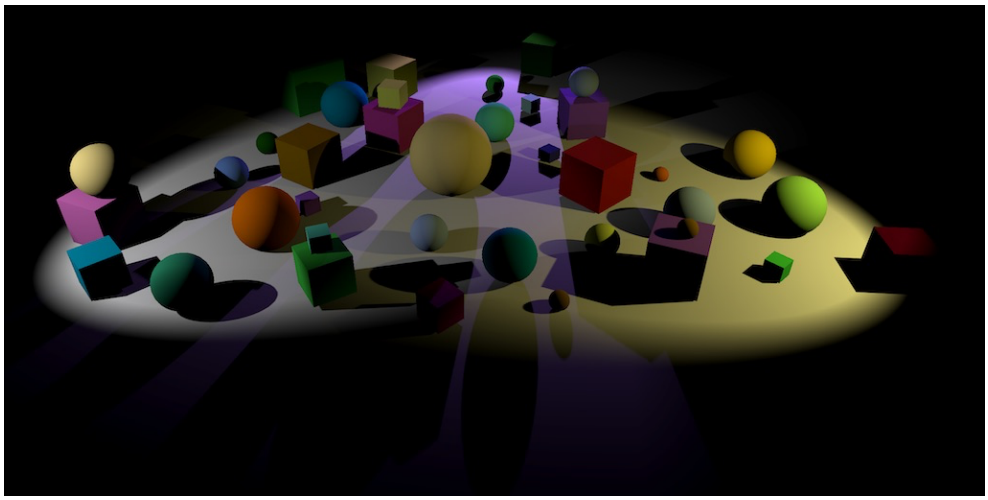
Spotlight Shadows Shadow Maps

Both render to and read from a texture.
Render from the point of view of a light.
Add a shader pass for shadow casters.
Sample a shadow map.
Support a mix of hard and soft shadows.
Combine up to sixteen shadow maps in a single atlas.

This is the fourth installment of a tutorial series covering Unity's scriptable render pipeline. In it, we'll add support for up to sixteen spotlights with shadows.

This tutorial is made with Unity 2018.3.0f2.



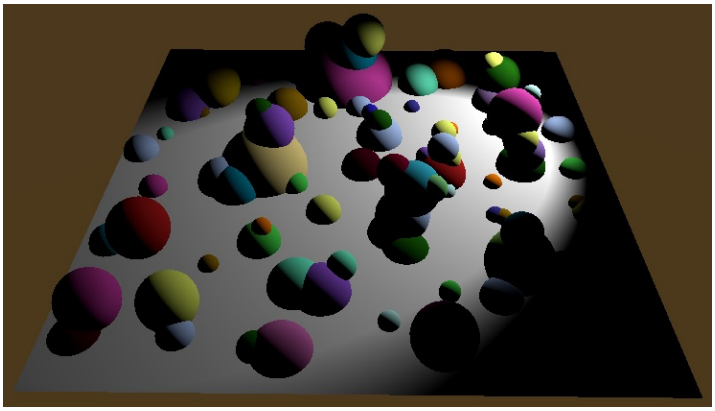
Three spotlights with shadows.

1 A Spotlight With Shadows

Shadows are very important, both to increase realism and to make the spatial relationship between objects more obvious. Without shadows, it can be hard to tell whether something floats about a surface or sits on top of it.

The Rendering 7, Shadows tutorial explains how shadows work in Unity's default render pipeline, but that exact same approach doesn't work for our single-pass forward renderer. It is still useful to skim to get the gist of shadow maps though. In this tutorial we'll limit ourselves to shadows for spotlights, are they're the least complicated.

We start with supporting exactly one shadowed light, so make a scene that contains a few objects and a single spotlight. A plane object is useful to receive shadows. All objects should use our *Lit Opaque* material.



A single spotlight, no shadows yet.

1.1 Shadow Map

There are a few different ways to deal with shadows, but we'll stick to the default approach of using a shadow map. This means that we'll render the scene from the light's point of view. We're only interested in the depth information of this render, because that tells us how far the light reaches before it hits a surface. Anything that's further away lies in shadows.

To use a shadow map we have to create it before rendering with the normal camera. To be able to sample the shadow map later, we have to render into a separate render texture instead of the usual frame buffer. Add a `RenderTexture` field to `MyPipeline` to keep a reference to the shadow map texture.

```
	RenderTexture shadowMap;
```

Create a separate method to render the shadows, with the context as a parameter. The first thing it has to do is get hold of a render texture. We'll do that by invoking the static `RenderTexture.GetTemporary` method. That either creates a new render texture or reuses an old one that hasn't been cleaned up yet. As we'll most likely need the shadow map every frame, it will get reused all the time.

Supply `RenderTexture.GetTemporary` with our map's width and height, the amount of bits used for the depth channel, and finally the texture format. We'll start with a fixed size of 512×512. We'll use 16 bits for the depth channel, so it is high-precision. As we're creating a shadow map, use the `RenderTextureFormat.Shadowmap` format.

```
	void RenderShadows (ScriptableRenderContext context) {
		shadowMap = RenderTexture.GetTemporary(
			512, 512, 16, RenderTextureFormat.Shadowmap
		);
	}
```

Make sure that the texture's filter mode is set the bilinear and its wrap mode is set to clamp.

```
		shadowMap = RenderTexture.GetTemporary(
			512, 512, 16, RenderTextureFormat.Shadowmap
		);
		shadowMap.filterMode = FilterMode.Bilinear;
		shadowMap.wrapMode = TextureWrapMode.Clamp;
```

The shadow map is to be rendered before the regular scene, so invoke `RenderShadows` in `Render` before we setup the regular camera, but after culling.

```
	void Render (ScriptableRenderContext context, Camera camera) {
		…
		CullResults.Cull(ref cullingParameters, context, ref cull);

		RenderShadows(context);

		context.SetupCameraProperties(camera);

		…
	}
```

Also, make sure to release the render texture when we're done, after we've submitted the context. If we have a shadow map at that point, pass it to the `RenderTexture.ReleaseTemporary` method and clear our field.

```
	void Render (ScriptableRenderContext context, Camera camera) {
		…

		context.Submit();

		if (shadowMap) {
			RenderTexture.ReleaseTemporary(shadowMap);
			shadowMap = null;
		}
	}
```

1.2 Shadow Command Buffer

We'll use a separate command buffer for all the shadow work, so we can see the shadow and regular rendering in separate sections in the frame debugger.

```
	CommandBuffer cameraBuffer = new CommandBuffer {
		name = "Render Camera"
	};

	CommandBuffer shadowBuffer = new CommandBuffer {
		name = "Render Shadows"
	};
```

The shadow rendering will happen in between `BeginSample` and `EndSample` commands, just like we do for regular rendering.

```
	void RenderShadows (ScriptableRenderContext context) {
		shadowMap = RenderTexture.GetTemporary(
			512, 512, 16, RenderTextureFormat.Shadowmap
		);
		shadowMap.filterMode = FilterMode.Bilinear;
		shadowMap.wrapMode = TextureWrapMode.Clamp;

		shadowBuffer.BeginSample("Render Shadows");
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();

		shadowBuffer.EndSample("Render Shadows");
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();
	}
```

1.3 Setting the Render Target

Before we can render shadows, we first have tell the GPU to render to our shadow map. A convenient way to do this is by invoking `CoreUtils.SetRenderTarget` with our command buffer and shadow map as arguments. As we start with clearing the map, invoke it before `BeginSample` so the frame debugger doesn't show and extra nested *Render Shadows* level.

```
		CoreUtils.SetRenderTarget(shadowBuffer, shadowMap);
		shadowBuffer.BeginSample("Render Shadows");
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();
```

We only care about the depth channel, so only that channel needs to be cleared. Indicate this by adding `ClearFlag.Depth` as a third argument to `SetRenderTarget`.

```
		CoreUtils.SetRenderTarget(
			shadowBuffer, shadowMap,
			ClearFlag.Depth
		);
```

While not necessary, we can also be more precise about the load and storage requirements of our texture. We don't care where it comes from, as we clear it anyway, which we can indicate with `RenderBufferLoadAction.DontCare`. That makes it possible for tile-based GPUs to be a bit more efficient. And we need to sample from the texture later, so it needs to be kept in memory, which we indicate with `RenderBufferStoreAction.Store`. Add these as the third and fourth arguments.

```
		CoreUtils.SetRenderTarget(
			shadowBuffer, shadowMap,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,
			ClearFlag.Depth
		);
```

The clear action for our shadow map now shows up in the frame debugger, before the regular camera render.

Clearing the shadow map.

1.4 Configuring the View and Projection Matrices

The idea is that we render from the point of view of the light source, which means that we're using the spotlight as if it were a camera. Thus, we have to provide appropriate view and projection matrices. We can retrieve these matrices by invoking `ComputeSpotShadowMatricesAndCullingPrimitives` on our cull results with the light index as an argument. As we only have a single spotlight in the scene, we simply supply zero. The view and projection matrices are made available via two output parameters. Besides that, there is a third `ShadowSplitData` output parameter. We don't need it, must supply the output argument.

```
		shadowBuffer.BeginSample("Render Shadows");
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();

		Matrix4x4 viewMatrix, projectionMatrix;
		ShadowSplitData splitData;
		cull.ComputeSpotShadowMatricesAndCullingPrimitives(
			0, out viewMatrix, out projectionMatrix, out splitData
		);
```

Once we have the matrices, set them up by invoking `SetViewProjectionMatrices` on the shadow command buffer, execute it, and clear it.

```
		cull.ComputeSpotShadowMatricesAndCullingPrimitives(
			0, out viewMatrix, out projectionMatrix, out splitData
		);
		shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();
```

1.5 Rendering Shadow Casters

With the correct matrices in place, we can move on to rendering all the shadow-casting objects. We do that by invoking `DrawShadows` on the context. That method has a `DrawShadowsSettings` reference parameter, which we can create via a constructor method that takes the cull results and light index as parameters.

```
		shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();

		var shadowSettings = new DrawShadowsSettings(cull, 0);
		context.DrawShadows(ref shadowSettings);
```

This only works if our spotlight's *Shadow Type* is set to either hard or soft. If it is set to none then Unity will complain that it isn't a valid shadow-casting light.

Light with shadows enabled.

2 Shadow Caster Pass

At this point all objects that are affected by our light should get rendered to the shadow map, but the frame debugger tells us it isn't happening. That's because `DrawShadows` uses the *ShadowCaster* shader pass and our shader currently doesn't have such a pass.

2.1 Shadow Include File

To create a shadow-caster pass, duplicate the *Lit.hlsl* file and rename it to *ShadowCaster.hlsl*. We only care about depth information, so remove everything from the new file that doesn't relate to the fragment position. The output of the fragment program is simply zero. Also rename its pass functions and include guard define.

```
#ifndef MYRP_SHADOWCASTER_INCLUDED
#define MYRP_SHADOWCASTER_INCLUDED

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"

CBUFFER_START(UnityPerFrame)
	float4x4 unity_MatrixVP;
CBUFFER_END

CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
CBUFFER_END

#define UNITY_MATRIX_M unity_ObjectToWorld

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/UnityInstancing.hlsl"

struct VertexInput {
	float4 pos : POSITION;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
};

VertexOutput ShadowCasterPassVertex (VertexInput input) {
	VertexOutput output;
	UNITY_SETUP_INSTANCE_ID(input);
	float4 worldPos = mul(UNITY_MATRIX_M, float4(input.pos.xyz, 1.0));
	output.clipPos = mul(unity_MatrixVP, worldPos);
	return output;
}

float4 ShadowCasterPassFragment (VertexOutput input) : SV_TARGET {
	return 0;
}

#endif // MYRP_SHADOWCASTER_INCLUDED
```

This is sufficient to render shadows, but it is possible for shadow casters to intersect the near place, which can cause holes to appear in shadows. To prevent this, we have to clamp the vertices to the near place in the vertex program. This is done by taking the maximum of the Z coordinate and the W coordinate of the clip-space position.

```
	output.clipPos = mul(unity_MatrixVP, worldPos);

	output.clipPos.z = max(output.clipPos.z, output.clipPos.w);
	return output;
```

However, this is complicated by the specifics of the clip space. It's most intuitive to think of the depth value at the near clip plane as zero and increasing the further away it is. But it's actually the reverse for all but OpenGL APIs, with the value being 1 at the near plane. And for OpenGL the near plane value is −1. We can cover all cases by relying on the `UNITY_REVERSED_Z` and `UNITY_NEAR_CLIP_VALUE` macros that we included via *Common.hlsl*.

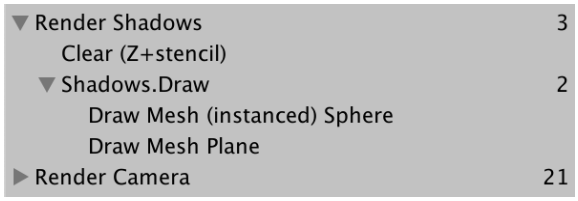
```
	//output.clipPos.z = max(output.clipPos.z, output.clipPos.w);
	#if UNITY_REVERSED_Z
		output.clipPos.z =
			min(output.clipPos.z, output.clipPos.w * UNITY_NEAR_CLIP_VALUE);
	#else
		output.clipPos.z =
			max(output.clipPos.z, output.clipPos.w * UNITY_NEAR_CLIP_VALUE);
	#endif
```

2.2 A Second Pass

To add the *ShadowCaster* pass to our *Lit* shader, we duplicate its pass block and give the second pass a `Tags` block in which we set *LightMode* to *ShadowCaster*. Then have it include *ShadowCaster.hlsl* instead of *Lit.hlsl* and use the appropriate vertex and fragment functions.

```
		Pass {
			HLSLPROGRAM

			#pragma target 3.5

			#pragma multi_compile_instancing
			#pragma instancing_options assumeuniformscaling

			#pragma vertex LitPassVertex
			#pragma fragment LitPassFragment

			#include "../ShaderLibrary/Lit.hlsl"

			ENDHLSL
		}

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

			HLSLPROGRAM

			#pragma target 3.5

			#pragma multi_compile_instancing
			#pragma instancing_options assumeuniformscaling

			#pragma vertex ShadowCasterPassVertex
			#pragma fragment ShadowCasterPassFragment

			#include "../ShaderLibrary/ShadowCaster.hlsl"

			ENDHLSL
		}
```

Now our objects get rendered into the shadow map. As the objects aren't affected by multiply lights at this point, GPU instancing is very effective.



All shadow casters rendered with two draw calls.

By selecting the *Shadows.Draw* entry you can see the final contents of the shadow map. As it is a depth-only texture, the frame debugger will show us the depth information, with white representing near and black far.

Rendered shadow map.

Because the shadow map is rendered with the spotlight acting as the camera, its orientation matches the light's. If the light has been rotated so its local up direction points down in world space, the shadow map will be upside down too.

3 Sampling the Shadow Map

At this point we have a shadow map with all the data that we need, but we're not using it yet. The next step is to sample the shadow map when rendering our objects the normal way.

3.1 From World Space to Shadow Space

The depth information stored in the shadow map is valid for the clip space that we used when rendering the map. We'll call that shadow space. It doesn't match the spaces that we use when rendering objects the normal way. To know where a fragment is relative to the shadow depth that we stored, we have to convert the fragment's position to shadow space.

Step one is to make the shadow map itself available to our shader. We do that via a shader texture variable, which we'll name *_ShadowMap*. Keep track of its identifier in `MyPipeline`.

```
	static int shadowMapId = Shader.PropertyToID("_ShadowMap");
```

Bind the shadow map to this variable globally at the end of `RenderShadows`, by invoking `SetGlobalTexture` on the shadow command buffer before it gets executed the last time.

```
		shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
		shadowBuffer.EndSample("Render Shadows");
```

Next, we'll add a shader matrix variable to convert from world space to shadow space, named *_WorldToShadowMatrix*. Keep track of its identifier too.

```
	static int worldToShadowMatrixId =
		Shader.PropertyToID("_WorldToShadowMatrix");
```

This matrix is found by multiplying the view and projection matrix that we used when rendering the shadows, then passing it to the GPU via `SetGlobalMatrix`.

```
		Matrix4x4 worldToShadowMatrix = projectionMatrix * viewMatrix;
		shadowBuffer.SetGlobalMatrix(worldToShadowMatrixId, worldToShadowMatrix);
		shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
```

However, again there is a difference based on whether the clip-space Z dimension is reversed, which we can check via `SystemInfo.usesReversedZBuffer`. If so, we have to negate the Z-component row—the row with index 2—of the projection matrix before multiplying. We can do that by directly adjusting the `m20` through `m23` fields of the matrix.

```
		if (SystemInfo.usesReversedZBuffer) {
			projectionMatrix.m20 = -projectionMatrix.m20;
			projectionMatrix.m21 = -projectionMatrix.m21;
			projectionMatrix.m22 = -projectionMatrix.m22;
			projectionMatrix.m23 = -projectionMatrix.m23;
		}
		Matrix4x4 worldToShadowMatrix = projectionMatrix * viewMatrix;
```

We now have a conversion matrix from world space to shadow clip space. But clip space goes from −1 to 1, while texture coordinates and depth go from 0 to 1. We can bake that range conversion into our matrix, via an additional multiplication with a matrix that scales and offsets by half a unit in all dimensions. We could use the `Matrix4x4.TRS` method to get such a matrix by providing a offset, rotation, and scale.

```
		var scaleOffset = Matrix4x4.TRS(
			Vector3.one * 0.5f, Quaternion.identity, Vector3.one * 0.5f
		);
		Matrix4x4 worldToShadowMatrix =
			scaleOffset * (projectionMatrix * viewMatrix);
```

But as it is a simple matrix, we can also simply start with the identity matrix and set the appropriate fields.

```
		var scaleOffset = Matrix4x4.identity;
		scaleOffset.m00 = scaleOffset.m11 = scaleOffset.m22 = 0.5f;
		scaleOffset.m03 = scaleOffset.m13 = scaleOffset.m23 = 0.5f;
```

3.2 Sampling Depth

In *Lit.hlsl*, add a buffer for light data and define `float4x4 _WorldToShadowMatrix` in it.

```
CBUFFER_START(_LightBuffer)
	…
CBUFFER_END

CBUFFER_START(_ShadowBuffer)
	float4x4 _WorldToShadowMatrix;
CBUFFER_END
```

Texture resources aren't part of buffers. Instead they're defined separately. In this case, we can use the `TEXTURE2D_SHADOW` macro to define `_ShadowMap`.

```
CBUFFER_START(_ShadowBuffer)
	float4x4 _WorldToShadowMatrix;
CBUFFER_END

TEXTURE2D_SHADOW(_ShadowMap);
```

What's the difference between `TEXTURE2D` and `TEXTURE2D_SHADOW`?

There is only a difference for OpenGL ES 2.0, because it doesn't support depth comparisons for shadow maps. But we don't support OpenGL ES 2.0, so we could've used `TEXTURE2D` instead. I used `TEXTURE2D_SHADOW` anyway to make it abundantly clear that we are dealing with shadow data.

The macros are defines per target platform via separate API include files in the Core library, which we included via *Common.hlsl*.

Next, we have to also define the sampler state used for sampling the texture. Usually this is done with the `SAMPLER` macro, but we're going to use a special comparison sampler so use `SAMPLER_CMP` instead. To end up with the correct sampler state, we have to give it the same name as the texture, with *sampler* written in front of it.

```
TEXTURE2D_SHADOW(_ShadowMap);
SAMPLER_CMP(sampler_ShadowMap);
```

What is a texture sampler?

In old GLSL code, we use `sampler2D` to define both a texture and a sampler state together. But they are two separate things, and both take up resources. Sampler states exist separate from textures, which makes it possible to mix their use, typically reusing the same sampler state to sample from multiple textures.

In our case, we set the sampler state to use clamping and bilinear filtering, via `MyPipeline`.

The comparison sampler that we're using will perform a depth comparison for us, before bilinear interpolation happens. That produces better results than first interpolating and then comparing.

Create a `ShadowAttenuation` function with the world position as a parameter. It will return the attenuation factor for our light's shadows. The first thing it needs to do is convert the world position to the shadow position.

```
TEXTURE2D_SHADOW(_ShadowMap);
SAMPLER_CMP(sampler_ShadowMap);

float ShadowAttenuation (float3 worldPos) {
	float4 shadowPos = mul(_WorldToShadowMatrix, float4(worldPos, 1.0));
}
```

The resulting position is defined with homogeneous coordinates, just like when we convert to clip space. But we need regular coordinates, so divide XYZ components by its W component.

```
	float4 shadowPos = mul(_WorldToShadowMatrix, float4(worldPos, 1.0));
	shadowPos.xyz /= shadowPos.w;
```

Now we can sample the shadow map, using the `SAMPLE_TEXTURE2D_SHADOW` macro. It needs the texture, the sampler state, and the shadow position as arguments. The result is 1 when the position's Z value is less than what's stored in the shadow map, meaning that it is closer to the light than whatever's casting a shadow. Otherwise, it is behind a shadow caster and the result is zero. Because the sampler performs the comparison before bilinear interpolation, the edges of shadows will blend across shadow map texels.

```
	shadowPos.xyz /= shadowPos.w;
	return SAMPLE_TEXTURE2D_SHADOW(_ShadowMap, sampler_ShadowMap, shadowPos.xyz);
```

3.3 Fading when Shadowed

To affect the lighting, add a parameter for the shadow attenuation to the `DiffuseLight` function. Factor it into the diffuse strength, along with the other fade factors.

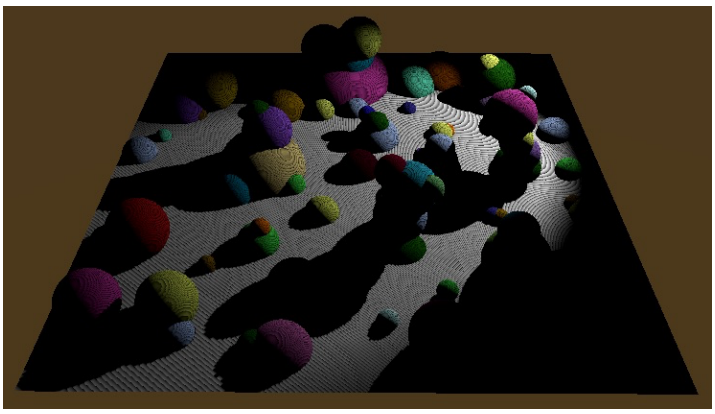
```
float3 DiffuseLight (
	int index, float3 normal, float3 worldPos, float shadowAttenuation
) {
	…
	diffuse *= shadowAttenuation * spotFade * rangeFade / distanceSqr;

	return diffuse * lightColor;
}
```

Shadows don't work for vertex lighting, so use 1 for the shadow attenuation in `LitPassVertex`.

```
VertexOutput LitPassVertex (VertexInput input) {
	…

	output.vertexLighting = 0;
	for (int i = 4; i < min(unity_LightIndicesOffsetAndCount.y, 8); i++) {
		int lightIndex = unity_4LightIndices1[i - 4];
		output.vertexLighting +=
			DiffuseLight(lightIndex, output.normal, output.worldPos, 1);
	}

	return output;
}
```

In `LitPassFragment`, invoke `ShadowAttenuation` with the world position as an argument and pass the result to `DiffuseLight`.

```
float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	…

	float3 diffuseLight = input.vertexLighting;
	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		float shadowAttenuation = ShadowAttenuation(input.worldPos);
		diffuseLight += DiffuseLight(
			lightIndex, input.normal, input.worldPos, shadowAttenuation
		);
	}
	float3 color = diffuseLight * albedo;
	return float4(color, 1);
}
```

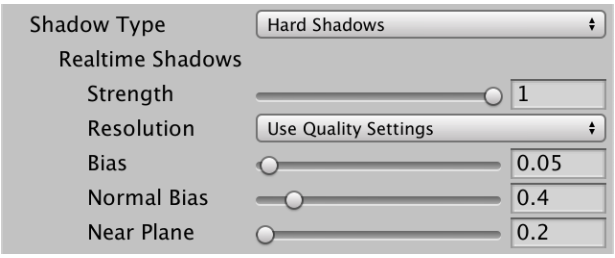


Sampled shadows.

Shadows finally appear, but with a severe case of shadow acne.

4 Shadow Settings

There are various ways to control the quality and appearance of the shadows. We'll add support for a few, specifically the shadow resolution, depth bias, strength, and soft shadows. These and more can be configured via each light's inspector.



Shadow settings per light.

4.1 Shadow Map Size

Although the light inspector has an option for its shadow resolution, this only indirectly controls the size of the shadow map. The actual size is set via the quality settings, at least for Unity's default pipeline. We use our own pipeline, so we'll add a shadow map size configuration option to `MyPipelineAsset`.

The shadow map is a square texture and we'll allow power-of-two sizes from 256×256 up to 4096×.4096. To make only those options available, define a `ShadowMapSize` enum inside `MyPipelineAsset` with the elements 256, 512, 1024, 2048, and 4096. Numbers cannot be used for enumeration labels, so prefix each with an underscore. The Unity editor will omit the underscores when displaying the enum's options. Then use the enum to add a configuration field for the shadow map size.

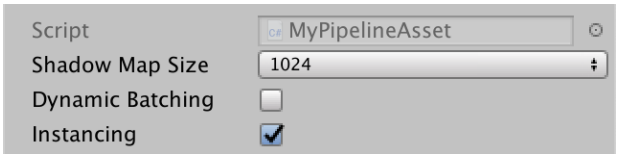
```
	public enum ShadowMapSize {
		_256,
		_512,
		_1024,
		_2048,
		_4096
	}

	[SerializeField]
	ShadowMapSize shadowMapSize;
```

By default, enums represent integers and start at zero. It's more convenient if our enumeration options directly map to the same integer, which we can do by assigning explicit values to them.

```
	public enum ShadowMapSize {
		_256 = 256,
		_512 = 512,
		_1024 = 1024,
		_2048 = 2048,
		_4096 = 4096
	}
```

This means that zero isn't a valid default value, so set the default to something else.

```
	ShadowMapSize shadowMapSize = ShadowMapSize._1024;
```



Shadow map size set to 1024.

Pass the shadow map size to our pipeline's constructor method, cast to an integer.

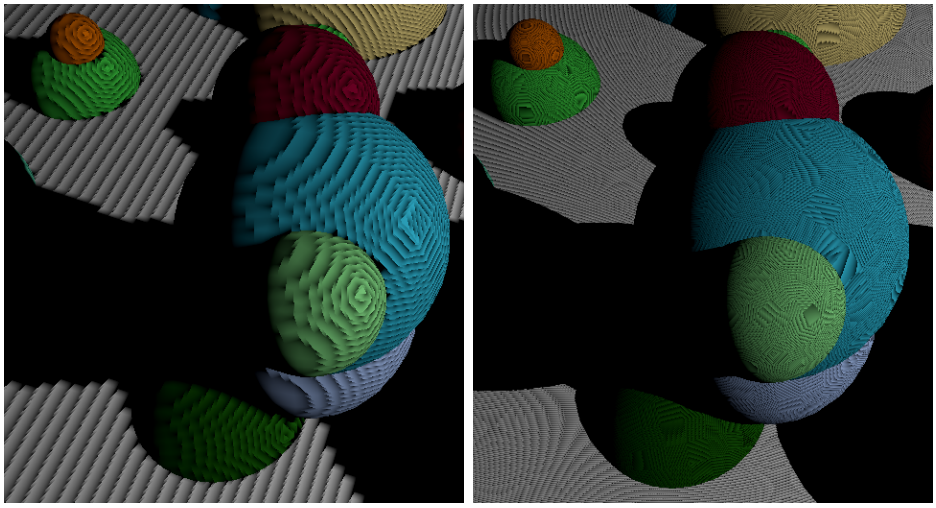
```
	protected override IRenderPipeline InternalCreatePipeline () {
		return new MyPipeline(
			dynamicBatching, instancing, (int)shadowMapSize
		);
	}
```

And add a field to keep track of the size to `MyPipeline`, initializing it in the constructor.

```
	int shadowMapSize;

	public MyPipeline (
		bool dynamicBatching, bool instancing, int shadowMapSize
	) {
		…
		this.shadowMapSize = shadowMapSize;
	}
```

When fetching a render texture in `RenderShadows`, we'll now use the variable shadow map size.

```
	void RenderShadows (ScriptableRenderContext context) {
		shadowMap = RenderTexture.GetTemporary(
			shadowMapSize, shadowMapSize, 16, RenderTextureFormat.Shadowmap
		);
		…
	}
```



Shadow map size 256 and 4096.

4.2 Shadow Bias

Shadow acne is caused by texels of the shadow map poking out of surfaces. For a more detailed explanation, see Rendering 7, Shadows. We'll support the simplest way to mitigate acne, which is by adding a small depth offset when rendering to the shadow map. This shadow bias is configured per light, so we must send it to the GPU. We'll do that via a *_ShadowBias* shader property, so keep track of its identifier.

```
	static int shadowBiasId = Shader.PropertyToID("_ShadowBias");
```

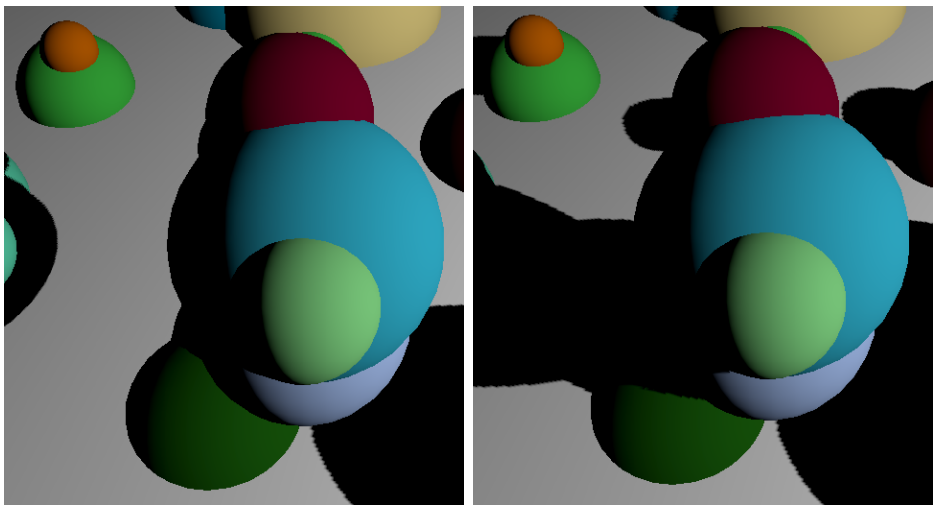
When setting the view and projection matrices in `RenderShadows`, also set the shadow bias. The `VisibleLight` struct doesn't contain this information, but it does have a `light` field that holds a reference to the `Light` component, which has the shadow bias.

```
		shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
		shadowBuffer.SetGlobalFloat(
			shadowBiasId, cull.visibleLights[0].light.shadowBias
		);
		context.ExecuteCommandBuffer(shadowBuffer);

		shadowBuffer.Clear();
```

Add the corresponding variable to *ShadowCaster.hlsl*, in a shadow caster buffer. Use it to offset the Z component of the clip-space position, before clamping. If Z is reversed, the bias should be subtracted, otherwise it is added.

```
CBUFFER_START(_ShadowCasterBuffer)
	float _ShadowBias;
CBUFFER_END

…

VertexOutput ShadowCasterPassVertex (VertexInput input) {
	…
	output.clipPos = mul(unity_MatrixVP, worldPos);

	#if UNITY_REVERSED_Z
		output.clipPos.z -= _ShadowBias;
		output.clipPos.z =
			min(output.clipPos.z, output.clipPos.w * UNITY_NEAR_CLIP_VALUE);
	#else
		output.clipPos.z += _ShadowBias;
		output.clipPos.z =
			max(output.clipPos.z, output.clipPos.w * UNITY_NEAR_CLIP_VALUE);
	#endif
	return output;
}
```

The shadow bias should be as small as possible, to prevent shadows from moving too far away, which causes peter-panning.



Shadow bias 0.05 and 0.01.

4.3 Shadow Strength

As we're only using a single light and don't have any environmental lighting, our shadows are completely black. But we could tone down the strength of the shadow attenuation, making it only partially fade the light's contribution. It would be like all shadow casters are semitransparent. We'll send the shadow strength to the shader via a *_ShadowStrength* property, so keep track of its identifier.

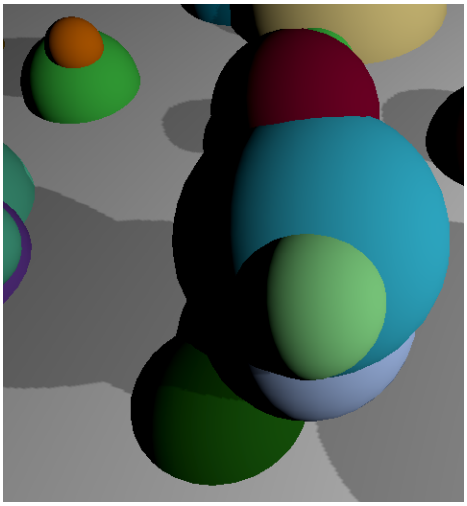
```
	static int shadowStrengthId = Shader.PropertyToID("_ShadowStrength");
```

The shadow strength is used when sampling the shadow map, so set it along with the world-to-shadow matrix and the shadow map itself. Like the depth bias, we can retrieve it from the `Light` component.

```
		shadowBuffer.SetGlobalMatrix(worldToShadowMatrixId, worldToShadowMatrix);
		shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
		shadowBuffer.SetGlobalFloat(
			shadowStrengthId, cull.visibleLights[0].light.shadowStrength
		);
		shadowBuffer.EndSample("Render Shadows");
```

Add the shadow strength to the shadow buffer, then use it to interpolate between 1 and the sampled attenuation in `ShadowAttenuation`.

```
CBUFFER_START(_ShadowBuffer)
	float4x4 _WorldToShadowMatrix;
	float _ShadowStrength;
CBUFFER_END

TEXTURE2D_SHADOW(_ShadowMap);
SAMPLER_CMP(sampler_ShadowMap);

float ShadowAttenuation (float3 worldPos) {
	float4 shadowPos = mul(_WorldToShadowMatrix, float4(worldPos, 1.0));
	shadowPos.xyz /= shadowPos.w;
	float attenuation =
		SAMPLE_TEXTURE2D_SHADOW(_ShadowMap, sampler_ShadowMap, shadowPos.xyz);
	return lerp(1, attenuation, _ShadowStrength);
}
```



Shadow strength set to 0.5.

4.4 Soft Shadows

The final setting that we will support is to switch between hard and soft shadows. We're currently using hard shadows, meaning that the only smoothing of the shadow's edge is caused by the bilinear interpolation while sampling the shadow map. When smooth shadows are enabled, the shadow transition is blurred, representing shadows with a larger penumbra. However, unlike in real life the penumbra will be uniform, instead of depending on the spatial relationship between light source, shadow caster, and shadow receiver.

Soft shadows are made by sampling the shadow map more than once, with samples further away from the original sample position contributing less to the final value. We'll use a 5×5 tent filter, requiring nine texture samples. We can use a function for this that is defined in the *Shadow/ShadowSamplingTent.hlsl* include file in the core library. Include it in *Lit.hlsl*.

```
#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"
#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Shadow/ShadowSamplingTent.hlsl"
```

How does a tent filter work?

The Bloom tutorial covers filter kernels that take advantage of bilinear texture sampling, while the Depth of Field tutorial contains an example of a 3×3 tent filter.

The tent filter requires us to know the size of the shadow map. The function that we'll use specifically requires a vector with the map's inverted with and height and the regular with and height in its four components. So add it to the shadow buffer.

```
CBUFFER_START(_ShadowBuffer)
	float4x4 _WorldToShadowMatrix;
	float _ShadowStrength;
	float4 _ShadowMapSize;
CBUFFER_END
```

Keep track of the corresponding identifier in `MyPipeline`.

```
	static int shadowMapSizeId = Shader.PropertyToID("_ShadowMapSize");
```

And set it at the end of `RenderShadows`.

```
		float invShadowMapSize = 1f / shadowMapSize;
		shadowBuffer.SetGlobalVector(
			shadowMapSizeId, new Vector4(
				invShadowMapSize, invShadowMapSize, shadowMapSize, shadowMapSize
			)
		);
		shadowBuffer.EndSample("Render Shadows");
```

We'll replace the regular sampling of the shadow map in the `ShadowAttenuation` function with our tent filter when the *_SHADOWS_SOFT* shader keyword is defined.

```
	float attenuation =
		SAMPLE_TEXTURE2D_SHADOW(_ShadowMap, sampler_ShadowMap, shadowPos.xyz);

	#if defined(_SHADOWS_SOFT)
	#endif

	return lerp(1, attenuation, _ShadowStrength);
```

Instead of a single sample, we'll have to accumulate nine samples to create the 5×5 tent filter. The `SampleShadow_ComputeSamples_Tent_5x5` function gives us the weights and UV coordinates to use, by passing the shadow map size and the XY coordinates of the shadow position as arguments. The weights and UV are provided via two output parameters, a `float` array and a `float2` array, both with nine elements.

```
	#if defined(_SHADOWS_SOFT)
		float tentWeights[9];
		float2 tentUVs[9];
		SampleShadow_ComputeSamples_Tent_5x5(
			_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs
		);
	#endif
```

However, the function is defined using `real` instead of `float` for its output parameters. That isn't an actual numeric type, but a macro that's used to create either a `float` or a `half` variant, as needed. We can usually ignore this, but to prevent compiler errors for some platforms, it's best to use `real` too for output parameters.

```
		real tentWeights[9];
		real2 tentUVs[9];
```

Now we can use a loop to sample the shadow map nine times, using the weight and UV coordinates from the arrays. It's a tight fixed loop, so the shader compiler will unroll it. We still need the Z coordinate of the shadow position, so use it to construct a `float3` for each shadow sample.

```
	#if defined(_SHADOWS_SOFT)
		real tentWeights[9];
		real2 tentUVs[9];
		SampleShadow_ComputeSamples_Tent_5x5(
			_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs
		);
		attenuation = 0;
		for (int i = 0; i < 9; i++) {
			attenuation += tentWeights[i] * SAMPLE_TEXTURE2D_SHADOW(
				_ShadowMap, sampler_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)
			);
		}
	#endif
```

To enable soft shadows, we have to create a shader pass variant for when the *_SHADOWS_SOFT* keyword is defined. That's done by adding a multi-compile pragma directive to the default pass of our *Lit* shader. We want it to generate two variants, one without and one with the keyword defined. We do that by writing an underscore to represent the variant without a keyword, followed by the *_SHADOWS_SOFT* keyword.

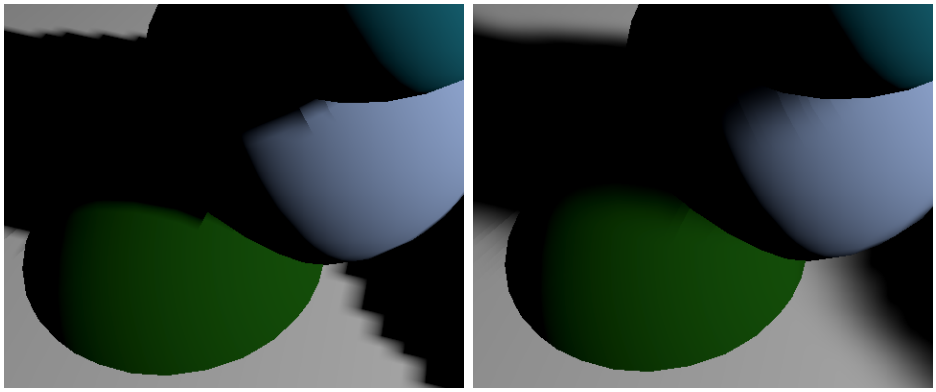
```
			#pragma multi_compile_instancing
			#pragma instancing_options assumeuniformscaling

			#pragma multi_compile _ _SHADOWS_SOFT
```

Finally, we have to toggle the keyword based on the value of the light's `shadows` property, at the end of `RenderShadows`. If it's set to `LightShadows.Soft` then invoke the `EnableShaderKeyword` method on our shadow buffer. Otherwise, invoke `DisableShaderKeyword`. Unity uses the keyword state to decide which pass variant to use when rendering.

```
	const string shadowsSoftKeyword = "_SHADOWS_SOFT";

	…

	void RenderShadows (ScriptableRenderContext context) {
		…

		if (cull.visibleLights[0].light.shadows == LightShadows.Soft) {
			shadowBuffer.EnableShaderKeyword(shadowsSoftKeyword);
		}
		else {
			shadowBuffer.DisableShaderKeyword(shadowsSoftKeyword);
		}
		shadowBuffer.EndSample("Render Shadows");
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();
	}
```

As it is a common to toggle a keyword based on a boolean, we could also use the `CoreUtils.SetKeyword` method to do the same.

```
		//if (cull.visibleLights[0].light.shadows == LightShadows.Soft) {
		//	shadowBuffer.EnableShaderKeyword(shadowsSoftKeyword);
		//}
		//else {
		//	shadowBuffer.DisableShaderKeyword(shadowsSoftKeyword);
		//}
		CoreUtils.SetKeyword(
			shadowBuffer, shadowsSoftKeyword,
			cull.visibleLights[0].light.shadows == LightShadows.Soft
		);
```



Hard and soft shadows.

5 More Lights With Shadows

So far we have only worked with a single spotlight, but our pipeline supports up to sixteen lights. We should also support up to sixteen spotlights that all cast shadows at the same time.

5.1 Shadow Data Per Light

If we're supporting multiple shadowed lights, while still doing all lighting in a single pass, then all shadow data—like shadow strength—must be made available at the same time. We'll collect this data in `ConfigureLights`, as we're setting up other light-related data there as well. So move the invocation of if to just before we invoke `RenderShadows`, and also only invoke `RenderShadows` if there are visible lights.

```
		CullResults.Cull(ref cullingParameters, context, ref cull);
		if (cull.visibleLights.Count > 0) {
			ConfigureLights();
			RenderShadows(context);
		}
		else {
			cameraBuffer.SetGlobalVector(
				lightIndicesOffsetAndCountID, Vector4.zero
			);
		}
		ConfigureLights();

		context.SetupCameraProperties(camera);

		CameraClearFlags clearFlags = camera.clearFlags;
		cameraBuffer.ClearRenderTarget(
			(clearFlags & CameraClearFlags.Depth) != 0,
			(clearFlags & CameraClearFlags.Color) != 0,
			camera.backgroundColor
		);

		//if (cull.visibleLights.Count > 0) {
		//	ConfigureLights();
		//}
		//else {
		//	cameraBuffer.SetGlobalVector(
		//		lightIndicesOffsetAndCountID, Vector4.zero
		//	);
		//}
```

We'll use an array of 4D vectors to store the shadow data, one element per light. Initialize each element to zero when looping through the lights in `ConfigureLights`, like we do for the attenuation data.

```
	Vector4[] shadowData = new Vector4[maxVisibleLights];

	…

	void ConfigureLights () {
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			if (i == maxVisibleLights) {
				break;
			}
			VisibleLight light = cull.visibleLights[i];
			visibleLightColors[i] = light.finalColor;
			Vector4 attenuation = Vector4.zero;
			attenuation.w = 1f;
			Vector4 shadow = Vector4.zero;

			…

			visibleLightAttenuations[i] = attenuation;
			shadowData[i] = shadow;
		}

		…
	}
```

When we have a spotlight, get a reference to its `Light` component. If its `shadows` property isn't set to `LightShadows.None`, then store its shadow strength in the X component of the vector.

```
				if (light.lightType == LightType.Spot) {
					…

					Light shadowLight = light.light;
					if (shadowLight.shadows != LightShadows.None) {
						shadow.x = shadowLight.shadowStrength;
					}
				}
```

As each light can either use hard or soft shadows, we'll store that in the vector's Y component. Use 1 for soft shadows and zero otherwise.

```
					if (shadowLight.shadows != LightShadows.None) {
						shadow.x = shadowLight.shadowStrength;
						shadow.y =
							shadowLight.shadows == LightShadows.Soft ? 1f : 0f;
					}
```

5.2 Excluding Lights

That a light is visible and has shadows enabled doesn't guarantee that it needs a shadow map. There might be no shadow casters or receivers in view. We can check this by invoking `GetShadowCasterBounds` on the cull results with the light index as a parameter. It returns whether it found valid bounds for the shadow volume of the light. If not, we can skip setting up the shadow data. It also provides the shadow bounds via an output parameter, which we'll have to provide even though we don't use it here.

```
Light shadowLight = light.light;
Bounds shadowBounds;
if (
    shadowLight.shadows != LightShadows.None &&
    cull.GetShadowCasterBounds(i, out shadowBounds)
) {
    shadow.x = shadowLight.shadowStrength;
    shadow.y =
        shadowLight.shadows == LightShadows.Soft ? 1f : 0f;
}
```

5.3 Rendering All Shadow Maps

Moving on to `RenderShadows`, we have to put a loop around the code between the first execution of the shadow buffer and setting the shadow map texture. Again we loop through all visible lights, aborting when we go beyond the maximum supported lights. Replace all usage of the hard-coded zero index in the loop with the iterator variable.

```
	shadowBuffer.BeginSample("Render Shadows");
	context.ExecuteCommandBuffer(shadowBuffer);
	shadowBuffer.Clear();

	for (int i = 0; i < cull.visibleLights.Count; i++) {
		if (i == maxVisibleLights) {
			break;
		}

		Matrix4x4 viewMatrix, projectionMatrix;
		ShadowSplitData splitData;
		cull.ComputeSpotShadowMatricesAndCullingPrimitives(
			i, out viewMatrix, out projectionMatrix, out splitData
		);
		shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
		shadowBuffer.SetGlobalFloat(
			shadowBiasId, cull.visibleLights[i].light.shadowBias
		);
		context.ExecuteCommandBuffer(shadowBuffer);
		shadowBuffer.Clear();

		var shadowSettings = new DrawShadowsSettings(cull, i);
		context.DrawShadows(ref shadowSettings);

		…
		shadowBuffer.SetGlobalMatrix(
			worldToShadowMatrixId, worldToShadowMatrix
		);
	}

	shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
```

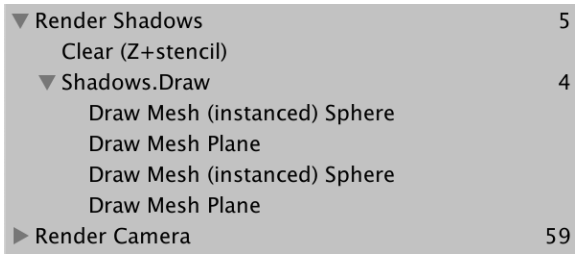
Each light that doesn't need a shadow map should be skipped. We can use the shadow strength that we put in the shadow data to determine this. If it's zero or less—either because that was the original strength or we left it at zero—directly go to the next iteration of the loop, by using the `continue` statement.

```
			if (i == maxVisibleLights) {
				break;
			}
			if (shadowData[i].x <= 0f) {
				continue;
			}
```

The `ComputeSpotShadowMatricesAndCullingPrimitives` method returns whether it was able to generate useful matrices. It should agree with the result of `GetShadowCasterBounds`, but to be sure set the strength to zero and skip the light if it fails.

```
			Matrix4x4 viewMatrix, projectionMatrix;
			ShadowSplitData splitData;
			if (!cull.ComputeSpotShadowMatricesAndCullingPrimitives(
				i, out viewMatrix, out projectionMatrix, out splitData
			)) {
				shadowData[i].x = 0f;
				continue;
			}
```

When using more than one light with shadows—provided that they're positioned such that they would produce visible shadows—the frame debugger will show us that indeed we render to the shadow map more than once.



Shadow map rendered twice.

However, the resulting shadows are a mess. We still have some work to do.

5.4 Using the Correct Shadow Data

Instead of a single shadow strength, we now have to send an array of shadow data to the GPU. Make the necessary changes in `MyPipeline`.

```
	//static int shadowStrengthId = Shader.PropertyToID("_ShadowStrength");
	static int shadowDataId = Shader.PropertyToID("_ShadowData");

	…

	void RenderShadows (ScriptableRenderContext context) {
		…
		//shadowBuffer.SetGlobalFloat(
		//	shadowStrengthId, cull.visibleLights[0].light.shadowStrength
		//);
		shadowBuffer.SetGlobalVectorArray(shadowDataId, shadowData);
		…
	}
```

Likewise, instead of a single world-to-shadow matrix we'll need an array of them. Fill the array inside the loop in `RenderShadows`, then send the array to the GPU afterwards.

```
	//static int worldToShadowMatrixId =
	//	Shader.PropertyToID("_WorldToShadowMatrix");
	static int worldToShadowMatricesId =
		Shader.PropertyToID("_WorldToShadowMatrices");

	…

	void RenderShadows (ScriptableRenderContext context) {
		…

		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…
			//Matrix4x4 worldToShadowMatrix =
			//	scaleOffset * (projectionMatrix * viewMatrix);
			//shadowBuffer.SetGlobalMatrix(
			//	worldToShadowMatrixId, worldToShadowMatrix
			//);
			worldToShadowMatrices[i] =
				scaleOffset * (projectionMatrix * viewMatrix);
		}
		shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
		shadowBuffer.SetGlobalMatrixArray(
			worldToShadowMatricesId, worldToShadowMatrices
		);
		shadowBuffer.SetGlobalVectorArray(shadowDataId, shadowData);
		…
	}
```

On the shader side, adjust the shadow buffer contents to match.

```
CBUFFER_START(_ShadowBuffer)
	//float4x4 _WorldToShadowMatrix;
	//float _ShadowStrength;
	float4x4 _WorldToShadowMatrices[MAX_VISIBLE_LIGHTS];
	float4 _ShadowData[MAX_VISIBLE_LIGHTS];
	float4 _ShadowMapSize;
CBUFFER_END
```

The `ShadowAttenuation` function now needs a light index parameter so it can retrieve the correct array elements. Also, begin by checking whether the shadow strength is positive. If not, there are no shadows and immediately return an attenuation value of 1. And instead of relying on the *_SHADOWS_SOFT* keyword, branch based on the Y component of the shadow data.

```
float ShadowAttenuation (int index, float3 worldPos) {
	if (_ShadowData[index].x <= 0) {
		return 1.0;
	}
	float4 shadowPos = mul(_WorldToShadowMatrices[index], float4(worldPos, 1.0));
	shadowPos.xyz /= shadowPos.w;
	float attenuation;

	if (_ShadowData[index].y == 0) {
		attenuation =
			SAMPLE_TEXTURE2D_SHADOW(_ShadowMap, sampler_ShadowMap, shadowPos.xyz);
	}
	//#if defined(_SHADOWS_SOFT)
	else {
		real tentWeights[9];
		real2 tentUVs[9];
		SampleShadow_ComputeSamples_Tent_5x5(
			_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs
		);
		attenuation = 0;
		for (int i = 0; i < 9; i++) {
			attenuation += tentWeights[i] * SAMPLE_TEXTURE2D_SHADOW(
				_ShadowMap, sampler_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)
			);
		}
	}
	//#endif

	return lerp(1, attenuation, _ShadowData[index].x);
}
```

Finally, supply the light index when invoking `ShadowAttenuation` in `LitPassFragment`.

```
		float shadowAttenuation = ShadowAttenuation(lightIndex, input.worldPos);
```

5.5 Shadow Map Atlas

Although we're now using the correct shadow data and matrices, we still end up with incorrect results when using more than one light with shadows. That's because all shadow maps are rendered to the same texture, producing a merged result that doesn't make sense. Unity's Lightweight pipeline solves this problem by using a shadow map atlas, partitioning the render texture into square tiles, one per light with shadows. We'll use the same approach.

> **Why not use a texture array?**
>
> That is possible, but unfortunately the support for shadow-caster render-texture arrays isn't good enough yet for a universal approach. For example, it currently works fine for Metal, but OpenGL Core requires targeting shader level 4.6 and—even though it works—will cause Unity to log a continuous stream of assertion errors. So instead we'll limit ourselves to a single render texture.

As we support up to sixteen lights, we'll treat our single shadow map as if it were a 4×4 grid of tiles. The size of each tile is thus equal to the size of the shadow map divided by four. We have to constrain rendering to a viewport of that size, so create a `Rect` struct value at the start of `RenderShadows` with the appropriate size.

```
	void RenderShadows (ScriptableRenderContext context) {
		float tileSize = shadowMapSize / 4;
		Rect tileViewport = new Rect(0f, 0f, tileSize, tileSize);

		…
	}
```

Tell the GPU to use the viewport by invoking `SetViewport` on the shadow command buffer when we're also setting the view and projection matrices.

```
			shadowBuffer.SetViewport(tileViewport);
			shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
```

Using a viewport.

All shadow maps now get rendered to a single tile in the corner of our render texture. The next step is to change the offset of the viewport for each light. We can derive the viewport position from the X and Y indices of each tile. The Y offset index is found by dividing the light index by four, using an integer division. The X offset index found via an integer remainder operation instead. The final X and Y position of the viewport is equal to those indices multiplied by the tile size.

```
			float tileOffsetX = i % 4;
			float tileOffsetY = i / 4;
			tileViewport.x = tileOffsetX * tileSize;
			tileViewport.y = tileOffsetY * tileSize;

			shadowBuffer.SetViewport(tileViewport);
```

Tiles for seven lights with shadows.

A downside of using an atlas is that sampling at the edge of a tile can lead to an interpolation between data from two tiles, which is incorrect. This gets worse when using soft shadows, because the tent filter samples up to four texels away from the original sample position. It's better to fade out shadows than mix data from adjacent tiles. We can do this by adding an empty border around tiles, by instructing the GPU to limit the writing of data to a region that's a bit smaller than the viewport. This is known as scissoring, and we can do it by invoking `shadowBuffer.EnableScissorRect` with a rectangle that is a bit smaller than the viewport. We need a border of four texels, so create another rect with four added to the viewport's position and eight subtracted from its size.

```
		shadowBuffer.SetViewport(tileViewport);
		shadowBuffer.EnableScissorRect(new Rect(
			tileViewport.x + 4f, tileViewport.y + 4f,
			tileSize - 8f, tileSize - 8f
		));
```

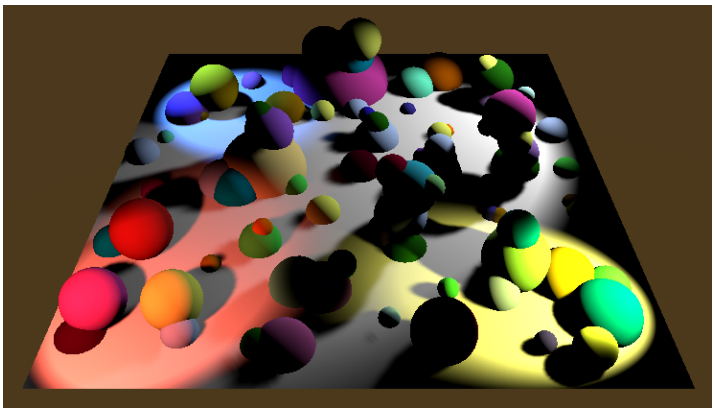
Tiles with borders.

We have to disable the scissor rectangle by invoking `DisableScissorRect` after we're done rendering shadows, otherwise regular rendering will be affected too.

```
        shadowBuffer.DisableScissorRect();
        shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);
```

The last thing that we have to do is adjust the world-to-shadow matrices so we end up sampling from the correct tile. This is done by multiplying them with a matrix that scales and offsets X and Y appropriately. The shader doesn't need to know that we're using an atlas.

```
            var scaleOffset = Matrix4x4.identity;
            scaleOffset.m00 = scaleOffset.m11 = scaleOffset.m22 = 0.5f;
            scaleOffset.m03 = scaleOffset.m13 = scaleOffset.m23 = 0.5f;
            worldToShadowMatrices[i] =
                scaleOffset * (projectionMatrix * viewMatrix);

            var tileMatrix = Matrix4x4.identity;
            tileMatrix.m00 = tileMatrix.m11 = 0.25f;
            tileMatrix.m03 = tileOffsetX * 0.25f;
            tileMatrix.m13 = tileOffsetY * 0.25f;
            worldToShadowMatrices[i] = tileMatrix * worldToShadowMatrices[i];
```



Four lights with shadows.

Keep in mind that we still only support up to four pixel lights per object. If you were to shine a fifth spotlight on a plane, one of them will turn into a vertex light, so the plane won't receive shadows from it.

6 Dynamic Tiling

An advantage of using a shadow map atlas is that we always use the same render texture, no matter how many shadow maps get rendered. So the texture memory usage is fixed. A downsize is that only part of the texture gets used per light, so we end up with shadow maps that have an effective lower resolution than we specified. We can end up with a large part of the texture not in use.

Only six out of sixteen tiles in use.

We can make better use of our texture by not always splitting it into sixteen tiles. It is possible to use a variable tile size, depending on how many tiles are needed. That way we can ensure that always at least half of the texture gets used.

6.1 Counting Shadow Tiles

First, we need to determine how many tiles are needed. We can do that by counting how many shadowed spotlights we encounter in `ConfigureLights`. Keep track of the total in a field so we can use it later.

```
	int shadowTileCount;

…

	void ConfigureLights () {
		shadowTileCount = 0;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…
			else {
				…

				if (light.lightType == LightType.Spot) {
					…
					if (
						shadowLight.shadows != LightShadows.None &&
						cull.GetShadowCasterBounds(i, out shadowBounds)
					) {
						shadowTileCount += 1;
						shadow.x = shadowLight.shadowStrength;
						shadow.y =
							shadowLight.shadows == LightShadows.Soft ? 1f : 0f;
					}
				}
			}
			…
		}
		…
	}
```

6.2 Splitting the Shadow Map

Next, we'll determine how we have to split the shadow map at the beginning of `RenderShadows`. Keep track of this with an integer variable. If we have at most a single tile, then we don't need to split at all, so the split amount is 1. Otherwise, if we have at most four tiles then the split becomes 2. The next step is up to nine tiles, with a split of 3. Only when ten or more tiles are in use do we need to use a split of 4.

```
	void RenderShadows (ScriptableRenderContext context) {
		int split;
		if (shadowTileCount <= 1) {
			split = 1;
		}
		else if (shadowTileCount <= 4) {
			split = 2;
		}
		else if (shadowTileCount <= 9) {
			split = 3;
		}
		else {
			split = 4;
		}
		…
	}
```

The tile size is now found by dividing the shadow map size by the split amount. Note that this is an integer division. In the case of a division by three, this means that we end up discarding fractional texels, which is intended. The tile scale then becomes one divided by the split amount, in this case using a float division. Then use the split amount when determining the tile offset and the tile scale when adjusting the world-to-shadow matrix.

```
		float tileSize = shadowMapSize / split;
		float tileScale = 1f / split;
		…

		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…

			float tileOffsetX = i % split;
			float tileOffsetY = i / split;
			…

			tileMatrix.m00 = tileMatrix.m11 = tileScale;
			tileMatrix.m03 = tileOffsetX * tileScale;
			tileMatrix.m13 = tileOffsetY * tileScale;
			…
		}
```

To pack all shadow maps in the available space, we must only increment the tile index when we used up a tile. So use a separate variable to keep track of it instead of relying on the light index. Increment it at the end of each iteration that we didn't skip.

```
		int tileIndex = 0;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…

			float tileOffsetX = tileIndex % split;
			float tileOffsetY = tileIndex / split;
			…
			tileIndex += 1;
		}
```

Six out of nine tiles in use.

6.3 One Tile is No Tile

Finally, if we end up with only a single tile it is not needed to set the viewport and change the scissor state at all. So only do that when there are multiple tiles.

```
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…

			if (split > 1) {
				shadowBuffer.SetViewport(tileViewport);
				shadowBuffer.EnableScissorRect(new Rect(
					tileViewport.x + 4f, tileViewport.y + 4f,
					tileSize - 8f, tileSize - 8f
				));
			}
			shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
			…

			if (split > 1) {
				var tileMatrix = Matrix4x4.identity;
				tileMatrix.m00 = tileMatrix.m11 = tileScale;
				tileMatrix.m03 = tileOffsetX * tileScale;
				tileMatrix.m13 = tileOffsetY * tileScale;
				worldToShadowMatrices[i] = tileMatrix * worldToShadowMatrices[i];
			}
			tileIndex += 1;
		}

		if (split > 1) {
			shadowBuffer.DisableScissorRect();
		}
```

6.4 Shader Keywords

We can now end up sampling shadows up to four times per fragment, which can be a combination of hard and soft shadows. In the worst case we have four soft shadows, requiring 36 shadow samples. But we have branches in our shader to only sample shadows when needed, which works well because fragments from the same object end up branching the same way. However, we can switch to less complex shader alternatives by distinguishing between different combinations of shadows.

There are four possible configurations. The first case is that there are no shadows at all. The second case is that there are only hard shadows. Third, there are only soft shadows. And the most complex case is a combination of hard and soft shadows. We can make shader variants for all cases by using the independent *_SHADOWS_HARD* and *_SHADOWS_SOFT* keywords.

In `RenderShadows`, use two boolean variables to separately keep track of whether we have hard or soft shadows, based on whether the Y component of the shadows data is set to 0 or 1. Use these booleans after the loop to toggle the shader keywords.

```
	const string shadowsHardKeyword = "_SHADOWS_HARD";
	…

	void RenderShadows (ScriptableRenderContext context) {
		…

		int tileIndex = 0;
		bool hardShadows = false;
		bool softShadows = false;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…

			if (shadowData[i].y <= 0f) {
				hardShadows = true;
			}
			else {
				softShadows = true;
			}
		}

		…
		CoreUtils.SetKeyword(shadowBuffer, shadowsHardKeyword, hardShadows);
		CoreUtils.SetKeyword(shadowBuffer, shadowsSoftKeyword, softShadows);
		…
	}
```

Add another multi-compile directive to the shader, now for *_SHADOWS_HARD*.

```
			#pragma multi_compile _ _SHADOWS_HARD
			#pragma multi_compile _ _SHADOWS_SOFT
```

In the `ShadowAttenuation` function, begin by returning 1 if neither keyword is defined. That cuts off the rest of the function, completely eliminating shadows.

```
float ShadowAttenuation (int index, float3 worldPos) {
	#if !defined(_SHADOWS_HARD) && !defined(_SHADOWS_SOFT)
		return 1.0;
	#endif
	if (_ShadowData[index].x <= 0) {
		return 1.0;
	}
	…
}
```

To make the code a more legible, move the sampling code for hard and soft shadows to their own functions.

```
float HardShadowAttenuation (float4 shadowPos) {
	return SAMPLE_TEXTURE2D_SHADOW(_ShadowMap, sampler_ShadowMap, shadowPos.xyz);
}

float SoftShadowAttenuation (float4 shadowPos) {
	real tentWeights[9];
	real2 tentUVs[9];
	SampleShadow_ComputeSamples_Tent_5x5(
		_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs
	);
	float attenuation = 0;
	for (int i = 0; i < 9; i++) {
		attenuation += tentWeights[i] * SAMPLE_TEXTURE2D_SHADOW(
			_ShadowMap, sampler_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)
		);
	}
	return attenuation;
}

float ShadowAttenuation (int index, float3 worldPos) {
	…
	float attenuation;

	if (_ShadowData[index].y == 0) {
		attenuation = HardShadowAttenuation(shadowPos);
	}
	else {
		attenuation = SoftShadowAttenuation(shadowPos);
	}

	return lerp(1, attenuation, _ShadowData[index].x);
}
```

Now we can use the keywords to create code for the other three cases. The original branch is only needed if both keywords are defined.

```
	#if defined(_SHADOWS_HARD)
		#if defined(_SHADOWS_SOFT)
			if (_ShadowData[index].y == 0) {
				attenuation = HardShadowAttenuation(shadowPos);
			}
			else {
				attenuation = SoftShadowAttenuation(shadowPos);
			}
		#else
			attenuation = HardShadowAttenuation(shadowPos);
		#endif
	#else
		attenuation = SoftShadowAttenuation(shadowPos);
	#endif
```

Finally, we can completely skip invoking `RenderShadows` in `MyPipeline.Render` if we need no shadow tiles at all. We won't even clear the shadow map. If we skip it, we do have to make sure that both shadow keywords are disabled. They can also be disabled when there are no visible lights.

```
		if (cull.visibleLights.Count > 0) {
			ConfigureLights();
			if (shadowTileCount > 0) {
				RenderShadows(context);
			}
			else {
				cameraBuffer.DisableShaderKeyword(shadowsHardKeyword);
				cameraBuffer.DisableShaderKeyword(shadowsSoftKeyword);
			}
		}
		else {
			cameraBuffer.SetGlobalVector(
				lightIndicesOffsetAndCountID, Vector4.zero
			);
			cameraBuffer.DisableShaderKeyword(shadowsHardKeyword);
			cameraBuffer.DisableShaderKeyword(shadowsSoftKeyword);
		}
```

The next tutorial is Directional Shadows.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick