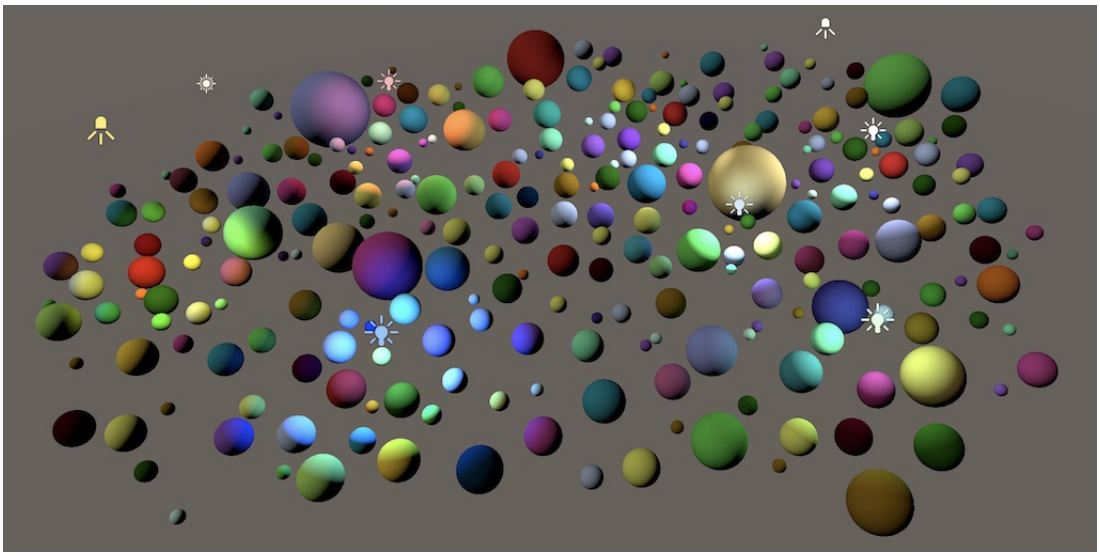
Lights Single-Pass Forward Rendering

Perform diffuse shading.
Support directional, point, and spotlights.
Allow up to sixteen visible lights per frame.
Calculate up to four pixel lights and four vertex lights per object.

This is the third installment of a tutorial series covering Unity's scriptable render pipeline. This time we'll add support for diffuse lighting, by shading up to eight lights per object with a single draw call.

This tutorial is made with Unity 2018.3.0f2.



256 spheres, 8 lights, 214 draw calls.

1 Shading With a Light

To support lights, we have to add a lit shader to our pipeline. Lighting complexity can go from very simple—only including diffuse light—to very complex—full-blown physically-based shading. It can also be unrealistic, like cell-shading. We'll start with the minimum of a lit shader that calculates diffuse directional lighting, without shadows.

1.1 Lit Shader

Duplicate *Unlit.hlsl* and rename it to *Lit.hlsl*. Replace all instances of *unlit* in the new file with *lit*, specifically the include define and the vertex and fragment function names.

```
#ifndef MYRP_LIT_INCLUDED
#define MYRP_LIT_INCLUDED

…

VertexOutput LitPassVertex (VertexInput input) {
	…
}

float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	…
}

#endif // MYRP_LIT_INCLUDED
```

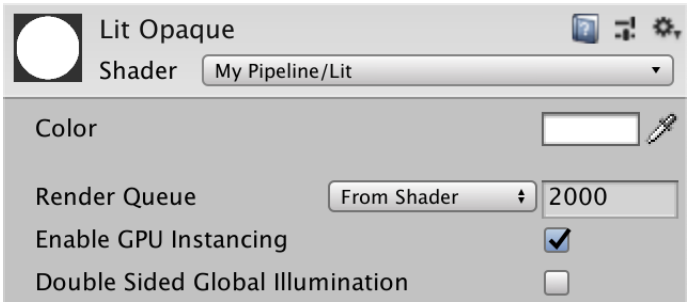
Also duplicate *Unlit.shader* and rename it to *Lit.shader*, again replacing *unlit* with *lit* in the new file.

```
Shader "My Pipeline/Lit" {

	Properties {
		_Color ("Color", Color) = (1, 1, 1, 1)
	}

	SubShader {

		Pass {
			HLSLPROGRAM

			#pragma target 3.5

			#pragma multi_compile_instancing
			#pragma instancing_options assumeuniformscaling

			#pragma vertex LitPassVertex
			#pragma fragment LitPassFragment

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

			ENDHLSL
		}
	}
}
```

Shouldn't we explicitly use a lit pass?

Because our pipeline is still very basic, we won't bother with dedicated passes yet.

Now we can create a lit opaque material with the new shader, although it still does exactly the same as the unlit variant.



Unlit shader asset.

1.2 Normal Vectors

In order to calculate the contribution of a directional light, we need to know the surface normal. So we have to add a normal vector to both the vertex input and output structures. For a detailed description of how the lighting is calculated, see Rendering 4, The First Light.

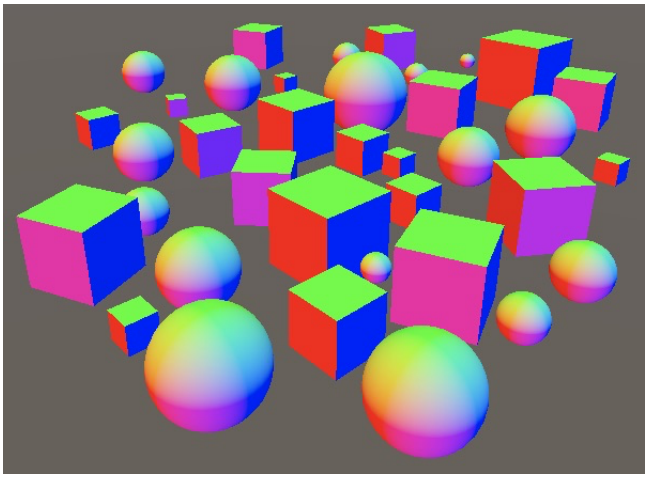
```
struct VertexInput {
	float4 pos : POSITION;
	float3 normal : NORMAL;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
	float3 normal : TEXCOORD0;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Convert the normal from object space to world space in `LitPassVertex`. As we assume that we're only using uniform scales, we can simply use the 3×3 part of the model matrix, followed by normalization per fragment in `LitPassFragment`. Support for nonuniform scales would require us to use a transposed world-to-object matrix instead.

```
VertexOutput LitPassVertex (VertexInput input) {
	…
	output.normal = mul((float3x3)UNITY_MATRIX_M, input.normal);
	return output;
}

float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	input.normal = normalize(input.normal);
	…
}
```

To verify that we end up with correct normal vectors, use them for the final color. But still keep track of the material's, as we'll use that for its albedo later.

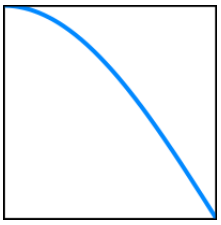
```
float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	input.normal = normalize(input.normal);
	float3 albedo = UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color).rgb;

	float3 color = input.normal;
	return float4(color, 1);
}
```



Showing raw world-space normal vectors.

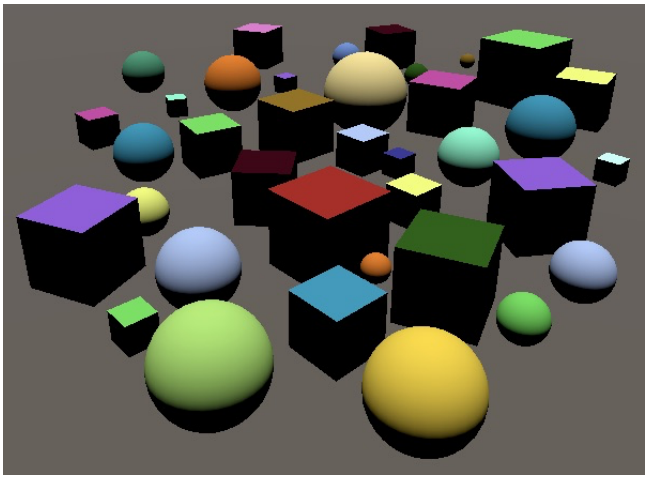
1.3 Diffuse Light

The diffuse light contribution depends on the angle at which light hits the surface, which is found by computing the dot product of the surface normal and the direction where the light is coming from, discarding negative results. In the case of a directional light, the light vector is constant. Let's use a hard-coded direction for now, pointing straight up. Multiply the diffuse light with the albedo to get the final color.



Diffuse falloff from 0° to 90° angle of incoming light.

```
float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	input.normal = normalize(input.normal);
	float3 albedo = UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color).rgb;

	float3 diffuseLight = saturate(dot(input.normal, float3(0, 1, 0)));
	float3 color = diffuseLight * albedo;
	return float4(color, 1);
}
```



Diffuse light from above.

2 Visible Lights

To be able to use lights defined in the scene, our pipeline has to send the light data to the GPU. It is possible to have multiple lights in a scene, so we should support multiple lights too. There are multiple ways to do that. Unity's default pipeline renders each light in a separate pass, per object. The Lightweight pipeline renders all lights in a single pass, per object. And the HD pipeline uses deferred rendering, which renders the surface data of all objects, followed by one pass per light.

We're going to use the same approach as the Lightweight pipeline, so each objects is rendered once, taking all lights into consideration. We do that by sending the data of all lights that are currently visible to the GPU. Lights that are in the scene but don't affect anything that is going to be rendered will be ignored.

2.1 Light Buffer

Rendering all lights in one pass means that all lighting data must be available at the same time. Limiting ourselves to directional lights only for now, that means we need to know both the color and the direction of each light. To support an arbitrary amount of lights, we'll use arrays to store this data, which we'll put in a separate buffer that we'll name `_LightBuffer`. Arrays are defined in shaders like in C#, except the brackets come after the variable name instead of the type.

```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
CBUFFER_END

CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[];
	float4 _VisibleLightDirections[];
CBUFFER_END
```

However, we cannot define arrays of arbitrary size. The array definition must immediately declare its size. Let's use an array length of 4. That means that we can support up to four visible lights at once. Define this limit with a macro for easy reference.

```
#define MAX_VISIBLE_LIGHTS 4

CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightDirections[MAX_VISIBLE_LIGHTS];
CBUFFER_END
```

Below the light buffer, add a `DiffuseLight` function that uses the light data to take care of the lighting calculation. It needs a light index and normal vector as parameters, extracts the relevant data from the arrays, then performs the diffuse lighting calculation and returns it, modulated by the light's color.

```
CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightDirections[MAX_VISIBLE_LIGHTS];
CBUFFER_END

float3 DiffuseLight (int index, float3 normal) {
	float3 lightColor = _VisibleLightColors[index].rgb;
	float3 lightDirection = _VisibleLightDirections[index].xyz;
	float diffuse = saturate(dot(normal, lightDirection));
	return diffuse * lightColor;
}
```

In `LitPassFragment`, use a `for` loop to invoke the new function once per light, accumulating the total diffuse light affecting the fragment.

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

	float3 diffuseLight = 0;
	for (int i = 0; i < MAX_VISIBLE_LIGHTS; i++) {
		diffuseLight += DiffuseLight(i, input.normal);
	}
	float3 color = diffuseLight * albedo;
	return float4(color, 1);
}
```

Note that even through we use a loop, the shader compiler will likely unroll it. As our shader becomes more complex, at some point the compiler will switch to using an actual loop.

2.2 Filling the Buffer

At the moment we end up with fully black shapes, because we aren't passing any light data to the GPU yet. We have to add the same arrays to `MyPipeline`, with the same size. Also, use the static `Shader.PropertyToID` method to find the identifiers of the relevant shader properties. The shader IDs are constant per session, so can be stored in static variables.

```
	const int maxVisibleLights = 4;

	static int visibleLightColorsId =
		Shader.PropertyToID("_VisibleLightColors");
	static int visibleLightDirectionsId =
		Shader.PropertyToID("_VisibleLightDirections");

	Vector4[] visibleLightColors = new Vector4[maxVisibleLights];
	Vector4[] visibleLightDirections = new Vector4[maxVisibleLights];
```

> **Why not use a `Color` array?**
>
> There is no way to directly pass a color array to the GPU. A `Vector4` array is the best alternative and matches the data format of the shader. We can directly assign colors to the array, as there is an implicit conversion from `Color` to `Vector4`.

The arrays can be copied to the GPU via invoking the `SetGlobalVectorArray` method on a command buffer, and then executing it. As we already have `cameraBuffer`, lets use that buffer, at the same moment that we start the *Render Camera* sample.

```
		cameraBuffer.BeginSample("Render Camera");
		cameraBuffer.SetGlobalVectorArray(
			visibleLightColorsId, visibleLightColors
		);
		cameraBuffer.SetGlobalVectorArray(
			visibleLightDirectionsId, visibleLightDirections
		);
		context.ExecuteCommandBuffer(cameraBuffer);
		cameraBuffer.Clear();
```

2.3 Configuring the Lights

We're now sending light data to the GPU each frame, but it's still the default data, so the objects remain black. We have to configure the lights before copying the vectors. Let's delegate that responsibility to a new `ConfigureLights` method.

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

		ConfigureLights();

		cameraBuffer.BeginSample("Render Camera");
```

During culling, Unity also figures out which lights are visible. This information is made available via a `visibleLights` list that is part of the cull results. The list's elements are `VisibleLight` structs that contain all the data that we need. Create the required `ConfigureLights` method and have it loop through the list.

```
	void ConfigureLights () {
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			VisibleLight light = cull.visibleLights[i];
		}
	}
```

The `VisibleLight.finalColor` field holds the light's color. It is the light's color multiplied by its intensity, and also converted to the correct color space. So we can directly copy it to `visibleLightColors`, at the same index.

```
			VisibleLight light = cull.visibleLights[i];
			visibleLightColors[i] = light.finalColor;
```

However, by default Unity considers the light's intensity to be defined in gamma space, even through we're working in linear space. This is a holdover of Unity's default render pipeline; the new pipelines consider it a linear value. This behavior is controlled via the boolean `GraphicsSettings.lightsUseLinearIntensity` property. It is a project setting, but can only be adjusted via code. We only need to set it once, so let's do that in the constructor method of `MyPipeline`.

```
	public MyPipeline (bool dynamicBatching, bool instancing) {
		GraphicsSettings.lightsUseLinearIntensity = true;
		…
	}
```

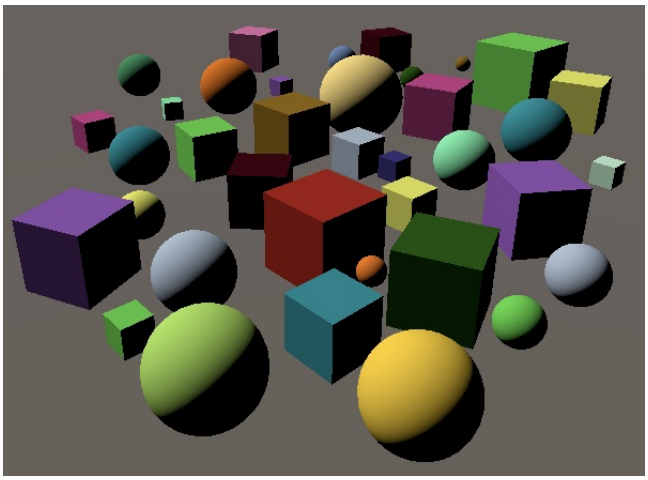
Changing this setting only affects the editor when it re-applies its graphics settings, which doesn't automatically happen. Entering and exiting play mode will apply it.

Besides that, the direction of a directional light is determined by its rotation. The light shines along its local Z axis. We can find this vector in world space via the `VisibleLight.localtoWorld` matrix field. The third column of that matrix defines the transformed local Z direction vector, which we can get via the `Matrix4x4.GetColumn` method, with index 2 as an argument.

That gives us the direction in which the light is shining, but in the shader we use the direction from the surface toward the light source. So we have to negate the vector before we assign it to `visibleLightDirections`. As the fourth component of a direction vector is always zero, we only have to negate X, Y, and Z.

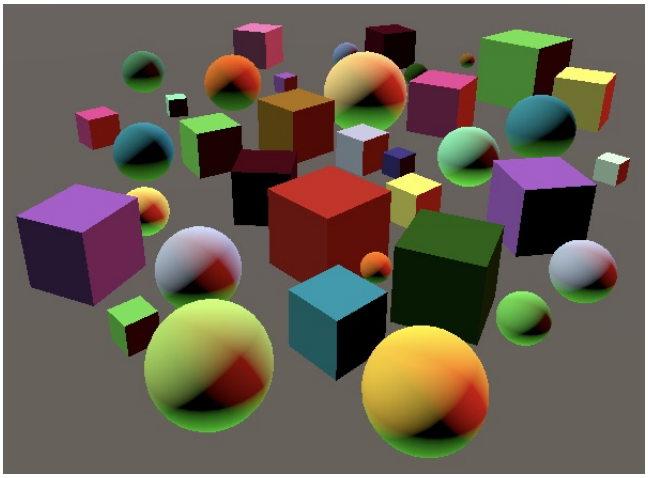
```
			VisibleLight light = cull.visibleLights[i];
			visibleLightColors[i] = light.finalColor;
			Vector4 v = light.localToWorld.GetColumn(2);
			v.x = -v.x;
			v.y = -v.y;
			v.z = -v.z;
			visibleLightDirections[i] = v;
```

Our objects are now shaded with the color and direction of the main direction light, assuming you have no other lights in the scene. If you don't have a light source in the scene, just add a single directional light.



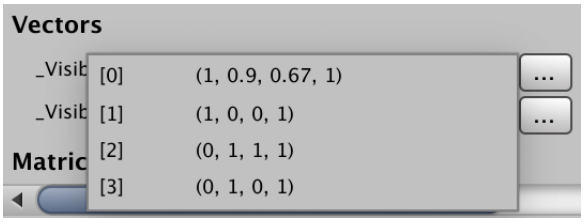
Diffuse shading with one directional light.

But our shader always calculates the lighting contribution of four lights, even if we only have a single light in the scene. So you could add three more directional lights and it wouldn't slow down the GPU.



Four directional lights.

You can inspect the light data that is sent to the GPU via the frame debugger. Select one of the draw calls that uses our shader, then expand the vector arrays to see their contents.



Light colors found via the frame debugger.

2.4 Varying the Number of Lights

Everything works as expected when using exactly four directional lights. We can even have more, as long as only four are visible at the same time. But when there are more than four visible lights our pipeline fails with an index-out-of-bounds exception. We can only support up to four visible lights, but Unity doesn't take that into consideration while culling. So `visibleLights` can end up with more elements than our arrays. We have to abort the loop when we exceed the maximum. That means that we simply ignore some of the visible lights.

```
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			if (i == maxVisibleLights) {
				break;
			}
			VisibleLight light = cull.visibleLights[i];
			…
		}
```

Which lights get omitted?

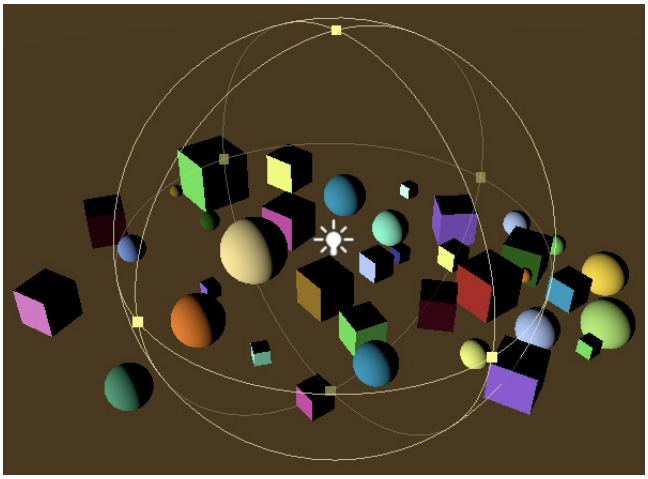
We simple skip the last lights in the `visibleLights` list. The lights are ordered based on various criteria, including light type, intensity, and whether they have shadows enabled. You can assume that the lights are ordered from most to least important. For example, the directional light with the highest intensity and shadows enabled will be the first element.

Another weird thing happens when the amount of visible lights decreases. They remain visible, because we don't reset their data. We can solve that by continuing to loop through our arrays after finishing the visible lights, clearing the color of all lights that aren't used.

