Lecture5 开始 Unity Shader 学习之旅

1. 一个最近简单的顶点/片元着色器

顶点/片元着色器的基本结构

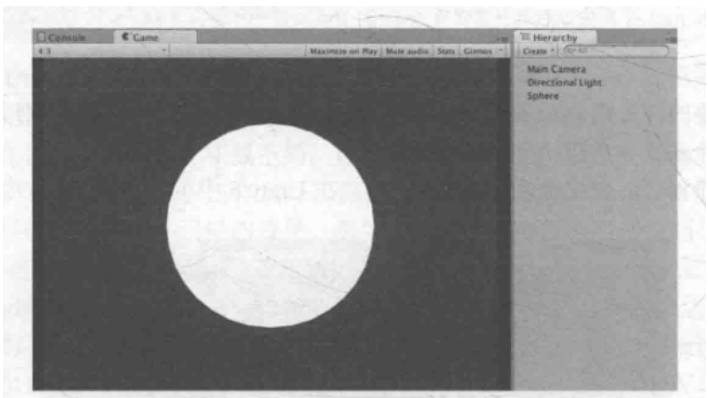
```
Shader "MyShaderName"{
    Properties{
        // 属性
    }
    SubShader{
        // 针对显卡A的SubShader
        Pass{
            // 设置渲染状态和标签
            // 开始CG代码片段
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            // CG 代码

            ENDCG
        }
        // 其它需要的Pass
    }
    SubShader{
        // 针对显卡B的SubShader
    }

    // 上述SubShader都失败后用于回调的Unity Shader
    Fallback "VertexLit"
}
```

SimpleShaderMat



▲图 5.2　用一个最简单的顶点/片元着色器得到一个白色的球

```
Shader "Custom/Assignment/Chapter5/1_SimpleShader"
{
    Properties {}
    SubShader
    {

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 v: POSITION): SV_POSITION
            {
                return UnityObjectToClipPos(v);
            }

            fixed4 frag():SV_Target
```

```
            {
                return fixed4(1.0, 1.0, 1.0, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Diffuse"
}
```

- 第一行 Shader 语义定义了 Unity Shader 的名字
- 我们并没有用到 Properties 语义块，它不是必须的，我们可以选择不声明材质的任何属性
- 声明了 SubShader 和 Pass 语义，在 SubShader 语义块中，我们定义了一个 Pass，这个 Pass 中我们没有进行任何自定义的渲染设置和标签设置
- 接着，就是由 CGPROGRAM 和 ENDCG 包裹的代码片段

```
#pragma vertex vert
#pragma fragment frag
```

- 它们告诉 Unity 哪个函数包含了**顶点着色器的代码**，哪个函数包含了**片元着色器的代码**，更通用的编译指令表示如下

```
#pragma vertex name
#pragma fragment name
```

- 其中 name 就是我们指定的**函数名**，这两个名字不一定是 vert 和 frag，它们可以是任意自定义的合法函数名，但我们**一般使用 vert 和 frag 来定义这两个函数**

顶点着色器 vert

```
float4 vert(float4 v: POSITION): SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, v);
}
```

- 顶点着色器代码，逐顶点执行
- vert 函数的输入 v 包含了这个顶点的位置，这是通过 POSITION 语义定义的，它的返回值是一个 **float4 类型**的变量，它是该顶点在**裁剪空间中的位置**
- POSITION 和 SV_POSITION 都是 CG/HLSL 中的**语义 semantics**，它们是不可省略的，这些语义告诉系统用户需要哪些输入值，以及用户的输出是声明
  - **POSITION**：告诉 Unity，把**模型空间的顶点坐标**填充到输入参数 v 中
  - **SV_POSITION**：告诉 Unity，顶点着色器的输出是**裁剪空间的顶点坐标**
- 本例中，顶点着色器只包含一行代码，就是把模型的顶点坐标从模型空间中转换到裁剪空间中

片元着色器 frag

```
fixed4 frag() : SV_Target
{
    return fixed4(1.0, 1.0, 1.0, 1.0);
}
```

- 它的没有输入，输出是一个 fixed4 类型的变量，并且使用了 SV_Target 语义进行限定，SV_Target 是一个 HLSL 中的系统语义，它告诉渲染器把用户的输出颜色存储到一个渲染目标 render target 中，这里将存储到默认的帧缓存中
- 片元着色器中的代码是返回一个白色的 fixed4 类型的变量

模型数据从哪里来

我们想要得到模型上每个顶点的纹理坐标和法线方向

- 纹理坐标：访问纹理
- 法线：计算光照

这时候需要为顶点着色器定义新的结构体

```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'
```

```
Shader "Custom/Assignment/Chapter5/2_SimpleShader2"
{
    Properties {}
    SubShader
    {

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            // 使用一个结构其来定义顶点着色器的输入
            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
            };

            // 使用一个结构体定义顶点着色器的输出
            struct v2f
            {
                // SV_POSITION语义告诉Unity, pos里包含了顶点在裁剪空间中的位置信息
                float4 pos : SV_POSITION;
                // COLOR0语义可以用于存储颜色信息
                fixed3 color : COLOR0;
            };

            v2f vert(a2v v)
            {
                // 声明输出结构
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                // v.normal 包含了顶点的法线方向，其分量在[-1.0, 1.0]
                // 下面的代码把分零六映射到[0.0, 1.0]
                // 存储到o.color中传递给片元着色器
                o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
                return o;
            }

            fixed4 frag(v2f i) :SV_Target
            {
                // 将插值后的 i.color 显示在屏幕上
                return fixed4(i.color, 1.0);
            }
            ENDCG
        }

    }
    FallBack "Diffuse"
}
```

结构体

为了创建一个自定义的结构体，必须按照如下结构定义

```
struct StructName{
    Type Name : Semantic;
    Type Name : Semantic;
    ...
};
```

- 语义不可省略

```
struct a2v
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};
```

- 声明了一个新的结构体 a2v，它包含了顶点着色器需要的模型数据，a2v 的定义中用到了更多 Unity 支持的语义，比如 NORMAL 和 TEXCOORD0
  - **a2v：application to vertex shader**，把数据从应用阶段传递到顶点着色器中
- 对于顶点着色器的输出，Unity 支持的语义有
  - POSITION
  - TANGENT
  - NORMAL
  - TEXCOORD0
  - TEXCOORD1
  - TEXCOORD2
  - TEXCOORD3
  - COLOR

```
struct v2f
{
    float4 pos : SV_POSITION;
    fixed3 color : COLOR0;
};
```

- 定义了一个新的结构体 v2f，它用于在顶点着色器和片元着色器之间传递信息
- 在顶点着色器的输出结构中，**必须包含一个变量，它的语义是 SV_POSITION**，否则渲染器无法得到裁剪空间的顶点坐标，也就无法把顶点渲染到屏幕上
- COLOR0 语义中的数据可以由用户自行定义，但一般是存储颜色，比如逐顶点的漫反射颜色或者逐顶点的高光反射

如何使用属性

Unity 材质提供给我们一个可以方便调节 Unity Shader 中参数的方式，通过参数可以随时调整材质的效果，而这些参数需要写在 Properties 语义块中

```
Shader "Custom/Assignment/Chapter5/3_ShaderProperties"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            // 在CG代码中，需要定义一个与属性名称和类型都匹配的变量
            fixed4 _Color;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
            };
```

```
        struct v2f
        {
            float4 pos : SV_POSITION;
            fixed3 color : COLOR0;
        };

        v2f vert(a2v v) : SV_POSITION
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);
            o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
            return o;
        }

        fixed4 frag(v2f i) : SV_Target
        {
            fixed3 c = i.color;
            c *= _Color.rgb; // 使用_Color属性控制输出颜色
            return fixed4(c, 1.0);
        }
        ENDCG
    }
  }
  FallBack "Diffuse"
}
```

- 在上面的代码中，我们首先添加了 Properties 语义块，并且声明了 _Color 属性，为了在 CG 代码中可以访问它，我们还需要在 CG 代码片段中提前一个新的变量，这个变量的名称和类型必须与 Properties 语义块中的属性定义相匹配

ShaderLab 属性与 CG 变量类型的匹配关系

ShaderLab	CG
Color, Vector	float4, half4, fixed4
Range, Float	float, half, fixed
2D	sampler2D
Cube	samplerCube
3D	sampler3D

2. Unity 提供的内置文件和变量

Unity 提供了很多内置文件，包括很多提前定义的函数、变量和宏等

- 如果在阅读别人写的 Unity Shader 代码时遇到很多未见过的变量、函数，又找不到对应的声明和定义，则这些代码很可能使用了 Unity 内置文件提供的函数和变量

内置包含文件 include file

包含文件类似 C++ 头文件，在 Unity 中，它们的文件后缀是 **.cginc**，在编写 Shader 时，我们可以使用 **#include 指令**把这些文件包含进来

例如：

```
CGPROGRAM
//...
#include "UnityCG.cginc"
//...
ENDCG
```

- Windows: 这些文件在 /Unity安装路径/Data/CGIncludes

CGIncludes 主要的包含文件

文件名	描述
UnityCG.cginc	包含最常用的帮助函数、宏和结构体等
UnityShaderVariables.cginc	在编译 Unity Shader 时，会被自动包含进来，包含了许多内置的全局变量，入 UNITY_MATRIX_MVP 等

文件名	描述
Lighting.cginc	包含了各种内置的光照模型，如果编写的是 Surface Shader 的话，会自动包含进来
HLSLSupport.cginc	在编译 Unity Shader 时，会被自动包含进来，声明了很多用于跨平台编译的宏和定义
UnityStandardBRDF.cginc	
UnityStandardCore.cginc	
AutoLight.cginc	

UnityCG.cginc

最长接触的包含文件，该文件提供很多常用结构体与函数

结构体

```
struct appdata_base {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_tan {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_full {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    float4 texcoord2 : TEXCOORD2;
    float4 texcoord3 : TEXCOORD3;
    fixed4 color : COLOR;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_img
{
    float4 vertex : POSITION;
    half2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f_vertex_lit {
    float2 uv   : TEXCOORD0;
    fixed4 diff : COLOR0;
    fixed4 spec : COLOR1;
};

struct v2f_img
{
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
    UNITY_VERTEX_OUTPUT_STEREO
};
```

名称	描述	变量
appdata_base	顶点着色器输入	顶点位置、顶点法线、第一组纹理坐标
appdata_tan	顶点着色器输入顶点着色器输入	顶点位置、顶点切线、顶点法线、第一组纹理坐标
appdata_full	顶点着色器输入	顶点位置、顶点切线、顶嗲法线、四组（或更多）纹理坐标
appdata_img	顶点着色器输入	顶点位置、第一组纹理坐标
v2f_img	顶点着色器输出	裁剪空间位置、纹理坐标

帮助函数

函数名	描述
float3 WorldSpaceViewDir(float4 v)	输入一个模**型空间中的顶点位置**，返回**世界空间中从该点到摄像机的观察方向**
float3 ObjSpaceViewDir(float4 v)	输入一个模**型空间中的顶点位置**，返回**模型空间中从该点到摄像机的观察方向**
float3 WorldSpaceLightDir(float4 v)	**仅用于前向渲染** 输入一个**模型空间中的顶点位置**，返回**世界空间中该点到光源的光照方向**，没有被归一化
float3 ObjSpaceLightDir(float4 v)	**仅用于前向渲染** 输入一个**模型空间中的顶点位置**，返回**模型空间中**从该点到光源的光照方向，没有被归一化
float3 UnityObjectToWorldNormal(float3 norm)	把**法线方向**从模型空间中转换到世界空间中
float3 UnityObjectToWorlir(in float3 dir)	把**方向向量**从模型空间变换到世界空间中
float3 UnityWorlToObjectir(float3 idr)	把**方向向量**从世界空间变换到模型空间中

3. Unity 提供的 CG/HLSL 语义

语义 semantics 是赋给 Shader 输入和输出的字符串，这个字符串表达了这个参数的含义。这些语义可以让 Shader 知道从哪里读取数据，并把数据输出到哪里，它是 CG/HLSL 的 Shader 流水线中不可或缺的

- Unity 并没有支持所有的语义，通常情况下，**这些输入输出变量并不需要有特别的意义**，比如 COLOR0 可以由我们自行决定，color 变量本身存储了什么，Shader 流水线并不关心
- 为了方便对模型数据传输，**有些语义有特别的含义规定**，例如**顶点着色器**输入结构体 a2f 用 **TEXCOORD0** 来描述 texcoord，将模型第一组纹理坐标填充到 texcoord 中，然而在顶点着色器的输出结构体 v2f 中。TEXCOORD0 修饰的变量的含义由我们自己决定
- 在 DirectX 10 之后，有一种**系统数值语义 system-value semantics**，这类语义由 **SV 开头**，有着特殊含义，例如，顶点着色器的输出中， SV_POSITION 包含了可用于光栅化的变化后的顶点坐标（齐次裁剪空间），这些语义描述的变量是不可以随意赋值的，流水线需要他们完成特定的目的

Unity 支持的语义

应用阶段传递给顶点着色器

语义	描述
POSITION	模型空间中的顶点位置，通常是 float4 类型
NORMAL	顶点法线，通常是 float3 类型
TANGENT	顶点切线，通常是 float4 类型
TEXCOORDn	该顶点的纹理坐标，TEXCOORD0 表示第一组纹理坐标，以此类推，通常是 float2 或者 float4 类型
COLOR	顶点颜色，通常是 fixed4 或者 float4 类型

- 通常情况下，一个模型的纹理坐标组数不超过 2，我们往往只使用 TEXCOORD0 和 TEXCOORD1

顶点着色器输出传递给片元着色器

语义	描述
SV_POSITION	裁剪空间中的顶点坐标，**结构体中必须包含一个用该语义修饰的变量**，等同于 DirectX 9 中的 POSITION，但最好使用 SV_POSITION
COLOR0	通常用于输出第一组顶点颜色，但不是必须的
COLOR1	通常用于输出第二组顶点颜色，但不是必须的
TEXCOORDN	通常用于输出纹理坐标，但不是必须的

- 除了 SV_POSITION 有要求以外，其它语义对变量的含义没有明确要求，可以存储任意值到这些语义描述变量中

片元着色器输出

语义	描述
SV_Target	输出值将会存储到渲染目标 render target 中，等于 DirectX 9 中的 COLOR 语义，但最好使用 SV_Target

注意

一个语义可以使用的寄存器只能处理 4 个浮点值 float，如果我们想要定义矩阵类型,如float3×4、float4×4 等变量就需要使用更多的空间

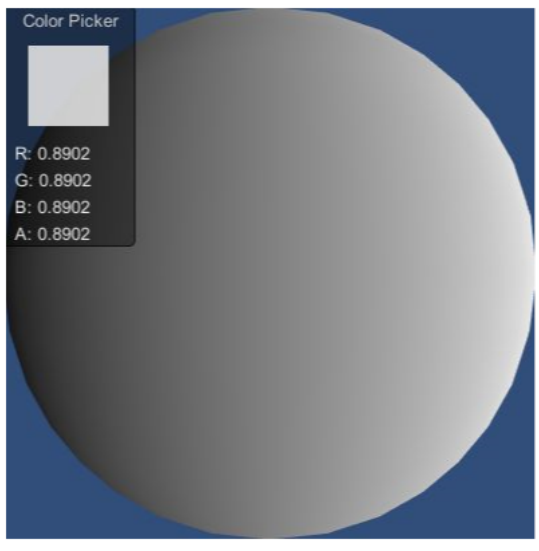
一种方法是,把这些变量拆分成多个变量，例如对于 float4×4 的矩阵类型,我们**可以拆分成 4 个 float4 类型的变量**，每个变量存储了矩阵中的一行数据

4. Unity Shader 的调试方法

假彩色图像 false-color image

把需要调试的变量映射到 [0, 1] 之间，把他们作为颜色输出到屏幕上，然后通过屏幕上显示的像素颜色来判断这个值是否正确

为了可以得到某点的颜色值，我们可以使用类似颜色拾取器的脚本的到屏幕上某点的 RGBA 值，从而推断出该点的调试信息



Visual Studio Graphics Debugger

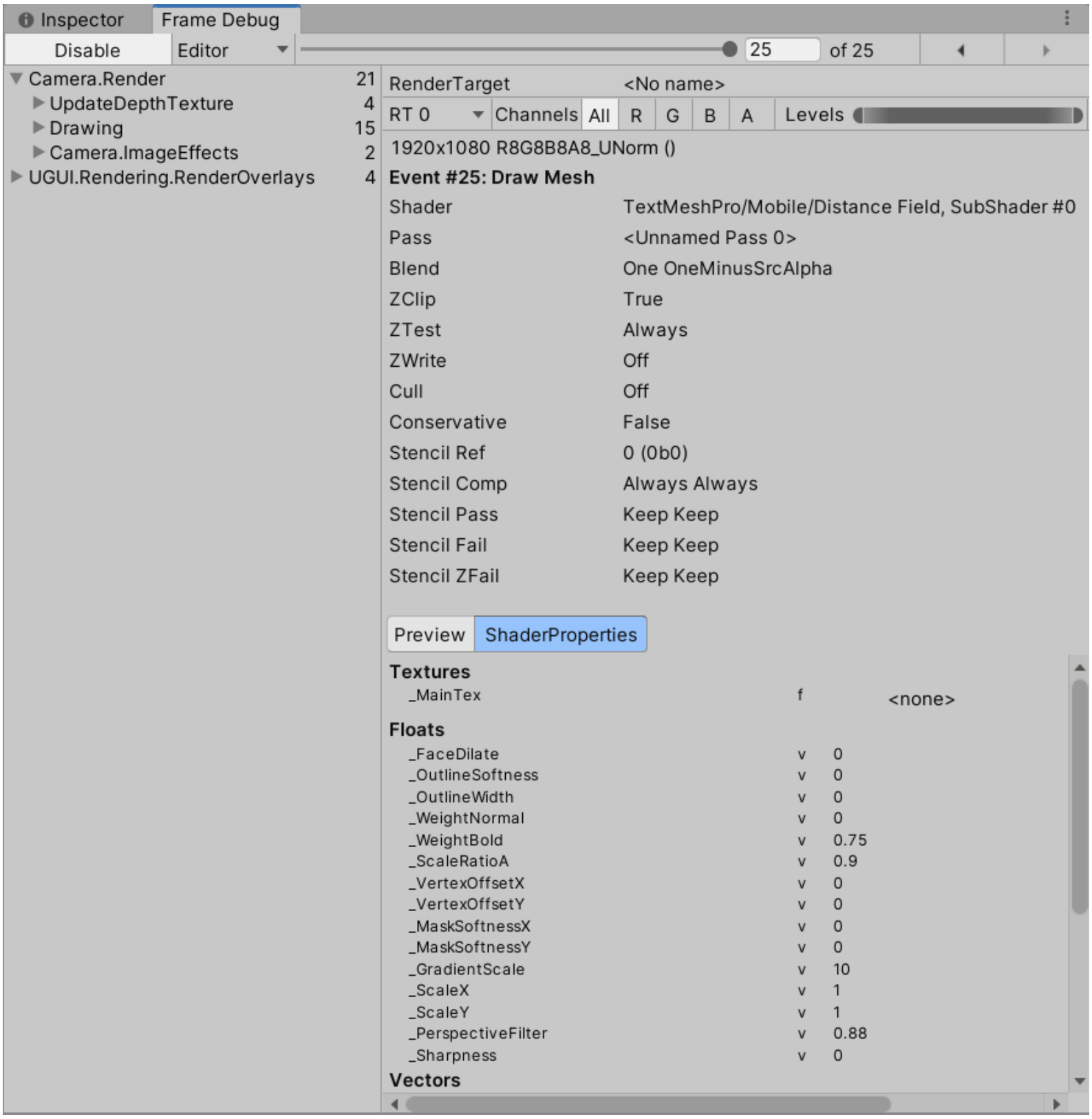
Visual Studio 提供了 Unity Shader 的调试功能 —— Graphics Debugger

可以查看每个像素的最终颜色、位置等信息，还可以对顶点着色器和片元着色器进行单步调试

- 具体参考 Unity 官方文档：使用 Visual Studio 对 DirectX 11 的 Shader 进行调试

帧调试器 frame debugger

在 Windows/Ayalysis/Frame Debugger 中可以打开帧调试器窗口



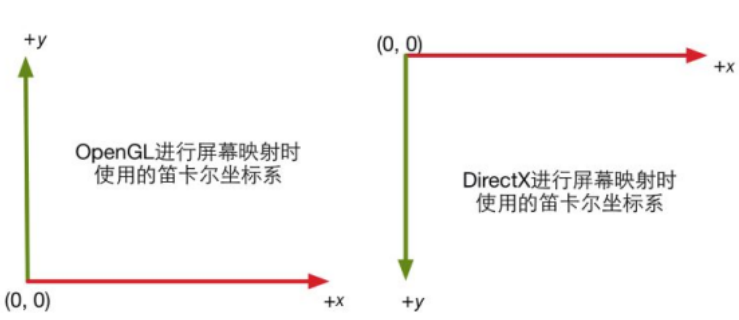
帧调试器用于查看渲染该帧时进行的各种**渲染事件 event**

5. 渲染平台的差异

可以去 Unity 官方文档上查找 SL-PlatformDifference 中寻找更多资料

渲染纹理的坐标差异

OpenGL 和 DirectX 的屏幕空间坐标有差异



在一种特殊的情况下，我们开启了抗锯齿 Anti Aliasing 时，Unity 不会处理 DirectX 的坐标差异，这时候图像不会被 Unity 翻转，我么需要在顶点着色器上自行翻转渲染纹理

```
#if UNITY_UV_STARTS_AT_TOP
if (_MainTex.TexselSize.y < 0)
    uv.y = 1-uv.y;
#endif
```

Shader 语法差异

> *incorrect number of arguments to numeric-type constructor (compiling for d3dii)*

DirectX 9/11 z对 Shader 语义更加严格

```
float4 v = float4(0.0);
```

- 在 OpenGL 上，上述代码合法，但是在 DirectX 11 上，我们必须写成

```
float4 v = float4(0.0, 0.0, 0.0, 0.0);
```

Shader 语义差异

- 用 SV_POSITION 描述顶点着色器输出的顶点位置
  - 一些 Shader 使用 POSITION 语义，但是这些 Shader 无法在索尼 PS4 平台上或者细分着色器的情况下正常工作
- 用 SV_Target 描述片元着色器的输出颜色
  - 一些 Shader 使用 COLOR/COLOR0 语义，这些 Shader 无法在索尼 PS4 平台上正常工作

6. Shader 整洁之道

float、half 还是 fixed

在 CG/HLSL 中，有 3 种精度的数值类型，这些精度决定**计算的取值范围**

类型	精度
float	最高精度的浮点值，通常使用 32 位存储
half	通常使用 16 位存储，精度范围 [-60000, 60000]
fixed	通常使用 11 位存储，精度范围 [-2.0, 2.0]

- 大多数现代桌面 GPU 会把所有计算按照最高浮点精度计算
- 在移动平台 GPU 上拥有不同的精度范围，不同精度运算速度有差异，应该确保在真正的移动平台上验证 Shader
- 一个基本的建议是，尽可能使用精度较低的类型，这样可以优化 Shader 的性能，这点在移动平台上尤其重要

规范语法

DirectX 平台对 Shader 的语义有着严格的要求，尽可能使用规范的语法

避免不必要的计算

在 Shader 中进行大量的计算可能会得到错误提示

> *temporary register limit of 8 exceeded*
> *Arithmetic instruction limit of 64 exceeded; 65 arithmetic instructions needed to compile program*

当我们在 Shader 中进行了过多的运算，使得需要的临时寄存器数目或指令数目超过当前支持数目，则可能报错

不同的 Shader Target，着色器阶段，可使用的临时寄存器和指令数目都是不同的

指令	描述
#pragma traget 2.0	默认等级，相当于 Direct3D 9 上的 Shader Model 2.0
#pragma traget 3.0	相当于 Direct3D 9 上的 Shader Model 3.0
#pragma traget 4.0	相当于 Direct3D 10 上的 Shader Model 4.0，只在 DirectX 11 和 XboxOne/PS4 平台上提供支持
#pragma traget 5.0	相当于 Direct3D 11 上的 Shader Model 5.0，只在 DirectX 11 和 XboxOne/PS4 平台上提供支持

- 所有类似 OpenGL 平台，包括移动平台，都被当成支持 Shader Model 3.0 的
- WP8/WinRT 平台只支持到 Shader Model 2.0

> Shader Model 是由微软提出的一套规范，通俗地理解就是它们决定了 Shader 中各个特性 feature 的能力 capability。这些特性和能力体现在Shader能使用的运算指令数目、寄存器个数等各个方面。Shader Model等级越高，Shader的能力就越大。

慎用分支和循环

尽管 GPU 也可以实现 if-else, for, while 这种流程控制指令，但是 GPU 使用不同于 CPU 的技术来实现分支语句，在 Shader 种不推荐使用流程控制，因为它们会降低 GPU 的并行处理数据

比较好的处理方式是，**尽可能把计算向流水线上端移动**，例如把片元着色器的计算放在顶点着色器，或者直接在 CPU 种进行预计算，再把结果传递给 Shader

如果不可避免需要分支语句

- 分支判断语句中使用的条件变量最好是常数，即在 Shader 运行过程中不会发生变化
- 每个分支中包含的操作指令数尽可能少
- 分支的嵌套层数尽可能少