Introduction

This is a guide to convert a custom shader from Built-in RP to Universal RP (URP) compatibility.

The project with the shader using the Built-in RP is located under the **CustomShaderBuiltIn** directory. The final result project with the shader converted to have URP compatibility is located under the **CustomShaderURP** directory, so you can use it as a reference while following the guide.

This guide assumes basic familiarity with writing vertex/fragment shaders using the Built-in RP and ShaderLab, and basic familiarity with diffuse and specular lighting.

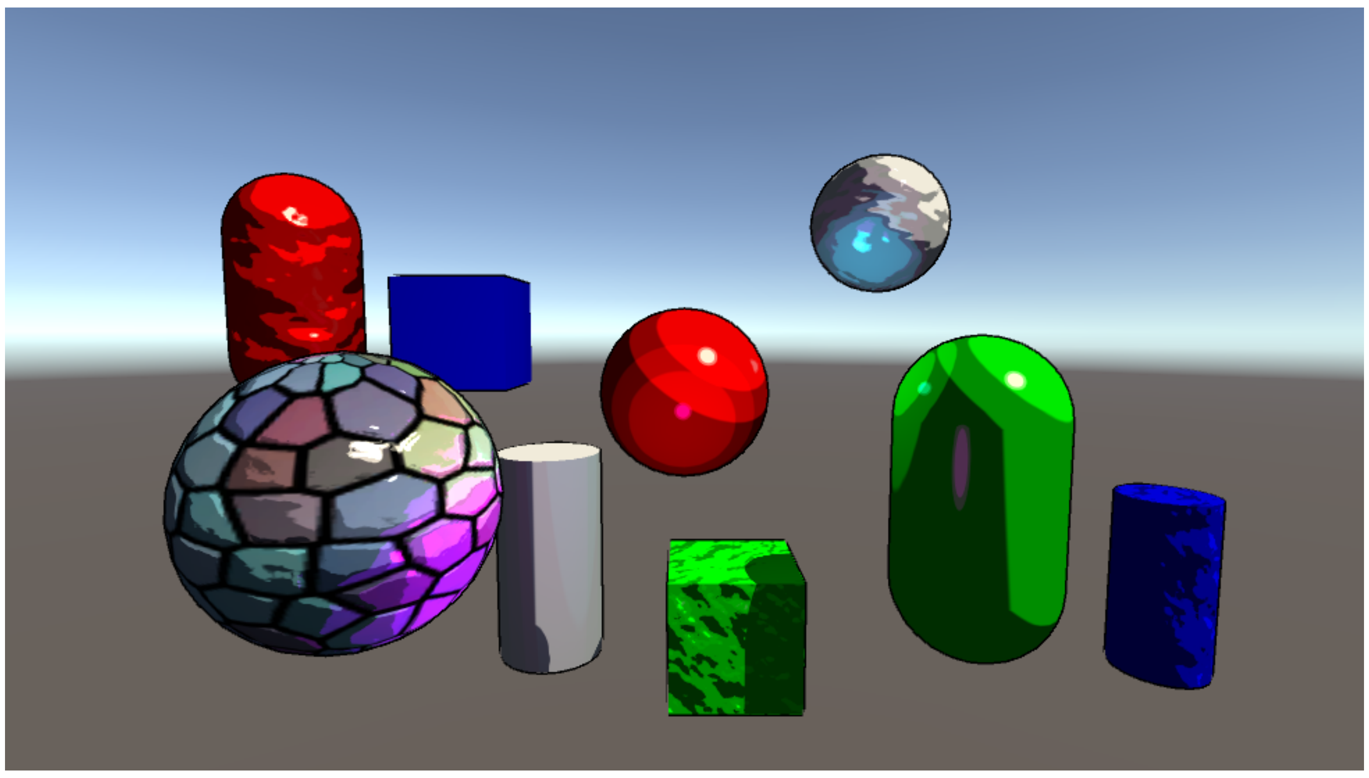
The Unity version used is 2019.4.25f1, with URP 7.5.2.

Original Project with Built-in RP

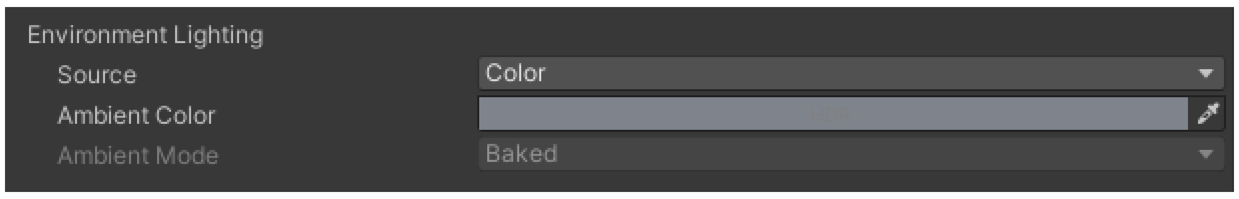
Let's open the project that we want to convert, and see what our final result should end up looking like.

- Open the **CustomShaderBuiltIn** project.
- Open the SampleScene scene under **Scenes/SampleScene**.

You should see a scene with several shapes of varying colors, with toon shading and black outlines.



- The scene contains 1 directional light (default light color), and 2 point lights (cyan and magenta).
- In the scene's Lighting settings, see that the Environment Lighting is set to use the default solid gray color (R=54, G=58, B=66).



Environment Lighting
Source: Color
Ambient Color
Ambient Mode: Baked

The shader we will be converting is located under **Shaders/ToonOutline.shader**.

- The shader is a basic toon shader which uses a simple texture for the shading ramp.
- The shader supports both the main light and additional lights, using both the **ForwardBase** pass and **ForwardAdd** pass.
- The lighting pass is included in a separate .cginc file under **Shaders/Inc/LightingToonPass.cginc**. The lighting functions are included in **Shaders/Inc/LightingToon.cginc**.
- The shader supports ambient lighting, diffuse lighting, and specular highlights.
- The shader supports normal maps.
- The shader supports casting and receiving shadows, for both the main light and additional lights. Shadow casting is done in the **ShadowCaster** pass.
- The outline is vertex-based, and is done in a separate **Outline** pass.
- The shader supports fog.
- For simplicity, baked lightmaps are NOT supported.

The materials used by the shapes are located under **Materials**. Most of the materials are simply a solid color, but some have textures and normal maps assigned. The textures are located under **Textures**.

You can modify the scene and materials to get a feel of how the shader is supposed to look.

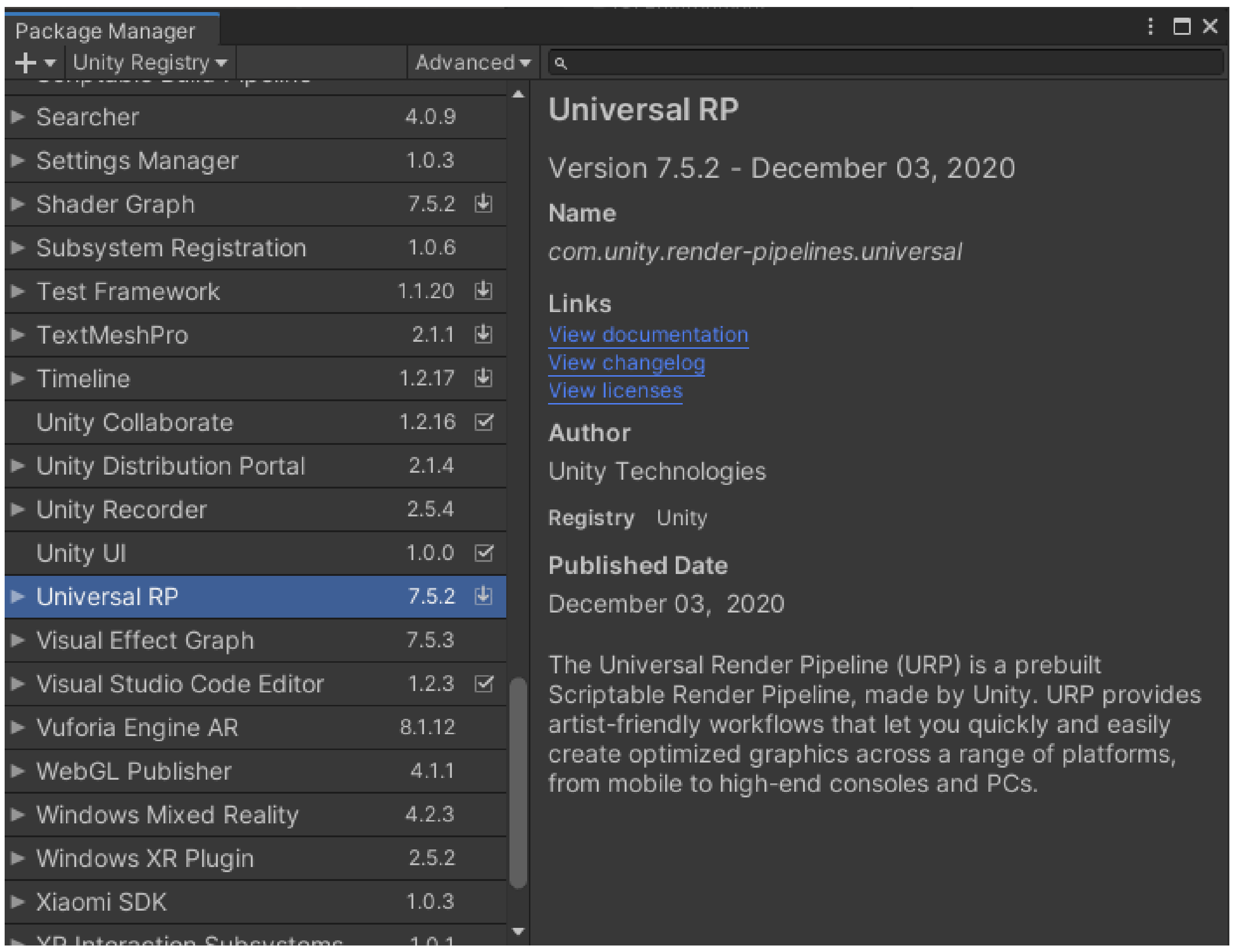
Set Up URP

We will switch the project to use URP instead of the Built-in RP.

First, install the **Universal RP** package.

- Open the Package Manager from **Window -> Package Manager**.

- Install the **Universal RP** package.



Package Manager
Unity Registry ▾ Advanced ▾

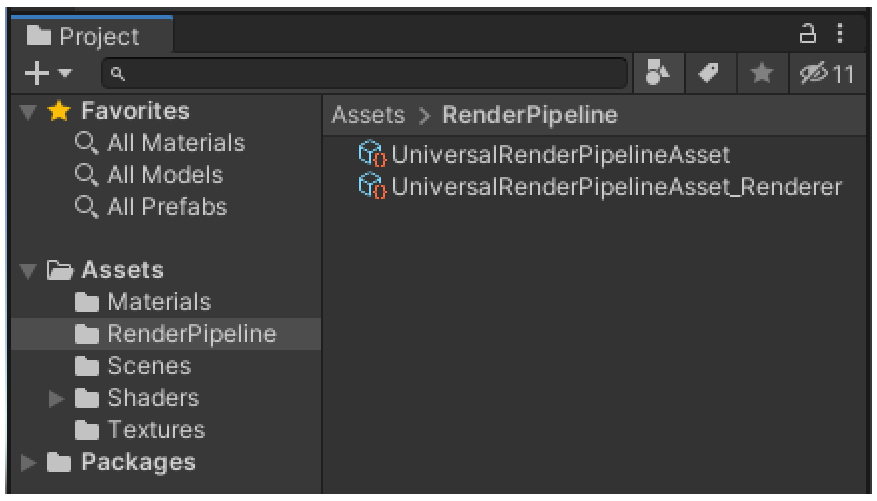
Searcher	4.0.9
Settings Manager	1.0.3
Shader Graph	7.5.2
Subsystem Registration	1.0.6
Test Framework	1.1.20
TextMeshPro	2.1.1
Timeline	1.2.17
Unity Collaborate	1.2.16
Unity Distribution Portal	2.1.4
Unity Recorder	2.5.4
Unity UI	1.0.0
Universal RP	7.5.2
Visual Effect Graph	7.5.3
Visual Studio Code Editor	1.2.3
Vuforia Engine AR	8.1.12
WebGL Publisher	4.1.1
Windows Mixed Reality	4.2.3
Windows XR Plugin	2.5.2
Xiaomi SDK	1.0.3

Universal RP
Version 7.5.2 - December 03, 2020
Name
com.unity.render-pipelines.universal
Links
View documentation
View changelog
View licenses
Author
Unity Technologies
Registry Unity
Published Date
December 03, 2020

The Universal Render Pipeline (URP) is a prebuilt Scriptable Render Pipeline, made by Unity. URP provides artist-friendly workflows that let you quickly and easily create optimized graphics across a range of platforms, from mobile to high-end consoles and PCs.

We can now switch the project's render pipeline to URP.

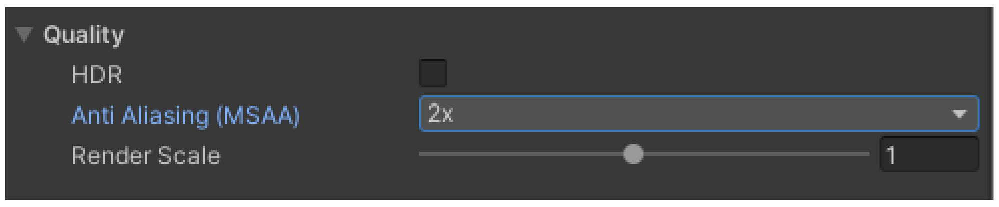
- Create a new folder under Assets, and name it **RenderPipeline**.
- In this folder, right-click and select **Create -> Rendering -> Universal Render Pipeline -> Pipeline Asset (Forward Renderer)**.
- This will create a new **UniversalRenderPipelineAsset**, along with a default **UniversalRenderPipelineAsset_Renderer**.



Project
Favorites: All Materials, All Models, All Prefabs
Assets: Materials, RenderPipeline, Scenes, Shaders, Textures
Packages

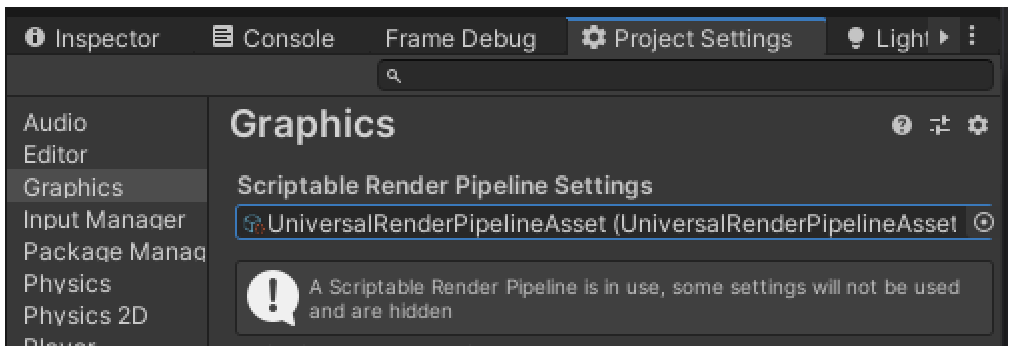
Assets > RenderPipeline
UniversalRenderPipelineAsset
UniversalRenderPipelineAsset_Renderer

- Anti aliasing is disabled by default in URP, so you can optionally enable it by selecting the UniversalRendererPipelineAsset in the inspector, and changing the **Quality -> Anti Aliasing** setting.



Quality
HDR ☐
Anti Aliasing (MSAA) 2x
Render Scale 1

- Open the Graphics settings from **Edit -> Project Settings -> Graphics**.
- Assign the UniversalRenderPipelineAsset to the **Scriptable Renderer Pipeline Settings** to change the project's render pipeline to URP.

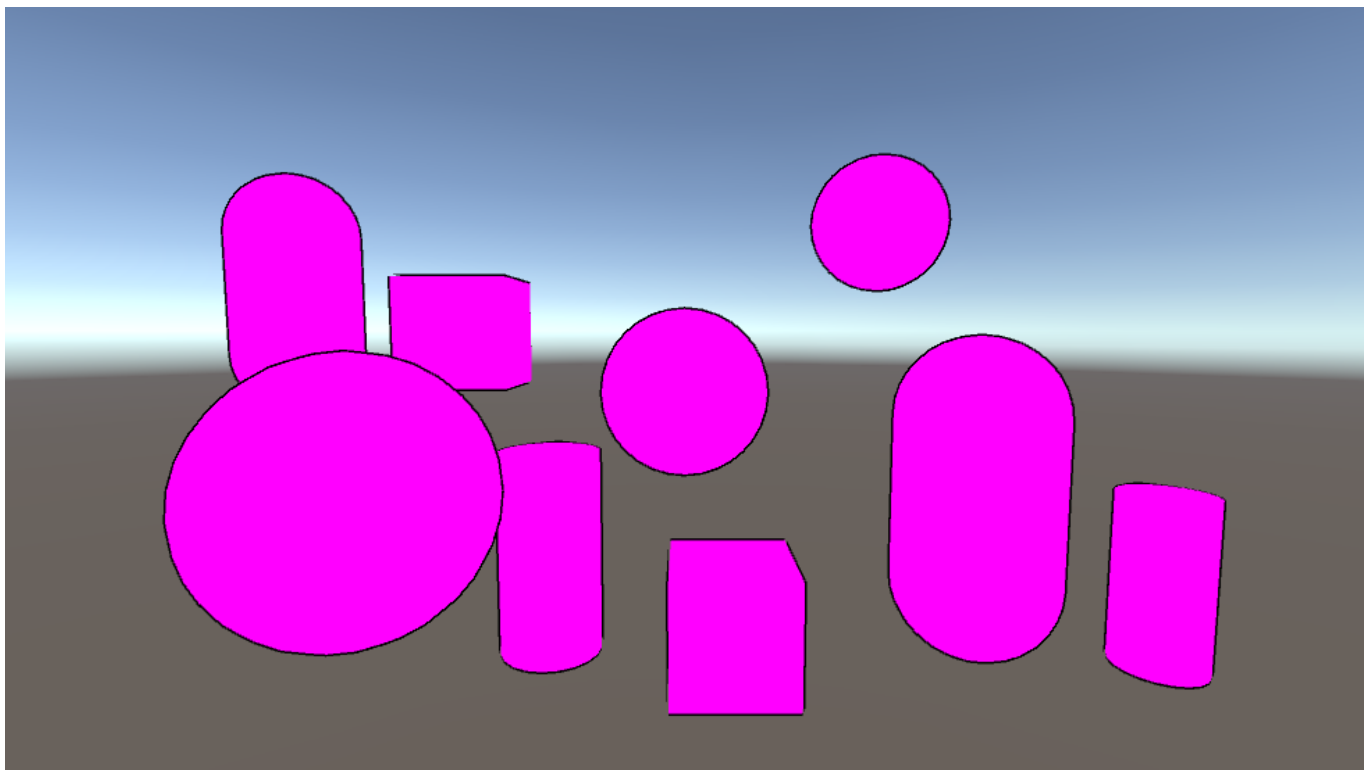


Inspector | Console | Frame Debug | Project Settings | Light

Audio, Editor, Graphics, Input Manager, Package Manag, Physics, Physics 2D

Graphics
Scriptable Render Pipeline Settings
UniversalRenderPipelineAsset (UniversalRenderPipelineAsset
A Scriptable Render Pipeline is in use, some settings will not be used and are hidden

The project's render pipeline is now set to URP. Since the ToonOutline shader is not yet compatible with URP, the scene objects are rendered with the error shader instead (with the exception of the Outline pass).



Open Shader Files

From here, we will modify the ToonOutline shader to be compatible with URP.

Open the following files in an IDE:

- **Shaders/ToonOutline.shader**: This is the shader file for the ToonOutline shader. It contains ForwardBase and ForwardAdd passes for the main light and additional lights, an Outline pass for the outline, and a ShadowCaster pass to support shadow casting. The lighting pass is included in LightingToonPass.cginc.
- **Shaders/Inc/LightingToonPass.cginc**: This is the lighting pass used by the ForwardBase and ForwardAdd passes. It contains the vertex input and interpolator definitions, and the vertex/fragment function definitions. The lighting functions are included in LightingToon.cginc.
- **Shaders/Inc/LightingToon.cginc**: This contains the toon lighting functions used by the LightingToonPass.cginc. It contains a basic toon lighting implementation for ambient light, diffuse lighting, and specular highlights. A texture is used for the toon ramp.

Modify Required Tags for URP

We will add the tags required for URP to the SubShader blocks and Passes.

First, tell Unity to use URP for the shader's SubShader block.

- In ToonOutline.shader, locate the SubShader tags.
- Add a **RenderPipeline** tag, and set the value to **UniversalPipeline** to tell Unity to use URP for this SubShader block.

ToonOutline.shader

```
/* Add RenderPipeline tag to SubShader tags */

SubShader
{
    ~~Tags { "Queue" = "Geometry" "RenderType" = "Opaque" }~~

    Tags { "Queue" = "Geometry" "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }

    /* ... */
```

Next, change the LightMode tag and Name in the current ForwardBase pass.

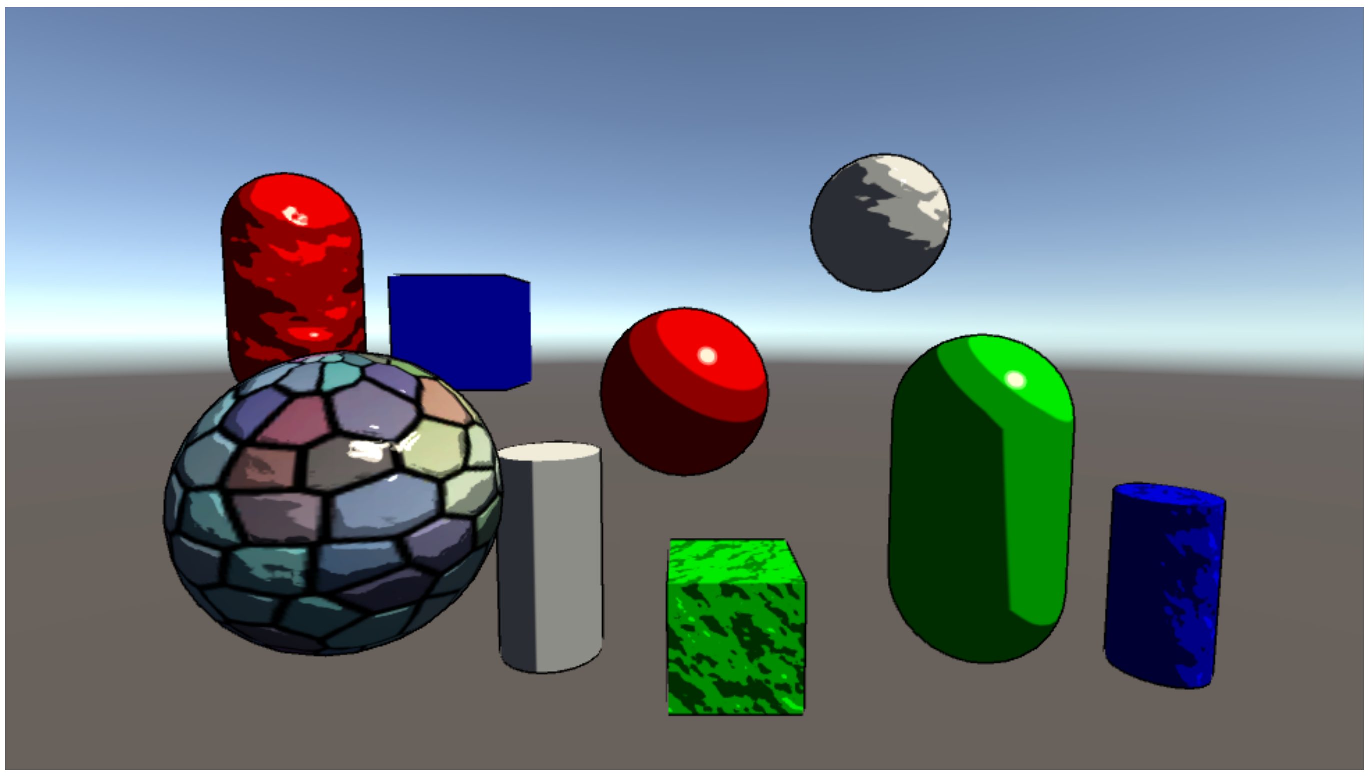
- In ToonOutline.shader, locate the ForwardBase pass.
- Change the **LightMode** tag to **UniversalForward**.
- Change the **Name** from ForwardBase to **Forward** (URP uses a single pass for both the main light and additional lights, so the ForwardAdd pass will later be removed).

ToonOutline.shader

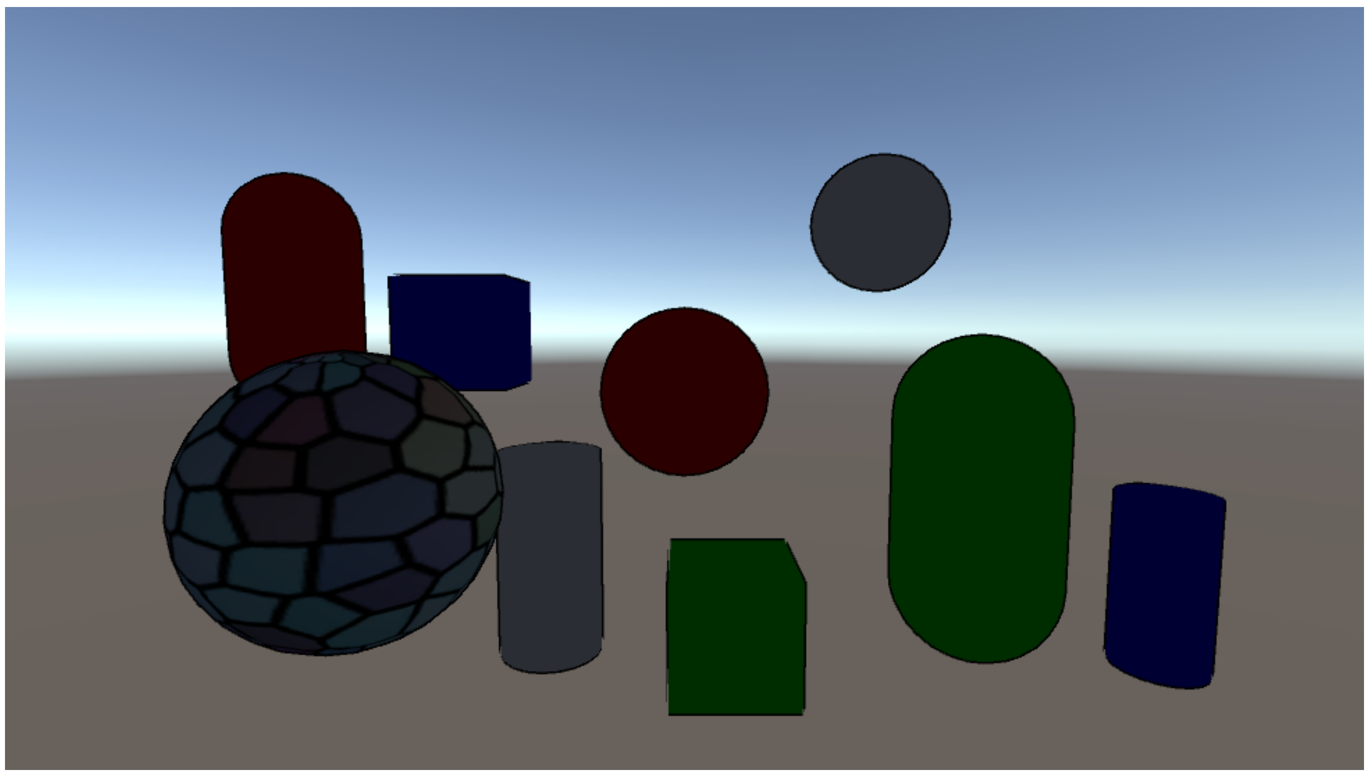
```
/* Change LightMode to UniversalForward, and Name to Forward for ForwardBase pass */

~~// Forward base pass~~
// Forward pass
Pass
{
    ~~Name "ForwardBase"~~
    ~~Tags { "LightMode" = "ForwardBase" }~~

    Name "Forward"
    Tags { "LightMode" = "UniversalForward" }
    /* ... */
```

This is actually enough to get the main light working, since some compatibility with the Built-in RP is maintained. However, you will notice that:

- Additional lights (the point lights) are not working, since the Built-in RP uses one ForwardAdd pass per additional light, but URP uses one UniversalForward pass for all lights.
- Shadow casting and receiving is not working.



NOTE: It's possible that the objects end up not lighted, like this:



This appears to be because the UNITY_SHADOW_ATTENUATION macro is always returning 0 for the shadow attenuation value that is used for the lighting calculation (which means only the ambient light is seen).

- If this happens, change the UNITY_SHADOW_ATTENUATION macro to SHADOW_ATTENUATION instead in LightingToonPass.cginc.

LightingToonPass.cginc

```
/* Change UNITY_SHADOW_ATTENUATION to SHADOW_ATTENUATION if objects appear not lighted */

// Fragment function
fixed4 fragToon(v2f_toon i) : SV_Target
{
    /* ... */

    // Construct light data
    UnityLight light;
    light.color = _LightColor0;
#if defined(POINT) || defined(SPOT)
    light.dir = normalize(_WorldSpaceLightPos0.xyz - i.positionWS);  // Direction to light source
#else
    light.dir = _WorldSpaceLightPos0.xyz;  // Pos is direction for directional lights
#endif
    float distanceAttenuation = DistanceAttenuation(i.positionWS);  // Distance attenuation
    float shadowAttenuation = SHADOW_ATTENUATION(i); ~~UNITY_SHADOW_ATTENUATION(i, i.positionWS);~~ // Shadow attenuation

    /* ... */
}
```

Remove ForwardAdd Pass

From here, we will modify the shader to remove dependence on legacy shader functions and abide by URP standards.

As mentioned earlier, URP uses one pass for both the main light and additional lights. This means the ForwardAdd pass is not needed.

- Remove the ForwardAdd pass from ToonOutline.shader.

ToonOutline.shader

```
/* Remove ForwardAdd pass */

~~// Forward add pass~~
~~Pass~~
~~{~~
    ~~Name "ForwardAdd"~~
    ~~Tags { "LightMode" = "ForwardAdd" }~~

    ~~Blend One One~~
    ~~ZWrite Off // Don't write to depth twice~~

    ~~CGPROGRAM~~

    ~~#pragma target 2.0~~

    ~~// Vertex/fragment functions~~
    ~~#pragma vertex vertToon~~
    ~~#pragma fragment fragToon~~

    ~~// GPU Instancing~~
    ~~#pragma multi_compile_instancing~~
    ~~// Fog~~
    ~~#pragma multi_compile_fog~~
    ~~// Lighting variations~~
    ~~#pragma multi_compile_fwdadd_fullshadows~~
    ~~#pragma skip_variants DIRECTIONAL_COOKIE POINT_COOKIE SPOT_COOKIE~~

    ~~// Unity includes~~
    ~~#include "UnityCG.cginc"~~

    ~~// Pass includes~~
    ~~#include "Inc/LightingToonPass.cginc"~~

    ~~ENDCG~~
~~}~~
```

Remove ShadowCaster Pass

A ShadowCaster pass is required in order to cast shadows, but we will be rewriting it later.

- For now, remove the ShadowCaster pass from ToonOutline.shader.

ToonOutline.shader

```
/* Remove ShadowCaster pass */

~~// ShadowCaster pass~~
~~Pass~~
~~{~~
    ~~Name "ShadowCaster"~~
    ~~Tags { "LightMode" = "ShadowCaster" }~~

    ~~CGPROGRAM~~

    ~~#pragma target 2.0~~

    ~~// Vertex/fragment functions~~
    ~~#pragma vertex vert~~
    ~~#pragma fragment frag~~

    ~~// GPU Instancing~~
    ~~#pragma multi_compile_instancing~~

    ~~// Unity includes~~
    ~~#include "UnityCG.cginc"~~

    ~~// Vert input~~
    ~~struct appdata~~
    ~~{~~
        ~~float4 vertex        : POSITION;     // Object space position~~
        ~~float3 normalOS      : NORMAL;       // Object space normal~~

        ~~UNITY_VERTEX_INPUT_INSTANCE_ID~~
    ~~};~~

    ~~// Vert output/Frag input~~
    ~~struct v2f~~
    ~~{~~
        ~~float4 pos   : SV_POSITION; // Clip space position~~
    ~~};~~

    ~~// Vertex function~~
    ~~v2f vert(appdata v)~~
    ~~{~~
        ~~v2f o;~~

        ~~UNITY_SETUP_INSTANCE_ID(v);~~

        ~~// Transformations and shadow bias~~
        ~~float4 positionCS = UnityClipSpaceShadowCasterPos(v.vertex.xyz, v.normalOS);~~
        ~~positionCS = UnityApplyLinearShadowBias(positionCS);~~

        ~~// Set output~~
        ~~o.pos = positionCS;~~

        ~~return o;~~
    ~~}~~

    ~~// Fragment function~~
    ~~fixed4 frag(v2f i) : SV_Target~~
    ~~{~~
        ~~return 0;~~
    ~~}~~

    ~~ENDCG~~
~~}~~
```

Remove Legacy Shader Macros and Keywords

To further remove dependency on legacy shader code, we will remove references to legacy shader macros and keywords.

First, the Forward pass uses legacy keywords, so remove them.

- In ToonOutline.shader, remove the **SHADOWS_SCREEN** keyword from the Forward pass.

ToonOutline.shader

```
/* Remove legacy SHADOWS_SCREEN keyword from Forward pass*/

// Forward pass
Pass
{
    /* ... */

    ~~// Shadows~~
    ~~#pragma multi_compile _ SHADOWS_SCREEN~~

    /* ... */
```

Next, we will remove legacy macros.

- In ToonOutline.shader, remove any instances of the following macros:
    - UNITY_INITIALIZE_OUTPUT
    - UNITY_FOG_COORDS
    - UNITY_TRANSFER_FOG
    - UNITY_APPLY_FOG

ToonOutline.shader

```
/* Remove UNITY_INITIALIZE_OUTPUT, UNITY_FOG_COORDS, UNITY_TRANSFER_FOG, UNITY_APPLY_FOG macros from Forward pass and Outline pass */

~~UNITY_INITIALIZE_OUTPUT~~
~~UNITY_FOG_COORDS~~
~~UNITY_TRANSFER_FOG~~
~~UNITY_APPLY_FOG~~
```

- Similarly, In LightingToonPass.cginc, remove any instances of the following macros:
    - UNITY_INITIALIZE_OUTPUT
    - UNITY_SHADOW_COORDS
    - UNITY_FOG_COORDS
    - UNITY_TRANSFER_SHADOW
    - UNITY_TRANSFER_FOG
    - UNITY_SHADOW_ATTENUATION or SHADOW_ATTENUATION
    - UNITY_APPLY_FOG

LightingToonPass.cginc

```
/* Remove UNITY_INITIALIZE_OUTPUT, UNITY_SHADOW_COORDS, UNITY_FOG_COORDS, UNITY_TRANSFER_SHADOW, UNITY_TRANSFER_FOG,
UNITY_SHADOW_ATTENUATION, UNITY_APPLY_FOG macros */

~~UNITY_INITIALIZE_OUTPUT~~
~~UNITY_SHADOW_COORDS~~
~~UNITY_FOG_COORDS~~
~~UNITY_TRANSFER_SHADOW~~
~~UNITY_TRANSFER_FOG~~
~~UNITY_SHADOW_ATTENUATION~~
~~UNITY_APPLY_FOG~~
```

Remove Legacy .cginc Includes

Remove any legacy .cginc includes. We will be replacing these with URP's .hlsl includes shortly.

