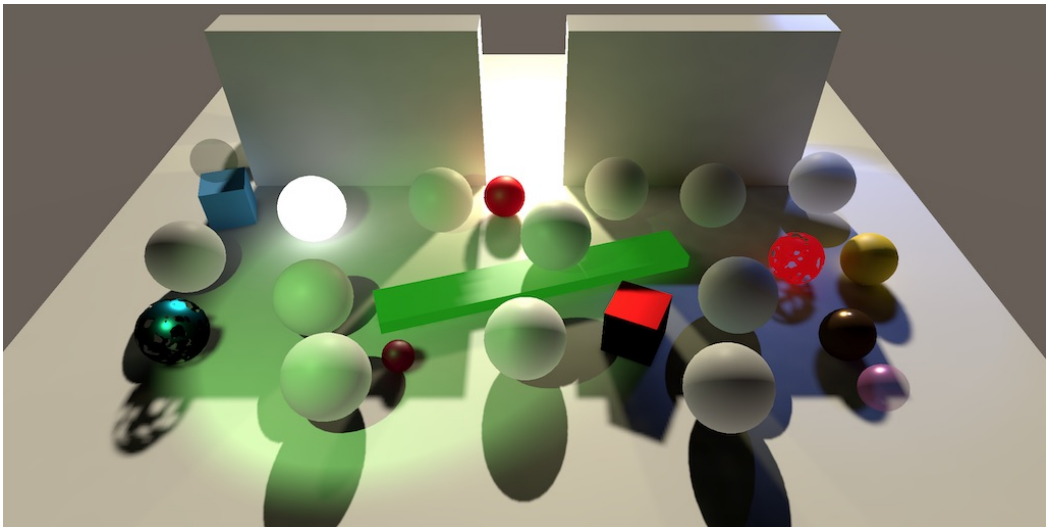
Baked Shadows Mixed Lighting

Fade realtime shadows.
Apply a shadowmask and shadow probes.
Use up to four baked shadows per object.
Choose between normal and distance shadowmask mode.
Support subtractive lighting.

This is the ninth installment of a tutorial series covering Unity's scriptable render pipeline. It's about combining realtime lighting with baked shadows, and baked lighting with realtime shadows in the case of subtractive lighting.

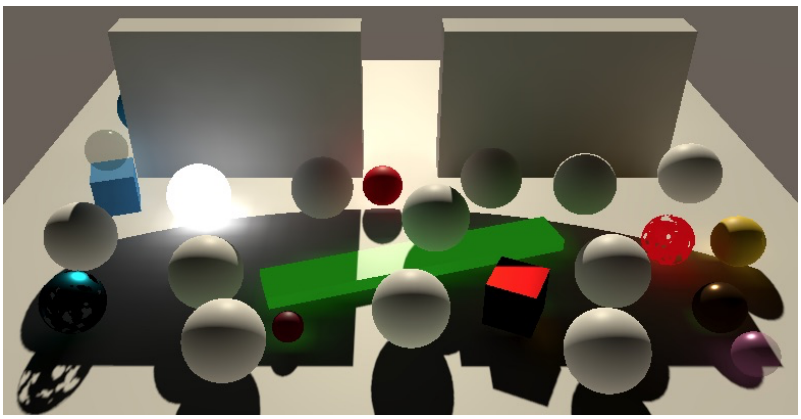
This tutorial is made with Unity 2018.3.0f2.



Baked and realtime shadows working together.

1 Shadow Fading

Realtime lighting with shadows is expensive to render. Baked lighting is much cheaper, but doesn't contain specular reflections and cannot change at runtime. Unity supports a third approach, which combines realtime lighting with baked shadows. But some realtime shadows are used still, so both types of shadows have to be mixed somehow. The baked shadows are not subject to the shadow distance, but realtime shadows are. To make the sudden disappearance of realtime shadows less jarring, we'll begin by adding support to make them fade out as they approach the shadow distance.

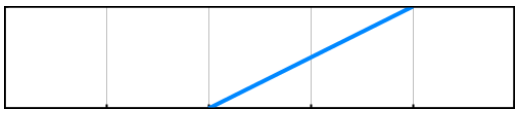


Realtime shadows clamped by shadow distance.

1.1 Fade Range

The simplest way to fade out shadows is to subtract the shadow distance from the fragment's distance to the camera, add one, and then saturate the result: $c - s + 1$. The final value is zero up to one unit less than the shadow distance, after which it linearly increases to 1 as it reaches the shadow distance. At that point realtime shadows are gone and we have to rely on baked shadows only.

But we don't need to always fade across a range of one unit. We can use an arbitrary positive fade range by dividing both distances by it: $\frac{c - s}{r} + 1$.



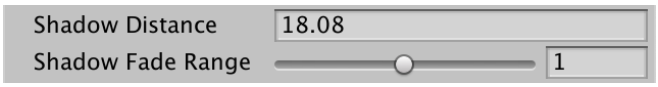
$\frac{c - 4}{2} + 1.$

Add a configuration option to `MyPipelineAsset` for the fade range, with reasonable limits like 0.01–2, and a default of 1. Add it to the constructor arguments, after the shadow distance.

```
	[SerializeField, Range(0.01f, 2f)]
	float shadowFadeRange = 1f;

	…

	protected override IRenderPipeline InternalCreatePipeline () {
		Vector3 shadowCascadeSplit = shadowCascades == ShadowCascades.Four ?
			fourCascadesSplit : new Vector3(twoCascadesSplit, 0f);
		return new MyPipeline(
			dynamicBatching, instancing, (int)shadowMapSize,
			shadowDistance, shadowFadeRange,
			(int)shadowCascades, shadowCascadeSplit
		);
	}
```

`MyPipeline` doesn't need to keep track of the actual fade range. We can rewrite the fade function to $\frac{c}{r} + \left(1 - \frac{s}{r}\right)$, so we can pass two values to the shader and can make do with a single multiply-add instruction. We'll put $\frac{1}{r}$ in the Y component of the global shadow data and $1 - \frac{s}{r}$ in its Z component. Let's move the global shadow data to a field and immediately set its Y value in the constructor method.

```
	Vector4 globalShadowData;

	public MyPipeline (
		bool dynamicBatching, bool instancing,
		int shadowMapSize, float shadowDistance, float shadowFadeRange,
		int shadowCascades, Vector3 shadowCascasdeSplit
	) {
		…
		this.shadowDistance = shadowDistance;
		globalShadowData.y = 1f / shadowFadeRange;
		…
	}
```



Shadow fade range set to 1.

From now on we'll set the global shadow data in `Render`, where we also calculate its Z component, for which we can rely on its Y value.

```
	void Render (ScriptableRenderContext context, Camera camera) {
		…
		cameraBuffer.SetGlobalVectorArray(
			visibleLightSpotDirectionsId, visibleLightSpotDirections
		);
		globalShadowData.z =
			1f - cullingParameters.shadowDistance * globalShadowData.y;
		cameraBuffer.SetGlobalVector(globalShadowDataId, globalShadowData);
		context.ExecuteCommandBuffer(cameraBuffer);
		…
	}
```

Remove the global shadow data code from both `RenderCascadedShadows` and `RenderShadows`, except that in the latter case we still need to put the tile scale in its X component.

```
	void RenderCascadedShadows (ScriptableRenderContext context) {
		…
		//shadowBuffer.SetGlobalVector(
		//	globalShadowDataId, new Vector4(0f, shadowDistance * shadowDistance)
		//);
		…
	}

	void RenderShadows (ScriptableRenderContext context) {
		…
		float tileScale = 1f / split;
		globalShadowData.x = tileScale;
		shadowMap = SetShadowRenderTarget();
		shadowBuffer.BeginSample("Render Shadows");
		//shadowBuffer.SetGlobalVector(
		//	globalShadowDataId, new Vector4(
		//		tileScale, shadowDistance * shadowDistance
		//	)
		//);
		…
	}
```

1.2 Fixing Shadow Clipping

Shadows no longer get clipped at the shadow distance, because we changed the global shadow data. To fix that, first remove the `DistanceToCameraSqr` function from *Lit.hlsl*. Replace it with a function that calculates the shadow blend factor based on the global shadow data.

```
//float DistanceToCameraSqr (float3 worldPos) {
//	float3 cameraToFragment = worldPos - _WorldSpaceCameraPos;
//	return dot(cameraToFragment, cameraToFragment);
//}

float RealtimeToBakedShadowsInterpolator (float3 worldPos) {
	float d = distance(worldPos, _WorldSpaceCameraPos);
	return saturate(d * _GlobalShadowData.y + _GlobalShadowData.z);
}
```

When this value reaches 1 then realtime shadows are no longer used, so we can skip sampling them. Create a convenient function that checks this and use it in `ShadowAttenuation` and `CascadedShadowAttenuation`.

```
bool SkipRealtimeShadows (float3 worldPos) {
	return RealtimeToBakedShadowsInterpolator(worldPos) >= 1.0;
}

…

float ShadowAttenuation (int index, float3 worldPos) {
	…
	if (
		_ShadowData[index].x <= 0 || SkipRealtimeShadows(worldPos)
		//DistanceToCameraSqr(worldPos) > _GlobalShadowData.y
	) {
		return 1.0;
	}
	…
}

…

float CascadedShadowAttenuation (float3 worldPos) {
	…
	//if (DistanceToCameraSqr(worldPos) > _GlobalShadowData.y) {
	if (SkipRealtimeShadows(worldPos)) {
		return 1.0;
	}

	…
}
```

The shadows are now once again clipped beyond the shadow distance.

1.3 Fading

Fading realtime shadows is just a special case of mixing realtime and baked shadows, when no baked shadows are available. We'll do that in a new `MixRealtimeAndBakedShadowAttenuation` function, which initially has parameters for only the realtime shadow attenuation and the world position. Nonexistent baked shadows have an attenuation of 1, so interpolate between the realtime shadows and that based on the interpolator.

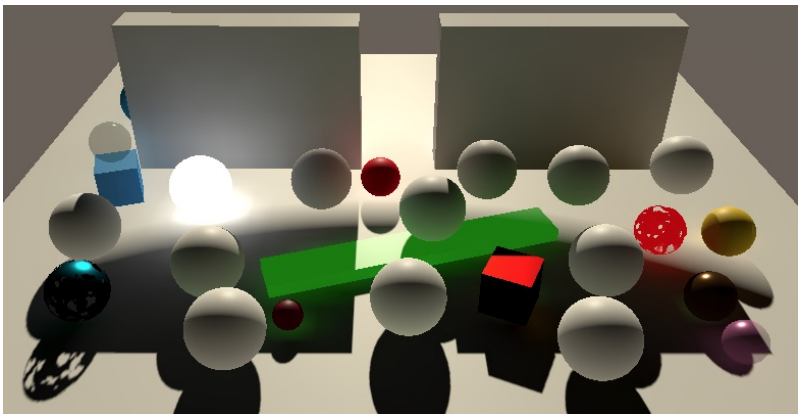
```
float RealtimeToBakedShadowsInterpolator (float3 worldPos) {
	float d = distance(worldPos, _WorldSpaceCameraPos);
	return saturate(d * _GlobalShadowData.y + _GlobalShadowData.z);
}

float MixRealtimeAndBakedShadowAttenuation (float realtime, float3 worldPos) {
	float t = RealtimeToBakedShadowsInterpolator(worldPos);
	float fadedRealtime = lerp(realtime, 1, t);
	return fadedRealtime;
}
```

Shadow attenuation is either 0 or 1, with a little bit of filtering along the edges. Effectively we end up of with either `lerp(0, 1, t)` which is simply `t`, or `lerp(1, 1, t)` which is always `1`. We can get the same results by adding the interpolator to the realtime shadow attenuation and saturating the result, which is a little cheaper to compute.

```
	float fadedRealtime = saturate(realtime + t);
```

Pull the shadow sampling out of `MainLight` to be consistent with `GenericLight`, then mix the shadows before passing the final attenuation to the two light functions in `LitPassFragment`.

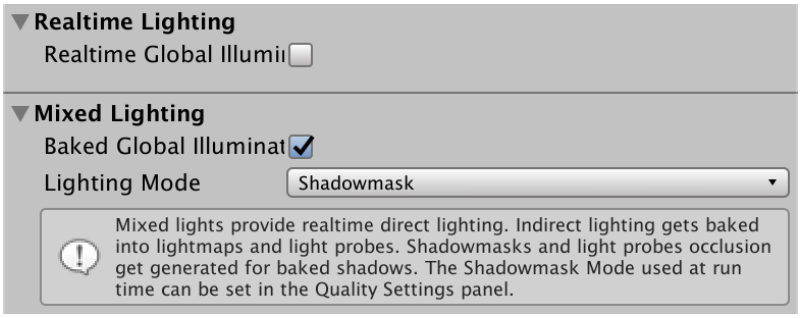
```
float3 MainLight (LitSurface s, float shadowAttenuation) {
	//float shadowAttenuation = CascadedShadowAttenuation(s.position);
	…
}

…

float4 LitPassFragment (
	VertexOutput input, FRONT_FACE_TYPE isFrontFace : FRONT_FACE_SEMANTIC
) : SV_TARGET {
	…

	float3 color = input.vertexLighting * surface.diffuse;
	#if defined(_CASCADED_SHADOWS_HARD) || defined(_CASCADED_SHADOWS_SOFT)
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			CascadedShadowAttenuation(surface.position), surface.position
		);
		color += MainLight(surface, shadowAttenuation);

	#endif

	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		//float shadowAttenuation = ShadowAttenuation(lightIndex, input.worldPos);
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			ShadowAttenuation(lightIndex, surface.position), surface.position
		);
		color += GenericLight(lightIndex, surface, shadowAttenuation);
	}

	…
}
```



Fading realtime shadows.

2 Shadowmask

To bake shadows, set Unity's mixed lighting mode to *Shadowmask*. Also disable realtime global illumination so we can focus on the shadows. We'll initially work with only the main directional light, which should be set to *Mixed* mode.



Baking shadowmask.

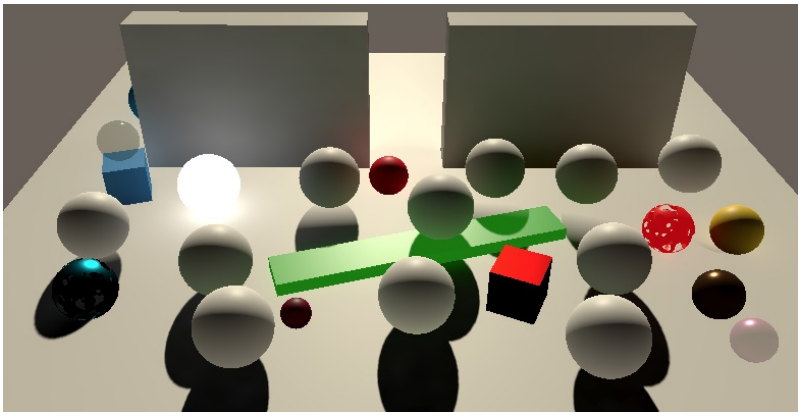
There are two ways that shadowmask mode can be used: either regular *Shadowmask* or *Distance Shadowmask*. We'll use the regular mode for now, which is a quality setting found under the project settings.

Shadowmask mode.

