Unity Graphics Programming

Unityグラフィックスプログラミング vol.4

IndieVisualLab

Unity Graphics Programming vol.4

IndieVisualLab　著

2019-04-14 版　　**IndieVisualLab** 発行

まえがき

本書は Unity によるグラフィクスプログラミングに関する技術を解説する「Unity グラフィックスプログラミング」シリーズの第三巻です。本シリーズでは、執筆者たちの興味の赴くままに取り上げられた様々なトピックについて、初心者向けの入門的な内容と応用を解説したり、中級者以上向けの tips を掲載しています。

各章で解説されているソースコードについては github リポジトリ (https://github.com/IndieVisualLab/UnityGraphicsProgramming4)にて公開していますので、手元で実行しながら本書を読み進めることができます。

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

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

推奨する実行環境

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

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

本についての要望や感想

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

目次

第 1 章

GPU-Based Space Colonization Algorithm

本章では、点群に沿ってブランチングする形状を生成するアルゴリズム、Space Colonization Algorithm の GPU 実装とその応用例を紹介します。

本章のサンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming4`
の「SpaceColonization」です。

▲図 1.1　SkinnedAnimation.scene

1.1 はじめに

Space Colonization Algorithm は、Adam ら[*1]による樹木のモデリング手法として開発されました。

与えられた点群からブランチングする形状を生成する手法で、

- 枝同士が密着しすぎず、程よくばらけさせながらブランチングさせられる
- 初期の点群配置によって枝の配置が決まるので、形状をコントロールしやすい
- シンプルなパラメータで枝の粗密具合をコントロールできる

という特徴を持ちます。

本章ではこのアルゴリズムの GPU 実装と、スキニングアニメーションと組み合わせた応用例について紹介します。

1.2 アルゴリズム

まずは Space Colonization Algorithm を解説します。アルゴリズムの大まかなステップは以下のように分けられます。

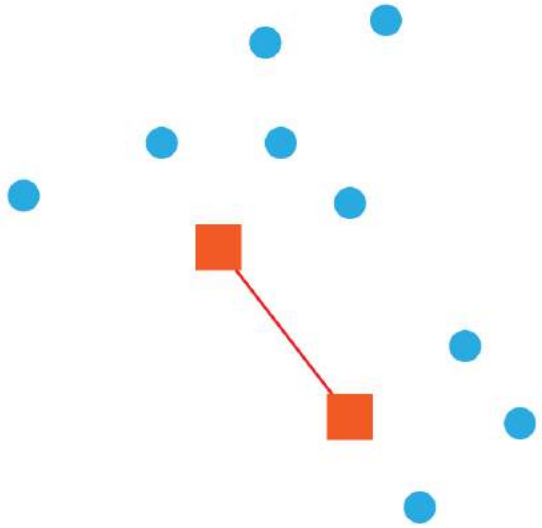
1. Setup - 初期化
2. Search - 影響する Attraction の検索
3. Attract - ブランチの引き寄せ
4. Connect - 新規 Node の生成と既存 Node との接続
5. Remove - Attraction の削除
6. Grow - Node の成長

1.2.1 Setup - 初期化

初期化フェーズでは、点群を Attraction（ブランチのシードとなる点）として用意します。その Attraction が散りばめられた中に、Node（ブランチの分岐点）を 1 つ以上配置します。この最初に配置された Node がブランチの開始点となります。

以下の図中では、Attraction を丸い点、Node を四角い点で表しています。

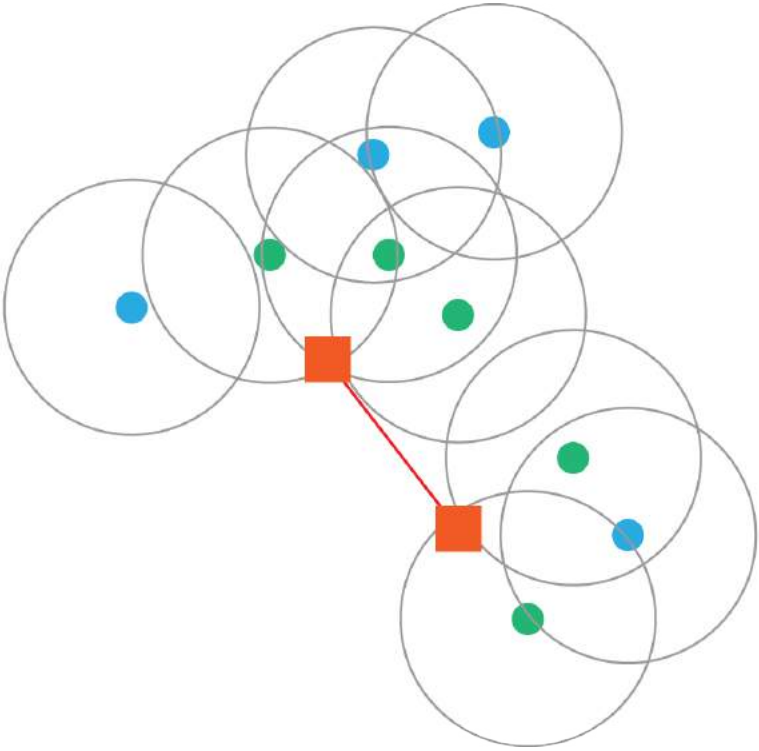
[*1] http://algorithmicbotany.org/papers/colonization.egwnp2007.html



▲図 1.2　Setup - Attraction と Node の初期化　丸い点が Attraction、四角い点が Node を表している

1.2.2　Search - 影響する Attraction の検索

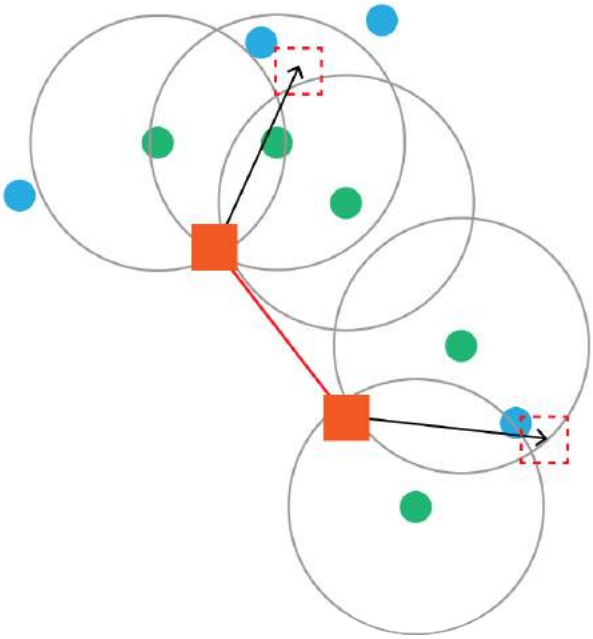
それぞれの Attraction について、影響範囲（influence distance）内で最も近傍にある Node を検索します。



▲図 1.3　Search - それぞれの Attraction から影響範囲内で最も近傍の Node を検索する

1.2.3 Attract - ブランチの引き寄せ

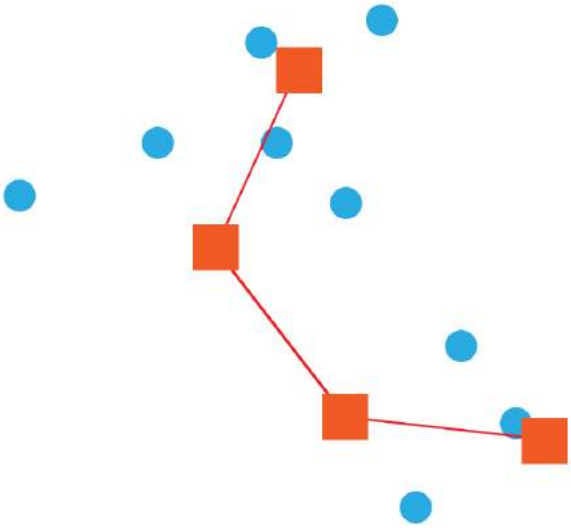
それぞれの Node について、影響範囲内にある Attraction を元にブランチを伸ばす方向を決め、成長の長さ（growth distance）分だけ伸ばした先の点を、新たに Node を生成する点の候補点（Candidate）とします。



▲図 1.4　Attract - それぞれの Node からブランチを伸ばして新たに Node を生成する候補点を決める

1.2.4　Connect - 新たな Node と既存の Node との接続

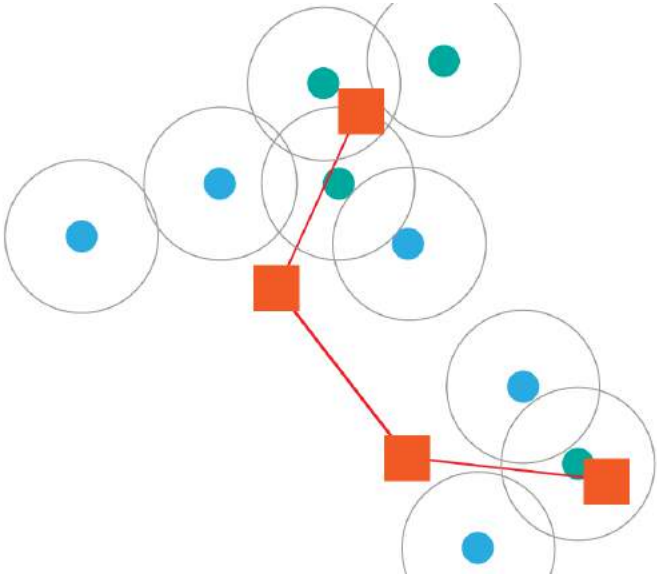
Candidate の位置に新たな Node を生成し、元の Node と Edge で繋いでブランチを拡張します。



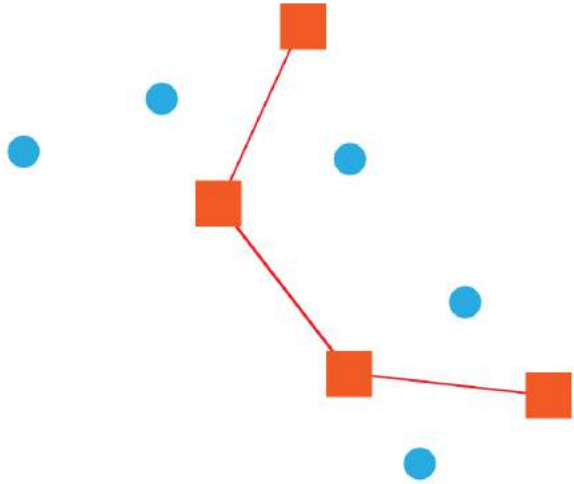
▲図 1.5　Connect - 新たな Node と既存 Node とを接続してブランチを拡張する

1.2.5　Remove - 削除範囲内にある Attraction の削除

Node から削除範囲（kill distance）内にある Attraction を削除します。



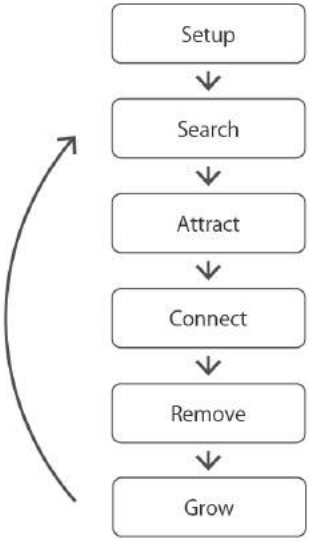
▲図 1.6　Remove - Node から削除範囲内にある Attraction を検索する



▲図 1.7　Remove - 削除範囲内に見つかった Attraction を削除する

1.2.6 Grow - Node の成長

Node を成長させ、Step.2 に戻ります。
アルゴリズム全体の大まかな流れは以下の図のようになります。



▲図 1.8　アルゴリズムの大まかな流れ

1.3 実装

それではアルゴリズムの具体的な実装について解説していきます。

1.3.1 リソースの用意

Space Colonization Algorithm では増減する要素として

- Attraction：ブランチのシードとなる点群
- Node：ブランチの分岐点（ノード）
- Candidate：新たにノードを生成する候補となる点

- Edge：ブランチのノード間をつなぐエッジ

が必要となりますが、これらを GPGPU 上で表現するために、いくつかの要素に Append/ConsumeStructuredBuffer を利用します。

Append/ConsumeStructuredBuffer については Unity Graphics Programming vol.3「GPU-Based Cellular Growth Simulation」で解説しています。

Attraction（ブランチのシードとなる点）

Attraction の構造は以下のように定義します。

▼ Attraction.cs

```
public struct Attraction {
    public Vector3 position; // 位置
    public int nearest; // 最近傍 Node の index
    public uint found; // 近傍 Node が見つかったかどうか
    public uint active; // 有効な Attraction かどうか（1 なら有効、0 なら削除済み）
}
```

active フラグによって削除済みの Attraction かどうかを判別することで、Attraction の増減を表現します。

Space Colonization では Attraction の点群を初期化フェーズで用意する必要があります。サンプルの SpaceColonization.cs では球形の内部に点群をランダムに散りばめ、Attraction の位置として利用しています。

▼ SpaceColonization.cs

```
    // 球形の内部にランダムに点を散りばめて Attraction を生成する
    var attractions = GenerateSphereAttractions();
    count = attractions.Length;

    // Attraction バッファの初期化
    attractionBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(Attraction)),
        ComputeBufferType.Default
    );
    attractionBuffer.SetData(attractions);
```

Node（ブランチの分岐点）

Node の構造は以下のように定義します。

▼ Node.cs

```
public struct Node {
    public Vector3 position; // 位置
    public float t; // 成長率 (0.0 ~ 1.0)
    public float offset; // Root からの距離 (Node の深さ)
    public float mass; // 質量
    public int from; // 分岐元 Node の index
    public uint active; // 有効な Node かどうか (1 なら有効)
}
```

Node のリソースは

- Node の実データを表すバッファ
- オブジェクトプールとして利用する、active でない Node の index を管理するバッファ

の 2 つのバッファで管理します。

▼ SpaceColonization.cs

```
// Node の実データ
nodeBuffer = new ComputeBuffer(
  count,
  Marshal.SizeOf(typeof(Node)),
  ComputeBufferType.Default
);

// オブジェクトプール
nodePoolBuffer = new ComputeBuffer(
  count,
  Marshal.SizeOf(typeof(int)),
  ComputeBufferType.Append
);
nodePoolBuffer.SetCounterValue(0);
```

Candidate（新たな Node の候補点）

Candidate の構造は以下のように定義します。

▼ Candidate.cs

```
public struct Candidate
{
    public Vector3 position; // 位置
    public int node; // 候補点の元 Node の index
}
```

Candidate は Append/ConsumeStructuredBuffer で表現します。

▼ SpaceColonization.cs

```
    candidateBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(Candidate)),
        ComputeBufferType.Append
    );
    candidateBuffer.SetCounterValue(0);
```

Edge（Node 間をつなぐ Edge）

Edge の構造は以下のように定義します。

▼ Edge.cs

```
public struct Edge {
    public int a, b; // Edge が繋ぐ 2 つの Node の index
}
```

Edge は Candidate と同様に Append/ConsumeStructuredBuffer で表現します。

▼ SpaceColonization.cs

```
    edgeBuffer = new ComputeBuffer(
        count * 2,
        Marshal.SizeOf(typeof(Edge)),
        ComputeBufferType.Append
    );
    edgeBuffer.SetCounterValue(0);
```

1.3.2 ComputeShader でのアルゴリズムの実装

これで必要なリソースがそろったので、アルゴリズムの各ステップを ComputeShader による GPGPU で実装していきます。

Setup

初期化フェーズでは、

- Node のオブジェクトプールの初期化
- 初期 Node をシードとして追加

を行います。

用意した Attraction からいくつかをピックアップし、その位置に初期 Node を生成します。

▼ SpaceColonization.cs

```
    var seeds = Enumerable.Range(0, seedCount).Select((_) => {
        return attractions[Random.Range(0, count)].position;
    }).ToArray();
    Setup(seeds);
```

▼ SpaceColonization.cs

```
protected void Setup(Vector3[] seeds)
{
    var kernel = compute.FindKernel("Setup");
    compute.SetBuffer(kernel, "_NodesPoolAppend", nodePoolBuffer);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
    GPUHelper.Dispatch1D(compute, kernel, count);

    ...
}
```

Setup カーネルではオブジェクトプールの初期化を行います。Node のオブジェクトプールに index を格納し、該当する Node の active フラグをオフにします。

▼ SpaceColonization.compute

```
void Setup (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Nodes.GetDimensions(count, stride);
  if (idx >= count)
    return;

  _NodesPoolAppend.Append(idx);

  Node n = _Nodes[idx];
  n.active = false;
  _Nodes[idx] = n;
}
```

これですべての Node の active フラグがオフの状態となり、Node の index を持ったオブジェクトプールが生成されます。

オブジェクトプールの初期化が済んだので、次はシードとなる初期ノードを生成します。

先ほど用意したシード位置（Vector3[]）を入力として、Seed カーネルを実行することで初期ノードを生成します。

▼ SpaceColonization.cs

```
...

// seedBuffer はスコープを抜けると自動的に Dispose される
using(
```

```
    ComputeBuffer seedBuffer = new ComputeBuffer(
        seeds.Length,
        Marshal.SizeOf(typeof(Vector3))
    )
)
{
    seedBuffer.SetData(seeds);
    kernel = compute.FindKernel("Seed");
    compute.SetFloat("_MassMin", massMin);
    compute.SetFloat("_MassMax", massMax);
    compute.SetBuffer(kernel, "_Seeds", seedBuffer);
    compute.SetBuffer(kernel, "_NodesPoolConsume", nodePoolBuffer);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
    GPUHelper.Dispatch1D(compute, kernel, seedBuffer.count);
}

// Node と Edge の数の初期化
nodesCount = nodePoolBuffer.count;
edgesCount = 0;

...
```

Seed カーネルは、Seeds バッファから位置を取り出し、その位置に Node を生成します。

▼ SpaceColonization.compute

```
void Seed (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;

  uint count, stride;
  _Seeds.GetDimensions(count, stride);
  if (idx >= count)
    return;

  Node n;

  // 新たな Node を生成 (後述)
  uint i = CreateNode(n);

  // Seed の位置を Node の position に設定
  n.position = _Seeds[idx];
  n.t = 1;
  n.offset = 0;
  n.from = -1;
  n.mass = lerp(_MassMin, _MassMax, nrand(id.xy));
  _Nodes[i] = n;
}
```

CreateNode 関数で新たな Node を生成します。オブジェクトプールである ConsumeStructuredBuffer から index を取り出し、初期化した Node を返します。

▼ SpaceColonization.compute

```
uint CreateNode(out Node node)
{
  uint i = _NodesPoolConsume.Consume();
  node.position = float3(0, 0, 0);
  node.t = 0;
  node.offset = 0;
  node.from = -1;
  node.mass = 0;
  node.active = true;
  return i;
}
```

これで初期化フェーズは終わりです。

図 1.8 で示しているループするアルゴリズムの各ステップは Step 関数内で実行しています。

▼ SpaceColonization.cs

```
protected void Step(float dt)
{
    // オブジェクトプールが空のときは実行しないようにする
    if (nodesCount > 0)
    {
        Search();   // Step.2
        Attract();  // Step.3
        Connect();  // Step.4
        Remove();   // Step.5

        // Append/ConsumeStructuredBuffer が持つデータ数を取得
        CopyNodesCount();
        CopyEdgesCount();
    }
    Grow(dt);       // Step.6
}
```

Search

各 Attraction から、影響範囲（influence distance) 内で最も近傍にある Node を検索します。

▼ SpaceColonization.cs

```
protected void Search()
{
    var kernel = compute.FindKernel("Search");
    compute.SetBuffer(kernel, "_Attractions", attractionBuffer);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
    compute.SetFloat("_InfluenceDistance", unitDistance * influenceDistance);
    GPUHelper.Dispatch1D(compute, kernel, count);
}
```

GPU カーネルの実装は以下の通りです。

▼ SpaceColonization.compute

```
void Search (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Attractions.GetDimensions(count, stride);
  if (idx >= count)
    return;

  Attraction attr = _Attractions[idx];

  attr.found = false;
  if (attr.active)
  {
    _Nodes.GetDimensions(count, stride);

    // influence distance よりも近傍の Node を探索する
    float min_dist = _InfluenceDistance;

    // 最も近傍の Node の index
    uint nearest = -1;

    // すべての Node についてループを実行
    for (uint i = 0; i < count; i++)
    {
      Node n = _Nodes[i];

      if (n.active)
      {
        float3 dir = attr.position - n.position;
        float d = length(dir);
        if (d < min_dist)
        {
          // 最も近傍の Node を更新
          min_dist = d;
          nearest = i;

          // 近傍 Node の index を設定する
          attr.found = true;
          attr.nearest = nearest;
        }
      }
    }

    _Attractions[idx] = attr;
  }
}
```

Attract

それぞれの Node について、影響範囲内にある Attraction を元にブランチを伸ばす方向を決め、成長の長さ（growth distance）分だけ伸ばした先の点を、新たに Node を生成する点の候補点（Candidate）とします。

▼ SpaceColonization.cs

```
protected void Attract()
{
    var kernel = compute.FindKernel("Attract");
    compute.SetBuffer(kernel, "_Attractions", attractionBuffer);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);

    candidateBuffer.SetCounterValue(0); // 候補点を格納するバッファの初期化
    compute.SetBuffer(kernel, "_CandidatesAppend", candidateBuffer);

    compute.SetFloat("_GrowthDistance", unitDistance * growthDistance);

    GPUHelper.Dispatch1D(compute, kernel, count);
}
```

GPU カーネルの実装は以下の通りです。候補点の位置の計算方法は Attract カーネルのコード内容とコメントを参照してください。

▼ SpaceColonization.compute

```
void Attract (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Nodes.GetDimensions(count, stride);
  if (idx >= count)
    return;

  Node n = _Nodes[idx];

  // Node が有効かつ、
  // 成長率 (t) が閾値 (1.0) 以上あれば新たな Node を生成する
  if (n.active && n.t >= 1.0)
  {
    // ブランチを伸ばす先の累算用変数
    float3 dir = (0.0).xxx;
    uint counter = 0;

    // 全 Attraction についてループを実行する
    _Attractions.GetDimensions(count, stride);
    for (uint i = 0; i < count; i++)
    {
      Attraction attr = _Attractions[i];
      // 該当 Node が最近傍である Attraction を検索する
      if (attr.active && attr.found && attr.nearest == idx)
      {
        // Node から Attraction へ向かうベクトルを正規化して累算用変数に加算する
        float3 dir2 = (attr.position - n.position);
        dir += normalize(dir2);
        counter++;
      }
    }

    if (counter > 0)
    {
```

```
        Candidate c;

        // Node から Attraction へ向かう単位ベクトルの平均を取り、
        // それを growth distance 分、Node から伸ばした先を候補点の位置とする
        dir = dir / counter;
        c.position = n.position + (dir * _GrowthDistance);

        // 候補点へ伸びる元 Node の index を設定する
        c.node = idx;

        // 候補点バッファに加える
        _CandidatesAppend.Append(c);
      }
    }
  }
```

Connect

Attract カーネルで生成された候補点バッファを元に新たな Node を生成し、Node 同士を Edge で繋げることでブランチを拡張します。

Connect 関数では、Node オブジェクトプール（nodePoolBuffer）が空の状態でデータの取り出し（Consume）が実行されないよう、オブジェクトプールの残り数（nodesCount）と候補点バッファのサイズを比較してカーネルの実行回数を決めています。

▼ SpaceColonization.cs

```
protected void Connect()
{
    var kernel = compute.FindKernel("Connect");
    compute.SetFloat("_MassMin", massMin);
    compute.SetFloat("_MassMax", massMax);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
    compute.SetBuffer(kernel, "_NodesPoolConsume", nodePoolBuffer);
    compute.SetBuffer(kernel, "_EdgesAppend", edgeBuffer);
    compute.SetBuffer(kernel, "_CandidatesConsume", candidateBuffer);

    // CopyNodeCount で取得した Node オブジェクトプールの持つデータ数 (nodeCount) を
    // 越えないように制限をかける
    var connectCount = Mathf.Min(nodesCount, CopyCount(candidateBuffer));
    if (connectCount > 0)
    {
        compute.SetInt("_ConnectCount", connectCount);
        GPUHelper.Dispatch1D(compute, kernel, connectCount);
    }
}
```

以下が GPU カーネルの実装になります。

▼ SpaceColonization.compute

```
void Connect (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  if (idx >= _ConnectCount)
    return;

  // 候補点バッファから候補点を取り出し
  Candidate c = _CandidatesConsume.Consume();

  Node n1 = _Nodes[c.node];
  Node n2;

  // 候補点の位置に Node を生成
  uint idx2 = CreateNode(n2);
  n2.position = c.position;
  n2.offset = n1.offset + 1.0; // Root からの距離を設定 (元 Node + 1.0)
  n2.from = c.node; // 元 Node の index を設定
  n2.mass = lerp(_MassMin, _MassMax, nrand(float2(c.node, idx2)));

  // Node バッファを更新
  _Nodes[c.node] = n1;
  _Nodes[idx2] = n2;

  // 2 つの Node を Edge で繋ぐ (後述)
  CreateEdge(c.node, idx2);
}
```

CreateEdge 関数は渡された 2 つの Node の index を元に Edge を生成し、Edge バッファに追加します。

▼ SpaceColonization.compute

```
void CreateEdge(int a, int b)
{
  Edge e;
  e.a = a;
  e.b = b;
  _EdgesAppend.Append(e);
}
```

Remove

Node から kill distance 内にある Attraction を削除します。

▼ SpaceColonization.cs

```
protected void Remove()
{
    var kernel = compute.FindKernel("Remove");
    compute.SetBuffer(kernel, "_Attractions", attractionBuffer);
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
    compute.SetFloat("_KillDistance", unitDistance * killDistance);
    GPUHelper.Dispatch1D(compute, kernel, count);
}
```

GPU カーネルの実装は以下の通りです。

▼ SpaceColonization.compute

```
void Remove(uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Attractions.GetDimensions(count, stride);
  if (idx >= count)
    return;

  Attraction attr = _Attractions[idx];
  // 削除済みの Attraction の場合は実行しない
  if (!attr.active)
    return;

  // 全 Node についてループを実行する
  _Nodes.GetDimensions(count, stride);
  for (uint i = 0; i < count; i++)
  {
    Node n = _Nodes[i];
    if (n.active)
    {
      // 削除範囲内の Node があれば、Attraction の active フラグをオフにして削除
      float d = distance(attr.position, n.position);
      if (d < _KillDistance)
      {
        attr.active = false;
        _Attractions[idx] = attr;
        return;
      }
    }
  }
}
```