- In ToonOutline.shader, remove all **UnityCG.cginc** includes.

ToonOutline.shader

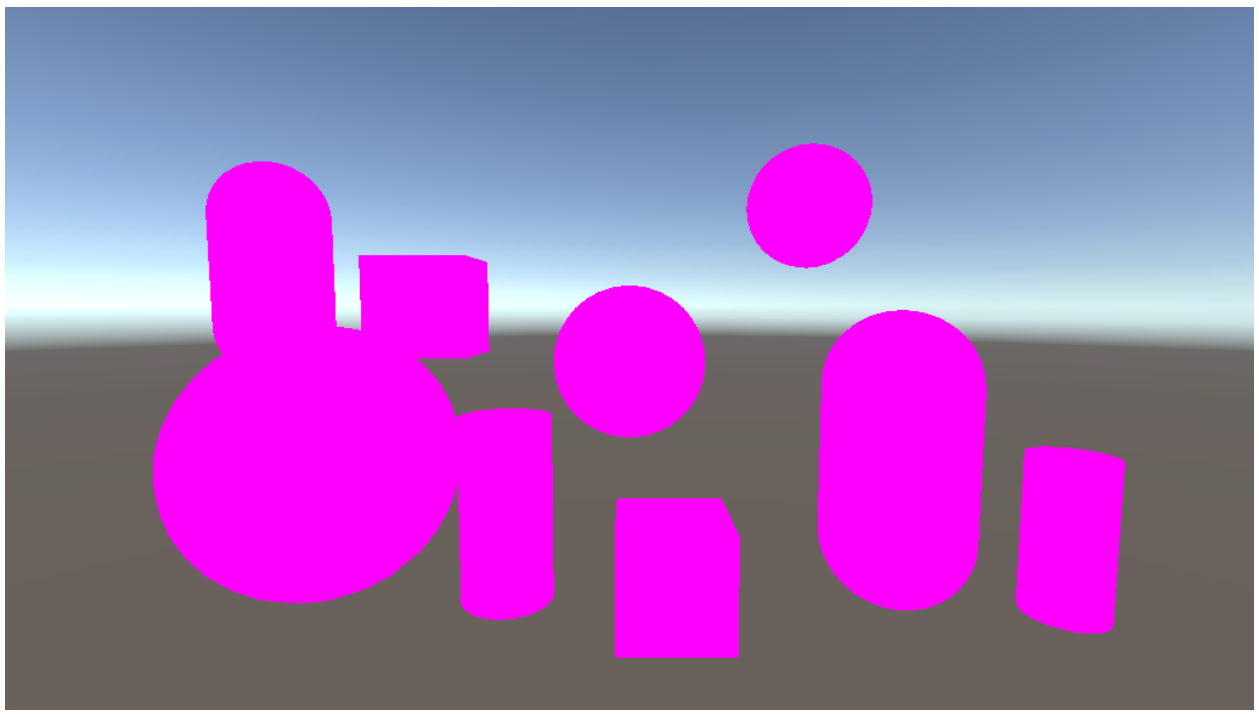
```
/* Remove UnityCG.cginc includes from Forward pass and Outline pass */

~~// Unity includes~~
~~#include "UnityCG.cginc"~~
```

- In LightingToonPass.cginc, remove the **UnityLightingCommon.cginc** and **AutoLight.cginc** includes.

LightingToonPass.cginc

```
~~// Unity includes~~
~~#include "UnityLightingCommon.cginc"~~
~~#include "AutoLight.cginc"~~
```

Since we have removed the shader's legacy dependencies, the error material is now shown.



Add Compiler Directive to Compile GLES 2.0 with SRP Library

From here, we will set up the shader for URP compatibility.

URP shaders use a specific compiler directive to compile GLES 2.0 with the standard SRP library. Most URP shaders include this directive, so we will add it.

- In ToonOutline.shader, add the **prefer_hlslcc gles** directive before the target directive.

ToonOutline.shader

```
/* Add to Forward pass and Outline pass before target directive */

// Required to compile gles 2.0 with standard SRP library
#pragma prefer_hlslcc gles
#pragma target 2.0
```

Change CGPROGRAM to HLSLPROGRAM

SRP uses HLSLPROGRAM and .hlsl files, as opposed to the legacy shaders which use CGPROGRAM and .cginc files. While these are functionally identical (Unity uses HLSL), we will match SRP's standards.

- In ToonOutline.shader, replace CGPROGRAM/ENDCG with **HLSLPROGRAM/ENDHLSL** for all passes.

ToonOutline.shader

```
/* Replace CGPROGRAM/ENDCG with HLSLPROGRAM/ENDHLSL in all passes */

~~CGPROGRAM~~
/* ... */
~~ENDCG~~

HLSLPROGRAM
/* ... */
ENDHLSL
```

Change Custom .cginc Files to .hlsl Files

Similarly, we will change the custom .cginc includes to .hlsl.

- Rename Inc/LightingToonPass.cginc to **Inc/LightingToonPass.hlsl**.
- Rename Inc/LightingToon.cginc to **Inc/LightingToon.hlsl**.

Also, rename the references to the custom includes in the shader code.

- Rename Inc/LightingToonPass.cginc includes to **Inc/LightingToonPass.hlsl** includes.
- Rename Inc/LightingToon.cginc includes to **Inc/LightingToon.hlsl** includes.

ToonOutline.shader

```
/* Change custom .cginc includes to .hlsl in Forward pass */

~~// Pass includes~~
~~#include "Inc/LightingToonPass.cginc"~~

// Pass includes
#include "Inc/LightingToonPass.hlsl"
```

LightingToonPass.hlsl

```
/* Change custom .cginc includes to .hlsl */
```

```
~~// Custom includes~~
~~#include "LightingToon.cginc"~~

// Custom includes
#include "LightingToon.hlsl"
```

Include URP’s Common Functionality with Core.hlsl

Similar to UnityCG.cginc, URP’s most common functionality is contained in Core.hlsl, so we will include it.

- In ToonOutline.shader, include URP’s **Core.hlsl** in all passes.

ToonOutline.shader

```
/* Remove UnityCG.cginc includes from Forward pass and Outline pass */

~~// Unity includes~~
~~#include "UnityCG.cginc"~~

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

Replace fixed Variables with half Variables

The fixed type is no longer supported in SRP, so we will replace it with the half type.

- In ToonOutline.shader, LightingToonPass.hlsl, and LightingToon.hlsl, replace all **fixed** variables with **half** variables (a simple find/replace will suffice).

Rename Vertex Input and Interpolators

URP calls vertex input and interpolators Attributes and Varyings respectively, so we will match this standard.

- In ToonOutline.shader, rename the appdata (vertex input) struct to **Attributes** in the Outline pass.
- Additionally, rename the v2f (interpolator) struct to **Varyings** in the Outline pass.

ToonOutline.shader

```
/* Rename vertex and interpolators in Outline pass */

/* Rename appdata struct to Attributes */

~~// Vert input~~
~~struct appdata { /* ... */ }~~

// Vert input
struct Attributes { /* ... */ }

/* Rename v2f struct to Varyings */

~~// Vert output/Frag input~~
~~struct v2f { /* ... */ }~~

// Vert output/Frag input
struct Varyings { /* ... */ }

/* Change vertex function to take Attributes parameter, and return Varyings output */

~~// Vertex function~~
~~v2f vert(appdata v) { /* ... */ }~~

// Vertex function
Varyings vert(Attributes input) { /* ... */ }

/* Change fragment function to take Varyings parameter */

~~// Fragment function~~
~~half4 frag(v2f i) : SV_Target { /* ... */ }~~

// Fragment function
half4 frag(Varyings input) : SV_Target { /* ... */ }
```

We will do the same for the Forward pass.

- In LightingToonPass.hlsl, rename the appdata_toon (vertex input) struct to **AttributesToon**.
- Additionally, rename the v2f_toon (interpolator) struct to **VaryingsToon**.

LightingToonPass.hlsl

```
/* Rename vertex and interpolators in Forward pass */

/* Rename appdata_toon struct to AttributesToon */

~~// Vert input~~
~~struct appdata_toon { /* ... */ }~~

// Vert input
struct AttributesToon { /* ... */ }

/* Rename v2f_toon struct to VaryingsToon */

~~// Vert output/Frag input~~
~~struct v2f_toon { /* ... */ }~~

// Vert output/Frag input
struct VaryingsToon { /* ... */ }

/* Change vertex function to take AttributesToon parameter, and return VaryingsToon output */

~~// Vertex function~~
~~v2f_toon vertToon(appdata_toon v) { /* ... */ }~~

// Vertex function
VaryingsToon vertToon(AttributesToon input) { /* ... */ }

/* Change fragment function to take VaryingsToon parameter */

~~// Fragment function~~
~~half4 fragToon(v2f_toon i) : SV_Target { /* ... */ }~~

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target { /* ... */ }
```

Declare Properties Inside CBUFFER block

SRP render pipelines (including URP) have support for the SRP Batcher.

- See https://docs.unity3d.com/Manual/SRPBatcher.html for more information.

In order for a shader to be compatible with the SRP Batcher, the shader's properties must be declared inside a **CBUFFER** block with the **UnityPerMaterial** name.

- See: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@10.3/manual/writing-shaders-urp-unlit-color.html for more information.

To have our shader be compatible with the SRP Batcher, we will declare our properties inside a CBUFFER block.

- In LightingToonPass.hlsl, put the property declarations inside a **CBUFFER_START/CBUFFER_END** block with the **UnityPerMaterial** name.

LightingToonPass.hlsl

```
/* Put property declarations inside CBUFFER block for Forward pass */

// Properties
CBUFFER_START(UnityPerMaterial)
    half4 _Color;
    sampler2D _MainTex;
    float4 _MainTex_ST;

    sampler2D _ToonRampTex;

    float _ShadowRampBlend;

    sampler2D _NormalTex;

    half3 _SpecularColor;
    float _Smoothness;
CBUFFER_END
```

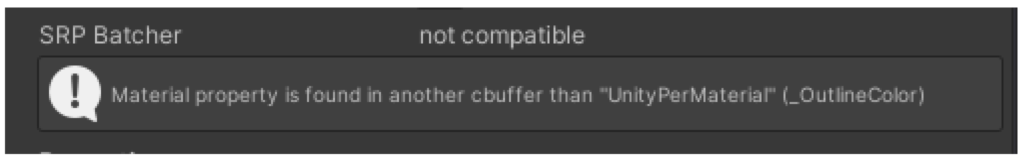
We will also move the texture and sampler declarations outside the CBUFFER block, and use macros provided by SRP.

- In LightingToonPass.hlsl, remove the **sampler2D** declarations from the CBUFFER block.
- Newly declare the textures and samplers after the CBUFFER block, using the **TEXTURE2D** and **SAMPLER** macros.

LightingToonPass.hlsl

```
/* Move texture and sampler declarations outside of CBUFFER block */

// Properties
CBUFFER_START(UnityPerMaterial)
    half4 _Color;
    ~~sampler2D _MainTex;~~
    float4 _MainTex_ST;

    ~~sampler2D _ToonRampTex;~~

    float _ShadowRampBlend;

    ~~sampler2D _NormalTex;~~

    half3 _SpecularColor;
    float _Smoothness;
CBUFFER_END

// Texture samplers
TEXTURE2D(_MainTex);
SAMPLER(sampler_MainTex);

TEXTURE2D(_ToonRampTex);
SAMPLER(sampler_ToonRampTex);

TEXTURE2D(_NormalTex);
SAMPLER(sampler_NormalTex);
```

At this point, if we look at the shader in the inspector, we can see that the shader is not yet compatible with the SRP Batcher, because the **_OutlineColor** property "is found in another CBUFFER other than 'UnityPerMaterial.'"


SRP Batcher   not compatible

Material property is found in another cbuffer than "UnityPerMaterial" (_OutlineColor)

This is because in order for a shader to be compatible with the SRP batcher, the CBUFFER block's properties must be the same across ALL passes. So we must include the Outline pass' _OutlineColor and _OutlineThickness properties in the Forward pass.

- In LightingToonPass.hlsl, add the **_OutlineColor** and **_OutlineThickness** properties to the CBUFFER block.

LightingToonPass.hlsl

```
/* Add _OutlineColor, _OutlineThickness to LightingToonPass properties */

// Properties (needs to be same across all passes for SRP batcher compatibility)
CBUFFER_START(UnityPerMaterial)
    half4 _Color;
    float4 _MainTex_ST;

    float _ShadowRampBlend;

    half3 _SpecularColor;
    float _Smoothness;

    half4 _OutlineColor;
    half _OutlineThickness;
CBUFFER_END
```

Since the CBUFFER block must be the same across all passes, we also need to declare the properties in the Outline pass.

- In ToonOutline.shader, add the same property declarations (in a CBUFFER block) to the Outline pass.

ToonOutline.shader

```
/* Declare the same properties in the Outline pass */

// Properties (needs to be same across all passes for SRP batcher compatibility)
CBUFFER_START(UnityPerMaterial)
    half4 _Color;
    float4 _MainTex_ST;
    float _ShadowRampBlend;

    half3 _SpecularColor;
    float _Smoothness;

    half4 _OutlineColor;
    half _OutlineThickness;
CBUFFER_END
```

If we look at the shader in the inspector, the SRP Batcher should now appear as compatible.

SRP Batcher compatible

Replace Space Transformation Functions

SRP uses different functions for space transformations than Built-in RP, we need to replace them.

- For example, Built-in RP's UnityObjectToClipPos function becomes **TransformObjectToHClip** in SRP.
    - See **Packages/com.unity.render-pipelines.core/ShaderLibrary/SpaceTransforms.hlsl** for SRP's space transformation functions.

- In ToonOutline.shader, replace the space transformation functions in the Outline pass with their SRP equivalents.

ToonOutline.shader

```
/* Replace space transformation functions in Outline pass */

// Vertex function
Varyings vert(Attributes input)
{
    /* ... */

    ~~// Transformations~~
    ~~float3 normalCS = mul((float3x3) UNITY_MATRIX_VP, mul((float3x3) UNITY_MATRIX_M, input.normalOS));~~
    ~~float4 positionCS = UnityObjectToClipPos(input.vertex);~~

    // Transformations
    float3 normalWS = TransformObjectToWorldNormal(input.normalOS);
    float3 normalCS = TransformWorldToHClipDir(normalWS);
    float4 positionCS = TransformObjectToHClip(input.vertex.xyz);

    /* ... */
}
```

- In LightingToonPass.hlsl, replace the space transformation functions in the Forward pass with their SRP equivalents.

LightingToonPass.hlsl

```
/* Replace space transformation functions in Forward pass */

// Vertex function
VaryingsToon vertToon(AttributesToon input)
{
    /* ... */

    ~~// Transformations~~
    ~~float3 positionWS = mul(unity_ObjectToWorld, input.vertex).xyz;~~
    ~~half3 normalWS = UnityObjectToWorldNormal(input.normalOS);~~
    ~~half3 tangentWS = UnityObjectToWorldDir(input.tangentOS.xyz);~~
    ~~half3 bitangentWS = cross(normalWS, tangentWS) * input.tangentOS.w;~~
    ~~float4 positionCS = UnityObjectToClipPos(input.vertex);~~

    // Transformations
    float3 positionWS = TransformObjectToWorld(input.vertex.xyz);
    half3 normalWS = TransformObjectToWorldNormal(input.normalOS);
    half3 tangentWS = TransformObjectToWorldDir(input.tangentOS.xyz);
    half3 bitangentWS = cross(normalWS, tangentWS) * input.tangentOS.w;
    float4 positionCS = TransformObjectToHClip(input.vertex.xyz);

    /* For viewDirectionWS, change _WorldSpaceCameraPos to GetCameraPositionWS function */

    ~~output.viewDirectionWS = normalize(_WorldSpaceCameraPos - positionWS);~~

    output.viewDirectionWS = normalize(GetCameraPositionWS() - positionWS);

    /* ... */
}
```

Rename Main Texture and Main Color Property Names

We will now move on to texture sampling.

For the Built-in RP, the main texture and main color property names are reserved as _MainTex and _Color, respectively. These are named differently in SRP.

- The main texture is named **_BaseMap** in SRP.
    - https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@10.3/manual/writing-shaders-urp-unlit-texture.html
- The main color is named **_BaseColor** in SRP.
    - https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@10.3/manual/writing-shaders-urp-unlit-color.html

We will change the main texture and main properties in the shader to match SRP standards.

- In ToonOutline.shader, rename _MainTex and _Color to **_BaseMap** and **_BaseColor** in the Properties block.

ToonOutline.shader

```
/* Rename _MainTex, _Color to _BaseMap, _BaseColor in Properties block */

~~// Main texture and tint~~
~~_Color("Main Color", Color) = (1, 1, 1, 1)~~
~~_MainTex("Main Tex", 2D) = "white" {}~~

// Main texture and tint
_BaseColor("Base Color", Color) = (1, 1, 1, 1)
_BaseMap("Base Map", 2D) = "white" {}
```

- In ToonOutline.shader and LightingToonPass.hlsl, rename the _MainTex and _Color properties to the new _BaseMap and _BaseColor.

ToonOutline.shader

```
/* Rename _MainTex, _Color to _BaseMap, _BaseColor in property declarations */

CBUFFER_START(UnityPerMaterial)
    ~~half4 _Color;~~
    ~~float4 _MainTex_ST;~~
    half4 _BaseColor;
    float4 _BaseMap_ST;

    /* ... */
CBUFFER_END
```

LightingToonPass.hlsl

```
/* Rename _MainTex, _Color to _BaseMap, _BaseColor in property declarations */
```