When inspecting the baked light map, you can now select *Baked Shadowmask* from the top-right dropdown menu. When using only a single directional light, the resulting map is black and red. Unshadowed fragments are red, because the red channel is used to store the shadow attenuation. Besides that, the regular lightmap contains baked indirect lighting, exactly like the *Baked Indirect* mixed lighting mode.

Baked shadowmask for plane.

Now that the static shadows are baked, static geometry is no longer included when rendering realtime shadow maps. As we're not yet using the baked shadows, they have disappeared.



No static shadows, only realtime.

2.1 Detecting a Shadowmask

To use baked shadows we must first know that they exist. Whether a shadowmask is used varies per light, so we must check this in `MyPipeline.ConfigureLights`. If a shadow mask exists, we'll enable the *_SHADOWMASK* shader keyword.

```
	const string shadowmaskKeyword = "_SHADOWMASK";

	…

	void ConfigureLights () {
		mainLightExists = false;
		bool shadowmaskExists = false;
		shadowTileCount = 0;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…
		}

		CoreUtils.SetKeyword(cameraBuffer, shadowmaskKeyword, shadowmaskExists);

		…
	}
```

For each visible light, we can check how it was baked by retrieving the baking output from the light object. If its bake type is mixed, then a shadowmask is in used if the light's mixed baking mode is set to shadowmask.

```
		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;

			LightBakingOutput baking = light.light.bakingOutput;
			if (baking.lightmapBakeType == LightmapBakeType.Mixed) {
				shadowmaskExists |=
					baking.mixedLightingMode == MixedLightingMode.Shadowmask;
			}

			…
		}
```

Add a multi-compile directive to our shader for the keyword.

```
			#pragma multi_compile _ DYNAMICLIGHTMAP_ON
			#pragma multi_compile _ _SHADOWMASK
```

2.2 Sampling Baked Shadows

The shadow mask is made available via a `unity_ShadowMask` texture handle and its associated sampler state. Add those to *Lit.hlsl*.

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

The shadow mask uses the same texture coordinates as the light map. Create a function to get the baked shadows, like `GlobalIllumination` with the input and surface as parameters. We don't use the position yet but we will later. The default is to return 1, indicating no baked shadows. What else needs to be done depends on whether we're rendering a static or dynamic object. The result is a `float4` because a shadow mask sample can contain shadow attenuation for up to four lights.

```
float3 GlobalIllumination (VertexOutput input, LitSurface surface) {
	…
}

float4 BakedShadows (VertexOutput input, LitSurface surface) {
	#if defined(LIGHTMAP_ON)
	#else
	#endif
	return 1.0;
}
```

In the case of a static fragment, if there is a shadow mask we'll sample it and that's the result.

```
	#if defined(LIGHTMAP_ON)
		#if defined(_SHADOWMASK)
			return SAMPLE_TEXTURE2D(
				unity_ShadowMask, samplerunity_ShadowMask, input.lightmapUV
			);
		#endif
	#else
```

Retrieve the baked shadows in `LitPassFragment` and pass them to both invocations of `MixRealtimeAndBakedShadowAttenuation`, after the realtime shadow attenuation argument.

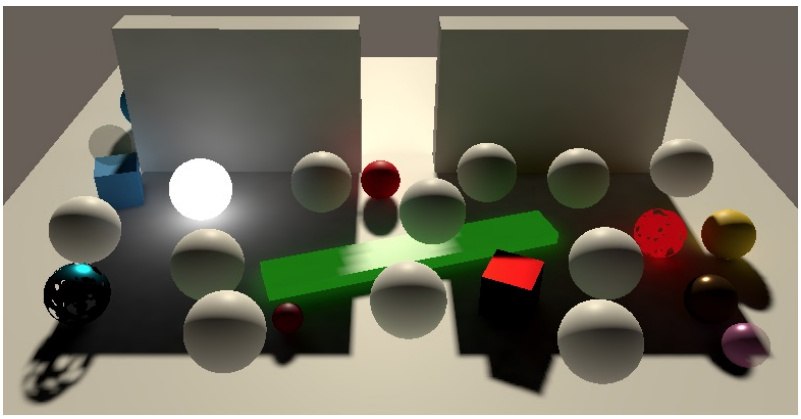
```
float4 LitPassFragment (
	VertexOutput input, FRONT_FACE_TYPE isFrontFace : FRONT_FACE_SEMANTIC
) : SV_TARGET {
	…

	float4 bakedShadows = BakedShadows(input, surface);

	float3 color = input.vertexLighting * surface.diffuse;
	#if defined(_CASCADED_SHADOWS_HARD) || defined(_CASCADED_SHADOWS_SOFT)
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			CascadedShadowAttenuation(surface.position), bakedShadows,
			surface.position
		);
		color += MainLight(surface, shadowAttenuation);
	#endif

	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			ShadowAttenuation(lightIndex, surface.position), bakedShadows,
			surface.position
		);
		color += GenericLight(lightIndex, surface, shadowAttenuation);
	}

	…
}
```

Add a corresponding parameter to `MixRealtimeAndBakedShadowAttenuation`. As we only support the main light, the baked shadow attenuation that we need is stored in the first channel of the baked shadows. Return it instead of the faded realtime shadow attenuation if there is a shadow mask.

```
float MixRealtimeAndBakedShadowAttenuation (
	float realtime, float4 bakedShadows, float3 worldPos
) {
	float t = RealtimeToBakedShadowsInterpolator(worldPos);
	float fadedRealtime = saturate(realtime + t);
	float baked = bakedShadows.x;

	#if defined(_SHADOWMASK)
		return baked;
	#endif
	return fadedRealtime;
}
```

This causes all shadows to disappear, because we haven't told Unity yet that it should send the shadow mask data to the GPU. That's done by enabling the `RendererConfiguration.PerObjectShadowMask` flag of the renderer configuration.

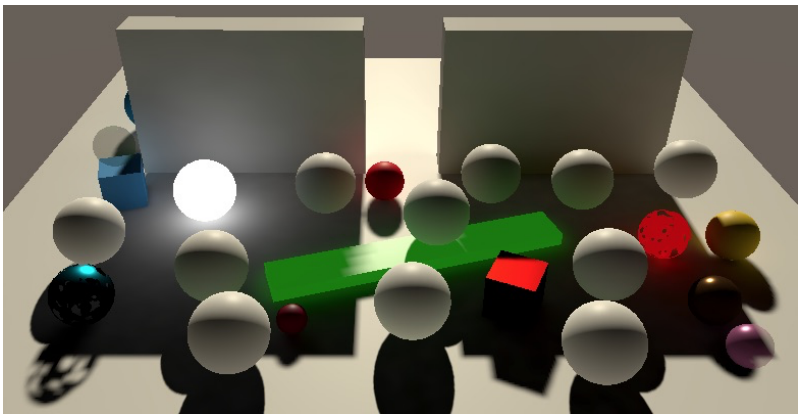
```
		drawSettings.rendererConfiguration |=
			RendererConfiguration.PerObjectReflectionProbes |
			RendererConfiguration.PerObjectLightmaps |
			RendererConfiguration.PerObjectLightProbe |
			RendererConfiguration.PerObjectLightProbeProxyVolume |
			RendererConfiguration.PerObjectShadowMask;
```



Only baked shadows.

Baked shadows now show up. To mix them with the realtime shadows return the minimum of both attenuations.

```
	#if defined(_SHADOWMASK)
		return min(fadedRealtime, baked);
	#endif
```



Mixed realtime and baked shadows.

Note that although the baked shadows cannot change in play mode, the light itself can be adjusted. Changing the light's orientation would produce obviously wrong results, as only the realtime shadows will change along with it. But the light's color and intensity can be changed without invalidating the baked shadows. However, if indirect lighting is baked then the light shouldn't change too much. For example the inconsistency of a red light with blue indirect lighting would be obvious, but a slight inconsistency in intensity won't be.

2.3 Shadow Probes

Because dynamic objects do not have light maps, they also cannot sample the shadow mask texture. But just as with regular baked lighting Unity also bakes shadow attenuation in light probes. So light probes also function as shadow probes. We can tell Unity to send this data to the GPU by enabling the `RendererConfiguration.PerObjectOcclusionProbe` flag.

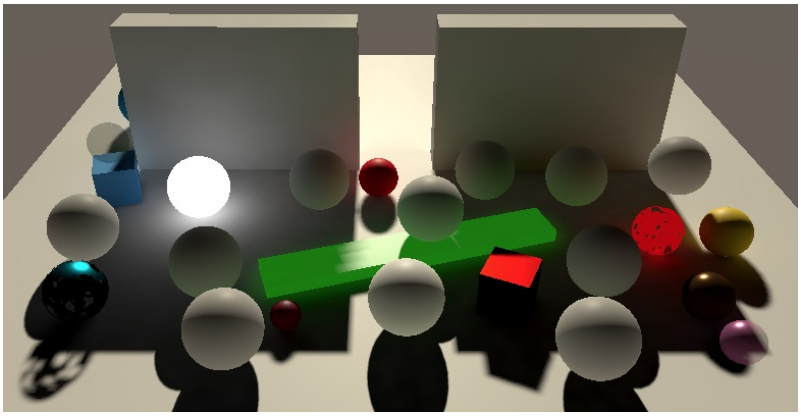
```
			RendererConfiguration.PerObjectShadowMask |
			RendererConfiguration.PerObjectOcclusionProbe;
```

It's made available via `float4 unity_ProbesOcclusion`, which is part of the *UnityPerDraw* buffer.

```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld, unity_WorldToObject;
	float4 unity_LightIndicesOffsetAndCount;
	float4 unity_4LightIndices0, unity_4LightIndices1;
	float4 unity_ProbesOcclusion;
	…
CBUFFER_END
```

Although this data is provided via interpolated light probes, it serves the exact same purpose as the shadow mask, but for dynamic objects. So return it in `BakedShadows` when appropriate.

```
float4 BakedShadows (VertexOutput input, LitSurface surface) {
	#if defined(LIGHTMAP_ON)
		…
	#elif defined(_SHADOWMASK)
		return unity_ProbesOcclusion;
	#endif
	return 1.0;
}
```



Baked shadows also via light probes.

GPU Instancing can also work with `unity_ProbesOcclusion`, but it relies on *SHADOWS_SHADOWMASK* being defined, which doesn't happen automatically. We have to do this ourselves before including *UnityInstancing.hlsl*. This should only be done when necessary, so only for dynamic objects while a shadow mask is in use.

```
#if !defined(LIGHTMAP_ON)
	#if defined(_SHADOWMASK)
		#define SHADOWS_SHADOWMASK
	#endif
#endif
#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/UnityInstancing.hlsl"
```

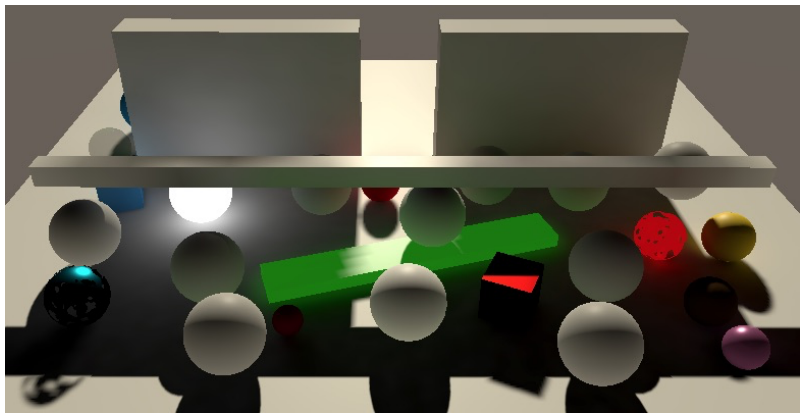
2.4 LPPV Shadows

Shadow probes can also work with light probe proxy volumes. Once again, we have to explicitly enable this, via the `RendererConfiguration.PerObjectOcclusionProbeProxyVolume` flag.

```
			RendererConfiguration.PerObjectShadowMask |
			RendererConfiguration.PerObjectOcclusionProbe |
			RendererConfiguration.PerObjectOcclusionProbeProxyVolume;
```

`BakedShadows` can use the exact same approach as `SampleLightProbes`, except that it needs to invoke the `SampleProbeOcclusion` function instead of `SampleProbeVolumeSH4`, which doesn't have a normal vector parameter.

```
	#elif defined(_SHADOWMASK)
		if (unity_ProbeVolumeParams.x) {
			return SampleProbeOcclusion(
				TEXTURE3D_PARAM(unity_ProbeVolumeSH, samplerunity_ProbeVolumeSH),
				surface.position, unity_ProbeVolumeWorldToObject,
				unity_ProbeVolumeParams.y, unity_ProbeVolumeParams.z,
				unity_ProbeVolumeMin, unity_ProbeVolumeSizeInv
			);
		}
		return unity_ProbesOcclusion;
	#endif
```

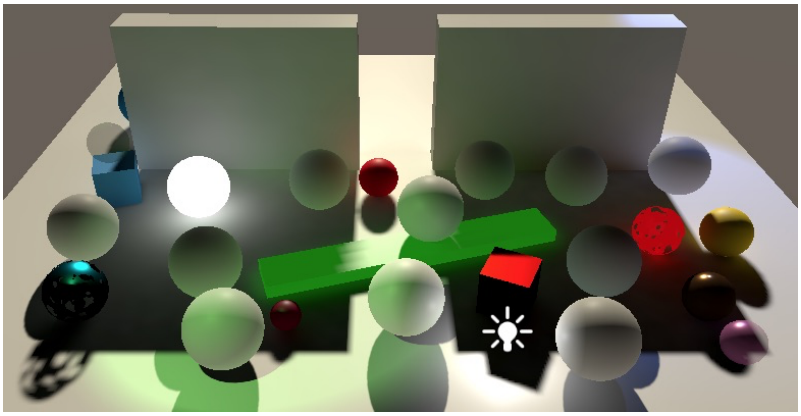


Baked shadows via LPPV.

2.5 Multiple Lights

The shadow mask texture has four channels, so can support up to four lights. That's true per fragment, but it can support an arbitrary amount of lights, by reusing the same channel for multiple lights. The only restriction is that no more than four lights affect the same fragment of the map. If too many lights affect the same area then some are forced to fall back to being fully baked.

Because we only support the main light, additional lights all end up using the same baked shadows, even if they are realtime lights. For example, add two mixed-mode spotlights to the scene, plus a realtime point light. Make sure that the spotlights cast shadows. The point light cannot cast shadows as we don't support it, but it still ends up affected by the baked shadows of the main light.



Four lights all affected by the main baked shadows.

Inspecting the shadow mask will reveal that the spotlights got baked in the R and G channels. It is also possible that a light gets baked in the A channel, but then it won't be visible via the preview window.

Baked shadows for three lights.

Each light gets its own channel in the map. We can select the correct one by taking the dot product of the baked shadows and a mask that has the appropriate channel set to 1. We have to send those masks to the shader, for which we'll create an occlusion mask array. Add a shader identifier and vector array for that to `MyPipeline`.

```
	static int visibleLightOcclusionMasksId =
		Shader.PropertyToID("_VisibleLightOcclusionMasks");
	…
	Vector4[] visibleLightOcclusionMasks = new Vector4[maxVisibleLights];
```

There are four possible mask, which we can predefine in a static array. But it is also possible that some lights don't use the shadow mask. We'll indicate that by setting the first mask component to −1. Make that case the first element of the array, so its length is five.

```
	static Vector4[] occlusionMasks = {
		new Vector4(-1f, 0f, 0f, 0f),
		new Vector4(1f, 0f, 0f, 0f),
		new Vector4(0f, 1f, 0f, 0f),
		new Vector4(0f, 0f, 1f, 0f),
		new Vector4(0f, 0f, 0f, 1f)
	};
```

In `ConfigureLights`, set the occlusion mask for each visible light based on the occlusion mask channel of the baking output. The channel is −1 if the light doesn't use a shadow mask, so add 1 when retrieving the predefined occlusion masks.

```
			LightBakingOutput baking = light.light.bakingOutput;
			visibleLightOcclusionMasks[i] =
				occlusionMasks[baking.occlusionMaskChannel + 1];
```

Set the occlusion masks array in `Render`, along with the other visible light data.

```
		cameraBuffer.SetGlobalVectorArray(
			visibleLightSpotDirectionsId, visibleLightSpotDirections
		);
		cameraBuffer.SetGlobalVectorArray(
			visibleLightOcclusionMasksId, visibleLightOcclusionMasks
		);
```

In *Lit.hlsl*, add the array to the light buffer.

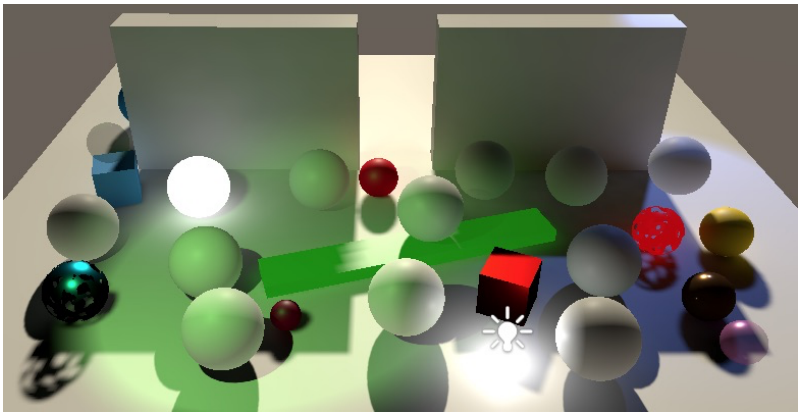
```
CBUFFER_START(_LightBuffer)
	…
	float4 _VisibleLightOcclusionMasks[MAX_VISIBLE_LIGHTS];
CBUFFER_END
```

Add a light index parameter to `MixRealtimeAndBakedShadowAttenuation`. Then we can have it retrieve the occlusion mask, extract the relevant baked shadow attenuation, and check whether the light has baked shadows at all. Only mix realtime and baked shadows when we have valid baked data.

```
float MixRealtimeAndBakedShadowAttenuation (
	float realtime, float4 bakedShadows, int lightIndex, float3 worldPos
) {
	float t = RealtimeToBakedShadowsInterpolator(worldPos);
	float fadedRealtime = saturate(realtime + t);
	float4 occlusionMask = _VisibleLightOcclusionMasks[lightIndex];
	float baked = dot(bakedShadows, occlusionMask);
	bool hasBakedShadows = occlusionMask.x >= 0.0;
	#if defined(_SHADOWMASK)
		if (hasBakedShadows) {
			return min(fadedRealtime, baked);
		}
	#endif
	return fadedRealtime;
}
```

Add the required light index arguments in `LitPassFragment`.

```
	#if defined(_CASCADED_SHADOWS_HARD) || defined(_CASCADED_SHADOWS_SOFT)
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			CascadedShadowAttenuation(surface.position), bakedShadows,
			0, surface.position
		);
		color += MainLight(surface, shadowAttenuation);
	#endif

	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			ShadowAttenuation(lightIndex, surface.position), bakedShadows,
			lightIndex, surface.position
		);
		color += GenericLight(lightIndex, surface, shadowAttenuation);
	}
```



Baked shadows affect the correct lights.

2.6 Distance Shadowmask

While using the regular shadowmask mode, only dynamic objects cast realtime shadows. That can eliminate a lot of realtime shadows, replacing them with shadowmask samples and interpolated probe data. While potentially cheaper to render, the results are of lower quality than when everything uses realtime shadows. On the other hand, the baked shadows are not limited to the shadow distance. The *Distance Shadowmask* mode takes advantage of the latter while doing away with the former. All shadows are realtime, while baked shadows are used beyond the shadow distance. Hence, this mode is more expensive than using only realtime shadows, instead of being cheaper.

Distance shadowmask mode.

The baked data of both shadowmask modes is the same. The only difference is which objects get included when rendering realtime shadows and how the shader combines bakes and realtime shadows. So we need yet another shader variant, this time controlled via the *_DISTANCE_SHADOWMASK* keyword. It's an alternative shadowmask mode, so add it to the same multi-compile directive as *_SHADOWMASK*.

```
			#pragma multi_compile _ _SHADOWMASK _DISTANCE_SHADOWMASK
```

Now we must define *SHADOWS_SHADOWMASK* when either *_SHADOWMASK* or *_DISTANCE_SHADOWMASK* is defined.

```
#if !defined(LIGHTMAP_ON)
	#if defined(_SHADOWMASK) || defined(_DISTANCE_SHADOWMASK)
		#define SHADOWS_SHADOWMASK
	#endif
#endif
```

And the same goes for the conditional compilation in `BakedShadows`.

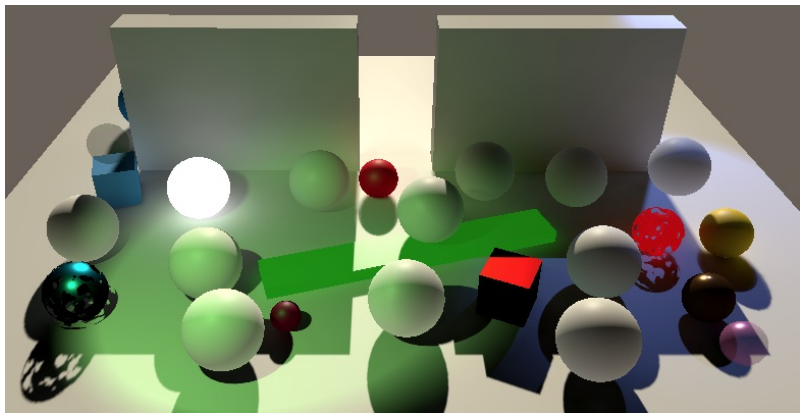
```
float4 BakedShadows (VertexOutput input, LitSurface surface) {
	#if defined(LIGHTMAP_ON)
		#if defined(_SHADOWMASK) || defined(_DISTANCE_SHADOWMASK)
			…
		#endif
	#elif defined(_SHADOWMASK) || defined(_DISTANCE_SHADOWMASK)
		…
	#endif
	return 1.0;
}
```

But in `MixRealtimeAndBakedShadowAttenuation` we have to do something different for each mode. In the case of the regular shadowmask mode, we take the minimum of faded realtime and baked shadows attenuation. But for the distance shadowmask mode we have to transition from the realtime to the baked shadow attenuation based on the interpolator.

```
	#if defined(_SHADOWMASK)
		if (hasBakedShadows) {
			return min(fadedRealtime, baked);
		}
	#elif defined(_DISTANCE_SHADOWMASK)
		if (hasBakedShadows) {
			return lerp(realtime, baked, t);
		}
	#endif
```

Finally, in `MyPipeline` we have to enable the correct keyword, based on the `QualitySettings.shadowmaskMode` property.

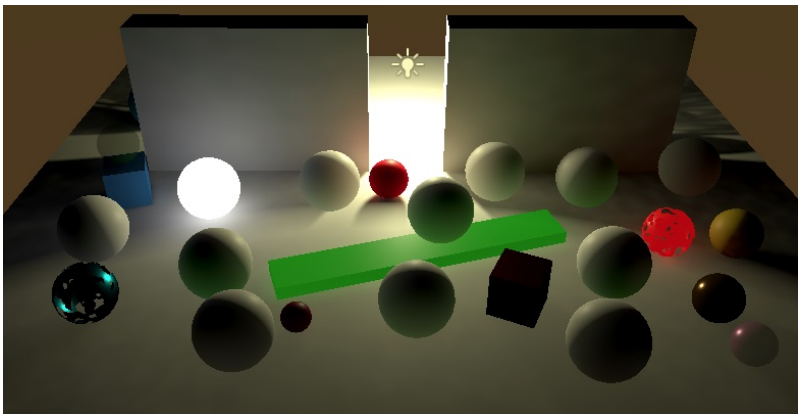
```
	const string distanceShadowmaskKeyword = "_DISTANCE_SHADOWMASK";

	…

	void ConfigureLights () {
		…

		bool useDistanceShadowmask =
			QualitySettings.shadowmaskMode == ShadowmaskMode.DistanceShadowmask;
		CoreUtils.SetKeyword(
			cameraBuffer, shadowmaskKeyword,
			shadowmaskExists && !useDistanceShadowmask
		);
		CoreUtils.SetKeyword(
			cameraBuffer, distanceShadowmaskKeyword,
			shadowmaskExists && useDistanceShadowmask
		);

		…
	}
```



Distance shadowmask mode.

2.7 Baked Point Light Shadows

Although we do not support realtime shadows for point lights, this restriction does not apply to baked shadows. Thus, the baked shadows of a mixed-mode point light show up. At least, that is the case when the regular shadowmask mode is used. Because distance shadowmask mode transitions from realtime to baked shadows, we end up with no point light shadows up to the shadow distance, and baked shadows beyond that.



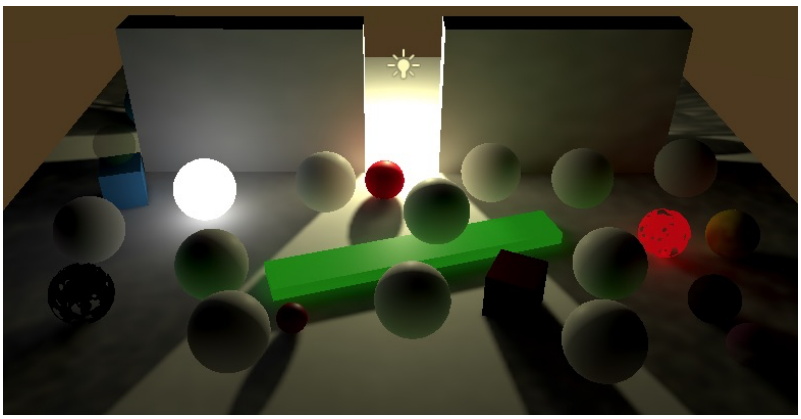
Partially missing point light shadows.

As baked shadows are better than no shadows, let's always use baked shadows for point lights in the case of distance shadowmask mode. To make that possible the shader must be able to detect a point light. We're currently not using the fourth component of the `visibleLightSpotDirections` vectors, so let's set it to 1 in case of a point light, instead of adding yet another array. As the rest of the vector ends up unused anyway, we can simply use `Vector4.one` in `ConfigureLights`.

```
			if (light.lightType == LightType.Spot) {
				…
			}
			else {
				visibleLightSpotDirections[i] = Vector4.one;
			}
```

In *Lit.hlsl*, have `MixRealtimeAndBakedShadowAttenuation` return the baked attenuation in case of a point light, but only in distance shadowmask mode.

```
	#elif defined(_DISTANCE_SHADOWMASK)
		if (hasBakedShadows) {
			bool bakedOnly = _VisibleLightSpotDirections[lightIndex].w > 0.0;
			if (bakedOnly) {
				return baked;
			}
			return lerp(realtime, baked, t);
		}
	#endif
```



Always baked point light shadows.

This check never has to be made for the main light, so let's optimize that away by adding an optional boolean parameter that indicates whether we're mixing shadows for the main light.

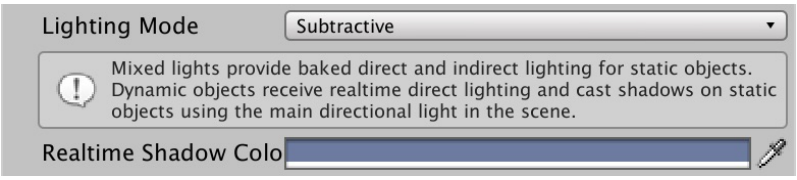
```
float MixRealtimeAndBakedShadowAttenuation (
	float realtime, float4 bakedShadows, int lightIndex, float3 worldPos,
	bool isMainLight = false
) {
	…
		if (!isMainLight && bakedOnly) {
			return baked;
		}
	…
}
```

Enable the optimization when working on the main light in `LitPassFragment`.

```
	#if defined(_CASCADED_SHADOWS_HARD) || defined(_CASCADED_SHADOWS_SOFT)
		float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
			CascadedShadowAttenuation(surface.position), bakedShadows,
			0, surface.position, true
		);
		color += MainLight(surface, shadowAttenuation);
	#endif
```

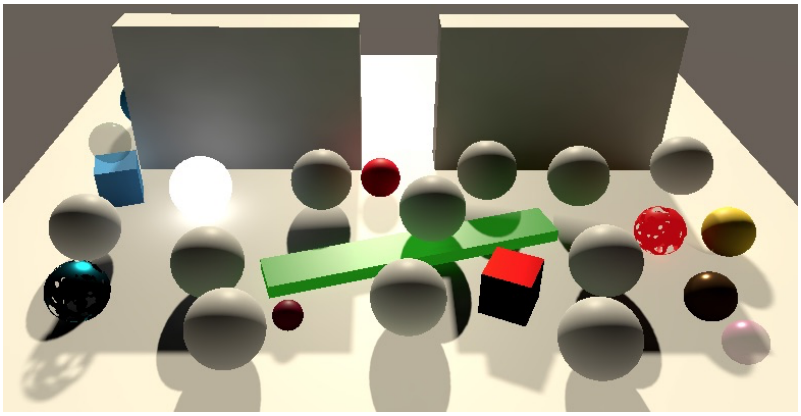
3 Subtractive Lighting

There is a third mixed lighting mode: subtractive. This is a budget option that supports mixed lighting for the main directional light only. When this mode is selected another option is revealed to set the realtime shadow color, which we'll use later.



Subtractive mixed lighting mode.

When subtractive lighting is enabled the main light gets fully baked. The light map is used for static objects, but dynamic objects are still lit realtime and cast realtime shadows. This is true for all other mixed-mode lights as well, but only the shadows of the main light can be mixed. But initially our shader applies both the light map and realtime lighting to static objects, making them too bright.



Main light applied twice to static objects.

3.1 Fixing the Main Light

We need another shader variant for mixed lighting. This time we'll use the *_SUBTRACTIVE_LIGHTING* keyword. Add it to the multi-compile directive.

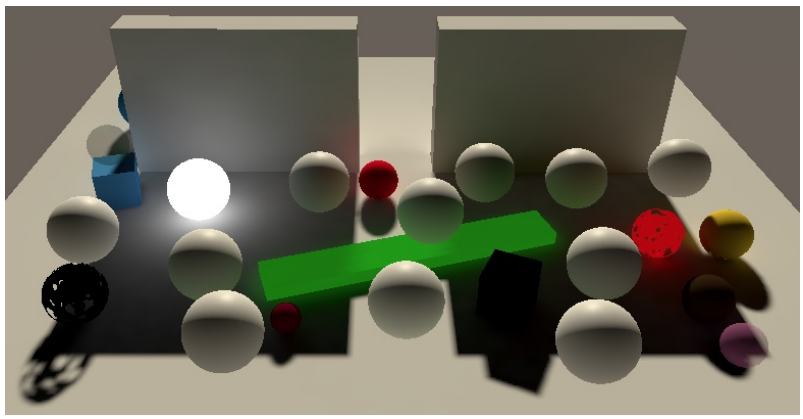
```
			#pragma multi_compile _ _SHADOWMASK _DISTANCE_SHADOWMASK _SUBTRACTIVE_LIGHTING
```

Detecting subtractive lighting and setting the keyword works the same as for the other shadowmask modes. The modes are exclusive, so we can check them separately in `MyPipeline.ConfigureLights`.

```
	const string subtractiveLightingKeyword = "_SUBTRACTIVE_LIGHTING";

	…

	void ConfigureLights () {
		mainLightExists = false;
		bool shadowmaskExists = false;
		bool subtractiveLighting = false;
		shadowTileCount = 0;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…
			if (baking.lightmapBakeType == LightmapBakeType.Mixed) {
				shadowmaskExists |=
					baking.mixedLightingMode == MixedLightingMode.Shadowmask;
				subtractiveLighting |=
					baking.mixedLightingMode == MixedLightingMode.Subtractive;
			}

			…
		}

		…
		CoreUtils.SetKeyword(
			cameraBuffer, distanceShadowmaskKeyword,
			shadowmaskExists && useDistanceShadowmask
		);
		CoreUtils.SetKeyword(
			cameraBuffer, subtractiveLightingKeyword, subtractiveLighting
		);

		…
	}
```

In *Lit.hlsl*, we must skip the realtime main light in `LitPassFragment` for static objects when subtractive lighting is used.

```
	#if defined(_CASCADED_SHADOWS_HARD) || defined(_CASCADED_SHADOWS_SOFT)
		#if !(defined(LIGHTMAP_ON) && defined(_SUBTRACTIVE_LIGHTING))
			float shadowAttenuation = MixRealtimeAndBakedShadowAttenuation(
				CascadedShadowAttenuation(surface.position), bakedShadows,
				0, surface.position, true
			);
			color += MainLight(surface, shadowAttenuation);
		#endif
	#endif
```



Only baked main light used for static objects.

3.2 Shadowing Baked Light

The idea of subtractive lighting is that realtime and baked shadows are mixed, even though a fully-baked light map is used. This means that we have to adjust the baked lighting. We'll do that in a new `SubtractiveLighting` function that takes the surface and sampled global illumination as arguments. Invoke it in `GlobalIllumination` to modify its result, if needed.

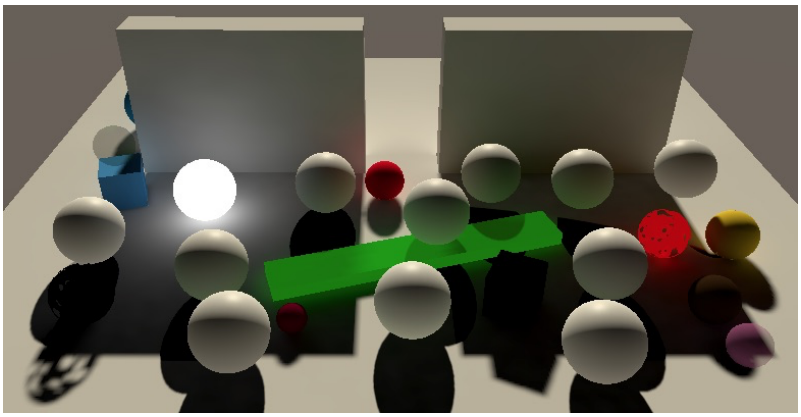
```
float3 GlobalIllumination (VertexOutput input, LitSurface surface) {
	#if defined(LIGHTMAP_ON)
		float3 gi = SampleLightmap(input.lightmapUV);
		#if defined(_SUBTRACTIVE_LIGHTING)
			gi = SubtractiveLighting(surface, gi);
		#endif
		#if defined(DYNAMICLIGHTMAP_ON)
			gi += SampleDynamicLightmap(input.dynamicLightmapUV);
		#endif
		return gi;
	#elif defined(DYNAMICLIGHTMAP_ON)
	…
}
```

`SubtractiveLighting` has to somehow figure out whether a baked lighting sample is shadowed or not. As baked lighting is diffuse only, begin by computing the diffuse lighting of the main light, as if it were realtime. If we use that as the result we simply end up with an unshadowed diffuse-only main light.

```
float3 SubtractiveLighting (LitSurface s, float3 bakedLighting) {
	float3 lightColor = _VisibleLightColors[0].rgb;
	float3 lightDirection = _VisibleLightDirectionsOrPositions[0].xyz;
	float3 diffuse = lightColor * saturate(dot(lightDirection, s.normal));
	return diffuse;
}
```

Also compute the faded realtime shadow attenuation. That's the shadowing caused by dynamic objects, which are not part of the baked lighting. We can guess how much of the baked lighting would've been shadowed if those objects were baked instead, by scaling the realtime diffuse lighting by one minus the realtime shadow attenuation. That would be correct if there were no baked shadows, no indirect lighting, nor any other baked lights. So it's not perfect but the best that we can do. Finally, we find the final lighting by subtracting the guess from the baked lighting. The result should be saturated because an incorrect guess could produce negative lighting.

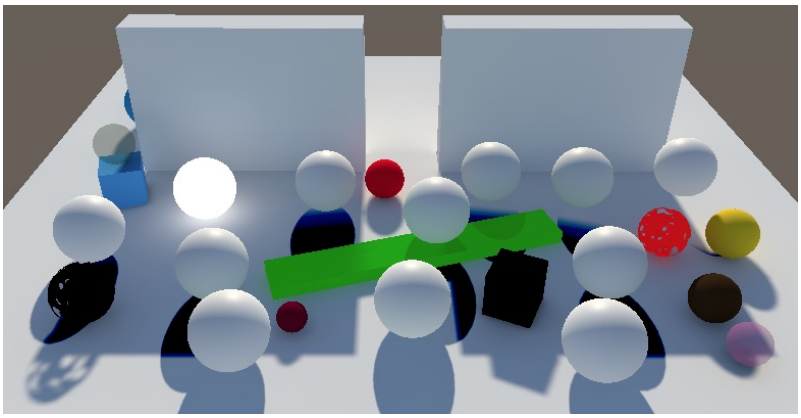
```
float3 SubtractiveLighting (LitSurface s, float3 bakedLighting) {
	float3 lightColor = _VisibleLightColors[0].rgb;
	float3 lightDirection = _VisibleLightDirectionsOrPositions[0].xyz;
	float3 diffuse = lightColor * saturate(dot(lightDirection, s.normal));
	float shadowAttenuation = saturate(
		CascadedShadowAttenuation(s.position) +
		RealtimeToBakedShadowsInterpolator(s.position)
	);
	float3 shadowedLightingGuess = diffuse * (1.0 - shadowAttenuation);
	float3 subtractedLighting = bakedLighting - shadowedLightingGuess;
	return saturate(subtractedLighting);
}
```



Subtracted lighting.

3.3 Shadow Color

The initial results look acceptable, but only when black shadows are correct. Setting the intensity multiplier of the environmental lighting to 1 reveals that our guess is quite wrong.



Guessed too dark.

We cannot improve our guess in the shader, but what we can do is limit how much light we subtract. That's what the shadow color setting is for. It can be retrieved via `RenderSettings.subtractiveShadowColor` and we should set it in `ConfigureLights` when we detect subtractive lighting mode, via a `_SubtractiveShadowColor` shader property.

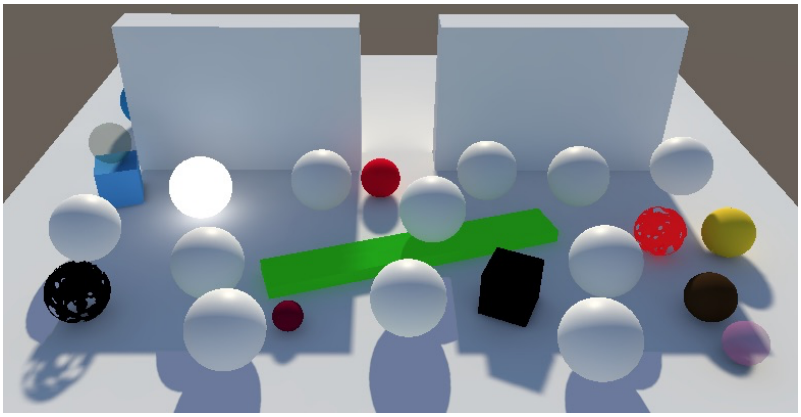
```
	static int subtractiveShadowColorId =
		Shader.PropertyToID("_SubtractiveShadowColor");

	…

	void ConfigureLights () {
		…
				shadowmaskExists |=
					baking.mixedLightingMode == MixedLightingMode.Shadowmask;
				//subtractiveLighting =
					//baking.mixedLightingMode |= MixedLightingMode.Subtractive;
				if (baking.mixedLightingMode == MixedLightingMode.Subtractive) {
					subtractiveLighting = true;
					cameraBuffer.SetGlobalColor(
						subtractiveShadowColorId,
						RenderSettings.subtractiveShadowColor.linear
					);
				}
		…
	}
```

Add the color to the shadow buffer.

```
CBUFFER_START(_ShadowBuffer)
	…
	float4 _SubtractiveShadowColor;
CBUFFER_END
```

In `SubtractiveLighting`, take the maximum of the subtracted lighting and the shadow color, to limit how much light gets removed. But this could brighten the baked lighting, which should never happen. So the final result is the minimum of the baked and subtracted lighting.

```
	float3 subtractedLighting = bakedLighting - shadowedLightingGuess;
	subtractedLighting = max(subtractedLighting, _SubtractiveShadowColor);
	return min(bakedLighting, subtractedLighting);
```



Colored shadows.

This produces reasonable results if the environmental lighting is mostly uniform and matches the shadow color. The default shadow color is a good guess for Unity's standard environmental lighting settings.

3.4 Shadow Strength

If the main light's shadows strength is reduced, the effect of the subtracted lighting should be reduced by the same amount. Because that also affects the shadow color, we should apply the shadow strength after applying the color, by interpolating between the baked and subtracted light based on the shadow strength.

```
	subtractedLighting = max(subtractedLighting, _SubtractiveShadowColor);
	subtractedLighting =
		lerp(bakedLighting, subtractedLighting, _CascadedShadowStrength);
	return min(bakedLighting, subtractedLighting);
```

But that would apply the shadow strength twice to the realtime shadows, because `CascadedShadowAttenuation` also applies it. So let's make it possible for that function to ignore the shadow strength.

```
float CascadedShadowAttenuation (float3 worldPos, bool applyStrength = true) {
	…

	if (applyStrength) {
		return lerp(1, attenuation, _CascadedShadowStrength);
	}
	else {
		return attenuation;
	}
}
```

Only in `SubtractiveLighting` should it not apply the strength.

```
	float shadowAttenuation = saturate(
		CascadedShadowAttenuation(s.position, false) +
		RealtimeToBakedShadowsInterpolator(s.position)
	);
```

3.5 Shadow Probes

Subtractive lighting now works correctly for static objects, but dynamic objects only receive realtime shadows. Once again we can rely on shadow probes. First, also define *SHADOWS_SHADOWMASK* for dynamic objects when subtractive lighting is used, so GPU instancing remains functional.

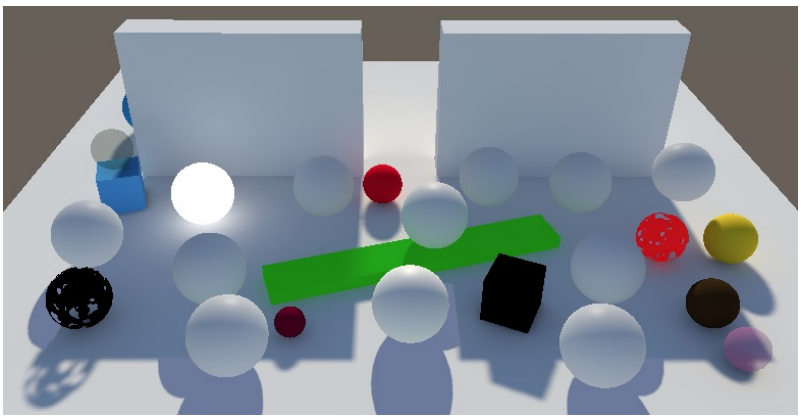
```
#if !defined(LIGHTMAP_ON)
	#if defined(_SHADOWMASK) || defined(_DISTANCE_SHADOWMASK) || \
		defined(_SUBTRACTIVE_LIGHTING)

		#define SHADOWS_SHADOWMASK
	#endif
#endif
```

Sample the shadow probes in `BakedShadows` in the same case.

```
float4 BakedShadows (VertexOutput input, LitSurface surface) {
	…
	#elif defined(_SHADOWMASK) || defined(_DISTANCE_SHADOWMASK) || \
		defined(_SUBTRACTIVE_LIGHTING)

		if (unity_ProbeVolumeParams.x) {
			…
		}
		return unity_ProbesOcclusion;
	#endif
	return 1.0;
}
```

In `MixRealtimeAndBakedShadowAttenuation`, if we have subtractive lighting and are working on dynamic objects, then we must mix shadows like the regular shadowmask mode, but only for the main light. Thus, we always use the first channel of the baked shadows.

```
float MixRealtimeAndBakedShadowAttenuation (
	float realtime, float4 bakedShadows, int lightIndex, float3 worldPos,
	bool isMainLight = false
) {
	…
	#if defined(_SHADOWMASK)
		…
	#elif defined(_DISTANCE_SHADOWMASK)
		…
	#elif defined(_SUBTRACTIVE_LIGHTING)
		#if !defined(LIGHTMAP_ON)
			if (isMainLight) {
				return min(fadedRealtime, bakedShadows.x);
			}
		#endif
	#endif
	return fadedRealtime;
}
```



Shadow probes in use.

This works when a main light exists, but it is possible that we don't render a main light even though subtractive lighting is used. That happens when realtime shadows are omitted because none end up within the shadow distance. In that case we can suffice with directly returning the baked shadow attenuation. But because there is no separate main light we have to check whether we're dealing with the first light index.

```
	#elif defined(_SUBTRACTIVE_LIGHTING)
		#if !defined(LIGHTMAP_ON)
			if (isMainLight) {
				return min(fadedRealtime, bakedShadows.x);
			}
		#endif
		#if !defined(_CASCADED_SHADOWS_HARD) && !defined(_CASCADED_SHADOWS_SOFT)
			if (lightIndex == 0) {
				return bakedShadows.x;
			}
		#endif
	#endif
```

The next tutorial is Level of Detail.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

BECOME A PATRON

Or make a direct donation!

made by Jasper Flick