



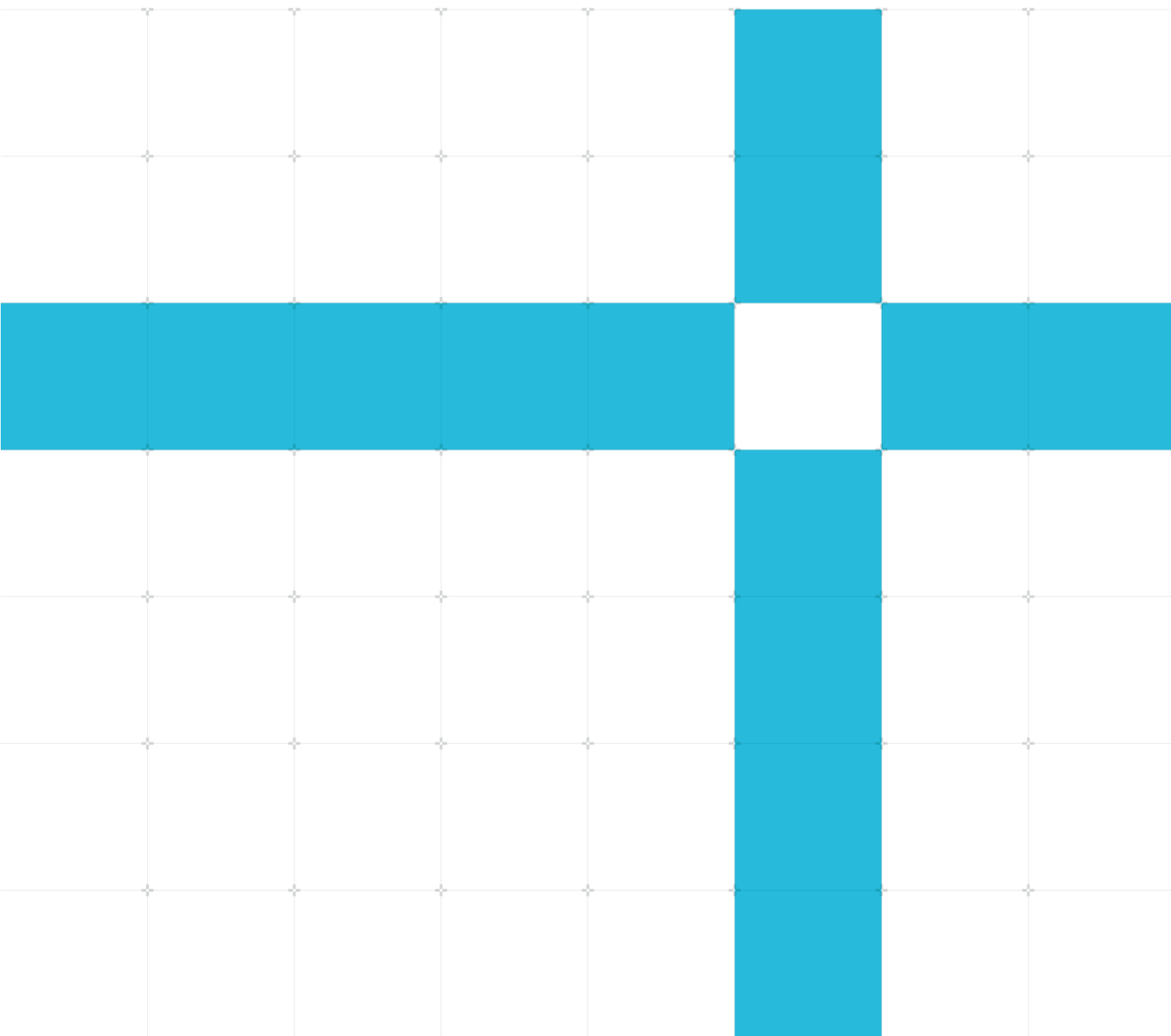
Migrating Unity shaders to Universal Render Pipeline

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 0100

102487_0100_00



Migrating Unity shaders to Universal Render Pipeline

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
01	30 th April 2021	Non-confidential	First release

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names

mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email terms@arm.com.

