[2024.01.19]间章：数学部分的补充

选择3x3或者4x4矩阵

对于线性变换来说（例如旋转或缩放），3x3矩阵已经足够了，因为我们知道，是为了把平移、也能够和线性变换放在一个矩阵里表示，才把它转换到齐次坐标下成为4维矩阵。

而在4x4的矩阵中，通常我们对点的w分量是1，因为平移对点的位置是有影响的，而对向量的w分量是0，因为方向不受平移影响。

CG中的向量和矩阵

在CG中，矩阵类型是由float3x3或者float4x4等关键词进行声明和定义的，而对于float3、float4等类型的变量，我们既可以把它当成一个向量，也可以把它当成是一个1xn的行矩阵或nx1的列矩阵。但是，当我们进行点乘运算时，这两个float就会被当做向量类型：

```
float4 a = float4(1.0, 2.0, 3.0, 4.0);
float4 b = float4(1.0, 2.0, 3.0, 4.0);
// 对两个矢量进行点积操作
float result = dot(a, b);
```

但在进行矩阵乘法时，参数的位置将决定是按行矩阵还是按列矩阵进行乘法，我们通过**mul**函数实现乘法；

```
float4 v = float4(1.0, 2.0, 3.0, 4.0);
float4x4 M = float4x4(1.0, 0.0, 0.0, 0.0,
                      0.0, 1.0, 0.0, 0.0,
                      0.0, 0.0, 1.0, 0.0,
                      0.0, 0.0, 0.0, 1.0);
// 把 v 当成列矩阵和矩阵 M 进行右乘
float4 column_mul_result = mul(M, v);
// 把 v 当成行矩阵和矩阵 M 进行左乘
float4 row_mul_result = mul(v, M);
// 注意: column_mul_result 不等于 row_mul_result, 而是:
// mul(M,v) == mul(v, tranpose(M))
// mul(v,M) == mul(tranpose(M), v)
```

高版本的Unity已经将很多矩阵乘法封装进了函数中，后续可能与笔记中的公式不太一样。

参数的位置会直接影响结果值，通常在变换顶点时，我们都是用右乘的方式，按列向量进行乘法，Unity提供的很多内置矩阵，如UNITY_MATRIX_MVP等都是按列来存储的，但有时也可以直接使用**左乘，这样可以省去对矩阵转置的操作**。

此外，当填充矩阵时，CG是按照行优先的方式，即一行一行填充矩阵的，如果要访问矩阵的某个元素也是同样的：

```
// 按行优先的方式初始化矩阵 M
float3x3 M = float3x3(1.0, 2.0, 3.0,
                      4.0, 5.0, 6.0,
                      7.0, 8.0, 9.0);
// 得到 M 的第一行，即(1.0, 2.0, 3.0)
float3 row = M[0];

// 得到 M 的第 2 行第 1 列的元素，即 4.0
float ele = M[1][0];
```

同时，Unity中提供了一个Matrix4x4类型，它是按列优先的方式来生成矩阵的，这点要注意区分。

Unity中的屏幕坐标：ComputeScreenPos/VPOS/WPOS

在顶点/片元着色器中，有两种方式来获得片元的屏幕坐标：

1. 在片元着色器的输入中声明VPOS或WPOS语义。他们两个分别是HLSL、CG中对屏幕坐标的语义，且在Unity Shader中是等价的。

```
fixed4 frag(float4 sp : VPOS) : SV_Target {
    // 用屏幕坐标除以屏幕分辨率_ScreenParams.xy，得到视口空间中的坐标
    return fixed4(sp.xy/_ScreenParams.xy,0.0,1.0);
}
```

VPOS/WPOS语义定义的输入是一个float4类型的变量，它的xy值代表了在屏幕空间中的像素坐标，以一个400x300的屏幕分辨率为例，则x的范围是[0.5,400.5]，y的范围是[0.5,300.5]，因为OpenGL和DX10以后的版本认为像素中心对应的浮点值是0.5。

而zw分量，在Unity中，VPOS/WPOS的z分量范围是[0,1]，在近裁剪平面处z为0，在远裁剪平面处z为1。对于w分量，要考虑摄像机的投影类型，如果是透视投影，则w分量的范围是[1/Near, 1/Far]，如果是正交投影，则w分量恒为1。这些值是通过对经过投影矩阵变换后的w分量取倒数得到的。

