UNITY FOR GAMES

E-BOOK

INTRODUCTION TO THE

UNIVERSAL RENDER PIPELINE FOR ADVANCED UNITY CREATORS

2021 LTS EDITION

Unity

Contents

INTRODUCTION

Circuit Superstars by Original Fire Games, made with URP and published by Square Enix Collective, for console and PC

This guide is intended to help experienced Unity developers and technical artists migrate their projects from the Built-in Render Pipeline to the Universal Render Pipeline (URP). Topics covered include how to:

- Set up URP for a new project, or convert an existing Built-in Render Pipeline-based project to URP
- Update Built-in Render Pipeline-based lights, shaders, and special effects for URP
- Understand callback differences between the two rendering pipelines, performance optimization in URP, and more

The limitations of the Built-in Render Pipeline have become apparent as the number of platforms supported by Unity continues to grow. Each additional platform and API adds complexity to modifying and maintaining Built-in Render Pipeline architecture.

In 2018, Unity released two new Scriptable Render Pipelines (SRPs): the High Definition Render Pipeline (HDRP) and URP. These SRPs enable you to customize the culling of objects, their drawing, and the post-processing of the frame without having to use low-level programming languages like C++. You can also create your own fully customized SRP.

The aim of SRP architecture is to provide deep flexibility and customization, enhanced performance across the gamut of supported and future platforms, and quick iteration to unleash your creativity.

Multiplatform deployment is a key factor in the success of many games. Players often play the same game on different devices, such as console and mobile, meaning Unity developers require rendering options that scale up and down for numerous devices, with as few steps and little complexity as possible.

After several years of dedicated development, URP technology is now solid and production-ready. This guide will help you leverage its benefits for the successful development of your game.

Author and contributors

Nik Lever, the author of this e-book, has been creating real-time 3D content since the mid-nineties and using Unity since 2006. For over 30 years he's led the small development company, Catalyst Pictures, and has provided courses since 2018 with the aim of helping game developers expand their knowledge in a rapidly evolving industry.

Unity contributors

Felipe Lira is a senior manager of graphics and the URP. With over 13 years of experience as a software engineer in the games industry, he specializes in graphics programming and multiplatform game development.

Ali Mohebali is a senior manager on the graphics product management team. Ali has 18 years of experience working in the games industry, and has contributed to hit titles such as *Fruit Ninja* and *Jetpack Joyride* both by Halfbrick Studios.

Steven Cannavan is a senior development consultant on the Accelerate Solutions Games team, specializing in the Scriptable Rendering Pipelines. He has over 15 years of experience in the game development industry.

Important contributions were also made by:

- David Sena, principal graphics software engineer at NaturalMotion, creators of the *CSR Racing* franchise
- Unity URP engineering and *Gigaya* development teams

EVOLUTION OF RENDERING: FROM BUILT-IN RENDER PIPELINE TO SRP

One of Unity's biggest strengths is its platform reach. The ideal for all game studios is to create once and efficiently deploy their game to their desired range of platforms, from high-end PCs to low-end mobile.

The Built-in Render Pipeline was developed to be a turnkey solution for all platforms supported by Unity. It supports a mix of graphics features and is convenient to use with Forward and Deferred pipelines.

However, as Unity continues to add support for more platforms, we have perceived the following shortcomings surrounding the Built-in Render Pipeline:

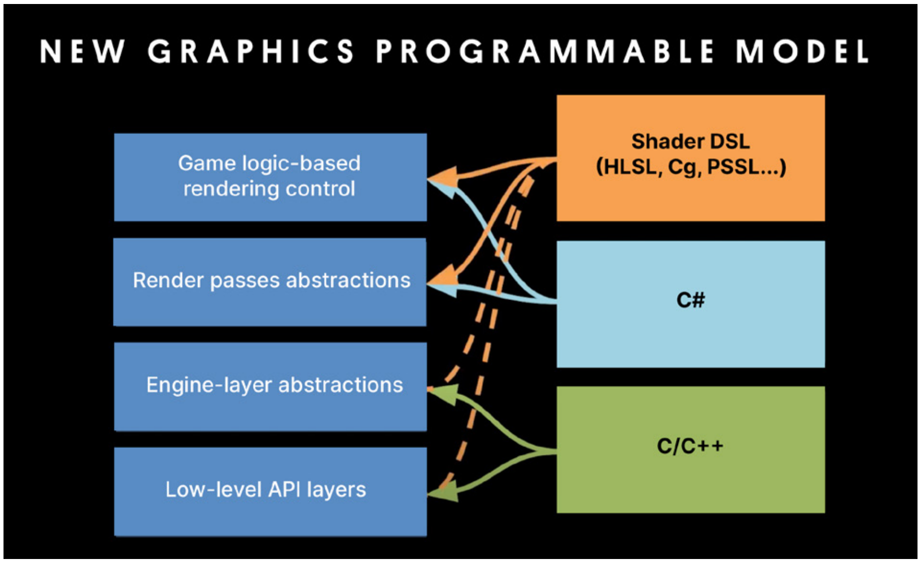
- The bulk of the code is written in C++ and can't be modified by developers, making it a blackbox system
- The render flow and render passes are prestructured
- The rendering algorithm is hardcoded
- Unconstrained customization makes achieving good performance on all platforms difficult
- It exposes callbacks in the rendering code that trigger sync points in the pipeline. Those callbacks prevent multithreaded rendering optimizations, enabling changes for injection of state at any point in the frame dynamically by calling to C#
- Caching data to manage the persistence state for user injection is difficult

The solution: Scriptable Render Pipelines

The SRPs were developed to support an efficient multiplatform workflow by providing:

- Intelligent and reliable scaling for the maximum number of hardware platforms, from high- to low-end devices
- The ability to customize rendering processes using C#, not C++. Using C# means a new executable does not need to be compiled for every change
- Flexibility to support architecture evolution
- Flexibility to create sharp graphics that are performant across many platforms

The image below illustrates how SRPs work. Use C# to control and customize render passes and rendering control, as well as HLSL Shaders that can be created using artist-friendly tools such as Shader Graph. Shaders give you access to even the lower-level API and engine-layer abstractions.



The new graphics programmable model for the Scriptable Render Pipelines

An advanced user can create a new SRP from scratch or modify the HDRP or URP. The graphics stack is open source and available for use on GitHub.

Why choose URP

— **Accessible to a wide range of users:** URP is configurable by artists and technical artists alike, providing more flexibility for prototyping and refining rendering techniques for full game production.

— **Extendable and customizable:** URP allows users to modify existing capabilities and extend the pipeline with new ones, making it a solid choice for advanced users, including Asset Store and third-party package creators, experienced studios, and advanced teams.

While the low-level rendering API is written in C++ for performance purposes, a URP developer can write a simple C# script to be called during the render pipeline, enabling high-level customization without sacrificing performance.

— **Multiple rendering options:** URP provides a Universal Renderer that supports both Forward and Deferred Renderer Paths, as well as a 2D Renderer.

  These renderers can be extended with additional features and Scriptable Render Passes. The Render Objects feature can be used to render objects from a given Layer Mask at different events in the rendering pipeline. It also allows you to override material and other render states when rendering those objects, making it possible to customize the rendering without code. URP can be extended with custom renderers to suit specific needs.

— **Better performance:** URP provides equal, if not better performance than the Built-in Render Pipeline for comparable Quality settings in the majority of cases. In particular:

  — URP evaluates real-time lighting more efficiently. In Forward rendering it evaluates all lighting in a single pass. In Deferred rendering it supports the Native RenderPass API, allowing G-buffer and lighting passes to be combined into a single render pass.

  — There are CPU and GPU improvements when drawing meshes. This is due to SRP Batcher, which ensures fewer draw calls and improvements on how depth is handled.

  — URP makes more efficient use of tile memory on mobile devices, leading to less power consumption, a longer battery life, and therefore, the possibility of longer play sessions.

  — URP comes with an integrated post-processing stack that allows for better performance compared to the Built-in Render Pipeline. Using the Volume framework, you can create post-processing effects that are dependent on the Camera position without writing any code.

— **Compatible with the latest tools:** URP supports the latest artist-friendly tools, such as Shader Graph, VFX Graph, and the Rendering Debugger.

There is no set date yet for URP to replace the Built-in Render Pipeline as the default rendering pipeline in Unity. The Built-in Render Pipeline will remain an available option at least for the next release cycle in 2022.

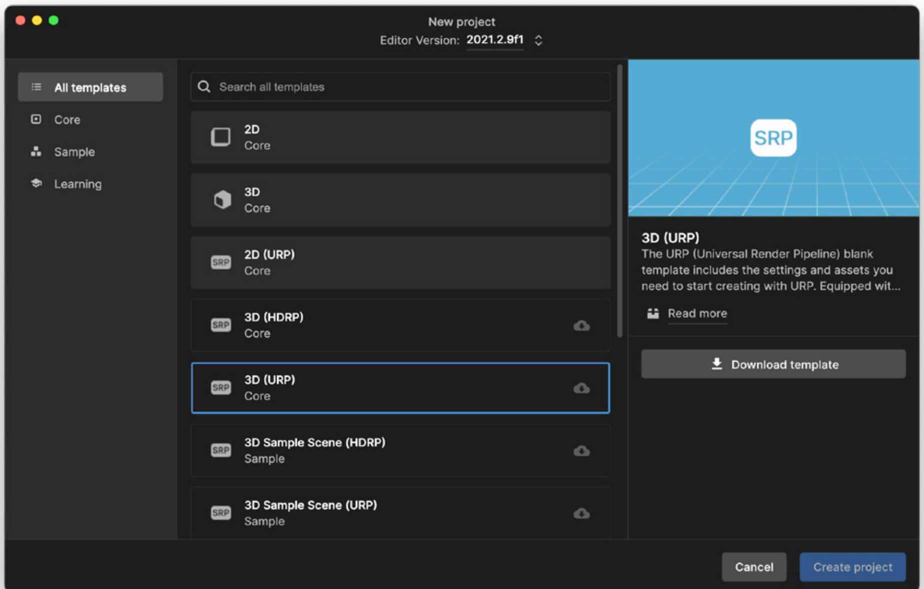
Follow this link for a comprehensive comparison of the Built-in Render Pipeline and URP capabilities.

THE CONVERSION PROCESS

This section covers the steps for starting a new project with URP or converting an existing project to URP.

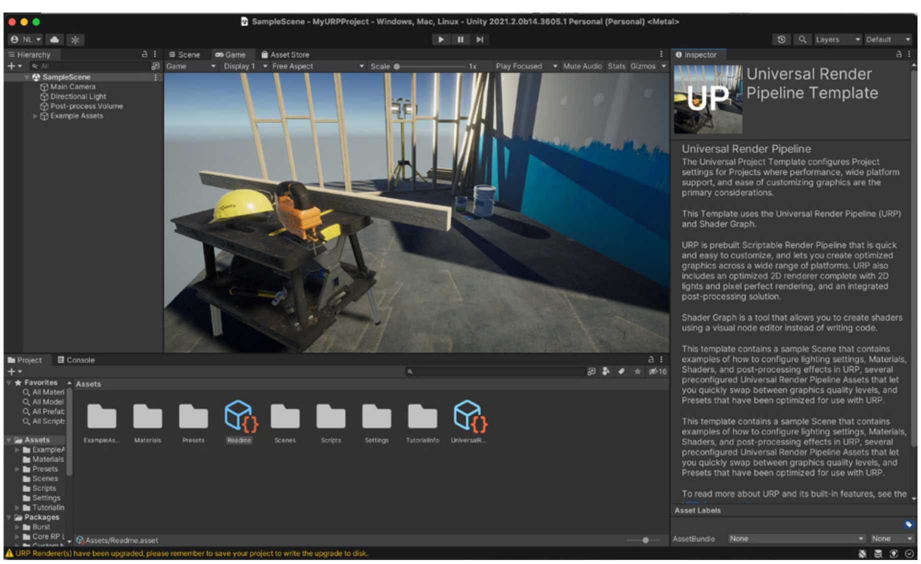
How to open a new URP project

Open a new project using URP via the Unity Hub. Click on **New** and verify that the Unity version selected at the top of the window is 2021.2 or newer. Choose a name and location for the project, select the **3D (URP)** template or **3D Sample Scene (URP)**, and click **Create**.



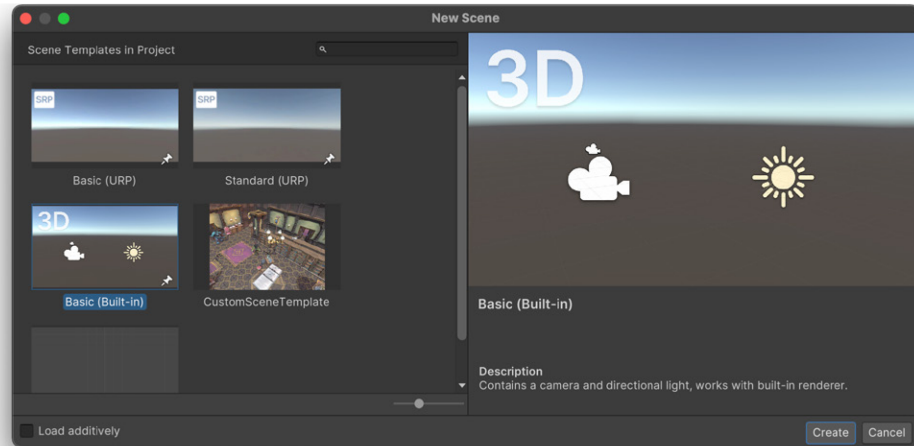
Creating a new project with the URP template, which might require you to download the template for the first time

Note: The template ensures that your project is set to use a linear color space, which is required for calculating lighting correctly.



This Sample Scene is included in the 3D Sample Scene (URP) project template.

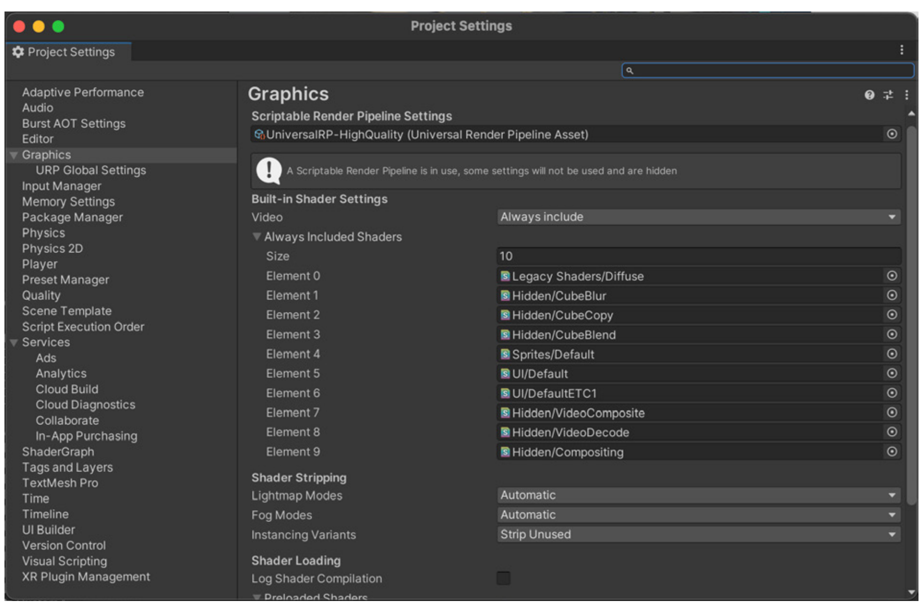
You can create new scenes via **File > New Scene**, with essential GameObjects such as Camera and Directional light, and even create your own scene template with prepopulated objects. Read more in the URP Scene Templates documentation.



The New Scene dialog displaying Scene Templates

Go to **Edit > Project Settings** and open the **Graphics** panel. To use URP in-Editor, you must select a URP Asset from the **Scriptable Render Pipeline Settings**. The URP Asset controls the global rendering and Quality settings of a project and creates the rendering pipeline instance. Meanwhile, the rendering pipeline instance contains intermediate resources and the render pipeline implementation.

UniversalRP-HighFidelity is the default URP Asset selected, but you can switch to **UniversalRP-Balanced** or **UniversalRP-Performant**.



The Graphics panel in Project Settings

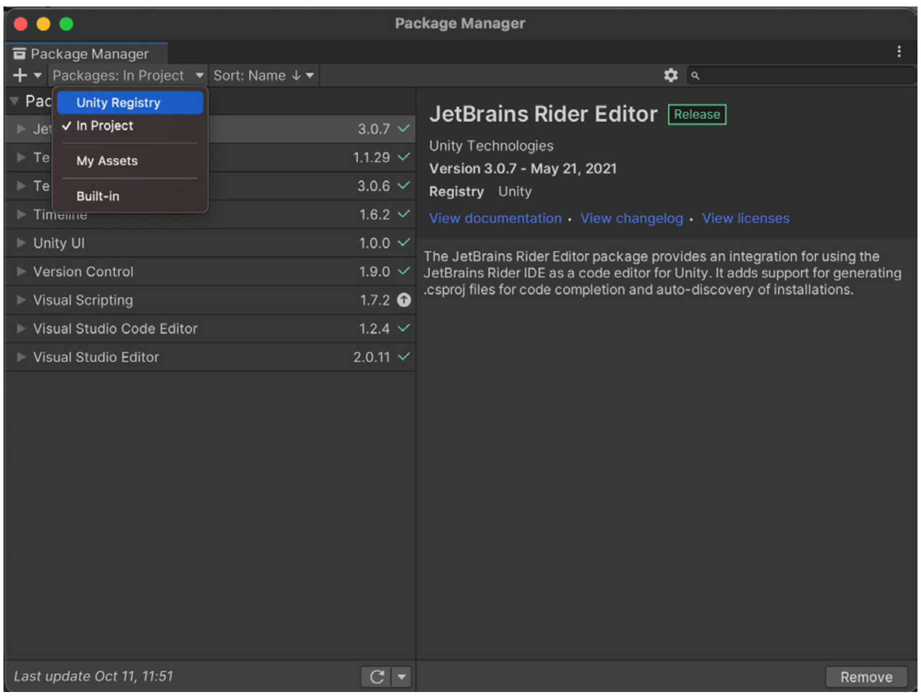
A later section of this guide details how to adjust the settings of a URP Asset.

To open a clean project, use the 3D (URP) template or, if you want to remove the sample scene from the Sample Scene (URP) template, delete the **ExampleAssets** and **Tutorial Info folders**, as well as the **Settings > SampleSceneProfile.asset, Scenes > SampleScene.unity**, and **Scenes > SampleSceneLightingSettings.lighting**.

How to add URP to an existing Built-in Render Pipeline project

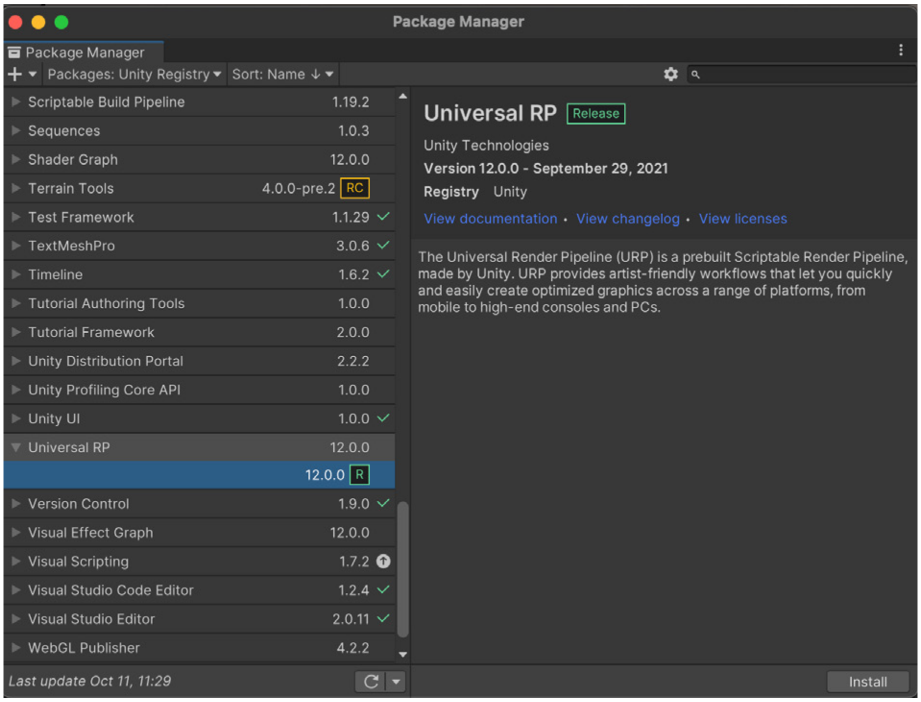
Important: Be sure to backup your project using source control before following the steps in this section. This process will convert assets, and Unity does not provide an undo option. If you use source control, you will be able to revert to previous versions of the assets if necessary.

If you upgrade an existing Built-in Render Pipeline project, you'll need to add the **URP package** to your project as it's not included in Unity 2021.2 or 2021 LTS.



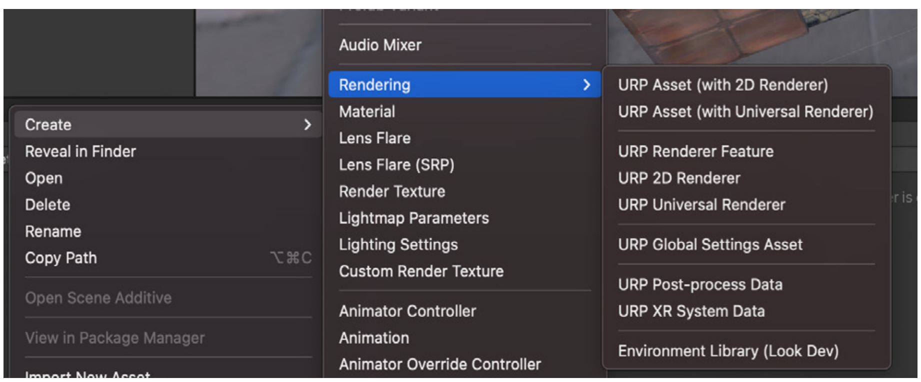
The Package Manager displaying the Unity Registry packages

Go to **Window > Package Manager** and click the **Packages** drop-down to add URP to your project. Make sure to select the **Unity Registry**, followed by **Universal RP**. Click **Download** in the lower-right corner of the window if the URP package is not yet installed on your development computer. Then click **Install** once it's downloaded.



Installing URP via the Package Manager

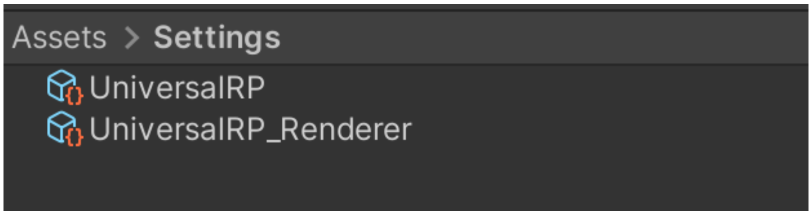
To create a URP Asset, right-click in the **Project** window and choose **Create > Rendering > URP Asset (with Universal Renderer)**. Name the asset.



Creating a URP Asset

Remember, if you create a new project using the Universal Render Pipeline or 3D (URP) templates, these URP Assets and the URP package are already available in the project.

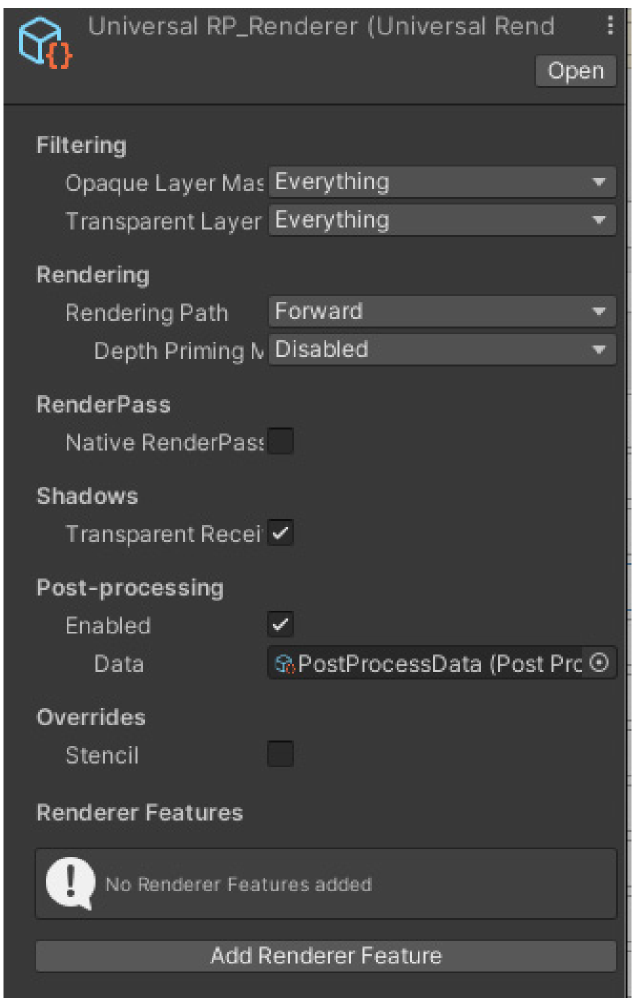
Rather than creating a single URP Asset, URP uses two files, each with an Asset extension.



Two Assets in URP, one for URP settings and the other for Renderer Data

One of the assets will have a file name ending with _Renderer; in the following example it's called **Universal RP_Renderer**. This is 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. This way, you can facilitate the creation of high-quality effects. See the section on Pipeline callbacks for more information.

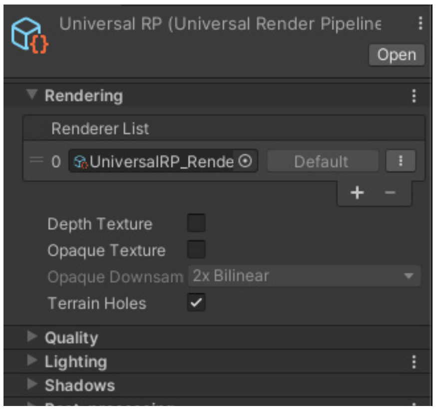
Additionally, the UniversalRP_Renderer controls high-level rendering logic and passes for URP. It supports Forward and Deferred paths, and a 2D Renderer that enables features such as 2D lights, 2D shadows, and Light Blend Styles. You can even extend URP to create your own renderers.



The Inspector for UniversalRP_Renderer Data Asset

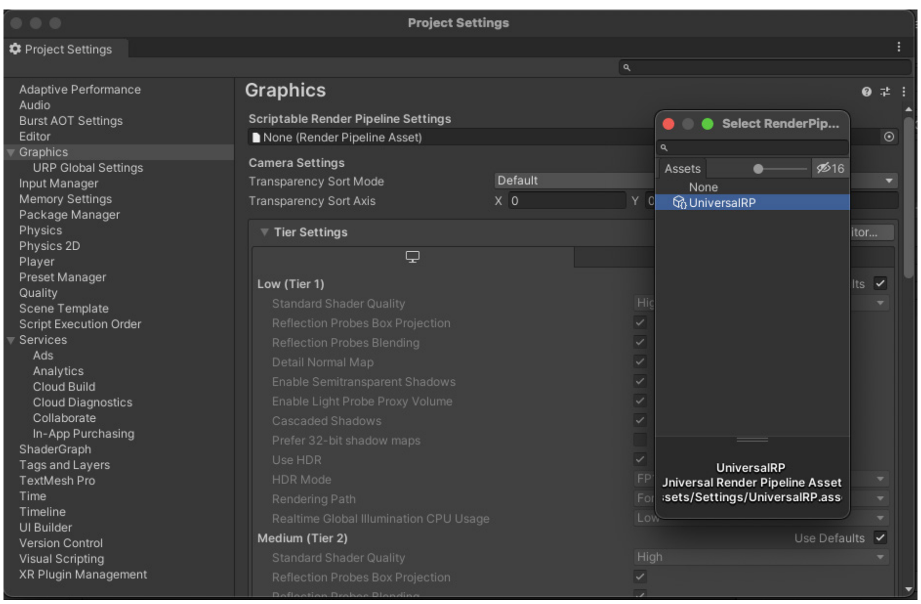
The other URP Asset serves to control settings for Quality, Lighting, Shadows, and Post-processing. You can use different URP Assets to control the Quality settings, a process outlined further down in this section. This Settings Asset is linked to the Renderer Data Asset via the Renderer List. When you create a new URP Asset, the Settings Asset 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.

The default renderer is used for all Cameras, including the Scene view. A Camera can override the default renderer by selecting another one from the Renderer List. This can be done through the use of a script, as needed.



A URP Asset in the Inspector

Despite following these steps to create a URP Asset, an open scene in the Scene or Game view will still use the Built-in Render Pipeline. You must complete one last step to make the switch to URP: Go to **Edit > Project Settings** and open the **Graphics** panel. Click the small dot next to **None (Render Pipeline Asset)**. In the open panel, select **UniversalRP**.



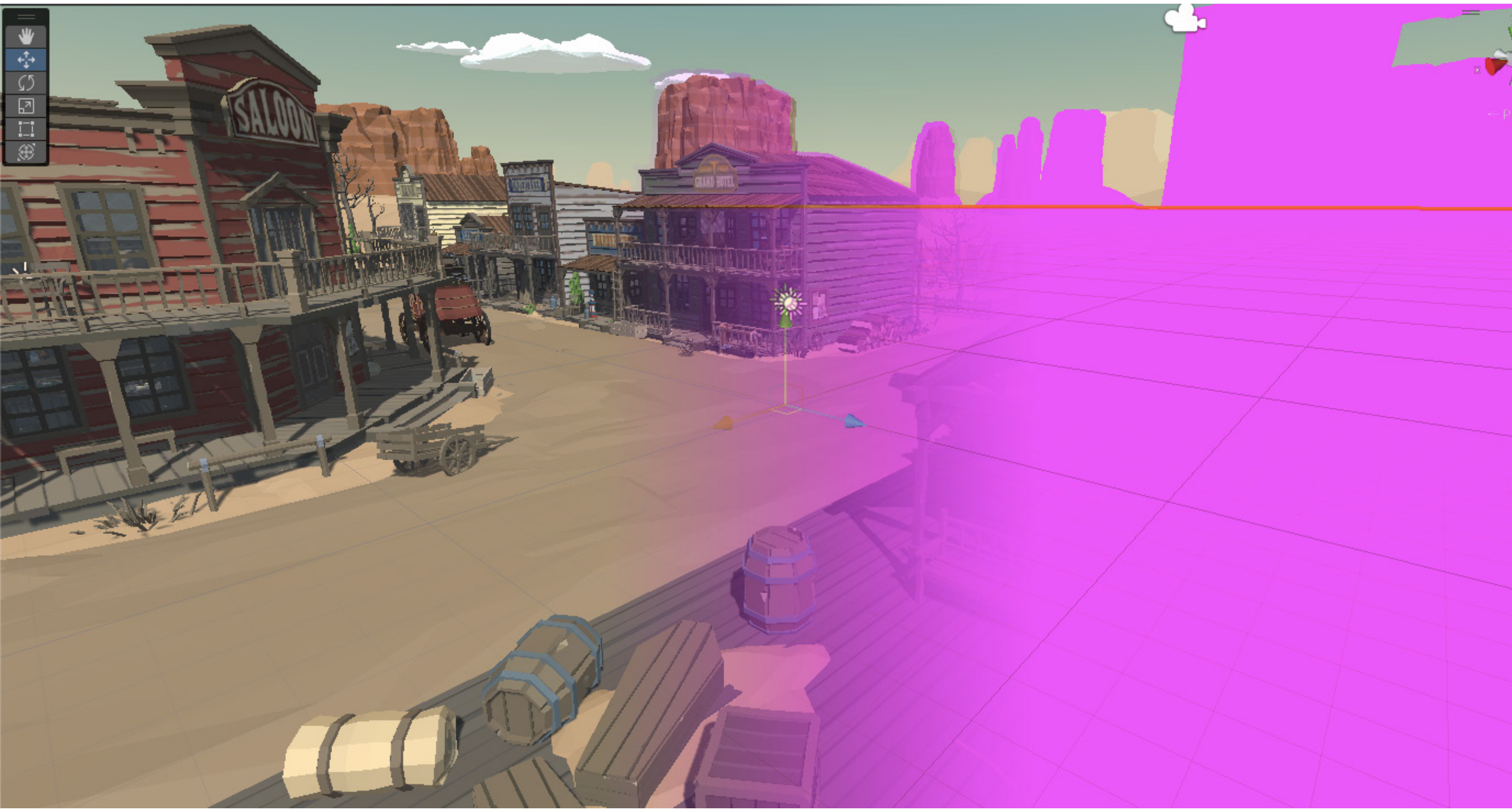
Selecting a Scriptable Render Pipeline Asset

A warning message will pop up regarding the switch, but just press **Continue**.

As there is no content in your project yet, changing the render pipeline will be almost instantaneous. You're now ready to use URP.

Converting the scenes of an existing project

After you complete the above steps, you'll find that your beautiful scenes are suddenly colored magenta. This is because the shaders used by the materials in a Built-in Render Pipeline project are not supported in URP. Fortunately, there is a method for restoring your scenes to their original quality.



Materials in a scene appear in magenta because their Built-in Render Pipeline-based shaders must be converted for use in URP.

Go to **Window > Rendering > Render Pipeline Converter**. Choose **Convert Built-In to 2D (URP)** for a 2D project, or **Built-In to URP** for a 3D project. Assuming that your project is 3D, you'll need to select the appropriate converters:

— **Rendering Settings:** Select this to create multiple Render Pipeline setting assets that will match Built-in Render Pipeline Quality settings as closely as possible. This lets you test different Quality Levels more efficiently. See the section on comparing Built-in Render Pipeline and URP Quality options for more details.

— **Material Upgrade:** Use this to convert materials from the Built-in Render Pipeline to URP.

— **Animation Clip Converter:** This converts animation clips. It runs once the Material Upgrade converter finishes.

— **Read-only Material Converter:** This converts the prebuilt, read-only Materials included in a Unity project. It indexes the project and creates the temporary .index file. Note that it can take significant time.

Converting custom shaders

Custom shaders are not converted using the Material Upgrade converter. The Shaders and New tools sections outline the steps for converting custom Built-in Render Pipeline shaders to URP. Using Shader Graph is often the quickest way to update a custom shader to URP.

