

AA

1 简要介绍

由于渲染过程中采样不足的问题，会产生锯齿，给用户带来视觉上的不smoothness。所以需要引入抗锯齿（Anti-Aliasing, AA）技术，让画面的过渡处更平滑，从而解决这个问题。

锯齿问题分为两类：

（1）几何锯齿：产生的主要原因是对几何覆盖函数的采样不足，会在物体边缘产生锯齿。

（2）着色锯齿：在着色阶段，对渲染方程的采样不足，导致高光等效果在快速变换时会产生闪烁或噪点。

1.1 采样定理

奈奎斯特-香农采样定理（Niquist-Shannon sampling theorem）：

要从离散样本中重构信号，必须以至少是该信号最高频率两倍的速率进行采样。这个最低采样频率被称为奈奎斯特速率。如果采样速率低于此值，就会发生频谱混叠，导致信息丢失且无法恢复原始信号。

2 SSAA

超级采样抗锯齿（Super-Sampling Anti-Aliasing, SSAA），其核心思想为：当目标渲染分辨率为 $W \times H$ 时，先以 $kW \times kH (k = 2, 3, 4)$ 的分辨率渲染场景，包含颜色、深度等信息，然后对结果进行下采样，得到 $W \times H$ 的图像。下采样时，可以使用 box filter, Gaussian filter 等滤波器。

优点：实现简单（提高采样率）

缺点：资源消耗大，渲染速度慢

3 MSAA

多重采样抗锯齿 (Multi-Sample Anti-Aliasing, MSAA)，其核心思想为：每个像素设置多个 (4、8、16) 子像素，每个子像素有自己的 color buffer 和 depth buffer，所有子像素共享该像素的着色计算结果 (即每个像素只运行一次片元着色器)。

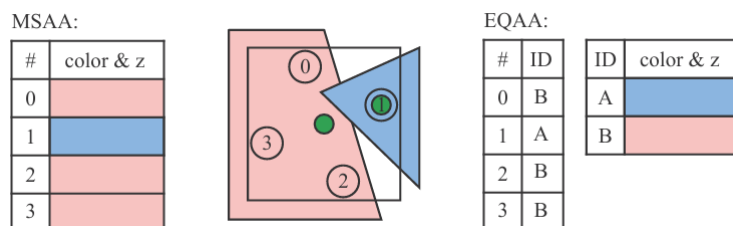


图 1: MSAA and EQAA

如图1所示，每个像素对应 4 个子像素，其运算过程如下：

(1) 对 4 个子像素执行三角形覆盖测试，每个子像素用 coverage mask 表示是否被三角形覆盖。

(2) 在像素着色阶段，对至少一个子像素被覆盖的像素，用像素中心点坐标或某个子像素的坐标执行一次片元着色器，计算出该像素的颜色。

注：三角形可能没有覆盖到像素中心点，此时用像素中心点坐标执行片元着色器，可能会导致颜色不准确。GPU 硬件会通过 centroid sampling 来调整采样坐标，如果覆盖了像素中心点，则使用像素中心点坐标，否则使用最近的被覆盖的子像素坐标。

(3) 对每个被覆盖的子像素，执行深度测试和模板测试，更新 depth buffer 和 stencil buffer，深度根据通过测试的 sample 进行插值获得。

(4) 将像素计算的结果写入被覆盖的子像素的 color buffer 中。

(5) 所有着色计算结束后，将所有子像素的颜色 resolve 到该像素的 color buffer 中，通常是对所有子像素的颜色进行平均。

注：

在图1中，左侧为 MSAA，右侧为 EQAA (Enhanced Quality Anti-Aliasing)，EQAA 在每个子像素只存储对应 color buffer 和 depth buffer 的索引，通过索引访问共享的 color buffer 和 depth buffer，从而节省显存。

优点：相比 SSAA，渲染速度更快；硬件支持

缺点: 需要存储多个子像素的 color buffer 和 depth buffer 以及 coverage mask, 显存消耗较大

3.1 SSAA 和 MSAA 的不同

以 2 倍目标分辨率为例, SSAA 需要进行 $2W \times 2H$ 次着色计算, 而 MSAA 只需要进行 $W \times H$ 次着色计算。

3.2 延迟渲染和 MSAA

延迟渲染是可以使用 MSAA 的。延迟渲染使用 GBuffer 存储深度、法线等数据, 在着色阶段进行计算时, 已经丢失了场景的几何信息, 无法判断三角形对像素的覆盖情况, 所以无法进行 MSAA 的多重采样。如果一定要用 MSAA, 需要在生成 GBuffer 阶段记录几何遮挡信息。

目前 PC 或主机平台在使用延迟渲染时, 一般不使用 MSAA, 而是使用 FXAA、TAA、DLAA 等后处理抗锯齿技术。

4 FXAA

快速近似抗锯齿 (Fast Approximate Anti-Aliasing, FXAA), 其核心思想为: 对渲染结果进行后处理, 检测出锯齿边缘, 然后对边缘进行模糊处理, 从而达到抗锯齿的效果。

4.1 算法流程

此处讲述两个版本的 FXAA, 分别为 FXAA Quality (用于 PC) 和 FXAA Console (用于主机)。

4.1.1 FXAA Quality

(1) 对比度计算 (找到边缘)

如图2所示, 计算当前像素和其上、下、左、右四个像素的亮度, 最大值和最小值的差值即为对比度。

```
1 float maxLuma = max(lumaM, lumaN, lumaW, lumaE, lumaS);
2 float minLuma = min(lumaM, lumaN, lumaW, lumaE, lumaS);
3 float contrast = maxLuma - minLuma;
```

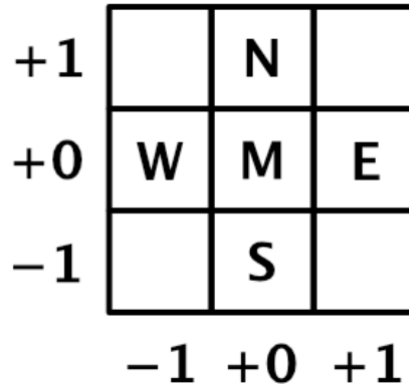


图 2: FXAA Compute Contrast

```

4  if (contrast < threshold) {
5      // No edge detected, return original color
6      return colorM;
7  }
8
9  // 上述判断会将对比度高的地方都当作边缘，会丢失局部高频信息
10 // 考虑添加额外的阈值修正操作
11 // 如果当前区域亮度较高，需要更高的对比度才能被认为是边缘
12 if (contrast > max(minThreshold, maxLuma * threshold)) {
13     // Edge detected, proceed to next step
14 } else {
15     return colorM;
16 }

```

(2) 对相邻像素进行加权混合，和中间像素的亮度计算差值，并进行归一化处理，计算混合系数

如图3所示，采样当前像素周围的 8 个像素，对应权重如图 4所示，进行加权混合。

```

1  float GetSubpixelBlendFactor (LumaNeighborhood luma) {
2      float filter = 2.0 * (luma.n + luma.e + luma.s + luma.w);
3      filter += luma.ne + luma.nw + luma.se + luma.sw;
4      filter *= 1.0 / 12.0;
5      filter = abs(filter - luma.m);
6      filter = saturate(filter / luma.range);

```

+1	NW	N	NE
+0	W	M	E
-1	SW	S	SE
	-1	+0	+1

图 3: FXAA Neighbour Sample

1	2	1
2		2
1	2	1

图 4: FXAA Neighbour Weight

```

7     filter = smoothstep(0, 1, filter);
8     return filter * filter;
9 }

```

(3) 计算混合方向

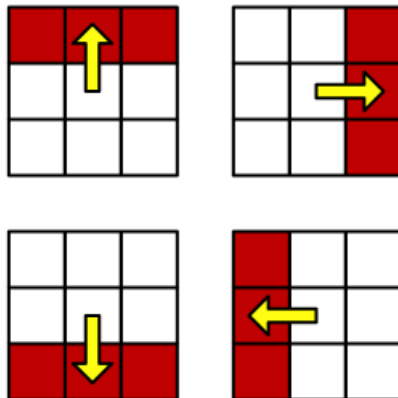


图 5: Possible Blend Directions

如图5所示，从四个可能的方向中，选择一个最接近的方向。

```

1 float horizontal =
2     2.0 * abs(luma.n + luma.s - 2.0 * luma.m) +
3     abs(luma.ne + luma.se - 2.0 * luma.e) +
4     abs(luma.nw + luma.sw - 2.0 * luma.w);
5 float vertical =
6     2.0 * abs(luma.e + luma.w - 2.0 * luma.m) +
7     abs(luma.ne + luma.nw - 2.0 * luma.n) +
8     abs(luma.se + luma.sw - 2.0 * luma.s);
9
10
11 bool isHorizontal = horizontal >= vertical;
12 float2 pixelStep = isHorizontal ? float2(0, TexelSize.y) : float2(TexelSize.x, 0);
13 float lumaP = isHorizontal ? lumaN : lumaE;
14 float lumaN = isHorizontal ? lumaS : lumaW;
15 float gradientP = abs(lumaP - lumaM);
16 float gradientN = abs(lumaN - lumaM);
17 if (gradientP < gradientN) {
18     pixelStep = -pixelStep;

```

```
19 }
```

先判断是水平方向还是垂直方向，然后再判断是正方向还是负方向。

(4) 混合

根据 (2) 中计算的混合系数 (blend Factor) 和 (3) 中计算的混合方向，对当前像素进行混合。

```
1 float4 result = tex2D(_MainTex, UV + blendFactor * pixelStep);
```

4.1.2 FXAA Quality 改进

问题:

(1) 我们只在 3×3 的像素块范围内进行采样，采样范围过小，可能会漏掉一些边缘。

(2) 只考虑了水平和垂直方向的边缘，没有考虑带角度的边缘。

改进:

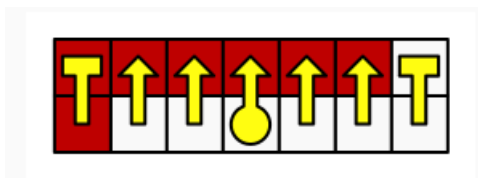


图 6: Search for the end points

如图6所示，先确认边缘是水平方向还是垂直方向，然后沿着两个相反方向进行采样，计算亮度值的差，判断是否为边界。

```
1 if (gradientP < gradientN) {
2     edge.pixelStep = -edge.pixelStep;
3     edge.lumaGradient = gradientN;
4     edge.otherLuma = lumaN;
5 }
6 else {
7     edge.lumaGradient = gradientP;
8     edge.otherLuma = lumaP;
9 }
10
11 float edgeLuma = 0.5 * (luma.m + edge.otherLuma);
12 float gradientThreshold = 0.25 * edge.lumaGradient;
```

```

13 float2 uvP = edgeUV + uvStep;
14 float lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
15 bool atEndP = lumaGradientP >= gradientThreshold;

```

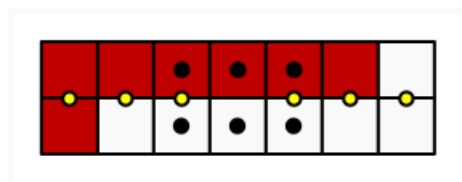


图 7: Search (yellow) and neighborhood (black) samples

如图7所示，采样的优化，利用双线性过滤，在边界处采样（即图中的黄点），就可以得到两侧的平均亮度值。

4.1.3 FXAA Console

相比于 FXAA Quality, FXAA Console 减少采样次数，降低性能开销，适用于主机平台。

（1）如图8所示，采样当前像素的四个角上的亮度，用 2×2 的 box filter 计算边缘走向在 x 方向和 y 方向上的投影。



图 8: FXAA Console Sample

```

1 float2 dir;
2 dir.x = -((lumaNW + lumaNE) - (lumaSW + lumaSE));
3 dir.y = ((lumaNW + lumaSW) - (lumaNE + lumaSE));

```

（2）计算对比度，判断是否为边缘

（3）沿着切线方向进行采样

（3.1）沿着切线方向分别向正负两个方向进行采样，取平均值。

```

1 float2 direction1 = normalize(dir) * _MainTex_TexelSize.xy * _Scale;

```



```

2
3 float4 N1 = tex2D(_MainTex, uv + direction1);
4 float4 P1 = tex2D(_MainTex, uv - direction1);
5 float4 result = 0.5 * (N1 + P1);

```

(3.2) 如果只有 (3.1) 的采样结果, 对于水平和垂直方向的锯齿, 效果不太好; 考虑再进行一次采样, 往更远处偏移。

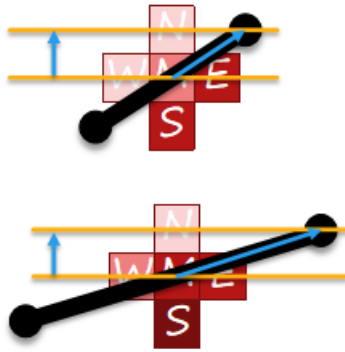


图 9: FXAA Sample Direction

```

1 float dirAbsMinTimesC = min(abs(dir.x), abs(dir.y)) * _Sharpness;
2 float2 direction2 = clamp(direction1 / dirAbsMinTimesC, -2, 2) * 2;
3
4 float4 N2 = tex2D(_MainTex, uv + direction2 * _MainTex_TexelSize.xy);
5 float4 P2 = tex2D(_MainTex, uv - direction2 * _MainTex_TexelSize.xy);
6 float4 result2 = result * 0.5f + (N2 + P2) * 0.25f;
7
8 // 如果第二次采样到亮度变化较大的区域, 丢弃新结果
9 if (Luminance(Result2.xyz) > MinLuma && Luminance(Result2.xyz) < MaxLuma) {
10     Result = Result2;
11 }

```

4.2 Unity 的 HDRP 中的 FXAA

主要代码在 FXAA.compute 和 FXAA.hlsl 中, 其主要采用的是 FXAA Console 的思路。

注意点:

(1) 若此时在 HDR 模式下, 需先进行 Tonemap, 映射到 LDR, 再计算亮度。

(2) Unity 对 direction 的采样策略 (同样采样 9 次) 如下

```
1 #define FXAA_SPAN_MAX      (8.0)
2 #define FXAA_REDUCE_MUL    (1.0 / 8.0)
3 #define FXAA_REDUCE_MIN    (1.0 / 128.0)
4
5 float2 dir;
6 dir.x = -((lumaNW + lumaNE) - (lumaSW + lumaSE));
7 dir.y = ((lumaNW + lumaSW) - (lumaNE + lumaSE));
8
9 float lumaSum = lumaNW + lumaNE + lumaSW + lumaSE;
10 float dirReduce = max(lumaSum * (0.25 * FXAA_REDUCE_MUL), FXAA_REDUCE_MIN);
11 float rcpDirMin = rcp(min(abs(dir.x), abs(dir.y)) + dirReduce);
12
13 dir = min((FXAA_SPAN_MAX).xx, max((-FXAA_SPAN_MAX).xx, dir * rcpDirMin))
14         * _ScreenSize.zw;
15
16 float3 rgb03 = tex2D(_MainTex, uv + dir * (0.0 / 3.0 - 0.5)).xyz;
17 float3 rgb13 = tex2D(_MainTex, uv + dir * (1.0 / 3.0 - 0.5)).xyz;
18 float3 rgb23 = tex2D(_MainTex, uv + dir * (2.0 / 3.0 - 0.5)).xyz;
19 float3 rgb33 = tex2D(_MainTex, uv + dir * (3.0 / 3.0 - 0.5)).xyz;
20
21 float3 rgbA = 0.5 * (rgb13 + rgb23);
22 float3 rgbB = rgbA * 0.5 + 0.25 * (rgb03 + rgb33);
23
24 float lumaB = Luminance(rgbB);
25
26 float lumaMin = Min3(lumaM, lumaNW, Min3(lumaNE, lumaSW, lumaSE));
27 float lumaMax = Max3(lumaM, lumaNW, Max3(lumaNE, lumaSW, lumaSE));
28
29 float3 rgb = ((lumaB < lumaMin) || (lumaB > lumaMax)) ? rgbA : rgbB;
```

(3) 如果有 alpha 通道, alpha 通道单独处理。

4.3 参考

FXAA White Paper
FXAA 3.11

5 MLAA

形态抗锯齿 (Morphological Anti-Aliasing, MLAA), 也是后处理的 AA 算法, 其思想与 FXAA 类似, 也是通过计算相邻像素的差值, 来判断像素的边缘, 从而进行边缘处理。

5.1 算法流程

此处以 Jimenez 等人的论文为例, 介绍 MLAA 的算法流程, 如图10所示, 主要有三个步骤。

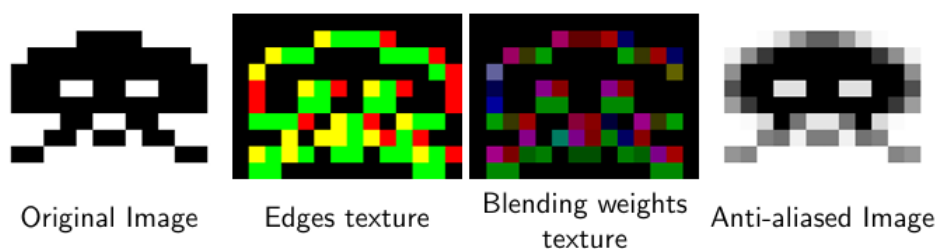


图 10: MLAA Process

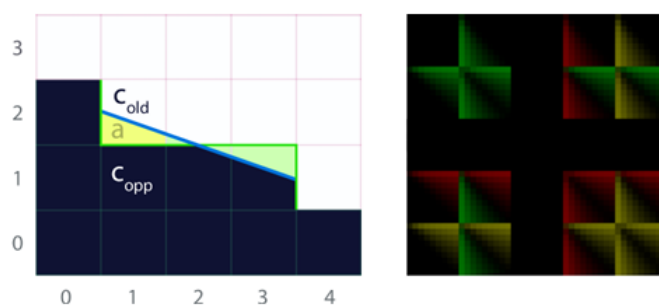


图 11: MLAA Process and Area Texture

如图11所示, MLAA 的核心在于对像素进行重矢量化 (revectorization), 图中蓝色线即为重矢量化线。

$$c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}$$

通过上述公式计算 MLAA 的结果，其中 a 即图11的黄色面积， a 是重矢量化线样式和长度的函数，可以通过预计算的 Area Texture 获得。

5.1.1 边缘检测

论文中提到用深度作为边界检测的依据，也可以使用亮度作为依据。

区别：

- (1) 用深度作为边界检测依据，用时更短，对几何边缘更敏感。
- (2) 用亮度作为边界检测依据，可以处理着色锯齿和镜面高光。

```

1 float4 EdgeDetectionPS(float4 position : SVPOSITION,
2   float2 texcoord : TEXCOORD0): SVTARGET {
3   float D = depthTex.SampleLevel(PointSampler,
4   texcoord , 0);
5   float Dleft = depthTex.SampleLevel(PointSampler,
6   texcoord , 0, int2(1, 0));
7   float Dtop = depthTex.SampleLevel(PointSampler,
8   texcoord , 0, int2(0, 1));
9   // We need these for updating the stencil buffer .
10  float Dright = depthTex.SampleLevel(PointSampler,
11  texcoord , 0, int2(1, 0));
12  float Dbottom = depthTex.SampleLevel(PointSampler,
13  texcoord , 0, int2(0, 1));
14  float4 delta = abs(D.xxxx -
15    float4(Dleft , Dtop, Dright , Dbottom));
16  float4 edges = step(threshold .xxxx, delta );
17  if (dot(edges , 1.0) == 0.0) {
18    discard;
19  }
20
21  return edges;
22 }
```

优化：因为边界信息是共享的，不用为每个像素保存四个方向的边界信息，只需保存左边和上边像素的边界信息即可。

5.1.2 计算混合权重

(1) 距离查找

在边缘检测的过程中,我们将检测结果存储到 Texture 中,依靠 Texture 的双线性过滤,对 Texture 的边缘位置进行采样,可以一次获得两个像素的边缘信息,有以下三种情况:

- (1) 0.0: 两个像素都不在边缘,
- (2) 1.0: 两个像素都在边缘,
- (3) 0.5: 一个像素在边缘,一个像素不在边缘。

```
1 float SearchXLeft(float2 texcoord) {
2     texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
3     float e = 0.0;
4     // We offset by 0.5 to sample between edgels, thus fetching
5     // two in a row.
6     for (int i = 0; i < maxSearchSteps; i++) {
7         e = edgesTex.SampleLevel(LinearSampler, texcoord, 0).g;
8         // We compare with 0.9 to prevent bilinear access precision
9         // problems.
10        [flatten] if (e < 0.9) break;
11        // 此处用步长为2加速查找
12        texcoord -= float2(2.0, 0.0) * PIXEL_SIZE;
13    }
14    // When we exit the loop without finding the end, we return
15    // 2 * maxSearchSteps.
16    return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
17 }
```

(2) 获取交叉边界

在距离查找的过程中,对于边缘检测的结果,相邻的两个像素,如果一个像素值为 1.0,另一个像素值为 0.0,在中间位置进行采样得到 0.5 的结果,无法判断哪一个边界,所以这里用 offset 为 0.25 进行采样,判断哪一边是边缘。

如图12所示,分别为交叉边界 (Cross Edge) 的四种情况。

(3) 预计算 Area Texture

如图13所示,覆盖率共有 16 中情况,此处用预计算的贴图。

```
1 #define NUMDISTANCES 9
2 #define AREA_SIZE (NUMDISTANCES * 5)
```



图 12: Cross Edge

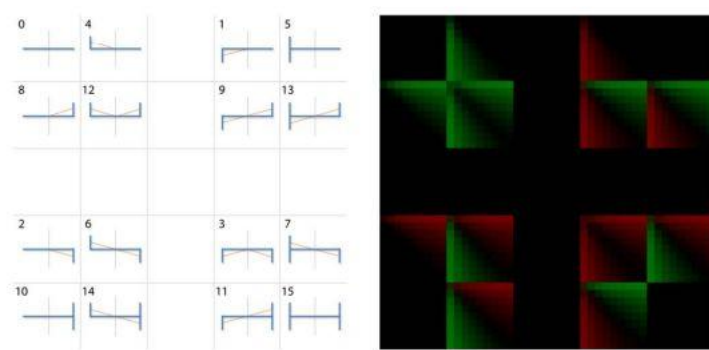


图 13: Precomputed Area Texture

```

3 float2 Area(float2 distance , float e1 , float e2) {
4     // * By dividing by AREA SIZE 1.0 below we are
5     // implicitly offsetting to always fall inside a pixel .
6     // * Rounding prevents bilinear access precision problems.
7     float2 pixcoord = NUMDISTANCES * round(4.0 * float2(e1 , e2)) + distance;
8     float2 texcoord = pixcoord / (AREA_SIZE - 1.0);
9     return areaTex.SampleLevel(PointSampler , texcoord , 0).rg;
10 }
11
12
13 float4 BlendingWeightCalculationPS(
14     float4 position : SVPOSITION,
15     float2 texcoord : TEXCOORD0): SVTARGET {
16     float4 weights = 0.0;
17     float2 e = edgesTex.SampleLevel(PointSampler , texcoord , 0). rg ;
18     [branch]
19     if (e.g) {
20         // Edge at north
21         float2 d = float2(SearchXLeft(texcoord),
22                             SearchXRight(texcoord));
23         // Instead of sampling between edgels , we sample at 0.25,
24         // to be able to discern what value each edgel has.
25         float4 coords = mad(float4(d.x, 0.25, d.y + 1.0, 0.25),
26                             PIXEL_SIZE.xyxy , texcoord.xyxy);
27         float e1 = edgesTex.SampleLevel(LinearSampler ,
28                                         coords.xy , 0).r ;
29         float e2 = edgesTex.SampleLevel(LinearSampler ,
30                                         coords.zw , 0).r ;
31         weights.rg = Area(abs(d), e1 , e2);
32     }
33     [branch]
34     if (e.r) {
35         // Edge at west
36         float2 d = float2(SearchYUp(texcoord),
37                             SearchYDown(texcoord));
38         float4 coords = mad(float4( 0.25, d.x, 0.25, d.y + 1.0),
39                             PIXEL_SIZE.xyxy , texcoord.xyxy);
40         float e1 = edgesTex.SampleLevel(LinearSampler ,
41                                         coords.xy , 0).g;

```

```

42     float e2 = edgesTex.SampleLevel(LinearSampler,
43                                     coords.zw, 0).g;
44     weights.ba = Area(abs(d), e1, e2);
45 }
46 return weights;
47 }

```

5.1.3 对 4 个相邻结果进行混合

```

1 float4 NeighborhoodBlendingPS(
2     float4 position : SVPOSITION,
3     float2 texcoord : TEXCOORD0): SVTARGET {
4     float4 topLeft = blendTex.SampleLevel(PointSampler,
5                                           texcoord, 0);
6     float right = blendTex.SampleLevel(PointSampler,
7                                         texcoord, 0, int2(0, 1)).g;
8     float bottom = blendTex.SampleLevel(PointSampler,
9                                         texcoord, 0, int2(1, 0)).a;
10    float4 a = float4(topLeft.r, right, topLeft.b, bottom);
11    float sum = dot(a, 1.0);
12    [branch]
13    if (sum > 0.0) {
14        float4 o = a * PIXEL_SIZE.yyxx;
15        float4 color = 0.0;
16        color = mad(colorTex.SampleLevel(LinearSampler,
17                                         texcoord + float2(0.0, o.r), 0), a.r, color);
18        color = mad(colorTex.SampleLevel(LinearSampler,
19                                         texcoord + float2(0.0, o.g), 0), a.g, color);
20        color = mad(colorTex.SampleLevel(LinearSampler,
21                                         texcoord + float2(o.b, 0.0), 0), a.b, color);
22        color = mad(colorTex.SampleLevel(LinearSampler,
23                                         texcoord + float2(o.a, 0.0), 0), a.a, color);
24        return color / sum;
25    } else {
26        return colorTex.SampleLevel(LinearSampler, texcoord, 0);
27    }
28 }

```


5.2 参考

Morphological Antialiasing

Practical Morphological Anti-Aliasing

AMD MLAA Sample

主流抗锯齿方案详解（四）SMAA

6 SMAA

子像素形态抗锯齿 (Subpixel Morphological Anti-Aliasing, SMAA), 是 MLAA 的改进版, 整体流程如图14所示。

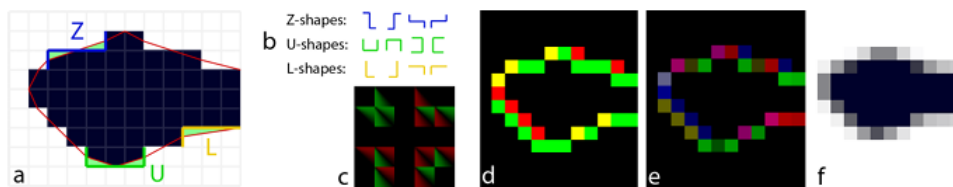


图 14: SMAA Overview

6.1 算法流程

6.1.1 边界检测

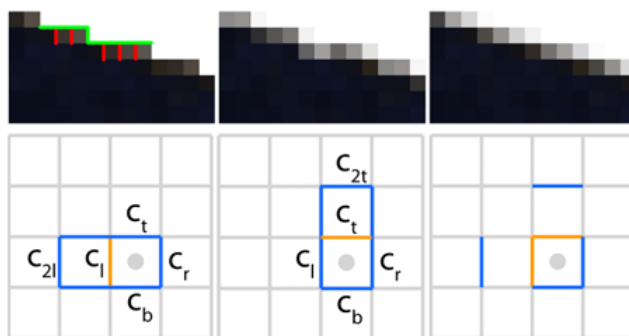


图 15: Local Contrast Adaptation

当我们进行边缘检测的时候，由于忽略了局部对比度，很可能选择了错误的边缘，导致对边缘形状的分类出现错误。如图15所示，第一行左侧是原图，中间是没有考虑局部对比度产生的 AA 结果，右边是 SMAA 产生的结果，可以看到 SMAA 的结果更好。

在图15中，第二行第一幅图中，我们需要找到左侧正确的边缘，其中有 c_l 和 c_{2l} 两个可能的边缘。论文给出的判断方法如下：

- (1) 计算最大对比度 c_{max}

$$c_{max} = \max(c_t, c_r, c_b, c_l, c_{2l})$$

- (2) 判断亮度差是否大于阈值

$$e_l = |L - L_l| > T$$

- (3) 判断是否为边界

$$e'_l = e_l \ \& \ (c_l > 0.5 \cdot c_{max})$$

只有同时满足局部对比度的判断和亮度差的判断，才能认为是边界。

6.1.2 边界分类

- (1) 保留更尖锐的几何特征

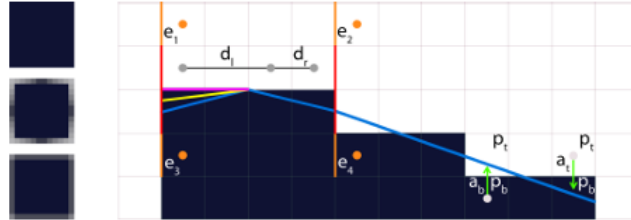


图 16: Sharp Geometric Features

如图16所示，左上的正方形是没有经过 AA 处理，左中的正方形是经过 MLAA 处理，左下的正方形是经过 SMAA 处理，可以看到 SMAA 更好地保留了形状特征。

对于这种情况，我们需要检测到正确的拐角（corner），保留物体真正的形状。引入缩放因子 r ，对 MLAA 计算的覆盖区域进行调整， r 的取值范围为 $[0, 1]$ ； $r = 1.0, 0.5, 0.0$ 分别对应图16中的蓝线、黄线和粉线。

(1.1) 用 MLAA 中的边缘形状检测的方法，对应的 cross edge 为图16中的红边。由此得到两块区域 a_b 和 a_t ， a_b 用于将 p_b 和它的上邻居 p_t 混合， a_t 用于将 p_t 和它的下邻居 p_b 混合。

(1.2) 将红色的锯齿边界在往外取边界值，如图16中的橙色边界。对覆盖区域进行调整：

$$a'_t = \begin{cases} r \cdot a_t, & \text{if } d_l < d_r \text{ and } e1 \\ r \cdot a_t, & \text{if } d_l \geq d_r \text{ and } e2 \\ a_t, & \text{otherwise} \end{cases}$$

$$a'_b = \begin{cases} r \cdot a_b, & \text{if } d_l < d_r \text{ and } e3 \\ r \cdot a_b, & \text{if } d_l \geq d_r \text{ and } e4 \\ a_b, & \text{otherwise} \end{cases}$$

(2) 对角线情况

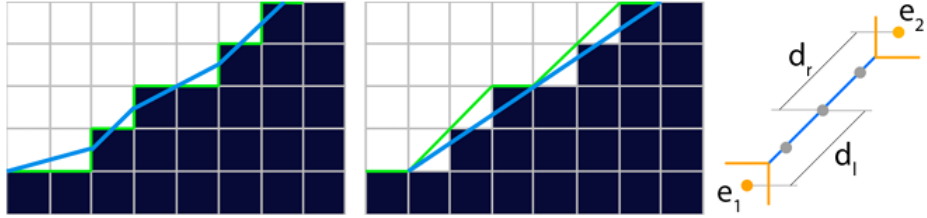


图 17: Diagonal Pattern

如图17所示，计算对角线覆盖的区域：

(2.1) 计算到对角线两端的距离 d_l, d_r 。

(2.2) 获取对应的交叉边界 e_1, e_2 。

(2.3) 根据输入 (d_l, d_r, e_1, e_2) 定义对角线的变化情况，对预计算的 Area Texture 进行采样，获取覆盖区域 a_t 和 a_b 。

对角线可能的情况如图18所示，先进行对角线情况检测，如果失败，在进行水平方向和垂直方向的检测。

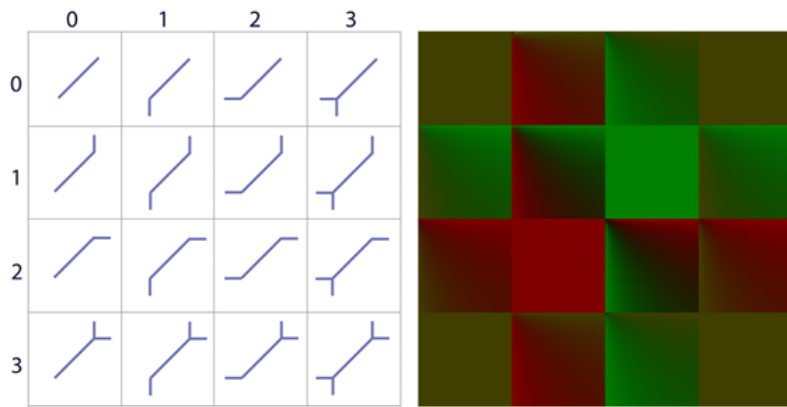


图 18: Diagonal Pattern Map and precomputed area texture

(3) 更准确的距离搜索

MLAA 仅沿着水平方向和垂直方向进行距离搜索，容易忽略交叉边界。

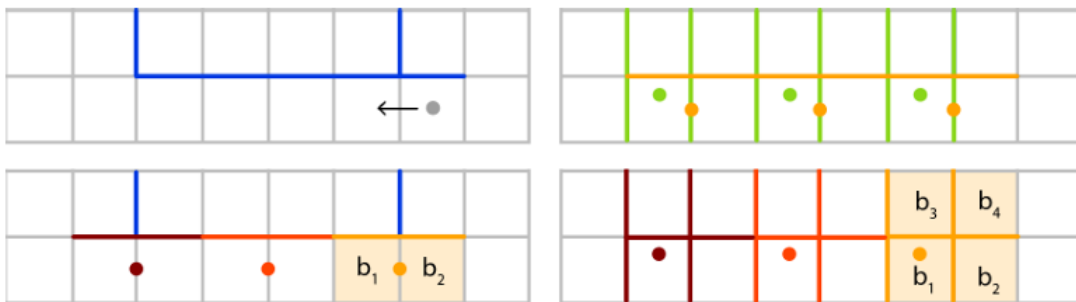


图 19: SMAA Search

如图19所示，左上方的图为向左查找边缘的端点的情况。在右上方的图中，橙色的点为 MLAA 的采样点，绿色的点为 SMAA 的采样点，当搜索过程能在第一次找到交叉边界时就停止。左下方的图对应 MLAA 的搜索过程，它可能会错过交叉边缘，如图中的蓝边。右下方的图对应 SMAA 的搜索过程，可以找到正确的边缘。

MLAA 的一次采样，应用硬件的线性插值，只能获取两个像素点的边界信息；SMAA 的一次采样，应用硬件的双线性插值，可以获取四个像素点的边界信息，准确度更高。

6.2 参考

SMAA

7 TAA

时间抗锯齿 (Temporal Anti-Aliasing, TAA), 是利用时间上的信息进行抗锯齿处理的一种方法。TAA 的基本思想是通过结合当前帧和之前帧的信息, 来减少锯齿和闪烁现象, 从而提高图像质量。

7.1 jittering

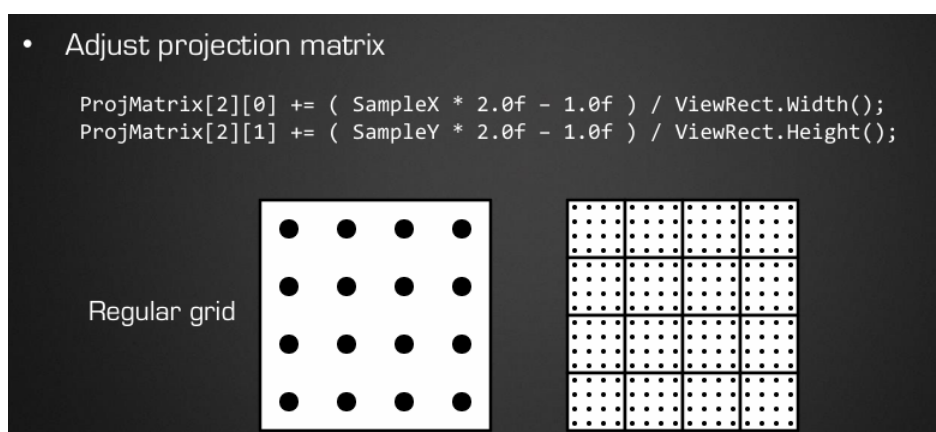


图 20: Jittering

MSAA 要在一帧内对像素进行多次采样, 而 TAA 是通过在不同帧之间对像素进行采样来实现抗锯齿效果。通过在每一帧中计算一个 jittering, 一般是由低差异序列生成, 常见的由 Halton 序列, 将其应用于投影矩阵, 实现采样的抖动, 如图20和图21所示。

7.2 TAA 的位置

如图22所示, TAA 一般放在渲染管线的后期处理阶段, 先进行一次 Tone Mapping, 然后进行 TAA 处理, 然后进行 Invert Tone Mapping, 再进行其它后处理流程。

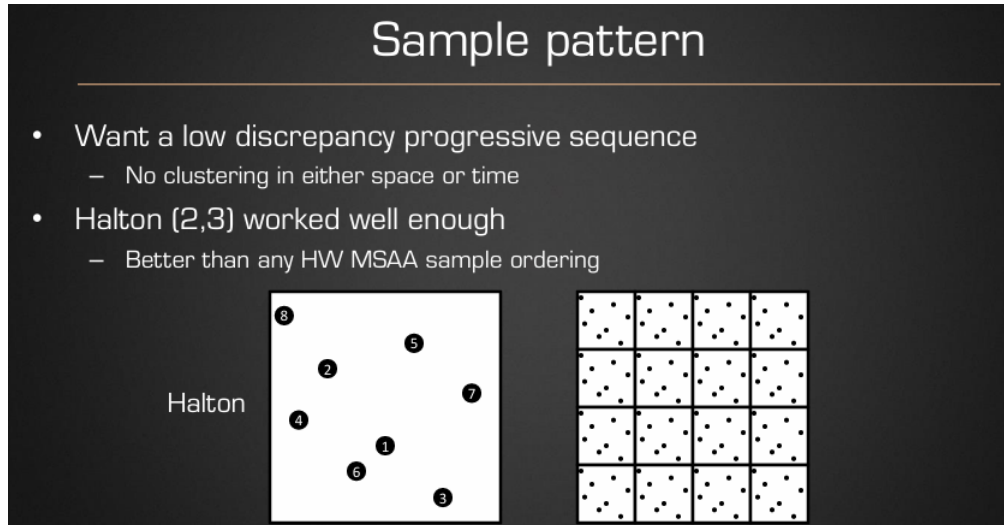


图 21: Sample Pattern

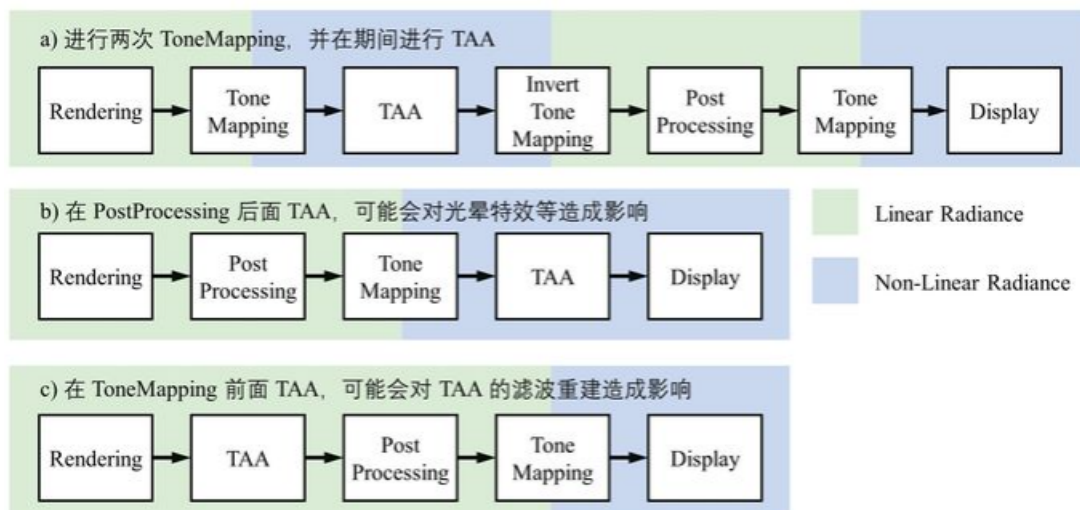


图 22: TAA Pipeline Position

7.3 重投影

当物体不动，只有镜头的移动，可以利用当前帧的深度信息，计算出物体的世界坐标，然后根据上一帧的观察矩阵和投影矩阵，计算出物体上一帧在屏幕上的位置。

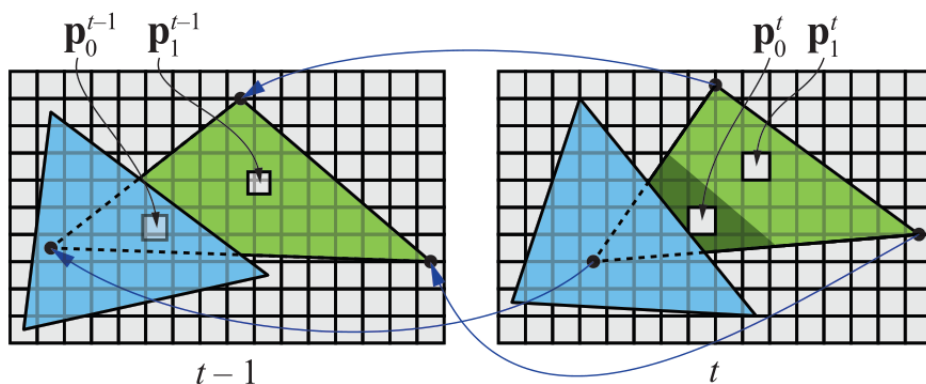


图 23: TAA Reprojection

7.4 动态物体

计算并存储 motion vector，根据 motion vector 计算出当前物体在上一帧的屏幕空间位置，找到正确的历史帧信息。

7.5 Ghosting

如图24所示，物体移动时，历史帧的信息可能会与当前帧的信息不匹配（物体之间的遮挡），导致重影现象。

UE 给出的两种解决方案是（通过历史帧的颜色信息和当前帧领域内的颜色进行比较）：

(1) clamp

在当前像素的 3×3 的领域内，计算颜色空间（转换到 YCoCg 空间效果更好）的 AABB，将历史帧的颜色与 AABB 进行 clamp。

(2) clip

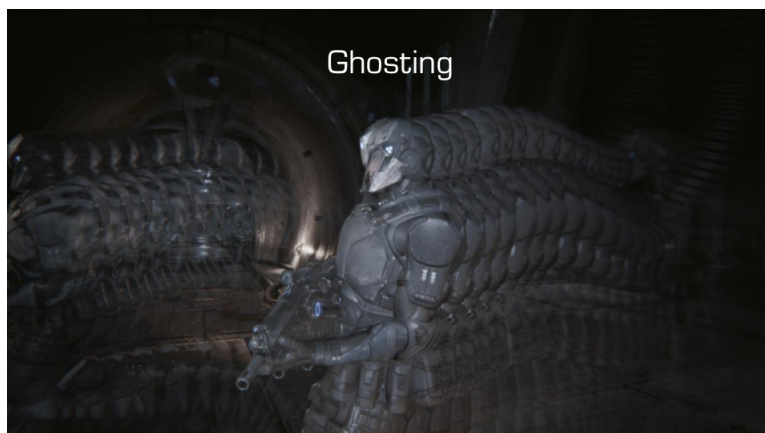


图 24: Ghosting

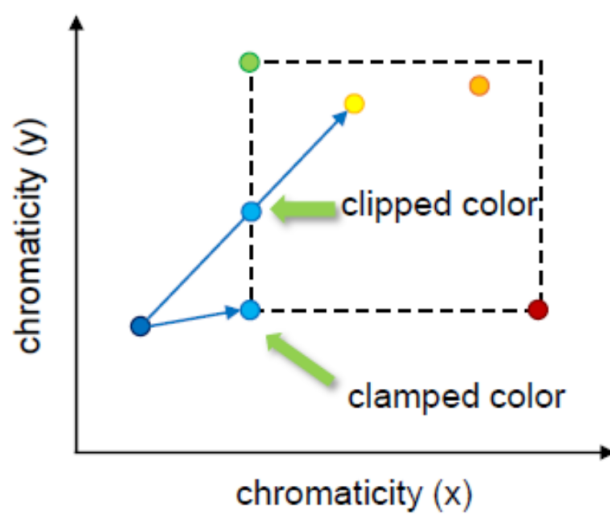


图 25: Clip

在颜色空间内，连接 AABB 中心和历史帧的颜色，找到与 AABB 的交点。

```
1 vec3 aabbCenter = 0.5 * (aabbMin + aabbMax);
2 vec3 aabbHalf = 0.5 * (aabbMax - aabbMin);
3
4 vec3 dir = historyColor - aabbCenter;
5 vec3 unit_dir = dir / aabbHalf;
6 float max_comp = max(abs(unit_dir.x), max(abs(unit_dir.y), abs(unit_dir.z)));
7
8 vec3 result;
9 if (max_comp > 1.0) {
10     // 计算交点
11     result = aabbCenter + dir / max_comp;
12 } else {
13     // 在AABB内
14     result = historyColor;
15 }
```

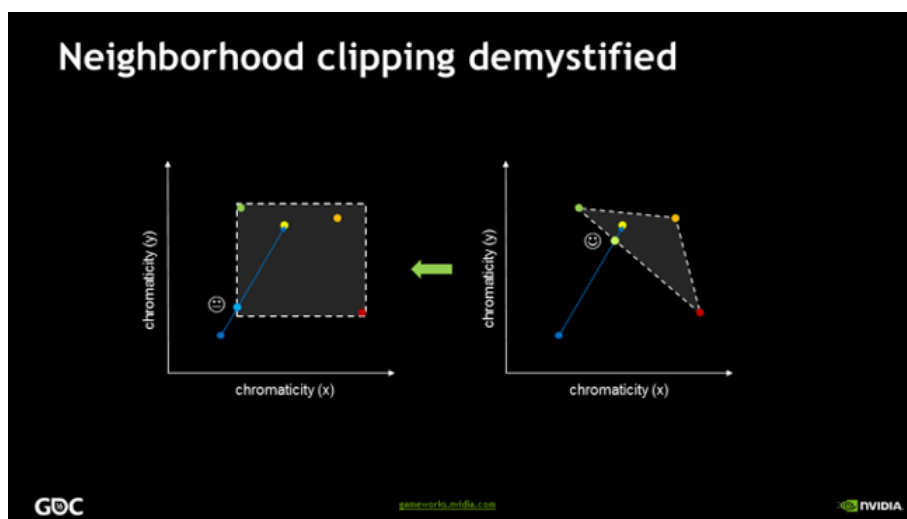


图 26: Neighbour Clipping

如图26所示，图中绿点、橙点、红点为当前邻域内的 sample，左下方的蓝色为历史 sample，由图可知和 AABB 的交点，与邻域内的 sample 差距较大，这种情况也会产生 ghosting 现象。NVIDIA 提出了 Variance Clipping

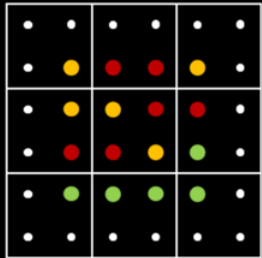
的方法。

Variance clipping

- Compute 1st and 2nd color moments
- AABB from mean μ and variance σ^2
 - $\mu \pm \gamma\sigma$

```
mu = m1 / N;  
sigma = sqrt(m2 / N - mu * mu);  
minc = mu - gamma * sigma;  
maxc = mu + gamma * sigma;
```
 - Scale down σ for reduced ghosting
 - $\gamma = 1$ works well
 - Can clip new AABB against old AABB

for all local samples..
m1 += color[i];
m2 += color[i] * color[i];



GDC gameworks.mdda.com NVIDIA

图 27: Variance Clipping

如图27所示，计算邻域内 sample 的均值 μ 和方差 σ^2 ，然后计算出 AABB。

7.6 Unity URP 的 TAA 解析

7.6.1 质量控制

URP 的 TAA 通过四个参数控制 TAA 的质量和性能：

(1) clampQuality

0 :Cross (5 taps)

1 :3 × 3 (9 taps)

2 :Variance + MinMax 3 × 3 (9 taps)

3 :Variance Clipping

(1.1) 0 表示只考虑当前像素和上下左右四个像素，用 Variance Clipping，计算均值和方差；

(1.2) 1 表示对 3×3 邻域计算均值和方差；

(1.3) 2 表示根据标准差调整 AABB，但用的是 clamp 策略；

```
1 if(clampQuality >= 2)
2 {
3     half perSample = 1 / half(9);
4     half3 mean = moment1 * perSample;
5     half3 stdDev = sqrt(abs(moment2 * perSample - mean * mean));
6
7     half devScale = _TaaVarianceClampScale;
8     half3 devMin = mean - devScale * stdDev;
9     half3 devMax = mean + devScale * stdDev;
10
11     // Ensure that the variance color box is
12     // not worse than simple neighborhood color box.
13     boxMin = max(boxMin, devMin);
14     boxMax = min(boxMax, devMax);
15 }
```

(1.4) 3 在 2 基础上，使用的是 clip 策略。

(2) motionQuality

0:None

1:5 taps

2:9 taps

(2.1) 什么都不做；

(2.2) 对当前像素和上下左右四个像素进行如下操作：

```
1 // bestDepth为采样 sample内的最小深度
2 // bestX, bestY为采样 sample内最小深度对应的像素位置
3 void AdjustBestDepthOffset(inout half bestDepth, inout half bestX, inout half bestY,
4     float2 uv, half currX, half currY)
5 {
6     // Half precision should be fine, as we are only concerned about choosing
```

```

7      // the better value along sharp edges, so it's
8      // acceptable to have banding on continuous surfaces
9      half depth = SAMPLE_TEXTURE2D_X(_CameraDepthTexture, sampler_PointClamp,
10         uv.xy + _BlitTexture_TexelSize.xy * half2(currX, currY)).r;
11
12  #if UNITY_REVERSED_Z
13      depth = 1.0 - depth;
14  #endif
15
16      bool isBest = depth < bestDepth;
17      bestDepth = isBest ? depth : bestDepth;
18      bestX = isBest ? currX : bestX;
19      bestY = isBest ? currY : bestY;
20  }

```

将 bestX 和 bestY 应用于 motion vector 的采样 uv;

(2.3) 对 3×3 邻域内进行采样, 步骤同 (2.2)。

(3) historyQuality

0 : *Bilinear*

1 : *Bilinear + discard history for UVs out of buffer*

2 : *Bicubic (5 taps)*

(3.1) 0 表示使用 bilinear 插值对历史帧进行插值采样;

(3.2) 1 表示应用双线性插值, 但是会丢弃超出历史帧的信息;

```

1  // Discard (some) history when outside of history buffer (e.g. camera jump)
2  // frameInfluence 为当前帧和历史帧数据的插值因子
3  half frameInfluence = ((historyQuality >= 1) && any(abs(uv - 0.5 + velocity) > 0.5)) ?
4      1 : _TaaFrameInfluence;

```

(3.3) 2 表示应用 bicubic 插值对历史帧进行插值采样。

(4) centralFiltering

0 : *direct sample*

1 : *filter color*

(4.1) 0 表示直接对当前像素进行采样；

(4.2) 1 表示对当前像素的 3×3 领域内进行滤波采样，权重根据相邻像素中心和 jitter 处的距离进行计算。（需要根据 jitter 更新权重）

7.6.2 质量等级

URP 的 TAA 分别为以下等级（数字表示四个质量参数的取值）：

(1) Very Low: (0, 0, 0, 0)（用 RGB 空间，而非 YCoCg 空间，性能考虑）

(2) Low: (0, 1, 1, 0)（用 RGB 空间，而非 YCoCg 空间，性能考虑）

(3) Medium: (2, 2, 1, 0)

(4) High: (2, 2, 2, 0)

(5) Very High:

(5.1) (2, 2, 2, 1)，在低精度下使用 clamp 而不是 clip，避免 flicker

(5.2) (3, 2, 2, 1)

7.7 参考

UE4 TAA

主流抗锯齿方案详解（二）TAA

TAA 原理及 OpenGL 实现

NVIDIA TAA

8 DLAA

深度学习抗锯齿（Deep Learning Anti-Aliasing, DLAA）是在 NVIDIA 提出的 DLSS 框架内的抗锯齿算法，DLSS 为深度学习超采样算法，DLSS 本身也有抗锯齿的效果，DLAA 对应超分比例为 1.0 的情况。

8.1 算法输入

DLAA 本身也是后处理的抗锯齿算法，需要提供以下输入：

- (1) color buffer (当前帧渲染结果)
- (2) depth buffer (当前帧的深度数据)
- (3) motion vector buffer (当前帧的运动向量数据)
- (4) jitter (与 TAA 的 jitter 相同)

8.2 个人使用记录

当前主流游戏使用的抗锯齿算法一般是 DLAA、TAA 和 MSAA，卡普空在生化 4 中应用过 TAA+FXAA 的策略（效果还可以）。

抗锯齿效果：DLAA > MSAA > TAA

性能：DLAA > TAA > MSAA

个人在自研引擎的开发工作中应用过 DLSS，主要是在用离线引擎渲染视频，应用超分功能提高渲染效率。测试过后，DLSS 不仅能大幅提高渲染效率，而且 DLSS 对帧间信息的处理可以缓解视频闪烁的问题。

DLSS 还有 Ray Reconstruction，将之应用于 1spp 渲染结果的降噪和超分，加上引擎对玻璃材质的优化，整体的视觉效果不错，在 RTX3080 上以 720p 作为输出分辨率，能达到 30fps。

8.3 参考

NVIDIA DLSS

Github DLSS

NVIDIA Streamline

9 参考

Real-Time Rendering, Fourth Edition