Shader 笔记

郑华

2018 年 8 月 30 日

目录

第一章　基础概念

1.1　Draw Call

1.2　渲染状态

1.2.1　Texture

1.2.2　Material

1.2.3　Etc

1.3　批处理

1.4　Shader

可以干啥　屏幕特效实现下雪

静态纹理的动态化，如波浪、水流等

光照的应用

投影

常用的 Shader 语言

- OpenGL GLSL(opengGL Shader Language) SGI 公司开发的跨平台。
- DirectX HLSL(High Level Shader Language) 微软

- **->CG** 微软和英伟达联合开发的跨平台，基于 C 语言。
- **->Shader Lab** Unity3D 自己的 shader 语言。

矩阵变换 Unity 3D 矩阵一般是**左乘**。在Shader 中的前缀为UNITY_MATRIX_

1. M: 将物体坐标系变换到世界坐标系
2. V: 将世界坐标系变换到相机坐标系
3. P: 将 3D 坐标系转换成 2 维屏幕坐标系

当然在脚本中也可以完成类似功能，如

```
Transform.parent.localToWorldMatrix.MultiplyPoint(tranform.localPosition).
```

第二章　流水线

The Main Function of the pipeline is **to Generate** or **Render**, **a two-dimensional Image**, *given a virtual camera, three-dimensional objects, light sources, and more.*

2.1　渲染流水线

应用阶段　这个阶段主要由应用主导的，通常由 CPU 负责实现。即这个阶段开发者拥有绝对控制权。

1. **准备场景数据**
   - 模型、贴图
   - 相对位置 (世界变换)
   - 摄像机
   - 视椎体 (投影变换)
   - 光源等。
2. **粗粒度剔除工作**
   - 将不在场景中的物体剔除出去。
3. **设置好每个模型的渲染状态**
   - 使用的材质

- 使用的纹理
- 使用的 Shader

几何阶段　这个阶段决定需要绘制的图元是什么，怎么绘制他们，在哪里绘制他们，这一阶段一般在 GPU 上。

1. **顶点着色器**
   - 计算顶点的颜色
   - 将物体坐标系转换到相机坐标系 (相机变换)
   -
2. 裁剪
3. 屏幕映射

光栅化阶段　将**顶点**转换为 **像素**。

1. 三角形设置
2. 三角形遍历
   - 输入：顶点
   - 插值：将适当地方填充
   - 输出：像素
3. **片段着色器**
   - 输入：像素 (RGBA 4 通道组成)
   - **纹理采样**: 从纹理像素赋值给像素，**覆盖**以前默认像素（三角形遍历阶段插值的像素颜色）颜色值。
   - 像素跟**灯光计算**。
4. **三大测试**
   - alpha 测试：挑选合格的 **alpha 像素**显示。即透明度达到某个阈值予以显示。
   - 模板测试：达到合格 RGBA 像素，像素还可以携带**模板信息**，达到条件的模板值予以显示。

- 深度测试：测试与相机间的距离。

5. Blend 混合：将当前**要渲染的像素**和 **已经渲染出来的像素**进行混合。

6. GPU Buffer ：RGBA、模板值、深度值等

7. Front Buffer

8. Frame Buffer

9. 显示器

Notice 当场景中有一个 4 个顶点的组成的矩形，并假设其映射在屏幕的区域为 100*100，那么在 GPU 计算的过程中:

- 在几何阶段的顶点着色器阶段运行 4 次.
- 在光栅化阶段的片段着色器阶段运行 10000 次.

所以在顶点着色器和片段着色器对的运算级别不是一个级别。故而能把运算放到顶点着色器阶段的时候尽量不要放到片段着色器中。ss

2.2 OpenGL 渲染流程

第三章　CPU 与 GPU

第四章　Unity -Shader 着色器

4.1　基础概念

CPU　将 FBX、OBJ 等文件信息 (UV，顶点位置，法线，切线等) 读入`MeshRender`.

MeshRender　将这些信息传递到 GPU。主要有两种

- Skin Mesh Render : 带蒙皮的骨骼。
- Mesh Render + Mesh Filter：不带蒙皮的模型、没骨骼的，如 cube 等基础场景信息。其中 meshRender 将数据传递到 GPU，而 filter 做的是决定将哪个模型传递给 GPU。

哪些我们可以操作

- 顶点着色器
- 片段着色器
- 三大测试

Unity Shader 分类

- Fixed Shader: Shader 1.0，开关式
- 顶点、片段着色器：Shader 2.0，功能里面的**公式**可以自己定义。
- Surface Shader: Fixed + 顶点、片段着色器

Shader 结构　Unity Shader Lab 的语法结构。

```
Shader "Custom/My_XX_Shader"
{
    // 定义面板显示的属性
    Properities
    {

    }

    // 定义某种条件为显卡，如果合适就进行运行。
    // 可以定于多个，但是最少要有一个
    SubShader
    {
        // subShader 的标签
        [Tags]

        // 给多个 pass 公用的设置，如是否 开启测试等。
        [Common State]

        // 可能存在多个pass，每个pass 都会引起一次渲染过程。
        Pass
        {
            // Pass 的标签
            [Pass Tags]

            // 渲染设置，如颜色混合
            [Render Setup]

            // 纹理设置，只有在 Fixed Function Shader 中才可以使用。
            [Texture Setup]
        }
    }

    // 当所有的 subShader 失败的时候，使用该项指定的shader.
    [Fallback]


}
```

材质球与 Shader 的关系

材质球　比喻为人的衣服

Shader　材质（衣服）跟灯光的作用，Shader **决定**这个衣服怎么去作用。

Properities 使用说明　定义使用 3 步曲

1. 定义每个属性的名称
2. 定义每个属性对应的变量名称与类型
3. 定义变量的默认值

```
name ("Display Namee", Int) = number
```

```
Properities
{
    _TestInt ("TestInt", Int) = 1
    _TestFloat ("TestFloat", Float) = 1.0
    _TestRange ("TestRange", Range(1,5)) = 2
    _TestVector ("TestVector", Vector) = (1,1,1,1)
}
```

SubShader 使用说明

Tags　设置按照什么方式、什么时候渲染。

```
Tags {"Queue" = "Transparent"}
Tags {"RenderType" = "Opaquee"}
Tags {"DisableBatching" = "True"}
Tags {"ForceNoShadowCasting" = "True"}
Tags {"IgnoreProjector" = "True"}
Tags {"CanUseSpriteAtlas" = "True"}
Tags {"PreviewType" = "Plane"}
```

RenderSetup　设置显卡的各种状态

```
Cull Back //Front Off 设置剔除方式
ZTest Less Greater // GEqual 设置深度测试使用的函数
ZWrite On // Off 开启关闭深度写入
Blend SrcFactor DstFactor // 开启并设置混合模式
```

Pass

4.2 顶点着色器

计算顶点颜色 在 Pass 中有如下两种方式：

- 直接使用自定义颜色值：`Color (0,1,0,1)`
- 使用属性变量：`Color [_Var] //_Var ("TestColor", Color) = (1,1,1,1)`

顶点变换

灯光作用

$$\textbf{Ambient} * AmbientIntensity + (LightColor * \textbf{Diffuse} + LightColor * \textbf{Specular}) + \textbf{Emission}$$

Ambient 环境光

Diffuse 漫反射

Specular 镜面反射

Emission 自发光

渲染脚本灯光开关

- 所有灯光的开关 `Lighting On`
- 高光开关 `SeperateSpecular On`

```
Pass{
    Color[_TestCorlor]

    Material
    {
        Ambient[_TestColor]
        Diffuse[_TestColor] // Or Diffuse(1,1,1,1)
        Specular[_TestColor]
        Emission[_TestColor]
    }
```

```
    Lighting On
    SeparateSpecular On
}
```

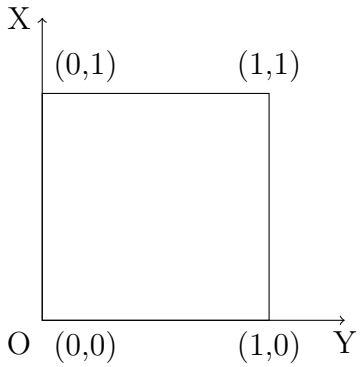
4.3 片段着色器-1.0 各显卡均可

纹理采样

纹理等于显示区域时 1 对 1 映射

等比例映射：UV 坐标

.



纹理大于显示区域时 这样的话，一个屏幕像素需要显示纹理的多个像素，此时就存在多种选择方式，常用的方式有

- Point：就近采样,
- Bi-linear: 就近周围 4 个像素的平均
- Tri-linear: 就近周围 8 个像素的平均

纹理小于显示区域时 那么屏幕像素将出现多对一的情况，即多个屏幕像素格都将显示同一个纹理像素，这样将产生锯齿和马赛克现象。

所以在实际的应用中，会在这种情况应用很多抗锯齿的算法。

设置纹理 格式：`SetTexture[VarTextureName]{block}`

在 block 中常常使用 Combine 进行混合，并且常常使用以下预定于的变量，需要注意以下几点

- Previous: 表示前一个 SetTexture 的 Texture 变量
- Primary: 表示顶点计算出来的颜色
- Texture: 等于 SetTexture 当前的 Texture
- Constant: 表示一个固定的颜色值

在使用 combine 利用中间图进行插值计算是,具体使用如下:函数格式:`combine src1 lerp(src2) src` 混合 src1 与 src3. 混合的方式取决于 Src2 的 alpha 值，并且 lerp 的具体公式如下 $(1-t)A+tB$

```
Pass{
    Color[_TestColor]
    SetTexture[_TestTexture]
    {
        // Texture Block
        combine Primary*Texture // 融合顶点颜色与纹理颜色
        combine Texture // 只使用纹理颜色，不使用顶点颜色
    }

    SetTexure[_TestTexture2]
    {
        combine Previous*Texture // 将之前调用的SetTexture 的纹理变量与当前Texture 混合
    }

    SetTexture[_TestTexture]
    {
        combine Texture lerp(Previous) Previous // 利用之前的纹理进行之前纹理与当前纹理的
            混合。
    }

    SetTexture[_TestTexture]
    {
        constantColor(1,1,1,1)
        combine Texture + constant
    }
}
```

示例-轮廓

方法一：一大一小　写两个 Pass，先渲染大的，然后再渲染小的。

```
SubShader{
```

```
// 先渲染轮廓
Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"

    struct appdata
    {
        float4 vertex : POSITION;
        float2 uv: TEXCOORDO;
    };

    struct v2f
    {
        float2 uv: TEXCOORDO;
        float4 vertex: SV_POSITION;
    };

    // 定义的顶点着色器
    v2f vert(appdata v)
    {
        v2f o;
        v.vertex.xy *= 1.2f; // 将顶点坐标放大的x、y 放大
        o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
        o.uv = v.uv;
        return o;
    }

    // 定义的片元着色器
    fixed4 frag(v2f i):SV_Target
    {
        reuturn fixed4(1,0,0,1); //给每个片元都返回红色
    }
    ENDCG
}

// 再渲染原纹理
Pass
{
    ZTest Always // 深度测试总是通过
```

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

        #include "UnityCG.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float2 uv: TEXCOORDO;
        };

        struct v2f
        {
            float2 uv: TEXCOORDO;
            float4 vertex: SV_POSITION;
        };

        // 定义的顶点着色器
        v2f vert(appdata v)
        {
            v2f o;
            o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
            o.uv = v.uv;
            return o;
        }

        //引用变量需要在这块重新声明
        sampler2D _MainTex;
        // 定义的片元着色器
        fixed4 frag(v2f i):SV_Target
        {
            fixed4 col = tex2D(_MainTex, i.uv);
            return col;
        }
        ENDCG
    }

}
```

方法二：梯度求边缘

- 找到边缘

- 给边缘着色
- 非边缘地带正常纹理采样

示例-屏幕后期特效 Render 运行完成后，会生成一张已经渲染完成的**图片**，此后会进入 RenderImage 步骤，完成屏幕后期渲染。

具体的实践过程如下：

1. 创建脚本，重写`OnRenderImage()`方法
2. 在脚本中创建一个新的材质 (Material), 并将新的 shader 添加到该材质球中，在此处即是下一步 Blit 用到的 Shader.
3. 在该方法中用 Unity 提供的接口完成对图片的渲染，`Graphics.Blit()`
4. 最后将该脚本挂到主相机上。

一般实现方法如下 :

1- 重写 OnRenderImage:

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture desTexture)
{
    Graphics.Blit(sourceTexture, desTexture, newShader);
}
```

2- 添加新的片元 Shader, 上述的 newShader

```
float _Offset;
fixed4 frag(v2f i):SV_Target
{
    float2 tmpUV = i.uv;
    fixed4 col1 = tex2D(_MainTex, tmpUV);
    fixed4 col2 = tex2D(_MainTex, tmpUV+float2(0, -Offset));
    fixed4 col3 = tex2D(_MainTex, tmpUV+float2(0, Offset));
    fixed4 col4 = tex2D(_MainTex, tmpUV+float2(-Offset,0));
    fixed4 col5 = tex2D(_MainTex, tmpUV+float2(Offset,0));

    return (col1 + col2 + col3 + col4 + col5)/6.0;
}
```

4.4 片段着色器-2.0

第五章　光照

如何在 shader 中实现光照模型，如漫反射、高光反射等。

$$\textbf{Ambient} + (LightColor * \textbf{Diffuse} + LightColor * \textbf{Specular}) + \textbf{Emission}$$

第六章　纹理

可以理解为贴图，但是纹理远不止皮肤，还有皮肤的沟壑信息等，也是需要渲染的，以模拟的更加真实。

第七章　深度、法线纹理

第八章　透明效果

如何实现透明度测试，如何透明度混合。

第九章　动态效果

使用时间变量来实现纹理动画。

第十章　屏幕后处理

高斯模糊、边缘处理。

第十一章　噪声处理

第十二章　渲染优化

第十三章　基于物理的渲染

第十四章　数学理论