スクリーンスペース・リフレクション(SSR)

mebiusbox

2022 年 12 月 28 日

はじめに

今, この記事を皆さんは PC ディスプレイで見たり, スマートフォンやタブレット, またはテレビなどで見ていると思います.その画面で黒地のものを表示してよく見てみると周囲の景色が映り込んでいないでしょうか.反射防止フィルターやノングレア液晶を使っている場合はかなりボケた映像になっているかもしれません.液晶に限った話ではなく, 例えば大理石の床やガラス, 金属や光沢紙, 濡れた表面などを見てみると同じように景色が写り込んでいるのがわかると思います.これは今見ている物体の周辺に光源から放出された光子がぶつかって反射され, さらにその反射された光子が見ている物体に反射されて私達の目に届いて写り込んで見えています.コンピュータグラフィックスではこのような映り込みはレイトレーシング法で表現することが出来ます.ただし, 周辺の物体を含めた反射の計算量はとても大きく, 時間のかかる処理になっています.これをリアルタイムで処理するのは難しいので, テクスチャにあらかじめ周辺からの反射光を書き込んで参照する手法が使われています.このテクスチャを使った方法にはスフィアやデュアルパラボロイド, キューブマップといったものがあり, 環境マッピングと呼ばれています.昨今では, 周辺からの輝度をテクスチャや圧縮データにしてリアルタイムで反射光を計算する手法も主流になっていると思います.しかし, これらの手法では事前にデータを作成するので, 動的に変化するシーン(例えばキャラクターが移動するなど)での映り込みに対応できないという問題があります.環境マッピング用のテクスチャをリアルタイムに作成するといった方法もありますが, 制約やコストも高いため, 扱いづらいものとなっています.それ以外にも様々な方法が研究・開発されていますが, そんな中, Crytek がリアルタイム・ローカル・リフレクション(RealTime Local Reflection: RLR)を開発し, Crysis2 で採用されました.これはリアルタイムで周辺の映り込みを描画する手法で, 高速に処理することができます.この手法ではポストプロセス, つまりスクリーン空間上で処理するため, 画面に表示されていないものは写り込まないといった欠点があるものの, 動的なシーンに対応できるところから現在では広く使われている手法です.この手法はスクリーン空間上で処理するため, スクリーンスペース・リフレクション(ScreenSpace Reflection: SSR)とも呼ばれています.本記事では, 基本的なアルゴリズムの説明と, 改良版である Morgan McGuire 氏と Michael Mara 氏の論文「**Efficient GPU Screen-Space Ray Tracing**」に基づいた実装について簡単に解説したいと思います.なお, 実装は Three.js を使います.

※メモ書き程度の内容です.随時更新していこうと思います.

目次

1. 基本的なアルゴリズム

まず, 一度シーンを描画します.そのとき, カメラ空間における位置情報, 深度情報と法線情報を出力します(位置情報や法線情報は深度から復元することができます).また, 必要に応じてラフネスやメタルネスといった付加情報も出力します.次にカメラ位置からスクリーン上の各ピクセルにおける位置に向かうベクトルと法線ベクトルから反射ベクトルを求めます.位置から反射ベクトル方向にレイトレーシングを行い, 物体と交差したときに, その位置を投影したピクセルをサンプリングして周囲からの反射光として計算します.品質や計算負荷を考慮してパラメータを調整していきますが, 反射を計算したシーンはアーティファクトが目立ってしまうので, ぼかしをかけます.それと, 元のシーンをブレンドします.実際は間接光のスペキュラー項とブレンドすることになると思います.

1.1 反射ベクトル

最初に反射ベクトルを求めます.反射ベクトルを計算するにはカメラの位置, 対象ピクセルにおける位置と法線が必要です.深度からの位置情報の計算については「スクリーンスペース・アンビエント・オクルージョン: SSAO」などを参照してください.位置と法線の復元は次のようになります.

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

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

次にカメラの位置ですが, これはカメラ空間なので原点つまり $(0, 0, 0)$ の位置にカメラがあります.反射ベクトルは **reflect** 関数を使って求められます.この関数は入射ベクトルと法線ベクトルを渡します.入射ベクトルは

```
vec3 incidentVec = normalize(viewPosition);
```

ですので, 反射ベクトルは

```
vec3 reflectVec = reflect(normalize(viewPosition), viewNormal);
```

となります.

1.2 衝突判定

反射ベクトルを使ってシーンとの衝突判定を行います.次の図を見てください.

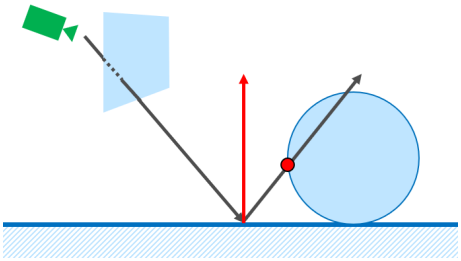

Fig.1: 反射ベクトルとオブジェクトとの衝突判定

この場合，反射ベクトルは球体にぶつかって赤い点が映り込むことになります.まず，次図のように反射ベクトルの開始点から少しずつ進めていきます.

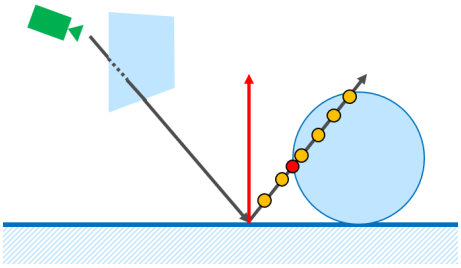

Fig.2: レイトレーシング

このコードは次のようになっています.

```
vec3 Q0 = rayOrg;
vec3 Q1 = rayEnd;
vec3 delta = Q1 - Q0;
vec3 deltaStep = delta / Iterations;
```

`rayOrg` はレイの開始点，`rayEnd` はレイの終止点です.処理負荷のため，ある程度レイを飛ばす距離を制限する必要があります.最大距離を `RayMaxDistance` として，

```
vec3 rayEnd = rayOrg + rayDir * RayMaxDistance;
```

となります. `rayDir` は反射ベクトルのことです.実際はクリッピング領域の範囲内に収まるように調整しています.

```
// Clip to the near plane
float rayLength = ((rayOrg.z + rayDir.z * MaxRayDistance) > -CameraNear) ?
    (-CameraNear - rayOrg.z) / rayDir.z : MaxRayDistance;
vec3 rayEnd = rayOrg + rayDir * rayLength;
```

1ステップごとの進む距離 `deltaStep` は次のようになっています.

```
vec3 delta = Q1 - Q0;
vec3 deltaStep = delta / Iterations;
```

`Iterations` は最大ステップ数です.

次はシーンの深度情報をどのように使うか見ていきます.シーン描画時に出力したカメラからの深度情報から，進めた先での深度情報を参照することができます.

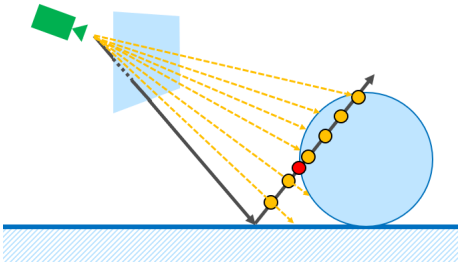

Fig.3: 深度情報

開始点から反射ベクトルを 1 ステップごとに進めたその位置における深度を計算します.

```
vec3 Q = Q0;
vec2 P;
bool intersect = false;
for (int i=0; i<MAX_ITERATIONS; i++)
{
    if (float(i) >= Iterations) break;
    if (intersect) break;

    Q += deltaStep;
```

`Q.z` が理想の深度となります.これとシーンを描画したときに出力した深度情報を比較します.このとき理想の深度より, 実際のシーンにおける深度が手間の場合に物体に衝突したと判断すること出来そうです.

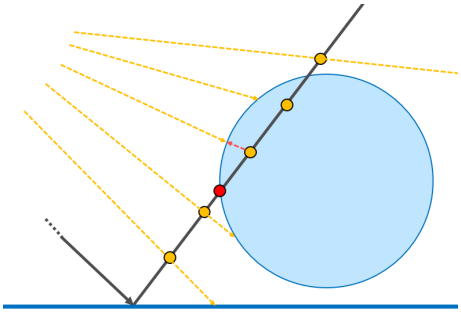

Fig.4: 衝突判定

しかし, 次図を見てください.

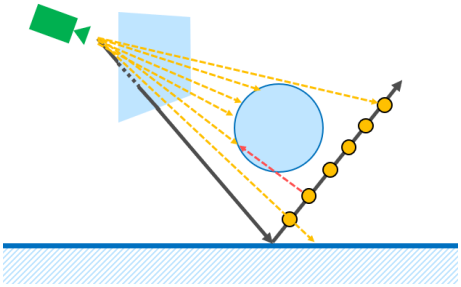

Fig.5: 衝突判定の問題

この場合，深度が手前になっていますが，実際には衝突していません.そのため，物体の厚さが必要になってきます.ここでは単純に，固定幅の厚さで対応します.つまり，理想の深度から手前に厚さ分の領域に実際の深度が入っているときに衝突したとみなします.

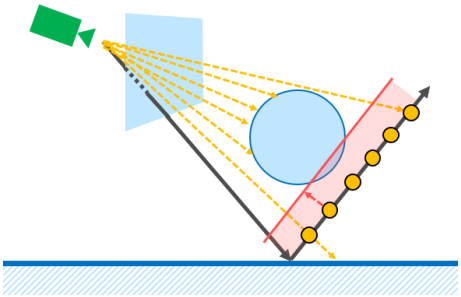

Fig.6: 衝突判定時に厚さを考慮

この問題については，背面を描画したシーンの深度を使用することでより正確な厚さ情報を使用することもできます.

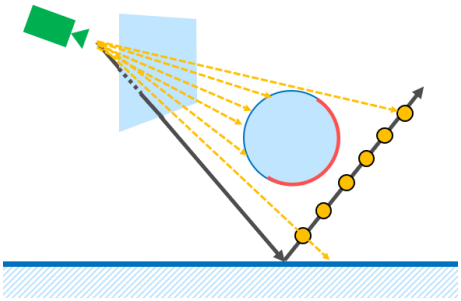

Fig.7: 衝突判定に背面の深度を使う

この場合は前面と背面の間に理想の深度が入っていれば衝突したと判定できます.ただし，閉じた物体でないとうまく判定できませんので，先程の固定幅と併用する形になります.

では，実際の衝突判定のコード部分を見ていきます.まず，シーンの深度情報を参照します.

```
vec4 clip = CameraProjectionMatrix * vec4(Q,1.0);
P = clip.xy / clip.w;

hitPixel = P.xy*0.5+0.5;
sceneZMax = getViewZ(getDepth(hitPixel));
```

clip は Q をクリップ空間に変換した座標です.その値を w 成分で割ることで正規化デバイス座標 P に変換します.そして正規化デバイス座標を UV 座標 **uv** に変換して，その値を使ってシーンの深度に変換します.

次に厚さの値を **Thickness** とすると，衝突判定は次のようになります.

```
bool rayIntersectsDepth(float z, vec2 uv)
{
    float sceneZMax = getViewZ(getDepth(uv));
    float dist = z - sceneZMax;
    return dist < 0.0 && dist > -Thickness;
```

```
}
```

ここで，`z` は理想の深度です.この関数を使った衝突判定は次のようになります.

```
bool intersect = false;
for (int i=0; i<MAX_ITERATIONS; i++)
{
    if (float(i) >= Iterations) break;
    if (intersect) break;

    Q += deltaStep;
    vec4 clip = CameraProjectionMatrix * vec4(Q,1.0);
    P = clip.xy / clip.w;

    hitPixel = P.xy*0.5+0.5;
    intersect = rayIntersectsDepth(Q.z, hitPixel);
}
```

intersect が **true** なら衝突しており，**hitPixel** には衝突先の UV 値が格納されています.この値を使ってシーンをサンプリングし，ラフネスやメタルネス・スペキュラーや後述するフェードによるブレンド率を計算して，合成します.これで，SSR の基本的な部分はできました.ここからは品質を上げるための処理となります.

1.3 二分探索

SSR の品質を上げるためには，ステップ数や最大距離を上げる必要がありますが，その分処理負荷が高くなってしまいますので，なるべくこの2つのパラメータは小さくしたいところです.例えば，ステップ数が少ないと深度の比較が十分に行えずに抜けてしまったり，衝突位置が大きくずれてしまいます.このずれを抑えるために，衝突したと判定されたら，さらに二分探索を行います.コードは次のようになります.

```
if (BinarySearchIterations > 0.0 && intersect)
{
    Q -= deltaStep;
    deltaStep /= BinarySearchIterations;

    float originalStride = pixelStride * 0.5;
    float stride = originalStride;

    for (int j=0; j<MAX_BINARY_SEARCH_ITERATIONS; j++)
    {
        if (float(j) >= BinarySearchIterations) break;

        Q += deltaStep * stride;
        vec4 clip = CameraProjectionMatrix * vec4(Q,1.0);
        P = clip.xy / clip.w;
```

```
        hitPixel = P.xy*0.5+0.5;

        originalStride *= 0.5;
        stride = rayIntersectsDepth(Q.z, hitPixel) ? -originalStride : originalStride;
    }
}
```

BinarySearchIterations は2分探索の回数です

1.4 スクリーンエッジフェード

SSR の弱点としてポストエフェクトで処理しているため, 画面に描画されているものしか反射できません.また, 画面の外側になるほど, 画面外に向かってレイトレーシングが行われてしまい, 正しい結果になりません.そこで, 画面の外側に行くほど SSR による反射を弱くします.単純に画面に対して円状の減衰をする場合は次のようになります.

```
float uvFactor = 2.0 * length(hitPixel - vec2(0.5,0.5));
uvFactor *= uvFactor;
float edge = max(0.0, 1.0 - uvFactor);
```

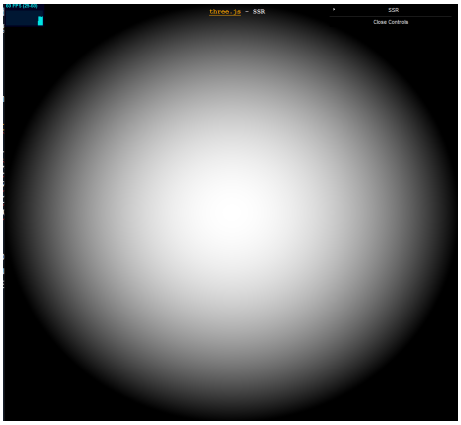

Fig.8: 円状の減衰

または, 周辺減光(ビネッティング)のように調整することもできます.

```
vec2 edgeuv = vUv * (1.0 - vUv.yx);
float edge = edgeuv.x * edgeuv.y * EdgeDistance;
edge = saturate(pow(abs(edge), EdgeExponent));
```

EdgeDistance と **EdgeExponent** は調整用パラメータです.

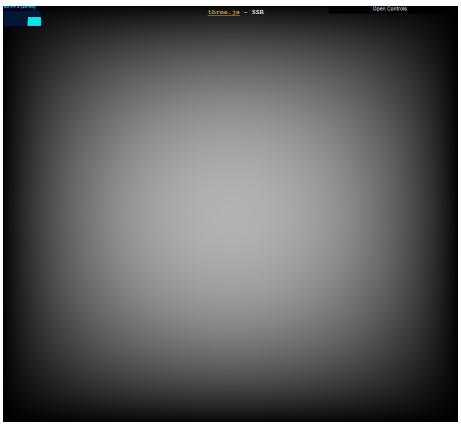

Fig.9: 周辺減光

1.5 距離フェード

衝突した位置が開始点から離れているほど反射を弱くします.これは光源と同様に離れているほど届く光子の量が減っているからです.これを行うために,衝突点のカメラ座標系での位置を計算して,開始点との距離から算出します.

```
float d = length(rayOrg - hitPoint);
alpha *= saturate(1.0 - pow(d / FadeDistance, FadeExponent));
```

hitPoint は衝突点のカメラ座標系での位置です.少しでも処理負荷を下げたいなら単純に高さで処理してもいいかもしれません.

1.6 デモ

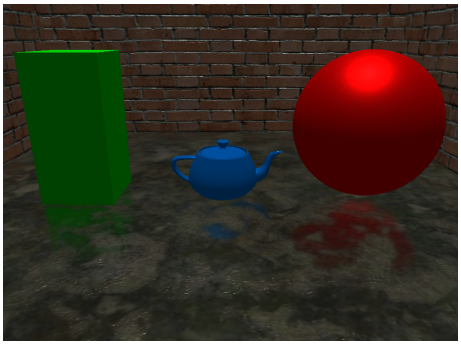

Fig.10: サンプル

デモサイトへのリンク

2. Efficient GPU Screen-Space Ray Tracing

Morgan McGuire 氏と Michael Mara 氏の論文「Efficient GPU Screen-Space Ray Tracing」を参考にした SSR を実装してみます.この手法の特徴はレイトレーシング時に, 2D の線描画アルゴリズムである DDA(Digital differential analyze) を行うというものです.ここでの通常の SSR ではカメラ座標系においてレイトレーシングを行っていました.その場合,スクリーン上に投影すると,サンプリング位置が飛び飛びであったり,同じピクセルを参照(オーバーサンプリング)してしまい,品質がよくありません. DDA を使用すれば,スクリーン上で必

ずIピクセルずらして参照したり,オーバーサンプリングを回避することができます.

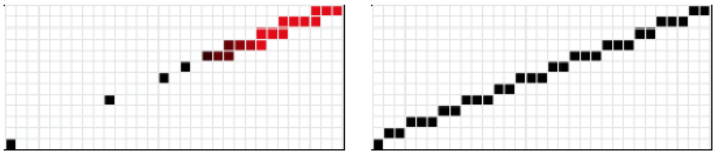

Fig.11: Morgan McGuire, Michael Mara「Efficient GPU Screen-Space Ray Tracing」より引用

上図の左側を見てみると,サンプリング位置が飛び飛びになっているのがわかります.また,赤くなっているところはオーバーサンプリングが多いことを示しています.右図では DDA を適用したものです.

2.1 DDA とは

デジタル差分解析器のことで,図形を描くときに整数や浮動小数点数を使ってコンピュータが高速に演算できるようにして解析する方法です.例えば,ブレゼンハムのアルゴリズムなどが有名です.

ここでは参考程度に, DDA による線描画と,ブレゼンハムによる線描画のコード(言語は Processing)を記載しておきます.

```
// DDA
class Point {
    int x,y;

    Point(int px, int py) {
        x = px;
        y = py;
    }

    void swap() {
        int tmp = x;
        x = y;
        y = tmp;
    }

    void swap(Point other) {
        int px = x;
        int py = y;
        x = other.x;
        y = other.y;
        other.x = px;
        other.y = py;
    }
}

void setup() {
    size(20,20);
```

```
}

void drawline(Point p0, Point p1, color c)
{
    int dx = abs(p1.x - p0.x);
    int dy = abs(p1.y - p0.y);
    int stepX = (p0.x < p1.x) ? 1 : -1;
    int stepY = (p0.y < p1.y) ? 1 : -1;
    int err = dx - dy;
    int x = p0.x;
    int y = p0.y;
    int err2;
    for (;;) {
        set(x,y,c);
        if (x == p1.x && y == p1.y) break;
        err2 = 2*err;
        if (err2 > -dy) {
            err -= dy;
            x += stepX;
        }
        if (err2 < dx) {
            err += dx;
            y += stepY;
        }
    }
}

void draw() {
    background(255,255,255);
    drawline(new Point(2,2), new Point(18,5),color(0));
    drawline(new Point(5,2), new Point(8,18),color(255,0,0));
}
```

ブレゼンハムによる線描画

```
class Point {
    int x,y;

    Point(int px, int py) {
        x = px;
        y = py;
    }

    void swap() {
        int tmp = x;
        x = y;
```

```
        y = tmp;
    }

    void swap(Point other) {
        int px = x;
        int py = y;
        x = other.x;
        y = other.y;
        other.x = px;
        other.y = py;
    }
}

void setup() {
    size(20,20);
}

void drawline(Point p0, Point p1, color c)
{
    boolean steep = abs(p1.y-p0.y) > abs(p1.x-p0.x);
    if (steep) {
        p0.swap();
        p1.swap();
    }
    if (p0.x>p1.x) {
        p0.swap(p1);
    }
    int deltaX = p1.x-p0.x;
    int deltaY = abs(p1.y-p0.y);
    int err = deltaX/2;
    int y = p0.y;
    int stepY = p0.y < p1.y ? 1 : -1;
    for (int x=p0.x; x<=p1.x; x++) {
        if (steep) set(y,x,c); else set(x,y,c);
        err = err - deltaY;
        if (err < 0) {
            y += stepY;
            err += deltaX;
        }
    }
}

void draw() {
    background(255,255,255);
    drawline(new Point(2,2), new Point(18,5),color(0));
```

```
    drawline(new Point(5,2), new Point(8,18),color(255,0,0));
}
```

2.2 レイトレーシング

実際のレイトレーシング処理ですが，通常の SSR ではカメラ空間でベクトルを進めていましたが，この手法では同次座標とスクリーン座標での開始点，終止点を求めて補間します．

```
// Project into homogeneous clip space
vec4 H0 = CameraProjectionMatrix * vec4(rayOrg, 1.0);
vec4 H1 = CameraProjectionMatrix * vec4(rayEnd, 1.0);

float k0 = 1.0 / H0.w, k1 = 1.0 / H1.w;

// The interpolated homogeneous version of the camera-space points
vec3 Q0 = rayOrg * k0, Q1 = rayEnd * k1;

// Screen-space endpoints
vec2 P0 = H0.xy * k0, P1 = H1.xy * k1;
P0 = (P0*0.5+0.5) * Resolution.xy;
P1 = (P1*0.5+0.5) * Resolution.xy;
```

$Q0, Q1$ は同次座標系，$P0, P1$ はスクリーン座標系です．ここで注意するところは透視補正補間を行っていることです．詳しくは「付録A:透視補正補間(6)」を参照してください．次に，スクリーン座標において，差分値が0にならないように調整します．

```
P1 += (distanceSquared(P0,P1) < 0.0001) ? 0.01 : 0.0;
```

distanceSquared 関数は次のようになっています．

```
float distanceSquared(vec2 a, vec2 b) { a -= b; return dot(a,a); }
```

そして，DDA 用の変数を初期化します．

```
vec2 delta = P1 - P0;

bool permute = false;
if (abs(delta.x) < abs(delta.y)) {
    // This is a more-vertical line
    permute = true;
    delta = delta.yx; P0 = P0.yx; P1 = P1.yx;
}

float stepDir = sign(delta.x);
float invdx = stepDir / delta.x;
```

```
// Track the derivatives of Q and K
vec3  dQ = (Q1 - Q0) * invdx;
float dk = (k1 - k0) * invdx;
vec2  dP = vec2(stepDir, delta.y * invdx);
```

PixelStride というパラメータで走査するときのピクセルの間隔を調整します.ここで, カメラと開始点との距離が離れるほど, この値が小さくなるようにして, 遠距離における精度を改善します.

```
float strideScalar = 1.0 - min(1.0, -rayOrg.z / PixelStrideZCutoff);
float pixelStride = 1.0 + strideScalar * PixelStride;
```

この **PixelStride** の値を大きくすると処理速度が向上しますが, 品質が下がります.そのため, **PixelStride** を大きくしたときのアーティファクト軽減としてジッタリングを加えることもできます.

```
vec2 uv2 = vUv * Resolution.xy;
float c = (uv2.x + uv2.y) * 0.25;
float jitter = mod(c, 1.0) * Jitter;
```

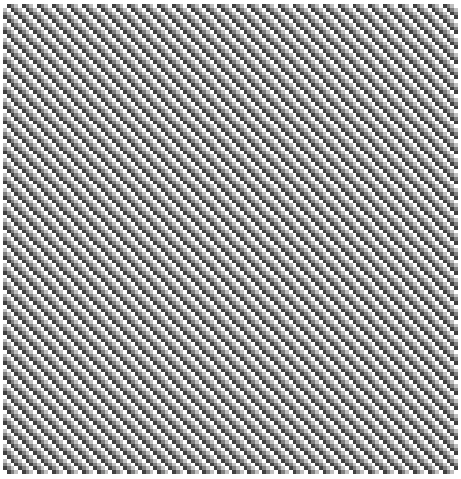

Fig.12: ジッタリング

この値は次のように使います.

```
dP *= pixelStride; dQ *= pixelStride; dk *= pixelStride;
P0 += dP * jitter; Q0 += dQ * jitter; k0 += dk * jitter;
```

繰り返しの部分は次のようになっています.

```
float zA, zB = Q0.z;

vec4 pqk = vec4(P0, Q0.z, k0);
vec4 dPQK = vec4(dP, dQ.z, dk);
```

```
bool intersect = false;
float count = 0.0;
for (int i=0; i<MAX_ITERATIONS; i++)
{
    if (float(i) >= Iterations) break;
    if (intersect) break;

    pqk += dPQK;

    zA = zB;
    zB = (dPQK.z * 0.5 + pqk.z) / (dPQK.w * 0.5 + pqk.w);
    swapGEQ(zB, zA);

    hitPixel = permute ? pqk.yx : pqk.xy;
    hitPixel *= Resolution.zw;

    intersect = rayIntersectsDepth(zA, zB, hitPixel);

    count = float(i);
}
```

swapGEQ は第一引数が第二引数より大きいときに値を交換します.また,深度を計算するときに 1 ステップの半分を足している理由は申し訳ないですけど,よくわかっていません.

```
void swapGEQ(inout float aa, inout float bb)
{
    if (aa > bb)
    {
        float tmp = aa;
        aa = bb;
        bb = tmp;
    }
}
```

また,この後に通常の SSR と同じように二分探索を行って精度を上げています.

最後に衝突した位置 **hitPoint** を計算して,衝突判定結果を返します.

```
Q0.xy += dQ.xy * count;
Q0.z = pqk.z;
hitPoint = Q0 / pqk.w;
return intersect;
```

2.3 デモ

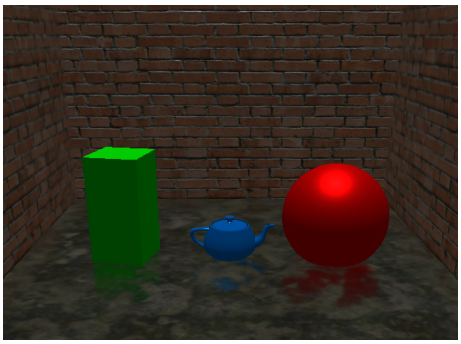

Fig.13: デモ

デモサイトへのリンク

3. ソースコード

デモのソースコードは以下の場所にあります.

https://github.com/mebiusbox/ssr

4. SSR の問題点

最後に SSR の問題点をいくつか見ていきます.まずは下図を見てください.

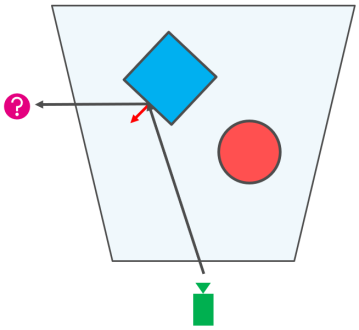

Fig.14: 反射先がカメラの外

図中の台形領域は視錐台で,レンダリングされる領域です.青い箱からの反射ベクトル方向は視錐台から外れてしまうため描画されず,衝突判定が行えません.これはスクリーンスペースにおけるポストエフェクトに共通の問題です.

今度は次のような場合です.

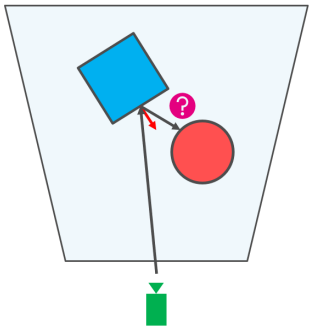

Fig.15: 反射先がオブジェクトの裏側

この場合, 赤い球体の背面に衝突することを期待しますが, 今回の実装では基本的に衝突しません.例外は赤い球体が小さく厚みが薄い(Thickness 以下)場合は衝突します.また, 衝突したとしても背面における反射光をサンプリングしたいのですが, 実際はレンダリングしたシーンからサンプリングするため, 前面における反射光になってしまいます.

3つ目は次のような場合です.

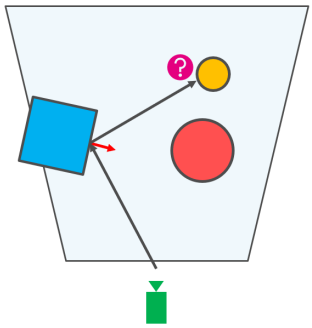

Fig.16: 反射先が隠れている

この場合, 黄色の球体に衝突してほしいですが, カメラからは赤の球体に遮られて見えないため衝突しません.つまりカメラから奥の方にある物体からの反射が出来ません.

このように様々な問題点を抱えていますが, それでも見映えがよくなるので, 有用な手法であることには変わりません.

5. さいごに

SSAO と同様に現在は一般的に使われている(と思う) SSR について書いてみました.参考になれば幸いです.私も完全に理解しているわけではないので, 間違いなど見つけたらご連絡していただけると助かります.

6. 付録 A:透視補正補間

コンピュータグラフィックスでは, 一般的に三角形のリストを GPU に送ってレンダリングしています. GPU は各三角形を描画するときにラスタライズという処理を行います.ラスタライズは三角形をスキャンライン(走査線)ごとに描画します.このとき, 各ピクセルをレンダリングするときに参照される情報は, 三角形の各頂点情報を補間した値です.次の図を見てください.

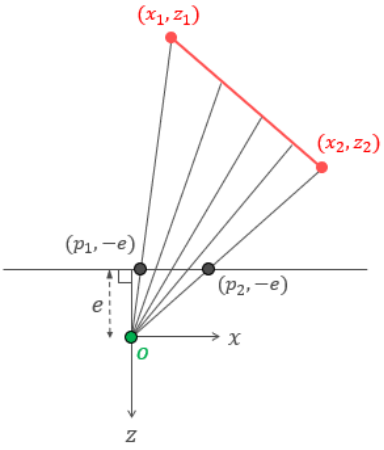

Fig.17: 透視補正補間

O はカメラの位置です.カメラから z 方向に $-e$ のところに投影面があります. (x_1, z_1) と (x_2, z_2) は三角形の面上の任意の2点です.この面をレンダリングすると $(p_1, -e)$ と $(p_2, -e)$ の範囲に投影されます.このとき投影面上で均等の間隔にある位置は, 三角形の面上だと均等になっておらず, カメラからの距離が大きくなるほど間隔が大きくなっていることがわかります.ラスタライズは投影面上でピクセルごとに処理していくため, 頂点情報を非線形で補間する必要があります.このような補正を透視補正(パースペクティブコレクト)補間といいます.通常は GPU が頂点シェーダから出力した値を透視補正補間してくれます.ところが, 今回の GPU レイトレーシングでは, ラスタライズと同じようなことをしており, 透視補正補間を行う必要があります.では透視補正補間とはどのような処理なのか, これから解説していきます.
図のカメラ O から $(p_1, -e)$ から $(p_2, -e)$ の間を通り, 三角形の面上に到達する直線は次の式で記述できます.

$$ax + bz = c \quad (c \neq 0) \tag{1}$$

相似の三角形の関係から, 三角形の面上の点 (x, y) に対応する投影面上の x の位置 p は次のようになります.

$$\frac{p}{x} = \frac{-e}{z} \tag{2}$$

これを式(1)に代入すると

$$\left(-\frac{ap}{e} + b\right)z = c \tag{3}$$

この式を整理して次のようにします.

$$\frac{1}{z} = -\frac{ap}{ce} + \frac{b}{c} \tag{4}$$

ここで, 投影面上の $(p_1, -e)$ と $(p_2, -e)$ の間にある任意の点を p' としたとき, $0 \leq t \leq 1$ を満たす t を使うと

$$p' = (1 - t)p_1 + tp_2 \tag{5}$$

となります.この直線と三角形の面に交差する点を (x', z') とすると, 式(5)を式(4)に代入すれば

$$
\begin{aligned}
\frac{1}{z'} &= -\frac{-ap'}{ce} + \frac{b}{c} \\
&= -\frac{ap_1}{ce}(1-t) - \frac{ap_2}{ce}t + \frac{b}{c} \\
&= \left(-\frac{ap_1}{ce} + \frac{b}{c}\right)(1-t) + \left(-\frac{ap_2}{ce} + \frac{b}{c}\right)t \\
&= \frac{1}{z_1}(1-t) + \frac{1}{z_2}t
\end{aligned} \tag{6}
$$

が得られます.これは $\frac{1}{z}$ を補間すると三角形の面上において,線形に補間されていることを示しています.

次に三角形の頂点情報のうち,任意のスカラー値を b とすると,点 (x_1, z_1) と (x_2, z_2) のスカラー値を b_1, b_2,補間されたスカラー値を b' とします.このとき,深度の補間の比率とスカラー値の補間の比率が等しくなるはずです.

$$
\frac{b'-b_1}{b_2-b_1} = \frac{z'-z_1}{z_2-z_1} \tag{7}
$$

式(7)を整理して, b' を求めると

$$
\begin{aligned}
\frac{b'-b_1}{b_2-b_1} &= \frac{z'-z_1}{z_2-z_1} \\
b'-b_1 &= \frac{(z'-z_1)(b_2-b_1)}{z_2-z_1} \\
b' &= \frac{(z'-z_1)(b_2-b_1)}{z_2-z_1} + b_1 \\
&= \frac{z'(b_2-b_1) - b_2z_1 + b_1z_1 + b_1z_2 - b_1z_1}{z_2-z_1} \\
&= \frac{z'(b_2-b_1) - b_2z_1 + b_1z_2}{z_2-z_1}
\end{aligned} \tag{8}
$$

この式(8)に式(6)を代入して, b' について解きます.まずは分子の $z'(b_2-b_1)$ を展開すると

$$
\begin{aligned}
z'(b_2-b_1) &= \frac{b_2-b_1}{\frac{1}{z_1}(1-t) + \frac{1}{z_2}t} = \frac{b_2-b_1}{\frac{1}{z_1} - \frac{t}{z_1} + \frac{t}{z_2}} = \frac{b_2-b_1}{\frac{z_2}{z_1z_2} - \frac{z_2t}{z_1z_2} + \frac{z_1t}{z_1z_2}} \\
&= \frac{b_2-b_1}{\frac{z_2-z_2t+z_1t}{z_1z_2}} = \frac{(b_2-b_1)z_1z_2}{z_2-z_2t+z_1t} = \frac{(b_2-b_1)z_1z_2}{t(z_1-z_2)+z_2}
\end{aligned} \tag{9}
$$

次に分子全体です.

$$
\begin{aligned}
z'(b_2-b_1) - b_2z_1 + b_1z_2 &= \frac{(b_2-b_1)z_1z_2}{t(z_1-z_2)+z_2} - b_2z_1 + b_1z_2 \\
&= \frac{b_2z_1z_2 - b_1z_1z_2}{t(z_1-z_2)+z_2} - \frac{b_2z_1t(z_1-z_2) - b_2z_1z_2}{t(z_1-z_2)+z_2} + \frac{b_1z_2t(z_1-z_2) + b_1z_2^2}{t(z_1-z_2)+z_2} \\
&= \frac{-b_1z_1z_2 + b_1z_2^2 - b_2z_1t(z_1-z_2) + b_1z_2t(z_1-z_2)}{t(z_1-z_2)+z_2} \\
&= \frac{-b_1z_2(z_1-z_2) - b_2z_1t(z_1-z_2) + b_1z_2t(z_1-z_2)}{t(z_1-z_2)+z_2} \\
&= \frac{(z_1-z_2)(b_1z_2t - b_2z_1t - b_1z_2)}{t(z_1-z_2)+z_2}
\end{aligned} \tag{10}
$$

式(8)の分母も含めて解くと

$$\begin{aligned}\frac{z'(b_2-b_1)-b_2z_1+b_1z_2}{z_2-z_1} &= \frac{(z_1-z_2)(b_1z_2t-b_2z_1t-b_1z_2)}{t(z_1-z_2)+z_2}\cdot\frac{1}{z_2-z_1}\\ &= \frac{(z_1-z_2)(b_1z_2t-b_2z_1t-b_1z_2)}{(z_2-z_1)(z_1t-z_2t+z_2)}\\ &= \frac{-(z_2-z_1)(b_1z_2t-b_2z_1t-b_1z_2)}{(z_2-z_1)(z_1t-z_2t+z_2)}\\ &= \frac{-(b_1z_2t-b_2z_1t-b_1z_2)}{z_1t-z_2t+z_2}\\ &= \frac{b_1z_2-b_1z_2t+b_2z_1t}{z_2-z_2t+z_1t}\end{aligned} \tag{11}$$

となります.最後に式(11)の分母分子に $1/z_1z_2$ を掛けると

$$\frac{b_1z_2-b_1z_2t+b_2z_1t}{z_2-z_2t+z_1t}\cdot\frac{\frac{1}{z_1z_2}}{\frac{1}{z_1z_2}} = \frac{\frac{b_1}{z_1}-\frac{b_1t}{z_1}+\frac{b_2t}{z_2}}{\frac{1}{z_1}-\frac{t}{z_1}+\frac{t}{z_2}} = \frac{\frac{b_1}{z_1}(1-t)+\frac{b_2}{z_2}t}{\frac{1}{z_1}(1-t)+\frac{1}{z_2}t} \tag{12}$$

よって

$$b' = \frac{\frac{b_1}{z_1}(1-t)+\frac{b_2}{z_2}t}{\frac{1}{z_1}(1-t)+\frac{1}{z_2}t} \tag{13}$$

ここで式(6)から

$$\frac{1}{z'} = \frac{1}{z_1}(1-t)+\frac{1}{z_2}t \tag{14}$$

なので

$$b' = z'\left\{\frac{b_1}{z_1}(1-t)+\frac{b_2}{z_2}t\right\} \tag{15}$$

となります.これは頂点情報のスカラー値に z 値の逆数を掛けて補間した値を，z 値の逆数を補間した値で割ることで，投影面上で線形に補間された値が得られることを示しています.実際の GPU レイトレーシングのコードを見てみましょう. $k0, k1$ が w の逆数ですが，w は透視投影の行列変換によって $-z$ 値が入っています.

```
vec4 H0 = CameraProjectionMatrix * vec4(rayOrg, 1.0);
vec4 H1 = CameraProjectionMatrix * vec4(rayEnd, 1.0);

float k0 = 1.0 / H0.w, k1 = 1.0 / H1.w;
```

ここでの頂点情報のスカラー値は xyz のことですから，

```
vec3 Q0 = rayOrg * k0, Q1 = rayEnd * k1;
```

は式(13)の分子の部分に当たります.そして，衝突判定時に補間した z 値を計算して使用するときは

```
zB = (dPQK.z * 0.5 + pqk.z) / (dPQK.w * 0.5 + pqk.w);
```

また,衝突した位置を計算するときも

```
Q0.xy += dQ.xy * count;
Q0.z = pqk.z;
hitPoint = Q0 / pqk.w;
```

としています.

参考文献

[1] Nickolay Kasyan, Nicolas Schulz, Tiago Sousa「Secrets of CryENGINE 3 Graphics Technology」2011

[2] Morgan McGuire, Michael Mara「Efficient GPU Screen-Space Ray Tracing」2014

[3] kode80,「Screen Space Reflections in Unity 5」
http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/

[4] Morgan McGuire,「Screen Space Ray Tracing」
http://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html

[5] Bart Wronski,「The future of screenspace reflections」
https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/

[6] Wikipedia,「Bresenham's line algorithm」
https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

[7] Wikipedia,「Digital differential analyzer」
https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)

[8] Eric Lengyel, 狩野智英訳「ゲームプログラミングのための 3D グラフィックス数学」ボーンデジタル, 2006