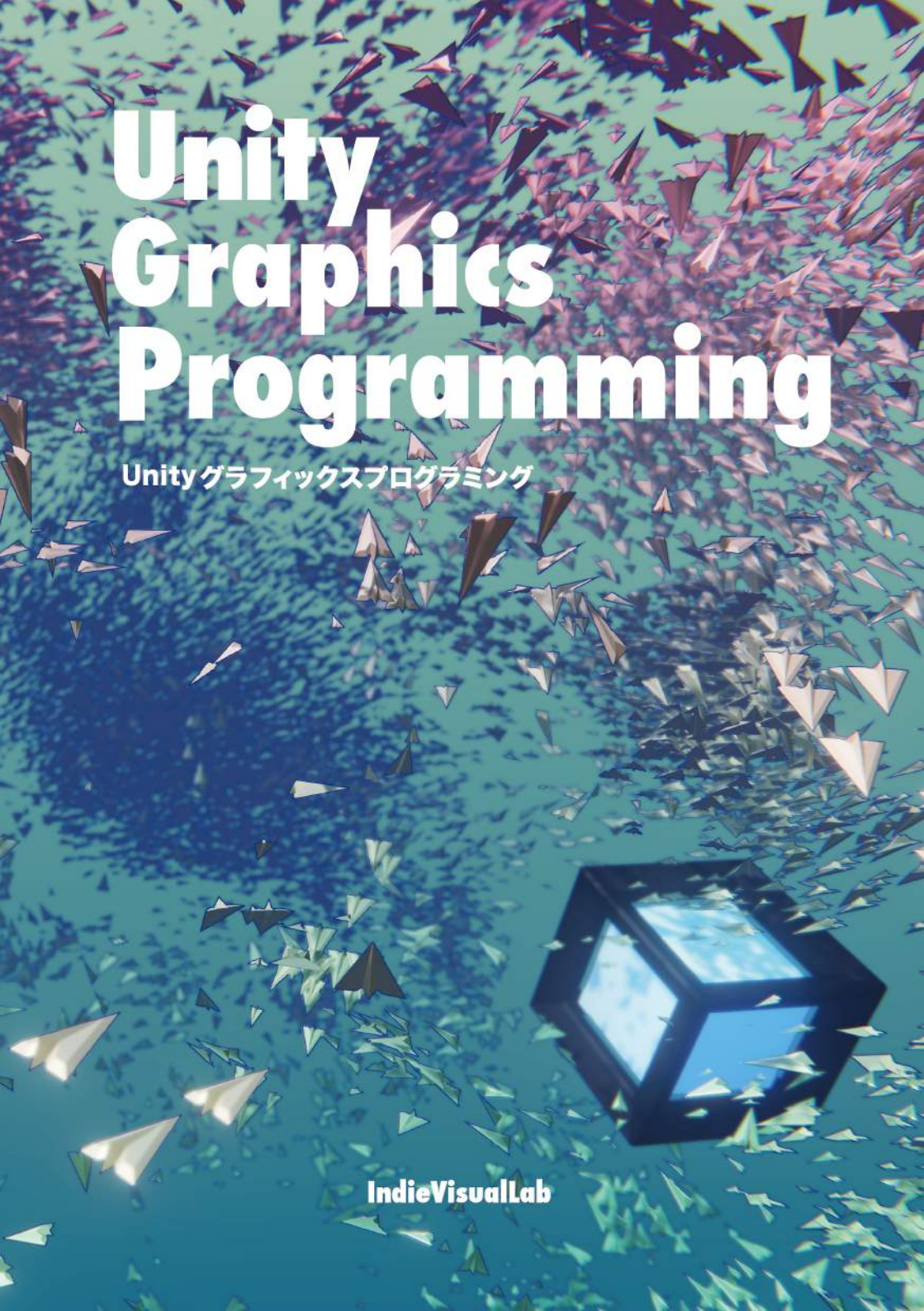

Unity
Graphics
Programming
Unityグラフィックスプログラミング
IndieVisualLab

Unity Graphics Programming

IndieVisualLab　著

2017-10-22 版　　IndieVisualLab 発行

まえがき

本書は主に 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本です。グラフィクスプログラミングと一言で言っても幅広く、Shader テクニックだけ取り上げても多くの書籍が出版されています。本書でも、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の記事が掲載されていますが、ビジュアルとしての結果が見えやすく、自分のエフェクト作成に役立てやすい内容が多いはずです。また、各章で解説されているソースコードについては https://github.com/IndieVisualLab/UnityGraphicsProgramming 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。

目次

第 1 章

Unity ではじめるプロシージャルモデリング

1.1 はじめに

プロシージャルモデリング（Procedural Modeling）とは、ルールを利用して 3D モデルを構築するテクニックのことです。モデリングというと、一般的にはモデリングソフトである Blender や 3ds Max などを利用して、頂点や線分を動かしつつ目標とする形を得るように手で操作をしていくことを指しますが、それとは対象的に、ルールを記述し、自動化された一連の処理の結果、形を得るアプローチのことをプロシージャルモデリングと呼びます。

プロシージャルモデリングは様々な分野で応用されていて、例えばゲームでは、地形の生成や植物の造形、都市の構築などで利用されている例があり、この技術を用いることで、プレイするごとにステージ構造が変わるなどといったコンテンツデザインが可能になります。

また、建築やプロダクトデザインの分野でも、Rhinoceros[*1]という CAD ソフトのプラグインである Grasshopper[*2]を使って、プロシージャルに形状をデザインする手法が活発に利用されています。

プロシージャルモデリングを使えば以下のようなことが可能になります。

- パラメトリックな構造を作ることができる
- 柔軟に操作できるモデルをコンテンツに組み込むことができる

[*1] http://www.rhino3d.co.jp/
[*2] http://www.grasshopper3d.com/

1.1.1 パラメトリックな構造を作ることができる

パラメトリックな構造とは、あるパラメータに応じて構造が持つ要素を変形させられる構造のことで、例えば球（Sphere）のモデルであれば、大きさを表す半径（radius）と、球の滑らかさを表す分割数（segments）といったパラメータが定義でき、それらの値を変化させることで望むサイズや滑らかさを持つ球を得ることができます。

パラメトリックな構造を定義するプログラムを一度実装してしまえば、様々な場面で特定の構造を持つモデルを欲しい形で得ることができ、便利です。

1.1.2 柔軟に操作できるモデルをコンテンツに組み込むことができる

前述の通り、ゲームなどの分野においては、地形や樹木の生成にプロシージャルモデリングが利用される例はとても多く、一度モデルとして書き出されたものを組み込むのではなく、コンテンツ内でリアルタイムに生成されることもあります。リアルタイムなコンテンツにプロシージャルモデリングのテクニックを利用すると、例えば太陽に向かって生える木を任意の位置に生成したり、クリックした位置からビルが立ち並んでいくように街を構築したりするようなことが実現できます。

また、様々なパターンのモデルをコンテンツに組み込むとデータサイズが膨らんでしまいますが、プロシージャルモデリングを利用してモデルのバリエーションを増やせば、データサイズを抑えることができます。

プロシージャルモデリングのテクニックを学び、プログラムによってモデルを構築していくことを極めていけば、モデリングツールそのものを自分で開発することも可能になるでしょう。

1.2 Unity でのモデル表現

Unity では、モデルの形を表すジオメトリデータを Mesh クラスによって管理します。

モデルの形は 3D 空間に並べられた三角形から構成されていて、1 つの三角形は 3 つの頂点により定義されます。モデルが持つ頂点と三角形データの Mesh クラスでの管理方法について、Unity の公式ドキュメントで以下のように解説されています。

> Mesh クラスでは、すべての頂点はひとつの配列に格納されていて、それぞれの三角形は頂点配列のインデックスにあたる 3 つの整数により指定されます。三角形はさらに 1 つの整数の配列として集められます。この整数は配列の最初から 3 つごとにグルーピングされるため、要素 0、1、2 は最初の三角形を定義し、2 つ目の三角形は 3、4、5

と続いていきます。[a]

[a] https://docs.unity3d.com/jp/540/Manual/AnatomyofaMesh.html

モデルには、それぞれの頂点に対応するように、テクスチャマッピングを行うために必要なテクスチャ上の座標を表す uv 座標、ライティング時に光源の影響度を計算するために必要な法線ベクトル（normal とも呼ばれます）を含められます。

サンプルリポジトリ

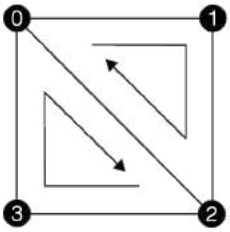
本章では https://github.com/IndieVisualLab/UnityGraphicsProgramming リポジトリ内にある Assets/ProceduralModeling 以下をサンプルプログラムとして用意しています。

C#スクリプトによるモデル生成が主な解説内容となるため、Assets/ProceduralModeling/Scripts 以下にある C#スクリプトを参照しつつ、解説を進めていきます。

■実行環境　本章のサンプルコードは Unity5.0 以上で動作することを確認しています。

1.2.1 Quad

基本的なモデルである Quad を例として、モデルをプログラムから構築する方法を解説していきます。Quad は 4 つの頂点からなる 2 枚の三角形を合わせた正方形モデルで、Unity では Primitive Mesh としてデフォルトで提供されていますが、最も基本的な形状であるため、モデルの構造を理解するための例として役立ちます。



▲図 1.1　Quad モデルの構造　黒丸はモデルの頂点を表し、黒丸内の 0〜3 の数字は頂点の index を示している　矢印は一枚の三角形を構築する頂点 index の指定順（右上は 0,1,2 の順番で指定された三角形、左下は 2,3,0 の順番で指定された三角形）

サンプルプログラム Quad.cs

まずは Mesh クラスのインスタンスを生成します。

▼

```
// Mesh のインスタンスを生成
var mesh = new Mesh();
```

次に Quad の四隅に位置する 4 つの頂点を表す Vector3 配列を生成します。また、uv 座標と法線のデータも 4 つの頂点それぞれに対応するように用意します。

▼

```
// Quad の横幅と縦幅がそれぞれ size の長さになるように半分の長さを求める
var hsize = size * 0.5f;

// Quad の頂点データ
var vertices = new Vector3[] {
    new Vector3(-hsize,  hsize, 0f), // 1 つ目の頂点 Quad の左上の位置
    new Vector3( hsize,  hsize, 0f), // 2 つ目の頂点 Quad の右上の位置
    new Vector3( hsize, -hsize, 0f), // 3 つ目の頂点 Quad の右下の位置
    new Vector3(-hsize, -hsize, 0f)  // 4 つ目の頂点 Quad の左下の位置
};

// Quad の uv 座標データ
var uv = new Vector2[] {
    new Vector2(0f, 0f), // 1 つ目の頂点の uv 座標
    new Vector2(1f, 0f), // 2 つ目の頂点の uv 座標
    new Vector2(1f, 1f), // 3 つ目の頂点の uv 座標
    new Vector2(0f, 1f)  // 4 つ目の頂点の uv 座標
};

// Quad の法線データ
var normals = new Vector3[] {
    new Vector3(0f, 0f, -1f), // 1 つ目の頂点の法線
    new Vector3(0f, 0f, -1f), // 2 つ目の頂点の法線
    new Vector3(0f, 0f, -1f), // 3 つ目の頂点の法線
    new Vector3(0f, 0f, -1f)  // 4 つ目の頂点の法線
};
```

次に、モデルの面を表す三角形データを生成します。三角形データは整数配列によって指定され、それぞれの整数は頂点配列の index に対応しています。

▼

```
// Quad の面データ 頂点の index を 3 つ並べて 1 つの面（三角形）として認識する
var triangles = new int[] {
    0, 1, 2, // 1 つ目の三角形
    2, 3, 0  // 2 つ目の三角形
};
```

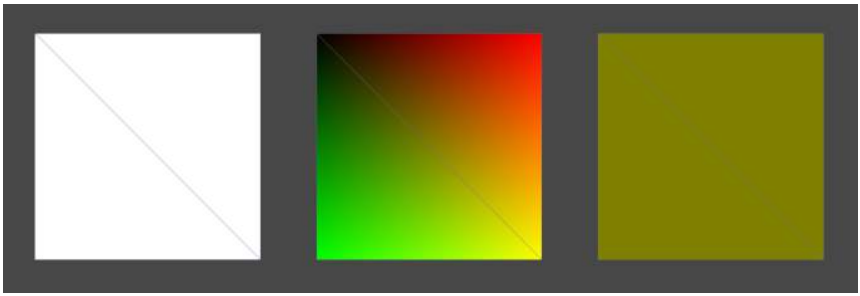
最後に生成したデータを Mesh のインスタンスに設定していきます。

```
mesh.vertices = vertices;
mesh.uv = uv;
mesh.normals = normals;
mesh.triangles = triangles;

// Mesh が占める境界領域を計算する（culling に必要）
mesh.RecalculateBounds();

return mesh;
```

1.2.2 ProceduralModelingBase

本章で利用するサンプルコードでは、ProceduralModelingBase という基底クラスを利用しています。このクラスの継承クラスでは、モデルのパラメータ（例えば、Quad では横幅と縦幅を表す size）を変更するたびに新たな Mesh インスタンスを生成し、MeshFilter に適用することで、変更結果をすぐさま確認することができます。（Editor スクリプトを利用してこの機能を実現しています。ProceduralModelingEditor.cs）

また、ProceduralModelingMaterial という enum 型のパラメータを変更することで、モデルの UV 座標や法線方向を可視化することができます。



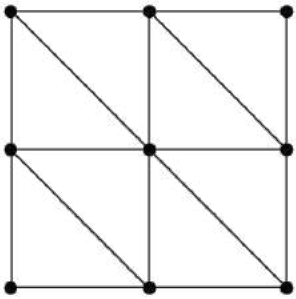
▲図 1.2　左から、ProceduralModelingMaterial.Standard、ProceduralModelingMaterial.UV、ProceduralModelingMaterial.Normal が適用されたモデル

1.3 プリミティブな形状

モデルの構造を理解できたところで、いくつかプリミティブな形状を作っていきましょう。

1.3.1 Plane

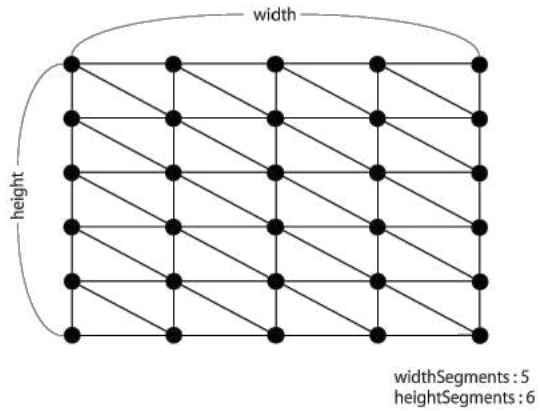
Plane は Quad をグリッド上に並べたような形をしています。



▲図 1.3　Plane モデル

グリッドの行数と列数を決め、それぞれのグリッドの交点に頂点を配置し、グリッドの各マスを埋めるように Quad を構築し、それらをまとめることで 1 つの Plane モデルを生成します。

サンプルプログラム Plane.cs では、Plane の縦に並べる頂点の数 heightSegments、横に並べる頂点の数 widthSegments と、縦の長さ height、横の長さ width のパラメータを用意しています。それぞれのパラメータは次の図のように Plane の形状に影響します。



▲図 1.4　Plane パラメータ

サンプルプログラム **Plane.cs**

まずはグリッドの交点に配置する頂点データを生成していきます。

▼

```
var vertices = new List<Vector3>();
var uv = new List<Vector2>();
var normals = new List<Vector3>();

// 頂点のグリッド上での位置の割合 (0.0 ~ 1.0) を算出するための行列数の逆数
var winv = 1f / (widthSegments - 1);
var hinv = 1f / (heightSegments - 1);

for(int y = 0; y < heightSegments; y++) {
    // 行の位置の割合 (0.0 ~ 1.0)
    var ry = y * hinv;

    for(int x = 0; x < widthSegments; x++) {
        // 列の位置の割合 (0.0 ~ 1.0)
        var rx = x * winv;

        vertices.Add(new Vector3(
            (rx - 0.5f) * width,
            0f,
            (0.5f - ry) * height
        ));
        uv.Add(new Vector2(rx, ry));
        normals.Add(new Vector3(0f, 1f, 0f));
    }
```

```
}
```

次に三角形データですが、各三角形に設定する頂点 index は行と列を辿るループの中で、下記のように参照します。

▼

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

for(int y = 0; y < heightSegments - 1; y++) {
    for(int x = 0; x < widthSegments - 1; x++) {
        int index = y * widthSegments + x;
        var a = index;
        var b = index + 1;
        var c = index + 1 + widthSegments;
        var d = index + widthSegments;

        triangles.Add(a);
        triangles.Add(b);
        triangles.Add(c);

        triangles.Add(c);
        triangles.Add(d);
        triangles.Add(a);
    }
}
```

ParametricPlaneBase

Plane の各頂点の高さ（y 座標）の値は 0 に設定していましたが、この高さを操作することで、単なる水平な面だけではなく、凸凹した地形や小高い山のような形を得ることができます。

ParametricPlaneBase クラスは Plane クラスを継承しており、Mesh を生成する Build 関数を override しています。まずは元の Plane モデルを生成し、各頂点の uv 座標をインプットにして高さを求める Depth(float u, float v) 関数を、全ての頂点について呼び出し、高さを設定し直すことで柔軟に形を変形します。

この ParametricPlaneBase クラスを継承したクラスを実装することで、頂点によって高さが変化する Plane モデルを生成できます。

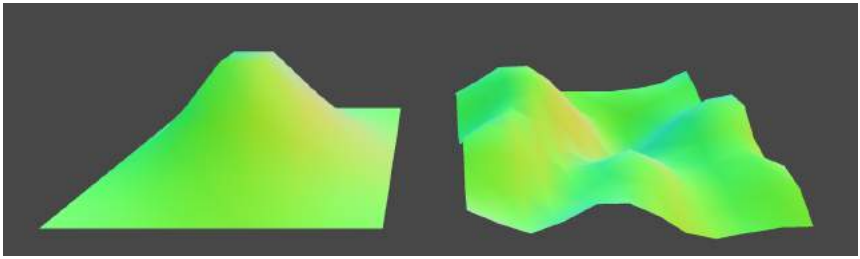
サンプルプログラム ParametricPlaneBase.cs

▼

```
protected override Mesh Build() {
    // 元の Plane モデルを生成
    var mesh = base.Build();

    // Plane モデルが持つ頂点の高さを再設定する
```

```
        var vertices = mesh.vertices;

        // 頂点のグリッド上での位置の割合 (0.0 ~ 1.0) を算出するための行列数の逆数
        var winv = 1f / (widthSegments - 1);
        var hinv = 1f / (heightSegments - 1);

        for(int y = 0; y < heightSegments; y++) {
            // 行の位置の割合 (0.0 ~ 1.0)
            var ry = y * hinv;
            for(int x = 0; x < widthSegments; x++) {
                // 列の位置の割合 (0.0 ~ 1.0)
                var rx = x * winv;

                int index = y * widthSegments + x;
                vertices[index].y = Depth(rx, ry);
            }
        }

        // 頂点位置の再設定
        mesh.vertices = vertices;
        mesh.RecalculateBounds();

        // 法線方向を自動算出
        mesh.RecalculateNormals();

        return mesh;
    }
```

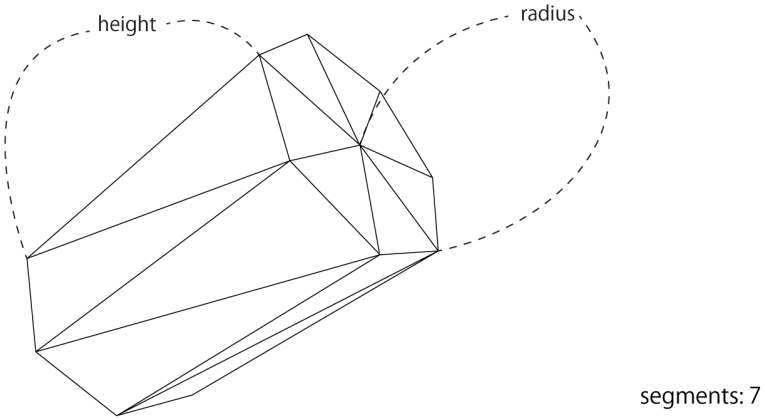
サンプルシーン ParametricPlane.scene では、この ParametricPlaneBase を継承したクラス（MountainPlane、TerrainPlane クラス）を利用した GameObject が配置してあります。それぞれのパラメータを変えながら、形が変化していく様子を確認してみてください。



▲図 1.5　ParametricPlane.scene　左が MountainPlane クラス、右が TerrainPlane クラスによって生成されたモデル

1.3.2 Cylinder

Cylinder は円筒型のモデルで、次の図のような形をしています。



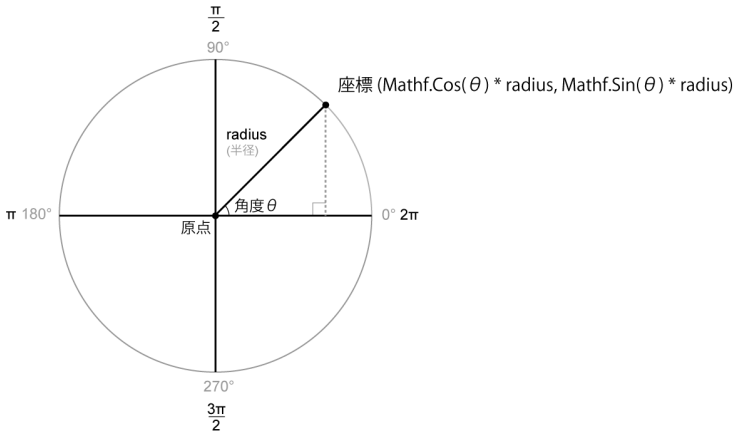
▲図 1.6　Cylinder の構造

円筒型の円のなめらかさは segments、縦の長さと太さはそれぞれ height と radius パラメータで制御することができます。上図の例のように、segments に 7 を指定すると Cylinder は正 7 角形を縦に引き伸ばしたような形になり、segments の数値を大きくするほど円形に近づいていきます。

円周に沿って均等に並ぶ頂点

Cylinder の頂点は、筒の端に位置する円の周りに沿って均等に並べる必要があります。

円周に沿って均等に並ぶ頂点を配置するには、三角関数（Mathf.Sin, Mathf.Cos）を利用します。ここでは三角関数の詳細については割愛しますが、これらの関数を利用すると角度を元に円周上の位置を得ることができます。



▲図 1.7　三角関数から円周上の点の位置を得る

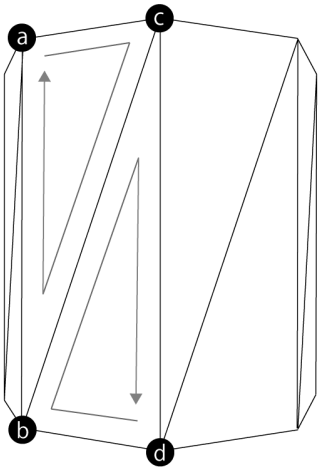
この図のように角度θ (シータ) から半径 radius の円上に位置する点は、(x, y) = (Mathf.Cos(θ) * radius, Mathf.Sin(θ) * radius) で取得することができます。

これを元に、半径 radius の円周上に均等に並べられた segments 個の頂点位置を得るには以下のような処理を行います。

▼

```
for (int i = 0; i < segments; i++) {
    // 0.0 ~ 1.0
    float ratio = (float)i / (segments - 1);

    // [0.0 ~ 1.0] を [0.0 ~ 2 π] に変換
    float rad = ratio * PI2;

    // 円周上の位置を得る
    float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
    float x = cos * radius, y = sin * radius;
}
```

Cylinder のモデリングでは、円筒の端に位置する円周に沿って均等に頂点を配置し、それらの頂点をつなぎ合わせて側面を形作ります。側面の 1 つ 1 つは Quad を構築するのと同じように、上端と下端から対応する頂点を 2 つずつ取り出して三角形を向かい合わせて配置し、1 つの側面、つまり四角形を構築します。Cylinder の側面は、Quad が円形に沿って配置されているものだとイメージできます。



▲図 1.8　Cylinder の側面のモデリング　黒丸は端に位置する円周に沿って均等に配置された頂点　頂点内の a〜d は Cylinder.cs プログラム内で三角形を構築する際に頂点に割り振られる index 変数

サンプルプログラム **Cylinder.cs**

まずは側面を構築していきますが、Cylinder クラスでは上端と下端に位置する円周に並べられた頂点のデータを生成するための関数 GenerateCap を用意しています。

▼

```
var vertices = new List<Vector3>();
var normals = new List<Vector3>();
var uvs = new List<Vector2>();
var triangles = new List<int>();

// 上端の高さと、下端の高さ
float top = height * 0.5f, bottom = -height * 0.5f;

// 側面を構成する頂点データを生成
GenerateCap(segments + 1, top, bottom, radius, vertices, uvs, normals, true);

// 側面の三角形を構築する際、円上の頂点を参照するために、
// index が円を一周するための除数
var len = (segments + 1) * 2;

// 上端と下端をつなぎ合わせて側面を構築
for (int i = 0; i < segments + 1; i++) {
    int idx = i * 2;
    int a = idx, b = idx + 1, c = (idx + 2) % len, d = (idx + 3) % len;
```

```
    triangles.Add(a);
    triangles.Add(c);
    triangles.Add(b);

    triangles.Add(d);
    triangles.Add(b);
    triangles.Add(c);
}
```

GenerateCap 関数では、List 型で渡された変数に頂点や法線データを設定します。

▼

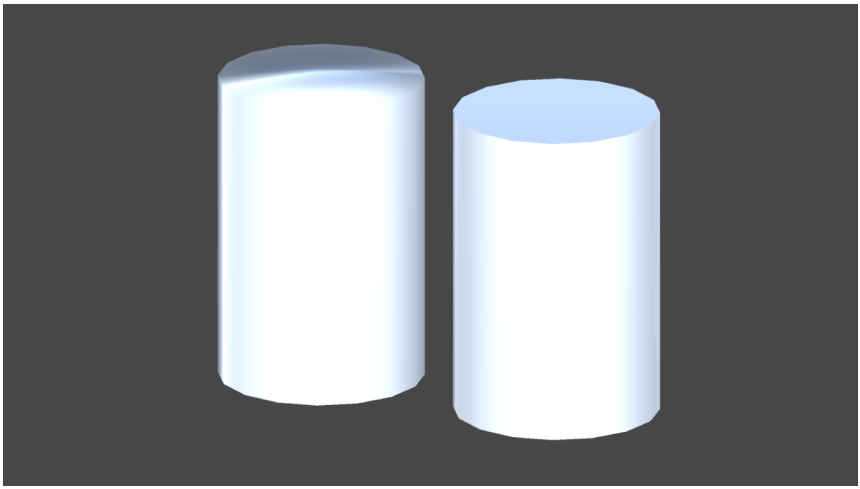
```
void GenerateCap(
    int segments,
    float top,
    float bottom,
    float radius,
    List<Vector3> vertices,
    List<Vector2> uvs,
    List<Vector3> normals,
    bool side
) {
    for (int i = 0; i < segments; i++) {
        // 0.0 ~ 1.0
        float ratio = (float)i / (segments - 1);

        // 0.0 ~ 2 π
        float rad = ratio * PI2;

        // 円周に沿って上端と下端に均等に頂点を配置する
        float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
        float x = cos * radius, z = sin * radius;
        Vector3 tp = new Vector3(x, top, z), bp = new Vector3(x, bottom, z);

        // 上端
        vertices.Add(tp);
        uvs.Add(new Vector2(ratio, 1f));

        // 下端
        vertices.Add(bp);
        uvs.Add(new Vector2(ratio, 0f));

        if(side) {
            // 側面の外側を向く法線
            var normal = new Vector3(cos, 0f, sin);
            normals.Add(normal);
            normals.Add(normal);
        } else {
            normals.Add(new Vector3(0f, 1f, 0f)); // 蓋の上を向く法線
            normals.Add(new Vector3(0f, -1f, 0f)); // 蓋の下を向く法線
        }
    }
}
```

Cylinder クラスでは、上端と下端を閉じたモデルにするかどうかを openEnded フラグで設定することができます。上端と下端を閉じる場合は、円形の「蓋」を形作り、

端に栓をします。

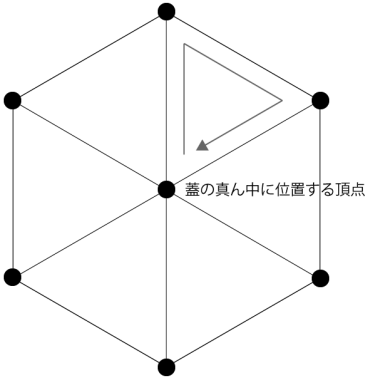
蓋の面を構成する頂点は、側面を構成している頂点を利用せずに、側面と同じ位置に別途新しく頂点を生成します。これは、側面と蓋の部分とで法線を分け、自然なライティングを施すためです。（側面の頂点データを構築する場合は GenerateCap の引数の side 変数に true を、蓋を構築する場合は false を指定し、適切な法線方向が設定されるようにしています。）

もし、側面と蓋の部分で同じ頂点を共有してしまうと、側面と蓋面で同じ法線を参照することになってしまうので、ライティングが不自然になってしまいます。



▲図 1.9　Cylinder の側面と蓋の頂点を共有した場合（左:BadCylinder.cs）と、サンプルプログラムのように別の頂点を用意した場合（右:Cylinder.cs）　左はライティングが不自然になっている

円形の蓋をモデリングするには、（GenerateCap 関数から生成される）円周上に均等に並べられた頂点と、円の真ん中に位置する頂点を用意し、真ん中の頂点から円周に沿った頂点をつなぎ合わせて、均等に分けられたピザのように三角形を構築することで円形の蓋を形作ります。



▲図 1.10　Cylinder の蓋のモデリング　segments パラメータが 6 の場合の例

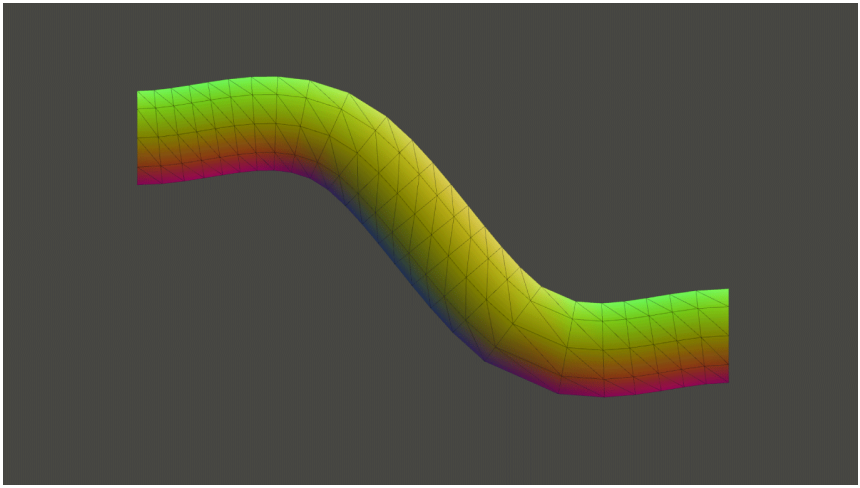
▼

```
// 上端と下端の蓋を生成
if(openEnded) {
    // 蓋のモデルのための頂点は、ライティング時に異なった法線を利用するために、側面と
は共有せずに新しく追加する
    GenerateCap(
        segments + 1,
        top,
        bottom,
        radius,
        vertices,
        uvs,
        normals,
        false
    );

    // 上端の蓋の真ん中の頂点
    vertices.Add(new Vector3(0f, top, 0f));
    uvs.Add(new Vector2(0.5f, 1f));
    normals.Add(new Vector3(0f, 1f, 0f));

    // 下端の蓋の真ん中の頂点
    vertices.Add(new Vector3(0f, bottom, 0f)); // bottom
    uvs.Add(new Vector2(0.5f, 0f));
    normals.Add(new Vector3(0f, -1f, 0f));

    var it = vertices.Count - 2;
    var ib = vertices.Count - 1;
```

```
        // 側面の分の頂点 index を参照しないようにするための offset
        var offset = len;

        // 上端の蓋の面
        for (int i = 0; i < len; i += 2) {
            triangles.Add(it);
            triangles.Add((i + 2) % len + offset);
            triangles.Add(i + offset);
        }

        // 下端の蓋の面
        for (int i = 1; i < len; i += 2) {
            triangles.Add(ib);
            triangles.Add(i + offset);
            triangles.Add((i + 2) % len + offset);
        }
    }
```

1.3.3 Tubular

Tubular は筒型のモデルで、次の図のような形をしています。

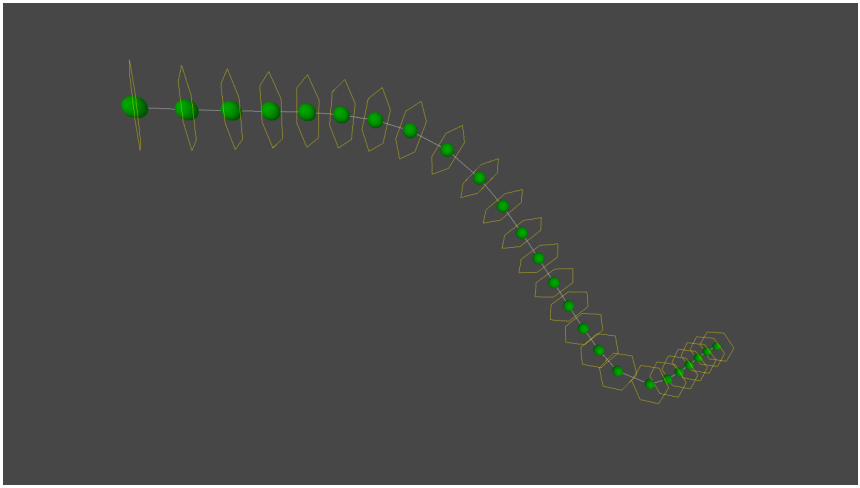


▲図 1.11　Tubular モデル

Cylinder モデルはまっすぐに伸びる円筒形状ですが、Tubular は曲線に沿ったねじれのない筒型をしています。後述する樹木モデルの例では、一本の枝を Tubular で表現し、その組み合わせで一本の木を構築する手法を採用しているのですが、滑らかに曲がる筒型が必要な場面で Tubular は活躍します。

筒型の構造

筒型モデルの構造は次の図のようになっています。



▲図 1.12　筒型の構造　Tubular が沿う曲線を分割する点を球で、側面を構成する節を六角形で可視化している

曲線を分割し、分割点によって区切られた節ごとに側面を構築していき、それらを組み合わせることで 1 つの Tubular モデルを生成します。

1 つ 1 つの節の側面は Cylinder の側面と同じように、側面の上端と下端の頂点を円形に沿って均等に配置し、それらをつなぎ合わせて構築するため、Cylinder を曲線に沿って連結したものが Tubular 型だと考えることができます。

曲線について

サンプルプログラムでは、曲線を表す基底クラス CurveBase を用意しています。3 次元空間上の曲線の描き方については、様々なアルゴリズムが考案されており、用途に応じて使いやすい手法を選択する必要があります。サンプルプログラムでは、CurveBase クラスを継承したクラス CatmullRomCurve を利用しています。

ここでは詳細は割愛しますが、CatmullRomCurve は渡された制御点全てを通るように点と点の間を補間しつつ曲線を形作るという特徴があり、曲線に経由させたい点を指定できるため、使い勝手の良さに定評があります。

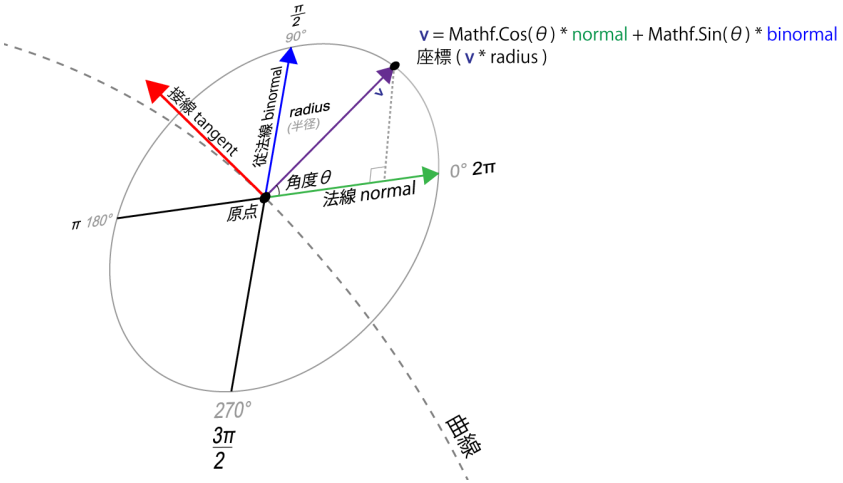
曲線を表す CurveBase クラスでは、曲線上の点の位置と傾き（tangent ベクトル）

を得るために GetPointAt(float)・GetTangentAt(float) 関数を用意しており、引数に [0.0 ~ 1.0] の値を指定することで、始点（0.0）から終点（1.0）の間にある点の位置と傾きを取得できます。

Frenet frame

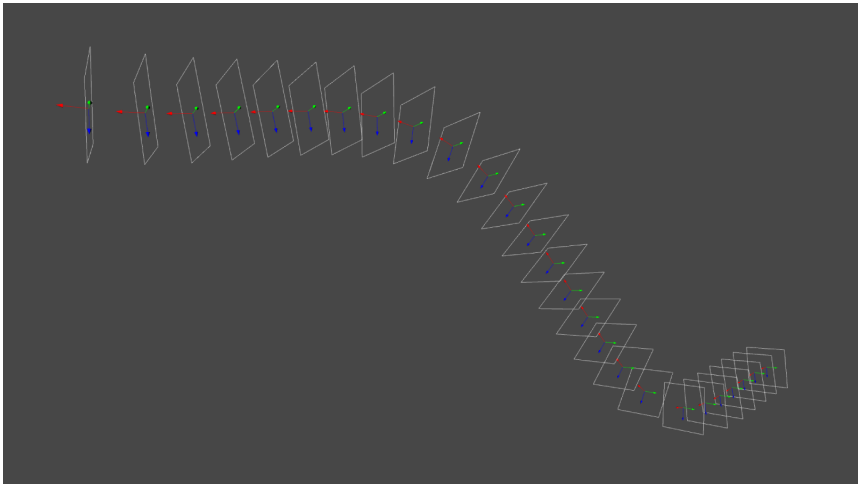
曲線に沿ったねじれのない筒型を作るには、曲線に沿ってなめらかに変化する 3 つの直交するベクトル「接線（tangent）ベクトル、法線（normal）ベクトル、従法線（binormal）ベクトル」の配列が必要となります。接線ベクトルは、曲線上の一点における傾きを表す単位ベクトルのことで、法線ベクトルと従法線ベクトルはお互いに直交するベクトルとして求めます。

これらの直交するベクトルによって、曲線上のある一点において「曲線に直交する円周上の座標」を得ることができます。



▲図 1.13　法線（normal）と従法線（binormal）から、円周上の座標を指す単位ベクトル（v）を求める　この単位ベクトル（v）に半径 radius を乗算することで、曲線に直交する半径 radius の円周上の座標を得ることができる

この曲線上のある一点における 3 つの直交するベクトルの組のことを Frenet frame（フレネフレーム）と呼びます。



▲図 1.14　Tubular を構成する Frenet frame 配列の可視化　枠が 1 つの Frenet frame を表し、3 つの矢印は接線（tangent）ベクトル、法線（normal）ベクトル、従法線（binormal）ベクトルを示している

Tubular のモデリングは、この Frenet frame から得られた法線と従法線を元に節ごとの頂点データを求め、それらをつなぎ合わせていくという手順で行います。

サンプルプログラムでは、CurveBase クラスがこの Frenet frame 配列を生成するための関数 ComputeFrenetFrames を持っています。

サンプルプログラム **Tubular.cs**

Tubular クラスは曲線を表す CatmullRomCurve クラスを持ち、この CatmullRomCurve が描く曲線に沿って筒型を形成します。

CatmullRomCurve クラスは 4 つ以上の制御点が必要で、制御点を操作すると曲線の形状が変化し、それに伴って Tubular モデルの形状も変化していきます。

▼