Contents

1 Overview	5
1.1 Before you begin	5
2 What is the Universal Render Pipeline?.....	6
3 Migrating built-in shaders to the Universal Render Pipeline.....	7
3.1 Differences between the built-in shader and URP shaders	8
4 Migrating custom shaders to the Universal Render Pipeline.....	10
4.1 Migrating include files and functions.....	10
4.1.1 cginc.....	10
4.1.2 Space transforms.....	11
4.1.3 Shader variable functions	11
4.1.4 Common include	12
4.2 Preprocessor macros	12
4.3 LightMode tags	13
4.4 Changing replacement shaders.....	14
4.5 Post-processing.....	18
4.5.1 If you are using custom post-processing and Unity 2019.4.....	18
4.5.2 Mobile-friendly features.....	19
5 Related information	20
6 Next steps.....	21

1 Overview

The Universal Render Pipeline (URP) in Unity optimizes your graphics across a range of platforms, from mobile to computer. URP produces good quality graphics on high end devices and optimized performances on lower-end devices. Other advantages of URP are described in [What is Universal Render Pipeline?](#)

In this guide, we describe how to migrate Unity shaders that have been written for the built-in pipeline to the URP. The guide also describes how to migrate your custom shaders to URP. This is because these shaders cannot automatically migrate to URP.

At the end of the guide, you will:

- Be familiar with URP
- Understand how you can migrate your shaders to URP

1.1 Before you begin

Before you work through this guide, you will need general familiarity with Unity, specifically implementing shaders in Unity. To learn more, read our guides:

- [Real-time 3D art best practices: materials and shaders](#)
- [Advanced graphic techniques](#)

2 What is the Universal Render Pipeline?

The Universal Render Pipeline is a prebuilt implementation of the Scriptable Render Pipeline (SRP). The URP is optimized to deliver high graphics performance and is the successor of the Lightweight Render Pipeline (LWRP). The URP makes some tradeoffs around lighting and shading, to make sure that there is consistent performance on a platform.

The URP provides a friendly workflow that allow artists to easily create optimized graphics across a range of platforms.

Migrating your current project to the URP makes it is easier for you to build custom shaders. Migrating your built-in pipeline to the URP also gives you access to the following features:

- Screen Space Ambient Occlusion (SSAO)
- Clear Coat
- Camera Normals Texture
- Detail Map and Detail Normal Map
- Shadow Distance Fade
- Shadow Cascade
- Shadowmask
- Parallax mapping and Height Map property

3 Migrating built-in shaders to the Universal Render Pipeline

If your project contains built-in shaders, these must be converted to URP shaders. This is because built-in shaders are not compatible with URP shaders.

This section of the guide describes how to migrate your built-in shaders to the Universal Render Pipeline. Migrating built-in shaders uses the Unity upgrader. We will review migrating custom shaders in Migrating custom shaders to the Universal Render Pipeline.

To upgrade your built-in shaders, follow these steps:

1. Open your project in Unity.
2. Go to **Window > Package Manager**.
3. In the **Packages** dropdown list, select **Unity Registry**. This option lists all packages available for your version of Unity.
4. From the list of packages, select **Universal RP**.
5. Click **Install**. Unity installs the URP for your project.

The following screenshot shows Universal RP selected in the Unity Registry list:

