

SpaceTransform

关于“ComputeScreenPos”:

unity的原定义是:

```
inline float4 ComputeNonStereoScreenPos(float4 pos) {
    float4 o = pos * 0.5f;
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    o.zw = pos.zw;
    return o;
}

inline float4 ComputeScreenPos(float4 pos) {
    float4 o = ComputeNonStereoScreenPos(pos);
#ifdef UNITY_SINGLE_PASS_STEREO
    o.xy = TransformStereoScreenSpaceTex(o.xy, pos.w);
#endif
    return o;
}
#ifdef UNITY_SINGLE_PASS_STEREO
float2 TransformStereoScreenSpaceTex(float2 uv, float w)
{
    float4 scaleOffset = unity_StereoScaleOffset[unity_StereoEyeIndex];
    return uv.xy * scaleOffset.xy + scaleOffset.zw * w;
}
```

这里接收的参数应该是, 也就是

```
UnityObjectToClipPos(vertex);
```

根据unity官方的定义应该是:

Transforms a point from object space to the camera's clip space in homogeneous coordinates. This is the equivalent of `mul(UNITY_MATRIX_MVP, float4(pos, 1.0))`, and should be used in its place.

所以这一步转换到的应该是相机裁剪空间齐次坐标。

通常的流程在ObjectSpace转换到ClipSpace下之后, z是深度值, w是原来的-z, 而xyz都是裁剪空间下的点, 同时满足 $-w \leq xyz \leq w$ 。w是用于后面做齐次除法用的。

而在裁剪空间→屏幕空间要经历如下过程：

1. xyz分量分别除以w分量。这一步就是齐次除法，就是NDC的过程。
2. 然后对这些“缩放”后的x和y分量映射到屏幕的宽高。如 $Screen_x = HomogeneousDivision_x * pixelWidth/2 + pixelWidth/2$ 。这里的HomogeneousDivisionX其实就是用 步骤1中x除以w得出来的结果。所以也有一些会写成 $Screen_x = clip_x * pixelWidth / (2 * clip_w) + pixelWidth/2$ 。为什么要除以2再加上宽高的一半参考 类似于把 [-1,1]映射到[0,1]的过程就知道了。这个pixelWidth是屏幕的宽。对于Y坐标同理。

回到Unity里面的ComputeScreenPos，它的计算方法略微和上面说的有一些不同。

```
inline float4 ComputeNonStereoScreenPos(float4 pos) {  
    float4 o = pos * 0.5f;  
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;  
    o.zw = pos.zw;  
    return o;  
}
```

转换成公式应该是：

$$Screen_x = Clip_x/2 + Clip_w/2$$

$$Screen_y = Clip_y * ProjectionParams.x/2 + Clip_w/2$$

这个很明显和我们的是对不上的。那么问题出在哪里呢，第一个_ProjectionParams.x是用于不同平台下可能会出现屏幕的结果上下翻转的情况。而我们这里返回的坐标其实是0-1，也就是说并非是原来屏幕空间的像素数值。所以我们同时除以pixelWidth，就会得到一个类似于**Formula1**: $Screen_x = clip_x / (2 * clip_w) + 1/2$ 。

此时的结果和Unity计算出来的已经很接近了，就只差了一个 $Clip_w$ 。也就是说我们只要对Unity的结果同时除以 $Clip_w$ 我们就可以得到**Formula1**的结果了。

所以说Unity在这一刻计算出来的结果是并非经过齐次除法的。

这么做的原因其中之一是当我们把uv坐标发送到片段着色器时会经过插值，我们想要在插值之后应用除法而不是插值之前应用除法，这可以保证结果的准确性。同时也有一个叫tex2Dproj的指令宏可以用于齐次除法后的采样方法。

至此我们可以根据裁剪空间下的坐标来返回片段在屏幕空间下的坐标了。

关于使用深度值重建世界坐标：

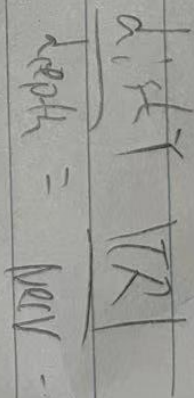
引用《Shader入门精要》里面的说法就是，我们可以通过一个顶点相对于另外一个顶点的偏移量来计算这个顶点的大小。那么做法就是我们确认摄像机在世界空间下的坐标，以及顶点和摄像机的相对位置，就可以换算出这个顶点在世界空间下的坐标了。注意顶点和摄像机的相对位置不是在世界空间下进行计算的，否则我们就不需要费老大劲做这事了。