```
var vertices = new List<Vector3>();
var normals = new List<Vector3>();
var tangents = new List<Vector4>();
var uvs = new List<Vector2>();
var triangles = new List<int>();

// 曲線から Frenet frame を取得
var frames = curve.ComputeFrenetFrames(tubularSegments, closed);
```

```
// Tubular の頂点データを生成
for(int i = 0; i < tubularSegments; i++) {
    GenerateSegment(curve, frames, vertices, normals, tangents, i);
}
// 閉じた筒型を生成する場合は曲線の始点に最後の頂点を配置し、閉じない場合は曲線の終点
に配置する
GenerateSegment(
    curve,
    frames,
    vertices,
    normals,
    tangents,
    (!closed) ? tubularSegments : 0
);

// 曲線の始点から終点に向かって uv 座標を設定していく
for (int i = 0; i <= tubularSegments; i++) {
    for (int j = 0; j <= radialSegments; j++) {
        float u = 1f * j / radialSegments;
        float v = 1f * i / tubularSegments;
        uvs.Add(new Vector2(u, v));
    }
}

// 側面を構築
for (int j = 1; j <= tubularSegments; j++) {
    for (int i = 1; i <= radialSegments; i++) {
        int a = (radialSegments + 1) * (j - 1) + (i - 1);
        int b = (radialSegments + 1) * j + (i - 1);
        int c = (radialSegments + 1) * j + i;
        int d = (radialSegments + 1) * (j - 1) + i;

        triangles.Add(a); triangles.Add(d); triangles.Add(b);
        triangles.Add(b); triangles.Add(d); triangles.Add(c);
    }
}

var mesh = new Mesh();
mesh.vertices = vertices.ToArray();
mesh.normals = normals.ToArray();
mesh.tangents = tangents.ToArray();
mesh.uv = uvs.ToArray();
mesh.triangles = triangles.ToArray();
```

関数 GenerateSegment は先述した Frenet frame から取り出した法線と従法線を元に、指定された節の頂点データを計算し、List 型で渡された変数に設定します。

▼

```
void GenerateSegment(
    CurveBase curve,
    List<FrenetFrame> frames,
    List<Vector3> vertices,
    List<Vector3> normals,
    List<Vector4> tangents,
    int index
```

```
) {
    // 0.0 ~ 1.0
    var u = 1f * index / tubularSegments;

    var p = curve.GetPointAt(u);
    var fr = frames[index];

    var N = fr.Normal;
    var B = fr.Binormal;

    for(int j = 0; j <= radialSegments; j++) {
        // 0.0 ~ 2 π
        float rad = 1f * j / radialSegments * PI2;

        // 円周に沿って均等に頂点を配置する
        float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
        var v = (cos * N + sin * B).normalized;
        vertices.Add(p + radius * v);
        normals.Add(v);

        var tangent = fr.Tangent;
        tangents.Add(new Vector4(tangent.x, tangent.y, tangent.z, 0f));
    }
}
```

1.4 複雑な形状

この節では、これまで説明した ProceduralModeling のテクニックを使って、より複雑なモデルを生成する手法について紹介します。

1.4.1 植物

植物のモデリングは、ProceduralModeling のテクニックの応用例としてよく取り上げられています。Unity 内でも樹木を Editor 内でモデリングするための Tree API[*3]が用意されていますし、Speed Tree[*4]という植物のモデリング専用のソフトが存在します。

この節では、植物の中でも比較的モデリング手法が単純な樹木のモデリングについて取り上げます。

1.4.2 L-System

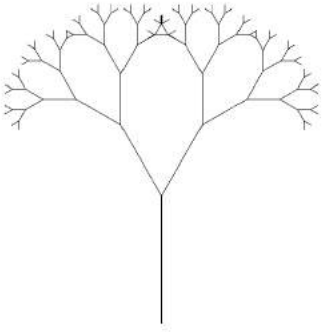
植物の構造を記述・表現できるアルゴリズムとして L-System があります。L-System は植物学者である Aristid Lindenmayer によって 1968 年に提唱されたもの

[*3] https://docs.unity3d.com/ja/540/Manual/tree-FirstTree.html
[*4] http://www.speedtree.com/

で、L-System の L は彼の名前から来ています。

L-System を用いると、植物の形状に見られる自己相似性を表現することができます。

自己相似性とは、物体の細部の形を拡大してみると、大きなスケールで見たその物体の形と一致することで、例えば樹木の枝分かれを観察すると、幹に近い部分の枝の分かれ方と、先端に近い部分の枝の分かれ方に相似性があります。



▲図 1.15　それぞれの枝が 30 度ずつの変化で枝分かれした図形 根元の部分と枝先の部分で相似になっていることがわかるが、このようなシンプルな図形でも樹木のような形に見える（サンプルプログラム LSystem.scene）

L-System は、要素を記号で表し、記号を置き換える規則を定め、記号に対して規則を繰り返し適用していくことで、記号の列を複雑に発展させていくメカニズムを提供します。

例えば簡単な例をあげると、

- 初期文字列:a

を

- 書き換え規則 1: a -> ab
- 書き換え規則 2: b -> a

に従って書き換えていくと、

a -> ab -> aba -> abaab -> abaababa -> ...

という風にステップを経るごとに複雑な結果を生み出します。

この L-System をグラフィック生成に利用した例がサンプルプログラムの LSystem

クラスです。

LSystem クラスでは、以下の操作

- Draw: 向いている方向に向かって線を引きつつ進む
- Turn Left: 左に θ 度回転する
- Turn Right: 右に θ 度回転する

を用意しており、

- 初期操作: Draw

を

- 書き換え規則 1: Draw -> Turn Left | Turn Right
- 書き換え規則 2: Turn Left -> Draw
- 書き換え規則 3: Turn Right -> Draw

に従って、決まられた回数だけ規則の適用を繰り返しています。

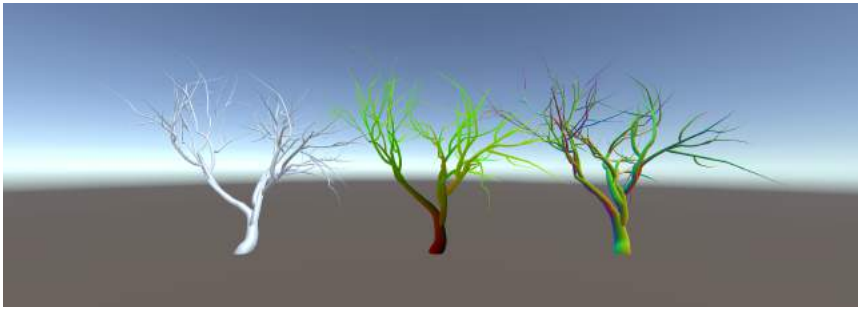
その結果、サンプルの LSystem.scene に示すような、自己相似性を持つ図を描くことができます。この L-System の持つ「状態を再帰的に書き換えていく」という性質が自己相似性を生み出すのです。自己相似性は Fractal（フラクタル）とも呼ばれ、1つの研究分野にもなっています。

1.4.3 サンプルプログラム ProceduralTree.cs

実際に L-System を樹木のモデルを生成するプログラムに応用した例として、ProceduralTree というクラスを用意しました。

ProceduralTree では、前項で解説した LSystem クラスと同様に「枝を進めては分岐し、さらに枝を進める」というルーチンを再帰的に呼び出すことで木の形を生成していきます。

前項の LSystem クラスでは、枝の分岐に関しては「一定角度、左と右の二方向に分岐する」という単純なルールでしたが、ProceduralTree では乱数を用い、分岐する数や分岐方向にランダム性を持たせ、枝が複雑に分岐するようなルールを設定しています。



▲図 1.16　ProceduralTree.scene

TreeData クラス

TreeData クラスは枝の分岐具合を定めるパラメータや、木のサイズ感やモデルのメッシュの細かさを決めるパラメータを内包したクラスです。このクラスのインスタンスのパラメータを調整することで、木の形をデザインすることができます。

枝分かれ

TreeData クラス内のいくつかのパラメータを用いて枝の分かれ具合を調整します。

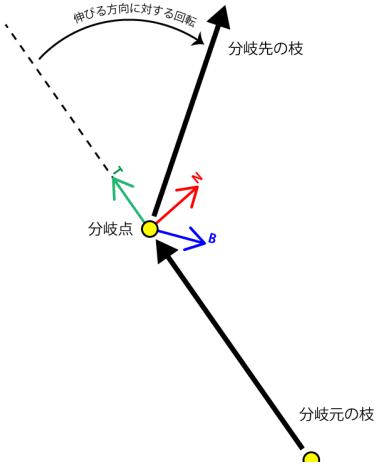
■branchesMin, branchesMax　1 つの枝から分岐する枝の数は branchesMin・branchesMax パラメータで調整します。branchesMin が分岐する枝の最小数、branchesMax が分岐する枝の最大数を表しており、branchesMin から branchesMax の間の数をランダムに選び、分岐する数を決めます。

■growthAngleMin, growthAngleMax, growthAngleScale　分岐する枝が生える方向は growthAngleMin・growthAngleMax パラメータで調整します。growthAngleMin は分岐する方向の最小角度、growthAngleMax が最大角度を表しており、growthAngleMin から growthAngleMax の間の数をランダムに選び、分岐する方向を決めます。

それぞれの枝は伸びる方向を表す tangent ベクトルと、それと直交するベクトルとして normal ベクトルと binormal ベクトルを持ちます。

growthAngleMin・growAngleMax パラメータからランダムに得られた値は、分岐点から伸びる方向の tangent ベクトルに対して、normal ベクトルの方向と binormal ベクトルの方向に回転が加えられます。

分岐点から伸びる方向 tangent ベクトルに対してランダムな回転を加えることで、分岐先の枝が生える方向を変化させ、枝分かれを複雑に変化させます。



▲図 1.17　分岐点から伸びる方向に対してかけられるランダムな回転　分岐点での T の矢印は伸びる方向（tangent ベクトル）、N の矢印は法線（normal ベクトル）、B の矢印は従法線（binormal ベクトル）を表し、伸びる方向に対して法線と従法線の方向にランダムな回転がかけられる

枝が生える方向にランダムにかけられる回転の角度が枝先にいくほど大きくなるように growthAngleScale パラメータを用意しています。この growthAngleScale パラメータは、枝のインスタンスが持つ世代を表す generation パラメータが 0 に近づくほど、つまり枝先に近づくほど、回転する角度に強く影響し、回転の角度を大きくします。

▼

```
// 枝先ほど分岐する角度が大きくなる
var scale = Mathf.Lerp(
    1f,
    data.growthAngleScale,
    1f - 1f * generation / generations
);

// normal 方向の回転
var qn = Quaternion.AngleAxis(scale * data.GetRandomGrowthAngle(), normal);

// binormal 方向の回転
var qb = Quaternion.AngleAxis(scale * data.GetRandomGrowthAngle(), binormal);

// 枝先が向いている tangent 方向に qn * qb の回転をかけつつ、枝先の位置を決める
this.to = from + (qn * qb) * tangent * length;
```

TreeBranch クラス

枝は TreeBranch クラスで表現されます。

世代数（generations）と基本となる長さ（length）と太さ（radius）のパラメータに加えて、分岐パターンを設定するための TreeData を引数に指定してコンストラクタを呼び出すと、内部で再帰的に TreeBranch のインスタンスが生成されていきます。

1 つの TreeBranch から分岐した TreeBranch は、元の TreeBranch 内にある List<TreeBranch>型である children 変数に格納され、根元の TreeBranch から全ての枝に辿れるようにしています。

TreeSegment クラス

一本の枝のモデルは、Tubular 同様、一本の曲線を分割し、分割された節を 1 つの Cylinder としてモデル化し、それらをつなぎ合わせていくように構築していきます。

TreeSegment クラスは一本の曲線を分割する節（Segment）を表現するクラスです。

▼

```
public class TreeSegment {
    public FrenetFrame Frame { get { return frame; } }
    public Vector3 Position { get { return position; } }
    public float Radius { get { return radius; } }

    // TreeSegment が向いている方向ベクトル tangent、
    // それと直交するベクトル normal、binormal を持つ FrenetFrame
    FrenetFrame frame;

    // TreeSegment の位置
    Vector3 position;

    // TreeSegment の幅（半径）
    float radius;

    public TreeSegment(FrenetFrame frame, Vector3 position, float radius) {
        this.frame = frame;
        this.position = position;
        this.radius = radius;
    }
}
```

1 つの TreeSegment は節が向いている方向のベクトルと直交ベクトルがセットになった FrenetFrame、位置と幅を表す変数を持ち、Cylinder を構築する際の上端と下端に必要な情報を保持します。

ProceduralTree モデル生成

ProceduralTree のモデル生成ロジックは Tubular を応用したもので、一本の枝 TreeBranch が持つ TreeSegment の配列から Tubular モデルを生成し、それらを 1

つのモデルに集約することで全体の一本の木を形作る、というアプローチでモデリングしています。

▼

```
var root = new TreeBranch(
    generations,
    length,
    radius,
    data
);

var vertices = new List<Vector3>();
var normals = new List<Vector3>();
var tangents = new List<Vector4>();
var uvs = new List<Vector2>();
var triangles = new List<int>();

// 木の全長を取得
// 枝の長さを全長で割ることで、uv 座標の高さ (uv.y) が
// 根元から枝先に至るまで [0.0 ~ 1.0] で変化するように設定する
float maxLength = TraverseMaxLength(root);

// 再帰的に全ての枝を辿り、1 つ 1 つの枝に対応する Mesh を生成する
Traverse(root, (branch) => {
    var offset = vertices.Count;

    var vOffset = branch.Offset / maxLength;
    var vLength = branch.Length / maxLength;

    // 一本の枝から頂点データを生成する
    for(int i = 0, n = branch.Segments.Count; i < n; i++) {
        var t = 1f * i / (n - 1);
        var v = vOffset + vLength * t;

        var segment = branch.Segments[i];
        var N = segment.Frame.Normal;
        var B = segment.Frame.Binormal;
        for(int j = 0; j <= data.radialSegments; j++) {
            // 0.0 ~ 2 π
            var u = 1f * j / data.radialSegments;
            float rad = u * PI2;

            float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
            var normal = (cos * N + sin * B).normalized;
            vertices.Add(segment.Position + segment.Radius * normal);
            normals.Add(normal);

            var tangent = segment.Frame.Tangent;
            tangents.Add(new Vector4(tangent.x, tangent.y, tangent.z, 0f));

            uvs.Add(new Vector2(u, v));
        }
    }

    // 一本の枝の三角形を構築する
    for (int j = 1; j <= data.heightSegments; j++) {
        for (int i = 1; i <= data.radialSegments; i++) {
```

```
            int a = (data.radialSegments + 1) * (j - 1) + (i - 1);
            int b = (data.radialSegments + 1) * j + (i - 1);
            int c = (data.radialSegments + 1) * j + i;
            int d = (data.radialSegments + 1) * (j - 1) + i;

            a += offset;
            b += offset;
            c += offset;
            d += offset;

            triangles.Add(a); triangles.Add(d); triangles.Add(b);
            triangles.Add(b); triangles.Add(d); triangles.Add(c);
        }
    }
});

var mesh = new Mesh();
mesh.vertices = vertices.ToArray();
mesh.normals = normals.ToArray();
mesh.tangents = tangents.ToArray();
mesh.uv = uvs.ToArray();
mesh.triangles = triangles.ToArray();
mesh.RecalculateBounds();
```

植物のプロシージャルモデリングは樹木だけでも奥深く、日光の照射率が高くなるように枝分かれすることで自然な木のモデルを得るようにする、といった手法などが考案されています。

こうした植物のモデリングに興味がある方は L-System を考案した Aristid Lindenmayer により執筆された The Algorithmic Beauty of Plants[*5]に様々な手法が紹介されていますので、参考にしてみてください。

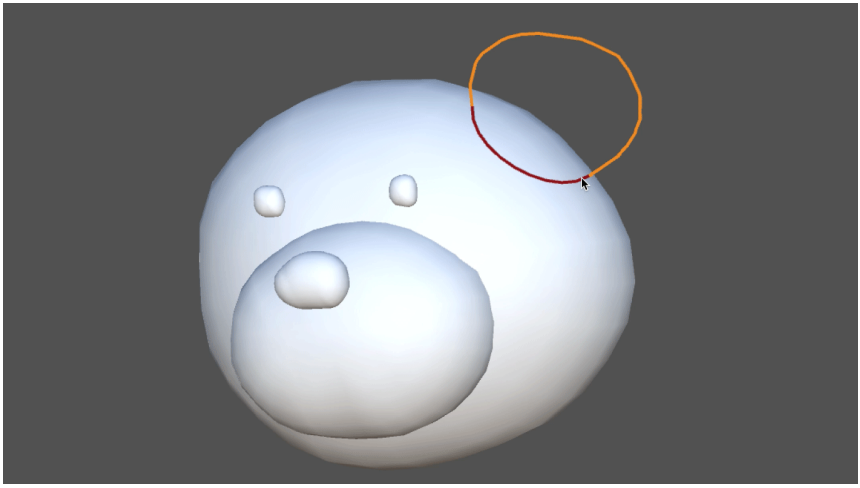
1.5 プロシージャルモデリングの応用例

これまで紹介したプロシージャルモデリングの例から、「モデルをパラメータによって変化させながら動的に生成できる」というテクニックの利点を知ることができました。効率的に様々なバリエーションのモデルを作成できるため、コンテンツ開発の効率化のための技術という印象を受けるかもしれません。

しかし、世の中にあるモデリングツールやスカルプトツールのように、プロシージャルモデリングのテクニックは「ユーザの入力に応じて、インタラクティブにモデルを生成する」という応用も可能です。

応用例として、東京大学大学院情報工学科の五十嵐健夫氏により考案された、手書きスケッチによる輪郭線から立体モデルを生成する技術「Teddy」についてご紹介します。

[*5] http://algorithmicbotany.org/papers/#abop



▲図 1.18　手書きスケッチによる 3 次元モデリングを行う技術「Teddy」の Unity アセット　http://unitеddy.info/ja

2002 年にプレイステーション 2 用のソフトとして発売された「ガラクタ名作劇場 ラクガキ王国」[*6]というゲームでは実際にこの技術が用いられ、「自分の描いた絵を 3D 化してゲーム内のキャラクターとして動かす」という応用が実現されています。

この技術では、

- 2 次元平面上に描かれた線を輪郭として定義する
- 輪郭線を構成する点配列に対してドロネー三角形分割（Delaunay Triangulation）[*7]と呼ばれるメッシュ化処理を施す
- 得られた 2 次元平面上のメッシュに対して、立体に膨らませるアルゴリズムを適用する

という手順で 3 次元モデルを生成しています。アルゴリズムの詳細に関してはコンピュータグラフィクスを扱う国際会議 SIGGRAPH にて発表された論文が公開されています。[*8]

Teddy は Unity に移植されたバージョンが Asset Store に公開されているので、誰でもコンテンツにこの技術を組み込むことができます。[*9]

[*6] https://ja.wikipedia.org/wiki/ラクガキ王国
[*7] https://en.wikipedia.org/wiki/Delaunay_triangulation
[*8] http://www-ui.is.s.u-tokyo.ac.jp/~takeo/papers/siggraph99.pdf
[*9] http://unitеddy.info/ja/

このようにプロシージャルモデリングのテクニックを用いれば、独自のモデリングツールを開発することができ、ユーザの創作によって発展していくようなコンテンツを作ることも可能になります。

1.6 まとめ

プロシージャルモデリングのテクニックを使えば、

- （ある条件下での）モデル生成の効率化
- ユーザの操作に応じてインタラクティブにモデルを生成するツールやコンテンツの開発

が実現できることを見てきました。

Unity 自体はゲームエンジンであるため、本章で紹介した例からはゲームや映像コンテンツ内での応用を想像されるでしょう。

しかし、コンピュータグラフィックスの技術自体の応用範囲が広いように、モデルを生成する技術の応用範囲も広いものだと考えることができます。冒頭でも述べましたが、建築やプロダクトデザインの分野でもプロシージャルモデリングの手法が利用されていますし、3D プリンタ技術などのデジタルファブリケーションの発展にともなって、デザインした形を実生活で利用できる機会が個人レベルでも増えてきています。

このように、どのような分野でデザインした形を利用するかを広い視野で考えると、プロシージャルモデリングのテクニックを応用できる場面が様々なところから見つかるかもしれません。

1.7 参考

- CEDEC2008 コンピュータが知性でコンテンツを自動生成--プロシージャル技術とは - http://news.mynavi.jp/articles/2008/10/08/cedec03/
- The Algorithmic Beauty of Plants - http://algorithmicbotany.org/papers
- nervous system - http://n-e-r-v-o-u-s.com/

第 2 章

ComputeShader 入門

Unity で ComputeShader (以降必要に応じて"コンピュートシェーダ") を使う方法について、シンプルに解説します。コンピュートシェーダとは、GPU を使って単純処理を並列化し、大量の演算を高速に実行するために用いられます。また GPU に処理を委譲しますが、通常のレンダリングパイプラインとは異なることが特徴に挙げられます。CG においては、大量のパーティクルの動きを表現するためなどに良く用いられます。

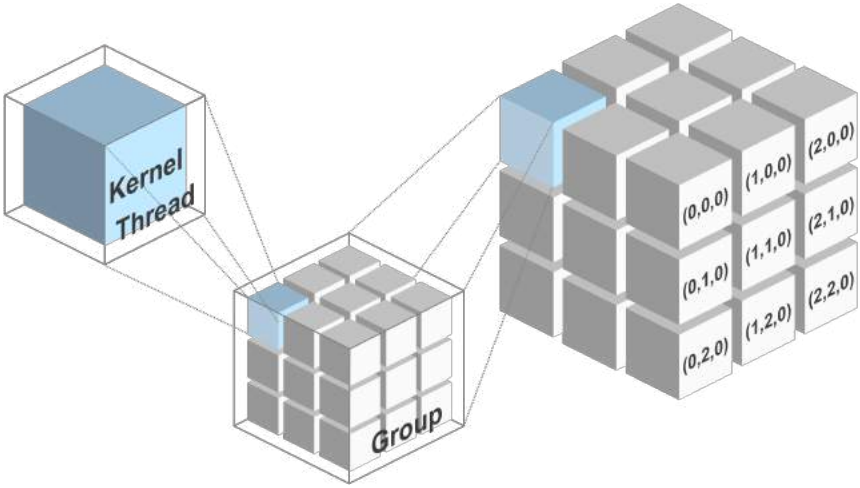
本章の以降に続く内容の一部にも、コンピュートシェーダが用いられたものがあり、それらを読み進める上で、コンピュートシェーダの知識が必要になります。

ここではコンピュートシェーダを学習するにあたって、一番最初の足掛かりになるような内容について、2 つの簡単なサンプルを用いて解説しています。これらはコンピュートシェーダのすべての事について扱うものではありませんので、必要に応じて情報を補うようにしてください。

Unity においては ComputeShader と呼称していますが、類似する技術に OpenCL, DirectCompute, CUDA などが挙げられます。基本概念は類似しており、特に DirectCompute(DirectX) と非常に近い関係にあります。もしアーキテクチャ周辺の概念や更なる詳細情報が必要になるときは、これらについても合わせて情報を集めるようにすると良いと思います。

本章のサンプルは `https://github.com/IndieVisualLab/UnityGraphicsProgramming` の「SimpleComputeShader」です。

2.1 カーネル、スレッド、グループの概念



▲図 2.1　カーネル、スレッド、グループのイメージ

具体的な実装を解説するよりも前に、コンピュートシェーダで取り扱われる カーネル **(Kernel)**、スレッド **(Thread)**、グループ **(Group)** の概念を説明しておく必要があります。

カーネル とは、GPU で実行される 1 つの処理を指し、コード上では 1 つの関数として扱われます (一般的なシステム用語における意味でのカーネルに相当)。

スレッドとは、カーネルを実行する単位です。1 スレッドが、1 カーネルを実行します。コンピュートシェーダではカーネルを複数のスレッドで並行して同時に実行することができます。スレッドは (x, y, z) の 3 次元で指定しす。

例えば、(4, 1, 1) なら 4 * 1 * 1 = 4 つのスレッドが同時に実行されます。(2, 2, 1) なら、2 * 2 * 1 = 4 つのスレッドが同時に実行されます。同じ 4 つのスレッドが実行されますが、状況に応じて、後者のような 2 次元でスレッドを指定する方が効率が良いことがあります。これについては後に続いて解説します。ひとまずスレッド数は 3 次元で指定されるという認識が必要です。

最後にグループとは、スレッドを実行する単位です。また、あるグループが実行するスレッドはグループスレッドと呼ばれます。例えば、あるグループが単位当たり、(4, 1, 1) スレッドを持つとします。このグループが 2 つあるとき、それぞれのグルー

プが、(4, 1, 1) のスレッドを持ちます。

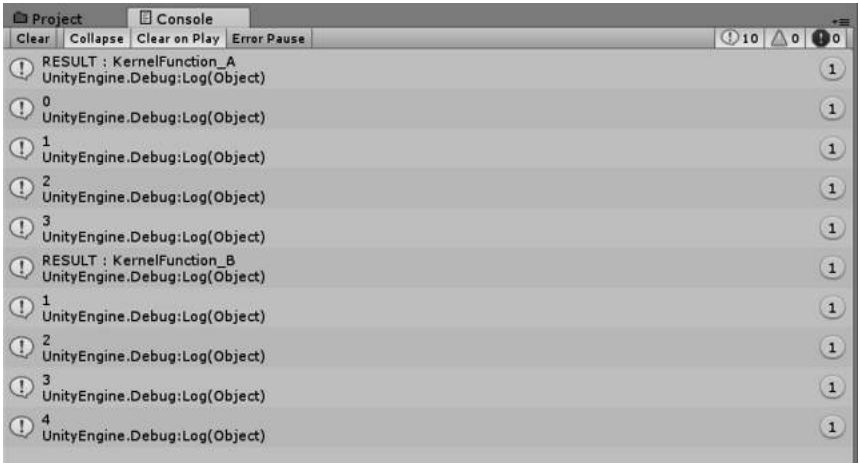
グループもスレッドと同様に 3 次元で指定されます。例えば、(2, 1, 1) グループが、(4, 4, 1) スレッドで実行されるカーネルを実行するとき、グループ数は 2 * 1 * 1 = 2 です。この 2 つのグループは、それぞれ 4 * 4 * 1 = 16 スレッドを持つことになります。したがって、合計スレッド数は、2 * 16 = 32 となります。

2.2 サンプル (1) : GPU で演算した結果を取得する

サンプル (1) 「SampleScene_Array」では、適当な計算をコンピュートシェーダで実行し、その結果を配列として取得する方法について扱います。サンプルには次のような操作が含まれます。

- コンピュートシェーダを使って複数のデータを処理し、その結果を取得する。
- コンピュートシェーダに複数の機能を実装し、使い分ける。
- コンピュートシェーダ (GPU) にスクリプト (CPU) から値を渡す。

サンプル (1) の実行結果は次の通りになります。デバッグ出力だけですから、ソースコードを読みながら動作を確認してください。



▲図 2.2　サンプル (1) の実行結果

2.2.1 コンピュートシェーダの実装

ここからサンプルを実例に解説を進めます。非常に短いので、コンピュートシェーダの実装については先に一通り目を通して頂くのが良いと思います。基本構成として、関数の定義、関数の実装、バッファがあり、必要に応じて変数があります。

▼ SimpleComputeShader_Array.compute

```
#pragma kernel KernelFunction_A
#pragma kernel KernelFunction_B

RWStructuredBuffer<int> intBuffer;
float floatValue;

[numthreads(4, 1, 1)]
void KernelFunction_A(uint3 groupID : SV_GroupID,
                      uint3 groupThreadID : SV_GroupThreadID)
{
    intBuffer[groupThreadID.x] = groupThreadID.x * floatValue;
}

[numthreads(4, 1, 1)]
void KernelFunction_B(uint3 groupID : SV_GroupID,
                      uint3 groupThreadID : SV_GroupThreadID)
{
    intBuffer[groupThreadID.x] += 1;
}
```

特徴として、**numthreads** 属性と、**SV_GroupID** セマンティクスなどがありますが、これについては後述します。

2.2.2 カーネルの定義

先に解説した通り、正確な定義はさておき、カーネルは **GPU** で実行される **1** つの処理を指し、コード上では **1** つの関数として扱われます。カーネルは 1 つのコンピュートシェーダに複数実装することができます。

この例では、カーネルは `KernelFunction_A` ないし `KernelFunction_B` 関数がカーネルに相当します。また、カーネルとして扱う関数は `#pragma kernel` を使って定義します。これによってカーネルとそれ以外の関数と識別します。

定義された複数のカーネルのうち、任意の 1 つを識別するために、固有のインデックスがカーネルに与えられます。インデックスは `#pragma kernel` で定義された順に、上から 0, 1 … と与えられます。

2.2.3 バッファや変数の用意

コンピュートシェーダで実行した結果を保存するバッファ領域を作っておきます。サンプルの変数 RWStructuredBuffer<int> intBuffer} がこれに相当します。

またスクリプト (CPU) 側から任意の値を与えたい場合には、一般的な CPU プログラミングと同じように変数を用意します。この例では変数 intValue がこれに相当し、スクリプトから値を渡します。

2.2.4 numthreads による実行スレッド数の指定

numthreads 属性 (Attribute) は、カーネル (関数) を実行するスレッドの数を指定します。スレッド数の指定は、(x, y, z) で指定し、例えば (4, 1, 1) なら、 4 * 1 * 1 = 4 スレッドでカーネルを実行します。他に、(2, 2, 1) なら 2 * 2 * 1 = 4 スレッドでカーネルを実行します。共に 4 スレッドで実行されますが、この違いや使い分けについては後述します。

2.2.5 カーネル (関数) の引数

カーネルに設定できる引数には制約があり、一般的な CPU プログラミングと比較して自由度は極めて低いです。

引数に続く値を**セマンティクス**と呼び、この例では groupID : SV_GroupID と groupThreadID : SV_GroupThreadID を設定しています。セマンティクスは引数がどのような値であるかを示すための物であり、他の名前に変更することができません。

引数名 (変数名) は自由に定義することができますが、コンピュートシェーダを使うにあたって定義されるセマンティクスのいずれかを設定する必要があります。つまり、任意の型の引数を定義してカーネル内で参照する、といった実装はできず、カーネルで参照することができる引数は、定められた限定的なものから選択する、ということです。

SV_GroupID は、カーネルを実行するスレッドが、どのグループで実行されているかを (x, y, z) で示します。SV_GroupThreadID は、カーネルを実行するスレッドが、グループ内の何番目のスレッドであるかを (x, y, z) で示します。

例えば (4, 4, 1) のグループで、(2, 2, 1) のスレッドを実行するとき、SV_GroupID は (0 ~ 3, 0 ~ 3, 0) の値を返します。SV_GroupThreadID は (0 ~ 1, 0 ~ 1, 0) の値を返します。

サンプルで設定されるセマンティクス以外にも SV_~ から始まるセマンティクスがあり、利用することができますが、ここでは説明を割愛します。一通りコンピュートシェーダの動きが分かった後に目を通すほうが良いと思います。

- SV_GroupID - Microsoft Developer Network
  - https://msdn.microsoft.com/ja-jp/library/ee422449(v=vs.85).aspx
  - 異なる SV~ セマンティクスとその値について確認することができます。

2.2.6 カーネル (関数) の処理内容

サンプルでは、用意したバッファに、順にスレッド番号を代入していく処理を行っています。groupThreadID は、あるグループで実行されるスレッド番号が与えられます。このカーネルは (4, 1, 1) スレッドで実行されますから、groupThreadID は (0 ~ 3, 0, 0) が与えられます。

▼ SimpleComputeShader_Array.compute

```
[numthreads(4, 1, 1)]
void KernelFunction_A(uint3 groupID : SV_GroupID,
                      uint3 groupThreadID : SV_GroupThreadID)
{
    intBuffer[groupThreadID.x] = groupThreadID.x * intValue;
}
```

今回のサンプルはこのスレッドを、(1, 1, 1) のグループで実行します (後述するスクリプトから) 。すなわちグループを 1 つだけ実行し、そのグループには、4 * 1 * 1 のスレッドが含まれます。結果として groupThreadID.x には 0 ~ 3 の値が与えられることを確認してください。

※この例では groupID を利用していませんが、スレッドと同様に、3 次元で指定されるグループ数が与えられます。代入してみるなどして、コンピュートシェーダの動きを確認するために使ってみてください。

2.2.7 スクリプトからコンピュートシェーダを実行する

実装したコンピュートシェーダをスクリプトから実行します。スクリプト側で必要になるものは概ね次の通りです。

- コンピュートシェーダへの参照 | comuteShader
- 実行するカーネルのインデックス | kernelIndex_KernelFunction_A, B
- コンピュートシェーダの実行結果を保存するバッファ | intComputeBuffer

▼ SimpleComputeShader_Array.cs

```
public ComputeShader computeShader;
int kernelIndex_KernelFunction_A;
int kernelIndex_KernelFunction_B;
ComputeBuffer intComputeBuffer;

void Start()
{
    this.kernelIndex_KernelFunction_A
        = this.computeShader.FindKernel("KernelFunction_A");
    this.kernelIndex_KernelFunction_B
        = this.computeShader.FindKernel("KernelFunction_B");

    this.intComputeBuffer = new ComputeBuffer(4, sizeof(int));
    this.computeShader.SetBuffer
        (this.kernelIndex_KernelFunction_A,
         "intBuffer", this.intComputeBuffer);

    this.computeShader.SetInt("intValue", 1);
    …
```

2.2.8 実行するカーネルのインデックスを取得する

あるカーネルを実行するためには、そのカーネルを指定するためのインデックス情報が必要です。インデックスは `#pragma kernel` で定義された順に、上から 0, 1 … と与えられますが、スクリプト側から `FindKernel` 関数を使うのが良いでしょう。

▼ SimpleComputeShader_Array.cs

```
this.kernelIndex_KernelFunction_A
    = this.computeShader.FindKernel("KernelFunction_A");

this.kernelIndex_KernelFunction_B
    = this.computeShader.FindKernel("KernelFunction_B");
```

2.2.9 演算結果を保存するバッファの生成する

コンピュートシェーダ (GPU) による演算結果を CPU 側に保存するためのバッファ領域を用意します。Unity　では `ComputeBuffer` として定義されています。

▼ SimpleComputeShader_Array.cs

```
this.intComputeBuffer = new ComputeBuffer(4, sizeof(int));
this.computeShader.SetBuffer
    (this.kernelIndex_KernelFunction_A, "intBuffer", this.intComputeBuffer);
```

`ComputeBuffer` を (1) 保存する領域のサイズ、(2) 保存するデータの単位当たりのサイズを指定して初期化します。ここでは int 型のサイズ 4 つ分の領域が用意され

ています。これはコンピュートシェーダの実行結果が int[4] として保存されるためです。必要に応じてサイズを変更します。

次いで、コンピュートシェーダに実装された、(1) どのカーネルが実行するときに、(2) どの GPU 上のバッファを使うのかを指定し、(3) CPU 上のどのバッファに相当するのか、を指定します。

この例では、(1) `KernelFunction_A` が実行されるときに参照される、(2) `intBuffer` なるバッファ領域は、(3) `intComputeBuffer` に相当する、と指定されます。

2.2.10 スクリプトからコンピュートシェーダに値を渡す

▼ SimpleComputeShader_Array.cs

```
this.computeShader.SetInt("intValue", 1);
```

処理したい内容によってはスクリプト (CPU) 側からコンピュートシェーダ (GPU) 側に値を渡し、参照したい場合があると思います。ほとんどの型の値は `ComputeShader.Set~` を使って、コンピュートシェーダ内にある変数に設定することができます。このとき、引数に設定する引数の変数名と、コンピュートシェーダ内に定義された変数名は一致する必要があります。この例では `intValue` に 1 を渡しています。

2.2.11 コンピュートシェーダの実行

コンピュートシェーダに実装 (定義) されたカーネルは、`ComputeShader.Dispatch` メソッドで実行します。指定したインデックスのカーネルを、指定したグループ数で実行します。グループ数は X * Y * Z で指定します。このサンプルでは 1 * 1 * 1 = 1 グループです。

▼ SimpleComputeShader_Array.cs

```
this.computeShader.Dispatch
    (this.kernelIndex_KernelFunction_A, 1, 1, 1);

int[] result = new int[4];

this.intComputeBuffer.GetData(result);

for (int i = 0; i < 4; i++)
{
    Debug.Log(result[i]);
}
```

コンピュートシェーダ (カーネル) の実行結果は、`ComputeBuffer.GetData` で取得します。

2.2.12 実行結果の確認 (A)

あらためてコンピュートシェーダ側の実装を確認します。このサンプルでは次のカーネルを 1 * 1 * 1 = 1 グループで実行しています。スレッドは、4 * 1 * 1 = 4 スレッドです。また intValue にはスクリプトから 1 を与えています。

▼ SimpleComputeShader_Array.compute

```
[numthreads(4, 1, 1)]
void KernelFunction_A(uint3 groupID : SV_GroupID,
                      uint3 groupThreadID : SV_GroupThreadID)
{
    intBuffer[groupThreadID.x] = groupThreadID.x * intValue;
}
```

groupThreadID(SV_GroupThreadID) は、今このカーネルが、グループ内の何番目のスレッドで実行されているかを示す値が入るので、この例では $(0 \sim 3, 0, 0)$ が入ります。したがって、groupThreadID.x は $0 \sim 3$ です。つまり、intBuffer[0] = 0 ～ intBuffer[3] = 3 までが並列して実行されることになります。

2.2.13 異なるカーネル (B) を実行する

1 つのコンピュートシェーダに実装した異なるカーネルを実行するときは、別のカーネルのインデックスを指定します。この例では、KernelFunction_A を実行した後に KernelFunction_B を実行します。さらに KernelFunction_A で利用したバッファ領域を、KernelFunction_B でも使っています。

▼ SimpleComputeShader_Array.cs

```
this.computeShader.SetBuffer
(this.kernelIndex_KernelFunction_B, "intBuffer", this.intComputeBuffer);

this.computeShader.Dispatch(this.kernelIndex_KernelFunction_B, 1, 1, 1);

this.intComputeBuffer.GetData(result);

for (int i = 0; i < 4; i++)
{
    Debug.Log(result[i]);
}
```

2.2.14 実行結果の確認 (B)

KernelFunction_B は次のようなコードを実行します。このとき intBuffer は KernelFunction_A で使ったものを引き続き指定している点に注意してください。

▼ SimpleComputeShader_Array.compute

```
RWStructuredBuffer<int> intBuffer;

[numthreads(4, 1, 1)]
void KernelFunction_B
(uint3 groupID : SV_GroupID, uint3 groupThreadID : SV_GroupThreadID)
{
    intBuffer[groupThreadID.x] += 1;
}
```

このサンプルでは、KernelFunction_A によって intBuffer に 0 ~ 3 が順に与えられています。したがって KernelFunction_B を実行した後は、値が 1 ~ 4 になることを確認します。

2.2.15 バッファの破棄

利用し終えた ComputeBuffer は、明示的に破棄する必要があります。

▼ SimpleComputeShader_Array.cs

```
this.intComputeBuffer.Release();
```

2.2.16 サンプル (1) で解決していない問題

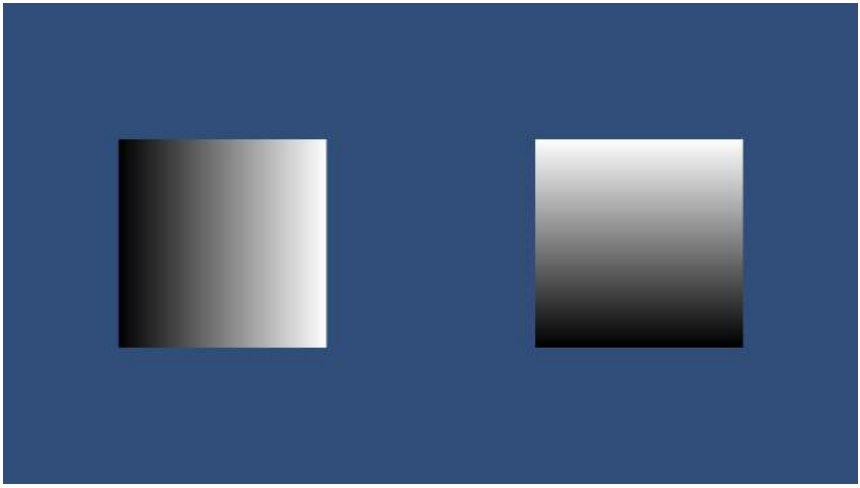
多次元のスレッドまたはグループを指定する意図について、このサンプルでは解説していません。例えば、 (4, 1, 1) スレッドと、(2, 2, 1) スレッドは、どちらも 4 スレッド実行されますが、この 2 つは使い分ける意味があります。これについては後に続くサンプル (2) で解説します。

2.3 サンプル (2) : GPU の演算結果をテクスチャにする

サンプル (2) 「SampleScene_Texture」では、コンピュートシェーダの算出結果をテクスチャにして取得します。サンプルには次のような操作が含まれます。

- コンピュートシェーダを使って、テクスチャに情報を書き込む。
- 多次元 (2 次元) のスレッドを有効に活用する。

サンプル (2) の実行結果は次の通りになります。横方向と縦方向にグラデーションするテクスチャを生成します。



▲図 2.3　サンプル (2) の実行結果

2.3.1　カーネルの実装

全体の実装についてはサンプルを参照してください。このサンプルでは概ね次のようなコードをコンピュートシェーダで実行します。カーネルが多次元スレッドで実行される点に注目してください。(8, 8, 1) なので、1 グループあたり、$8 * 8 * 1 = 64$ スレッドで実行されます。また演算結果の保存先が `RWTexture2D<float4>` であることも大きな変更点です。

▼ SimpleComputeShader_Texture.compute

```
RWTexture2D<float4> textureBuffer;

[numthreads(8, 8, 1)]
void KernelFunction_A(uint3 dispatchThreadID : SV_DispatchThreadID)
{
    float width, height;
    textureBuffer.GetDimensions(width, height);

    textureBuffer[dispatchThreadID.xy]
        = float4(dispatchThreadID.x / width,
                 dispatchThreadID.x / width,
                 dispatchThreadID.x / width,
                 1);
}
```

2.3.2 特殊な引数 SV_DispatchThreadID

サンプル (1) では SV_DispatchThradID セマンティクスは使いませんでした。少々複雑ですが、「あるカーネルを実行するスレッドが、すべてのスレッドの中のどこに位置するか **(x,y,z)** 」を示しています。

SV_DispathThreadID は、SV_Group_ID * numthreads + SV_GroupThreadID で算出される値です。SV_Group_ID はあるグループを (x, y, z) で示し、SV_GroupThreadID は、あるグループに含まれるスレッドを (x, y, z) で示します。

具体的な計算例 (1)

例えば、(2, 2, 1) グループで、(4, 1, 1) スレッドで実行される、カーネルを実行するとします。その内の 1 つのカーネルは、(0, 1, 0) 番目のグループに含まれる、(2, 0, 0) 番目のスレッドで実行されます。このとき SV_DispatchThreadID は、(0, 1, 0) * (4, 1, 1) + (2, 0, 0) = (0, 1, 0) + (2, 0, 0) = (2, 1, 0) になります。

具体的な計算例 (2)

今度は最大値を考えましょう。(2, 2, 1) グループで、(4, 1, 1) スレッドでカーネルが実行されるとき、(1, 1, 0) 番目のグループに含まれる、(3, 0, 0) 番目のスレッドが最後のスレッドです。このとき SV_DispatchThreadID は、(1, 1, 0) * (4, 1, 1) + (3, 0, 0) = (4, 1, 0) + (3, 0, 0) = (7, 1, 0) になります。

2.3.3 テクスチャ (ピクセル) に値を書き込む

以降は時系列順に解説するのが困難ですので、サンプル全体に目を通しながら確認してください。

サンプル (2) の dispatchThreadID.xy は、テクスチャ上にあるすべてのピクセルを示すように、グループとスレッドを設定しています。グループを設定するのはスクリプト側なので、スクリプトとコンピュートシェーダを横断して確認する必要があります。

▼ SimpleComputeShader_Texture.compute

```
textureBuffer[dispatchThreadID.xy]
    = float4(dispatchThreadID.x / width,
             dispatchThreadID.x / width,
             dispatchThreadID.x / width,
             1);
```

このサンプルでは仮に 512x512 のテクスチャを用意していますが、dispatchThreadID.x が 0 ~ 511 を示すとき、dispatchThreadID / width は、0 ~ 0.998… を

示します。つまり dispatchThreadID.xy の値 (= ピクセル座標) が大きくなるにつれて、黒から白に塗りつぶしていくことになります。

テクスチャは、RGBA チャネルから構成され、それぞれ 0 ~ 1 で設定します。すべて 0 のとき、完全に黒くなり、すべて 1 のとき、完全に白くなります。

2.3.4 テクスチャの用意

以降がスクリプト側の実装の解説です。サンプル (1) では、コンピュートシェーダの計算結果を保存するために配列のバッファを用意しました。サンプル (2) では、その代わりにテクスチャを用意します。

▼ SimpleComputeShader_Texture.cs

```
RenderTexture renderTexture_A;
…
void Start()
{
    this.renderTexture_A = new RenderTexture
        (512, 512, 0, RenderTextureFormat.ARGB32);
    this.renderTexture_A.enableRandomWrite = true;
    this.renderTexture_A.Create();
…
```

解像度とフォーマットを指定して RenderTexture を初期化します。このとき RenderTexture.enableRandomWrite を有効にして、テクスチャへの書き込みを有効にしている点に注意します。

- RenderTexture.enableRandomWrite - Unity
  - https://docs.unity3d.com/ScriptReference/RenderTexture-enableRandomWrite.html

2.3.5 スレッド数の取得

カーネルのインデックスが取得できるように、カーネルがどれくらいのスレッド数で実行できるかも取得できます (スレッドサイズ)。

▼ SimpleComputeShader_Texture.cs

```
void Start()
{
…
    uint threadSizeX, threadSizeY, threadSizeZ;

    this.computeShader.GetKernelThreadGroupSizes
     (this.kernelIndex_KernelFunction_A,
      out threadSizeX, out threadSizeY, out threadSizeZ);
…
```

2.3.6 カーネルの実行

Dispath メソッドで処理を実行します。このとき、グループ数の指定方法に注意します。この例では、グループ数は「テクスチャの水平 (垂直) 方向の解像度 / 水平 (垂直) 方向のスレッド数」で算出しています。

水平方向について考えるとき、テクスチャの解像度は 512、スレッド数は 8 ですから、水平方向のグループ数は 512 / 8 = 64 になります。同様に垂直方向も 64 です。したがって、合計グループ数は 64 * 64 = 4096 になります。

▼ SimpleComputeShader_Texture.cs

```
void Update()
{
    this.computeShader.Dispatch
    (this.kernelIndex_KernelFunction_A,
     this.renderTexture_A.width  / this.kernelThreadSize_KernelFunction_A.x,
     this.renderTexture_A.height / this.kernelThreadSize_KernelFunction_A.y,
     this.kernelThreadSize_KernelFunction_A.z);

    plane_A.GetComponent<Renderer>()
        .material.mainTexture = this.renderTexture_A;
```

言い換えれば、各グループは 8 * 8 * 1 = 64 (= スレッド数) ピクセルずつ処理することになります。グループは 4096 あるので、4096 * 64 = 262,144 ピクセル処理します。画像は、512 * 512 = 262,144 ピクセルなので、ちょうどすべてのピクセルを並列に処理できたことになります。

異なるカーネルの実行

もう一方のカーネルは、x ではなく、 y 座標を使って塗りつぶしていきます。このとき 0 に近い値、黒い色が下のほうに表れている点に注意します。テクスチャを操作するときは原点を考慮しなければならないこともあります。

2.3.7 多次元スレッド、グループの利点

サンプル (2) のように、多次元の結果が必要な場合、あるいは多次元の演算が必要な場合には、多次元のスレッドやグループが有効に働きます。もしサンプル (2) を 1 次元のスレッドで処理しようとすると、縦方向のピクセル座標を任意に算出する必要があります。

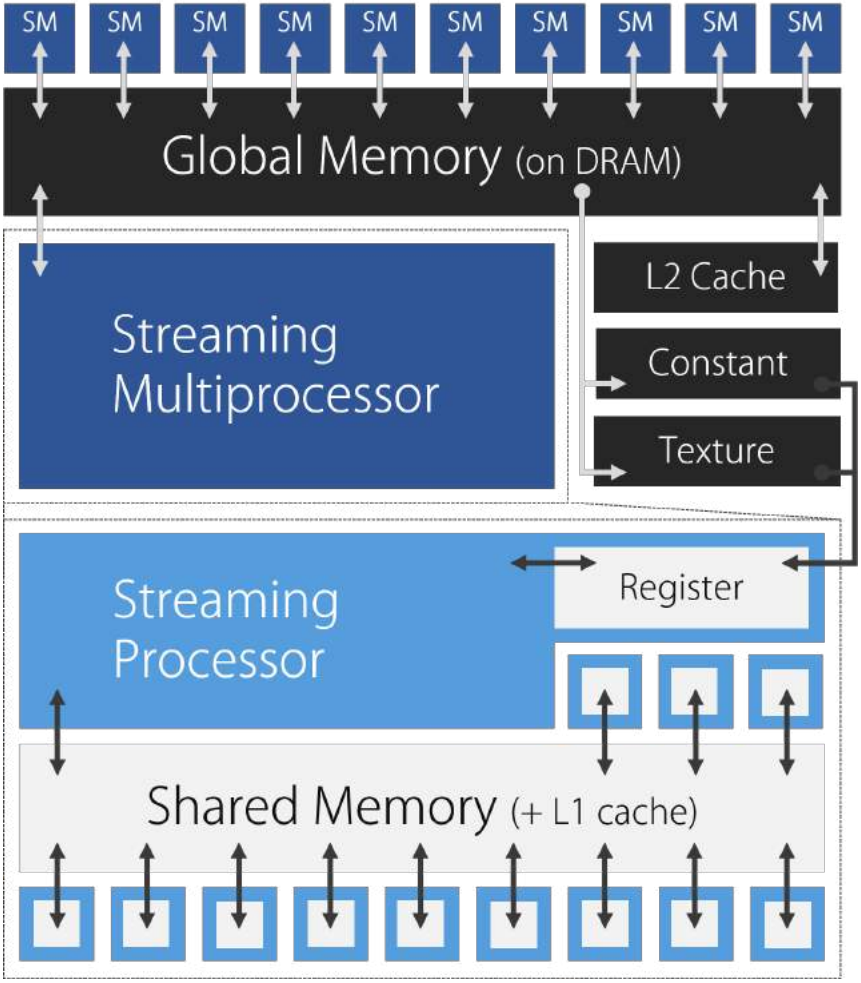
> 実際に実装しようとすると確認できますが、画像処理でいうところのストライド、例えば 512x512 の画像があるとき、その 513 番目のピクセルは、(0, 1) 座標になる、といった算出が必要になります。

演算数は削減したほうが良いですし、高度な処理を行うにしたがって複雑さは増します。コンピュートシェーダを使った処理を設計するときは、上手く多次元を活用できないか検討するのが良いです。

2.4 さらなる学習のための補足情報

本章ではコンピュートシェーダについてサンプルを解説する形式で入門情報としましたが、これから先、学習を進める上で必要ないくつかの情報を補足します。

2.4.1 GPU アーキテクチャ・基本構造



▲図 2.4　GPU アーキテクチャのイメージ

GPU のアーキテクチャ・構造についての基本的な知識があれば、コンピュートシェーダを使った処理の実装の際、それを最適化するために役に立つので、少しだけ

ここで紹介します。

GPU は多数の **Streaming Multiprocessor(SM)** が搭載されていて、それらが分担・並列化して与えられた処理を実行します。

SM には更に小さな **Streaming Processor(SP)** が複数搭載されていて、SM に割り当てられた処理を SP が計算する、といった形式です。

SM にはレジスタとシェアードメモリが搭載されていて、グローバルメモリ **(DRAM** 上のメモリ**)** よりも高速に読み書きすることができます。レジスタは関数内でのみ参照されるローカル変数に使われ、シェアードメモリは同一 SM 内に所属するすべての SP から参照し書き込むことができます。

つまり、各メモリの最大サイズやスコープを把握し、無駄なく高速にメモリを読み書きできる最適な実装を実現できるのが理想です。

例えば最も考慮する必要があるであろうシェアードメモリは、クラス修飾子 (storage-class modifiers) `groupshared` を使って定義します。ここでは入門なので具体的な導入例を割愛しますが、最適化に必要な技術・用語として覚えておいて、以降の学習に役立ててください。

- Variable Syntax - Microsoft Developer Network
  - `https://msdn.microsoft.com/en-us/library/bb509706(v=vs.85).aspx`

レジスタ

SP に最も近い位置に置かれ、最も高速にアクセスできるメモリ領域です。4 byte 単位で構成され、カーネル (関数) スコープの変数が配置されます。スレッドごとに独立するため共有することができません。

シェアードメモリ

SM に置かれるメモリ領域で、L1 キャッシュと合わせて管理されています。同じ SM 内にある SP(= スレッド) で共有することができ、かつ十分に高速にアクセスすることができます。

グローバルメモリ

GPU 上ではなく DRAM 上のメモリ領域です。GPU 上にのプロセッサからは離れた位置にあるため参照は低速です。一方で、容量が大きく、すべてのスレッドからデータの読み書きが可能です。

ローカルメモリ
GPU 上ではなく DRAM 上のメモリ領域で、レジスタに収まらないデータを格納します。GPU 上のプロセッサからは離れた位置にあるため参照は低速です。

テクスチャメモリ
テクスチャデータ専用のメモリで、グローバルメモリをテクスチャ専用に扱います。

コンスタントメモリ
読み込み専用のメモリで、カーネル (関数) の引数や定数を保存しておくためなどに使われます。専用のキャッシュを持っていて、グローバルメモリよりも高速に参照できます。

2.4.2 効率の良いスレッド数指定のヒント

総スレッド数が実際に処理したいデータ数よりも大きい場合は、無意味に実行される (あるいは処理されない) スレッドが生じることになり非効率です。総スレッド数は可能な限り処理したいデータ数と一致させるように設計します。

2.4.3 現行スペック上の限界

執筆時時点での現行スペックの上限を紹介します。最新版でない可能性があることに十分に注意してください。ただし、これらのような制限を考慮しつつ実装することが求められます。

- Compute Shader Overview - Microsoft Developer Network
  - `https://msdn.microsoft.com/en-us/library/ff476331(v=vs.85).aspx`

スレッドとグループ数
スレッド数やグループ数の限界については、解説中に言及しませんでした。これはシェーダモデル (バージョン) によって変更されるためです。今後も並列できる数は増えていくものと思われます。

- ShaderModel cs_4_x
  - Z の最大値が 1
  - X * Y * Z の最大値が 768
- ShaderModel cs_5_0

- Z の最大値が 64
- X * Y * Z の最大値は 1024

またグループの限界は (x, y, z) でそれぞれ 65535 です。

メモリ領域

シェアードメモリの上限は、単位グループあたり 16 KB, あるスレッドが書き込めるシェアードメモリのサイズは、単位あたり 256 byte までと制限されています。

2.5 参考

本章でのその他の参考は以下の通りです。

- 第 5 回　GPU の構造 - 日本 GPU コンピューティングパートナーシップ - http://www.gdep.jp/page/view/252
- Windows で始める CUDA 入門 - エヌビディアジャパン - http://on-demand.gputechconf.com/gtc/2013/jp/sessions/8001.pdf

第 3 章

群のシミュレーションの GPU 実装

3.1 はじめに

この章では、ComputeShader を使った Boids アルゴリズムを用いた群のシミュレーションの実装について解説いたします。鳥や魚、その他の陸上動物は時として群を作ります。この群の動きには規則性と複雑性が見られ、ある種の美しさを持っており人を惹きつけてきました。コンピューターグラフィックスにおいては、それらの個体の振る舞いを一つ一つ人の手で制御することは現実的でなく、Boids と呼ばれる群を作るためのアルゴリズムが考案されました。このシミュレーションアルゴリズムは、いくつかのシンプルな規則で構成されており実装も容易ですが、単純な実装では、すべての個体との位置関係を調べる必要があり、個体数が増えると、その 2 乗に比例して計算量が増加してしまいます。多くの個体を制御したいという場合、CPU による実装では非常に困難です。そこで、GPU による強力な並列計算能力を利用します。Unity には、GPU によるこのような汎用的な計算（GPGPU）を行うため、ComputeShader というシェーダプログラムが用意されています。GPU には共有メモリと呼ばれる特殊な記憶領域が組み込まれており、ComputeShader を用いると、このメモリを有効に活用することができます。また、Unity には GPU インスタンシングという高度なレンダリング機能があり、任意のメッシュを大量に描画することが可能です。これらの Unity の GPU の計算能力を生かした機能を使い、多数の Boid オブジェクトを制御し描画するプログラムを紹介いたします。

3.2 Boids のアルゴリズム

Boids と呼ばれる群のシミュレーションアルゴリズムは、Craig Reynolds によって 1986 年に開発され、翌年 1987 年の ACM SIGGRAPH に「Flocks, Herds, and

Schools: A Distributed Behavioral Model」というタイトルの論文として発表されました。

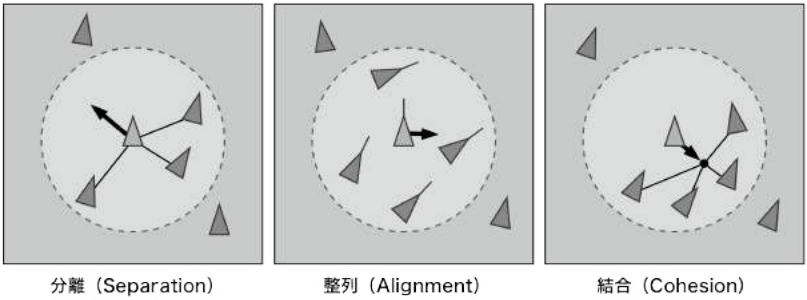
Reynolds は、群れというものは、それぞれの個体が視覚や聴覚などの知覚によって、周囲の他の個体の位置や動く方向に基づいて自身の行動を修正することにより、結果として複雑な振る舞いを生み出している、ということに着目します。

それぞれの個体は以下の 3 つのシンプルな行動規則に従います。

■1. 分離（Separation） ある一定の距離内にある個体と密集することを避けるように動く

■2. 整列（Alignment） ある一定の距離内にある個体が向いている方向の平均に向かおうと動く

■3. 結合（Cohesion） ある一定の距離内にある個体の平均位置に動く



▲図 3.1　Boids の基本的なルール

これらのルールに従って、個々の動きを制御することにより、群れの動きをプログラムすることができます。

3.3　サンプルプログラム

3.3.1　リポジトリ

https://github.com/IndieVisualLab/UnityGraphicsProgramming

本書のサンプル Unity プロジェクトにある、Assets/**BoidsSimulationOnGPU** フォルダ内の **BoidsSimulationOnGPU.unity** シーンデータを開いてください。

3.3.2 実行条件

本章で紹介するプログラムは、ComputeShader、GPU インスタンシングを使用しています。

ComputeShader は、以下のプラットフォームまたは API で動作します。

- DirectX11、または DirectX12 グラフィックス API およびシェーダモデル 5.0GPU を搭載した Windows および Windows ストアアプリ
- MacOS と Metal グラフィックス API を使用した iOS
- Vulkan API を搭載した Android、Linux、Windows プラットフォーム
- 最新の OpenGL プラットフォーム（Linux または Windows では OpenGL 4.3、Android では OpenGL ES 3.1）。（MacOSX は OpenGL4.3 をサポートしていないので注意してください）
- 現段階で一般的に使用されているコンソール機（Sony PS4、Microsoft Xbox One）

GPU インスタンシングは以下のプラットフォームまたは API で利用可能です。

- Windows 上の DirectX 11 および DirectX 12
- Windows、MacOS、Linux、iOS、Android 上の OpenGL コア 4.1 + / ES3.0 +
- MacOS と iOS 上の Metal
- Windows と Android の Vulkan
- プレイステーション 4 と Xbox One
- WebGL（WebGL 2.0 API が必要）

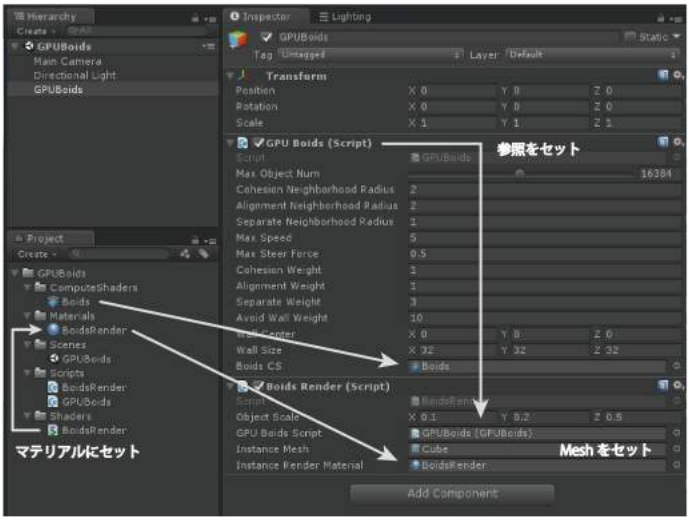
本サンプルプログラムでは、Graphics.DrawMeshInstacedIndirect メソッドを使用しています。そのため、Unity のバージョンは 5.6 以降である必要があります。

3.4 実装コードの解説

本サンプルプログラムは以下のコードで構成されます。

- GPUBoids.cs - Boids のシミュレーションを行う ComputeShader を制御するスクリプト
- Boids.compute - Boids のシミュレーションを行う ComputeShader
- BoidsRender.cs - Boids を描画するシェーダを制御する C#スクリプト
- BoidsRender.shader - GPU インスタンシングによってオブジェクトを描画するためのシェーダ

スクリプトやマテリアルリソースなどはこのようにセットします



▲図 3.2　UnityEditor 上での設定

3.4.1　GPUBoids.cs

このコードでは、Boids シミュレーションのパラメータや、GPU 上での計算のために必要なバッファや計算命令を記述した ComputeShader の管理などを行います。

▼ GPUBoids.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.InteropServices;

public class GPUBoids : MonoBehaviour
{
    // Boid データの構造体
    [System.Serializable]
    struct BoidData
    {
        public Vector3 Velocity; // 速度
        public Vector3 Position; // 位置
    }
```

```
// スレッドグループのスレッドのサイズ
const int SIMULATION_BLOCK_SIZE = 256;

#region Boids Parameters
// 最大オブジェクト数
[Range(256, 32768)]
public int MaxObjectNum = 16384;

// 結合を適用する他の個体との半径
public float CohesionNeighborhoodRadius  = 2.0f;
// 整列を適用する他の個体との半径
public float AlignmentNeighborhoodRadius = 2.0f;
// 分離を適用する他の個体との半径
public float SeparateNeighborhoodRadius  = 1.0f;

// 速度の最大値
public float MaxSpeed        = 5.0f;
// 操舵力の最大値
public float MaxSteerForce   = 0.5f;

// 結合する力の重み
public float CohesionWeight  = 1.0f;
// 整列する力の重み
public float AlignmentWeight = 1.0f;
// 分離する力の重み
public float SeparateWeight  = 3.0f;

// 壁を避ける力の重み
public float AvoidWallWeight = 10.0f;

// 壁の中心座標
public Vector3 WallCenter = Vector3.zero;
// 壁のサイズ
public Vector3 WallSize = new Vector3(32.0f, 32.0f, 32.0f);
#endregion

#region Built-in Resources
// Boids シミュレーションを行う ComputeShader の参照
public ComputeShader BoidsCS;
#endregion

#region Private Resources
// Boid の操舵力（Force）を格納したバッファ
ComputeBuffer _boidForceBuffer;
// Boid の基本データ（速度，位置）を格納したバッファ
ComputeBuffer _boidDataBuffer;
#endregion

#region Accessors
// Boid の基本データを格納したバッファを取得
public ComputeBuffer GetBoidDataBuffer()
{
    return this._boidDataBuffer != null ? this._boidDataBuffer : null;
}

// オブジェクト数を取得
public int GetMaxObjectNum()
{
```

```
    return this.MaxObjectNum;
}

// シミュレーション領域の中心座標を返す
public Vector3 GetSimulationAreaCenter()
{
    return this.WallCenter;
}

// シミュレーション領域のボックスのサイズを返す
public Vector3 GetSimulationAreaSize()
{
    return this.WallSize;
}
#endregion

#region MonoBehaviour Functions
void Start()
{
    // バッファを初期化
    InitBuffer();
}

void Update()
{
    // シミュレーション
    Simulation();
}

void OnDestroy()
{
    // バッファを破棄
    ReleaseBuffer();
}

void OnDrawGizmos()
{
    // デバッグとしてシミュレーション領域をワイヤーフレームで描画
    Gizmos.color = Color.cyan;
    Gizmos.DrawWireCube(WallCenter, WallSize);
}
#endregion

#region Private Functions
// バッファを初期化
void InitBuffer()
{
    // バッファを初期化
    _boidDataBuffer  = new ComputeBuffer(MaxObjectNum,
        Marshal.SizeOf(typeof(BoidData)));
    _boidForceBuffer = new ComputeBuffer(MaxObjectNum,
        Marshal.SizeOf(typeof(Vector3)));

    // Boid データ, Force バッファを初期化
    var forceArr = new Vector3[MaxObjectNum];
    var boidDataArr = new BoidData[MaxObjectNum];
    for (var i = 0; i < MaxObjectNum; i++)
    {
```

```
            forceArr[i] = Vector3.zero;
            boidDataArr[i].Position = Random.insideUnitSphere * 1.0f;
            boidDataArr[i].Velocity = Random.insideUnitSphere * 0.1f;
        }
        _boidForceBuffer.SetData(forceArr);
        _boidDataBuffer.SetData(boidDataArr);
        forceArr    = null;
        boidDataArr = null;
    }

    // シミュレーション
    void Simulation()
    {
        ComputeShader cs = BoidsCS;
        int id = -1;

        // スレッドグループの数を求める
        int threadGroupSize = Mathf.CeilToInt(MaxObjectNum
            / SIMULATION_BLOCK_SIZE);

        // 操舵力を計算
        id = cs.FindKernel("ForceCS"); // カーネル ID を取得
        cs.SetInt("_MaxBoidObjectNum", MaxObjectNum);
        cs.SetFloat("_CohesionNeighborhoodRadius",
            CohesionNeighborhoodRadius);
        cs.SetFloat("_AlignmentNeighborhoodRadius",
            AlignmentNeighborhoodRadius);
        cs.SetFloat("_SeparateNeighborhoodRadius",
            SeparateNeighborhoodRadius);
        cs.SetFloat("_MaxSpeed", MaxSpeed);
        cs.SetFloat("_MaxSteerForce", MaxSteerForce);
        cs.SetFloat("_SeparateWeight", SeparateWeight);
        cs.SetFloat("_CohesionWeight", CohesionWeight);
        cs.SetFloat("_AlignmentWeight", AlignmentWeight);
        cs.SetVector("_WallCenter", WallCenter);
        cs.SetVector("_WallSize", WallSize);
        cs.SetFloat("_AvoidWallWeight", AvoidWallWeight);
        cs.SetBuffer(id, "_BoidDataBufferRead", _boidDataBuffer);
        cs.SetBuffer(id, "_BoidForceBufferWrite", _boidForceBuffer);
        cs.Dispatch(id, threadGroupSize, 1, 1); // ComputeShader を実行

        // 操舵力から、速度と位置を計算
        id = cs.FindKernel("IntegrateCS"); // カーネル ID を取得
        cs.SetFloat("_DeltaTime", Time.deltaTime);
        cs.SetBuffer(id, "_BoidForceBufferRead", _boidForceBuffer);
        cs.SetBuffer(id, "_BoidDataBufferWrite", _boidDataBuffer);
        cs.Dispatch(id, threadGroupSize, 1, 1); // ComputeShader を実行
    }

    // バッファを解放
    void ReleaseBuffer()
    {
        if (_boidDataBuffer != null)
        {
            _boidDataBuffer.Release();
            _boidDataBuffer = null;
        }
```

```
        if (_boidForceBuffer != null)
        {
            _boidForceBuffer.Release();
            _boidForceBuffer = null;
        }
    }
    #endregion
}
```

■ComputeBuffer の初期化　InitBuffer 関数では、GPU 上で計算を行う際に使用するバッファを宣言しています。GPU 上で計算するためのデータを格納するバッファとして、ComputeBuffer というクラスを使用します。ComputeBuffer は ComputeShader のためにデータを格納するデータバッファです。C#スクリプトから GPU 上のメモリバッファに対して読み込みや書き込みができるようになります。初期化時の引数には、バッファの要素の数と、要素 1 つのサイズ（バイト数）を渡します。Marshal.SizeOf() メソッドを使用することで、型のサイズ（バイト数）を取得することができます。ComputeBuffer では、SetData() を用いて、任意の構造体の配列の値をセットすることができます。

■ComputeShader に記述した関数の実行　Simulation 関数では、ComputeShader に必要なパラメータを渡し、計算命令を発行します。

ComputeShader に記述された、実際に GPU に計算をさせる関数はカーネルと呼ばれます。このカーネルの実行単位をスレッドと言い、GPU アーキテクチャに即した並列計算処理を行うために、任意の数まとめてグループとして扱い、それらはスレッドグループと呼ばれます。このスレッドの数とスレッドグループ数の積が、Boid オブジェクトの個体数と同じかそれを超えるように設定します。

カーネルは、ComputeShader スクリプト内で #pragma kernel ディレクティブを用いて指定されます。これにはそれぞれ ID が割り当てられており、C#スクリプトからは FindKernel メソッドを用いることで、この ID を取得することができます。

SetFloat メソッド、SetVector メソッド、SetBuffer メソッドなどを使用し、シミュレーションに必要なパラメータやバッファを ComputeShader に渡します。バッファやテクスチャをセットするときにはカーネル ID が必要になります。

Dispatch メソッドを実行することで、ComputeShader に定義したカーネルを GPU で計算処理を行うように命令を発行します。引数には、カーネル ID とスレッドグループの数を指定します。

3.4.2 Boids.compute

GPU への計算命令を記述します。カーネルは 2 つで、1 つは操舵力を計算するもの、もう 1 つは、その力を適用させ速度や位置を更新するものです。

▼ Boids.compute

```
// カーネル関数を指定
#pragma kernel ForceCS      // 操舵力を計算
#pragma kernel IntegrateCS  // 速度，位置を計算

// Boid データの構造体
struct BoidData
{
    float3 velocity; // 速度
    float3 position; // 位置
};

// スレッドグループのスレッドのサイズ
#define SIMULATION_BLOCK_SIZE 256

// Boid データのバッファ（読み取り用）
StructuredBuffer<BoidData>   _BoidDataBufferRead;
// Boid データのバッファ（読み取り，書き込み用）
RWStructuredBuffer<BoidData> _BoidDataBufferWrite;
// Boid の操舵力のバッファ（読み取り用）
StructuredBuffer<float3>     _BoidForceBufferRead;
// Boid の操舵力のバッファ（読み取り，書き込み用）
RWStructuredBuffer<float3>   _BoidForceBufferWrite;

int _MaxBoidObjectNum; // Boid オブジェクト数

float _DeltaTime;      // 前フレームから経過した時間

float _SeparateNeighborhoodRadius;  // 分離を適用する他の個体との距離
float _AlignmentNeighborhoodRadius; // 整列を適用する他の個体との距離
float _CohesionNeighborhoodRadius;  // 結合を適用する他の個体との距離

float _MaxSpeed;         // 速度の最大値
float _MaxSteerForce;    // 操舵する力の最大値

float _SeparateWeight;  // 分離適用時の重み
float _AlignmentWeight; // 整列適用時の重み
float _CohesionWeight;  // 結合適用時の重み

float4 _WallCenter;      // 壁の中心座標
float4 _WallSize;        // 壁のサイズ
float  _AvoidWallWeight; // 壁を避ける強さの重み


// ベクトルの大きさを制限する
float3 limit(float3 vec, float max)
{
    float length = sqrt(dot(vec, vec)); // 大きさ
    return (length > max && length > 0) ? vec.xyz * (max / length) : vec.xyz;
```

```
}

// 壁に当たった時に逆向きの力を返す
float3 avoidWall(float3 position)
{
    float3 wc = _WallCenter.xyz;
    float3 ws = _WallSize.xyz;
    float3 acc = float3(0, 0, 0);
    // x
    acc.x = (position.x < wc.x - ws.x * 0.5) ? acc.x + 1.0 : acc.x;
    acc.x = (position.x > wc.x + ws.x * 0.5) ? acc.x - 1.0 : acc.x;

    // y
    acc.y = (position.y < wc.y - ws.y * 0.5) ? acc.y + 1.0 : acc.y;
    acc.y = (position.y > wc.y + ws.y * 0.5) ? acc.y - 1.0 : acc.y;

    // z
    acc.z = (position.z < wc.z - ws.z * 0.5) ? acc.z + 1.0 : acc.z;
    acc.z = (position.z > wc.z + ws.z * 0.5) ? acc.z - 1.0 : acc.z;

    return acc;
}

// シェアードメモリ Boid データ格納用
groupshared BoidData boid_data[SIMULATION_BLOCK_SIZE];

// 操舵力の計算用カーネル関数
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void ForceCS
(
    uint3 DTid : SV_DispatchThreadID, // スレッド全体で固有の ID
    uint3 Gid : SV_GroupID, // グループの ID
    uint3 GTid : SV_GroupThreadID, // グループ内のスレッド ID
    uint  GI : SV_GroupIndex // SV_GroupThreadID を一次元にしたもの 0-255
)
{
    const unsigned int P_ID = DTid.x; // 自身の ID
    float3 P_position = _BoidDataBufferRead[P_ID].position; // 自身の位置
    float3 P_velocity = _BoidDataBufferRead[P_ID].velocity; // 自身の速度

    float3 force = float3(0, 0, 0); // 操舵力を初期化

    float3 sepPosSum = float3(0, 0, 0); // 分離計算用 位置加算変数
    int sepCount = 0; // 分離のために計算した他の個体の数のカウント用変数

    float3 aliVelSum = float3(0, 0, 0); // 整列計算用 速度加算変数
    int aliCount = 0; // 整列のために計算した他の個体の数のカウント用変数

    float3 cohPosSum = float3(0, 0, 0); // 結合計算用 位置加算変数
    int cohCount = 0; // 結合のために計算した他の個体の数のカウント用変数

    // SIMULATION_BLOCK_SIZE（グループスレッド数）ごとの実行（グループ数分実行）
    [loop]
    for (uint N_block_ID = 0; N_block_ID < (uint)_MaxBoidObjectNum;
        N_block_ID += SIMULATION_BLOCK_SIZE)
    {
        // SIMULATION_BLOCK_SIZE 分の Boid データを、シェアードメモリに格納
        boid_data[GI] = _BoidDataBufferRead[N_block_ID + GI];
```

```
        // すべてのグループ共有アクセスが完了し、
        // グループ内のすべてのスレッドがこの呼び出しに到達するまで、
        // グループ内のすべてのスレッドの実行をブロックする
        GroupMemoryBarrierWithGroupSync();

        // 他の個体との計算
        for (int N_tile_ID = 0; N_tile_ID < SIMULATION_BLOCK_SIZE;
            N_tile_ID++)
        {
            // 他の個体の位置
            float3 N_position = boid_data[N_tile_ID].position;
            // 他の個体の速度
            float3 N_velocity = boid_data[N_tile_ID].velocity;

            // 自身と他の個体の位置の差
            float3 diff = P_position - N_position;
            // 自身と他の個体の位置の距離
            float  dist = sqrt(dot(diff, diff));

            // --- 分離（Separation） ---
            if (dist > 0.0 && dist <= _SeparateNeighborhoodRadius)
            {
                // 他の個体の位置から自身へ向かうベクトル
                float3 repulse = normalize(P_position - N_position);
                // 自身と他の個体の位置の距離で割る（距離が遠ければ影響を小さく）
                repulse /= dist;
                sepPosSum += repulse; // 加算
                sepCount++;           // 個体数カウント
            }

            // --- 整列（Alignment） ---
            if (dist > 0.0 && dist <= _AlignmentNeighborhoodRadius)
            {
                aliVelSum += N_velocity; // 加算
                aliCount++;              // 個体数カウント
            }

            // --- 結合（Cohesion） ---
            if (dist > 0.0 && dist <= _CohesionNeighborhoodRadius)
            {
                cohPosSum += N_position; // 加算
                cohCount++;              // 個体数カウント
            }
        }
        GroupMemoryBarrierWithGroupSync();
    }

    // 操舵力（分離）
    float3 sepSteer = (float3)0.0;
    if (sepCount > 0)
    {
        sepSteer = sepPosSum / (float)sepCount;     // 平均を求める
        sepSteer = normalize(sepSteer) * _MaxSpeed; // 最大速度に調整
        sepSteer = sepSteer - P_velocity;           // 操舵力を計算
        sepSteer = limit(sepSteer, _MaxSteerForce); // 操舵力を制限
    }
```

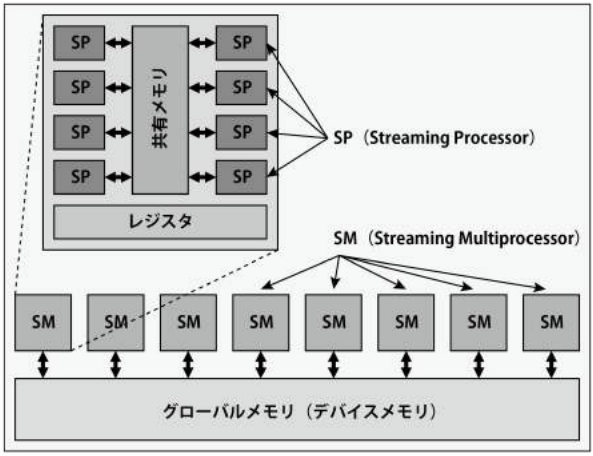
```
    // 操舵力（整列）
    float3 aliSteer = (float3)0.0;
    if (aliCount > 0)
    {
        aliSteer = aliVelSum / (float)aliCount; // 近い個体の速度の平均を求める
        aliSteer = normalize(aliSteer) * _MaxSpeed; // 最大速度に調整
        aliSteer = aliSteer - P_velocity;           // 操舵力を計算
        aliSteer = limit(aliSteer, _MaxSteerForce); // 操舵力を制限
    }
    // 操舵力（結合）
    float3 cohSteer = (float3)0.0;
    if (cohCount > 0)
    {
        // / 近い個体の位置の平均を求める
        cohPosSum = cohPosSum / (float)cohCount;
        cohSteer = cohPosSum - P_position; // 平均位置方向へのベクトルを求める
        cohSteer = normalize(cohSteer) * _MaxSpeed; // 最大速度に調整
        cohSteer = cohSteer - P_velocity;           // 操舵力を計算
        cohSteer = limit(cohSteer, _MaxSteerForce); // 操舵力を制限
    }
    force += aliSteer * _AlignmentWeight; // 操舵力に整列する力を加える
    force += cohSteer * _CohesionWeight;  // 操舵力に結合する力を加える
    force += sepSteer * _SeparateWeight;  // 操舵力に分離する力を加える

    _BoidForceBufferWrite[P_ID] = force; // 書き込み
}

// 速度，位置計算用カーネル関数
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void IntegrateCS
(
    uint3 DTid : SV_DispatchThreadID // スレッド全体で固有の ID
)
{
    const unsigned int P_ID = DTid.x; // インデックスを取得

    BoidData b = _BoidDataBufferWrite[P_ID]; // 現在の Boid データを読み込む
    float3 force = _BoidForceBufferRead[P_ID]; // 操舵力を読み込む

    // 壁に近づいたら反発する力を与える
    force += avoidWall(b.position) * _AvoidWallWeight;

    b.velocity += force * _DeltaTime; // 操舵力を速度に適用
    b.velocity = limit(b.velocity, _MaxSpeed); // 速度を制限
    b.position += b.velocity * _DeltaTime; // 位置を更新

    _BoidDataBufferWrite[P_ID] = b; // 計算結果を書き込む
}
```

操舵力の計算

ForceCS カーネルでは、操舵力の計算を行います。

■共有メモリの活用　groupshared という記憶域修飾子をつけられた変数は共有メモリ（shared memory）に書き込まれるようになります。共有メモリは多くのデータ量

を書き込むことはできませんが、レジスタに近く配置されており非常に高速にアクセスができます。この共有メモリはスレッドグループ内で共有することができます。SIMULATION_BLOCK_SIZE 分の他の個体の情報をまとめて共有メモリに書き込んでおいて、同一スレッドグループ内で高速に読みこむことができるようにすることで、他の個体との位置関係を考慮した計算を効率的に行っていきます。



▲図 3.3　GPU の基本的なアーキテクチャ

GroupMemoryBarrierWithGroupSync()　共有メモリに書き込まれたデータにアクセスする時は、GroupMemoryBarrierWithGroupSync() メソッドを記述し、スレッドグループ内のすべてのスレッドの処理の同期をとっておく必要があります。GroupMemoryBarrierWithGroupSync() は、スレッドグループ内のすべてのスレッドが、この呼び出しに到達するまで、グループ内のすべてのスレッドの実行をブロックします。これにより、スレッドグループ内のすべてのスレッドで boid_data 配列の初期化が適切に終わっていることが保証されるようになります。

■他の個体との距離によって操舵力を計算

分離（Separation）　指定した距離より近い個体があった場合、その個体の位置から自身の位置へ向かうベクトルを求め、正規化します。そのベクトルを、距離の値で割ることで、近ければより避けるように、遠ければ小さく避けるように重みをつけ他の個体と衝突しないようにする力として加算していきます。全ての個体との計算が終わったら、その値を用いて、現在の速度との関係から操舵力を求めます。

整列（Alignment） 指定した距離より近い個体があった場合、その個体の速度（Velocity）を足し合わせていき、同時にその個体数をカウントしていき、それらの値で、近い個体の速度（つまり向いている方向）の平均を求めます。全ての個体との計算が終わったら、その値を用いて、現在の速度との関係から操舵力を求めます。
結合（Cohesion） 指定した距離より近い個体があった場合、その個体の位置を加算していき、同時にその個体数をカウントしていき、それらの値で、近い個体の位置の平均（重心）を求めます。さらに、そこへ向かうベクトルを求め、現在の速度との関係から操舵力を求めます。

■個々の Boid の速度と位置の更新 IntegrateCS カーネルでは、ForceCS() で求めた操舵力を元に、Boid の速度と位置を更新します。AvoidWall では、指定したエリアの外に出ようとした場合、逆向きの力を与え領域の内部に留まるようにしています。

3.4.3 BoidsRender.cs

このスクリプトでは、Boids シミュレーションで得られた結果を、指定したメッシュで描画することを行います。

▼ BoidsRender.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// 同 GameObject に、GPUBoids コンポーネントがアタッチされていることを保証
[RequireComponent(typeof(GPUBoids))]
public class BoidsRender : MonoBehaviour
{
    #region Paremeters
    // 描画する Boids オブジェクトのスケール
    public Vector3 ObjectScale = new Vector3(0.1f, 0.2f, 0.5f);
    #endregion

    #region Script References
    // GPUBoids スクリプトの参照
    public GPUBoids GPUBoidsScript;
    #endregion

    #region Built-in Resources
    // 描画するメッシュの参照
    public Mesh InstanceMesh;
    // 描画のためのマテリアルの参照
    public Material InstanceRenderMaterial;
    #endregion

    #region Private Variables
    // GPU インスタンシングのための引数（ComputeBuffer への転送用）
    // インスタンスあたりのインデックス数，インスタンス数，
    // 開始インデックス位置，ベース頂点位置，インスタンスの開始位置
```

```
    uint[] args = new uint[5] { 0, 0, 0, 0, 0 };
    // GPU インスタンシングのための引数バッファ
    ComputeBuffer argsBuffer;
    #endregion

    #region MonoBehaviour Functions
    void Start ()
    {
        // 引数バッファを初期化
        argsBuffer = new ComputeBuffer(1, args.Length * sizeof(uint),
            ComputeBufferType.IndirectArguments);
    }

    void Update ()
    {
        // メッシュをインスタンシング
        RenderInstancedMesh();
    }

    void OnDisable()
    {
        // 引数バッファを解放
        if (argsBuffer != null)
            argsBuffer.Release();
        argsBuffer = null;
    }
    #endregion

    #region Private Functions
    void RenderInstancedMesh()
    {
        // 描画用マテリアルが Null，または，GPUBoids スクリプトが Null，
        // または GPU インスタンシングがサポートされていなければ，処理をしない
        if (InstanceRenderMaterial == null || GPUBoidsScript == null ||
            !SystemInfo.supportsInstancing)
            return;

        // 指定したメッシュのインデックス数を取得
        uint numIndices = (InstanceMesh != null) ?
            (uint)InstanceMesh.GetIndexCount(0) : 0;
        // メッシュのインデックス数をセット
        args[0] = numIndices;
        // インスタンス数をセット
        args[1] = (uint)GPUBoidsScript.GetMaxObjectNum();
        argsBuffer.SetData(args); // バッファにセット

        // Boid データを格納したバッファをマテリアルにセット
        InstanceRenderMaterial.SetBuffer("_BoidDataBuffer",
            GPUBoidsScript.GetBoidDataBuffer());
        // Boid オブジェクトスケールをセット
        InstanceRenderMaterial.SetVector("_ObjectScale", ObjectScale);
        // 境界領域を定義
        var bounds = new Bounds
        (
            GPUBoidsScript.GetSimulationAreaCenter(), // 中心
            GPUBoidsScript.GetSimulationAreaSize()    // サイズ
        );
        // メッシュを GPU インスタンシングして描画
```

```
        Graphics.DrawMeshInstancedIndirect
        (
            InstanceMesh,           // インスタンシングするメッシュ
            0,                      // submesh のインデックス
            InstanceRenderMaterial, // 描画を行うマテリアル
            bounds,                 // 境界領域
            argsBuffer              // GPU インスタンシングのための引数のバッファ
        );
    }
    #endregion
}
```

GPU インスタンシング

大量の同一の Mesh を描画したい時、一つ一つ GameObject を生成するのでは、ドローコールが上がり描画負荷が増大していきます。また、ComputeShader での計算結果を CPU メモリに転送するコストが高く、高速に処理を行いたい場合、GPU での計算結果をそのまま描画用シェーダに渡し描画処理をさせることが必要です。Unity の GPU インスタンシングを使えば、不要な GameObject の生成を行うことなく、大量の同一の Mesh を少ないドローコールで高速に描画することができます。

Graphics.DrawMeshInstancedIndirect() メソッド　このスクリプトでは、Graphics.DrawMeshInstancedIndirect メソッドを用いて GPU インスタンシングによるメッシュ描画を行います。このメソッドでは、メッシュのインデックス数やインスタンス数を ComputeBuffer として渡すことができます。GPU からすべてのインスタンスデータを読み込みたい場合に便利です。

Start() では、この GPU インスタンシングのための引数バッファを初期化しています。初期化時のコンストラクタの 3 つ目の引数には **ComputeBufferType.IndirectArguments** を指定します.

RenderInstancedMesh() では、GPU インスタンシングによるメッシュ描画を実行しています。描画のためのマテリアル InstanceRenderMaterial に、SetBuffer メソッドで、Boids シミュレーションによって得られた Boid のデータ（速度、位置の配列）を渡しています。

Graphics.DrawMeshInstancedIndrect メソッドには、インスタンシングするメッシュ、submesh のインデックス、描画用マテリアル、境界データ、また、インスタンス数などのデータを格納したバッファを引数に渡します。

このメソッドは通常 Update() 内で呼ばれるようにします。

3.4.4 BoidsRender.shader

Graphics.DrawMeshInstancedIndrect メソッドに対応した描画用のシェーダです。

▼ BoidsRender.shader

```
Shader "Hidden/GPUBoids/BoidsRender"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Standard vertex:vert addshadow
        #pragma instancing_options procedural:setup

        struct Input
        {
            float2 uv_MainTex;
        };
        // Boid の構造体
        struct BoidData
        {
            float3 velocity; // 速度
            float3 position; // 位置
        };

        #ifdef UNITY_PROCEDURAL_INSTANCING_ENABLED
        // Boid データの構造体バッファ
        StructuredBuffer<BoidData> _BoidDataBuffer;
        #endif

        sampler2D _MainTex; // テクスチャ

        half   _Glossiness; // 光沢
        half   _Metallic;   // 金属特性
        fixed4 _Color;      // カラー

        float3 _ObjectScale; // Boid オブジェクトのスケール

        // オイラー角（ラジアン）を回転行列に変換
        float4x4 eulerAnglesToRotationMatrix(float3 angles)
        {
            float ch = cos(angles.y); float sh = sin(angles.y); // heading
            float ca = cos(angles.z); float sa = sin(angles.z); // attitude
            float cb = cos(angles.x); float sb = sin(angles.x); // bank

            // RyRxRz (Heading Bank Attitude)
            return float4x4(
                ch * ca + sh * sb * sa, -ch * sa + sh * sb * ca, sh * cb, 0,
                cb * sa, cb * ca, -sb, 0,
                -sh * ca + ch * sb * sa, sh * sa + ch * sb * ca, ch * cb, 0,
                0, 0, 0, 1
            );
```

```
        }

        // 頂点シェーダ
        void vert(inout appdata_full v)
        {
            #ifdef UNITY_PROCEDURAL_INSTANCING_ENABLED

            // インスタンス ID から Boid のデータを取得
            BoidData boidData = _BoidDataBuffer[unity_InstanceID];

            float3 pos = boidData.position.xyz; // Boid の位置を取得
            float3 scl = _ObjectScale;          // Boid のスケールを取得

            // オブジェクト座標からワールド座標に変換する行列を定義
            float4x4 object2world = (float4x4)0;
            // スケール値を代入
            object2world._11_22_33_44 = float4(scl.xyz, 1.0);
            // 速度から Y 軸についての回転を算出
            float rotY =
                atan2(boidData.velocity.x, boidData.velocity.z);
            // 速度から X 軸についての回転を算出
            float rotX =
                -asin(boidData.velocity.y / (length(boidData.velocity.xyz)
                + 1e-8)); // 0 除算防止
            // オイラー角（ラジアン）から回転行列を求める
            float4x4 rotMatrix =
                eulerAnglesToRotationMatrix(float3(rotX, rotY, 0));
            // 行列に回転を適用
            object2world = mul(rotMatrix, object2world);
            // 行列に位置（平行移動）を適用
            object2world._14_24_34 += pos.xyz;

            // 頂点を座標変換
            v.vertex = mul(object2world, v.vertex);
            // 法線を座標変換
            v.normal = normalize(mul(object2world, v.normal));
            #endif
        }

        void setup()
        {
        }

        // サーフェスシェーダ
        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

#pragma surface surf Standard vertex:vert addshadow この部分では、サーフェ

スシェーダとして surf()、ライティングモデルは Standard、カスタム頂点シェーダとして vert() を指定するという処理を行っています。

#pragma instancing_options ディレクティブで procedural:FunctionName と記述することによって、Graphics.DrawMeshInstancedIndirect メソッドを使うときのための追加のバリアントを生成するように Unity に指示することができ、頂点シェーダステージの始めに、FunctionName で指定した関数が呼ばれるようになります。公式のサンプル（https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html）などを見ると、この関数内で、個々のインスタンスの位置や回転、スケールに基づき、unity_ObjectToWorld 行列, unity_WorldToObject 行列の書き換えを行っていますが、このサンプルプログラムでは、頂点シェーダ内で Boids のデータを受け取り、頂点や法線の座標変換を行っています（良いのかわかりませんが…）。そのため、指定した setup 関数内では何も記述していません。

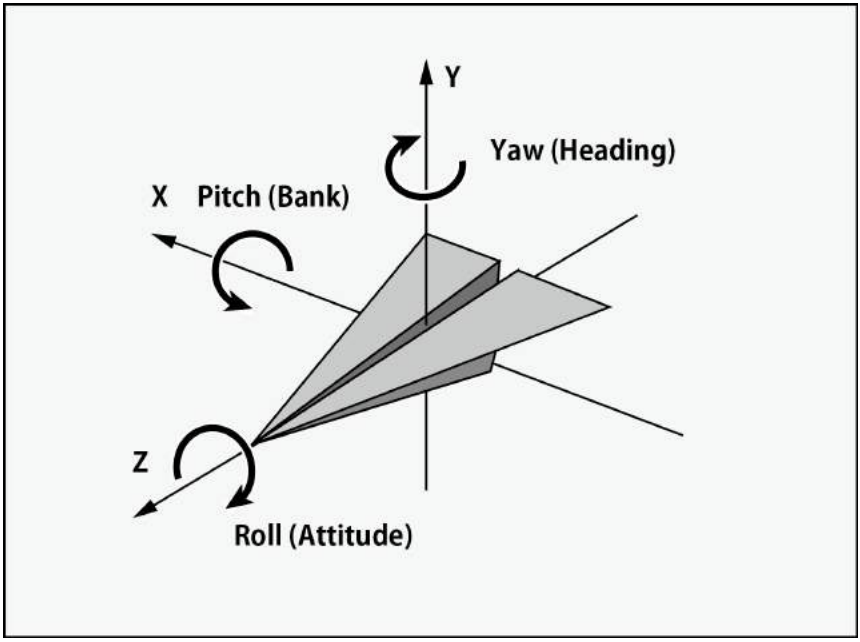
頂点シェーダでインスタンスごとの Boid のデータを取得し座標変換をする

頂点シェーダ（Vertex Shader）に、シェーダに渡されたメッシュの頂点に対して行う処理を記述します。

unity_InstanceID によってインスタンスごとに固有の ID を取得することができます。この ID を Boid データのバッファとして宣言した StructuredBuffer の配列のインデックスに指定することによって、インスタンスごとに固有の Boid データを得ることができます。

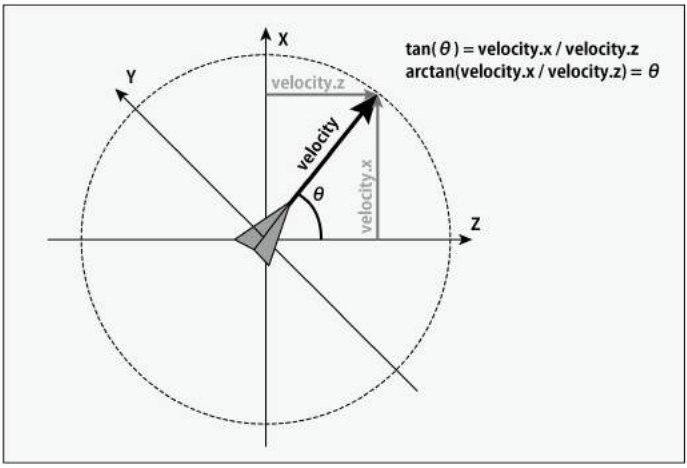
回転を求める

Boid の速度データから、進行方向を向くような回転の値を算出します。ここでは直感的に扱うために、回転はオイラー角で表現することにします。Boid を飛行体と捉えると、オブジェクトを基準とした座標の 3 軸の回転は、それぞれ、ピッチ、ヨー、ロールと呼ばれます。



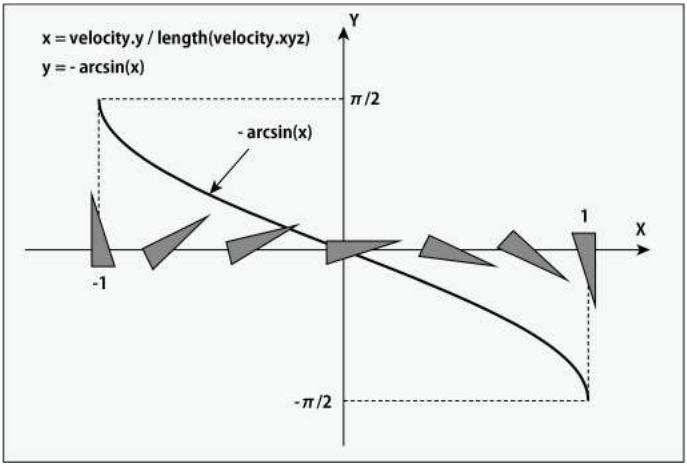
▲図 3.4　軸と回転の呼称

まず、Z 軸についての速度と X 軸についての速度から、逆正接（アークタンジェント）を返す atan2 メソッドを用いてヨー（水平面に対してどの方向を向いているか）を求めます。



▲図 3.5　速度と角度（ヨー）の関係

次に、速度の大きさと、Y 軸についての速度の比率から、逆正弦（アークサイン）を返す asin メソッドを用いてピッチ（上下の傾き）を求めています。それぞれの軸についての速度の中で Y 軸の速度が小さい場合は、変化が少なく水平を保つように重みのついた回転量になるようになっています。



▲図 3.6　速度と角度（ピッチ）の関係

Boid のトランスフォームを適用する行列を計算

移動、回転、拡大縮小といった座標変換処理は、まとめて一つの行列で表現することができます。4x4 の行列 object2world を定義します。

■拡大縮小　まず、スケール値を代入します。XYZ 軸それぞれに $S_xS_yS_z$ だけ拡大縮小を行う行列 S は以下のように表現されます。

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

HLSL の float4x4 型の変数は、._11_22_33_44 のようなスィズルを用いて行列の特定の要素を指定できます。デフォルトであれば、成分は以下のように整列してます。

11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

ここでは、11、22、33、に XYZ それぞれのスケールの値、44 には 1 を代入します。

■**回転**　次に、回転を適用します。XYZ 軸それぞれについての回転 $\mathrm{R_x R_y R_z}$ を行列で表現すると、

$$\mathrm{R_x}(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathrm{R_y}(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathrm{R_z}(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

これを一つに行列に合成します。このとき、合成する回転の軸の順によって回転時の挙動が変化しますが、この順に合成すると、Unity の標準の回転と同様のものになるはずです。

$$\begin{aligned}
&\mathrm{R_y}(\theta)\mathrm{R_x}(\phi)\mathrm{R_z}(\psi) \\
&= \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos(\theta) & \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos(\theta)\cos(\psi) + \sin(\theta)\sin(\phi)\sin(\psi) & -\cos(\theta)\sin(\psi) + \sin(\theta)\sin(\phi)\cos(\psi) & \sin(\theta)\cos(\phi) & 0 \\ \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi) & -\sin(\phi) & 0 \\ -\sin(\theta)\cos(\psi) + \cos(\theta)\sin(\phi)\sin(\psi) & \sin(\theta)\sin(\psi) + \cos(\theta)\sin(\phi)\cos(\psi) & \cos(\theta)\cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}$$

▲図 3.7　回転行列の合成

これによって求められた回転行列と、上のスケールを適用した行列との積を求めることによって、回転を適用します。

■**平行移動**　次に、平行移動を適用します。それぞれの軸に、$\mathrm{T_x T_y T_z}$ 平行移動するとすると、行列は以下のように表現されます。

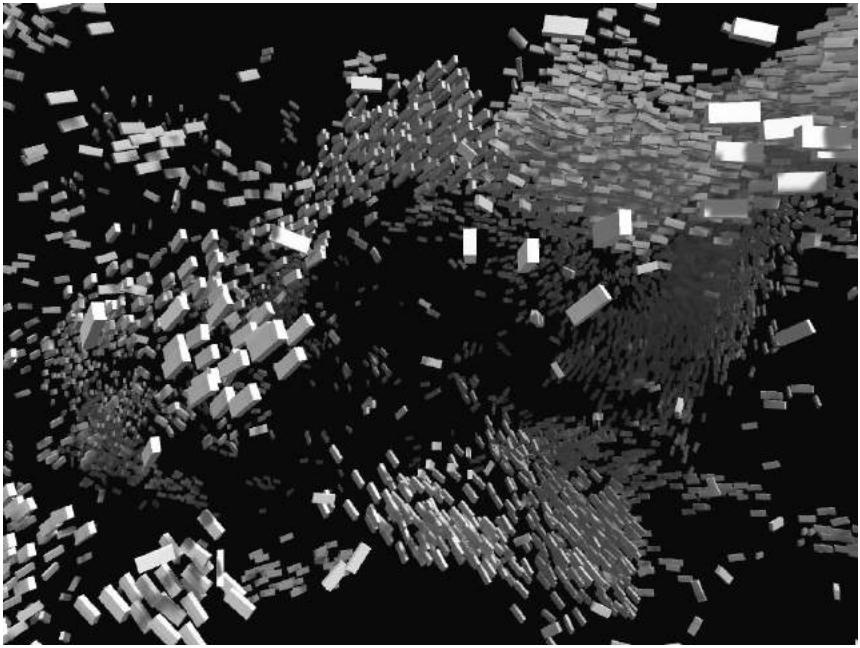
$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

この平行移動は、14, 24, 34 成分に XYZ それぞれの軸についての位置（Position）データを加算することで適用できます。

これらの計算によって得られた行列を、頂点、法線に適用させることによって、Boid のトランスフォームデータを反映します。

3.4.5 描画結果

このように群れっぽい動きをするオブジェクトが描画されると思います。



▲図 3.8　実行結果

3.5 まとめ

この章で紹介した実装は、最低限の Boids のアルゴリズムを利用したものですが、パラメータの調整によっても、群は大きなまとまりになったり、幾つもの小群体が作られたりと、異なる特徴を持った動きを見せると思います。ここで示した基本的な行動規則の他にも、考慮すべきルールが存在します。例えば、これが魚の群だとして、それらを捕食する外敵が現れたとすると当然逃げるような動きをし、地形など障害物があるとすれば魚はぶつからないように避けるでしょう。視覚について考えると、動物の種によっては視野や精度も異なり、視界の外の他の個体は計算処理から除外するなどすると、より実際のものに近づいていくと思います。空を飛ぶのか、水の中を動くのか、陸上を移動するのかといった環境や、移動運動のための運動器官の特性によっても動きの特徴が変わってきます。個体差にも着眼すべきです。

GPU による並列処理は、CPU による演算に比べれば多くの個体を計算できますが、基本的には他の個体との計算は総当たりで行っており、計算効率はあまり良いとは言えません。それには、個体をその位置によってグリッドやブロックで分割した領域に登録しておき、隣接した領域に存在する個体についてだけ計算処理を行うというように、近傍個体探索の効率化を図ることで計算コストを抑えることができます。

このように改良の余地は多く残されており、適切な実装と行動のルールを適用することにより、いっそう美しく、迫力、密度と味わいのある群の動きが表現できるようになることと思います。できるようになりたいです。

3.6 参照

- Boids Background and Update - https://www.red3d.com/cwr/boids/
- THE NATURE OF CODE - http://natureofcode.com/
- Real-Time Particle Systems on the GPU in Dynamic Environments - http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/02/Chapter7-Drone-Real-Time**Particle**Systems**On**The_GPU.pdf
- Practical Rendering and Computation with Direct3D 11 - https://dl.acm.org/citation.cfm?id=2050039
- GPU 並列図形処理入門 - http://gihyo.jp/book/2014/978-4-7741-6304-8

第 4 章

格子法による流体シミュレーション

4.1 この章について

本章では、ComputeShader を使った格子法による流体シミュレーションについて解説します。

4.2 サンプルデータ

4.2.1 コード

https://github.com/IndieVisualLab/UnityGraphicsProgramming/
の Assets/StabeFluids に格納されています。

4.2.2 実行環境

- ComputeShader が実行できる、シェーダーモデル 5.0 対応環境
- 執筆時環境、Unity5.6.2, Unity2017.1.1 で動作確認済み

4.3 はじめに

本章では、格子法による流体シミュレーションと、それらを実現するにあたって必要となる、数式の計算方法や捉え方を解説していきます。まず格子法とは何でしょう。その意味を探る為に、一度流体力学での「流れ」の解析方法に少し迫ってみましょう。

4.3.1 流体力学での捉え方

流体力学とは、自然現象である「流れ」を数式化して、計算可能なものとする事に特徴をおいています。この「流れ」、一体どうすれば数値化し解析することが出来るでしょうか。
端的に行ってしまいますと、「時間が一瞬進んだ時の流速」を導く事で数値化する事ができます。少し数学的に言うと、時間で微分した際の流速ベクトルの変化量の解析と言い換える事ができます。
ただ、この流れを解析する方法として、二つの手法が考えられます。
一つは、お風呂のお湯をイメージした際に、お風呂にはったお湯を格子状に分割し、その固定された各格子空間の流速ベクトルを測定する方法。
そしてもう一つは、お風呂にアヒルを浮かべ、アヒルの動き自体を解析する方法です。
この二つの方法の内、前者を「オイラーの方法」、後者を「ラグランジュの方法」と呼びます。

4.3.2 様々な流体シミュレーション

さて、一旦コンピューターグラフィックスの方に話を戻しましょう。流体シミュレーションにも、「オイラーの方法」や「ラグランジュの方法」の様にいくつかのシミュレーション方法が存在しますが、大きく分けて、以下の 3 種類に大別する事ができます。

- 格子法 (e.g. Stable Fluid)
- 粒子法 (e.g. SPH)
- 格子法＋粒子法 (e.g. FLIP)

漢字の意味合いから少し想像することができるかもしれませんが、格子法は「オイラーの方法」の様に、流れをシミュレーションする際に格子状の「場」を作り、時間で微分した際にその各格子がどういった速度になっているかをシミュレーションする手法をいいます。また粒子法は「ラグランジュの方法」の様に、その粒子の方に着目し、粒子自体の移流をシミュレーションする方法を言います。
格子法・粒子法と共に、お互いに得意不得意な範囲があります。
格子法は流体のシミュレーションにおいて、圧力・粘性・拡散等の計算は得意ですが、移流の計算が不得意です。
これとは逆に、粒子法は移流の計算が得意です。（これらの得意不得意は、オイラーの方法とラグランジュの方法の解析の仕方を思い浮かべると想像がつくかもしれません。）
これらを補う為に、FLIP 法に代表される、格子法＋粒子法と言った得意分野を補い

合う手法も生まれています。

本稿では SIGGRAPH 1999. で発表された Jon Stam 氏の格子法における非圧縮性粘性流体シミュレーションの論文である Stable Fluids を元に、流体シミュレーションの実装方法やシミュレーションにおける必要な数式の説明を行なっていきます。

4.4 ナビエ・ストークス方程式について

まずは、格子法におけるナビエ・ストークスの方程式について見ていきましょう。

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\,\vec{u} + \nu \nabla^2 \vec{u} + \vec{f}$$

$$\frac{\partial \rho}{\partial t} = -(\vec{u} \cdot \nabla)\,\rho + \kappa \nabla^2 \rho + S$$

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

上記の内、一つ目の方程式は速度場、二つ目は密度場を表します。また、3 つ目は「連続の式（質量保存則）」となります。これらの 3 つの式を一つずつ紐解いて見ましょう。

4.5 連続の式（質量保存則）

まずは式としても短く、「非圧縮性」流体をシミュレーションする際の条件として働く「連続の式（質量保存則）」から紐解いて見ましょう。

流体をシミュレーションする際に、その対象が圧縮性か非圧縮性かを明確に区別する必要があります。例えば、気体等の密度が圧力によって変化する物が対象である場合は圧縮性流体となります。逆に、水などの密度がどの場所でも一定である物は、非圧縮性流体となります。

本章では非圧縮性流体のシミュレーションを取り扱いますので、速度場の各セルの発散は 0 に保つ必要があります。つまり、速度場の流入と流出を相殺させ、0 になるように維持します。流入があれば流出させる為、流速は伝搬して行く事となります。この条件は連続の式（質量保存則）として、以下の方程式で表す事ができます。

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

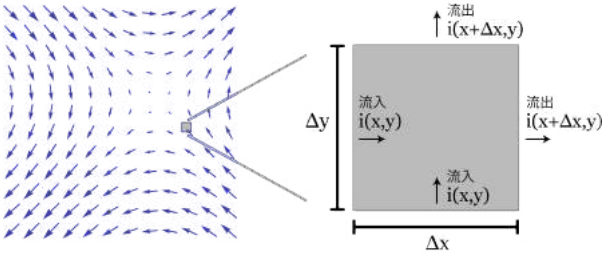
上記は「発散（ダイバージェンス）が 0」であるという意味になります。まずは「発散（ダイバージェンス）」の数式を確認しておきましょう。

4.5.1 発散（Divergence）

$$\nabla \cdot \overrightarrow{u} = \nabla \cdot (u, v) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

∇ （ナブラ演算子）はベクトル微分演算子といいます。例えばベクトル場が 2 次元と想定した場合に、図のように $\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$ の偏微分を取る際の、偏微分の表記を簡略化した演算子として作用します。 ∇ 演算子は演算子ですので、それだけでは意味を持ちませんが、一緒に組み合わせる式が内積なのか、外積なのか、それとも単に ∇f といった関数なのかで演算内容が変わってきます。
今回は偏微分の内積をとる「発散（ダイバージェンス）」について説明しておきましょう。まず、なぜこの式が「発散」という意味になるのかを見てみます。

発散を理解する為に、まずは下記のような格子空間の一つのセルを切り出して考えてみましょう。



▲図 4.1　ベクトル場から微分区間（Δ x, Δ y）のセルを抽出

発散とは、ベクトル場の一つのセルにどれくらいのベクトルが流出、流入しているかを算出する事を言います。なお流出を＋、流入を－とします。

発散は上記のように、ベクトル場のセルを切り取った際の偏微分をみた際に、x 方向の特定のポイント x と微量に進んだ Δx との変化量、また、y 方向の特定のポイント y と微量に進んだ Δy との変化量の内積で求める事ができます。なぜ偏微分との内積で流出が求まるかは、上記の図を微分演算する事で証明できます。

$$\frac{i(x+\Delta x, y)\Delta y - i(x,y)\Delta y + j(x, y+\Delta y)\Delta x - j(x,y)\Delta x}{\Delta x \Delta y}$$

$$= \frac{i(x+\Delta x, y) - i(x,y)}{\Delta x} + \frac{j(x, y+\Delta y) - j(x,y)}{\Delta y}$$

上記の式から極限をとり、

$$\lim_{\Delta x \to 0} \frac{i(x+\Delta x, y) - i(x,y)}{\Delta x} + \lim_{\Delta y \to 0} \frac{j(x, y+\Delta y) - j(x,y)}{\Delta y} = \frac{\partial i}{\partial x} + \frac{\partial j}{\partial y}$$

とする事で、最終的に偏微分との内積の式と等式になる事がわかります。

4.6 速度場

次に、格子法の本丸である速度場について説明していきます。その前に、速度場のナビエ・ストークス方程式を実装していくにあたって、先ほど確認した発散（divergence）に加え、勾配（gradient）とラプラシアン（Laplacian）について確認しておきましょう。

4.6.1 勾配（Gradient）

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

$\nabla f(grad\ f)$ は勾配を求める式となります。意味としては、各偏微分方向に微小に進んだ座標を、関数 f にてサンプリングし、求められた各偏微分方向の値を合成する事によって、最終的にどのベクトルを向くのかを意味しています。つまり、偏微分した際の値の大きい方向に向いたベクトルを算出する事ができます。

4.6.2 ラプラシアン（Laplacian）

$$\Delta f = \nabla^2 f = \nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

ラプラシアンはナブラを上下反転させた記号で表されます。(デルタと同じですが、文脈から読み取り、間違えないようにしましょう。)
$\nabla^2 f$ 、もしくは $\nabla \cdot \nabla f$ とも書き、二階偏微分として演算されます。
また、解体して考えると、関数の勾配をとって、発散を求めた形とも取れるでしょう。意味合い的に考えると、ベクトル場の中で勾配方向に集中した箇所は流入が多い為、発散をとった場合－に、逆に勾配の低い箇所は湧き出しが多いので発散を取った時に＋になる事が想像できます。
ラプラシアン演算子にはスカラーラプラシアンとベクトルラプラシアンがあり、ベクトル場に作用させる場合は、勾配・発散・回転（∇とベクトルの外積）を用いた、

$$\nabla^2 \overrightarrow{u} = \nabla\nabla \cdot \overrightarrow{u} - \nabla \times \nabla \times \overrightarrow{u}$$

といった式で導くのですが、直交座標系の場合のみ、ベクトルの成分毎に勾配と発散を求め、合成する事で求める事ができます。

$$\nabla^2 \overrightarrow{u} = \left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2}, \frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_y}{\partial z^2}, \frac{\partial^2 u_z}{\partial x^2} + \frac{\partial^2 u_z}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} \right)$$

以上で、格子法でのナビエ・ストークス方程式を解くための必要な数式の確認は完了しました。ここから、速度場の方程式を各項ごとに見ていきましょう。

4.6.3 ナビエ・ストークス方程式から速度場の確認

$$\frac{\partial \overrightarrow{u}}{\partial t} = -\left(\overrightarrow{u} \cdot \nabla\right) \overrightarrow{u} + \nu \nabla^2 \overrightarrow{u} + \overrightarrow{f}$$

上記の内、$\overrightarrow{u}$ は流速、ν は動粘性係数（kinematic viscosity）、$\overrightarrow{f}$ は外力（force）になります。
左辺側は時間で偏微分をとった際の流速である事がわかります。右辺側は第一項を移流項、第二項を拡散粘性項、第三項を圧力項、第四項を外力項とします。

これらは、計算時には一括でできるものであっても、実装時にはステップに分けて実装して行く必要があります。
まず、ステップとして、外力を受けなければ、初期条件のまま変化を起こす事ができませんので、第四項の外力項から考えて見たいと思います。

4.6.4 速度場外力項

これはシンプルに外部からのベクトルを加算する部分となります。つまり初期条件で速度場がベクトル量 0 の状態に対し、ベクトルの起点として UI であったりなんらかのイベントから、RWTexture2D の該当 ID にベクトルを加算する部分となります。
コンピュートシェーダーの外力項のカーネルは、以下の様に実装しておきます。また、コンピュートシェーダーにて使用予定の各係数やバッファの定義も記述しておきます。

```
float visc;                    //動粘性係数
float dt;                      //デルタタイム
float velocityCoef;            //速度場外力係数
float densityCoef;             //密度場外圧係数
```

```
//xy = velocity, z = density, 描画シェーダに渡す流体ソルバー
RWTexture2D<float4> solver;
//density field, 密度場
RWTexture2D<float>  density;
//velocity field, 速度場
RWTexture2D<float2> velocity;
//xy = pre vel, z = pre dens. when project, x = p, y = div
//1 ステップ前のバッファ保存、及び質量保存時の一時バッファ
RWTexture2D<float3> prev;
//xy = velocity source, z = density source 外力入力バッファ
Texture2D source;

[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void AddSourceVelocity(uint2 id : SV_DispatchThreadID)
{
    uint w, h;
    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        velocity[id] += source[id].xy * velocityCoef * dt;
        prev[id] = float3(source[id].xy * velocityCoef * dt, prev[id].z);
    }
}
```

次のステップとして、第二項の拡散粘性項を実装します。

4.6.5 速度場拡散粘性項

$$\nu\nabla^2\vec{u}$$

∇ 演算子や Δ 演算子の左右に値がある時には、「右の要素にのみ作用する」というルールがありますので、この場合、動粘性係数は一旦置いておいて、ベクトルラプラシアンの部分を先に考えます。
流速 $\vec{u}$ に対してベクトルラプラシアンで、ベクトルの各成分毎の勾配と発散をとり合成させ、流速を隣接へ拡散させています。そこに動粘性係数を乗算する事によって、拡散の勢いを調整します。
ここでは流速の各成分の勾配を取った上に拡散させていますので、隣接からの流入も隣接への流出も起こり、ステップ 1 で受けたベクトルが隣接へと影響していくという現象が分かるかと思います。
実装面においては、少し工夫が必要となります。数式通りに実装すると、粘性係数と微分時間・格子数を乗算させた拡散率が高くなってしまった場合に、振動が起こり、収束が取れず最後にはシミュレーション自体が発散してしまいます。
拡散を Stable な状態にする為に、ここではガウス・ザイデル法やヤコビ法、SOR 法等の反復法が用いられます。ここではガウス・ザイデル法でシミュレーションしてみましょう。

ガウス・ザイデル法とは、式を自セルに対する未知数からなる線形方程式に変換し、算出された値をすぐに次の反復時に使い、連鎖させることで近似の答えに収束させていく方法です。反復回数は多ければ多いほど正確な値へと収束していきますが、リアルタイムレンダリングにおけるグラフィックスで必要なのは、正確な結果ではなく、より良いフレームレートと見た目の美しさですので、イテレーション回数はマシンパフォーマンスや見た目を考慮し、調整しましょう。

```
#define GS_ITERATE 4

[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void DiffuseVelocity(uint2 id : SV_DispatchThreadID)
{
    uint w, h;
    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        float a = dt * visc * w * h;

        [unroll]
        for (int k = 0; k < GS_ITERATE; k++) {
            velocity[id] = (prev[id].xy + a * (
                            velocity[int2(id.x - 1, id.y)] +
                            velocity[int2(id.x + 1, id.y)] +
                            velocity[int2(id.x, id.y - 1)] +
                            velocity[int2(id.x, id.y + 1)]
                            )) / (1 + 4 * a);
            SetBoundaryVelocity(id, w, h);
        }
    }
}
```

上記の SetBoundaryVelocity 関数は境界用のメソッドになります。詳しくはリポジトリをご参照下さい。

4.6.6 質量保存

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

ここで一旦、項を進める前に質量保存側に立ち返りましょう。これまでの工程で、外力項で受けた力を速度場に拡散させましたが、現状、各セルの質量は保存されておらず、湧き出しっぱなしの場所と流入が多い場所とで、質量が保存されていない状態になっています。
上記の方程式の様に、質量は必ず保存させ各セルの発散を 0 に持っていかないといけませんから、ここで一旦質量を保存をしておきましょう。
なお、質量保存ステップを ComputeShader で行う際、隣接スレッドとの偏微分演算

を行う為、場を確定しておかなければなりません。グループシェアードメモリ内で偏微分演算ができれば高速化が見込めたのですが、別のグループスレッドから偏微分を取った時に、やはり値が取得できず汚い結果となってしまった為、ここはバッファを確定しながら、3 ステップに分け進めます。
速度場から発散算出 > Poisson 方程式をガウス・ザイデル法で算出 > 速度場に減算させ質量保存
の 3 ステップにカーネルをわけ、場を確定しながら質量保存に持っていきます。なお、SetBound~系は境界に対するメソッドの呼び出しになります。

```
//質量保存 Step1.
//step1 では、速度場から発散の算出
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep1(uint2 id : SV_DispatchThreadID)
{
    uint w, h;
    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        float2 uvd;
        uvd = float2(1.0 / w, 1.0 / h);

        prev[id] = float3(0.0,
                    -0.5 *
                    (uvd.x * (velocity[int2(id.x + 1, id.y)].x -
                              velocity[int2(id.x - 1, id.y)].x)) +
                    (uvd.y * (velocity[int2(id.x, id.y + 1)].y -
                              velocity[int2(id.x, id.y - 1)].y)),
                    prev[id].z);

        SetBoundaryDivergence(id, w, h);
        SetBoundaryDivPositive(id, w, h);
    }
}

//質量保存 Step2.
//step2 では、step1 で求めた発散から Poisson 方程式をガウス・ザイデル法で解く
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep2(uint2 id : SV_DispatchThreadID)
{
    uint w, h;

    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        for (int k = 0; k < GS_ITERATE; k++)
        {
            prev[id] = float3(
                        (prev[id].y + prev[uint2(id.x - 1, id.y)].x +
                                      prev[uint2(id.x + 1, id.y)].x +
                                      prev[uint2(id.x, id.y - 1)].x +
```

```
                                    prev[uint2(id.x, id.y + 1)].x) / 4,
                    prev[id].yz);
            SetBoundaryDivPositive(id, w, h);
        }
    }
}

//質量保存 Step3.
//step3 で、∇· u = 0 にする.
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep3(uint2 id : SV_DispatchThreadID)
{
    uint w, h;

    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        float  velX, velY;
        float2 uvd;
        uvd = float2(1.0 / w, 1.0 / h);

        velX = velocity[id].x;
        velY = velocity[id].y;

        velX -= 0.5 * (prev[uint2(id.x + 1, id.y)].x -
                       prev[uint2(id.x - 1, id.y)].x) / uvd.x;
        velY -= 0.5 * (prev[uint2(id.x, id.y + 1)].x -
                       prev[uint2(id.x, id.y - 1)].x) / uvd.y;

        velocity[id] = float2(velX, velY);
        SetBoundaryVelocity(id, w, h);
    }
}
```

これで速度場を質量保存がされた状態にできました。流出した箇所に流入がおき、流入が多い箇所からは流出がおきる為、流体らしい表現になりました。

4.6.7 移流項

$$-(\vec{u}\cdot\nabla)\,\vec{u}$$

移流項はラグランジュの方法的な手法が用いられるのですが、1 ステップ前の速度場のバックトレースを行い、該当セルから速度ベクトルを引いた箇所の値を、現在いる場所に移動するといった作業を各セルに対して行います。バックトレースした際に、格子にぴったり収まる場所に遡れる訳ではありませんので、移流の際は近傍 4 セルとの線形補間を行い、正しい値を移流させます。

```
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void AdvectVelocity(uint2 id : SV_DispatchThreadID)
{
    uint w, h;
    density.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        int ddx0, ddx1, ddy0, ddy1;
        float x, y, s0, t0, s1, t1, dfdt;

        dfdt = dt * (w + h) * 0.5;

        //バックトレースポイント割り出し.
        x = (float)id.x - dfdt * prev[id].x;
        y = (float)id.y - dfdt * prev[id].y;
        //ポイントがシミュレーション範囲内に収まるようにクランプ.
        clamp(x, 0.5, w + 0.5);
        clamp(y, 0.5, h + 0.5);
        //バックトレースポイントの近傍セル割り出し.
        ddx0 = floor(x);
        ddx1 = ddx0 + 1;
        ddy0 = floor(y);
        ddy1 = ddy0 + 1;
        //近傍セルとの線形補間用の差分を取っておく.
        s1 = x - ddx0;
        s0 = 1.0 - s1;
        t1 = y - ddy0;
        t0 = 1.0 - t1;

        //バックトレースし、1step 前の値を近傍との線形補間をとって、現在の速度場に代
入。
        velocity[id] = s0 * (t0 * prev[int2(ddx0, ddy0)].xy +
                             t1 * prev[int2(ddx0, ddy1)].xy) +
                       s1 * (t0 * prev[int2(ddx1, ddy0)].xy +
                             t1 * prev[int2(ddx1, ddy1)].xy);
        SetBoundaryVelocity(id, w, h);
    }
}
```

4.7 密度場

次に密度場の方程式をみてみましょう。

$$\frac{\partial \rho}{\partial t} = -\left(\vec{u} \cdot \nabla\right)\rho + \kappa \nabla^2 \rho + S$$

上記の内、 $\vec{u}$ は流速、 κ は拡散係数、 ρ は密度、S は外圧になります。
密度場は必ずしも必要ではありませんが、速度場を求めた際の各ベクトルに対し、密度場で拡散させた画面上のピクセルを乗せる事で、溶けながら流れる様な、より流体らしい表現が可能になります。
尚、密度場の数式を見て気づいた方もいらっしゃるかと思いますが、速度場と全く同

じフローになっており、違いはベクトルがスカラーになっている点と、動粘性係数 ν が拡散係数 κ になっている点、質量保存則を用いない点の 3 点のみしかありません。密度場は密度の変化の場ですので、非圧縮性である必要はなく、質量保存の必要がありません。また、動粘性係数と拡散係数は、係数としての使い所は同じになります。ですので、先ほど速度場で用いたカーネルの質量保存則以外のカーネルを、次元を落として作ることによって、密度場を実装する事が可能です。紙面上密度場の解説はしませんが、リポジトリには密度場も実装しておりますので、そちらもご参照ください。

4.8 シミュレーションの各項ステップ

上記の速度場及び密度場、質量保存則を用いることによって流体をシミュレーションする事ができるのですが、シミュレーションのステップについて、最後に見ておきましょう。

- 外力イベントを発生させ、速度場と密度場の外力項にインプット
- 速度場を以下のステップで更新
  - 拡散粘性項
  - 質量保存則
  - 移流項
  - 質量保存則
- その後密度場を以下のステップで更新
  - 拡散項
  - 速度場の速度を用いで密度を移流

上記が StableFluid のシミュレーションステップになります。

4.9 結果

実行して、マウスでスクリーン上をドラッグすると、以下の様な流体シミュレーションを起こす事が可能です。

▲図 4.2　実行例

4.10　まとめ

流体シミュレーションは、プリレンダリングと違い、Unity の様なリアルタイムゲームエンジンにとっては負荷の高い分野です。しかし、GPU 演算能力の向上から、2 次元であればある程度の解像度でも耐えうる FPS が出せる様になってきました。また、途中で出てきた GPU にとって負荷の高い演算部分、ガウス・ザイデル反復法を別の処理で実装してみたり、速度場自体をカールノイズで代用してみたり等の工夫をすれば、より軽い演算での流体表現も可能になる事でしょう。

もしこの章をお読みいただいて、少しでも流体に興味を持たれた方は、ぜひ次章の「粒子法による流体シミュレーション」にもトライして見て下さい。格子法とはまた違った角度から流体に迫れますので、流体シミュレーションの奥深さや実装の面白さを体験できる事かと思います。

4.11　参考

- Jos Stam. SIGGRAPH 1999. Stable Fluids

第 5 章

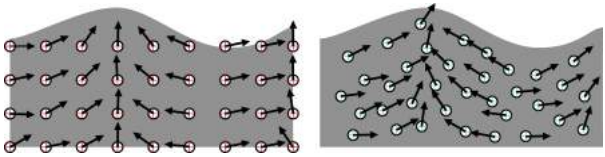
SPH 法による流体シミュレーション

前章では、格子法による流体シミュレーションの作成方法について解説しました。本章では、もう一つの流体のシミュレーション方法である粒子法、特に SPH 法を用いて流体の動きを表現していきます。多少噛み砕いて説明を行っているので、不十分な表現などありますがご了承ください。

5.1 基礎知識

5.1.1 オイラー的視点とラグランジュ的視点

流体の動きの観測方法として、オイラー的視点とラグランジュ的視点というものが存在します。オイラー的視点とは、流体に等間隔で観測点を固定し、その観測点での流体の動きを解析するものです。一方、ラグランジュ的視点とは、流体の流れに沿って動く観測点を浮かべ、その観測点での流体の動きを観測するものとなります (図 5.1 参照)。基本的に、オイラー的視点を用いた流体シミュレーション手法のことを格子法、ラグランジュ的視点を用いた流体シミュレーション手法のことを粒子法と呼びます。



▲図 5.1　左:オイラー的、右:ラグランジュ的

5.1.2 ラグランジュ微分 (物質微分)

オイラー的視点とラグランジュ的視点では、微分の演算の仕方が異なります。はじめに、オイラー的視点で表された物理量[*1]を以下に示してみます。

$$\phi = \phi(\vec{x}, t)$$

これは、時刻 t で位置 $\vec{x}$ にある物理量 ϕ という意味になります。この物理量の時間微分は、

$$\frac{\partial \phi}{\partial t}$$

と表せます。もちろんこれは、物理量の位置が $\vec{x}$ で固定されていますので、オイラー的視点での微分になります。

一方、ラグランジュ的視点では、観測点を流れに沿って移動[*2]させますので、観測点自体も時間の関数となっています。そのため、初期状態で $\vec{x}_0$ にあった観測点は、時刻 t で

$$\vec{x}(\vec{x}_0, t)$$

に存在します。 よって物理量の表記も

$$\phi = \phi(\vec{x}(\vec{x}_0, t), t)$$

となります。微分の定義に従って、現在の物理量と Δt 秒後の物理量の変化量を見てみると

$$\lim_{\Delta t \to 0} \frac{\phi(\vec{x}(\vec{x}_0, t + \Delta t), t + \Delta t) - \phi(\vec{x}(\vec{x}_0, t), t)}{\Delta t}$$

$$= \sum_i \frac{\partial \phi}{\partial x_i} \frac{\partial x_i}{\partial t} + \frac{\partial \phi}{\partial t}$$

$$= \left(\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} + \frac{\partial}{\partial t} \right) \phi$$

[*1] 物理量とは、観測できる速度や質量などのことを指します。 端的には「単位が有るもの」と捉えて良いでしょう。

[*2] 流れに沿った観測点の移動のことを、移流と呼びます。

$$= (\frac{\partial}{\partial t} + \overrightarrow{u} \cdot grad)\phi$$

となります。これが、観測点の移動を考慮した物理量の時間微分となります。しかしながら、この表記を用いていては式が複雑になりますので、

$$\frac{D}{Dt} := \frac{\partial}{\partial t} + \overrightarrow{u} \cdot grad$$

という演算子を導入することで、短く表すことができます。これら、観測点の移動を考慮した一連の操作を、ラグランジュ微分と呼びます。一見複雑そうに見えますが、観測点が移動する粒子法では、ラグランジュ的視点で式を表した方が都合が良くなります。

5.1.3 流体の非圧縮条件

流体は、流体の速度が音速よりも十分に小さい場合、体積の変化が起きないとみなすことができます。これは流体の非圧縮条件と呼ばれ、以下の数式で表されます。

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

これは、流体内で湧き出しや消失がないことを示しています。この式の導出には少し複雑な積分が入りますので、説明は割愛[*3]します。「流体は圧縮しない！」程度に捉えておいてください。

5.2 粒子法シミュレーション

粒子法では、流体を小さな粒子によって分割し、ラグランジュ的視点で流体の動きを観測します。この粒子が、前節の観測点にあたります。 一口に「粒子法」といっても、現在では多くの手法が提案されており、有名なものとして

- Smoothed Particle Hydrodynamics(SPH) 法
- Fluid Implicit Particle (FLIP) 法
- Particle In Cell (PIC) 法
- Moving Particle Semi-implicit (MPS) 法
- Material Point Method (MPM) 法

などがあります。

[*3] "Fluid Simulation for Computer Graphics - Robert Bridson" で詳しく解説されています。

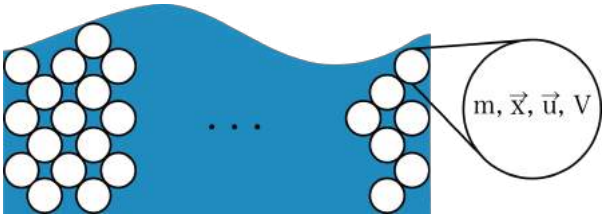
5.2.1 粒子法におけるナビエ・ストークス方程式の導出

はじめに、粒子法におけるナビエ・ストークス方程式 (以下 NS 方程式) は、以下のように記述されます。

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla\cdot\nabla\vec{u} + \vec{g}$$

前章の格子法で出てきた NS 方程式とは少し形が異なりますね。移流項がまるまる抜けてしまっていますが、先程のオイラー微分とラグランジュ微分の関係を見てみると、うまくこの形に変形できることがわかります。粒子法では観測点を流れに沿って移動させますから、NS 方程式計算時に移流項を考慮する必要がありません。移流の計算は NS 方程式で算出した加速度をもとに粒子位置を直接更新することで済ませる事ができます。

現実の流体は分子の集まりですので、ある種のパーティクルシステムであると言うことができます。しかし、コンピュータで実際の分子の数の計算を行うのは不可能ですので、計算可能な大きさに調節してあげる必要があります。図 5.2 に示されているそれぞれの粒 (*4) は、計算可能な大きさで分割した流体の一部分を表していています。これらの粒は、それぞれ質量 m 、位置ベクトル $\vec{x}$ 、速度ベクトル $\vec{u}$ 、体積 V を持つと考えることができます。



▲図 5.2　流体のパーティクル近似

これらそれぞれの粒について、外から受けた力 $\vec{f}$ を計算し、運動方程式 $m\vec{a} = \vec{f}$ を解くことで加速度が算出され、次のタイムステップでどのように移動するかを決めることができます。

前述の通り、それぞれの粒子は周りから何らかの力を受けて動きますが、その「力」とは一体何でしょうか。簡単な例として、重力 $m\vec{g}$ があげられますが、それ以外に

*4 英語では'Blob' と呼ばれます

周りの粒子からも何らかの力を受けるはずです。これらの力について、以下に解説します。

圧力

流体粒子にかかる力の 1 つ目は、圧力です。流体は必ず圧力の高い方から低い方に向かって流れます。もし圧力がどの方向からも同じだけかかっていたとすると、力は打ち消されて動きが止まってしまいますから、圧力のバランスが不均一である場合を考えます。前章で述べられたように、圧力のスカラー場の勾配を取ることで、自分の粒子位置から見て最も圧力上昇率の高い方向を算出することができます。粒子が力を受ける方向は、圧力の高い方から低い方ですので、マイナスを取って $-\nabla p$ となります。また、粒子は体積を持っていますから、粒子にかかる圧力は、 $-\nabla p$ に粒子の体積をかけて算出します[*5]。最終的に、 $-V\nabla p$ という結果が導出されます。

粘性力

流体粒子にかかる力の 2 つ目は、粘性力です。粘性 (ねばりけ) のある流体とは、はちみつや溶かしたチョコレートなどに代表される、変形しづらい流体のことを指します。粘性があるという言葉を粒子法の表現に当てはめてみると、粒子の速度は、周りの粒子速度の平均をとりやすいということになります。前章で述べられた通り、周囲の平均をとるという演算は、ラプラシアンを用いて行うことができます。

粘性の度合いを動粘性係数 μ を用いて表すと、 $\mu\nabla\cdot\nabla\overrightarrow{u}$ と表す事ができます。

圧力・粘性力・外力の統合

これらの力を運動方程式 $m\overrightarrow{a}=\overrightarrow{f}$ に当てはめて整理すると、

$$m\frac{D\overrightarrow{u}}{Dt}=-V\nabla p+V\mu\nabla\cdot\nabla\overrightarrow{u}+m\overrightarrow{g}$$

ここで、 m は ρV であることから、変形して (V が打ち消されます)

$$\rho\frac{D\overrightarrow{u}}{Dt}=-\nabla p+\mu\nabla\cdot\nabla\overrightarrow{u}+\rho\overrightarrow{g}$$

両辺 ρ で割り、

$$\frac{D\overrightarrow{u}}{Dt}=-\frac{1}{\rho}\nabla p+\frac{\mu}{\rho}\nabla\cdot\nabla\overrightarrow{u}+\overrightarrow{g}$$

最後に、粘性項の係数 $\dfrac{\mu}{\rho}$ に ν を導入して、

[*5] 流体の非圧縮条件により、単に体積をかけるだけで粒子にかかる圧力の積分を表すことができます。

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla\cdot\nabla\vec{u} + \vec{g}$$

となり、はじめに挙げた NS 方程式を導出することができました。

5.2.2 粒子法における移流の表現

粒子法では、粒子自体が流体の観測点を表現しているので、移流項の計算は単に粒子位置を移動させるだけで完了します。実際の時間微分の計算では、無限に小さい時間を用いますが、コンピューターでの計算では無限を表現できないため、十分小さい時間 Δt を用いて微分を表現します。これを差分と言い、 Δt を小さくすればするほど、正確な計算を行うことができます。

加速度について、差分の表現を導入すると、

$$\vec{a} = \frac{D\vec{u}}{Dt} \equiv \frac{\Delta\vec{u}}{\Delta t}$$

となります。よって速度の増分 $\Delta\vec{u}$ は、

$$\Delta\vec{u} = \Delta t\vec{a}$$

となり、また、位置の増分についても同様に、

$$\vec{u} = \frac{\partial\vec{x}}{\partial t} \equiv \frac{\Delta\vec{x}}{\Delta t}$$

より、

$$\Delta\vec{x} = \Delta t\vec{u}$$

となります。

この結果を利用することで、次のフレームでの速度ベクトルと位置ベクトルを算出できます。現在のフレームでの粒子速度が $\vec{u}_n$ であるとすると、次のフレームでの粒子速度は $\vec{u}_{n+1}$ で、

$$\vec{u}_{n+1} = \vec{u}_n + \Delta\vec{u} = \vec{u}_n + \Delta t\vec{a}$$

と表せます。

現在のフレームでの粒子位置が $\vec{x}_n$ であるとすると、次のフレームでの粒子位置は $\vec{x}_{n+1}$ で、

$$\vec{x}_{n+1} = \vec{x}_n + \Delta\vec{x} = \vec{x}_n + \Delta t\vec{u}$$

と表せます。

この手法は、前進オイラー法と呼ばれます。これを毎フレーム繰り返すことで、各時刻での粒子の移動を表現することができます。

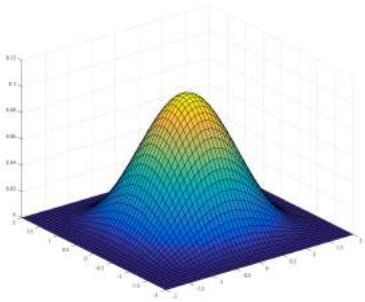
5.3 SPH 法による流体シミュレーション

前節では、粒子法における NS 方程式の導出方法について解説しました。もちろん、これらの微分方程式をコンピュータでそのまま解くことはできませんので、何らかの近似をしてあげる必要があります。その手法として、CG 分野でよく用いられる **SPH** 法について解説します。

SPH 法は、本来宇宙物理学における天体同士の衝突シミュレーションに用いられていた手法ですが、1996 年に Desbrun ら[*6]によって CG における流体シミュレーションにも応用されました。また、並列化も容易で、現在の GPU では大量の粒子の計算をリアルタイムに行うことが可能です。コンピュータシミュレーションでは、連続的な物理量を離散化して計算を行う必要がありますが、この離散化を、重み関数と呼ばれる関数を用いて行う手法を SPH 法と呼びます。

5.3.1 物理量の離散化

SPH 法では、粒子一つ一つが影響範囲を持っていて、他の粒子と距離が近いほどその粒子の影響が大きく受けるという動作をします。この影響範囲を図示すると図 5.3 のようになります。



▲図 5.3　2 次元の重み関数

[*6] Desbrun and Cani, Smoothed Particles: A new paradigm for animating highly deformable bodies, Eurographics Workshop on Computer Animation and Simulation (EGCAS), 1996.

この関数を重み関数[*7]と呼びます。

SPH 法における物理量を ϕ とすると、重み関数を用いて以下のように離散化されます。

$$\phi(\overrightarrow{x}) = \sum_{j \in N} m_j \frac{\phi_j}{\rho_j} W(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

N, m, ρ, h はそれぞれ、近傍粒子の集合、粒子の質量、粒子の密度、重み関数の影響半径です。また、関数 W が先ほど述べた重み関数になります。

さらに、この物理量には、勾配とラプラシアンなどの偏微分演算が適用でき、勾配は、

$$\nabla\phi(\overrightarrow{x}) = \sum_{j \in N} m_j \frac{\phi_j}{\rho_j} \nabla W(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

ラプラシアンは、

$$\nabla^2\phi(\overrightarrow{x}) = \sum_{j \in N} m_j \frac{\phi_j}{\rho_j} \nabla^2 W(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

と表せます。式からわかるように、物理量の勾配及びラプラシアンは、重み関数に対してのみ適用されるイメージになります。重み関数 W は、求めたい物理量によって異なるものを使用しまが、この理由の説明については割愛[*8]します。

5.3.2 密度の離散化

流体の粒子の密度は、先ほどの重み関数で離散化した物理量の式を利用して、

$$\rho(\overrightarrow{x}) = \sum_{j \in N} m_j W_{poly6}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

と与えられます。ここで、利用する重み関数 W は、以下で与えられます。

$$W_{poly6}(\overrightarrow{r}, h) = \frac{4}{\pi h^8} \begin{cases} (h^2 - \|\overrightarrow{r}\|^2)^3 & 0 \leq \|r\| \leq h \\ 0 & otherwise \end{cases}$$

▲図 5.4　Poly6 重み関数

[*7] 通常この関数はカーネル関数とも呼ばれますが、ComputeShader におけるカーネル関数と区別するためこの呼び方にしています。

[*8] "CG のための物理シミュレーションの基礎 - 藤澤誠" で詳しく解説されています。

5.3.3 粘性項の離散化

粘性項を離散化も密度の場合と同様重み関数を利用して、

$$f_i^{visc} = \mu \nabla^2 \overrightarrow{u}_i = \mu \sum_{j \in N} m_j \frac{\overrightarrow{u}_j - \overrightarrow{u}_i}{\rho_j} \nabla^2 W_{visc}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

と表されます。ここで、重み関数のラプラシアン $\nabla^2 W_{visc}$ は、以下で与えられます。

$$\nabla^2 W_{visc}(\overrightarrow{r}, h) = \frac{20}{3\pi h^5} \begin{cases} h - \|\overrightarrow{r}\| & 0 \leq \|r\| \leq h \\ 0 & otherwise \end{cases}$$

▲図 5.5　Viscosity 重み関数のラプラシアン

5.3.4 圧力項の離散化

同様に、圧力項を離散化していきます。

$$f_i^{press} = -\frac{1}{\rho_i} \nabla p_i = -\frac{1}{\rho_i} \sum_{j \in N} m_j \frac{p_j - p_i}{2\rho_j} \nabla W_{spiky}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

ここで、重み関数の勾配 W_{spiky} は以下で与えられます。

$$\nabla W_{spiky}(\overrightarrow{r}, h) = -\frac{30}{\pi h^5} \begin{cases} (h - \|\overrightarrow{r}\|)^2 \frac{\overrightarrow{r}}{\|r\|} & 0 \leq \|r\| \leq h \\ 0 & otherwise \end{cases}$$

▲図 5.6　Spiky 重み関数の勾配

この時、粒子の圧力は事前に、Tait 方程式と呼ばれる、

$$p = B \left\{ \left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right\}$$

で算出されています。 ここで、 B は気体定数です。非圧縮性を保証するためには、本来ポアソン方程式を解かなければならないのですが、リアルタイム計算には向きま

せん。その代わり SPH 法[*9]では、近似的に非圧縮性を確保する点で格子法よりも圧力項の計算が苦手であるといわれます。

5.4 SPH 法の実装

サンプルはこちらのリポジトリ (https://github.com/IndieVisualLab/UnityGraphicsProgramming) の Assets/SPHFluid 以下に掲載しています。今回の実装では、極力シンプルに SPH の手法を解説するために高速化や数値安定性は考慮していませんのでご了承ください。

5.4.1 パラメータ

シミュレーションに使用する諸々のパラメータの説明については、コード内コメントに記載しています。

▼リスト 5.1　シミュレーションに使用するパラメータ (FluidBase.cs)

```
 1: NumParticleEnum particleNum = NumParticleEnum.NUM_8K;    // 粒子数
 2: float smoothlen = 0.012f;                // 粒子半径
 3: float pressureStiffness = 200.0f;        // 圧力項係数
 4: float restDensity = 1000.0f;             // 静止密度
 5: float particleMass = 0.0002f;            // 粒子質量
 6: float viscosity = 0.1f;                  // 粘性係数
 7: float maxAllowableTimestep = 0.005f;     // 時間刻み幅
 8: float wallStiffness = 3000.0f;           // ペナルティ法の壁の力
 9: int iterations = 4;                      // イテレーション回数
10: Vector2 gravity = new Vector2(0.0f, -0.5f);     // 重力
11: Vector2 range = new Vector2(1, 1);              // シミュレーション空間
12: bool simulate = true;                           // 実行 or 一時停止
13:
14: int numParticles;               // パーティクルの個数
15: float timeStep;                 // 時間刻み幅
16: float densityCoef;              // Poly6 カーネルの密度係数
17: float gradPressureCoef;         // Spiky カーネルの圧力係数
18: float lapViscosityCoef;         // Laplacian カーネルの粘性係数
```

今回のデモシーンでは、コードに記載されているパラメータの初期化値とは異なる値をインスペクタで設定していますので注意してください。

5.4.2 SPH 重み関数の係数の計算

重み関数の係数はシミュレーション中で変化しないため、初期化時に CPU 側で計算しておきます。(ただし、実行途中でパラメータを編集する可能性も踏まえて

[*9] Tait 方程式を用いた圧力計算を行う SPH 法を、特別に WCSPH 法と呼びます。

Update 関数内で更新しています)

今回、粒子ごとの質量はすべて一定にしているので、物理量の式内にある質量 m はシグマの外に出て以下になります。

$$\phi(\overrightarrow{x}) = m \sum_{j \in N} \frac{\phi_j}{\rho_j} W(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

そのため、係数計算の中に質量を含めてしまうことができます。

重み関数の種類で係数も変化してきますから、それぞれに関して係数を計算します。

▼リスト 5.2　重み関数の係数の事前計算 (FluidBase.cs)

```
densityCoef = particleMass * 4f / (Mathf.PI * Mathf.Pow(smoothlen, 8));
gradPressureCoef
    = particleMass * -30.0f / (Mathf.PI * Mathf.Pow(smoothlen, 5));
lapViscosityCoef
    = particleMass * 20f / (3 * Mathf.PI * Mathf.Pow(smoothlen, 5));
```

最終的に、これらの CPU 側で計算した係数 (及び各種パラメータ) を GPU 側の定数バッファに格納します。

▼リスト 5.3　ComputeShader の定数バッファに値を転送する (FluidBase.cs)

```
fluidCS.SetInt("_NumParticles", numParticles);
fluidCS.SetFloat("_TimeStep", timeStep);
fluidCS.SetFloat("_Smoothlen", smoothlen);
fluidCS.SetFloat("_PressureStiffness", pressureStiffness);
fluidCS.SetFloat("_RestDensity", restDensity);
fluidCS.SetFloat("_Viscosity", viscosity);
fluidCS.SetFloat("_DensityCoef", densityCoef);
fluidCS.SetFloat("_GradPressureCoef", gradPressureCoef);
fluidCS.SetFloat("_LapViscosityCoef", lapViscosityCoef);
fluidCS.SetFloat("_WallStiffness", wallStiffness);
fluidCS.SetVector("_Range", range);
fluidCS.SetVector("_Gravity", gravity);
```

▼リスト 5.4　ComputeShader の定数バッファ (SPH2D.compute)

```
int   _NumParticles;      // 粒子数
float _TimeStep;          // 時間刻み幅 (dt)
float _Smoothlen;         // 粒子半径
float _PressureStiffness; // Becker の係数
float _RestDensity;       // 静止密度
float _DensityCoef;       // 密度算出時の係数
float _GradPressureCoef;  // 圧力算出時の係数
float _LapViscosityCoef;  // 粘性算出時の係数
float _WallStiffness;     // ペナルティ法の押し返す力
float _Viscosity;         // 粘性係数
```

```
11: float2 _Gravity;          // 重力
12: float2 _Range;            // シミュレーション空間
13:
14: float3 _MousePos;         // マウス位置
15: float _MouseRadius;       // マウスインタラクションの半径
16: bool _MouseDown;          // マウスが押されているか
```

5.4.3 密度の計算

▼リスト 5.5　密度の計算を行うカーネル関数 (SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void DensityCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x;     // 現在処理しているパーティクル ID
 4:
 5:     float h_sq = _Smoothlen * _Smoothlen;
 6:     float2 P_position = _ParticlesBufferRead[P_ID].position;
 7:
 8:     // 近傍探索 (O(n^2))
 9:     float density = 0;
10:     for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
11:             if (N_ID == P_ID) continue;     // 自身の参照回避
12:
13:             float2 N_position = _ParticlesBufferRead[N_ID].position;
14:
15:             float2 diff = N_position - P_position;    // 粒子距離
16:             float r_sq = dot(diff, diff);             // 粒子距離の 2 乗
17:
18:             // 半径内に収まっていない粒子は除外
19:             if (r_sq < h_sq) {
20:             // 計算には 2 乗しか含まれないのでルートをとる必要なし
21:                     density += CalculateDensity(r_sq);
22:             }
23:     }
24:
25:     // 密度バッファを更新
26:     _ParticlesDensityBufferWrite[P_ID].density = density;
27: }
```

本来であれば粒子を全数調査せず、適切な近傍探索アルゴリズムを用いて近傍粒子を探す必要がありますが、今回の実装では簡単のために全数調査を行っています (10 行目の for ループ)。また、自分と相手粒子との距離計算を行うため、11 行目で自身の粒子同士で計算を行うのを回避しています。

重み関数の有効半径 h による場合分けは 19 行目の if 文で実現します。密度の足し合わせ (シグマの計算) は、9 行目で 0 で初期化しておいた変数に対してシグマ内部の計算結果を加算していくことで実現します。ここで、もう一度密度の計算式を示します。

$$\rho(\overrightarrow{x}) = \sum_{j \in N} m_j W_{poly6}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

密度の計算は上式のとおり、Poly6 重み関数を用います。 Poly6 重み関数はリスト 5.6 で計算します。

▼リスト 5.6　密度の計算 (SPH2D.compute)

```
1: inline float CalculateDensity(float r_sq) {
2:     const float h_sq = _Smoothlen * _Smoothlen;
3:     return _DensityCoef * (h_sq - r_sq) * (h_sq - r_sq) * (h_sq - r_sq);
4: }
```

最終的にリスト 5.5 の 25 行目で書き込み用バッファに書き込みます。

5.4.4　粒子単位の圧力の計算

▼リスト 5.7　粒子毎の圧力を計算する重み関数 (SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void PressureCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x;     // 現在処理しているパーティクル ID
 4:
 5:     float  P_density = _ParticlesDensityBufferRead[P_ID].density;
 6:     float  P_pressure = CalculatePressure(P_density);
 7:
 8:     // 圧力バッファを更新
 9:     _ParticlesPressureBufferWrite[P_ID].pressure = P_pressure;
10: }
```

圧力項を解く前に、粒子単位の圧力を算出しておき、後の圧力項の計算コストを下げます。先程も述べましたが、圧力の計算では本来、以下の式のようなポアソン方程式と呼ばれる方程式を解く必要があります。

$$\nabla^2 p = \rho \frac{\nabla \overrightarrow{u}}{\Delta t}$$

しかし、コンピュータで正確にポアソン方程式を解く操作は非常に計算コストが高いため、以下の Tait 方程式を用いて近似的に求めます。

$$p = B \left\{ \left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right\}$$

▼リスト 5.8　Tait 方程式の実装 (SPH2D.compute)

```
1: inline float CalculatePressure(float density) {
2:     return _PressureStiffness * max(pow(density / _RestDensity, 7) - 1, 0);
3: }
```

5.4.5 圧力項・粘性項の計算

▼リスト 5.9　圧力項・粘性項を計算するカーネル関数 (SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void ForceCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x; // 現在処理しているパーティクル ID
 4:
 5:     float2 P_position = _ParticlesBufferRead[P_ID].position;
 6:     float2 P_velocity = _ParticlesBufferRead[P_ID].velocity;
 7:     float  P_density = _ParticlesDensityBufferRead[P_ID].density;
 8:     float  P_pressure = _ParticlesPressureBufferRead[P_ID].pressure;
 9:
10:     const float h_sq = _Smoothlen * _Smoothlen;
11:
12:     // 近傍探索 (O(n^2))
13:     float2 press = float2(0, 0);
14:     float2 visco = float2(0, 0);
15:     for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
16:             if (N_ID == P_ID) continue;     // 自身を対象とした場合スキップ
17:
18:             float2 N_position = _ParticlesBufferRead[N_ID].position;
19:
20:             float2 diff = N_position - P_position;
21:             float r_sq = dot(diff, diff);
22:
23:             // 半径内に収まっていない粒子は除外
24:             if (r_sq < h_sq) {
25:                     float  N_density
26:                     = _ParticlesDensityBufferRead[N_ID].density;
27:                     float  N_pressure
28:                     = _ParticlesPressureBufferRead[N_ID].pressure;
29:                     float2 N_velocity
30:                     = _ParticlesBufferRead[N_ID].velocity;
31:                     float  r = sqrt(r_sq);
32:
33:                     // 圧力項
34:                     press += CalculateGradPressure(...);
35:
36:                     // 粘性項
37:                     visco += CalculateLapVelocity(...);
38:             }
39:     }
40:
41:     // 統合
42:     float2 force = press + _Viscosity * visco;
43:
44:     // 加速度バッファの更新
45:     _ParticlesForceBufferWrite[P_ID].acceleration = force / P_density;
```

```
46: }
```

圧力項、粘性項の計算も、密度の計算方法と同様に行います。
初めに、以下の圧力項による力の計算を 31 行目にて行っています。

$$f_i^{press} = -\frac{1}{\rho_i}\nabla p_i = -\frac{1}{\rho_i}\sum_{j \in N} m_j \frac{p_j - p_i}{2\rho_j}\nabla W_{press}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

シグマの中身の計算は以下の関数で行われます。

▼リスト 5.10　圧力項の要素の計算 (SPH2D.compute)

```
1: inline float2 CalculateGradPressure(...) {
2:     const float h = _Smoothlen;
3:     float avg_pressure = 0.5f * (N_pressure + P_pressure);
4:     return _GradPressureCoef * avg_pressure / N_density
5:             * pow(h - r, 2) / r * (diff);
6: }
```

次に、以下の粘性項による力の計算を 34 行目で行っています。

$$f_i^{visc} = \mu\nabla^2 \overrightarrow{u}_i = \mu \sum_{j \in N} m_j \frac{\overrightarrow{u}_j - \overrightarrow{u}_i}{\rho_j}\nabla^2 W_{visc}(\overrightarrow{x_j} - \overrightarrow{x}, h)$$

シグマの中身の計算は以下の関数で行われます。

▼リスト 5.11　粘性項の要素の計算 (SPH2D.compute)

```
1: inline float2 CalculateLapVelocity(...) {
2:     const float h = _Smoothlen;
3:     float2 vel_diff = (N_velocity - P_velocity);
4:     return _LapViscosityCoef / N_density * (h - r) * vel_diff;
5: }
```

最後に、リスト 5.9 の 39 行目にて圧力項と粘性項で算出した力を足し合わせ、最終的な出力としてバッファに書き込んでいます。

5.4.6　衝突判定と位置更新

▼リスト 5.12　衝突判定と位置更新を行うカーネル関数 (SPH2D.compute)

```
1: [numthreads(THREAD_SIZE_X, 1, 1)]
2: void IntegrateCS(uint3 DTid : SV_DispatchThreadID) {
3:     const unsigned int P_ID = DTid.x; // 現在処理しているパーティクル ID
```

```
 4:
 5:     // 更新前の位置と速度
 6:     float2 position = _ParticlesBufferRead[P_ID].position;
 7:     float2 velocity = _ParticlesBufferRead[P_ID].velocity;
 8:     float2 acceleration = _ParticlesForceBufferRead[P_ID].acceleration;
 9:
10:     // マウスインタラクション
11:     if (distance(position, _MousePos.xy) < _MouseRadius && _MouseDown) {
12:             float2 dir = position - _MousePos.xy;
13:             float pushBack = _MouseRadius-length(dir);
14:             acceleration += 100 * pushBack * normalize(dir);
15:     }
16:
17:     // 衝突判定を書くならここ -----
18:
19:     // 壁境界 (ペナルティ法)
20:     float dist = dot(float3(position, 1), float3(1, 0, 0));
21:     acceleration += min(dist, 0) * -_WallStiffness * float2(1, 0);
22:
23:     dist = dot(float3(position, 1), float3(0, 1, 0));
24:     acceleration += min(dist, 0) * -_WallStiffness * float2(0, 1);
25:
26:     dist = dot(float3(position, 1), float3(-1, 0, _Range.x));
27:     acceleration += min(dist, 0) * -_WallStiffness * float2(-1, 0);
28:
29:     dist = dot(float3(position, 1), float3(0, -1, _Range.y));
30:     acceleration += min(dist, 0) * -_WallStiffness * float2(0, -1);
31:
32:     // 重力の加算
33:     acceleration += _Gravity;
34:
35:     // 前進オイラー法で次の粒子位置を更新
36:     velocity += _TimeStep * acceleration;
37:     position += _TimeStep * velocity;
38:
39:     // パーティクルのバッファ更新
40:     _ParticlesBufferWrite[P_ID].position = position;
41:     _ParticlesBufferWrite[P_ID].velocity = velocity;
42: }
```

壁との衝突判定をペナルティ法を用いて行います (19-30 行目)。ペナルティ法とは、境界位置からはみ出した分だけ強い力で押し返すという手法になります。

本来は壁との衝突判定の前に障害物との衝突判定も行うのですが、今回の実装ではマウスとのインタラクションを行うようにしています (213-218 行目)。マウスが押されていれば、指定された力でマウス位置から遠ざかるような力を加えています。

33 行目にて外力である重力を加算しています。重力の値をゼロにすると無重力状態になり、面白い視覚効果が得られます。また、位置の更新は前述の前進オイラー法で行い (36-37 行目)、最終的な結果をバッファに書き込みます。

5.4.7 シミュレーションメインルーチン

▼リスト 5.13　シミュレーションの主要関数 (FluidBase.cs)

```
 1: private void RunFluidSolver() {
 2:
 3:   int kernelID = -1;
 4:   int threadGroupsX = numParticles / THREAD_SIZE_X;
 5:
 6:   // Density
 7:   kernelID = fluidCS.FindKernel("DensityCS");
 8:   fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
 9:   fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferWrite", ...);
10:   fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);
11:
12:   // Pressure
13:   kernelID = fluidCS.FindKernel("PressureCS");
14:   fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferRead", ...);
15:   fluidCS.SetBuffer(kernelID, "_ParticlesPressureBufferWrite", ...);
16:   fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);
17:
18:   // Force
19:   kernelID = fluidCS.FindKernel("ForceCS");
20:   fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
21:   fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferRead", ...);
22:   fluidCS.SetBuffer(kernelID, "_ParticlesPressureBufferRead", ...);
23:   fluidCS.SetBuffer(kernelID, "_ParticlesForceBufferWrite", ...);
24:   fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);
25:
26:   // Integrate
27:   kernelID = fluidCS.FindKernel("IntegrateCS");
28:   fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
29:   fluidCS.SetBuffer(kernelID, "_ParticlesForceBufferRead", ...);
30:   fluidCS.SetBuffer(kernelID, "_ParticlesBufferWrite", ...);
31:   fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);
32:
33:   SwapComputeBuffer(ref particlesBufferRead, ref particlesBufferWrite);
34: }
```

これまでに述べた ComputeShader のカーネル関数を、毎フレーム呼び出す部分です。それぞれのカーネル関数に対して適切な ComputeBuffer を与えてあげます。

ここで、タイムステップ幅 Δt を小さくすればするほどシミュレーションの誤差が出にくくなることを思い出してみてください。60FPS で実行する場合、 $\Delta t = 1/60$ となりますが、これでは誤差が大きく出てしまい粒子が爆発してしまいます。さらに、 $\Delta t = 1/60$ より小さいタイムステップ幅をとると、1 フレーム当たりの時間の進み方が実時間より遅くなり、スローモーションになってしまいます。これを回避するには、 $\Delta t = 1/(60 \times iterarion)$ として、メインルーチンを 1 フレームにつき iterarion 回回します。

▼リスト 5.14　主要関数のイテレーション (FluidBase.cs)

```
1: // 計算精度を上げるために時間刻み幅を小さくして複数回イテレーションする
2: for (int i = 0; i<iterations; i++) {
3:     RunFluidSolver();
4: }
```

こうすることで、小さいタイムステップ幅で実時間のシミュレーションを行うことができます。

5.4.8　バッファの使い方

通常のシングルアクセスのパーティクルシステムとは異なり、粒子同士が相互作用しますから、計算途中に他のデータが書き換わってしまっては困ります。これを回避するために、GPU で計算を行っている際に値を書き換えない読み込み用バッファと書き込み用バッファの 2 つを用意します。これらのバッファを毎フレーム入れ替えることで、競合なくデータを更新できます。

▼リスト 5.15　バッファを入れ替える関数 (FluidBase.cs)

```
1: void SwapComputeBuffer(ref ComputeBuffer ping, ref ComputeBuffer pong) {
2:     ComputeBuffer temp = ping;
3:     ping = pong;
4:     pong = temp;
5: }
```

5.4.9　粒子のレンダリング

▼リスト 5.16　パーティクルのレンダリング (FluidRenderer.cs)

```
 1: void DrawParticle() {
 2:
 3:   Material m = RenderParticleMat;
 4:
 5:   var inverseViewMatrix = Camera.main.worldToCameraMatrix.inverse;
 6:
 7:   m.SetPass(0);
 8:   m.SetMatrix("_InverseMatrix", inverseViewMatrix);
 9:   m.SetColor("_WaterColor", WaterColor);
10:   m.SetBuffer("_ParticlesBuffer", solver.ParticlesBufferRead);
11:   Graphics.DrawProcedural(MeshTopology.Points, solver.NumParticles);
12: }
```

10 行目にて、流体粒子の位置計算結果を格納したバッファをマテリアルにセット

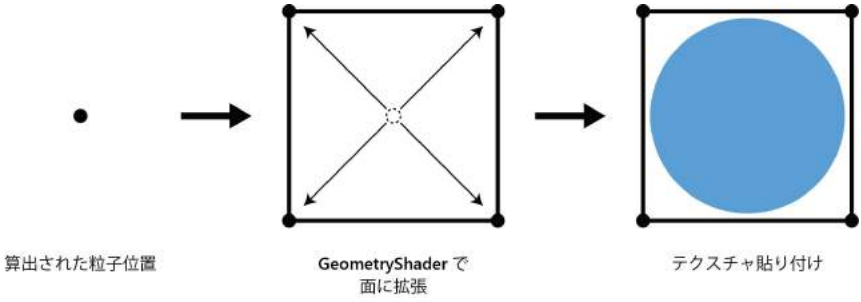
し、シェーダーに転送します。11 行目にて、パーティクルの個数分インスタンス描画をするよう命令しています。

▼リスト 5.17　パーティクルのレンダリング (Particle.shader)

```
 1: struct FluidParticle {
 2:     float2 position;
 3:     float2 velocity;
 4: };
 5:
 6: StructuredBuffer<FluidParticle> _ParticlesBuffer;
 7:
 8: // ------------------------------------------------------------------
 9: // Vertex Shader
10: // ------------------------------------------------------------------
11: v2g vert(uint id : SV_VertexID) {
12:
13:     v2g o = (v2g)0;
14:     o.pos = float3(_ParticlesBuffer[id].position.xy, 0);
15:     o.color = float4(0, 0.1, 0.1, 1);
16:     return o;
17: }
```

1-6 行目にて、流体粒子の情報を受け取るための情報の定義を行います。この時、スクリプトからマテリアルに転送したバッファの構造体と定義を一致させる必要があります。位置データの受け取りは、14 行目のように id：SV_VertexID でバッファの要素を参照することで行います。

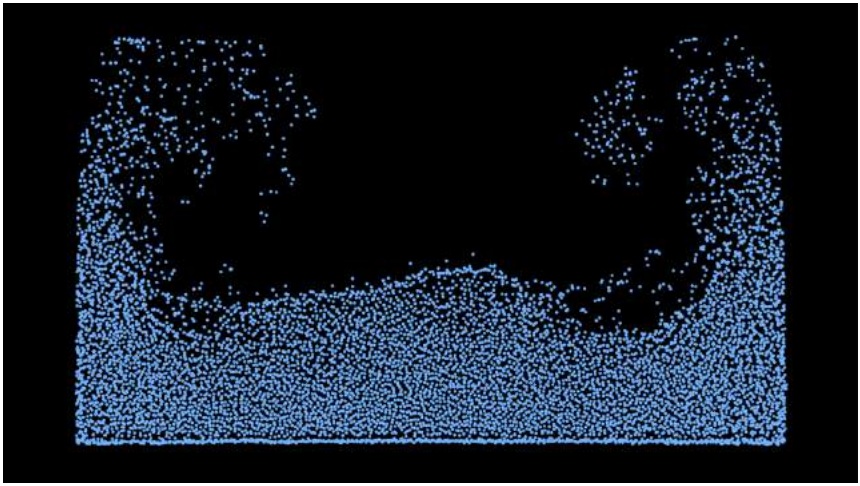
あとは通常のパーティクルシステムと同様、図 5.7 のようにジオメトリシェーダーで計算結果の位置データを中心としたビルボード[*10]を作成し、粒子画像をアタッチしてレンダリングします。



▲図 5.7　ビルボードの作成

[*10] 表が常に視点方向を向く Plane のことを指します。

5.5 結果



▲図 5.8　レンダリング結果

動画はこちら (https://youtu.be/KJVu26zeK2w) に掲載しています。

5.6 まとめ

本章では、SPH 法を用いた流体シミュレーションの手法を示しました。SPH 法を用いることで、流体の動きをパーティクルシステムのように汎用的に扱うことができるようになりました。

先述の通り、流体シミュレーションの手法は SPH 法以外にもたくさんの種類があります。本章を通して、他の流体シミュレーション手法に加え、他の物理シミュレーション自体についても興味を持っていただき、表現の幅を広げていただければ幸いです。

第 6 章

ジオメトリシェーダーで草を生やす

6.1 はじめに

本章ではレンダリングパイプラインのステージの一つである Geometry Shader(ジオメトリシェーダー) についての説明を主軸として、Geometry Shader を用いた動的な草生成シェーダー (俗に言う Grass Shader) を解説しています。

Geometry Shader の説明についてはいくらかの専門的用語を用いていますが、とりあえず Geometry Shader を使ってみたければサンプルコードを見て頂くのが手っ取り早いでしょう。

本章の Unity プロジェクトは以下の Github リポジトリにアップロードしてあります。

https://github.com/IndieVisualLab/UnityGraphicsProgramming/

6.2 Geometry Shader とは？

Geometry Shader とは、GPU 上で動的にプリミティブ (メッシュを構成する基本形状) の変換・生成・削除などが可能なプログラマブルシェーダーの一つです。

これまでプリミティブを変換するなど、動的にメッシュ形状を変化させようとすると、CPU 上で処理を行うか、事前に頂点にメタ情報を持たせておき Vertex Shader で変換するなどの工夫が必要でした。しかし、Vertex Shader では隣接する頂点に関する情報を取得することが出来ず、処理中の頂点を元に新しく頂点を生成したり、また逆に削除したりする事が出来ないなどの強い制約がありました。また、だからといって CPU で処理を行うと、リアルタイム処理という観点からすると非現実的なほど膨大な時間を要することになります。この様に、リアルタイムにメッシュを形状変化させることに関しては、今までいくつかの問題を抱えていました。

そこで、これらの問題を解決し、弱い制約の中で自由に変換処理を出来るようにするための機能として、DirectX10 や OpenGL3.2 にて標準搭載されたのが Geometry Shader です。なお、OpenGL では Primitive Shader とも呼ばれることがあります。

6.3 Geometry Shader の特徴

6.3.1 レンダリングパイプライン

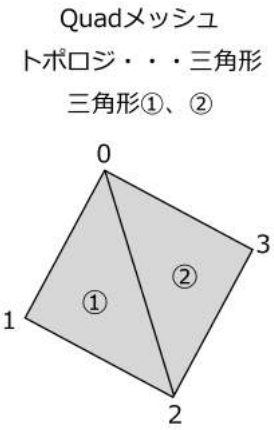
レンダリングパイプライン上では Vertex Shader の次、Fragment Shader やラスタライズ処理の前に位置しています。つまり、Fragment Shader 内では、Geometry Shader にて動的に生成した頂点と Vertex Shader に渡された元々の頂点とを区別せずに処理されます。

6.3.2 Geometry Shader への入力

通常 Vertex Shader への入力情報は頂点単位となっており、その頂点についての変換処理を行います。ですが、Geometry Shader への入力情報はユーザによって定義された入力用プリミティブ単位となります。

実際のプログラムは後述してありますが、Vertex Shader にて処理をした頂点情報群が、入力用プリミティブ型に基いて分割して入力されることになります。例えば入力のプリミティブ型を triangle とすれば 3 つの頂点情報が、line とすれば 2 つの頂点情報が、point とすれば 1 つの頂点情報が渡されます。これによって vertex shader では出来なかった、他の頂点情報を参照しながら処理を行なう事が可能となり、幅広い計算が出来るようになります。

なお一つ注意が必要な点として、Vertex Shader は頂点単位で処理が行われ、その処理する頂点についての情報が渡されますが、Geometry Shader は入力用プリミティブ型とは関係なく、プリミティブアセンブリのトポロジによって決定されるプリミティブを単位として処理が行われます。つまり、図 6.1 のようにトポロジが Triangles の Quad メッシュに Geometry Shader を実行する場合、Geometry Shader は三角形①と②について計 2 回実行されます。この時、入力用プリミティブ型を Line とした場合、入力に渡される情報は三角形①の時は頂点 0,1,2 のうちの二点の頂点、②の時は頂点 0,2,3 のうちの二点の頂点となります。



▲図 6.1　Quad メッシュ

6.3.3　Geometry Shader からの出力

Geometry Shader の出力はユーザ定義の出力用プリミティブ型用の頂点情報群となります。Vertex Shader では 1 入力 1 出力となっていましたが、Geometry Shader は複数の情報を出力する事になり、出力情報によって生成されるプリミティブは 1 つ以上でも問題ありません。

例えば出力プリミティブ型を triangle と定義した上で新しく計算によって求めた頂点を計 9 つ出力した場合は、3 つの三角形が Geometry Shader によって生成された事になります。この処理は前述の通りプリミティブ単位にて行われるため、元々 1 つだった三角形が 3 つに増えたとも考えられます。

また、Geometry Shader には MaxVertexCount という、一回の処理で最大何点の頂点を出力するかを事前に設定しておく必要があります。例えば MaxVertexCount を 9 と設定した場合は、Geometry Shader は 0 点 ~ 9 点までの頂点数を出力することが出来るようになります。この数値は後述する『Geometry Shader の制限』によって、一般的には 1024 が一応の最大値となります。

なお、頂点情報を出力する上で気を付けなければならない点として、元々のメッシュの形状を維持した状態で新しく頂点を追加する場合は、Vertex Shader から送られてきた頂点情報についても Geometry Shader にて出力する必要があります。Geometry Shader は Vertex Shader の出力に追加していくという挙動ではなく、Geometry Shader の出力がラスタライズ処理が行われ、Fragment Shader に渡され

ます。逆説的に言えば、Geometry Shader の出力を 0 にすることによって、動的に頂点数を減らすことも出来ます。

6.3.4 Geometry Shader の制限

Geometry Shader には 1 回の出力に関して、最大出力頂点数と最大出力要素数という制限があります。最大出力頂点数は文字通り頂点数の限界値であり、GPU に依存した数値ではありますが 1024 などが一般的なので、1 つの三角形から最大で 1024 点までしか頂点を増やすことが出来ます。最大出力要素数における要素とは座標や色などの頂点が持っている情報の事であり、一般的には (x, y, z, w) の位置要素と (r, g, b, a) の色要素の計 8 要素となります。この要素の最大出力数も GPU に依存しますが同じく 1024 が一般的なので、出力は最大でも 128(1024/8) に制限される事になります。

この二つの制限は両方を満たす必要があるため、頂点数的には 1024 点の出力が可能でも、要素数側の制約によって、実際の Geometry Shader の出力は 128 点までは限界となります。ですので、例えばプリミティブ数が 2 のメッシュ (Quad メッシュなど) に対して Geometry Shader を利用した場合は、最大でも 256 点 (128 点 * 2 プリミティブ) の頂点数までしか頂点を扱うことは出来ません。

この 128 点という数字が、前項の MaxVertexCount に設定できる数値の限界値となります。

6.4 簡単な Geometry Shader

以下にシンプルな挙動の Geometry Shader のプログラムが記載してあります。前項までの説明について実際のプログラムと照らし合わせながら改めて説明していきます。

なお、Geometry Shader 以外について、Unity でシェーダーを記述する際に必要な ShaderLab のシンタックスなどに関する説明は本章では省略しますので、もし分からない部分がありましたら下記の公式ドキュメントを参照してみてください。

https://docs.unity3d.com/ja/current/Manual/SL-Reference.html

▼

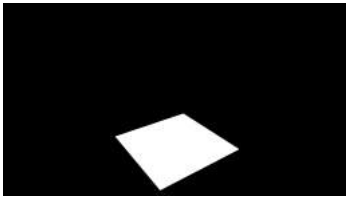
```
Shader "Custom/SimpleGeometryShader"
{
    Properties
    {
        _Height("Height", float) = 5.0
        _TopColor("Top Color", Color) = (0.0, 0.0, 1.0, 1.0)
        _BottomColor("Bottom Color", Color) = (1.0, 0.0, 0.0, 1.0)
    }
```

```
    SubShader
    {
        Tags { "RenderType" = "Opaque"}
        LOD 100

        Cull Off
        Lighting Off

        Pass
        {
            CGPROGRAM
            #pragma target 5.0
            #pragma vertex vert
            #pragma geometry geom
            #pragma fragment frag
            #include "UnityCG.cginc"

            uniform float _Height;
            uniform float4 _TopColor, _BottomColor;

            struct v2g
            {
                float4 pos : SV_POSITION;
            };

            struct g2f
            {
                float4 pos : SV_POSITION;
                float4 col : COLOR;
            };

            v2g vert(appdata_full v)
            {
                v2g o;
                o.pos = v.vertex;

                return o;
            }

            [maxvertexcount(12)]
            void geom(triangle v2g input[3],
                      inout TriangleStream<g2f> outStream)
            {
                float4 p0 = input[0].pos;
                float4 p1 = input[1].pos;
                float4 p2 = input[2].pos;

                float4 c = float4(0.0f, 0.0f, -_Height, 1.0f)
                            + (p0 + p1 + p2) * 0.33333f;

                g2f out0;
                out0.pos = UnityObjectToClipPos(p0);
                out0.col = _BottomColor;

                g2f out1;
                out1.pos = UnityObjectToClipPos(p1);
                out1.col = _BottomColor;
```

```
                g2f out2;
                out2.pos = UnityObjectToClipPos(p2);
                out2.col = _BottomColor;

                g2f o;
                o.pos = UnityObjectToClipPos(c);
                o.col = _TopColor;

                // bottom
                outStream.Append(out0);
                outStream.Append(out1);
                outStream.Append(out2);
                outStream.RestartStrip();

                // sides
                outStream.Append(out0);
                outStream.Append(out1);
                outStream.Append(o);
                outStream.RestartStrip();

                outStream.Append(out1);
                outStream.Append(out2);
                outStream.Append(o);
                outStream.RestartStrip();

                outStream.Append(out2);
                outStream.Append(out0);
                outStream.Append(o);
                outStream.RestartStrip();
            }

            float4 frag(g2f i) : COLOR
            {
                return i.col;
            }
            ENDCG
        }
    }
}
```

このシェーダーでは、渡された三角形の中心座標を計算してさらに上方向に移動させ、渡されてきた三角形の各頂点と計算して求めた新しい座標を接続させています。つまり、平面的な三角形から簡単な三角錐を生成していることになります。

なので、このシェーダーを Quad メッシュ (2 つの三角形から構成されている) に適用すると、図 6.2 から図 6.3 のようになります。



▲図 6.2　この様な平たい板から

▲図 6.3　立体的な二つの三角錐が表示されるようになります

このシェーダーの中で、特に Geometry Shader に関する部分だけを抜き出して説明していきます。

▼

```
#pragma target 5.0
#pragma vertex vert

// Geometry Shader の利用を宣言
#pragma geometry geom

#pragma fragment frag
#include "UnityCG.cginc"
```

上記の宣言部分にて、geomという名前の関数が Geometry Shader 用関数であることを宣言しています。これによって Geometry Shader ステージになった時に geom 関数が呼び出されるようになります。

▼

```
[maxvertexcount(12)]
void geom(triangle v2g input[3], inout TriangleStream<g2f> outStream)
```

これが Geometry Shader 用の関数宣言です。

6.4.1 入力

▼

```
triangle v2f input[3]
```

ここが入力に関する部分です。

今回は三角形を元に三角錐を生成したいので、入力は triangleとしています。これにより、単位プリミティブである三角形の各頂点情報が入力されるうになり、三角形は 3 点の頂点から構成されますので、受け取っている仮引数は長さ 3 の配列となります。なので、もし入力を triangleではなく pointにした場合は構成する頂点は 1 点のみなので、geom(point v2f input[1])の様に長さ 1 の配列で受け取ることになります。

6.4.2 出力

▼

```
inout TriangleStream<g2f> outStream
```

ここが出力に関する部分です。

今回生成するメッシュのプリミティブは三角形としたいため、TriangleStream型で宣言しています。TriangleStrema型は出力が三角形ストリップである事を意味しているため、出力した各頂点情報を元に三角形を生成してくれるようになります。他にも PointStream型や LineStream型などがありますので、目的に応じて出力のプリミティブ型を選択する必要があります。

また、[maxvertexcount(12)]の部分にて最大出力数を 12 に設定してあります。これは三角錐を構成する三角形の数は底辺の 1 つと側面の 3 つで計 4 つであり、一つの三角形に付き頂点数が 3 点必要なので、3 * 4 で 12 点の頂点を出力することになるため 12 と設定してあります。

6.4.3 処理

▼

```
g2f out0;
out0.pos = UnityObjectToClipPos(p0);
out0.col = _BottomColor;
```

```
g2f out1;
out1.pos = UnityObjectToClipPos(p1);
out1.col = _BottomColor;

g2f out2;
out2.pos = UnityObjectToClipPos(p2);
out2.col = _BottomColor;

g2f o;
o.pos = UnityObjectToClipPos(c);
o.col = _TopColor;

// bottom
outStream.Append(out0);
outStream.Append(out1);
outStream.Append(out2);
outStream.RestartStrip();

// sides
outStream.Append(out0);
outStream.Append(out1);
outStream.Append(o);
outStream.RestartStrip();

outStream.Append(out1);
outStream.Append(out2);
outStream.Append(o);
outStream.RestartStrip();

outStream.Append(out2);
outStream.Append(out0);
outStream.Append(o);
outStream.RestartStrip();
```

ここが実際の頂点を出力している処理の部分です。

まず最初に出力用の g2f 型の変数を宣言し、頂点座標と色情報を格納しています。この時 Vertex Shader と同じようにオブジェクト空間からカメラのクリップ空間への変換をしておく必要があります。

その後に、メッシュを構成する頂点の順序を意識しながら、頂点情報を出力しています。outStream変数の Append関数に出力用変数を渡すことで現在のストリームに追加されていき、RestartStrip関数を呼び出す事によって現在のプリミティブストリップを終了し、新しいストリームを開始しています。

これは、TriangleStreamは三角形ストリップなので、Append関数で頂点を追加していくほどそのストリームに追加されている全ての頂点を元に、接続された複数の三角形を生成していくことになります。なので、今回の様に三角形同士が Appendされた順序を元に接続されると困る時は、一旦 RestartStripを呼び出して新しいストリームを開始する必要があります。もちろん Append順を工夫することで RestartStrip関数の呼び出しを減らすことは可能です。

6.5 Grass Shader

本項では、前項の『簡単な Geometry Shader』から少し発展させて、Geometry Shader を使ってリアルタイムに草を生成する Grass Shader について説明します。

以下は説明する Grass Shader のプログラムです。

▼

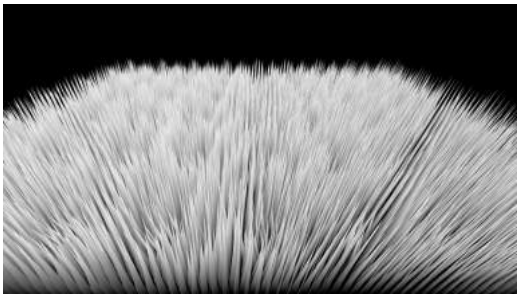
```
Shader "Custom/Grass" {
    Properties
    {
        // 草の高さ
        _Height("Height", float) = 80
        // 草の幅
        _Width("Width", float) = 2.5

        // 草の下部の高さ
        _BottomHeight("Bottom Height", float) = 0.3
        // 草の中間部の高さ
        _MiddleHeight("Middle Height", float) = 0.4
        // 草の上部の高さ
        _TopHeight("Top Height", float) = 0.5

        // 草の下部の幅
        _BottomWidth("Bottom Width", float) = 0.5
        // 草の中間部の幅
        _MiddleWidth("Middle Width", float) = 0.4
        // 草の上部の幅
        _TopWidth("Top Width", float) = 0.2

        // 草の下部の曲がり具合
        _BottomBend("Bottom Bend", float) = 1.0
        // 草の中間部の曲がり具合
        _MiddleBend("Middle Bend", float) = 1.0
        // 草の上部の曲がり具合
        _TopBend("Top Bend", float) = 2.0

        // 風の強さ
        _WindPower("Wind Power", float) = 1.0

        // 草の上部の色
        _TopColor("Top Color", Color) = (1.0, 1.0, 1.0, 1.0)
        // 草の下部の色
        _BottomColor("Bottom Color", Color) = (0.0, 0.0, 0.0, 1.0)

        // 草の高さにランダム性を与えるノイズテクスチャ
        _HeightMap("Height Map", 2D) = "white"
        // 草の向きにランダム性を与えるノイズテクスチャ
        _RotationMap("Rotation Map", 2D) = "black"
        // 風の強さにランダム性を与えるノイズテクスチャ
        _WindMap("Wind Map", 2D) = "black"
    }
    SubShader
    {
```

```
        Tags{ "RenderType" = "Opaque" }

        LOD 100
        Cull Off

        Pass
        {
            CGPROGRAM
            #pragma target 5.0
            #include "UnityCG.cginc"

            #pragma vertex vert
            #pragma geometry geom
            #pragma fragment frag

            float _Height, _Width;
            float _BottomHeight, _MiddleHeight, _TopHeight;
            float _BottomWidth, _MiddleWidth, _TopWidth;
            float _BottomBend, _MiddleBend, _TopBend;

            float _WindPower;
            float4 _TopColor, _BottomColor;
            sampler2D _HeightMap, _RotationMap, _WindMap;

            struct v2g
            {
                float4 pos : SV_POSITION;
                float3 nor : NORMAL;
                float4 hei : TEXCOORD0;
                float4 rot : TEXCOORD1;
                float4 wind : TEXCOORD2;
            };

            struct g2f
            {
                float4 pos : SV_POSITION;
                float4 color : COLOR;
            };

            v2g vert(appdata_full v)
            {
                v2g o;
                float4 uv = float4(v.texcoord.xy, 0.0f, 0.0f);

                o.pos = v.vertex;
                o.nor = v.normal;
                o.hei = tex2Dlod(_HeightMap, uv);
                o.rot = tex2Dlod(_RotationMap, uv);
                o.wind = tex2Dlod(_WindMap, uv);

                return o;
            }

            [maxvertexcount(7)]
            void geom(triangle v2g i[3], inout TriangleStream<g2f> stream)
            {
                float4 p0 = i[0].pos;
                float4 p1 = i[1].pos;
```

```
                float4 p2 = i[2].pos;

                float3 n0 = i[0].nor;
                float3 n1 = i[1].nor;
                float3 n2 = i[2].nor;

                float height = (i[0].hei.r + i[1].hei.r + i[2].hei.r) / 3.0f;
                float rot = (i[0].rot.r + i[1].rot.r + i[2].rot.r) / 3.0f;
                float wind = (i[0].wind.r + i[1].wind.r + i[2].wind.r) / 3.0f;

                float4 center = ((p0 + p1 + p2) / 3.0f);
                float4 normal = float4(((n0 + n1 + n2) / 3.0f).xyz, 1.0f);

                float bottomHeight = height * _Height * _BottomHeight;
                float middleHeight = height * _Height * _MiddleHeight;
                float topHeight = height * _Height * _TopHeight;

                float bottomWidth = _Width * _BottomWidth;
                float middleWidth = _Width * _MiddleWidth;
                float topWidth = _Width * _TopWidth;

                rot = rot - 0.5f;
                float4 dir = float4(normalize((p2 - p0) * rot).xyz, 1.0f);

                g2f o[7];

                // Bottom.
                o[0].pos = center - dir * bottomWidth;
                o[0].color = _BottomColor;

                o[1].pos = center + dir * bottomWidth;
                o[1].color = _BottomColor;

                // Bottom to Middle.
                o[2].pos = center - dir * middleWidth + normal * bottomHeight;
                o[2].color = lerp(_BottomColor, _TopColor, 0.33333f);

                o[3].pos = center + dir * middleWidth + normal * bottomHeight;
                o[3].color = lerp(_BottomColor, _TopColor, 0.33333f);

                // Middle to Top.
                o[4].pos = o[3].pos - dir * topWidth + normal * middleHeight;
                o[4].color = lerp(_BottomColor, _TopColor, 0.66666f);

                o[5].pos = o[3].pos + dir * topWidth + normal * middleHeight;
                o[5].color = lerp(_BottomColor, _TopColor, 0.66666f);

                // Top.
                o[6].pos = o[5].pos + dir * topWidth + normal * topHeight;
                o[6].color = _TopColor;

                // Bend.
                dir = float4(1.0f, 0.0f, 0.0f, 1.0f);

                o[2].pos += dir
                            * (_WindPower * wind * _BottomBend)
                            * sin(_Time);
                o[3].pos += dir
```

```
                            * (_WindPower * wind * _BottomBend)
                            * sin(_Time);
                o[4].pos += dir
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o[5].pos += dir
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o[6].pos += dir
                            * (_WindPower * wind * _TopBend)
                            * sin(_Time);

                [unroll]
                for (int i = 0; i < 7; i++) {
                    o[i].pos = UnityObjectToClipPos(o[i].pos);
                    stream.Append(o[i]);
                }
            }

            float4 frag(g2f i) : COLOR
            {
                return i.color;
            }
            ENDCG
        }
    }
}
```

このシェーダーを縦横に複数並べた Plane メッシュに適用すると、図 6.4 のようになります。

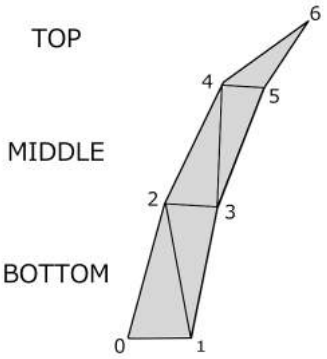


▲図 6.4　Grass Shader の結果

この中から草を生成する処理についての説明をします。

6.5.1 基本方針

今回は一つのプリミティブにつき 1 本の草を生成することにします。草の形状の生成については図 6.5 のように下部・中間部・上部に分けて頂点を合計 7 点生成し、上に行くほど斜めにしていくことで、草の斜め具合を簡易的に表現します。



▲図 6.5　草の形の作り方

6.5.2 パラメーター

詳細はコメントにて記載してありますが、一本の草の中の各部分 (下部・中間部・上部) の横幅と高さをコントロールする係数、草全体の横幅と高さをコントロールする係数を主なパラメーターとして用意しています。また一本一本の草が同じ形になるのは見栄えが悪いので、ランダム性を持たせるためのノイズテクスチャを使います。

6.5.3 処理

▼

```
float height = (i[0].hei.r + i[1].hei.r + i[2].hei.r) / 3.0f;
float rot = (i[0].rot.r + i[1].rot.r + i[2].rot.r) / 3.0f;
float wind = (i[0].wind.r + i[1].wind.r + i[2].wind.r) / 3.0f;

float4 center = ((p0 + p1 + p2) / 3.0f);
float4 normal = float4(((n0 + n1 + n2) / 3.0f).xyz, 1.0f);
```

この部分では草の高さと向き、風の強弱の基準となる数値を計算しています。Geometry Shader 内で計算しても良いのですが、頂点に対してメタ情報的に持たせた方が Geometry Shader 上で計算を行なう上での初期値の様な扱いが出来るので Vertex Shader で計算しています。

▼

```
float4 center = ((p0 + p1 + p2) / 3.0f);
float4 normal = float4(((n0 + n1 + n2) / 3.0f).xyz, 1.0f);
```

ここでは草の中心部分と、草を生やしていく方向を計算しています。ここの部分をノイズテクスチャなどで決定するようにすると、草が生える方向にランダム性を持たせることが出来ます。

▼

```
float bottomHeight = height * _Height * _BottomHeight;

...

o[6].pos += dir * (_WindPower * wind * _TopBend) * sin(_Time);
```

長いのでプログラムは略記してあります。この部分では下部・中間部・上部についての高さと幅をそれぞれ計算し、それを元に座標を求めています。

▼

```
[unroll]
for (int i = 0; i < 7; i++) {
    o[i].pos = UnityObjectToClipPos(o[i].pos);
    stream.Append(o[i]);
}
```

この部分にて計算した 7 点の頂点を Appendしています。今回は三角形が繋がりながら生成されていっても問題ないため、RestartStripはしていません。

なお、forステートメントに対して [unroll]というアトリビュートを適用しています。これはコンパイル時に、ループの回数分ループ内の処理を展開するというアトリビュートで、メモリサイズが大きくなるというデメリットはあるのですが、高速に動作するという利点があります。

6.6 まとめ

ここまで Geometry Shader についての説明から、基本と応用のプログラムまでを説明してきました。CPU 上で動くプログラムを書くのとは多少なりとも特徴が異なる所がありますが、基本的な部分を抑えさせすれば活用できるはずです。

実は通説として Geometry Shader は遅いと言われているそうです。筆者自身はあまり感じたことはないのですが、利用範囲が大規模になると大変なのかもしれません。もし Geometry Shader を大規模に使うということになりそうでしたら、ぜひ一度ベンチマークなどを取ってみてください。

それでも GPU 上で動的に且つ自由に新しいメッシュを作ったり、削除したり出来るというのはアイデアの幅をかなり広げることになると思います。個人的に最も重要なことは、どの技術を使ったのかではなく、それによって何を作り、表現するのかだと思っています。ぜひ本章にて Geometry Shader という一つの道具を知り学んだ上で、なにか新しい可能性を感じてもらえたら幸いです。

6.7 参考

- チュートリアル 13 : ジオメトリシェーダー - https://msdn.microsoft.com/ja-jp/library/bb172497
- ジオメトリシェーダーオブジェクト in MSDN - https://msdn.microsoft.com/ja-jp/library/ee418313
- ジオメトリシェーダのジオメトリ切断による透明ジオメトリのためのレンダリング手法 - http://t-pot.com/program/147_CGGONG2008/index.html

第 7 章

雰囲気で始めるマーチングキューブス法入門

7.1 マーチングキューブス法とは？

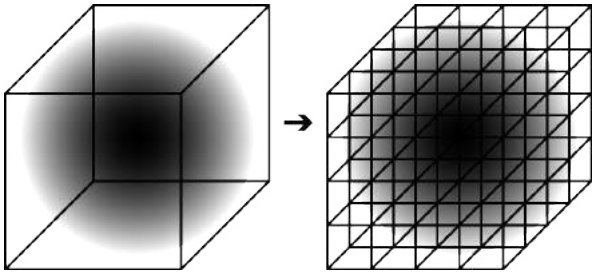
7.1.1 歴史と概要

マーチングキューブス法とは、ボリュームレンダリング法の一つで、スカラーデータで満たされた 3 次元ボクセルデータを、ポリゴンデータに変換するアルゴリズムです。William E. Lorensen と Harvey E. Cline によって 1987 年に最初の論文が発表されました。

マーチングキューブス法は特許が取得されていましたが、2005 年に特許が切れているので、現在は自由に使用できます。

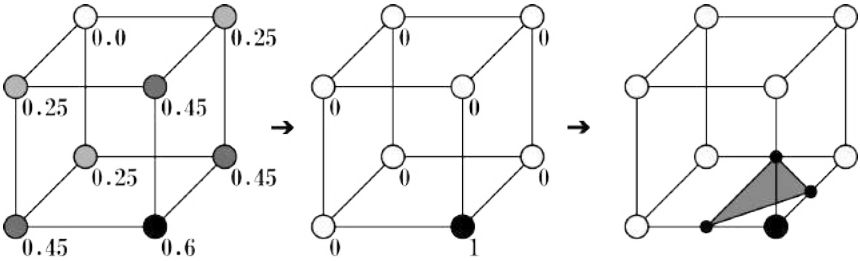
7.1.2 簡単な仕組みの解説

まず、ボリュームデータの空間を 3 次元グリッドで分割します。



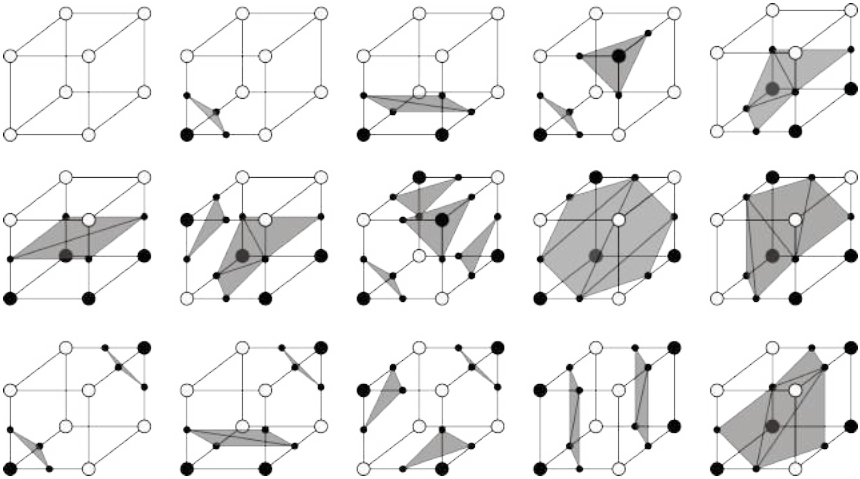
▲図 7.1　3 次元ボリュームデータとグリッド分割

次に分割したグリッドの 1 つを取り出してみましょう。グリッドの 8 つの角の値が閾値以上だったら 1 、閾値未満だったら 0 として、 8 頂点の境界を割り出します。以下の図は、閾値を 0.5 とした場合の流れです。



▲図 7.2　角の値に応じて境界を割り出す

その 8 つの角の組み合わせは 256 種類ありますが、回転や反転を駆使すると 15 種類に収まります。その 15 種類の組み合わせに対応した三角ポリゴンのパターンを割り当てます。



▲図 7.3　角の組み合わせ

7.2 サンプルリポジトリ

本章で解説するサンプルプロジェクトは、UnityGraphicsProgramming の Unity プロジェクト https://github.com/IndieVisualLab/UnityGraphicsProgramming 内にある Assets/GPUMarchingCubes にあります。

実装にあたり、Paul Bourke 氏の Polygonising a scalar field のサイト[*1]を参考に、Unity に移植させて頂きました。

今回はこのサンプルプロジェクトに沿って解説していきます。

実装は大きくわけて 3 つあります。

- メッシュの初期化、毎フレームの描画登録処理（C#スクリプト部分）
- ComputeBuffer の初期化
- 実際の描画処理（シェーダー部分）

まずは、メッシュの初期化や描画登録をする **GPUMarchingCubesDrawMesh** クラスから作っていきます。

7.2.1 GeometryShader 用のメッシュを作る

前項で説明したとおり、マーチングキューブス法はグリッドの 8 つの角の組み合わせでポリゴンを生成するアルゴリズムです。リアルタイムにそれを行うには、動的にポリゴンを作る必要があります。

しかし、毎フレーム CPU 側 (C#側) でメッシュの頂点配列を生成するのは非効率です。

そこで、GeometryShader を使います。GeometryShader は、大雑把に説明すると VertexShader と FragmentShader の間に位置する Shader で、VertexShader で処理された頂点を増減させることができます。

例えば、1 頂点の周囲に 6 つの頂点を追加して板ポリゴンを生成したりできます。

更に、Shader 側 (GPU 側) で処理するのでとても高速です。

今回は GeometryShader を使って MarchingCubes のポリゴンを生成して表示してみます。

まず、 **GPUMarchingCubesDrawMesh** クラスで使う変数群を定義します。

▼リスト 7.1　変数群の定義部分

[*1] Polygonising a scalar field http://paulbourke.net/geometry/polygonise/

```
using UnityEngine;

public class GPUMarchingCubesDrawMesh : MonoBehaviour {

    #region public
    public int segmentNum = 32;                 // グリッドの一辺の分割数

    [Range(0,1)]
    public float threashold = 0.5f;             // メッシュ化するスカラー値のしきい値
    public Material mat;                        // レンダリング用のマテリアル

    public Color DiffuseColor = Color.green;    // ディフューズカラー
    public Color EmissionColor = Color.black;   // 発光色
    public float EmissionIntensity = 0;         // 発光の強さ

    [Range(0,1)]
    public float metallic = 0;                  // メタリック感
    [Range(0, 1)]
    public float glossiness = 0.5f;             // 光沢感
    #endregion

     #region private
    int vertexMax = 0;                          // 頂点数
    Mesh[] meshs = null;                        // Mesh 配列
    Material[] materials = null;                // Mesh ごとのマテリアル配列
    float renderScale = 1f / 32f;               // 表示スケール
    MarchingCubesDefines mcDefines = null;      // MarchingCubes 用定数配列群
    #endregion

}
```

次に GeometryShader に渡すためのメッシュを作成します。メッシュの頂点は、分割した 3 次元グリッド内に 1 個ずつ配置するようにします。例えば、一辺の分割数が 64 の場合、64*64*64=262,144 個もの頂点が必要になります。

しかし、Unity2017.1.1f1 において、1 つのメッシュの頂点数は 65,535 個が上限となってます。その為、メッシュ 1 つにつき、頂点数を 65,535 個以内に収める形で分割します。

▼リスト 7.2　メッシュ作成部分

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log("VertexMax " + vertexMax);

  // 1Cube の大きさを segmentNum で分割してレンダリング時の大きさを決める
  renderScale = 1f / segmentNum;

  CreateMesh();

  // シェーダーで使う MarchingCubes 用の定数配列の初期化
```

```
    mcDefines = new MarchingCubesDefines();
  }

  void CreateMesh()
  {
    // Mesh の頂点数は 65535 が上限なので、Mesh を分割する
    int vertNum = 65535;
    int meshNum = Mathf.CeilToInt((float)vertexMax / vertNum);  // 分 割 す る
  Mesh の数
    Debug.Log("meshNum " + meshNum );

    meshs = new Mesh[meshNum];
    materials = new Material[meshNum];

    // Mesh のバウンズ計算
    Bounds bounds = new Bounds(
      transform.position,
      new Vector3(segmentNum, segmentNum, segmentNum) * renderScale
    );

    int id = 0;
    for (int i = 0; i < meshNum; i++)
    {
      // 頂点作成
      Vector3[] vertices = new Vector3[vertNum];
      int[] indices = new int[vertNum];
      for(int j = 0; j < vertNum; j++)
      {
        vertices[j].x = id % segmentNum;
        vertices[j].y = (id / segmentNum) % segmentNum;
        vertices[j].z = (id / (segmentNum * segmentNum)) % segmentNum;

        indices[j] = j;
        id++;
      }

      // Mesh 作成
      meshs[i] = new Mesh();
      meshs[i].vertices = vertices;
      // GeometryShader でポリゴンを作るので MeshTopology は Points で良い
      meshs[i].SetIndices(indices, MeshTopology.Points, 0);
      meshs[i].bounds = bounds;

      materials[i] = new Material(mat);
    }
  }
```

7.2.2 ComputeBuffer の初期化

MarchingCubesDefinces.cs というソースには、マーチングキューブス法のレンダリングで使う定数配列と、その定数配列をシェーダーに渡すための ComputeBuffer が定義されています。ComputeBuffer とは、シェーダーで使うデータを格納するバッファです。データは GPU 側のメモリに置かれるのでシェーダーからのアクセスが早

いです。

実は、マーチングキューブス法のレンダリングで使う定数配列は、シェーダー側で定義することは可能です。しかし、何故シェーダーで使う定数配列を、C#側で初期化しているのかというと、シェーダーにはリテラル値 (直書きした値) の個数が 4096 までしか登録出来ない制限があるためです。膨大な定数配列をシェーダー内に定義すると、あっという間にリテラル値の数の上限に到達してしまいます。

そこで、ComputeShader に格納して渡すことで、リテラル値ではなくなるので上限にひっかからなくなります。そのため、工程が少々増えてしまいますが、C#側で ComputeBuffer に定数配列を格納してシェーダーに渡すようにしています。

▼リスト 7.3　ComputeBuffer の初期化部分

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log("VertexMax " + vertexMax);

  // 1Cube の大きさを segmentNum で分割してレンダリング時の大きさを決める
  renderScale = 1f / segmentNum;

  CreateMesh();

  // シェーダーで使う MarchingCubes 用の定数配列の初期化
  mcDefines = new MarchingCubesDefines();
}
```

先程の Initialize() 関数の中で、MarchingCubesDefines の初期化を行っています。

7.2.3　レンダリング

次にレンダリング処理を呼び出す関数です。

今回は、複数のメッシュを一度にレンダリングするのと、Unity のライティングの影響を受けられるようにするため、Graphics.DrawMesh() を使います。public 変数で定義した DiffuseColor 等の意味は、シェーダー側の解説で説明します。

前項の、MarchingCubesDefines クラスの ComputeBuffer 達を material.setBuffer でシェーダーに渡しています。

▼リスト 7.4　レンダリング部分

```
void RenderMesh()
{
  Vector3 halfSize = new Vector3(segmentNum, segmentNum, segmentNum)
                   * renderScale * 0.5f;
  Matrix4x4 trs = Matrix4x4.TRS(
                   transform.position,
```

```
                    transform.rotation,
                    transform.localScale
                );

    for (int i = 0; i < meshs.Length; i++)
    {
      materials[i].SetPass(0);
      materials[i].SetInt("_SegmentNum", segmentNum);
      materials[i].SetFloat("_Scale", renderScale);
      materials[i].SetFloat("_Threashold", threashold);
      materials[i].SetFloat("_Metallic", metallic);
      materials[i].SetFloat("_Glossiness", glossiness);
      materials[i].SetFloat("_EmissionIntensity", EmissionIntensity);

      materials[i].SetVector("_HalfSize", halfSize);
      materials[i].SetColor("_DiffuseColor", DiffuseColor);
      materials[i].SetColor("_EmissionColor", EmissionColor);
      materials[i].SetMatrix("_Matrix", trs);

      Graphics.DrawMesh(meshs[i], Matrix4x4.identity, materials[i], 0);
    }
  }
```

7.3 呼び出し

▼リスト 7.5　呼び出し部分

```
// Use this for initialization
void Start ()
{
  Initialize();
}

void Update()
{
  RenderMesh();
}
```

Start() で Initialize() を呼び出してメッシュを生成、Update() 関数で RenderMesh() を呼び出してレンダリングします。
Update() で RenderMesh() を呼び出す理由は、Graphics.DrawMesh() が即座に描画するわけではなく、「レンダリング処理に一旦登録する」という感じのものだからです。
登録することで、Unity がライトやシャドウを適応してくれます。似たような関数に Graphics.DrawMeshNow() がありますが、こちらは即座に描画するので Unity のライトやシャドウが適応されません。また、Update() ではなく、OnRenderObject() や OnPostRender() などで呼び出す必要があります。

7.4 シェーダ側の実装

今回のシェーダは、大きく分けて「実体のレンダリング部」と「影のレンダリング部」の 2 つに分かれます。さらに、それぞれの中で、頂点シェーダ、ジオメトリシェーダ、フラグメントシェーダの 3 つのシェーダ関数が実行されます。

シェーダーのソースが長いので、実装全体はサンプルプロジェクトの方を見てもらうことにして、要所要所だけ解説します。解説するシェーダーのファイルは、GPUMarchingCubesRenderMesh.shader です。

7.4.1 変数の宣言

シェーダーの上の方では、レンダリングで使う構造体の定義をしています。

▼リスト 7.6　構造体の定義部分

```
// メッシュから渡ってくる頂点データ
struct appdata
{
  float4 vertex : POSITION; // 頂点座標
};

// 頂点シェーダからジオメトリシェーダに渡すデータ
struct v2g
{
  float4 pos : SV_POSITION; // 頂点座標
};

// 実体レンダリング時のジオメトリシェーダからフラグメントシェーダに渡すデータ
struct g2f_light
{
  float4 pos      : SV_POSITION;  // ローカル座標
  float3 normal   : NORMAL;       // 法線
  float4 worldPos : TEXCOORD0;    // ワールド座標
  half3 sh        : TEXCOORD3;    // SH
};

// 影のレンダリング時のジオメトリシェーダからフラグメントシェーダに渡すデータ
struct g2f_shadow
{
  float4 pos      : SV_POSITION;  // 座標
  float4 hpos     : TEXCOORD1;
};
```

次に変数の定義をしています。

▼リスト 7.7　変数の定義部分

```
int _SegmentNum;

float _Scale;
float _Threashold;

float4 _DiffuseColor;
float3 _HalfSize;
float4x4 _Matrix;

float _EmissionIntensity;
half3 _EmissionColor;

half _Glossiness;
half _Metallic;

StructuredBuffer<float3> vertexOffset;
StructuredBuffer<int> cubeEdgeFlags;
StructuredBuffer<int2> edgeConnection;
StructuredBuffer<float3> edgeDirection;
StructuredBuffer<int> triangleConnectionTable;
```

ここで定義している各種変数の中身は、C#側の RenderMesh() 関数の中で、material.Set ○○関数で受け渡しています。MarchingCubesDefines クラスの ComputeBuffer 達は、StructuredBuffer<○○>と型の呼び名が変わっています。

7.4.2 頂点シェーダ

ほとんどの処理はジオメトリシェーダの方で行うので、頂点シェーダは凄くシンプルです。単純にメッシュから渡される頂点データをそのままジオメトリシェーダに渡しているだけです。

▼リスト 7.8　頂点シェーダの実装部分

```
// メッシュから渡ってくる頂点データ
struct appdata
{
  float4 vertex : POSITION; // 頂点座標
};

// 頂点シェーダからジオメトリシェーダに渡すデータ
struct v2g
{
  float4 pos : SV_POSITION; // 座標
};

// 頂点シェーダ
v2g vert(appdata v)
{
  v2g o = (v2g)0;
  o.pos = v.vertex;
  return o;
}
```

ちなみに、頂点シェーダは実体と影で共通です。

7.4.3 実体のジオメトリシェーダ

長いので分割しながら説明します。

▼リスト 7.9　ジオメトリシェーダーの関数宣言部分

```
// 実体のジオメトリシェーダ
[maxvertexcount(15)]  // シェーダから出力する頂点の最大数の定義
void geom_light(point v2g input[1],
                inout TriangleStream<g2f_light> outStream)
```

まず、ジオメトリシェーダの宣言部です。

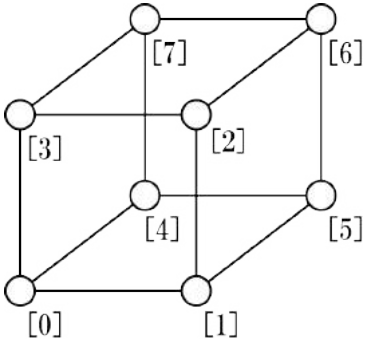
[maxvertexcount(15)]はシェーダから出力する頂点の最大数の定義です。今回のマーチングキューブス法のアルゴリズムでは 1 グリッドにつき、三角ポリゴンが最大 5 つできるので、3*5 で合計 15 個の頂点が出力されます。
そのため、maxvertexcount の () の中に 15 と記述します。

▼リスト 7.10　グリッドの 8 つの角のスカラー値取得部分

```
float cubeValue[8]; // グリッドの 8 つの角のスカラー値取得用の配列

// グリッドの 8 つの角のスカラー値を取得
for (i = 0; i < 8; i++) {
  cubeValue[i] = Sample(
                  pos.x + vertexOffset[i].x,
                  pos.y + vertexOffset[i].y,
                  pos.z + vertexOffset[i].z
  );
}
```

pos は、メッシュを作成する時にグリッド空間に配置した頂点の座標が入っています。vertexOffset は、名前の通り pos に加えるオフセット座標の配列です。

このループは、1 頂点＝ 1 つのグリッドの 8 つの角の座標のボリュームデータ中のスカラー値を取得しています。vertexOffset は、グリッドの角の順番を指しています。



▲図 7.4　グリッドの角の座標の順番

▼リスト 7.11　サンプリング関数部分

```
// サンプリング関数
float Sample(float x, float y, float z) {

  // 座標がグリッド空間からはみ出してたいないか？
  if ((x <= 1) ||
      (y <= 1) ||
      (z <= 1) ||
      (x >= (_SegmentNum - 1)) ||
      (y >= (_SegmentNum - 1)) ||
      (z >= (_SegmentNum - 1))
     )
    return 0;

  float3 size = float3(_SegmentNum, _SegmentNum, _SegmentNum);

  float3 pos = float3(x, y, z) / size;

  float3 spPos;
  float result = 0;

  // ３つの球の距離関数
  for (int i = 0; i < 3; i++) {
    float sp = -sphere(
      pos - float3(0.5, 0.25 + 0.25 * i, 0.5),
      0.1 + (sin(_Time.y * 8.0 + i * 23.365) * 0.5 + 0.5) * 0.025) + 0.5;
    result = smoothMax(result, sp, 14);
  }

  return result;
}
```

ボリュームデータから指定した座標のスカラー値を取ってくる関数です。今回は膨大な 3D ボリュームデータではなく、距離関数を使ったシンプルなアルゴリズムでス

カラー値を算出します。

距離関数について

今回マーチングキューブス法で描画する 3 次元形状は、「距離関数」と言うものを使って定義します。
ここでいう距離関数とは、ざっくり説明すると「距離の条件を満たす関数」です。
例えば、球体の距離関数は、以下になります。

▼リスト 7.12　球体の距離関数

```
inline float sphere(float3 pos, float radius)
{
    return length(pos) - radius;
}
```

pos には、座標が入るのですが、球体の中心座標を原点 (0,0,0) とした場合で考えます。radius は半径です。
length(pos) で長さを求めていますが、これは原点と pos までの距離で、それを半径 radius で引くので、半径以下の長さの場合、当たり前ですが負の値になります。
つまり、座標 pos を渡して負の値が返ってきた場合は、「座標は球体の中にいる」という判定ができます。
距離関数のメリットは、数行のシンプルな計算式で図形を表現できるので、プログラムが小さくしやすいところです。その他の距離関数についての情報は、Inigo Quilez 氏のサイトでたくさん紹介されています。
http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

▼リスト 7.13　3 つの球の距離関数を合成したもの

```
// 3 つの球の距離関数
for (int i = 0; i < 3; i++) {
  float sp = -sphere(
    pos - float3(0.5, 0.25 + 0.25 * i, 0.5),
    0.1 + (sin(_Time.y * 8.0 + i * 23.365) * 0.5 + 0.5) * 0.025) + 0.5;
  result = smoothMax(result, sp, 14);
}
```

今回は、グリッドの 1 マスの 8 つの角（頂点）を pos として使っています。球体の

中心からの距離を、そのままボリュームデータの濃度として扱います。

後述しますが、閾値が 0.5 以上の時にポリゴン化するため、符号を反転しています。また、座標を微妙にずらして 3 つの球体との距離を求めています。

▼リスト 7.14　smoothMax 関数

```
float smoothMax(float d1, float d2, float k)
{
  float h = exp(k * d1) + exp(k * d2);
  return log(h) / k;
}
```

smoothMax は、距離関数の結果をいい感じにブレンドする関数です。これを使って 3 つの球体をメタボールのように融合させることが出来ます。

▼リスト 7.15　閾値チェック

```
// グリッドの８つの角の値が閾値を超えているかチェック
for (i = 0; i < 8; i++) {
  if (cubeValue[i] <= _Threashold) {
    flagIndex |= (1 << i);
  }
}

int edgeFlags = cubeEdgeFlags[flagIndex];

// 空か完全に満たされている場合は何も描画しない
if ((edgeFlags == 0) || (edgeFlags == 255)) {
  return;
}
```

グリッドの角のスカラー値が閾値を越えていたら、flagIndex にビットを立てていきます。その flagIndex をインデックスとして、cubeEdgeFlags 配列からポリゴンを生成するための情報を取り出して edgeFlags に格納しています。グリッドの全ての角が閾値未満か閾値以上の場合は、完全に中か外なのでポリゴンは生成しません。

▼リスト 7.16　ポリゴンの頂点座標計算

```
float offset = 0.5;
float3 vertex;
for (i = 0; i < 12; i++) {
  if ((edgeFlags & (1 << i)) != 0) {
    // 角同士の閾値のオフセットを取得
    offset = getOffset(
               cubeValue[edgeConnection[i].x],
               cubeValue[edgeConnection[i].y], _
               Threashold
             );

    // オフセットを元に頂点の座標を補完
```

```
    vertex = vertexOffset[edgeConnection[i].x]
             + offset * edgeDirection[i];

    edgeVertices[i].x = pos.x + vertex.x * _Scale;
    edgeVertices[i].y = pos.y + vertex.y * _Scale;
    edgeVertices[i].z = pos.z + vertex.z * _Scale;

    // 法線計算（Sample し直すため、スケールを掛ける前の頂点座標が必要）
    edgeNormals[i] = getNormal(
                        defpos.x + vertex.x,
                        defpos.y + vertex.y,
                        defpos.z + vertex.z
                     );
  }
}
```

ポリゴンの頂点座標を計算している箇所です。先程の、edgeFlags のビットを見て、グリッドの辺上に置くポリゴンの頂点座標を計算しています。

getOffset は、グリッドの 2 つの角のスカラー値と閾値から、今の角から次の角までの割合 (offset) を出しています。今の角の座標から、次の角の方向へ offset 分ずらすことで、最終的になめらかなポリゴンになります。

getNormal では、サンプリングし直して勾配を出して法線を算出しています。

▼リスト 7.17　頂点を連結してポリゴンを作る

```
// 頂点を連結してポリゴンを作成
int vindex = 0;
int findex = 0;
// 最大 5 つの三角形ができる
for (i = 0; i < 5; i++) {
  findex = flagIndex * 16 + 3 * i;
  if (triangleConnectionTable[findex] < 0)
    break;

  // 三角形を作る
  for (j = 0; j < 3; j++) {
    vindex = triangleConnectionTable[findex + j];

    // Transform 行列を掛けてワールド座標に変換
    float4 ppos = mul(_Matrix, float4(edgeVertices[vindex], 1));
    o.pos = UnityObjectToClipPos(ppos);

    float3 norm = UnityObjectToWorldNormal(
                    normalize(edgeNormals[vindex])
                  );
    o.normal = normalize(mul(_Matrix, float4(norm,0)));

    outStream.Append(o);  // ストリップに頂点を追加
  }
  outStream.RestartStrip(); // 一旦区切って次のプリミティブストリップを開始
}
```

先程求めた頂点座標群を繋いでポリゴンを作っている箇所です。triangleConnec-

tionTable 配列に接続する頂点のインデックスが入っています。頂点座標に Transform の行列を掛けてワールド座標に変換し、UnityObjectToClipPos() でスクリーン座標に変換しています。

また、UnityObjectToWorldNormal() で法線もワールド座標系に変換しています。これらの頂点と法線は、次のフラグメントシェーダでライティングに使います。

TriangleStream.Append() や RestartStrip() は、ジオメトリシェーダ用の特殊な関数です。Append() は、現在のストリップに頂点データを追加します。RestartStrip() は、新しいストリップを作成します。TriangleStream なので 1 つのストリップには 3 つまで Append するイメージです。

7.4.4 実体のフラグメントシェーダ

Unity の GI(グローバルイル・ミネーション) などのライティングを反映させるため、Generate code 後の SurfaceShader のライティング処理部分を移植します。

▼リスト 7.18　フラグメントシェーダの定義

```
// 実体のフラグメントシェーダ
void frag_light(g2f_light IN,
  out half4 outDiffuse        : SV_Target0,
  out half4 outSpecSmoothness : SV_Target1,
  out half4 outNormal         : SV_Target2,
  out half4 outEmission       : SV_Target3)
```

G-Buffer に出力するため出力 (SV_Target) が 4 つあります。

▼リスト 7.19　SurfaceOutputStandard 構造体の初期化

```
#ifdef UNITY_COMPILER_HLSL
  SurfaceOutputStandard o = (SurfaceOutputStandard)0;
#else
  SurfaceOutputStandard o;
#endif
  o.Albedo = _DiffuseColor.rgb;
  o.Emission = _EmissionColor * _EmissionIntensity;
  o.Metallic = _Metallic;
  o.Smoothness = _Glossiness;
  o.Alpha = 1.0;
  o.Occlusion = 1.0;
  o.Normal = normal;
```

あとで使う SurfaceOutputStandard 構造体に、色や光沢感などのパラメータをセットします。

▼リスト 7.20　GI 関係の処理

```
// Setup lighting environment
UnityGI gi;
UNITY_INITIALIZE_OUTPUT(UnityGI, gi);
gi.indirect.diffuse = 0;
gi.indirect.specular = 0;
gi.light.color = 0;
gi.light.dir = half3(0, 1, 0);
gi.light.ndotl = LambertTerm(o.Normal, gi.light.dir);

// Call GI (lightmaps/SH/reflections) lighting function
UnityGIInput giInput;
UNITY_INITIALIZE_OUTPUT(UnityGIInput, giInput);
giInput.light = gi.light;
giInput.worldPos = worldPos;
giInput.worldViewDir = worldViewDir;
giInput.atten = 1.0;

giInput.ambient = IN.sh;

giInput.probeHDR[0] = unity_SpecCube0_HDR;
giInput.probeHDR[1] = unity_SpecCube1_HDR;

#if UNITY_SPECCUBE_BLENDING || UNITY_SPECCUBE_BOX_PROJECTION
// .w holds lerp value for blending
giInput.boxMin[0] = unity_SpecCube0_BoxMin;
#endif

#if UNITY_SPECCUBE_BOX_PROJECTION
giInput.boxMax[0] = unity_SpecCube0_BoxMax;
giInput.probePosition[0] = unity_SpecCube0_ProbePosition;
giInput.boxMax[1] = unity_SpecCube1_BoxMax;
giInput.boxMin[1] = unity_SpecCube1_BoxMin;
giInput.probePosition[1] = unity_SpecCube1_ProbePosition;
#endif

LightingStandard_GI(o, giInput, gi);
```

GI 関係の処理です。UnityGIInput に初期値を入れて、LightnintStandard_GI() で計算した GI の結果を UnityGI に書き込んでいます。

▼リスト 7.21　光の反射具合の計算

```
// call lighting function to output g-buffer
outEmission = LightingStandard_Deferred(o, worldViewDir, gi,
                                        outDiffuse,
                                        outSpecSmoothness,
                                        outNormal);
outDiffuse.a = 1.0;

#ifndef UNITY_HDR_ON
outEmission.rgb = exp2(-outEmission.rgb);
#endif
```

諸々の計算結果を LightingStandard_Deferred() に渡して光の反射具合を計算して、Emission バッファに書き込みます。HDR の場合は、exp で圧縮される部分を挟

んでから書き込みます。

7.4.5 影のジオメトリシェーダ

実体のジオメトリシェーダとほとんど同じです。違いがある所だけ解説します。

▼リスト 7.22 影のジオメトリシェーダ

```
int vindex = 0;
int findex = 0;
for (i = 0; i < 5; i++) {
  findex = flagIndex * 16 + 3 * i;
  if (triangleConnectionTable[findex] < 0)
    break;

  for (j = 0; j < 3; j++) {
    vindex = triangleConnectionTable[findex + j];

    float4 ppos = mul(_Matrix, float4(edgeVertices[vindex], 1));

    float3 norm;
    norm = UnityObjectToWorldNormal(normalize(edgeNormals[vindex]));

    float4 lpos1 = mul(unity_WorldToObject, ppos);
    o.pos = UnityClipSpaceShadowCasterPos(lpos1,
                                          normalize(
                                            mul(_Matrix,
                                              float4(norm, 0)
                                            )
                                          )
                                         );
    o.pos = UnityApplyLinearShadowBias(o.pos);
    o.hpos = o.pos;

    outStream.Append(o);
  }
  outStream.RestartStrip();
}
```

UnityClipSpaceShadowCasterPos() と UnityApplyLinearShadowBias() で頂点座標を影の投影先の座標に変換します。

7.4.6 影のフラグメントシェーダ

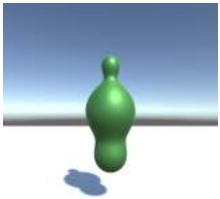
▼リスト 7.23 影のフラグメントシェーダ

```
// 影のフラグメントシェーダ
fixed4 frag_shadow(g2f_shadow i) : SV_Target
{
  return i.hpos.z / i.hpos.w;
}
```

短すぎて説明するところがないです。実は return 0; でも正常に影が描画されます。Unity が中でいい感じにやってくれているんでしょうか？

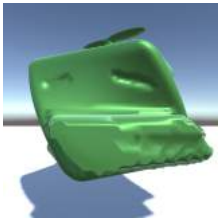
7.5 完成

実行するとこんな感じの絵が出てくるはずです。



▲図 7.5　うねうね

また、距離関数を組み合わせるといろいろな形が作れます。



▲図 7.6　かいわれーい

7.6 まとめ

今回は簡略化のために距離関数を使いましたが、他にも 3D テクスチャにボリュームデータを書き込んだものを使ったり、いろいろな三次元データを可視化するのにマーチングキューブス法は使えると思います。
ゲーム用途では、地形を掘ったり盛ったりできる ASTORONEER[*2]のようなゲーム

[*2] ASTRONEER http://store.steampowered.com/app/361420/ASTRONEER/?l=japanese

も作れるかもしれません。
みなさんもマーチングキューブス法でいろいろな表現を模索してみてください！

7.7 参考

- Polygonising a scalar field - http://paulbourke.net/geometry/polygonise/
- modeling with distance functions -

http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

第 8 章

MCMC で行う 3 次元空間サンプリング

8.1 はじめに

本章ではサンプリング手法について解説していきます。今回取り上げるのは、ある確率分布の中から適当な値を複数サンプリングしてくる MCMC（マルコフ連鎖モンテカルロ法）というサンプリング方法です。

ある確率分布からサンプリングしてくる方法として最も簡単な方法に棄却法という方法がありますが、3 次元空間でのサンプリングでは棄却される領域が大きく実際の運用に耐えません。そこで MCMC を使うことで高次元においても効率よくサンプリングできるというのが、本章の内容です。

MCMC に関する情報は、一方では書籍など体系だった情報は統計屋さん向けのものでプログラマにとっては冗長な割に実装までの手引が存在せず、他方ネットにある情報は１０数行のサンプルコードが記載されているだけで理論的な背景へのケアがないため、理論と実装を手早く一気通貫に理解できるコンテンツが存在しないのが実情です。次節以降の具体的な解説はできるだけそういった内容になるように心がけました。

MCMC の背景となる確率の解説は、厳密を期せばそれこそ本が一冊書けるほどの内容です。今回は安心して実装できる最小限の理論的背景の説明をモットーに、定義の厳密性は程々に、なるだけ直感的な表現を目指しました。数学については大学初年度程度、プログラムについては仕事で少しでも使ったことがある程度の方なら難なく読める内容かなと思います。

8.2 サンプルリポジトリ

本章では UnityGraphicsProgramming の Unity プロジェクト https://github.com/IndieVisualLab/UnityGraphicsProgramming 内にある Assets/ProceduralModeling 以下をサンプルプログラムとして用意しています。

8.3 確率に関する基礎知識

MCMC の理論を理解するには、まずは確率についての基礎的な内容を抑えておく必要があります。ただし今回 MCMC を理解するために押さえておくべき概念は少なく、以下の 4 つだけです。尤度も確率密度関数も必要なしです！

- 確率変数
- 確率分布
- 確率過程
- 定常分布

順に見ていきましょう。

8.3.1 確率変数

ある事象が確立 P(X) で起こるときの、この実数 X を確率変数と呼びます。例えば「サイコロの 5 の目が出る確率は 1/6 である」という時に「5 の目」が確率変数にあたり「1/6」が確率に当たります。先程の文を一般的に言い換えると「サイコロの X の目がでる確率は P(X) である」と言い換えることができます。

ちなみにすこし定義らしい書き方をすると、確率変数 X は標本空間 Ω（＝起こる可能性のある全ての事象）から選ばれた元 ω（＝起こった一つの事象）について、実数である X を返す写像 $X = X(\omega)$ と書くことができます。

8.3.2 確率過程

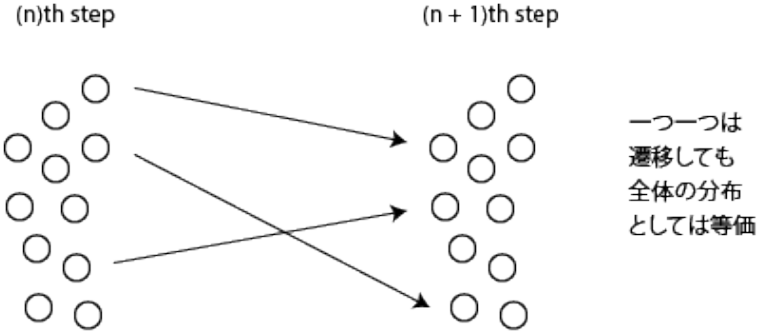
先程の確率変数の後半で若干ややこしい定義を付け加えたのは、確率変数 X が $X = X(\omega)$ という書き方で表されるという前提に立つと、確率過程の理解が簡単になるからです。確率過程とは、先程の X に時間の条件を付け加えたもので $X = X(\omega, t)$ と表すことができるもののこと。つまり確率過程は時間の条件を添えた確率変数の一種と考えることができます。

8.3.3 確率分布

確率分布は、確率変数 X と 確率 P(X) との対応関係を示すものです。よく縦軸に確率 P(X) 横軸に X を取ったグラフで表します。

8.3.4 定常分布

一つ一つの点は遷移しても全体の分布が不変であるような分布。分布 P とある遷移行列 π について、π P = P を満たす P を定常分布と呼びます。この定義だけではわかりにくいですが、以下の図を見れば明らかです。



▲図 8.1　stationaryDistribution

8.4 MCMC の概念

さて本節では MCMC を構成する概念について触れていきます。
MCMC は最初に述べたように、ある確率分布の中から適当な値をサンプリングしてくる手法なのですが、より具体的には、与えられた分布が定常分布であるという条件の下でモンテカルロ法 (Monte Carlo) とマルコフ連鎖 (Markov chain) によってサンプリングする手法を指します。以下ではモンテカルロ法、マルコフ連鎖、定常分布、の順に解説をおこなっていきます。

8.4.1 モンテカルロ法

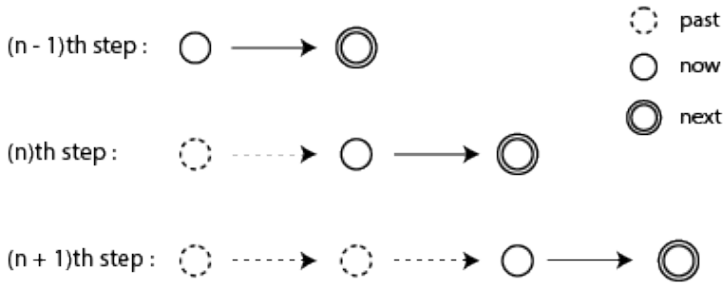
モンテカルロ法とは、擬似乱数を使った数値計算やシミュレーションの総称です。よくモンテカルロ法による数値計算の導入に使われる例に、以下のような円周率の計算があります。

```
float pi;
float trial = 10000;
float count = 0;

for(int i=0; i<trial; i++){
    float x = Random.value;
    float y = Random.value;
    if(x*x+y*y <= 1) count++;
}

pi = 4 * count / trial;
```

要するに１ｘ１の正方形の中で扇形の円の中に入った試行数と全体の試行数の比が面積比になるので、そこから円周率を出す事ができるというものです。簡単な例ですが、これもモンテカルロ法です。

8.4.2 マルコフ連鎖

マルコフ連鎖は、マルコフ性を満たす確率過程のうち、状態が離散的に記述できるものを指します。
マルコフ性とは、ある確率過程の将来状態の確率分布が現在状態のみに依存し、過去の状態に依存しない性質のことです。



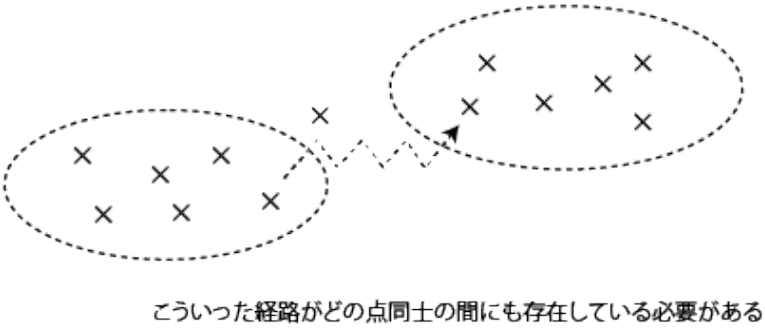
▲図 8.2　MarkovChain

上図のようにマルコフ連鎖では将来の状態は現在の状態のみに依存して、過去の状態には直接的には影響しません。

8.4.3　定常分布

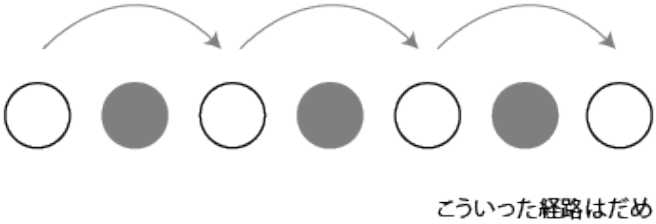
MCMC では擬似乱数を使ってある任意の分布から与えられた定常分布へと収束していく必要があります。というのも、与えられた分布に収束しないと毎回違う分布からサンプリングしてしまうし、定常分布でないと上手く連鎖的にサンプリングできません。任意の分布が与えられた分布へと収束するには、以下の二つの条件を満たす必要があります。

- 既約性・・・分布が複数の部分に別れていてはいけないという条件。確率分布上のある点から遷移を繰り返していく際に、到達できない点が存在してはならない



▲図 8.3　Irreducibility

- 非周期性・・・どんなｎに対してもｎ回で元いた場所に戻ってこれるという条件。例えば円周上に並んだ分布の中で、一つ飛ばしにしか遷移できないとった条件が存在してはならない。



▲図 8.4　Aperiodicity

この２つの条件を満たしていればある任意の分布は与えられた定常分布に収束することができます。これをマルコフ過程のエルゴード性といいます。

8.4.4 メトロポリス法

さて与えられた分布が先程のエルゴート性を満たす分布かどうかをいちいち調べるのは骨が折れることなので、多くの場合には条件を強めにとって「詳細釣り合い」という条件を満たす範囲で調べていきます。詳細釣り合いをみたすマルコフ連鎖の手法の一つがメトロポリス法と呼ばれるものです。

メトロポリス法は以下の2ステップを踏むことでサンプリングを行います

1. 擬似乱数で遷移先の候補 x を選ぶ。x は Q(x|x') = Q(x'|x) を満たすような分布 Q に従って生成され、この分布 Q を提案分布と呼ぶ。提案分布としてガウス分布が選ばれることが多い。
2. 1 と独立な乱数を発生させて、その乱数を使ってある基準が満たされれば遷移先候補を採用する。 具体的には、一様乱数 0 <= r < 1 に対して目標分布上の確率値 P(x) と遷移候補先の確率値 P(x') の比 P(x')/P(x) が、 P(x')/P(x) > r を満たせば遷移候補先へ遷移する。

メトロポリス法のメリットは、確率分布の極大値に遷移しきった後も r の値が小さければ確率値の小さい方に遷移するので、極大値周辺で確率値に比例したサンプリングができることです。

ちなみにメトロポリス法はメトロポリス・ヘイスティング法（MH 法）の一種です。メトロポリス法は提案分布に左右対称な分布を使いますが、MH 法ではこの限りではありません。

8.5 3次元サンプリング

では実際にコードの抜粋を見ながら、どのように MCMC を実装するかを見ていきましょう。

先ず3次元の確率分布を用意します。これを目標分布と呼びます。実際にサンプリングしたい分布なので「目標」分布です。

```
void Prepare()
{
    var sn = new SimplexNoiseGenerator();
    for (int x = 0; x < lEdge; x++)
        for (int y = 0; y < lEdge; y++)
            for (int z = 0; z < lEdge; z++)
            {
                var i = x + lEdge * y + lEdge * lEdge * z;
                var val = sn.noise(x, y, z);
                data[i] = new Vector4(x, y, z, val);
```

```
            }
}
```

今回はシンプレックスノイズを目標分布として採用しました。
次に実際に MCMC を走らせます。

```
public IEnumerable<Vector3> Sequence(int nInit, int limit, float th)
{
    Reset();

    for (var i = 0; i < nInit; i++)
        Next(th);

    for (var i = 0; i < limit; i++)
    {
        yield return _curr;
        Next(th);
    }
}
```

```
public void Reset()
{
     for (var i = 0; _currDensity <= 0f && i < limitResetLoopCount; i++)
     {
             _curr = new Vector3(
               Scale.x * Random.value,
               Scale.y * Random.value,
               Scale.z * Random.value
               );
             _currDensity = Density(_curr);
     }
}
```

コルーチンを使って処理を走らせます。MCMC は一つのマルコフ連鎖が終わると全く別のところから処理が始まるため、概念的には並列処理と考えることができます。今回は Reset 関数を使って、一連の処理が終わった後に別の処理を走らせるようにしています。この作業を行うことで、確率分布の極大値が多数存在する場合にも上手くサンプリングができるようになります。

遷移を始めて最初の方は目標分布から離れた点である可能性が高いので、この区間はサンプリングを行わず捨ててしまします（burn-in）。十分目標分布に近づいたらサンプリングと遷移のセットを一定回数行い、終わったらまた別の一連の処理に入ります。

最後に遷移を決定する処理です。
３次元ですので、提案分布は以下のように三変量の標準正規分布を用います。

```
public static Vector3 GenerateRandomPointStandard()
{
        var x = RandomGenerator.rand_gaussian(0f, 1f);
        var y = RandomGenerator.rand_gaussian(0f, 1f);
        var z = RandomGenerator.rand_gaussian(0f, 1f);
        return new Vector3(x, y, z);
}
```

```
public static float rand_gaussian(float mu, float sigma)
{
     float z = Mathf.Sqrt(-2.0f * Mathf.Log(Random.value))
               * Mathf.Sin(2.0f * Mathf.PI * Random.value);
     return mu + sigma * z;
}
```

メトロポリス法では左右対称な分布である必要があるので、平均値を 0 以外に設定することは無いですが、分散を 1 以外にする場合は、コレスキー分解を使って以下のように導出します。

```
public static Vector3 GenerateRandomPoint(Matrix4x4 sigma)
{
    var c00 = sigma.m00 / Mathf.Sqrt(sigma.m00);
    var c10 = sigma.m10 / Mathf.Sqrt(sigma.m00);
    var c20 = sigma.m21 / Mathf.Sqrt(sigma.m00);
    var c11 = Mathf.Sqrt(sigma.m11 - c10 * c10);
    var c21 = (sigma.m21 - c20 * c10) / c11;
    var c22 = Mathf.Sqrt(sigma.m22 - (c20 * c20 + c21 * c21));
    var r1 = RandomGenerator.rand_gaussian(0f, 1f);
    var r2 = RandomGenerator.rand_gaussian(0f, 1f);
    var r3 = RandomGenerator.rand_gaussian(0f, 1f);
    var x = c00 * r1;
    var y = c10 * r1 + c11 * r2;
    var z = c20 * r1 + c21 * r2 + c22 * r3;
    return new Vector3(x, y, z);
}
```

遷移先の決定は、提案分布（上の一点である）next と直前の点_curr それぞれの、目標分布上における確率の比を取り一様乱数より大きければ遷移、そうでなければ遷移しない、とします。
確率値は、遷移先の座標に対応する確立値を見つける処理が重いため (O(n^3) の処理量)、近似計算を行っています。今回は目標分布が連続的に変化する分布を用いているので、距離に反比例する加重平均を行うことで近似的に確立値を導出しています。

```
void Next(float threshold)
{
        Vector3 next =
          GaussianDistributionCubic.GenerateRandomPointStandard()
          + _curr;

        var densityNext = Density(next);
        bool flag1 =
          _currDensity <= 0f ||
          Mathf.Min(1f, densityNext / _currDensity) >= Random.value;
        bool flag2 = densityNext > threshold;
        if (flag1 && flag2)
        {
                _curr = next;
                _currDensity = densityNext;
        }
}

float Density(Vector3 pos)
{
        float weight = 0f;
        for (int i = 0; i < weightReferenceloopCount; i++)
        {
                int id = (int)Mathf.Floor(Random.value * (Data.Length - 1));
                Vector3 posi = Data[id];
                float mag = Vector3.SqrMagnitude(pos - posi);
                weight += Mathf.Exp(-mag) * Data[id].w;
        }
        return weight;
}
```

8.6 その他

今回リポジトリに 3 次元の棄却法（円の例で示したような簡単なモンテカルロ法）のサンプルも入っているので比較してみるとよいでしょう。棄却法では棄却の基準値を強めに取るとほとんどサンプリングが上手くできないのに対して、MCMC では同じようなサンプリング結果をよりスムーズに提示することができます。また MCMC ではステップ毎のランダムウォークの幅を小さくすれば、一連の連鎖の中では近しい空間からサンプリングするため、植物や花の群生を簡単に再現することができます。

8.7 参考文献

- 久保拓弥（2012）データ解析のための統計モデリング入門——一般化線形モデル・階層ベイズモデル・MCMC (確率と情報の科学) 岩波書店
- Olle Haggstrom, 野間口 謙太郎 (2017) やさしい MCMC 入門: 有限マルコフ連鎖とアルゴリズム 共立出版

第 9 章

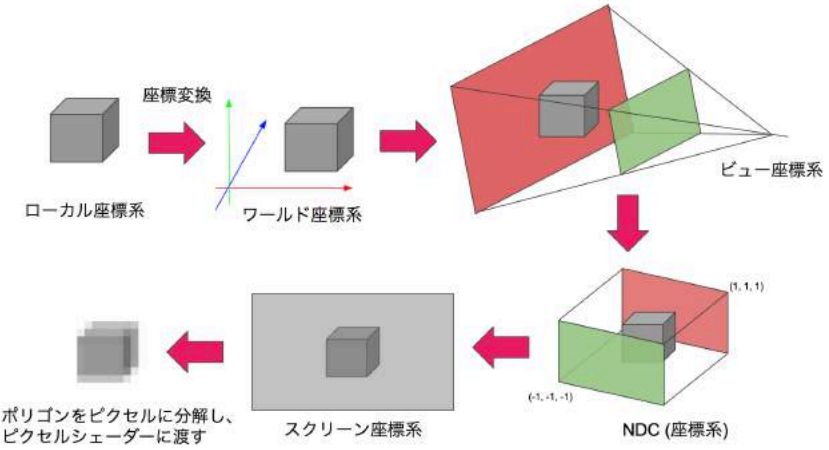
MultiPlane PerspectiveProjection

本章では直方体の形をした部屋の壁面や床面など複数の面にプロジェクターで映像を投影し CG 世界の中にいるような体験ができる映像投影方法を紹介します。また、そのバックグラウンドとして CG におけるカメラの処理とその応用例について解説します。サンプルプロジェクトは、UnityGraphicsProgramming の Unity プロジェクト[*1]内の Assets/RoomProjection にありますのでよかったらご覧ください。また、本内容は「数学セミナー 2016 年 12 月号」[*2]に 寄稿した内容を元に大幅に加筆修正を行ったものになります。

9.1 CG におけるカメラの仕組み

一般的な CG におけるカメラ処理とは、透視投影変換を用いて見えている範囲の 3 D モデルを 2 次元画像へ射影する処理を行います。透視投影変換は、各モデルの中心を原点に持つローカル座標系、CG 世界の一意に決めた場所を原点にするワールド座標系、カメラを中心としたビュー座標系、クリッピング用のクリップ座標系（これは w も意味を持つ 4 次元の座標系で、3 次元化したものを **NDC**（Normalized Device Coordinates, 正規化デバイス座標系）と呼びます）、出力画面の 2 次元の位置を表すスクリーン座標系、という順番で頂点の座標を射影していきます。

[*1] サンプルプロジェクト https://github.com/IndieVisualLab/UnityGraphicsProgramming
[*2] https://www.nippyo.co.jp/shop/magazine/7292.html

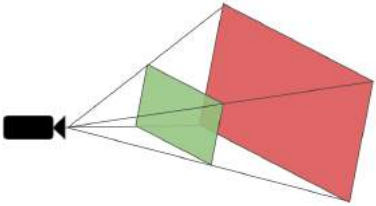


▲図 9.1　座標変換の流れ

また、これらの変換はそれぞれ 1 つの行列で表すことができるのであらかじめ行列同士を乗算しておくことで、いくつかの座標変換を行列とベクトルの乗算 1 回で済ませる方法もよく行われています。

9.2　複数カメラでのパースの整合性

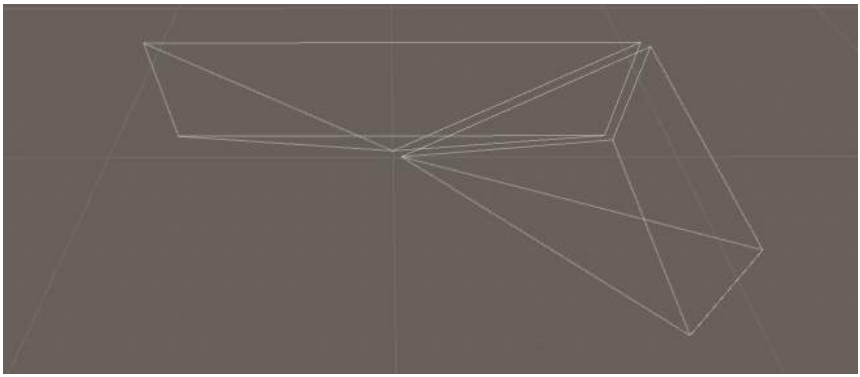
CG におけるカメラでは、頭頂点をカメラの位置に、底面をカメラの向きに合わせた四角錐を視錐台と呼び、カメラの射影を表す 3 D ボリュームとして図示できます。



▲図 9.2　視錐台

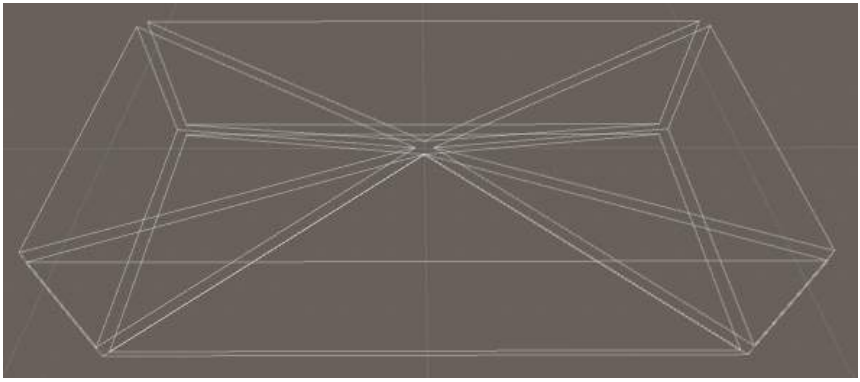
2 つのカメラの視錐台が頭頂点を共有し側面が接していれば、投影面が別々の方向

を向いていても映像的には繋がり、かつ、頭頂点から見たときのパースペクティブが一致します。



▲図 9.3　接する視錐台（わかりやすいように少し離して配置しています）

これは視錐台を無数の視線の集合とみなすことで「視線同士が連続している（＝パースペクティブ上矛盾のない映像を投影することができる）」と考えることで理解できます。この考えを５つのカメラまで拡張し、５つの視錐台が頭頂点を共有しそれぞれ隣接する視錐台と接するような配置になるよう画角を調整することで、部屋の各面に対応した映像を生成することができます。理論上は天井も含めた６面も可能ですが今回はプロジェクタの設置スペースとして考え、天井を除く５面を想定しています。



▲図 9.4　部屋に対応した５つの視錐台

この頭頂点、つまり全てのカメラの位置に相当する場所から鑑賞することで、部屋どの方向を見てもパースペクティブ上矛盾のない映像を鑑賞することができます。

9.3 プロジェクション行列の導出

プロジェクション行列（以下 $Proj$ ）はビュー座標系からクリップ座標系へ変換する行列です。

- C：クリップ座標系おける位置ベクトル
- V：をビュー座標系における位置ベクトル

として式で表すと以下のようになります。

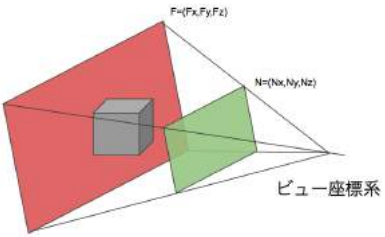
$$C = Proj * V$$

さらに C の各要素を C_w で除算することで NDC での位置座標となります。

$$NDC = (\frac{C_x}{C_w}, \frac{C_y}{C_w}, \frac{C_z}{C_w})$$

なお、 $C_w = -V_z$ とな（るように $Proj$ を作）ります。ビュー座標系の正面方向が Z マイナス方向なのでマイナスがかかっています。NDC では表示範囲を $-1 \leq x, y, z \leq 1$ とし、この変換で V_z に応じて $V_{x,y}$ が拡大縮小することにより遠近法の表現が得られます。

それでは、 $Proj$ をどのように作ればよいか考えてみましょう。ビュー座標系における nearClipPlane の右上の点の座標を N 、farClipPlane の右上の点の座標を F としておきます。



▲図 9.5　N,F

まずは x に注目してみると、

- 投影範囲が $-1 \leq x \leq 1$ になること

- あとで $C_w(=-V_z)$ で除算されること

を考慮すると

$$Proj[0,0]=\frac{N_z}{N_x}$$

とすれば良さそうです。x,z の比率は変わらないので $Proj[0][0]=\frac{F_z}{F_x}$ など視錐台の右端ならどの x,z でも構いません。

同様に

$$Proj[1,1]=\frac{N_z}{N_y}$$

も求まります。

z については少し工夫が必要です。$Proj*V$ で z に関わる計算は以下ようになります。

$$C_z=Proj[2,2]*V_z+Proj[2,3]*V_w \quad (\text{ただし、} V_w=1)$$

$$NDC_z=\frac{C_z}{C_w}(\text{ただし、} C_w=-V_z)$$

ここで、 $N_z\rightarrow -1, F_z\rightarrow 1$ と変換したいので、 $a=Proj[2,2], b=Proj[2,3]$ と置いて

$$-1=\frac{1}{N_z}(aN_z+b), \quad 1=\frac{1}{F_z}(aF_z+b)$$

この連立方程式から解が得られます。

$$Proj[2,2]=a=\frac{F_z+N_z}{F_z-N_z}, \quad Proj[2,3]=b=\frac{-2F_zN_z}{F_z-N_z}$$

また、 $C_w=-V_w$ となるようにしたいので

$$Proj[3,2]=-1$$

とします。

したがって求める $Proj$ は以下の形になります。

$$Proj=\begin{pmatrix} \frac{N_z}{N_x} & 0 & 0 & 0 \\ 0 & \frac{N_z}{N_y} & 0 & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

9.3.1 Camera.projectionMatrix の罠

シェーダー内などでプロジェクション行列を扱ったことがある方の中にはもしかしたらここまでの内容に違和感を持つ方もいらっしゃるかもしれません。実は Unity のプロジェクション行列の扱いはややこしく、ここまでの内容は Camera.projectionMatrix についての解説になります。この値はプラットフォームによらず OpenGL に準拠しています[*3]。 $-1 \leq NDC_z \leq 1$ や $C_w = -V_w$ となるのもこのためです。

しかし Unity 内でシェーダーに渡す際にプラットフォームに依存した形に変換するため、Camera.projectionMatrix をそのまま透視投影変換に使っているとは限りません。とくに NDC_z の範囲や向き（つまり Z バッファの扱い）は多様でひっかかりやすいポイントになっています[*4]。

9.4 視錐台の操作

9.4.1 投影面のサイズ合わせ

視錐台の底面つまり投影面の形はカメラの **fov**（fieldOfView, 画角）と **aspect**（アスペクト比）に依存しています。Unity のカメラでは画角は Inspector で公開されているものの、アスペクト比は公開されていないのでコードから編集する必要があります。**faceSize**（部屋の面のサイズ）、**distance**（視点から面までの距離）から画角とアスペクト比を求めるコードは以下のようになります。

▼リスト 9.1　画角とアスペクト比を求める

```
camera.aspect = faceSize.x / faceSize.y;
camera.fieldOfView = 2f * Mathf.Atan2(faceSize.y * 0.5f, distance)
                    * Mathf.Rad2Deg;
```

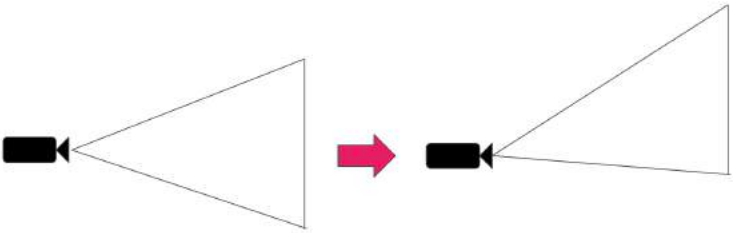
Mathf.Atan2() で fov の半分の角度を radian で求め、2 倍し、Camera.fieldOfView に代入するため degree に直している点に注意して下さい。

9.4.2 投影面の移動（レンズシフト）

視点が部屋の中心にない場合も考慮してみましょう。視点に対して投影面が上下左右に平行移動することができれば、投影面に対して視点が移動したことと同じ効果が

[*3] https://docs.unity3d.com/ScriptReference/GL.GetGPUProjectionMatrix.html
[*4] https://docs.unity3d.com/Manual/SL-PlatformDifferences.html

得られます。これは現実世界ではプロジェクターなどで映像の投影位置を調整するレンズシフトという機能に相当します。



▲図 9.6　レンズシフト

あらためてカメラが透視投影する仕組みに立ち返ってみるとレンズシフトはどの部分で行う処理になるでしょうか？ プロジェクション行列で NDC に射影する際に、x,y をずらせば良さそうですもう一度 Projection 行列を見てみましょう。

$$Proj = \begin{pmatrix} \frac{N_z}{N_x} & 0 & 0 & 0 \\ 0 & \frac{N_z}{N_y} & 0 & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

C_x, C_y がずれればいいので、行列の平行移動成分である $Proj[0,3], Pro[1,3]$ になにか入れたくなりますが、あとで C_w で除算されることを考慮すると、$Proj[0,2], Pro[1,2]$ に入れるのが正解です。

$$Proj = \begin{pmatrix} \frac{N_z}{N_x} & 0 & LensShift_x & 0 \\ 0 & \frac{N_z}{N_y} & LensShift_y & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

LensShift の単位は NDC ですので投影面のサイズを-1〜1 に正規化したものになります。コードにすると以下のようになります。

▼リスト 9.2　レンズシフトをプロジェクション行列に反映

```
var shift = new Vector2(
    positionOffset.x / faceSize.x,
    positionOffset.y / faceSize.y
) * 2f;
var projectionMatrix = camera.projectionMatrix;
projectionMatrix[0,2] = shift.x;
projectionMatrix[1,2] = shift.y;
camera.projectionMatrix = projectionMatrix;
```

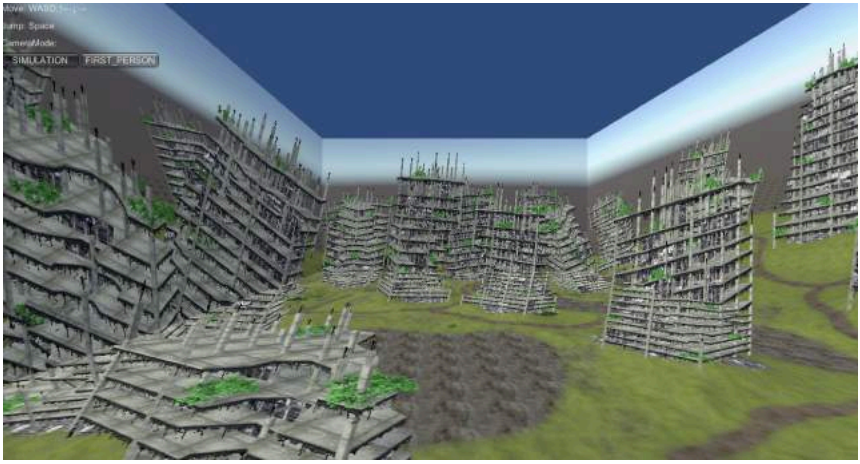
一度 Camera.projectionMatrix に set すると Camera.ResetProjectionMatrix() を呼ばない限りその後の Camera.fieldOfView の変更が反映されなくなる点に注意が必要です。[*5]

9.5 部屋プロジェクション

直方体の部屋で、鑑賞者の視点位置をトラッキングできているものとします。前節の方法で視錐台の投影面のサイズと平行移動ができるので、視点位置を視錐台の頭頂点、壁面や床面を投影面としたときその形状に合うような視錐台を動的に求める事ができます。各カメラをこのような視錐台にすることによって各投影面用の映像を作ることができます。この映像を実際の部屋に投影すれは鑑賞者からはパースのあった CG 世界が見えるようになります。

▲図 9.7　部屋のシミュレーション（俯瞰視点）

[*5] https://docs.unity3d.com/ScriptReference/Camera-projectionMatrix.html



▲図 9.8　部屋のシミュレーション（一人称視点）

9.6　まとめ

本章ではプロジェクション行列を応用することで複数の投影面でパースを合わせる投影方法を紹介しました。目の前にディスプレイを置くのではなく視界の広い範囲を動的に反応する映像にしてしまう点で、昨今の HMD 型と似て非なるアプローチの VR と言えるのではないかと思います。また、この方法では両眼視差や目のフォーカスを騙せるわけではないのでそのままでは立体視できずに「壁に投影された動く絵」に見えてしまう可能性があります。没入感を高めるためにはもう少し工夫する必要がありそうです。

- 両眼視差が小さくなるように部屋を大きくして鑑賞者から投影面までの距離を遠くする
- 反射光などで投影面の平面が意識されてしまうことをできるだけ防ぐ
  - 暗めの映像にする
  - 壁や床をできるだけ鏡面反射しない素材にする

なお、同様の手法を立体視と組み合わせる「CAVE」[*6]という仕組みが知られています。

[*6] https://en.wikipedia.org/wiki/Cave_automatic_virtual_environment

第 10 章

ProjectionSpray の紹介

10.1 はじめに

こんにちは！ すぎのひろのりです！ では残念ながらありません。

締め切りも近づいてきたある日、「すぎっちょ記事書いてる？」と問うたところ「あ！」とだけおっしゃり、どうやら完全に失念されておったようです。最近忙しそうなのですが、せっかくのこの機会、彼の実績を紹介したいのもあり、ここは簡単に代筆でお送りします。

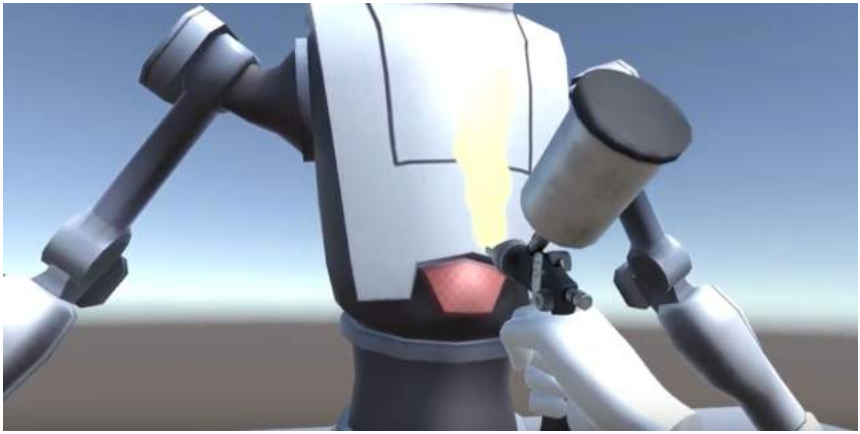
10.1.1 ProjectionSpray

すぎっちょは制作物を Github に積極的に公開しており、その中でも個人的に面白いなと思ったのがこちらです。

https://github.com/sugi-cho/ProjectionSpray

3D モデルにスプレーを吹きかけるようにして色をつけることができます。

デモ画像



▲図 10.1　デモ画像 1

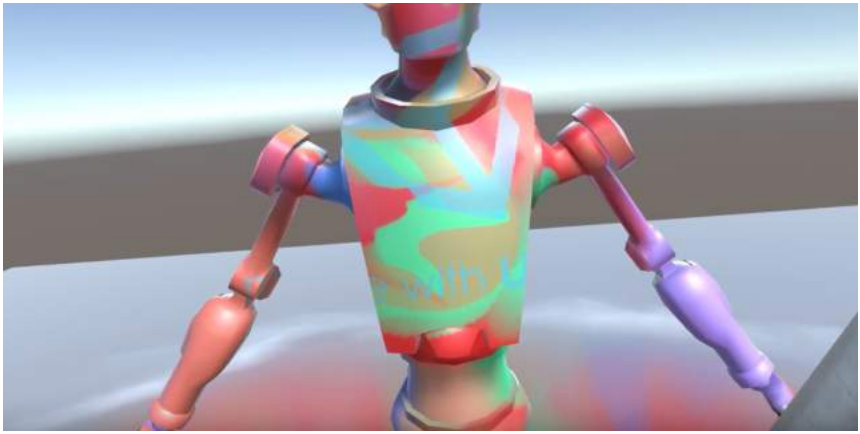
スプレーデバイスからスプレーが噴出し、ボディの表面に色が塗られます。

▲図 10.2　デモ画像 2

謎のフェチズムを感じます。

▲図 10.3　デモ画像 3

ステンシルのようなことも！



▲図 10.4　デモ画像 4

Unity！

10.2　まとめ

次回があれば、是非とも詳しい解説をお願いしたいものです。

すぎっちょが、面接時、自分と、似たものを、感じる、と言っていた同僚の仲田さんのリポジトリも、Unity の便利で優れたコードが多数あげられておりおすすめです。

https://github.com/nobnak

失礼いたしました。

(。･ˇ _ˇ･。)

著者紹介

第 1 章 Unity ではじめるプロシージャルモデリング - 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など、来た球はなるべく全部打つようにしています。

- https://twitter.com/mattatz
- https://github.com/mattatz
- http://mattatz.org/

第 2 章 ComputeShader 入門 - @XJINE

勢いと雰囲気だけで生きてるうちに、唐突にインタラクティブアーティスト・エンジニアになってしまって、とても大変なことになってしまった。周りの人たちに助けられながら、勉強させてもらいながら、なんとかやっています。

- https://twitter.com/XJINE
- https://github.com/XJINE
- http://neareal.com/

第 3 章 群のシミュレーションの GPU 実装 - 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 4 章 格子法による流体シミュレーション - 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 5 章 SPH 法による流体シミュレーション - 高尾航大 / @kodai100

インタラクティブアーティスト・エンジニア兼、学生。大学で雪の物理シミュレーションを研究している傍らエンジニアリングに勤しんでいる。最近は TouchDesigner に浮気中。是非 twitter でお話ししましょう。

- https://twitter.com/m1ke_wazowsk1
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

第 6 章 ジオメトリシェーダで草を生やす - @a3geek

インタラクションエンジニア・雑魚系疾風のプログラマ・ゆるふわガチ勢・わりと何でも作る何でも屋さん。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 7 章 雰囲気ではじめるマーチングキューブス法入門 - @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/

第 8 章 MCMC で行う 3 次元空間サンプリング - @ komietty

インタラクティブエンジニア。Web 制作、グラフィックデザインのお仕事も個人でやってます。制作のご依頼は twitter まで。

- https://github.com/komietty
- https://twitter.com/9_chinashi

第 9 章 MultiPlanePerspectiveProjection - 福永秀和 / @fuqunaga

元ゲーム開発者、現インタラクティブアーティスト・エンジニア。健康に気を使おうと朝ごはんを食べるようにしたらなぜか２ｋｇほど痩せた。

- https://twitter.com/fuqunaga
- https://github.com/fuqunaga
- https://fuquna.ga

第 10 章 ProjectionSpray の紹介 - 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。hi@sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc

権利について

Unity Graphics Programming

2017 年 10 月 22 日　技術書典 3 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/