UNITY FOR DEVELOPERS

E-BOOK

THE UNIVERSAL RENDER PIPELINE COOKBOOK:

RECIPES FOR SHADERS AND VISUAL EFFECTS

UNITY 2022 LTS EDITION

Unity®

Contents

INTRODUCTION

A dash of post-processing effects, a cup of decals, a pinch of color grading, and some sparkling water: It's time to cook up some high-quality lighting and visual effects in your games using the Universal Render Pipeline (URP).

In this cookbook, you can choose from 12 recipes for creating popular effects using URP. Additionally, sample scenes based on these recipes are available to download from this GitHub repo maintained by the guide's main author, Nik Lever.

This guide is aimed at intermediate to advanced Unity users. It assumes a foundational knowledge of developing a project in Unity, URP, and using HLSL to write shaders.

You'll get all the ingredients you need to:

— Create an x-ray-like image effect with stencils

— Build a toon and outline shader with Shader Graph

— Add an ambient occlusion effect with post-processing

— Use Photoshop and a LUT image to add color grading to your scenes

— Produce reflections and refraction, and much more.

It can be useful to reference this cookbook alongside the *Introduction to the Universal Render Pipeline for advanced Unity creators* guide. There is also a series of URP tutorials on Unity's YouTube channel, providing both general and more specialized tips for creating lighting and effects for your games.

We hope you have fun creating beautiful effects for your game.

This image and the cover image for the e-book are from *PRINCIPLES*, a sample of what URP can achieve in the hands of experienced developers. *PRINCIPLES* is an adventure game from COLOPL Creators, the technology brand of COLOPL Inc, who developed the series of *Shironeko Project* and *Quiz RPG: The World of Mystic Wiz*. Experience a deep underworld that makes use of Unity's latest features, including URP, for stunning graphics and immersive 3D sound. *PRINCIPLES* is available in the App Store and Google Play). You can also watch an interview with the studio here.

Author and contributors

Nik Lever, the main author of this e-book, has been creating real-time 3D content since the mid-90s 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.

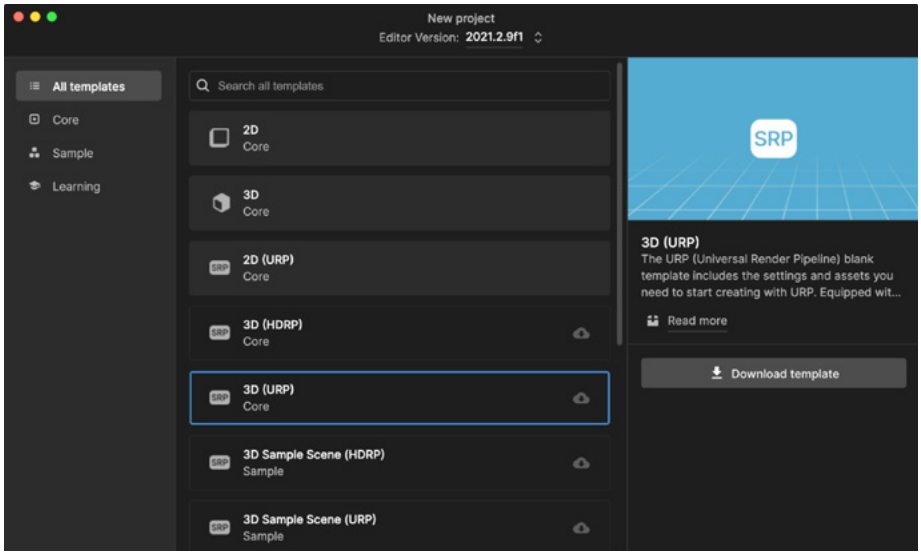
Steven Cannavan is a senior development consultant on the Accelerate Solutions Games team, specializing in the Scriptable Rendering Pipelines. He has over 15 years of experience in the game development industry.

GETTING STARTED WITH THIS GUIDE

You can follow the steps in each recipe to recreate the lighting and visual effects by opening a new URP project. Additionally, you can access the Github page that accompanies this guide, which provides you with downloadable sample scenes for each recipe.

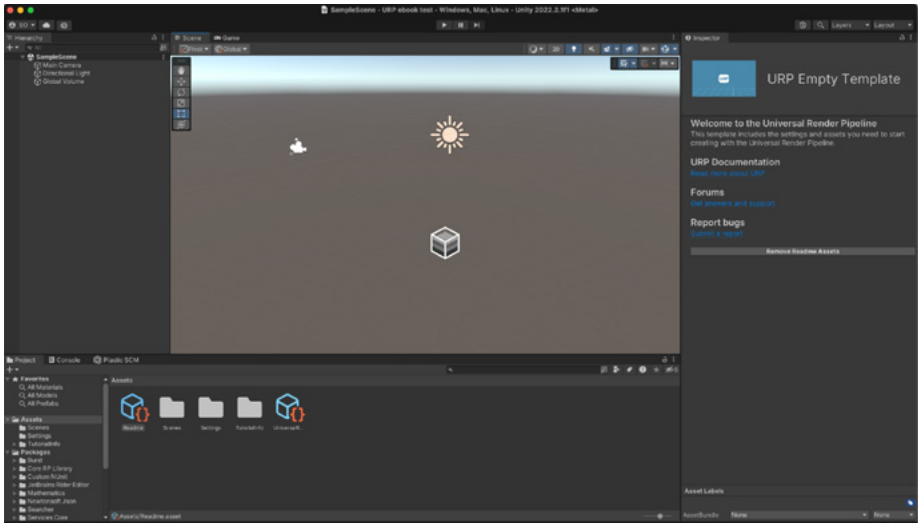
Starting a new URP project

Open a new project using URP via the Unity Hub. Click **New**, and verify that the Unity version selected at the top of the window is 2022.2 or newer. Choose a name and location for the project, select the **3D (URP)** template, and click **Create**.



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

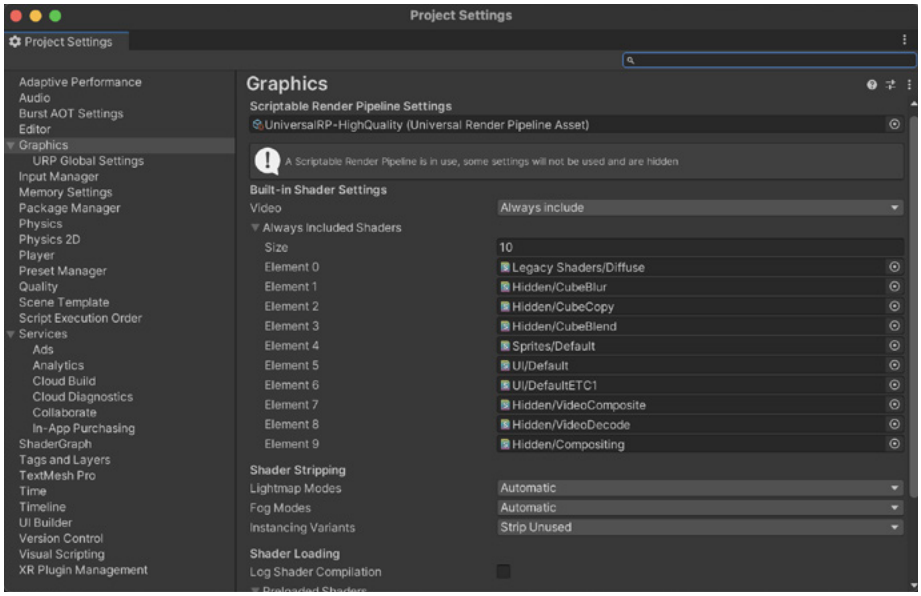
Note: The template ensures that your project is set to use a linear color space, which is required for calculating lighting correctly.



This template is empty but has URP and its assets preconfigured and installed.

Go to **Edit > Project Settings**, and open the **Graphics** panel. You'll see the URP Asset from the **Scriptable Render Pipeline Settings** as the selected SRP. 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.

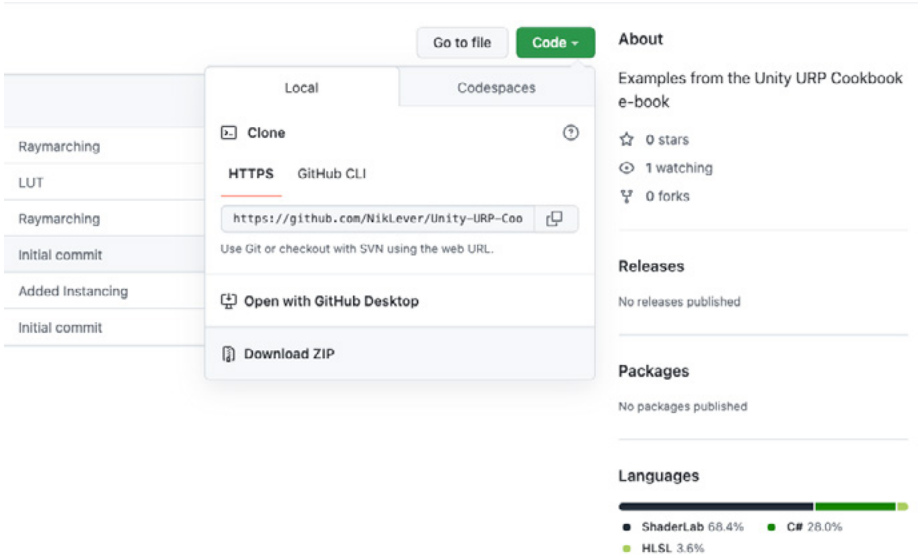
UniversalRP-HighFidelity is the default URP Asset selected, but you can switch to **UniversalRP-Balanced** or **UniversalRP-Performant**.



The Graphics panel in Project Settings

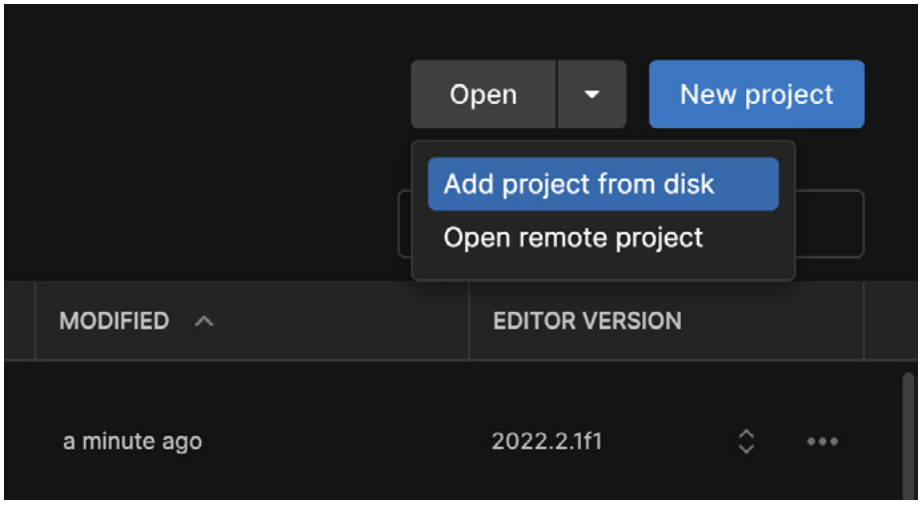
Importing e-book sample scenes

You can clone the repository from here or download the code in a zip file and unzip it.



The Github repository, where you can download the project by clicking the green Code button

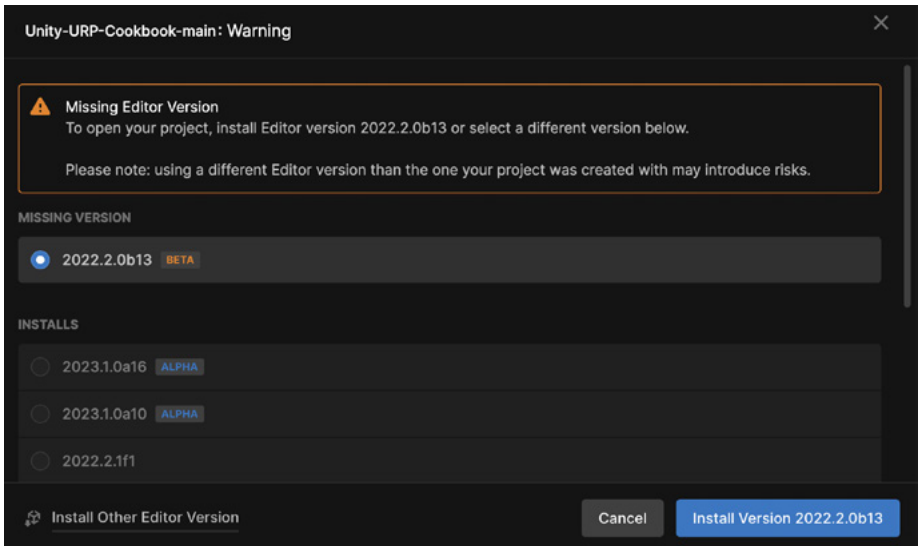
Once the project is unzipped and downloaded, import it from the Unity Hub via **Open > Add project from disk**.



How to import the sample project from Unity Hub

It's important that you are working in the same version of the Editor as that used for the sample project. If the Editor versions don't match, the Hub will show a warning message about a missing Editor version.

You can install the missing version from the blue button at the bottom right, as seen in this image.



It's good practice to install the version of the Unity Editor that matches any tutorial project you're following and/or downloading. Thankfully, this is easy to do via the Unity HUB.

Once the correct Editor version is installed, you will be able to open the project as normal.

Each recipe is contained in a folder along with the steps and files referred to in this book.

STENCILS

URP has two assets that control the final render, the Universal Renderer Data Asset and the URP Asset. From the former, you can add Renderer Features to be injected into any stage of the rendering pipeline, such as:

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

Renderer features provide you with ample opportunity to experiment with lighting and effects. This section will focus on Stencils, using only the bare minimum of required code.

To work along, open the sample scene via **Scenes > Renderer Features > SmallRoom - Stencil** in the Editor.

In the Made with Unity game *TUNIC* (created by Andrew Shouldice, TUNIC Team, 22nd Century Toys LLC, and Isometricorp Games Ltd., published by Finji), the main character's silhouette is drawn when props are blocking him. This effect can be achieved with Renderer Features in URP. It's also explained in this video tutorial. Stencils in action: As the magnifying glass moves over the desk, it can see through to reveal what is in the drawers.

Stencils in action: As the magnifying glass moves over the desk, it can see through to reveal what is in the drawers.

As the above image shows, the aim in this example is to convert the lens of the magnifying glass so it allows you to see through the desk, like an X-ray image. The approach uses a combination of Layer Masks, shaders, and Renderer Features. The first step is to change the material used by the lens, in this case a material called **MaskMat** with a shader called **Custom/StencilMask**.

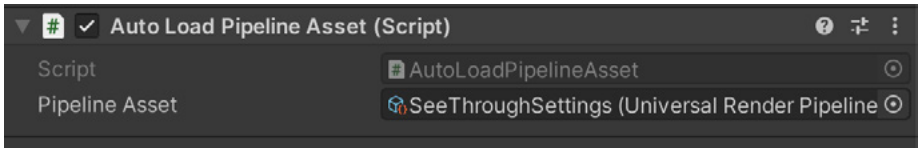
```
Shader "Custom/StencilMask"
{
       Properties{}

       SubShader{

              Tags {
                     "RenderType" = "Opaque"
              }

              Pass {
                     ZWrite Off

                     HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "Packages/com.unity.render-pipelines.
universal/ShaderLibrary/Core.hlsl"

            struct Attributes
            {
                float4 positionOS   : POSITION;
            };

            struct Varyings
            {
                float4 positionHCS  : SV_POSITION;
            };

            Varyings vert(Attributes IN)
            {
                Varyings OUT;

                OUT.positionHCS = TransformObjectToHClip(IN.
positionOS.xyz);

                return OUT;
            }

            half4 frag() : SV_Target
            {
                return (half4)0;
            }

                     ENDHLSL
              }
       }
}
```

Notice that Custom/StencilMask has the command `ZWrite Off`. In most cases, if you set `ZWrite Off` for an object, it will disappear as the render order of the object is changed and is rendered before the scene. If you change its render queue index to a higher value than Geometry, then it will reappear. For this example, it's been left at 2000, the Geometry value.

The only action you want the lens to perform is to write a value to the Stencil buffer. Since you need to consider the stencil writes and not the output of the shader to the color buffer, you can disable the color writes, `ColorMask 0`. This is a slightly optimized approach, especially if you want this to work with the Deferred Rendering path as the scene would be rendered before the lens mask.

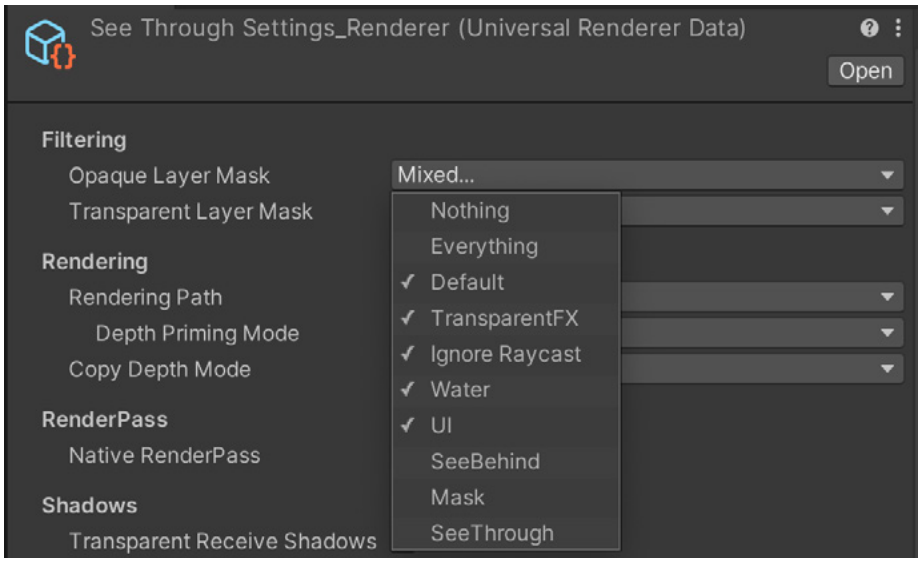
This example uses two custom layers, **Mask** and **SeeThrough**. The lens is in the Mask layer, while the desk, but not its children, is in the SeeThrough layer.

This scene uses the Renderer Data object named **See Through Settings_ Renderer**, located in the same folder as the scene file, materials, and shader: **Scenes > Renderer Feature Stencils**. The script attached to the Main Camera, **Auto Load Pipeline Asset**, ensures this is set as the Scriptable Render Pipeline Asset in **Project Settings > Graphics**. Now let's check the settings for this asset.



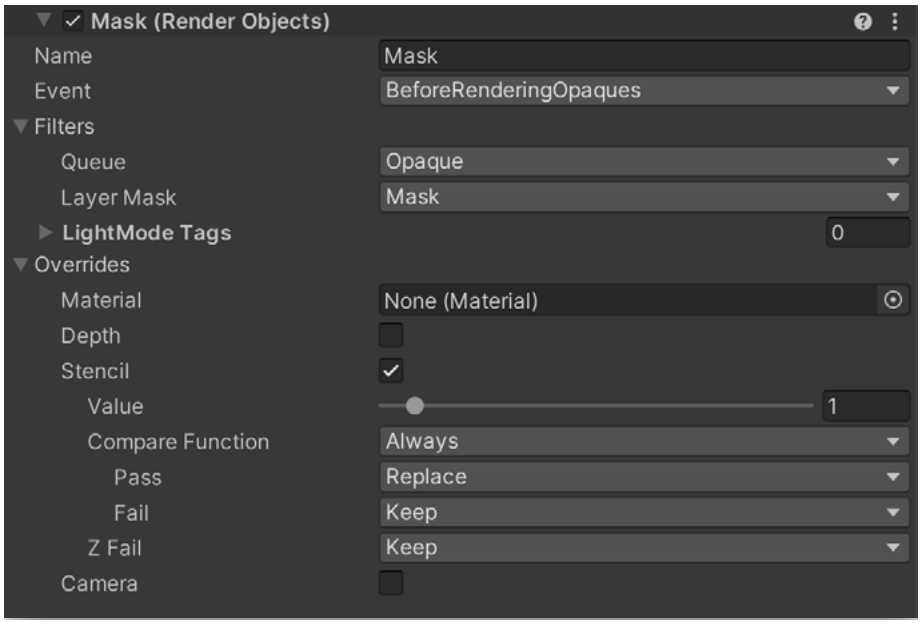
Pipeline Asset set for the Main Camera > Auto Load Pipeline Asset script

Select the **SeeThrough Settings_Renderer** via **Scenes > Renderer Feature Stencils**. The first setting changed from the default is the **Opaque Layer Mask**. Note that this excludes Mask and SeeThrough.



Changing the Opaque Layer Mask in the See Through Settings_Renderer

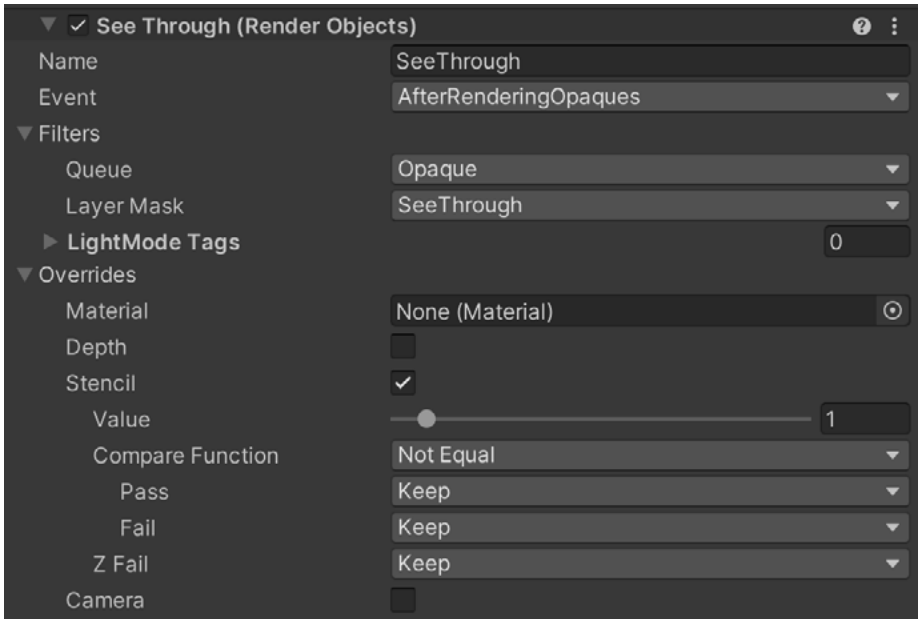
In the Renderer Features list in the Inspector, there are two Render Objects features named **Mask** and **SeeThrough**. If you disable the SeeThrough option, the desk disappears. This happens because, as part of a filtered-out layer that's using the Opaque Layer Mask, it's not a part of the default render – it only gets rendered because of the Render Objects feature.



Settings for Mask (Render Objects)

The image above shows that Mask is set to use the Event **BeforeRenderingOpaques** and be filtered so it only works on rendered pixels in the Mask Layer. In the Overrides panel, the **Stencil** option is enabled.The value it will save to the buffer is 1. To make sure this write happens, the **Compare Function** is set to **Always**, and **Pass** is set to **Replace** so it always replaces the existing value. **Fail** and **Z Fail** are set to **Keep**.

URP will attempt to render the Mask Layer. Since no override material is set, it will use the materials defined by the objects in this Mask Layer, which is just the lens with the MaskMat material and the StencilMask shader. Setting Compare Function to **Always** and Pass to **Replace** ensures that the Stencil buffer is wherever the lens is in vision, with the value for each pixel set to 1.



The settings for See Through (Render Objects)

Let's look at the second Render Objects Renderer Feature (shown above). This is set to use the Event **AfterRenderingOpaques**, meaning it will apply after the Stencil buffer has been set. Its **Layer Mask** is set to **SeeThrough** and **Value** set to **1**. If the Value 1 is found, the pixel shouldn't be rendered.

The Compare Function setting is set to **Not Equal**, while Pass, Fail, and Z Fail are all set to **Keep**. This Render Objects pass will only read from the Stencil buffer but not write to it. So this pass will render any pixel in the layer See Through, where the Stencil buffer does not contain the value 1. It leaves the default render only where the lens is. Try changing the Compare Function to **Equal** to flip the result so the desk appears in the lens only.

The effect of changing the Compare Function to Equal

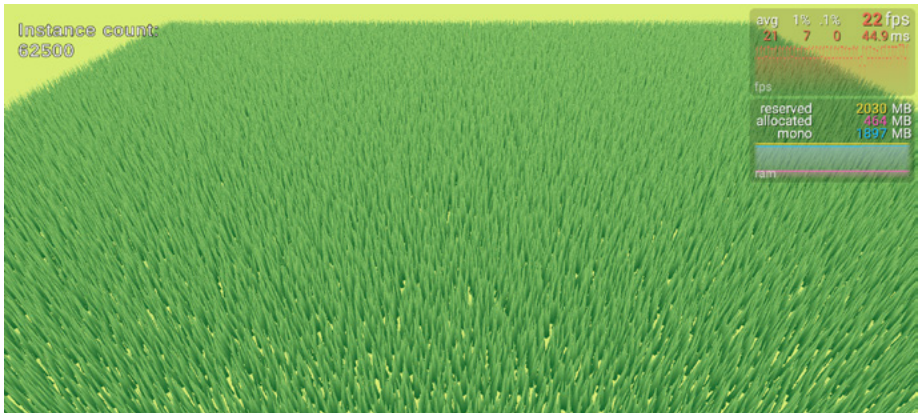
Renderer Features are a great way to achieve dramatic custom effects.

INSTANCING

Exchanging data between the CPU and GPU is a major bottleneck in the rendering pipeline. If you have a model that needs to be rendered many times using the same geometry and material, then Unity provides some great tools to do so, which are covered in this chapter.

A field full of grass will be used to illustrate the concept of instancing. It's far from photorealistic but sufficient to illustrate the techniques involved. You'll find the example in the **Scenes > Instancing** folder.

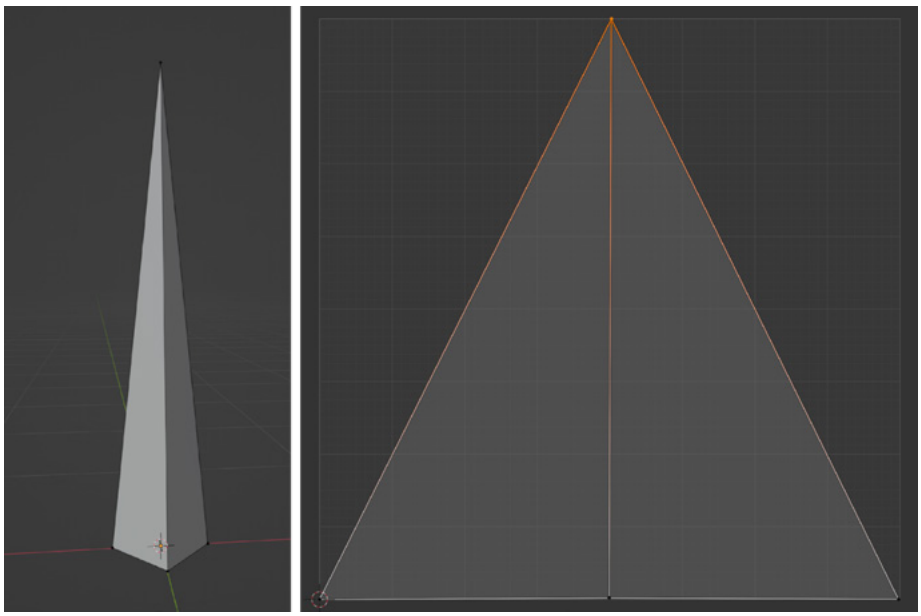
> **Note:** Thanks go to the author of the article, "Making Grass in Unity with GPU Instancing," for the assets.



A field of grass rendered using an SRP Batcher-compatible material

The popular Made with Unity game *Genshin Impact*, by HoYoverse, features a vast open world with lush vegetation. It runs on all the major platforms, from mobile devices to the latest consoles. This section offers tips on how to recreate a similar grass effect in a performant way.

To start, you need a single blade of grass and just two triangles, to keep things simple. The UV is set so the base of each grass blade has a **V** value of **0** and the tip a V value of **1**. You can use this to offset the tip vertex to simulate wind.

