紧接着上一篇文章的shader入门知识的总结，本文主要总结shader中的纹理贴图、透明度混合、顶点动画、后期特效处理等操作。如果有什么地方有错，请指出更正，谢谢。本文的代码主要来自开源书：unity入门精要

**一、Unity shader中的纹理**

**1、简单纹理**

      在unity shader中，纹理的主要作用是用来给模型贴上一个外表，这样得到的模型颜色就具有纹理的颜色混合。在常见的一些shader上，都会有一个\_MainTex的选项，这就是我们常常用的主纹理贴图。对于纹理贴图，其对应的需要有纹理坐标。在应用阶段，unity就会将模型的纹理坐标进行整理，存储在语义为TEXCOORD的语义中，通常我们会在shader中定义应用阶段到顶点计算阶段的结构体：

struct a2v{

vertex:POSITION;

texcoord:TEXCOORD;

}

　　其中的texcoord就是纹理坐标，注意这是模型上顶点的原始纹理坐标，还需要在顶点着色器上进行坐标转换操作：

TRANSFORM\_TEX(i.texcoord,\_MainTex)

= i.texcoord.xy \* \_MainTex.Scale + \_MainTex.Offset

　　如果模型的贴图没有缩放和偏移，则可以直接采用原生的顶点纹理坐标。

　　基于顶点的纹理坐标，我们可以在模型的纹理贴图中去采用该顶点坐标对应的纹理来作为帧素，从而作为漫反射光照计算的一部分，所以在片元shader中，我们通常会进行这样一步操作：

float4 c = tex2D(\_MainTex,i.texcoord)

　　通过这步操作，我们可以获取到纹理的rgba四个通道的值，基于得到的值，我们可以用来作为漫反射的颜色计算一部分。

**2、渐变纹理**

　　如果我们希望在模型上实现一种光照的渐变的效果，渐变纹理是一个不错的选择，通过对渐变纹理采样，其得到的采用结果具有渐变的效应，从而得到的漫反射颜色也带有渐变的效果。在对渐变纹理的采用过程中，通常不是基于纹理坐标来进行的采用操作，而是基于当前顶点的Lambert或者半Lambert光照的结果作为采样的坐标对渐变纹理进行采样：

float halfLam = 0.5\*lightResult + 0.5;

float3 c = tex2D(\_RampTex,float2(halfLam,halfLam));

**3、遮罩纹理**

　　前面也提起过，对于高光反射，其计算的结果较为高亮，在某些应用场景中，我们希望对高光的结果进行一定的平滑处理，最常见的操作对高光的结果进行一个因子的相乘。但是如果我们对不同的部位有不同的因子处理，这时候，我们就可以采用遮罩纹理。将影响因子合并在遮罩纹理的某个通道(RGBA)中，通过对遮罩纹理的采样，取出对应的通道的值作为游戏因子，实现遮罩的效果。关键几步代码为：

fixed3 halfDir = normalize(lightDir + viewDir);

fixed specularMask = tex2D(\_SpecularMask,i.uv).r \* \_SpecularScale;

fixed3 specular = \_LightColor0.rgb \* \_SpecularColor.rgb \* pow(max(0,dot(normal,halfDir)),Gloss) \* specularMask;

**4、法线纹理**

　　如果想要实现对模型的细节，常用的是法线纹理，用来在切线空间或者世界空间中实现对光照结果的调节。如果想在世界空间中对法线纹理进行操作，则关键的一点是计算出切线空间到世界空间的转换矩阵。通常这个转换矩阵是切线、副切线、法线按列组成的矩阵，最关键的几步shader为：

float3 worldPos = mul(\_Object2World,v.vertex).xyz;

fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);

fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);

fixed3 worldBinormal = cross(worldNormal,worldTangent)\*v.tangent.w;

o.TtoW0=float4(worldTangent.x,worldBinomal.x,worldNormal.x,worldPos.x);

o.TtoW1=float4(worldTangent.y,worldBinomal.y,worldNormal.y,worldPos.y);

o.TtoW2=float4(worldTangent.z,worldBinomal.z,worldNormal.z,worldPos.z);

　　对于得到的转换空间矩阵，在片元着色器中，其主要用来实现对切线空间的法线纹理采样结果的转换，关键的几步shader为：

fixed3 normal = UnpackNormal(tex2D(\_NormalMap,i.uv));

normal = nomalize(half3(dot(i.TtoW0.xyz,normal),dot(i.TtoW1.xyz,normal)),dot(i.TtoW2.xyz,normal)));

　　UnpackNormal()函数是对法线纹理的采样结果的一个反映射操作，其对应的法线纹理需要设置为Normal map的格式，才能使用该函数。关于法线纹理的实现细节，可以参看这儿：法线纹理的实现细节

**二、unity中的高级纹理**

　　除了常用的一些纹理，在某些时候，unity中需要使用一些高级纹理来实现更有效的纹理结果。主要可以归纳为立体纹理(CubeMap)、渲染纹理(MRT)以及程序纹理。程序纹理主要通过程序实现一些纹理的类似功能，应用场合比较特殊，这儿就不做过多的扩展，具体应用的时候可以查阅相关资料。

**1、立方体纹理**

　　立方体纹理是一种应用较多的环境映射纹理，通过将当前物体所在位置的前后左右上下共6个方向的环境映射为一个空间纹理贴图，通过反射、折射、菲涅尔反射等等方式来获取观察位置获取到的采样纹理，关键的一个函数是texCUBE(\_CubeMap,dir)。

　　立方体纹理有一个不大方便的是其只能反射凸面体，不能映射凹面体，也就是不能自身映射自身，所以在使用凹面体时候，需要注意这些细节。常见的反射和折射可以归纳为：

反射

o.worldRefl =reflect(-o.worldViewDir,o.worldNormal)

--

fixed3 reflection = texCBUE(\_Cubemap,i.worldRefl).rgb

折射

o.worldRefr = refract(-o.worldViewDir,o.worldNormal,\_RefractRatio);

--

fixed3 refraction = texCUBE(\_Cubemap,i.worldRefr).rgb;

--菲涅尔反射

o.worldRefl =reflect(-o.worldViewDir,o.worldNormal)

--

fixed3 reflection = texCBUE(\_Cubemap,i.worldRefl).rgb

fixed fresnel = \_FresnelScale + (1 - \_FresnelScale)\*pow(1-dot(worldViewDir,worldNormal),5);

fixed3 color = lerp(diffuse,reflection,saturate(fresnel));

**2、渲染纹理和GrabPass**

　　随着GPU技术的发展，渲染技术也在同步提升，现代的GPU允许将渲染结果存储在多个渲染目标中，也就是多级渲染对象MRT(Multiply render target)。想要实现MRT技术，则渲染纹理是一种较为常用的方法，通过将渲染结果存储在渲染纹理中，而不是直接存储在缓存中，可以实现对渲染结果的进一步操作。

　　渲染纹理的操作较为简单，通过在渲染相机中设置渲染纹理的对象，这样渲染的结果就会存储在制定的渲染纹理中。除了渲染纹理外，unity shader中还有一种获取当前渲染结果的指令:GrabPass 。使用GrabPass可以获取到当前的渲染结果，通过在后续的pass中使用这张渲染结果，我们可以获取进一步的shader表现效果。一般常用的是设置获取到的贴图的名字的方式：GrabPass{"TextureName"}，这样，在后期的渲染pass中可以直接采用这个TextureName对应的渲染纹理来进行操作。

**三、透明度操作**

　　透明度操作，主要是实现一些半透明的效果，当然也可以利用透明度操作中的剔除操作来剔除片元。透明度操作主要分为透明度剔除和透明度混合这两个基本的操作，透明度剔除，是设定一个判定的条件和阈值，在对片元进行操作的时候判定当前片元是否满足条件，如果满足则进行下一步的操作，如果不满足，则剔除该片元，可以用关键函数clip来实现。

　　在某些应用场景中，物体的透明度是需要根据时间来进行变化的，这时候可以在片元着色器中依赖时间来对透明度进行剔除操作，把与时间相关的操作用来实时的更新物体的透明度，这时候可以得到一种透明度渐变的效果。这可以应用在子弹的拖尾等场合中，这儿就不做深入的代码解释。

　　应用的较多的是透明度混合，这对于shader的渲染顺序有极高的要求。在unity shader中，基本的渲染顺序Queue是：

**BlackGround—>Geometry—>AlphaTest—>Transparent—>Overlay**，分别对应于背景，不透明物体，透明度测试，透明度混合，最终渲染对象的渲染顺序。

　　即使在同一个透明度混合的渲染顺序中，也需要注意两个半透明物体距离摄像机的渲染距离设置，否则会出现一些交叉重叠渲染的情况。这时候，其实是可以通过两个pass来实现规避两个半透明物体重叠交叉渲染的情况，在第一个pass中只需要进行一个深度测试，关键代码是 ColorMask 0，这样在第二个pass中进行的透明度混合就不会出现半透明物体交叉的现象。这样做会带来一些更高的性能损耗，在对性能要求较高的场合还是要慎用。

　　基本的透明度操作：Blend oper1 oper2，通过设置不同的oper1 和 oper2，我们可以实现预期的透明度操作，具体的oper可以有哪些参数，这个可以查阅对应的文档即可，最常用的透明度混合操作为：Blend SrcAlpha OneMinusAlpha 。

**四、顶点动画**

　　顶点动画，是一种用来实现模型动画的方式，在粒子特效等消耗较大的情况下，我们可以考虑采用顶点动画的方式来实现一些效果。顶点动画主要分为序列帧动画和基本的顶点动画。

　　序列帧动画，是将一个基本的动画分隔为多个UV动画纹理，通过时间来控制纹理的采样，可以获得一种连贯的纹理动画的效果。在一些基本的爆炸效果中，我们就可以采用序列帧动画来实现基本的爆炸动画。在序列帧动画中，最关键的几步shader操作是：

float time = floor(\_Time.y \* \_Speed);

float row = floor ( time/\_HorizontalAmount);

float column = time - row \* \_HorizontalAmount;

half2 uv = i.uv + half2(column,-row);

uv.x /= \_HorizontalAmount;

uv.y /= \_VerticalAmount;

fixed4 c = tex2D(\_MainTex,uv);

　    其基本的原理，就是在片元shader计算当前顶点所在的行列对应的纹理贴图的坐标，然后进行纹理采样。

　　 顶点动画，则是一种在顶点shader中实现的动画，通过设置顶点的位置的偏移，实现一种基本的顶点动画。比如在参看书中实现的一种河流的顶点动画：

float4 offset;

offset.yzw = float3(0,0,0);

offset.x = sin(\_Frequency \* \_Time.y + v.vertex.x \*\_InvWavelength+v.vertex.y \*\_InvWavelength + v.vertex.z \*\_InvWavelength)

o.pos = mul(UNITY\_MATRIX\_MVP,v.vertex + offset) + float2(0.0,\_Time.y \* \_Speed);

　　通过在x方向上实现正弦函数，可以实现模仿河流的流动的顶点动画的效果。

**五、屏幕后期特效**

　　在一些游戏中，不只是基于模型上的材质和shader我们可以实现一些基本的渲染，在更多时候我们需要对整体的游戏场景进行一些特定的渲染。这个时候，如果逐个去改变所有模型的shader，工作量较大，同时带来的性能损耗也需要慎重考虑。这时候可以采用屏幕后期特效处理的方式，对整体的场景进行一些特定的渲染设置。

　　屏幕后期特效，则需要采用两个基本的步骤，分别用c#和shader来实现对应的操作。c#主要用来挂载在主相机上，实现对指定相机的操作，shader则实现对应的特效处理。在c#中，最关键的是实现OnRenderImage()函数，结合Graphics.Blit()函数来实现对shader的属性设置和改变。在对应的shader中，我们可以实现指定的一些屏幕特效，比如对比度、透明度的调节，运动模糊，高亮(Bloom)操作，高斯模糊，基于深度纹理和法线纹理实现的一些特效等等。具体的一些实现流程不做过多的代码解释，可以参本文开头的开源项目中的代码。