```
		int i = 0;
		for (; i < cull.visibleLights.Count; i++) {
			…
		}
		for (; i < maxVisibleLights; i++) {
			visibleLightColors[i] = Color.clear;
		}
```

3 Point Lights

We currently only support directional lights, but typically a scene has only a single directional light plus additional point lights. While we can add points lights to the scene, they are currently interpreted as a directional lights. We're going to fix that now.



Point light interpreted as directional.

Rendering 5, Multiple Lights describes point lights and spotlights, but uses the old approach of Unity's default pipeline. We're going to use the same approach as the Lightweight pipeline.

3.1 Light Position

Unlike a directional light, the position of a point light matters. Rather than adding a separate array for positions, we'll store both direction and position data in the same array, each element containing either a direction or a position. Rename the variables in `MyPipeline` accordingly.

```
	static int visibleLightColorsId =
		Shader.PropertyToID("_VisibleLightColors");
	static int visibleLightDirectionsOrPositionsId =
		Shader.PropertyToID("_VisibleLightDirectionsOrPositions");

	Vector4[] visibleLightColors = new Vector4[maxVisibleLights];
	Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];
```

`ConfigureLights` can use `VisibleLight.lightType` to check the type of each light. In case of a direction light, storing the direction is correct. Otherwise, store the light's world position instead, which is can be extracted from the fourth column of its local-to-world matrix.

```
			if (light.lightType == LightType.Directional) {
				Vector4 v = light.localToWorld.GetColumn(2);
				v.x = -v.x;
				v.y = -v.y;
				v.z = -v.z;
				visibleLightDirectionsOrPositions[i] = v;
			}
			else {
				visibleLightDirectionsOrPositions[i] =
					light.localToWorld.GetColumn(3);
			}
```

Rename the array in the shader too. In `DiffuseLight`, begin by assuming that we're still always dealing with a directional light.

```
CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightDirectionsOrPositions[MAX_VISIBLE_LIGHTS];
CBUFFER_END

float3 DiffuseLight (int index, float3 normal) {
	float3 lightColor = _VisibleLightColors[index].rgb;
	float4 lightPositionOrDirection = _VisibleLightDirectionsOrPositions[index];
	float3 lightDirection = lightPositionOrDirection.xyz;
	float diffuse = saturate(dot(normal, lightDirection));
	return diffuse * lightColor;
}
```

But if we are dealing with a point light, we have to calculate the light direction ourselves. First, we subtract the surface position from the light position, which requires us to add an additional parameter to the function. That gives us the light vector, in world space, which we turn into a direction by normalizing it.

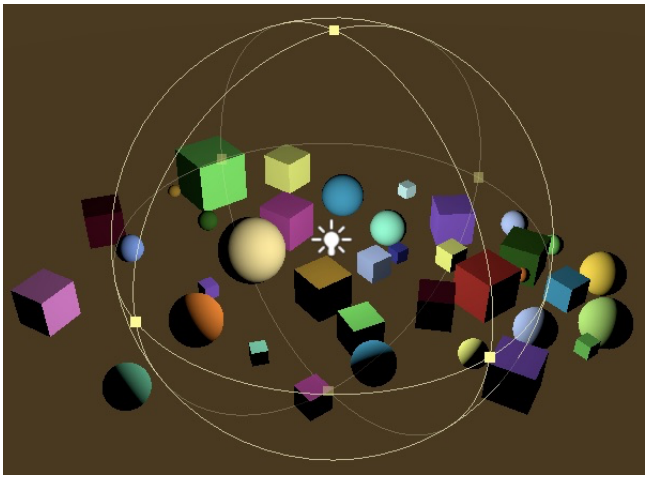
```
float3 DiffuseLight (int index, float3 normal, float3 worldPos) {
	float3 lightColor = _VisibleLightColors[index].rgb;
	float4 lightPositionOrDirection = _VisibleLightDirectionsOrPositions[index];
	float3 lightVector =
		lightPositionOrDirection.xyz - worldPos;
	float3 lightDirection = normalize(lightVector);
	float diffuse = saturate(dot(normal, lightDirection));
	return diffuse * lightColor;
}
```

That works for point lights, but is nonsensical for directional lights. We can support both with the same calculation, by multiplying the world position with the W component of the light's direction or position vector. If it's a position vector, then W is 1 and the calculation is unchanged. But if it's a direction vector, then W is 0 and the subtraction is eliminated. So we end up normalizing the original direction vector, which makes no difference. It does introduce an unneeded normalization for directional lights, but branching to avoid that isn't worth it.

```
		lightPositionOrDirection.xyz - worldPos * lightPositionOrDirection.w;
```

To make this work we need to know the fragment's world-space position in `LitPassFragment`. We already have it in `LitPassVertex`, so add it as an additional output and pass it along.

```
struct VertexOutput {
	float4 clipPos : SV_POSITION;
	float3 normal : TEXCOORD0;
	float3 worldPos : TEXCOORD1;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

VertexOutput LitPassVertex (VertexInput input) {
	…
	output.worldPos = worldPos.xyz;
	return output;
}

float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	…

	float3 diffuseLight = 0;
	for (int i = 0; i < MAX_VISIBLE_LIGHTS; i++) {
		diffuseLight += DiffuseLight(i, input.normal, input.worldPos);
	}
	float3 color = diffuseLight * albedo;
	return float4(color, 1);
}
```



Correct light direction.

3.2 Distance Attenuation

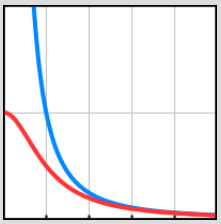
Except for directional lights—which are assumed to be infinitely far away—the intensity of a light decreases with distance. The relation is $\frac{i}{d^2}$, where i is the light's stated intensity and d is the distance between the light source and the surface. This is known as the inverse-square law. So we have to divide the final diffuse contribution by the square of the light vector. To avoid a division by zero, we enforce a tiny minimum for the square distance used.

```
	float diffuse = saturate(dot(normal, lightDirection));

	float distanceSqr = max(dot(lightVector, lightVector), 0.00001);
	diffuse /= distanceSqr;

