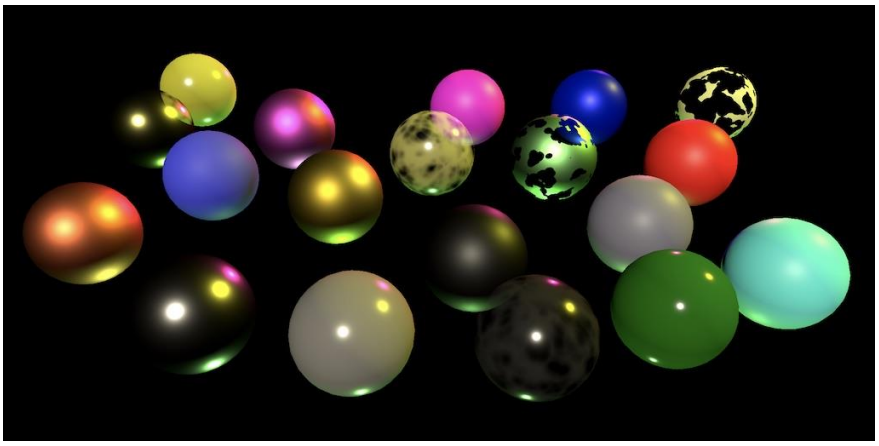
平行光　　*直接光照*

使用法线向量计算光照
支持四个平行光
采用 BRDF（双向反射分布函数）
制作带有预设的自定义着色器 GUI

这是关于创建自定义渲染管线系列教程的第三部分。它添加了对多平行光着色的支持。

本教程基于 Unity 2019.2.12f1。



受到四个光照的各种球体

1. 光照

如果我们想创建一个更真实的场景，就需要模拟光照在表面的反射。这需要一个比我们目前的不受光着色器更复杂的着色器。

1.1. 光照着色器

复制 UnlitPass.hlsl 并将其重命名为 LitPass。调整引用保护定义，并将顶点和片元函数对应修改。我们之后会添加光照计算。

```
1. #ifndef CUSTOM_LIT_PASS_INCLUDED
2. #define CUSTOM_LIT_PASS_INCLUDED
3.
4. …
5.
6. Varyings LitPassVertex (Attributes input) { … }
```

```
7.
8.  float4 LitPassFragment (Varyings input) : SV_TARGET { … }
9.
10. #endif
```

同样地，复制 Unlit 着色器并重命名为 Lit。更改它的菜单名，包含的文件和使用的函数。再把默认颜色改成灰色，因为纯白色表面在一个良好光照的场景中会非常亮。URP 也使用了灰色作为默认颜色。

```
1.  Shader "Custom RP/Lit" {
2.
3.      Properties {
4.          _BaseMap("Texture", 2D) = "white" {}
5.          _BaseColor("Color", Color) = (0.5, 0.5, 0.5, 1.0)
6.          …
7.      }
8.
9.      SubShader {
10.         Pass {
11.             …
12.             #pragma vertex LitPassVertex
13.             #pragma fragment LitPassFragment
14.             #include "LitPass.hlsl"
15.             ENDHLSL
16.         }
17.     }
18. }
```

我们将使用一种自定义的照明方法，这通过将着色器的光照模式设置成 CustomLit 来声明。给 Pass 添加 Tags 块，包含"LightMode" = “CustomLit”。

```
1.  Pass {
2.      Tags {
3.          "LightMode" = "CustomLit"
4.      }
5.
6.      …
7.  }
```

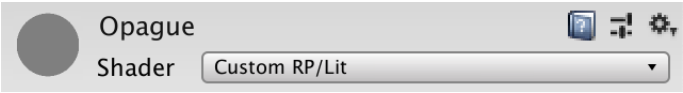
为了渲染使用该通道的对象，我们必须把它放到 CameraRenderer 里。首先新增一个着色器标签 ID。

```
1.  static ShaderTagId
2.      unlitShaderTagId = new ShaderTagId("SRPDefaultUnlit"),
3.      litShaderTagId = new ShaderTagId("CustomLit");
```

然后将它添加到在 DrawVisibleGeometry 渲染的通道中，向我们在 DrawUnsupportedShaders 里做的那样。

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

现在我们可以创建一个新的不透明材质，尽管现在它的效果和不受光材质一样。



默认不透明材质

1.2. 法向量

对象的光照程度取决于多个因素，其中包括光线与表面的夹角。为了知道表面朝向，我们需要调用表面法线，这是一个垂直于表面的单位向量。这个向量是顶点数据的一部分，像坐标一样定义在物体空间。所以把它添加到 LitPass 的 Attributes 里。

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

光照是逐片元计算的，所以我们也需要把法向量添加到 Varyings。我们要在世界空间进行光照计算，所以把它命名为 normalWS。

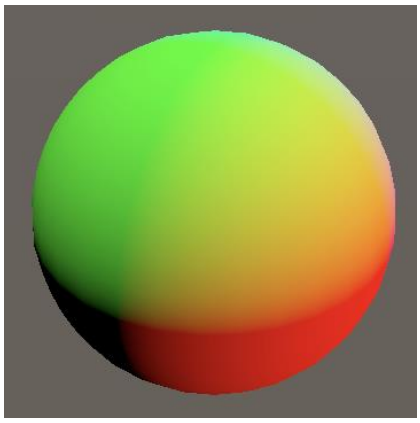
```
struct Varyings {
	float4 positionCS : SV_POSITION;
	float3 normalWS : VAR_NORMAL;
	float2 baseUV : VAR_BASE_UV;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

我们可以在 LitPassVertex 里用 SpaceTransforms 中的 TransformObjectToWorldNormal 将法向量转换到世界空间。

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

为了检查我们是否在 LitPassFragment 里得到了正确的法向量，我们可以把它当作颜色使用。

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



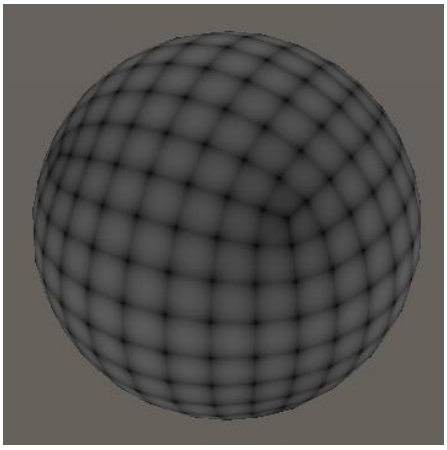
世界空间法向量

负值无法可视化，所以被限制为 0。

1.3. 插补法线

尽管法向量在顶点程序中是单位长度，三角形间的线性插补会影响它们的长度。我们可以把向量长度和 1 之间的差距扩大 10 倍后渲染出来，用来看到这种错误，

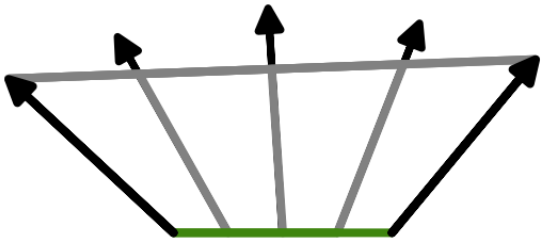
```
base.rgb = abs(length(input.normalWS) - 1.0) * 10.0;
```



放大后的插补法线错误

我们可以在 LitPassFragment 里通过将法向量归一化来平滑插补失真。其中的差别在观察法向量时不是很明显，但是在用于光照时会变得更明显。

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



插补后进行归一化

1.4. 表面属性

着色器中的光照是对光线照射到物体表面进行相互作用的模拟，这就意味着我们必须关注表面属性。现在我们有法向量和基础颜色。我们可以把后者分为两部分：RGB 颜色和 alpha 值。我们将要在若干个地方用到这个数据，所以我们定义一个方便的 Surface 结构体来包含所有相关数据。把它放在 ShaderLibrary 文件夹下一个单独的 Surface.hlsl 文件里。