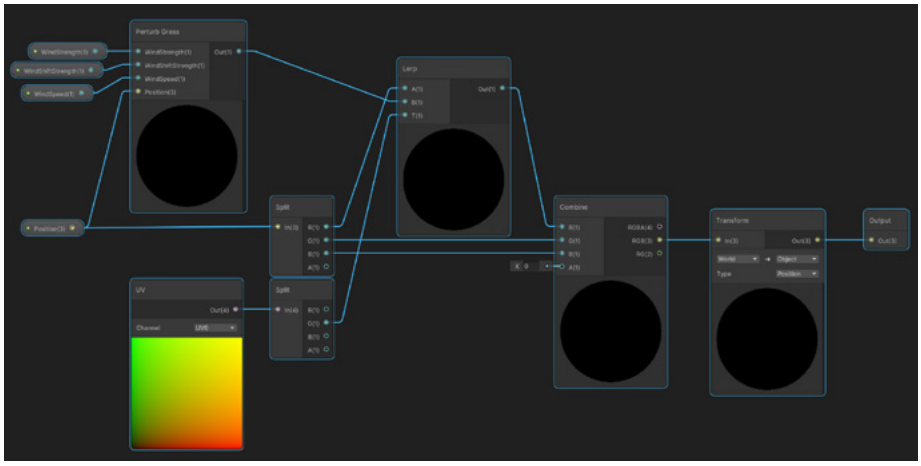


Grass blade model and UV

SRP Batcher

Take a look at the Shader Graph subgraph in the folder **Scenes > Instancing > Common > Grass Wave**. The aim of this is to perturb the X value of the object's vertex based on WindSpeed, WindShiftStrength, and WindStrength. To ensure that all the grass blades behave slightly differently, a **Noise** node is used in the subgraph Perturb Grass. The vertex Y and Z positions are passed directly to the output, but the offset for the X value is processed using a **Lerp** node.

The T input, which controls the interpolation, comes from the UV's V value. At the base of the grass blade, this is 0, meaning the result of lerp will be the A input to the lerp, which is the modeled position. The tip of the blade V is 1, ensuring that the result of the lerp is the B input, the processed offset.



The Grass Wave subgraph

Now that you have a method of deforming each blade, it's time to turn this into a complete shader that you can use as the material shader for each blade of grass.

Take a look in the folder **Scenes > Instancing > 1 - SRP Batcher > SRP Batcher Shader**. This is a simple shader, just the Grass Wave subgraph controlling the Vertex > Position and a Sample Texture 2D acting as the base color input for the fragment shader.

Now, let's use the following code to populate a field of grass.

```
_startPosition = -_fieldSize / 2.0f;
_cellSize = new Vector2(_fieldSize.x / GrassDensity, _
fieldSize.y / GrassDensity);

var grassEntities = new Vector2[GrassDensity, GrassDensity];
var halfCellSize = _cellSize / 2.0f;

for (var i = 0; i < grassEntities.GetLength(0); i++) {
for (var j = 0; j < grassEntities.GetLength(1); j++) {
```

```
            grassEntities[i, j] =
                  new Vector2(_cellSize.x * i + _
startPosition.x,
                              _cellSize.y * j + _
startPosition.y) +
new Vector2( Random.Range(-halfCellSize.x, halfCellSize.x),
                               Random.Range(-halfCellSize.y,
halfCellSize.y));
      }
}
_abstractGrassDrawer.Init(grassEntities, _fieldSize);
```

Looking more closely at this code example you see:

- `_fieldSize` is (40, 40)
- `_startPosition` is (-20, -20)
- `GrassDensity` is set to 250 in the Github sample
- `cellSize` is (0.16, 0.16).
- Two loops are iterated through setting each element of the `_grassEntities` 2D array
- Base position for each blade is `_startPosition` plus the current cell; then a small random factor is introduced
- `_abstractGrassDrawer` is a base class for two versions of using the grass-populating code
    - For the initial version, ignore GPU Instancing and see how well SRP batcher handles the problem by opening and running the scene **Scenes > Instancing > 1 - SRP Batcher > 1 - SRP**
    - First, you need to populate the scene with the grass blade model Prefab, at each position in the `grassEntities` 2D Array. The code is in the file **Scenes > Instancing > Scripts > GameObjectGrassDrawer.cs**

```
public override void Init(Vector2[,] grassEntities, Vector2
fieldSize) {
_grassEntities = new GameObject[grassEntities.GetLength(0),
     grassEntities.GetLength(1)];
      for (var i = 0; i < grassEntities.GetLength(0); i++) {
            for (var j = 0; j < grassEntities.GetLength(1);
j++) {
                  _grassEntities[i, j] =
Instantiate(_grassPrefab,
                                        new Vector3(
    grassEntities[i, j].x,
    0.0f,
```

```
    grassEntities[i, j].y),
     Quaternion.identity);
             }
       }
}
```

Here, you iterate over the `grassEntities` Array using Instantiate to create a new GameObject from the assigned Prefab. It works but dramatically impacts the frame rate for the scene. You can see from the image of the grass field on page 15 that the frame rate is a sluggish 22 fps for 62,500 blades, running on a 2020 iMac with the following specs:

- Retina 5K, 27-inch, 2020
- Processor: 3.8 GHz 8-Core Intel Core i7
- Memory: 32 GB 2667 MHz DDR4
- Startup Disk: Macintosh HD
- Graphics: AMD Radeon Pro 5500 XT 8 GB

How can you optimize the scene?

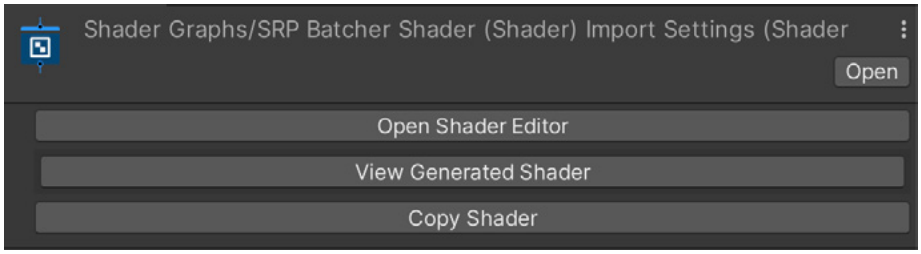
> **Note:** For a non-square terrain, you could create a draw tool saving each blade position in a list. For example, this blog post explains how to build a tool to streamline placing objects in the scene every time you click in it.

GPU Instancing

One optimization technique is to enable GPU instancing. Look at **Scenes > Instancing > 2 - GPU Instancing > 2 - GPU Instancing** from the Github samples for an example of this technique.

A material setting called **Enable GPU Instancing** instructs the renderer to batch any models that use the same material, thereby reducing the number of draw calls. The setting is available in the Advanced Options panel.

The SRP Batcher and GPU Instancing are mutually exclusive. When using URP, if a material is compatible with the SRP Batcher, then SRP Batcher will be used, even if Enable GPU Instancing is selected. A shader created with Shader Graph is compatible with SRP Batcher by default. To disable SRP Batcher compatibility, select the Shader Graph that will create the HLSL shader, and click on **View Generated Shader** in the Inspector.



Generating an HLSL shader from Shader Graph

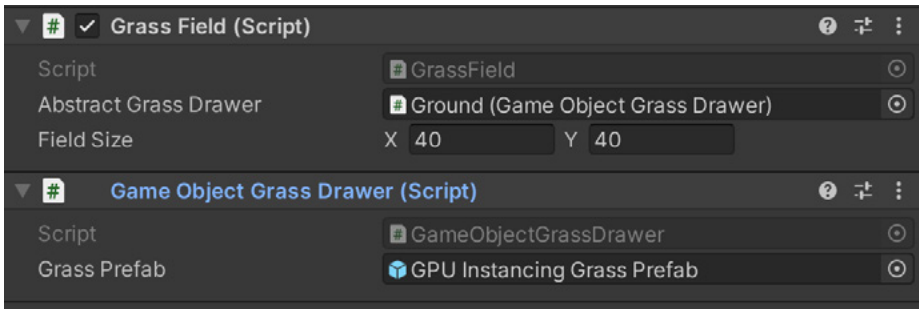
The shader will be created, placed in the Temp folder, and opened in Visual Studio or your chosen code editor. Change the Shader name to:

Shader "Custom/GPU Instancing Shader"

Then search for CBUFFER, and comment out the CBUFFER macros:

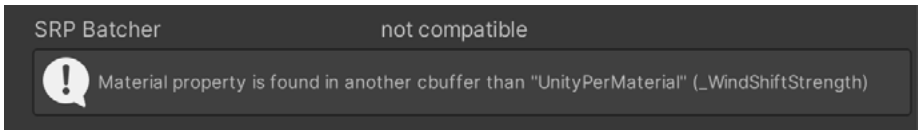
```
// Graph Properties
//CBUFFER_START(UnityPerMaterial)
            float4 _MainTexture_TexelSize;
            half _WindShiftStrength;
            half _WindSpeed;
            half _WindStrength;
//CBUFFER_END
```

Save the shader in Assets.



Scripts assigned to the Ground GameObject in the GPU Instancing scene

Notice the GPU Instancing scene uses the same version of **Abstract Grass Drawer** as the SRP Batcher scene. The only difference is the **GameObjectGrassDrawer** version in GPU Instancing is assigned a different Prefab with a material that uses the GPU Instancing shader.



GPU Instancing Shader is not compatible with SRP Batcher

If you check the GPU Instancing shader in the Inspector, you can see it's not compatible with SRP Batcher.

Any change to the graph that you used to generate the code will necessitate repeating the customization steps:

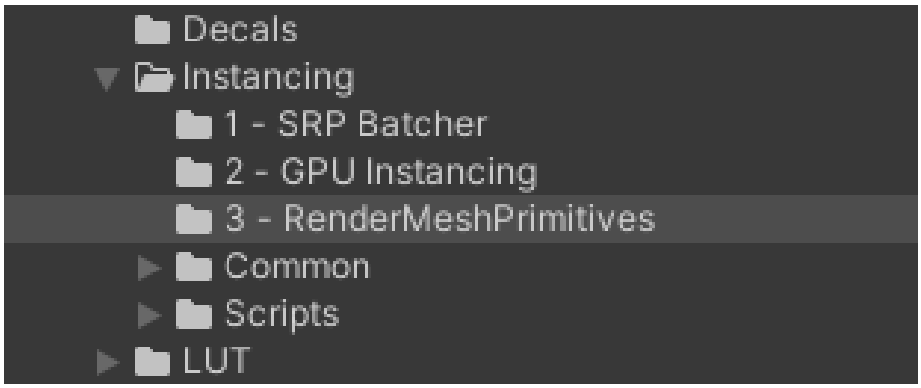
1. View Generated Shader or Regenerate
2. Edit the Shader name
3. Comment out the CBUFFER macros
4. Save to Assets

However, after all this work, the testing shows only a marginal improvement over SRP Batcher, probably due to being CPU bound. There has to be a better way.

RenderMeshPrimitives

The Unity Graphics API has a number of methods for directly rendering a mesh by bypassing the need for a GameObject. The method used here is RenderMeshPrimitives, a feature introduced in Unity LTS 2021. Prior to that, you would have needed to use DrawMeshInstancedProcedural, which is now marked as obsolete.

With RenderMeshPrimitives, you should use a material that sources the individual mesh position using a ComputeBuffer. You can see it in action by viewing the scene **Scenes > Instancing > 3 - RenderMeshPrimitives > 3 - RenderMeshPrimitives**.



The instancing scenes in the Project View

As you can see from the following image of the grass field, the improvement in frame rate is nothing short of a remarkable – 377 fps. The scenes created with SRP Batcher and GPU Instancing were running at around 20 fps.

The difference in this case is that the grass field is rendered using a single draw call.

Event #4: Draw Mesh (instanced)	(1 draw calls, 62500 instances)
Shader	Shader Graphs/Instanced Grass Shader Graph, SubShader #0
Pass	Universal Forward
Keywords	PROCEDURAL_INSTANCING_ON
Blend	One Zero
ZClip	True
ZTest	LessEqual
ZWrite	On
Cull	Back
Conservative	False

Frame Debugger stats for the grass field

The grass field rendered using RenderMeshPrimitives

You achieve this by making the positions of each blade a **Material** property. The data to render the blades resides on the GPU, which uses its parallelism to render the entire field at an optimal speed.

Let's review the code to generate the positions. You'll find it in the `UpdatePositions` method in the file **Scenes > Instancing > Scripts > InstancedGrassDrawer.cs**.

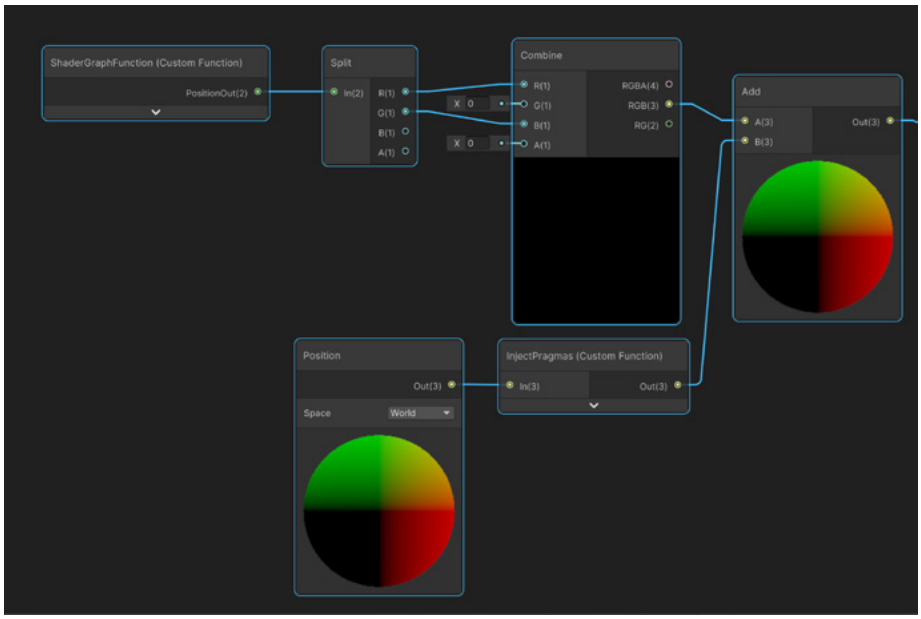
```
_positionsCount = _positions.Count;
_positionBuffer?.Release();
if (_positionsCount == 0) return;
_positionBuffer = new ComputeBuffer(_positionsCount, 8);
_positionBuffer.SetData(_positions);
_instanceMaterial.SetBuffer(Shader.
PropertyToID("PositionsBuffer"), _positionBuffer);
```

`_positions` holds a Vector2 List of grass positions. If `_positionsBuffer` exists, then you release it. If you're unfamiliar with a "?" following a variable, it's a null check, meaning it's shorthand for:

```
if (positionsBuffer != null) _positionsBuffer.Release()
```

You create a ComputeBuffer that takes a count parameter and the byte size of each item. A Vector2 contains two floats. A single float is 32 bits or 4 bytes, making two floats 8 bytes. It's simple to populate a ComputeBuffer by using `SetData` passing the `_positions` List. Now you can use the `SetBuffer` method to copy this to the material. You'll access this buffer in the material using the name `positionsBuffer`.

Take a look at the graph in **Scenes > Instancing > 3 - RenderMeshPrimitives > Instanced Grass Shader**.



Getting the vertex position from a Compute Buffer

Starting at the bottom, you can see the **Space** parameter for Grass Mesh vertex position is set to **World**. But there's an important code block that needs adding whenever you use this technique: A `#pragma` is required by any meshes rendered using RenderMeshPrimitive. This is done using a custom function. Instead of sourcing the function from a file, you add a string:

```
#pragma instancing_options procedural:ConfigureProcedural
Out = In;
```

The code method now used by this shader to generate positional values will come from a function with the name `ConfigureProcedural`. Other than that, this Custom Function node simply passes its input, In, to its output, Out.

The heavy lifting is done in the Custom Function called `ShaderGraphFunction`, which is found in the file InstancedPosition, in the same folder as the scene file.

```
#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
StructuredBuffer<float2> PositionsBuffer;
#endif

float2 position;

void ConfigureProcedural () {
	#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
	position = PositionsBuffer[unity_InstanceID];
	#endif
}

void ShaderGraphFunction_float (out float2 PositionOut) {
	PositionOut = position;
}
```

The position is set using the `ConfigureProcedural` method and passed to the output using the `ShaderGraphFunction` for which the script has float and half versions.

At this point in the graph, the individual blade location is a `float2` with the first float being the X value and the second the Z. A **Split** node is used to convert this into the individual floats, and a **Combine** node to move the second float to the third. The Split and Combine nodes call the individual floats RGBA not XYZW, but by moving G to B, you're effectively moving Y to Z. The blade and vertex positions are now established, and you can combine these to get the actual world position of the vertex.

With this shader ready, you now use it with a material that has the inputs WindSpeed, WindStrength, WindShiftStrength, and MainTexture, the same as those used by the SRP Batcher and GPU Instancing versions. The only difference is in how the position of each vertex is calculated. Refer back to the script `InstancedGrassDrawer.cs` to see how to render the grass blades. The variables in the script are initialized in the `Init` method called by the `Awake` method of the `GrassField.cs` script.

```
public override void Init(Vector2[,] grassEntities, Vector2
fieldSize) {

_grassEntities = grassEntities;
	_grassBounds = new Bounds(transform.position,
			new Vector3(fieldSize.x, 0.0f,
fieldSize.y));
	_positions = new List<Vector2>();
	_renderParams = new RenderParams(_instanceMaterial);
	_renderParams.worldBounds = _grassBounds;
	_renderParams.shadowCastingMode = ShadowCastingMode.Off;
}
```

To use `Graphics.RenderMeshPrimitives`,you need a `RenderParams` instance. This is created from the assigned Material, _instanceMaterial. Two other properties are additionally assigned.

The actual rendering is done using the Update callback:

```
private void Update() {
      if (_positionsCount == 0) return;
      Graphics.RenderMeshPrimitives(_renderParams, _instanceMesh, 0,
_positionsCount);
}
```

RenderMeshPrimitives takes four parameters, a `RenderParams` instance, the mesh to render, a submesh index, and a count value identifying how many copies to render. When using the shader, each copy will have a unique unity_InstanceID, which will have the value 0 to count -1.

Rendering using a ComputeBuffer is a fast and fairly simple setup. By manipulating the `_positionBuffer`, you could mow the grass or blow it away. To avoid passing data between the CPU and the GPU, this is best handled with a ComputeShader.

More resources

— Assets for this recipe

— Example project using DrawMeshInstancedIndirect

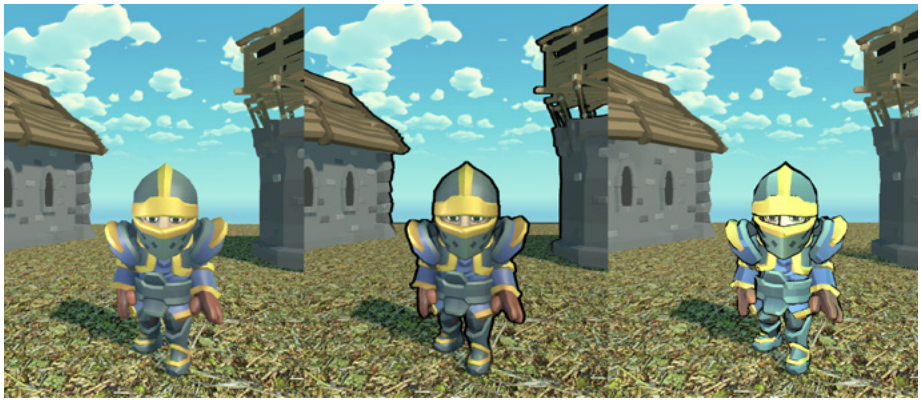
— GPU Instancing documentation

— GPU Instancing article from CatLikeCoding

— Using ComputeBuffers for instancing

TOON AND OUTLINE SHADING

This recipe is based on common ways of creating a toon shader and an outline shader.



One scene, three different looks: Standard shading (left), with post-processing (center), and per-material shading (right)

Often used together, toon and outline shaders present two distinct challenges. The toon shader takes the color that would be created using a URP-compatible Lit shader, and ramps the output rather than allowing continuous gradients, thereby requiring a custom lighting model. In this example, it will be created using Shader Graph. However, Shader Graph doesn't support custom lighting, so there's no node available to directly access the Main and Additional lights. Instead, you can leverage a custom node to access these.

The third-person action-shooter and Made with Unity game *Rollerdrome*, by Roll7, has a distinctive art direction that makes the game look like a comic book, achieved with cel shading techniques. Don't miss this interview with the creators.

To see what the shader looks like, go to **Scenes > Toon Shading > Simple Toon Shading**.

The scene with the simple ramped toon shader

Toon shader

This example only supports the Main light and a textured mesh to keep things simple. No outlining, additional lights, global illumination, or lightmapping is included. These features will be covered later in this chapter.

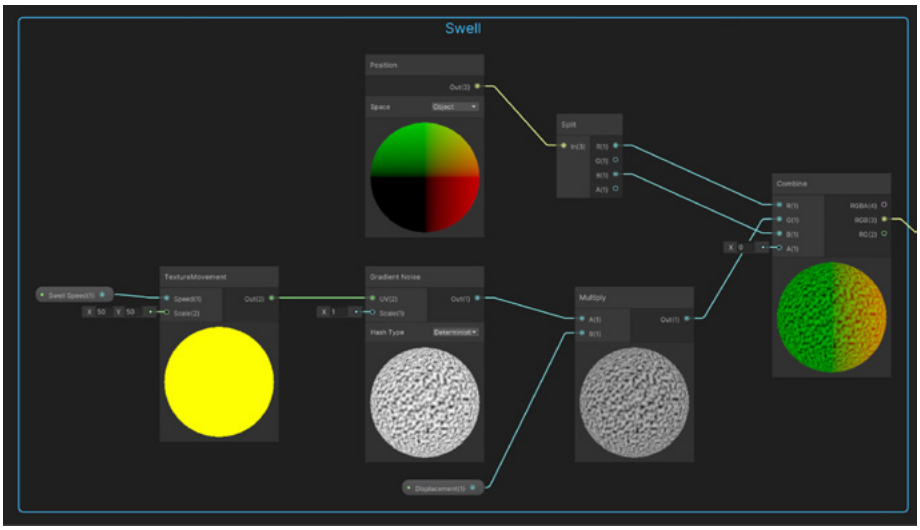
Let's start by accessing the Main Light using a custom function, which you'll find in the file **HLSL > Custom Lighting.hlsl**.

```
void MainLight_float(float3 WorldPos, out float3 Direction, out
float3 Color, out float DistanceAtten, out float ShadowAtten)
{
#ifdef SHADERGRAPH_PREVIEW
    Direction = float3(0.5, 0.5, 0);
    Color = 1;
    DistanceAtten = 1;
    ShadowAtten = 1;
#else
       float4 shadowCoord =
TransformWorldToShadowCoord(WorldPos);

    Light mainLight = GetMainLight(shadowCoord);
    Direction = mainLight.direction;
    Color = mainLight.color;
    DistanceAtten = mainLight.distanceAttenuation;

       #if !defined(_MAIN_LIGHT_SHADOWS) || defined(_RECEIVE_
SHADOWS_OFF)
              ShadowAtten = 1.0h;
       #else
         ShadowSamplingData shadowSamplingData =
GetMainLightShadowSamplingData();
         float shadowStrength = GetMainLightShadowStrength();
         ShadowAtten = SampleShadowmap(shadowCoord, TEXTURE2D_
ARGS(_MainLightShadowmapTexture,
         sampler_MainLightShadowmapTexture),
         shadowSamplingData, shadowStrength, false);
    #endif
#endif
}
```

It's good practice to add a block of code inside a `#ifdef SHADERGRAPH_PREVIEW` preprocessor directive that defines the behavior while creating the Shader Graph Asset. This specifies the values to default to in the graph preview window.

The WorldPos is converted into a shadow coordinate using the function `TransformWorldToShadowCoord`. The functions used in this code come from the Universal Render Pipeline package and are available to custom functions in Shader Graph. When the function `GetMainLight` is used with a float4, the **ShadowAttenuation** property of the returned light is set. This is needed in the graph that uses this custom function.

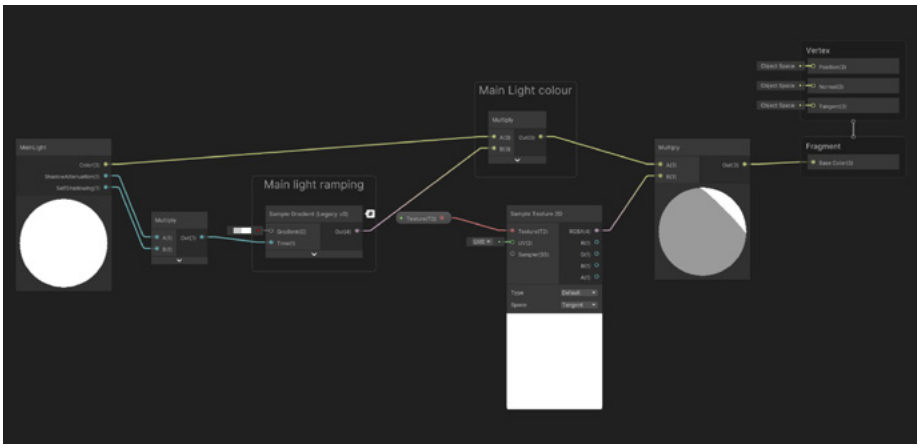
This code is used in the Main Light subgraph (see image below), which you'll find in the folder **Shaders > Subgraphs**. Let's review it.



Main Light subgraph

The Custom Function node takes a **Position** node set to Absolute World as its only input. The function returns Direction, Color, DistanceAtten (which remains unused), and ShadowAtten. To allow for self shadowing, you'll need to get the dot product of the light direction and the World normal, and clamp this between 0 and 1. You don't want negative values.

Now that you have a way of accessing the Main Light, you can use it to create a simple toon shader. Take a look at **Shaders > Simple Toon** to see the graph (also in image below).



Simple Toon graph

The first node is the Main Light subgraph. The ShadowAttenuation and SelfShadowing outputs are multiplied together. The trick is to pass this output into a **Sample Gradient** node that works with a ramped gradient, so light levels are not smooth, but instead jump in stages based on the gradient.

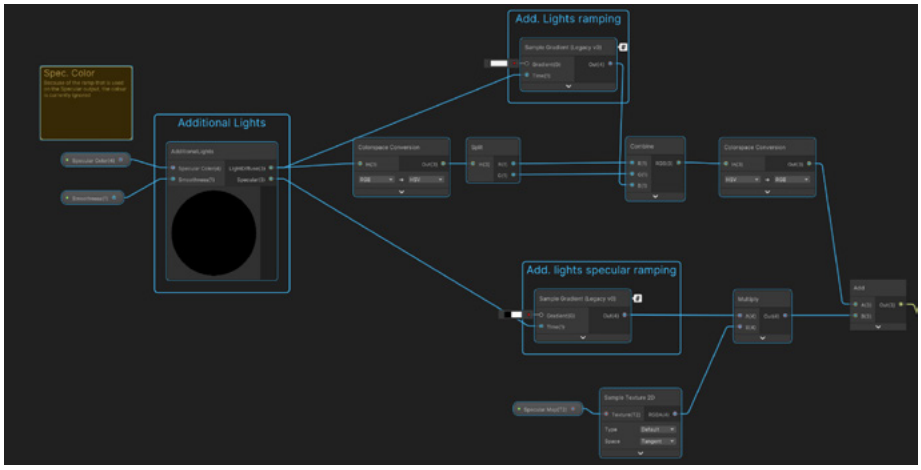
Taking a smoothly changing input and processing it with a gradient is a useful technique for a number of shading challenges. The rest of the graph combines light color with the ramped level, then combines this with the sampled texture to generate the color to use for the base color.

The limitation of this simple graph is that it does not take into account global illumination and additional lights. The graph **Shaders > Toon Shader** has all these features and adds an outline effect. Let's look at how to access the additional lights before covering how to add outlines.

Once again, a custom function is required, which is in the same HLSL file as the one for the Main Light, **HLSL > CustomLighting.hlsl**:

```
void AdditionalLights_float(float3 SpecColor, float Smoothness,
float3 WorldPosition, float3 WorldNormal, float3 WorldView, out
float3 Diffuse, out float3 Specular)
{
    float3 diffuseColor = 0;
    float3 specularColor = 0;

#ifndef SHADERGRAPH_PREVIEW
    Smoothness = exp2(10 * Smoothness + 1);
    WorldNormal = normalize(WorldNormal);
    WorldView = SafeNormalize(WorldView);
    int pixelLightCount = GetAdditionalLightsCount();
    for (int i = 0; i < pixelLightCount; ++i)
    {
        Light light = GetAdditionalLight(i, WorldPosition);
        half3 attenuatedLightColor = light.color * (light.
distanceAttenuation * light.shadowAttenuation);
        diffuseColor += LightingLambert(attenuatedLightColor,
light.direction, WorldNormal);
        specularColor += LightingSpecular(attenuatedLightColor,
light.direction, WorldNormal, WorldView, float4(SpecColor, 0),
Smoothness);
    }
#endif

    Diffuse = diffuseColor;
    Specular = specularColor;
}
```

The code requires several inputs: Specular Color, a Smoothness float, the WorldPosition, WorldNormal, and WorldView direction. It outputs a combined Diffuse Color and Specular Color. It does this by iterating over each additional light and accumulating the diffuseColor using the `LightingLambert` function. This is the simplest lighting model using only the light direction and the WorldNormal. The specularColor is accumulated using the `LightingSpecular` function.



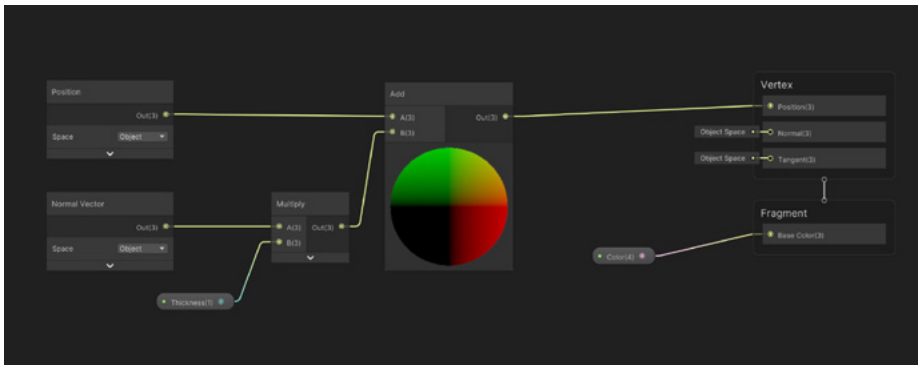
Additional lights contribution to the Toon Shader Graph

The image above shows how this code is used. The Diffuse level is adjusted using a ramp. This displays another useful trick. Converting the colorspace from RGB to HSV and scaling the V or B component is similar to adjusting the light level, then you convert back to RGB.

The complete Toon graph is worth a closer study because it displays many useful techniques to use in your own custom shaders.

Outline

The simplest technique for adding outlines is to add a second pass that only renders back-facing polygons and uses a vertex shader that moves the vertex a small amount along the vertex normal. This shader is included in the Github samples via **Shaders > VertexOutline**; its graph is shown here:



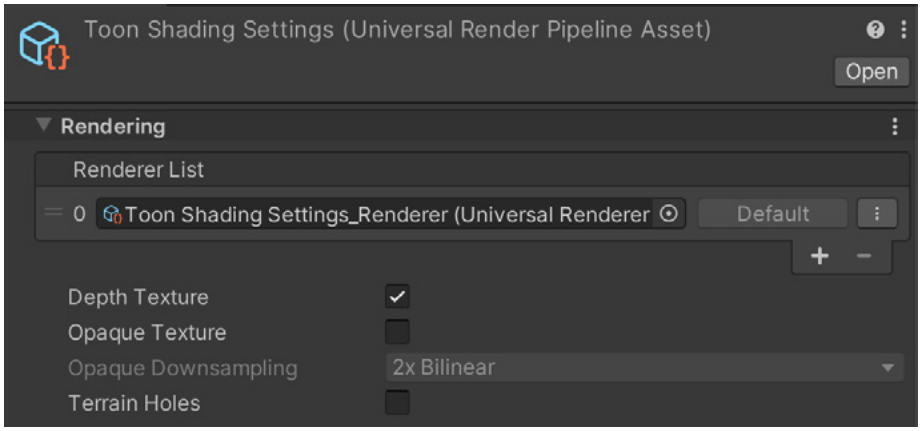
An outline shader using a back-facing vertex shift technique

The **Normal Vector** node, with Space set to Object, is fed into a **Multiply** node. This is multiplied by **Thickness** value for the material. The output from this is added to the Object Position, moving the vertex position slightly out from the object modeled location. This is the input to the **Vertex Position** property. The shader property **Universal > Render Face** is set to **Back** using the panel in **Graph Inspector > Graph Settings**. A shader graph allows a single pass only so to add this to the render you need to add a second material using the GameObject Inspector.

Adding a second material

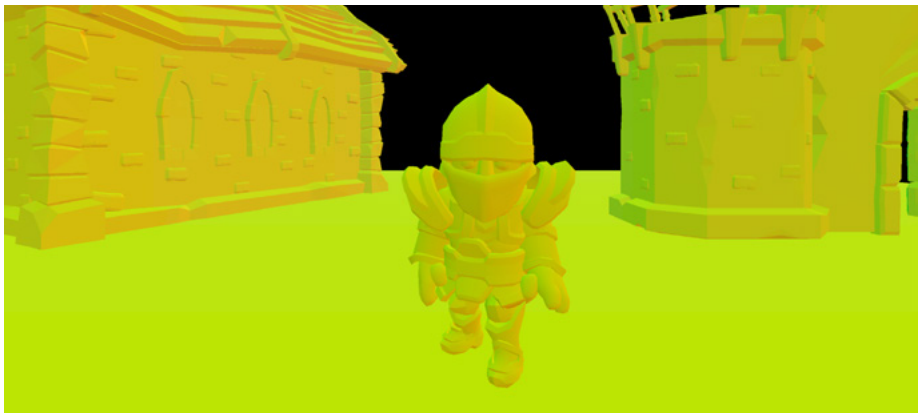
The scene in **Scenes > Toon Shading > Simple Toon Shading** shows the second material being used. View the material **VertexOutline** in the same folder, in the Inspector, and set Thickness to 0.01.

A more sophisticated technique uses edge detection, which is illustrated in the following two variations that sample either the depth or normal textures, or both, in a number of places to look for sharp differences indicating an edge. The depth texture is supplied by using the Inspector for the URP Asset. Checking this means a texture called _CameraDepthTexture is created with depth information stored in the red channel.



Creating _CameraDepthTexture

The HLSL script used by the Toon Shader graph for edge detection also scans the normals. How do you get the normal information? You'll need to use a Renderer Feature. Take a look at **Renderer Features >** **DepthNormalsFeature.cs**, and notice this is set as a Renderer Feature for **Toon Shading Settings_Renderer**. This Renderer Feature saves the normal information to the texture named _CameraDepthNormalsTexture.



The _CameraDepthNormalsTexture created by theDepthNormalsFeature Renderer Feature

Now that you have these two textures, generating the outline requires scanning both images at left, right, up, and down of the current UV position. You can see the code used in the Toon Shader graph by looking at the file **HLSL > Outline.hlsl**:

```
void OutlineObject_float(float2 UV, float OutlineThickness,
float DepthSensitivity, float NormalsSensitivity, out float
Out)
{
    float halfScaleFloor = floor(OutlineThickness * 0.5);
    float halfScaleCeil = ceil(OutlineThickness * 0.5);

    float2 uvSamples[4];
    float depthSamples[4];
    float3 normalSamples[4];

    uvSamples[0] = UV - float2(_CameraDepthTexture_TexelSize.x,
_CameraDepthTexture_TexelSize.y) * halfScaleFloor;
    uvSamples[1] = UV + float2(_CameraDepthTexture_TexelSize.x,
_CameraDepthTexture_TexelSize.y) * halfScaleCeil;
    uvSamples[2] = UV + float2(_CameraDepthTexture_TexelSize.x
* halfScaleCeil, -_CameraDepthTexture_TexelSize.y *
halfScaleFloor);
    uvSamples[3] = UV + float2(-_CameraDepthTexture_
TexelSize.x * halfScaleFloor, _CameraDepthTexture_TexelSize.y *
halfScaleCeil);

    for(int i = 0; i < 4 ; i++)
    {
        depthSamples[i] = SAMPLE_TEXTURE2D(_CameraDepthTexture,
sampler_CameraDepthTexture, uvSamples[i]).r;
        normalSamples[i] = DecodeNormal(SAMPLE_TEXTURE2D(_
CameraDepthNormalsTexture, sampler_CameraDepthNormalsTexture,
uvSamples[i]));
    }

    // Depth
    float depthFiniteDifference0 = depthSamples[1] -
depthSamples[0];
    float depthFiniteDifference1 = depthSamples[3] -
depthSamples[2];
    float edgeDepth = sqrt(pow(depthFiniteDifference0, 2) +
pow(depthFiniteDifference1, 2)) * 100;
    float depthThreshold = (1/DepthSensitivity) *
depthSamples[0];
    edgeDepth = edgeDepth > depthThreshold ? 1 : 0;

    // Normals
    float3 normalFiniteDifference0 = normalSamples[1] -
normalSamples[0];
    float3 normalFiniteDifference1 = normalSamples[3] -
normalSamples[2];
    float edgeNormal = sqrt(dot(normalFiniteDifference0,
normalFiniteDifference0) + dot(normalFiniteDifference1,
normalFiniteDifference1));
```

```
    edgeNormal = edgeNormal > (1/NormalsSensitivity) ? 1 : 0;

    float edge = max(edgeDepth, edgeNormal);
    Out = edge;
}
```

Notice how the `uvSamples` array is created by adding or subtracting a float2 value from the current UV position, which is one input to the function. The size of the offset from the input UV is based on the **OutlineThickness** property.

You build up an array of depthSamples and normalSamples by sampling the textures _CameraDepthTexture and _CameraDepthNormalsTexture respectively. Then, for both the depth and normal, you get the difference0 by subtracting the 0th item in each generated samples array from the first `(depthSamples[1] - depthSamples[0])` and the difference1 by subtracting the second item from the third. These are the differences from bottom-left to top-right and from bottom-right to top-left.

For `depthSamples`, which are simple floats, you square the differences, add them together, get the square root, then test this against a calculated depthThreshold based on the reciprocal of the DepthSensitivity input. For normals which are float3 values, instead of squaring each difference, you get the dot product of each difference with itself. The dot product of a vector with itself is the square of its magnitude which is exactly what is required.

Finally, the output of the function is the max of either `edgeDepth` or `edgeNormal`. This function returns 0 if you show an outline at this pixel, and 1 if not. It can be added to a graph and used to multiply the calculated color. If the function returns 0, then the calculated color will revert to black.

Another approach to this problem is to add a post-processing effect. Take a look at the scene in **Scenes > Toon Shading > SobelFilter Shading**. A post-processing effect works on the rendered output. The Renderer Feature named BlitMaterialFeature is used (located in the Renderer Features folder). This script processes the rendered image using a Blit function. Optionally, this can take a material to use when copying pixels, so rather than simply copying each pixel from source to destination, each pixel can be processed and the final color of the pixel adjusted.

The material used in this example is **SobelFilter**, which uses the shader **Shaders > SobelFilter.shader.** The main work is done using a Sobel filter:

```
float sobel (float2 uv)
{
float2 delta = float2(_Delta, _Delta);

float hr = 0;
float vt = 0;
```

```
hr += SampleDepth(uv + float2(-1.0, -1.0) * delta) *  1.0;
hr += SampleDepth(uv + float2( 1.0, -1.0) * delta) * -1.0;
hr += SampleDepth(uv + float2(-1.0,  0.0) * delta) *  2.0;
hr += SampleDepth(uv + float2( 1.0,  0.0) * delta) * -2.0;
hr += SampleDepth(uv + float2(-1.0,  1.0) * delta) *  1.0;
hr += SampleDepth(uv + float2( 1.0,  1.0) * delta) * -1.0;

vt += SampleDepth(uv + float2(-1.0, -1.0) * delta) *  1.0;
vt += SampleDepth(uv + float2( 0.0, -1.0) * delta) *  2.0;
vt += SampleDepth(uv + float2( 1.0, -1.0) * delta) *  1.0;
vt += SampleDepth(uv + float2(-1.0,  1.0) * delta) * -1.0;
vt += SampleDepth(uv + float2( 0.0,  1.0) * delta) * -2.0;
vt += SampleDepth(uv + float2( 1.0,  1.0) * delta) * -1.0;

return sqrt(hr * hr + vt * vt);
}
```

A Sobel filter is a 3×3 matrix that works to analyze each pixel of a rendered image. It gets its name from the digital image researcher, Irwin Sobel. You scan the image both horizontally and vertically, accumulating a value and return the square root of the sum of the squares of both passes. The final color of each fragment/pixel is generated using this code:

```
half4 frag(Varyings input) : SV_Target
{
float s = pow(1 - saturate(sobel(input.uv)), 50);
half4 col = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.
uv);
return col * s
}
```

The output of Sobel is clamped to between 0 and 1 using the HLSL function `saturate`; subtract this from 1 to invert the value, and raise this to the 50th power. Invert the value, since the Sobel function returns a larger value at an edge. In this case, you want 0 at an edge and 1 away from the edge. Then you sample the rendered image _MainTex, and return the multiple of these two values col * s.

More resources

— Unity Open Project Github (most of the code from this chapter is from the Open Project)

— Unity Open Project on YouTube

— YouTube tutorials from Ned Makes Games

— YouTube tutorial about using Unity's SobelFilter.shader from AE Tuts

— Edge detection using a Sobel filter by Alexander Ameye

AMBIENT OCCLUSION

Ambient Occlusion

Ambient Occlusion is a post-processing technique available in versions Unity 2020.2 and URP 10.0 and newer. The effect darkens creases, holes, intersections, and surfaces that are close to one another. In the real world, such areas tend to block out or occlude ambient light, thereby appearing darker. In the previous image, the left side is rendered without Ambient Occlusion and on the right, rendered with it. Notice how the edges around the steps are darkened.

The racing game, *Circuit Superstars* by Original Fire Games, is a game made with Unity that includes some of the newer URP features, such as SSAO, to ground the cars and models in the environment and add depth to the visuals.

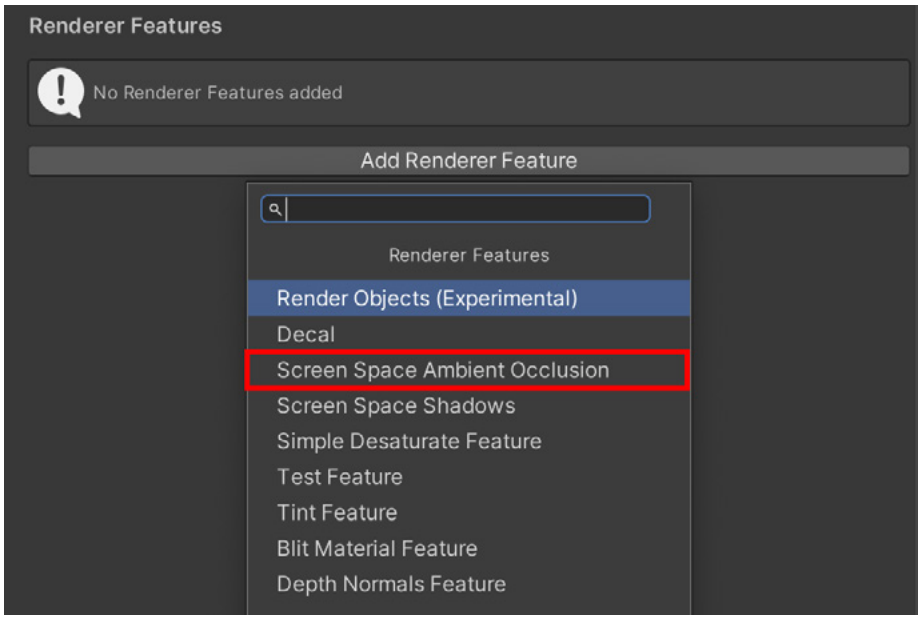
URP implements the real-time Screen Space Ambient Occlusion (SSAO) effect as a Renderer Feature. The pass code it uses can be viewed here.

> **Note:** The SSAO effect is a Renderer Feature and works independently from the post-processing effects in URP. This effect does not depend on or interact with Volumes.

To see it in action, open **Scenes > Ambient Occlusion > Ambient Occlusion**. This scene is a low polygon city environment available as a free asset on the Unity Asset Store.

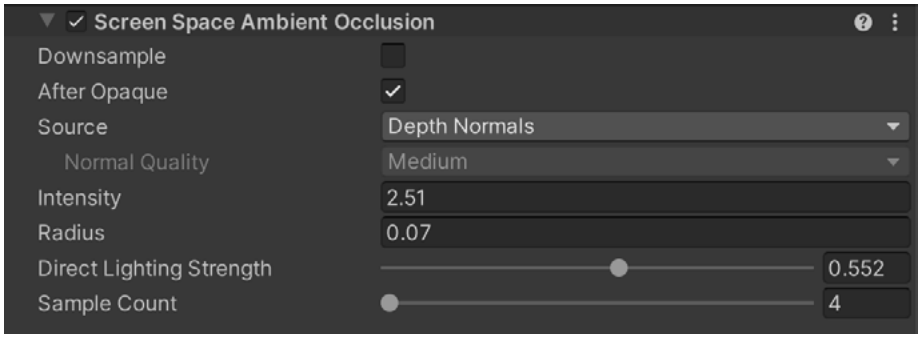
The scene uses the URP Asset named **Ambient_Occlusion_URP_Settings**. This is loaded automatically when you open the scene via the `AutoLoadPipelineAsset` script attached to **Scene > Main Camera**. The URP Asset uses the **Ambient_Occlusion_URP_Settings_Renderer**.

To add SSAO to your scene, view the Universal Renderer Data asset in the Inspector, and click on **Add Renderer Feature**. In the options drop-down menu, select **Screen Space Ambient Occlusion**.



Adding a SSAO Renderer Feature

Having added a SSAO Renderer Feature, you can now control the result via the Inspector. Let's look at the available properties.



SSAO options

Downsample

Selecting this halves the resolution of the processing in both the X and Y directions. Since this effectively reduces the number of pixels to process by 75%, it also reduces GPU load significantly but results in an effect with fewer details.

After Opaque

This option affects the look of the final render, but it comes with performance implications:

— **If disabled**: SSAO has a Depth or Depth Normals prepass (see the Source option below). The SSAO is then calculated after them and applied in the DrawOpaques pass when doing the lighting calculations. It gives a better-looking Ambient Occlusion, and the user can control the Direct Lighting Strength value for SSAO, but it has a negative impact on performance.

— **If enabled**: SSAO requires a Depth Normals if After Opaque is selected. If Depth is selected, then it either gets the depth from Depth prepass, if that was made, or a CopyDepth pass done after rendering opaques. The SSAO is then added on top of everything after the DrawOpaques pass, instead of being part of the lighting calculations. The benefit here is that a prepass can be skipped, which can help performance.

> **Note**: You want to also be able to render Depth + Normals in the Render Opaque pass so you can fully skip any prepass with that option enabled to save performance.

Source

This option selects the source of the normal vector values. The SSAO Renderer Feature uses normal vectors for calculating how exposed each point on a surface is to ambient lighting.

Available choices for Source:

— **Depth Normals**: SSAO uses the normal texture generated by the DepthNormals pass. This option lets Unity make use of a more accurate normal texture.

— **Depth**: SSAO reconstructs the normal vectors using the depth texture instead. Use this option only if you want to avoid using the DepthNormals pass block in your custom shaders. Selecting this option enables the **Normal Quality** property.

When switching between these two options, there might be a variation in performance, which depends on the target platform and the application. In a wide range of applications the difference in performance is small. In most cases, Depth Normals produces a better visual look.

Normal Quality

This is active when the **Source** property is set to **Depth**.

The options in this property (Low, Medium, and High) determine the number of samples of the depth texture that Unity takes when reconstructing the normal vector from the depth texture. The number of samples per quality level are:

— Low: 1

— Medium: 5

— High: 9

The performance impact is regarded as medium.

Intensity

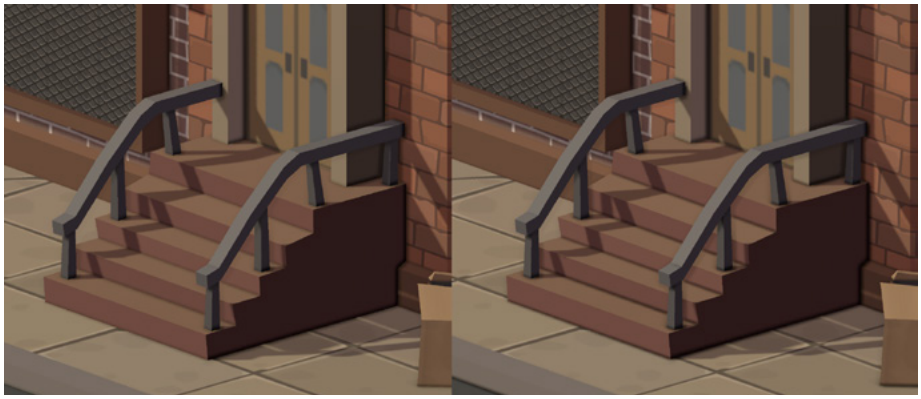
This controls the strength of the darkening.

Radius

This property controls how many samples of the normal texture are taken around the current pixel. Larger values have a significant impact on performance, so keep them as low as possible. The radius value is scaled based on the distance from the camera to the object that is being rendered at the target pixel.

Direct Lighting Strength

This property is dependent on the **After Opaque** option being disabled since it relies on being handled when lighting calculations are being done. It affects the strength of Ambient Occlusion where direct light hits.



Two variations of Direct Lighting Strength: 0.2 (left) and 0.9 (right)

Sample Count

For each pixel, the SSAO Renderer Feature takes the number of samples within the specified radius to calculate the Ambient Occlusion value. Increasing this value makes the effect smoother and more detailed, but reduces the performance. Doubling the Sample Count value doubles the computational load on the GPU.

SSAO is another great example of the flexibility of URP. The number of problems that can be addressed using Renderer Features is limited only by your imagination.

More resources

— YouTube tutorial from UGuruz

— Ambient Occlusion documentation

— Assets used in recipe (thanks to Marcelo Barrio)

DECALS

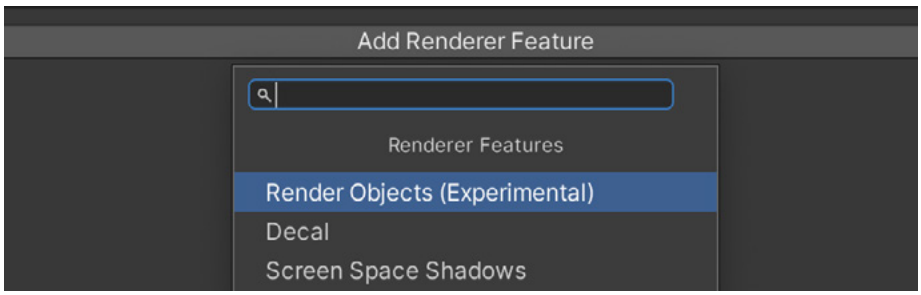
Decals are a great way to add overlays to a surface. They are often used to add visuals such as bullet holes or tire treads to the game environment as the player interacts with the scene. As you can see from the steps in the following image, the decal wraps around a mesh. You'll find the scene file and assets for this recipe in the folder **Scenes > Decals**.

Decals added to a simple scene

Decals are rendered in a scene using a Renderer Feature. If you look at **Decals_URP_Settings_Renderer** from the recipe folder you'll see that the Decals Renderer Feature is added. As usual, the `AutoLoadPipelineAsset.cs` script attached to the Main Camera ensures the correct pipeline asset is used when you load the scene.

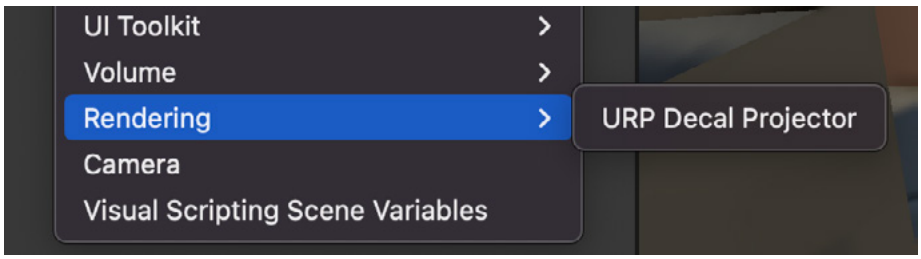
One of many use cases for decals is to project a blob shadow onto a 3D surface, like the character Milo from the game *Tinykin*, made with Unity by Splashteam.

To add decals to your custom scene, select the Universal Renderer Data asset currently being used by the Player for rendering, and in the Inspector, choose **Decal** from the Add Renderer Feature dropdown.



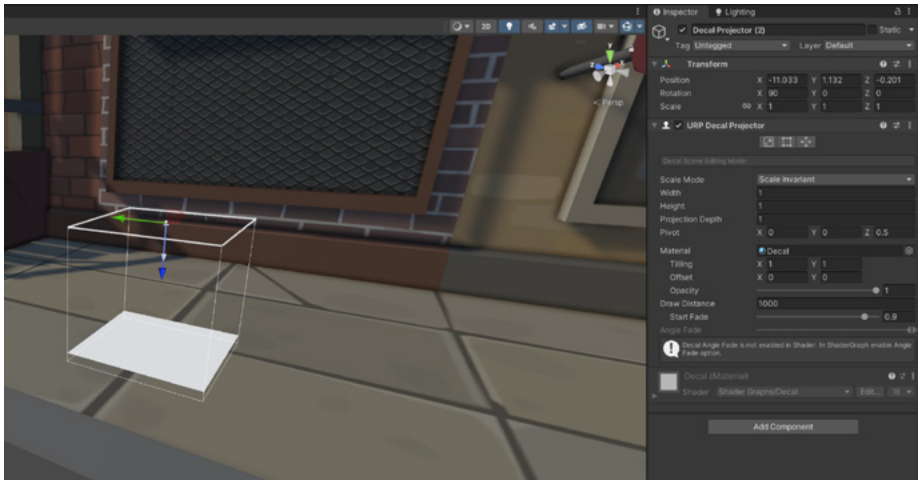
The Decal option in the Add Renderer dropdown

To add a decal to a scene when working in the Editor, right-click the Hierarchy window, and select **Create > Rendering > URP Decal Projector**.



Creating a URP Decal Projector

Position and orient a URP Decal Projector in the Editor as you usually would. A Decal Projector uses orthographic projection, so the size of a decal cast on a surface is unaffected by the distance of the projector from the surface. Initially, a new Decal Projector will display as a white block. In addition to the axis arrows, you'll see a white arrow indicating the direction of projection.



A new Decal Projector

URP Decal Projection properties

— **Scale Mode**: By default, the URP Decal Projection component has **Scale Mode** set to Scale Invariant. That means the size of the decal is determined solely by the **Width** and **Height** properties. Switching to **Inherit from Hierarchy** will combine the GameObject's Transform Scale with the Width and Height properties.

— **Width and Height**: These properties control the size of the decal.

— **Projection Depth**: Sets the depth of the projector bounding box; the projector projects decals along the local Z axis.

— **Pivot**: Sets the offset position of the center of the projector bounding box, relative to the origin of the root GameObject.

— **Materia**l: Sets the Material to project; the Material must use the Shader Graph/Decal (more details about this shortly).

— **Tiling and Offse**t: The tiling and offset values for the Decal Material along its UV axes.

— **Opacity**: Lets you specify the opacity value; a value of 0 makes the decal fully transparent, a value of 1 makes the decal as opaque, as defined by the Material.

— **Draw Distance**: The distance from the Camera to the decal at which this projector stops projecting the decal and URP no longer renders it.

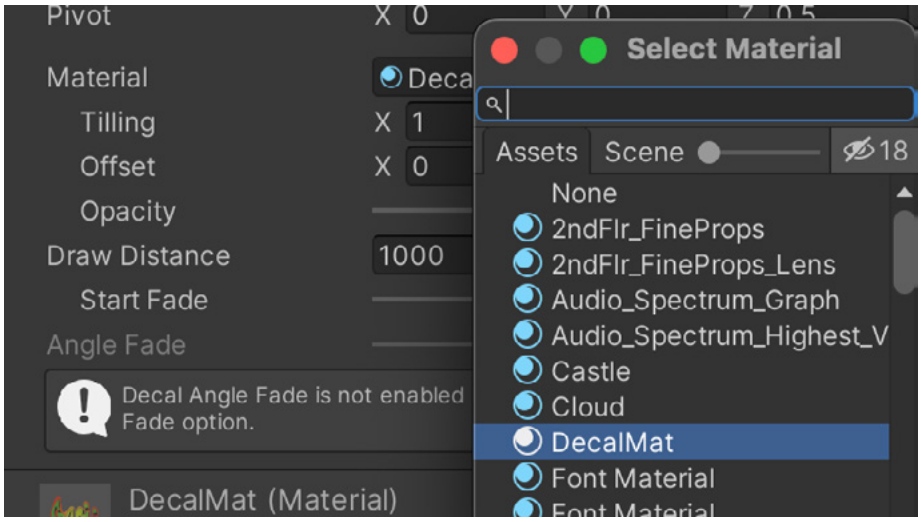
— **Start Fade**: Sets the distance (via a slider) from the Camera at which the projector begins to fade out the decal; values from 0 to 1 represent a fraction of the Draw Distance; with a value of 0.9, Unity starts fading the decal out at 90% of the Draw Distance and finishes fading it out at the Draw Distance.

— **Angle Fade**: Use the slider to set the fade out range of the decal based on the angle between the decal's backward direction and the vertex normal of the receiving surface.

Creating the Material

A Decal Projector must use a Material that uses the shader Shader Graph/Decal. This example uses the Material DecalMat found in the Scene folder. There is a Base Map assigned but no Normal Map; this is useful if you want the appearance of a lumpy surface for the decal.

The Material is assigned to the projector in the Inspector.



Assigning the URP Decal Projector Material

Adding a decal with code

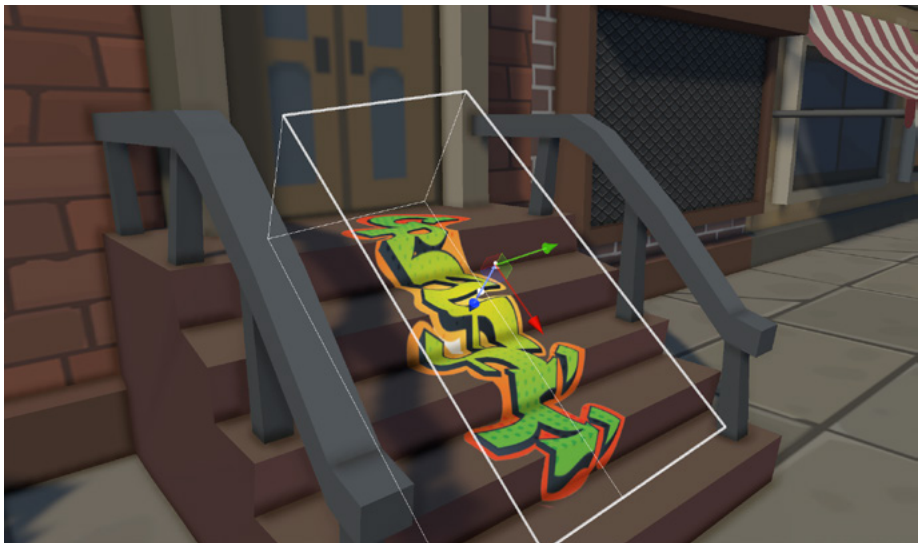
Although you can add a URP Decal Projector to your scene while developing in the Editor, it's more common to add them as a result of user interaction at runtime. You can create a Prefab to establish the Material, Width, and Height, although you can easily update this at runtime in code. This code example focuses on instantiation, positioning, and orientation only. The complete code to add a decal as a result of a mouse press on a Collider can be found in the AddDecal.cs script in the recipe folder.

```
void AddDecalProjector(Vector3 pos, Vector3 normal)
{
    GameObject decalProjectorObject =
Instantiate(decalProjectorPrefab);

    // Creates a new material instance for the DecalProjector
    //if you want individual Decal control over the material
    //DecalProjector decalProjectorComponent =
decalProjectorObject.GetComponent<DecalProjector>();
```

```
    //decalProjectorComponent.material = new
Material(decalProjectorComponent.material);

    //Move away from surface
    pos += normal * 0.5f;

    Quaternion up = Quaternion.AngleAxis(Random.Range(0, 360),
Vector3.left);
    Quaternion rot = Quaternion.LookRotation(-normal,
up.eulerAngles);

    decalProjectorObject.transform.SetPositionAndRotation(pos,
rot);
}
```

This function is called when there is a RaycastHit after a mouse-down event over a Collider. **pos** is the hit.point and **normal** the hit.normal. The Prefab called decalProjectorObject is instantiated. To get the position, you need to move the pos Vector3 away from the surface without exceeding the Projection Depth. This is achieved by moving the point along the normal. To orientate the decal, you first create a randomized up vector. To get the necessary rotation to align the decal to the surface and rotate a random amount around the normal, use the parameters inverse normal and the randomized up vector.

Decals have many uses in games, and the URP Decal Projector is a great tool in your toolbox.



A decal in the Scene View

More resources

— Decal Renderer documentation

— YouTube tutorial by Llam Academy

WATER

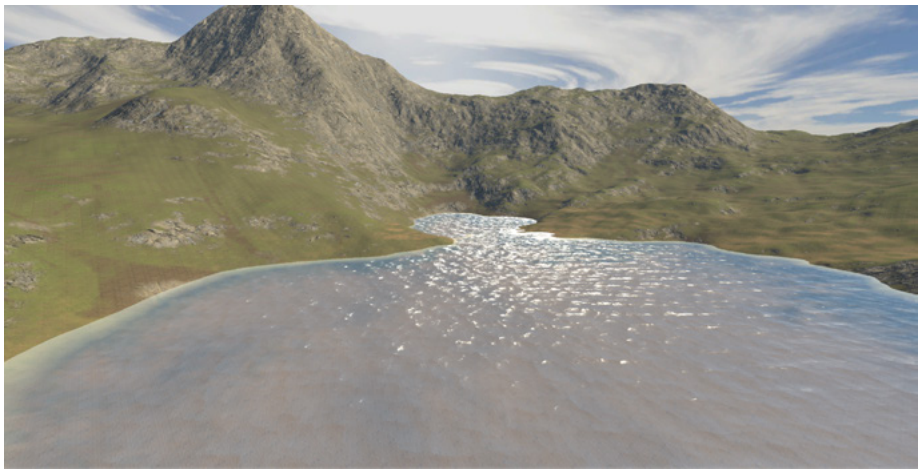
This recipe is for making a simple water shader. It's created in Shader Graph to make the steps more accessible to artists and designers.

The shader is built in three stages:

— Creating the water color

— Moving tiled normal maps to add wavelets to the surface

— Adding moving displacement to the vertex positions to create a swell effect

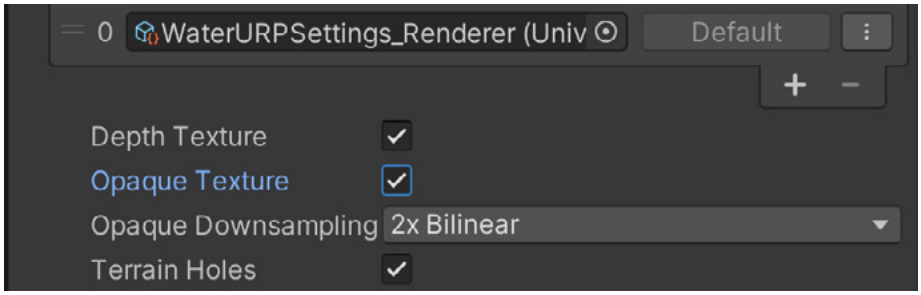


A still from a video showing a simple water shader in motion.

To view the final result, open the Water scene in the folder **Scenes > Water**. The final shader uses two subgraphs, DepthFade and TextureMovement; it's a good

Water and aquatic vegetation are two important visual elements for creating beautiful open environments in video games. This image is from the survival game *Len's island*, made with Unity by Flowstudio.

idea to look at them before you review the water shader. The Water scene uses the **WaterURPSettings Asset**, with the **Depth Texture** and **Opaque Texture** options enabled. Note that the Opaque Texture is only required if you add further effects not covered in this recipe, such as refraction.



Depth Texture and Opaque Texture selected in the WaterURPSettings asset

DepthFade subgraph

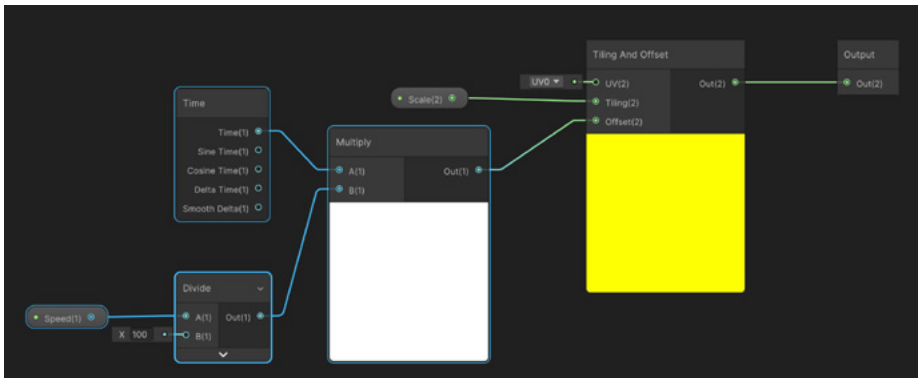
The DepthFade subgraph

The shallow and deep parts of the water each require their own color. The final color of the water will be a blend of these two colors, based on a **Depth** property. Depth is the distance between the surface of the water and the geometry below it. Since the water shader is set as transparent, opaque geometry will already be rendered, and because Depth Texture is selected for the URP Settings Asset, the current depth can be read.

A **Scene Depth** node with Sampling set to Eye mode gives the distance from the eye to the opaque geometry at the current pixel. The **Screen Position** node, with Raw selected as the mode of its output value, holds the information about rendering the current pixel of water. A **Split** node is used since you want the W component, which stores the distance from the eye to the current pixel of water.

Subtracting the water distance from the distance of the existing opaque geometry gives a guide to the depth of the water, albeit a ray from the eye position, not a ray directly down. Next, a **Divide** node controls where the edge between shallow and deep appears. The output from this subgraph should be between 0 and 1, so you'll use a **Saturate** node which acts as a specialized **Clamp** node by always restricting the output between 0 and 1.

TextureMovement subgraph

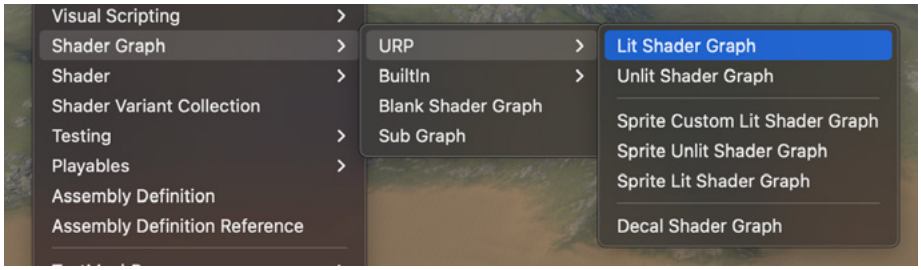


TextureMovement subgraph

The water shader has a number of moving textures that are handled using the TextureMovement subgraph. In this subgraph, a **Time** node is used as one input to a **Multiply** node. The input Speed is divided by 100 and forms the second input to the Multiply node. The output from the Multiply node acts as the Offset input to a **Tiling and Offset** node. The **Scale** property forms the Tiling input. Over time, this simple subgraph will update the UV used by a **Sample Texture 2D** node given a Speed and Scale input.

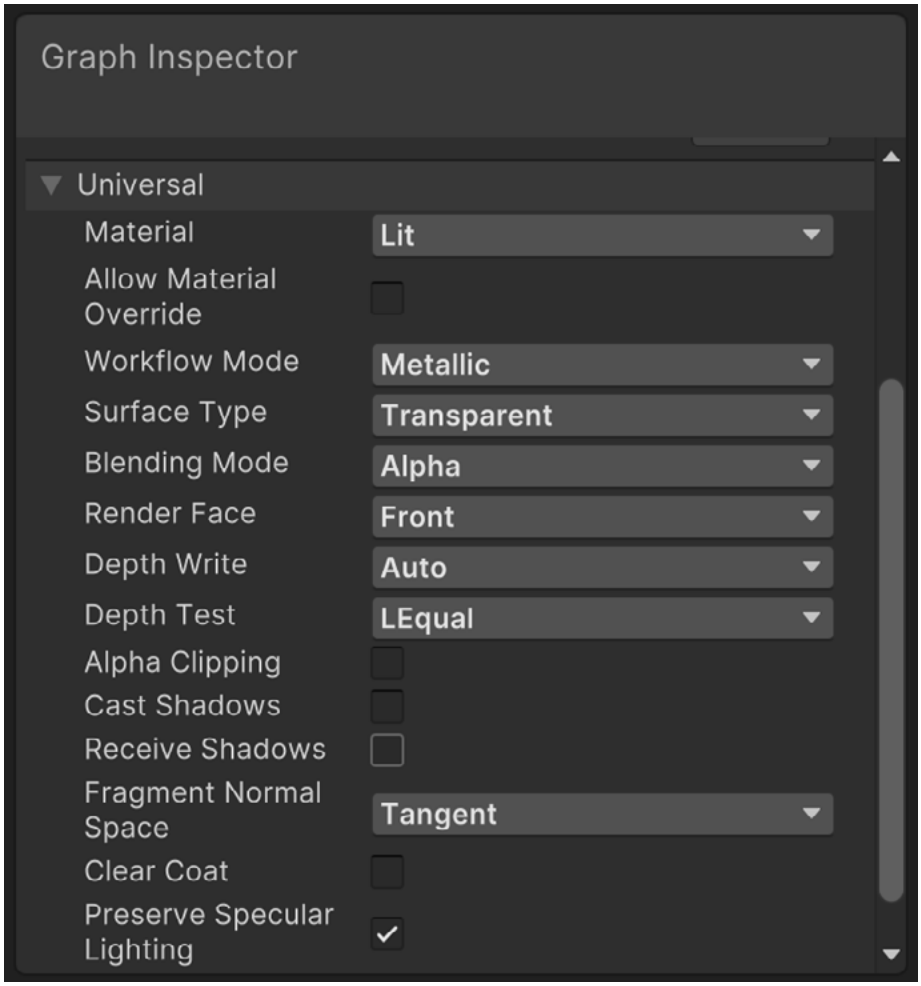
Water shader

Now it's time to create the water shader, based on a Lit Shader Graph, via **URP > Lit Shader Graph**.



Create > Shader Graph > URP > Lit Shader Graph

Next, you'll use the Graph Inspector to set the **Surface Type**.



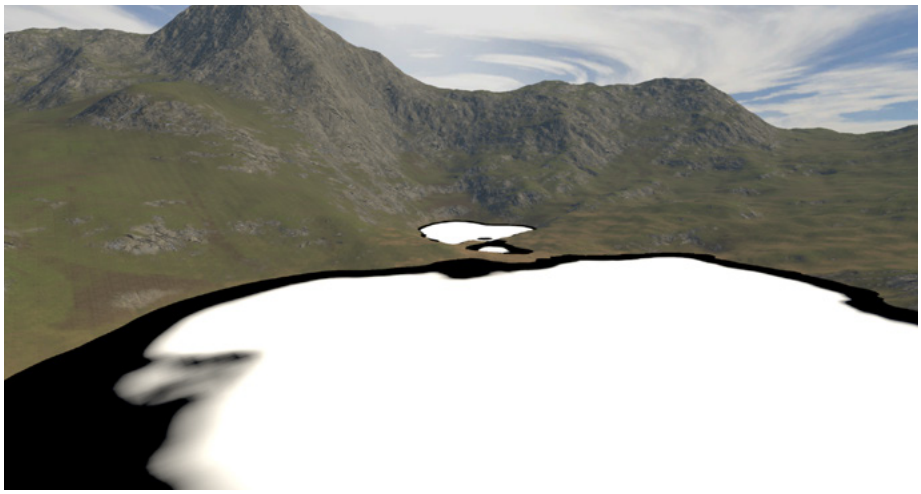
Setting the Surface Type

Now you'll edit the graph, starting with Color.

Color

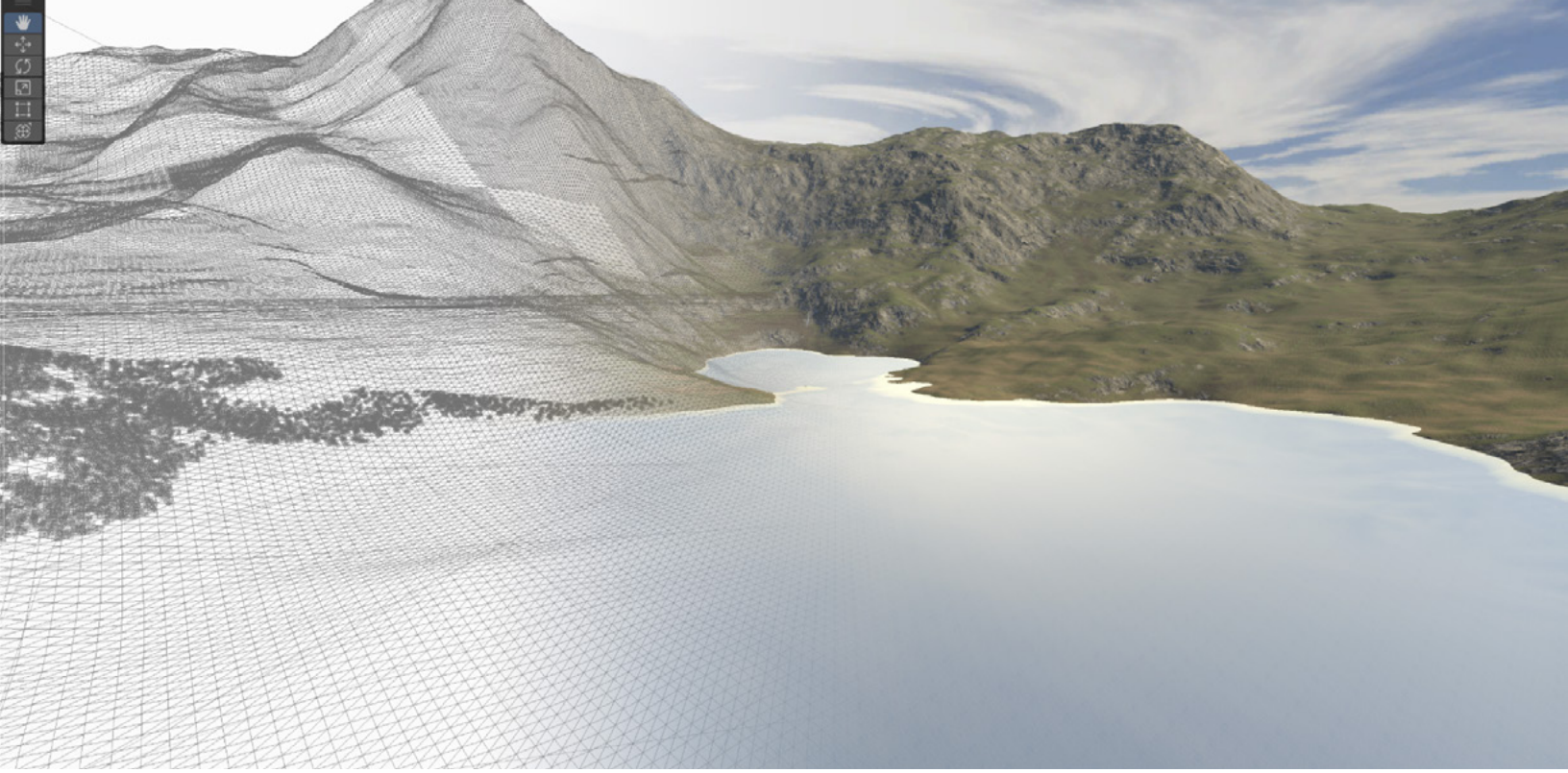
Color and Alpha

Color is handled by adding a DepthFade subgraph. The subgraph uses a float **Depth** property for control. If the output goes directly to the Base Color input of the Fragment shader, it results in the following image: shallow water is black and deeper water white. The higher the value of Depth, the more the black spreads. Black indicates 0 and white 1.



Plugging the output from DepthFade directly into **Fragment > Base Color**

Instead of linking the DepthFade directly to the Base Color input, it goes to a **Lerp** node. **ShallowWaterColor** is input A, replacing the black color, and **DeepWaterColor** is input B, replacing the white. When setting the alpha for these colors make sure the shallow water is more transparent. The Lerp output goes to **Fragment > Base Color**. For the Alpha, you'll use a Split node, linking the A output with **Fragment > Alpha**. This produces the result seen in the following image.

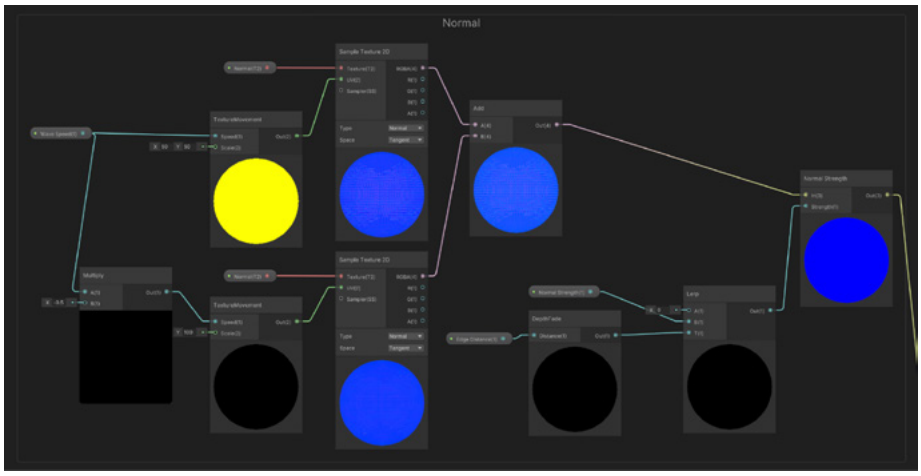


Dense mesh and colored water

The actual water is a plane, but to allow for vertex displacement, the mesh has many more vertices as the image shows.

This simple flat surface is a start but needs more work, namely, normal maps.

Normal maps

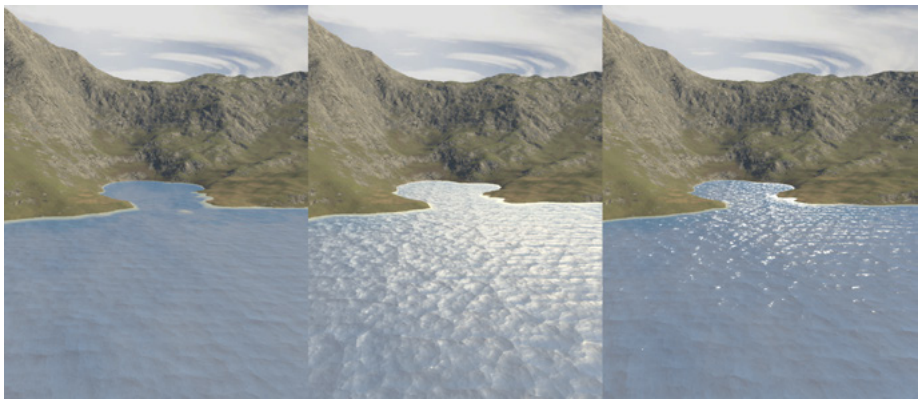


Controlling the **Fragment > Normal**

Normal maps add moving wavelets to the surface. The first input property is **Wave Speed**, which is used as the Speed input to a TextureMovement subgraph. **Scale** is hard set to 50,50, and, via a second TextureMovement node, **Speed** is preprocessed by a Multiply node to be minus half the Wave Speed property.

The next step in calculating the normal is to sample the Normal texture twice using the UV processed by the two TextureMovement subgraph nodes. We add the two normals together to get the combined effect of the two moving textures. The shader has a **Normal Strength** float property, which could be used as the Strength input to a Normal Strength node. But you want the wavelets to die back nearer to the edge. To control this, use the DepthFade subgraph node with the shader property **Edge Distance** controlling the spread. This is used as the T input to a Lerp node blending between 0 and Normal Strength. The output of this stage of the graph goes to Fragment > Normal.

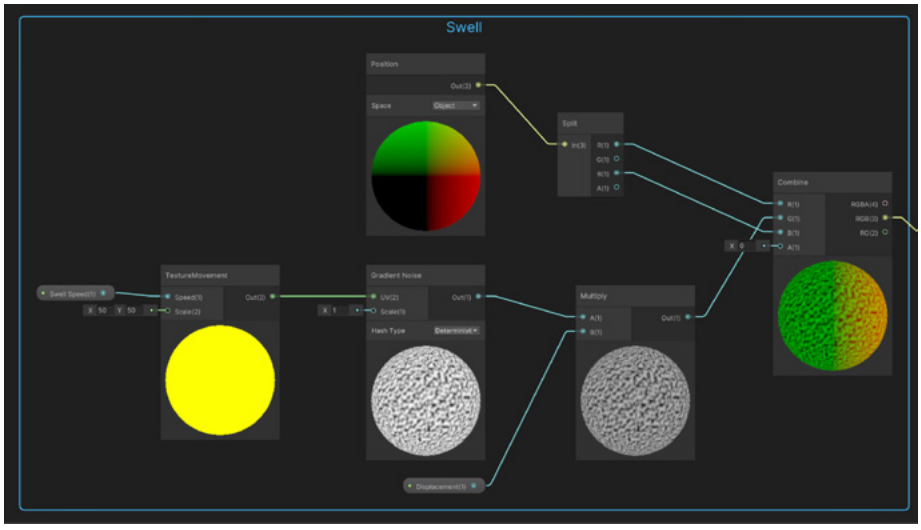
Now you have controllable wavelets whose reflective property can be tweaked by controlling the **Smoothness** of the Fragment using a simple float property. The following image shows the effect of changing the Smoothness value.



Applying different levels of smoothness to the wavelets, left to right: 0, 0.5, and 1

The next step is to enable vertex displacement to add motion to the water.

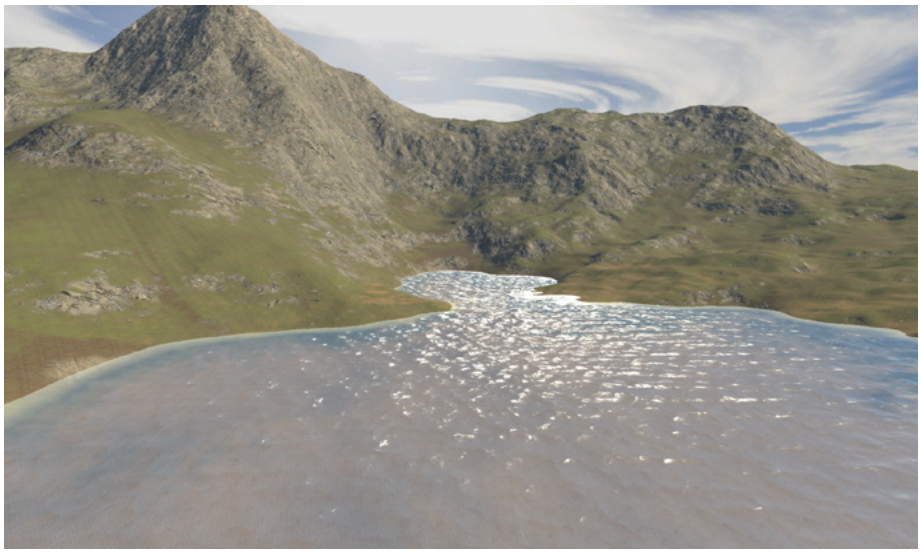
Swell



Controlling the swell using Gradient Noise

For this step, you'll use a TextureMovement subgraph node again. Speed is set using the shader float property **Swell Speed**, and Scale is hard set to 50,50. This acts as the UV input to a Gradient Noise node with Scale hard set to 1. You use a Multiply node to control this value using the shader float property **Displacement**. The purpose of these nodes is to set a Y value for vertex in object space. Notice the **Space** parameter of the Position node is set to Object. This links with a Split node and then a Combine node; Combine receives the R and B values directly from the Split node, with R being Position X and B being Position Z. The G value for Y comes from the Gradient Noise path. The RGB(3) output links to the Vertex > Position.

If you view the scene in Play mode you can see the swell moving through the water, especially at the edges.



The final result

While this recipe forms the basis of a simple water shader, you can enhance it using Caustic Reflections, Refraction, and Foam. See the links below for additional guidance.

More information

— Unity YouTube tutorial

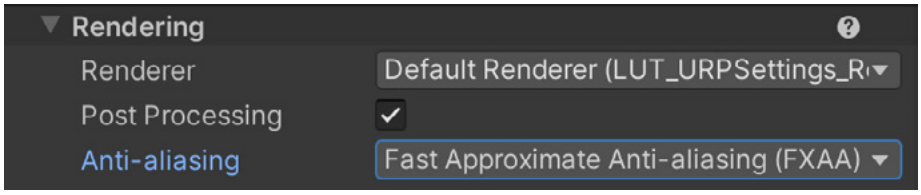
— Caustic reflections tutorial by Alan Zucconi

— Stylized water tutorial by Binary Lunar

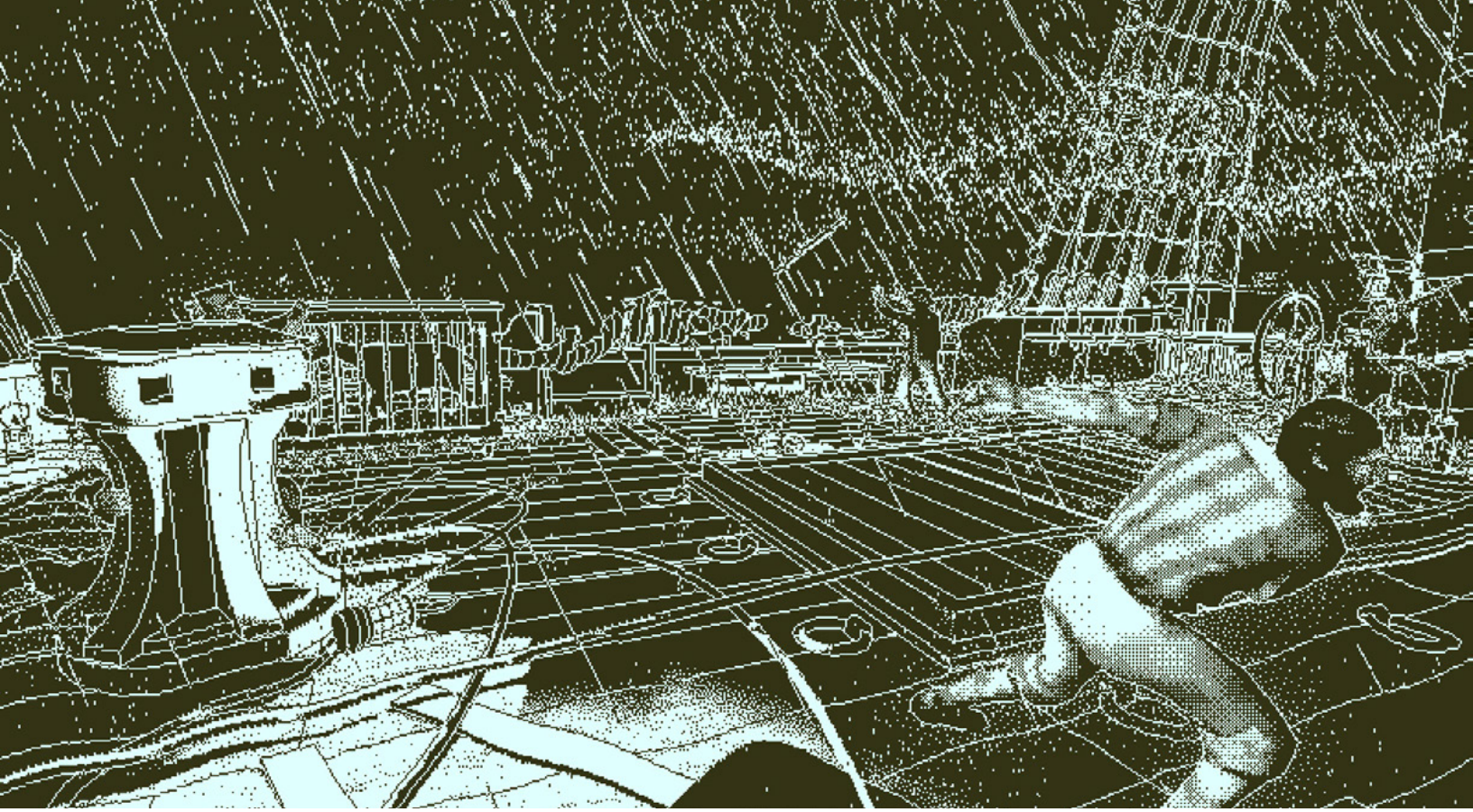
LUT FOR COLOR GRADING

Using Color Lookup to create grading effects

If you've yet to use the post-processing filters available with URP, you're in for a treat. This recipe involves using one filter, but the steps employed apply to all them. By default, a new URP scene has post-processing disabled, so make sure to enable it via the **Camera > Rendering** panel.

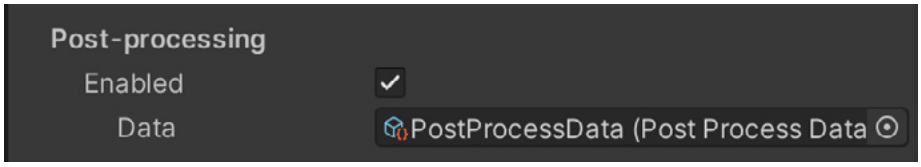


Select Post Processing in Camera > Rendering



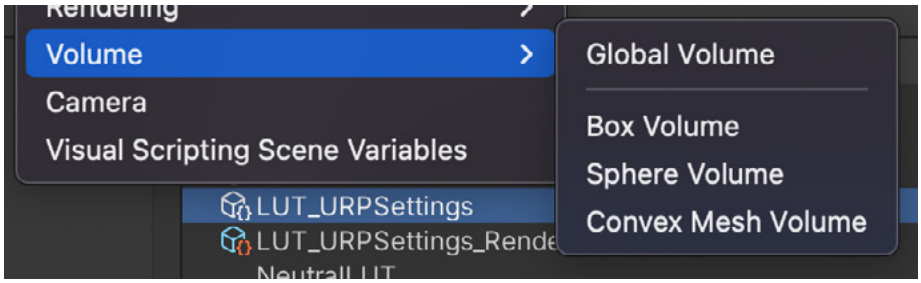
The mystery adventure FPS game *Return of the Obra Dinn*, made with Unity by Lucas Pope, achieves a unique look and feel thanks to its lo-fi art style and unique color palette that could be achieved following this recipe.

Additionally, you'll need to enable post-processing in the Universal Renderer Data asset.



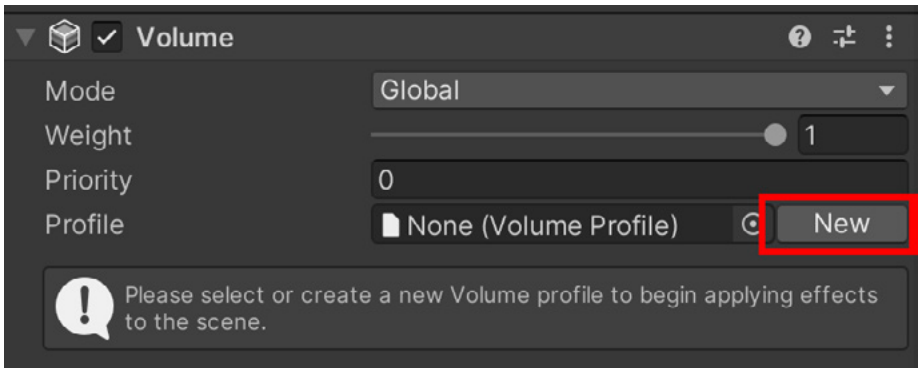
Selecting post-processing in the Universal Renderer Data asset

To apply the filter where the camera is located, add a Global Volume. Right-click the Hierarchy window, and select **Volume > Global Volume**.



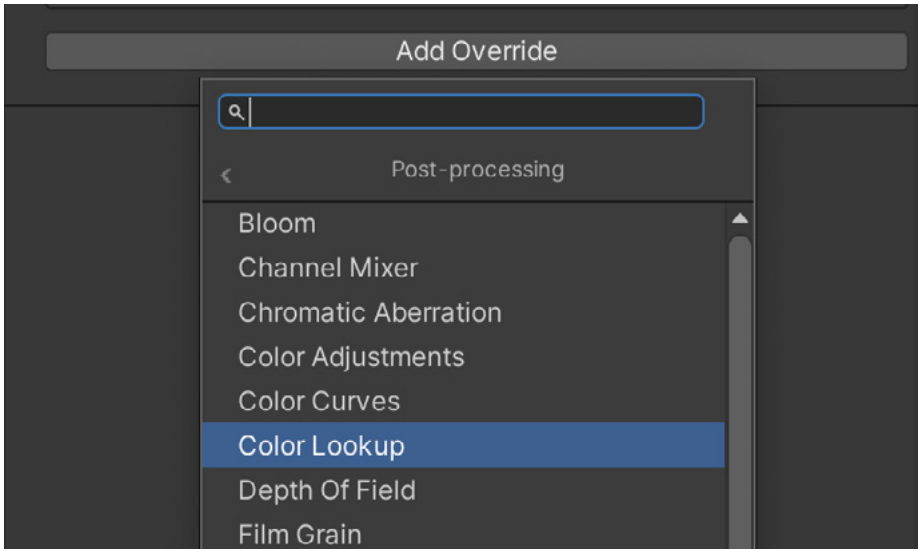
Creating a Global Volume

Select the new GameObject, and create a new Profile by clicking **New**.



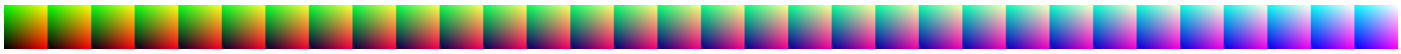
Creating a new Profile

Now you can add an override. Press the **Add Override** button, select post-processing, then choose **Color Lookup**.



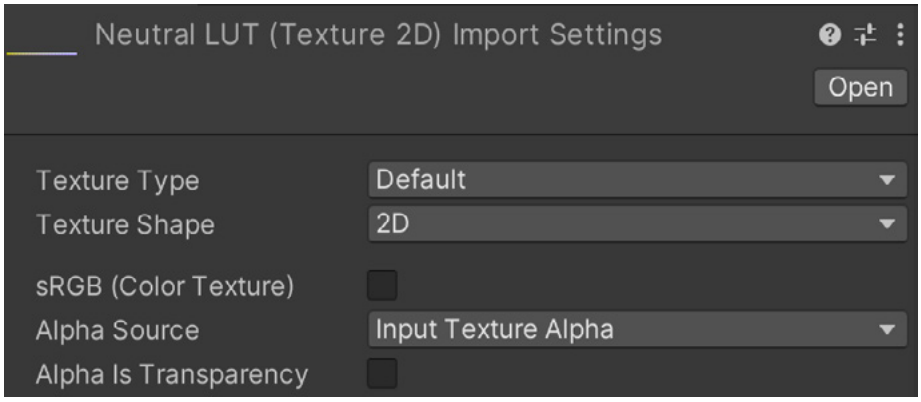
Adding a Color Lookup post-processing filter

Click the **All** button. Now you need a LUT (Lookup Table) image texture. This is a strip image that will be used by the filter to change the default rendered colors. You'll find the image file in **Scenes > LUT > NeutralLUT.png**, or download it using this link.



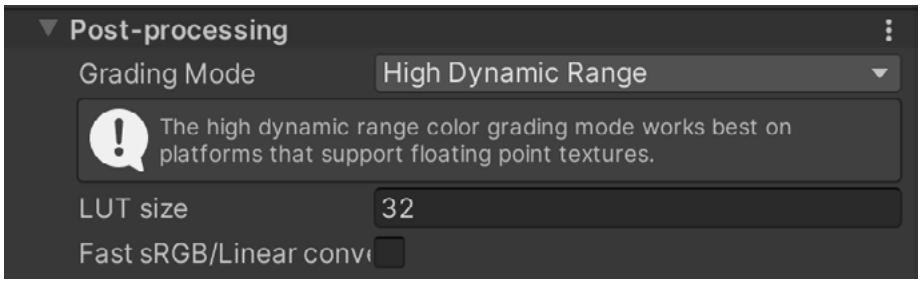
NeutralLUT.png

A LUT image must have **sRGB (Color Texture)** disabled, which you do by selecting the image and viewing the Inspector.



Disable sRGB (color Texture) for all LUT textures

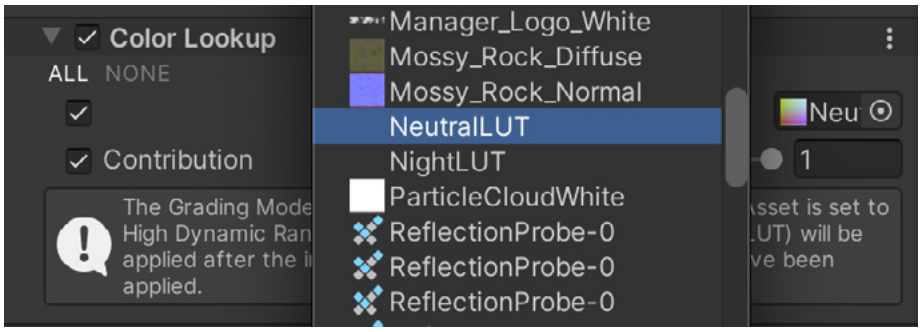
Count the blocks in the NeutralLUT image above, and you'll find there are 32 of them. Alternatively, you can use 16 blocks; whether you choose 32 or 16 blocks, ensure the settings for your URP Asset match your choice. If you choose 32, make sure the post-processing panel has **LUT size** set to **32**. Feel free to experiment with the **Grading Mode** option.



Setting the LUT size

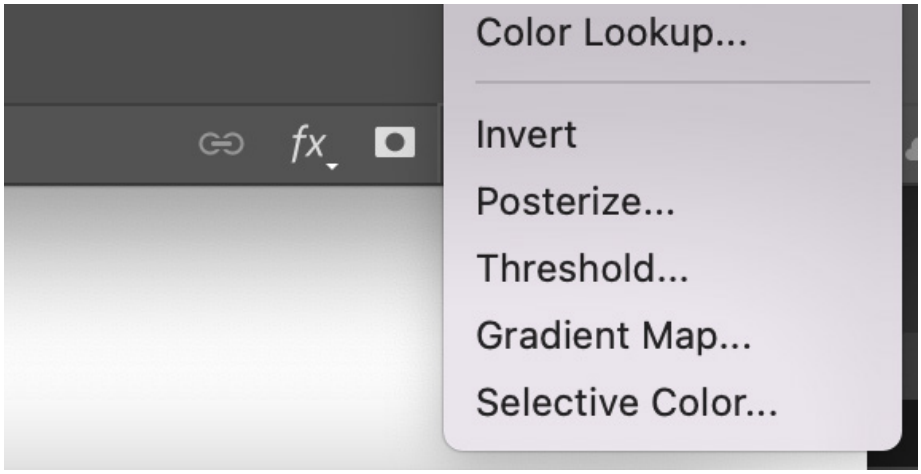
If you assign **NeutralLUT.png** as the Lookup Texture using the **Color Lookup** settings panel, you'll see no change to the rendered image. The filter uses the texture to set a new color. The code takes the current pixel color and uses this to find a texel on the LUT image. With a neutral LUT image, the texel color will be the same as the current pixel color.

The real magic occurs when you process the image you use as the Lookup Texture using a paint program, such as Photoshop or Krita (there's a link under More resources, at the end of this section, to a YouTube video explaining how to use Krita for color grading).



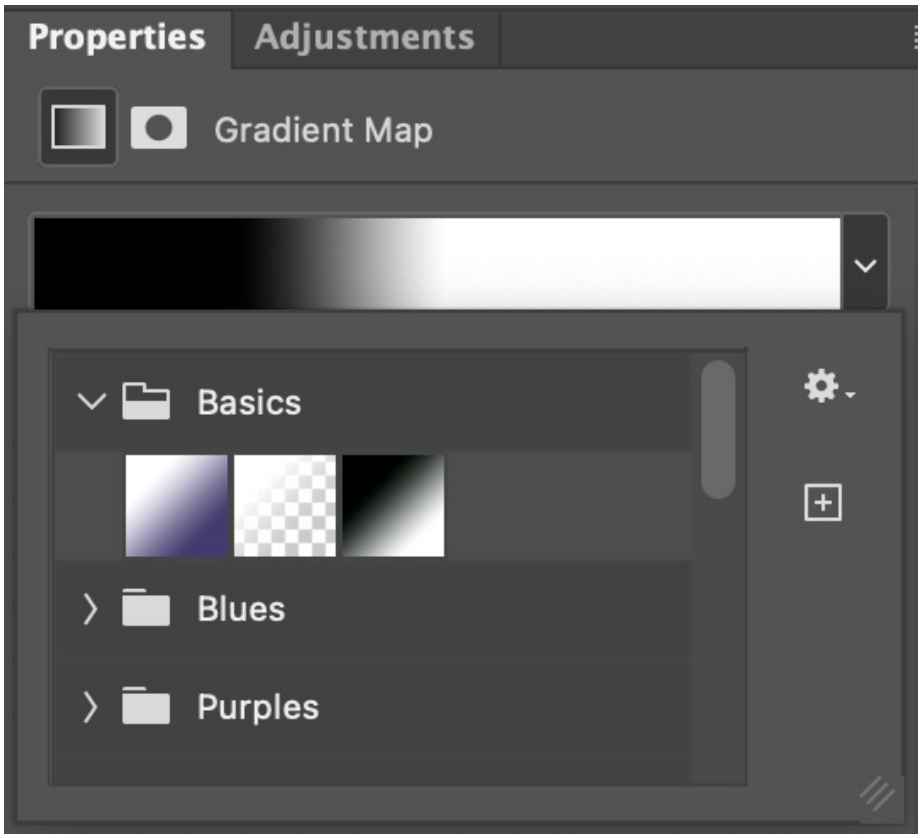
Assigning the Lookup Texture

Take a screen grab of your scene, and open it in Photoshop. At the bottom of the **Layers** panel, find the half black/half white circular button. Select it, and in the panel find **Gradient Map**. A new color adjustment layer is added.



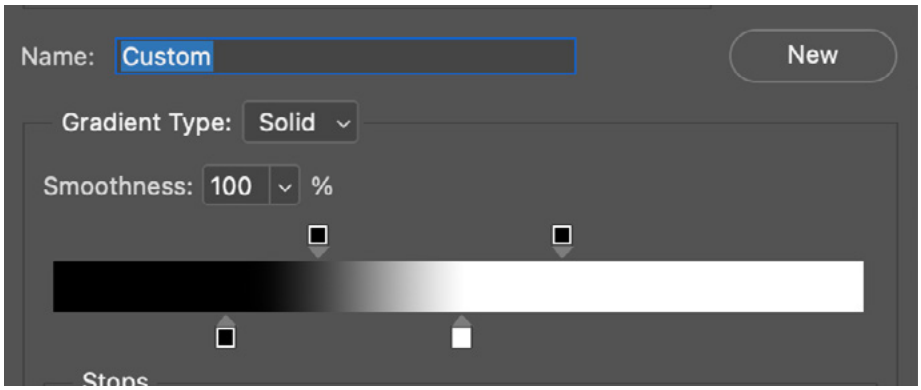
Creating a color adjustment layer

To create a color adjustment layer that results in a high-contrast black-and-white image, click the Gradient Map drop-down and select Basics, black and white.



Selecting a black and white gradient

To boost the contrast, click the gradient to open a new window. Use the stops to adjust the contrast.

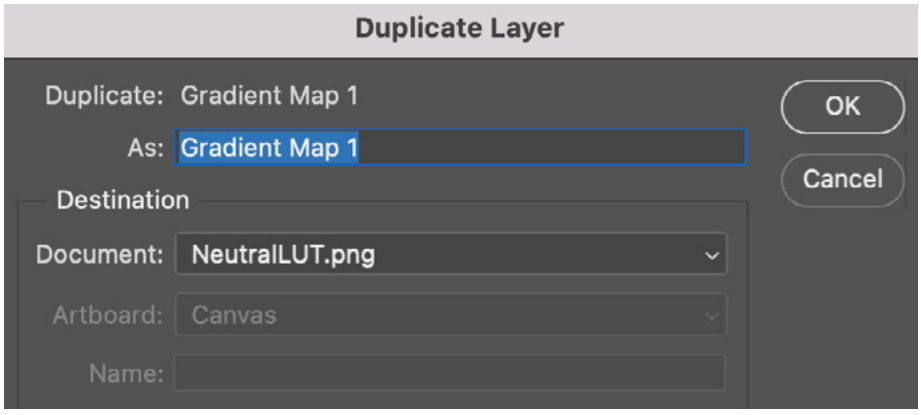


Changing the stops to boost the contrast

The screengrab should now look black and white.

The effect of the Gradient Map

Once you have the grading of your choice, you need to apply this layer to the NeutralLUT.png file. Open the file in Photoshop. Back in the screen grab, right-click the adjustment layer, and select **Duplicate Layer**. In the new panel, select NeutralLUT.png as the **Destination > Document**.

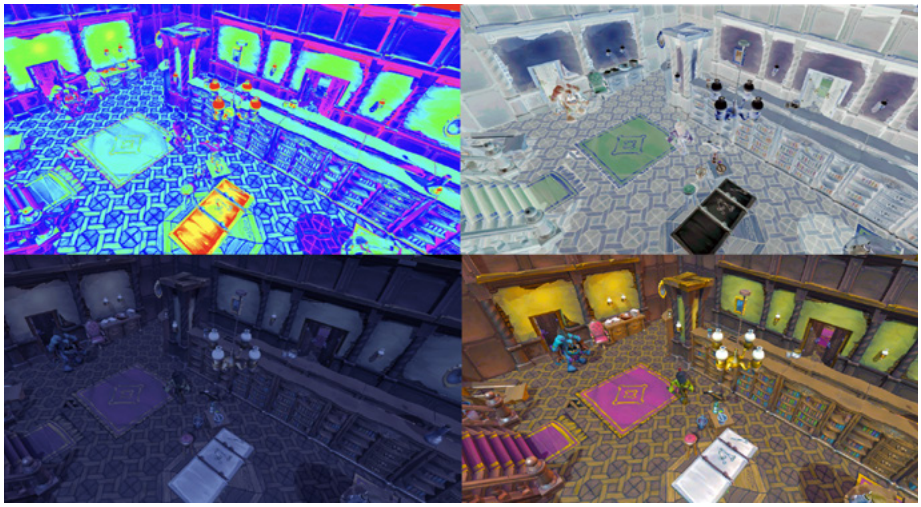


Duplicating the adjustment layer

Now the texture looks like this:

B&WLUT.png

Save it, and drag it to your project's Assets folder. Make sure to disable sRGB (Color Texture) using the Inspector panel. The last step is to assign the new LUT texture as the Lookup Texture for the Color Lookup filter.



Using various LUT textures

Using LUT Textures is an efficient way to create dramatic color grading, and this approach can be useful in many games.

More resources

— Documentation for post-processing in URP

— YouTube tutorial by PHLEARN

— YouTube tutorial by GDQuest

LIGHTING

Lighting with URP is similar to using the Built-In Render Pipeline. The main difference is where to find the settings.

This section covers related recipes for real-time lighting and shadows, including baked and mixed lighting using the GPU Progressive Lightmapper, Light Probes, and Reflection Probes. You'll pick up enough instruction for a five-course meal.

Before starting, here are some pointers to keep in mind about shaders and color space.

Shaders

When using lighting in URP, you have a choice of shaders. Generally, you wouldn't mix Lit, which uses a Physically Based Rendering (PBR) model, with Simple Lit, which uses a Blinn-Phong model. The following table provides descriptions for URP shaders.

The choice between a Lit Shader and Simple Lit Shader is largely an artistic decision. It's 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.

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.

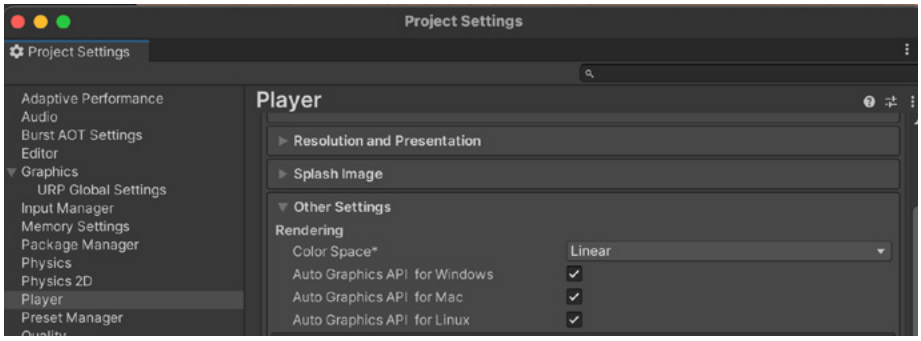
The Unity and URP-made game *LEGO® Bricktales* by Clockstone immerses players in the world of LEGO, where great lighting plays a huge role in creating its atmosphere and realism of the blocks.

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.

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.

Color Space

When setting up lighting for a URP project, you have a choice to work in linear or gamma color space. The former is the default and strongly recommended. This is set using **Edit > Project Settings... > Player > Color Space**. A project created with the URP 3D template is set to Linear by default.

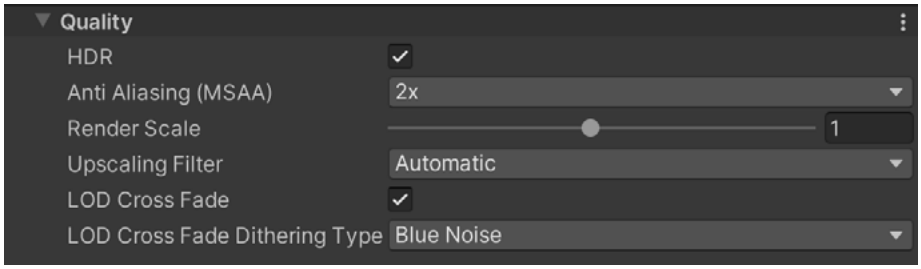


Setting the Color Space

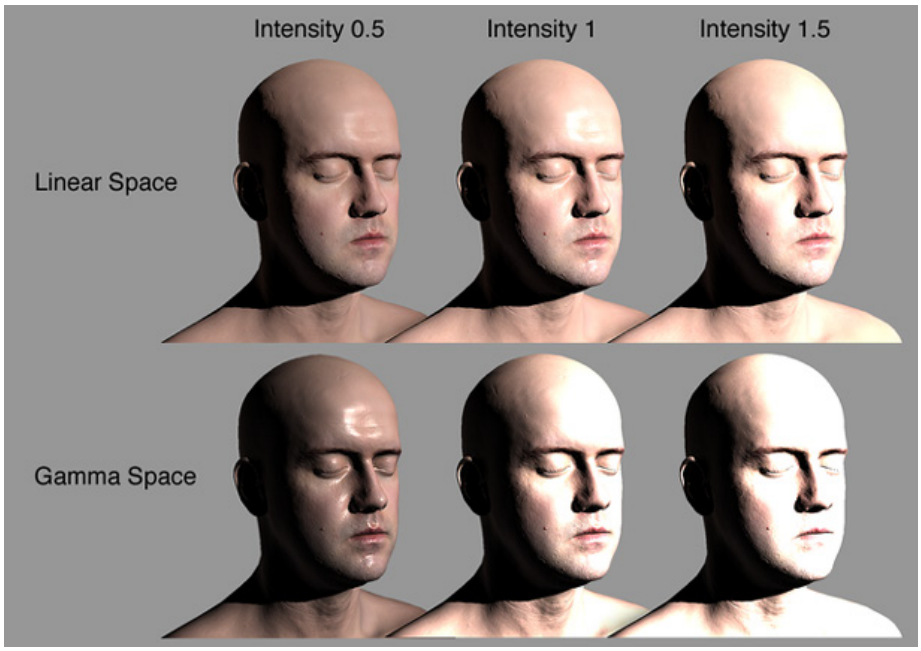
The two different color spaces relate to how the human eye responds to light. The human eye is more sensitive to bright light, so it doesn't respond to light intensity linearly. Gamma space shifts the linear values to better represent what the human eye can distinguish, however this is not mathematically accurate, so as hardware has improved and PBR lighting models are preferred, there has been a shift to using a linear color space.

There are situations where you need to use gamma, such as if the target hardware is old and doesn't support sRGB formats, but in most cases it is fully supported. There can be artistic reasons to switch to gamma, such as when the project is set up for LDR rather than HDR and uses the Simple Lit lighting model or Unlit. In such a case, switching to a gamma color space could be beneficial for achieving the target art style.

LDR or HDR is set using the URP Settings asset, under the Quality section.



Setting URP to use HDR



Comparison of linear and gamma space lighting with different intensities

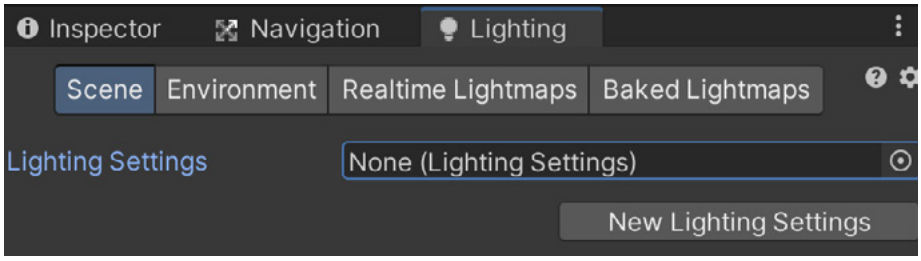
Further reading on gamma and linear color space and the workflows for using gamma and linear textures can be found in the documentation.

The diorama scene

Real-time Global Illumination and mixed lighting

The screenshots used for most of this recipe are from the Unity project *FPS Sample: The Inspection* that you can download here. This sample was made for Unity 2020 LTS, but the lighting principles still apply to Unity 2022 LTS.

If you open the **scene Scenes > Small_Indirect**, you'll see a diorama featuring a cave, mechanical arm, and a robot.

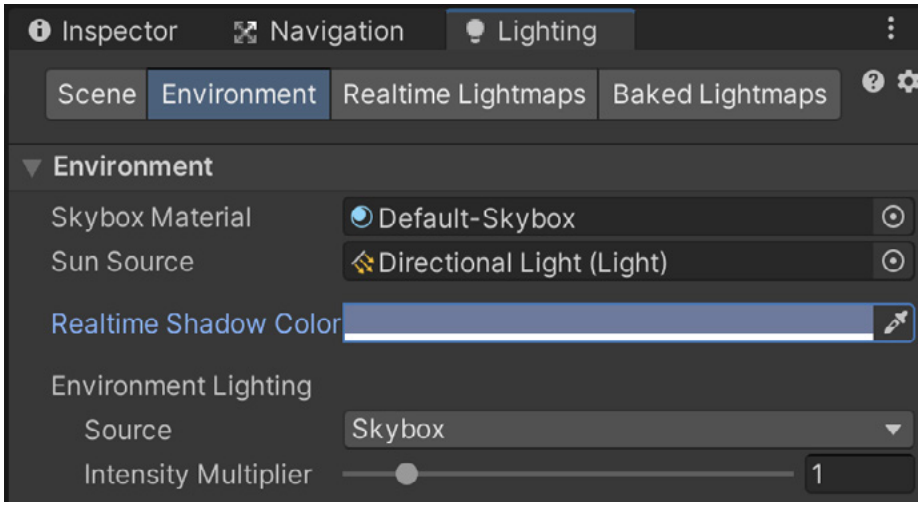


Creating a Lighting Settings Asset

The first step to lighting a new scene for URP is to create a new Lighting Settings Asset. Open **Window > Rendering > Lighting**, click the **Scene** tab, then **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.

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**.

Light from weapon on cave wall



Setting the Sun Source

The Main Light, named the Sun in the diorama scene, is set as light mode Mixed, meaning it contributes to both real-time lighting and baked lighting. If you have dynamic objects in the scene, then you need at least one real-time light illuminating them so the shadows they cast are updated as they move. The robot's weapon also features a light, set as Realtime. It is most noticeable when inside the cave. Play the scene, use the WASD keys to move the robot, and you'll see how it casts shadows from the props in the cave.

With URP, lighting settings are adjusted from three places:

— **Window > Rendering > Lighting:** This panel allows you to set lightmapping and environment settings, and view real-time and baked lightmaps.

— **Light component in the Inspector view:** The Light component attached to the GameObject that acts as light. See the following table for details.

— **URP Asset Inspector:** This is the principal place for shadow settings. Lighting in URP relies heavily on the settings chosen in this Inspector.

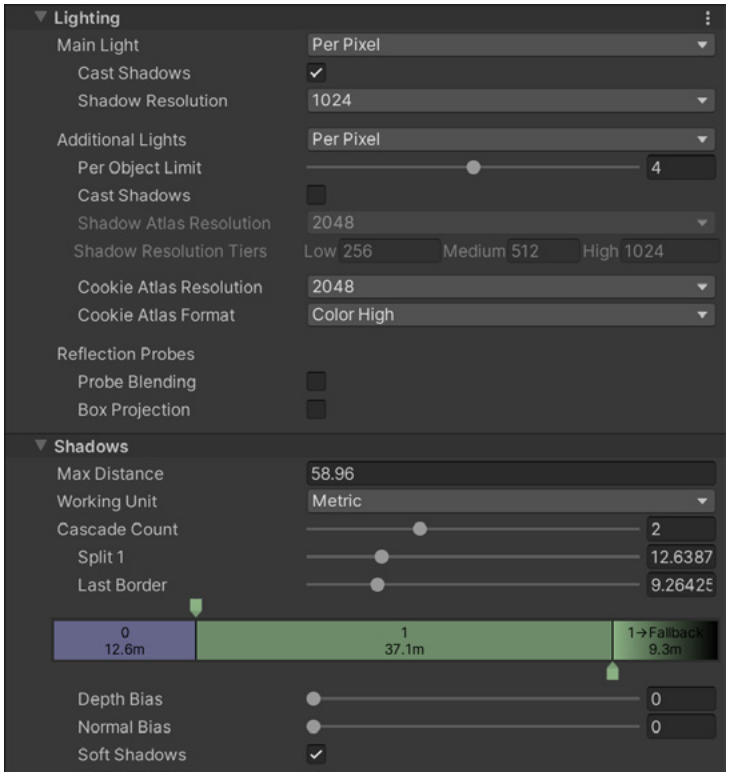
This table lists the differences between the URP and Built-In Render Pipeline Light Inspectors.

URP Light Inspector properties	Description
General	
Type	Spot, Directional, Point, or Area
Mode	Baked, Mixed, or Realtime
Shape	
Spot	You can control both the inner and outer cone angles for Spot lights.
Area	This is used to control the shape of an Area light.

Emission	
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.
Color	Use the color picker to set the color of the emitted light.
Intensity	Set the brightness of the light. The default value for a Directional light is 0.5. The default value for a Point, Spot, or Area (Rectangle or Disc) light is 1.
Indirect Multiplier	Use this value to vary the intensity of indirect light. If you set the Indirect Multiplier to a value lower than 1, the bounced light becomes dimmer with every bounce. A value higher than 1 makes light brighter with each bounce. This is useful, for example, when a dark surface in shadow (such as the interior of a cave) needs to be brighter to make detail visible.
Range	Control how far from the GameObject position the light affects the render.
Rendering	
Render Mode	Auto – decided at runtime by light proximity to camera Important – always per-pixel quality Not important – always faster per-vertex quality
Culling Mask	Used to control which layers are affected by the light
Shadows	
Shadow type	No shadows, Soft shadows, or Hard shadows.
Baked Shadow Radius	If Type is set to Point or Spot and Shadow Type is set to Soft Shadows, this property adds some artificial softening to the edges of shadows and gives them a more natural look.
Light Cookie	
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.

SHADOWS

Shadow settings are set using a Renderer Data object and a URP Asset when using URP. You can use these assets to define the fidelity of your shadows.



The URP Asset

Syberia: The World Before by Microids is another example of a game made with Unity that makes great use of lights and shadows to recreate beautiful architecture and streets in this story-driven console and PC game.

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.

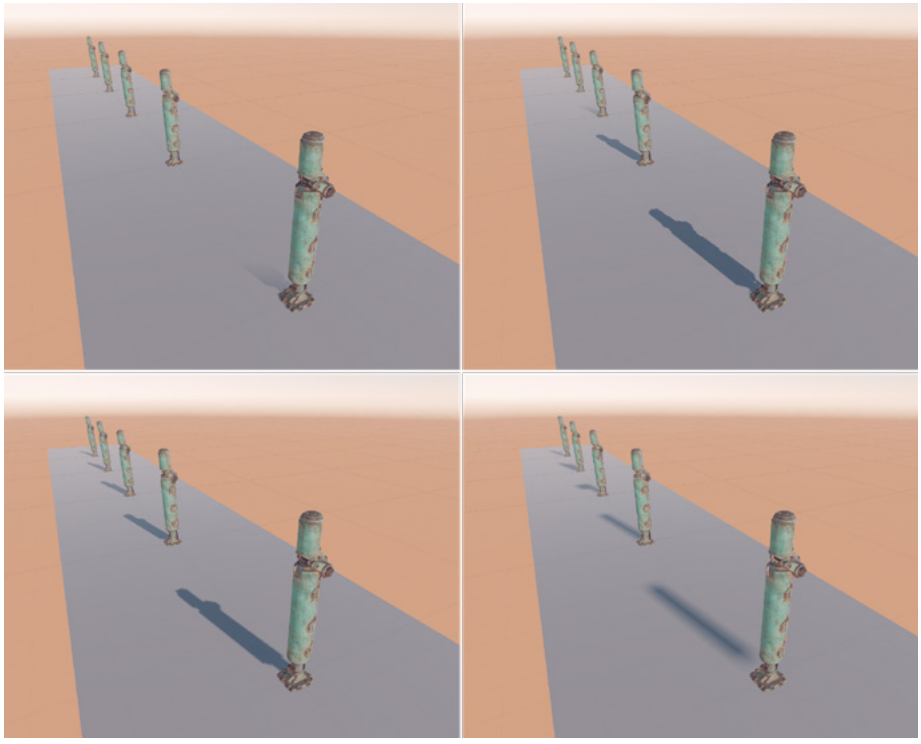
If you've worked with shadows in Unity before, you know that real-time shadows require rendering a shadow map that contains the depth of objects from the perspective of the light. The higher the resolution of this shadow map, the greater the visual fidelity – though both more processing power and memory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

1. The number of Shadow Casters rendered in the shadow map – number for the Main Light depends on the Shadow Distance (far plane of shadow frustum)
2. Shadow Receivers that are visible (you have to encompass them all)
3. Shadow Cascades splits
4. 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 that 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, 1024 in the bottom-left image, 2048 in the top-right image, and 4096 in the bottom right image. Middle image 1024

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.

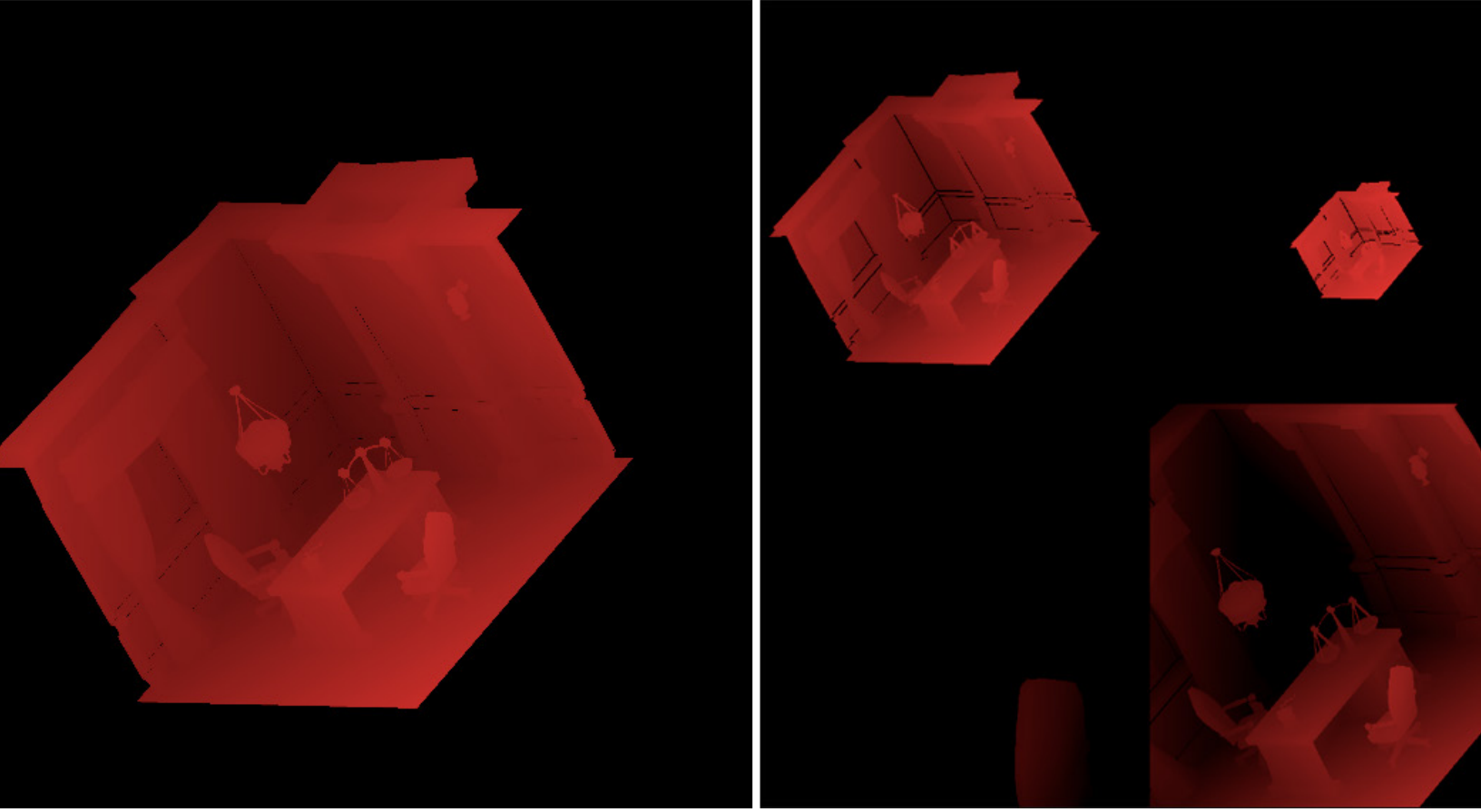
> **Note:** URP only supports **Stable Fit** Shadow Projection, which relies on the user to set up the Max Distance. The Built-in Render Pipeline supports both Stable Fit and Close Fit for the Shadow Projection property. In the latter mode, the bottom-left and -right images would have the same quality, 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's convenient to decrease Shadow Resolution, thereby devoting more of the shadow map to shadows closer to the Camera. Shadow Cascades can help with this.

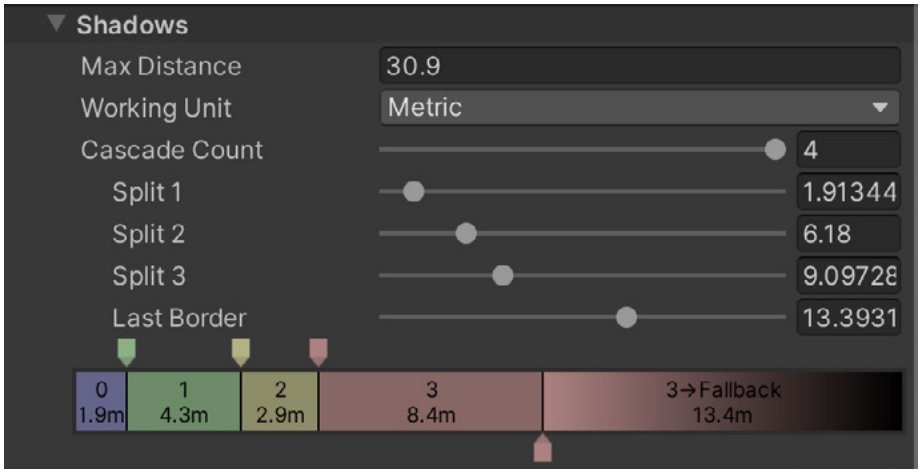
The images below show the shadow map of the scene with the chair and desk in the haunted room, **Scenes > Renderer Features Stencil > SmallRoom-Stencil**, that you saw in the first recipe. The cascade count is 1 in the image to the left. The map takes up the whole area. In the image to the right, the 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.



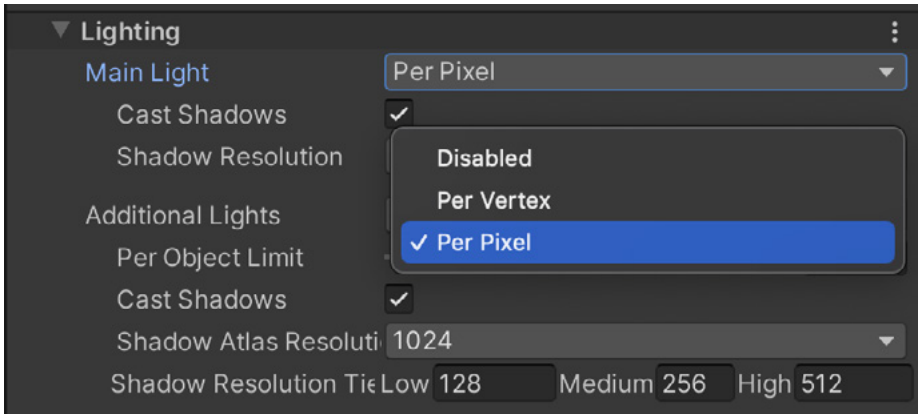
Shadow map when cascade count is set to 1 (left image) and 4 (right image)

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 the image that follows). 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 the distance of the last Shadow Caster, at most.



Adjusting the range of a Shadow Cascade

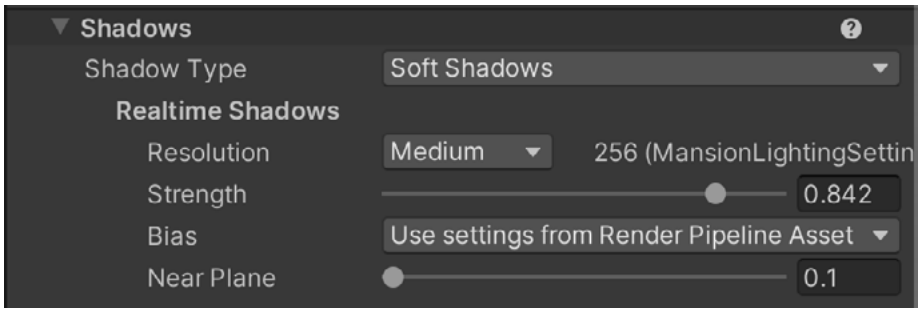
Additional Light shadows



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 image above), 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.

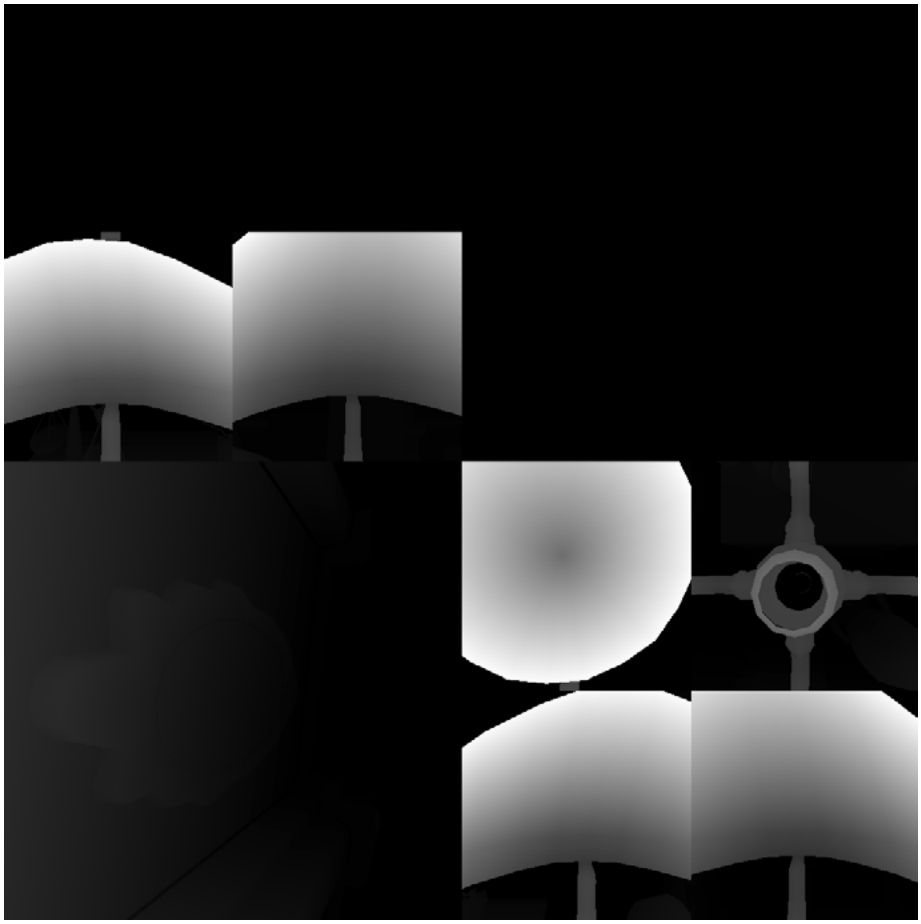


Shadows group in the Light Inspector

In the haunted room, one of the projects explored in the e-book *Introduction to the Universal Render Pipeline for advanced Unity creators*, 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



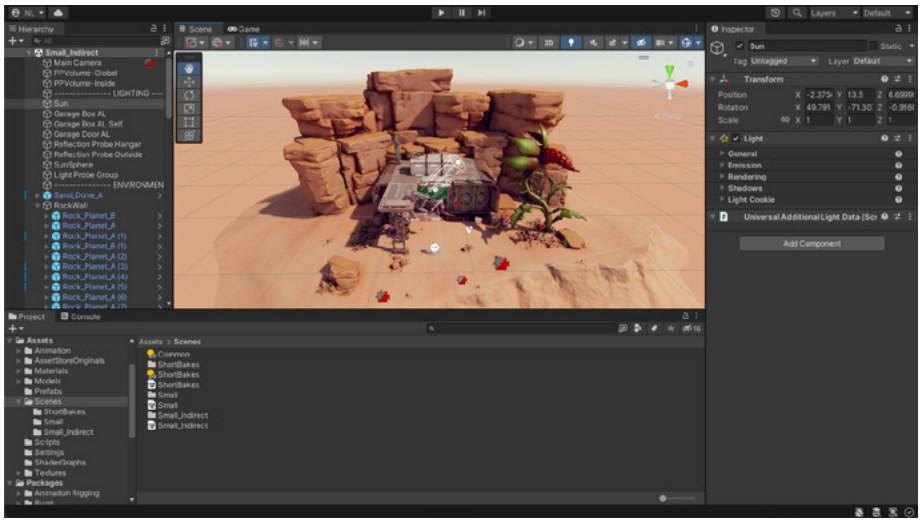
Shadow Atlas for Additional Lights

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.

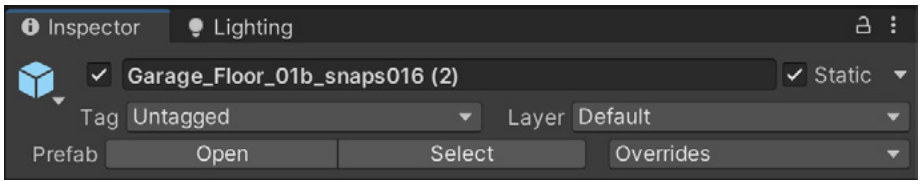
Baked lighting

Let's go through the steps using an FPS Sample project by Unity. The scene demonstrates how to use real-time and baked lighting in URP.



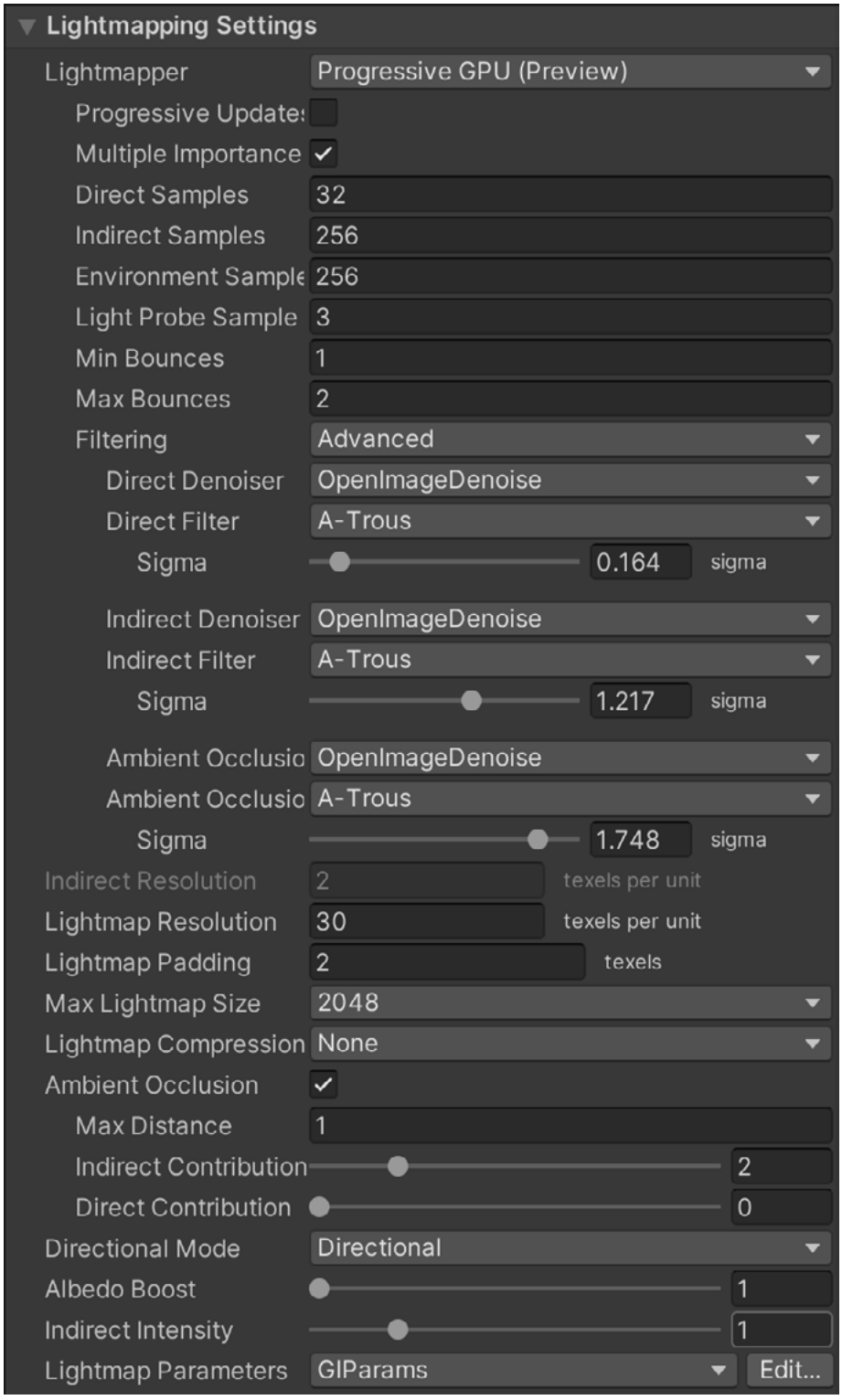
The scene from *FPS Sample: The Inspection* by Unity

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.



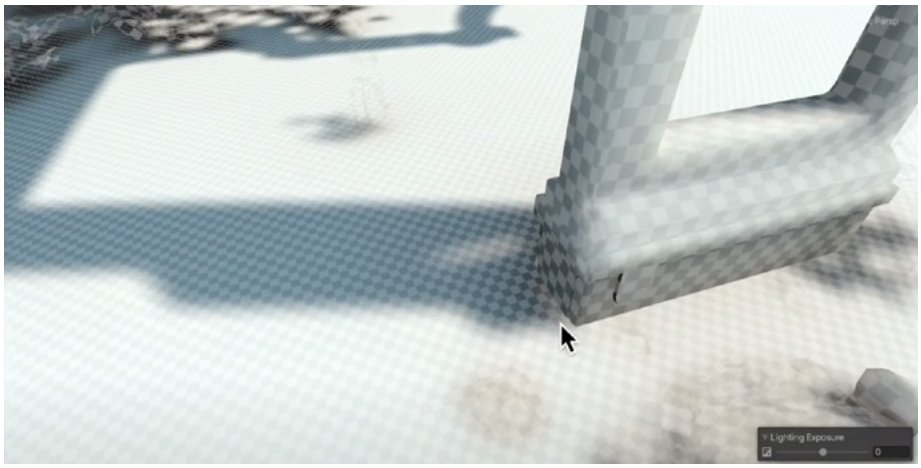
Including geometry in lightmapping

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.



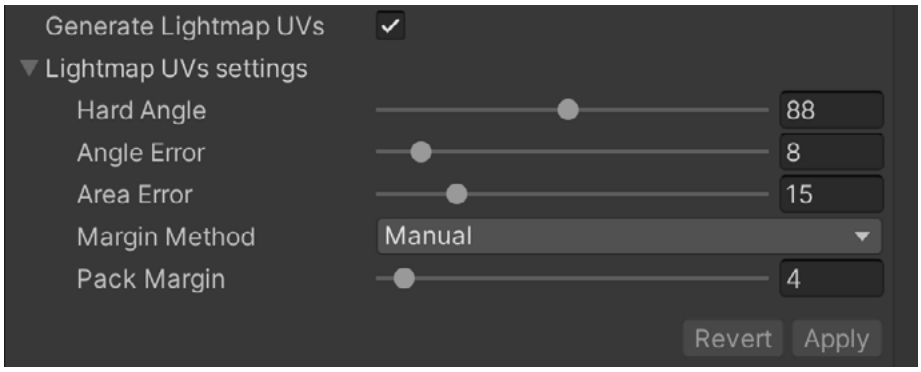
Lightmapping settings

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 the Progressive Lightmapping documentation for more details on the settings available for lightmapping.



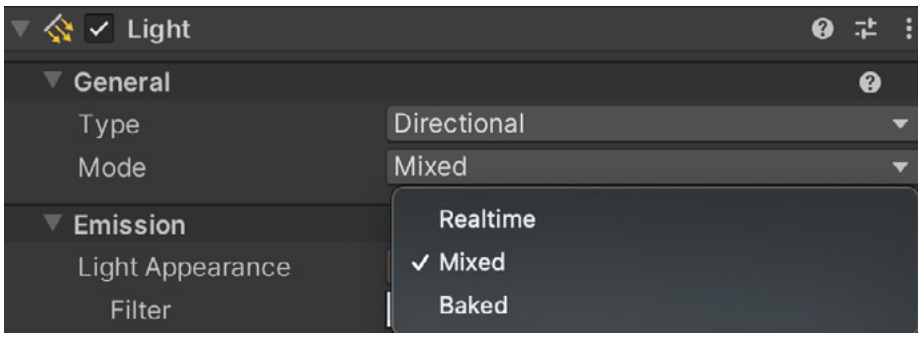
How filtering affects the shadow between objects

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



Generate Lightmap UVs

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.



Set Light Mode to Baked or Mixed

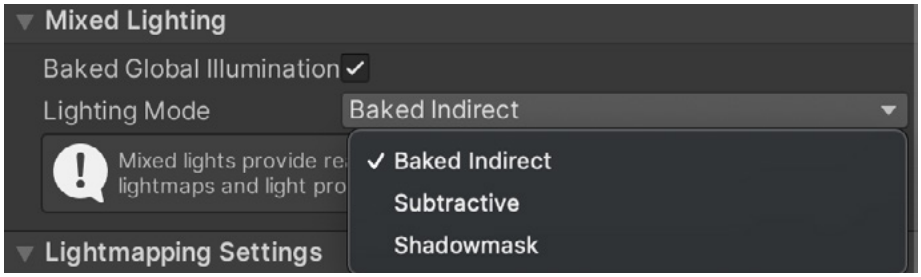
When using Mixed Lights, set the **Light Mode** to **Baked Indirect**, **Subtractive**, or **Shadowmask** via **Window > Rendering > Lighting > Scene**.

— **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.

— **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.

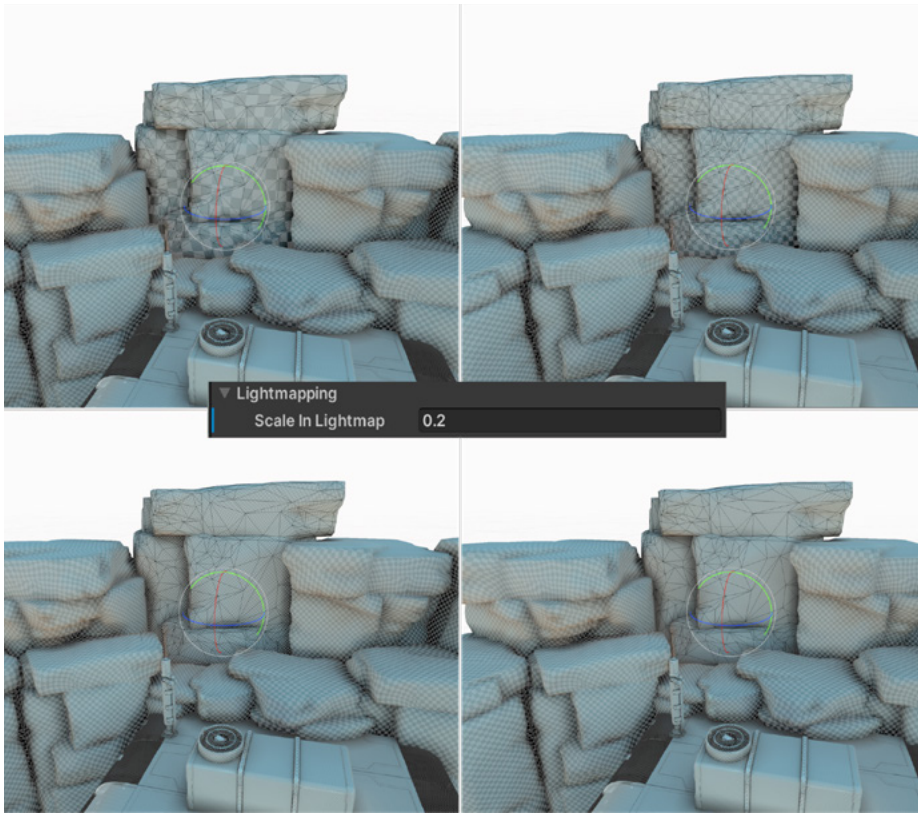
  URP calculates an estimate of the contribution of the light from the Directional Light and subtracts that from the baked Global Illumination. The estimate is clamped by the Real-time Shadow Color setting in the Environment section of the Lighting window, so the color subtracted is never darker than this color. Choose the minimum color of your subtracted value and the original baked color. This is the most suitable option for low-end hardware.

— **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.



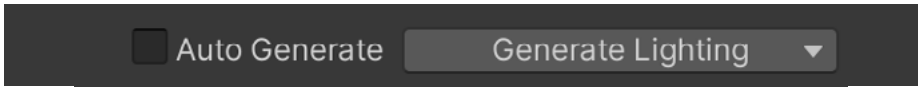
Baked Indirect Global Illumination

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.



Texel size by scale setting: In the top-left image, texel size is set to 0.5; in the top-right image, 0.2; in the bottom-left image, 0.1, and in the bottom-right image, 0.05.

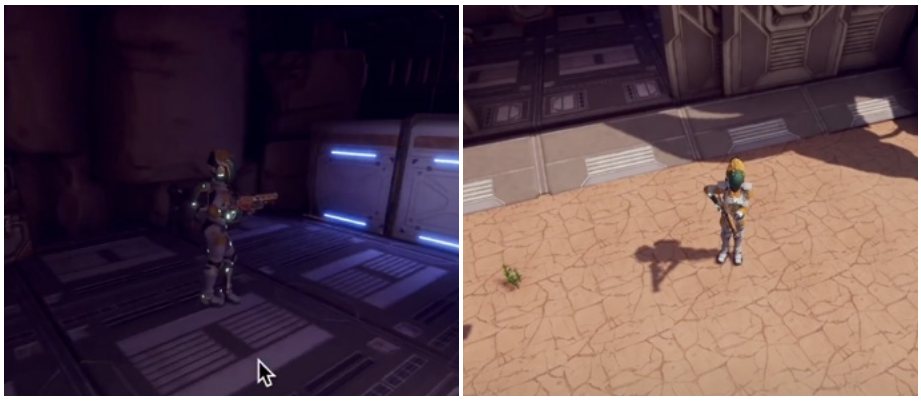
Click **Generate Lighting** to bake. The baking time depends on the number of static objects, the complexity of the meshes, lights set to Mixed or Baked mode, and the settings chosen for lightmapping, particularly the Max Lightmap Size and the Lightmap Resolution.



Generate Lighting to bake

LIGHT PROBES

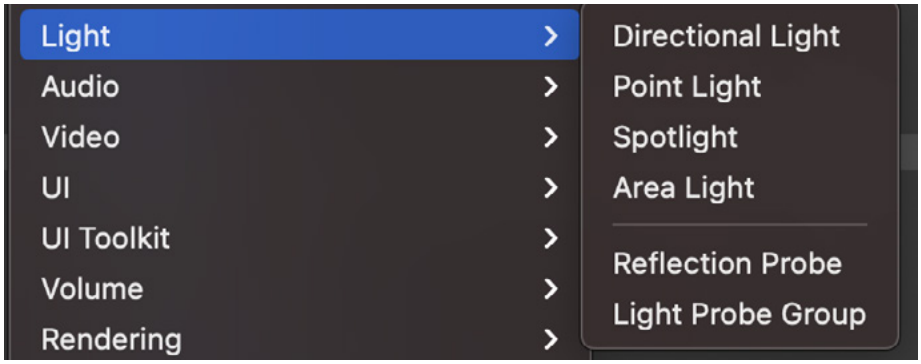
It's recommended to add Light Probes to your scene when using Mixed mode lighting. Light Probes save the light data at a particular position within an environment when you bake the lighting by clicking **Generate Lighting** via **Window > Rendering > Lighting** panel. This ensures that the illumination 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

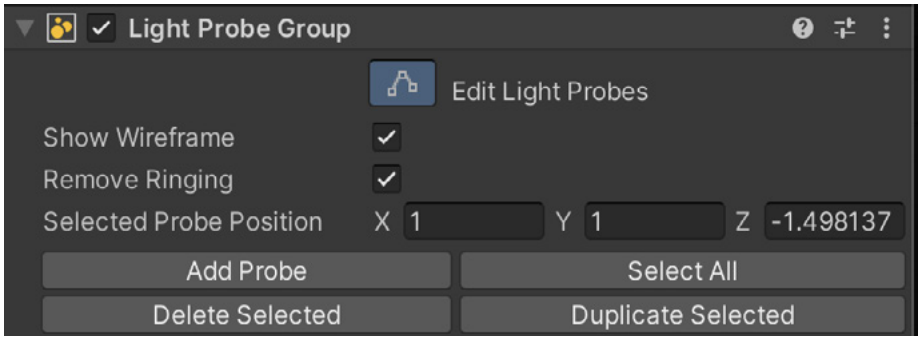
To create Light Probes, right-click in the **Hierarchy** window, and choose **Light > Light Probe Group**.

Slime Rancher 2 by Monomi Park is a colorful, fast-paced first-person adventure game made with Unity using a customized rendering pipeline. To achieve beautiful lighting in an action game, you can combine different lighting techniques like Light Probes for dynamic objects and baking lights for static objects.



Creating a new GameObject for the 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**.

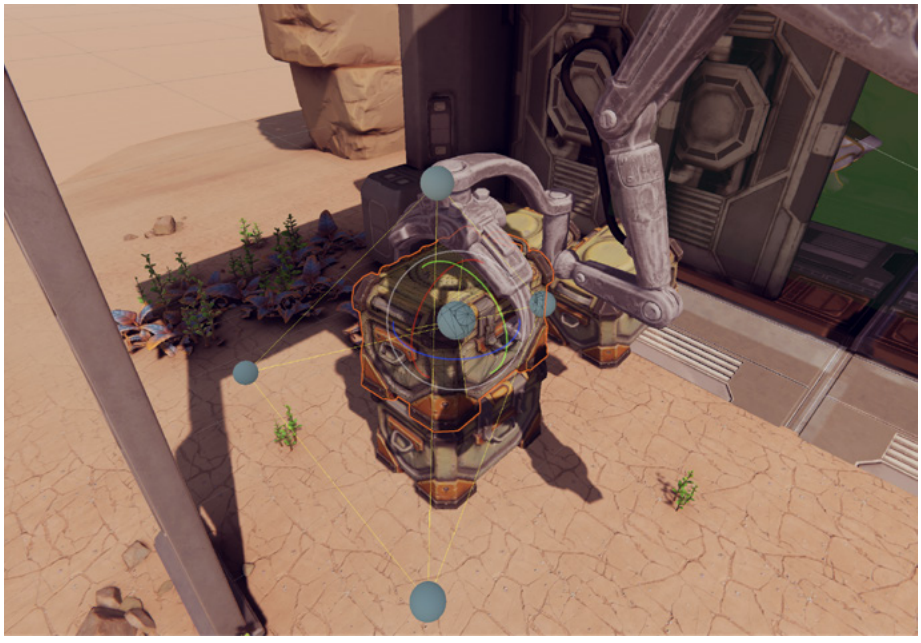


Add or remove Light Probes and modify their position from the Inspector

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 the documentation.

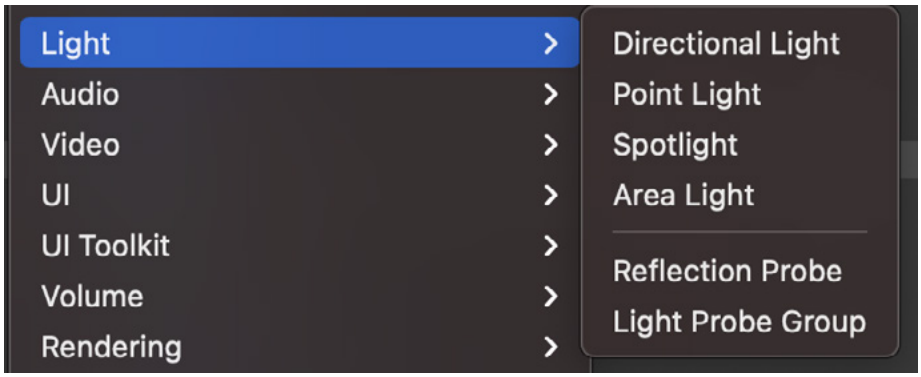
Reflection Probes

A ray-tracing tool like 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 (prerendered 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.

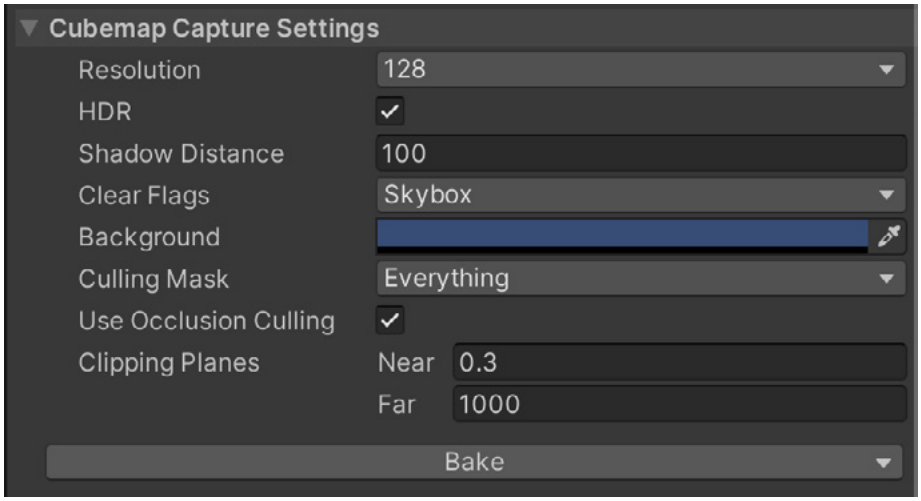
A Reflection Probe is simply a prerendered cubemap placed at a key position in the scene. You can use several Reflection Probes in a single scene. As a dynamic object moves through the scene, it can select the nearest Reflection Probe and use that as the basis of its reflections. You can also set up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

More resources

— Lighting and Lightmapping documentation

— Introduction to lighting and rendering from Unity Learn

— The art of lighting game environments by CGCookie

— Real-time lighting in Unity by Brackeys

— Harnessing light with the URP and the GPU Lightmapper from Unite Now

— Configuring lightmaps from Unity Learn

— Lighting Settings Asset documentation

— Lighting Explorer documentation

SCREEN SPACE REFRACTION

Screen Space refraction uses the current opaque texture created by the render pipeline as the source texture to map pixels to the model being rendered. It can't show models that are not part of the opaque texture. The method is about deforming the UV used to sample the image.

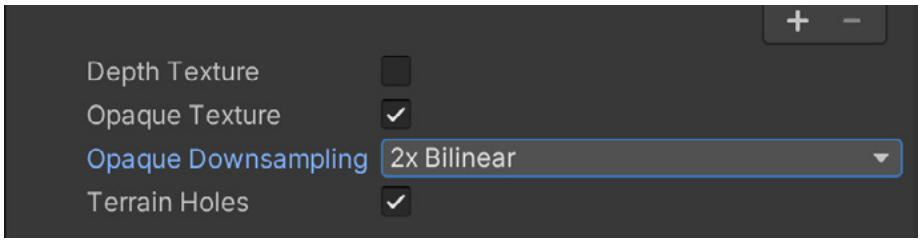
An example of Screen Space Refraction

In the indie game *Arctico*, by the devs Claudio and Antonio, you have to build your base camp and explore a glacier landscape. The abundant water in the game reflects the surface, an effect that can be achieved with Screen Space Reflection. Screen Space Reflection is used to fake a reflective surface in real-time, while Screen Space Refraction is used to simulate transparency and the bending of light as it passes through a medium.

In this recipe, you'll learn how to use a normal map to create refraction effects as well as tint a refraction effect. The additional tinting seen in the previous image is achieved by lerping the calculated pixel color with a **Color** property.

To see the effect in action, take a look at **Scenes > Refraction > Refraction**.

The technique requires the opaque texture to be available to the shader. Find the URP Settings Asset currently assigned in **Edit > Project Settings... > Graphics > Scriptable Render Pipeline Settings**. In the Inspector, make sure **Opaque Texture** is enabled. If you also enable Opaque Downsampling, you'll get a small performance boost. It also introduces a small blur to what you see through the refractive object, which can improve the visual appearance.



Setting Opaque Texture and Opaque Downsampling

The first step to creating the shader is to create a new Shader Graph Asset. Right-click in the Project window, and select **Create > Shader Graph > URP > Lit Shader Graph**.

Creating a new Lit Shader Graph

Create a Material using this shader by selecting the Shader Graph Asset and choosing **Create > Material**. Apply this Material to the object you want to be refractive.

Now double-click on the Shader Graph Asset to open it. Create a **Scene Color** node, and connect this to **Fragment > Base Color**.

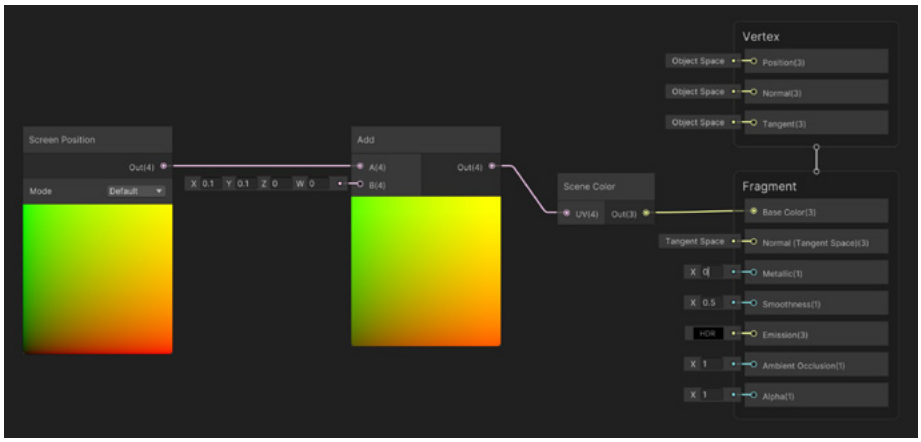
Using a Scene Color node

Scene Color only works with transparent materials since it relies on opaque objects having been rendered. In the render pipeline, transparent objects are rendered after opaque objects. Set **Graph Inspector > Graph Settings > Surface Type** to **Transparent**.

The Scene Color node by default uses normalized screen coordinates for the UV and so maps the opaque texture to each pixel with lighting affected by the smoothness, resulting in the image below.

The result of using Scene Color

Since the goal is to manipulate the UV used by the Scene Color node, you need to override the default UV behavior. Create a **Screen Position** node and an **Add** node. Drag the output of the Screen Position node to input A of the Add node and set the B input as [0.1, 0.1, 0, 0].



Adding nodes to control the UV

Now you'll see the Opaque Texture offset.

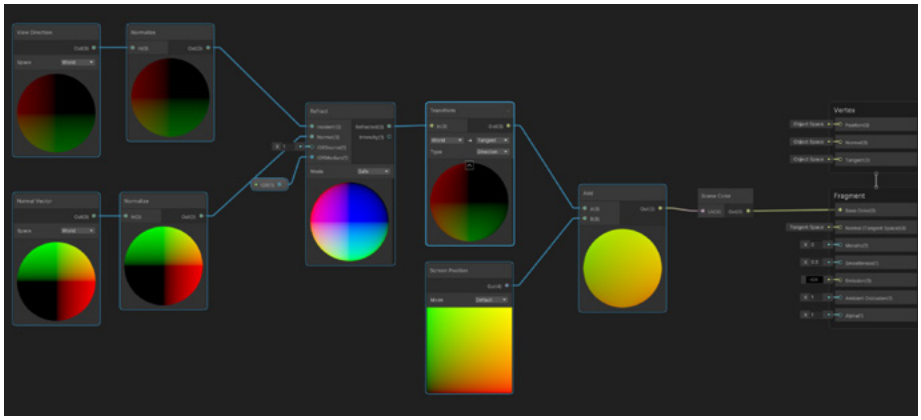
Opaque Texture offset

For each rendered pixel, you want the offset to be controlled by the camera's view direction, the normal for the object at the current screen position, and a scaling value. Shader Graph has a node that will calculate refraction given these three inputs; it actually has two scaling values, but you'll only use one.

You can add a new float property called **IOR**, short for Index of Refraction for scaling. Set it as a slider, with min 1 and max 6. For view direction, add a **View Direction** node and link it to a **Normalize** node to guarantee it's of unit length.

Add a Normal node set to World Space, and again link it to a Normalize node. Create a Refract node, and link the normalized View Direction to the Incident input. Link the normalized Normal to the Refract node Normal input and link the IOR property to the IORMedium input.

At this point, the Refracted output is in World space, but to offset the Screen space UV we need it in Tangent space. Add a Transform node setting the input as World and the output as Tangent. For the type, choose Direction. Use this as the A input to the Add node with Screen Position as the B input. You get the graph you see below.

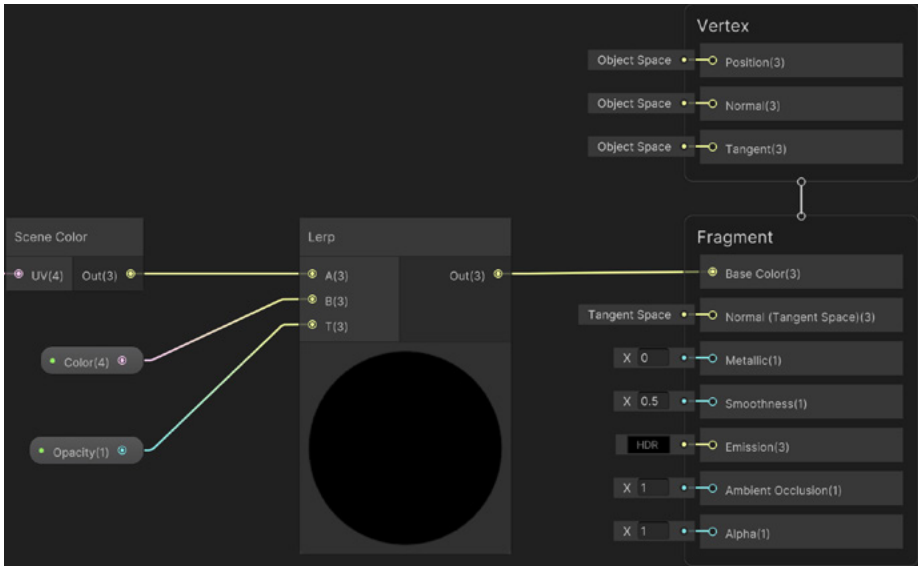


Basic refraction graph

An IOR of 5.44 will result in the visual effect seen in the following image.

Basic Screen Space Refraction

You can tint the result by adding a **Color** property. Add a Lerp node, and use an **Opacity** property set to slider mode, range 0-1, as the T input. The output from Scene Color is set as input A and Color as input B.



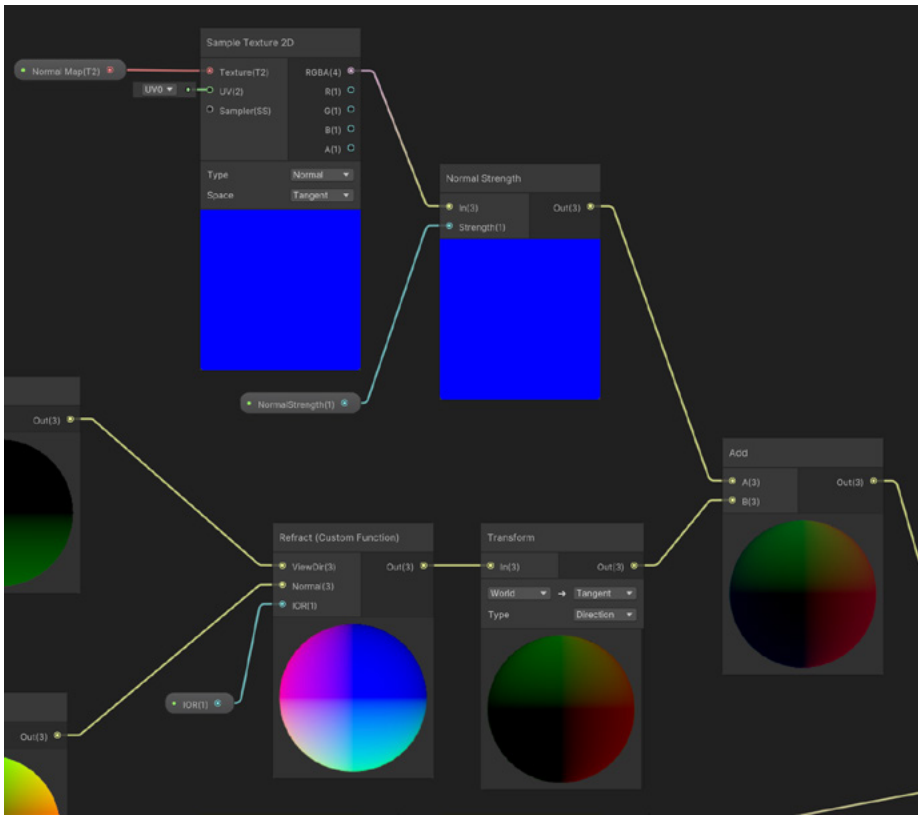
Adding a tinting stage to the graph

You should now be able to tint the output.

A tinted version

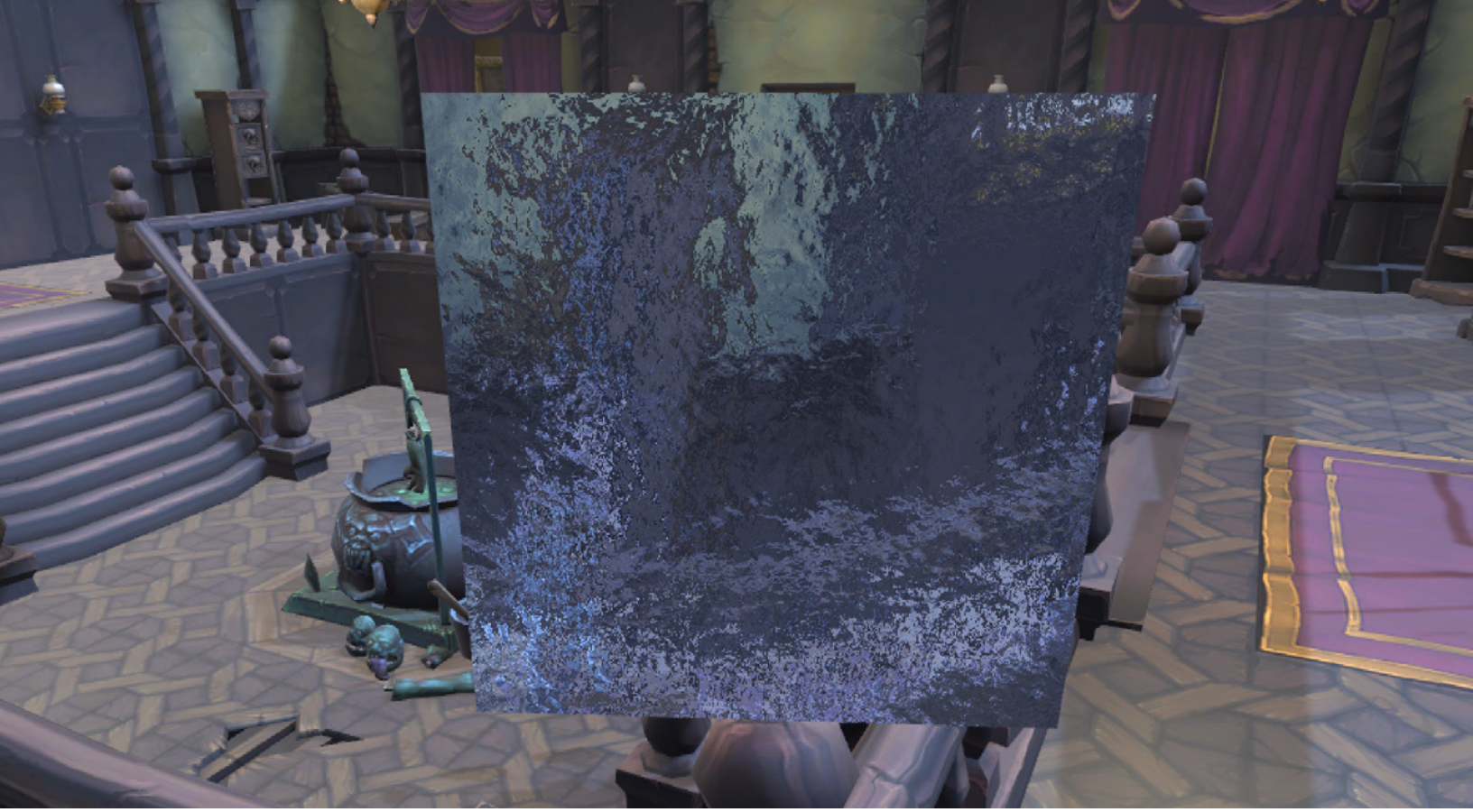
The normal affects the refraction, so a single plane will just get an offset version of the Opaque Texture.

Now it's time to add a normal map. You start by adding a **Texture 2D** property to the shader that you name **Normal Map**, and a float property as a slider, called **Normal Strength**, with a range of 0-1. Create a Sample Texture 2D node, and set it as Type Normal, Space Tangent. Set the Texture input to the Normal Map property. Create a **Normal Strength** node and set input as the RGBA(4) output from the Sample Texture 2D node. Set the Strength input as the Normal Strength property. Create an Add node with input A as the output from the Normal Strength node and input B from the Transform World to the Tangent node. Follow these steps, and you should end up with this graph.



Adding a normal map

Using a suitable normal map should result in the effect seen in the following image, in this case using a single quad instead of the diamond. Refraction for a planar mesh simply shows an offset of the Opaque Texture. Using a normal map with a planar mesh can be a useful way to hide this artifact.



Using a Normal Map

An alternative to using the Refract node is to add a Custom Function Node with a Vector3 viewDir input, a Vector3 normal input, and an IOR input. If you use this option, set your IOR property as a slider with the range -0.15 to 2, not 1-6. Set a Vector3 as the output. The code is very simple so just use a String not a file.

Out = refract(viewDir, normal, IOR);

It gives different results and is worth experimenting with.

More resources

— Screen space refraction by David Lettier

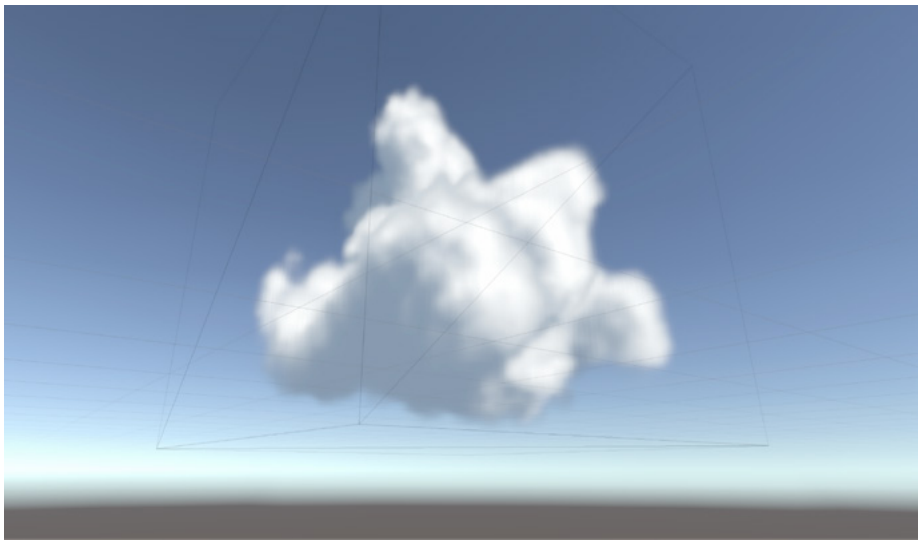
— ScreenSpace planar reflection GitHub repo by Steven Cannavan

— Reflection probes vs Screen space reflection by Kyle W. Powers

— Shader Graph refraction tutorial by AE Tuts

— Crystal Shader Graph in Unity by Binary Lunar

VOLUMETRICS

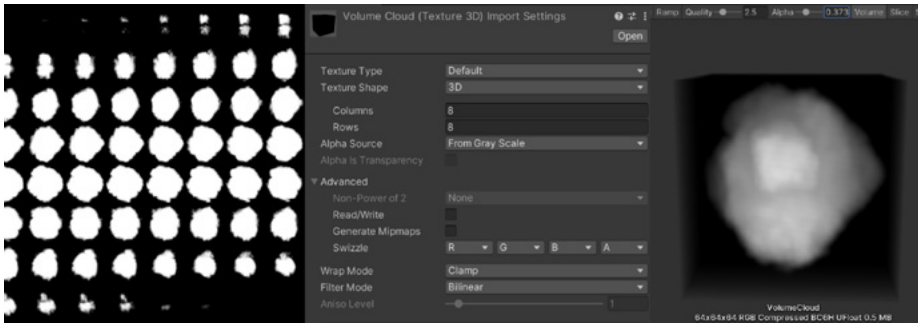


Volumetric cloud

This is a recipe for using ray marching to render a 3D texture. Unity supports 3D textures, which are an array of images placed in a grid on a single texture, rather like a Texture Atlas. The difference is that each image is the same size. Using a 3D UV value, you can source a texel from the grid of images with UV.Z defining the row and column of the individual image to use.

The game *Lost in Random* by Zoink! immerses players into a fantasy kingdom with a very unique art direction where great lighting plays a huge role in creating its atmosphere. They recreated volumetric fog in URP, as seen in this article.

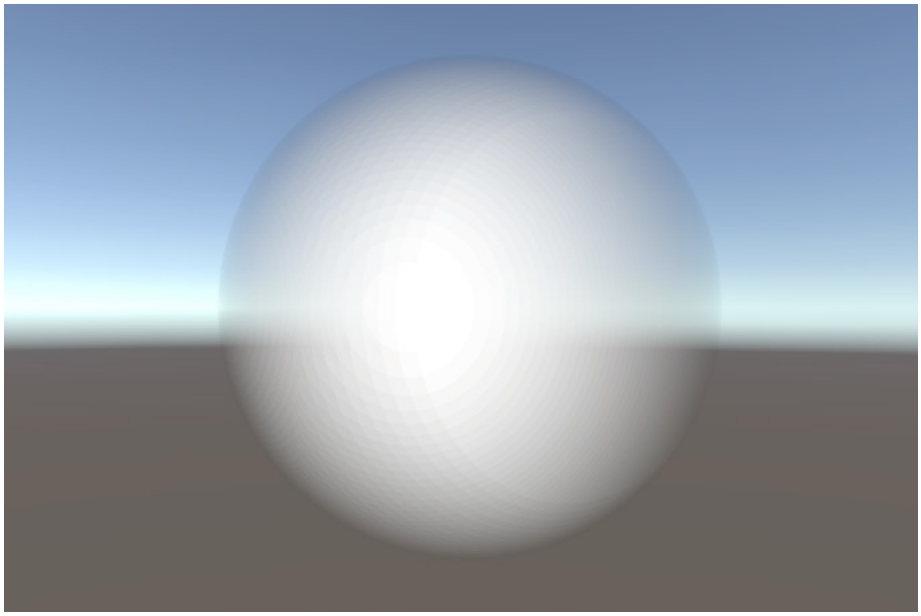
The image below shows a typical 3D texture, its import settings, and a preview in the Inspector.



Left to right: A 3D texture, its import settings, and a preview of it in the Inspector

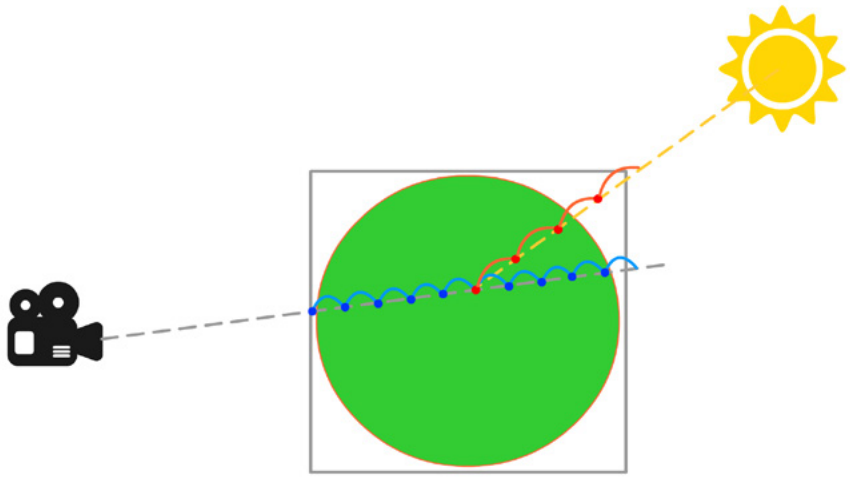
As with the previous recipes, this shader will be built with Shader Graph. To view the finished product, go to **Scenes > Volumetric Clouds**, and open the **VolumetricClouds** scene. Note that the scene includes a Camera, Directional Light, and a cube. The cube uses the Material RaymarchMat.

To start the recipe, you'll need to give the RaymarchMat material the shader named **Shader Graphs/Raymarchv1SG**, created by Nik Lever. You should now see a sphere. If you adjust the **densityScale**, you can begin to see transparency at the edges.



Using the Shader Graph Raymarchv1SG

You're supposed to be rendering a cube, but instead you see a sphere: What's going on? The answer is ray marching. Ray marching, according to its Wikipedia page, "is a class of rendering methods for 3D computer graphics where rays are traversed iteratively, effectively dividing each ray into smaller ray segments, sampling some function at each step. This function can encode volumetric data for volume ray casting, distance fields for accelerated intersection finding of surfaces, among other information."



Ray marching

With this first version, a sphere is defined using a Vector4. XYZ defines the position of the sphere, relative to the object and W its radius. For each pixel, a direction is calculated for a ray that comes directly from the camera (represented by the dotted gray line in the diagram above). Set a density value to 0, then move along this line calculating at each blue dot inside the sphere to add a small value to density. When the ray has traveled through the sphere,

you'll have a value for how much of the sphere is in a line directly from the camera to the pixel you're rendering. This density value is used as the Base Color in the Shader Graph. Ignore the Sun and the red dots for now; these will be considered later when it's time to add lighting, in the fourth version of this shader.

This graph uses a Custom Function Node based on the file via **Scripts > HLSL > Raymarch.hlsl**. For this first version, you'll use the function `raymarchv1`. The variable density is initialized to 0. Then you enter a for loop for numSteps count. The rayOrigin is moved by stepSize in the direction defined by rayDirection.

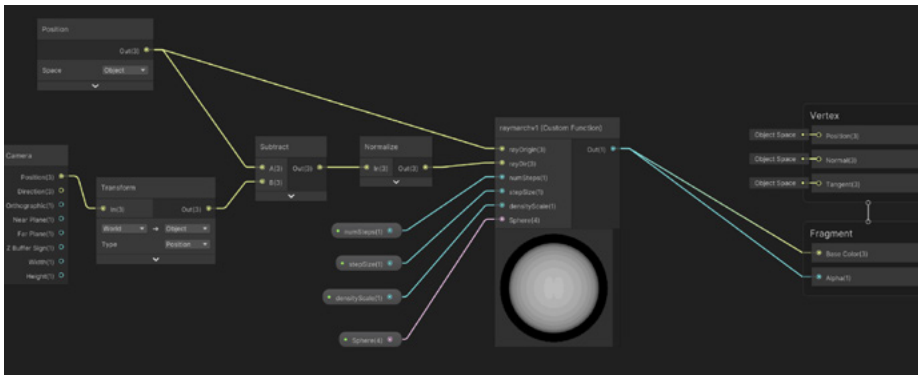
How far are you from the sphere origin? You can use the HLSL function distance to calculate the length of a vector from the sphere origin to the current value for rayOrigin. If this is less than the sphere radius (Sphere.w), then add 0.1 to the density value. The output value result is the accumulated density value times densityScale.

```
void raymarchv1_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                       float stepSize, float densityScale,
float4 Sphere,
                       out float result )
{
       float density = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              //Calculate density
              float sphereDist = distance(rayOrigin, Sphere.
xyz);

              if(sphereDist < Sphere.w){
                     density += 0.1;
        }

       }

       result = density * densityScale;
}
```

For your calculations, you'll work in Object Space. You get the rayOrigin using a **Position** node and to get the rayDirection, you need a **Camera** node that links the position output to a **Transform** node, with the input set as World and the output Object.

You now have the pixel position and Camera Position in Object Space, which enables you to get the ray direction using a **Subtract** node, with Position as

input A and Camera Position as input B. This rayDirection is normalized using a **Normalize** Node. The other inputs to the Custom Function Node are the float properties, numSteps, number of blue dots per ray, stepSize, the distance between blue dots, densityScale, and the Vector4 sphere discussed earlier. The density output goes directly to Base Color and Alpha. Note that this shader is set to be transparent and unlit, requiring you to calculate the lighting.



Version 1 of the Ray march shader

Ray marching comes to life when a 3D texture is added to determine the shape. You'll introduce a 3D texture in version 2. Start by setting RaymarchMat to use **Shader Graphs > Raymarch2SG**. The Custom Function used is raymarchv2.

```
void raymarchv2_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                       float stepSize, float densityScale,
UnityTexture3D volumeTex,
                       UnitySamplerState volumeSampler, float3
offset,
                       out float result )
{
       float density = 0;
       float transmission = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              //Calculate density
              float sampledDensity = SAMPLE_TEXTURE3D(volumeTex,
volumeSampler, rayOrigin + offset).r;
              density += sampledDensity;

       }

       result = density * densityScale;
}
```

You'll notice there are three new inputs:

— A UnityTexture3D volumeTex that comes directly from a **Material** property.

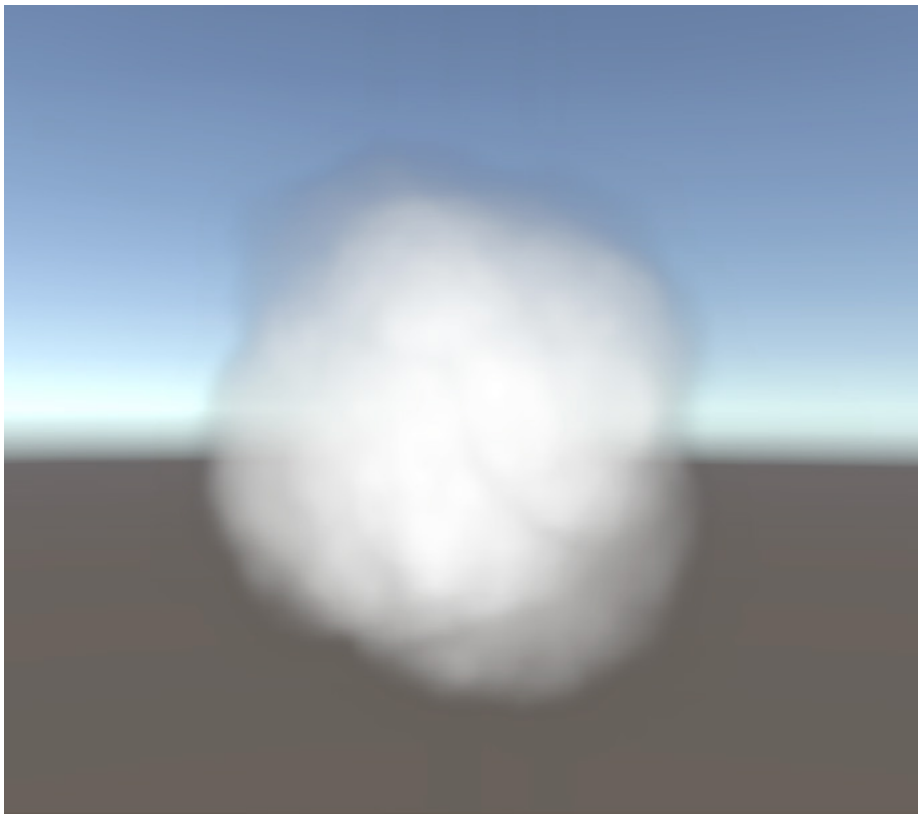
— A MACRO, SAMPLE_TEXTURE3D, necessary when working with 3D textures, that needs a SamplerState instance.

    — There is a node for SamplerState that allows you to select the wrapping option. You'll set that to clamp so that UV values outside of the range 0 - 1 are clamped at 0 for values less than 0 and at 1 for values above 1.

— Offset, which is a value you can use to move around our 3D texture inside the Cube.

Now, instead of checking whether you are inside a sphere, you'll get a **sampledDensity** value using the float3 sample position of rayOrigin plus offset. You only need one channel here, the red channel, R.

The image below shows a render of version 2. It's beginning to look like a cloud.



Version 2 of the shader

The final version of the shader introduces lighting. Use the shader named Shader Graphs/Raymarchv3SG for the Material RaymarchMat. This time, you'll use the function `raymarch`. The function uses six new parameters: **numLightSteps**, **lightStepSize**, **lightDir**, **lightAbsorb**, and **transmittance**, and returns a float3 vector.

To build up the final values, initialize three new variables: **transmission**, **lightAccumulation** and **finalLight**. The code is the same as version 2 up to the light loop comment. Look again at the "ray marching" illustration shown earlier: For each step along the view direction ray, represented by the blue dots, you get a ray towards the main light, which is yellow in the diagram. The red dots represent the step-by-step sampling of the 3D texture. The more cloud you find, the less light that will hit that part of the view direction ray. This process determines how bright each pixel is.

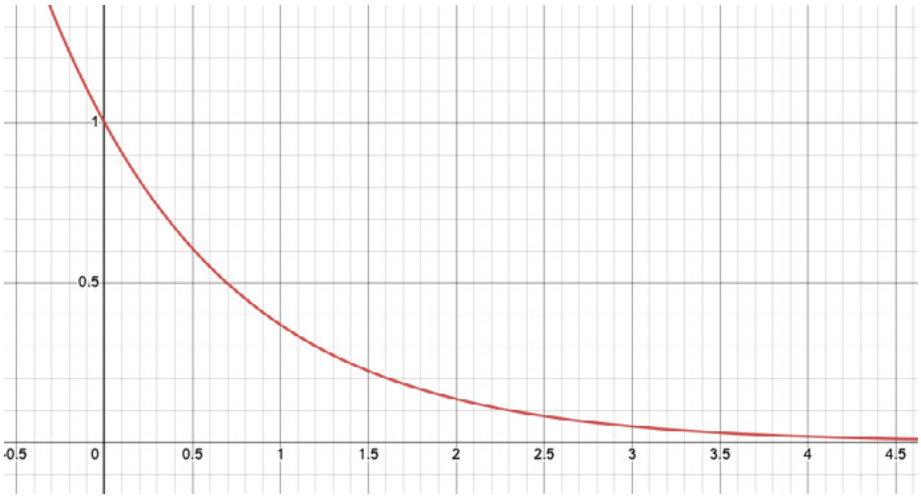
```
void raymarch_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                     float stepSize, float densityScale,
UnityTexture3D volumeTex,
                     UnitySamplerState volumeSampler, float3
offset,
                     float numLightSteps, float lightStepSize,
float3 lightDir,
                     float lightAbsorb, float
darknessThreshold, float transmittance,
                     out float3 result )
{
       float density = 0;
       float transmission = 0;
       float lightAccumulation = 0;
       float finalLight = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              float3 samplePos = rayOrigin+offset;
              float sampledDensity =
                  SAMPLE_TEXTURE3D(volumeTex, volumeSampler,
samplePos).r;
              density += sampledDensity*densityScale;
              //light loop
              float3 lightRayOrigin = samplePos;

              for(int j = 0; j < numLightSteps; j++){
                     lightRayOrigin += -lightDir*lightStepSize;
                     float lightDensity =
             SAMPLE_TEXTURE3D(volumeTex, volumeSampler,
lightRayOrigin).r;
                     lightAccumulation += lightDensity;
              }

             float lightTransmission = exp(-lightAccumulation);
              float shadow = darknessThreshold +
                                 lightTransmission * (1.0 -
darknessThreshold);
              finalLight += density*transmittance*shadow;
```

```
            transmittance *= exp(-density*lightAbsorb);

        }

        transmission = exp(-density);

        result = float3(finalLight, transmission, transmittance);
}
```

The light loop is easy to understand, repeating the number of times that have been specified for the **numLightSteps** variable. Bear in mind that this is a nested loop, so keep the numLightSteps count as low as possible. You move from samplePos towards the main light by using minus lightDir. Then, lightDensity is added to lightAccumulation. Some math is required outside the light loop:

float lightTransmission = exp(-lightAccumulation);

First, lightTransmission is set as $e^{-lightAccumulation}$. The constant e, Euler's number, is about 2.718. The graph below shows the result of this function. The horizontal axis is the value of lightAccumulation and the vertical axis exp(-lightAccumulation). When the accumulated light density, lightAccumulation, is 0, exp(-lightAccumulation) equals 1. As lightAccumulation increases exp(-lightAccumulation) quickly drops away nearing 0 if lightAccumulation is 5 or more.

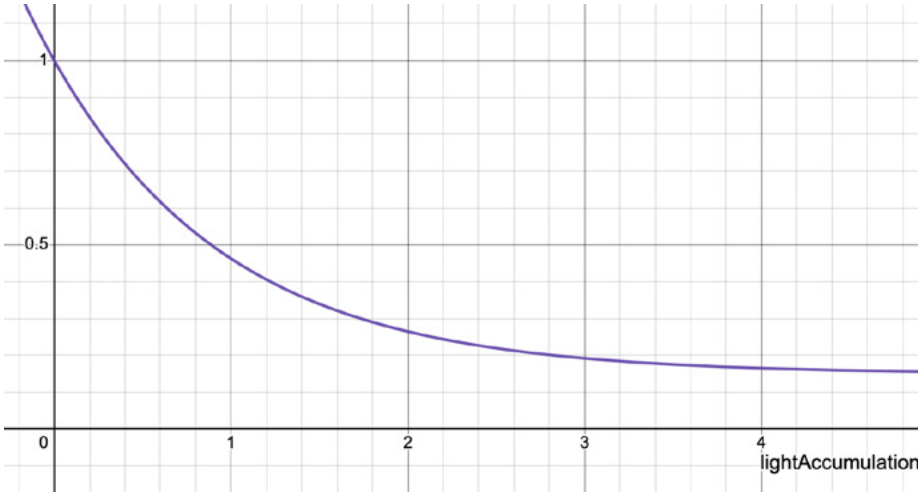


Graph of e^{-x} for the range 0 to 4

float shadow = darknessThreshold +
lightTransmission * (1.0 - darknessThreshold);

A shadow value is calculated next. Use the property called **darknessThreshold**. The graph below shows the shadow value in the vertical axis, for a

darknessThreshold of 0.15. If lightAccumulation is 0 then shadow equals 1, whereas if lightAccumulation approaches 5, then shadow tends to the darknessThreshold constant value.



The shadow value

finalLight += density*transmittance*shadow;

Density * transmittance * shadow is added to the finalLight accumulated value. If the accumulated light density, lightAccumulation, is high, then shadow will tend to 0 and therefore, the accumulated value for finalLight will be less.

transmittance *= exp(-density*lightAbsorb);

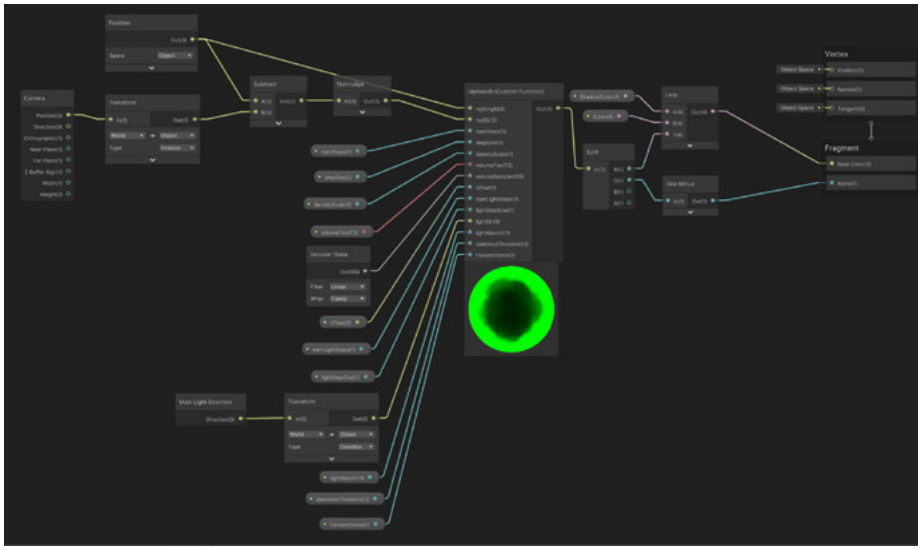
The initial value of transmittance is a passed-in property, but for each view direction step, its value is multiplied by $e^{-density*lightAbsorb}$. The property **lightAbsorb** controls how much light gets lost in the cloud by scattering.

For version 3, the result is a float3 containing the finalLight, transmission, and transmittance.

The graph for version 3 is shown below. Now that the output from the Custom Function is a float3, a Split node is added. Output R goes to a Lerp node T input. Version 3 has several new properties, including **color** and **shadowColor**, with the former the B input and the latter the A input.

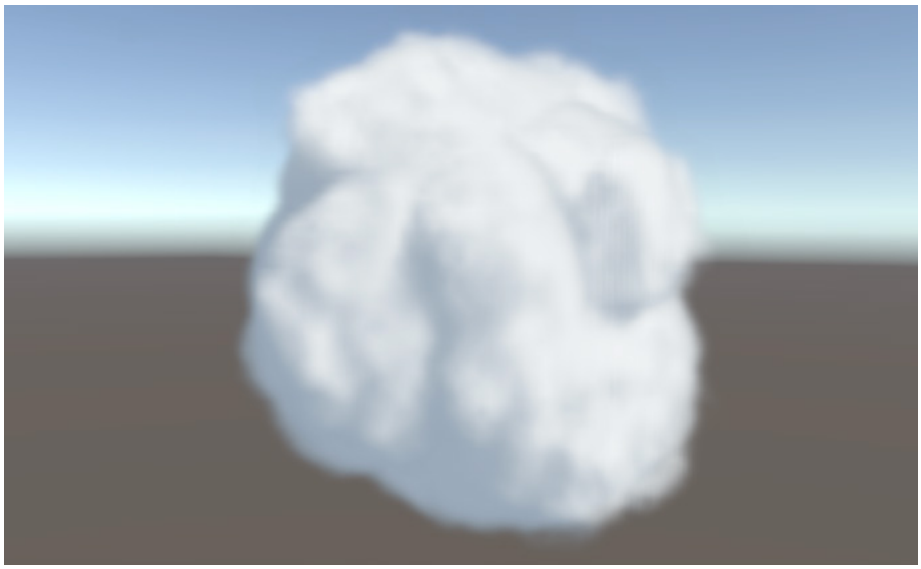
If finalLight raymarch node Out.x is 0, then the shadowColor will be passed to the Lerp node output. If finalLight is 1, then color is passed to the output. In the range 0-1 a linear interpolation of shadowColor and color is the output. The Lerp node output goes directly to Fragment > Base Color.

Alpha uses the raymarch node with transmission value Out.y. This value is 0 when Alpha should be 1 and 1 when it should be 0. A One Minus node is used to correct the Split node B value and link this to Fragment > Alpha.



Final version

This gives the result you see below.



A cloud with ray marching

Houdini is a useful tool when creating the 3D texture. Alternatives to a 3D texture include using multilayered Perlin noise, or pre-baking a tileable noise texture using Unity. Hopefully, this recipe will be a starting point for your journey into ray marching.

More resources

— Volumetric ray marching cloud shader by dmeville

— Coding adventure: Clouds by Sebastian Lague

— Creating Volumetric Clouds with Houdini by Camelia Slimani

— Altos sky system, by OccaSoftware

CONCLUSION

There are many advanced resources available for free from Unity. As mentioned in the introduction to this guide, the e-book *Introduction to the Universal Render Pipeline for advanced Unity creators* is a valuable guide for helping experienced Unity developers and technical artists migrate their projects from the Built-in Render Pipeline to the URP. Topics covered in the e-book 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 the callback differences between the two rendering pipelines, performance optimization in URP, and more

All of the advanced technical e-books and articles are available from the Unity best practices hub. E-books can also be found on the advanced best practices Documentation page.

Professional training for Unity creators

Unity Professional Training gives you the skills and knowledge to work more productively and collaborate efficiently in Unity. Find an extensive training catalog designed for professionals in any industry, at any skill level, in multiple delivery formats.

All materials are created by experienced Instructional Designers in partnership with our engineers and product teams. This means that you always receive the most up-to-date training on the latest Unity tech.

Learn more about how Unity Professional Training can support you and your team.


Unity®