

游戏设计与开发 简单shader 参考文档

SJTU Dalab 2020.03

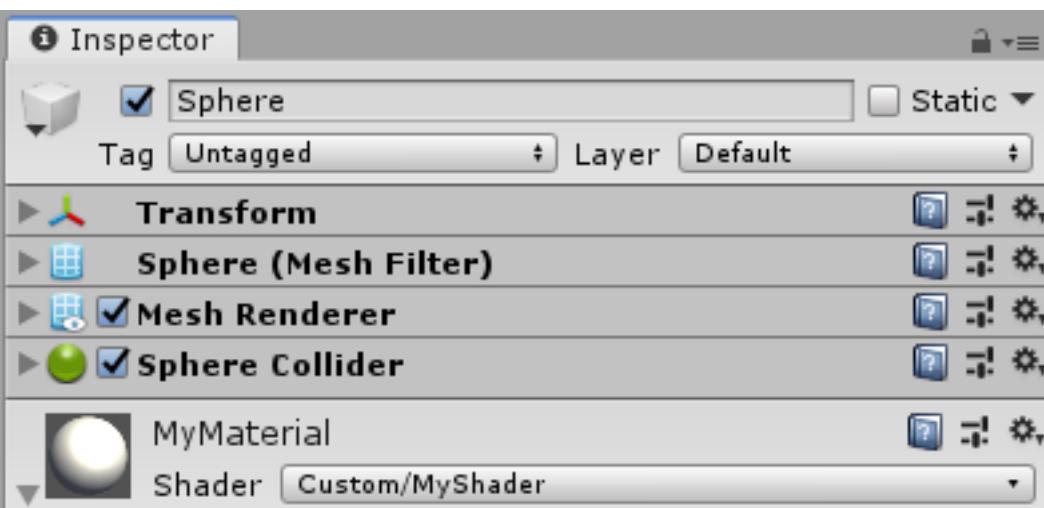
Shader（着色器）是一段能够针对3D对象进行操作、并被GPU所执行的程序。Shader负责将输入的Mesh（网格）以指定的方式和输入的贴图或者颜色等组合作用，然后输出。绘图单元可以依据这个输出来将图像绘制到屏幕上。

Material（材质）则用于描述了渲染一个对象时使用的Shader、纹理、颜色等参数设置。Unity中通过将Material绑定到物体上，便可以得到相应的渲染效果。

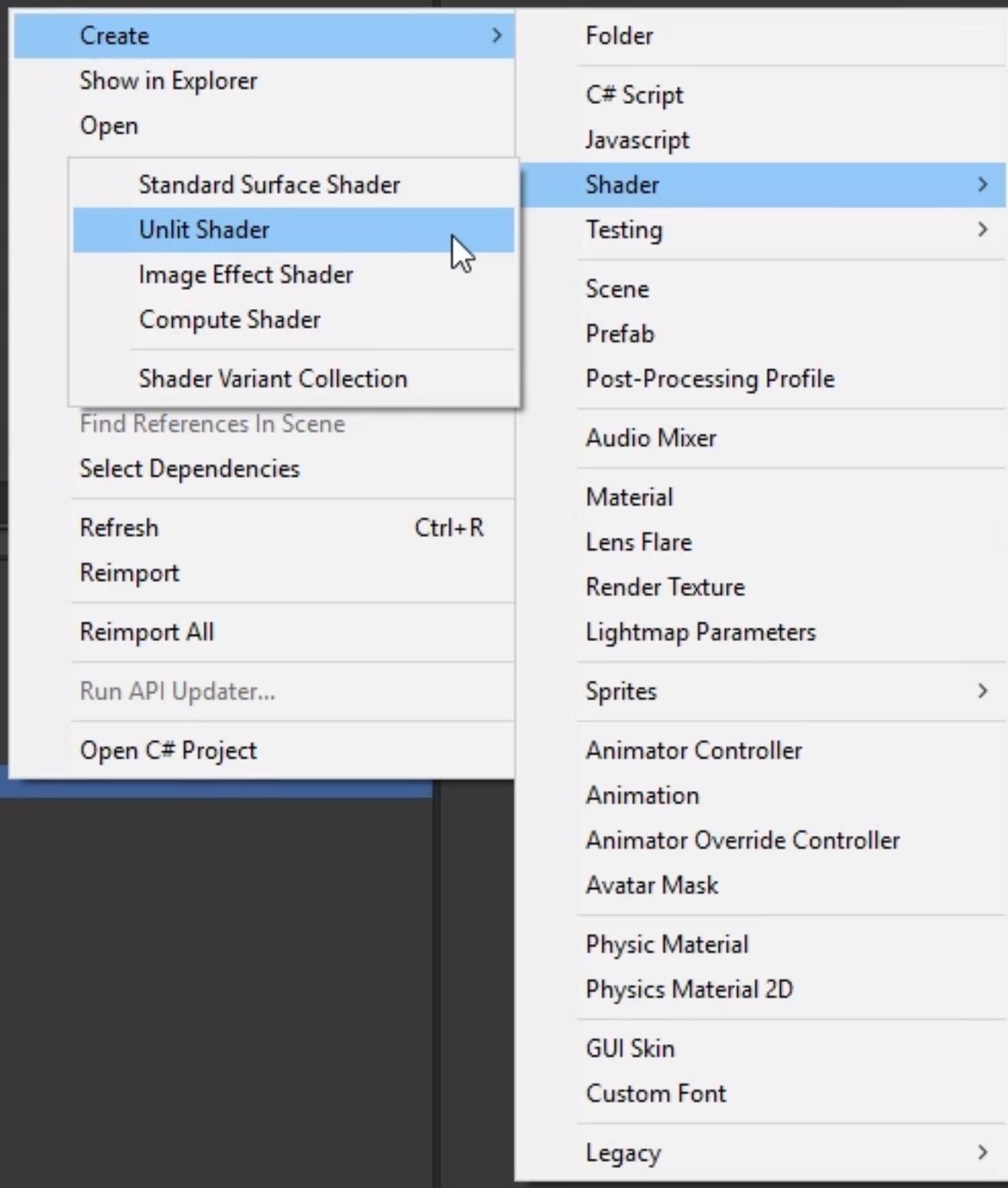
本文档将会介绍Shader的基础知识、编写方式、自定义的Shader GUI。

1. Shader的创建及使用

创建一个Material，命名为"MyMaterial"，并将其绑定到场景中的一个球体对象上。



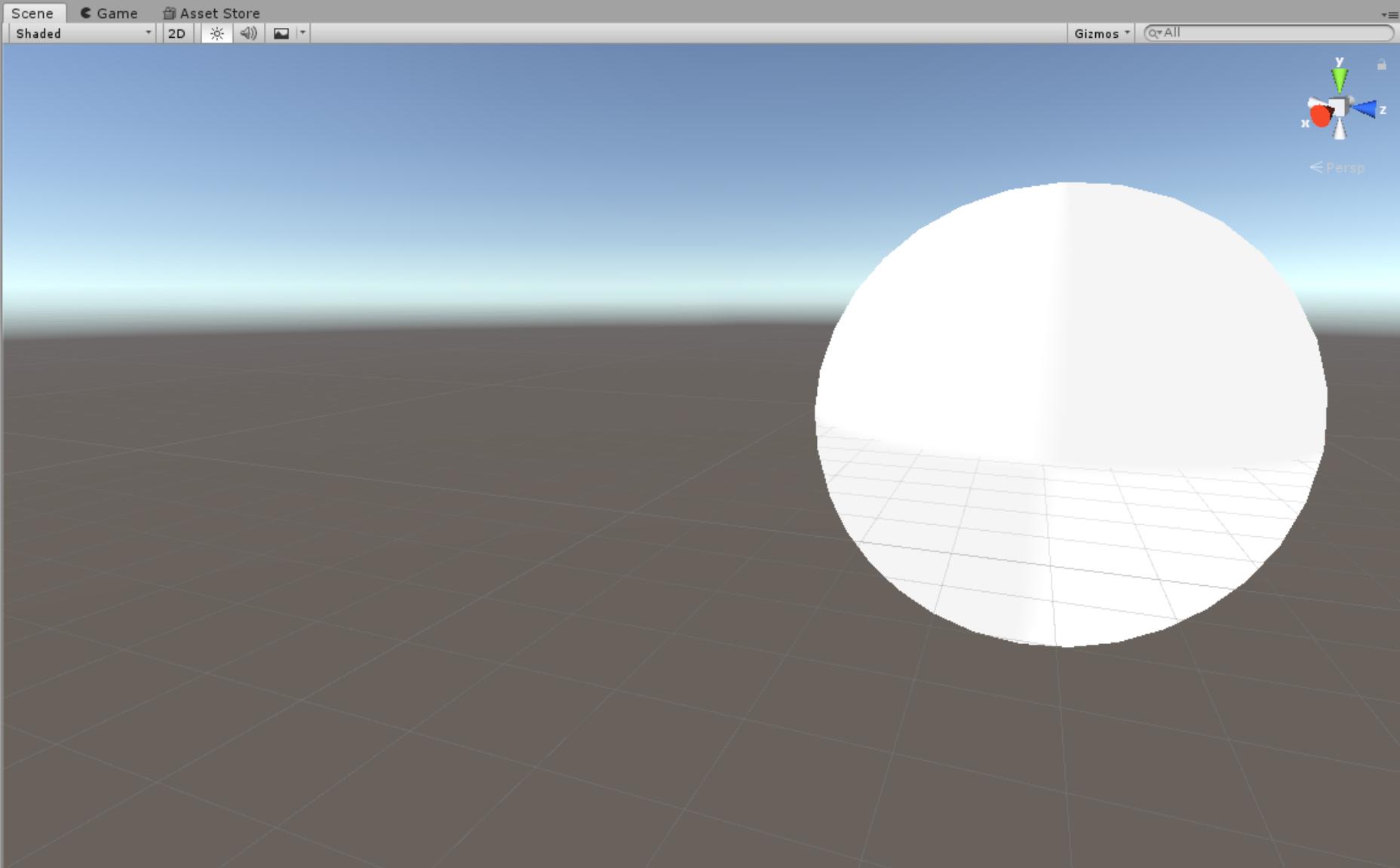
如图新建一个Unlit Shader，将其命名为"MyShader.shader"。



在MyMaterial中选择MyShader。



此时我们便得到了一个用"MyShader.shader"渲染的球体了。它现在还不是很好看，我们接下来学习如何改进它。



2. Shader的结构介绍

打开之前创建的 "MyShader" ，你会看到一段代码。

开头的这部分定义了 Shader 的名字，并声明 Properties (属性) ，定义了一个 SubShader (子着色器) 。

Properties 中声明的属性会在 GUI 界面显示并供调整设置，而在Shader中使用该属性时，需要先在 `CGPROGRAM ... ENDCG` 语句块中再次声明相同名字的变量。

在 Shader 中使用多个 SubShader ，可以让 Unity 根据 SubShader 的 Tag ，在不同情况下使用不同的 SubShader 。 SubShader 中的 Pass (通道) 则包含了一次渲染中的参数和程序。对于一些特殊效果，往往需要多个 Pass 对对象进行多次渲染。

Pass内则通过 `CGPROGRAM` 和 `ENDCG` 标记程序代码的开始和终止。本次作业中，我们将使用 Vertex Shader (顶点着色器) 和 Fragment Shader (片段着色器) 来实现，因此我们需要在 Pass 中定义相应的两个函数，并通过 pragma 指令指定编译器使用这两个函数。

```
Shader "Custom/MyShader" {
    Properties {...}

    SubShader {
        Pass {
            CGPROGRAM
                #pragma vertex MyVertexProgram
                #pragma fragment MyFragmentProgram
                ...
                void MyVertexProgram () {}
                void MyFragmentProgram () {}

                ...
            ENDCG
        }
    }
}
```

在 Shader 程序中，我们常常会使用到一些固定的功能或变量，如不同坐标系的转换等。Unity 中已经实现了一部分功能，我们可以通过 `#include` 指令将他们加载到程序中，如 `#include "UnityCG.cginc"`。

3. 编写简单的 Shader

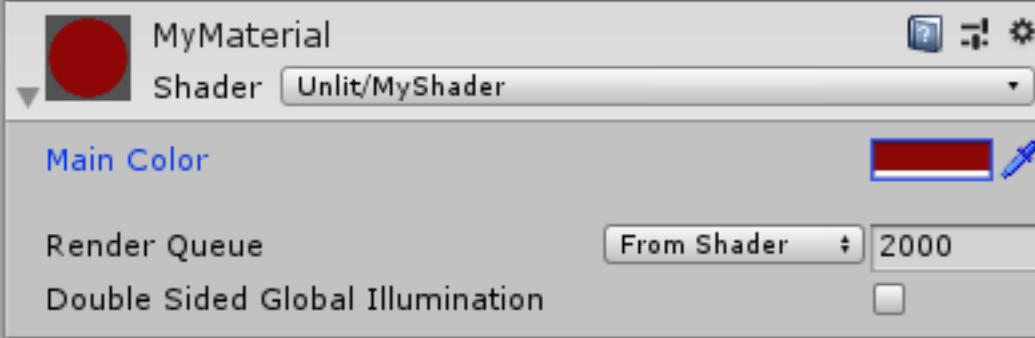
3.1. 纯色 Shader

首先我们来实现一个简单的纯色 Shader，它将通过直接使用 GUI 传入的颜色，渲染出一个纯色的物体。

由于需要使用 GUI 传入的颜色，我们需要在 Properties 中声明一个颜色类的变量，如下：

```
Properties {
    _MainColor ("Main Color", Color) = (1,1,1,1)
}
```

Properties 中声明的语法是：`变量名("显示名", 类型) = 默认值`，如上我们便声明了一个名为 `_MainColor` 的 Color 类型变量，它在 GUI 面板中显示的名字是 `Main Color`。



为了在程序中使用这个颜色，我们需要先在 `CGPROGRAM ... ENDCG` 语句块中再次声明相同名字的变量。由于 `Color` 类型在 Shader 中表现为 `float4`，这里的声明使用 `float4`。

```
CGPROGRAM
```

```
float4 _MainColor;
```

```
...
```

```
ENDCG
```

为了让程序能够明确的知道哪些位置需要被渲染成我们制定的颜色，我们首先要在 Vertex Shader 中对顶点位置进行操作。

为了后续方便，我们使用自定义的类型作为 Vertex Shader 的输入。在 Vertex Shader 之前，先定义一个 `VertexData` 的类型，这一步和C++中类似。由于这一节我们要实现的效果较为简单，只需要用到顶点的位置信息，我们只需要在 `VertexData` 中定义一个 `float4` 类型的变量 `position`，并指定它的语义为 `POSITION`。这样一来，程序便会让 `VertexData.position` 被解释为顶点的位置信息。

```
struct VertexData {  
    float4 position : POSITION;  
};
```

同样的我们使用自定义类型作为 Vertex Shader 的输出，也是 Fragment Shader 的输入。

```
struct FragmentData {  
    float4 position : SV_POSITION;  
};
```

在这个 Shader 中，我们只在 Vertex Shader 和 Fragment Shader 之间传递顶点位置值。同样地，为这个变量指定语义为 `SV_POSITION`，表示这是计算 Fragment Shader 时使用的顶点位置信息。

完成了以上的所有内容，我们现在可以开始正式编写 Shader 了。

在 Vertex Shader 中，程序应该通过输入的局部坐标系下的顶点位置，即这里定义的 `VertexData.position`，利用 MVP 矩阵（模型观察投影矩阵）计算得到屏幕空间中的位置。在新的 Unity 版本

中，这个矩阵运算可以直接用 `UnityObjectToClipPos` 来代替。

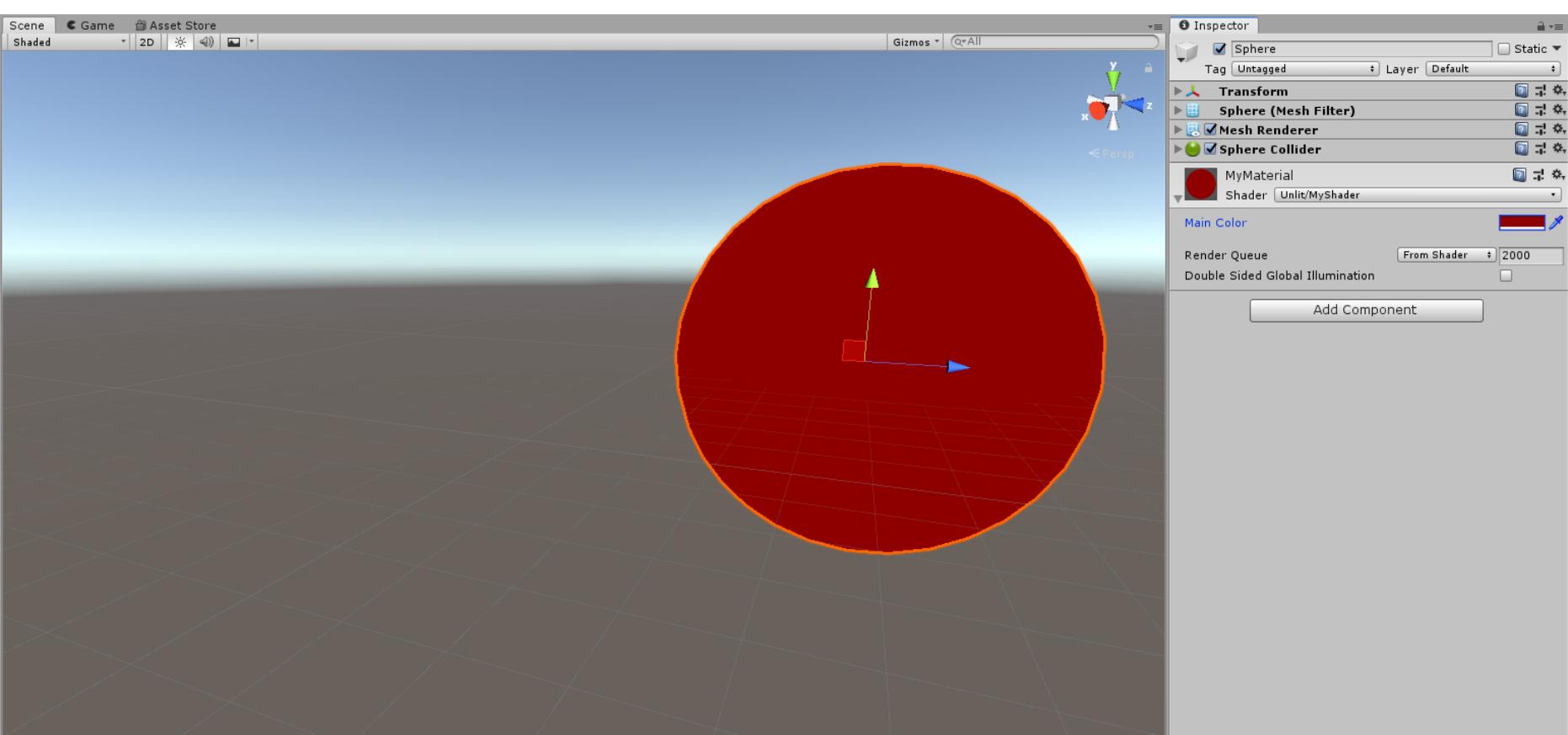
```
FragmentData MyVertexProgram (VertexData v) {  
    FragmentData i;  
    // / old version: i.position = mul(UNITY_MATRIX_MVP, v.position);  
    i.position = UnityObjectToClipPos(v.position);  
    return i;  
}
```

在 Fragment Shader 中，程序应该直接以颜色 `_MainColor` 为结果返回该颜色。

```
float4 MyFragmentProgram (FragmentData i) : SV_TARGET {  
    return _MainColor;  
}
```

注意这里的返回类型为 `float4`，我们需要指定它的语义为 `SV_TARGET`。

这样，我们便完成了一个简单的纯色 Shader。我们可以将其绑定到物体上，并通过 GUI 改变颜色设置得到不同颜色的物体。



3.2. 法线 Shader

在 Shader 的开发过程中，通常要进行 Debug 是非常困难的。一旦出现了错误，我们会直接得到一个完全错误的视觉效果。虽然可以通过 Unity 直接看到 Shader 的语法错误，但是语义上的错误却很难定位。我们无法使用传统步进的方式检查代码，同时也不便输出调试信息。

对此，一个常用的 Debug 方法是将 Shader 中的中间变量直接作为 Fragment Shader 的颜色输出，从视觉上来检查判断。接下来，我们以法线为例，尝试来实现一个展示法线方向的 Shader。

为了增加法线这个属性，我们需要在 `VertexData` 和 `FragmentData` 中都增加上 `normal` 属性。不同的是，在 `VertexData` 中，我们指定它的语义为传入的物体法向 `NORMAL`，而在 `FragmentData` 中，将其指定使用 `TEXCOORD0` 来储存这个信息。

```
struct VertexData {  
    float4 position : POSITION;  
    float3 normal : NORMAL;  
};  
  
struct FragmentData {  
    float4 position : SV_POSITION;  
    float3 normal : TEXCOORD0;  
};
```

传入的 `VertexData.normal` 只是物体空间中的法线，这会导致渲染出来的法线方向不统一，并且受限于拉伸和旋转。因此我们需要首先在 Vertex Shader 中将法线方向转换到世界坐标系下。我们是使用矩阵 `transpose((float3x3)unity_WorldToObject)` 来实现，或者可以在 `#include 'UnityCG.cginc'` 的前提下使用其中的函数 `UnityObjectToWorldNormal`。

```
FragmentData MyVertexProgram (VertexData v) {  
    FragmentData i;  
    i.position = UnityObjectToClipPos(v.position);  
    i.normal = mul(transpose((float3x3)unity_WorldToObject), v.normal);  
    i.normal = normalize(i.normal);  
    return i;  
}
```

或

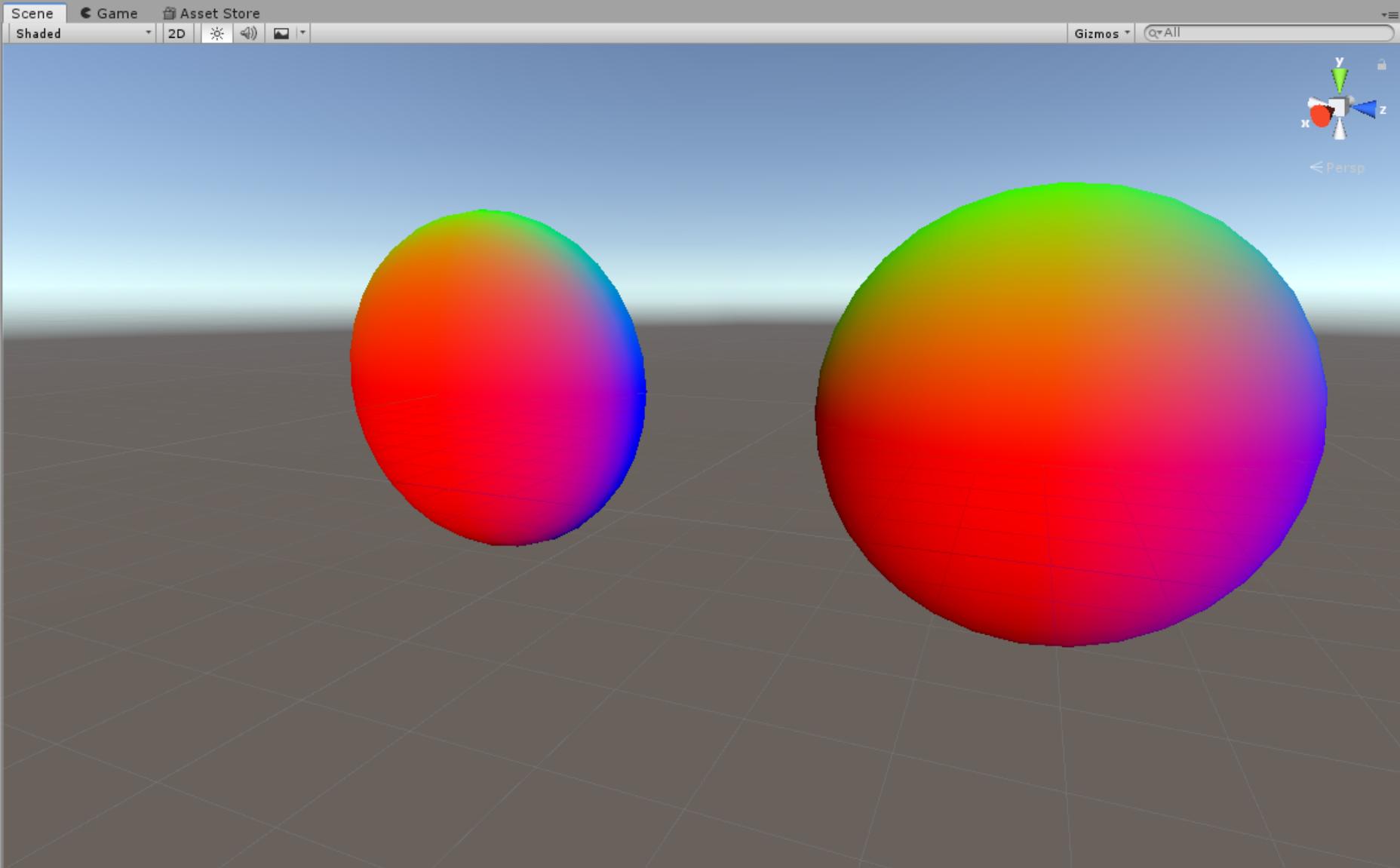
```
#include "UnityCG.cginc"
```

```
FragmentData MyVertexProgram (VertexData v) {  
    FragmentData i;  
    i.position = UnityObjectToClipPos(v.position);  
    i.normal = UnityObjectToWorldNormal(v.normal);  
    return i;  
}
```

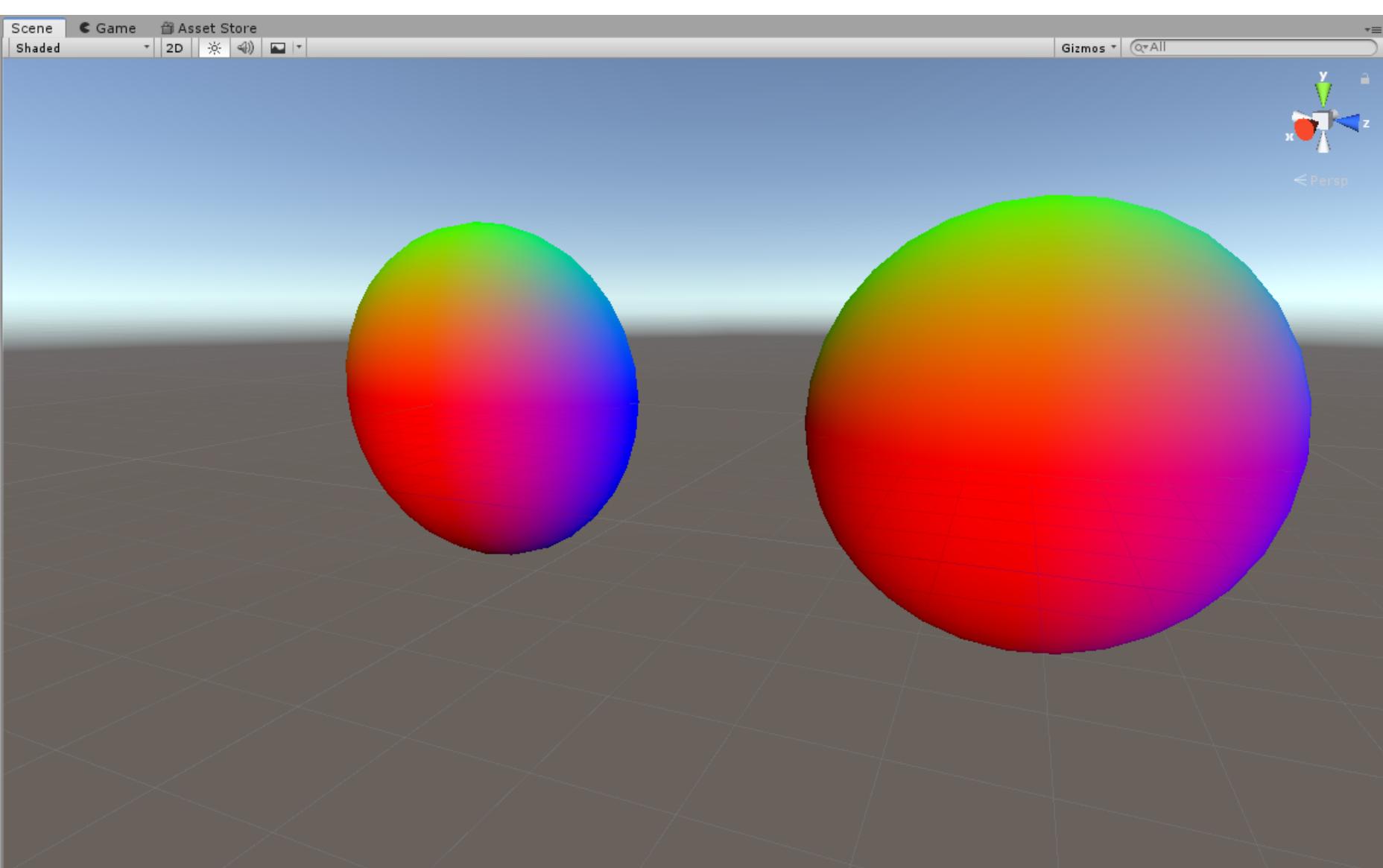
而在 Fragment Shader 中则可以直接返回法线方向

```
float4 MyFragmentProgram (FragmentData i) : SV_TARGET {  
    return float4(i.normal, 1);  
}
```

结果如下，注意左侧扁球体法线方向仍应是正确的。



而如果在 Vertex Shader 中没有对法线方向进行变换则会得到如下结果，注意左侧的扁球体法线仍是局部法线，和右侧球体不一致。



3.3. 纹理 Shader

纹理可以将图像投影到模型的网格三角面片，从而为模型增添更多的外观细节。纹理坐标是一个标准单位大小的二维坐标，也被称为UV坐标。通过纹理坐标，我们可以控制纹理到模型三角面片的映射关系。

在 Unity 中，使用纹理，我们首先需要在 `Properties` 中增加一个纹理属性

```
Properties{
    _MainTex ("Main Texture", 2D) = "white" {}
}
```

然后在 `CGPROGRAM ... ENDCG` 语句块中再次声明相同名字的变量

```
sampler2D _MainTex;
```

在 `VertexData` 和 `FragmentData` 中都添加上 `uv` 属性，其中 `VertexData.uv` 指定语义为 `TEXCOORD0`。

```
struct VertexData {
    float4 position : POSITION;
    float2 uv : TEXCOORD0;
};

struct FragmentData {
    float4 position : SV_POSITION;
    float2 uv : TEXCOORD0;
};
```

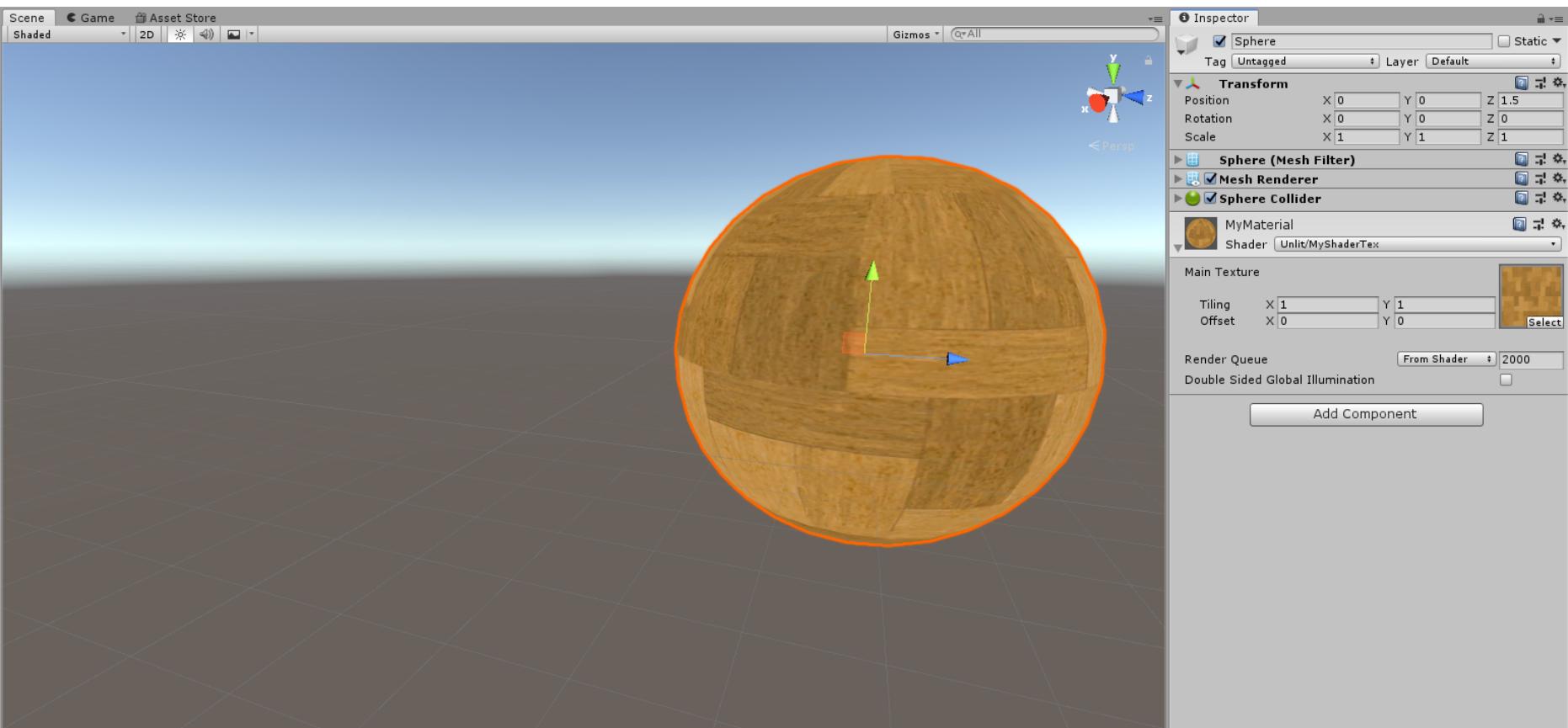
在 `Vertex Shader` 中，我们直接将 `VertexData.uv` 赋值给 `FragmentData.uv`，而在 `Fragment Shader` 中则返回 `_MainTex` 中位于 `FragmentData.uv` 位置的颜色值。

```
FragmentData MyVertexProgram (VertexData v) {
    FragmentData i;
    i.position = UnityObjectToClipPos(v.position);
    i.uv = v.uv;
    return i;
}

float4 MyFragmentProgram (FragmentData i) : SV_TARGET {
    return tex2D(_MainTex, i.uv);
}
```

这样，我们便完成了一个最基本的纹理贴图的 `Shader`，我们可以试着将任意一张图片贴到我们的

球体上



你也许注意到了在 GUI 中的纹理部分，材质检查器不仅添加了纹理，还添加了 Tiling 和 Offset 两个参数。这两个参数是用于调整纹理的缩放和偏移的，现在我们的 Shader 还不能够应用这两个参数。为了应用这两个参数，我们需要进一步修改 Shader 程序。

在 Shader 中，我们使用和“材质名+_ST后缀”，添加材质的缩放、偏移属性的声明。在本 Shader 中，即添加 `_MainTex_ST` 属性。

```
sampler2D _MainTex;  
float4 _MainTex_ST;
```

在 Vertex Shader 中，需要对 uv 坐标进行缩放偏移。这里，`float4` 的前两项 `xy` 表示 uv 坐标的缩放，后两项 `zw` 表示偏移。因此我们进行坐标变换。

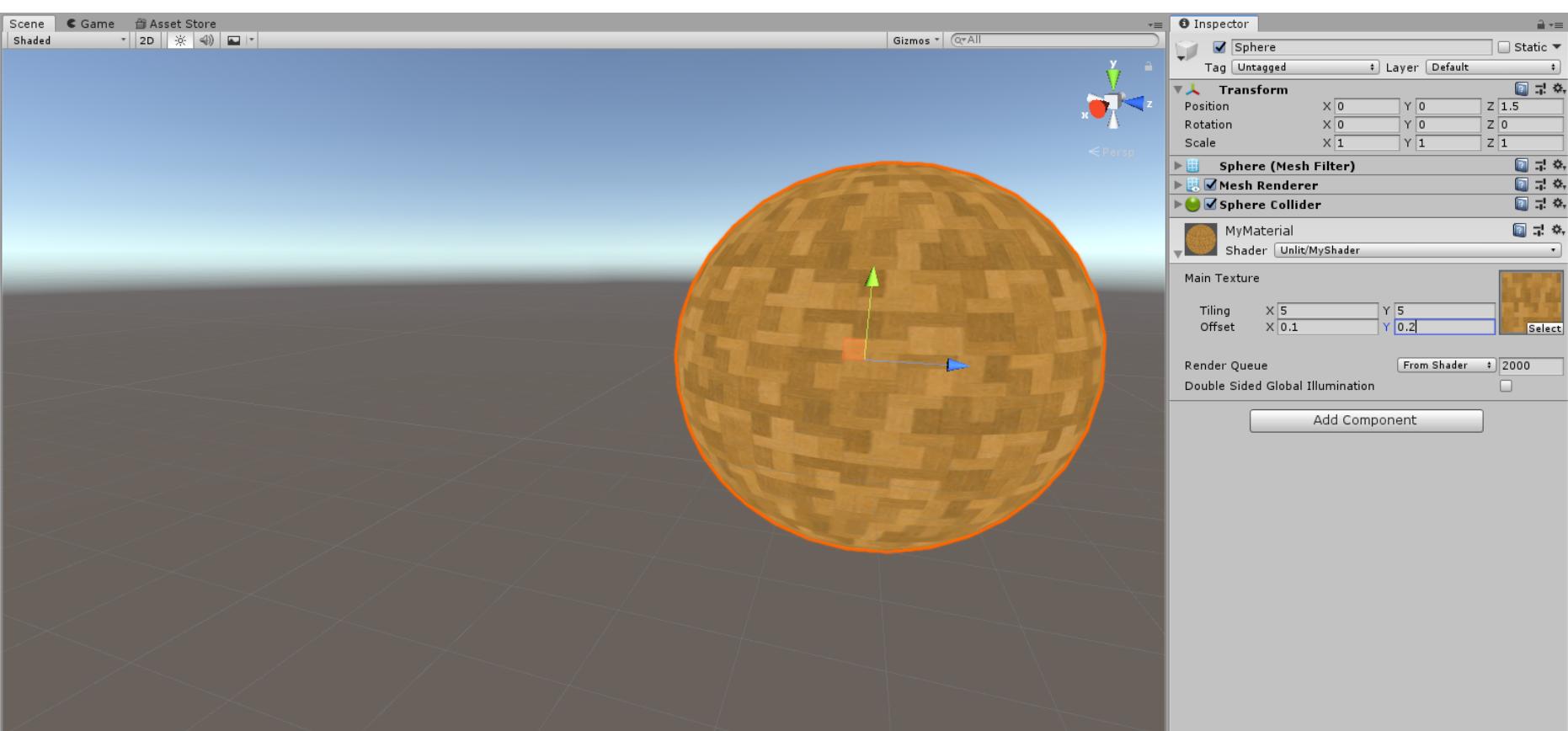
```
FragmentData MyVertexProgram (VertexData v) {  
    FragmentData i;  
    i.position = UnityObjectToClipPos(v.position);  
    i.uv = v.uv * _MainTex_ST.xy + _MainTex_ST.zw;  
    return i;  
}
```

在 `UnityCG.cginc` 中同样有简化后的函数可以使用。

```
#include "UnityCG.cginc"

FragmentData MyVertexProgram (VertexData v) {
    FragmentData i;
    i.position = UnityObjectToClipPos(v.position);
    i.uv = TRANSFORM_TEX(v.uv, _MainTex);
    return i;
}
```

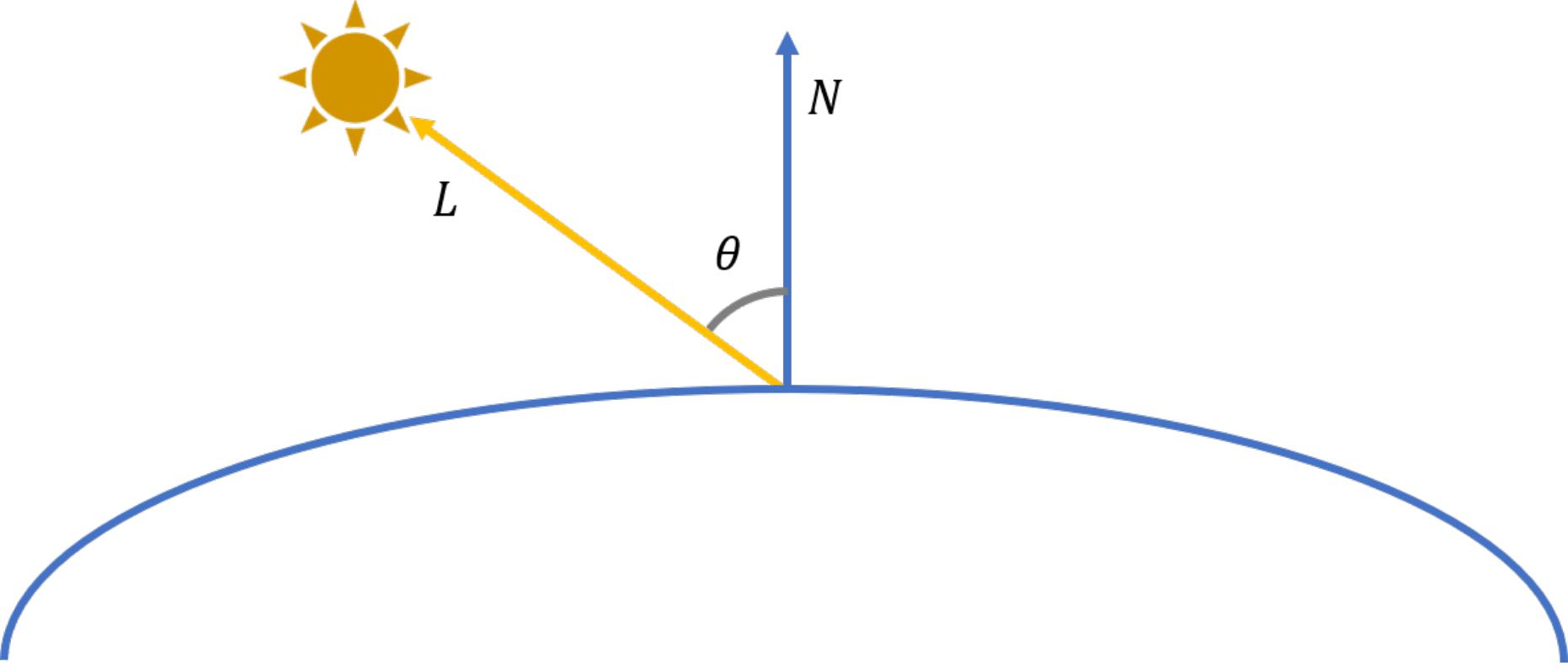
这样，我们便可以通过 GUI 中的 Tiling 和 Offset 对纹理进行变换了。



4. 光照模型

4.1 环境光和漫反射

在真实世界中，我们通过物体反射光来看到物体。而反射光中，漫反射是最常见的一种。在计算机中，我们常常使用 Lambert 光照模型来模拟漫反射效果。



Lambert 光照模型中认为漫反射光的光强仅与入射光的方向和反射点处表面法向夹角的余弦成正比。因此，我们可以将公式写为如下形式：

$$I_{diffuse} = I_{in} * \cos\theta = I_{in}(L \cdot N)$$

将这条公式用 Shader 表现出来，我们需要修改 Vertex Shader 和 Fragment Shader。首先我们需要 include `UnityStandardBRDF.cginc`，它会包含一些我们需要的函数。在 Fragment Shader 中，首先需要计算入射光的方向以及颜色

```
float3 lightDir = _WorldSpaceLightPos0.xyz;  
float3 lightColor = _LightColor0.rgb;
```

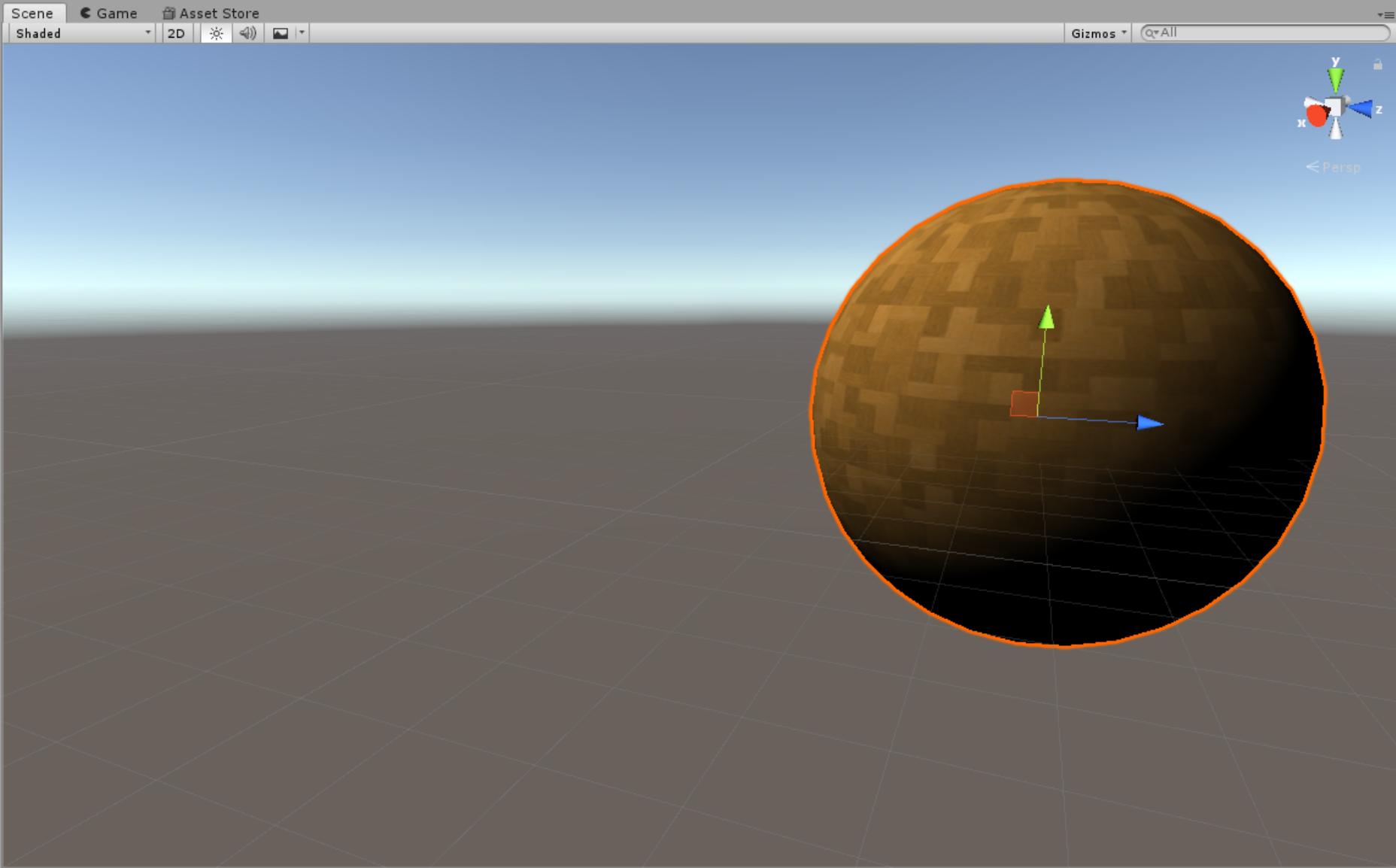
然后依照公式计算漫反射光，这里的 `DotClamped` 函数可以将负数的点乘结果截断至0，以防负数产生的错误光照效果。

```
float3 diffuse = lightColor * DotClamped(lightDir, i.normal);  
return float4(diffuse, 1);
```

如果在漫反射光中考虑物体本身的颜色

```
float3 diffuse = tex2D(_MainTex, i.uv).rgb * lightColor * DotClamped(lightDir, i.nor
```

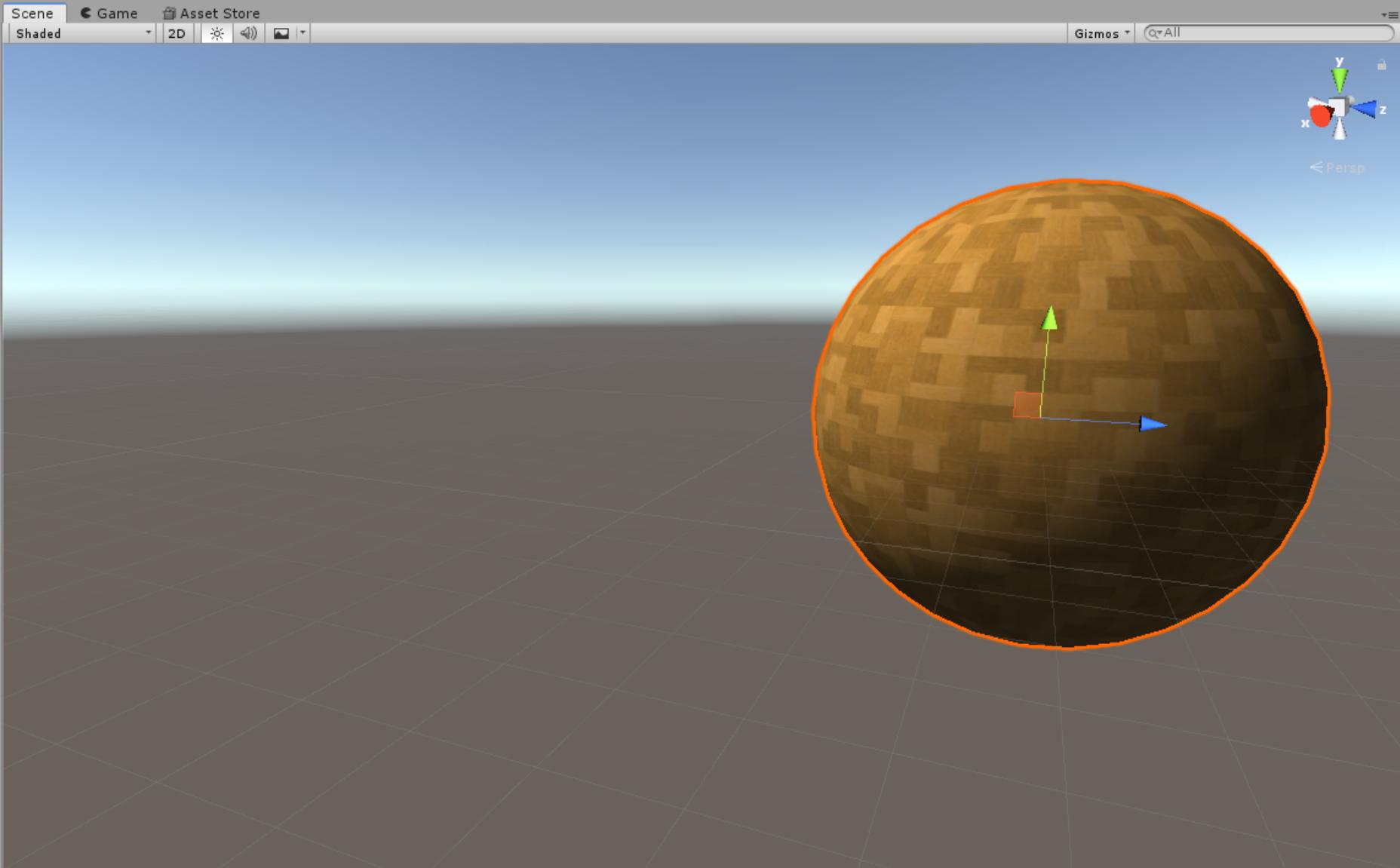
效果如下



此时会发现，球体没有被照射的地方表现得过暗。我们可以通过增加环境光效果来改善这一情况。
在Fragment Shader 中加入环境光。

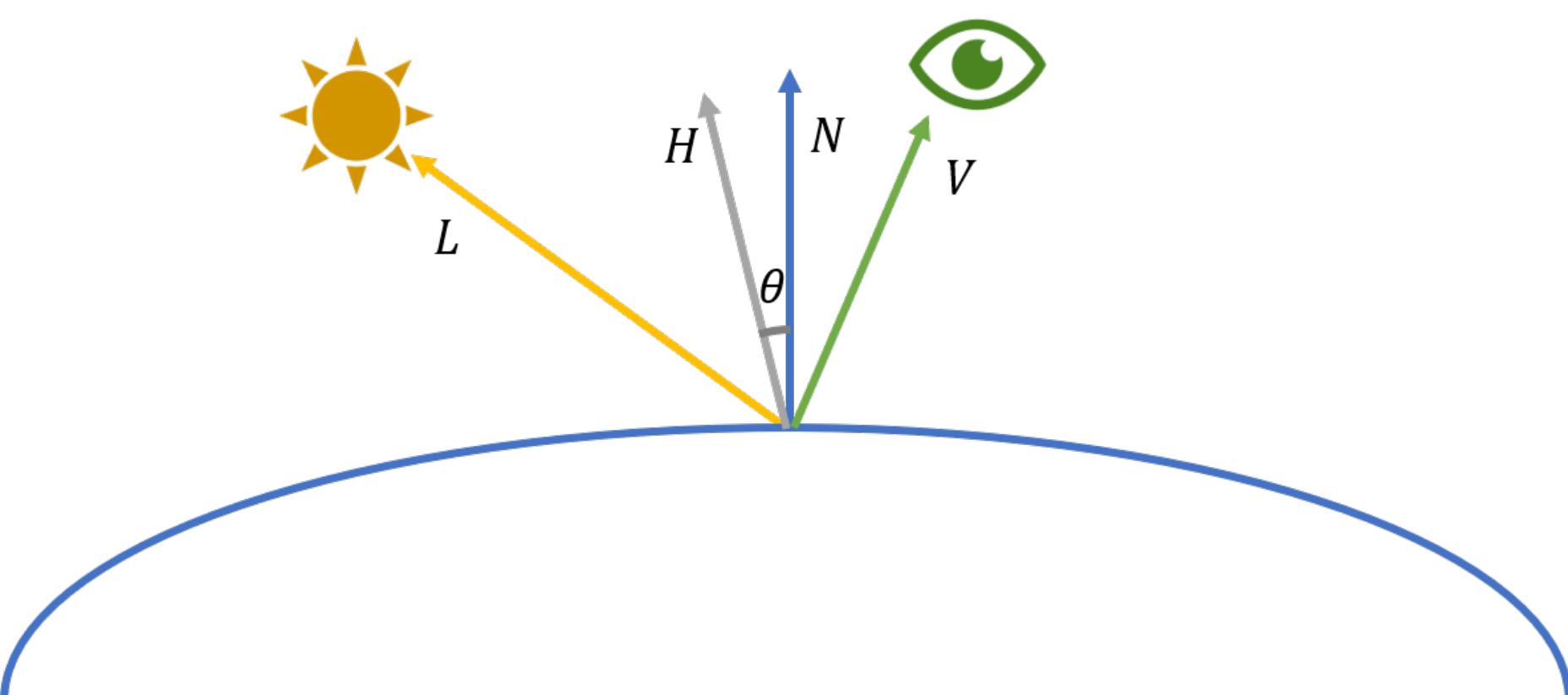
```
float3 diffuse = ...  
fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * tex2D(_MainTex, i.uv).rgb;  
return float4(ambient + diffuse, 1);
```

最终我们可以得到如下效果



4.2. Blinn Phong模型

镜面反射光是表现一些金属、光滑材质的重要效果，而 Blinn Phong 模型是一个经典的计算镜面反射光的光照模型。



Blinn Phong 模型引入了一个新的向量 H 的概念，它是光入射方向和视线方向之间角平分线的方向，也叫半矢量 (Half Vector)。可以通过以下公式求解

$$H = \frac{L + V}{\|L_V\|}$$

通过法线和半矢量之间的点积可以衡量镜面高光的亮度。直观的理解，当点积越大，表示两个向量越接近，即法线方向越接近光入射方向和视线方向的角平分线，换句话说，视线方向也就越接近镜面反射的出射光方向，此时镜面反射自然会增大。具体的，有如下公式

$$I_{specular} = (N \cdot H)^{n_{shininess}}$$

为了产生这样的效果，首先需要修改数据结构，在 `FragmentData` 中，我们会额外需要一个顶点在世界坐标系中的位置 `worldPos`，用以计算视线方向。

```
struct VertexData {  
    float4 position : POSITION;  
    float3 normal : NORMAL;  
    float2 uv : TEXCOORD0;  
};  
  
struct FragmentData {  
    float4 position : SV_POSITION;  
    float2 uv : TEXCOORD0;  
    float3 normal : TEXCOORD1;  
    float3 worldPos : TEXCOORD2;  
};
```

在 Vertex Shader 中，通过将局部坐标系乘上坐标系转换矩阵，计算 `FragmentData.worldPos`。

```
i.worldPos = mul(unity_ObjectToWorld, v.position);
```

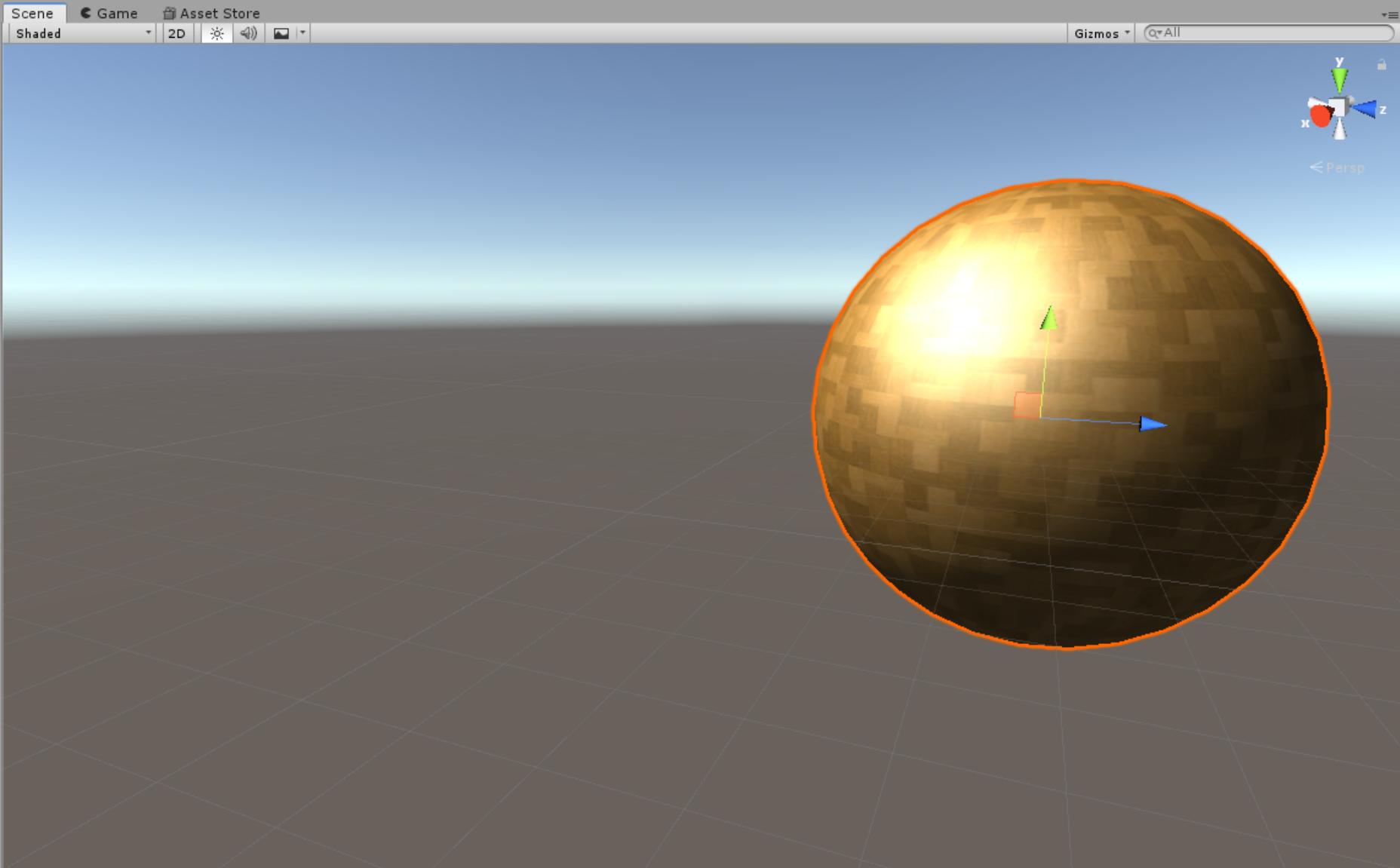
然后，在 Fragment Shader 中，视线方向 V 可以计算为

```
float3 viewDir = normalize(_WorldSpaceCameraPos - i.worldPos);
```

半矢量 H 可以计算为

```
float3 halfVector = normalize(lightDir + viewDir);
```

为了调节高光效果，我们将在 Properties 中添加浮点数 `_Shininess` 属性，对应公式中的乘方系数。按照公式编写，并调节 `_Shininess` 为 10，可以得到较好的镜面反射光效果。



5. 编写自定义的Shader GUI

Unity 中支持对各种 UI 界面进行自定义。程序员可以通过编写自定义的 Editor 脚本，为美术、策划提供直观的交互界面，方便他们调整程序的参数，实现理想的效果。这里我们介绍 Shader GUI 的自定义。

首先创建脚本 `CustomShaderGUI.cs`，使之继承 `ShaderGUI` 类。

```
using UnityEngine;
using UnityEditor;
using System;

public class CustomShaderGUI : ShaderGUI
{
    public override void OnGUI(MaterialEditor editor, MaterialProperty[] properties)
    {
        // Your custom GUI code here
    }
}
```

然后在 `MyShader.shader` 最后声明使用这个类作为自定义的 Editor。

```
Shader "Custom/MyShader"
{
    Properties
    {...}
    SubShader
    {...}
    CustomEditor "CustomShaderGUI"
}
```

现在，`MyShader` 已经使用了自定义的 Editor 了，但此时面板上将不显示任何可改参数，我们需要人为地添加参数。

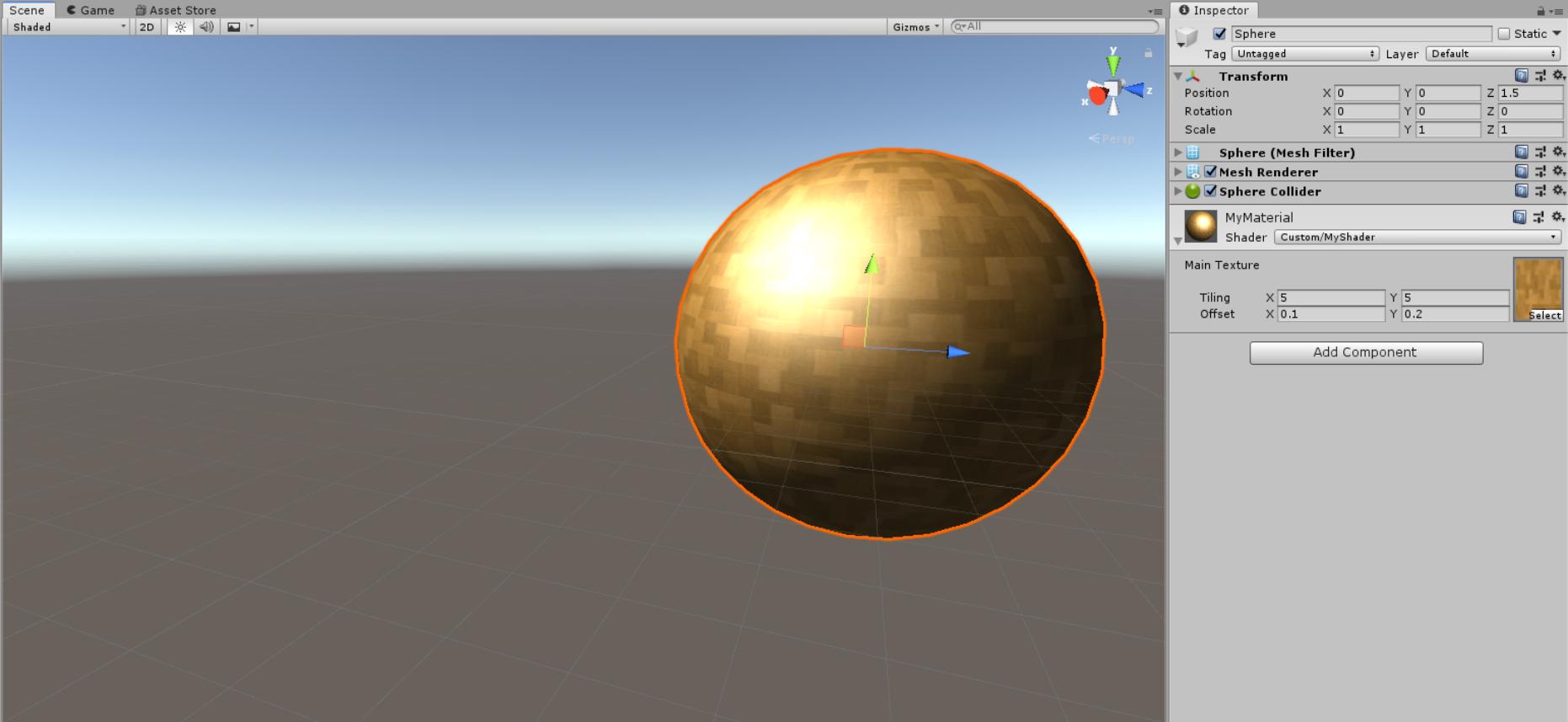
为了使后续操作方便，我们在 `CustomShaderGUI` 中定义变量

```
MaterialEditor editor;
MaterialProperty[] properties;
Material target;
```

然后，我们便可以在面板上添加材质属性了

```
public override void OnGUI(MaterialEditor editor, MaterialProperty[] properties)
{
    this.editor = editor;
    this.properties = properties;
    this.target = editor.target as Material;

    MaterialProperty mainTex = FindProperty("_MainTex", properties);
    GUIContent mainTexLabel = new GUIContent(mainTex.displayName);
    editor.TextureProperty(mainTex, mainTexLabel.text);
}
```



你也可以尝试修改上面的参数，来实现不同的参数调整。

此外，Shader 中还允许使用 Keyword（关键字），这和 C++ 中的定义宏很类似。通过关键字，我们可以实现类似开关的效果，如当一个 Keyword 被 Enable 时，Shader 代码中被 `#if keyword` ... `#endif` 包含的语句就会被执行，而反之则会被忽略。

要使用关键字，我们首先要在 Shader 的 Pass 中定义一个着色器特征，比如 `#pragma shader_feature USE_SPECULAR`。而在代码中使用时，我们可以这样使用

```
float3 specular = float3(0,0,0);
#if USE_SPECULAR
    specular = ...
#endif
```

这样，只有在其被 Enable 的时候，镜面反射光才会被计算。而在代码中，我们可以通过 `Material.EnableKeyword` 和 `Material.DisableKeyword` 函数来指定是否是用镜面高光。

为了让 GUI 界面看起来更友好，我们可以使用下拉栏来实现这一效果。

首先，我们需要在 `CustomShaderGUI` 中定义一个枚举类 `SpecularChoice`，它表示是否使用镜面高光。

```
enum SpecularChoice {
    True, False
}
```

在 `OnGUI` 中，我们首先需要判断当前的 Keyword 状态。

```
SpecularChoice specularChoice = SpecularChoice.False;
if (target.IsKeywordEnabled("USE_SPECULAR"))
    specularChoice = SpecularChoice.True;
```

然后创建下拉栏，同时判断下拉栏选项是否被更新，在更新时设置相应的 Keyword 状态。

```
EditorGUI.BeginChangeCheck();
specularChoice = (SpecularChoice)EditorGUILayout.EnumPopup(
    new GUIContent("Use Specular?"), specularChoice
);

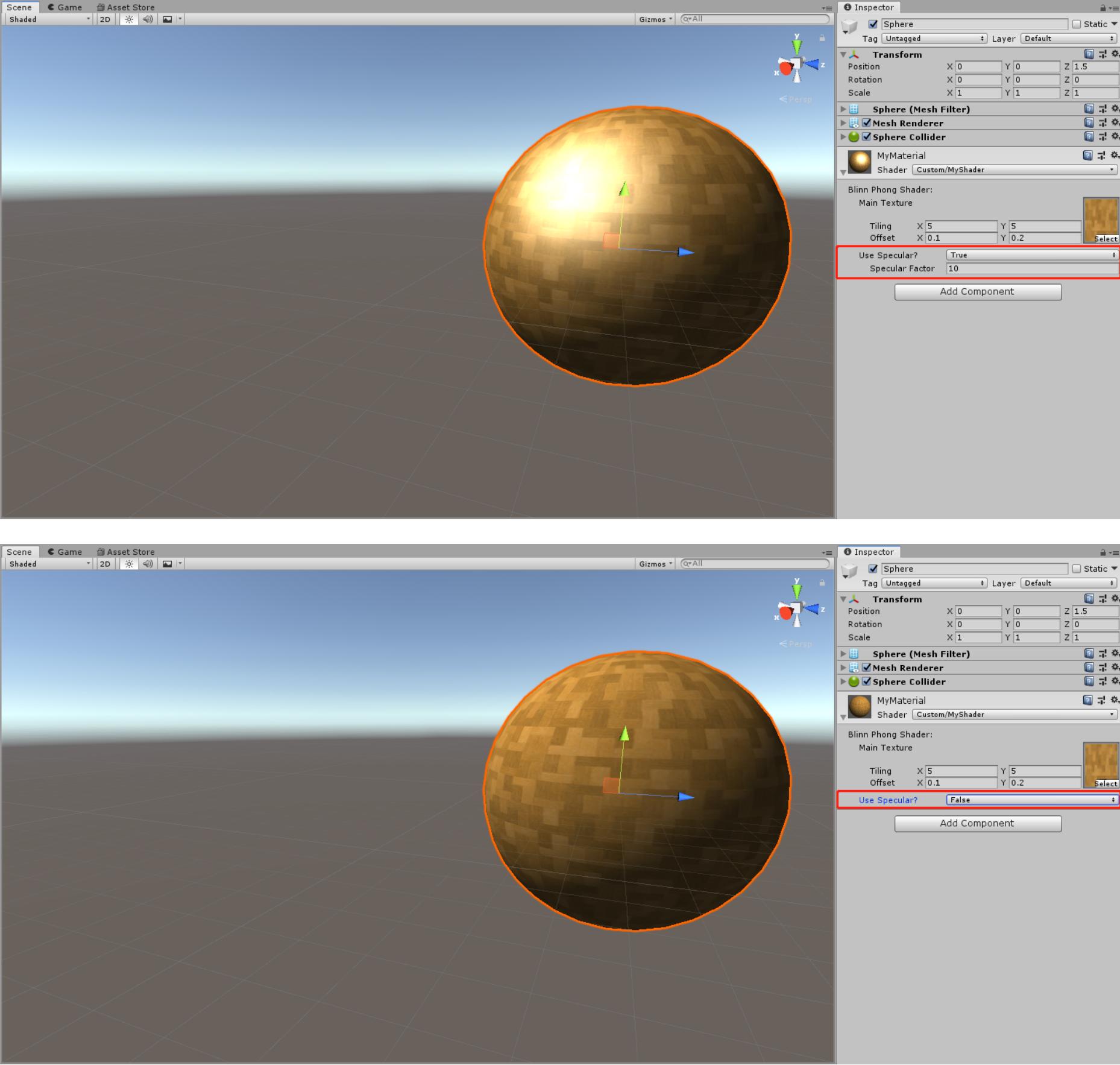
if (EditorGUI.EndChangeCheck()) {
    if(specularChoice == SpecularChoice.True)
        target.EnableKeyword("USE_SPECULAR");
    else
        target.DisableKeyword("USE_SPECULAR");
}
```

此时，我们已经实现了一个通过下拉栏选项修改渲染设置的功能。进一步的，我们希望当选中镜面高光时，可以进一步显示高光参数；而当没选中时隐藏这个参数。

在上面的基础上，我们对当前 Keyword 状态进行判断，并根据结果选择是否显示更多的选项。

```
if(specularChoice == SpecularChoice.True){
    MaterialProperty shininess = FindProperty("_Shininess", properties);
    GUIContent shininessLabel = new GUIContent(shininess.displayName);
    editor.FloatProperty(shininess, "Specular Factor");
}
```

最终实现效果如下



6. 作业要求

请你最终实现一个 Shader，其中能通过 GUI 面板下拉栏选择不同的渲染效果，如法线可视化、BlinnPhong光照模型，在不同选项中提供不同的参数设置。并在之后应用到你的游戏中吧！