表达为公式为：

$$WorldPos = WorldSpaceCameraPos + offset$$

所以我们就是要研究如何计算offset的问题。

如果我们把相机、顶点、裁剪金字塔画一张图的话：（这个图是草稿，手残实在是画不出来找机会再重新画一次。。凑合看吧。shader入门精要都有这些图画的很好的）



$$\frac{\text{depth}}{\text{width}} = \frac{\text{new}}{\text{new}}$$

根据三角形相似，我们可以得出任意一个点到摄像机的距离 $dist = |TL|/near * depth$ 。depth是我们从深度缓冲采样出来的结果，而TL（TR\BL\BR）都是指三角形近平面到相机的向量。而它们的模长就代表了欧氏距离，而near则是代表了近平面的“深度值”。

我们只需要令 $|TL|/near = Scale$ ，这个可以作为一个常值用在片段着色器中，只要我们采样到的深度值和这个相乘，就可以得出偏移量的多少，从而重建世界空间。

在Unity中，在LightingPass中四个四边形顶点的射线是由NORMAL提供的。这意味着我们不用自己去计算。

所以在DefaultRP中可以看到这样的写法：

```
float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);
depth = Linear01Depth (depth);
float4 vpos = float4(i.ray * depth,1);
float3 wpos = mul (unity_CameraToWorld, vpos).xyz;
```

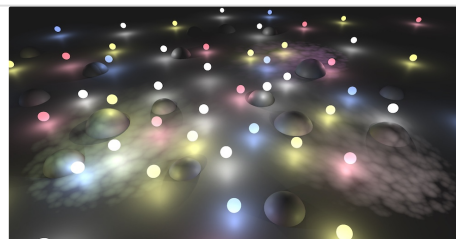
而这个i.ray则是上面的NORMAL，也就是由Unity提供给我们的四个角到相机的射线。而在片段着色器中这些射线都是经过插值的。

而

Rendering 15

Deferred Lights Use a custom light shader. Decode LDR colors. Add lighting in a separate pass. Support directional, spotlight, and point lights. Manually sample shadow maps. We added support for the

 <https://catlikecoding.com/unity/tutorials/rendering/part-15/>



这里提供了一个射线到远平面的计算方法。它的描述是这样的：

However, a big difference is that we supplied rays that reached the far plane to our fog shader. In this case, we are supplied with rays that reach the near plane. We have to scale them so we get rays that reach the far plane. This can be done by scaling the ray so its Z coordinate becomes 1, and multiplying it with the far plane distance.

它的代码是这样的

```
float3 rayToFarPlane = i.ray * _ProjectionParams.z / i.ray.z;
```

这个远平面射线的计算也很简单，`_ProjectionParams.z`是远平面也就是far

我们可以通过 $Ray_{near}/near = Ray_{far}/Far$

得出 $Ray_{far} = Ray_{near} * Far/near$

对比Unity的做法，我不太确定他到底为什么要找远平面。

而根据《SHADER入门精要》里面的描述也是使用近平面。（详见其276页到278页的描述）。

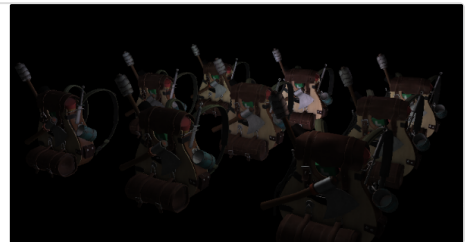
至此我们就获得通过深度值重构世界空间坐标的一个方法。

附：其实如果觉得这个方法太麻烦可以往GBUFFER填充世界坐标就一劳永逸了。比如说这里的做法：

Deferred Shading

The way we did lighting so far was called forward rendering or forward shading. A straightforward approach where we render an object and light it according to all light sources in a scene. We do


 <https://learnopengl.com/Advanced-Lighting/Deferred-Shading>



但是实际上重构世界空间是很常见的做法，毕竟位置信息要占用一个Vector3，而且还涉及到精度的问题。在Unity中的Gbuffer Layout就没有储存世界空间坐标。例如这些人说的：

Don't store position in gbuffer · Issue #1 · JoshuaSenouf/gl-engine

It's a common technique but if you pass the inverse world-view-projection matrix, you can sample your depth buffer and reconstruct position from it using the matrix. This will save significantly on your gbuffer size and

 <https://github.com/JoshuaSenouf/gl-engine/issues/1>

JoshuaSenouf/gl-engine

#1 Don't store position in gbuffer

1 comment



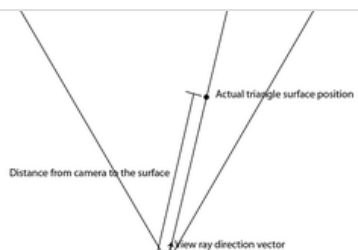
 graphitemaster opened on April 21, 2017

而这一篇文章对上面描述的原理也有一个很不错的解释：

Position From Depth 3: Back In The Habit

A friend of mine once told me that you could use "back in the habit" as the subtitle for any movie sequel. I think it works. So a lot of people still have trouble with reconstructing position from depth

 <https://mynameismjp.wordpress.com/2010/09/05/position-from-depth-3/>

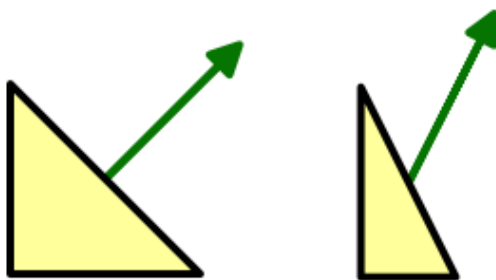


法线rescale的问题：

如果模型本身不是统一缩放的话，如果我们直接使用

```
i.normal = mul(unity_ObjectToWorld, float4(v.normal, 0));  
i.normal = normalize(i.normal);
```

法线的结果会不正确。



假设unity_ObjectToWorld这个矩阵含有的组合为缩放S、旋转R、位移P。

法线是方向所以位移是不会影响法线的结果的。以我们希望抵消掉对法线的缩放矩阵的影响。所以我们希望这个矩阵N能够抵消缩放的影响但是保留旋转的矩阵。即 $N = R^{-1}T$

也就是我们要对这个经过unity_ObjectToWorld矩阵变换的矩阵再应用一次这个N才能获取正确的结果。

所以关键是如何获得这个N。

缩放矩阵的逆等于这个

$$\begin{bmatrix} X & 0 & 0 \\ 0 & Y & 0 \\ 0 & 0 & Z \end{bmatrix}^{-1} = \begin{bmatrix} 1/X & 0 & 0 \\ 0 & 1/Y & 0 \\ 0 & 0 & 1/Z \end{bmatrix}$$

然后我们可以获取世界坐标到本地坐标的矩阵，也就是unity_ObjectToWorld的逆矩阵。即 $P^{-1}R^{-1}S^{-1}$ 。

由于我们的目标是保留旋转但是还原缩放，所以当我们直接对这个法线向量应用 unity_ObjectToWorld 的逆矩阵的话，我们也会把旋转还原回去。（即此时的目标要保留 unity_ObjectToWorld 的旋转不受 unity_ObjectToWorld 的逆矩阵 unity_WorldToObject 的变化）。

所以我们可以对 unity_WorldToObject 进行转置就可以保留这个旋转的变化。

为什么呢，因为

旋转矩阵 Z（围绕 Z 做旋转）的逆矩阵为：

$$\begin{bmatrix} \cos(z) & -\sin(z) & 0 \\ \sin(z) & \cos(z) & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \cos(z) & \sin(z) & 0 \\ -\sin(z) & \cos(z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

而旋转矩阵 Z 的转置为：

$$\begin{bmatrix} \cos(z) & -\sin(z) & 0 \\ \sin(z) & \cos(z) & 0 \\ 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} \cos(z) & \sin(z) & 0 \\ -\sin(z) & \cos(z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

可以看出来旋转矩阵的逆矩阵是等于其转置的。

所以当我们对 $P^{-1}R^{-1}S^{-1}$ 进行转置（这里 P 可以忽略），我们就会得到 $S^T R^T$ （两个矩阵相乘的转置等于调换两个矩阵的顺序后的转置的相乘）。而由于 $R^T = R^{-1}$ ，即 unity_WorldToObject 的旋转矩阵的逆矩阵就是等于 unity_ObjectToWorld 的旋转矩阵了。所以我们就保留了旋转矩阵的结果的同时抵消了缩放矩阵的影响。

这里有一个比较有意思的细节是，unity 的 URP 中对这个方法的函数是这么写的：

```
float4x4 GetWorldToObjectMatrix()
{
    return UNITY_MATRIX_I_M;
}
// Transforms normal from object to world space
float3 TransformObjectToWorldNormal(float3 normalOS, bool doNormalize = true)
{
#ifdef UNITY_ASSUME_UNIFORM_SCALING
    return TransformObjectToWorldDir(normalOS, doNormalize);
#else
    // Normal need to be multiply by inverse transpose
    float3 normalWS = mul(normalOS, (float3x3)GetWorldToObjectMatrix());
    if (doNormalize)
        return SafeNormalize(normalWS);

    return normalWS;
#endif
}
```


可以留意到这行

```
float3 normalWS = mul(normalOS, (float3x3)GetWorldToObjectMatrix())
```

Unity是对矩阵进行右乘而不是左乘。而这个UNITY_MATRIX_I_M就是我们的unity_WorldToObject。

那么为什么这里没有进行转置呢？因为 $A^T * a = a * A$ 。

所以我们对这个UNITY_MATRIX_I_M进行右乘，就等于对这个矩阵的转置左乘。（可以用123456789 和 123 来交换顺序计算一下，实在不想手算网上找个计算器完事了，我就是这么做的）

至此我们就得到了正确的法线向量：

