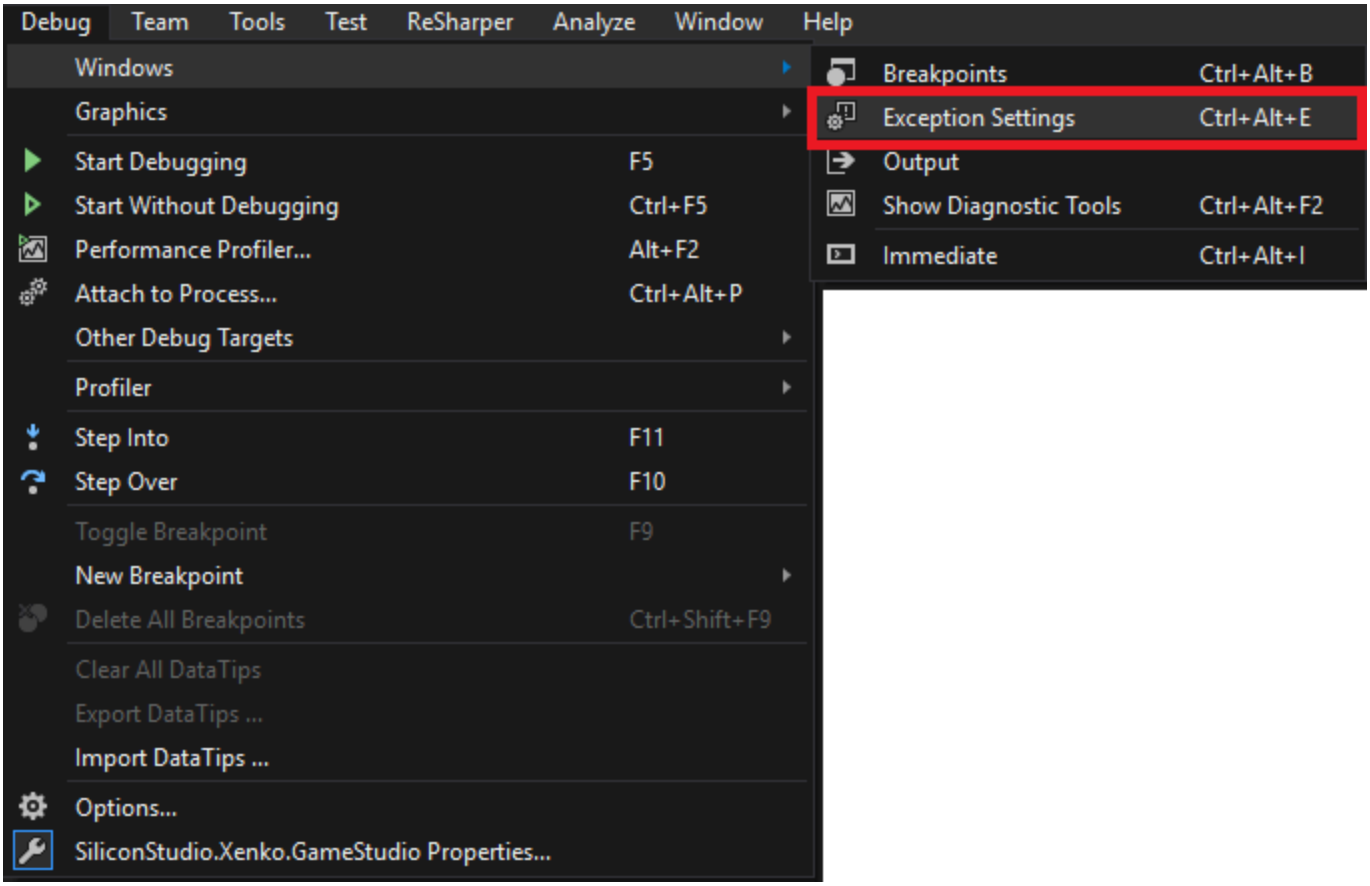
- Graphics — Shadows
- Graphics — Directional lights

Full call stack not available

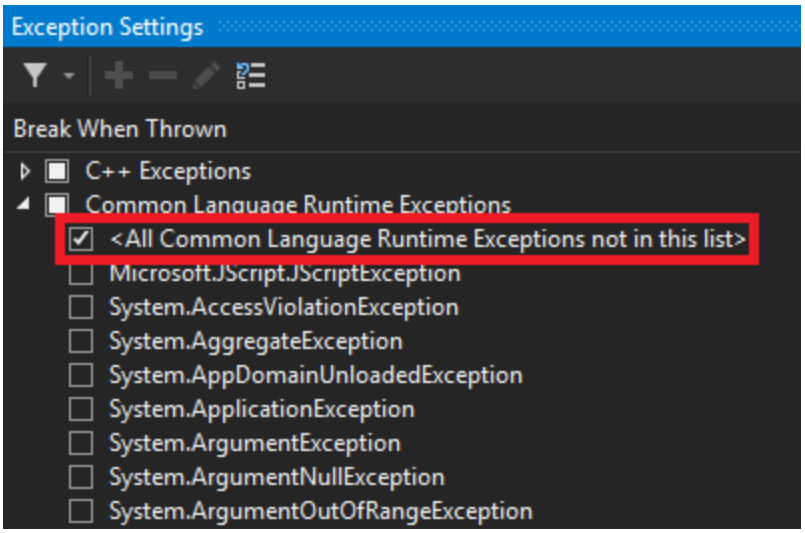
Depending on your Visual Studio settings, when an exception is thrown in Stride, Visual Studio might only show the call stack from the Stride runtime .DLL files or .NET framework assemblies, not user code.

To break as soon as an exception is thrown, add additional conditions to the Visual Studio **Exception Settings**.

1. In the Visual Studio toolbar, under the **Debug** menu, select **Windows > Exception Settings**.



2. Expand **Common Language Runtime Exceptions** and select **All Common Language Runtime Exceptions not in this list**. You might need to select other conditions too.



> **TIP**
>
> To restore the default list of exceptions, right-click and select **Restore Defaults**.

For more information about managing exceptions in Visual Studio, see Manage exceptions with the debugger in Visual Studio in the Microsoft Visual Studio documentation.

Error: "A SceneCameraRenderer in use has no camera assigned to its [Slot]. Make sure a camera is enabled and assigned to the [Slot]."

> **NOTE**
>
> In earlier versions of Stride, this error message was: "A SceneCameraRenderer in use has no camera set. Make sure the camera component to use is enabled and has its [Slot] property correctly set."

This error means there's no camera available for the scene renderer to use. This has several possible causes:

- there's no enabled camera
- the camera is set to the wrong camera slot
- there are multiple enabled cameras assigned to the same camera slot

Fix

If you create your camera components in Game Studio, make sure:

- the camera slots are set to the **Main** slot (see Graphics — Camera slots)
- only the initial camera is enabled

If you create your camera components in code, make sure you retrieve the correct slot from the graphics compositor. Use:

```
var camera = new CameraComponent();
camera.Slot = SceneSystem.GraphicsCompositor.Cameras[0].ToSlotId();
```

To change the camera at runtime, toggle the `Enabled` property.

> **NOTE**
>
> Make sure you:
>
> - always have at least one enabled camera
> - don't have multiple cameras enabled and assigned to the same slot at the same time

See also

- Graphics — Camera slots
- Graphics — Cameras

NuGet.Config misconfiguration

Unable to resolve Stride.GameStudio

If you encounter the error:

```
Error: Unable to resolve 'Stride.GameStudio (>= 4.2.0.2381)' for 'net8.0-windows7.0'.

Exception

Could not restore package Stride.GameStudio
at Stride.Core.Packages.NugetStore.InstallPackage(String packageId, PackageVersion version,
IEnumerable`1 targetFrameworks, ProgressReport progress)
at Stride.LauncherApp.ViewModels.PackageVersionViewModel.<>c__DisplayClass55_0.
<b__0>d.MoveNext()
```

This usually means your NuGet package sources are misconfigured.

How to fix

1. Locate your NuGet configuration file:
   - `C:\Users\YourUserName\AppData\Roaming\NuGet\NuGet.Config`
2. Make a backup of the file before editing.
3. Open the file and ensure it contains the following lines:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
    <add key="Stride Dev" value="C:\Users\YourUserName\AppData\Local\Stride\NugetDev" />
  </packageSources>
</configuration>
```

4. Save the file and try restoring packages again.

The launcher only finds 3.x versions, not 4.x

If the install program (Launcher) only shows Stride 3.x versions and not 4.x, your `NuGet.Config` file is likely missing the `nuget.org` package source.

How to fix

1. Open your NuGet configuration file:

- `C:\Users\YourUserName\AppData\Roaming\NuGet\NuGet.Config`

2. Make a backup before editing.
3. Ensure your `NuGet.Config` contains at least the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
  </packageSources>
</configuration>
```

4. Save the file and restart the Launcher.

You should now see the latest Stride 4.x versions available for installation.

NuGet.Config misconfiguration

Unable to resolve Stride.GameStudio

If you encounter the error:

```
Error: Unable to resolve 'Stride.GameStudio (>= 4.2.0.2381)' for 'net8.0-windows7.0'.

Exception

Could not restore package Stride.GameStudio
at Stride.Core.Packages.NugetStore.InstallPackage(String packageId, PackageVersion version,
IEnumerable`1 targetFrameworks, ProgressReport progress)
at Stride.LauncherApp.ViewModels.PackageVersionViewModel.<>c__DisplayClass55_0.
<b__0>d.MoveNext()
```

This usually means your NuGet package sources are misconfigured.

How to fix

1. Locate your NuGet configuration file:
   - `C:\Users\YourUserName\AppData\Roaming\NuGet\NuGet.Config`
2. Make a backup of the file before editing.
3. Open the file and ensure it contains the following lines:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
    <add key="Stride Dev" value="C:\Users\YourUserName\AppData\Local\Stride\NugetDev" />
  </packageSources>
</configuration>
```

4. Save the file and try restoring packages again.

The launcher only finds 3.x versions, not 4.x

If the install program (Launcher) only shows Stride 3.x versions and not 4.x, your `NuGet.Config` file is likely missing the `nuget.org` package source.

How to fix

1. Open your NuGet configuration file:

- `C:\Users\YourUserName\AppData\Roaming\NuGet\NuGet.Config`

2. Make a backup before editing.
3. Ensure your `NuGet.Config` contains at least the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
  </packageSources>
</configuration>
```

4. Save the file and restart the Launcher.

You should now see the latest Stride 4.x versions available for installation.

Glossary

- Stride: A C# game engine for creating 2D/3D games and visualizations.
- Stride Community Toolkit: A collection of extensions and helpers for the Stride engine.
- Code-Only: A feature of the toolkit that allows you to create a game without using the Game Studio.

Animation terms

- Additive animation: Blends an animation additively onto a base pose.
- Animation: Moving or changing values of objects over time, typically applied to models, lights, or component properties.
- Animation clip: A timeline containing curves and keyframes that animate values over a defined duration.
- Animation curve: A function describing how a value changes over time; composed of keyframes and an interpolation mode.
- Blend tree: Logic that blends multiple animations based on parameters.
- Custom blend trees: Complex blending graphs that combine multiple clips.
- Import animations: Bring animations from DCC tools into Stride.
- Keyframe: A time/value sample on a curve.
- Model node links: Link animation channels to specific nodes in a model.
- Preview animations: Inspect and debug clips in the editor.
- Procedural animation: Animation generated at runtime via code rather than authored keyframes.
- Set up animations: Assign clips and configure animation components.
- Skeleton / rig: A hierarchy of bones used to pose and animate skinned models.

Asset terms

- Archetype: A template for asset properties that other assets can inherit from.
- Asset: Content used by the game (models, textures, materials, scripts, etc.) managed by the asset pipeline.
- Asset URL: The path used in code to load an asset, for example with `Content.Load<T>("MyFolder/MyAsset")`. See Create a model from code and Create a script.
- Asset bundles: Groups of assets packaged for deployment/streaming.
- Asset control: Control asset loading, references, and lifetime.
- Build pipeline: Compiles assets into runtime-ready formats.
- Prefab: A reusable entity hierarchy that can be instantiated at design time or runtime.
- Root asset: An asset included in the build so it is available at runtime. See Manage assets.

Audio terms

- Audio emitter: A component that plays sounds in 3D space.

- Audio listener: Represents the listener position/orientation used for spatial audio.
- Doppler effect: Pitch shift proportional to relative velocity between emitter and listener.
- Global audio settings: Project-wide audio configuration.
- Non-spatialized audio: 2D audio that does not attenuate with distance.
- Spatialized audio: Audio that changes with distance and direction; can use HRTF for realism.
- Stream audio: Play long audio from disk instead of preloading.

Core terms

- Delta Time: The time elapsed between frames, used for frame-independent movement.
- Frame rate: The number of frames rendered per second (FPS).
- GameTime: Time information passed to the game loop.
- Vector3: A 3D vector representing a point or direction.

Editor terms

- Add entities: Create new entities in a scene.
- Archetypes: Share default properties across assets.
- Asset View: Panel used to manage assets in your project.
- Create assets: Create and organize project assets.
- Game settings: Project-wide defaults such as the graphics compositor and rendering options.
- Graphics compositor editor: Node-based editor to configure the rendering pipeline.
- Manage entities: Select, group, and organize entities.
- Manage scenes: Open, set defaults, and load/unload scenes.
- Navigate the Scene Editor: Viewport navigation and controls.
- Property Grid: Panel used to view and edit properties of selected entities/components.
- Scene Editor: The 3D/2D viewport for arranging entities in a scene.
- Splash screen: Configure startup visuals.
- Use assets: Reference and place assets in scenes.
- World units: Scene scale conventions (unit size, conversions).

General terms

- ECS (Entity Component System): The architecture where entities host components and systems process them.
- Entity: An object in the scene that can represent anything from a 3D model to a camera or light and aggregates multiple components.
- Entity component: A piece of data/behavior attached to an entity (for example Transform, Model, Light).
- Game: A Stride application or interactive experience running the game loop.
- Game loop: The main loop that updates and renders the game each frame.
- Scene: The container for entities, defining the game world or environment.

- Transform: Defines an entity's position, rotation, and scale in the scene.

Graphics terms

- 3D primitive: A basic geometric mesh such as a cube, sphere, or cylinder.
- Camera: Captures the scene from a perspective or orthographic view.
- Camera slot: Binds scene cameras to the graphics compositor.
- Debug renderers: On-screen debug drawings (lines, shapes, labels).
- Effects and shaders: Shader authoring and customization for rendering.
- Graphics compositor: Organizes how scenes are rendered, including cameras, render targets, and post effects.
- Material: Defines how a surface is shaded (color, textures, and lighting response).
- Mesh: Geometry to render, defined by vertex and optionally index buffers.
- Model: A container of one or more meshes and a list of materials.
- Pipeline state: Input layout, primitive type, shaders, and states used for a draw.
- Post effects: Image effects (bloom, color grading, etc.) applied after scene rendering.
- Procedural Model: A model built at runtime from generated meshes (e.g., via built-in geometric primitives or custom vertex/index buffers) rather than imported as an asset. See also Create a model from code and Draw vertices.
- Render groups and masks: Filter which objects render in each pass.
- Render texture: A texture that receives the output of a camera or rendering pass.
- Resource binding: Bind textures/buffers and constants to shaders.
- Scene renderers: Components that draw the scene (for example forward renderer, debug renderer).
- Skybox: A background using a cubemap or 360° panorama to simulate distant surroundings.
- SpriteBatch: Batched 2D drawing of sprites.
- SpriteFont: Fonts for drawing text in 2D/3D with SpriteBatch.
- Vertex buffer / index buffer / primitive type: Low-level geometry data and topology used to render meshes.

Post-effect terms

- Ambient occlusion: Screen-space occlusion to enhance contact shadows.
- Anti-aliasing: Techniques to smooth jagged edges.
- Bloom: Glow effect from bright areas.
- Bright filter: Extract bright regions for bloom/compositing.
- Depth of field: Simulates camera focus blur.
- Lens flare: Light artifacts caused by bright sources.
- Light streaks: Streaking effect from bright lights.
- Local reflections: Screen-space reflections for shiny surfaces.
- Tone mapping: Map HDR values to displayable range.

Input terms

- Gamepads: Controller input support and mapping.
- Gestures: Touch gesture detection.
- Keyboards: Keyboard input handling.
- Mouse: Mouse input handling.
- Pointers: Unified pointer input across devices.
- Sensors: Device sensors like accelerometers and gyroscopes.
- Virtual buttons: Map multiple physical inputs to actions.

Lighting terms

- Ambient light: Uniform lighting applied across the scene.
- Directional light: Light from a distant source with parallel rays (e.g., sunlight).
- Light probes: Sampled lighting information used for indirect lighting.
- Point light: Light that radiates from a point in all directions.
- Shadows: Techniques to render occlusion from lights.
- Skybox light: Image-based lighting derived from a skybox.
- Spot light: Light that radiates within a cone.
- Voxel cone tracing GI: Real-time global illumination technique.

Navigation terms

- Dynamic navigation: Runtime updates to navigation data.
- Navigation bounding boxes: Volumes defining navigation constraints.
- Navigation components: Components that enable navigation behaviors.
- Navigation groups: Layers/groups to partition navigation.
- Navigation meshes: Walkable areas computed from scene geometry.

Particles terms

- Emitters: Particle sources that spawn particles over time.
- Initializers: Set initial particle properties when spawned.
- Materials (particles): Materials used for rendering particles.
- Ribbons and trails: Ribbon/trail rendering driven by particles.
- Shapes: Spawn shapes for emitters (sphere, cone, box, etc.).
- Spawners: Control particle spawn rates and bursts.
- Updaters: Modify particle properties over their lifetime.

Performance terms

- Profiler: A tool that monitors performance metrics like frames per second (FPS) and memory usage.

Physics terms

- Bepu Physics: The preferred physics engine for Stride, providing advanced Bepu physics simulation capabilities.
- BodyComponent: The runtime component for dynamic bodies (API reference).
- Character: Specialized collider and controller for player or NPC movement.
- Collidable: Base component category for physics objects (statics, bodies, characters).
- Collider shapes: Primitive and complex shapes that define collision bounds.
- Collision layers and groups: Settings that filter which objects can collide with each other.
- Constraints: Joints and limits connecting physics objects.
- Impulse: An instantaneous change in momentum applied to a rigid body.
- Kinematic body: Animated or code-driven colliders that affect others but are not affected by forces.
- Movement
  - Non-physical movement: Moving entities by directly changing their position without physics interactions.
  - Physics-based movement: Moving entities using the physics engine to simulate realistic interactions.
- Physics engine: Simulates physical interactions between entities in the scene.
- Physics queries: Raycasts and overlaps to test collisions.
- Physics update: Fixed timestep and update order considerations.
- Rigid body: Dynamic colliders affected by forces (gravity, collisions).
- Static: Non-moving colliders (terrain, walls) that other objects collide with.
- Trigger: A collider that raises events when other objects overlap without blocking them.

Scripting terms

- Asynchronous script: Runs a task-like Execute method; can await operations and return to the main thread with `await Script.NextFrame()`.
- Best practices: Recommended coding patterns for Stride projects.
- Camera controller: A script enabling camera movement using input.
- Content.Load: Load assets at runtime via the content system.
- Debugging: Inspect and troubleshoot scripts at runtime.
- Entity.GetOrCreate: Retrieve or add a component to the entity.
- Events: Script event patterns (subscribe, raise, handle).
- Gizmos: Draw editor helpers for custom components.
- Preprocessor variables: Conditional compilation for scripts.
- Public properties and fields: Expose script fields to the editor.
- SceneSystem.SceneInstance.RootScene: Access the active scene instance.
- Scheduling and priorities: Control script execution order and timing.
- Startup script: Runs when added/removed at runtime; used to initialize or tear down game elements.
- Synchronous script: Has an Update method that runs every frame on the main thread.
- Update method: A callback that runs every frame to update game logic.

Sprites terms

- Nine-slice (sprite borders): Scalable UI frames using sprite borders.
- Sprite: A 2D image used in UI or scenes.
- Sprite sheet: Texture atlas containing multiple sprites.

UI terms

- Add a UI to a scene: How to display UI in your scene.
- Layout system: Mechanism that arranges UI elements (stack panels, grids, anchors).
- UI editor: Tooling to author and preview UI within Game Studio.
- UI libraries: Reusable sets of UI controls and styles.
- UI pages: Screens composed of UI elements.

Virtual reality terms

- Enable VR: Enable VR support for your project.
- Overlays: Display UI or HUDs in VR using overlays.
- Preview a scene in VR: Test your scene with a VR headset.