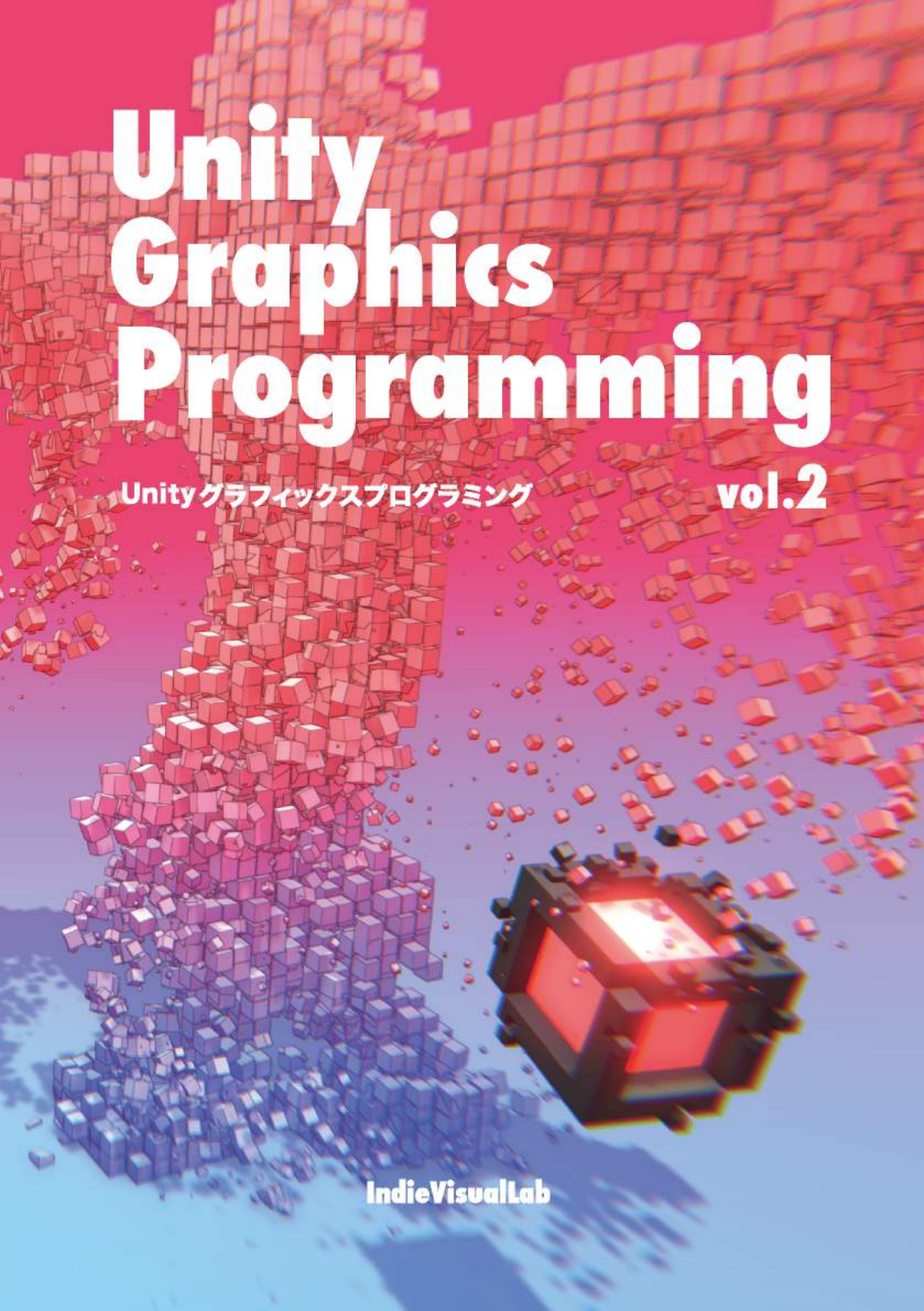
Unity Graphics Programming

Unityグラフィックスプログラミング vol.2



IndieVisualLab

Unity Graphics Programming vol.2

IndieVisualLab　著

2018-04-22 版　　IndieVisualLab 発行

まえがき

本書は Unity によるグラフィクスプログラミングに関する技術を解説する「Unity グラフィックスプログラミング」シリーズの第二巻です。本シリーズでは、執筆者たちの興味の赴くままに取り上げられた様々なトピックについて、初心者向けの入門的な内容と応用を解説したり、中級者以上向けの tips を掲載しています。

各章で解説されているソースコードについては github リポジトリ (https://github.com/IndieVisualLab/UnityGraphicsProgramming2) にて公開していますので、手元で実行しながら本書を読み進めることができます。

記事によって難易度は様々で、読者の知識量によっては、物足りなかったり、難しすぎる内容のものがあるかと思います。自分の知識量に応じて、気になったトピックの記事を読むのが良いでしょう。普段仕事でグラフィクスプログラミングを行っている人にとって、エフェクトの引き出しを増やすことにつながれば幸いですし、学生の方でビジュアルコーディングに興味があり、Processing や openFrameworks などは触ったことはあるが、まだまだ 3DCG に高い敷居を感じている方にとっては、Unity を導入として 3DCG での表現力の高さや開発の取っ掛かりを知る機会になれば嬉しいです。

IndieVisualLab は、会社の同僚 (&元同僚) たちによって立ち上げられたサークルです。社内では Unity を使って、一般的にメディアアートと呼ばれる部類の展示作品のコンテンツプログラミングをやっており、ゲーム系とはまた一味違った Unity の活用をしています。本書の中にも節々に展示作品の中で Unity を活用する際に役立つ知識が散りばめられているかもしれません。

推奨する実行環境

本書で解説する内容の中には Compute Shader や Geometry Shader 等を使ったものがあり、DirectX11 が動作する実行環境を推奨しますが、CPU 側のプログラム (C#) で内容が完結する章もあります。

環境の違いによって公開しているサンプルコードの挙動が正しくならない、といったことが起こり得るかと思いますが、github リポジトリへの issue 報告、適宜読み替える、等の対処をお願いします。

本についての要望や感想

本書についての感想や気になった点、その他要望（○○についての解説が読みたい等）がありましたら、ぜひ Web フォーム（https://docs.google.com/forms/d/e/1FAIpQLSdxeansJvQGTWfZTBN_2RTuCK_kRqhA6QHTZKVXHCijQnC8zw/viewform）、またはメール（lab.indievisual@gmail.com）よりお知らせください。

目次

第 1 章

Real-Time GPU-Based Voxelizer

1.1 はじめに

本章では、GPU を活用してリアルタイムにメッシュのボクセル化を行うプログラム、GPUVoxelizer を開発します。

本章のサンプルは

`https://github.com/IndieVisualLab/UnityGraphicsProgramming2`

の「RealTimeGPUBasedVoxelizer」です。

まずは CPU での実装をもとにボクセル化の手順と得られる結果について確認した後、GPU での実装方法を解説し、高速なボクセル化を応用したエフェクト例を紹介します。

1.1.1 ボクセル（Voxel）とは

ボクセル（Voxel）は、3 次元の正規格子空間における基本単位を表します。2 次元の正規格子空間の基本単位として用いられるピクセル（Pixel）の次元が 1 つ増えたものとしてイメージすることができ、Volume を持った Pixel という意味合いで Voxel（ボクセル）と命名されています。ボクセルでは体積を表現することができ、各ボクセルに濃度などの値を格納したデータフォーマットを用意して、医療や科学データの可視化や解析に用いられることがあります。

また、ゲームでは Minecraft[*1]がボクセルを利用したものとして挙げられます。

細かいモデルやステージを作り込むのは手間がかかりますが、ボクセルモデルであ

[*1] https://minecraft.net

れば比較的少ない手間で作ることができ、フリーでも MagicaVoxel[*2]などの優秀なエディターもあり、3D ドット絵のようにモデルを作成することができます。

1.2 ボクセル化のアルゴリズム

ボクセル化のアルゴリズムについて、CPU での実装をもとに解説していきます。CPU 実装は CPUVoxelizer.cs 内に記述しています。

1.2.1 ボクセル化の大まかな流れ

ボクセル化の大まかな流れは以下になります。

1. ボクセルの解像度を設定する
2. ボクセル化を行う範囲を設定する
3. ボクセルデータを格納する 3 次元の配列データを生成する
4. メッシュの表面に位置するボクセルを生成する
5. メッシュの表面を表すボクセルデータから、メッシュ内部に位置するボクセルを埋める

CPU でのボクセル化は、CPUVoxelizer クラスの static 関数

▼ CPUVoxelizer.cs

```
public class CPUVoxelizer
{
    public static void Voxelize (
        Mesh mesh,
        int resolution,
        out List<Vector3> voxels,
        out float unit,
        bool surfaceOnly = false
    ) {
    ...
    }
    ...
}
```

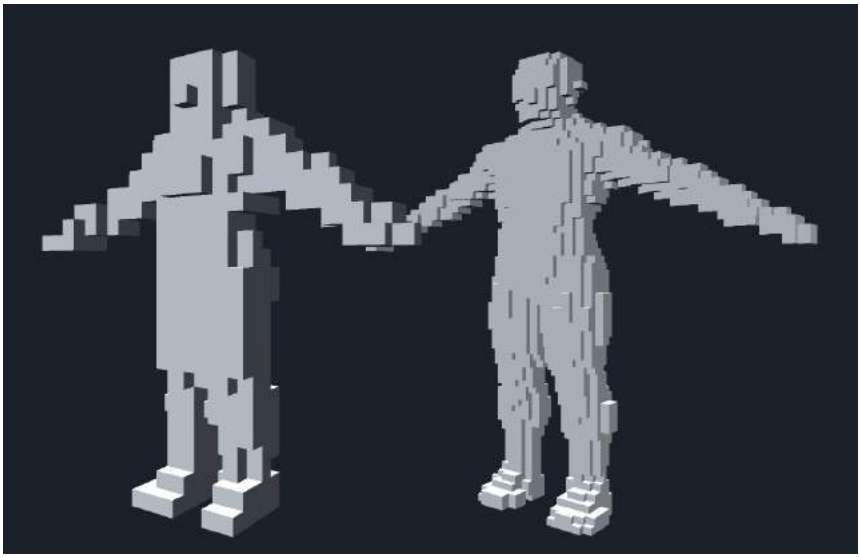
をコールすることで実行します。引数にボクセル化したいメッシュ、解像度を指定して実行すると、ボクセル配列 voxels と、一つのボクセルのサイズを表す unit を参照引数経由で返します。

以下では Voxelize 関数の内部で行われていることを大まかな流れに沿って解説していきます。

[*2] http://ephtracy.github.io/

1.2.2 ボクセルの解像度を設定する

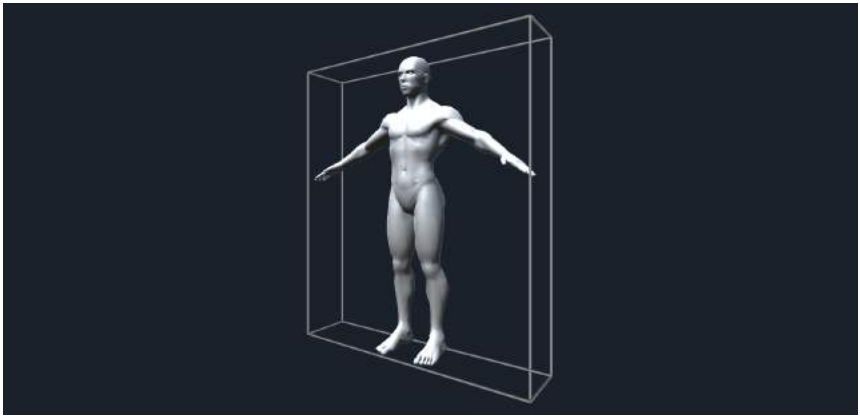
ボクセル化を行うには、まず、ボクセルの解像度を設定します。解像度が細かいほど小さなキューブでモデルを構築することになるので、詳細なボクセルモデルを生成できますが、その分計算時間を必要とします。



▲図 1.1　ボクセル解像度の違い

1.2.3 ボクセル化を行う範囲を設定する

対象のメッシュモデルをボクセル化する範囲を指定します。メッシュモデルが持つBoundingBox（モデルの頂点が全て収まる最小サイズの直方体）をボクセル化の範囲として指定すれば、メッシュモデル全体をボクセル化することができます。



▲図 1.2　メッシュの BoundingBox

ここで注意したいのは、メッシュモデルが持つ BoundingBox をそのままボクセル化の範囲として用いると、Cube メッシュのように、BoundingBox にぴったり重なるような面を持つメッシュをボクセル化する際に問題が生じてしまいます。

詳しくは後述しますが、ボクセル化の際に、メッシュを構成する三角形とボクセルとの交差判定を行うのですが、三角形とボクセルの面がぴったり重なる場合、正しく交差判定できない場合があります。

そのため、メッシュモデルが持つ BoundingBox を「一つのボクセルを構成する単位長の半分の長さ分」拡張した範囲を、ボクセル化の範囲として指定します。

▼ CPUVoxelizer.cs

```
mesh.RecalculateBounds();
var bounds = mesh.bounds;

// 指定された解像度から、一つのボクセルを構成する単位長を計算する
float maxLength = Mathf.Max(
    bounds.size.x,
    Mathf.Max(bounds.size.y, bounds.size.z)
);
unit = maxLength / resolution;

// 単位長の半分の長さ
var hunit = unit * 0.5f;

// 「一つのボクセルを構成する単位長の半分の長さ分」拡張した範囲を
// ボクセル化の範囲とする

// ボクセル化する bounds の最小値
var start = bounds.min - new Vector3(hunit, hunit, hunit);
```

```
// ボクセル化する bounds の最大値
var end = bounds.max + new Vector3(hunit, hunit, hunit);

// ボクセル化する bounds のサイズ
var size = end - start;
```

1.2.4 ボクセルデータを格納する 3 次元の配列データを生成する

ボクセルを表す構造体として、サンプルコードでは Voxel_t 構造体を用意しています。

▼ Voxel.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Voxel_t {
    public Vector3 position;   // ボクセルの位置
    public uint fill;          // ボクセルを埋めるべきかどうかのフラグ
    public uint front;         // ボクセルと交差した 3 角形が決められた方向から見
て前面かどうかのフラグ
    ...
}
```

この Voxel_t の三次元配列を生成し、そこにボクセルデータを格納していきます。

▼ CPUVoxelizer.cs

```
// ボクセルの単位長さとボクセル化を行う範囲に基づいて、3 次元ボクセルデータのサイズを決
定する
var width = Mathf.CeilToInt(size.x / unit);
var height = Mathf.CeilToInt(size.y / unit);
var depth = Mathf.CeilToInt(size.z / unit);
var volume = new Voxel_t[width, height, depth];
```

また、後に続く処理の中で各ボクセルの位置やサイズを参照するため、事前に 3 次元ボクセルデータと一致する AABB 配列を生成しておきます。

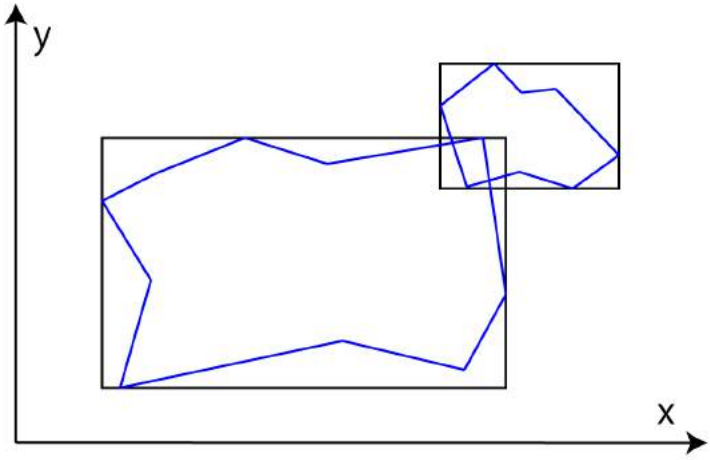
▼ CPUVoxelizer.cs

```
var boxes = new Bounds[width, height, depth];
var voxelUnitSize = Vector3.one * unit;
for(int x = 0; x < width; x++)
{
    for(int y = 0; y < height; y++)
    {
        for(int z = 0; z < depth; z++)
        {
            var p = new Vector3(x, y, z) * unit + start;
            var aabb = new Bounds(p, voxelUnitSize);
            boxes[x, y, z] = aabb;
        }
```

```
    }
}
```

AABB

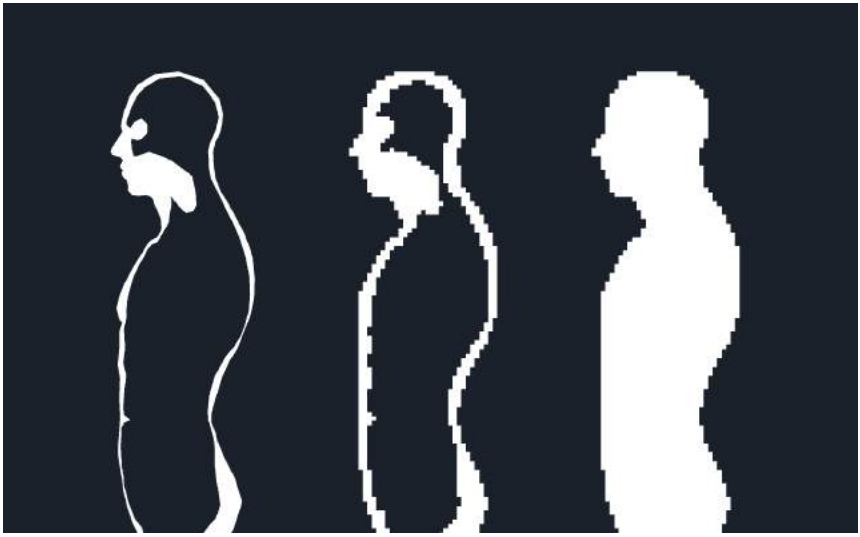
AABB（Axis-Aligned Bounding Box、軸平行境界ボックス）とは、3 次元空間の XYZ 軸に対して各辺が平行な直方体の境界図形を指します。
AABB は衝突判定に利用されることが多く、2 つのメッシュ同士の衝突判定や、あるメッシュと光線との衝突判定を簡易的に行う際に使われたりするケースがあります。
メッシュに対する衝突判定を厳密にやろうとすると、メッシュを構成する三角形全てに対して判定を行わなければなりませんが、メッシュを含む AABB のみであれば高速に計算することができ、便利です。



▲図 1.3　2 つの多角形オブジェクトの AABB 同士の衝突判定

1.2.5　メッシュの表面に位置するボクセルを生成する

以下の図のように、メッシュの表面に位置するボクセルを生成します。



▲図 1.4　まずメッシュの表面に位置するボクセルを生成し、それを基にしてメッシュの中身を埋めていくようにボクセルを生成する

メッシュの表面に位置するボクセルを求めるには、メッシュを構成するそれぞれの 3 角形とボクセルとの交差判定を行う必要があります。

3 角形とボクセルの交差判定（SAT を用いた交差判定アルゴリズム）

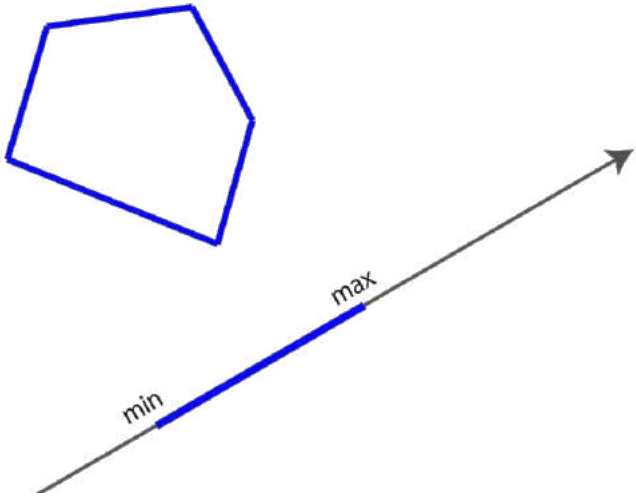
3 角形とボクセルとの交差判定には、SAT（Separating Axis Theorem、分離軸定理）を用います。SAT を用いた交差判定アルゴリズムは 3 角形とボクセル同士に限らず、凸面同士の交差判定として汎用的に用いることができます。

SAT は以下のことを証明しました。

> **SAT（Separating Axis Theorem、分離軸定理）の簡単な解説**
>
> 一方にオブジェクト A 全体、もう一方にオブジェクト B 全体が存在するような直線が求められれば、オブジェクト A とオブジェクト B は交わりません。このような、2 つのオブジェクトを分離する直線を分離直線と呼び、その分離直線は常に分離軸と直交します。

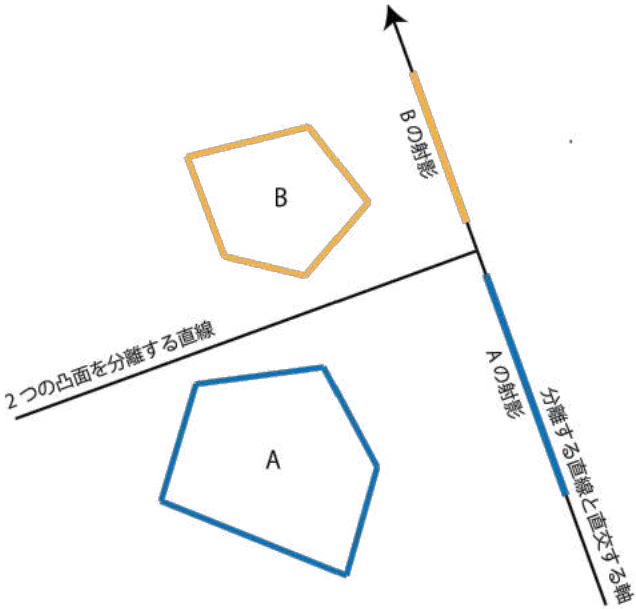
SAT により、2 つの凸面同士の射影が重ならない軸（分離軸）が求められれば、その 2 つの凸面を分離する直線が存在するので、2 つの凸面が交わらないということを導き出せます。逆に、分離軸が見つからなければ 2 つの凸面同士は交差していると判断できます。（形状が凹である場合、分離軸が見つからなくとも交差していない場合があります。）

凸面形状は、ある軸に射影されると、その形状の影がその軸を表す線の上に投影されたように映ります。これは軸上の線分として表すことができ、範囲区間 [min, max] で表わせます。



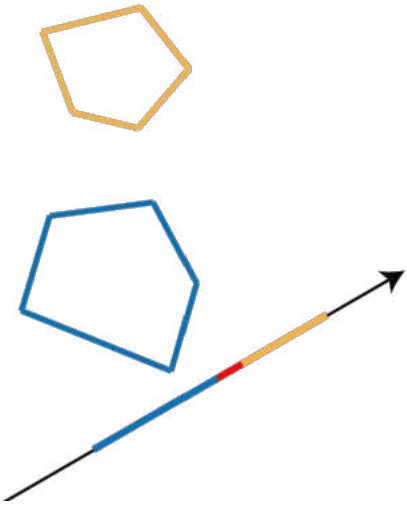
▲図 1.5　凸面形状をある軸に射影し、軸上に映された凸面形状の範囲（min, max）

以下の図のように、2 つの凸面形状の分離直線が存在する場合、その直線に直交する分離軸に対する凸面形状の射影区間同士は重なりません。



▲図 1.6　　2つの凸面形状を分離する直線がある場合、その直線に直交する軸への射影区間同士は重ならない

しかし同じ2つの凸面であっても、以下の図のように、他の非分離軸への射影は重なることがあります。



▲図 1.7　2 つの凸面形状を分離しない直線と直交する軸に射影した場合、射影同士が重なることがある

形状によっては、分離軸になる可能性のある軸が明白であるものがあり、そのような 2 つの形状 A と B の間の交わりを判定するには、分離軸になり得る軸それぞれに 2 つの形状を射影し、2 つの射影区間 [Amin, Amax] と [Bmin, Bmax] が互いに重なっていないかどうかを調べていけば交差判定ができます。式で表すと、Amax < Bmin または Bmax < Amin であれば、2 つの区間は重なっていません。

凸面同士の分離軸になる可能性がある軸は

- 凸面 1 の辺と凸面 2 の辺のクロス積
- 凸面 1 の法線
- 凸面 2 の法線

であることが導かれており、このことから三角形とボクセル（AABB）同士では、分離軸になる可能性がある軸は

- 三角形の 3 つの辺と AABB の直交する 3 つの辺同士の組み合わせから得られる 9 通りのクロス積
- AABB の 3 つの法線
- 三角形の法線

であるため、これら 13 通りの軸それぞれについて、射影が重なるかどうか判定していくことで、三角形とボクセルとの交差判定を行います。

一つの 3 角形とすべてのボクセルデータとの交差判定を行うのは無駄な処理が多くなる可能性があるため、3 角形を含む AABB を計算し、そこに含まれるボクセルとの交差判定を行います。

▼ CPUVoxelizer.cs

```
// 三角形の AABB を計算
var min = tri.bounds.min - start;
var max = tri.bounds.max - start;
int iminX = Mathf.RoundToInt(min.x / unit);
int iminY = Mathf.RoundToInt(min.y / unit);
int iminZ = Mathf.RoundToInt(min.z / unit);
int imaxX = Mathf.RoundToInt(max.x / unit);
int imaxY = Mathf.RoundToInt(max.y / unit);
int imaxZ = Mathf.RoundToInt(max.z / unit);
iminX = Mathf.Clamp(iminX, 0, width - 1);
iminY = Mathf.Clamp(iminY, 0, height - 1);
iminZ = Mathf.Clamp(iminZ, 0, depth - 1);
imaxX = Mathf.Clamp(imaxX, 0, width - 1);
imaxY = Mathf.Clamp(imaxY, 0, height - 1);
imaxZ = Mathf.Clamp(imaxZ, 0, depth - 1);

// 三角形の AABB の中でボクセルとの交差判定を行う
for(int x = iminX; x <= imaxX; x++) {
    for(int y = iminY; y <= imaxY; y++) {
        for(int z = iminZ; z <= imaxZ; z++) {
            if(Intersects(tri, boxes[x, y, z])) {
                ...
            }
        }
    }
}
```

三角形とボクセルとの交差判定は Intersects(Triangle, Bounds) 関数で行います。

▼ CPUVoxelizer.cs

```
public static bool Intersects(Triangle tri, Bounds aabb)
{
    ...
}
```

この関数内では、前述の 13 通りの軸について交差判定を行っていきますが、AABB の 3 つの法線が既知である（XYZ 軸に沿った辺を持つので、単に X 軸（1, 0, 0）、Y 軸（0, 1, 0）、Z 軸（0, 0, 1）の法線を持つ）ことを利用したり、AABB の中心が原点（0, 0, 0）に来るように三角形と AABB の座標を平行移動させることで、交差判定を最適化しています。

▼ CPUVoxelizer.cs

```
// AABB の中心座標と各辺の半分の長さ (extents) を取得
Vector3 center = aabb.center, extents = aabb.max - center;

// AABB の中心が原点 (0, 0, 0) に来るように三角形の座標を平行移動
Vector3 v0 = tri.a - center,
        v1 = tri.b - center,
        v2 = tri.c - center;

// 三角形の三辺を表すベクトルを取得
Vector3 f0 = v1 - v0,
        f1 = v2 - v1,
        f2 = v0 - v2;
```

まずは、三角形の 3 つの辺と AABB の直交する 3 つの辺同士の組み合わせから得られる 9 通りのクロス積を軸とした交差判定を行っていきますが、AABB の 3 つの辺の方向が XYZ 軸に平行であることを利用し、クロス積を得るための計算を省くことができます。

▼ CPUVoxelizer.cs

```
// AABB の辺はそれぞれ方向ベクトル x(1, 0, 0)、y(0, 1, 0)、z(0, 0, 1) であるので、
// 計算を行わずに 9 通りのクロス積を得ることができる
Vector3
    a00 = new Vector3(0, -f0.z, f0.y), // X 軸と f0 とのクロス積
    a01 = new Vector3(0, -f1.z, f1.y), // X と f1
    a02 = new Vector3(0, -f2.z, f2.y), // X と f2
    a10 = new Vector3(f0.z, 0, -f0.x), // Y と f0
    a11 = new Vector3(f1.z, 0, -f1.x), // Y と f1
    a12 = new Vector3(f2.z, 0, -f2.x), // Y と f2
    a20 = new Vector3(-f0.y, f0.x, 0), // Z と f0
    a21 = new Vector3(-f1.y, f1.x, 0), // Z と f1
    a22 = new Vector3(-f2.y, f2.x, 0); // Z と f2

// 9 つの軸について交差判定を行う（後述）
// （どれか一つでも交差しない軸があれば、三角形と AABB は交差していないので false を返
す）
if (
    !Intersects(v0, v1, v2, extents, a00) ||
    !Intersects(v0, v1, v2, extents, a01) ||
    !Intersects(v0, v1, v2, extents, a02) ||
    !Intersects(v0, v1, v2, extents, a10) ||
    !Intersects(v0, v1, v2, extents, a11) ||
    !Intersects(v0, v1, v2, extents, a12) ||
    !Intersects(v0, v1, v2, extents, a20) ||
    !Intersects(v0, v1, v2, extents, a21) ||
    !Intersects(v0, v1, v2, extents, a22)
)
{
    return false;
}
```

これらの軸上で三角形と AABB を射影して交差を判定するのが以下の関数です。

▼ CPUVoxelizer.cs

```
protected static bool Intersects(
    Vector3 v0,
    Vector3 v1,
    Vector3 v2,
    Vector3 extents,
    Vector3 axis
)
{
    ...
}
```

ここで注意したいのが、AABB の中心を原点に持ってくることによる最適化を行っていることです。AABB の各頂点を全て軸に射影する必要はなく、AABB の XYZ 軸について最大値を持つ頂点、つまり各辺の半分の長さ（extents）を射影するだけで軸上の区間を得ることができます。

extents を射影して得られた値 r は AABB の射影軸上の区間 [-r, r] を表すので、AABB については射影の計算が一回で済む、ということです。

▼ CPUVoxelizer.cs

```
// 三角形の頂点を軸上に射影する
float p0 = Vector3.Dot(v0, axis);
float p1 = Vector3.Dot(v1, axis);
float p2 = Vector3.Dot(v2, axis);

// AABB の XYZ 軸について最大値を持つ頂点（extents）を軸上に射影し、値 r を得る
// AABB の区間は [-r, r] であるので、AABB については全頂点を射影する必要はない
float r =
    extents.x * Mathf.Abs(axis.x) +
    extents.y * Mathf.Abs(axis.y) +
    extents.z * Mathf.Abs(axis.z);

// 三角形の射影区間
float minP = Mathf.Min(p0, p1, p2);
float maxP = Mathf.Max(p0, p1, p2);

// 三角形の区間と AABB の区間が重なっているかの判別
return !((maxP < -r) || (r < minP));
```

9 通りのクロス積を軸とした判別の次は、AABB の 3 つの法線を軸として判別を行います。

AABB の法線が XYZ 軸に平行であるという特性を利用し、AABB の中心を原点に持ってくるよう座標値を平行移動しているので、単に三角形の各頂点の XYZ 成分についての最小値・最大値と extents を比較するだけで交差判定が行えます。

▼ CPUVoxelizer.cs

```
// X 軸
if (
    Mathf.Max(v0.x, v1.x, v2.x) < -extents.x ||
    Mathf.Min(v0.x, v1.x, v2.x) > extents.x
)
{
    return false;
}

// Y 軸
if (
    Mathf.Max(v0.y, v1.y, v2.y) < -extents.y ||
    Mathf.Min(v0.y, v1.y, v2.y) > extents.y
)
{
    return false;
}

// Z 軸
if (
    Mathf.Max(v0.z, v1.z, v2.z) < -extents.z ||
    Mathf.Min(v0.z, v1.z, v2.z) > extents.z
)
{
    return false;
}
```

最後は三角形の法線についてですが、三角形の法線を持つ Plane と AABB との交差について判定を行っています。

▼ CPUVoxelizer.cs

```
var normal = Vector3.Cross(f1, f0).normalized;
var pl = new Plane(normal, Vector3.Dot(normal, tri.a));
return Intersects(pl, aabb);
```

Intersects(Plane, Bounds) 関数で Plane と AABB との交差を判定します。

▼ CPUVoxelizer.cs

```
public static bool Intersects(Plane pl, Bounds aabb)
{
    Vector3 center = aabb.center;
    var extents = aabb.max - center;

    // Plane の normal 上に extents を射影
    var r =
        extents.x * Mathf.Abs(pl.normal.x) +
        extents.y * Mathf.Abs(pl.normal.y) +
        extents.z * Mathf.Abs(pl.normal.z);

    // Plane と AABB の中心との距離を計算
    var s = Vector3.Dot(pl.normal, center) - pl.distance;

    // s が [-r, r] の範囲にあるかどうか判定
```

```
    return Mathf.Abs(s) <= r;
}
```

交差したボクセルを配列データに書き込む

一つの三角形について、交差するボクセルを判定できると、ボクセルデータの fill フラグを立て、その三角形が決められた方向から見て前面か背面かを示す front フラグを設定します。(front フラグについては後述)

ボクセルによっては前面を向いている三角形と背面を向いている三角形両方と交差している場合がありますが、その場合、front フラグは背面を優先するようにします。

▼ CPUVoxelizer.cs

```
if(Intersects(tri, boxes[x, y, z])) {
    // 交差した (x, y, z) にあるボクセルを取得する
    var voxel = volume[x, y, z];

    // ボクセルの位置を設定する
    voxel.position = boxes[x, y, z].center;

    if(voxel.fill & 1 == 0) {
        // ボクセルがまだ埋まっていない場合
        // ボクセルと交差した三角形が前面かどうかのフラグを立てる
        voxel.front = front;
    } else {
        // ボクセルが既にほかの三角形で埋められている場合
        // 背面のフラグを優先する
        voxel.front = voxel.front & front;
    }

    // ボクセルを埋めるフラグを立てる
    voxel.fill = 1;
    volume[x, y, z] = voxel;
}
```

front フラグは後述の「メッシュの中身を埋める処理」に必要で、「中身を埋めていく方向」から見て前面か背面かを設定します。

サンプルコードでは、forward(0, 0, 1) 方向にメッシュの中身を埋めていくので、三角形が forward(0, 0, 1) から見て前面かどうかを判定します。

3 角形の法線とボクセルを埋める方向との内積が 0 以下であれば、3 角形はその方向から見て前面であることがわかります。

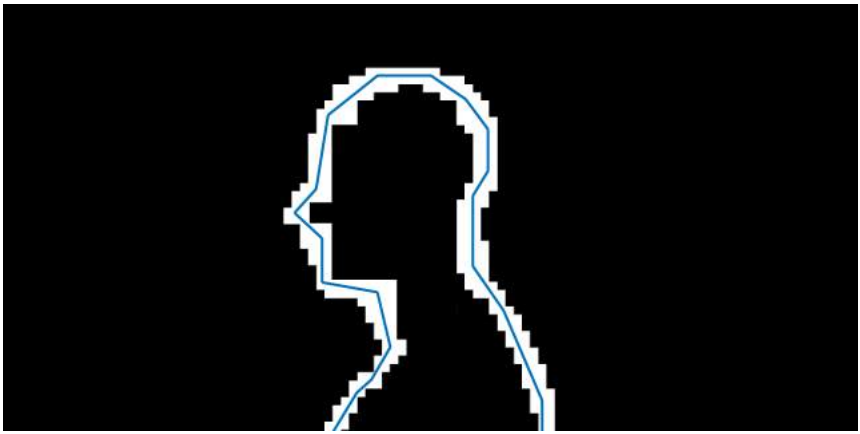
▼ CPUVoxelizer.cs

```
public class Triangle {
    public Vector3 a, b, c;     // 3 角形を構成する 3 点
    public bool frontFacing;    // 3 角形がボクセルを埋める方向から見て表面かどう
かのフラグ
    public Bounds bounds;       // 3 角形の AABB
```

```
    public Triangle (Vector3 a, Vector3 b, Vector3 c, Vector3 dir) {
        this.a = a;
        this.b = b;
        this.c = c;

        // 3 角形がボクセルを埋める方向から見て前面かどうかを判定する
        var normal = Vector3.Cross(b - a, c - a);
        this.frontFacing = (Vector3.Dot(normal, dir) <= 0f);

        ...
    }
}
```

1.2.6 メッシュの表面を表すボクセルデータから、メッシュ内部に位置するボクセルを埋める

メッシュ表面に位置するボクセルデータが計算できたので、次はその内部を埋めていきます。

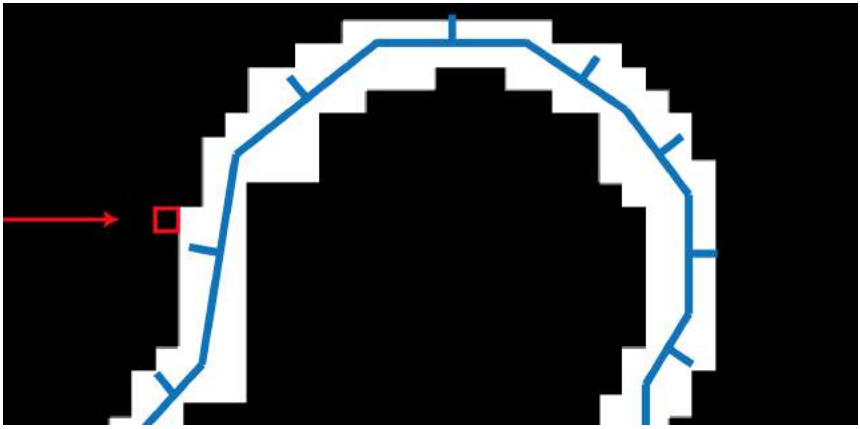


▲図 1.8　メッシュ表面に位置するボクセルデータを生成した後の状態

ボクセルを埋める流れ

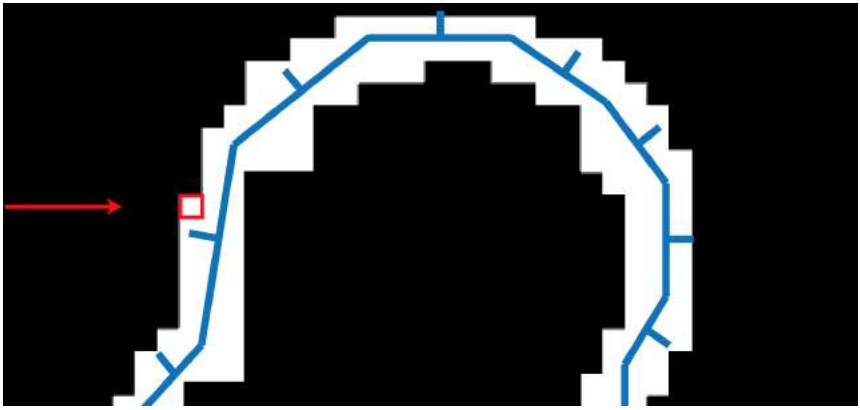
ボクセルを埋める方向からみて前面を向いているボクセルを探索します。

以下の図のように空のボクセルは素通りします。

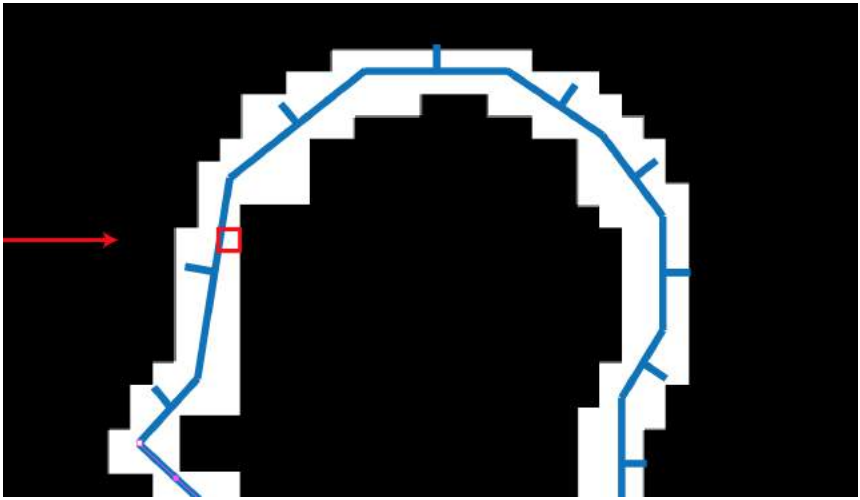


▲図 1.9　ボクセルを埋める方向からみて前面を向いているボクセルを探索する　空のボクセルは素通りする（矢印がボクセルを埋める方向で、枠が探索中のボクセル位置を表す）

前面を向いているボクセルを見つけたら、前面を向いているボクセルの中を進んでいきます。

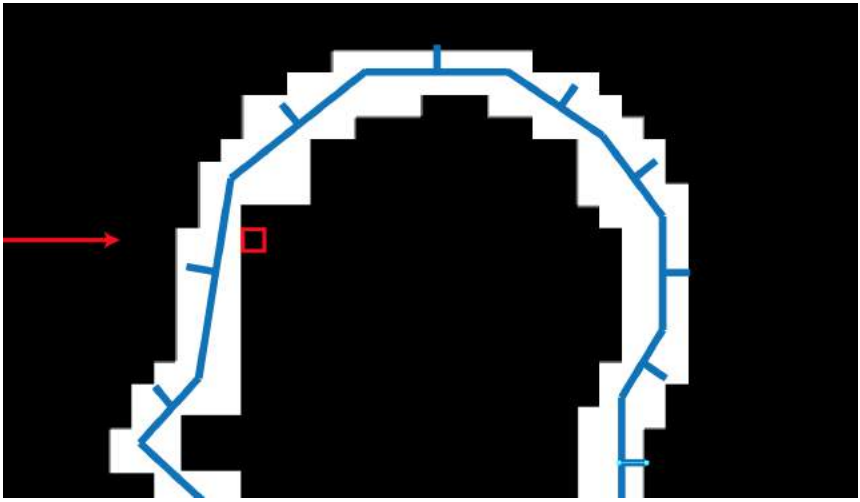


▲図 1.10　前面を向いているボクセルを見つけた状態（メッシュ表面から出ている線はメッシュの法線であり、図ではメッシュの法線とボクセルを埋める方向が向かいあっているので、枠の位置のボクセルは前面に位置していることがわかる）



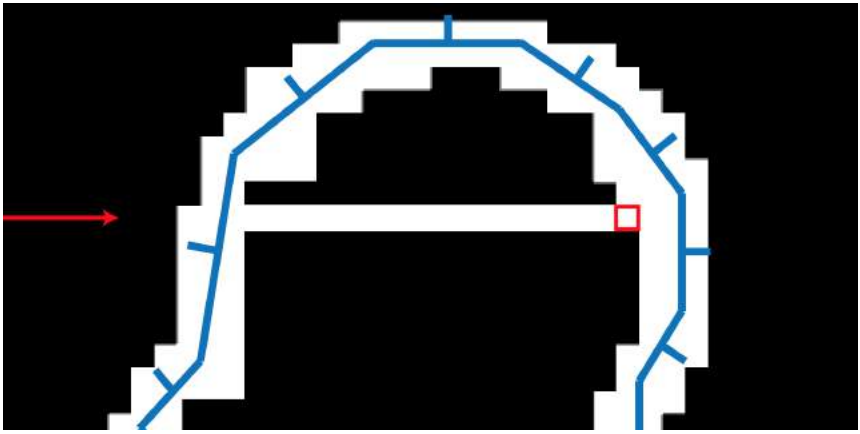
▲図 1.11　前面を向いているボクセルの中を進む

前面を向いているボクセルの中を抜けると、メッシュ内部に到達します。



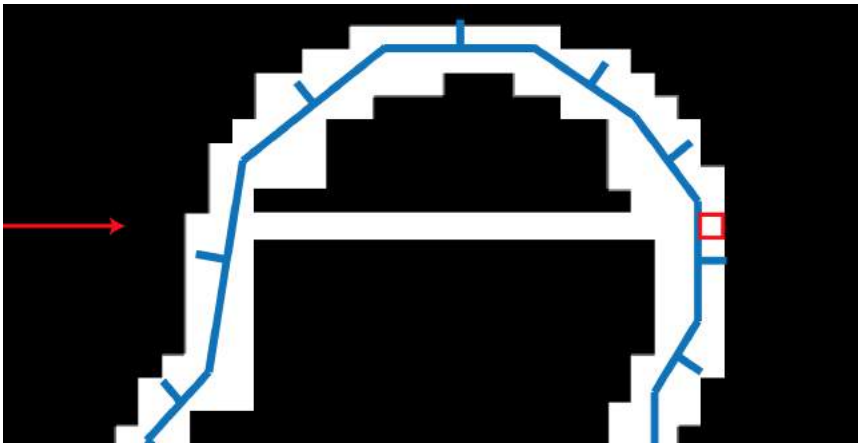
▲図 1.12　前面を向いているボクセルの中を抜け、メッシュ内部に到達した状態

メッシュ内部の中を進んでいき、到達したボクセルを埋めていきます。



▲図 1.13　メッシュ内部を埋めていくように、到達したボクセルを埋めていく

そして、ボクセルを埋める方向からみて背面を向いているボクセルに到達すると、メッシュの内部を埋めきったことがわかります。背面を向いたボクセルの中を進んでいき、またメッシュの外に到達すると、また前面を向いているボクセルを探索しはじめます。



▲図 1.14　ボクセルを埋める方向から背面を向いているボクセルの中を進んでいき、またメッシュの外に到達する

ボクセルを埋める実装

前項で決めたように forward(0, 0, 1) 方向に向かって内部を埋めていくので、3 次元ボクセル配列では z 方向に向かって内部を埋めていくことになります。

z 方向の一番手前側、volume[x, y, 0] からはじめて volume[x, y, depth - 1] まで中身を埋める処理を進めます。

▼ CPUVoxelizer.cs

```
// メッシュの内部を埋める
for(int x = 0; x < width; x++)
{
    for(int y = 0; y < height; y++)
    {
        // z 方向の手前側から奥に向かってメッシュ内部を埋めていく
        for(int z = 0; z < depth; z++)
        {
            ...
        }
    }
}
```

既にボクセルデータに書き込まれてある front フラグ（z 方向に向かって前面か背面か）を基に、前述のボクセルを埋める流れに沿って処理を進めます。

▼ CPUVoxelizer.cs

```
...
// z 方向の手前側から奥に向かってメッシュ内部を埋めていく
for(int z = 0; z < depth; z++)
{
    // (x, y, z) が空の場合は無視
    if (volume[x, y, z].IsEmpty()) continue;

    // 前面に位置するボクセルを進む
    int ifront = z;
    for(; ifront < depth && volume[x, y, ifront].IsFrontFace(); ifront++) {}

    // 最後までいけば終わり
    if(ifront >= depth) break;

    // 背面に位置するボクセルを探す
    int iback = ifront;

    // メッシュ内部を進んでいく
    for (; iback < depth && volume[x, y, iback].IsEmpty(); iback++) {}

    // 最後までいけば終わり
    if (iback >= depth) break;

    // (x, y, iback) が背面かどうかを判定
    if(volume[x, y, iback].IsBackFace()) {
        // 背面に位置するボクセルを進む
```

```
            for (; iback < depth && volume[x, y, iback].IsBackFace(); iback++) {}
        }

        // (x, y, ifront) から (x, y, iback) の位置までボクセルを埋める
        for(int z2 = ifront; z2 < iback; z2++)
        {
            var p = boxes[x, y, z2].center;
            var voxel = volume[x, y, z2];
            voxel.position = p;
            voxel.fill = 1;
            volume[x, y, z2] = voxel;
        }

        // 処理し終えた (x, y, iback) までループを進める
        z = iback;
    }
```

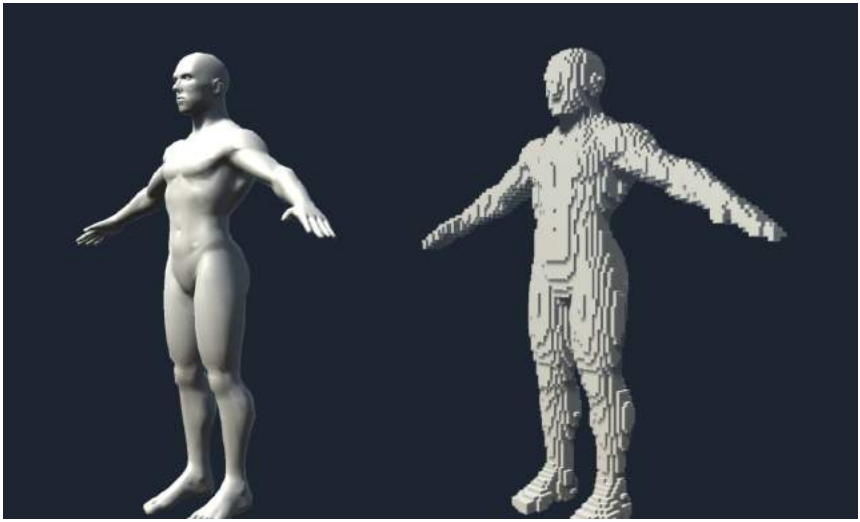
ここまででメッシュの中身を充填したボクセルデータを得ることができました。

処理し終えた 3 次元のボクセルデータの中には、空のボクセルが含まれているため、CPUVoxelizer.Voxelize ではメッシュの表面と充填された中身を構成するボクセルのみを返すようにしています。

▼ CPUVoxelizer.cs

```
// 空でないボクセルを取得する
voxels = new List<Voxel_t>();
for(int x = 0; x < width; x++) {
    for(int y = 0; y < height; y++) {
        for(int z = 0; z < depth; z++) {
            if(!volume[x, y, z].IsEmpty())
            {
                voxels.Add(volume[x, y, z]);
            }
        }
    }
}
```

CPUVoxelizerTest.cs で、CPUVoxelizer で得たボクセルデータを用いてメッシュを構築し、ボクセルを可視化しています。



▲図 1.15　CPUVoxelizer.Voxelize で得たボクセルデータを Mesh にして可視化したデモ（CPUVoxelizerTest.scene）

1.3　ボクセルのメッシュ表現

ボクセルデータの配列 Voxel_t[] と、一つのボクセルの単位長さの情報を基にメッシュを構築する処理を VoxelMesh クラスに記述しています。

前節の CPUVoxelizerTest.cs ではこのクラスを用いてボクセルメッシュの生成を行っています。

▼ VoxelMesh.cs

```
public class VoxelMesh {

    public static Mesh Build (Voxel_t[] voxels, float size)
    {
        var hsize = size * 0.5f;
        var forward = Vector3.forward * hsize;
        var back = -forward;
        var up = Vector3.up * hsize;
        var down = -up;
        var right = Vector3.right * hsize;
        var left = -right;

        var vertices = new List<Vector3>();
        var normals = new List<Vector3>();
```

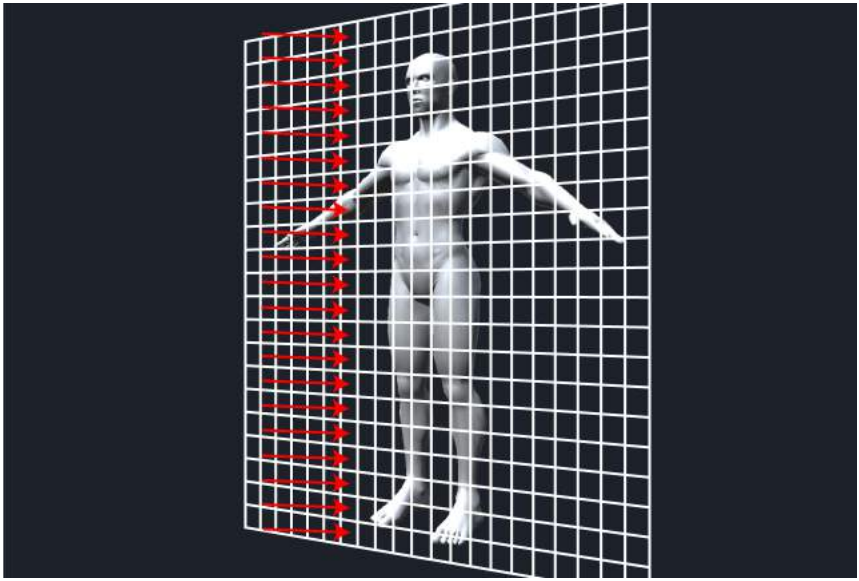
```
var triangles = new List<int>();

for(int i = 0, n = voxels.Length; i < n; i++)
{
    if(voxel[i].fill == 0) continue;

    var p = voxels[i].position;

    // 一つのボクセルを表現する Cube を構成する 8 隅の頂点
    var corners = new Vector3[8] {
        p + forward + left + up,
        p + back + left + up,
        p + back + right + up,
        p + forward + right + up,

        p + forward + left + down,
        p + back + left + down,
        p + back + right + down,
        p + forward + right + down,
    };

    // Cube を構成する 6 面を構築する

    // up
    AddTriangle(
        corners[0], corners[3], corners[1],
        up, vertices, normals, triangles
    );
    AddTriangle(
        corners[2], corners[1], corners[3],
        up, vertices, normals, triangles
    );

    // down
    AddTriangle(
        corners[4], corners[5], corners[7],
        down, vertices, normals, triangles
    );
    AddTriangle(
        corners[6], corners[7], corners[5],
        down, vertices, normals, triangles
    );

    // right
    AddTriangle(
        corners[7], corners[6], corners[3],
        right, vertices, normals, triangles
    );
    AddTriangle(
        corners[2], corners[3], corners[6],
        right, vertices, normals, triangles
    );

    // left
    AddTriangle(
        corners[5], corners[4], corners[1],
        left, vertices, normals, triangles
    );
```

```
            AddTriangle(
                corners[0], corners[1], corners[4],
                left, vertices, normals, triangles
            );

            // forward
            AddTriangle(
                corners[4], corners[7], corners[0],
                forward, vertices, normals, triangles
            );
            AddTriangle(
                corners[3], corners[0], corners[7],
                forward, vertices, normals, triangles
            );

            // back
            AddTriangle(
                corners[6], corners[5], corners[2],
                forward, vertices, normals, triangles
            );
            AddTriangle(
                corners[1], corners[2], corners[5],
                forward, vertices, normals, triangles
            );
        }

        var mesh = new Mesh();
        mesh.SetVertices(vertices);

        // 頂点数が 16bit でサポートできる数を超えていたら 32bit index format を適
用する
        mesh.indexFormat =
            (vertices.Count <= 65535)
            ? IndexFormat.UInt16 : IndexFormat.UInt32;
        mesh.SetNormals(normals);
        mesh.SetIndices(triangles.ToArray(), MeshTopology.Triangles, 0);
        mesh.RecalculateBounds();
        return mesh;
    }
}
```

1.4 GPU での実装

ここからは GPU を用いて CPUVoxelizer で実装したボクセル化をより高速に実行する方法について解説します。

CPUVoxelizer で実装したボクセル化のアルゴリズムは、XY 平面上でボクセルの単位長さで区切られた格子空間上の各座標ごとに並列化することができます。



▲図 1.16　XY 平面上でボクセルの単位長さで区切られた格子空間　ボクセル化はこの各格子ごとに並列化できるので GPU 実装が可能

並列化できるそれぞれの処理を GPU スレッドに割り振れば、GPU の高速な並列計算の恩恵により、高速に処理が実行できます。

GPU でのボクセル化の実装は GPUVoxelizer.cs と Voxelizer.compute に記述しています。

（本節から登場する、Unity で GPGPU プログラミングをする上で欠かせない ComputeShader の基本については Unity Graphics Programming vol.1「Compute-Shader 入門」で解説しています）

GPU でのボクセル化は、GPUVoxelizer クラスの static 関数

▼ GPUVoxelizer.cs

```
public class GPUVoxelizer
{
    public static GPUVoxelData Voxelize (
        ComputeShader voxelizer,
        Mesh mesh,
        int resolution
    ) {
    ...
    }
}
```

をコールすることで実行します。引数に Voxelizer.compute、ボクセル化したいメッシュ、解像度を指定して実行すると、ボクセルデータを示す GPUVoxelData を返します。

1.4.1 GPU でのボクセル生成に必要なデータのセットアップ

ボクセル化の大まかな流れ (1) ~ (3) と同じようにして、ボクセル生成に必要なデータのセットアップを行います。

▼ GPUVoxelizer.cs

```
public static GPUVoxelData Voxelize (
    ComputeShader voxelizer,
    Mesh mesh,
    int resolution
) {
    // CPUVoxelizer.Voxelize と同じ処理 -------
    mesh.RecalculateBounds();
    var bounds = mesh.bounds;

    float maxLength = Mathf.Max(
        bounds.size.x,
        Mathf.Max(bounds.size.y, bounds.size.z)
    );
    var unit = maxLength / resolution;

    var hunit = unit * 0.5f;

    var start = bounds.min - new Vector3(hunit, hunit, hunit);
    var end = bounds.max + new Vector3(hunit, hunit, hunit);
    var size = end - start;

    int width = Mathf.CeilToInt(size.x / unit);
    int height = Mathf.CeilToInt(size.y / unit);
    int depth = Mathf.CeilToInt(size.z / unit);
    // ------- ここまで CPUVoxelizer.Voxelize と同じ
    ...
}
```

Voxel_t の配列は、GPU 上で扱えるようにするために ComputeBuffer として定義します。ここで注意したいのが、CPU 実装だと 3 次元配列として生成した Voxel_t 配列を 1 次元配列として定義していることです。

これは、GPU では多次元配列を扱うのが困難なためで、1 次元配列として定義しておき、ComputeShader 内では 3 次元上の位置 (x, y, z) から 1 次元配列上の index を取得することで、1 次元配列を 3 次元配列のように処理しています。

▼ GPUVoxelizer.cs

```
// Voxel_t 配列を表す ComputeBuffer を生成
var voxelBuffer = new ComputeBuffer(
```

```
    width * height * depth,
    Marshal.SizeOf(typeof(Voxel_t))
);
var voxels = new Voxel_t[voxelBuffer.count];
voxelBuffer.SetData(voxels); // 初期化
```

これらセットアップしたデータを GPU 側に転送します。

▼ GPUVoxelizer.cs

```
// ボクセルデータを GPU 側に転送
voxelizer.SetVector("_Start", start);
voxelizer.SetVector("_End", end);
voxelizer.SetVector("_Size", size);

voxelizer.SetFloat("_Unit", unit);
voxelizer.SetFloat("_InvUnit", 1f / unit);
voxelizer.SetFloat("_HalfUnit", hunit);
voxelizer.SetInt("_Width", width);
voxelizer.SetInt("_Height", height);
voxelizer.SetInt("_Depth", depth);
```

メッシュを構成する三角形とボクセル同士の交差判定を行うため、メッシュを表す ComputeBuffer を生成します。

▼ GPUVoxelizer.cs

```
// メッシュの頂点配列を表す ComputeBuffer を生成
var vertices = mesh.vertices;
var vertBuffer = new ComputeBuffer(
    vertices.Length,
    Marshal.SizeOf(typeof(Vector3))
);
vertBuffer.SetData(vertices);

// メッシュの三角形配列を表す ComputeBuffer を生成
var triangles = mesh.triangles;
var triBuffer = new ComputeBuffer(
    triangles.Length,
    Marshal.SizeOf(typeof(int))
);
triBuffer.SetData(triangles);
```

1.4.2 GPU でメッシュの表面に位置するボクセルを生成する

GPU でメッシュの表面に位置するボクセルを生成する処理では、前面を向いている三角形と交差しているボクセルを生成した後で、背面を向いている三角形と交差しているボクセルを生成します。

これは、同じ位置のボクセルに対して複数の三角形が交差している場合に、ボクセルに書き込まれる front フラグの値が一意に定まらない恐れがあるためです。

GPU の並列計算で気をつけないといけないのは、同じデータに複数のスレッドが同時にアクセスしてしまうことによる、結果の不定性です。

この表面を生成する処理においては、front フラグの値が背面（false）であることを優先し、前面→背面という順でボクセル生成を実行することで結果の不定性を防いでいます。

前面を向いている三角形と交差しているボクセルを生成する GPU カーネル SurfaceFront に、先ほど生成したメッシュデータを転送します。

▼ GPUVoxelizer.cs

```
// GPU カーネル SurfaceFront にメッシュデータを転送
var surfaceFrontKer = new Kernel(voxelizer, "SurfaceFront");
voxelizer.SetBuffer(surfaceFrontKer.Index, "_VoxelBuffer", voxelBuffer);
voxelizer.SetBuffer(surfaceFrontKer.Index, "_VertBuffer", vertBuffer);
voxelizer.SetBuffer(surfaceFrontKer.Index, "_TriBuffer", triBuffer);

// メッシュを構成する三角形の数を設定
var triangleCount = triBuffer.count / 3; // (三角形を構成する頂点 index の
数 / 3) が三角形の数
voxelizer.SetInt("_TriangleCount", triangleCount);
```

この処理はメッシュを構成する三角形ごとに並列に実行します。全ての三角形が処理されるように、カーネルのスレッドグループを (三角形の数 triangleCount / カーネルのスレッド数 + 1, 1, 1) に設定し、カーネルを実行します。

▼ GPUVoxelizer.cs

```
// 前面を向いている三角形と交差するボクセルを構築
voxelizer.Dispatch(
    surfaceFrontKer.Index,
    triangleCount / (int)surfaceFrontKer.ThreadX + 1,
    (int)surfaceFrontKer.ThreadY,
    (int)surfaceFrontKer.ThreadZ
);
```

SurfaceFront カーネルは前面を向いている三角形のみを処理するため、三角形の前面背面をチェックし、背面である場合はそのまま処理を終了するように return し、前面である場合はメッシュ表面を構築する surface 関数を実行しています。

▼ Voxelizer.compute

```
[numthreads(8, 1, 1)]
void SurfaceFront (uint3 id : SV_DispatchThreadID)
{
    // 三角形の数を超えていると return
    int idx = (int)id.x;
    if(idx >= _TriangleCount) return;

    // 三角形の頂点位置と前面背面フラグを取得
```

```
    float3 va, vb, vc;
    bool front;
    get_triangle(idx, va, vb, vc, front);

    // 背面である場合は return
    if (!front) return;

    // メッシュ表面を構築
    surface(va, vb, vc, front);
}
```

get_triangle 関数は、CPU から GPU 側に渡されたメッシュデータ（三角形を構成する頂点 index を表す_TriBuffer と頂点を表す_VertBuffer）に基づいて、三角形の頂点位置と前面背面フラグを取得します。

▼ Voxelizer.compute

```
void get_triangle(
    int idx,
    out float3 va, out float3 vb, out float3 vc,
    out bool front
)
{
    int ia = _TriBuffer[idx * 3];
    int ib = _TriBuffer[idx * 3 + 1];
    int ic = _TriBuffer[idx * 3 + 2];

    va = _VertBuffer[ia];
    vb = _VertBuffer[ib];
    vc = _VertBuffer[ic];

    // 三角形が forward(0, 0, 1) 方向から見て前面か背面かを判断
    float3 normal = cross((vb - va), (vc - vb));
    front = dot(normal, float3(0, 0, 1)) < 0;
}
```

ボクセルと三角形との交差判定を行い、その結果をボクセルデータに書き込む surface 関数は、1 次元配列として生成したボクセルデータの index を取得する手間があれど、その処理の内容は CPUVoxelizer 上に実装したものとほぼ同じものになります。

▼ Voxelizer.compute

```
void surface (float3 va, float3 vb, float3 vc, bool front)
{
    // 三角形の AABB を計算
    float3 tbmin = min(min(va, vb), vc);
    float3 tbmax = max(max(va, vb), vc);

    float3 bmin = tbmin - _Start;
    float3 bmax = tbmax - _Start;
    int iminX = round(bmin.x / _Unit);
```

```
    int iminY = round(bmin.y / _Unit);
    int iminZ = round(bmin.z / _Unit);
    int imaxX = round(bmax.x / _Unit);
    int imaxY = round(bmax.y / _Unit);
    int imaxZ = round(bmax.z / _Unit);
    iminX = clamp(iminX, 0, _Width - 1);
    iminY = clamp(iminY, 0, _Height - 1);
    iminZ = clamp(iminZ, 0, _Depth - 1);
    imaxX = clamp(imaxX, 0, _Width - 1);
    imaxY = clamp(imaxY, 0, _Height - 1);
    imaxZ = clamp(imaxZ, 0, _Depth - 1);

    // 三角形の AABB の中でボクセルとの交差判定を行う
    for(int x = iminX; x <= imaxX; x++) {
        for(int y = iminY; y <= imaxY; y++) {
            for(int z = iminZ; z <= imaxZ; z++) {
                // (x, y, z) に位置するボクセルの AABB を生成
                float3 center = float3(x, y, z) * _Unit + _Start;
                AABB aabb;
                aabb.min = center - _HalfUnit;
                aabb.center = center;
                aabb.max = center + _HalfUnit;
                if(intersects_tri_aabb(va, vb, vc, aabb))
                {
                    // (x, y, z) の位置から 1 次元のボクセル配列の index を取得
                    uint vid = get_voxel_index(x, y, z);
                    Voxel voxel = _VoxelBuffer[vid];
                    voxel.position = get_voxel_position(x, y, z);
                    voxel.front = front;
                    voxel.fill = true;
                    _VoxelBuffer[vid] = voxel;
                }
            }
        }
    }
}
```

これで前面を向いている三角形についてボクセルが生成できたので、次は背面を向いている三角形について処理していきます。

背面を向いている三角形と交差するボクセルを生成する GPU カーネル SurfaceBack に先程と同じようにメッシュデータを転送し、実行します。

▼ GPUVoxelizer.cs

```
var surfaceBackKer = new Kernel(voxelizer, "SurfaceBack");
voxelizer.SetBuffer(surfaceBackKer.Index, "_VoxelBuffer", voxelBuffer);
voxelizer.SetBuffer(surfaceBackKer.Index, "_VertBuffer", vertBuffer);
voxelizer.SetBuffer(surfaceBackKer.Index, "_TriBuffer", triBuffer);
voxelizer.Dispatch(
    surfaceBackKer.Index,
    triangleCount / (int)surfaceBackKer.ThreadX + 1,
    (int)surfaceBackKer.ThreadY,
    (int)surfaceBackKer.ThreadZ
);
```

SurfaceBack の処理は、三角形が前面を向いている場合に return を返す以外は SurfaceFront と同じです。SurfaceFront の後に SurfaceBack を実行することによって、もし前面を向いている三角形と背面を向いている三角形両方と交差しているボクセルが存在していても、ボクセルの front フラグが SurfaceBack によって上書きされる形になり、背面を向いていることが優先されるようになります。

▼ Voxelizer.compute

```
[numthreads(8, 1, 1)]
void SurfaceBack (uint3 id : SV_DispatchThreadID)
{
    int idx = (int)id.x;
    if(idx >= _TriangleCount) return;

    float3 va, vb, vc;
    bool front;
    get_triangle(idx, va, vb, vc, front);

    // 前面である場合は return
    if (front) return;

    surface(va, vb, vc, front);
}
```

1.4.3 GPU でメッシュの表面を表すボクセルデータから、メッシュ内部に位置するボクセルを埋める

メッシュの内部を埋める処理は Volume カーネルで行います。

Volume カーネルは XY 平面上でボクセルの単位長さで区切られた格子空間上の各座標ごとにスレッドを用意して実行します。つまり、CPU 実装だと XY 座標について二重ループを実行していたところを GPU で並列化し、高速化しているということになります。

▼ GPUVoxelizer.cs

```
// Volume カーネルにボクセルデータを転送
var volumeKer = new Kernel(voxelizer, "Volume");
voxelizer.SetBuffer(volumeKer.Index, "_VoxelBuffer", voxelBuffer);

// メッシュ内部を埋める
voxelizer.Dispatch(
    volumeKer.Index,
    width / (int)volumeKer.ThreadX + 1,
    height / (int)volumeKer.ThreadY + 1,
    (int)volumeKer.ThreadZ
);
```

Volume カーネルの実装は GPUVoxelizer に実装したものとほぼ同じものになり

ます。

▼ Voxelizer.compute

```
[numthreads(8, 8, 1)]
void Volume (uint3 id : SV_DispatchThreadID)
{
    int x = (int)id.x;
    int y = (int)id.y;
    if(x >= _Width) return;
    if(y >= _Height) return;

    for (int z = 0; z < _Depth; z++)
    {
        Voxel voxel = _VoxelBuffer[get_voxel_index(x, y, z)];
        // CPUVoxelizer.Voxelize 内の処理とほぼ同じ処理が続く
        ...
    }
}
```

このようにしてボクセルデータが得られると、不要になったメッシュデータを破棄し、ボクセルのビジュアル表現をつくる際に必要なデータを持った GPUVoxelData を生成します。

▼ GPUVoxelizer.cs

```
// 不要になったメッシュデータを破棄
vertBuffer.Release();
triBuffer.Release();

return new GPUVoxelData(voxelBuffer, width, height, depth, unit);
```

これで GPU 実装によるボクセル化が完了しました。GPUVoxelizerTest.cs で実際に GPUVoxelData を用いてボクセルデータを可視化しています。

1.5 CPU 実装と GPU 実装の速度差

テスト用の Scene では Play 時に Voxelizer を実行しているので、CPU 実装と GPU 実装の速度差がわかりにくいですが、GPU 実装でかなりの高速化を実現しています。

実行環境と、ボクセル化する対象のメッシュのポリゴン数、ボクセル化の解像度にパフォーマンスは大きく依存しますが、

- 実行環境 OS: Windows10、CPU: Core i7、メモリ: 32GB、GPU: GeForce GTX 980
- 頂点数 5319、三角形数 9761 のメッシュ
- ボクセル化の解像度 256

という条件では、GPU 実装は CPU 実装の 50 倍以上高速に動作しています。

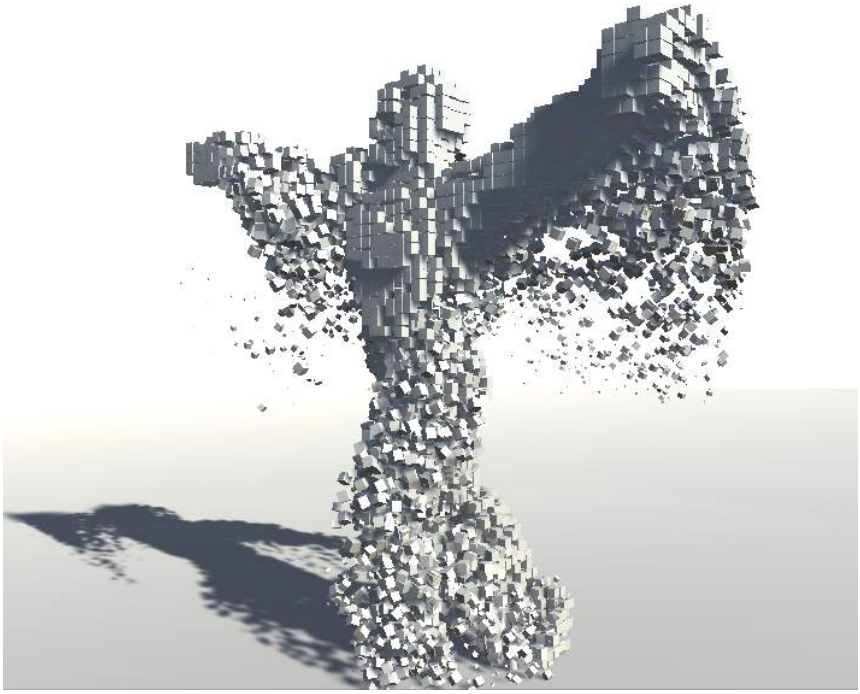
1.6 応用例

GPU 実装の ParticleSystem を利用した応用例（GPUVoxelParticleSystem）を紹介します。

GPUVoxelParticleSystem は GPUVoxelizer から得られたボクセルデータを表す ComputeBuffer を、ComputeShader でのパーティクルの位置計算に利用します。

1. GPUVoxelizer でアニメーションモデルを毎フレームボクセル化
2. GPUVoxelData が持つ ComputeBuffer を、パーティクルの位置計算を行う ComputeShader に渡す
3. パーティクルを GPU インスタンシングで描画

という流れでエフェクトを作成しています。



▲図 1.17　GPU 実装の ParticleSystem を利用した応用例（GPUVoxelParticleSystem）

大量のパーティクルをボクセルの位置から出現させることで、パーティクルで構成されるアニメーションモデルのようなビジュアルを実現しています。

アニメーションモデルに対してフレーム毎にボクセル化を施せるのは、GPU 実装による高速化があってこそで、リアルタイムで利用できるビジュアル表現の幅を広げるためにも、こうした GPU での高速化は欠かせないものになっています。

1.7 まとめ

本章では、メッシュモデルのボクセル化を行うアルゴリズムを CPU 実装を例に紹介し、GPU 実装によってボクセル化を高速化するところまで行いました。

三角形とボクセルとの交差判定を用いてボクセルを生成するアプローチをとりましたが、平行投影によってモデルを XYZ 方向から 3D テクスチャにレンダリングしていくことでボクセルデータを構築する方法もあります。

本章で紹介した手法だと、ボクセル化後のモデルにどうテクスチャを貼るかといった点に課題がありますが、3D テクスチャにモデルをレンダリングする手法であれば、ボクセルへの色付けはより手軽かつ正確に実現できるかもしれません。

1.8 参考

- http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization
- http://www.dyn4j.org/2010/01/sat/
- https://gdbooks.gitbooks.io/3dcollisions/content/Chapter4/aabb-triangle.html
- ゲームエンジン・アーキテクチャ第 2 版 第 12 章
- https://developer.nvidia.com/content/basics-gpu-voxelization

第 2 章

GPU-Based Trail

2.1 はじめに

本章では、GPU を活用して Trail（軌跡）を作る方法を紹介します。本章のサンプルは `https://github.com/IndieVisualLab/UnityGraphicsProgramming2` の「GPUBasedTrail」です。

2.1.1 Trail（軌跡）とは

移動する物体の軌跡を Trail と呼んでいます。広義の意味では車の轍や船の航跡、スキーのシュプールなども含みますが、CG で印象的なものは車のテールランプやシューティングゲームのホーミングレーザーのような曲線を描く光跡表現ではないでしょうか。

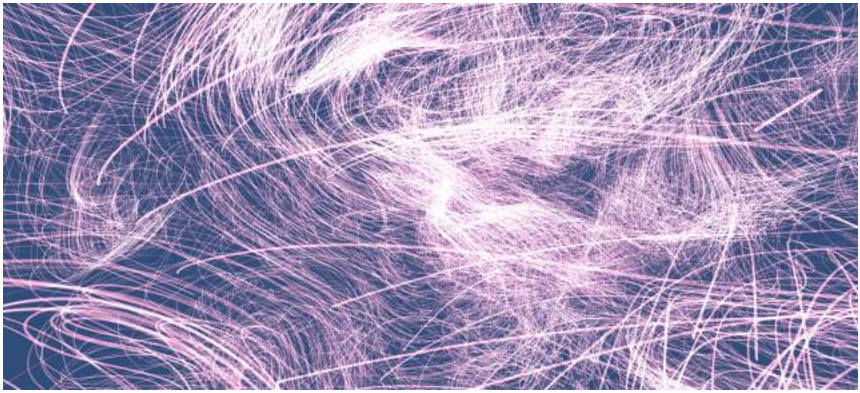
2.1.2 Unity 標準の Trail

Unity には標準で 2 種類の Trail が用意されています。

- **TrailRenderer**[*1] GameObject の軌跡を描くために使われます
- **Trails module**[*2] Particle の軌跡を描くために使われます

本章では Trail 自体の作り方に焦点を当てるため、あえてこれらの機能は使わず、また GPU 上での実装にすることで Trails module 以上の物量表現を可能にします。

[*1] https://docs.unity3d.com/ja/current/Manual/class-TrailRenderer.html
[*2] https://docs.unity3d.com/Manual/PartSysTrailsModule.html



▲図 2.1　サンプルコードの実行画面。10000 本の Trail を表示

2.2　データの作成

それでは Trail を作成していきましょう。

2.2.1　データの定義

使用する構造体は主に 3 つです。

▼ GPUTrails.cs

```
public struct Trail
{
  public int currentNodeIdx;
}
```

Trail 構造体は 1 つが 1 本の Trail に対応します。currentNodeIdx 最後に書き込みした Node バッファのインデックスを保存しています。

▼ GPUTrails.cs

```
public struct Node
{
  public float time;
  public Vector3 pos;
}
```

Node 構造体は Trail 内の制御点です。Node の位置と更新した時間を保存しています。

▼ GPUTrails.cs

```
public struct Input
{
  public Vector3 pos;
}
```

Input 構造体はエミッタ（軌跡を残すもの）からの 1 フレーム分の入力です。ここでは位置だけですが、色などを追加しても面白いと思います。

2.2.2 初期化

GPUTrails.Start() で使用するバッファを初期化していきます

▼ GPUTrails.cs

```
trailBuffer = new ComputeBuffer(trailNum, Marshal.SizeOf(typeof(Trail)));
nodeBuffer = new ComputeBuffer(totalNodeNum, Marshal.SizeOf(typeof(Node)));
inputBuffer = new ComputeBuffer(trailNum, Marshal.SizeOf(typeof(Input)));
```

trailNum 個分の trailBuffer を初期化しています。つまりこのプログラムでは複数本の Trail をまとめて処理しています。nodeBuffer ではすべての Trail 分の Node をまとめて 1 つのバッファで扱っています。インデックス 0 ～ nodeNum-1 までが 1 本目、nodeNum ～ 2*nodeNum-1 までが 2 本目、といった具合です。inputBuffer も trailNum 個保持し、全 Trail の入力を管理します。

▼ GPUTrails.cs

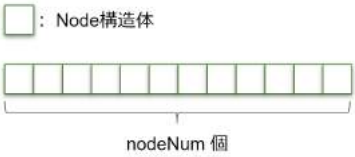
```
var initTrail = new Trail() { currentNodeIdx = -1 };
var initNode = new Node() { time = -1 };

trailBuffer.SetData(Enumerable.Repeat(initTrail, trailNum).ToArray());
nodeBuffer.SetData(Enumerable.Repeat(initNode, totalNodeNum).ToArray());
```

各バッファに初期値を入れています。Trail.currentNodeIdx、Node.time を負数にしておき、あとでこれらを未使用かどうかの判定に使います。inputBuffer は最初の更新ですべて値が書き込まれるので初期化の必要がなくノータッチです。

2.2.3 Node バッファの使い方

ここで Node バッファの使い方について解説します。

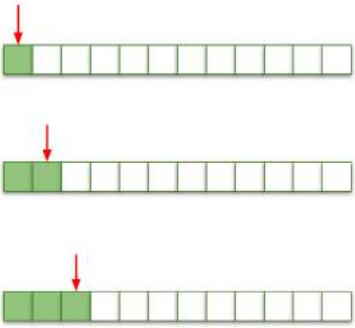
初期状態



▲図 2.2　初期状態

まだ何も入力されていない状態です。

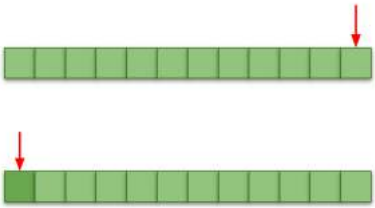
入力中



▲図 2.3　入力中

1 ノードづつ入力されて行きます。まだ未使用の Node があります。

ループ



▲図 2.4　ループ

すべての Node を使い尽くすと、はじめに戻りの Node を上書きして行きます。リングバッファ状に使用しています。

2.2.4　インプット

ここからは毎フレーム呼ばれる処理になります。エミッタの位置を入力して、Node を追加、更新していきます。

まずは外部で inputBuffer を更新していきます。これはどんな処理でもかまいません。はじめは CPU で計算して `ComputeBuffer.SetData()` するのが簡単で良いかもしれません。サンプルコードでは簡単な GPU 実装のパーティクルを動かしこれらをエミッタとして扱っています。

> **Curl Noise**
>
> サンプルコードのパーティクルは、Curl Noise で受ける力を求めて移動する挙動にしています。このように Curl Noise は簡単に疑似流体っぽい動きを作れたりするのでとても便利です。本書の第 6 章「Curl Noise - 疑似流体のためのノイズアルゴリズムの解説」で**@sakope** さんが詳しく解説しているのでぜひ御覧ください。

エミッタの更新

▼ GPUTrailParticles.cs

```
void Update()
{
 cs.SetInt(CSPARAM.PARTICLE_NUM, particleNum);
 cs.SetFloat(CSPARAM.TIME, Time.time);
 cs.SetFloat(CSPARAM.TIME_SCALE, _timeScale);
 cs.SetFloat(CSPARAM.POSITION_SCALE, _positionScale);
 cs.SetFloat(CSPARAM.NOISE_SCALE, _noiseScale);

 var kernelUpdate = cs.FindKernel(CSPARAM.UPDATE);
 cs.SetBuffer(kernelUpdate, CSPARAM.PARTICLE_BUFFER_WRITE, _particleBuffer);

 var updateThureadNum = new Vector3(particleNum, 1f, 1f);
 ComputeShaderUtil.Dispatch(cs, kernelUpdate, updateThureadNum);


 var kernelInput = cs.FindKernel(CSPARAM.WRITE_TO_INPUT);
 cs.SetBuffer(kernelInput, CSPARAM.PARTICLE_BUFFER_READ, _particleBuffer);
 cs.SetBuffer(kernelInput, CSPARAM.INPUT_BUFFER, trails.inputBuffer);

 var inputThreadNum = new Vector3(particleNum, 1f, 1f);
 ComputeShaderUtil.Dispatch(cs, kernelInput, inputThreadNum);
}
```

2 つのカーネルを実行しています。

CSPARAM.UPDATE

エミッタとして使用するパーティクルを更新しています。

CSPARAM.WRITE_TO_INPUT

エミッタの現在の位置を inputBuffer に書き込んでいます。これを Trail の入力して使用します。

Trail への入力

さて、それでは inputBuffer を参照して、nodeBuffer を更新しましょう。

▼ GPUTrailParticles.cs

```
void LateUpdate()
{
  cs.SetFloat(CSPARAM.TIME, Time.time);
  cs.SetFloat(CSPARAM.UPDATE_DISTANCE_MIN, updateDistaceMin);
  cs.SetInt(CSPARAM.TRAIL_NUM, trailNum);
  cs.SetInt(CSPARAM.NODE_NUM_PER_TRAIL, nodeNum);

  var kernel = cs.FindKernel(CSPARAM.CALC_INPUT);
  cs.SetBuffer(kernel, CSPARAM.TRAIL_BUFFER, trailBuffer);
  cs.SetBuffer(kernel, CSPARAM.NODE_BUFFER, nodeBuffer);
  cs.SetBuffer(kernel, CSPARAM.INPUT_BUFFER, inputBuffer);

  ComputeShaderUtil.Dispatch(cs, kernel, new Vector3(trailNum, 1f, 1f));
}
```

CPU 側では必要なパラメータを渡して ComputeShader を Dispatch() している

だけです。メインの ComputeShader 側の処理は次のようになっています。

▼ GPUTrail.compute

```
[numthreads(256,1,1)]
void CalcInput (uint3 id : SV_DispatchThreadID)
{
  uint trailIdx = id.x;
  if ( trailIdx < _TrailNum)
  {
        Trail trail = _TrailBuffer[trailIdx];
        Input input = _InputBuffer[trailIdx];
        int currentNodeIdx = trail.currentNodeIdx;

        bool update = true;
        if ( trail.currentNodeIdx >= 0 )
        {
          Node node = GetNode(trailIdx, currentNodeIdx);
          float dist = distance(input.position, node.position);
          update = dist > _UpdateDistanceMin;
        }

        if ( update )
        {
          Node node;
          node.time = _Time;
          node.position = input.position;

          currentNodeIdx++;
          currentNodeIdx %= _NodeNumPerTrail;

          // write new node
          SetNode(node, trailIdx, currentNodeIdx);

          // update trail
          trail.currentNodeIdx = currentNodeIdx;
          _TrailBuffer[trailIdx] = trail;
        }
  }
}
```

くわしく見ていきましょう。

```
uint trailIdx = id.x;
if ( trailIdx < _TrailNum)
```

まずは引数の id を Trail のインデックスとして使用しています。スレッド数の関係で Trail 数以上の id で呼ばれてしまうこともあるので範囲外のものを if 文で弾いています。

```
int currentNodeIdx = trail.currentNodeIdx;

bool update = true;
if ( trail.currentNodeIdx >= 0 )
{
    Node node = GetNode(trailIdx, currentNodeIdx);
    update = distance(input.position, node.position) > _UpdateDistanceMin;
}
```

次に Trail.currentNodeIdx を確認しています。負数の場合は未使用の Trail です。

GetNode() は、_NodeBuffer から指定の Node を取得する関数です。インデックスの計算が間違いの元なので関数化しています。

すでに使用されている Trail では、最新の Node とインプット位置との距離を比較して、_UpdateDistanceMin より離れていれば更新、近ければ更新しない、としています。エミッタの挙動によりますが、前回の Node とほぼ同じ位置のインプットはたいていほぼ停止状態で微妙に誤差で移動している状態なので、これらを律儀に Node 化して Trail を生成しようとすると、連続する Node 間で方向が大きく異なりかなり汚くなることが多いです。したがってあまり近い距離ではあえて Node の追加をせずにスキップしています。

▼ GPUTrail.compute

```
if ( update )
{
  Node node;
  node.time = _Time;
  node.position = input.position;

  currentNodeIdx++;
  currentNodeIdx %= _NodeNumPerTrail;

  // write new node
  SetNode(node, trailIdx, currentNodeIdx);

  // update trail
  trail.currentNodeIdx = currentNodeIdx;
  _TrailBuffer[trailIdx] = trail;
}
```

最後に _NodeBuffer および _TrailBuffer を更新しています。Trail には入力した Node のインデックスを currentNodeIdx として保存します。Trail あたりの Node 数を超えたらリングバッファ状になるようゼロに戻しています。Node には入力の時間と位置を保存しています。

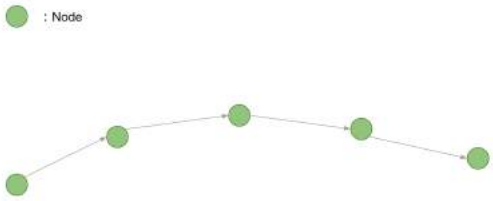
さて、これで Trail の論理的な処理は完成です。次はこの情報から描画する処理について見ていきましょう。

2.3 描画

Trail の描画は基本的には Node 間をラインで繋いでいく処理になります。ここでは個々の Trail はできるだけ簡素にして物量を重視していこうと思います。そのためラインはできるだけポリゴン数を少なくしたいのでカメラと正対する板ポリゴンとして生成します。

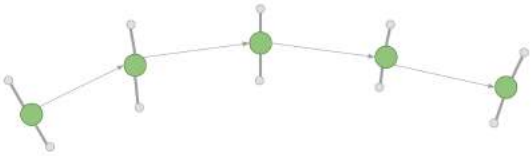
2.3.1 カメラと正対する板ポリゴンの生成

カメラと正対する板ポリゴンを生成する方法は次のようになります。



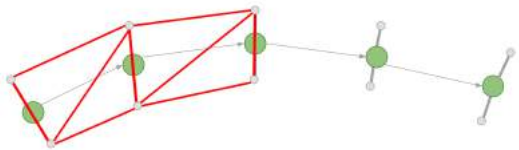
▲図 2.5　Node 列

このような Node 列から



▲図 2.6　Node から生成した頂点

各 Node から視線方向と垂直な方向に指定の幅だけ移動した頂点を求めます。



▲図 2.7　ポリゴン化

生成した頂点同士を繋いでポリゴンにします。それでは実際のコードを見ていきましょう。

2.3.2　CPU 側

CPU 側は単純にパラメータをマテリアルに渡して DrawProcedual() するだけの処理になっています。

▼ GPUTrailRenderer.cs

```
void OnRenderObject()
{
  _material.SetInt(GPUTrails.CSPARAM.NODE_NUM_PER_TRAIL, trails.nodeNum);
  _material.SetFloat(GPUTrails.CSPARAM.LIFE, trails._life);
  _material.SetBuffer(GPUTrails.CSPARAM.TRAIL_BUFFER, trails.trailBuffer);
  _material.SetBuffer(GPUTrails.CSPARAM.NODE_BUFFER, trails.nodeBuffer);
  _material.SetPass(0);

  var nodeNum = trails.nodeNum;
  var trailNum = trails.trailNum;
  Graphics.DrawProcedural(MeshTopology.Points, nodeNum, trailNum);
}
```

いままで出てこなかったパラメータ `trails._life`が登場しています。これは Node の生存時間で Node 自身が持っている生成時刻と照らし合わせて、これだけの時間が経つと透明にしていくような処理に使います。こうすることで Trail の末端がなめらかに消えていく表現ができます。

特に入力すべきメッシュやポリゴンもないので、`Graphics.DrawProcedural()`で trails.nodeNum 個の頂点あるモデルを trails.trailNum 個のインスタンスまとめて描画する命令を発行しています。

2.3.3 GPU 側

vertex shader

▼ GPUTrails.shader

```
vs_out vert (uint id : SV_VertexID, uint instanceId : SV_InstanceID)
{
  vs_out Out;
  Trail trail = _TrailBuffer[instanceId];
  int currentNodeIdx = trail.currentNodeIdx;

  Node node0 = GetNode(instanceId, id-1);
  Node node1 = GetNode(instanceId, id); // current
  Node node2 = GetNode(instanceId, id+1);
  Node node3 = GetNode(instanceId, id+2);

  bool isLastNode = (currentNodeIdx == (int)id);

  if ( isLastNode || !IsValid(node1))
  {
        node0 = node1 = node2 = node3 = GetNode(instanceId, currentNodeIdx);
  }

  float3 pos1 = node1.position;
  float3 pos0 = IsValid(node0) ? node0.position : pos1;
  float3 pos2 = IsValid(node2) ? node2.position : pos1;
  float3 pos3 = IsValid(node3) ? node3.position : pos2;

  Out.pos = float4(pos1, 1);
  Out.posNext = float4(pos2, 1);

  Out.dir = normalize(pos2 - pos0);
  Out.dirNext = normalize(pos3 - pos1);

  float ageRate = saturate((_Time.y - node1.time) / _Life);
  float ageRateNext = saturate((_Time.y - node2.time) / _Life);
  Out.col = lerp(_StartColor, _EndColor, ageRate);
  Out.colNext = lerp(_StartColor, _EndColor, ageRateNext);

  return Out;
}
```

まずは vertex shader の処理です。このスレッドに対応した現在の Node とその次の Node の情報を出力します。

▼ GPUTrails.shader

```
Node node0 = GetNode(instanceId, id-1);
Node node1 = GetNode(instanceId, id); // current
Node node2 = GetNode(instanceId, id+1);
Node node3 = GetNode(instanceId, id+2);
```

現在の Node を node1 として、1 つ前の node0、1 つ先の node2、2 つ先の node3 と計 4 つの Node を参照しています。

▼ GPUTrails.shader

```
bool isLastNode = (currentNodeIdx == (int)id);

if ( isLastNode || !IsValid(node1))
{
  node0 = node1 = node2 = node3 = GetNode(instanceId, currentNodeIdx);
}
```

現在の Node が末端であるか、まだ未入力である場合、node0〜3 を末端の Node のコピーとして扱います。つまり末端より先のまだ情報が無い Node をすべて末端に"折りたたんでいる"扱いにしています。こうすることでこのあとのポリゴン生成の処理にそのまま流すことができます。

▼ GPUTrails.shader

```
float3 pos1 = node1.position;
float3 pos0 = IsValid(node0) ? node0.position : pos1;
float3 pos2 = IsValid(node2) ? node2.position : pos1;
float3 pos3 = IsValid(node3) ? node3.position : pos2;

Out.pos = float4(pos1, 1);
Out.posNext = float4(pos2, 1);
```

さて、4 つの Node から位置情報を取り出します。現在の Node （node1) 以外はすべて未入力である可能性があるので注意が必要です。node0 が未入力なケースがちょっと意外かもしれませんが、currentNodeIdx == 0 のときリングバッファを遡って node0 はバッファの一番最後の Node を指しているのでこのようなケースがありえます。この場合も node1 の位置をコピーすることで、同じ場所に折りたたみます。node2,3 も同様です。このうち、pos1、pos2 を geometry shader に向けて出力します。

▼ GPUTrails.shader

```
Out.dir = normalize(pos2 - pos0);
Out.dirNext = normalize(pos3 - pos1);
```

さらに pos0 → pos2 の方向ベクトルを pos1 における接線（tangent)、pos1 → pos3 の方向ベクトルを pos2 における tangent として出力します。

▼ GPUTrails.shader

```
float ageRate = saturate((_Time.y - node1.time) / _Life);
float ageRateNext = saturate((_Time.y - node2.time) / _Life);
Out.col = lerp(_StartColor, _EndColor, ageRate);
Out.colNext = lerp(_StartColor, _EndColor, ageRateNext);
```

最後に node1、node2 の書き込み時間と現在の時間を比較して色を求めています。

geometry shader

▼ GPUTrails.shader

```
[maxvertexcount(4)]
void geom (point vs_out input[1], inout TriangleStream<gs_out> outStream)
{
  gs_out output0, output1, output2, output3;
  float3 pos = input[0].pos;
  float3 dir = input[0].dir;
  float3 posNext = input[0].posNext;
  float3 dirNext = input[0].dirNext;

  float3 camPos = _WorldSpaceCameraPos;
  float3 toCamDir = normalize(camPos - pos);
  float3 sideDir = normalize(cross(toCamDir, dir));

  float3 toCamDirNext = normalize(camPos - posNext);
  float3 sideDirNext = normalize(cross(toCamDirNext, dirNext));
  float width = _Width * 0.5;

  output0.pos = UnityWorldToClipPos(pos + (sideDir * width));
  output1.pos = UnityWorldToClipPos(pos - (sideDir * width));
  output2.pos = UnityWorldToClipPos(posNext + (sideDirNext * width));
  output3.pos = UnityWorldToClipPos(posNext - (sideDirNext * width));

  output0.col =
  output1.col = input[0].col;
  output2.col =
  output3.col = input[0].colNext;

  outStream.Append (output0);
  outStream.Append (output1);
  outStream.Append (output2);
  outStream.Append (output3);

  outStream.RestartStrip();
}
```

次に geometry shader の処理です。vertex shader から渡された Node 2 つ分の情報からいよいよポリゴンを生成します。2 つの pos と dir から、4 つの位置＝4 角形を求め TriangleStream として出力しています。

▼ GPUTrails.shader

```
float3 camPos = _WorldSpaceCameraPos;
float3 toCamDir = normalize(camPos - pos);
float3 sideDir = normalize(cross(toCamDir, dir));
```

pos からカメラへの方向ベクトル (toCameraDir) と、接線ベクトル (dir) の外積を求め、これをラインの幅として広げる方向 (sideDir) にしています。

▼ GPUTrails.shader

```
output0.pos = UnityWorldToClipPos(pos + (sideDir * width));
output1.pos = UnityWorldToClipPos(pos - (sideDir * width));
```

正負の sideDir 方向に移動した頂点を求めます。ここで Clip 座標系にして fragment shader へ渡すための座標変換まで済ませておきます。posNext に関しても同じ処理をすることで計 4 つの頂点が求まりました。

▼ GPUTrails.shader

```
output0.col =
output1.col = input[0].col;
output2.col =
output3.col = input[0].colNext;
```

各頂点に色を乗せて完成です。

fragment shader

▼ GPUTrails.shader

```
fixed4 frag (gs_out In) : COLOR
{
  return In.col;
}
```

最後に fragment shader です。これ以上無いくらい単純です。色を出力してるだけですね（笑

2.4 応用

以上で Trail の生成ができたかと思います。今回は色だけの処理でしたが、テクスチャをのせたり、幅を変えてみたりといろいろな応用ができると思います。また、GPUTrails.cs、GPUTRailsRenderer.cs とソースコードも別れているとおり、GPUTrails.shader 側は単なるバッファを見て描画するだけの処理なので_TrailBuffer、_NodeBuffer さえ用意すれば実は Trail に限らずライン状の表示に流用できます。

今回は _NodeBuffer に追加するだけの Trail でしたが、毎フレーム全 Node を更新することで触手のようなウネウネしたものも表現したりできると思います。

2.5 まとめ

本章では Trail の GPU 実装のできるだけシンプルな例を紹介しました。GPU を使うとデバッグが大変になる反面、 CPU ではできないような圧倒的な物量表現が可能になります。その「うひょー！」感を本書を通して体験できる方が一人でも増えたらいいなと思います。また、Trail は「モデルを表示する」「スクリーンスペースでアルゴリズムで描画する」の間くらいの応用の幅が広く面白い領域の表現ではないかと思います。この過程で得る理解は Trail に限らずいろいろな映像表現をプログラミングするときに役立つのではないかと思います。

第 3 章

ライン表現のための GeometryShader の応用

3.1 はじめに

今年は個人で Art Hack Day 2018[*1] というアート作品を制作するハッカソンに参加しまして、そこで Unity を使ったビジュアル作品を制作しました。

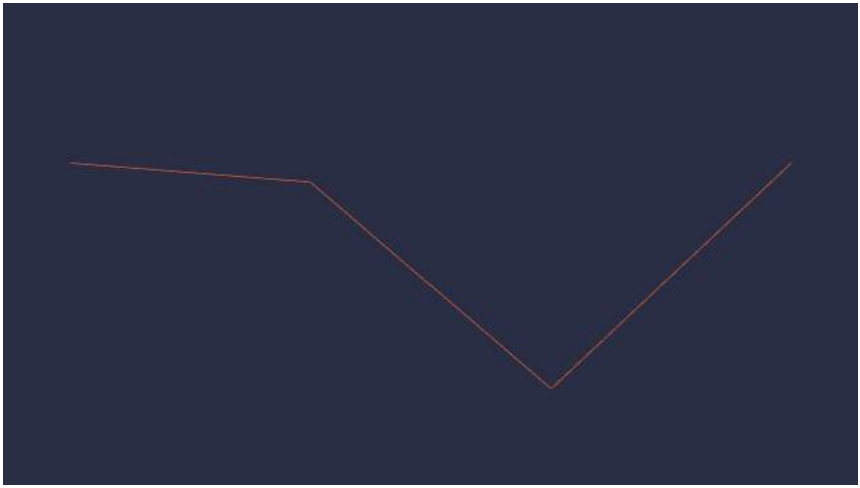
▲図 3.1　Already There のビジュアル部分

作品内で、Geometry Shader を使ってワイヤーフレームの多角形を描画するという手法を使いました。本章ではその手法の解説を行っていきます。本章のサンプルは `https://github.com/IndieVisualLab/UnityGraphicsProgramming2` の「GeometryWireframe」です。

[*1] Art Hack Day 2018 http://arthackday.jp/

3.2 とりあえず線を書いてみる

Unity で線を引くには、LineRenderer や GL を使うことが多いと思いますが、今回は後々描画量が増えることを想定して、Graphics.DrawProcedural を使って行くことにします。
まず最初に、簡単な Sin 波を描画してみます。サンプルの **SampleWaveLine** シーンを見てください。



▲図 3.2　SampleWaveLine シーン

とりあえず、再生ボタンを押して実行すると、Game ビューにオレンジ色の Sin 波が表示されるはずです。Hierarchy ウィンドウの WabeLine オブジェクトを選択し、Inspector ウィンドウで RenderWaveLine コンポーネントの Vertex Num のスライダーを動かすと、Sin 波の滑らかさが変化します。その RenderWaveLine クラスの実装は次のようになっています。

▼リスト 3.1　RenderWaveLine.cs

```
using UnityEngine;

[ExecuteInEditMode]
public class RenderWaveLine : MonoBehaviour {
    [Range(2,50)]
    public int vertexNum = 4;
```

```
    public Material material;

    private void OnRenderObject()
    {
        material.SetInt("_VertexNum", vertexNum - 1);
        material.SetPass(0);
        Graphics.DrawProcedural(MeshTopology.LineStrip, vertexNum);
    }
}
```

Graphics.DrawProcedural は呼び出し後すぐに実行されるので、OnRenderObject の中で呼ばなければなりません。OnRenderObject は、全てのカメラがシーンをレンダリングした後に呼び出されます。Graphics.DrawProcedural の第 1 引数は **MeshTopology** です。MeshTopology はメッシュをどのように構成するかの指定です。指定できる構成は、Triangles（三角ポリゴン）,Quads（四角ポリゴン）,Lines（2 点をつなぐ線）,LineStrip（全ての点を連続してつなぐ）,Points（独立した点）の 6 つです。第 2 引数は頂点数です。

今回は、Sin 波の線上に頂点を配置して、線を結ぶようにしたいため、**MeshTopology.LineStrip** を使います。第 2 引数の vertexNum は、sin 波を描画するのに使う頂点数を指定しています。ここでカンのよい方なら気づくかもしれませんが、どこにも頂点座標の配列を Shader に渡していません。頂点座標は次の Shader の Vertex Shader（頂点シェーダ）の中で計算しています。次に WaveLine.shader です。

▼リスト 3.2　WaveLine.shader

```
Shader "Custom/WaveLine"
{
  Properties
  {
    _Color ("Color", Color) = (1,1,1,1)
    _ScaleX ("Scale X", Float) = 1
    _ScaleY ("Scale Y", Float) = 1
    _Speed ("Speed",Float) = 1
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #pragma target 3.5

      #include "UnityCG.cginc"

      #define PI  3.14159265359
```

```
        struct v2f
        {
          float4 vertex : SV_POSITION;
        };

        float4 _Color;
        int _VertexNum;
        float _ScaleX;
        float _ScaleY;
        float _Speed;

        v2f vert (uint id : SV_VertexID)
        {
          float div = (float)id / _VertexNum;
          float4 pos = float4((div - 0.5) * _ScaleX,
            sin(div * 2 * PI + _Time.y * _Speed) * _ScaleY, 0, 1);

          v2f o;
          o.vertex = UnityObjectToClipPos(pos);
          return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
          return _Color;
        }
        ENDCG
      }
    }
  }
```

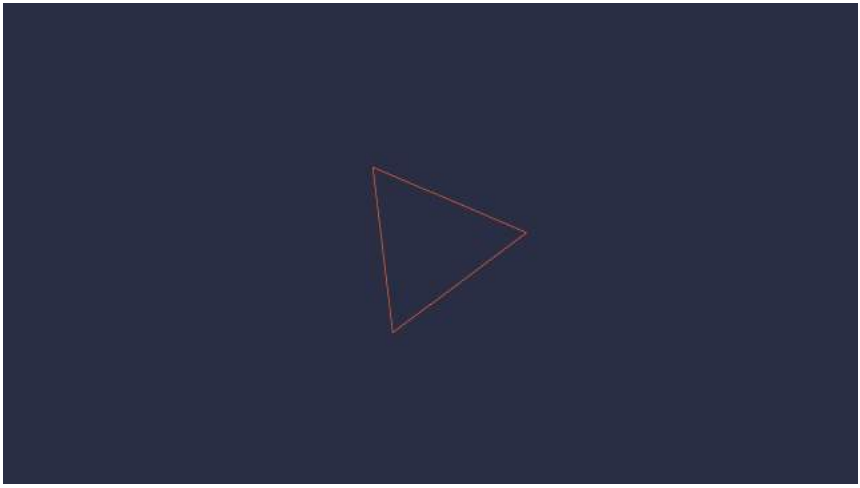
Vertex Shader の関数 vert の引数に SV_VertexID（頂点 ID）が渡ってくるようになっています。頂点 ID は、頂点固有の通し番号です。感覚的には、Graphics.DrawProcedural の第 2 引数に使用する頂点数を渡すと、Vertex Shader が頂点数の回数分呼び出され、引数の頂点 ID には 0〜頂点数-1 までの値が入る感じです。Vertex Shader 内では、頂点 ID を頂点数で割ることで 0〜1 までの割合を計算しています。その求めた割合をもとに頂点座標（pos）を計算しています。Y 座標の計算で先ほど求めた割合を sin 関数に与えて sin 波上の座標を求めています。ついでに _Time.y を足すことで時間の進行による高さの変化のアニメーションも行っています。Vertex Shader 内で頂点座標を計算しているので C#側から頂点座標を渡す必要がないのです。それから、UnityObjectToClipPos でオブジェクト空間からカメラのクリップ空間へ変換した座標を Fragment Shader に渡しています。

3.3 Geometry Shader で動的に二次元の多角形を描く

3.3.1 Geometry Shader で頂点を増やす

次に多角形を描画してみます。多角形を描画するにはそれぞれの角の分だけ頂点が必要です。前項のように頂点をつないで閉じたらできてしまいますが、今回は Geometry Shader を使って 1 個の頂点から多角形を描画してみます。Geometry Shader の詳細は、UnityGraphicsProgramming vol.1[*2]の「第 6 章 ジオメトリシェーダーで草を生やす」を参照してください。ざっくり解説すると、Geometry Shader は、Vertex Shader と Fragment Shader の間に位置する、頂点を増やすことができるシェーダです。

サンプルの **SamplePolygonLine** シーンを見てください。



▲図 3.3　SamplePolygonLine シーン

再生ボタンを押して実行すると、Game ビュー上で三角形が回転が回転しているはずです。Hierarchy ウィンドウの PolygonLine オブジェクトを選択し、Inspector ウィンドウで SinglePolygon2D コンポーネントの Vertex Num のスライダーを動かすと、三角形の角数が増減できます。その SimglePolygon2D クラスの実装は次のようになっています。

[*2] UnityGraphicsProgramming vol.1 https://indievisuallab.stores.jp/items/59edf11ac8f22c0152002588

▼リスト 3.3　SinglePolygon2D.cs

```
  using UnityEngine;

[ExecuteInEditMode]
public class SinglePolygon2D : MonoBehaviour {

    [Range(2, 64)]
    public int vertexNum = 3;

    public Material material;

    private void OnRenderObject()
    {
        material.SetInt("_VertexNum", vertexNum);
        material.SetMatrix("_TRS", transform.localToWorldMatrix);
        material.SetPass(0);
        Graphics.DrawProcedural(MeshTopology.Points, 1);
    }
}
```

RenderWaveLine クラスとほぼ同じ実装になっています。
大きく違う点が 2 つあります。ひとつ目は、Graphics.DrawProcedural の第 1 引数が **MeshTopology.LineStrip** から **MeshTopology.Points** になっている点です。もうひとつは、Graphics.DrawProcedural の第 2 引数が 1 固定になっている点です。前項の RenderWaveLine クラスは、頂点同士をつないで線を引いていたので、**MeshTopology.LineStrip** を指定していましたが、今回は 1 個の頂点だけ渡して多角形を描画したいので、**MeshTopology.Points** を指定しています。というのも、MeshTopology の指定によって、描画に最低限必要な頂点数が変わり、それを下回っていると何も描画されません。MeshTopology.Lines と MeshTopology.LineStrip は線なので 2、MeshTopology.Triangles は三角形なので 3、MeshTopology.Points は点なので 1 です。ちなみに、material.SetMatrix("_TRS", transform.localToWorldMatrix); の部分ですが、SinglePolygon2D コンポーネントを割り当てている GameObject のローカル座標系からワールド座標系へ変換した行列をシェーダーに渡しています。これをシェーダー内で頂点座標に掛けることで、GameObject の transform、すなわち、座標（position）、向き（rotation）、大きさ（scale）が描画する図形に反映されます。

続いて SinglePolygonLine.Shader の実装を見てみましょう。

▼リスト 3.4　SinglePolygonLine.shader

```
Shader "Custom/Single Polygon Line"
{
  Properties
  {
    _Color ("Color", Color) = (1,1,1,1)
```

```
    _Scale ("Scale", Float) = 1
    _Speed ("Speed",Float) = 1
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma geometry geom  // Geometry Shader の宣言
      #pragma fragment frag
      #pragma target 4.0

      #include "UnityCG.cginc"

      #define PI  3.14159265359

      // 出力構造体
      struct Output
      {
        float4 pos : SV_POSITION;
      };

      float4 _Color;
      int _VertexNum;
      float _Scale;
      float _Speed;
      float4x4 _TRS;

      Output vert (uint id : SV_VertexID)
      {
        Output o;
        o.pos = mul(_TRS, float4(0, 0, 0, 1));
        return o;
      }

      // ジオメトリシェーダ
      [maxvertexcount(65)]
      void geom(point Output input[1], inout LineStream<Output> outStream)
      {
        Output o;
        float rad = 2.0 * PI / (float)_VertexNum;
        float time = _Time.y * _Speed;

        float4 pos;

        for (int i = 0; i <= _VertexNum; i++) {
          pos.x = cos(i * rad + time) * _Scale;
          pos.y = sin(i * rad + time) * _Scale;
          pos.z = 0;
          pos.w = 1;
          o.pos = UnityObjectToClipPos(pos);

          outStream.Append(o);
        }
```

```
        outStream.RestartStrip();
      }

      fixed4 frag (Output i) : SV_Target
      {
        return _Color;
      }
      ENDCG
    }
  }
}
```

#pragma vertex vert と**#pragma fragment frag** の間に、新たに**#pragma geometry geom** の宣言が追加されています。これは、geom という名前の Geometry Shader の関数を宣言するという意味です。Vertex Shader の vert は、今回は頂点の座標をとりあえず原点（0,0,0,1）にして、それに C#から渡された_TRS 行列（ローカル座標系からワールド座標系へ変換する行列）を掛けるようになっています。多角形の各頂点の座標計算は次の Geometry Shader の中で行います。

▼ Geometry Shader の定義

```
// ジオメトリシェーダ
[maxvertexcount(65)]
void geom(point Output input[1], inout LineStream<Output> outStream)
```

maxvertexcount

Geometry Shader から出力する頂点の最大数です。今回は、SinglePolygonLine クラスの VertexNum で 64 頂点まで増やせるようにしていますが、64 個目の頂点から 0 個目の頂点を結ぶ線が必要な為、65 を指定しています。

point Output input[1]

Vertex Shader からの入力情報を表しています。point は primitiveType で頂点 1 個分受け取るという意味で、Output は構造体名、input[1] は長さ 1 の配列を表しています。今回は 1 つの頂点しか使わないので point と input[1] を指定しましたが、メッシュなど三角ポリゴンの頂点をいじりたい時は triangle と input[3] にしたりします。

inout LineStream<Output> outStream

Geometry Shader からの出力情報を表しています。LineStream<Output>は、Output 構造体の線を出力するという意味です。他にも PointStream（点）、TriangleStream（三角ポリゴン）があります。次に関数内の説明です。

▼ 関数内の実装

```
Output o;
float rad = 2.0 * PI / (float)_VertexNum;
float time = _Time.y * _Speed;

float4 pos;

for (int i = 0; i <= _VertexNum; i++) {
  pos.x = cos(i * rad + time) * _Scale;
  pos.y = sin(i * rad + time) * _Scale;
  pos.z = 0;
  pos.w = 1;
  o.pos = UnityObjectToClipPos(pos);

  outStream.Append(o);
}

outStream.RestartStrip();
```

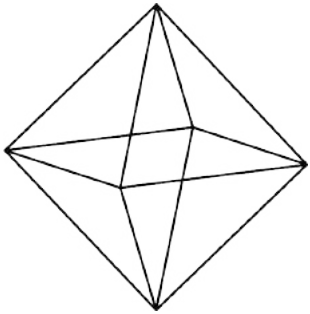
多角形の各頂点の座標を計算するために、2 π（360 度）を頂点数で割って、一角の角度を求めています。それをループ内で三角関数（sin, cos）を使って頂点座標を計算しています。計算した座標を outStream.Append(o) で頂点として出力します。_VertexNum の数だけループを回して頂点を出力したあと、outStream.RestartStrip() で現在のストリップを終了して次のストリップを開始します。Append() で追加していく限り、LineStream として線が繋がっていきます。RestartStrip() を実行することで一旦現在の線を終了します。次に Append() が呼ばれると、前の線とは繋がらず、新しい線が始まります。

3.4 Octahedron Sphere を作ってみる

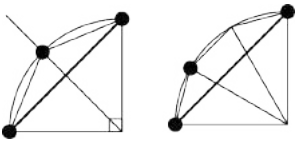
3.4.1 Octahedron Sphere とは？

正八面体（Regular octahedron）とは、図 3.4 にあるとおり、8 つの正三角形で構成された多面体です。Octahedron Sphere とは、正八面体を構成する正三角形の 3 つの頂点を球面線形補間[*3]して分割していくことで作られる球体です。通常の線形補間が 2 点間を直線でつなぐように補間するのに対し、球面線形補間は、図 3.5 のように 2 点間を球面上を通るように補間します。

[*3] spherical linear interpolation, 略して slerp

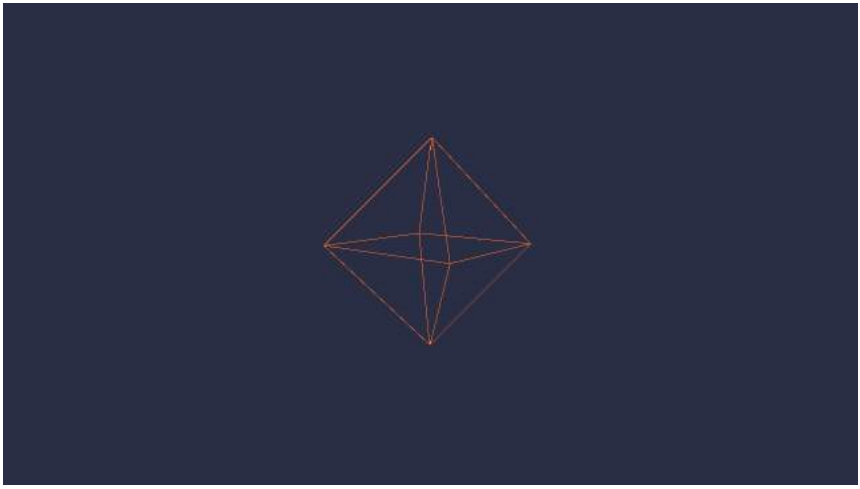


▲図 3.4　正八面体



▲図 3.5　正八面体

サンプルの **SampleOctahedronSample** シーンを見てください。



▲図 3.6　SampleOctahedronSample シーン

実行ボタンを押すと、Game ビューの中央にゆっくり回転する正八面体が表示されているはずです。また、Hierarchy ウィンドウの SingleOctahedronSphere オブジェクトの Geometry Octahedron Sphere コンポーネントの Level のスライダーを変更すると、正八面体の辺が分割されて少しずつ球体に近づいていくと思います。

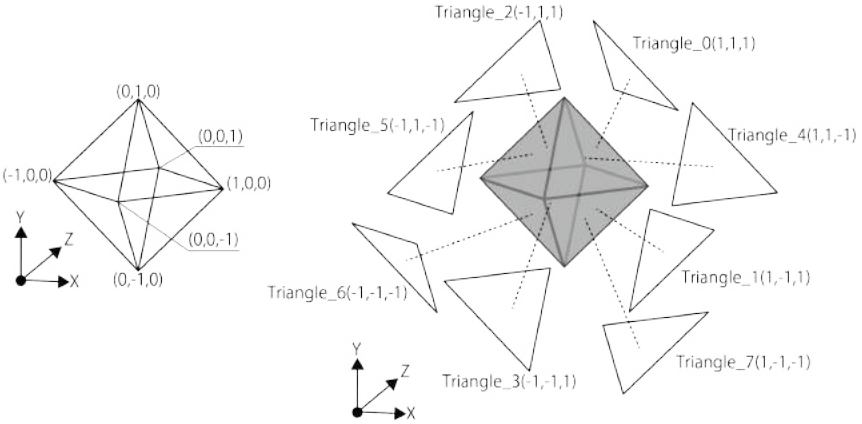
3.4.2 Geometry Shader の中で正八面体を分割していく

次に、実装を見てみましょう。C#側の実装は、前項の SinplePolygon2D.cs とほぼほぼ同じなので省略します。OctahedronSphere.shader は、ソース全体は長いので、Geometry Shader の中だけ解説していきます。

▼リスト 3.5　OctahedronSphere.shader の Gometry Shader の先頭部分

```
// ジオメトリシェーダ
  float4 init_vectors[24];
  // 0 : the triangle vertical to (1,1,1)
  init_vectors[0] = float4(0, 1, 0, 0);
  init_vectors[1] = float4(0, 0, 1, 0);
  init_vectors[2] = float4(1, 0, 0, 0);
  // 1 : to (1,-1,1)
  init_vectors[3] = float4(0, -1, 0, 0);
  init_vectors[4] = float4(1, 0, 0, 0);
  init_vectors[5] = float4(0, 0, 1, 0);
  // 2 : to (-1,1,1)
  init_vectors[6] = float4(0, 1, 0, 0);
  init_vectors[7] = float4(-1, 0, 0, 0);
  init_vectors[8] = float4(0, 0, 1, 0);
  // 3 : to (-1,-1,1)
  init_vectors[9] = float4(0, -1, 0, 0);
  init_vectors[10] = float4(0, 0, 1, 0);
  init_vectors[11] = float4(-1, 0, 0, 0);
  // 4 : to (1,1,-1)
  init_vectors[12] = float4(0, 1, 0, 0);
  init_vectors[13] = float4(1, 0, 0, 0);
  init_vectors[14] = float4(0, 0, -1, 0);
  // 5 : to (-1,1,-1)
  init_vectors[15] = float4(0, 1, 0, 0);
  init_vectors[16] = float4(0, 0, -1, 0);
  init_vectors[17] = float4(-1, 0, 0, 0);
  // 6 : to (-1,-1,-1)
  init_vectors[18] = float4(0, -1, 0, 0);
  init_vectors[19] = float4(-1, 0, 0, 0);
  init_vectors[20] = float4(0, 0, -1, 0);
  // 7 : to (1,-1,-1)
  init_vectors[21] = float4(0, -1, 0, 0);
  init_vectors[22] = float4(0, 0, -1, 0);
  init_vectors[23] = float4(1, 0, 0, 0);
```

まず、図 3.7 のように初期値となる“正規化された”正八面体の三角系を定義しています。



▲図 3.7　正八面体の頂点座標と三角形

float4 で定義しているのは、クォータニオンとして定義しているためです。

▼リスト 3.6　OctahedronSphere.shader の三角形の球面線形補間分割処理部分

```
for (int i = 0; i < 24; i += 3)
{
  for (int p = 0; p < n; p++)
  {
    // edge index 1
    float4 edge_p1 = qslerp(init_vectors[i],
      init_vectors[i + 2], (float)p / n);
    float4 edge_p2 = qslerp(init_vectors[i + 1],
      init_vectors[i + 2], (float)p / n);
    float4 edge_p3 = qslerp(init_vectors[i],
      init_vectors[i + 2], (float)(p + 1) / n);
    float4 edge_p4 = qslerp(init_vectors[i + 1],
      init_vectors[i + 2], (float)(p + 1) / n);

    for (int q = 0; q < (n - p); q++)
    {
      // edge index 2
      float4 a = qslerp(edge_p1, edge_p2, (float)q / (n - p));
      float4 b = qslerp(edge_p1, edge_p2, (float)(q + 1) / (n - p));
      float4 c, d;

      if(distance(edge_p3, edge_p4) < 0.00001)
      {
        c = edge_p3;
        d = edge_p3;
      }
      else {
        c = qslerp(edge_p3, edge_p4, (float)q / (n - p - 1));
        d = qslerp(edge_p3, edge_p4, (float)(q + 1) / (n - p - 1));
```

```
      }

      output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, a));
      output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, b));
      output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, c));

      outStream.Append(output1);
      outStream.Append(output2);
      outStream.Append(output3);
      outStream.RestartStrip();

      if (q < (n - p - 1))
      {
        output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, c));
        output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, b));
        output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, d));

        outStream.Append(output1);
        outStream.Append(output2);
        outStream.Append(output3);
        outStream.RestartStrip();
      }
    }
  }
}
```

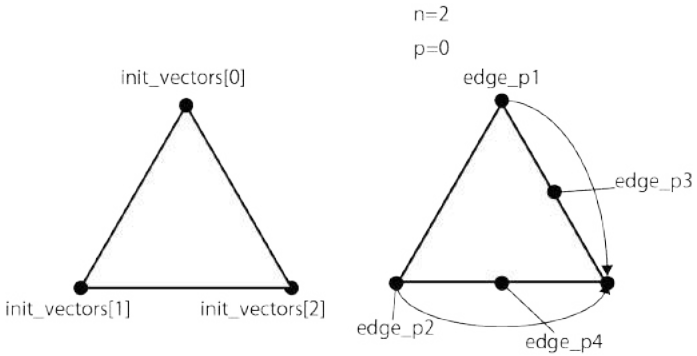
三角形を球面線形補間で分割している部分です。n は三角形の分割数です。edge_p1 と edge_p2 は三角形の開始点を、edge_p3 と ege_p4 は、分割した辺の中点を求めています。qslerp 関数は、球面線形補間を求める関数です。qslerp の定義は次のとおりです。

▼リスト 3.7　Quaternion.cginc の qslerp の定義

```
// a:開始 Quaternion b:目標 Quaternion t:比率
float4 qslerp(float4 a, float4 b, float t)
{
  float4 r;
  float t_ = 1 - t;
  float wa, wb;
  float theta = acos(a.x * b.x + a.y * b.y + a.z * b.z + a.w * b.w);
  float sn = sin(theta);
  wa = sin(t_ * theta) / sn;
  wb = sin(t * theta) / sn;
  r.x = wa * a.x + wb * b.x;
  r.y = wa * a.y + wb * b.y;
  r.z = wa * a.z + wb * b.z;
  r.w = wa * a.w + wb * b.w;
  normalize(r);
  return r;
}
```

三角形分割の流れ 1

続いて、三角形の分割処理の流れを説明します。例として、分割数 2（n=2）の場合の流れです。



▲図 3.8　三角形の分割処理の流れ 1 、edge_p1～p4 の計算

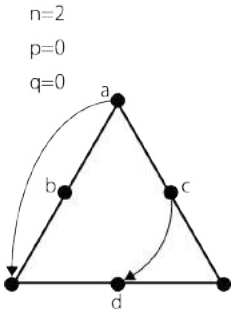
図 3.8 は、次のコードを表しています。

▼リスト 3.8　edge_p1～p4 の計算

```
for (int p = 0; p < n; p++)
{
  // edge index 1
  float4 edge_p1 = qslerp(init_vectors[i],
    init_vectors[i + 2], (float)p / n);
  float4 edge_p2 = qslerp(init_vectors[i + 1],
    init_vectors[i + 2], (float)p / n);
  float4 edge_p3 = qslerp(init_vectors[i],
    init_vectors[i + 2], (float)(p + 1) / n);
  float4 edge_p4 = qslerp(init_vectors[i + 1],
    init_vectors[i + 2], (float)(p + 1) / n);
```

init_vectors 配列の 3 点から、edge_p1～edge_p4 の座標を求めています。p=0 の時は、p/n = 0/2 = 0 で edge_p1 = init_vectors[0]、edge_p2=init_vectors[1] になります。edge_p3 と edge_p4 は、(p+1)/n = (0+1)/2 = 0.5 でそれぞれ init_vectors[0] と init_vectors[2] の間、init_vectors[1] と init_vectors[2] の間になります。主に三角形の右側を分割する流れです。

三角形分割の流れ２



▲図 3.9　三角形の分割処理の流れ２、abcd の計算

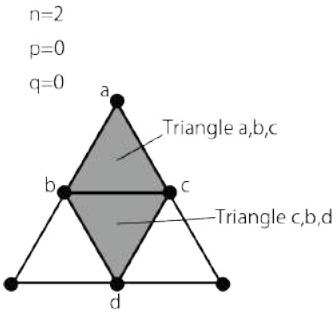
図 3.9 は、次のコードを表しています。

▼リスト 3.9　座標 a,b,c,d の計算

```
for (int q = 0; q < (n - p); q++)
{
  // edge index 2
  float4 a = qslerp(edge_p1, edge_p2, (float)q / (n - p));
  float4 b = qslerp(edge_p1, edge_p2, (float)(q + 1) / (n - p));
  float4 c, d;

  if(distance(edge_p3, edge_p4) < 0.00001)
  {
    c = edge_p3;
    d = edge_p3;
  }
  else {
    c = qslerp(edge_p3, edge_p4, (float)q / (n - p - 1));
    d = qslerp(edge_p3, edge_p4, (float)(q + 1) / (n - p - 1));
  }
```

前項で求めた edge_p1～p4 を使って、頂点 abcd の座標を求めています。主に三角形の左側を分割する流れです。条件によっては edge_p3 と edge_p4 の座標が同じになります。これは、三角形の右側がこれ以上分割できない段階になった時に発生します。その場合は c,d はどちらも三角形の右下の座標を取ります。

三角形分割の流れ３



▲図 3.10　三角形の分割処理の流れ３、三角形 abc, 三角形 cbd を出力

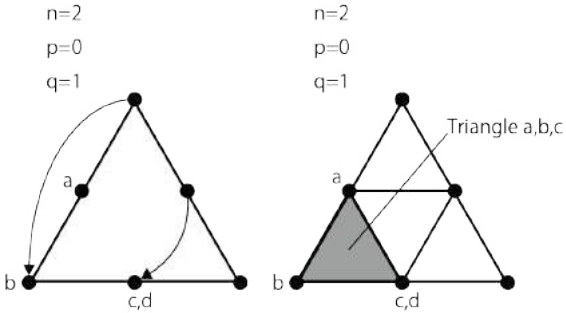
図 3.10 は、次のコードを表しています。

▼リスト 3.10　座標 a,b,c を結ぶ三角形＆座標 c,b,d を結ぶ三角形を出力

```
output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, a));
output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, b));
output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, c));

outStream.Append(output1);
outStream.Append(output2);
outStream.Append(output3);
outStream.RestartStrip();

if (q < (n - p - 1))
{
  output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, c));
  output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, b));
  output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, d));
  outStream.Append(output1);
  outStream.Append(output2);
  outStream.Append(output3);
  outStream.RestartStrip();
}
```

計算した a,b,c,d の座標を、UnityObjectToClipPos やワールド座標変換行列を掛けてスクリーン用の座標に変換します。その後、outStream.Append と outStream.RestartStrip で、a,b,c と c,b,d を結ぶ２つの三角形を出力します。

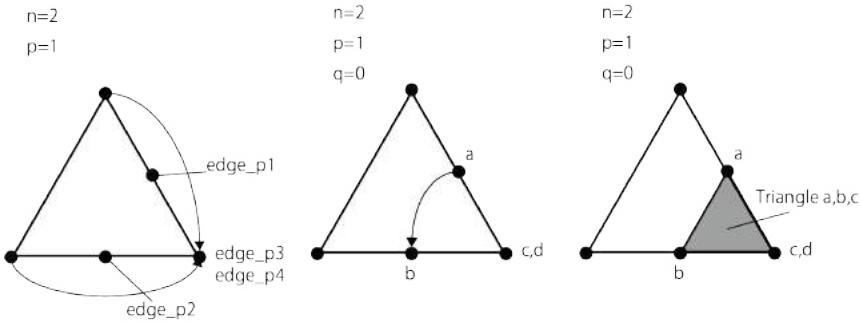
三角形分割の流れ４



▲図 3.11　三角形の分割処理の流れ４、q=1 の場合

q=1 の場合、a は 1/2=0.5 なので edge_p1 と edge_p2 の中間に、b は 1/1=1 なので edge_p2 の位置になります。c は 1/1=1 なので edge_p4 に、d は一応計算してますが、if (q < (n - p - 1)) の条件に引っかからないので使われません。a,b,c を結ぶ三角形を出力します。

三角形分割の流れ５



▲図 3.12　三角形の分割処理の流れ５、p=1 の場合

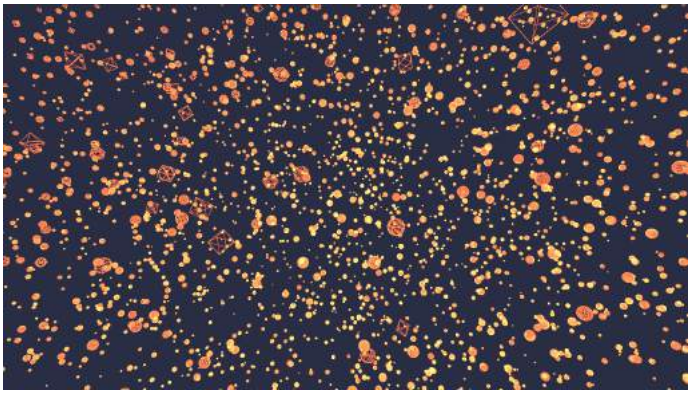
q の for 文が終わり、p=1 になった時の流れです。p/n = 1/2 = 0.5 なので、edge_p1 は init_vectors[0] と init_vectors[2] の中間に、edge_p2 は init_vectors[1] と init_vectors[2] の中間になります。あとの a,b,c,d の座標計算と三角形 a,b,c の出

力は前述の処理と同じです。これで 1 つの三角形を 4 つに分割できました。正八面体の全ての三角形に対して前述までを処理を行っています。

3.4.3 おまけ

この他にも、紙面の都合上、紹介しきれないサンプルを 3 つ用意していますので、興味がある方は是非ご覧になってください。

- OctahedronSphere の GPUInstancing 版（SampleOctahedronSphereInstancing シーン）
- 9 段階まで分割可能な OctahedronSphere 単体版（SampleOctahedronSphereMultiVertex シーン）
- 9 段階まで分割可能な OctahedronSphere の GPUInstancing 版（SampleOctahedronSphereMultiVertexInstancing シーン）



▲図 3.13　SampleOctahedronSphereMultiVertexInstancing シーン

3.5 まとめ

本章では、ライン表現のための Geometry Shader の応用について解説しました。Geometry Shader は普段はポリゴンを分割したり、パーティクルの板ポリゴンを作ったりという事例をよく見かけますが、皆さんも動的に頂点数を増やせる特性を利用して面白い表現を模索してみてください。

第 4 章

Projection Spray

4.1 はじめに

こんにちは！ すぎのひろのりです！ 前回の『UnityGraphicsPrograming Vol1』ではキチンと記事書けなくてすみませんでした！ 代筆してくれた大石くんには、感謝です (._.)

ここでは、前回書けなかったスプレーのプログラムについて、解説していきたいと思います。Unity のコード的には、Vol1 時点のモノよりちょっと良くなりました！

まず、Unity の Built-in のライトを使用せずにシンプルな照明効果を独自実装してみます。そして、その応用として、3D オブジェクトをスプレーでペイントする処理の開発について、解説していきます。この章のコンセプトとしては、UnityCG.cginc やビルトインの処理を参考に機能を自作し、新しい機能に応用していくという流れを追えたらな。と思います。

4.1.1 サンプルリポジトリ

本章のサンプルは、
`https://github.com/IndieVisualLab/UnityGraphicsProgramming2` の ProjectionSpray フォルダ内です。

4.2 LightComponent の実装

Unity の Light はものすごく便利です。ライトオブジェクトを設置するだけで、世界は明るくなります。Inspector から影を選択すると、自動的にシャドウマップが作られ、オブジェクトから影が落ちます。

まずは、Unity がどのようにライトを実装しているのかを見ながら、独自にライトを実装していきます。

4.2.1 Unity Built-in Shader

Unity では、デフォルトで入っているマテリアルのシェーダーや、内部的に使われている CGINC ファイルを Unity ダウンロードアーカイブから、ダウンロードすることができます。

独自のシェーダーを書くときの参考になったり、より深く CG の描写について知ることができるので、是非、ダウンロードして見ることをお勧めします！

▲図 4.1　https://unity3d.com/jp/get-unity/download/archive

本章で関係ありそうな、ライティング関連のファイルは以下です

- CGIncludes/UnityCG.cginc
- CGIncludes/AutoLight.cginc
- CGIncludes/Lighting.cginc
- CGIncludes/UnityLightingCommon.cginc

基本的な LambertLighting について見てみます。（リスト 4.1）ランバートさんが考えました。

▼リスト 4.1　Lighting.cginc

```
 1: struct SurfaceOutput {
 2:     fixed3 Albedo;
 3:     fixed3 Normal;
 4:     fixed3 Emission;
 5:     half Specular;
 6:     fixed Gloss;
 7:     fixed Alpha;
 8: };
 9: ~~
10: inline fixed4 UnityLambertLight (SurfaceOutput s, UnityLight light)
11: {
12:     fixed diff = max (0, dot (s.Normal, light.dir));
13:
14:     fixed4 c;
15:     c.rgb = s.Albedo * light.color * diff;
16:     c.a = s.Alpha;
17:     return c;
18: }
```

実際のライティング計算ですが、ライトからメッシュまでの方向とメッシュの法線方向との内積から、ディフューズの値を計算しています。リスト 4.1

```
fixed diff = max (0, dot (s.Normal, light.dir));
```

Lighting.cginc 内で未定義の UnityLight については、UnityLightingCommon.cginc 内で定義していて、ライトの色と方向の情報が格納されています。リスト 4.2

▼リスト 4.2　UnityLightingCommon.cginc

```
1: struct UnityLight
2: {
3:     half3 color;
4:     half3 dir;
5:
6:     // Deprecated: Ndotl is now calculated on the fly
7:     // and is no longer stored. Do not used it.
8:     half  ndotl;
9: };
```

4.2.2　メッシュの法線の表示

実際のライティングの処理を見てみて、ライティングの計算にはメッシュの法線（Normal）情報が必要だということが分かったので、Shader でメッシュの法線情報を表示する方法を軽く見ていきます。

サンプルプロジェクト内の **00_viewNormal.unity** のシーンを参照してください。

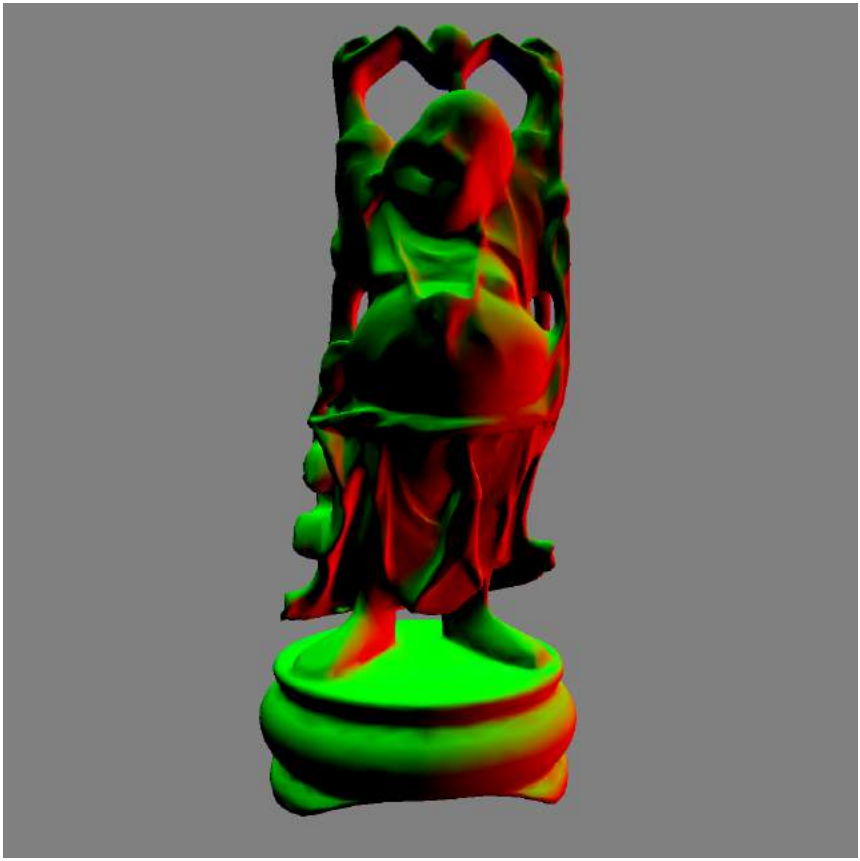
オブジェクトに、法線情報をカラーとして出力するマテリアルが設定してあり、そのシェーダーはリスト 4.3 のようになっています。

▼リスト 4.3　simple-showNormal.shader

```
 1: struct appdata
 2: {
 3:     float4 vertex : POSITION;
 4:     float3 normal : NORMAL;
 5: };
 6:
 7: struct v2f
 8: {
 9:     float3 worldPos : TEXCOORD0;
10:     float3 normal : TEXCOORD1;
11:     float4 vertex : SV_POSITION;
12: };
13:
14: v2f vert (appdata v)
15: {
16:     v2f o;
17:     o.vertex = UnityObjectToClipPos(v.vertex);
18:     o.normal = UnityObjectToWorldNormal(v.normal);
19:     return o;
20: }
21:
22: half4 frag (v2f i) : SV_Target
23: {
24:     fixed4 col = half4(i.normal,1);
25:     return col;
26: }
```

頂点シェーダ（v2f vert）で、メッシュのワールド座標系における法線方向を算出しフラグメントシェーダ（half4 frag）に渡します。フラグメントシェーダでは渡された法線情報を、x 成分を R、y 成分を G, z 成分を B として、カラーに変換してそのまま出力しています。リスト 4.3

イメージ上は黒に見える部分でも、実際は法線 x に負の値が入っている場合もあります。図 4.2



▲図 4.2　00_viewNormal.unity

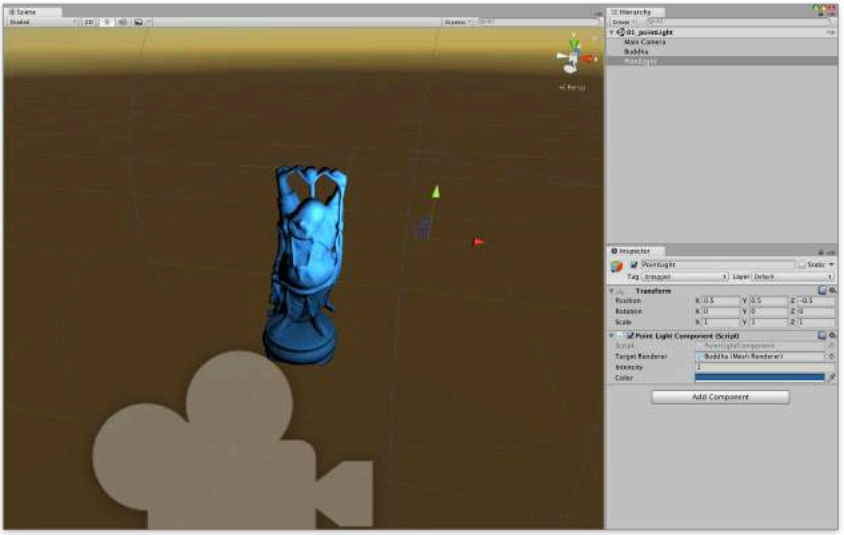
これで、メッシュにライティングを施す準備ができました。

ビルトインシェーダーヘルパー機能

UnityCG.cginc 内にはシェーダーを簡単に書くためのビルトインユーティリティ関数があります。たとえば、リスト 4.3 で使われている `UnityObjectToClipPos`は頂点の位置をオブジェクト（ローカル）座標系からクリップ座標系に変換します。また、`UnityObjectToWorldNormal`の関数は法線方向をオブジェクト座標系からワールド座標系に変換しています。
シェーダーの記述に便利なので、その他の関数に関しては、UnityCG.cginc を参照するか、公式マニュアルをご参照下さい。`https://docs.unity3d.com/ja/current/Manual/SL-BuiltinFunctions.html`
また、座標変換や各座標系について詳しく知りたい場合は、Unity Graphics Programming vol.1、福永さんの『第 9 章 Multi Plane Perspective Projection』をご参照いただけると、詳しくなれるかもしれません。

4.2.3 点光源（PointLight）

サンプルプロジェクト内の **01_pointLight.unity** のシーンを参照してください。



▲図 4.3　01_pointLight.unity

点光源は、ある一点から全方向を照らす光源です。シーンには、Buddha のメッシュオブジェクトと PointLight のオブジェクトがあります。PointLight オブジェクトには、メッシュにライトの情報を送るためのスクリプト（リスト 4.4）が付いていて、そのライトの情報をもとに、ライティングした結果をマテリアル（リスト 4.5）で表示しています。

▼リスト 4.4　PointLightComponent.cs

```
 1: using UnityEngine;
 2:
 3: [ExecuteInEditMode]
 4: public class PointLightComponent : MonoBehaviour
 5: {
 6:     static MaterialPropertyBlock mpb;
 7:
 8:     public Renderer targetRenderer;
 9:     public float intensity = 1f;
10:     public Color color = Color.white;
11:
12:     void Update()
13:     {
14:         if (targetRenderer == null)
15:             return;
16:         if (mpb == null)
17:             mpb = new MaterialPropertyBlock();
```

```
18:
19:         targetRenderer.GetPropertyBlock(mpb);
20:         mpb.SetVector("_LitPos", transform.position);
21:         mpb.SetFloat("_Intensity", intensity);
22:         mpb.SetColor("_LitCol", color);
23:         targetRenderer.SetPropertyBlock(mpb);
24:     }
25:
26:     private void OnDrawGizmos()
27:     {
28:         Gizmos.color = color;
29:         Gizmos.DrawWireSphere(transform.position, intensity);
30:     }
31: }
```

このコンポーネントは、ライトの位置と強さと色を対象のメッシュに受け渡しています。そして、受け取った情報をもとにライティング処理をするマテリアルが設定してあります。

マテリアルの "_LitPos", "_LitCol", "_Intensity"の各プロパティに、CSharpから値が設定されています。

▼リスト 4.5　simple-pointLight.shader

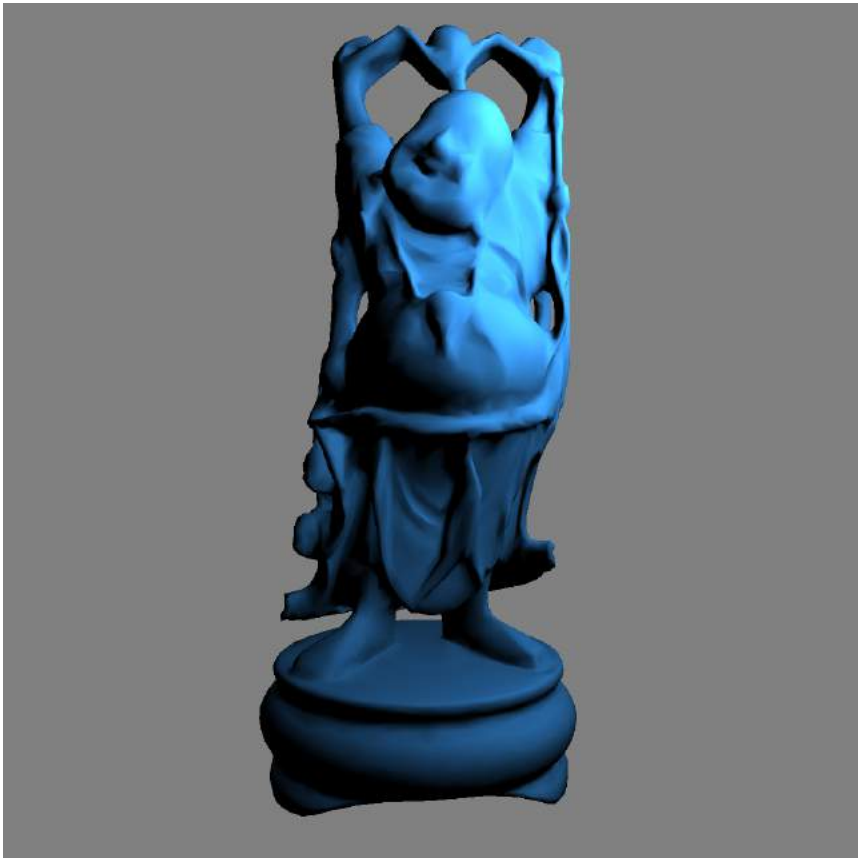
```
 1: Shader "Unlit/Simple/PointLight-Reciever"
 2: {
 3:     Properties
 4:     {
 5:         _LitPos("light position", Vector) = (0,0,0,0)
 6:         _LitCol("light color", Color) = (1,1,1,1)
 7:         _Intensity("light intensity", Float) = 1
 8:     }
 9:     SubShader
10:     {
11:         Tags { "RenderType"="Opaque" }
12:         LOD 100
13:
14:         Pass
15:         {
16:             CGPROGRAM
17:             #pragma vertex vert
18:             #pragma fragment frag
19:
20:             #include "UnityCG.cginc"
21:
22:             struct appdata
23:             {
24:                 float4 vertex : POSITION;
25:                 float3 normal : NORMAL;
26:             };
27:
28:             struct v2f
29:             {
30:                 float3 worldPos : TEXCOORD0;
31:                 float3 normal : TEXCOORD1;
```

```
32:                 float4 vertex : SV_POSITION;
33:             };
34:
35:             half4 _LitPos, _LitCol;
36:             half _Intensity;
37:
38:             v2f vert (appdata v)
39:             {
40:                 v2f o;
41:                 o.vertex = UnityObjectToClipPos(v.vertex);
42:                 o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
43:                 //ワールド座標系におけるメッシュの位置をフラグメントシェーダに
渡す
44:                 o.normal = UnityObjectToWorldNormal(v.normal);
45:                 return o;
46:             }
47:
48:             fixed4 frag (v2f i) : SV_Target
49:             {
50:                 half3 to = i.worldPos - _LitPos;
51:                 //ライトの位置からメッシュ位置へのベクトル
52:                 half3 lightDir = normalize(to);
53:                 half dist = length(to);
54:                 half atten =
55:                     _Intensity * dot(-lightDir, i.normal) / (dist * dist);
56:
57:                 half4 col = max(0.0, atten) * _LitCol;
58:                 return col;
59:             }
60:             ENDCG
61:         }
62:     }
63: }
```

ライティングの計算は、基本の LambertLighting（リスト 4.1）の計算をもとに拡散光を計算し、強さは距離の二乗に反比例して減衰するようにしている。リスト 4.5

```
half atten = _Intensity * dot(-lightDir, i.normal) / (dist * dist
);
```

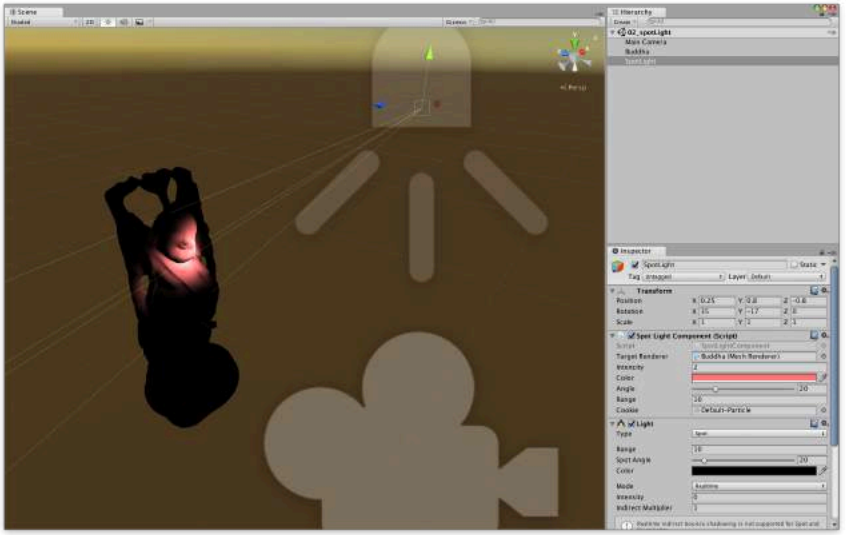
1 つのモデルに対して 1 つのライトの簡単なシステムですが、ライティング処理を実装できました。



▲図 4.4　01_pointLight.unity

4.2.4　スポットライト（SpotLight）

次に、スポットライトの実装をしてみます。スポットライトは、ポイントライトと違い、方向を持ったライトで、一方方向に向けて光を出しています。



▲図 4.5　02_spotLight.unity

ここでは、スポットライトの Gizmo 表示のためだけに Unity 標準のライトを使用しています。図 4.5

スポットライトは方向性があるため、位置情報の他にライトの向き、スポットアングル（角度）の情報がポイントライトのときに加えて増えます。これらの情報はライトの `worldToLightMatrix`、`projectionMatrix`としてそれぞれ `Matrix4x4`（Shader だと `float4x4`）のプロパティで受け渡します。

さらに、スポットライトは LightCookie を設定することもできます。（Unity にはデフォルトの LightCookie があるのですが、エディタから選択できなかったので、Default-Particle のテクスチャを使用しています）

▼リスト 4.6　SpotLightComponent.cs

```
 1: using UnityEngine;
 2:
 3: [ExecuteInEditMode]
 4: public class SpotLightComponent : MonoBehaviour
 5: {
 6:     static MaterialPropertyBlock mpb;
 7:
 8:     public Renderer targetRenderer;
 9:     public float intensity = 1f;
10:     public Color color = Color.white;
```

```
11:     [Range(0.01f, 90f)] public float angle = 30f;
12:     public float range = 10f;
13:     public Texture cookie;
14:
15:     void Update()
16:     {
17:         if (targetRenderer == null)
18:             return;
19:         if (mpb == null)
20:             mpb = new MaterialPropertyBlock();
21:
22:         //projectionMatrix を計算している
23:         var projMatrix = Matrix4x4.Perspective(angle, 1f, 0f, range);
24:         var worldToLightMatrix = transform.worldToLocalMatrix;
25:
26:         targetRenderer.GetPropertyBlock(mpb);
27:         mpb.SetVector("_LitPos", transform.position);
28:         mpb.SetFloat("_Intensity", intensity);
29:         mpb.SetColor("_LitCol", color);
30:         mpb.SetMatrix("_WorldToLitMatrix", worldToLightMatrix);
31:         mpb.SetMatrix("_ProjMatrix", projMatrix);
32:         mpb.SetTexture("_Cookie", cookie);
33:         targetRenderer.SetPropertyBlock(mpb);
34:     }
35: }
```

projectionMatrix は、`Matrix4x4.Perspective(angle, 1f, 0f, range)`で、算出しています。

スポットライトから受け取ったパラメータ情報をもとに Shader でライティング処理を計算し、表示します。リスト 4.7

▼リスト 4.7　simple-spotLight.shader

```
 1: uniform float4x4 _ProjMatrix, _WorldToLitMatrix;
 2:
 3: sampler2D _Cookie;
 4: half4 _LitPos, _LitCol;
 5: half _Intensity;
 6:
 7: ~~
 8:
 9: fixed4 frag (v2f i) : SV_Target
10: {
11:     half3 to = i.worldPos - _LitPos.xyz;
12:     half3 lightDir = normalize(to);
13:     half dist = length(to);
14:     half atten = _Intensity * dot(-lightDir, i.normal) / (dist * dist);
15:
16:     half4 lightSpacePos = mul(_WorldToLitMatrix, half4(i.worldPos, 1.0));
17:     half4 projPos = mul(_ProjMatrix, lightSpacePos);
18:     projPos.z *= -1;
19:     half2 litUv = projPos.xy / projPos.z;
20:     litUv = litUv * 0.5 + 0.5;
21:     half lightCookie = tex2D(_Cookie, litUv);
```

```
22:     lightCookie *=
23:         0<litUv.x && litUv.x<1 && 0<litUv.y && litUv.y<1 && 0<projPos.z;
24:
25:     half4 col = max(0.0, atten) * _LitCol * lightCookie;
26:     return col;
27: }
28:
```

基本的に、フラグメントシェーダー以外はポイントライトと同じだということが分かるかと思います。リスト 4.7

16 行目から 22 行目にかけてが、スポットライトの各地点における強度を計算しています。ライトの位置から見たとき、その地点がライトの範囲に入っているかと、その地点におけるライト Cookie をサンプリングしてライトの強度を求めます。

▲図 4.6　02_spotLight.unity

ビルトインの Cookie の処理

スポットライトの Cookie の処理については、ビルトイン CGINC、AutoLight.cginc 内の `UnitySpotCookie()`の部分が参考になります。

▼リスト 4.8　AutoLight.cginc

```
 1: #ifdef SPOT
 2: sampler2D _LightTexture0;
 3: unityShadowCoord4x4 unity_WorldToLight;
 4: sampler2D _LightTextureB0;
 5: inline fixed UnitySpotCookie(unityShadowCoord4 LightCoord)
 6: {
 7:     return tex2D(
 8:         _LightTexture0,
 9:         LightCoord.xy / LightCoord.w + 0.5
10:     ).w;
11: }
12: inline fixed UnitySpotAttenuate(unityShadowCoord3 LightCoord)
13: {
14:     return tex2D(
15:         _LightTextureB0,
16:         dot(LightCoord, LightCoord).xx
17:     ).UNITY_ATTEN_CHANNEL;
18: }
19: #define UNITY_LIGHT_ATTENUATION(destName, input, worldPos) \
20:     unityShadowCoord4 lightCoord = mul( \
21:         unity_WorldToLight, \
22:         unityShadowCoord4(worldPos, 1) \
23:     ); \
24:     fixed shadow = UNITY_SHADOW_ATTENUATION(input, worldPos); \
25:     fixed destName = \
26:         (lightCoord.z > 0) * \
27:         UnitySpotCookie(lightCoord) * \
28:         UnitySpotAttenuate(lightCoord.xyz) * shadow;
29: #endif
```

4.2.5　Shadow 影の実装

ライティングの実装として最後に、影を実装してみます。

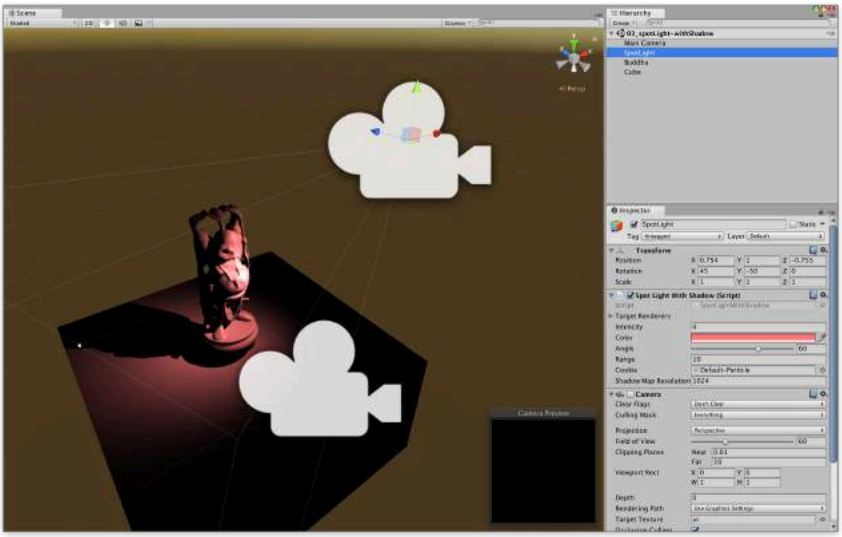
ライトから光が出て、直接光が当たったメッシュは明るくなり、ライトとメッシュの間に別の何かがあり、光が遮られたメッシュは暗くなります。これが影です。

手順としては、ざっくりと

- ライトから見た深度テクスチャを作成する。
- オブジェクトレンダリング時、その地点のライトからの深度が深度テクスチャ

よりも大きかったらその地点は他のオブジェクトに遮られているので影になる。

という形になります。今回はライトの位置からの深度テクスチャが必要になるので、SpotLight に Camera コンポーネントを付けて、ライトから見た深度テクスチャを作成します。



▲図 4.7　03_spotLight-withShadow.unity

SpotLightComponent（自作）に Camera（ビルトイン）が付いています。図 4.7

▼リスト 4.9　SpotLightWithShadow.cs

```
 1: Shader depthRenderShader {
 2:     get { return Shader.Find("Unlit/depthRender"); }
 3: }
 4:
 5: new Camera camera
 6: {
 7:     get
 8:     {
 9:         if (_c == null)
10:         {
11:             _c = GetComponent<Camera>();
12:             if (_c == null)
13:                 _c = gameObject.AddComponent<Camera>();
```

```
14:             depthOutput = new RenderTexture(
15:                 shadowMapResolution,
16:                 shadowMapResolution,
17:                 16,
18:                 RenderTextureFormat.RFloat
19:             );
20:             depthOutput.wrapMode = TextureWrapMode.Clamp;
21:             depthOutput.Create();
22:             _c.targetTexture = depthOutput;
23:             _c.SetReplacementShader(depthRenderShader, "RenderType");
24:             _c.clearFlags = CameraClearFlags.Nothing;
25:             _c.nearClipPlane = 0.01f;
26:             _c.enabled = false;
27:         }
28:         return _c;
29:     }
30: }
31: Camera _c;
32: RenderTexture depthOutput;
33:
34: void Update()
35: {
36:     if (mpb == null)
37:         mpb = new MaterialPropertyBlock();
38:
39:     var currentRt = RenderTexture.active;
40:     RenderTexture.active = depthOutput;
41:     GL.Clear(true, true, Color.white * camera.farClipPlane);
42:     camera.fieldOfView = angle;
43:     camera.nearClipPlane = 0.01f;
44:     camera.farClipPlane = range;
45:     camera.Render();
46:     //カメラのレンダリングはスクリプト上、マニュアルで行う
47:     RenderTexture.active = currentRt;
48:
49:     var projMatrix = camera.projectionMatrix;
50:     //プロジェクション行列は、カメラのものを使う
51:     var worldToLightMatrix = transform.worldToLocalMatrix;
52:
53:     ~~
54: }
```

C#スクリプトはほとんど影なしバージョンと同じなのですが、深度テクスチャをレンダリングするカメラと深度をレンダリングする ReplacementShader の設定をしています。また、今回はカメラがあるので、プロジェクション行列は、`Matrix4x4.Perspective`ではなく、`Camera.projectionMatrix`を使用します。

深度テクスチャ生成用シェーダは、次のコードになります。

▼リスト 4.10　depthRender.shader

```
1:     v2f vert (float4 pos : POSITION)
2:     {
3:         v2f o;
```

```
 4:         o.vertex = UnityObjectToClipPos(pos);
 5:         o.depth = abs(UnityObjectToViewPos(pos).z);
 6:         return o;
 7:     }
 8:
 9:     float frag (v2f i) : SV_Target
10:     {
11:         return i.depth;
12:     }
```

生成した深度テクスチャ（図 4.8）ライト座標系（カメラ座標系）におけるオブジェクトの位置の z 座標の値が出力されています。

▲図 4.8　light depthTexture

生成した深度テクスチャ（depthOutput）をメッシュオブジェクトに渡し、オブジェクトをレンダリングします。オブジェクトの影の計算部分は、次のようになります。

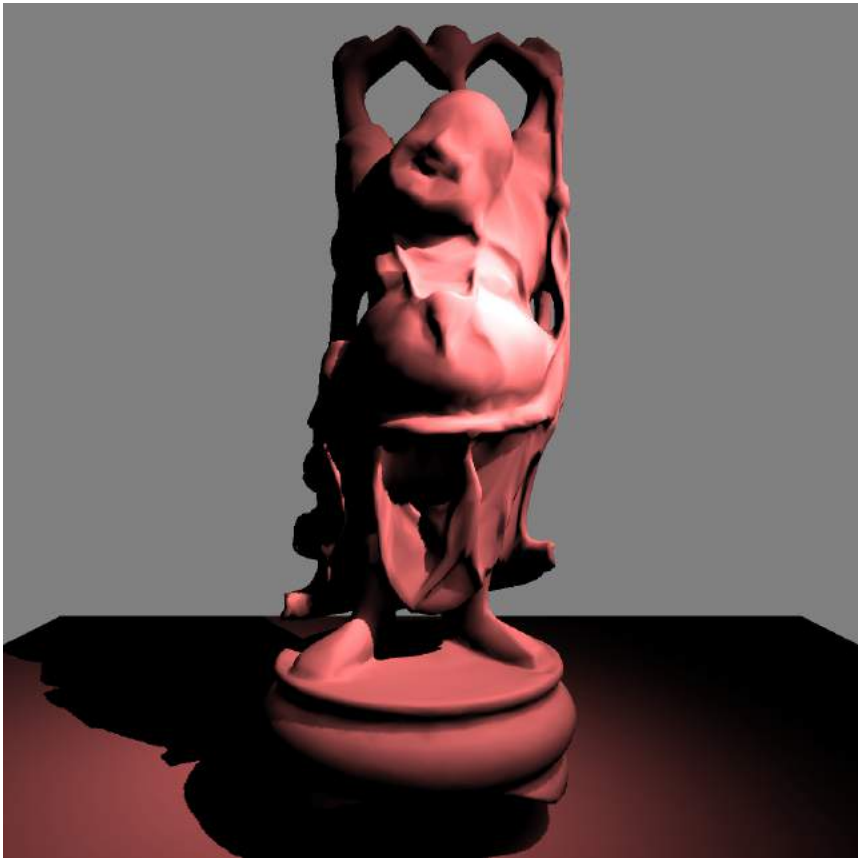
▼リスト 4.11　simple-spotLight-withShadow.shader

```
 1: fixed4 frag (v2f i) : SV_Target
 2: {
 3:     ///diffuse lighting
 4:     half3 to = i.worldPos - _LitPos.xyz;
 5:     half3 lightDir = normalize(to);
 6:     half dist = length(to);
 7:     half atten = _Intensity * dot(-lightDir, i.normal) / (dist * dist);
 8:
 9:     ///spot-light cookie
10:     half4 lightSpacePos = mul(_WorldToLitMatrix, half4(i.worldPos, 1.0));
11:     half4 projPos = mul(_ProjMatrix, lightSpacePos);
12:     projPos.z *= -1;
13:     half2 litUv = projPos.xy / projPos.z;
14:     litUv = litUv * 0.5 + 0.5;
15:     half lightCookie = tex2D(_Cookie, litUv);
16:     lightCookie *=
17:         0<litUv.x && litUv.x<1 && 0<litUv.y && litUv.y<1 && 0<projPos.z;
18:
19:     ///shadow
20:     half lightDepth = tex2D(_LitDepth, litUv).r;
21:     //_LitDepth にライトから見た深度テクスチャが渡されている
22:     atten *= 1.0 - saturate(10*abs(lightSpacePos.z) - 10*lightDepth);
23:
24:     half4 col = max(0.0, atten) * _LitCol * lightCookie;
25:     return col;
26: }
```

カメラによって作られた深度テクスチャ tex2D(_LitTexture, litUv).rと lightSpacePos.zは、どちらもライトから見たオブジェクトの頂点位置の z 値が格納されています。テクスチャである_LitTextureはライトから見えている面＝光が当たっている面の情報なので、深度テクスチャからサンプルした値（lightDepth）と lightSpacePos.zを比較して、影かどうかの判定をしている。

atten *= 1.0 - saturate(10*abs(lightSpacePos.z) - 10*lightDepth);

ここのコードで、lightDepthより lightSpacePos.zの値が大きかったら、サーフェイスは暗くなります。



▲図 4.9　03_spotLight-withShadow.unity

これで、スポットライトでオブジェクトの影を表示できました。

このスポットライトと影の実装を使って、オブジェクトにリアルタイムで色を塗る、スプレーの機能を実装していきます。

Camera.projectionMatrix と Matrix4x4.Perspective は同じ行列

Unity シーン：Example 内、**compareMatrix.unity**

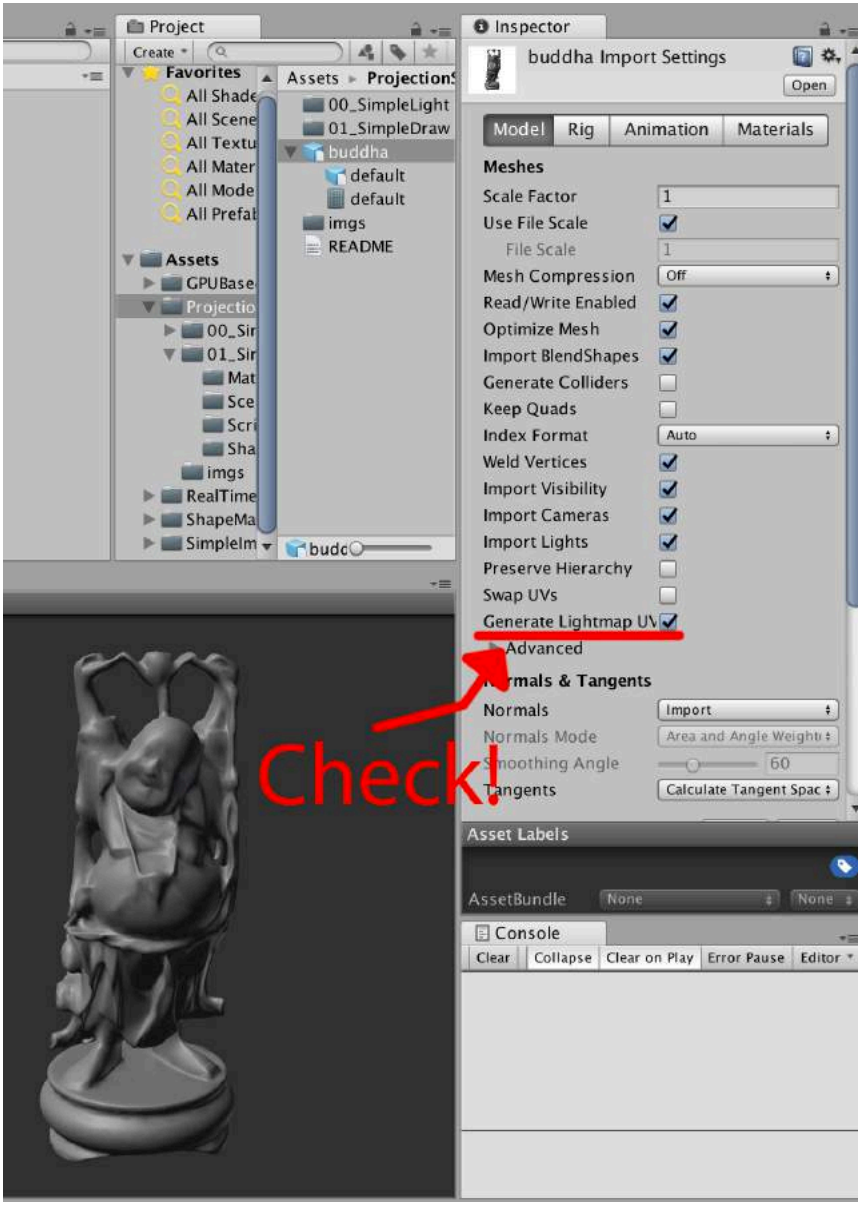
▼リスト 4.12　CompareMatrix.cs

```
 1:     float fov = 30f;
 2:     float near = 0.01f;
 3:     float far = 1000f;
 4:
 5:     camera.fieldOfView = fov;
 6:     camera.nearClipPlane = near;
 7:     camera.farClipPlane = far;
 8:
 9:     Matrix4x4 cameraMatrix = camera.projectionMatrix;
10:     Matrix4x4 perseMatrix = Matrix4x4.Perspective(
11:         fov,
12:         1f,
13:         near,
14:         far
15:     );
```

4.3　ProjectionSpray の実装

ここからは、自作した SpotLightComponent を応用して、オブジェクトに色を塗れるスプレーの機能を実装していきます。

基本的には、ライティングの強度の値をもとに、オブジェクトのテクスチャに描き込みを行います。今回使用している、Buddha のオブジェクトは uv データが存在しないので、そのままではテクスチャを貼ることができないのですが、Unity には LightMap 用の UV を生成する機能があります。



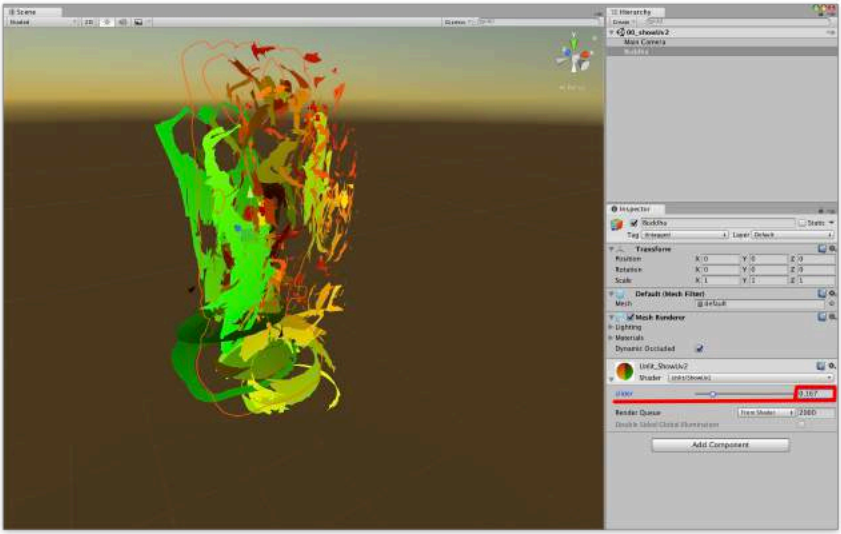
▲図 4.10　buddha Import Setting

モデルの ImportSetting において、"Generate Lightmap UVs"の項目にチェックを入れると、ライトマップ用の UV が生成されます。（`v.uv2 : TEXCOORD1`）この Uv2 用に描き込み可能な RenderTexture を作成し、描き込みを行います。

4.3.1 showUv2

サンプルシーンは、**00_showUv2.unity** を参照してください。

mesh.uv2 にマッピングするテクスチャに書き込むためには、メッシュから UV2 に展開したテクスチャを生成する必要があります。まずは、メッシュの頂点を UV2 の座標に展開するシェーダーを作ってみます。



▲図 4.11　00_showUv2.unity

シーン内で Buddha オブジェクトを選択し、マテリアルの"slider"のパラメータを操作すると、オブジェクトが元の形状から Uv2 に展開された形状に変化します。色付けは、`uv2.xy`を `color.rg`に割り当てています。

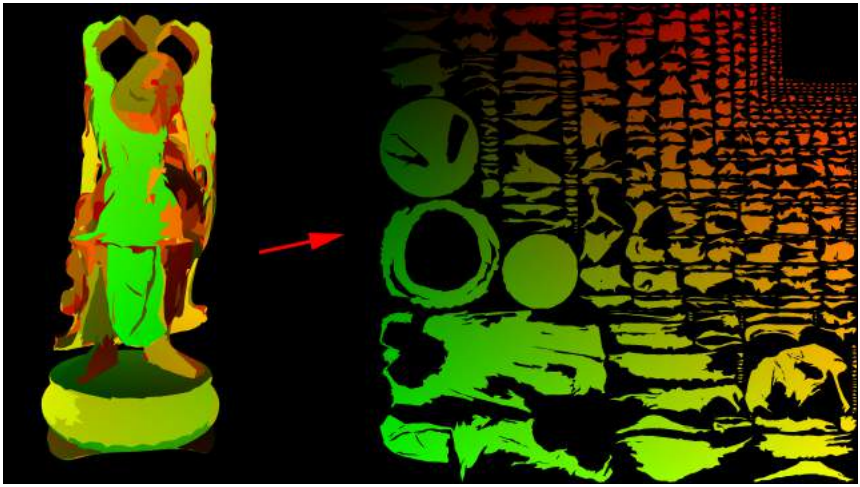
▼リスト 4.13　showUv2.shader

```
1: float _T;
2:
3: v2f vert(appdata v)
4: {
```

```
 5: #if UNITY_UV_STARTS_AT_TOP
 6:     v.uv2.y = 1.0 - v.uv2.y;
 7: #endif
 8:     float4 pos0 = UnityObjectToClipPos(v.vertex);
 9:     float4 pos1 = float4(v.uv2*2.0 - 1.0, 0.0, 1.0);
10:
11:     v2f o;
12:     o.vertex = lerp(pos0, pos1, _T);
13:     o.uv2 = v.uv2;
14:     o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
15:     o.normal = UnityObjectToWorldNormal(v.normal);
16:     return o;
17: }
18:
19: half4 frag(v2f i) : SV_Target
20: {
21:     return half4(i.uv2,0,1);
22: }
```

`float4 pos1 = float4(v.uv2*2.0 - 1.0, 0.0, 1.0);` の値が、クリップ座標系における Uv2 に展開された位置になります。リスト 4.13

フラグメントシェーダに `worldPos`と `normal`の値を渡しているので、この値を使用してスポットライトの計算におけるライティングの処理を行います。

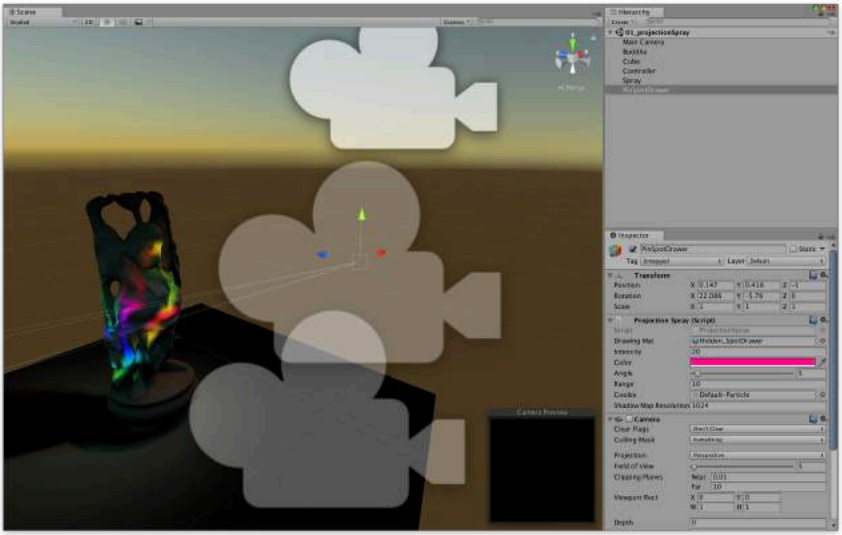


▲図 4.12　00_showUv2.unity

メッシュから Uv2 に展開したテクスチャを生成できる！

4.3.2 ProjectionSpray

それでは、準備が整ったので、スプレーの機能を実装します。**01_projectionSpray.unity** のシーンを参照してください。



▲図 4.13　01_projectionSpray.unity

このシーンを実行すると、黒い Buddha のオブジェクトにだんだん、色が付いていきます。そして、画面をクリックすると、その部分にスプレーでカラフルな色が塗られます。

実装の内容的には、今まで実装してきた自作スポットライトの応用になります。スポットライトのライティングの計算をそのままライティングに使用せず、`RenderTexture`の更新に使用しています。この例では、書き込んだテクスチャをライトマップ用に生成された `mesh.uv2`にマッピングしています。

`Drawable`はスプレーで書き込まれる対象のオブジェクトに付いているコンポーネントで、テクスチャに描き込む処理を行なっています。`ProjectionSpray`コンポーネントは、スプレーの位置などのテクスチャに描き込みを行う `Material`のプロパティの設定を行なっています。処理の流れとしては、`DrawableController`の `Update`関数内で、`projectionSpray.Draw(drawable)`を呼び、テクスチャに描き込みを行なっています。

ProjectionSpray.cs

- `Material drawMat`: 描き込みを行うためのマテリアル
- `UpdateDrawingMat()`: 描き込み前にマテリアルの設定を更新する
- `Draw(Drawable drawable)`: `drawMat`を `drawable.Draw(Material mat)` に渡して、描き込みを行う。

▼リスト 4.14　projectionSpray.cs

```
 1: public class ProjectionSpray : MonoBehaviour {
 2:
 3:     public Material drawingMat;
 4:
 5:     public float intensity = 1f;
 6:     public Color color = Color.white;
 7:     [Range(0.01f, 90f)] public float angle = 30f;
 8:     public float range = 10f;
 9:     public Texture cookie;
10:     public int shadowMapResolution = 1024;
11:
12:     Shader depthRenderShader {
13:         get { return Shader.Find("Unlit/depthRender"); }
14:     }
15:
16:     new Camera camera{get{~~}}
17:     Camera _c;
18:     RenderTexture depthOutput;
19:
20:     public void UpdateDrawingMat()
21:     {
22:         var currentRt = RenderTexture.active;
23:         RenderTexture.active = depthOutput;
24:         GL.Clear(true, true, Color.white * camera.farClipPlane);
25:         camera.fieldOfView = angle;
26:         camera.nearClipPlane = 0.01f;
27:         camera.farClipPlane = range;
28:         camera.Render();
29:         //深度テクスチャの更新
30:         RenderTexture.active = currentRt;
31:
32:         var projMatrix = camera.projectionMatrix;
33:         var worldToDrawerMatrix = transform.worldToLocalMatrix;
34:
35:         drawingMat.SetVector("_DrawerPos", transform.position);
36:         drawingMat.SetFloat("_Emission", intensity * Time.smoothDeltaTime);
37:         drawingMat.SetColor("_Color", color);
38:         drawingMat.SetMatrix("_WorldToDrawerMatrix", worldToDrawerMatrix);
39:         drawingMat.SetMatrix("_ProjMatrix", projMatrix);
40:         drawingMat.SetTexture("_Cookie", cookie);
41:         drawingMat.SetTexture("_DrawerDepth", depthOutput);
42:         //プロパティ名は違うけど、渡す情報は、スポットライトと同じ。
43:     }
44:
45:     public void Draw(Drawable drawable)
```

```
46:     {
47:         drawable.Draw(drawingMat);
48:         //描き込みの処理自体は、Drawable で行う。
49:         //描き込むための Material は ProjectionSpray が持っている。
50:     }
51: }
```

Drawable.cs

スプレーで描き込まれる対象のオブジェクト。描き込むためのテクスチャを持っています。Start()関数内で RenderTextureを作成しています。クラシックな Ping-pong Buffer を使用しています。

テクスチャに描き込みを行う部分の処理を見ていきましょう

▼リスト 4.15　Drawable.cs

```
 1:     //この関数は、projectionSpray.Draw(Drawable drawable) から呼ばれる
 2:     public void Draw(Material drawingMat)
 3:     {
 4:         drawingMat.SetTexture("_MainTex", pingPongRts[0]);
 5:         //描き込む対象のテクスチャの現状をマテリアルに設定。
 6:
 7:         var currentActive = RenderTexture.active;
 8:         RenderTexture.active = pingPongRts[1];
 9:         //描き込む対象のテクスチャを設定。
10:         GL.Clear(true, true, Color.clear);
11:         //描き込む対象のテクスチャをクリアする。
12:         drawingMat.SetPass(0);
13:         Graphics.DrawMeshNow(mesh, transform.localToWorldMatrix);
14:         //描き込む対象のメッシュとトランスフォーム値を使用してテクスチャを更新。
15:         RenderTexture.active = currentActive;
16:
17:         Swap(pingPongRts);
18:
19:         if(fillCrack!=null)
20:         {
21:             //Uv のつなぎ目にヒビができてしまうのを防ぐ処理です。
22:             Graphics.Blit(pingPongRts[0], pingPongRts[1], fillCrack);
23:             Swap(pingPongRts);
24:         }
25:
26:         Graphics.CopyTexture(pingPongRts[0], output);
27:         //更新した後のテクスチャを output にコピー
28:     }
```

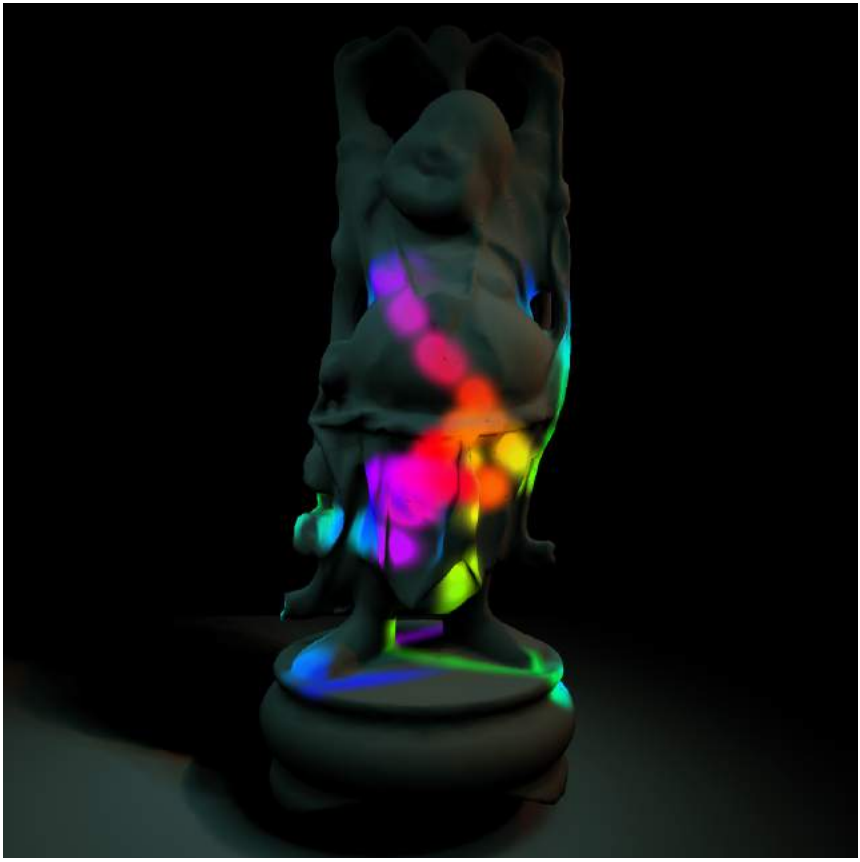
ここでのポイントは、Graphics.DrawMeshNow(mesh, matrix)を使って RenderTextureの更新を行なっているところです。（リスト 4.15）drawingMatの頂点シェーダでメッシュの頂点を mesh.uv2の形状に展開しているので、メッシュの頂点位置や法線、トランスフォーム情報をフラグメントシェーダに渡した上で、テクスチャの更新が可能になっています。（リスト 4.16）

▼リスト 4.16　ProjectionSpray.shader

```
 1: v2f vert (appdata v)
 2: {
 3:     v.uv2.y = 1.0 - v.uv2.y;
 4:     //y を反転しています！
 5:
 6:     v2f o;
 7:     o.vertex = float4(v.uv2*2.0 - 1.0, 0.0, 1.0);
 8:     //showUv2 と同じ処理です！
 9:     o.uv = v.uv2;
10:     o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
11:     o.normal = UnityObjectToWorldNormal(v.normal);
12:     return o;
13: }
14:
15: sampler2D _MainTex;
16:
17: uniform float4x4 _ProjMatrix, _WorldToDrawerMatrix;
18:
19: sampler2D _Cookie, _DrawerDepth;
20: half4 _DrawerPos, _Color;
21: half _Emission;
22:
23: half4 frag (v2f i) : SV_Target
24: {
25:     ///diffuse
26:     half3 to = i.worldPos - _DrawerPos.xyz;
27:     half3 dir = normalize(to);
28:     half dist = length(to);
29:     half atten = _Emission * dot(-dir, i.normal) / (dist * dist);
30:
31:     ///spot cookie
32:     half4 drawerSpacePos = mul(
33:         _WorldToDrawerMatrix,
34:         half4(i.worldPos, 1.0)
35:     );
36:     half4 projPos = mul(_ProjMatrix, drawerSpacePos);
37:     projPos.z *= -1;
38:     half2 drawerUv = projPos.xy / projPos.z;
39:     drawerUv = drawerUv * 0.5 + 0.5;
40:     half cookie = tex2D(_Cookie, drawerUv);
41:     cookie *=
42:         0<drawerUv.x && drawerUv.x<1 &&
43:         0<drawerUv.y && drawerUv.y<1 && 0<projPos.z;
44:
45:     ///shadow
46:     half drawerDepth = tex2D(_DrawerDepth, drawerUv).r;
47:     atten *= 1.0 - saturate(10 * abs(drawerSpacePos.z) - 10 * drawerDepth);
48:     //ここまでは、スポットライトの処理と同じです！
49:
50:     i.uv.y = 1 - i.uv.y;
51:     half4 col = tex2D(_MainTex, i.uv);
52:     //_MainTex には、drawable.pingPongRts[0] が割り当てられています
53:     col.rgb = lerp(
54:         col.rgb,
55:         _Color.rgb,
56:         saturate(col.a * _Emission * atten * cookie)
```

```
57:     );
58:     //ここが、描き込みを行う処理です！
59:     //計算したライティングの強度によって元のテクスチャから描き込みカラーに補完処
理しています。
60:
61:     col.a = 1;
62:     return col;
63:     //値は、drawable.pingPongRts[1] に出力されます
64: }
```

これで、3D モデルに対するスプレーによる描き込みができるようになりました。（図 4.14）



▲図 4.14　01_projectionSpray.unity

4.4 まとめ

UnityCG.cginc や Lighting.cginc などを見るとビルトインの処理が書いてあって、いろいろな処理を実装する参考になるので見てみるとよいです！

第 5 章

プロシージャルノイズ入門

5.1 はじめに

本章では、コンピュータグラフィックスにおいて用いられるノイズについての解説を行います。ノイズは、1980 年代、テクスチャマッピングのための画像生成の新しい手法として開発されました。オブジェクトに画像を貼って、その複雑性を演出するテクスチャマッピングは、今日の CG においてはよく知られている手法ですが、その当時のコンピュータは非常に限られた少ない記憶領域しか持っておらず、テクスチャマッピングに画像データを使用することはハードウェアと相性が良いとは言えませんでした。そこで、このノイズパターンを手続き的に生成する手法が考案されました。山、砂漠のような地形、雲、水面、炎、大理石、木目、岩、水晶、泡の膜といった自然界に存在する物質や現象は、視覚的な複雑性と、規則的なパターンを持っています。ノイズはこのような自然界に存在する物質や現象の表現に最適なテクスチャパターンを生成することができ、プロシージャルにグラフィックスを生成したいときに欠かせないテクニックとなりました。代表的なノイズアルゴリズムに、**Ken Perlin** の業績である **Perlin Noise**、**Simplex Noise** というものがあります。ここでは、数多あるノイズの応用への足がかりとして、これらのノイズのアルゴリズムの解説と、シェーダによる実装を中心に説明していきたいと思います。

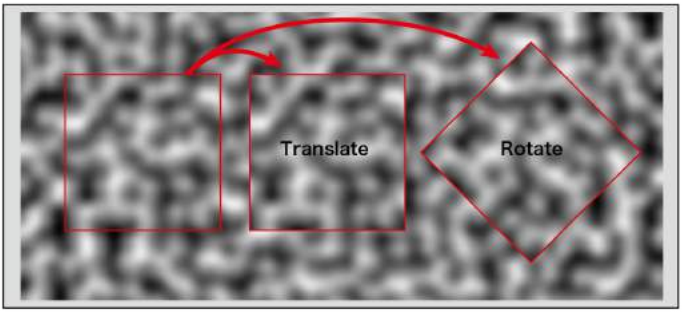
この章のサンプルデータは、共通 Unity サンプルプロジェクトの
Assets/**TheStudyOfProceduralNoise**
内にあります。合わせてご参照ください。

5.2 ノイズとは

ノイズ（noise）という言葉は、音響の分野では雑音と訳されうるさい音を意味し、映像の分野でも画像の荒れを示したり、処理したい内容に対して不必要な情報一般を指して日常的にも使用します。コンピュータグラフィックスにおけるノイズは、N 次

元のベクトルを入力として、以下のような特徴をもつランダムなパターンのスカラー値（1 次元の値）を返す関数を指します。

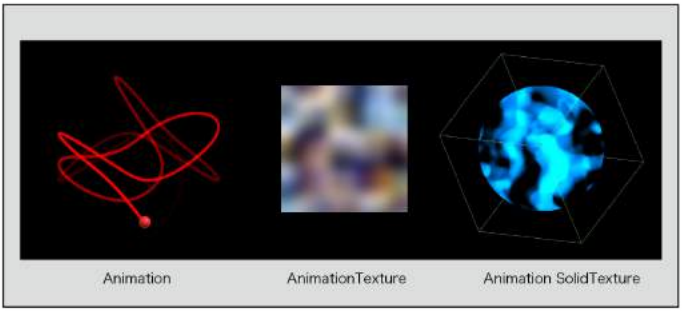
- 隣接する領域に対して、連続的に変化する
- 回転に対して、特徴が統計的に不変（特定の領域を切り取って回転させたとしても、特徴に変化がない）（=等方性がある）
- 移動に対して、特徴が統計的に不変（特定の領域を切り取って移動させたとしても、特徴に変化がない）
- 信号として捉えたときに、周波数の帯域が限定される（ほとんどのエネルギーは、特定の周波数スペクトルに集中している）



▲図 5.1　ノイズの特徴

ノイズは、N 次元のベクトルを入力として受けることで、以下のような用途に使用することができます。

- アニメーション ・・・ 1D（時間）
- テクスチャ ・・・ 2D（オブジェクトの UV 座標）
- アニメーションするテクスチャ ・・・ 3D（オブジェクトの UV 座標＋時間）
- ソリッド（3D）テクスチャ ・・・ 3D（オブジェクトのローカル座標）
- アニメーションするソリッドテクスチャ ・・・ 4D（オブジェクトのローカル座標＋時間）



▲図 5.2　ノイズの応用例

5.3　ノイズのアルゴリズムについての解説

Value Noise、**Perlin Noise**、**Improved Perlin Noise**（改良パーリンノイズ）、**Simplex Noise** についてのアルゴリズムの解説を行っていきます。

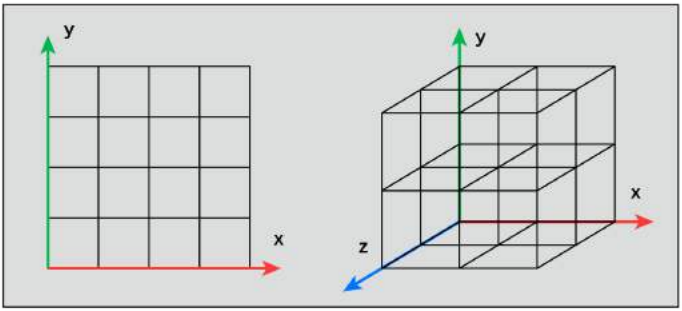
5.3.1　Value Noise (バリューノイズ)

ノイズ関数としての条件や精度を厳密には満たしていませんが、最も実装が容易でノイズについての理解を助けるものとして、**Value Noise** というノイズアルゴリズムを紹介します。

アルゴリズム

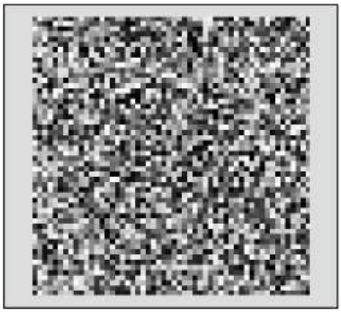
1. 空間上のそれぞれの軸に一定の間隔で配置された格子点を定義する
2. それぞれの格子点について、疑似乱数の値を求める
3. それぞれの格子点と格子点の間の点の値を、補間によって求める

■格子を定義　2 次元の場合、x、y 軸それぞれに等間隔な格子を定義します。格子は正方形の形状をしており、このそれぞれの格子点において、格子点の座標値を参照した疑似乱数の値を計算します。3 次元の場合は、x、y、z 軸それぞれに等間隔な格子を定義し、格子の形状は立方体になります。



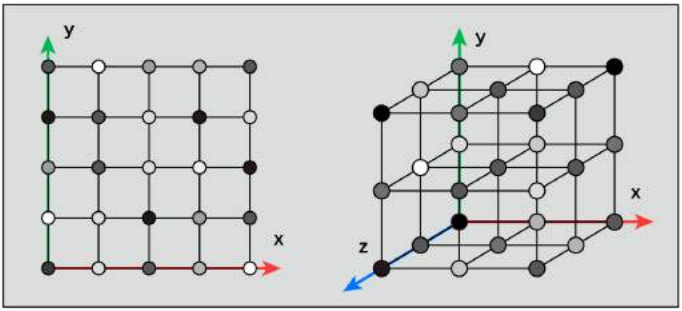
▲図 5.3　格子（2 次元）, 格子（3 次元）

■疑似乱数（**Pseudo Random**）の生成　乱数というのは、無秩序に、出現の確率が同じになるように並べられた数字の列を言います。乱数にも真の乱数と疑似乱数と呼ばれるものがあり、例えば、サイコロを振るとき、今まで出た目から次に出る目を予測することは不可能であり、このようなものを真の乱数と呼びます。これに対して、規則性と再現性があるものを、疑似乱数（**Pseudo Random**）と呼びます。（コンピュータで乱数列を生成する場合、確定的な計算で求めるので、生成された乱数はほとんどは疑似乱数ということができます。）ノイズの計算を行う場合、共通のパラメータさえ用いれば、同じ結果が得られる疑似乱数を使用します。



▲図 5.4　擬似乱数

この擬似乱数を生成する関数の引数に、それぞれの格子点の座標値を与えることで、格子点ごとに固有の擬似乱数の値を得ることができます。



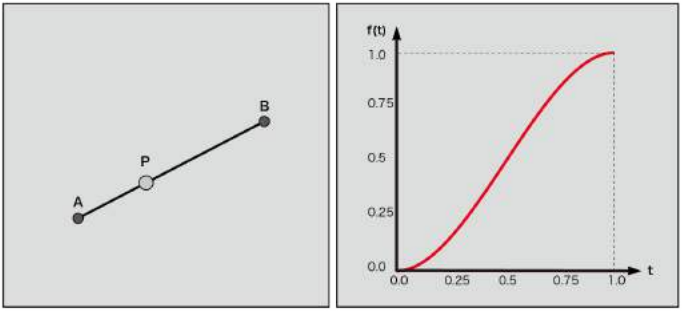
▲図 5.5　各格子点上の擬似乱数

■補間（**Interpolation**） A と B という値があり、その間の P の値が、A から B に直線的に変化するとして、その値を近似的に求めることを線形補間（**Linear Interpolation**）と言います。もっとも単純な補間方法ですが、これを使って、格子点の間の値を求めるとすると、補間の始点と終点（格子点付近）で、値の変化が鋭利になってしまいます。

そこで、スムーズに値が変化するように、**3** 次エルミート曲線を補間係数に使用します。

$$f(t) = 3t^2 - 2t^3$$

これを `t=0`から `t=1`へ変化させたとき、値は右下図のようになります。

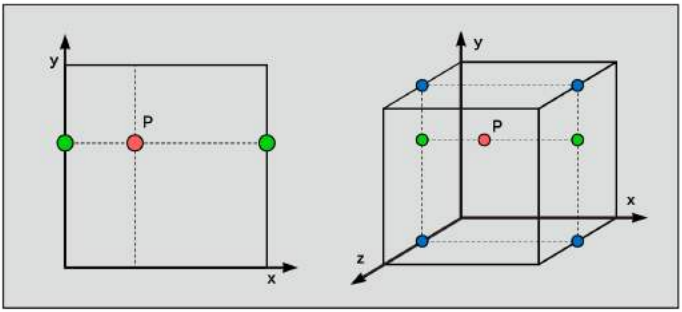


▲図 5.6　2 次元平面での線形補間（左）, 3 次エルミート曲線

※ 3 次エルミート曲線は、GLSL、HLSL では、**smoothstep** 関数として実装され

ています。

この補間関数を使って、それぞれの格子点で求めた値をそれぞれの軸で補間します。2 次元の場合、まず格子の両端で x についての補間を行い、次にそれらの値を y 軸についての補間し、合計 3 回の計算を行います。3 次元の場合は、下の図のように、z 軸について 4 つ、y 軸について 2 つ、x 軸について 1 つ、合計 7 回の補間を行います。



▲図 5.7　補間（2 次元空間）, 補間（3 次元空間）

実装

2 次元について説明いたします。各格子点の座標を求めます。

```
floor()
```

整数部については、floor()関数を使って求めます。floor()は、入力された実数に対してそれ以下の最小の整数を返す関数です。1.0 以上の実数を入力値に与えた場合、1,2,3…という値が得られ等間隔に同じ値が得られるため、これを格子の座標値として用いることができます。

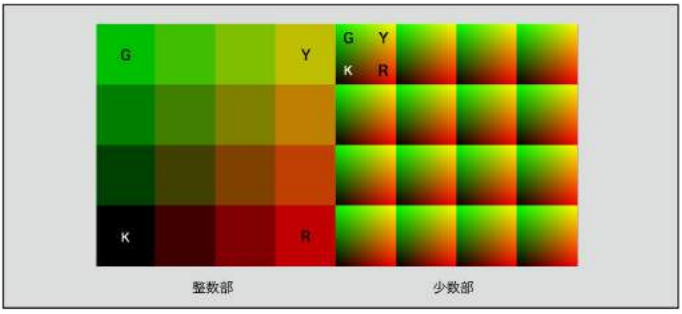
小数部については、frac()関数を使って求めます。

```
frac()
```

frac()は、与えられた実数値の、小数部の値を返し、0 以上 1 未満の値をとります。これにより、それぞれの格子内部の座標値を得ることができます。

```
// 格子点の座標値
float2 i00 = i;
float2 i10 = i + float2(1.0, 0.0);
float2 i01 = i + float2(0.0, 1.0);
float2 i11 = i + float2(1.0, 1.0);
```

上で求めた座標値を、フラグメントカラーの R と G に割り当てると以下のような画像が得られます。（整数部については、1 以上の値を取り得るため、視覚化のため結果が 1 を超えないようにスケーリングを施しています）



▲図 5.8　整数部と少数部を RG として描画したもの

■疑似乱数生成関数　random 関数をインターネットで検索すると、よくこの関数が結果として返されます。

```
float rand(float2 co)
{
    return frac(sin(dot(co.xy, float2(12.9898,78.233))) * 43758.5453);
}
```

1 つ 1 つ処理を見ていくと、まず入力された 2 次元のベクトルを内積で 1 次元に丸め扱いやすくし、それを sin 関数の引数として与え、大きな数を掛け合わせ、その小数部を求めるというもので、これにより、規則性と再現性はあるが、無秩序に連続した値を得ることができます。

この関数については、出自が定かではなく、

https://stackoverflow.com/questions/12964279/whats-the-origin-of-this-glsl-rand-one-liner

によると、1998 年に発表された **"On generating random numbers, with help of y= [(a+x)sin(bx)] mod 1"** という論文という論文が起源であると書か

れています。

シンプルで扱いやすい反面、同じ乱数列が出てきてしまう周期が短く、大きな解像度のテクスチャであると、視覚的に確認できてしまうパターンが発生し、あまり良い疑似乱数とは言えません。

```
// 格子点の座標上での疑似乱数の値
float n00 = pseudoRandom(i00);
float n10 = pseudoRandom(i10);
float n01 = pseudoRandom(i01);
float n11 = pseudoRandom(i11);
```

それぞれの格子点の座標値（整数）を、疑似乱数の引数に与えることで、各格子点上でのノイズの値を求めます。

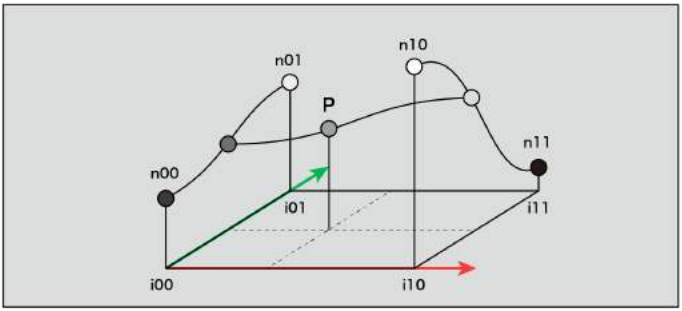
■補間（Interpolation）

```
// 補間関数（3 次エルミート曲線）= smoothstep
float2 interpolate(float2 t)
{
    return t * t * (3.0 - 2.0 * t);
}
```

```
// 補間係数を求める
float2 u = interpolate(f);
// 2 次元格子の補間
return  lerp(lerp(n00, n10, u.x), lerp(n01, n11, u.x), u.y);
```

事前に定義した `interpolate()`関数で補間係数を算出します。格子の小数部を引数とすることで、格子の始点と終点付近で滑らかに変化する曲線が得られます。

`lerp()`は 線形補間を行う関数で、**Linear Interpolate** の略です。第 1 引数と第 2 引数に与えた値の線形補間された値を計算することができ、第 3 引数に補間係数として求めた **u** を代入することで、格子間の値を滑らかにつなげます。



▲図 5.9　格子点の補間（2 次元空間）

結果

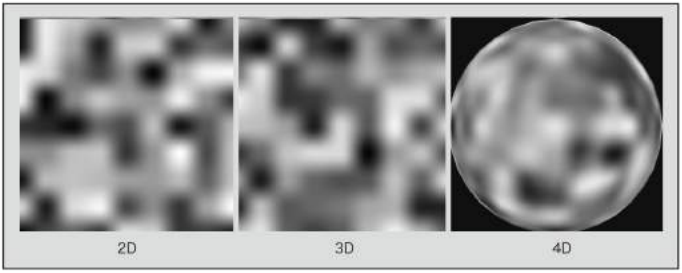
サンプルプロジェクト内の

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

シーンを開くと、**Value Noise** の実装結果を見ることができます。コードについては、

- Shaders/ProceduralNoise/**ValueNoise2D.cginc**
- Shaders/ProceduralNoise/**ValueNoise3D.cginc**
- Shaders/ProceduralNoise/**ValueNoise4D.cginc**

に実装をしたものがあります。



▲図 5.10　Value Noise（2D, 3D, 4D） 描画結果

結果画像を見てみると、ある程度格子の形状が見えてしまうことがわかります。こ

のように **Value Noise** は、実装は容易ですが、ある領域を回転させたときに特徴が不変であるという等方性が保証されておらず、ノイズというには不十分です。しかし、**Value Noise** の実装で行った、「規則的に配置された格子点で求められた疑似乱数の値を補間して空間上のすべての点の連続的でスムースな値を求める」というプロセスは、ノイズ関数の基本的なアルゴリズムの構造をしています。

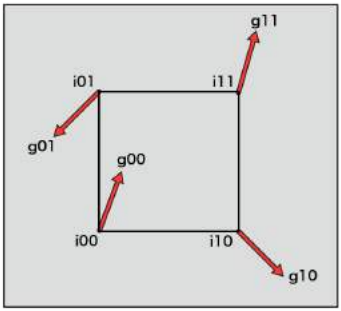
5.3.2 Perlin Noise (パーリンノイズ)

Perlin Noise は、プロシージャルノイズの伝統的、代表的手法であり、その名の主である **Ken Perlin** によって開発されました。もともとは、世界で初めて全面的にコンピュータグラフィックスを導入した映画として知られる、1982 年に製作されたアメリカの SF 映画「Tron」の視覚表現ためのテクスチャ生成の実験の中で生み出され、その成果は、1985 年の SIGGRAPH に **"An Image Synthesizer"** というタイトルの論文にまとめられ発表されました。

アルゴリズム

1. 格子（Lattice）の座標を求める
2. 格子点上の勾配（Gradient）を求める
3. 各格子点から、格子の中の点 P へのベクトルを求める
4. 2 で求めた勾配と、3 で求めたベクトルの内積を計算し、各格子点上のノイズの値を計算する
5. 3 次エルミート曲線で、4 で求めた各格子点のノイズの値を補間する

■勾配（Gradient） Value Noise と異なる点は、格子点のノイズの値を 1 次元の値で定義するのではなく、傾きを持った勾配（**Gradient**）として定義するところです。2 次元であれば 2 次元の勾配、3 次元であれば 3 次元の勾配を定義します。

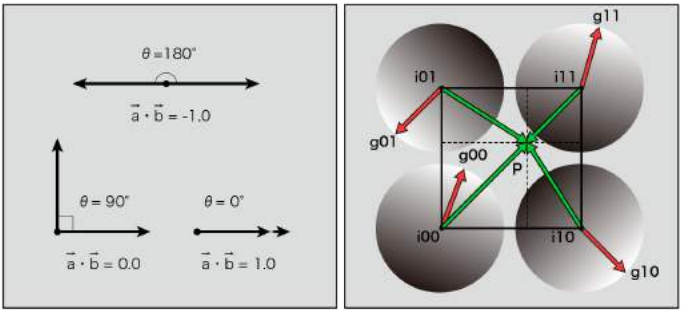


▲図 5.11　Perlin Noise 勾配ベクトル

■内積（**Dot Product**）　内積は、

$$\overrightarrow{a} \cdot \overrightarrow{b} = |a|\,|b| \cos\theta = (a.x * b.x) + (a.y * b.y)$$

で定義されるベクトルの演算で、幾何学的な意味は、2 つのベクトルがどれぐらい同じ方向を向いているかを示す比であり、内積のとる値は、同じ方向→ **1**、直交→ **0**、逆向き→**-1** となります。つまり、勾配と、各格子点から格子内のノイズの値を求めたい点 P へ向かうベクトルの内積を求めるということは、それらのベクトルが同じ方向を向いていれば、高いノイズの値が、異なる方向を向いていれば小さい値が返されることになります。



▲図 5.12　内積（左） Perlin Noise 勾配と内挿ベクトル（右）

■補間（Interpolation） ここでは、3 次エルミート曲線を補間のための関数として用いていますが、後に、Ken Perlin は、5 次エルミート曲線に修正しています。それについては、**Improved Perlin Noise**（改良パーリンノイズ）の項で説明いたします。

実装

サンプルプロジェクト内の
TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**
シーンを開くと、**Perlin Noise** の実装結果を見ることができます。コードについては、

- Shaders/ProceduralNoise/**OriginalPerlinNoise2D.cginc**
- Shaders/ProceduralNoise/**OriginalPerlinNoise3D.cginc**
- Shaders/ProceduralNoise/**OriginalPerlinNoise4D.cginc**

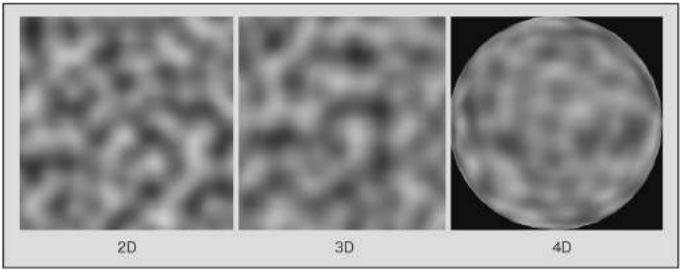
に実装をしたものがあります。
2 次元についての実装を掲載します。

```
// Original Perlin Noise 2D
float originalPerlinNoise(float2 v)
{
    // 格子の整数部の座標
    float2 i = floor(v);
    // 格子の小数部の座標
    float2 f = frac(v);

    // 格子の 4 つの角の座標値
    float2 i00 = i;
    float2 i10 = i + float2(1.0, 0.0);
    float2 i01 = i + float2(0.0, 1.0);
    float2 i11 = i + float2(1.0, 1.0);

    // 格子内部のそれぞれの格子点からのベクトル
    float2 p00 = f;
    float2 p10 = f - float2(1.0, 0.0);
    float2 p01 = f - float2(0.0, 1.0);
    float2 p11 = f - float2(1.0, 1.0);

    // 格子点それぞれの勾配
    float2 g00 = pseudoRandom(i00);
    float2 g10 = pseudoRandom(i10);
    float2 g01 = pseudoRandom(i01);
    float2 g11 = pseudoRandom(i11);

    // 正規化（ベクトルの大きさを 1 にそろえる）
    g00 = normalize(g00);
    g10 = normalize(g10);
    g01 = normalize(g01);
```

```
    g11 = normalize(g11);

    // 各格子点のノイズの値を計算
    float n00 = dot(g00, p00);
    float n10 = dot(g10, p10);
    float n01 = dot(g01, p01);
    float n11 = dot(g11, p11);

    // 補間
    float2 u_xy = interpolate(f.xy);
    float2 n_x  = lerp(float2(n00, n01), float2(n10, n11), u_xy.x);
    float n_xy = lerp(n_x.x, n_x.y, u_xy.y);
    return n_xy;
}
```

結果

Value Noise で見られたような不自然な格子の形状はなく、等方性のあるノイズが得られます。**Perlin Noise** は、**Value Noise** に対して、勾配を用いることから **Gradient Noise** とも呼ばれます。



▲図 5.13　Perlin Noise（2D, 3D, 4D） 結果

5.3.3　Improved Perlin Noise (改良パーリンノイズ)

Improved Perlin Noise（改良パーリンノイズ）は、Ken Perlin 氏によって、**Perlin Noise** の欠点を改良するものとして 2001 年に発表されました。詳細については、ここで確認することができます。

http://mrl.nyu.edu/~perlin/paper445.pdf

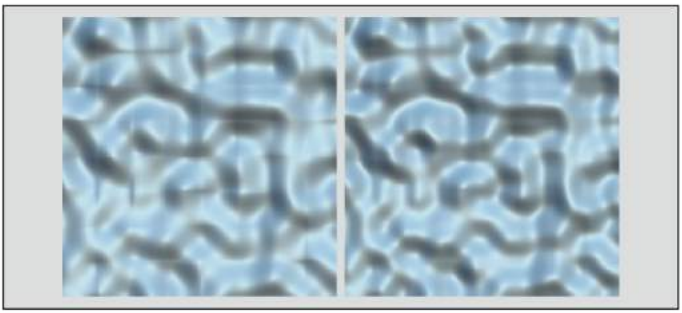
現在、**Perlin Noise** というと、この **Improved Perlin Noise** に基づいて実装されたものがほとんどです。

Ken Perlin が行った改良とは、主に次の 2 点です。

1. 格子間の勾配の補間のための補間関数
2. 勾配の計算法

■格子間の勾配の補間のための補間関数　補間のエルミート曲線について、オリジナルの **Perlin Noise** では **3** 次エルミート曲線を用いました。しかし、この 3 次の式であると、2 階微分（微分して得られた結果がさらに微分可能なとき、これを微分する事）`6-12t`が `t=0`, `t=1`の時に `0`になりません。曲線を微分すると、接線の傾きが得られます。もう 1 回微分すると、その曲率が得られ、これがゼロでないということはわずかな変化があるということです。このため、バンプマッピングのための法線として用いる場合、隣接する格子と値が厳密に連続にならなず、視覚的なアーティファクトが発生します。

比較した図です。



▲図 5.14　3 次エルミート曲線による補間（左） 5 次エルミート曲線による補間（右）

サンプルプロジェクト

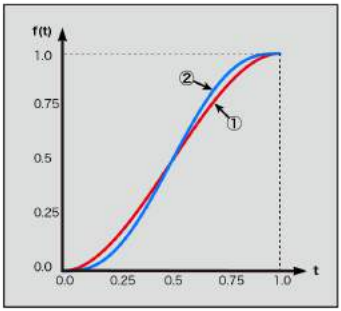
TheStudyOfProceduralNoise/Scenes/**CompareBumpmap**

シーンを開くと、これを確認することができます。

図を見てみると、左の **3** 次エルミート曲線によって補間を行った方は、格子の境界で視覚的に不自然な法線の不連続が認められます。これを回避するために、次の **5** 次エルミート曲線を用います。

$$f\left(t\right) = 6t^5 - 15t^4 + 10t^3$$

それぞれの曲線図を示します。①は **3** 次エルミート曲線、②は **5** 次エルミート曲線です。

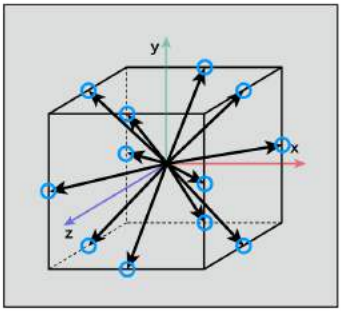


▲図 5.15　3 次と 5 次のエルミート曲線

t=0, t=1あたりでなめらな変化をしていることがわかります。1 階微分、2 階微分ともに t=0または t=1の時に 0となるので、連続性が保たれます。

■勾配の計算　3 次元について考えます。勾配 G は球状に均一に分布していますが、立方体格子はその軸に対しては短く、その対角線については長く、それ自体方向的な偏りを持っています。勾配が軸と平行に近い場合、それが近接するものと整列すると、距離が近いためそれらの領域では異常に高い値をとなり、斑点に見えるようなノイズの分布を生じさせることがあります。この勾配の偏りを取り除くために、座標軸に平行なものと、対角線上にあるものを取り除いた以下の 12 のベクトルに限定することを行います。

```
(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),
(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)
```



▲図 5.16　改良パーリンノイズの勾配（3 次元）

認知心理学的な見地から、実際には、格子の中の点 P が十分なランダム性を与えてくれるのもあり、勾配 G が全方位にランダムである必要はないと Ken Perlin は述べています。また、たとえば、`(1, 1, 0)`と`(x, y, z)`の内積は、単純に`x + y`として計算することができ、後に行う内積計算を単純化し、多くの乗算を避けることができます。これによって 24 個の乗算が計算から取り除かれ、計算コストを抑えることができます。

実装と結果

サンプルプロジェクト内の

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

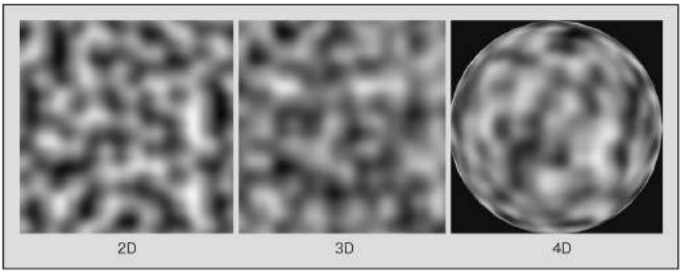
シーンを開くと、**Improved Perlin Noise** の実装結果を見ることができます。コードについては、

- Shaders/ProceduralNoise/**ClassicPerlinNoise2D.cginc**
- Shaders/ProceduralNoise/**ClassicPerlinNoise3D.cginc**
- Shaders/ProceduralNoise/**ClassicPerlinNoise4D.cginc**

この **Improved Perlin Noise** の実装は、次の **Simplex Noise** でも紹介する、論文**"Effecient computational noise in GLSL"** において発表されたものに基づいています。（ここでは、**Classic Perlin Noise** という名称で扱われています。そのため、少しややこしいですが、その名前を使用しています。）。この実装は、勾配計算について、Ken Perlin が論文で説明したものとは異なりますが、十分に類似した結果を得ることができます。

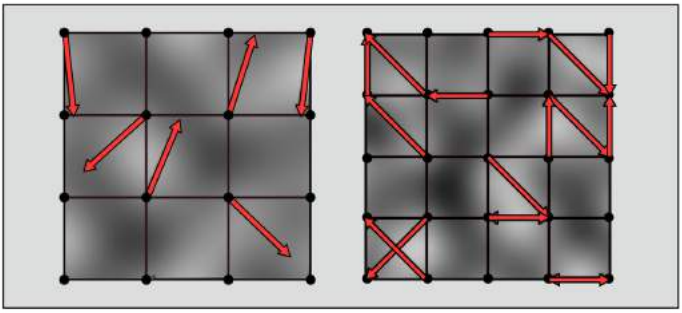
下記の URL から Ken Perlin のオリジナルの実装を確認することができます。

http://mrl.nyu.edu/~perlin/noise/



▲図 5.17　Improved Perlin Noise（2D, 3D, 4D）

下の図は、ノイズの勾配と結果を比較したものです。左がオリジナルの **Perlin Noise**、右が **Improved Perlin Noise** です。



▲図 5.18　Perlin Noise, Improved Perlin Noise の勾配と結果の比較

5.3.4　Simplex Noise (シンプレックスノイズ)

Simplex Noise は、Ken Perlin によって、従来の **Perlin Noise** よりも優れたアルゴリズムとして 2001 年に発表されました。

Simplex Noise は従来の **Perlin Noise** と比較して、以下のような優位性があります。

- 計算の複雑性が低く、乗算の回数が少ない。
- ノイズの次元を 4 次元、5 次元またはそれ以上と上げていったとき、計算負荷の増加が少なく、**Perlin Noise** が $O(2^N)$ の計算オーダーであるところ、

Simplex Noise は $O(N^2)$ で済む

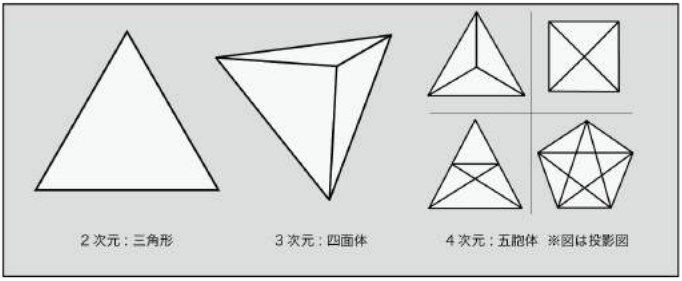
- 勾配ベクトルの方向的な偏りが引き起こす視覚的なアーティファクトがない
- 少ない計算負荷で、連続的な勾配がある
- ハードウェア（シェーダ）で実装しやすい

ここでは、**"Simplex Noise Demystify"**
http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
の内容をもとに解説いたします。

アルゴリズム

1. 単体（Simplex）による格子を定義する
2. ノイズの値を求める点 P がどの単体にあるかを計算
3. 単体の角における勾配を計算
4. それぞれの単体の周囲の角における勾配の値から、点 P におけるノイズの値を計算

■単体 (Simplex) による格子　シンプレックスとは、数学の位相幾何学においては、単体と呼ばれます。単体とは、図形を作る一番小さな単位のことです。0 次元単体は点、1 次元単体は線分、2 次元単体は三角形、3 次元単体は四面体、4 次元単体は五胞体です。

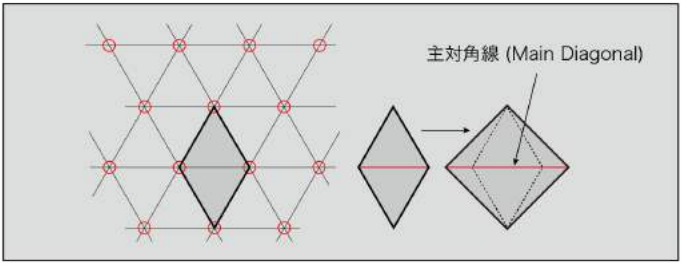


▲図 5.19　それぞれの次元での単体

Perlin Noise では、2 次元の時は正方形の格子、3 次元の時は立方体の格子を用いていましたが、**Simplex Noise** ではこの単体を格子に使います。

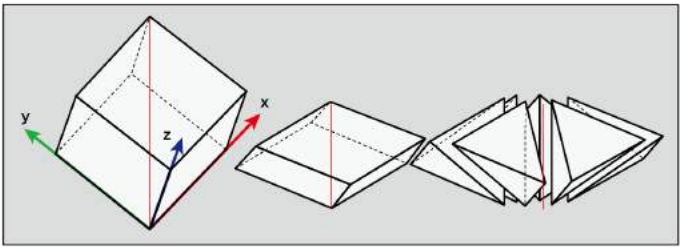
1 次元の場合、空間を充てんする最も単純な形状は、等間隔に配置された線です。2 次元の場合、空間を充てんする最も単純な形状は三角形となります。

これらの三角形で構成されたタイルのうち二つは、その主対角線に沿って押しつぶされた正方形と考えることができます。



▲図 5.20　2 次元の単体格子

3 次元では、単体形状はわずかに歪んだ四面体です。それら 6 つの四面体は主対角線に沿って押しつぶされた立方体を作ります。



▲図 5.21　3 次元の単体格子

4 次元では、単体形状は非常に視覚化が困難です。その単体形状は 5 つの角を持っており、24 個のこれらの形状が、主対角線に沿ってつぶれた 4 次元の超立方体を形成します。

N 次元の単体形状は、**N+1** 個の角を持ち、**N!**（**3!**は **3** × **2** × **1=6)** 個のそれらの形状は、主対角線に沿ってつぶれた N 次元超立方体を満たす、と言えます。

格子に単体形状を用いる利点は、次元に対して可能な限り少ない角をもつ格子を定義できるので、格子の内部の点の値を求めるとき、周囲の格子点の値から補間を行いますが、その計算回数を抑えることができるところにあります。N 次元の超立方体は

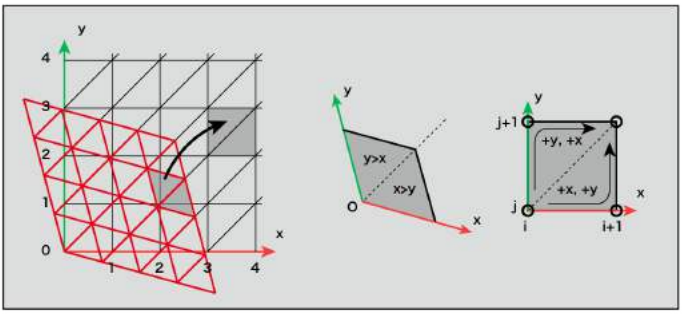
2^N 個の角を持ちますが、N 次元の単体形状は $N+1$ 個だけの角しか持ちません。

より高い次元のノイズの値を求めようとするとき、従来の **Perlin Noise** では、超立方体のそれぞれの角における計算の複雑性や、それぞれの主軸についての補間の計算量は $O(2^N)$ 問題であり、すぐに扱いにくいものになります。一方、**Simplex Noise** では、次元に対する単体形状の頂点数が少ないため、その計算量は $O(N^2)$ に抑えられます。

■ノイズの値を求める点 P が、どの単体にあるかの決定　Perlin Noise では、求めたい点 P が、どの格子にあるかの計算は、座標の整数部を `floor()`で求めることができました。Simplex Noise では、以下に示す 2 つの手順で行います。

1. 主対角線に沿って入力座標空間を歪曲させ、それぞれの軸の座標の整数部を見ることで、どの単体のユニットに属するかを判断する
2. 単体のユニットの原点から、点 P へのそれぞれの次元での距離の大きさを比較することで、単体のユニットのどの単体に属するかを判断する

視覚的な理解のために、2 次元の場合についての図を見ていきましょう。

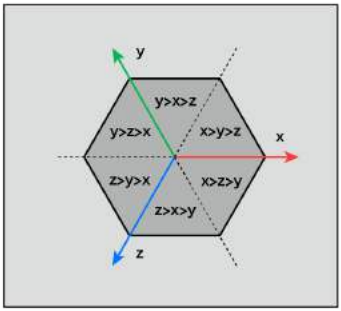


▲図 5.22　2 次元の場合の単体格子の変形の様子

2 次元の三角形の単体格子は、スケーリングによって二等辺三角形の格子にゆがめることができます。二等辺三角形は 2 つで辺の長さが 1 の四角形（単体のユニットというのはこの四角形を指します）を形成します。移動後の座標 `(x, y)`の整数部を見ることによって、ノイズの値を求めたい点 P がどの単体のユニットの四角形にあるかを判断することができます。また、単体のユニットの原点からの x、y の大きさを比較することによって、点 P を含む単体がユニットのどちらであるかがわかり、点 P を囲む単体の 3 点の座標が決まります。

3 次元の場合、2 次元の正三角形の単体格子を二等辺三角形の格子へと変形させる

ことができるように、3 次元の単体格子は、その主対角線に沿ってスケーリングすることで、規則正しく並んだ立方体の格子に変形させることができます。2 次元の場合と同様、移動した点 P の座標の整数部を見ることで、どの 6 個からなる単体のユニットに属するかを判定できます。そしてさらに、そのユニットのどの単体に属するかは、ユニットの原点からの各軸の相対的な大きさ比較で判定することができます。

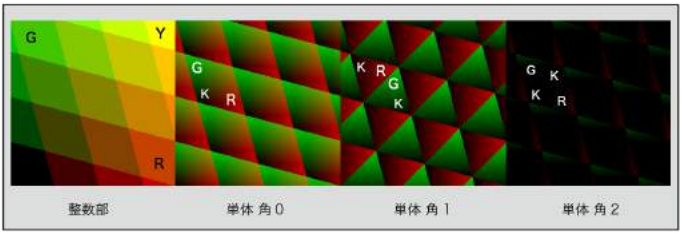


▲図 5.23　3 次元の場合の単体のユニットのうちどの単体に点 P が属するかを決定するルール

この上の図は、3 次元の単体のユニットが作る立方体を主対角線に沿って見たものであり、点 P の座標値の x、y、z 軸についてのそれぞれの大きさによってどの単体に属するのかのルールを示したものです。

4 次元の場合、視覚化は困難ですが、2 次元と 3 次元の規則と同様に考えることができます。空間を満たす 4 次元の超立方体の座標 (`x, y, z, w`)それぞれの軸についての大きさの組み合わせは、4!=24 通りとなり、超立方体内の 24 個の単体それぞれに固有のものとなり、点 P がどの単体に属するかを判定することができます。

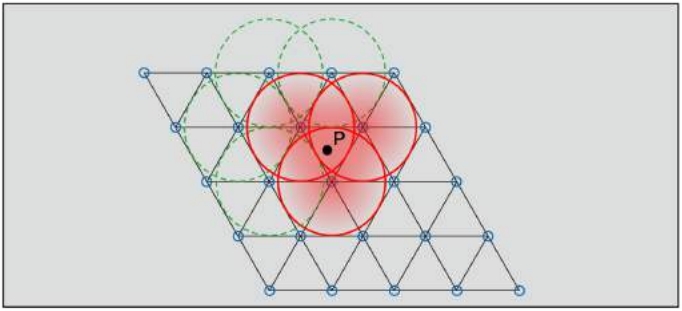
下の図は、2 次元における単体格子をフラグメントカラーで可視化したものです。



▲図 5.24　単体（2D）の整数部と少数部

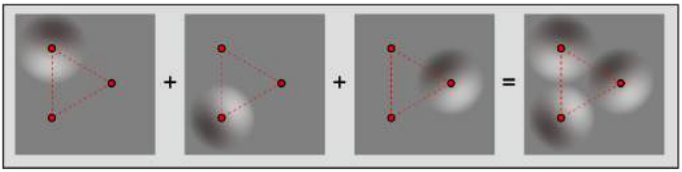
■補間から総和への移行　従来の **Perlin Noise** では、格子内部の点の値を周囲の格子点の値から補間によって求めていました。しかし、**Simplex Noise** では、代わりに、それぞれの単体形状の頂点の値の影響度合いを、単純な総和計算で求めます。具体的には、単体それぞれの角の勾配の外挿と、各頂点からの距離によって放射円状に減衰する関数の積の足しあわせを行います。

2 次元について考えます。



▲図 5.25　放射円状減衰関数とその影響範囲

単体の内部の点 P の値は、それを囲んでいる単体の 3 つの各頂点からの値のみ影響します。離れた位置にある頂点の値は、点 P を含んだ単体の境界を越える前に 0 に減衰するので、影響を及ぼしません。このように、点 P のノイズの値は、3 つの頂点の値とその影響度合いの合計として計算することができます。



▲図 5.26　各頂点の寄与率と総和

実装

実装については、2012 年に、Ian McEwan、David Sheets、 Stefan Gustavson 、Mark Richardson によって発表された **"Effecient computational noise in GLSL"**

https://pdfs.semanticscholar.org/8e58/ad9f2cc98d87d978f2bd85713d6c909c8a85.pdf

に従った方法で示します。

現状、シェーダによるノイズの実装を行いたい場合、ハードウェア依存が少なく、計算も効率的で、テクスチャを参照するなどの必要がなく扱いやすいアルゴリズムです。（おそらく）

ソースコードは 2018 年 4 月現在、https://github.com/stegu/webgl-noise/ で管理されています。オリジナルはこちら (https://github.com/ashima/webgl-noise) でしたが、現在これを管理していた Ashima Arts は会社として機能していないようなので、Stefan Gustavson によってクローンされました。

実装の特徴としては、以下の 3 つが挙げられます。

- 勾配ベクトル計算のためのランダムに並んだインデックスを、テーブルを参照するのではなく、多項式による計算で求める
- 正軸体（Cross Polytope）の幾何学形状を勾配ベクトル計算に用いる
- 単体選択の条件を順序付け（Rank Ordering）と置き換える

■勾配のインデックス並べ換えのための多項式　過去に発表されたノイズの実装では、勾配計算時のインデックス生成のために、事前に計算されたインデックスの値を格納したテーブルやビット入れ替えによるハッシュを使っていたりしましたが、どちらのアプローチもシェーダによる実装には向いているとは言えません。そこで、インデックス並べ替えのために、

$$\left(Ax^2 + Bx\right) mod\ M$$

というシンプルな形をした多項式を使用する方法を提案しています。

（mod=modulo ある数を割った時の余りの数（剰余））例えば、$\left(6x^2+x\right) mod\ 9$ は（0 1 2 3 4 5 6 7 8）を（0 7 8 3 1 2 6 4 5）というように、0~8 の入力に対して重複のない 0~8 の 9 つの数字を返します。

勾配を十分に良く分散するためのインデックス生成には、少なくとも数百の数字を並べ替える必要があり、$\left(34x^2+x\right) mod\ 289$ を選択することとします。

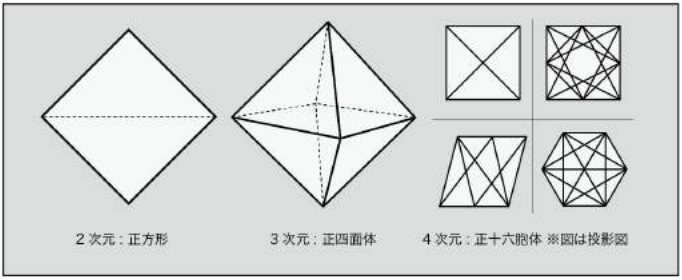
この置換多項式は、シェーダ言語の変数の精度の問題で、整数領域で、$34x^2+x > 2^{24}$, または、$|x| > 702$ の時に、切り捨てが発生してしまいます。そこで、オーバーフローのリスクなしに並べ替えのための多項式を計算するために、多項式の計算を行う前に、x の 289 の剰余計算を行って、x を 0~288 の範囲に制限します。

具体的には、以下のように実装されます。

```
// 289 の剰余を求める
float3 mod289(float3 x)
{
    return x - floor(x * (1.0 / 289.0)) * 289.0;
}

// 置換多項式による並べ替え
float3 permute(float3 x)
{
    return fmod(((x * 34.0) + 1.0) * x, 289.0);
}
```

論文では、2、3 次元のときは問題ないが、4 次元の場合は、この多項式では視覚的なアーティファクトが発生してしまっていることを認めています。4 次元の場合、289 のインデックスでは不十分であるようです。

■正軸体（Cross Polytope）の幾何学形状を勾配ベクトル計算に用いる　従来の実装では、勾配計算に疑似乱数を使用し、事前に計算した勾配のインデックスの計算のために、インデックスを格納したテーブルを参照したり、ビット操作を行ったりしました。ここでは、よりシェーダによる実装に最適で、さまざまな次元で効率よく分散した勾配を得るために、正軸体（**Cross Polytope)** を勾配計算に使用します。正軸体とは、2 次元の正方形、3 次元の正八面体、4 次元の正十六胞体を各次元に一般化した形状の事を指します。各次元下図のような幾何学的形状をとります。



▲図 5.27　各次元における正軸体

勾配ベクトルは、それぞれの次元で、2 次元であれば正方形、3 次元であれば正八面体、4 次元であれば（一部切り詰められた）正十六胞体の表面上に分散します。

各次元、方程式は以下の通りです。

```
2-D: x0 ∈ [− 2, 2], y = 1 − |x0|
if y > 0 then x = x0 else x = x0 − sign(x0)

3-D: x0, y0 ∈ [− 1, 1], z = 1 − |x0| − |y0|
if z > 0 then x = x0, y = y0
else x = x0 − sign(x0), y = y0 − sign(y0)

4-D: x0, y0, z0 ∈ [− 1, 1], w = 1.5 − |x0| − |y0| − |z0|
if w > 0 then x = x0, y = y0, z = z0
else x = x0 − sign(x0), y = y0 − sign(y0), z = z0 − sign(z0)
```

■勾配の正規化　ほとんどの **Perlin Noise** の実装では大きさが等しい勾配ベクトルを使っていました。しかし、N 次元の正軸体の表面のベクトルの最短のものと最長のものでは、$\sqrt{N}$ の因数分、長さに差があります。これは強いアーティファクトを引き起こしませんが、次元が高くなると、このベクトルを明示的に正規化しなければノイズパターンの等方性が低くなってしまいます。正規化とは、ベクトルをそのベクトルの大きさで割ることにより、大きさを 1 にそろえる処理です。勾配ベクトルの大きさを r とすると、勾配ベクトルに r の逆平方根 $\frac{1}{\sqrt{r}}$ を掛け合わせることで、正規化が実現できます。ここでは、パフォーマンス向上のため、この逆平方根をテイラー展開を用いて近似的に計算しています。テイラー展開とは、無限に微分可能な関数において、x が a の近辺であるなら、以下の式で近似的に計算できる、というものです。

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$\frac{1}{\sqrt{a}}$ の 1 階微分を求めると、

$$f(a) = \frac{1}{\sqrt{a}} = a^{-\frac{1}{2}}$$
$$f'(a) = -\frac{1}{2}a^{-\frac{3}{2}}$$

となり、テイラー展開による a 近辺での近似式は以下のようになります。

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$$= a^{-\frac{1}{2}} - \tfrac{1}{2}a^{-\frac{3}{2}}(x-a)$$
$$= \tfrac{3}{2}a^{-\frac{1}{2}} - \tfrac{1}{2}a^{-\frac{3}{2}}x$$

ここで、a=0.7(勾配ベクトルの長さの範囲が 0.5〜1.0 であるからなんだろうと思います) とすると、 $1.79284291400159 - 0.85373472095314 * x$ が得られます。
実装ではこのようになっています。

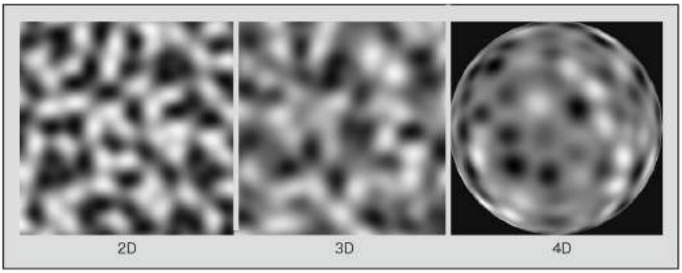
```
float3 taylorInvSqrt(float3 r)
{
    return 1.79284291400159 - 0.85373472095314 * r;
}
```

5.3.5 実装と結果

サンプルプロジェクト内の
TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**
シーンを開くと、**Simplex Noise** の実装結果を見ることができます。実装したコードは、

- Shaders/ProceduralNoise/**SimplexNoise2D.cginc**
- Shaders/ProceduralNoise/**SimplexNoise3D.cginc**
- Shaders/ProceduralNoise/**SimplexNoise4D.cginc**

にあります。



▲図 5.28　Simplex Noise（2D, 3D, 4D）結果

Simplex Noise は、**Perlin Noise** と比較すると、少し粒感のある結果が得られます。

5.4　まとめ

プロシージャルノイズの代表的手法のアルゴリズムと実装について詳細に見てきましたが、それぞれ得られるノイズパターンの特徴や、計算コストに違いがあることが確認できたと思います。リアルタイムアプリケーションにおいてノイズを用いる場合、それが高解像度となるときは、画素一つ一つに対して計算を行うため、この計算負荷については無視することはできず、どのような計算がなされているかは、ある程度留意しておく必要があります。最近では、ノイズ関数がはじめから開発環境に組み込まれているものも多いですが、それを十分に使いこなすためにも、ノイズのアルゴリズムを理解しておくことは重要です。ここではその応用については解説できませんでしたが、グラフィックス生成において、ノイズの応用は極めて多岐にわたり多大な効果をもたらします。（次章ではその例の一つを示します。）この記事が、数え切れないほどの応用への足がかりとなれば幸いです。最後に、先人たちが積み上げてきた知恵と、主に Ken Perlin の素晴らしい業績について敬意を表したいと思います。

5.5　参照

- [1] An Image Synthesizer, Ken Perlin, SIGGRAPH 1985
- [2] Improving Noise, Ken Perlin —http://mrl.nyu.edu/~perlin/paper445.pdf
- [3] Noise hardware. In Real-Time Shading SIGGRAPH Course Notes, Ken Perlin, 2001 — https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf
- [4] Improved Noise reference implementation, Ken Perlin, SIGGRAPH 2002http://mrl.nyu.edu/~perlin/noise/

- [5] GPU Gems Chapter 5. Implementing Improved Perlin Noise, Ken Perlin —http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch05.html
- [6] Simplex noise demystified. Technical Report, Stefan Gustavson, 2005 —http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
- [7] Efficient computational noise in GLSL, Ian McEwan, David Sheets, Stefan Gustavson and Mark Richardson, 6 Apr 2012 —http://webstaff.itn.liu.se/~stegu/jgt2012/article.pdf
- [8] Direct computational noise in GLSL Supplementary material, Ian McEwan, David Sheets, Stefan Gustavson and Mark Richardson, 2012 — http://weber.itn.liu.se/~stegu/jgt2011/supplement.pdf
- [9] Texturing and Modeling; A Procedural Approach, Second Edition —
- [10] The Book of Shaers Noise, Patricio Gonzalez Vivo & Jen Lowe — https://thebookofshaders.com/11/
- [11] Building Up Perlin Noise ー http://eastfarthing.com/blog/2015-04-21-noise/
- [12] Z で行こう！ Extension for 3ds Max ２０１５を調べてみた その 24 3ds-max 2015 ー http://blog.livedoor.jp/takezultima/archives/2015-05.html

第 6 章

Curl Noise - 疑似流体のためのノイズアルゴリズムの解説

6.1 はじめに

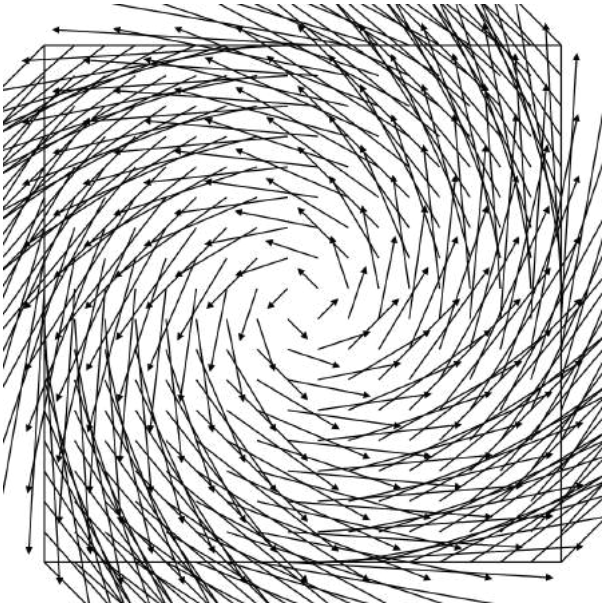
本章では、擬似流体アルゴリズムである Curl Noise（カールノイズ）の GPU 実装についての解説を行なっていきます。
本章のサンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming2`
の「CurlNoise」です。

6.1.1 Curl Noise とは

Curl Noise とは、FLIP 法等の流体アルゴリズムの開発者としても知られているブリティッシュ・コロンビア大学の Robert Bridson 教授が、2007 年に発表した擬似流体ノイズアルゴリズムです。
前作の「Unity Graphics Programming vol.1」にて、ナビエ・ストークス方程式を使った流体シミュレーションの解説をさせて頂きましたが、Curl Noise はそれらの流体シミュレーションと比べて擬似的ではありますが軽負荷にて流体表現をすることが可能です。
特に昨今、ディスプレイやプロジェクターの技術躍進に伴い 4K や 8K 等の高解像においてリアルタイムレンダリングを行う必要性が高まっていますので、Curl Noise の様な低負荷なアルゴリズムは、流体表現を高解像度や低マシンスペックで表現する為の有用な選択肢となります。

6.2 Curl Noise のアルゴリズム

流体シミュレーションにおいて、まず必要となるものが「速度場」と呼ばれるベクトル場です。まずは速度場というものがどういう物かを図でイメージしてみましょう。
以下が 2 次元の場合の速度場のイメージ図です。平面上の各点において、ベクトルが定義されているのが見て取れるかと思います。



▲図 6.1　二次元上での速度場の観測

上図の様に、平面上の各微分区間において、それぞれベクトルが個別に定義されている状態をベクトル場といい、それぞれのベクトルが速度である物を速度場と言います。
これらは 3 次元上であっても、立方体の中の各微分ブロックにおいてベクトルが定義されている状態と想像していただけると、解りやすいかと思います。
それでは、Curl Noise はどの様にして、この速度場を導出しているのかを見ていきます。

Curl Noise の面白いところは、前章の「プロシージャルノイズ入門」でも解説された Perlin Noise や Simplex Noise 等の勾配ノイズをポテンシャル場として用い、そこから流体の速度場を導出するところにあります。
本章ではポテンシャル場として、3 次元 Simplex Noise を利用する事とします。

以下、まずは Curl Noise の数式からアルゴリズムを紐解いていきたいと思います。

$$\overrightarrow{u} = \nabla \times \psi$$

上記は Curl Noise のアルゴリズムになります。
左辺 $\overrightarrow{u}$ は導出された速度ベクトル、右辺 ∇ はベクトル微分演算子（ナブラと読み、偏微分の作用素として働きます）、 ψ はポテンシャル場です。（本章では 3 次元 Simplex Noise）
Curl Noise はこの右辺二項の外積を取ったものとして表せます。

つまり、Curl Noise は Simplex Noise とベクトル各要素の偏微分 $\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ の外積をとったものであり、過去にベクトル解析を学ばれた事がある方にとっては、rotA（回転）の形そのものである事が見て取れるかと思います。
それでは、3D Simplex Noise と偏微分の外積を計算してみましょう

$$\overrightarrow{u} = \left(\frac{\partial \psi_3}{\partial y} - \frac{\partial \psi_2}{\partial z}, \frac{\partial \psi_1}{\partial z} - \frac{\partial \psi_3}{\partial x}, \frac{\partial \psi_2}{\partial x} - \frac{\partial \psi_1}{\partial y}\right)$$

一般的に外積は、二つのベクトルのお互いにとっての垂直方向を向き、その長さは両ベクトルで張られる面の面積と同じになるという特徴を持ったものですが、ベクトル解析での rotA（回転）での外積演算のイメージの掴み方は、「ねじれた各偏微分要素方向のポテンシャル場のベクトルをルックアップし、その項同士を引いているから回転が起きる」と、上記の計算式からシンプルに捉えた方がイメージを掴みやすいかもしれません。
実装自体はとても単純で、偏微分の各要素方向に微小にルックアップポイントをずらしながら、上記 ψ の各点、つまり 3 次元 SimplexNoise からベクトルをルックアップし、上記の数式の様に、外積演算をするだけです。

6.2.1 質量保存則について

もし前作の「Unity Graphics Programming vol.1」の流体シミュレーションの章をお読みになられた方は、質量保存則はどうなっているのかと気になられた方もいらっしゃるかもしれません。
質量保存則とは、流体は速度場での各ポイントにおいて、必ず、流入と流出の均衡が

とれ、流入した分流出し、流出した分は流入してきており、最終的には発散ゼロ（ダイバージェンスフリー）というルールでした。

$$\nabla \cdot \vec{u} = 0$$

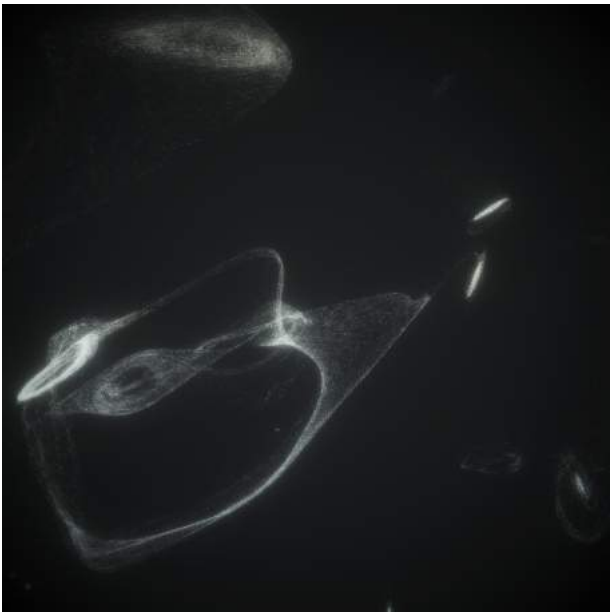
この事については、論文でも述べられているのですが、そもそも勾配ノイズ自体がなだらかに変化している物ですので（2 次元グラデーションでイメージするなら、左側のピクセルが薄ければ、右側は濃くなっている様に）、ダイバージェンスフリーはポテンシャル場の時点で保証されているという物でした。パーリンノイズの特徴から考えてみれば至極当然な事ですね。

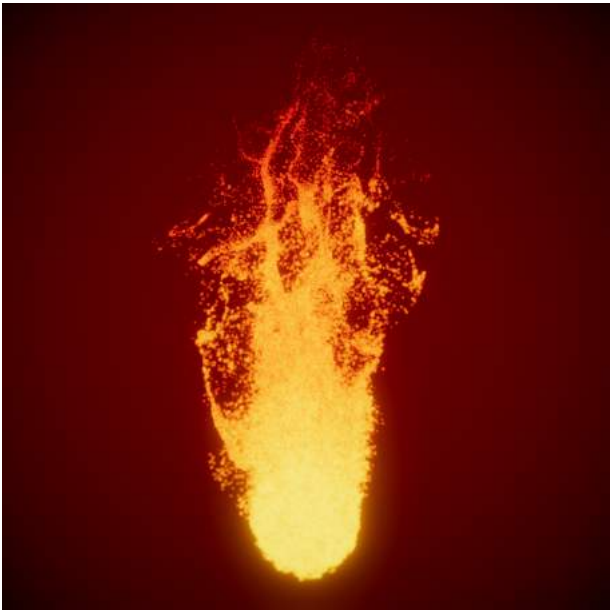
6.2.2 Curl Noise の実装

それでは、数式に基づいて、Compute shader、若しくは Shader で CurlNoise 関数を GPU 実装してみましょう。

```
#define EPSILON 1e-3

float3 CurlNoise(float3 coord)
{
    float3 dx = float3(EPSILON, 0.0, 0.0);
    float3 dy = float3(0.0, EPSILON, 0.0);
    float3 dz = float3(0.0, 0.0, EPSILON);

    float3 dpdx0 = snoise(coord - dx);
    float3 dpdx1 = snoise(coord + dx);
    float3 dpdy0 = snoise(coord - dy);
    float3 dpdy1 = snoise(coord + dy);
    float3 dpdz0 = snoise(coord - dz);
    float3 dpdz1 = snoise(coord + dz);

    float x = dpdy1.z - dpdy0.z + dpdz1.y - dpdz0.y;
    float y = dpdz1.x - dpdz0.x + dpdx1.z - dpdx0.z;
    float z = dpdx1.y - dpdx0.y + dpdy1.x - dpdy0.x;

    return float3(x, y, z) / EPSILON * 2.0;
}
```

上記の様に、このアルゴリズムはシンプルな四則演算の形に落とし込む事ができますので、実装自体はとても簡単で、これだけの行で実装ができてしまいます。
以下に今回コンピュートシェーダにて実装した Curl Noise のサンプルを貼っておきます。パーティクルの粒を移流たり、上昇ベクトルを加えて炎の様に見せたり、アイデア次第で様々な表情を引き出す事が可能です。





6.3 まとめ

本章では Curl Noise による擬似流体の実装について解説しました。
少ない負荷と実装で 3 次元の擬似流体を再現可能な為、高解像度でのリアルタイムレンダリングにおいては特に有用に働くアルゴリズムではないでしょうか。

まとめとして、Curl Noise アルゴリズムをはじめ、様々な技法を今もあみ出されている Robert Bridson 教授への最大限の謝辞を込めて、この章のまとめとさせていただきたいと思います。
説明が至らない点、わかりづらい部分もあったかと思いますが、ぜひ読者の方々にもグラフィックスプログラミングを楽しんで頂けますと幸いです。

6.4 References

- Robert Bridson, Jim Hourihan, Marcus Nordenstam. 2007, Curl-noise for procedural fluid flow. In proc, ACM SIGGRAPH 46.

第 7 章

Shape Matching - 線形代数の CG への応用

7.1 はじめに

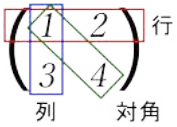
本章では, 線形代数学の基礎・応用から, 特異値分解の学習, 特異値分解を用いたアプリケーションの紹介を行います. 高校生で行列, 大学生で線形代数を学んだものの, 「CG の世界でどのように生かされているのかがわからない」という方も多いのではないかと思い, 今回の執筆にあたりました. 本章では, 理解しやすさを優先するために, **2** 次元かつ実数の範囲内での解説を行います. そのため, 実際の線形代数の定義とは少し異なる部分がありますが, 適宜読み替えていただけると幸いです.

7.2 行列とは？

ほとんどの読者の皆様は, 1 度は行列という言葉を耳にしたことがあるかと思います (現在, 高校では行列を習わないのだとか...). 行列とは, 以下のように数字を縦と横に並べたものを指します.

$$\boldsymbol{M} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{7.1}$$

横方向を行, 縦方向を列, 斜め方向を対角, それぞれの数字を行列の要素と呼びます.



▲図 7.1　行列

ちなみに, 行列は, 英語で Matrix(マトリックス) と呼ばれます.

7.3　行列演算のおさらい

行列演算の基本について軽くおさらいしていきましょう. 既にご存じの方は, この節を読み飛ばしていただいて構いません.

7.3.1　加減乗除

行列には, スカラーの四則演算と同様, 加減乗除算が存在します. 簡単のために, 2 次正方行列[*1] $\boldsymbol{A}$ と $\boldsymbol{B}$, 2 次元ベクトル $\boldsymbol{c}$ を以下で定義しておきます.

$$\boldsymbol{A} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \boldsymbol{B} = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \tag{7.2}$$

加算

行列の加算は, 式 (7.3) のように, 要素ごとに和を計算します.

$$\boldsymbol{A} + \boldsymbol{B} = \begin{pmatrix} a_{00} + b_{00} & a_{01} + b_{01} \\ a_{10} + b_{10} & a_{11} + b_{11} \end{pmatrix} \tag{7.3}$$

減算

行列の減算は, 式 (7.4) のように, 要素ごとに差を計算します.

$$\boldsymbol{A} - \boldsymbol{B} = \begin{pmatrix} a_{00} - b_{00} & a_{01} - b_{01} \\ a_{10} - b_{10} & a_{11} - b_{11} \end{pmatrix} \tag{7.4}$$

乗算

行列の乗算は少し複雑で, 式 (7.5) のようになります.

[*1] 正方行列 ··· 行と列の数が等しい行列のこと.

$$\boldsymbol{AB} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix} \tag{7.5}$$

掛ける順序を逆にすると, 計算結果も変化するので, 注意してください.

$$\boldsymbol{BA} = \begin{pmatrix} b_{00}a_{00} + b_{01}a_{10} & b_{00}a_{01} + b_{01}a_{11} \\ b_{10}a_{00} + b_{11}a_{10} & b_{10}a_{01} + b_{11}a_{11} \end{pmatrix} \tag{7.6}$$

除算 (逆行列)

行列の除算は, スカラーにおける除算と少し異なる, 逆行列と呼ばれる概念を使用します. まずはじめに, スカラーでは, 自分自身の逆数を掛けると, 必ず 1 になる性質を持っています.

$$4 \times \frac{1}{4} = 1 \tag{7.7}$$

つまり, 除算という行為は, 「逆数を掛ける」という演算と等しくなります.

$$7 \times \frac{1}{4} = \frac{7}{4} \tag{7.8}$$

これを行列に置き換えると, 行列にかけて単位行列を生み出すものが, 除算を表す行列であるということができます. 行列において, スカラーの 1 に対応するものは単位行列と呼ばれ, 以下で定義されます.

$$\boldsymbol{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{7.9}$$

スカラーの 1 と同様, 単位行列をどのような行列にかけても, 値が変化しません.

これらを踏まえて, 行列の除算について考えます. 逆行列を $\boldsymbol{M}^{-1}$ とすると, 逆行列の定義は以下となります.

$$\boldsymbol{M}\boldsymbol{M}^{-1} = \boldsymbol{M}^{-1}\boldsymbol{M} = \boldsymbol{I} \tag{7.10}$$

導出は省略しますが, 行列 $\boldsymbol{A}$ の逆行列の要素は, 以下で定義されます.

$$\boldsymbol{A}^{-1} = \frac{1}{a_{00}a_{11} - a_{01}a_{10}} \begin{pmatrix} a_{11} & -a_{01} \\ -a_{10} & a_{00} \end{pmatrix} \tag{7.11}$$

この時の, $a_{00}a_{11} - a_{01}a_{10}$ を行列式 (Determinant) と呼び, $det(\boldsymbol{A})$ と表します.

7.3.2 行列のベクトルへの作用

行列は, ベクトルと乗算することによってベクトルの指す座標を変換することができます. ご存知の通り, CG では, 座標変換行列 (ワールド, プロジェクション, ビュー変換行列) などとして用いられることがほとんどです. 行列とベクトルの積は, 以下で定義されます.

$$\boldsymbol{Ac} = \begin{pmatrix} a_{00}c_0 + a_{01}c_1 \\ a_{10}c_0 + a_{11}c_1 \end{pmatrix} \tag{7.12}$$

7.4 行列演算の応用

本節からは, 大学で習う範囲の行列の概念を解説していきます. 少々難しいと感じられる部分が多いかとは思いますが, Shape Matching を理解するには, これだけの概念が必要ですので, 頑張って理解していきましょう. と言いつつ, 本節の話は行列演算ライブラリ内部で吸収される部分でもありますので, 第 7.7 節まで読み飛ばしていただいても実装に支障はありません.

7.4.1 転置行列

転置行列は, 要素の行と列を入れ替えたもので, 以下で定義されます.

$$\boldsymbol{A}^T = \begin{pmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \end{pmatrix} \tag{7.13}$$

7.4.2 対称行列

$\boldsymbol{A}^T = \boldsymbol{A}$ を満たす行列を, 対称行列と呼びます.

7.4.3 固有値と固有ベクトル

正方行列 $\boldsymbol{A}$ が与えられたとき,

$$\boldsymbol{Av} = \lambda \boldsymbol{v}, (\boldsymbol{v} \neq 0) \tag{7.14}$$

を満たす λ を, $\boldsymbol{A}$ の固有値, $\boldsymbol{v}$ を固有ベクトルと呼びます.
固有値と固有ベクトルの算出方法を以下に示します. まず, 式 (7.14) を変形して,

$$(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{v} = 0 \tag{7.15}$$

とします. ここで, $\boldsymbol{v} \neq 0$ という条件を用いると, 式 (7.15) は,

$$det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0 \tag{7.16}$$

となります. この式を展開すると, λ についての 2 次方程式になるため, これを解くことで λ を算出することができます. また, 算出されたそれぞれの λ を式 (7.15) に代入することで, 固有ベクトル $\boldsymbol{v}$ を算出することができます.

数式だけでは, 固有値・固有ベクトルの概念はわかりにくいので, @kenmatsu4 氏による固有値の可視化が行われた Qiita 記事 (章末に記載) を併せてご覧いただけるとよいかと思います.

7.4.4 固有値分解

正方行列 $\boldsymbol{A}$ における固有値と固有ベクトルを用いて, 行列 $\boldsymbol{A}$ を違う形で表すことができます. まず, 固有値 λ を大きさ順に並べ替え, それを対角要素に持つ行列 $\boldsymbol{\Lambda}$ を作成します. 次に, それぞれの固有値に対応する固有ベクトルを左から順に並べた行列 $\boldsymbol{V}$ を作成します. すると, 式 (7.14) を, これらの行列を用いて以下のように書き換えることができます.

$$\boldsymbol{AV} = \boldsymbol{V\Lambda} \tag{7.17}$$

さらに, これを左辺に行列 $\boldsymbol{A}$ が残るよう, 両辺右から $\boldsymbol{V}^{-1}$ を掛けると,

$$\boldsymbol{A} = \boldsymbol{V\Lambda V}^{-1} \tag{7.18}$$

となります.

このように, 行列を式 (7.18) のような形式に分解することを, 行列の固有値分解と呼びます.

7.4.5 正規直交基底

互いに垂直であり, かつそれぞれが単位ベクトルであるベクトルの組を, 正規直交基底と呼びます. 任意のベクトルは, 正規直交基底の組を用いて表すことができます[*2]. 2 次元の場合, 正規直交基底となりうるベクトルは 2 つとなります. 例えば, よく用いられる x 軸と y 軸は, $\boldsymbol{x} = (1, 0), \boldsymbol{y} = (0, 1)$ の組で正規直交基底を成しているので, 任意のベクトルをこの $\boldsymbol{x}, \boldsymbol{y}$ で表すことができます. $\boldsymbol{v} = (4, 13)$ を正規直交基底 $\boldsymbol{x}, \boldsymbol{y}$ を用いて表すと, $\boldsymbol{v} = 4\boldsymbol{x} + 13\boldsymbol{y}$ となります.

[*2] 正式にはベクトルの一次結合と呼びます.

7.4.6 エルミート行列

エルミート行列が定義されるのは, 本来複素数の範囲であり本章の領域を超えますので, 実数の範囲で簡単に説明します. 実数の範囲では, 行列 $\boldsymbol{A}$ のエルミート行列 $\boldsymbol{A}^*$ は, 単に対称行列であることを意味しており,

$$\boldsymbol{A}^* = (\overline{\boldsymbol{A}})^T = \boldsymbol{A}^T \tag{7.19}$$

となります.

7.4.7 直交行列

正方行列 $\boldsymbol{Q}$ を列ベクトル $Q = (\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_n)$ に分解したとき, これらのベクトルの組が正規直交系をなしている, つまり,

$$\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{Q}^{-1} = \boldsymbol{Q}^T \tag{7.20}$$

が成り立つとき, $\boldsymbol{Q}$ は直交行列であるといいます. また, 直交行列を行ベクトルに分解したとしても, 正規直交系をなしている特徴があります.

7.4.8 ユニタリ行列

$$\boldsymbol{U}^*\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^* = \boldsymbol{I} \tag{7.21}$$

を満たす行列を, ユニタリ行列と呼びます. ユニタリ行列 $\boldsymbol{U}$ の要素がすべて実数 (実行列) の場合, $\boldsymbol{U}^* = \boldsymbol{U}^T$ となるため, 実ユニタリ行列 $\boldsymbol{U}$ は直交行列であることがわかります.

7.5 特異値分解

任意の $m \times n$ 行列 $\boldsymbol{A}$ を, 以下の形に分解することを, 行列の特異値分解と呼びます.

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{7.22}$$

尚, $\boldsymbol{U}$ と $\boldsymbol{V}^T$ は, $m \times m$ の直交行列, $\boldsymbol{\Sigma}$ は, $m \times n$ の対角行列 (対角成分は非負で大きさの順に並んだ行列) となっています.

「任意の」という言葉が肝で, 行列の固有値分解は正方行列に対してのみ定義されますが, 特異値分解は長方行列でも行うことができます. CG の世界では, 扱う行列が正

方行列であることがほとんどですので, 固有値分解とさほど計算方法は変わりません. また, $\boldsymbol{A}$ が対称行列のとき, $\boldsymbol{A}$ の固有値と特異値は一致します. さらに, $\boldsymbol{A}^T\boldsymbol{A}$ の 0 でない固有値の正の平方根は $\boldsymbol{A}$ の特異値です.

7.5.1 特異値分解のアルゴリズム

固有値分解をプログラムに落とし込むには, 式 (7.22) を式変形するとわかりやすいです. 行列 $\boldsymbol{A}$ の左からその行列の転置 $\boldsymbol{A}^T$ を掛けると, 以下のようになります.

$$\begin{aligned}\boldsymbol{A}^T\boldsymbol{A} &= (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T && (7.23)\\ &= (\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T)\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T && (7.24)\\ &= \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\boldsymbol{V}^T && (7.25)\\ &= \boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^T && (7.26)\end{aligned}$$

形が固有値分解と同じになっているのに気づくかと思います. 実際, 特異値行列の自乗は固有値行列になることが知られています. よって, 特異値の算出は, 行列を固有値分解し, 固有値の 2 乗根を取ることで行うことが可能です. このことから, 固有値分解をアルゴリズム内に組み込むことになりますが, 固有値を求めるために 2 次方程式を解く必要が出てきます. 幸い, 2 次方程式は解の公式が単純ですので, プログラムに落とすのも簡単です[*3].

$\boldsymbol{A}^T\boldsymbol{A}$ 固有値分解することで, $\boldsymbol{\Sigma}$ と $\boldsymbol{V}^T$ が算出できましたので, 残りの $\boldsymbol{U}$ は式 (7.22) を変形して以下で算出できます.

$$\begin{aligned}\boldsymbol{U} &= \boldsymbol{A}\boldsymbol{\Sigma}^{-1}(\boldsymbol{V}^T)^{-1} && (7.27)\\ &= \boldsymbol{A}\boldsymbol{\Sigma}^{-1}\boldsymbol{V} && (7.28)\\ & && (7.29)\end{aligned}$$

尚, V は直交行列なので, 転置と逆行列が一致します.
これをプログラムで表すと, 以下のようになります.

▼リスト 7.1　特異値分解のアルゴリズム (Matrix2x2.cs)

```
1: /// <summary>
2: /// 特異値分解
3: /// </summary>
4: /// <param name="u">Returns rotation matrix u</param>
5: /// <param name="s">Returns sigma matrix</param>
```

[*3] 3 次方程式や 4 次方程式にも解の公式は存在しますが, 一般的にニュートン法等を用いて計算を行います.

```
 6: /// <param name="v">Returns rotation matrix v(not transposed)</param>
 7: public void SVD(ref Matrix2x2 u, ref Matrix2x2 s, ref Matrix2x2 v)
 8: {
 9:     // 対角行列であった場合、特異値分解は単純に以下で与えられる。
10:     if (Mathf.Abs(this[1, 0] - this[0, 1]) < MATRIX_EPSILON
11:         && Mathf.Abs(this[1, 0]) < MATRIX_EPSILON)
12:     {
13:         u.SetValue(this[0, 0] < 0 ? -1 : 1, 0,
14:                    0, this[1, 1] < 0 ? -1 : 1);
15:         s.SetValue(Mathf.Abs(this[0, 0]), Mathf.Abs(this[1, 1]));
16:         v.LoadIdentity();
17:     }
18:
19:     // 対角行列でない場合、A^T*A を計算する。
20:     else
21:     {
22:         // 0 列ベクトルの長さ (非ルート)
23:         float i   = this[0, 0] * this[0, 0] + this[1, 0] * this[1, 0];
24:         // 1 列ベクトルの長さ (非ルート)
25:         float j   = this[0, 1] * this[0, 1] + this[1, 1] * this[1, 1];
26:         // 列ベクトルの内積
27:         float i_dot_j = this[0, 0] * this[0, 1]
28:                           + this[1, 0] * this[1, 1];
29:
30:         // A^T*A が直交行列であった場合
31:         if (Mathf.Abs(i_dot_j) < MATRIX_EPSILON)
32:         {
33:             // 特異値行列の対角要素の計算
34:             float s1 = Mathf.Sqrt(i);
35:             float s2 = Mathf.Abs(i - j) <
36:                        MATRIX_EPSILON ? s1 : Mathf.Sqrt(j);
37:
38:             u.SetValue(this[0, 0] / s1, this[0, 1] / s2,
39:                        this[1, 0] / s1, this[1, 1] / s2);
40:             s.SetValue(s1, s2);
41:             v.LoadIdentity();
42:         }
43:         // A^T*A が直交行列でない場合、固有値を求めるために二次方程式を解く。
44:         else
45:         {
46:             // 固有値/固有ベクトルの算出
47:             float i_minus_j = i - j;    // 列ベクトルの長さの差
48:             float i_plus_j = i + j;     // 列ベクトルの長さの和
49:
50:             // 2 次方程式の解の公式
51:             float root = Mathf.Sqrt(i_minus_j * i_minus_j
52:                                     + 4 * i_dot_j * i_dot_j);
53:             float eig = (i_plus_j + root) * 0.5f;
54:             float s1 = Mathf.Sqrt(eig);
55:             float s2 = Mathf.Abs(root) <
56:                        MATRIX_EPSILON ? s1 :
57:                          Mathf.Sqrt((i_plus_j - root) / 2);
58:
59:             s.SetValue(s1, s2);
60:
61:             // A^T*A の固有ベクトルを V として用いる。
62:             float v_s = eig - i;
63:             float len = Mathf.Sqrt(v_s * v_s + i_dot_j * i_dot_j);
```

```
64:             i_dot_j /= len;
65:             v_s /= len;
66:             v.SetValue(i_dot_j, -v_s, v_s, i_dot_j);
67:
68:             // v と s が算出済みなので、回転行列 u を Av/s で算出
69:             u.SetValue(
70:                 (this[0, 0] * i_dot_j + this[0, 1] * v_s) / s1,
71:                 (this[0, 1] * i_dot_j - this[0, 0] * v_s) / s2,
72:                 (this[1, 0] * i_dot_j + this[1, 1] * v_s) / s1,
73:                 (this[1, 1] * i_dot_j - this[1, 0] * v_s) / s2
74:             );
75:         }
76:     }
77: }
```

7.6 特異値分解を用いるアルゴリズム

特異値分解は, 多種多様な分野で活躍しており, 主に統計学における主成分分析 (PCA) で用いられることが多いようです. CG で利用される例も少なくはなく,

- Shape Matching[*4]
- Anisotropic Kernel[*5]
- Material Point Method[*6]

などが挙げられます.

今回は, Shape Matching にフォーカスを当て, 基礎的な考え方について解説していきます.

7.7 Shape Matching

7.7.1 概要

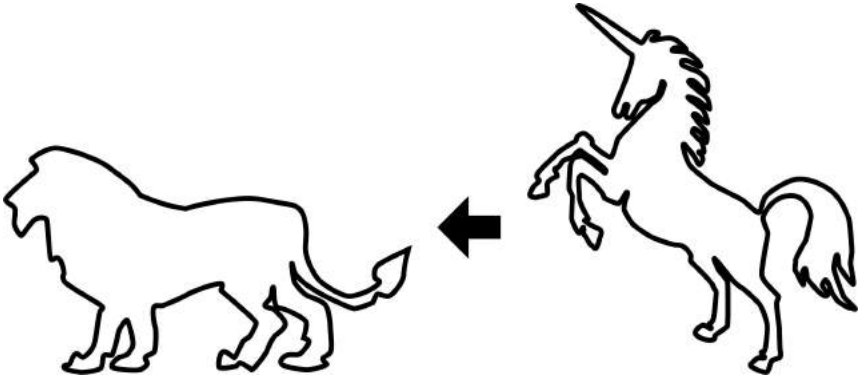
Shape Matching とは, 異なる 2 つの形状を, 極力誤差のない範囲で整列させる技術を指します. 現在では, Shape Matching を用いて, 弾性体を疑似的にシミュレートするような手法も開発されています.

本節では, 図 7.2, 図 7.3 のように, ユニコーンのオブジェクトの配置を, ライオンのオブジェクトの配置へ整列するアルゴリズムを解説します.

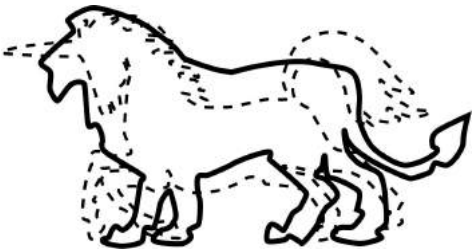
[*4] Meshless deformations based on shape matching, Matthias Muller et al., SIGGRAPH 2005

[*5] Reconstructing surfaces of particle-based fluids using anisotropic kernels, Jihun Yu et al., ACM Transaction on Graphics 2013

[*6] A material point method for snow simulation, Alexey Stomakhin et al., SIGGRAPH 2013



▲図 7.2　2 つのオブジェクト



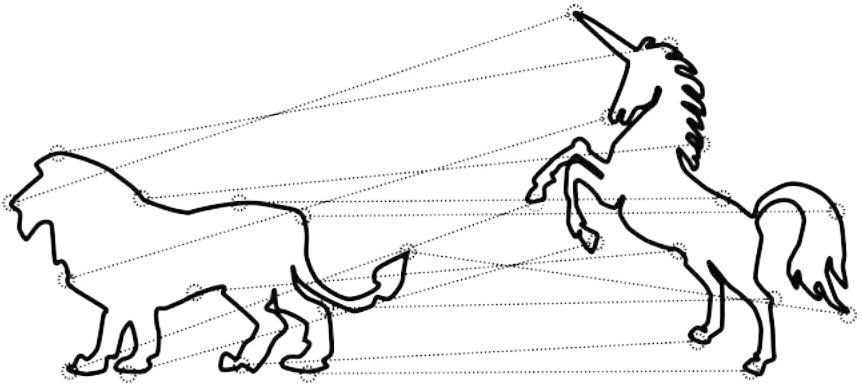
▲図 7.3　整列した結果

7.7.2　アルゴリズム

初めに, それぞれの形状の上に同数の点の集合を定義します. (ライオンの点集合を P, ユニコーンの点集合を Q とします.)

$$P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots, \boldsymbol{p}_n\}, Q = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_n\} \tag{7.30}$$

このとき, 下添え字が同一のものは, 図 7.4 のように, 幾何学的に対応した位置にあることに注意してください.



▲図 7.4　点集合の対応

次に, それぞれの点集合の重心を計算します.

$$\boldsymbol{p} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{p}_i \tag{7.31}$$

$$\boldsymbol{q} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{q}_i \tag{7.32}$$

ユニコーンの点集合の重心が, ライオンの点集合の重心と同じ位置に来るとすると, ユニコーンの点集合に回転行列 $\boldsymbol{R}$ を作用させ, ベクトル $\boldsymbol{t}$ 平行移動させた結果が, ライオンの重心と等しくなることから, 以下の式が導出できます.

$$\boldsymbol{p} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{R}\boldsymbol{q}_i + \boldsymbol{t} \tag{7.33}$$

これを変形すると,

$$\boldsymbol{p} = \boldsymbol{R}(\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{q}_i) + \boldsymbol{t} \tag{7.34}$$

$$= \boldsymbol{R}\boldsymbol{q} + \boldsymbol{t} \tag{7.35}$$

となり, さらに変形して,

$$\boldsymbol{t} = \boldsymbol{p} - \boldsymbol{R}\boldsymbol{q} \tag{7.36}$$

となります.

よって, この式から, 回転行列 $\boldsymbol{R}$ が求まれば, 自動的に平行移動ベクトル $\boldsymbol{t}$ が求まることがわかります. ここで, もともとの点の位置から, それぞれの重心を引いた点集合を定義します.

$$\boldsymbol{p}'_i = \boldsymbol{p}_i - \boldsymbol{p}, \boldsymbol{q}'_i = \boldsymbol{q}_i - \boldsymbol{q} \tag{7.37}$$

これにより, それぞれの点集合の重心を原点としたローカル座標で計算を行うことができるようになります.

次に, $\boldsymbol{p}'_i, \boldsymbol{q}'_i$ の分散共分散行列 $\boldsymbol{H}$ を計算します.

$$\boldsymbol{H} = \sum_{i=1}^{n} \boldsymbol{q}'_i \boldsymbol{p}'^{T}_i \tag{7.38}$$

この分散共分散行列 $\boldsymbol{H}$ には, 2 つの点集合のばらつき具合などの情報が格納されます. ここでのベクトルの積 $\boldsymbol{q}'_i \boldsymbol{p}'^{T}_i$ は, 通常のベクトルの内積演算とは異なり, 直積 (outer product) と呼ばれる演算となります. ベクトル同士の直積をとると, 行列が生成されます. 2 次元ベクトルでの直積は, 以下で定義されています.

$$\boldsymbol{a}\boldsymbol{b}^T = \begin{pmatrix} a_0 b_0 & a_0 b_1 \\ a_1 b_0 & a_1 b_1 \end{pmatrix} \tag{7.39}$$

さらに, 分散共分散行列 $\boldsymbol{H}$ を特異値分解します.

$$\boldsymbol{H} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{7.40}$$

特異値分解をした結果の中で, $\sum$ は伸縮を表す行列であるため, 求める回転行列 $\boldsymbol{R}$ は,

$$\boldsymbol{R} = \boldsymbol{V}\boldsymbol{U}^T \tag{7.41}$$

となります. (詳細な導出方法はやや高度となりますので, ここでは省略させていただきます.)

最後に, 求めた回転行列と式 (7.36) から, 平行移動ベクトル $\boldsymbol{t}$ を算出することができます.

7.7.3 実装

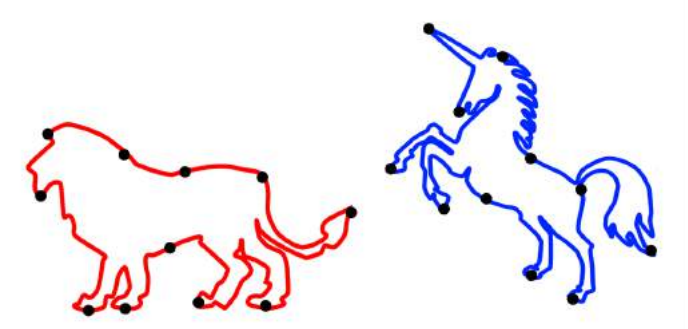
今回の実装は, 前節でのアルゴリズムをほぼそのままコードに落とすだけなので, 詳しい説明は省略させていただきます. 尚, ShapeMaching.cs 内の Start 関数内ですべての処理を完結させています.

▼リスト 7.2　ShapeMatching(ShapeMaching.cs)

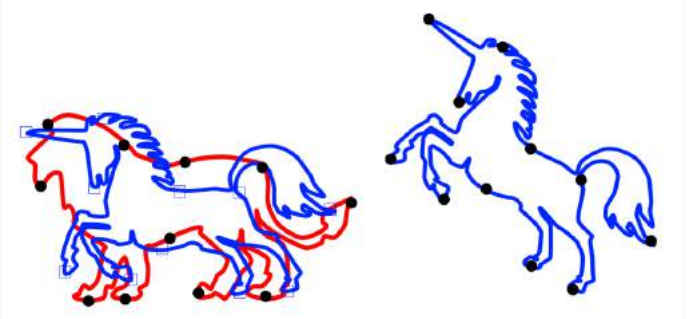
```
 1: // Set p, q
 2: p = new Vector2[n];
 3: q = new Vector2[n];
 4: centerP = Vector2.zero;
 5: centerQ = Vector2.zero;
 6:
 7: for(int i = 0; i < n; i++)
 8: {
 9:     Vector2 pos = _destination.transform.GetChild(i).position;
10:     p[i] = pos;
11:     centerP += pos;
12:
13:     pos = _target.transform.GetChild(i).position;
14:     q[i] = pos;
15:     centerQ += pos;
16: }
17: centerP /= n;
18: centerQ /= n;
19:
20: // Calc p', q'
21: Matrix2x2 H = new Matrix2x2(0, 0, 0, 0);
22: for (int i = 0; i < n; i++)
23: {
24:     p[i] = p[i] - centerP;
25:     q[i] = q[i] - centerQ;
26:
27:     H += Matrix2x2.OuterProduct(q[i], p[i]);
28: }
29:
30: Matrix2x2 u = new Matrix2x2();
31: Matrix2x2 s = new Matrix2x2();
32: Matrix2x2 v = new Matrix2x2();
33: H.SVD(ref u, ref s, ref v);
34:
35: R = v * u.Transpose();
36: Debug.Log(Mathf.Rad2Deg * Mathf.Acos(R.m00));
37: t = centerP - R * centerQ;
```

7.8 結果

無事, ユニコーンの形状をライオンの形状に整列することができました.



▲図 7.5　実行前



▲図 7.6　実行後

7.9　まとめ

本節では, 特異値分解を使用した Shape Matching 法の実装について解説しました. 今回は 2 次元での実装でしたが, 3 次元での実装も同じアルゴリズムで行うことができます. 説明が至らない部分も多々あったかと思いますが, これを機に, 行列演算の CG 分野での応用方法に興味を持っていただき, 学習を深めていただければ幸いです.

7.10　References

- 3D Geometry for Computer Graphics (`https://igl.ethz.ch/teaching/tau/cg/cg2005/svd.ppt`)

- 理工系の数理 線形代数 (永井敏隆, 永井敦 著) 裳華房
- Singular Value Decomposition (the SVD) : MIT OpenCourseWare (https://www.youtube.com/watch?v=mBcLRGuAFUk)
- Lecture: The Singular Value Decomposition (SVD) : AMATH 301 (https://www.youtube.com/watch?v=EokL7E6o1AE)
- 固有値・固有ベクトルとは何かを可視化してみる @kenmatsu4 (https://qiita.com/kenmatsu4/items/2a8573e3c878fc2da306)

第 8 章

Space Filling

8.1 はじめに

本章では **Space filling** 問題[*1]にフォーカスし、それを解決する手法のひとつであるアポロニウスのギャスケットについて解説します。

なお、本章ではアポロニウスのギャスケットのアルゴリズム解説がメインとなるため、グラフィックプログラミングの話からは少し逸脱しています。

8.2 Space filling 問題

Space filling 問題とは、ひとつの閉じた平面の内部を重なることなく、ある形状で可能な限り埋め尽くす手法を発見するという問題です。この問題は特に幾何学分野や組合せ最適化分野では古くから研究がされてきた領域です。どのような形状の平面に、どうのような形状で埋め尽くすかという組み合わせは無数に存在するので、それぞれの組み合わせに対してさまざまな手法が提案されています。

いくつか例を挙げると

- 矩形パッキング[*2]　：O-Tree 法
- 多角形パッキング[*3]：Bottom-Left 法
- 円パッキング[*4]　　：アポロニウスのギャスケット
- 三角形パッキング[*5]：シェルピンスキーのギャスケット

などがあり、他にも多くの手法が存在します。本章では上記のうち、アポロニウス

[*1] 他にも「平面充填」「パッキング問題」「詰め込み問題」などという呼ばれ方をされています
[*2] 矩形パッキング ... 矩形状の平面内を矩形で埋める
[*3] 多角形パッキング ... 矩形状の平面内を多角形で埋める
[*4] 円パッキング ... 円状の平面内を円で埋める
[*5] 三角形パッキング ... 三角形状の平面内を三角形で埋める

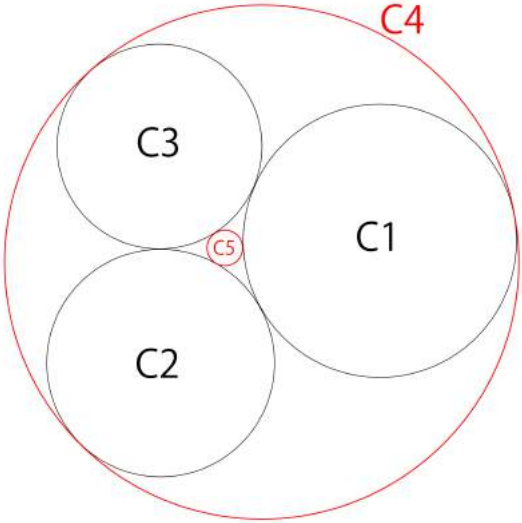
のギャスケットについて解説を行います。

なお、Space filling 問題は NP 困難であることが知られており、上記のどのアルゴリズムを用いても、平面内を常に 100 %埋めるのは現状では難しいとされています。アポロニウスのギャスケットに関しても同様であり、円の内部を円で完全に埋めきることはできません。

8.3 アポロニウスのギャスケット

アポロニウスのギャスケットとは、互いに接する 3 つの円から生成されるフラクタル図形の一種です。これは最初期のフラクタル図形の一種であり、Space filling 問題を解決するために提案されたアルゴリズムなどではなく、平面幾何学の研究成果の一つが Space filling 問題の一つの解に成り得たという話です。この名前は、紀元前のギリシャ人数学者であるペルガのアポロニウスにちなんで付けられています。

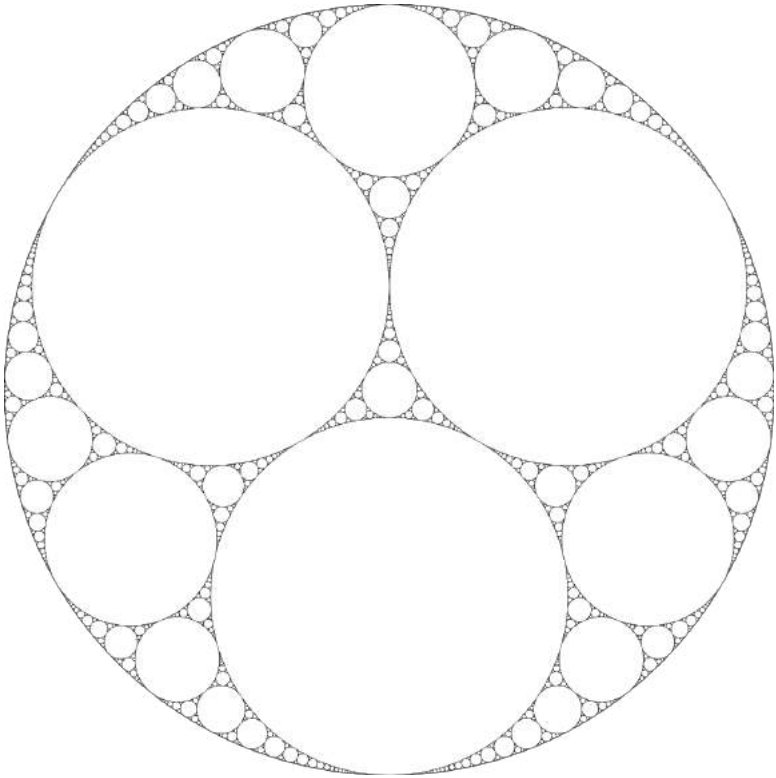
まず、互いに接する 3 つの円をそれぞれ C1,C2,C3 とすると、アポロニウスは C1,C2,C3 全てと接する交差しない 2 つの円 C4,C5 が存在することを発見しました。この C4,C5 は、C1,C2,C3 に対するアポロニウスの円（詳細は後述）となります。



▲図 8.1　C1,C2,C3 とそれに接する 2 つの円 C4,C5

ここで、C1,C2 に対して C4 を考えると、C1,C2,C4 について新たな 2 つのアポロニウスの円を得ることができます。この 2 つの円のうち、一方は C3 となりますが、他方は新しい円 C6 となります。

これを（C1,C2,C5）,（C2, C3, C4）,（C1, C3, C4）のように全ての組み合わせについてアポロニウスの円を考えると、それぞれについて最低 1 つの新しい円を得ることができます。これを無限に繰り返していくことにより、それぞれが接し合う円の集合が作り出されます。この円の集合が、アポロニウスのギャスケットです。



▲図 8.2　アポロニウスのギャスケット

https://upload.wikimedia.org/wikipedia/commons/e/e6/Apollonian_gasket.svg

■メモ: アポロニウスの円
2 つの定点 A,B を取り点 P を $AP : BP =$ 一定 となるように取ったときの点 P の軌跡のことです。 ただ、これとは別にアポロニウスの問題に対する解を指して、アポロニウスの円と呼ばれることもあり、アポロニウスのギャスケットにおいてはこちらの意味合いの方が強いです。

■メモ: アポロニウスの問題
ユークリッド平面幾何学において、与えられた 3 つの円に接する 4 つ目の円を描くという問題です。 この 4 つ目の円は最大で 8 つの解が存在するとされており、その内 2 つの解は常に 3 円と外接し、2 つの円は常に 3 円と内接するとされています。
ちなみに、条件として与えられる 3 つの円は接し合っている必要はなく、あくまでその 3 つの円に接する 4 つ目の円を描くというのが問題です。

8.4 アポロニウスのギャスケットの計算

ここからは実際のプログラムをみながら、アポロニウスのギャスケットの計算方法を順に説明していきます。サンプルプログラムが Gihub に上がっていますので、必要であればそちらから DL してご活用ください。

URL : `https://github.com/IndieVisualLab/UnityGraphicsProgramming2`

8.4.1 事前準備

アポロニウスのギャスケットをプログラムするにあたり、今回は円を表現するクラスと複素数を扱うための構造体を自前で用意しています。

▼ Circle.cs

```
using UnityEngine;

public class Circle
{
```

```
    public float Curvature
    {
        get { return 1f / this.radius; }
    }
    public Complex Complex
    {
        get; private set;
    }
    public float Radius
    {
        get { return Mathf.Abs(this.radius); }
    }
    public Vector2 Position
    {
        get { return this.Complex.Vec2; }
    }

    private float radius = 0f;


    public Circle(Complex complex, float radius)
    {
        this.radius = radius;
        this.Complex = complex;
    }

    /// ...
    /// 以下、円同士の関係性を調べる関数が実装されています
    /// 接しているか、交わっているか、内包しているか、など
    /// ...
}
```

円を表現するクラスの実装の一部です。

基本的なプログラミングの知識があれば難しいことは何も無いと思います。なお、Complexとは今回自前で用意した複素数構造体のことで、Curvatureは曲率と呼ばれるもので、どちらもアポロニウスのギャスケットを計算するのに必要な値です。

▼ Complex.cs

```
using UnityEngine;
using System;
using System.Globalization;

public struct Complex
{
    public static readonly Complex Zero = new Complex(0f, 0f);
    public static readonly Complex One = new Complex(1f, 0f);
    public static readonly Complex ImaginaryOne = new Complex(0f, 1f);

    public float Real
    {
        get { return this.real; }
    }
    public float Imaginary
```

```
    {
        get { return this.imaginary; }
    }
    public float Magnitude
    {
        get { return Abs(this); }
    }
    public float SqrMagnitude
    {
        get { return SqrAbs(this); }
    }
    public float Phase
    {
        get { return Mathf.Atan2(this.imaginary, this.real); }
    }
    public Vector2 Vec2
    {
        get { return new Vector2(this.real, this.imaginary); }
    }

    [SerializeField]
    private float real;
    [SerializeField]
    private float imaginary;


    public Complex(Vector2 vec2) : this(vec2.x, vec2.y) { }

    public Complex(Complex other) : this(other.real, other.imaginary) { }

    public Complex(float real, float imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    /// ...
    /// 以下、複素数計算をする関数が実装されています
    /// 四則演算や絶対値計算など
    /// ...
}
```

複素数を扱うための構造体です。

C#は Complex構造体を有していますが.Net4.0 から搭載されています。本章を執筆している時点では Unity の.Net4.6 サポートは Experimental 段階だったので、自前で用意することにしました。

8.4.2 最初の 3 つの円の計算

アポロニウスのギャスケットを計算するための前提条件として、互いに接する 3 つの円が存在している必要があります。そこで今回のプログラムでは、ランダムに半径を決めた 3 つの円を生成し、それらが互いに接する様に座標を計算して配置しています。

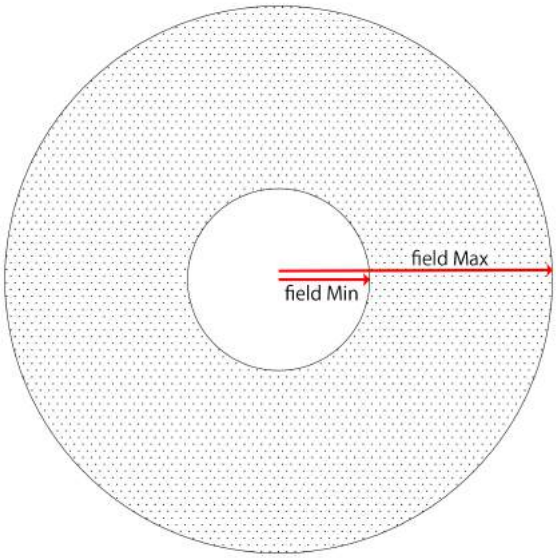
▼ ApollonianGaskets.cs

```
private void CreateFirstCircles(
    out Circle c1, out Circle c2, out Circle c3)
{
    var r1 = Random.Range(
        this.firstRadiusMin, this.firstRadiusMax
    );
    var r2 = Random.Range(
        this.firstRadiusMin, this.firstRadiusMax
    );
    var r3 = Random.Range(
        this.firstRadiusMin, this.firstRadiusMax
    );

    // ランダムな座標を取得
    var p1 = this.GetRandPosInCircle(
        this.fieldRadiusMin,
        this.fieldRadiusMax
    );
    c1 = new Circle(new Complex(p1), r1);

    // p1 を元に接する円の中心座標を計算
    var p2 = -p1.normalized * ((r1 - p1.magnitude) + r2);
    c2 = new Circle(new Complex(p2), r2);

    // 2 つの円に接する円の中心座標を計算
    var p3 = this.GetThirdVertex(p1, p2, r1 + r2, r2 + r3, r1 + r3);
    c3 = new Circle(new Complex(p3), r3);
}

private Vector2 GetRandPosInCircle(float fieldMin, float fieldMax)
{
    // 適当な角度を取得
    var theta = Random.Range(0f, Mathf.PI * 2f);

    // 適当な距離を計算
    var radius = Mathf.Sqrt(
        2f * Random.Range(
            0.5f * fieldMin * fieldMin,
            0.5f * fieldMax * fieldMax
        )
    );

    // 極座標系からユークリッド平面に変換
    return new Vector2(
        radius * Mathf.Cos(theta),
        radius * Mathf.Sin(theta)
    );
}

private Vector2 GetThirdVertex(
    Vector2 p1, Vector2 p2, float rab, float rbc, float rca)
{
    var p21 = p2 - p1;

    // 余弦定理によって角度を計算
    var theta = Mathf.Acos(
```

```
            (rab * rab + rca * rca - rbc * rbc) / (2f * rca * rab)
        );

        // 起点となる点の角度を計算して加算
        // theta はあくまで三角形内における角度でしかないので、平面における角度ではない
        theta += Mathf.Atan2(p21.y, p21.x);

        // 極座標系からユークリッド平面に変換した座標を起点となる座標に加算
        return p1 + new Vector2(
            rca * Mathf.Cos(theta),
            rca * Mathf.Sin(theta)
        );
    }
```

CreateFirstCircles関数を呼び出すことで、初期条件の 3 円が生成されます。

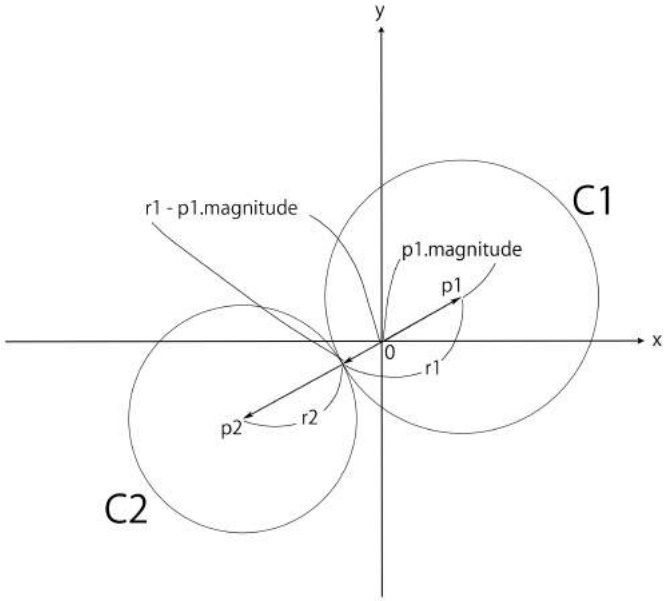
まずランダムに 3 つの半径 r1,r2,r3を決め、次に GetRandPosInCircle関数によって r1の半径をもつ円（以下 C1）の中心座標を決定します。この関数は、原点中心の半径 fieldMin以上 fieldMax以下の円の内部のランダム座標を返します。



▲図 8.3　ランダムな座標が生成される領域

次に r2の半径をもつ円（以下 C2）の中心座標を計算します。まず（$r1 - p1.magnitude$）$+ r2$ によって原点から C2 の中心までの距離を計算します。こ

れを符号反転した C1 の正規化座標に乗算することで、C1 と隣接する r2の半径をもつ円の中心座標を求めることができます。

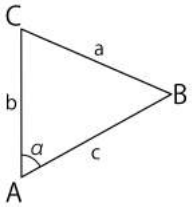


▲図 8.4　p2 ベクトルの表す位置

最後に r3の半径をもつ円（以下 C3）の中心座標を GetThirdVertex関数にて計算しますが、この計算では余弦定理を用います。読者のほとんどの方が高校で習っているかと思いますが、余弦定理というのは三角形の辺の長さと内角の余弦（cos のことです）の間に成り立つ定理のことです。△ ABC において、 $a = BC, b = CA, c = AB$, $\alpha = \angle\ CAB$ とすると

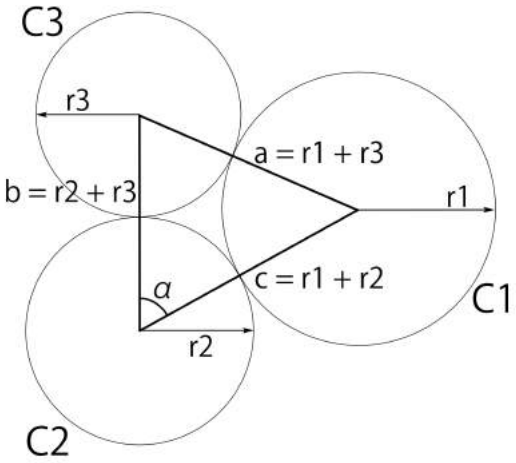
$$a^2 = c^2 + b^2 - 2cbcos\ \alpha$$

が成り立つというのが余弦定理です。



▲図 8.5　三角形 ABC

なぜ円の中心を考えるのに三角形が必要になるのかと思われた方もいらっしゃるかと思いますが、実は 3 つの円の間に成り立つ関係性によって、とても使い勝手の良い三角形を考えることができます。C1,C2,C3 の各中心を頂点とする三角形を考えると、この 3 つの円はそれぞれが接しているので、円の半径から三角形の各辺の長さを知ることができます。



▲図 8.6　三角形 ABC と円 C1,C2,C3

余弦定理を変形すると

$$cos\ \alpha\ = \frac{c^2 + b^2 - a^2}{2cb}$$

となるため、3 つの辺の長さから余弦について解くことができます。二辺のなす角と距離が求まれば、そこから C1 の中心座標を元にして C3 の中心座標を求められるようになります。。

これで、初期条件として必要な互いに接する 3 つの円を生成することができました。

8.4.3 C1,C2,C3 に接する円の計算

前項で生成した 3 つの円 C1,C2,C3 を元にして、これらに接する円を計算します。新しく円を作るためには半径と中心座標の二つのパラメーターが必要となるので、それぞれを計算にて求めます。

半径

まず半径から計算を行いますが、これはデカルトの円定理によって求めることができます。デカルトの円定理とは、互いに接する 4 つの円 C1,C2,C3,C4 について、曲率[*6]をそれぞれ k1,k2,k3,k4 とすると

$$(k_1 + k_2 + k_3 + k_4)^2 = 2({k_1}^2 + {k_2}^2 + {k_3}^2 + {k_4}^2)$$

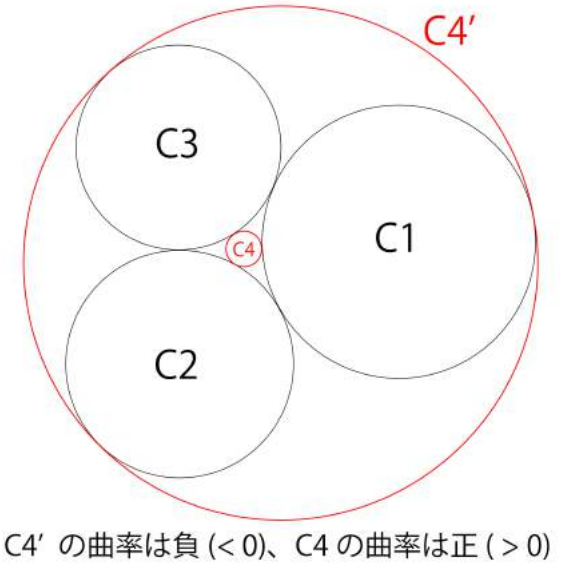
が成り立つというものです。これは 4 つの円の半径に関する 2 次方程式ですが、この式を整理すると

$$k_4 = k_1 + k_2 + k_3 \pm 2\sqrt{k_1k_2 + k_2k_3 + k_3k_1}$$

と変形できるため、3 つの円 C1,C2,C3 が分かっていれば 4 つ目の円 C4 の曲率を求められます。曲率とは半径の逆数であるので、曲率の逆数を取ることにより円の半径を知ることができます。

ここで、C4 の曲率は複合により 2 つ求められますが、一方の解は常に正となり、もう一方は正負のどちらかとなります。C4 の曲率が正の時は C1,C2,C3 と外接し、負の時は C1,C2,C3 と内接（3 円を内包）しています。つまり、4 つ目の円 C4 は 2 パターン考えることができ、その両方について取り得る可能性があります。

[*6] 半径の逆数のことで $k = \pm\frac{1}{r}$ で定義されます



▲図 8.7　曲率の正負

これをプログラミングしているのが以下の部分となります。

▼ SoddyCircles.cs

```
// 曲率の計算
var k1 = this.Circle1.Curvature;
var k2 = this.Circle2.Curvature;
var k3 = this.Circle3.Curvature;

var plusK = k1 + k2 + k3 + 2f * Mathf.Sqrt(k1 * k2 + k2 * k3 + k3 * k1);
var minusK = k1 + k2 + k3 - 2f * Mathf.Sqrt(k1 * k2 + k2 * k3 + k3 * k1);
```

なお、このデカルトの円定理は後にソディという化学者により再発見されており、C1,C2,C3,C4 の円はソディの円と呼ばれています。

■メモ: ソディの円とアポロニウスの円
前項にてアポロニウスの円という話をしましたが、これはソディの円と何が違うのかと思った方もいらっしゃるかと思います。
アポロニウスの円というのは、アポロニウスの問題を解決する円の総称のことです。 ソディの円というのは、デカルトの円定理を満たす 4 つの円を指している言葉です。
つまり、ソディの円がアポロニウスの問題に対する解の一つであるので、アポロニウスの円でもあるということです。

中心座標

次に中心座標の計算ですが、デカルトの円定理と似た形のデカルトの複素数定理によって求められます。デカルトの複素数定理とは、複素数平面上の互いに接する円 C1,C2,C3,C4 の中心座標を z1,z2,z3,z4 として、曲率を k1,k2,k3,k4 とすると

$$(k_1z_1 + k_2z_2 + k_3z_3 + k_4z_4)^2 = 2({k_1}^2{z_1}^2 + {k_2}^2{z_2}^2 + {k_3}^2{z_3}^2 + {k_4}^2{z_4}^2)$$

が成り立つというものです。この式を z4 について整理すると

$$z4 = \frac{z_1k_1 + z_2k_2 + z_3k_3 \pm 2\sqrt{k_1k_2z_1z_2 + k_2k_3z_2z_3 + k_3k_1z_3z_1}}{k4}$$

と変形できるので、これで円 C4 の中心座標を求めることができます。

ここで、半径の計算の時に曲率が 2 つ求められましたが、デカルトの複素数定理についても複合により 2 つ求めることができます。ただし曲率の計算とは違い、2 つのうちどちらか一方が正しいソディの円であるため、どちらが正しいかを判定する必要があります。

これをプログラミングしているのが以下の部分となります。

▼ SoddyCircles.cs

```
/// 中心座標の計算
var ck1 = Complex.Multiply(this.Circle1.Complex, k1);
var ck2 = Complex.Multiply(this.Circle2.Complex, k2);
var ck3 = Complex.Multiply(this.Circle3.Complex, k3);

var plusZ = ck1 + ck2 + ck3
    + Complex.Multiply(Complex.Sqrt(ck1 * ck2 + ck2 * ck3 + ck3 * ck1), 2f);
var minusZ = ck1 + ck2 + ck3
    - Complex.Multiply(Complex.Sqrt(ck1 * ck2 + ck2 * ck3 + ck3 * ck1), 2f);
```

```
var recPlusK = 1f / plusK;
var recMinusK = 1f / minusK;

// ソディの円の判定
this.GetGasket(
    new Circle(Complex.Divide(plusZ, plusK), recPlusK),
    new Circle(Complex.Divide(minusZ, plusK), recPlusK),
    out c4
);

this.GetGasket(
    new Circle(Complex.Divide(plusZ, minusK), recMinusK),
    new Circle(Complex.Divide(minusZ, minusK), recMinusK),
    out c5
);
```

▼ SoddyCircles.cs

```
/// ソディの円の判定
(c1.IsCircumscribed(c4, CalculationAccuracy)
    || c1.IsInscribed(c4, CalculationAccuracy)) &&
(c2.IsCircumscribed(c4, CalculationAccuracy)
    || c2.IsInscribed(c4, CalculationAccuracy)) &&
(c3.IsCircumscribed(c4, CalculationAccuracy)
    || c3.IsInscribed(c4, CalculationAccuracy))
```

▼ Circle.cs

```
public bool IsCircumscribed(Circle c, float accuracy)
{
    var d = (this.Position - c.Position).sqrMagnitude;
    var abs = Mathf.Abs(d - Mathf.Pow(this.Radius + c.Radius, 2));

    return abs <= accuracy * accuracy;
}

public bool IsInscribed(Circle c, float accuracy)
{
    var d = (this.Position - c.Position).sqrMagnitude;
    var abs = Mathf.Abs(d - Mathf.Pow(this.Radius - c.Radius, 2));

    return abs <= accuracy * accuracy;
}
```

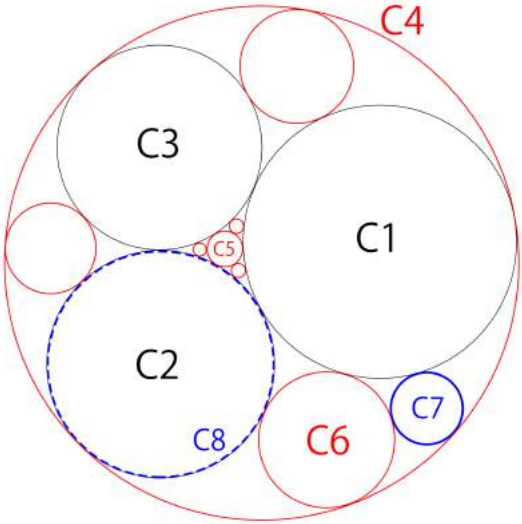
これで、初期条件の C1,C2,C3 を元にして、これらに接する円を 2 つ（以下 C4,C5）得ることができました。

8.4.4 アポロニウスのギャスケットの計算

ここまで来たら、あとは簡単にアポロニウスのギャスケットを計算することができます。「8.4.3 C1,C2,C3 に接する円の計算」で行った計算を繰り返すだけです。

前項では「8.4.2 最初の 3 つの円の計算」で求めた C1,C2,C3 について接する円 C4,C5 を求めました。次は（C1,C2,C4）（C1,C2,C5）（C2,C3,C4）（C2,C3,C5）（C3,C1,C4）（C3,C1,C5）について接する円をそれぞれ求めていきます。

ここで、組み合わせの上では接している円であっても、実際には他の円に重なってしまっている可能性があります。なので正しいソディの円であるかの判定を行ったあとに、今まで求めてきた全ての円に重なっていないことも確認する必要があります。



▲図 8.8　C1,C4,C6 に接する円 C7,C8 のうち、C8 は C2 と重なっているためアポロニウスのギャスケットには含まれない

そうして、それぞれについて接する円を新たに得ることができます。あとは同じよう、接する円を求めるために元とした円と、新しく求めた接する円との全ての組み合わせについて、改めて新しい接する円を求め続けます。

数学的にはこの手順を無限回繰り返した時に得られる円の集合がアポロニウスのギャスケットなのですが、プログラムについて無限回を扱うことはできません。なので、今回のプログラムにおいては、新しく求めた接する円の半径が一定値以下だった場合は、その組み合わせについては処理を終了という条件を与えています。

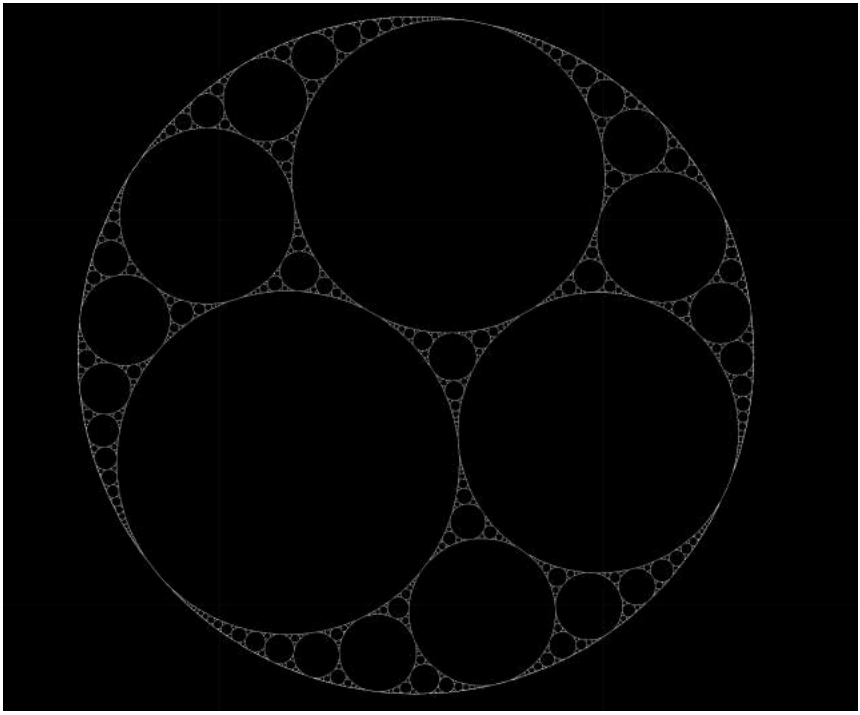
これをプログラミングしているのが以下の部分となります。

▼ ApollonianGaskets.cs

```
private void Awake()
{
    // 初期条件の 3 円を生成
    Circle c1, c2, c3;
    this.CreateFirstCircles(out c1, out c2, out c3);
    this.circles.Add(c1);
    this.circles.Add(c2);
    this.circles.Add(c3);

    this.soddys.Enqueue(new SoddyCircles(c1, c2, c3));

    while(this.soddys.Count > 0)
    {
        // ソディの円を計算
        var soddy = this.soddys.Dequeue();

        Circle c4, c5;
        soddy.GetApollonianGaskets(out c4, out c5);

        this.AddCircle(c4, soddy);
        this.AddCircle(c5, soddy);
    }
}

private void AddCircle(Circle c, SoddyCircles soddy)
{
    if(c == null || c.Radius <= MinimumRadius)
    {
        return;
    }
    // 曲率が負の場合は問答無用で追加
    // 曲率が負の円は一度しか出てこない
    else if(c.Curvature < 0f)
    {
        this.circles.Add(c);
        soddy.GetSoddyCircles(c).ForEach(s => this.soddys.Enqueue(s));

        return;
    }

    // 他の円と被ってないか確認
    for(var i = 0; i < this.circles.Count; i++)
    {
        var o = this.circles[i];

        if(o.Curvature < 0f)
        {
            continue;
```

```
            }
            else if(o.IsMatch(c, CalculationAccuracy) == true)
            {
                return;
            }
        }

        this.circles.Add(c);
        soddy.GetSoddyCircles(c).ForEach(s => this.soddys.Enqueue(s));
    }
```

これで無事アポロニウスのギャスケットを求めることができました。



▲図 8.9　Unity 上での実行結果

8.5 まとめ

ここまでアポロニウスのギャスケットを計算するために必要な手順を順序的に説明してきました。冒頭でも説明したましたが、本来アポロニウスのギャスケットというのはフラクタル図形という意味合いの方が強いです。

しかし、今回の平面という制限を外し空間という世界に飛び出すと、途端に話が難しくなり、フラクタル図形から充填（パッキング）としての意味合いが強くなります。空間を球充填するという命題は、ケプラー予想などの有名な数学的予想が存在するなど、何百年に渡って議論の的とされてきた分野です。

Space filling 問題についても、実用的な面で有用な物です。VLSI のレイアウト設計の最適化、布などの部材切り出しの最適化、UV 展開の自動化と最適化など、幅広い分野で応用されています。

今回はアポロニウスのギャスケットという比較的理解しやすく見た目的にも面白い物を選びました。もしパッキングそのものに対する興味が湧いてきましたら、冒頭で紹介したアルゴリズムなどを調べてみてください。

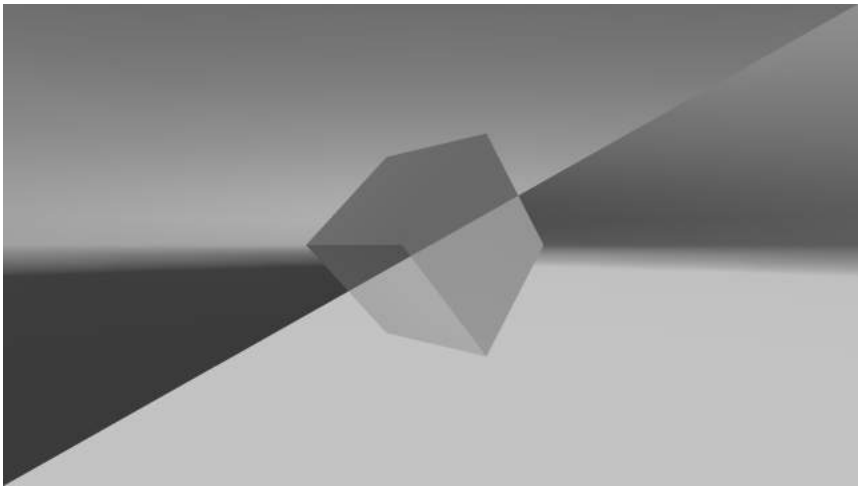
物体内部を物体で埋め尽くす。というのは、意外な所で新しい表現方法として活用できると思います。

8.6 参考

- https://ja.wikipedia.org/wiki/アポロニウスのギャスケット
- https://ja.wikipedia.org/wiki/デカルトの円定理
- http://paulbourke.net/fractals/randomtile/

第 9 章

ImageEffect 入門



▲図 9.1　ImageEffect によるネガポジ反転

出力される映像に対してシェーダ（GPU）を使ってエフェクトを適用する技術 ImageEffect（イメージエフェクト）を Unity で実現する方法についてシンプルに解説します。同技術は PostEffect（ポストエフェクト）とも呼ばれています。

ImageEffect は光を表現するグロー効果、ジャギーを軽減するアンチエイリアシング、被写界深度 DOF の実現、その他、のあらゆる演出のために用いられています。もっとも簡単な例は、ここで紹介するサンプルも扱うような色の変更や修正でしょう。

本章では Unlit シェーダや、Surface シェーダについての基本的な知識や使い方について、多少の前提知識があることを前提に書いていますが、もっとも簡単な構成の

シェーダであるため、前提知識がなくても、読み進めて使うことはできると思います。

本章のサンプルは

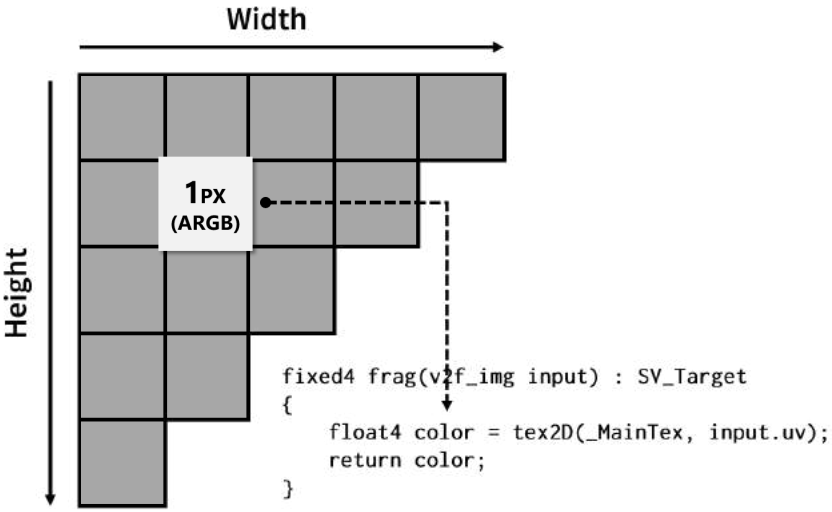
`https://github.com/IndieVisualLab/UnityGraphicsProgramming2`

の「SimpleImageEffect」です。

9.1 ImageEffect の働き

ImageEffect がどのようにしてさまざまな効果を実現しているのかというと、簡潔に述べれば画像処理、すなわち画面を画素単位で操作することによって、さまざまな効果を実現しています。

シェーダによって画素を処理するといえば、フラグメントシェーダです。実質的に、ImageEffect を実装することはフラグメントシェーダを実装することに等しいといえます。



▲図 9.2　ImageEffect の実装はフラグメントシェーダの実装

9.2 Unity における ImageEffect の簡単な流れ

Unity において、ImageEffect の処理順序は概ね次のようになります。

1. カメラがシーンを描画する。

2. カメラの描画内容が OnRenderImage メソッドに入力される。
3. ImageEffect 用のシェーダで、入力された描画内容を修正する。
4. OnRenderImage メソッドが修正された描画内容を出力する。

9.3 サンプルシーンから構成を確認する

もっとも簡単なサンプルシーンを用意しました。サンプルの "ImageEffectBase" シーンを開いて確認してください。関連するスクリプトなどのリソースは、それと同じ名前のものです。

よく似たサンプルに ImageEffect シーンと、同じ名前のリソースがありますが、そちらは後で解説するので注意してください。

サンプルを開くと、シーン中のカメラが映し出すイメージは、ImageEffect によって色がネガポジ反転します。これは Unity が標準で生成する ImageEffect 用のシェーダと同等の効果ですが、実際のソースコードはわずかに異なります。

サンプルシーンの "Main Camera" に "ImageEffectBase" スクリプトがアタッチされていることを確認してください。さらに "ImageEffectBase" では同じ名前のマテリアルが参照され、そのマテリアルには同じ名前のシェーダが設定されています。

9.4 スクリプトの実装

まずは ImageEffect のシェーダをスクリプトから呼び出すまでの処理を先に解説しようと思います。

9.4.1 OnRenderImage メソッド

Unity が出力する映像に変更を加えたいとき、ほとんどの場合には OnRenderImage メソッドを実装する必要があります。OnRenderImage は Start や Update などと同じように Unity の標準のワークフローに定義されるメソッドです。

▼ ImageEffectBase.cs

```
[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
public class ImageEffectBase : MonoBehaviour
{
…
protected virtual void OnRenderImage
    (RenderTexture source, RenderTexture destination)
{
    Graphics.Blit(source, destination, this.material);
}
```

OnRenderImage は、Camera コンポーネントをもつ GameObject に追加されたときにだけ呼び出されます。したがって、ImageEffect クラスには、`[RequireComponent(typeof(Camera))]` を定義しています。

また ImageEffect が適用された結果は、Scene を実行する前から見えていた方がよいので、`ExcludeInEditMode` 属性も定義されています。複数の ImageEffect を切り替えて確認するときや、無効にした状態を確認するときは、ImageEffect のスクリプトを disable にします。

引数 source, destination について

OnRenderImage は引数の 1 つ目（source）に入力、2 つ目（destination）に出力先が与えられています。いずれも RenderTexture 型ですが、特別の設定がないとき、source にはカメラの描画結果が与えられ、destination には null が与えられています。

ImageEffect は、source に入力された絵を変更して destination に書き込む処理を行いますが、destination が null のとき、変更された絵の出力先はフレームバッファ、すなわちディスプレイに見えている領域になります。

また Camera の出力先に RenderTexture が設定されているとき、source はその RenderTexture と同等になります。

Graphics.Blit

`Graphics.Blit` メソッドは、指定したマテリアルとシェーダを使って、入力された RenderTexture を出力となる RenderTexture に描画する処理です。ここでの入力と出力は OnRenderImage の source および destination になります。また、マテリアルは ImageEffect のためのシェーダが設定されたものになります。

原則として、 OnRenderImage メソッドでは、引数の destination に必ず何かしらのイメージデータを渡す必要があります。したがってほとんどの場合に OnRenderImage 内では Graphics.Blit が呼び出されます。

応用的に、たとえば別のエフェクトに使うためのテクスチャを作るとき、テクスチャを複製するときなどにも `Graphics.Blit` を用いることがあります。あるいは、別の方法を用いて destination にデータを渡すことがありますが、ここでは入門のためにそれらの応用例については割愛します。

次の項目は ImageEffect を適用する過程の話と少々異なるので、もし、はじめて読まれるときは、飛ばしてシェーダの解説に進むことを推奨します。

9.4.2 ImageEffect が利用可能か検証する

ImageEffect を解説するにあたり、この項目の実装および解説は不要と思うのですが、より実用的な実装がなされた資料を読むときに障害にならないよう、解説することにしました。Unity が用意する ImageEffect の資料にも同等の機能が実装されています。

ImageEffect は画素毎に演算される処理です。したがって高度な GPU を持たない実行環境では、演算数の多さのために　ImageEffect が歓迎されないことがあります。そこで、ImageEffect が実行環境で利用できるかどうか、開始時に検証し、利用できなければ無効化するのが親切です。

▼ ImageEffectBase.cs

```
protected virtual void Start()
{
    if (!SystemInfo.supportsImageEffects
     || !this.material
     || !this.material.shader.isSupported)
    {
        base.enabled = false;
    }
}
```

検証は Unity が用意する `SystemInfo.supportsImageEffects` によって簡単に実現することができます。

この実装はほとんどのケースで有用と思いますが、たとえばシェーダ側に実装される fallback（フォールバック）機能を使う場合などのときは、異なる実装が必要になることがあります。あくまで参考にしてください。

1 つだけ注意する必要があるとすれば、`this.material` の参照を検証するタイミングです。例では Start メソッドで検証していますが、これが Awake や OnEnable のとき、たとえば `this.material` に参照が与えられていたとしても、Unity は null を示します (そして `base.enabled = false` によってスクリプトが無効になります)。詳細は割愛しますが、`ExcludeInEditMode` の仕様に依存するものです（弊害とは言い難い)。

9.5 もっとも簡単な ImageEffect シェーダの実装

続いて ImageEffect のシェーダについて解説します。ここで紹介するもっとも基本的なサンプルは、Unity が標準的に作成するものと同様に、出力する色を反転するだけの効果を実装しています。

▼ ImageEffectBase.shader

```
Shader "ImageEffectBase"
{
    Properties
    {
        _MainTex("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            CGPROGRAM

            #include "UnityCG.cginc"
            #pragma vertex vert_img
            #pragma fragment frag

            sampler2D _MainTex;

            fixed4 frag(v2f_img input) : SV_Target
            {
                float4 color = tex2D(_MainTex, input.uv);
                color.rgb = 1 - color.rgb;

                return color;
            }

            ENDCG
        }
    }
}
```

大まかな処理の流れとして、_MainTex にカメラが描画するイメージが入力され、フラグメントシェーダで画素に最終的に表示する色を決定する、といったイメージです。

ここで _MainTex に与えられるテクスチャ情報は OnRenderImage の source, Graphics.Blit の source に等しいです。

_MainTex は Graphics.Blit の入力のために Unity によって予約されている点に注意してください。異なる他の名前に変更すると、Graphics.Blit の source が正しくシェーダに入力されません。

9.5.1 Unity が標準で生成するシェーダとの差異

Unity が標準で生成する ImageEffect は次のように少々長く複雑です（抜粋しています）。ImageEffect もシェーダですから、標準的なレンダリングパイプラインを経て最終的な出力を得ます。したがって、ImageEffect が実現する効果には本質的に影響しないように見えるバーテックスシェーダも、ImageEffect のシェーダには定義されている必要があります。

▼ NewImageEffectShader.shader

```
SubShader
{
    Cull Off ZWrite Off ZTest Always

    Pass
    {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.uv;
            return o;
        }

        sampler2D _MainTex;

        fixed4 frag (v2f i) : SV_Target
        {
            fixed4 col = tex2D(_MainTex, i.uv);
            col.rgb = 1 - col.rgb;
            return col;
        }
        ENDCG
    }
}
```

ImageEffect におけるバーテックスシェーダは、カメラに正面を向いて、その全面を埋めるような四角形のメッシュとその UV 座標を、フラグメントシェーダに渡しているだけです。このバーテックスシェーダに手を加えることで実現できる効果もありますが、ほとんどの ImageEffect では不要です。

そのためか、Unity には標準的なバーテックスシェーダとその入力を定義するための構造体が用意されています。それらは "UnityCG.cginc" に定義されています。ここで用意した標準ではないシェーダのソースコードでは、UnityCg.cginc に定義された `vertex vert_img` や `appdata`、`v2f_img` を利用することによって、ソースコー

ド全体を簡潔にしています。

9.5.2 Cull, ZWrite, ZTest

一見するとカリングと Z バッファの書き込み、参照については、標準の値で問題なさそうです。しかしながら Unity は不用意なZバッファへの書き込みを防ぐために、`Cull Off ZWrite Off ZTest Always` の定義を推奨しています。

9.6 もっとも簡単な練習

ImageEffect を簡単に練習してみましょう。サンプルでは単純に全画面をネガポジ反転するようにしていますが、この章の初めに掲載した図のように、イメージ全体の "斜め半分にだけ" ネガポジ反転を適用するようにしてみます。

`input.uv` には、イメージ全体のうち 1 画素分を示す座標が与えられていますから、これを活用します。イメージ全体の各画素は 0 ~ 1 で正規化された x * y 座標で示されます。

動作する一例のコードは、サンプルの "Prtactice" フォルダに含まれていますし、後に続いて解説しますが、もし初心者の方に目を通して頂けているのであれば、まずご自身で実装してみることをオススメします。

9.6.1 上下左右の半分は簡単

上下半分に色を変化させるのはすごく簡単です。これによって、ImageEffect の座標の原点を確認するのがよいと思います。たとえば次の 2 行のコードでは、それぞれ x 座標と y 座標が半分より小さいときに色を反転します。

▼ Practice/ImageEffectShader_01.shader

```
color.rgb = input.uv.x < 0.5 ? 1 - color.rgb : color.rgb;
color.rgb = input.uv.y < 0.5 ? 1 - color.rgb : color.rgb;
```

色の変化から、ImageEffect に与えられる座標の原点は、左下であることが確認できたでしょうか。

9.6.2 斜め半分も簡単

上下左右の半分は簡単だと先にいっていますが、実際には斜め半分も簡単です。次のようなソースコードで斜め半分にエフェクトを適用する（色を反転する）ことができます。

▼ Practice/ImageEffectShader_02.shader

```
color.rgb = input.uv.y < input.uv.x ? 1 - color.rgb : color.rgb;
```

9.7 座標に関する便利な定義値

紹介したように UnityCg.cginc には vertex vert_img や appdata のような便利な関数や構造体が定義されていますが、これら以外にも ImageEffect を実装する上で便利な値が定義されています。

9.7.1 _ScreenParams

_ScreenParams は、float4 型の値で、x, y にはそれぞれ出力するイメージのピクセル幅と高さが、w, z には 1 + 1 / x, 1 + 1 / y が与えられています。

たとえば 640x480 サイズのレンダリングを実行するとき、x = 640, y = 480, z = 1 + 1 / 640, z = 1 + 1 / 480 となります。実際のところ、w と z はそれほど使うことがないでしょう。

一方で x、y の値はたとえばイメージ上の何ピクセルに相当するのか算出したり、あるいはアスペクト比を算出するために頻繁に用いられます。これらは凝ったエフェクトを作る上で重要ですが、わざわざスクリプトから値を与えずとも Unity 側が用意してくれるのは助かりますね。頭の片隅に入れておけば、他の方のシェーダを読むときの助けになることもあると思います。

9.7.2 _TexelSize

同じような定義値の 1 つに、<sampler2Dの変数名>_TexelSize という定義値があります。ここでは _MainTex_TexelSize になります。

_ScreenParams と同じく float4 型の値ですが、x = 1 / width, y = 1 / height, z = width, y = height と、各要素に与えられる値が異なります。また対応する sampler2D 型によって値が異なる点も特徴です。_MainTex にかかわらず、対応する ~_TexlSize を定義すれば、Unity から値が与えられます。

_ScreenParams を使っている ImageEffect も沢山ありますが、どちらかというと _MainTex_TexelSize の方が使いやすいと思います。

9.7.3 1 つとなりの画素を参照する

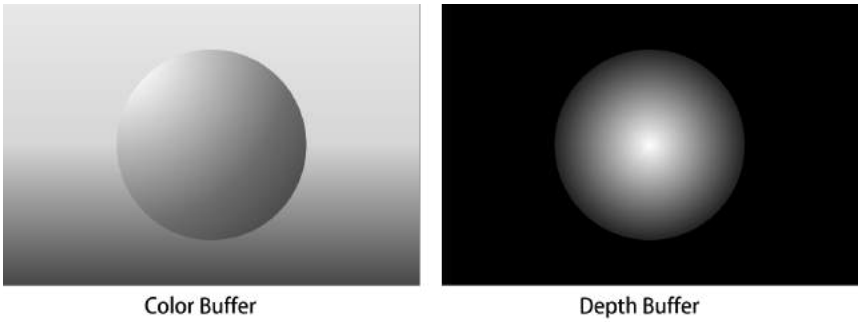
たとえば 1 つ隣の画素の色（値）を参照したい、ということは画像処理などでは頻繁にありますが、次のようなコードによって、隣の画素の値を参照することができます。

▼ Practice/ImageEffectShader_03.shader

```
sampler2D _MainTex;
float4    _MainTex_TexelSize;

fixed4 frag(v2f_img input) : SV_Target
{
    float4 color = tex2D(_MainTex, input.uv);

    color += tex2D(_MainTex, input.uv + float2(_MainTex_TexelSize.x, 0));
    color += tex2D(_MainTex, input.uv - float2(_MainTex_TexelSize.x, 0));
    color += tex2D(_MainTex, input.uv + float2(0, _MainTex_TexelSize.y));
    color += tex2D(_MainTex, input.uv - float2(0, _MainTex_TexelSize.y));

    color = color / 5;

    return color;
}
```

このコードは周辺 4 つの画素を参照して平均の値を返すものです。画像処理では文字どおり平滑化フィルタなどと呼ばれます。他にもより高品質なノイズ軽減のためのフィルタが同じように周辺の画素を参照して実装されることもありますし、たとえばエッジ・輪郭線検出フィルタなどでも用いられています。

9.8 深度と法線の取得



▲図 9.3 G-Buffer のイメージ

モデルに適用するマテリアル（シェーダ）を実装するときは、その多くの場合にモデルの深度や法線情報を参照すると思います。2 次元のイメージ情報を操作する ImageEffect においては、深度や法線情報を取得できないように思われますが、イメージ上のある画素に映されるオブジェクトの深度や法線情報を取得する方法は用意

されています。

技術的な詳細を解説するにはレンダリングパイプラインの解説が必要になり、少々長くなるので割愛させてください。簡潔に説明すると、描画するイメージ上のある画素に対応する深度情報や法線情報はバッファしておくことができます。それらのバッファは G-Buffer と呼ばれます。G-Buffer には色を保存したり、深度を保存したりするものがあります。（ちなみに G-Buffer の読み方は「ゲーバッファー」であると原著論文には示されています。）

オブジェクトの描画時に、深度や法線情報もバッファに書きこんでおき、描画の最後に実行される ImageEffect でそれを参照する、というイメージです。この技術は Deffered レンダリングでは重要な役割を持っていますが、Forward レンダリングでも使うことができます。

これらの解説にはサンプルの "ImageEffect" シーンおよび、それと同名のリソースを使います。

9.8.1 深度と法線情報を取得するための設定

深度と法線情報を ImageEffect で参照するためには、少し設定が必要です。基本的な機能は共通なので、ここでは ImageEffectBase.cs を継承した ImageEffect.cs でその設定を行うようにします。

▼ ImageEffect.cs

```
public class ImageEffect : ImageEffectBase
{
    protected new Camera camera;
    public DepthTextureMode depthTextureMode;

    protected override void Start()
    {
        base.Start();

        this.camera = base.GetComponent<Camera>();
        this.camera.depthTextureMode = this.depthTextureMode;
    }

    protected virtual void OnValidate()
    {
        if (this.camera != null)
        {
            this.camera.depthTextureMode = this.depthTextureMode;
        }
    }
}
```

深度と法線情報を取得するためには、カメラに `DepthTextureMode` を設定する必要があります。これは深度や法線などの情報を、どのように書き込むかを制御するための設定です。初期値は `None`です。

残念ながら DepthTextureMode はカメラの Inspector に表示されないパラメータなので、スクリプトから任意にカメラの参照を取得して設定する必要があります。

OnValidate メソッドについて、あまり利用したことがない方のために説明しておくと、Inspector 上でパラメータが更新されたときに呼び出されるメソッドです。

9.8.2 DepthTextureMode の値

ここで紹介するコードを使って、DepthTextureMode の値を Inspector 上から変更します。いくつか値がありますが、ここでは DepthNormals を使う点に注意してください。

Depth を設定すれば深度情報のみを取得するための設定になります。ただし Depth と DepthNormals とでは、シェーダから深度情報を取得する手順が少々異なります。また MotionVectors を設定すれば、各画素に対応する動きの情報を取得することができ大変面白いのですが、すべて解説すると少々長くなるので、この場では割愛させてください。

9.9 シェーダ上での深度と法線の取得

カメラに DepthTextureMode を設定したとき、シェーダ上から深度情報と法線情報を取得する方法は次のとおりです。

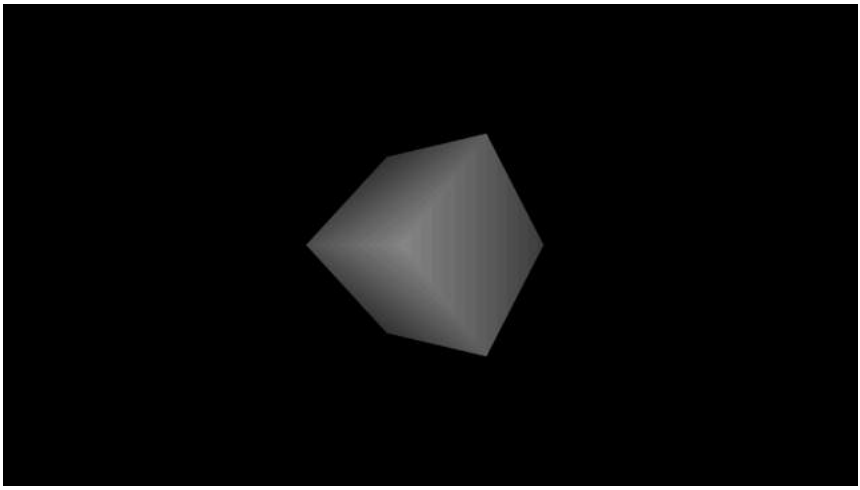
_CameraDepthNormalsTexture は、_MainTex に描画するイメージが与えられるのと同様に、深度と法線の情報が与えられる sampler2D です。したがって input.uv を使って参照すれば、描画するイメージのある画素に対応する深度と法線情報を取得することができます。

▼ ImageEffect.shader

```
sampler2D _MainTex;
sampler2D _CameraDepthNormalsTexture;

fixed4 frag(v2f_img input) : SV_Target
{
    float4 color = tex2D(_MainTex, input.uv);
    float3 normal;
    float  depth;

    DecodeDepthNormal
    (tex2D(_CameraDepthNormalsTexture, input.uv), depth, normal);

    depth = Linear01Depth(depth);
    return fixed4(depth, depth, depth, 1);

    return fixed4(normal.xyz, 1);
}
```

_CameraDepthNormalsTexture から取得することができる値は深度と法線の値が合わさったものなので、これをそれぞれの値に分解する必要があります。分解するための関数は Unity が用意してくれているものを使います。DecodeDepthNormal 関数に、分解したい値とその結果を代入するための変数を与えます。

9.9.1 深度情報の取得と可視化

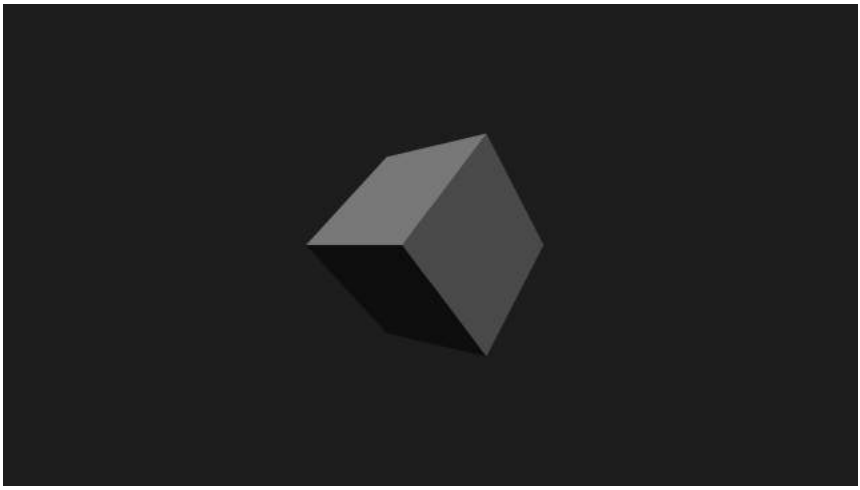


▲図 9.4 ImageEffect による深度の可視化

先に深度情報について説明します。深度情報は実はプラットフォームによって扱いが異なります。Unity ではその差を吸収するためのいくつかの仕組みが用意されていますが、ImageEffect の実装にあたっては、Linear01Depth 関数を使うのがよいと思います。Linear01Depth は、取得した深度の値を 0 ~ 1 に正規化するための関数です。

サンプルでは取得した深度の値を R,G,B に与えることで、深度の値を可視化しています。シーン中のカメラを動かしたり、Clipping Planes の値を Inspector 上から変更するなどして、どのように変化するのかを確認することをオススメします。

9.9.2 法線情報の可視化



▲図 9.5　ImageEffect による法線の可視化

法線情報の可視化については、深度情報ほどの複雑さはありません。法線の情報はスクリプトや一般的なシェーダから参照されるものと同等です。ある画素に映される面の方向を示す X,Y Z の情報が 0 ~ 1 に正規化された形式で与えられています。

法線が正しく取得できているかどうかを確認するだけなら、X,Y,Z の値をそのまま R,G,B とみなして出力すればよいです。つまり右を向く面ほど X = R の値が大きくなり、より赤に、上を向く面ほど Y = G の値が大きくなり、より緑になります。

9.10 参考

本章での主な参考資料は次のとおりです。いずれも Unity 公式のものです。

- Writing Image Effects- `https://docs.unity3d.com/540/Documentation/Manual/WritingImageEffects.html`
- Accessing shader properties in Cg/HLSL - `https://docs.unity3d.com/Manual/SL-PropertiesInPrograms.html`
- Using Depth Textures - `https://docs.unity3d.com/ja/current/Manual/SL-DepthTextures.html`

第 10 章

ImageEffect 応用（SSR）

10.1 はじめに

本章では ImageEffect の応用として、Screen Space Reflection の理論と実装について解説します。3 次元空間を構成する際に、映り込みや反射は陰影と並んでリアリティを表現するために役立つものです。しかしながら、反射や映り込みは、我々が日常で目にする現象の単純さにも関わらず、3 DCG の世界でレイトレーシング（後述）などを用いて物理現象に忠実に再現しようとすると、膨大な演算量が必要とされる表現でもあります。最近では Unity で OctanRenderer が使えるようになり、映像作品として制作する場合は Unity でもかなりフォトリアルな演出が可能になってきてはいますが、リアルタイムレンダリングではまだまだ工夫を凝らして擬似的に再現する必要があります。

リアルタイムレンダリングで反射を表現するテクニックはいくつか存在しますが、本章では Screen Space Reflection (以下 SSR) という、ポストエフェクトに属する手法を紹介していきます。

本章の構成としては、先ずポストエフェクトの肩慣らしとして、サンプルプログラム中でも使用されているブラー処理について先取りして解説していきます。その後、SSR について、できるだけ小さな処理の単位に分解しながら、解説していきます。

また、本章のサンプルは

`https://github.com/IndieVisualLab/UnityGraphicsProgramming2`

の「SSR」に入っています。

10.2 Blur

本節ではブラー処理について解説していきます。アンチエイリアシングまで含めると、ブラー処理も大変ややこしい手続きを理解する必要がありますが、今回は肩慣らしなので基本的な処理です。ブラー処理の基本は、処理を施したい画像の各テクセル

（ラスタライズされた後のピクセルのこと[*1]）に対して、その周辺のテクセルを参照する行列を掛け合わせることで、テクセルの色を均質化していきます。この周辺のテクセルを参照する行列をカーネル（核）と呼びます。カーネルはテクセルの色を混ぜ合わせる割合を決める行列といえます。

ブラー処理のなかでもっともよく使われるのがガウシアンブラーです。これは名前の通り、カーネルにガウス分布を利用する処理を指します。ガウシアンブラーの実装を斜め読みしながら、ポストエフェクトにおける処理の感覚を掴んでいきましょう。

ガウシアンカーネルは処理対象の画素を中心に、ガウス分布に従う割合で輝度を混ぜ合わせます。こうすることで輝度が非線形に変わる輪郭部分のボケを抑えることができます。

数学の復習になりますが、ガウス分布は以下の式で表すことができます。

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

ここでガウス分布は二項分布に近似できるので、以下のように二項分布に従う重み付けの組み合わせでガウス分布の代用ができます（ガウス分布と二項分布の近似については脚注参照[*2]）

▼ GaussianBlur.shader

```
float4 x_blur (v2f i) : SV_Target
{
  float weight [5] = { 0.2270270, 0.1945945, 0.1216216, 0.0540540, 0.0162162 };
  float offset [5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };
  float2 size = _MainTex_TexelSize;
  fixed4 col = tex2D(_MainTex, i.uv) * weight[0];
  for(int j=1; j<5; j++)
  {
    col += tex2D(_MainTex, i.uv + float2(offset[j], 0) * size) * weight[j];
    col += tex2D(_MainTex, i.uv - float2(offset[j], 0) * size) * weight[j];
  }
  return col;
}
```

上記のコードは x 方向のみですが、y 方向もほぼ同様の処理になります。ここで x 方向と y 方向のブラーを分けているのは、2 方向に分割することで、輝度取得の回数を $n * n = n^2$ 回から $n * 2 + 1 = 2n + 1$ 回に縮減できるからです。

[*1] https://msdn.microsoft.com/ja-jp/library/bb219690(v=vs.85).aspx
[*2] https://ja.wikipedia.org/wiki/%E4%BA%8C%E9%A0%85%E5%88%86%E5%B8%83

▲図 10.1　各方向の Blur の合成で正しくブラーがかかることの確認

スクリプト側では`OnRenderImage`で xy それぞれの方向について src と一時的な RenderTexture の間を交互に Blit し、最後にに src から dst へ Blit して出力しています。MacOS では src だけで Blit が可能でしたが、Windows では結果が出力されなかったため、`RenderTexture.GetTemporary`を使用しています。（OnRenderImage と Blit については前章の ImageEffect 入門を参照して下さい。）

▼ GaussianBlur.cs

```
void OnRenderImage(RenderTexture src, RenderTexture dst)
{
  var rt = RenderTexture.GetTemporary(src.width, src.height, 0, src.format);

  for (int i = 0; i < blurNum; i++)
  {
    Graphics.Blit(src, rt, mat, 0);
    Graphics.Blit(rt, src, mat, 1);
  }
  Graphics.Blit(src, dst);

  RenderTexture.ReleaseTemporary(rt);
}
```

以上でガウシアンブラーの解説は終わりです。ポストエフェクトがどのように行われるかの感覚が掴めてきたかと思いますので、次節から SSR の解説を行います。

10.3　SSR

SSR はポストエフェクトの範囲内で反射・映り込みを再現しようとするテクニックです。SSR に必要なのはカメラで撮られた画そのものと、深度情報が書き込まれたデプスバッファ、あとは法線情報が書き込まれたノーマルバッファです。デプスバッファやノーマルバッファなどは G-buffer と総称されるもので、SSR のような Deferred レンダリングにとっては必要不可欠なものとなります。（Deferred レンダリングについては前章の ImageEffect 入門に素晴らしい解説がありますので、ぜひそち

らを参照してください。）

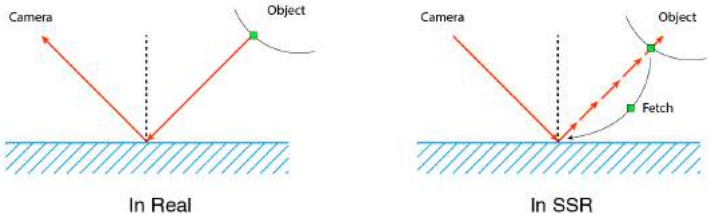
本節を読む際の前提ですが、本節ではレイトレーシングについての基本的な知識を前提にして解説を進めていきます。レイトレーシングについては、入門レベルでも別にもう一章書けるくらい大きなテーマなので、残念ながらここでは説明は割愛させていただきます。ただ、レイトレーシングが何か分からないと以下の内容も理解できないので、分からないという方は PeterShirley の良入門書 "Ray Tracing in One Weekend"[*3]がありますので、先ずそちらを読まれることをおすすめします。

また、SSR の Unity 実装での解説テキストとしては kode80 の「Screen Space Reflections in Unity 5 [*4]」が有名です。また日本語のテキストでは「Unity で Screen Space Reflection の実装をしてみた [*5]」があります。本節では上記のテキストで解説されていることは、極力解説を簡略化したり、枝葉のテクニックについては解説を省略しています。ソースコードを読んで不明点が合った場合は、これらを当たるようにしてみて下さい。

10.3.1 理論概観

SSR の基本的な考え方は、レイトレーシングのテクニックを使い、カメラ、反射表面、オブジェクト（光源）、の関係をシミュレートするものです。

SSR では通常の光学と異なり、カメラに入射する光の道筋から逆算することで、光源を特定した後、反射面にその色をフェッチしてくることで、反射面に反射が再現されます。



▲図 10.2　現実の光学と SSR の光の考え方の違い

SSR ではカメラの各ピクセルに対して、これを行います。

[*3] https://www.amazon.co.jp/gp/product/B01B5AODD8
[*4] http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/
[*5] http://tips.hecomi.com/entry/2016/04/04/022550

処理の大筋は以下のようにまとめることができます。

1. スクリーン座標系を、深度情報を用いることでワールド座標系に戻す
2. 視線ベクトルと法線情報から、反射ベクトルを求める
3. 反射ベクトルを少しだけ伸ばし、その先端（＝レイ）の位置を再度スクリーン座標系に戻す
4. スクリーン座標系のレイの位置について、レイの深度とデプスバッファに書き込まれている深度を比較する
5. レイの方が深度が小さい場合、レイはまだ空中をさまよっている状態。3 に戻りレイをもう少し進める
6. レイの方が深度が大きい場合、レイがなんらか物体を通過したということなので、反射される色を取得できる
7. もとのピクセルに戻り、取得した色を反映する

図では説明しにくい手続きですが、言葉で説明してもややこしいですね。分解して見ていきましょう。

10.3.2 座標変換

先ずはスクリーン座標系とワールド座標系とを変換するための行列をシェーダーに渡してあげます。`_ViewProj` がワールド座標系からスクリーン座標系への変換行列、`_InvViewProj` はその逆行列になります。

▼ SSR.cs

```
void OnRenderImage(RenderTexture src, RenderTexture dst)
{

  ....

  // world <-> screen matrix
  var view = cam.worldToCameraMatrix;
  var proj = GL.GetGPUProjectionMatrix(cam.projectionMatrix, false);
  var viewProj = proj * view;
  mat.SetMatrix("_ViewProj", viewProj);
  mat.SetMatrix("_InvViewProj", viewProj.inverse);

  ....

}
```

さて渡された変換行列を用いて、法線ベクトルと反射ベクトルが求まります。該当するシェーダーの処理を見てみましょう。

▼ SSR.shader

```
float4 reflection (v2f i) : SV_Target
{
  float2 uv = i.screen.xy / i.screen.w;
  float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);

  ...

  float2 screenpos = 2.0 * uv - 1.0;
  float4 pos = mul(_InvViewProj, float4(screenpos, depth, 1.0));
  pos /= pos.w;
  float3 camDir = normalize(pos - _WorldSpaceCameraPos);
  float3 normal = tex2D(_CameraGBufferTexture2, uv) * 2.0 - 1.0;
  float3 refDir = reflect(camDir, normal);

  ....

  if (_ViewMode == 1) col = float4((normal.xyz * 0.5 + 0.5), 1);
  if (_ViewMode == 2) col = float4((refDir.xyz * 0.5 + 0.5), 1);

  ....

  return col;
  }
```

まず該当ピクセルの深度は _CameraDepthTexture に書き込まれており、これ利用します。つぎにスクリーン上の位置情報と震度情報から、該当ピクセルに写っているポリゴンのワールド座標系での位置が分かるので、これを pos に保持します。pos と _WorldSpaceCameraPos から、カメラへ向かうベクトルが分かるので、これと法線情報から、反射ベクトルが分かるようになります。

メインカメラにアタッチされたスクリプトから、法線ベクトルと反射ベクトルがどこを向いているかを確認できるようになってます。各ベクトルとも -1 ~ 1 の間に規格化されているため、0 以下の値の色情報は表示されません。ベクトルが x 軸成分が大きいときは赤っぽく、y 軸成分が大きいときは緑っぽく、z 軸成分が大きいときは青っぽく表示されます。ViewMode を Normal もしくは　Reflection にして確認してみて下さい。

10.3.3　レイトレーシング

では次にレイトレーシングを行う処理を見ていきましょう。

▼ SSR.shader

```
float4 reflection(v2f i) : SV_Target
{

  ...

  [loop]
  for (int n = 1; n <= _MaxLoop; n++)
```

```
    {
      float3 step = refDir * _RayLenCoeff * (lod + 1);
      ray += step * (1 + rand(uv + _Time.x) * (1 - smooth));

      float4 rayScreen  = mul(_ViewProj, float4(ray, 1.0));
      float2 rayUV      = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
      float  rayDepth   = ComputeDepth(rayScreen);
      float  worldDepth = (lod == 0)?
             SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, rayUV) :
             tex2Dlod(_CameraDepthMipmap, float4(rayUV, 0, lod))
             + _BaseRaise * lod;

      ...

      if(rayDepth < worldDepth)
      {

        ....

        return outcol;
      }
    }
  }
```

後に説明する処理に関係する変数も混じっていますが、気にせず読み進めて下さい。ループ内では、まずレイをステップ分だけ伸ばしてあげ、それを再度スクリーン座標系に戻します。スクリーン座標系でのレイの深度と、デプスバッファに書き込まれた深度を比較して、レイの方が奥にある場合、その色を返すことにします。（デプスは最も近い時 1.0 で、遠くなるほど小さくなるので、`rayDepth` が `worldDepth` よりも小さい時、レイが奥にあるという判定になります。）

またループ回数が未定の場合は HLSL はエラーを吐くので、スクリプトからループの回数を渡したい場合は `[loop]` アトリビュートを先頭に書いておく必要があります。

レイトレーシングの骨組みの部分は以上で完成です。基本的な処理はイメージさえできてしまえばそんなに難しくありません。ただし、きれいな反射を再現するためには、これからいくつか処理を追加していく必要があります。重要な改善すべきポイントとしては以下の４つが挙げられるでしょうか。

1. ループ数に制限があるため、レイが進める距離が十分大きくならなず、遠くの物体の映り込みが再現できない
2. レイのステップ数が大きくなると、映り込むオブジェクトを通過してしまい、間違った色をサンプリングしてしまう
3. ループを多数回行うので、単純に処理が重い
4. マテリアルの差異を考慮できていない

アンチエイリアスを含むポストエフェクトでは、処理の効率化のためのテクニック

がむしろ本質的です。処理の骨子の理解が済んだところで、SSR を映像として成立させるテクニックを見ていきましょう。

10.3.4 Mipmap

以下では Chalmers University of Technology の記事 [*6] を参考に、Mipmap 使うことで処理効率をあげる方法について解説します。(Mipmap が何かについては脚注参照 [*7]) レイトレーシングはレイのステップ幅を決めてレイを徐々に進めていくのが基本ですが、Mipmap を使うことでオブジェクトとの交差判定までのレイのステップ幅を可変にすることができます。こうすることで、限られたループ回数の中でも遠くまでレイを飛ばすことができるようになり、また処理効率も上がります。

RenderTexture から Mipmap を使うデモシーンを用意してますのでそちらから確認していきましょう。

▼ Mipmap.cs

```
public class Mipmap : MonoBehaviour
{
  Material mat;
  RenderTexture rt;
  [SerializeField] Shader shader;
  [SerializeField] int lod;

  void OnEnable()
  {
    mat = new Material(shader);
    rt = new RenderTexture(Screen.width, Screen.height, 24);
    rt.useMipMap = true;
  }

  void OnDisable()
  {
    Destroy(mat);
    rt.Release();
  }

  void OnRenderImage(RenderTexture src, RenderTexture dst)
  {
    mat.SetInt("_LOD", lod);
    Graphics.Blit(src, rt);
    Graphics.Blit(rt, dst, mat);
  }
}
```

既製の RenderTexture に対しては mipmap は設定できないようになっているので、ここでは新しく RenderTexture を生成し `src` をコピーしたあと、処理を加えて

[*6] http://www.cse.chalmers.se/edu/year/2017/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf

[*7] https://answers.unity.com/questions/441984/what-is-mip-maps-pictures.html

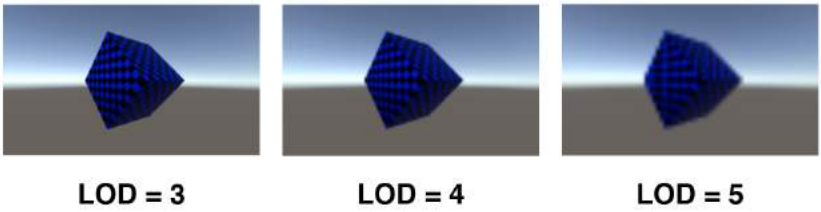
います。

▼ Mipmap.shader

```
sampler2D _MainTex;
float4 _MainTex_ST;
int _LOD;

....

fixed4 frag (v2f i) : SV_Target
{
        return tex2Dlod(_MainTex, float4(i.uv, 0, _LOD));
}
```

tex2Dlod(_MainTex, float4(i.uv, 0, _LOD))でLODに応じたMipmapが取得できるようになってます。

シーン上でカメラにアタッチされたスクリプトからLODを上げていくと、画像が粗くなっていくのが確認できるかと思います。



▲図 10.3　LODの上昇とMipmapの画質の比較

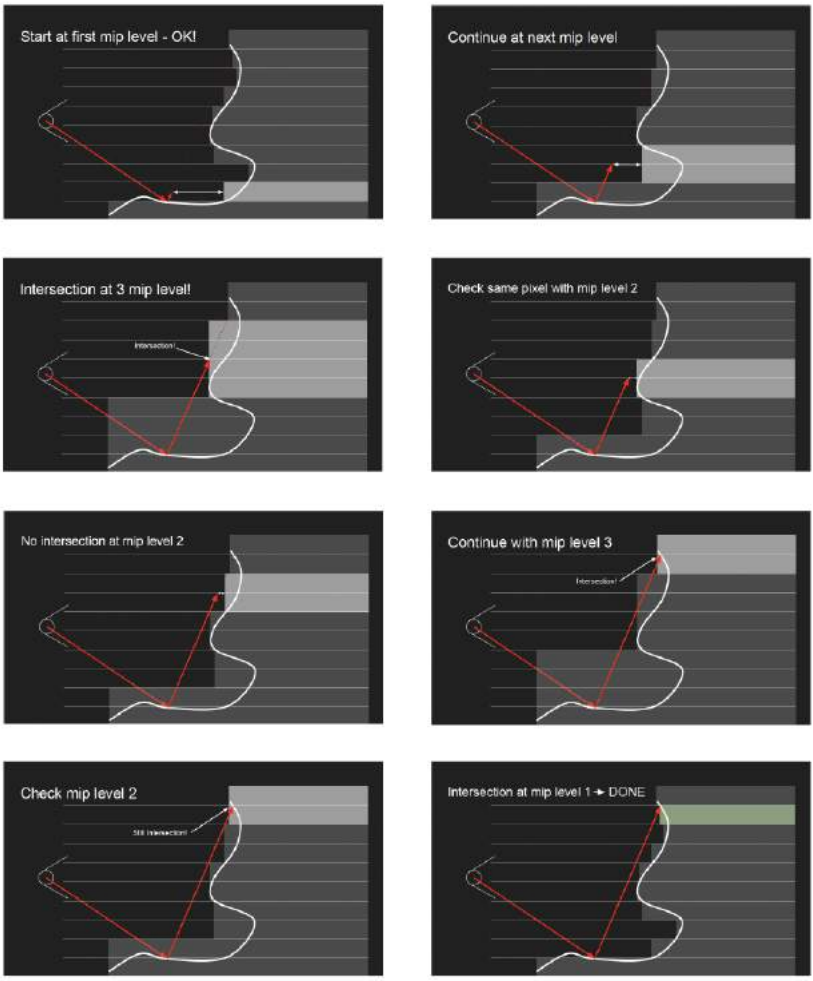
Mipmapの使い方が確認できたところで、SSRシーンのなかでMipmapがどのように利用されているか見ていきましょう。

▼ SSR.shader

```
[loop]
for (int n = 1; n <= _MaxLoop; n++)
{
  float3 step = refDir * _RayLenCoeff * (lod + 1);
  ray += step;

  ....

  if(rayDepth < worldDepth)
  {
    if(lod == 0)
```

```
    {
      if (rayDepth + _Thickness > worldDepth)
      {
        float sign = -1.0;
        for (int m = 1; m <= 8; ++m)
        {
          ray += sign * pow(0.5, m) * step;
          rayScreen = mul(_ViewProj, float4(ray, 1.0));
          rayUV = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
          rayDepth = ComputeDepth(rayScreen);
          worldDepth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, rayUV);
          sign = (rayDepth < worldDepth) ? -1 : 1;
        }
        refcol = tex2D(_MainTex, rayUV);
      }
      break;
    }
    else
    {
      ray -= step;
      lod--;
    }
  }
  else if(n <= _MaxLOD)
  {
    lod++;
  }
  calcTimes = n;
}
if (_ViewMode == 3) return float4(1, 1, 1, 1) * calc / _MaxLoop;

....
```

Chalmers の記事にある図を用いて解説を進めていきます。

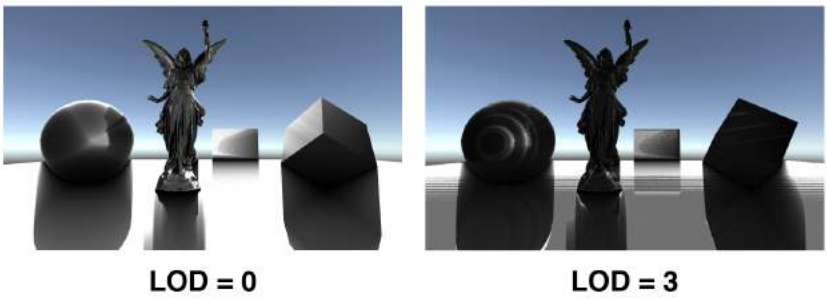


▲図 10.4　Mipmap を用いた計算方法

図のように、最初の数回は慎重に交差判定を行いながら LOD を上げてきます。他メッシュとの交差が無いうちは、大きなステップのまま進めていきます。交差があった場合、Unity の MipMap は平均値を取りながら画素を荒くしていくので、記事の場合とは異なり、レイが行き過ぎてしまう場合があります。そのため一旦単位ステップ分後退し、再度一つ小さな LOD でレイを進めます。最終的に LOD=0 で画像で交

差判定を行うことで、レイの移動距離を伸ばし、処理を効率化することができます。

メインカメラにアタッチされたスクリプトから、LOD を上げた場合どのくらい計算量が変化するかが確認できるようになっています。計算量が多いほど白っぽく、少ないほど黒っぽく見えるようにしています。ViewMode を `CalcCount` にして LOD を変更しながら計算量の変化を確認してみて下さい。



▲図 10.5　LOD の変化による計算量の違い（黒に近いほど計算量が少ない）

10.3.5　二分木探索

二分木探索で交差近傍の精度を上げていく方法を見ていきましょう。早速コードから確認します。

▼ SSR.shader

```
if (lod == 0)
{
  if (rayDepth + _Thickness > worldDepth)
  {
    float sign = -1.0;
    for (int m = 1; m <= 8; ++m)
    {
      ray += sign * pow(0.5, m) * step;
      rayScreen = mul(_ViewProj, float4(ray, 1.0));
      rayUV = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
      rayDepth = ComputeDepth(rayScreen);
      worldDepth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, rayUV);
      sign = (rayDepth < worldDepth) ? -1 : 1;
    }
    refcol = tex2D(_MainTex, rayUV);
  }
  break;
}
```

交差の直後は交差したオブジェクトよりも奥にあるので、まずレイを後退させます。その後レイとメッシュとの前後関係を確認しながら、レイの進行方向を前後どちらか変更していきます。同時にレイのステップ幅を短くすることで、より少ない誤差でメッシュとの交差位置を特定することができます。

10.3.6 マテリアルの違いを反映

ここまでの方法ではスクリーン内のオブジェクトのマテリアルの違いは考慮していませんでした。そのため、全てのオブジェクトが同程度に反射してしまうという問題があります。そこで、再度 G-buffer を使います。 `_CameraGBufferTexture1.w` にはマテリアルの smoothness が格納されているので、これを使います。

▼ SSR.shader

```
if (_ViewMode == 8)
  return float4(1, 1, 1, 1) * tex2D(_CameraGBufferTexture1, uv).w;

....

return
  (col * (1 - smooth) + refcol * smooth) * _ReflectionRate
  + col * (1 - _ReflectionRate);
```

シーン内のオブジェクトに付帯しているマテリアルの smoothness の値を変更すると、そのオブジェクトだけ反射の程度が変更しているのが見て取れます。またメインカメラにアタッチされたスクリプトの ViewMode を `Smoothness` にすることで、シーン内の smoothness を一覧できます。白っぽいほど smoothness が大きくなっています。

10.3.7 Blur 処理

第一節で解説したガウシアンブラー使用している部分です。レイのステップ幅が十分小さくない場合、二分木探索を行っても反射をうまく取得できないことがあります。レイのステップ幅を小さくするとレイの全長が短くなってしまい、また計算量も増えるので、ステップ幅はただ小さくすれば良いというものではなく、適当な小ささにとどめておく必要があります。反射を上手く取得できなかった部分はブラー処理を掛けて、それらしく見せていきます。

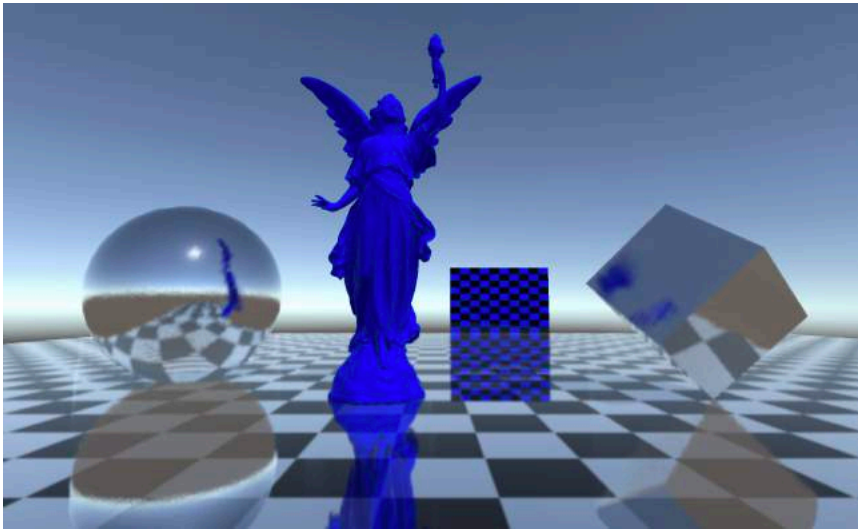
▼ SSR.shader

```
float4 xblur(v2f i) : SV_Target
{
  float2 uv = i.screen.xy / i.screen.w;
  float2 size = _ReflectionTexture_TexelSize;
```

```
    float smooth = tex2D(_CameraGBufferTexture1, uv).w;

    // compare depth
    float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);
    float depthR =
          SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv + float2(1, 0) * size);
    float depthL =
          SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv - float2(1, 0) * size);

    if (depth <= 0) return tex2D(_ReflectionTexture, uv);

    float weight[5] = { 0.2270270, 0.1945945, 0.1216216, 0.0540540, 0.0162162 };
    float offset[5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };

    float4 originalColor = tex2D(_ReflectionTexture, uv);
    float4 blurredColor = tex2D(_ReflectionTexture, uv) * weight[0];

    for (int j = 1; j < 5; ++j)
    {
      blurredColor
          += tex2D(_ReflectionTexture, uv + float2(offset[j], 0) * size)
             * weight[j];

      blurredColor
          += tex2D(_ReflectionTexture, uv - float2(offset[j], 0) * size)
             * weight[j];
    }

    float4 o = (abs(depthR - depthL) > _BlurThreshold) ? originalColor
                : blurredColor * smooth + originalColor * (1 - smooth);
    return o;
  }
```

ここでも前述の理由から xblur と yblur で処理を分けています。またブラー処理を掛けたいのは同一の反射面内のみなので、輪郭部分にはブラー処理が行われないようにしています。左右のデプスの差分が大きい場合、輪郭部分と判定しています。（ yblur では上下の差分を評価しています。）

ここまでの処理を追加した結果が以下になります。



▲図 10.6　結果

10.3.8　おまけ

おまけ程度ですが、メインカメラとサブカメラの 2 台を使って、存在しないオブジェクトが写り込んでいるかのようなテクニックを紹介します。

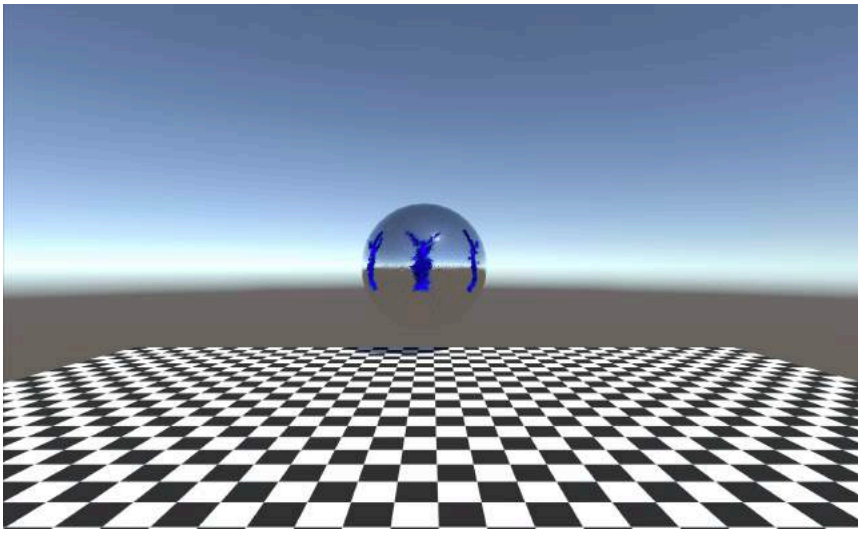
▼ SSRMainCamera.shader

```
float4 reflection(v2f i) : SV_Target
{

  ....

  for (int n = 1; n <= 100; ++n)
  {
    float3 ray = n * step;
    float3 rayPos = pos + ray;
    float4 vpPos = mul(_ViewProj, float4(rayPos, 1.0));
    float2 rayUv = vpPos.xy / vpPos.w * 0.5 + 0.5;
    float rayDepth = vpPos.z / vpPos.w;
    float subCameraDepth = SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv);

    if (rayDepth < subCameraDepth && rayDepth + thickness > subCameraDepth)
    {
      float sign = -1.0;
      for (int m = 1; m <= 4; ++m)
      {
```

```
            rayPos += sign * pow(0.5, m) * step;
            vpPos = mul(_ViewProj, float4(rayPos, 1.0));
            rayUv = vpPos.xy / vpPos.w * 0.5 + 0.5;
            rayDepth = vpPos.z / vpPos.w;
            subCameraDepth = SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv);
            sign = rayDepth - subCameraDepth < 0 ? -1 : 1;
          }
          col = tex2D(_SubCameraMainTex, rayUv);
        }
      }
      return col * smooth + tex2D(_MainTex, uv) * (1 - smooth);
    }
```

極力余分な処理を省いたシンプルな作りにしています。ポイントはデプス評価のために SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv) の代わりに SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv) を使い、参照するオブジェクト情報も _SubCameraMainTex から取得している点です。_CameraDepthTexture, _SubCameraDepthTex はサブカメラからグローバルテクチャとしてセットしています。

欠点は、お互いのカメラが影になって映るべきでないオブジェクトも写してしまう点です。実用性はそれほどないかもしれませんが、ちょっとしたおもしろエフェクトということで。



▲図 10.7　カメラ 2 台を用いた方法

10.4 まとめ

以上で SSR の解説は終わりです。

SSR は大きな処理容量が必要になってくる手法のため、全ての位置にあるオブジェクトをきれいに反射させることは現実的ではありません。そこで、注目するオブジェクトの反射の見映えを良くしながら、些末な反射をいかに少ない処理でそれらしく見せるかがポイントになってきます。またレンダリングされるスクリーンサイズがそのまま計算量に直結するので、想定されるスクリーンサイズと、GPU の性能を加味しながら、映像として成立するポイントを探っていくことが重要です。シーン内のオブジェクトを動かしながら、パラメータを調整することで、各パラメータの役割とトレードオフを確認してみて下さい。

また、ここまでに挙げた、Mipmap や二分木探索、カメラバッファの使い方、その他数々の細かいテクニックは SSR に限らず様々なところで応用が利きます。読者の方々にとって部分的にでも参考になる内容があれば幸いです。

著者紹介

第 1 章 Real-Time GPU-Based Voxelizer - 中村将達 / @mattatz

インスタレーション、サイネージ、Web（フロントエンド・バックエンド）、スマートフォンアプリ等をつくるプログラマー。映像表現やデザインツール開発に興味があります。

- https://twitter.com/mattatz
- https://github.com/mattatz
- http://mattatz.org/

第 2 章 GPU-Based Trail - 福永秀和 / @fuqunaga

元ゲーム開発者、インタラクティブアートを作ってるプログラマ。そこそこややこしい仕組みやライブラリの設計開発が好き。夜型。

- https://twitter.com/fuqunaga
- https://github.com/fuqunaga
- https://fuquna.ga

第 3 章 ライン表現のための GeometryShader の応用 - @kaiware007

雰囲気でやるインタラクティブアーティスト・エンジニア。三度のメシよりインタラクティブコンテンツ好き。お芋が好きでカイワレは食べない。ジェネ系の動画を Twitter によく上げている。たまに VJ をやる。

- https://twitter.com/kaiware007
- https://github.com/kaiware007
- https://www.instagram.com/kaiware007/
- https://kaiware007.github.io/

第 4 章 Projection Spray - すぎのひろのり / @sugi_cho

Unity でインタラクティブアートを作る人間。フリーランス。お仕事お待ちしてます => hi@sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc

第 5 章 プロシージャルノイズ入門 - 大石啓明 / @irishoak

インタラクションエンジニア。インスタレーション、サイネージ、舞台演出、MV、コンサート映像、VJ などの映像表現領域で、リアルタイム、プロシージャルの特性を生かしたコンテンツの制作を行っている。sugi-cho と mattatz とで Aqueduct というユニットを組んで数回活動したことがある。

- https://twitter.com/_irishoak
- https://github.com/hiroakioishi
- http://irishoak.tumblr.com/
- https://a9ueduct.github.io/

第 6 章 Curl Noise - 疑似流体のためのノイズアルゴリズムの解説 - 迫田吉昭 / @sakope

元ゲーム開発会社テクニカルアーティスト。アート・デザイン・音楽が好きで、インタラクティブアートに転向。趣味はサンプラー・シンセ・楽器・レコード・機材いじり。Twitter はじめました。

- https://twitter.com/sakope
- https://github.com/sakope

第 7 章 Shape Matching - 線形代数の CG への応用 - 高尾航大 / @kodai100

元 VFX プロダクションテクニカルアーティスト。現インタラクティブアーティスト・エンジニア。一応まだ学生です。

- https://twitter.com/kodai100_tw
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

第 8 章 Space Filling - @a3geek

インタラクションエンジニア。CG によるシミュレーションの可視化に興味があり、特に正確に可視化するのではなく、より人の感情を揺さぶる可視化がしてみたい。作ることも好きだが、それ以上に知ることの方が楽しいと感じるタイプ。好きな学校の教室は図工室か図書室。

- https://twitter.com/a3geek
- https://github.com/a3geek

第 9 章 ImageEffect 入門 - @XJINE

前回の ComputeShader 入門に引き続き、今回も Graphics Programming というより、入門用のゆるふわ内容でした :-) 他の方が執筆される高度内容にまだついていけなかった方などにリーチしたなら幸いです。

- https://twitter.com/XJINE
- https://github.com/XJINE
- http://neareal.com/

第 10 章 ImageEffect 応用（SSR）- @komietty

物理、Web を経て、インタラクティブエンジニア。新芸術校 3 期。舞台演出に興味あり。

- https://github.com/komietty
- https://www.instagram.com/komietty/

Unity Graphics Programming vol.2

2018 年 4 月 22 日　技術書典 4 版 v1.0.0
著　者　IndieVisualLab
編　集　IndieVisualLab
発行所　IndieVisualLab

IndieVisualLab

a3geek
fuqunaga
irishoak
kaiware007
kodai100
komietty
mattatz
sakope
sugi-cho
XJINE

https://indievisuallab.github.io/