图片描边效果

**1. 导语**

项目中需要对人物图片进行描边，不同于3D物体描边，图片描边有自己的要求和局限。虽然也可以使用后处理，但是考虑到项目实际还是没有采用。目前可行的方法主要可以分为两类，一是查找图片人物外部的像素点，判定一定范围内是否有人物像素点，最终形成描边；二是利用边缘检测算法，检测人物边缘，进行一定程度的外扩形成描边。两种方法各有优缺点，下面进行了具体对比，并进行了一定的优化得到了比较满意的效果。

**2. 基于像素点查找的描边算法**

图片中人物部分通道一般非零，其余部分一般为零，对透明部分的像素点进行判断，如果一定范围内有像素点非透明，则这个像素点可以视作描边部分。其中主要的问题就是如何判断像素点的一定范围内。采用简化处理，可以直接判断描边范围上下左右4个采样点

if (\_Outline > 0 && c.a == 0) {

// Get the neighbouring four pixels.

fixed4 pixelUp = tex2D(\_MainTex, IN.texcoord + fixed2(0, K \* \_MainTex\_TexelSize.y));

fixed4 pixelDown = tex2D(\_MainTex, IN.texcoord - fixed2(0, K \* \_MainTex\_TexelSize.y));

fixed4 pixelRight = tex2D(\_MainTex, IN.texcoord + fixed2(K \* \_MainTex\_TexelSize.x, 0));

fixed4 pixelLeft = tex2D(\_MainTex, IN.texcoord - fixed2(K \* \_MainTex\_TexelSize.x, 0));

// If one of the neighbouring pixels is visible, we render an outline.

if (pixelUp.a + pixelDown.a + pixelRight.a + pixelLeft.a > \_AlphaThreshold) {

c.rgba = \_OutlineColor;

}

}

其中，\_MainTex\_TexelSize是Unity Shader环境中自带的变量，定义如下

float4(1 / texWidth, 1 / texHeight, texWidth, texHeight)

但是当图片中存在细窄区域时，这种采样方法会出现问题，在描边宽度较大时更为明显。可以优化为不仅在描边边缘采样判断，在描边内部也进行判断，即采样点从1到K，

for (int k = 1; k < \_OutlineSize + 1; k++) {…}

有时候为了效率，可以每间隔2,3个点进行采样。

更进一步，当上下左右四个方向填充不够时，可以加上斜上斜下4个方向。

即便如此，当描边宽度较大时，在图片变化较大区域描边边缘会有多出很多蜂刺，不够平滑。



这种现象就是因为采样点数不够，造成描边起伏很大，要想获得比较平滑的效果，只能增加采样点数。这次以像素点为中心，描边宽度为半径的圆形范围内的采样点进行判断，只要不是透明，返回描边颜色。

for(int i = 0; i < steps; i++)

{

float angle = i \* angle\_step \* 2 \* 3.14 / 360;

if(SampleSpriteTexture(uv + fixed2(\_OutlineSize \* cos(angle), \_OutlineSize \* sin(angle))).a > \_AlphaThreshold)

{

outline = true;

break;

}

}



可以看到这样处理之后，描边明显圆滑了不少，效果比较好，可惜代价是采样点数太多，性能消耗太大。

**3. 基于图像边缘检测的描边算法**

边缘检测是数字图像中用来检测图像边缘，即局部变化明显区域的算法。一般算法是将图像通过滤波算子卷积得到结果。常用的边缘检测算子有Soble算子，Robert算子，Prewitt算子，LOG算子和Canny算子等。

这里主要利用Sobel算子和Roberts算子进行边缘检测。

   

Roberts Gx Roberts Gy Sobel Gx Sobel Gy

我们直接检测图片的通道边缘。

// Sobel Edge Detection

for (int k = 0; k < 9; ++k)

{

fixed4 color = tex2D(\_MainTex, i.uv[k]);

edgeX += color .a \* Gx[k];

edgeY += color .a \* Gy[k];

}

float edge = 1 - abs(edgeX) - abs(edgeY);



可以看到，基于图像边缘检测的描边效果还是不错的，不过在图片变化明显的地方，会检测到多条边界，这是因为描边宽度导致的，变化部分仍然不够平滑。

**4. 边缘模糊**

在图像处理中，进行边缘检测之前通常会进行图像模糊，降低噪声影响，以获得较好的检测结果。目前只是利用边缘检测，得到的结果不够平滑，需要对人物边缘进行模糊。我们只是简单地采用一次降采样进行模糊，避免过多的性能消耗。

因为要对边缘进行模糊，所以申请一张RenderTexture，设置RenderTexture的长宽分别为屏幕长宽的一般，利用CommandBuffer将边缘绘制到申请的RenderTexture上。CommandBuffer添加相机事件CameraEvent.BeforeForwardAlpha

cameara.AddCommandBuffer (CameraEvent.BeforeForwardAlpha, cmdBuffer);

绘制边缘时，利用CommandBuffer.DrawRenderder(…)进行绘制。绘制完成后，设置原始图片材质的\_OutlineTex为RenderTex。原始图片材质使用\_OutlineTex时需要注意，因为\_OutlineTex是降采样后的图片，直接使用可能会有模糊现象，需要再进行一次平滑处理，对周围4个采样点进行平均即可。

fixed4 frag (v2f i) : SV\_TARGET

{

fixed4 col;

fixed4 mainColor = tex2D(\_MainTex, i.uv);

fixed4 outlineColor = SoftOutline(\_OutlineTex, i.edgeUV);

…

col = lerp(outlineColor, mainColor, step(\_OriginAlphaThreshold, mainColor.a));

return col;

}

最后得到的结果如图所示。



从图中可以看出，人物描边比较平滑，效果不错，整体性能也在可以接受的范围内。

**5. 总结**

基于像素点查找的描边方法和基于图像边缘检测的描边方法，都可以绘制人物的边缘，在描边宽度不大时，都具有良好的效果，但是当描边宽度较大时，两种方法都会出现固有的缺陷，出现较多的蜂刺，效果不好。后面将边缘检测结合模糊处理，对边缘进行降采样，再进行绘制能够得到比较满意的效果，性能消耗可以接受。

虽然只是简单的图片描边，但是在效果和性能上进行平衡还是一件需要仔细考虑的事啊。

**6. 参考资料**

<https://nielson.io/2016/04/2d-sprite-outlines-in-unity>

<https://blog.csdn.net/BuladeMian/article/details/57082556>

<https://assetstore.unity.com/packages/vfx/shaders/2d-sprite-outline-109669>

<https://blog.csdn.net/qq_33288536/article/details/82765330?tdsourcetag=s_pctim_aiomsg>