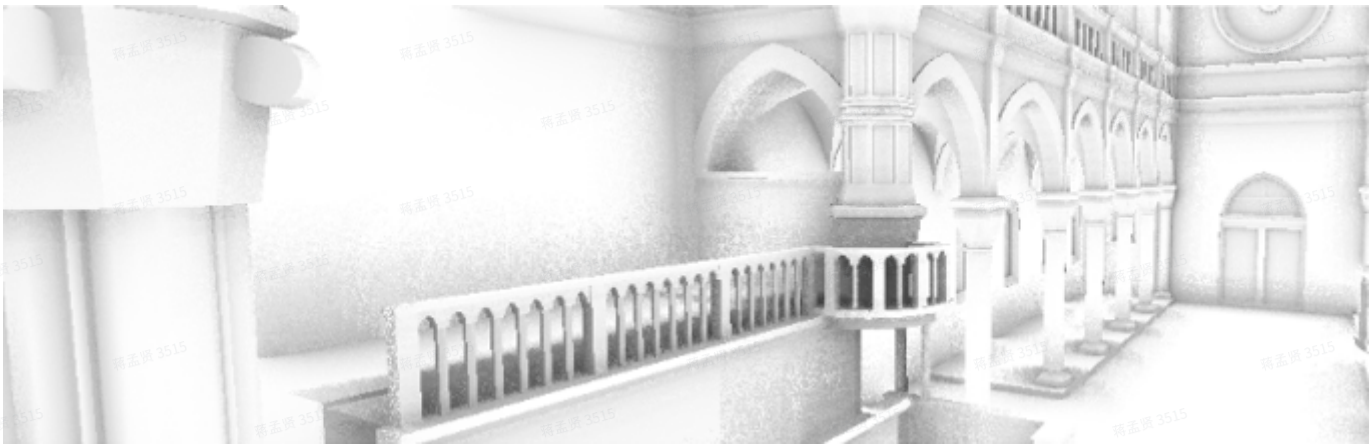
HBAO技术分享

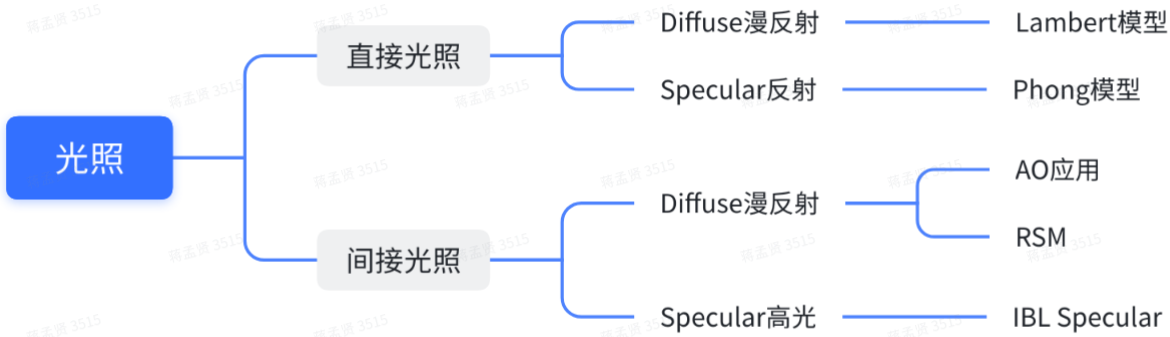
基于图像的Horizon-Based Ambient Occlusion



1. 背景

`SSAO` 相对离线领域肯定是做了极大的优化，但依然存在高昂的采样开销（一般采样次数：`64`）。而SS-HBAO（基于图像空间的Horizon-Based环境光遮蔽）则只需要采样 `20` 左右，并且效果还好于 `SSAO`，而在性能有限的移动端平台得到了广泛的使用。

> ✏️ AO：环境光遮蔽。用于计算间接光照的一部分，是对光照积分做 `split-sum` 的得到的衍生物之一，是对渲染点**宏观可见性**情况的估计



2. 算法分析

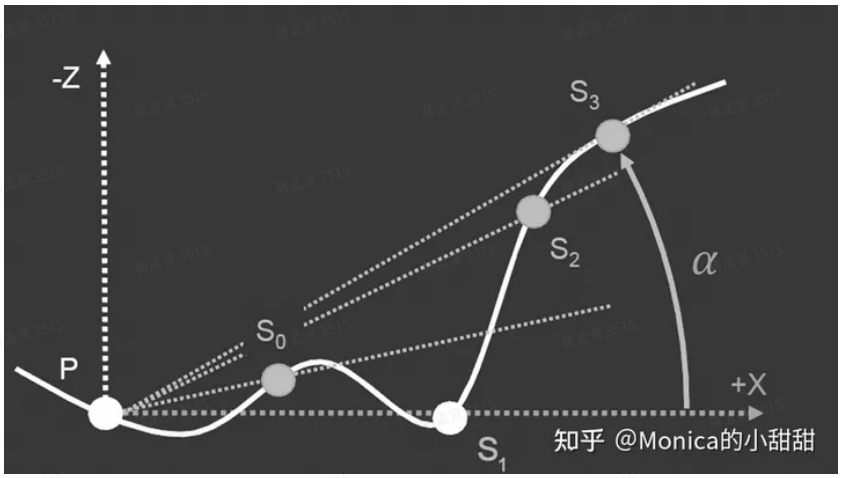
$$AO = 1 - \frac{1}{2\pi}\int_{\Omega} V(\vec{w})W(\vec{w})d\vec{w}$$

这是原论文的AO计算方程，其中：

- $V(\cdot)$：可见性函数。从渲染点发出一条射线，如果击中某个物体，返回`1`；逃逸出场景，则返回`0`
- $W(\cdot)$：线性衰减函数。离渲染点越近的遮挡点，其遮挡效果越强
- $\vec{w}$：从渲染点发出的射线
- Ω：采样域

我们以**当前渲染点** P（着色像素）为中心点，选取4个水平角度 θ_i，采样`4`个方向 $\vec{w}_i$。那么问题就是：针对每一个方向 $\vec{w}_i$，怎么计算 $V(\vec{w}_i)\cdot W(\vec{w}_i)$ 的值?

HBAO的巧思



HBAO 算法认为计算AO不需要这么麻烦。按照上图，我们可以得出一个简单的逻辑模型：垂直角 α 越大，说明这个像素点 P 周围被遮挡的概率越大——我们不需要计算可见性函数 $V(\cdot)$，而只需要获得最大的 α。如此，上诉公式变成了：

$$AO = 1 - \frac{1}{2\pi}\int_{\Omega} cos(\alpha)\cdot W(\vec{w})d\vec{w}$$

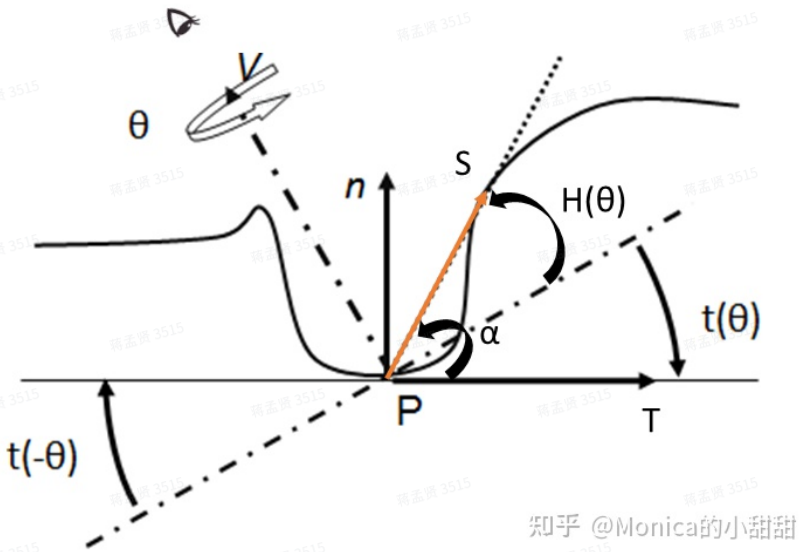
此外，距离因素也会影响遮挡的效果，因此衰减函数依旧保留，根据原论文，衰减函数如下：

$$W(w_i) = max(0, 1 - \frac{r(S_i)}{R})$$

> 其中：$r(S_i)$ 为采样点 S_i 到渲染点 P 的距离，R 是**最大采样距离**

如何计算最大的垂直角

在ScreenSpace发射N（一般为 4 ）条二维射线 $\vec{w_i}$，在每个射线方向<均匀+随机偏移>采样M（一般为 5 ）个领域像素点：S_i（$i = [0, M-1]$），**采样点的最大采样距离**被限制为 R。



HBAO 依然选择在Screen Space计算 α，根据上图可以知道：$\alpha = H(\theta) - t(\theta)$（减法：因为上图的 $t(\theta)$ 是负的——顺时针）。上图中，虚线代表的是View Space，向量 $\vec{T}$ 是渲染点 P 的切线。对之前的积分公式，进行展开可以得到如下公式：

$$
\begin{aligned}
AO &= 1 - \frac{1}{2\pi}\int_{\Omega} cos(\alpha)\cdot W(\vec{w})d\vec{w} \\
&= 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\int_{\alpha=t(\theta)}^{H(\theta)} cos(\alpha)\cdot W(\vec{w})d\vec{w} \\
&= 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}(sin(H(\theta)) - sin(t(\theta)))\cdot W(\vec{w})d\vec{w}
\end{aligned}
$$

这个时候，我们需要怎么计算？由初中数学可知：$sin = \frac{tan}{\sqrt{1+tan^2}}$，而对于一个向量 $\vec{V}(x,y,z)$ 来说，可以计算：$tan_V = \frac{z}{\sqrt{x^2+y^2}}$，因此我们的问题变成了：如何计算得到向量 $\vec{S}$ 和 $\vec{T}$ ——三维向量存在于三位空间，这个时候我们需要从一直操作的二维屏幕空间，移到三维空间，例如：View Space

计算S和T

1. 对于采样向量 $\vec{S}$，一个简单的思路：根据P点的UV和深度，S点的UV和深度，变换到View Space，然后相减得到结果

```
float ViewSpaceZFromDepth(float d)
{
    d = d * 2.0 - 1.0;
    return -(2.0 * u_Near * u_Far) / (u_Far + u_Near - d * (u_Far - u_Near));
}


// Maps standard viewport UV to an unprojected viewpos.
// Viewpos can then be achieved via out.xy / out.z
float3 ScreenToViewPos(float2 ViewportUV, float SceneDepth)
{
        float2 ProjViewPos;

        ProjViewPos.x =  ViewportUV.x * View.ScreenToViewSpace.x  + View.ScreenT
        ProjViewPos.y =  ViewportUV.y * View.ScreenToViewSpace.y  + View.ScreenT
        return float3(ProjViewPos * SceneDepth, SceneDepth);
}
//
vec3 UVToViewSpace(vec2 uv, float z)
{
    uv = uv * 2.0 - 1.0;
    uv.x = uv.x * tan(u_Fov / 2.0) * u_WindowWidth / u_WindowHeight  * z ;
    uv.y = uv.y * tan(u_Fov / 2.0)  * z ;
```

```
    return vec3(-uv, z);
}

vec3 GetViewPos(vec2 uv)
{
    float z = ViewSpaceZFromDepth(texture(u_DepthTexture, uv).r);
    return UVToViewSpace(uv, z);
}
```

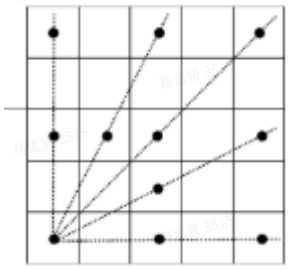
2. 对于切线 $\vec{T}$，使用偏移采样法，具体原理和计算方式见博客，也要依赖View Space

```
Pr = GetViewPos(TexCoord + vec2( 1.0 / u_WindowWidth, 0));
Pl = GetViewPos(TexCoord + vec2(-1.0 / u_WindowWidth, 0));
Pt = GetViewPos(TexCoord + vec2( 0, 1.0 / u_WindowHeight));
Pb = GetViewPos(TexCoord + vec2( 0, -1.0 / u_WindowHeight));
p1 = minDis(p, pr, pl); // 切线T
p2 = minDis(p, pt, pb); // 副切线B
// 但实际我们不关心是切线T，还是副切线B
// 我们关心的是和 S 关联的 T，以确保计算的统一性
// 何谓关联？ S在<p1,p2>平面的投影是T
// 同时注意：p1,p2和 P点的法线N构成了针对P点的局部坐标系
T = p1 * w_i.x + p2 * w_i.y
```

关于采样——均匀且随机

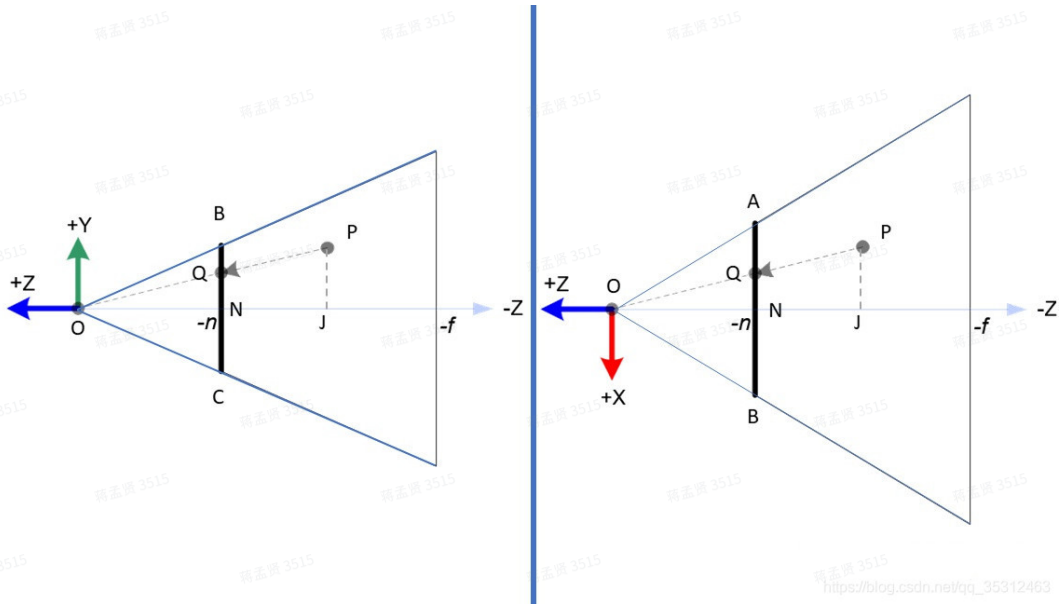
1. 射线的方向：$dir_i = rotate(\frac{2\pi}{dir\_num} \cdot i, \quad random\_offset\_angle)$

2. 采样点：每个方向之间使用不同的步长



关于UVToViewSpace

1. 实际上就是利用的三角形近似，具体参考博客



先看上面的左图，其中 BON 为1/2Fov角，根据几何关系得出：

$$\frac{PJ}{QN} = \frac{z_c}{n}$$

$$\frac{1}{V} = \frac{BN}{QN}$$

$$tan(fov/2) = \frac{BN}{n}$$

其中V为读取深度的v坐标。其中 PJ **就是我们要求的y值**，根据上面推导得出y为：

$$y_c = V * tan(fov/2) * z_c$$

注意这里的UV坐标是0-1的因此也需要像深度一样将其变换到-1到1才代入计算。

2.2.2.2 求x值

x的求法和y的求法类似，不过要注意 [公式] 跟Fov角没关系，因此还需要求 [公式] 的长度，而根据屏幕长宽可以得出：

$$\frac{height}{width} = \frac{BN}{AN}$$

$$tan(fov/2) = \frac{BN}{n}$$

如此一来就求出了 AN 的长度，因此得到x值为：

$$x_c = U * tan(fov/2) * width/height * z$$

至此就利用深度信息还原出了位置信息，不过需要注意的是上面求出的 ***(Xc,Yc,Zc)*** 为观察空间下的不是世界空间。

3. 总结

总结下目前的计算思路，并给出计算方式：

a. 对每个渲染像素 P ，在屏幕空间发射4条射线

i. 根据P的深度计算它的切线T，并计算sin(T)

b. 对其中的一条射线 $\vec{w_i}$ ，均匀但又带随机的采样5个点

i. 对于其中一个采样点 S_i ，计算 $\vec{PS_i}$ 向量，并获得其sin值

1. 选择其中最大的sin(S)值

ii. Ao += FallOFF(dis) * (sin(S) - sin(T))

c. Ao = 1 - Ao

UE4 SSAO

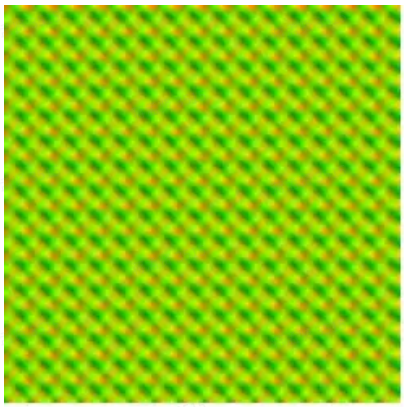
1. 算法和经典SSAO毫无相似，反而和HBAO非常相似

a. 均匀且随机的选择采样方向

i. SAMPLESET_ARRAY_SIZE： 方向数 （均匀）

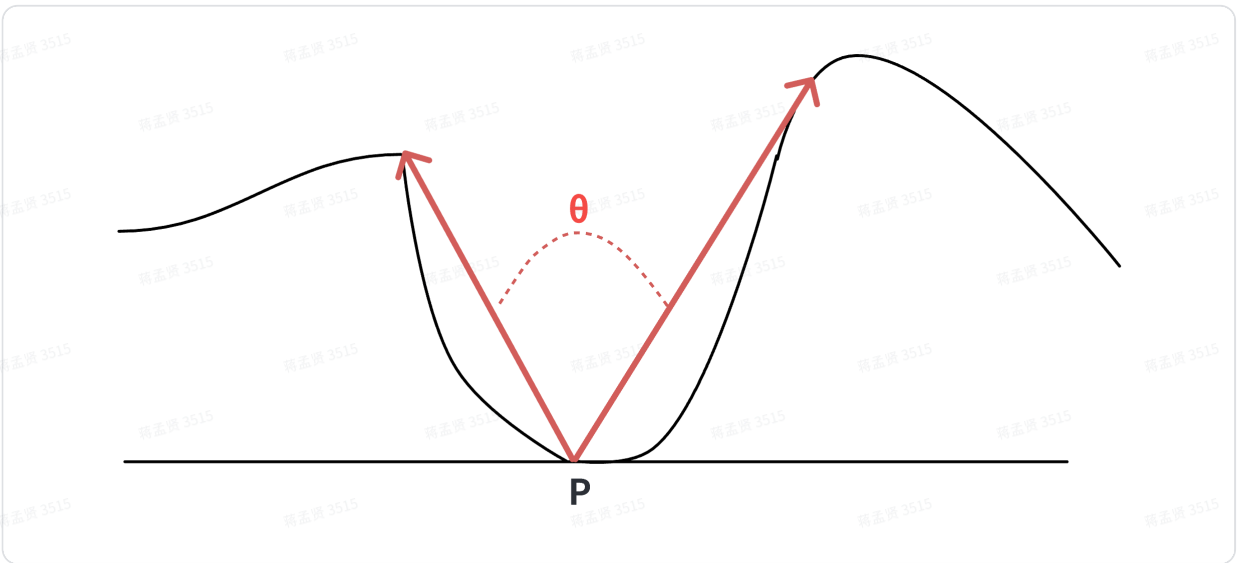
```
#elif USE_SAMPLESET == 2
    #define SAMPLESET_ARRAY_SIZE 5
    static const float2 OcclusionSamplesOffsets[SAMPLESET_ARRAY_SIZE]=
    {
        // 5 points distributed on a ring
        float2(0.156434, 0.987688),
        float2(0.987688, 0.156434)*0.9,
        float2(0.453990, -0.891007)*0.8,
        float2(-0.707107, -0.707107)*0.7,
        float2(-0.891006, 0.453991)*0.65,
    };
#else // USE_SAMPLESET == 3
    #define SAMPLESET_ARRAY_SIZE 6
    static const float2 OcclusionSamplesOffsets[SAMPLESET_ARRAY_SIZE]=
    {
        // 6 points distributed on the unit disc, spiral order and distanc
        float2(0.000, 0.200),
        float2(0.325, 0.101),
        float2(0.272, -0.396),
        float2(-0.385, -0.488),
        float2(-0.711, 0.274),
        float2(0.060, 0.900)
    };
#endif // USE_SAMPLESET
```

ii. RandomNormalTexture（随机）



b. `Step` 之间是均匀的

2. 具体计算AO的算法分为两种情况
   a. 法线版
      i. ToDo
   b. 深度版
      i. 没有比较深度
      ii. 而是和 `HBAO` 类似，找到最大的采样方向，返回最小的夹角 θ



      iii. 成对的思路和 `GTAO` 一样
      iv. 关于 `Weight` ，经验模型：离屏幕的距离，越远越重

```
// @return float2(InvNormAngle, Weight)
float2 WedgeNoNormal(float2 ScreenSpacePosCenter, float2 InLocalRandom, float3 I
{
    float2 ScreenSpacePosL = ScreenSpacePosCenter + InLocalRandom;
    float2 ScreenSpacePosR = ScreenSpacePosCenter - InLocalRandom;

    float AbsL = GetHZBDepth(ScreenSpacePosL, MipLevel);
```

```
    float AbsR = GetHZBDepth(ScreenSpacePosR, MipLevel);

    float3 SamplePositionL = ReconstructCSPos(AbsL, ScreenSpacePosL);
    float3 SamplePositionR = ReconstructCSPos(AbsR, ScreenSpacePosR);

    float3 DeltaL = (SamplePositionL - ViewSpacePosition) * InvFovFix;
    float3 DeltaR = (SamplePositionR - ViewSpacePosition) * InvFovFix;

    float WeightLeft;
    float3 SamplePositionLeft;
    {
        WeightLeft = 1;

#if !OPTIMIZATION_O1
        WeightLeft = saturate(1.0f - length(DeltaL) * InvHaloSize);
#endif
    }

    float WeightRight;
    float3 SamplePositionRight;
    {
        WeightRight = 1;

#if !OPTIMIZATION_O1
        WeightRight = saturate(1.0f - length(DeltaR) * InvHaloSize);
#endif
    }


    float FlatSurfaceBias = 5.0f;

    float left = ViewSpacePosition.z - AbsL;
    float right = ViewSpacePosition.z - AbsR;

    // OptionA: accurate angle computation
    float NormAngle = acosApproxNormAngle( dot(DeltaL, DeltaR) / sqrt(length2(De
    // OptionB(fade out in near distance): float NormAngle = acosApproxNormAngle
    // OptionC(look consistent but more noisy, should be much faster): float Nor


    // not 100% correct but simple
    // bias is needed to avoid flickering on almost perfectly flat surfaces
    //            if((leftAbs + rightAbs) * 0.5f > SceneDepth - 0.0001f)
    if(left + right < FlatSurfaceBias)
    {
        // fix concave case
        NormAngle = 1;
```

```
    }

    // to avoid halos around objects
    float Weight = 1;

    float InvAmbientOcclusionDistance = ScreenSpaceAOParams[0].z;
    float ViewDepthAdd = 1.0f - ViewSpacePosition.z * InvAmbientOcclusionDistanc

    Weight *= saturate(SamplePositionL.z * InvAmbientOcclusionDistance + ViewDep
    Weight *= saturate(SamplePositionR.z * InvAmbientOcclusionDistance + ViewDep

//        return float2(1 - NormAngle, (WeightLeft + WeightRight) * 0.5f);
    return float2((1-NormAngle) / (Weight + 0.001f), Weight);
}
```

UE4 GTAO

1. `Engine\Shaders\Private\PostProcessAmbientOcclusion.usf`

```
float2 SearchForLargestAngleDual(uint NumSteps, float2 BaseUV, float2 ScreenDir,
{
    float SceneDepth, LenSq, OOLen, Ang, FallOff;
    float3 V;
    float2 SceneDepths =0;

    float2 BestAng = float2(-1,-1);
    float Thickness = GTAOParams[1].y;

    for(uint i=0; i<NumSteps; i++)
    {
        float fi                = (float) i;

        float2 UVOffset  = ScreenDir * max( SearchRadius * (fi + InitialOffset),
        UVOffset.y                *= -1;
        float4 UV2                 = BaseUV.xyxy + float4( UVOffset.xy, -UVOffse

        // Positive Direction
        SceneDepths.x        = ConvertFromDeviceZ(LookupDeviceZ(UV2.xy));
        SceneDepths.y        = ConvertFromDeviceZ(LookupDeviceZ(UV2.zw));

        V                              = ScreenToViewPos(UV2.xy, SceneDepths.x
        LenSq                    = dot(V,V);
        OOLen                    = rsqrt(LenSq + 0.0001);
        Ang                          = dot(V,ViewDir) * OOLen;
```

```
26
27          FallOff           = saturate(LenSq * AttenFactor);
28          Ang                         = lerp(Ang, BestAng.x, FallOff);
29          BestAng.x  = ( Ang > BestAng.x ) ? Ang : lerp( Ang, BestAng.x, Thickness
30
31          // Negative Direction
32          V                           = ScreenToViewPos(UV2.zw, SceneDepths.y) - View
33          LenSq                = dot(V,V);
34          OOLen                = rsqrt(LenSq + 0.0001);
35          Ang                         = dot(V,ViewDir) * OOLen;
36
37          FallOff           = saturate(LenSq * AttenFactor);
38          Ang                         = lerp(Ang, BestAng.y, FallOff);
39
40          BestAng.y  = ( Ang > BestAng.y ) ? Ang : lerp( Ang, BestAng.y, Thickness
41      }
42
43      BestAng.x = acosFast(clamp(BestAng.x, -1.0,  1.0));
44      BestAng.y = acosFast(clamp(BestAng.y, -1.0,  1.0));
45
46      return BestAng;
47  }
```

2. 具体分析： UE4的GTAO简析