There are six different URP shaders:

— **Universal Render Pipeline/Lit:** This physically based render (PBR) shader, similar to the built-in Standard Shader, can be used to represent most real-life materials. It supports all the Standard Shader features with both metallic and specular workflows.

— **Universal Render Pipeline/Simple Lit:** This uses a Blinn-Phong model, and is suitable for low-end mobile devices or games that don't use PBR workflows.

— **Universal Render Pipeline/Unlit:** This is a GPU performant shader that doesn't use lighting equations.

— **Universal Render Pipeline/Terrain/Lit:** This is suitable to use with the Terrain Tools package.

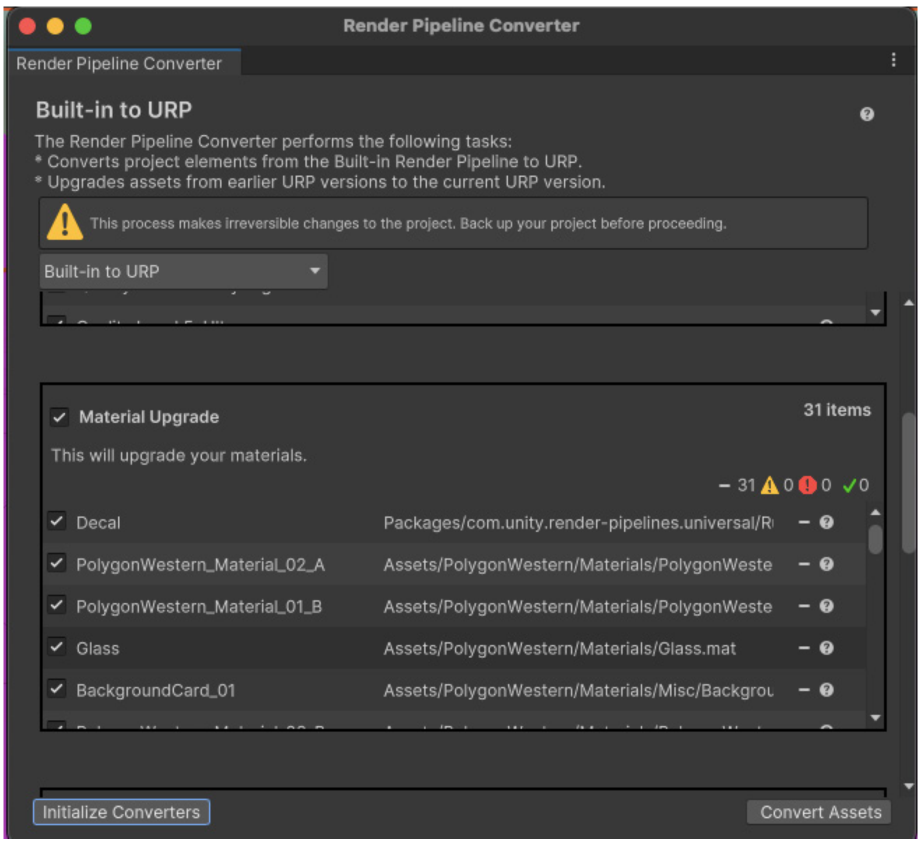
— **Universal Render Pipeline/Particles/Lit:** This particle shader uses a PBR lighting model.

— **Universal Render Pipeline/Particles/Unlit:** This unlit particle shader is light on the GPU.

Although Simple Lit replaces many legacy/mobile shaders, the performance is not the same. Legacy/mobile shaders only partially evaluate lighting, whereas Simple Lit considers all lights as defined by the URP Asset.

Refer to this table in our URP documentation to see how each URP shader maps to its Built-in Render Pipeline equivalent.

Once you select one or more of the above converters, click **Initialize Converters**. The project will be scanned and those assets that need converting will be added to each of the converter panels. You can limit the conversions by deselecting the items using the checkbox provided for each one. At this stage, click **Convert Assets** to start the conversion process. Once it's complete you might be asked to reopen the scene that is active in the Editor.



The Render Pipeline Converter

Comparing Quality options between the Built-in Render Pipeline and URP

There are several default Quality options available in the Built-in Render Pipeline, from Very low to Ultra. The Quality settings impact the fidelity of the scene, including Texture resolution, lighting, shadow rendering, and so on.

Go to **Edit > Project Settings** and select the **Quality** panel. Here, you can switch between these Quality options by picking the current quality. This will change the render settings used by the Scene and Game views. You can also edit each of the Quality options from this panel.

If you select the **Rendering Settings** option while using the Render Pipeline Converter and switching from the Built-in Render Pipeline to URP, a set of URP Assets that attempt to match the Built-in Render Pipeline Quality options will be created. The first table below shows how the Built-in Render Pipeline maps to URP for Low settings, while the second table displays a comparison for High settings. In both the Built-in Render Pipeline and URP, settings are chosen via the Quality panel. The URP Asset settings are available via the Inspector when selecting a URP Asset. Refer to the URP documentation for more details.

Built-in Render Pipeline to URP: Low settings

Setting	Built-in Render Pipeline	URP	URP Asset settings
Rendering			
Pixel Light Count	0	Not applicable (NA) *	NA
Anti-aliasing	Disabled	NA	Disabled
Render Scale	NA	NA	1
Real-time Reflection Probes	No	No	
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Don’t sync	Don’t sync	
Depth Texture	NA	NA	No
Opaque Texture	NA	NA	No
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	Yes
HDR	NA	NA	Yes
Textures			
Texture Quality	Half res	Half res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	No	No	NA
Particles			
Soft Particles	No	NA	NA
Particle Raycast Budget	16	16	NA
Terrain			
Billboards Face Camera Position	No	No	NA

Setting	Built-in Render Pipeline	URP	URP Asset settings
Shadows			
Shadowmask Mode	Shadowmask	Shadowmask	NA
Shadows	Disabled	NA	NA
Shadow Resolution	Low resolution	NA	NA
Shadow Projection	Stable fit	NA	NA
Shadow Distance	20	NA	NA
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	No Cascades	NA	NA
Cascade splits	NA	NA	NA
Working unit	NA	NA	NA
Depth Bias	NA	NA	NA
Normal Bias	NA	NA	NA
Soft Shadows	NA	NA	NA
Async Asset Upload			
Time Slice	2	2	NA
Buffer Size	16	16	NA
Persistent Buffer	Yes	Yes	NA
Level of Detail			
LOD Bias	0.4	0.4	NA
Maximum LOD level	0	0	NA
Meshes			
Skin Weights	4 bones	4 bones	NA
Lighting			
Main Light:	NA	NA	Per pixel
• Cast Shadows	NA	NA	No
• Shadow Resolution	NA	NA	NA
Additional Lights:	NA	NA	Disabled
• Per Object Limit	NA	NA	NA
• Cast Shadows	NA	NA	NA
• Shadow Atlas Resolution	NA	NA	NA
• Shadow Resolution tiers	NA	NA	NA
• Cookie Atlas Resolution	NA	NA	NA
• Cookie Atlas Format	NA	NA	NA
Reflection Probes:	NA	NA	NA
• Probe Blending	NA	NA	No
• Box Projection	NA	NA	No
Post-processing			
Grading Mode	NA	NA	Low Dynamic Range
LUT size	NA	NA	16
Fast sRGB/Linear conversion	NA	NA	No

* In URP, Pixel Light Count is handled using **Additional Lights > (Per pixel) > Per Object Limit**.

Built-in Render Pipeline to URP: High settings

Setting	Built-in Render Pipeline	URP	URP Asset settings
Rendering			
Pixel Light Count	2	Not applicable (NA)	NA
Anti-aliasing	Disabled	NA	2x
Render Scale	NA	NA	1
Real-time Reflection Probes	Yes	Yes	NA
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Every V Blank	Every V Blank	NA
Depth Texture	NA	NA	No
Opaque Texture	NA	NA	No
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	Yes
HDR	NA	NA	Yes
Textures			
Texture Quality	Full res	Full res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	No	No	NA
Particles			
Soft Particles	No	NA	NA
Particle Raycast Budget	256	256	NA
Terrain			
Billboards Face Camera Position	Yes	Yes	NA
Shadows			
Shadowmask Mode	Distance Shadowmask	Distance Shadowmask	NA
Shadows	Hard and Soft Shadows	NA	NA
Shadow Resolution	Medium resolution	NA	2048
Shadow Projection	Stable fit	NA	NA
Shadow Distance	40	NA	50
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	2 Cascades	NA	2
Cascade splits	33/67	NA	12.5/33.8/3.8
Working unit	Percent	Percent	Metric

Setting	Built-in Render Pipeline	URP	URP Asset settings
Depth Bias	NA	NA	1
Normal Bias	NA	NA	1
Soft Shadows	NA	NA	Yes
Async Asset Upload			
Time Slice	2	2	NA
Buffer Size	16	16	NA
Persistent Buffer	Yes	Yes	NA
Level of Detail			
LOD Bias	1	1	NA
Maximum LOD level	0	0	NA
Meshes			
Skin Weights	Unlimited	Unlimited	NA
Lighting			
Main Light:	NA	NA	Per pixel
• Cast Shadows	NA	NA	Yes
• Shadow Resolution	NA	NA	
Additional Lights:	NA	NA	Per pixel
• Per Object Limit	NA	NA	4
• Cast Shadows	NA	NA	Yes
• Shadow Atlas Resolution	NA	NA	2048
• Shadow Resolution tiers	NA	NA	512/1024/2048
• Cookie Atlas Resolution	NA	NA	2048
• Cookie Atlas Format	NA	NA	Color high
Reflection Probes:	NA	NA	NA
• Probe Blending	NA	NA	Yes
• Box Projection	NA	NA	No
Post-processing			
Grading Mode	NA	NA	Low Dynamic Range
LUT size	NA	NA	32
Fast sRGB/Linear conversion	NA	NA	No

How to work with Quality settings

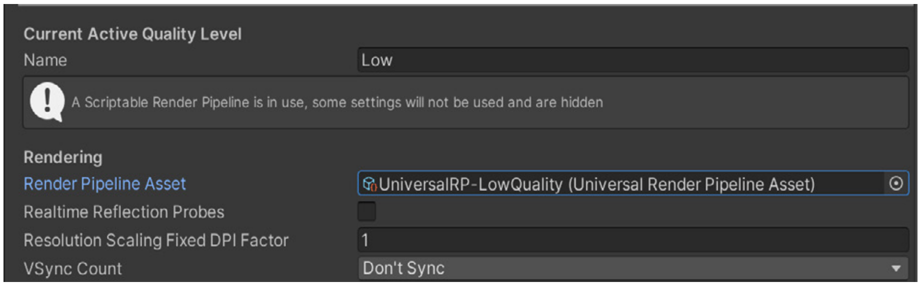
Quality settings were previously handled in the Quality panel of the Project Settings dialog box. When using URP, settings are divided between the Quality panel and those for each URP Asset. The following table shows where each setting can be found.

Quality settings when using URP

Setting	Quality panel	URP Asset
Rendering		
Anti-aliasing		√
Render Scale		√
Resolution Scaling Fixed DPI Factor	√	
VSync Count	√	
Depth Texture		√
Opaque Texture		√
Opaque Downsampling		√
Terrain Holes		√
HDR		√
Textures		
Texture Quality	√	
Anisotropic Textures	√	
Texture Streaming	√	
Particles		
Particle Raycast Budget	√	
Terrain		
Billboards Face Camera Position	√	
Shadows		
Shadowmask Mode	√	
Shadow Resolution		√
Shadow Distance		√
Shadow Cascades		√
Cascade splits		√
Working unit		√
Depth Bias		√

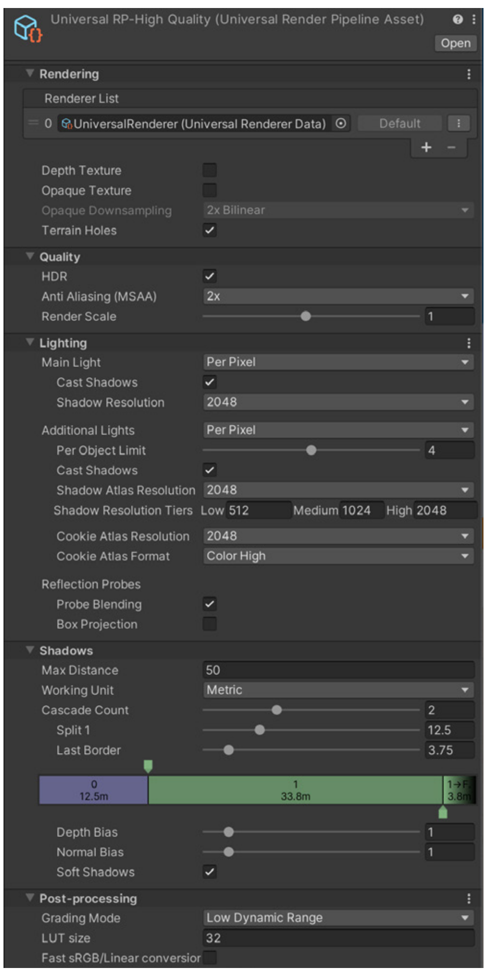
Setting	Quality panel	URP Asset
Normal Bias		√
Soft Shadows		√
Async Asset Upload		
Time Slice	√	
Buffer Size	√	
Persistent Buffer	√	
Level of Detail		
LOD Bias	√	
Maximum LOD level	√	
Meshes		
Skin Weights	√	
Lighting		
Main Light:		√
• Cast Shadows		√
• Shadow Resolution		√
Additional Lights:		√
• Per Object Limit		√
• Cast Shadows		√
• Shadow Atlas Resolution		√
• Shadow Resolution tiers		√
• Cookie Atlas Resolution		√
• Cookie Atlas Format		√
Reflection Probes:	√	
• Probe Blending		√
• Box Projection		√
Post-processing		
Grading Mode		√
LUT size		√
Fast sRGB/Linear conversion		√

If you switch between Quality options, choose a **Quality Level** for the Render Pipeline Asset in the **Quality** panel via **Project Settings**. Note that if the Quality Level is not set, the Render Pipeline Asset will default to the one set as the Scriptable Render Pipeline Asset in the Graphics panel. This can cause some confusion as you attempt to adjust the Quality settings of a URP Asset. For instance, you might accidentally assume that the Quality Level set in the URP Asset is the one currently used by the Scene and Game views.



Setting the Quality Level for the Render Pipeline Asset

Modifying a URP Asset



A URP Asset in the Inspector

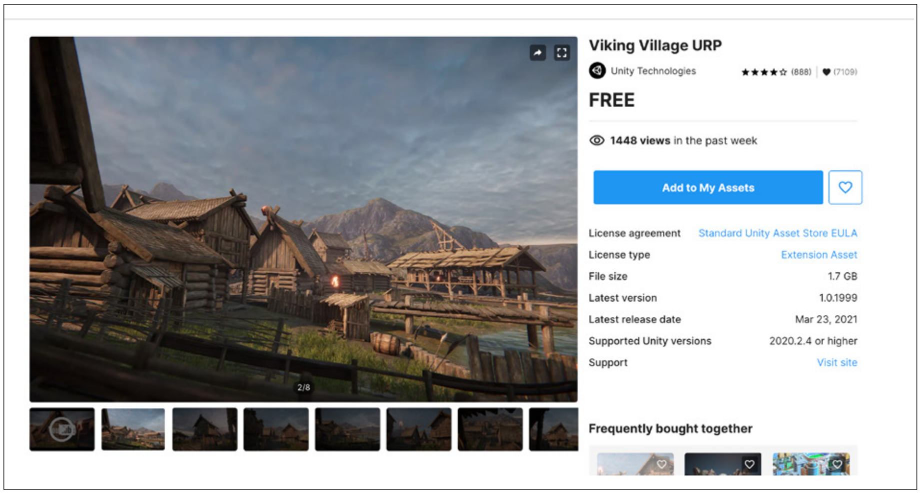
This image shows a URP Asset in the Inspector with all of its available settings. See the URP documentation to learn more about each setting.

> **Note:** If you have the URP 2D Renderer enabled, some of the options related to 3D rendering in the URP Asset will not impact your final app or game. The 2D Renderer Asset is available under **Scriptable Render Pipeline Settings** via **Edit > Project Settings > Graphics**.

Converting an example project from the Built-in Render Pipeline to URP

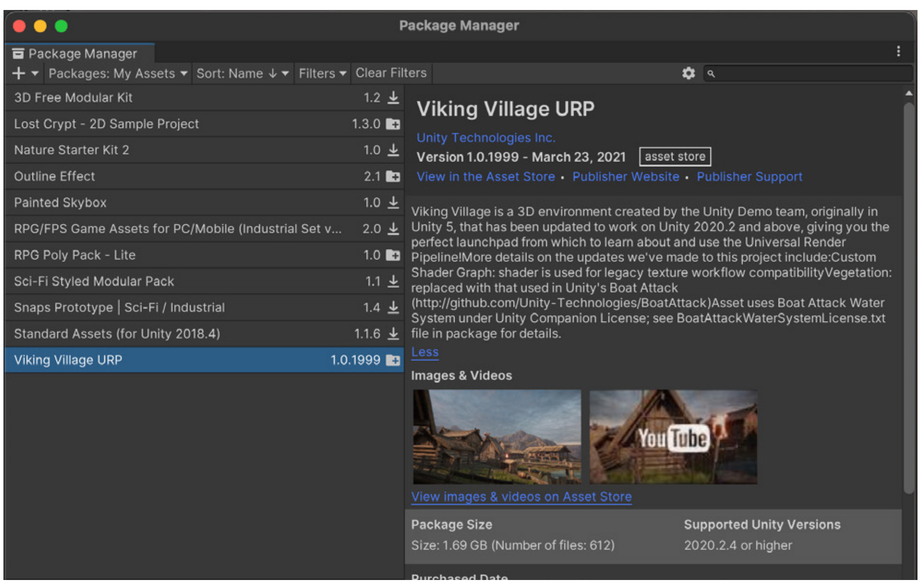
The Unity demo project Viking Village URP shows off URP capabilities with Light Probes, Reflection Probes, water special effects that use a custom ScriptableRenderPass, shaders converted via Shader Graph, and URP post-processing. The project is available for free on the Asset Store.

Open Viking Village URP in the Editor to follow along with the steps in this section. Start by clicking **Add to My Assets** to add this demo to the Packages list available in-Editor.



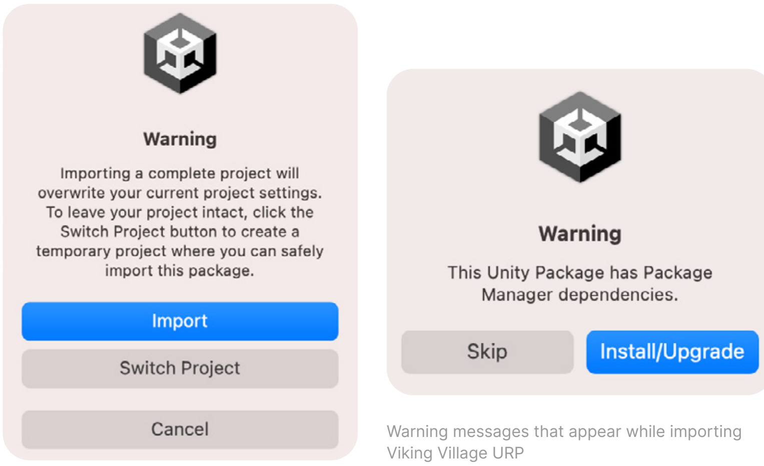
Viking Village URP on the Unity Asset Store

Then create a new 3D project from the Unity Hub (you don't need to use the URP template). Go to **Window > Package Manager**, select **My Assets > Viking Village URP** from the **Packages** drop-down, and click **Import**.



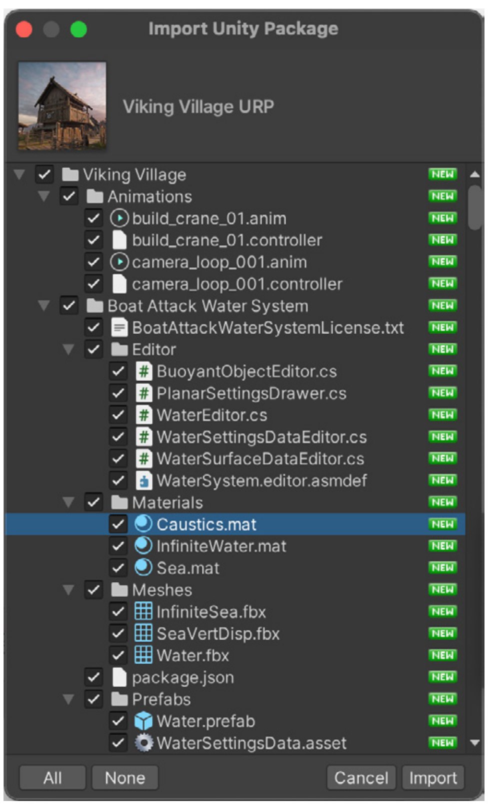
Viking Village URP in Package Manager

A couple of warning messages will appear (see below). The first one warns you that importing a complete project will affect your current Project Settings, but since you've created an empty project it's safe to proceed. The second warning alerts you about installing or upgrading certain packages. Click on the default blue button. This is required to avoid an incorrect lighting setup as the URP default is a linear color space, opposite the Built-in Render Pipeline which defaults to a gamma color space.



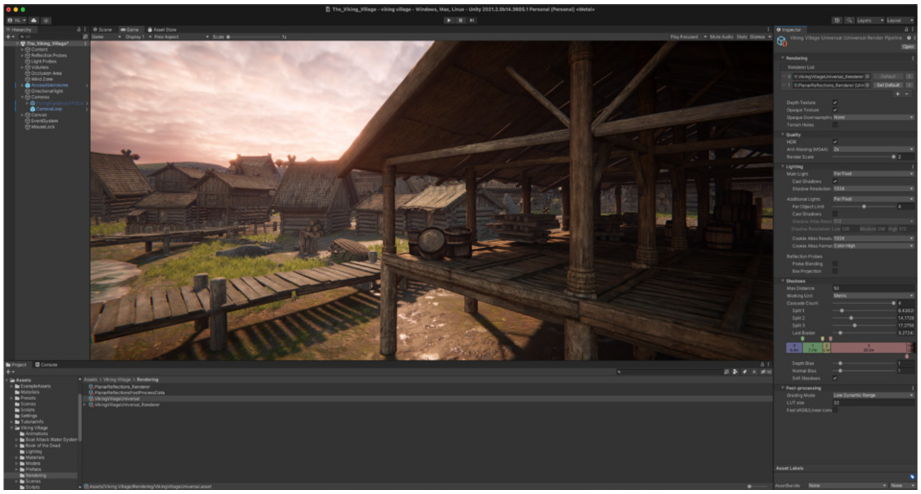
Warning messages that appear while importing Viking Village URP

Once the download is complete, the panel shown below will open. Make sure to leave everything selected and click **Import**.



Importing the demo project

Wait for all the assets to finish importing, then go to the demo located in **Viking Village > Scenes > The_Viking_Village**. Click **Window > Package Manager**, and in the drop-down select **Unity Registry**, followed by **Universal RP**. Update the URP package to 12.x.



Viking Village in Game view

Viking Village in Scene view

The URP Asset set in the Graphics panel, via the Scriptable Render Pipeline Asset, is named **Viking Village > Rendering > VikingVillageUniversal**. It is configured for high-end hardware, and therefore, might play at a low frame rate on older hardware.

Follow these steps to test different Quality Levels:

1. Generate a set of assets via **Window > Rendering > Render Pipeline Converter**.

2. Choose the **Built-in Render Pipeline to URP** option, then select **Rendering Settings**.

3. Click **Initialize Converters**.

4. A number of Settings options will appear in the panel; Click **Convert Assets** to create the URP Assets.

5. The URP Assets will be assigned to the available Quality levels via the **Project Settings > Quality** panel.

6. The highest quality asset will replace VikingVillageUniversal in the Graphics panel. The **Viking Village > Rendering > VikingVillageUniversal_Renderer** makes use of Renderer Features and water effects.

7. To restore these, add the above renderer to the **Renderer List** and set it as the default for each URP Asset used in Quality Levels. Now you can quickly switch Quality Levels in the Quality panel.

LIGHTING IN URP

This section shows how lighting in URP works and describes the differences between the workflows of the two rendering pipelines.

Start with these resources if you are new to lighting in Unity:

— Lighting documentation
— Introduction to lighting and rendering
— The art of lighting game environments
— Real-time lighting in Unity
— Harnessing light with the URP and the GPU Lightmapper

If you convert a project from the Built-in Render Pipeline to URP, you might notice differences in the lighting. This is because the Built-in Render Pipeline uses a gamma lighting model by default and URP uses a linear model. As such, any light with an intensity value differing from 1.0 will need to be adjusted.

There are also differences in where to find the Settings controls in-Editor, as well as how to handle the challenge of widely differing hardware specs. The rest of this section covers some tricks you can use to achieve balance between graphic fidelity and performance.

As before, you'll set properties in the three places listed here. A and B are essentially the same for both render pipelines, while C applies to URP only:

A. **Window > Rendering > Lighting:** This panel allows you to set lightmapping and environment settings, and view real-time and baked lightmaps. It is unchanged from the Built-in Render Pipeline to URP.

B. **Light Inspector:** There are significant differences between the Built-in Render Pipeline and URP Inspectors. See the Light Inspector section for details.

C. **URP Asset Inspector:** This is the principal place where you will set shadows. Lighting in URP relies heavily on the settings chosen in this panel.

Quality settings are handled via **Edit > Project Settings > Quality** in the Built-in Render Pipeline. In URP, this depends on the URP Asset settings which can be swapped using the **Quality** panel (see the Quality settings section).

As the focus here is on lighting, the methods apply to materials that use the shaders in the following table.

URP shaders for lit scenes

Shader	Description
Complex Lit	This shader has all the features of the Lit Shader. Select it when using the Clear Coat option to give a metallic sheen to a car, for example. The specular reflection is calculated twice – once for the base layer, and again to simulate a transparent thin layer on top of the base layer.
Lit	The Lit Shader lets you render real-world surfaces, such as stone, wood, glass, plastic, and metals with photorealistic quality. The light levels and reflections look lifelike and react across various lighting conditions, from bright sunlight to a dark cave. This is the default choice for most materials that use lighting. It supports baked, mixed, and real-time lighting, and works with Forward or Deferred rendering. It is a physically based shading (PBS) model. Due to the complexity of the shading calculations, it's best to avoid using this shader on low-end mobile hardware.
Simple Lit	This shader is not physically based. It uses a non-energy conserving Blinn-Phong shading model and gives a less photorealistic result. Nonetheless, it can provide an excellent visual appearance. It is more suited to use on non-physically based projects when targeting low-end mobile devices.
Baked Lit	This shader provides a performance boost for objects that don't need to support real-time lighting, including distant static objects that will never be affected by dynamic objects, real-time lights, or dynamic shadows.

Built-in Render Pipeline vs URP lighting falloff and attenuation

Another difference between the Built-in Render Pipeline and URP is how they compute light falloff/attenuation that applies to Spot and Point lights.

URP uses the physically based InverseSquared falloff, described here.
The Built-in Render Pipeline uses the Legacy falloff, which is not physically based, as described on the same page. The light radius affects the falloff, which can result in lights with a big radius, and thereby impact culling performance as the light will touch more objects than necessary.

Lit or Simple Lit?

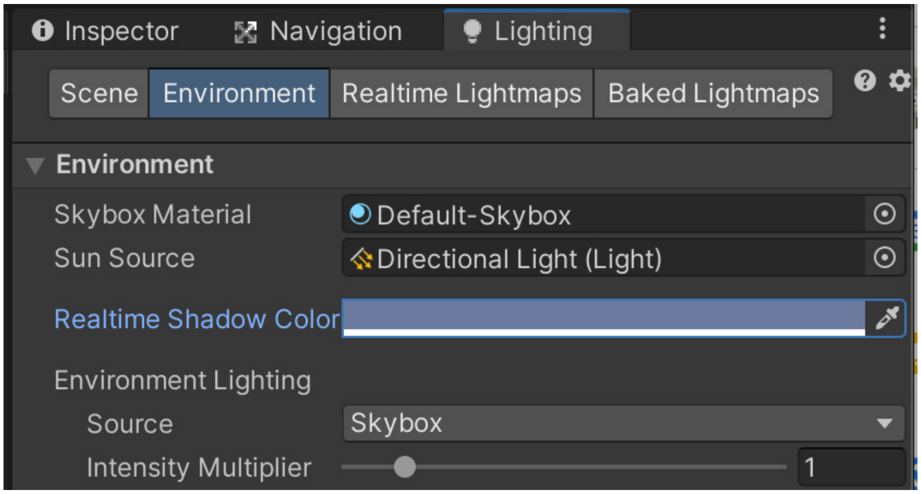
The choice between a Lit Shader and Simple Lit Shader is largely an artistic decision. It is easier for artists to get a realistic render using the Lit Shader, but if a more stylized render is desired, Simple Lit provides stellar results.

Comparing scenes rendered using different shaders: The top-left image uses the Lit Shader, the top-right, the Simple Lit Shader, and the bottom image, the Baked Lit Shader.

It's possible to implement your own custom lighting model by writing a custom shader or using Shader Graph (see the Additional tools chapter).

Lighting overview

Lights are divided into Main Light and Additional Lights in URP. The Main Light is the most significant Directional light. This is either the brightest light or the one set via **Window > Rendering > Lighting > Environment > Sun Source**.



Setting the Sun Source

Later in the guide, you'll learn how to use the URP Asset settings to set the number of dynamic lights that affect an object via the Object Per Light limit, which is capped at eight for the URP Forward Renderer. However, the number of dynamic lights that can be used per Camera is also limited by different hardware, as shown in the following table.

Camera light limits when using the URP Forward Renderer

Light type	Category	Maximum possible lights rendered (non-mobile)	Maximum possible lights rendered (mobile)	Maximum possible lights rendered (OpenGLES 2.0)	Supports shadows
Directional	Main	1	1	1	True
Spot	Additional	256*	32*	16*	True
Point	Additional	256*	32*	16*	True
Directional	Additional	256*	32*	16*	False

* All Additional Lights share the same budget.

When you cull a scene, choose up to the maximum number of supported dynamic lights for that frame. Meanwhile, when rendering an object, choose only the most significant of these lights to light each object dynamically.

Projects with a small number of dynamic lights might not encounter any issues, however, as you add more lights, you might experience light popping as different lights are dynamically culled. Of course, there is a performance cost to having more dynamic lights in a scene. Each dynamic light will need to be culled against the Camera and then sorted by priority. There is also the cost of rendering each light per object. As always, try to maintain a balance between fidelity and performance.

Real-time and Mixed Mode lights

Disabling a light set to Real-time or Mixed Mode does not stop it from being included in the light culling process. You might be limiting the lights for low-end hardware using the Settings Asset, but they will still cause a small performance hit due to light culling (this workflow is being corrected for future releases).

Real-time and Mixed Mode lights are first culled against the Camera frustum. If occlusion culling is enabled, lights hidden by other objects in the scene are also culled.

The visible list of lights that survive the culling process is then sorted by each light's distance to Camera. If there are visible lights, the limits in the table above come into play. For example, if you have 1,000 lights in the scene, but only 200 visible by the Camera, all those would fit the limit for non-mobile platforms.

Now the list of visible lights is culled per object. Lights are sorted by intensity at the pivot of the object – this way, brighter lights are prioritized first. If an object is affected by more than the maximum number of lights allowed per object, the excessive lights are discarded.

Consider the following options if you are hitting light limits:

— If the light's position and intensity are static, bake it and use Light Probes to add the light to the rendering of dynamic geometry. See the section on Light Probes for more information.

— Use Light Layers to limit which geometry is affected by which light.

— Limit the range of the light. This option does not apply to Directional lights, as they're global.

— Fake the lighting using emissive materials.

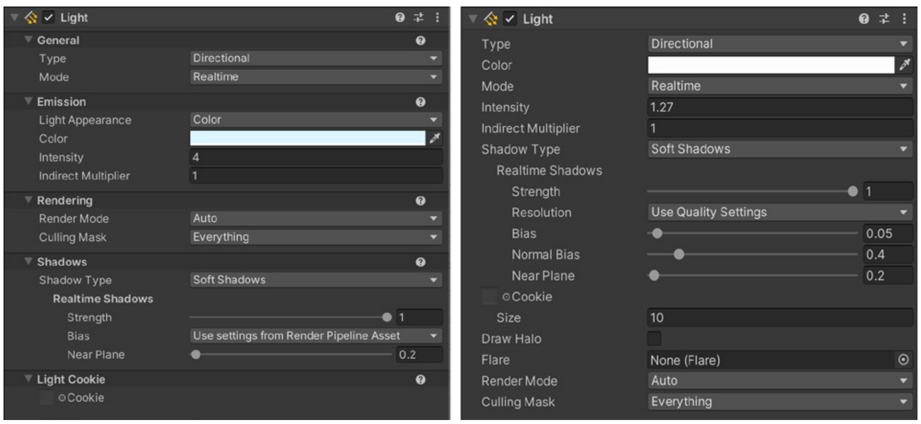
The light limits discussed here are those that apply with the Forward Renderer – the default renderer when using URP.

The Forward Renderer uses a single-pass approach to calculate the lighting of an object in a single draw call. This is a performant option, however, as GPU limitations restrict the number of lights that an object can consider when setting the color for a pixel. If you're targeting high-end hardware, you can avoid these limitations by using the Deferred Rendering Path in URP.

Light Inspector

The Light Inspector is one of three places where you can set up lighting.

Just as with the Built-in Render Pipeline, URP supports Directional, Spot, Point, and Area lights, though Area lights only work in Baked Indirect Mode. See the Light Mode section for more details.



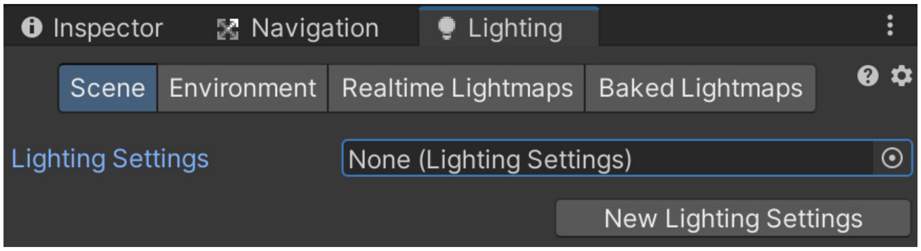
A side-by-side comparison of the Light Inspector panel in URP (left) and the Built-in Render Pipeline (right)

The image above shows how light properties are presented in the two versions of the Light Inspector. The URP version has five groupings of controls, based on whether the light is Directional or Point, and an additional Shape grouping for Spot and Area lights.