```
1.  #ifndef CUSTOM_SURFACE_INCLUDED
2.  #define CUSTOM_SURFACE_INCLUDED
3.
4.  struct Surface {
5.      float3 normal;
6.      float3 color;
7.      float alpha;
8.  };
9.
10. #endif
```

在 LitPass 的 Common 之后把它引入。我们可以通过这种方法把 LitPass 保持得比较短。从现在开始我们将把特定的代码放在它自己的 hlsl 文件中，这样更容易定位相关的功能。

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

在 LitPassFragment 里定义一个 surface 变量并赋值。然后把最终结果改为 surface 的 color 和 alpha。

```
1.  Surface surface;
2.  surface.normal = normalize(input.normalWS);
```

```
3.  surface.color = base.rgb;
4.  surface.alpha = base.a;
5.
6.  return float4(surface.color, surface.alpha);
```

1.5. 计算光照

为了计算实际光照，我们要创建一个带 Surface 参数的 GetLighting 方法。首先让它返回表面发现的 Y 分量。因为这是光照功能，所以我们把它放到单独的 Lighting.hlsl 文件里。

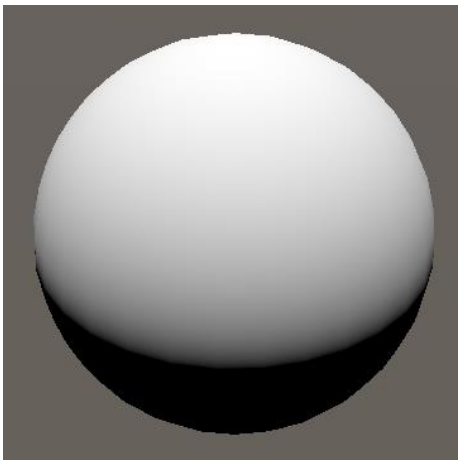
```
1.  #ifndef CUSTOM_LIGHTING_INCLUDED
2.  #define CUSTOM_LIGHTING_INCLUDED
3.
4.  float3 GetLighting (Surface surface) {
5.      return surface.normal.y;
6.  }
7.
8.  #endif
```

在 LitPass 的 Surface 之后引入该文件，因为 Lighting 依赖于 Surface。

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

现在我们可以在 LitPassFragment 中获取光照，并且用它来计算片元的 RGB 部分。

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

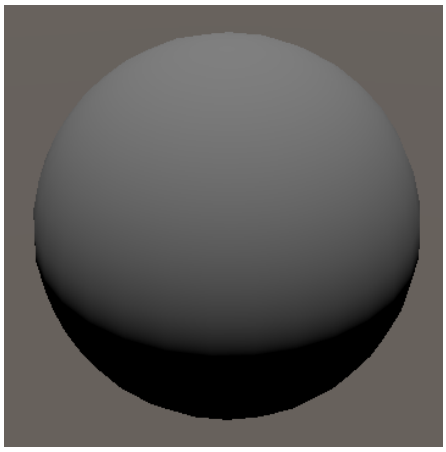


上方的漫反射光照

目前计算结果是表面法线的 Y 分量，所以它在球体顶部为 1，侧面的时候减少到 0。再往下这个值就变成负的，底端的值为-1，但我们无法看到负值。它与法线和向上向量的夹角余弦

相匹配。抛开负值以外，这个看起来和指向下方的漫反射平行光照相匹配。最后一步是把表面颜色的影响纳入 GetLighting 的结果中，可以将其解释为表面反照率（albedo）。

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



使用了反照率

2. 光源

为了表现适当的照明，我们还需要知道光源的属性。在这篇教程里我们将只是用平行光。平行光表现为一个足够远的光源以至于它的位置不重要，只需要关心它的方向。这是一种简化处理，但对于模拟地球上的太阳光或者其他差不多是单向光的场景已经足够了。

2.1. 光源结构

我们要使用一个结构体来存储光线数据。目前一个颜色和一个方向就可以满足要求。把它放在单独的 Light.hlsl 文件里。同样定义一个 GetDirectionalLight 方法返回一个配置好的光源。首先使用白色和向上的分量，这与我们目前使用的光源数据相匹配。需要注意的是光源方向是定义为光线来源方向，而不是照射方向。

```
#ifndef CUSTOM_LIGHT_INCLUDED
#define CUSTOM_LIGHT_INCLUDED

struct Light {
    float3 color;
    float3 direction;
};
```

```
8.
9.  Light GetDirectionalLight () {
10.     Light light;
11.     light.color = 1.0;
12.     light.direction = float3(0.0, 1.0, 0.0);
13.     return light;
14. }
15.
16. #endif
```

把这个文件在 LitPass 的 Lighting 之前引入。

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

2.2. 光照函数

在 Lighting 里添加 GetIncomingLight 方法用来计算给定表面和光源的入射光强度。对于任意的光源方向，我们要使用表面法线和光源方向的点积。我们可以用 dot 函数来进行计算。将结果用光源颜色进行调制。

```
1.  float3 GetIncomingLight (Surface surface, Light light) {
2.      return dot(surface.normal, light.direction) * light.color;
3.  }
```

但这只有在表面面向光源时才正确。当点积为负时，我们需要将结果限制为 0，这可以通过 saturate 函数来实现。

```
1.  float3 IncomingLight (Surface surface, Light light) {
2.      return saturate(dot(surface.normal, light.direction)) * light.color;
3.  }
```

添加另一个 GetLighting 函数，用来返回表面和光线的最终光照。现在它是叠加了表面颜色的入射光。在其他方法之前定义它。

```
1.  float3 GetLighting (Surface surface, Light light) {
2.      return IncomingLight(surface, light) * surface.color;
3.  }
```

最后，调整只有一个 surface 参数的 GetLighting 方法，让它调用另一个 GetLighting，用 GetDirectionalLight 来提供光源数据。

```
1.  float3 GetLighting (Surface surface) {
2.      return GetLighting(surface, GetDirectionalLight());
3.  }
```

2.3. 发送光源数据到 GPU

我们应该使用当前场景中的光源，而不是总用上方的白光。默认场景带有一个代表太阳的方向光，略带黄色（十六进制 FFF4D6），并且绕 X 轴旋转 50°，绕 Y 轴旋转-30°。如果没有这样的光源，就新建一个。

为了让光源数据在着色器中可以被调用，我们要像着色器属性一样为它创建统一值。这种情况下我们要定义两个 float3 向量：_DirectionalLightColor 和_DirectionalLightDirection。把它们放到在 Light 开头定义的_CustomLight 缓冲区里。

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

用这些值代替 GetDirectionalLight 里的常量。

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

现在我们的渲染管线必须将光源数据发送给 GPU。我们创建一个 Lighting 类来进行处理。原理类似 CameraRenderer，只不过是用于光源。给它添加一个公共的带有上下文参数的 Setup 方法，在其中调用一个单独的 SetDirectionalLight 方法。尽管不是绝对必要，我们也给它创建一个专用的命令缓冲区，这会方便调试。另一个选择是添加一个缓冲区参数。

```
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);
```

```
16.         context.ExecuteCommandBuffer(buffer);
17.         buffer.Clear();
18.     }
19.
20.     void SetupDirectionalLight () {}
21. }
```

获取这两个着色器属性的 ID。

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

我们可以通过 RenderSettings.sun 来调用场景的主光源。这是我们默认情况下使用的最重要的平行光，并且可以通过 Window/Rendering/Lighting Settings 来显式配置。使用 CommandBuffer.SetGlobalVector 将光源数据发送到 GPU。颜色对应光源的线性空间颜色，同时方向对应光源坐标系 Z 轴负方向。

```
1.  void SetupDirectionalLight () {
2.      Light light = RenderSettings.sun;
3.      buffer.SetGlobalVector(dirLightColorId, light.color.linear);
4.      buffer.SetGlobalVector(dirLightDirectionId, -light.transform.forward);
5.  }
```

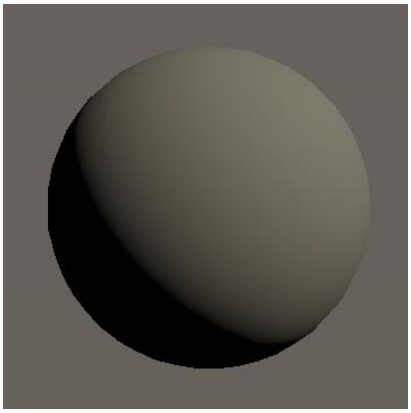
光源的 color 属性就是配置的颜色，但光源同样还有一个单独的强度参数。最终的颜色需要两者相乘。

```
1.  buffer.SetGlobalVector(
2.      dirLightColorId, light.color.linear * light.intensity
3.  );
```

在 CameraRenderer 创建一个 Lighting 实例，并且用它在绘制可见几何对象之前设置光源。

```
1.  Lighting lighting = new Lighting();
2.
3.  public void Render (
4.      ScriptableRenderContext context, Camera camera,
5.      bool useDynamicBatching, bool useGPUInstancing
6.  ) {
7.      …
8.
9.      Setup();
10.     lighting.Setup(context);
11.     DrawVisibleGeometry(useDynamicBatching, useGPUInstancing);
12.     DrawUnsupportedShaders();
```

```
13.     DrawGizmos();
14.     Submit();
15. }
```



受到太阳光照

2.4. 可见光

当进行剔除时，Unity 也会计算那些光源对摄像机可见空间有影响。我们可以依赖这个数据来代替全局阳光。为了实现这个功能，Lighting 需要调用剔除结果，所以在 Setup 里为其添加一个参数，并为了方便起见，将其存为字段。然后我们可以支持多余一个光源，所以把调用 SetupDirectionalLight 的地方改成一个新的 SetupLights 方法。

```
1.  CullingResults cullingResults;
2.
3.  public void Setup (
4.      ScriptableRenderContext context, CullingResults cullingResults
5.  ) {
6.      this.cullingResults = cullingResults;
7.      buffer.BeginSample(bufferName);
8.      //SetupDirectionalLight();
9.      SetupLights();
10.     …
11. }
12.
13. void SetupLights () {}
```

当在 CameraRenerer.Render 里调用 Setup 时，把剔除结果作为参数。

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

现在 Lighting.SetupLIghts 可以通过剔除结果的 visibleLights 属性来获取所需要的数据。这是一种带有 VisibleLight 元素类型的 Unity.Collections.NativeArray。

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

public class Lighting {
	…

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

	…
}
```

2.5. 多平行光

使用可见光数据让我们可以支持多平行光，但我们必须将所有光源的数据发送到 GPU。所以我们将使用两个长度 Vector4 数组和一个代表光源数量的整数来代替一对向量。同样我们要定义平行光的最大数量，用来初始化缓冲数据的两个数组字段。我们把最大数量设置为 4，对于大多数场景来说足够了。

```
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];
```

给 SetupDirectionalLight 添加索引和 VisibleLight 参数，用提供的索引设置颜色和方向参数。目前最终颜色由 VisibleLight.finalColor 属性提供。前向量可以在 VisibleLight.localToWorldMatrix 属性里找到，它是矩阵的第三列，并且同样地取负。

```
void SetupDirectionalLight (int index, VisibleLight visibleLight) {
	dirLightColors[index] = visibleLight.finalColor;
```

```
        dirLightDirections[index] = -
    visibleLight.localToWorldMatrix.GetColumn(2);
}
```

finalColor 已经计算了光强，但默认情况下 Unity 不会将其转换到线性空间。我们需要把 GraphicsSettings.lightsUseLinearIntensity 设置为 true，这一步可以在 CustomRenderPipeline 的构造函数里执行一次。

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

接下来，在 Lighting.SetupLights 里循环所有可见光，并且对每个元素执行 SetupDirectionalLight。然后调用缓冲区的 SetGlobalInt 和 SetGlobalVectorArray 将数据上传给 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);
```

但我们只支持四个平行光，所以当到达最大数量时要中止循环。让我们在循环迭代器外单独记录平行光索引。

```
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);
```

因为我们只支持平行光，我们应当忽略其他的光源类型。我们可以通过检测可见光的 lightType 属性是否等于 LightType.Directional 来实现。

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

现在可以了，但 VisibleLight 结构相当大。理想情况下我们只需从 NativeArray 里获取一次，并且也不要将它作为常规参数传给 SetupDirectionalLight，因为那样会复制一份。我们可以使用像 Unity 在 ScriptableRenderContext.DrawRenderers 里那样的技巧，传递一个引用参数。

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

这要求我们也要变参数定为引用。

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

2.6. 着色器循环

调整 Light 里的_CustomLight 缓冲区，使其与新的数据格式匹配。这种情况下，我们需要用 float4 作为数组类型。着色器中的数组长度固定，不能被更改。保证使用了和 Lighting 中定义的相同的最大数量。

```
#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
```

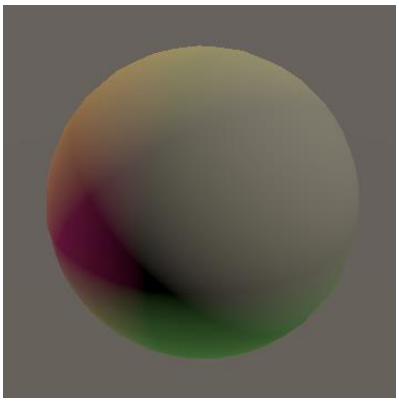
添加一个函数来获取平行光数量，并且调整 GetDirectionalLIght，使其可以获取特定光源索引的数据。

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

```

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

然后调整 suface 参数的 GetLighting，使用 for 循环来叠加所有平行光的贡献。

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



四个方向光

现在我们的着色器支持四个方向光。通常只需要一个方向光来表现太阳或者月亮，但可能有一个场景是有多个太阳的行星。方向光也可以用来近似多个大型照明设备，比如大型体育场的照明。

如果你的游戏始终只有一个方向光，那你就可以避免使用循环，或者使用多个着色器变体。但对这篇教程来说，我们让它通用一些，并只使用单个通用循环。最好的性能总是通过去掉所有不需要的内容实现的，尽管并不总是带来很大的不同。

2.7. 着色器目标等级

可变长度的循环以前对于着色器来说是个问题，但是现代 GPU 可以毫无问题的处理它们，特别是当一次绘制的所有片元通过同样的方式迭代相同数据时。然后，OpenGL ES 2.0 和 WebGL 1.0 默认情况下无法处理这些循环。我们可以与硬编码的最大值合并使其生效，比如让 GetDirectionalLight 返回 min(_DirectionalLightCount, MAX_DIRECTIONAL_LIGHT_COUNT)。

这使得它可以展开循环，将其转变为一个条件代码块序列。不幸的是最终的着色器代码变得一团糟，同时性能也下降得很快。在非常老式的硬件上，所有的代码块都将始终执行，它们的贡献通过条件分配来控制。尽管我们可以使其生效，但代码会变得更复杂，因为哦我们还必须做其他调整。所以为了简单起见，我选择忽略这些限制，并取消对 WebGL 1.0 和 OpenGL ES 2.0 的支持。他们也同样不支持线性光照。我们也可以通过把我们的着色器通道的目标等级提升到 3.5，来避免编译 OpenGL ES 2.0 的着色器变体，这需要使用#pragma target 3.5 指令。让我们给所有的着色器做同样的限制。

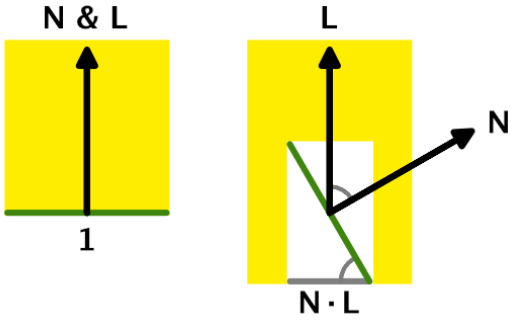
```
1.  HLSLPROGRAM
2.  #pragma target 3.5
3.  …
4.  ENDHLSL
```

3. BRDF（双向反射分布函数）

我们现在使用的是一个分厂简单的光照模型，只适用于完美漫反射表面。我们可以通过使用双向反射分布函数（简称 BRDF）来实现更多样化和逼真的光照。我们将使用和通用渲染管线相同的方法，它为了性能牺牲了一部分真实度。

3.1. 入射光

当一束光正面照射到片面表面，所有的能量都会对片元产生影响。简单起见，我们假设光束的宽度和片元宽度一致。这种情况下光线方向L和表面法线N对齐，所以$N \cdot L = 1$。当它们没有对齐时，至少有一部分光束没有照射到片元表面，所以片元受到了较少能量的影响。影响到片元的能量部分是$N \cdot L$。负值意味着表面朝向远离光源的方向，所以不会被它影响。

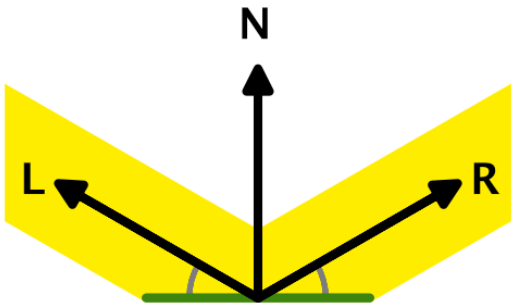


入射光部分

3.2. 出射光

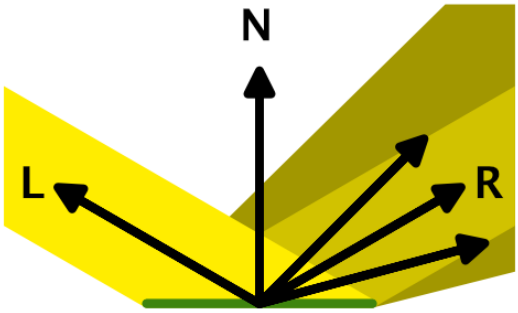
我们不是直接观察射向表面的光的。我们只能看到由表面反射出来，并且到达摄像机或我们眼睛的那部分。如果表面是完全平整的镜面，那么光线就会反射出来，同时出射角等于入射

角。如果摄像机对齐出射光，我们就能看到它。这被称作镜面反射。这是光线交互的简化，但对我们的目的来说足够了。



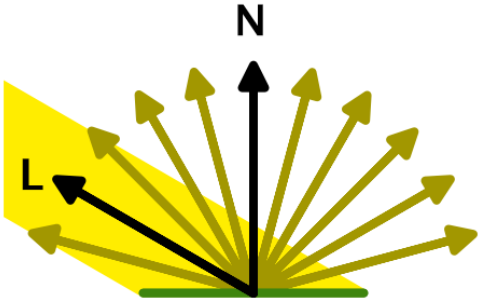
完美的镜面反射

但如果表面不是完全平整，那么光线就会发生散射，因为片元是由许多不同朝向的更小片元所组成。这把光束分解成了许多不同方向的更小光束，它们有效模糊了镜面反射。即使不与完美的反射方向对齐，我们最终还是能看到一些散射光。



分散的镜面反射

除此之外，光线也会穿透表面，反弹并以不同的角度射出，还有其他一些我们不需要考虑的东西。极端情况下，我们最终会看到一个完美的漫反射表面，它在几乎所有可能的方向上都散射了光线。这就是我们目前要在着色器中计算的光照。



完美的漫反射

不论摄像机在哪，从表面获取的漫反射光数量都是相同的。但这就意味着我们观测到的光能量远远小于到达片元表面的数量。这意味着我们需要用某些系数来缩放入射光。然而，因为这个系数始终相同，所以我们可以将它烘培到光源颜色和强度里。因此我们使用的最终光源颜色就代表了从正面照亮的完美白色漫反射表面片元反射时观测到的光量。这是光源实际发出的总量的一个微分。还有其他配置光源的方法，比如给定流明或勒克斯，这更易于配置显示光源，但我们关注于当前的方法。

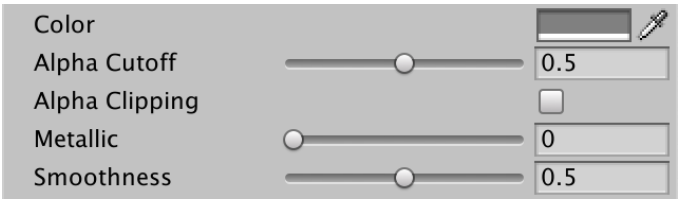
3.3. 表面属性

表面可以是完美漫反射，完美镜面，或者两者之间的任意状态。我们可以有多种方法来控制这些。我们将使用金属度工作流，这要求我们在 Lit 着色器里添加两个表面属性。

第一个属性表示表面是金属的还是非金属的，也称为导电性。因为表面可能包含混合状态，我们给它添加一个 0-1 范围的滑杆，其中 1 代表完全金属化。默认值是完全导电。

第二个属性控制表面的光滑程度。我们也给它使用一个 0-1 范围的滑杆，其中 0 表示完全粗糙而 1 表示完全光滑。我们使用 0.5 作为默认值。

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



带有金属度和光滑度滑杆的材质

把这些属性添加到 UnityPerMaterial 缓冲区里。

```
1.  UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
2.      UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
3.      UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
4.      UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
5.      UNITY_DEFINE_INSTANCED_PROP(float, _Metallic)
6.      UNITY_DEFINE_INSTANCED_PROP(float, _Smoothness)
7.  UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

同样添加到 Surface 结构里。

```
1.  struct Surface {
2.      float3 normal;
3.      float3 color;
4.      float alpha;
5.      float metallic;
6.      float smoothness;
7.  };
```

在 LitPassFragment 里把它们复制到 surface 里。

```
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);
```

并且在 PerObjectMaterialProperties 里也添加对它们的支持。

```
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 属性

我们使用表面属性来计算 BRDF 方程。它告诉我们最终看到多少从表面反射的光量，该结果包含了漫反射和镜面反射。我们需要将表面颜色拆分成漫反射部分和镜面反射部分，并且我们也需要知道表面的粗糙程度。让我们在 BRDF 结构体里记录这三个数值，将它放到单独的 BRDF.hlsl 文件里。

```
#ifndef CUSTOM_BRDF_INCLUDED
#define CUSTOM_BRDF_INCLUDED

struct BRDF {
	float3 diffuse;
	float3 specular;
```

```
	float roughness;
};

#endif
```

并且添加一个函数来对给定的表面获取 BRDF 数据。我们从完美漫反射表面开始，所以漫反射部分等于表面颜色，同时镜面反射部分为黑色，并且粗糙度为 1。

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

在 Light 和 Lighting 之间引入 BRDF。

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

跟两个 GetLighting 函数都添加 BRDF 参数，然后用入射光乘以漫反射部分，而不是整个表面颜色。

```
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;
}
```

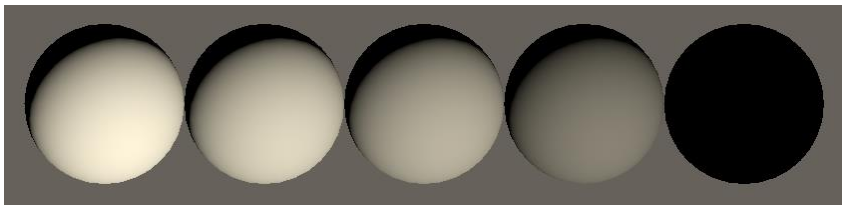
最后，在 LitPassFragment 里获取 BRDF 数据，并将其传给 GetLIghting。

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

3.5. 反射率

表面反射的方式有所不同，但通常金属会通过镜面反射反射所有光，并且没有漫反射。所以我们声明反射率等于表面金属度属性。反射出的光没有漫反射，所以我们应该在 GetBRDF 里将漫反射颜色用 1 减反射率来进行缩放。

```
1.  float oneMinusReflectivity = 1.0 - surface.metallic;
2.
3.  brdf.diffuse = surface.color * oneMinusReflectivity;
```



金属度分别为 0，0.25，0.5，0.75 和 1 的白色球体

实际上，一些光会从电介质表面反射出来，这使得它们有高光。非金属的反射率有所不同，但平均差不多是 0.04。我们把它定义为最小反射率，并且添加一个 OneMinusReflectivity 函数，把范围从 0-1 调整为 0-0.96。这个范围调整与 URP 的方法相同。

```
1.  #define MIN_REFLECTIVITY 0.04
2.
3.  float OneMinusReflectivity (float metallic) {
4.      float range = 1.0 - MIN_REFLECTIVITY;
5.      return range - metallic * range;
6.  }
```

在 GetBRDF 里使用这个方法可以保证最小值。当之渲染漫反射时，几乎不会注意到差别，但添加镜面反射时影响就会变大。如果没有这个最小值，非金属将不会有镜面高光。

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

3.6. 镜面颜色

通过一种方式反射的光不会再通过另一种方式反射。这被称作能量守恒，意味着出射光总量不会多于入射光总量。这意味着镜面颜色应该等于表面颜色减去漫反射颜色。

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

然而，这忽略了金属会影响镜面反射颜色的事实，然而非金属并不会。电介质表面的镜面反射颜色应该是白色，我们可以用金属度属性在最小反射率和表面颜色之间进行线性插值来获

得该颜色。

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

3.7. 粗糙度

粗糙度是光滑度的反义词，所以我们可以简单地使用 1 减光滑度。Core RP 库有一个名为 PerceptualSmoothnessToPerceptualRoughness 的函数执行了这个功能。我们将使用这个方法来确保定义的是感知光滑度和感知粗糙度。我们可以通过 PerceptualRoughnessToRoughness 函数来转换出实际粗糙度，该值为感知粗糙度的平方。这种算法符合迪斯尼光照模型。这么做的原因是在编辑材质时，调整感知版本更为直观。

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

这些方法定义在 Core RP 库的 CommonMaterial.hlsl 文件里。在 Common 文件中的核心 Common 之后引入这个文件。

```
#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. 观察方向

为了确定摄像机与完美反射方向的对齐程度，我们需要知道摄像机的位置。Unity 通过 float3 _WorldSpaceCameraPos 提供了这个数据，所以将它添加到 UnityInput 里。

```
float3 _WorldSpaceCameraPos;
```

为了在 LitPassFragment 里获得观察方向（从物体表面到摄像机的方向），我们需要给 Varyings 添加世界空间下表面的位置。

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

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

```
    output.positionCS = TransformWorldToHClip(output.positionWS);
    …
}
```

考虑到观察方向是物体表面数据的一部分，所以将它添加到 Surface。

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

在 LitPassFragment 里给它赋值。它等于摄像机位置减去片元位置之后的归一化向量。

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

3.9. 镜面反射强度

我们观察到的镜面反射强度取决于我们的观察方向与完美反射方向的匹配程度。我们将使用和 URP 里相同的方程，它是 Minimalist CookTorrance BRDF 的变体。该方程包含了一些平方计算，所以我们先在 Common 里添加一个方便调用的 Square 函数。

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

然后在 BRDF 里添加一个 SpecularStrength 函数，并使用 Surface，BRDF 和 Light 作为参数。

它用来计算 $\frac{r^2}{d^2 \max(0.1,(L\cdot H)^2)n}$，其中 r 是粗糙度，并且所有的点积都需要执行 saturate。

另外，$d=(N\cdot H)^2(r^2-1)+1.0001$，$N$ 为表面法线，L 是光源方向，并且 $H=L+V$ 的归一化向量，意味着光源方向和观察方向的半角向量。使用 SafeNormalize 函数来归一化向量，以避免当这些向量相反时除以零。最后，$n=4r+2$，并且是个归一化项。

```
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);
```

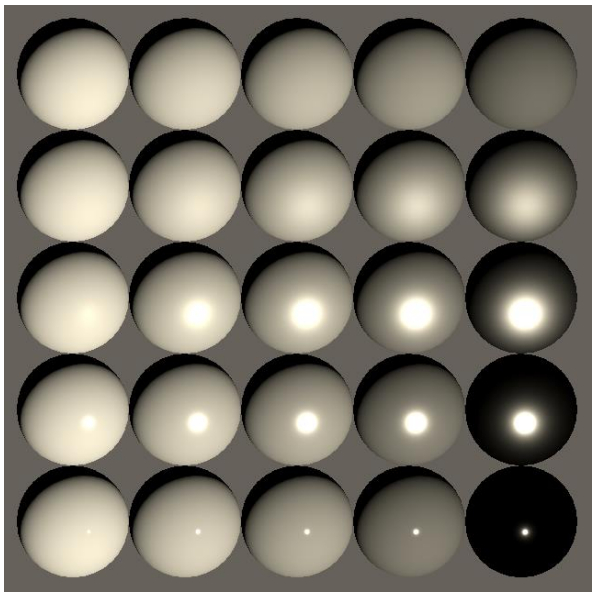
```
	float d2 = Square(nh2 * (r2 - 1.0) + 1.00001);
	float normalization = brdf.roughness * 4.0 + 2.0;
	return r2 / (d2 * max(0.1, lh2) * normalization);
}
```

接下来，添加 DirectBRDF 函数，返回通过直接照明获得的颜色（给定表面，BRDF 和灯光）。该结果是由镜面反射强度修正的镜面反射颜色加上漫反射颜色。

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

接下来 GetLighting 必须将入射光乘以刚才函数的结果。

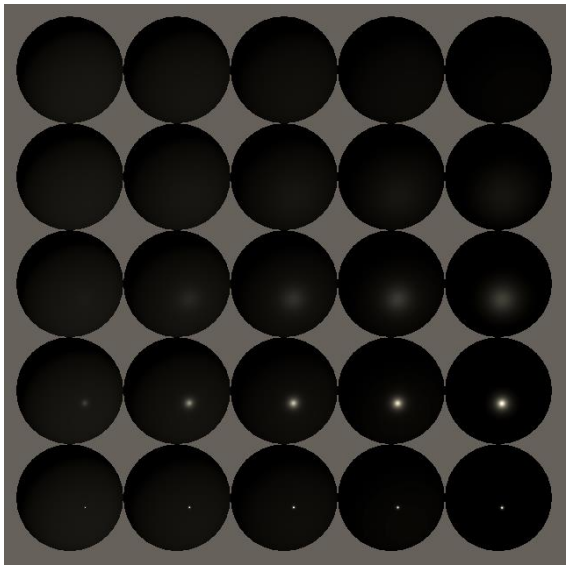
```
float3 GetLighting (Surface surface, BRDF brdf, Light light) {
	return IncomingLight(surface, light) * DirectBRDF(surface, brdf, light);
}
```



光滑度从上到下是分别是 0，0.25，0.5，0.75 和 0.95

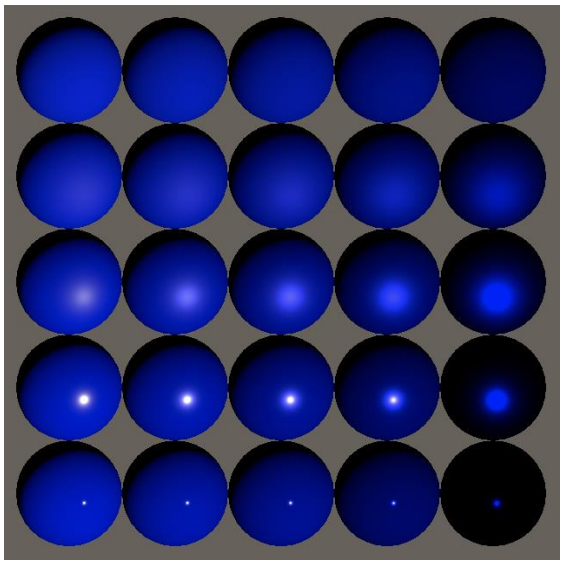
现在我们有了镜面反射，给表面添加了高光。对于完全粗糙的表面，高光表现类似于漫反射。越光滑的表面会有越集中的高光。完全光滑的表面有一个无穷小的高光，所以无法看到。需要一部分散射来让它可见。

由于能量守恒，光滑表面的高光会非常亮，因为大多数照在表面片元的光都被聚焦了。因此在高光可见的地方，我们可能最终会看到远高于原本漫反射的亮度。你可以通过大幅缩减最终渲染颜色来辨别它。



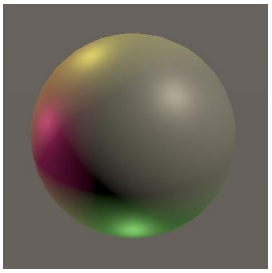
最终颜色除以 100

你也可以通过使用白色以外的基础颜色来分辨金属影响镜面反射颜色，而非金属不会。



蓝色基础色

现在我们可以使用功能强大的直接照明了，尽管目前结果显得太暗，特别是对于金属，这是因为我们还没有支持环境反射。一个均匀的黑色环境目前会比默认的天空盒更真实，但这样会是的我们的对象更难分辨。添加更多的光也比较有效。



四盏光

3.10. 网格球

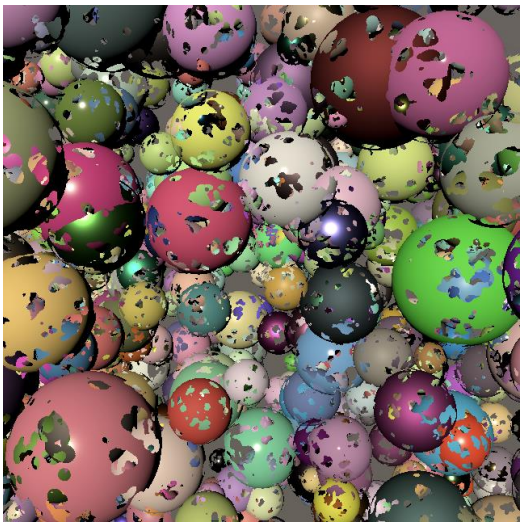
让我们同样对 MeshBall 支持可变的金属度和光滑度。这需要添加两个浮点数组。

```
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);
}
```

让我们在 Awake 里设置 25%的实例金属化，同时光滑度范围从 0.05-0.95。

```
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);
```

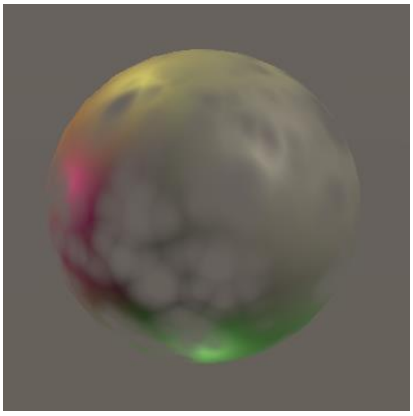
然后用受光材质创建网格球。



受光网格球

4. 透明度

让我们再来考虑透明度。对象依然会随着 alpha 值变淡，但现在是反射光变淡。这对漫反射有意义，因为只有部分光线被反射，剩下的穿透了表面。

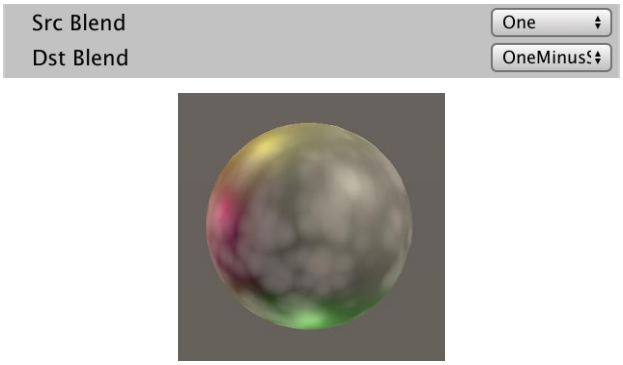


变淡的球体

然而，镜面反射也变淡了。对于完全干净的玻璃来说，光线要么穿透，要么被反射。镜面反射部分不会衰减。我们不能用当前的方法来表现镜面反射。

4.1. 预乘 Alpha

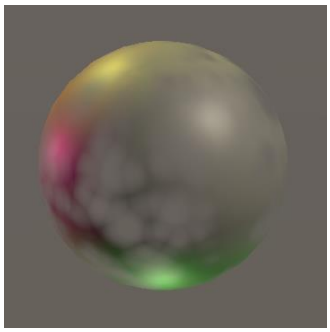
解决办法是在保持镜面反射强度的情况下衰减漫反射光。因为源目标混合模式应用到所有我们不能使用的物体上，所以让我们将其设置为 One 同时保持目标混合模式为 OneMinusSrcAlpha。



源目标混合模式设为 1

这样就恢复了镜面反射，但漫反射不再发生衰减。我们通过将漫反射颜色乘以系数表面 alpha 来修正这个表现。因此我们将漫反射预乘 alpha，而不是基于之后的 GPU 混合。这个方法被称作预乘 alpha 混合。在 GetBRDF 里实现它。

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



预乘后的漫反射

4.2. 预乘开关

将漫反射预乘 alpha 可以有效地将对象变成玻璃，而常规 alpha 混合可以有效地使对象部分存在。让我们通过给 GetBRDF 添加一个布尔值参数（默认为 false）来控制是否需要预乘，从而同时支持两种表现。

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

	…
}
```

我们可以使用_PREMULTIPLY_APLHA 关键字来决定在 LitPassFragment 里使用哪种方法，类似于控制 alpha 裁剪的方式。

```
#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);
```

给 Lit 的 Pass 添加这个关键字的着色器功能。

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

并且同样给着色器添加一个开关属性。

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

预乘 Alpha 开关

5. 着色器 GUI

我们现在支持了多种渲染模式，每一种都需要特定的设置。为了更易于在多种模式间切换，让我们给材质面板添加一些按钮来应用预设配置。

5.1. 自定义着色器 GUI

为 Lit 着色器的主块添加一个 CustomEditor “CustomShaderGUI 的声明。

```
Shader "Custom RP/Lit" {
	…

	CustomEditor "CustomShaderGUI"
}
```

这告诉 Unity 编辑器使用一个 CustomShaderGUI 类的实例来绘制使用了 Lit 着色器的材质面

板。为这个类创建一个脚本资产，然后放到新建的 Custom RP/Editor 文件夹里。

我们需要用到 UnityEditor，UnityEngine 和 UnityEngine.Rendering 三个命名空间。这个类必须继承 ShaderGUI，并且重写公共 OnGUI 方法，这个方法带有一个 MaterialEditor 和一个 MaterialProperty 数组参数。让它调用基类方法，鱼是我们最终得到了默认的 inspector 面板。

```
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. 设置属性和关键字

要完成功能，我们需要访问三项内容，并将其存在字段中。第一是材质编辑器，它是用于显示和编辑材质的基础编辑器对象。第二是被编辑材质的引用，我们可以通过编辑器的 targets 属性获取。它被定义为一个 Object 数组，因为 targets 是通用 Editor 类的属性。第三是可编辑的属性数组。

```
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;
}
```

为了设置一条属性，必须先从数组中找到它，我们可以使用 ShaderGUI.FindProperty 方法，传入名称和属性数组。然后我们可以通过给 floatValue 属性赋值来调整它的属性。将它封装在一个带有 name 和 value 参数的 SetProperty 方法里。

```
void SetProperty (string name, float value) {
```

```
    FindProperty(name, properties).floatValue = value;
}
```

设置关键字涉及的内容更多一些。我们给它创建一个 SetKeyword 方法，带有一个 name 和布尔值参数用来表明该关键字启用或禁用。我们必须调用所有材质的 EnableKeyword 或者 DisableKeyword，传入关键字 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);
        }
    }
}
```

我们同样创建一个 SetProperty 变体统一控制属性和关键字的开关。

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

现在我们可以简单定义 Clipping, PremultiplyAlpha, SrcBlend, DstBlend 和 ZWrite 的 setter 属性。

```
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);
}
```

```
16.
17. bool ZWrite {
18.     set => SetProperty("_ZWrite", value ? 1f : 0f);
19. }
```

最后，将渲染队列分配给所有材质的 RenderQueue 属性。我们可以在这里使用 RenderQueue 枚举。

```
1.  RenderQueue RenderQueue {
2.      set {
3.          foreach (Material m in materials) {
4.              m.renderQueue = (int)value;
5.          }
6.      }
7.  }
```

5.3. 预设按钮

可以通过 GUILayout.Button 方法传入一个预设名文本来创建一个按钮。如果该方法返回 true，那么按下它。在应用预设之前，我们应当对编辑器注册一个撤销步骤，这可以通过调用 RegisterPropertyChangeUndo 传入名称来实现。因为代码对所有预设都相同，所以把它放在 PresetButton 方法里，并返回是否需要应用预设。

```
1.  bool PresetButton (string name) {
2.      if (GUILayout.Button(name)) {
3.          editor.RegisterPropertyChangeUndo(name);
4.          return true;
5.      }
6.      return false;
7.  }
```

我们为每种预设创建一个单独的方法，从默认的 Opaque 开始。激活后设置适当的属性。

```
1.  void OpaquePreset () {
2.      if (PresetButton("Opaque")) {
3.          Clipping = false;
4.          PremultiplyAlpha = false;
5.          SrcBlend = BlendMode.One;
6.          DstBlend = BlendMode.Zero;
7.          ZWrite = true;
8.          RenderQueue = RenderQueue.Geometry;
9.      }
10. }
```

第二个预设是 Clip，复制 Opaque 之后将 clipping 启用，并且队列设置为 AlphaTest。

```
1.  void ClipPreset () {
2.      if (PresetButton("Clip")) {
3.          Clipping = true;
4.          PremultiplyAlpha = false;
5.          SrcBlend = BlendMode.One;
6.          DstBlend = BlendMode.Zero;
7.          ZWrite = true;
8.          RenderQueue = RenderQueue.AlphaTest;
9.      }
10. }
```

第三个预设是面向标准透明度，即淡出对象，所以我们命名为 Fade。再复制一份 Opaque，调整混合模式和队列，再关闭深度写入。

```
1.  void FadePreset () {
2.      if (PresetButton("Fade")) {
3.          Clipping = false;
4.          PremultiplyAlpha = false;
5.          SrcBlend = BlendMode.SrcAlpha;
6.          DstBlend = BlendMode.OneMinusSrcAlpha;
7.          ZWrite = false;
8.          RenderQueue = RenderQueue.Transparent;
9.      }
10. }
```

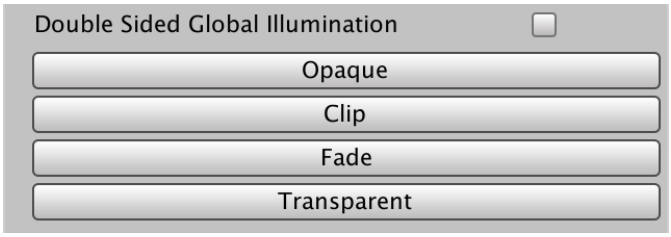
第四个预设是 Fade 的变体，包含了预乘 alpha 混合。我们命名为 Transparent，因为它用于应用正确照明的半透明表面。

```
1.  void TransparentPreset () {
2.      if (PresetButton("Transparent")) {
3.          Clipping = false;
4.          PremultiplyAlpha = true;
5.          SrcBlend = BlendMode.One;
6.          DstBlend = BlendMode.OneMinusSrcAlpha;
7.          ZWrite = false;
8.          RenderQueue = RenderQueue.Transparent;
9.      }
10. }
```

在 OnGUI 最后调用这些预设方法，鱼是他们在默认 inspector 面板下面显示出来。

```
1.  public override void OnGUI (
2.      MaterialEditor materialEditor, MaterialProperty[] properties
```

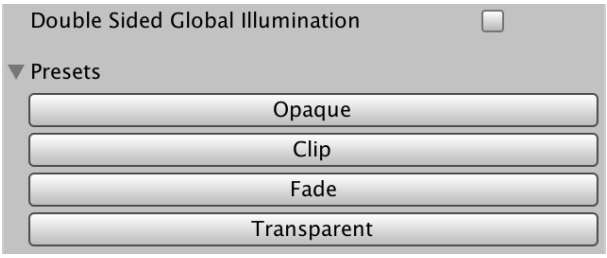
```
3.  ) {
4.      …
5.
6.      OpaquePreset();
7.      ClipPreset();
8.      FadePreset();
9.      TransparentPreset();
10. }
```



预设按钮

这些预设按钮不会经常使用，所以我们放在一个默认的折叠选项里。调用 EditorGUILayout.Foldout，并传入当前折叠状态，文本和 true 用来表示点击切换状态。它返新的折叠状态，我们将其存在字段里。只有当折叠打开时才绘制按钮。

```
1.  bool showPresets;
2.
3.  …
4.
5.  public override void OnGUI (
6.      MaterialEditor materialEditor, MaterialProperty[] properties
7.  ) {
8.      …
9.
10.     EditorGUILayout.Space();
11.     showPresets = EditorGUILayout.Foldout(showPresets, "Presets", true);
12.     if (showPresets) {
13.         OpaquePreset();
14.         ClipPreset();
15.         FadePreset();
16.         TransparentPreset();
17.     }
18. }
```



预设折叠栏

5.4. 不受光预设

我们同样可以给 Unlit 着色器使用自定义着色器 GUI。

```
Shader "Custom RP/Unlit" {
	…

	CustomEditor "CustomShaderGUI"
}
```

然而，开启预设会导致报错，因为我们尝试设置的属性在着色器中不存在。我们可以通过调整 SetProperty 来规避这个问题。嗲用 FindPropery 并把 false 作为可选参数，意味着如果属性没找到不会报错。返回结果将变成 null，所以只有不是这种情况时才会赋值。同样返回属性是否存在。

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

然后调整关键字版本的 SetProperty，以便只在对应属性存在时才设置关键字。

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

5.5. 没有透明度

现在这些预设对使用 Unlit 着色器的材质也有效了，尽管 Transparent 模式在这种情况下没什么意义，因为相关的属性不存在。让我们把不相关的预设隐藏掉。

首先，添加一个 HasProperty 方法，返回属性是否存在。

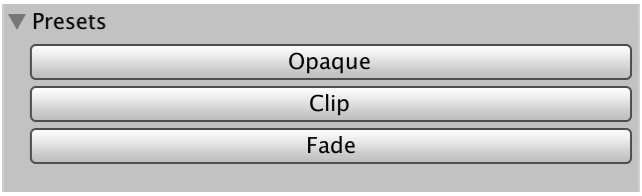
```
1.  bool HasProperty (string name) =>
2.      FindProperty(name, properties, false) != null;
```

第二，创建一个方便的属性来检查_PremultiplyAlpha 是否存在。

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

最后，在 TransparentPreset 里先检查该属性，然后让所有 Transparent 预设的内容都取决于该属性。

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



缺少 Transparent 预设的 Unlit 材质

到目前为止，我们实现了实用的光照，但是仅适用于方向光的直接照明。现在没有环境光，没有阴影，也不支持其他类型的光。我们将在将来添加这些。