这个边缘检测算法基于<http://williamchyr.com/tag/edge-detection/>给出的算法。十分感谢作者的无私分享。

1. 顶点着色器和片元着色器

简单介绍一下顶点着色器和片元着色器。作者本人也没有深入的学习，对两者的理解可能并不完全正确。顶点着色器的目的是处理物体的每个顶点，计算顶点的法线、顶点颜色等等属性。而片元着色器处理显示出来的当前物体的所有像素，它需要利用顶点着色器处理的结果——经过插值之后计算出面上任意一个点的属性，如法线等等。可以说，顶点着色器是后台处理，片元着色器是前端显示。

1. Unity的OnRenderImage函数

OnRenderImage函数附加在Unity具有Camera组件的脚本中，它是制作边缘检测算法（或者景深算法，等等）所必须的，因为它能够把它所看到的物体的法线和深度信息转换成灰度图传到Texture2D中，供着色器使用。具体写法如下：

void Start()

{

Camera.main.depthTextureMode |= DepthTextureMode.DepthNormals;

}

void OnRenderImage(RenderTexture source, RenderTexture destination)

{

mat.SetFloat("\_SampleDistance", sampleDist);

Graphics.Blit(source, destination, mat);

}

之后在着色器里声明sampler2D \_CameraDepthNormalsTexture变量就可以使用法线和深度信息了。

如果需要传参数给着色器，还可以使用诸如mat.SetFloat ("\_Threshold", lumThreshold);的语句。

三、边缘检测算法

首先和unity自带的边缘检测着色器一样，首先修改顶点着色器，每个点取五个采样点，分别叫做sample0-4，如下：

|  |  |  |
| --- | --- | --- |
| 3 |  | 1 |
|  | 0 |  |
| 2 |  | 4 |

其中，每个格子表示一个像素，sample0表示当前像素。

struct v2f {

float4 pos : SV\_POSITION;

float4 scrPos:TEXCOORD0;

float2 uv[5] : TEXCOORD1;

};

//Vertex Shader

v2f vert (appdata\_base v){

v2f o;

o.pos = mul (UNITY\_MATRIX\_MVP, v.vertex);

o.scrPos=ComputeScreenPos(o.pos);

float2 uv = v.texcoord.xy;

o.uv[0] = uv;

#if UNITY\_UV\_STARTS\_AT\_TOP

if (\_MainTex\_TexelSize.y < 0)

uv.y = 1-uv.y;

#endif

o.uv[1] = uv + \_MainTex\_TexelSize.xy \* half2(1,1) \* \_SampleDistance;

o.uv[2] = uv + \_MainTex\_TexelSize.xy \* half2(-1,-1) \* \_SampleDistance;

o.uv[3] = uv + \_MainTex\_TexelSize.xy \* half2(-1,1) \* \_SampleDistance;

o.uv[4] = uv + \_MainTex\_TexelSize.xy \* half2(1,-1) \* \_SampleDistance;

return o;

}

然后，我们就要仅仅用这5个采样点来决定：这个点是否属于物体的“边缘”，然后把边缘描黑。

在片面着色器中，我们把这5个采样点的法线和距离信息用DecodeDepthNormal函数解码出来：

DecodeDepthNormal(tex2D(\_CameraDepthNormalsTexture, i.scrPos.xy), sample0Depth, sample0Normal);

DecodeDepthNormal(tex2D(\_CameraDepthNormalsTexture, i.uv[1].xy), sample1Depth, sample1Normal);

DecodeDepthNormal(tex2D(\_CameraDepthNormalsTexture, i.uv[2].xy), sample2Depth, sample2Normal);

DecodeDepthNormal(tex2D(\_CameraDepthNormalsTexture, i.uv[3].xy), sample3Depth, sample3Normal);

DecodeDepthNormal(tex2D(\_CameraDepthNormalsTexture, i.uv[4].xy), sample4Depth, sample4Normal);

物体的边缘有两种情况：第一种是处于法线的转折点。这种情况下，我们进行两次比较：sample1和sample2的法线是否相同，sample3和sample4的法线是否相同。如果有任一者不成立，则断定当前点处于法线的转折点，将该点染黑。

inline bool SameNormal(half3 normalA,half3 normalB)

{

half3 normalDelta=abs(normalA-normalB);

return (normalDelta.x+normalDelta.y)<0.3;

}

Unity自带的边缘检测使用同样的方式检测这种情况。

这种情况下一般不会带来误判，除非模型中有**曲面**存在。曲面每点的法线处处连续但不同，因此相邻法线会有较小的差值。因此要设置合理的阈值，防止曲面黑化。

第二种情况是：该点和周围点的高度明显不同，若不同，则这个点是个“阶梯”的边缘（见图），一定要被染黑。这一步判断很容易产生误判，因为接近无穷远处的地面相邻两个像素也会产生较大的深度差。

其实，这种误判是**可以避免**的，通过计算几何，使用法线、深度以及相邻像素距离可以计算出两个深度差大的点究竟属于同一平面还是不同平面，但是需要使用大量的三角函数运算，着色器语言不允许这样低效率的算法。

Unity自带的Edge Detect Normal采用的方法是：仍然进行两次比较：比较sample1和sample2的相对深度差以及sample3和sample4的相对深度差：

float zdiff = abs(centerDepth-sampleDepth);

half isSameDepth = zdiff \* \_Sensitivity.x < 0.09 \* centerDepth;

但这个算法的避免误判的效果不佳。我采用的算法是来源于[williamchyr的博客](williamchyr.com/tag/edge-detection/)的，但又作了少许修改：首先把sample1到sample4的深度取平均数，跟sample0的深度进行比较，比较的结果再乘以sample0的法线z坐标后和阈值相比较：

half avgDepth=(sample1Depth+sample2Depth+sample3Depth+sample4Depth)/4;

edge \*= abs((avgDepth-sample0Depth)\*sample0Normal.z)<0.001;

其中乘以sample0的法向量可以减少无穷远处地面的误判问题，并且尽量不导致近景判断时缺边的情况。

最后渲染结果如下：