	return diffuse * lightColor;
```

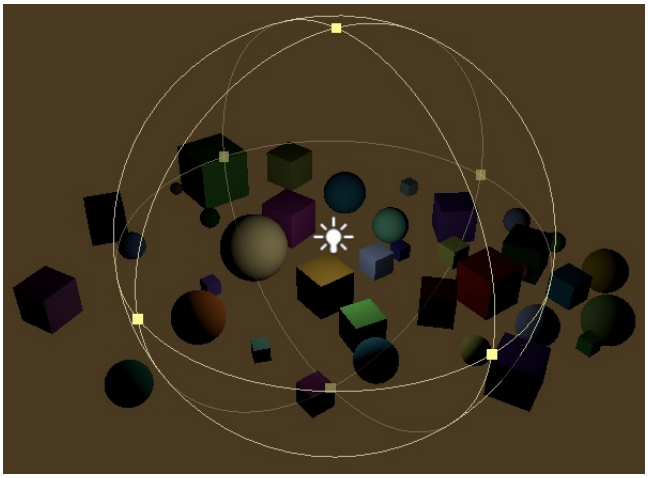
Doesn't that increase the intensity very close to point lights?

Indeed, when d is less than 1 a light's intensity goes up. When d approaches its minimum the intensity becomes enormous.

Unity's default pipeline uses $\frac{i}{1 + d^2}$ to avoid increasing the brightness, but is less realistic and produces results that are too dark close to the light. The Lightweight pipeline initially used the same falloff, but starting with version 3.3.0 it uses the correct square falloff.



$\frac{1}{d^2}$ *and* $\frac{1}{1 + d^2}$.



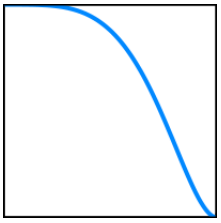
Light fades with distance.

As the light vector is the same as the direction vector for directional lights, the square distance ends up as 1. That means that directional lights aren't affected by distance attenuation, which is correct.

3.3 Light Range

Point lights also have a configured range, which limits their area of influence. Nothing outside this range is affected by the light, even though it could still illuminate objects. This isn't realistic, but allows better control of lighting and limits how many objects are affected by the light. Without this range limit every light would always be considered visible.

The range limit isn't a sudden cutoff. Instead, the light's intensity is smoothly faded out, based on the square distance. The Lightweight pipeline and lightmapper use $\left(1 - \left(\frac{d^2}{r^2}\right)^2\right)^2$ where r is the light's range. We'll use the same fade curve.



Range fade curve.

The light ranges are part of the scene data, so we have to send it to the GPU, per light. We'll use another array for this attenuation data. While we could suffice with a float array, we'll once again use a vector array, as we'll need to include more data later.

```
	static int visibleLightColorsId =
		Shader.PropertyToID("_VisibleLightColors");
	static int visibleLightDirectionsOrPositionsId =
		Shader.PropertyToID("_VisibleLightDirectionsOrPositions");
	static int visibleLightAttenuationsId =
		Shader.PropertyToID("_VisibleLightAttenuations");

	Vector4[] visibleLightColors = new Vector4[maxVisibleLights];
	Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];
	Vector4[] visibleLightAttenuations = new Vector4[maxVisibleLights];
```

Also copy the new array to the GPU in `Render`.

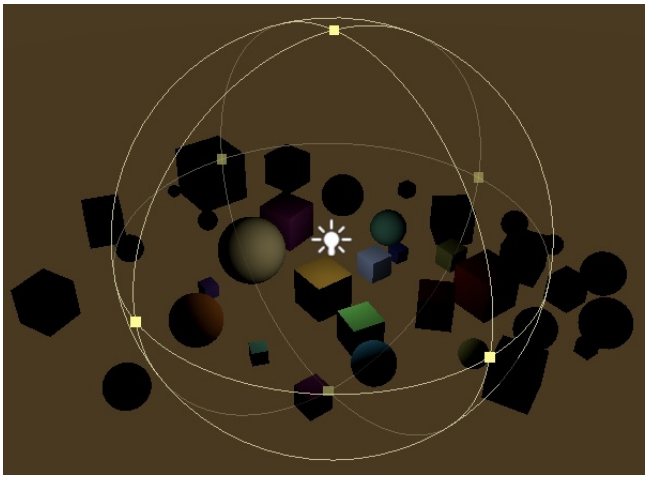
```
		cameraBuffer.SetGlobalVectorArray(
			visibleLightDirectionsOrPositionsId, visibleLightDirectionsOrPositions
		);
		cameraBuffer.SetGlobalVectorArray(
			visibleLightAttenuationsId, visibleLightAttenuations
		);
```

And fill it in `ConfigureLights`. Directional lights have no range limit, so they can use the zero vector. In the case of points lights, we put their range in the X component of the vector. But rather than store the range directly, we reduce the work that the shader has to do by storing $\frac{1}{r^2}$ and avoiding a division by zero.

```
			Vector4 attenuation = Vector4.zero;

			if (light.lightType == LightType.Directional) {
				…
			}
			else {
				visibleLightDirectionsOrPositions[i] =
					light.localToWorld.GetColumn(3);
				attenuation.x = 1f /
					Mathf.Max(light.range * light.range, 0.00001f);
			}

			visibleLightAttenuations[i] = attenuation;
```

Add the new array to the shader, calculate the fading caused by range, and factor that into the final diffuse contribution.

```
CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightDirectionsOrPositions[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightAttenuations[MAX_VISIBLE_LIGHTS];
CBUFFER_END

float3 DiffuseLight (int index, float3 normal, float3 worldPos) {
	float3 lightColor = _VisibleLightColors[index].rgb;
	float4 lightPositionOrDirection = _VisibleLightDirectionsOrPositions[index];
	float4 lightAttenuation = _VisibleLightAttenuations[index];

	float3 lightVector =
		lightPositionOrDirection.xyz - worldPos * lightPositionOrDirection.w;
	float3 lightDirection = normalize(lightVector);
	float diffuse = saturate(dot(normal, lightDirection));

	float rangeFade = dot(lightVector, lightVector) * lightAttenuation.x;
	rangeFade = saturate(1.0 - rangeFade * rangeFade);
	rangeFade *= rangeFade;

	float distanceSqr = max(dot(lightVector, lightVector), 0.00001);
	diffuse *= rangeFade / distanceSqr;

	return diffuse * lightColor;
}
```



Light fades out based on range.

Once again directional lights aren't affected, because in their case `lightAttenuation.x` is always 0, thus `rangeFade` is always 1.

4 Spotlights

The Lightweight pipeline also supports spotlights, so we'll add them too. Spotlights work like point lights, but are restricted to a cone instead of shining in all directions.

4.1 Spot Direction

Like a directional light, a spotlight shines along its local Z axis, but in a cone. And it also has a position, which means that we have to provide both for spotlights. So add an additional array for the spot direction to `MyPipeline`.

```
	static int visibleLightAttenuationsId =
		Shader.PropertyToID("_VisibleLightAttenuations");
	static int visibleLightSpotDirectionsId =
		Shader.PropertyToID("_VisibleLightSpotDirections");

	Vector4[] visibleLightColors = new Vector4[maxVisibleLights];
	Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];
	Vector4[] visibleLightAttenuations = new Vector4[maxVisibleLights];
	Vector4[] visibleLightSpotDirections = new Vector4[maxVisibleLights];

	…

	void Render (ScriptableRenderContext context, Camera camera) {
		…
			cameraBuffer.SetGlobalVectorArray(
			visibleLightAttenuationsId, visibleLightAttenuations
		);
		cameraBuffer.SetGlobalVectorArray(
			visibleLightSpotDirectionsId, visibleLightSpotDirections
		);
		…
	}
```

In `ConfigureLights`, when not dealing with a directional light, also check whether the light is a spotlight. If so, setup the direction vector, just like for a directional light, but assign it to `visibleLightSpotDirections` instead.

```
			if (light.lightType == LightType.Directional) {
				Vector4 v = light.localToWorld.GetColumn(2);
				v.x = -v.x;
				v.y = -v.y;
				v.z = -v.z;
				visibleLightDirectionsOrPositions[i] = v;
			}
			else {
				visibleLightDirectionsOrPositions[i] =
					light.localToWorld.GetColumn(3);
				attenuation.x = 1f /
					Mathf.Max(light.range * light.range, 0.00001f);

				if (light.lightType == LightType.Spot) {
					Vector4 v = light.localToWorld.GetColumn(2);
					v.x = -v.x;
					v.y = -v.y;
					v.z = -v.z;
					visibleLightSpotDirections[i] = v;
				}
			}
```

Add the new data to the shader too.

```
CBUFFER_START(_LightBuffer)
	float4 _VisibleLightColors[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightDirectionsOrPositions[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightAttenuations[MAX_VISIBLE_LIGHTS];
	float4 _VisibleLightSpotDirections[MAX_VISIBLE_LIGHTS];
CBUFFER_END

float3 DiffuseLight (int index, float3 normal, float3 worldPos) {
	float3 lightColor = _VisibleLightColors[index].rgb;
	float4 lightPositionOrDirection = _VisibleLightDirectionsOrPositions[index];
	float4 lightAttenuation = _VisibleLightAttenuations[index];
	float3 spotDirection = _VisibleLightSpotDirections[index].xyz;

	…
}
```

4.2 Angle Falloff

The cone of a spotlight is specified with a positive angle that's less than 180°. We can determine whether a surface point lies within the cone by taking the dot product of the spot's direction and the light direction. If the result is at most the cosine of half the configured spot angle, then the fragment is affected by the light.

There isn't an instant cutoff at the edge of the cone. Instead, there is a transition range in which the light fades out. This range can be defined by an inner spot angle where the fading begins and an outer spot angle where the light intensity reaches zero. However, Unity's spotlight only allows us to set the outer angle. Unity's default pipeline uses a light cookie to determine the falloff, while the Lightweight pipeline computes the falloff with a smooth function that assumes a fixed relationship between the inner and outer angles.

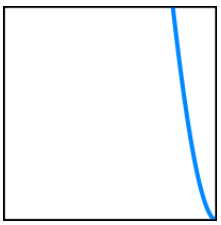
To determine the falloff, begin by converting half the spot's angle from degrees to radians, then compute its cosine. The configured angle is made available via `VisibleLight.spotAngle`.

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

			float outerRad = Mathf.Deg2Rad * 0.5f * light.spotAngle;
			float outerCos = Mathf.Cos(outerRad);
		}
```

The Lightweight pipeline and the lightmapper define the inner angle with the relationship $\tan(r_i) = \frac{46}{64}\tan(r_o)$, where r_i and r_o are half the inner and outer spot angles in radians. We need to use the cosine of the inner angle, so the complete relationship is $\cos(r_i) = \cos\left(\arctan\left(\frac{46}{64}\tan(r_o)\right)\right)$.

```
			float outerRad = Mathf.Deg2Rad * 0.5f * light.spotAngle;
			float outerCos = Mathf.Cos(outerRad);
			float outerTan = Mathf.Tan(outerRad);
			float innerCos =
				Mathf.Cos(Mathf.Atan(((46f / 64f) * outerTan));
```

The angle-based falloff is defined as $\frac{D_s \cdot D_l - \cos(r_o)}{\cos(r_i) - \cos(r_o)}$ clamped to 0–1 and then squared, with $D_s \cdot D_l$ being the dot product of the spot direction and light direction.



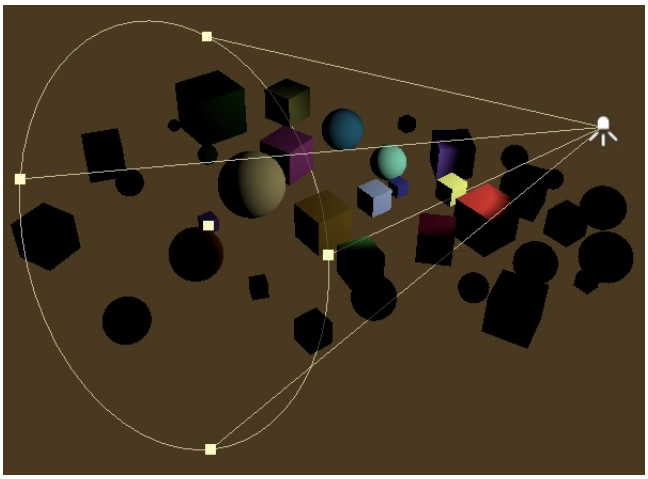
Spot falloff from 0° to 45° for 90° spot angle.

That expression can be simplified to $(D_s \cdot D_l)a + b$, with $a = \frac{1}{\cos(r_i) - \cos(r_o)}$ and $b = -\cos(r_o)a$. That allows us to compute a and b in `ConfigureLights` and store them in the last two components of the attenuation data vector.

```
			float outerRad = Mathf.Deg2Rad * 0.5f * light.spotAngle;
			float outerCos = Mathf.Cos(outerRad);
			float outerTan = Mathf.Tan(outerRad);
			float innerCos =
				Mathf.Cos(Mathf.Atan(((64f - 18f) / 64f) * outerTan));
			float angleRange = Mathf.Max(innerCos - outerCos, 0.001f);
			attenuation.z = 1f / angleRange;
			attenuation.w = -outerCos * attenuation.z;
```

In the shader, the spot fade factor can then be computed with a dot product, multiplication, addition, saturation, and finally squaring. Then use the result to modulate the diffuse light.

```
	float rangeFade = dot(lightVector, lightVector) * lightAttenuation.x;
	rangeFade = saturate(1.0 - rangeFade * rangeFade);
	rangeFade *= rangeFade;

	float spotFade = dot(spotDirection, lightDirection);
	spotFade = saturate(spotFade * lightAttenuation.z + lightAttenuation.w);
	spotFade *= spotFade;

	float distanceSqr = max(dot(lightVector, lightVector), 0.00001);
	diffuse *= spotFade * rangeFade / distanceSqr;
```



Spotlight, intensity 4.

To keep the spot fade calculation from affecting the other light types, set the W component of their attenuation vector to 1.

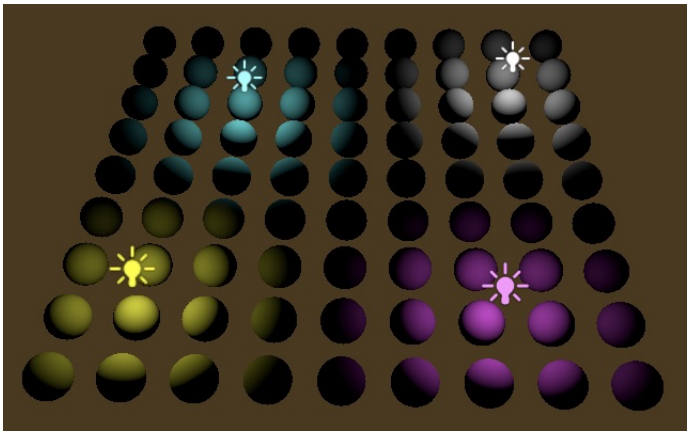
```
			Vector4 attenuation = Vector4.zero;
			attenuation.w = 1f;
```

What about area lights?