```
CBUFFER_START(UnityPerMaterial)
    ~~half4 _Color;~~
    ~~float4 _MainTex_ST;~~
    half4 _BaseColor;
    float4 _BaseMap_ST;

    /* ... */
CBUFFER_END

/* ... */

~~// Texture samplers~~
~~TEXTURE2D(_MainTex);~~
~~SAMPLER(sampler_MainTex);~~

// Texture samplers
TEXTURE2D(_BaseMap);
SAMPLER(sampler_BaseMap);
```

- In LightingToon.hlsl, rename any uses of _MainTex and _Color to _BaseMap and _BaseColor.

LightingToonPass.hlsl

```
/* Change any uses of _MainTex, _Color to _BaseMap, _BaseColor */

// Vertex function
VaryingsToon vertToon(AttributesToon input)
{
    /* ... */

    ~~// Set output~~
    ~~output.uv = TRANSFORM_TEX(input.uv, _MainTex);~~

    // Set output
    output.uv = TRANSFORM_TEX(input.uv, _BaseMap);

    /* ... */
}

/* ... */

half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    ~~// Main tex~~
    ~~half4 mainTex = tex2D(_MainTex, uv);~~

    // Main tex
    half4 baseMap = tex2D(_BaseMap, uv);

    ~~half4 mainColor = mainTex * _Color;~~

    half4 mainColor = baseMap * _BaseColor;

    /* ... */
}
```

Change Texture Sampling Functions to Macros

SRP provides macros for texture sampling that we can use instead of directly calling the tex2D function.

- Generally, you can use the **SAMPLE_TEXTURE2D** macro to sample textures.

We will change the texture sampling to use the SAMPLE_TEXTURE2D macro.

- In LightingToonPass.hlsl, change the _BaseMap and _NormalTex sampling from tex2D to **SAMPLE_TEXTURE2D**.

LightingToonPass.hlsl

```
/* Change texture sampling function from tex2D to SAMPLE_TEXTURE2D */

// Fragment function
```

```
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    ~~// Main tex~~
    ~~half4 baseMap = tex2D(_BaseMap, uv);~~

    // Main tex
    half4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, uv);

    ~~// Unpack normals from normal map~~
    ~~half3 normalTS = UnpackNormal(tex2D(_NormalTex, uv));~~

    // Unpack normals from normal map
    half3 normalTS = UnpackNormal(SAMPLE_TEXTURE2D(_NormalTex, sampler_NormalTex, uv));

    /* ... */
}
```

SRP also provides macros for texture and sampler parameters in function definitions, and for passing texture and sampler variables to these functions.

- The **TEXTURE2D_PARAM** macro is used for TEXTURE2D and SAMPLER parameters in function definitions.
- The **TEXTURE2D_ARGS** macro is used to pass TEXTURE2D and SAMPLER variables to functions.

We will change our functions that take sampler2D parameters to use the macros instead.

- In LightingToon.hlsl, change the functions that take sampler2D parameters to use the **TEXTURE2D_PARAM** macro.
- Additionally, change the code that calls these functions to use the **TEXTURE2D_ARGS** macro to pass the variables.

LightingToon.hlsl

```
/* Change function texture parameter to TEXTURE2D_PARAM */

/* GetLightingToonColor function */

~~// Get lighting toon color~~
~~half3 GetLightingToonColor(half3 color, UnityLight light, float distanceAttenuation, float shadowAttenuation, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, sampler2D toonRampTex, float shadowRampBlend, half3 ambient)~~

// Get lighting toon color
half3 GetLightingToonColor(half3 color, UnityLight light, float distanceAttenuation, float shadowAttenuation, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), ~~sampler2D toonRampTex,~~ float shadowRampBlend, half3 ambient)

/* GetLightingToonDiffuse function */

~~// Diffuse~~
~~half3 GetLightingToonDiffuse(UnityLight light, half3 normalWS, sampler2D toonRampTex, float shadowRampBlend)~~

// Diffuse
half3 GetLightingToonDiffuse(UnityLight light, half3 normalWS, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), ~~sampler2D toonRampTex,~~ float shadowRampBlend)

/* GetLightingToonSpecular function */

~~// Specular~~
~~half3 GetLightingToonSpecular(UnityLight light, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, sampler2D toonRampTex)~~

// Specular
half3 GetLightingToonSpecular(UnityLight light, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler) ~~sampler2D toonRampTex~~)
```

LightingToon.hlsl

```
/* Change function texture parameter variable passing to TEXTURE2D_ARGS */

// Get lighting toon color
half3 GetLightingToonColor( /* ... */ )
{
    ~~// Diffuse~~
    ~~half3 diffuse = GetLightingToonDiffuse(light, distanceAttenuation, shadowAttenuation, normalWS, toonRampTex, shadowRampBlend);~~
    ~~// Specular~~
    ~~half3 specular = GetLightingToonSpecular(light, normalWS, viewDirectionWS, specularColor, smoothness, toonRampTex);~~

    // Diffuse
    half3 diffuse = GetLightingToonDiffuse(light, distanceAttenuation, shadowAttenuation, normalWS, TEXTURE2D_ARGS(toonRampTex,
toonRampTexSampler), ~~toonRampTex,~~ shadowRampBlend);
    // Specular
    half3 specular = GetLightingToonSpecular(light, normalWS, viewDirectionWS, specularColor, smoothness, TEXTURE2D_ARGS(toonRampTex,
toonRampTexSampler) ~~toonRampTex~~);

    /* ... */
}
```

Finally, we will change the _ToonRampTex texture sampling in LightingToon.hlsl from tex2D to SAMPLE_TEXTURE2D.

- In LightingToon.hlsl, change the _ToonRampTex texture sampling from tex2D to **SAMPLE_TEXTURE2D**.

LightingToon.hlsl

```
/* Change texture sampling function from tex2D to SAMPLE_TEXTURE2D */

// Diffuse
half3 GetLightingToonDiffuse(UnityLight light, half3 normalWS, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), float shadowRampBlend)
{
    /* ... */

    ~~half3 toonRamp = tex2D(toonRampTex, toonRampUV).rgb;~~

    half3 toonRamp = SAMPLE_TEXTURE2D(toonRampTex, toonRampTexSampler, toonRampUV).rgb;

    /* ... */
}

// Specular
half3 GetLightingToonSpecular(UnityLight light, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness,
TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler))
{
    /* ... */

    ~~half3 toonRamp = tex2D(toonRampTex, toonRampUV).rgb;~~

    half3 toonRamp = SAMPLE_TEXTURE2D(toonRampTex, toonRampTexSampler, toonRampUV).rgb;

    /* ... */
}
```

Replace Light struct

We will now move on to lighting.

The UnityLight struct used in Built-in RP is replaced with the **Light** struct in URP.

- The Light struct is defined in **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl**.

We first need to include Lighting.hlsl in order to use the Light struct.

- In LightingToonPass.hlsl, include **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl**.

LightingToonPass.hlsl

```
/* Include Lighting.hlsl at top of file */

// URP includes
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

Next, we will change usage of UnityLight to be Light instead.

- In LightingToonPass.hlsl, change the UnityLight variable to be **Light**.

LightingToonPass.hlsl

```
/* Change UnityLight variable to Light */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    ~~// Construct light data~~
    ~~UnityLight light;~~
    ~~light.color = _LightColor0;~~
~~#if defined(POINT) || defined(SPOT)~~
    ~~light.dir = normalize(_WorldSpaceLightPos0.xyz - input.positionWS);  // Direction to light source~~
~~#else~~
    ~~light.dir = _WorldSpaceLightPos0.xyz;  // Pos is direction for directional lights~~
~~#endif~~
    ~~float distanceAttenuation = DistanceAttenuation(input.positionWS);  // Distance attenuation~~

    Light light;
    light.color = _LightColor0;
#if defined(POINT) || defined(SPOT)
    light.direction = normalize(_WorldSpaceLightPos0.xyz - input.positionWS);  // Direction to light source
#else
    light.direction = _WorldSpaceLightPos0.xyz;  // Pos is direction for directional lights
#endif
    light.distanceAttenuation = DistanceAttenuation(input.positionWS);  // Distance attenuation

    /* ... */
}
```

We also must change any functions which take UnityLight parameters to take Light instead.

- In LightingToon.hlsl, change any UnityLight function parameters to **Light**.
  - Also, remove the distanceAttenuation and/or shadowAttenuation parameters. These variables are included in the Light struct (unlike UnityLight).

LightingToon.hlsl

```
/* Change UnityLight parameter to Light, and remove distanceAttenuation/shadowAttenuation parameters */

/* GetLightingToonColor function */

~~// Get lighting toon color~~
~~half3 GetLightingToonColor(half3 color, UnityLight light, float distanceAttenuation, float shadowAttenuation, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), float shadowRampBlend, half3 ambient)~~

// Get lighting toon color
half3 GetLightingToonColor(half3 color, Light light, ~~float distanceAttenuation, float shadowAttenuation,~~ half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), float shadowRampBlend, half3 ambient)

/* GetLightingToonDiffuse function */

~~// Diffuse~~
~~half3 GetLightingToonDiffuse(UnityLight light, float distanceAttenuation, float shadowAttenuation, half3 normalWS, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), float shadowRampBlend)~~

// Diffuse
half3 GetLightingToonDiffuse(Light light, ~~float distanceAttenuation, float shadowAttenuation,~~ half3 normalWS, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler), float shadowRampBlend)

/* GetLightingToonSpecular function */

~~// Specular~~
~~half3 GetLightingToonSpecular(UnityLight light, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler))~~

// Specular
half3 GetLightingToonSpecular(UnityLight light, half3 normalWS, half3 viewDirectionWS, half3 specularColor, float smoothness, TEXTURE2D_PARAM(toonRampTex, toonRampTexSampler))
```

Replace SH Function for Ambient Light

URP uses a differently named function for SH for getting ambient light, so we will replace it.

- In LightingToonPass.hlsl, replace the ShadeSH9 function with URP's **SampleSH** function.
  - The function is included in **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl**.
- Additionally, remove the **SAMPLE_AMBIENT** check, which we used to only add ambient light in the ForwardBase pass, and not the ForwardAdd passes.
  - URP uses one Forward pass for all lights, so this check is no longer necessary.

LightingToonPass.hlsl

```
/* Change SH function to URP's SampleSH */
```

```
// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    ~~// Ambient light (only ForwardBase pass)~~
~~#ifdef SAMPLE_AMBIENT~~
        ~~half3 ambient = max(0, ShadeSH9(half4(normalWS, 1)));~~
~~#else~~
        ~~half3 ambient = 0;~~
~~#endif~~

    // Ambient light
    half3 ambient = SampleSH(normalWS);

    /* ... */
}
```

Looping Through Lights

From here, we will handle lighting.

As mentioned, Built-in RP uses the ForwardBase pass for the main light, and ForwardAdd passes for each additional light. URP uses one Forward pass for all lights.

To handle all lights in one Forward pass, URP loops through additional lights after handling the main light.
- The main light is retrieved with the **GetMainLight** function.
- Additional lights are retrieved by index with the **GetAdditionalLight(index)** function.
- These functions will handle distance attenuation and shadow attenuation for us, where we were doing this manually in Built-in RP.
- These functions are included in **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl**.

Main Light Handling

First, we will handle the main light.

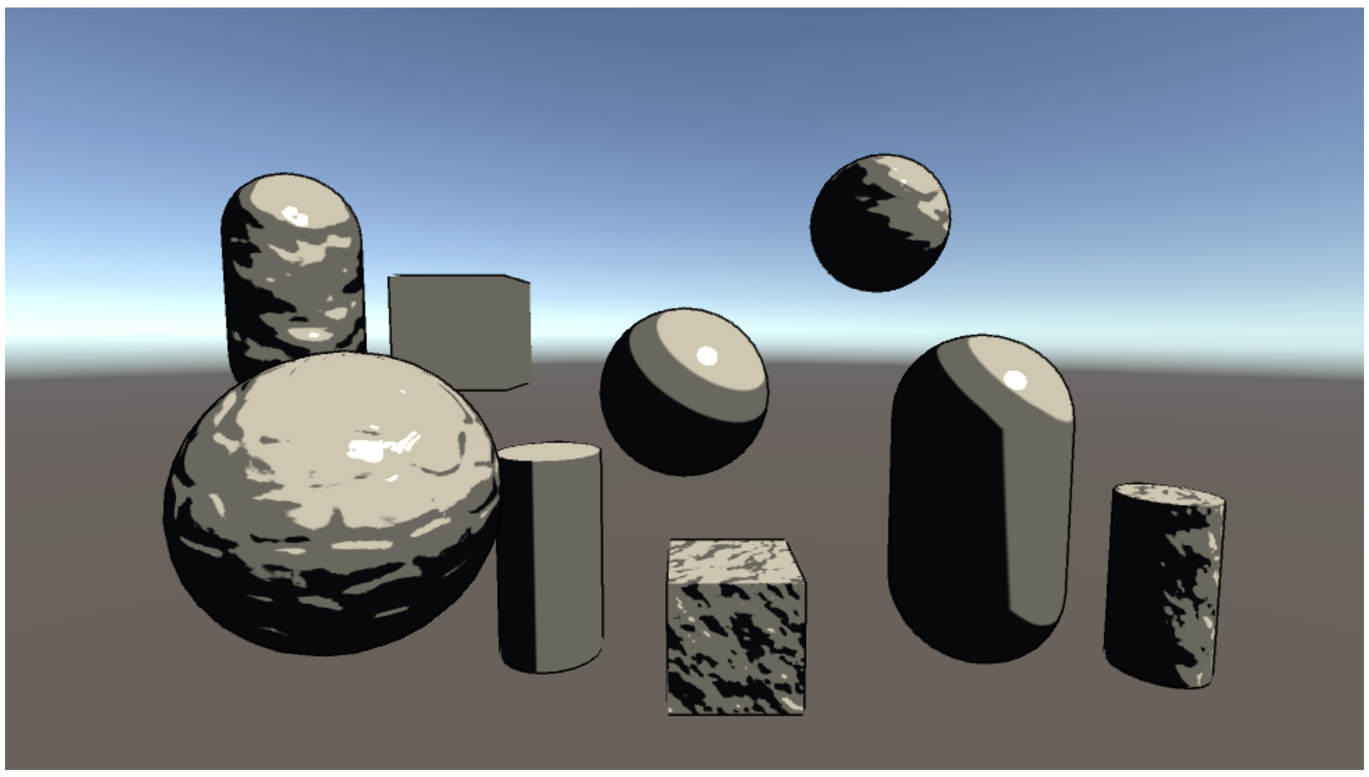
The main light can be retrieved with the GetMainLight function. A Light struct is returned, and it will contain the light color, light direction, distance attenuation, and shadow attenuation.
- In LightingToonPass.hlsl, remove the Light struct setup, and replace it with **GetMainLight**.

LightingToonPass.hlsl
```
/* Use URP's GetMainLight function to retrieve main light */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    ~~Light light;~~
    ~~light.color = _LightColor0;~~
    ~~#if defined(POINT) || defined(SPOT)~~
    ~~light.direction = normalize(_WorldSpaceLightPos0.xyz - input.positionWS);  // Direction to light source~~
    ~~#else~~
    ~~light.direction = _WorldSpaceLightPos0.xyz;  // Pos is direction for directional lights~~
    ~~#endif~~
    ~~light.distanceAttenuation = DistanceAttenuation(input.positionWS);  // Distance attenuation~~

    // Main light
    Light mainLight = GetMainLight();

    /* ... */
}
```

- Since distance attenuation is handled automatically, we can remove the **DistanceAttenuation** function used previously.

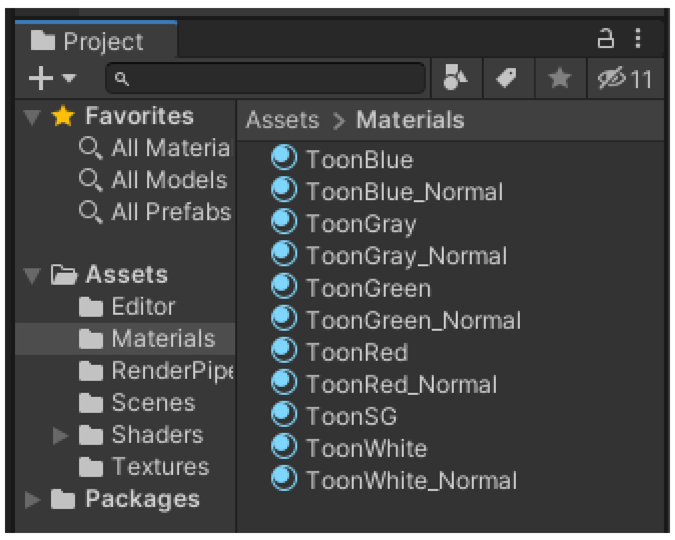
LightingToon.hlsl
```
/* Remove DistanceAttenuation function as we are no longer using it */

~~// Distance attenuation implementation taken from AutoLight.cginc~~
~~float DistanceAttenuation(float3 positionWS)~~
~~{~~
    ~~float distanceAttenuation;~~
~~#if defined(POINT)~~
    ~~float3 lightCoord = mul(unity_WorldToLight, float4(positionWS, 1)).xyz;~~
    ~~distanceAttenuation = tex2D(_LightTexture0, dot(lightCoord, lightCoord).rr).r;~~
~~#elif defined(SPOT)~~
    ~~float4 lightCoord = mul(unity_WorldToLight, float4(positionWS, 1));~~
    ~~distanceAttenuation = (lightCoord.z > 0) * UnitySpotCookie(lightCoord) * UnitySpotAttenuate(lightCoord.xyz);~~
~~#else~~
    ~~distanceAttenuation = 1; // 1 for directional light~~
~~#endif~~
    ~~return distanceAttenuation;~~
~~}~~
```

With these changes, we can see that the main light is now working (shadows are not supported yet).