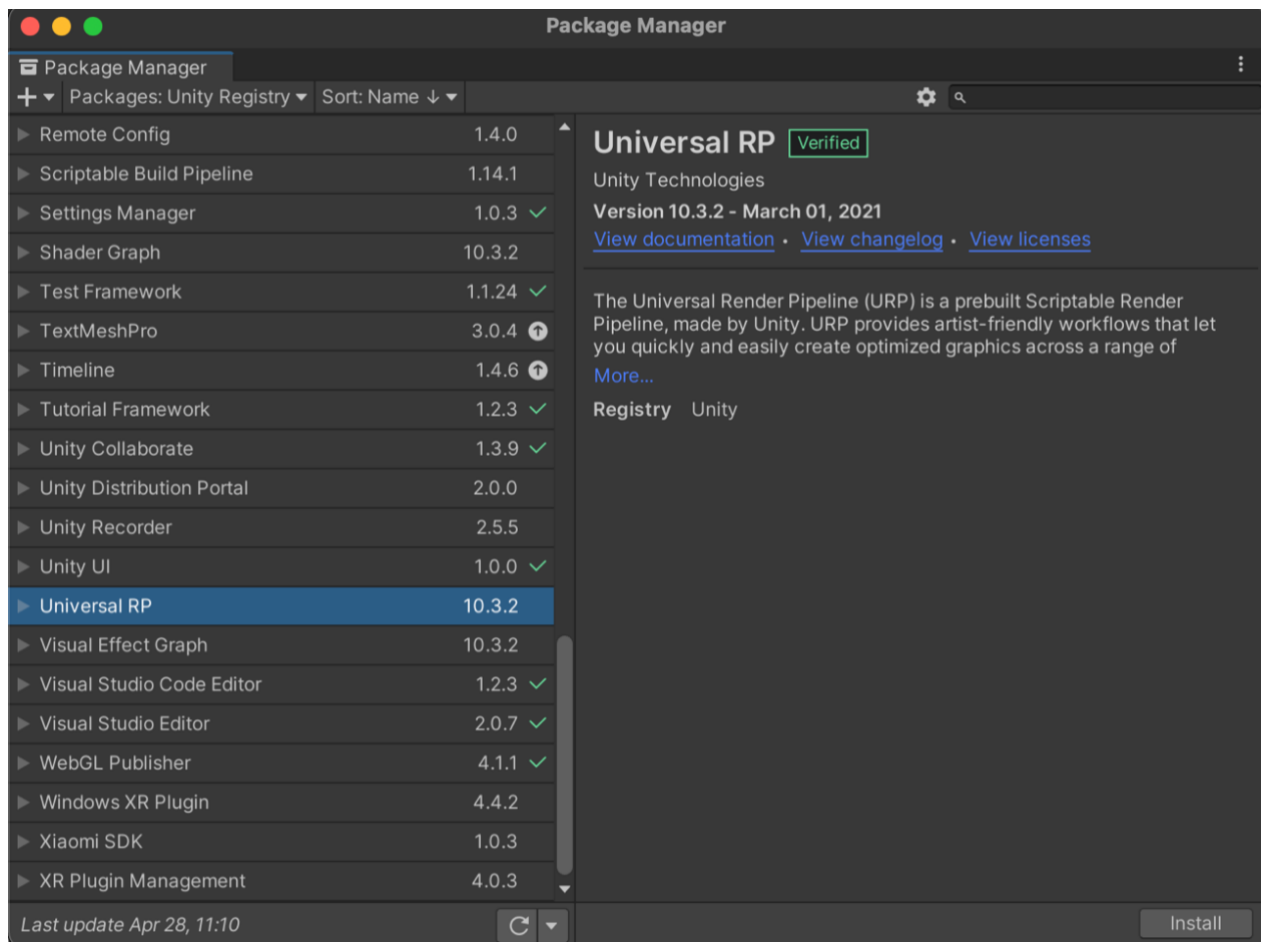


Image 1: Selecting URP in the Package Manager

6. Go to **Edit > Render Pipeline > Universal Render Pipeline**.
7. Select either **Upgrade Project Materials to URP Materials** or **Upgrade Selected Materials to URP Materials**.

When you run the shader upgrader, the Unity built-in shaders convert automatically to a set of URP shaders.

The Unity guide [Upgrading your shaders](#) contains a table that shows you which built-in shaders convert to which URP shader. Many built-in shaders are converted to the Universal Render Pipeline Simple Lit shader.

3.1 Differences between the built-in shader and URP shaders

There are two differences between built-in shaders and URP shaders:

- In URP, the C for graphics (Cg) shader programming language has been replaced with High Level Shading Language (HLSL). However, using HLSL has not dramatically changed the shader syntax and the functionality.
- The shaders in Unity are written in ShaderLab syntax to define the shader properties, subshaders, and passes. However, in the URP the shader code inside the passes is written in HLSL. This means that the shaders that are written for the built-in pipeline are automatically disabled in the URP. This is because the shaders from the built-in pipelines perform separate shader passes for every light that reaches an object. However, the URP handles all lighting and shading in a single pass using arrays. This change leads to different structures to store light data and new shading libraries with new conventions.