最后，把屏幕空间除以屏幕分辨率来得到视口空间（viewpost space）中的坐标，视口空间就是把屏幕坐标归一化，这样左下角是(0,0)，右上角是(1,1)，如果已知屏幕坐标，则把xy值除以屏幕分辨率即可。

2. 另一种方式是通过ComputeScreenPos函数，这个函数在UnityCG.cginc里被定义。通常首先在顶点着色器中将ComputeScreenPos的结果保存在输出结构体中，然后再片元着色器中进行一个齐次除法运算后得到视口空间下的坐标。

```
struct vertOut {
    float4 pos : SV_POSITION;
    float4 scrPos : TEXCOORD0;
};

vertOut vert(appdata_base v) {
    vertOut o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    // 第一步：把 ComputeScreenPos 的结果保存到 scrPos 中
    o.scrPos = ComputeScreenPos(o.pos);
    return o;
}

fixed4 frag(vertOut i) : SV_Target {
    // 第二步：用 scrPos.xy 除以 scrPos.w 得到视口空间中的坐标
    float2 wcoord = (i.scrPos.xy/i.scrPos.w);
    return fixed4(wcoord,0.0,1.0);
}
```

上面的步骤实际上是手动实现了屏幕映射的过程，而且它得到的坐标直接就是视口空间中的坐标，根据之前从裁剪空间映射的点映射到屏幕坐标中，我们可以得到公式如下：

$$viewport_x = \frac{clip_x}{2 \cdot clip_w} + \frac{1}{2}$$

$$viewport_y = \frac{clip_y}{2 \cdot clip_w} + \frac{1}{2}$$

上面公式的思想是，首先对裁剪空间下的坐标进行齐次除法，得到范围在[-1, 1]的NDC，然后将其映射到范围在[0, 1]的视口空间下的坐标。看UnityCG.cginc中ComputeScreenPos函数的定义如下：

```
inline float4 ComputeScreenPos (float4 pos) {
    float4 o = pos * 0.5f;
    #if defined(UNITY_HALF_TEXEL_OFFSET)
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w * _ScreenParams.zw;
    #else
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    #endif

    o.zw = pos.zw;
    return o;
}
```

ComputeScreenPos的输入参数pos是经过MVP矩阵变换后在裁剪空间的顶点坐标。
UNITY_HALF_TEXTL_OFFSET是Unity在某些DX平台上使用的宏，可以先忽略。

_ProjectionParams.x默认情况下是1，如果使用了一个反转的投影矩阵的话就是-1，但这种情况不多见，那么上面的代码实际上是：

$$Output_x = \frac{clip_x}{2} + \frac{clip_w}{2}$$

$$Output_y = \frac{clip_y}{2} + \frac{clip_w}{2}$$

$$Output_z = clip_z$$

$$Output_w = clip_w$$

这里的xy不是真正的视口空间下的坐标，所以需要在片元着色器中再进行一步处理，除以裁剪坐标的w分量，结果就是我们视口空间的坐标公式，至此才完成整个映射的过程。

可以发现，虽然函数的名字叫ComputeScreenPos，但结果确不是一步到位的，这是因为如果Unity在函数中直接除以w分量，就会破坏插值的结果。

我们知道从顶点着色器到片元着色器中间其实有一个插值的过程，如果在顶点着色器就进行除法，则最终得到的x/w和y/w就是错误的，插值的结果会不准确。但反过来就是正确的，可以认为是齐次除法抵消了插值的影响。

究其原因，我们不可以在投影空间中进行插值，因为这不是一个线性空间，而插值往往是线性的。

最终经过除法后，我们就得到了片元在视口空间的坐标，也就是一个xy范围都在[0,1]之间的值。而它的zw值，可以看到我们在顶点着色器中直接把裁剪空间的zw值存进了输出结构体中，而片元着色器的输入就是这些插值后的裁剪空间中的zw值。

这意味着，在透视投影中，z值的范围是[-Near,Far]，w值的范围是[Near,Far]；如果是正交投影，那么z值范围是[-1,1]，w值恒为1。