Grow

Node を成長させます。

Attract カーネルで候補点を生成する際、Node の成長率（t）が閾値以上かどうかを条件に用いていますが（閾値以下の場合は候補点を生成しない）、成長率パラメータはこの Grow カーネルで増分させています。

▼ SpaceColonization.cs

```
protected void Grow(float dt)
{
    var kernel = compute.FindKernel("Grow");
    compute.SetBuffer(kernel, "_Nodes", nodeBuffer);

    var delta = dt * growthSpeed;
    compute.SetFloat("_DT", delta);

    GPUHelper.Dispatch1D(compute, kernel, count);
```

```
}
```

▼ SpaceColonization.compute

```
void Grow (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Nodes.GetDimensions(count, stride);
  if (idx >= count)
    return;

  Node n = _Nodes[idx];

  if (n.active)
  {
    // Node ごとにランダムに設定された mass パラメータで成長速度をバラバラにする
    n.t = saturate(n.t + _DT * n.mass);
    _Nodes[idx] = n;
  }
}
```

1.3.3 レンダリング

以上までの実装でブランチングする形状が得られたので、その形状をどうレンダリングするかについて解説します。

Line Topology によるレンダリング

まずはシンプルに Line Mesh を用いてレンダリングします。

1 本の Edge を表す Line を描画するために、シンプルな Line Topology の Mesh を生成します。

▼ SpaceColonization.cs

```
protected Mesh BuildSegment()
{
    var mesh = new Mesh();
    mesh.hideFlags = HideFlags.DontSave;
    mesh.vertices = new Vector3[2] { Vector3.zero, Vector3.up };
    mesh.uv = new Vector2[2] { new Vector2(0f, 0f), new Vector2(0f, 1f) };
    mesh.SetIndices(new int[2] { 0, 1 }, MeshTopology.Lines, 0);
    return mesh;
}
```

▲図 1.9　シンプルな 2 頂点だけを持つ Line Topology の Mesh

2 つの頂点だけを持つ Segment（線分）を、GPU instancing を用いて Edge の数だけレンダリングすることで生成したブランチを表示します。

▼ SpaceColonization.cs

```
// GPU instancing に必要な、レンダリングする Mesh の数を決めるバッファを生成する
protected void SetupDrawArgumentsBuffers(int count)
{
    if (drawArgs[1] == (uint)count) return;

    drawArgs[0] = segment.GetIndexCount(0);
    drawArgs[1] = (uint)count;

    if (drawBuffer != null) drawBuffer.Dispose();
    drawBuffer = new ComputeBuffer(
        1,
        sizeof(uint) * drawArgs.Length,
        ComputeBufferType.IndirectArguments
    );
    drawBuffer.SetData(drawArgs);
}

...

// GPU instancing によるレンダリングを実行する
protected void Render(float extents = 100f)
{
    block.SetBuffer("_Nodes", nodeBuffer);
    block.SetBuffer("_Edges", edgeBuffer);
    block.SetInt("_EdgesCount", edgesCount);
    block.SetMatrix("_World2Local", transform.worldToLocalMatrix);
    block.SetMatrix("_Local2World", transform.localToWorldMatrix);
    Graphics.DrawMeshInstancedIndirect(
        segment, 0,
```

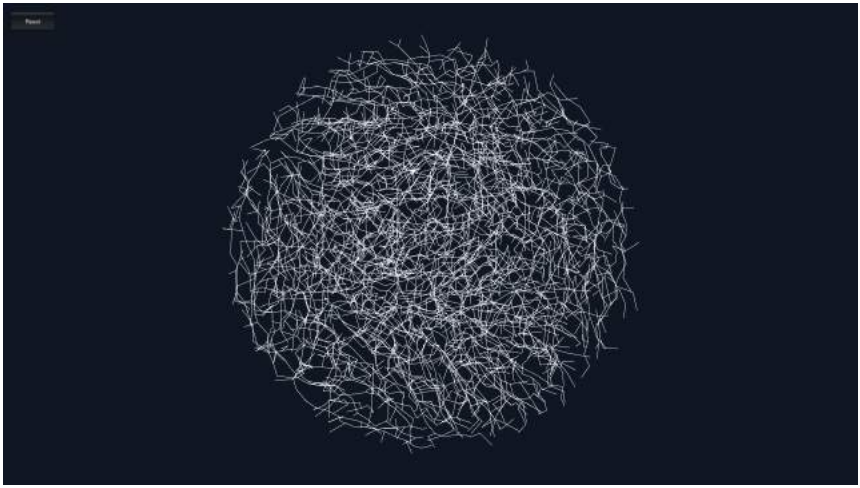
```
        material, new Bounds(Vector3.zero, Vector3.one * extents),
        drawBuffer, 0, block
    );
}
```

レンダリング用のシェーダ（Edge.shader）では、Node の成長率パラメータ（t）に応じて Edge の長さを制御することで、分岐点から伸びていくブランチのアニメーションを生成しています。

▼ Edge.shader

```
v2f vert(appdata IN, uint iid : SV_InstanceID)
{
  v2f OUT;
  UNITY_SETUP_INSTANCE_ID(IN);
  UNITY_TRANSFER_INSTANCE_ID(IN, OUT);

  // インスタンス ID から該当する Edge を取得
  Edge e = _Edges[iid];

  // Edge が持つ index から 2 つの Node を取得
  Node a = _Nodes[e.a];
  Node b = _Nodes[e.b];

  float3 ap = a.position;
  float3 bp = b.position;
  float3 dir = bp - ap;

  // Node b の成長率 (t) に応じて、a から b への Edge の長さを決める
  bp = ap + normalize(dir) * length(dir) * b.t;

  // 頂点 ID(IN.vid) は 0 か 1 なので、0 の場合は a の Node、1 の場合は b の Node の
position を参照する
  float3 position = lerp(ap, bp, IN.vid);

  float4 vertex = float4(position, 1);
  OUT.position = UnityObjectToClipPos(vertex);
  OUT.uv = IN.uv;

  // Node が不活性、またはインスタンス ID が Edge の総数外であれば alpha を 0 にして描画
しない
  OUT.alpha = (a.active && b.active) && (iid < _EdgesCount);

  return OUT;
}
```

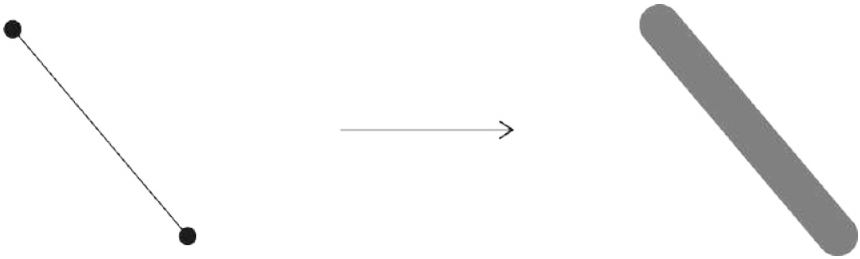
これらの実装で Space Colonization Algorithm で得られた形状を Line Topology を用いてレンダリングすることができます。Line.scene を実行すると以下のような絵を得ることができます。



▲図 1.10　Line.scene - Edge.shader によるレンダリングの例

Geometry Shader によるレンダリング

Line Topology の Segment を Geometry Shader で Capsule 形状に変換することで、厚みのある線を描けるようにします。



▲図 1.11　Line Topology の Segment を GeometryShader で Capsule 形状に変換

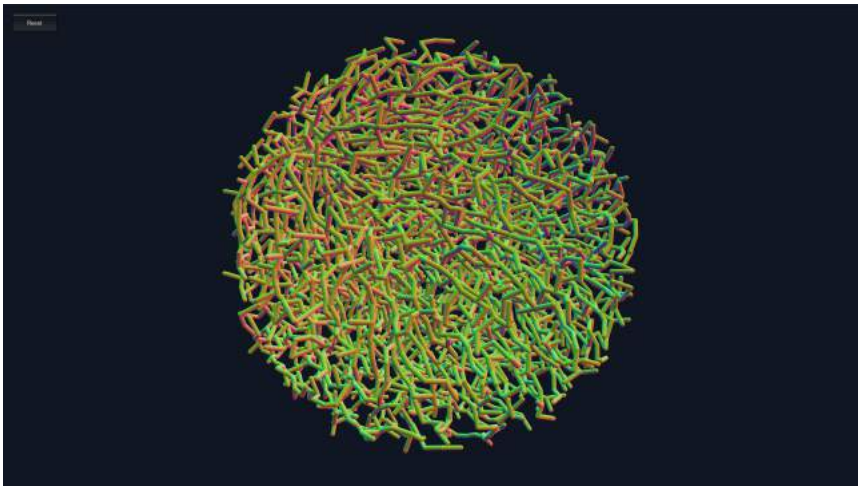
頂点シェーダは Edge.shader とほぼ変わらず、Geometry Shader で Capsule 形状を構築します。以下では重要な Geometry Shader の実装のみ記載します。

▼ TubularEdge.shader

```
...
[maxvertexcount(64)]
void geom(line v2g IN[2], inout TriangleStream<g2f> OUT) {
  v2g p0 = IN[0];
  v2g p1 = IN[1];

  float alpha = p0.alpha;

  float3 t = normalize(p1.position - p0.position);
  float3 n = normalize(p0.viewDir);
  float3 bn = cross(t, n);
  n = cross(t, bn);

  float3 tp = lerp(p0.position, p1.position, alpha);
  float thickness = _Thickness * alpha;

  // Capsule メッシュの解像度の定義
  static const uint rows = 6, cols = 6;
  static const float rows_inv = 1.0 / rows, cols_inv = 1.0 / (cols - 1);

  g2f o0, o1;
  o0.uv = p0.uv; o0.uv2 = p0.uv2;
  o1.uv = p1.uv; o1.uv2 = p1.uv2;

  // Capsule の側面の構築
  for (uint i = 0; i < cols; i++) {
    float r = (i * cols_inv) * UNITY_TWO_PI;

    float s, c;
    sincos(r, s, c);
    float3 normal = normalize(n * c + bn * s);

    float3 w0 = p0.position + normal * thickness;
    float3 w1 = p1.position + normal * thickness;
    o0.normal = o1.normal = normal;

    o0.position = UnityWorldToClipPos(w0);
    OUT.Append(o0);

    o1.position = UnityWorldToClipPos(w1);
    OUT.Append(o1);
  }
  OUT.RestartStrip();

  // Capsule の先端 (半球型) の構築
  uint row, col;
  for (row = 0; row < rows; row++)
  {
    float s0 = sin((row * rows_inv) * UNITY_HALF_PI);
    float s1 = sin(((row + 1) * rows_inv) * UNITY_HALF_PI);
    for (col = 0; col < cols; col++)
    {
      float r = (col * cols_inv) * UNITY_TWO_PI;

      float s, c;
      sincos(r, s, c);

      float3 n0 = normalize(n * c * (1.0 - s0) + bn * s * (1.0 - s0) + t * s0);
      float3 n1 = normalize(n * c * (1.0 - s1) + bn * s * (1.0 - s1) + t * s1);
```

```
        o0.position = UnityWorldToClipPos(float4(tp + n0 * thickness, 1));
        o0.normal = n0;
        OUT.Append(o0);

        o1.position = UnityWorldToClipPos(float4(tp + n1 * thickness, 1));
        o1.normal = n1;
        OUT.Append(o1);
      }
      OUT.RestartStrip();
    }
  }

  ...
```

結果は以下のようなものになります。（TubularEdge.scene）



▲図 1.12　TubularEdge.scene - TubularEdge.shader によるレンダリングの例

これで Edge を厚みのある Mesh でレンダリングが可能になったので、ライティングなどを施すことができるようになります。

1.4　応用

以上までで Space Colonization の GPU 実装を実現できました。本節では応用例として、スキニングアニメーションとの連携について紹介します。

この応用により、アニメーションするモデル形状に沿ってブランチングする表現を実現することができます。

1.4.1 大まかな流れ

スキニングアニメーションとの連携は、以下のような流れで開発します。

1. アニメーションモデルを用意する
2. モデルのボリュームを充填する点群を用意する（Attraction となる点群）
3. Attraction や Node に Bone 情報を持たせる
4. Node に Bone の変形を適用（スキニング）して位置を変化させる

1.4.2 リソースの用意

先の例の構造体

- Attraction
- Node
- Candidate

に Bone の index を持たせるように変更を加えます。

> **影響を受ける Bone の制限について**
>
> 今回の応用では各 Node について影響する Bone の数を 1 つに制限しています。本来であれば、スキニングアニメーションでは各頂点について影響する Bone の数を複数持たせることができるのですが、この例ではシンプルに影響する Bone を 1 つだけに制限しています。

▼ SkinnedAttraction.cs

```
public struct SkinnedAttraction {
    public Vector3 position;
    public int bone; // bone の index
    public int nearest;
    public uint found;
    public uint active;
}
```

▼ SkinnedNode.cs

```
public struct SkinnedNode {
    public Vector3 position;
    public Vector3 animated; // スキニングアニメーション後の Node の position
    public int index0; // bone の index
    public float t;
    public float offset;
    public float mass;
    public int from;
    public uint active;
}
```

▼ SkinnedCandidate.cs

```
public struct SkinnedCandidate
{
    public Vector3 position;
    public int node;
    public int bone; // bone の index
}
```

アニメーションモデルとボリューム

連携させたいアニメーションモデルを用意します。

今回の例では Clara.io[*2]からダウンロードしたモデルを使い（MeshLab[*3]でリダクションをかけてポリゴン数を減らしたもの）、アニメーションは mixamo[*4]で生成したものを利用しています。

モデルのボリュームから Attraction の位置を取得するために、モデルのボリューム内の点群を生成する VolumeSampler[*5]というパッケージを利用します。

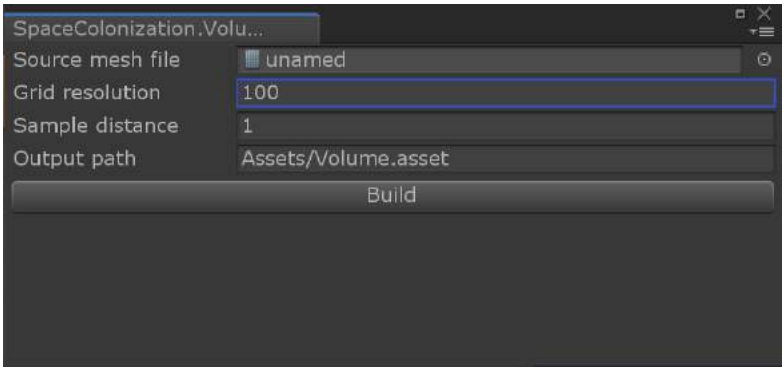
[*2] https://clara.io/view/d49ee603-8e6c-4720-bd20-9e3d7b13978a
[*3] http://www.meshlab.net/
[*4] https://mixamo.com
[*5] https://github.com/mattatz/unity-volume-sampler

VolumeSampler

VolumeSampler では Unity Graphics Programming vol.2「Real-Time GPU-Based Voxelizer」で解説しているテクニックを利用してモデルのボリュームを取得しています。まず、メッシュ内部のボリュームを Voxel として取得し、それを元に Poisson Disk Sampling を実行することで、メッシュ内部を充填するような点群を生成しています。
VolumeSampler を使って点群アセットを生成する方法は、Unity ツールバー内から Window → VolumeSampler をクリックして Window を表示し、以下の図のように、

- 対象の Mesh への参照
- 点群のサンプリング解像度

を設定し、アセット生成ボタンをクリックすると、指定したパスに Volume アセットが生成されます。



▲図 1.13　VolumeSamplerWindow

VolumeSampler から生成された点群アセット（Volume クラス）から SkinnedAttraction 配列を生成し、ComputeBuffer に適用します。

▼ SkinnedSpaceColonization.cs

```
protected void Start() {
    ...
    // Volume が持つ点群から SkinnedAttraction 配列を生成する
```

```
    attractions = GenerateAttractions(volume);
    count = attractions.Length;
    attractionBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(SkinnedAttraction)),
        ComputeBufferType.Default
    );
    attractionBuffer.SetData(attractions);
    ...
}
```

Bone 情報の適用

Mesh のボリュームから生成された SkinnedAttraction には、各位置から最も近い頂点の Bone 情報を適用します。

SetupSkin 関数では Mesh の頂点と Bone バッファを用意し、GPU 上ですべての SkinnedAttraction に Bone の index を割り振ります。

▼ SkinnedSpaceColonization.cs

```
protected void Start() {
    ...
    SetupSkin();
    ...
}

...

protected void SetupSkin()
{
    var mesh = skinnedRenderer.sharedMesh;
    var vertices = mesh.vertices;
    var weights = mesh.boneWeights;
    var indices = new int[weights.Length];
    for(int i = 0, n = weights.Length; i < n; i++)
        indices[i] = weights[i].boneIndex0;

    using (
        ComputeBuffer
        vertBuffer = new ComputeBuffer(
            vertices.Length,
            Marshal.SizeOf(typeof(Vector3))
        ),
        boneBuffer = new ComputeBuffer(
            weights.Length,
            Marshal.SizeOf(typeof(uint))
        )
    )
    {
        vertBuffer.SetData(vertices);
        boneBuffer.SetData(indices);

        var kernel = compute.FindKernel("SetupSkin");
```

```
        compute.SetBuffer(kernel, "_Vertices", vertBuffer);
        compute.SetBuffer(kernel, "_Bones", boneBuffer);
        compute.SetBuffer(kernel, "_Attractions", attractionBuffer);
        GPUHelper.Dispatch1D(compute, kernel, attractionBuffer.count);
    }
}
```

以下が GPU カーネルの実装になります。

▼ SkinnedSpaceColonization.compute

```
void SetupSkin (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Attractions.GetDimensions(count, stride);
  if (idx >= count)
    return;

  SkinnedAttraction attr = _Attractions[idx];

  // SkinnedAttraction の位置から最も近い頂点の index を取得する
  float3 p = attr.position;
  uint closest = -1;
  float dist = 1e8;
  _Vertices.GetDimensions(count, stride);
  for (uint i = 0; i < count; i++)
  {
    float3 v = _Vertices[i];
    float l = distance(v, p);
    if (l < dist)
    {
      dist = l;
      closest = i;
    }
  }

  // 最も近い頂点の Bone index を SkinnedAttraction に設定する
  attr.bone = _Bones[closest];
  _Attractions[idx] = attr;
}
```

1.4.3 ComputeShader でのアルゴリズムの実装

この応用例では、Space Colonization Algorithm における各ステップの中でスキニングアニメーションに必要な Bone 情報を取得するため、いくつかの GPU カーネルに修正を加えます。

概ね GPU カーネルの中身は同じなのですが、生成される SkinnedNode について、最も近傍の SkinnedAttraction から Bone 情報（Bone の index）を得る必要があるため、

- Seed
- Attract

の 2 つの GPU カーネルでは、近傍探索のロジックが追加されています。

▼ SkinnedSpaceColonization.compute

```
void Seed (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;

  uint count, stride;
  _Seeds.GetDimensions(count, stride);
  if (idx >= count)
    return;

  SkinnedNode n;
  uint i = CreateNode(n);
  n.position = n.animated = _Seeds[idx];
  n.t = 1;
  n.offset = 0;
  n.from = -1;
  n.mass = lerp(_MassMin, _MassMax, nrand(id.xy));

  // 最近傍の SkinnedAttraction を探索し、
  // Bone index をコピーする
  uint nearest = -1;
  float dist = 1e8;
  _Attractions.GetDimensions(count, stride);
  for (uint j = 0; j < count; j++)
  {
    SkinnedAttraction attr = _Attractions[j];
    float l = distance(attr.position, n.position);
    if (l < dist)
    {
      nearest = j;
      dist = l;
    }
  }
  n.index0 = _Attractions[nearest].bone;

  _Nodes[i] = n;
}

...

void Attract (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Nodes.GetDimensions(count, stride);
  if (idx >= count)
    return;

  SkinnedNode n = _Nodes[idx];

  if (n.active && n.t >= 1.0)
```

```
  {
    float3 dir = (0.0).xxx;
    uint counter = 0;

    float dist = 1e8;
    uint nearest = -1;

    _Attractions.GetDimensions(count, stride);
    for (uint i = 0; i < count; i++)
    {
      SkinnedAttraction attr = _Attractions[i];
      if (attr.active && attr.found && attr.nearest == idx)
      {
        float3 dir2 = (attr.position - n.position);
        dir += normalize(dir2);
        counter++;

        // 最近傍の SkinnedAttraction を探索する
        float l2 = length(dir2);
        if (l2 < dist)
        {
          dist = l2;
          nearest = i;
        }
      }
    }

    if (counter > 0)
    {
      SkinnedCandidate c;
      dir = dir / counter;
      c.position = n.position + (dir * _GrowthDistance);
      c.node = idx;
      // 最近傍の SkinnedAttraction の持つ Bone index を設定する
      c.bone = _Attractions[nearest].bone;
      _CandidatesAppend.Append(c);
    }
  }
}
```

スキニングアニメーション

以上までの実装で、Node に対して Bone 情報を設定しながら Space Colonization Algorithm を実行できるようになりました。

あとは必要な Bone 行列を SkinnedMeshRenderer から取得し、GPU 上で SkinnedNode の位置を Bone の変形に応じて動かすことで、Node に対してスキニングアニメーションを施すことができます。

▼ SkinnedSpaceColonization.cs

```
protected void Start() {
    ...
```

```
    // bind pose 行列のバッファを作成しておく
    var bindposes = skinnedRenderer.sharedMesh.bindposes;
    bindPoseBuffer = new ComputeBuffer(
        bindposes.Length,
        Marshal.SizeOf(typeof(Matrix4x4))
    );
    bindPoseBuffer.SetData(bindposes);
    ...
}

protected void Animate()
{
    // アニメーションが再生されることで更新される、SkinnedMeshRenderer の Bone 行列
を表すバッファを作成する
    var bones = skinnedRenderer.bones.Select(bone => {
        return bone.localToWorldMatrix;
    }).ToArray();
    using (
        ComputeBuffer boneMatrixBuffer = new ComputeBuffer(
            bones.Length,
            Marshal.SizeOf(typeof(Matrix4x4))
        )
    )
    {
        boneMatrixBuffer.SetData(bones);

        // Bone と Node のバッファを渡し、GPU スキニングを実行する
        var kernel = compute.FindKernel("Animate");
        compute.SetBuffer(kernel, "_BindPoses", bindPoseBuffer);
        compute.SetBuffer(kernel, "_BoneMatrices", boneMatrixBuffer);
        compute.SetBuffer(kernel, "_Nodes", nodeBuffer);
        GPUHelper.Dispatch1D(compute, kernel, count);
    }
}
```

▼ SkinnedSpaceColonization.compute

```
void Animate (uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, stride;
  _Nodes.GetDimensions(count, stride);
  if (idx >= count)
    return;

  SkinnedNode node = _Nodes[idx];
  if (node.active)
  {
    // スキニングの実行
    float4x4 bind = _BindPoses[node.index0];
    float4x4 m = _BoneMatrices[node.index0];
    node.animated = mul(mul(m, bind), float4(node.position, 1)).xyz;
    _Nodes[idx] = node;
  }
}
```

1.4.4 レンダリング

レンダリング用のシェーダはほぼ同じなのですが、SkinnedNode の元の位置（position）ではなく、スキニングアニメーションを施した後の位置（animated）を参照して Edge を描画する点だけが異なります。

▼ SkinnedTubularEdge.hlsl

```
v2g vert(appdata IN, uint iid : SV_InstanceID)
{
  ...
  Edge e = _Edges[iid];

  // スキニングアニメーションを施した後の位置を参照する
  SkinnedNode a = _Nodes[e.a], b = _Nodes[e.b];
  float3 ap = a.animated, bp = b.animated;

  float3 dir = bp - ap;
  bp = ap + normalize(dir) * length(dir) * b.t;
  float3 position = lerp(ap, bp, IN.vid);
  OUT.position = mul(unity_ObjectToWorld, float4(position, 1)).xyz;
  ...
}
```

以上までの実装で冒頭に示したキャプチャの絵を得ることができます。（図 1.1 SkinnedAnimation.scene）

1.5 まとめ

本章では、点群に沿ってブランチング形状を生成する Space Colonization Algorithm の GPU 実装と、スキニングアニメーションと組み合わせる応用例を紹介しました。

この手法では、

- Attraction の影響範囲（influence distance）
- Node が伸びる長さ（growth distance）
- Attraction を削除する範囲（kill distance）

という 3 つのパラメータで枝の粗密具合をコントロールできますが、さらにこれらのパラメータを局所的に変えたり、時間によって変化させたりすることで、より多彩なモデルを生成することができます。

また、サンプルでは Attraction は初期化の際に生成したものだけでアルゴリズムを実行していますが、Attraction を動的に増やすことでもより多くの異なるパターンを生成することができるはずです。

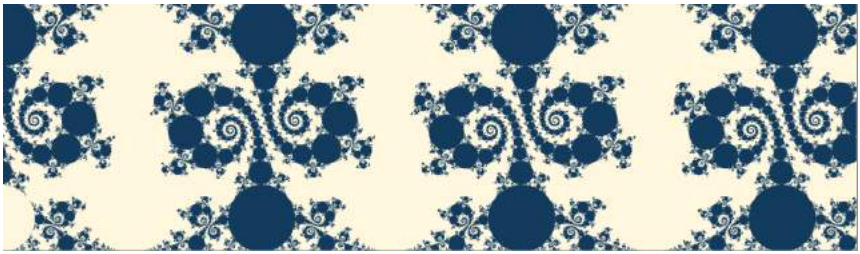
興味のある方は本アルゴリズムの様々な応用を試して面白いパターンを探してみてください。

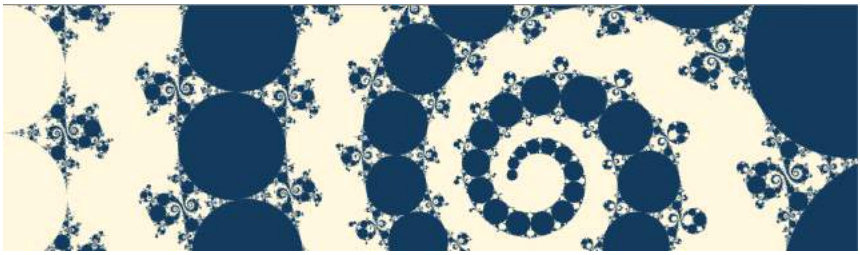
1.6 参考

- Modeling Trees with a Space Colonization Algorithm - http://algorithmicbotany.org/papers/colonization.egwnp2007.large.pdf
- Algorithmic Design with Houdini - https://vimeo.com/305061631#t=1500s

第 2 章

Limit sets of Kleinian groups





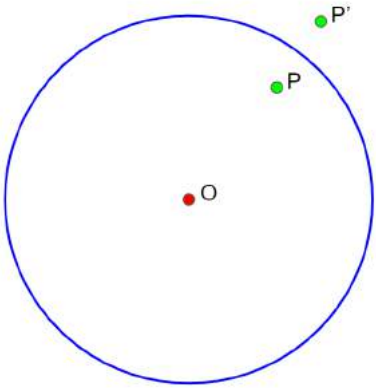
本章ではクライン群の極限集合をシェーダーで描き、出来上がったフラクタル図形をアニメーションさせる方法を紹介します。フラクタルのアニメーションといえば拡大縮小することで自己相似形を眺める面白みがありますが、こちらの手法ではさらに直線と円周がなめらかに遷移する特徴的な動きが見られます。

本章のサンプルは

https://github.com/IndieVisualLab/UnityGraphicsProgramming4
の「KleinianGroup」になります。

2.1 円反転

まず図形の反転について紹介します。直線を境に鏡写しのように反転した図形は線対称、点を中心に反転した場合は点対称、というのはおなじみだと思います。さらに円に関しての反転というものが存在します。２次元平面を円の内部と外部を互いに入れ替える操作になります。

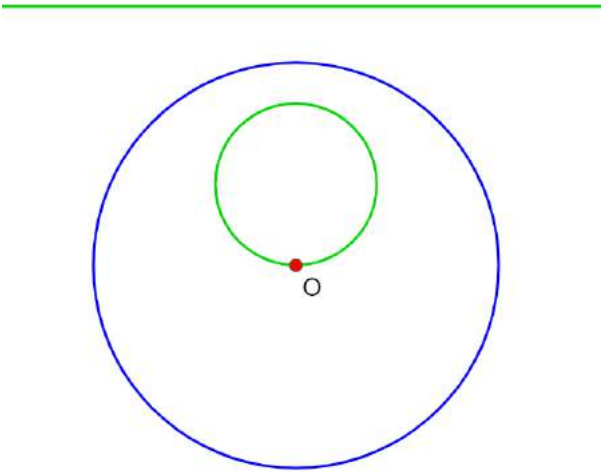


▲図 2.1　円反転 $P \rightarrow P'$

円の中心 O , 半径 r の円に関しての反転は、 $|OP|$ と $|OP'|$ が同じ向きのまま次の式を満たすように P を P' へ移す操作になります。

$$|OP| \cdot |OP'| = r^2$$

円周付近では内側と外側が歪んだ線対称のように入れ替わっているように見え、円周から遥かに離れた無限遠と円の中心が入れ替わる形になります。面白いのは円の外側の直線を円反転した場合で、円に近い付近では円周を挟んだ内側に移りそこから離れていくと無限遠まで続く、つまり反転後は円の中心に繋がる形になります。



▲図 2.2　直線の円反転

これは円の内側に小さい円として現れます。直線も半径無限の円と捉えると、円反転は円の内側と外側の円同士を入れ替える操作と言えます。

2.1.1　数式で表す

複素平面上の単位円の円反転を式で表すと次のようになります。

$$z \to \frac{1}{\overline{z}}$$

z は複素数で $\overline{z}$ はその複素共益です。

次のように式変形してみると z の長さの 2 乗分の 1 で z をスケールしている操作であることがわかります。

$$z \to \frac{1}{\overline{z}} = \frac{1}{x - iy} = \frac{x + iy}{(x - iy)(x + iy)} = \frac{x + iy}{x^2 + y^2} = \frac{z}{|z|^2}$$

複素平面での図形操作として、

- 複素数の和：平行移動
- 複素数の積：回転（と拡大縮小）

と認識されている方も多いかと思いますが、ここに新しく割り算を含む操作が入ってきた形になります。

2.2 メビウス変換

複素平面における変換に割り算まで含めて一般化した形がメビウス変換[*1]です。

$$z \to \frac{az+b}{cz+d}$$

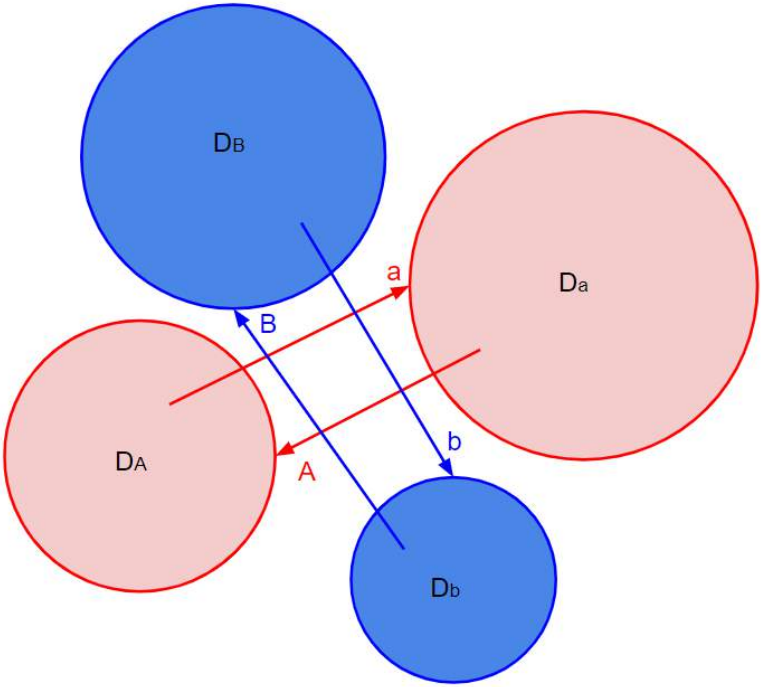
a, b, c, d はすべて複素数です。

2.3 ショットキー群、クライン群

メビウス変換を繰り返し用いてフラクタル図形を作ることを考えます。

互いに交わらない四組の円 D_A , D_a , D_B , D_b を用意します。まずこのうち二組の円に注目し D_A の外部を D_a 内部に、 D_A の内部を D_a 外部に移すメビウス変換 a を作ります。同様に別の二組の円 D_B 、 D_b からメビウス変換 b を作ります。またそれぞれの逆変換 A , B も用意します。

[*1] メビウスの帯でおなじみに数学者アウグスト・フェルディナント・メビウスに由来します。



こうしてできた 4 つのメビウス変換 a , A , b , B をどのような順番であれ合成した変換全体（たとえば $aaBAbbaB$ ）を、「 a , b を元にしたショットキー群[*2]」と呼びます。

これをさらに一般化しメビウス変換からなる離散群のことをクライン群と呼びます。どうもこちらの呼び方のほうが広く使われている印象です。

2.4 極限集合

ショットキー群の像を表示していくと、円の内部に円がありその内部にも円があり、と無限に続く円が現れます。これらの集合を「 a , b におけるショットキー群の極限集合」と呼びます。この極限集合を描くことが本章の目的になっています。

[*2] このような群を最初に考案した数学者フリードリッヒ・ヘルマン・ショットキーに由来します。

2.5 Jos Leys 氏のアルゴリズム

2.5.1 概要

シェーダーで極限集合を描画する方法について紹介していきます。変換の組み合わせが無限に続いてしまうので愚直に実装すると大変なのですが、Jos Lays 氏がこのためのアルゴリズムを公開している[*3]のでこれに沿ってやってみます。

まず 2 つのメビウス変換を用意します。

$$a : z \to \frac{tz - i}{-iz}$$

$$b : z \to z + 2$$

t は複素数 $u + iv$ です。この値をパラメータとして変化させることで図形の形状を変えることができます。

変換 a を詳しく見ていくと、

$$a : z \to \frac{tz - i}{-iz} = \frac{t}{-i} + \frac{1}{z} = \frac{1}{z} + (-v + iu)$$

$$\frac{1}{z} = \frac{1}{x + iy} = \frac{x - iy}{(x + iy)(x - iy)} = \frac{x - iy}{x^2 + y^2} = \frac{x - iy}{|z|^2}$$

したがって、

$$a : z \to \frac{tz - i}{-iz} = \frac{x - iy}{|z|^2} + (-v + iu)$$

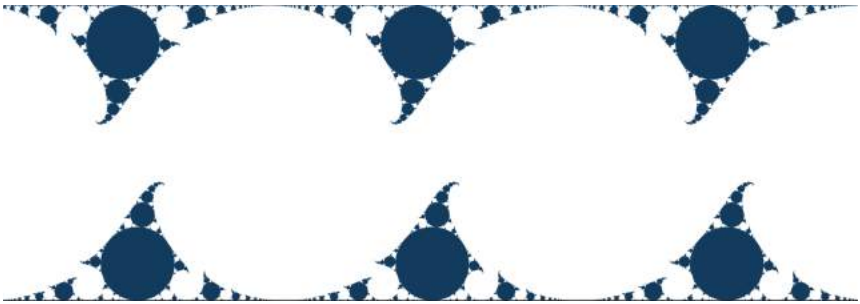
なので、

1. 単位円における円反転し、
2. y の符号を反転し、
3. $-v + iu$ 平行移動する

という操作になっていることがわかります。

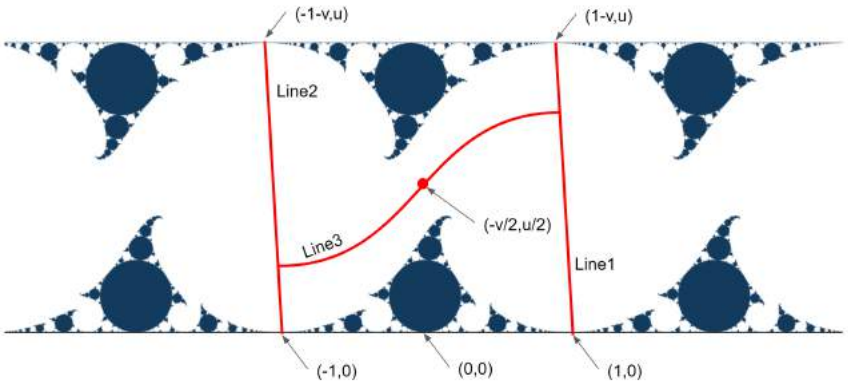
変換 a,b とその逆変換を用いた極限集合は大小の円がつながった次のような帯状の図形になります。

[*3] http://www.josleys.com/articles/Kleinian%20escape-time_3.pdf



▲図 2.3　極限集合

この図形の特徴について詳しく見ていきます。



$0 \leq y \leq u$ の帯状になっており、左右方向には Line1,2 で区切られた平行四辺形が繰り返している形になっています。Line1 は点 $(1,0)$, 点 $(1-v,u)$ を通る直線で、Line2 は点 $(-1,0)$, 点 $(-1-v,u)$ を通る直線です。Line3 は図形を上限に分ける線になり平行四辺形の中においてこの線で分けられた上下の図形は点 $z=-\dfrac{v}{2}+\dfrac{iu}{2}$ において点対称になっています。

2.5.2　アルゴリズム

任意の点が極限集合に含まれるかどうかを判定します。左右には平行四辺形領域が繰り返している、上下には Line3 を境界に点対称になっていることを利用し各点にお

ける判定を最終的には中心下半分の図形の判定に持っていきます。
ある点について次のように処理していきます。

- もし Line1 よりも右にあるなら平行四辺形範囲に収まるように左に移動
- もし Line2 よりも左にあるなら同様に右に移動
- Line3 よりも上にあるなら点対称であることを利用し Line3 の下になるように対象点に移動
- Line3 よりも下にある点には変換 a を適用する

直線 $y = 0$ に接している一番大きい円は、直線 $y = u$ を単位円で反転したものになります。この中の点に変換 a をかけると $y < 0$ となり $0 \leq y \leq u$ の帯から外れます。したがって、

> ある点に変換 a をかけたら $y < 0$ となった = ある点はこの一番大きな円に含まれる
> = 極限集合に含まれる

として判定を行います。
逆に含まれない場合はどうでしょうか。上記手順を繰り返しても $0 \leq y \leq u$ の帯から出られず、最終的には Line3 を挟んだ 2 点の移動を繰り返す形になります。したがって 2 つ前と同じ点だった場合は極限集合には含まれない点であると判定できます。
まとめると次のように処理していく形になります。

1. $y < 0$, $u < y$ であれば範囲外
2. Line1 より右、Line2 より左にあれば中心の平行四辺形へ移動
3. Line3 より上にあれば中心点について反転
4. 変換 a を適用
5. もし $y < 0$ なら極限集合と判定
6. 2 つ前の点と同じなら極限集合ではないと判定
7. どちらでもなければ 2. に戻る

2.6 実装

それではコードを見ていきます。

▼ KleinianGroup.cs

```
private void OnRenderImage(RenderTexture source, RenderTexture destination)
{
    material.SetColor("_HitColor", hitColor);
    material.SetColor("_BackColor", backColor);
    material.SetInt("_Iteration", iteration);
    material.SetFloat("_Scale", scale);
    material.SetVector("_Offset", offset);
    material.SetVector("_Circle", circle);

    Vector2 uv = kleinUV;
    if ( useUVAnimation)
    {
        uv = new Vector2(
            animKleinU.Evaluate(time),
            animKleinV.Evaluate(time)
            );
    }
    material.SetVector("_KleinUV", uv);
    Graphics.Blit(source, destination, material, pass);
}
```

C#側は KleinianGroup.cs でインスペクタで設定したパラメータを渡しつつマテリアルを描画する `OnRenderImage()`のみの処理になっています。

シェーダーを見てきましょう。

▼ KleinianGroup.shader

```
#pragma vertex vert_img
```

Vertex シェーダーは Unity 標準の vert_img を使っています。メインは Fragment シェーダーです。Fragment シェーダーは 3 つ用意してありそれぞれ別のパスになっています。1 つ目が標準的なもの、2 つ目がぼかし処理が入り少し見た目がきれいになったもの、3 つ目が後述のさらに円反転を加えたものになっています。KleinianGroup.cs でどのパスを使うか選択できるようになっています。ここでは 1 つ目のものを見ていきます。

▼ KleinianGroup.shader

```
fixed4 frag (v2f_img i) : SV_Target
{
    float2 pos = i.uv;
    float aspect = _ScreenParams.x / _ScreenParams.y;
    pos.x *= aspect;
    pos += _Offset;
    pos *= _Scale;

    bool hit = josKleinian(pos, _KleinUV);
    return hit ? _HitColor : _BackColor;
}
```

_ScreenParamsからアスペクト比を求めて pos.x に乗算しています。これで pos で表される画面上の領域は 0 ≦ y ≦ 1 で、x はアスペクト比に応じた範囲になります。さらに C#側から渡される_Offset,_Scaleを適用することで表示する位置と範囲を調整できるようにしています。josKleinian()で極限集合の可否判定を行い出力する色を決定しています。

josKleinian()を詳しく見ていきます。

▼ KleinianGroup.shader

```
bool josKleinian(float2 z, float2 t)
{
    float u = t.x;
    float v = t.y;

    float2 lz=z+(1).xx;
    float2 llz=z+(-1).xx;

    for (uint i = 0; i < _Iteration ; i++)
    {
        ～
```

点 z とメビウス変換のパラメータ t を受け取って、z が極限集合に含まれるかどうか判定する関数です。lz,llz は、集合の外部であることを示す「2 つ前と同じ点」判定用の変数です。とりあえず開始時の z と異なりまたお互いも異なるように値を初期化しています。_Iterationは手順を繰り返す最大回数です。細部を拡大して見ていくのでなければそれほど大きな値でなくても十分だと思います。

▼ KleinianGroup.shader

```
// wrap if outside of Line1,2
float offset_x = abs(v) / u * z.y;
z.x += offset_x;
z.x = wrap(z.x, 2, -1);
z.x -= offset_x;
```

▼ KleinianGroup.shader

```
float wrap(float x, float width, float left_side){
    x -= left_side;
    return (x - width * floor(x/width)) + left_side;
}
```

ここが

```
Line1 より右、Line2 より左にあれば中心の平行四辺形へ移動
```

の部分になります。

wrap()は点の位置、長方形の横幅、長方形の左端の座標を受け取り、左右にはみ出している点を収める関数です。offset_x で平行四辺形を長方形に変換し wrap()で範囲内に収め、再度 offset_x で平行四辺形に戻す処理になっています。

次に Line3 の判定です。

▼ KleinianGroup.shader

```
//if above Line3, inverse at (-v/2, u/2)
float separate_line = u * 0.5
    + sign(v) *(2 * u - 1.95) / 4 * sign(z.x + v * 0.5)
    * (1 - exp(-(7.2 - (1.95 - u) * 15)* abs(z.x + v * 0.5)));

if  (z.y >= separate_line)
{
    z = float2(-v, u) - z;
}
```

separate_lineを求めている部分が Line3 の条件式になります。この部分は導出がよくわからず図形の対称性からおおよそで求めているものかなと思います。複雑な図形になるように突き詰めた t の値によっては上下の図形がギザギザに噛み合うことがありちゃんと分割するにはこの条件式では不十分なこともありますが、おおよそ一般的な形では有効なので今回はこのまま使用します。

▼ KleinianGroup.shader

```
z = TransA(z, t);
```

▼ KleinianGroup.shader

```
float2 TransA(float2 z, float2 t){
    return float2(z.x, -z.y) / dot(z,z) + float2(-t.y, t.x);
}
```

いよいよ点 z にメビウス変換 a を適用します。コーディングしやすいように前述の式変形を利用して、

$$a : z \to \frac{tz - i}{-iz} = \frac{x - iy}{|z|^2} + (-v + iu)$$

これを実装しています。

▼ KleinianGroup.shader

```
//hit!
if (z.y<0) { return true; }
```

変換の結果、 $y < 0$ なら極限集合判定、

▼ KleinianGroup.shader

```
//2cycle
if(length(z-llz) < 1e-6) {break;}

llz=lz;
lz=z;
```

２つ前の点とほぼ同値なら極限集合ではないと判定します。また、_Iteration繰り返しても判定結果が出ない場合も極限集合ではないと判定しています。

以上でシェーダーの実装は完了です。パラメータ t は $(2, 0)$ が最も標準的な値で $(1.94, 0.02)$ 付近で面白い形になりやすいです。サンプルプロジェクトではKleinianGroupDemo.cs のインスペクタ上で編集できるのでぜひ遊んでみてください。

2.7 さらに円反転

極限集合の表示は以上なのですがアニメーションとして面白くするために最後に強力なトッピングを加えます。josKleinian()にわたす前にポジションを円反転します。円反転は、円の外側の無限に広がる領域と内側を入れかえ、しかも円は円として移されるのでした。そして極限集合は無数の円で構成されています。この反転円を動かしたり半径を変えたりすることで、フラクタルの面白さを活かしつつ予想もできないような不思議な見た目が出来上がります。

▼ KleinianGroup.shader

```
float4 calc_color(float2 pos)
{
    bool hit = josKleinian(pos, _KleinUV);
    return hit ? _HitColor : _BackColor;
}

~
float4 _Circle;

float2 circleInverse(float2 pos, float2 center, float radius)
{
    float2 p = pos - center;
    p = (p * radius) / dot(p,p);
    p += center;
    return p;
}
```

```
fixed4 frag_circle_inverse(v2f_img i) : SV_Target
{
    float2 pos = i.uv;
    float aspect = _ScreenParams.x / _ScreenParams.y;
    pos.x *= aspect;
    pos *= _Scale;
    pos += _Offset;

    int sample_num = 10;
    float4 sum;
    for (int i = 0; i < sample_num; ++i)
    {
        float2 offset = rand2n(pos, i) * (1/_ScreenParams.y) * 3;
        float2 p = circleInverse(pos + offset, _Circle.xy, _Circle.w);
        sum += calc_color(p);
    }

    return sum / sample_num;
}
```

これは 3 つ目のパスで定義されている円反転を加えた Fragment シェーダーです。`sample_num`のループは周辺のピクセルも計算して少しぼかすことで見た目を綺麗にする処理になります。`calc_color()`がいままでの色を計算する処理になりその前に`cieclelnverse()`で円反転しています。

KleinianGroupCircleInverse シーンではこちらのシェーダーのパラメータを Animator で変化させることでフラクタルらしいアニメーションが動作するようになっています。

2.8 まとめ

本章ではクライン群の極限集合をシェーダーで描く方法とさらに円反転の用いることでより面白いフラクタル図形の作り方を紹介しました。フラクタルやメビウス変換は普段馴染みのない分野でとっつきにくい部分もあったのですが次々と予想もしないような模様が動いてく様はとても刺激的でした。よかったらみなさんもチャレンジしてみてください！

2.9 参考

- インドラの真珠[*4]
  最強のバイブルです。ちょっと値が張り、内容も高度なので以下のところを見てより深く理解したくなったら手に取るのがおすすめです。

[*4] https://www.amazon.co.jp/dp/4535783616

- Mathe Vital [*5]
  インドラの真珠のエッセンスを抜き出してわかりやすく紹介されています。まずここから見ていくと良いかと思います。
- 今回紹介した Jos Leys 氏のアルゴリズム[*6]
- Jos Leys 氏の Shadertoy [*7]
  貴重なシェーダー実装の例です
- Morph [*8]
  @soma_arc さん[*9]が TokyoDemoFest2018[*10]で発表されていた作品です。今回私自身クライン群を学ぶきっかけになりました。こちらも貴重なシェーダー実装例。

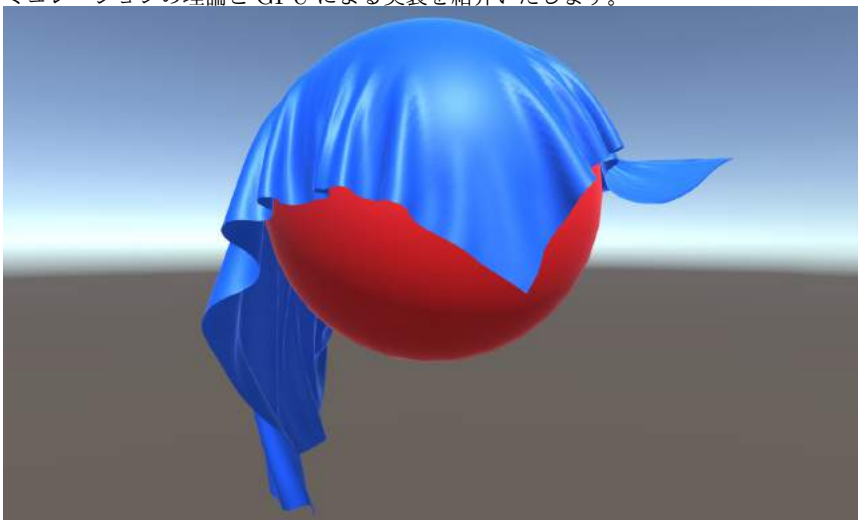
[*5] http://userweb.pep.ne.jp/hannyalab/MatheVital/IndrasPearls/IndrasPearlsindex.html
[*6] http://www.josleys.com/articles/Kleinian%20escape-time_3.pdf
[*7] https://www.shadertoy.com/user/JosLeys
[*8] https://www.shadertoy.com/view/MlGfDG
[*9] https://twitter.com/soma_arc
[*10] http://tokyodemofest.jp/2018/

第 3 章

GPU-Based Cloth Simulation

3.1　はじめに

旗・衣服のような外からの力によって変形する平面状のオブジェクトの形状のシミュレーションは布シミュレーション（**Cloth Simulation**）と呼ばれ、CG 分野ではアニメーション作成に必須のものとして、多くの研究がなされています。Unity にもすでに実装されていますが、GPU を活用した並列計算の学習、シミュレーションの性質やパラメータの意味を理解することを目的として、この章では、簡単な布シミュレーションの理論と GPU による実装を紹介いたします。

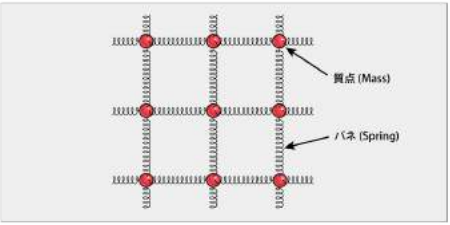


▲図 3.1　布シミュレーション

3.2 アルゴリズム解説

3.2.1 質点-バネ系（Mass-Spring System）

バネやゴム、クッションのように、力をかけると変形し、力をかけるのをやめると、元の形状に戻るような物体を弾性体（**Elastics Body**）と言います。このような弾性体は、1 つの位置や姿勢で表すことができないため、物体を点とその間の接続により表し、それぞれの点の動きで形状全体をシミュレートします。この点は、質点（**Mass**）と言い、大きさのない質量の塊であると考えます。また、質点同士の接続はバネ（Spring）の性質を持ちます。各バネの伸び縮みを計算することで弾性体をシミュレートする手法を、質点-バネ系（Mass-Spring System）と言い、2 次元状に並んだ質点の集合の運動を計算することによる旗や衣服などのシミュレーションは、布シミュレーション（Cloth Simulation）と呼ばれます。



▲図 3.2　質点-バネ系

3.2.2 布シミュレーションに使用するアルゴリズム

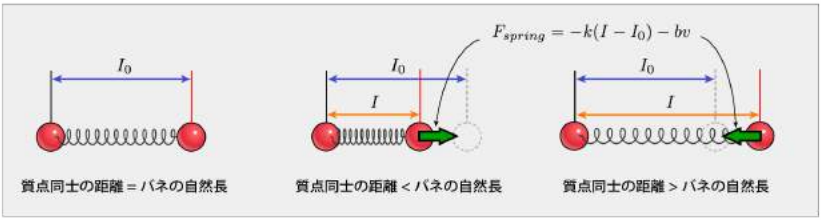
バネの力

それぞれのバネは、接続された質点に対して、以下の式に従った力を与えます。

$$F_{spring} = -k\,(I - I_0) - bv$$

ここで、I は、バネの現在の長さ（接続された質点同士の距離）、I_0 は、シミュレーション開始時のバネの自然長（バネに何も荷重がかかっていない時の長さ）を表します。k は、バネの硬さを表す定数で、v は質点の速度、b は速度の減衰度合いを決定する定数です。この方程式は、バネは接続された質点同士の距離がバネの初期の自然長に戻ろうとする力が常に働くということを意味します。バネの現在の距離が、

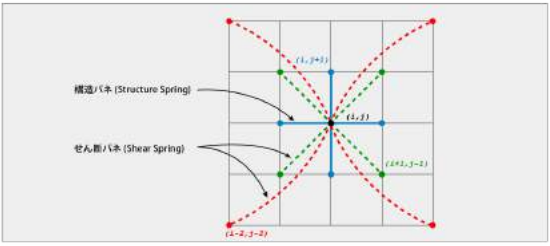
バネの自然長と大きく異なっていればそれだけ大きな力が働き、質点の現在の速度に比例して減衰します。



▲図 3.3　バネの力

バネの構造

このシミュレーションでは、横・縦方向に、基本構造を作るバネを接続し、斜め方向の極端なズレを防ぐため、対角線状に位置する質点間にもバネを接続します。それぞれ、構造バネ（Structure Spring）、せん断バネ（Shear Spring）と呼び、1 つの質点は近隣の 12 個の質点に対してバネを接続します。



▲図 3.4　バネの構造

ベレ法（Verlet Method）による位置計算

このシミュレーションでは、ベレ法というリアルタイムアプリケーションによく用いられる手法で質点の位置の計算を行います。ベレ法は、ニュートンの運動方程式の数値解法の一つであり、分子動力学において原子の動きなどを計算する時に用いられる手法です。物体の運動を計算する際、通常は速度から位置を求めますが、ベレ法で

は、現在の位置と、前のタイムステップでの位置から次のタイムステップにおける位置を求めます。

以下、ベレ法の代数方程式の導出を示します。F は質点に適用される力、 m は質点の質量、 v は速度、 x は位置、 t は時刻、 Δt はシミュレーションのタイムステップ（シミュレーション 1 回の計算につき、どれだけの時間を進めるのか）とすると、質点の運動方程式は、

$$m\frac{d^2x\left(t\right)}{dt^2} = F$$

と書けます。この運動方程式を、次の 2 つのテイラー展開を用いて、代数方程式にすると、

$$x\left(t+\Delta t\right) = x\left(t\right) + \Delta t\frac{dx\left(t\right)}{dt} + \frac{1}{2!}\Delta t^2\frac{dx^2\left(t\right)}{dt^2} + \frac{1}{3!}\Delta t^3\frac{dx^3\left(t\right)}{dt^3} + \ldots$$

$$x\left(t-\Delta t\right) = x\left(t\right) - \Delta t\frac{dx\left(t\right)}{dt} + \frac{1}{2!}\Delta t^2\frac{dx^2\left(t\right)}{dt^2} - \frac{1}{3!}\Delta t^3\frac{dx^3\left(t\right)}{dt^3} + \ldots$$

となります。この 2 つのテイラー展開式から、2 階の微分項について解き、 Δt の 2 次以上の項を十分に小さいものとして無視すると、次のように書けます。

$$\frac{dx^2\left(t\right)}{dt^2} = \frac{x\left(t+\Delta t\right) - 2x\left(t\right) + x\left(t-\Delta t\right)}{\Delta t^2}$$

2 階の微分項を、運動方程式より質量 m と力 F で表して整理すると、

$$x\left(t+\Delta t\right) = 2x\left(t\right) - x\left(t-\Delta t\right) + \frac{\Delta t^2}{m}F\left(t\right)$$

という代数方程式が得られます。このように 1 つ後のタイムステップにおける位置を、現在の位置、一つ前のタイムステップにおける位置、質量、力、タイムステップの値から求める式が得られます。

速度については、現在の位置と、時間的に一つ前の位置から求めます。

$$v\left(t\right) = \frac{x\left(t\right) - x\left(t-\Delta t\right)}{\Delta t}$$

この計算によって求められる速度は、精度が高いとは言えませんが、バネの減衰の計算に用いるのみであるので、問題ではないと言えます。

衝突の計算

■布と球の衝突の計算　衝突の処理は、「衝突の検出」と「それに対する反応」との二つのフェーズで行います。

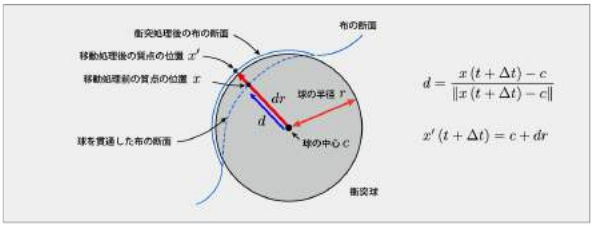
衝突の検出は以下の式で行います。

$$\|x\left(t+\Delta t\right)-c\|-r<0$$

c と r は、球の中心と半径、$x\left(t+\Delta t\right)$ は、ベレ法によって求められた次のタイムステップにおける位置です。もし衝突が検出されたら、質点を球の表面上に移動させることによって、球と布が衝突しない状態にします。詳細には、衝突点の表面の法線方向に、球の内部にある質点をずらすことによって行います。質点の位置の更新は、以下の式に従って行います。

$$d=\frac{x\left(t+\Delta t\right)-c}{\|x\left(t+\Delta t\right)-c\|}$$
$$x'\left(t+\Delta t\right)=c+dr$$

$x'\left(t+\Delta t\right)$ は、衝突の後更新された位置です。d は、質点が深く貫通しないとすると、衝突点における表面への法線の許容できる精度の近似とみなすことができます。



▲図 3.5　衝突の計算

3.3　サンプルプログラム

3.3.1　リポジトリ

サンプルプログラムは、下記リポジトリ内の、**Assets/GPUClothSimulation** フォルダ内にあります。

https://github.com/IndieVisualLab/UnityGraphicsProgramming4

実行条件

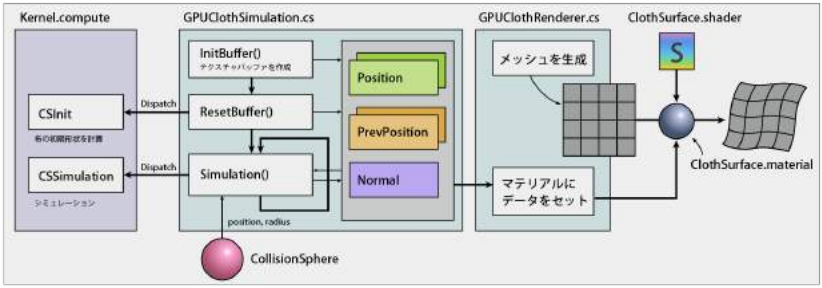
このサンプルプログラムでは、ComputeShader を使用しています。ComputeShader の動作環境についてはこちらをご確認下さい。

https://docs.unity3d.com/ja/2018.3/Manual/class-ComputeShader.html

3.4 実装の解説

3.4.1 処理の概略

それぞれのコンポーネントやコードがどのように関連して処理を行うかを示した概略図です。



▲図 3.6　各コンポーネント, コードの構造と処理の流れ

GPUClothSimulation.cs はシミュレーションに使用するデータや処理を管理する C#スプリプトです。このスクリプトは、シミュレーションに使用する位置や法線のデータを RenderTexture の形式で作成し管理します。また、Kernels.compute に記述されたカーネルを呼び出すことによって処理を行います。計算結果の視覚化は、GPUClothRenderer.cs スクリプトが行います。このスクリプトによって生成された Mesh オブジェクトは、計算結果である位置データ、法線データが格納された RenderTexture を参照した ClothSurface.shader の処理によって、ジオメトリを変形して描画されます。

GPUClothSimulation.cs

シミュレーションを制御する C#スクリプトです。

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

public class GPUClothSimulation : MonoBehaviour
```

```
{
    [Header("Simulation Parameters")]
    // タイムステップ
    public float   TimeStep = 0.01f;
    // シミュレーションの反復回数
    [Range(1, 16)]
    public int     VerletIterationNum = 4;
    // 布の解像度（横，縦）
    public Vector2Int ClothResolution = new Vector2Int(128, 128);
    // 布のグリッドの間隔（バネの自然長）
    public float   RestLength = 0.02f;
    // 布の伸縮性を決定する定数
    public float   Stiffness = 10000.0f;
    // 速度の減衰定数
    public float   Damp = 0.996f;
    // 質量
    public float   Mass = 1.0f;
    // 重力
    public Vector3 Gravity = new Vector3(0.0f, -9.81f, 0.0f);

    [Header("References")]
    // 衝突用球体の Transform の参照
    public Transform CollisionSphereTransform;
    [Header("Resources")]
    // シミュレーションを行うカーネル
    public ComputeShader KernelCS;

    // 布シミュレーション 位置データのバッファ
    private RenderTexture[] _posBuff;
    // 布シミュレーション 位置データ（ひとつ前のタイムステップ）のバッファ
    private RenderTexture[] _posPrevBuff;
    // 布シミュレーション 法線データのバッファ
    private RenderTexture _normBuff;

    // 布の長さ（横，縦）
    private Vector2 _totalClothLength;

    [Header("Debug")]
    // デバッグ用にシミュレーションバッファを表示する
    public bool EnableDebugOnGUI = true;
    // デバッグ表示時のバッファの表示スケール
    private float _debugOnGUIScale = 1.0f;

    // シミュレーションリソースを初期化したか
    public bool IsInit { private set; get; }

    // 位置データのバッファを取得
    public RenderTexture GetPositionBuffer()
    {
        return this.IsInit ? _posBuff[0] : null;
    }
    // 法線データのバッファを取得
    public RenderTexture GetNormalBuffer()
    {
        return this.IsInit ? _normBuff : null;
    }
    // 布の解像度を取得
    public Vector2Int GetClothResolution()
```

```
{
    return ClothResolution;
}

// ComputeShader カーネルの X, Y 次元のスレッドの数
const int numThreadsXY = 32;

void Start()
{
    var w = ClothResolution.x;
    var h = ClothResolution.y;
    var format = RenderTextureFormat.ARGBFloat;
    var filter = FilterMode.Point; // テクセル間の補間がなされないように
    // シミュレーション用のデータを格納する RenderTexture を作成
    CreateRenderTexture(ref _posBuff,     w, h, format, filter);
    CreateRenderTexture(ref _posPrevBuff, w, h, format, filter);
    CreateRenderTexture(ref _normBuff,    w, h, format, filter);
    // シミュレーション用のデータをリセット
    ResetBuffer();
    // 初期化済みのフラグを True
    IsInit = true;
}

void Update()
{
    // r キーを押したら，シミュレーション用のデータをリセットする
    if (Input.GetKeyUp("r"))
        ResetBuffer();

    // シミュレーションを行う
    Simulation();
}

void OnDestroy()
{
    // シミュレーション用のデータを格納する RenderTexture を削除
    DestroyRenderTexture(ref _posBuff    );
    DestroyRenderTexture(ref _posPrevBuff);
    DestroyRenderTexture(ref _normBuff   );
}

void OnGUI()
{
    // デバッグ用にシミュレーション用データを格納した RenderTexture を描画
    DrawSimulationBufferOnGUI();
}

// シミュレーション用データをリセット
void ResetBuffer()
{
    ComputeShader cs = KernelCS;
    // カーネル ID を取得
    int kernelId = cs.FindKernel("CSInit");
    // ComputeShader カーネルの実行スレッドグループの数を計算
    int groupThreadsX =
        Mathf.CeilToInt((float)ClothResolution.x / numThreadsXY);
    int groupThreadsY =
        Mathf.CeilToInt((float)ClothResolution.y / numThreadsXY);
```

```
        // 布の長さ（横，縦）の計算
        _totalClothLength = new Vector2(
            RestLength * ClothResolution.x,
            RestLength * ClothResolution.y
        );
        // パラメータ，バッファをセット
        cs.SetInts  ("_ClothResolution",
            new int[2] { ClothResolution.x, ClothResolution.y });
        cs.SetFloats("_TotalClothLength",
            new float[2] { _totalClothLength.x, _totalClothLength.y });
        cs.SetFloat ("_RestLength", RestLength);
        cs.SetTexture(kernelId, "_PositionBufferRW",     _posBuff[0]);
        cs.SetTexture(kernelId, "_PositionPrevBufferRW", _posPrevBuff[0]);
        cs.SetTexture(kernelId, "_NormalBufferRW",       _normBuff);
        // カーネルを実行
        cs.Dispatch(kernelId, groupThreadsX, groupThreadsY, 1);
        // バッファをコピー
        Graphics.Blit(_posBuff[0],     _posBuff[1]);
        Graphics.Blit(_posPrevBuff[0], _posPrevBuff[1]);
    }

    // シミュレーション
    void Simulation()
    {
        ComputeShader cs = KernelCS;
        // CSSimulation の計算 1 回あたりのタイムステップの値の計算
        float timestep = (float)TimeStep / VerletIterationNum;
        // カーネル ID を取得
        int kernelId = cs.FindKernel("CSSimulation");
        // ComputeShader カーネルの実行スレッドグループの数を計算
        int groupThreadsX =
            Mathf.CeilToInt((float)ClothResolution.x / numThreadsXY);
        int groupThreadsY =
            Mathf.CeilToInt((float)ClothResolution.y / numThreadsXY);

        // パラメータをセット
        cs.SetVector("_Gravity", Gravity);
        cs.SetFloat ("_Stiffness", Stiffness);
        cs.SetFloat ("_Damp", Damp);
        cs.SetFloat ("_InverseMass", (float)1.0f / Mass);
        cs.SetFloat ("_TimeStep", timestep);
        cs.SetFloat ("_RestLength", RestLength);
        cs.SetInts  ("_ClothResolution",
            new int[2] { ClothResolution.x, ClothResolution.y });

        // 衝突用球のパラメータをセット
        if (CollisionSphereTransform != null)
        {
            Vector3 collisionSpherePos = CollisionSphereTransform.position;
            float collisionSphereRad =
                CollisionSphereTransform.localScale.x * 0.5f + 0.01f;
            cs.SetBool  ("_EnableCollideSphere", true);
            cs.SetFloats("_CollideSphereParams",
                new float[4] {
                    collisionSpherePos.x,
                    collisionSpherePos.y,
                    collisionSpherePos.z,
                    collisionSphereRad
```

```
            });
        }
        else
            cs.SetBool("_EnableCollideSphere", false);

        for (var i = 0; i < VerletIterationNum; i++)
        {
            // バッファをセット
            cs.SetTexture(kernelId, "_PositionBufferRO",     _posBuff[0]);
            cs.SetTexture(kernelId, "_PositionPrevBufferRO", _posPrevBuff[0]);
            cs.SetTexture(kernelId, "_PositionBufferRW",     _posBuff[1]);
            cs.SetTexture(kernelId, "_PositionPrevBufferRW", _posPrevBuff[1]);
            cs.SetTexture(kernelId, "_NormalBufferRW",       _normBuff);
            // スレッドを実行
            cs.Dispatch(kernelId, groupThreadsX, groupThreadsY, 1);
            // 読み込み用バッファ，書き込み用バッファを入れ替え
            SwapBuffer(ref _posBuff[0],     ref _posBuff[1]    );
            SwapBuffer(ref _posPrevBuff[0], ref _posPrevBuff[1]);
        }
    }

    // シミュレーション用のデータを格納する RenderTexture を作成
    void CreateRenderTexture(ref RenderTexture buffer, int w, int h,
        RenderTextureFormat format, FilterMode filter)
    {
        buffer = new RenderTexture(w, h, 0, format)
        {
            filterMode = filter,
            wrapMode   = TextureWrapMode.Clamp,
            hideFlags  = HideFlags.HideAndDontSave,
            enableRandomWrite = true
        };
        buffer.Create();
    }

    // シミュレーション用のデータを格納する RenderTexture[] を作成
    void CreateRenderTexture(ref RenderTexture[] buffer, int w, int h,
        RenderTextureFormat format, FilterMode filter)
    {
        // ～ 略 ～
    }

    // シミュレーション用のデータを格納する RenderTexture を削除
    void DestroyRenderTexture(ref RenderTexture buffer)
    {
        // ～ 略 ～
    }

    // シミュレーション用のデータを格納する RenderTexture[] を削除
    void DestroyRenderTexture(ref RenderTexture[] buffer)
    {
        // ～ 略 ～
    }

    // マテリアルを削除
    void DestroyMaterial(ref Material mat)
    {
        // ～ 略 ～
```

```
    }

    // バッファを入れ替え
    void SwapBuffer(ref RenderTexture ping, ref RenderTexture pong)
    {
        RenderTexture temp = ping;
        ping = pong;
        pong = temp;
    }

    // デバッグ用に、シミュレーションのためのバッファを OnGUI 関数内で描画
    void DrawSimulationBufferOnGUI()
    {
        // ～ 略 ～
    }
}
```

冒頭では、シミュレーションに必要なパラメータを宣言しています。また、シミュレーションの結果を保持するものとして RenderTexture を用います。このシミュレーションに使用し、また得られるデータは

1. 位置
2. 一つ前のタイムステップにおける位置
3. 法線

です。

■InitBuffer 関数　InitBuffer 関数では、計算に必要なデータを格納する RenderTexture を作成します。位置と、一つ前のタイムステップにおける位置については、それぞれ、一つ前のタイムステップにおけるデータを使用し、それをもとに計算を行うので、読み込み用と書き込み用に 2 つ作成します。このように、読み込み用のデータと書き込み用のデータを作成し、効率良くシェーダに計算させる手法を、**Ping Pong Buffering** と呼びます。

RenderTexture の作成ですが、format では、テクスチャの精度（チャンネル数やそれぞれのチャンネルの bit 数）を設定します。一般には低い方が処理は高速になりますが、ARGBHalf（チャンネルあたり 16bit）では精度が低く、計算結果が不安定になるので、ARGBFloat（チャンネルあたり 32bit）に設定します。また、ComputeShader で計算結果の書き込みを可能にするため、enableRandomWrite は、true にします。RenderTexture は、コンストラクタの呼び出しだけではハードウェア上に作成されないので、Create 関数を実行し、シェーダで使用できる状態にします。

■ResetBuffer 関数　ResetBuffer 関数では、シミュレーションに必要なデータを格納する RenderTexture を初期化します。カーネル ID の取得、スレッドグループ数の計算、布の長さなど各種パラメータ、計算に使用する RenderTexture のセットを

ComputeShader に対して行い、Kernels.compute 内に書かれた CSInit カーネルを呼び出して処理を行います。CSInit カーネルの内容については、次の Kernels.compute の詳細の中で説明します。

■Simulation 関数　Simulation 関数は、実際の布のシミュレーションを行います。冒頭では、ResetBuffer 関数同様、カーネル ID の取得、スレッドグループ数の計算、シミュレーションに使用する各種パラメータや RenderTexture のセットを行います。一度に大きなタイムステップで計算すると、シミュレーションが不安定になるので、Update() 内で、タイムステップを小さな値に細かく分けて、シミュレーションを何回かに分割して安定して計算できるようにしています。反復回数については、**VerletIterationNum** で設定します。

Kernels.compute

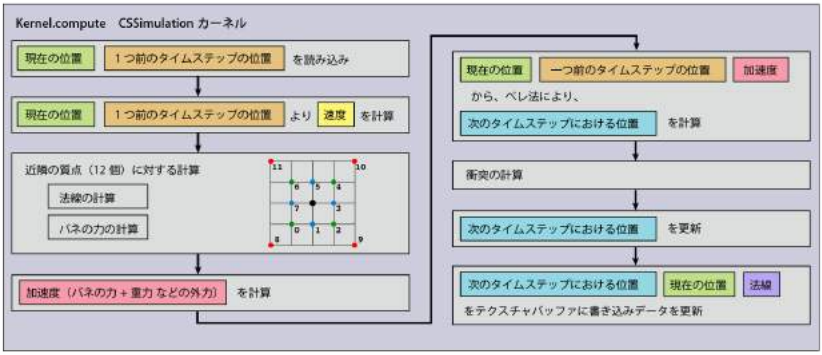
実際のシミュレーションなどの処理を記述した ComputeShader です。
この ComputeShader には、

1. CSInit
2. CSSimulation

の二つのカーネルがあります。
カーネルは、それぞれ以下のような処理を行います。

■CSInit　位置と法線の初期値の計算を行います。質点の位置については、スレッドの ID（2 次元）をもとに、X-Y 平面上に格子状に配置されるように計算します。

■CSSimulation　シミュレーションを行います。CSSimulation カーネルの処理の流れを概略的に示すと下記の図のようになります。



▲図 3.7　CSSimulation カーネルにおける計算の流れ

以下、コードを示します。

```
#pragma kernel CSInit
#pragma kernel CSSimulation

#define NUM_THREADS_XY 32 // カーネルのスレッド数

// 位置データ（1 つ前のタイムステップ）読み込み用
Texture2D<float4>   _PositionPrevBufferRO;
// 位置データ 読み込み用
Texture2D<float4>   _PositionBufferRO;
// 位置データ（1 つ前のタイムステップ）書き込み用
RWTexture2D<float4> _PositionPrevBufferRW;
// 位置データ 書き込み用
RWTexture2D<float4> _PositionBufferRW;
// 法線データ 書き込み用
RWTexture2D<float4> _NormalBufferRW;

int2   _ClothResolution;  // 布の解像度（パーティクル数）（横，縦）
float2 _TotalClothLength; // 布の総合的な長さ

float  _RestLength;       // バネの自然長

float3 _Gravity;          // 重力
float  _Stiffness;        // 布の伸縮度合いを決定する定数
float  _Damp;             // 布の速度の減衰率
float  _InverseMass;      // 1.0/質量

float  _TimeStep;         // タイムステップの大きさ

bool   _EnableCollideSphere; // 衝突処理を行うかのフラグ
float4 _CollideSphereParams; // 衝突処理用パラメータ（pos.xyz，radius）
```

```
// 近傍のパーティクルの ID オフセット（x, y）の配列
static const int2 m_Directions[12] =
{
    int2(-1, -1), //  0
    int2( 0, -1), //  1
    int2( 1, -1), //  2
    int2( 1,  0), //  3
    int2( 1,  1), //  4
    int2( 0,  1), //  5
    int2(-1,  1), //  6
    int2(-1,  0), //  7
    int2(-2, -2), //  8
    int2( 2, -2), //  9
    int2( 2,  2), // 10
    int2(-2,  2)  // 11
};
// 近傍のパーティクルの ID のオフセットを返す
int2 NextNeigh(int n)
{
    return m_Directions[n];
}

// シミュレーション用バッファの初期化を行うカーネル
[numthreads(NUM_THREADS_XY, NUM_THREADS_XY, 1)]
void CSInit(uint3 DTid : SV_DispatchThreadID)
{
    uint2 idx = DTid.xy;

    // 位置
    float3 pos = float3(idx.x * _RestLength, idx.y * _RestLength, 0);
    pos.xy -= _TotalClothLength.xy * 0.5;
    // 法線
    float3 nrm = float3(0, 0, -1);
    // バッファに書き込み
    _PositionPrevBufferRW[idx] = float4(pos.xyz, 1.0);
    _PositionBufferRW[idx]     = float4(pos.xyz, 1.0);
    _NormalBufferRW[idx]       = float4(nrm.xyz, 1.0);
}

// シミュレーションを行うカーネル
[numthreads(NUM_THREADS_XY, NUM_THREADS_XY, 1)]
void CSSimulation(uint2 DTid : SV_DispatchThreadID)
{
    int2 idx = (int2)DTid.xy;
    // 布の解像度（パーティクル数）（横，縦）
    int2   res = _ClothResolution.xy;
    // 位置を読み込み
    float3 pos = _PositionBufferRO[idx.xy].xyz;
    // 位置（ひとつ前のタイムステップ）の読み込み
    float3 posPrev = _PositionPrevBufferRO[idx.xy].xyz;
    // 位置とひとつ前のタイムステップの位置より，速度を計算
    float3 vel = (pos - posPrev) / _TimeStep;

    float3 normal   = (float3)0; // 法線
    float3 lastDiff = (float3)0; // 法線計算時に使用する方向ベクトル格納用変数
    float  iters    = 0.0;       // 法線計算時のイテレーション数加算用変数

    // パーティクルにかかる力，初期値として重力の値を代入
```

```
float3 force = _Gravity.xyz;
// 1.0 / 質量
float  invMass = _InverseMass;

// 布の上辺であれば位置を固定するために計算しない
if (idx.y == _ClothResolution.y - 1)
    return;

// 近傍のパーティクル（12 個）についての計算を行う
[unroll]
for (int k = 0; k < 12; k++)
{
    // 近傍パーティクルの ID（座標）のオフセット
    int2 neighCoord = NextNeigh(k);
    // X 軸，端のパーティクルについては計算しない
    if (((idx.x+neighCoord.x) < 0) || ((idx.x+neighCoord.x) > (res.x-1)))
        continue;
    // Y 軸，端のパーティクルについては計算しない
    if (((idx.y+neighCoord.y) < 0) || ((idx.y+neighCoord.y) > (res.y-1)))
        continue;
    // 近傍のパーティクルの ID
    int2   idxNeigh = int2(idx.x + neighCoord.x, idx.y + neighCoord.y);
    // 近傍のパーティクルの位置
    float3 posNeigh = _PositionBufferRO[idxNeigh].xyz;
    // 近傍のパーティクルの位置の差
    float3 posDiff = posNeigh - pos;

    // 法線の計算
    // 基点から近傍のパーティクルへの方向ベクトル
    float3 currDiff = normalize(posDiff);
    if ((iters > 0.0) && (k < 8))
    {
        // 1 つ前に調べた近傍パーティクルとの方向ベクトルと
        // 現在のものの角度が鈍角であれば
        float a = dot(currDiff, lastDiff);
        if (a > 0.0) {
            // 外積による直行するベクトルを求めて加算
            normal += cross(lastDiff, currDiff);
        }
    }
    lastDiff = currDiff; // 次の近傍パーティクルとの計算時のために保持

    // 近傍パーティクルとのバネの自然長を計算
    float  restLength = length(neighCoord * _RestLength);
    // バネの力を計算
    force += (currDiff*(length(posDiff)-restLength))*_Stiffness-vel*_Damp;
    // 加算
    if (k < 8) iters += 1.0;
}
// 法線ベクトルを計算
normal = normalize(normal / -(iters - 1.0));

// 加速度
float3 acc = (float3)0.0;
// 運動の法則を適用（加速度の大きさは，力の大きさに比例し質量に反比例する）
acc = force * invMass;

// ベレ法による位置計算
```

```
    float3 tmp = pos;
    pos = pos * 2.0 - posPrev + acc * (_TimeStep * _TimeStep);
    posPrev = tmp; // ひとつ前のタイムステップの位置

    // 衝突を計算
    if (_EnableCollideSphere)
    {
        float3 center = _CollideSphereParams.xyz; // 中心位置
        float  radius = _CollideSphereParams.w;   // 半径

        if (length(pos - center) < radius)
        {
            // 衝突球の中心から, 布のパーティクルの位置への単位ベクトルを計算
            float3 collDir = normalize(pos - center);
            // 衝突球の表面にパーティクルの位置を移動
            pos = center + collDir * radius;
        }
    }

    // 書き込み
    _PositionBufferRW[idx.xy]     = float4(pos.xyz,     1.0);
    _PositionPrevBufferRW[idx.xy] = float4(posPrev.xyz, 1.0);
    _NormalBufferRW[idx.xy]       = float4(normal.xyz,  1.0);
}
```

GPUClothRenderer.cs

このコンポーネントでは、

1. MeshRenderer を取得、または作成
2. 布のシミュレーションの解像度と一致した Mesh オブジェクトを生成
3. Mesh を描画するマテリアルに、シミュレーション結果（位置, 法線）を代入

を行います。
詳細はサンプルコードをご確認ください。

ClothSurface.shader

このシェーダでは、シミュレーションによって得られた位置、法線データを頂点シェーダ内で取得し、頂点の書き換えることによって、Mesh の形状を変化させます。
詳細はサンプルコードをご確認ください

3.4.2 実行結果

実行をすると、球に対して衝突をする布のような動きをするオブジェクトが確認できます。シミュレーションに用いる各種パラメータを変更すると、動き方の変化がみられます。

TimeStep は、Update 関数が 1 回実行されるときに進むシミュレーションの時間です。大きくすると、動きの変化が大きくなりますが、大きすぎる値を設定すると、シミュレーションが不安定になり、値が発散してしまいます。

VerletIterationNum は、Simulation 関数内で実行する CSSimulation カーネルの回数で、同じタイムステップの値でも、この値を大きくするとシミュレーションは安定しやすくなりますが、計算負荷は増大してしまいます。

ClothResolution は、布の解像度です。大きくすると、しわなどのディティールが多く見られるようになりますが、大きくしすぎるとシミュレーションが不安定になります。ComputeShader 上で、スレッドサイズを 32 で設定しているので 32 の倍数であることが望ましいです。

RestLength は、バネの自然長です。バネ同士の距離になりますので、布の長さは ClothResolution × RestLength となります。

Stiffness は、バネの硬さです。この値を大きくすると、布の伸縮が小さくなりますが、あまり大きくするとシミュレーションが不安定になります。

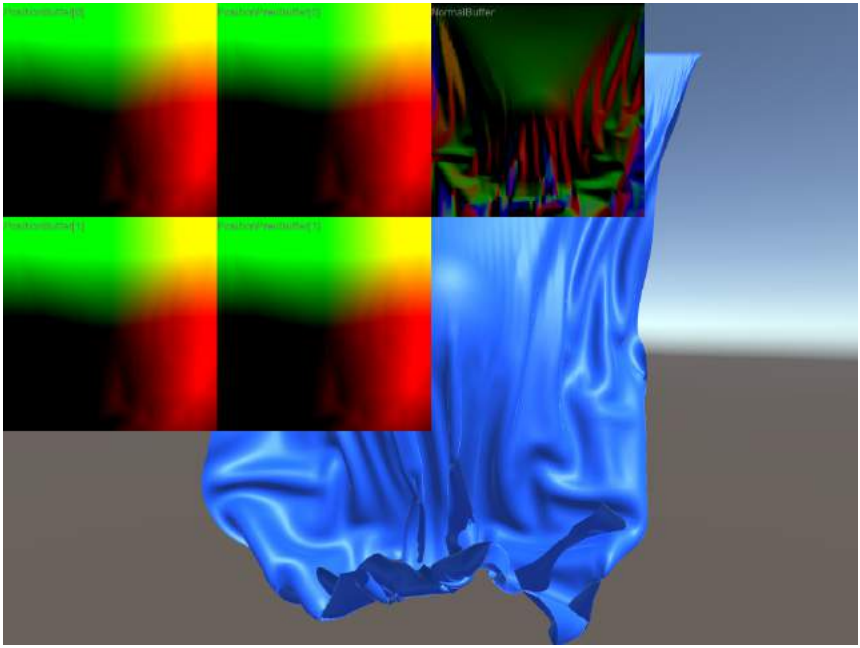
Damp は、バネの動く速度の減衰値です。この値を大きくすると、質点が速く動かなくなるようになり振動しにくくなりますが、変化が小さくなってしまいます。

Mass は、質点の質量です。この値を大きくすると、布が重たくなったような大きな動きをするようになります。

Gravity は、布にかかる重力です。布の加速度は、この重力とバネの力が合わさったものとなります。

EnableDebugOnGUI のチェックを入れると、スクリーンの左上に確認用に、シミュレーションの結果である位置・一つ前のタイムステップにおける位置・法線のデータを格納したテクスチャが描画されるようになっています。

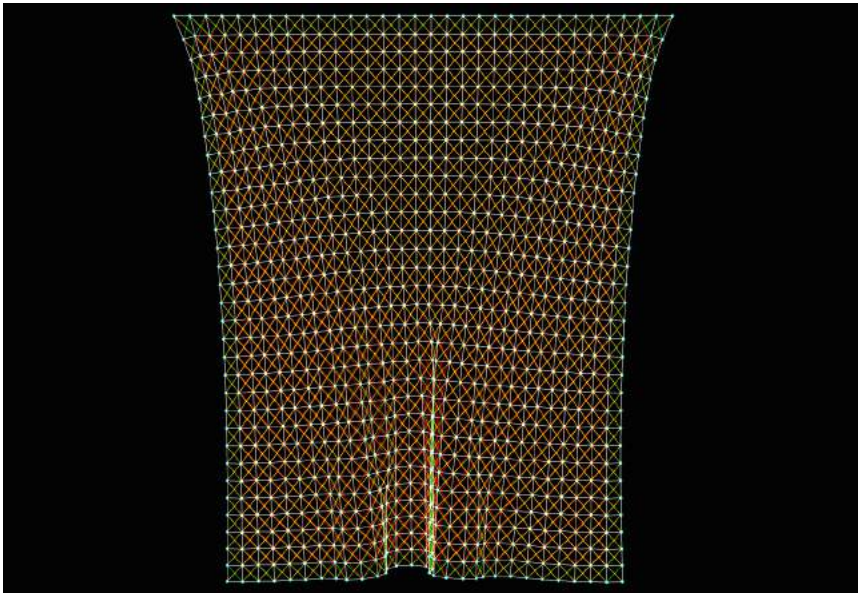
R キーを押すと、布を初期状態に戻るようにしています。シミュレーションが不安定になり、値が発散してしまったときは初期状態に戻すようにしてください。



▲図 3.8　実行結果（計算結果 RenderTexture をスクリーンスペースに描画）

デバッグとしてスプリングを描画

Assets/GPUClothSimulation/Debug/**GPUClothSimulationDebugRender.unity** を開くと、質点と質点間を接続したバネをパーティクルとラインで描画したものが確認できます。



▲図 3.9　質点とバネをパーティクルとラインで描画

3.5　まとめ

質点-バネ系のシミュレーションで得られる動きは、力の加え方によっては複雑に変化し、面白い形状が作成できます。この章で紹介した布シミュレーションは非常に単純なものです。球のような単純なものではなく複雑なジオメトリのオブジェクトとの衝突、布同士の衝突、摩擦、布の繊維構造の考慮、大きなタイムステップをとったときのシミュレーションの安定性など、様々な課題を克服するために、多くの研究がなされています。物理エンジンに興味のある方は追究してみてはいかがでしょうか。

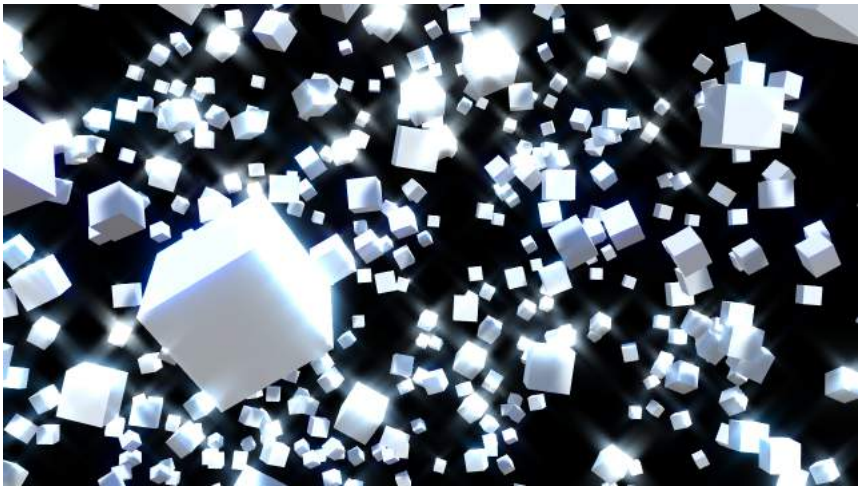
3.6　参考

- [1] Marco Fratarcangeli, "Game Engine Gems 2, GPGPU Cloth simulation using GLSL, OpenCL, and CUDA", (参照 2019-04-06) - http://www.cse.chalmers.se/~marcof/publication/geg2011/
- [2] Wikipedia - Verlet integration, (参照 2019-04-06) - https://en.wikipedia.org/wiki/Verlet_integration

- [3] 藤澤 誠, CG のための物理シミュレーションの基礎, 株式会社マイナビ, 2013
- [4] 酒井幸市, WebGL による物理シミュレーション, 株式会社工学社, 2014
- [5] 酒井幸市, OpenGL で作る力学アニメーション入門, 森北出版株式会社, 2005

第 4 章

StarGlow



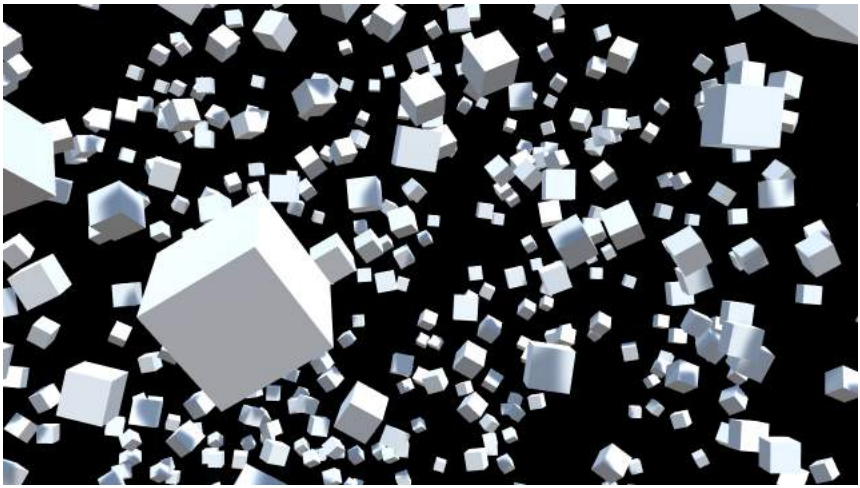
▲図 4.1　輝度の高い部分から伸びる光線

強い光の照り返しが生じるときに伸びる光線、LightLeak だったり LightStreak だったり、StarGlow と呼ばれますが、ポストエフェクトでこれを表現してみましょう。ここでは便宜上 StarGlow(スターグロー) と呼びます。

ここで紹介するこのポストエフェクトは GDC 2003 で Masaki Kawase 氏によって発表されたものです。

本章のサンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming4`
の「StarGlow」です。

4.1　STEP 1：輝度画像を生成する



▲図 4.2　オリジナルのイメージ

▲図 4.3　高い輝度の画素だけを検出したイメージ

まずは明るい部分だけを検出した画像 (輝度画像) を作りましょう。一般的なグローについても同じような処理を必要としますね。輝度画像を作るためのシェーダとスクリプトのソースコードは次のようになります。シェーダパスが 1 である点に注意してください。

▼ StarGlow.cs

```
RenderTexture brightnessTex
= RenderTexture.GetTemporary(source.width  / this.divide,
                             source.height / this.divide,
                             source.depth,
                             source.format);
…
base.material.SetVector
(this.idParameter, new Vector3(threshold, intensity, attenuation));

Graphics.Blit(source, brightnessTex, base.material, 1);
```

▼ StarGlow.shader

```
#define BRIGHTNESS_THRESHOLD _Parameter.x
#define INTENSITY            _Parameter.y
#define ATTENUATION          _Parameter.z
…
fixed4 frag(v2f_img input) : SV_Target
{
    float4 color = tex2D(_MainTex, input.uv);
    return max(color - BRIGHTNESS_THRESHOLD, 0) * INTENSITY;
}
```

輝度の算出方法についてはさまざまな方法がありますが、古典的なグローの実装などでも使われている算出方式をそのまま活用しました。他に一度グレースケールにしてから輝度を比較するなどの処理を行っているシェーダも見かけます。

BRIGHTNESS_THRESHOLD は、輝度と判定する閾値、INTENSITY は輝度に乗算するバイアスです。color に与えられる値が大きいほど、つまり明るい色ほど、大きな値が返りやすいことを確認してください。閾値が大きいほど、0 より大きな値が返る確率は減りますね。またバイアスが大きいほど、より強い輝度画像が得られるようになっています。

ATTENUATION についてはこの時点では使いません。一度にパラメータとして渡した方が CPU → GPU 間での値のやり取りにかかるオーバーヘッドが小さくなるために、ここでは Vector3 としてまとめて渡しています。

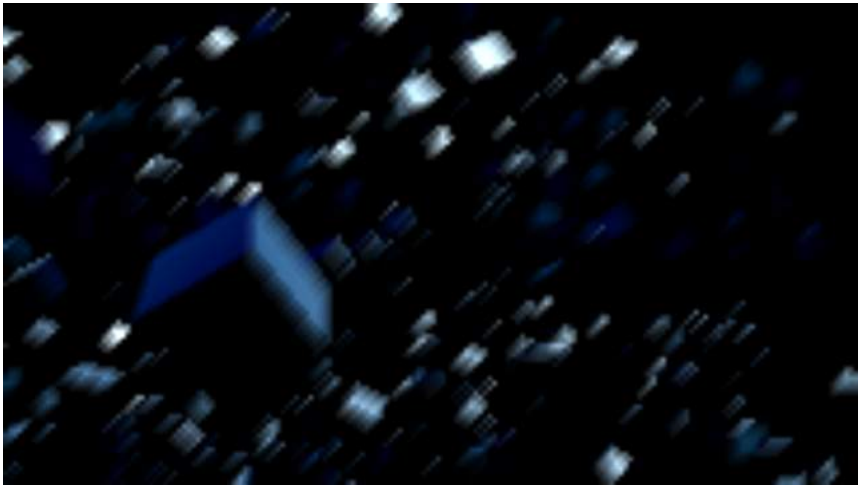
この時点でもっとも重要なのは、輝度画像をサイズの小さい RenderTexture として取得している点です。

一般に、ポストエフェクトは解像度が大きくなればなるほど、Fragment シェーダの呼び出し数、その算出回数が増えて負荷が大きくなります。さらにグロー効果につ

いては繰り返し処理が発生するために処理負荷はさらに大きくなるのです。スターグローもこの例に漏れません。したがって、効果の解像度を必要な分まで下げることによって、かかる負荷を軽減します。

繰り返し処理については後述します。

4.2 STEP 2：輝度画像に指向性ブラーをかける



▲図 4.4　斜めに引き伸ばされた輝度画像

STEP1 で得られた輝度画像にブラーをかけて引き伸ばします。この引き伸ばし方を工夫することによって、一般的なグローとは異なる鋭く伸びる光線を表現します。

一般的なグローの場合は全方向に向かってガウス関数による引き伸ばしをしますが、スターグローの場合は指向性のある引き伸ばしをする、ということです。

▼ StarGlow.cs

```
Vector2 offset = new Vector2(-1, -1);
// (Quaternion.AngleAxis(angle * x + this.angleOfStreak,
//                       Vector3.forward) * Vector2.down).normalized;

base.material.SetVector(this.idOffset, offset);
base.material.SetInt   (this.idIteration, 1);

Graphics.Blit(brightnessTex, blurredTex1, base.material, 2);

for (int i = 2; i <= this.iteration; i++)
```

```
{
    繰り返し描画
}
```

実際の処理とは異なりますが、ここでは説明のために `offset = (1, 1)` としましょう。さらに、`offset` と `iteration` をシェーダに渡していることに注目してください。

続いてスクリプト側ではシェーダパス 2 で繰り返し描画を実行していますが、簡単に考えるために、ここで一度シェーダの方に移りましょう。シェーダパス 2 で描画していることに注意してください。

▼ StarGlow.shader

```
int    _Iteration;
float2 _Offset;

struct v2f_starglow
{
    …
    half   power  : TEXCOORD1;
    half2  offset : TEXCOORD2;
};

v2f_starglow vert(appdata_img v)
{
    v2f_starglow o;
    …
    o.power  = pow(4, _Iteration - 1);
    o.offset = _MainTex_TexelSize.xy * _Offset * o.power;
    return o;
}

float4 frag(v2f_starglow input) : SV_Target
{
    half4 color = half4(0, 0, 0, 0);
    half2 uv    = input.uv;

    for (int j = 0; j < 4; j++)
    {
        color += saturate(tex2D(_MainTex, uv)
               * pow(ATTENUATION, input.power * j));
        uv += input.offset;
    }

    return color;
}
```

まずは Vertex シェーダから確認します。`power` は引き伸ばされるときに輝度が減衰する力、`offset` はブラーによって輝度を引き伸ばす方向を示します。後述する Fragment シェーダで参照します。

Vertex シェーダ内でこれらを算出しているのは、Fragment シェーダ内において共通の値を参照するためです。Fragment シェーダ内で逐次算出するのは演算回数が増

えるので良くありませんね。

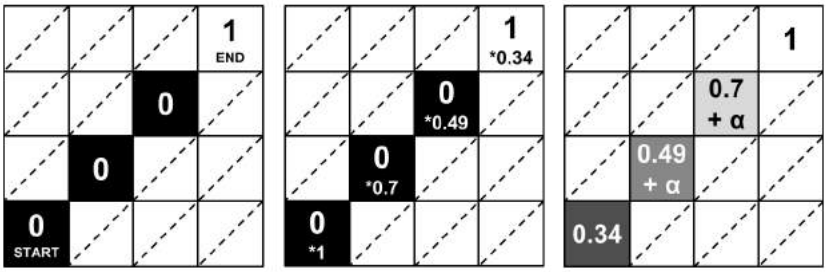
ここで _Iteration = 1 です。したがって power = 4^0 = 1 になります。そうると offset = 画素の大きさ * (1, 1) が得られます。

これで、ちょうど 1 画素分だけオフセット方向にズレた画素をサンプリングするための準備が整いました。

次に Fragment シェーダです。参照する uv を offset の分だけ 1 つずつ移動しながら 4 回参照し、その画素の値を合算しています。ただし画素の値には pow(ATTENUATION, input.power * j) が乗算されていますね。

ATTENUATION はその画素の値をどれくらい減衰するかを表す値です。引き伸ばしたときのボケ、減衰具合に影響します。

仮に ATTENUATION = 0.7 とすれば、最初にサンプリングする画素は * 0.7, 次にサンプリングする画素は 0.7 ^ 2 = * 0.49 となりますね。図で見るとイメージが付きやすいです。



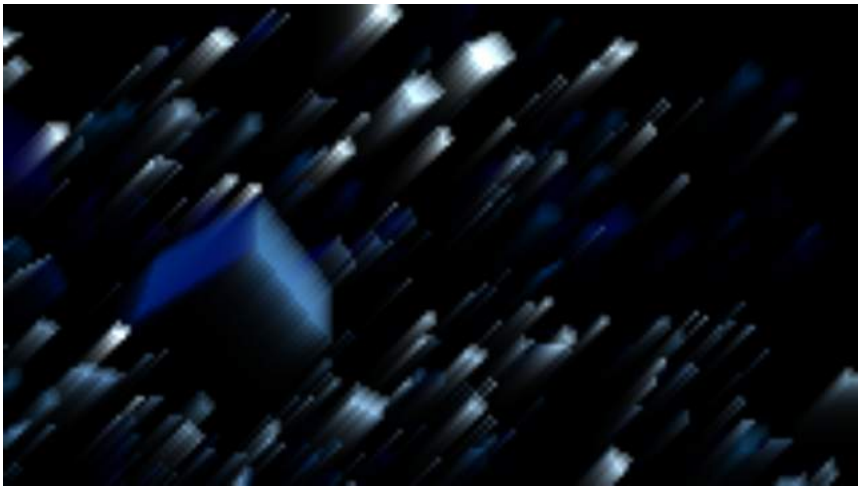
▲図 4.5　ブラーがかかる過程を表した図

左の図が減衰前のオリジナルの輝度画像です。_MainTex に相当します。今 Fragment シェーダに与えられる uv が参照する画素を左下の START としましょう。offset = (1, 1) ですから、4 回のイテレーションで参照する画素は右上の END までです。

画素の中の値は、その画素のもつ輝度の値です。START から 3 つが 0 で、END だけが 1 です。先のソースコードはイテレーションするたびに減衰率が上がりますから、ちょうど真ん中の図のようなイメージになります。これを合算すると、START の画素が最終的に得られる値は color = 0.34 になります。

同じようにして Fragment シェーダが各画素を処理していけば、右の図のような結果が得られることが分かると思います。ブラーのようなグラデーションが得られていますね。また offset が引き伸ばす方向を示すパラメータであると先に説明していますが、見た目上の効果としては、指定した値と反対方向に伸びることになります。

4.2.1 繰り返してさらに引き伸ばす



▲図 4.6　さらに引き伸ばされた輝度画像

少しだけスクリプト側に話を戻しましょう。先までの説明は `this.iteration` ないし `_Iteration` が 1 であるとしていました。実際には RenderTexture を入れ替えながら、任意の回数だけ同じ処理を繰り返しています。

▼ StarGlow.cs

```
Vector2 offset = new Vector2(-1, -1);

base.material.SetVector(this.idOffset, offset);
base.material.SetInt   (this.idIteration, 1);

Graphics.Blit(brightnessTex, blurredTex1, base.material, 2);

== ここから上が先までの解説に相当する ==

for (int i = 2; i <= this.iteration; i++)
{
    base.material.SetInt(this.idIteration, i);

    Graphics.Blit(blurredTex1, blurredTex2, base.material, 2);

    RenderTexture temp = blurredTex1;
    blurredTex1 = blurredTex2;
    blurredTex2 = temp;
```

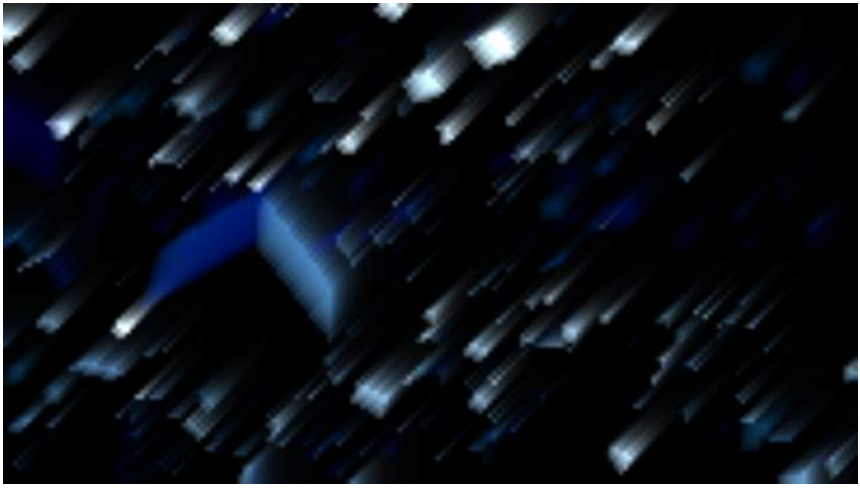
```
}
```

同じパスを使って同じ処理を繰り返していますから、得られる効果は変わりません。ただし、シェーダパラメータの `_Iteration` の値が大きくなりますね、そうすると、先に説明したシェーダ内の減衰率が上がります。また、入力される画像はすでに引き伸ばされているブラー画像になります。

端的にいえば、この繰り返し処理によって、1 回目よりもさらに伸びたブラー画像が `blurredTex1` に得られます。

この処理はコストがかかるので、現実的には繰り返し回数は精々 3 回くらいとは思います。また、シェーダ内のイテレーションは 4 回ですが、この値は Kawase 氏の発表で提案されているものです。

4.3 STEP 2.5：複数方向に伸びるブラー画像を合成する



▲図 4.7　別の方向に引き伸ばされた輝度画像

ソースコード上では STEP2.5 はコメントしていないのですが、説明のために 2.5 としました。先の説明では `offset = (1, 1)` としていましたが、複数方向に伸びる光線を作るために、`offset` を回転してもう一度ブラーをかけましょう。

仮に `offset = (1, 1)` と反対方向に伸びる光線を定義するとすれば、`offset = (-1, -1)` ですね。実際のソースコードでは光線の数だけ `offset` を回転しています

が、説明の上では offset = (-1, -1) とします。

▼ StarGlow.cs

```
for (int x = 1; x <= this.numOfStreak; x++)
{
    Vector2 offset = Quaternion.AngleAxis(angle * x + this.angleOfStreak,
                                          Vector3.forward) * Vector2.down;
    offset = offset.normalized;

    for (int i = 2; i <= this.iteration; i++) {
        bluuredTex1 が繰り返し処理で伸ばされる
    }

    Graphics.Blit(blurredTex1, compositeTex, base.material, 3);
}
```

最終的に得られたブラー画像 blurredTex1 を、合成用の画像 compositeTex に出力しています。compositeTex は複数方向に伸びるブラー画像がすべて合成された画像になりますね。

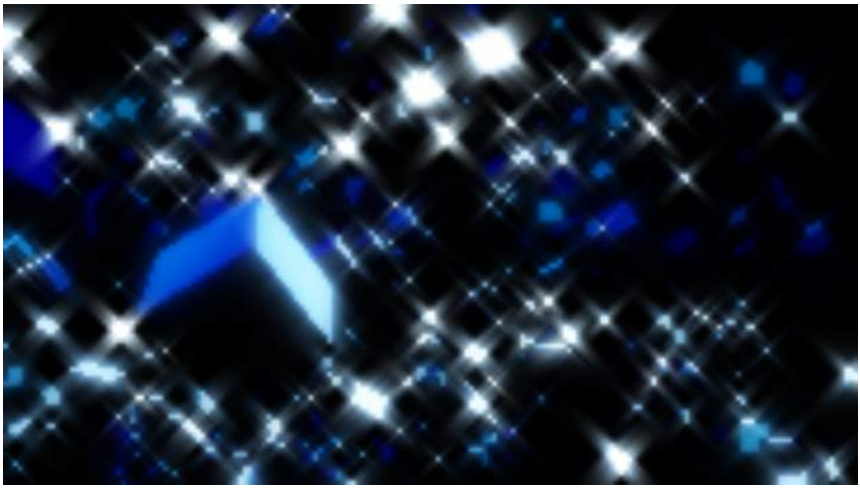
このとき、ブラー画像を合成するために使うシェーダーパスは 3 です。

▼ StarGlow.shader

```
Blend OneMinusDstColor One
…
fixed4 frag(v2f_img input) : SV_Target
{
    return tex2D(_MainTex, input.uv);
}
```

このパスでは特別な処理は一切していませんが、Blend 構文を使って画像を合成しています。合成方法は演出によって作り替えてしまってよいと思いますが、ここでは OneMinusDstColor One としました。ソフトな合成方法です。

4.4 STEP 3 : ブラー画像を元画像に合成する



▲図 4.8　最終的に得られたブラー画像

複数方向に延びるブラー画像が得られたら、あとは一般的なグローと同じように元画像にブラー画像を合成して出力します。先の STEP 2.5 と同じように `Blend` 構文を使って合成して出力する方法でもよいですが、ここでは `Blit` 回数の軽減と、合成方法の柔軟性のために合成用の `Pass 4` を使って合成するようにしています。

▼ StarGlow.cs

```
base.material.EnableKeyword(StarGlow.CompositeTypes[this.compositeType]);
base.material.SetColor(this.idCompositeColor, this.color);
base.material.SetTexture(this.idCompositeTex, compositeTex);

Graphics.Blit(source, destination, base.material, 4);
```

▼ StarGlow.shader

```
#pragma multi_compile _COMPOSITE_TYPE_ADDITIVE _COMPOSITE_TYPE_SCREEN …
…
fixed4 frag(v2f_img input) : SV_Target
{
    float4 mainColor      = tex2D(_MainTex,      input.uv);
    float4 compositeColor = tex2D(_CompositeTex, input.uv);
```

```
    #if defined(_COMPOSITE_TYPE_COLORED_ADDITIVE)…
     || defined(_COMPOSITE_TYPE_COLORED_SCREEN)

    compositeColor.rgb
        = (compositeColor.r + compositeColor.g + compositeColor.b)
        * 0.3333 * _CompositeColor;

    #endif

    #if defined(_COMPOSITE_TYPE_SCREEN)…
     || defined(_COMPOSITE_TYPE_COLORED_SCREEN)

    return saturate(mainColor + compositeColor
                  - saturate(mainColor * compositeColor));

    #elif defined(_COMPOSITE_TYPE_ADDITIVE)…
       || defined(_COMPOSITE_TYPE_COLORED_ADDITIVE)

    return saturate(mainColor + compositeColor);

    #else

    return compositeColor;

    #endif
}
```

Blend 構文は使っていないものの、スクリーン合成、加算合成をそのまま再現しています。さらにここでは任意に乗算する色を追加することによって、色の強く付いたスターグローを表現できるようにしています。

4.5 STEP 4：リソースの開放

使ったリソースはすべて開放しましょう。特別解説することもありませんが、ソースコード上のサンプルにも記載してるので念のため。実装環境などが限定的な場合は確保済みのリソースを使い回すなどの対応も可能でしょうが、ここではシンプルに Release します。

▼ StarGlow.cs

```
base.material.DisableKeyword(StarGlow.CompositeTypes[this.compositeType]);

RenderTexture.ReleaseTemporary(brightnessTex);
RenderTexture.ReleaseTemporary(blurredTex1);
RenderTexture.ReleaseTemporary(blurredTex2);
RenderTexture.ReleaseTemporary(compositeTex);
```

4.6 まとめ

基本的 (Kawase 氏の発表のまま) なスターグローの実装方法について解説しましたが、リアルタイム性にこだわらなければ、輝度画像の算出方法やパラメータを複数回切り替えることで、多用な光線を表現する方法なども考えられますね。

ここで解説した範囲でも、たとえばイテレーションのタイミングでパラメータを変更するなどすると、不均質でより "らしい", "味のある" 光線が作れるでしょう。あるいはノイズを使って時間ごとにパラメータを変化するのもよいと思います。

物理的に正しい光線ではありませんし、より演出的で高度な光線の表現が必要なときは、ポストエフェクト以外の方法で実現することになるのでしょうが、比較的シンプルなつくりで華やかにできるこのエフェクトもすごく面白いのでぜひ試してみてください。

…少々処理が重いですけど。

4.7 参照

- Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L(Wreckless)
  - http://www.daionet.gr.jp/~masa/archives/GDC2003_DSTEAL.ppt

第 5 章

Triangulation by Ear Clipping

5.1 はじめに

本章では、多角形の三角形分割の手法のひとつである、「耳刈り取り法（Ear Clipping 法）、以下耳刈り取り法」について説明します。通常の単純多角形の三角形分割だけでなく、穴のある多角形や階層構造になっている多角形の三角形分割についても説明します。

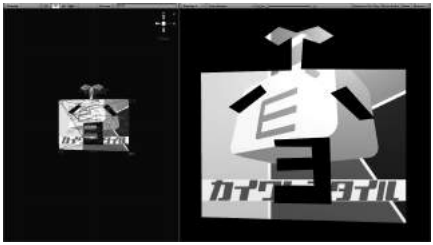
本章のサンプルは

`https://github.com/IndieVisualLab/UnityGraphicsProgramming4`

の「TriangulationByEarClipping」です。

5.1.1 サンプルの操作方法

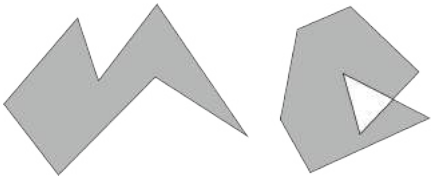
サンプルの DrawTest シーンを実行します。GameView を左クリックすると画面上に点を打ちます。続けて他の地点を左クリックすると最初の点と線で結びます。繰り返していくと多角形ができます。線を引く時、線が交差しないように注意してください。右クリックで多角形を三角形分割してメッシュを生成します。生成したメッシュの中で多角形を生成すると、穴のある多角形ができます。



▲図 5.1　サンプルを実行した画面

5.2　単純多角形の三角形分割

単純多角形とは、自己の線分で交差しない閉じた多角形の事を言います。

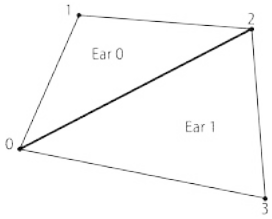


▲図 5.2　左：単純多角形、右：非単純多角形

どんな単純多角形も三角形分割が可能です。n 個の頂点をもつ単純多角形を三角形分割すると、n-2 個の三角形ができます。

5.3　耳刈り取り法（EarClipping 法）

多角形の三角形分割の手法はいくつも存在しますが、今回は実装がシンプルな「耳刈り取り法」について説明します。「耳刈り取り法」は、「Two ears theorem」という定理に基づいて分割します。この「Ear(耳)」というのは、「2 辺が多角形の辺であり、残り 1 辺が多角形の内部に存在する三角形」のことを指し、この定理は、「4 本以上の辺を持つ穴のない単純多角形は少なくとも 2 つの耳を持っている」という定理です。



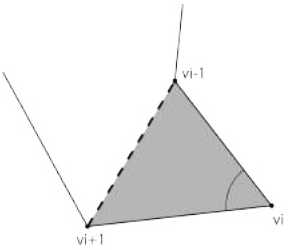
▲図 5.3　耳

「耳刈り取り法」は、この「耳」三角形を探し、多角形から取り除いていくアルゴリズムです。この「耳刈り取り法」ですが、他の分割アルゴリズムに比べてシンプルな半面、処理が遅いので速度が求められる場面ではあまり使えないと思います。

5.3.1　三角形分割の流れ

まず、与えられた多角形の頂点配列の中から「耳」を探します。「耳」の条件は以下の 2 点です。

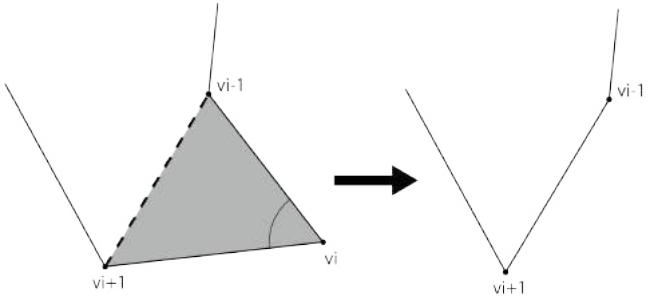
- 多角形の頂点 vi の前後の頂点 (vi-1,vi+1) との線分のなす角度 (内角) が 180 度以内（凸頂点という）
- 多角形の頂点 vi-1, vi, vi+1 からなる三角形の中に他の頂点が含まれていない



▲図 5.4　耳の条件（180 度以内、三角形の中に他の頂点が含まれない）

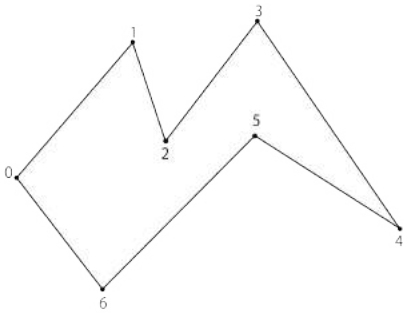
上記の条件を満たした頂点 vi を耳リストに追加します。この処理はサンプルの Triangulation.cs の InitializeVertices 関数が該当します。そして、耳リストの先頭から耳を構成する三角形を作成し、頂点 vi を頂点配列から取り除きます。
頂点 vi を取り除くと、多角形の形が変わります。残された頂点 vi-1,vi+1 に対して、再度前述の耳判定を行います。頂点 vi-1,vi+1 が耳の条件を満たしていれば、耳リス

トの末尾に追加されますが、逆に耳リストから削除される場合もあります。この処理はサンプルの Triangulation.cs の CheckVertex 関数と、EarClipping 関数が該当します。



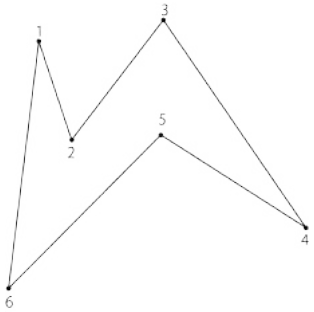
▲図 5.5　頂点 vi を削除する前の多角形と、削除したあとの多角形

ある単純多角形を例に一連の流れを図で表してみます。



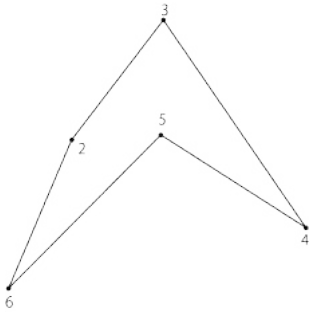
▲図 5.6　ある単純多角形

まず、耳を探します。この場合、耳リストには頂点 0,1,4,6 が含まれます。頂点 2,5 は凸頂点ではないので除外、頂点 3 は三角形 2,3,4 の中に頂点 5 が含まれているので除外されます。
まず、耳リストの先頭の頂点 0 を取り出します。頂点 0 の前後の頂点 1,6 で三角形を作ります。頂点 0 を頂点配列から削除し、前後の頂点 1,6 を結んで新しい多角形にします。そして頂点 1,6 について耳判定をします。もともと 2 つとも耳でしたが、耳判定後も耳のままです。この時の耳リストは、1,4,6 です。



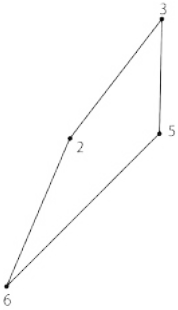
▲図 5.7　頂点 0 を削除した多角形

次に、耳リストの先頭から頂点 1 を取り出します。頂点 1 の前後の頂点 2,6 で三角形を作ります。頂点 1 を頂点配列から削除し、前後の頂点 2,6 を結んで新しい多角形にします。そして頂点 2,6 について耳判定をします。頂点 1 がなくなったことで、頂点 2 が凸頂点になり、耳の条件を満たしましたので、耳リストに追加します。頂点 6 は耳のままです。この時の耳リストは、4,6,2 です。



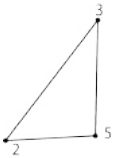
▲図 5.8　頂点 1 を削除した多角形

次に、耳リストの先頭から頂点 4 を取り出します。頂点 4 の前後の頂点 3,5 で三角形を作ります。頂点 4 を頂点配列から削除し、前後の頂点 3,5 を結んで新しい多角形にします。そして頂点 3,5 について耳判定をします。頂点 4 がなくなったことで、頂点 3 の前後の頂点 2,5 で作られる三角形の中に他の頂点が含まれなくなったので、頂点 3 を耳リストに追加します。また、頂点 5 の内角が 180 度以下になったことで凸頂点になり、耳の条件を満たしましたので、耳リストに追加します。この時の耳リストは、6,2,3,5 です。



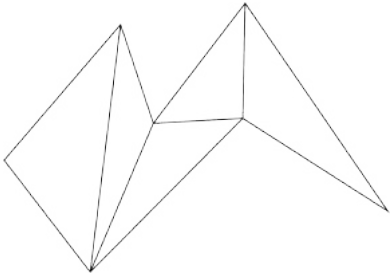
▲図 5.9　頂点 4 を削除した多角形

次に、耳リストの先頭から頂点 6 を取り出します。頂点 6 の前後の頂点 2,5 で三角形を作ります。頂点 6 を頂点配列から削除し、前後の頂点 2,5 を結んで新しい多角形にします。そして頂点 2,5 について耳判定をします。もともと 2 つとも耳でしたが、耳判定後も耳のままです。この時の耳リストは、2,3,5 です。



▲図 5.10　頂点 6 を削除した多角形

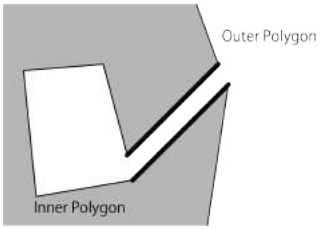
次に、耳リストの先頭から頂点 2 を取り出…そうと思いましたが、残された多角形の頂点が 3 つしか無いのでこのまま三角形にして三角形分割は終了です。最終的な三角形分割の結果は次のとおりです。



▲図 5.11　三角形分割の結果

5.4　穴空き多角形の三角形分割

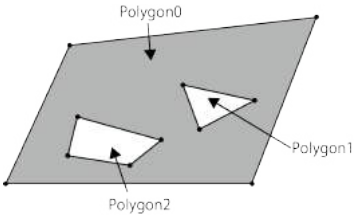
次に、穴のある多角形の三角形分割について解説します。もともと「耳刈り取り法」は穴のある多角形には適用できませんが、図のように外側の多角形に切れ込みをいれて内側の多角形と繋げてしまえば、内側の多角形が外側の多角形の一部になり、耳刈り取り法が適用できるようになります。この方法は複数の穴がある多角形でも可能です。



▲図 5.12　内外多角形の結合（図はかなり大げさな表現）

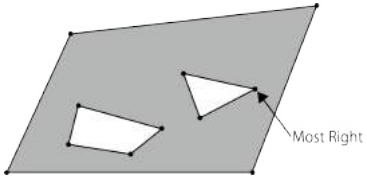
5.4.1　外側の多角形と内側の多角形の結合の流れ

前提として、外側の多角形と内側の多角形の頂点の順序は逆にする必要があります。たとえば、外側の多角形が時計回りに頂点が並んでいる場合、内側の多角形は反時計回りに並んでいる必要があります。次の多角形を例に結合の流れを説明します。



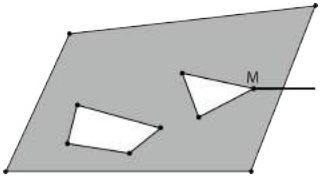
▲図 5.13　穴のある多角形

1. 複数の穴（内側の多角形）がある場合、内側の多角形の中で最も X 座標が大きい（右側にある）多角形とその頂点を探します。



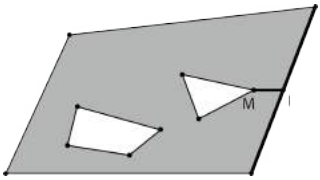
▲図 5.14　最も X 座標が大きい頂点

2. 最も X 座標の大きい頂点を M とします。M から右にまっすぐ線を引きます。



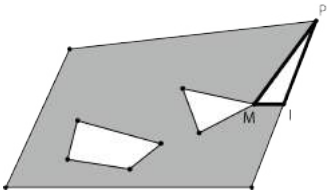
▲図 5.15　頂点 M から右に線を引く

3. 頂点 M から右に伸ばした線と交差する外側の多角形の辺と交点 I を探します。複数の辺と交差する場合、最も頂点 M に近い交点の辺を選択します。



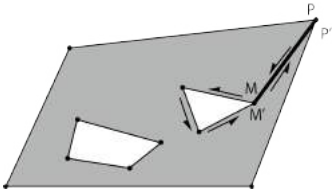
▲図 5.16　頂点 M と交点 I

4. 交差する辺の頂点のうち、最も X 座標が大きい頂点 P を選択します。頂点 M,I,P を結ぶ三角形の中に他の頂点が含まれないかチェックします。



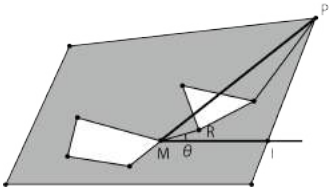
▲図 5.17　三角形 M,I,P

5. 三角形 M,I,P に他の頂点が含まれない場合は、分割が可能なので、外側の多角形の頂点 P から内側の多角形の頂点 M を繋げ、内側の多角形を反時計回りに一周します。再度 M から、外側の多角形の頂点 P に繋げる時、頂点 M と頂点 P を複製して別の頂点とします (頂点 M',P')。入る線と出る線を分けることで、見かけ上は線が重なっていますが、頂点の順序としては交差していない、ひとつの単純多角形となります。



▲図 5.18　外側の多角形と内側の多角形を繋げた図

6. 三角形 M,I,P の中に他の頂点 R が含まれていた場合、その頂点 R を選択しますが、複数の頂点が含まれている場合、線分 M,I と線分 M,R のなす角 θ が最も小さい頂点 R を選択して、5 の処理を行います。



▲図 5.19　線分 MI,MR のなす角 θ が最も小さい頂点 R

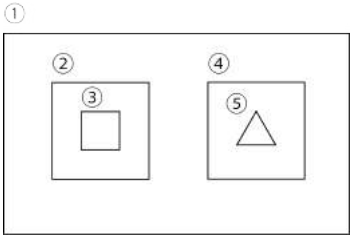
7. 1 に戻って他の内側の多角形と結合していきます。

5.5　入れ子構造の多角形の三角形分割

次に、入れ子構造の多角形の三角形分割について説明します。穴のある多角形の結合処理と三角形分割処理は前項で説明したので、ここでは主に多角形の親子関係のツリー構築の手順について説明します。

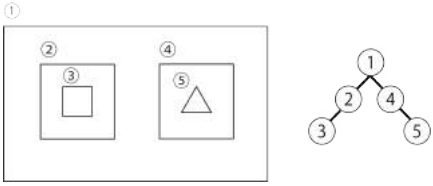
1. 多角形を矩形領域としたときの面積が大きい順にソートします。多角形の最小/最大座標の頂点で作る矩形領域の面積です。
2. 面積が大きい多角形の中に、他の多角形が全頂点が含まれているかを再帰的に判定し、親子関係のツリーを作ります。この時、最上位のルートは空の多角形（いわゆるダミー）として、あとの多角形の結合処理には使いません。何故、最上位をダミーにするのかというと、複数の全く被らない多角形の集合が渡された場合に、最上位が 1 つにならないからです。ダミーの最上位の下に全く重ならない複数の多角形をぶら下げることで、処理が単純にできます。また、階層が偶数階の時は、内側の多角形になるので、該当多角形の頂点配列を反時計回りに並べ替えます。
3. 親子関係ツリーができたら、上から 1 つ多角形を取り出します。それが外側の多角形になります。
4. 外側の多角形の 1 階層下 (子) の多角形を取り出します。そして、内側の多角形として外側の多角形と結合して、三角形分割を行います。子がない場合はそのまま三角形分割します。
5. 3 に戻って結合と分割を繰り返します。4 で外側の多角形と内側の多角形群が 1 つの組み合わせとして処理されるので、次に取り出す三角形はまた外側の多角形になります。

次の多角形の集合を例として説明します。



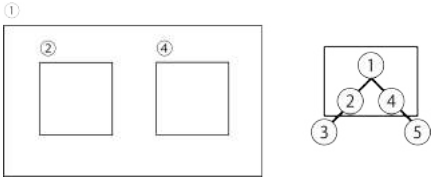
▲図 5.20　入れ子構造の多角形

多角形の親子関係を作ると次のようなツリーになりました。



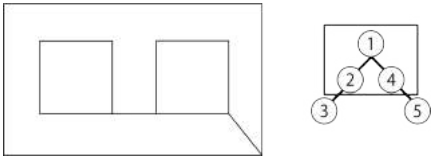
▲図 5.21　左：入れ子構造の多角形 右：親子関係

ツリーの最上位（ダミーは除く）の多角形 1 と、その子の多角形 2,4 を取り出します。



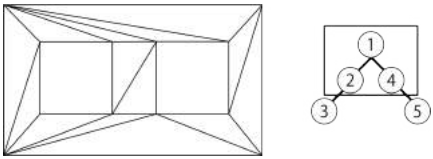
▲図 5.22　多角形 1,2,4 を取り出す

多角形 1,2,4 を右から順に結合していきます。



▲図 5.23　多角形 1,2,4 を結合

結合した多角形を三角形分割します。



▲図 5.24　結合した多角形を三角形分割

三角形分割した多角形を取り除きます。親子関係ツリーの残りは 3 と 5 です。まず、3 から取り出します。

▲図 5.25　多角形 3

3 には子は無いのでそのまま三角形分割します。

▲図 5.26　多角形 3 を三角形分割

三角形分割した多角形を取り除きます。親子関係ツリーの残りは 5 だけです。5 は三角形で子がないので、そのまま終了します。これで入れ子構造の多角形の三角形分割は完了です。

▲図 5.27　最後の多角形 5

5.6　実装

今まで説明した 3 つのアルゴリズムを全て実装したサンプルのソースコードの説明に移ります。

5.6.1　Polygon クラス

まず、多角形の頂点配列を管理する Polygon クラスを定義します。Polygon クラスは、頂点座標の配列やループの方向などの情報の保持、多角形の中に多角形が入っているかの判定を行います。

▼ Polygon.cs

```
public class Polygon
{
    // ループの方向
    public enum LoopType
    {
        CW,     // 時計回り
        CCW,    // 反時計回り
        ERR,    // 不定（向きがない）
    }

    public Vector3[] vertices;  // 頂点配列
    public LoopType loopType;   // ループの方向
```

```
    //～省略～
}
```

5.6.2 Triangulation クラス

実際に多角形の三角形分割を行う Triangulation クラスです。Triangulation クラスの Triangulate 関数がメインです。

三角形分割に使うデータ構造

▼ Triangulation.cs のデータ構造定義

```
// 頂点配列
List<Vector3> vertices = new List<Vector3>();

// 頂点番号のリスト（末尾と先頭がつながってることにする）
LinkedList<int> indices = new LinkedList<int>();

// 耳頂点リスト
List<int> earTipList = new List<int>();
```

処理対象の多角形の頂点座標の配列を格納する vertices、多角形の頂点の番号（インデックス）を格納する indices、耳を格納する earTipList を定義しています。indices は前後の頂点を参照する必要があるため、双方向リストの性質がある LinkedList を使っています。

階層構造を作る

まず、外から多角形を構成する頂点配列を渡されたら、それを Polygon クラスとしてリストに格納します。

▼ Polygon リスト

```
// 多角形リスト
List<Polygon> polygonList = new List<Polygon>();

public void AddPolygon(Polygon polygon)
{
    polygonList.Add(polygon);
}
```

Triangulate 関数の冒頭で、多角形データを追加した Polygon リストを矩形領域の面積の大きい順にソートします。

▼ Polygon リストのソート部分

```
// 多角形リストの矩形領域の面積の大きい順にソート
polygonList.Sort((a, b) => Mathf.FloorToInt(
  (b.rect.width * b.rect.height) - (a.rect.width * a.rect.height)
  ));
```

次に、ソートした Polygon リストをツリー構造を作る TreeNode クラスに詰め込んでいきます。

▼ Polygon リストを TreeNode に詰める部分

```
// ルート作成（空っぽ）
polygonTree = new TreeNode<Polygon>();

// 多角形の階層構造を作る
foreach (Polygon polygon in polygonList)
{
   TreeNode<Polygon> tree = polygonTree;

   CheckInPolygonTree(tree, polygon, 1);
}
```

TreeNode は以下のようになっています。よくあるツリー構造だと思いますが、空の最上位ノードのために、中身が存在するかのフラグ isValue を定義しています。

▼ TreeNode.cs

```
public class TreeNode<T>
{
    public TreeNode<T> parent = null;
    public List<TreeNode<T>> children = new List<TreeNode<T>>();

    public T Value;
    public bool isValue = false;

    public TreeNode(T val)
    {
        Value = val;
        isValue = true;
    }

    public TreeNode()
    {
        isValue = false;
    }

    public void AddChild(T val)
    {
        AddChild(new TreeNode<T>(val));
    }

    public void AddChild(TreeNode<T> tree)
    {
        children.Add(tree);
        tree.parent = this;
```

```
    }

    public void RemoveChild(TreeNode<T> tree)
    {
        if (children.Contains(tree))
        {
            children.Remove(tree);
            tree.parent = null;
        }
    }
}
```

Triangulation.cs に戻り、多角形の階層構造を作る CheckInPolygonTree 関数の中身です。自身の多角形の中に渡された多角形が入るか確認し、更に自身の子の中にも入るかを再帰的に判定しています。自身には含まれるが、子には含まれない、または子が存在しない場合に、渡された多角形を自身の子にします。

▼ CheckInPolygonTree 関数

```
bool CheckInPolygonTree(TreeNode<Polygon> tree, Polygon polygon, int lv)
{
    // 自身に多角形が存在するか？
    bool isInChild = false;
    if (tree.isValue)
    {
        if (tree.Value.IsPointInPolygon(polygon))
        {
            // 自身に含まれる場合、子にも含まれるか検索
            for(int i = 0; i < tree.children.Count; i++)
            {
                isInChild |= CheckInPolygonTree(
                    tree.children[i], polygon, lv + 1);
            }

            // 子に含まれない場合は自身の子にする
            if (!isInChild)
            {
                // 必要であれば頂点の順番を反転する
                // 偶数ネストの時は Inner なので CW
                // 奇数ネストの時は Outer なので CCW
                if (
                    ((lv % 2 == 0) &&
                     (polygon.loopType == Polygon.LoopType.CW)) ||
                    ((lv % 2 == 1) &&
                     (polygon.loopType == Polygon.LoopType.CCW))
                    )
                {
                    polygon.ReverseIndices();
                }

                tree.children.Add(new TreeNode<Polygon>(polygon));
                return true;
            }
        }
        else
```

```
        {
            // 含まれない
            return false;
        }
    }
    else
    {
        // 自身に値がない場合、子の方だけ検索
        for (int i = 0; i < tree.children.Count; i++)
        {
            isInChild |= CheckInPolygonTree(
                tree.children[i], polygon, lv + 1);
        }

        // 子に含まれない場合は自身の子にする
        if (!isInChild)
        {
            // 必要であれば頂点の順番を反転する
            // 偶数ネストの時は Inner なので CW
            // 奇数ネストの時は Outer なので CCW
            if (
                ((lv % 2 == 0) &&
                 (polygon.loopType == Polygon.LoopType.CW)) ||
                ((lv % 2 == 1) &&
                 (polygon.loopType == Polygon.LoopType.CCW))
                )
            {
                polygon.ReverseIndices();
            }
            tree.children.Add(new TreeNode<Polygon>(polygon));
            return true;
        }
    }

    return isInChild;
}
```

穴のある多角形の処理（内外多角形の結合）

複数の内側の多角形がある場合、内側の多角形の中で最も X 座標が大きい頂点とその多角形を選択します。その時、判定用に X 座標と頂点番号、多角形の番号の情報をまとめておくクラスを定義します。

▼ XMaxData 構造体

```
/// <summary>
/// X 座標最大値と多角形の情報
/// </summary>
struct XMaxData
{
    public float xmax;  // x 座標最大値
    public int no;      // 多角形の番号
    public int index;   // xmax の頂点番号

    public XMaxData(float x, int n, int ind)
    {
        xmax = x;
        no = n;
        index = ind;
    }
}
```

次に、実際の結合処理ですが、複数の多角形を X 座標が大きい順にソートする処理と、結合する処理の 2 つに分かれています。まずは、複数の多角形を X 座標が大きい順にソートする処理です。

▼ CombineOuterAndInners 関数

```
Vector3[] CombineOuterAndInners(Vector3[] outer, List<Polygon> inners)
{
    List<XMaxData> pairs = new List<XMaxData>();

    // 内側の多角形の中で最も X 座標が大きい頂点を持つものを探す
    for (int i = 0; i < inners.Count; i++)
    {
        float xmax = inners[i].vertices[0].x;
        int len = inners[i].vertices.Length;
        int xmaxIndex = 0;
        for (int j = 1; j < len; j++)
        {
            float x = inners[i].vertices[j].x;
            if(x > xmax)
            {
                xmax = x;
                xmaxIndex = j;
            }
        }
        pairs.Add(new XMaxData(xmax, i, xmaxIndex));
    }

    // 右順 (xmax が大きい順) にソート
    pairs.Sort((a, b) => Mathf.FloorToInt(b.xmax - a.xmax));

    // 右から順に結合
    for (int i = 0; i < pairs.Count; i++)
    {
        outer = CombinePolygon(outer, inners[pairs[i].no], pairs[i].index);
    }
```

```
    return outer;
}
```

次に結合処理の部分です。CombinePolygon 関数の中で、内側の多角形の最も X 座標が大きい頂点 M から横に線を引き、その線と交差する外側の多角形の線分を探します。

▼ CombinePolygon 関数の序盤

```
Vector3[] CombinePolygon(Vector3[] outer, Polygon inner, int xmaxIndex)
{
    Vector3 M = inner.vertices[xmaxIndex];

    // 交点を探す
    Vector3 intersectionPoint = Vector3.zero;
    int index0 = 0;
    int index1 = 0;

    if (GeomUtil.GetIntersectionPoint(M,
      new Vector3(maxX + 0.1f, M.y, M.z),
      outer, ref intersectionPoint,
      ref index0, ref index1))
    {
      ～省略～
```

線分 M,I と外側の多角形の線分の交点を探す関数、GeometryUtil.GetIntersectionPoint は以下のようになっています。ポイントは、外側の多角形は時計回りなので、交差する線分の始点が線分 M,I より上で終点が下になるものだけを探すようにしている点です。そうすることで、すでに結合した内外多角形で、外側の多角形から内側の多角形に接続した線分を選択してしまうと、頂点の順序がおかしくなってしまうのを防ぎます。

▼ GetIntersectionPoint 関数

```
public static bool GetIntersectionPoint(Vector3 start, Vector3 end,
                                        Vector3[] vertices,
                                        ref Vector3 intersectionP,
                                        ref int index0, ref int index1)
{
    float distanceMin = float.MaxValue;
    bool isHit = false;

    for(int i = 0; i < vertices.Length; i++)
    {
        int index = i;
        int next = (i + 1) % vertices.Length;        // 次の頂点

        Vector3 iP = Vector3.zero;
        Vector3 vstart = vertices[index];
        Vector3 vend = vertices[next];
```

```
        // 交差する多角形の線分の始点が線分 M,I 以上にいること
        Vector3 diff0 = vstart - start;
        if (diff0.y < 0f)
        {
            continue;
        }

        // 交差する多角形の線分の終点が線分 M,I 以下にいること
        Vector3 diff1 = vend - start;
        if (diff1.y > 0f)
        {
            continue;
        }

        if (IsIntersectLine(start, end, vstart, vend, ref iP))
        {
            float distance = Vector3.Distance(start, iP);

            if (distanceMin >= distance)
            {
                distanceMin = distance;
                index0 = index;
                index1 = next;
                intersectionP = iP;
                isHit = true;
            }
        }

    }

    return isHit;
}
```

交点を見つけた後は、交差した線分の一番 X 座標が大きい頂点、頂点 M、交点 I から作る三角形に他の頂点が含まれてないか確認します。三角形の中に頂点が含まれているかの判定には、二次元の外積を使って頂点が三角形の線分のどちら側にいるかを出しています。頂点がすべての線の右側にいれば三角形の中に含まれていることになります。

▼ GeometryUtil.cs の IsTriangle 関数と CheckLine 関数

```
/// <summary>
/// 線と頂点の位置関係を返す
/// </summary>
/// <param name="o"></param>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <returns> +1 : 線の右 -1 : 線の左 0 : 線上</returns>
public static int CheckLine(Vector3 o, Vector3 p1, Vector3 p2)
{
    double x0 = o.x - p1.x;
    double y0 = o.y - p1.y;
    double x1 = p2.x - p1.x;
    double y1 = p2.y - p1.y;
```

```
    double x0y1 = x0 * y1;
    double x1y0 = x1 * y0;
    double det = x0y1 - x1y0;

    return (det > 0D ? +1 : (det < 0D ? -1 : 0));
}

/// <summary>
/// 三角形（時計回り）と点の内外判定
/// </summary>
/// <param name="o"></param>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <param name="p3"></param>
/// <returns> +1 : 外側 -1 : 内側 0 : 線上</returns>
public static int IsInTriangle(Vector3 o,
                               Vector3 p1,
                               Vector3 p2,
                               Vector3 p3)
{
    int sign1 = CheckLine(o, p2, p3);
    if (sign1 < 0)
    {
        return +1;
    }

    int sign2 = CheckLine(o, p3, p1);
    if (sign2 < 0)
    {
        return +1;
    }

    int sign3 = CheckLine(o, p1, p2);
    if (sign3 < 0)
    {
        return +1;
    }

    return (((sign1 != 0) && (sign2 != 0) && (sign3 != 0)) ? -1 : 0);
}
```

さて、CombinePolygon の続きです。交点を見つけた後に、三角形の中に他の頂点が入っているか判定しますが、外積を使って内外判定をする関係上、三角形の接続の向きが時計回りになるようにしています。

▼ CombinePolygon 関数の中盤 1

```
if (GeomUtil.GetIntersectionPoint(M,
  new Vector3(maxX + 0.1f, M.y, M.z), outer,
  ref intersectionPoint, ref index0, ref index1))
{
    // 交点発見

    // 交差した線分の一番右の頂点を取得
    int pindex;
```

```
    Vector3[] triangle = new Vector3[3];
    if (outer[index0].x > outer[index1].x)
    {
        pindex = index0;
        // 選択した線分の頂点によって三角形が逆向きになってしまうので、時計回りになる
ように調整する
        triangle[0] = M;
        triangle[1] = outer[pindex];
        triangle[2] = intersectionPoint;
    }
    else
    {
        pindex = index1;
        triangle[0] = M;
        triangle[1] = intersectionPoint;
        triangle[2] = outer[pindex];
    }
```

交点 I と線分の最も X 座標が大きい頂点が同一だった場合、頂点 M から見て遮るものが無いということなので、三角形の中に他の頂点が含まれているかのチェックは行いません。同一でなかった場合は、他の頂点が含まれているかのチェックを行いますが、三角形に含まれる頂点は、内側に凹んだ頂点なので、その条件を満たしつつ内包判定を行います。三角形に複数の頂点が含まれていた場合、線分 M,I と線分 M, 該当頂点のなす角が最も小さい頂点を選択して、finalIndex に格納します。

▼ CombinePolygon 関数の中盤 2

```
Vector3 P = outer[pindex];

int finalIndex = pindex;

// 交点と P が同じだったら遮るものがないので三角形チェックしない
if((Vector3.Distance(intersectionPoint, P) > float.Epsilon))
{
    float angleMin = 361f;

    for(int i = 0; i < outer.Length; i++)
    {

        // 凸頂点/反射頂点チェック
        int prevIndex = (i == 0) ? outer.Length - 1 : i - 1; // 一つ前の頂点
        int nextIndex = (i + 1) % outer.Length;        // 次の頂点
        int nowIndex = i;

        if (nowIndex == pindex) continue;

        Vector3 outerP = outer[nowIndex];

        if (outerP.x < M.x) continue;

        // 分割時に複製した同一座標だったら無視
        if (Vector3.Distance(outerP, P) <= float.Epsilon) continue;
```

```
        Vector3 prevVertex = outer[prevIndex];
        Vector3 nextVertex = outer[nextIndex];
        Vector3 nowVertex = outer[nowIndex];

        // 反射頂点か？
        bool isReflex = !GeomUtil.IsAngleLessPI(nowVertex,
                                                prevVertex,
                                                nextVertex);

        // 三角形の中に「反射頂点」が含まれているか？
        if ((GeomUtil.IsInTriangle(outerP,
                                   triangle[0],
                                   triangle[1],
                                   triangle[2]) <= 0)&&(isReflex))
        {
            // 三角形の中に頂点が含まれてるので不可視

            // M,I と M,outerP の線分のなす角度を求める（一番角度が浅い頂点を選択す
る）
            float angle = Vector3.Angle(intersectionPoint - M, outerP - M);
            if (angle < angleMin)
            {
                angleMin = angle;
                finalIndex = nowIndex;
            }
        }
    }
}
```

結合させる頂点（finalIndex）を探しだしたら、内外多角形の頂点配列をつなぎ合わせます。

1. 新しい頂点配列のリストを作成します。
2. リストに外側の多角形の結合させる頂点（finalIndex）までを追加します。
3. 結合する内側の多角形の頂点配列を、頂点 M から順番に一周するようにリストに追加します。
4. 再び内側の多角形から外側の多角形につなげる為、頂点 M と結合させる頂点（finalIndex）を複製して、リストに追加します。
5. 残りの外側の多角形の頂点をリストに追加して完了です。

▼ CombinePolygon の後半

```
Vector3 FinalP = outer[finalIndex];

// 結合（新しい多角形を作成）
List<Vector3> newOuterVertices = new List<Vector3>();

// outer を分割する Index まで追加
for (int i = 0; i <= finalIndex; i++)
{
    newOuterVertices.Add(outer[i]);
```

```
}

// inner をすべて追加
for (int i = xmaxIndex; i < inner.vertices.Length; i++)
{
    newOuterVertices.Add(inner.vertices[i]);
}
for (int i = 0; i < xmaxIndex; i++)
{
    newOuterVertices.Add(inner.vertices[i]);
}

// 分割するために頂点を 2 つ増やす
newOuterVertices.Add(M);
newOuterVertices.Add(FinalP);

// 残りの outer の index を追加
for (int i = finalIndex + 1; i < outer.Length; i++)
{
    newOuterVertices.Add(outer[i]);
}

outer = newOuterVertices.ToArray();
```

三角形分割

内外多角形が一つの多角形になったら、いよいよ三角形分割です。まず、頂点のインデックス配列の初期化や、耳リストの作成を行います。

▼ InitializeVertices 関数

```
/// <summary>
/// 初期化
/// </summary>
void InitializeVertices(Vector3[] points)
{
    vertices.Clear();
    indices.Clear();
    earTipList.Clear();

    // インデックス配列の作成
    resultTriangulationOffset = resultVertices.Count;
    for (int i = 0; i < points.Length; i++)
    {
        Vector3 nowVertex = points[i];
        vertices.Add(nowVertex);

        indices.AddLast(i);

        resultVertices.Add(nowVertex);
    }

    // 凸三角形と耳の検索
    LinkedListNode<int> node = indices.First;
    while (node != null)
```

```
    {
        CheckVertex(node);
        node = node.Next;
    }
}
```

頂点が耳かどうか判定する CheckVertex 関数は次のようになっています。

▼ CheckVertex 関数

```
void CheckVertex(LinkedListNode<int> node)
{
    // 凸頂点/反射頂点チェック
    int prevIndex = (node.Previous == null) ?
                    indices.Last.Value :
                    node.Previous.Value; // 一つ前の頂点
    int nextIndex = (node.Next == null) ?
                    indices.First.Value :
                    node.Next.Value;        // 次の頂点
    int nowIndex = node.Value;

    Vector3 prevVertex = vertices[prevIndex];
    Vector3 nextVertex = vertices[nextIndex];
    Vector3 nowVertex = vertices[nowIndex];

    bool isEar = false;

    // 内角が 180 度以内か？
    if (GeomUtil.IsAngleLessPI(nowVertex, prevVertex, nextVertex))
    {
        // 耳チェック
        // 180 度以内、三角形の中に他の頂点が含まれない
        isEar = true;
        foreach(int i in indices)
        {
            if ((i == prevIndex) || (i == nowIndex) || (i == nextIndex))
                continue;

            Vector3 p = vertices[i];

            // 分割時に複製した同一座標だったら無視
            if (Vector3.Distance(p, prevVertex) <= float.Epsilon) continue;
            if (Vector3.Distance(p, nowVertex) <= float.Epsilon) continue;
            if (Vector3.Distance(p, nextVertex) <= float.Epsilon) continue;

            if(GeomUtil.IsInTriangle(p,
                                     prevVertex,
                                     nowVertex,
                                     nextVertex) <= 0)
            {
                isEar = false;
                break;
            }

        }
        if (isEar)
        {
```

```
            if (!earTipList.Contains(nowIndex))
            {
                // 耳追加
                earTipList.Add(nowIndex);
            }
        }
        else
        {
            // すでに耳のときに耳ではなくなった場合除外
            if (earTipList.Contains(nowIndex))
            {
                // 耳削除
                earTipList.Remove(nowIndex);
            }
        }

    }

}
```

実際の三角形分割は次の EarClipping 関数の中で行います。前述したとおり、耳リストの先頭から頂点を取り出し、前後の頂点と結んだ三角形を出力。そして、頂点インデックス配列から耳の頂点を削除し、前後の頂点が耳になるか判定するという手順を繰り返します。

▼ EarClipping 関数

```
void EarClipping()
{
    int triangleIndex = 0;

    while (earTipList.Count > 0)
    {
        int nowIndex = earTipList[0];   // 先頭取り出し

        LinkedListNode<int> indexNode = indices.Find(nowIndex);
        if (indexNode != null)
        {
            int prevIndex = (indexNode.Previous == null) ?
                              indices.Last.Value :
                              indexNode.Previous.Value; // 一つ前の頂点
            int nextIndex = (indexNode.Next == null) ?
                              indices.First.Value :
                              indexNode.Next.Value;        // 次の頂点

            Vector3 prevVertex = vertices[prevIndex];
            Vector3 nextVertex = vertices[nextIndex];
            Vector3 nowVertex = vertices[nowIndex];

            // 三角形作成
            triangles.Add(new Triangle(
              prevVertex,
              nowVertex,
              nextVertex, "(" + triangleIndex + ")"));

            resultTriangulation.Add(resultTriangulationOffset + prevIndex);
            resultTriangulation.Add(resultTriangulationOffset + nowIndex);
            resultTriangulation.Add(resultTriangulationOffset + nextIndex);

            triangleIndex++;

            if (indices.Count == 3)
            {
                // 最後の三角形なので終了
                break;
            }

            // 耳の頂点削除
            earTipList.RemoveAt(0);     // 先頭削除
            indices.Remove(nowIndex);

            // 前後の頂点のチェック
            int[] bothlist = { prevIndex, nextIndex };
            for (int i = 0; i < bothlist.Length; i++)
            {
                LinkedListNode<int> node = indices.Find(bothlist[i]);
```

```
                CheckVertex(node);
            }
        }
        else
        {
            Debug.LogError("index now found");
            break;
        }
    }

    // UV 計算
    for (int i = 0; i < vertices.Count; i++)
    {
        Vector2 uv2 = CalcUV(vertices[i], uvRect);
        resultUVs.Add(uv2);
    }
}
```

メッシュ生成

三角形分割した結果を Mesh にします。EarClipping 関数の中で、必要な頂点配列とインデックス配列（resultVertices と resultTriangulation）を用意して、Mesh に流し込んでいます。

▼ MakeMesh 関数

```
void MakeMesh()
{
    mesh = new Mesh();
    mesh.vertices = resultVertices.ToArray();
    mesh.SetIndices(resultTriangulation.ToArray(),
                    MeshTopology.Triangles, 0);
    mesh.RecalculateNormals();
    mesh.SetUVs(0, resultUVs);

    mesh.RecalculateBounds();

    MeshFilter filter = GetComponent<MeshFilter>();
    if(filter != null)
    {
        filter.mesh = mesh;
    }
}
```

ついでに、UV 座標もセットしています。UV 座標は、多角形の矩形領域の中で割り当てるようにしてます。

▼ CalcUV

```
Vector2 CalcUV(Vector3 vertex, Rect uvRect)
{
    float u = (vertex.x - uvRect.x) / uvRect.width;
    float v = (vertex.y - uvRect.y) / uvRect.height;
```

```
    return new Vector2(u, v);
}
```

5.7 まとめ

耳刈り取り法による多角形の三角形分割について説明してきました。肝心の使いみちですが、マウスで図形を書いたらリアルタイムにメッシュになる、フォントのアウトラインデータをメッシュにする、等が思いつきますが、そんなに高速なアルゴリズムではないので頂点数が増えてくると速度面で問題が出てくるでしょう（CPU で順番に計算してるし）それでも、複雑な多角形がシンプルなアルゴリズムで三角形に分割できるのは、興味深いと思います。

5.8 参照

- Triangulation by Ear Clipping

https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

- 多角形の三角形分割

https://ja.wikipedia.org/wiki/多角形の三角形分割

第 6 章

Tessellation & Displacement

6.1 はじめに

本章では「Tessellation」と呼ばれる GPU 上でポリゴンを分割する機能と、分割後の頂点群に対して Displacement map によって変位させる方法について解説していきます。
本章のサンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming4`
の「Tessellation」です。

6.1.1 実行環境

- ComputeShader が実行できる、シェーダーモデル 5.0 以上の対応環境
- Tessellation Shader のみの使用であれば、シェーダーモデル 4.6 以上の対応環境
- 執筆時環境、Unity2018.3.9 で動作確認済み

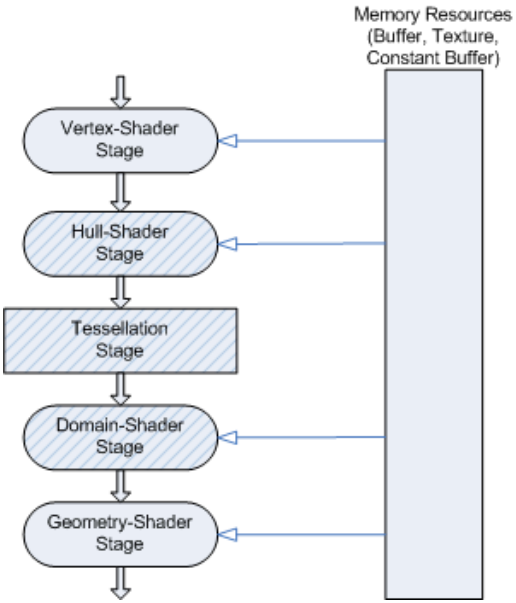
6.2 Tessellation とは

Tessellation（テッセレーション）とは、DirectX、OpenGL、Metal 等のレンダリングパイプラインに標準導入されている、GPU 上でポリゴンを分割する機能です。
通常、頂点や法線・接線・UV 情報等は CPU から GPU に転送されレンダリングパイプラインに流れますが、ハイポリゴンを処理する場合は CPU と GPU 間の転送帯域に負荷が生じ、描画速度のボトルネックとなります。
Tessellation は GPU 上でメッシュを分割する機能を提供しますので、ある程度リダクションされたポリゴンを CPU で処理しつつも、GPU で細分化し、Displacement map ルックアップにより、きめ細やかな変位に戻すといったことが可能になります。

本著では主に Unity での Tessellation 機能について解説を行っていきます。

6.2.1 Tessellation の各ステージ

Tessellation では描画パイプライン上に、「Hull Shader」「Tessellation」「Domain Shader」という 3 つのステージが追加されます。追加されるステージは 3 つですが、プログラマブルステージは「Hull Shader」と「Domain Shader」の 2 つのステージのみになります。



▲図 6.1　Tessellation pipeline 出典：Microsoft

ここで各ステージの詳細を理解し、Hull Shader と Domain Shader の実装を行っていくことも Tessellation に対しての理解を深める為の 1 つの手ではありますが、Unity では非常に便利な Wrapper が Surface Shader に組み込み可能な形で用意されています。
まずはこの Surface Shader を元に、Tessellation と Displacement を行っていきましょう。

6.3 Surface Shader と Tessellation

Unity が Surface Shader でサポートしている Tessellation について、コメントで注釈をいれつつ解説を行っていきます。

▼ TessellationSurface.Shader

```
Shader "Custom/TessellationDisplacement"
{
    Properties
    {
        _EdgeLength ("Edge length", Range(2,50)) = 15
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "black" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
        _Specular ("Specular", Range(0, 1) ) = 0
        _Gloss ("Gloss", Range(0, 1) ) = 0
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM

        //tessellate:tessEdge としてパッチの分割数と手法を定義する関数を指定
        //vertex:disp として、Displacement を行う関数に disp を指定
        //Wrapper 内部では Domain Shader 内で Call されます
        #pragma surface surf BlinnPhong addshadow fullforwardshadows
            vertex:disp tessellate:tessEdge nolightmap
        #pragma target 4.6
        #include "Tessellation.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        sampler2D _DispTex;
        float _Displacement;
        float _EdgeLength;
        float _Specular;
        float _Gloss;

        //分割数と分割手法を指定する関数です
        //この関数は頂点毎ではなくパッチ毎に Call されます
        //xyz に 3 頂点で構成されたパッチのエッジの分割数を指定し、
        //w にパッチ内部の分割数を指定して返します
        float4 tessEdge (appdata v0, appdata v1, appdata v2)
```

```
        {
            //Tessellation.cginc 内に、3 種類の分割手法を定義した関数があります

            //カメラからの距離に応じた Tessellation
            //UnityDistanceBasedTess

            //Mesh のエッジの長さに応じた Tessellation
            //UnityEdgeLengthBasedTess

            //UnityEdgeLengthBasedTess 関数にカリング機能
            //UnityEdgeLengthBasedTessCull

            return UnityEdgeLengthBasedTessCull(
                v0.vertex, v1.vertex, v2.vertex,
                _EdgeLength, _Displacement * 1.5f
            );
        }

        //Displacement 処理関数に指定した disp 関数です。
        //この関数は Wrapper 内では Tessellator 後の
        //Domain Shader の中で呼ばれています。
        //この関数内で appdata に定義された要素はすべてアクセス可能ですので、
        //Displacement であったりの頂点変調等の処理はここで行います。
        void disp (inout appdata v)
        {
            //ここでは、Displacement map を使った法線方向への頂点変調を行っています
            float d = tex2Dlod(
                _DispTex,
                float4(v.texcoord.xy,0,0)
            ).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input
        {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o)
        {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = _Specular;
            o.Gloss = _Gloss;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

上記の Shader で Surface Shader を利用した Displacement 処理が実現します。非常に安価な実装で大きな恩恵を得ることができます。

6.4 Vertex/Fragment Shader と Tessellation

Vertex/Fragment Shader に各 Tessellation ステージを書く場合の実装です。

6.4.1 Hull Shader Stage

Hull Shader はプログラマブル可能なステージで、Vertex Shader の直後に呼ばれるステージです。ここでは主に「分割方法」と「何分割するか」を定義します。
Hull Shader は「コントロールポイント関数」と「パッチ定数関数」という 2 つの関数から構成されており、これらは GPU によって並列に処理されます。コントロールポイントは分割元の制御点のことで、パッチは分割するトポロジーになります。たとえば 3 角ポリゴン毎にパッチを形成して、そこから Tessellator で分割したい場合は、コントロールポイントが 3 つで、パッチが 1 つとなります。
コントロールポイント関数はコントロールポイント毎に動作し、パッチ定数関数はパッチ毎に動作します。

▼ Tessellation.Shader

```
#pragma hull hull_shader

//hull shader 系の入力として使用する構造体
struct InternalTessInterp_appdata
{
    float4 vertex : INTERNALTESSPOS;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float2 texcoord : TEXCOORD0;
};

//パッチ定数関数で定義して返すテッセレーション係数構造体
struct TessellationFactors
{
    float edge[3] : SV_TessFactor;
    float inside : SV_InsideTessFactor;
};

// hull constant shader (パッチ定数関数)
TessellationFactors hull_const (InputPatch<InternalTessInterp_appdata, 3> v)
{
    TessellationFactors o;
    float4 tf;

    //Surface shader での Tessellation 時のコメントで解説した分割 Utility 関数
    tf = UnityEdgeLengthBasedTessCull(
        v[0].vertex, v[1].vertex, v[2].vertex,
        _EdgeLength, _Displacement * 1.5f
    );

    //エッジの分割数をセット
```

```
    o.edge[0] = tf.x;
    o.edge[1] = tf.y;
    o.edge[2] = tf.z;
    //中央部の分割数をセット
    o.inside = tf.w;
    return o;
}

// hull shader（制御点関数）

//分割プリミティブタイプ tri で 3 角ポリゴン
[UNITY_domain("tri")]
//integer,fractional_odd,fractional_even から分割割合を選択
[UNITY_partitioning("fractional_odd")]
//分割後のトポロジー triangle_cw は時計回り 3 角ポリゴン 反時計回りは triangle_ccw
[UNITY_outputtopology("triangle_cw")]
//パッチ定数関数名を指定
[UNITY_patchconstantfunc("hull_const")]
//出力コントロールポイント。3 角ポリゴンの為 3 つ出力
[UNITY_outputcontrolpoints(3)]
InternalTessInterp_appdata hull_shader (
    InputPatch<InternalTessInterp_appdata,3> v,
    uint id : SV_OutputControlPointID
)
{
    return v[id];
}
```

6.4.2 Tessellation Stage

ここで Hull shader で返したテッセレーション係数（TessellationFactors 構造体）に応じて、パッチに分割処理がかけられます。
Tessellation Stage はプログラマブルではありませんので、関数を記述することはできません。

6.4.3 Domain Shader Stage

Domain Shader は Tessellation Stage の処理結果を元に頂点や法線、接線、UV 等のポジションの反映を行うプログラマブルな Stage です。
SV_DomainLocation というセマンティクスのパラメーターが Domain Shader に入力されますので、このパラメーターを使って座標を反映させていく形になります。
また、Displacement 処理を行いたい場合も Domain Shader に記述しましょう。
Domain Shader の後は、Fragment Shader に処理が流れ最終的な描画処理を行う形になりますが、#pragma で Geometry Shader 関数を指定している場合は、Geometory Shader に流すことも可能です。

▼ Tessellation.Shader

```
#pragma domain domain_shader

struct v2f
{
    UNITY_POSITION(pos);
    float2 uv_MainTex : TEXCOORD0;
    float4 tSpace0 : TEXCOORD1;
    float4 tSpace1 : TEXCOORD2;
    float4 tSpace2 : TEXCOORD3;
};

sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _DispTex;
float _Displacement

v2f vert_process (appdata v)
{
    v2f o;
    UNITY_INITIALIZE_OUTPUT(v2f,o);
    o.pos = UnityObjectToClipPos(v.vertex);
    o.uv_MainTex.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
    float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
    float3 worldNormal = UnityObjectToWorldNormal(v.normal);
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
    fixed tangentSign = v.tangent.w * unity_WorldTransformParams.w;
    fixed3 worldBinormal = cross(worldNormal, worldTangent) * tangentSign;
    o.tSpace0 = float4(
        worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x
    );
    o.tSpace1 = float4(
        worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y
    );
    o.tSpace2 = float4(
        worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z
    );
    return o;
}

void disp (inout appdata v)
{
    float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
    v.vertex.xyz -= v.normal * d;
}

// ドメインシェーダー関数
[UNITY_domain("tri")]
v2f domain_shader (
    TessellationFactors tessFactors,
    const OutputPatch<InternalTessInterp_appdata, 3> vi,
    float3 bary : SV_DomainLocation
)
{
    appdata v;
    UNITY_INITIALIZE_OUTPUT(appdata,v);
    //Tessellation stage で計算された SV_DomainLocation セマンティクスの bary を元
に各座標を設定します
    v.vertex   =
```

```
            vi[0].vertex * bary.x +
            vi[1].vertex * bary.y +
            vi[2].vertex * bary.z;
        v.tangent  =
            vi[0].tangent * bary.x +
            vi[1].tangent * bary.y +
            vi[2].tangent * bary.z;
        v.normal   =
            vi[0].normal * bary.x +
            vi[1].normal * bary.y +
            vi[2].normal * bary.z;
        v.texcoord =
            vi[0].texcoord * bary.x +
            vi[1].texcoord * bary.y +
            vi[2].texcoord * bary.z;

        //Displacement 処理を行う場合はここが最適になります。
        disp (v);

        //最後にフラグメントシェーダーへ渡す直前の処理を記述します
        v2f o = vert_process (v);
        return o;
    }
```

以上が Vertex/Fragment Shader に Tessellation を組み込む際の処理になります。

最後に作例を添付しておきます。この作例は、「Unity Graphics Programming vol.1」に記載している格子法の流体 RenderTexture 出力を Height map とし、Unity に元々入っている Plane メッシュに Tessellation と Displacement 処理を施した物です。

元は頂点数の限られた Plane メッシュですが、破綻すること無く、メッシュが高い粒度で追従していることが確認できるかと思います。

▲図 6.2　流体による Displacement

6.5　まとめ

本章では、「Tessellation」についてご紹介しました。
Tessellation はある程度枯れた技術ではありますが、パフォーマンスの最適化からクリエイティブ用途まで、使い勝手がよいものかと思いますので、ぜひ必要な箇所で使っていただければと思います。

6.6　参考

- https://docs.unity3d.com/jp/current/Manual/SL-SurfaceShaderTessellation.html
- https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-

11-advanced-stages-tessellation

第 7 章

Poisson Disk Sampling

7.1 はじめに

本章では、Poisson Disk Sampling（ポアソンディスクサンプリング：以下、「PDS」を使用）の概要と CPU での実装アルゴリズムの説明をおこないます。

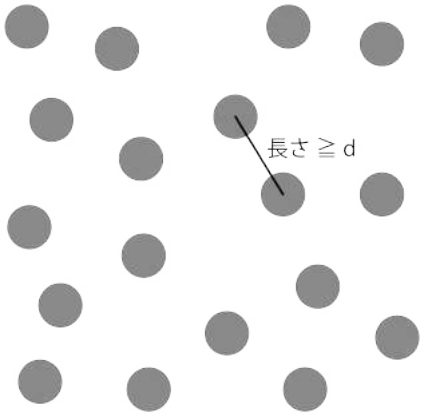
CPU における実装には Fast Poisson Disk Sampling in Arbitary Dimensions（任意次元における高速ポアソンディスクサンプリング：以下、「FPDS」を使用）を採用しています。このアルゴリズムは、ブリティッシュ・コロンビア大学のロバート・ブリッドソン氏によって、2007 年の SIGGRAPH に提出された論文にて提案されました。

7.2 概要

そもそも PDS とは何なのかご存じない方もいらっしゃると思いますので、本節では PDS について説明します。

まず、適当な空間に多数の点をプロットしていきます。次に、0 より大きい距離 d を考えます。このとき、図 7.1 に示すように全ての点がランダムな位置にありながら、全ての点同士が最低でも d 以上離れているような分布を、Poisson-disk 分布といいます。つまり、図 7.1 からランダムに 2 点を選んだとして、どの組み合わせにおいても、必ずその間の距離が d より小さくなることはありません。このような Poisson-disk 分布をサンプリングすること、言い換えれば Poisson-disk 分布を計算によって生成することを、PDS といいます。

この PDS によって、一様性をもつランダムな点群を得ることができます。そのため、アンチエイリアスを始めとするフィルタリング処理におけるサンプリングや、テクスチャ合成処理での合成ピクセルを決定するためのサンプリングなどに利用されています。



▲図 7.1　ランダムな点をプロット

7.3 Fast Poisson Disk Sampling in Arbitary Dimensions （CPU 実装）

FPDS の最大の特徴は、次元に依存しないという点です。これまで提案されてきた手法のほとんどは二次元上での計算を前提としているものであり、効率的に三次元上でのサンプリングをおこなうことができませんでした。そこで、任意の次元において、高速に計算を行うために提案されたのが FPDS です。

FPDS は O(N) で計算することができ、またどの次元においてでも、同一のアルゴリズムに基づきて実装することができます。本節では、FPDS の処理を順を追って説明します。

7.3.1 FPDS の導入

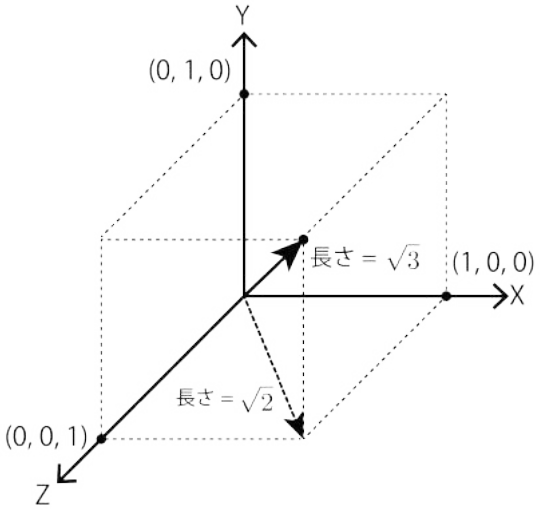
FPDS の計算において、3 つのパラメーターを用います。

- サンプリング空間の範囲 ： Rn
- 点同士の最小距離 ： r
- サンプリング限界数 ： k

これらのパラメーターは、利用者によって自由に入力することができます。以降の説明においても上記のパラメーターを利用します。

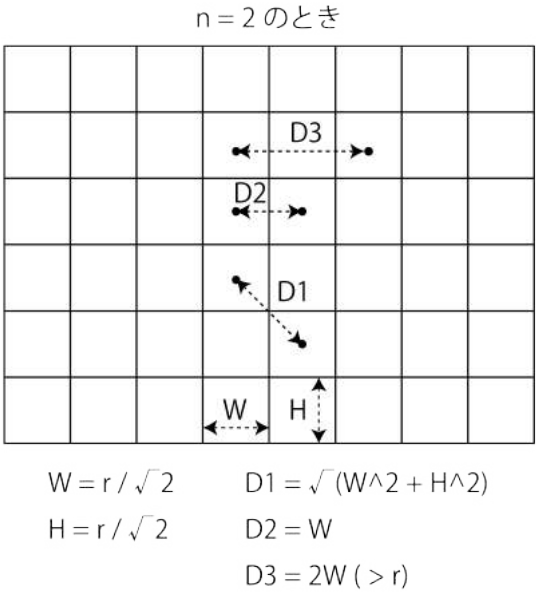
7.3.2 サンプリング空間を Grid に分割する

点同士の距離の計算を高速化するために、サンプリング空間を Grid に分割します。ここで、サンプリング空間の次元数を n として、分割されてできた各分割空間（以下、「セル」とする）のサイズは $\frac{r}{\sqrt{n}}$ とします。図 7.2 に示すように $\sqrt{n}$ は、 n 次元における各軸の長さが 1 のベクトルの絶対値を表しています。



▲図 7.2　n=3 のとき

そして、サンプリングした点はその座標が含まれているセルに所属させます。各セルのサイズは $\frac{r}{\sqrt{n}}$ となっているため、隣り合っているセル同士の中心距離も $\frac{r}{\sqrt{n}}$ となります。つまり $\frac{r}{\sqrt{n}}$ によって空間を分割することで、図 7.3 から分かるように各セルには最大でも一点だけが所属することになります。さらに、あるセルの周辺を近傍探索するときは $\pm n$ だけ周辺セルを探索することで、最小距離 r 以内の距離にある全てのセルを探索できることになります。



▲図 7.3　n=2 における平面分割

また、この Grid は n 次元の配列によって表現することが可能なため、とても実装が簡単です。サンプリングした点を、その座標に対応するセルに所属させることで、近くに存在する他点の探索を容易におこなえます。

▼

```
// 3 次元グリッドを表現する 3 次元配列を取得する
Vector3?[, ,] GetGrid(Vector3 bottomLeftBack, Vector3 topRightForward
    , float min, int iteration)
{
    // サンプリング空間
    var dimension = (topRightForward - bottomLeftBack);
    // 最小距離に√ 3 の逆数をかける（割り算を避けたい）
    var cell = min * InvertRootTwo;

    return new Vector3?[
        Mathf.CeilToInt(dimension.x / cell) + 1,
        Mathf.CeilToInt(dimension.y / cell) + 1,
        Mathf.CeilToInt(dimension.z / cell) + 1
    ];
}
```

```
// 座標に対応するセルの Index を取得する
Vector3Int GetGridIndex(Vector3 point, Settings set)
{
    // 基準点からの距離をセルサイズで割って Index を算出
    return new Vector3Int(
        Mathf.FloorToInt((point.x - set.BottomLeftBack.x) / set.CellSize),
        Mathf.FloorToInt((point.y - set.BottomLeftBack.y) / set.CellSize),
        Mathf.FloorToInt((point.z - set.BottomLeftBack.z) / set.CellSize)
    );
}
```

7.3.3 初期のサンプル点を計算する

計算の起点となる、最初のサンプル点を計算します。この時点では、どの座標にサンプリングをおこなっても距離 r より近い他点は存在しないので、完全にランダムな座標を一点決めます。

この計算したサンプル点の座標を元に、対応するセルに所属させ、さらにアクティブリストとサンプリングリストに追加します。このアクティブリストは、サンプリングをおこなう際に起点となる点を保存しておくリストです。このアクティブリストに保存されている点を基準に、順次サンプリングをおこなっていきます。

▼

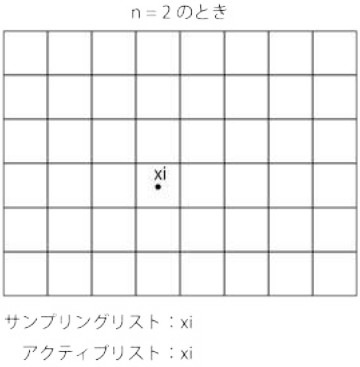
```
// ランダムな座標を一点求める
void GetFirstPoint(Settings set, Bags bags)
{
    var first = new Vector3(
        Random.Range(set.BottomLeftBack.x, set.TopRightForward.x),
        Random.Range(set.BottomLeftBack.y, set.TopRightForward.y),
        Random.Range(set.BottomLeftBack.z, set.TopRightForward.z)
    );
    var index = GetGridIndex(first, set);

    bags.Grid[index.x, index.y, index.z] = first;
    // サンプリングリスト、最終的にはこの List を結果として返す
    bags.SamplePoints.Add(first);
    // アクティブリスト、この List を基準にサンプリングをまわす
    bags.ActivePoints.Add(first);
}
```

7.3.4 アクティブリストから基準となる点を選ぶ

アクティブリストからランダムに Index i を選び、 i に保存されている座標を x_i とします。（当然ですが一番最初のときは「7.3.3 初期のサンプル点を計算する」にて生成した点だけなので、 i は 0 になります。）

この x_i を中心に、付近の座標に対して他点をサンプリングしていくことになりま

す。これを繰り返していくことで、空間全体をサンプリングすることができます。



▲図 7.4　初期のサンプリング点を選択

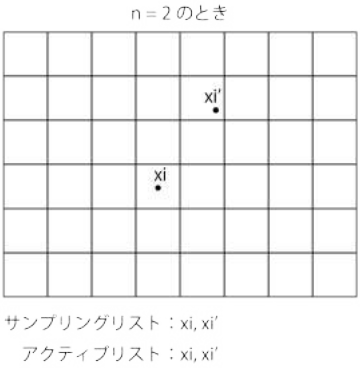
▼

```
// アクティブリストからランダムに点を選ぶ
var index = Random.Range(0, bags.ActivePoints.Count);
var point = bags.ActivePoints[index];
```

7.3.5　サンプリングする

x_i を中心として半径 r 以上、 $2r$ 以下の n 次元球面（2 次元なら円、3 次元なら球になります）内で、ランダムな座標 x_i' を計算します。次に、計算した x_i' の周辺に距離が r より近い他点が存在するかを調べます。

ここで、全ての他点について距離計算をおこなうのは、他点の数が多いほど負荷の高い処理となります。そこで、この問題を解決するために「7.3.2 サンプリング空間を Grid に分割する」にて生成した Grid を用い、 x_i' が所属するセルの周辺セルだけを調べることで、計算量を削減します。周辺セルに他点が存在するときは x_i' を破棄し、他点が存在しなかったときは x_i' を対応するセルに所属させ、アクティブリストとサンプリングリストに追加します。



▲図 7.5　初期のサンプリング点を基準に他点をサンプリング

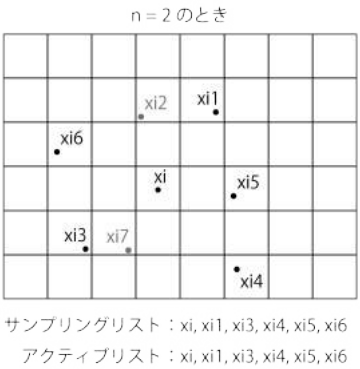
▼

```
// point 座標を元に、次のサンプリング点を求める
private static bool GetNextPoint(Vector3 point, Settings set, Bags bags)
{
    // point 座標を中心に r ~ 2r の範囲にあるランダムな点を一点求める
    var p = point +
        GetRandPosInSphere(set.MinimumDistance, 2f * set.MinimumDistance);

    // サンプリング空間の範囲外だった場合はサンプリング失敗扱いにする
    if(set.Dimension.Contains(p) == false) { return false; }

    var minimum = set.MinimumDistance * set.MinimumDistance;
    var index = GetGridIndex(p, set);
    var drop = false;

    // 探索する Grid の範囲を計算
    var around = 3;
    var fieldMin = new Vector3Int(
        Mathf.Max(0, index.x - around), Mathf.Max(0, index.y - around),
        Mathf.Max(0, index.z - around)
    );
    var fieldMax = new Vector3Int(
        Mathf.Min(set.GridWidth, index.x + around),
        Mathf.Min(set.GridHeight, index.y + around),
        Mathf.Min(set.GridDepth, index.z + around)
    );

    // 周辺の Grid に他点が存在しているかを確認
    for(var i = fieldMin.x; i <= fieldMax.x && drop == false; i++)
    {
        for(var j = fieldMin.y; j <= fieldMax.y && drop == false; j++)
        {
```

```
            for(var k = fieldMin.z; k <= fieldMax.z && drop == false; k++)
            {
                var q = bags.Grid[i, j, k];
                if(q.HasValue && (q.Value - p).sqrMagnitude <= minimum)
                {
                    drop = true;
                }
            }
        }
    }

    if(drop == true) { return false; }

    // 周辺に他点が存在していないので採用
    bags.SamplePoints.Add(p);
    bags.ActivePoints.Add(p);
    bags.Grid[index.x, index.y, index.z] = p;
    return true;
}
```

7.3.6 サンプリングを繰り返す

x_i を中心に、「7.3.5 サンプリングする」にて一点だけ他点をサンプリングしましたが、これを k 回繰り返します。もし x_i について k 回繰り返して、一点もサンプリングすることができなかった場合は、 x_i をアクティブリストから削除します。



▲図 7.6　$k = 7$ のとき

そして k の繰り返しが終わったら、また「7.3.4 アクティブリストから基準となる点を選ぶ」に戻ります。これをアクティブリストが 0 になるまで繰り返すことで、空

間全体をサンプリングすることができます。

▼

```
// サンプリングを繰り返す
public static List<Vector3> Sampling(Vector3 bottomLeft, Vector3 topRight,
    float minimumDistance, int iterationPerPoint)
{
    var settings = GetSettings(
        bottomLeft,
        topRight,
        minimumDistance,
        iterationPerPoint <= 0 ?
            DefaultIterationPerPoint : iterationPerPoint
    );
    var bags = new Bags()
    {
        Grid = new Vector3?[
            settings.GridWidth + 1,
            settings.GridHeight + 1,
            settings.GridDepth + 1
        ],
        SamplePoints = new List<Vector3>(),
        ActivePoints = new List<Vector3>()
    };
    GetFirstPoint(settings, bags);

    do
    {
        var index = Random.Range(0, bags.ActivePoints.Count);
        var point = bags.ActivePoints[index];

        var found = false;
        for(var k = 0; k < settings.IterationPerPoint; k++)
        {
            found = found | GetNextPoint(point, settings, bags);
        }

        if(found == false) { bags.ActivePoints.RemoveAt(index); }
    }
    while(bags.ActivePoints.Count > 0);

    return bags.SamplePoints;
}
```

つまり、簡単に全体の流れを説明すると

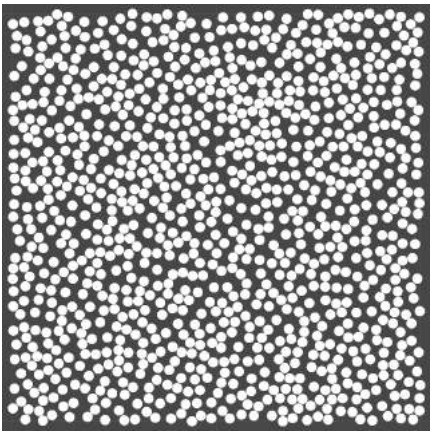
- 空間を決めてグリッドに分割する
- 適当に初期点を決める
- 初期点の周辺にサンプリングする
- サンプリングした点についても、同様に周辺をサンプリングする
- すべての点について、周辺のサンプリングができたら終了

ということになります。本アルゴリズムでは並列化の提案などはされていないため、サンプリング空間 Rn が広かったり、最小距離 r が小さかったりするとそれなり

の計算時間が必要ですが、任意次元において簡単にサンプリングすることができるというのは、非常に魅力的な利点です。

7.4 まとめ

ここまでの処理によって図 7.7 のようにサンプリング結果を得ることができます。この画像は、サンプリングされた座標に GeometryShader で円を生成し、配置した物です。円同士が重なることがなく、範囲いっぱいに埋め尽くされているのが分かります。



▲図 7.7　サンプリング結果の可視化

冒頭でも少し述べましたが、ポアソンディスクサンプリングはアンチエイリアスや画像合成を始めとして、Blur などのイメージエフェクトからオブジェクトの等間隔配置まで、幅広い場所で活用されています。これ単体で明確な見た目の成果を得られたりする訳ではありませんが、私たちが普段目にしているハイクオリティなビジュアルの裏では頻繁に利用されています。ビジュアルプログラミングをおこなっていく上で、知っておいて損のないアルゴリズムのひとつといえるでしょう。

7.5 参考

- Fast Poisson Disk Sampling in Arbitrary Dimensions - `https://www.cct.lsu.edu/~fharhad/ganbatte/siggraph2007/CD2/content/sketches/0250.pdf`

著者紹介

第 1 章 GPU-Based Space Colonization Algorithm

- 中村将達 / @mattatz

インスタレーション、サイネージ、Web（フロントエンド・バックエンド）、スマートフォンアプリ等をつくるプログラマー。映像表現やデザインツール開発に興味があります。

- https://twitter.com/mattatz
- https://github.com/mattatz
- http://mattatz.org/

第 2 章 Limit sets of Kleinian groups - 福永秀和 / @fuqunaga

元ゲーム開発者、インタラクティブアートを作ってるプログラマ。そこそこややこしい仕組みやライブラリの設計開発が好き。最近良く寝てる。

- https://twitter.com/fuqunaga
- https://github.com/fuqunaga
- https://fuquna.ga

第 3 章 GPU-Based Cloth Simulation - 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 StarGlow - @XJINE

ついていくのも必至なレベルでボロボロになりながらどうにか生きています。シェーダ入門用の「UnityShaderProgramming」もよろしくお願いいたします。

- https://twitter.com/XJINE
- https://github.com/XJINE
- http://neareal.com/

第 5 章 Triangulation by Ear Clipping - @kaiware007

雰囲気でやるインタラクティブ・エンジニア。ジェネ系の動画を Twitter によく上げている。たまに VJ をやる。最近は VR に興味あり。

- https://twitter.com/kaiware007
- https://github.com/kaiware007
- https://kaiware007.github.io/

第 6 章 Tessellation & Displacement - 迫田吉昭 / @sakope

元ゲーム開発会社テクニカルアーティスト。アート・デザイン・音楽が好きで、インタラクティブアートに転向。趣味はサンプラー・シンセ・楽器・レコード・機材いじり。Twitter はじめました。

- https://twitter.com/sakope
- https://github.com/sakope

第 7 章 Poisson Disk Sampling - @a3geek

インタラクションエンジニア。CG によるシミュレーションの可視化に興味があり、特に正確に可視化するのではなく、より人の感情を揺さぶる可視化がしてみたい。作ることも好きだが、それ以上に知ることの方が楽しいと感じるタイプ。好きな学校の教室は図工室か図書室。

- https://twitter.com/a3geek
- https://github.com/a3geek

Unity Graphics Programming vol.4

2019 年 4 月 14 日　技術書典 6 版 v1.0.0
著　者　IndieVisualLab
編　集　IndieVisualLab
発行所　IndieVisualLab

IndieVisualLab

a3geek
fuqunaga
irishoak
kaiware007
mattatz
sakope
XJINE

https://indievisuallab.github.io/