Unity's Lightweight and default pipelines do not support realtime area lights, and neither will we. Area lights are only used for lightmapping, which we'll support later.

5 Lights Per Object

We currently support up to four lights per object. In fact, we always calculate lighting for four lights per object, even when that would not be necessary. For example, consider a 9×9 grid of 81 spheres with four point lights near its corners. If the range of those lights is set so they cover roughly a quarter of the grid each, then most spheres end up affected by only a single light, some by two, and some by none.



Grid of 81 spheres, with four point lights.

At the moment those 81 spheres are rendered with one draw call—assuming GPU instancing is enabled—but light contribution is calculated four times per sphere fragment. It would be better if we could somehow only calculate the lights that are needed per object. That would allow us to also increase the number of visible lights supported.

5.1 Light Indices

During culling, Unity determines the lights that are visible, which also involves figuring out which lights affect each object. We can ask Unity to send this information to the GPU, in the form of a list of light indices.

Unity currently supports two formats for the light indices. The first approach is to store up to eight indices in two `float4` variables, which are set per object. The second approach is to put a list of the light indices for all objects in a single buffer, similar to how GPU-instanced data is stored. However, the second approach is disabled in Unity 2018.3, with only the first option supported. So although not ideal, we'll limit ourselves to the first option too, for now.

We instruct Unity to set the light indices via `float4` fields by setting the `rendererConfiguration` field of the draw settings to `RendererConfiguration.PerObjectLightIndices8`.

```
		var drawSettings = new DrawRendererSettings(
			camera, new ShaderPassName("SRPDefaultUnlit")
		) {
			flags = drawFlags,
			rendererConfiguration = RendererConfiguration.PerObjectLightIndices8
		};
		//drawSettings.flags = drawFlags;
		drawSettings.sorting.flags = SortFlags.CommonOpaque;
```

Unity now has to setup additional GPU data per object, which affects GPU instancing. Unity tries to group objects that are affected by the same lights, but prefers to group based on distance. Also, the light indices are ordered based on the relative light importance per object, which can split batches further. In the case of the grid example, I ended up with 30 draw calls, which is a lot more than 1, but still much less than 81.

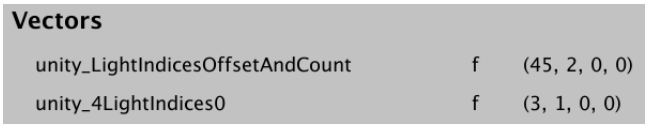
The indices are made available via the `unity_4LightIndices0` and `unity_4LightIndices1` vectors, which should be part of the `UnityPerDraw` buffer. Besides that, there's also `unity_LightIndicesOffsetAndCount`, another `float4` vector. Its Y component contains the number of lights affecting the object. Its X component contains an offset for when the second approach is used, so we can ignore that.

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

Now we can limit ourselves to invoking `DiffuseLight` only as often as needed. But we have to retrieve the correct light indices. As we currently support up to four visible lights, all we need is `unity_4LightIndices0`, which we can index as an array to retrieve its appropriate component.

```
	for (int i = 0; i < unity_LightIndicesOffsetAndCount.y; i++) {
		int lightIndex = unity_4LightIndices0[i];
		diffuseLight +=
			DiffuseLight(lightIndex, input.normal, input.worldPos);

	}
```

While there should be no visible change—assuming there are only up to four visible lights—the GPU now has less work to do, because it only calculates light contribution of the relevant lights. You can use the frame debugger to check how many lights end up being used per draw call. The shader did become more complex, because we're now using a variable loop instead of a fixed one. Whether that results in better or worse performance can vary. The more visible lights we support, the better this new approach is.



Two lights affecting an object, indices 3 and 1.

Note that as we no longer loop through the maximum visible lights, we no longer need to clear light data that ends up not being used.

```
	void ConfigureLights () {
		//int i = 0;
		for (int i = 0; i < cull.visibleLights.Count; i++) {
			…
		}
		//for (; i < maxVisibleLights; i++) {
		//	visibleLightColors[i] = Color.clear;
		//}
	}
```

5.2 More Visible Lights

Our new approach makes it possible to support more visible lights without automatically increasing the work that the GPU has to do. Let's increase the limit to 16, the same used by the Lightweight pipeline. That requires us to send a bit more data to the GPU each frame, but most objects will only be affected by a few lights. Adjust `MAX_VISIBLE_LIGHTS` in the shader.

```
#define MAX_VISIBLE_LIGHTS 16
```

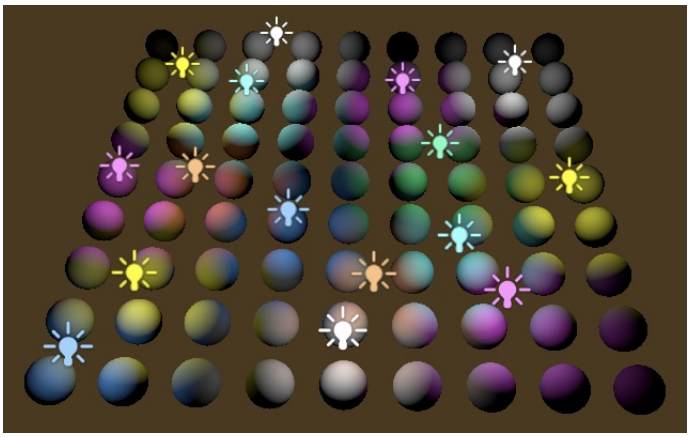
And `maxVisibleLights` in `MyPipeline`.

```
	const int maxVisibleLights = 16;
```

After recompiling, Unity will warn us that that we're exceeding the previous array size. Unfortunately, it's not possible to just change the size of a fixed array in a shader. That's a graphics API limitation, not something that we can do anything about. The application has to be restarted before the new size is used, so you'll have to restart the Unity editor.

Before we go ahead and add more lights to the scene, we have to realize that `unity_4LightIndices0` only contains up to four indices, even though an object can now be affected by more than four lights. To prevent incorrect results, we have to make sure that our light loop doesn't go beyond four.

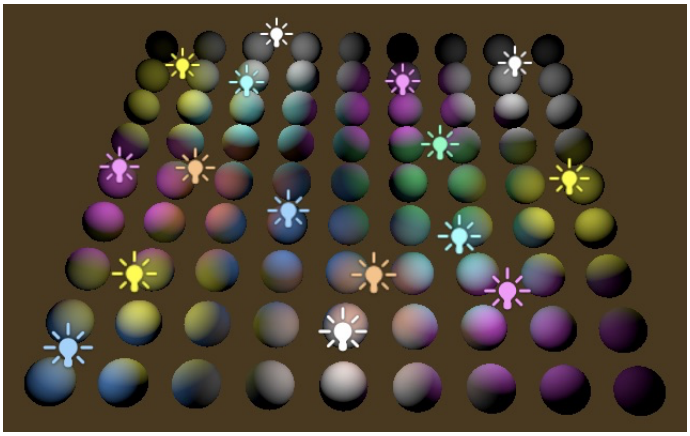
```
	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		diffuseLight +=
			DiffuseLight(lightIndex, input.normal, input.worldPos);
	}
```



Sixteen lights, at most four per object.

