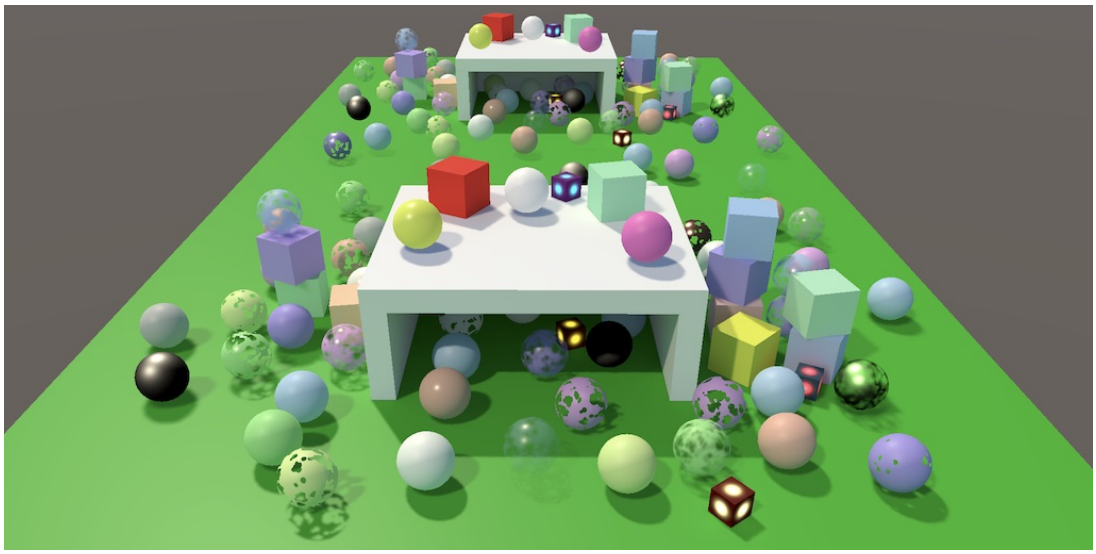
updated 2020-04-30 published 2020-02-27

Shadow Masks Baking Direct Occlusion

Bake static shadows.
Combine realtime lighting with baked shadows.
Mix realtime and baked shadows.
Support up to four shadow mask lights.

This is the sixth part of a tutorial series about creating a custom scriptable render pipeline. It uses shadow masks to bake shadows while still calculating realtime lighting.

This tutorial is made with Unity 2019.2.21f1.



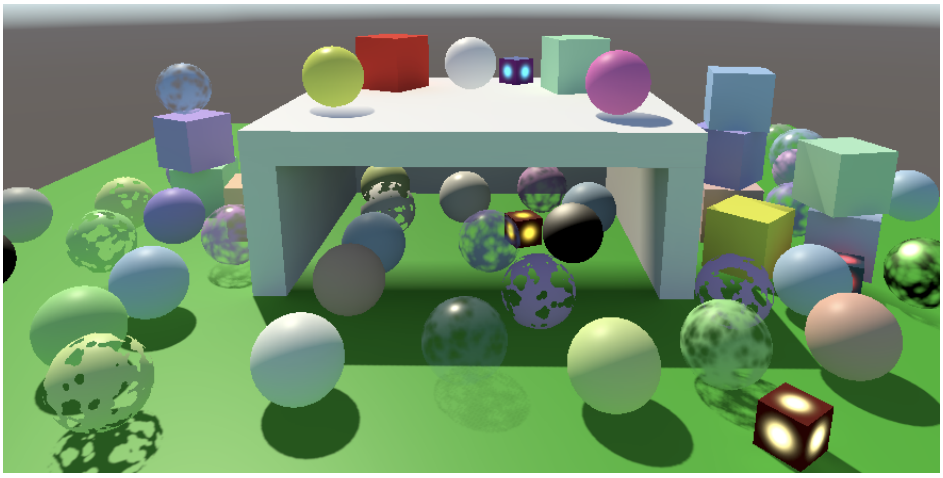
Realtime shadows nearby, baked shadows farther away.

1 Baking Shadows

The advantage of using a light map is that we're not limited to a max shadow distance. Baked shadows don't get culled, but they also cannot change. Ideally, we could use realtime shadows up to the max shadow distance and baked shadows beyond that. Unity's shadow mask mixed lighting mode makes this possible.

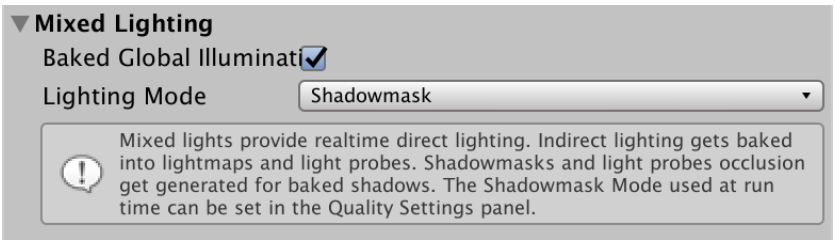
1.1 Distance Shadow Mask

Let's consider the same scene from the previous tutorial, but with the max shadow distance reduced such that part of the structure's interior doesn't get shadowed. This makes it very clear where realtime shadows end. We start with only a single light source.



Baked indirect mixed lighting, max distance 11.

Switch the mixed lighting mode to *Shadowmask*. This will invalidate the lighting data so it'll have to get baked again.

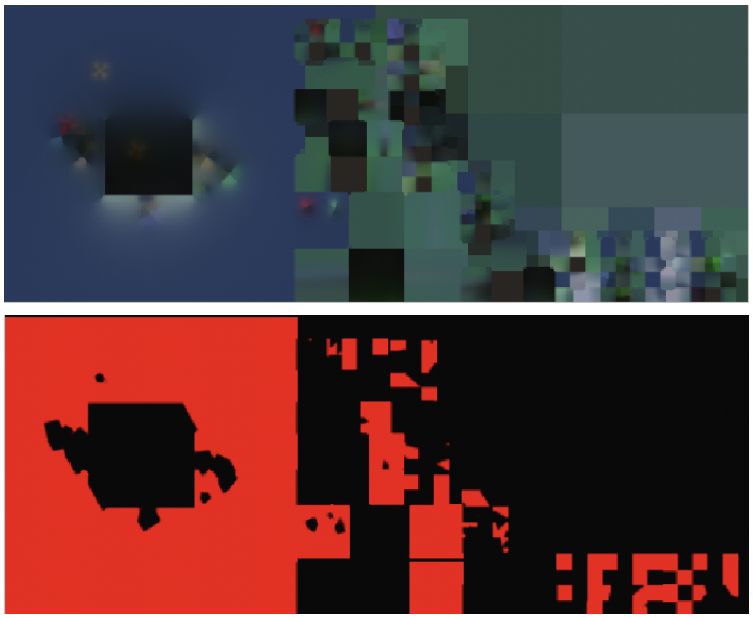


Shadowmask mixed lighting mode.

There are two ways to use shadow mask mixed lighting, which can be configured via the *Quality* project settings. We'll use the *Distance Shadowmask* mode. The other mode is known as just *Shadowmask*, which we'll cover later.

Shadow mask mode set to distance.

The two flavors of shadow mask mode use the same baked lighting data. In both cases the light map ends up containing the indirect lighting, exactly the same as the *Baked Indirect* mixed lighting mode. What's different is that there's now also a baked shadow mask map, which you can inspect via the baked light map preview window.



Baked indirect light and shadow mask.

The shadow mask map contains the shadow attenuation of our single mixed directional light, representing the shadows cast by all static objects that contribute to global illumination. The data is stored in the red channel so the map is black and red.

Just like the baked indirect lighting the baked shadows cannot change at runtime. However, the shadows will remain valid no matter the intensity or color of the light. But the light should not be rotated otherwise its shadows won't make sense. Also, you shoudn't vary the light too much if its indirect lighting is baked. For example, it would be obviously wrong if indirect lighting remained after a light is turned off. If a light changes a lot then you could set its *Indirect Multiplier* to zero so no indirect light gets baked for it.

1.2 Detecting a Shadow Mask

To use the shadow mask our pipeline must first know that it exists. As it's all about shadows that's the job of our `Shadows` class. We'll use shader keywords to control whether shadow masks are used. As there are two modes we'll introduce another static keyword array, even though it contains just one keyword for now: *_SHADOW_MASK_DISTANCE*.

```
	static string[] shadowMaskKeywords = {
		"_SHADOW_MASK_DISTANCE"
	};
```

Add a boolean field to track whether we're using a shadow mask. We re-evaluate this each frame so initialize it to `false` in `Setup`.

```
	bool useShadowMask;

	public void Setup (…) {
		…
		useShadowMask = false;
	}
```

Enable or distable the keyword at the end of `Render`. We have to do this even if we end up not rendering any realtime shadows, because the shadow mask isn't realtime.

```
	public void Render () {
		…
		buffer.BeginSample(bufferName);
		SetKeywords(shadowMaskKeywords, useShadowMask ? 0 : -1);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}
```

To know whether a shadow mask is needed we have to check if there is a light that uses it. We'll do this in `ReserveDirectionalShadows`, when we end up with a valid shadow-casting light.

Each light contains information about its baked data. It's stored in a `LightBakingOutput` struct that can be retrieved via the `Light.bakingOutput` property. If we encounter a light with its light map bake type set to mixed and its mixed lighting mode set to shadow mask then we're using the shadow mask.

```
	public Vector3 ReserveDirectionalShadows (
		Light light, int visibleLightIndex
	) {
		if (…) {
			LightBakingOutput lightBaking = light.bakingOutput;
			if (
				lightBaking.lightmapBakeType == LightmapBakeType.Mixed &&
				lightBaking.mixedLightingMode == MixedLightingMode.Shadowmask
			) {
				useShadowMask = true;
			}

			…
		}
		return Vector3.zero;
	}
```

That enables the shader keyword when needed. Add a corresponding multi-compile directive for it to the *CustomLit* pass of the *Lit* shader.

```
			#pragma multi_compile _ _CASCADE_BLEND_SOFT _CASCADE_BLEND_DITHER
			#pragma multi_compile _ _SHADOW_MASK_DISTANCE
			#pragma multi_compile _ LIGHTMAP_ON
```

1.3 Shadow Mask Data

On the shader side we have to know whether a shadow mask is in use and if so what the baked shadows are. Let's add a `ShadowMask` struct to *Shadows* to keep track of both, with a boolean and a float vector field. Name the boolean `distance` to indicate whether distance shadow mask mode is enabled. Then add this struct to the global `ShadowData` struct as a field.

```
struct ShadowMask {
	bool distance;
	float4 shadows;
};

struct ShadowData {
	int cascadeIndex;
	float cascadeBlend;
	float strength;
	ShadowMask shadowMask;
};
```

Initialize the shadow mask to not-in-use by default in `GetShadowData`.

```
ShadowData GetShadowData (Surface surfaceWS) {
	ShadowData data;
	data.shadowMask.distance = false;
	data.shadowMask.shadows = 1.0;
	…
}
```

Although the shadow mask is used for shadowing it is part of the baked lighting data of the scene. As such retrieving it is the responsibility of *GI*. So add a shadow mask field to the `GI` struct as well and also initialize it to not-in-use in `GetGI`.

```
struct GI {
	float3 diffuse;
	ShadowMask shadowMask;
};

…

GI GetGI (float2 lightMapUV, Surface surfaceWS) {
	GI gi;
	gi.diffuse = SampleLightMap(lightMapUV) + SampleLightProbe(surfaceWS);
	gi.shadowMask.distance = false;
	gi.shadowMask.shadows = 1.0;
	return gi;
}
```

Unity makes the shadow mask map available to the shader via a `unity_ShadowMask` texture and accompanying sampler state. Define those in *GI* along with the other light map texture and sampler state.

```
TEXTURE2D(unity_Lightmap);
SAMPLER(samplerunity_Lightmap);

TEXTURE2D(unity_ShadowMask);
SAMPLER(samplerunity_ShadowMask);
```

Then add a `SampleBakedShadows` function that samples the map, using the light map UV coordinates. Just like for the regular light map this only makes sense for lightmapped geometry, so when *LIGHTMAP_ON* is defined. Otherwise there are no baked shadows and the attenuation is always 1.

```
float4 SampleBakedShadows (float2 lightMapUV) {
	#if defined(LIGHTMAP_ON)
		return SAMPLE_TEXTURE2D(
			unity_ShadowMask, samplerunity_ShadowMask, lightMapUV
		);
	#else
		return 1.0;
	#endif
}
```

Now we can adjust `GetGI` so it enables the distance shadow mask mode and samples the baked shadows if *_SHADOW_MASK_DISTANCE* is defined. Note that this makes the `distance` boolean a compile-time constant so its usage won't result in dynamic branching.