This table lists the differences between the URP and Built-in Render Pipeline Inspectors.

URP Light Inspector properties	**Description**	**Built-in Render Pipeline Light Inspector properties**
Light Appearance	Choose between Color or Filter and Temperature. Color sets the emitted light color. Filter and Temperature use both a color (filter) and a temperature to switch between cool and warm lighting.	NA
Bias	Bias controls shadow acne. The default is to use the URP Asset. Alternatively, you can set custom values using this Inspector.	Bias/Normal Bias
Light Cookie	If a texture is set to use a light cookie and the light type is Directional, then a new panel will allow you to control the x and y size of the cookie, as well as its offset. A cookie for a Point light must be a cubemap. URP supports colored cookies, whereas the Built-in Render Pipeline is greyscale only.	Cookie
Shape: Spot	You can now control both the inner and outer cone angles for Spot lights.	Spot Angle, Range
Shape: Area	This is used to control the shape of an Area light.	Shape, Width, Height, Radius
NA	This is easily reproduced using a billboard or a Fresnel shader controlling the alpha value of a sphere that sits at the center of the light. See the section on Halo light for more information.	Draw Halo
NA	Check out the Lens Flare section to see how to implement a Lens Flare in URP.	Flare

Lighting a new scene

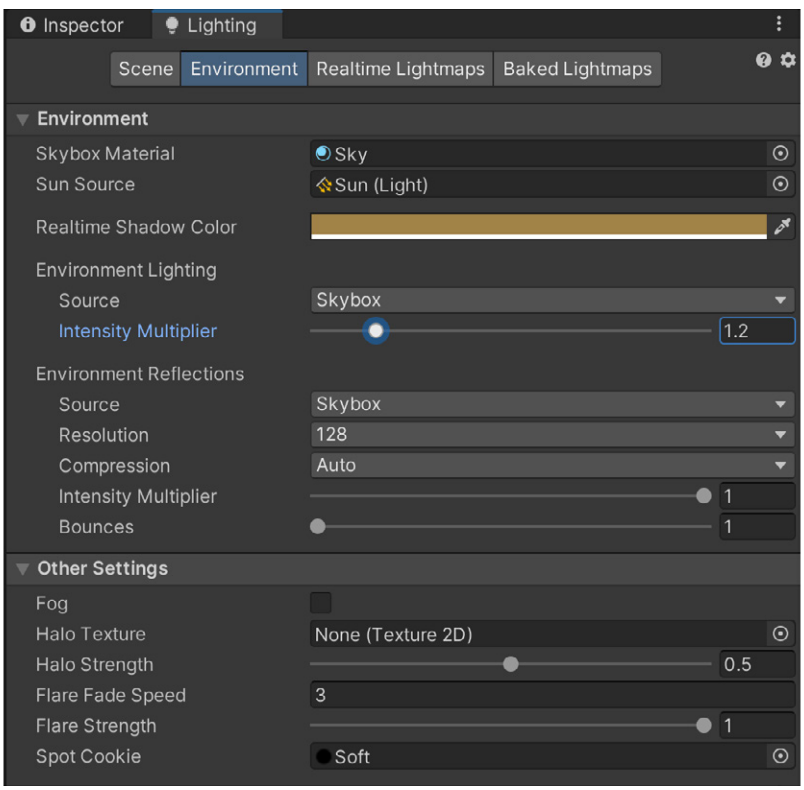


Creating a Lighting Settings Asset

The first step to lighting a new scene for URP is to create a new Lighting Settings Asset (see image above). Open **Window > Rendering > Lighting**, and once you're on the **Scene** tab, click **New Lighting Settings**, and give the new asset a name. The settings that you apply in Lighting panels are now saved to it. Switch between settings by switching the Lighting Settings Asset.

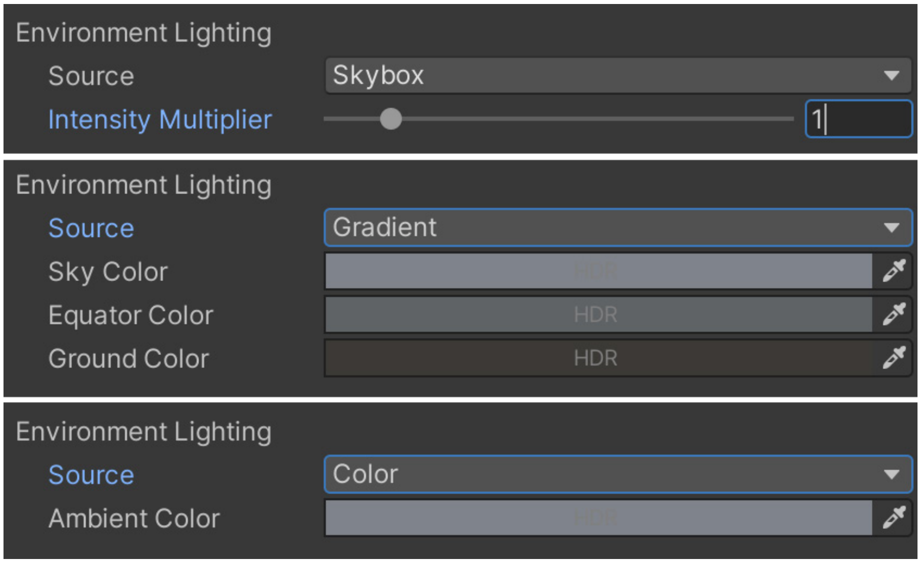
Ambient or Environment lighting

There is no change in the way that Ambient/Environment lighting is defined from the Built-in Render Pipeline to URP. The main ambient light is calculated from the panel accessible via **Window > Rendering > Lighting > Environment**.



The available settings for lighting in the Environment panel

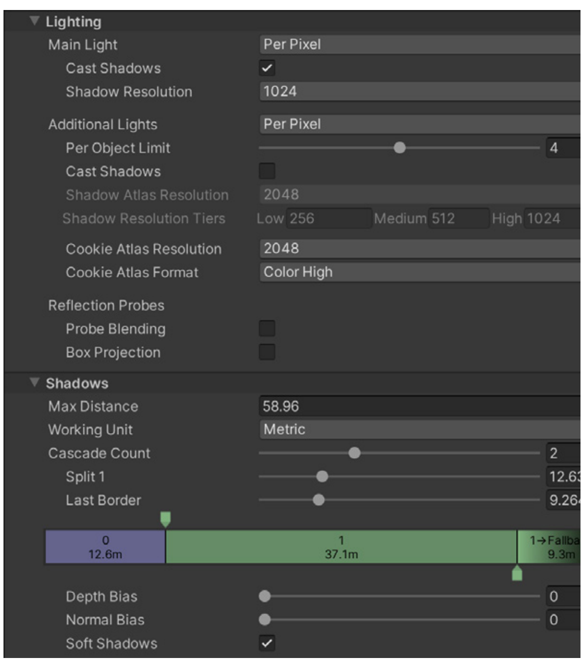
You can set **Environment Lighting** to use the scene's Skybox, with an option to adjust the Intensity, Gradient, or Color.



Environment Lighting options

Shadows

The biggest change from working with the Built-in Render Pipeline to URP lies in how you set up shadows.



The URP Asset

Shadow settings are no longer available via **Project Settings > Quality**. As discussed earlier, you need a Renderer Data object and a Render Pipeline Asset when using URP. The section on setting up a project for URP covers how to view your scene via Render Pipeline Asset, which you can use to define the fidelity of your shadows.

Main Light Shadow Resolution

The Lighting and Shadow groups in the URP Asset are key to setting up shadows in your scene. First, set the **Main Light Shadow** to **Disabled** or **Per Pixel**, then go to the checkbox to enable **Cast Shadows**. The last setting is the resolution of the shadow map.

If you've worked with shadows in Unity before, you know that real-time shadows require rendering a shadow map that contains the depth of objects from the perspective of the light. The higher the resolution of this shadow map, the higher the visual fidelity – though both more processing power and increased memory are required. Factors that increase shadow processing include:

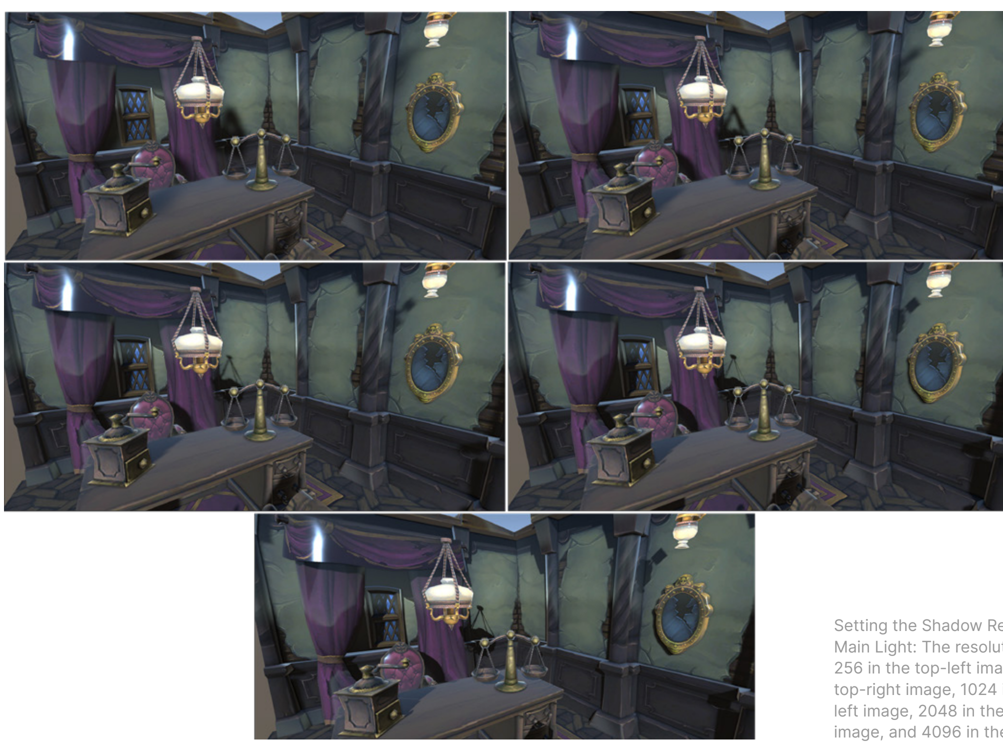
— The number of Shadow Casters rendered in the shadow map – this number for the Main Light depends on the Shadow Distance (far plane of shadow frustum)

— Shadow Receivers that are visible (you have to encompass them all)

— Shadow Cascades splits

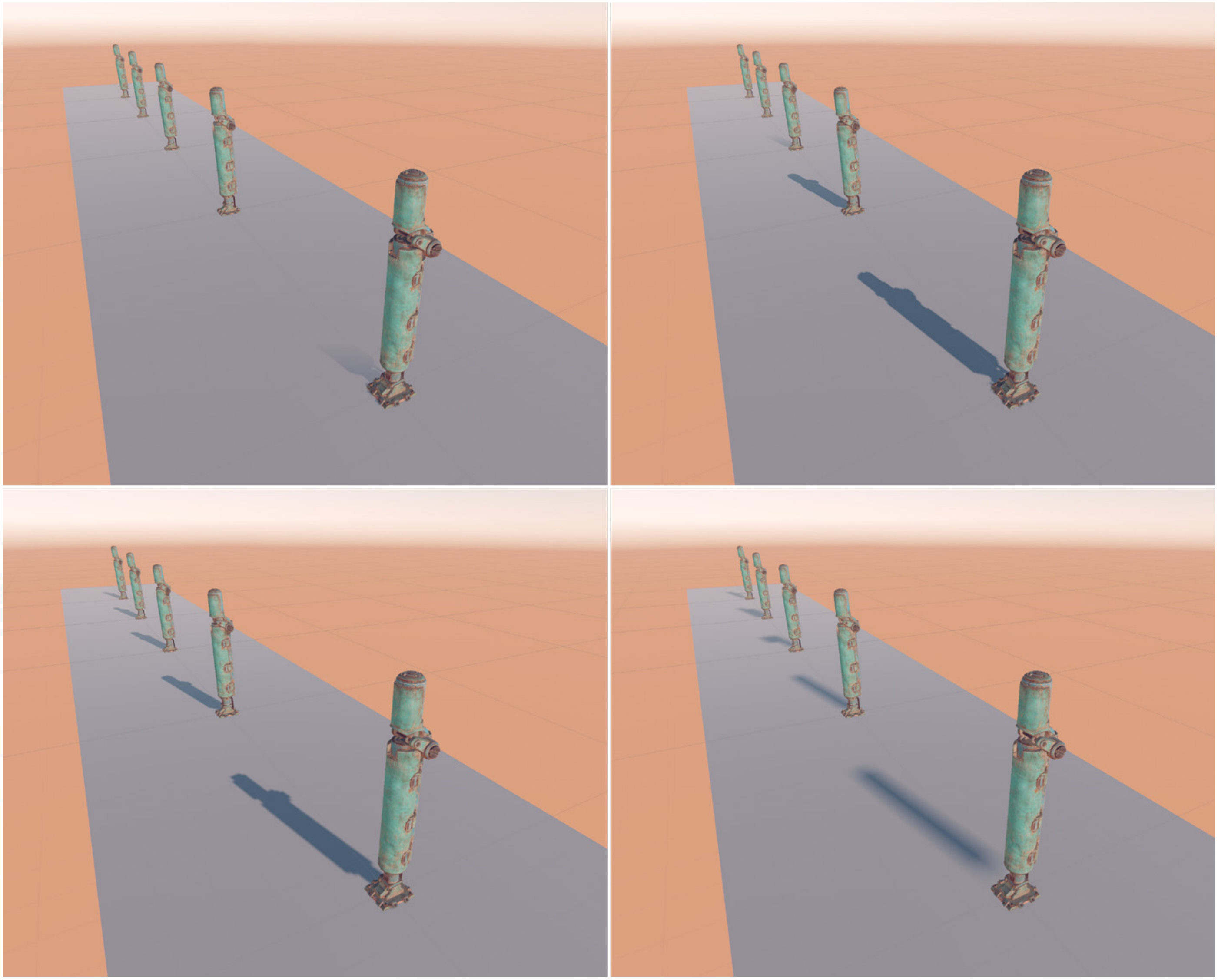
— Shadow filtering (Soft Shadows)

The highest resolution isn't always ideal. For example, the Soft Shadows option has the effect of blurring the map. In the following image of a cartoon-like haunted room, you can see that the chair in the foreground casts a shadow on the desk drawers, which appears too crisp when the resolution is greater than 1024.



Setting the Shadow Resolution for the Main Light: The resolution is set to 256 in the top-left image, 512 in the top-right image, 1024 in the middle-left image, 2048 in the middle-right image, and 4096 in the bottom image.

Main Light: Shadow Max Distance



Varying Max Distance for the Main Light Shadow: Top-left image – 10, top-right image – 30, bottom-left image – 60, bottom-right image – 400

Another important setting for the Main Light Shadow is Max Distance. This is set in scene units. In the image above, the poles are 10 units apart. The Max Distance varies from 10 to 400 units. Notice that only the first pole casts a shadow, and this is cut short at 10 units from the Camera location. At 60 units (bottom-left image), all shadows are in view – the shadow fidelity is adequate. When the Max Distance is much greater than the visible assets, the shadow map is being spread over too large an area. This means that the region in-shot has a much lower resolution than required.

The Max Distance property needs to relate directly to what the user can see, as well as the units used in the scene. Aim for the minimum distance that gives acceptable shadows (see note below). If the player only sees shadows from dynamic objects 60 units from the Camera, then set Max Distance to 60. When the Lighting Mode for Mixed Lights is set to Shadowmask, the shadows of objects beyond Shadow Distance are baked. If this was a static scene then you would see shadows on all objects, but only dynamic shadows would be drawn up to the Shadow Distance.

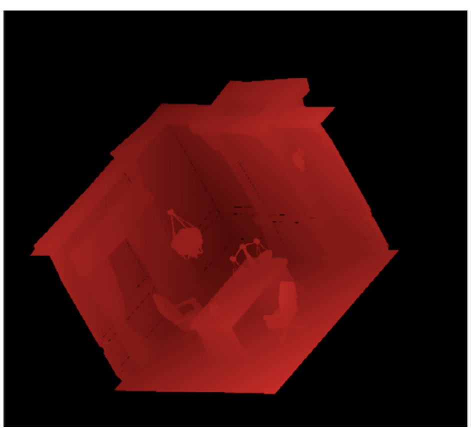
Note: URP only supports **Stable Fit** Shadow Projection, which relies on the user to set up the Max Distance. The Built-in Render Pipeline supports both Stable Fit and Close Fit for the Shadow Projection property. In the latter mode, the bottom-left and -right images would have the same quality, as Close Fit reduces the shadow distance plane to fit the last caster. The disadvantage is that, since Close Fit changes the shadow frustum "dynamically," it can cause a shimmer/dancing effect in the shadows.

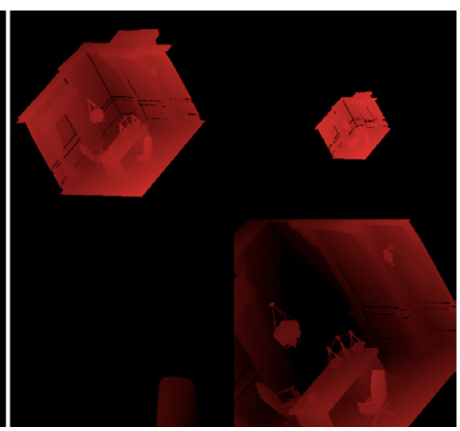
Shadow Cascades

As assets disappear into the distance due to perspective, it is convenient to decrease Shadow Resolution, thereby devoting more of the shadow map to shadows closer to the Camera. Shadow Cascades can help with this.

The images below show the shadow map of the scene with the chair and desk in the haunted room. The cascade count is 1 in the image to the left. The map takes up the whole area. In the image to the right, the cascade count is 4. Notice that the map includes four different maps, with each area receiving a lower resolution map.

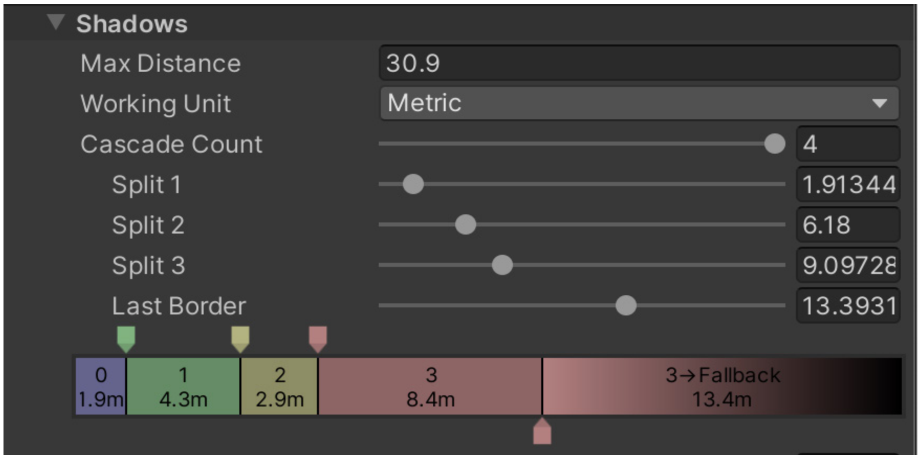
A cascade count of 1 is likely to give the best result for small scenes like this. But if your Max Distance is a large value, then a cascade count of 2 or 3 will give better shadows for foreground objects, as these receive a larger proportion of the shadow map. Notice that the chair in the left image is much bigger, resulting in a sharper shadow.





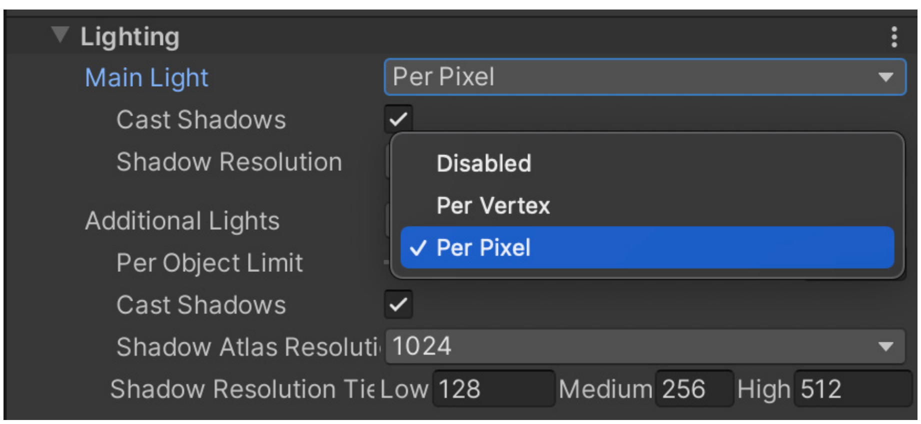
Shadow map when cascade count is set to 1 (left image) and 4 (right image)

You can adjust the start and end ranges for each section of the cascade using the draggable pointers, or by setting the units in the relevant fields (see following image). Always adjust Max Distance to a value that is a close fit for your scene and choose the slider positions carefully. If you use metric as the working unit, always choose the last cascade to be, at most, the distance of the last Shadow Caster.



Adjusting the range of a Shadow Cascade

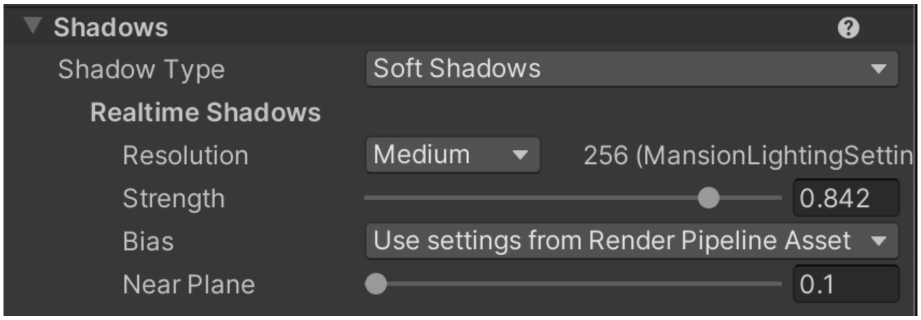
Additional Light Shadows



Settings available for Additional Lights in URP Asset

Having sorted the shadows for the Main Light, it's time to move on to **Additional Lights Mode**. Enable additional lights to cast shadows by setting the Additional Lights Mode for the URP Asset to **Per Pixel**. While the mode can be set to Disabled, Per Vertex, or Per Pixel (see above image), only the latter works with shadows.

Check the **Cast Shadows** box. Then, select the resolution of the **Shadow Atlas**. This is the map that will be used to combine all the maps for every light casting shadows. Bear in mind that a Point light casts six shadow maps, creating a cubemap, since it casts light in all directions. This makes a Point light the most demanding performance-wise. The individual resolution of an additional light shadow map is set using a combination of the three Shadow Resolution tiers, plus the resolution chosen via the Light Inspector when selecting the light in the Hierarchy window.

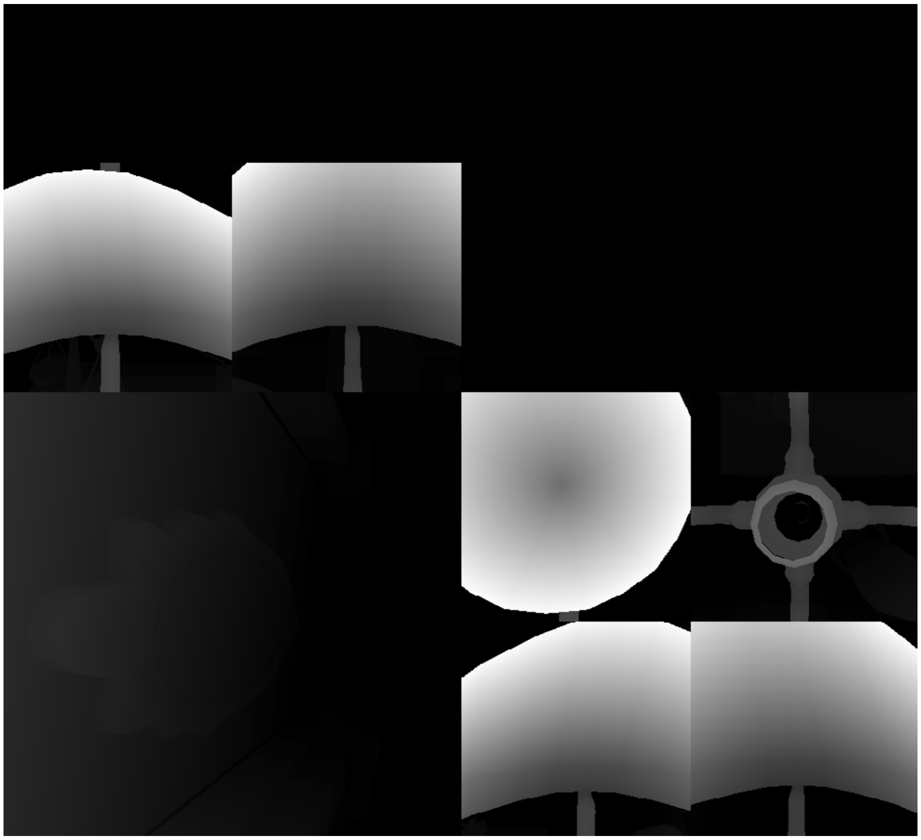


Shadows group in the Light Inspector

In the haunted room, there is a Spot light over the mirror and a Point light over the desk. There are also seven maps. To fit these seven maps onto a 1024px square map, the size of each map needs to be 256px or smaller. If you exceed this size, the resolution of shadow maps will shrink to fit the atlas, resulting in a warning message in the console.

Number of maps	Atlas tiling	Atlas size (multiply shadow tier size by)
1	1×1	1
2–4	2×2	2
5–16	4×4	4

Setting the Shadow Atlas size based on the number of Additional Lights shadow maps and the tier size chosen per map



Shadow Atlas for Additional Lights

The image above shows the six maps used by the Point light where the resolution is set to medium and the tier value to 256px. The Spot light has a resolution set to high, with a tier value of 512px.

This is a low-polygon version of the haunted room, lit with a Main Directional light, a Point light over the desk, and a Spot light over the mirror. All lights are real-time and casting shadows.

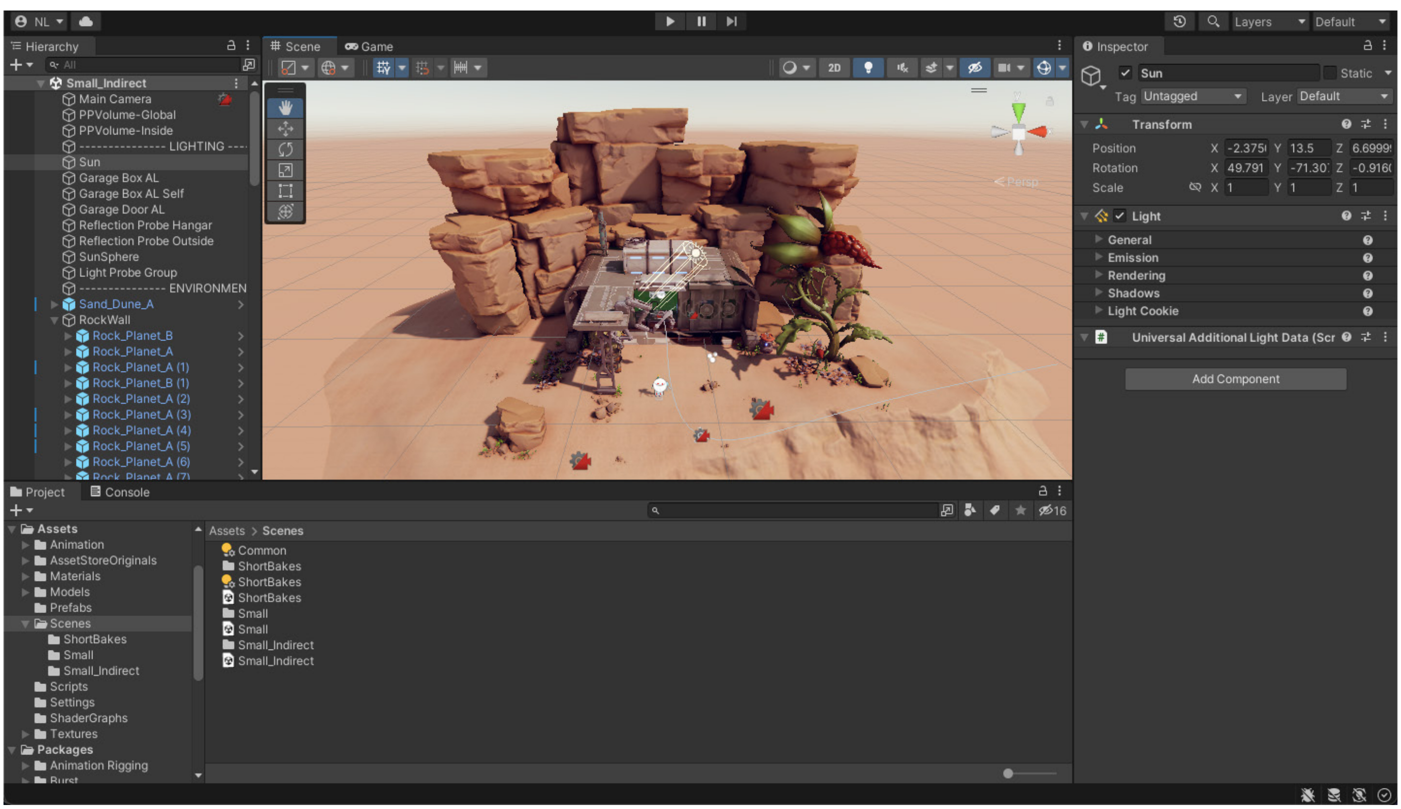
Light Modes

Environments have predominantly static geometry, so that if a light is static, you don't need to calculate the lighting and shadows for it repeatedly. You can calculate this once at design time, and then use that data when rendering the geometry. This is called lightmapping or baking.

The workflow for lightmapping is unchanged between the Built-in Render Pipeline and URP. Let's go through the steps using an FPS Sample project by Unity.

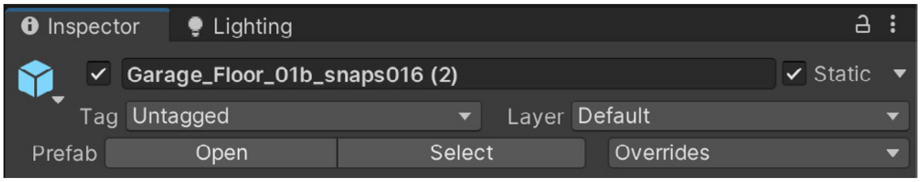
The following screenshots are from the Unity project FPS Sample: The Inspection, which you can download here. The scene demonstrates how to use real-time and baked lighting in URP.

Note: Low frequency refers to the fact that lightmaps are updated at a much lower rate than screen updates. Specular Lobes can only be computed by real-time lights. You can apply Global Illumination (GI) to dynamic objects by using Light Probes, but those also only capture low frequency diffuse light. The Built-in Render Pipeline supports Light Probe Proxy Volume (LPPV), which provides the same level of quality for Light Probes as lightmaps do for dynamic objects. However, in URP, LPPV is not supported due to it being a relatively slow system that doesn't scale. Instead, URP plans to support Adaptive Probe Volumes, which could replace lightmaps and work for both static and dynamic objects.

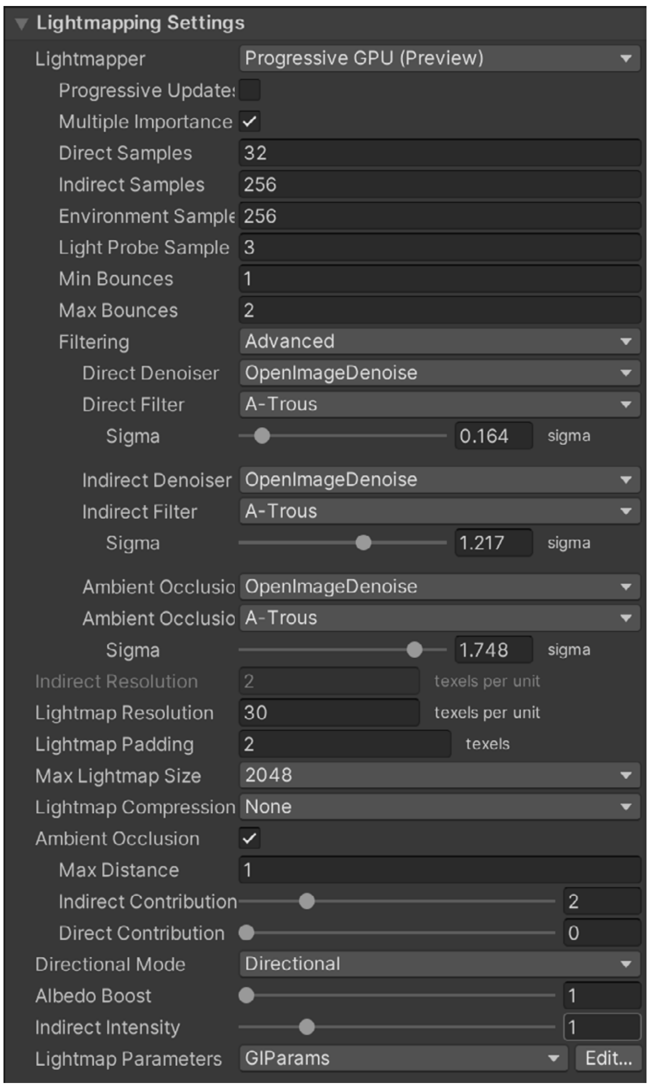


The scene from FPS Sample: The Inspection by Unity

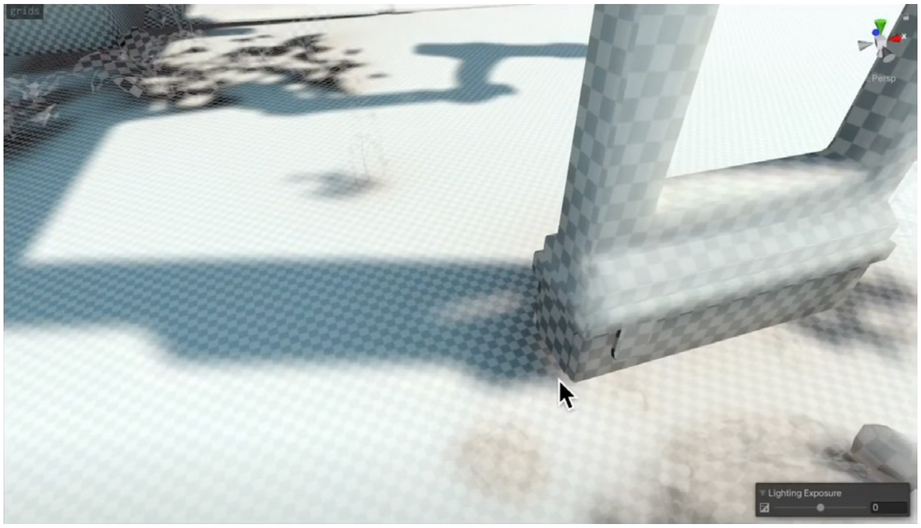
1. The scene from the FPS sample project contains largely static geometry. To include the geometry in lightmapping, click the **Static** box to the right side of the Inspector.



2. Choose the lightmapping settings via **Window > Rendering > Lighting > Scene**. Keep the Lightmap Resolution low while adjusting the settings. Once you have your desired settings, increase the value when generating the final lightmaps. Choose **Progressive GPU (Preview)** to speed up the lightmap generation, if your GPU supports it.

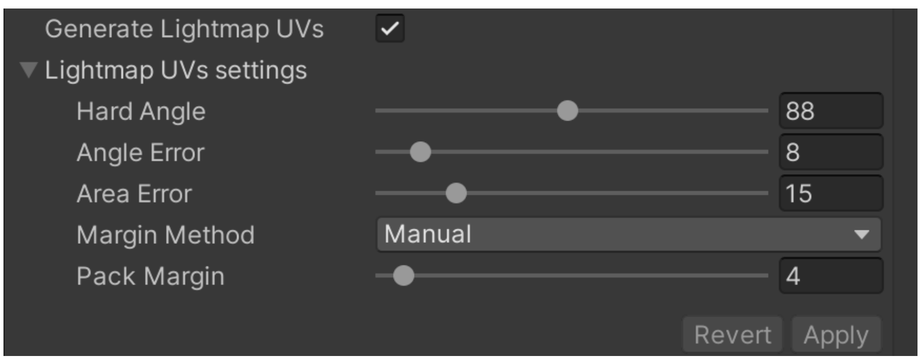


3. Filtering blurs the map to minimize noise. This can result in gaps in a shadow where one object meets another. Use **A-Trous** filtering to minimize this artifact. See Progressive Lightmapping documentation for more details on the settings available for lightmapping.

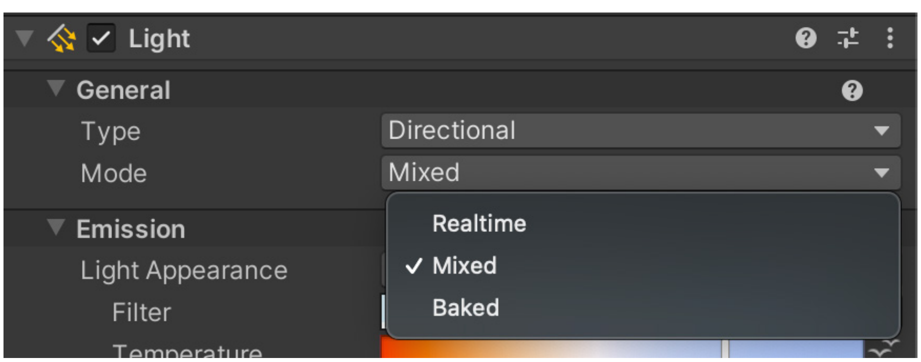


How filtering affects the shadow between objects

4. Make sure all static geometry has no overlapping UV values, or is generating lighting **UVs** on import.



5. Set **Light Mode** to **Baked** or **Mixed**. Select the light in the **Hierarchy** window and use the **Inspector**. Mixed Lights will illuminate dynamic objects as well as static ones.

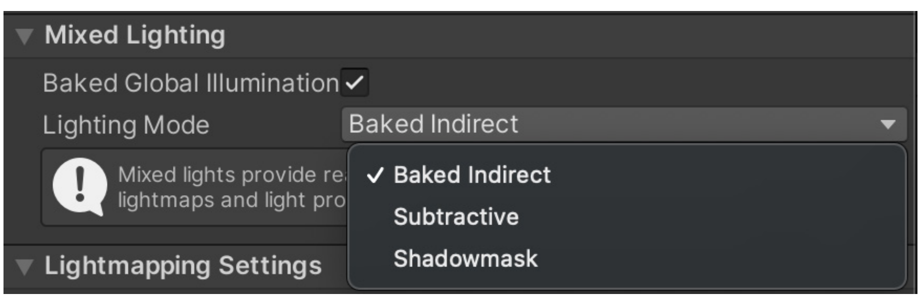


6. When using Mixed Lights, set the **Light Mode** to **Baked Indirect**, **Subtractive**, or **Shadowmask** via **Window > Rendering > Lighting > Scene**.

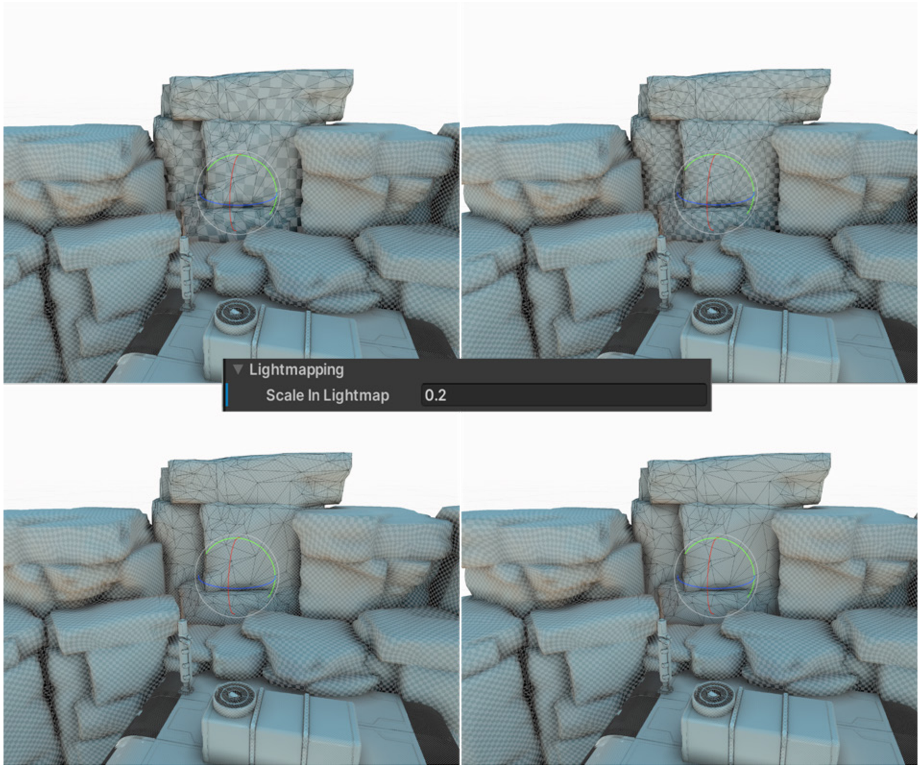
    a. **Baked Indirect:** Only the indirect light contribution will be baked into the lightmaps and Light Probes (the bounces of the lights only). Direct lighting and shadows will be real-time. This is an expensive option and not ideal for mobile platforms. However, it does mean that you get correct shadows and direct light for both static and dynamic geometry.

    b. **Subtractive:** Here, you bake the direct lighting from a Directional light set to Mixed into the static geometry, and subtract the lighting from shadows cast by dynamic geometry. This results in the static geometry unable to cast a shadow on dynamic objects, unless Light Probes are used, which can cause unpleasant visual discontinuities. URP calculates an estimate of the contribution of the light from the Directional Light and subtracts that from the baked Global Illumination. The estimate is clamped by the Real-time Shadow Color setting in the Environment section of the Lighting window, so the color subtracted is never darker than this color. Then choose the minimum color of your subtracted value and the original baked color. This is the most suitable option for low-end hardware.

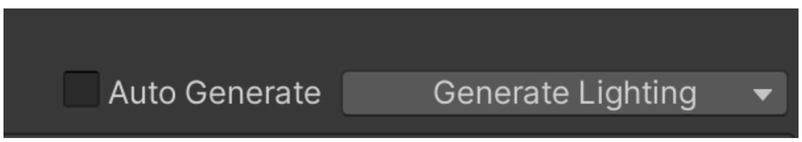
    c. **Shadowmask:** Though similar to Baked Indirect Mode, Shadowmask combines both dynamic and baked shadows, rendering shadows at a distance. It does this by using an additional Shadowmask texture and storing additional information in the Light Probes. This provides the highest fidelity shadows, but is also the most expensive option in terms of memory use and performance. Visually, it's identical to Baked Indirect for shots up close. The difference is apparent when looking in the far distance, making it well-suited for open-world scenes. Due to the processing cost, it's recommended for mid- to high-end hardware only.



7. Adjust the **Lightmap Scale** via **Asset > Inspector > Lightmapping > Scale In Lightmap**, so that distant objects take up less space on the lightmap. The following image shows the texel size of the background rock lightmap with a setting varying from 0.05 to 0.5.



8. Click **Generate Lighting** to bake. The baking time depends on the number of static objects, lights set to Mixed or Baked mode, and the settings chosen for lightmapping, particularly the Max Lightmap Size and the Lightmap Resolution.



More resources:

— Lightmapping documentation

— Configuring lightmaps from Unity Learn

— Lighting Settings Asset documentation

— Lighting Explorer documentation

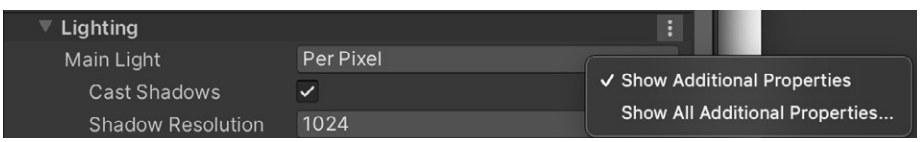
Light Layers

The Light Layers feature lets you configure certain lights to affect only specific GameObjects so you can emphasize and draw attention to them in a scene. In the image below, the syringe, a key collectable, appears in a shaded part of the scene. With a Light Layer, it becomes visible and helps ensure that the player doesn't miss picking it up.

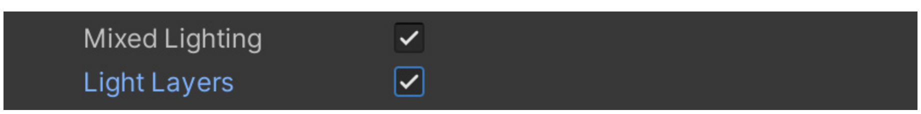
Highlighting an object using Light Layers

Here are the steps for setting up Light Layers.

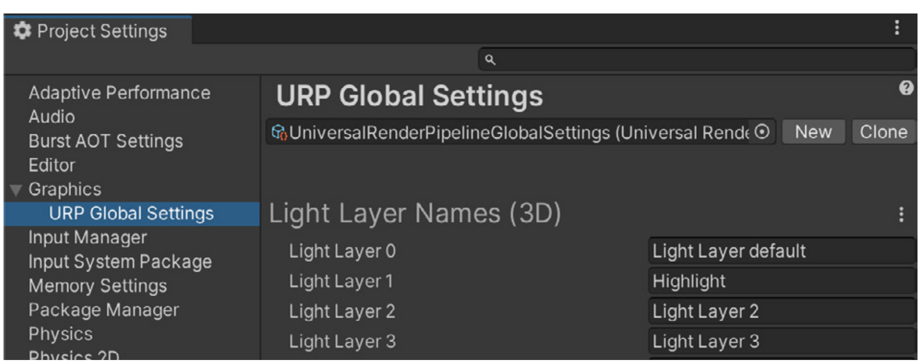
1. Select the **URP Asset**. In the Lighting section, click the vertical ellipsis icon **(⋮)** and select **Show Additional Properties**.



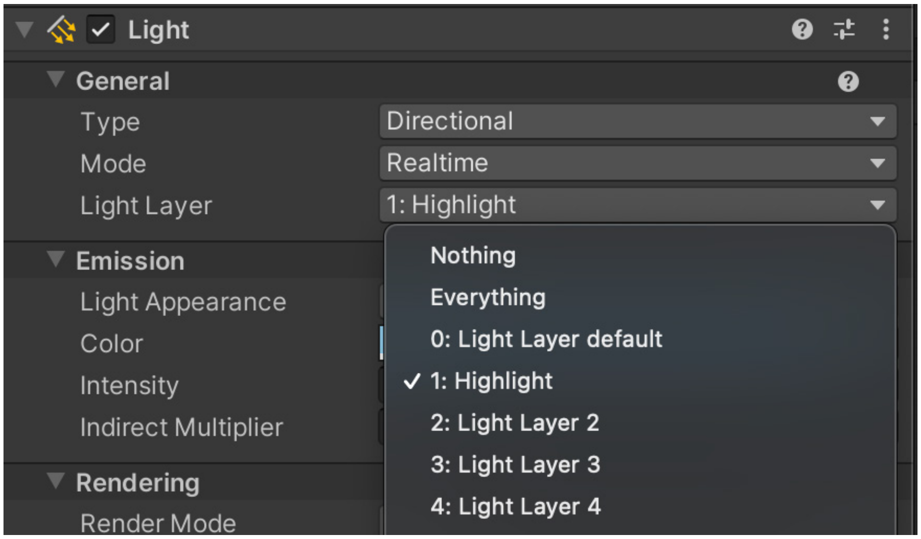
2. A new setting, **Light Layers**, will appear under the Lighting section.



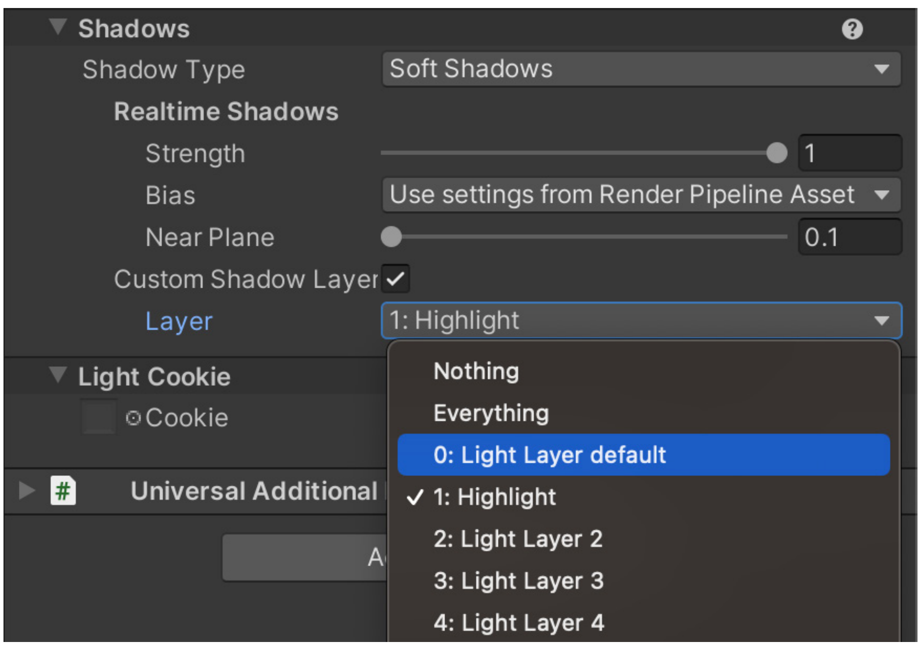
3. Rename a Light Layer via **Project Settings > Graphics > URP Global Settings**.



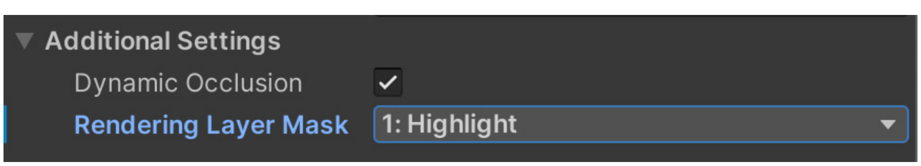
4. Now that Light Layers are enabled, the **Light Inspector** will include a **Light Layer** drop-down. A light can contribute to more than one layer.



5. With Light Layers enabled, you need to set up a custom shadow layer. The new light can cast shadows from the scene's **Main Light** or from its own frustum.



6. Lastly, select the object this applies to in the **Hierarchy** window and then set the **Rendering Layer Mask**.



This can also be dynamically set in code.

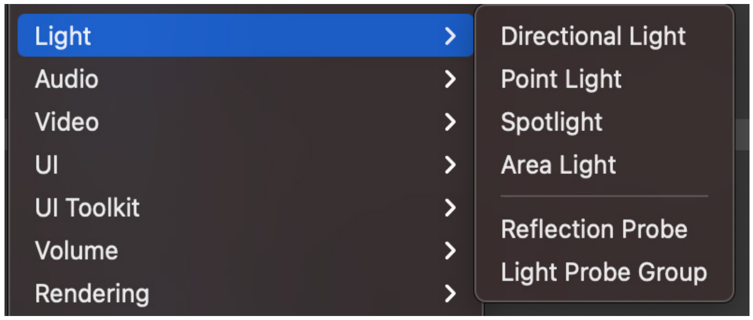
```
Renderer renderer = GetComponent<Renderer>();
int layerID = 1;
int mask = 1 << layerID;
renderer.renderingLayerMask = (uint)mask;
```

Light Probes

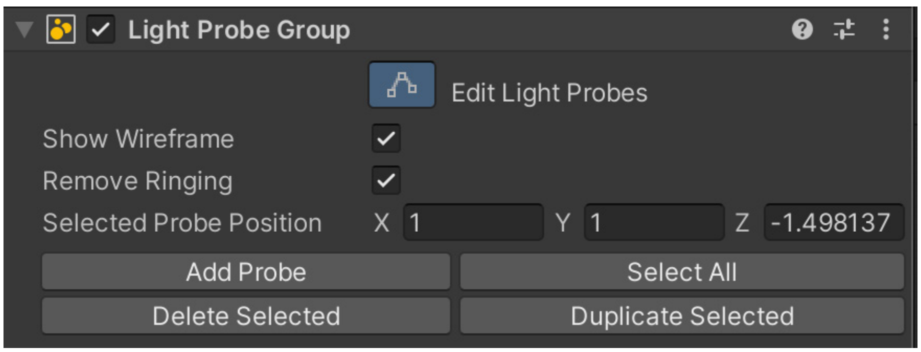
As covered in an earlier section, you can combine baked and dynamic objects in the Light Mode section using Mixed Lighting Mode. It's recommended to also add Light Probes to your scene when using this mode. Light Probes save the light data at a particular position within an environment when you bake the lighting by clicking **Generate Lighting** via **Window > Rendering > Lighting** panel. This ensures that the illumination of a dynamic object moving through an environment reflects the lighting levels used by the baked objects. In a dark area it will be dark, and in a lighter area it will be brighter. Below, you can see the robot character inside and outside of the hangar in the FPS Sample: The Inspection.

The robot inside and outside of the cave, with lighting level affected by Light Probes

To create Light Probes, right-click in the **Hierarchy** window and choose **Light > Light Probe Group**.



Initially, there will be a cube of Light Probes, eight in total. To view and edit the positioning of the Light Probes and add additional ones, select the **Light Probe Group** in the Hierarchy window, and in the Inspector click **Light Probe Group > Edit Light Probes**.



The Scene view will now be in an editing mode where only Light Probes can be selected. Use the Move tool to move them around.

Moving a Light Probe

Light Probes should be positioned, first, in an area where a dynamic object might move to, and second, where there is a significant change in lighting level. When calculating the lighting level for an object, the engine finds a pyramid of the nearest Light Probes and uses those to determine an interpolated value for the illumination level.

The nearest Light Probes for the selected crate

Positioning Light Probes can be time-consuming, but a code-based approach such as this one can speed up your editing, especially for a large scene.

Further details on how a Mesh Renderer works with Light Probes and how to adjust the configuration can be found in this documentation.

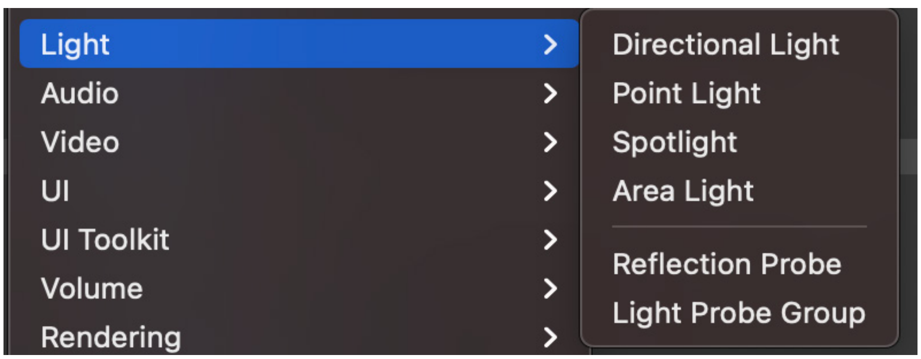
Reflection Probes

A ray-tracing tool, such as Maya or Blender, can take the time to accurately calculate the values for each frame pixel of a reflective surface. This process takes far too long for a real-time renderer, which is why shortcuts are often used.

Reflections in a real-time renderer use environment maps (pre-rendered cubemaps). Unity supplies a default map using the SkyManager. Having a single map as the source of reflections from all locations in a scene can lead to unconvincing reflections. Take the example of the robot shown in this section. If the metal parts of this character always reflect the sky, then it will look very strange when inside the hangar where the sky is not visible. This is where Reflection Probes are helpful.

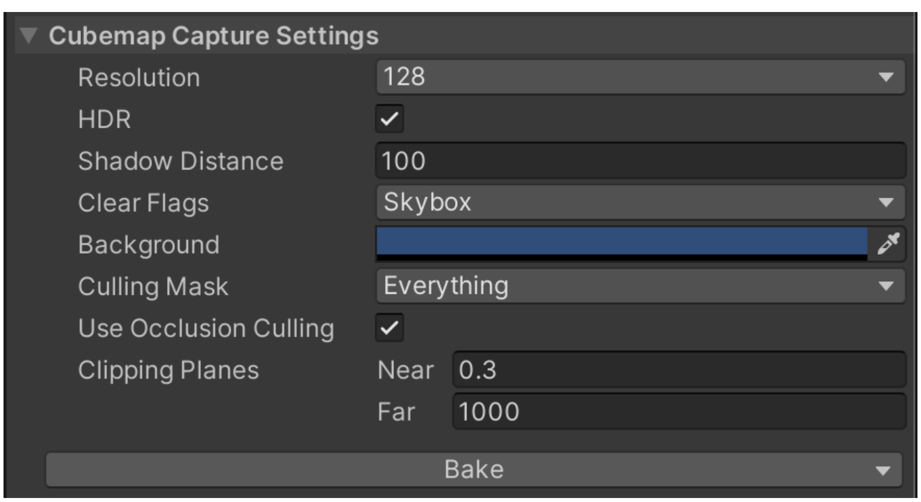
A Reflection Probe is simply a pre-rendered cubemap placed at a key position in the scene. You can use several Reflection Probes in a single scene. As a dynamic object moves through the scene, it can select the nearest Reflection Probe and use that as the basis of its reflections. You can also set up the scene to blend between probes.

To create a Reflection Probe, right-click the **Hierarchy** window and select **Light > Reflection Probe**.



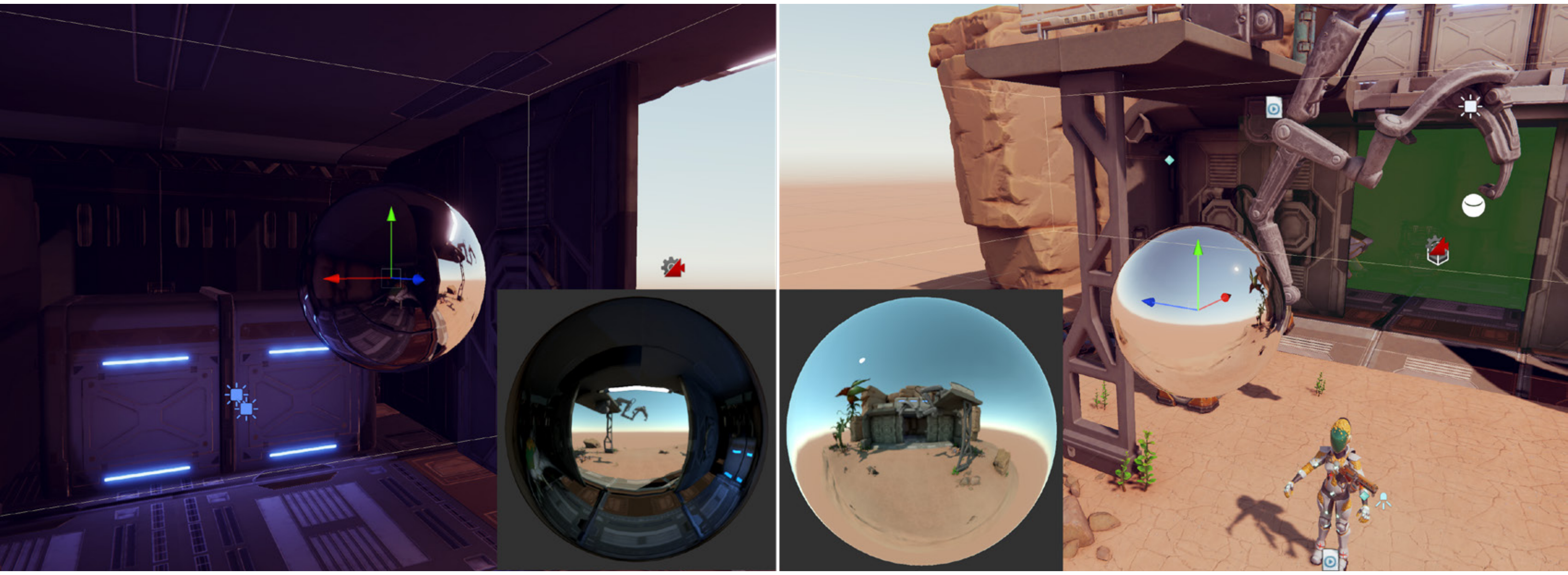
Creating a Reflection Probe

Then position the probe and adjust its settings. Once the probe is placed correctly and the settings are adjusted, click **Bake** to generate a cubemap.



Reflection Probe settings

The following image shows the two Reflection Probes used in FPS Sample: The Inspection, one inside the hangar and one outside.



Each Reflection Probe captures an image of its surroundings in a cubemap texture.

Reflection Probe blending

Blending is a great feature of Reflection Probes. You can enable blending via the **Renderer Asset Settings** panel.

Blending gradually fades out one probe's cubemap, while fading in the other as the reflective object passes from one zone to the other. This gradual transition avoids the situation where a distinctive object suddenly "pops" into the reflection as an object crosses the zone boundary.

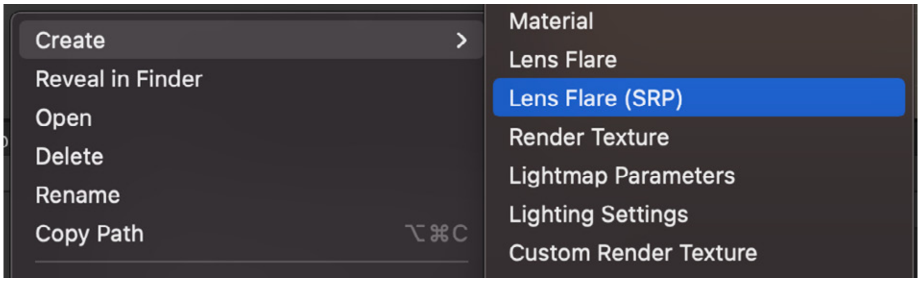
Box Projection

Normally, the reflection cubemap is assumed to be at an infinite distance from any given object. Different angles of the cubemap will be visible as the object turns, but it's not possible for the object to move closer or further away from the reflected surroundings. While this works well for outdoor scenes, its limitations show in an indoor scene. The interior walls of a room are clearly not an infinite distance away, and the reflection of a wall should get larger as the object nears it.

The Box Projection option enables you to create a reflection cubemap at a finite distance from the probe, allowing objects to show reflections of different sizes according to their distance from the cubemap's walls. The size of the surrounding cubemap is determined by the probe's zone of effect, depending on its Box Size property. For example, with a probe that reflects the interior of a room, you should set the size to match the dimensions of the room.

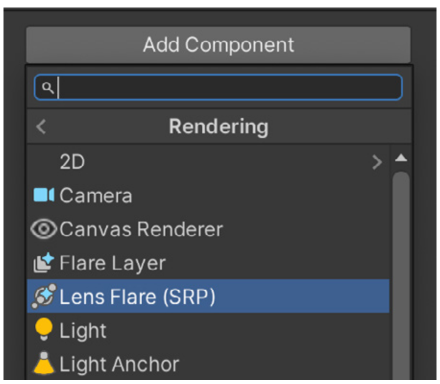
Lens Flare

The workflow for creating a Lens Flare has been updated for URP. The first step in configuring it is to create a Lens Flare (SRP) Data asset. Right-click in the **Project** window, in a suitable Assets folder, and select **Create > Lens Flare (SRP)**.



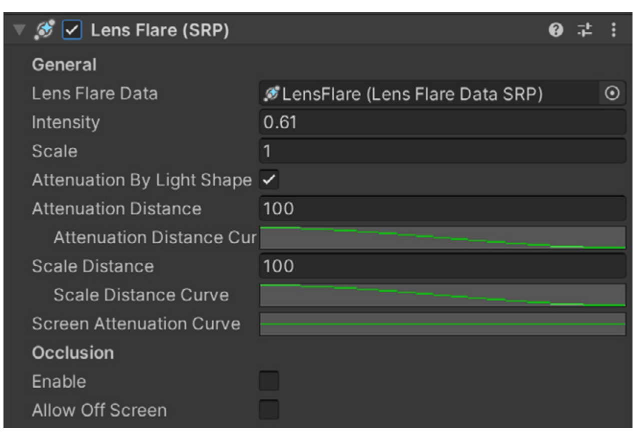
Creating a Lens Flare (SRP) Data asset

Use this asset to configure the shape of your flare. To render a Lens Flare, choose the light source that will cause the flare and then select **Add Component > Rendering > Lens Flare (SRP)**.



Setting up rendering for a Lens Flare

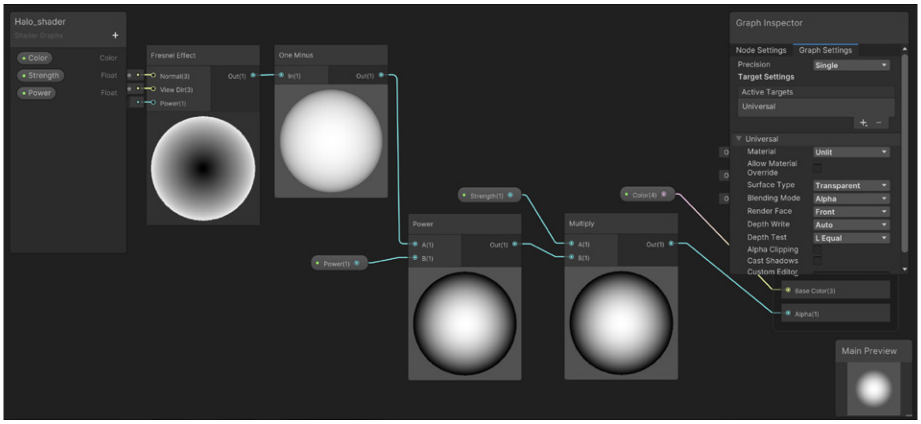
In the **Settings** panel for this component (see following image), assign the **Lens Flare Data asset** you created to the **Lens Flare Data property**.



Settings for the Lens Flare (SRP) component

Light Halos

The Draw Halo option is not available for lights in URP, but it’s easily mimicked with a billboard. Another option is to set the alpha transparency of a sphere. The first image below shows the Shader Graph for such a shader, and the second image depicts the result. For more information on using Shader Graph to create this shader, see the Additional tools chapter.

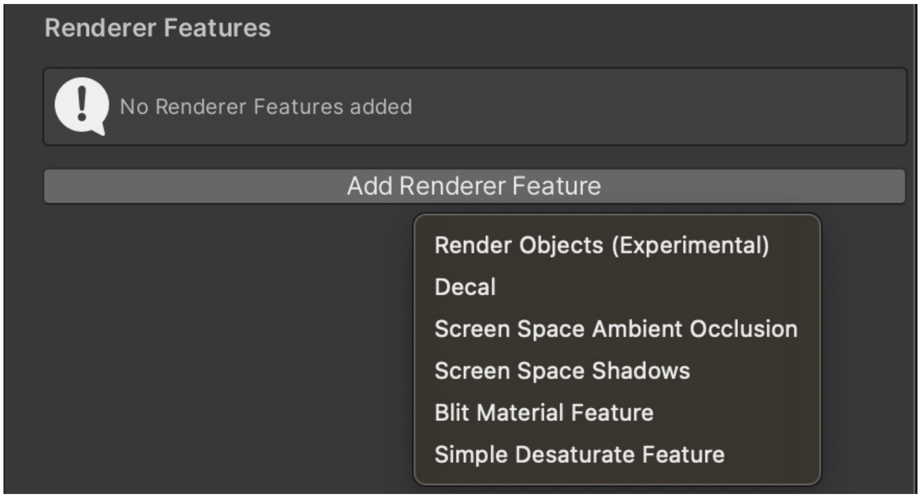


Fresnel transparency using Shader Graph

Light Halo using a sphere with a material using the Shader Graph shader from above

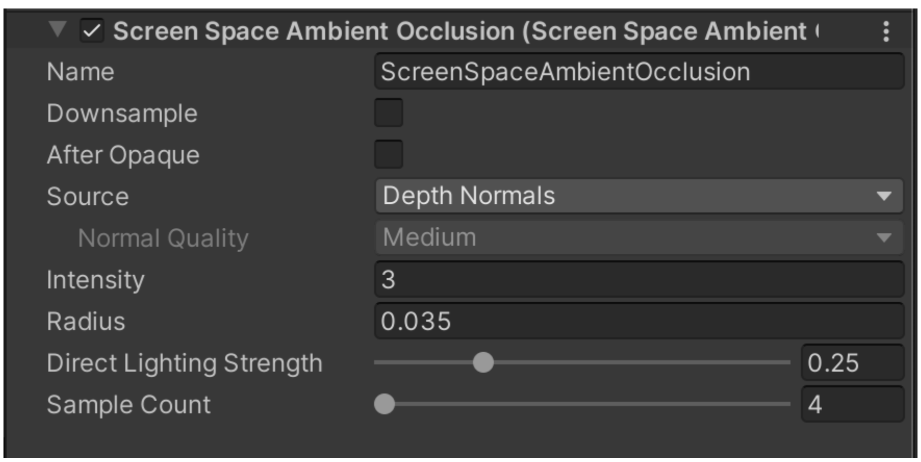
Screen Space Ambient Occlusion

Since ambient light does not consider geometry by default, high levels of ambient light can lead to unconvincing renders. In the real world, a narrow gap between two objects is likely to be darker than a much wider gap. Ambient Occlusion can help deal with this issue in your Unity project. To use it with URP, select the Renderer that the URP Asset is using. Go to **Add Renderer Feature** and choose **Screen Space Ambient Occlusion** (SSAO).



Add Renderer Feature

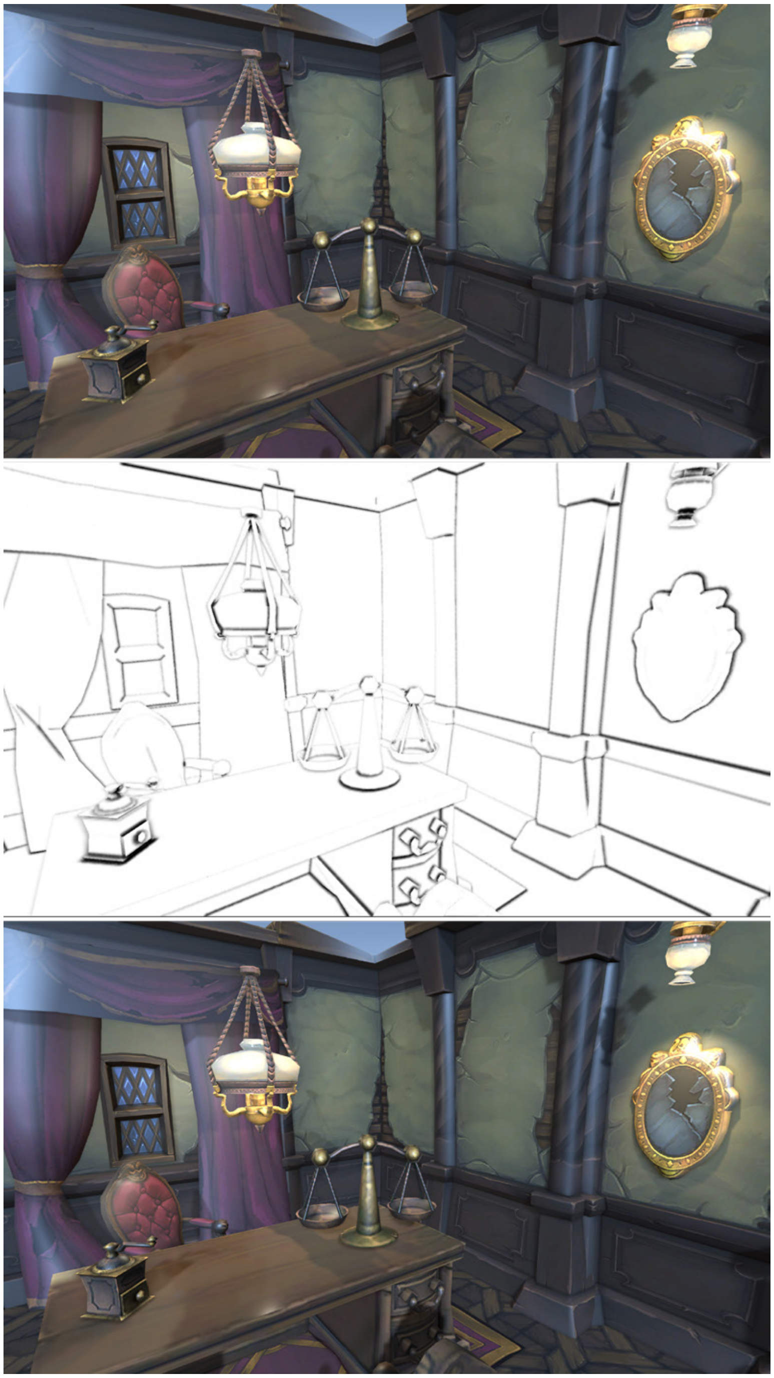
Either use the default SSAO settings or adjust as needed:



The SSAO settings

The effect adds shading to narrow gaps. Let's look at the following three images.

The top image has no SSAO. The middle image shows the calculated SSAO, while the bottom image shows the result of SSAO. Notice that the grinder and scales have a stronger edge where they meet the desk.



The haunted room screenshot, with no SSAO at the top, with SSAO applied in the middle, and rendered with SSAO at the bottom

SSAO is a post-processing technique covered in detail later in this guide.

SHADERS

This section is for users who want to convert an existing custom shader to work with URP and/or want to write a custom shader in code without using Shader Graph. It provides the information required to port both basic and advanced shaders to URP from the Built-in Render Pipeline. The tables included show helpful samples of available HLSL shader functions, macros, and so on. In each case, a link is provided to the relevant include containing many other useful functions.

For those who already have experience coding shaders, the includes provide you with a clear idea of what's available in HLSL to write compact and efficient shaders. After considering the information here, hopefully porting your shaders to URP won't seem so daunting.

Another approach is to use Shader Graph to create versions of your custom shaders. An introduction to Shader Graph is provided in the Additional tools section.

Comparing URP and Built-in Render Pipeline shaders

URP shaders use the ShaderLab structure, as seen in the code snippet below. As such, Property, SubShader, Tags, and Pass will all be familiar to shader coders.

```
SubShader {
   Tags {"RenderPipeline" = "UniversalPipeline" }
   Pass {
      HLSLPROGRAM
      ...
      ENDHLSL
   }
}
```

The basic structure of a SubShader block

The first thing to notice when comparing a URP shader with a Built-in Render Pipeline shader is the use of the key-value pair `"RenderPipeline" = "UniversalPipeline"` in the SubShader tag.

A SubShader tag with the name `RenderPipeline` tells Unity which render pipelines to use this SubShader with. The value of `UniversalPipeline` indicates that Unity should use this SubShader with URP.

Looking at the render Pass code, you'll see the shader code contained between the `HLSLPROGRAM / ENDHLSL` macros. This indicates the former CG (C for Graphics) shader programming language has been replaced by HLSL (High Level Shading Language) although the shader syntax and functionality are near-identical. Unity switched to HLSL a long time ago, so this shouldn't come as a surprise, but now the `CGPROGRAM / ENDCG` macros are not recommended. Using these macros implies using UnityCG.cginc. Mixing the SRP and Built-in Render Pipeline shader libraries in this way can cause several problems.

For URP, the shader code inside those passes is written in HLSL. Although most of the ShaderLab hasn't changed compared to the Built-in Render Pipeline, shaders written for the Built-in Render Pipeline are automatically disabled by URP.

The reason for this is the change in the internal lighting process. While the Built-in Render Pipeline performs separate shader passes for every light that reaches an object (multipass), the URP Forward Renderer evaluates all lighting in a light loop in a single pass. This change leads to different data structures that store light data and new shading libraries with new conventions.

Unity will use the first SubShader block that is supported on the GPU. If the first SubShader block doesn't have a "`RenderPipeline`" = "`UniversalPipeline`" tag, it won't run in the URP. Instead, Unity will try to run the next SubShader, if any. If none of the SubShaders are supported, Unity will render the well-known magenta error shader.

A SubShader can contain multiple Pass blocks, but each of them should be tagged with a specific LightMode. As URP uses a single-pass Forward Renderer, only the first "`UniversalForward`" Pass supported by the GPU will be used to render objects in Forward rendering.

As covered earlier, using **Window > Rendering > Render Pipeline Converter** converts Built-in Render Pipeline shaders to URP shaders for all materials automatically. But what about custom shaders?

Custom shaders

Custom shaders require some work when upgrading to URP. Listed below are the actions needed to perform on legacy shaders as part of the upgrading process.

We made this step-by-step video tutorial on how to convert a custom unlit Built-in shader to URP, including a Unity project to follow.

Includes

Replace .cginc includes files with the following HLSL equivalents:

Built-in Render Pipeline	**HLSL**
UnityCG.cginc	Github link
AutoLight.cginc	Github link Github link

Other useful HLSL includes

Space transform-related functions are found in this include, which is added by default when you use Core.hlsl.

HLSL space transform functions

URP helper function	Description
`float4x4 GetObjectToWorldMatrix()`	Returns UNITY_MATRIX_M matrix that converts from Object to World Space This is the equivalent of Built-in Render Pipeline `unity_ObjectToWorld`.
`float4x4 GetWorldToObjectMatrix()`	Returns UNITY_MATRIX_I_M matrix that converts from World to Object Space This matrix is the inverse of UNITY_MATRIX_M. It is the equivalent of Built-in Render Pipeline `unity_WorldToObject`.
`float4x4 GetWorldToHClipMatrix()`	Returns UNITY_MATRIX_VP matrix that converts from World to Clip Space
`float4x4 GetViewToHClipMatrix()`	Returns UNITY_MATRIX_P matrix that converts from View to Clip Space
`float3 TransformObjectToWorld(float3 positionOS)`	Given a position in Object Space, returns the position in World Space
`float3 TransformObjectToWorldDir(float3 dirOS, bool doNormalize = true)`	Given a direction in Object Space, returns the direction in World Space
`float3 TransformWorldToObject(float3 positionWS)`	Given a position in World Space, returns the position in Object Space
`float3 TransformWorldToView(float3 positionWS)`	Given a position in World Space, returns the position in View Space
`real3x3 CreateTangentToWorld(real3 normal, real3 tangent, real flipSign)`	Create a Tangent to World matrix given a normal and a tangent
`real3 TransformTangentToWorld(real3 normalTS, real3x3 tangentToWorld, bool doNormalize = false)`	Given a normal in Tangent Space, returns a normal in World Space
`real3 TransformWorldToTangent(real3 normalWS, real3x3 tangentToWorld, bool doNormalize = true)`	Given a normal in World Space, returns a normal in Tangent Space

Notation for the space type at the end of the variable name:

— WS: World Space
— TS: Tangent Space
— VS: View Space
— OS: Object Space

Other shader functions, including fog and UV, can be found in this include, which is added by default when you use Core.hlsl. The following table lists a few examples.

URP helper function	Description
`VertexPositionInputs GetVertexPositionInputs(float3 positionOS)`	Given a position in Object Space, returns a struct containing position in World, View, and Clip Space This function should be used only in vertex shader.
`VertexNormalInputs GetVertexNormalInputs(float3 normalOS)`	Given a normal in Object Space, returns a struct with the World Space normal, tangent, and bitangent vectors These vectors can be used to construct a Tangent to World matrix using CreateTangentToWorld. Returns input. tangentWS, input. bitangentWS, input. normalWS.
`float3 GetCameraPositionWS()`	Returns the Camera position in World Space This is similar to the Built-in Render Pipeline's _WorldSpaceCameraPos variable.
`float3 GetViewForwardDir()`	Returns the forward (central) direction of the current view in World Space
`float3 GetWorldSpaceViewDir(float3 positionWS)`	Computes the World Space view direction (pointing toward the viewer)

Shader helper functions are fundamental for shader coding. They not only save you time, but are highly optimized implementations of commonly used calculations. This include contains many helper functions related to:

— Platforms-specific functions

— Common math functions

— Texture utilities

— Texture format sampling

— Depth encoding/decoding

— Space transformations

— Terrain/brush heightmap encoding/decoding and miscellaneous utilities

Some of them are listed in the table below. The type `real` is set in the file; depending on various flags, it could be a half or a float.

Helper function	**Helper function**
`real DegToRad(real deg)`	`real RadToDeg(real rad)`
`bool IsPower2(uint x)`	`real FastACosPos(real inX)`
`real FastASin(real x)`	`real FastATan(real x)`
`uint FastLog2(uint x)`	`real3 Orthonormalize(real3 tangent, real3 normal)`
`real Pow4(real x)`	`float4x4 Inverse(float4x4 m)`
`float ComputeTextureLOD(float2 uv, float bias = 0.0)`	`float Linear01Depth(float depth, float4 zBufferParam)`

Preprocessor macros

Preprocessor macros are handy and regularly used. When porting the Built-in Render Pipeline shaders to new URP shaders, you'll need to replace the Built-in Render Pipeline macros with their URP equivalents.

This table highlights a few examples.

Built-in Render Pipeline	**URP**
`UNITY_PROJ_COORD(a)`	Replace with `a.xy/a.w`
`UNITY_INITIALIZE_OUTPUT(type, name)`	`ZERO_INITIALIZE(type, name)`
Shadow mapping*	
`UNITY_DECLARE_SHADOWMAP(tex)`	`TEXTURE2D_SHADOW_PARAM(textureName, samplerName)**`
`UNITY_SAMPLE_SHADOW(tex, uv)`	`SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord3)`
`UNITY_SAMPLE_SHADOW_PROJ(tex, uv)`	`SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord4.xyz/coord4.w)`
Texture/sampler declaration* **	
`UNITY_DECLARE_TEX2D(name)`	`TEXTURE2D(textureName); SAMPLER(samplerName);`
`UNITY_DECLARE_TEX2D_NOSAMPLER(name)`	`TEXTURE2D(textureName);`
`UNITY_SAMPLE_TEX2D_SAMPLER(name,samplername,uv)`	`SAMPLE_TEXTURE2D(textureName, samplerName, coord2)`

Notes for table:

* Shadow mapping macros need this shadow include.

** The _PARAM are macros that can be used to declare functions with texture and sampler arguments. Check out this document for more information.

*** For Built-in Render Pipeline texture/sampler declaration, read this documentation.

LightMode tags

The LightMode tag defines the role of Pass in the lighting pipeline. In the Built-in Render Pipeline, most shaders that need to interact with lighting are written as Surface Shaders with all the necessary details taken care of. However, custom shaders in the Built-in Render Pipeline need to use the `LightMode` tag to specify how the Pass is considered in the lighting pipeline.

The table below indicates the correspondence between the `LightMode` tags used in the Built-in Render Pipeline and the tags that URP expects. Several legacy Built-in Render Pipeline tags are not supported in URP: `PrepassBase`, `PrepassFinal`, `Vertex`, `VertexLMRGBM`, and `VertexLM`. At the same time, there are other tags in URP with no equivalent in the Built-in Render Pipeline.

Built-in Render Pipeline (read more here)	**Description**	**URP** (read more here)
`Always`	Always rendered; no lighting applied	-
`ForwardBase`	Used in Forward rendering; Ambient, main Directional light, vertex/SH lights, and lightmaps applied	`UniversalForward`
`ForwardAdd`	Used in Forward rendering; Additive per-pixel lights applied, one Pass per light	`UniversalForward`
`Deferred`	Used in Deferred Shading; renders G-buffer	`UniversalGBuffer`
`ShadowCaster`	Renders object depth into the shadow map or a depth texture	`ShadowCaster`
`MotionVectors`	Used to calculate per-object motion vectors	`MotionVectors`
	URP uses this tag value in the Forward Rendering Path; the Pass renders object geometry and evaluates all light contributions.	`UniversalForwardOnly`
-	URP uses this tag value in the 2D Renderer; the Pass renders objects and evaluates 2D light contributions.	`Universal2D`
-	The Pass renders only depth information from the perspective of a Camera into a depth texture.	`DepthOnly`
-	This Pass is executed only when baking lightmaps in the Unity Editor; Unity strips this Pass from shaders when building a Player.	`Meta`
-	Use this tag value to draw an extra Pass when rendering objects; it is valid for both the Forward and Deferred Rendering Paths. URP uses this tag value as the default value when a Pass does not have a `LightMode` tag.	`SRPDefaultUnlit`

When writing a shader for URP, it's a good idea to look at the provided shaders and the Passes they use. The following code example shows some of the code from the Lit Shader. The complete shader is here.

The `UniversalForward` and `ShadowCaster` Passes involve many pragmas and two include files. Examining the code in the include files will help you create the custom version that suits your needs.

```
//  Forward pass. Shades all light in a single pass. GI + emission
+ Fog
Pass
{
   // Lightmode matches the ShaderPassName set in
   // UniversalRenderPipeline.cs. SRPDefaultUnlit and passes with
   // no LightMode tag are also rendered by Universal Render Pipe-
line
   Name "ForwardLit"
   Tags{"LightMode" = "UniversalForward"}

   Blend[_SrcBlend][_DstBlend]
   ZWrite[_ZWrite]
   Cull[_Cull]

   HLSLPROGRAM
   #pragma exclude_renderers gles gles3 glcore
   #pragma target 4.5
   …

   #pragma vertex LitPassVertex
   #pragma fragment LitPassFragment

   #include "Packages/com.unity.render-pipelines.universal/Shad-
ers/LitInput.hlsl"
   #include "Packages/com.unity.render-pipelines.universal/Shad-
ers/LitForwardPass.hlsl"
   ENDHLSL
}

Pass
{
   Name "ShadowCaster"
   Tags{"LightMode" = "ShadowCaster"}

   ZWrite On
   ZTest LEqual
   ColorMask 0
   Cull[_Cull]

   HLSLPROGRAM
   #pragma exclude_renderers gles gles3 glcore
    …

   #pragma vertex ShadowPassVertex
   #pragma fragment ShadowPassFragment

   #include "Packages/com.unity.render-pipelines.universal/Shad-
ers/LitInput.hlsl"
   #include "Packages/com.unity.render-pipelines.universal/Shad-
ers/ShadowCasterPass.hlsl"
   ENDHLSL
}
```

Note: A great resource for users planning to write shaders for URP is this tutorial by Cyanilux.

PIPELINE CALLBACKS

A great feature of SRPs is that you can add code at just about any stage of the rendering process using a C# script. Scripts can be injected at stages such as:

— Rendering shadows
— Rendering prepasses
— Rendering G-buffer
— Rendering Deferred lights
— Rendering opaques
— Rendering Skybox
— Rendering transparents
— Rendering post-processing

You can inject scripts in the rendering process via the **Add Renderer Feature** option in the Inspector for the **Universal Renderer Data Asset**. Remember, when using URP, there is a Universal Renderer Data object and a URP Asset. The URP Asset has a Renderer List with at least one Universal Renderer Data object assigned. It is the asset you assign in **Project Settings > Graphics > Scriptable Render Pipeline Settings**.

If you are experimenting with multiple setting assets for different scenes, then attaching the following script to your Main Camera can be useful. Set the **Pipeline Asset** in the Inspector. Then it will switch the asset when the new scene is loaded.

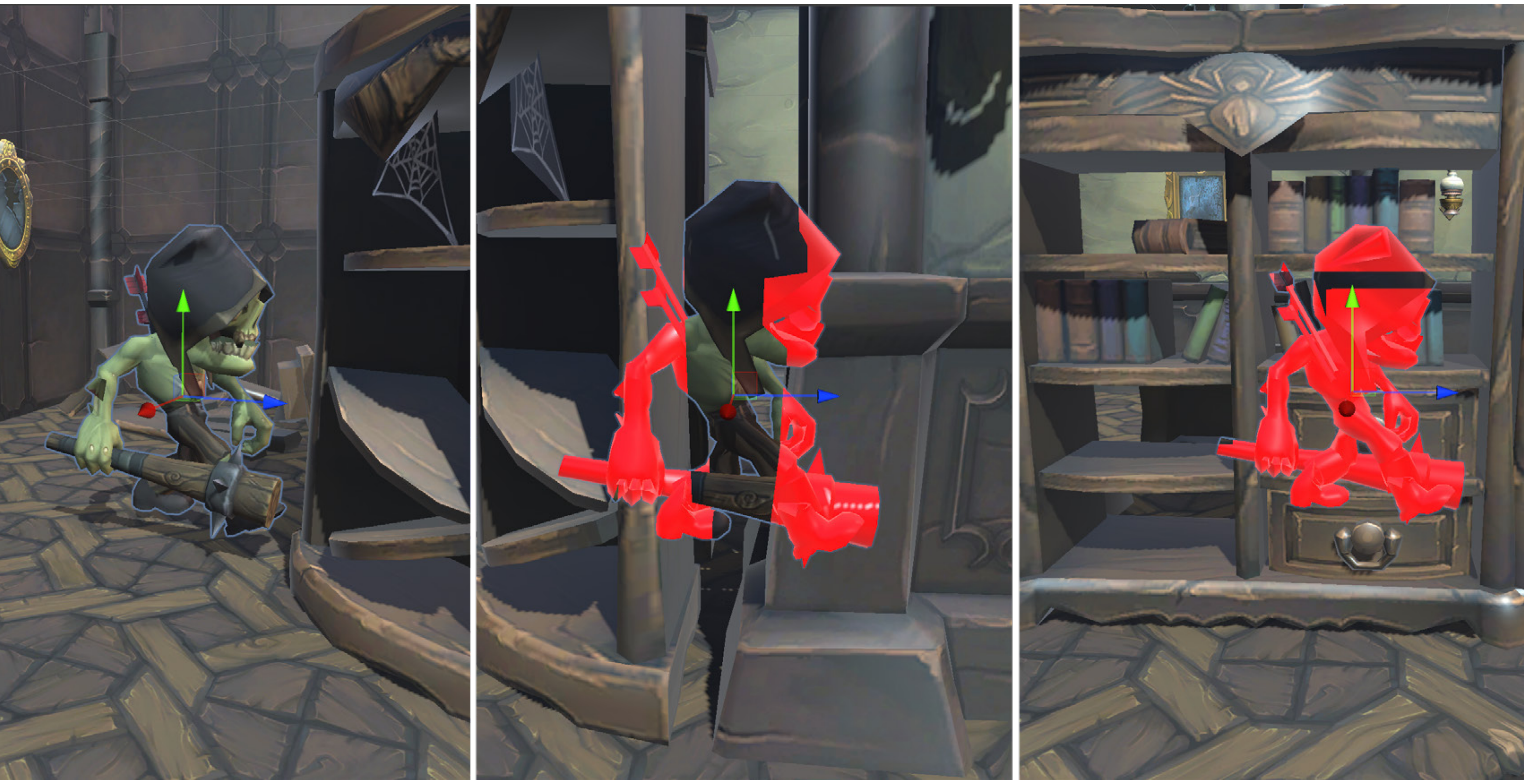
```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

[ExecuteAlways]
public class AutoLoadPipelineAsset : MonoBehaviour
{
        public UniversalRenderPipelineAsset pipelineAsset;

        // Start is called before the first frame update
        void OnEnable()
        {
        if (pipelineAsset)
        {
                GraphicsSettings.renderPipelineAsset = pipelineAs-
set;
        }
        }
}
```

Script to switch Universal Render Pipeline Asset on scene load

The next section covers two different types of Renderer Features, one for artists and the other for experienced programmers.

Render Objects

A common problem in games is losing sight of the player character as they disappear behind environment objects. You could attempt to move the Camera so that the character is always in view, or adjust the environment to be as open as possible. But such options are not always available. A good trick is to show a silhouette of the character when an environment model appears between the character and the Camera, as shown in the image below.

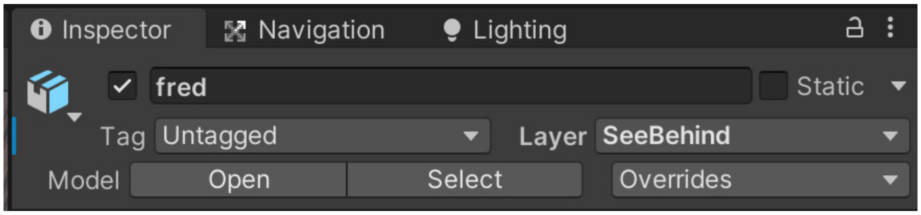


Showing a silhouette when an environment model masks the character

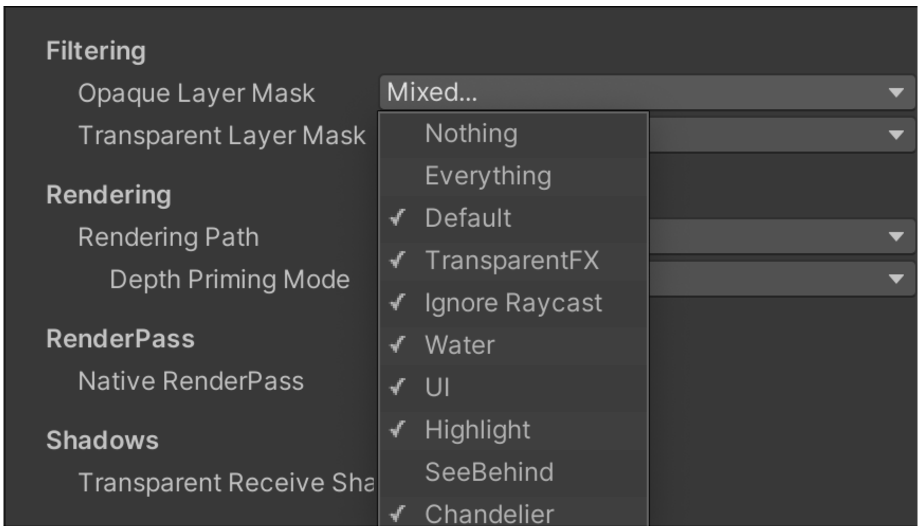
Here's how you can you create this silhouette:

1. First, you need a material to use when the character is masked. Create a material and set the shader to **Universal Render Pipeline > Lit or Unlit** (the previous image shows the Lit option). Set the **Surface Inputs > Base Map** color. In this example, the material is called Character.

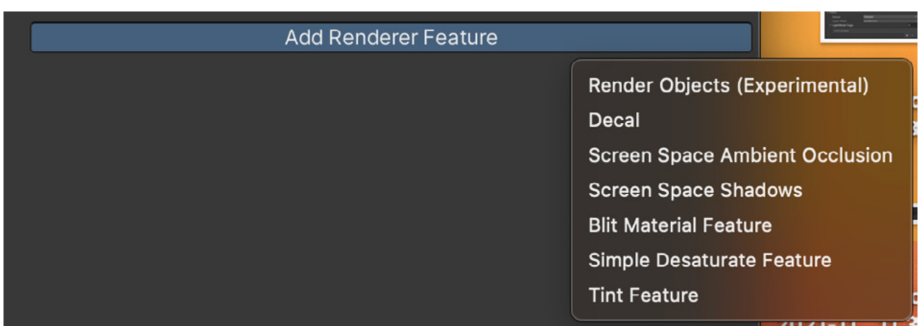
2. To avoid rendering the character more times than necessary, let's place it on a special layer. Select the character, add a **SeeBehind** layer to the Layers list and select it for the character.



3. Select the **Renderer Data** object used by the URP Asset. Go to the **Opaque Layer Mask** and exclude the SeeBehind layer. The character will then disappear.

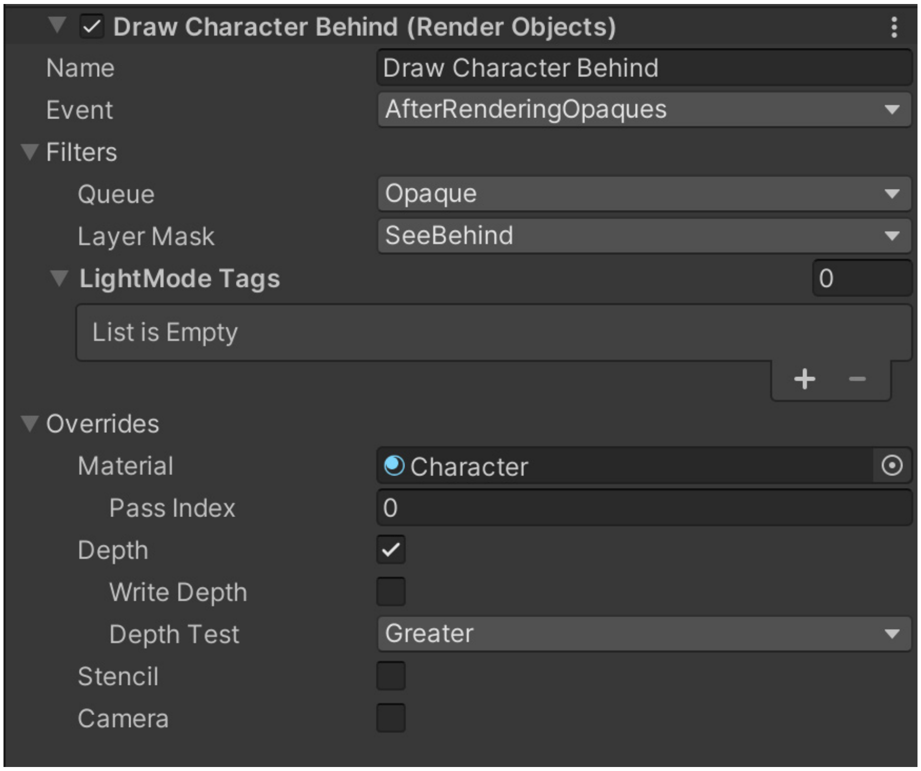


4. Click **Add Renderer Feature** and select **Render Objects (Experimental)**.

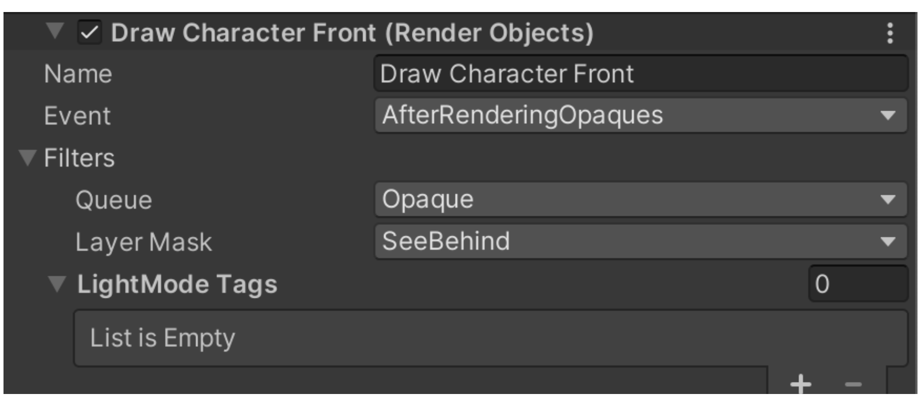


5. Fill out the settings for this Render Object's **Pass**. Give it a name and choose when the render should be triggered. In this example, it's called AfterRenderingOpaques.

Set the **Layer Mask** to the **SeeBehind** layer, which was the layer chosen for the character. Expand the **Overrides** and set the material created in step 1. You'll want to use **Depth** when rendering, without having to update the depth buffer by writing to it. Set the **Depth Test** to **Greater** so that this Pass only renders when the distance to the rendered pixel is further from the Camera than the distance currently stored in the depth buffer.



6. At this stage, you only see the silhouette of the character when it's behind another object. You don't see the character at all when it's in full view. To fix this, add another **Render Objects** feature. This time you don't need to update the Overrides panel. This Pass will draw the character when not masked by another object.



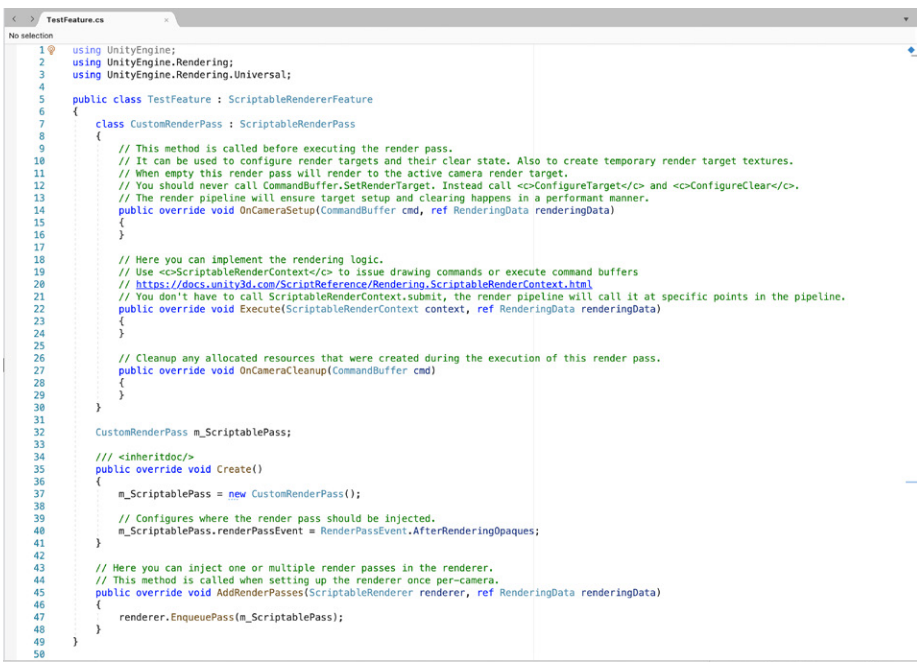
The silhouette trick is a good example of using the URP workflow to add effects that are difficult to achieve with the Built-in Render Pipeline workflow due to its reliance on coding.

Renderer Feature

A Renderer Feature can be used at any stage in URP to affect the final render. Let's go through a simple example of adding a post-processing effect. In a project using the Built-in Render Pipeline, you would have to add a Graphics. Blit using the OnRenderImage callback. This example uses the version of the function with a material to process each pixel in the image.

1. Start by finding a suitable folder in the project Assets folder. Right-click and choose **Create > Rendering > URP Renderer Feature**. Give it the name **TintFeature**.

2. Double-click the default **TintFeature** file. It is a C# script containing boilerplate for a Renderer Feature.



```
TestFeature.cs
No selection
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

public class TestFeature : ScriptableRendererFeature
{
    class CustomRenderPass : ScriptableRenderPass
    {
        // This method is called before executing the render pass.
        // It can be used to configure render targets and their clear state. Also to create temporary render target textures.
        // When empty this render pass will render to the active camera render target.
        // You should never call CommandBuffer.SetRenderTarget. Instead call <c>ConfigureTarget</c> and <c>ConfigureClear</c>.
        // The render pipeline will ensure target setup and clearing happens in a performant manner.
        public override void OnCameraSetup(CommandBuffer cmd, ref RenderingData renderingData)
        {
        }

        // Here you can implement the rendering logic.
        // Use <c>ScriptableRenderContext</c> to issue drawing commands or execute command buffers
        // https://docs.unity3d.com/ScriptReference/Rendering.ScriptableRenderContext.html
        // You don't have to call ScriptableRenderContext.submit, the render pipeline will call it at specific points in the pipeline.
        public override void Execute(ScriptableRenderContext context, ref RenderingData renderingData)
        {
        }

        // Cleanup any allocated resources that were created during the execution of this render pass.
        public override void OnCameraCleanup(CommandBuffer cmd)
        {
        }
    }

    CustomRenderPass m_ScriptablePass;

    /// <inheritdoc/>
    public override void Create()
    {
        m_ScriptablePass = new CustomRenderPass();

        // Configures where the render pass should be injected.
        m_ScriptablePass.renderPassEvent = RenderPassEvent.AfterRenderingOpaques;
    }

    // Here you can inject one or multiple render passes in the renderer.
    // This method is called when setting up the renderer once per-camera.
    public override void AddRenderPasses(ScriptableRenderer renderer, ref RenderingData renderingData)
    {
        renderer.EnqueuePass(m_ScriptablePass);
    }
}
```

Default code for a Renderer Feature

3. Add these properties to the **CustomRenderPass** class. The material will contain the shader you apply to the current state of the rendered image.

```
private Material material;
private RenderTargetIdentifier source;
private RenderTargetHandle tempTexture;
```

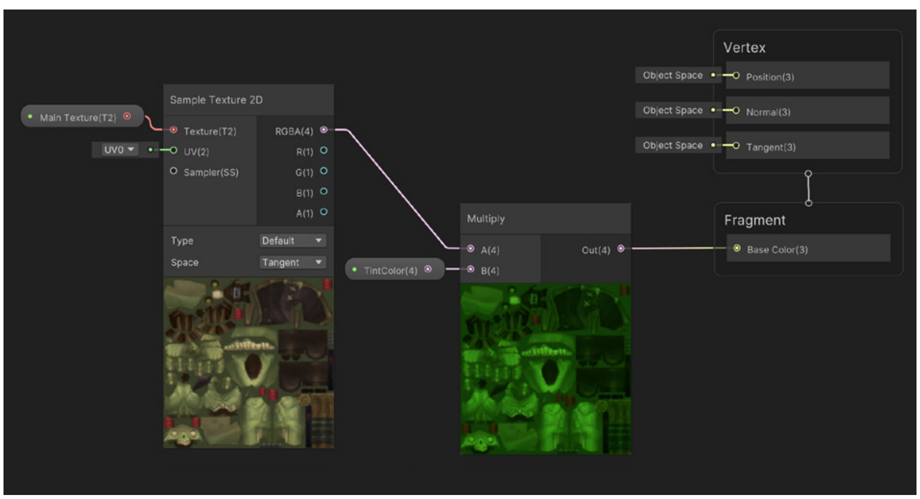
4. Add a constructor to the CustomRenderPass to initialize the current rendered source texture and the material to use when processing this texture. You'll also need to initialize a temporary texture to store the result of processing the current Render Texture with your material.

```
public CustomRenderPass(Material material) : base()
{
this.material = material;
tempTexture.Init("_TempTintTexture");
}
```

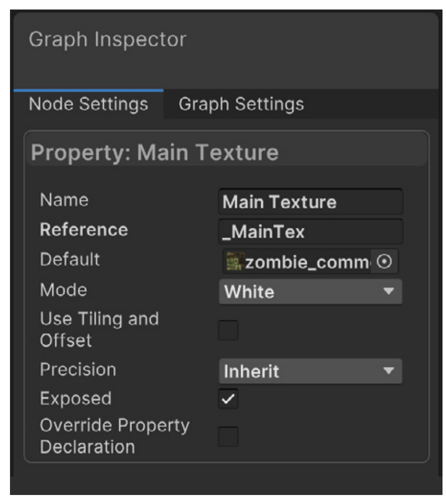
5. Add a **SetSource** method to initialize the source property of the CustomRenderPass class.

```
public void SetSource(RenderTargetIdentifier source)
{
this.source = source;
}
```

6. Create a new Shader Graph shader. Call it Tint and set up the nodes as in the image below. It uses two properties, a texture and a color. It's important to set the Reference of the Texture to **_MainTex**. See the second image below. This ensures that the code in the **TintFeature** finds the right Render Texture.



The Tint Shader Graph



Setting the Reference for the texture

7. Add the following code to the **Create** method. This function is called when the TintFeature is created. It will be used to initialize a new instance of the CustomRenderPass class using a new Material created from the Tint shader. Pass the material to the custom constructor. You also need to set where this feature is used in the render pipeline using the **renderPassEvent** property of a **ScriptableRendererFeature**.

```
var material = new Material(Shader.Find("Shader Graphs/Tint"));
m_ScriptablePass = new CustomRenderPass(material);
 m_ScriptablePass.renderPassEvent = RenderPassEvent.AfterRenderingOpaques;
```

8. Now that you have created an instance of the CustomRenderPass, add it to the render queue. Add the next code snippet in the **AddRenderPasses** method. Then set the source to the **renderer.cameraColorTarget** and enqueue the pass.

```
m_ScriptablePass.SetSource(renderer.cameraColorTarget);
renderer.EnqueuePass(m_ScriptablePass);
```

9. Before the pass can do anything, it needs to get the temporary texture. Add the next code snippet to **OnCameraSetup**:

```
RenderTextureDescriptor cameraTextureDesc =
        renderingData.cameraData.cameraTargetDescriptor;
cameraTextureDesc.depthBufferBits = 0;
cmd.GetTemporaryRT(tempTexture.id, cameraTextureDesc, FilterMode.Bilinear);
```

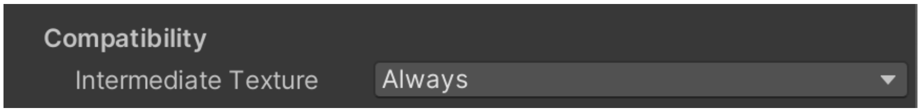
10. Since you have a texture, you need to release it. Add this code to **OnCameraCleanup**:

```
cmd.ReleaseTemporaryRT(tempTexture.id);
```

11. Now that everything is initialized, you can do the actual work of copying the current Render Texture using a material to process the result and passing it back to the source. Add this code to the **Execute** method:

```
CommandBuffer cmd = CommandBufferPool.Get("TintFeature");

Blit(cmd, source, tempTexture.Identifier(), material, 0);
Blit(cmd, tempTexture.Identifier(), source);

context.ExecuteCommandBuffer(cmd);
CommandBufferPool.Release(cmd);
```

12. To see the effect in action, select the **Renderer Data** object and click **Add Renderer Feature**. TintFeature will appear in the list. Make sure to also set the **Compatibility > Intermediate Texture** to **Always**.



13. Here is the complete **TintFeature code**, with the final result shown in the following code example:

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

public class TintFeature : ScriptableRendererFeature
{
        class CustomRenderPass : ScriptableRenderPass
        {
        private Material material;
        private RenderTargetIdentifier source;
        private RenderTargetHandle tempTexture;

        public CustomRenderPass(Material material) : base()
        {
                this.material = material;
                tempTexture.Init("_TempTintTexture");
        }

        public void SetSource(RenderTargetIdentifier source)
        {
                this.source = source;
        }

        public override void OnCameraSetup(CommandBuffer cmd, ref
RenderingData renderingData)
        {
                RenderTextureDescriptor cameraTextureDesc = renderingData.
cameraData.cameraTargetDescriptor;
                cameraTextureDesc.depthBufferBits = 0;
                cmd.GetTemporaryRT(tempTexture.id, cameraTextureDesc,
FilterMode.Bilinear);
        }

        public override void Execute(ScriptableRenderContext context, ref
RenderingData renderingData)
        {
                CommandBuffer cmd = CommandBufferPool.Get("TintFeature");

                Blit(cmd, source, tempTexture.Identifier(), material, 0);
                Blit(cmd, tempTexture.Identifier(), source);

                context.ExecuteCommandBuffer(cmd);
                CommandBufferPool.Release(cmd);
        }

        // Cleanup any allocated resources that were created during the
execution of this render pass.
        public override void OnCameraCleanup(CommandBuffer cmd)
```

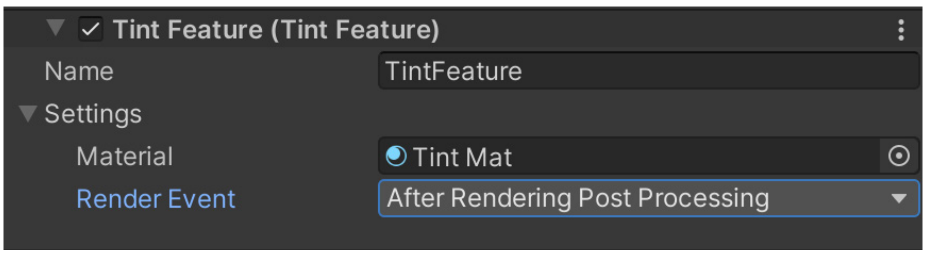
```
        {
                cmd.ReleaseTemporaryRT(tempTexture.id);
        }
        }

        CustomRenderPass m_ScriptablePass;

        /// <inheritdoc/>
        public override void Create()
        {
        var material = new Material(Shader.Find("Shader Graphs/Tint"));
        m_ScriptablePass = new CustomRenderPass(material);

        // Configures where the render pass should be injected.
        m_ScriptablePass.renderPassEvent = RenderPassEvent.
AfterRenderingOpaques;
        }

        // Here you can inject one or multiple render passes in the
renderer.
        // This method is called when setting up the renderer once per-
camera.
        public override void AddRenderPasses(ScriptableRenderer renderer,
ref RenderingData renderingData)
        {
        m_ScriptablePass.SetSource(renderer.cameraColorTarget);
        renderer.EnqueuePass(m_ScriptablePass);
        }
}
```

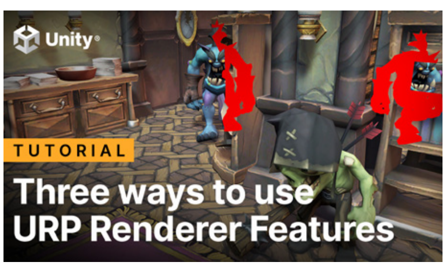
Effect of TintFeature: unprocessed to the left, tinted on the right

A more flexible option for assigning different materials and controlling where in the render pipeline to add the event is to use a Settings class, as in the following code example. You then use `settings.material` and `settings.renderEvent` in the Create method.

```
[System.Serializable]
public class Settings
{
public Material material;
public RenderPassEvent renderEvent =
                RenderPassEvent.AfterRenderingOpaques;
}

[SerializeField]
private Settings settings = new Settings();
```

Use a Settings class to assign the properties in the Inspector.





In this video tutorial, we show you three practical exercises using Renderer Features – namely, how to create a custom post-processing effect, stencil effect, and characters occluded by their environment.

You can find more community-driven examples of Renderer Feature best practices, including this video tutorial on how to control a custom Renderer Feature by Ned Makes Games.

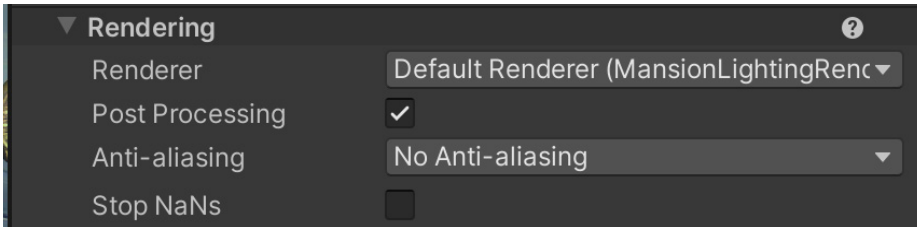
POST-PROCESSING

Applying post-processing effects: The top-left image has no effects applied, the top-right image has Bloom applied, the bottom-left has Vignette applied, and the bottom-right has Color Adjustment added.

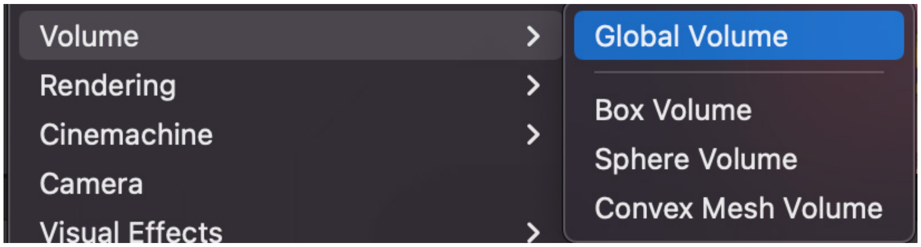
The Built-in Post-Processing Stack v2 package is not compatible with URP. URP does not require an additional package for post-processing effects. Instead, it uses a Volume framework. When you add Volumes to a scene, you can choose which post-processing effects apply to the Volume. A Volume can be Global or Local. If Global, the Volume affects the Camera everywhere in the scene. With the Mode set to Local, Volumes affect the Camera if it's within the bounds of the Collider.

Using the URP post-processing framework

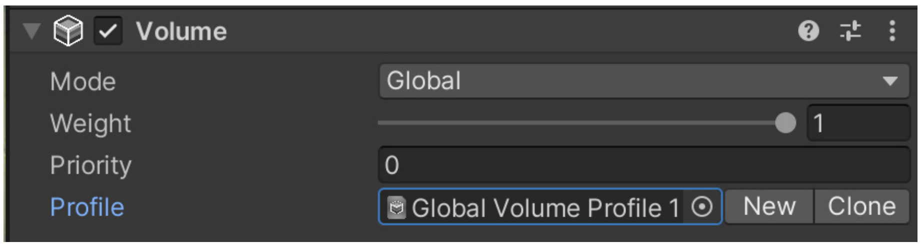
1. The first step is to make sure your Main Camera has post-processing enabled. Select the **Main Camera** in the **Hierarchy** window, go to the **Inspector**, and expand the **Rendering** panel. Check the **Post Processing** option.



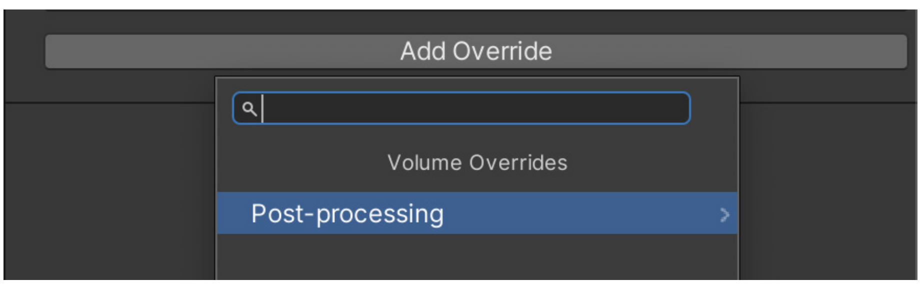
2. Right-click the **Hierarchy** window and select **Create > Volume > Global Volume** to create a Global Volume.

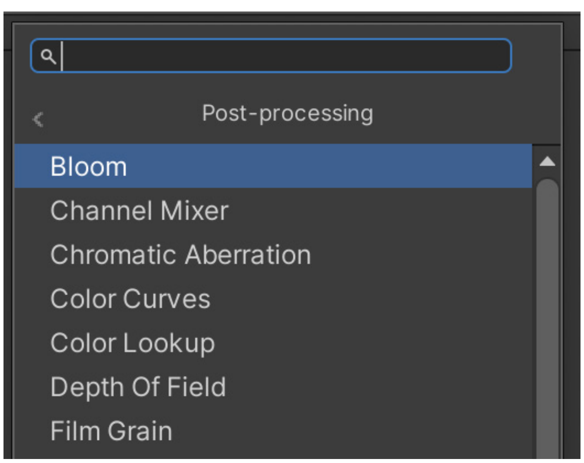


3. With the Global Volume selected in the Hierarchy window, find the **Volume** panel in the Inspector and create a new **Profile** by clicking on **New**.

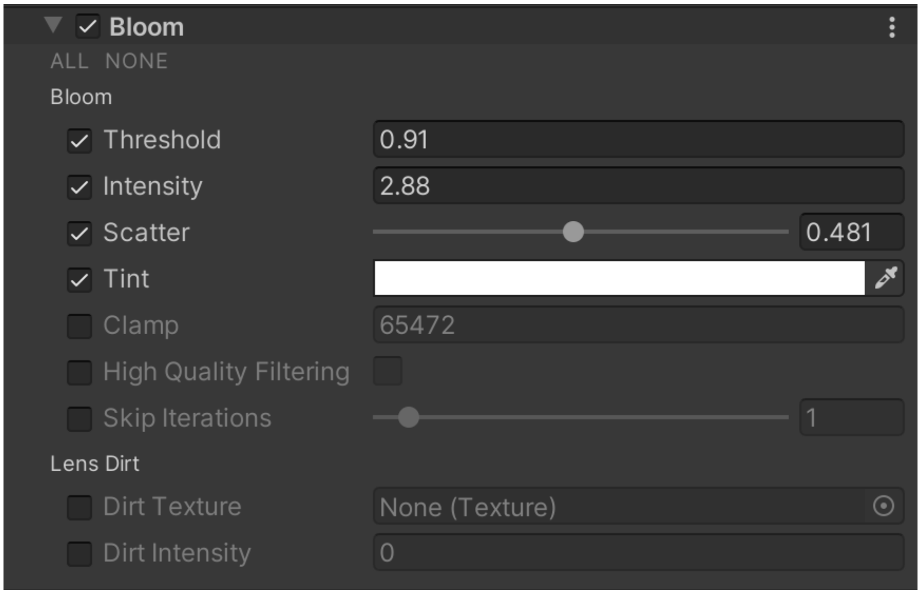


4. Start adding post-processing effects. See the table further down that lists available effects. Click **Add Override** and select **Post-processing**. In this example, the Bloom effect is chosen.

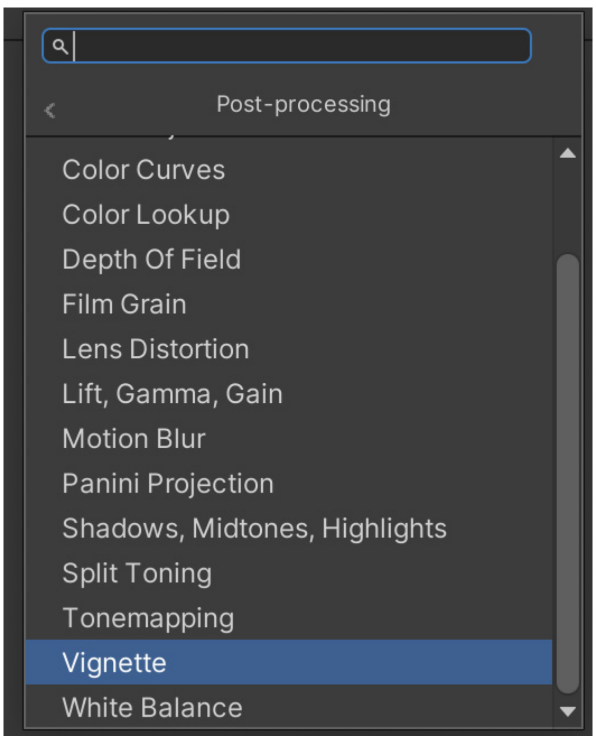




5. Each effect has a dedicated Settings panel. The image here shows the settings for Bloom.



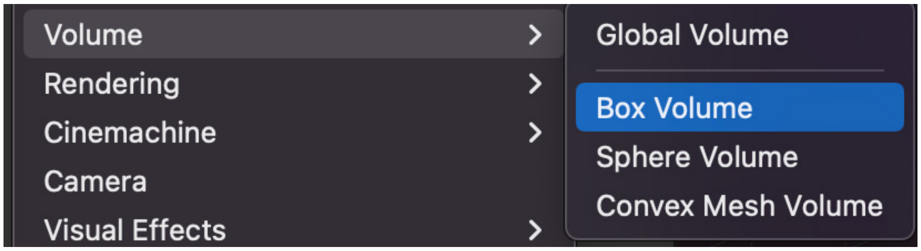
6. You can easily add multiple effects (such as Vignette in this example) and configure each one using their Settings panel.



Adding a Local Volume

With the Volume framework, you can configure the scene so that as a Camera moves around it, different post-processing profiles are triggered. This is achieved by adding a Local Volume. Let's go through the steps for setting this up.

1. In the **Hierarchy** window, right-click and choose **Create > Volume > Box Volume**. Alternatively, choose **Sphere Volume** if this shape is more suited to your purpose, or **Convex Mesh Volume** for a tighter control over the shape of the Collider that defines the Volume region.

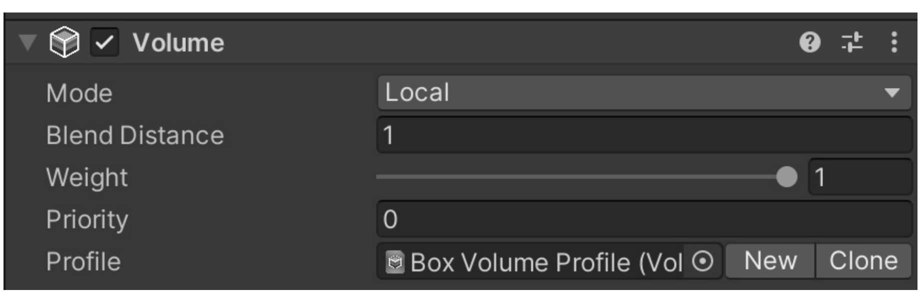


2. From the **Volume** panel in the **Inspector**, create a new **Profile** to store this Volume data. The panel can also be used to set:

   a. **Blend Distance:** This is the furthest distance from the Volume's Collider that URP starts blending from, and the distance in Collider dimensions where this profile fades in. At the edge of the Collider, the post-processing effects will fade out and the Blend Distance from the edge of the Collider will fully fade in.

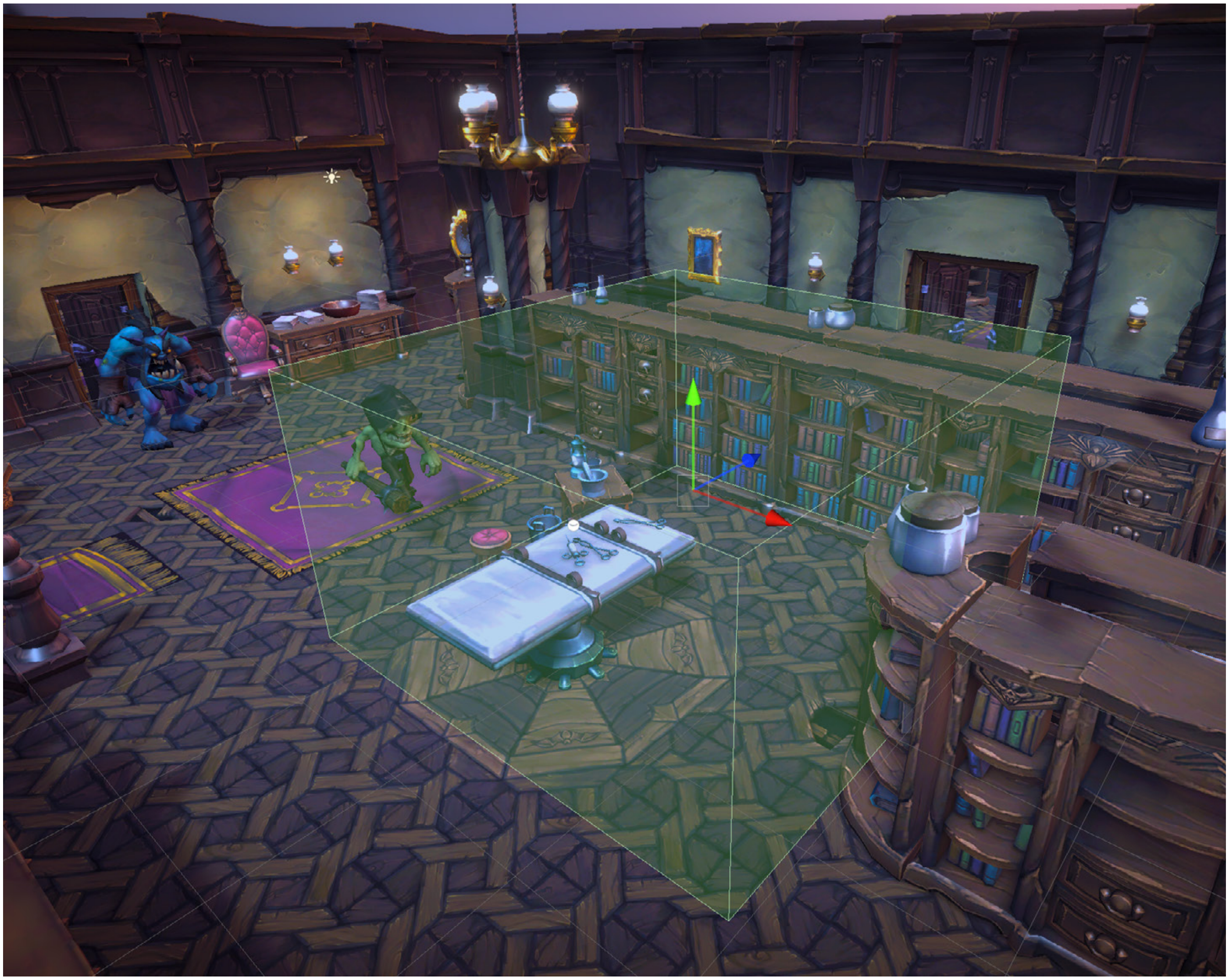
   b. **Weight:** Weight defines the maximum strength of the post-processing effects. If Weight is set to 1, then the effect will reach full strength. A setting of 0 means there is no effect, while 0.5 sets the strength of the effect at a maximum of 50%.

   c. **Priority:** Use this value to determine which Volume URP is used when multiple Volumes have an equal amount of influence on the scene. The higher the number, the higher the Priority. If you are merging Global and Local, then keep Global at the default 0 setting and set the Local Volume(s) to 1 or more.



Settings for a Local Volume

3. Position the Volume and control its dimensions using the **Box Collider** component, as shown in the image below.



Positioning and sizing a Box Volume using the attached Box Collider component

Post-processing can weigh heavily on your processor, so carefully consider the effects on low-end hardware and mobile devices. If your project must use it, then test on the target hardware. Some filters are less processor intensive than others. This document outlines the mobile-friendly effects.

These are the available post-processing effects in URP.

Effect	Description
Bloom	Adds a glow around pixels above a defined brightness level.
Channel Mixer	Modifies the influence of each input color channel on the overall mix.
Chromatic Aberration	Creates fringes of color along boundaries that separate dark and light parts of the image.
Color Adjustments	Use this effect to tweak the overall tone, brightness, and contrast of the final rendered image.
Color Curves	Grading curves are an advanced way to adjust specific ranges in hue, saturation, or luminosity.
Depth of Field	This effect simulates the focus properties of a camera lens.
Film Grain	This simulates the random optical texture of photographic film.
Lens Distortion	Distorts the final rendered picture to simulate the shape of a real-world camera lens.
Lift Gamma Gain	Use the different trackballs to affect different ranges within the image. Adjust the slider under the trackball to offset the color lightness of that range.
Motion Blur	This simulates the blur that occurs in an image when a real-world camera films objects moving faster than the camera's exposure time.
Panini Projection	This effect helps you render perspective views in scenes with a very large field of view.
Shadows Midtones Highlights	This effect separately controls the shadows, midtones, and highlights of the render.
Split Toning	Use this to add different color tones to the shadows and highlights in your scene.
Tonemapping	Tonemapping is the process of remapping the HDR values of an image to a new range of values.
Vignette	This effect comprises darkening toward the edges of an image compared to the center.
White Balance	Removes unrealistic color casts, so items that would appear white in real life render as white in your final image.

Controlling post-processing with code

You can also dynamically adjust your post-processing profile using a C# script. The following code example shows how to adjust the intensity of the Bloom effect. If a Vignette is applied, you can control the vignetting color via code. For example, if the player character takes damage, you can temporarily tint it red.

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

public class PPController : MonoBehaviour
{
        // Start is called before the first frame update
        void Start()
        {
        Volume volume = GetComponent<Volume>();
        Bloom bloom;
        if (volume.profile.TryGet<Bloom>(out bloom))
        {
                bloom.intensity.value = 0;
        }
    }
}
```

CAMERA STACKING

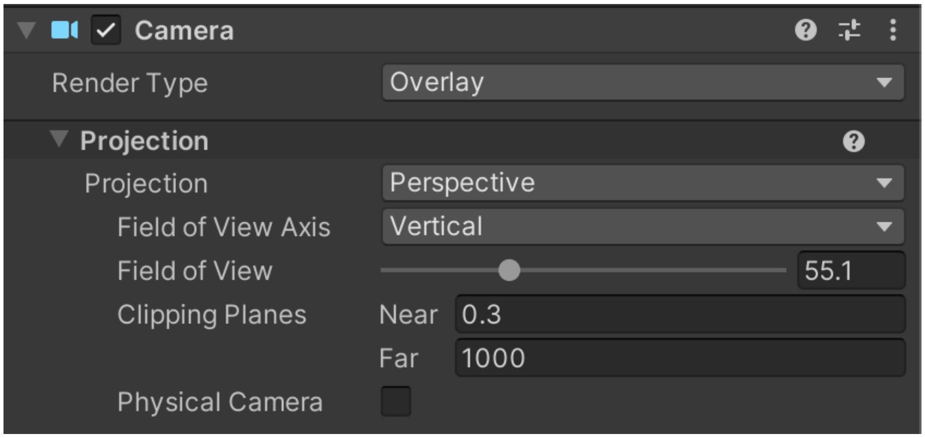
An example using Camera Stacking

A common requirement in games is the ability to combine geometry viewed from different cameras in a single render. The image above shows a shelf in the foreground acting as an inventory within the game. Collected items are added to the shelf and can be selected at key points by the player. Notice that it has a different field of view, as well as different lighting and post-processing.
This has been set up using the Camera Stacking feature in URP.

Let's look at how to set this feature up.

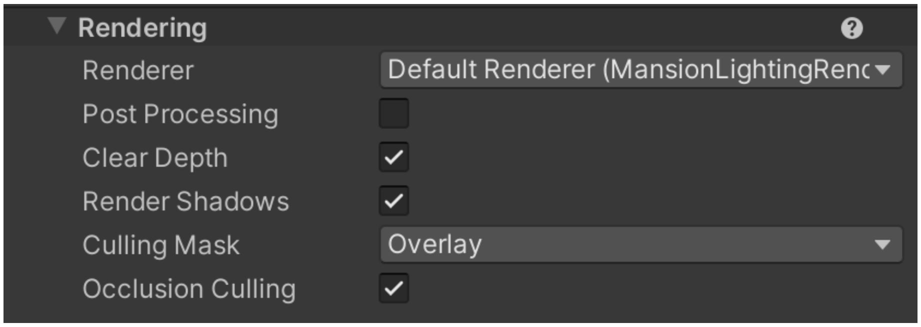
1. Create a Camera by right-clicking the **Hierarchy** view and choosing **Create > Camera**. Remove the audio listener component.

2. Use the **Inspector > Camera Settings** panel to set this Camera as **Render Type Overlay**.



3. Create a new **Layer** for the Camera and the GameObjects it renders.

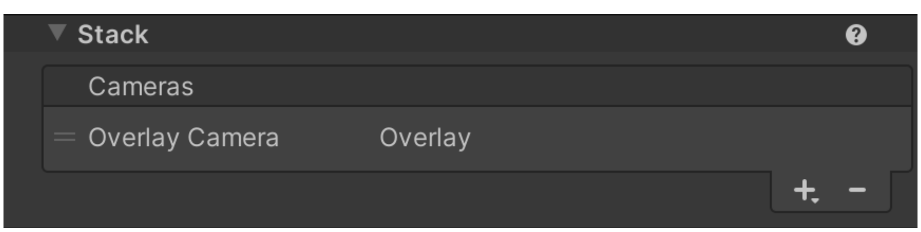
4. Update the **Rendering > Culling Mask** for the Camera using the Inspector.



5. Move the Camera to a suitable place in the scene, then add and position **GameObjects** by placing them in **Layer Overlay**.

6. Make sure the **Main Camera** does not render Overlay by updating its **Rendering > Culling Mask**.

7. In the **Stack** panel, use the "**+**" button to add the **Overlay Camera**.



Controlling a stack with code

As with post-processing, you can control the stack from code, and add or remove cameras dynamically during runtime. See this code example:

```
using UnityEngine;
using UnityEngine.Rendering.Universal;

public class StackController : MonoBehaviour
{
       public Camera overlayCamera;

       // Start is called before the first frame update
       void Start()
       {
       Camera camera = GetComponent<Camera>();
       var cameraData = camera.GetUniversalAdditionalCameraData();
       cameraData.cameraStack.Remove(overlayCamera);
       }
}
```

Post-processing and Camera Stacking, both easily configured using URP, are powerful tools for creating rich, atmospheric effects in your games.

ADDITIONAL TOOLS COMPATIBLE WITH URP

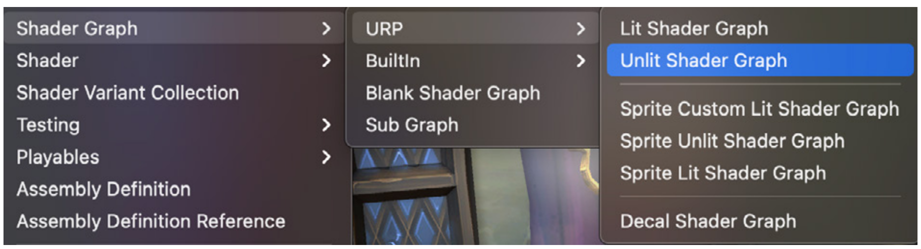
Another benefit of using URP is its compatibility with Unity's latest authoring tools that bring complex creation tasks into the reach of technical artists. This chapter unpacks how to create shaders using Shader Graph, and how to create particle effects using the Visual Effects (VFX) Graph.