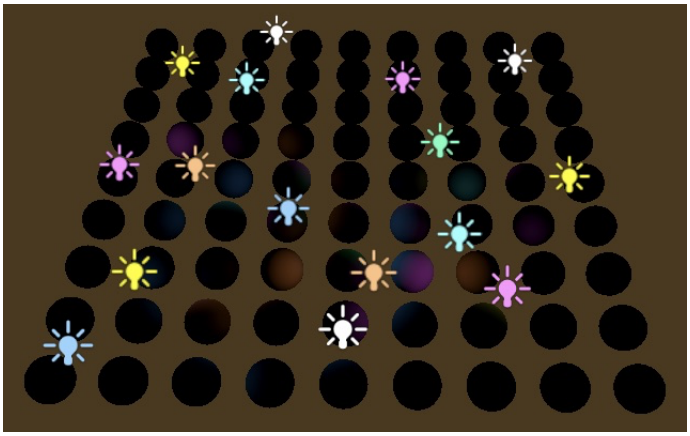
But we don't have to limit ourselves to at most four lights per object. There's also `unity_4LightIndices1`, which can contain another four light indices. Let's simply add a second loop after the first one, starting at index 4 and retrieving the light indices from `unity_4LightIndices1`. That increases the maximum lights per object to eight. We should make sure that we don't exceed eight though, because it's possible for objects to be affected by even more lights in the scene.

```
	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		int lightIndex = unity_4LightIndices0[i];
		diffuseLight +=
			DiffuseLight(lightIndex, input.normal, input.worldPos);
	}
	for (int i = 4; i < min(unity_LightIndicesOffsetAndCount.y, 8); i++) {
		int lightIndex = unity_4LightIndices1[i - 4];
		diffuseLight +=
			DiffuseLight(lightIndex, input.normal, input.worldPos);
	}
```



Up to eight lights per object.

As the light indices are sorted based on relative importance, usually the second quartet of lights isn't as obvious as the first. Also, most objects won't be affected by that many lights. To see the difference that the extra four lights make, you can temporarily disable the first loop.



Skipping the first four lights per object.

5.3 Vertex Lights

As the second quartet of lights is likely much less visually important than the first, we could make them less costly by computing their contribution per vertex instead of per light. The light contribution will be linearly interpolated between vertices, which is less accurate, but is acceptable for subtle diffuse lighting, as long as the light distance is reasonably large compared to the triangle edge length.

It's possible to fine-tune how many pixel and vertex lights we support, but we'll simply move the second light loop to `LitPassVertex`, which only requires adjusting the variables used. That means we support up to four pixel lights plus four vertex lights. The vertex lighting has to be added to `VertexOutput` and used as the initial value for `diffuseLight` in `LitPassFragment`.

```
struct VertexOutput {
	float4 clipPos : SV_POSITION;
	float3 normal : TEXCOORD0;
	float3 worldPos : TEXCOORD1;
	float3 vertexLighting : TEXCOORD2;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

VertexOutput LitPassVertex (VertexInput input) {
	VertexOutput output;
	UNITY_SETUP_INSTANCE_ID(input);
	UNITY_TRANSFER_INSTANCE_ID(input, output);
	float4 worldPos = mul(UNITY_MATRIX_M, float4(input.pos.xyz, 1.0));
	output.clipPos = mul(unity_MatrixVP, worldPos);
	output.normal = mul((float3x3)UNITY_MATRIX_M, input.normal);
	output.worldPos = worldPos.xyz;

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

	return output;
}

float4 LitPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	input.normal = normalize(input.normal);
	float3 albedo = UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color).rgb;

	float3 diffuseLight = input.vertexLighting;
	for (int i = 0; i < min(unity_LightIndicesOffsetAndCount.y, 4); i++) {
		…
	}
	//for (int i = 4; i < min(unity_LightIndicesOffsetAndCount.y, 8); i++) {
	//	…
	//}
	float3 color = diffuseLight * albedo;
	return float4(color, 1);
}
```

5.4 Too Many Visible Lights

Although we now support up to 16 visible lights, with enough lights in the scene we can still end up exceeding that limit. When that happens, the overall least important lights are omitted when rendering. However, that's only because we don't copy their data to the shader. Unity doesn't know about that and doesn't eliminate those lights from the light index list per object. So we can end up with light indices that are out of bounds. To prevent that, we have to tell Unity that some lights were eliminated.

We can get a list of the indices of all visible lights via invoking `GetLightIndexMap` on the cull results. Unity allows us to modify this mapping and then assign it back to the cull results via `SetLightIndexMap`. The point of this is that Unity will skip all lights whose index has been changed to −1. Do this at the end of `ConfigureLights`, for all lights beyond the maximum.

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

		int[] lightIndices = cull.GetLightIndexMap();
		for (int i = maxVisibleLights; i < cull.visibleLights.Count; i++) {
			lightIndices[i] = -1;
		}
		cull.SetLightIndexMap(lightIndices);
	}
```

We only really need to do that when we end up with too many visible lights, which shouldn't happen all the time.

```
		if (cull.visibleLights.Count > maxVisibleLights) {
			int[] lightIndices = cull.GetLightIndexMap();
			for (int i = maxVisibleLights; i < cull.visibleLights.Count; i++) {
				lightIndices[i] = -1;
			}
			cull.SetLightIndexMap(lightIndices);
		}
```

Unfortunately, `GetLightIndexMap` creates a new array each invocation, so our pipeline now allocates memory every frame we end up with too many visible lights. We currently cannot do anything about that, but future Unity releases will give us access to an allocation-free alternative of `GetLightIndexMap`.

5.5 Zero Visible Lights

Another possibility is that there are zero visible lights. This should just work, but unfortunately Unity crashes when trying to set the light indices in this case. We can avoid the crash by only using per-object light indices when we have at least one visible light.

```
		var drawSettings = new DrawRendererSettings(
			camera, new ShaderPassName("SRPDefaultUnlit")
		) {
			flags = drawFlags//,
			//rendererConfiguration = RendererConfiguration.PerObjectLightIndices8
		};
		if (cull.visibleLights.Count > 0) {
			drawSettings.rendererConfiguration =
				RendererConfiguration.PerObjectLightIndices8;
		}
```

If there are no lights, we can also completely skip invoking `ConfigureLights`.

```
		if (cull.visibleLights.Count > 0) {
			ConfigureLights();
		}
```

A side effect of not having Unity set the light data is that they keep the value set for the last object. So we can end up with a nonzero light count for all objects. To avoid that, we'll manually set *unity_LightIndicesOffsetAndCount* to zero.

```
	static int lightIndicesOffsetAndCountID =
		Shader.PropertyToID("unity_LightIndicesOffsetAndCount");

	…

	void Render (ScriptableRenderContext context, Camera camera) {
		…

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

		…
	}
```

The next tutorial is Spotlight Shadows.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick