SJTU-SE

自定义特效 shader

实验报告

陈　　诺

516030910199

目录

1. 实验目的与要求

1.1 实验目的

1. 了解 Shader 和它在游戏中的作用
2. 学会编写 Shader 实现一些简单的视觉特效
3. 理解基于物理的渲染模型

1.2 实验要求

1. 基于物理的渲染模型（必做项，50%）
   - ✓ 了解双向反射分布函数（ BRDF ）的原理以及代码实现
   - ✓ 在本页提供的 Unity Package 基础上，参照作业文档实现 Cook-Torrance 模型
   - ✓ 请在实验报告中描述实现方法，并使用和作业文档中类似的材质球矩阵图展示效果
2. 非真实感渲染（必做项，50%三选一）
   - ✓ 卡通风格渲染（推荐）
     风格化卡通高光渲染
     素描风格渲染
     请在实验报告中描述实现思路、方法，并展示效果
3. 屏幕后处理效果（可选加分项）
     Bloom 效果
     运动模糊
   - ✓ 全局雾效
     景深效果
     其他:毛发渲染

2. 实验步骤

2.1 基于物理的渲染模型

2.1.1 BRDF 函数原理

Bidirectional Reflectance Distribution Function，双向反射分布函数

物体表面将光能从任何一个入射方向反射到任何一个视点方向的反射特性，即入射光线经过某个表面反射后如何在各个出射方向上分布

BRDF 在游戏、电影中有着广泛的应用。例如下图迪士尼的高口碑电影超能陆战队：

图 1. 基于 BRDF 渲染的场景图（Disney《超能陆战队》）

在渲染方程中，反射部分用公式描述为：

$$L_r(v) = \int_{\pi} L_i(l) f_r(l,v)(n \cdot l) dl$$

其中：

$$f_r = k_d f_d + k_s f_S$$

偶尔也会用到最简单的 Lambertian BRDF：

$$R(l) = \pi f(l,v)$$

在实践中会遇到各种 BRDF 的变种：BSSBRDF、SVBRDF 等等。

在本次作业中，模拟的 BRDF 用公式描述为：

$$f(l,v) = \begin{cases} k_d f_d + K_s \frac{D(h)F(v,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)} & \theta < 90^\circ \\ 0 & \text{else} \end{cases}$$

2.1.2 BRDF 核心代码

本部分的场景在作业/Assets/Scene/ShaderScene 下。
本次作业中需要填写 BRDF 中的 D、F 和 G 项。

代码实现参照 BRDF 实现原理以及 unity 的 built-in shader：

1. D 项

```
float GGX_D(float roughness, float NdotH)
{
    //// TODO: your implementation
    float roughPow = pow(roughness, 2);
    float tmp = ((NdotH * NdotH)*(roughPow - 1)+1);
    return roughPow / (3.1415926f * pow(tmp, 2));
}
```

2. F 项

```
float3 Schlick_F(half3 R, half cosA)
{
    //// TODO: your implementation
    // return float3(1,1,1);
    half t = pow(1 - cosA, 5);
    return R + (1 - R) * t;
}
```

3. G 项

```
float CookTorrence_G
(float NdotL, float NdotV, float VdotH, float NdotH)
{
     //// TODO: your implementation
     // return 1;
     float tmp = (2 * NdotH) / VdotH;
     if (NdotV > NdotL) {
         return min(1, tmp * NdotL);
     }
     return min(1, tmp * NdotV);
}
```

2.1.3 实现效果

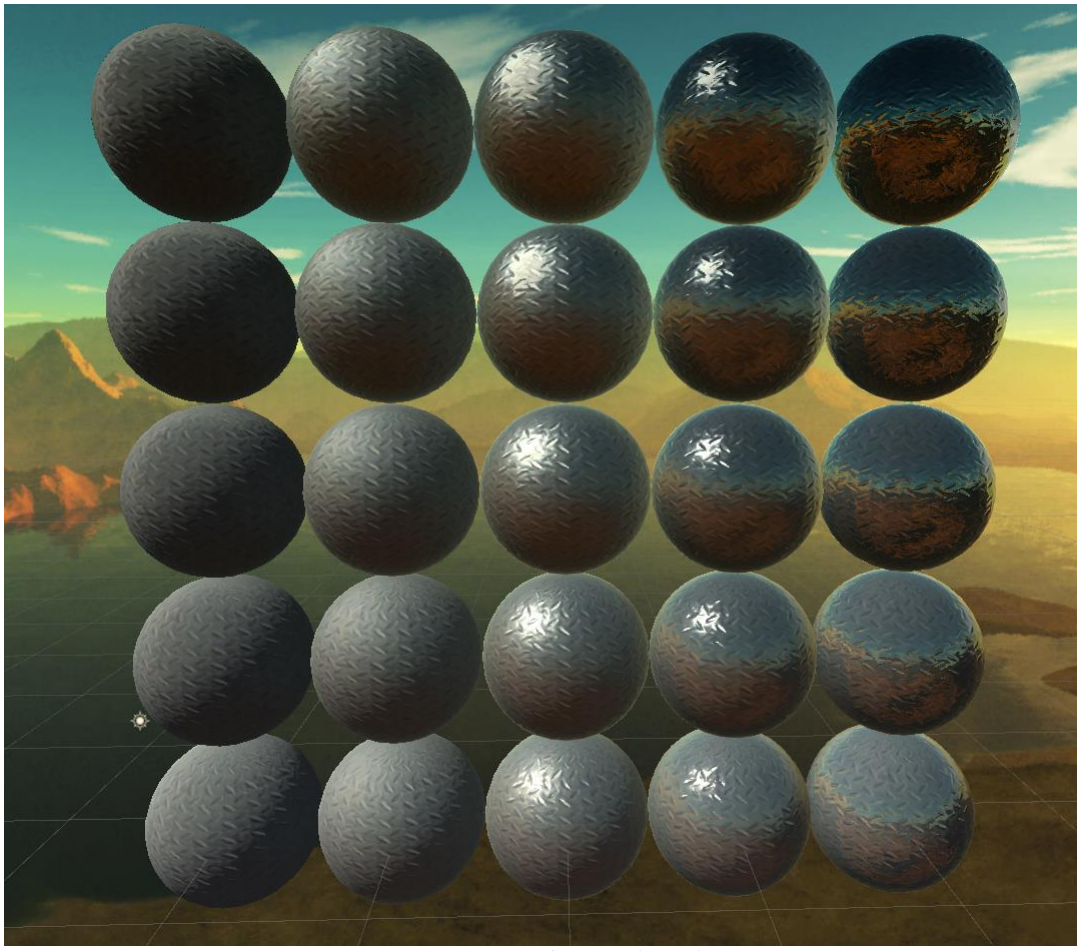

图 2. BRDF 实现矩阵球效果

2.2 非真实感渲染（卡通风格渲染）

本部分实现效果在/Assets/Scene/CartoonScene 下。

2.2.1 实现原理

在本次作业中我通过渲染轮廓线以及用纯色表示高光来实现卡通风格的渲染。为了使效果更好，本部分搭建了一个卡通场景，并且使用两种高光（颜色不同且范围不同）来使得渲染效果更好。

图 3. 卡通风格场景

场景中的人物、水井、木头及背包、房屋都使用 CartoonShader 来渲染。

1. 轮廓线实现原理

在渲染时，将模型背面的面片全部使用轮廓线的颜色渲染，并且把模型的顶点向外扩充，利用多出来的那部分面片的颜色来充当轮廓线。

这种方法有个缺点就是不适用与低模，因为低模的顶点数量较少，在进行扩充时轮廓线会出现间隔的情况。不仅如此，高光的渲染也会变成一整块的色块。所以使用低模时，这种方法就不再适合了。

图 4. 使用 CartoonShader 渲染的低模

可以看到，图中的轮廓线全部离散间断了并且高光的效果也不是很好。

2. 高光实现原理

为了实现整块的高光色块，在 CartoonShader 中不再使用 Blinn-Phong 模型中的方法，而是将法线点乘视角方向和一半的值与一个预设的阈值比较，根据比较结果将高光系数设定为 0 或 1 的独热编码。

这样实现出来的高光效果会有锯齿现象（因为 0 和 1 会有突变）。为了获得更好的渲染效果，就需要避免 0 到 1 的突变。实现这个功能很简单，只需要用 smoothstep()来代替 step()函数即可。

2.2.2 核心代码

1. 轮廓线

```
Pass
    {
     // 只渲染背面的三角面片，切除正面三角面片
    NAME "OUTLINE"
    Cull Front

    float _Outline; // 轮廓线宽度
    fixed4 _OutlineColor; // 轮廓线颜色

    v2f vert(a2v v) {
        v2f o;

        float4 pos = mul(UNITY_MATRIX_MV, v.vertex);

        float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);
        normal.z = -1;// 扩展顶点
        pos = pos + float4(normalize(normal), 0) * _Outline;
        o.pos =mul(UNITY_MATRIX_P, pos);

        return o;
    }

    float4 frag(v2f i) : SV_Target {
        return float4(_OutlineColor.rgb, 1); // 直接返回轮廓线颜色
    }

    ENDCG
}
```

2. 高光

```
Pass {
    // 正面的面片，剔除背面的面片
    Tags { "LightMode" = "ForwardBase" }
    Cull Back

    // 声明纹理、高光颜色、高光阈值等
    fixed4 _Color;
```

```
8.      ...
9.
10.
11.
12.     v2f vert(a2v v) {
13.        // 计算法线方向和顶点位置
14.     }
15.
16.     float4 frag(v2f i) : SV_Target {
17.         // 计算各个矢量
18.                 ...
19.         // 计算反射率
20.         fixed4 c = tex2D(_MainTex, i.uv);
21.         fixed3 albedo = c.rgb * _Color.rgb;
22.         // 计算漫反射系数 diff 并对 ramp 纹理采样
23.
24.         fixed3 diffuse = _LightColor0.rgb * albedo * tex2D(_Ramp, float2(diff
    , diff)).rgb;
25.         // 计算高光
26.         fixed spec = dot(worldNormal, worldHalfDir);
27.         fixed w = fwidth(spec) * 2.0;
28.         fixed3 specular = _Specular.rgb * lerp(0, 1, smoothstep(-
    w, w, spec + _SpecularScale - 1)) * step(0.0001, _SpecularScale);
29.         fixed3 specular2 = _Specular2.rgb * lerp(0, 1, smoothstep(-
    w, w, spec + _SpecularScale2 - 1)) * step(0.1, _SpecularScale2);
30.         return fixed4(ambient + diffuse + specular + specular2, 1.0);
31.     }
32.
33.     ENDCG
34. }
```

2.2.3 实现效果

这里以场景中的一个包裹着木头的筐为例。

1. 总体效果

图 5. 木筐总体效果

可以在总体效果中看到明显的轮廓线和两种颜色不同的高光。

2. 变量

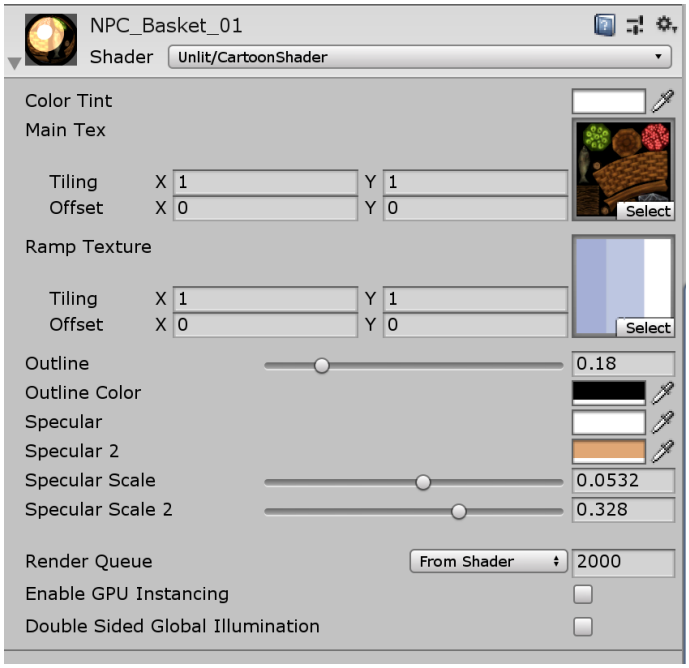

图 6. 选择面板

与轮廓线有关的选项为 Outline、OutlineColor。这两个变量一个决定了轮廓线的颜色，另一个决定了轮廓线的宽度。

与高光有关的选项为 Specular、Sepcular 2、Specular Scale、Specular Scale2，分别决定了高光的颜色和高光的范围。其中，Specular 是颜色更浅的高光，所以范围应该更小，于是 Scale 的滑动条范围要远远小于 Specular 2 的滑动条范围。

2.3 屏幕后处理效果

本次作业选做的屏幕后处理效果为全局雾效。

场景在/Assets/Scene/FogScene 下。

2.3.1 实现原理

1. 屏幕后处理

屏幕后处理技术是指在渲染完整个场景之后，直接对已经得到的图像进行处理来获得特效。

实现这一效果主要借助了 unity 中的函数：

```
1. MonoBehaviour.OnRenderImage(RenderTexture src, RenderTexture dst)
2. public static void Blit(…)
```

其中第一个函数是用来截取渲染好的屏幕图像 src，并且将处理好的图像 dst 返回给屏幕。

第二个函数是对渲染纹理处理的函数。

整个实现的流程大致如下：

1. 在 Main Camera 上添加一个屏幕后处理的脚本，截取前文提到的 RenderTexture src
2. 使用 Graphics.Blit 处理 src 获得 dst
3. 将 RenderTexture dst 返回给屏幕。

在核心代码中，要计算屏幕四个 point 的位置。这是通过深度纹理和 camera 的 transform 来确定的。确定四个 point 之后，就将生成的材质渲染到四个 point 围成的四边形中即可。

2. 雾的模拟

为了模拟雾，我们需要定义雾的基本颜色、雾的浓度、雾的位置等物理信息。

在本次作业中，使用线性公式来模拟雾的浓度随着高度的升高而降低的效果。雾的最深浓度和雾的颜色取决于面板中的设置。

雾的浓度变化定义为：

$$density = \frac{(Height_{end} - Height_{current})}{Height_{end} - Height_{begin}}$$

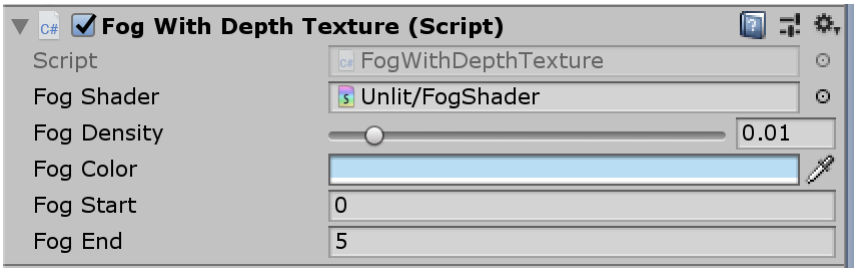

图 7. 雾的选择面板

2.3.2 核心代码

为了实现这个效果，本次作业中先定义了一个屏幕后处理的基类：/Assets/Scripts/PostEffectsBase.cs

在 PostEffectsBase.cs 中，核心代码为：

```
1.  protected void CheckResources()  // 检查各种条件是否满足
2.
3.  protected bool CheckSupport()  // 检查平台等
4.
5.  protected void NotSupported()  // 平台等资源不满足时的报错
6.    // 在开始时调用，并调用检查函数
7.  protected void Start()
8.    // 指定 shader 来处理材质
9.  protected Material CheckShaderAndCreateMaterial(Shader shader, Material mate
    rial)
```

同时新建 FogWithDepthTexture.cs 脚本，继承自 PostEffectsBase.cs，并且将本脚本绑定到 Main Camera 上。

新建 FogShader.shader，在 FogWithDepthTexture.cs 中声明一个 shader，并且将 FogShader 绑定到 FogWithDepthTexture 上。

在 FogWithDepthTexture.cs 中，核心代码为：

```
1.  // 定义 Shader 和 Material
2.  // …
3.  // 定义 Camera（为了方便计算世界坐标系下的位置）
4.  // …
5.  // 定义 Camera 的 transform，便于计算世界坐标系中的位置
6.  // …
7.  // 定义雾的浓度、颜色、开始位置和结束位置
8.  // …
9.  // 这里即前文提到的 OnRenderImage(RenderTexture src, RenderTexture dst)
10. void OnRenderImage(RenderTexture src, RenderTexture dest)
11. {
12.     if (material != null)
13.     {
```

```
        // 屏幕后处理中对应的平面的四个角
        Matrix4x4 frustumCorners = Matrix4x4.identity;
        // 获得这四个 point 的位置
        // …
        // 将这四个 point 放进 frustumCorners 中
        frustumCorners.SetRow(0, bottomLeft);
        // …
        // 设置 Material 属性
        material.SetMatrix("_FrustumCornersRay", frustumCorners);
       // …

        // 将 frustumCorners 中的值传递给其他材质并且渲染出结果
        Graphics.Blit(src, dest, material);
    }
    else
    {
        Graphics.Blit(src, dest);
    }
}
```

2.3.3 最终效果

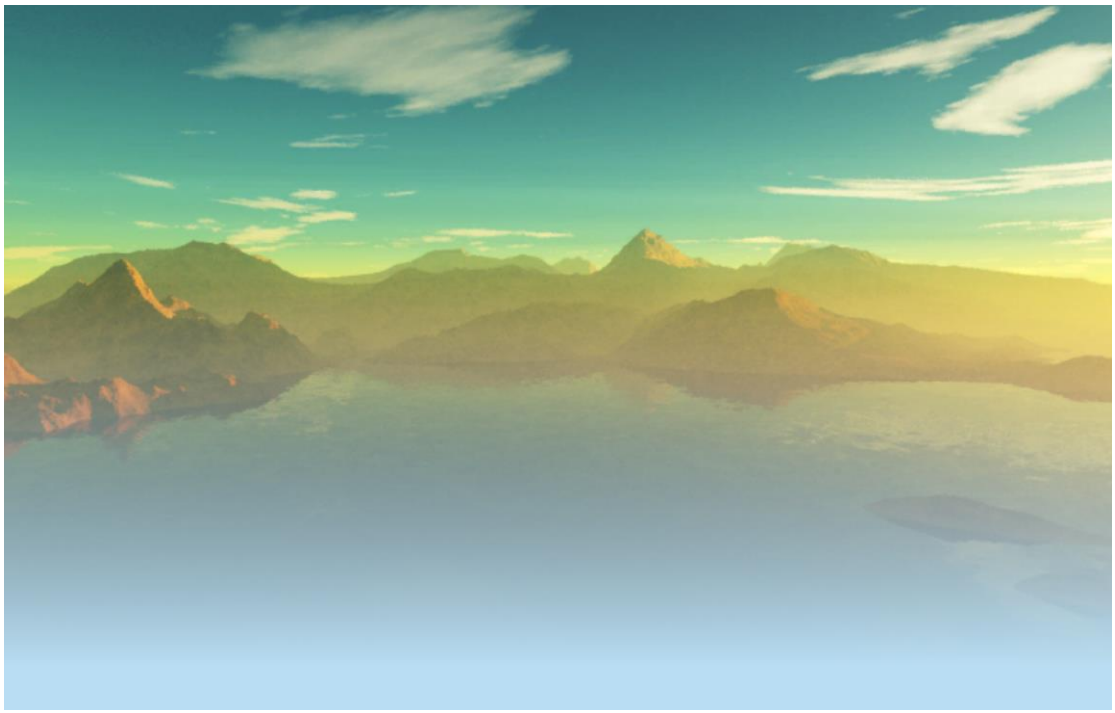

图 8. 雾的效果

这里为了效果明显，把雾的颜色选为了蓝色。

3. 实验有感

Unity 是一个非常强大的引擎工具，使用 unity 的有经验的大师也很多。Unity shader 中还有很多东西可以探索。本次作业学习到了很多经验，看了很多大师做出来的令人经验的效果，不由得感叹任重道远，还有很多需要学习和实践的知识。

Unity 开发中利用 shader、skybox、model 就可以实现非常酷炫的效果，希望将来可以学到更多有关本方面的知识。

4. 参考

1. unity built-in shaders:
   https://unity3d.com/get-unity/download/archive
2. 卡通风格渲染
   冯乐乐 《unity shader 入门精要》