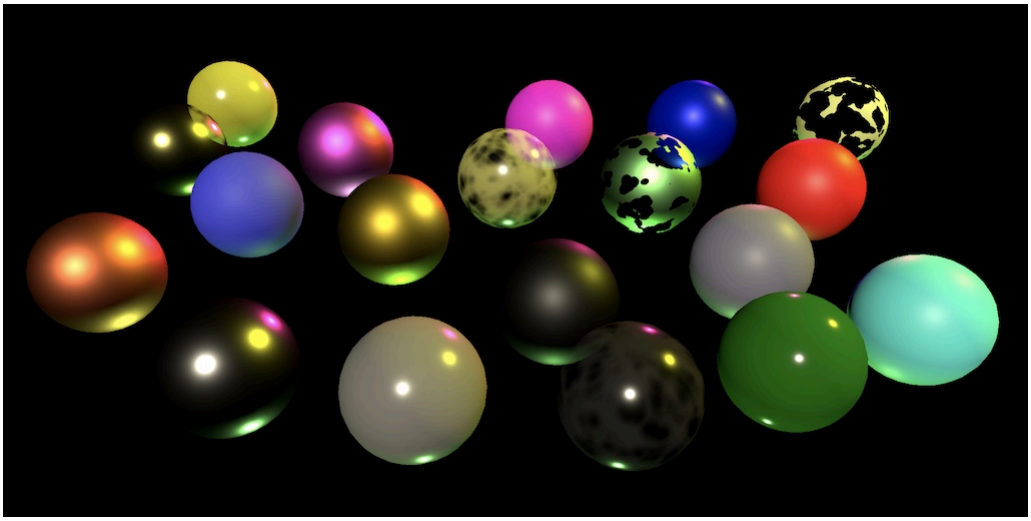
updated 2023-08-03 published 2019-11-30

Directional Lights Direct Illumination

Use normal vectors to calculate lighting.
Support up to four directional lights.
Apply a BRDF.
Make lit transparent materials.
Create a custom shader GUI with presets.

This is the third part of a tutorial series about creating a custom scriptable render pipeline. It adds support for shading with multiple directional lights.

This tutorial is made with Unity 2019.2.12f1 and upgraded to 2022.3.5f1.



A variety of spheres illuminated by four lights.

1 Lighting

If we want to create a more realistic scene then we'll have to simulate how light interacts with surfaces. This requires a more complex shader than the unlit one that we currently have.

1.1 Lit Shader

Duplicate the *UnlitPass* HLSL file and rename it to *LitPass*. Adjust the include guard define and the vertex and fragment function names to match. We'll add lighting calculations later.

```
#ifndef CUSTOM_LIT_PASS_INCLUDED
#define CUSTOM_LIT_PASS_INCLUDED

…

Varyings LitPassVertex (Attributes input) { … }

float4 LitPassFragment (Varyings input) : SV_TARGET { … }

#endif
```

Also duplicate the *Unlit* shader and rename it to *Lit*. Change its menu name, the file it includes, and the functions it uses. Let's also change the default color to gray, as a fully white surface in a well-lit scene can appear very bright. The Universal pipeline uses a gray color by default as well.

```
Shader "Custom RP/Lit" {

	Properties {
		_BaseMap("Texture", 2D) = "white" {}
		_BaseColor("Color", Color) = (0.5, 0.5, 0.5, 1.0)
		…
	}

	SubShader {
		Pass {
			…
			#pragma vertex LitPassVertex
			#pragma fragment LitPassFragment
			#include "LitPass.hlsl"
			ENDHLSL
		}
	}
}
```

We're going to use a custom lighting approach, which we'll indicate by setting the light mode of our shader to *CustomLit*. Add a `Tags` block to the `Pass`, containing `"LightMode" = "CustomLit"`.

```
		Pass {
			Tags {
				"LightMode" = "CustomLit"
			}

			…
		}
```

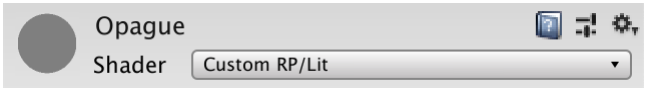
To render objects that use this pass we have to include it in `CameraRenderer`. First add a shader tag identifier for it.

```
	static ShaderTagId
		unlitShaderTagId = new ShaderTagId("SRPDefaultUnlit"),
		litShaderTagId = new ShaderTagId("CustomLit");
```

Then add it to the passes to be rendered in `DrawVisibleGeometry`, like we did in `DrawUnsupportedShaders`.

```
		var drawingSettings = new DrawingSettings(
			unlitShaderTagId, sortingSettings
		) {
			enableDynamicBatching = useDynamicBatching,
			enableInstancing = useGPUInstancing
		};
		drawingSettings.SetShaderPassName(1, litShaderTagId);
```

Now we can create a new opaque material, though at this point it produces the same results an as unlit material.



Default opaque material.

1.2 Normal Vectors

How well an object is lit depends on multiple factors, including the relative angle between the light and surface. To know the surface's orientation we need to access the surface normal, which is a unit-length vector pointing straight away from it. This vector is part of the vertex data, defined in object space just like the position. So add it to `Attributes` in *LitPass*.

```
struct Attributes {
	float3 positionOS : POSITION;
	float3 normalOS : NORMAL;
	float2 baseUV : TEXCOORD0;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Lighting is calculated per fragment, so we have to add the normal vector to `Varyings` as well. We'll perform the calculations in world space, so name it `normalWS`.

```
struct Varyings {
	float4 positionCS : SV_POSITION;
	float3 normalWS : VAR_NORMAL;
	float2 baseUV : VAR_BASE_UV;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

We can use `TransformObjectToWorldNormal` from *SpaceTransforms* to convert the normal to world space in `LitPassVertex`.

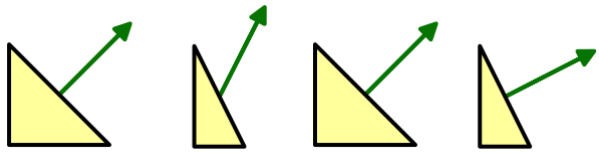
```
	output.positionWS = TransformObjectToWorld(input.positionOS);
	output.positionCS = TransformWorldToHClip(positionWS);
	output.normalWS = TransformObjectToWorldNormal(input.normalOS);
```

How does `TransformObjectToWorldNormal` work?

When you check the code you'll see that it uses one of two approaches, based on whether `UNITY_ASSUME_UNIFORM_SCALING` is defined.

When `UNITY_ASSUME_UNIFORM_SCALING` is defined it invokes `TransformObjectToWorldDir`, which does the same as `TransformObjectToWorld` except that it ignores the translation part, as we're dealing with a direction vector instead of a position. But the vector also gets uniformly scaled, so it should get normalized later.

In the other case uniform scaling is not assumed. This is more complicated, because when an object gets deformed by nonuniform scaling the normal vectors have to get scaled in reverse to match the new surface orientation. This requires a multiplication with the transposed **`UNITY_MATRIX_I_M`** matrix instead, plus normalization.

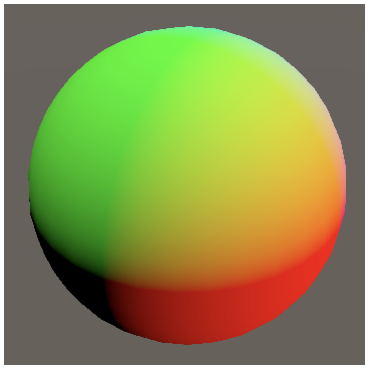


Incorrect and correct normal transformation.

Using `UNITY_ASSUME_UNIFORM_SCALING` is a slight optimization, which you can enable by defining it yourself. However, it makes more of a difference when GPU instancing is used, because then an array of **`UNITY_MATRIX_I_M`** matrices has to be send to the GPU. Avoiding that when not needed is worthwhile. You can enable it by adding the **`#pragma instancing_options assumeuniformscaling`** directive to the shader, but only do this if you're exclusively rendering objects with uniform scale.

To verify whether we get a correct normal vector in `LitPassFragment` we can use it as a color.

```
	base.rgb = input.normalWS;
	return base;
```



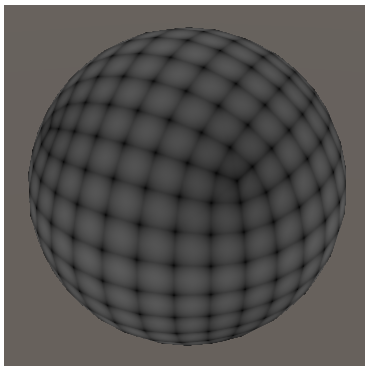
World-space normal vectors.

Negative values cannot be visualized, so they're clamped to zero.

1.3 Interpolated Normals

Although the normal vectors are unit-lengh in the vertex program, linear interpolation across triangles affects their length. We can visualize the error by rendering the difference between one and the vector's length, magnified by ten to make it more obvious.

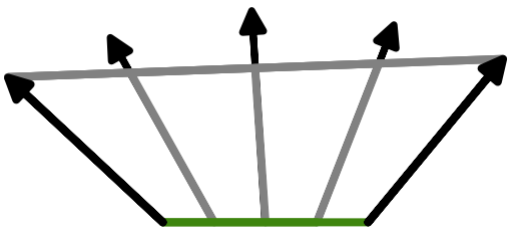
```
	base.rgb = abs(length(input.normalWS) - 1.0) * 10.0;
```



Interpolated normal error, exaggerated.

We can smooth out the interpolation distortion by normalizing the normal vector in `LitPassFragment`. The difference isn't really noticeable when just looking at the normal vectors, but it's more obvious when used for lighting.

```
	base.rgb = normalize(input.normalWS);
```



Normalization after interpolation.

1.4 Surface Properties

Lighting in a shader is about simulating the interactions between light that hits a surface, which means that we must keep track of the surface's properties. Right now we have a normal vector and a base color. We can split the latter in two: the RGB color and the alpha value. We'll be using this data in a few places, so let's define a convenient `Surface` struct to contain all relevant data. Put it in a separate *Surface* HLSL file in the *ShaderLibrary* folder.

```
#ifndef CUSTOM_SURFACE_INCLUDED
#define CUSTOM_SURFACE_INCLUDED

struct Surface {
	float3 normal;
	float3 color;
	float alpha;
};

#endif
```

Shouldn't we define the normal as `normalWS`?

We could, but the surface doesn't care in what space the normal is defined. Lighting calculations could be performed in any proper 3D space. So we leave the space undefined. When filling the data we just have to use the same space everywhere. We'll use world space, but we could switch to another space later and everything would still work the same.

Include it in *LitPass*, after *Common*. That way we can keep *LitPass* short. We'll put specialized code in its own HLSL file from now on, to make it easier to locate the relevant functionality.

```
#include "../ShaderLibrary/Common.hlsl"
#include "../ShaderLibrary/Surface.hlsl"
```

Define a `surface` variable in *LitPassFragment* and fill it. Then the final result becomes the surface's color and alpha.

```
	Surface surface;
	surface.normal = normalize(input.normalWS);
	surface.color = base.rgb;
	surface.alpha = base.a;

	return float4(surface.color, surface.alpha);
```

Isn't that inefficient code?

It makes no difference, because the shader compiler generates highly optimized programs, completely rewriting our code. The struct is purely for our convenience. You can inspect the compiler's work via the *Compile and show code* button in the shader's inspector.

1.5 Calculating Lighting

To calculate the actual lighting we'll create a `GetLighting` function that has a `Surface` parameter. Initially have it return the Y component of the surface normal. As this is lighting functionality we'll put it in a separate *Lighting* HLSL file.

```
#ifndef CUSTOM_LIGHTING_INCLUDED
#define CUSTOM_LIGHTING_INCLUDED

float3 GetLighting (Surface surface) {
	return surface.normal.y;
}

#endif
```

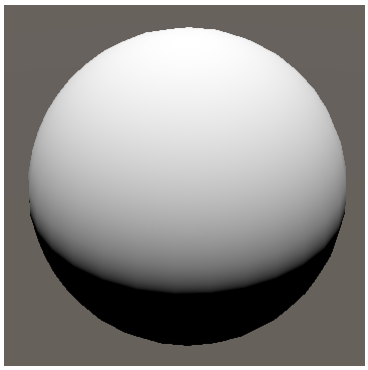
Include it in *LitPass*, after including *Surface* because *Lighting* depends on it.

```
#include "../ShaderLibrary/Surface.hlsl"
#include "../ShaderLibrary/Lighting.hlsl"
```

> **Why not include *Surface* in *Lighting*?**
>
> We could do that, but we'll end up with multiple files depending on multiple other files. I choose to instead put all include statements in one place, which makes the dependencies clear. That also makes it easy to replace one file with another, to change how the shader works, as long as the new file defines the same functionality that others rely on.

Now we can get the lighting in `LitPassFragment` and use that for the RGB part of the fragment.

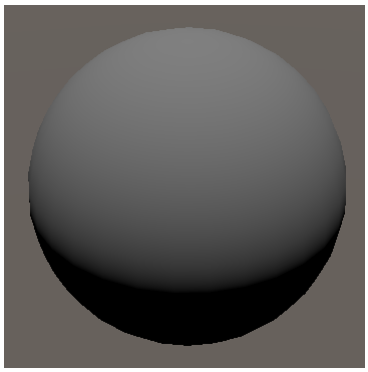
```
	float3 color = GetLighting(surface);
	return float4(color, surface.alpha);
```



Diffuse lighting from above.

At this point the result is the Y component of the surface normal, so it is one at the top of the sphere and drops down to zero at its sides. Below that the result becomes negative, reaching −1 at the bottom, but we cannot see negative values. It matches the cosine of the angle between the normal and up vectors. Ignoring the negative part, this visually matches diffuse lighting of a directional light pointing straight down. The finishing touch would be to factor the surface color into the result in `GetLighting`, interpreting it as the surface albedo.

```
float3 GetLighting (Surface surface) {
	return surface.normal.y * surface.color;
}
```



Albedo applied.

What does albedo mean?

Albedo means whiteness in Latin. It's a measure of how much light is diffusely reflected by a surface. If albedo isn't fully white then part of the light energy gets absorbed instead of reflected.

2 Lights

To perform proper lighting we also need to know the properties of the light. In this tutorial we'll limit ourselves to directional lights only. A directional light represents a source of light so far away that its position doesn't matter, only its direction. This is a simplification, but it's good enough to simulate the Sun's light on Earth and other situations where incoming light is more or less unidirectional.

2.1 Light Structure

We'll use a struct to store the light data. For now we can suffice with a color and a direction. Put it in a separate *Light* HLSL file. Also define a `GetDirectionalLight` function that returns a configured directional light. Initially use a white color and the up vector, matching the light data that we're currently using. Note that the light's direction is thus defined as the direction from where the light is coming, not where it is going.

```
#ifndef CUSTOM_LIGHT_INCLUDED
#define CUSTOM_LIGHT_INCLUDED

struct Light {
	float3 color;
	float3 direction;
};

Light GetDirectionalLight () {
	Light light;
	light.color = 1.0;
	light.direction = float3(0.0, 1.0, 0.0);
	return light;
}

#endif
```

Include the file in *LitPass* before *Lighting*.

```
#include "../ShaderLibrary/Light.hlsl"
#include "../ShaderLibrary/Lighting.hlsl"
```

2.2 Lighting Functions

Add an `IncomingLight` function to *Lighting* that calculates how much incoming light there is for a given surface and light. For an arbitrary light direction we have to take the dot product of the surface normal and the direction. We can use the `dot` function for that. The result should be modulated by the light's color.

```
float3 IncomingLight (Surface surface, Light light) {
	return dot(surface.normal, light.direction) * light.color;
}
```

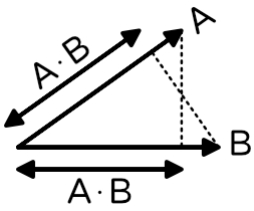
What's a dot product?

The dot product between two vectors is geometrically defined as $A \cdot B = ||A|| \, ||B|| \cos \theta$. This means that it is the cosine of the angle between the vectors, multiplied by their lengths. So in the case of two unit-length vectors $A \cdot B = \cos \theta$.

Algebraically, it is defined as $A \cdot B = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \ldots + A_n B_n$. This means that you can compute it by multiplying all component pairs and summing them.

```
float dotProduct = a.x * b.x + a.y * b.y + a.z * b.z;
```

Visually, this operation projects one vector straight down to the other, as if casting a shadow on it. In doing so, you end up with a right triangle of which the bottom side's length is the result of the dot product. And if both vectors are unit length, that's the cosine of their angle.



Dot product.

But this is only correct when the surface is oriented toward the light. When the dot product is negative we have to clamp it to zero, which we can do via the `saturate` function.

```
float3 IncomingLight (Surface surface, Light light) {
	return saturate(dot(surface.normal, light.direction)) * light.color;
}
```

What does `saturate` do?

It clamps the value so it lies between zero and one inclusive. We only need to specify a minimum because the dot product should never be greater than one, but saturation is such a common shader operation that it usually a free operation modifier.

Add another `GetLighting` function, which returns the final lighting for a surface and light. For now it's the incoming light multiplied by the surface color. Define it above the other function.

```
float3 GetLighting (Surface surface, Light light) {
	return IncomingLight(surface, light) * surface.color;
}
```

Finally, adjust the `GetLighting` function that only has a surface parameter so it invokes the other one, using `GetDirectionalLight` to provide the light data.

```
float3 GetLighting (Surface surface) {
	return GetLighting(surface, GetDirectionalLight());
}
```

2.3 Sending Light Data to the GPU

Instead of always using a white light from above we should use the light of the current scene. The default scene came with a directional light that represents the Sun, has a slightly yellowish color—FFF4D6 hexadecimal—and is rotated 50° around the X axis and −30° around the Y axis. If such a light doesn't exist create one.

To make the light's data accessible in the shader we'll have to create uniform values for it, just like for shader properties. In this case we'll define two `float3` vectors: `_DirectionalLightColor` and `_DirectionalLightDirection`. Put them in a `_CustomLight` buffer defined at the top of *Light*.

```
CBUFFER_START(_CustomLight)
	float3 _DirectionalLightColor;
	float3 _DirectionalLightDirection;
CBUFFER_END
```

Use these values instead of constants in `GetDirectionalLight`.

```
Light GetDirectionalLight () {
	Light light;
	light.color = _DirectionalLightColor;
	light.direction = _DirectionalLightDirection;
	return light;
}
```

Now our RP must send the light data to the GPU. We'll create a new `Lighting` class for that. It works like `CameraRenderer` but for lights. Give it a public `Setup` method with a context parameter, in which it invokes a separate `SetupDirectionalLight` method. Although not strictly necessary, let's also give it a dedicated command buffer that we execute when done, which can be handy for debugging. The alternative would be to add a buffer parameter.

```
using UnityEngine;
using UnityEngine.Rendering;

public class Lighting {

	const string bufferName = "Lighting";

	CommandBuffer buffer = new CommandBuffer {
		name = bufferName
	};

	public void Setup (ScriptableRenderContext context) {
		buffer.BeginSample(bufferName);
		SetupDirectionalLight();
		buffer.EndSample(bufferName);
		context.ExecuteCommandBuffer(buffer);
		buffer.Clear();
	}

	void SetupDirectionalLight () {}
}
```

Keep track of the identifiers of the two shader properties.

```
	static int
		dirLightColorId = Shader.PropertyToID("_DirectionalLightColor"),
		dirLightDirectionId = Shader.PropertyToID("_DirectionalLightDirection");
```

We can access the scene's main light via `RenderSettings.sun`. That gets us the most important directional light by default and it can also be explicitly configured via *Window / Rendering / Lighting Settings*. Use `CommandBuffer.SetGlobalVector` to send the light data to the GPU. The color is the light's color in linear space, while the direction is the light transformation's forward vector negated.

```
	void SetupDirectionalLight () {
		Light light = RenderSettings.sun;
		buffer.SetGlobalVector(dirLightColorId, light.color.linear);
		buffer.SetGlobalVector(dirLightDirectionId, -light.transform.forward);
	}
```

Doesn't `SetGlobalVector` require a `Vector4`?

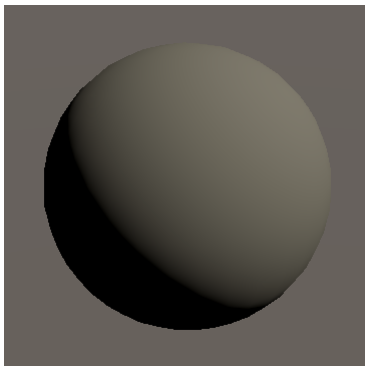
Yes, vectors send to the GPU always have four components, even if we define them with less. The extra components are implicitly masked out in the shader. Likewise, there's an implicit conversion from `Vector3` to `Vector4`, though not in the other direction.

The light's `color` property is its configured color, but lights also have a separate intensity factor. The final color is both multiplied.

```
		buffer.SetGlobalVector(
			dirLightColorId, light.color.linear * light.intensity
		);
```

Give `CameraRenderer` a `Lighting` instance and use it to set up the lighting before drawing the visible geometry.

```
	Lighting lighting = new Lighting();

	public void Render (
		ScriptableRenderContext context, Camera camera,
		bool useDynamicBatching, bool useGPUInstancing
	) {
		…

		Setup();
		lighting.Setup(context);
		DrawVisibleGeometry(useDynamicBatching, useGPUInstancing);
		DrawUnsupportedShaders();
		DrawGizmos();
		Submit();
	}
```



Lit by the sun.

2.4 Visible Lights

When culling Unity also figures out which lights affect the space visible to the camera. We can rely on that information instead of the global sun. To do so `Lighting` needs access to the culling results, so add a parameter for that to `Setup` and store it in a field for convenience. Then we can support more than one light, so replace the invocation of `SetupDirectionalLight` with a new `SetupLights` method.

```
	CullingResults cullingResults;

	public void Setup (
		ScriptableRenderContext context, CullingResults cullingResults
	) {
		this.cullingResults = cullingResults;
		buffer.BeginSample(bufferName);
		//SetupDirectionalLight();
		SetupLights();
		…
	}

	void SetupLights () {}
```

Add the culling results as an argument when invoking `Setup` in `CameraRenderer.Render`.

```
		lighting.Setup(context, cullingResults);
```

Now `Lighting.SetupLights` can retrieve the required data via the `visibleLights` property of the culling results. It's made available as a `Unity.Collections.NativeArray` with the `VisibleLight` element type.

```
using Unity.Collections;
using UnityEngine;
using UnityEngine.Rendering;

public class Lighting {
	…

	void SetupLights () {
		NativeArray<VisibleLight> visibleLights = cullingResults.visibleLights;
	}

	…
}
```

What's a `NativeArray`?

It's a struct that acts like an array, but provides a connection to a native memory buffer. It makes it possible to efficiently share data between managed C# code and the native Unity engine code.

2.5 Multiple Directional Lights

Using the visible light data makes it possible to support multiple directional lights, but we have to send the data of all those lights to the GPU. So instead of a pair of vectors we'll use two `Vector4` arrays, plus an integer for the light count. We'll also define a maximum amount of directional lights, which we can use to initialize two array fields to buffer the data. Let's set the maximum to four, which should be enough for most scenes.

```
	const int maxDirLightCount = 4;

	static int
		//dirLightColorId = Shader.PropertyToID("_DirectionalLightColor"),
		//dirLightDirectionId = Shader.PropertyToID("_DirectionalLightDirection");
		dirLightCountId = Shader.PropertyToID("_DirectionalLightCount"),
		dirLightColorsId = Shader.PropertyToID("_DirectionalLightColors"),
		dirLightDirectionsId = Shader.PropertyToID("_DirectionalLightDirections");

	static Vector4[]
		dirLightColors = new Vector4[maxDirLightCount],
		dirLightDirections = new Vector4[maxDirLightCount];
```

Why not use structured buffers?

That's possible, but I won't because shader support for structured buffers isn't good enough yet. Either they're not supported at all, are only in fragment programs, or perform worse than regular arrays. The good news is that the specifics of how data is passed between CPU and GPU only matter in a few places, so it's is easy to change. That's another benefit of using the `Light` struct.

Add an index and a `VisibleLight` parameter to `SetupDirectionalLight`. Have it set the color and direction elements with the supplied index. In this case the final color is provided via the `VisibleLight.finalColor` property. The forward vector can be found via the `VisibleLight.localToWorldMatrix` property. It's the third column of the matrix and once again has to be negated.

```
	void SetupDirectionalLight (int index, VisibleLight visibleLight) {
		dirLightColors[index] = visibleLight.finalColor;
		dirLightDirections[index] = -visibleLight.localToWorldMatrix.GetColumn(2);
	}
```

The final color already applied the light's intensity, but by default Unity doesn't convert it to linear space. We have to set `GraphicsSettings.lightsUseLinearIntensity` to `true`, which we can do once in the constructor of `CustomRenderPipeline`.

```
	public CustomRenderPipeline (
		bool useDynamicBatching, bool useGPUInstancing, bool useSRPBatcher
	) {
		this.useDynamicBatching = useDynamicBatching;
		this.useGPUInstancing = useGPUInstancing;
		GraphicsSettings.useScriptableRenderPipelineBatching = useSRPBatcher;
		GraphicsSettings.lightsUseLinearIntensity = true;
	}
```

Next, loop through all visible lights in `Lighting.SetupLights` and invoke `SetupDirectionalLight` for each element. Then invoke `SetGlobalInt` and `SetGlobalVectorArray` on the buffer to send the data to the GPU.

```
		NativeArray<VisibleLight> visibleLights = cullingResults.visibleLights;
		for (int i = 0; i < visibleLights.Length; i++) {
			VisibleLight visibleLight = visibleLights[i];
			SetupDirectionalLight(i, visibleLight);
		}

		buffer.SetGlobalInt(dirLightCountId, visibleLights.Length);
		buffer.SetGlobalVectorArray(dirLightColorsId, dirLightColors);
		buffer.SetGlobalVectorArray(dirLightDirectionsId, dirLightDirections);
```

But we only support up to four directional lights, so we should abort the loop when we reach that maximum. Let's keep track of the directional light index separate from the loop's iterator.

```
		int dirLightCount = 0;
		for (int i = 0; i < visibleLights.Length; i++) {
			VisibleLight visibleLight = visibleLights[i];
			SetupDirectionalLight(dirLightCount++, visibleLight);
			if (dirLightCount >= maxDirLightCount) {
				break;
			}
		}

		buffer.SetGlobalInt(dirLightCountId, dirLightCount);
```

Because we only support directional lights we should ignore other light types. We can do this by checking whether the `lightType` property of the visible lights is equal to `LightType.Directional`.

```
			VisibleLight visibleLight = visibleLights[i];
			if (visibleLight.lightType == LightType.Directional) {
				SetupDirectionalLight(dirLightCount++, visibleLight);
				if (dirLightCount >= maxDirLightCount) {
					break;
				}
			}
```

This works, but the `VisibleLight` struct is rather big. Ideally we only retrieve it once from the native array and don't also pass it as a regular argument to `SetupDirectionalLight`, because that copies it. We can use the same trick that Unity uses for the `ScriptableRenderContext.DrawRenderers` method, which is passing the argument by reference.

```
SetupDirectionalLight(dirLightCount++, ref visibleLight);
```

That requires us to also define the parameter as a reference.

```
void SetupDirectionalLight (int index, ref VisibleLight visibleLight) { … }
```

2.6 Shader Loop

Adjust the `_CustomLight` buffer in *Light* so it matches our new data format. In this case we'll explicitly use `float4` for the array types. Arrays have a fixed size in shaders, they cannot be resized. Make sure to use the same maximum that we defined in `Lighting`.

```
#define MAX_DIRECTIONAL_LIGHT_COUNT 4

CBUFFER_START(_CustomLight)
	//float4 _DirectionalLightColor;
	//float4 _DirectionalLightDirection;
	int _DirectionalLightCount;
	float4 _DirectionalLightColors[MAX_DIRECTIONAL_LIGHT_COUNT];
	float4 _DirectionalLightDirections[MAX_DIRECTIONAL_LIGHT_COUNT];
CBUFFER_END
```

Add a function to get the directional light count and adjust `GetDirectionalLight` so it retrieves the data for a specific light index.

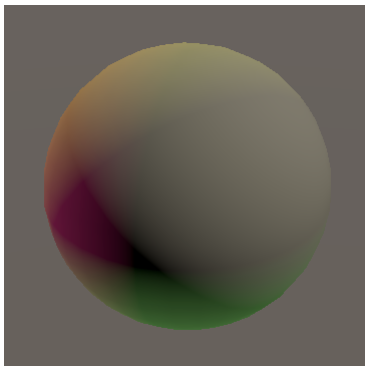
```
int GetDirectionalLightCount () {
	return _DirectionalLightCount;
}

Light GetDirectionalLight (int index) {
	Light light;
	light.color = _DirectionalLightColors[index].rgb;
	light.direction = _DirectionalLightDirections[index].xyz;
	return light;
}
```

Is there a difference between `rgb` and `xyz`?

They're semantic aliases. Swizzling using `rgba` and `xyzw` is equivalent.

Then adjust `GetLight` for a surface so it uses a `for` loop to accumulate the contribution of all directional lights.

```
float3 GetLighting (Surface surface) {
	float3 color = 0.0;
	for (int i = 0; i < GetDirectionalLightCount(); i++) {
		color += GetLighting(surface, GetDirectionalLight(i));
	}
	return color;
}
```



Four directional lights.

Now our shader supports up to four directional lights. Usually only a single directional light is needed to represent the Sun or Moon, but maybe there's a scene on a planet with multiple suns. Directional lights could also be used to approximate multiple large light rigs, for example those of a big stadium.

If your game always has a single directional light then you could get rid of the loop, or make multiple shader variants. But for this tutorial we'll keep it simple and stick to a single general-purpose loop. The best performance is always achieved by ripping out everything that you do not need, although it doesn't always make a significant difference.

2.7 Shader Target Level

Loops with a variable length used to be a problem for shaders, but modern GPUs can deal with them without issues, especially when all fragments of a draw call iterate over the same data in the same way. However, the OpenGL ES 2.0 and WebGL 1.0 graphics APIs can't deal with such loops by default. We could make it work by incorporating a hard-coded maximum, for example by having `GetDirectionalLight` return `min(_DirectionalLightCount, MAX_DIRECTIONAL_LIGHT_COUNT)`. This makes it possible to unroll the loop, turning it into a sequence of conditional code blocks. Unfortunately the resulting shader code is a mess and performance goes down fast. On very old-fashioned hardware all code blocks will always get executed, their contribution controlled via conditional assignments. While we could make this work it makes the code more complex, because we'd have to make other adjustments as well. So I opt to ignore these limitations and turn off WebGL 1.0 and OpenGL ES 2.0 support in builds for the sake of simplicity. They don't support linear lighting anyway. We can also avoid compiling OpenGL ES 2.0 shader variants for them by raising the target level of our shader pass to 3.5, via the `#pragma target 3.5` directive. Let's be consistent and do this for both shaders.

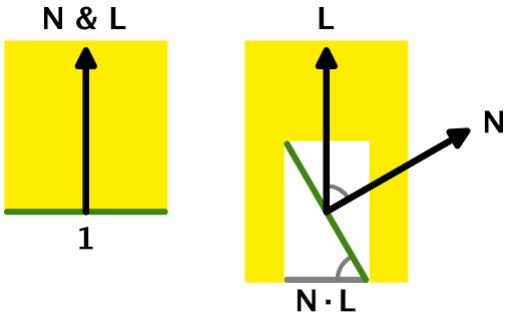
```
			HLSLPROGRAM
			#pragma target 3.5
			…
			ENDHLSL
```

3 BRDF

We're currently using a very simplistic lighting model, appropriate for perfectly diffuse surfaces only. We can achieve more varied and realistic lighting by applying a bidirectional reflectance distribution function, BRDF for short. There are many such functions. We'll use the same one that's used by the Universal RP, which trades some realism for performance.

3.1 Incoming Light

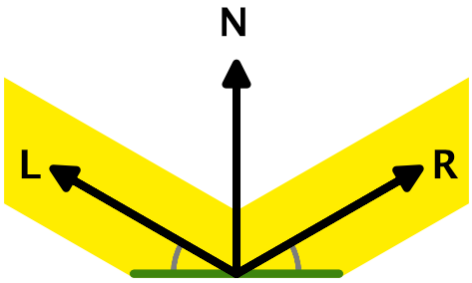
When a beam of light hits a surface fragment head-on then all its energy will affect the fragment. For simplicity we'll assume that the beam's width matches the fragment's. This is the case where the light direction L and surface normal N align, so $N \cdot L = 1$. When they don't align at least part of the beam misses the surface fragment, so less energy affects the fragment. The energy portion that affects the fragment is $N \cdot L$. A negative results means that the surface is oriented away from the light, so it cannot be affected by it.



Incoming light portion.

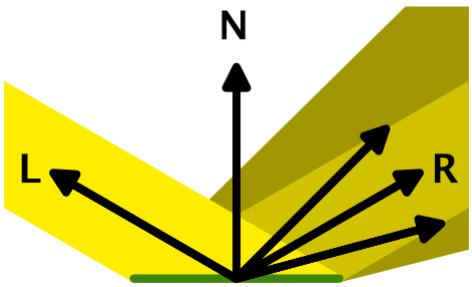
3.2 Outgoing Light

We don't see the light that arrives at a surface directly. We only see the portion that bounces off the surface and arrives at the camera or our eyes. If the surface were a perfectly flat mirror then the light would reflect off it, with an outgoing angle equal to the incoming angle. We would only see this light if the camera were aligned with it. This is known as specular reflection. It's a simplification of light interaction, but it's good enough for our purposes.



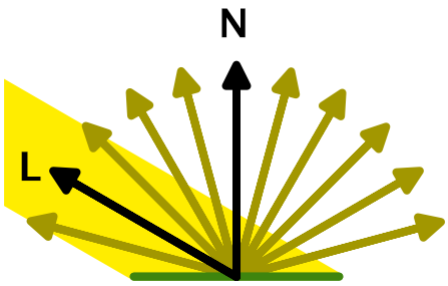
Perfect specular reflection.

But if the surface isn't perfectly flat then the light gets scattered, because the fragment effectively consists of many smaller fragments that have different orientations. This splits the beam of light into smaller beams that go in different directions, which effectively blurs the specular reflection. We could end up seeing some of the scattered light even when not aligned with the perfect reflection direction.



Scattered specular reflection.

Besides that, the light also penetrates the surface, bounces around, and exits at different angles, plus other things that we don't need to consider. Taken to the extreme, we end up with a perfectly diffuse surface that scatters light evenly in all possible directions. This is the lighting that we're currently calculating in our shader.



Perfect diffuse reflection.

No matter where the camera is, the amount of diffused light received from the surface is the same. But this means that the light energy that we observe is far less than the amount that arrived at the surface fragment. This suggests that we should scale the incoming light by some factor. However, because the factor is always the same we can bake it into the light's color and intensity. Thus the final light color that we use represents the amount observed when reflected from a perfectly white diffuse surface fragment illuminated head-on. This is a tiny fraction of the total amount of light that is actually emitted. There are other ways to configure lights, for example by specifying lumen or lux, which make it easier to configure realistic light sources, but we'll stick with the current approach.

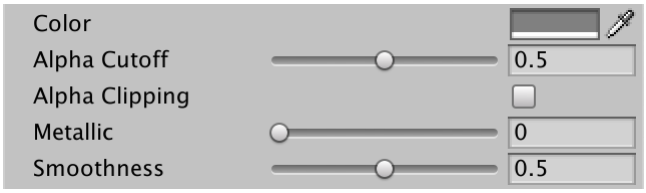
3.3 Surface Properties

Surfaces can be perfectly diffuse, perfect mirrors, or anything in between. There are multiple ways in which we could control this. We'll use the metallic workflow, which requires us to add two surface properties to the *Lit* shader.

The first property is whether a surface is metallic or nonmetalic, also known as a dielectric. Because a surface can contain a mix of both we'll add a range 0–1 slider for it, with 1 indicating that it is fully metallic. The default is fully dielectric.

The second property controls how smooth the surface is. We'll also use a range 0–1 slider for this, with 0 being perfectly rough and 1 being perfectly smooth. We'll use 0.5 as the default.

```
		_Metallic ("Metallic", Range(0, 1)) = 0
		_Smoothness ("Smoothness", Range(0, 1)) = 0.5
```



Material with metallic and smoothness sliders.

Add the properties to the `UnityPerMaterial` buffer.

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
	UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
	UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
	UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
	UNITY_DEFINE_INSTANCED_PROP(float, _Metallic)
	UNITY_DEFINE_INSTANCED_PROP(float, _Smoothness)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

And also to the `Surface` struct.

```
struct Surface {
	float3 normal;
	float3 color;
	float alpha;
	float metallic;
	float smoothness;
};
```

Copy them to the surface in `LitPassFragment`.

```
	Surface surface;
	surface.normal = normalize(input.normalWS);
	surface.color = base.rgb;
	surface.alpha = base.a;
	surface.metallic = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Metallic);
	surface.smoothness =
		UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Smoothness);
```

And also add support for them to `PerObjectMaterialProperties`.

```
	static int
		baseColorId = Shader.PropertyToID("_BaseColor"),
		cutoffId = Shader.PropertyToID("_Cutoff"),
		metallicId = Shader.PropertyToID("_Metallic"),
		smoothnessId = Shader.PropertyToID("_Smoothness");

	…

	[SerializeField, Range(0f, 1f)]
	float alphaCutoff = 0.5f, metallic = 0f, smoothness = 0.5f;

	…

	void OnValidate () {
		…
		block.SetFloat(metallicId, metallic);
		block.SetFloat(smoothnessId, smoothness);
		GetComponent<Renderer>().SetPropertyBlock(block);
	}
}
```

3.4 BRDF Properties

We'll use the surface properties to calculate the BRDF equation. It tells us how much light we end up seeing reflected off a surface, which is a combination of diffuse reflection and specular reflection. We need to split the surface color in a diffuse and a specular part, and we'll also need to know how rough the surface is. Let's keep track of these three values in a `BRDF` struct, put in a separate *BRDF* HLSL file.

```
#ifndef CUSTOM_BRDF_INCLUDED
#define CUSTOM_BRDF_INCLUDED

struct BRDF {
	float3 diffuse;
	float3 specular;
	float roughness;
};

#endif
```

Add a function to get the BRDF data for a given surface. Start with a perfect diffuse surface, so the diffuse part is equal to the surface color while specular is black and roughness is one.

```
BRDF GetBRDF (Surface surface) {
	BRDF brdf;
	brdf.diffuse = surface.color;
	brdf.specular = 0.0;
	brdf.roughness = 1.0;
	return brdf;
}
```

Include *BRDF* after *Light* and before *Lighting*.

```
#include "../ShaderLibrary/Common.hlsl"
#include "../ShaderLibrary/Surface.hlsl"
#include "../ShaderLibrary/Light.hlsl"
#include "../ShaderLibrary/BRDF.hlsl"
#include "../ShaderLibrary/Lighting.hlsl"
```

Add a `BRDF` parameter to both `GetLighting` functions, then multiply the incoming light with the diffuse portion instead of the entire surface color.

```
float3 GetLighting (Surface surface, BRDF brdf, Light light) {
	return IncomingLight(surface, light) * brdf.diffuse;
}

float3 GetLighting (Surface surface, BRDF brdf) {
	float3 color = 0.0;
	for (int i = 0; i < GetDirectionalLightCount(); i++) {
		color += GetLighting(surface, brdf, GetDirectionalLight(i));
	}
	return color;
}
```

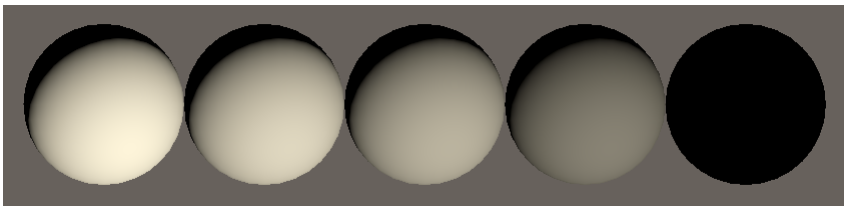
Finally, get the BRDF data in `LitPassFragment` and pass it to `GetLighting`.

```
	BRDF brdf = GetBRDF(surface);
	float3 color = GetLighting(surface, brdf);
```

3.5 Reflectivity

How reflective a surface is varies, but in general metals reflect all light via specular reflection and have zero diffuse reflection. So we'll declare reflectivity to be equal to the metallic surface property. Light that gets reflected doesn't get diffused, so we should scale the diffuse color by one minus the reflectivity in `GetBRDF`.

```
	float oneMinusReflectivity = 1.0 - surface.metallic;

	brdf.diffuse = surface.color * oneMinusReflectivity;
```



White spheres with metallic 0, 0.25, 0.5, 0.75, and 1.

In reality some light also bounces off dielectric surfaces, which gives them their highlight. The reflectivity of nonmetals varies, but is about 0.04 on average. Let's define that as the minimum reflectivity and add a `OneMinusReflectivity` function that adjusts the range from 0–1 to 0–0.96. This range adjustments matches the Universal RP's approach.

```
#define MIN_REFLECTIVITY 0.04

float OneMinusReflectivity (float metallic) {
	float range = 1.0 - MIN_REFLECTIVITY;
	return range - metallic * range;
}
```

Use that function in `GetBRDF` to enforce the minimum. The difference is hardly noticeable when only rendering the diffuse reflections, but will matter a lot when we add specular reflections. Without it nonmetals won't get specular highlights.

```
	float oneMinusReflectivity = OneMinusReflectivity(surface.metallic);
```

3.6 Specular Color

Light that gets reflected one way cannot also get reflected another way. This is known as energy conservation, which means that the amount of outgoing light cannot exceed the amount of incoming light. This suggests that the specular color should be equal to the surface color minus the diffuse color.

```
	brdf.diffuse = surface.color * oneMinusReflectivity;
	brdf.specular = surface.color - brdf.diffuse;
```

However, this ignores the fact that metals affect the color of specular reflections while nonmetals don't. The specular color of dielectric surfaces should be white, which we can achieve by using the metallic property to interpolate between the minimum reflectivity and the surface color instead.

```
	brdf.specular = lerp(MIN_REFLECTIVITY, surface.color, surface.metallic);
```

3.7 Roughness

Roughness is the opposite of smoothness, so we can simply take one minus smoothness. The *Core RP Library* has a function that does this, named `PerceptualSmoothnessToPerceptualRoughness`. We'll use this function, to make clear that the smoothness and thus also the roughness are defined as perceptual. We can convert to the actual roughness value via the `PerceptualRoughnessToRoughness` function, which squares the perceptual value. This matches the Disney lighting model. It's done this way because adjusting the perceptual version is more intuitive when editing materials.

```
	float perceptualRoughness =
		PerceptualSmoothnessToPerceptualRoughness(surface.smoothness);
	brdf.roughness = PerceptualRoughnessToRoughness(perceptualRoughness);
```

These functions are defined in the *CommonMaterial* HLSL file of the *Core RP Libary*. Include it in our *Common* file after including the core's *Common*.

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

3.8 View Direction

To determine how well the camera is aligned with the perfect reflection direction we'll need to know the camera's position. Unity makes this data available via `float3 _WorldSpaceCameraPos`, so add it to *UnityInput*.

```
float3 _WorldSpaceCameraPos;
```

To get the view direction—the direction from surface to camera—in `LitPassFragment` we need to add the world-space surface position to `Varyings`.

```
struct Varyings {
	float4 positionCS : SV_POSITION;
	float3 positionWS : VAR_POSITION;
	…
};

Varyings LitPassVertex (Attributes input) {
	…
	output.positionWS = TransformObjectToWorld(input.positionOS);
	output.positionCS = TransformWorldToHClip(output.positionWS);
	…
}
```

We'll consider the view direction to be part of the surface data, so add it to `Surface`.

```
struct Surface {
	float3 normal;
	float3 viewDirection;
	float3 color;
	float alpha;
	float metallic;
	float smoothness;
};
```

Assign it in `LitPassFragment`. It's equal to the camera position minus the fragment position, normalized.

```
	surface.normal = normalize(input.normalWS);
	surface.viewDirection = normalize(_WorldSpaceCameraPos - input.positionWS);
```

3.9 Specular Strength

The strength of the specular reflection that we observe depends on how well our view direction matches the perfect reflection direction. We'll use the same formula that's used in the Universal RP, which is a variant of the Minimalist CookTorrance BRDF. The formula contains a few squares, so let's add a convenient `Square` function to *Common* first.

```
float Square (float v) {
	return v * v;
}
```

Then add a `SpecularStrength` function to *BRDF* with a surface, BRDF data, and light as parameters. It should calculate $\frac{r^2}{d^2 \max\left(0.1, (L \cdot H)^2\right) n}$, where r is the roughness and all dot products should be saturated. Furthermore, $d = (N \cdot H)^2 (r^2 - 1) + 1.0001$, N is the surface normal, L is the light direction, and $H = L + V$ normalized, which is the halfway vector between the light and view directions. Use the `SafeNormalize` function to normalize that vector, to avoid a division by zero in case the vectors are opposed. Finally, $n = 4r + 2$ and is a normalization term.

```
float SpecularStrength (Surface surface, BRDF brdf, Light light) {
	float3 h = SafeNormalize(light.direction + surface.viewDirection);
	float nh2 = Square(saturate(dot(surface.normal, h)));
	float lh2 = Square(saturate(dot(light.direction, h)));
	float r2 = Square(brdf.roughness);
	float d2 = Square(nh2 * (r2 - 1.0) + 1.00001);
	float normalization = brdf.roughness * 4.0 + 2.0;
	return r2 / (d2 * max(0.1, lh2) * normalization);
}
```

How does that function work?

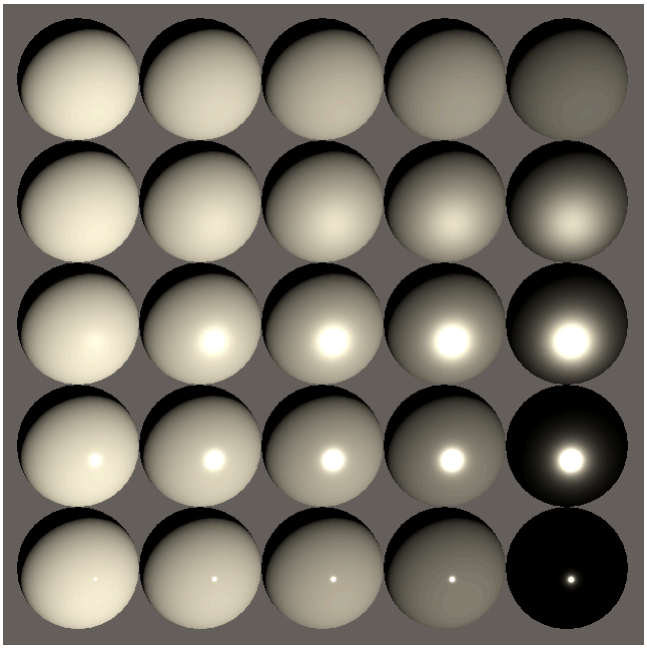
BRDF theory is too complex to fully explain in short and isn't the focus of this tutorial. You can check the *Lighting* HLSL file of the Universal RP for some code documentation and references.

Next, add a `DirectBRDF` that returns the color obtained via direct lighting, given a surface, BRDF, and light. The result is the specular color modulated by the specular strength, plus the diffuse color.

```
float3 DirectBRDF (Surface surface, BRDF brdf, Light light) {
	return SpecularStrength(surface, brdf, light) * brdf.specular + brdf.diffuse;
}
```

`GetLighting` then has to multiply the incoming light by the result of that function.

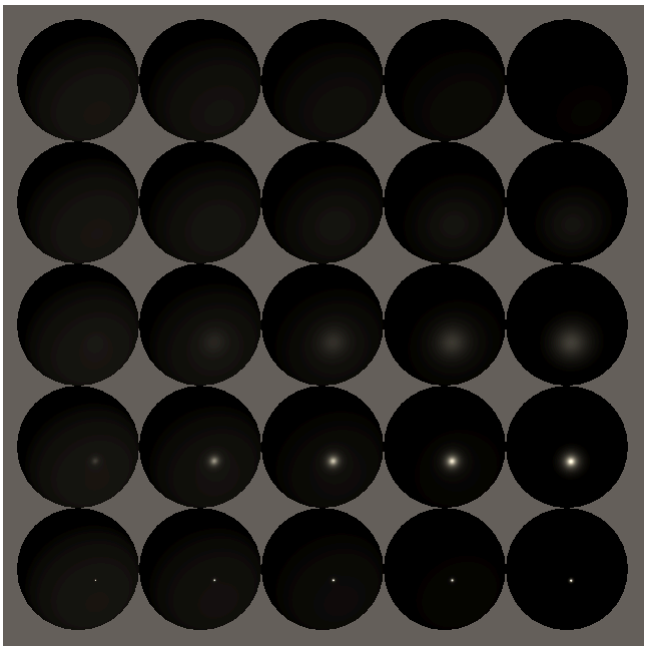
```
float3 GetLighting (Surface surface, BRDF brdf, Light light) {
	return IncomingLight(surface, light) * DirectBRDF(surface, brdf, light);
}
```



Smoothness top to bottom 0, 0.25, 0.5, 0.75, and 0.95.

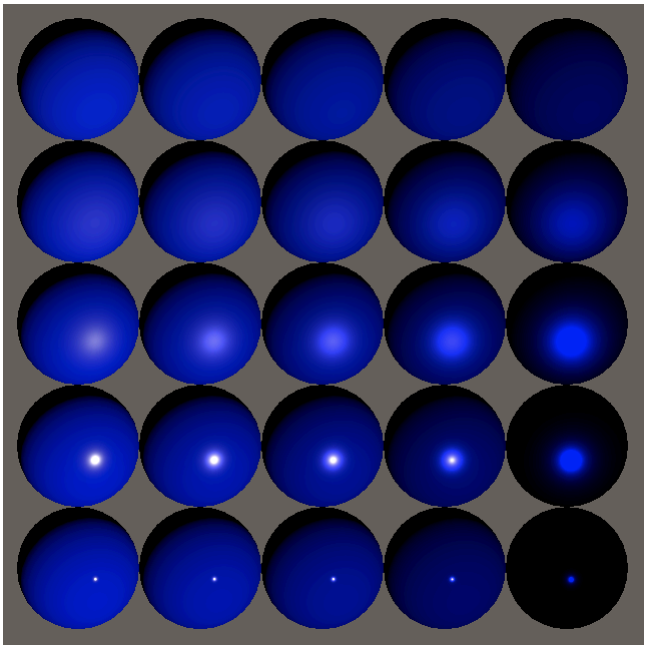
We now get specular reflections, which add highlights to our surfaces. For perfectly rough surfaces the highlight mimics diffuse reflection. Smoother surfaces get a more focused highlight. A perfectly smooth surface gets an infinitesimal highlight, which we cannot see. Some scattering is needed to make it visible.

Due to energy conservation highlights can get very bright for smooth surfaces, because most of the light arriving at the surface fragment gets focused. Thus we end up seeing far more light than would be possible due to diffuse reflection where the highlight is visible. You can verify this by scaling down the final rendered color a lot.



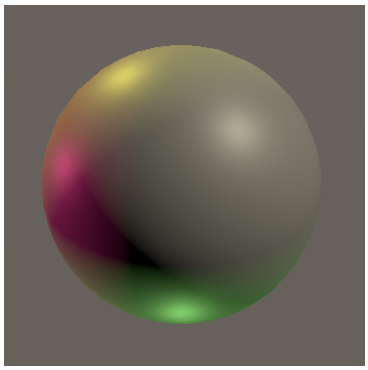
Final color divided by 100.

You can also verify that metals affect the color of specular reflections while nonmetals don't, by using a base color other than white.



Blue base color.

We now have functional direct lighting that is believable, although currently the results are too dark—especially for metals—because we don't support environmental reflections yet. A uniform black environment would be more realistic than the default skybox at this point, but that makes our objects harder to see. Adding more lights works as well.