Shader Graph

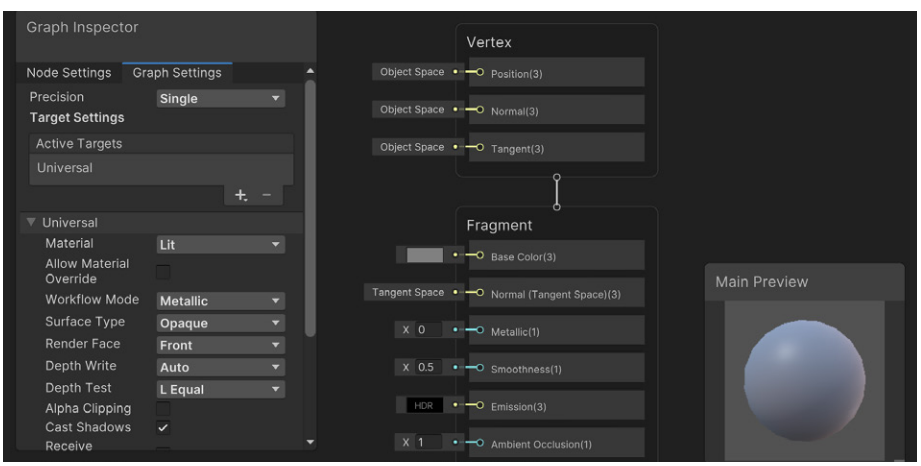
Shader Graph brings custom shaders to an artist's workflow. The Shader Graph tool is included when you start a project using the URP template or import the URP package.

Covering Shader Graph warrants a separate guide, but let's go over some basic yet crucial steps by creating the Light Halo shader from the Lighting chapter.

1. Right-click in the **Project** window, find a suitable folder, and choose **Create > Shader Graph > URP > Unlit Shader Graph**. For this example, choose Unlit. Name the new asset FresnelAlpha.

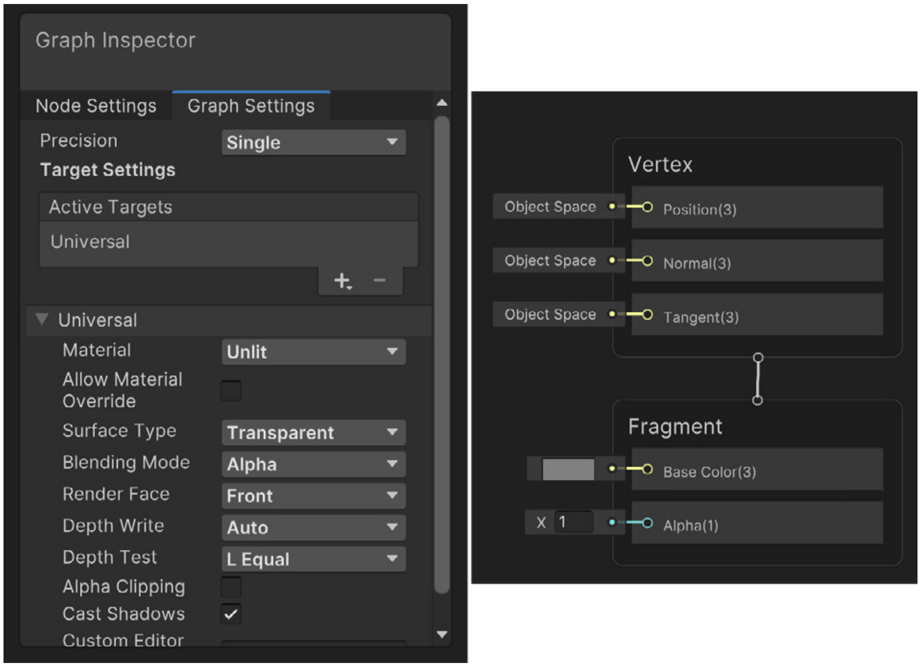


2. Double-click the new **Shader Graph Asset** to launch the Shader Graph editor.

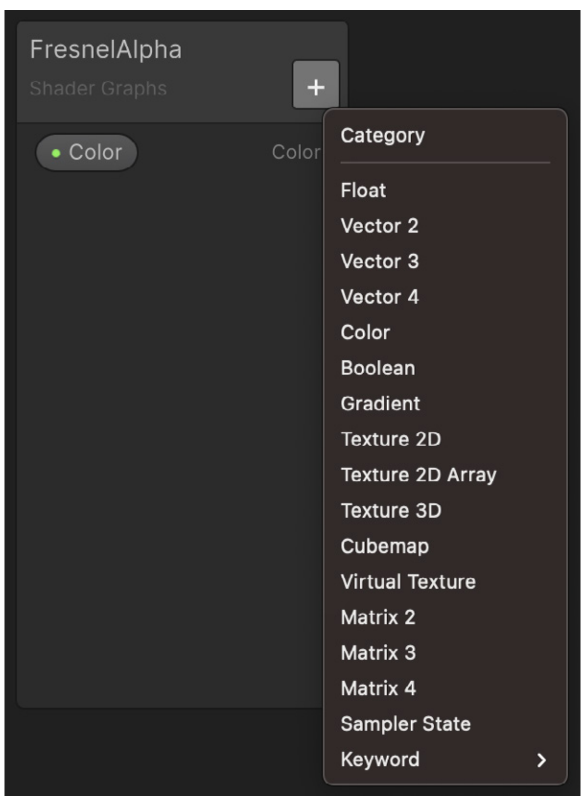


If you're familiar with shaders, then you'll recognize the Vertex and Fragment nodes. By default, this shader will ensure any model with a material using it that it is correctly placed in the Camera view using the Vertex node, and that each pixel is set to a grey color using the Fragment node.

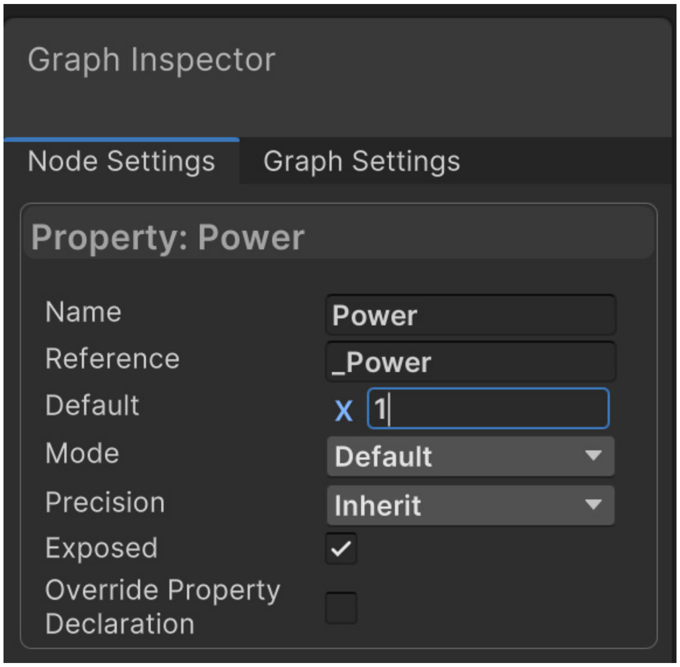
3. This shader is going to set the alpha transparency of the object. It therefore needs to apply to the Transparent queue. Change the **Graph Inspector > Graph Settings > Surface Type** to **Transparent**. You'll see that the Fragment node now has an Alpha input as well as Base Color.



4. Add properties to the shader. For instance, add Color as a Color, and Power and Strength as Float values.



5. Set the default values using **Graph Inspector > Node Settings > Default**. Set Color to white, Power to 4, and Strength to 1.

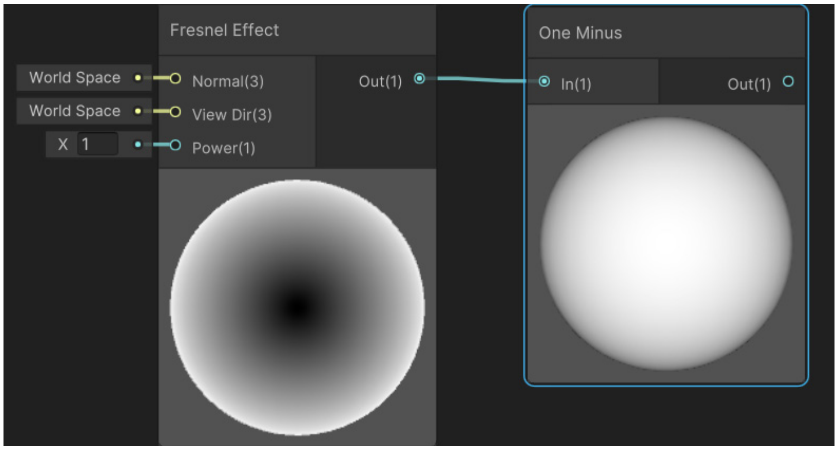


6. Shader Graph functions by joining nodes together. A node will have one or more inputs and an output. To add a node, right-click and choose **Create Node** in the **Search** panel at the top, then enter **Fre**. The results will show a **Fresnel Effect** node.

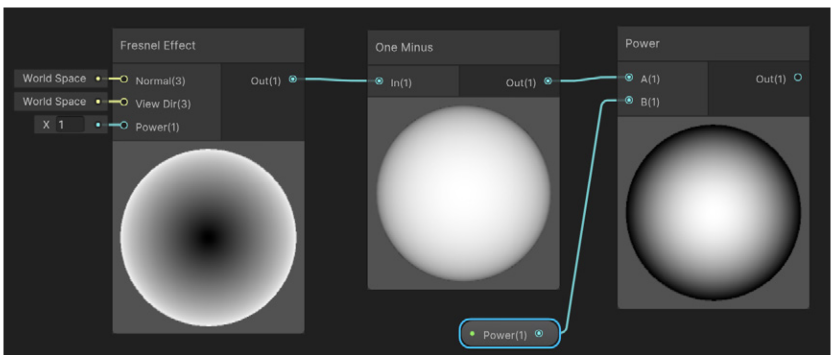
7. A node shows a preview of its effect. Notice that the Fresnel Effect is bright toward the edge. The value is the difference between the View direction and the Normal direction – and for a sphere, this is greatest at the edge.

   The alpha value should be lowest at the edge. You can flip the result using a One Minus node. To do this, click **Create Node** and enter **One**. Select the **One Minus** node. Now drag from Out(1) on the Fresnel Effect node to In(1) on the One Minus node. The 1 means that the value type is a single float. If it was 3, then it would be a vector with three components.

The nodes should be joined like this:

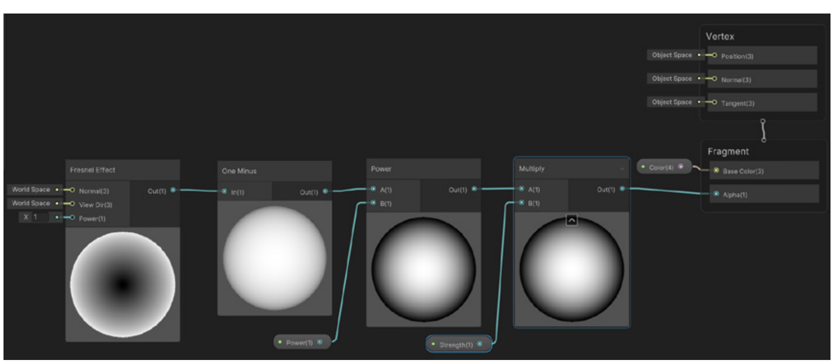


8. Let's look at how to control the size of gradient and the overall transparency. Use a **Power** node for sizing the gradient. Create a Power node and connect One Minus Out(1) to Power A(1). Drag the Power property to the graph and join it to Power B(1). The graph should now look like this:

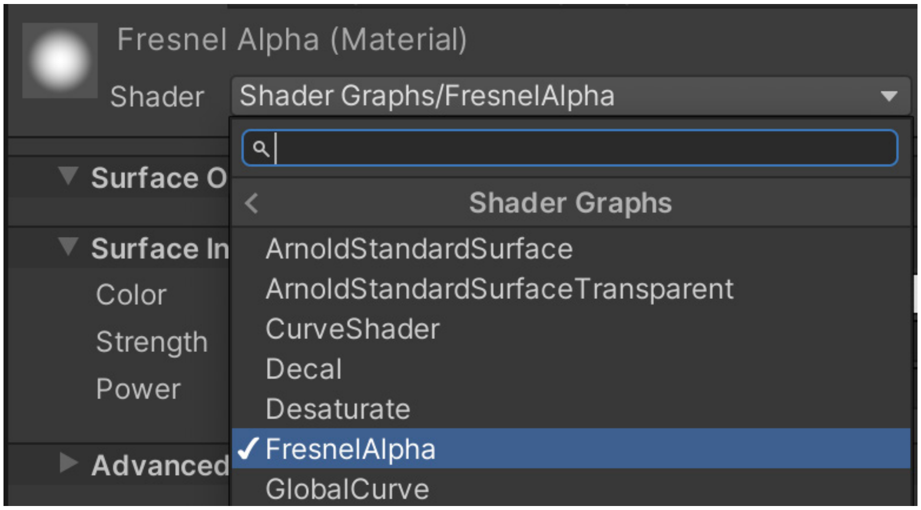


9. Control the overall transparency using a **Multiply** node. Create it and connect Power Out(1) to Multiply A(1). Drag the Strength property to the graph and join it to Multiply B(1). Then join the Multiply Out(1) to Fragment Alpha(1) and drag the Color(4) property to the graph and join it to Fragment Base Color(3).

   Notice here that the property Color comprises a four-component vector, while Base Color is a three-component vector. Shader Graph will map the first three components of Color to the Base Color vector.



10. Save the asset and create a new material. Assign this shader to the new material, which is located in **Shader Graphs/FresnelAlpha**.



11. Now you can apply the material to an object, controlling its visibility at the edges.

Shader applied to a sphere parented to a Point light, to give the halo effect around the hanging light

Related links:

— This blog post goes through the Shader Graph process with an example project and some advanced suggestions.

— Check out Shader Graph on the Unity website.

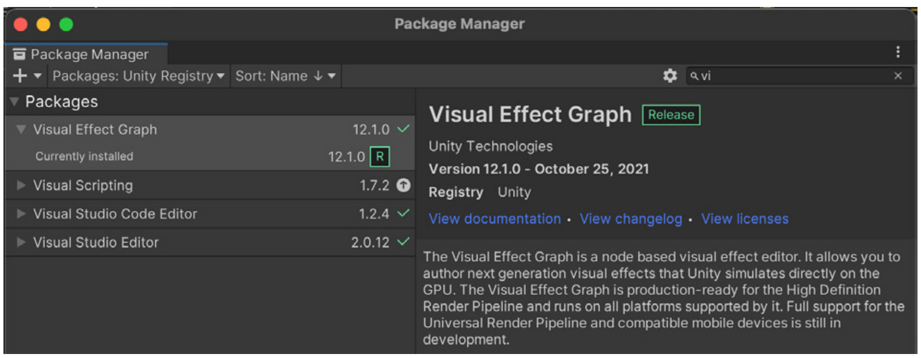
VFX Graph

The Visual Effect (VFX) Graph enables you to create myriad particle effects with an artist-friendly, node-based graph. Use a VFX Graph to add fire, smoke, mist, sparks, magic orbs, and many other effects to your project.

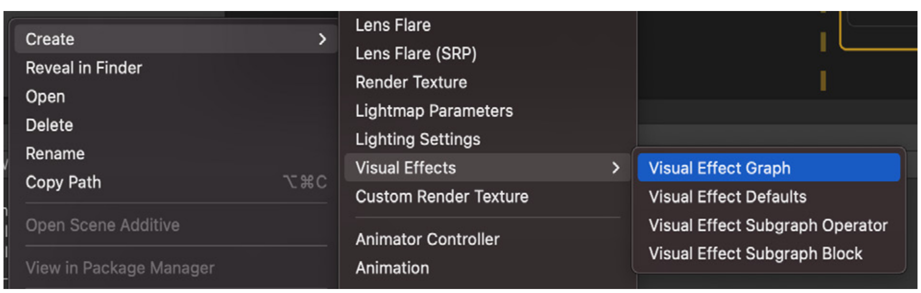
The target devices for any games containing effects created with VFX Graph must be compute-capable because VFX Graph uses compute shaders running on the GPU to ensure the best possible performance. Test your code and include a non-compute fallback, and use VFX Graph sparingly for games targeting low-end mobile devices.

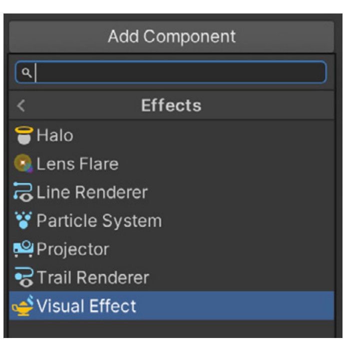
To get better acquainted with VFX Graph, let's go through the steps for creating a smoke effect:

1. VFX Graph can be downloaded as a package using **Package Manager**.



2. Once VFX Graph is installed, there will be a new option when you right-click in the **Project window > Assets** folder. Choose **Create > Visual Effects > Visual Effect Graph**, and name the new asset Smoke.

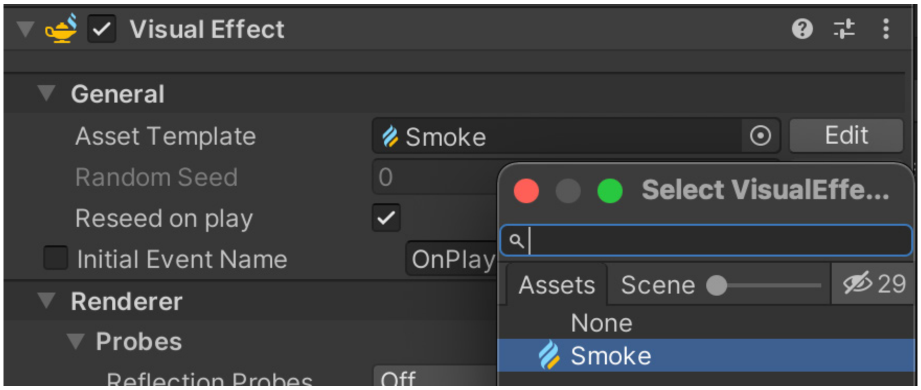




3. Create an empty **GameObject** and select it in the Hierarchy window. In the **Inspector**, choose **Add Component > Effects > Visual Effect**.

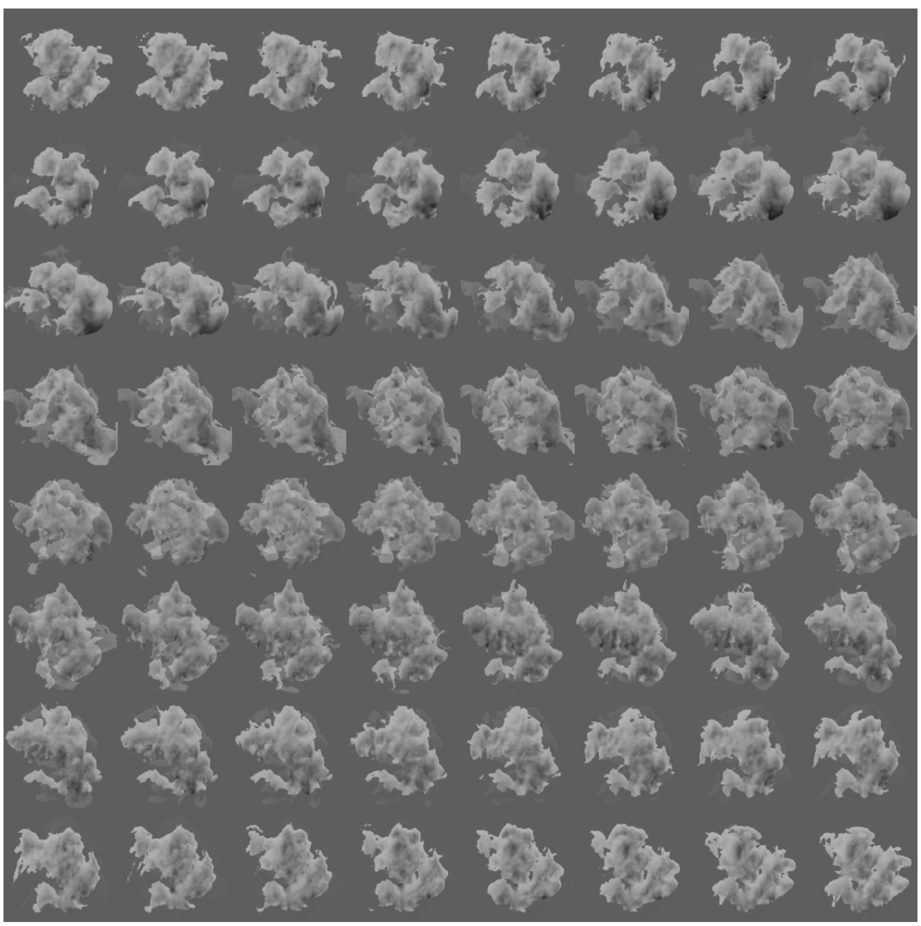
Alternatively, you can add the Visual Effect Graph Asset to the **Hierarchy** view in-Editor. This will add the component with the asset, allowing you to skip steps 3 and 4.

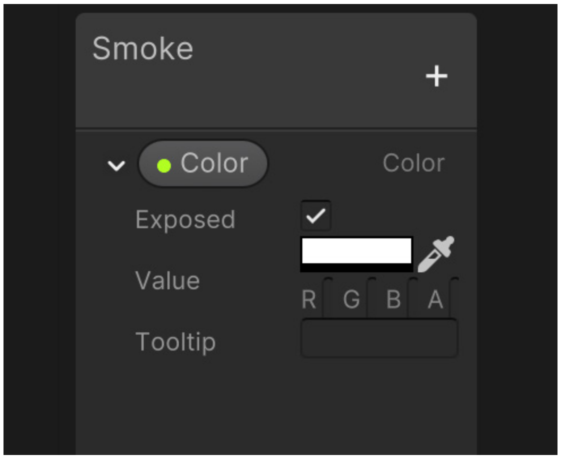
4. Select the **Smoke VFX Graph** as the **Asset Template** using the **Component Settings** panel.



5. Now you can edit the VFX Graph. Double-click to launch the **Visual Effect Graph** window. There you'll find Spawn, Initialize, Update, and Output Context nodes already prepopulated.

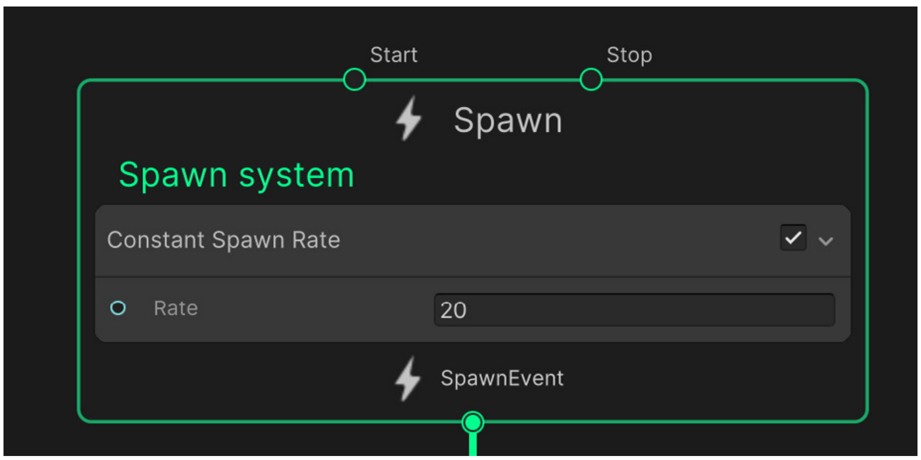
You'll use a Texture in the form of an Atlas that contains an animated smoke sprite. A series of 64 images in an 8×8 grid will act as the source for an individual particle. At any single frame, a single particle will display just one image from the grid. It will cycle through the images at a predefined rate as each frame is rendered. Here is the Smoke Sprite Atlas:



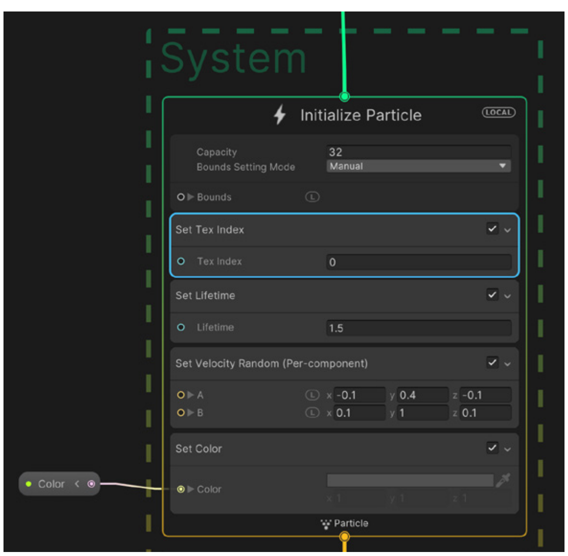


6. Click the “**+**” button and add a **Color** property. This will allow the user to manipulate the color of the smoke in the Inspector.

7. Let’s look at the Spawn block. The default Spawn block comes with a **Constant Spawn Rate** node. Set this to 20.



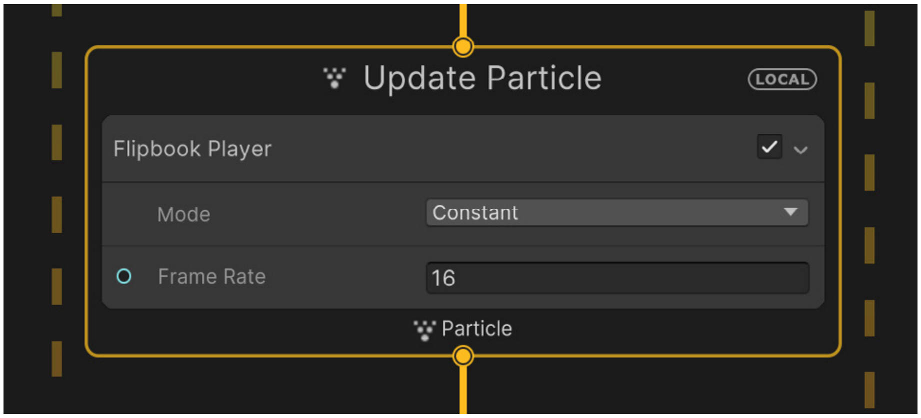
8. The next block, Initialize, defines how to handle a particle when it’s first created. Remove the **Set Lifetime Random** node. Then add a **Set Tex Index**, and set it to a random value from 0 to 64, so that each smoke particle has a different look. This is important because the particle displays an image from the Smoke Sprite sheet shown earlier and you’ll want the first index used to be 0.



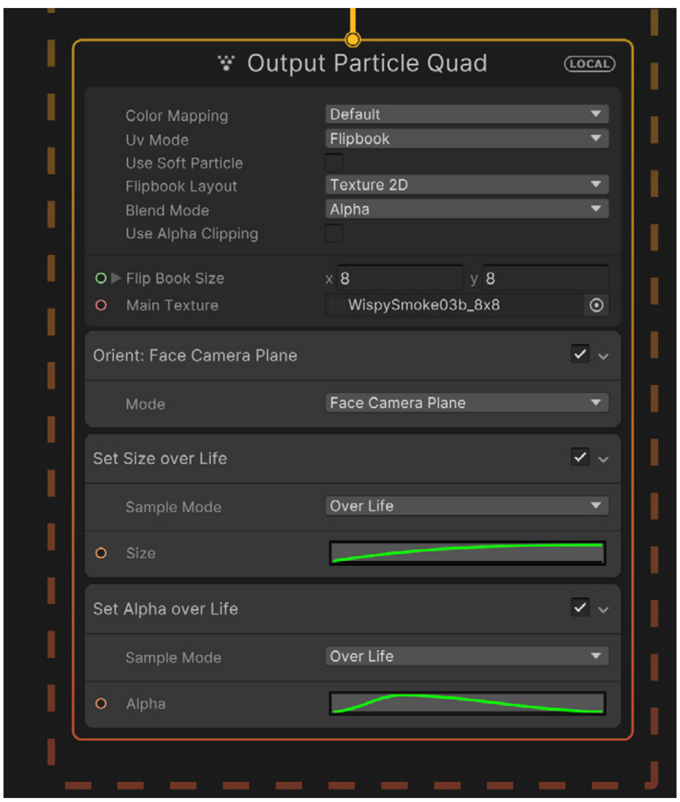
Then add a **Set Lifetime** node set to 1.5 seconds. To add some variation in the speed at which a particle is launched, use the **Set Velocity Random** node. Set A to -0.1, 0.4, -0.1 and B to 0.1, 1, 0.1. To set the Color of a particle to brighten or darken the Sprite, add a **Color** node and drag the Color property created to its input.

9. The next block, Update, defines what happens at each frame update. By default, this appears as an empty block, but it actually contains some implicit hidden blocks that can be disabled in the Inspector when Update is selected.

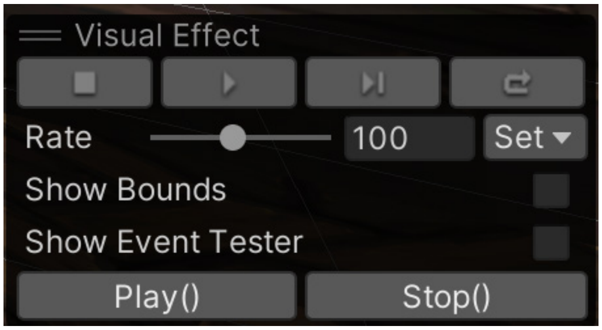
   Recall that you're using a Sprite sheet for the image of each particle. In VFX Graph, this means you're using a Flipbook. Add a **Flipbook Player** node, set its **Mode** to **Constant**, and the **Frame Rate** to 16. It will cycle through consecutive frames in the Flipbook at 16 frame changes per second.



10. Next, set the final output of the Particle. Set the **UV Mode** to **Flipbook** (or **Flipbook Blend** for a smoother transition between frames) and the **Flipbook Layout** to **Texture 2D**. Using the Sprite sheet, set the **Flipbook Size** to 8×8, and set the **Main Texture** to this **Texture**. Replace **Set Color Over Life** with **Set Alpha Over Life**. The default curve will blend the particle in and out over its lifetime.



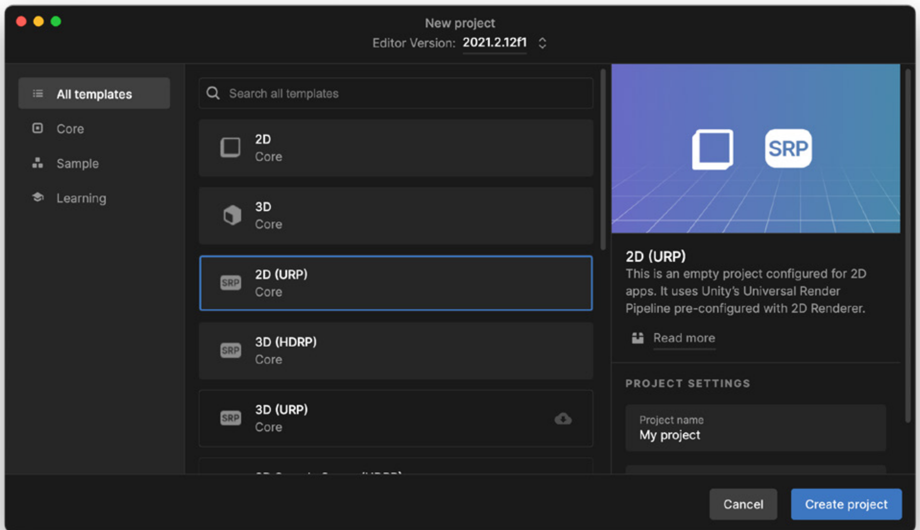
11. Select the **GameObject** with this VFX Graph attached. In the Scene view, a panel should be visible that you can use to demo the effect outside of runtime. If you don't see it, make sure the toggle for visualizing **Particle Systems** is on.



Here's an image of the final smoke effect in action:

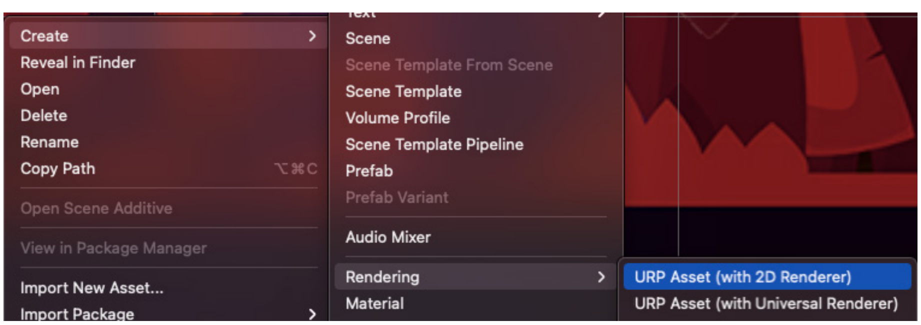
2D Renderer and 2D lights

If you are working on a 2D game, you'll be pleased to know there is a dedicated URP 2D Renderer. The simplest way to get started is to use the 2D URP template from the Unity Hub. This template ensures that your project has a **URP 2D Renderer** assigned via **Project Settings > Graphics > Scriptable Render Pipeline Settings**. All verified and precompiled 2D packages are installed with the 2D URP template and the default settings optimized for 2D projects. This also ensures that the project loads faster than installing all the packages manually.



The 2D URP template in the Unity Hub

If you're upgrading an existing project, then you need to find a suitable folder in your project's Assets folder. Right-click and select **Create > Rendering > URP Asset (with 2D Renderer)**. Give it a name, and select it using **Project Settings > Graphics > Scriptable Render Pipeline Settings**. In the Scene view, be sure to select the **2D** button when editing.



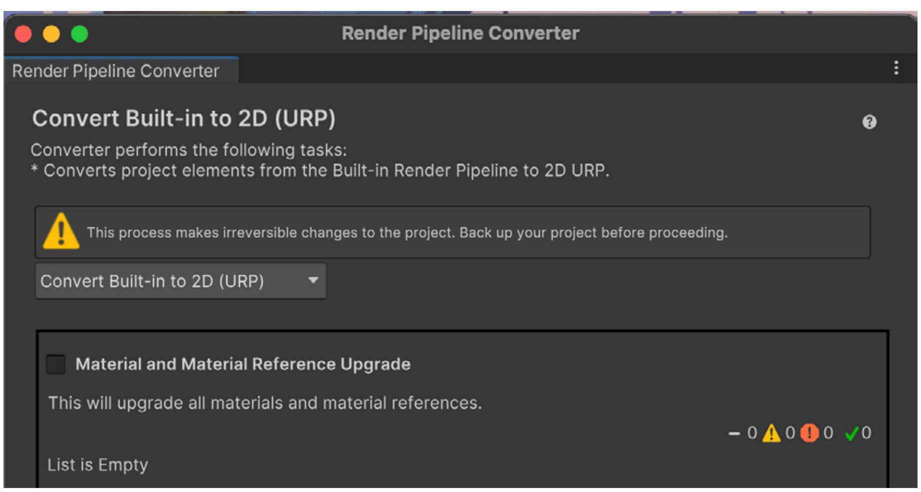
Creating a 2D Renderer and Settings Asset

If you're updating an existing project, then you might find switching to the URP 2D Renderer gives a classic magenta render error.

Updating an existing project with URP 2D Renderer can result in rendering errors in your scene.

Fortunately, the **Window > Rendering > Render Pipeline Converter** has got you covered. Select **Convert Built-in to 2D (URP)** and click the **Material** and **Material Reference Upgrade** panel. Then click **Initialize Converters**, followed by **Convert Assets**. If you still have magenta sprites, you might need to manually replace the shader in some of your materials. Choose one of the shaders in the following table.

2D shaders available in URP	
Shader	**Description**
Sprite-Lit-Default	Uses 2D lights when rendering
Sprite-Mask-Default	Works with the stencil buffer
Sprite-Unlit-Default	Uses only the texture colors when rendering

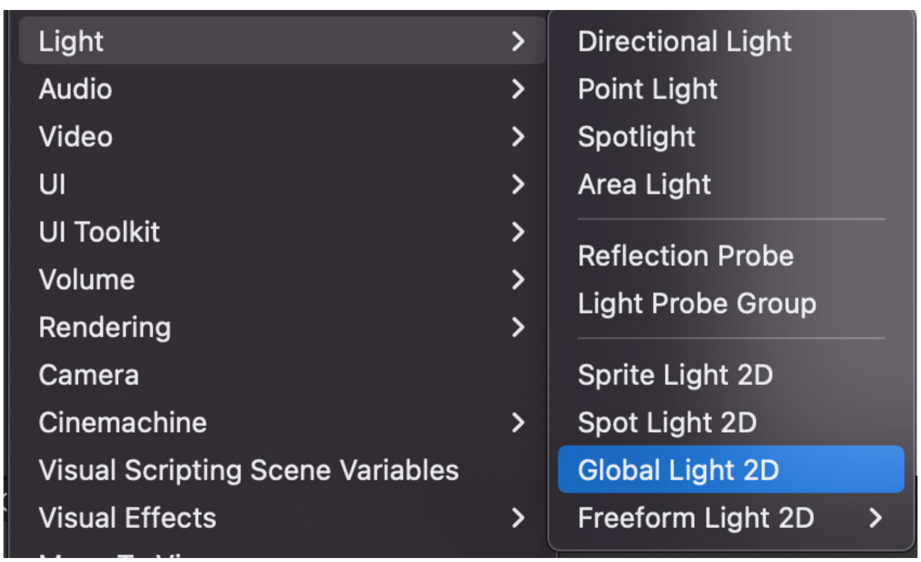


Converting a Built-in Render Pipeline 2D project to URP 2D

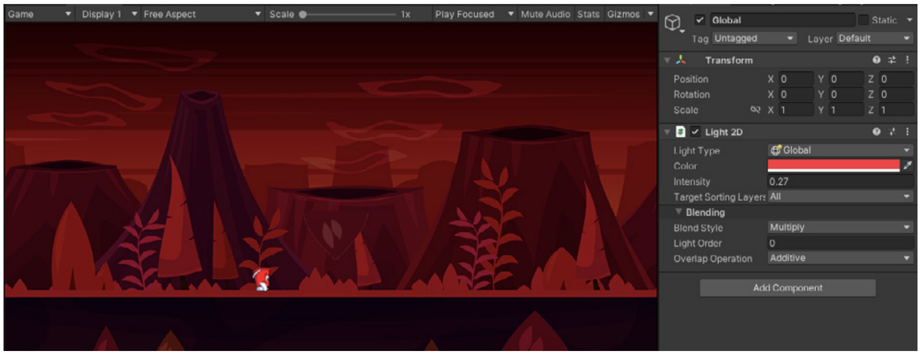
2D lights are available with the URP 2D Renderer. These offer enhanced performance and flexibility. Using the new tools, you can create a more immersive experience and save time preparing different Sprite variations by using baked lights to create new gameplay possibilities. If you have migrated an existing project, then you will have no URP 2D lights in your scene. If your Sprites use the Sprite-Lit-Default shader, you might be surprised to see a lit render. But with no lights, you get a default Global Light assigned to the scene for an unlit appearance.

With no lights in the scene, the render defaults to Unlit.

Add a light using the **Hierarchy** window. Right-click and choose **Create > Light > Global Light 2D**.



Now you can adjust the **Settings**, **Color**, **Intensity**, as well as the **Sorting Layers** they affect.



In the Global Light 2D Settings, the character uses an Unlit shader.

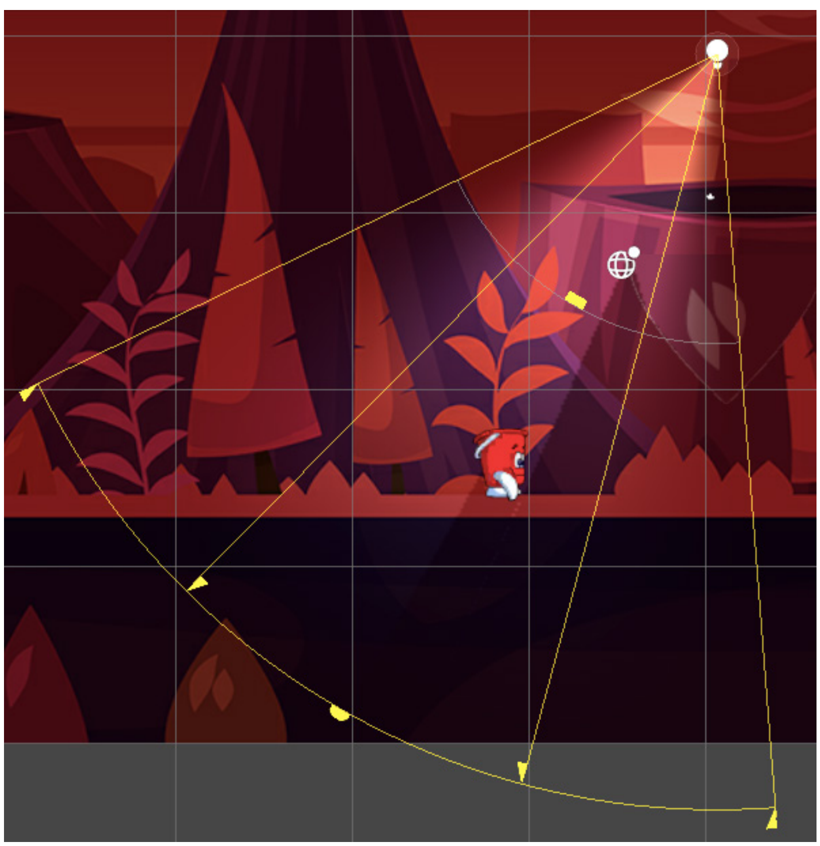
The 2D URP framework includes four light types:

— **Sprite:** Uses a Sprite to control the illumination level.

— **Freeform:** For creating a polygonal-shaped light.

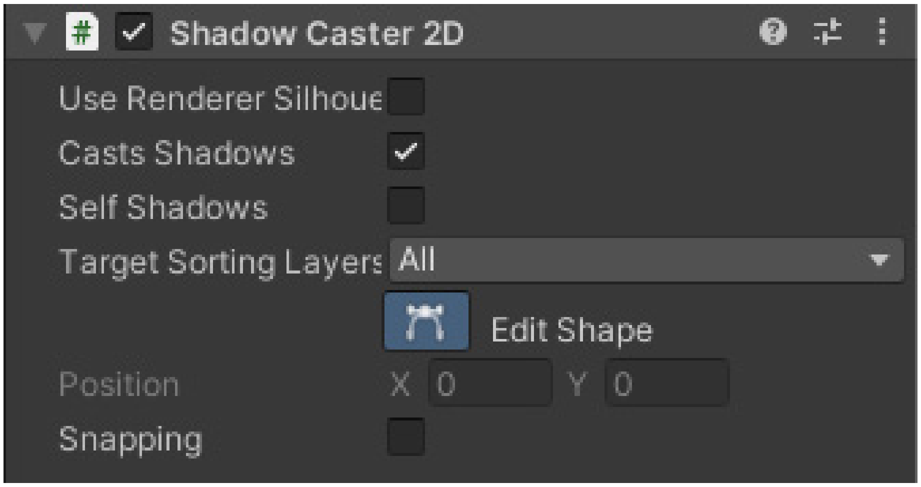
— **Spot:** Provides great control over the angle and direction of the selected light. Use it as a Point light. By default, the inner and outer cones span 360 degrees. You can also adjust the inner and outer radius and decide whether the light casts shadows, as well as the strength of those shadows.

— **Global:** Lights all objects on targeted sorting layers.



Editing a Spot light 2D

If a Sprite casts a shadow, then it needs a Shadow Caster 2D component added.



Adding a Shadow Caster 2D component

The URP 2D Renderer provides all the tools necessary to create first-class 2D games that will perform well on even low-end hardware.

An image from the Unity 2D demo *Dragon Crashers*; Unity's 2D development e-book, *2D game art, animation, and lighting for artists*, was authored by the creative director of *Dragon Crashers*.

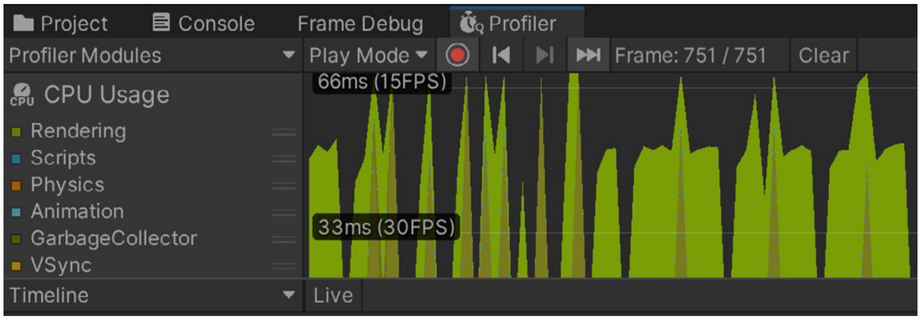
Related links:

— The Unity 2D demo *Dragon Crashers* is available on the Asset Store.

— The free e-book, *2D game art, animation, and lighting for artists*, is an advanced development guide created for Unity developers and artists planning to make a commercial 2D game.

PERFORMANCE

Performance is highly dependent on the project you're working on. Always profile and test your game throughout the development cycle. Open the Profiler via **Window > Analysis > Profiler**, and follow the suggestions in this chapter.



The Profiler window

This section looks at seven ways to improve the performance of your games:

— Managing your lighting
— Light Probes
— Reflection Probes
— Camera settings
— Pipeline settings
— Frame Debugger
— Profiler

These optimizations are also covered in this tutorial.

Optimizing lighting and rendering in URP

URP is built with optimized real-time lighting in mind. The URP Forward Renderer supports up to eight real-time lights per object and up to 256 real-time lights per camera for desktop games, plus 32 real-time lights per camera for mobile and other handheld platforms. URP also allows for configurable per-object Light settings inside the Pipeline Asset for refined control over lighting.

As explained in the Lighting chapter, baked lighting is one of the best ways to improve the performance of your scene. Real-time lighting can be expensive, whereas baking lights can help you gain back performance, assuming the lights in your scene are static. The baked lighting textures are batched into a single draw call, without needing to be continuously calculated. This is especially useful if your scene uses multiple lights. Another great reason to bake your lighting is that it allows you to render bounced or indirect lighting in your scene and improve the visual quality of the render.

Global Illumination is similarly covered in the Lighting section. This process simulates rays of light bouncing around the environment and illuminating other nearby objects with the bounced light. The figure below shows three lighting setups for the same scene: with no baked light data, with baked lighting, and with post-processing applied.

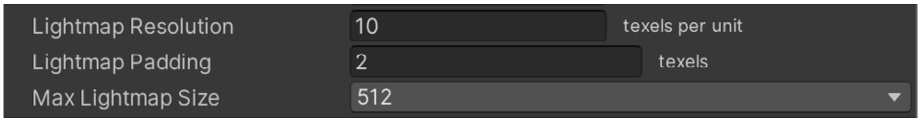
From left to right: no lighting data, baked lighting, post-processing added

When baked, areas of shadow in a scene receive the bounced light and are illuminated. It can be subtle, but this technique spreads the light around a scene more realistically and improves its overall appearance.

In the previous image, you can see that the specular highlights on the ground are lost when baking. Baked lights only contain diffuse lighting. Whenever possible, compute the direct lighting contribution from real-time, and have Global Illumination come from Image Based Lighting (IBL)/shadow maps/Probes.

The effect of light baking on shadows: before baking on the left, and after baking on the right

Use the lowest possible **Lightmap Resolution** and **Lightmap Size** when baking your lights; go to **Window > Rendering > Lighting > Scene**. This helps to lower the texture memory requirement.



Setting the Lightmap Resolution and Max Lightmap Size

Light Probes

As explained in the Lighting section, Light Probes sample the lighting data in the scene during baking and allow the bounced light information to be used by dynamic objects as they move or change. This helps them blend into and feel more natural in the baked lighting environment.

Light Probes add naturalism to a render without increasing the processing time for a rendered frame. This makes them suitable for all hardware, even low-end mobile devices.

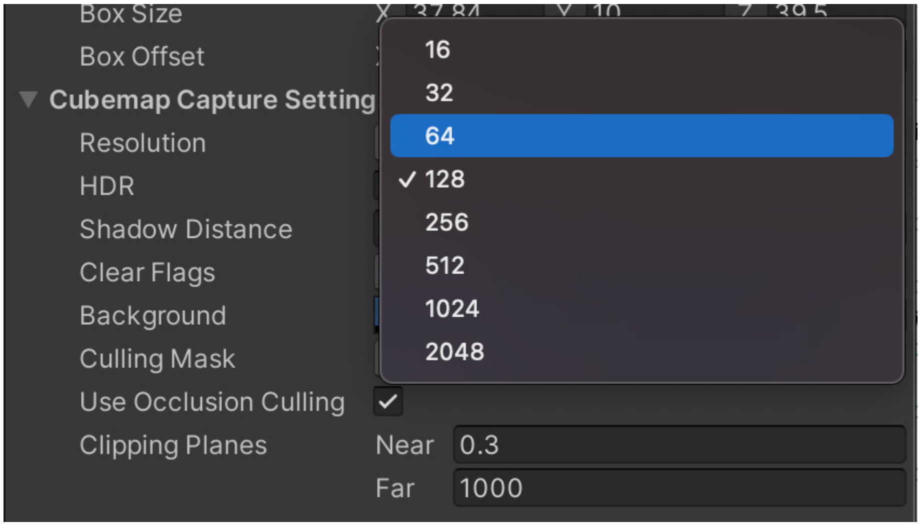
The effect of using Light Probes when rendering a dynamic object: with Light Probe on the left, and without on the right

Reflection Probes

You can also use Reflection Probes to optimize your scene. Reflection Probes project parts of the environment onto nearby geometry to create more realistic reflections. By default, Unity uses the Skybox as the reflection map. But by using one or more Reflection Probes, the reflections will match their surroundings more closely.

The effect of using Reflection Probes on smooth surfaces: with Reflection Probes on the left and without on the right

The size of the cubemap generated when baking the Reflection Probes depends on how close the Camera gets to a reflective object. Always make sure to use the smallest map size that suits your needs to optimize your scene.



Adjusting the size of the Reflection Probe cubemap

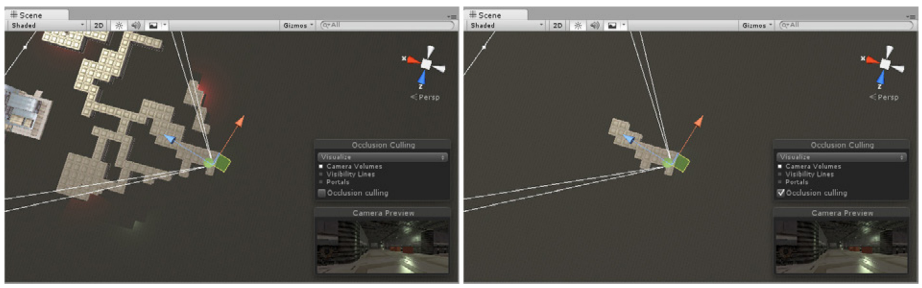
Camera settings

The URP enables you to disable unwanted renderer processes on your cameras for performance optimization. This is useful if you're targeting both high- and low-end devices in your project. Disabling expensive processes, such as post-processing, shadow rendering, or depth texture can reduce visual fidelity but improve performance on low-end devices.

Occlusion culling

Another great way to optimize your Camera is with occlusion culling. By default, the Camera in Unity will always draw everything in the Camera's frustum, including geometry that might be hidden behind walls or other objects. There's no point in drawing geometry that the player can't see, and that takes up precious milliseconds. This is where occlusion culling comes in.

Occlusion culling is best suited to a scene where significant numbers of objects might be masked when another item appears between them and the Camera. A cellular corridor maze-type game is an ideal candidate for using occlusion culling, as seen in the images below.

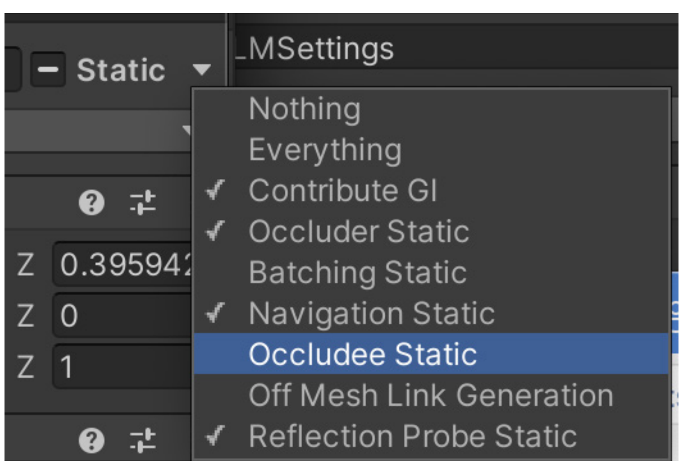


Frustum culling in the image on left, and occlusion culling in the image on right

By baking occlusion data, Unity ignores the parts of your scene that are blocked. Reducing the geometry being drawn per frame provides a significant performance boost.

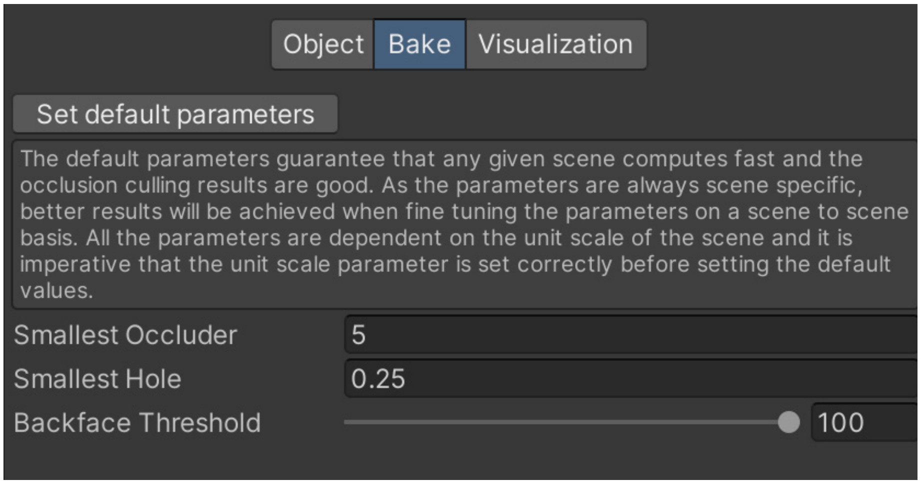
To enable occlusion culling in your scene, mark any geometry as either **Occluder Static** or **Occludee Static**. Occluders are medium to large objects that can occlude objects marked as Occludees. To be an Occulder, an object must be opaque, have a Terrain or Mesh Renderer component, and not move at runtime. Occludees can be any object with a Renderer component, including small and transparent objects that similarly do not move at runtime.

You set the static properties using the usual drop-down.



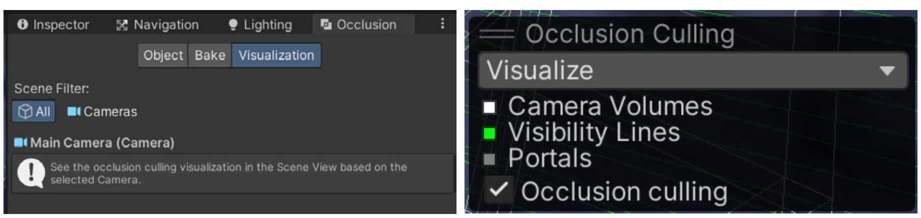
Settings for an object included in occlusion data

Open **Window > Rendering > Occlusion Culling**, and select the **Bake** tab. In the bottom-right corner of the **Inspector**, press **Bake.** Unity generates occlusion data, saving the data as an asset in your project and linking the asset to the current scene.



Occlusion culling Bake tab

You can see occlusion culling in action using the **Visualization** tab. Select the **Camera** in the scene and use the **Occlusion Culling** pop-up window in the **Scene** view to configure the visualization. The pop-up might be hidden behind the small Camera view window. Right-click the double-line icon and choose **Collapse** if this is the case. Move the pop-up, then restore the Camera view using right-click expand.



Visualization tab and Occlusion Culling pop-up

As you move the Camera, you should see objects popping on and off.

The effect of occlusion culling off in the left image, and on in the right image

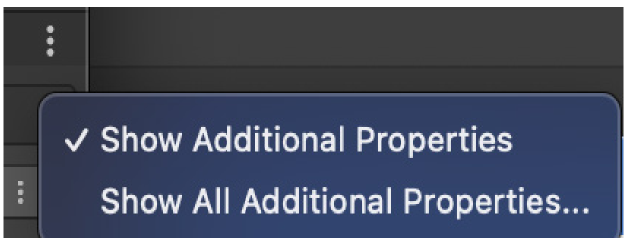
Pipeline settings

While the effects of changing the settings for the URP Asset and using different Quality tiers were previously covered, here are some additional tips for experimenting with Quality tiers to get the best results for your project:

— Reduce Shadow Resolution and distance for performance gains.

— Disable features that your project does not require, such as depth texture and opaque texture.

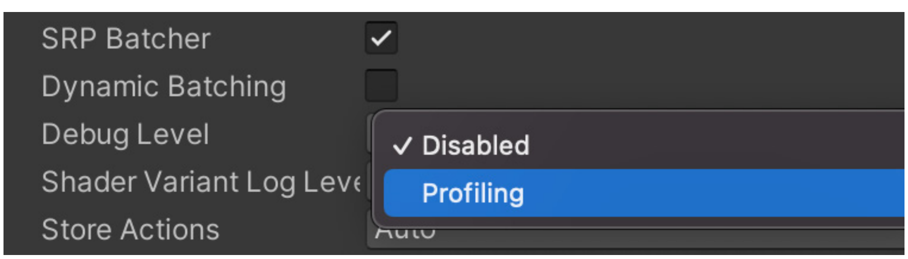
— Enable the SRP Batcher to use the new batching method. The SRP Batcher will automatically batch together meshes that use the same shader variant, thereby reducing draw calls. If you have numerous dynamic objects in your scene, this can be a useful way to gain performance. If the SRP Batcher checkbox is not visible, then click the three vertical dots icon (⋮) and select **Show Additional Properties**.



Enabling additional properties for the URP Asset Inspector

Frame Debugger

Use the Frame Debugger to gain a better understanding of what's happening during rendering. To view additional information in the Frame Debugger window, adjust the **Debug Level** using the **URP Asset**. As with the SRP Batcher checkbox, this is only visible in the Inspector with **Show Additional Properties** enabled.

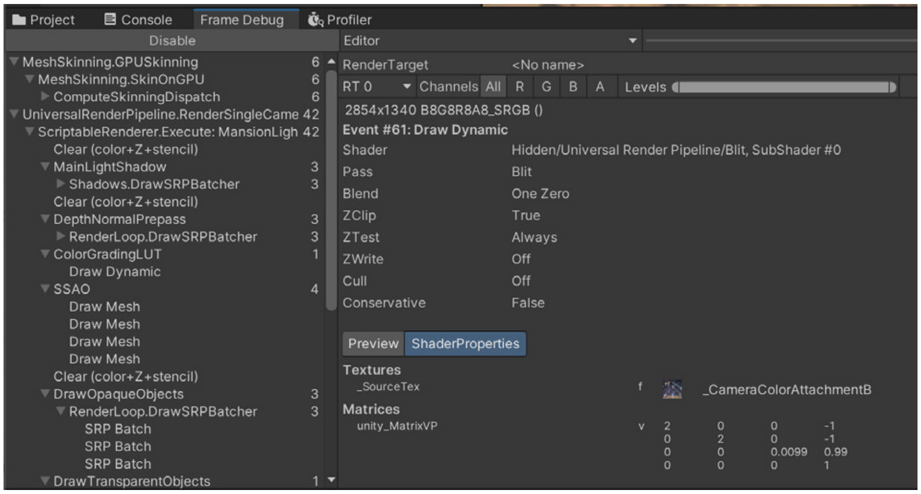


Setting the Debug Level

Adjusting the Debug Level can affect performance. Always turn it off when the Frame Debugger is not in use.

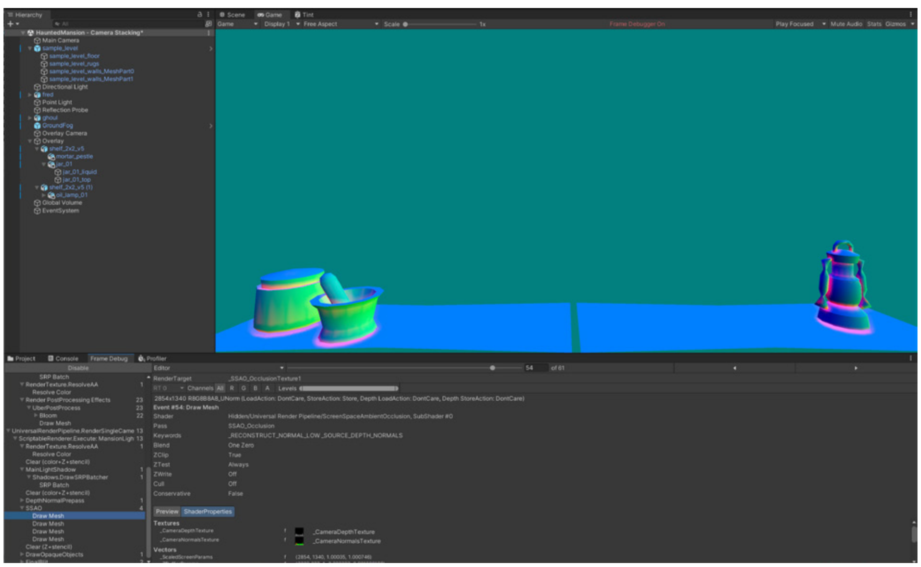
The Frame Debugger shows a list of all the draw calls made before rendering the final image and can help you pinpoint why certain frames are taking a long time to render. It can also identify why your scene's draw call count is so high.

Open the Frame Debugger by going to **Window > Analysis > Frame Debugger**. When your game is playing, select the **Enable** button. This will pause the game and let you examine the draw calls.



Frame Debugger detail

Clicking a stage in the render pipeline (left pane) will show a preview of this stage in **Game** view.



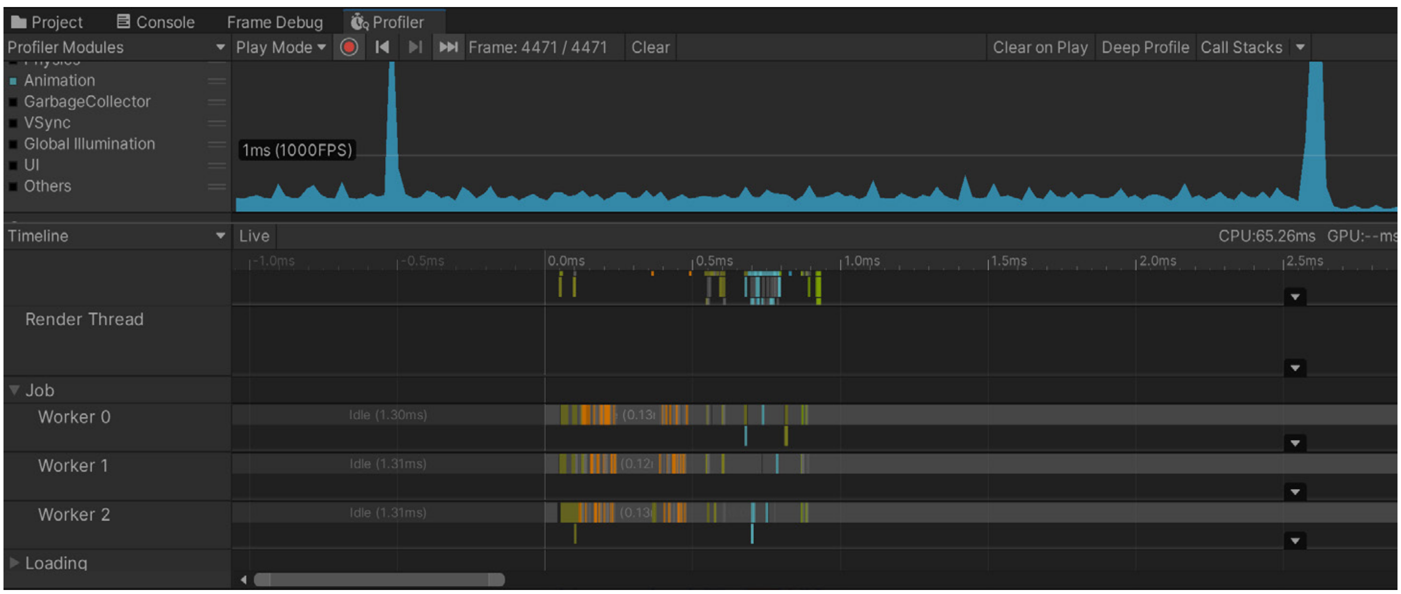
The Frame Debugger shows every step of the rendering process in the Game View – in this case, the SSAO generation step.

Unity Profiler

Like the Frame Debugger, the Profiler is a great way to determine how long it takes to complete a frame cycle in your project. It provides an overview of rendering, memory, and scripting. You can identify scripts that take a long time to complete, helping you to pinpoint potential bottlenecks in your code.

Open the Profiler via **Window > Analysis > Profiler**. When in **Play Mode**, the window provides an overview of the overall performance of your game. You can also pause the live view and use the **Hierarchy Mode** to get a breakdown of the time taken to complete a single frame. The Profiler will show you each call Unity has made during the frame.

For an even more detailed analysis, use the low-level native plug-in Profiler API. You can use this Profiler API to extend the Profiler, and profile the performance of native plug-in code, or to prepare profiling data to send to third-party profiling tools such as Razor for Sony Playstation, PIX for Microsoft (Windows and Xbox), as well as Chrome Tracing, ETW, ITT, VTune, or Telemetry.



The Profiler window using the low-level native plug-in Profiler API

Here's an example of using the low-level native plug-in Profiler API:

```
#include <IUnityInterface.h>
#include <IUnityProfiler.h>

static IUnityProfiler* s_UnityProfiler = NULL;
static const UnityProfilerMarkerDesc* s_MyPluginMarker = NULL;
static bool s_IsDevelopmentBuild = false;

static void MyPluginWorkMethod()
{
        if (s_IsDevelopmentBuild)
        s_UnityProfiler->BeginSample(s_MyPluginMarker);

        // Code I want to see in Unity Profiler as "MyPluginMethod".
        // ...

        if (s_IsDevelopmentBuild)
        s_UnityProfiler->EndSample(s_MyPluginMarker);
}

extern "C" void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPlugin-
Load(IUnityInterfaces* unityInterfaces)
{
        s_UnityProfiler = unityInterfaces->Get<IUnityProfiler>();
        if (s_UnityProfiler == NULL)
        return;
        s_IsDevelopmentBuild = s_UnityProfiler->IsAvailable() != 0;
        s_UnityProfiler->CreateMarker(&s_MyPluginMarker, "MyPluginMeth-
od", kUnityProfilerCategoryOther, kUnityProfilerMarkerFlagDefault, 0);
}

extern "C" void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPluginUn-
load()
{
        s_UnityProfiler  = NULL;
}
```

Additional resources

If you're interested in building advanced profiling skills in Unity, start by downloading the free e-book, *Ultimate guide to profiling Unity games*. This guide brings together advanced advice and knowledge on how to profile an application in Unity, manage its memory, and optimize its power consumption from start to finish.

A couple of other useful resources recommended by Nik include Measuring Performance by Catlike Coding, and Unity Draw Call Batching by The Gamedev Guru.

Conclusion

For developers and artists looking to switch to URP, be sure to check out the full **Unity Documentation**, as well as **Unity Learn**, the **Unity Blog**, and the **URP Forum**.

The Unity **Product Board** provides an overview of current URP features being developed, in addition to what's coming up next. You can even add your own feature requests.

To wrap up this e-book, here are just a few of the stunning and original games made with the rendering power and flexibility of Unity's URP.

Good luck with your game development.

Crash Bandicoot: On the Run! by King Digital Entertainment, for mobile

Lost in Random by Thunderful Games, published by Electronic Arts, for console and PC

Tales of Iron by Odd Bug Studio, CI Games, for console and PC

Circuit Superstars by Original Fire Games, published by Square Enix Collective, for console and PC

Neon White by Angel Matrix and Ben Esposito, published by Annapurna Interactive

Card Shark by Nerial, published by Devolver Digital, for console and PC

Spirit of the Island by 1M Bits Horde, published by META publishing, for Windows unity.com