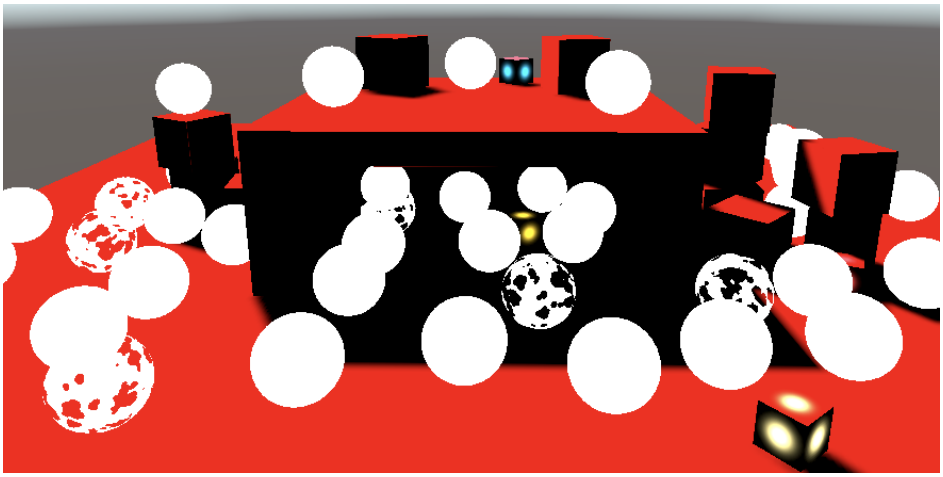
```
GI GetGI (float2 lightMapUV, Surface surfaceWS) {
	GI gi;
	gi.diffuse = SampleLightMap(lightMapUV) + SampleLightProbe(surfaceWS);
	gi.shadowMask.distance = false;
	gi.shadowMask.shadows = 1.0;

	#if defined(_SHADOW_MASK_DISTANCE)
		gi.shadowMask.distance = true;
		gi.shadowMask.shadows = SampleBakedShadows(lightMapUV);
	#endif
	return gi;
}
```

It's up to *Lighting* to copy the shadow mask data from `GI` to `ShadowData`, in `GetLighting` before looping through the lights. At this point we can also debug the shadow mask data by directly returning it as the final lighting color.

```
float3 GetLighting (Surface surfaceWS, BRDF brdf, GI gi) {
	ShadowData shadowData = GetShadowData(surfaceWS);
	shadowData.shadowMask = gi.shadowMask;
	return gi.shadowMask.shadows.rgb;

	…
}
```

Initially it doesn't appear to work, as everything ends up white. We have to instruct Unity to send the relevant data to the GPU, just like we did in the previous tutorial for the light map and probes in `CameraRenderer.DrawVisibleGeometry`. In this case we have to add `PerObjectData.ShadowMask` to the per-object data.

```
			perObjectData =
				PerObjectData.Lightmaps | PerObjectData.ShadowMask |
				PerObjectData.LightProbe |
				PerObjectData.LightProbeProxyVolume
```



Sampling shadow mask.

Why does Unity bake lighting each time we change shader code?

That happens when we change HLSL files that are included by a meta pass. You can prevent needless baking by temporarily disabling *Auto Generate*.

1.4 Occlusion Probes

We can see that the shadow mask gets applied to lightmapped objects correctly. We also see that dynamic objects have no shadow mask data, as expected. They use light probes instead of light maps. However, Unity also bakes shadow mask data into light probes, referring to it as occlusion probes. We can access this data by adding a `unity_ProbesOcclusion` vector to the `UnityPerDraw` buffer in *UnityInput*. Place it in between the world transform parameters and light map UV transformation vector.

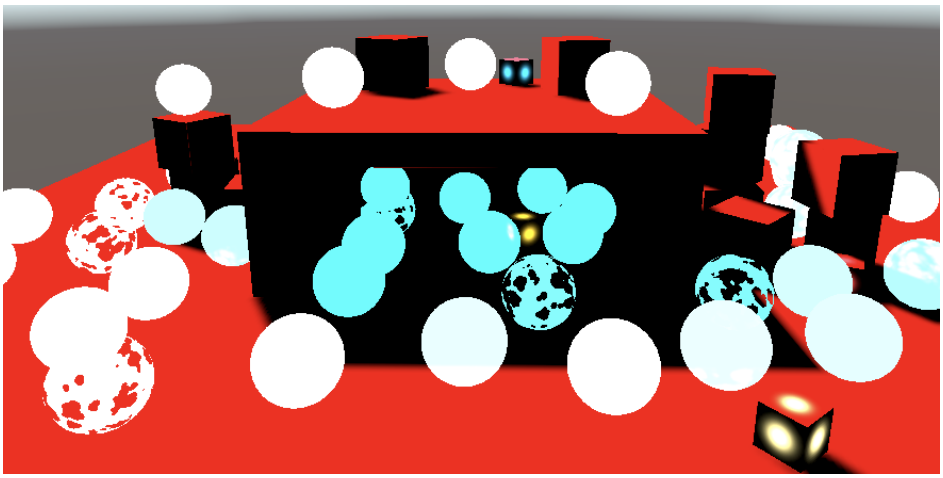
```
	real4 unity_WorldTransformParams;

	float4 unity_ProbesOcclusion;

	float4 unity_LightmapST;
```

Now we can simply return that vector in `SampleBakedShadows` for dynamic objects.

```
float4 SampleBakedShadows (float2 lightMapUV) {
	#if defined(LIGHTMAP_ON)
		…
	#else
		return unity_ProbesOcclusion;
	#endif
}
```

Again, we have to instruct Unity to send this data to the GPU, this time by enabling the `PerObjectData.OcclusionProbe` flag.

```
			perObjectData =
				PerObjectData.Lightmaps | PerObjectData.ShadowMask |
				PerObjectData.LightProbe | PerObjectData.OcclusionProbe |
				PerObjectData.LightProbeProxyVolume
```



Sampling occlusion probes.

Unused channels of the shadow mask are set to white for probes, so dynamic objects end up white when fully lit and cyan when fully shadowed, instead of red and black.

Although this is enough to get shadow masks working via probes, it breaks GPU instancing. The occlusion data can get instanced automatically, but *UnityInstancing* only does this when *SHADOWS_SHADOWMASK* is defined. So define it when needed in *Common* before including *UnityInstancing*. This is the only other place where we have to explicitly check whether *_SHADOW_MASK_DISTANCE* is defined.

```
#if defined(_SHADOW_MASK_DISTANCE)
	#define SHADOWS_SHADOWMASK
#endif

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

1.5 LPPVs

Light probe proxy volumes can also work with shadow masks. Again we have to enable this by setting a flag, this time `PerObjectData.OcclusionProbeProxyVolume`.

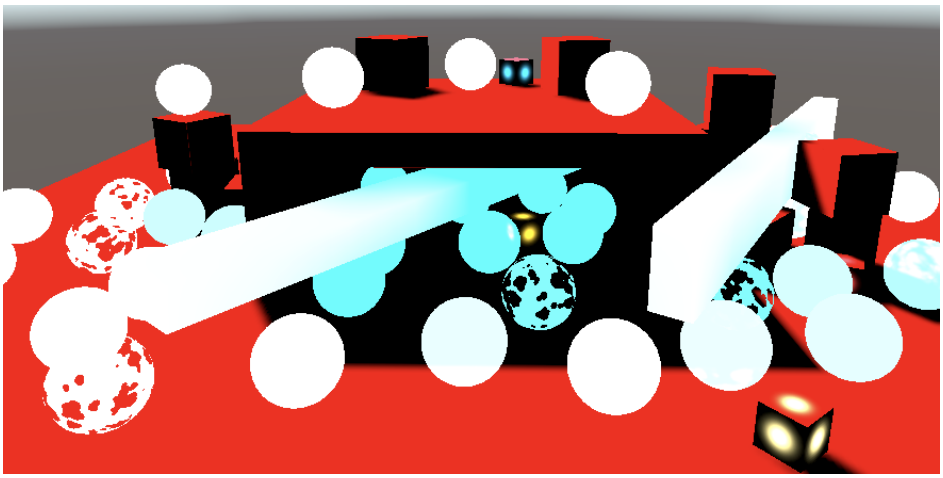
```
			perObjectData =
				PerObjectData.Lightmaps | PerObjectData.ShadowMask |
				PerObjectData.LightProbe | PerObjectData.OcclusionProbe |
				PerObjectData.LightProbeProxyVolume |
				PerObjectData.OcclusionProbeProxyVolume
```

Retrieving the LPPV occlusion data works the same as retrieving its light data, except that we have to invoke `SampleProbeOcclusion` instead of `SampleProbeVolumeSH4`. It's stored in the same texture and requires the same arguments, with the sole exception that a normal vector isn't needed. Add a branch for this to `SampleBakedShadows`, along with a surface parameter for the now required world position.

```
float4 SampleBakedShadows (float2 lightMapUV, Surface surfaceWS) {
	#if defined(LIGHTMAP_ON)
		…
	#else
		if (unity_ProbeVolumeParams.x) {
			return SampleProbeOcclusion(
				TEXTURE3D_ARGS(unity_ProbeVolumeSH, samplerunity_ProbeVolumeSH),
				surfaceWS.position, unity_ProbeVolumeWorldToObject,
				unity_ProbeVolumeParams.y, unity_ProbeVolumeParams.z,
				unity_ProbeVolumeMin.xyz, unity_ProbeVolumeSizeInv.xyz
			);
		}
		else {
			return unity_ProbesOcclusion;
		}
	#endif
}
```

Add the new surface argument when invoking the function in `GetGI`.

```
		gi.shadowMask.shadows = SampleBakedShadows(lightMapUV, surfaceWS);
```



Sampling LPPV occlusion.

1.6 Mesh Ball

If our mesh ball uses an LPPV it already supports shadow masks. But when it interpolates light probes itself we have to add the occlusion probe data in `MeshBall.Update`. That's done by using a temporary `Vector4` array for the last argument of `CalculateInterpolatedLightAndOcclusionProbes` and passing it to the property block via the `CopyProbeOcclusionArrayFrom` method.

```
		var lightProbes = new SphericalHarmonicsL2[1023];
		var occlusionProbes = new Vector4[1023];
		LightProbes.CalculateInterpolatedLightAndOcclusionProbes(
			positions, lightProbes, occlusionProbes
		);
		block.CopySHCoefficientArraysFrom(lightProbes);
		block.CopyProbeOcclusionArrayFrom(occlusionProbes);
```

After verifying that the shadow mask data is correctly sent to the shader we can remove its debug visualization from `GetLighting`.

```
	//return gi.shadowMask.shadows.rgb;
```

2 Mixing Shadows

Now that we have the shadow mask available the next step is to use it when realtime shadows aren't, which is the case when a fragment ends up beyond the max shadow distance.

2.1 Use Baked when Available

Mixing baked and realtime shadows will make the work of `GetDirectionalShadowAttenuation` more complicated. Let's begin by isolating all realtime shadow sampling code, moving it to a new `GetCascadedShadow` function in *Shadows*.

```
float GetCascadedShadow (
	DirectionalShadowData directional, ShadowData global, Surface surfaceWS
) {
	float3 normalBias = surfaceWS.normal *
		(directional.normalBias * _CascadeData[global.cascadeIndex].y);
	float3 positionSTS = mul(
		_DirectionalShadowMatrices[directional.tileIndex],
		float4(surfaceWS.position + normalBias, 1.0)
	).xyz;
	float shadow = FilterDirectionalShadow(positionSTS);
	if (global.cascadeBlend < 1.0) {
		normalBias = surfaceWS.normal *
			(directional.normalBias * _CascadeData[global.cascadeIndex + 1].y);
		positionSTS = mul(
			_DirectionalShadowMatrices[directional.tileIndex + 1],
			float4(surfaceWS.position + normalBias, 1.0)
		).xyz;
		shadow = lerp(
			FilterDirectionalShadow(positionSTS), shadow, global.cascadeBlend
		);
	}
	return shadow;
}

float GetDirectionalShadowAttenuation (
	DirectionalShadowData directional, ShadowData global, Surface surfaceWS
) {
	#if !defined(_RECEIVE_SHADOWS)
		return 1.0;
	#endif

	float shadow;
	if (directional.strength <= 0.0) {
		shadow = 1.0;
	}
	else {
		shadow = GetCascadedShadow(directional, global, surfaceWS);
		shadow = lerp(1.0, shadow, directional.strength);
	}
	return shadow;
}
```

Then add a new `GetBakedShadow` function that returns the baked shadow attenuation for a given shadow mask. If the mask's distance mode is enabled then we need the first component of its shadows vector, otherwise there is no attenuation available and the result is 1.

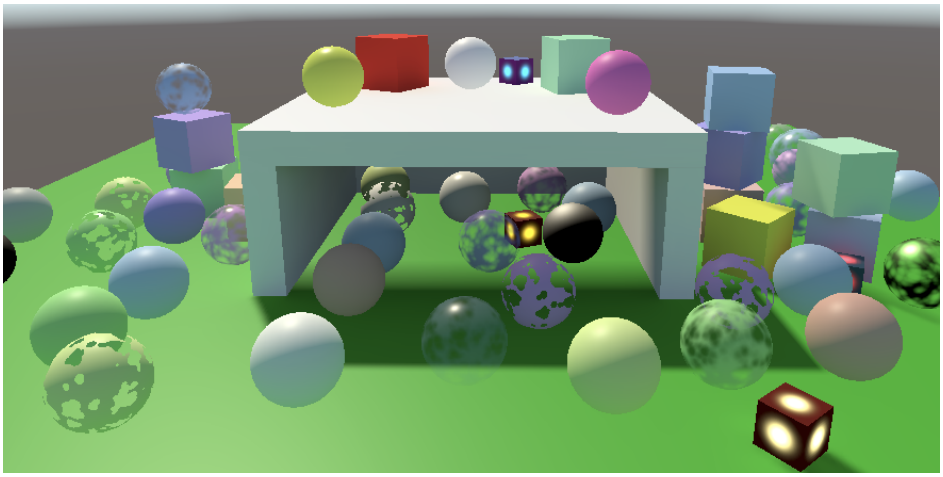
```
float GetBakedShadow (ShadowMask mask) {
	float shadow = 1.0;
	if (mask.distance) {
		shadow = mask.shadows.r;
	}
	return shadow;
}
```

Next, create a `MixBakedAndRealtimeShadows` function with a `ShadowData`, realtime shadow, and shadow strength parameter. It simply applies the strength to the shadow, except when there is a distance shadow mask. If so, replace the realtime shadow with the baked one.

```
float MixBakedAndRealtimeShadows (
	ShadowData global, float shadow, float strength
) {
	float baked = GetBakedShadow(global.shadowMask);
	if (global.shadowMask.distance) {
		shadow = baked;
	}
	return lerp(1.0, shadow, strength);
}
```

Have `GetDirectionalShadowAttenuation` use that function instead of applying the strength itself.

```
		shadow = GetCascadedShadow(directional, global, surfaceWS);
		shadow = MixBakedAndRealtimeShadows(global, shadow, directional.strength);
```



Faded baked shadows.

The result is that we now always use the shadow mask, so we can see that it works. However, the baked shadows fade with distance exactly like the realtime shadows.

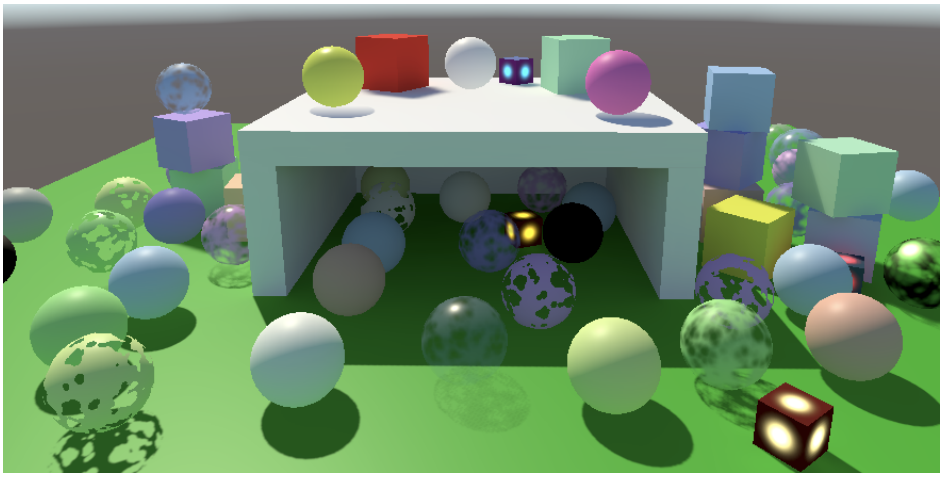
2.2 Transitioning to Baked

To transition from realtime to baked shadows based on depth we have to interpolate between them based on the global shadow strength. However, we also have to apply the light's shadow strength, which we have to do after the interpolation. So we can no longer immediately combine both strengths in `GetDirectionalShadowData`.

```
	data.strength =
		_DirectionalLightShadowData[lightIndex].x; // * shadowData.strength;
```

In `MixBakedAndRealtimeShadows` perform the interpolation between baked and realtime based on the global strength and after that apply the light's shadow strength. But when there isn't a shadow mask apply the combined strengths to the realtime shadow only, as we did before.

```
float MixBakedAndRealtimeShadows (
	ShadowData global, float shadow, float strength
) {
	float baked = GetBakedShadow(global.shadowMask);
	if (global.shadowMask.distance) {
		shadow = lerp(baked, shadow, global.strength);
		return lerp(1.0, shadow, strength);
	}
	return lerp(1.0, shadow, strength * global.strength);
}
```

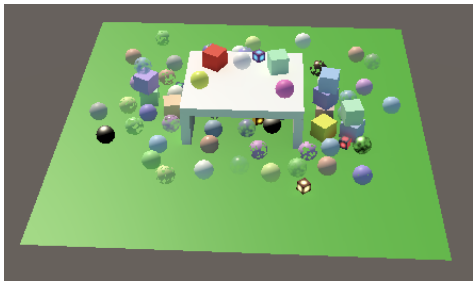


Mixed shadows.

The result is that shadows cast by dynamic objects fade as usual, while shadows cast by static objects transition to the shadow mask.

2.3 Only Baked Shadows

Currently our approach only works when there are realtime shadows to render. If there aren't then the shadow mask disappears as well. This can be verified by zooming out the scene view until everything lies beyond the max shadow distance.



Neither realtime nor baked shadows.

We have to support the case when there is a shadow mask but no realtime shadows. Let's begin by creating a `GetBakedShadow` function variant that also has a strength parameter, so we can conveniently get a strength-modulated baked shadow.

```
float GetBakedShadow (ShadowMask mask, float strength) {
	if (mask.distance) {
		return lerp(1.0, GetBakedShadow(mask), strength);
	}
	return 1.0;
}
```

Next, in `GetDirectionalShadowAttenuation` check whether the combined strengths end up as zero or less. If so, rather than always returning 1 return the modulated baked shadow only, still skipping realtime shadow sampling.

```
	if (directional.strength * global.strength <= 0.0) {
		shadow = GetBakedShadow(global.shadowMask, directional.strength);
	}
```

Besides that, we have to change `Shadows.ReserveDirectionalShadows` so it doesn't immediately skip lights that end up with no realtime shadow casters. Instead, first determine whether the light uses the shadow mask. After that check whether there aren't realtime shadow casters, in which case only the shadow strength is relevant.

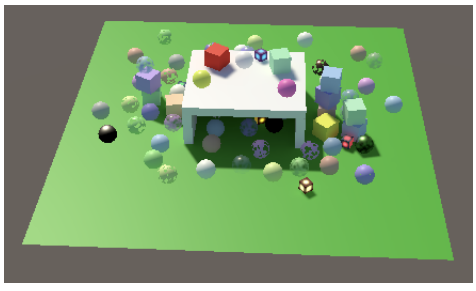
```
		if (
			shadowedDirLightCount < maxShadowedDirLightCount &&
			light.shadows != LightShadows.None && light.shadowStrength > 0f //&&
			//cullingResults.GetShadowCasterBounds(visibleLightIndex, out Bounds b)
		) {
			LightBakingOutput lightBaking = light.bakingOutput;
			if (
				lightBaking.lightmapBakeType == LightmapBakeType.Mixed &&
				lightBaking.mixedLightingMode == MixedLightingMode.Shadowmask
			) {
				useShadowMask = true;
			}

			if (!cullingResults.GetShadowCasterBounds(
				visibleLightIndex, out Bounds b
			)) {
				return new Vector3(light.shadowStrength, 0f, 0f);
			}

			…
		}
```

But when the shadow strength is greater than zero the shader will sample the shadow map, even though that would be incorrect. We can make this work by negating the shadow strength in this case.

```
				return new Vector3(-light.shadowStrength, 0f, 0f);
```

Then pass the absolute strength to `GetBakedShadow` in `GetDirectionalShadowAttenuation` when we skip realtime shadows. That way it works both when there aren't realtime shadow casters and when we're beyond the max shadow distance.

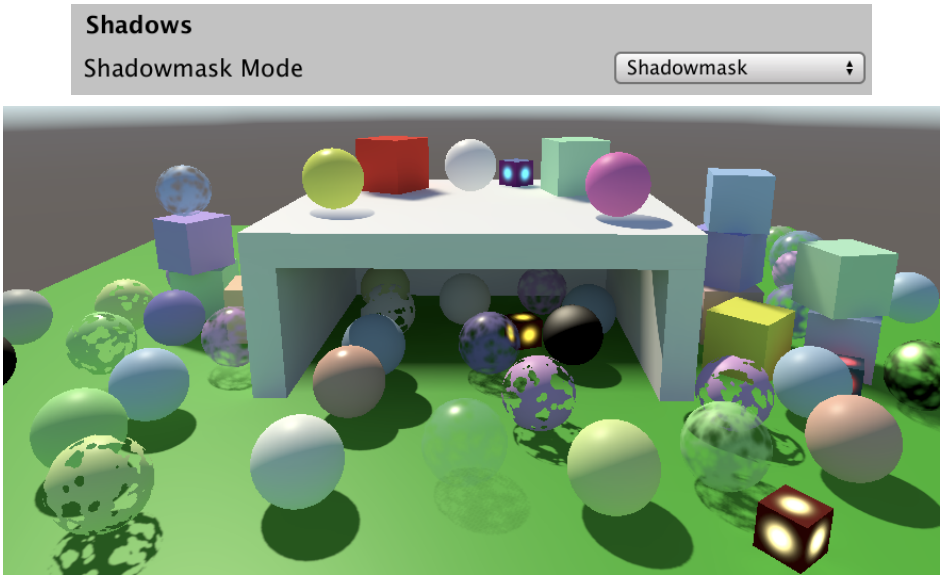
```
		shadow = GetBakedShadow(global.shadowMask, abs(directional.strength));
```



Only baked shadows.

2.4 Always use the Shadow Mask

There is another shadow mask mode, which is simply known as *Shadowmask*. It works exactly the same as the distance mode, except that Unity will omit static shadow casters for lights that use the shadow mask.



No realtime shadows cast by static geometry.

The idea is that because the shadow mask is available everywhere we could use it for static shadows everywhere as well. That means less realtime shadows, which makes rendering faster, at the cost of lower-quality static shadows up close.

To support this mode, add a *_SHADOW_MASK_ALWAYS* keyword as the first element of the shadow mask keyword array in `Shadows`. We can determe which should be enabled in `Render` by checking the `QualitySettings.shadowmaskMode` property.

```
	static string[] shadowMaskKeywords = {
		"_SHADOW_MASK_ALWAYS",
		"_SHADOW_MASK_DISTANCE"
	};

	…

	public void Render () {
		…
		buffer.BeginSample(bufferName);
		SetKeywords(shadowMaskKeywords, useShadowMask ?
			QualitySettings.shadowmaskMode == ShadowmaskMode.Shadowmask ? 0 : 1 :
			-1
		);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}
```

Add the keyword to the multi-compile directive in our shader.

```
			#pragma multi_compile _ _SHADOW_MASK_ALWAYS _SHADOW_MASK_DISTANCE
```

And also check for it in *Common* when deciding to define *SHADOWS_SHADOWMASK*.

```
#if defined(_SHADOW_MASK_ALWAYS) || defined(_SHADOW_MASK_DISTANCE)
	#define SHADOWS_SHADOWMASK
#endif
```

Give the `ShadowMask` struct a separate boolean field to indicate whether the shadow mask should always be used.

```
struct ShadowMask {
	bool always;
	bool distance;
	float4 shadows;
};

…

ShadowData GetShadowData (Surface surfaceWS) {
	ShadowData data;
	data.shadowMask.always = false;
	…
}
```

Then set it when appropriate in `GetGI`, along with its shadow data.

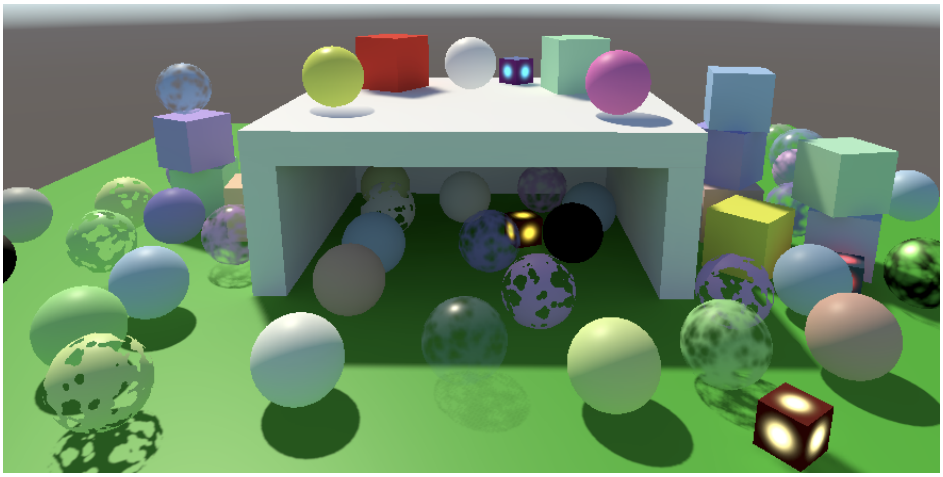
```
GI GetGI (float2 lightMapUV, Surface surfaceWS) {
	GI gi;
	gi.diffuse = SampleLightMap(lightMapUV) + SampleLightProbe(surfaceWS);
	gi.shadowMask.always = false;
	gi.shadowMask.distance = false;
	gi.shadowMask.shadows = 1.0;

	#if defined(_SHADOW_MASK_ALWAYS)
		gi.shadowMask.always = true;
		gi.shadowMask.shadows = SampleBakedShadows(lightMapUV, surfaceWS);
	#elif defined(_SHADOW_MASK_DISTANCE)
		gi.shadowMask.distance = true;
		gi.shadowMask.shadows = SampleBakedShadows(lightMapUV, surfaceWS);
	#endif
	return gi;
}
```

Both versions of `GetBakedShadow` should select the mask when either mode is in use.

```
float GetBakedShadow (ShadowMask mask) {
	float shadow = 1.0;
	if (mask.always || mask.distance) {
		shadow = mask.shadows.r;
	}
	return shadow;
}

float GetBakedShadow (ShadowMask mask, float strength) {
	if (mask.always || mask.distance) {
		return lerp(1.0, GetBakedShadow(mask), strength);
	}
	return 1.0;
}
```

Finally, `MixBakedAndRealtimeShadows` must now use a different approach when the shadow mask is always active. First, the realtime shadow must be modulated by the global strength to fade it based on depth. Then the baked and realtime shadows are combined, by taking their minimum. After that the light's shadow strength is applied to the merged shadows.

```
float MixBakedAndRealtimeShadows (
	ShadowData global, float shadow, float strength
) {
	float baked = GetBakedShadow(global.shadowMask);
	if (global.shadowMask.always) {
		shadow = lerp(1.0, shadow, global.strength);
		shadow = min(baked, shadow);
		return lerp(1.0, shadow, strength);
	}
	if (global.shadowMask.distance) {
		shadow = lerp(baked, shadow, global.strength);
		return lerp(1.0, shadow, strength);
	}
	return lerp(1.0, shadow, strength * global.strength);
}
```

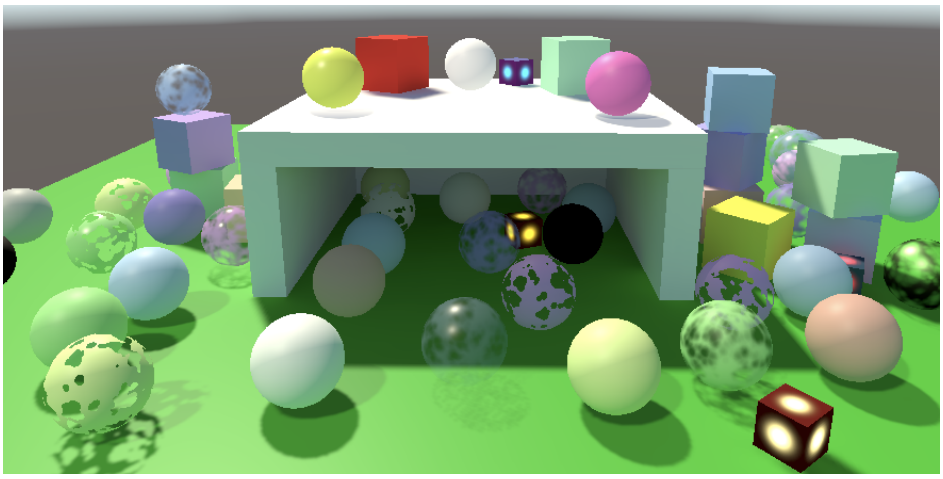


Baked static shadows mixed with realtime dynamic shadows.

3 Multiple Lights

Because the shadow mask map has four channels it can support up to four mixed lights. The most important light while baking gets the red channel, the second light gets the green channel, and so on. Let's try this out by duplicating our single directional light, rotating it a bit, and reducing its intensity so the new light ends up using the green channel.

> **What happens when there are more than four mixed-mode lights?**
>
> Unity will convert all mixed-mode lights beyond first four to fully-baked lights. That's assuming that all lights are directional, which is the only light type that we currently support. Other light types have a limited area of influence, which could make it possible to use the same channel for more that one light.

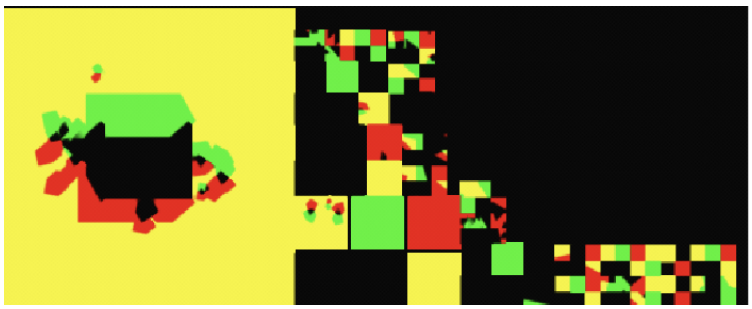


Two lights sharing the same baked shadows.

The realtime shadows of the second light work as expected, but it ends up using the mask of the first light for baked shadows, which is clearly wrong. This is easiest to see when using the always-shadow-mask mode.

3.1 Shadow Mask Channels

Inspecting the baked shadow mask map reveals that the shadows are baked correctly. Areas lit by only the first light are red, areas lit by only the second light are green, and areas lit by both are yellow. This works for up to four lights, although the fourth wouldn't be visible in the preview because the alpha channel isn't shown.



Baked shadows for two lights.

Both lights use the same baked shadows because we always use the red channel. To make this work we have to send the light's channel index to the GPU. We cannot rely on the light order because it can vary at runtime, as lights can be changed and even disabled.

We can retrieve a light's mask channel index in `Shadows.ReserveDirectionalShadows` via the `LightBakingOutput.occlusionMaskChannel` field. As we're sending a 4D vectors to the GPU we can store it in the fourth channel of the vector that we return, changing the return type to `Vector4`. And when the light doesn't use a shadow mask we indicate that by setting its index to -1.

```
	public Vector4 ReserveDirectionalShadows (
		Light light, int visibleLightIndex
	) {
		if (
			shadowedDirLightCount < maxShadowedDirLightCount &&
			light.shadows != LightShadows.None && light.shadowStrength > 0f
		) {
			float maskChannel = -1;
			LightBakingOutput lightBaking = light.bakingOutput;
			if (
				lightBaking.lightmapBakeType == LightmapBakeType.Mixed &&
				lightBaking.mixedLightingMode == MixedLightingMode.Shadowmask
			) {
				useShadowMask = true;
				maskChannel = lightBaking.occlusionMaskChannel;
			}

			if (!cullingResults.GetShadowCasterBounds(
				visibleLightIndex, out Bounds b
			)) {
				return new Vector4(-light.shadowStrength, 0f, 0f, maskChannel);
			}

			shadowedDirectionalLights[shadowedDirLightCount] =
				new ShadowedDirectionalLight {
					visibleLightIndex = visibleLightIndex,
					slopeScaleBias = light.shadowBias,
					nearPlaneOffset = light.shadowNearPlane
				};
			return new Vector4(
				light.shadowStrength,
				settings.directional.cascadeCount * shadowedDirLightCount++,
				light.shadowNormalBias, maskChannel
			);
		}
		return new Vector4(0f, 0f, 0f, -1f);
	}
```

3.2 Selecting the Appropriate Channel

On the shader size, add the shadow mask channel as an additional integer field to the `DirectionalShadowData` struct defined in *Shadows*.

```
struct DirectionalShadowData {
	float strength;
	int tileIndex;
	float normalBias;
	int shadowMaskChannel;
};
```

GI then has to set the channel, in `GetDirectionalShadowData`.

```
DirectionalShadowData GetDirectionalShadowData (
	int lightIndex, ShadowData shadowData
) {
	…
	data.shadowMaskChannel = _DirectionalLightShadowData[lightIndex].w;
	return data;
}
```

Add a channel parameter to both versions of `GetBakedShadow` and use it to return the appropriate shadow mask data. But only do this if the light uses the shadow mask, so when the channel is at least zero.

```
float GetBakedShadow (ShadowMask mask, int channel) {
	float shadow = 1.0;
	if (mask.always || mask.distance) {
		if (channel >= 0) {
			shadow = mask.shadows[channel];
		}
	}
	return shadow;
}

float GetBakedShadow (ShadowMask mask, int channel, float strength) {
	if (mask.always || mask.distance) {
		return lerp(1.0, GetBakedShadow(mask, channel), strength);
	}
	return 1.0;
}
```

Isn't a dot product better than indexing a channel?

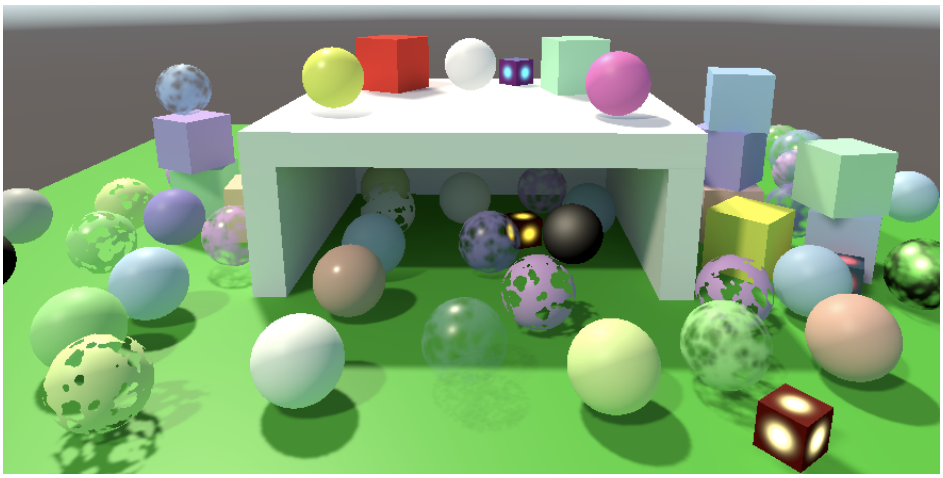
Yes, but the shader compiler will take care of that for us. It will use the channel to index a static buffer of vectors with the appropriate components set to 1, which it will then use to perform a dot product with the mask to filter it. We could also send the dot products to the GPU to skip the lookup step, but that would require sending an additional vector array which would have to be indexed anyway.

Adjust `MixBakedAndRealtimeShadows` so it passes along the required shadow mask channel.

```
float MixBakedAndRealtimeShadows (
	ShadowData global, float shadow, int shadowMaskChannel, float strength
) {
	float baked = GetBakedShadow(global.shadowMask, shadowMaskChannel);
	…
}
```

Finally, add the needed channel arguments in `GetDirectionalShadowAttenuation`.

```
float GetDirectionalShadowAttenuation (
	DirectionalShadowData directional, ShadowData global, Surface surfaceWS
) {
	#if !defined(_RECEIVE_SHADOWS)
		return 1.0;
	#endif

	float shadow;
	if (directional.strength * global.strength <= 0.0) {
		shadow = GetBakedShadow(
			global.shadowMask, directional.shadowMaskChannel,
			abs(directional.strength)
		);
	}
	else {
		shadow = GetCascadedShadow(directional, global, surfaceWS);
		shadow = MixBakedAndRealtimeShadows(
			global, shadow, directional.shadowMaskChannel, directional.strength
		);
	}
	return shadow;
}
```



Both lights using their own channel.

What about the *Subtractive* mixed lighting mode?

Subtractive lighting is an alternative way to combine baked lighting and shadows, using only a single light map. The idea is that you fully bake a light but also use it for realtime lighting. You then calculate realtime diffuse lighting for that light, sample realtime shadows, and use that to determine how much diffuse light was shadowed, which you subtract from the diffuse GI.

So you end up with static objects that use baked lighting—even though diffuse realtime lighting is calculated for them—that can receive realtime shadows. Dynamic objects have to rely on occlusion probes to receive static shadows.

It is a budget approach that is severely limited. It only works for a single directional light that cannot change. All indirect lighting or any other baked light produces incorrect results, which is mitigated by constraining the darkening via a configurable shadow color, which should match the average indirect GI color of the scene.

I won't include support for the subtractive mode in this series. If you have space for a shadow mask map then always-shadow-mask mode is superior to subtractive. If not then consider going fully-baked, which allows a more complex lighting setup.

The next tutorial is LOD and Reflections.

made by Jasper Flick