# スクリーンスペース・アンビエント・オクルージョン (SSAO)

mebiusbox software

2019 年 6 月 27 日

# 目次

## 1．はじめに

一般的に私たちの眼に映る光景には陰影があります．これは光源から放出される光の粒子（フォトン）が物体に反射されて眼に入射してくる量によって明るさが変わります．本来，陰になっている場所はそこに到達する光の量が少ないからで，光源の方向や距離，周辺の遮蔽物によって変わってきます．コンピュータ・グラフィックスではレイトレーシング法を使って光の伝達経路を計算することで物体に到達する光量を正確に求めることが出来ますが，正確になればなるほど計算量が膨大になり，レンダリングにとても時間がかかってしまいます．特に光源からの直接な光よりも，遮蔽物や周りの物体からの反射による照明が重要になってきます．このような間接的な光の照明を**グローバル・イルミネーション**（GI）といいます．リアルタイムのような計算時間に制限のある場合では，直接光や間接光による照明は大胆な近似で計算しており，また，遮蔽物による光量の減衰を計算して擬似的に陰を作り出しています．ここで，物体のある 1 点に注目し，そこに到達する光量を考えてみます．この物体は透過しないとします．この点に入ってくる光量は表面の半球内のあらゆる方向から入射してくる光量を合わせたものです．光源や別の物体（反射光）との間に遮蔽物があれば，到達する光の量が減少します．このように周辺の遮蔽物によって到達する光量がどの程度減衰するかを表したものを**アンビエント・オクルージョン**といいます．アンビエント・オクルージョンは一般的にレイトレーシング法を使って計算するのですが，正確になればなるほど時間がかかってしまいます．リアルタイムレンダリングにおいて，アンビエント・オクルージョンは計算に時間がかかってしまうため，事前に計算しておいたアンビエント・オクルージョンをテクスチャにして参照しています．この方法は現在でも使われていますが，動的なシーンに対応していない，テクスチャの作成コストや容量の問題もあります．しかし 2007 年 Crytek が Crysis というゲームで**スクリーンスペース・アンビエント・オクルージョン**（Screen-Space Ambient Occlusion: SSAO）というリアルタイムにアンビエント・オクルージョンを計算する手法を開発しました．この手法は非常に高速で動作し，その後，様々な改良版が開発されていきました．現在では SSAO は標準で入っているぐらいに一般的な方法となっています．ここでは，スクリーンスペース・アンビエント・オクルージョンについてメモ書き程度にまとめたもので，基本的なアルゴリズムと NVIDIA が開発した **Horizon-Based Ambient Occlusion**: HBAO について書いたものです．本記事では基本的なアルゴリズムによるスクリーンスペース・アンビエント・オクルージョンのことを SSAO と表記します．

## 2．基本アルゴリズム

一度シーンをレンダリングしたあとに，画面全体に対して処理をするポストプロセスを行います．準備として，シーンをレンダリングするときに深度と法線を出力します．法線は深度を使って復元することが出来ますが，その分計算負荷が高くなります．各ピクセルにおけるカメラ座標系での座標を深度から復元し，復元した位置からランダムにオフセットした位置の理想の深度と実際の深度を比較して，実際の深度が小さい（手前にある）場合は遮蔽されているとします．このとき，サンプリング数が多いほど遮蔽の精度が上がりますが，計算負荷が高くなります．

周辺をサンプリングする位置はオリジナルの Crysis による方法では球状の領域をランダムサンプリングします．

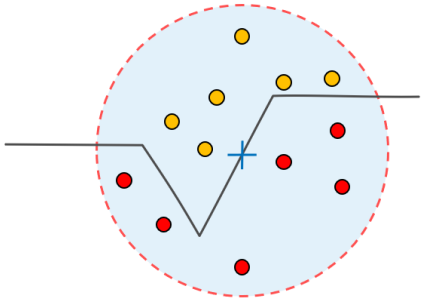

**Fig.1:** 球状サンプリング

次の図ではこの手法で計算した遮蔽を表示したものです．遮蔽されているほど黒くなっています．

**Fig.2:** Martin Mittring「Finding Next Gen - CryEngine 2」より引用

それっぽい陰影が出来ていることがわかりますが，全体的に陰が出来てしまっています．

そこで，サンプリングする範囲を球状ではなく，対象点の法線方向を中心とした半球領域にします．

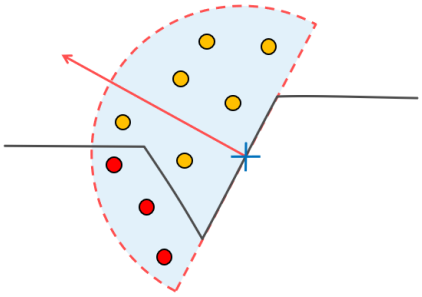

**Fig.3:** 半球サンプリング

また，ランダムサンプリングでは均一ではなく，対象点付近に集中するような分布に変更することで精度を上げます．

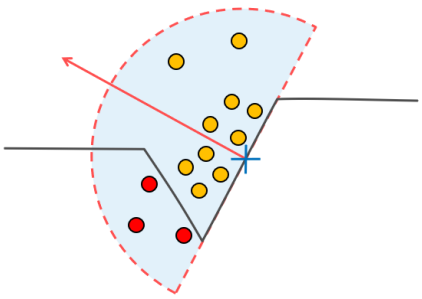

**Fig.4:** 非一様の半球サンプリング

サンプル数が足りないとバンディングやノイズのようなアーティファクトが発生します.

**Fig.5:** バンディングやノイズのようなアーティファクト

サンプル数を上げるとアーティファクトを軽減することが出来ますが，処理負荷が高くなってしまうため，対処方法としてブラーをかけます.

**Fig.6:** ブラー適用

また，別の問題として，周辺の遮蔽物がないところにも遮蔽が発生してしまうことがあります．これは物体の境界における周辺の深度差が大きく，本来はほとんど影響しない遮蔽物による遮蔽が発生してしまっているからです．このため，深度の差分を比較するときに遮蔽として影響する距離内かどうかをチェックします．たとえば次のようなシーンの場合

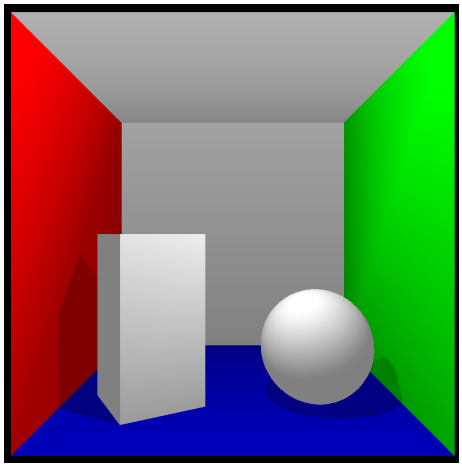

**Fig.7:** サンプルシーン

この遮蔽は次のようになって，赤枠の部分に強い陰影が出来ています.

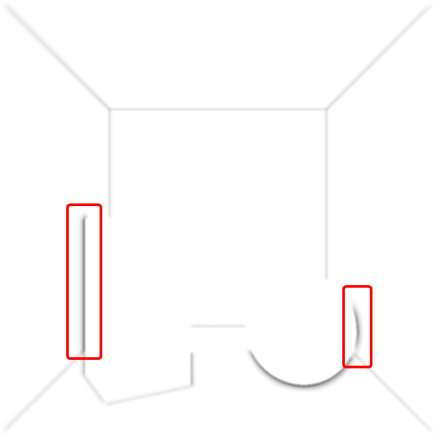

**Fig.8:** 不要な陰影

ここで，適切な距離を設定して調整します．

**Fig.9:** 距離を調整後

このランダムサンプリングの部分を改善することで，陰影の品質が向上します．また，深度バッファの精度も問題となる場合があり，Z ファイティングが発生して意図した結果にならないこともあります．その場合，たとえば半球のサンプリング領域から，さらに表面に近い位置（赤い領域）を無視するといった対応が考えられます．

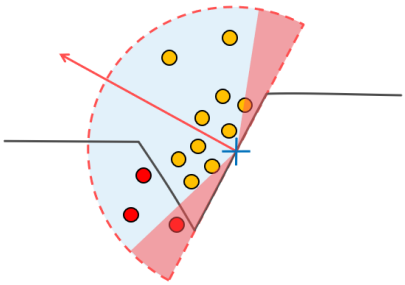

**Fig.10:** 半球サンプリング領域の調整

## 3． 実装

今回は Three.js を使って実装を行います．ここでは，主にフラグメントシェーダの実装コードについて説明していきます．Three.js や頂点シェーダに関する説明は省きます．

レンダリングではカラーバッファ，深度バッファ，法線バッファ，AO バッファ，ブラーバッファ，合成バッファのレンダーターゲットを作成します．最終的に合成バッファの内容をスクリーンに表示しています．
処理としては次の順番になります．

- カラーバッファと深度バッファ，法線バッファに描画
- AO バッファに遮蔽係数を描画
- AO バッファの内容にブラーをかけてブラーバッファに描画
- カラーバッファとブラーバッファを合成して合成バッファに描画
- 合成バッファをスクリーンに描画

## 3.1 カラーバッファと深度バッファ，法線バッファに描画

Three.js の機能を使って普通に描画します．深度バッファは後で使いますので，深度バッファを作成して設定します．

```
var colorTarget = createRenderTarget(THREE.RGBFormat);
colorTarget.depthBuffer = true;
colorTarget.depthTexture = new THREE.DepthTexture();
```

精度を 32bit にする場合は **THREE.UnsignedIntType** を指定します．

```
var colorTarget = createRenderTarget(THREE.RGBFormat);
colorTarget.depthBuffer = true;
colorTarget.depthTexture = new THREE.DepthTexture();
colorTarget.depthTexture.type = THREE.UnsignedIntType;
```

法線バッファへの描画は Three.js の **THREE.MeshNormalMaterial()** をシーンのオーバーライドマテリアルに設定しています．

## 3.2 AO バッファに遮蔽係数を描画

深度バッファ，法線バッファを使って遮蔽係数を計算します．まず，深度バッファから線形深度やカメラ座標系の Z 値を計算する関数が定義されています．詳しくは「３次元座標変換のメモ書き」の付録を参照してください．
次にスクリーン座標と深度値を使ってカメラ座標系での位置を復元する ‘getViewPosition‘関数があります．

```
vec3 getViewPosition(vec2 screenPosition, float depth, float viewZ) {
    float clipW = CameraProjectionMatrix[2][3] * viewZ + CameraProjectionMatrix[3][3];
    vec4 clipPosition = vec4((vec3(screenPosition, depth) - 0.5)*2.0, 1.0);
    clipPosition *= clipW; // unprojection
    return (CameraInverseProjectionMatrix * clipPosition).xyz;
}
```

これはクリップ座標系の XYZ を求めて，透視投影行列の逆行列を使ってカメラ座標系に変換しています．
次に法線ですが，法線バッファは RGB フォーマットで $[0,1]$ の値が格納できるため，法線を出力時に

```
gl_FragColor = vec4((normal.xyz+1.0)*0.5, 1.0);
```

として $[-1,1]$ を $[0,1]$ に写像しています．そのため，テクスチャから参照するときに，復元する必要があります．

```
vec3 getViewNormal(vec2 screenPosition) {
    vec3 rgb = texture2D(NormalSampler, screenPosition).xyz;
    return 2.0*rgb.xyz - 1.0;
}
```

まずは処理しているピクセルのカメラ座標系での座標を計算します．

```
float depth = getDepth(vUv);
float viewZ = getViewZ(depth);

vec3 viewPosition = getViewPosition(vUv, depth, viewZ);
vec3 viewNormal   = getViewNormal(vUv);
```

次に半球状にランダムサンプリングするのですが，サンプリング数を減少させるために，ノイズテクスチャによってランダムな方向に回転させています．回転した法線からグラム・シュミットの直交化法を使って正規直交基底を計算します．グラム・シュミットの直交化法については「CG のための線形代数入門 ベクトル編」などを参照してください．また，ノイズテクスチャは半球状の分布した値が格納されています．

```
vec2 noiseScale = vec2(Resolution.x / 4.0, Resolution.y / 4.0);
vec3 random = texture2D(NoiseSampler, vUv * noiseScale).xyz;

vec3 tangent = normalize(random - viewNormal * dot(random,viewNormal));
vec3 bitangent = cross(viewNormal, tangent);
mat3 kernelMatrix = mat3(tangent, bitangent, viewNormal);
```

次にランダムサンプリングして遮蔽係数 ( **occlusion** ) を求めます．

```
float occlusion = 0.0;
for (int i=0; i<KERNEL_SIZE; i++) {
    vec3 sampleVector = kernelMatrix * Kernel[i];
    vec3 samplePoint = viewPosition + (sampleVector * KernelRadius);
    vec4 samplePointNDC = CameraProjectionMatrix * vec4(samplePoint, 1.0);
    samplePointNDC /= samplePointNDC.w;

    vec2 samplePointUv = samplePointNDC.xy * 0.5 + 0.5;
    float realDepth = getLinearDepth(samplePointUv);
    float sampleDepth = viewZToOrthographicDepth(samplePoint.z);
    float delta = sampleDepth - realDepth;
    if (delta > MinDistance && delta < MaxDistance) {
        occlusion += 1.0;
    }
}
```

**kernelRadius** は半球サンプリング領域の半径です． **MinDistance** と **MaxDistance** は深度の差分の範囲です．この範囲外では遮蔽に影響しません． **sampleVector** はオフセット方向で，それに **kernelRadius** を掛けた分をオフセット移動しています．オフセット移動した座標を透視変換した深度と，実際のシーンに書かれた最前面の深度を比較し，遮蔽と判断された場合は **occlusion** をインクリメントしています．

最後に単純な算術平均を計算します．この値は遮蔽されているほど高い値になります．合成時にカラーバッファの値に掛けられる値になりますので， **1.0 - occlusion** として出力します．また， **Strength** で遮蔽の強さを調整できるようにしています．

```
occlusion = clamp(occlusion / float(KERNEL_SIZE), 0.0, 1.0);
gl_FragColor = vec4(vec3(1.0 - occlusion * Strength), 1.0);
```

## 3.3 AO バッファの内容にブラーをかけてブラーバッファに描画

ここでは，単純な 5x5 の平均を求めるブラーをかけます．

```
void main()
{
    vec2 texelSize = (1.0 / Resolution);
    float result = 0.0;

    for (int i=-2; i<=2; i++) {
        for (int j=-2; j<=2; j++) {
            vec2 offset = ((vec2(float(i),float(j)))*texelSize);
            result += texture2D(ColorSampler, vUv+offset).r;
        }
    }

    gl_FragColor = vec4(vec3(result / (5.0*5.0)), 1.0);
}
```

## 3.4 カラーバッファとブラーバッファを合成して合成バッファに描画

カラーバッファの内容に対してブラーをかけた遮蔽係数を掛けたものを出力します．ここで，遮蔽係数の指数を計算して調整できるようになっています．

```
void main() {
    vec4 color = texture2D(ColorSampler, vUv);
    float occlusion = pow(texture2D(OcclusionSampler, vUv).x, OcclusionPower);
    gl_FragColor = vec4(mix(OcclusionColor, color.xyz, occlusion), 1.0);
}
```

## 3.5 合成バッファをスクリーンに描画

単純に合成バッファの内容をそのまま描画しています．

## 3.6 デモ

このアルゴリズムによる結果は次のようになります．

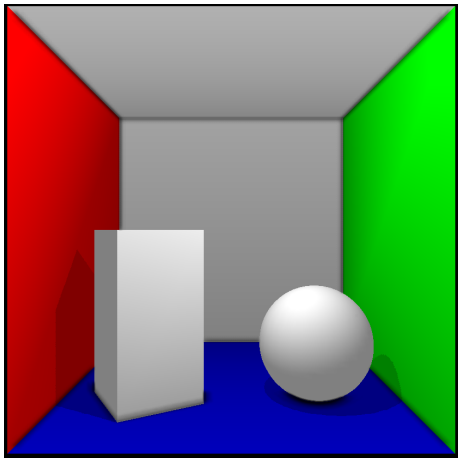

**Fig.11**

隠蔽係数を表示したものは次のようになっています．

**Fig.12**

実際に動作するデモを用意しました.

SSAO Demo

## 4． Horizon-Based Ambient Occlusion (HBAO)

NVIDIA が発表した SSAO を改良した手法です．これまでの内容から大きな変更として，ランダムサンプリング時のアルゴリズムが変わったこと，ブラー処理にバイラテラルフィルターを採用したことです.

### 4.1 Horizon-Based

SSAO では半球状にランダムサンプリングでした．HBAO では対象の点からスクリーン上で全方位にランダムでレイを飛ばします．レイは少しずづ進めていくレイマーチングで，レイを進めるたびにその位置の深度から，地平線ベクトルとの角度を求めて遮蔽係数を求めます．そのため，地平線ベース（Horizon-Based）と呼ばれています.

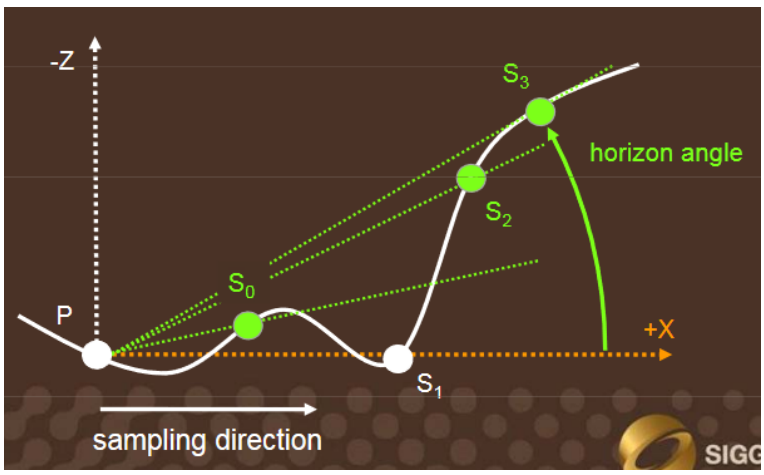


**Fig.13:** Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」NVIDIA から引用

実際に遮蔽係数は対象点における接ベクトルと，地平線ベクトルとの角度を求めます．この角度と，レイマーチングによって進めていった位置と地平線ベクトルとの角度から次のような計算をします.

$$ao = \sin(h) - \sin(t)$$

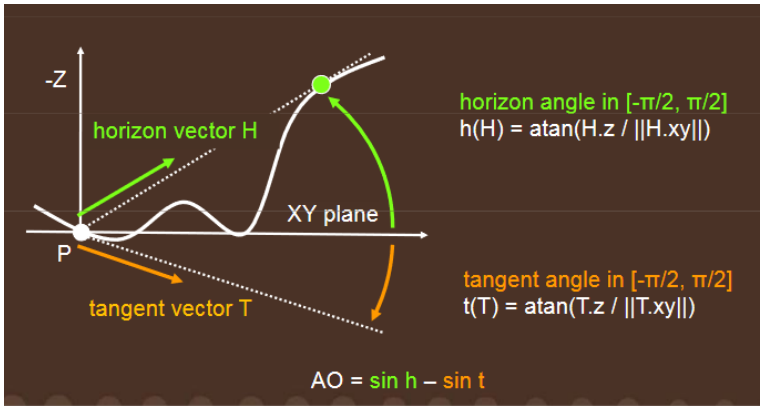


**Fig.14:** Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」NVIDIA から引用

また，対象点と距離が離れるほど遮蔽の影響を弱くします．ここで，単純な線形で減衰しないように調整します．対象点を $P$，レイによるサンプリング点を $S$ とすると

$$r = ||S - P|| / R$$

$$W(r) = 1 - r^2$$

と計算します．この値を遮蔽係数に乗算します．

$$ao = W(r) \cdot (\sin(h) - \sin(t))$$

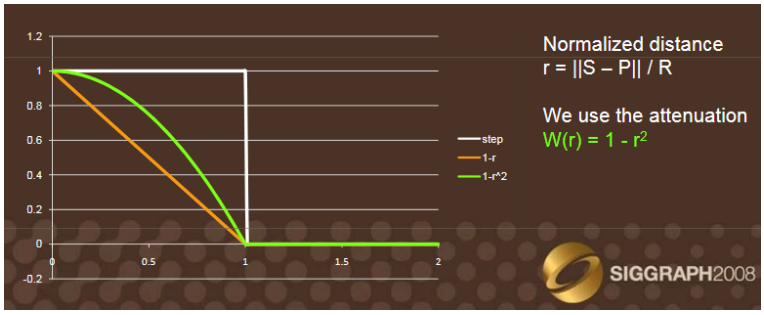


**Fig.15:** Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」NVIDIA から引用

ポリゴンの分割数が少ないと隣接した面がなめらかではなく，エッジ部分に意図しない遮蔽が発生してしまいます．そのため，接ベクトル付近は無視するようにします．具体的にはバイアスを加えて調整します．

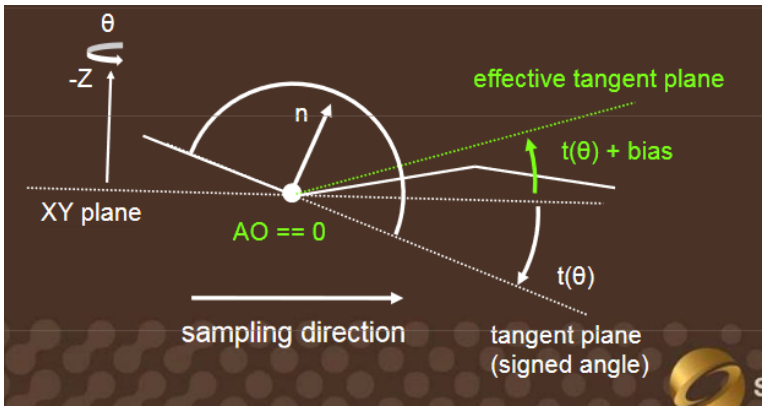


**Fig.16:** Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」NVIDIA から引用

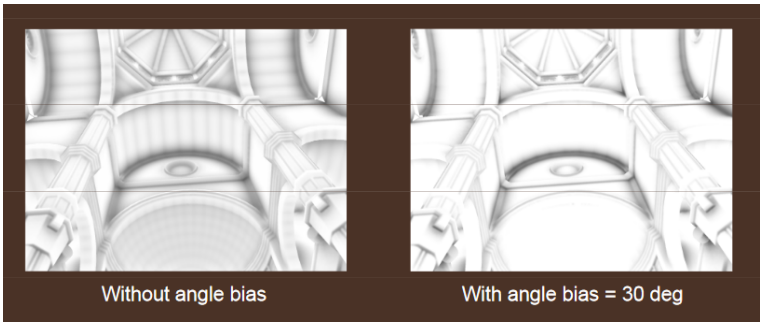


**Fig.17:** Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」NVIDIA から引用

## 4.2 バイラテラルフィルタ

ブラー処理には２パスのバイラテラルフィルタを使用します．バイラテラルフィルタは重み付きブラー処理で，隣接の深度との差分によって重みを計算します．これによって，遮蔽係数が大きい付近に強くブラーがかかるようになります．

## 4.3 実装

基本的な処理の流れは SSAO と変わりません．遮蔽係数を計算し，ブラーをかけて合成します．まず，UV 値からカメラ座標系への変換ですが，SSAO デモとは違う実装になっています．

```
vec3 P = viewPos(uv);
```

**viewPos** 関数は UV 座標をカメラ空間の座標に変換します．この実装は次のようになっています．

```
vec3 viewPos(vec2 uv)
{
    float depth = texture2D(DepthSampler, uv).x;
    float viewZ = perspectiveDepthToViewZ(depth, near, far);
    return uvToView(uv, viewZ);
}
```

また， **uvToView** 関数は次のようになっています．

```
vec3 uvToView(vec2 uv, float viewZ)
{
    uv = UvToViewParams.xy * uv + UvToViewParams.zw;
    return vec3(uv*viewZ, viewZ);
}
```

この関数を満たす **UvToViewParams** の各要素を導出してみます．まず，透視変換行列は画角とアスペクト比から計算しているので，

$$M_{perspective} = \begin{pmatrix} \frac{F}{aspect} & 0 & 0 & 0 \\ 0 & F & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

ここで $F$ と $aspect$ は次のとおりです．

$$F = \frac{1}{\tan(fov/2)} = \cot\left(\frac{fov}{2}\right), \qquad aspect = \frac{W}{H}$$

カメラ座標系の座標を $xyz_{view}$，クリップ座標系の座標を $xyz_{clip}$，正規化デバイス座標系の座標を $xyz_{ndc}$ とおくと，カメラ座標系の X 座標から UV 座標の $u$ に変換する式は次のようになります．

$$x_{clip} = x_{view} \cdot F \cdot \text{aspect}^{-1}$$

$$x_{ndc} = \frac{x_{clip}}{-z_{view}}$$

$$u = (x_{ndc} + 1) \cdot \frac{1}{2}$$

よって，UV 座標からカメラ座標に変換するにはこれの逆演算をすればよいことになります．

$$x_{ndc} = 2u - 1$$

$$x_{clip} = x_{ndc} \cdot -z_{view}$$

$$x_{view} = \frac{x_{clip}}{F \cdot \text{aspect}^{-1}}$$

整理すると次のようになります．

$$\begin{aligned} x_{view} &= \frac{(2u-1) \cdot -z_{view}}{F \cdot \text{aspect}^{-1}} \\ &= \frac{2u \cdot -z_{view}}{F \cdot \text{aspect}^{-1}} + \frac{-1 \cdot -z_{view}}{F \cdot \text{aspect}^{-1}} \\ &= \left( \frac{2u}{F \cdot \text{aspect}^{-1}} + \frac{-1}{F \cdot \text{aspect}^{-1}} \right) \cdot (-z_{view}) \\ &= \left( \frac{-2u}{F \cdot \text{aspect}^{-1}} + \frac{1}{F \cdot \text{aspect}^{-1}} \right) \cdot z_{view} \end{aligned}$$

$y$ も同様に次のようになります．

$$\begin{aligned} y_{view} &= \frac{(2v-1) \cdot -z_{view}}{F} \\ &= \frac{2v \cdot -z_{view}}{F} + \frac{-1 \cdot -z_{view}}{F} \\ &= \left( \frac{2v}{F} + \frac{-1}{F} \right) \cdot (-z_{view}) \\ &= \left( \frac{-2v}{F} + \frac{1}{F} \right) \cdot z_{view} \end{aligned}$$

$\text{aspect}^{-1} = h/w$ ですから， **UvToViewParams** の各要素は次のようになっています．

```
var focal1 = 1.0 / Math.tan(THREE.Math.degToRad(fov * 0.5)) * (Height / Width);
var focal2 = 1.0 / Math.tan(THREE.Math.degToRad(fov * 0.5));
var invFocal1 = 1.0 / focal1;
var invFocal2 = 1.0 / focal2;
var uvToVA0 = -2.0 * invFocal1;
var uvToVA1 = -2.0 * invFocal2;
var uvToVB0 = 1.0 * invFocal1;
var uvToVB1 = 1.0 * invFocal2;
UvToViewParams = new THREE.Vector4(uvToVA0, uvToVA1, uvToVB0, uvToVB1);
```

次に，スクリーン上におけるサンプリング領域の長さを求めます．

```
float diskRadiusInUV = 0.5 * R * FocalLength / -P.z;
float radiusInPixels = diskRadiusInUV * AORes.x;
```

ここで， **R** はカメラ座標系でのサンプリング領域の半径で， **FocalLength** は

$$FocalLength = \frac{1}{\tan(fov/2)} \cdot \frac{ResY}{ResX}$$

となっています． **AORes** は AO バッファのサイズです．

次に接ベクトルを計算するための基底ベクトルを計算します．これはカメラ座標系での座標から勾配を求めて計算します．

```
vec3 Pr = viewPos(uv + vec2( InvAORes.x, 0));
vec3 Pl = viewPos(uv + vec2(-InvAORes.x, 0));
vec3 Pt = viewPos(uv + vec2(0,  InvAORes.y));
vec3 Pb = viewPos(uv + vec2(0, -InvAORes.y));
vec3 dPdu = minDiff(P, Pr, Pl);
vec3 dPdv = minDiff(P, Pt, Pb) * (AORes.y * InvAORes.x);
```

ここで **InvAORes** は **AORes** の逆数です． **dPdu** と **dPdv** を使って次のように接ベクトルを計算します．

```
vec3 T = deltaUV.x * dPdu + deltaUV.y * dPdv;
```

HBAO では，ランダムな方向にレイを飛ばします．また，レイを少しずつ進めるレイマーチング法を行っています．このレイの数と，レイを飛ばす距離をどれだけ細かく区切るかの値を指定します．レイの数は **NUM_DIRECTION** で固定です．区切る数は最大数 **NUM_STEPS** を決めておき，その値以上にならないように，また，１ステップ進めるごとにスクリーン上で１ピクセル以上進むように調整します．それが **calculateNumSteps** 関数です．

```
void calculateNumSteps(inout vec2 stepSizeInUV,
                       inout float numSteps,
                       float radiusInPixels,
                       float rand)
{
    float MaxRadiusPixels = RadiusParams.w;
    vec2  InvAORes = ScreenParams.zw;

    numSteps = min(float(NUM_STEPS), radiusInPixels);

    float stepSizeInPixels = radiusInPixels / (numSteps + 1.0);

    float maxNumSteps = MaxRadiusPixels / stepSizeInPixels;
    if (maxNumSteps < numSteps)
    {
        numSteps = floor(maxNumSteps + rand);
        numSteps = max(numSteps, 1.0);
        stepSizeInPixels = MaxRadiusPixels / numSteps;
    }

    stepSizeInUV = stepSizeInPixels * InvAORes.xy;
}
```

ここで **MaxRadiusPixels** はスクリーン上での最大半径（単位がピクセル）です．次に，レイごとに遮蔽係数を計算していきます．

```
for (int d=0; d<NUM_DIRECTIONS; d++) {
    // Apply noise to the direction
```

```
    float angle = alpha * float(d);
    vec2 dir = rotateDirections(vec2(cos(angle), sin(angle)), rand.xy);
    vec2 deltaUV = dir * stepSize.xy;
    vec2 texelDeltaUV = dir * InvAORes.xy;
    ao += calculateHorizonOcclusion(deltaUV, texelDeltaUV, uv, P, numSteps, rand.z, dPdu, dPdv);
}

ao = 1.0 - ao / float(NUM_DIRECTIONS) * AOStrength;
```

**rotateDirections** はランダムな方向に回転したレイの向きを計算します．各方向ごとに **calculateHorizonOcclusion** 関数を呼び出して遮蔽係数の総和を求め，算術平均を取ります．ここで， **calculateHorizonOcclusion** 関数は次のようになっています．

```
float calculateHorizonOcclusion(vec2 dUv,
                                vec2 texelDeltaUV,
                                vec2 uv0,
                                vec3 P,
                                float numSteps,
                                float randstep,
                                vec3 dPdu,
                                vec3 dPdv)
{
    float ao = 0.0;

    vec2 uv = uv0 + snapUVOffset(randstep * dUv);
    vec2 deltaUV = snapUVOffset(dUv);
    vec3 T = deltaUV.x * dPdu + deltaUV.y * dPdv;

    float tanH = getBiasedTangent(T);

    vec2 snapped_duv = snapUVOffset(randstep * deltaUV + texelDeltaUV);
    ao = integrateOcclusion(uv0, snapped_duv, P, dPdu, dPdv, tanH);
    --numSteps;

    float sinH = TanToSin(tanH);
    for (int j=1; j<MAX_STEPS; ++j)
    {
        if (float(j) >= numSteps)
        {
            break;
        }

        uv += deltaUV;
        vec3 S = viewPos(uv);
        vec3 diff = S - P;
        float tanS = getTangent(diff);
        float d2 = lengthSqr(diff);
        float R2 = RadiusParams.y; // R*R
        if ((d2 < R2) && (tanS > tanH))
        {
            // Accumulate AO between the horizon and the sample
            float sinS = TanToSin(tanS);
            ao += falloffFactor(d2) * saturate(sinS - sinH);

            // Update the current horizon angle
            tanH = tanS;
            sinH = sinS;
        }
    }

    return ao;
}
```

**snapUVOffset** はテクセルの中心位置を揃えています．これはテクスチャ参照時にフィルタリングによって値が変動してしまうため，スクリーン上のピクセルを参照時に必ず同じ値になるように調整します．

```
vec2 snapUVOffset(vec2 uv)
{
    return round(uv*AORes.xy) * InvAORes.xy;
}
```

**getBiasedTangent()** はバイアスされたタンジェントを求める関数で次のようになっています．

```
float getBiasedTangent(vec3 T)
{
    return T.z * rsqrt(dot(T.xy,T.xy)) + TanAngleBias;
}
```

**TanToSin** 関数はタンジェントからサインの値を計算します．三角関数は他の関数と比べて処理負荷が高いため，なるべく呼ばないようにします．ここではタンジェントの値が $[-\pi/2, \pi/2]$ の範囲内なので，

$$\sin(x) = \frac{\tan(x)}{\sqrt{1+\tan^2(x)}}$$

で求めることができます．

```
float TanToSin(float x)
{
    return x*rsqrt(x*x+1.0);
}
```

次にバイラテラルフィルタですが，次のようになっています．

```
#define KERNEL_RADIUS 15
uniform sampler2D OcclusionSampler;
uniform vec4 BlurParams;
varying vec2 vUv;

float CrossBilateralWeight(float r, float ddiff, inout float weightTotal) {
    float w = exp(-r*r*BlurParams.z) * (ddiff < BlurParams.w ? 1.0 : 0.0);
    weightTotal += w;
    return w;
}

vec2 Blur(vec2 texScale) {
    vec2 centerCoord = vUv;
    float weightTotal = 1.0;
    vec2 aoDepth = texture2D(OcclusionSampler, centerCoord).xy;
    float totalAO = aoDepth.x;
    float centerZ = aoDepth.y;
    for (int i=-KERNEL_RADIUS; i<KERNEL_RADIUS; i++) {
        vec2 texCoord = centerCoord + (float(i)*texScale);
        vec2 sampleAOZ = texture2D(OcclusionSampler, texCoord).xy;
        float diff = abs(sampleAOZ.y - centerZ);
        float weight = CrossBilateralWeight(float(i), diff, weightTotal);
        totalAO += sampleAOZ.x * weight;
    }

    return vec2(totalAO / weightTotal, centerZ);
}

void main() {
    gl_FragColor = vec4(Blur(BlurParams.xy), 0.0, 1.0);
}
```

重みを計算するために，遮蔽係数を出力するときにカメラ座標系での Z 値を一緒に出力しています．その値を使って重みを次の

ように計算しています.

```
float w = exp(-r*r*BlurParams.z) * (ddiff < BlurParams.w ? 1.0 : 0.0);
```

**blurParams.z** は **blurFalloff** で，ブラーの減衰係数です． **blurParams.w** は **threshold** で，深度差のしきい値です.

### 4.4 デモ

HBAO の結果は次のようになります.

**Fig.18:** HBAO

隠蔽係数を表示したものは次のようになっています.

**Fig.19:** HBAO - Occlusion

実際に動作するデモを用意しました.

HBAO Demo

## 5． ソースコード

各デモのソースコードは次の場所にあります.

https://github.com/mebiusbox/ssao

## 6．さいごに

かなり説明不足ではありますが，少しずつ補填していこうと思います．

# 参考文献


[1] Martin Mittring「Finding Next Gen - Cry Engine 2」
http://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf

[2] Louis Bavoil, Miguel Sainz「Image-Space Horizon-Based Ambient Occlusion」
https://developer.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf

[3] John Chapman「SSAO Tutorial」
http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html

[4] Joey de Vries「SSAO」
https://learnopengl.com/Advanced-Lighting/SSAO

[5] mtnphil「Know your SSAO artifacts」
https://mtnphil.wordpress.com/2013/06/26/know-your-ssao-artifacts/

[6] 床井浩平「SSAO (Screen Space Ambient Occlusion)」
http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20101122

[7] AO ちゃん「アンビエントオクルージョン・はじめの一歩」
http://ambientocclusion.hatenablog.com/entry/2013/10/15/223302