第三次实验报告

姓名：陈　　诺

学号：516030910199

目录

1 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. 学会自定义一个 Shader 的 GUI
4. 学会使用 Shader Debug 工具
5. 能够利用 Shader 实现特别的渲染效果

1.2 基本实验要求

1.2.1 编写 Vertex Shader 和 Fragment Shader

1. 实现展示模型法线方向的 Shader
2. 实现显示纹理贴图的 Shader
3. 实现简单光照渲染的 Shader
4. 实现 Lambert 模型漫反射、全局光、Blinn Phong 模型镜面反射
5. 将纹理、镜面反射系数作为可调参数供用户选择

1.2.2 编写 Shader 的 GUI

1. 通过继承 Unity 自带的 ShaderGUI 类，实现自定义的 ShaderGUI
2. 实现在自定义 GUI 中调整各项参数 Shader 参数，如贴图、高光颜色、镜面高光系数等
3. 实现在自定义 GUI 中的 Shader 效果切换，使得用户可以通过下拉框选取不同的渲染效果，包含法向 Shader、纹理贴图 Shader、光照 Shader

1.2.3 了解、尝试至少一个 Debug 工具

1. Unity Frame Debugger (Unity 自带工具)
2. RenderDoc (推荐，但 Mac 可能不支持)
3. Nvidia Nsight (Nvidia 显卡专用)
4. Visual Studio Frame Debugger (Visual Studio 工具)
5. XCode Instruments (XCode 工具)

1.2.4 撰写实验报告

1.3 进阶作业要求

1.3.1 实现一些更复杂的效果 Shader 效果

1. 叠加波实现水面、海浪动态 Shader
2. 多张纹理实现不同地形自然混合 Shader
3. 使用噪声纹理实现动态消融效果
4. 基于 Geometry Shader 实现毛发效果
5. 基于 Geometry Shader 实现草地效果
6. ...

1.3.2 在实验报告中描述实现原理、实现过程

2. 实验步骤

2.1 基础要求部分

2.1.1 Vertex Shader 和 Fragment Shader

本部分按照实验文档实现了基本的 vertex shader 和 fragment shader，实现了纯色 shader、法向量 shader、纯纹理 shader、漫反射 shader 和高光 shader。

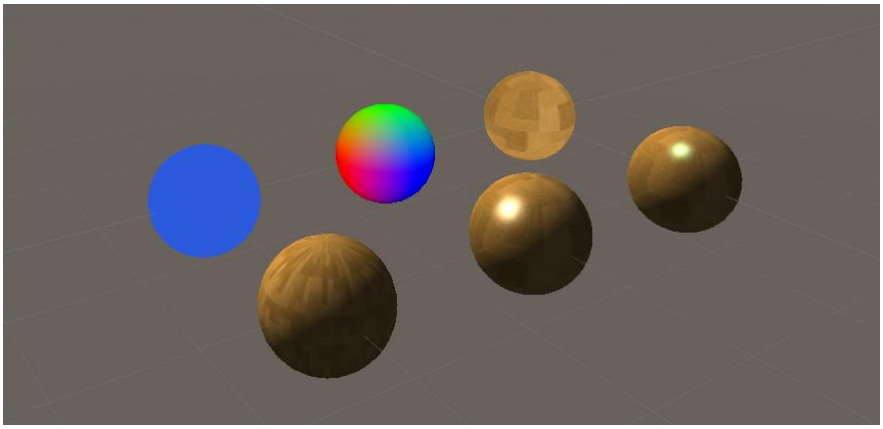

图 1. 各 shader

从后往前从左到右依次是：纯色 shader、法向量 shader、纯纹理 shader、漫反射 shader、高光 shader 和用户可调 shader。其中最后一个用户可调 shader 详见

2.1.2 节。

2.1.2 Shader GUI

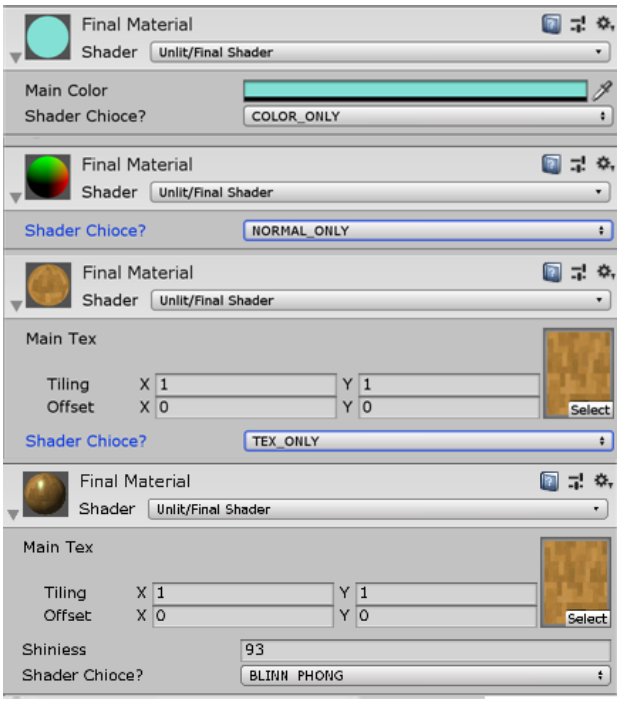

图 2. 不同的 shader choice 及其面板

新建一个 Unlit Shader,命名为 Final Shader,然后为 Final Shader 声明一个 CustomEditor：

```
1.  CustomEditor "CustomShaderGUI"
```

同时在 Assests 文件夹下新建文件夹“Editor”，然后在 Editor 下新建 C#文件 CustomEditor.cs。这时 CustomEditor.cs 就成为了本 shader 的 GUI。

在 CustomEditor.cs 中，声明一个 Enum popup 选项：Shader Choice 让用户选择 shader 类型。

```
1.  enum ShaderChioce
2.  {
3.      COLOR_ONLY,
4.      NORMAL_ONLY,
5.      TEX_ONLY,
6.      BLINN_PHONG
7.  }
8.
9.  public override void OnGUI(){
```

```
    ...
    ShaderChioce shaderChioce = ShaderChioce.BLINN_PHONG;
    EditorGUI.BeginChangeCheck();
    shaderChioce = (ShaderChioce)EditorGUILayout.EnumPopup(
        new GUIContent("Shader Chioce?"), shaderChioce
    );

```

然后 Disable 所有的 Keyword，再根据用户的选择来 EnableKeyword。

```
if (EditorGUI.EndChangeCheck())
{
    // Disable 所有的 Keyword
    target.DisableKeyword("USE_TEXTURE");
    target.DisableKeyword("USE_COLOR");
    target.DisableKeyword("USE_NORMAL");
    target.DisableKeyword("USE_BLINN");
    // 检查 shader choice 并 Enable 相应的 Keyword
    // 另外三个为了篇幅省略
    if (shaderChioce == ShaderChioce.COLOR_ONLY)
    {
        target.EnableKeyword("USE_COLOR");
    }
}
```

根据 Enable 的 Keyword 来更新面板中的组件。

```
if (target.IsKeywordEnabled("USE_COLOR"))
{
    shaderChioce = ShaderChioce.COLOR_ONLY;
    MaterialProperty mainColor = FindProperty("_MainColor", properties);
    GUIContent mainColorLaber = new GUIContent(mainColor.displayName);
    editor.ColorProperty(mainColor, mainColorLaber.text);
}
// 另三个选项基本同上
```

在 Final Shader 中，需要修改 MyVertProgram()和 MyFragProgram()来适应 CustomEditor 中的选择。

首先声明 Keyword:

```
#pragma shader_feature USE_COLOR
#pragma shader_feature USE_NORMAL
#pragma shader_feature USE_TEXTURE
#pragma shader_feature USE_BLINN
```

然后在 MyVertProgram()和 MyFragProgram()中根据 keyword 来选择返回值：

```
v2f MyVertProgram(a2v v) {
    v2f i;
    i.pos = UnityObjectToClipPos(v.vertex);

    #if USE_NORMAL
    i.worldNormal = UnityObjectToWorldNormal(v.normal);
    #endif

    #if USE_TEXTURE
    i.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    #endif

    #if USE_BLINN
    i.worldNormal = UnityObjectToWorldNormal(v.normal);
    i.worldPos = UnityObjectToClipPos(v.vertex).xyz;
    i.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    #endif

    return i;
}
```

MyFragProgram()原理同上。

2.1.3 了解 Debug 工具

我选择的 debug 工具为 RenderDoc，主要考虑到 RenderDoc 可以直接集成到 Unity 中，在实际调试时很方便。

安装 RenderDoc 并从重启后，在 Unity 中就可以自动关联 RenderDoc。

在 Game tab 中右键选择打开 RenderDoc，在运行时就能看到 RenderDoc 的 icon。

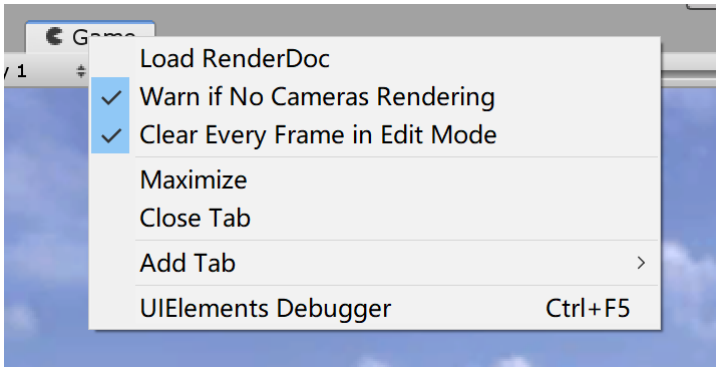

图 3. 打开 RenderDoc

图 4. RenderDoc 的图标

在运行程序时，只要点击 RenderDoc 的图标就能够截取当前帧，在 RenderDoc 中会保留当前帧的数据，并且能够看到相应的信息，在 RenderDoc 的 Timeline，Pipeline State，Event Browser，API Calls，Mesh Output，Texture Viewer 窗口中就能够看到我们需要的数据了。

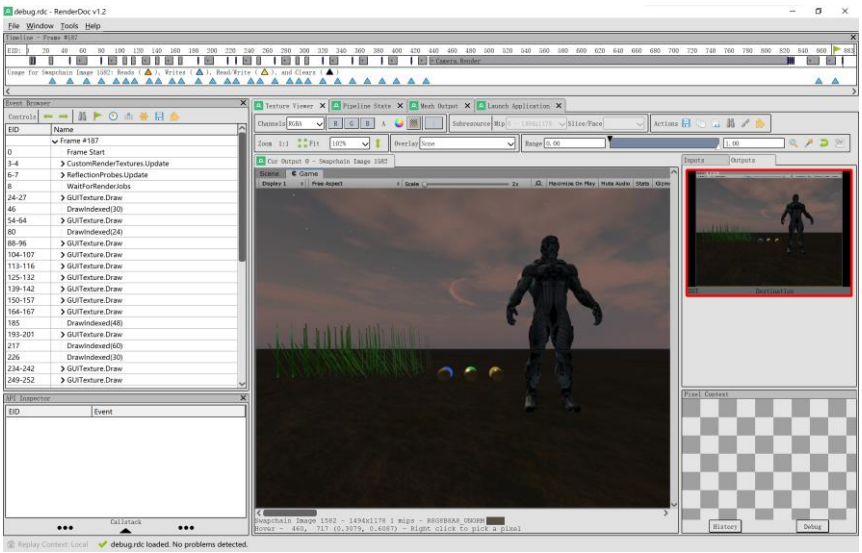

图 5. RenderDoc 截取的信息

3. 进阶要求部分

3.1 基于 Geometry Shader 实现草地效果

Geometry Shader 是从 DirectX10 开始支持的一个新特性。顶点着色器和片段着色器是必选的两个渲染管线，几何体着色器是可选的一个管线。

几何着色器在渲染 pipeline 中位于顶点着色器和片段着色器之间，和顶点着色器有相似性。但是顶点着色器是逐顶点操作，而几何着色器是逐图元操作，并且输入和输出都是图元。（输入的图元是点，输出的图元是三角形流）

3.1.1 实现原理

在.shader 文件中定义几何着色器，并且在几何着色器中实现如下原理：

1. 设置顶点着色器向几何着色器输出的最大顶点数量
2. 设置草的位置、高度、宽度等属性
3. 生成草的叶片网格
4. 使用随着_Time 变化的变量来模拟风
5. 使用风来改变叶片的位置
6. 将叶片 Append 到输出的三角形流中

3.1.2 实现过程

新建一个 Material: geom-grass 和一个 Shader: geom-grass，并且在场景中新建一个 Plane 并且将 geom-grass material 绑定到 Plane 上。

然后在 geom-grass shader 中声明需要的一些 properties：

```
Properties{
    // 草的纹理
    _MainTex("Albedo (RGB)", 2D) = "white" {}
    // 用 alpha 纹理来决定草的形状
    _AlphaTex("Alpha (A)", 2D) = "white" {}
    // 草的高度和宽度
    _Height("Grass Height", float) = 3
    _Width("Grass Width", range(0, 0.1)) = 0.05
}
```

其中_MainTex 选择一张绿色的图片即可，_AlphaTex 需要选择一张黑白图片勾勒出草的轮廓。

图 6. _AlphaTex

（注：蓝色的线条为实验报告中方便查看添加）

在 Pass{}段代码块中声明如下：

```
// 打开 AlphaToMask 来通过 alpha to coverage 呈现草的具体轮廓
AlphaToMask On

// 定义顶点着色器、片段着色器和几何着色器
#pragma vertex vert
#pragma fragment frag
#pragma geometry geom

// 着色器编译目标级别
#pragma target 5.0
```

其中的 target 需要大于 4.0（为了支持几何着色器）。

然后完成几何着色器函数：

```
// 设置顶点着色器向几何体着色器输出的最大顶点数量
[maxvertexcount(30)]
void geom(point v2g points[1], inout TriangleStream<g2f> triStream)
{
    // 设置草的位置
    float4 root = points[0].pos;
    // 给草的属性设置随机性
    float random = sin(UNITY_HALF_PI * frac(root.x) + UNITY_HALF_PI * frac(root.z));
    _Width = _Width + (random / 50);
```

```
    _Height = _Height + (random / 5);

    // 生成一片叶子的网格
    for (int i = 0; i < vertexCount; i++)
    {
        // 假设顶点的法线为(0, 0, 1)
        v[i].norm = float3(0, 0, 1);
        ...
        // 生成一个随着_Time 变化的风，来促使草摇摆
        float2 wind = float2(sin(_Time.x * UNITY_PI * 5), sin(_Time.x * UNITY_PI * 5));
        // 使得草的位置随着风而变化
         …
    }

    for (int p = 0; p < (vertexCount - 2); p++) {
        // 向三角形流中 append 生成的三角形
    }
}
```

这时，开始时设置的 Plane 已经有如下的效果了：

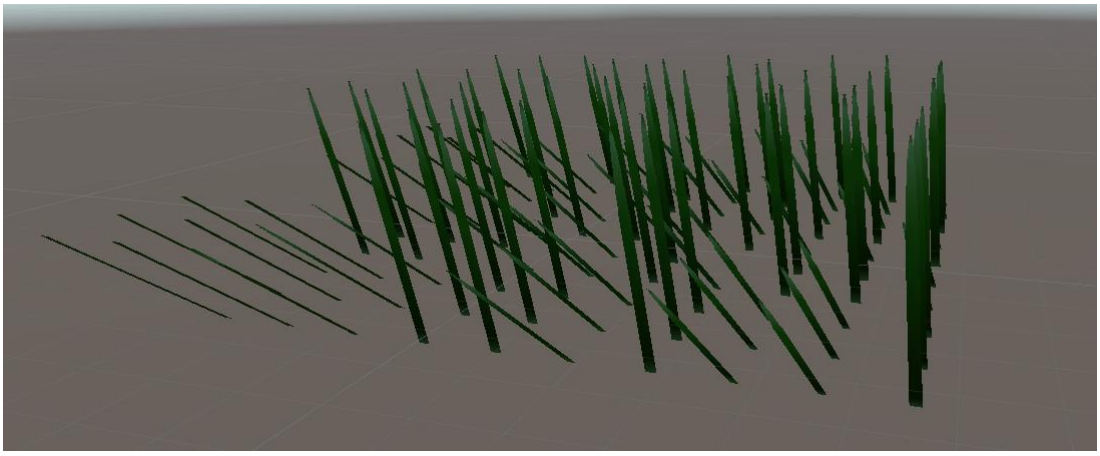

图 7. 初始的草地效果

此时可以进行进一步的优化。

新建一个 c#文件: GrassSpaner.cs，并且将其绑定到 Main Camera 上。

在 GrassSpanner.cs 中生成 250x250 片草，铺成比较浓密的草地。

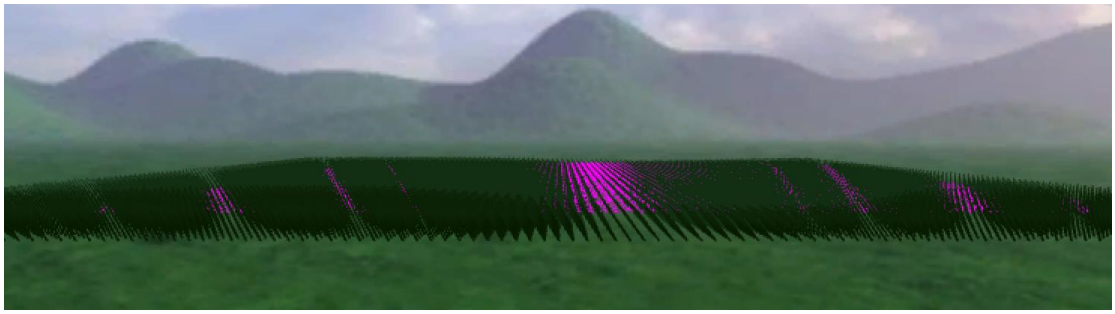

图 8．较为浓密的草地

3.2 使用噪声纹理实现动态消融效果

3.2.1 实现原理

使用噪声图片来实现消融效果使用到的方法是透明度测试。

具体的操作方法为对噪声纹理取样，并且将结果和控制消融效果的阈值进行比较。如果小于阈值就把对应的像素使用 clip()函数裁剪掉，在裁剪的边缘使用颜色与原有纹理颜色混合来显示出烧焦的效果。

3.2.2 实现过程

首先向场景中加入一个模型，本次实验中选取了一匹小马。

新建 material：Dissolve，新建 Shader：Dissolve，并且将 material 绑定到模型上，将 shader 绑定到 material 上。

然后在 Dissolve.shader 文件中的 properties 中声明以下变量：

```
1.  Properties
2.      {
3.          // 控制消融程度
4.          _BurnAmount("Burn Amount", Range(0.0, 1.0)) = 0.0
5.          // 烧焦时的线宽
6.          _LineWidth("Burn Line Width", Range(0.0, 0.2)) = 0.1
7.          // 漫反射纹理
8.          _MainTex("Base (RGB)", 2D) = "white" {}
9.          // 法线纹理
10.         _BumpMap("Normal Map", 2D) = "bump" {}
11.         // 火焰的颜色值
12.         _BurnFirstColor("Burn First Color", Color) = (1, 0, 0, 1)
13.         _BurnSecondColor("Burn Second Color", Color) = (1, 0, 0, 1)
14.         // 噪声纹理
15.         _BurnMap("Burn Map", 2D) = "white"{}
16.     }
```

然后在 SubShader 中定义消融需要的 Pass{}模块：

```
Pass {
    Tags { "LightMode" = "ShadowCaster" }
    CGPROGRAM
    Cull Off
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_shadowcaster
 }
```

其中要注意的是使用 Cull off 关闭了 shader 的面片剔除，使得模型内部的面片也被渲染（因为消融会导致内部的面片暴露）。

然后需要定义顶点着色器：

```
v2f vert(a2v v) {
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    o.uvMainTex = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uvBumpMap = TRANSFORM_TEX(v.texcoord, _BumpMap);
    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);
    TANGENT_SPACE_ROTATION;
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
    TRANSFER_SHADOW(o);
    return o;
}
```

在顶点着色器中，计算了三张纹理对应的纹理坐标，然后把光源方向换到了切线空间。

然后在片元着色器中模拟消融效果：

```
fixed4 frag(v2f i) : SV_Target {
    // 对噪声纹理采样，然后传递给 clip()函数
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;
    _BurnAmount = _Time.y*0.1;
    clip(burn.r - _BurnAmount);
    // 如果像素没有被剔除，那么进行正常的计算
    float3 tangentLightDir = normalize(i.lightDir);
    fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap, i.uvBumpMap));
    // 反射率
```

```
    fixed3 albedo = tex2D(_MainTex, i.uvMainTex).rgb;
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));
    // t 为 1 代表像素处于消融的边界，需要将颜色与烧焦的颜色混合
    fixed t = 1 - smoothstep(0.0, _LineWidth, burn.r - _BurnAmount);
    fixed3 burnColor = lerp(_BurnFirstColor, _BurnSecondColor, t);
    burnColor = pow(burnColor, 5);
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);
    fixed3 finalColor = lerp(ambient + diffuse * atten, burnColor, t * step(0.0001, _BurnAmount));
    return fixed4(finalColor, 1);
}
```

由于被剔除的代码不应该再有阴影，所以需要再定义一个 Pass{}代码段来处理阴影。

```
fixed4 frag(v2f i) : SV_Target {
    _BurnAmount = _Time.y * 0.1;
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;
    clip(burn.r - _BurnAmount);
    SHADOW_CASTER_FRAGMENT(i)
}
v2f vert(appdata_base v) {
    v2f o;
    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);
    return o;
}
```

其中使用 SHADOW_CASTER_FRAGMENT 与 TRANSFER_SHADOW_CASTER_NORMALOFFSET 宏来帮助计算。

图 9．消融效果

4 参考

1. realistic real time grass rendering with unity https://connect.unity.com/p/realistic-real-time-grass-rendering-with-unity

2. 《unity shader 入门精要》