Four lights.

3.10 Mesh Ball

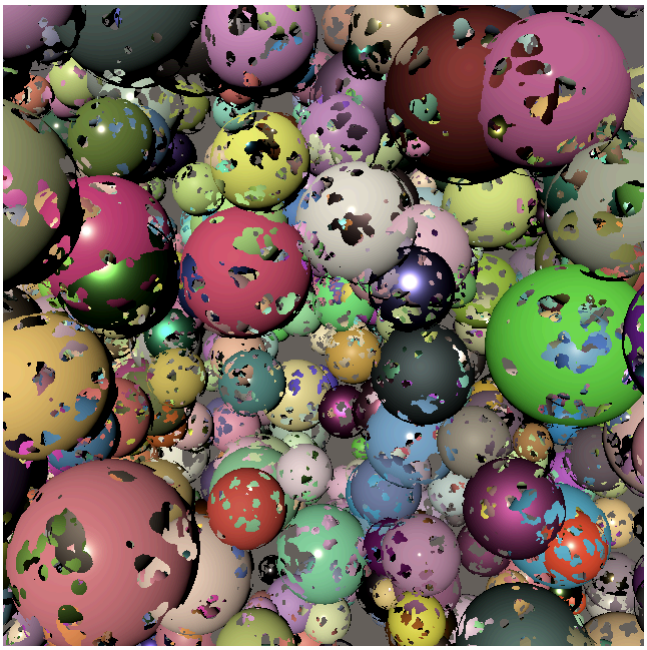
Let's also add support for varying metallic and smoothness properties to `MeshBall`. This requires adding two float arrays.

```
	static int
		baseColorId = Shader.PropertyToID("_BaseColor"),
		metallicId = Shader.PropertyToID("_Metallic"),
		smoothnessId = Shader.PropertyToID("_Smoothness");

	…
	float[]
		metallic = new float[1023],
		smoothness = new float[1023];

	…

	void Update () {
		if (block == null) {
			block = new MaterialPropertyBlock();
			block.SetVectorArray(baseColorId, baseColors);
			block.SetFloatArray(metallicId, metallic);
			block.SetFloatArray(smoothnessId, smoothness);
		}
		Graphics.DrawMeshInstanced(mesh, 0, material, matrices, 1023, block);
	}
```

Let's make 25% of the instances metallic and vary smoothness from 0.05 to 0.95 in `Awake`.

```
			baseColors[i] =
				new Vector4(
					Random.value, Random.value, Random.value,
					Random.Range(0.5f, 1f)
				);
			metallic[i] = Random.value < 0.25f ? 1f : 0f;
			smoothness[i] = Random.Range(0.05f, 0.95f);
```

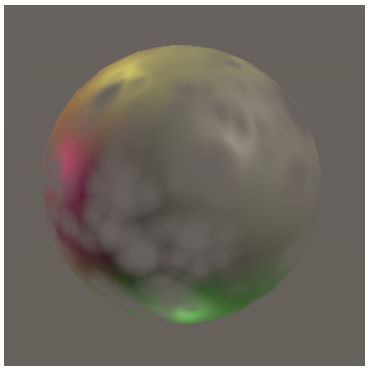
Then make the mesh ball use a lit material.



Lit mesh ball.

4 Transparency

Let's again consider transparency. Objects still fade based on their alpha value, but now it's the reflected light that fades. This makes sense for diffuse reflections, as only part of the light gets reflected while the rest travels through the surface.

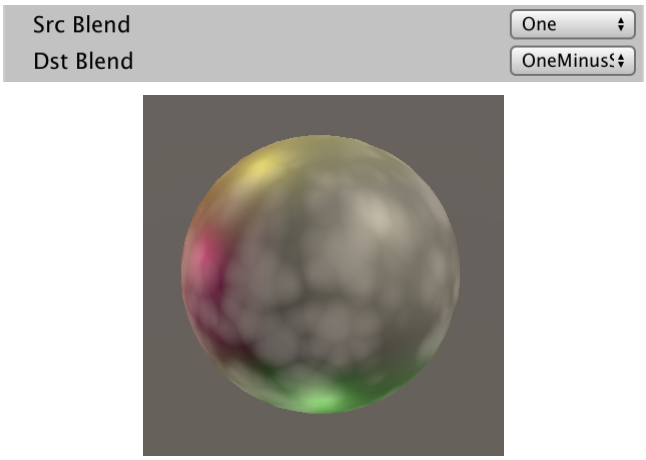


Fading sphere.

However, the specular reflections fade as well. In the case of a perfectly clear glass light either goes through or gets reflected. The specular reflections don't fade. We cannot represent this with our current approach.

4.1 Premultiplied Alpha

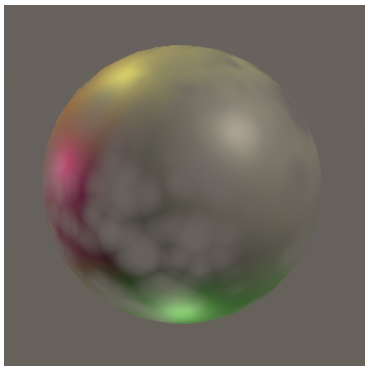
The solution is to only fade the diffuse light, while keeping the specular reflections at full strength. As the source blend mode applies to everything we cannot use it, so let's set it to one while still using one-minus-source-alpha for the destination blend mode.



Source blend mode set to one.

This restores specular reflections, but the diffuse reflections no longer fade out. We fix that by factoring the surface alpha into the diffuse color. Thus we premultiply diffuse by alpha, instead of relying on GPU blending later. This approach is known as premultiplied alpha blending. Do it in `GetBRDF`.

```
	brdf.diffuse = surface.color * oneMinusReflectivity;
	brdf.diffuse *= surface.alpha;
```



Premultiplied diffuse.

4.2 Premultiplication Toggle

Premultiplying alpha with diffuse effectively turns objects into glass, while regular alpha blending makes objects effectively exist only partially. Let's support both, by adding a boolean parameter to `GetBRDF` to control whether we premultiply alpha, set to false by default.

```
BRDF GetBRDF (inout Surface surface, bool applyAlphaToDiffuse = false) {
	…
	if (applyAlphaToDiffuse) {
		brdf.diffuse *= surface.alpha;
	}

	…
}
```

We can use a `_PREMULTIPLY_ALPHA` keyword to decide which approach to use in `LitPassFragment`, similar to how we control alpha clipping.

```
	#if defined(_PREMULTIPLY_ALPHA)
		BRDF brdf = GetBRDF(surface, true);
	#else
		BRDF brdf = GetBRDF(surface);
	#endif
	float3 color = GetLighting(surface, brdf);
	return float4(color, surface.alpha);
```

Add a shader feature for the keyword to the `Pass` of *Lit*.

```
			#pragma shader_feature _CLIPPING
			#pragma shader_feature _PREMULTIPLY_ALPHA
```

And add a toggle property to the shader as well.

```
		[Toggle(_PREMULTIPLY_ALPHA)] _PremulAlpha ("Premultiply Alpha", Float) = 0
```

Premultiply alpha toggle.

5 Shader GUI

We now support multiple rendering modes, each requiring specific settings. To make it easier to switch between modes let's add some buttons to our material inspector to apply preset configurations.

5.1 Custom Shader GUI

Add a `CustomEditor "CustomShaderGUI"` statement at the bottom of the main block of the *Lit* shader.

```
Shader "Custom RP/Lit" {
	…

	CustomEditor "CustomShaderGUI"
}
```

That instructs the Unity editor to use an instance of the `CustomShaderGUI` class to draw the inspector for materials that use the *Lit* shader. Create a script asset for that class and put it in a new *Custom RP / Editor* folder.

We'll need to use the `UnityEditor`, `UnityEngine`, and `UnityEngine.Rendering` namespaces. The class has to extend `ShaderGUI` and override the public `OnGUI` method, which has a `MaterialEditor` and a `MaterialProperty` array parameter. Have it invoke the base method, so we end up with the default inspector.

```
using UnityEditor;
using UnityEngine;
using UnityEngine.Rendering;

public class CustomShaderGUI : ShaderGUI {

	public override void OnGUI (
		MaterialEditor materialEditor, MaterialProperty[] properties
	) {
		base.OnGUI(materialEditor, properties);
	}
}
```

5.2 Setting Properties and Keywords

To do our work we'll need access to three things, which we'll store in fields. First is the material editor, which is the underlying editor object responsible for showing and editing materials. Second is a reference to the materials being edited, which we can retrieve via the `targets` property of the editor. It's defined as an `Object` array because `targets` is a property of the general-purpose `Editor` class. Third is the array of properties that can be edited.

```
	MaterialEditor editor;
	Object[] materials;
	MaterialProperty[] properties;

	public override void OnGUI (
		MaterialEditor materialEditor, MaterialProperty[] properties
	) {
		base.OnGUI(materialEditor, properties);
		editor = materialEditor;
		materials = materialEditor.targets;
		this.properties = properties;
	}
```

Why are there multiple materials?

