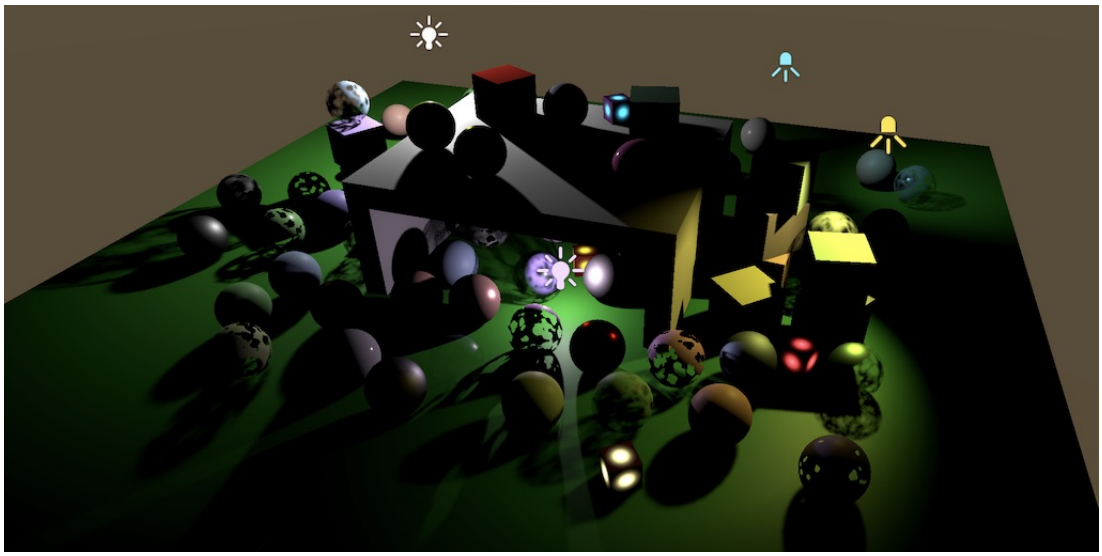
updated 2023-08-03 published 2020-06-30

Point and Spot Shadows Perspective Shadows

Mix baked and realtime shadows for point and spot lights.
Add a second shadow atlas.
Render and sample shadows with a perspective projection.
Use custom cube maps.

This is the tenth part of a tutorial series about creating a custom scriptable render pipeline. It adds support for realtime shadows of point and spot lights.

This tutorial is made with Unity 2019.4.1f1 and upgraded to 2022.3.5f1.



100% realtime shadows.

1 Spot Light Shadows

We'll start by supporting realtime shadows for spot lights. We'll use the same approach that we use for directional lights, with a few changes. We'll also keep support as simple as possible, using a uniformly tiled shadow atlas and filling it with shadowed lights in the order provided by Unity.

1.1 Shadow Mixing

The first step is to make it possible to mix baked and realtime shadows. Adjust `GetOtherShadowAttenuation` in *Shadows* so it behaves like `GetDirectionalShadowAttenuation`, except that it uses the other shadow data and relies on a new `GetOtherShadow` function. The new function initially returns 1 because other lights don't have realtime shadows yet.

```
float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	return 1.0;
}

float GetOtherShadowAttenuation (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	#if !defined(_RECEIVE_SHADOWS)
		return 1.0;
	#endif

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

The global strength is used to determine whether we can skip sampling realtime shadows, either because we're beyond the shadow distance or outside the largest cascade sphere. However, cascades are only applicable to directional shadows. They don't make sense for other light because those have a fixed position, thus their shadow maps don't move with the view. Having said that, it's a good idea to fade all shadows out the same way, otherwise we could end up with some areas of the screen that don't have directional shadows but do have other shadows. So we'll use the same global shadow strength for everything.

One corner case that we have to deal with is when no directional shadows exist while we do have other shadows. When this happens there aren't any cascades, so they shouldn't affect the global shadow strength. And we still need the shadow distance fade values. So let's move the code to set the cascade count and distance fade from `Shadows.RenderDirectionShadows` to `Shadows.Render` and set the cascade count to zero when appropriate.

```
	public void Render () {
		…
		buffer.SetGlobalInt(
			cascadeCountId,
			shadowedDirLightCount > 0 ? settings.directional.cascadeCount : 0
		);
		float f = 1f - settings.directional.cascadeFade;
		buffer.SetGlobalVector(
			shadowDistanceFadeId, new Vector4(
				1f / settings.maxDistance, 1f / settings.distanceFade,
				1f / (1f - f * f)
			)
		);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}

	void RenderDirectionalShadows () {
		…

		//buffer.SetGlobalInt(cascadeCountId, settings.directional.cascadeCount);
		buffer.SetGlobalVectorArray(
			cascadeCullingSpheresId, cascadeCullingSpheres
		);
		buffer.SetGlobalVectorArray(cascadeDataId, cascadeData);
		buffer.SetGlobalMatrixArray(dirShadowMatricesId, dirShadowMatrices);
		//float f = 1f - settings.directional.cascadeFade;
		//buffer.SetGlobalVector(
		//	shadowDistanceFadeId, new Vector4(
		//		1f / settings.maxDistance, 1f / settings.distanceFade,
		//		1f / (1f - f * f)
		//	)
		//);
		…
	}
```

Then we have to ensure that the global strength isn't incorrectly set to zero after the cascade loop in `GetShadowData`.

```
	if (i == _CascadeCount && _CascadeCount > 0) {
		data.strength = 0.0;
	}
```

1.2 Other Realtime Shadows

Directional shadows have their own atlas map. We'll use a separate atlas for all other shadowed lights and count them separately. Let's use a maximum of sixteen other lights with realtime shadows.

```
	const int maxShadowedDirLightCount = 4, maxShadowedOtherLightCount = 16;
	const int maxCascades = 4;

	…

	int shadowedDirLightCount, shadowedOtherLightCount;

	…

	public void Setup (…) {
		…
		shadowedDirLightCount = shadowedOtherLightCount = 0;
		useShadowMask = false;
	}
```

This means that we can end up with lights that have shadows enabled but won't fit in the atlas. Which lights won't get shadows depends on their place in the visible light list. We simply won't reserve shadows for lights that lose out, but if they have baked shadows we can still allow those. To make this possible first refactor `ReserveOtherShadows` so it immediately returns when the light doesn't have shadows. Otherwise it checks for a shadow mask channel—using −1 by default—and then always returns the shadow strength and channel.

```
	public Vector4 ReserveOtherShadows (Light light, int visibleLightIndex) {
		if (light.shadows == LightShadows.None || light.shadowStrength <= 0f) {
			return new Vector4(0f, 0f, 0f, -1f);
		}

		float maskChannel = -1f;
		//if (light.shadows != LightShadows.None && light.shadowStrength > 0f) {
		LightBakingOutput lightBaking = light.bakingOutput;
		if (
			lightBaking.lightmapBakeType == LightmapBakeType.Mixed &&
			lightBaking.mixedLightingMode == MixedLightingMode.Shadowmask
		) {
			useShadowMask = true;
			maskChannel = lightBaking.occlusionMaskChannel;
		}
		return new Vector4(
			light.shadowStrength, 0f, 0f,
			maskChannel
		);
			//}
		//}
		//return new Vector4(0f, 0f, 0f, -1f);
	}
```

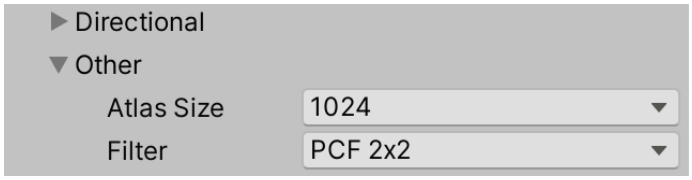
Then before returning check whether increasing the light count would go over the max, or if there are no shadows to render for this light. If so return with a negative shadow strength and the mask channel, so baked shadows are used when appropriate. Otherwise proceed to increment the light count and set the tile index.

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

		return new Vector4(
			light.shadowStrength, shadowedOtherLightCount++, 0f,
			maskChannel
		);
```

1.3 Two Atlases

Because directional and other shadows are kept separate we can configure them differently. Add a new configuration struct and field to `ShadowSettings` for the other shadows, only containing an atlas size and filter as cascades don't apply.

```
	[System.Serializable]
	public struct Other {

		public MapSize atlasSize;

		public FilterMode filter;
	}

	public Other other = new Other {
		atlasSize = MapSize._1024,
		filter = FilterMode.PCF2x2
	};
```



Settings for other shadows.

Add a multi-compile directive to the *CustomLit* pass of our *Lit* shader to support shadow filtering for other shadows.

```
			#pragma multi_compile _ _OTHER_PCF3 _OTHER_PCF5 _OTHER_PCF7
```

And add a corresponding keyword array to `Shadows`.

```
	static string[] otherFilterKeywords = {
		"_OTHER_PCF3",
		"_OTHER_PCF5",
		"_OTHER_PCF7",
	};
```

We also need to keep track of shader property identifiers for the other shadow atlas and matrices, plus an array to hold the matrices.

```
	static int
		dirShadowAtlasId = Shader.PropertyToID("_DirectionalShadowAtlas"),
		dirShadowMatricesId = Shader.PropertyToID("_DirectionalShadowMatrices"),
		otherShadowAtlasId = Shader.PropertyToID("_OtherShadowAtlas"),
		otherShadowMatricesId = Shader.PropertyToID("_OtherShadowMatrices"),
		…;

	…

	static Matrix4x4[]
		dirShadowMatrices = new Matrix4x4[maxShadowedDirLightCount * maxCascades],
		otherShadowMatrices = new Matrix4x4[maxShadowedOtherLightCount];
```

We already send the atlas size of the directional atlas to the GPU, using the XY components of a vector. We now also need to send the size of the other atlas, which we can put in ZW components of the same vector. Promote it to a field and move setting the global vector from `RenderDirectionalShadows` to `Render`. Then `RenderDirectionalShadows` only has to assign to the XY components of the field.

```
	Vector4 atlasSizes;

	…

	public void Render () {
		…
		buffer.SetGlobalVector(shadowAtlasSizeId, atlasSizes);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}

	void RenderDirectionalShadows () {
		int atlasSize = (int)settings.directional.atlasSize;
		atlasSizes.x = atlasSize;
		atlasSizes.y = 1f / atlasSize;
		…
		//buffer.SetGlobalVector(
		//	shadowAtlasSizeId, new Vector4(atlasSize, 1f / atlasSize)
		//);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}
```

After that, duplicate `RenderDirectionalShadows` and rename it to `RenderOtherShadows`. Change it so it uses the correct settings, atlas, matrices, and sets the correct size components. Then remove the cascade and culling sphere code from it. Also remove the invocation of `RenderDirectionalShadows`, but keep the loop.

```
	void RenderOtherShadows () {
		int atlasSize = (int)settings.other.atlasSize;
		atlasSizes.z = atlasSize;
		atlasSizes.w = 1f / atlasSize;
		buffer.GetTemporaryRT(
			otherShadowAtlasId, atlasSize, atlasSize,
			32, FilterMode.Bilinear, RenderTextureFormat.Shadowmap
		);
		buffer.SetRenderTarget(
			otherShadowAtlasId,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
		);
		buffer.ClearRenderTarget(true, false, Color.clear);
		buffer.BeginSample(bufferName);
		ExecuteBuffer();

		int tiles = shadowedOtherLightCount;
		int split = tiles <= 1 ? 1 : tiles <= 4 ? 2 : 4;
		int tileSize = atlasSize / split;

		for (int i = 0; i < shadowedOtherLightCount; i++) {
			//RenderDirectionalShadows(i, split, tileSize);
		}

		//buffer.SetGlobalVectorArray(
		//	cascadeCullingSpheresId, cascadeCullingSpheres
		//);
		//buffer.SetGlobalVectorArray(cascadeDataId, cascadeData);
		buffer.SetGlobalMatrixArray(otherShadowMatricesId, otherShadowMatrices);
		SetKeywords(
			otherFilterKeywords, (int)settings.other.filter - 1
		);
		//SetKeywords(
		//	cascadeBlendKeywords, (int)settings.directional.cascadeBlend - 1
		//);
		buffer.EndSample(bufferName);
		ExecuteBuffer();
	}
```

Now we can render both directional and other shadows in `RenderShadows` when needed. If there aren't other shadows then we need a dummy texture for them, just like for directional shadows. We can simply use the directional shadow atlas as the dummy.

```
	public void Render () {
		if (shadowedDirLightCount > 0) {
			RenderDirectionalShadows();
		}
		else {
			buffer.GetTemporaryRT(
				dirShadowAtlasId, 1, 1,
				32, FilterMode.Bilinear, RenderTextureFormat.Shadowmap
			);
		}
		if (shadowedOtherLightCount > 0) {
			RenderOtherShadows();
		}
		else {
			buffer.SetGlobalTexture(otherShadowAtlasId, dirShadowAtlasId);
		}

		…
	}
```

And release the other shadow atlas in `Cleanup`, in this case only if we did get one.

```
	public void Cleanup () {
		buffer.ReleaseTemporaryRT(dirShadowAtlasId);
		if (shadowedOtherLightCount > 0) {
			buffer.ReleaseTemporaryRT(otherShadowAtlasId);
		}
		ExecuteBuffer();
	}
```

1.4 Rendering Spot Shadows

To render the shadows of a spot light we need to know its visible light index, slope scale bias, and normal bias. So create a `ShadowedOtherLight` struct with fields for those and add an array field for them, similar to how we keep track of data for directional shadows.

```
	struct ShadowedOtherLight {
		public int visibleLightIndex;
		public float slopeScaleBias;
		public float normalBias;
	}

	ShadowedOtherLight[] shadowedOtherLights =
		new ShadowedOtherLight[maxShadowedOtherLightCount];
```

Copy the relevant data at the end of `ReserveOtherShadows`, before returning.

```
	public Vector4 ReserveOtherShadows (Light light, int visibleLightIndex) {
		…

		shadowedOtherLights[shadowedOtherLightCount] = new ShadowedOtherLight {
			visibleLightIndex = visibleLightIndex,
			slopeScaleBias = light.shadowBias,
			normalBias = light.shadowNormalBias
		};

		return new Vector4(
			light.shadowStrength, shadowedOtherLightCount++, 0f,
			maskChannel
		);
	}
```

However, at this point we should realize that we're not guaranteed to send the correct light index to `ReserveOtherShadows` in `Lighting`, because it's passing its own index for other lights. When there are shadowed directional lights the index will be wrong. We fix this by adding a parameter for the correct visible light index to the light setup methods and use that one when reserving shadows. Let's also do this for directional lights, for consistency.

```
	void SetupDirectionalLight (
		int index, int visibleIndex, ref VisibleLight visibleLight
	) {
		…
		dirLightShadowData[index] =
			shadows.ReserveDirectionalShadows(visibleLight.light, visibleIndex);
	}

	void SetupPointLight (
		int index, int visibleIndex, ref VisibleLight visibleLight
	) {
		…
		otherLightShadowData[index] =
			shadows.ReserveOtherShadows(light, visibleIndex);
	}

	void SetupSpotLight (
		int index, int visibleIndex, ref VisibleLight visibleLight
	) {
		…
		otherLightShadowData[index] =
			shadows.ReserveOtherShadows(light, visibleIndex);
	}
```

Adjust `SetupLights` so it passes the visible light index to the setup methods.

```
			switch (visibleLight.lightType) {
				case LightType.Directional:
					if (dirLightCount < maxDirLightCount) {
						SetupDirectionalLight(
							dirLightCount++, i, ref visibleLight
						);
					}
					break;
				case LightType.Point:
					if (otherLightCount < maxOtherLightCount) {
						newIndex = otherLightCount;
						SetupPointLight(otherLightCount++, i, ref visibleLight);
					}
					break;
				case LightType.Spot:
					if (otherLightCount < maxOtherLightCount) {
						newIndex = otherLightCount;
						SetupSpotLight(otherLightCount++, i, ref visibleLight);
					}
					break;
			}
```

Back to `Shadows`, create a `RenderSpotShadows` method that does the same as the `RenderDirectionalShadows` method with parameters, except that it doesn't loop over multiple tiles, has no cascades, and no culling factor. In this case we can use `CullingResults.ComputeSpotShadowMatricesAndCullingPrimitives`, which works like `ComputeDirectionalShadowMatricesAndCullingPrimitives` except that it only has the visible light index, matrices, and split data as arguments. In Unity 2022 we also have to use `BatchCullingProjectionType.Perspective` instead of orthographic.

```
	void RenderSpotShadows (int index, int split, int tileSize) {
		ShadowedOtherLight light = shadowedOtherLights[index];
		var shadowSettings = new ShadowDrawingSettings(
			cullingResults, light.visibleLightIndex,
			BatchCullingProjectionType.Perspective
		);
		cullingResults.ComputeSpotShadowMatricesAndCullingPrimitives(
			light.visibleLightIndex, out Matrix4x4 viewMatrix,
			out Matrix4x4 projectionMatrix, out ShadowSplitData splitData
		);
		shadowSettings.splitData = splitData;
		otherShadowMatrices[index] = ConvertToAtlasMatrix(
			projectionMatrix * viewMatrix,
			SetTileViewport(index, split, tileSize), split
		);
		buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
		buffer.SetGlobalDepthBias(0f, light.slopeScaleBias);
		ExecuteBuffer();
		context.DrawShadows(ref shadowSettings);
		buffer.SetGlobalDepthBias(0f, 0f);
	}
```

Invoke this method inside the loop of `RenderOtherShadows`.

```
		for (int i = 0; i < shadowedOtherLightCount; i++) {
			RenderSpotShadows(i, split, tileSize);
		}
```

Shadow atlas for three spot lights.

1.5 No Pancaking

Shadows now get rendered for spot lights, using the same *ShadowCaster* pass that is used for directional shadows. This works fine, except that shadow pancaking is only valid for orthographic shadow projections, used for directional lights that are assumed to be infinitely far away. In the case of spot lights—which do have a position—shadow casters can end up partially behind the light's position. As we're using a perspective projection in this case, clamping vertices to the near plane would severely distort such shadows. So we should turn off clamping when pancaking isn't appropriate.

We can tell the shader whether pancaking is active via a global shader property, which we'll name *_ShadowPancaking*. Keep track of its identifier in `Shadows`.

```
	static int
		…
		shadowDistanceFadeId = Shader.PropertyToID("_ShadowDistanceFade"),
		shadowPancakingId = Shader.PropertyToID("_ShadowPancaking");
```

Set it to 1 before rendering shadows `RenderDirectionalShadows`.

```
		buffer.ClearRenderTarget(true, false, Color.clear);
		buffer.SetGlobalFloat(shadowPancakingId, 1f);
		buffer.BeginSample(bufferName);
```

And to zero in `RenderOtherShadows`.

```
		buffer.ClearRenderTarget(true, false, Color.clear);
		buffer.SetGlobalFloat(shadowPancakingId, 0f);
		buffer.BeginSample(bufferName);
```

Then add it to the *ShadowCaster* pass of our *Lit* shader as a boolean, using it to only clamp when appropriate.

```
bool _ShadowPancaking;

Varyings ShadowCasterPassVertex (Attributes input) {
	…

	if (_ShadowPancaking) {
		#if UNITY_REVERSED_Z
			output.positionCS.z = min(
				output.positionCS.z, output.positionCS.w * UNITY_NEAR_CLIP_VALUE
			);
		#else
			output.positionCS.z = max(
				output.positionCS.z, output.positionCS.w * UNITY_NEAR_CLIP_VALUE
			);
		#endif
	}

	output.baseUV = TransformBaseUV(input.baseUV);
	return output;
}
```

1.6 Sampling Spot Shadows

To sample the other shadows we have to adjust *Shadows*. Begin by defining the other filter and max shadowed other light count macros. Then add the other shadow atlas and other shadow matrices array.

```
#if defined(_OTHER_PCF3)
	#define OTHER_FILTER_SAMPLES 4
	#define OTHER_FILTER_SETUP SampleShadow_ComputeSamples_Tent_3x3
#elif defined(_OTHER_PCF5)
	#define OTHER_FILTER_SAMPLES 9
	#define OTHER_FILTER_SETUP SampleShadow_ComputeSamples_Tent_5x5
#elif defined(_OTHER_PCF7)
	#define OTHER_FILTER_SAMPLES 16
	#define OTHER_FILTER_SETUP SampleShadow_ComputeSamples_Tent_7x7
#endif

#define MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT 4
#define MAX_SHADOWED_OTHER_LIGHT_COUNT 16
#define MAX_CASCADE_COUNT 4

TEXTURE2D_SHADOW(_DirectionalShadowAtlas);
TEXTURE2D_SHADOW(_OtherShadowAtlas);
#define SHADOW_SAMPLER sampler_linear_clamp_compare
SAMPLER_CMP(SHADOW_SAMPLER);

CBUFFER_START(_CustomShadows)
	…
	float4x4 _DirectionalShadowMatrices
		[MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT * MAX_CASCADE_COUNT];
	float4x4 _OtherShadowMatrices[MAX_SHADOWED_OTHER_LIGHT_COUNT];
	…
CBUFFER_END
```

Duplicate `SampleDirectionalShadowAtlas` and `FilterDirectionalShadow` and rename and adjust them to so they work for other shadows. Note that we need to use the other component pair of the atlas size vector for this version.

```
float SampleOtherShadowAtlas (float3 positionSTS) {
	return SAMPLE_TEXTURE2D_SHADOW(
		_OtherShadowAtlas, SHADOW_SAMPLER, positionSTS
	);
}

float FilterOtherShadow (float3 positionSTS) {
	#if defined(OTHER_FILTER_SETUP)
		real weights[OTHER_FILTER_SAMPLES];
		real2 positions[OTHER_FILTER_SAMPLES];
		float4 size = _ShadowAtlasSize.wwzz;
		OTHER_FILTER_SETUP(size, positionSTS.xy, weights, positions);
		float shadow = 0;
		for (int i = 0; i < OTHER_FILTER_SAMPLES; i++) {
			shadow += weights[i] * SampleOtherShadowAtlas(
				float3(positions[i].xy, positionSTS.z)
			);
		}
		return shadow;
	#else
		return SampleOtherShadowAtlas(positionSTS);
	#endif
}
```

The `OtherShadowData` struct now also needs a tile index.

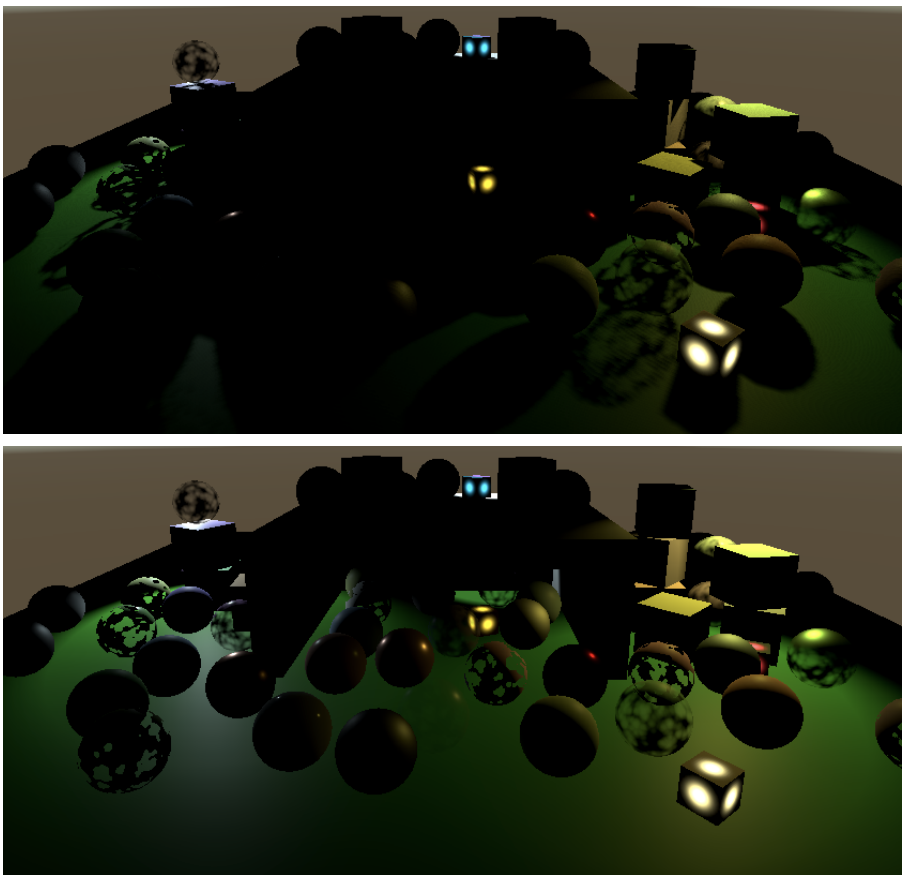
```
struct OtherShadowData {
	float strength;
	int tileIndex;
	int shadowMaskChannel;
};
```

Which is set by `GetOtherShadowData` in *Light*.

```
OtherShadowData GetOtherShadowData (int lightIndex) {
	OtherShadowData data;
	data.strength = _OtherLightShadowData[lightIndex].x;
	data.tileIndex = _OtherLightShadowData[lightIndex].y;
	data.shadowMaskChannel = _OtherLightShadowData[lightIndex].w;
	return data;
}
```

Now we can sample the shadow map in `GetOtherShadow` instead of always returning 1. It works like `GetCascadedShadow` except that there's no second cascade to blend with and it's a perspective projection, so we have to divide the XYZ components of the transformed position by its W component. Also, we don't have a functional normal bias yet so we'll multiply it with zero for now.

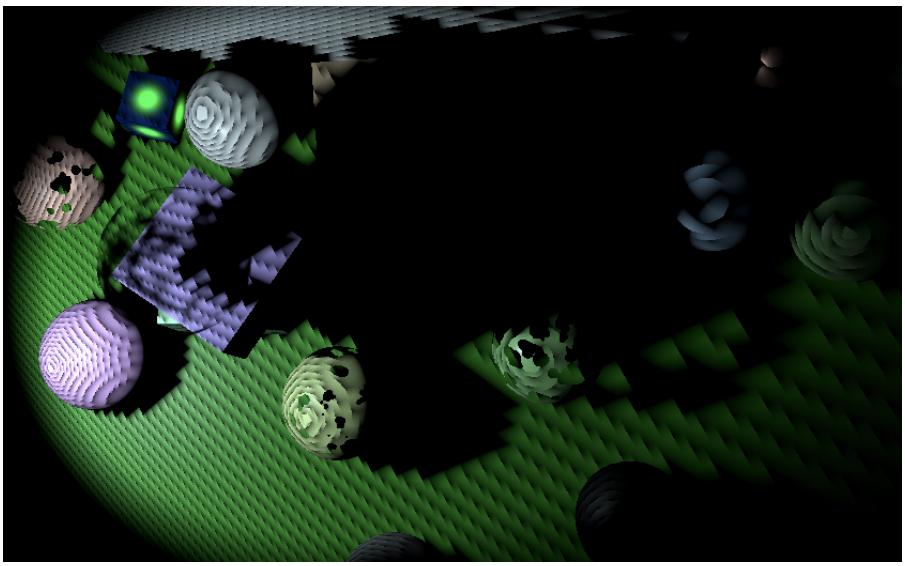
```
float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	float3 normalBias = surfaceWS.interpolatedNormal * 0.0;
	float4 positionSTS = mul(
		_OtherShadowMatrices[other.tileIndex],
		float4(surfaceWS.position + normalBias, 1.0)
	);
	return FilterOtherShadow(positionSTS.xyz / positionSTS.w);
}
```



Direct spot lighting only, with and without realtime shadows.

1.7 Normal Bias

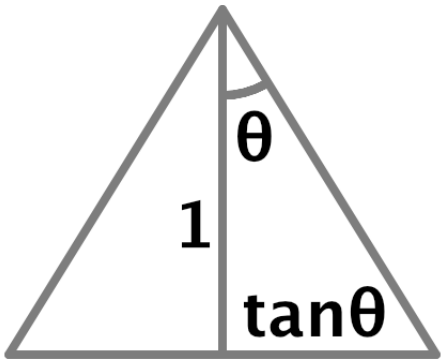
Spot lights suffer from shadow acne just like directional lights do. But because of the perspective projection the texel size isn't constant, so acne isn't constant either. The further away from the light, the bigger the acne.



Texel size increases with distance from light.

Texel size increases linearly with distance from the light plane, which is the plane that spits the world into what's in front of or behind the light. So we can calculate the texel size and thus the normal bias at distance 1 and send that to the shader, where we'll scale it to the appropriate size.

In world space, at distance 1 from the light plane the size of the shadow tile is twice the tangent of half the spot angle in radians.



World-space tile size derivation.

This matches the perspective projection, so the world-space texel size at distance 1 is equal to two divided by the projection scale, for which we can use the top left value of its matrix. We can use that to compute the normal bias the same way as we do for directional lights, except that we can factor the light's normal bias into it immediately as there aren't multiple cascades. Do this before setting the shadow matrix in `Shadows.RenderSpotShadows`.

```
		float texelSize = 2f / (tileSize * projectionMatrix.m00);
		float filterSize = texelSize * ((float)settings.other.filter + 1f);
		float bias = light.normalBias * filterSize * 1.4142136f;
		otherShadowMatrices[index] = ConvertToAtlasMatrix(
			projectionMatrix * viewMatrix,
			SetTileViewport(index, split, tileSize), tileScale
		);
```

Now we have to send the bias to the shader. We'll need to send some more data per tile later, so let's add an *_OtherShadowTiles* vector array shader property. Add an identifier and array for it to `Shadows` and set it in `RenderOtherShadows` along with the matrices.

```
	static int
		…
		otherShadowMatricesId = Shader.PropertyToID("_OtherShadowMatrices"),
		otherShadowTilesId = Shader.PropertyToID("_OtherShadowTiles"),
		…;

	static Vector4[]
		cascadeCullingSpheres = new Vector4[maxCascades],
		cascadeData = new Vector4[maxCascades],
		otherShadowTiles = new Vector4[maxShadowedOtherLightCount];

	…

	void RenderOtherShadows () {
		…

		buffer.SetGlobalMatrixArray(otherShadowMatricesId, otherShadowMatrices);
		buffer.SetGlobalVectorArray(otherShadowTilesId, otherShadowTiles);
		…
	}
```

Create a new `SetOtherTileData` method with an index and bias. Have it put the bias in the last component of a vector and then store it in the tile data array.

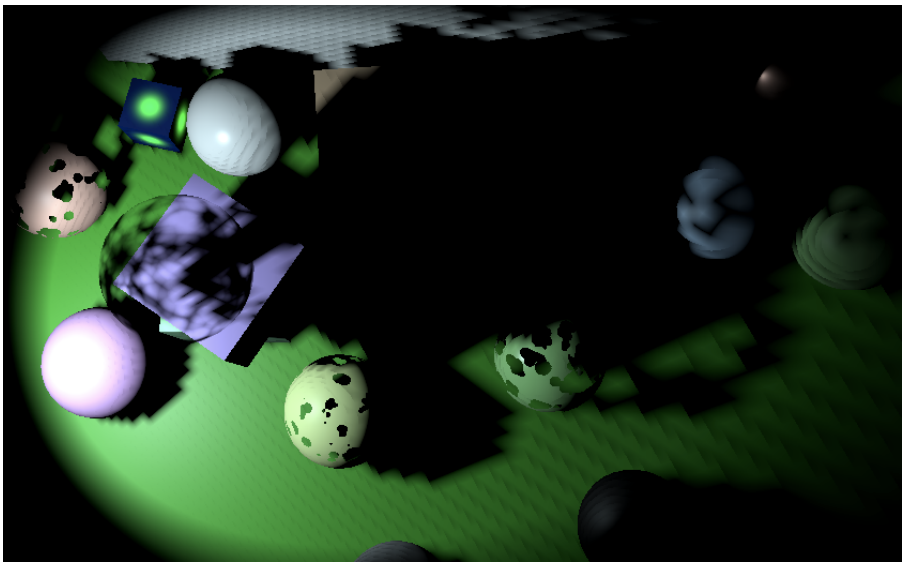
```
	void SetOtherTileData (int index, float bias) {
		Vector4 data = Vector4.zero;
		data.w = bias;
		otherShadowTiles[index] = data;
	}
```

Invoke it in `RenderSpotShadows` once we have the bias.

```
			float bias = light.normalBias * filterSize * 1.4142136f;
			SetOtherTileData(index, bias);
```

Then add the other shadow tile array to the shadow buffer and use it to scale the normal bias in *Shadows*.

```
CBUFFER_START(_CustomShadows)
	…
	float4x4 _OtherShadowMatrices[MAX_SHADOWED_OTHER_LIGHT_COUNT];
	float4 _OtherShadowTiles[MAX_SHADOWED_OTHER_LIGHT_COUNT];
	float4 _ShadowAtlasSize;
	float4 _ShadowDistanceFade;
CBUFFER_END

…

float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	float4 tileData = _OtherShadowTiles[other.tileIndex];
	float3 normalBias = surfaceWS.interpolatedNormal * tileData.w;
	…
}
```



Constant normal bias, set to 1.

At this point we have a normal bias that's only correct at at fixed distance. To scale it with distance from the light plane we need to know the world-space light position and spot direction, so add them to `OtherShadowData`.

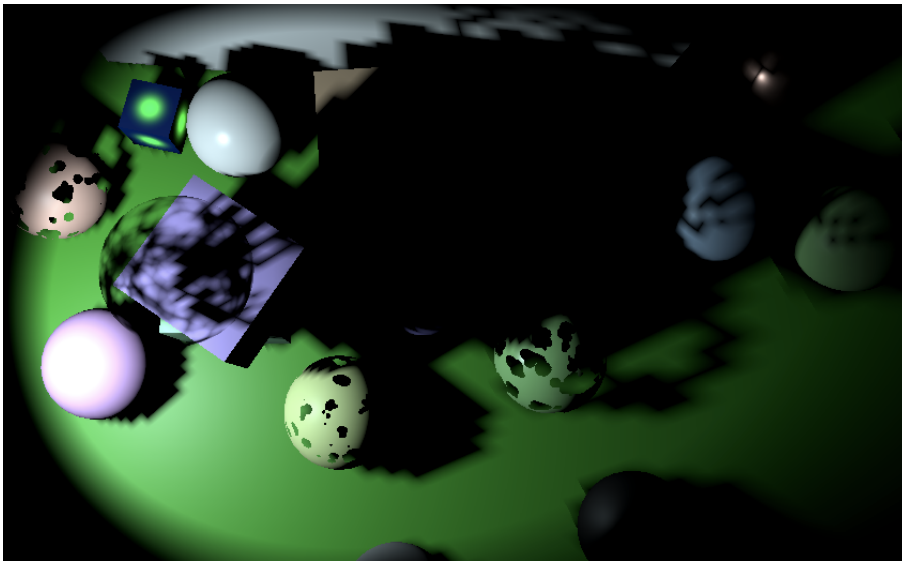
```
struct OtherShadowData {
	float strength;
	int tileIndex;
	int shadowMaskChannel;
	float3 lightPositionWS;
	float3 spotDirectionWS;
};
```

Have *Light* copy the values to it. As these values come from the light itself and not the shadow data set them to zero in `GetOtherShadowData` and copy them in `GetOtherLight`.

```
OtherShadowData GetOtherShadowData (int lightIndex) {
	…
	data.lightPositionWS = 0.0;
	data.spotDirectionWS = 0.0;
	return data;
}

Light GetOtherLight (int index, Surface surfaceWS, ShadowData shadowData) {
	Light light;
	light.color = _OtherLightColors[index].rgb;
	float3 position = _OtherLightPositions[index].xyz;
	float3 ray = position - surfaceWS.position;
	…
	float3 spotDirection = _OtherLightDirections[index].xyz;
	float spotAttenuation = Square(
		saturate(dot(spotDirection, light.direction) *
		spotAngles.x + spotAngles.y)
	);
	OtherShadowData otherShadowData = GetOtherShadowData(index);
	otherShadowData.lightPositionWS = position;
	otherShadowData.spotDirectionWS = spotDirection;
	…
}
```

We find the distance to the plane by taking the dot product of the surface-to-light vector and the spot direction in `GetOtherShadow`. Use it to scale the normal bias.

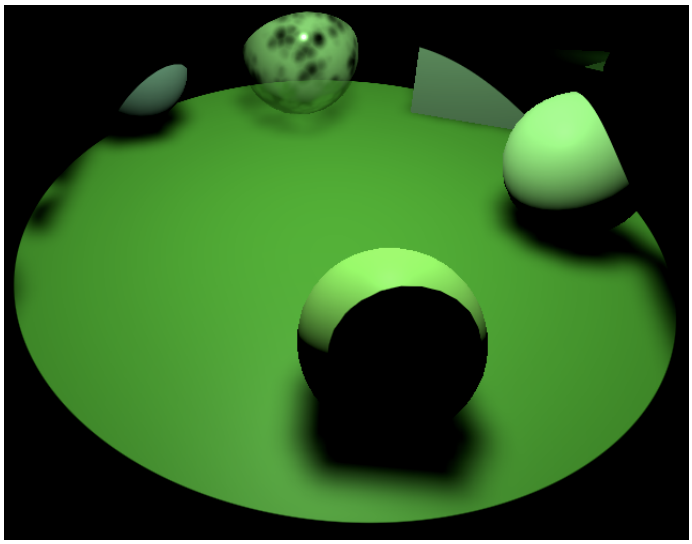
```
	float4 tileData = _OtherShadowTiles[other.tileIndex];
	float3 surfaceToLight = other.lightPositionWS - surfaceWS.position;
	float distanceToLightPlane = dot(surfaceToLight, other.spotDirectionWS);
	float3 normalBias =
		surfaceWS.interpolatedNormal * (distanceToLightPlane * tileData.w);
```



Correct normal bias everywhere.

1.8 Clamped Sampling

We configured the cascade spheres for directional shadows to ensure that we never end up sampling outside the appropriate shadow tile, but we cannot use the same approach for other shadows. In the case of spot lights their tiles are tightly fit to their cones, so the normal bias and filter size will push sampling outside the tile bounds where the cone edge approaches the tile edge.



Shadows from wrong tiles intrude near edges.

The simplest way to solve that is to manually clamp sampling to stay within the tile bounds, as if each tile were its own separate texture. That will still stretch shadows near the edges, but won't introduce invalid shadows.

Adjust the `SetOtherTileData` method so it also calculates and stores the tile bounds, based on an offset and scale provided via new parameters. The tile's minimum texture coordinates are the scaled offset, which we'll store in the XY components of the data vector. As tiles are square we can suffice with storing the tile's scale in the Z component, leaving W to the bias. We also have to shrink the bounds by half a texel in both dimensions to make sure sampling won't bleed beyond the edge.

```
	void SetOtherTileData (int index, Vector2 offset, float scale, float bias) {
		float border = atlasSizes.w * 0.5f;
		Vector4 data;
		data.x = offset.x * scale + border;
		data.y = offset.y * scale + border;
		data.z = scale - border - border;
		data.w = bias;
		otherShadowTiles[index] = data;
	}
```

In `RenderSpotShadows`, use the offset found via `SetTileViewport` and the inverse of the split for the new arguments of `SetOtherTileData`.

```
		Vector2 offset = SetTileViewport(index, split, tileSize);
		SetOtherTileData(index, offset, 1f / split, bias);
		otherShadowMatrices[index] = ConvertToAtlasMatrix(
			projectionMatrix * viewMatrix, offset, split
		);
```

The `ConverToAtlasMatrix` method also uses the inverse of the split, so we can calculate it once and pass that to both methods.

```
		float tileScale = 1f / split;
		SetOtherTileData(index, offset, tileScale);
		otherShadowMatrices[index] = ConvertToAtlasMatrix(
			projectionMatrix * viewMatrix, offset, tileScale
		);
```

Then `ConvertToAtlasMatrix` doesn't have to perform the division itself.

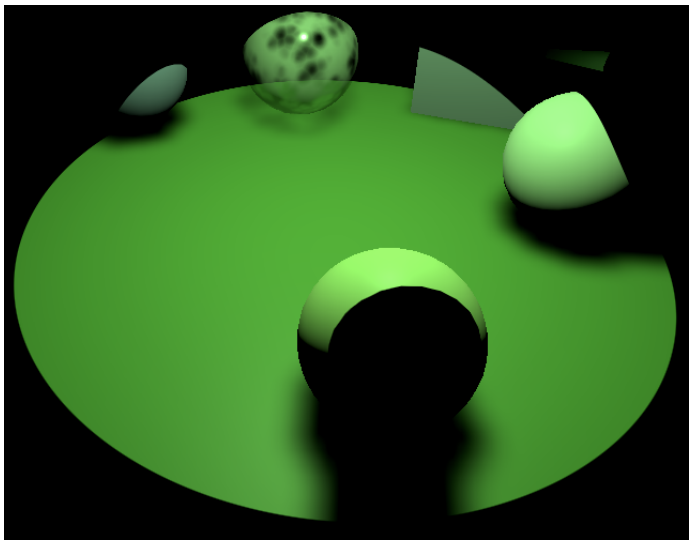
```
	Matrix4x4 ConvertToAtlasMatrix (Matrix4x4 m, Vector2 offset, float scale) {
		…
		//float scale = 1f / split;
		…
	}
```

This requires `RenderDirectionalShadows` to perform the division instead, which it only needs to do once for all cascades.

```
	void RenderDirectionalShadows (int index, int split, int tileSize) {
		…
		float tileScale = 1f / split;

		for (int i = 0; i < cascadeCount; i++) {
			…
			dirShadowMatrices[tileIndex] = ConvertToAtlasMatrix(
				projectionMatrix * viewMatrix,
				SetTileViewport(tileIndex, split, tileSize), tileScale
			);
			…
		}
	}
```

To apply the bounds add a `float3` parameter for it to `SampleOtherShadowAtlas` and use it to clamp the position in shadow tile space. `FilterOtherShadows` needs the same parameter so it can pass it along. And `GetOtherShadow` retrieves it from the tile data.

```
float SampleOtherShadowAtlas (float3 positionSTS, float3 bounds) {
	positionSTS.xy = clamp(positionSTS.xy, bounds.xy, bounds.xy + bounds.z);
	return SAMPLE_TEXTURE2D_SHADOW(
		_OtherShadowAtlas, SHADOW_SAMPLER, positionSTS
	);
}

float FilterOtherShadow (float3 positionSTS, float3 bounds) {
	#if defined(OTHER_FILTER_SETUP)
		…
		for (int i = 0; i < OTHER_FILTER_SAMPLES; i++) {
			shadow += weights[i] * SampleOtherShadowAtlas(
				float3(positions[i].xy, positionSTS.z), bounds
			);
		}
		return shadow;
	#else
		return SampleOtherShadowAtlas(positionSTS, bounds);
	#endif
}

float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	…
	return FilterOtherShadow(positionSTS.xyz / positionSTS.w, tileData.xyz);
}
```



No more shadows from wrong tiles.

2 Point Light Shadows

Shadows for point lights works like those for spot lights. The difference is that point lights aren't limited to a cone, so we need to render their shadows to a cube map. This is done by rendering shadows for all six faces of the cube separately. Thus we'll treat a point light as if it were six lights for the purpose of realtime shadows. It'll take up six tiles in the shadow atlas. This means that we can support realtime shadows for up to two point lights at the same time, as they would claim twelve of the sixteen available tiles. If there's fewer than six tiles free a point light cannot get realtime shadows.

2.1 Six Tiles for One Light

First, we need to know that we're dealing with a point light when rendering shadows, so add a boolean to indicate this to `ShadowedOtherLight`.

```
	struct ShadowedOtherLight {
		…
		public bool isPoint;
	}
```

Check whether we have a point light in `ReserveOtherShadows`. If so, the new light count with this one included would be six greater than the current count, otherwise it's just one greater. If that would exceed the max then the light can have baked shadows at best. If there's enough room in the atlas then also store whether it's a point light in the third component of the returned shadow data, to make it easy to detect point lights in the shader.

```
	public Vector4 ReserveOtherShadows (Light light, int visibleLightIndex) {
		…

		bool isPoint = light.type == LightType.Point;
		int newLightCount = shadowedOtherLightCount + (isPoint ? 6 : 1);
		if (
			newLightCount > maxShadowedOtherLightCount ||
			!cullingResults.GetShadowCasterBounds(visibleLightIndex, out Bounds b)
		) {
			return new Vector4(-light.shadowStrength, 0f, 0f, maskChannel);
		}

		shadowedOtherLights[shadowedOtherLightCount] = new ShadowedOtherLight {
			visibleLightIndex = visibleLightIndex,
			slopeScaleBias = light.shadowBias,
			normalBias = light.shadowNormalBias,
			isPoint = isPoint
		};

		Vector4 data = new Vector4(
			light.shadowStrength, shadowedOtherLightCount,
			isPoint ? 1f : 0f, maskChannel
		);
		shadowedOtherLightCount = newLightCount;
		return data;
	}
```

2.2 Rendering Point Shadows

Adjust `RenderOtherShadows` so it invokes either a new `RenderPointShadows` method or the existing `RenderSpotShadows` method in its loop, as appropriate. Also, as point lights count for six increase the iterator by the correct amount for each light type, instead of just incrementing it.

```
		for (int i = 0; i < shadowedOtherLightCount;) { //i++) {
			if (shadowedOtherLights[i].isPoint) {
				RenderPointShadows(i, split, tileSize);
				i += 6;
			}
			else {
				RenderSpotShadows(i, split, tileSize);
				i += 1;
			}
		}
```

The new `RenderPointShadows` method is a duplicate of `RenderSpotShadows`, with two differences. First, it has to render six times instead of just once, looping through its six tiles. Second, it has to use `ComputePointShadowMatricesAndCullingPrimitives` instead of `ComputeSpotShadowMatricesAndCullingPrimitives`. This method requires two extra arguments after the light index: a `CubemapFace` index and a bias. We render once for each face and leave the bias at zero for now.

```
	void RenderPointShadows (int index, int split, int tileSize) {
		ShadowedOtherLight light = shadowedOtherLights[index];
		var shadowSettings = new ShadowDrawingSettings(
			cullingResults, light.visibleLightIndex,
			BatchCullingProjectionType.Perspective
		);
		for (int i = 0; i < 6; i++) {
			cullingResults.ComputePointShadowMatricesAndCullingPrimitives(
				light.visibleLightIndex, (CubemapFace)i, 0f,
				out Matrix4x4 viewMatrix, out Matrix4x4 projectionMatrix,
				out ShadowSplitData splitData
			);
			shadowSettings.splitData = splitData;
			int tileIndex = index + i;
			float texelSize = 2f / (tileSize * projectionMatrix.m00);
			float filterSize = texelSize * ((float)settings.other.filter + 1f);
			float bias = light.normalBias * filterSize * 1.4142136f;
			Vector2 offset = SetTileViewport(tileIndex, split, tileSize);
			float tileScale = 1f / split;
			SetOtherTileData(tileIndex, offset, tileScale, bias);
			otherShadowMatrices[tileIndex] = ConvertToAtlasMatrix(
				projectionMatrix * viewMatrix, offset, tileScale
			);

			buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
			buffer.SetGlobalDepthBias(0f, light.slopeScaleBias);
			ExecuteBuffer();
			context.DrawShadows(ref shadowSettings);
			buffer.SetGlobalDepthBias(0f, 0f);
		}
	}
```

Shadow atlas for two point lights.

The field of view for cubemap faces is always 90°, thus the world-space tile size at distance 1 is always 2. This means that we can hoist the calculation of the bias out of the loop. We can also do that with the tile scale.

```
		float texelSize = 2f / tileSize;
		float filterSize = texelSize * ((float)settings.other.filter + 1f);
		float bias = light.normalBias * filterSize * 1.4142136f;
		float tileScale = 1f / split;

		for (int i = 0; i < 6; i++) {
			…
			//float texelSize = 2f / (tileSize * projectionMatrix.m00);
			//float filterSize = texelSize * ((float)settings.other.filter + 1f);
			//float bias = light.normalBias * filterSize * 1.4142136f;
			Vector2 offset = SetTileViewport(tileIndex, split, tileSize);
			//float tileScale = 1f / split;
			…
		}
```

2.3 Sampling Point Shadows

The idea is that point light shadows are stored in a cube map, which our shader samples. However, we store the cube map faces as tiles in an atlas, so we cannot use standard cube map sampling. We have to determine the appropriate face to sample from ourselves. To do this we need to know whether we're dealing with a point light as well as the surface-to-light direction. Add both to `OtherShadowData`.

```
struct OtherShadowData {
	float strength;
	int tileIndex;
	bool isPoint;
	int shadowMaskChannel;
	float3 lightPositionWS;
	float3 lightDirectionWS;
	float3 spotDirectionWS;
};
```

Set both values in *Light*. It's a point light if the third component of the other light's shadow data equals 1.

```
OtherShadowData GetOtherShadowData (int lightIndex) {
	…
	data.isPoint = _OtherLightShadowData[lightIndex].z == 1.0;
	data.lightPositionWS = 0.0;
	data.lightDirectionWS = 0.0;
	data.spotDirectionWS = 0.0;
	return data;
}

Light GetOtherLight (int index, Surface surfaceWS, ShadowData shadowData) {
	…
	otherShadowData.lightPositionWS = position;
	otherShadowData.lightDirectionWS = light.direction;
	otherShadowData.spotDirectionWS = spotDirection;
	…
}
```

Next, we have to adjust the tile index and light plane in `GetOtherShadow` in case of a point light. Begin by turning them into variables, initially configured for spot lights. Make the tile index a `float` because we'll add an offset to it that's also defined as a `float`.

```
float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	float tileIndex = other.tileIndex;
	float3 lightPlane = other.spotDirectionWS;
	float4 tileData = _OtherShadowTiles[tileIndex];
	float3 surfaceToLight = other.lightPositionWS - surfaceWS.position;
	float distanceToLightPlane = dot(surfaceToLight, lightPlane);
	float3 normalBias =
		surfaceWS.interpolatedNormal * (distanceToLightPlane * tileData.w);
	float4 positionSTS = mul(
		_OtherShadowMatrices[tileIndex],
		float4(surfaceWS.position + normalBias, 1.0)
	);
	return FilterOtherShadow(positionSTS.xyz / positionSTS.w, tileData.xyz);
}
```

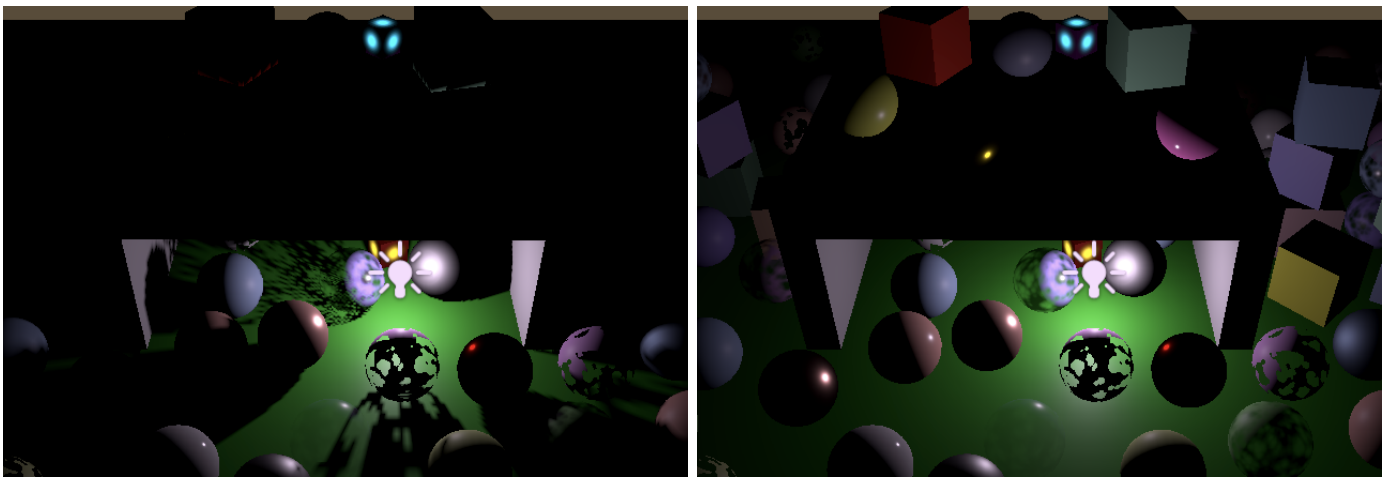
If we have a point light then we must use the appropriate axis-aligned plane instead. We can use the `CubeMapFaceID` function to find the face offset, by passing it the negated light direction. This function either intrinsic or defined in the *Core RP Library*, returning a `float`. The order of the cubemap faces is +X, −X, +Y, −Y, +Z, −Z, which matches how we rendered them. Add the offset to the tile index.

```
float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	float tileIndex = other.tileIndex;
	float3 lightPlane = other.spotDirectionWS;
	if (other.isPoint) {
		float faceOffset = CubeMapFaceID(-other.lightDirectionWS);
		tileIndex += faceOffset;
	}
	…
}
```

```
	if (other.isPoint) {

		plane = pointShadowPlanes[CubeMapFaceID(-other.lightDirectionWS)];
	}
```

Next, we need to use a light plane that matches the face orientation. Create a static constant array for them and use the face offset to index it. The plane normals have to point in the opposite direction as the faces, like the spot direction points toward the light.

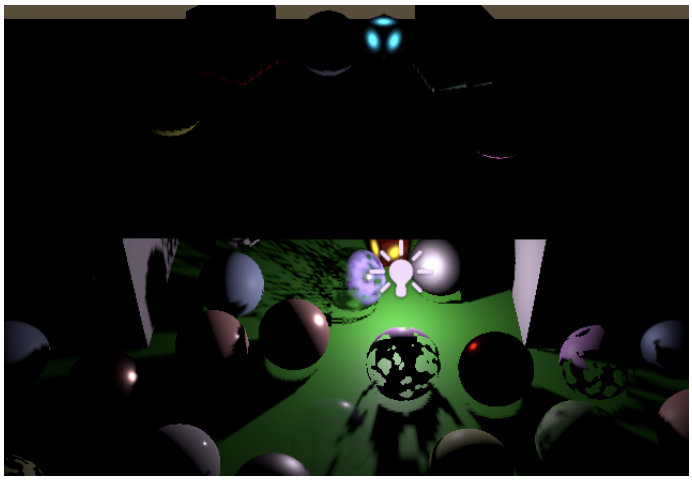
```
static const float3 pointShadowPlanes[6] = {
	float3(-1.0, 0.0, 0.0),
	float3(1.0, 0.0, 0.0),
	float3(0.0, -1.0, 0.0),
	float3(0.0, 1.0, 0.0),
	float3(0.0, 0.0, -1.0),
	float3(0.0, 0.0, 1.0)
};

float GetOtherShadow (
	OtherShadowData other, ShadowData global, Surface surfaceWS
) {
	float tileIndex = other.tileIndex;
	float3 plane = other.spotDirectionWS;
	if (other.isPoint) {
		float faceOffset = CubeMapFaceID(-other.lightDirectionWS);
		tileIndex += faceOffset;
		lightPlane = pointShadowPlanes[faceOffset];
	}
	…
}
```



Direct point light only, with and without realtime shadows; no biases.

2.4 Drawing the Correct Faces

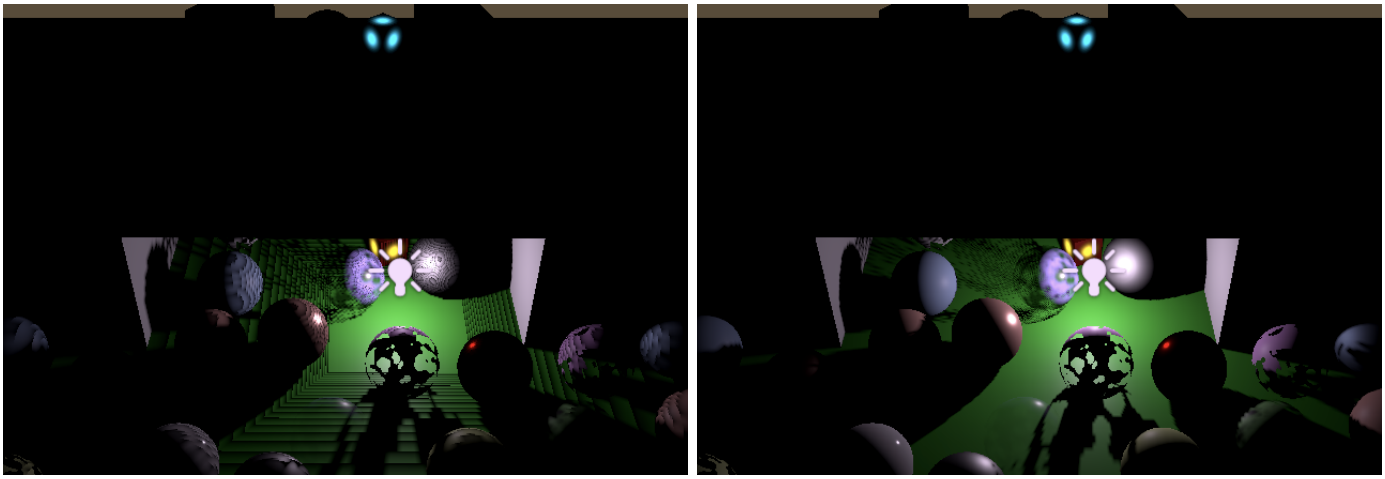
We can now see realtime shadows for point lights. They don't appear to suffer from shadow acne, even with zero bias. Unfortunately, light now leaks through objects to surfaces very close to them on the opposite side. Increasing the shadow bias makes this worse and also appears to cut holes in the shadows of objects close to other surfaces.