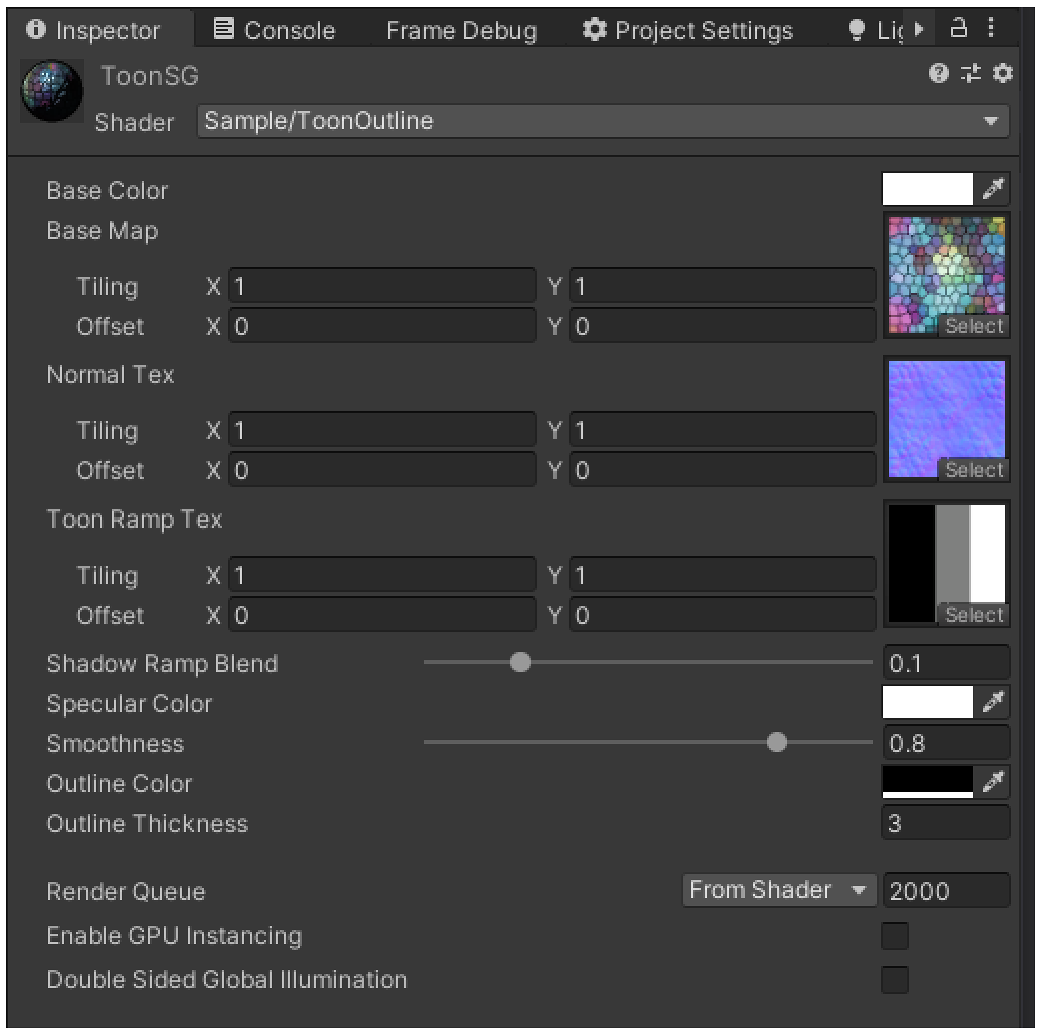
The reason the objects are all white and untextured is because we changed the _MainTex and _Color properties to _BaseMap and _BaseColor, so they have their default values. We need to go through each material and set the _BaseMap and _BaseColor properties to their original values.

- Go through each material under **Materials/** and change the **_BaseMap** and **_BaseColor** properties accordingly.



Assets > Materials

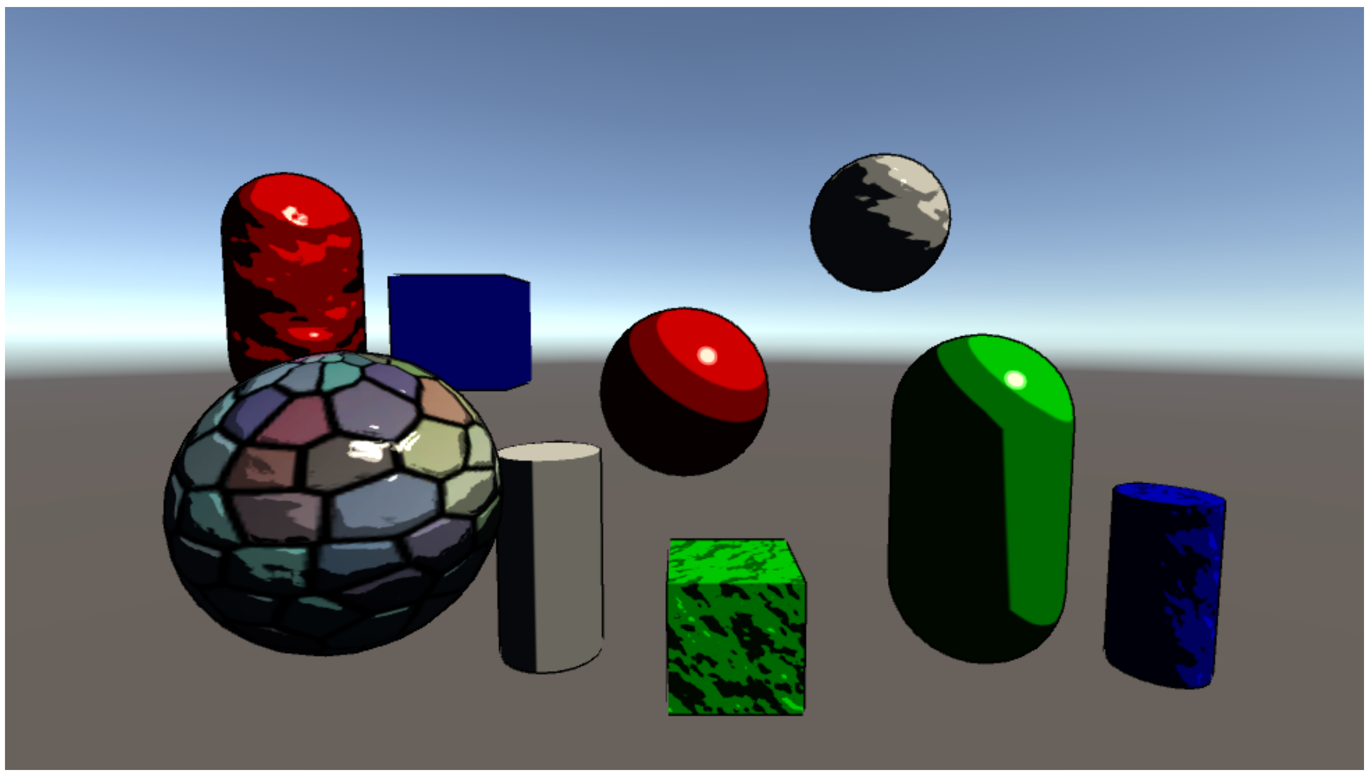
- ToonBlue
- ToonBlue_Normal
- ToonGray
- ToonGray_Normal
- ToonGreen
- ToonGreen_Normal
- ToonRed
- ToonRed_Normal
- ToonSG
- ToonWhite
- ToonWhite_Normal



ToonSG
Shader Sample/ToonOutline

Base Color				
Base Map				
Tiling	X	1	Y	1
Offset	X	0	Y	0
Normal Tex				
Tiling	X	1	Y	1
Offset	X	0	Y	0
Toon Ramp Tex				
Tiling	X	1	Y	1
Offset	X	0	Y	0
Shadow Ramp Blend		0.1		
Specular Color				
Smoothness		0.8		
Outline Color				
Outline Thickness		3		
Render Queue	From Shader	2000		
Enable GPU Instancing				
Double Sided Global Illumination				

The textures and colors will now have their original values.



Additional Light Handling

Next, we will handle additional lights.

We first need to add keywords to enable additional lights.

- In ToonOutline.shader, add the **_ADDITIONAL_LIGHTS** keyword to the Forward pass.
    - If you are using vertex lighting, you will also need the **_ADDITIONAL_LIGHTS_VERTEX** keyword.

ToonOutline.shader

```
/* Add keywords for additional lights to Forward pass */

// Enable additional lights
// Also add _ADDITIONAL_LIGHTS_VERTEX if doing vertex lighting with additional lights
#pragma multi_compile _ _ADDITIONAL_LIGHTS
```

As mentioned earlier, in order to handle additional lights, we will loop through each light and add them to the result color. This consists of the following steps:

1. Get the number of additional lights with the **GetAdditionalLightsCount** function.
2. Loop through each additional light, retrieving the light data by index with the **GetAdditionalLight** function.
3. Calculate the lighting color in the same way as the main light (here, using the GetLightingToonColor function).
4. Add the lighting color to the result color.

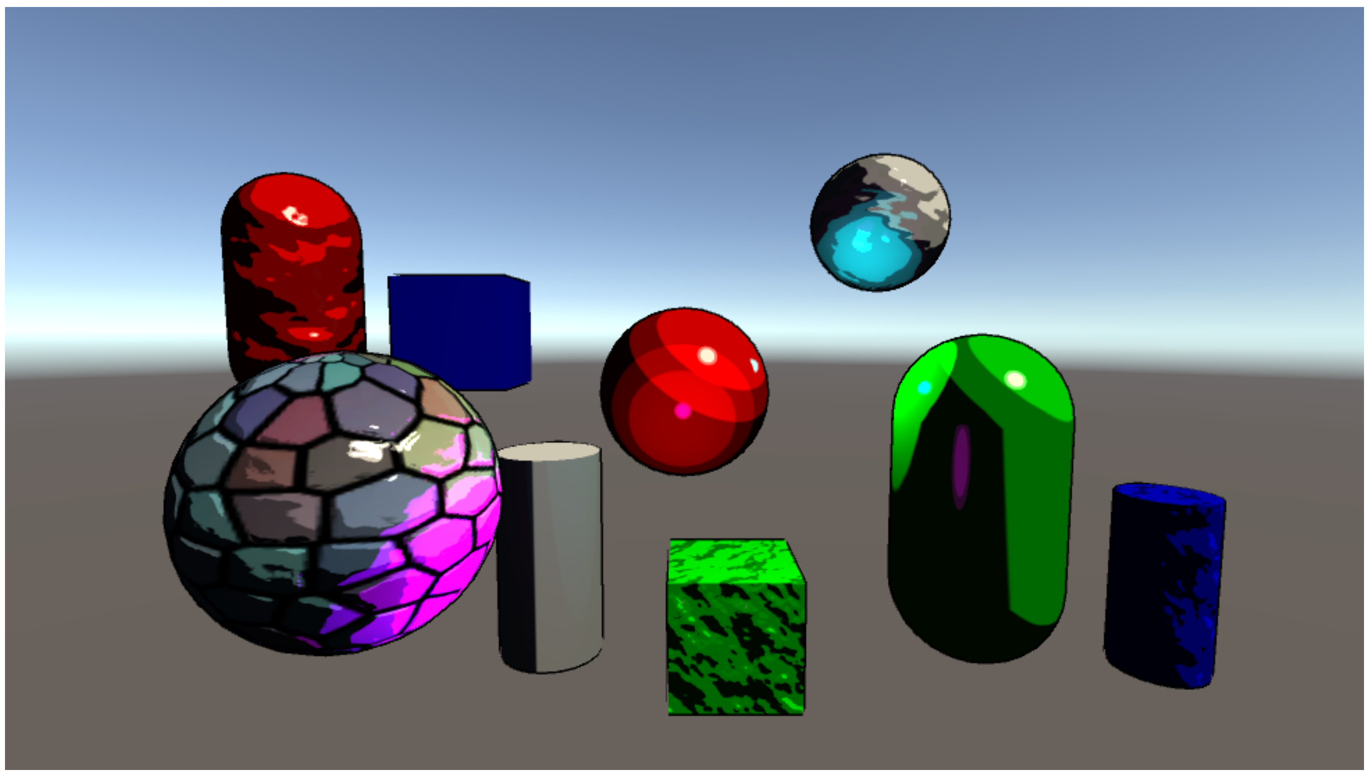
Add the additional light handling to the Forward pass.

- In LightingToonPass.hlsl, add the code to handle additional lights, if _ADDITIONAL_LIGHTS is defined.

LightingToonPass.hlsl

```
/* Add additional light handling after main light in fragment function, if _ADDITIONAL_LIGHTS is defined */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    // Additional lights
#ifdef _ADDITIONAL_LIGHTS
    // Get additional light count
    int additionalLightCount = GetAdditionalLightsCount();

    // Loop through additional lights
    for (int lightIndex = 0; lightIndex < additionalLightCount; lightIndex++) {
        // Get additional light data by index
        Light additionalLight = GetAdditionalLight(lightIndex, input.positionWS);

        // Add lighting color
        color.rgb += GetLightingToonColor(
            mainColor.rgb,
            additionalLight,
            normalWS,
            input.viewDirectionWS,
            _SpecularColor,
            _Smoothness,
            TEXTURE2D_ARGS(_ToonRampTex, sampler_ToonRampTex),
            _ShadowRampBlend,
            0 // Only add ambient lighting during main light
        );
    }
#endif

    /* ... */
}
```

With these changes, additional lights (the point lights) will now be working.



Receiving Shadows (Main Light)

We will now move on to receiving and casting shadows. We will start with receiving shadows for the main light.

First, we need to add keywords to enable shadows for the main light.

- In ToonOutline.shader, add the following keywords:
  - **_MAIN_LIGHT_SHADOWS**
  - **_MAIN_LIGHT_SHADOWS_CASCADE** (if using shadow cascades)
  - **_SOFT_SHADOWS** (if using soft shadows)

ToonOutline.shader

```
/* Add shadow keywords to Forward pass */

// Enable shadows on main light
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS
// Shadow cascades
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
// Enable additional lights
// Also add _ADDITIONAL_LIGHTS_VERTEX if doing vertex lighting with additional lights
#pragma multi_compile _ _ADDITIONAL_LIGHTS
// Soft shadows
#pragma multi_compile _ _SHADOWS_SOFT
```

Next, we will get the shadow coordinates.

- You can retrieve the shadow coordinates with URP's **TransformWorldToShadowCoord** function and passing the world-space position.
  - The function is included in **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Shadows.hlsl**.
- You can retrieve the shadow coordinates in the vertex function or fragment function, depending on whether you want the shadow coordinates per-vertex or per-pixel (e.g. when using shadow cascades).
  - URP provides the **REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR** keyword to determine which to use. When REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR is defined, shadow coordinates should be retrieved in the vertex function.

Let's retrieve the shadow coordinates in either the vertex function or fragment function depending on whether we want the shadow coordinates per-vertex or per-pixel.

- In LightingPassToon.hlsl, retrieve the shadow coordinates with the **TransformWorldToShadowCoord** function.
- If **REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR** is defined, get the shadow coordinates in the vertex function and pass it to the fragment function.
- If **REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR** is not defined, get the shadow coordinates in the fragment function.

LightingPassToon.hlsl

```
/* Retrieve shadow coordinates */
```

```
// Vert output/Frag input
struct VaryingsToon
{
    float2 uv               : TEXCOORD0;
    float3 positionWS       : TEXCOORD1;   // World-space position
    half3 normalWS          : TEXCOORD2;   // World-space normal
    half3 viewDirectionWS   : TEXCOORD3;   // World-space view direction
    half3 tangentWS         : TEXCOORD4;   // World-space tangent
    half3 bitangentWS       : TEXCOORD5;   // World-space bitangent
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    float4 shadowCoord      : TEXCOORD6;   // Vertex shadow coords if required
#endif
    float4 pos              : SV_POSITION; // Clip-space position
};

/* Get shadowCoords in vertex function */

// Vertex function
VaryingsToon vertToon(AttributesToon input)
{
    /* ... */

    // Set output
    output.uv = TRANSFORM_TEX(input.uv, _BaseMap);
    output.pos = positionCS;
    output.normalWS = normalWS;
    output.tangentWS = tangentWS;
    output.bitangentWS = bitangentWS;
    output.positionWS = positionWS;
    output.viewDirectionWS = normalize(GetCameraPositionWS() - positionWS);
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    // Vertex shadow coords if required
    output.shadowCoord = TransformWorldToShadowCoord(positionWS);
#endif

    return output;
}

/* ... */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    // Main light
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    // Use vertex shadow coords if required
    float4 shadowCoord = input.shadowCoord;
#else
    // Otherwise, get per-pixel shadow coords
    float4 shadowCoord = TransformWorldToShadowCoord(input.positionWS);
#endif
    Light mainLight = GetMainLight();

    /* ... */
}
```

Next, we will get the shadow attenuation. The **GetMainLight** function will calculate the shadow attenuation and add it to the light data if shadow coordinates are passed.

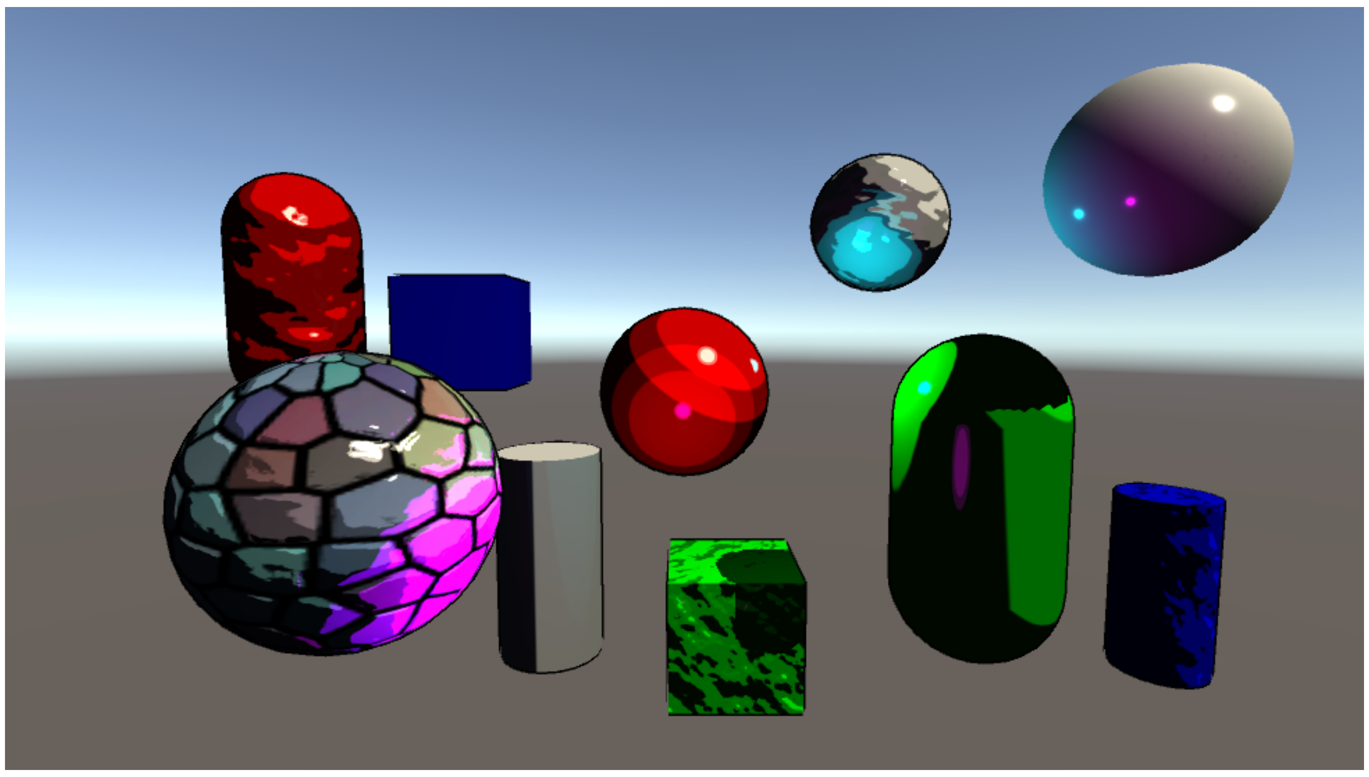
- In LightingToonPass.hlsl, pass the shadow coordinates to the **GetMainLight** function when retrieving the main light data.
- Only do this if **_MAIN_LIGHT_SHADOWS** is defined. Otherwise (i.e. shadows are not enabled), don't pass anything to the GetMainLight function.

LightingToonPass.hlsl

```
/* Get shadow attenuation when retrieving main light */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */
    // Main light
#ifdef _MAIN_LIGHT_SHADOWS
    // Receiving shadows
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    // Use vertex shadow coords if required
    float4 shadowCoord = input.shadowCoord;
#else
    // Otherwise, get per-pixel shadow coords
    float4 shadowCoord = TransformWorldToShadowCoord(input.positionWS);
#endif
    Light mainLight = GetMainLight(shadowCoord);
#else
    // No shadows
    Light mainLight = GetMainLight();
#endif

    /* ... */
}
```

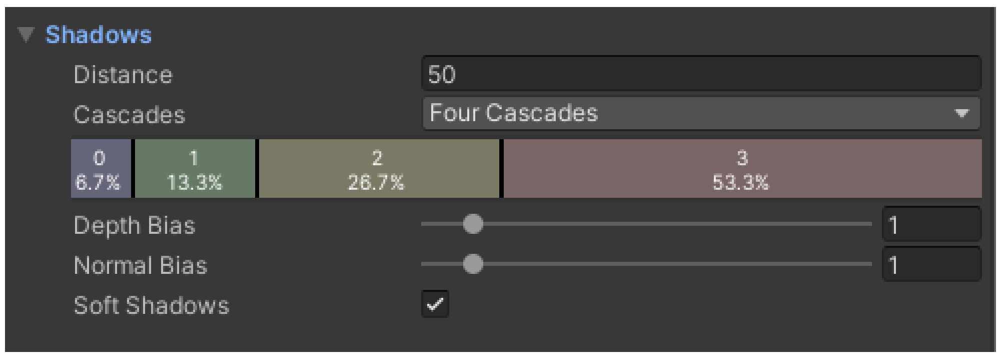
Since we are calculating shadow attenuation, main light shadows are now visible.

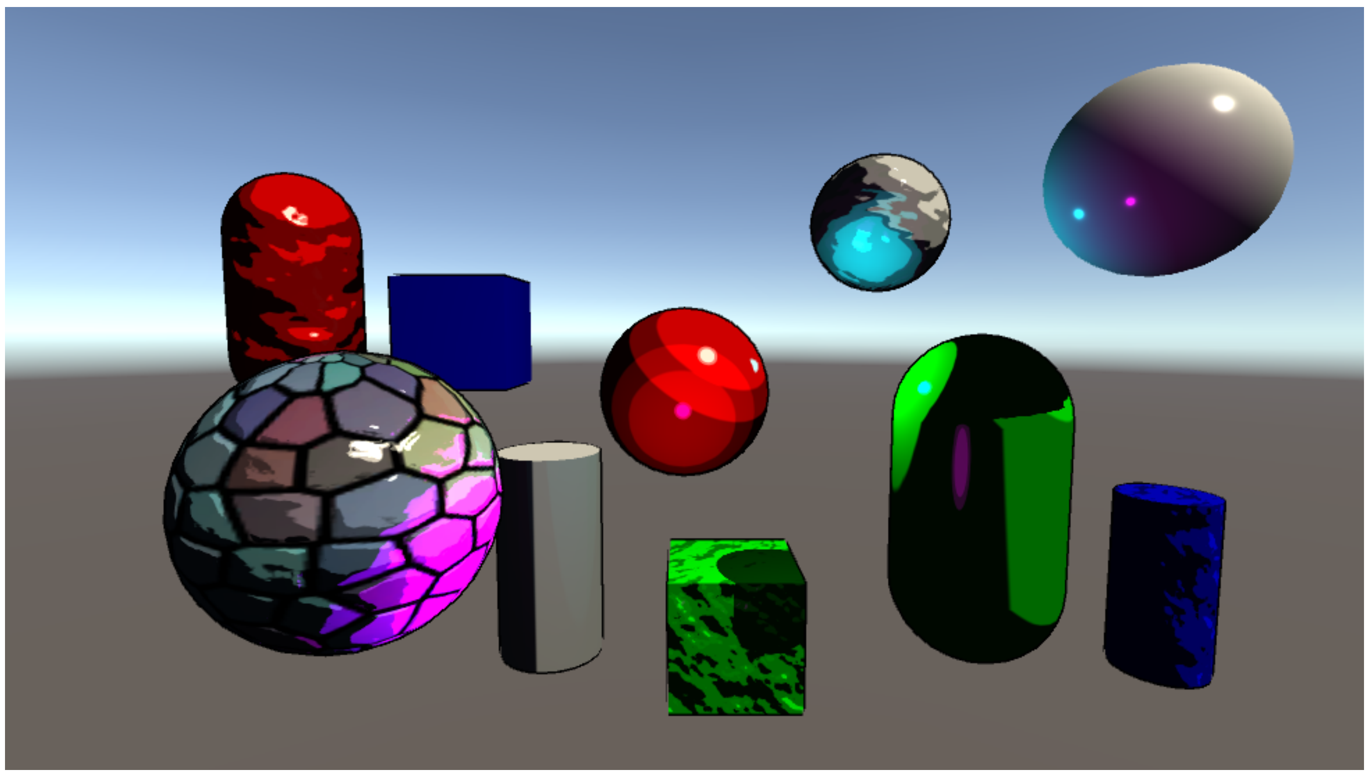
- Note that since our shader still does not support CASTING shadows yet, you will need to test shadows by placing an object in the scene that uses the default URP Lit shader.



If the shadows appear jagged, we can enable main light shadow cascades and soft shadows in the UniversalRenderPipelineAsset (since we support them in the shader).

- Select **RenderPipline/UniversalRenderPipelineAsset** in the inspector.
- Under **Shadows**, set **Cascades** to Two Cascades or Four Cascades, and enable **Soft Shadows**.


Shadows
Distance 50
Cascades Four Cascades
0 6.7% | 1 13.3% | 2 26.7% | 3 53.3%
Depth Bias 1
Normal Bias 1
Soft Shadows ✓



Receiving Shadows (Additional Lights)

Next, we will enable shadows on additional lights.

We will add keywords to enable shadows for additional lights.

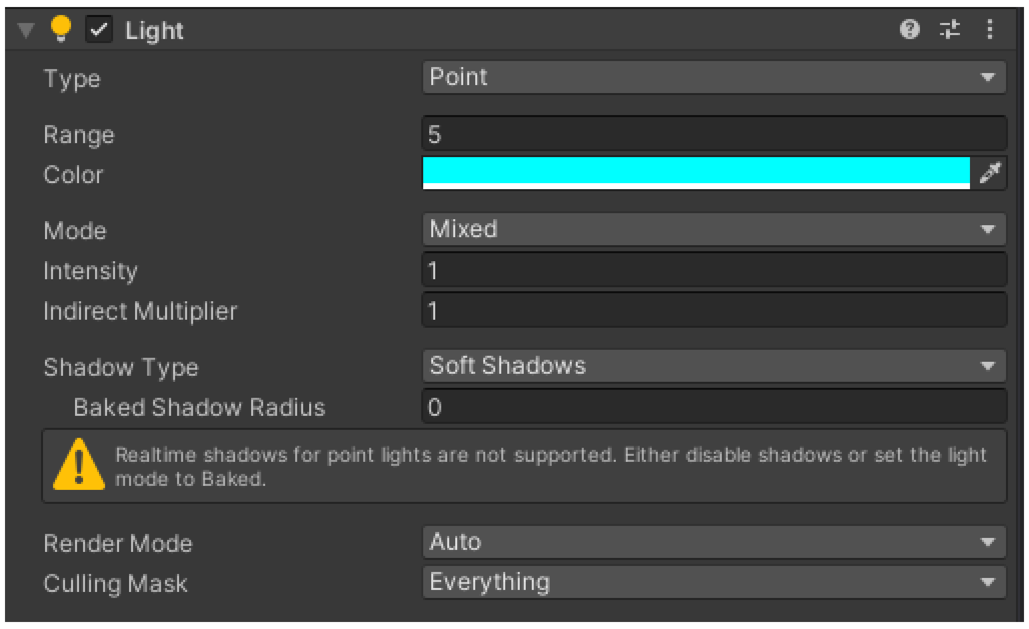
- In ToonOutline.shader, add the **_ADDITIONAL_LIGHT_SHADOWS** to the Forward pass.

ToonOutline.shader

```
/* Add keyword for additional light shadows to Forward pass */

// Enable shadows on main light
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS
// Shadow cascades
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
// Enable additional lights
// Also add _ADDITIONAL_LIGHTS_VERTEX if doing vertex lighting with additional lights
#pragma multi_compile _ _ADDITIONAL_LIGHTS
// Enable shadows on additional lights
#pragma multi_compile _ _ADDITIONAL_LIGHT_SHADOWS
// Soft shadows
#pragma multi_compile _ _SHADOWS_SOFT
```

This is all that is required. The GetAdditionalLight function will compute shadows if _ADDITIONAL_LIGHT_SHADOWS is defined.

In order to test shadows for additional lights, we cannot use point lights, as URP does not support realtime point light shadows.



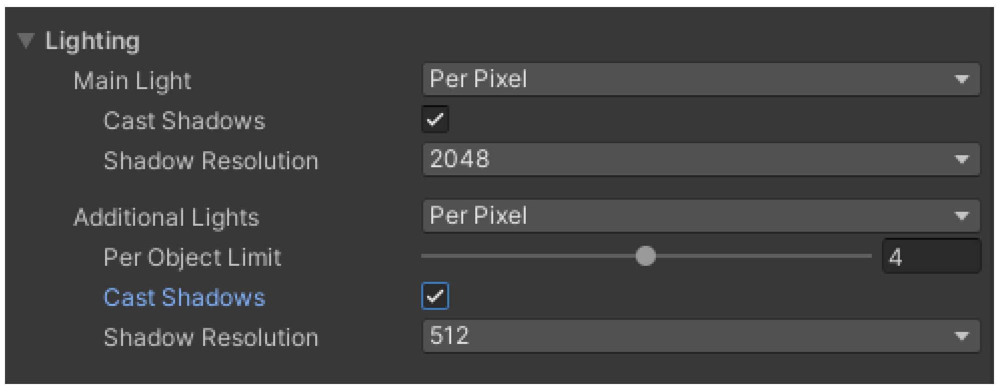
Light

Type	Point
Range	5
Color	🟦
Mode	Mixed
Intensity	1
Indirect Multiplier	1
Shadow Type	Soft Shadows
Baked Shadow Radius	0
Render Mode	Auto
Culling Mask	Everything

Realtime shadows for point lights are not supported. Either disable shadows or set the light mode to Baked.

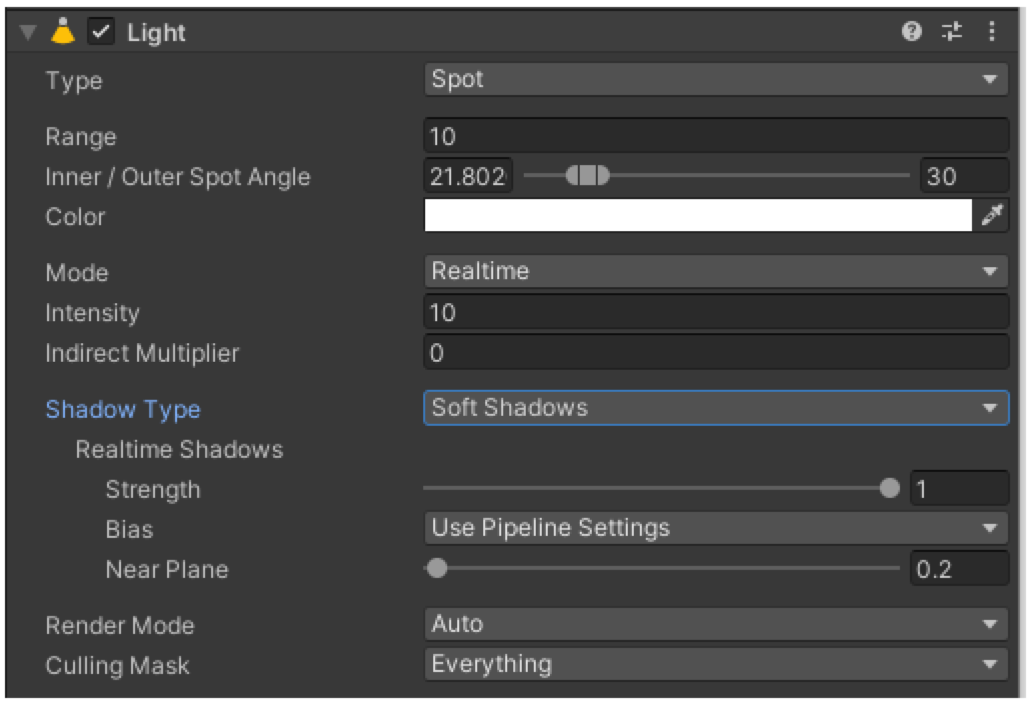
So, we will use spot lights instead. Place a spot light in the scene, and position an object using the default URP lit shader to cast a shadow. Make sure additional light shadows are enabled in the UniversalRenderPipelineAsset, and that the spot light has shadows enabled.

- Select **RenderPipline/UniversalRenderPipelineAsset** in the inspector.
- Under **Lighting -> Additional Lights**, enable **Cast Shadows**.



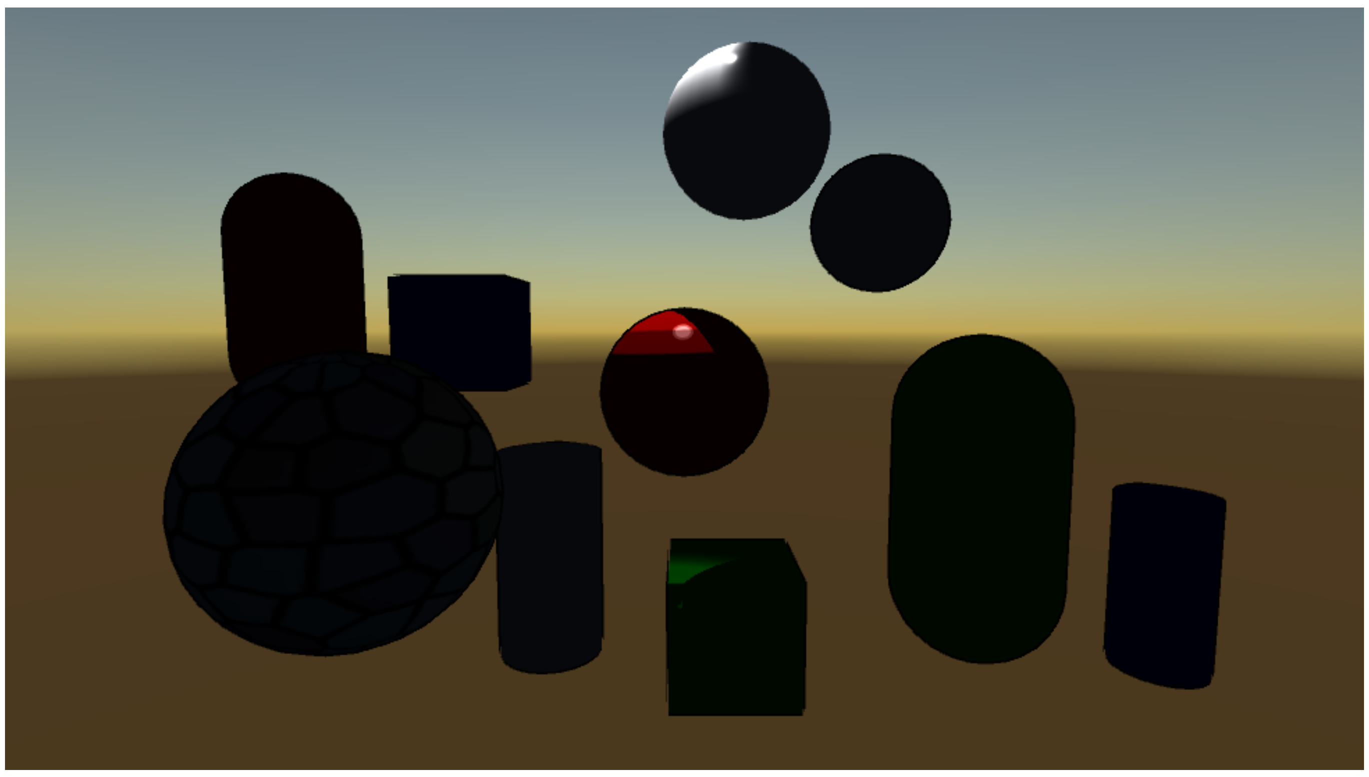
Lighting
Main Light: Per Pixel
Cast Shadows: ✓
Shadow Resolution: 2048
Additional Lights: Per Pixel
Per Object Limit: 4
Cast Shadows: ✓
Shadow Resolution: 512

- In the scene, select the spot light in the inspector.
- Set **Shadow Type** to Hard Shadows or Soft Shadows.



Light
Type: Spot
Range: 10
Inner / Outer Spot Angle: 21.802 / 30
Color
Mode: Realtime
Intensity: 10
Indirect Multiplier: 0
Shadow Type: Soft Shadows
Realtime Shadows
Strength: 1
Bias: Use Pipeline Settings
Near Plane: 0.2
Render Mode: Auto
Culling Mask: Everything

Shadows for additional lights (for spot lights) will now be visible.



Casting Shadows

Finally, we will add shadow casting support.

We will be using URP's **ShadowCasterPass** to do the shadow casting for us.

- See URP's **Lit** shader and **Packages/com.unity.render-pipelines.universal/Shaders/ShadowCasterPass.hlsl** for more information.

Add a shadow caster pass to the shader.

- In ToonOutline.shader, add a new shadow caster pass with the **ShadowCaster** light mode.
- Using URP's Lit shader as a reference, set it up to use URP's ShadowCasterPass.

ToonOutline.shader

```
/* Add new ShadowCaster pass */

// Shadow caster pass
Pass
```

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

    ZWrite On
    ZTest LEqual

    HLSLPROGRAM

    // Required to compile gles 2.0 with standard SRP library
    #pragma prefer_hlslcc gles
    #pragma target 2.0

    // GPU Instancing
    #pragma multi_compile_instancing

    // Vertex/fragment functions used by ShadowCasterPass.hlsl
    #pragma vertex ShadowPassVertex
    #pragma fragment ShadowPassFragment

    // URP includes
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"
    // Required by URP ShadowCasterPass.hlsl
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/SurfaceInput.hlsl"

    // URP Shadow caster pass
    #include "Packages/com.unity.render-pipelines.universal/Shaders/ShadowCasterPass.hlsl"

    ENDHLSL
}
```

- ShadowCasterPass requires specific properties, so add **_BaseColor**, **_BaseMap_ST**, and **_Cutoff** property declarations.

ToonOutline.shader

```
/* Add properties required by ShadowCasterPass.hlsl */

// Shadow caster pass
Pass
{
    /* ... */

    HLSLPROGRAM

    /* ... */

    // Properties required by ShadowCasterPass.hlsl
    half4 _BaseColor;
    float4 _BaseMap_ST;
    half _Cutoff;

    // URP Shadow caster pass
    #include "Packages/com.unity.render-pipelines.universal/Shaders/ShadowCasterPass.hlsl"

    ENDHLSL
}
```

However, in order to maintain SRP batcher compatibility, we must declare these properties in a CBUFFER block. Additionally, the CBUFFER block must be the same across all passes, so make sure to add the properties from the other passes as well.

- Declare the ShadowCaster pass properties inside a CBUFFER block, and add the properties from the Forward and Outline passes.

ToonOutline.shader

```
/* Declare ShadowCaster pass properties inside CBUFFER block */

// Shadow caster pass
Pass
{
    /* ... */

    ~~// Properties required by ShadowCasterPass.hlsl~~
    ~~half4 _BaseColor;~~
    ~~float4 _BaseMap_ST;~~
    ~~half _Cutoff;~~

    // Properties (needs to be same across all passes for SRP batcher compatibility)
    CBUFFER_START(UnityPerMaterial)
        half4 _BaseColor;
        float4 _BaseMap_ST;

        float _ShadowRampBlend;

        half3 _SpecularColor;
        float _Smoothness;

        half4 _OutlineColor;
        half _OutlineThickness;

        // Properties required by URP ShadowCasterPass.hlsl
        half _Cutoff;
    CBUFFER_END

    /* ... */
}
```

Similarly, the CBUFFER blocks for the other passes must be modified, since we added a new _Cutoff property to the ShadowCaster pass.

- In ToonOutline.shader, add the _Cutoff property to the Outline pass.
- In LightingToonPass.hlsl, add the _Cutoff property to the Forward pass.

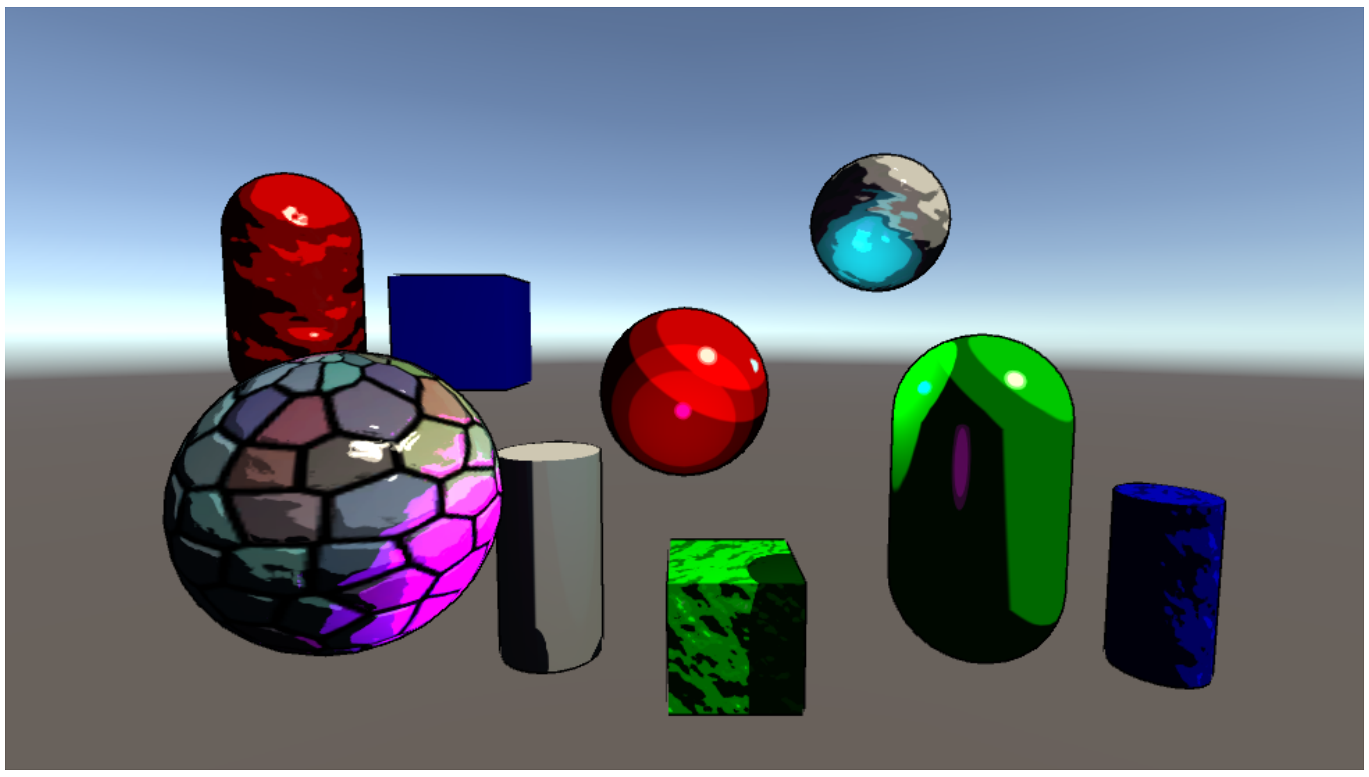
ToonOutline.shader

```
/* Add _Cutoff property to Outline pass */

// Outline pass
Pass
{
    /* ... */

    // Properties (needs to be same across all passes for SRP batcher compatibility)
    CBUFFER_START(UnityPerMaterial)
    /* ... */

        // Properties required by URP ShadowCasterPass.hlsl
        half _Cutoff;
    CBUFFER_END

    /* ... */
}
```

LightingToonPass.hlsl

```
/* Add _Cutoff property to Forward pass */

// Properties (needs to be same across all passes for SRP batcher compatibility)
CBUFFER_START(UnityPerMaterial)
    /* ... */

    // Properties required by URP ShadowCasterPass.hlsl
    half _Cutoff;
CBUFFER_END
```

With these changes, the objects are now casting and receiving shadows, and SRP batch compatibility is maintained.

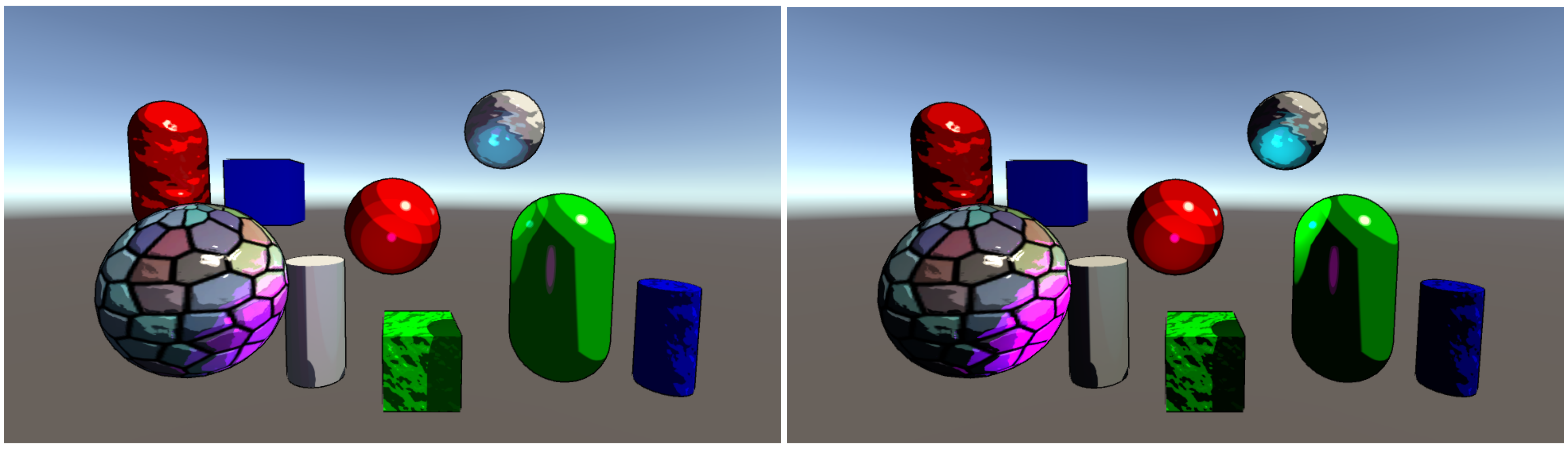


SRGB Conversion of Ambient Light

NOTE: This section is only required for URP 10.1.0 or lower.

When compared with the result from Built-in RP, the URP result may appear slightly darker.

- Below are the Built-in RP result on the left, and URP on the right.



The reason for this is SRGB conversion on ambient light. In Built-in RP, SRGB conversion is done on ambient light when the color space is set to Gamma (in the ShadeSH9 function).

URP's SampleSH function does not do the same. This is the case for URP's shaders as well (ambient light will appear darker).

- This issue was acknowledged as a bug, and was **fixed in URP 10.2.0 and above**.
    - See https://github.com/Unity-Technologies/Graphics/blob/v10.2.0/com.unity.render-pipelines.core/ShaderLibrary/EntityLighting.hlsl#L79 where SRGB conversion was added to the SampleSH function.

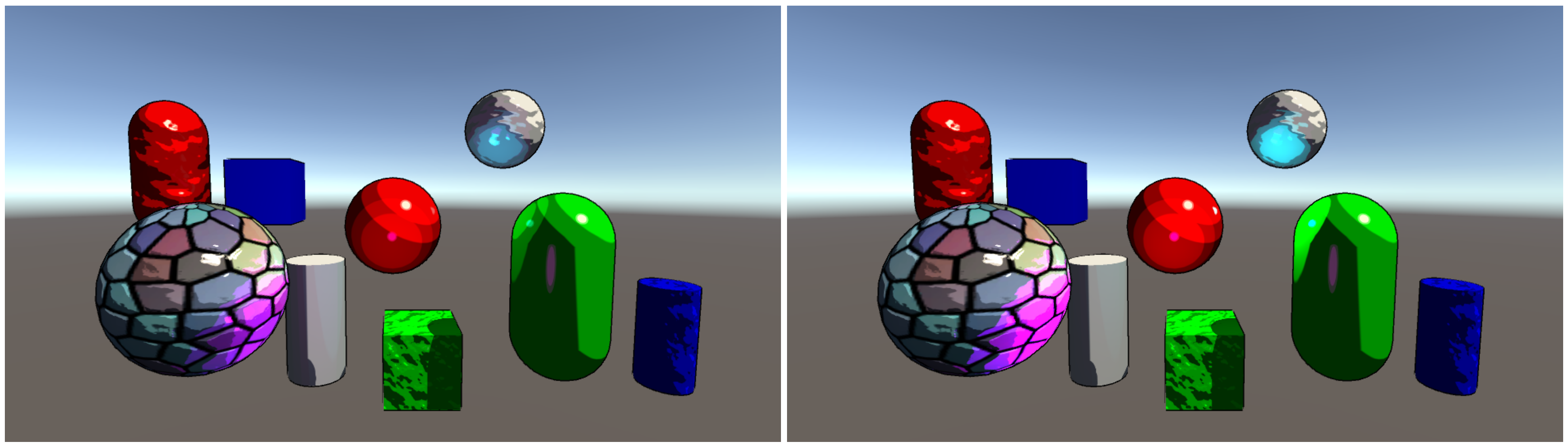
To address this, we will do SRGB conversion manually on ambient light if the project's color space is set to Gamma.

- In LightingToonPass.hlsl, convert the ambient light color to SRGB with URP's **FastLinearToSRGB** function, when **UNITY_COLORSPACE_GAMMA** is defined.
    - FastLinearToSRGB is defined in **Packages/com.unity.render-pipelines.core/ShaderLibrary/Color.hlsl**.

LightingToonPass.hlsl

```
/* Manually add SRGB conversion to ambient light if color space is Gamma */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    // Ambient light
    half3 ambient = SampleSH(normalWS);
    // Manual SRGB conversion of ambient light if color space is set to Gamma
    // (This is fixed in SRP version 10.2.0 and higher, so remove in that case)
#if UNITY_COLORSPACE_GAMMA
    ambient = FastLinearToSRGB(ambient);
#endif

    /* ... */
}
```

The result will now appear closer to Built-in RP.



Fog

Finally, we will re-enable fog, using URP's built-in fog functions.

Add a fog factor variable to the fragment function input to determine the fog thickness.

- In LightingToonPass.hlsl, add a **fogFactor** variable to VaryingsToon.

LightingToonPass.hlsl

```
/* Add fog factor to fragment input */

// Vert output/Frag input
struct VaryingsToon
{
    float2 uv                : TEXCOORD0;
    float3 positionWS        : TEXCOORD1;   // World-space position
    half3 normalWS           : TEXCOORD2;   // World-space normal
    half3 viewDirectionWS    : TEXCOORD3;   // World-space view direction
    half3 tangentWS          : TEXCOORD4;   // World-space tangent
    half3 bitangentWS        : TEXCOORD5;   // World-space bitangent

#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    float4 shadowCoord       : TEXCOORD6;   // Vertex shadow coords if required
#endif
    half fogFactor           : TEXCOORD7;   // Fog factor
    float4 pos               : SV_POSITION; // Clip-space position
};
```

In the vertex function, calculate the fog factor from the vertex position.

- In LightingToonPass.hlsl, calculate the fog factor in the vertex function with URP's **ComputeFogFactor** function.
    - See **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl** for more information.

LightingToonPass.hlsl

```
/* Compute and output fog factor in vertex function */

// Vertex function
VaryingsToon vertToon(AttributesToon input)
{
    /* ... */

    // Set output
    output.uv = TRANSFORM_TEX(input.uv, _BaseMap);
    output.pos = positionCS;
    output.normalWS = normalWS;
    output.tangentWS = tangentWS;
    output.bitangentWS = bitangentWS;
    output.positionWS = positionWS;
    output.viewDirectionWS = normalize(GetCameraPositionWS() - positionWS);
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    // Vertex shadow coords if required
    output.shadowCoord = TransformWorldToShadowCoord(positionWS);
#endif
    output.fogFactor = ComputeFogFactor(positionCS.z);
    return output;
}
```

In the fragment function, we will mix the fog with the lit color to get the final fragment color.

- In LightingToonPass.hlsl, get the final color with fog with URP’s **MixFog** function.
    - See **Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl** for more information.

LightingToonPass.hlsl

```
/* Mix color with fog in fragment function */

// Fragment function
half4 fragToon(VaryingsToon input) : SV_Target
{
    /* ... */

    // Mix fog
    color.rgb = MixFog(color.rgb, input.fogFactor);

    return color;
}
```

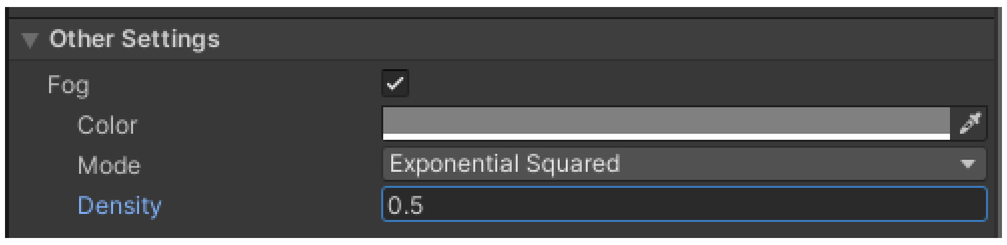
Since we also have an outline, we will apply fog to the Outline pass as well, in ToonOutline.shader.

ToonOutline.shader

```
// Outline pass
Pass
{
    /* ... */

/* Add fog factor to fragment input */

    // Vert output/Frag input
    struct Varyings
    {
        half fogFactor  : TEXCOORD0;   // Fog factor
        float4 pos      : SV_POSITION; // Clip-space position
    };

/* Compute and output fog factor in vertex function */

    // Vertex function
    Varyings vert(Attributes input)
    {
        /* ... */

        // Set output
        output.pos = positionCS;
        output.fogFactor = ComputeFogFactor(positionCS.z);
        return output;
    }

/* Mix color with fog in fragment function */

    // Fragment function
    half4 frag(Varyings input) : SV_Target
    {
        half4 color = _OutlineColor;

        // Mix fog
        color.rgb = MixFog(color.rgb, input.fogFactor);

        return color;
    }

    /* ... */

}
```

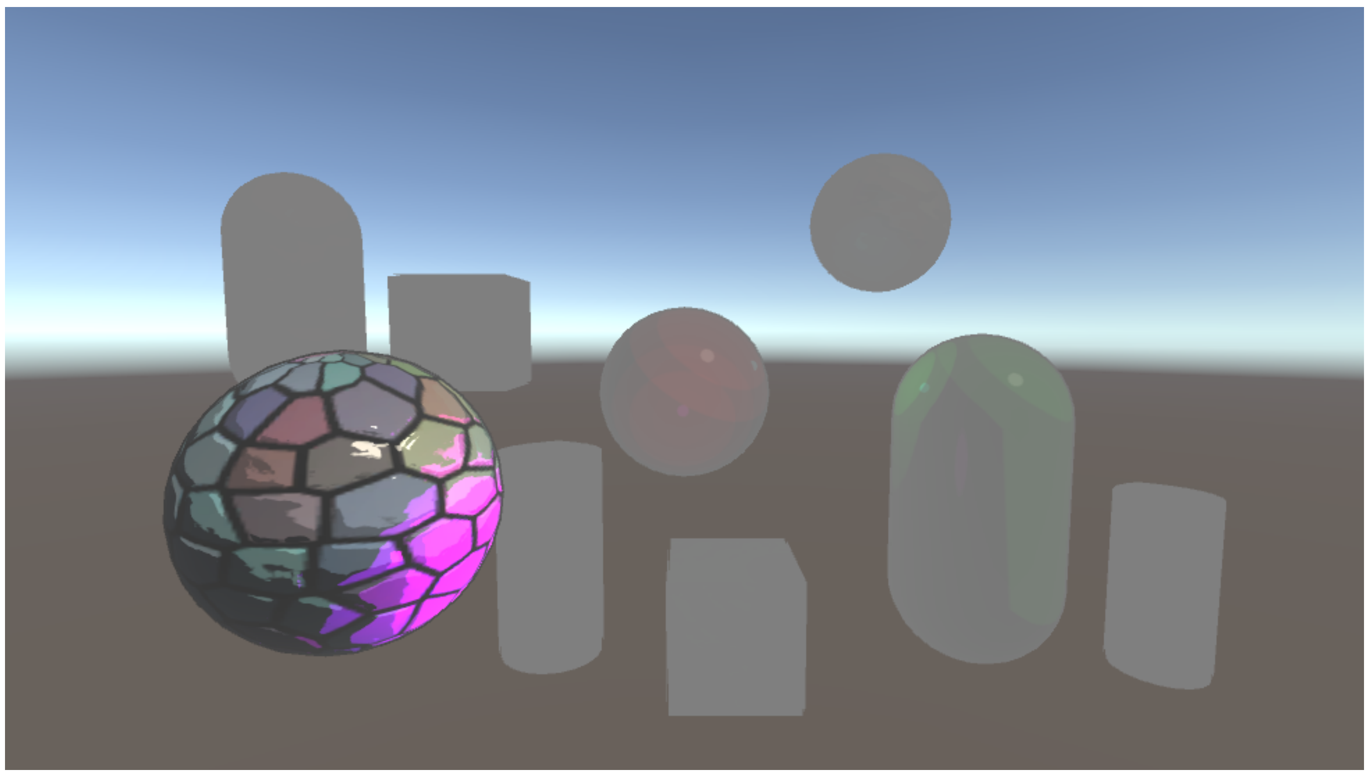
To test, enable fog in the Lighting settings.

- In the Lighting window, go to **Other Settings**.
- Enable **Fog**, and set the **Density** to around 0.5.


Other Settings

Fog	✓
Color	
Mode	Exponential Squared
Density	0.5

The fog should now be visible on the objects.



Note About Outline Pass and SRP Batcher

For one last note, we will touch upon multi-pass shaders in URP.

For this toon shader, we have a vertex-based outline implementation done in a separate Outline pass.

URP technically does not support multi-pass shaders. URP has passes differentiated with the LightMode tag, and what tags you can use are predetermined.

- See https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.5/manual/urp-shaders/urp-shaderlab-pass-tags.html for details.
- Since the Outline pass current has no LightMode tag specified, it is equivalent to SRPDefaultUnlit. We can explicitly set it to this if we want, but it is not necessary.

ToonOutline.shader

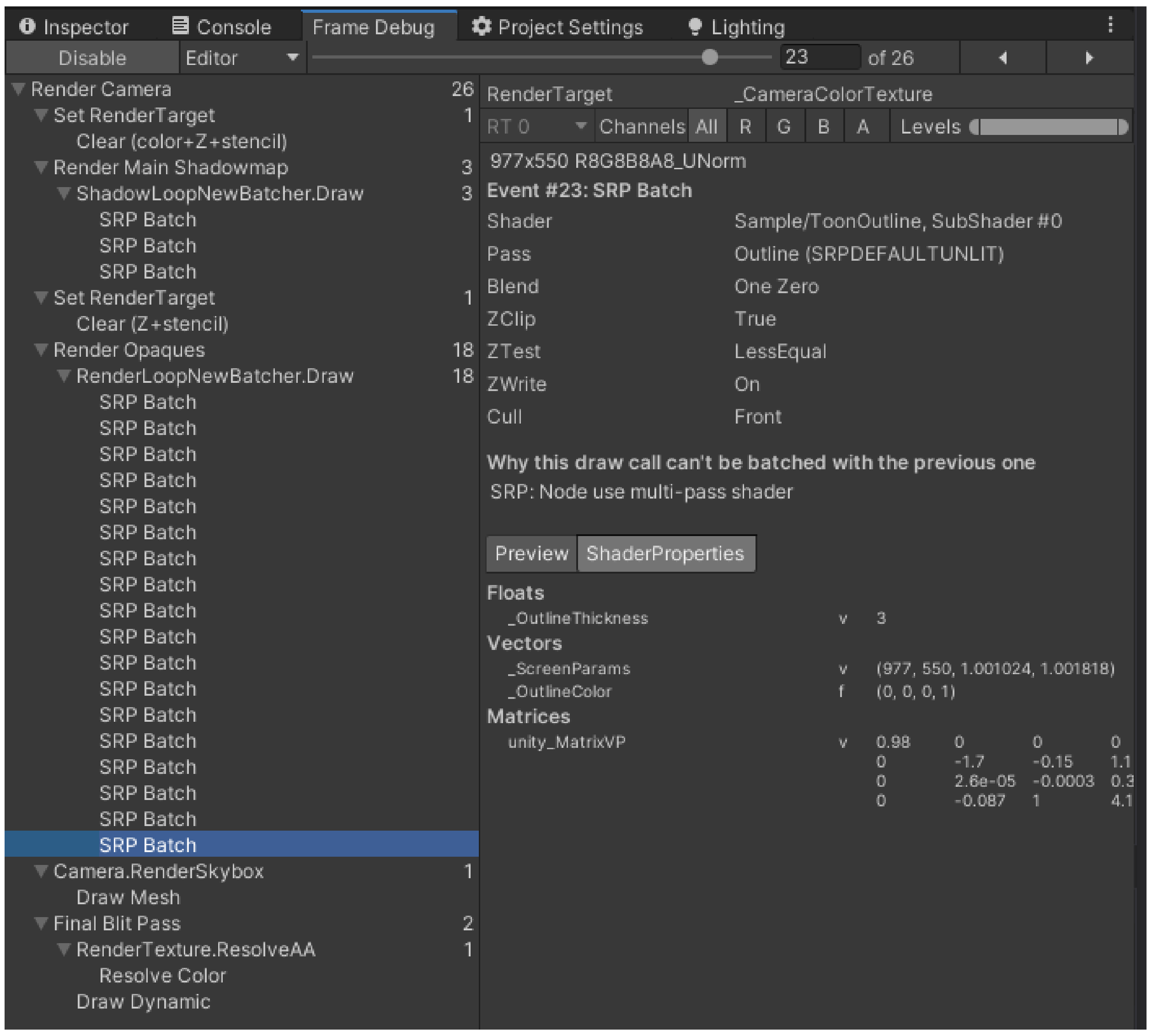
```
Name "Outline"
Tags { "LightMode" = "SRPDefaultUnlit" }
```

Since SRPDefaultUnlit is treated as an “extra” pass, URP will view shaders that use SRPDefaultUnlit as multi-pass shaders. One important thing to note is that the SRP batcher will be unable to batch draw calls with materials that use multi-pass shaders.

This can be verified with the Frame Debugger (Window -> Analysis Frame Debugger).

With our current setup, we can see that the draw calls are not being batched by the SRP batcher

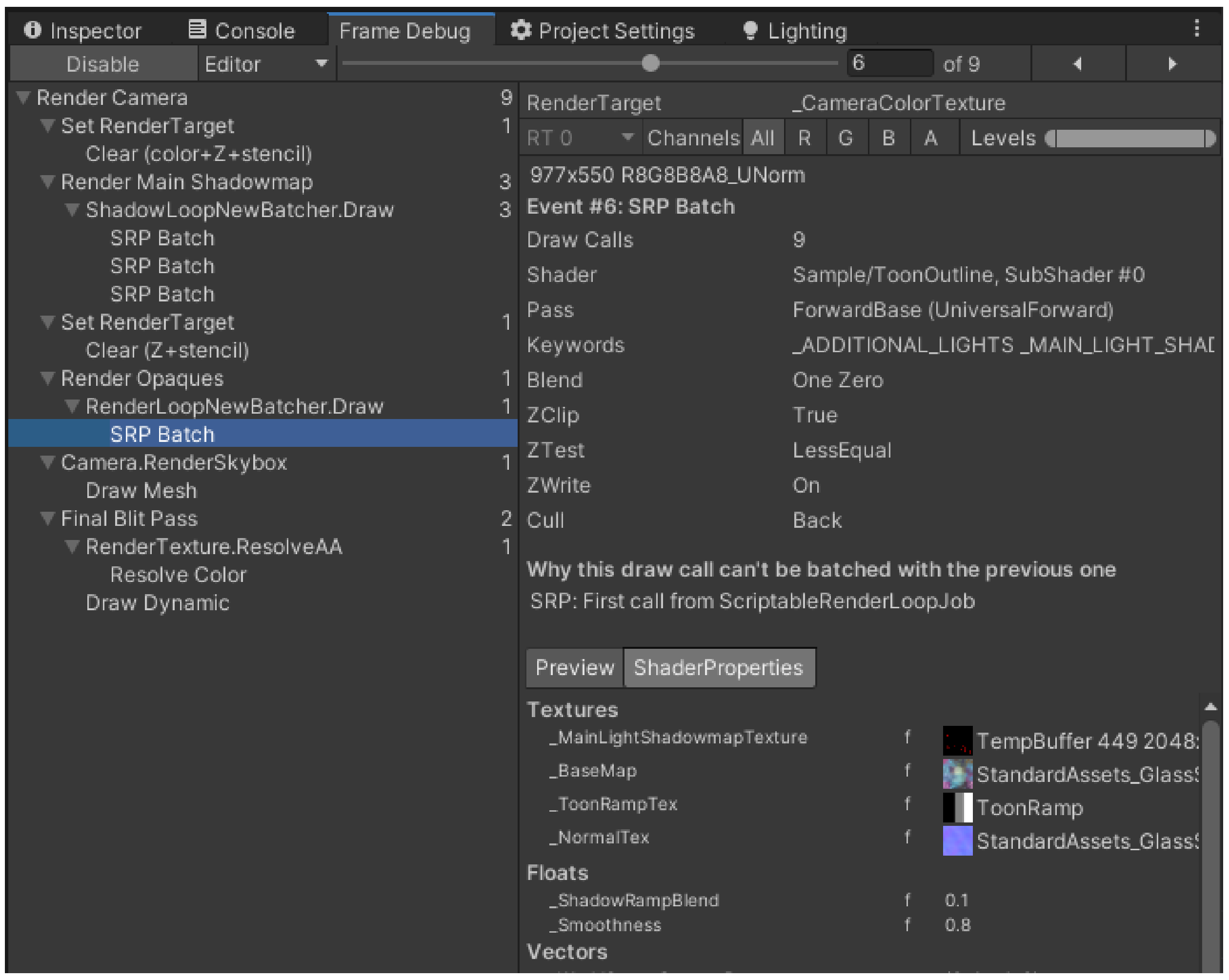
- The message “**SRP: Node use multi-pass shader**” is given as the reason, indicating that the material uses a multi-pass shader.



```
Inspector   Console   Frame Debug   Project Settings   Lighting
Disable   Editor        23   of 26
Render Camera                                26
  Set RenderTarget                            1
    Clear (color+Z+stencil)
  Render Main Shadowmap                       3
    ShadowLoopNewBatcher.Draw                 3
      SRP Batch
      SRP Batch
      SRP Batch
  Set RenderTarget                            1
    Clear (Z+stencil)
  Render Opaques                             18
    RenderLoopNewBatcher.Draw                18
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
      SRP Batch
  Camera.RenderSkybox                         1
    Draw Mesh
  Final Blit Pass                             2
    RenderTexture.ResolveAA                   1
      Resolve Color
    Draw Dynamic
```

```
RenderTarget    _CameraColorTexture
RT 0   Channels All R G B A   Levels
977x550 R8G8B8A8_UNorm
Event #23: SRP Batch
Shader          Sample/ToonOutline, SubShader #0
Pass            Outline (SRPDEFAULTUNLIT)
Blend           One Zero
ZClip           True
ZTest           LessEqual
ZWrite          On
Cull            Front

Why this draw call can't be batched with the previous one
 SRP: Node use multi-pass shader

Preview  ShaderProperties
Floats
  _OutlineThickness     v   3
Vectors
  _ScreenParams         v   (977, 550, 1.001024, 1.001818)
  _OutlineColor         f   (0, 0, 0, 1)
Matrices
  unity_MatrixVP        v   0.98    0        0        0
                            0       -1.7     -0.15    1.1
                            0       2.6e-05  -0.0003  0.3
                            0       -0.087   1        4.1
```

If we comment out the Outline pass in ToonOutline.shader and look at the Frame Debugger again, we can see that the draw calls are now batched.



```
Inspector   Console   Frame Debug   Project Settings   Lighting
Disable   Editor        6   of 9
Render Camera                                 9
  Set RenderTarget                            1
    Clear (color+Z+stencil)
  Render Main Shadowmap                       3
    ShadowLoopNewBatcher.Draw                 3
      SRP Batch
      SRP Batch
      SRP Batch
  Set RenderTarget                            1
    Clear (Z+stencil)
  Render Opaques                              1
    RenderLoopNewBatcher.Draw                 1
      SRP Batch
  Camera.RenderSkybox                         1
    Draw Mesh
  Final Blit Pass                             2
    RenderTexture.ResolveAA                   1
      Resolve Color
    Draw Dynamic
```

```
RenderTarget    _CameraColorTexture
RT 0   Channels All R G B A   Levels
977x550 R8G8B8A8_UNorm
Event #6: SRP Batch
Draw Calls      9
Shader          Sample/ToonOutline, SubShader #0
Pass            ForwardBase (UniversalForward)
Keywords        _ADDITIONAL_LIGHTS _MAIN_LIGHT_SHAI
Blend           One Zero
ZClip           True
ZTest           LessEqual
ZWrite          On
Cull            Back

Why this draw call can't be batched with the previous one
 SRP: First call from ScriptableRenderLoopJob

Preview  ShaderProperties
Textures
  _MainLightShadowmapTexture   f   TempBuffer 449 2048:
  _BaseMap                     f   StandardAssets_Glass
  _ToonRampTex                 f   ToonRamp
  _NormalTex                   f   StandardAssets_Glass
Floats
  _ShadowRampBlend             f   0.1
  _Smoothness                  f   0.8
Vectors
```

To keep SRP batcher compatibility, it may be a good idea to look into a different method of achieving the outline, such as:

- Doing the outline as a post-process effect
- Adding a separate material that only handles the outline