# 第三次实验报告

姓名：陈 诺

学号：516030910199

目录

[第三次实验报告 1](#_Toc5132025)

[1目的与要求 3](#_Toc5132026)

[1.1 实验目的 3](#_Toc5132027)

[1.2 基本实验要求 3](#_Toc5132028)

[1.2.1编写Vertex Shader和Fragment Shader 3](#_Toc5132029)

[1.2.2编写Shader的GUI 3](#_Toc5132030)

[1.2.3 了解、尝试至少一个Debug工具 3](#_Toc5132031)

[1.2.4 撰写实验报告 4](#_Toc5132032)

[1.3 进阶作业要求 4](#_Toc5132033)

[1.3.1 实现一些更复杂的效果Shader效果 4](#_Toc5132034)

[1.3.2 在实验报告中描述实现原理、实现过程 4](#_Toc5132035)

[2. 实验步骤 4](#_Toc5132036)

[2.1 基础要求部分 4](#_Toc5132037)

[2.1.1 Vertex Shader和Fragment Shader 4](#_Toc5132038)

[2.1.2 Shader GUI 5](#_Toc5132039)

[2.1.3 了解Debug工具 8](#_Toc5132040)

[3. 进阶要求部分 9](#_Toc5132041)

[3.1 基于Geometry Shader实现草地效果 9](#_Toc5132042)

[3.1.1 实现原理 9](#_Toc5132043)

[3.1.2 实现过程 9](#_Toc5132044)

[3.2 使用噪声纹理实现动态消融效果 12](#_Toc5132045)

[3.2.1 实现原理 12](#_Toc5132046)

[3.2.2 实现过程 12](#_Toc5132047)

[4 参考 15](#_Toc5132048)

# 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。

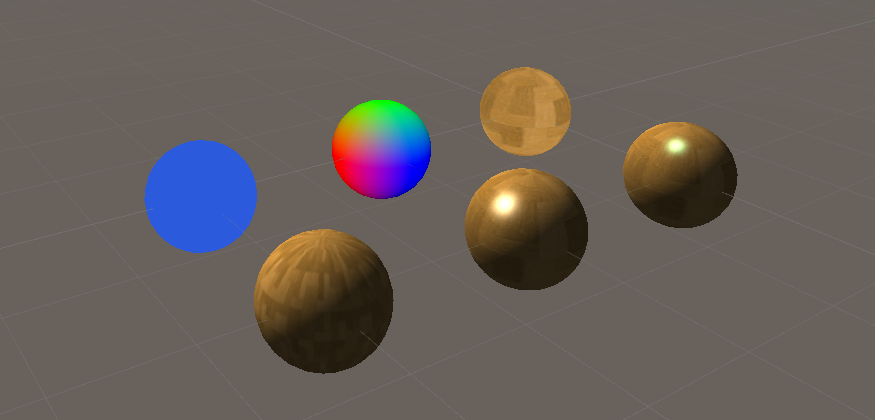


图1. 各shader

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

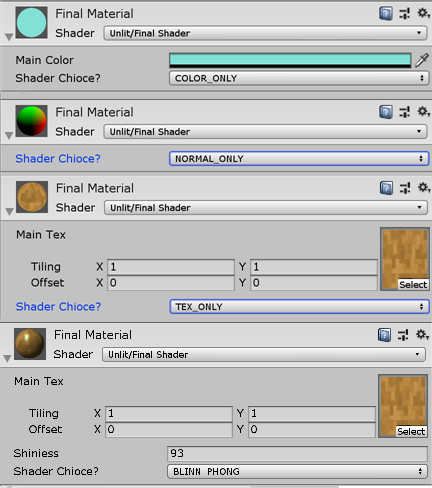


图2. 不同的shader choice及其面板

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. }
9. **public** **override** **void** OnGUI(){
10. ...
11. ShaderChioce shaderChioce = ShaderChioce.BLINN\_PHONG;
12. EditorGUI.BeginChangeCheck();
13. shaderChioce = (ShaderChioce)EditorGUILayout.EnumPopup(
14. **new** GUIContent("Shader Chioce?"), shaderChioce
15. );

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

1. **if** (EditorGUI.EndChangeCheck())
2. {
3. // Disable所有的Keyword
4. target.DisableKeyword("USE\_TEXTURE");
5. target.DisableKeyword("USE\_COLOR");
6. target.DisableKeyword("USE\_NORMAL");
7. target.DisableKeyword("USE\_BLINN");
8. // 检查shader choice并Enable相应的Keyword
9. // 另外三个为了篇幅省略
10. **if** (shaderChioce == ShaderChioce.COLOR\_ONLY)
11. {
12. target.EnableKeyword("USE\_COLOR");
13. }
14. }

根据Enable的Keyword来更新面板中的组件。

1. **if** (target.IsKeywordEnabled("USE\_COLOR"))
2. {
3. shaderChioce = ShaderChioce.COLOR\_ONLY;
4. MaterialProperty mainColor = FindProperty("\_MainColor", properties);
5. GUIContent mainColorLaber = **new** GUIContent(mainColor.displayName);
6. editor.ColorProperty(mainColor, mainColorLaber.text);
7. }
8. // 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:

1. #pragma shader\_feature USE\_COLOR
2. #pragma shader\_feature USE\_NORMAL
3. #pragma shader\_feature USE\_TEXTURE
4. #pragma shader\_feature USE\_BLINN

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

1. v2f MyVertProgram(a2v v) {
2. v2f i;
3. i.pos = UnityObjectToClipPos(v.vertex);
5. #if USE\_NORMAL
6. i.worldNormal = UnityObjectToWorldNormal(v.normal);
7. #endif
9. #if USE\_TEXTURE
10. i.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);
11. #endif
13. #if USE\_BLINN
14. i.worldNormal = UnityObjectToWorldNormal(v.normal);
15. i.worldPos = UnityObjectToClipPos(v.vertex).xyz;
16. i.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);
17. #endif
19. **return** i;
20. }

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。

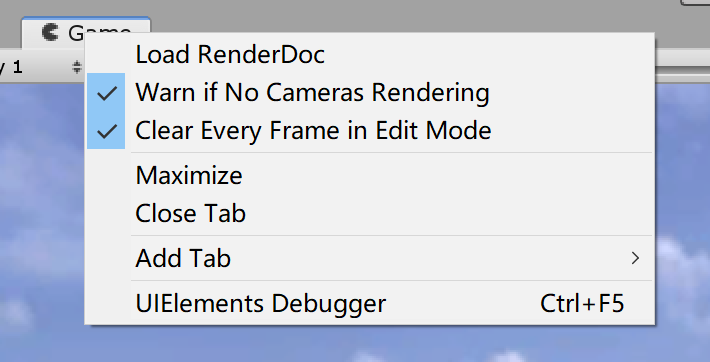


图3. 打开RenderDoc



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。

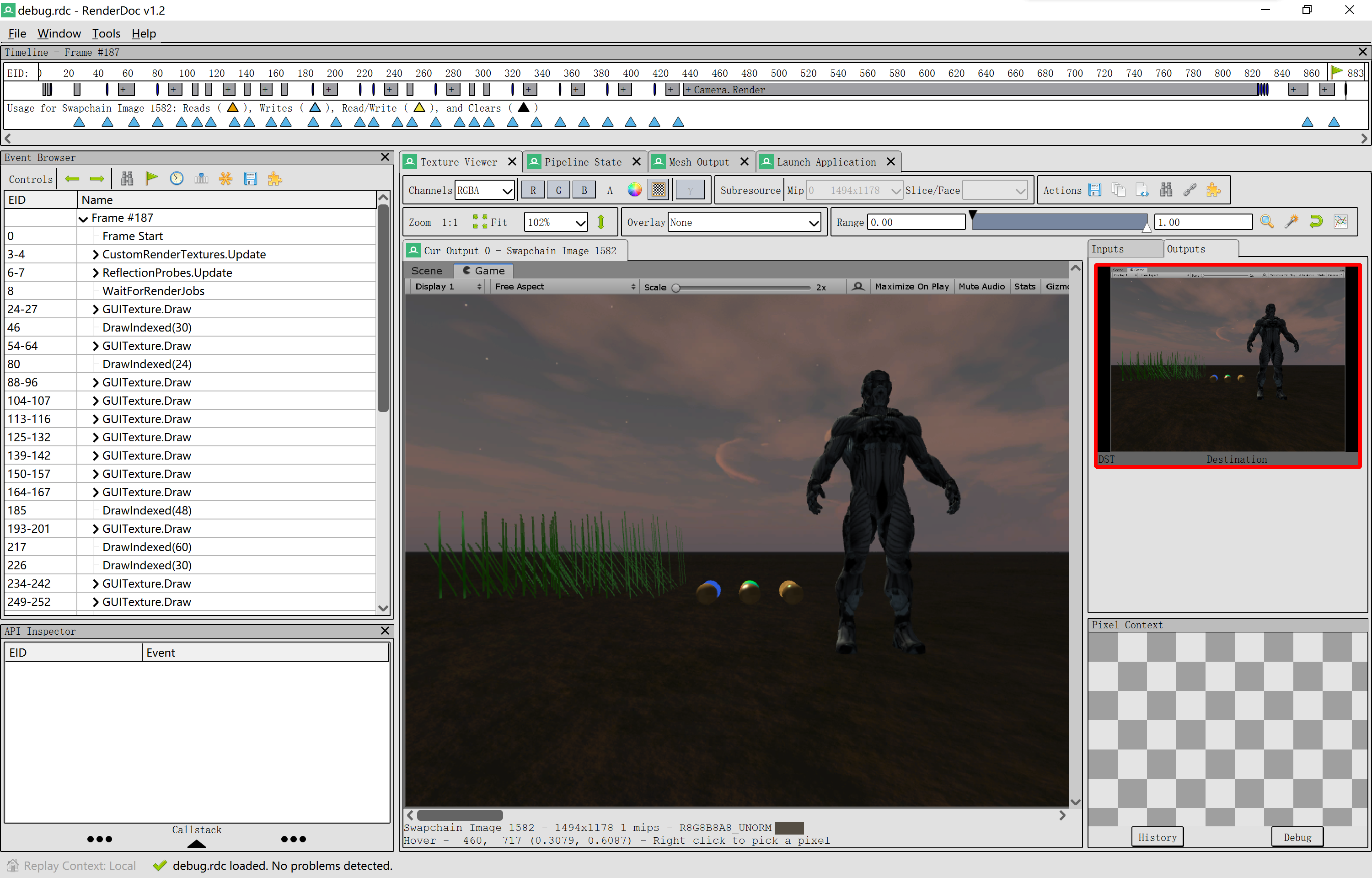


图5. RenderDoc截取的信息

# 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：

1. Properties{
2. // 草的纹理
3. \_MainTex("Albedo (RGB)", 2D) = "white" {}
4. // 用alpha纹理来决定草的形状
5. \_AlphaTex("Alpha (A)", 2D) = "white" {}
6. // 草的高度和宽度
7. \_Height("Grass Height", **float**) = 3
8. \_Width("Grass Width", range(0, 0.1)) = 0.05
9. }

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：

1. // 打开AlphaToMask来通过alpha to coverage呈现草的具体轮廓
2. AlphaToMask On
4. // 定义顶点着色器、片段着色器和几何着色器
5. #pragma vertex vert
6. #pragma fragment frag
7. #pragma geometry geom
9. // 着色器编译目标级别
10. #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：

1. // 设置顶点着色器向几何体着色器输出的最大顶点数量
2. [maxvertexcount(30)]
3. **void** geom(point v2g points[1], inout TriangleStream<g2f> triStream)
4. {
5. // 设置草的位置
6. float4 root = points[0].pos;
7. // 给草的属性设置随机性
8. **float** random = sin(UNITY\_HALF\_PI \* frac(root.x) + UNITY\_HALF\_PI \* frac(root.z));
9. \_Width = \_Width + (random / 50);
10. \_Height = \_Height + (random / 5);
12. // 生成一片叶子的网格
13. **for** (**int** i = 0; i < vertexCount; i++)
14. {
15. // 假设顶点的法线为(0, 0, 1)
16. v[i].norm = float3(0, 0, 1);
17. ...
18. // 生成一个随着\_Time变化的风，来促使草摇摆
19. float2 wind = float2(sin(\_Time.x \* UNITY\_PI \* 5), sin(\_Time.x \* UNITY\_PI \* 5));
20. // 使得草的位置随着风而变化
21. …
22. }
24. **for** (**int** p = 0; p < (vertexCount - 2); p++) {
25. // 向三角形流中append生成的三角形
26. }
27. }

这时，开始时设置的Plane已经有如下的效果了：

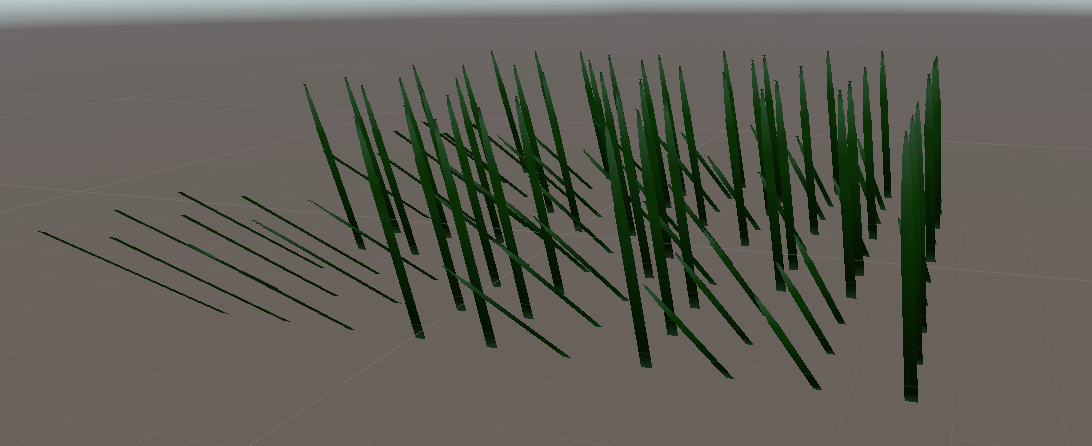


图7. 初始的草地效果

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。

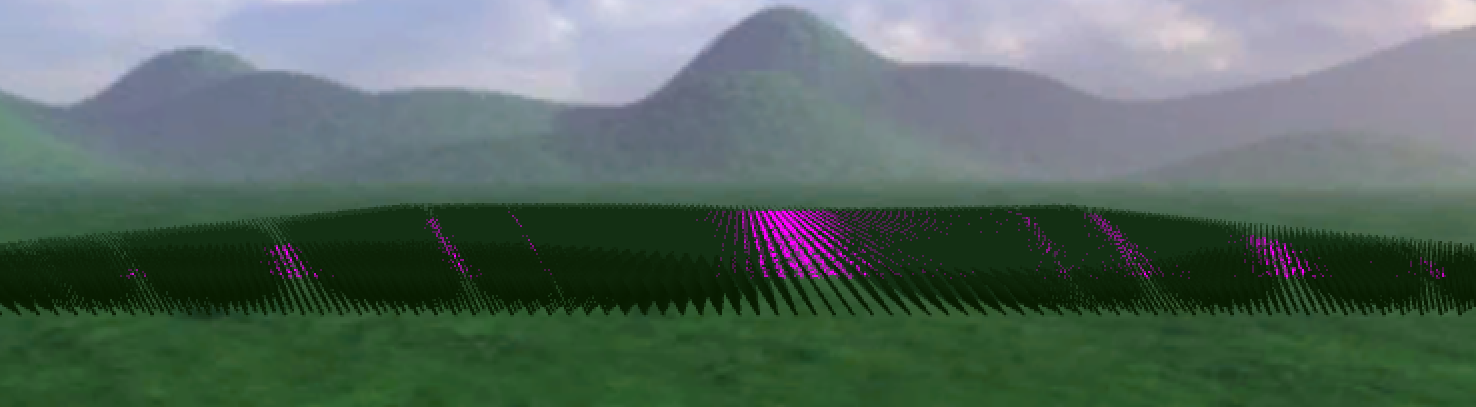


图8. 较为浓密的草地

## 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:

1. Pass {
2. Tags { "LightMode" = "ShadowCaster" }
3. CGPROGRAM
4. Cull Off
5. #pragma vertex vert
6. #pragma fragment frag
7. #pragma multi\_compile\_shadowcaster
8. }

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：

1. v2f vert(a2v v) {
2. v2f o;
3. o.pos = UnityObjectToClipPos(v.vertex);
4. o.uvMainTex = TRANSFORM\_TEX(v.texcoord, \_MainTex);
5. o.uvBumpMap = TRANSFORM\_TEX(v.texcoord, \_BumpMap);
6. o.uvBurnMap = TRANSFORM\_TEX(v.texcoord, \_BurnMap);
7. TANGENT\_SPACE\_ROTATION;
8. o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
9. o.worldPos = mul(unity\_ObjectToWorld, v.vertex).xyz;
10. TRANSFER\_SHADOW(o);
11. **return** o;
12. }

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：

1. fixed4 frag(v2f i) : SV\_Target {
2. // 对噪声纹理采样，然后传递给clip()函数
3. fixed3 burn = tex2D(\_BurnMap, i.uvBurnMap).rgb;
4. \_BurnAmount = \_Time.y\*0.1;
5. clip(burn.r - \_BurnAmount);
6. // 如果像素没有被剔除，那么进行正常的计算
7. float3 tangentLightDir = normalize(i.lightDir);
8. fixed3 tangentNormal = UnpackNormal(tex2D(\_BumpMap, i.uvBumpMap));
9. // 反射率
10. fixed3 albedo = tex2D(\_MainTex, i.uvMainTex).rgb;
11. fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.xyz \* albedo;
12. fixed3 diffuse = \_LightColor0.rgb \* albedo \* max(0, dot(tangentNormal, tangentLightDir));
13. // 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
14. **fixed** t = 1 - smoothstep(0.0, \_LineWidth, burn.r - \_BurnAmount);
15. fixed3 burnColor = lerp(\_BurnFirstColor, \_BurnSecondColor, t);
16. burnColor = pow(burnColor, 5);
17. UNITY\_LIGHT\_ATTENUATION(atten, i, i.worldPos);
18. fixed3 finalColor = lerp(ambient + diffuse \* atten, burnColor, t \* step(0.0001, \_BurnAmount));
19. **return** fixed4(finalColor, 1);
20. }

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。

1. fixed4 frag(v2f i) : SV\_Target {
2. \_BurnAmount = \_Time.y \* 0.1;
3. fixed3 burn = tex2D(\_BurnMap, i.uvBurnMap).rgb;
4. clip(burn.r - \_BurnAmount);
5. SHADOW\_CASTER\_FRAGMENT(i)
6. }
7. v2f vert(appdata\_base v) {
8. v2f o;
9. TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o)
10. o.uvBurnMap = TRANSFORM\_TEX(v.texcoord, \_BurnMap);
11. **return** o;
12. }

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》