The following render pass code from the URP shows that the shader code is delimited using the `HLSLPROGRAM / ENDHLSL` macros:

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

4 Migrating custom shaders to the Universal Render Pipeline

Unlike built-in shaders, custom shaders cannot be automatically converted by the shader upgrader that we used in the previous section of the guide. Therefore, migrating custom shaders to URP requires some manual actions.

This section of the guide reviews the actions that you need to take when migrating your custom shaders to the URP. These actions include changing the following elements:

- **Includes**
- **Preprocessor Macros**
- **LightMode tags**
- **Replacement shaders**
- **Post-processing**

4.1 Migrating include files and functions

First, we describe how to migrate your include files and functions to the Universal Render Pipeline.

4.1.1 cginc

Replace `cginc` include files with the HLSL equivalents.

You can find the `cginc` files in your Unity installation folder. You can find the HLSL includes and headers in the Unity Graphics GitHub repository, and see how the functions are implemented.

Note: In CG, the include files have the extension `.cginc`, and shader files have the extension `.shader`. In HLSL, the includes have the extension `.hlsi`, and the shaders files have the same `.shader` extension as CG.

The following table shows the most commonly included files:

CGS	HLSL
UnityCG.cginc	Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsi

CGS	HLSL
<code>AutoLight.cginc</code>	Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl Packages/com.unity.render-pipelines.universal/ShaderLibrary/Shadows.hlsl

Table 1. Example of CGS and HLSL include files

4.1.2 Space transforms

Replace the functions that you use in your custom shaders with equivalent HLSL functions.

The following table shows a few space transform HLSL functions and what they do:

Built-in function	Action
<code>float4x4 GetObjectToWorldMatrix()</code>	Return the PreTranslated ObjectToWorld Matrix
<code>float4x4 GetWorldToObjectMatrix()</code>	Return the WorldToObject Matrix
<code>float4x4 GetWorldToHClipMatrix()</code>	Transform to homogenous clip space
<code>float4x4 GetViewToHClipMatrix()</code>	Transform to homogenous clip space
<code>float3 TransformObjectToWorld(float3 position Object Space)</code>	As the function name indicates
<code>float3 TransformWorldToObject(float3 position World Space)</code>	As the function name indicates
<code>float3 TransformWorldToView(float3 position World Space)</code>	As the function name indicates

Table 2. Examples of HLSL space transform functions

Note: You can find more space transform HLSL functions in the include file [SpaceTransforms](#).

4.1.3 Shader variable functions

Replace the functions that you use in your custom shaders with equivalent HLSL functions.

The following table shows some shader variable functions and what they do:

Built-in function	Action
<code>VertexPositionInputs GetVertexPositionInputs(float3 positionOS)</code>	Returns input.positionWS, input.positionVS, input.positionCS, input.positionNDC
<code>VertexNormalInputs GetVertexNormalInputs(float3 normalOS)</code>	Returns input.tangentWS, input.bitangentWS, input.normalWS
<code>float3 GetCameraPositionWS()</code>	Returns camera position in world space.
<code>float3 GetCurrentViewPosition()</code>	Returns current view position in world space.
<code>float3 GetViewForwardDir()</code>	Returns the forward (central) direction of the current view in the world space.

Built-in function	Action
float3 GetWorldSpaceViewDir(float3 positionWS)	Computes the world space view direction (pointing towards the viewer).

Table 3. Examples of HLSL shader variable functions

Note: Other fog and UV-related functions can be found in the include file **ShaderVariablesFunctions**.

4.1.4 Common include

The Common include contains many built-in functions that are related to some platform-specific functions, common math functions, and space transformations. Replace the functions that you use in your custom shaders with equivalent HLSL functions.

The following list shows some of the HLSL functions in the common include:

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

4.2 Preprocessor macros

Preprocessor macros are defined when compiling each shader. However, when migrating the built-in shaders to new URP shaders, you must replace the C for graphics (Cg) macros with their High-Level Shading Language (HLSL) equivalents.

The following table shows a few of the replacements:

Built-in Cg macros	URP HLSL equivalents
UNITY_PROJ_COORD(a)	Replace with a.xy/a.w
UNITY_INITIALIZE_OUTPUT(type, name)	ZERO_INITIALIZE(type, name)
Shadow Mapping. Shadow Mapping Macros need the shadows include .	
UNITY_DECLARE_SHADOWMAP(tex)	TEXTURE2D_SHADOW_PARAM(textureName, samplerName)
UNITY_SAMPLE_SHADOW(tex, uv)	SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord3)
UNITY_SAMPLE_SHADOW_PROJ(tex, uv)	SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord4.xyz/coord4.w)
Texture or Sampler declaration For built-in Texture/Sampler declarations see the Unity documentation	
UNITY_DECLARE_TEX2D(name)	TEXTURE2D(textureName); SAMPLER(samplerName);
UNITY_DECLARE_TEX2D_NOSAMPLER(name)	TEXTURE2D(textureName);
UNITY_SAMPLE_TEX2D_SAMPLER(name,samplername,uv)	SAMPLE_TEXTURE2D(textureName, samplerName, coord2)

Table 4. Examples of relevant Cg macros and their HLSL equivalents

4.3 LightMode tags

LightMode tags define the role of Pass in the lighting pipeline. For custom shaders that are in the built-in pipeline, the LightMode tags must specify how the pass is considered in the lighting pipeline.

To migrate your LightMode tags to the URP, replace them with their equivalent URP tags.

The following table shows the equivalent LightMode tags that are used in the **built-in pipeline** and the **URP tags**:

Built-in	Description	URP
Always	Always rendered; no lighting is applied	Not supported
ForwardBase	Used in Forward rendering. Ambient, main directional light, vertex/ SH lights and lightmaps are applied.	UniversalForward
ForwardAdd	Used in Forward rendering. Additive per-pixel lights are applied, one pass per light.	Not supported
Deferred	Used in Deferred Shading; renders g-buffer	UniversalGBuffer
ShadowCaster	Renders object depth into the shadow map or a depth texture.	ShadowCaster

Built-in	Description	URP
MotionVectors	Used to calculate per-object motion vectors	Not supported yet
No equivalent	The URP uses this tag value in the Forward Rendering Path. The Pass renders object geometry and evaluates all light contributions.	UniversalForwardOnly
No equivalent	The URP uses this tag value in the 2D Renderer. The Pass renders objects and evaluates 2D light contributions.	Universal2D
No equivalent	The Pass renders only depth information from the perspective of a Camera into a depth texture.	DepthOnly
No equivalent	This Pass is executed only when baking lightmaps in the Unity Editor. Unity strips this Pass from shaders when building a Player.	Meta
No equivalent	Use this tag value to draw an extra Pass when rendering objects. It is valid for both the Forward and the Deferred Rendering Paths. The URP uses this tag value as the default value when a Pass does not have a LightMode tag.	SRPDefaultUnlit

Table 5. LightMode tags used in the built-in pipeline and the equivalent URP tags

Note: Several legacy built-in tags are not supported at all in the URP: MotionVectors, PrepassBase, PrepassFinal, Vertex, VertexLMRGBM, and VertexLM. Also, some tags are only in URP and have no equivalent in the built-in pipeline.

4.4 Changing replacement shaders

The URP does not support a replacement shader. You can use a replacement renderer to implement the behavior of a replacement shader.

The project **Dynamic Soft Shadows Based on Local Cubemaps** was originally written for the built-in pipeline. This project uses a replacement shader to render the chess-pieces with the very simple Custom/ctShadowMap shader. This means that the shadow camera is placed at the light position and uses the Custom/ctShadowMap shader to render the geometry of the chess pieces to a texture. The resulting texture is later projected on the chessboard to produce the shadows from the chess pieces, as shown in the following two images:

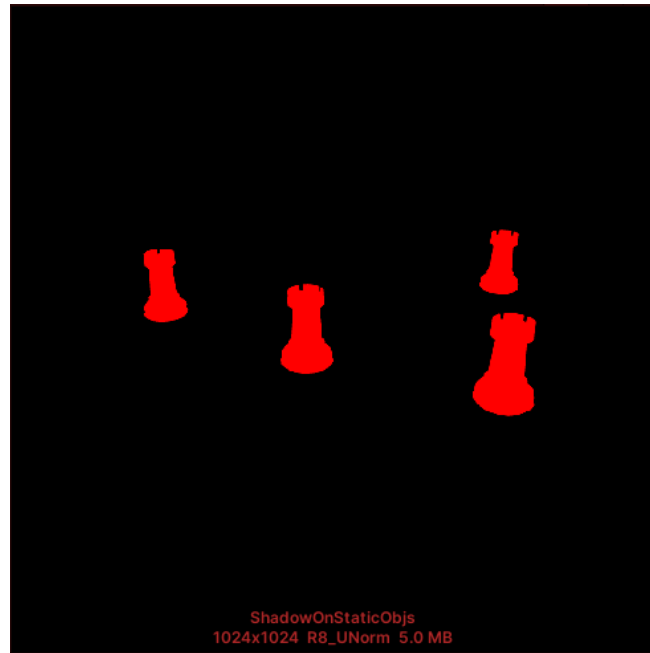


Image 2: Chess pieces rendered to texture with the replacement shader



Image 3: Projected shadows using the rendered texture

To use a replacement renderer to implement the behavior of a replacement shader:

1. Create a new URP renderer asset: right-click anywhere in the Project view, then select **Create > Rendering > Universal Render Pipeline > Forward Renderer**.
2. Name the Forward Renderer. For example, CameraReplacementRenderer.

3. Click on the **CameraReplacementRenderer** asset to edit its properties. This is shown in the following screenshot:

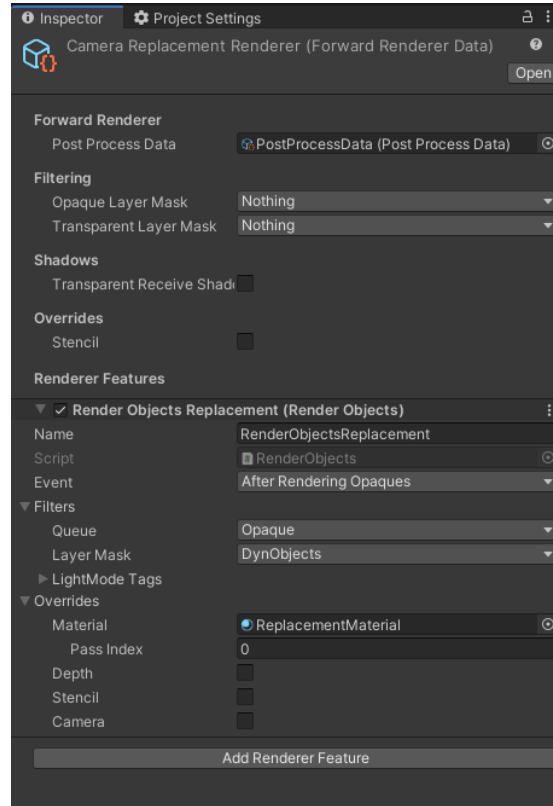


Image 4: Properties of a Forward Renderer

4. Set both **Opaque Layer Mask** and **Transparent Layer Mask** to **Nothing**. This changes the filtering and disables normal camera rendering, otherwise we would render everything twice.
5. Click **Add Renderer Feature** and select **Render Objects (experimental)**.
6. Click the **New Render Objects (Render Objects)** that you just created. There are several options listed, and the following describes what happens if you set filters or overrides:
 - Set **Filters > Layer Mask** to the layers you want it to render.
 - Set **Overrides > Material** to a material that is using your replacement shader.
7. Select **Assets/Settings/UniversalRP-HighQuality.asset**, click **+** to add a new renderer, and drag and drop the **CameraReplacementRenderer** asset. This is shown in the following screenshot:

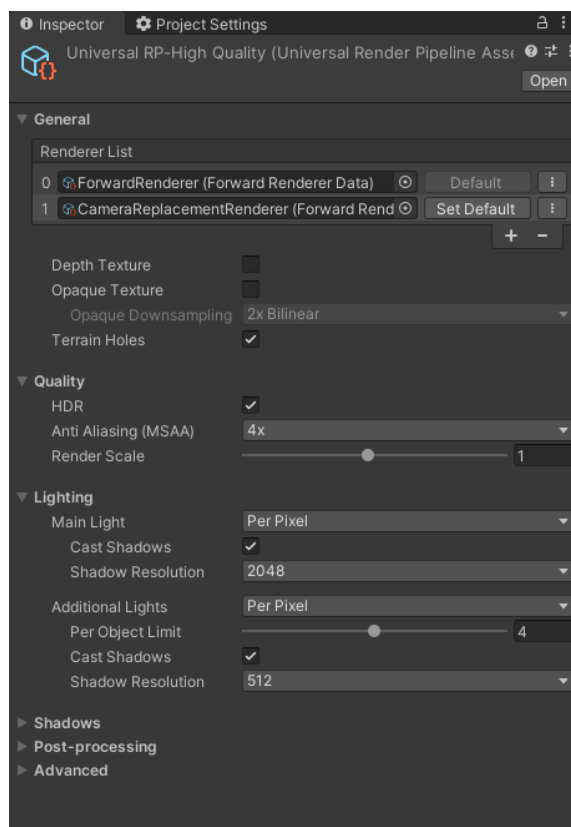


Image 5: UniversalRP-HighQuality.asset

8. Select the camera that was using the replacement shader.
9. Set Renderer to the new URP renderer asset that you configured for the CameraReplacementRenderer. This selection is shown in the following screenshot:

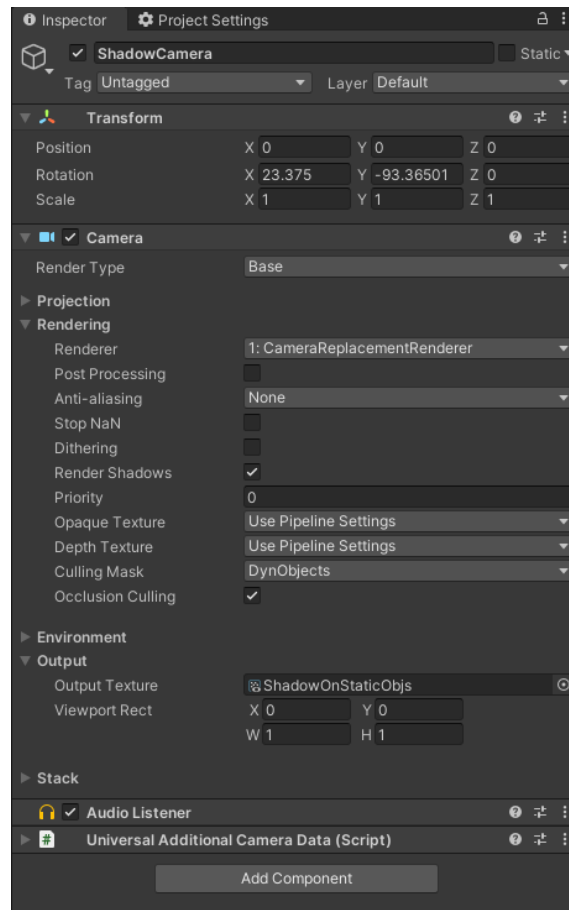


Image 6: Camera using replacement shader

4.5 Post-processing

When migrating a project to the URP, use the post-processing solution that Unity created for the URP, rather than the Post Processing v2 stack. See the [URP post-processing documentation](#) for more information.

4.5.1 If you are using custom post-processing and Unity 2019.4

In 2019, Unity dropped support for PPv2 in the URP and relied on the new integrated post-processing stack. However, the new stack did not include the custom post-processing feature. Unity therefore added support for PPv2 as a fallback mode in 2019.4 LTS, for users who do not want to upgrade to 2020.

To use custom post-processing in 2019.4 LTS, change the **Post Processing > Feature Set** from the integrated solution to **Post Processing V2**.

4.5.2 Mobile-friendly features

Post-processing is an expensive operation on mobile, because these effects can take up lots of frame time. If you are using URP for mobile devices, the following effects are the most mobile-friendly:

- Bloom. For mobile, disable the High-Quality Filtering option to reduce resource use.
- Chromatic Aberration
- Color Grading
- Lens Distortion
- Vignette

5 Related information

Here are some resources related to material in this guide:

- [Dynamic soft shadows based on local cubemaps](#)
- [LightMode tags](#)
- [Unity Render Pipeline tutorials](#)
- [Unity Universal Render Pipeline support](#)
- [What's new in URP](#)

6 Next steps

This guide has shown you how to migrate your built-in pipeline to Universal Render Pipeline and the advantages of doing this. To make the best use of URP in your game, use the [Unity URP microsite](#) to explore URP concepts and features. To optimize your game's performance further, see our guide [Optimization opportunities in Unity](#).