Multiple materials that use the same shader can be edited at the same time, just like you can select and edit multiple game objects.

To set a property we first have to find it in the array, for which we can use the `ShaderGUI.FindPropery` method, passing it a name and property array. We can then adjust its value, by assigning to its `floatValue` property. Encapsulate this in a convenient `SetProperty` method with a name and a value parameter.

```
	void SetProperty (string name, float value) {
		FindProperty(name, properties).floatValue = value;
	}
```

Settings a keyword is a bit more involved. We'll create a `SetKeyword` method for this, with a name and a boolean parameter to indicate whether the keyword should be enabled or disabled. We have to invoke either `EnableKeyword` or `DisableKeyword` on all materials, passing them the keyword name.

```
	void SetKeyword (string keyword, bool enabled) {
		if (enabled) {
			foreach (Material m in materials) {
				m.EnableKeyword(keyword);
			}
		}
		else {
			foreach (Material m in materials) {
				m.DisableKeyword(keyword);
			}
		}
	}
```

Let's also create a `SetProperty` variant that toggles a property–keyword combination.

```
	void SetProperty (string name, string keyword, bool value) {
		SetProperty(name, value ? 1f : 0f);
		SetKeyword(keyword, value);
	}
```

Now we can define simple `Clipping`, `PremultiplyAlpha`, `SrcBlend`, `DstBlend`, and `ZWrite` setter properties.

```
	bool Clipping {
		set => SetProperty("_Clipping", "_CLIPPING", value);
	}

	bool PremultiplyAlpha {
		set => SetProperty("_PremulAlpha", "_PREMULTIPLY_ALPHA", value);
	}

	BlendMode SrcBlend {
		set => SetProperty("_SrcBlend", (float)value);
	}

	BlendMode DstBlend {
		set => SetProperty("_DstBlend", (float)value);
	}

	bool ZWrite {
		set => SetProperty("_ZWrite", value ? 1f : 0f);
	}
```

Finally, the render queue is set by assigning to the `RenderQueue` property of all materials. We can use the `RenderQueue` enum for this.

```
	RenderQueue RenderQueue {
		set {
			foreach (Material m in materials) {
				m.renderQueue = (int)value;
			}
		}
	}
```

5.3 Preset Buttons

A button can be created via the `GUILayout.Button` method, passing it a label, which will be a preset's name. If the method returns `true` then it was pressed. Before applying the preset we should register an undo step with the editor, which can be done by invoking `RegisterPropertyChangeUndo` on it with the name. As this code is the same for all presets, put it in a `PresetButton` method that returns whether the preset should be applied.

```
	bool PresetButton (string name) {
		if (GUILayout.Button(name)) {
			editor.RegisterPropertyChangeUndo(name);
			return true;
		}
		return false;
	}
```

We'll create a separate method per preset, beginning with the default *Opaque* mode. Have it set the properties appropriately when activated.

```
	void OpaquePreset () {
		if (PresetButton("Opaque")) {
			Clipping = false;
			PremultiplyAlpha = false;
			SrcBlend = BlendMode.One;
			DstBlend = BlendMode.Zero;
			ZWrite = true;
			RenderQueue = RenderQueue.Geometry;
		}
	}
```

The second preset is *Clip*, which is a copy of *Opaque* with clipping turned on and the queue set to *AlphaTest*.

```
	void ClipPreset () {
		if (PresetButton("Clip")) {
			Clipping = true;
			PremultiplyAlpha = false;
			SrcBlend = BlendMode.One;
			DstBlend = BlendMode.Zero;
			ZWrite = true;
			RenderQueue = RenderQueue.AlphaTest;
		}
	}
```

The third preset is for standard transparency, which fades out objects so we'll name it *Fade*. It's another copy of *Opaque*, with adjusted blend modes and queue, plus no depth writing.

```
	void FadePreset () {
		if (PresetButton("Fade")) {
			Clipping = false;
			PremultiplyAlpha = false;
			SrcBlend = BlendMode.SrcAlpha;
			DstBlend = BlendMode.OneMinusSrcAlpha;
			ZWrite = false;
			RenderQueue = RenderQueue.Transparent;
		}
	}
```

The fourth preset is a variant of *Fade* that applies premultiplied alpha blending. We'll name it *Transparent* as it's for semitransparent surfaces with correct lighting.

```
	void TransparentPreset () {
		if (PresetButton("Transparent")) {
			Clipping = false;
			PremultiplyAlpha = true;
			SrcBlend = BlendMode.One;
			DstBlend = BlendMode.OneMinusSrcAlpha;
			ZWrite = false;
			RenderQueue = RenderQueue.Transparent;
		}
	}
```

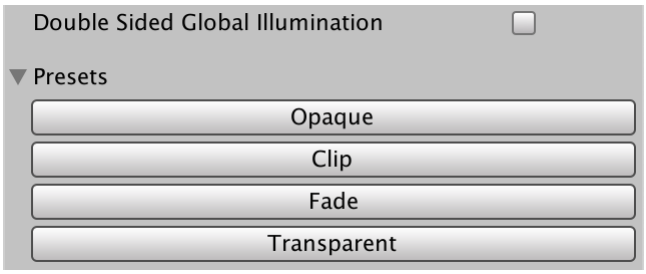
Invoke the preset methods at the end of `OnGUI` so they show up below the default inspector.

```
	public override void OnGUI (
		MaterialEditor materialEditor, MaterialProperty[] properties
	) {
		…

		OpaquePreset();
		ClipPreset();
		FadePreset();
		TransparentPreset();
	}
```

Preset buttons.

The preset buttons won't be used often, so let's put them inside a foldout that is collapsed by default. This is done by invoking `EditorGUILayout.Foldout` with the current foldout state, label, and `true` to indicate that clicking it should toggle its state. It returns the new foldout state, which we should store in a field. Only draw the buttons when the foldout is open.

```
	bool showPresets;

	…

	public override void OnGUI (
		MaterialEditor materialEditor, MaterialProperty[] properties
	) {
		…

		EditorGUILayout.Space();
		showPresets = EditorGUILayout.Foldout(showPresets, "Presets", true);
		if (showPresets) {
			OpaquePreset();
			ClipPreset();
			FadePreset();
			TransparentPreset();
		}
	}
```



Preset foldout.

5.4 Presets for Unlit

We can also use the custom shader GUI for our *Unlit* shader.

```
Shader "Custom RP/Unlit" {
	…

	CustomEditor "CustomShaderGUI"
}
```

However, activating a preset will result in an error, because we're trying to set a property that the shader doesn't have. We can guard against that by adjusting `SetProperty`. Have it invoke `FindProperty` with `false` as an additional argument, indicating that it shouldn't log an error if the property isn't found. The result will then be `null`, so only set the value if that's not the case. Also return whether the property exists.

```
	bool SetProperty (string name, float value) {
		MaterialProperty property = FindProperty(name, properties, false);
		if (property != null) {
			property.floatValue = value;
			return true;
		}
		return false;
	}
```

Then adjust the keyword version of `SetProperty` so it only sets the keyword if the relevant property exists.

```
	void SetProperty (string name, string keyword, bool value) {
		if (SetProperty(name, value ? 1f : 0f)) {
			SetKeyword(keyword, value);
		}
	}
```

5.5 No Transparency

Now the presets also work for materials that use the *Unlit* shader, although the *Transparent* mode doesn't make much sense in this case, as the relevant property doesn't exist. Let's hide this preset when it's not relevant.

First, add a `HasProperty` method that returns whether a property exists.

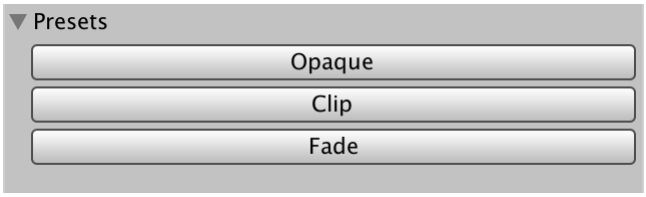
```
	bool HasProperty (string name) =>
		FindProperty(name, properties, false) != null;
```

Second, create a convenient property to check whether *_PremultiplyAlpha* exists.

```
	bool HasPremultiplyAlpha => HasProperty("_PremulAlpha");
```

Finally, make everything of the *Transparent* preset conditional on that property, by checking it first in `TransparentPreset`.

```
		if (HasPremultiplyAlpha && PresetButton("Transparent")) { … }
```



Unlit materials lack Transparent preset.

The next tutorial is Directional Shadows.

license

repository

Enjoying the tutorials? Are they useful?

Please support me on Patreon or Ko-fi!

Or make a direct donation!

made by Jasper Flick