→ E-BOOK

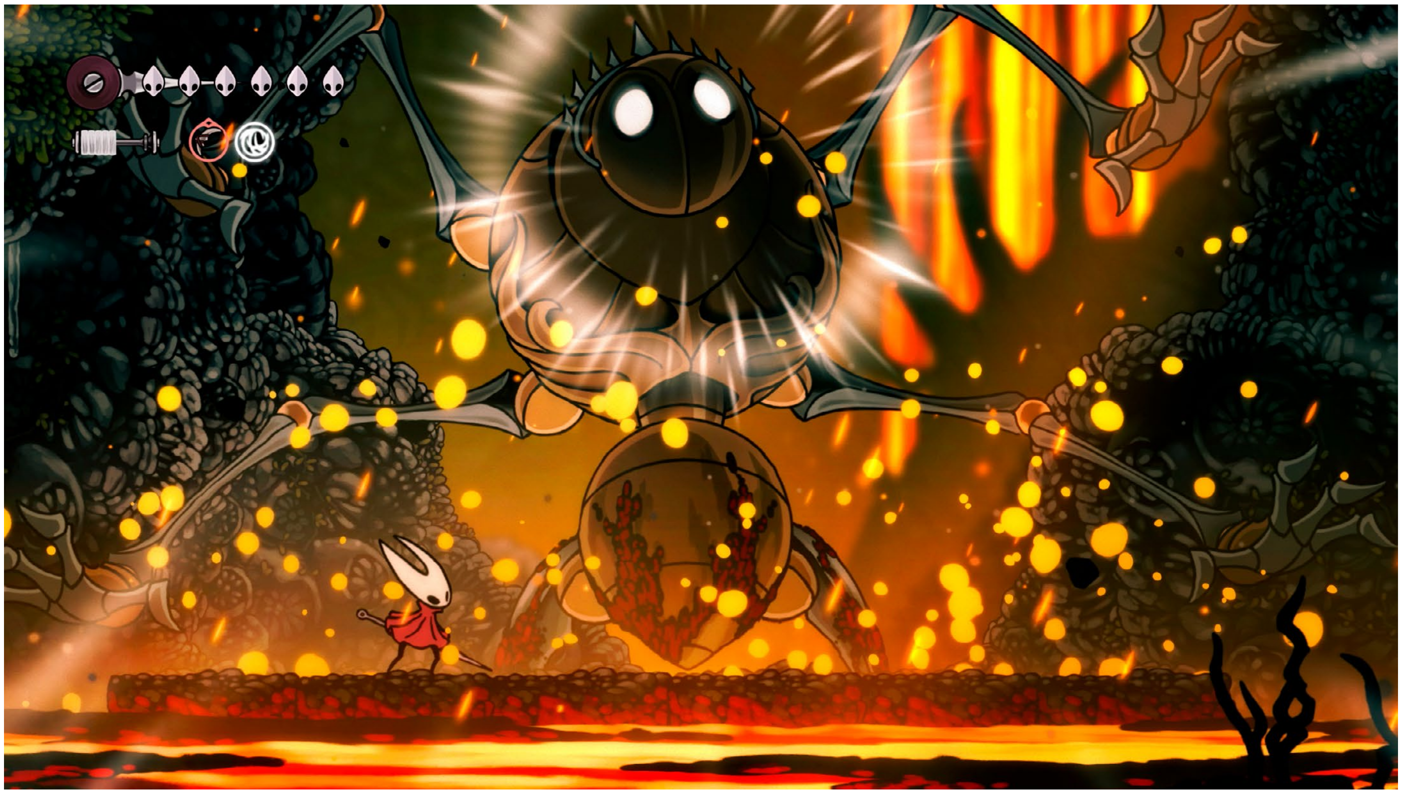
2D game art, animation, and lighting for artists

(Unity 6.3 LTS edition)

Unity®

Contents

Introduction


Hollow Knight: Silksong, by Team Cherry, available for PCs and consoles, is a 2D game made with Unity.

The evolution of gaming platforms, Unity's 2D toolset, and other graphics development software make it possible today to create 2D games with real-time lights, high-resolution textures, and an almost unlimited sprite count. The flatness of 2D graphics frees artists to create a game in any art style that looks great on any device.

This e-book is for developers and artists with intermediate Unity experience who want to make a commercial 2D game, whether they work independently or with a team.

It's our most comprehensive 2D game development guide, written with key contributions from Unity experts to help creators get the most out of the Unity 2D toolset.

Unity 6.3 LTS is the current production-ready Unity release and our 2D guide is now updated with explanations of workflows of its 2D features and systems. Your takeaways from this e-book can also be applied to future Unity versions.

You'll learn about the techniques and workflows for setting up a 2D project; getting your sprites into Unity from DCC software, as well as creating sprites in Unity; creating sophisticated light, shadow, and visual effects; designing levels with Tilemap 2D and other features; animation, and finally, a handy list of optimization tips.

Many of the feature capabilities in the e-book are illustrated using assets and scenes from Unity's recent 2D samples: *Dragon Crashers*, *Happy Harvest*, *Gem Hunter Match*, and *Bunny Blitz,* which blends 2D and 3D (available in early 2026). They're available for free from the Unity Asset Store and we encourage you to download them to use as learning references alongside this e-book.

We hope you enjoy this guide and wish you the best of luck with your game development. The future is bright for 2D creators.

Contributors

Original author

Jarek Majewski is a professional 2D artist, tech artist and self-taught Unity developer with extensive C# scripting skills. He has created art for mobile and console games. Besides his professional work at game studios he is working on his own game *Ultimate Action Hero*, currently in development and which placed second in the 2019 Unity 2D Challenge.

Jarek was the art director and artist for the Unity 2D samples *Dragon Crashers*, *Happy Harvest*, *Gem Hunter Match* and *Bunny Blitz*.

Unity contributors

Eduardo Oriz led the production of this guide. He is a senior content marketing manager at Unity with many years of experience working with game development teams. He works closely with the 2D tools team, and has a broad understanding of what Unity offers to game developers and studios.

Rus Scammell is the product manager on Unity's 2D development team. Rus has over 15 years' experience in game and software development. He uses his extensive knowledge of games technology to ensure that Unity 2D tools and workflows are accessible to artists, programmers, and game designers.

Melvyn May is the main developer of Unity's 2D physics system, and a key collaborator in the creation of the *Bunny Blitz* and 2D physics samples. Melvy is active in the Unity community, often seen in Discussions threads answering users' questions.

The rest of the 2D development team at Unity contributed by reviewing and validating this guide and are excited to see what 2D games users will create in Unity, in 2026 and beyond.

2D in Unity: The essentials

How to open a new 2D project

Use these install steps to follow along with this e-book.

1. Install the latest version of Unity 6 from the Unity Hub. At the time of writing this e-book Unity 6.3 is the latest official release.
2. In the Hub, create a new project with the Universal 2D template; this is the template that uses the Universal Render Pipeline (URP). In any version of Unity from 2021 and later, the 2D Renderer is already configured to work with URP.

A number of packages are included in the 2D template:

— **2D Animation**: For implementing native skeletal animation at runtime

— **2D Aseprite Importer**: Enables the import of .aseprite files from the pixel art creation tool Aseprite

— **2D Common:** Contains the code used by several 2D packages

— **2D Pixel Perfect**: Provides a camera component to ensure crisp pixel art at different resolution and ratios, with features like snapping, upscale rotations, etc.

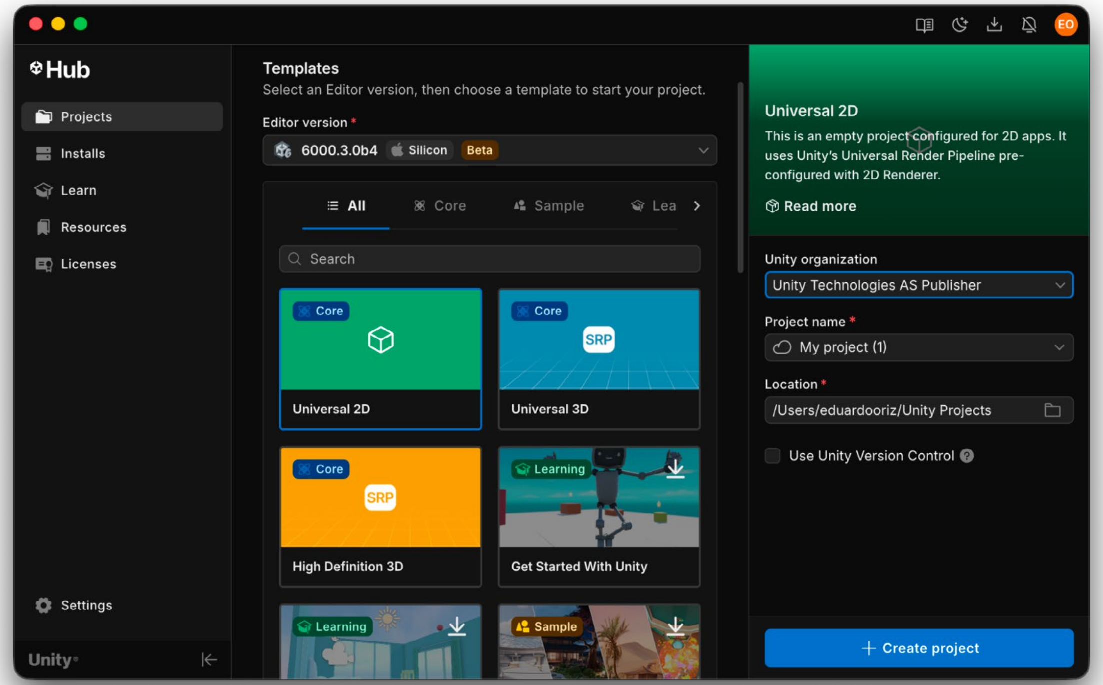
— **2D PSD Importer**: Provides support for layered .psd and .psb files, convenient for multisprite character animation

— **2D Sprite**: The Sprite Editor window used to configure sprite properties.

— **2D SpriteShape**: For creating sprite tiling along a shape's outline that automatically deforms and swaps sprites based on the angle of the outline

— **2D Tilemap Editor**: For creating and editing tilemaps to populate large grid-based worlds

— **2D Tilemap Extras**: Contains extra scripts for use with 2D Tilemap, like custom tiles and brushes



Templates
Select an Editor version, then choose a template to start your project.

Editor version * 6000.3.0b4 Silicon Beta

Universal 2D
This is an empty project configured for 2D apps. It uses Unity's Universal Render Pipeline pre-configured with 2D Renderer.

Unity organization: Unity Technologies AS Publisher

Project name *: My project (1)

Location *: /Users/eduardooriz/Unity Projects

The Unity Hub window

The Universal 2D template comes with several project settings that are optimized for a 2D game:

— The images are imported as sprites and set to Sprite mode.

— The Scene view is set to 2D.

— The default scene includes a Global Light 2D.

— The default Sprite Renderer material is Sprites-Lit-Default.

— The camera's default position is at 0, 0, –10.

— The camera is set to Orthographic.

— In the Lighting window settings:

  — All Global Illumination is disabled.

  — Skybox material is set to none.

  — Ambient Source is set to Color.

URP for 2D games

URP is Unity's default renderer for 2D and 3D games. It's the successor to Unity's Built-In Render Pipeline, and is designed to be efficient for you to learn, customize, and scale to all Unity-supported platforms. In the Unity 6.x cycle, it offers the same functionality as the Built-In Render Pipeline, and in a number of areas, exceeds its quality levels and performance.

URP provides a graphics pipeline for 2D Lighting, allowing you to create 2D lights and lighting effects. It integrates with 2D renderers, such as the Sprite Renderer, Tilemap Renderer, and Sprite Shape Renderer, as well as 3D renderers like MeshRenderer or SkinnedMeshRenderer. It's also compatible with Shader Graph, VFX Graph, post-processing effects, and camera stacking.

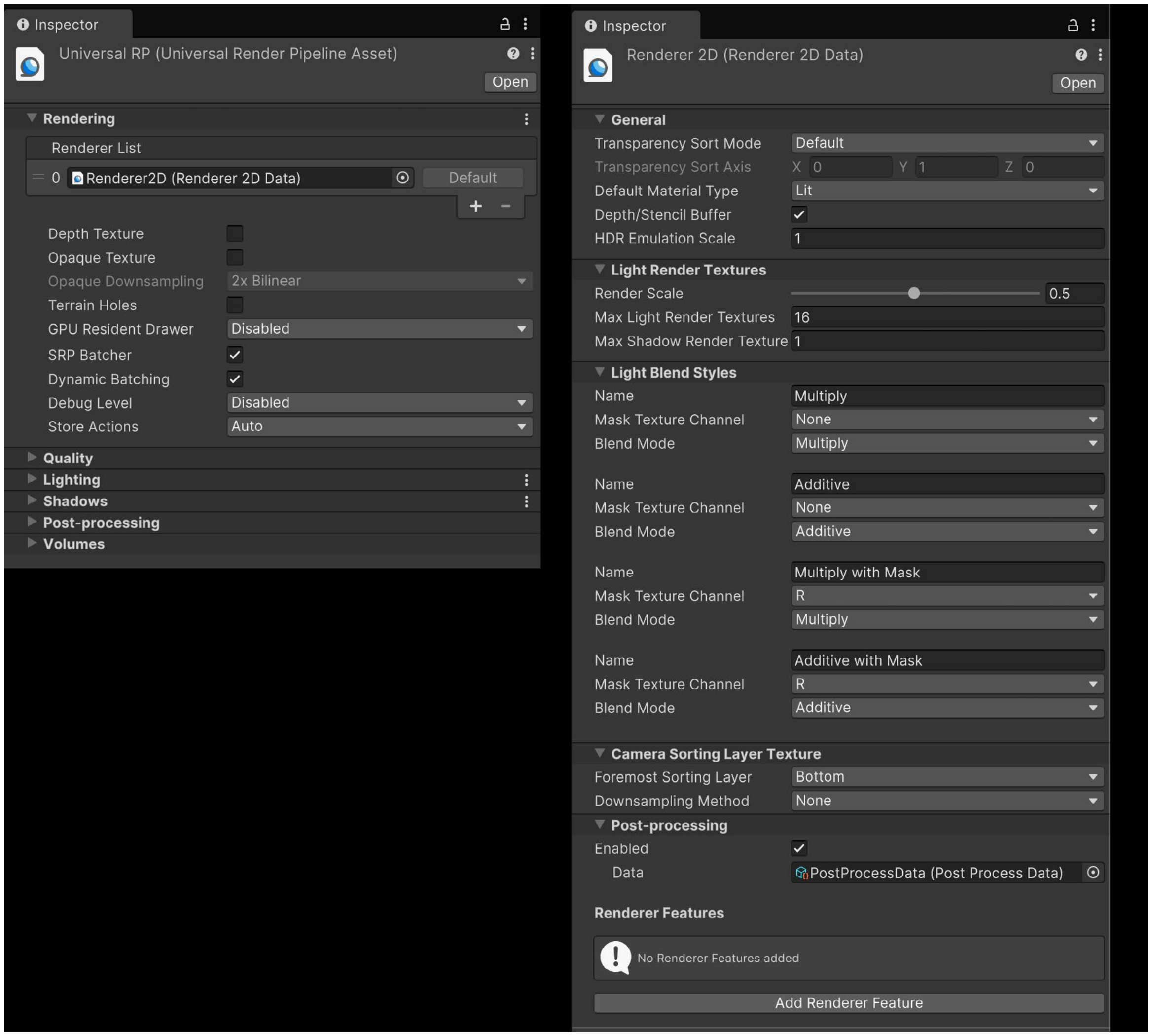
URP settings

URP base configurations are available in two assets in your project:

— The **Universal Render Pipeline Asset** provides settings for quality, lighting (3D), shadows (3D), and post-processing.

  If you want to have different settings for different hardware, you can configure Quality settings across multiple URP assets and switch them out as needed.
  The URP Asset is linked to the **Renderer Data Asset** via the **Renderer List**. When you create a new URP Asset it will have a Renderer List containing a single item, the Renderer Data Asset created at the same time, set as the default. You can add alternative Renderer Data Assets to this list.

— A **Renderer Data Asset** that you can use to filter the layers the renderer works on, and intercept the rendering pipeline to customize how the scene is rendered. It controls high-level rendering logic and passes for URP. It supports Forward and Deferred paths, and the 2D Renderer that enables features such as 2D lights, shadows, and light blend styles. You can even extend URP to create your own renderers.



Inspector
Universal RP (Universal Render Pipeline Asset)

Rendering

Renderer List
0 Renderer2D (Renderer 2D Data) — Default

Depth Texture	☐
Opaque Texture	☐
Opaque Downsampling	2x Bilinear
Terrain Holes	☐
GPU Resident Drawer	Disabled
SRP Batcher	✓
Dynamic Batching	✓
Debug Level	Disabled
Store Actions	Auto

Quality
Lighting
Shadows
Post-processing
Volumes

Inspector
Renderer 2D (Renderer 2D Data)

General

Transparency Sort Mode	Default
Transparency Sort Axis	X 0 Y 1 Z 0
Default Material Type	Lit
Depth/Stencil Buffer	✓
HDR Emulation Scale	1

Light Render Textures

Render Scale	0.5
Max Light Render Textures	16
Max Shadow Render Texture	1

Light Blend Styles

Name	Multiply
Mask Texture Channel	None
Blend Mode	Multiply
Name	Additive
Mask Texture Channel	None
Blend Mode	Additive
Name	Multiply with Mask
Mask Texture Channel	R
Blend Mode	Multiply
Name	Additive with Mask
Mask Texture Channel	R
Blend Mode	Additive

Camera Sorting Layer Texture

Foremost Sorting Layer	Bottom
Downsampling Method	None

Post-processing

Enabled	✓
Data	PostProcessData (Post Process Data)

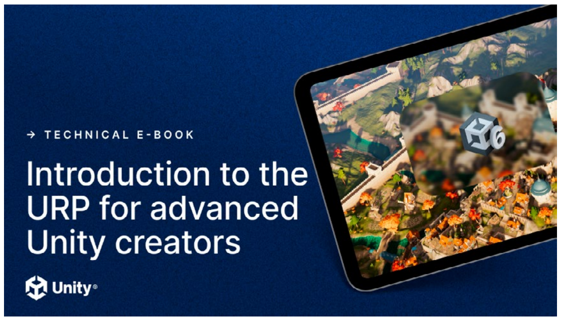
Renderer Features

No Renderer Features added

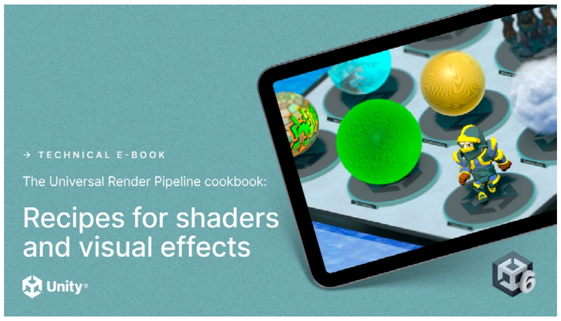
Add Renderer Feature

The URP configuration assets in a 2D project in Unity 6

This e-book does not cover all of the technical and visual possibilities with URP. To understand the full capabilities of URP, download these e-books from Unity:



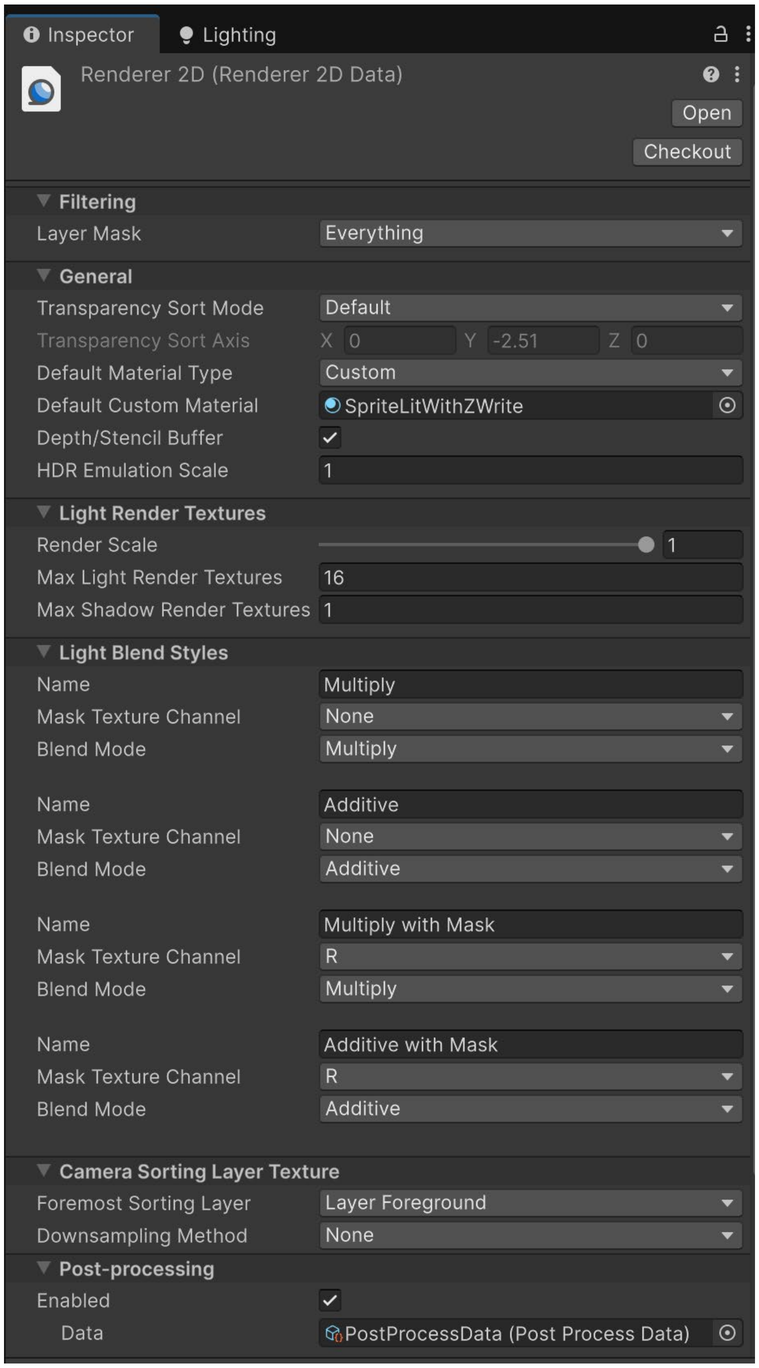
Get the e-book


→ TECHNICAL E-BOOK
The Universal Render Pipeline cookbook:
Recipes for shaders and visual effects
Unity

Get the e-book

2D Renderer settings

Here's an overview of the 2D Renderer settings in Unity 6.3; please see the documentation for further details.



Renderer 2D (Renderer 2D Data)

Filtering

Layer Mask	Everything

General

Transparency Sort Mode	Default
Transparency Sort Axis	X 0 Y -2.51 Z 0
Default Material Type	Custom
Default Custom Material	SpriteLitWithZWrite
Depth/Stencil Buffer	✓
HDR Emulation Scale	1

Light Render Textures

Render Scale	1
Max Light Render Textures	16
Max Shadow Render Textures	1

Light Blend Styles

Name	Multiply
Mask Texture Channel	None
Blend Mode	Multiply
Name	Additive
Mask Texture Channel	None
Blend Mode	Additive
Name	Multiply with Mask
Mask Texture Channel	R
Blend Mode	Multiply
Name	Additive with Mask
Mask Texture Channel	R
Blend Mode	Additive

Camera Sorting Layer Texture

Foremost Sorting Layer	Layer Foreground
Downsampling Method	None

Post-processing

Enabled	✓
Data	PostProcessData (Post Process Data)

— **Layer Mask:** The layers that you plan to render, by default it should be set to All

— **Transparency sort mode:** Will sort sprites based on the camera type that you use

— **Default Material Type:** Useful to define what material to add to any new 2D object, Lit if you use 2D lights, Unlit if not, or custom for your own material

— **Depth/Stencil Buffer:** Enabled by default, you can use depth effects if you make use of z-depth and perspective

— **HDR Emulation Scale:** Scale used to remap when using dynamic range

— **Light Render textures:** Scale and limits when creating 2D light textures; lower the values to increase performance when using 2D lights

— **Light Blend Modes:** Four possible blend styles to use on 2D lights; Add, multiply, subtract RGB values and affect only mask map channels

— **Camera Sorting Layer Texture:** Enables access in Shader Graph to the image created by the 2D renderer to apply effects to it, like a water reflection

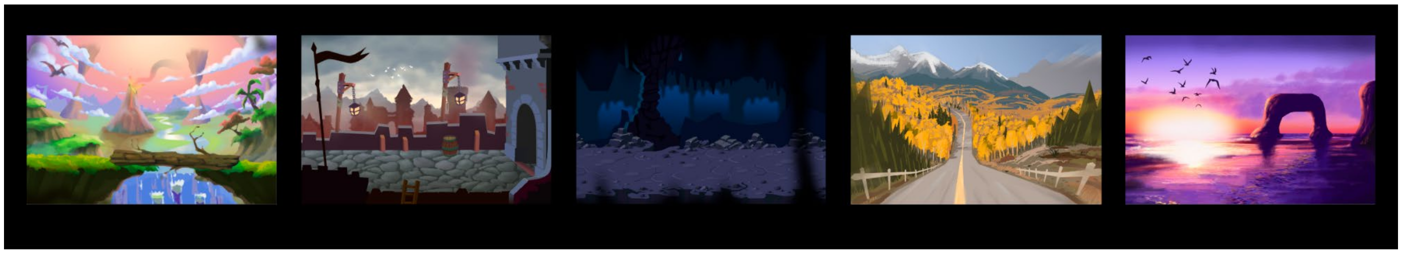
— **Post-Processing:** To add effects to the frame once processed

Preparing 2D art and assets

During the art concept phase, you'll need to make many decisions that will impact the technical side of your project.

In game design, it's common practice to mock up a screenshot or series of images to show the game's art style, action, and UI. Mockups are a quick way to determine if the concept is viable and get a sense of how the final game would look.

You can also explore multiple art directions with simple thumbnail mockups. These mockups would have simplified shapes and focus more on the overall look, camera angle, object sizes, color palette, and contrast. Using thumbnails is a great way to present different approaches to the team for testing and iteration.


Testing ideas by drawing simple thumbnails

A few things to consider when making a mockup:

— What will be the camera angle and perspective of your game?

— What will be the size of the player character in relation to the screen size of your target platform?

— Does your art style fit the target platform, theme, and audience? For example, will this style appeal to casual gamers, young players, strategy fans, and so on?

— How does the art style fit with the overall graphics approach?

    — If the gameplay requires fast reaction times, then it's a good idea to make elements such as the player character, enemies, and projectiles visible against the background at first glance.

    — Mobile games should be brighter and have higher contrast to be visible in the sunlight and smaller screens.

— What are the size, position, and visibility of UI elements?

Technical considerations

There are also important technical questions to consider in the mockup stage. These include:

— **Animation:** What elements will you animate with skeletal animation? Which ones will have frame-by-frame animation or be animated with a shader?

— **Environment:** Will you create it with a tilemap or sprite shapes? Or, do you want to place platform sprites in the scene manually? You can paint your mockup to mimic the look of these tools so that you can use the mockups directly in your game.

— **Sorting:** Plan your sprite sorting by grouping mockup layers similarly to how you anticipate the sorting will be done in Unity. It might take some time, but you'll avoid having to sort through potentially thousands of sprites later in production.

— **Lighting:** Will your sprites be lit with lighting and shadows painted onto them or with real-time lighting? One tip is to paint images with no lighting, then add shadows and lights on separate layers in your image-editing software. That way, you can always change the look of the lighting later in production if necessary.

Take the time in this stage to create assets that are as close as possible to what they will look like in the final game. This allows you to move from concept to game production more quickly.

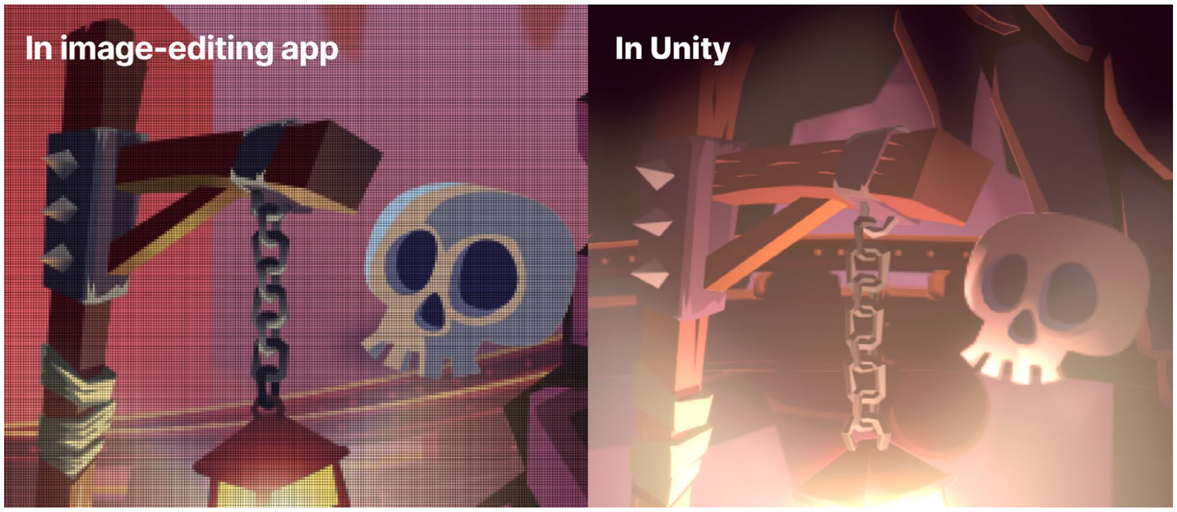
Original concept
Unity

A comparison of the original concept art and the final version of the Unity sample *Dragon Crashers*

Resolution of your assets

Unity's 2D tools evolved from an editor built initially for 3D games. As a result, there are some unique features. For example, 2D sprites in the Scene view aren't bound tightly to screen resolution. Sprites in Unity are textures drawn onto a mesh and are easily scalable. The camera in 2D games is also scalable and can zoom in and out as you want.

Therefore, creating 2D content in Unity requires a different approach to that of working in traditional raster graphics software such as Adobe Photoshop, Affinity Photo by Serif, GIMP, or Krita. In these conventional apps, you have a specific document canvas size available with a set resolution, and all layers are bound to this resolution. There is a 1:1 ratio between the pixel size of each layer and the document pixel size.



When drawing game assets, the pixel size is consistent, and the image looks great because it's always constrained to the pixel grid. However, in Unity, the sprites can have a different resolution. The camera zoom level will also affect the final look.

In Unity, however, screen and asset resolution are independent of each other, so you need to calculate the resolution of your sprites.

You start with your target platforms because the capabilities of the hardware determine the maximum resolution you can set.

For mobile devices, the resolution range is broad, but a safe assumption is 1920×1080, since this will allow you to target devices from low- to high-end.

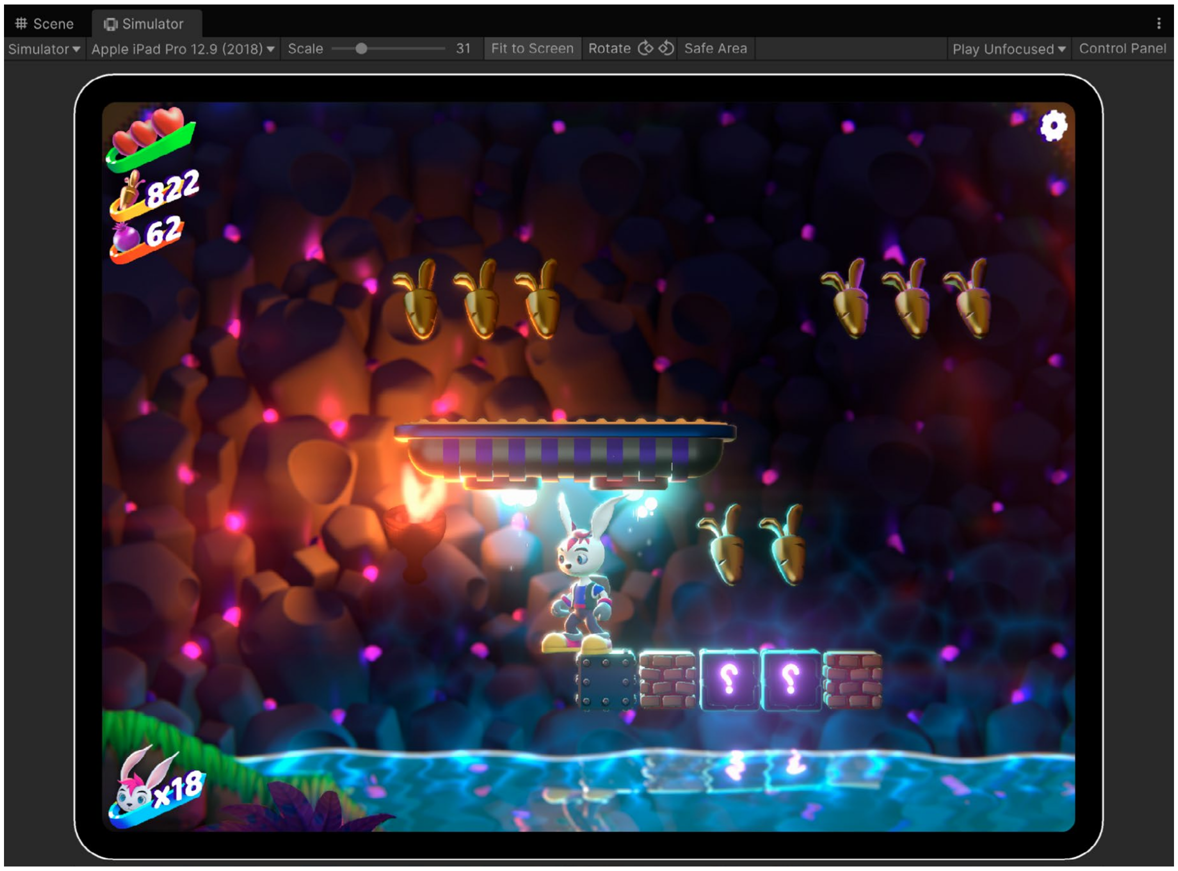
For PCs, the majority of desktop gamers use full HD (1920×1080), according to a survey from Steam. You might also want to consider supporting ultra-wide screens, making sure you test screen ratios like 21:9 to ensure the camera won't display unwanted areas of your level.

To target full HD or 4K, keep a couple of best practices in mind (these rules don't apply to pixel art):

- Paint art in the highest resolution for your target devices. Don't scale up raster art because it can result in pixelation and blurriness.
- Stick to one resolution for all assets, and scale down later if necessary to support lower-end devices.

One helpful trick is drawing your art at twice the size you need, then scaling it to 50% when you export to Unity. This technique will make your sprites look polished and crisp, and brush lines won't look so shaky. Don't overdo it on the details, since the art will be scaled down. This is a good trick for hiding the small imperfections resulting from hand-drawn art. Of course, if you want a hand-drawn visual style for your game, then don't try this.

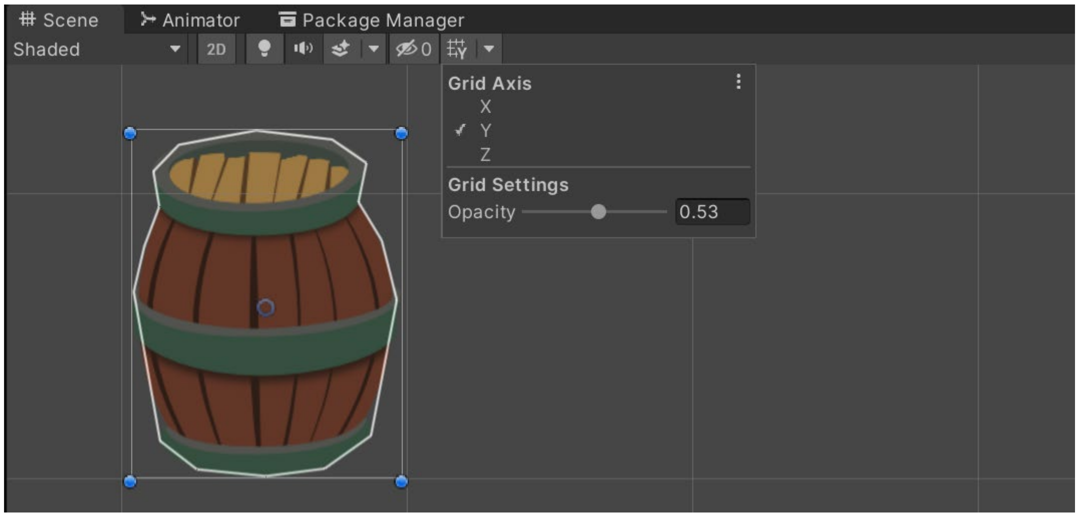
Once you've chosen the resolution, test the look of your art in the Game view to see how it will appear on target devices.



You can easily preview how the game looks at different ratios and resolutions in Game and Simulator view in the Editor.

To keep it simple, calculate sprite resolution in orthographic view instead of perspective. Set their scale to 1,1,1 and Z depth to 0 for easy visualization.

Use the Unity grid and units to maintain consistent placement and appearance of your sprites, as well as to calculate camera zoom and object sizes.



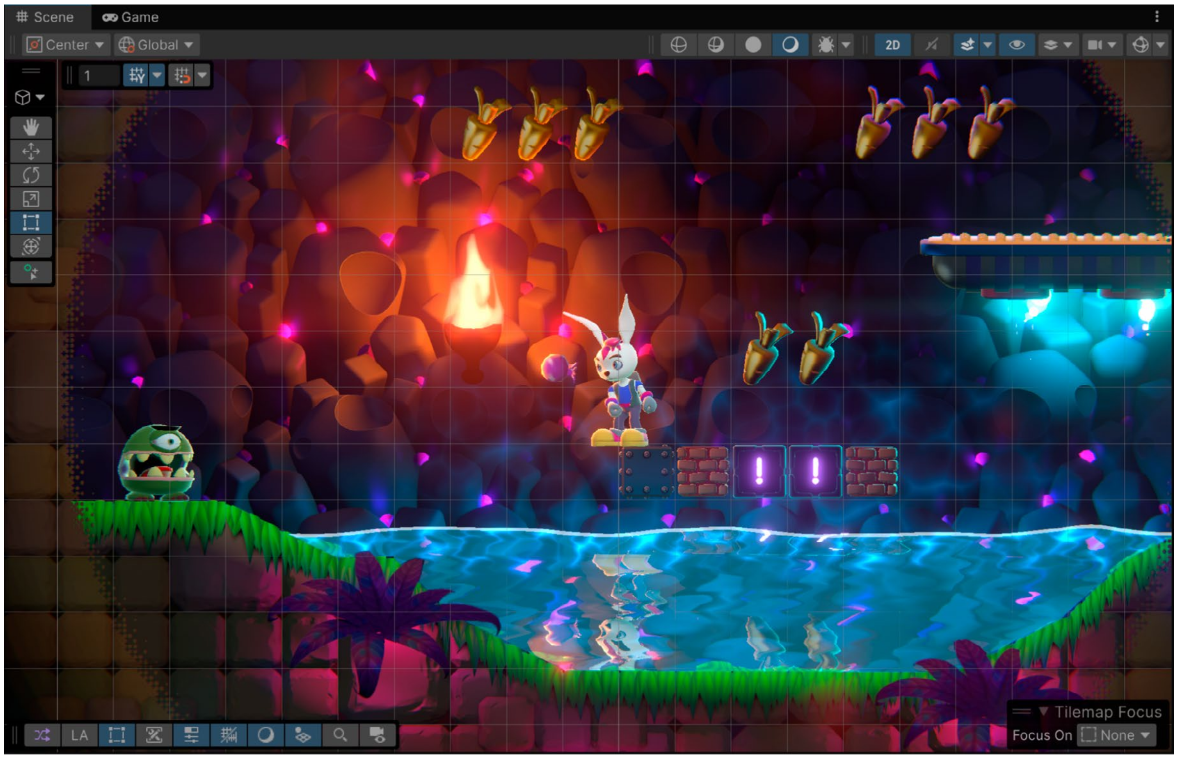
Unity's Grid in the Scene view

A Unity unit is visualized by the grid in the Scene view. Assume that 1 Unity unit is equal to 1 meter. Set your base sizes first, and keep them consistent across your game.

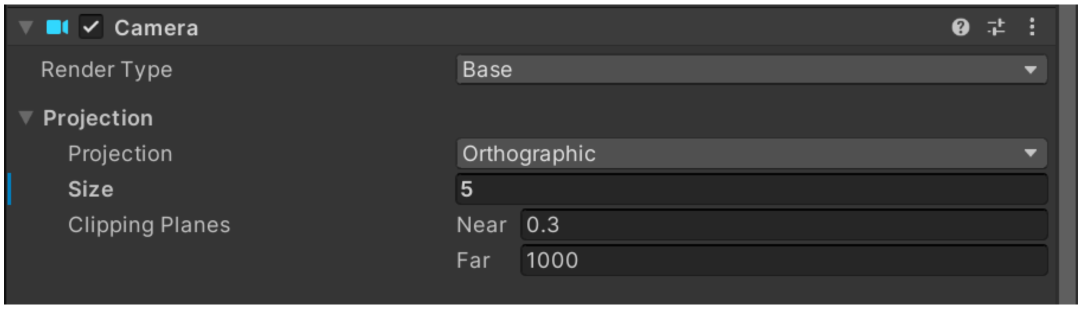
Start with the size of your player character. Keep the height of the player character between 0.5 and 2 units. If you're using Tilemap, set the tile size to 1 unit.

Characters and objects that are too small or large compared to other visual elements in a game will lead to strange transform numbers and issues with physics calculations.

Once you've established the sizes of your base objects in the main scene (Player character, enemies, collectibles, level hazards), choose your zoom level by setting the size property of the orthographic camera. After that, check the camera size value. When you multiply this value by 2, you'll get the camera vertical size in Unity units.



Determine early on how much gameplay area will be displayed on the screen, e.g., if you're developing a tight platformer, you'll want to lock this early on for game design and art preparation purposes.



Camera

Render Type	Base
Projection	
Projection	Orthographic
Size	5
Clipping Planes	Near 0.3
	Far 1000

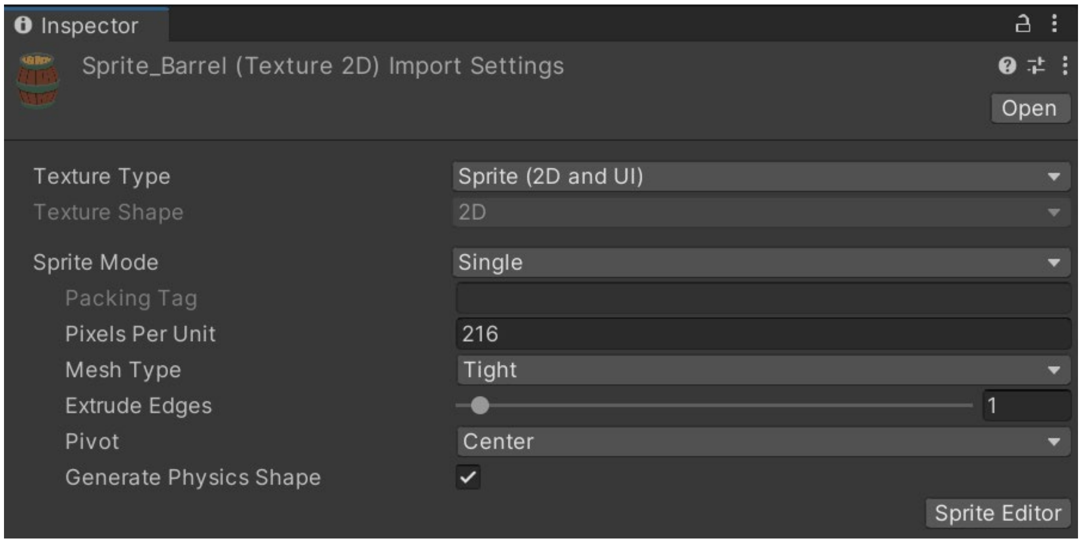
The size of the orthographic camera is expressed in units that denote the vertical size of the camera from the center to the top.

If the camera size equals 5, it means that it's displaying 5 units from the center of the screen to the top, with a total height then equal to 10 units. For example, if you target 4K resolution, then the screen (or camera) height is 2160 pixels. With a simple calculation, you can determine how many pixels per unit (PPU) your art needs:

2160 : 10 = 216

Max vertical resolution : (orthographic camera size *2) = sprites PPU

So every sprite in your game needs to have around 216 PPU to look good at a native 4K resolution.



Texture Type: Sprite (2D and UI)
Texture Shape: 2D
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 216
Mesh Type: Tight
Extrude Edges: 1
Pivot: Center
Generate Physics Shape: ✓

Setting the PPU for a sprite

That's a simple example. If you want your camera to zoom in and out, you'll need to account for that. If you set the maximum zoom percentage to size 3 in Orthographic view, the PPU needs to be 360 (2160 : (3 *2)).

If you use skeletal animation, set the resolution of sprites slightly higher than the suggested PPU. This is necessary because the meshes of the skeleton are rotated, stretched, and warped, sometimes to extremes, which can produce odd or poor-looking results.

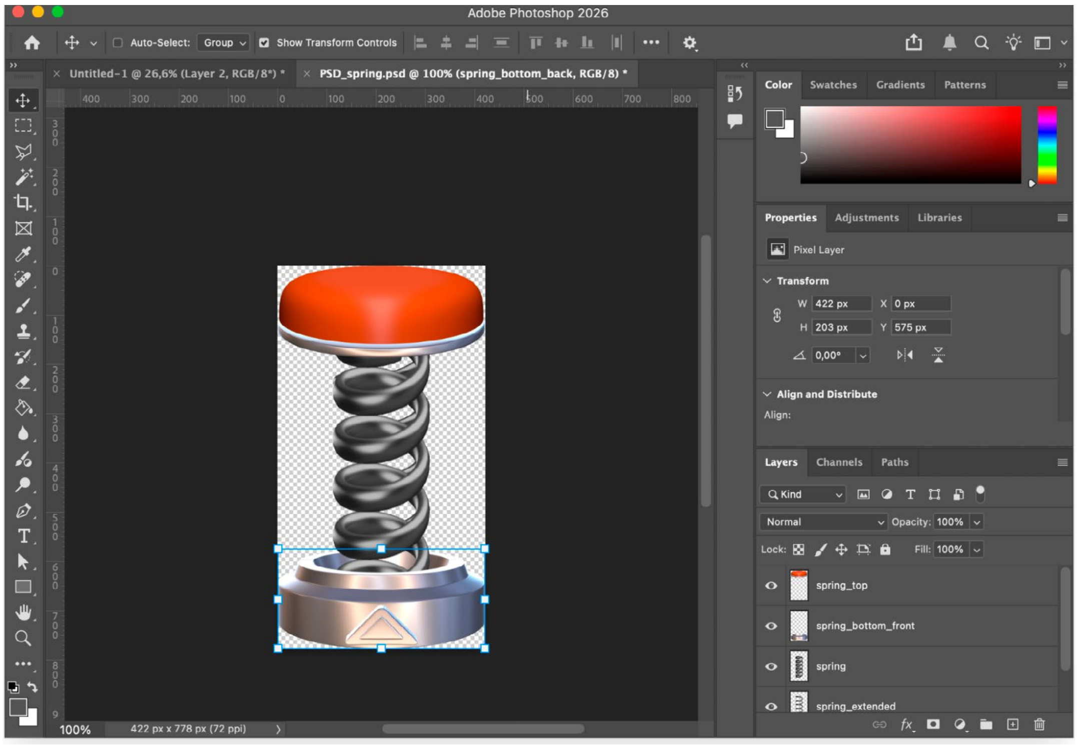
Test your game on devices to check how it looks. In many cases, the benefits of using high-resolution assets aren't apparent. Instead, you can allocate this drawing time and device memory to other, more essential, game elements like the main characters, visual effects, or UI. On mobile devices, game size is crucial, so check which assets can be scaled down and keep memory limitations in mind. Read more about 2D asset resolution in this blog post.

Get Photoshop PSD files into Unity

Exporting to PNG file format is the most common workflow for sprites, but you can also import PSD files directly into Unity. By default, Unity will flatten the PSD layers into one image if you don't have the 2D PSD Importer package installed, which is the case if you start your project from the 2D template.

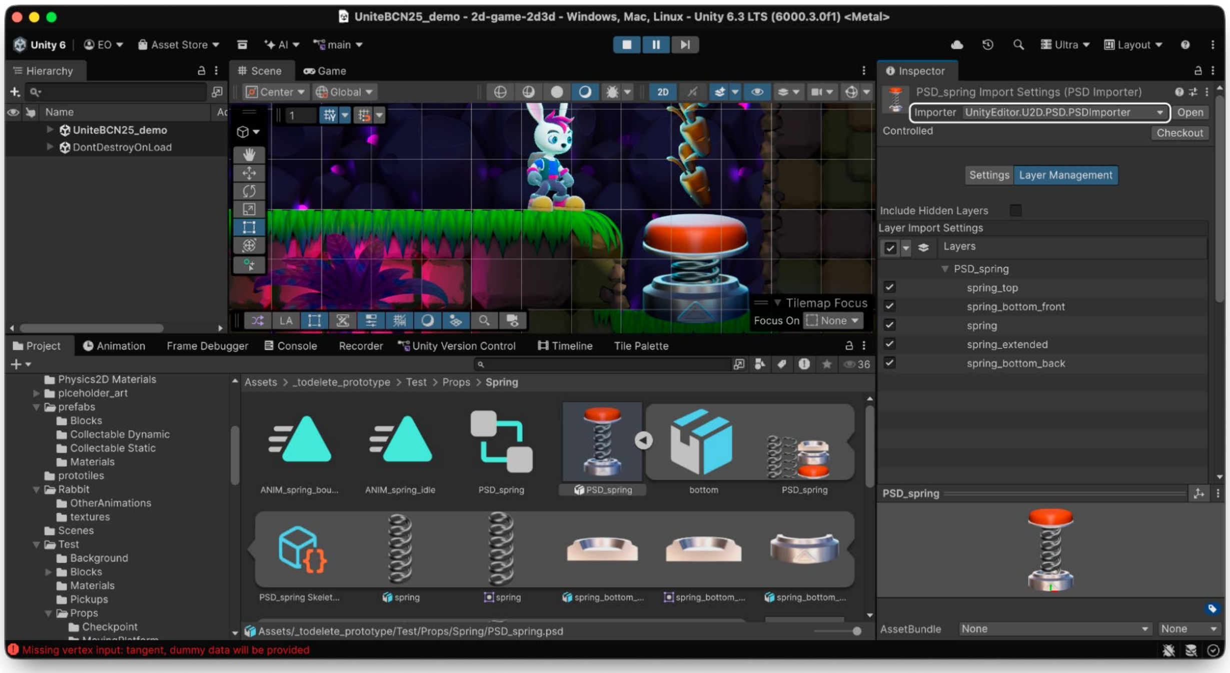
Unity's 2D PSD Importer package gives you the option of importing layers as separate sprites. This package was designed for use with 2D animation, but you can use it to import multiple regular sprites contained in one PSD file.

The PSD layers can have different blend modes, opacities, effects, or be smart objects that will display correctly in Unity.



Creating a prop in Photoshop

The PSD workflow is very convenient to use for your game sprites or UI elements. You can edit, paint, or create new layers, and when you save the file, the changes will be instantly visible in the Editor. With your PSD source file directly in the Unity project, the file can be included in source control of the project like Unity Version Control.


UniteBCN25_demo - 2d-game-2d3d - Windows, Mac, Linux - Unity 6.3 LTS (6000.3.0f1) <Metal>
Hierarchy
UniteBCN25_demo
DontDestroyOnLoad
Scene
Game
Inspector
PSD_spring Import Settings (PSD Importer)
Importer UnityEditor.U2D.PSD.PSDImporter
Settings
Layer Management
Include Hidden Layers
Layer Import Settings
PSD_spring
spring_top
spring_bottom_front
spring
spring_extended
spring_bottom_back
Project
Assets > _todelete_prototype > Test > Props > Spring
Assets/_todelete_prototype/Test/Props/Spring/PSD_spring.psd
Missing vertex input: tangent, dummy data will be provided

The PSD prop imported with its layers ready to use in Unity without intermediate standalone PNG files

Draw order

In a 2D game, all sprites and objects have the same depth. How can you sort them so that some appear in front of others?

Unity sorts Renderers according to a priority order that depends on their types and usages. You can specify the render order of Renderers using their Render Queue. In general, there are two main queues: the Opaque queue and the Transparent queue. 2D Renderers are mainly within the Transparent queue and include the Sprite Renderer, Tilemap Renderer, and Sprite Shape Renderer types.

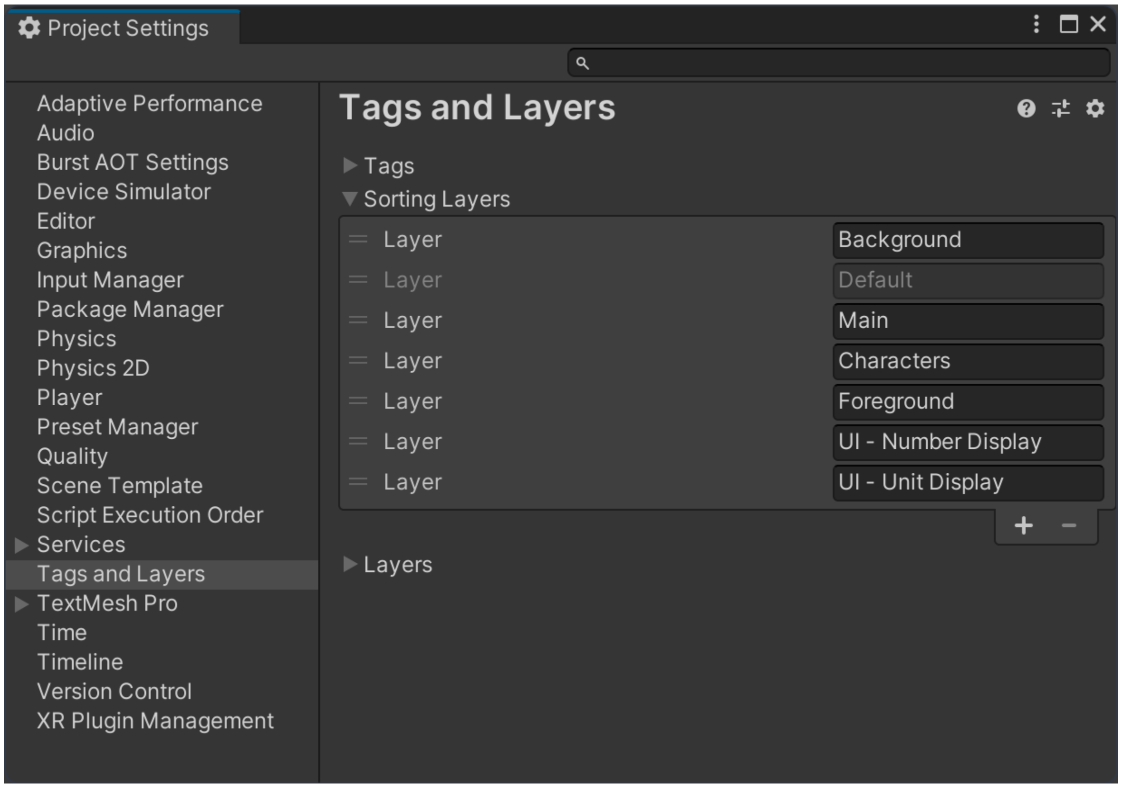
2D Renderers within the Transparent Queue generally follow a priority order. When two or more objects occupy the same space, Unity goes through this list and checks which object should be drawn on top. When there's a tie, and both objects have the same value, the next criteria on the list is evaluated:

1. Distance to camera or custom axis
2. Order in Layer
3. Sorting Layer
4. Sorting Group
5. Specify Render Queue
6. Material / Shader

When all of the above values are the same for both objects and a tiebreaker is needed, this process must choose which one to render on top. It's not an ideal solution, so make sure to set a distinct sorting order using Sorting Layers and Sorting Groups.

Sorting Layers

The most important sorting criteria are Sorting Layers. All 2D Renderers have this option, and it's the first thing you need to set up. There is a default sorting layer which you can edit by opening Project Settings, then specifying Tags and Layers options.



Layer	Background
Layer	Default
Layer	Main
Layer	Characters
Layer	Foreground
Layer	UI - Number Display
Layer	UI - Unit Display

Editing the Sorting Layer

Add, delete, or change the order by dragging the handle on the left of the Layers. Layers that are higher on the list are rendered first and will appear further away from the camera.

Plan the Sorting Layers structure, even when designing a mockup to organize your scene early on. Set a sprite's sorting group as soon as you place it in the scene. These practices will help you to avoid a situation later in your project development when you suddenly realize you have to change sorting settings on potentially thousands of sprites.

Avoid too many Sorting Layers

2D Lights rely on Sorting Layers, so consider lighting when you set the Sorting Layers structure. Know beforehand how you want the lights to behave in your game and what groups will be affected by them.

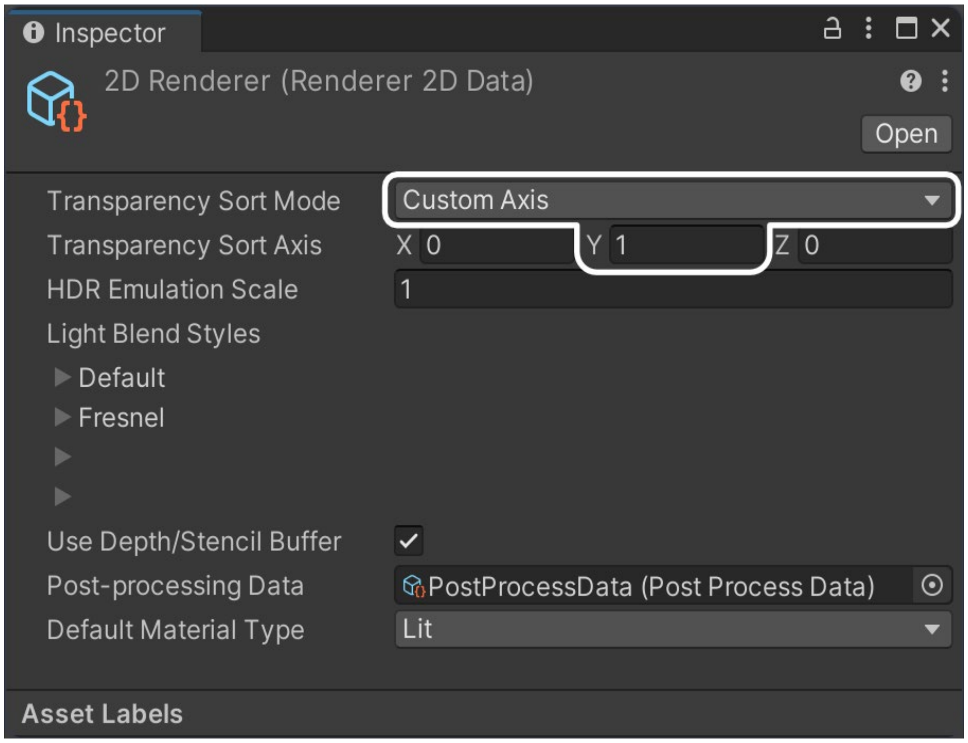
For example, if you're making an isometric game and plan to have torches that use 2D Lights, will the torches affect their entire surroundings or only the walls? Will characters be affected, or will they be drawn in front of the light? Will you need a Sorting Layer just for the objects that will receive light? Keep these kinds of questions in mind when you're editing Sorting Layers, and learn more about planning lighting in the 2D Lights section.

Be wary of using too many Sorting Layers, which can cause you to lose oversight of all the details. When you need to sort renderers further, use Order in Layer. This allows you to sort objects that are on the same layer easily.

Transparency Sort Mode

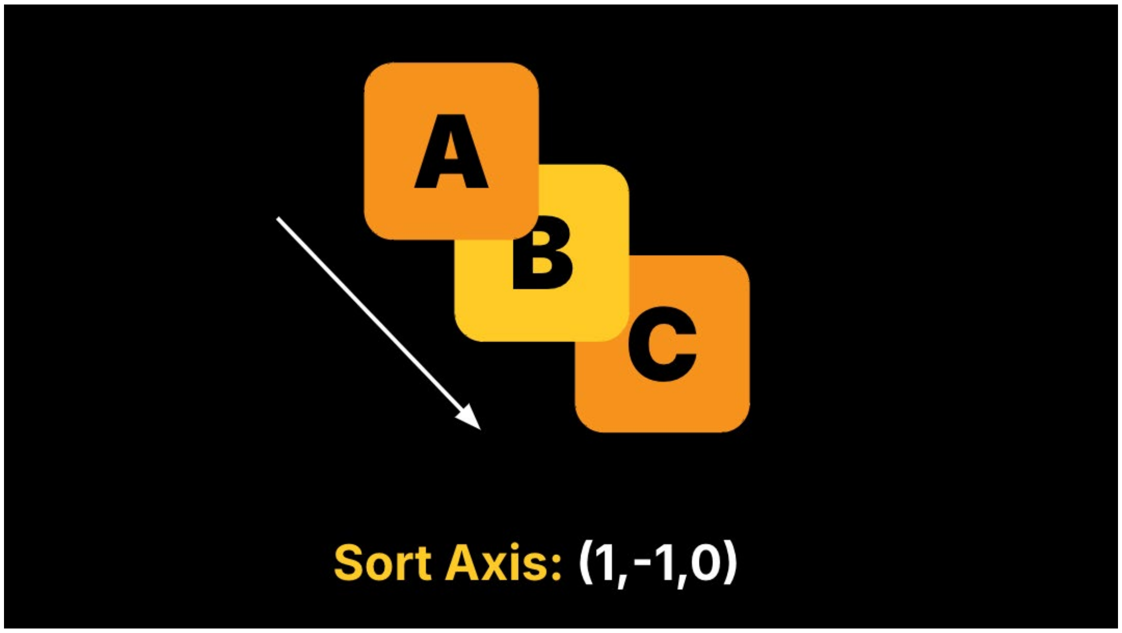
If you use an orthographic camera you might want a fake 3D view in your game, such as an axonometric or oblique projection for an isometric top-down game with a bit of an angle, or a cabinet projection in brawler games. If so, you'll need to custom sort the renderers by Distance to Camera.

Choose a custom axis to sort objects. In the case of a top-down game or brawler, sort objects on the Y axis, so higher characters will be rendered below other characters, giving the illusion that they are further away. To edit this option, find the 2D Renderer Asset created when setting up the URP, and change its Transparency Sort Mode option to Custom Axis, then set its Transparency Sort Axis value to 0, 1, 0.



Transparency Sort Mode: Custom Axis
Transparency Sort Axis: X 0 Y 1 Z 0
HDR Emulation Scale: 1
Use Depth/Stencil Buffer: ✓
Post-processing Data: PostProcessData (Post Process Data)
Default Material Type: Lit

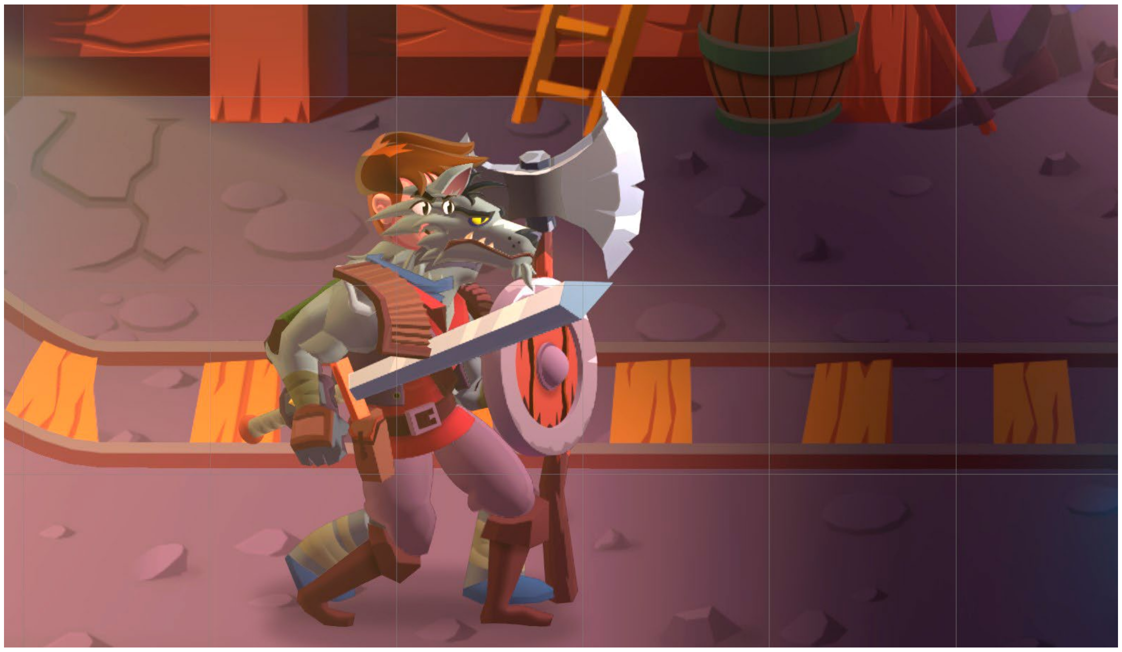
Sorting sprites on the Y axis, the value (X:0, Y:1) 1 tells Unity to follow a downward vector (as in Vector2.down) for the sorting criteria. The sprites at the top will be drawn first, and the ones at the bottom later, making them visible at the front.



You can represent any vector with sort axis that will be the direction in which sprites will be rendered from top to bottom.

Sorting Groups

Even if you follow the previous steps and sort on the Y axis, the appearance of your character's parts or child objects might still display incorrectly.

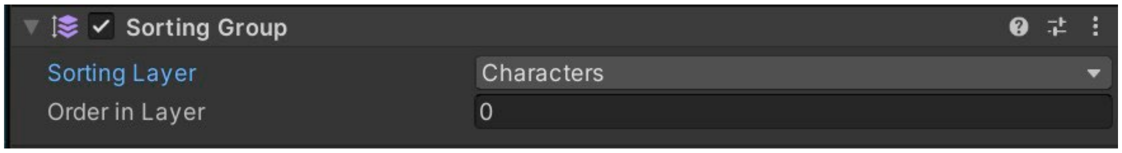


Parts of the characters that are on the same Sorting Layer are mixed between two characters.

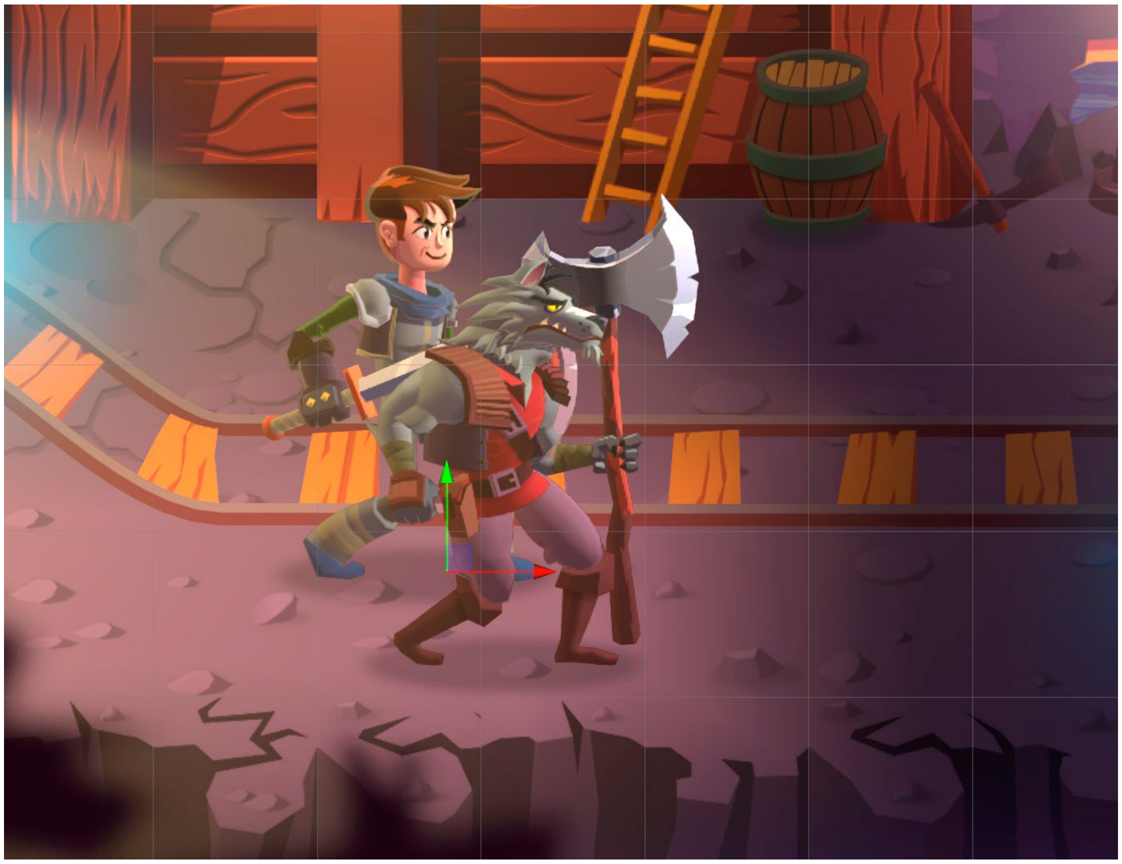
It's not a bug. This is just how sorting works. Sorting doesn't know which parts belong to which character if they are on the same Sorting Layer. Fortunately, Sorting Group is the fourth criteria on the sorting priority list.

Sorting Group is a component that groups Renderers that share a common root for sorting purposes. All Renderers within the same Sorting Group share the same Sorting Layer, Order in Layer, and Distance to Camera.

When using characters or other objects that consist of a couple of sprites, put this component on its Parent object (highest in the hierarchy), and set Sorting Layer and Order in Layer just like on any other 2D Renderer. You can see an example of this in this Unite Now technical session.


Sorting Group
Sorting Layer: Characters
Order in Layer: 0

Creating a Sorting Group in the parent GameObject to ensure that this object and its child objects get sorted as a single element, avoiding potential conflicts between its parts and other sprites in the game


The Sorting Group makes the characters appear as they should.

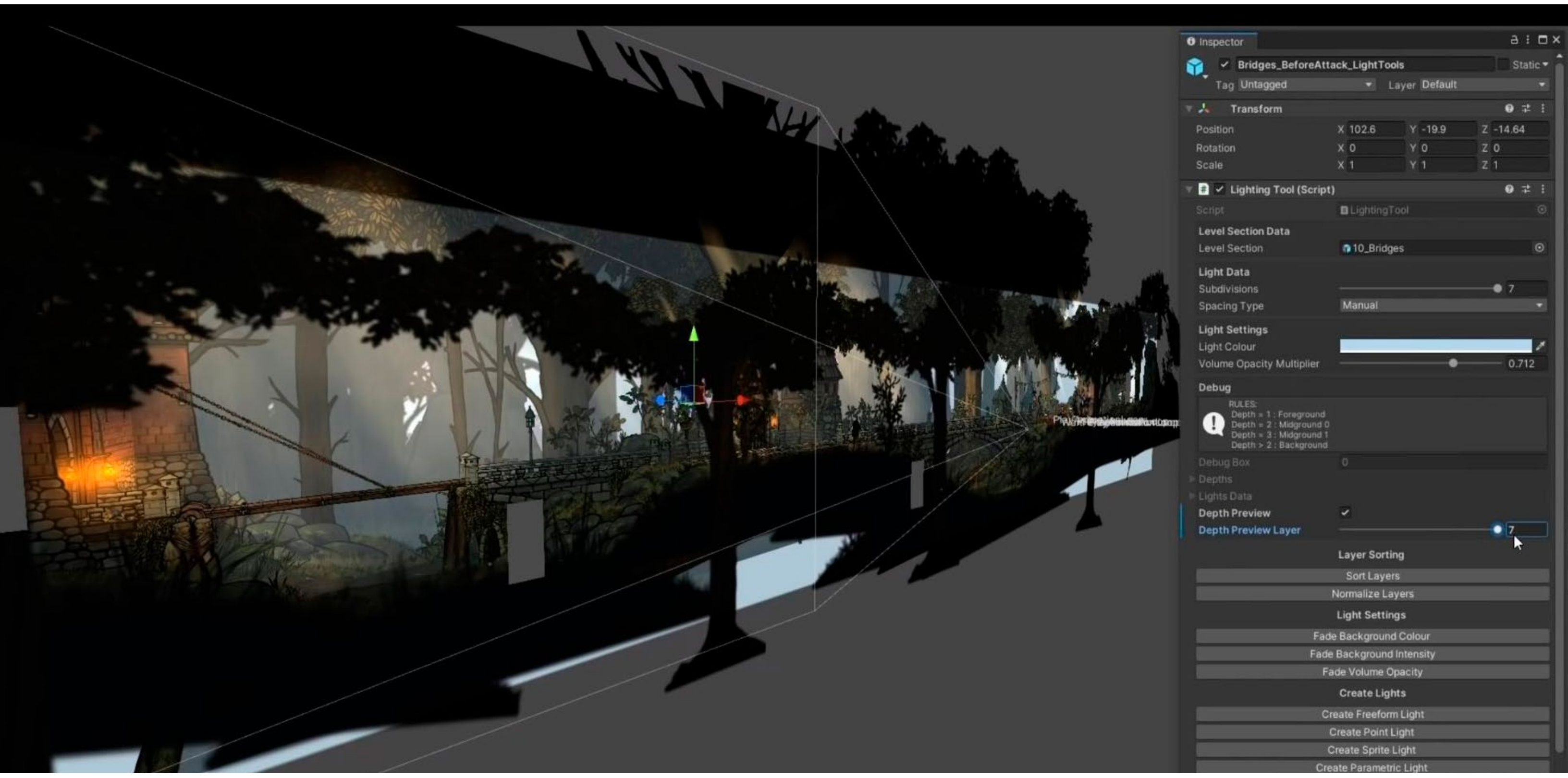
You can also nest Sorting Groups. If you have weapons consisting of multiple sprites that you want to be generated randomly, put them in Sorting Groups to make them display correctly in characters' hands.

Cameras and perspective in 2D

Orthographic or perspective camera

For most 2D games, you'll want to set your camera to Orthographic mode. It's the default choice for 2D projects because lines of one dimension are parallel, and you work only in two dimensions. The orthographic mode works for all styles – pixel art games, puzzles, 2D isometric, top-down, platformers.

Although 2D games have no real depth, the illusion of depth can be created using the parallax effect. The parallax effect scrolls multiple layers in the background and foreground at different speeds when the camera moves. The intent is to mimic human depth perception, wherein objects farther away appear to move more slowly than those closer to us.



```
Bridges_BeforeAttack_LightTools
Tag Untagged          Layer Default
Position   X 102.6   Y -19.9   Z -14.64
Rotation   X 0       Y 0       Z 0
Scale      X 1       Y 1       Z 1
Script     LightingTool
Level Section Data
Level Section   10_Bridges
Light Data
Subdivisions    7
Spacing Type    Manual
Light Settings
Light Colour
Volume Opacity Multiplier   0.712
Debug
RULES:
 Depth = 1 : Foreground
 Depth = 2 : Midground 0
 Depth = 3 : Midground 1
 Depth > 2 : Background
Debug Box   0
Depths
Lights Data
Depth Preview   ✓
Depth Preview Layer   7
```

Perspective camera setup in Tails of Iron By Oddbug Studio; learn how they created the lighting in their game with Unity 2D Lights

If orthographic mode is suitable for all 2D game styles, why even use perspective mode?

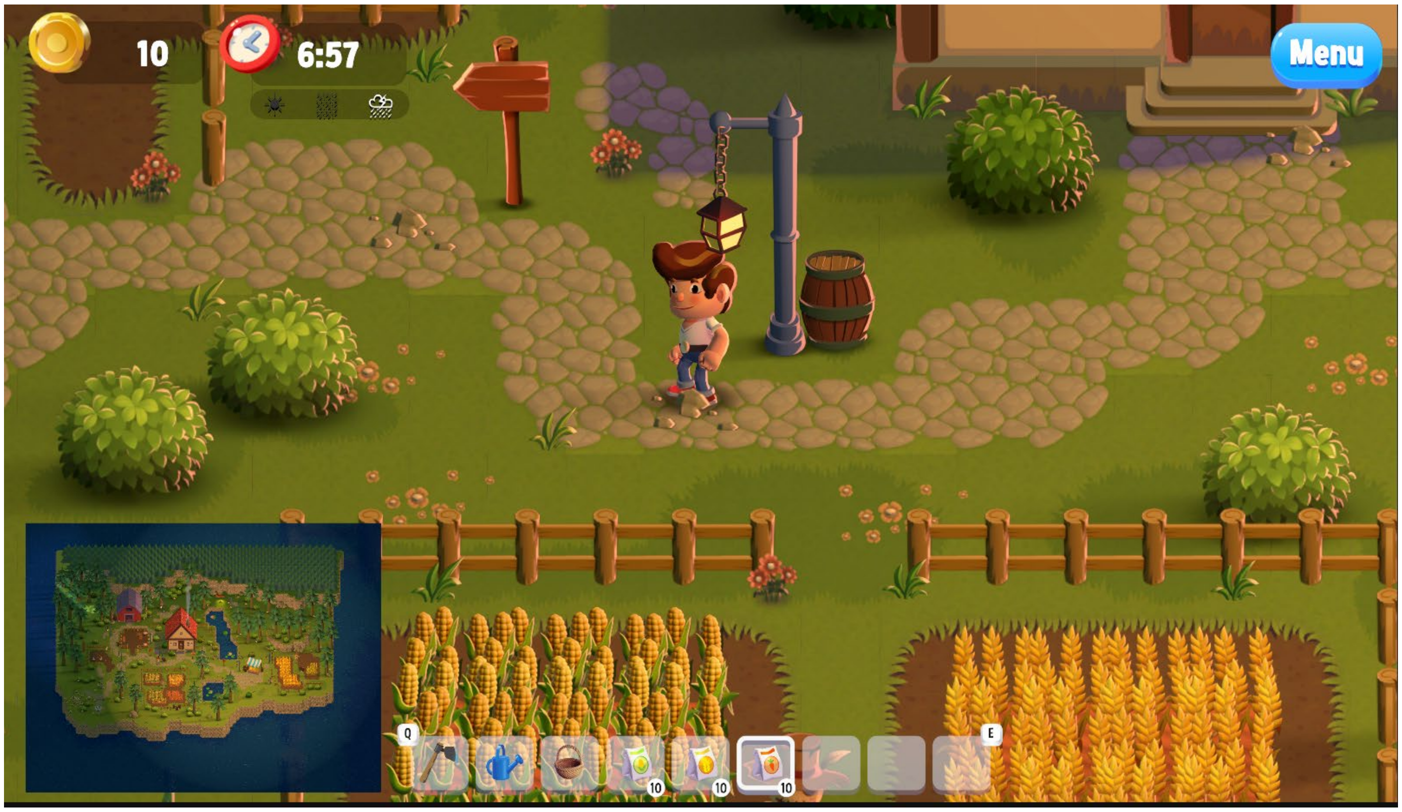
One reason is that this offers access to parallax effects out of the box; as the name suggests, a perspective camera uses a perspective viewpoint. You don't need to use scripts to handle the scrolling of parallax layers. Instead, you position elements further from the camera on the Z axis and scale them up to account for perspective based on their relative distance from the camera.

The camera you choose doesn't impact the art creation process, but it does determine how the levels are set up and the look and feel of the game.

To sum up, orthographic mode is likely the most common choice, except when you're using advanced scrolling and parallax. In that case, you probably want to go with perspective mode.

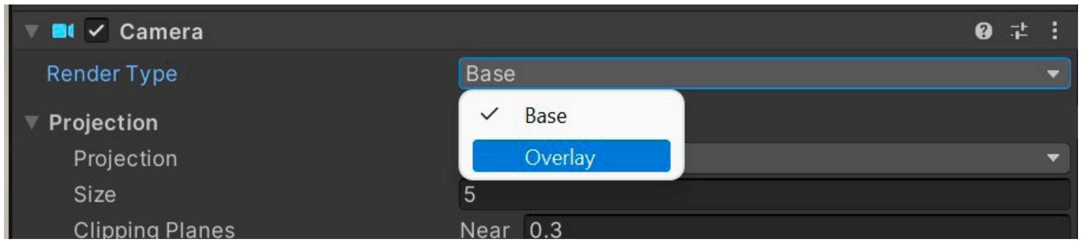
Camera stacking

Camera stacking in URP allows you to layer camera outputs and combine them in one final image. You can combine 2D, 3D, and UI objects with camera stacking.



A use case for camera stacking can be to display other areas of the map on top of the main camera, for example, to enable a second player to explore other parts of the map, like a split screen, or in this case, to provide an aerial view of the scene.

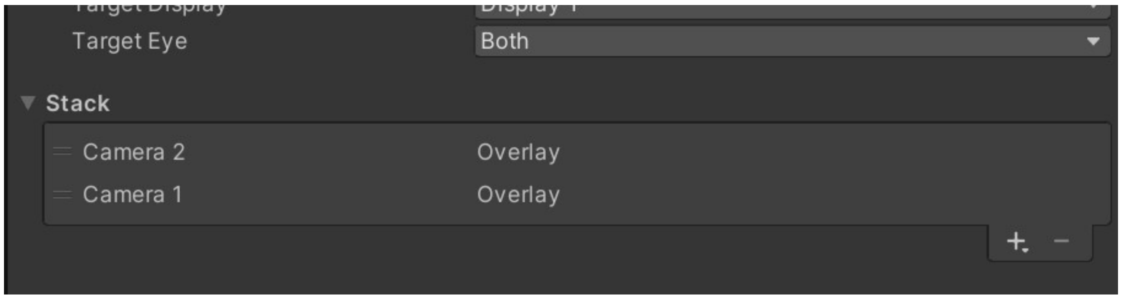
At least two cameras are required in the scene for stacking. Choose which one will be the base camera for the image you want to be rendered first, and any additional cameras need to be set to Overlay.



Camera
Render Type: Base
Base
Overlay
Projection
Projection
Size: 5
Clipping Planes: Near 0.3

Setting up camera stacking

The base camera needs to know which cameras overlay on top and in what order. To do that, go to the Stack option, and select a camera to overlay with the + button. Change the rendering order of Overlay Cameras as needed by dragging them up or down in the list.



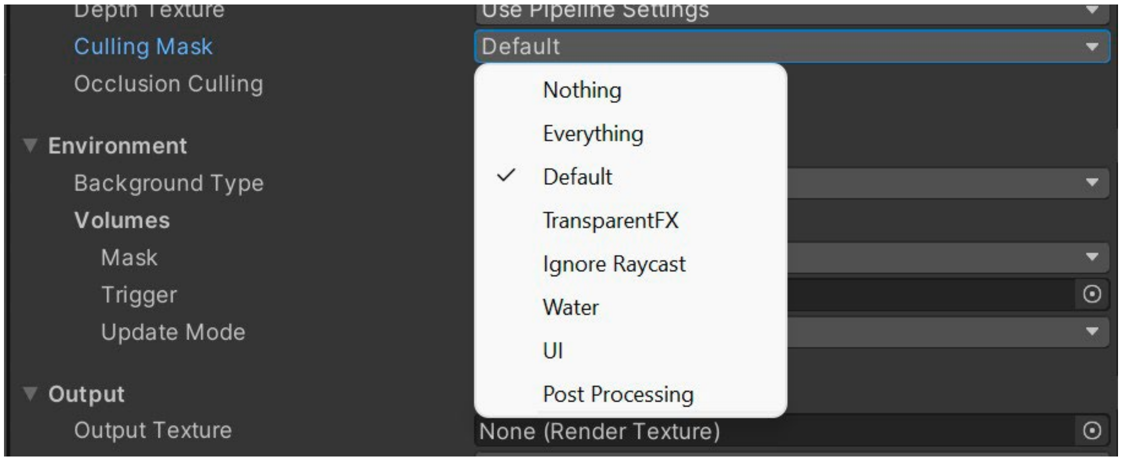
Target Eye: Both

Stack

Camera 2	Overlay
Camera 1	Overlay

Selecting cameras to be rendered as overlays

Use layers to set which objects are rendered by each of the cameras. By default, cameras render every layer, but you can change that by selecting which layers are rendered by the camera in the Culling Mask option on a Camera component.



Culling Mask: Default

Choosing layers to render in the camera's Culling Mask option

Now, select the GameObjects you want the camera to render, and change their layer to match the one set earlier. That way, objects are categorized to be rendered by different cameras, such as a 3D background and 2D platform layers.

Cinemachine for 2D

Once you've selected the camera type, you'll need to set it up to follow the gameplay. Unity's Cinemachine system provides the functionality to do this and much more, such as confining the camera to level bounds, setting up camera transitions, noise, and so on. This section highlights some of the core Cinemachine functionality for 2D games.

To get started, install Cinemachine from the Package Manager. Then complete the following steps to add a Cinemachine camera to the Scene view in your project:

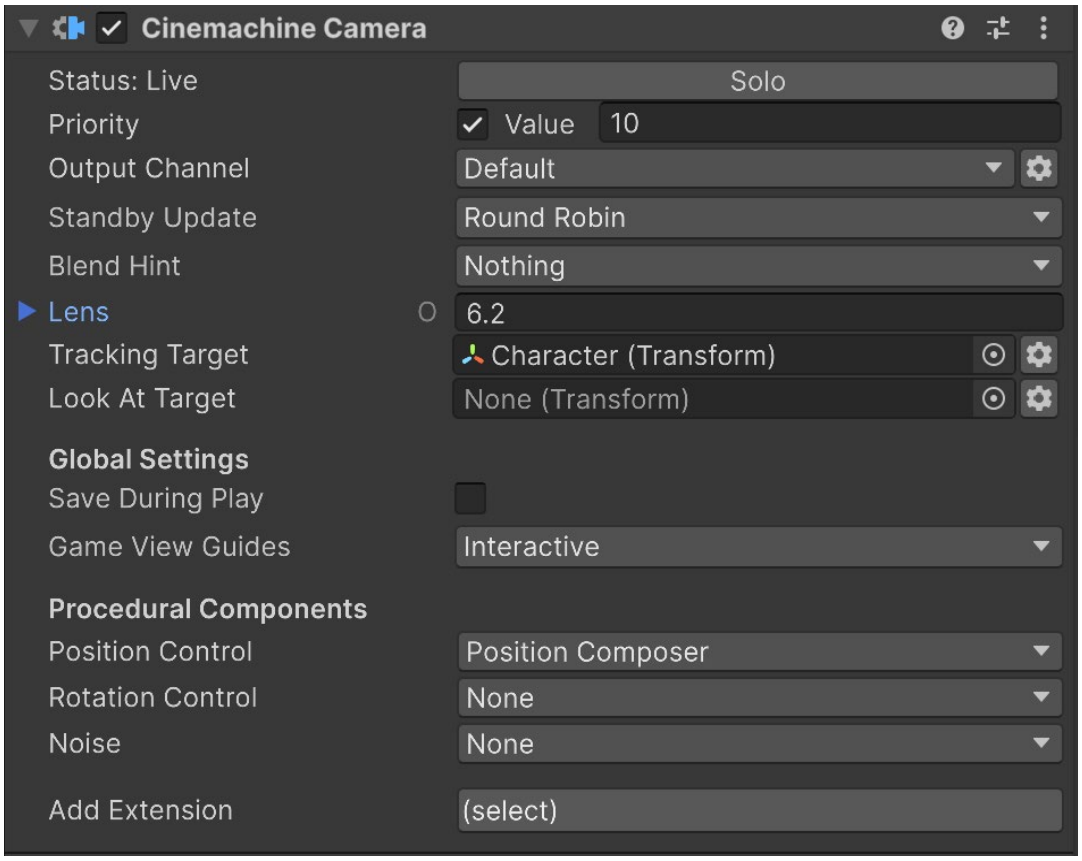
1. In the Scene view, navigate the Scene to get the point of view you want to frame with the Cinemachine Camera.
2. In the Unity menu, select **GameObject** > **Cinemachine** > **Cinemachine Camera**.
   Unity adds a new GameObject with:
   - A Cinemachine Camera component and
   - A Transform that matches the latest position and orientation of the Scene view camera.

When you create the first Cinemachine Camera in a scene, Unity automatically adds a Cinemachine Brain component to the Unity Camera, unless the Unity Camera already includes one. To verify it:

1. In the Hierarchy, select your Unity Camera (this is the GameObject that includes the Camera component).
2. In the Inspector, verify that the GameObject includes a Cinemachine Brain component.

The Cinemachine Brain monitors all active virtual cameras in the scene. Animate the virtual cameras with keyframes, blend or smoothly transition between the cameras, or create a combination of the two to animate one camera while transitioning to another. All of the animation will be processed by the Cinemachine Brain and applied to the main camera – think of it as a powerful animation system that drives your main camera.

To create a 2D virtual camera, click on **Cinemachine > Create 2D Camera**. This will create a virtual camera set up for a 2D environment. If it's your first Virtual Camera in the scene, it will also add a Cinemachine Brain Component to your main camera.



Cinemachine Camera

Status: Live	Solo
Priority	✓ Value 10
Output Channel	Default
Standby Update	Round Robin
Blend Hint	Nothing
Lens	6.2
Tracking Target	Character (Transform)
Look At Target	None (Transform)
Global Settings	
Save During Play	☐
Game View Guides	Interactive
Procedural Components	
Position Control	Position Composer
Rotation Control	None
Noise	None
Add Extension	(select)

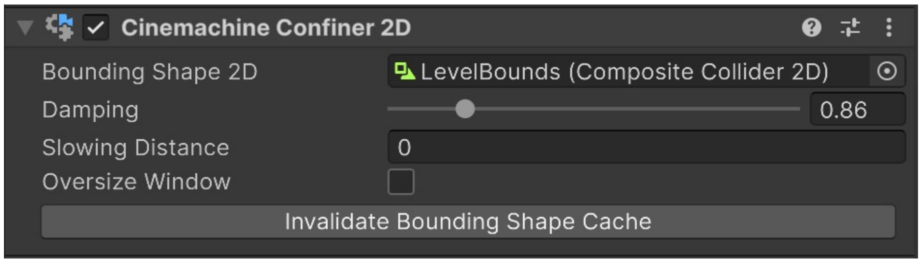
Creating a Cinemachine Virtual Camera

Cinemachine needs an object to follow, so assign a player character in the Tracking Target field. To ensure that you are operating on a 2D plane and that the camera isn't panning or tilting, check that the Look At Target field is empty and Position Control is set to Follow or Position Composer. Finally, set the Lens properties for your projects. Keep in mind that some options will be inherited from the main camera's properties.

In the Position Composer component find useful options to modify how the Virtual Camera follows its target, such as Offset, Damping, Dead, and Soft Zones. Experiment with these options during Play mode, and the changes will be saved if you check the Save During Play option.

There is also an extension for the Virtual Camera, the Cinemachine Confiner 2D, which, when activated, limits the camera from moving outside Level bounds, ensuring the player will see only the parts that you want them to see. It also ensures you don't need to design unnecessary parts of the level.

To add Cinemachine Confiner 2D, select it from the Add Extension dropdown menu.



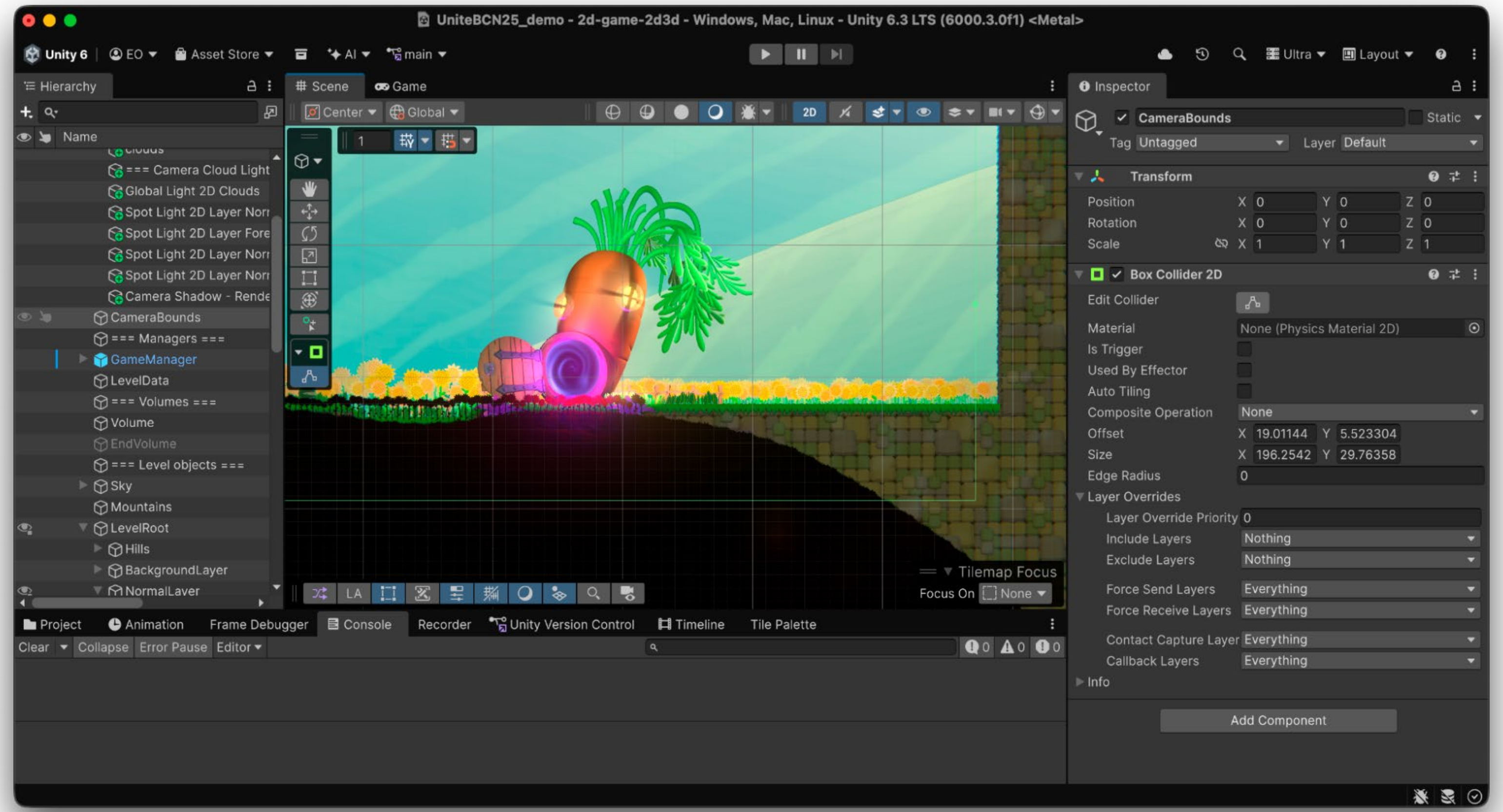
Cinemachine Confiner 2D

Bounding Shape 2D	LevelBounds (Composite Collider 2D)
Damping	0.86
Slowing Distance	0
Oversize Window	☐

Invalidate Bounding Shape Cache

Adding CinemachineConfiner2D extension

Cinemachine Confiner 2D requires a Collider 2D (Composite or Polygon) as a Bounding Shape 2D. Create an empty GameObject, and add a Composite Collider 2D and Box Collider 2D. A RigidBody 2D will be added automatically; set its Body Type to Static. Also, check the Used by Composite option on the Box Collider 2D. Now, drag this GameObject into the Bounding Shape 2D field of the Cinemachine Confiner 2D script. Don't forget to edit the size of the Box Collider 2D so that it's larger than the size of the camera.



CameraBounds — Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Box Collider 2D

Material	None (Physics Material 2D)
Is Trigger	☐
Used By Effector	☐
Auto Tiling	☐
Composite Operation	None
Offset	X 19.01144 Y 5.523304
Size	X 196.2542 Y 29.76358
Edge Radius	0
Layer Override Priority	0
Include Layers	Nothing
Exclude Layers	Nothing
Force Send Layers	Everything
Force Receive Layers	Everything
Contact Capture Layer	Everything
Callback Layers	Everything

The Collider 2D called CameraBounds prevents the camera from showing beyond the boundaries.

Now the camera's frustum won't go out the bounding box of the collider. For more details, check the Cinemachine Confiner 2D documentation.

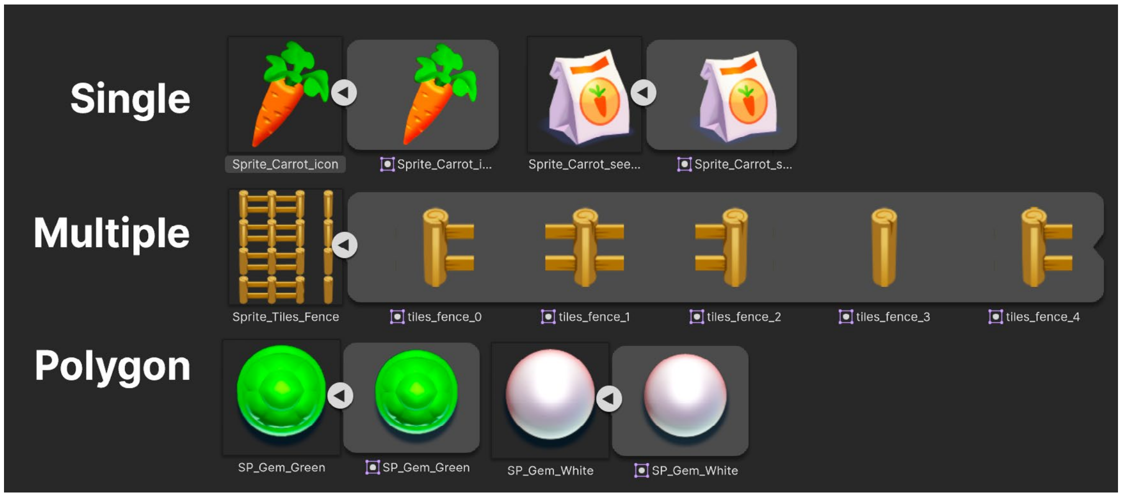
Working with sprites

Sprites are the key elements in any 2D game. In a project using the 2D template, your visual assets will be imported as sprites by default. They contain your art and are ready to be added to your scene and used together with Unity's full set of 2D tools.

Sprite assets

Sprites are 2D textures mapped onto flat, 3D meshes. Sprite modes that are available in the sprite asset include:

- **Single:** This is the default mode, where the image only contains a single image element.
- **Multiple:** Choose this value if the texture source file has several elements in the same image. Then define the location of the elements in the Sprite Editor so that Unity knows how to split the image into different sub-assets. Once sliced, each graphic becomes an individual sprite that can be used separately in the UI Toolkit. PSD assets imported with PSD Importer will by default show a sprite per layer in the Photoshop file.
  - Images that need to be sliced, like tileable images or 9-slice sprites, will need to be of the type Multiple as well.
- **Polygon**: Best for images that are circular or a regular polygon, this mode helps you to set up an outline that closely matches the image shape, resulting in a cleaner and more optimal outline, this can be useful, for example, for particle effects and reduce overdraw.



Examples of assets using different sprite modes

Note that sprites meant to be used as **secondary textures** (see the section on lighting that covers working with secondary textures for more information) should be imported as **Normal map** (normal maps) and **Default** or **Single Channel** (mask maps).

Additional sprite-specific settings can be found in the Inspector:

— **Pixels Per Unit:** Indicates the expected pixel density per grid unit (as described in the earlier section on setting the resolution of your assets)

— **Mesh Type:** Indicates if the sprite will be a quad 3D mesh with **Full Rect,** or a tight 3D mesh that will wrap around the opaque pixels of the texture to optimize rendering with **Tight**; the mesh is referred to as Outline in the Sprite Editor, from where you can adjust it

— **Extrude Edges**: An option that adds some extra margin between the texture's opaque pixels and the mesh

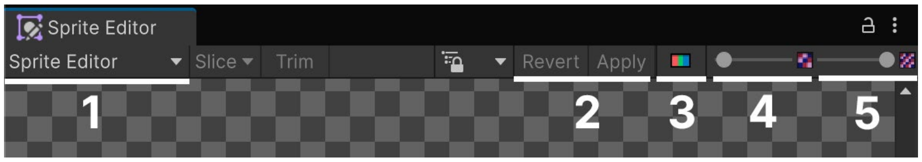
— **Pivot**: Indicates reference point in the mesh from which to scale, move, and rotate the sprite; values are normalized: 0 is bottom or left, 1 is top or right, and 0.5 the middle; you can also adjust it from the Sprite Editor window

— **Generate Physics Shape**: Uses the outline of the sprite for physics if no other shape has been defined specifically in the Sprite Editor

Sprite Editor

Sprites have many properties shared throughout Unity's 2D toolset. Simple sprites will be ready to use as soon as you import them into Unity. Other sprites, however, will require further refinement based on their use case. The Sprite Editor is accessible from the Inspector window when you select a sprite asset or from the top menu via **Windows > 2D > Sprite Editor**.

Here is a quick introduction to the menu options in the Sprite Editor:



1. Click on the Sprite Editor menu to access a drop-down list providing additional options for sprite editing.
2. Choose one of these options if you need to apply or revert changes to the sprite when you make changes.
3. Switch between color and alpha previsualization mode.
4. Zoom in and out (you can also use your mouse wheel for zoom).
5. Enable this option for the sprite to get a preview of the different mipmap levels.

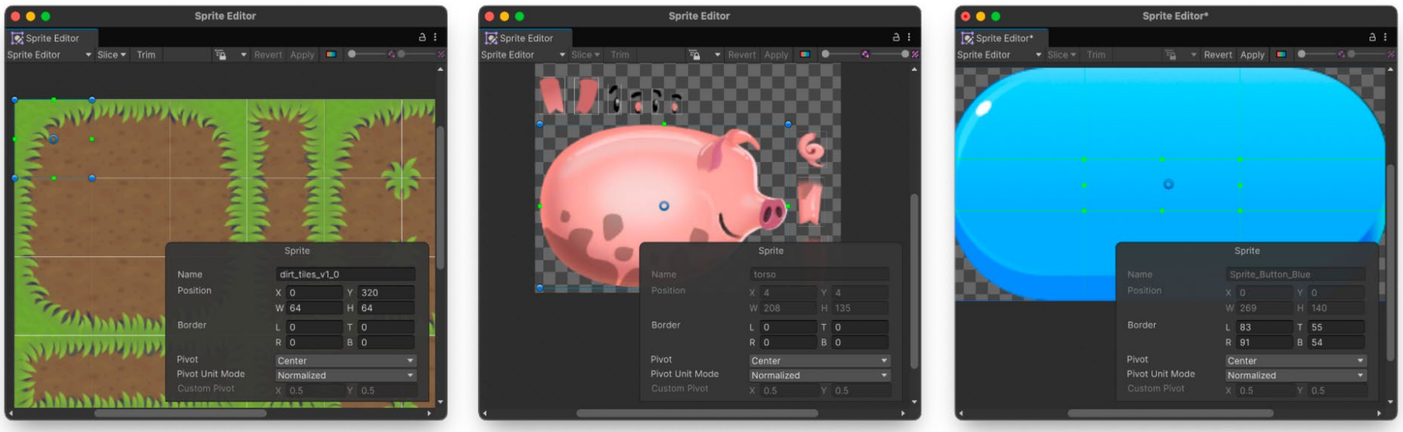
Let's look in more detail at each of these options.

Sprite Editor

In this menu, you'll find tools to slice, adjust borders, and pivot each sprite. The slice options will be available if **Multiple** mode is chosen in the Sprite Asset, for example, if you import a tile sheet.

The parts of a multi-part character can be sliced manually if they are all in the same sheet, but the recommended workflow with PSD Importer is to generate a sprite for each layer in your Photoshop file, so no slicing will be needed.

Lastly, for sprites that are resized but need to keep their borders intact, like a platform sprite or a UI button, you can define those borders and the inner repeatable area with the green handles that are available with the 9-slice feature. And you can adjust the pivot for every sprite by moving the pivot handle.



Sprite
Name dirt_tiles_v1_0
Position X 0 Y 320
W 64 H 64
Border L 0 T 0
R 0 B 0
Pivot Center
Pivot Unit Mode Normalized
Custom Pivot X 0.5 Y 0.5

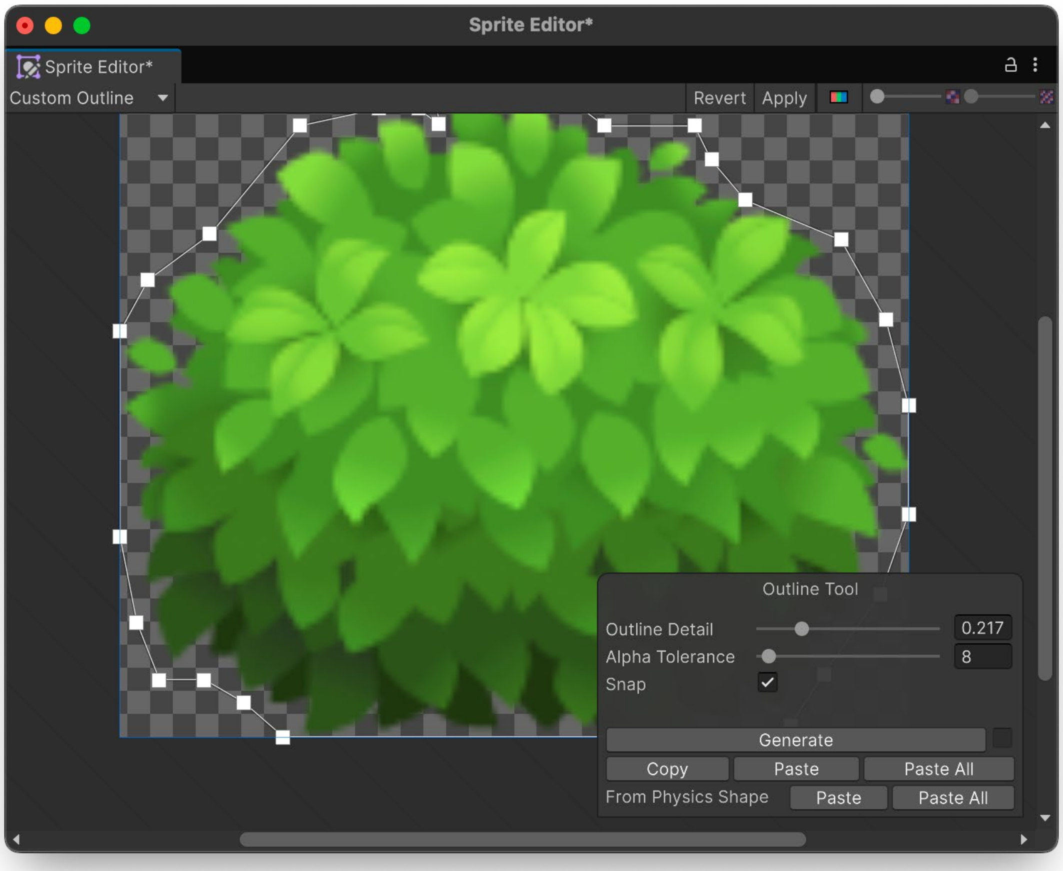
Sprite
Name torso
Position X 4 Y 4
W 208 H 135
Border L 0 T 0
R 0 B 0
Pivot Center
Pivot Unit Mode Normalized
Custom Pivot X 0.5 Y 0.5

Sprite
Name Sprite_Button_Blue
Position X 0 Y 0
W 269 H 140
Border L 83 T 55
R 91 B 54
Pivot Center
Pivot Unit Mode Normalized
Custom Pivot X 0.5 Y 0.5

Left to right: A tilesheet being sliced, a multi-part character for animation, and a 9-slice sprite

Custom Outline

Use the **Custom Outline** option to create a tight mesh that's adjusted to the opaque pixels of the sprite, thereby improving performance (the pixels outside the outline are ignored instead of being rendered). You can fine tune the mesh or outline generation by moving the path points manually to better conform to the shape without creating too complex of an outline shape.

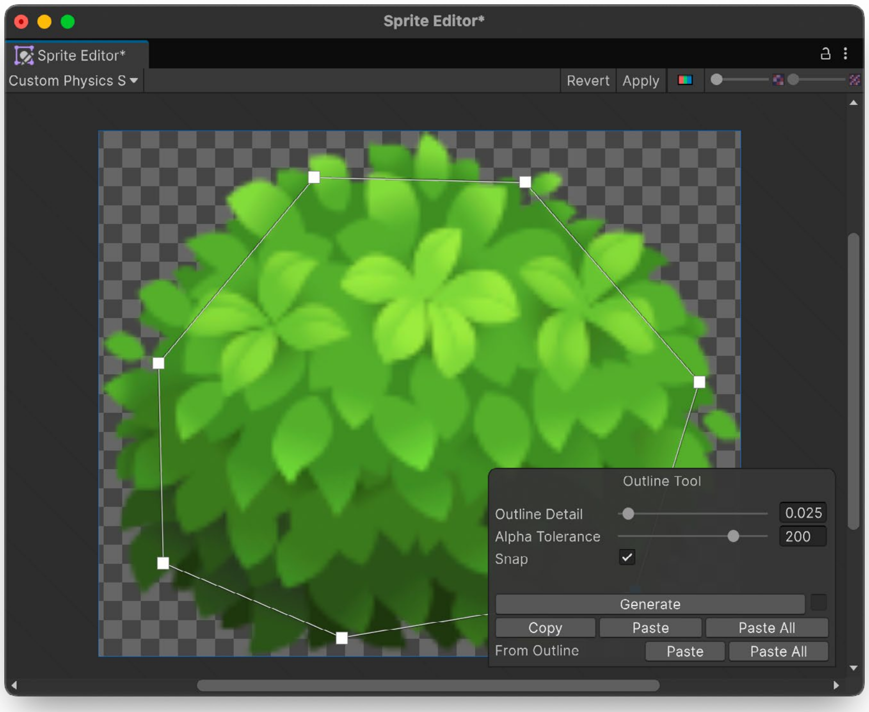


Outline Detail 0.217
Alpha Tolerance 8
Snap ✓

An outline generated and adjusted with the Custom Outline option

Custom Physics Shape

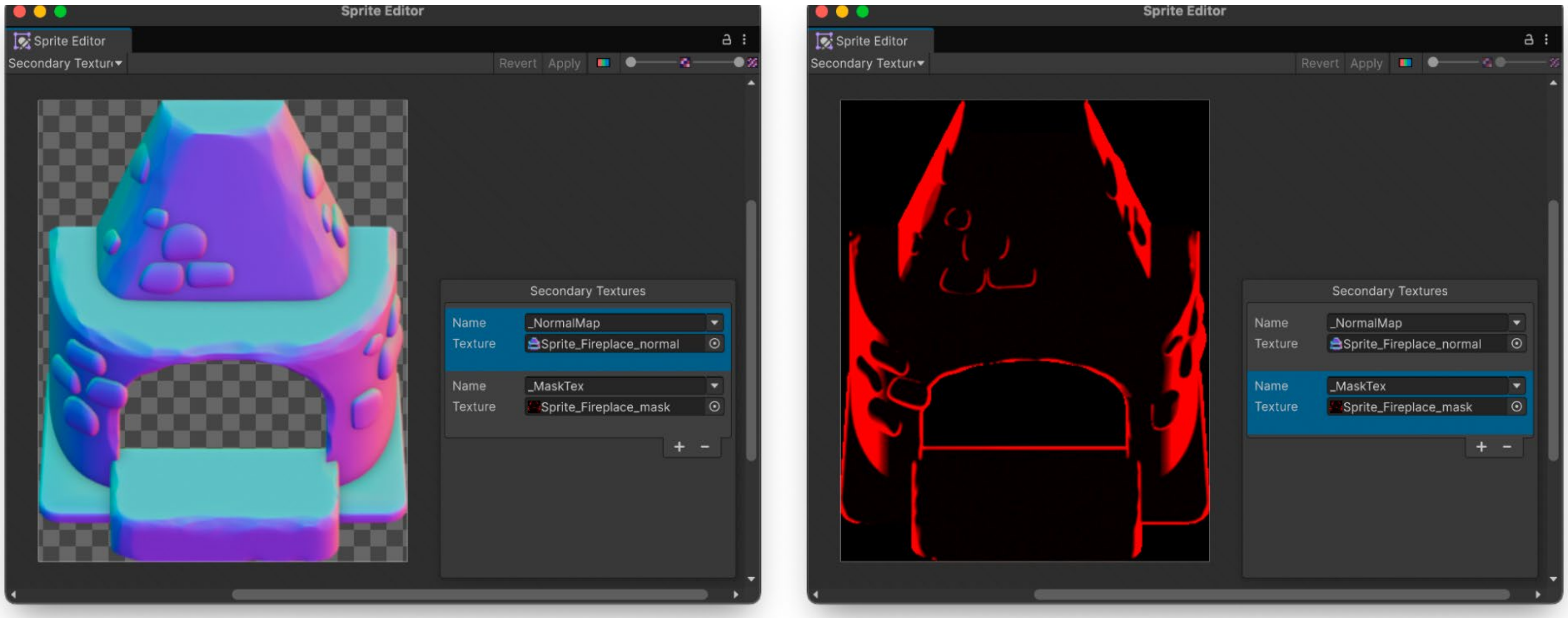
Although it shares the same generation options as the Custom Outline feature, the use case for the Custom Physics Shape module is different. With this feature, you can define an area of a sprite that will collide with another physics object that does not follow closely to the sprite's visual silhouette. Keep in mind that complex physics shapes have a performance cost so keep them as simple as possible.



A custom physics shape created in UI Builder

Secondary textures

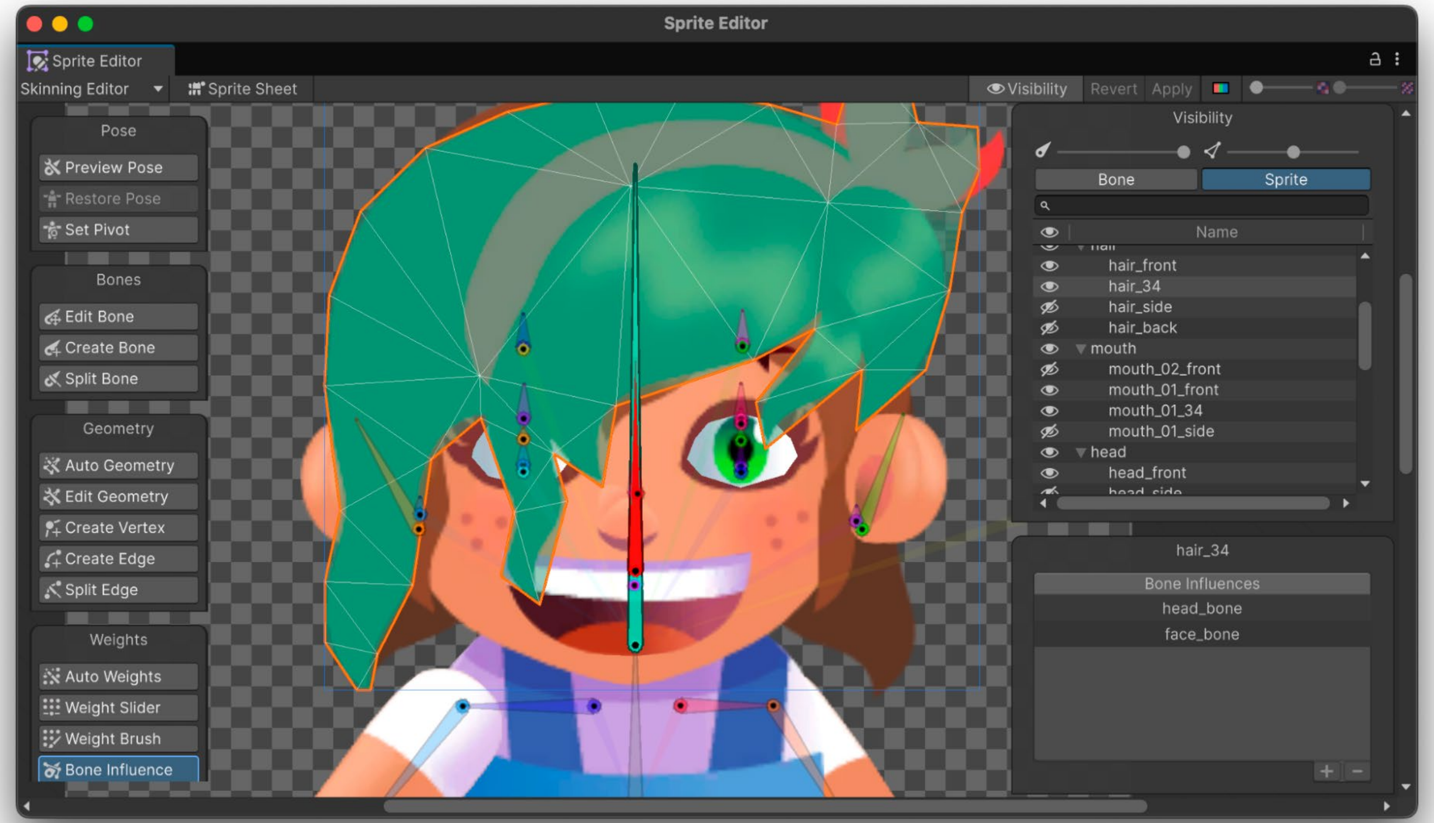
With this module you can choose which normal map or mask map asset will be used by the 2D light system or Shader Graph. While it's not mandatory to add secondary textures, they can provide a visual boost to the look of your game when combined with 2D lights. Read more on secondary textures in the 2D lights and shadows chapter below.



A normal map and a mask map added to the sprite to create interesting lighting effects

Skinning Editor

Unity comes with a complete solution for skeletal animation called 2D Animation. In this section you can create geometry for each sprite, associate bones to them and adjust the bone influence over each vertex of the sprite geometry (weights). Once set up, you can use Unity's animation system to create animations where sprites bend, similar to how they do in skinned meshes in 3D characters. You can read more on it in the 2D animation chapter.



A character with multiple parts, rigged in the Sprite Editor, and ready for animation

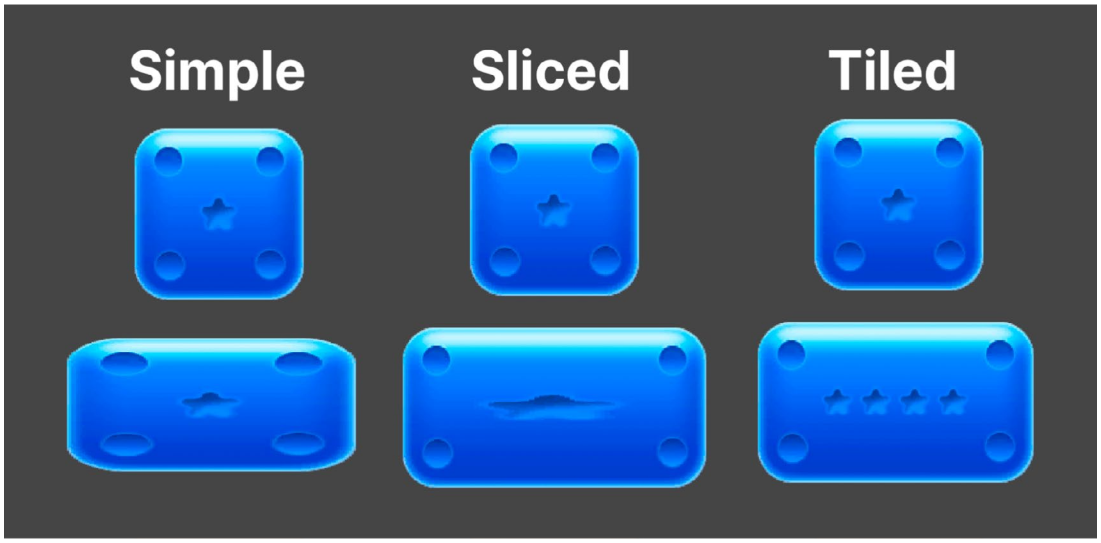
Sprite Renderer component

Once you have a sprite asset ready to use you can start using it by simply dragging the asset into the scene or hierarchy view. A new GameObject will be created with the usual Transform component, and the Sprite Renderer. The following options are available under the Sprite Renderer:

— **Color:** Tint the sprite with a color to create quick variations of the GameObject or add effects during gameplay; you can use the API to adjust it from code.

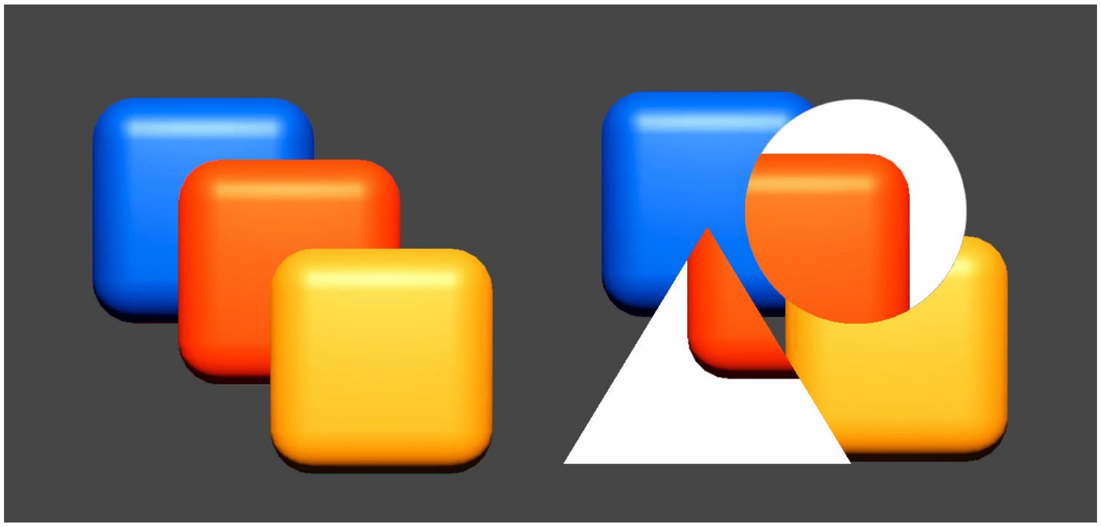
— **Flip:** Flip the direction of the sprite to quickly point sprites to other directions.

— **Draw Mode**: Define how the sprite scales when its dimensions change. Choose from options like Sliced, Tiled, and Simple.



The same sprite scaled with different draw modes

— **Mask Interaction:** Sprites can be occluded or occluders. If you want the sprite to only be visible inside or outside other sprites acting as masks, use the Mask Interaction option. If you want the sprite to behave as the Sprite Mask it needs to have a Sprite Mask component added. See the next section for more on using these options.



Three sprites before and after using mask interactions; the triangle and circle sprites act as the Sprite Masks, the orange cube is only visible inside the mask, and the yellow cube only outside the mask

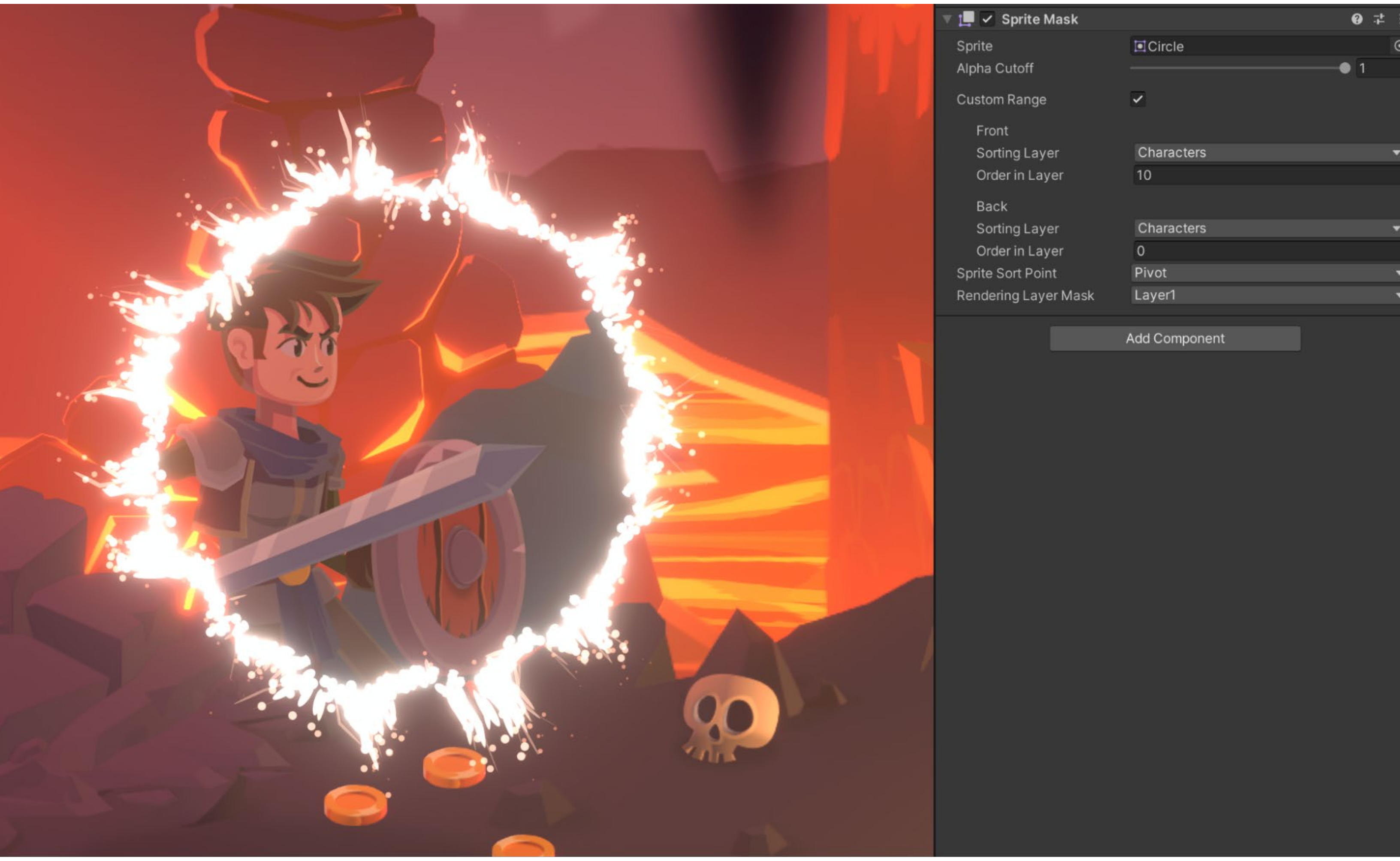
— **Sprite Sort Point**: Choose between the sprite's center or its pivot point when calculating the distance between the sprite and the camera.

— **Material**: Select the material for newly created sprites. The default material is **sprites - Default**, which is compatible with 2D tools like the sorting system and light system. You can click the circle icon to open the object picker window and select other materials, like **sprite - Unlit-Material** if you won't be using 2D lights. Other shaders compatible with Sprite Renderer can be created with Shader Graph.

Sprite masks

With sprite masks you can create interesting effects with a sprite mask. For example, you could hide part of an image to produce a portal effect or make a collectible card with a 3D effect.

The sprite used as the mask image can itself be animated with the animation tool. This enables you to create some interesting effects. For example, you can make a portal appear, then grow bigger or change shape.

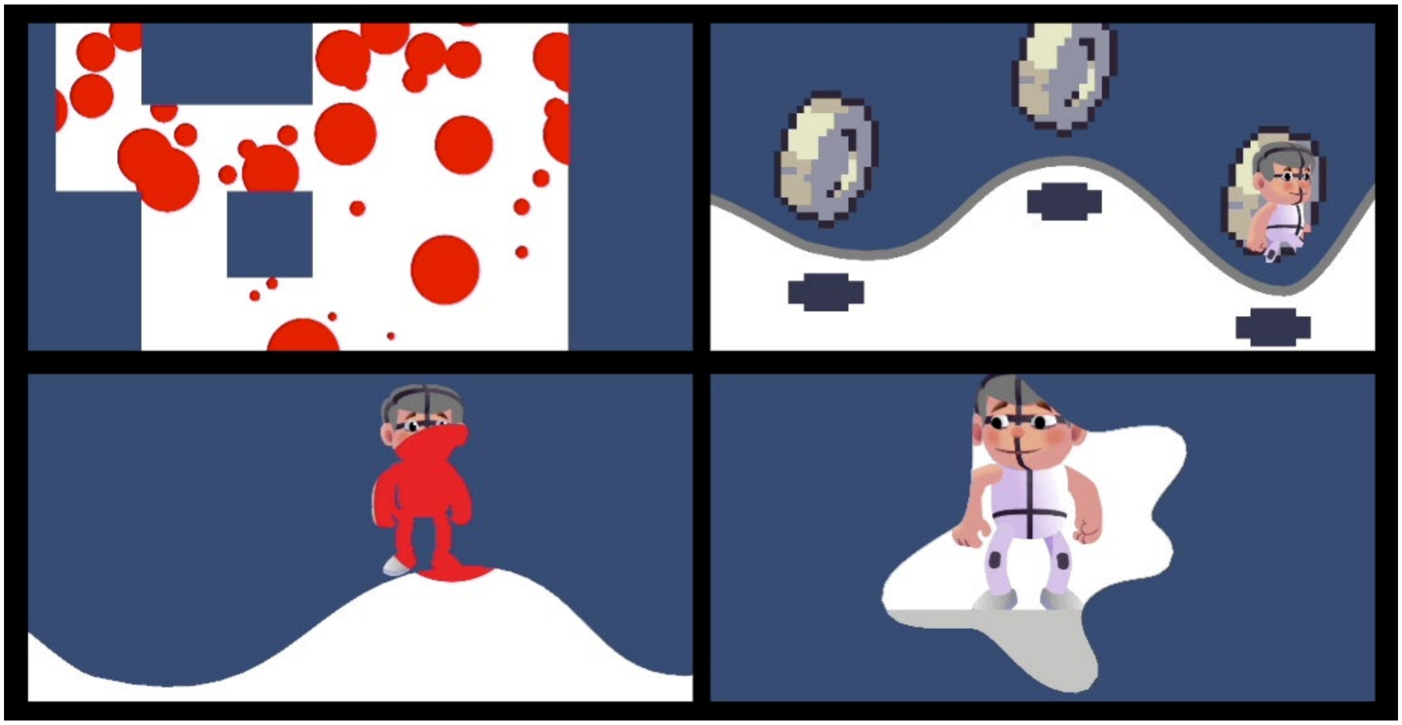


Sprite Mask

Sprite	Circle
Alpha Cutoff	1
Custom Range	✓
Front	
Sorting Layer	Characters
Order in Layer	10
Back	
Sorting Layer	Characters
Order in Layer	0
Sprite Sort Point	Pivot
Rendering Layer Mask	Layer1

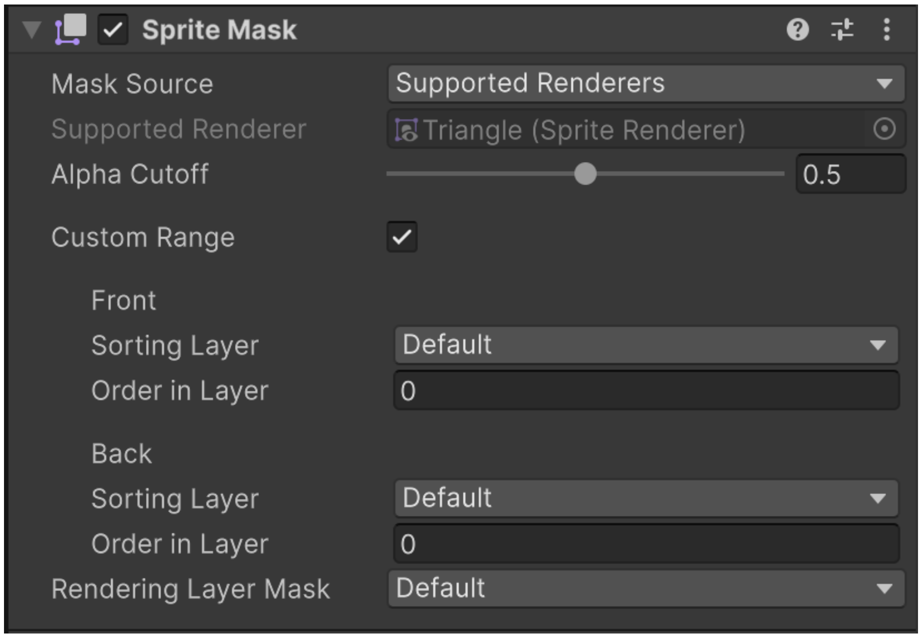
A portal effect created with a sprite mask and the Built-In Particle System

In previous versions of Unity, the sprite mask was defined by a specific sprite that would be used as the mask. While this already makes many effects possible, in Unity 6, the mask system is greatly improved.



Some examples shown at Unite Barcelona 24 in the 2D workflows talk

The Sprite Mask component can use any 2D renderer as the Mask Source. This includes sprites, tilemaps, sprite shapes and even characters rigged to skeletons – any 2D object can become a mask, even when animated, unlocking new creative possibilities.



Sprite Mask

Mask Source	Supported Renderers
Supported Renderer	Triangle (Sprite Renderer)
Alpha Cutoff	0.5
Custom Range	✓
Front	
Sorting Layer	Default
Order in Layer	0
Back	
Sorting Layer	Default
Order in Layer	0
Rendering Layer Mask	Default

For even further control, you can select what sprites will be affected by the mask, by sorting layer, or order in layer.

Sprite Library Asset

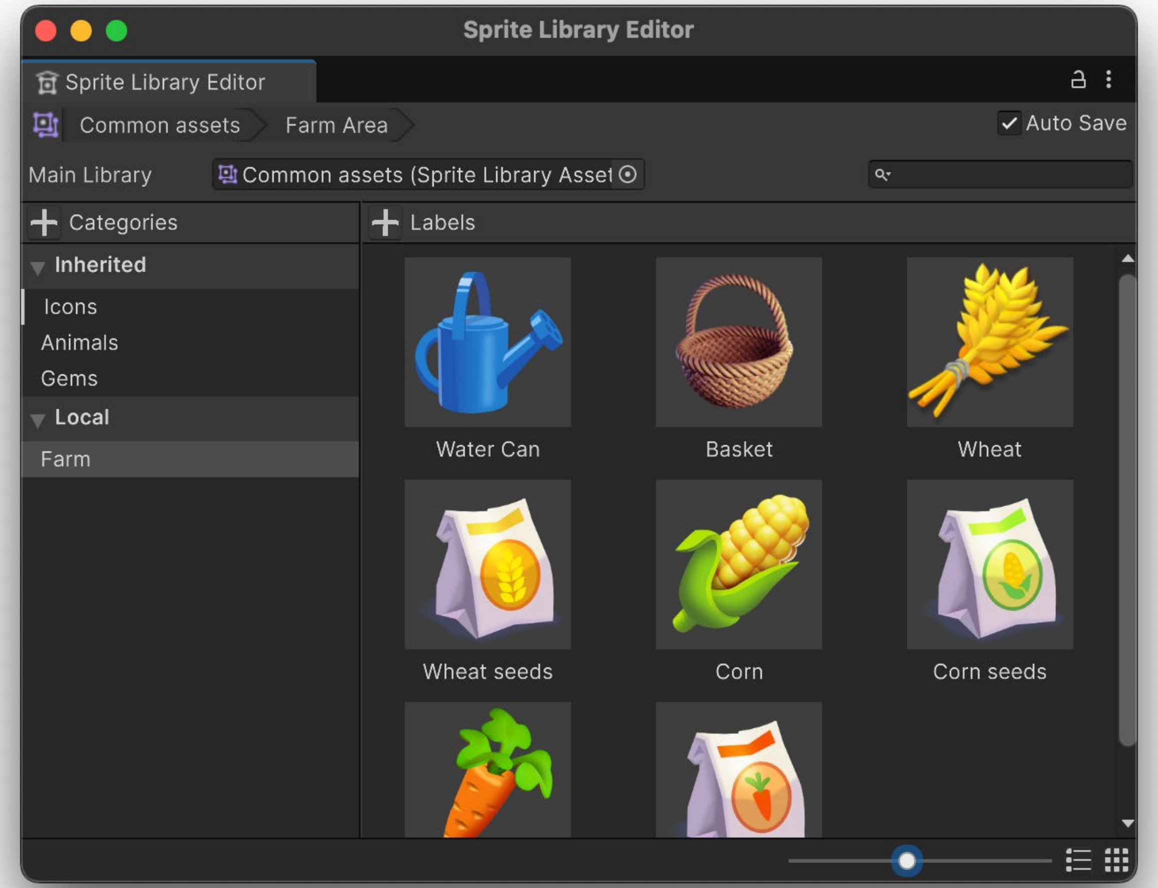
You can build a collection of sprites grouped by category with a Sprite Library Asset. A Sprite Library provides an easy and visual way to swap sprites in the Editor or at runtime based on their category, which can be very useful if the sprite count of your projects becomes very large (the sprite count of a 2D indie game with a lot of hand drawn animation can run into the tens of thousands).

You can group types of sprites by categories such as:

— **Gameplay purpose:** For example a category weapon could have sprites labelled as Sword, Axe, or Mace.

— **Animation**: A character's left-hand category could contain the hand drawn in different positions, or with different visual accessories, like in a hats category

— **Level design**: This category could include elements like background buildings – apartments, skyscrapers, or offices.

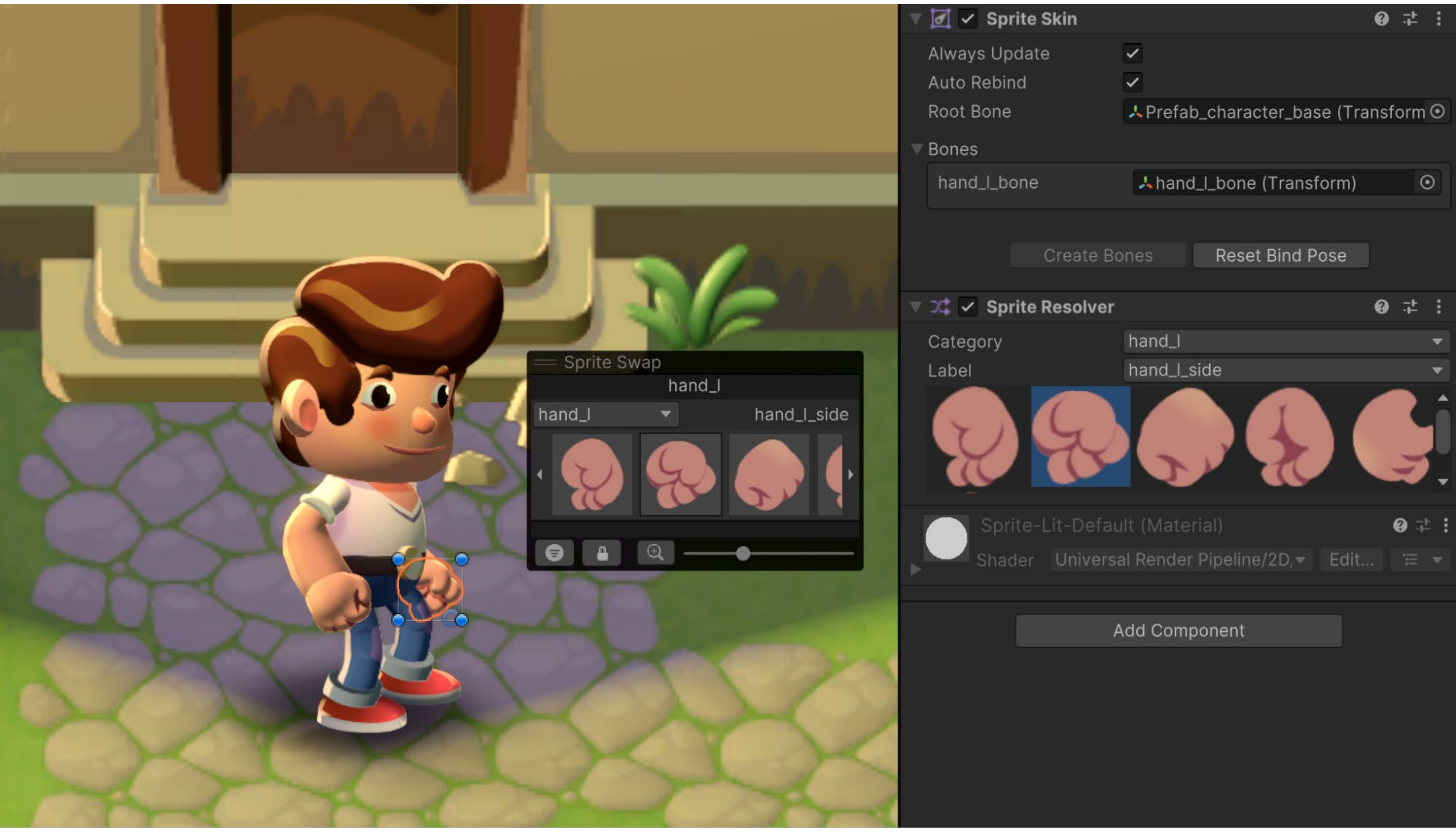


The Sprite Library Editor window

The Sprite Resolver component

One of the core use cases for sprite libraries and their companion component, Sprite Resolver, is to set up and sprite swap the animation of multi-part characters.

In Unity 6 there is a new **Sprite Swap overlay** to the Scene View. This overlay reveals all the relevant labels that a selected sprite of a certain category can swap to.



Sprite Skin
Always Update ✓
Auto Rebind ✓
Root Bone: Prefab_character_base (Transform
Bones
hand_l_bone: hand_l_bone (Transform)

Sprite Resolver
Category: hand_l
Label: hand_l_side

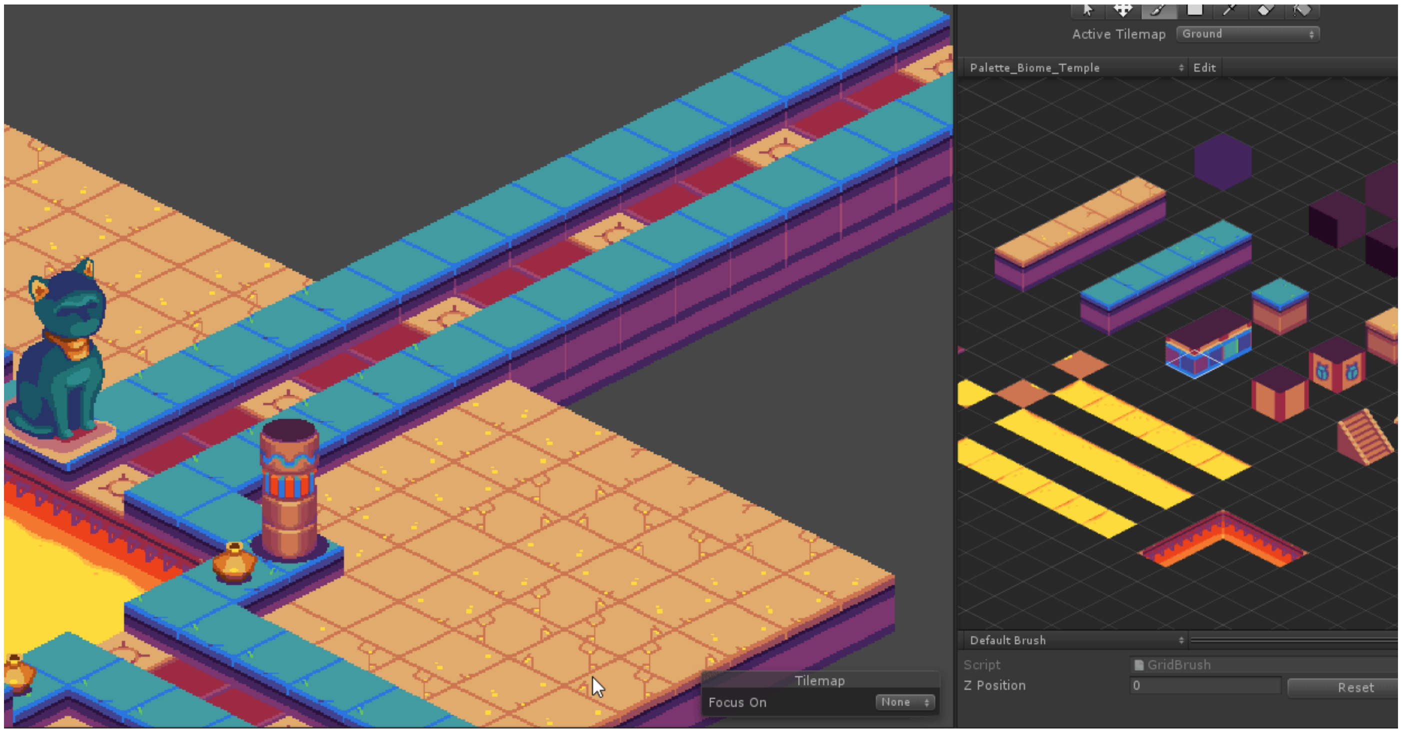
Sprite-Lit-Default (Material)
Shader: Universal Render Pipeline/2D,

The overlay is handy during the animation or level design stage, enabling you to select between sprites of the same category without having to search in the Project window.

Read all about Sprite Swap and the Sprite Library Asset in the documentation and start using them early on in the development cycle to keep your libraries organized as the project scales.

2D tilemaps

Tilemap offers a way to create a game world using small sprites called tiles, that you place on a grid. Instead of laying out a game world that is one big image, you can split it into brick-like chunks that are repeated through a whole level.



Active Tilemap Ground

Palette_Biome_Temple

Default Brush

Script GridBrush

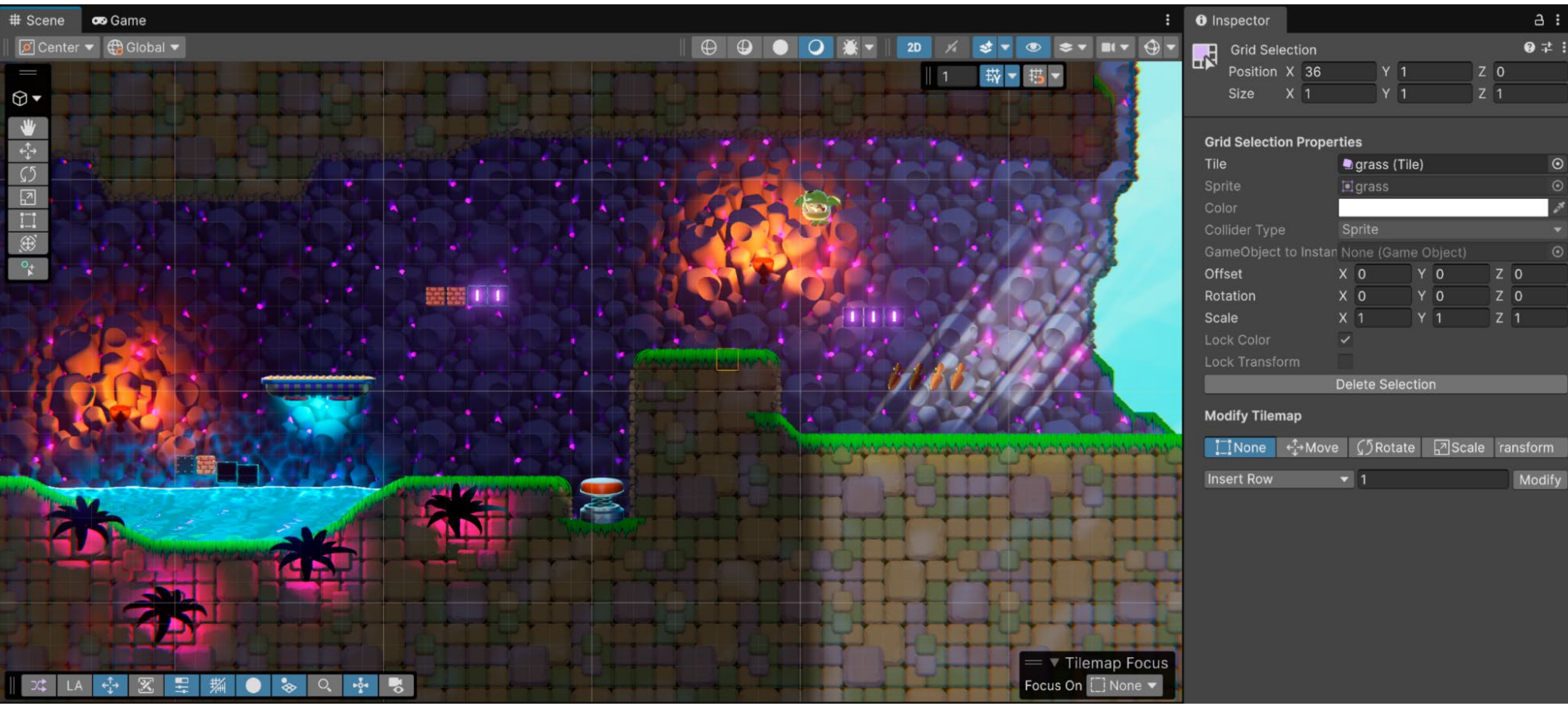
Z Position 0

Focus On None

Unity supports three types of tilemaps: Isometric, hexagonal or rectangular.

Create a tilemap

Tilemaps can help save on memory and CPU power because tiles that are not visible on the screen can be disabled. In Unity, a brush tool makes it efficient to paint them on a grid, and they can also be scripted to use some painting rules. They come with automatic collision generation for efficient testing and editing.



Grid Selection
Position X 36 Y 1 Z 0
Size X 1 Y 1 Z 1

Grid Selection Properties
Tile grass (Tile)
Sprite grass
Color
Collider Type Sprite
GameObject to Instan None (Game Object)
Offset X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Lock Color ✓
Lock Transform
Delete Selection

Modify Tilemap
None Move Rotate Scale ransform
Insert Row 1 Modify

Tilemaps in Bunny Blitz

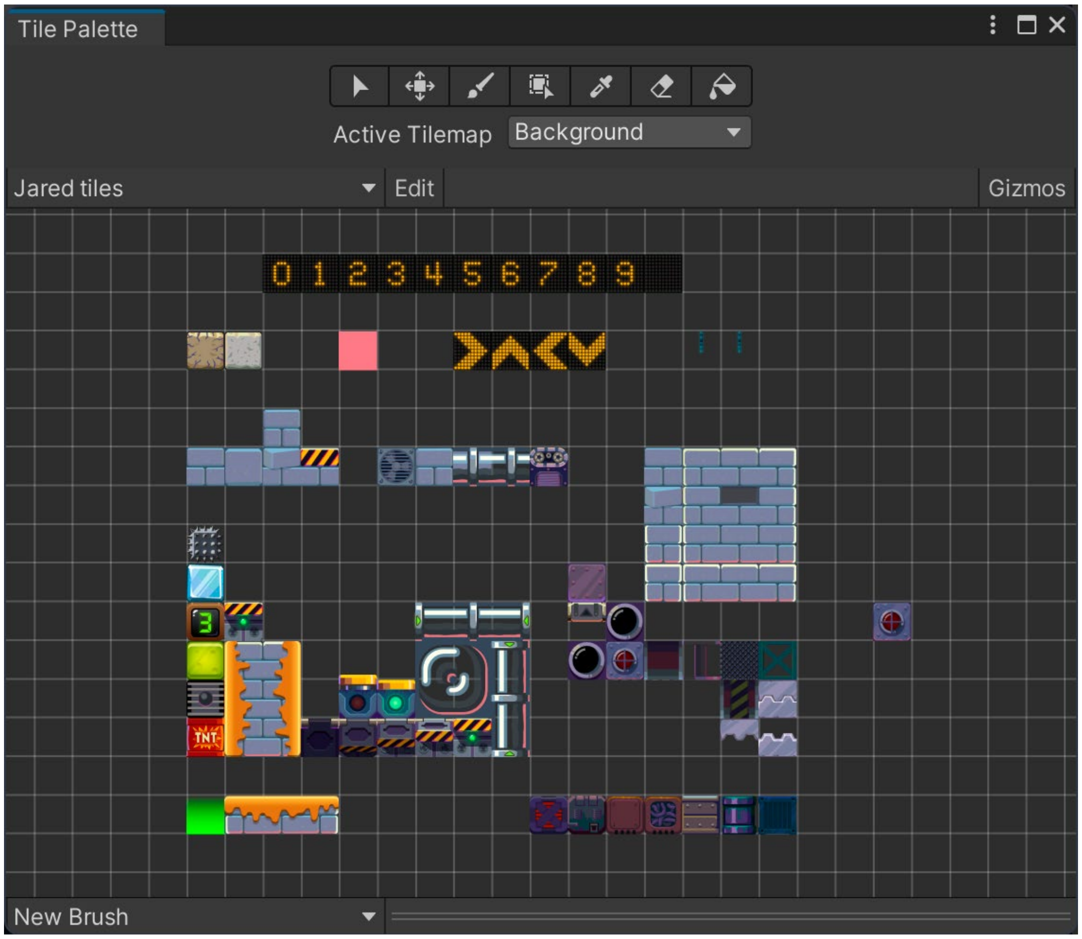
By default, the Tilemap package isn't included in the Editor, so you must download the 2D Tilemap package from the Package Manager, or start the project from the 2D template.

Along with the Tilemap package, you should also install the 2D Tilemap Extras. This provides reusable 2D and tilemap Editor scripts that you can use for your own projects, as well as providing features for creating custom brushes and tiles.

You can read more about the Scriptable Brushes and Scriptable Tiles included in the package.

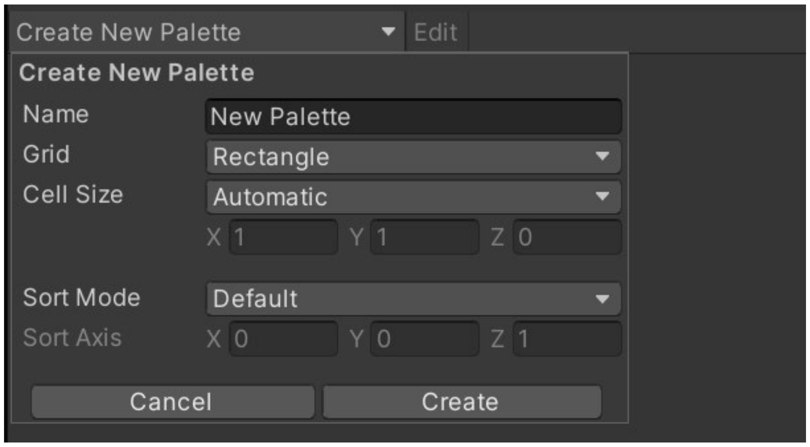
Once both packages are installed, open the **Tile Palette** window via **Window > 2D > Tile Palette**.

This window provides all the tiles and tools to help you paint or edit tilemaps.



Tile Palette window with a palette loaded

Create a new palette by clicking the **Create New Palette** button. A drop-down window with options will appear. Give the palette a name, set its options, click **Create**, and save it to a selected folder.

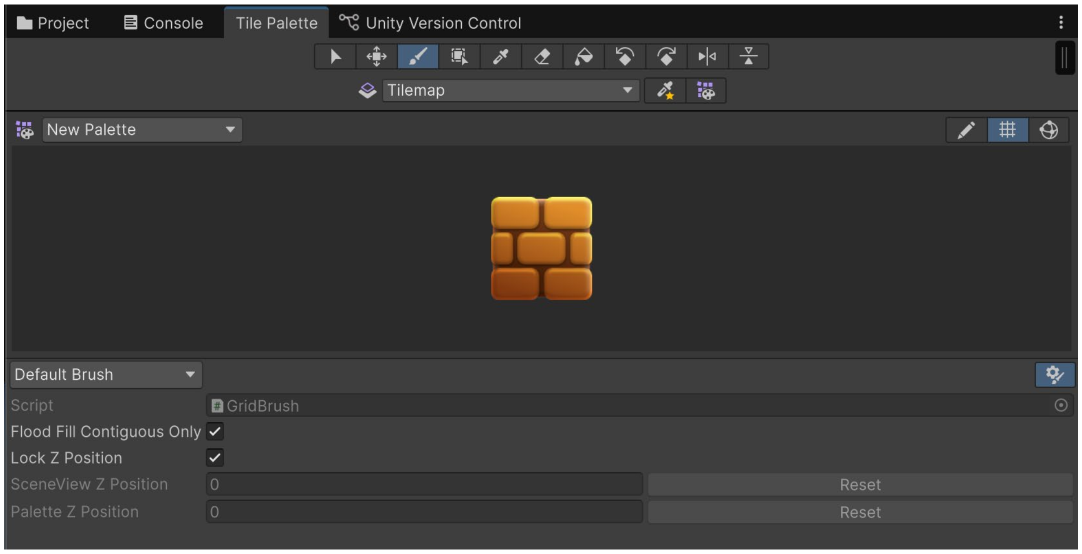


Create New Palette

Name	New Palette
Grid	Rectangle
Cell Size	Automatic
	X 1 Y 1 Z 0
Sort Mode	Default
Sort Axis	X 0 Y 0 Z 1

Creating a new palette

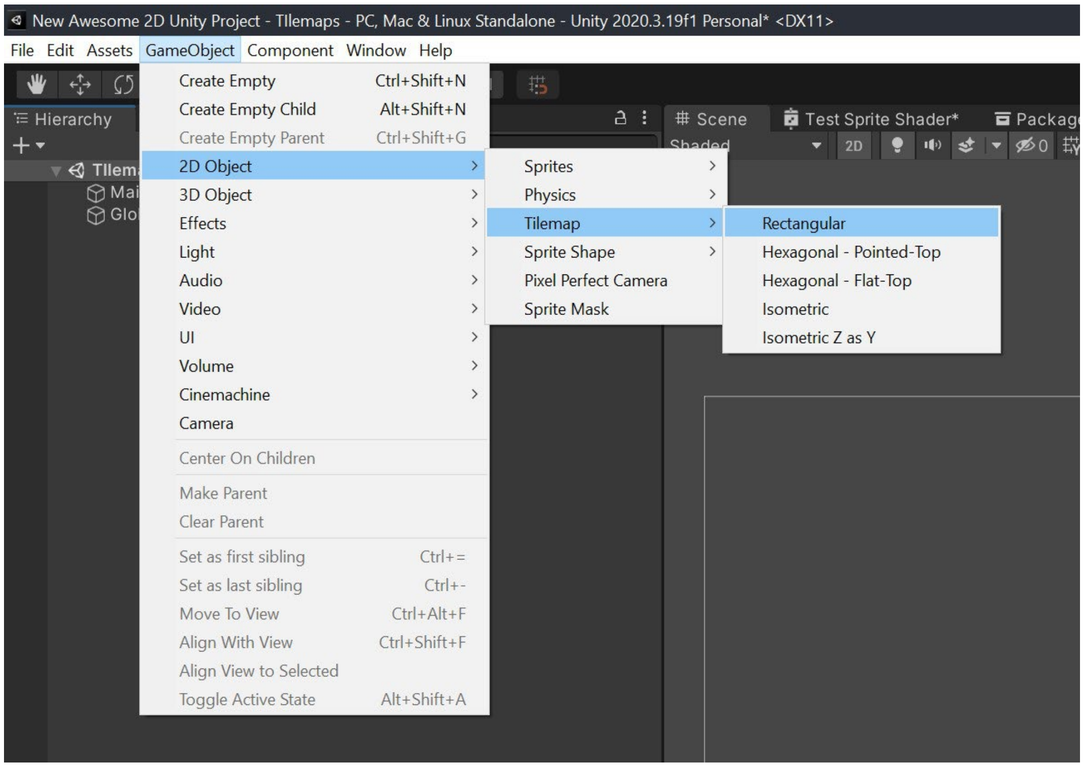
Now, add tiles to the palette. Drag a sprite, texture, or Tile Asset into the Palette window, (you might need to save the Tile Asset first if you dragged a sprite). You should now see your sprite in the grid of the palette.



Script: GridBrush
Flood Fill Contiguous Only: ✓
Lock Z Position: ✓
SceneView Z Position: 0 — Reset
Palette Z Position: 0 — Reset

Dragging a sprite into a palette creates a new tile.

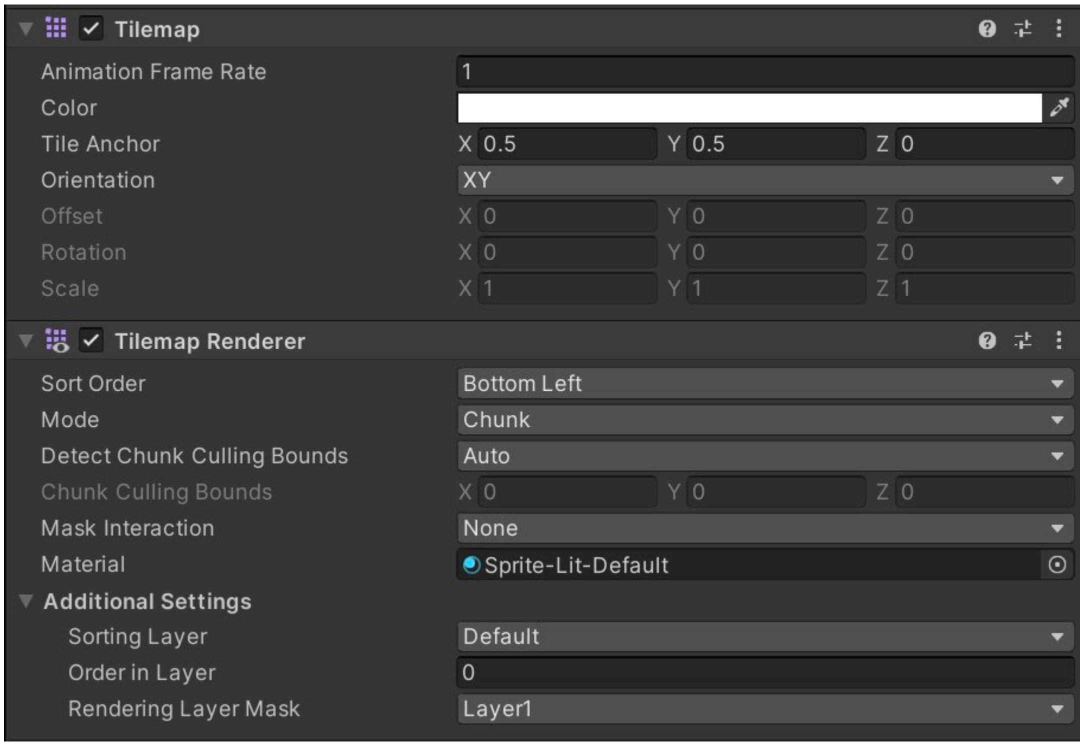
Now, create a tilemap to paint the tiles. Click the menu item **Gameobject > 2D Object > Tilemap > Rectangular**.


New Awesome 2D Unity Project - TIlemaps - PC, Mac & Linux Standalone - Unity 2020.3.19f1 Personal* <DX11>
File Edit Assets GameObject Component Window Help
Create Empty Ctrl+Shift+N
Create Empty Child Alt+Shift+N
Create Empty Parent Ctrl+Shift+G
2D Object
3D Object
Effects
Light
Audio
Video
UI
Volume
Cinemachine
Camera
Center On Children
Make Parent
Clear Parent
Set as first sibling Ctrl+=
Set as last sibling Ctrl+-
Move To View Ctrl+Alt+F
Align With View Ctrl+Shift+F
Align View to Selected
Toggle Active State Alt+Shift+A
Sprites
Physics
Tilemap
Sprite Shape
Pixel Perfect Camera
Sprite Mask
Rectangular
Hexagonal - Pointed-Top
Hexagonal - Flat-Top
Isometric
Isometric Z as Y
Hierarchy
Scene
Test Sprite Shader*

Creating a Tilemap

This creates a **Grid** GameObject and a Tilemap GameObject within it. Rename the Tilemap GameObject to something more descriptive. The Grid GameObject can hold multiple tilemaps, and it's possible to have multiple grids per scene, for example, with different cell sizes. For now, leave all the default settings of the Grid component.

The **Tilemap** GameObject has two components: **Tilemap** and **Tilemap Renderer**. Leave the settings as they are. If you need to, change the Sorting Layer settings to fit your layer structure.



Tilemap

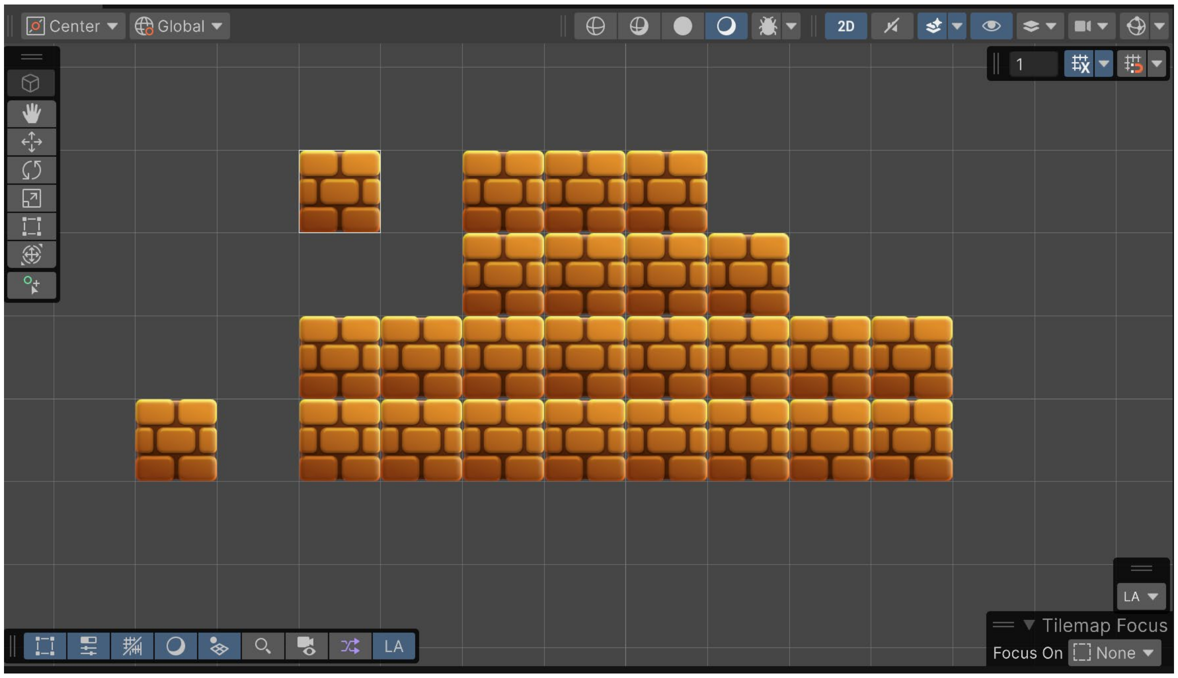
Animation Frame Rate	1
Color	
Tile Anchor	X 0.5 Y 0.5 Z 0
Orientation	XY
Offset	X 0 Y 0 Z 0
Rotation	X 0 Y 0 Z 0
Scale	X 1 Y 1 Z 1

Tilemap Renderer

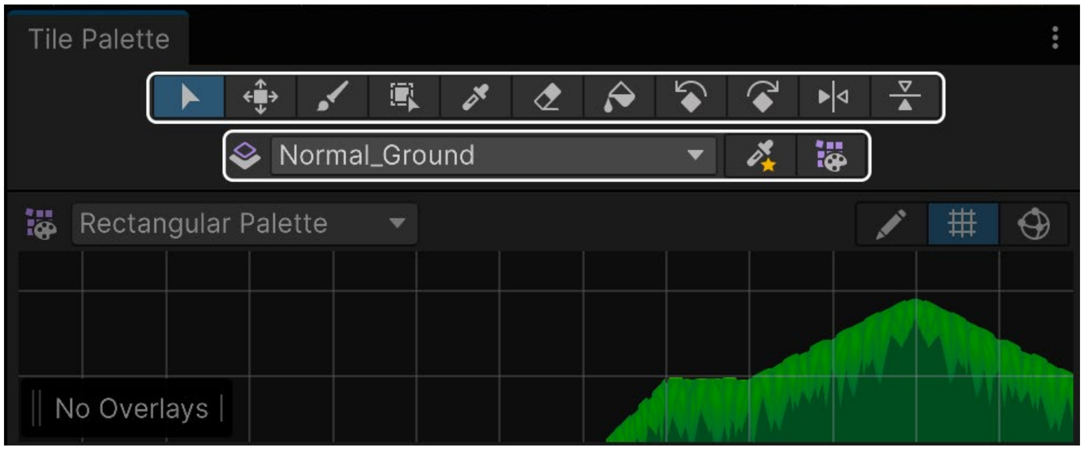
Sort Order	Bottom Left
Mode	Chunk
Detect Chunk Culling Bounds	Auto
Chunk Culling Bounds	X 0 Y 0 Z 0
Mask Interaction	None
Material	Sprite-Lit-Default
Additional Settings	
Sorting Layer	Default
Order in Layer	0
Rendering Layer Mask	Layer1

The Tilemap and Tilemap Renderer components

With your tilemap and tile on a palette, you are ready to start painting. Click on the brush tool in the Palette window toolbar, and select a tile to paint.



Simple brick tiles being painted on the tilemap



Tile Palette tools

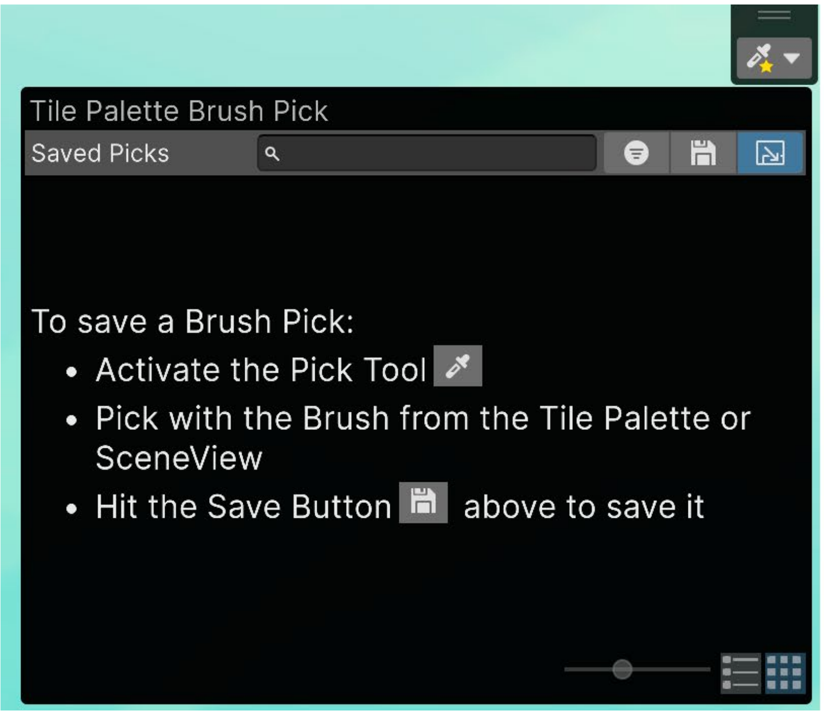
Let's look at the different Palette tools, starting with the top bar menu in the image above (the keyboard shortcuts are noted in brackets).

1. **Selection (S)**: Click to select one tile, or drag to select tiles within a rectangular area.
2. **Move (M)**: Move the selected tiles.
3. **Brush (B)**: Paint on an active tilemap (select one from the Active Tilemap dropdown) with the selected tile and brush.
4. **Fill Selection (U)**: Drag to fill a rectangular area using a selected tile.
5. **Tile Sampler (I)**: Pick a tile from a tilemap, and set it as active to paint.

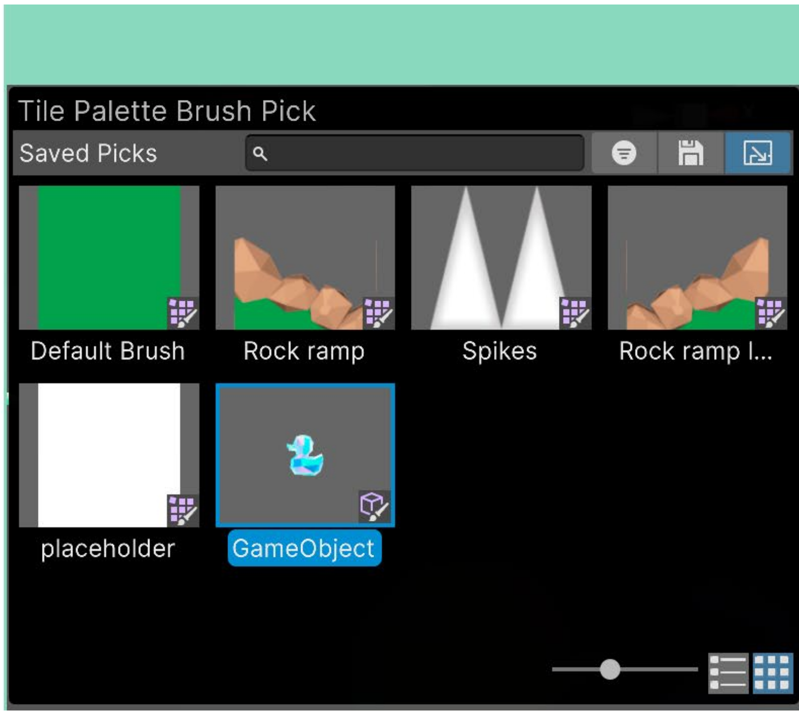
6. **Eraser (D)**: Delete tiles from a tilemap.
7. **Fill (G)**: Fill an area with a tile (the area needs to be bordered with other tiles otherwise a large rectangle will be drawn).
8. **Rotate left ([) and rotate right (])**: Turn the active tile in the brush 90 degrees clockwise or counterclockwise; this is useful for tiles that you use sporadically, for which you don't have all per-orientation variations.
9. **Flip horizontally (shift + [) and flip vertically (shift +]):** A similar use case to the rotate tool, the flip tool enables you to flip the tile to create quick variations of the active tile in the brush.

The tools included in the bottom bar menu (left to right) are:

1. **Active target**: Select the tilemap on which you'll be painting.
2. **Tile Palette Brush Picks (')**: Enable or disable the tool overlay on the Scene View; this allows you to save tiles for quick selection, including rotated and flipped tiles, and even GameObjects.
3. **Tile Palette Overlay (;)**: Enable or disable the tile palette overlay on the Scene View.

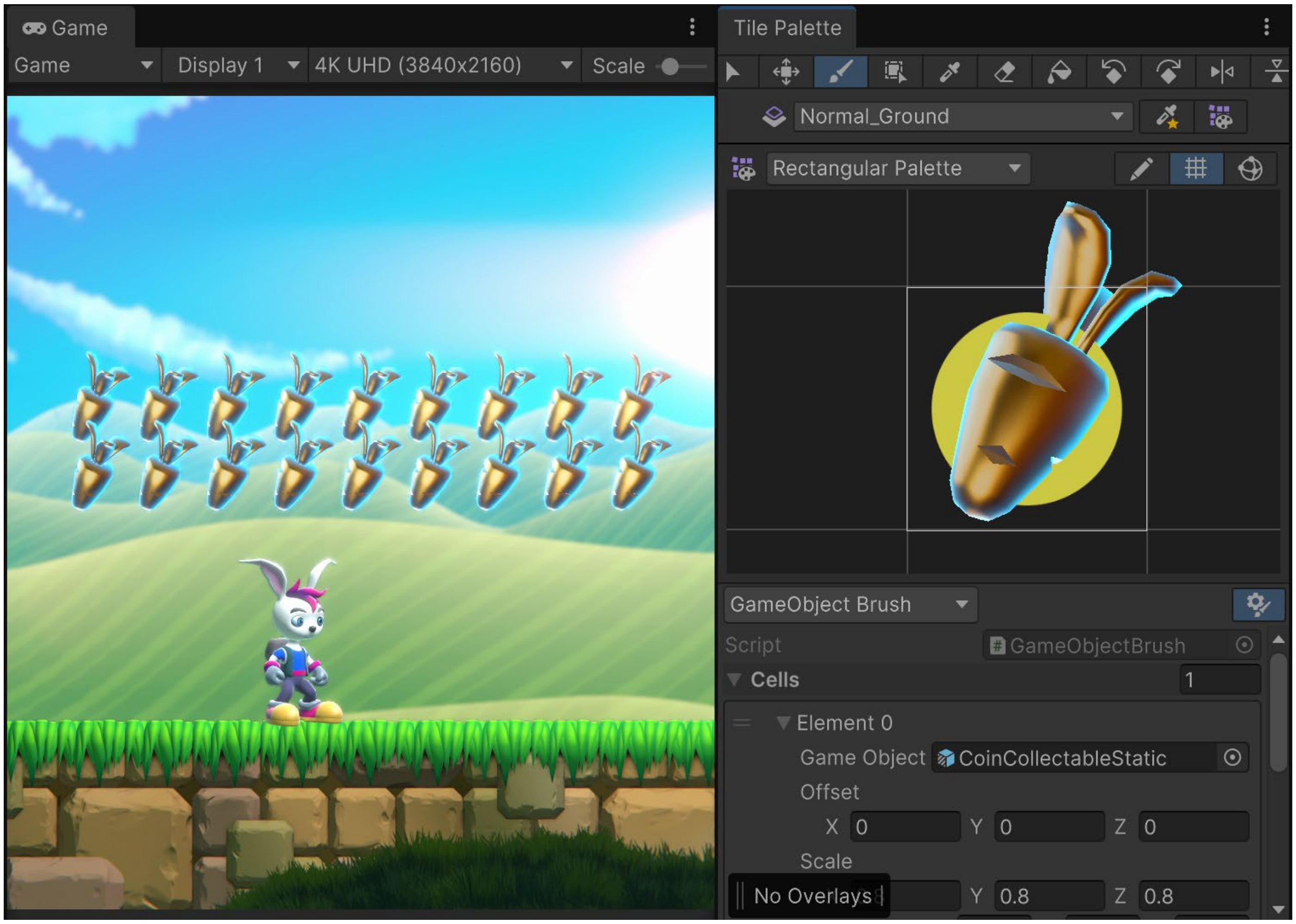

Tile Palette Brush Pick
Saved Picks

To save a Brush Pick:
- Activate the Pick Tool
- Pick with the Brush from the Tile Palette or SceneView
- Hit the Save Button above to save it



The Tile Palette Brush Picks overlay is a handy way to bookmark your most used tiles for quick access.

With these tools, you can paint and edit tiles efficiently. Additionally, the Tilemap Extras Asset provides useful scripts such as **Rule Tile** (see the following section for an example of Rule Tile in action) or the **GameObject Brush** to place GameObjects on the tile palette and paint them on the scene.



Script GameObjectBrush
Cells 1
Element 0
Game Object CoinCollectableStatic
Offset
X 0 Y 0 Z 0
Scale
Y 0.8 Z 0.8

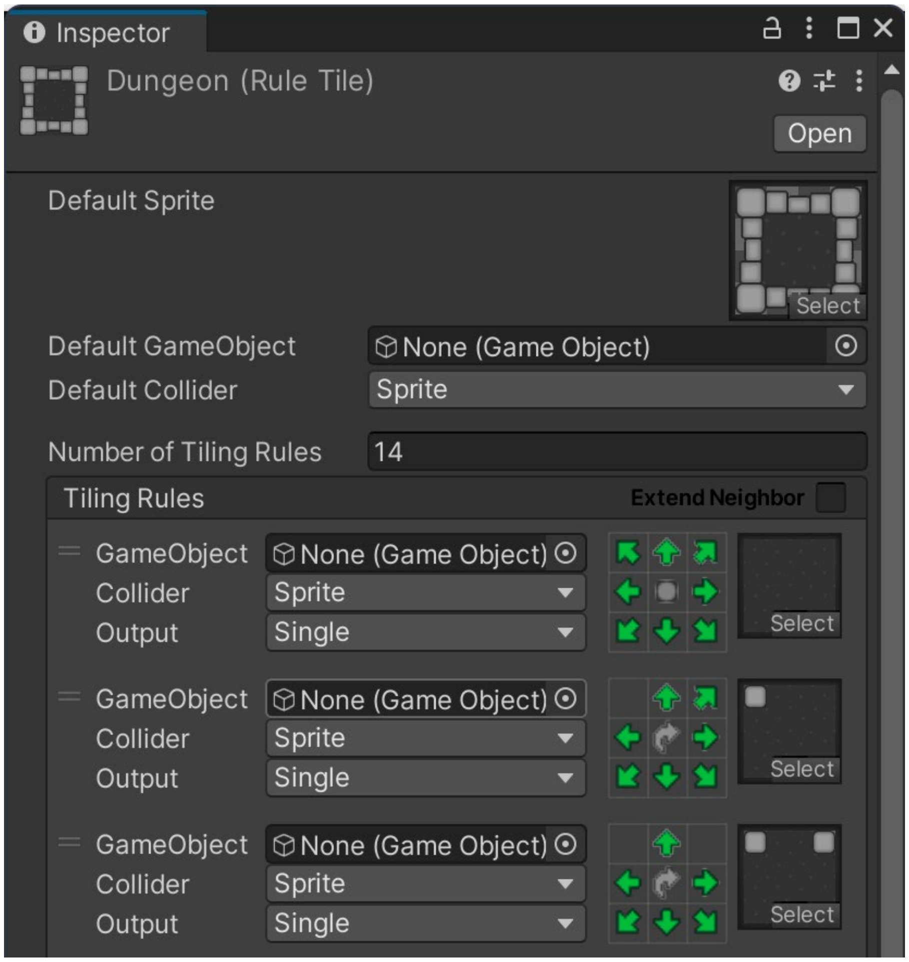
A GameObject Brush is used to place collectable objects in the level using the tilemap grid.

Rule Tile

The Tile Palette tool is used to efficiently populate the grid of each tilemap with tiles. But in some cases, manually placing the edges or corners of the shapes formed by a path can be time-consuming and prone to error. You would need to repeat the manual work every time changes were made to the path or any other tile-based shape. To avoid that you can use the Rule Tile feature, which does the job of painting the correct border tiles based on the shape's neighboring tiles.

Rule Tile, which is included in the 2D Tilemap Extras package, is a scripted tile that recognizes its surroundings and selects the appropriate image, for example, a ground tile with grass on top and a shadow on the bottom.

Rule Tile automatically recognizes where adjacent tiles are and where there should be borders. It chooses the proper sprite for you, so you don't have to select different tiles to paint the borders.


Inspector

Dungeon (Rule Tile)

Open

Default Sprite

Default GameObject: None (Game Object)

Default Collider: Sprite

Number of Tiling Rules: 14

Tiling Rules — Extend Neighbor

GameObject	None (Game Object)	Select
Collider	Sprite	
Output	Single	
GameObject	None (Game Object)	Select
Collider	Sprite	
Output	Single	
GameObject	None (Game Object)	Select
Collider	Sprite	
Output	Single	

The Rule Tile asset in the Inspector

The Rule Tile Inspector provides a list of rules for which sprite to choose based on the adjacent tiles. There's a matrix and a sprite on the right side of each rule. The sprite is used when neighbor tiles are on all sides where the green arrows are pointing. Learn more about the Rule Tile feature here.

AutoTile in Unity 6.3

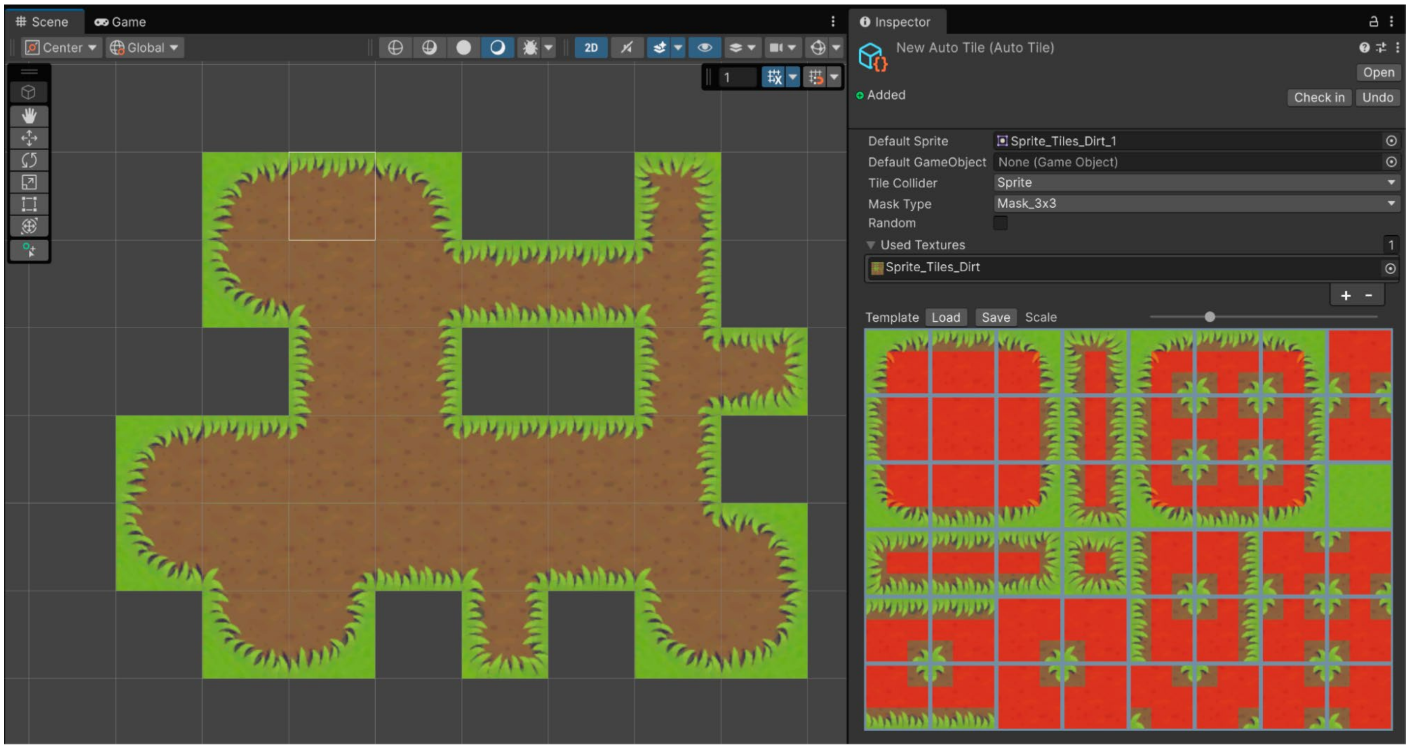
Introduced in Unity 6.1, AutoTile is a tile asset that displays the right neighbouring tile based on a mask that you define in the AutoTile asset settings.

Compared to Rule Tile, AutoTile provides a more intuitive setup and the possibility to create templated variations of a tile sheet without having to set the mask again.

The steps to create an AutoTile are:

1. Import a tilesheet into your project.
2. Slice the tilesheet in the Sprite Editor. In the example shown below, the tilesheet is 8×6: 8 columns and 6 rows. Apply the changes.

3. Create a new AutoTile in the Project window.
4. In the Inspector, add the sliced tilesheet in the **Used Textures** field.
5. Start masking with the paint brush by filling up the hollow areas of the tiles. If a tile has the same masking as another tile, it will be highlighted, allowing you to double check that the masking is correct.
6. Test the AutoTile by painting in a tilemap.



New Auto Tile (Auto Tile)

Default Sprite: Sprite_Tiles_Dirt_1
Default GameObject: None (Game Object)
Tile Collider: Sprite
Mask Type: Mask_3x3
Random
Used Textures: 1
Sprite_Tiles_Dirt

An example of AutoTile from *Happy Harvest*

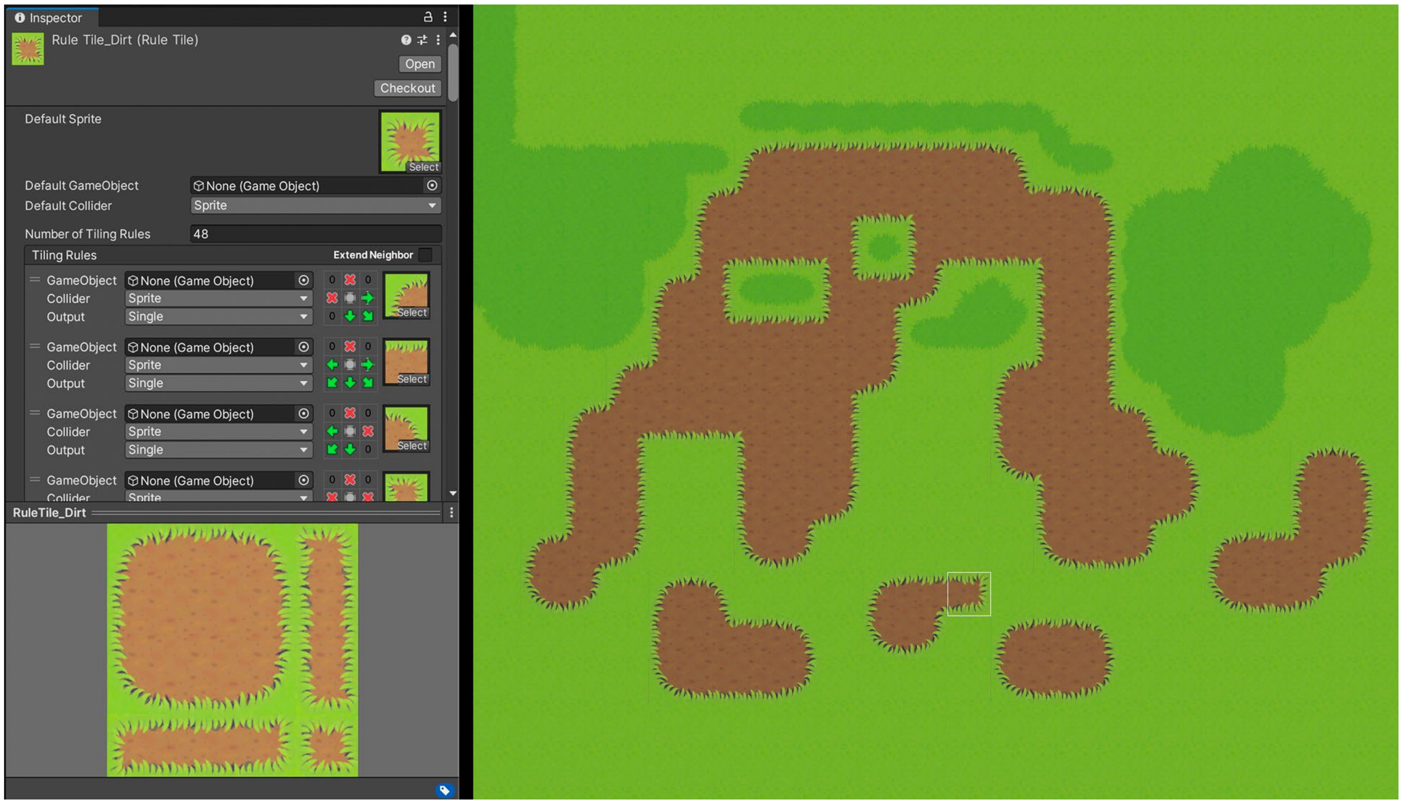
You can load additional textures in the Used Textures field. For example, you can load a variant of the same tilesheet and then check the **Random** option to randomly pick between tiles matching the same neighbouring criteria, helping you break the illusion or a repeated pattern in a big scene.

Tilemaps in Unity 2D samples

Efficient environment design

Rule tiles are used extensively throughout *Happy Harvest*, for example, to create soil patches with grass. In the sample's Project window is a tile palette named **Palette_Tiles**. In the first row of the palette are rule tiles. If you've downloaded and opened the project, you can select the GameObject in the Inspector so the asset will be shown in the Project window and then select this asset to see its Rule Tile setup.

Note: If applicable to your project, you can save some time by reusing these tile assets or painting over the textures for your own use. You can also replace the sprites that come with this sample or create a new Rule Tile that matches your game's needs.



Rule Tile_Dirt (Rule Tile)

Default Sprite

Default GameObject: None (Game Object)

Default Collider: Sprite

Number of Tiling Rules: 48

Tiling Rules — Extend Neighbor

GameObject: None (Game Object)
Collider: Sprite
Output: Single

GameObject: None (Game Object)
Collider: Sprite
Output: Single

GameObject: None (Game Object)
Collider: Sprite
Output: Single

GameObject: None (Game Object)
Collider: Sprite

RuleTile_Dirt

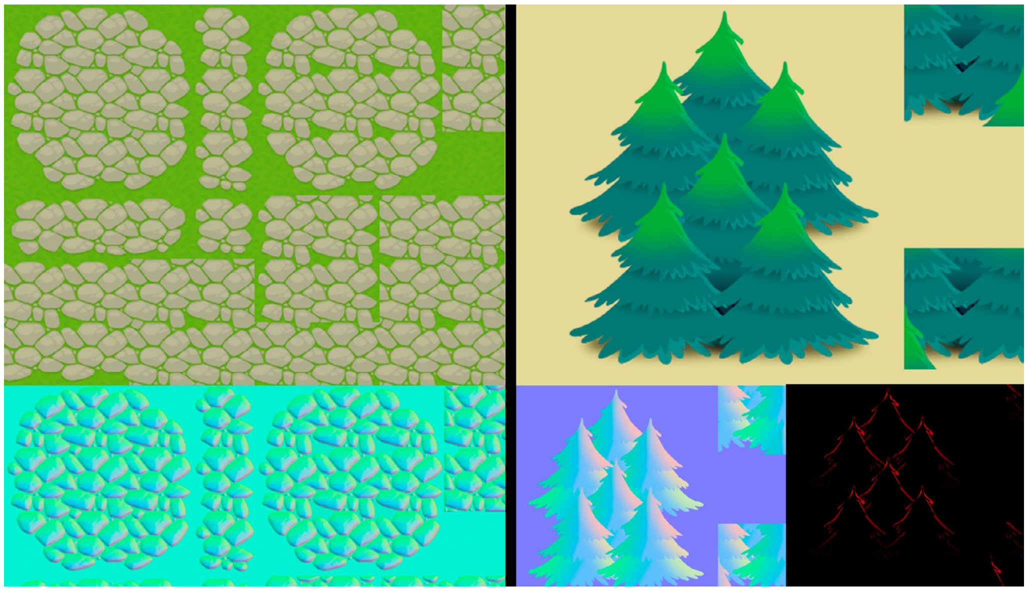
Once you have set up rule tiles or auto tiles, painting continuous paths will be much easier and time efficient.

This sample was made before the Auto Tile feature was introduced, but the same functionality can likely be replicated with Auto Tile as described in the section before.

Secondary textures for tilemaps

In *Happy Harvest*, every tilemap under the Tilemaps folder has its counterpart normal map and mask map textures that share the same dimensions and layout but are painted for displaying the lighting.

The normal map and mask map textures are added to the main texture of the tile set in the Sprite Editor. In the sample, mask maps are used to create rim light silhouette effects for the character and the props. However, since most of the tiles are to be used for the ground they don't need a rim light effect. This is why most of the texture of this tile set is black, to avoid it reflecting any light created for the rim effect. An exception was made for the building tiles used for the roof of the buildings; in this case, there was a need to highlight the edges of these tiles.



A more intricate tilesheet for the stone path on the left and a simpler tilesheet on the right for background props.

Level and gameplay design with tilemaps

All of the tiles in *Happy Harvest* are the same size and shape and contained in the same Grid GameObject. This helps to keep the number of tilemaps in the game low. A component called **Terrain Manager** is attached to the Grid GameObject and provides some gameplay possibilities (see the following section)

Here are some of the tilemaps under the Grid GameObject:

— **UnderwaterTiles:** These are used for the tiles that are seen under the water, for the cliff, and a green ground for the pond area.

— **Water:** These tiles use a Spite-Lit-Material shader, which is made in ShaderGraph to simulate the water animation.

— **GroundBase:** This is the ground artwork comprising tiles for the paths, grass, mud, and stone tiles to fill the gaps from the elevation tiles.

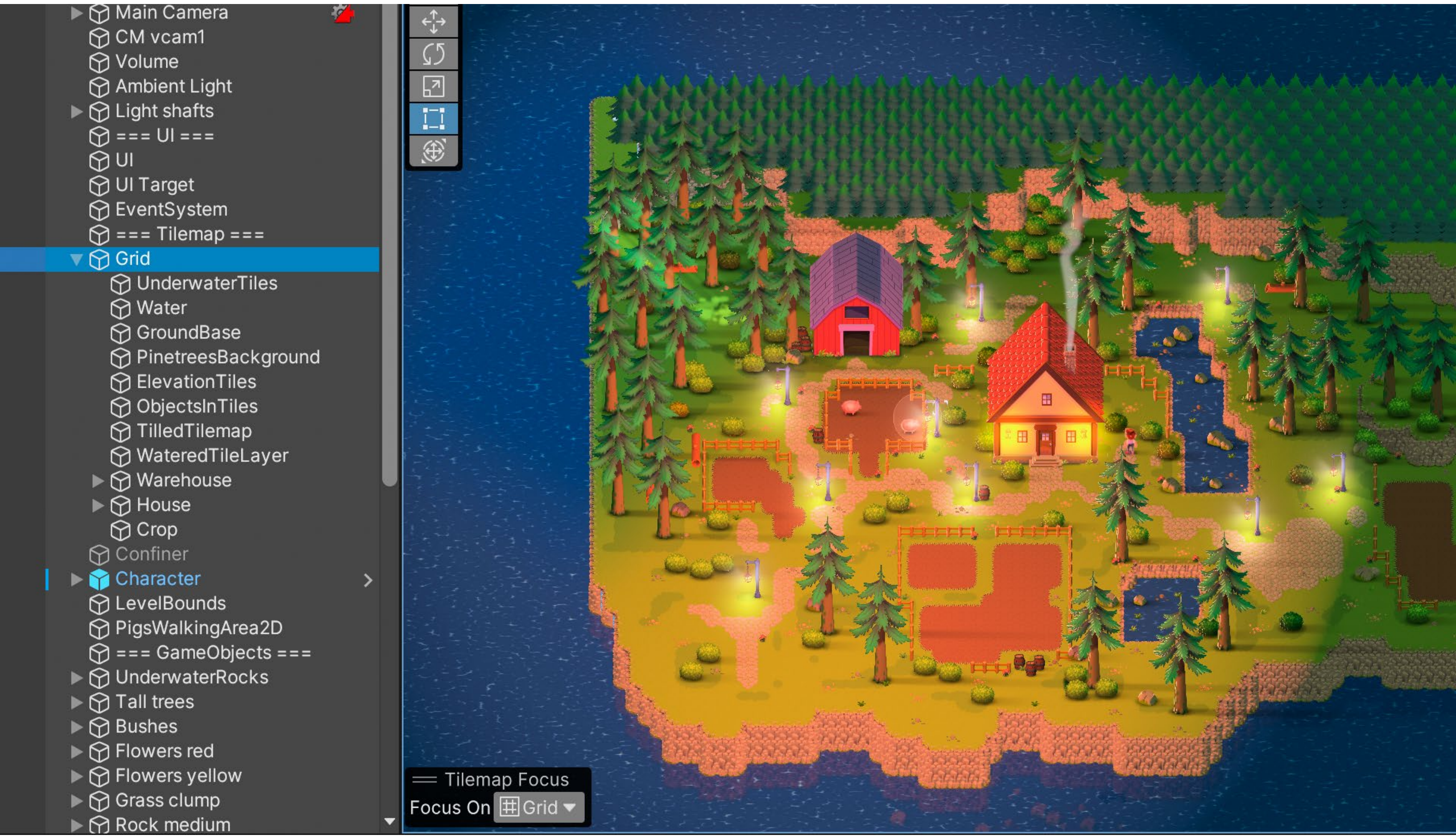
— **ObjectsInTiles:** This is used for GameObjects painted in a tilemap with the GameObject Brush from the Tile Palette, such as the fences.

— **TilledTilemap:** These tiles are used in the code to detect tiles that seeds can be planted in.

— **WateredTileLayer:** Similar to the previous tilemap, this is used by the game logic to make the tiles look wet; when the tiles dry, the “wet” tiles are removed.

— **Warehouse, House:** These tilemaps are used to create the warehouse and house. By creating the buildings with tilemaps, the artist had flexibility to reshape and resize them more efficiently, as well as to save on texture space, since parts of the building use the same texture.

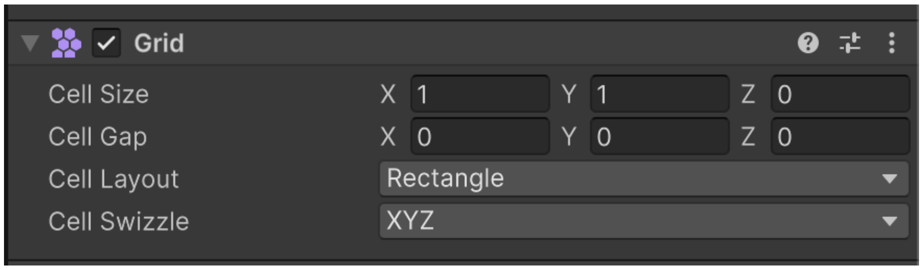
— **Crop:** This is used to keep all the plants under one GameObject, and is also used by the Terrain Manager.



The Grid GameObject containing the tilemaps; the Tilemap Focus tool can help you focus on the tilemap at hand by only drawing the contents of the selected tilemap in the Scene view

Tip: avoid texture bleeding on tilemaps

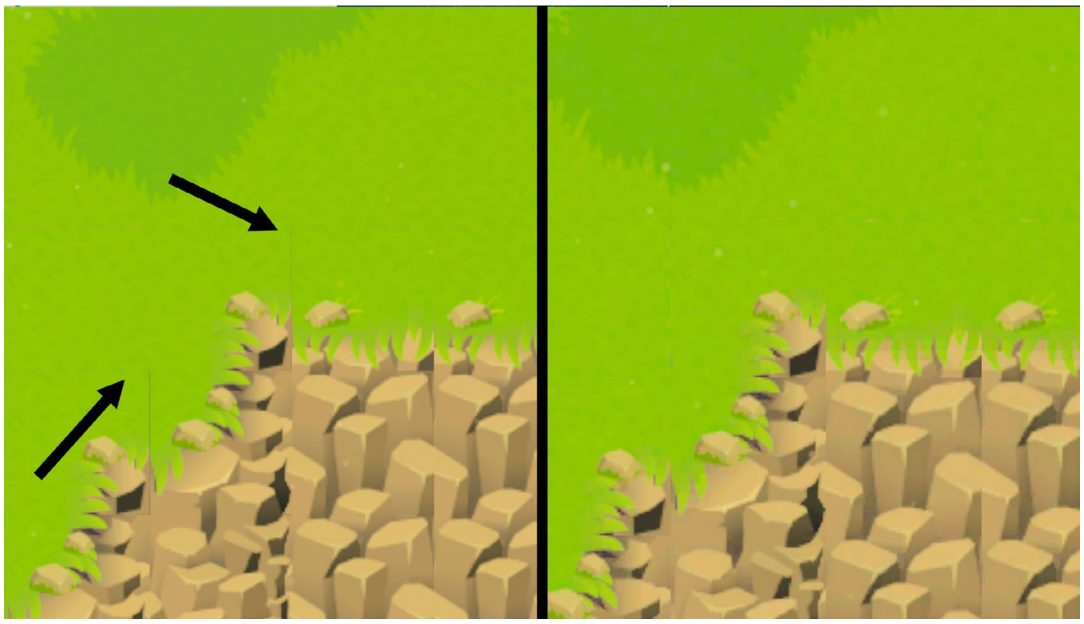
With all the effort put into art and tilemaps, you'll want to avoid the appearance of bleeding or small gaps in between tiles due to interpolation and smoothing of edges (this isn't an issue in pixel art games, where sprites are not smoothed). If you're not using a tile sheet then we recommend packing the sprites with Sprite Atlas, which will help to smoothen seams by not having a transparent end on the edge of the tile. Additionally, internal sorting in the Tilemap Renderer can perhaps help with enlarging the tile slightly to avoid gaps and create a bit of overlap.


Grid

Cell Size X 1 Y 1 Z 0
Cell Gap X 0 Y 0 Z 0
Cell Layout Rectangle
Cell Swizzle XYZ

Some options in the Grid component to help you mitigate possible gaps

Some default settings can work for most sprites but tilemaps might require a bit more tweaking. Features like avoiding rotation of sprites when packed or alpha dilation helps tiles maintain their sharp edges.


In the left image you can see some bleeding happening, which was fixed by adjusting settings in the Sprite Atlas.

Remember that if you are working with pixel art, changing the filter mode of the sprites to Point (no filter) should make tiles perfectly match without leaving any possible gaps.

Tilemap API for gameplay

Tilemaps are well-suited to grid-based games. The API provides an easy way to read and write tiles in a visual grid, becoming a powerful level design tool for authoring as well as game logic. Typically, grid-based gameplay uses multidimensional lists or dictionaries which only live in your code and require some sort of visual implementation and authoring tooling. Tilemap can provide all of that out of the box.

Tilemap API in Happy Harvest

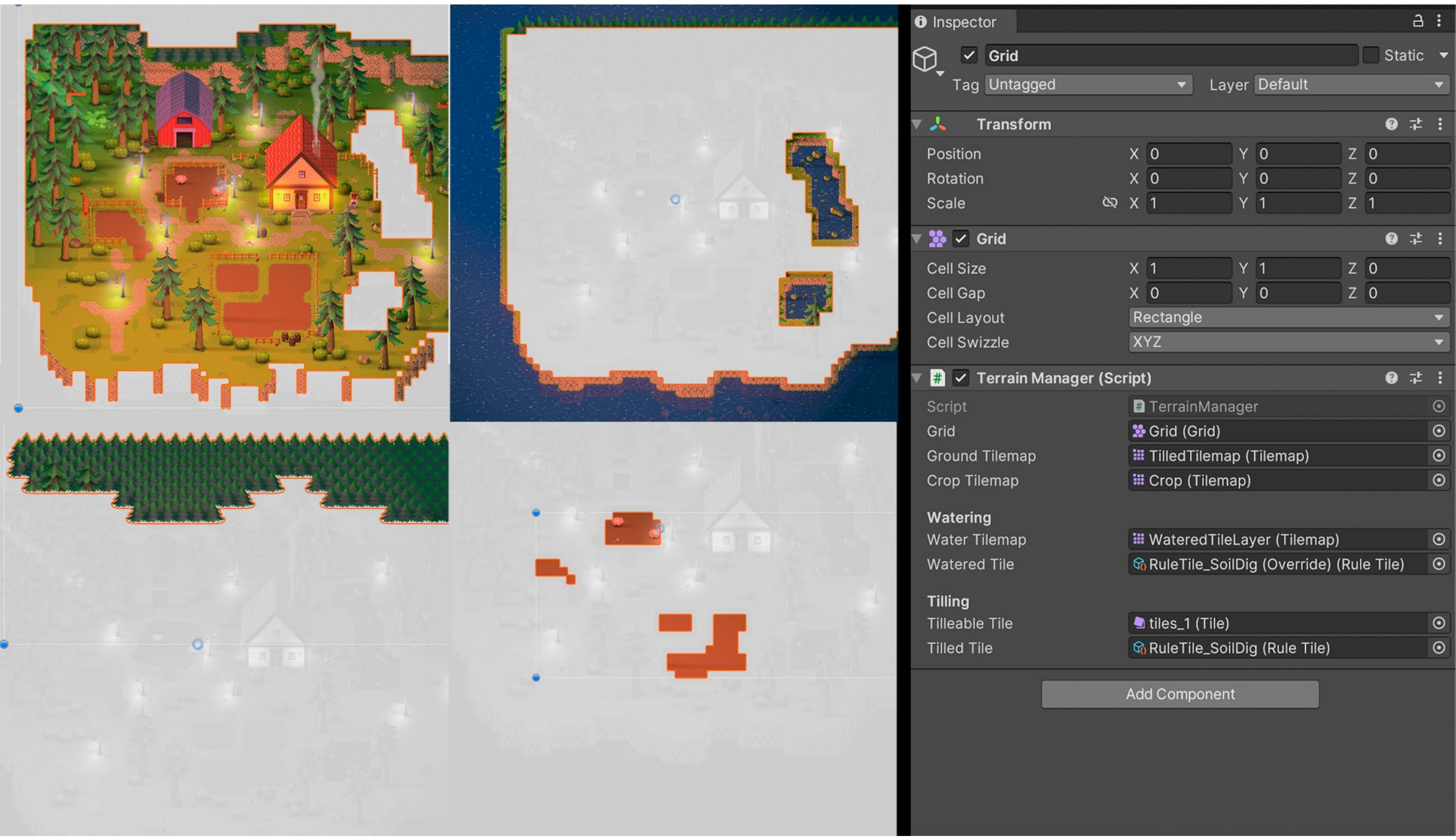
The **Terrain Manager** script, which is inside the Grid GameObject, is the main manager for handling changes in tiles and using the tiles to keep track of the plantation. It uses the Tilemap API which can be useful for setting up grid-based gameplay, helping you to easily identify the position of items in a two-dimensional position.

This Monobehaviour class creates two generic classes called **GroundData** and **CropData**. They include gameplay-related variables, such as the length of time the tiles appear wet, how fast a plant grows, or how long it lasts without harvesting before it dies.

The Terrain Manager script references the tilemaps and tiles that are used for gameplay purposes, including:

- **Ground Tilemap**: This references the tilemap containing the soil tiles that indicate where the player can dig and plant seeds. If you dig in any tile outside of this tilemap nothing will happen.
- **Crop Tilemap:** This references the Tilemap GameObject to be used as the parent object of the crops. With the Tilemap API we can place, update, and remove the crops tiles from this tilemap at runtime.
- **Water Tilemap:** This is used to paint over the tilled soil tiles that simulate water, visually communicating to the player that those tiles are ready for planting and growing.
- **Watered tile:** This is a rule override tile with the water graphic, which allows you to create Tiles that provide variations of a Rule Tile without setting new rules. It overrides the rules of tilled soil tiles.
- **Tilleable tile:** This is used to identify the tiles that can be used for digging.
- **Tiled tile:** This is the Rule Tile used to paint the visual for the tilled soil.

The Terrain Manager script includes functions that use the Tilemap API to read tile information in a VectorInt location format and update the tiles accordingly. Those functions are called from the different tools when the PlayerController triggers the function on them.



Grid — Static
Tag Untagged — Layer Default

Transform
	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Grid
	X	Y	Z
Cell Size	1	1	0
Cell Gap	0	0	0

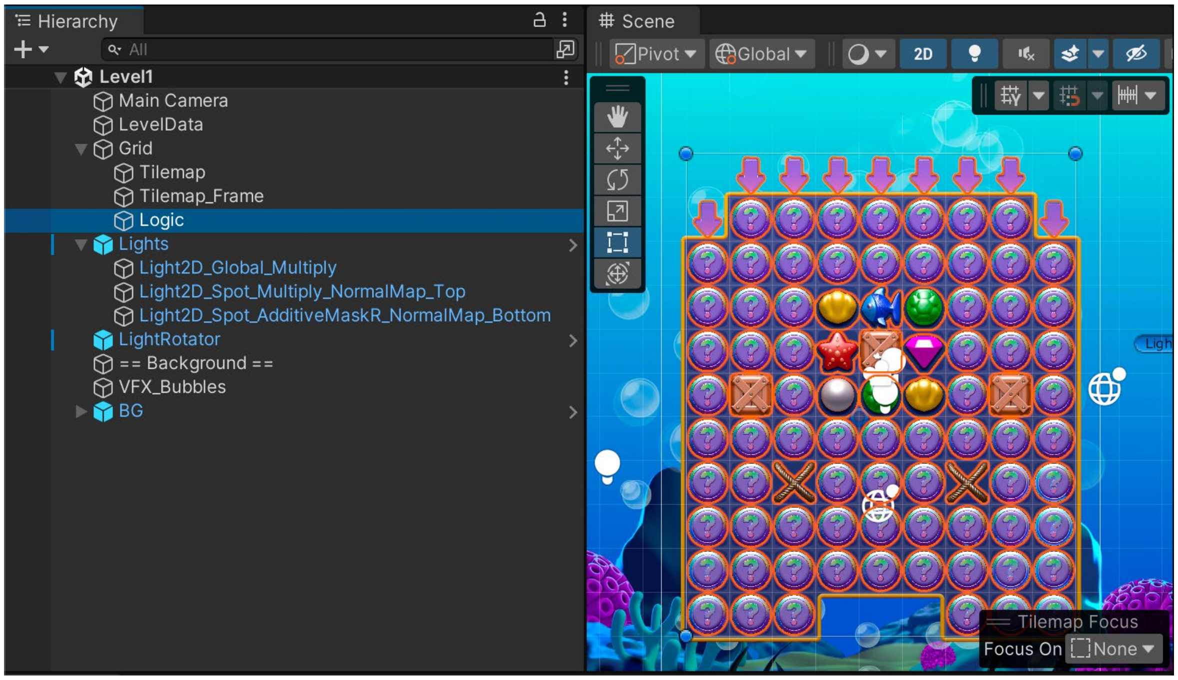
Cell Layout: Rectangle
Cell Swizzle: XYZ

Terrain Manager (Script)
Script	TerrainManager
Grid	Grid (Grid)
Ground Tilemap	TilledTilemap (Tilemap)
Crop Tilemap	Crop (Tilemap)
Watering	
Water Tilemap	WateredTileLayer (Tilemap)
Watered Tile	RuleTile_SoilDig (Override) (Rule Tile)
Tilling	
Tilleable Tile	tiles_1 (Tile)
Tilled Tile	RuleTile_SoilDig (Rule Tile)

The Terrain Manager system, which uses some of the tilemaps for gameplay logic

Tilemap API in Gem Hunter Match

Gem Hunter Match, an official Unity sample, is a match 3 puzzle game designed mainly for mobile, but able to run on multiple platforms.



Level scenes in Gem Hunter Match have a simple structure.

If you look at the level structure, you'll find tilemaps that are used for visual assets but also game logic. These are:

— **The Grid GameObject:** This GameObject contains a script called **Board** that determines the type of gems available in a level, as well as references to VFX Graph-based visual effects.

— **The child GameObjects**: There are a few child GameObjects included in the Grid GameObject. The child GameObjects include:

  — **Tilemap:** This is used to create the background tiles and decoration, with the Tilemap API offering convenient ways to track the position of the many pieces.

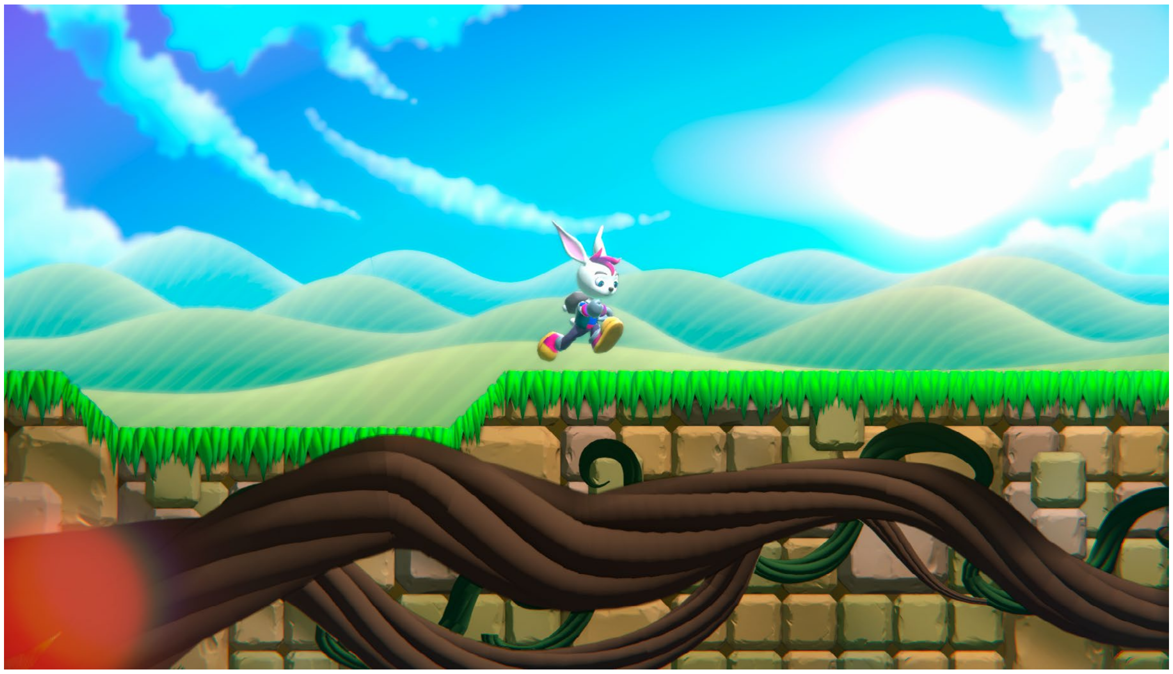
  — **Tilemap_Frame:** This is used to create the frame decoration around the background tiles.

  — **Logic:** This contains placeholder tiles for the level design; it's from here that all the placeholder "random gem" tiles (indicated by a question mark) are swapped out for actual gameplay GameObjects (generated randomly via code). However, you can also choose the type of gem or item to spawn on a given tile position, by using the other tiles in the palette.

Learn more about Tilemap best practices in this video.

2D Sprite Shape

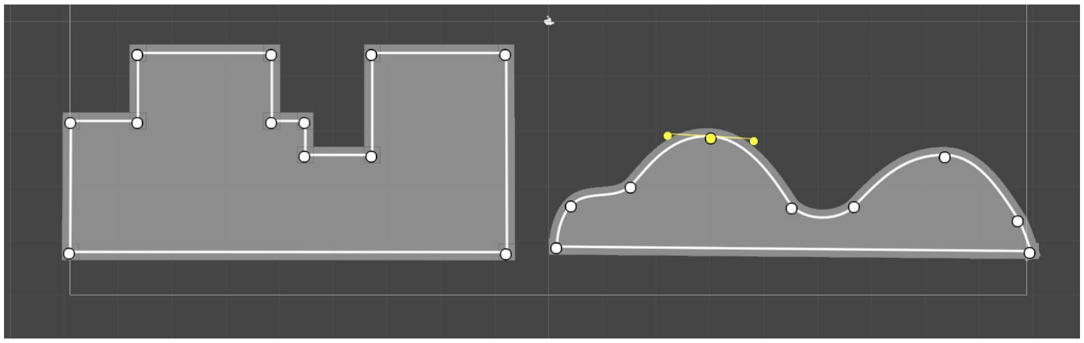
The 2D Sprite Shape tool features sprite tiling along a shape's outline that automatically deforms and swaps sprites based on the angle of the outline. Additionally, you can assign a Fill texture to a 2D Sprite Shape GameObject to create filled shapes with tiled textures as backgrounds or other large level-building props.



Bunny Blitz, a new Unity 2D sample project, has decorative elements made with 2D Sprite Shape, like the vines and the hills in the background.

The paths that you create with 2D Sprite Shape work similarly to the well-known pen tool used in graphics software. They are Bézier curves that can be edited directly in the scene and optionally closed and filled with tiling texture.

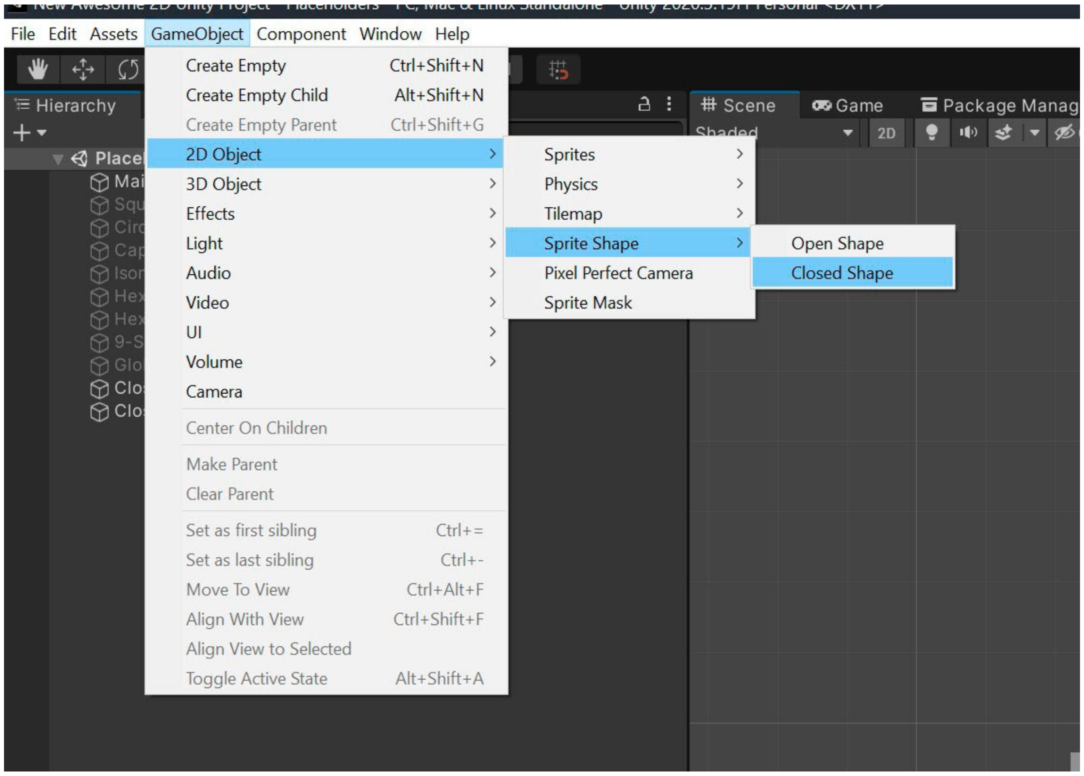
Sprite Shape is a handy prototyping feature in Unity because it makes it efficient to create and edit new sprite shapes.



Two Tangent Modes available for Sprite Shape: Linear and Continuous

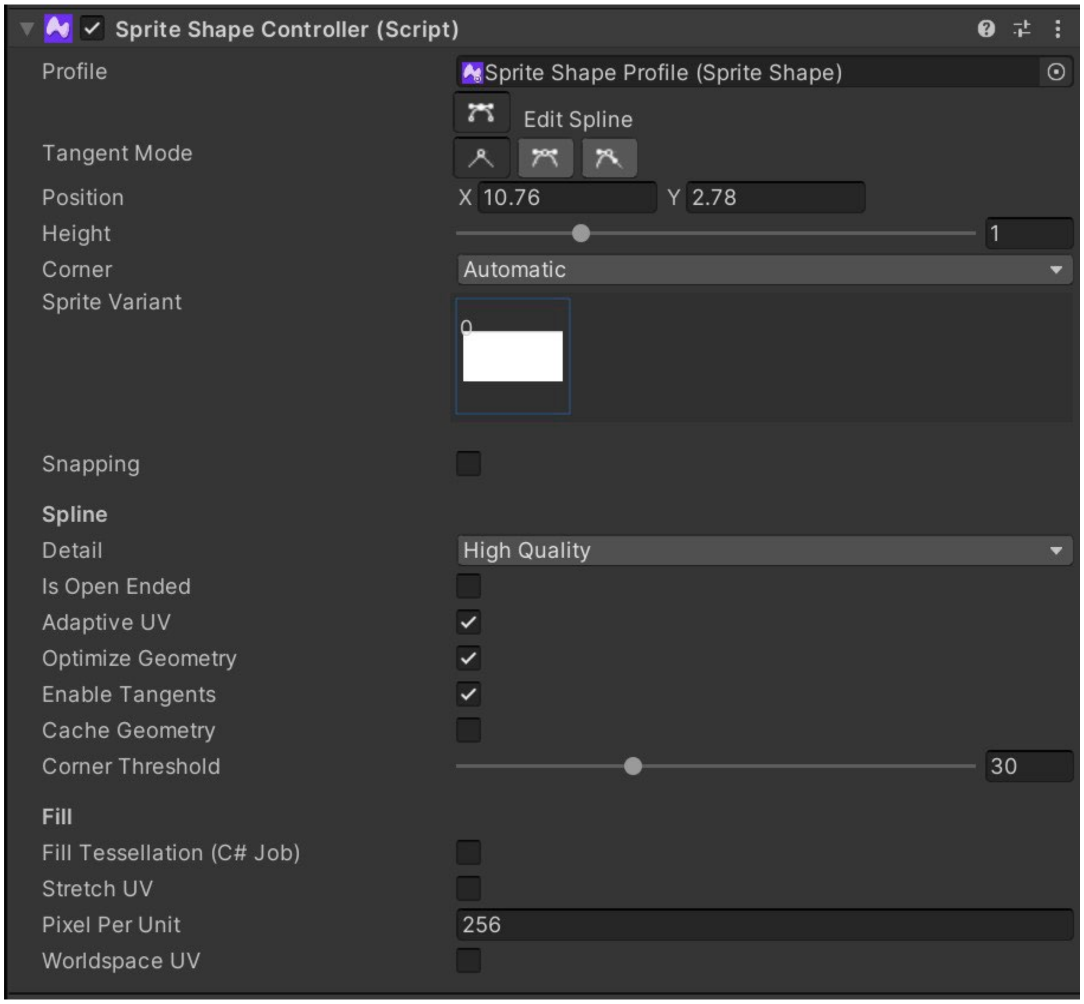
If you choose to create your 2D project with the 2D Template, then Sprite Shape is installed by default. Otherwise, you can install it using the Package Manager.

To make a new Sprite Shape, click on the GameObject menu, **2D Object > Sprite Shape,** and select either Open or Closed Shape.



Creating a Sprite Shape

To modify the Sprite Shape, select it, then click the Edit button in the Inspector.



Sprite Shape Controller (Script)

Profile	Sprite Shape Profile (Sprite Shape)
Tangent Mode	
Position	X 10.76 Y 2.78
Height	1
Corner	Automatic
Sprite Variant	0
Snapping	☐
Spline	
Detail	High Quality
Is Open Ended	☐
Adaptive UV	✓
Optimize Geometry	✓
Enable Tangents	✓
Cache Geometry	☐
Corner Threshold	30
Fill	
Fill Tessellation (C# Job)	☐
Stretch UV	☐
Pixel Per Unit	256
Worldspace UV	☐

Modifying a Sprite Shape in the Inspector window

Left-click on any part of the Sprite Shape to add or delete points with the delete key. By selecting a point, you can change its mode by selecting one of the three **Tangent Mode** buttons:

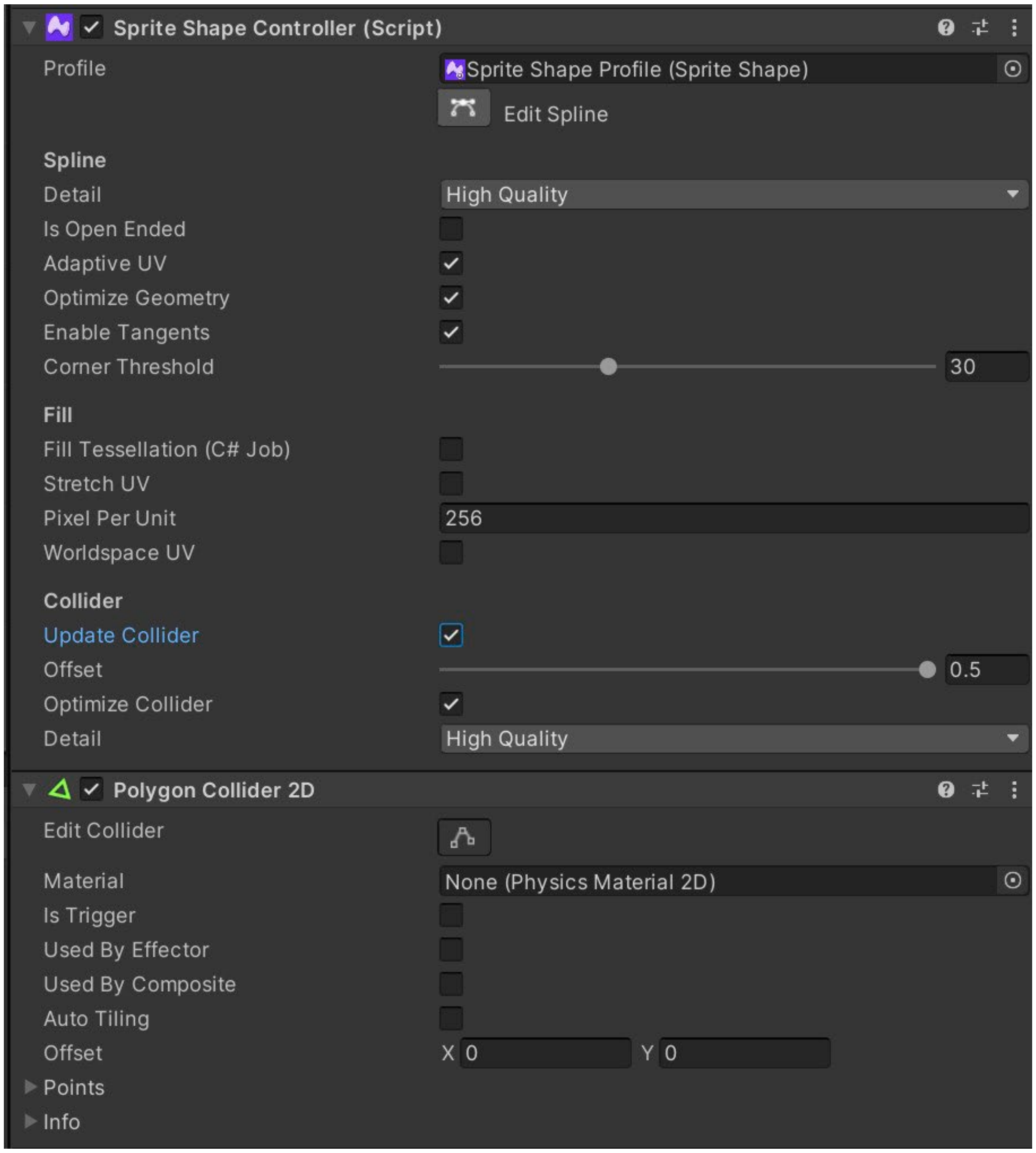
— **Linear**: Creates no curve, with straight lines on both sides of the points

— **Continuous**: Creates a curve around the point with handles that face opposite directions

— **Broken**: Creates a curve around the point with handles that can move independently

You can also select how the corners around the points look. The snapping tool is another useful option that snaps the points to the grid.

Add a **Polygon Collider 2D** component for physics. The default properties should be sufficient for prototyping purposes.



Sprite Shape Controller (Script)

Profile	Sprite Shape Profile (Sprite Shape)
	Edit Spline
Spline	
Detail	High Quality
Is Open Ended	☐
Adaptive UV	✓
Optimize Geometry	✓
Enable Tangents	✓
Corner Threshold	30
Fill	
Fill Tessellation (C# Job)	☐
Stretch UV	☐
Pixel Per Unit	256
Worldspace UV	☐
Collider	
Update Collider	✓
Offset	0.5
Optimize Collider	✓
Detail	High Quality

Polygon Collider 2D

Edit Collider	
Material	None (Physics Material 2D)
Is Trigger	☐
Used By Effector	☐
Used By Composite	☐
Auto Tiling	☐
Offset	X 0 Y 0
Points	
Info	

Add a Polygon Collider 2D component to a Sprite Shape GameObject.

By following these steps, you can create sprite shapes that can interact with one another and be modified through their Inspectors.

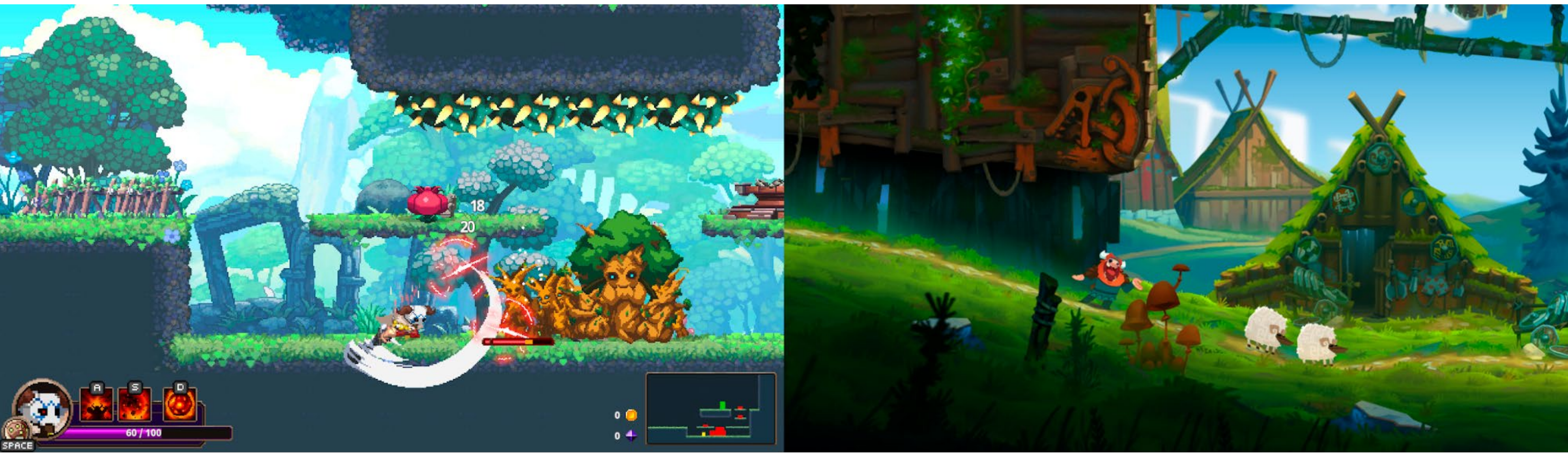
Sprite shape level design

Grid or tile-based design is well-suited to a retro 2D visual style, especially when you're using pixel art. It's also great for side-scrolling horizontal platformers, top-down RPGs, and non-organic shapes such as buildings and castles.

The grid-based design makes pathfinding and level creation more straightforward. Pathfinding is simpler when constrained to a grid. It makes it easier to maintain constant distances when you're designing levels. If your character's jump is 3 units high, for example, you can easily plan where to put platforms, and the player will be able to estimate if he can make the jump or not instantly.

When you want a more organic, less block-based style, use 2D Sprite Shape. You can make any shape you want using curves. Terrains, hills, grassy fields, and smooth surfaces are all suited to Sprite Shape, which lends a modern look to your scenes compared to the crisper pixel-art style.

Turn on the snapping tool to set any angle or curve easily and place elements precisely.



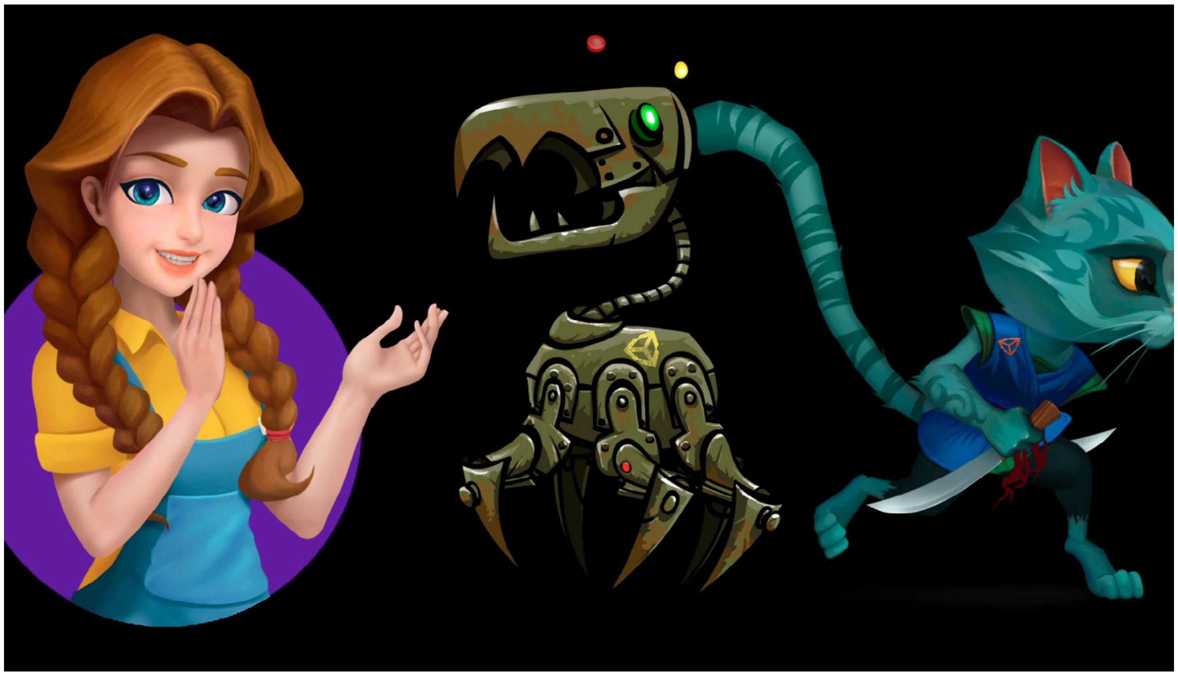
On the left, Skul: *The Hero Slayer* by SouthPAW Games is a tilemap-based game, while *Oddmar*, by Mobge LTD on the right, is a spline-based game.

Levels and elements created with sprite shapes are easy to edit later. Depending on your visual style, you can use both systems in your game.

You can use any Shader Graph sprite-based shader with Sprite Shape's fill or border material to give it your own flavor, or take it even further, modifying the shape generated. From Unity 6, the Custom Geometry feature is available in the Sprite Shape Controller, allowing you to use a custom script to generate or modify Sprite Shape geometry.

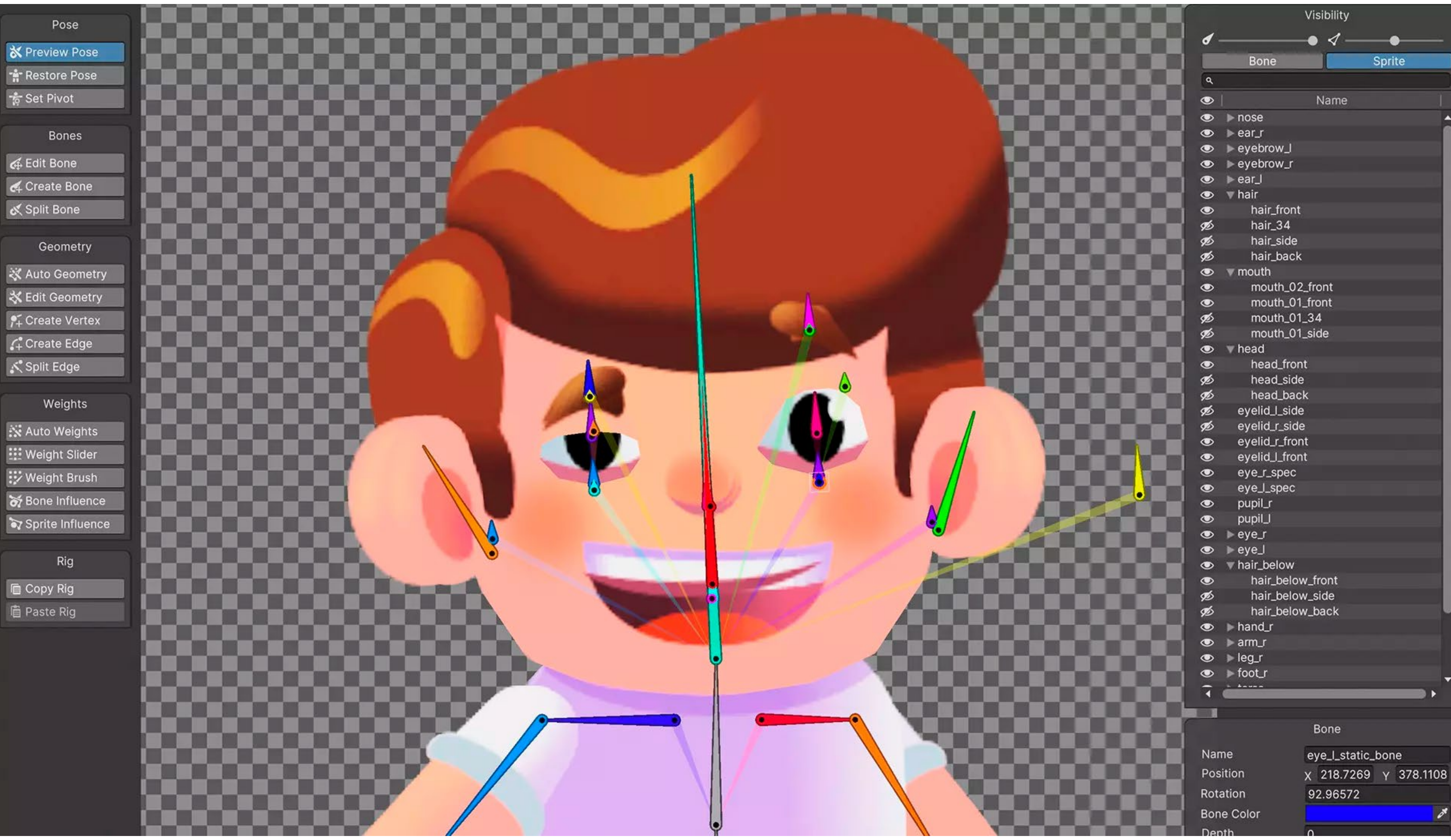
Both tilemaps and sprite shapes can be modified during runtime to create exciting new gameplay possibilities. For example, tilemaps can be destructive, and sprite shape can be morphed – the possibilities are many. Read more about the options in the Tilemap API and SpriteShape API docs.

2D animation: Skeletal animation


Use Unity's 2D animation features to animate characters from many different genres and styles. Get the free 2D Animation Samples package from the Unity Asset Store to learn more.

2D animation can be the most time-consuming and challenging part of developing art for a game. Creating animated characters requires knowledge of animation timing, momentum, kinematics, and more. Polishing every frame can take a long time, and storing and displaying these frames requires a lot of memory. Changing the timing or any part of a character can require every frame to be redrawn.

Historically, 3D animation has been more straightforward than 2D. You make a 3D model, rig it by adding a skeleton, set up bone weights, and animate it by setting keyframes that the software interpolates between. You make adjustments by editing the keyframes.



Name: eye_l_static_bone
Position: X 218.7269 Y 378.1108
Rotation: 92.96572
Bone Color
Depth: 0

Testing the rigging of the expressions of the Happy Harvest character in the Sprite editor

Unity provides an efficient 2D animation toolset that in a number of ways mimics the 3D animation process, for example, by enabling you to import your character artwork directly from Photoshop into Unity.

Design, import, and rig

The most vital element in a game is the player's character, so take enough time to think through the character's design. There are a few key aspects to consider while planning.

Perspective

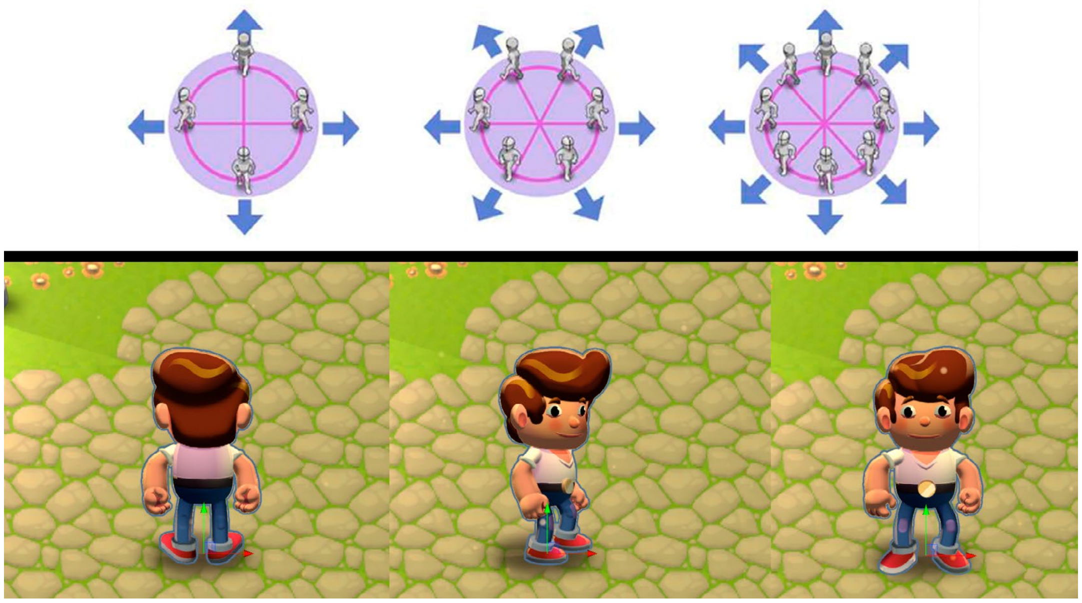
The perspective you choose will impact how the game characters look and are animated. In most game views, characters can be drawn in profile. A slight rotation into a ¾ view allows more facial features to be shown.

When using an isometric or ¾ top-down perspective, you can create a similar view, but with a camera tilted a bit from the top to show facial detail. In this perspective, the character will be drawn facing multiple directions. Depending on your desired outcome and budget, you can choose from one of three commonly-used options:

— 4 directions

— 6 directions

— 8 directions



From Happy Harvest: Different possibilities for character rotation when creating a 2D top-down or isometric game

No matter which you choose, you will have to decide whether right- or left-facing animations will have to be flipped to face the opposite direction. In this case, you'll need to draw respectively:

— 3 directions: right, up, down

— 3 directions: right, right-up, right-down

— 5 directions: right, right-up, up, right-down, down

If the animations are flipped, the character's hands will flip to match. If the character faces right and holds a sword in their right hand and a shield in their left, when the character is flipped to be left-facing, both weapons will have switched hands. Decide whether to accept this trade-off or spend more time making all of the animations face in the right direction.

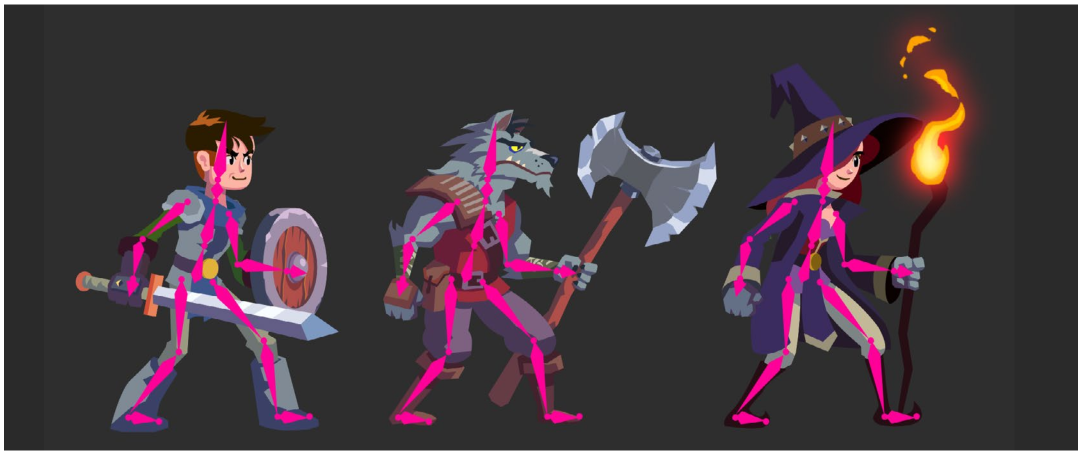
If you're designing a traditional side-scrolling beat-'em-up game, you can simply use one facing direction. Characters moving up and down will look like they're sidestepping.

Ultimately, it's up to you whether or not your character looks good in the game's environment after testing. Your vision is more important than having a realistic view angle and perspective.

Reskinning characters

A benefit with the 2D Animation package is that multiple characters can share the same skeleton and animations. Once you design, rig, and animate one base character, you can then simply swap its skin. A feature called Skeleton sharing streamlines this process.

Save time by planning ahead. Draw concepts of all the characters that will share the same skeleton, then check whether the skeleton will fit all of them. Do this by drawing a skeleton on a separate layer and overlaying it onto every character. Bear in mind that the layer count must be the same for all characters.


Overlaying skeleton drawing in an image-editing application

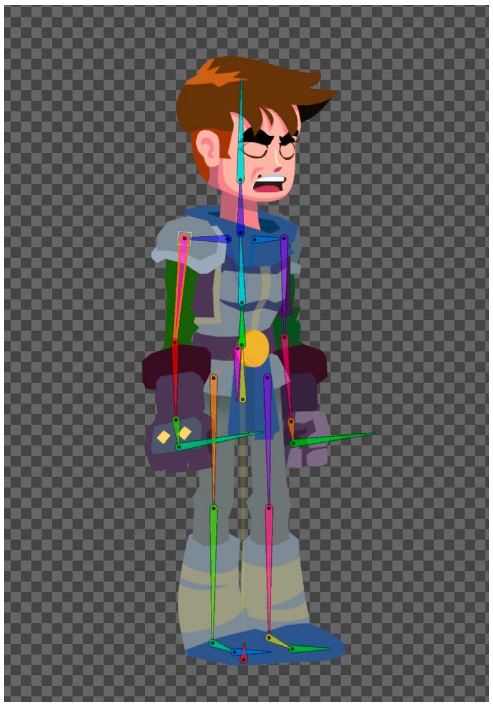
General 2D animation rules

Here are some good rules to go by when designing a character that will be rigged and animated later:

— Draw the character in a neutral position with arms and legs straight. If the parts of the body are drawn bent, it can cause issues when you're animating.

— Make the resolution a little higher than your game's PPU suggests. While resolution can look good at rest, rotating and stretching images can cause pixelation.

— A tip that we share later in the 2D lights chapter is to avoid painting the light and shadow onto your sprite if you plan to use normal maps. Instead, paint some nondirectional shadows. This technique is called ambient occlusion; your sprite will look better, but avoid using any directional light like sunlight.


An example of a neutral character pose: Legs and arms can be any position, but they should be unbent; straining bent limbs will cause pixel stretching

— Body part layers swapped using the Sprite Swap feature should be grouped accordingly. For example, all layers with mouth positions should be placed in a group called “mouth” in the image-editing app.

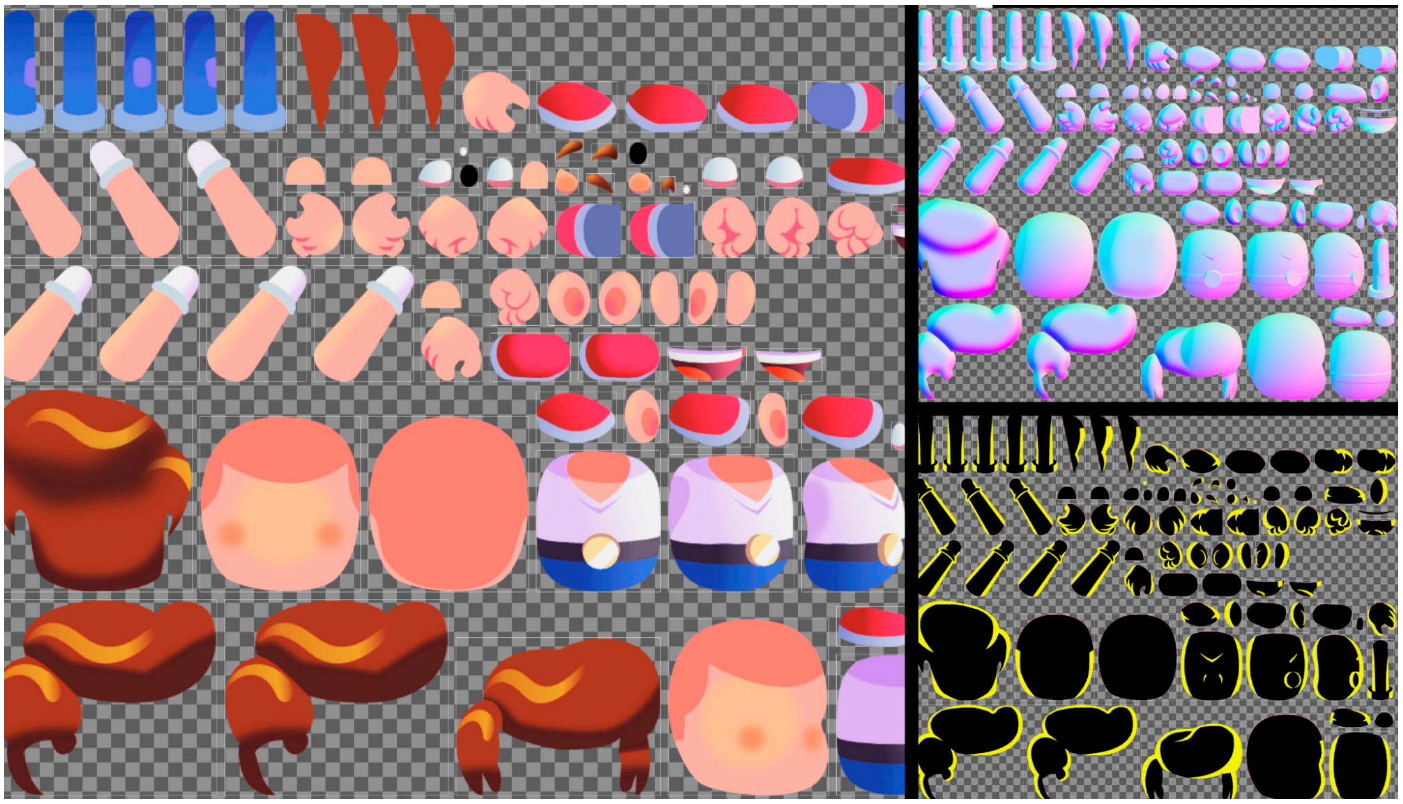
Now that you know what to consider during designing, start up your graphic app of choice and let’s start painting.

Import a character into Unity

The 2D Animation package allows you to import your character artwork directly from Photoshop into Unity. The PSD Importer package enables the functionality of handling layered PSD and PSB files for this purpose. This will import all of the character’s layers as sprites and place them exactly as they were painted in the app.

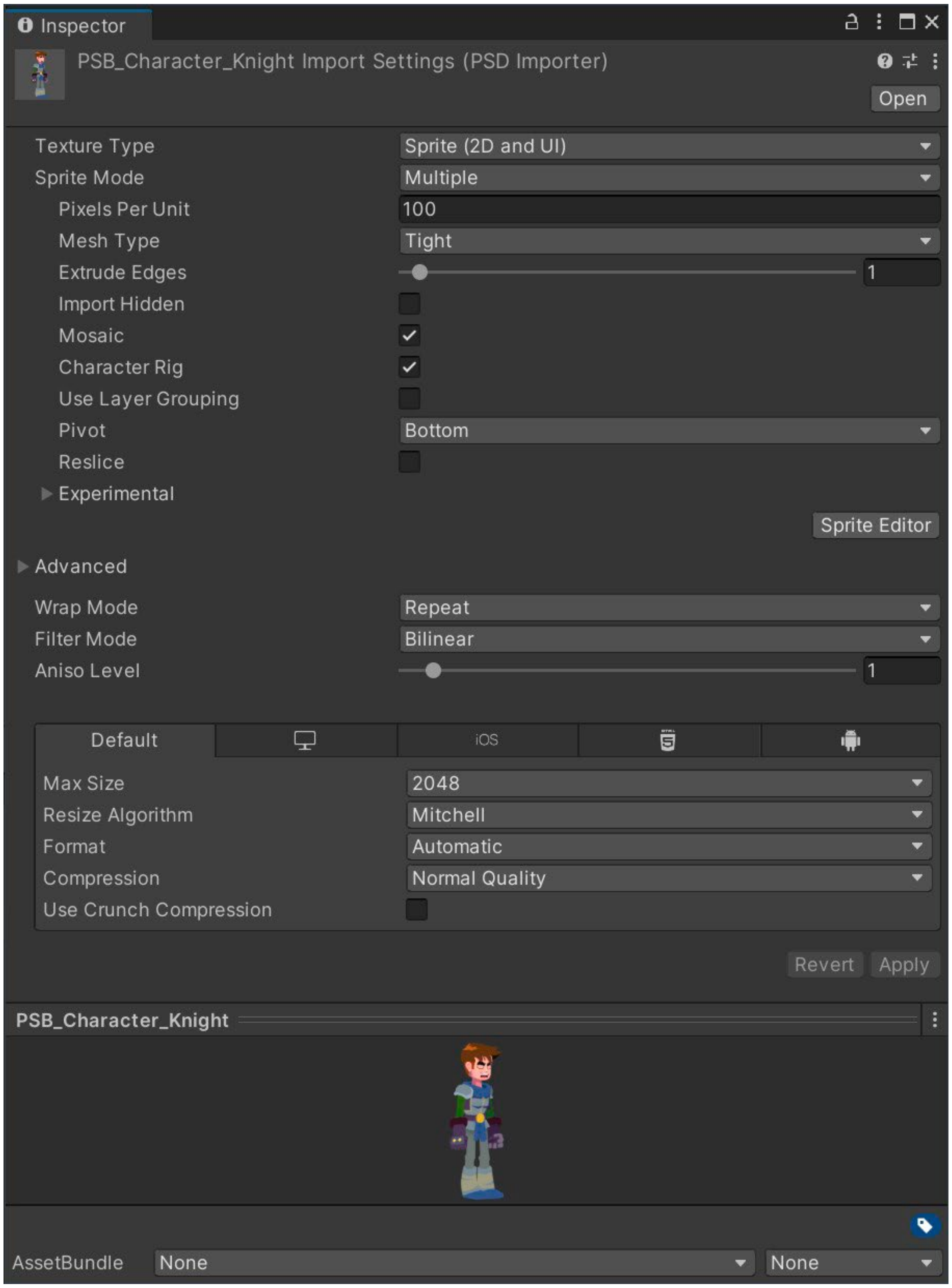
If you’re using the 2D project template in Unity, the importer should be already installed. If not, install it using Package Manager.

The exporting process simply requires files to be saved as PSD or PSB format. PSD being common in Photoshop and other creation tools.



The sprite sheet of the importer PSD with the main character parts in layers: On the right you can see the PSD files added as Secondary Textures for lighting purposes

Importing into Unity is the same as with other assets. Save the file to the Assets folder, or drag it into the Project window.



PSB_Character_Knight Import Settings (PSD Importer)

Texture Type	Sprite (2D and UI)
Sprite Mode	Multiple
Pixels Per Unit	100
Mesh Type	Tight
Extrude Edges	1
Import Hidden	☐
Mosaic	✓
Character Rig	✓
Use Layer Grouping	☐
Pivot	Bottom
Reslice	☐
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1
Max Size	2048
Resize Algorithm	Mitchell
Format	Automatic
Compression	Normal Quality
Use Crunch Compression	☐
AssetBundle	None / None

The Inspector for a character

When an imported PSD or PSB file is selected, it will show options similar to the sprite import settings but with additional ones for 2D animation and rigging. The key options to set are:

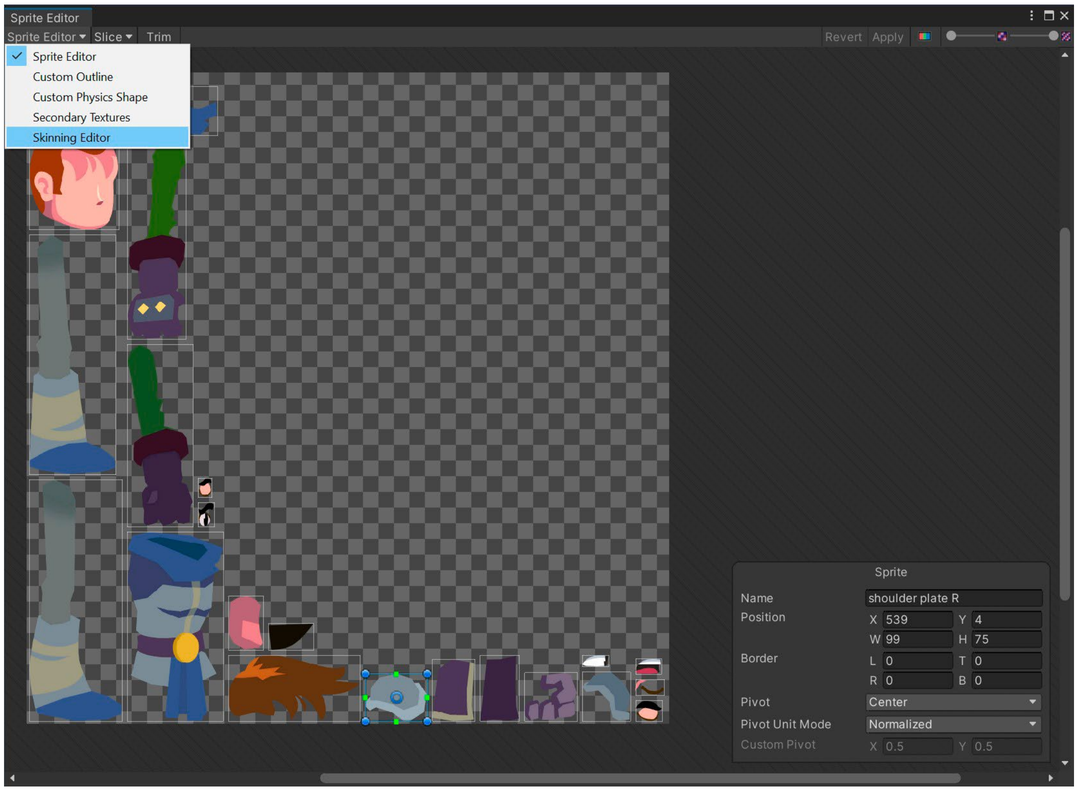
- **Import Hidden Layers:** This option imports all layers from a PSD or PSB file, including hidden layers.
- **Individual sprites (Mosaic):** This setting is only available if the Texture Type is set to Multiple. It makes sprites from imported layers and puts them onto a texture atlas. Enable this option if you want to rig a character.
- **Use as Rig:** This option generates a character prefab with the same layers hierarchy and position as in the PSD or PSB file. This must be on.
- **Use Layer Group:** This adds layer grouping from the PSD or PSB file. Turn it on to group parts of a character, such as sprite swapped parts.

In the **Layer Management** section you can see the hierarchy of layers and manually add or remove layers to be imported. After setting the options, click Apply. This finalizes the character prefab, which can now be dragged into a scene.

Get the 2D Animation Sample package from the Unity Asset Store.

Rigging in the Sprite Editor

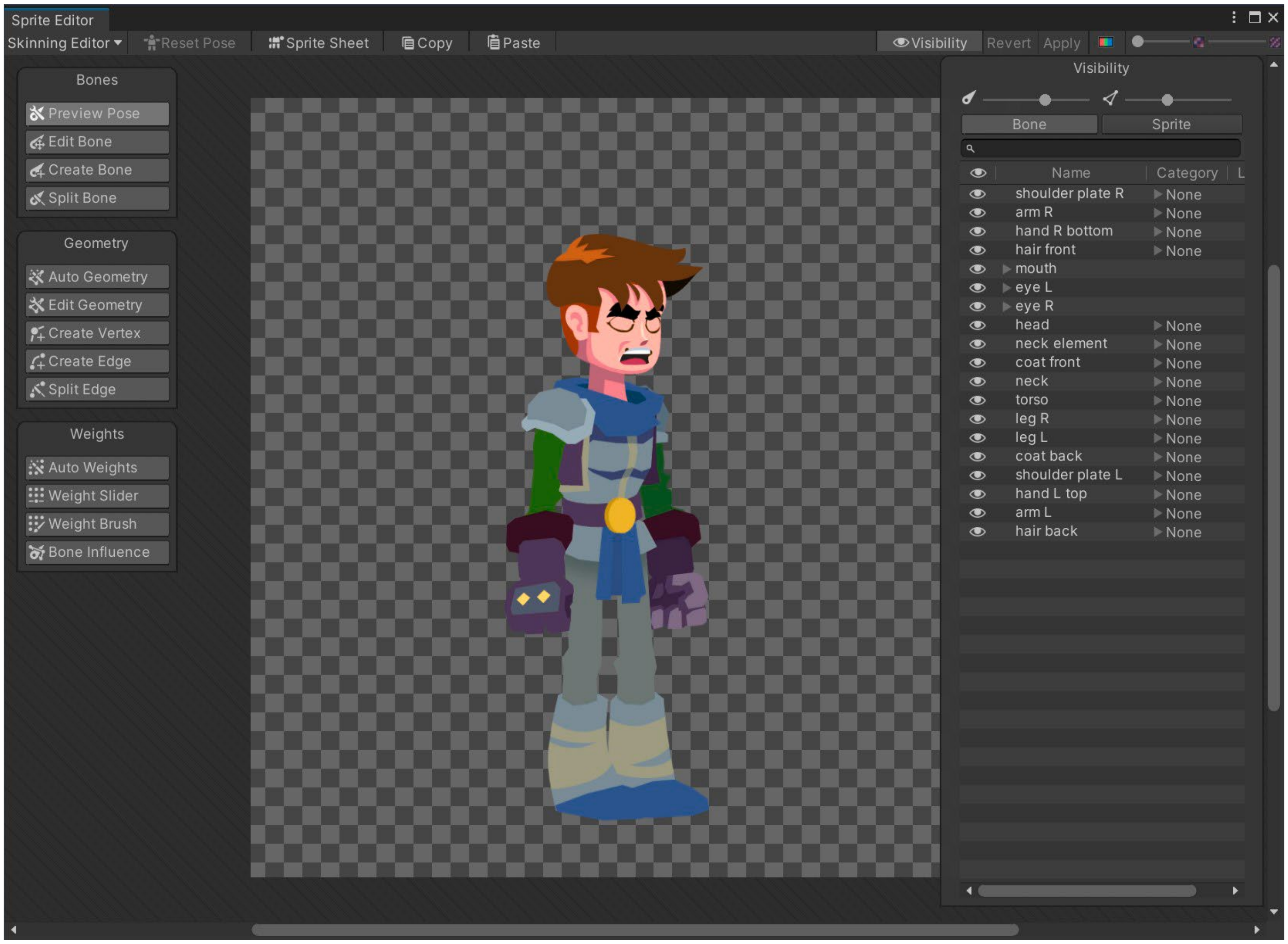
To start rigging a character, click the Sprite Editor button in the Inspector to access the Sprite Import Settings.



Name: shoulder plate R
Position: X 539, Y 4, W 99, H 75
Border: L 0, T 0, R 0, B 0
Pivot: Center
Pivot Unit Mode: Normalized
Custom Pivot: X 0.5, Y 0.5

How to choose tools in Sprite Editor's window

Select Skinning Editor from the drop-down menu in the top-left corner of the Sprite Editor window.



Name	Category
shoulder plate R	None
arm R	None
hand R bottom	None
hair front	None
mouth	
eye L	
eye R	
head	None
neck element	None
coat front	None
neck	None
torso	None
leg R	None
leg L	None
coat back	None
shoulder plate L	None
hand L top	None
arm L	None
hair back	None

The Skinning Editor with options for animating a character

In this window, you can:

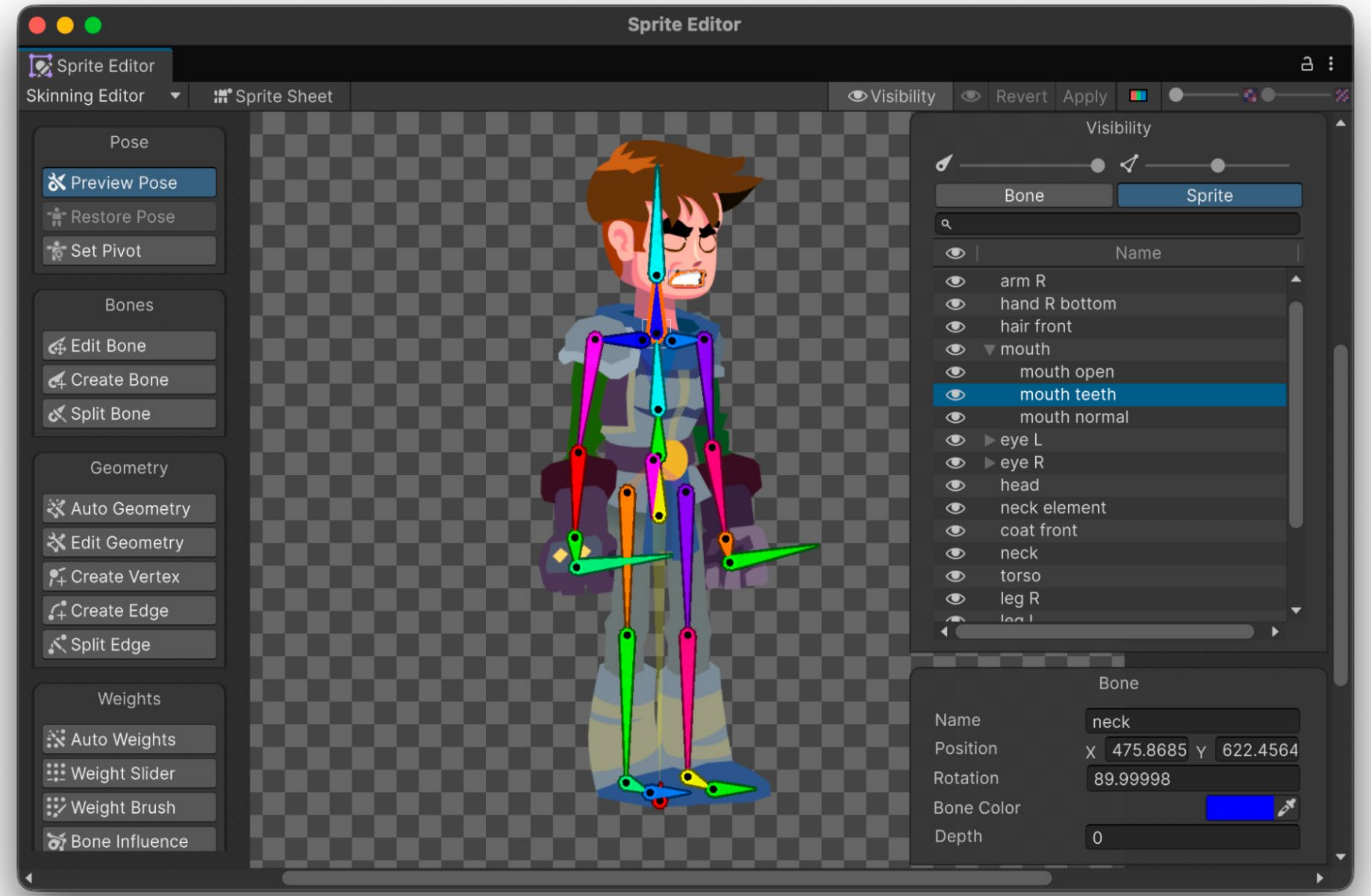
- Create and edit bones.
- Create and edit sprite geometry.
- Edit bone influence on sprites.
- Create categories and labels for use in skins and sprite swaps.

Creating the skeleton

Start by making the character's skeleton. Choose the **Create Bone** button and left-click on the main window area. The first click will create a bone center, and the second will mark the bone tip location. This tool chains and nests bones inside one another.

If you want to change the bone placement, press the right mouse button and left-click where you want the new bone to be located. To control which existing bone will be the parent of the bones to be created, select the existing bone with the left mouse button, then create the new bone.

Use the **Edit Bone** button to refine bones. The **Split Bone** button allows you to split bones in two. This is a good option for making limbs. If you make one leg bone and click where the knee should be, the bone will then split into thigh and calf.



Bone

Name	neck
Position	X 475.8685 Y 622.4564
Rotation	89.99998
Bone Color	🟦
Depth	0

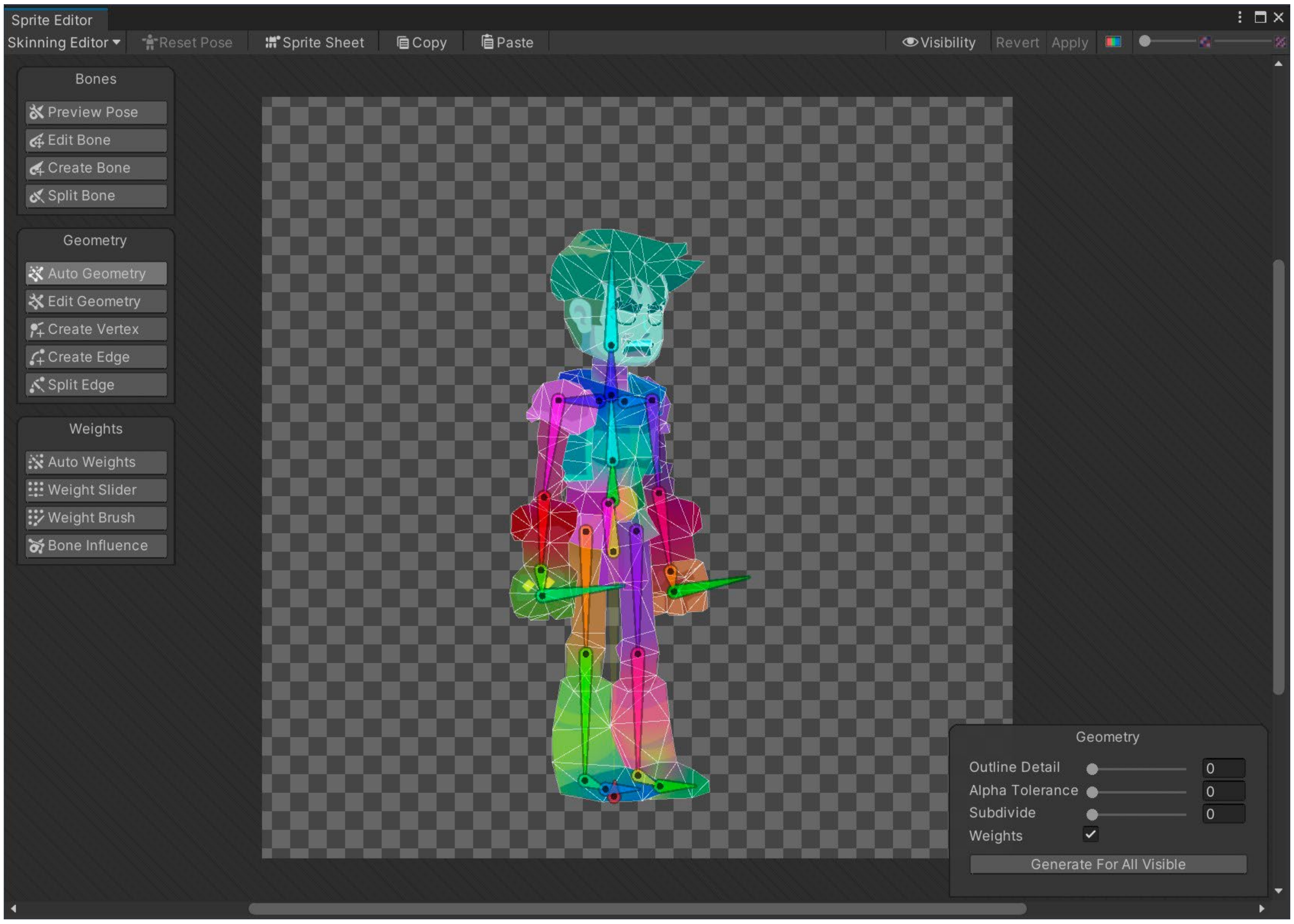
The Bone tab opened to organize parent and children bones

You can also reparent and rename bones in the bone list view. To open this view, click the Visibility button on the right of the top bar, then select the Bone tab. To change the bone's parent, simply drag it in the list view. You can rename bones by clicking on the active bone's name. Giving bones a name will help you find them later. To verify that the hierarchy is correct, select the Preview Pose button and test some poses. To reset bones positions, press the Reset Pose button on the toolbar.

Sprite geometry

In order to assign sprites to bones, they'll need to have geometry created. Start by pressing the Auto Geometry button. A small pop-up window will open, allowing you to define how the geometry will be created.

It's a good idea to set all sliders to 0 to keep the geometry as simple as possible. Enable the Weights option to bind bones to the sprites automatically. Clicking the Generate For All Visible button will create and set bone weights for all sprites. To do this for an individual sprite, simply double-click on the sprite. This is useful for tweaking the geometry of certain sprites.



Geometry

Outline Detail 0
Alpha Tolerance 0
Subdivide 0
Weights ✓

Auto-generating sprites' geometry

To go beyond this generated geometry and have full control on a number of vertices and how the geometry bends, the mesh will need to be edited manually. Use the following tools:

— **Create Vertex:** Create new vertices (or points), or move or delete those points.

— **Create Edge:** Creates edges, either between two existing points or by creating a new point using the left mouse button; the right mouse button deselects the currently selected point and lets you make an edge between other vertices.

— **Edit Geometry:** Move vertices.

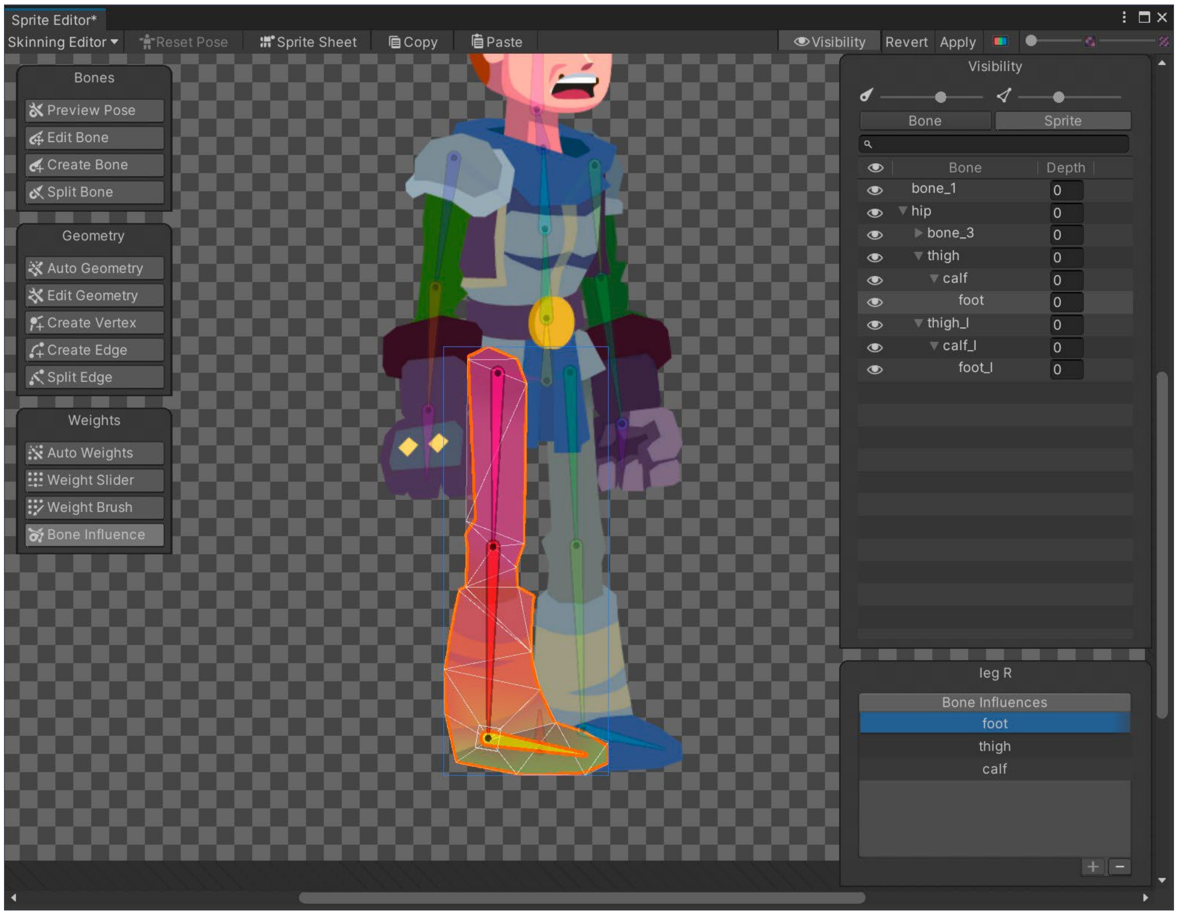
— **Split Edge:** Split an edge by creating a vertex in the middle.

Use as few vertices as possible. Fewer vertices will help you save on performance, since each vertex position needs to be calculated based on the bones' rotation. Fewer vertices also make for better-looking mesh bending since it's easier to set weights for fewer points. Additionally, each target platform will have an ideal game vertex count, so it's always a good idea to optimize geometry.

Weights

Once the geometry is nice and clean for all of your sprites, it's time to set up weights. Weights define bone influence on each vertex from 0 to 1, where 0 means that a bone has no influence over the vertex, and 1 means that the vertex will move like it's glued to the bone. Good weights on the mesh can allow for great-looking bending. Setting weights incorrectly can break the game's illusion and distort your sprites.

To start setting up weights, you need to define which bones will affect a certain sprite. Click the Bone Influence button, and select a sprite.

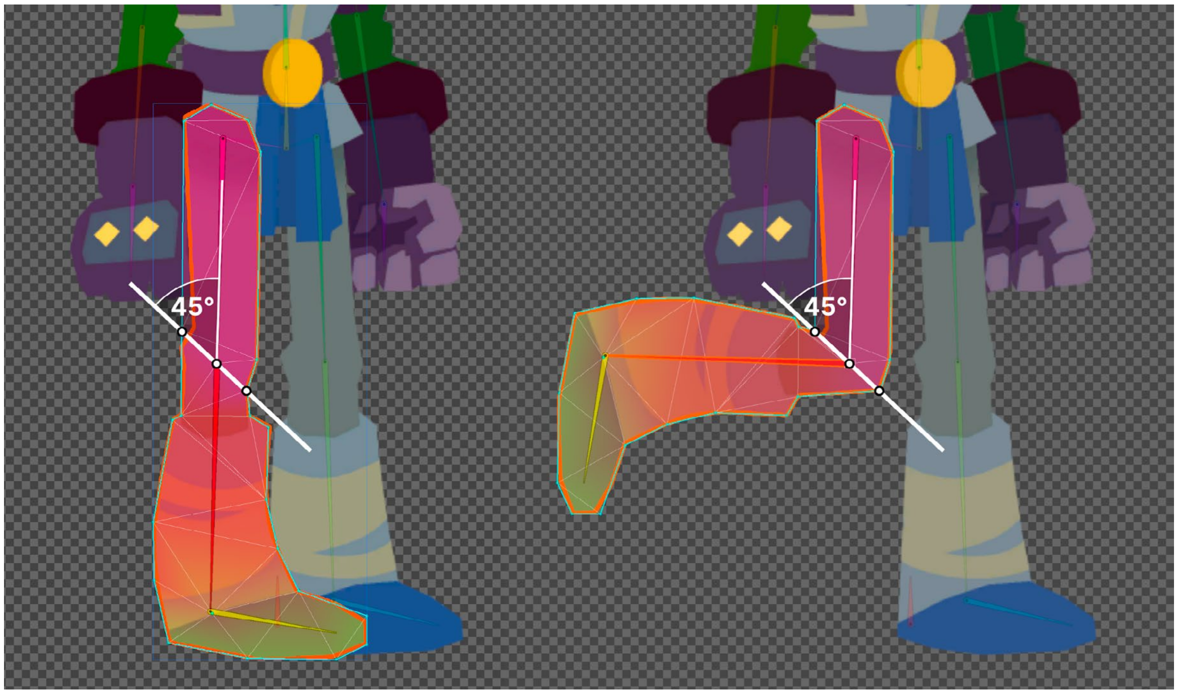


Bone	Depth
bone_1	0
hip	0
bone_3	0
thigh	0
calf	0
foot	0
thigh_l	0
calf_l	0
foot_l	0

Setting bone influence on a sprite

In the small pop-up window, you can set which bones influence the selected sprite. Set only the relevant bones that need to affect the sprite, and remove the rest.

From here, set up weight by using the **Weight Brush** and **Weight Slider**. The Weight Brush quickly adds the influence of the selected bone to the vertices by painting on them with the mouse. The Weight Slider is more precise, allowing you to select one or more points, then determine the exact influence of each bone with a slider. The brush is useful for quick weight setting, and the slider is useful for fine-tuning the areas where bones bend, like elbows and knees.



How to place vertices for the best looking limb bending

Tip: When you're working with elbows and knees, get the best-looking results by aligning the inner and outer vertices to a line which runs through the middle of the bending point where the two bones meet. This line should cross the bones at a 45-degree angle. These vertices should be influenced only by the upper bone. However, every character will be different, so feel free to fiddle with the weights and customize for the best results.

To speed things along, open the *Dragon Crashers* project, and find the following character prefabs in the section for Prefabs and Prefab Variants:

- Prefab_Character_Base
  - *PV_Character_Witch* (uses a different Sprite Library for auto rebinding)
  - *PV_Character_Knight*
  - *PV_Character_Wolfman*
  - *PV_Character_Skeleton*

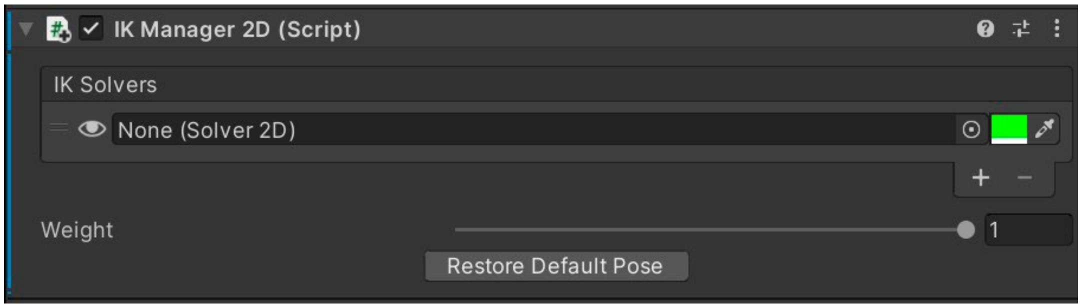
These prefabs are great examples of shareable animations between characters of the same structure.

2D inverse kinematics

You probably don't think about it every day, but the human body's movement is very complex. If you want to grab a glass of water off of a table, your hand must move to the point in space that the glass occupies. You do all this without even thinking about rotating your arm and forearm, as your brain processes these calculations in the background.

To animate the same movement in a game, the rotation of both arm and forearm need to be animated at the same time. It's a challenging task to match both rotations while making a believable hand movement. The 2D Inverse Kinematics (IK) tool, which is a part of the 2D Animation package, calculates the rotations and allows a chain of bones to move them to target positions.

To begin, you'll first need to add an IK Manager 2D on the object at the top of the hierarchy. This component will be responsible for managing all the IK Solvers on the character.



IK Manager 2D component

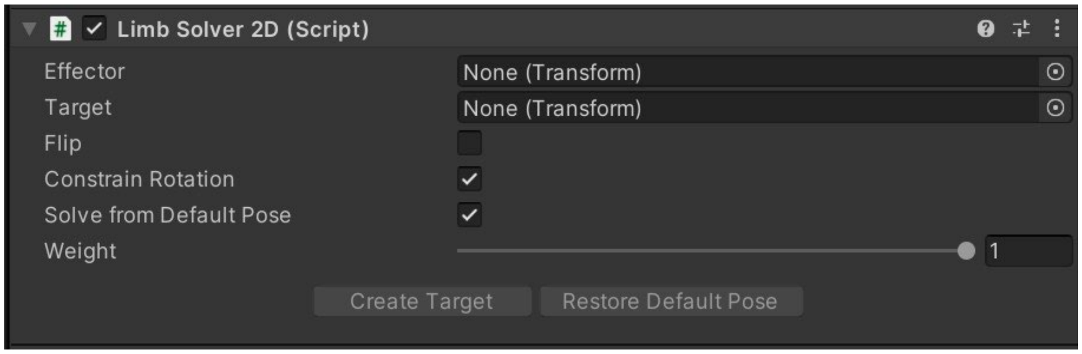
Clicking the **+** button will add a new solver. The solver will calculate bones' rotation to match the target transform. There are three solvers available:

— **Limb:** This is the standard solver used for legs and arms, which can solve up to two bones and the Effector.

— **Chain (CCD) – Cyclic Coordinate Descent:** This solver gradually becomes more accurate the more times the algorithm is run and is suitable for a longer chain of bones.

— **Chain (FABRIK) – Forward And Backward Reaching Inverse Kinematics:** Like Chain (CCD), this solution becomes more accurate the more times its algorithm is run. It adapts quickly if the bones are manipulated in real-time to different positions.

In humanoid characters, the Limb Solver is the best choice because it is fastest and optimized for two-bone limbs. Start by adding Limb Solver to IK Manager 2D's list, which will create a new GameObject that will include the Limb Solver 2D component. Rename this GameObject to something descriptive like "Leg R LimbSolver2D" or "Arm L LimbSolver2D."



Limb Solver 2D (Script)

Effector	None (Transform)
Target	None (Transform)
Flip	☐
Constrain Rotation	✓
Solve from Default Pose	✓
Weight	1

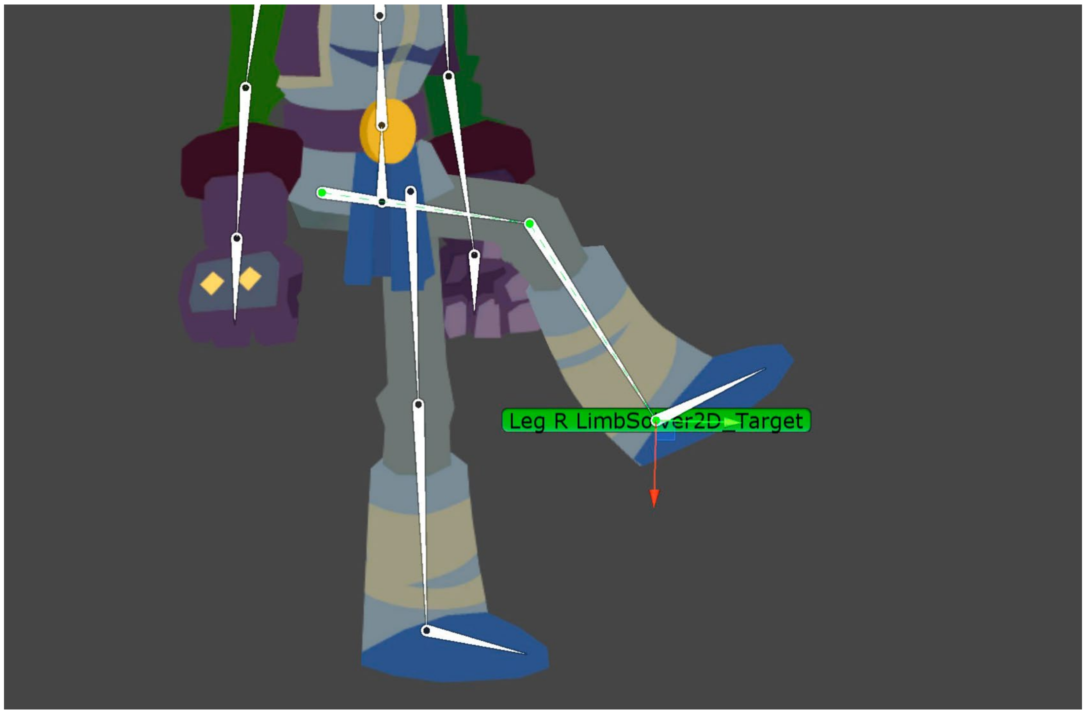
Limb Solver 2D component

In order to work, the solver needs two GameObject Transforms: the Effector and Target. The Effector is placed inside the last bone in a chain, for example, the tip bone of a finger, and from there, it will try to reach the Target's position.

First, create the Effector. When creating an IK for a leg, there are two bones: the thigh and calf (or shin). Place the Effector as a child of the calf bone by selecting it and creating a new Empty GameObject. Rename the GameObject as Leg Effector, and move the Effector to the tip of the bone.

Add the newly created Effector to the Effector field on the Limb Solver 2D, and press the Create Target button. A target GameObject will be created inside of the Limb Solver object.

Now, when the Target object moves the leg will follow.



2D IK in action: When you move the Target GameObject, thigh and calf rotations will be calculated automatically to match the Target's position.

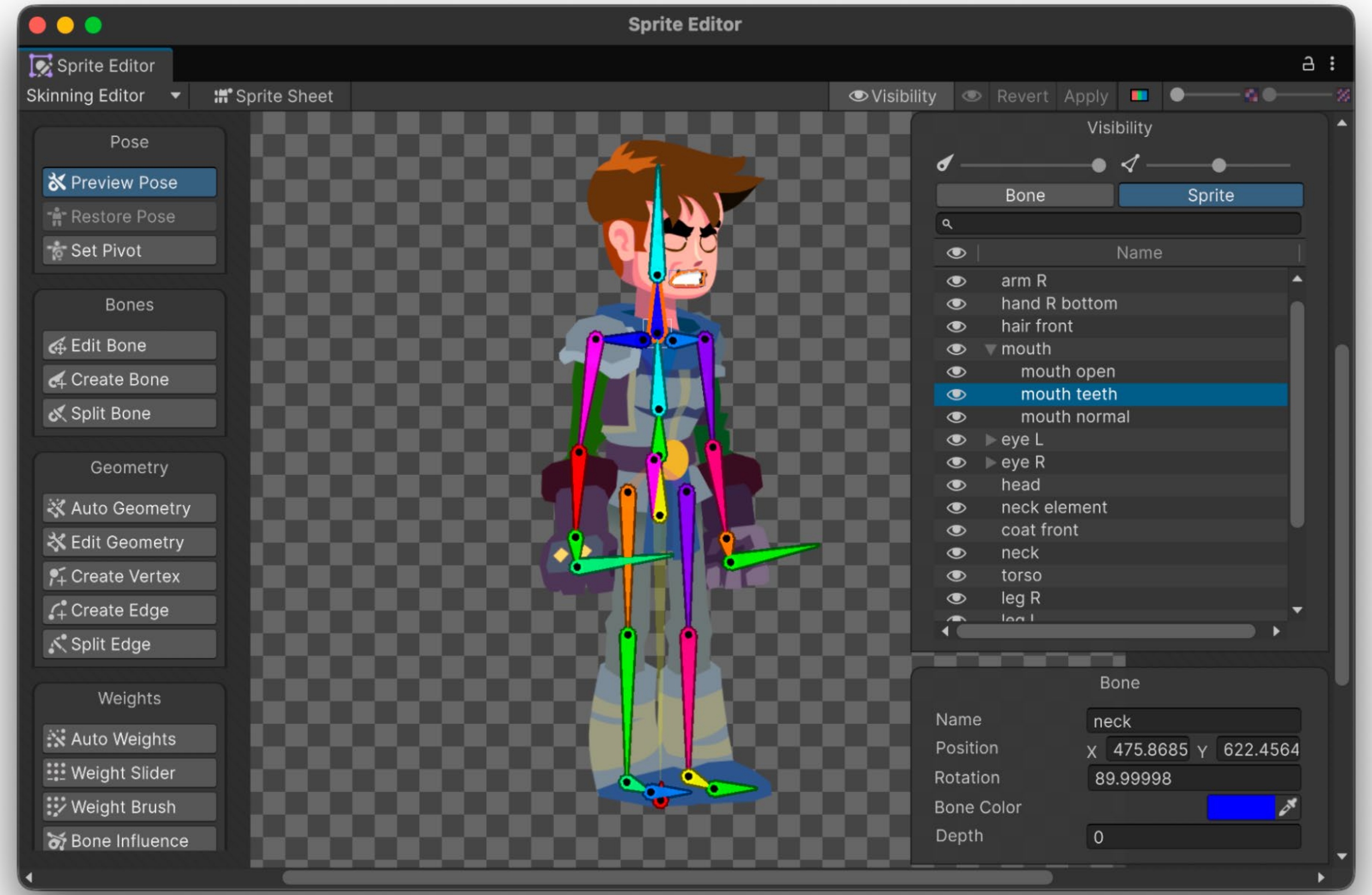
Repeat the whole process of adding IKs with the other limbs. IKs can also be added on non-limb bones, for example, on the head and neck, which allows the character to look around at things.

Once you've mastered this, there are more advanced use cases. For example, you can set up IKs to make your character aim a gun or to create a procedural walking animation.

Sprite Swap and skins

Not everything can be animated by rotating a bone. Sometimes you need another facial expression or hand pose. In this case, you can exchange one sprite with another by using Sprite Swap.

Previously Sprite Swap worked by assigning a category and label to each of the character's sprites from the Skinning Editor. In Unity 6 categories and labels are now part of sprite libraries, as explained in the Sprite Swap section on page 47.

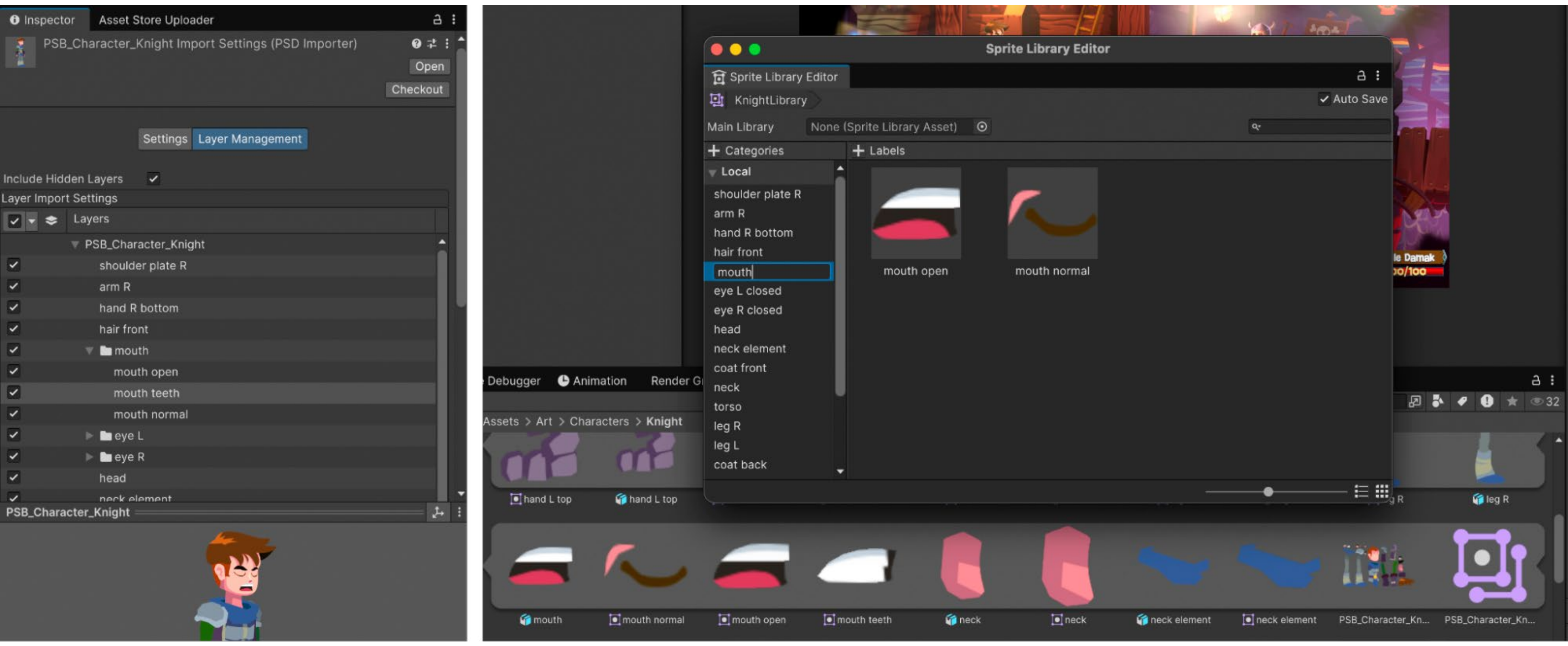


arm R
hand R bottom
hair front
mouth
mouth open
mouth teeth
mouth normal
eye L
eye R
head
neck element
coat front
neck
torso
leg R
leg L

Name	neck
Position	X 475.8685 Y 622.4564
Rotation	89.99998
Bone Color	🟦
Depth	0

Enable the Visibility setting in the Skinning Editor to see the sprites and bones of a character.

Let’s look at the **mouth** sprites of the knight character from *Dragon Crashers*. You can see that the variations are children of the mouth category, which matches the grouping in the original PSD or PSB file.



In the image on the left, you can see the layers in the PSD file in the Inspector with the 2D PSD Importer. On the right, a Sprite Library is created for the sprites that make up the knight character.

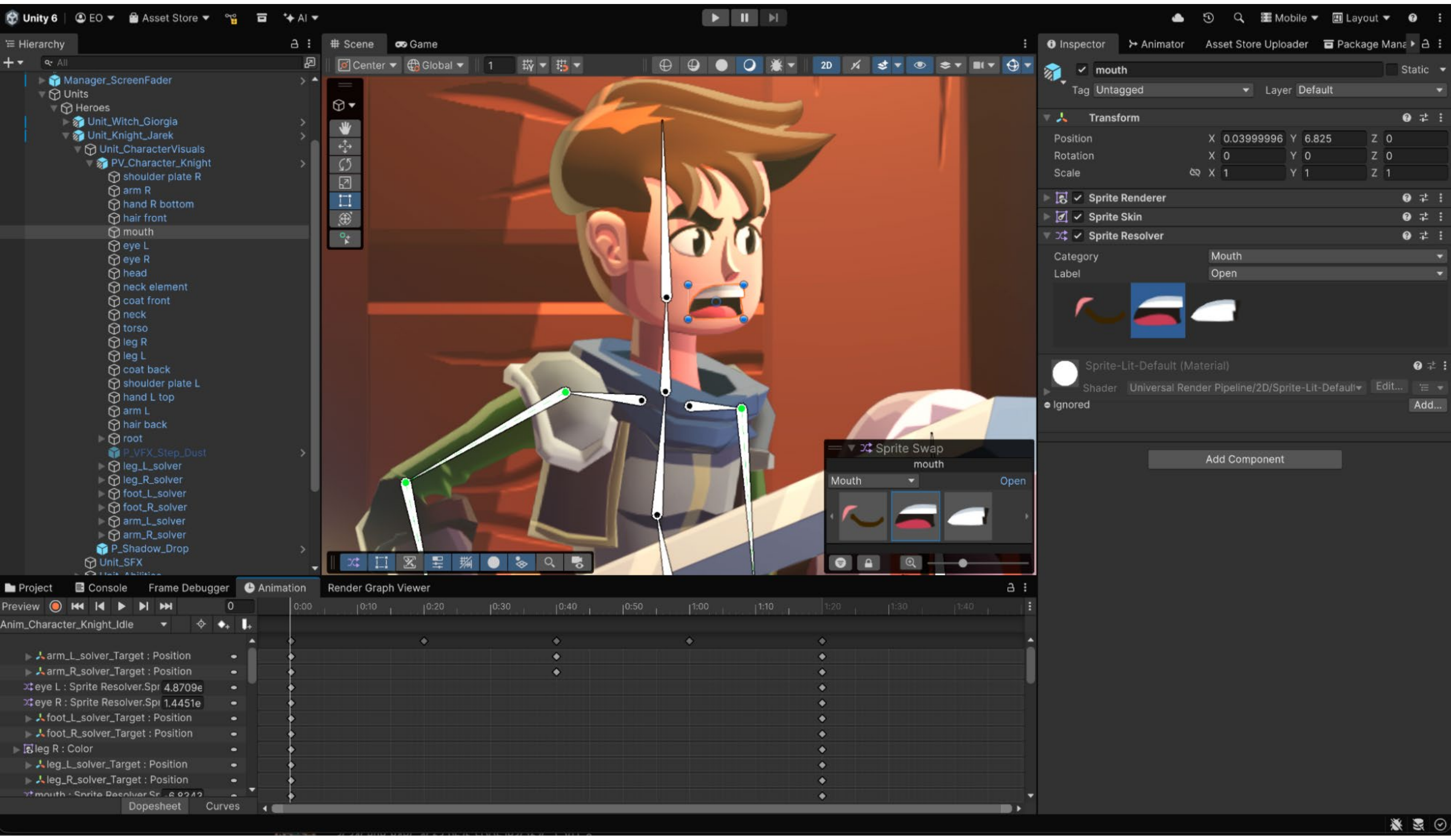
You can create a new Sprite Library from the create menu. Name it something related to the character. Drag and drop the PSD file into the category column of the Sprite Library window.

The categories are automatically created based on the visible sprites in the PSD file.

You probably want to rename the category names; in the example of the knight character, the category for mouth variations is named mouth.

Not all mouth variations were added, as they were hidden layers in the PSD file, but we can add the mouth variations to the mouth category by dragging and dropping them into the category. Each mouth variation will then have a label or name.

Sprites that were meant to be swapped have been added to the Sprite Resolver, and you can now implement sprite swap by clicking on them. For example while animating you can conveniently change the sprite with the resolver.

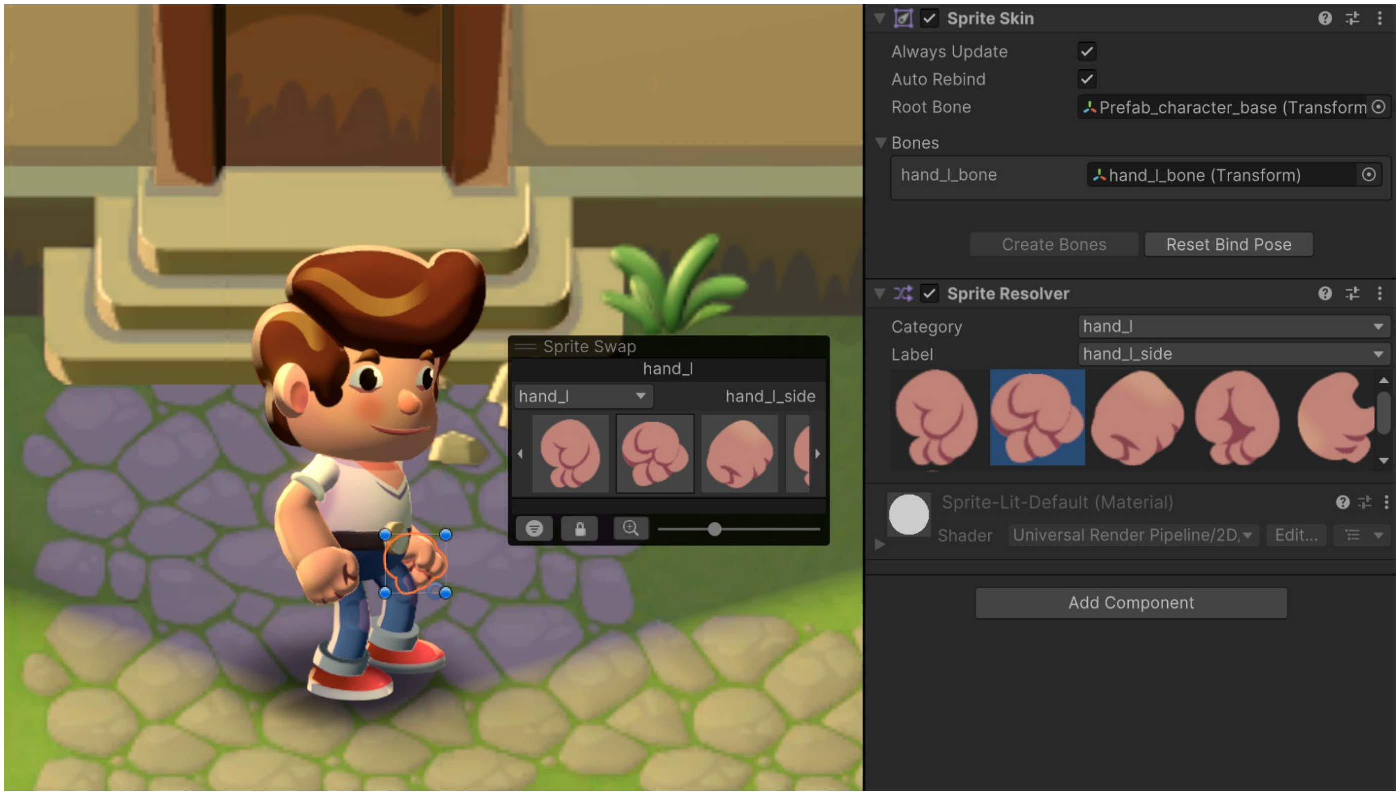


Swapping mouth sprites on the knight character from Dragon *Crashers*

Now, select the mouth in the character; if no category is selected in the Sprite Resolver, select one, and you'll see the variations to easily swap. Sprite swapping can be used from the Overlay menu as well.

Use Sprite Swap to conveniently animate facial expressions, eyes and mouths, make lip-synced animations, change hand gestures, and more. Sprite Swap also allows character equipment like hats and armor to be swapped.

Sprite Swap overlay



2D animated characters can benefit from sprite libraries with categories and labels to easily swap sprites from the Inspector or scene Overlay.

A new **Sprite Swap overlay** is now available in the Scene View in Unity 6. This overlay reveals all the relevant labels that a selected sprite can swap to.

It also scales up to all the swappable parts of a hierarchy. When any part of a multi-part character is selected, all the child objects with swappable sprites are displayed together in the overlay. Use this in combination with the animation window to make your sprite swap animation workflow more efficient.

Skins

Sprite Library assets let you create skins for your character. A skin changes the look of the character while retaining its animations, which is a great timesaver. One base character can have all the scripts, and any change will be also applied to other characters.

Here is the workflow for setting up skin support on characters in the *Dragon Crashers* and *Happy Harvest* samples:

1. Finish a base character: Rig it, set up IKs and a Sprite Library. Make a prefab from the character. You can animate the character at any point.

2. Go to the Skinning Editor of the base character, and click the **Copy** button on the toolbar. This will copy bones and meshes with weights.

3. Create a Prefab Variant of this character – this will be the new character with a new look.

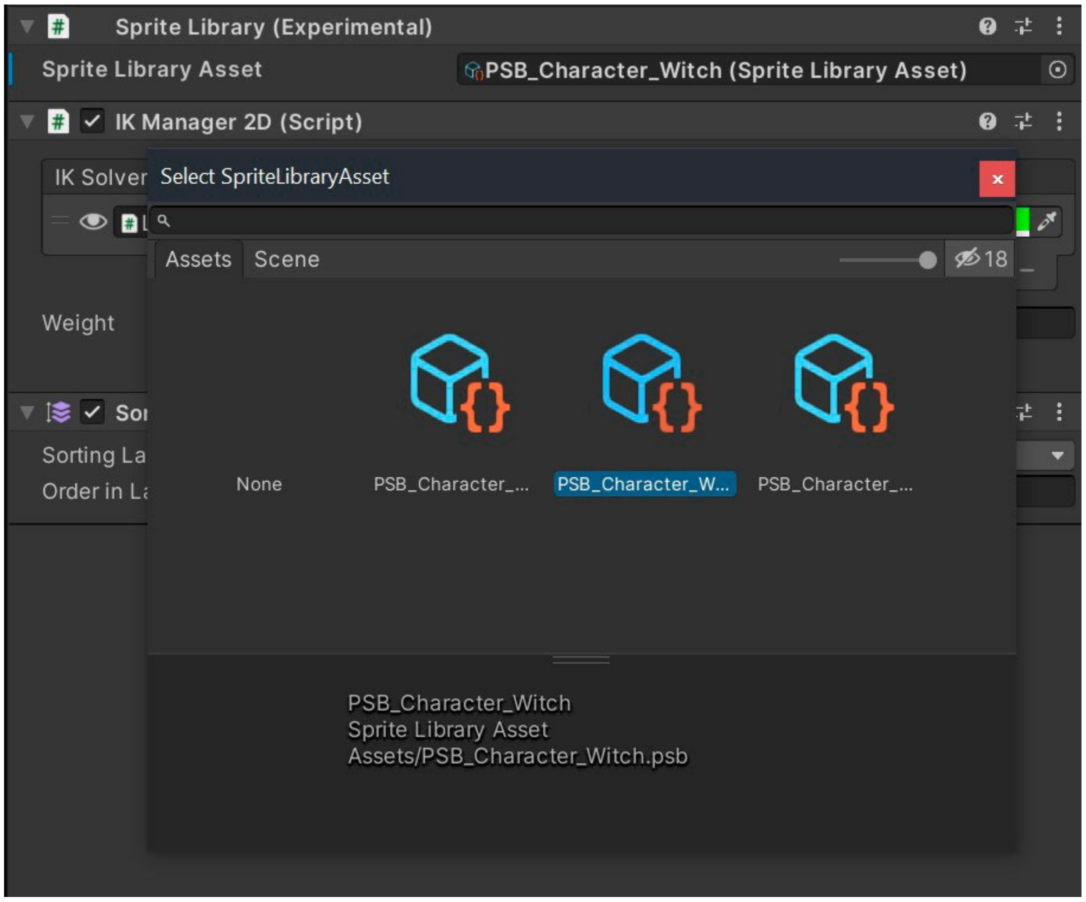
4. Import the PSD or PSB file of the new character, and open its Skinning Editor.

5. Paste in the skeleton copied from step 2 by clicking the **Paste** button on the toolbar. Make sure **Bones** and **Mesh** options are selected in the pop-up window, and click the Paste button to confirm.

6. Fix the geometry so it fits the new sprites. Double check the weights.

7. Go to the Prefab Variant created in step 3. Go to the Sprite Library component and swap the Sprite Library asset for a new character.

8. To create new characters, follow steps 2 through 8.



Sprite Library Asset: PSB_Character_Witch (Sprite Library Asset)

PSB_Character_Witch
Sprite Library Asset
Assets/PSB_Character_Witch.psb

Choose the Sprite Library Asset in the Sprite Library component.

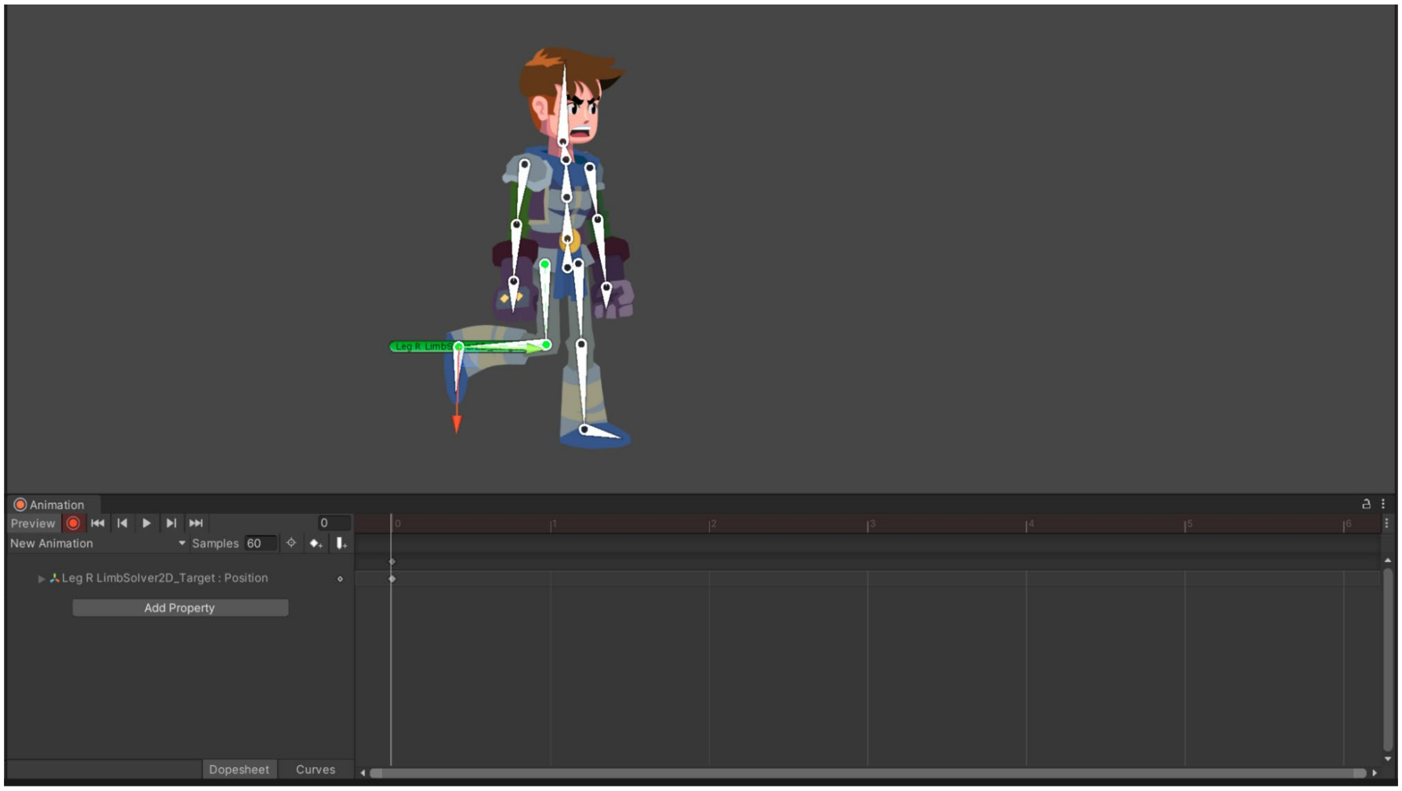
By following this workflow, you'll have one base character prefab. Other characters will be variants of this prefab, so they will have the same components as the base prefab. If changes are made to the base character, for example, adding or changing IKs or Sorting Groups, other characters will inherit those changes.

Animation basics

Animations are very important if you want to create convincing characters.

Creating great animation requires you to learn animation principles and the tools to apply them. Learning basic animation principles is often neglected, but your game will benefit greatly if you spend a little time polishing your character movement.

To create your first animation, open an Animation window by going to **Window > Animation > Animation.**



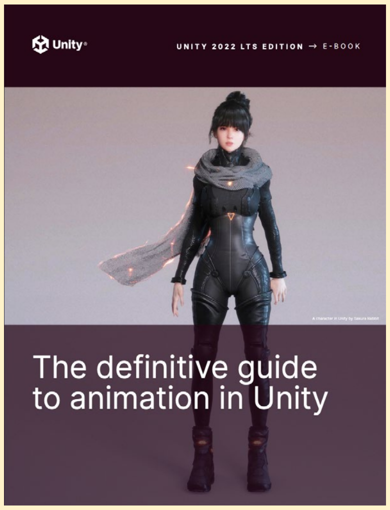
The Animation window, with a character ready to be animated

Select your character. Click the **Create** button to create the first animation clip, and you can start by giving the clip a name. An animation clip is like a linear recording of how an object's position, rotation, scale, and other properties change over time. It also creates an Animator Controller, which takes charge of all animation clips and keeps track of which clip should be playing and when the animations should change or blend together.

Start animating by clicking the red **Record** button. Now, all the changes made to the character will be recorded to an animation clip. You can move bones, IK solvers or swap sprites with the Sprite Resolver and play it back until you are satisfied with the animation clip.

Animation is an art that requires technical understanding of the tools you'll be using. In addition to this e-book, Unity has an in-depth guide focused solely on animation that will help you make your characters truly alive whether they are 3D or 2D. See below for more links, and check out the 2D Animation Sample package in the Unity Asset Store.

Learn more about 3D and 2D animation in Unity with these resources.


Unity
UNITY 2022 LTS EDITION → E-BOOK
The definitive guide to animation in Unity

E-book: The definitive guide to animation in Unity

How to work with humanoid animations in Unity

2D physics

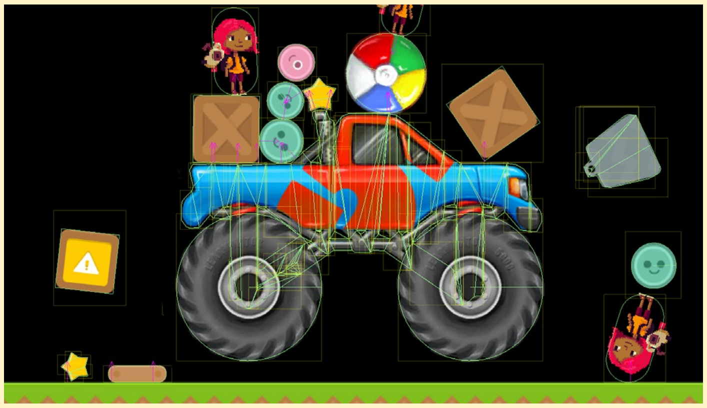
This book is primarily written for artists and tech artists that are new to Unity and want to make the most out of the 2D tools available for visual fidelity.

However, it's more likely than not that you will be working with 2D physics as part of the gameplay logic, whether that's in a small or big team. So, let's look at some of the fundamentals of 2D physics as of Unity 6.3.

For those coming from 3D projects, it's important to know that 3D physics and 2D physics are entirely different systems despite sharing similarities in implementation. The physics for 3D objects will never interact with that of 2D objects.

2D physics enables creating a living world, with systems reacting realistically (or not!) to physics. Characters, vehicles, objects, chains, platforms, can interact with each other and the environment consisting of solid terrain, water, wind forces, or anything that you wish to build.

To get an overview of all the possibilities in 2D physics we recommend getting the sandbox 2D physics sample to see all the systems in action. If you wish to learn more about game design generally in Unity, including physics systems, get the game design e-book for free. The e-book is based on an older Unity version, but most of its content is still applicable with Unity 6.3.



2D Physics samples

E-book: The Unity Game Designer Playbook

Colliders: The interactable area of the object

Objects need to have an area defined in order to interact in the physics world. In 2D development with Unity, you add this area via a 2D Collider component attached to a GameObject. The simpler the shape the more performant it will be. The following Collider components are available:

Edge Collider 2D	—	A collider that's an edge made of line segments that you can adjust to fit the shape of a sprite or any other shape
Box Collider 2D	—	A rectangle with a defined position, width, and height in the local coordinate space of a sprite
Circle Collider 2D	—	A circular collider with a defined position and radius within the local coordinate space of a sprite
Capsule Collider 2D	—	Capsule-shaped collider with smooth, round circumference and absence of vertex corners; allows for smoother movement along other colliders and prevents snagging on sharp corners
Polygon Collider 2D	—	A concave/convex outline that defines an area where physics will create multiple convex polygon shapes
Custom Collider 2D	—	Shape of collider is configured by assigning PhysicsShape2D geometry to it via the PhysicsShapeGroup2D API
Tilemap Collider 2D	—	Generates collider shapes for tiles on a Tilemap component on the same GameObject; add or remove tiles on the Tilemap component and the Tilemap Collider 2D updates the collider shapes during `LateUpdate`
Composite Collider 2D	—	Doesn't have an inherent shape; instead, it merges the shapes of any Box Collider 2D, Polygon Collider 2D, Circle Collider 2D or Capsule Collider 2D that it's set to use

If you want to detect the collision between rigidbody and other rigidbody or collider, you can detect when collision events happen with the use of the API OnCollisionEnter2D, OnCollisionStay2D and OnCollisionExit2D.

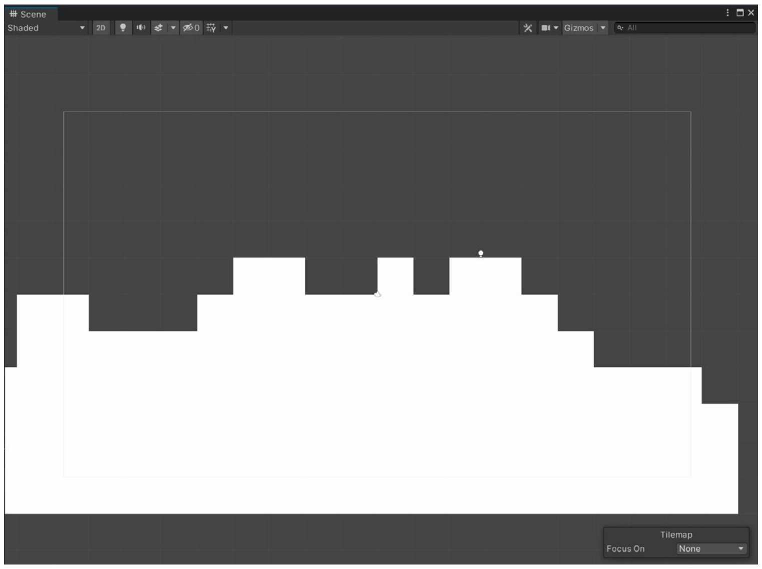
Is Trigger

By checking the box, **Is Trigger**, on the collider 2D, we will make this collider detect another collider without producing any physical movement (ignoring the Rigidbody 2D component if there was one attached to the object). Think of it as sensor, for example, in a stealth game; the enemies could have a Collider 2D with **IsTrigger** to detect the presence of the player in a certain area. When the collider detects your presence they can start chasing the player.

In that case, in C# we would use the OnTriggerEnter2D, OnTriggerStay2D or OnTriggerExit2D events to write some logic when the collision with a trigger happens.

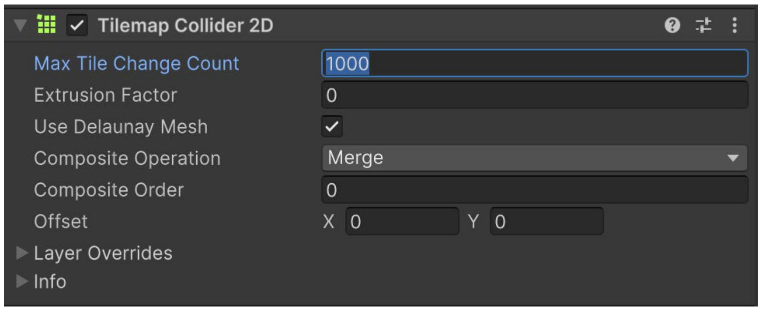
Creating a tilemap collider 2D

We can test physics 2D quickly by creating a tilemap ground, made of simple square tiles for a rectangular tilemap.



A tilemap with painted placeholder tiles

Next, add a Tilemap Collider 2D component to a Tilemap for physics in the Inspector window. This will add colliders to each of the tiles based on the collider type set in the Tile Asset.



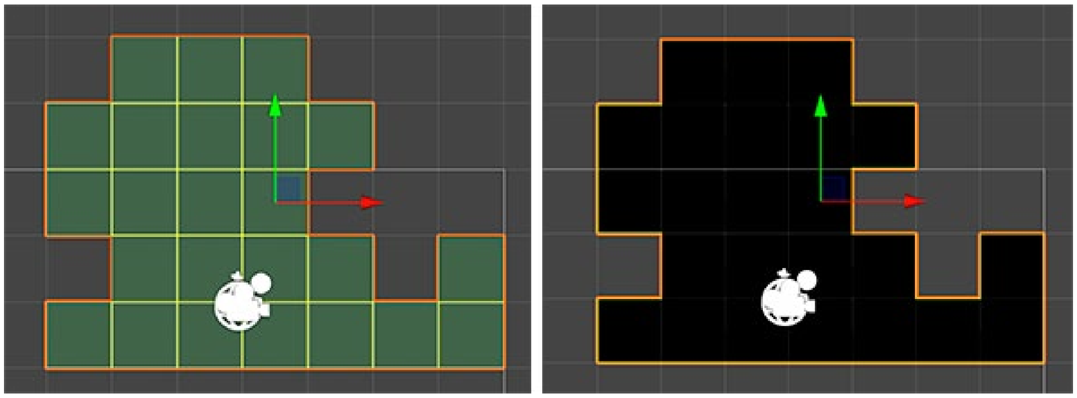
Tilemap Collider 2D

Max Tile Change Count	1000
Extrusion Factor	0
Use Delaunay Mesh	✓
Composite Operation	Merge
Composite Order	0
Offset	X 0 Y 0

▶ Layer Overrides
▶ Info

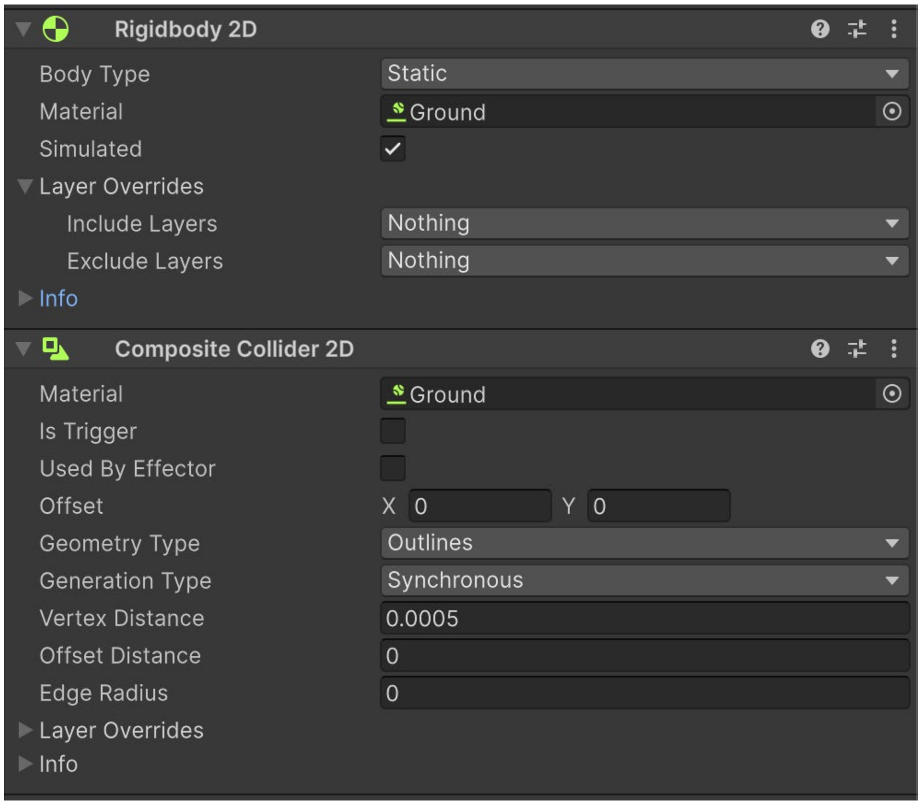
In this example, the option **Use Delaunay Mesh** is enabled, which reduces the number of shapes created in the Collider mesh, as well as the number of small triangles produced, helping to improve overall physics performance.

Add a Composite Collider 2D component in the same GameObject or the parent Grid to combine the colliders into one, resulting in smoother collision behavior with geometry set as Outline. The tiles will behave like a continuous terrain rather than as individual tiles. This setup also brings a slight performance boost. However, if you plan for your game to add or remove tiles at runtime, you might want to keep a collider on each tile.



You can group tiles so they form a single surface with the Composite Collider 2D and the Merge operation. In the left image the tiles aren't merged while in the right image they are.

Remember to set the Composite Operation setting in the Tilemap Collider 2D component to Merge and set the automatically added Rigidbody 2D Type to static so the tilemap, which is supposed to be the ground, doesn't fall.



Rigidbody 2D

Body Type	Static
Material	Ground
Simulated	✓
Layer Overrides	
Include Layers	Nothing
Exclude Layers	Nothing
Info	

Composite Collider 2D

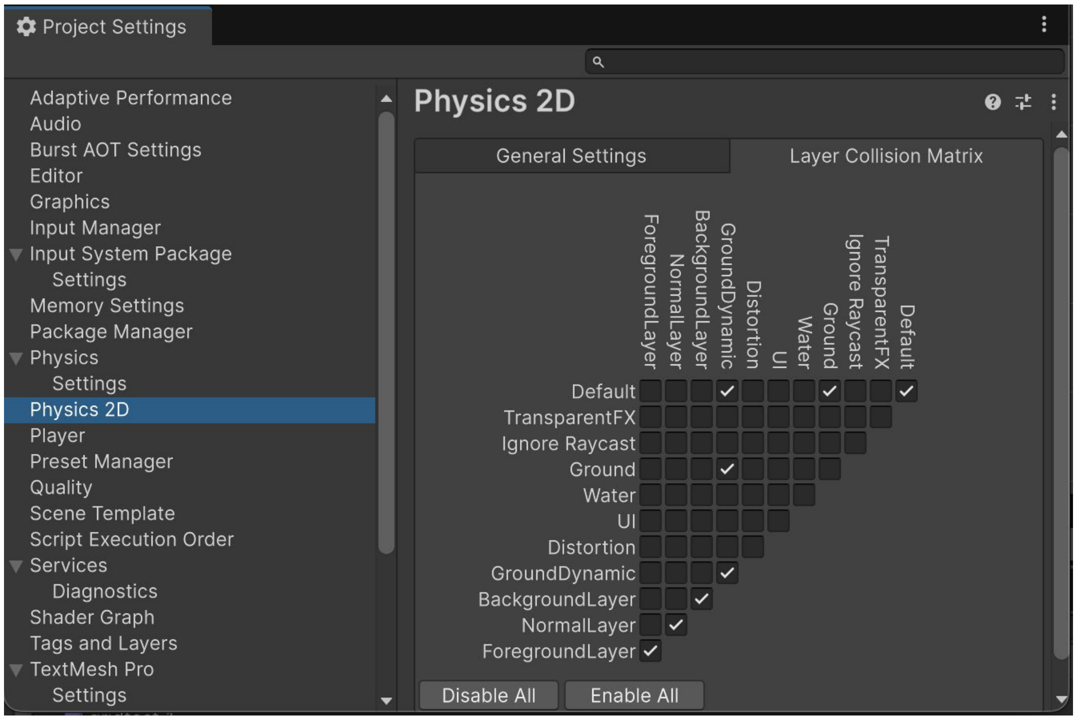
Material	Ground
Is Trigger	
Used By Effector	
Offset	X 0 Y 0
Geometry Type	Outlines
Generation Type	Synchronous
Vertex Distance	0.0005
Offset Distance	0
Edge Radius	0
Layer Overrides	
Info	

The Collider 2D component on a Tilemap GameObject

Collision matrix

GameObject Layers are used by Physics 2D to allow objects to interact with each other. Not everything needs to interact with everything, for example, in a platformer game, you want an enemy to collide with the ground, but not necessarily with other enemies at the same time. In that case you would check that Enemies collide with Ground but Enemies don't collide with the Enemies layer.

You can find these settings under **Project Settings > Physics 2D > Layer Collision Matrix**



	ForegroundLayer	NormalLayer	BackgroundLayer	GroundDynamic	Distortion	UI	Water	Ground	Ignore Raycast	TransparentFX	Default
Default	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	✓
TransparentFX	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	
Ignore Raycast	☐	☐	☐	☐	☐	☐	☐	☐	☐		
Ground	☐	☐	☐	✓	☐	☐	☐	☐			
Water	☐	☐	☐	☐	☐	☐	☐				
UI	☐	☐	☐	☐	☐	☐					
Distortion	☐	☐	☐	☐	☐						
GroundDynamic	☐	☐	☐	✓							
BackgroundLayer	☐	☐	✓								
NormalLayer	☐	✓									
ForegroundLayer	✓										

2D Colliders from Unity 6 also allow specific physics layers to be ignored or considered in physics calculations, on a per object basis, from the Rigidbody 2D component in the Inspector window.

Rigidbody: Make the object move with physics

Now that an object has a defined collider 2D area, you can use it to physically react with one or more Rigidbody 2D components.

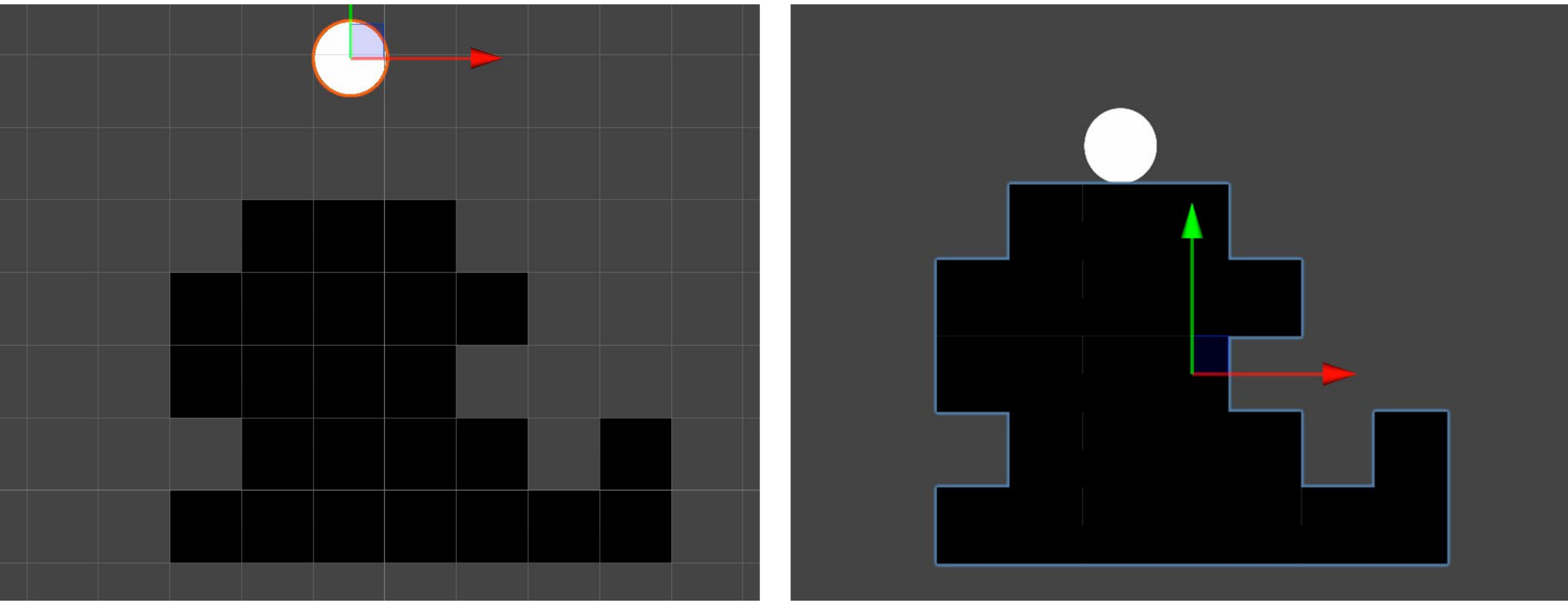
Rigidbodies can be:

— **Static:** they don't move, but other objects will react to them, for example, a ground object would be a static rigidbody

— **Kinematic:** they move without being affected by other collisions, for example, a moving platform, would move the same, regardless of how many objects are on top of it

— **Dynamic:** they move based on their physical properties such as mass, velocity, force, and they are affected by other rigidbodies in contact with them

Let's quickly test this by using the previous tilemap scene; create a new object in the scene via **GameObject > 2D Object > Physics > Dynamic Sprite**. Place this circular object on top of the tilemap and go into Play mode.


We can see the dynamic object bouncing off the surface of the static tilemap realistically

Moving a Rigidbody 2D in code

2D physics objects will behave as real physics objects when they get instantiated on the scene, but it's likely that you need a moving platform to move or a character to move in different directions or jump.

Any physics driven object needs to use the Rigidbody2D APIs to change position; refrain from moving those objects with the GameObject Transform. Instead, use these functions in cases like:

— Control a character's movement by using the AddForce or Slide methods.

— Propel an object or make a character jump by using the AddForce method with the Impulse mode.

— Make a kinematic rigidbody move endlessly, such as a rotating platform, with MoveRotation or MovePosition.

— Teleport a rigidbody from one location to another by changing its position property.

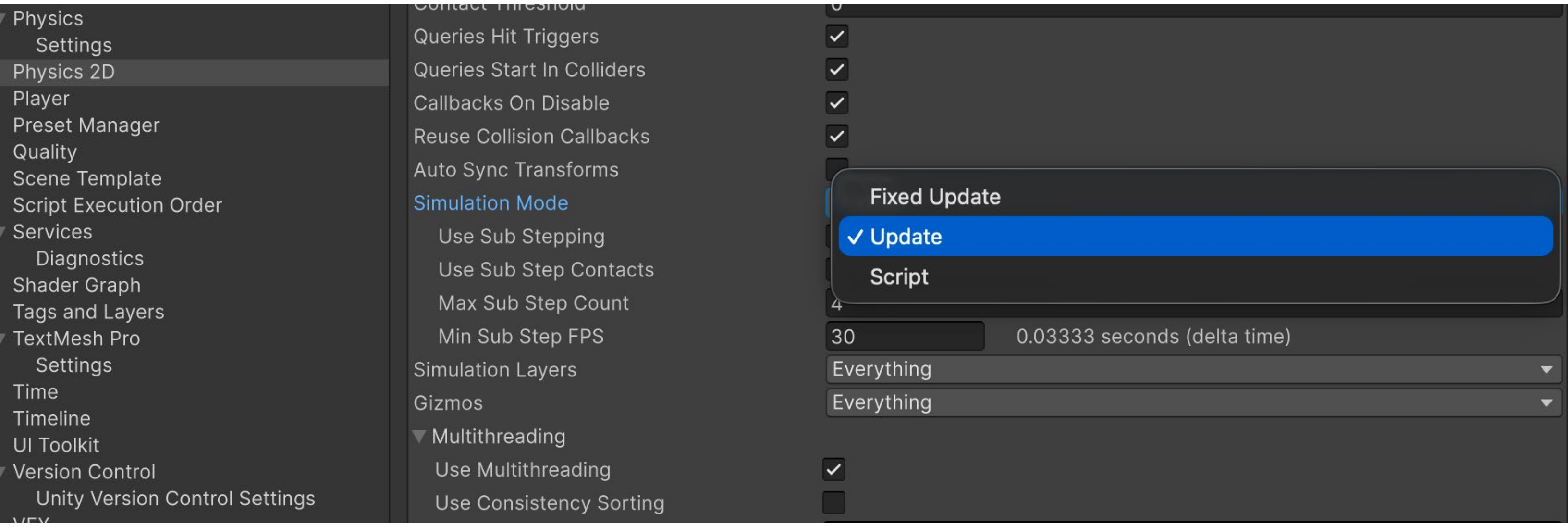
— Evaluate the speed at which a character moves using the linear velocity property.

In code, you should use rigidbody methods in the Update or FixedUpdate cycles. In the project settings you can change from one to the other.

— Update happens every frame and it's a smoother and more precise simulation.

— Fixed update happens at set intervals, by default, every 0.2 seconds.

In both cases remember to multiply values by Time.deltaTime in order to have a constant movement. Time.deltaTime is the time it takes in seconds to process one frame; multiply your motion-related value by this to get a constant result in the simulation.



Queries Hit Triggers ✓
Queries Start In Colliders ✓
Callbacks On Disable ✓
Reuse Collision Callbacks ✓
Auto Sync Transforms
Simulation Mode: ✓ Update
Use Sub Stepping
Use Sub Step Contacts
Max Sub Step Count 4
Min Sub Step FPS 30 0.03333 seconds (delta time)
Simulation Layers Everything
Gizmos Everything
Multithreading
Use Multithreading ✓
Use Consistency Sorting

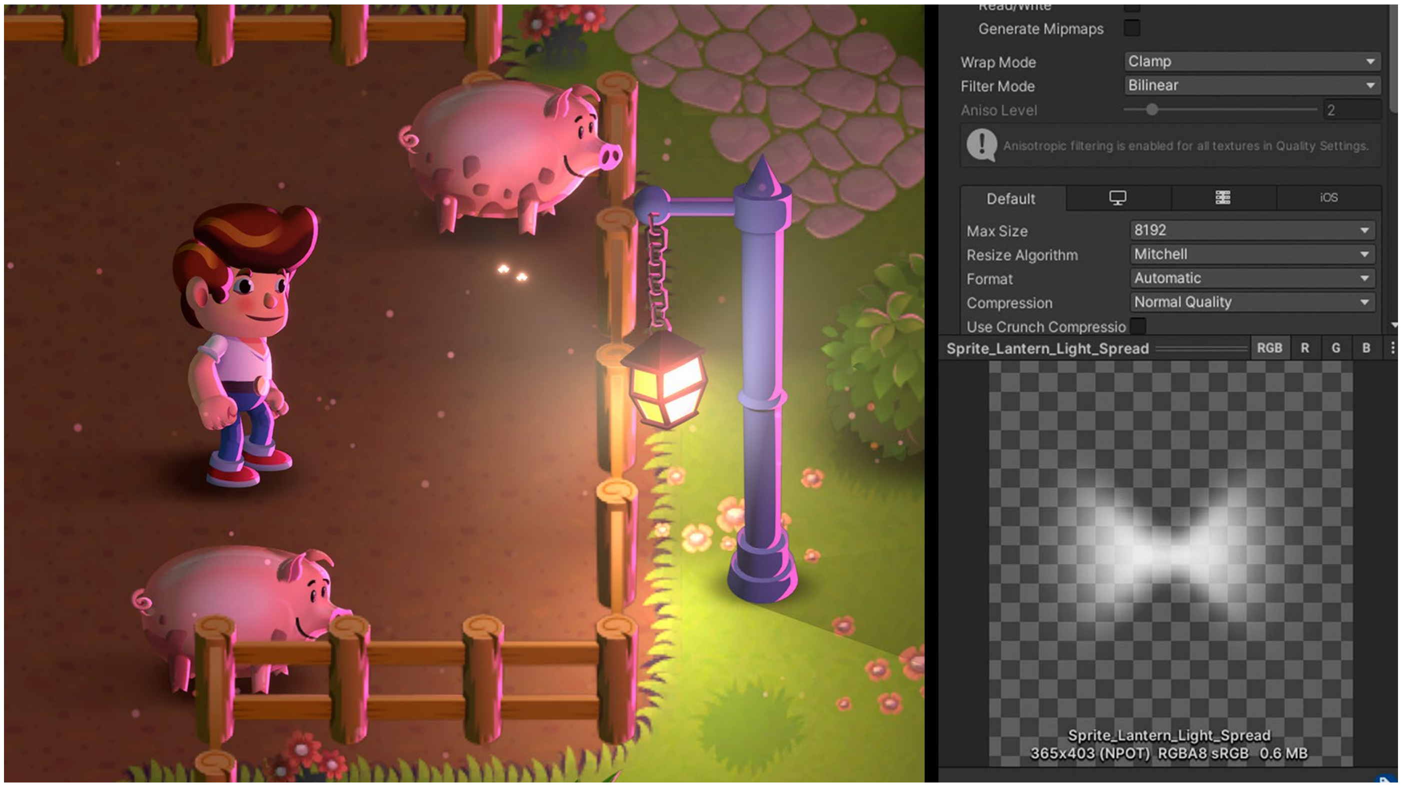
The simulation modes settings

2D lights and shadows

2D games can be as immersive as you aim to make them. Like digital puppet shows, they have many moving parts and layers, including the lighting, and you want your drawings to dynamically react to their environments or lit sources of light.

The Unity 2D light system provides you with ample features for creating rich and varied lighting and visual effects for your game, helping you achieve any type of ambience, atmosphere, or mood you're aiming for.

2D light types and settings



Generate Mipmaps

Wrap Mode: Clamp
Filter Mode: Bilinear
Aniso Level: 2

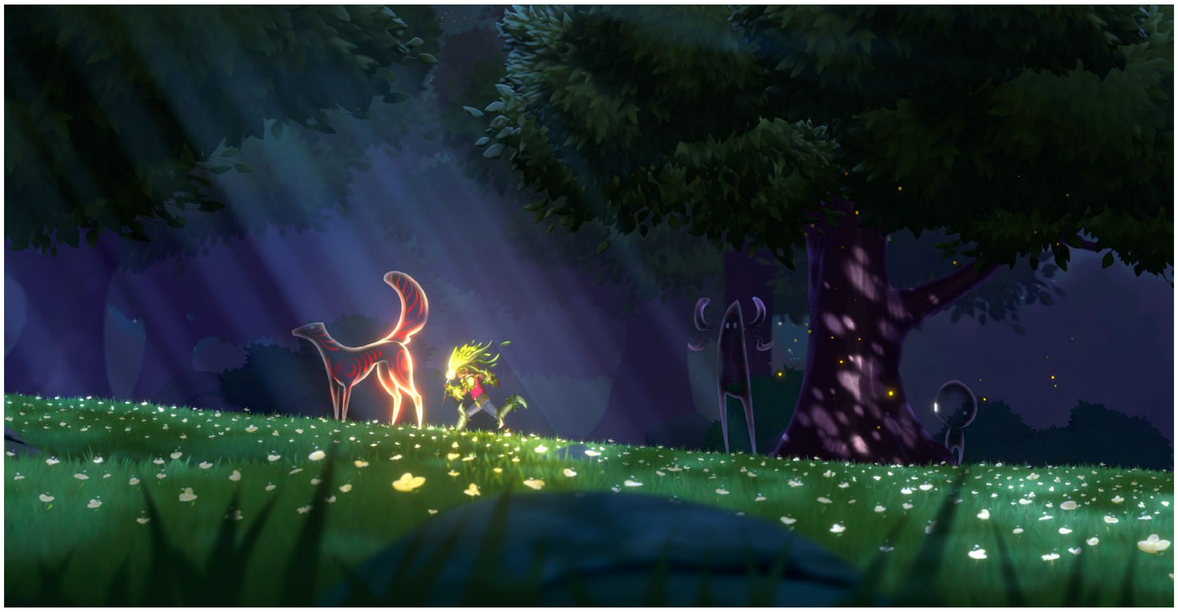
Anisotropic filtering is enabled for all textures in Quality Settings.

Max Size: 8192
Resize Algorithm: Mitchell
Format: Automatic
Compression: Normal Quality
Use Crunch Compressio

Sprite_Lantern_Light_Spread
365x403 (NPOT) RGBA8 sRGB 0.6 MB

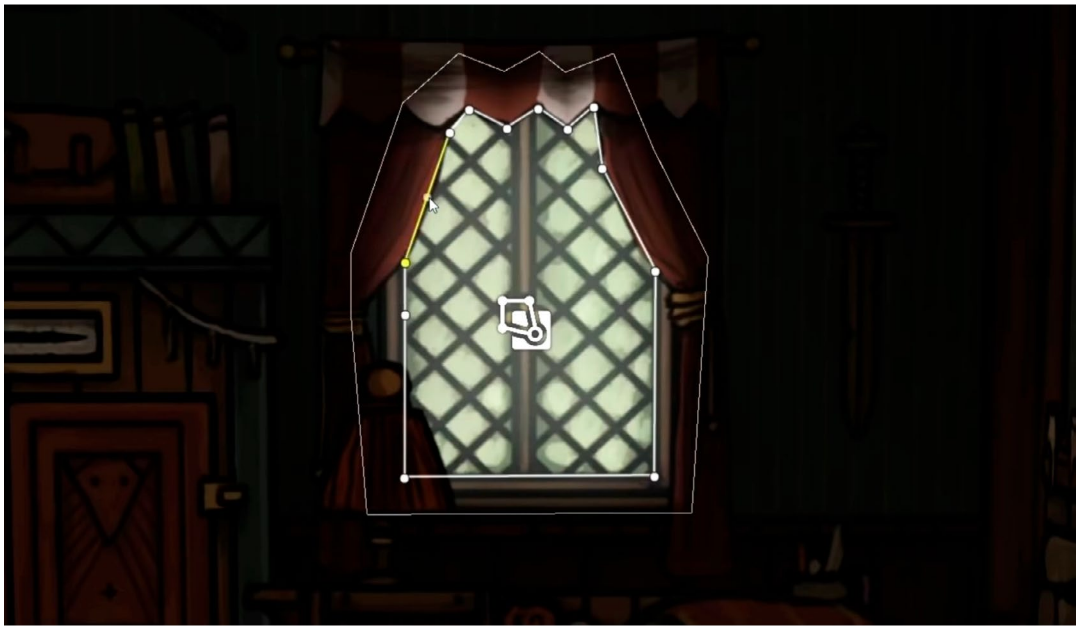
Using a sprite with the light halo for the hanging lamp in *Happy Harvest*

Using Unity's advanced 2D dynamic lighting system, together with techniques like secondary textures on your sprites, can dramatically convey a level's mood and enhance the gameplay.



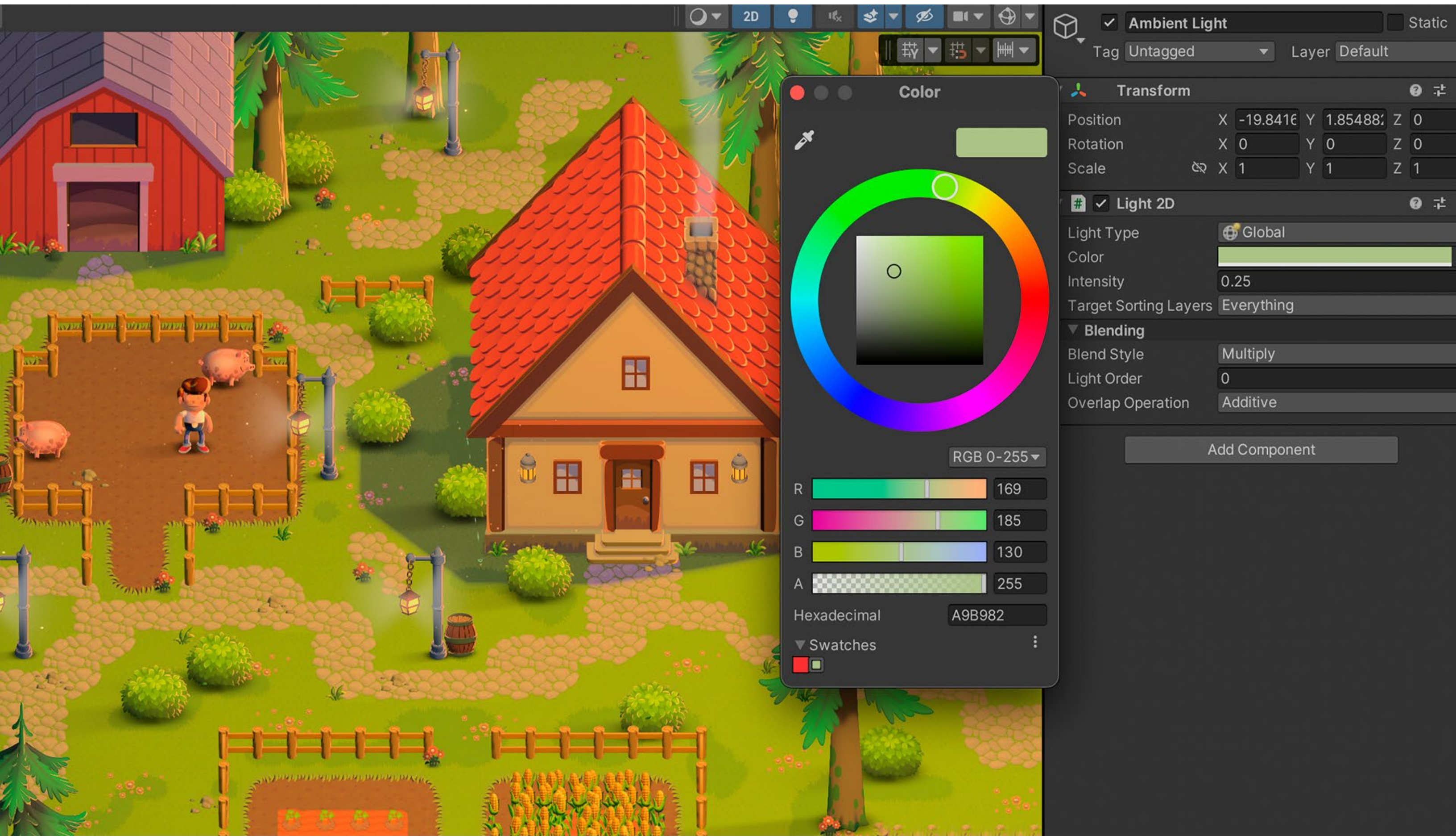
2D games can greatly enhance the mood and immersion with the right usage of the 2D light system. This image is from the Unity sample project, Lost Crypt.

2D Lights are GameObjects that have the Light 2D component attached. They work with the Sprite Renderer, Sprite Shape Renderer, Tilemap Renderer, and from Unity 6.3, with 3D GameObjects with a MeshRenderer and SkinnedMeshRenderer (see the section on mixing 3D and 2D for more information). They also use Sorting Layers, with each light able to affect one or more sorting layers. You can select which layers will be affected in the Target Sorting Layers dropdown list. To control the rendering order of lights that are on the same sorting layer, use the Light Order option.


Examples of a use case for a 2D free form light

There are four different types of 2D lights:

- **Freeform:** These lights can be shaped like an n-sided polygon. This type of light can be used in non-organic or stylized environments. It's a good choice for lighting a large part of the environment (such as a lava pool), simulating light shapes (like god rays coming through a ceiling hole), or conforming to the shape of a window where the light is projected.
- **Sprite**: This shape allows the use of any sprite as a light's texture. This comes in handy if you want a particular shape that is impossible to achieve with other light types. Some good examples of possible textures are lens flares, glares, light cookies, light shape projections, such as disco ball lights, or baby lamps projecting stars against the wall.
- **Spot:** The shape of this light can be a circle or a circle sector. This option is good for spotlights or to light a specific point with torch fires, candles, car lights, flashlights, volumetric light, and so on.
- **Global:** This light doesn't have a shape but instead lights all objects on the targeted sorting layers. Only one Global Light can be used per Blend Style (the method of interaction between light and the sprites), and per sorting layer. Use it first to add a base environment light.



Ambient Light — Static
Tag: Untagged — Layer: Default

Transform
Position: X -19.8416 Y 1.854882 Z 0
Rotation: X 0 Y 0 Z 0
Scale: X 1 Y 1 Z 1

Light 2D
Light Type: Global
Color
Intensity: 0.25
Target Sorting Layers: Everything

Blending
Blend Style: Multiply
Light Order: 0
Overlap Operation: Additive

Color
RGB 0-255
R 169
G 185
B 130
A 255
Hexadecimal: A9B982
Swatches

Global lights affect the whole scene, making it easy to change the mood of the game world. There should only be one Global light in the scene; by manipulating its parameters you can easily simulate different environment conditions, such as night time, by lowering intensity and applying a purple tint to the scene.

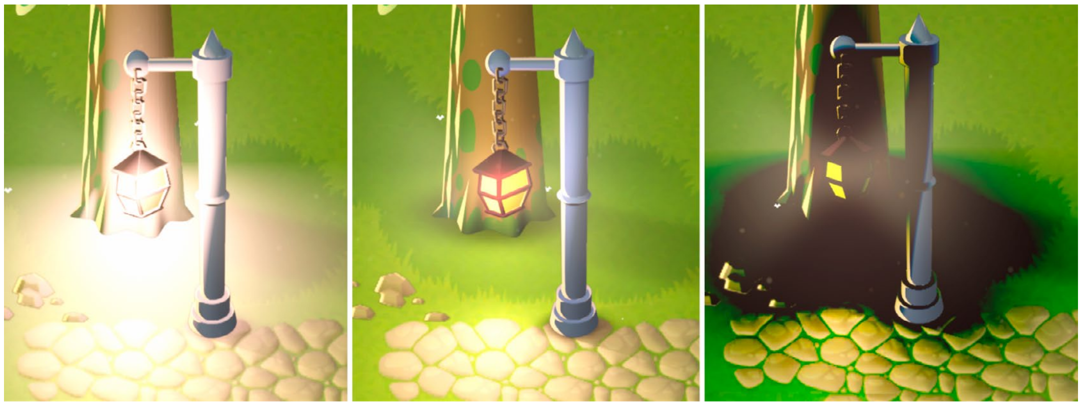
These settings are available for each light type:

— **Blend style:** A Blend Style determines the way a particular light interacts with sprites in the scene. All lights in a scene must pick from one of the four available Blend Styles defined in the 2D Renderer. There are three possible Blend Modes, with each mode controlling the way a sprite is lit by light:

  — **Additive:** Adds each pixel RGB value to the value of the pixel in the base layer. Adds brightness

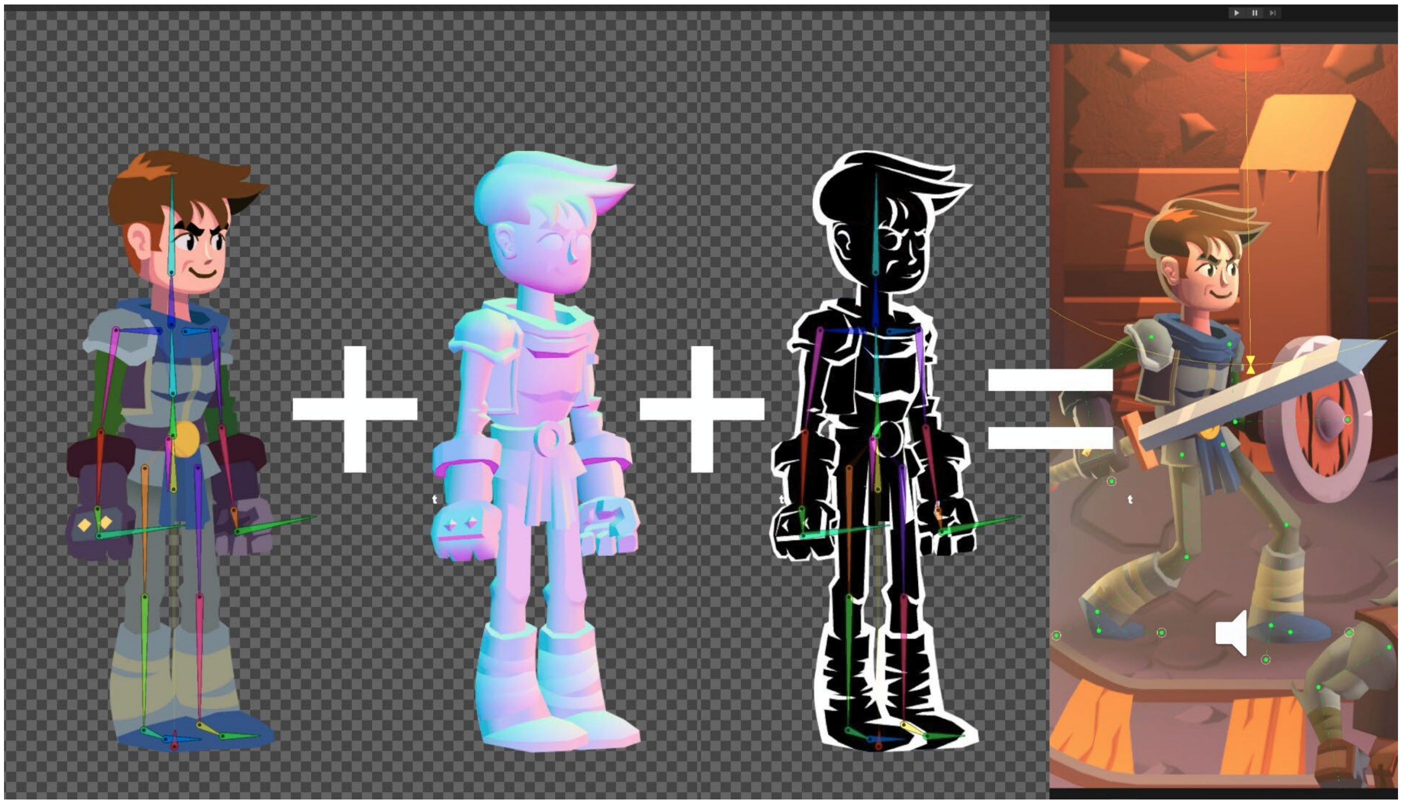
  — **Multiply:** Multiplies each pixel RGB (normalized) value with the value of the pixel in the base layer; blends colors

  — **Substractive:** Substracts each pixel RGB value to the value of the pixel in the base layer; makes darker tones

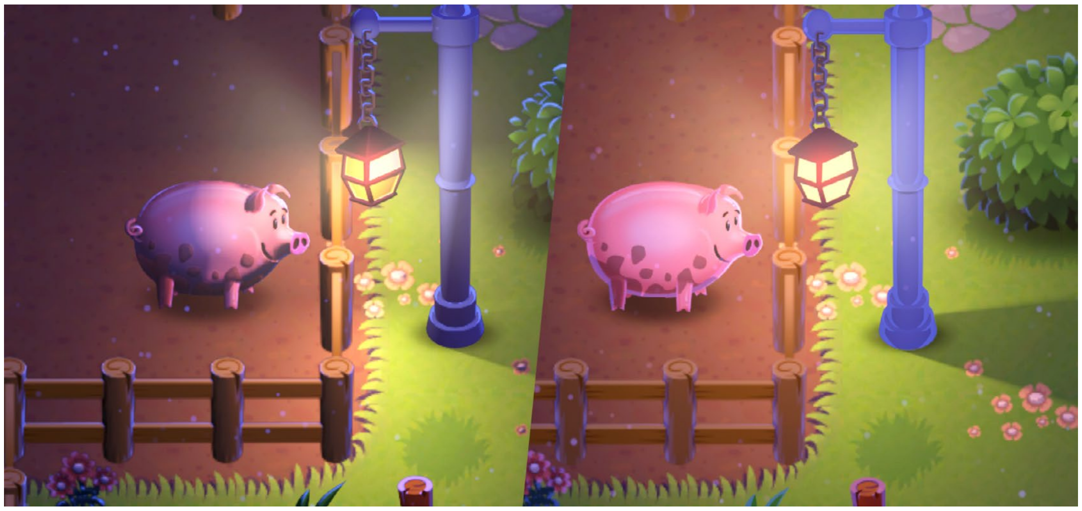

Blend styles in action, from left to right: additive, multiply, subtractive

- **Light Order**: This value determines the position of the light in the render queue relative to other lights that target the same sorting layer(s).
- **Overlap Operation**: This property controls the way the selected light interacts with other rendered lights. There are two modes available with this property:
  - **Additive:** The light is blended with other lights additively, where the pixel values of intersecting lights are added together; this is the default light blending behavior.
  - **Alpha Blend:** Lights are blended together based on their alpha values. This can be used to overwrite one light with another where they intersect, but the render order of the lights is also dependent on the Light Order of the different lights.
- **Shadows**: When you enable this option, the GameObjects with a Shadow Caster 2D component will cast shadows from this light source.
- **Volumetric**: This allows you to control the falloff of the light and darkness of the shadow projected.
- **Normal maps**: When enabled the normal map information on the sprite will be used to calculate the amount of light based on the pixel direction (read more on secondary textures below).

Secondary textures


Adding secondary textures to sprites

Secondary textures are created with normal maps and mask maps. They're an option to use with 2D lights that can take your game's visuals to a whole new level if you have the budget, time, and art resources for it. Characters and the environment will have even more details that react to light, making their shapes look more defined and three-dimensional.

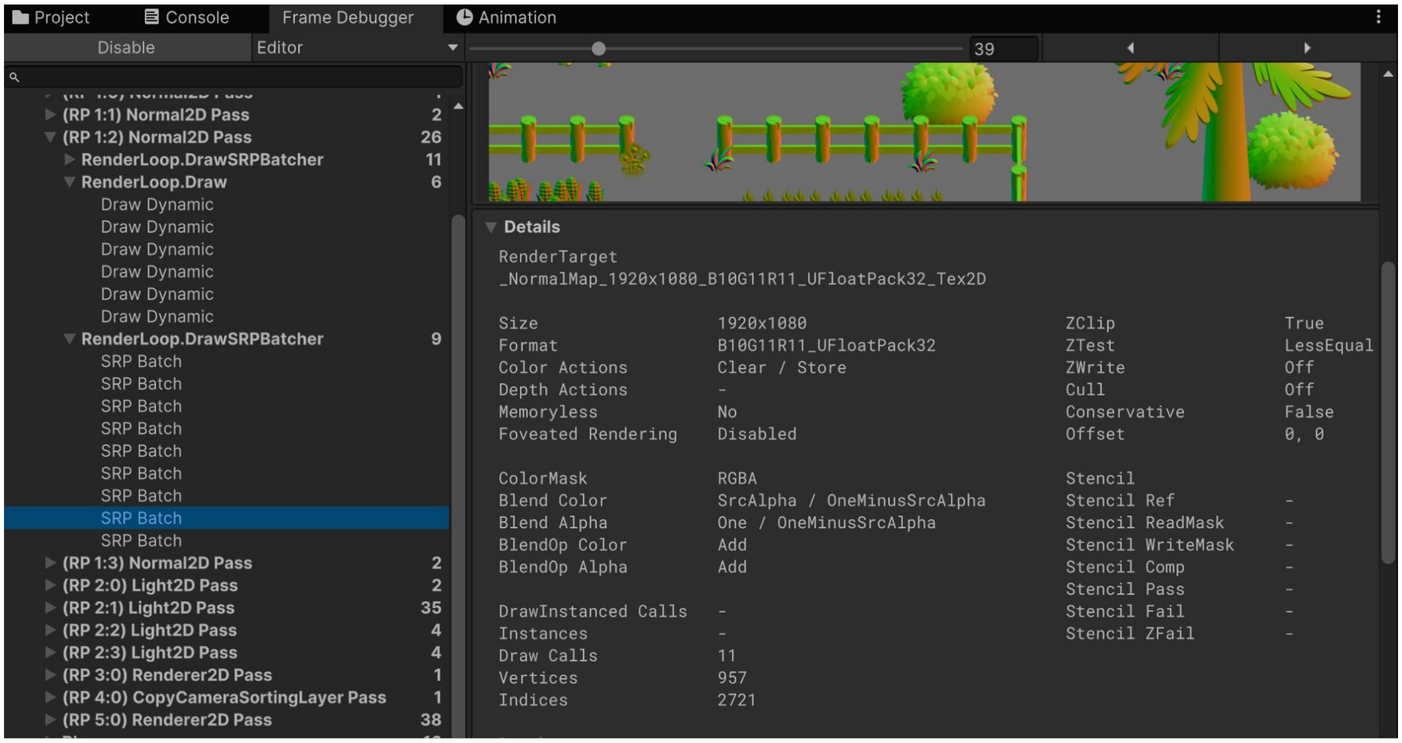

A happy piggie in *Happy Harvest*, bathed in the warm glow of a lantern: Lights using normal maps on the left, and without normal maps on the right

Every sprite asset can have optional secondary textures assigned that can then be used by materials. Change the secondary textures via **Sprite Editor > Secondary Textures**. By default, you can assign two types of textures: normal map and mask map.

There are some good reasons to assign textures on an asset level instead of in the material.

One benefit is that renderers that use sprites can share a material (Sprite-Lit-Material), even when the sprites and secondary textures are different. This means they can be batched to render more efficiently.

You'll need to add normal maps and mask maps to a material when you have a sprite shader. Every sprite would need its own material with a sprite, normal map, and mask map. That would quickly escalate to hundreds of materials in the project. However, when setting the additional textures for every sprite, you can use just one sprite material. One material also means one draw call instead of hundreds.



```
(RP 1:1) Normal2D Pass                        2
(RP 1:2) Normal2D Pass                       26
  RenderLoop.DrawSRPBatcher                  11
  RenderLoop.Draw                             6
    Draw Dynamic
    Draw Dynamic
    Draw Dynamic
    Draw Dynamic
    Draw Dynamic
    Draw Dynamic
  RenderLoop.DrawSRPBatcher                   9
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
    SRP Batch
(RP 1:3) Normal2D Pass                        2
(RP 2:0) Light2D Pass                         2
(RP 2:1) Light2D Pass                        35
(RP 2:2) Light2D Pass                         4
(RP 2:3) Light2D Pass                         4
(RP 3:0) Renderer2D Pass                      1
(RP 4:0) CopyCameraSortingLayer Pass          1
(RP 5:0) Renderer2D Pass                     38
```

```
Details
RenderTarget
_NormalMap_1920x1080_B10G11R11_UFloatPack32_Tex2D

Size                 1920x1080                        ZClip              True
Format               B10G11R11_UFloatPack32           ZTest              LessEqual
Color Actions        Clear / Store                    ZWrite             Off
Depth Actions        -                                Cull               Off
Memoryless           No                               Conservative       False
Foveated Rendering   Disabled                         Offset             0, 0

ColorMask            RGBA                             Stencil
Blend Color          SrcAlpha / OneMinusSrcAlpha      Stencil Ref        -
Blend Alpha          One / OneMinusSrcAlpha           Stencil ReadMask   -
BlendOp Color        Add                              Stencil WriteMask  -
BlendOp Alpha        Add                              Stencil Comp       -
                                                      Stencil Pass       -
DrawInstanced Calls  -                                Stencil Fail       -
Instances            -                                Stencil ZFail      -
Draw Calls           11
Vertices             957
Indices              2721
```

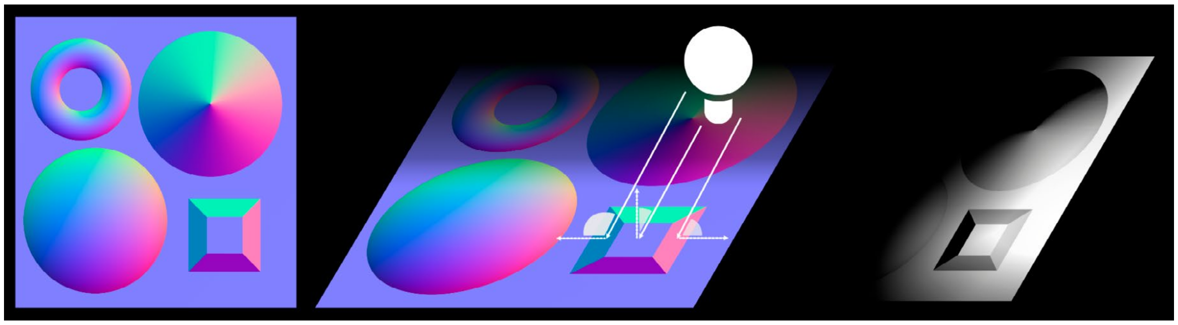
Using the Frame Debugger allows you to observe every step taken to render a frame. Having many materials that can't be batched takes a toll on performance.

Get more tips from the optimization section towards the end of the e-book.

Normal maps

A normal map done well can make or break the illusion of a sprite being 3D. Every pixel in a normal map stores data about the angles of the main texture. The red, green, and blue (RGB) channels store angle data for the X, Y, and Z coordinates. Every light that uses a normal map has a direction, and pixels on a texture with a normal map are shaded based on this direction and the direction of the pixel. This works the way it does in real life – if a pixel is facing the light's direction then it will be lit, and if it's facing away it receives no light.

Let's look at how RGB values affect the angles of a normal map.



In a normal map, the light received depends on how big the angle is between the vector that the pixel is facing and the source of light (also known as dot product). The smaller the angle the more aligned both vectors are and the more light the pixel receives. On the right in the image above, the normal map is lit by a 2D light in Unity.

The above image is a normal map in which the purple pixels are facing the camera. Its RGB values are 127, 127, and 255, respectively. Each color channel can have a value from 0 to 255, so 127 is near the middle. To face the surface left (-90 degrees), the R color value needs to be set to 0. To face the surface right, set R to 255. To face straight down or up, set the G channel to 0 or 255, respectively.

Examples of normal and mask maps applied to animated characters, props, and tilemaps in *Happy Harvest*

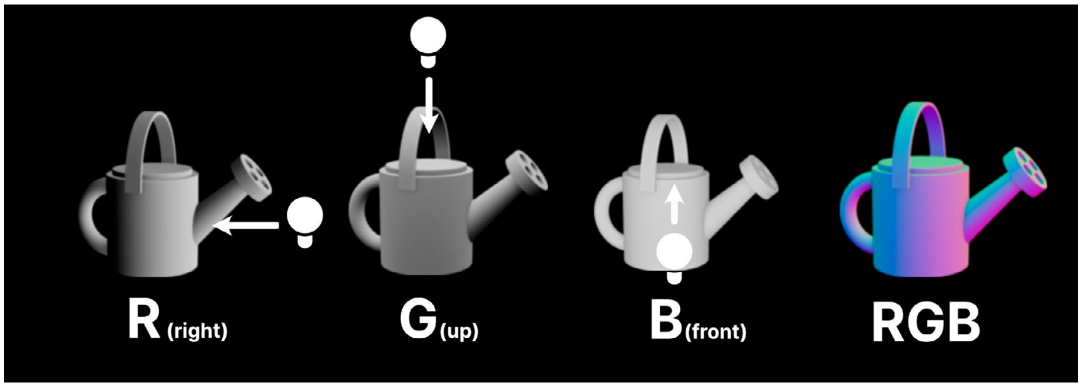
Painting techniques for normal maps

How you create normal maps depends largely on the art styles and skills you're employing. The techniques to create normal maps for 3D materials apply generally to 2D sprites as well. Use the following tips to help you have full control on the appearance.

Paint light directions manually

One way to paint a normal map is to make drawings of your sprite that are lit from different angles, then combine them into one texture. The sprite will be lit with one light from the right in the R channel and one light from the top in the G channel. In the B channel, the sprite is lit from the front, but for the sake of simplicity, you can omit this channel when using a normal map with 2D sprites. This is because front lighting in 2D won't add that much to the overall shading.

However, this approach can be time-consuming, as you will need to paint your shading at least twice for the X and Y axis.



How to combine two shaded grayscale (lit from right and top) images into a normal map; the front shaded grayscale can be optional

Normal map generators

Another painting approach is to use a normal map generator app. Open a sprite in such an app and you can generate a normal map with just a couple of clicks. Generator apps do not take into account the angles of your sprite, so avoid using them on the entire sprite. They also don't recognize the objects. They instead estimate shapes from the sprite colors or by adding a general filter similar to bevel or emboss from image editing apps or a relief sculpture. They can't recognize the angles of the face, for example, but attempt to guess where there should be a change in an angle. It's a limitation, but they're still useful for generating the normal maps of sprite sections that are beveled, like chains, cables, or a dragon's tail, and for surface normals for things like bricks, stones, and wood.

Import a section into the normal map generator, tweak the values, export, and then add the necessary parts and details yourself.

Some normal map generators include:

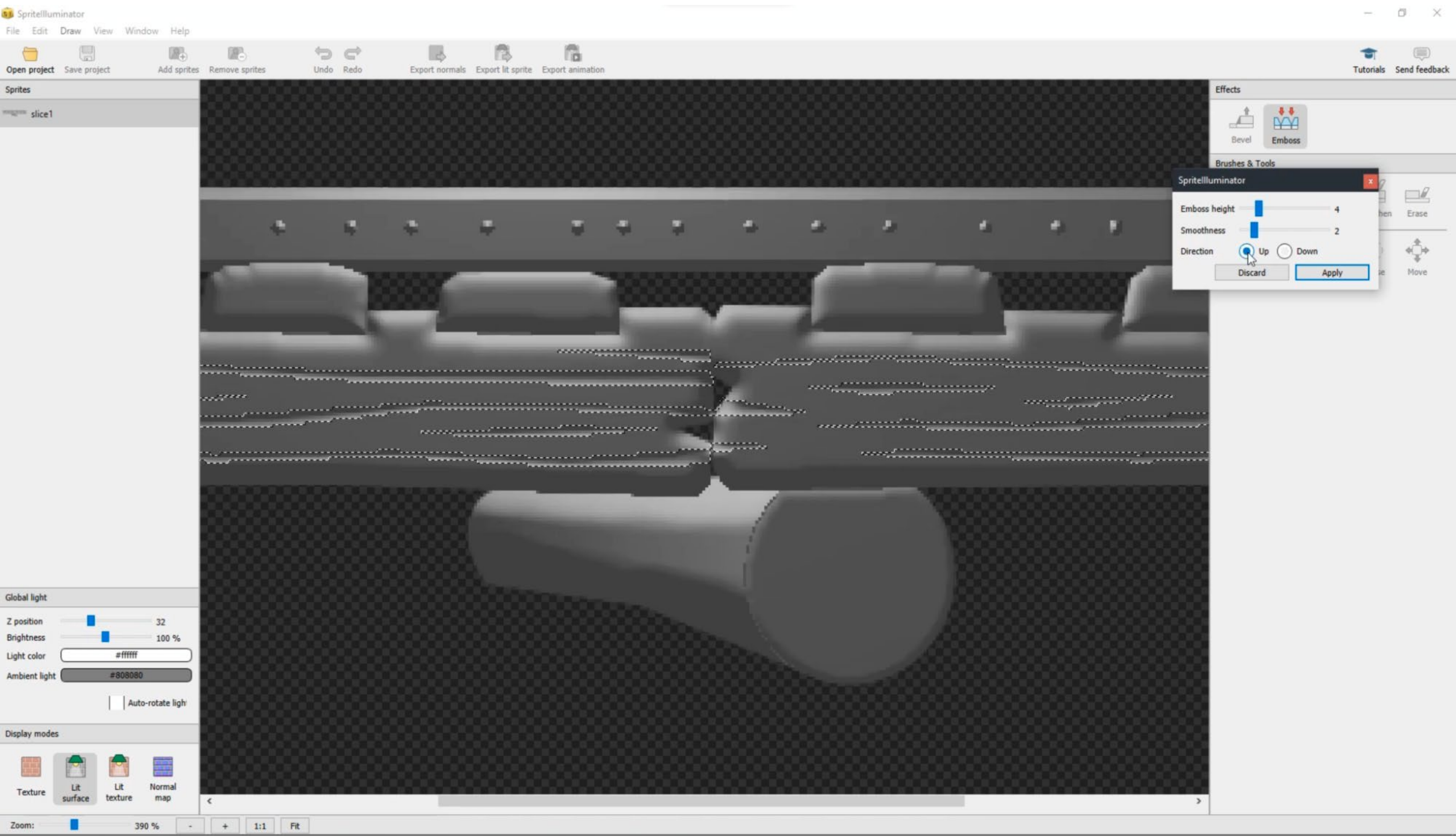
— SpriteIlluminator

— NormalPainter

— Krita's Tangent Normal Brush

— Laigter

— Sprite Lamp



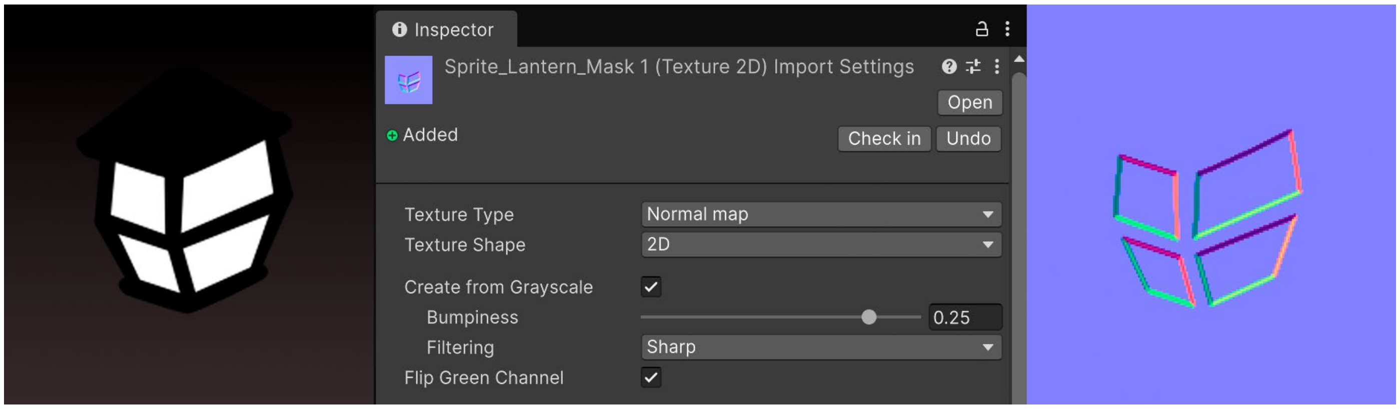
Emboss height 4
Smoothness 2
Direction [✓ Up] Down

Z position 32
Brightness 100 %
Light color #ffffff
Ambient light #808080
Auto-rotate ligh
Zoom: 390 %

Generating crevices in a normal map by using the emboss tool in SpriteIlluminator

Normal map from greyscale in Unity

For this technique, Unity offers a way to generate normal maps from a grayscale heightmap. This is a texture where black represents the minimum surface height and white the maximum height. You need to import an image as a normal map and check the **Create from Grayscale** option. This technique is handy for quickly generating normal maps without leaving the engine.



Sprite_Lantern_Mask 1 (Texture 2D) Import Settings

Texture Type: Normal map
Texture Shape: 2D
Create from Grayscale: ✓
Bumpiness: 0.25
Filtering: Sharp
Flip Green Channel: ✓

Converting a texture to a normal map using the Create from Grayscale option; the **Flip Green Channel** option can make the emboss effect look like it's either protruding or indented

When using this method, the **Bumpiness** slider will appear in the Inspector. Unity uses pixel brightness and converts the height differences to normal map angles, with the steepness of the angles controlled by the slider. A low bumpiness value means that even a sharp contrast in the heightmap will be translated into softer angles and bumps.

Sampling colors

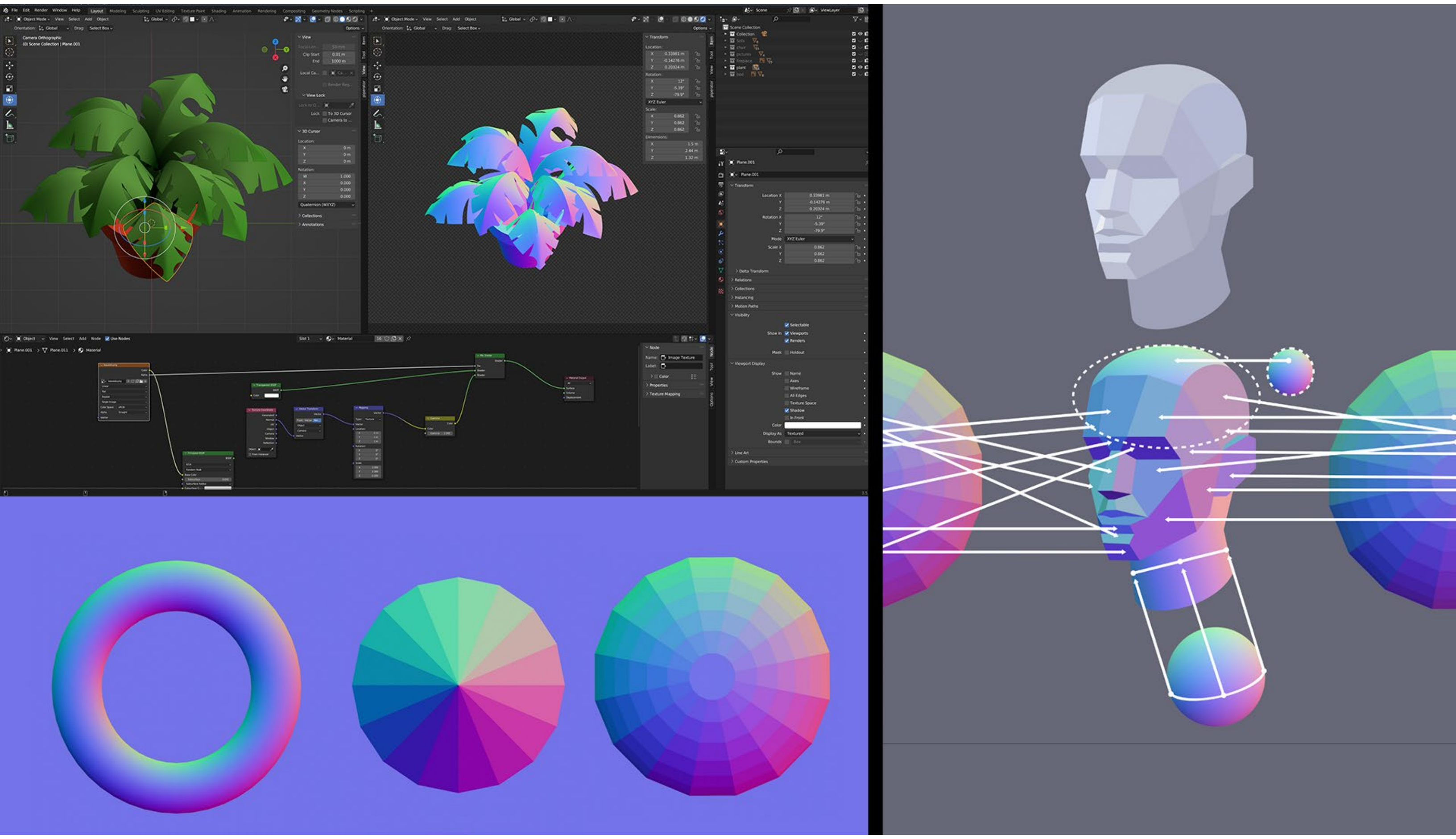
To paint a normal map, start by learning which colors to use for different angles.

First, obtain a normal map palette so you can sample the colors used to represent the surface angles in any template online. No matter how your painting workflow looks, you'll only need to copy a palette to your favorite painting app and use the color picker to select a color to paint on your normal map.

Angle colors don't need to be 100% accurate; a few degrees won't make a difference. However, be sure to keep the overall shape of the sprite believable. If you use an angle color that doesn't make sense in-context, the whole shape will fall apart when lit.

Painting normal maps can be tricky initially because it requires a good spatial imagination. A great place to start is with something simple like the base planes of the head. The simplified model of a human head to the right in the image below has a low-poly look, you can check out the prepared example by downloading this zip file.

When painting a normal map, try to imagine the basic 3D shapes that are parts of your sprite, then visualize the angles of each individual part. If you know the angle, you'll know from which part of the palette sprite to sample the color.

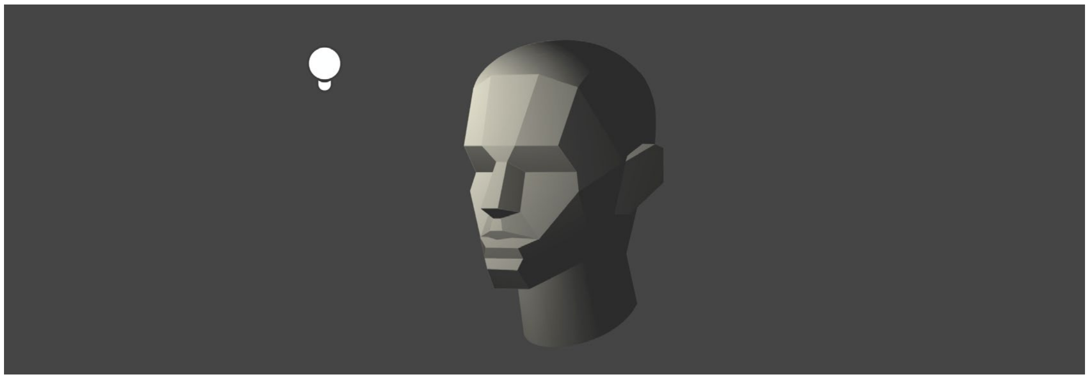


From the top left: Creating a prop in Blender for normal map generation, painting a head's normal map by sampling colors from a template, and a template of shapes

The example of the human head is working on a flat surface, but the process is similar when you're painting with softer brushes. You can blend hard edges to achieve a more natural look.

A couple of shortcuts to note: When there's a spherical shape, you can paste the normal sphere from your palette. When you have a cylindrical shape, you can take a part of the sphere, paste and stretch it, or make a gradient.

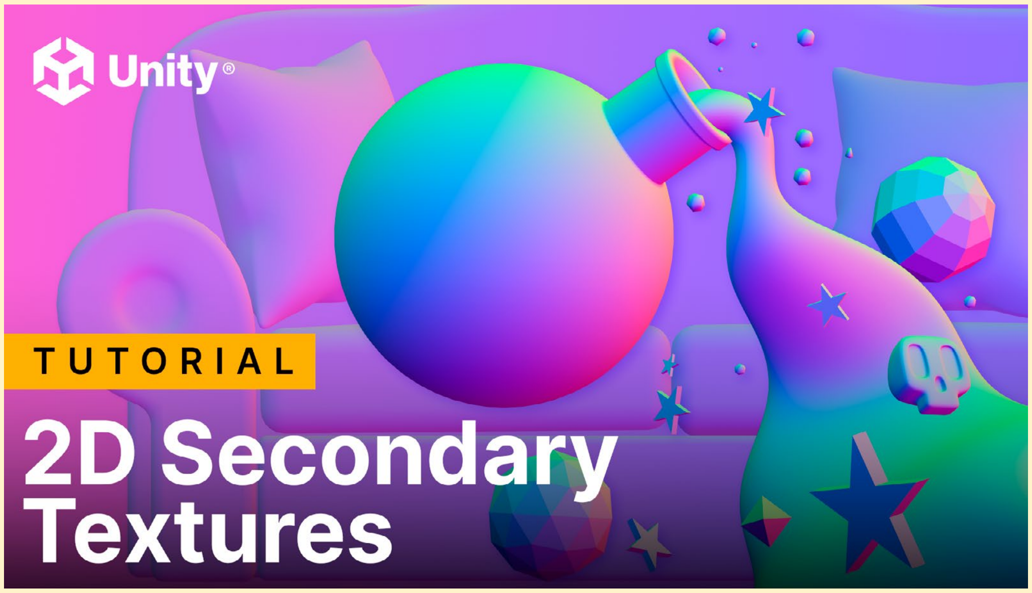
Be aware that copying and pasting parts of normal maps and rotating them breaks the shading. However, this can also be used to your advantage. For example, when you need a concave spherical shape, just rotate the sphere 180 degrees to create a hole.



This is how the normal map from this book's sample looks on a white sprite in-engine.

Choose the method of generating normal maps that works best for you. Most likely there will be many assets made for your game, so focus on the objects that will be most visible and simplify the other parts of the game.

The different painting techniques are explained in the following video:

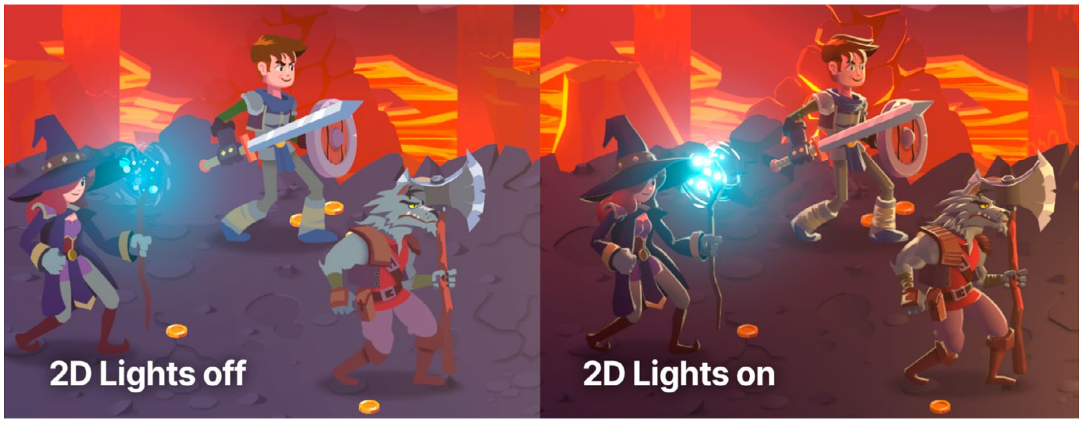


Secondary textures: Lit sprites and 2D VFX

Preparing sprites for 2D lighting

Take note of the base-color sprite. If you plan to use 2D lighting extensively in your game and want to make the most of the normal maps, don't paint the light and shadow onto the sprite.

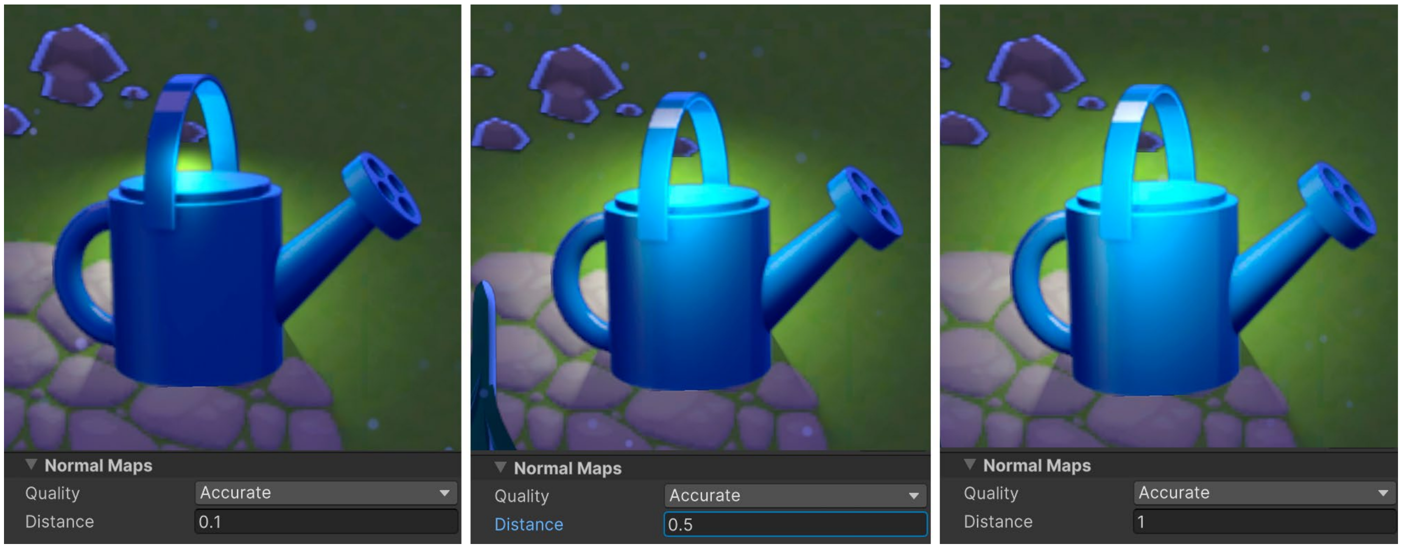
2D lighting doesn't look good on a sprite that already has shadows painted on. You will also end up doing double the amount of work because you'll be painting the lighting in the normal maps. Instead, paint some non-directional shadows, and your sprite will look better as long as you avoid any directional light, such as from the sun.



With 2D lights turned off, the sprite has the color information (albedo) but looks flat because it doesn't contain light or shadow information.

Enable normal maps

Lights and normal maps are used everywhere in the Unity samples *Happy Harvest* and *Dragon Crashers* to create the illusion of volume. You can use normal maps with Spot, Point, and Freeform lights. Remember that you need to enable normal maps in the 2D light object to make use of them in the sprites. You can also control the distance between the light and the sprite surface. Increasing or decreasing the effect.


Normal Maps
Quality: Accurate
Distance: 0.1

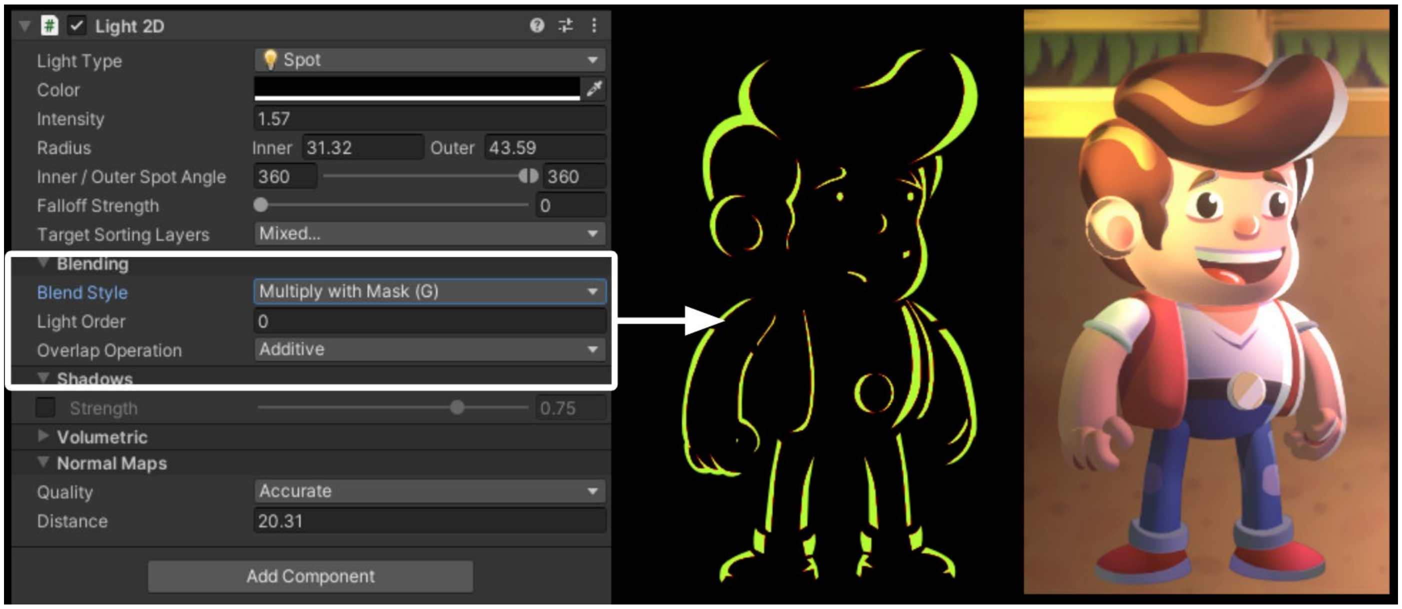
Normal Maps
Quality: Accurate
Distance: 0.5

Normal Maps
Quality: Accurate
Distance: 1

Normal map support settings on 2D Light and the effect with different settings for distance

Mask maps

Mask maps can be used for additional lighting like rim lighting and more. You can add your own textures and reference them by name in Shader Graph shaders for any other visual effect.



Light 2D
Light Type: Spot
Color
Intensity: 1.57
Radius: Inner 31.32 Outer 43.59
Inner / Outer Spot Angle: 360 360
Falloff Strength: 0
Target Sorting Layers: Mixed...
Blending
Blend Style: Multiply with Mask (G)
Light Order: 0
Overlap Operation: Additive
Shadows
Strength: 0.75
Volumetric
Normal Maps
Quality: Accurate
Distance: 20.31

A “hero” lighting style was created to help the farmer character from *Happy Harvest* stand out. This effect uses a blending style that only affects the mask map in the green channel, the one that is used to paint the highlights of the character.

Rim lighting is an effect that’s used to highlight the contours of a character. It simulates light coming from behind an object and the natural properties of light scattering. This is called the Fresnel effect. In real life, more light reflects from objects when the angle at which the light hits the surface is wide.

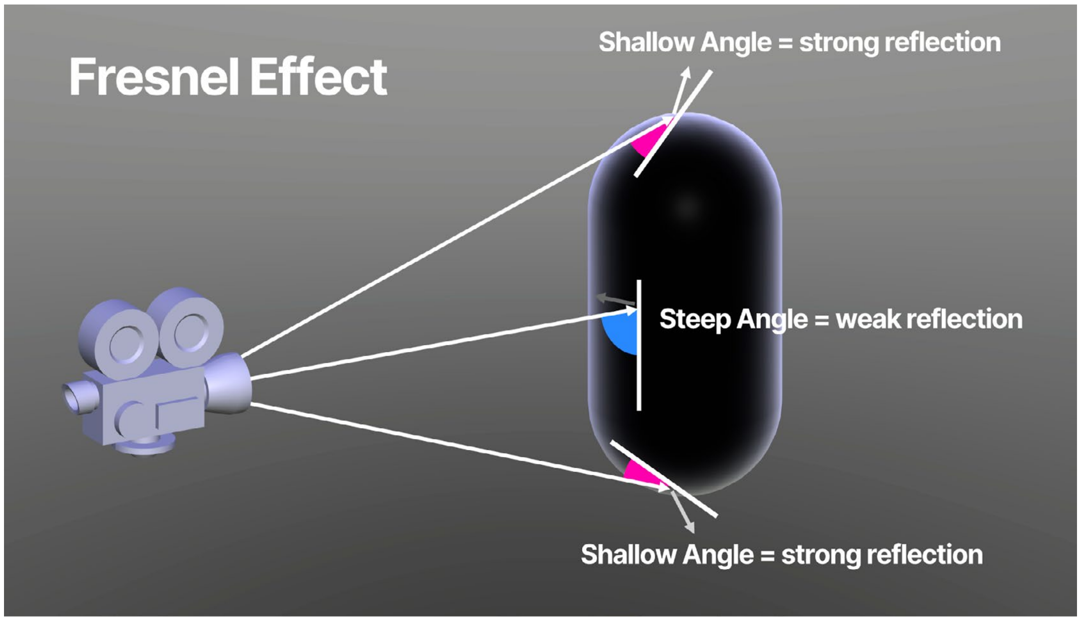
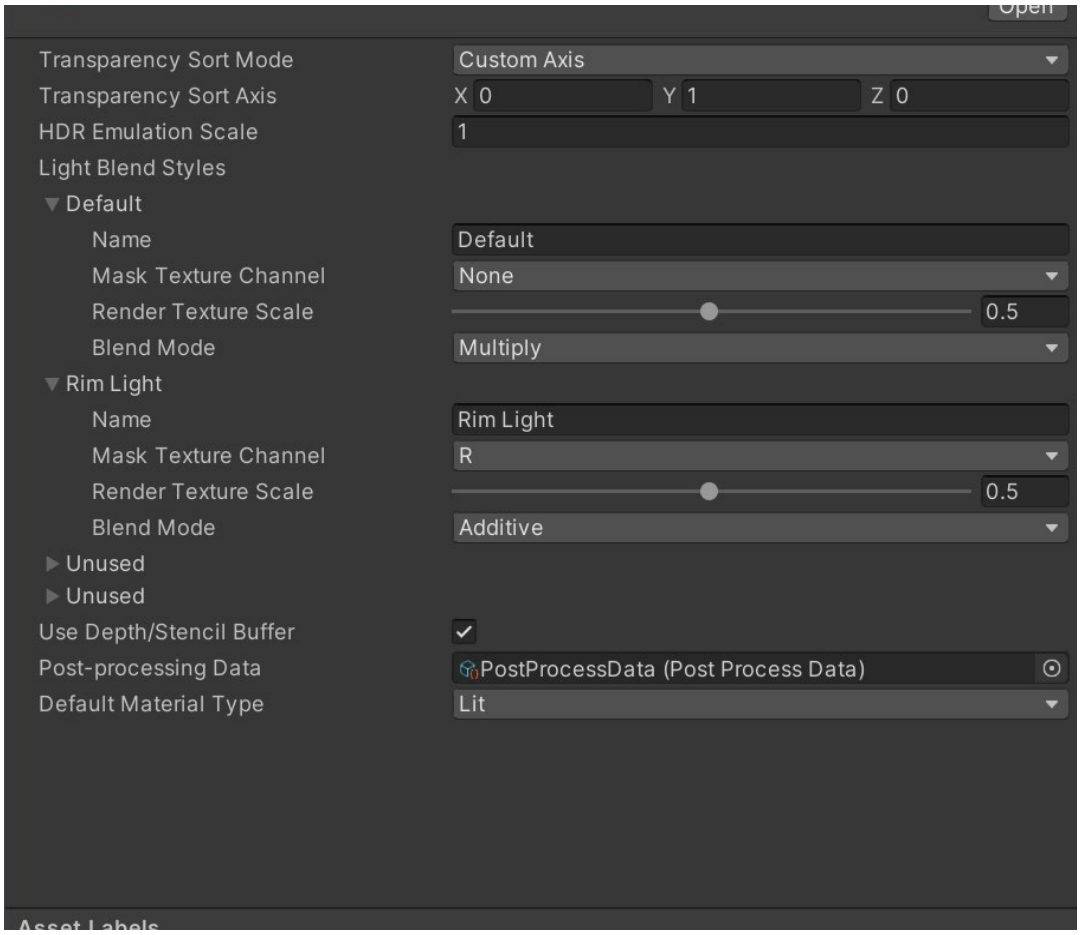

Fresnel Effect
Shallow Angle = strong reflection
Steep Angle = weak reflection
Shallow Angle = strong reflection

Illustration of the Fresnel effect

That's why the water in a lake is more reflective from a distance, but we can see our feet in the water when we look down. This is also why objects are more reflective on their edges. In 2D graphics, you can simulate this effect by using a mask map and a Blend Style.



Transparency Sort Mode	Custom Axis
Transparency Sort Axis	X 0 Y 1 Z 0
HDR Emulation Scale	1
Light Blend Styles	
▼ Default	
Name	Default
Mask Texture Channel	None
Render Texture Scale	0.5
Blend Mode	Multiply
▼ Rim Light	
Name	Rim Light
Mask Texture Channel	R
Render Texture Scale	0.5
Blend Mode	Additive
▶ Unused	
▶ Unused	
Use Depth/Stencil Buffer	✓
Post-processing Data	PostProcessData (Post Process Data)
Default Material Type	Lit

Setting Light Blend Styles options in 2D Renderer Data asset

How to create a Fresnel effect

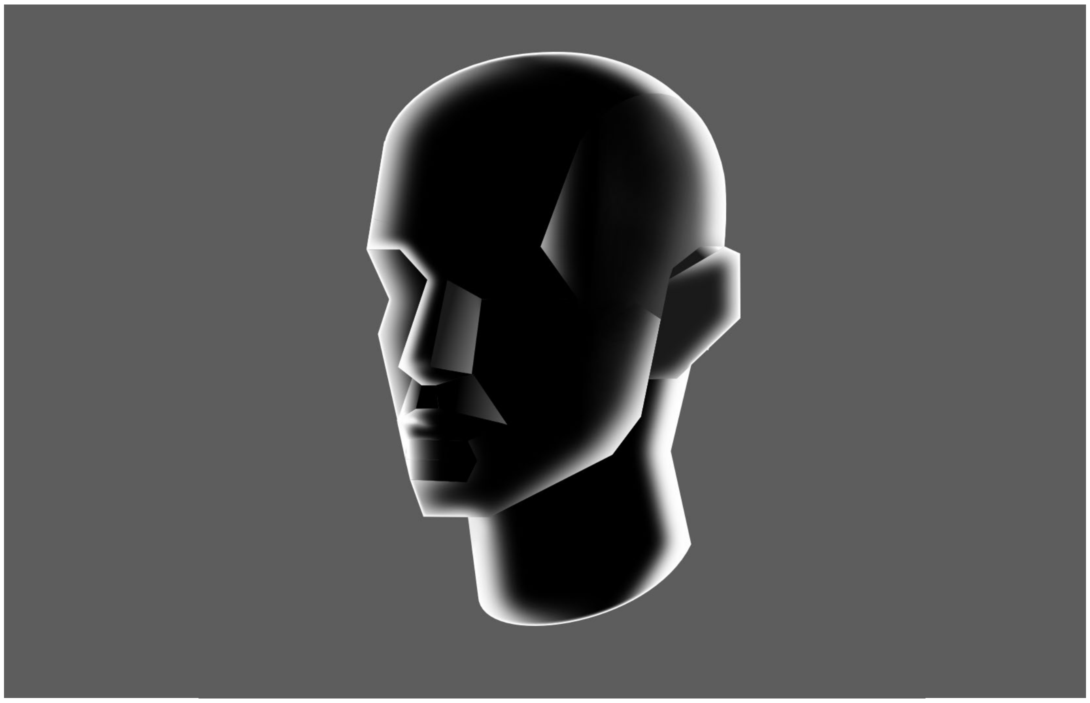
If you downloaded the sample of the human head from above, you can open the sprite in Unity and play around with creating a Fresnel effect by following these steps:

1. In the 2D Renderer Data asset, there are four options for different blending styles under the **Light Blend Styles** option. Leave the first option untouched because it's the default one and work on the second option instead. Give it a name like "Rim Light" or "Fresnel" or whatever works for you.
2. Set the **Mask Texture** channel to R – lights will use the red color of our Mask Map texture.
3. Set the **Blend Mode** to **Additive**. This will make lights be added on top of existing lighting, increasing the brightness.

4. Fresnel light uses only one channel of the mask map (R in our case), but for the sake of simplicity, let's paint the mask map in black and white.

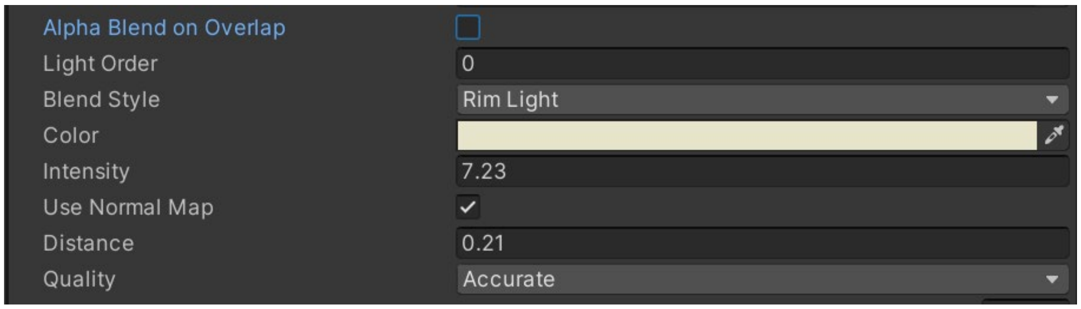
   — The parts that will reflect light will be white, and the unreflective parts will be black. The Fresnel effect impacts the edges of objects, so copy the base sprite, paint it black, then highlight the edges white. It should start to resemble an object that has a bright light shining behind it.

5. To speed things up, add an inner glow effect on the object, and paint some details on top. Unfortunately, there's no app that can speed up this process, so you'll need to rely on your painting skills here.



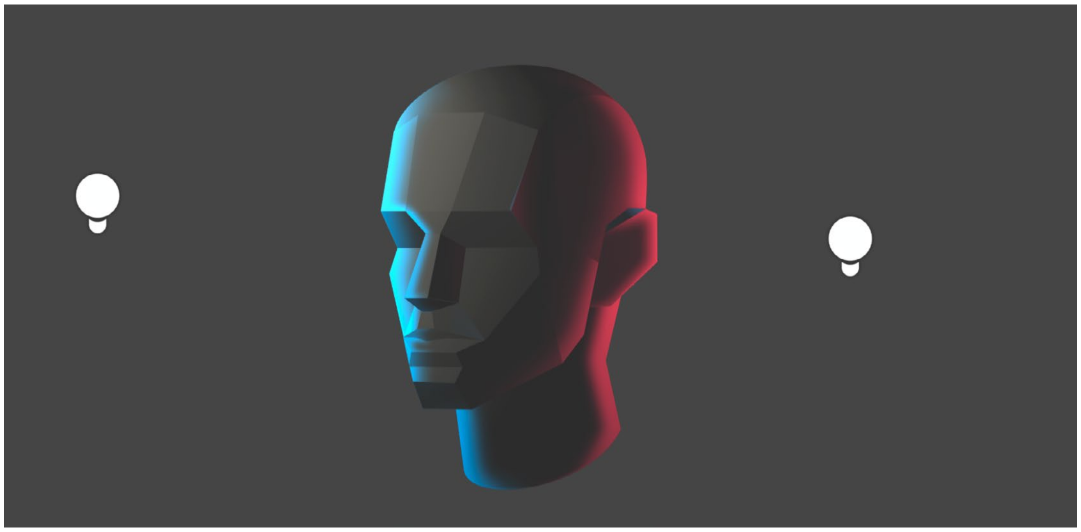
Mask map for a base plane head sprite from the sample

6. To make the light affect the mask map, change its Blend Style to the one created earlier.



Alpha Blend on Overlap	☐
Light Order	0
Blend Style	Rim Light
Color	▭
Intensity	7.23
Use Normal Map	✓
Distance	0.21
Quality	Accurate

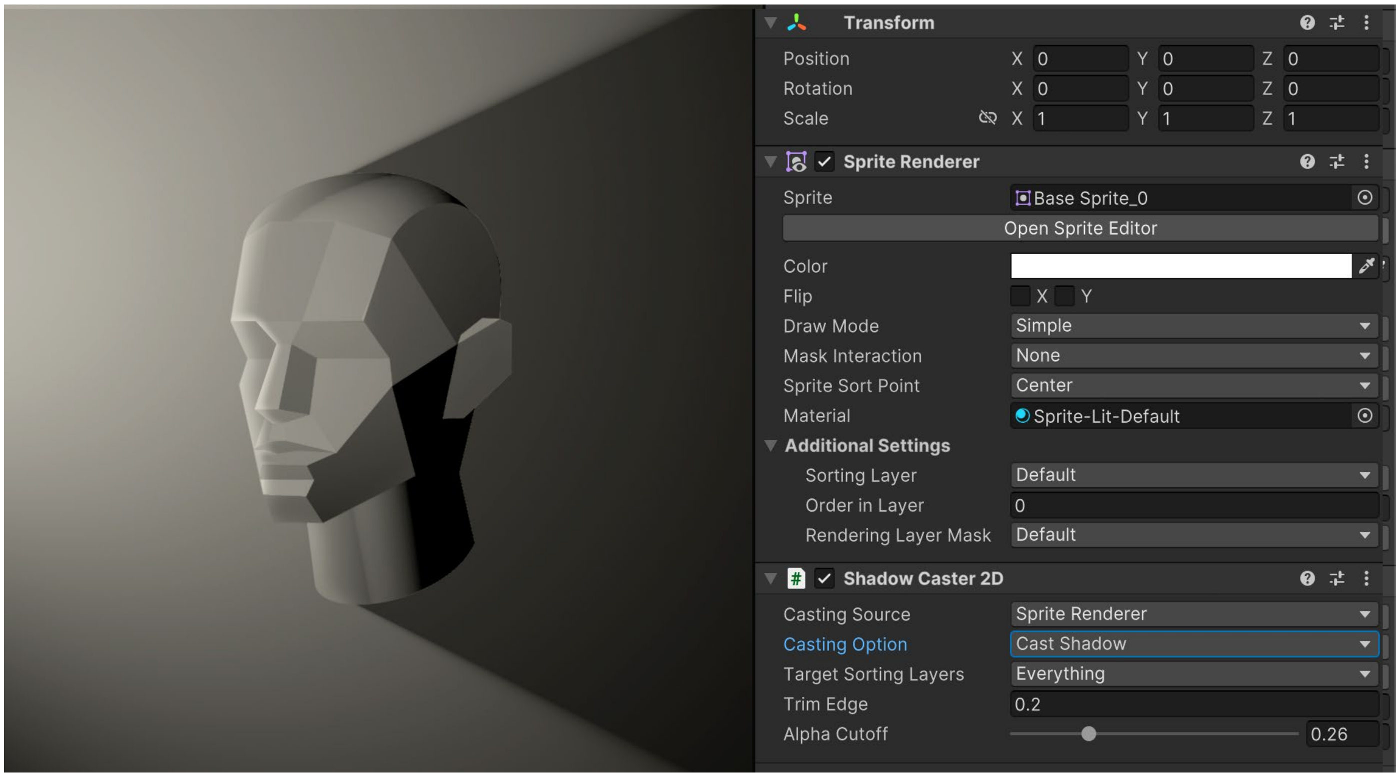
Changing the light's Blend Style to Rim Light


Rim lighting effect on a sprite in the Unity Editor

7. Fresnel lights work best with the Use Normal Maps option enabled and with Distance set to a low value. This prevents highlighting the other side of the object.

2D shadows

For a GameObject to cast a shadow from 2D light, add a Shadow Caster 2D component to the object.


Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Sprite Renderer

Sprite	Base Sprite_0
	Open Sprite Editor
Color	
Flip	X Y
Draw Mode	Simple
Mask Interaction	None
Sprite Sort Point	Center
Material	Sprite-Lit-Default

Additional Settings

Sorting Layer	Default
Order in Layer	0
Rendering Layer Mask	Default

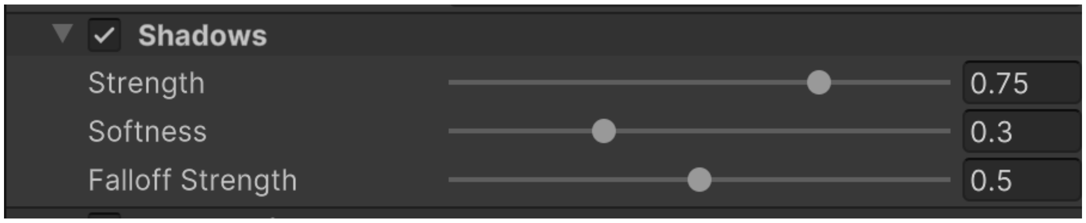
Shadow Caster 2D

Casting Source	Sprite Renderer
Casting Option	Cast Shadow
Target Sorting Layers	Everything
Trim Edge	0.2
Alpha Cutoff	0.26

The Shadow Caster 2D component

By default in Unity 6, the casting source will be the Sprite Renderer, using the auto-generated outline on the Sprite asset. There are additional settings for trimming and alpha cutoff of the outline.

You can also swap to use the Shadow Caster's shape editor to edit its geometry so it fits the object's silhouette or follow any other shape that you create. The light also needs to have some options set to cast a shadow.



Shadows ✓

Strength	0.75
Softness	0.3
Falloff Strength	0.5

Options for a 2D light to cast a shadow

The **Strength** option determines the light's opacity on other objects that are in the shadows. When it's set to 1, the light will not illuminate anything in the shadow area.

The **Softness** option blurs the edges of the shadow based on how it is from the shadow caster, giving the shadow a more natural look, and **Falloff Strength** adjusts the intensity of the edges of the shadow.

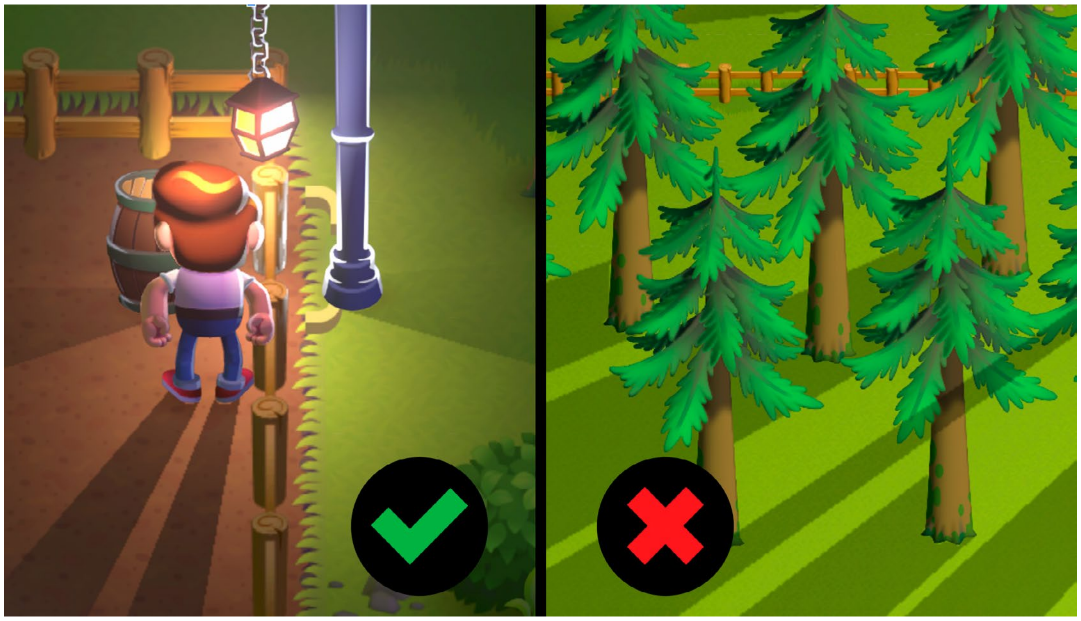
Visual techniques for 2D lights and shadows

In this section, we'll look at creative ways to mimic lighting scenarios in a game; these examples come from the scripts and assets in the *Happy Harvest* project.

Infinite shadows

An object casting shadows produces infinite shadows which are appropriated when the area of light is limited or there's a strong and focused light source like a street lamp or fireplace.

The shadow caster is typically the whole object or character in a sidescrolling game, but in isometric or top-down perspective this effect would look unrealistic. A way to improve the perception of depth is to make shadow casters that conform to the base of the object touching the ground. In the following example from *Happy Harvest,* a Shadow Caster 2D component affects the feet only, by being attached to the feet bones that are found inside the Character GameObject.

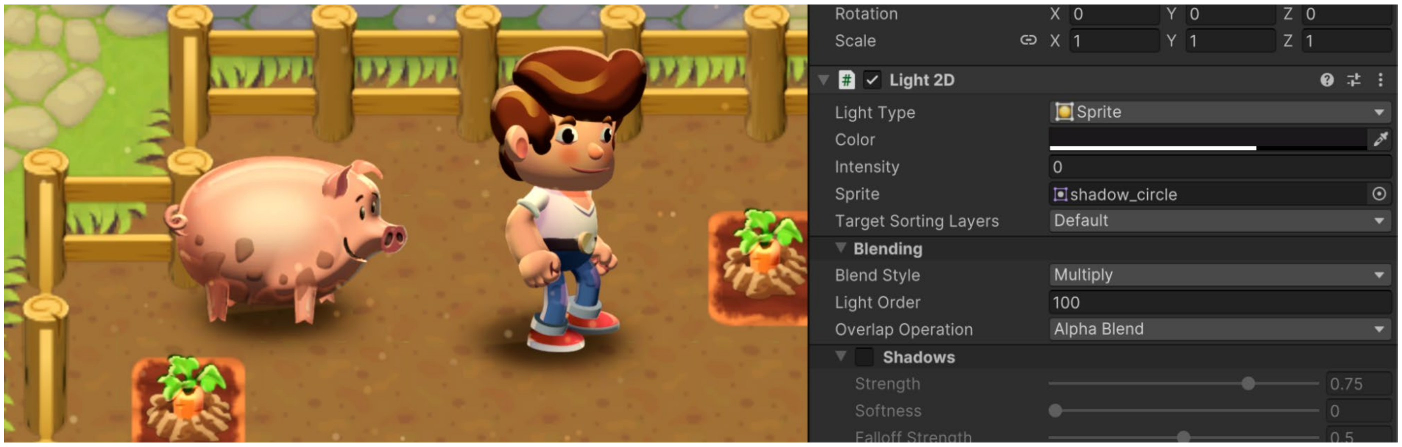

In the left image, the shadow casters work well because they are attached to the feet bones of the character, but in the right image, they don't work correctly for ambient, general lighting.

This technique works well in strong and confined sources of light, but when the light comes from a sunlight source in an open space, the infinite shadow effect looks odd and unnatural, like in the image above of the trees.

Unity's 2D Light system brings flexibility that can be used in creative ways to replicate other types of lighting and shadows.

Blob shadows

An old school technique that can help ground moving characters with the environment is by adding a simple blob shadow sprite as a child object to the shadow casting object. The trick here is to set the **Light Type** to **Sprite** on the Light 2D component and set the **Blend Style** to **Multiply**. If the light is a dark color it will darken the area underneath creating an effect called **negative lighting.**

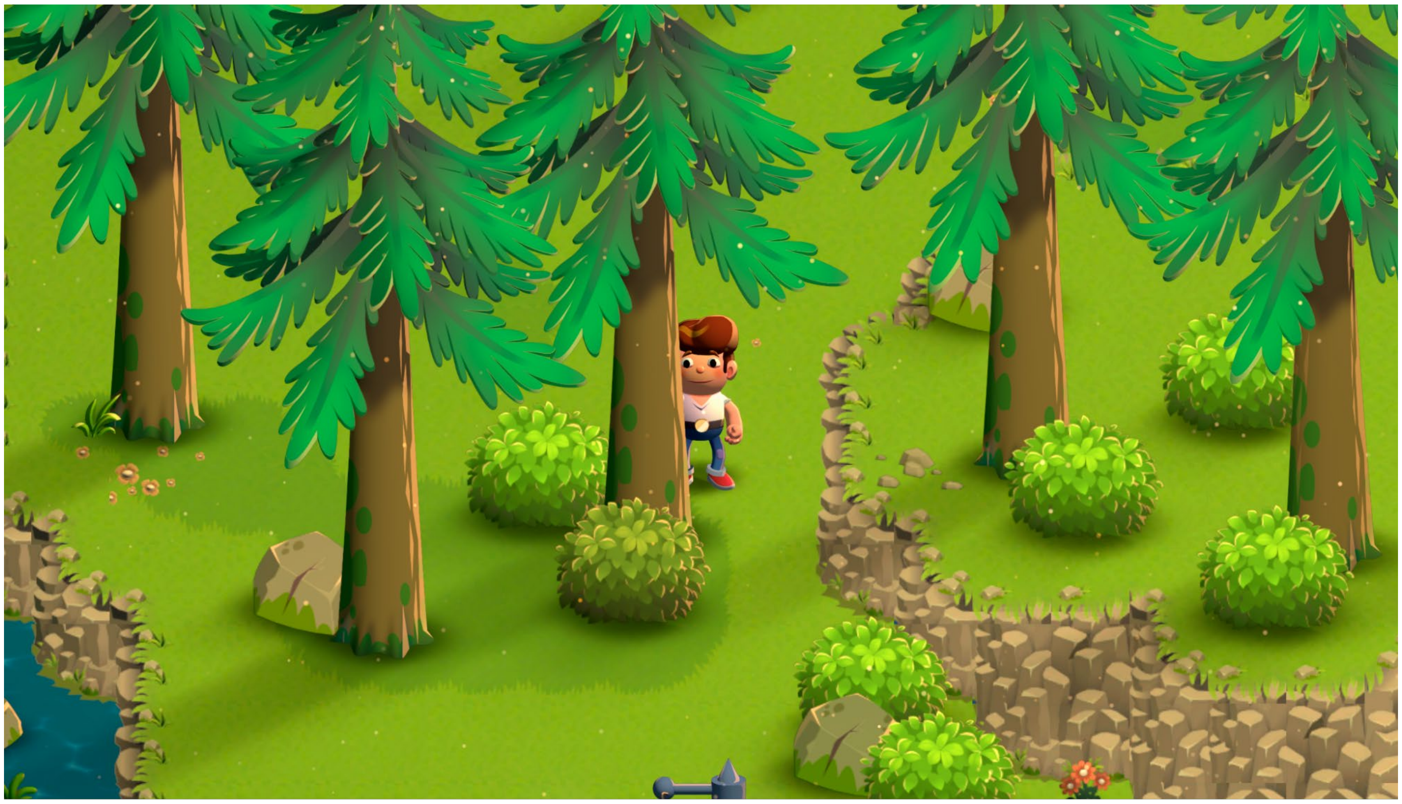

Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Light 2D
Light Type: Sprite
Color
Intensity: 0
Sprite: shadow_circle
Target Sorting Layers: Default
Blending
Blend Style: Multiply
Light Order: 100
Overlap Operation: Alpha Blend
Shadows
Strength: 0.75
Softness: 0
Falloff Strength: 0.5

An inexpensive, but effective, trick for shadows is to use a 2D light as a blob shadow under the characters.

In a top-down game, an endless shadow projection coming from "sunlight" could look strange, as can a standard blob shadow for static objects like trees. In *Happy Harvest* we created a longer blob shadow that rotates and stretches based on the time of the day. The result is a softer shadow that follows the art direction better.

An `UpdateShadow` function in the `DayCycleHandler` script rotates this shadow. This script acts as the "manager", using a function to loop through and update the size and rotation of the shadows.

The elongated blog shadow, like other blob shadows, is a sprite-based light. If you download and open *Happy Harvest*, you can check this by inspecting any child object inside the parent GameObject called **Trees**.


Stretched blob shadows for the trees and bushes in *Happy Harvest*

Day-night cycle

A 2D scene doesn't have a direction light like in 3D scenes. However, you can use a light source that will light up the sprites from the X and Y positions and use their normal or mask maps to enhance the effect. This enables you to create effects like the sun moving as the day progresses, a feature that can be important in top-down and simulation games. It's also worth noting that using large lights comes at a cost if you are targeting low-end platforms.

A large spotlight is used in *Happy Harvest* as the key light. It's attached to the main camera, so it's always on the screen and rotates with a script simulating the movement of the sun.

A child GameObject in the project named **LightsRotator** has four lights attached:

— **NightLight**: Simulates the moon light direction

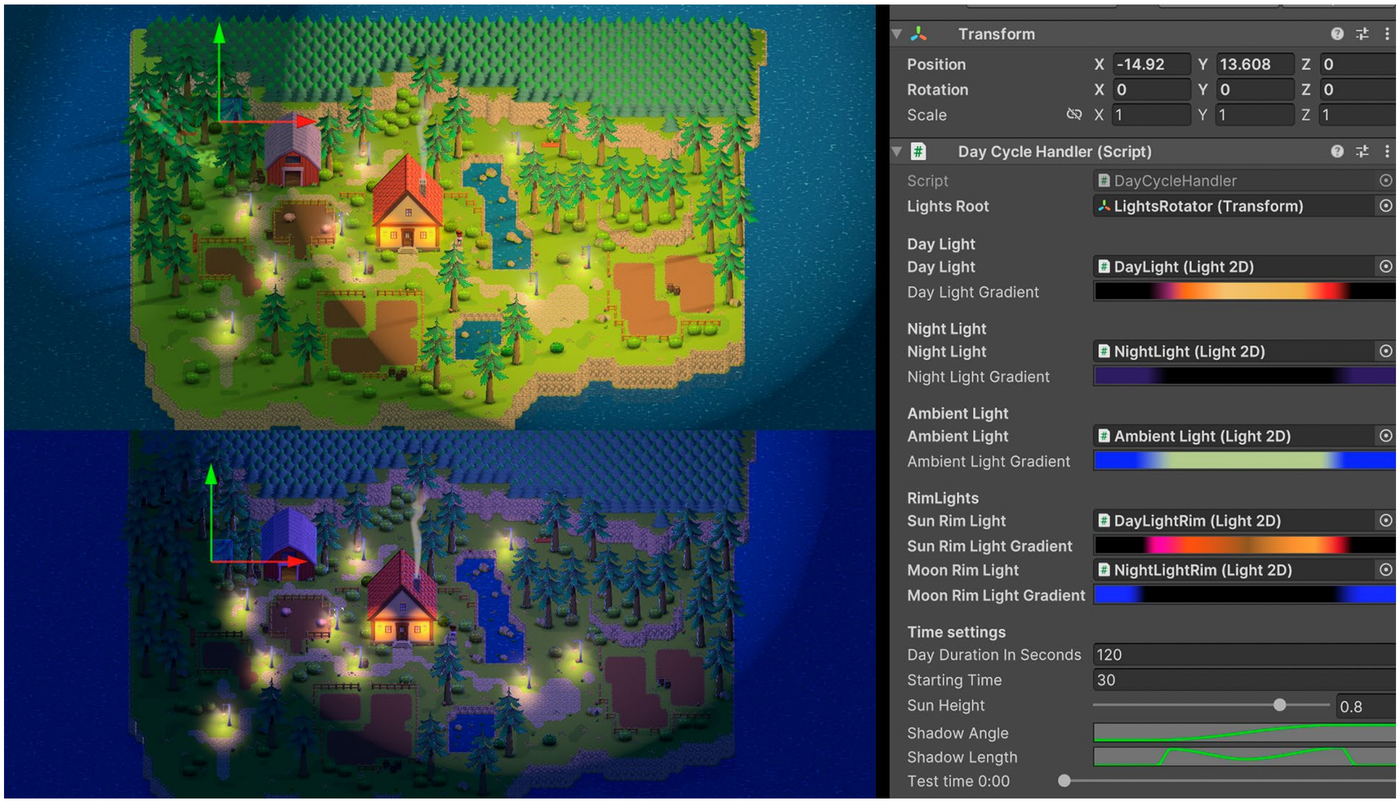
— **DayLight**: Simulates the sun light direction (opposite position to the NightLight)

— **NightLightRim:** Similar to the NightLight moon direction light but only for the character's and props rim lights

— **DayLightRim:** Similar to the DayLight sunlight direction light but only for the character's and props rim lights

The script that controls the movement of these lights also controls the change of color throughout the day. Gradients were used to set the color for each light at a particular time. You can see these gradients in the **DayCycleHandle** script attached to the **DayCycleHandler** GameObject.

There are additional parameters to control the finite shadow effects, **Shadow Angle** and **Shadow Length.** In each of those fields, the Animation curve indicates the clockwise angle of the shadows through the day. The length parameter defines the length of the shadows in a given moment, for example, a longer shadow might be needed when the sun is setting and a shorter one when the "sun" would be illuminating perpendicularly the scene. Note that you might need to move the Test Time slider to refresh the Shadow Angle and Shadow Length settings.



Transform
Position X -14.92 Y 13.608 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Day Cycle Handler (Script)
Script: DayCycleHandler
Lights Root: LightsRotator (Transform)

Day Light
Day Light: DayLight (Light 2D)
Day Light Gradient

Night Light
Night Light: NightLight (Light 2D)
Night Light Gradient

Ambient Light
Ambient Light: Ambient Light (Light 2D)
Ambient Light Gradient

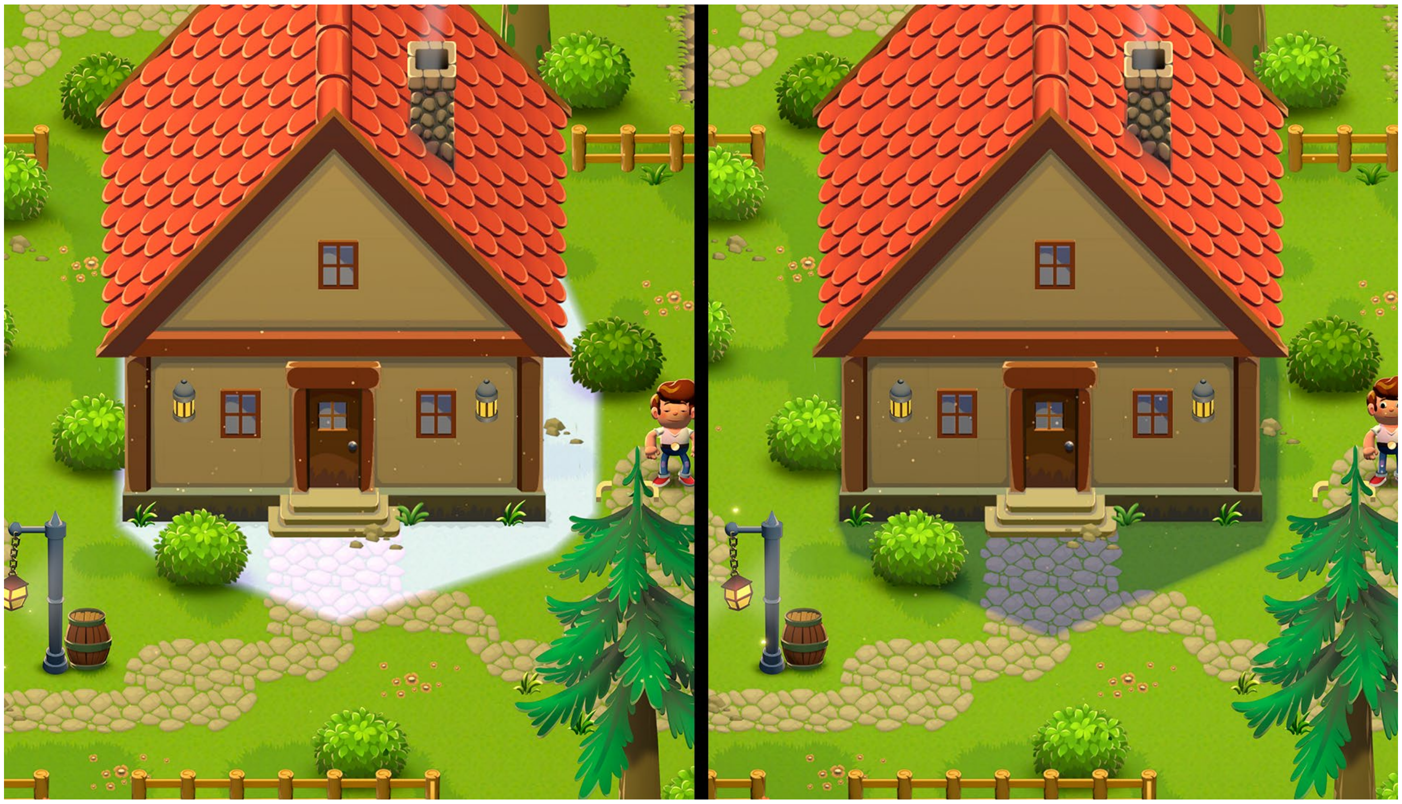
RimLights
Sun Rim Light: DayLightRim (Light 2D)
Sun Rim Light Gradient
Moon Rim Light: NightLightRim (Light 2D)
Moon Rim Light Gradient

Time settings
Day Duration In Seconds: 120
Starting Time: 30
Sun Height: 0.8
Shadow Angle
Shadow Length
Test time 0:00

The large spot lights in Happy Harvest, one for the sun one for the moon

Manipulating freeform lights

In a top-down game, a big tall building needs to cast a precise-looking shadow with sharp edges. In *Happy Harvest*, Freeform lights were used to create the buildings' shadows. They mimic the projection that the building would produce on the ground, an approximation that's necessary to make since there's no depth information to work with in 2D.



The freeform lights used to create the building shadow use the negative lighting technique, which consists of using the Multiply blend mode and a darker color

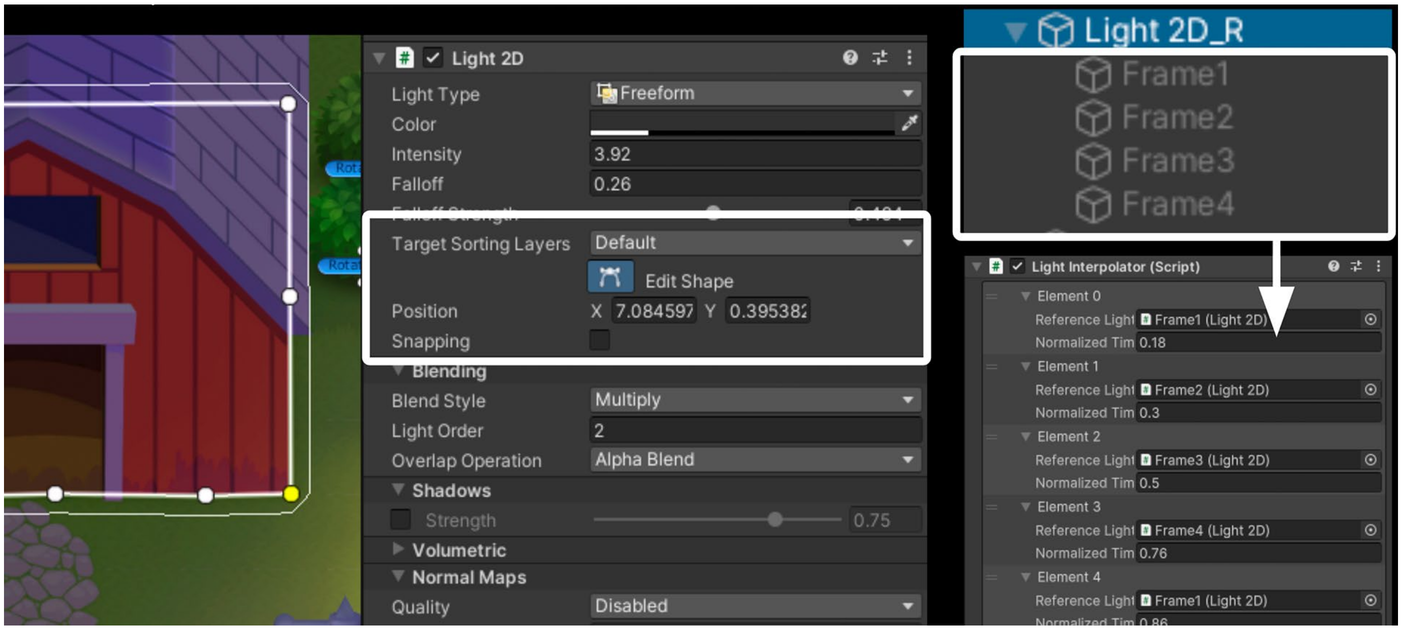
The challenge with well-defined shadows in the sample is how to make them work with the day-to-night cycle. In order to make the shadows react to the different positions of the "sun", a `Light Interpolator` script was created to tween the vector points of the Freeform light between different reference shadows.

In the Hierarchy window of the sample is a GameObject named **Light_2D_Warehouse.** Attached to it are four Freeform lights, each one mimicking the shadow that the building would project when the sun is up, right, down, and to the left of the building. This script creates a smooth interpolation moving the different vector points using the API.

A more defined shadow under the warehouse that moves throughout the day

The top shadow is created first and then modified to create the bottom shadow and the ones to each side of the building. It's important to ensure that each shadow has the same number of points and the transition between those points is considered when each shadow is created.

Once the shadows were created, they were added to the `Light Interpolator` component script, with a Normalized parameter that indicates the weight in time that each shadow has during the day time. The **Preview Time** feature in the script allows you to previsualize how they will look like in the Editor.


Light 2D

Light Type	Freeform
Color	
Intensity	3.92
Falloff	0.26
Target Sorting Layers	Default
	Edit Shape
Position	X 7.084597 Y 0.395382
Snapping	

Blending

Blend Style	Multiply
Light Order	2
Overlap Operation	Alpha Blend

Shadows

Strength	0.75

Volumetric

Normal Maps

Quality	Disabled

Light 2D_R
Frame1
Frame2
Frame3
Frame4

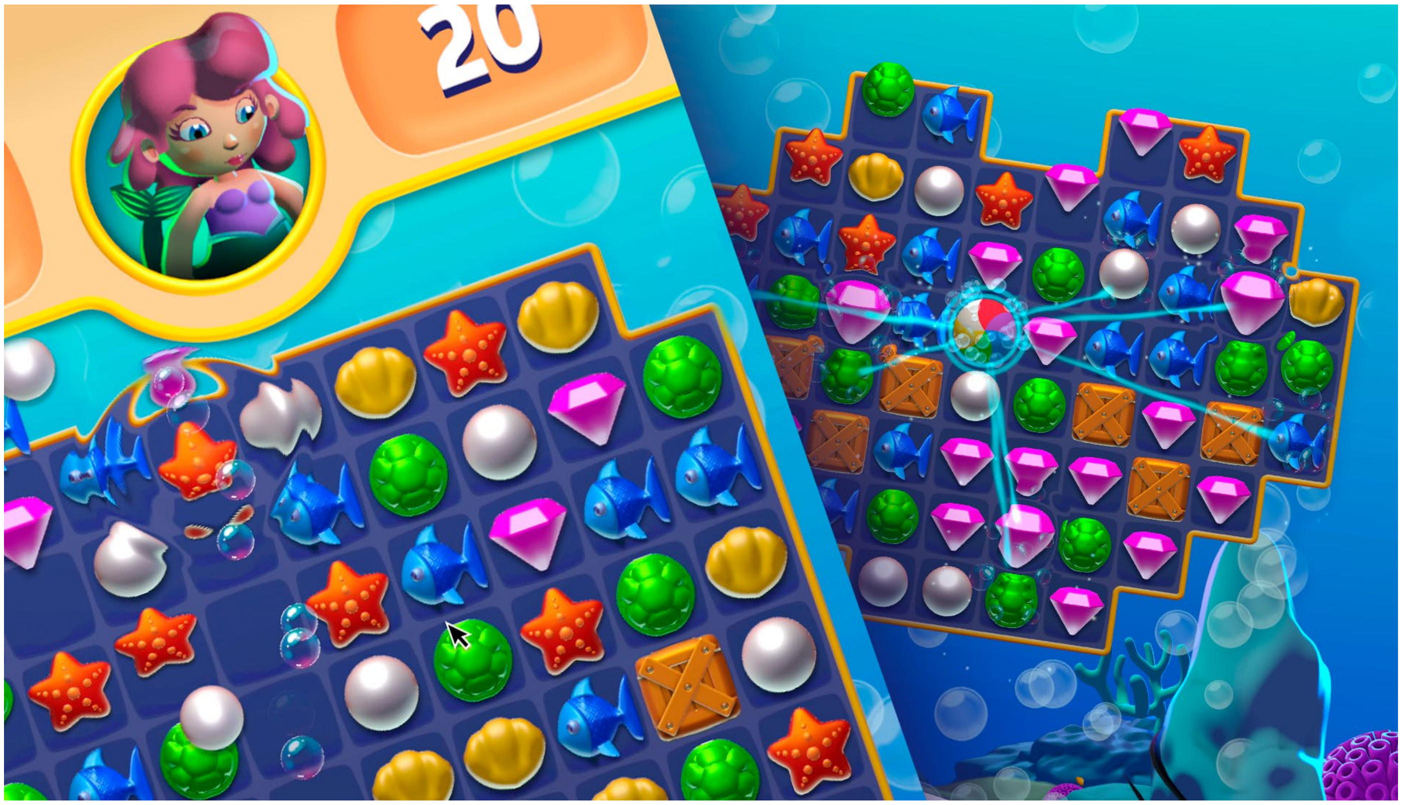
Light Interpolator (Script)

Element 0	
Reference Light	Frame1 (Light 2D)
Normalized Tim	0.18
Element 1	
Reference Light	Frame2 (Light 2D)
Normalized Tim	0.3
Element 2	
Reference Light	Frame3 (Light 2D)
Normalized Tim	0.5
Element 3	
Reference Light	Frame4 (Light 2D)
Normalized Tim	0.76
Element 4	
Reference Light	Frame1 (Light 2D)
Normalized Tim	0.86

An interpolator script used in the sample tweens the positions of the vectors of a Freeform light.

Sprite Custom Lit shader

A standard use of the 2D light system is to apply it so objects light each other or the environment, but how can you create your own lighting system? The **Sprite Custom Lit** shader, available in Shader Graph, allows you to read the light maps generated by the 2D light system and apply any effects you need to these maps to use in your sprites.

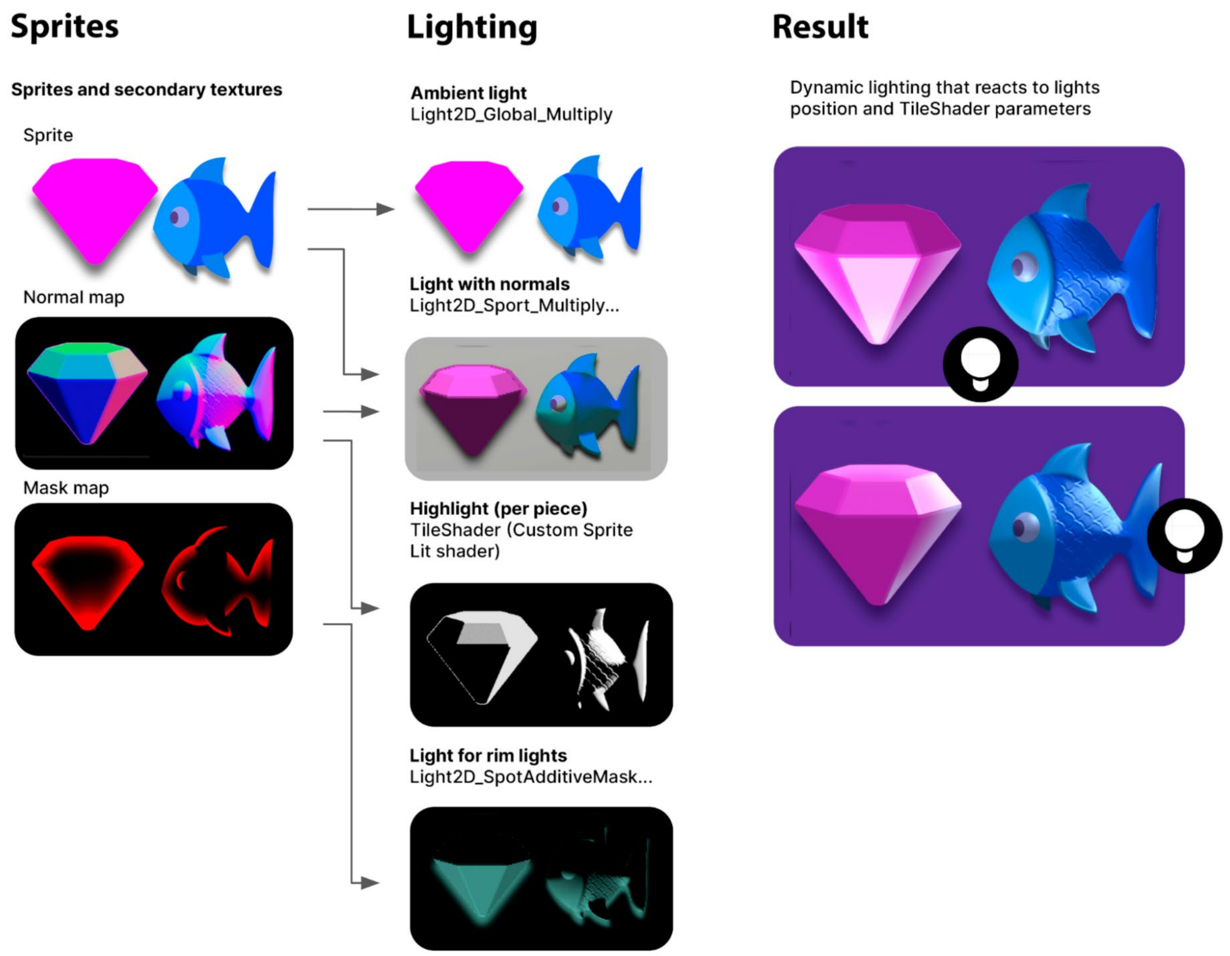


The sample Gem Hunter Match uses a Sprite Custom Lit shader to illuminate efficiently hundreds of pieces

A great example of this is in the Gem Hunter Match sample, where each piece in the game board is lit precisely. We achieved this by by combining:

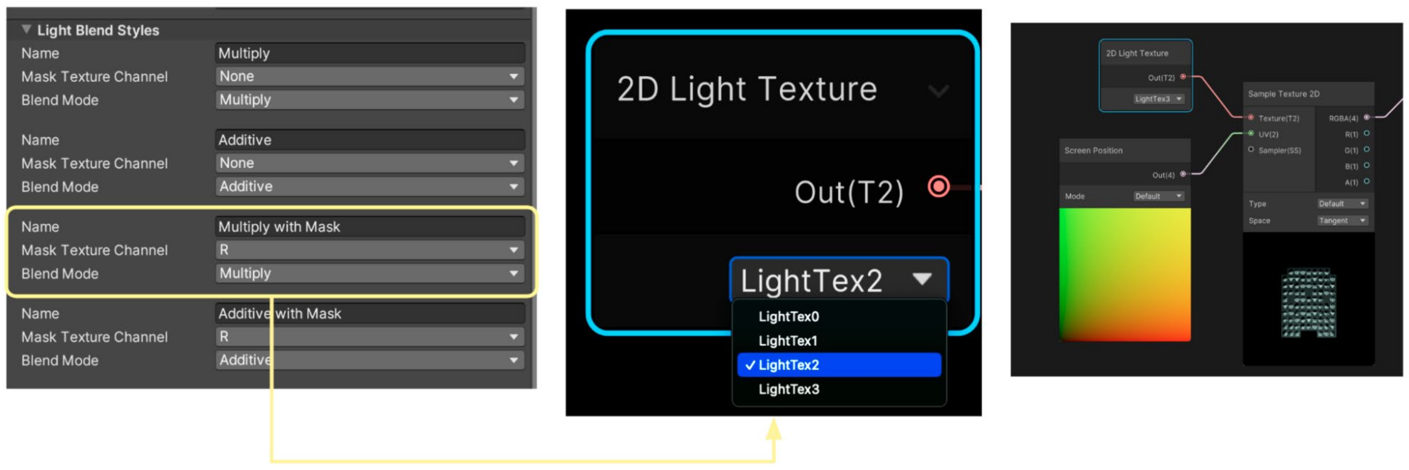
- Custom illumination per piece, using the normal map and a fictional position of the light source; with the Dot Product node, we could calculate how much light a pixel should receive

- Blending in light information in the 2D Light Texture: A spot light used for the rim light or fresnel effect together with another spotlight for a world space source of light


Sprites
Sprites and secondary textures
Sprite
Normal map
Mask map
Lighting
Ambient light
Light2D_Global_Multiply
Light with normals
Light2D_Sport_Multiply...
Highlight (per piece)
TileShader (Custom Sprite Lit shader)
Light for rim lights
Light2D_SpotAdditiveMask...
Result
Dynamic lighting that reacts to lights position and TileShader parameters

Sprite preparation: This image captures how the board piece sprites in Gem Hunter Match were prepared for dynamic lighting effects with normal maps and mask maps.

Take a closer look at each of the 2D Light Textures corresponding to each Blending Style from the 2D Light Texture node and use it to apply your custom 2D lighting on the object with this shader.



Light Blend Styles

Name	Multiply
Mask Texture Channel	None
Blend Mode	Multiply
Name	Additive
Mask Texture Channel	None
Blend Mode	Additive
Name	Multiply with Mask
Mask Texture Channel	R
Blend Mode	Multiply
Name	Additive with Mask
Mask Texture Channel	R
Blend Mode	Additive

Find out more about custom sprite lighting in Gem Hunter Match and other creative ideas in the following videos:

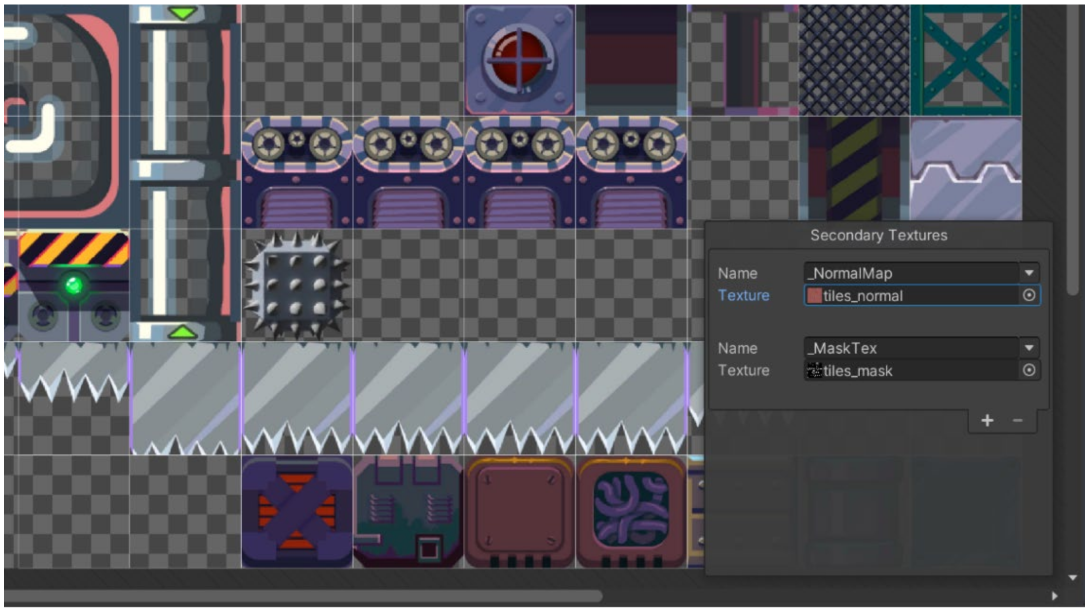
Part 1 and Part 2

Using 2D lights on 2D renderers

2D lights can be used together with normal maps and mask maps on sprites, but you can also use them on different renderers. Let's see how.

2D tilemaps

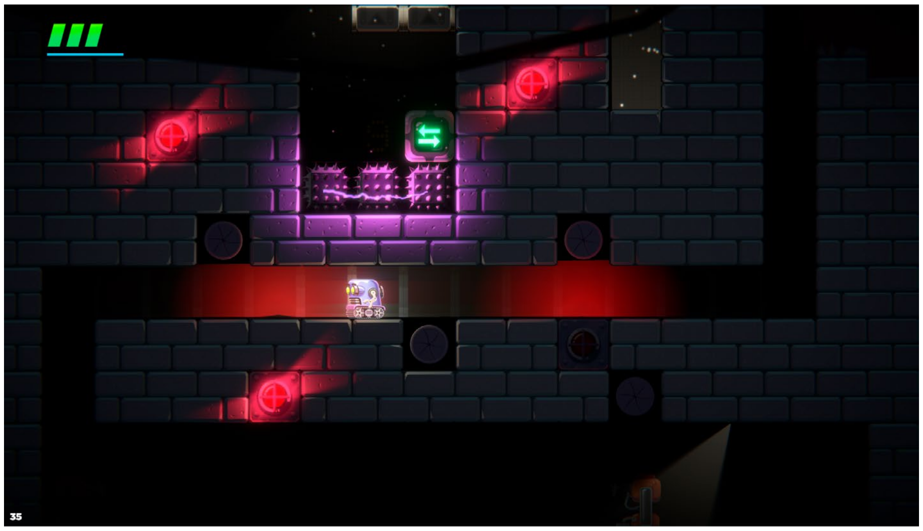
To use 2D lights on tilemaps, assign the normal map and mask map textures to the sprites that are used as tiles by using the Sprite Editor's Secondary Textures module. Secondary textures will be used automatically by the 2D lights system.



Name _NormalMap
Texture tiles_normal

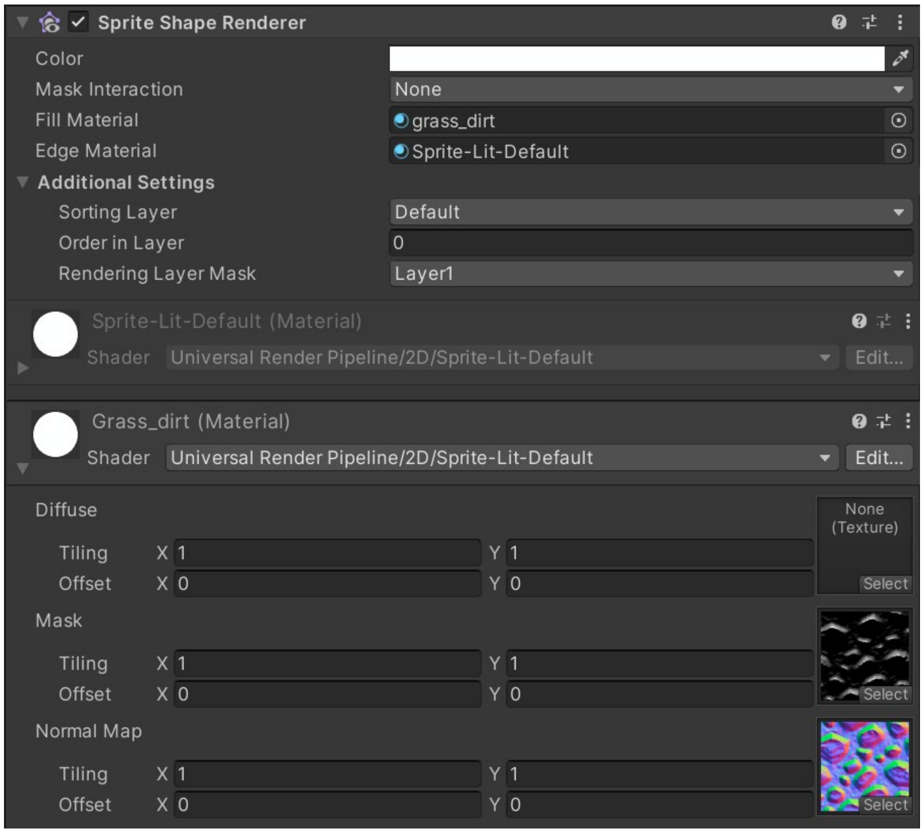
Name _MaskTex
Texture tiles_mask

Tilemaps with secondary maps


2D lights with normal mapping and mask maps on a tilemap

2D Sprite Shape

The Sprite Shape Renderer uses two different materials per object: one assigned to the **Fill Material** field and the other assigned to the **Edge Material** field. The Edge Material uses secondary maps from the sprite asset. For the Fill Material, use the texture with **Wrap** mode set to **Repeat** for tiling instead of the sprite. You'll need to create a new Fill Material with **Material Property Block** enabled for setting secondary textures.


Sprite Shape Renderer

Color	
Mask Interaction	None
Fill Material	grass_dirt
Edge Material	Sprite-Lit-Default
Additional Settings	
Sorting Layer	Default
Order in Layer	0
Rendering Layer Mask	Layer1

Sprite-Lit-Default (Material)
Shader Universal Render Pipeline/2D/Sprite-Lit-Default Edit...

Grass_dirt (Material)
Shader Universal Render Pipeline/2D/Sprite-Lit-Default Edit...

		X	Y	Texture
Diffuse				None (Texture)
	Tiling	1	1	
	Offset	0	0	
Mask				
	Tiling	1	1	
	Offset	0	0	
Normal Map				
	Tiling	1	1	
	Offset	0	0	

Example of Sprite Shape GameObjects using normal maps

2D animated characters with secondary textures

Setting secondary textures in the 2D PSD Importer works similarly, with one minor difference regarding normal maps.

The fastest way to make a normal map and mask map for a 2D PSD imported character is by working from a base character .PSD file.

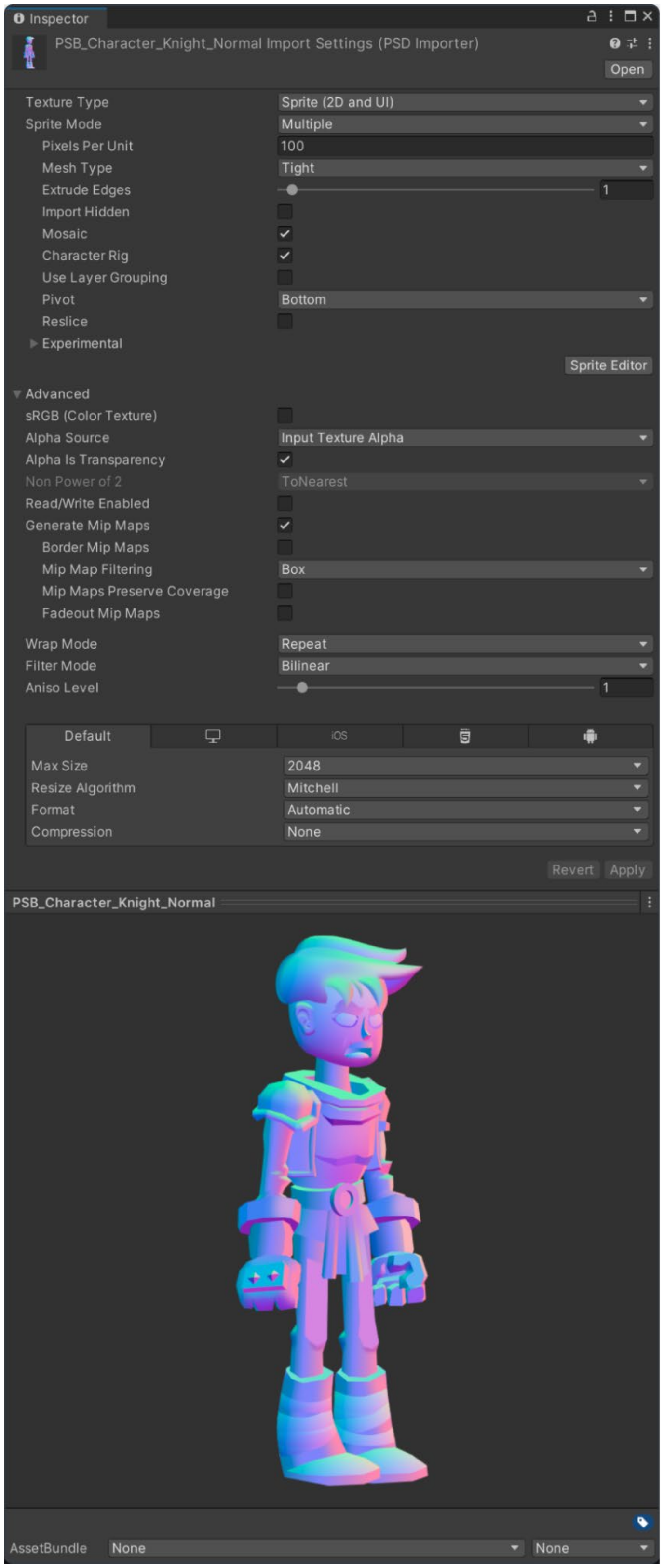
Complete these steps to add a normal map to an animated character:

1. Duplicate the base character .PSD file. Rename the duplicated file by adding a suffix to its name, for example: _normal

2. Open this file in your preferred image editor, and paint a normal map onto each layer. Save the file.

3. When a PSD with a normal map is imported into Unity, you'll need to set the Texture Type to Sprite and go into Advanced settings to uncheck sRGB(Color Texture) option. A normal map doesn't contain color sRGB data, only angle values.

4. Assign this PSD file as a secondary texture of your base character. To make a mask map, repeat the process, give the duplicated file another suffix, and just skip step 3.



PSB_Character_Knight_Normal Import Settings (PSD Importer)

Texture Type	Sprite (2D and UI)
Sprite Mode	Multiple
Pixels Per Unit	100
Mesh Type	Tight
Extrude Edges	1
Import Hidden	
Mosaic	✓
Character Rig	✓
Use Layer Grouping	
Pivot	Bottom
Reslice	
Experimental	
Advanced	
sRGB (Color Texture)	
Alpha Source	Input Texture Alpha
Alpha Is Transparency	✓
Non Power of 2	ToNearest
Read/Write Enabled	
Generate Mip Maps	✓
Border Mip Maps	
Mip Map Filtering	Box
Mip Maps Preserve Coverage	
Fadeout Mip Maps	
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1
Max Size	2048
Resize Algorithm	Mitchell
Format	Automatic
Compression	None
AssetBundle	None / None

Adding a normal map to an animated character

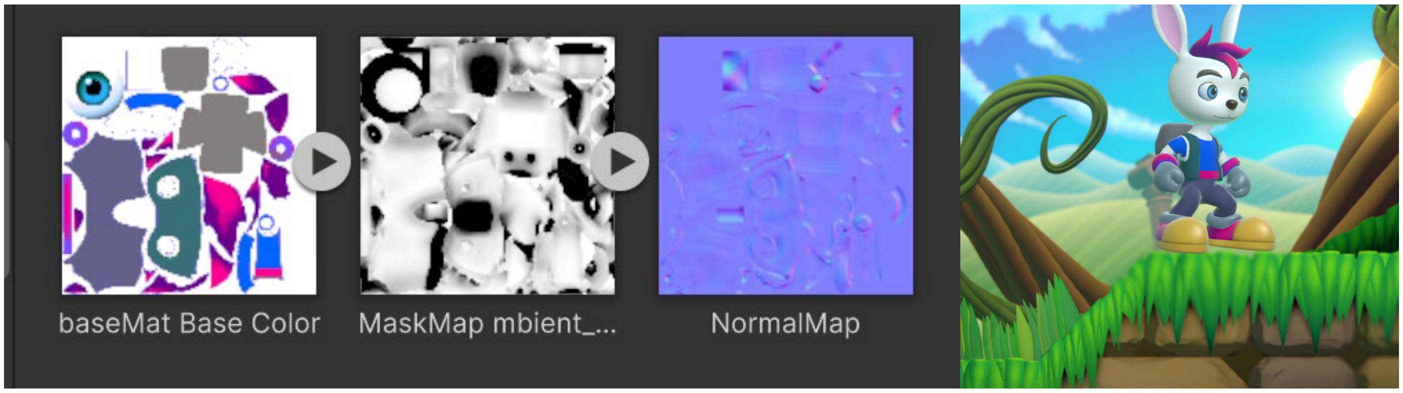
Render 3D as 2D

Many made-with-Unity games mix 2D and 3D elements in their gameplay and environments. This can entail building a system to represent 2D graphics in a 3D project or bringing 3D to 2D with custom rendering. Teams can mix 2D and 3D to serve the artistic themes, production pipeline, and performance goals of their games. A few specific use cases include:

- Animated characters that are shown from different angles can be easier to create in 3D, for example, in isometric games.
- Repurposing of animations, accessories, or customization might also be easier with 3D characters.
- To increase perception of depth, you can create 3D backgrounds with 2D characters.

Unity 6.3 LTS provides a new workflow to bring 3D assets into your 2D project with benefits like:

- All the 2D tooling at your disposal
- A more unified look using the 2D light system and shader material across 3D and 2D assets
- Natural integration of 3D for developers used to 2D workflows
- Easy sorting between sprites, tilemaps, sprite shapes, and 3D assets



The textures used for the 3D character in Bunny Blitz are the same as the secondary textures you can add to a 2D sprite.

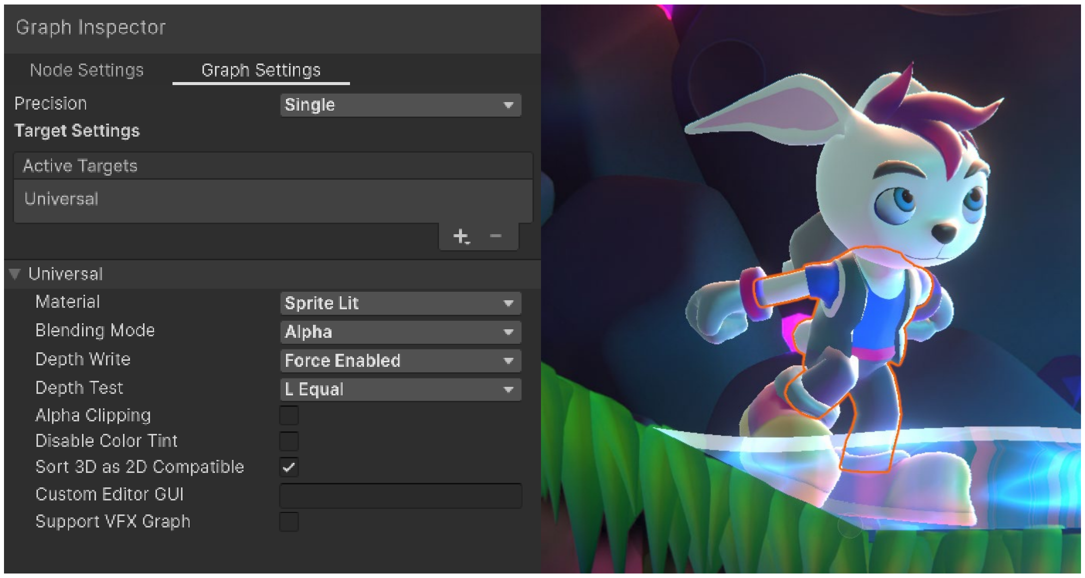
Compatibility

MeshRenderer and the SkinnedMeshRenderer are compatible 3D renderers that can be used in your 2D project with the URP's 2D Renderer.

Compatible Materials

The **Mesh2D-Lit-Default** and **Mesh2D-Unlit-Default** are included and ensure your 3D assets are lit by 2D lights. Textures you can assign to these materials are the same as those for 2D sprites: main texture, normal map, and mask map.

Additionally, Shader Graph sprite materials with the setting **Sort 3D as 2D** enabled can be used in 3D assets and also receive the 2D lighting.



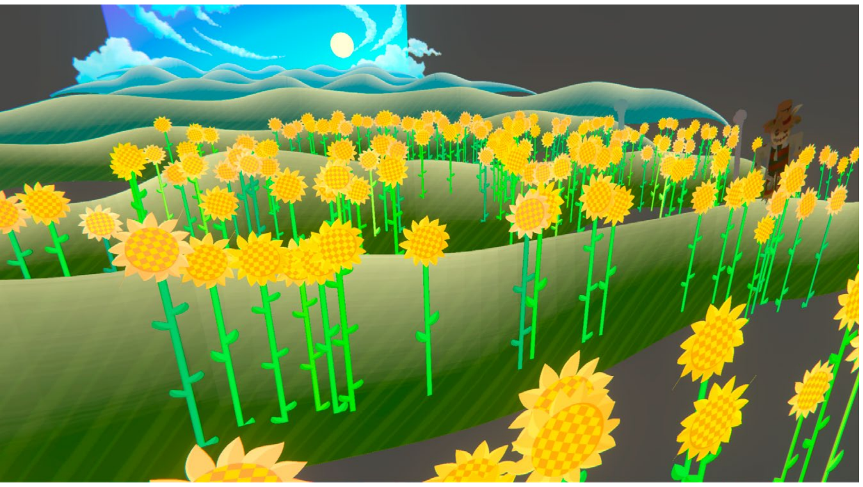
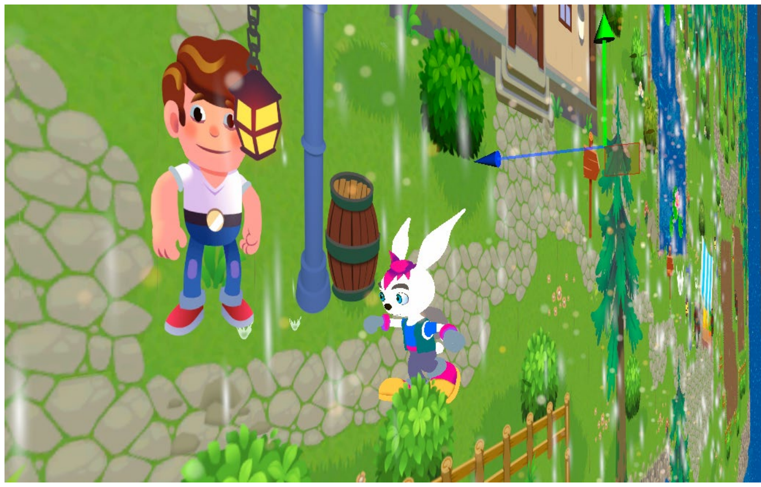
Precision	Single
Material	Sprite Lit
Blending Mode	Alpha
Depth Write	Force Enabled
Depth Test	L Equal
Alpha Clipping	
Disable Color Tint	
Sort 3D as 2D Compatible	✓
Custom Editor GUI	
Support VFX Graph	

A Sprite Lit shader with the Sort 3D as 2D mode enabled

Sorting 3D and 2D assets

You have two ways to plan for the sorting of objects in your project:

1. Do you want to arrange elements in the Z-axis and make use of the perspective effect?
   - Just place 3D and 2D objects in different planes and will be sorted correctly.
   - **Note** that if you want those 3D objects to receive 2D lights, they will need to have a **Sorting Group** component attached to indicate which sorting layer lights they will receive lights from.
   - One benefit of using this technique is that by writing in the depth buffer post-processing effects like Depth of Field will be available.
2. Do you want to arrange objects using the sorting layer system from 2D projects?
   - Bring 3D objects to your 2D world and add the Sorting Group component to them. If you check the **3D as 2D mode**, they will get flattened to avoid clipping and behave just like another sprite.

The image on the left is from a 2D scene with elements arranged on the Z-axis, while on the right is an image from the Happy Harvest sample, which is using sorting layers to sort the 3D rabbit character from Bunny Blitz so it blends in between the sprites.

Masking 3D and 2D

Sprite mask is compatible with both 3D and 2D, with any object able to be masked by any 2D renderer.

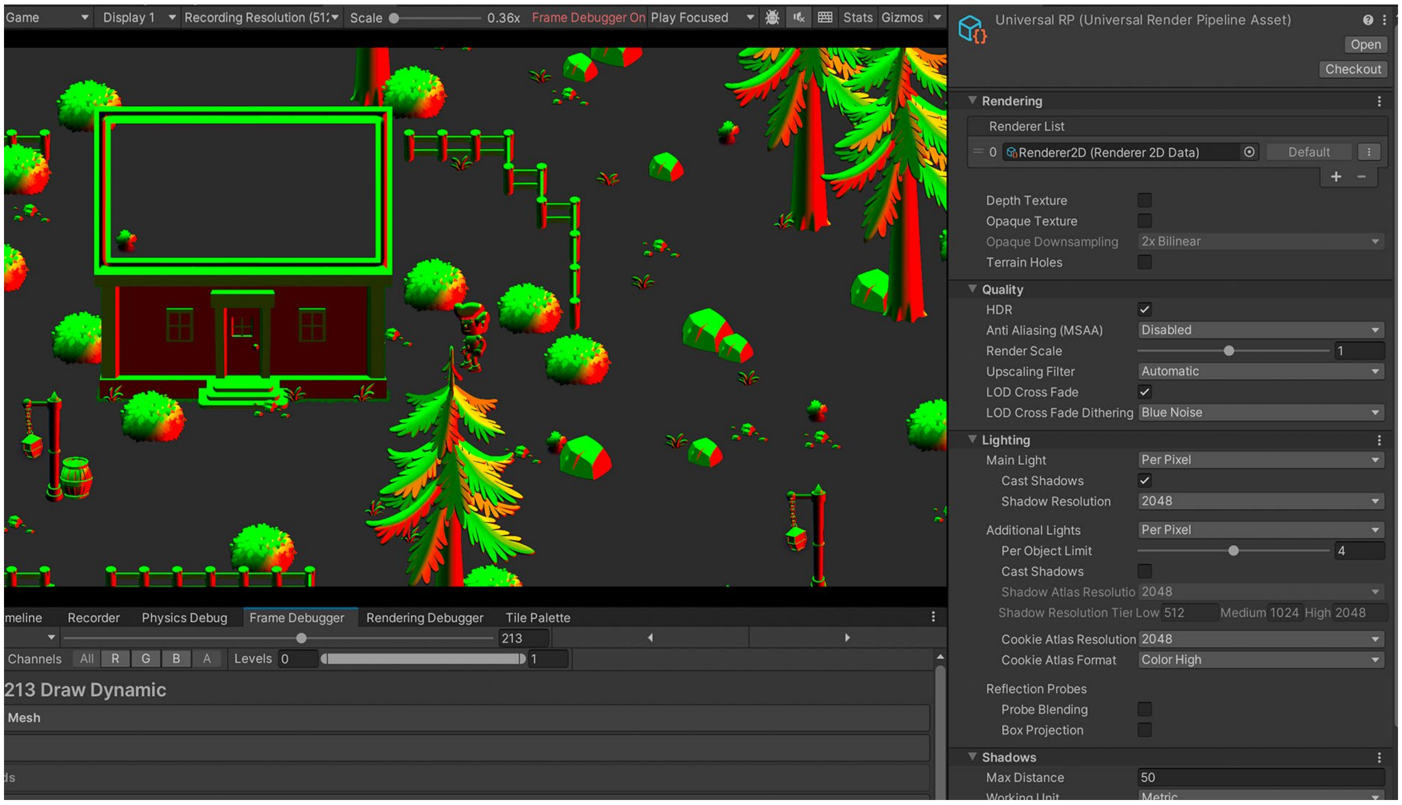
Look out for the **Mask Interaction** option inside the Mesh Renderer or Skinned Mesh Renderer, as well as inside any of the 2D renderers, including 2D Tilemap, Sprite Shape or sprite renderers. They can only be visible inside a mask, outside a mask, or on any 2D renderer acting as the mask.

You can change the mask interaction of any 3D or 2D object at runtime by using the new API in Unity 6.3 LTS.

The rabbit gets masked inside the oval sprite of the portal and becomes only visible inside the sprite.

You can learn more about the new 2D/3D workflow from the Unite Barcelona 2025 keynote and 2D session.

2D optimization tips



213 Draw Dynamic

Mesh

Universal RP (Universal Render Pipeline Asset)

Rendering

Renderer List

0 Renderer2D (Renderer 2D Data) Default

Depth Texture	
Opaque Texture	
Opaque Downsampling	2x Bilinear
Terrain Holes	

Quality

HDR	✓
Anti Aliasing (MSAA)	Disabled
Render Scale	1
Upscaling Filter	Automatic
LOD Cross Fade	✓
LOD Cross Fade Dithering	Blue Noise

Lighting

Main Light	Per Pixel
Cast Shadows	✓
Shadow Resolution	2048
Additional Lights	Per Pixel
Per Object Limit	4
Cast Shadows	
Shadow Atlas Resolutio	2048
Shadow Resolution Tier	Low 512 Medium 1024 High 2048
Cookie Atlas Resolution	2048
Cookie Atlas Format	Color High
Reflection Probes	
Probe Blending	
Box Projection	

Shadows

Max Distance	50
Working Unit	Metric

The 2D Renderer settings in Happy Harvest as seen in the Frame Debugger; observe what's happening at each step of rendering frame

General optimizations for 2D lights

A common concern of using 2D lighting, especially if you target mobile platforms, is the cost of adding lights in the game. Always test on the actual target hardware, including the lowest specs supported. And try these general optimization tips to help boost the performance of your project:

— Keep the fill rate as low as possible. One large light can have worse performance than several small lights.

— Lights perform best when batchable. Lights with the same lighting setup across contiguous layers can all be drawn together.

— Keep your render scale as low as possible. Render scale adjusts the texture size used when rendering lighting, and a lower texture size means fewer pixels to be rendered.

— Minimize the number of shadow casting lights on screen. There is a performance cost when switching to draw shadows that is non-trivial.

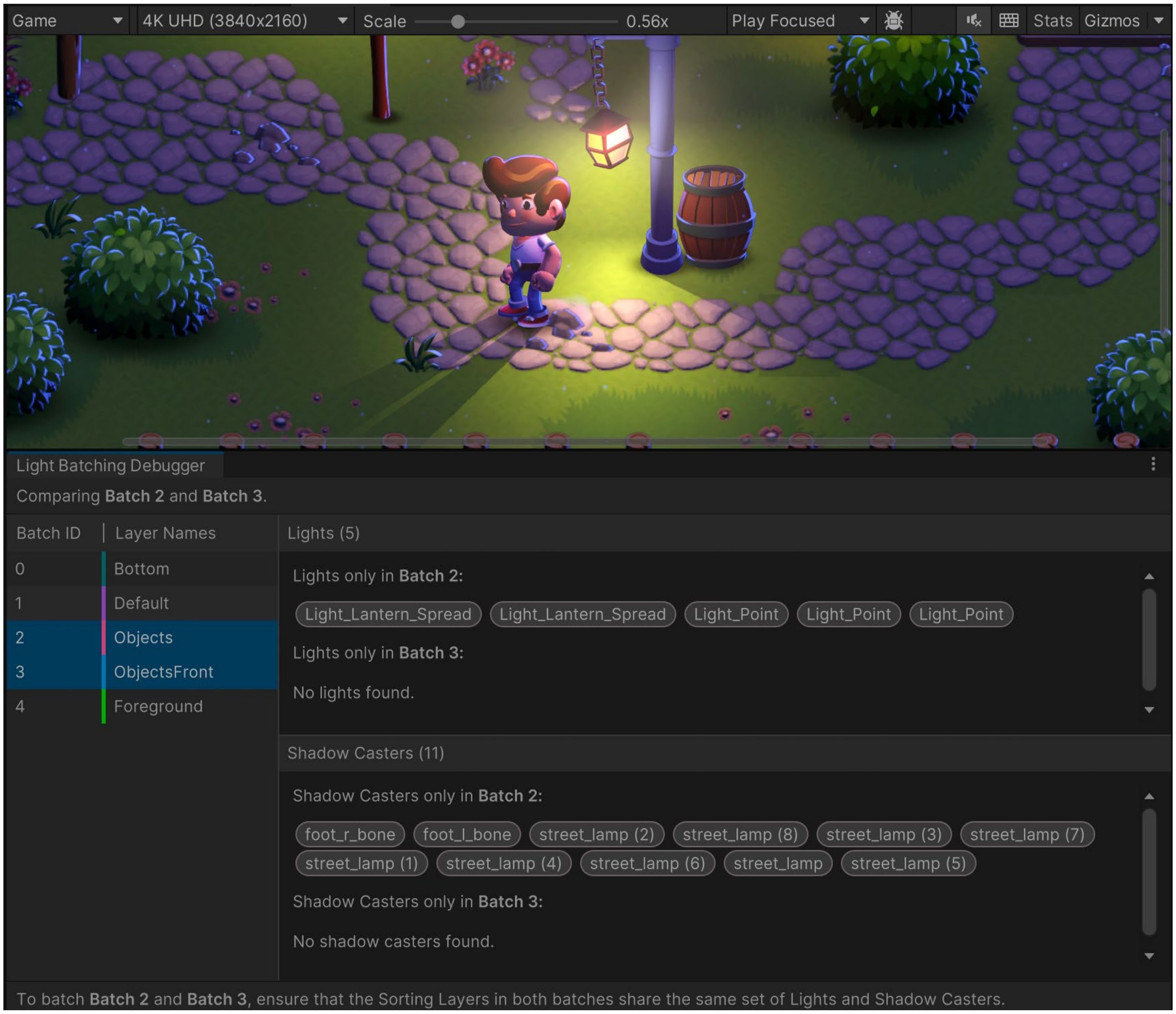
— Minimize the number of Sorting Layers and different blend styles onscreen. TThere can be a performance cost when switching to draw the blend styles.

— Adjust the number of Max Light Render Textures and Max Shadow Render Textures to fit your project's needs. Higher numbers will increase performance (up to a limit), but they will also increase the memory needed. You will need to find the right number to fit your memory and rendering needs.

— When using normal maps, set normal lighting Quality of lights to Fast instead of Accurate.

— Use 2D shadows on only a few lights.

Use the 2D Light Batching Debugger



Light Batching Debugger

Comparing **Batch 2** and **Batch 3**.

Batch ID	Layer Names
0	Bottom
1	Default
2	Objects
3	ObjectsFront
4	Foreground

Lights (5)

Lights only in **Batch 2:**

Light_Lantern_Spread Light_Lantern_Spread Light_Point Light_Point Light_Point

Lights only in **Batch 3:**

No lights found.

Shadow Casters (11)

Shadow Casters only in **Batch 2:**

foot_r_bone foot_l_bone street_lamp (2) street_lamp (8) street_lamp (3) street_lamp (7) street_lamp (1) street_lamp (4) street_lamp (6) street_lamp street_lamp (5)

Shadow Casters only in **Batch 3:**

No shadow casters found.

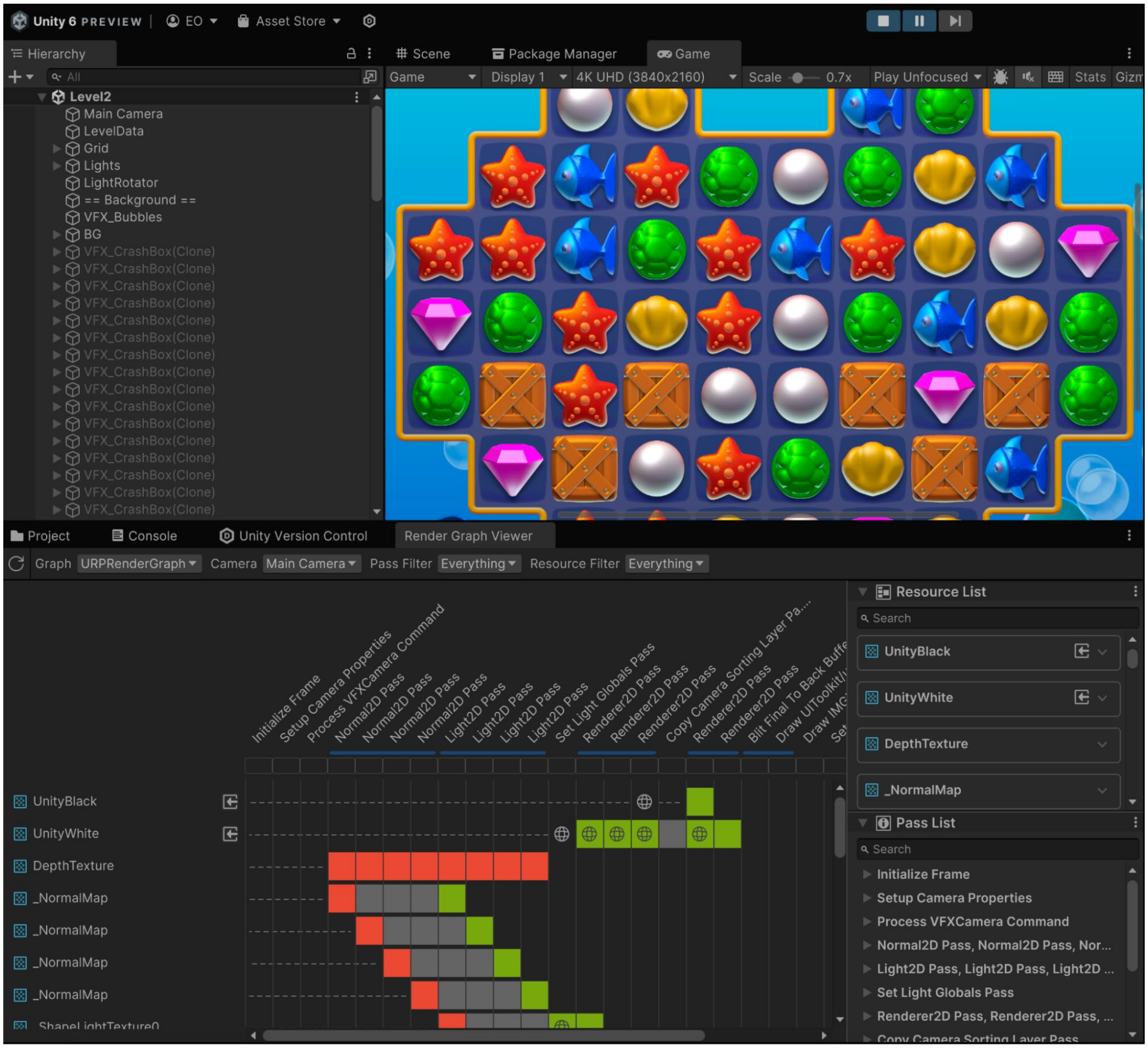
To batch **Batch 2** and **Batch 3**, ensure that the Sorting Layers in both batches share the same set of Lights and Shadow Casters.

— Use the 2D Light Batching Debugger to visualize how Unity batches 2D lights and Shadow Caster 2D components according to the Sorting Layers they target in the scene.

  — The debugger compares adjacent batches and highlights the lights or casters that target each Sorting Layer, and displays which ones you need to add or remove for Unity to be able to batch the Sorting Layers.

  — Overall, the debugger can help you to improve the performance of your game by making better decisions about how lights and shadows should be affecting Sorting Layers.

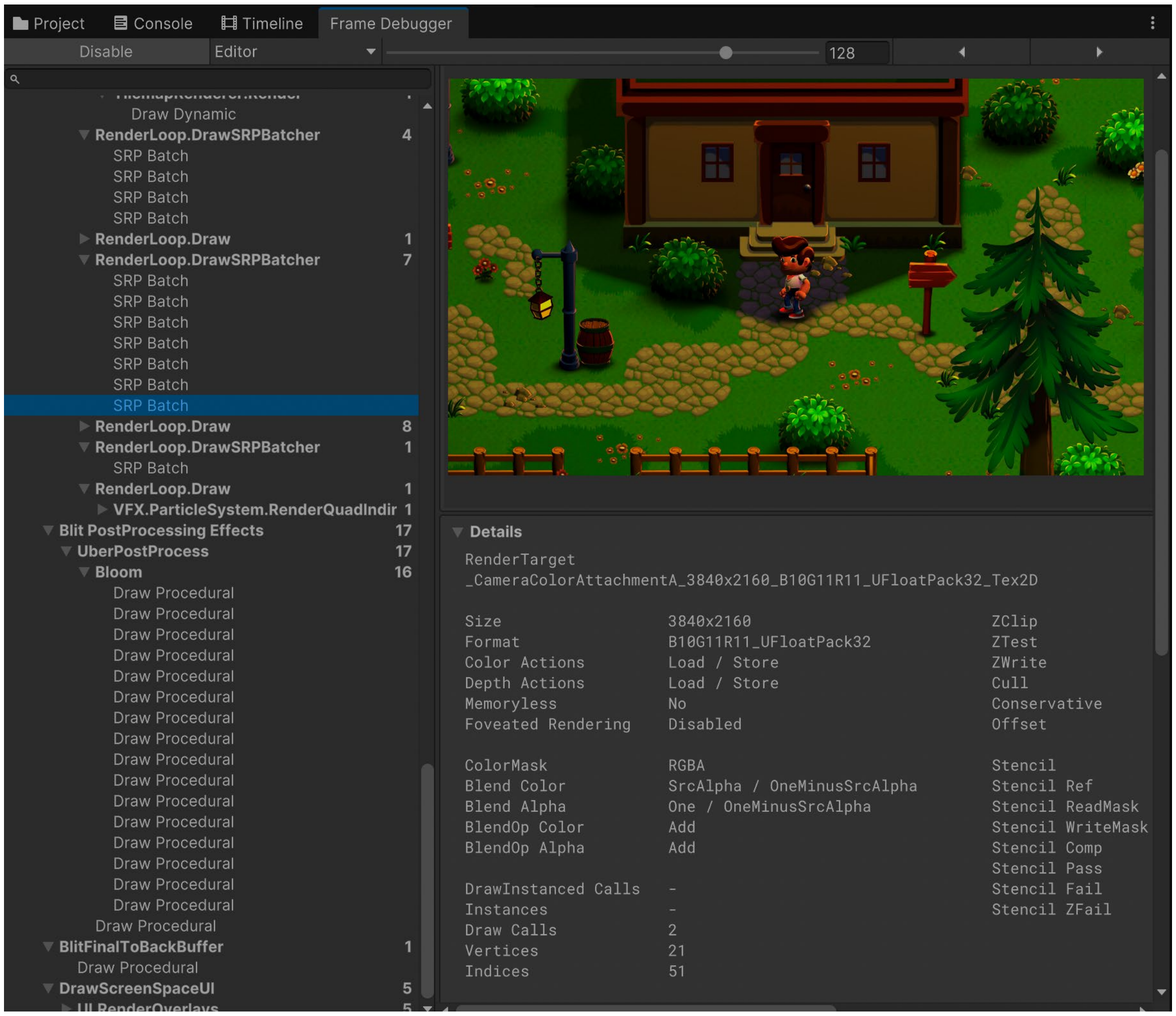
Use the 2D Renderer with Render Graph



— Render Graph automatically optimizes your render pass to minimize the number of passes, and the memory and bandwidth each render pass uses:

- — Avoids allocating extra resources that the frame doesn't use
- — Merges multiple render passes into a single render pass

Use the SRP Batcher



```
            Draw Dynamic
        ▼ RenderLoop.DrawSRPBatcher          4
            SRP Batch
            SRP Batch
            SRP Batch
            SRP Batch
        ▶ RenderLoop.Draw                    1
        ▼ RenderLoop.DrawSRPBatcher          7
            SRP Batch
            SRP Batch
            SRP Batch
            SRP Batch
            SRP Batch
            SRP Batch
            SRP Batch
        ▶ RenderLoop.Draw                    8
        ▼ RenderLoop.DrawSRPBatcher          1
            SRP Batch
        ▼ RenderLoop.Draw                    1
          ▶ VFX.ParticleSystem.RenderQuadIndir 1
    ▼ Blit PostProcessing Effects           17
      ▼ UberPostProcess                     17
        ▼ Bloom                             16
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
            Draw Procedural
          Draw Procedural
    ▼ BlitFinalToBackBuffer                  1
        Draw Procedural
    ▼ DrawScreenSpaceUI                      5
```

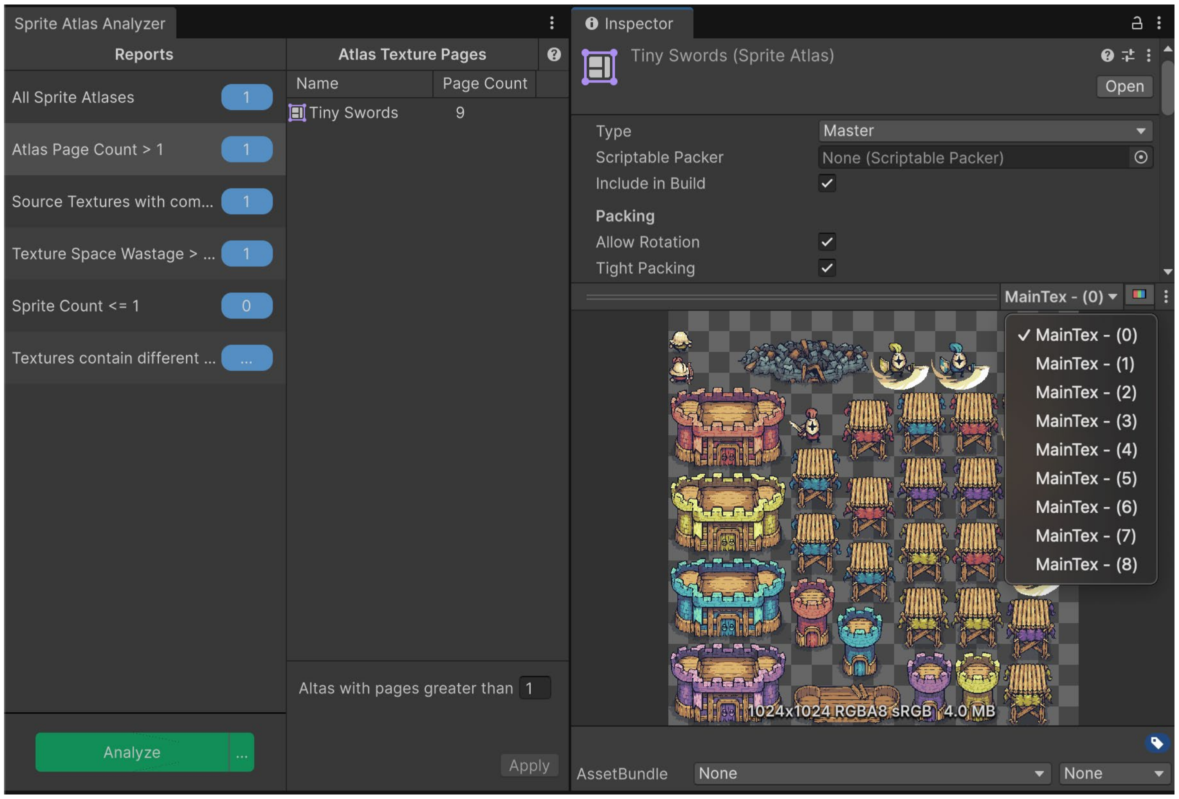
```
▼ Details
RenderTarget
_CameraColorAttachmentA_3840x2160_B10G11R11_UFloatPack32_Tex2D

Size                 3840x2160                    ZClip
Format               B10G11R11_UFloatPack32       ZTest
Color Actions        Load / Store                 ZWrite
Depth Actions        Load / Store                 Cull
Memoryless           No                           Conservative
Foveated Rendering   Disabled                     Offset

ColorMask            RGBA                         Stencil
Blend Color          SrcAlpha / OneMinusSrcAlpha  Stencil Ref
Blend Alpha          One / OneMinusSrcAlpha       Stencil ReadMask
BlendOp Color        Add                          Stencil WriteMask
BlendOp Alpha        Add                          Stencil Comp
                                                  Stencil Pass
DrawInstanced Calls  -                            Stencil Fail
Instances            -                            Stencil ZFail
Draw Calls           2
Vertices             21
Indices              51
```

— The SRP Batcher moves tasks that previously loaded the CPU to the GPU memory, freeing CPU load for gameplay.

— For sprites with dense mesh or skinned, the SRP Batcher usually results in better performance.

— **Note**: For sprites that are full rect (quads) or with very few vertices, dynamic-batching (disabling SRP Batcher) is preferable (per object with API).

Use the Sprite Atlas Analyzer



Reports	
All Sprite Atlases	1
Atlas Page Count > 1	1
Source Textures with com...	1
Texture Space Wastage > ...	1
Sprite Count <= 1	0
Textures contain different ...	...

Atlas Texture Pages

Name	Page Count
Tiny Swords	9

Altas with pages greater than 1

Tiny Swords (Sprite Atlas)

Type	Master
Scriptable Packer	None (Scriptable Packer)
Include in Build	✓
Packing	
Allow Rotation	✓
Tight Packing	✓

1024x1024 RGBA8 sRGB 4.0 MB

AssetBundle None None

Looking at the Atlas Page Count in the Sprite Atlas Analyzer window

New in Unity 6.3 LTS, the Sprite Atlas Analyzer window can help you pinpoint potential performance issues with your sprite atlases. It provides detailed information about all of the 2D texture assets contained in all sprite atlases within your project.

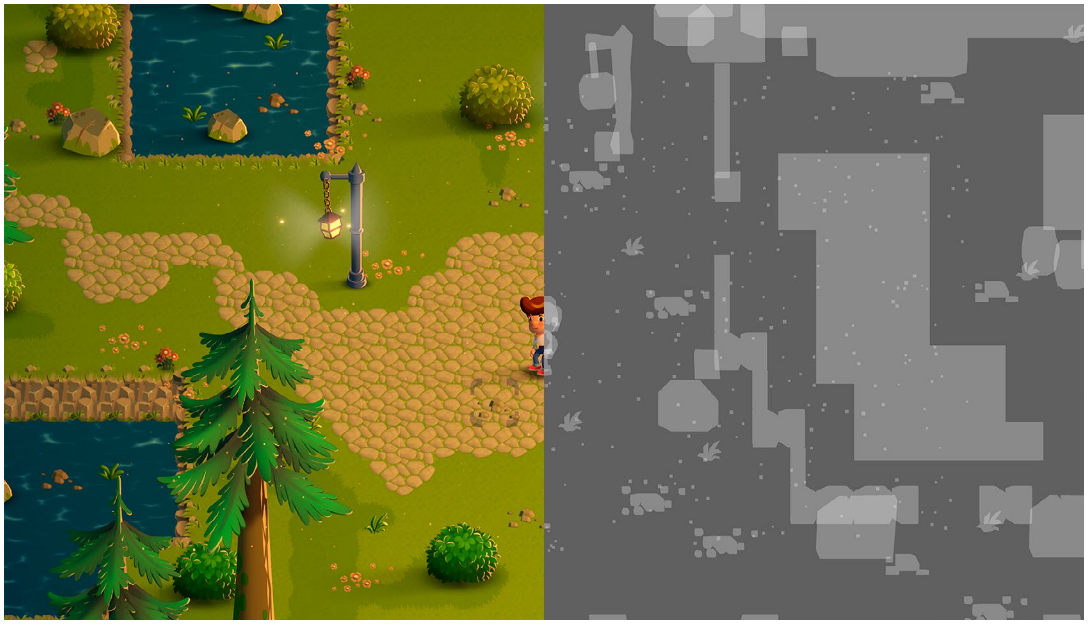
To use it, install the package, then go to **Window > Analysis > Sprite Atlas Analyzer** to launch it. When its window opens, click the **Analyze** button to begin analyzing the contents of the sprite atlases in the project.

The Sprite Analyzer comes with six built-in reports:

— All Sprite Atlases
— Atlas page count
— Source textures with compression in Sprite Atlas
— Texture Space Wastage
— Sprite count
— Textures contain different secondary texture count in Sprite Atlas

To learn more, see the Sprite Atlas Analyzer section in the Unite 2025 session on 2D tools.

Optimize tilemaps



From *Happy Harvest*: On the right side, the Overdraw Mode is enabled in the Rendering Debugger, the light areas show overlapping pixels, the dark areas where there's less overlap.

— Consider using tilemaps when your scene consists of many sprites. This enables you to trade many Sprite Renderers (they have an overhead) for just one Tilemap Renderer.

— Mark your level colliders as Static for batching.

— Put sprites that will likely be together on the scene in a Sprite Atlas.

— Simplify sprites' meshes by using the Custom Outline option in the Sprite Editor.

— Cache Sprite Shape geometry by enabling the Cache Geometry option (it appears when the Edit Spline button is on).

Optimize your animations

— Install the Burst package to improve 2D Animation performance.

— When making a skeleton, avoid using more bones and sprites than you need.

— Simplify skeleton meshes by using as few vertices as possible.

— Turn on Culling in the animators. This option will be turned off when the animated character is offscreen to save performance.

— Avoid using skeletal animations for objects or characters that are small or come in large numbers, such as background birds, flies, or small animals. Instead, use different techniques like frame-by-frame animation or shaders.

— Try to maintain a low frame count/texture size when using frame-by-frame animation.

VFX and post-processing effects

— Use as few post-processing effects as possible.

— Use less intensive effects, such as Bloom, Chromatic Aberration, Color Grading, Lens Distortion, or Vignette, wherever possible.

— Film Grain and Panini Projection are more costly.

— Disable High Quality Filtering on Bloom

— With Object Pooling, you can avoid instantiating and destroying VFX assets. Simply activate and deactivate GameObjects.

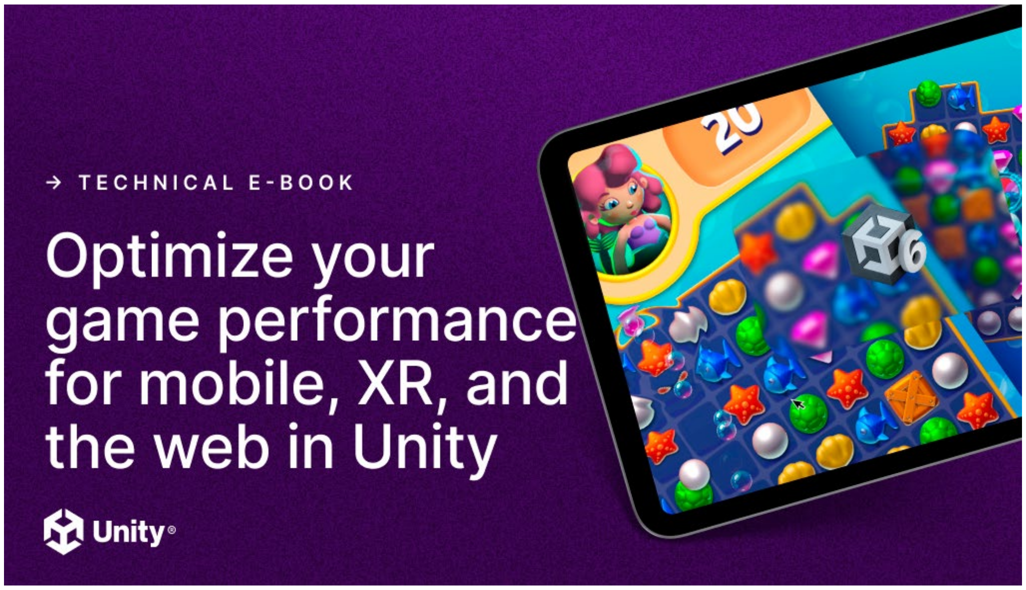
— In the Shader Graph, use half precision whenever possible.

Learn how to use performance tools in Unity

Our Unity 6 e-books will help you develop advanced performance optimization skills for your Unity projects:

This 70+ page guide brings together advanced knowledge and advice from external and in-house Unity experts on how to profile an application in Unity, manage its memory, and optimize its power consumption from start to finish.

Download the guide


→ TECHNICAL E-BOOK
Optimize your game performance for mobile, XR, and the web in Unity
Unity®

This guide can help you launch a performant application on the widest range of devices to maximize your opportunity for success at launch and beyond.

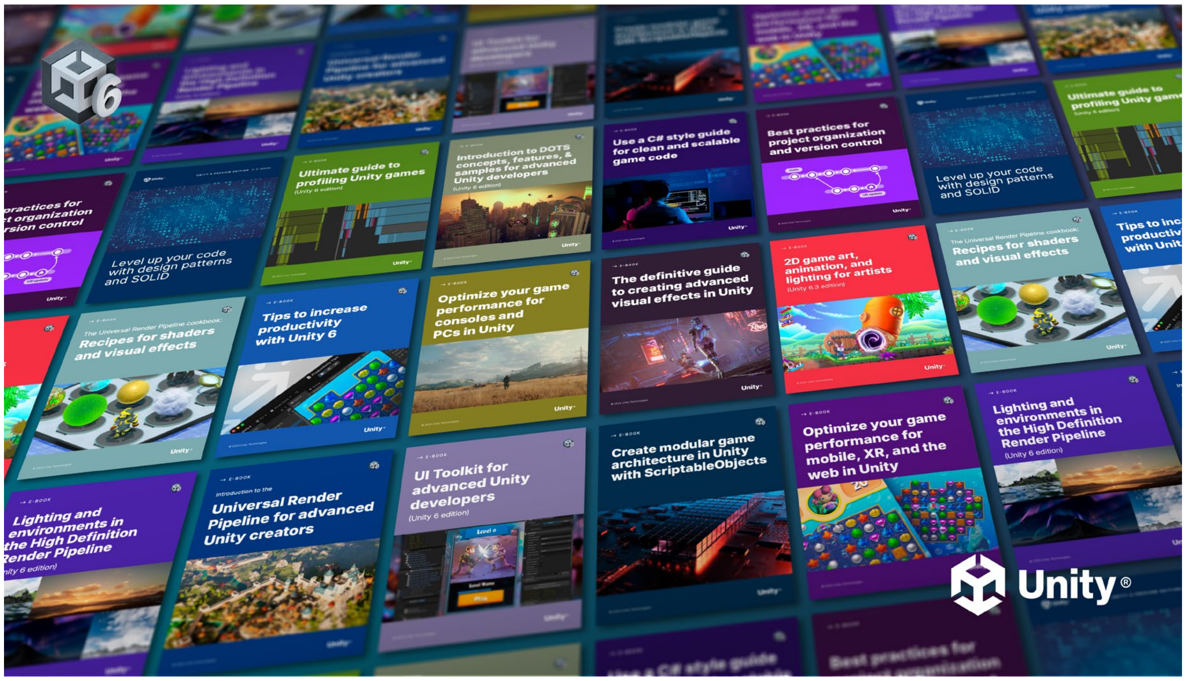
Download the guide

→ TECHNICAL E-BOOK
Optimize your game performance for consoles and PCs in Unity
Unity®

Learn how to optimize your code architecture and art assets, and launch a game on target hardware that plays smoothly and within its limitations.

Download the guide

Conclusion



You can download many more e-books for advanced Unity developers and creators from the Unity best practices hub. Choose from over 30 guides, created by industry experts, and Unity engineers and technical artists.

And find more tips and news on the Unity Blog, Unity Discussions, Unity Learn and via the #unitytips hashtag.

Appendix: 2D visual effects

On the left is an image from the Unity demo Dragon Crashers – 2D Sample Project and on the right, an image from Epic Toon FX by Archanor VFX. Both packages are available on the Asset Store.

Visual effects (VFX) are the cherry on top of any great-looking game, and they're vital to the gamers' experience as they play. VFX communicate game events such as environmental hazards and healing zones, and they visually reward the player for well-executed actions, like an action scene culminating in a big, fiery explosion.

You can create 2D visual effects in Unity with several different methods. For a fire effect, for example, you can:

— Animate the flames frame-by-frame.

— Animate a sprite with a shader made in Shader Graph.

— Spawn fiery particles either with the Built-in Particle System which runs on the CPU, or the VFX Graph, which harnesses GPU power to spawn potentially millions of particles (both systems can be used in the same project so you can pick and choose what's best for a particular effect).

There are a few key points to consider before creating special effects in a Unity 2D project:

— **Cameras**: If you use a perspective camera your effects can happen in a tridimensional space like they do in 3D games. If you use an orthographic camera, you need to set up the renderer of the system to use the right Sorting Layer. Read more about camera perspective here.

— **Post-processing**: You can apply post-processing effects to your 2D project if you choose the Universal Render Pipeline (URP), e.g., adding a bloom effect to a particle effect via the URP Volume framework.

— Effects can be achieved in different ways. For example, a GameObject representing an effect can have several child GameObjects using different systems, like sprites with custom shaders or several particle systems.

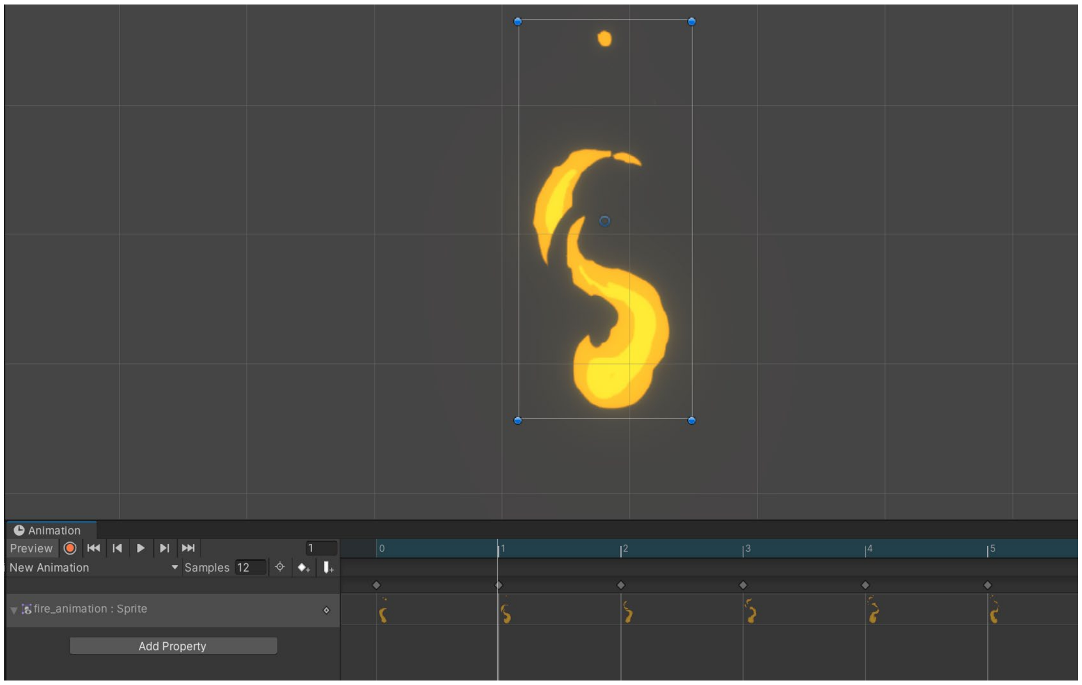
— **Animation clips**: Animation clips can help you orchestrate smooth sprite-based animation effects. You can even use Unity's skeletal animation system (sparingly) to control parts of a sprite in a sprite animation.

— **Unity Asset Store**: You can get a head start on adding visual effects to your project by heading over to the Unity Asset Store, where you'll find many ready-made 2D effects.

This section covers all of these methods, with a focus on VFX Graph, mainly by looking at specific effects in the *Dragon Crashers, Happy Harvest,* and *Gem Hunter Match* samples.

Unity toolsets for 2D VFX

Frame-by-frame animation



Frame-by-frame animation of a flame

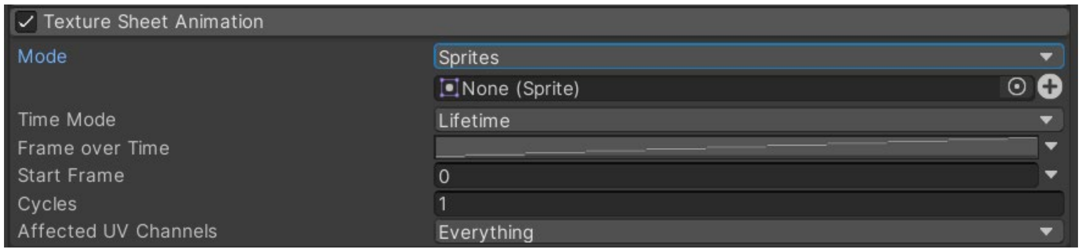
Frame-by-frame (or flipbook) animation lets you add effects quickly in Unity. Simply import frames as sprites, then animate the Sprite Property over time in the Animation window. You can also select and drag all the frame sprites into the Scene, which will then animate them automatically.

While importing frames is fast, drawing each frame to be animated is not, and this process can require much more time and skill than you have available. To save time, you can instead export VFX as frames from other apps.

Another shortcut you can use is to select all of the animation frame assets from the Project view, and drag them to the Hierarchy view or Scene. A new GameObject will then be created with an animation that uses the sequence made of the previously selected images.

The flipbook animation technique is very performant, so use it when you want to have many instances of the effect onscreen. Remember that many frames in high resolution will require more memory.

Frame-by-frame animation can also be used in Particle Systems, where it is referred to as Texture Sheet Animation.

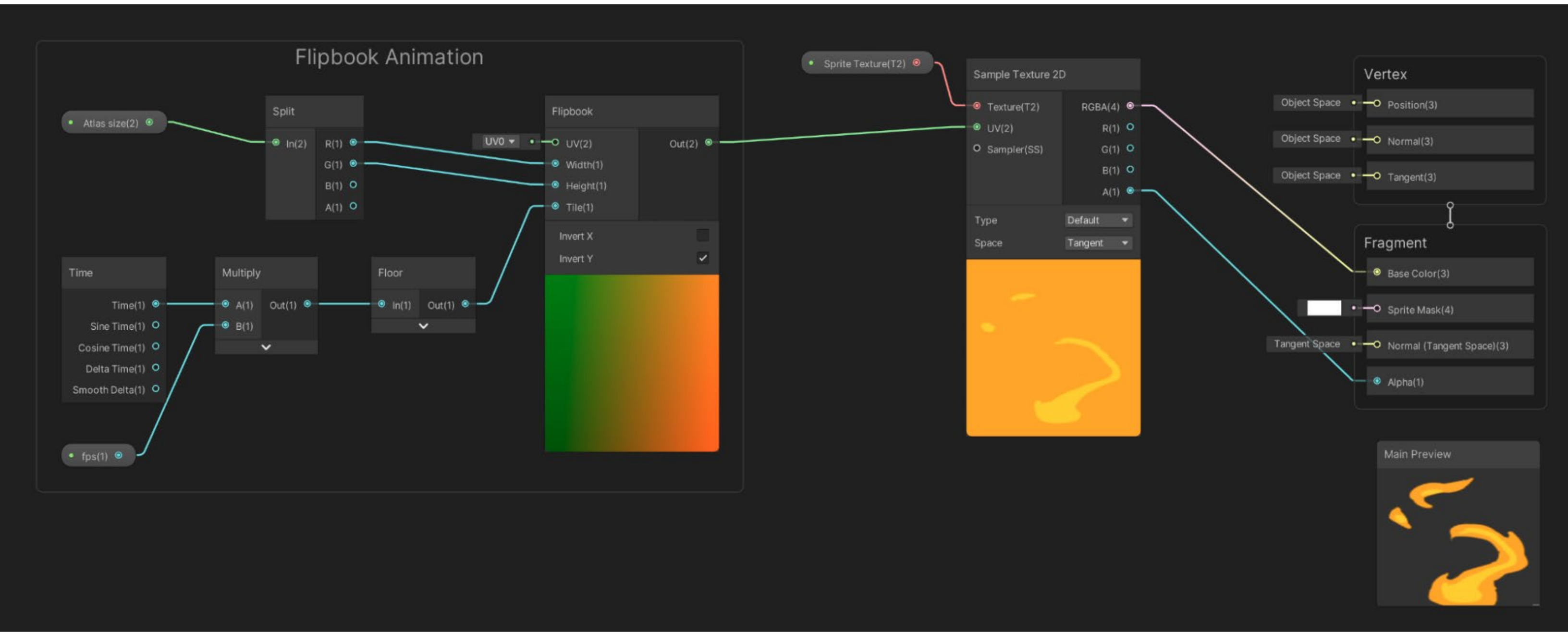


✓ Texture Sheet Animation

Mode	Sprites
	None (Sprite)
Time Mode	Lifetime
Frame over Time	
Start Frame	0
Cycles	1
Affected UV Channels	Everything

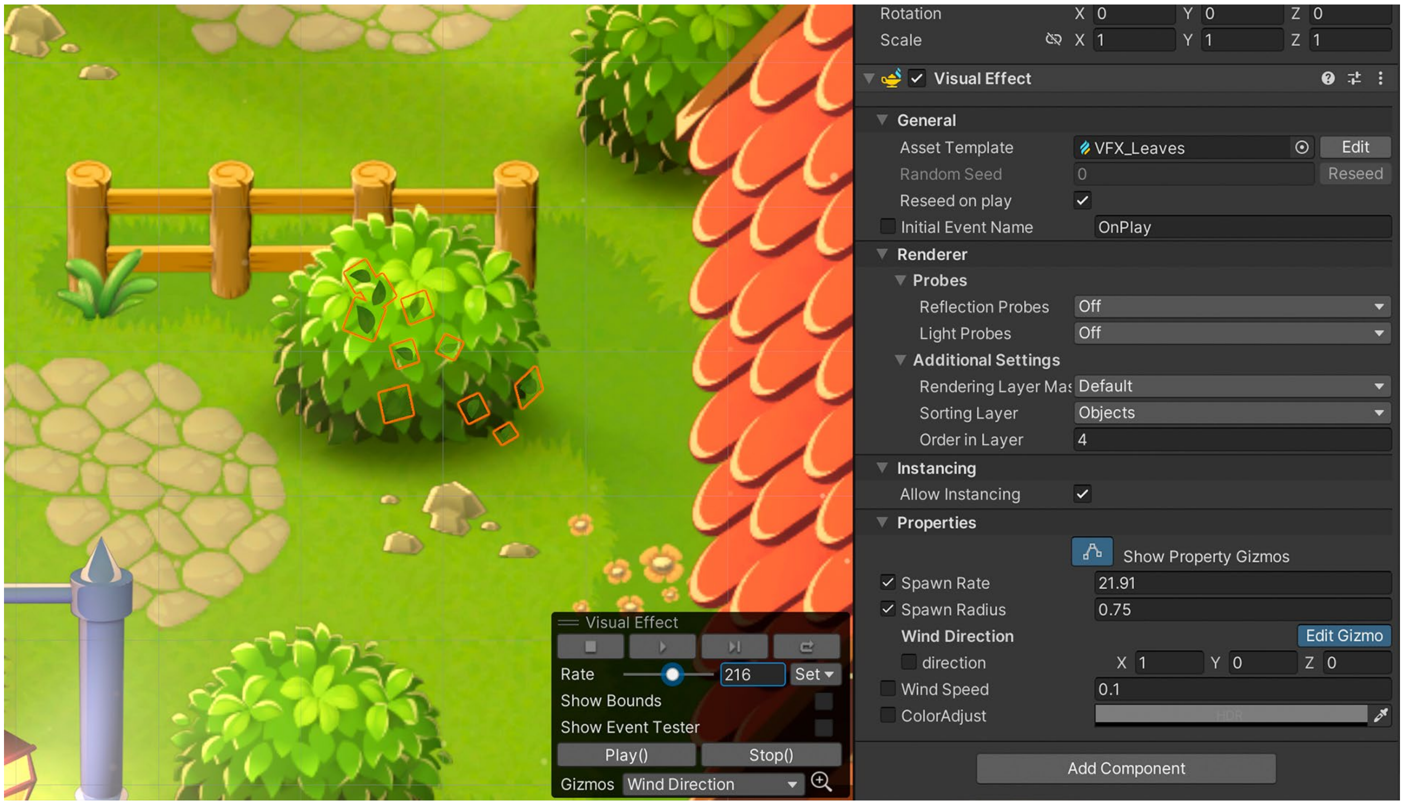
Setting up frame-by-frame animation in Particle System

Flipbook animations can also be used in the Shader Graph by animating the UV position of texture over time, which creates the illusion of moving frames.



Flipbook animation in Shader Graph

The VFX Graph for 2D


```
Rotation                X 0        Y 0        Z 0
Scale                   X 1        Y 1        Z 1
✓ Visual Effect
General
  Asset Template        VFX_Leaves            Edit
  Random Seed           0                     Reseed
  Reseed on play        ✓
  Initial Event Name    OnPlay
Renderer
  Probes
    Reflection Probes   Off
    Light Probes        Off
  Additional Settings
    Rendering Layer Mas Default
    Sorting Layer       Objects
    Order in Layer      4
Instancing
  Allow Instancing      ✓
Properties
                        Show Property Gizmos
✓ Spawn Rate            21.91
✓ Spawn Radius          0.75
  Wind Direction                              Edit Gizmo
    direction           X 1    Y 0    Z 0
  Wind Speed            0.1
  ColorAdjust
                  Add Component

Visual Effect
Rate                216   Set
Show Bounds
Show Event Tester
Play()              Stop()
Gizmos  Wind Direction
```

In *Happy Harvest*, a VFX Graph is used to create an effect of leaves falling off the bushes as the character walks by.

The VFX Graph is a node-based editor that enables technical and VFX artists to design dynamic, GPU-accelerated particle effects – from simple common particle behaviors to complex simulations involving particles, lines, ribbons, trails, and more.

The VFX Graph system works with both HDRP and URP, and is compatible with the 2D Renderer using Shader Graph in versions of Unity from 2022 LTS and newer.

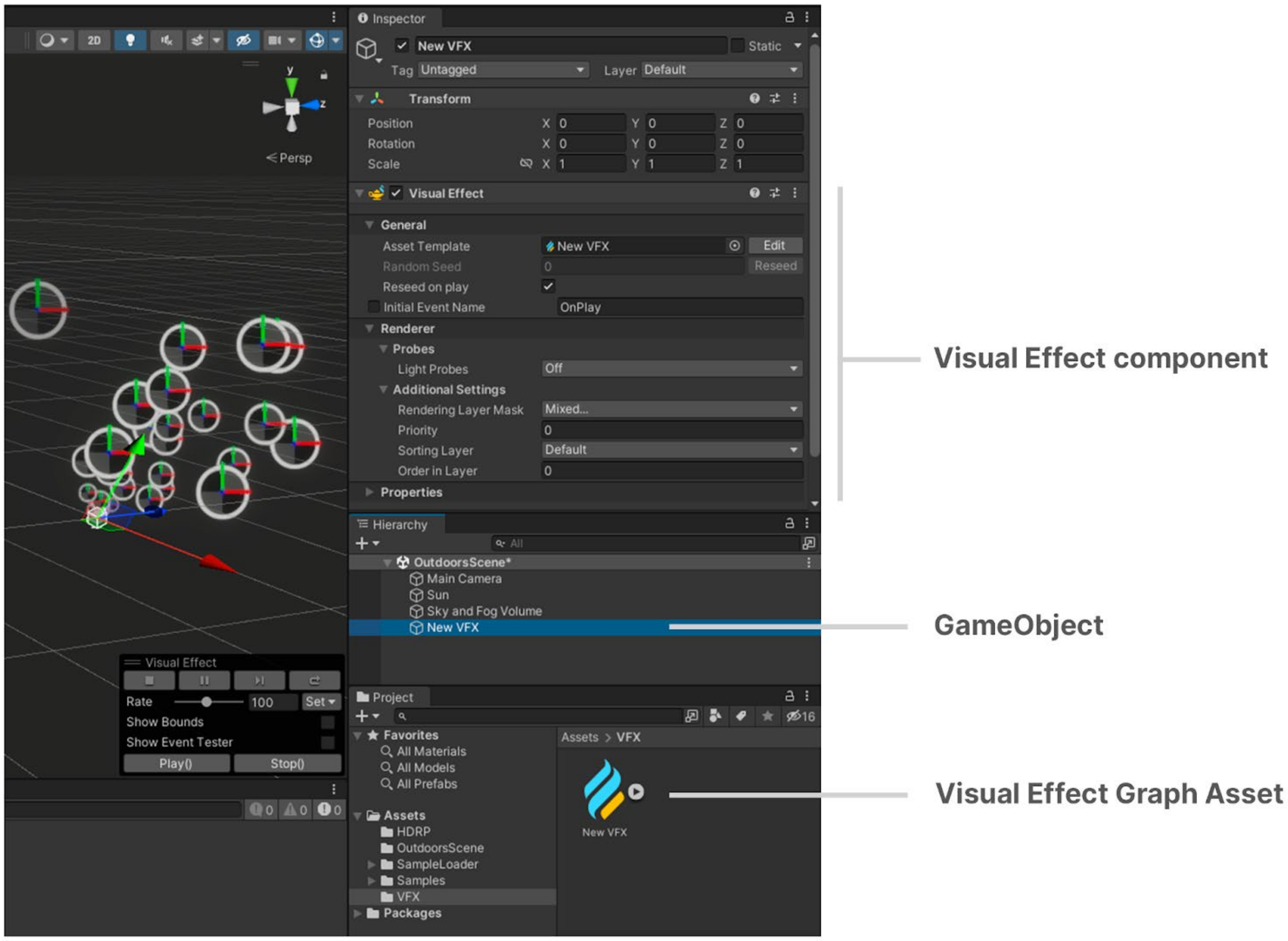
We recommend that you reference VFX Graph documentation to create a new graph in your project. Let's look at the basic outline of a VFX Graph, which will help you understand why it's such a powerful tool for creating effects in both 2D and 3D games.

Any visual effect in the VFX Graph is made up of these two parts:

— Visual Effect (VFX) component attached to a GameObject in the scene

— Visual Effect (VFX) Graph Asset that lives at the project level

As Unity stores each VFX Graph in the Assets folder, you must connect each asset to a Visual Effect component in your scene. Keep in mind that different GameObjects can refer to the same graph at runtime.



New VFX — Static
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Visual Effect
General
Asset Template: New VFX (Edit)
Random Seed: 0 (Reseed)
Reseed on play: ✓
Initial Event Name: OnPlay
Renderer
Probes
Light Probes: Off
Additional Settings
Rendering Layer Mask: Mixed...
Priority: 0
Sorting Layer: Default
Order in Layer: 0
Properties

Hierarchy: OutdoorsScene* — Main Camera, Sun, Sky and Fog Volume, New VFX

Visual Effect panel: Rate 100 (Set); Show Bounds; Show Event Tester

Project: Favorites — All Materials, All Models, All Prefabs; Assets — HDRP, OutdoorsScene, SampleLoader, Samples, VFX; Packages
Assets > VFX: New VFX

The VFX Graph Asset and Visual Effect component

The VFX Graph window

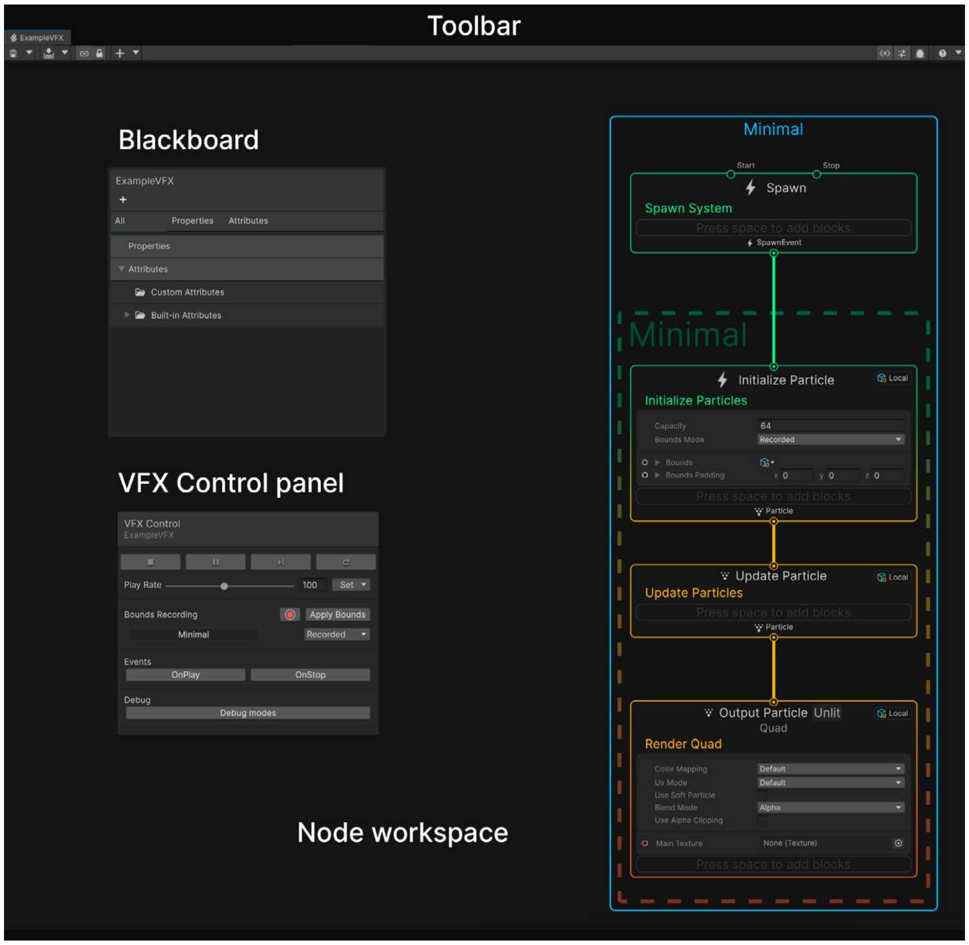
The window's layout includes:

— **Toolbar**: To access Global settings, as well as toggles for several panels

— **Node workspace**: To compose and edit the VFX Graph

— **Blackboard**: To manage attributes and properties that are reusable throughout the graph

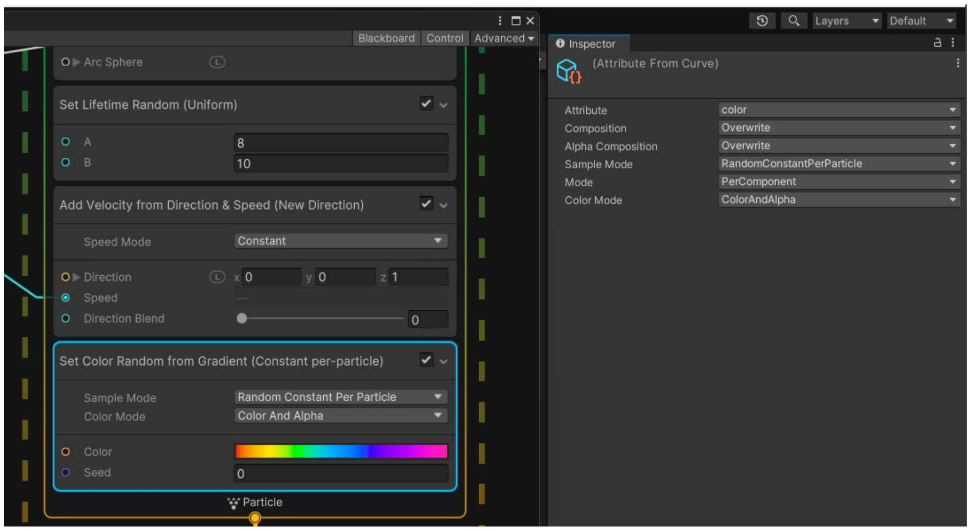
— **VFX Control panel**: To modify playback on the attached GameObject



Capacity 64
Bounds Mode Recorded
Color Mapping Default
Uv Mode Default
Blend Mode Alpha
Main Texture None (Texture)

The VFX Graph window

Make sure you leave some space in the Editor layout for the Inspector. Selecting part of the graph can expose certain parameters, such as partition options and render states.



Set Lifetime Random (Uniform)
A 8
B 10

Add Velocity from Direction & Speed (New Direction)
Speed Mode Constant
Direction x 0 y 0 z 1
Speed
Direction Blend 0

Set Color Random from Gradient (Constant per-particle)
Sample Mode Random Constant Per Particle
Color Mode Color And Alpha
Color
Seed 0

(Attribute From Curve)

Attribute	color
Composition	Overwrite
Alpha Composition	Overwrite
Sample Mode	RandomConstantPerParticle
Mode	PerComponent
Color Mode	ColorAndAlpha

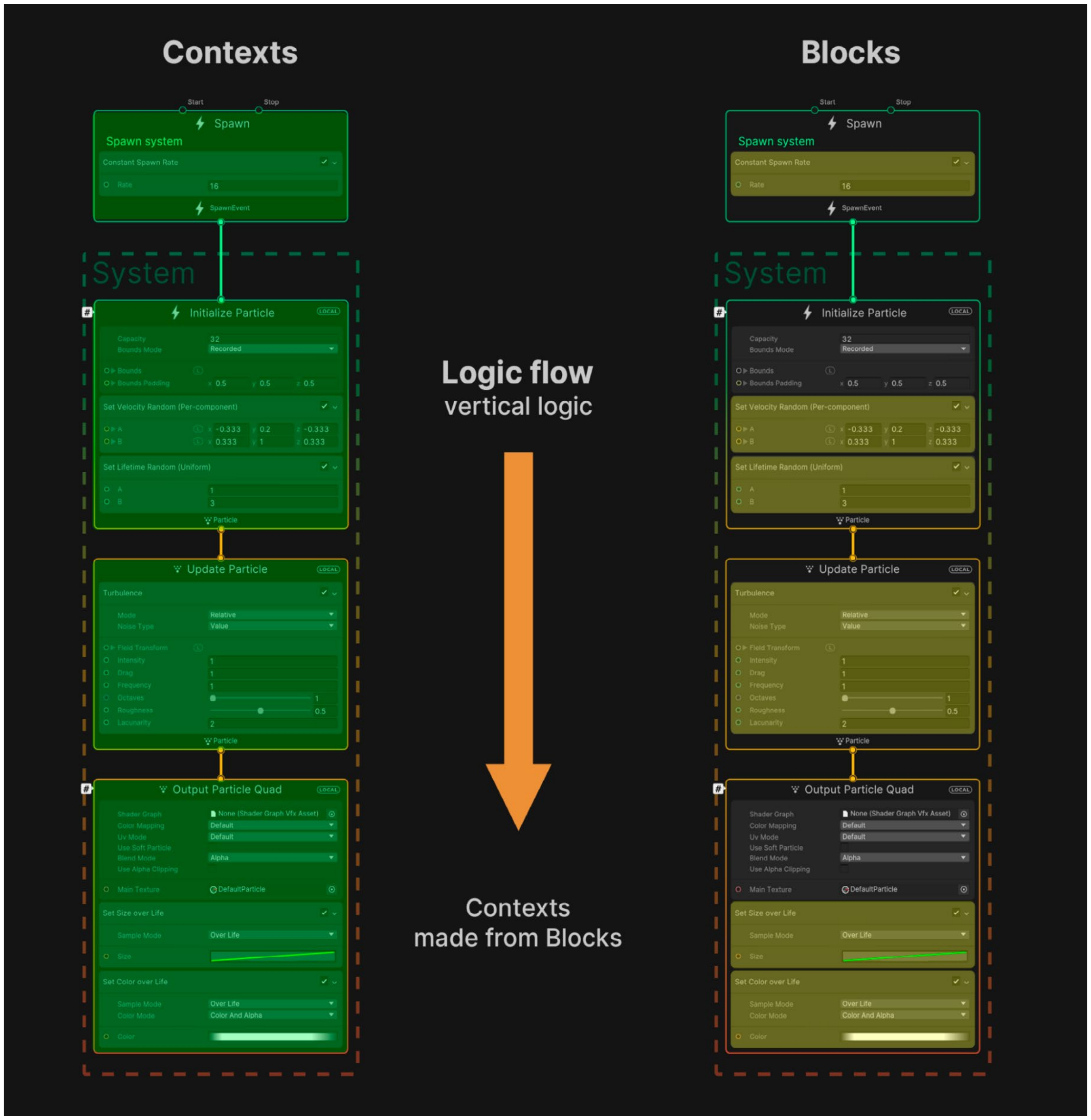
Use the Inspector to change certain parameters.

Graph logic

You'll build your visual effect from a network of nodes inside the window's workspace. The VFX Graph uses an interface similar to other node-based tools, such as Shader Graph, which is introduced later on in this chapter.

A VFX Graph includes one or more vertical stacks called Systems. Systems define standalone portions of the graph and encompass several Contexts. A System is denoted by the dotted line that frames the Contexts it consists of.

Each Context is composed of individual Blocks, which can set Attributes (size, color, velocity, etc.) for its particles and meshes. Multiple Systems can work together within one graph to create the final visual effect.



Constant Spawn Rate
Rate 16

Capacity 32
Bounds Mode Recorded
Bounds
Bounds Padding x 0.5 y 0.5 z 0.5

Set Velocity Random (Per-component)
A x -0.333 y 0.2 z -0.333
B x 0.333 y 1 z 0.333

Set Lifetime Random (Uniform)
A 1
B 3

Turbulence
Mode Relative
Noise Type Value
Field Transform
Intensity 1
Drag 1
Frequency 1
Octaves 1
Roughness 0.5
Lacunarity 2

Shader Graph None (Shader Graph Vfx Asset)
Color Mapping Default
Uv Mode Default
Use Soft Particle
Blend Mode Alpha
Use Alpha Clipping
Main Texture DefaultParticle

Set Size over Life
Sample Mode Over Life
Size

Set Color over Life
Sample Mode Over Life
Color Mode Color And Alpha
Color

The vertical logic in a graph flows downward.

The Output Contexts in a VFX Graph define how the effect will be rendered. Output Contexts for meshes like Output Particle Quad offer the option to add a Shader Graph. It doesn't matter if you start with a Lit or Unlit Output as the lighting model will be changed to adapt to the Shader Graph. You can use the default shaders shown listed below, or your own Shader Graph-based shader.

Quick tips for VFX Graph in 2D

For 2D projects in URP you can use one of these three sprite-based Shader Graphs:

— Lit, which receives light from 2D light system

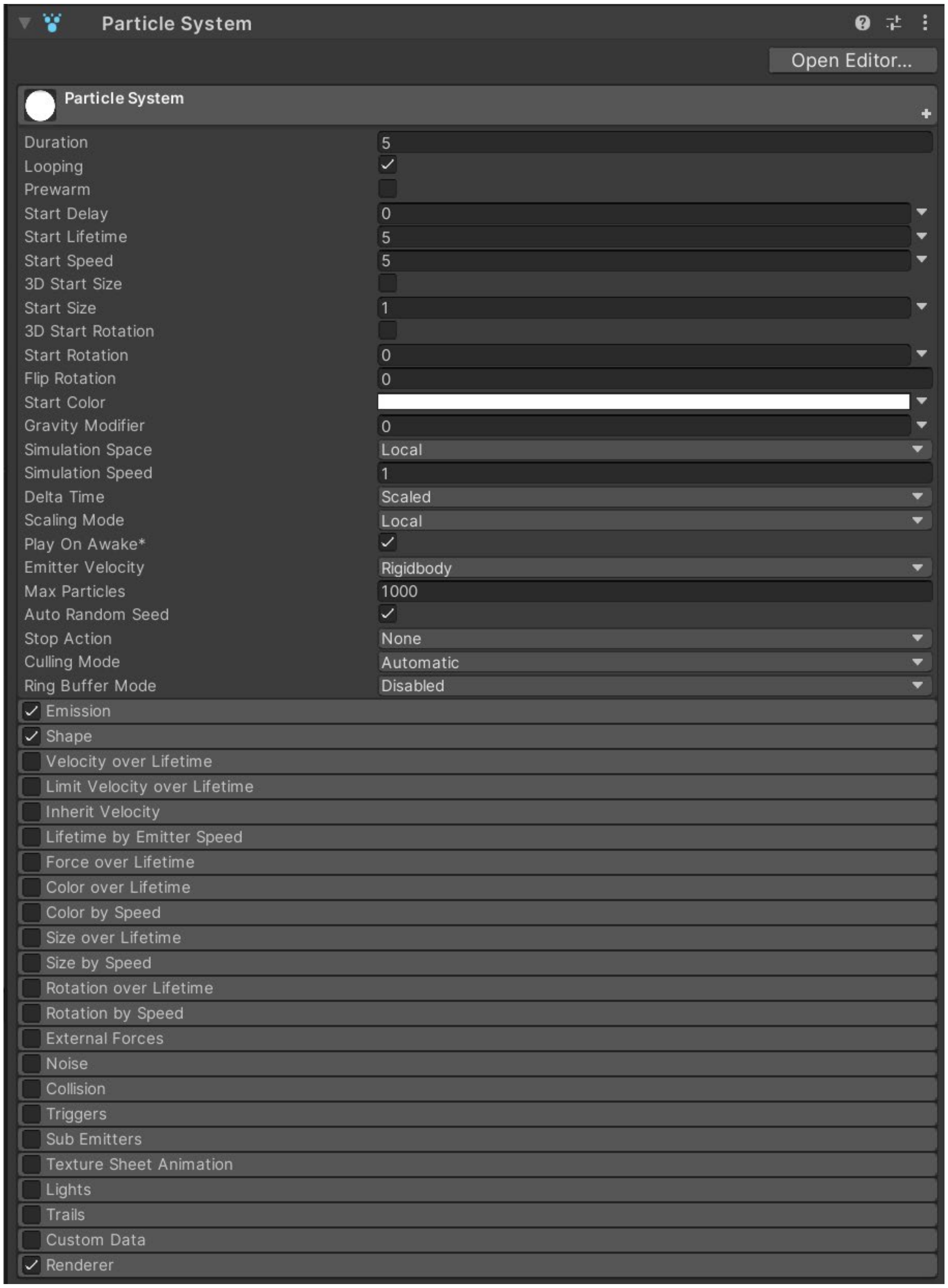
— Unlit, which is not affected by the 2D lighting system

— Custom, if you want to implement a custom 2D lighting model

Remember to check the **Support VFX Graph** option in the SRP target inside Shader Graph.

Properties exposed in your Shader Graph are shown in your VFX Graph. You can override them if needed, e.g., if you want a VFX Graph particle system to control the changing color of the shader over time.

The Built-In Particle System



Particle System

Duration	5
Looping	✓
Prewarm	
Start Delay	0
Start Lifetime	5
Start Speed	5
3D Start Size	
Start Size	1
3D Start Rotation	
Start Rotation	0
Flip Rotation	0
Start Color	⬜
Gravity Modifier	0
Simulation Space	Local
Simulation Speed	1
Delta Time	Scaled
Scaling Mode	Local
Play On Awake*	✓
Emitter Velocity	Rigidbody
Max Particles	1000
Auto Random Seed	✓
Stop Action	None
Culling Mode	Automatic
Ring Buffer Mode	Disabled

- ✓ Emission
- ✓ Shape
- Velocity over Lifetime
- Limit Velocity over Lifetime
- Inherit Velocity
- Lifetime by Emitter Speed
- Force over Lifetime
- Color over Lifetime
- Color by Speed
- Size over Lifetime
- Size by Speed
- Rotation over Lifetime
- Rotation by Speed
- External Forces
- Noise
- Collision
- Triggers
- Sub Emitters
- Texture Sheet Animation
- Lights
- Trails
- Custom Data
- ✓ Renderer

The properties included in the Built-In Particle System

The Built-In Particle System allows you to display and animate many smaller images or meshes in order to achieve a single visual effect. Particle properties like size, velocity, color, and rotation can be animated over time using certain predefined rules and randomization. This allows you to create dynamic effects like fire, explosions, smoke, magic spells, and so on.

You create a new Particle System by selecting the menu option **GameObject > Effects > Particle System**.

The Particle System Main module contains global properties that affect the whole system. Most of these properties control the initial state of newly created particles. Key properties include:

— Duration: How long the system will run

— Looping: Whether the whole system will loop forever

— Speed: The initial speed of the particles

— Start Size: The initial size of the particles

— Color: The initial color of the particles

— Gravity: Gravity force applied to the particles

See the extensive Particle System module component reference section in documentation to learn about all of the properties, or modules, included in the Built-In Particle System.

Particle Systems can also be nested within one another. Nesting causes systems to play simultaneously, which allows you to achieve some very complex effects. For example, to create an explosion, you could combine one system for smoke, one for small sparks, and one for flying debris.

The Particle System is a very powerful tool that allows you to create almost any VFX for your game. You can even make your effects glow using HDR unlit shader and Bloom post-processing. Test out every property, for even the simplest particles will make your game feel polished.

To speed up your game creation process significantly, check the Asset Store for premade particle VFX.

Shader Graph

Another way to make VFX is using shaders. Shaders are a set of instructions for the graphics processor, and they can, for example, calculate pixel colors or vertex positions.

Shader Graph enables you to build shaders visually. Instead of writing code, you create and connect nodes in a graph framework. Shader Graph gives instant feedback that reflects your changes, and it's simple enough for users who are new to shader creation.

With the Shader Graph you can:

— Warp and animate UVs.

— Procedurally alter surface appearance.

— Ad image filters similar to image editing apps.

— Change an object's surface based on information like world location, normals, distance from the camera, and more.

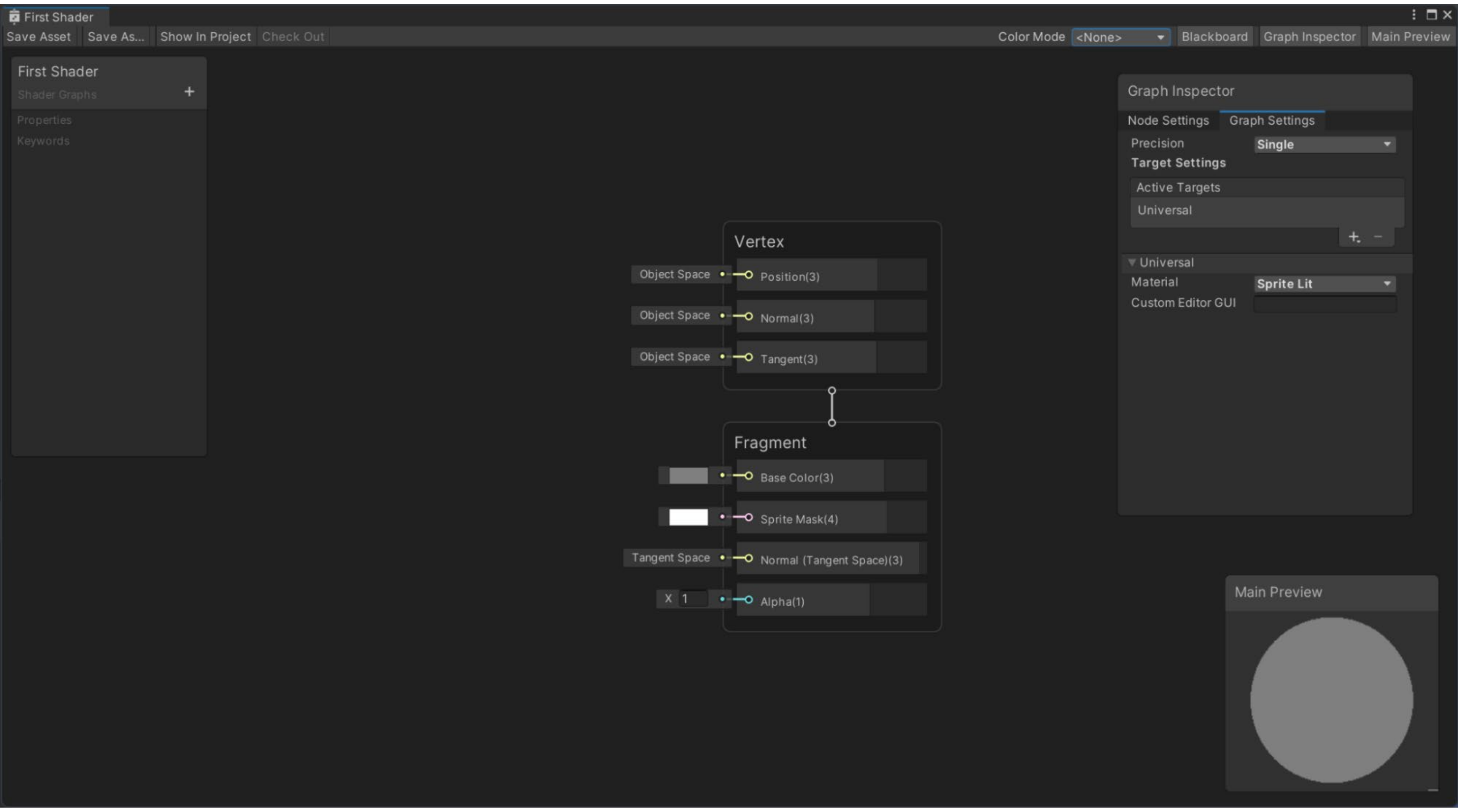
— Tweak shader visuals in a Scene's context by exposing properties to Material Inspector.

Using the Shader Graph

To start using the Shader Graph for 2D, go to the Project window, right-click, and select **Create > Shader > Universal Render Pipeline > Sprite Lit Shader Graph** (or **Sprite Unlit Shader Graph**).

Use the Sprite Lit Shader when you want your sprite to be affected by lights, which will be the case for almost all characters and objects. Use Sprite Unlit for unlit or emissive surfaces such as fire, light sources, electricity, spells, holograms, or characters like ghosts.

To start creating a shader, name the file first. This will create your initial shader. Then, double-click the file to enter Edit mode.



Precision: Single
Target Settings
Universal
Material: Sprite Lit
Custom Editor GUI

The Shader Graph window

This opens the Shader Graph window. Within the central part, called the Workspace, is the Master Stack. This is the final output of the Shader Graph, which is responsible for the shader's ultimate look. There should only be one Master Stack per shader.

The Master Stack contains two Contexts, which correspond to vertex and fragment (or pixel) functions.

On the left hand, you'll see a Blackboard. This contains properties that will be visible in the Inspector and Keywords.

At the top right is a Graph Inspector, which has two tabs: Node settings, which allows you to change a selected Node's settings, and Graph Settings, which changes the entire graph's settings.

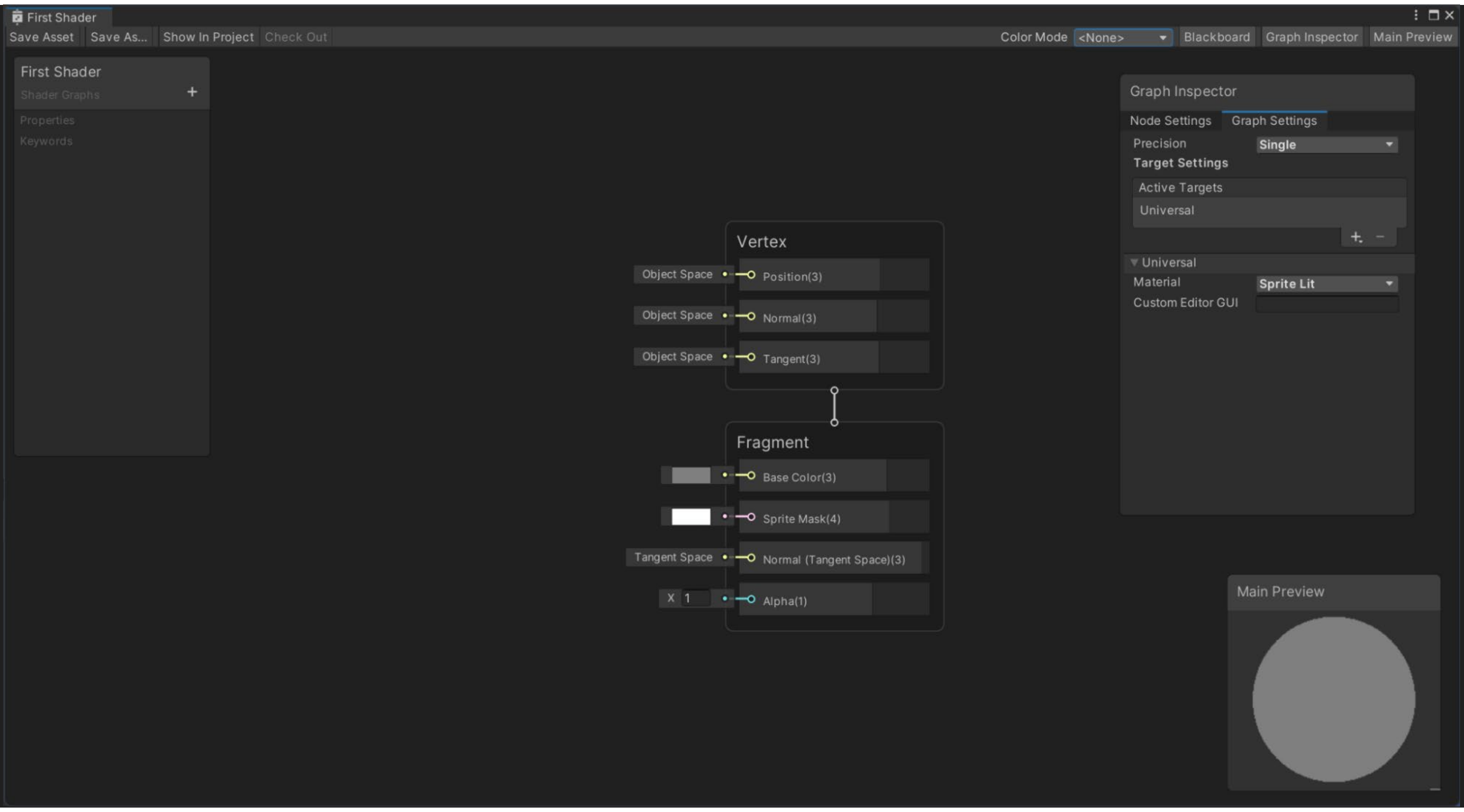
The Main Preview at the bottom is where you can view the final shader.

The nodes in Shader Graph are the building blocks that hold inputs, effects, and interactions. A graph proceeds from left to right, so nodes will have input slots on the left, and outputs on the right.

You'll find all of the steps to creating a 2D Shader Graph here. Shader Graph documentation is here for further, detailed reading about each part of a shader graph and how to set up its logic.

See the following section to read about a few shader graph examples in *Dragon Crashers* and *Gem Hunter Match*.

To start creating a shader, name the file first. This will create your initial shader. Then, double-click the file to enter Edit mode.



Precision: Single
Target Settings
Active Targets: Universal
Universal
Material: Sprite Lit
Custom Editor GUI

The Shader Graph window

This opens the Shader Graph window. Within the central part, called the Workspace, is the Master Stack. This is the final output of the Shader Graph, which is responsible for the shader's ultimate look. There should only be one Master Stack per shader.

The Master Stack contains two Contexts, which correspond to vertex and fragment (or pixel) functions.

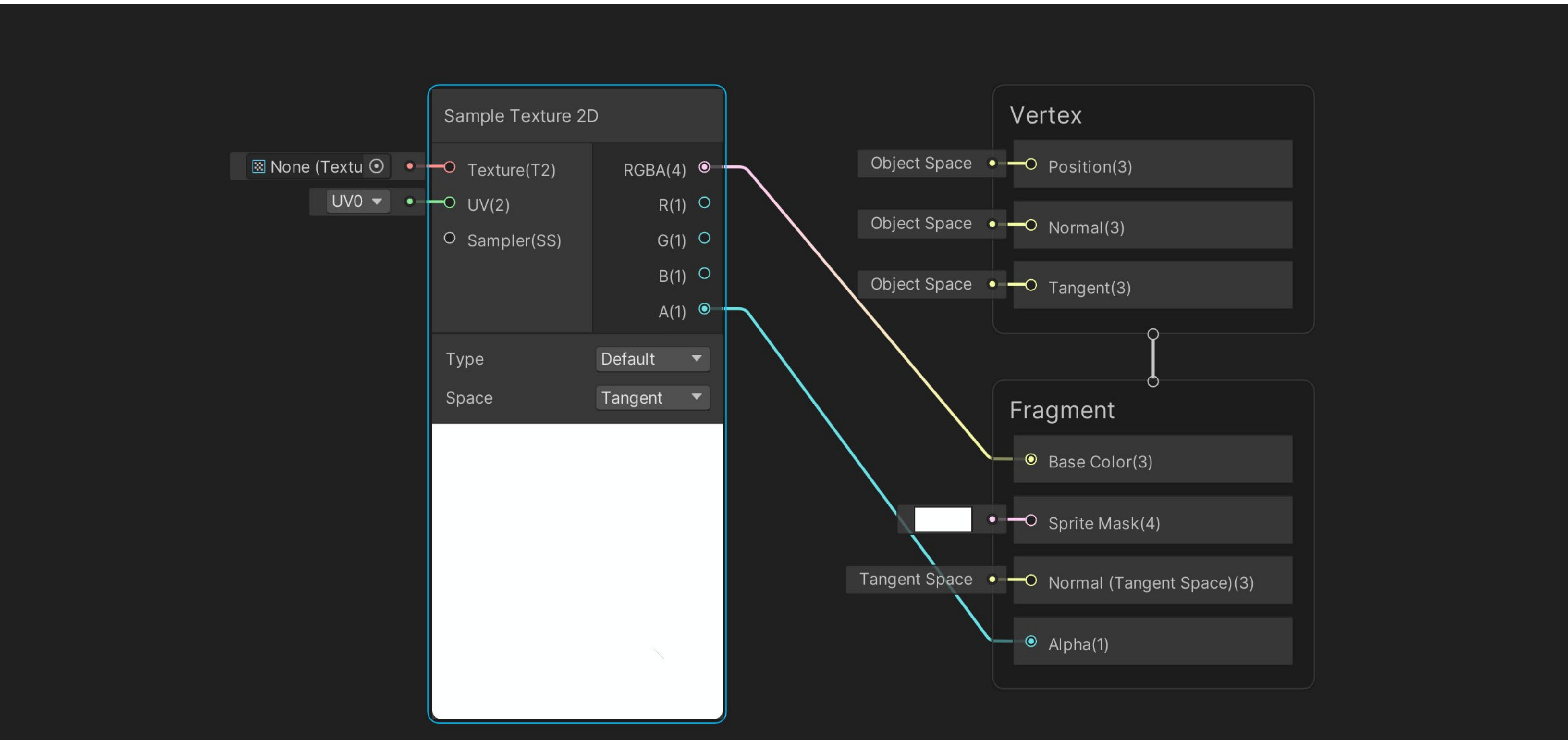
On the left hand, you'll see a Blackboard. This contains properties that will be visible in the Inspector and Keywords.

At the top right is a Graph Inspector, which has two tabs: Node settings, which allows you to change a selected Node's settings, and Graph Settings, which changes the entire graph's settings. From the Inspector, you can switch the Precision mode; this is a good idea to set it to half on low-end devices. You can also swap the shader Material between lit and unlit.

The Main Preview at the bottom is where you can view the final shader.

The nodes in Shader Graph are the building blocks that hold inputs, effects, and interactions. A graph proceeds from left to right, so nodes will have input slots on the left, and outputs on the right.

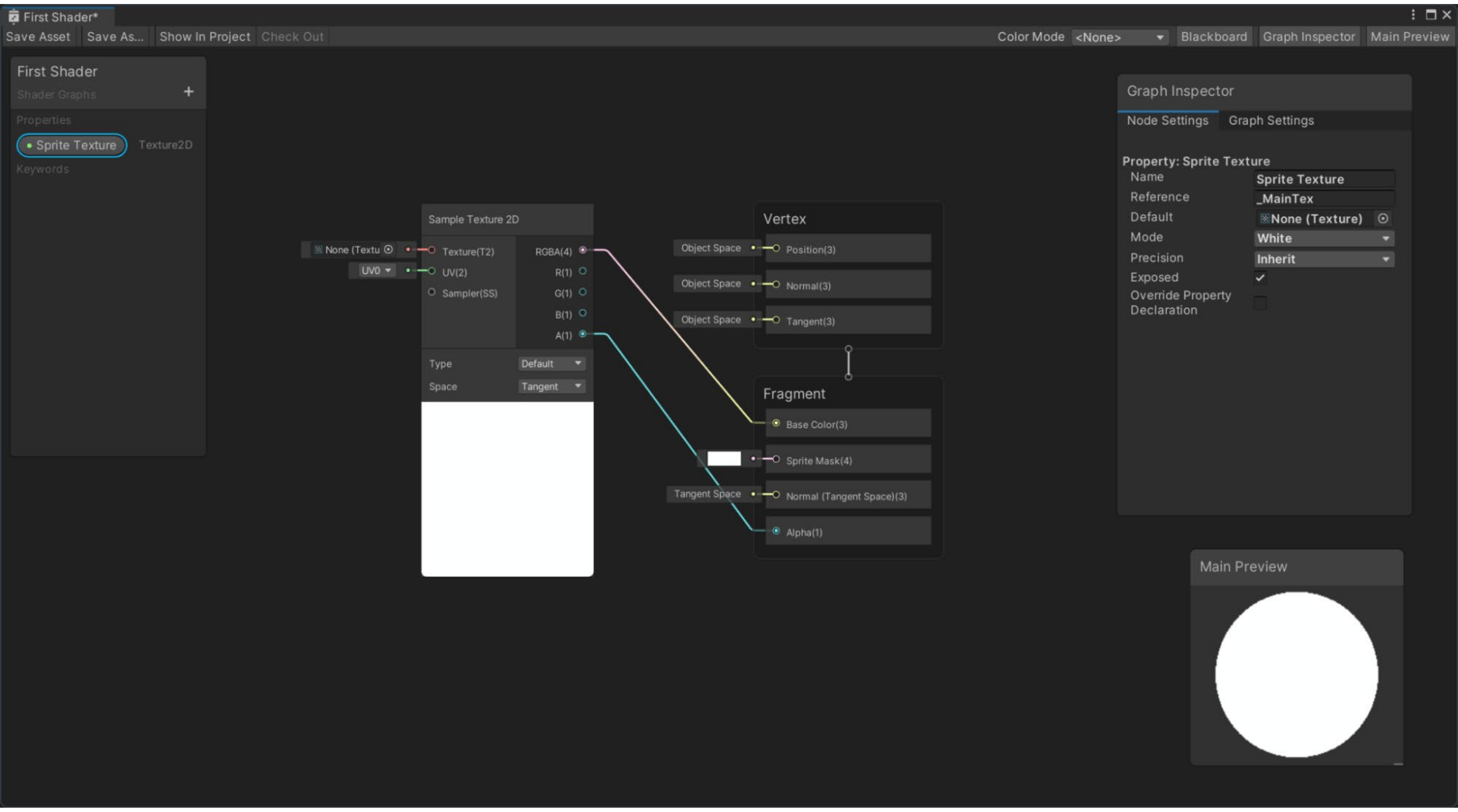
To add a new node, right-click anywhere on the workspace, and choose **Create Node** from the context menu. You'll be presented with a list of all nodes grouped by type. Select **Input > Texture > Sample Texture 2D**.



An example of a Texture 2D node

Connect the RGBA(4) slot of the Sample Texture 2D node to the Base Color(3) in the Master Stack's input slot. Connect the A(1) slot to Alpha(1). Now, the Master Stack's Fragment Context will use the color and alpha of the texture you provide to the Sample Texture 2D node.

To reference this texture from outside the shader, you need a property.



Property: Sprite Texture
Name: Sprite Texture
Reference: _MainTex
Default: None (Texture)
Mode: White
Precision: Inherit
Exposed: ✓
Override Property Declaration: ☐

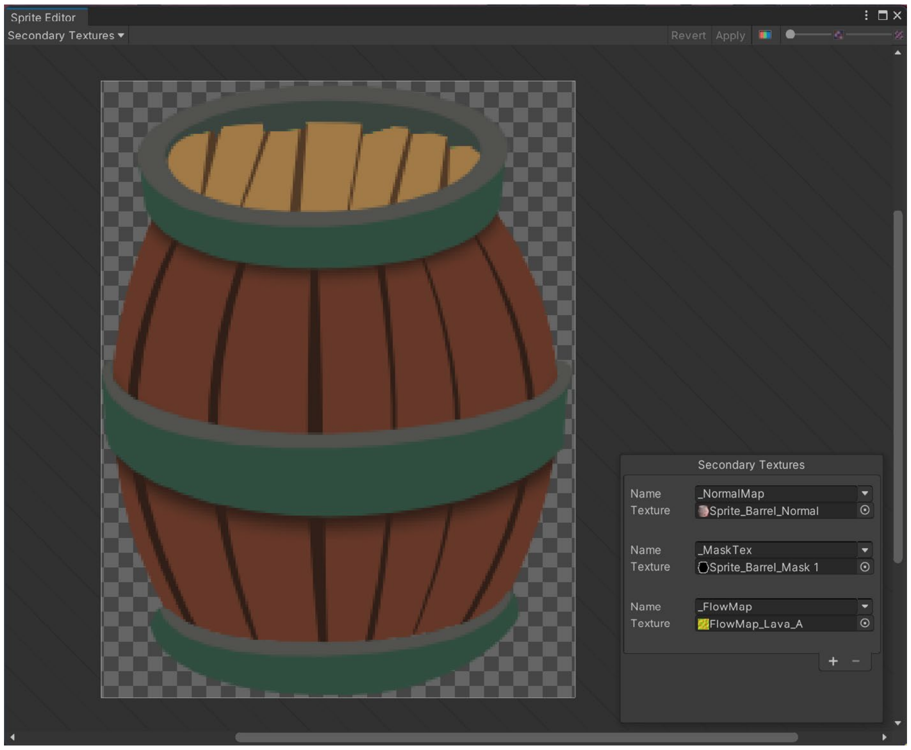
Exposed properties will show in the Inspector view where you can modify the parameters that will be used in the Graph

Click the Plus button located on the Blackboard, and select Texture 2D. Select the newly created property, and options will appear in the Graph Inspector. You can change the property's name, but more important is the Reference option – change this to _MainTex.

For the texture, use the sprite, which is set in every Sprite Renderer. Sprite Renderer will check for _MainTex property in the current Material's shader and will set its sprite texture as the texture in the shader. Finally, drag this property from the Blackboard to the workspace, and connect it to the Texture(T2) input slot on the Sample Texture 2D node. Hit the Save Asset button on the toolbar, and the shader will be compiled.

You can use the Shader Graph just like any other shader in Unity. Right-click in the Project window, choose **Create > Material**, and select your shader from the list. All of its properties will be available to customize.

You can test your shader by applying the created material to a sprite. Of course, this is the simplest shader with just a texture. Feel free to experiment and try out the rest of the nodes. Or maybe you can try to add a normal map and Mask Map (Sprite Mask) to this shader? Hints: Sample Texture 2D Type needs to be set to Normal to correctly display normal map, and you can check reference names of Normal Map and Mask Map textures in the Secondary Textures window in the Sprite Editor. Good luck!

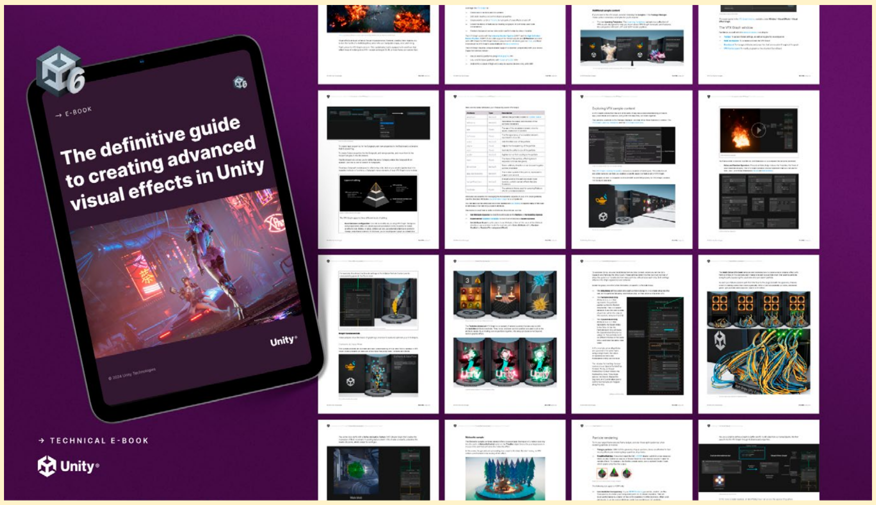


Secondary Textures

Name	_NormalMap
Texture	Sprite_Barrel_Normal
Name	_MaskTex
Texture	Sprite_Barrel_Mask 1
Name	_FlowMap
Texture	FlowMap_Lava_A

Secondary Textures for the barrel sprite. You can check their names and add your own textures that can be referenced in Shader Graph by name.

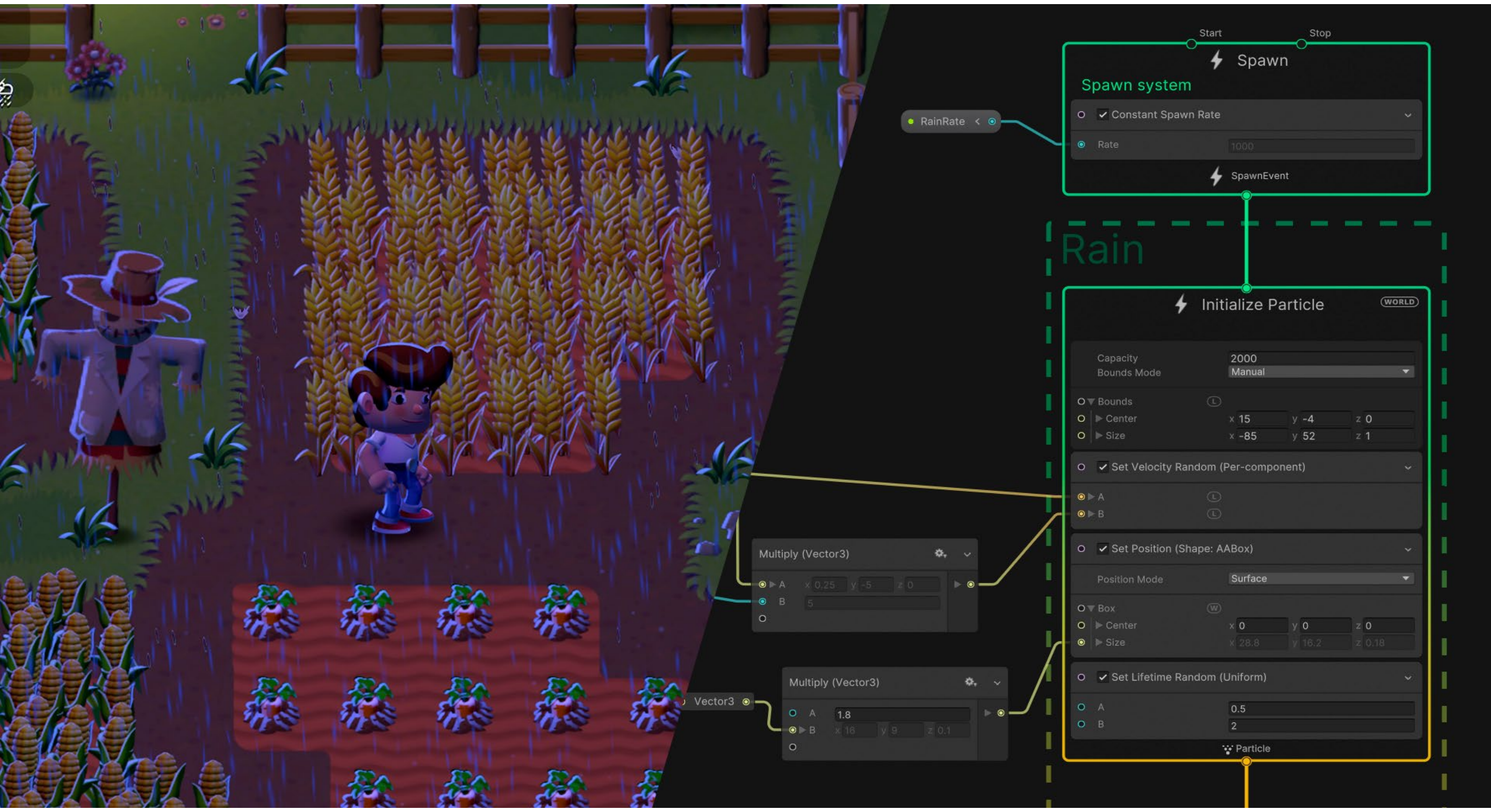
For a complete walkthrough of the many capabilities of the VFX Graph, download Unity's comprehensive guide to creating advanced visual effects in Unity 6:


E-BOOK
The definitive guide to creating advanced visual effects in Unity
TECHNICAL E-BOOK
Unity

Download the e-book

VFX in the Unity samples

The rain in *Happy Harvest*



Constant Spawn Rate
Rate 1000

Capacity 2000
Bounds Mode Manual
Bounds
Center x 15 y -4 z 0
Size x -85 y 52 z 1

Set Velocity Random (Per-component)
A
B

Set Position (Shape: AABox)
Position Mode Surface
Box
Center x 0 y 0 z 0
Size x 28.8 y 16.2 z 0.18

Set Lifetime Random (Uniform)
A 0.5
B 2

Multiply (Vector3)
A x 0.25 y -5 z 0
B 5

Multiply (Vector3)
A 1.8
B x 16 y 9 z 0.1

Setting up the rain effect with a VFX Graph in Happy Harvest

Rain is a common particle-based effect in games. Here are the main steps used to create the rain in *Happy Harvest* using VFX Graph.

1. The rain particles (named **Visual Effect Rain** in the Hierarchy) are rendered in the Sorting Layer above the ground, but behind objects. The reason for this is water splash effects spawn when the drop particles meet the ground to simulate how real rain reacts. If the rain drops are visible on a vertical surface like a lamp post, it could look incorrect.

   In the real world light affects all of its environment, including rain drops. In *Happy Harvest*, the VFX Graph for the rain effect uses a Sprite Lit-based Shader Graph in the Output Context to light the rain particles. Without a Lit Shader Graph match for the rain, its tone can't change to reflect the changing light throughout the day.

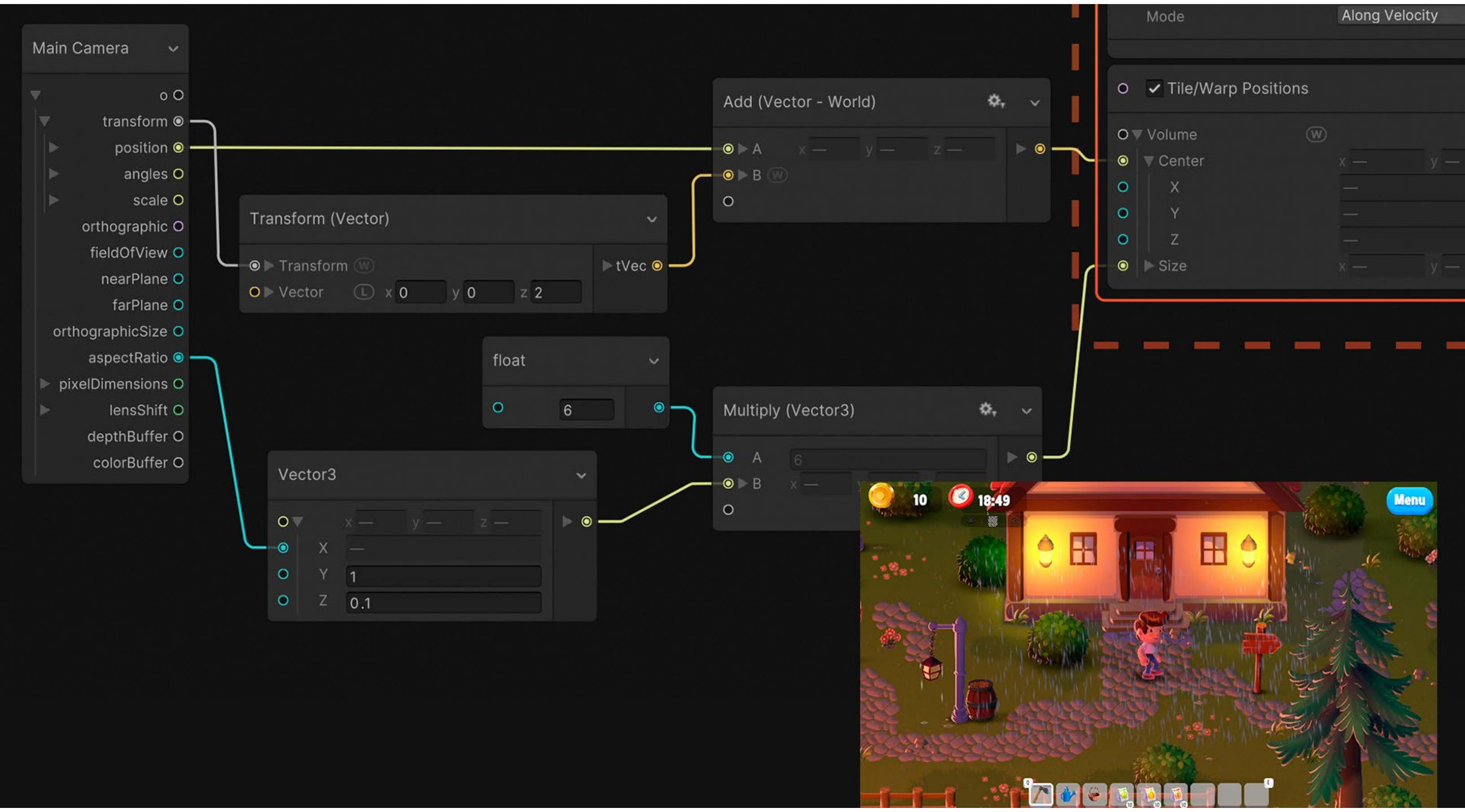
   Frame-by-frame splashes: When the rain particle dies, a Trigger On Die Block triggers a GPU event that spawns a single flipbook particle that plays a simple splash animation. Happy Harvest uses the Flipbook Shader Graph UV block to display the different frames inside the texture. Alternatively, you can use the Flipbook Player block inside VFX Graph to achieve the same effect.

2. Rain particles appear in the foreground (named **Visual Effect Rain Foreground** in the Hierarchy); since they simulate being close to the camera lens they don't spawn a water splash upon hitting the ground.

3. A variation of the previous effect is used with the flashing thunder effect enabled (**Visual Effect Rain Foreground Thunder**). A sound effect is added for the thunder with a VFX Output Event Play Audio component. This component comes with the sample scripts of Output Event Handlers; the Output Event Context triggers it in the graph.

   A flash of thunder: A simple but effective way to create this effect was to spawn a VFX Graph-based single particle initialized at the center of the camera position and big enough to fill up the screen. The Color Over Lifetime block in the Output node changes the color of the particle by evaluating a gradient over time. In that gradient, transparency changes simulate the flashing and fading of the thunder flash.

Optimization with camera tiling



Transform (Vector): Vector x 0, y 0, z 2

float: 6

Vector3: X —, Y 1, Z 0.1

Adjusting the particles' AABox size multiplier to make them fit on camera

The rain affects the entire scene in *Happy Harvest*, but because only a portion of it is visible through the camera, many of the calculated particles are unnecessary. You can check the **Frustum Culling** option in the Inspector to avoid rendering those particles, or use camera tiling, a trick employed in the demo.

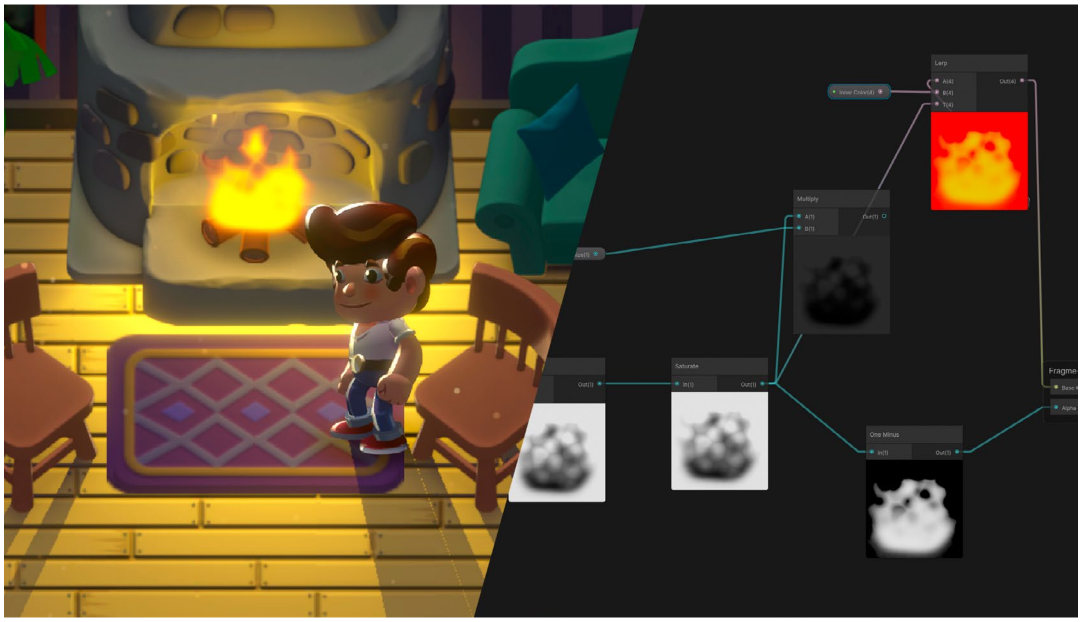
Camera tilling allows you to create and render particles in a box that matches your Scene view, giving the impression that it tiles as if the particles are simulated everywhere in the scene. No particles are created outside of the camera, allowing you to reduce the amount of generated particles and capacity allocation of the effect without additional settings.

To use camera tiling in *Happy Harvest*, the rain spawn is set to match camera dimensions using a Set position shape: AABox. The camera format (enter the size manually, use the MainCamera directly, or use the aspectRatio slot) is scaled by a size multiplier to match the camera settings in the scene. The AABox is slightly larger than the actual camera size to avoid a popping effect caused by the culling. The Shape Spawn mode is set to Surface because there's no need to spawn on the entire volume in a 2D project (Z axis is not used).

Once this setup is complete you need a Tile/Warp Positions Block in the system's output, which is linked to the camera Transform to match its position and the camera dimensions, as you would also want to do in the Initialize Context of the system.

This technique also applies to rain effects in 3D, with the exception that Shape Spawn mode should be set to Volume and with a value other than zero for Z axis if you want depth look.

A cozy fire



The farmer enjoys a VFX Graph-made fire at the end of his busy day.

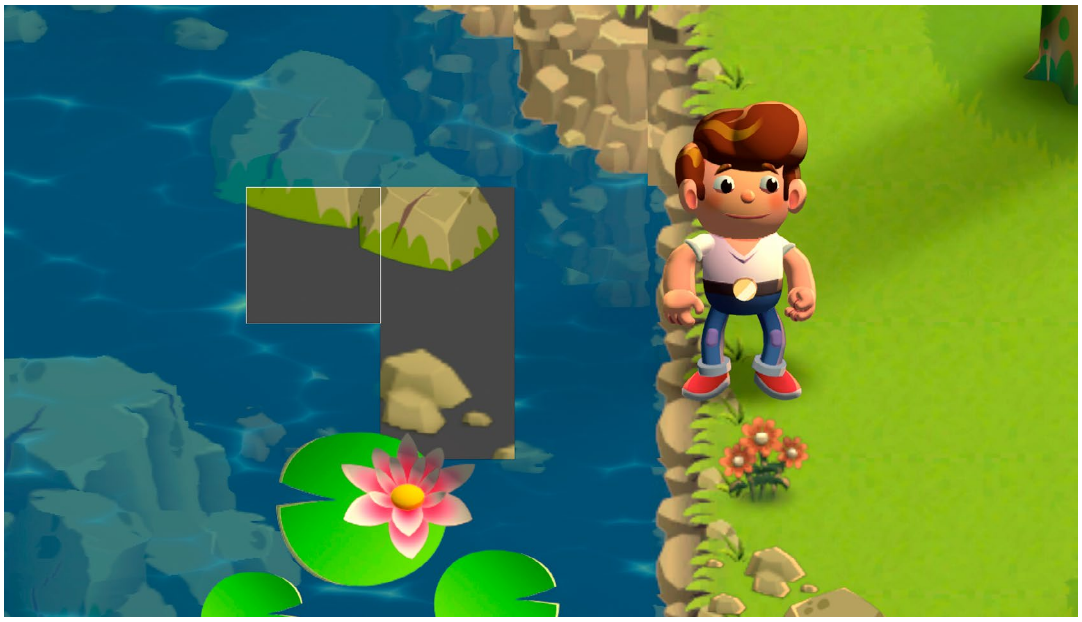
Another common effect in games is fire. In *Happy Harvest*, there's a cozy bonfire in the yard and in the fireplace in the farmer's house. The fire effect consists of three sets of particles: Flames, sparks, and smoke, all spawned from the same graph called **VFX_Fire**.

Animating the flame

The smooth animation of the flame happens at the shader level in Shader Graph and shows what you can achieve by using the power of the GPU and shaders.

If you open the **ShaderGraph_Fire** file in the project, you'll notice that a single sprite (**Mask texture2D**) is used to create a mask to give the flame an oval shape. The animation and sprite is created via a scrolling "Voronoi" pattern with noise and a tint color for the flame and the background. The alpha mask uses the same flame shape but with a Saturate Operator that produces a more visible outline.
Sparks and smoke effects, which add polished details to the fire, use the standard Lit Shader Graph. A combination of alterations on the Velocity, Scale and Initial Position Blocks, creates a lively effect with only one sprite required. The graph for the sparks and smoke effects includes notes to help you recreate the effects in your own project.

Water tiles animation


The effects added to water help it to match the cheery, cartoon style of the demo.

The water uses cartoon-like caustics or waves with transparency effects so it matches the overall art style in the demo. The water shader is optimized to work well with tilemaps, from which most of the demo's environment is made.

The default 2D Lit material, called **Water,** is found in the Tilemap Renderer of the tilemap. This has been swapped for a custom material with the water shader. All of the tiles inside a tilemap are rendered using the same material, so this tilemap is only used for the water tiles.

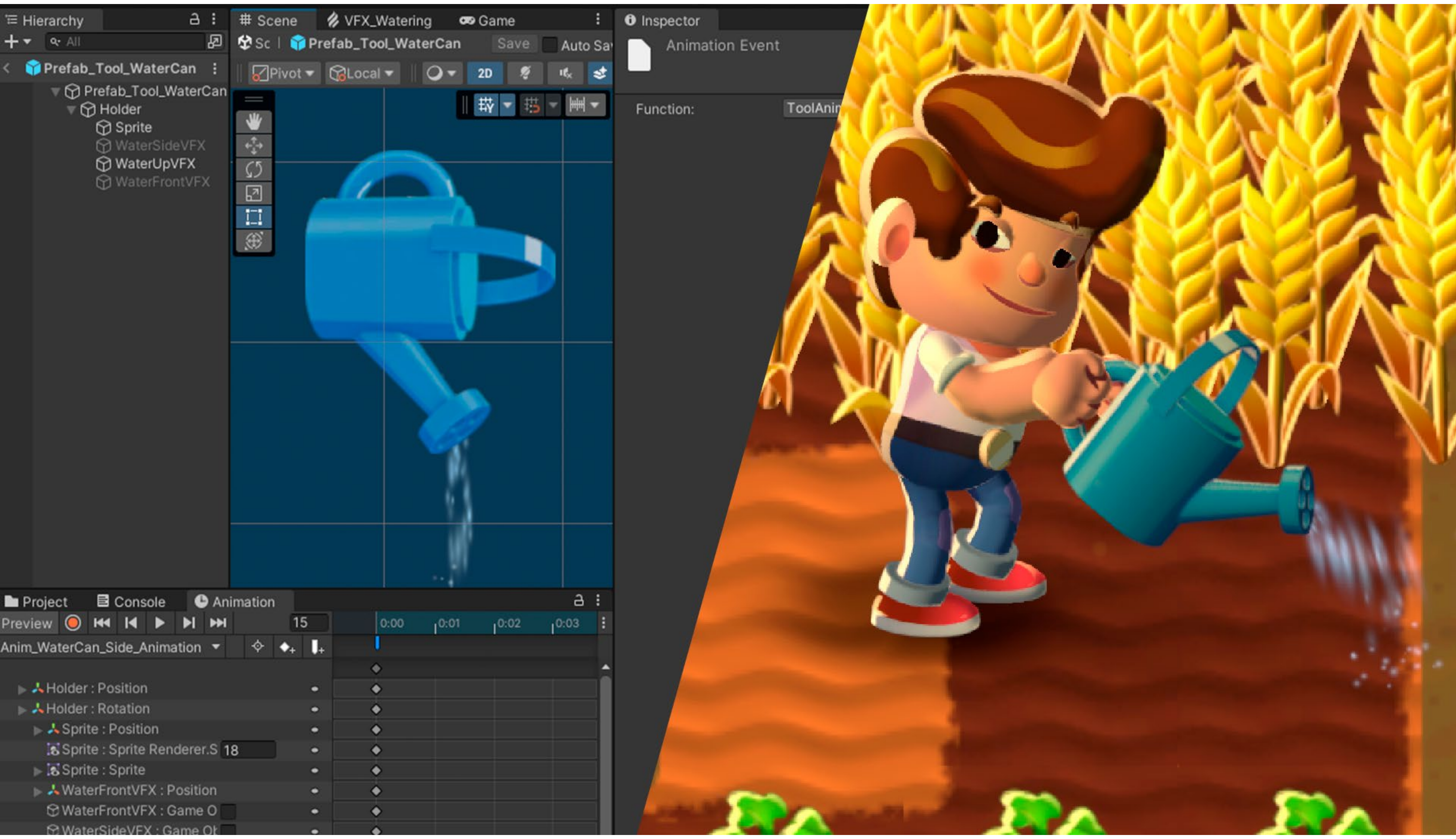
In the **ShaderGraph_Water** graph the water effects are also generated from a stretched inverted Voronoi noise pattern to simulate the caustics (a common method in water caustics). The oscillating motion is a product of a tiling and offset UV overtime with a noise effect applied to it.

Refraction effect

The refraction effect is blended into the surface texture with the Lerp node. The refraction uses the Camera Sorting Layer texture, which is a snapshot of the camera rendering up until the defined sorting layer, meaning that all the elements in previous sorting layers will appear in that texture. In *Happy Harvest*, rendering occurs up until the **Objects** sorting layer, which is the one used for elements like the character. The UV of the Camera Sorting Layer texture is distorted with a normal map generated with the Normal and Height node, which uses the Voronoi texture that's also used for the surface texture.

Watch this tutorial using the Lost Crypt 2D demo to get a step by step tutorial on other water effects in 2D

VFX in gameplay



The water can prefab has three animation clips attached to it that change the sprite based on the direction the character is facing.

The effects discussed so far don't interact with the gameplay in *Happy Harvest*; they are a part of the environment and play regardless of what's happening in the demo. However, in most games, you'll need to trigger some of the effects to play based on when an action occurs in the environment and/or with the character. Let's look at some triggered effects in *Happy Harvest.*

Effects when using items

Some effects are attached to the character tools like the water can. The prefab for the water can, **Prefab_Tool_WaterCan**, has three animation clips that change the sprite based on the direction the character faces (up, down, sides). In those animations, there's an animation event that calls a function inside a Tool Animation Event Handler component.

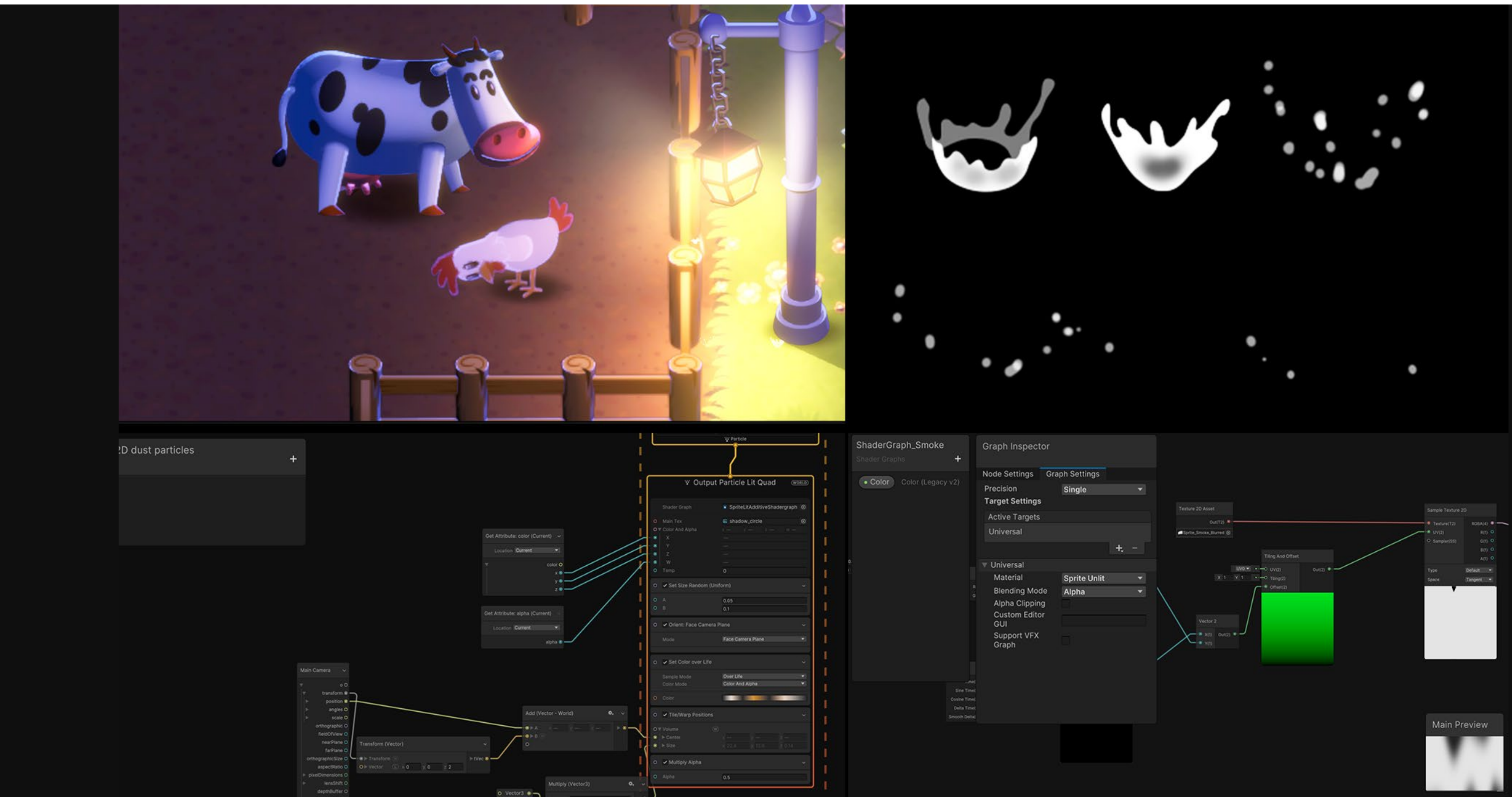
This script enables or disables the corresponding instances of VFX Graphs for each position that are referenced from the Inspector.

Effects when iterating with tiles

Some effects happen on the tiles, like digging and harvesting. This ensures they don't cause issues with the character animations or change of position. When it's time to till a cell, the tilling effect plays on that cell, and when it's time to harvest, the cell calls on the harvest effect.

When the tilling effect is triggered, it references the **TillingPuff_prefab** effect from the **TerrainManager** component attached to the **Grid** GameObject. It moves to the center of the tile and then plays. Other effects like crop harvesting call the VFX Graph that is referenced from the **Data/Crops** ScriptableObject. At the initialization of the **TerrainManager** script, a VFX pool dictionary is created (the variable in the script **m_HarvestEffectPool**) that instantiates several VFX prefabs to play when necessary.

Small touches that make a difference



Small effects in *Happy Harvest* add polish to the demo and make it more enjoyable to explore. Some of these small effects include:

— **VFX_DustParticles**: To spread ambient dust in the environment

— **VFX/Water**: For subtle water waves and splashes

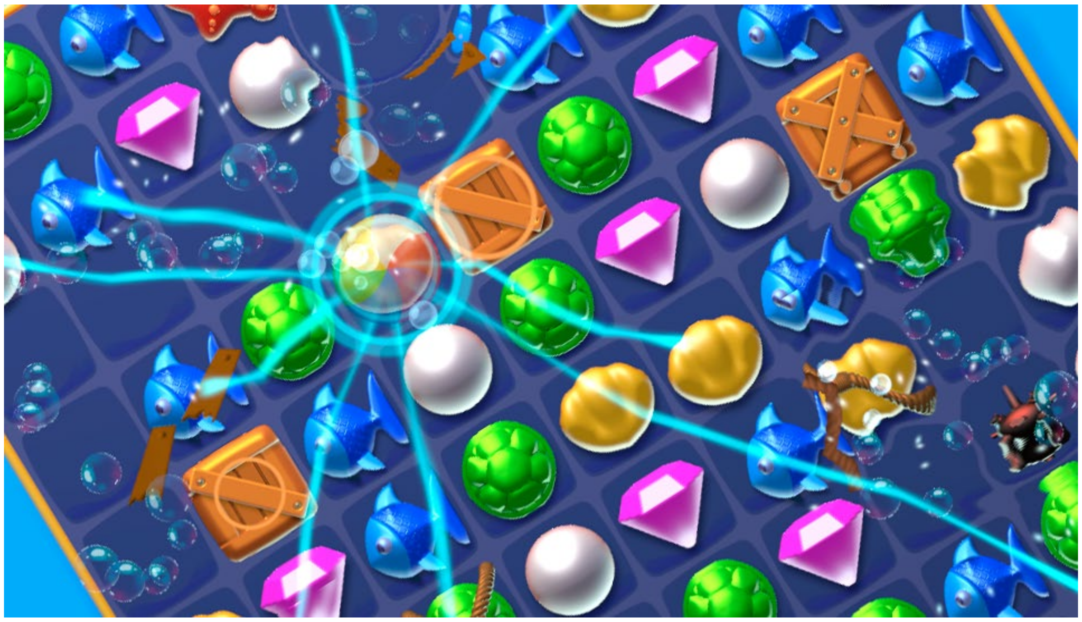
— **VFX_Leaves**: Triggers leaves falling from bushes when the player passes by

— **Character**: Triggers a smoke puff effect attached to the character; this is made with the Built-In Particle System

— **P_VFX_Moths**: Triggers moths flying near spotlights; this is also made with the Built-In Particle System

Each of the VFX Graph-based effects in the list above include notes explaining the purpose of each node.

VFX in Gem Hunter Match



The VFX Graph- and Shader Graph-based visual effects in *Gem Hunter Match* include explanatory comments in each of their graphs, to help you understand how each system works together with 2D lights.

The special effects in Gem Hunter Match include:

— **VFX Graph-based effects that display an unlit particle:** This effect uses the Output Particle Quad.

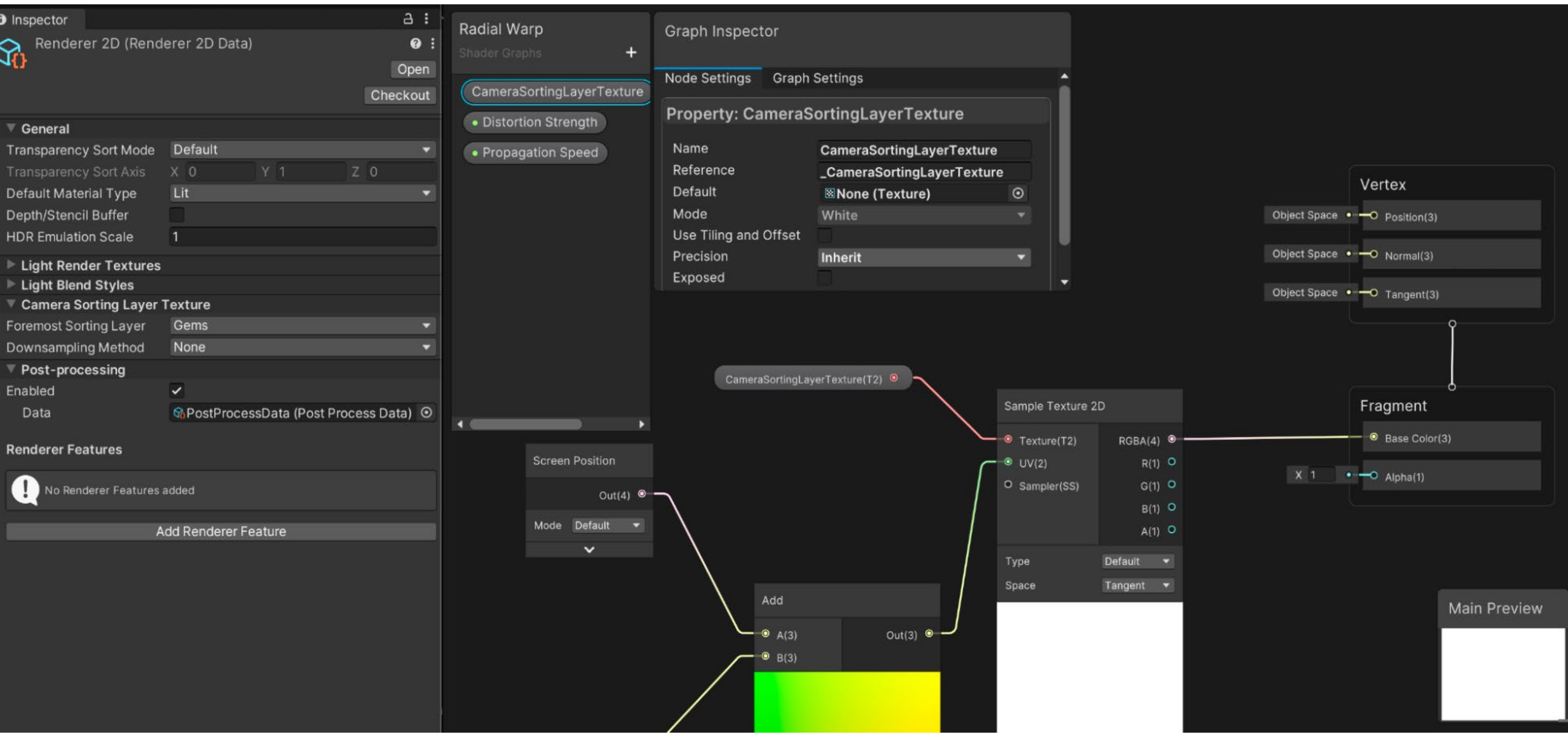
— **VFX Graph-based particle effects that use a custom shader**: These use a shader graph to render the particles. For example, the effect called **VFX_CellHoldDistortion** uses a 2D shader in the output with the **Support VFX Graph** option enabled in the shader. When you add the 2D Shader Graph in the Output node, the output becomes one of the following:

- **Output Particle Sprite Unlit Quad:** When the Master node in the Shader Graph is Sprite Unlit, you can still apply effects to the particle but it won't react to 2D Lights.
- **Output Particle Sprite Lit Quad**: When the Master node in the Shader Graph is Sprite Lit, it will react to 2D lights.

Polished-looking effects are usually a coordinated sequence of effects smoothly linked together with several Spawn nodes that spawn in specific moments with GPU events. When you create your own particle effects, make sure to use the right Sorting Layer in the Inspector for each VFX Graph

Camera Sorting Layer Texture

The **Radial Warp** shader uses the URP 2D Camera Sorting Layer Texture setting. This handy feature gives you access to the graphics generated up to the indicated Sorting Layer in the URP 2D Renderer settings, which you can then use in Shader Graph to apply effects. In the *Happy Harvest* sample the Camera Sorting Layer Texture is used to create a water refraction effect, and in *Dragon Crashers* it's used for smoke distortion. In this sample we use it to apply a distortion that simulates a shockwave, adding extra visual appeal when you make a match.



Renderer 2D (Renderer 2D Data)

General	
Transparency Sort Mode	Default
Transparency Sort Axis	X 0 Y 1 Z 0
Default Material Type	Lit
Depth/Stencil Buffer	☐
HDR Emulation Scale	1
Light Render Textures	
Light Blend Styles	
Camera Sorting Layer Texture	
Foremost Sorting Layer	Gems
Downsampling Method	None
Post-processing	
Enabled	✓
Data	PostProcessData (Post Process Data)

Renderer Features

No Renderer Features added

Add Renderer Feature

Property: CameraSortingLayerTexture

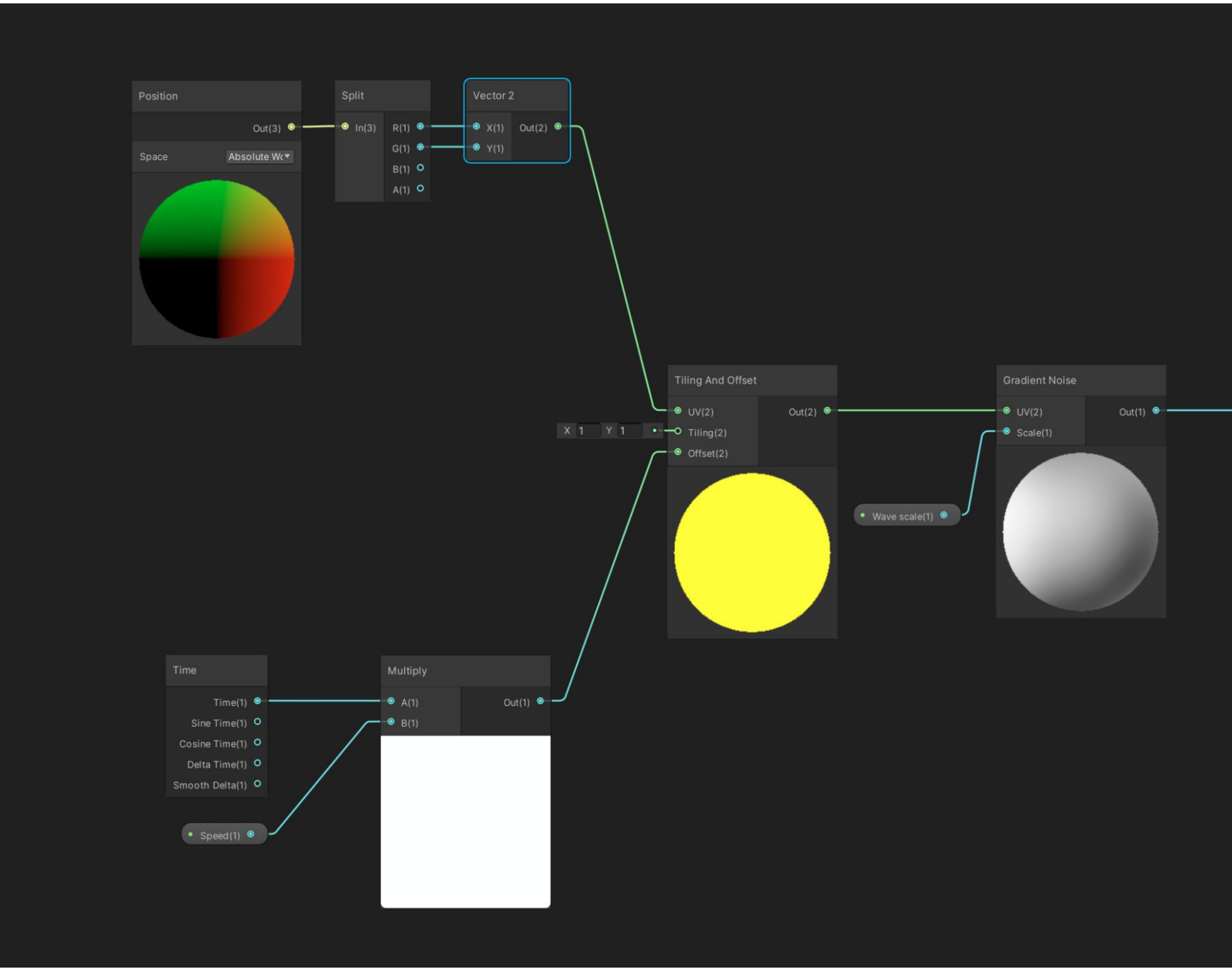
Name	CameraSortingLayerTexture
Reference	_CameraSortingLayerTexture
Default	None (Texture)
Mode	White
Use Tiling and Offset	☐
Precision	Inherit
Exposed	☐

The Renderer 2D Data Asset, far left, and the Radial Warp shader read the onscreen image as a texture for the Base Color of the shader, and then apply deformation in the UV

Shader Graph examples from *Dragon Crashers*

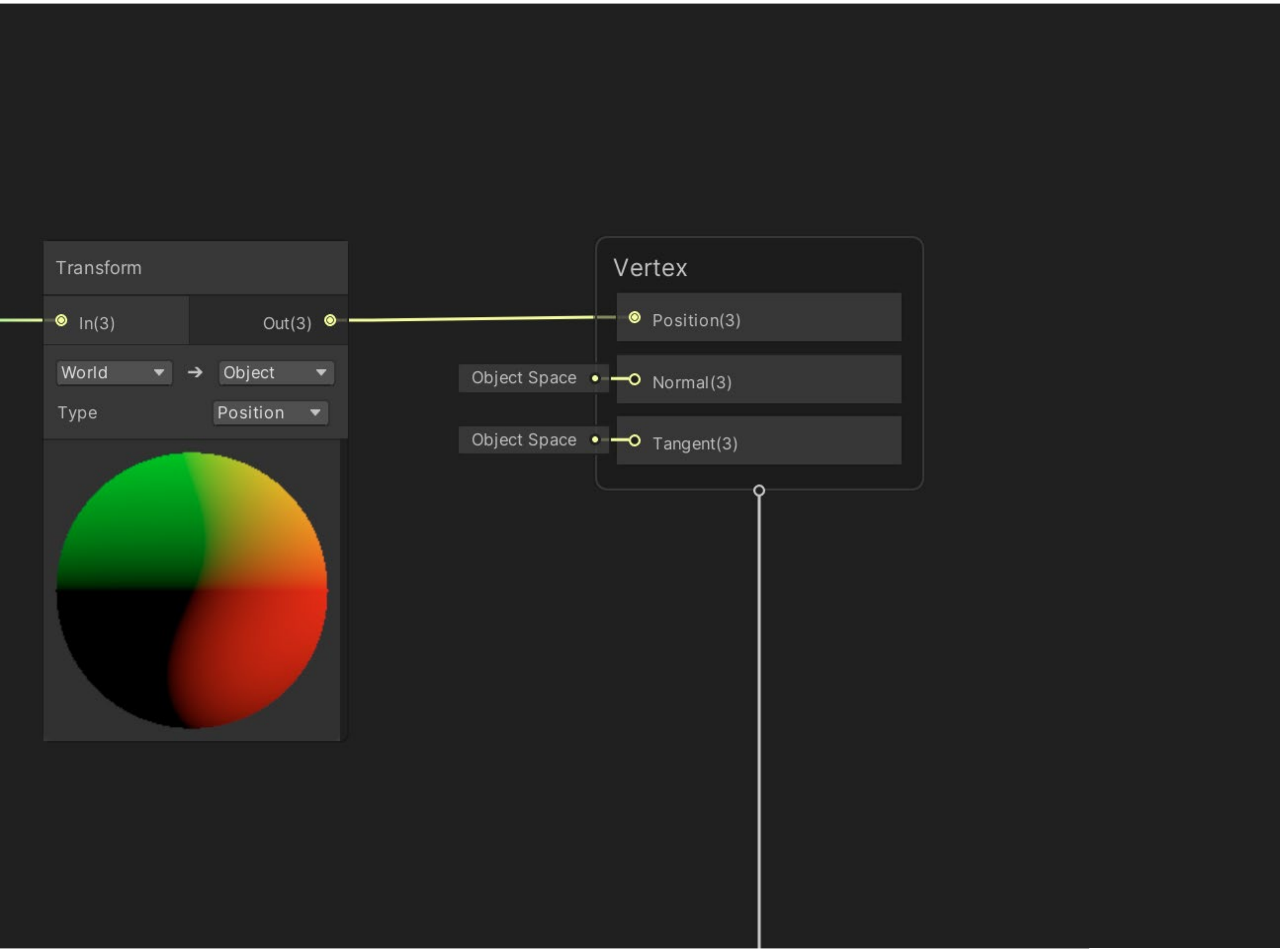
Vertex displacement

The Vertex displacement shader affects the positions of vertices in a geometry of the mesh. You can displace vertices in many ways, and the simplest way is to apply a moving noise. This shader is good for waving objects: foliage, hanging vines and ropes, flags, and so on. You can find an example of this Shader Graph file in the *Dragon Crashers* project, with the file name *ShaderGraph_Sprite_Lit_Waving.*



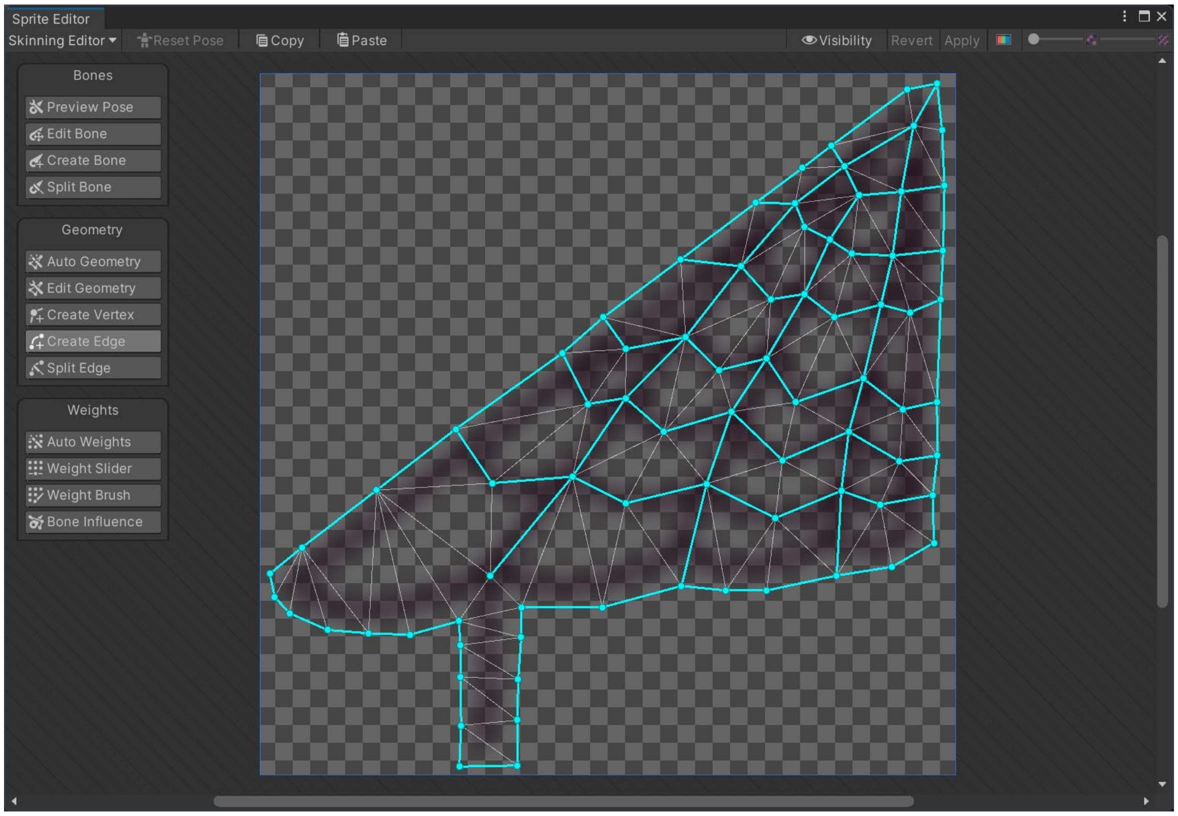
Main part of Waving shader

To achieve the wave animation, Gradient Noise is used as a wave mask. Its UV Offset is animated by the Time node multiplied by a Float named Speed. These all affect the position of vertices in the Vertex Context.



The Vertex context in Shader Graph modifies the vertices position and the Fragment master stack the pixel information.

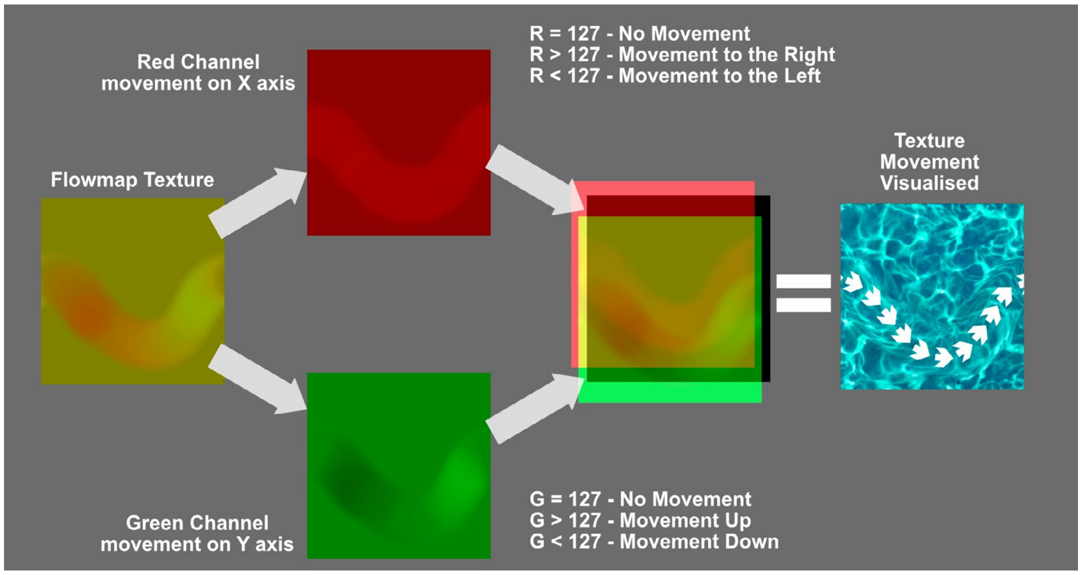
The sprite that will be affected by the Waving shader needs to have a sufficient vertex count, otherwise the animation will look rough. To edit the vertices, use Skinning Editor Geometry tools.



Cobweb shader in action (click through to see the animation).

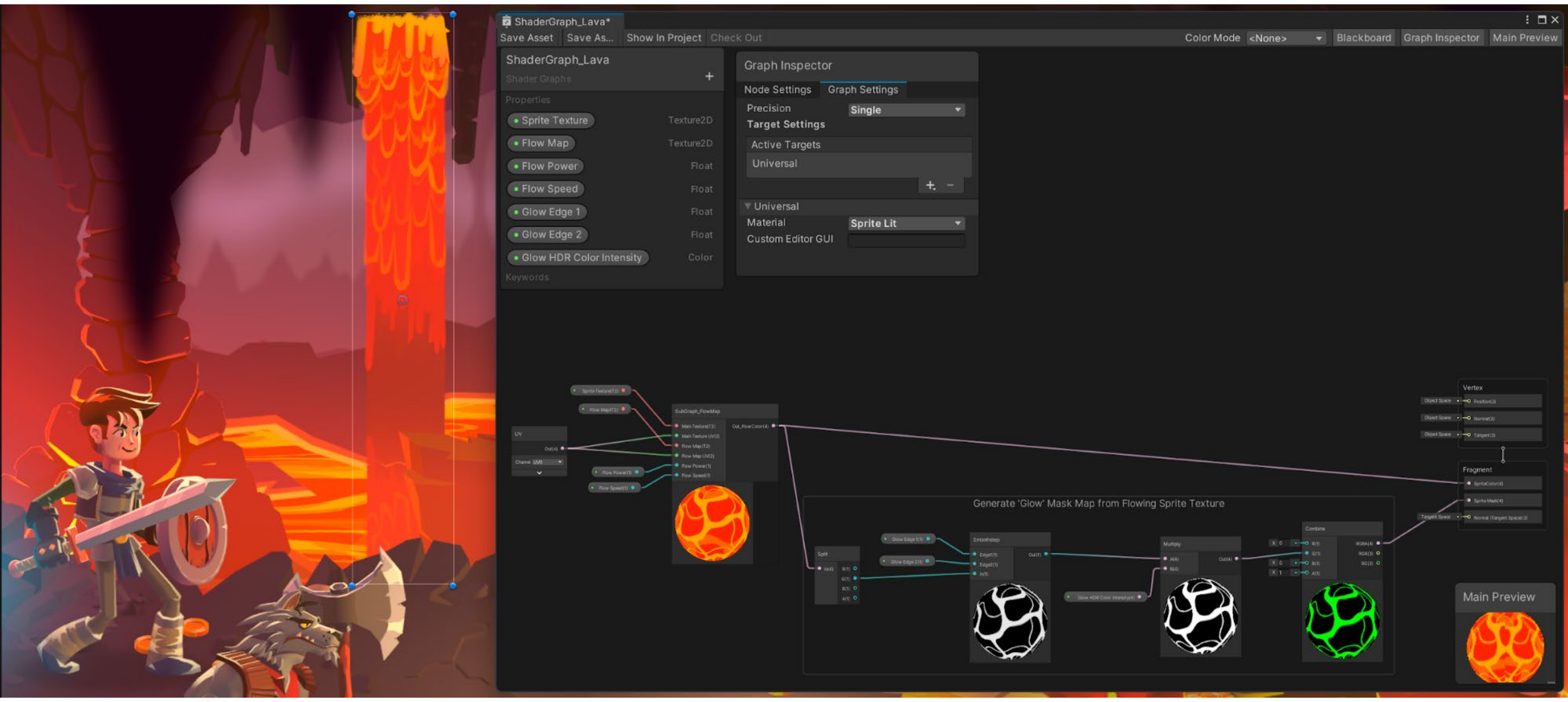
Flow maps

Flow maps are textures that store directional information. You can find a flow map shader in the *Dragon Crashers* demo, with the name *ShaderGraph_Lava.* The shader uses the flow map texture to control the direction of the main texture's UV coordinates. The colors red and green are used to indicate the XY direction that pixels follow in every frame, making the pixels of the main texture "flow."


Red Channel movement on X axis
R = 127 - No Movement
R > 127 - Movement to the Right
R < 127 - Movement to the Left
Flowmap Texture
Texture Movement Visualised
Green Channel movement on Y axis
G = 127 - No Movement
G > 127 - Movement Up
G < 127 - Movement Down

Flow map texture colors explained: The red color controls pixel movement on X axis, green color determines on the Y axis. The shader moves the pixels of the main texture in the directions that are visualized by the arrows.

Open the SubGraph FlowMap to learn how to achieve this effect.



ShaderGraph_Lava
Shader Graphs

Properties

Sprite Texture	Texture2D
Flow Map	Texture2D
Flow Power	Float
Flow Speed	Float
Glow Edge 1	Float
Glow Edge 2	Float
Glow HDR Color Intensity	Color

Keywords

Precision	Single

Target Settings

Active Targets
Universal

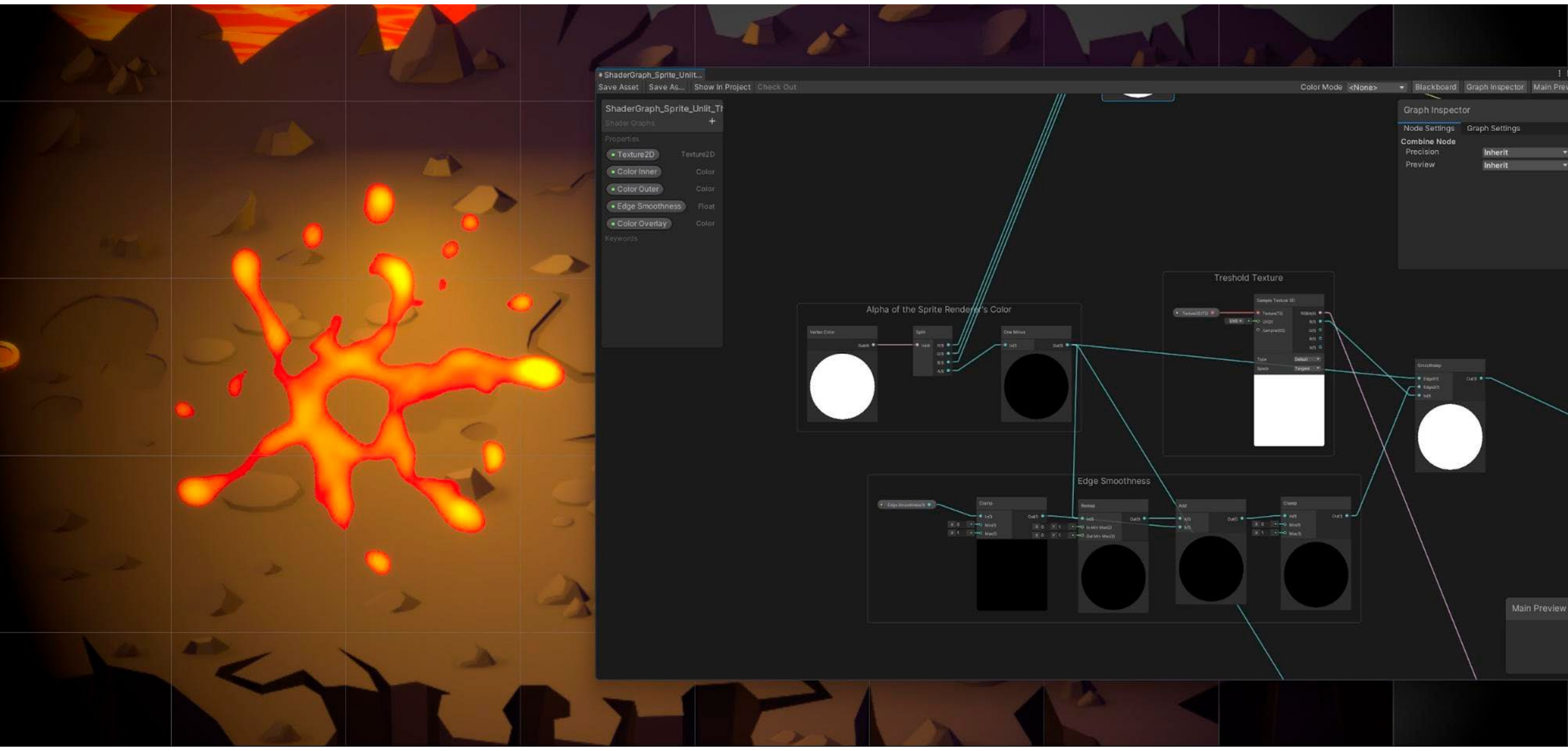
Universal

Material	Sprite Lit
Custom Editor GUI	

The hand-painted flow map gives the stream of lava a viscous and cartoonish effect, which matches the project's art direction. A couple of tools you can use for flow map creation include Flow Map Generator / Visualizer and Flow map painter.

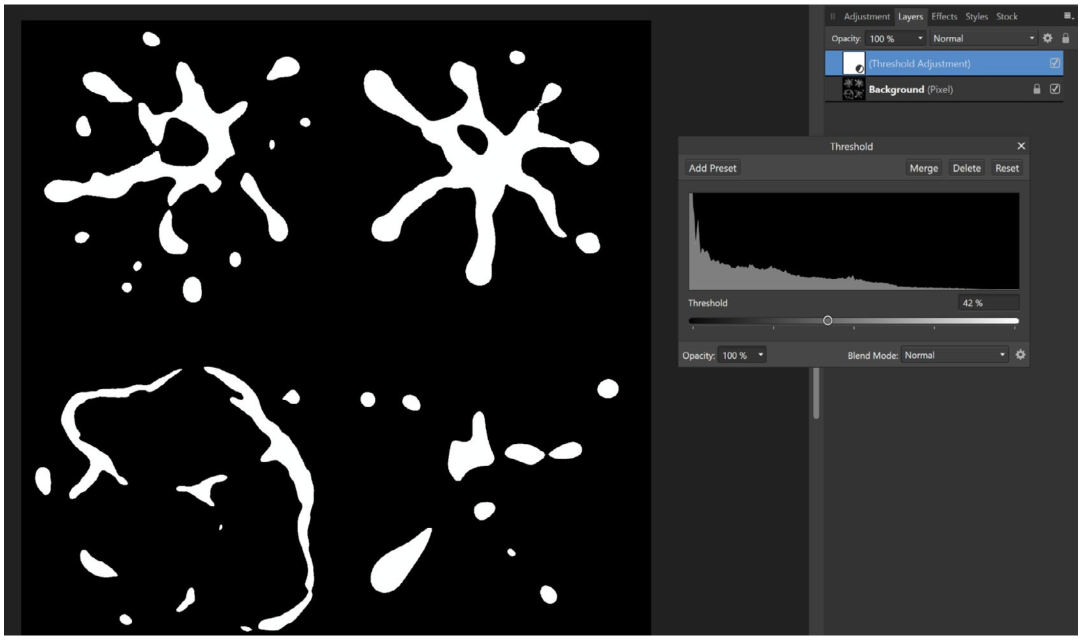
Threshold animations for fluid prop animation

Another shader animation technique you can try is called animated alpha clipping, which creates smooth animation from a single texture. This occurs by showing a specific range of pixels in each frame based on their alpha values. The texture is single channel, so the effect is simple to achieve. Find an example of it in *Dragon Crashers* with the name *ShaderGraph_Sprite_Unlit_ThresholdAnim*.

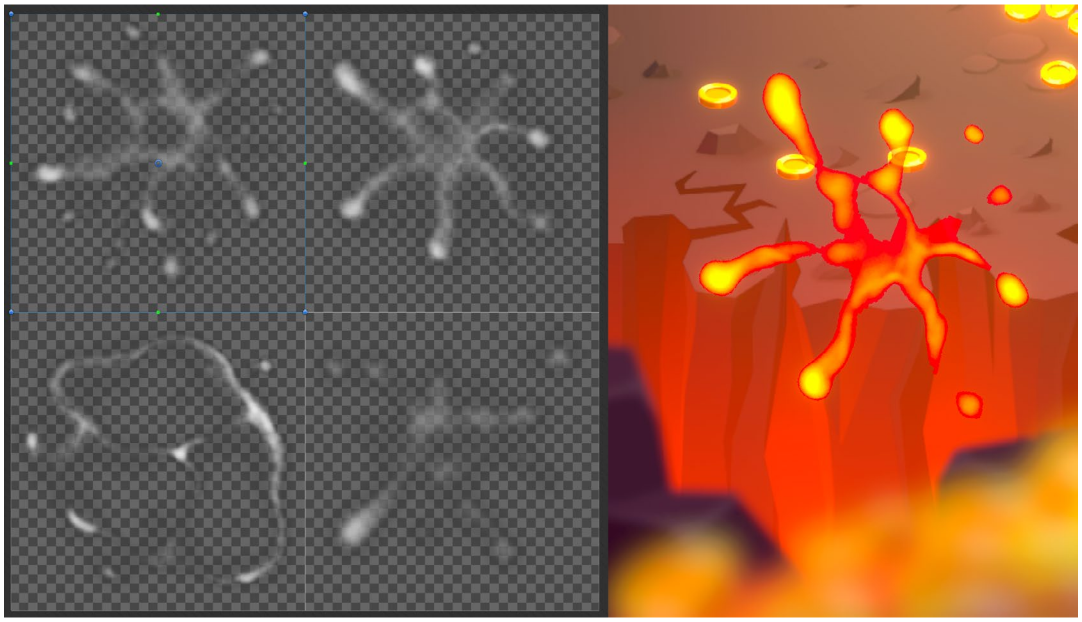


To achieve this effect, a Smoothstep node is used. The texture's red channel is connected to the In(1) slot, and Vertex Color alpha controls the evaluation of the edge and smoothness of the Smoothstep effect. Vertex Color is controlled by the Color property of the Sprite Renderer, so when you change the opacity of the sprite, the texture is animated. This allows the shader to be usable, even on particles.

If you use Affinity Photo, you can paint the texture for this shader black and white, then preview the animation by using Threshold Adjustment while moving the value slider.



Previewing threshold animation in Affinity Photo

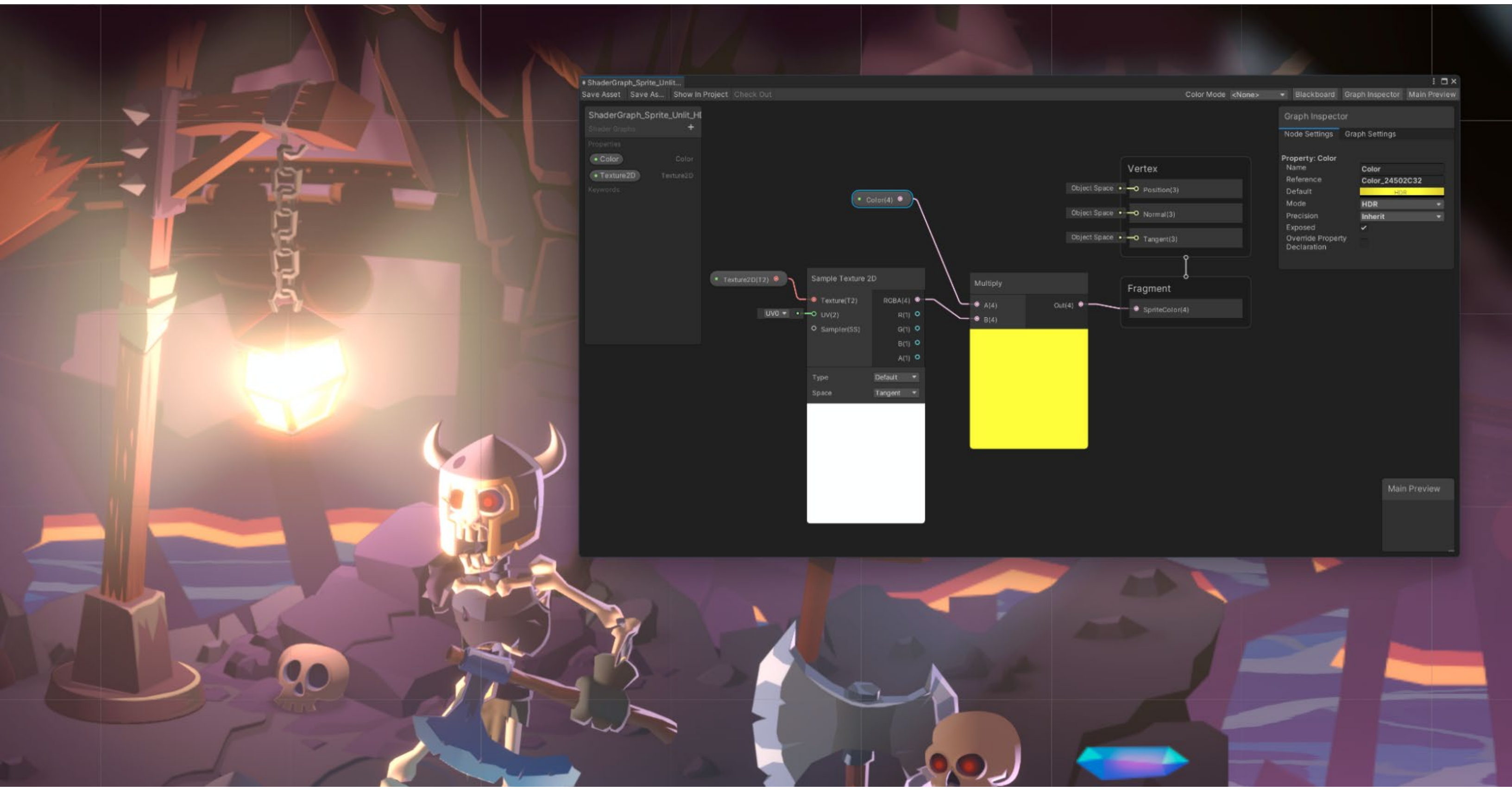


Texture used for the lava effect and how it looks in-game (see how it looks animated here)

This effect achieves the look of fluids or splatters, without the need for frame-by-frame animation. You can learn more about this animation in the Surface Tension community post.

Adding glow around lights

Another effect in *Dragon Crashers* is the intense glow coming from the lanterns. This is created by combining the HDR enhanced sprite shader with the Bloom post-processing effect.



Graph Inspector
Node Settings | Graph Settings

Property: Color

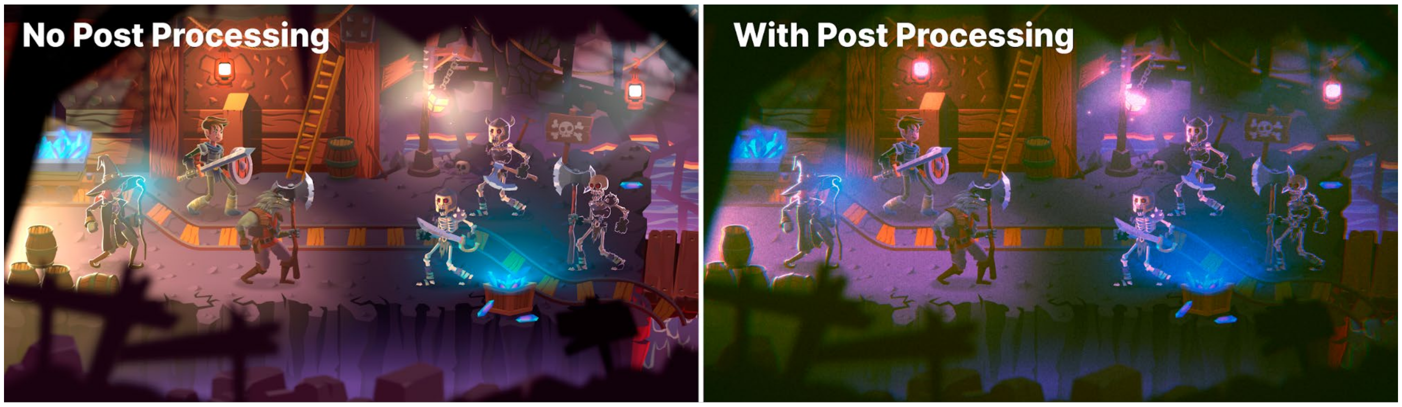
Name	Color
Reference	Color_24502C32
Default	HDR
Mode	HDR
Precision	Inherit
Exposed	✓
Override Property Declaration	

HDR shader in Dragon Crashers (ShaderGraph_Sprite_Unlit_HDRTint)

To make this effect we multiply texture by a Color node that uses HDR Color Mode. In the Material, we crank up the HDR value over 1, with the Bloom post-processing effect enabled on the main camera.

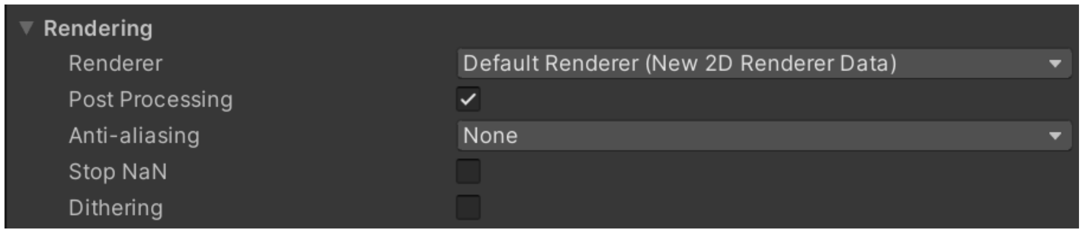
Post-processing

Post-processing enhances the look of your game by adding effects on the final image frame. These effects can simulate a physical video camera look or be completely stylized. URP includes post-processing capabilities, so there's no need to install a separate package, and configuring the effects is straight-forward.



On the left is an image of a camera output without post-processing. On the right is the same image with post-processing added. The effects have been exaggerated to show the difference. Apply post-processing effects cautiously, especially in mobile games.

To begin, select the main camera, and select the Post Processing option.

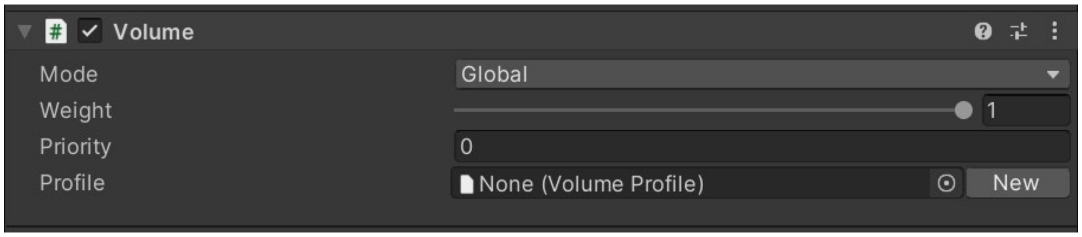


Rendering

Renderer	Default Renderer (New 2D Renderer Data)
Post Processing	✓
Anti-aliasing	None
Stop NaN	
Dithering	

Enabling Post Processing in the main camera

Post-processing uses the Volume system so to add a new post-processing Volume, select **GameObject > Volume > Global Volume.**



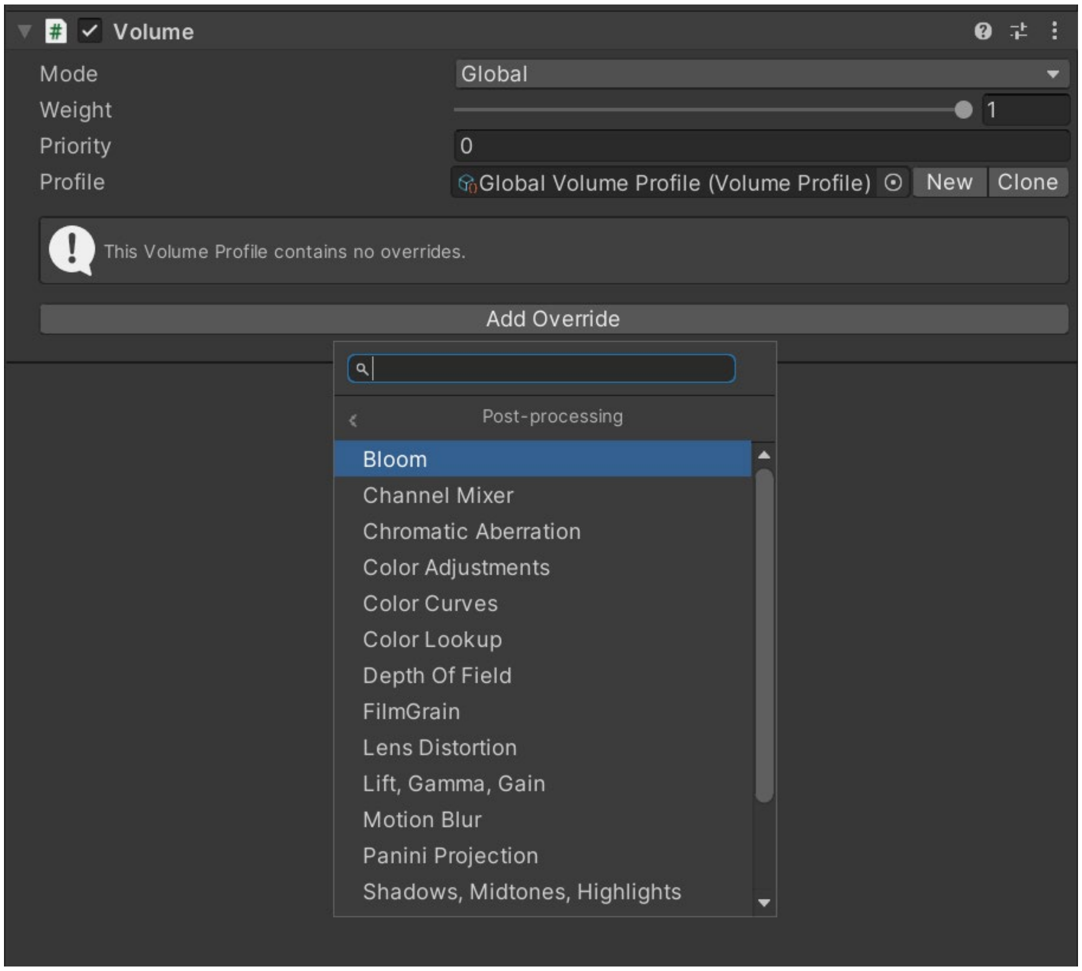
Volume

Mode	Global
Weight	1
Priority	0
Profile	None (Volume Profile)

Defining a Volume Profile

Now, create a Volume Profile by clicking the New button. Volume Profile will store your post-processing effects and can be easily shared between Volumes and Scenes.

To start adding post-processing effects, click the Add Override button, then select a desired effect from the list.

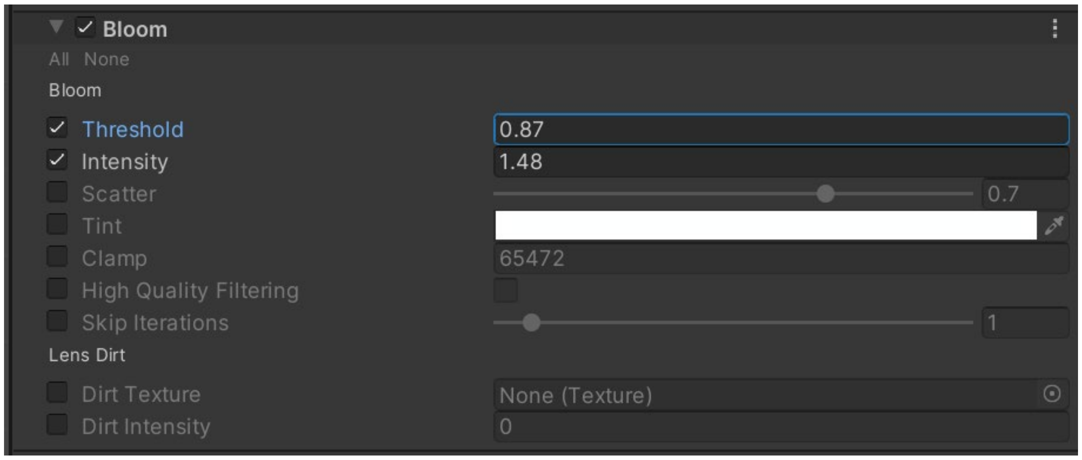

Volume

Mode	Global
Weight	1
Priority	0
Profile	Global Volume Profile (Volume Profile)

This Volume Profile contains no overrides.

Selecting the Bloom post-processing effect

Let’s choose Bloom to start with.


Bloom

Bloom

✓	Threshold	0.87
✓	Intensity	1.48
	Scatter	0.7
	Tint	
	Clamp	65472
	High Quality Filtering	
	Skip Iterations	1

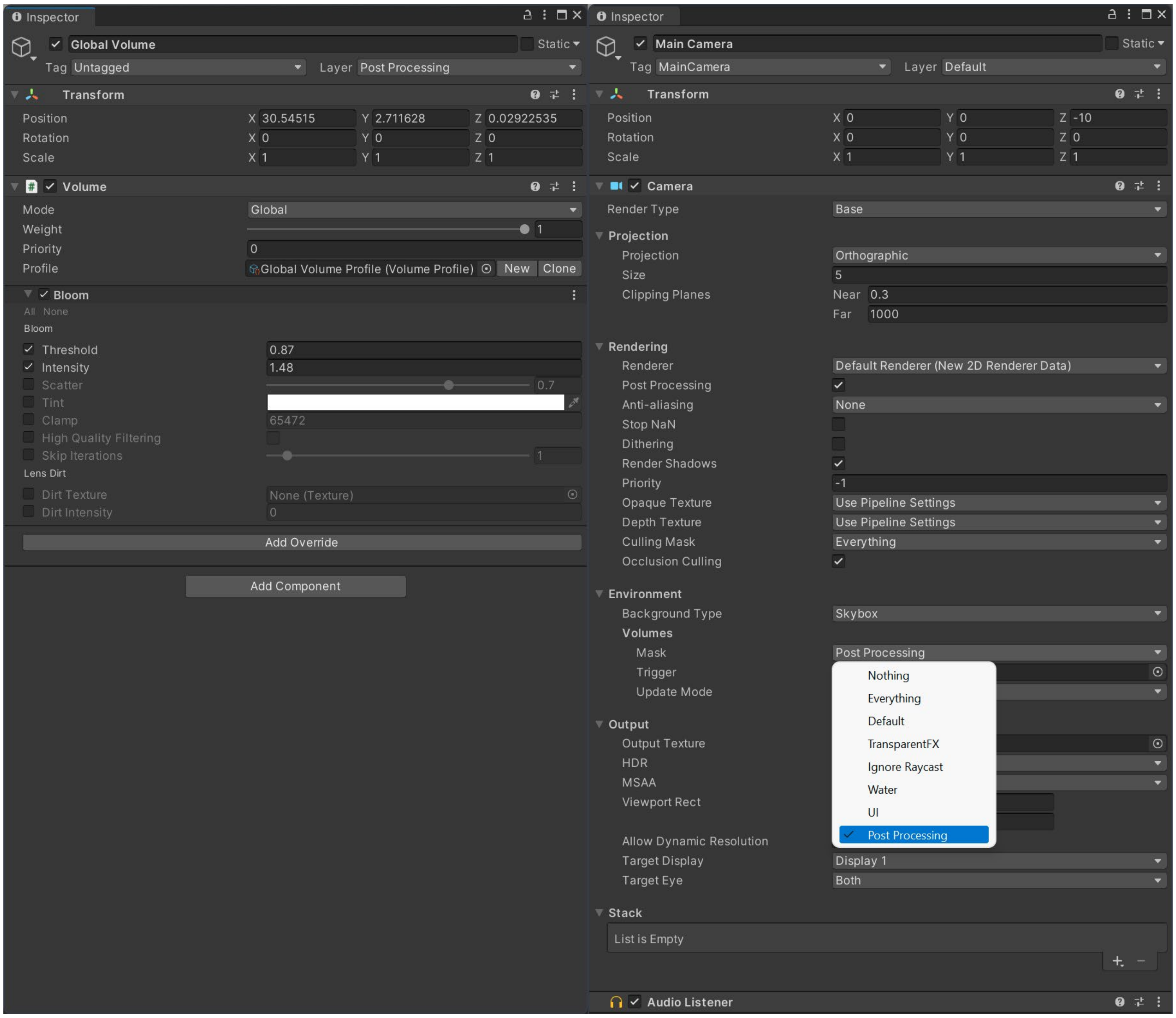
Lens Dirt

	Dirt Texture	None (Texture)
	Dirt Intensity	0

Defining Bloom effect properties

To adjust the effect's property, select the adjacent checkbox.

A quick reminder: Ensure that the Global Volume's GameObject layer matches the layer selected in the Mask option on the main camera.



```
Inspector
✓ Global Volume                                   Static
Tag Untagged            Layer Post Processing
Transform
Position    X 30.54515   Y 2.711628   Z 0.02922535
Rotation    X 0          Y 0          Z 0
Scale       X 1          Y 1          Z 1
✓ Volume
Mode        Global
Weight      1
Priority    0
Profile     Global Volume Profile (Volume Profile)   New   Clone
✓ Bloom
All  None
Bloom
✓ Threshold               0.87
✓ Intensity               1.48
  Scatter                 0.7
  Tint
  Clamp                   65472
  High Quality Filtering
  Skip Iterations         1
Lens Dirt
  Dirt Texture            None (Texture)
  Dirt Intensity          0
Add Override
Add Component
```

```
Inspector
✓ Main Camera                                     Static
Tag MainCamera          Layer Default
Transform
Position    X 0   Y 0   Z -10
Rotation    X 0   Y 0   Z 0
Scale       X 1   Y 1   Z 1
✓ Camera
Render Type                 Base
Projection
  Projection                Orthographic
  Size                      5
  Clipping Planes           Near 0.3
                            Far  1000
Rendering
  Renderer                  Default Renderer (New 2D Renderer Data)
  Post Processing           ✓
  Anti-aliasing             None
  Stop NaN
  Dithering
  Render Shadows            ✓
  Priority                  -1
  Opaque Texture            Use Pipeline Settings
  Depth Texture             Use Pipeline Settings
  Culling Mask              Everything
  Occlusion Culling         ✓
Environment
  Background Type           Skybox
  Volumes
    Mask                    Post Processing
    Trigger                   Nothing
    Update Mode               Everything
                              Default
Output                        TransparentFX
  Output Texture              Ignore Raycast
  HDR                         Water
  MSAA                        UI
  Viewport Rect             ✓ Post Processing
  Allow Dynamic Resolution
  Target Display            Display 1
  Target Eye                Both
Stack
  List is Empty
✓ Audio Listener
```

Make sure the Post Processing check box in the camera is enabled and that the Layer of the Volume's GameObject is selected in the Volumes Mask drop list of the camera

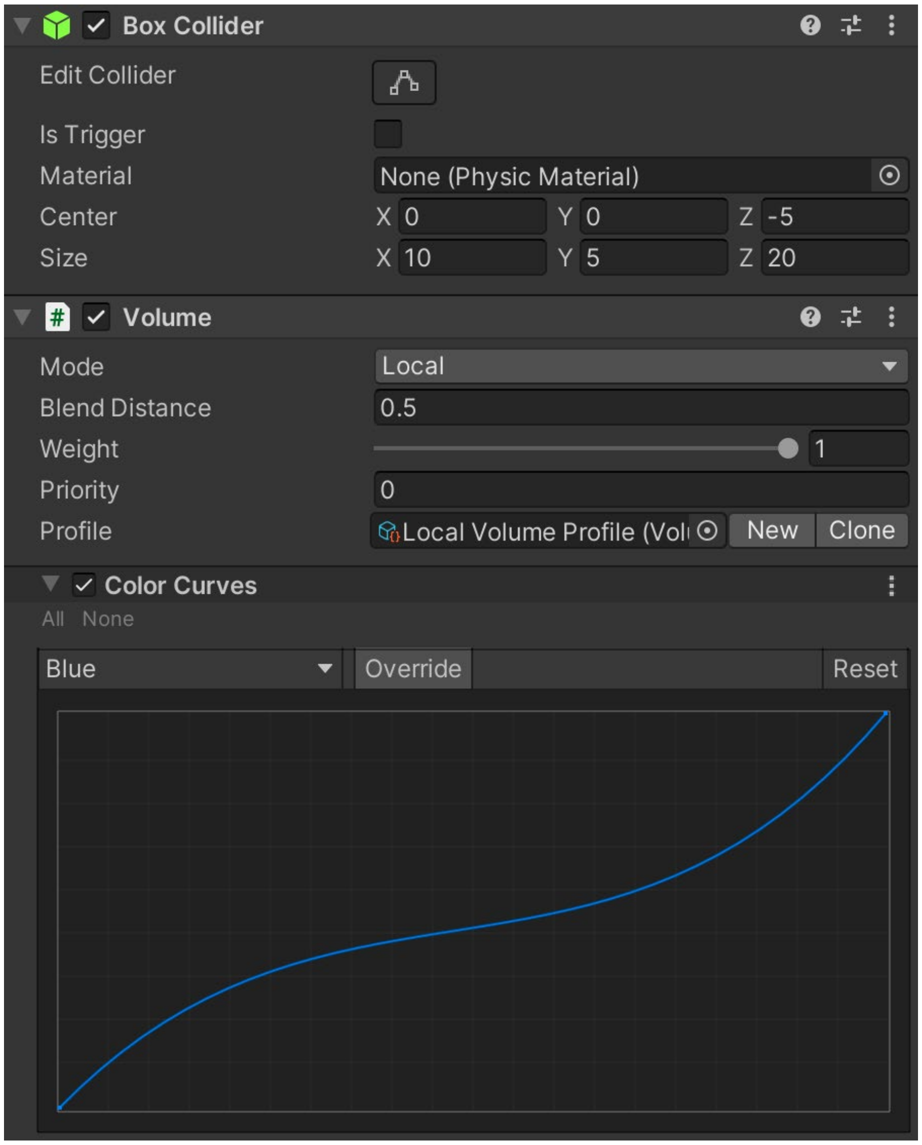
If everything is working correctly, the Bloom effect should be visible in the Game view.

Global Volume was used, so the effects should always be visible when the Volume object is active in the Scene.

Local Volumes

You can add post-processing effects to an entire level, but it's also possible to configure effects for parts of a level, such as when the player enters a building.

To do that, another Volume Object must be created with a different post-processing Profile attached.



Box Collider
Edit Collider
Is Trigger
Material: None (Physic Material)
Center: X 0 Y 0 Z -5
Size: X 10 Y 5 Z 20

Volume
Mode: Local
Blend Distance: 0.5
Weight: 1
Priority: 0
Profile: Local Volume Profile (Vol New Clone

Color Curves
All None
Blue Override Reset

Adding a 3D Box Collider to a Volume Object

Volume's Mode needs to be set as Local, then add the 3D Box Collider to the Volume Object. Post-processing will be applied when the camera enters the Box Collider's bounds, so make sure its along the Z axis is large enough to contain the camera object. By default, the collider will be too small, so double check the size by in Scene view.

Increasing the Blend Distance value will prevent the effects turning on abruptly when the camera enters the Volume zone.

Unity®

unity.com