Maximum normal bias 3.

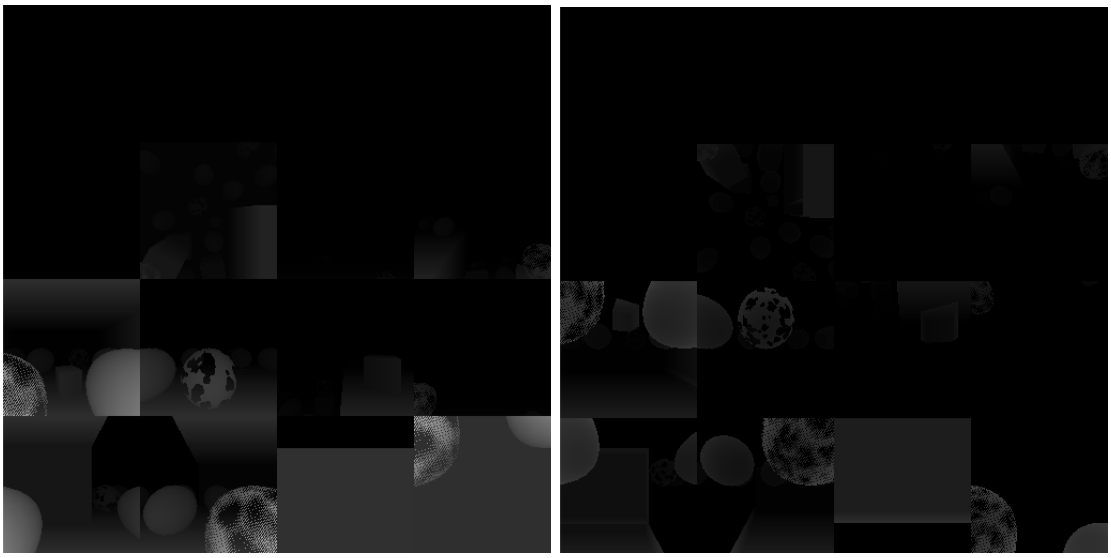
This happens because of the way Unity renders shadows for point lights. It draws them upside down, which reverses the winding order of triangles. Normally the front faces—from the point of view of the light—are drawn, but now the back faces get rendered. This prevents most acne but introduces light leaking. We cannot stop the flipping, but we can undo it by negating a row of the view matrix that we get from `ComputePointShadowMatricesAndCullingPrimitives`. Let's negate its second row. This flips everything upside down in the atlas at second time, which turns everything back to normal. Because the first component of that row is always zero we can suffice with only negating the other three components.

```
		cullingResults.ComputePointShadowMatricesAndCullingPrimitives(
			light.visibleLightIndex, (CubemapFace)i, fovBias*0,
			out Matrix4x4 viewMatrix, out Matrix4x4 projectionMatrix,
			out ShadowSplitData splitData
		);
		viewMatrix.m11 = -viewMatrix.m11;
		viewMatrix.m12 = -viewMatrix.m12;
		viewMatrix.m13 = -viewMatrix.m13;
```



Front-face shadow rendering, normal bias 0 and 1.

How this changes the rendered shadows is most obvious when comparing the shadow maps.



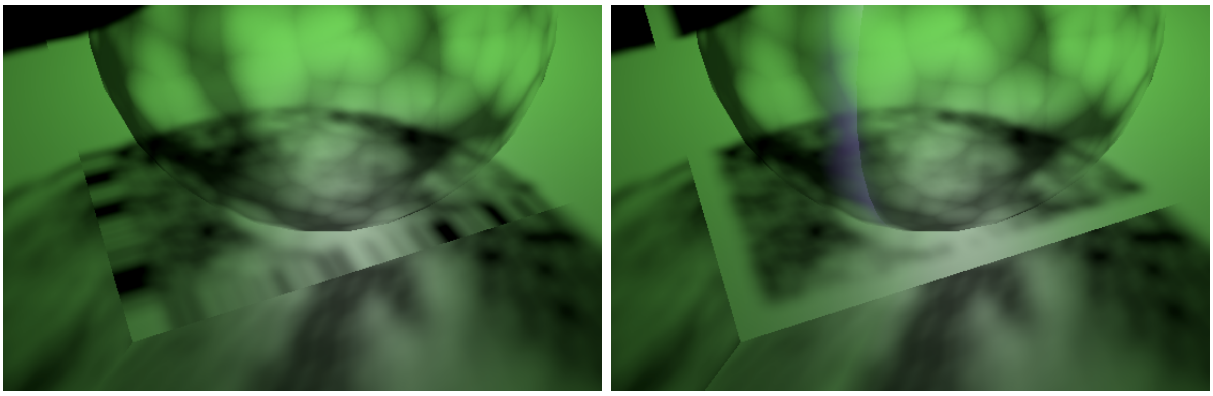
Front and back versions of the shadow map.

Note that objects that have the *Cast Shadows* mode of their `MeshRenderer` set to *Two Sided* aren't affected, because none of their faces are culled. For example, I've made all spheres with a clip or transparent material cast two-sided shadows, so they appear more solid.

Clip and transparent spheres with two-sided shadows.

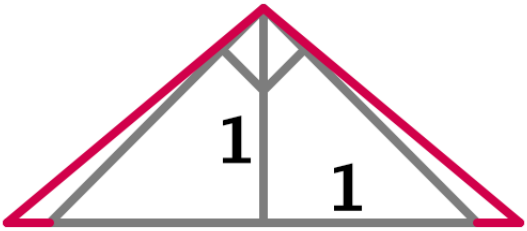
2.5 Field of View Bias

There's always a discontinuity between faces of a cube map, because the orientation of the texture plane suddenly changes 90°. Regular cubemap sampling can hide this somewhat because it can interpolate between faces, but we're sampling from a single tile per fragment. We get the same issues that exist at the edge of spot shadow tiles, but now they aren't hidden because there's no spot attenuation.



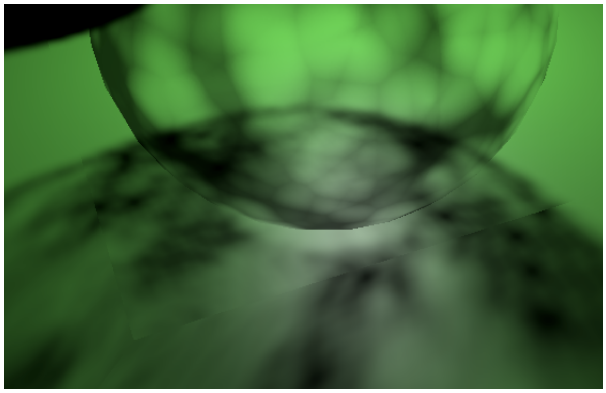
Discontinuities between faces, with and without tile clamping.

We can reduce these artifacts by increasing the field of view—FOV for short—a little when rendering the shadows, so we never sample beyond the edge of a tile. That's what the bias argument of `ComputePointShadowMatricesAndCullingPrimitives` is for. We do that by making our tile size a bit larger than 2 at distance 1 from the light. Specifically, we add the normal bias plus the filter size on each side. The tangent of the half corresponding FOV angle is then equal to 1 plus the bias and filter size. Double that, convert it to degrees, subtract 90°, and use it for the FOV bias in `RenderPointShadows`.



Increasing the world-space tile size.

```
	float fovBias =
		Mathf.Atan(1f + bias + filterSize) * Mathf.Rad2Deg * 2f - 90f;
	for (int i = 0; i < 6; i++) {
		cullingResults.ComputePointShadowMatricesAndCullingPrimitives(
			light.visibleLightIndex, (CubemapFace)i, fovBias,
			out Matrix4x4 viewMatrix, out Matrix4x4 projectionMatrix,
			out ShadowSplitData splitData
		);
		…
	}
```



With FOV bias.

Note that this approach isn't perfect, because by increasing the tile size the texel size increases as well. Thus the filter size increases and the normal bias should also increase, which means that we must increase the FOV again. However, the difference is usually small enough that we can get away with ignoring the tile size increase, unless a large normal bias and filter is used in combination with a small atlas size.

Could we use the same approach for spot lights?

We could, which would make tile clamping no longer necessary with a little extra work. However, `ComputeSpotShadowMatricesAndCullingPrimitives` doesn't have a FOV bias parameter, so we'd have to create our own variant of it, which is out the scope of this tutorial.

The next tutorial is Post Processing.

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick