```
22      ------------------------------------------------------------
23
24      模型空间:
25      {
26          a2v 里的数据都是模型空间的 vertex normal tangent
27          可以用内置矩阵方便转到 世界空间 观察空间 裁剪空间
28
29          ObjSpaceViewDir  ==》模型空间观察方向
30          ObjSpaceLightDir ==》模型空间光源方向
31      }
32
33      ------------------------------------------------------------
```

```
3       ------------------------------------------------------------
4
5       世界空间
6       {
7           unity_ObjectToWorld    ==》转到世界空间的矩阵
8
9           UnityObjectToWorldNormal ==》转到世界空间的法线
0
1           WorldSpaceViewDir          ==》转到世界空间的观察方向
2           UnityWorldSpaceViewDir
3
4           WorldSpaceLightDir          ==》转到世界空间的光方向
5           UnityWorldSpaceLightDir
6       }
7
```

```
49
50      观察空间
51      {
52          UNITY_MATRIX_MV ==>模型观察矩阵，用于从模型空间转到观察空间
53          mul(UNITY_MATRIX_MV,xxx)
54
55          UNITY_MATRIX_V ==>观察矩阵 ，用于从世界空间转到观察空间
56
57          UNITY_MATRIX_T_MV ==>模型观察矩阵转置矩阵 其实也是逆矩阵
58          mul((float3x3)UNITY_MATRIX_T_MV,xxx) 从观察矩阵转到模型矩阵
59
60          UnityObjectToViewPos
61      }
62
```

```
裁剪空间
{
    UNITY_MATRIX_MVP ==>模型观察投影矩阵 ，用于从模型空间转到裁剪空间
    mul(UNITY_MATRIX_MVP,xxx)

    UnityObjectToClipPos

    得到clipPos后，会判断该坐标是否需要被裁剪
    然后判断xyz坐标是否都在[-w,w]范围内，
    在坐标范围的话就在视椎体内，不在坐标范围就不在视椎体内 我们就看不见

    必须满足
        float w = clipPos.w
        -w<= clipPos.x <= w
        -w<= clipPos.y <= w
        -w<= clipPos.z <= w

    #####xyz坐标都要在[-w,w]范围内，如果不在，就把这个点剔除
}
```

```
84
85      NDC坐标系
86      {
87          经过顶点着色器之后会得到在视椎体内的裁剪坐标，clipPos
88          然后经过透视除法,除以w分量
89          clipPos = clipPos/clipPos.w
90
91          这时候坐标范围变为[-1,1] ==>归一化的设备坐标即NDC
92
93          ######透视除法之后就从裁剪空间转到到NDC中，clipPos.xyz的范围都在[-1,1]
94      }
95
```

```
97      ------------------------------------------------------------
98      视口坐标 viewPort
99      {
100         归一化的屏幕坐标 ==》 [0,1] $$$$$$$后屏幕处理可以直接当做采样坐标用
101         这里的clipPos.x以及被clipPos.w了，即 clipPos.x = clipPos.x/clipPos.w
102
103         viewport.x = (clipPos.x * 0.5 + 0.5)
104         viewport.y = (clipPos.y * 0.5 + 0.5)
105     }
106
```

```
    ------------------------------------------------------------
    屏幕坐标
    {
        透视除法之后(点的坐标范围为[-1,1])，会进行一次屏幕映射，其实就是第一个缩放过程
        屏幕空间左下角(0,0) 右上角(pixelWidth，pixelHeight)

        screenPos.x =  (clipPos.x * 0.5 + 0.5) * pixelWidth
        screenPos.y =  (clipPos.y * 0.5 + 0.5) * pixelHeight

    }
```

```
19
20      VPOS ==> 屏幕的像素坐标
21      _ScreenParams.xy ==> 屏幕的分辨率
22      VPOS.xy / _ScreenParams.xy ==> 视口坐标
23
```

```
123
124     //clipPos是裁剪坐标（在顶点着色器里还没有经过透视除法）
125     //计算得到屏幕坐标
126     {
127         v2f vert(a2f v)
128         {
129             v2f o
130
131             // clipPos只是经过mvp矩阵转换后的坐标，裁剪坐标，
132             // 并没有经过透视除法，透视除法要顶点着色器之后才进行的
133             float4 clipPos = UnityObjectToClipPos(v.vertex)
134             o.viewPort = ComputeScreenPos(clipPos)
135             return o
136         }
137
138         fixe4 frag()
139         {
140             //是为了避免影响插值，放到片元着色器里自己透视除法下
141             //然后得到视口坐标[0,1]，可以直接当做纹理坐标用
142             viewPort = viewPort/viewPort.w
143         }
144     }
145
```

```
45
46      //ComputeGrabScreenPos 和 ComputeScreenPos一样
47      //都是得到未透视除法的裁剪坐标 并且映射到视口坐标范围[0,1]
48      ComputeGrabScreenPos(pos)
49      {
50          o.x = pos.x * 0.5 + pos.w * 0.5
51          o.y = pos.y * 0.5 + pos.w * 0.5
52          o.zw = pos.zw;
53      }
54
55      srcPos = ComputeGrabScreenPos(clipPos)
56      srcPos = srcPos/srcPos.w ==>
57      {
58          //其实就是视口坐标
59          srcPos.x = 0.5 * clipPos.x/clipPos.w + 0.5
60          srcPos.y = 0.5 * clipPos.y/clipPos.w + 0.5
61      }
62
```

```
63      ----------------------------------------------------------
64      采样坐标
65      {
66          纹理的UV坐标[0,1]，如果是后屏幕处理的话，也可以把视口坐标当做采样坐标
67          我理解为就是视口坐标 ==> [0,1],超过1按照纹理的环绕模式来确定
68      }
69
```

```
-------------------------------------------------------------


//当前的深度值
float d = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv_depth);

i.uv其实是纹理坐标，类比于视口坐标，
==>把视口坐标转到NDC坐标，从[0,1] => [-1,1]
float4 H = float4(i.uv.x * 2 - 1, i.uv.y * 2 - 1, d * 2 - 1, 1);

// I_VP 观察投影矩阵的逆矩阵，从裁剪空间到世界空间
float4 D = mul(_CurrentViewProjectionInverseMatrix, H);
// Divide by w to get the world position.
float4 worldPos = D / D.w;
```

```
  -------------------------------------------------------------

  光源空间
  //光源与片段的距离
  float distance    = length(lightPositions[i] - WorldPos);

  _LightMartrix0 ==> 从世界空间到光源空间的变换矩阵
  _LightColor0    ==> 处理逐像素光源的颜色
  _worldSpaceLightPos ==> 逐像素光源的位置
  _worldSpaceLightPos.w == 0 --> 平行光
  _worldSpaceLightPos.w == 1 --> 点光源或者聚光
```

Unity坐标空间转换.lua
5.67KB