Unity Graphics Programming

Unityグラフィックスプログラミング vol.4

IndieVisualLab

a3geek
fuqunaga
irishoak
kaiware007
mattatz
sakope
XJINE

https://indievisuallab.github.io/

Unity Graphics Programming vol.4

IndieVisualLab

Preface

This book is the fourth volume of the "Unity Graphics Programming" series, which explains the technology related to graphics programming by Unity. This series provides introductory content and applications for beginners, as well as tips for intermediate and above, on a variety of topics that the authors are interested in.

The source code explained in each chapter is published in the github repository (https://github.com/IndieVisualLab/UnityGraphicsProgramming4), so you can read this manual while executing it at hand.

The difficulty level varies depending on the article, and depending on the amount of knowledge of the reader, some content may be unsatisfactory or too difficult. Depending on your knowledge, it's a good idea to read articles on the topic you are interested in. For those who usually do graphics programming at work, I hope it will lead to more effect drawers, and students are interested in visual coding, I have touched Processing and openFrameworks, but I still have 3DCG. For those who are feeling a high threshold, I would be happy if it would be an opportunity to introduce Unity and learn about the high expressiveness of 3DCG and the start of development.

IndieVisualLab is a circle created by colleagues (& former colleagues) in the company. In-house, we use Unity to program the contents of exhibited works in the category generally called media art, and we are using Unity, which is a bit different from the game system. In this book, knowledge that is useful for using Unity in the exhibited works may be scattered.

Recommended execution environment

Some of the contents explained in this manual use Compute Shader, Geometry Shader, etc., and the execution environment in which DirectX 11

operates is recommended, but there are also chapters where the contents are completed by the program (C #) on the CPU side.

I think that the behavior of the sample code released may not be correct due to the difference in environment, but please take measures such as reporting an issue to the github repository and replacing it as appropriate.

Requests and impressions about books

If you have any impressions, concerns, or other requests regarding this book (such as wanting to read the explanation about ◯◯), please feel free to use the Web form (https://docs.google.com/forms/d/e/1FAIpQLSdxeansJvQGTWfZTBN_2RTuCK_kRqhA6QHTZKVXHCijQnC8zw/ Please let us know via viewform) or email (lab.indievisual@ gmail.com).

第1章　GPU-Based Space Colonization Algorithm

This chapter introduces the GPU implementation of the Space Colonization Algorithm, an algorithm that generates a shape that blanchs along a point cloud, and its application examples.

The sample in this chapter is "Space Colonization" at
https://github.com/IndieVisualLab/UnityGraphicsProgramming4
.

図 1.1: SkinnedAnimation.scene

1.1　Introduction

The Space Colonization Algorithm was developed by Adam et al. * 1 as a tree modeling method.

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

A method of generating a branching shape from a given point cloud,

- The branches do not stick together too much and can be blanched while being properly separated.
- Since the arrangement of branches is determined by the initial point cloud arrangement, it is easy to control the shape.
- You can control the density of branches with simple parameters

It has the feature.

This chapter introduces the GPU implementation of this algorithm and application examples combined with skinning animation.

1.2 Algorithm

First, I will explain the Space Colonization Algorithm. The general steps of the algorithm are divided as follows.

1. Setup-Initialization
2. Search --Search for affected attractions
3. Attract-Attracting branches
4. Connect-Create a new node and connect to an existing node
5. Remove --Remove Attraction
6. Grow-Node Growth

1.2.1 Setup-Initialization

In the initialization phase, the point cloud is prepared as an attraction (point that will be the seed of the branch). Place one or more Nodes (branch branch points) in the attraction. This first placed Node will be the starting point for your branch.

In the figure below, Attraction is represented by a round dot and Node is represented by a square dot.

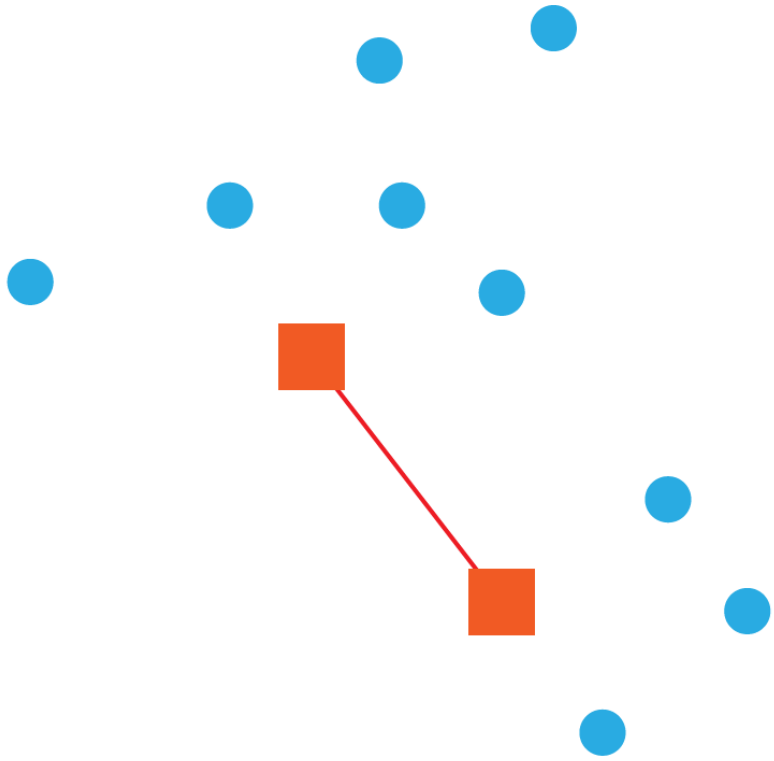

Figure 1.2: Setup --Attraction and Node Initialization Round dots represent Attraction and square dots represent Node.

1.2.2 Search-Search for affected Attractions

For each attraction, find the closest Node within the influence distance.

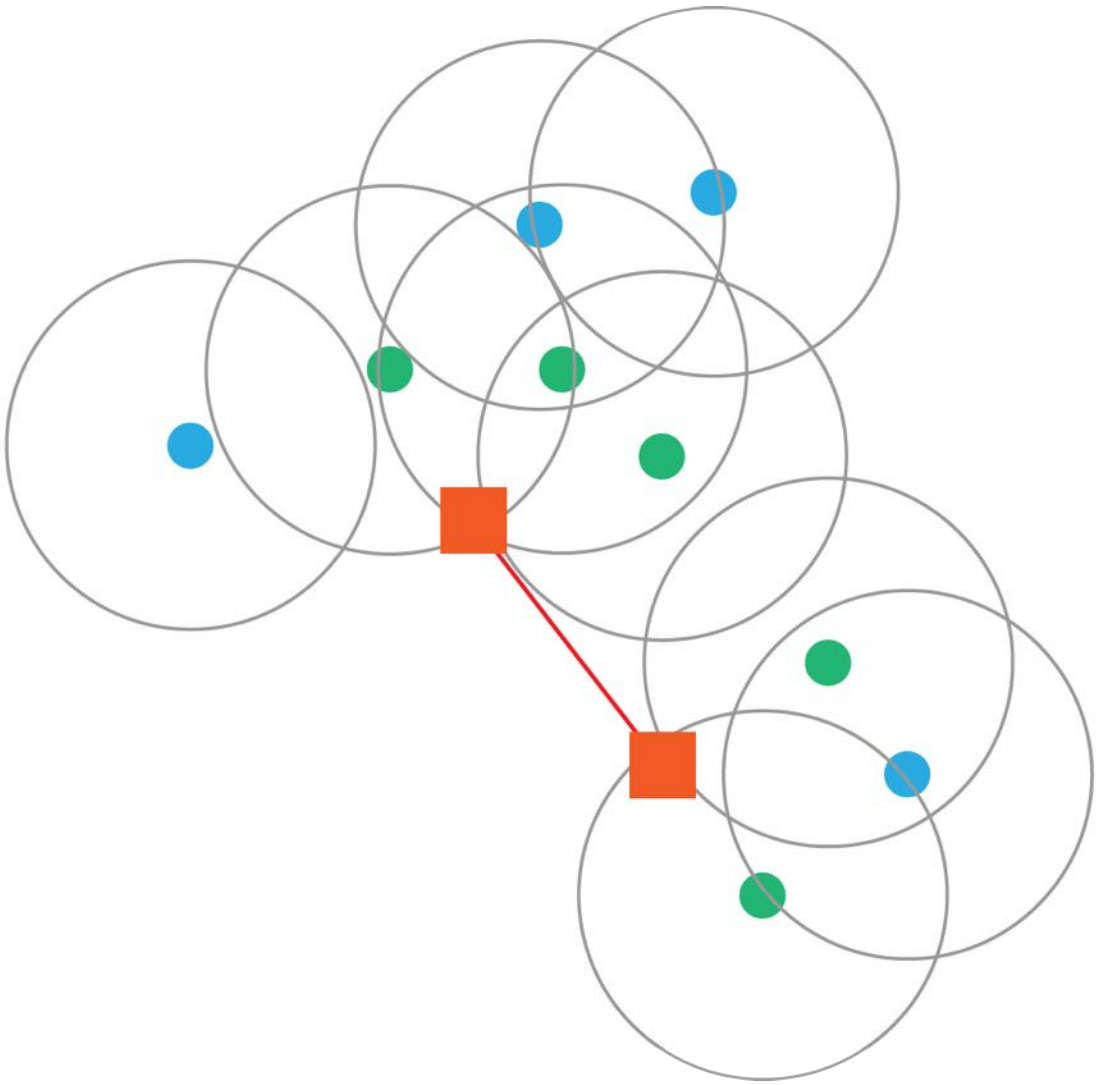

Figure 1.3: Search-Search for the nearest Node in the area of influence from each Attraction

1.2.3 Attract-Attracting branches

For each Node, determine the direction to extend the branch based on the attraction within the range of influence, and the point beyond the extension by the growth distance is the candidate point (Candidate) for the point to generate a new Node.)will do.

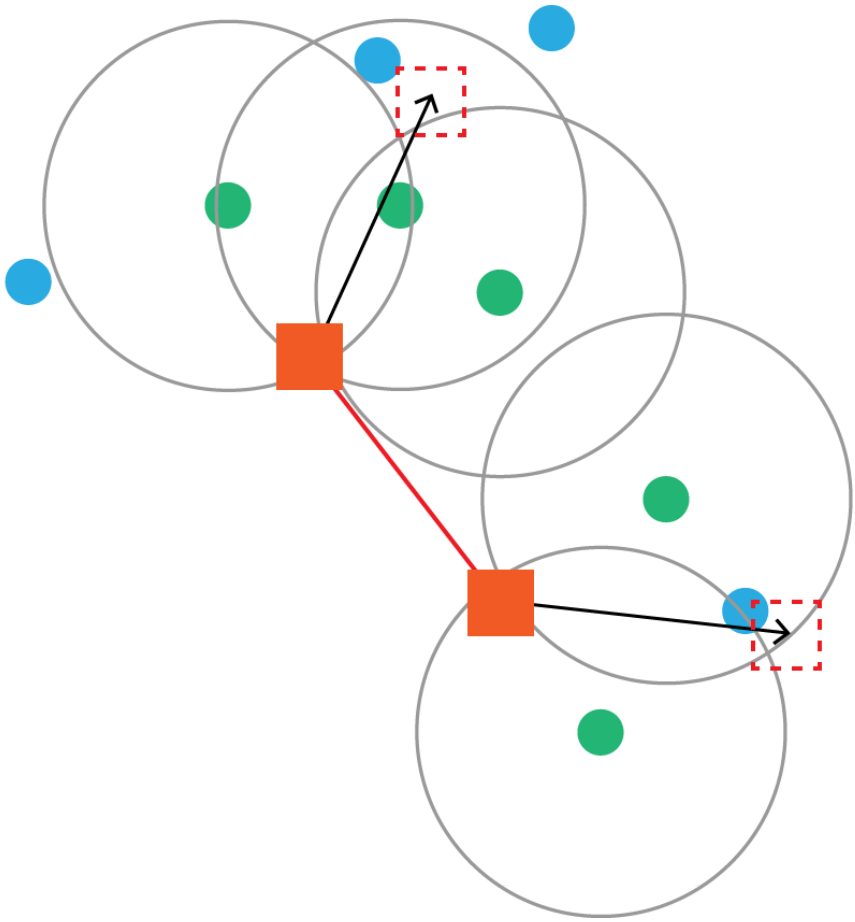

Figure 1.4: Attract-Extend a branch from each Node and determine candidate points to generate new Nodes

1.2.4 Connect-Connecting a new node to an existing node

Create a new Node at the Candidate position and connect the original Node with Edge to extend the branch.

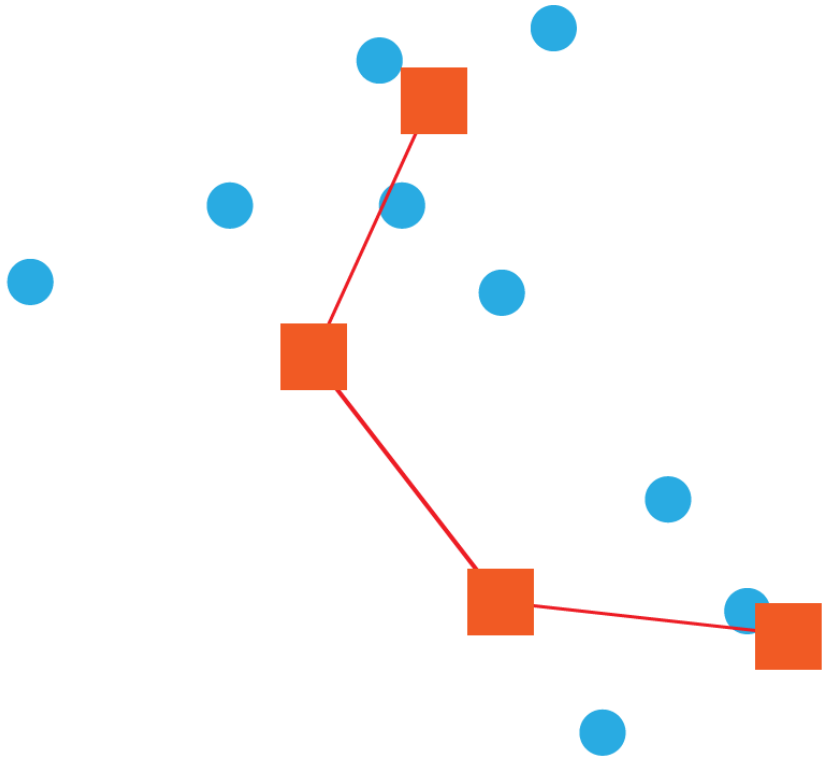

Figure 1.5: Connect-Connecting a new node to an existing node to extend a branch

1.2.5 Remove --Remove Attraction within the removal range

Deletes an attraction that is within the kill distance from the node.

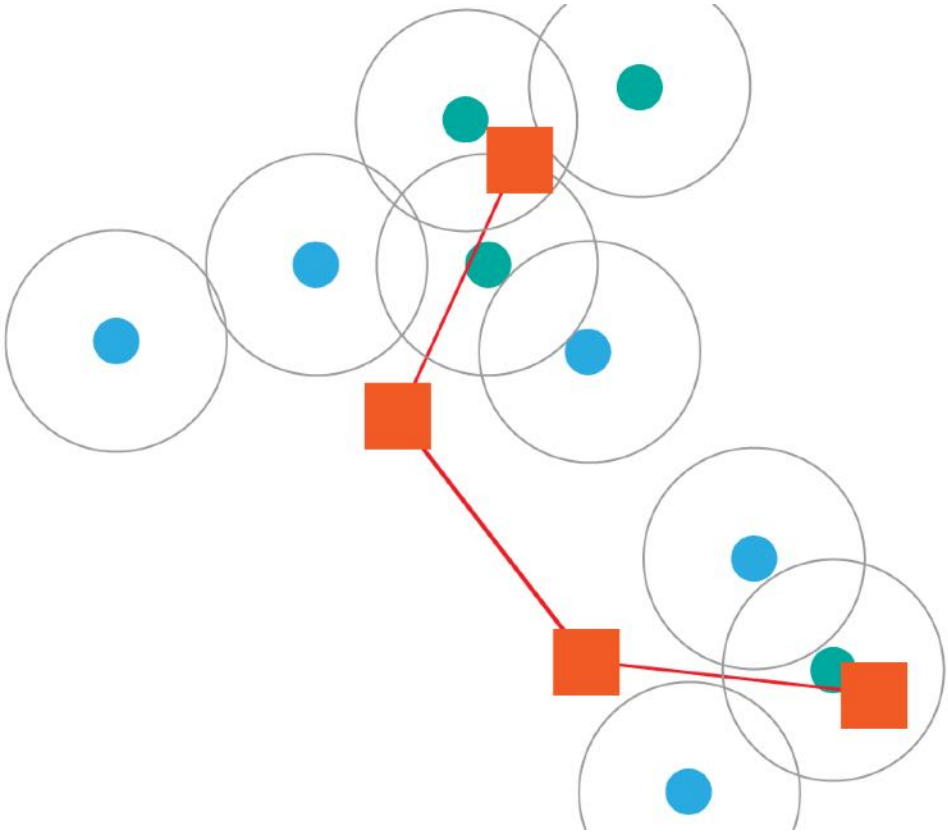

Figure 1.6: Remove --Search Node for attractions within the removal range

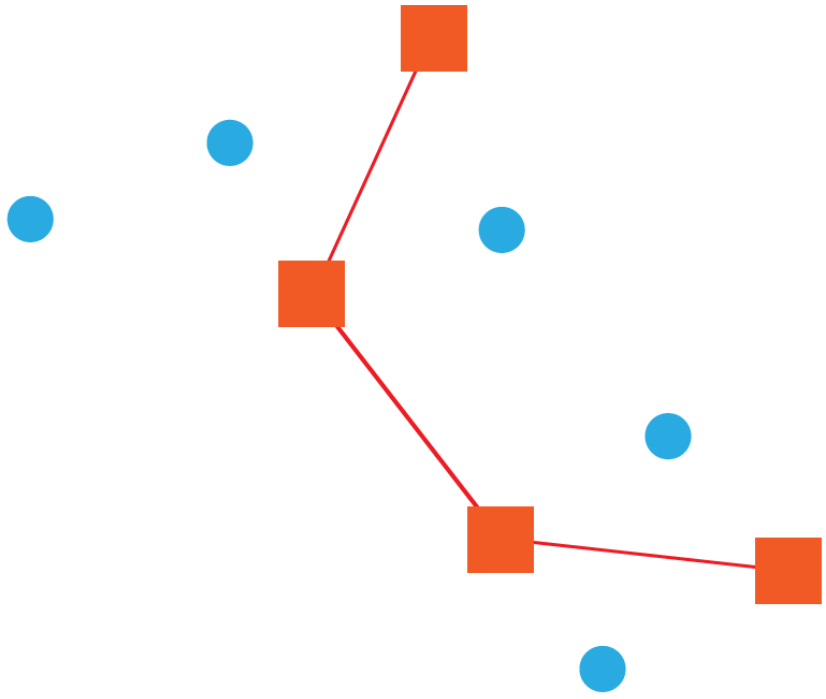

Figure 1.7: Remove-Removes Attraction found within the removal range

1.2.6 Grow --Node Growth

Grow Node and go back to Step.2.

The general flow of the entire algorithm is shown in the figure below.

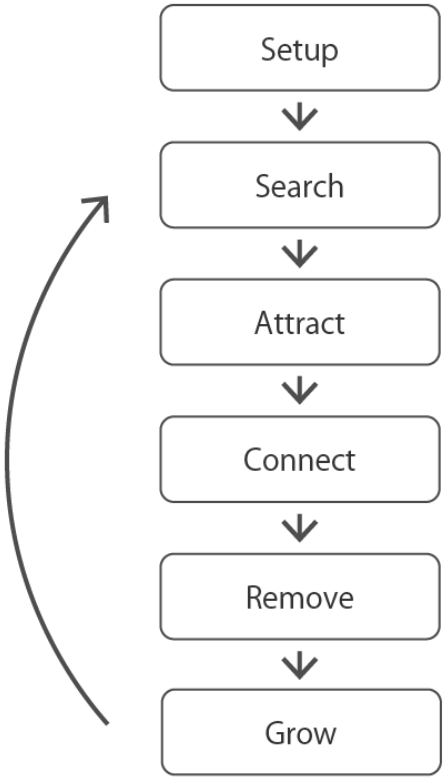

Figure 1.8: Rough flow of the algorithm

1.3 Implementation

Now, I will explain the concrete implementation of the algorithm.

1.3.1 Preparation of resources

As an element that increases or decreases in the Space Colonization Algorithm

- Attraction: Point cloud to seed the branch
- Node: Branch branch point (node)
- Candidate: Candidate points for creating new nodes
- Edge: An edge that connects the nodes of a branch

Is required, but in order to express these on GPGPU, Append / ConsumeStructuredBuffer is used for some elements.

Append / ConsumeStructuredBuffer is explained in Unity Graphics Programming vol.3 "GPU-Based Cellular Growth Simulation".

Attraction (branch seed point)

The structure of Attraction is defined as follows.

Attraction.cs

```
public struct Attraction {
    public Vector3 position; // position
    public int nearest; // index of the nearest Node
    public uint found; // Whether a nearby Node was found
     public uint active; // Whether it is a valid attraction (1
is valid, 0 is deleted)
}
```

The increase / decrease of attraction is expressed by determining whether it is a deleted attraction by the active flag.

In Space Colonization, it is necessary to prepare a point cloud of Attraction in the initialization phase. In the sample SpaceColonization.cs, point clouds are randomly scattered inside the sphere and used as the position of Attraction.

SpaceColonization.cs

```
    // Randomly sprinkle points inside the sphere to generate an
attraction
```

```
    var attractions = GenerateSphereAttractions();
    count = attractions.Length;

    // Initialize the Attraction buffer
    attractionBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(Attraction)),
        ComputeBufferType.Default
    );
    attractionBuffer.SetData(attractions);
```

Node (branch branch point)

The structure of Node is defined as follows.

Node.cs

```
public struct Node {
    public Vector3 position; // position
    public float t; // Growth rate (0.0 ~ 1.0)
    public float offset; // Distance from Root (Node depth)
    public float mass; // mass
    public int from; // index of branch source Node
     public uint active; // Whether it is a valid Node (1 is
valid)
}
```

Node resources

- Buffer representing the actual data of Node
- A buffer that manages the index of inactive Nodes used as an object pool

It is managed by two buffers.

SpaceColonization.cs

```
  // Actual Node data
  nodeBuffer = new ComputeBuffer(
    count,
    Marshal.SizeOf(typeof(Node)),
    ComputeBufferType.Default
  );
```

```
  // Object pool
  nodePoolBuffer = new ComputeBuffer(
    count,
    Marshal.SizeOf(typeof(int)),
    ComputeBufferType.Append
  );
  nodePoolBuffer.SetCounterValue(0);
```

Candidate (candidate for new Node)

The structure of Candidate is defined as follows.

Candidate.cs

```
public struct Candidate
{
    public Vector3 position; // position
      public int node; // index of the original Node of the
candidate point
}
```

Candidate is represented by Append / ConsumeStructuredBuffer.

SpaceColonization.cs

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

Edge (Edge that connects Nodes)

The structure of Edge is defined as follows.

Edge.cs

```
public struct Edge {
    public int a, b; // index of two Nodes connected by Edge
}
```

Edge is represented by Append / Consume Structured Buffer like Candidate.

SpaceColonization.cs

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

1.3.2 Implementing the algorithm in Compute Shader

Now that we have the necessary resources, we will implement each step of the algorithm in GPGPU with Compute Shader.

Setup

In the initialization phase

- Node object pool initialization
- Add initial Node as seed

to hold.

Pick up some from the prepared Attraction and generate an initial Node at that position.

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);
```

```
    ...
}
```

The Setup kernel initializes the object pool. Store the index in the Node's object pool and turn off the active flag for that Node.

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;
}
```

This will turn off the active flags for all Nodes and create an object pool with Node indexes.

Now that the object pool has been initialized, it's time to create the initial seed node.

The initial node is generated by executing the Seed kernel with the seed position (Vector3 [ ]) prepared earlier as input.

SpaceColonization.cs

```
...

// seedBuffer is automatically disposed when it goes out of
scope
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);
}

// Initialize the number of Nodes and Edges
nodesCount = nodePoolBuffer.count;
edgesCount = 0;

...
```

The Seed kernel takes a position from the Seeds buffer and creates a Node at that position.

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;

  // Create a new Node (see below)
  uint i = CreateNode(n);

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

Create a new Node with the CreateNode function. Extracts the index from the object pool ConsumeStructuredBuffer and returns the initialized Node.

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;
}
```

This is the end of the initialization phase.

Each step of the looping algorithm shown in Figure 1.8 is performed within the Step function.

SpaceColonization.cs

```
protected void Step(float dt)
{
    // Do not run when the object pool is empty
    if (nodesCount > 0)
    {
        Search();   // Step.2
        Attract();  // Step.3
        Connect();  // Step.4
        Remove();   // Step.5

                // Get the number of data that Append /
ConsumeStructuredBuffer has
        CopyNodesCount();
        CopyEdgesCount();
    }
    Grow(dt);       // Step.6
}
```

Search

From each attraction, find the closest Node within the influence distance.

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);
}
```

The GPU kernel implementation is as follows.

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);

    // Search for Nodes closer than influence distance
    float min_dist = _InfluenceDistance;

    // index of the nearest Node
    uint nearest = -1;

    // Execute a loop for all Nodes
    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)
        {
          // Update the nearest Node
          min_dist = d;
          nearest = i;

          // Set the index of the neighboring Node
          attr.found = true;
          attr.nearest = nearest;
        }
      }
    }

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

Attract

For each Node, determine the direction to extend the branch based on the attraction within the range of influence, and the point beyond the extension by the growth distance is the candidate point (Candidate) for the point to generate a new Node.)will do.

SpaceColonization.cs

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

       candidateBuffer.SetCounterValue (0); // Initialize the
buffer that stores the candidate points
               compute.SetBuffer(kernel,    "_CandidatesAppend",
candidateBuffer);

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

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

The GPU kernel implementation is as follows. Please refer to the code contents and comments of the Attract kernel for the calculation method of the position of the candidate point.

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 is valid and
  // Create a new Node if the growth rate (t) is greater than or
equal to the threshold (1.0)
  if (n.active && n.t >= 1.0)
  {
    // Accumulation variable to extend the branch
    float3 dir = (0.0).xxx;
    uint counter = 0;

    // Run a loop for all attractions
    _Attractions.GetDimensions(count, stride);
    for (uint i = 0; i < count; i++)
    {
      Attraction attr = _Attractions[i];
        // Search for the attraction whose node is the nearest
neighbor
      if (attr.active && attr.found && attr.nearest == idx)
      {
         // Normalize the vector from Node to Attraction and add
it to the accumulation variable
        float3 dir2 = (attr.position - n.position);
        dir + = normalize (dir2);
        counter++;
      }
    }

    if (counter > 0)
    {
      Candidate c;

         // Take the average of the unit vectors from Node to
```

```
Attraction
      // Set it as the position of the candidate point extended
from the Node by the growth distance
      dir = dir / counter;
      c.position = n.position + (dir * _GrowthDistance);

      // Set the index of the original Node that extends to the
candidate point
      c.node = idx;

      // Add to candidate point buffer
      _CandidatesAppend.Append(c);
    }
  }
}
```

Connect

Create a new Node based on the candidate point buffer generated by the Attract kernel, and extend the branch by connecting the Nodes with Edge.

In the Connect function, the number of kernel executions is determined by comparing the remaining number of object pools (nodesCount) with the size of the candidate point buffer so that data retrieval (Consume) is not executed when the Node object pool (nodePoolBuffer) is empty. I am.

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);

     // The number of data (nodeCount) of the Node object pool
acquired by CopyNodeCount
    // Restrict so that it does not exceed
                var    connectCount    =    Mathf.Min(nodesCount,
```

```
CopyCount(candidateBuffer));
    if (connectCount > 0)
    {
        compute.SetInt("_ConnectCount", connectCount);
        GPUHelper.Dispatch1D(compute, kernel, connectCount);
    }
}
```

Below is the implementation of the GPU kernel.

SpaceColonization.compute

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

  // Extract candidate points from the candidate point buffer
  Candidate c = _CandidatesConsume.Consume();

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

  // Generate Node at the position of the candidate point
  uint idx2 = CreateNode(n2);
  n2.position = c.position;
   n2.offset = n1.offset + 1.0; // Set the distance from Root
(original Node + 1.0)
  n2.from = c.node; // Set the index of the original Node
     n2.mass  =  lerp(_MassMin,  _MassMax,  nrand(float2(c.node,
idx2)));

  // Update Node buffer
  _Nodes[c.node] = n1;
  _Nodes[idx2] = n2;

  // Connect two Nodes with Edge (see below)
  CreateEdge(c.node, idx2);
}
```

The CreateEdge function creates an Edge based on the indexes of the two Nodes passed and adds it to the Edge buffer.

SpaceColonization.compute

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

Remove

Remove the Attraction that is within the kill distance from the Node.

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);
}
```

The GPU kernel implementation is as follows.

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];
  // Do not execute if the attraction has been deleted
  if (!attr.active)
    return;

  // Execute a loop for all Nodes
  _Nodes.GetDimensions(count, stride);
  for (uint i = 0; i < count; i++)
  {
```

```
    Node n = _Nodes[i];
    if (n.active)
    {
      // If there is a Node within the deletion range, turn off
the active flag of Attraction and delete it
      float d = distance(attr.position, n.position);
      if (d < _KillDistance)
      {
        attr.active = false;
        _Attractions[idx] = attr;
        return;
      }
    }
  }
}
```

Grow

Grow Node.

When generating candidate points with the Attract kernel, it is used as a condition whether the growth rate (t) of Node is above the threshold value (if it is below the threshold value, no candidate points are generated), but the growth rate parameter is in this Grow kernel. I'm incrementing.

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)
  {
    // Disperse the growth rate with the mass parameter randomly
set for each Node
    n.t = saturate(n.t + _DT * n.mass);
    _Nodes[idx] = n;
  }
}
```

1.3.3 Rendering

Now that we have a blanched shape with the above implementation, let's talk about how to render that shape.

Rendering with Line Topology

First, simply render using Line Mesh.

Generate a simple Line Topology Mesh to draw a Line that represents a single Edge.

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;
}
```

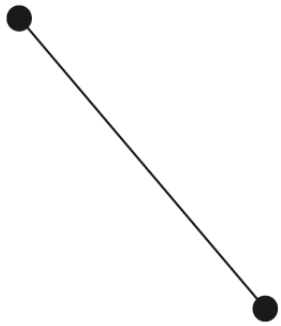

Figure 1.9: Line Topology Mesh with only two simple vertices

Display the branches generated by rendering a Segment (line segment) with only two vertices using GPU instancing for the number of Edges.

SpaceColonization.cs

```
// Generate a buffer that determines the number of meshes to
render, required for GPU instancing
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);
}

...

// Perform rendering with GPU instancing
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
    );
}
```

The shader for rendering (Edge.shader) generates an animation of the branch extending from the branch point by controlling the length of the Edge according to the growth rate parameter (t) of the Node.

Edge.shader

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

  // Get the corresponding Edge from the instance ID
  Edge e = _Edges[iid];

  // Get 2 Nodes from the index of Edge
  Node a = _Nodes[e.a];
  Node b = _Nodes [eb];

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

   // Determine the length of Edge from a to b according to the
growth rate (t) of Node b
  bp = ap + normalize (dir) * length (dir) * bt;

   // Since the vertex ID (IN.vid) is 0 or 1, if it is 0, it
refers to the node of a, and if it is 1, it refers to the
position of Node of b.
  float3 position = lerp(ap, bp, IN.vid);
```

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

  // If Node is inactive or the instance ID is outside the total
number of Edges, set alpha to 0 and do not draw
  OUT.alpha = (a.active && b.active) && (iid < _EdgesCount);

  return OUT;
}
```

With these implementations, the shape obtained by the Space Colonization Algorithm can be rendered using Line Topology. You can get the following picture by executing Line.scene.

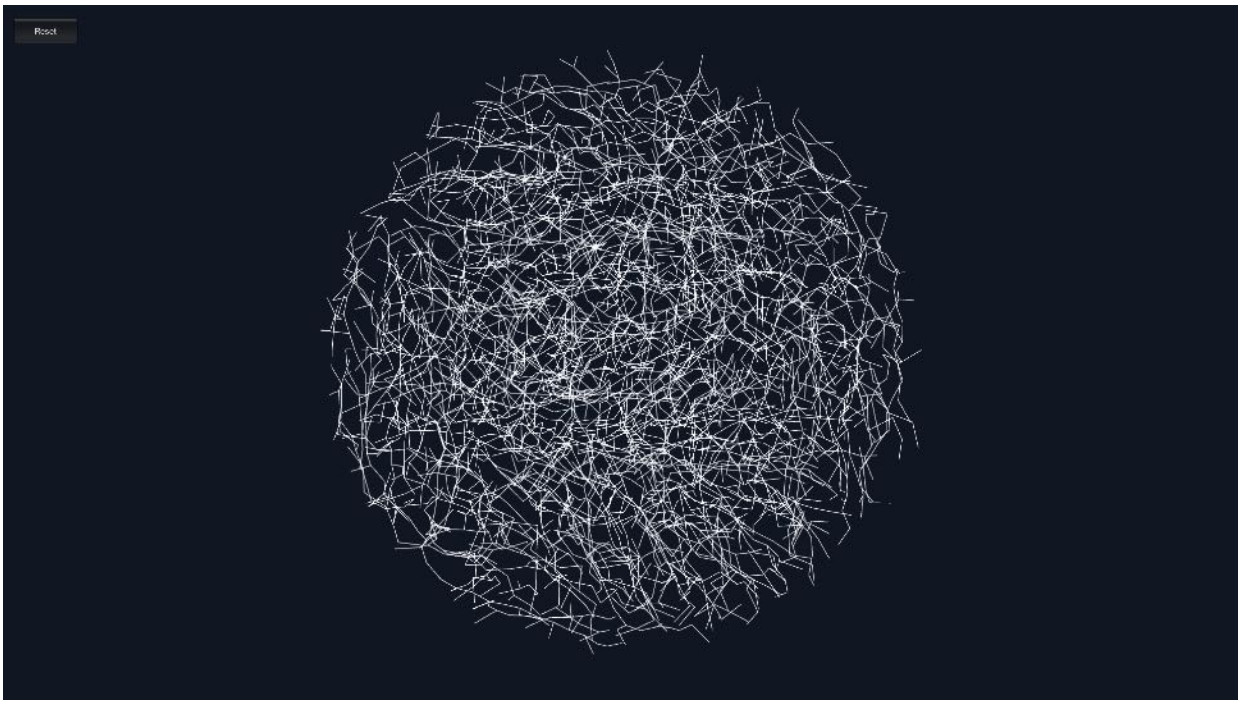

Figure 1.10: Line.scene --Example of rendering with Edge.shader

Rendering with Geometry Shader

By converting the Line Topology Segment to a Capsule shape with the Geometry Shader, you can draw thick lines.

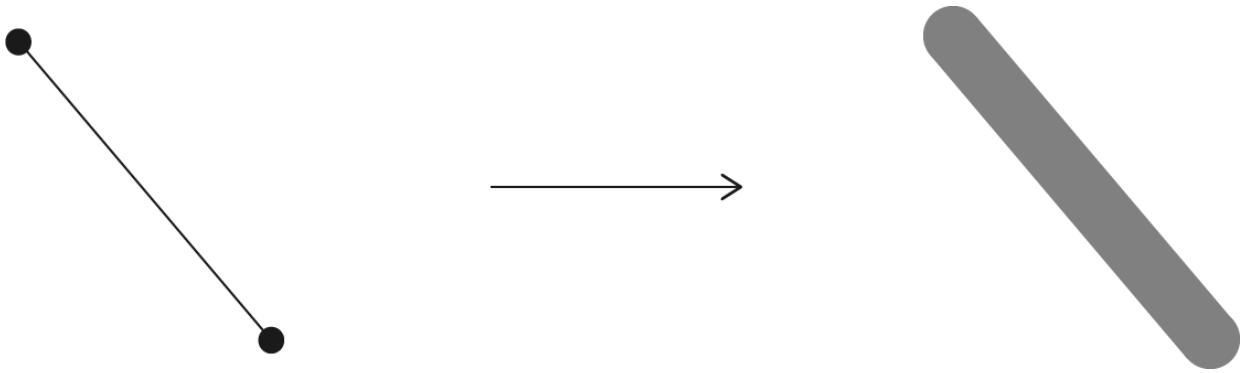

Figure 1.11: Convert Line Topology Segment to Capsule shape with Geometry Shader

The vertex shader is almost the same as Edge.shader, and Geometry Shader builds the Capsule shape. Only the important Geometry Shader implementations are listed below.

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;

  // Definition of Capsule mesh resolution
  static const uint rows = 6, cols = 6;
   static const float rows_inv = 1.0 / rows, cols_inv = 1.0 /
(cols - 1);

  g2f00, o1;
  o0.uv = p0.uv; o0.uv2 = p0.uv2;
  o1.uv = p1.uv; o1.uv2 = p1.uv2;
```

```
  // Build aspects of the Capsule
  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();

  // Construction of Capsule tip (hemispherical)
  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();
  }
}

...
```

The result looks like this: (TubularEdge.scene)

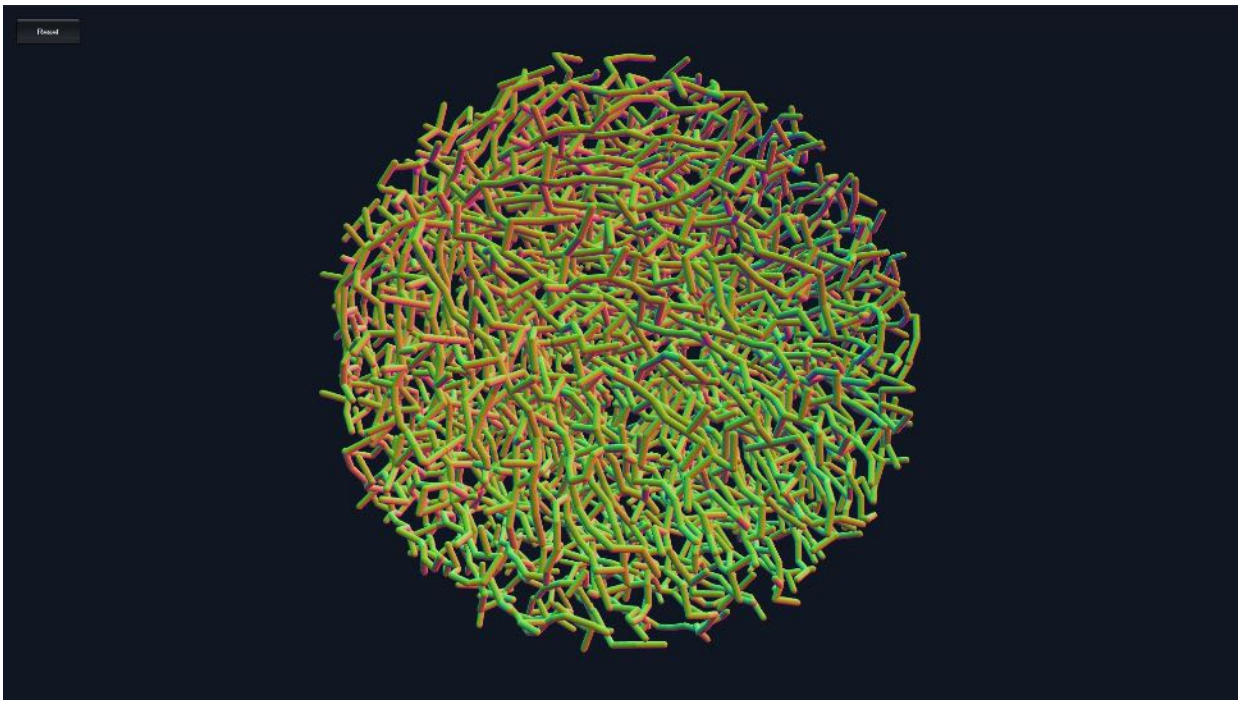

Figure 1.12: TubularEdge.scene --Example of rendering with TubularEdge.shader

Now that you can render Edge with a thick mesh, you can add lighting and so on.

1.4 Application

With the above, we have realized the GPU implementation of Space Colonization. In this section, we will introduce the cooperation with skinning animation as an application example.

With this application, it is possible to realize an expression that blanchs along the animated model shape.

1.4.1 Rough flow

The cooperation with skinning animation is developed according to the following flow.

1. Prepare an animation model
2. Prepare a point cloud that fills the volume of the model (point cloud that becomes an attraction)
3. Give Attraction and Node Bone information
4. Apply (skinning) Bone transformation to Node to change its position

1.4.2 Preparation of resources

The structure of the previous example

- Attraction
- Node
- Candidate

Make changes to have the Bone index.

About Bone Limits Affected

In this application, the number of bones that affect each node is limited to one. Originally, skinning animation could have multiple bones affecting each vertex, but in this example we simply limit it to only one.

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; // Node position after skinning
animation
    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
}
```

Animation model and volume

Prepare the animation model you want to link.

In this example, the model downloaded from Clara.io * 2 is used (the number of polygons is reduced by reduction with MeshLab * 3), and the animation is generated by mixamo * 4 .

[*2] https://clara.io/view/d49ee603-8e6c-4720-bd20-9e3d7b13978a

[*3] http://www.meshlab.net/

[*4] https://mixamo.com

In order to get the position of Attraction from the model volume, we use a package called VolumeSampler * 5 that generates a point cloud in the model

volume .

[*5] https://github.com/mattatz/unity-volume-sampler

VolumeSampler

VolumeSampler acquires the volume of the model using the technique explained in Unity Graphics Programming vol.2 "Real-Time GPU-Based Voxelizer". First, the volume inside the mesh is acquired as Voxel, and Poisson Disk Sampling is executed based on it to generate a point cloud that fills the inside of the mesh.

To generate a point cloud asset using VolumeSampler, click Window → VolumeSampler from the Unity toolbar to display Window, and as shown in the figure below.

- Reference to the target mesh
- Point cloud sampling resolution

If you set and click the asset generation button, the Volume asset will be generated in the specified path.

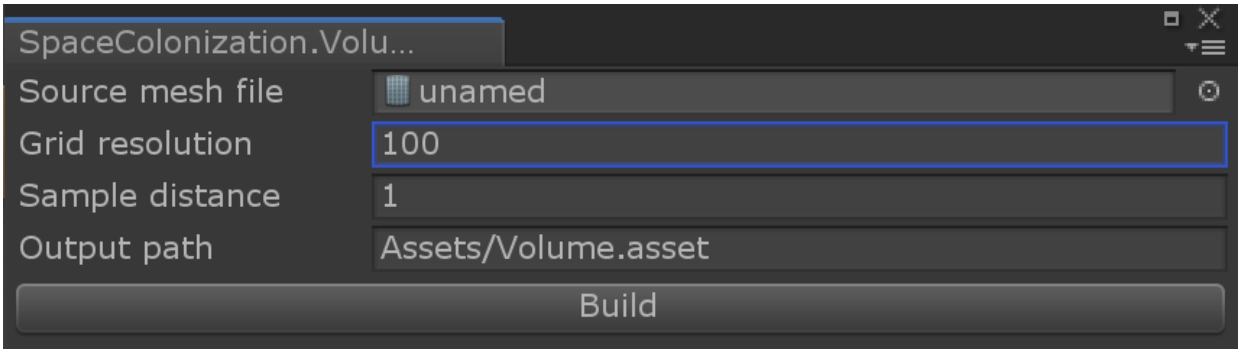

図 1.13: VolumeSamplerWindow

Generate a SkinnedAttraction array from the point cloud asset (Volume class) generated from VolumeSampler and apply it to ComputeBuffer.

SkinnedSpaceColonization.cs

```
protected void Start() {
    ...
    // Generate a Skinned Attraction array from the point cloud
of Volume
    attractions = GenerateAttractions(volume);
    count = attractions.Length;
    attractionBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(SkinnedAttraction)),
        ComputeBufferType.Default
    );
    attractionBuffer.SetData(attractions);
    ...
}
```

Applying Bone information

Bone information of the nearest vertex from each position is applied to Skinned Attraction generated from the volume of Mesh.

The SetupSkin function prepares the vertices of the mesh and the bone buffer, and assigns the bone index to all Skinned Attraction on the GPU.

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);
    }
}
```

Below is the implementation of the GPU kernel.

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];

   // Get the index of the nearest vertex from the position of
Skinned Attraction
  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;
    }
  }

    // Set the bone index of the nearest vertex to Skinned
Attraction
  attr.bone = _Bones[closest];
  _Attractions[idx] = attr;
}
```

1.4.3 Implementation of algorithm in Compute Shader

In this application, some GPU kernels are modified to get the bone information needed for skinning animation in each step of the Space Colonization Algorithm.

The contents of the GPU kernel are almost the same, but for the generated SkinnedNode, it is necessary to obtain Bone information (Bone index) from the nearest Skinned Attraction, so

- Seed
- Attract

In the two GPU kernels, neighborhood search logic has been added.

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));

  // Search for the nearest Skinned Attraction and
  // Copy the Bone index
  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++;

        // Search for the nearest Skinned Attraction
        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;
      // Set the bone index of the nearest Skinned Attraction
      c.bone = _Attractions[nearest].bone;
      _CandidatesAppend.Append(c);
    }
  }
}
```

Skinning animation

With the above implementation, you can now execute the Space Colonization Algorithm while setting the Bone information for Node.

After that, you can apply skinning animation to Node by getting the required Bone matrix from SkinnedMeshRenderer and moving the position of SkinnedNode on the GPU according to the deformation of Bone.

SkinnedSpaceColonization.cs

```
protected void Start() {
    ...
    // Create a buffer for the bind pose matrix
    var bindposes = skinnedRenderer.sharedMesh.bindposes;
    bindPoseBuffer = new ComputeBuffer(
        bindposes.Length,
        Marshal.SizeOf(typeof(Matrix4x4))
    );
    bindPoseBuffer.SetData(bindposes);
    ...
}

protected void Animate()
{
     // Create a buffer representing the SkinnedMeshRenderer' s
Bone matrix that is updated as the animation plays
    var bones = skinnedRenderer.bones.Select(bone => {
        return bone.localToWorldMatrix;
    }).ToArray();
    using (
        ComputeBuffer boneMatrixBuffer = new ComputeBuffer(
            bones.Length,
            Marshal.SizeOf(typeof(Matrix4x4))
        )
    )
    {
        boneMatrixBuffer.SetData(bones);

           // Pass the Bone and Node buffers and perform GPU
skinning
        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)
  {
    // Perform skinning
    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 Rendering

The shaders for rendering are almost the same, except that the Edge is drawn by referring to the animated position after skinning animation, not the original position of the SkinnedNode.

SkinnedTubularEdge.hlsl

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

  // Refer to the position after applying skinning animation
  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) * bt;
  float3 position = lerp(ap, bp, IN.vid);
```

```
    OUT.position = mul(unity_ObjectToWorld, float4(position,
1)).xyz;
  ...
}
```

With the above implementation, you can get the capture picture shown at the beginning. (Fig. 1.1 SkinnedAnimation.scene)

1.5 Summary

In this chapter, we introduced the GPU implementation of the Space Colonization Algorithm that generates a blanching shape along a point cloud, and an application example that combines it with skinning animation.

With this technique,

- Attraction distance
- Growth distance
- The range to kill the attraction (kill distance)

You can control the density of branches with these three parameters, but you can generate more diverse models by changing these parameters locally or with time.

Also, in the sample, Attraction runs the algorithm only with what was generated during initialization, but you should be able to generate more different patterns by dynamically increasing Attraction.

If you are interested, try various applications of this algorithm to find interesting patterns.

1.6 Reference

- 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

Chapter 2 Limit sets of Kleinian groups

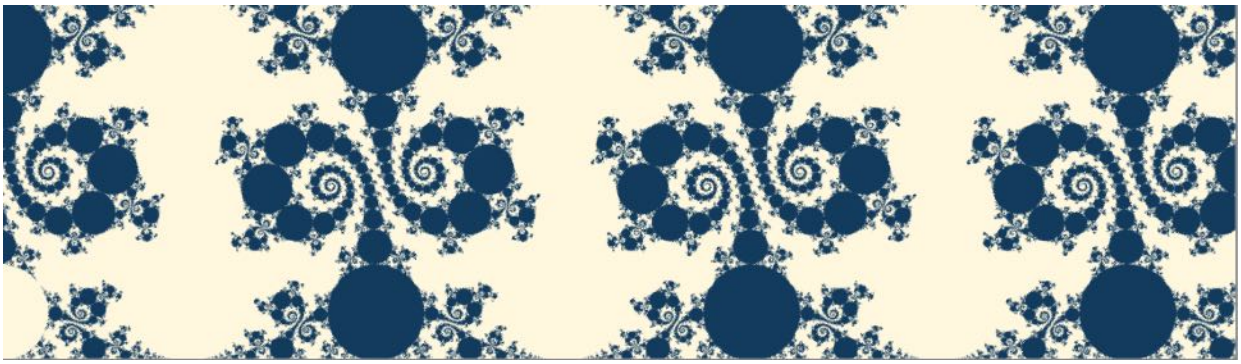

Figure 2.1:

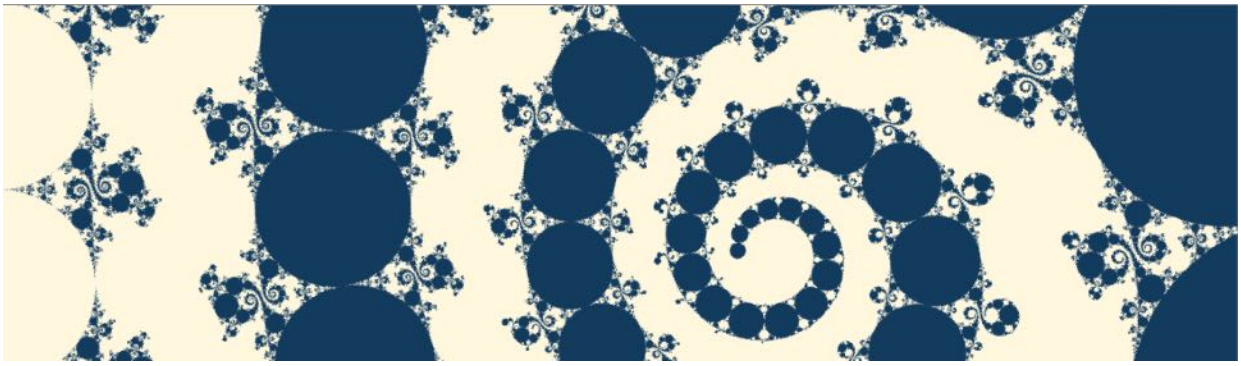

Figure 2.2:

In this chapter, we will introduce how to draw the limit set of Kleinian groups with a shader and animate the resulting fractal figure. Speaking of fractal animation, it is interesting to see a self-similar figure by scaling it up or down, but with this method, you can see a characteristic movement in which straight lines and circumferences transition more smoothly.

The sample in this chapter is "Kleinian Group" of
https://github.com/IndieVisualLab/UnityGraphicsProgramming4
.

2.1 Yen reversal

First, I will introduce the inversion of figures. I think it is familiar that a figure that is inverted like a mirror image with a straight line as a boundary is line-symmetrical, and if it is inverted around a point, it is point-symmetrical. There is also a reversal of circles. It is an operation to switch the inside and outside of the circle on the two-dimensional plane.

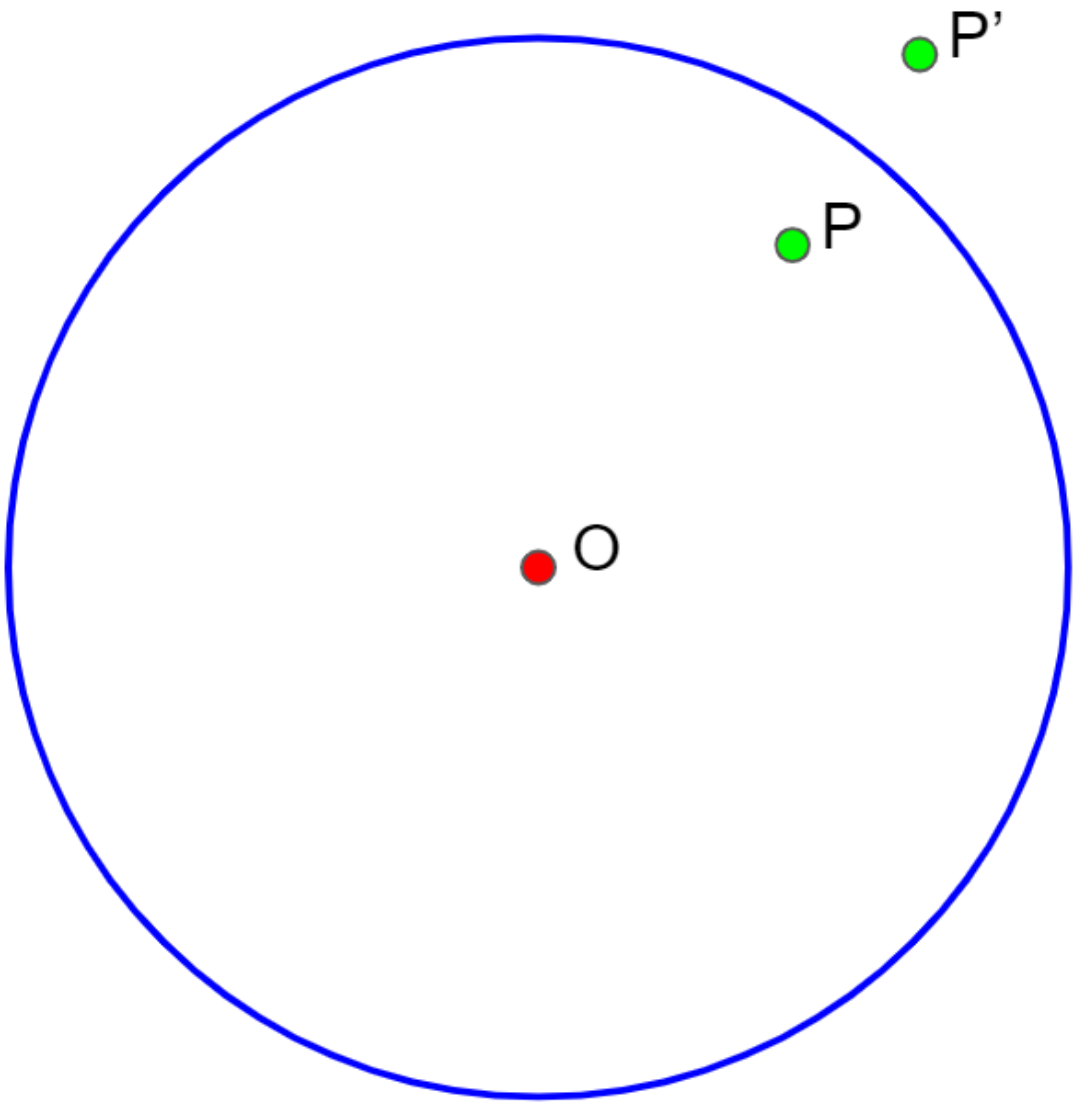

Figure 2.3: Circle Inversion P \ rightarrow P'

The center of the circle O , radius r inverted with respect to the circle of, \ Left | OP \ Right | a | '\ right | \ left OP is so as to satisfy the following equation remains the same direction P to P' to move to the operation Become.

```
\left| OP\right| \cdot \left| OP' \right| =r^{2}
```

In the vicinity of the circumference, the inside and outside appear to be interchanged like a distorted line symmetry, and the infinity far away from the circumference and the center of the circle are interchanged. What is interesting is the case where the straight line on the outside of the circle is inverted, and when it is close to the circle, it moves to the inside across the circumference and continues to infinity as it moves away from it. Become.

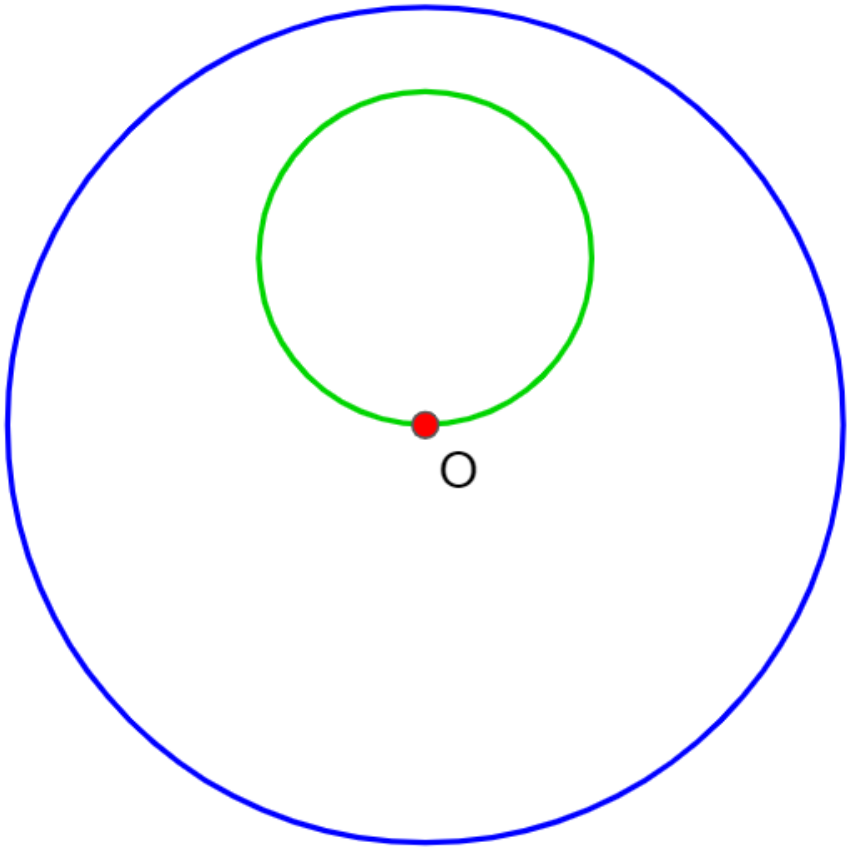

Figure 2.4: Straight line inversion

It will appear as a small circle inside the circle. If you think of a straight line as a circle with an infinite radius, you can say that circle inversion is an operation that swaps the circles inside and outside the circle.

2.1.1 Expressed by mathematical formula

The formula for the circle inversion of the unit circle on the complex plane is as follows.

```
z \ rightarrow \ dfrac {1} {\ overline {z}}
```

z is a complex number and \ overline {z} is its complex common benefit.

The next way and try to formula deformation z 1 divided by the square power of the length of the z you can see that is an operation that is scale.

```
z\rightarrow \dfrac {1}{\overline {z}}=\dfrac {1}{x-iy}=\dfrac
{x+iy}{\left( x-iy\right) \left( x+iy\right) }=\dfrac {x+iy}
{x^{2}+y^{2}}=\dfrac {z}{\left| z\right| ^{2}}
```

As a graphic operation on the complex plane,

- Sum of complex numbers: translation
- Complex product: rotation (and scaling)

I think that many people are aware that, but this is where operations including division are newly added.

2.2 Mobius transformation

The Mobius transformation * 1 is a generalized form that includes division in the transformation on the complex plane .

[* 1] It is derived from the mathematician August Ferdinand Mobius, who is familiar with Mobius strip.

```
z \ rightarrow \ dfrac {az + b} {cz + d}
```

a, b, c, d are all complex numbers.

2.3 Schottky group, Kleinian group

Consider creating a fractal figure by repeatedly using the Mobius transformations.

Prepare four sets of circles D_A , D_a , D_B , D_b that do not intersect each other . First of which focuses on two sets of circle D_A the outside of D_a inside, D_A the inside of D_a Mobius transformation transferred to outside a create a. Similarly, make a Mobius transformation b from two other sets of circles D_B and D_b . Also , prepare the inverse transformations A and B respectively .

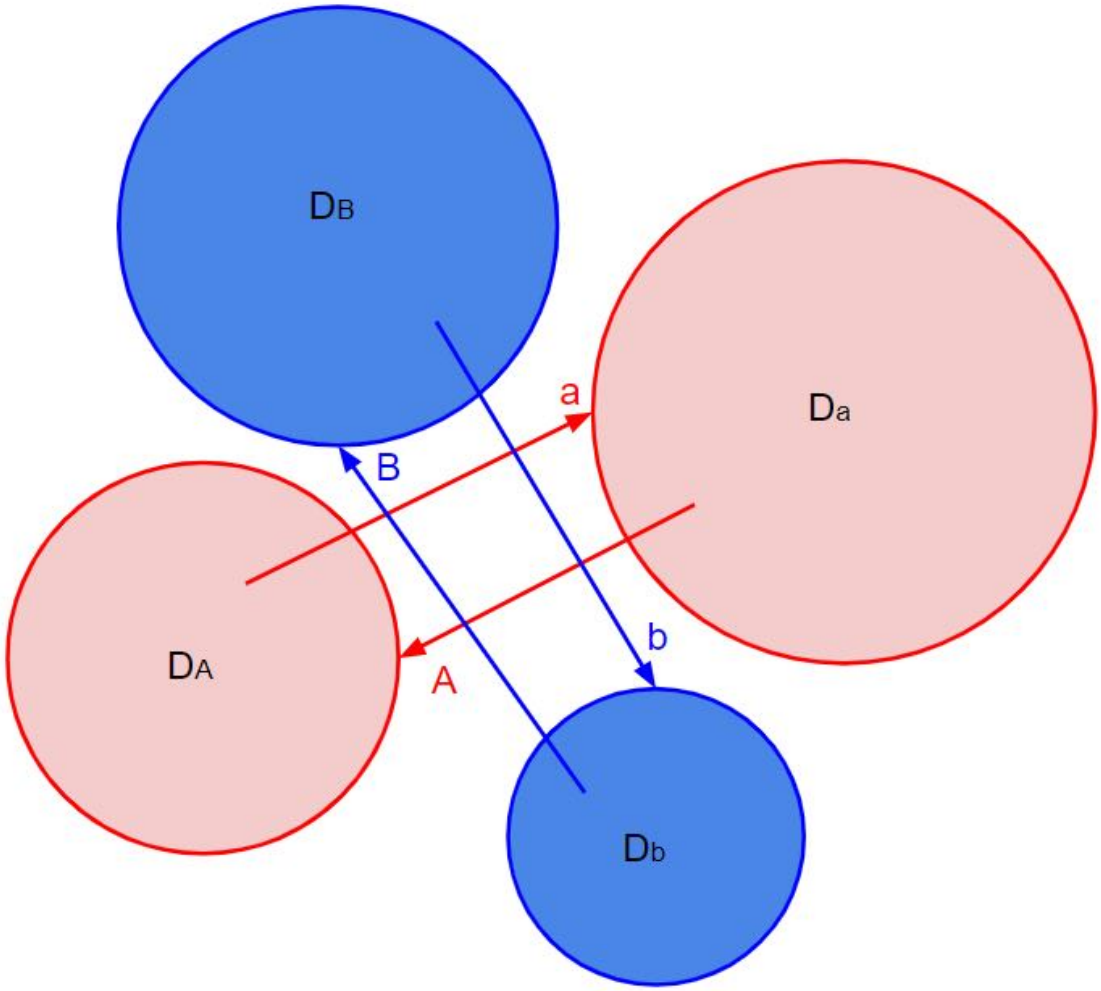

Figure 2.5:

The entire transformation (for example, aaBAbbaB) obtained by synthesizing the four Mobius transformations a , A , b , and B in any order is called " **Schottky group** * 2 based on a , b ".

[* 2] It is derived from the mathematician Friedrich Hermann Schottky who first devised such a group.

This is further generalized, and the discrete group consisting of Mobius transformations is called the **Kleinian group** . I have the impression that this name is more widely used.

2.4 Limit set

When you display the image of the Schottky group, you will see a circle inside the circle, a circle inside it, and so on. These sets are called " the **limit set** of Schottky groups in a and b ". The purpose of this chapter is to draw this **limit set** .

2.5 Jos Leys' algorithm

2.5.1 Overview

I will introduce how to draw the limit set with a shader. It is difficult to implement it honestly because the combination of conversions continues infinitely, but Jos Lays has published an algorithm for this * 5, so I will try to follow it.

First, prepare two Mobius transformations.

```
a: z\rightarrow \dfrac {tz-i}{-iz}

b: z\rightarrow z+2
```

t is the complex number u + iv . The shape of the figure can be changed by changing this value as a parameter.

If you take a closer look at transformation a ,

```
a: z \ rightarrow \ dfrac {tz-i} {- iz} = \ dfrac {t} {- i} + \
dfrac {1} {z} = \ dfrac {1} {z} + (- v + iu )

\dfrac    {1}{z}=\dfrac    {1}{x+iy}=\dfrac    {x-iy}{(x+iy)(x-
iy)}=\dfrac          {x-iy}{x^{2}+y^{2}}=\dfrac          {x-iy}
{\left|z\right|^{2}}
```

Therefore,

```
a:      z\rightarrow      \dfrac      {tz-i}{-iz}=\dfrac{x-iy}
{\left|z\right|^{2}}+(-v+iu)
```

So

1. Invert the circle in the unit circle,
2. Invert the sign of y and
3. -v + iu parallel movement

You can see that it is an operation.

The limit set using transformations a and b and their inverse transformations is the following strip-shaped figure in which large and small circles are connected.

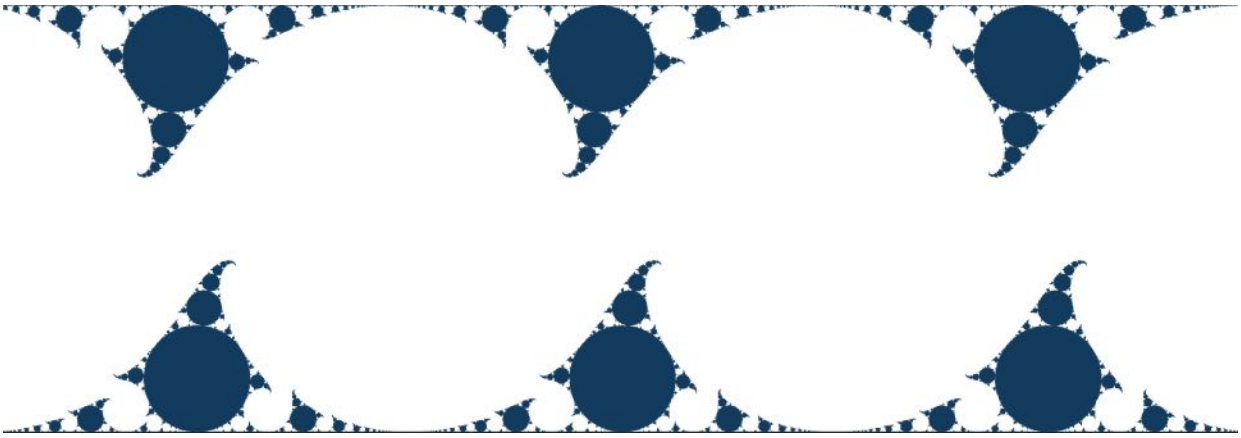

Figure 2.6: Limit set

Let's take a closer look at the features of this shape.

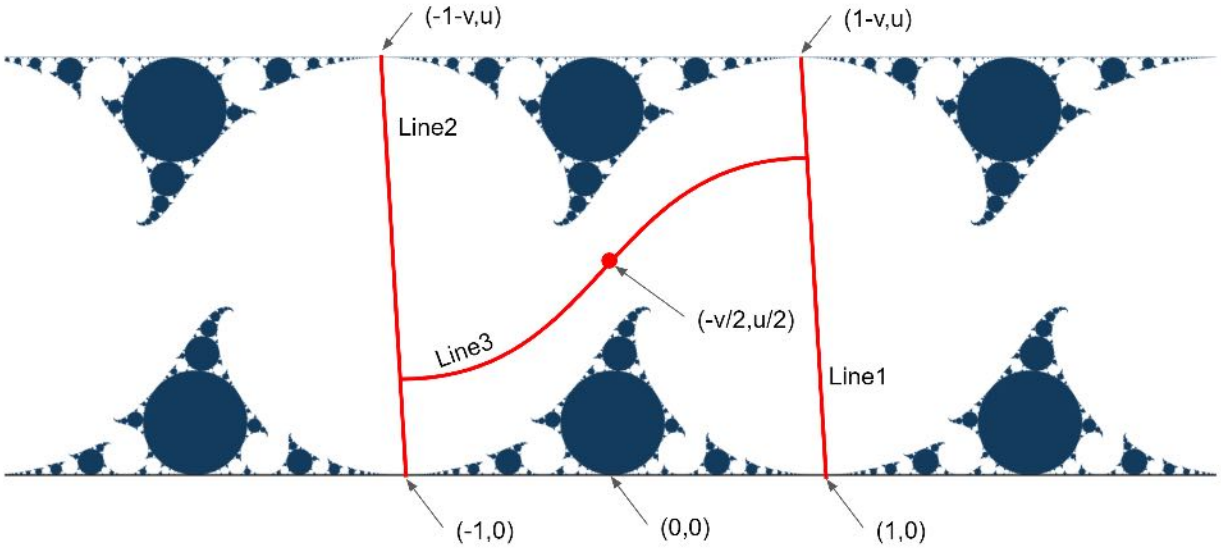

Figure 2.7:

It has a band shape of 0 \ leq y \ leq u , and the parallelograms separated by Lines 1 and 2 repeat in the left-right direction. Line1 is a straight line that passes through points (1,0) and points (1-v, u) , and Line2 is a straight line that passes through points (-1,0) and points (-1-v, u) . Line3 becomes a line that divides the figure into the upper limit, and the upper and lower figures divided by this line in the parallelogram are point symmetric at the point z =- \ dfrac {v} {2} + \ dfrac {iu} {2} It has become.

2.5.2 Algorithm

Determines if any point is included in the limit set. Utilizing the fact that parallelogram regions repeat on the left and right, and point symmetry on the top and bottom with Line 3 as the boundary, the judgment at each point is finally brought to the judgment of the figure in the lower half of the center. ..

We will process a certain point as follows.

- If it is to the right of Line 1, move it to the left to fit within the parallelogram range
- If it's to the left of Line2, move to the right as well
- If it is above Line3, use the point symmetry and move it to the target point so that it is below Line3.

- Apply transformation *a* to points below Line3

The largest circle tangent to the straight line y = 0 is the inverted line y = u in the unit circle. When the conversion *a* is applied to the points in this, y <0 and it deviates from the band of 0 \ leq y \ leq u . Therefore,

> When a point is multiplied by the transformation *a* , y <0 = A point is included in this largest circle = It is included in the limit set

Judgment is made as.

On the contrary, what if it is not included? Even if the above procedure is repeated, the band of 0 \ leq y \ leq u cannot be removed, and finally the movement of two points across Line 3 will be repeated. Therefore, if the points are the same as two points before, it can be judged that the points are not included in the limit set.

In summary, it will be processed as follows.

1. Out of range if y <0 , u <y
2. Move to the center parallelogram if it is to the right of Line1 and to the left of Line2
3. Invert about center point if above Line 3
4. Apply transformation *a*
5. If y <0, it is judged as the limit set
6. If it is the same as the previous point, it is judged that it is not the limit set
7. If neither, go back to 2.

2.6 Implementation

Let's take a look at the code.

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);
}
```

On the C # side, it is `OnRenderImage()`only a process to draw the material while passing the parameters set in the inspector in KleinianGroup.cs .

Let's take a look at the shader.

KleinianGroup.shader

```
#pragma vertex vert_img
```

The Vertex shader uses Unity standard vert_img. The main is the Fragment shader. There are 3 Fragment shaders, each with a different path. The first is the standard one, the second is the one with the blurring process and the appearance is a little cleaner, and the third is the one with the further circle inversion described later. You can now choose which path to use in KleinianGroup.cs. Let's look at the first one here.

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`The aspect ratio is calculated from and multiplied by pos.x. Now the area on the screen represented by pos is $0 \leq y \leq 1$, and x is the range according to the aspect ratio. Furthermore , the position and range to be displayed can be adjusted by applying `_Offset`, passed from the C # side `_Scale`. `josKleinian()`The color to be output is determined by judging whether or not the limit set is possible.

`josKleinian()`Let's take a closer look.

KleinianGroup.shader

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

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

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

A function that receives the point z and the Mobius transformation parameter t and determines whether z is included in the limit set. lz and llz are variables for judging "the same point as before" indicating that they are outside the set. For the time being, the values are initialized so that they are different from z at the start and also different from each other. `_Iteration`Is the maximum number of times to repeat the procedure. I think that it is enough that the value is not so large unless you look at the details in a magnified manner.

KleinianGroup.shader

```
// wrap if outside of Line1,2
float offset_x = abs (v) / u * zy;
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;
}
```

Here is

> Move to the center parallelogram if it is to the right of Line1 and to the left of Line2

It will be the part of.

`wrap()`Is a function that receives the position of the point, the width of the rectangle, and the coordinates of the left edge of the rectangle, and stores the points that extend to the left and right. It is a process to convert a parallelogram to a rectangle with `wrap()`offset_x, put it within the range with offset_x, and return it to a parallelogram again with offset_x.

Next is the judgment of Line3.

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`The part that asks for is the conditional expression of Line3. I don't know how to derive this part, and I think it is roughly calculated from the symmetry of the figure. Depending on the value of t that has been squeezed into a complicated figure, the upper and lower figures may mesh with each other in a jagged manner, and this conditional expression may not

be sufficient to divide it properly, but this time it is effective in a general form. Will be used as it is.

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);
}
```

Finally , apply the Mobius transformation *a* to the point z . Using the above formula transformation to make it easier to code,

```
a:       z\rightarrow       \dfrac       {tz-i}{-iz}=\dfrac{x-iy}
{\left|z\right|^{2}}+(-v+iu)
```

I'm implementing this.

KleinianGroup.shader

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

As a result of conversion, if y <0, limit set judgment,

KleinianGroup.shader

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

llz=lz;
lz=z;
```

If the value is almost the same as the point two points before, it is judged that it is not the limit set. Also, `_Iteration`if the judgment result is not obtained even if it is repeated, it is judged that it is not the limit set.

This completes the shader implementation. The parameter t is (2,0) is the most with a typical value of (1.94,0.02) is likely to be interesting shape in

the vicinity. The sample project can be edited in the inspector of KleinianGroupDemo.cs, so please play with it.

2.7 Further circle inversion

That's all for displaying the limit set, but to make it interesting as an animation, we will add a powerful topping at the end. `josKleinian()`Invert the position in a circle before passing it over. Circle inversion swaps the infinitely expanding area outside the circle with the inside, and the circle is transferred as a circle. And the limit set is made up of innumerable circles. By moving this inverted circle or changing the radius, you can create a mysterious appearance that you cannot predict while taking advantage of the fun of fractals.

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;
}
```

This is a Fragment shader with the circular inversion defined in the third path. `sample_num`The loop of is a process to make the appearance beautiful by calculating the surrounding pixels and blurring it a little. `calc_color()`However, it is the process of calculating the color so far, `ciecleInverse()`and the circle is inverted before that .

In the KleinianGroupCircleInverse scene, fractal-like animation works by changing the parameters of this shader with Animator.

2.8 Summary

In this chapter, we introduced how to draw the limit set of Kleinian groups with a shader and how to make more interesting fractal figures by using circle inversion. Fractal and Mobius transformations were difficult to get to in fields that I wasn't familiar with, but it was very exciting to see unexpected patterns moving one after another. If you like, please try it out!

2.9 Reference

- Indra's pearl * 3 The
  strongest bible. It's a little pricey and the content is advanced, so if you want to understand more deeply by looking at the following, it is recommended to pick it up.

- Mathe Vital * 4
  Indra's pearl essence is extracted and introduced in an easy-to-understand manner. I think it's a good idea to start from here.

- Jos Leys' algorithm introduced this time * 5

- Jos Leys' Shadertoy * 6
  An example of a valuable shader implementation

- Morph * 7
  @ soma_arc * 8 was announced at TokyoDemoFest2018 * 9 . This time, it was an opportunity for me to learn the Kleinian group myself. This is also a valuable shader implementation example.

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

[*4] http://userweb.pep.ne.jp/hannyalab/MatheVital/IndrasPearls/IndrasPearlsindex.html

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

[*6] https://www.shadertoy.com/user/JosLeys

[*7] https://www.shadertoy.com/view/MlGfDG

[*8] https://twitter.com/soma_arc

[*9] http://tokyodemofest.jp/2018/

Chapter 3 GPU-Based Cloth Simulation

3.1 Introduction

Simulation of the shape of a planar object that is deformed by an external force such as a flag or clothes is called **Cloth Simulation** , and much research has been done in the CG field as it is essential for animation creation. .. Already implemented in Unity, this chapter introduces a simple cloth simulation theory and GPU implementation for the purpose of learning parallel computing using GPU and understanding the nature of simulation and the meaning of parameters. I will do it.

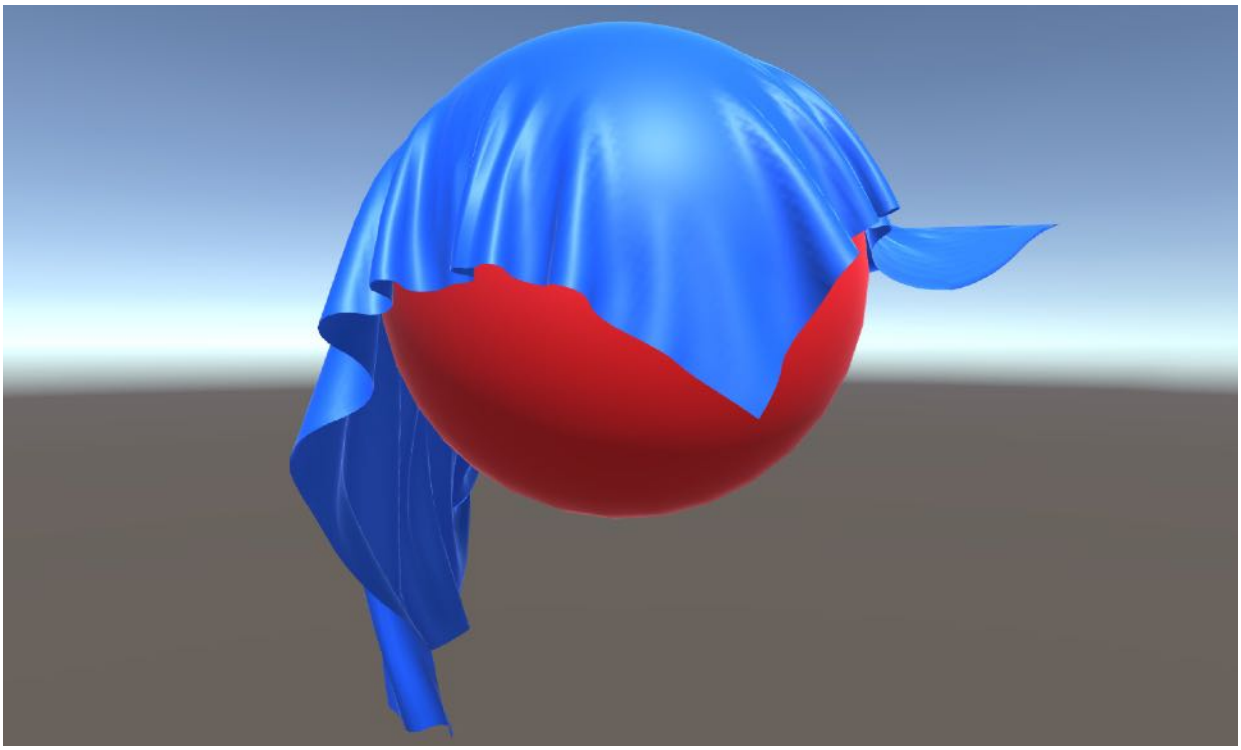

Figure 3.1: Cloth simulation

3.2 Algorithm explanation

3.2.1 Mass-Spring System

An object such as a spring, rubber, or cushion that deforms when a force is applied and returns to its original shape when the force is stopped is called an **elastic body** . Since such elastic bodies cannot be represented by a single position or orientation, they represent an object by points and connections between them, and the movement of each point simulates the entire shape. This point is called a **mass point** and is considered to be a mass of mass without size. In addition, the connection between mass points has the property of a spring. The method of simulating an elastic body by calculating the expansion and contraction of each spring is called the mass-spring system, and it is a flag by calculating the motion of a set of mass points arranged in a two-dimensional shape. Simulation of clothes etc. is called Cloth Simulation.

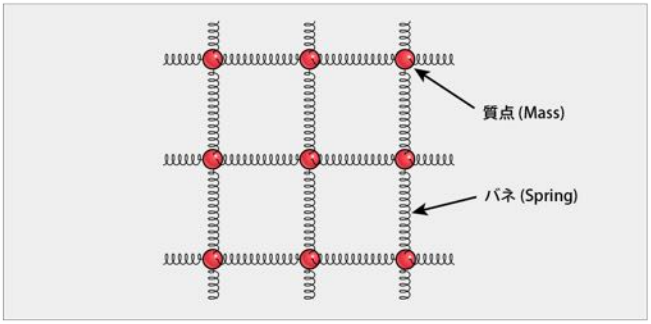

Figure 3.2: Mass-Spring system

3.2.2 Algorithm used for cloth simulation

Spring force

Each spring applies a force to the connected mass points according to the following equation.

```
F_{spring} = -k \left( I-I_{0} \right) - b v
```

Here, I is the current length of the spring (distance between the connected mass points), and I_ {0} is the natural length of the spring at the start of the

simulation (the length when no load is applied to the spring).). k is a constant that represents the hardness of the spring, v is the velocity of the mass point, and b is the constant that determines the degree of velocity attenuation. This equation means that the spring always exerts a force that tries to return the distance between the connected mass points to the initial natural length of the spring. If the current distance of the spring is significantly different from the natural length of the spring, a larger force will be applied and it will be attenuated in proportion to the current velocity of the mass point.

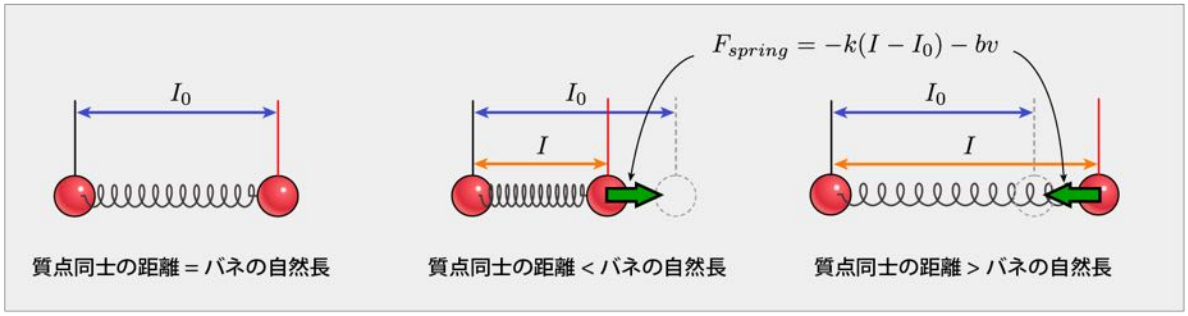

Figure 3.3: Spring force

Spring structure

In this simulation, the springs that make up the basic structure are connected in the horizontal and vertical directions, and the springs are also connected between the mass points located diagonally to prevent extreme deviation in the diagonal direction. They are called Structure Spring and Shear Spring, respectively, and one mass point connects the spring to 12 adjacent mass points, respectively.

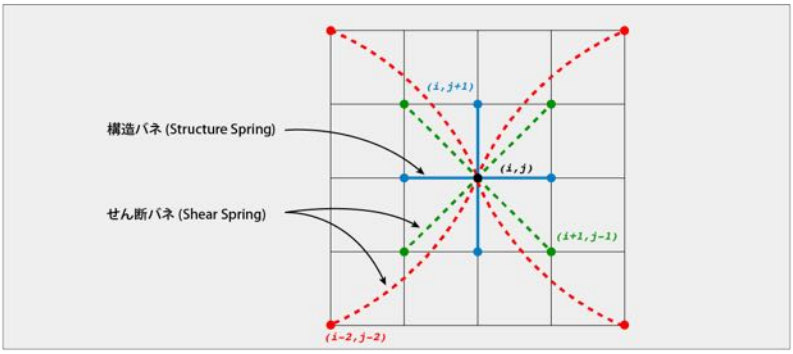

Figure 3.4: Spring structure

Position calculation by Verlet Method

In this simulation, the position of the mass point is calculated by the Verlet method, which is a method often used in real-time applications. The Verlet method is one of the numerical solutions of Newton's equation of motion, and is a method used in molecular dynamics to calculate the movement of atoms. In calculating the motion of the object, but usually seek position from the speed, the Belle method, **the current position** and, **the position of the previous time step** from **the position at the next time step** will seek.

The derivation of the Verlet algebraic equation is shown below. F is the force applied to the mass point , m is the mass of the mass point, v is the velocity, x is the position, t is the time, and \ Delta t is the time step of the simulation (how much time is advanced per simulation calculation).) Then, the equation of motion of the mass point is

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

Can be written. If this equation of motion is made into an algebraic equation using the following two Taylor expansions,

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

```
x\left( t-\Delta t\right) =x\left( t\right) -\Delta t\dfrac
{dx\left( t\right) }{dt}+\dfrac {1}{2!}\Delta t^{2}\dfrac
```

```
{dx^{2}\left(     t\right)     }{dt^{2}}-\dfrac     {1}{3!}\Delta t^{3}\dfrac {dx^{3}\left( t\right) }{dt^{3}}+\ldots
```

It will be. From these two Taylor expansion equations, if we solve the second-order derivative term and ignore the terms of \ Delta t of the second order or higher as sufficiently small, we can write as follows.

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

If the second-order differential term is expressed by mass m and force F from the equation of motion ,

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

The algebraic equation is obtained. In this way, the formula for calculating the position in the next time step from the current position, the position in the previous time step, the mass, the force, and the value of the time step can be obtained.

The speed is calculated from the current position and the previous position in time.

```
v\left( t\right) =\dfrac {x\left( t\right) -x\left( t-\Delta t\right) }{\Delta t}
```

The speed obtained by this calculation is not very accurate, but it is not a problem as it is only used to calculate the damping of the spring.

Collision calculation

Calculating the collision of cloth and sphere

Collision processing is performed in two phases: "collision detection" and "reaction to it".

Collision detection is performed by the following formula.

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

c and r are the sphere of **center** and **radius** , x \ left (t + \ Delta t \. right) is the position at the next time step is determined by Belle method. If a collision is detected, move the mass point onto the surface of the sphere to prevent the sphere from colliding with the cloth. Specifically, this is done by shifting the mass points inside the sphere in the direction normal to the surface of the collision point. The position of the mass point is updated according to the following formula.

```
\begin{aligned}
d=\dfrac {x\left( t+\Delta t\right) -c}{\left\| x\left( t+\Delta
t\right) -c\right\|}
\\
x^{\prime}\left(t + \Delta t\right) = c + dr
\end{aligned}
```

x ^ {\ prime} \ left (t + \ Delta t \ right) is the updated position after the collision. d can be regarded as an approximation of the acceptable accuracy of the normal to the surface at the collision point, provided that the mass does not penetrate deeply.

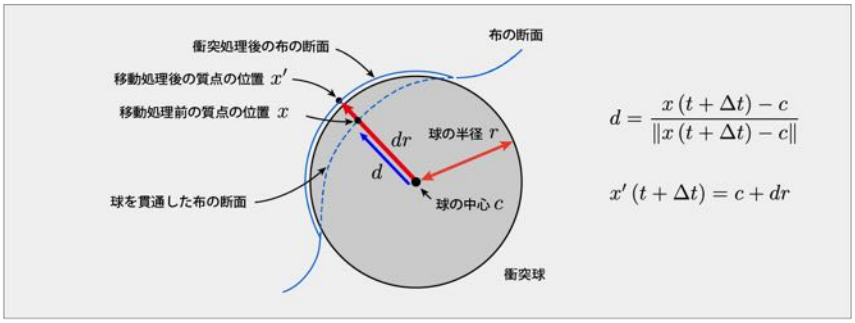

Figure 3.5: Collision calculation

3.3 Sample program

3.3.1 Repository

The sample program is in the **Assets / GPUClothSimulation** folder in the repository below .

https://github.com/IndieVisualLab/UnityGraphicsProgramming4

Execution condition

This sample program uses the Compute Shader. Please check here for the operating environment of Compute Shader.

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

3.4 Implementation description

3.4.1 Outline of processing

It is a schematic diagram showing how each component and code work in relation to each other.

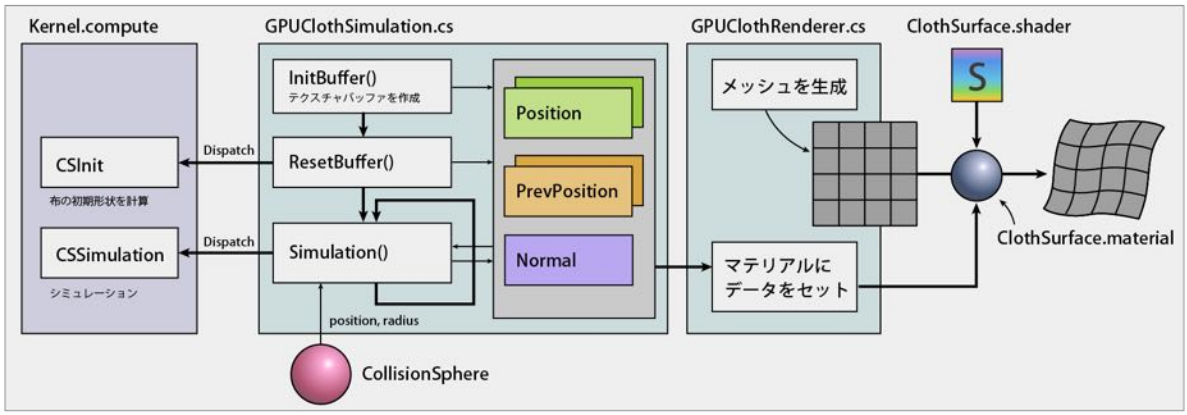

Figure 3.6: Structure and processing flow of each component and code

GPUClothSimulation.cs is a C # script that manages the data and processing used for simulation. This script creates and manages position and normal data used for simulation in RenderTexture format. In addition, processing is performed by calling the kernel described in Kernels.compute. The GPUClothRenderer.cs script provides visualization of the calculation results. The Mesh object generated by this script is drawn by transforming the geometry by the processing of ClothSurface.shader that refers to the RenderTexture that stores the position data and normal data that are the calculation results.

GPUClothSimulation.cs

A C # script that controls the simulation.

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

public class GPUClothSimulation : MonoBehaviour
{
    [Header("Simulation Parameters")]
    // Time step
    public float   TimeStep = 0.01f;
    // Number of simulation iterations
    [Range(1, 16)]
    public int VerletIterationNum = 4;
    // Cloth resolution (horizontal, vertical)
      public  Vector2Int  ClothResolution  =  new  Vector2Int(128,
128);
    // Cloth grid spacing (natural length of spring)
    public float   RestLength = 0.02f;
    // Constants that determine the elasticity of the fabric
    public float   Stiffness = 10000.0f;
    // Velocity decay constant
    public float   Damp = 0.996f;
    // quality
    public float   Mass = 1.0f;
    // gravity
    public Vector3 Gravity = new Vector3(0.0f, -9.81f, 0.0f);

    [Header("References")]
    // Reference to Transform of collision sphere
    public Transform CollisionSphereTransform;
    [Header("Resources")]
    // Kernel to simulate
    public ComputeShader KernelCS;

    // Cloth simulation position data buffer
    private RenderTexture[] _posBuff;
      //  Cloth  simulation  position  data  (previous  time  step)
buffer
    private RenderTexture[] _posPrevBuff;
    // Cloth simulation normal data buffer
    private RenderTexture _normBuff;

    // Cloth length (horizontal, vertical)
    private Vector2 _totalClothLength;
```

```
    [Header("Debug")]
    // Show simulation buffer for debugging
    public bool EnableDebugOnGUI = true;
    // Buffer display scale during debug display
    private float _debugOnGUIScale = 1.0f;

    // Did you initialize the simulation resource?
    public bool IsInit { private set; get; }

    // Get the position data buffer
    public RenderTexture GetPositionBuffer()
    {
        return this.IsInit ? _posBuff[0] : null;
    }
    // Get a buffer of normal data
    public RenderTexture GetNormalBuffer()
    {
        return this.IsInit ? _normBuff : null;
    }
    // Get the resolution of the cloth
    public Vector2Int GetClothResolution()
    {
        return ClothResolution;
    }

    // Number of X, Y dimensional threads in the Compute Shader
kernel
    const int numThreadsXY = 32;

    void Start()
    {
        var w = ClothResolution.x;
        var h = ClothResolution.y;
        var format = RenderTextureFormat.ARGBFloat;
         var filter = FilterMode.Point; // Prevent interpolation
between texels
        // Create RenderTexture to store data for simulation
           CreateRenderTexture(ref _posBuff,      w, h, format,
filter);
            CreateRenderTexture(ref _posPrevBuff, w, h, format,
filter);
           CreateRenderTexture(ref _normBuff,     w, h, format,
filter);
        // Reset the data for simulation
        ResetBuffer();
        // Set the initialized flag to True
        IsInit = true;
```

```
    }

    void Update()
    {
        // Press the r key to reset the simulation data
        if (Input.GetKeyUp("r"))
            ResetBuffer();

        // Perform a simulation
        Simulation();
    }

    void OnDestroy()
    {
        // Removed RenderTexture that stores data for simulation
        DestroyRenderTexture(ref _posBuff    );
        DestroyRenderTexture(ref _posPrevBuff);
        DestroyRenderTexture(ref _normBuff   );
    }

    void OnGUI ()
    {
           // Draw RenderTexture containing simulation data for
debugging
        DrawSimulationBufferOnGUI();
    }

    // Reset simulation data
    void ResetBuffer()
    {
        ComputeShader cs = KernelCS;
        // Get kernel ID
        int kernelId = cs.FindKernel("CSInit");
          // Calculate the number of execution thread groups in
the Compute Shader kernel
        int groupThreadsX =
                        Mathf.CeilToInt((float)ClothResolution.x  /
numThreadsXY);
        int groupThreadsY =
                        Mathf.CeilToInt((float)ClothResolution.y  /
numThreadsXY);
        // Calculation of cloth length (horizontal, vertical)
        _totalClothLength = new Vector2(
            RestLength * ClothResolution.x,
            RestLength * ClothResolution.y
        );
        // Set parameters and buffers
        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);
        // run the kernel
        cs.Dispatch(kernelId, groupThreadsX, groupThreadsY, 1);
        // copy the buffer
        Graphics.Blit (_posBuff [0], _posBuff [1]);
        Graphics.Blit (_posPrevBuff [0], _posPrevBuff [1]);
    }

    // simulation
    void Simulation()
    {
        ComputeShader cs = KernelCS;
         // Calculation of CSSimulation Calculation of the value
of the time step per time
        float timestep = (float)TimeStep / VerletIterationNum;
        // Get kernel ID
        int kernelId = cs.FindKernel("CSSimulation");
          // Calculate the number of execution thread groups in
the Compute Shader kernel
        int groupThreadsX =
                    Mathf.CeilToInt((float)ClothResolution.x  /
numThreadsXY);
        int groupThreadsY =
                    Mathf.CeilToInt((float)ClothResolution.y  /
numThreadsXY);

        // set parameters
        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
});
```

```
        // Set the parameters of the collision sphere
        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 ++)
        {
            // set the buffer
                    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);
            // run thread
             cs.Dispatch(kernelId, groupThreadsX, groupThreadsY,
1);
            // Swap the read buffer and write buffer
               SwapBuffer(ref _posBuff[0],      ref _posBuff[1]
);
              SwapBuffer (ref _posPrevBuff [0], ref _posPrevBuff
[1]);
        }
    }

    // Create RenderTexture to store data for simulation
     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();
    }

    // Create RenderTexture [] to store data for simulation
     void CreateRenderTexture(ref RenderTexture[] buffer, int w,
int h,
        RenderTextureFormat format, FilterMode filter)
    {
        // ~ slightly~
    }

    // Removed RenderTexture that stores data for simulation
    void DestroyRenderTexture(ref RenderTexture buffer)
    {
        // ~ slightly~
    }

    // Remove RenderTexture [] to store data for simulation
    void DestroyRenderTexture(ref RenderTexture[] buffer)
    {
        // ~ slightly~
    }

    // Delete material
    void DestroyMaterial(ref Material mat)
    {
        // ~ slightly~
    }

    // Swap buffers
      void  SwapBuffer(ref  RenderTexture  ping,  ref  RenderTexture
pong)
    {
        RenderTexture temp = ping;
        ping = pong;
        pong = temp;
    }

     // Draw a buffer for simulation in the OnGUI function for
debugging
```

```
    void DrawSimulationBufferOnGUI()
    {
        // ~ slightly~
    }
}
```

At the beginning, the parameters required for the simulation are declared. In addition, RenderTexture is used to hold the simulation results. The data used and obtained for this simulation

1. position
2. Position in the previous time step
3. Normal

is.

InitBuffer function

The InitBuffer function creates a RenderTexture that stores the data needed for the calculation. For the position and the position in the previous time step, the data in the previous time step is used and the calculation is performed based on it, so create two for reading and one for writing. In this way, the method of creating data for reading and data for writing and letting the shader calculate efficiently is called **Ping Pong Buffering** .

Regarding the creation of RenderTexture, in format, the precision of the texture (the number of channels and the number of bits of each channel) is set. Generally, the lower the value, the faster the processing, but ARGBHalf (16bit per channel) has low accuracy and the calculation result becomes unstable, so set it to ARGBFloat (32bit per channel). Also, enableRandomWrite should be true to allow ComputeShader to write the calculation result. RenderTexture is not created on the hardware just by calling the constructor, so execute the Create function to make it available in the shader.

ResetBuffer function

The ResetBuffer function initializes the RenderTexture that stores the data needed for the simulation. Get kernel ID, calculate number of thread groups,

set various parameters such as cloth length, RenderTexture used for calculation for Compute Shader, and call CSInit kernel written in Kernels.compute for processing. .. The contents of the CSInit kernel are described in the following Kernels.compute details.

Simulation function

The Simulation function simulates the actual cloth. At the beginning, like the ResetBuffer function, get the kernel ID, calculate the number of thread groups, set various parameters used for simulation and RenderTexture. If you calculate with a large time step at a time, the simulation becomes unstable, so in Update (), divide the time step into small values so that the simulation can be calculated stably by dividing it into several times. I will. The number of iterations is set in **VerletIterationNum** .

Kernels.compute

Compute Shader that describes processing such as actual simulation.

This Compute Shader has

1. CSInit
2. CSSimulation

There are two kernels.

Each kernel performs the following processing.

CSInit

Calculates the initial values of position and normal. The position of the mass point is calculated so that it is arranged in a grid pattern on the XY plane based on the thread ID (2D).

CSSimulation

Perform a simulation. The figure below outlines the processing flow of the CSSimulation kernel.

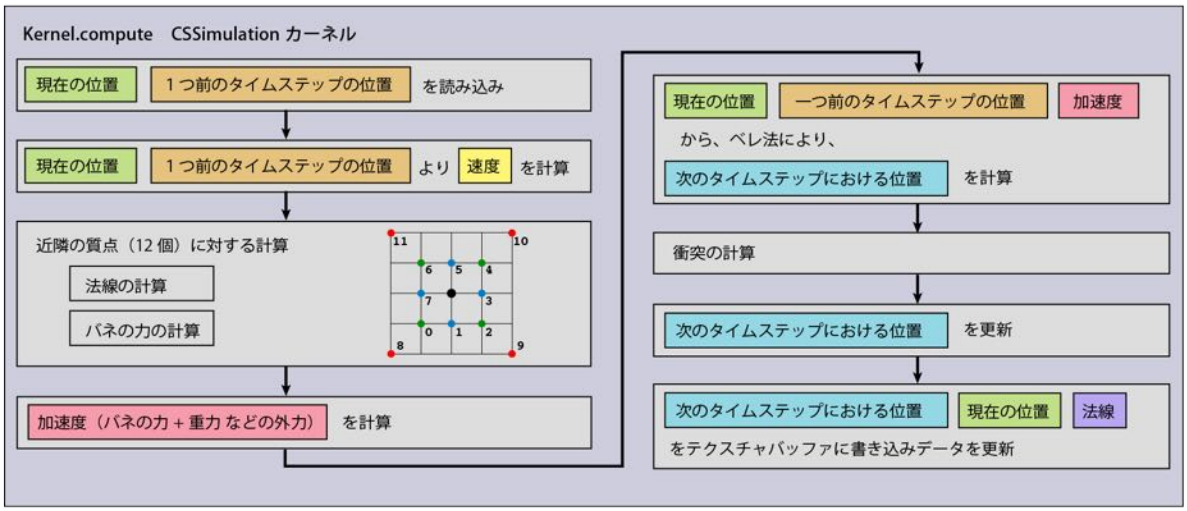

Figure 3.7: Calculation flow in CSSimulation kernel

The code is shown below.

```
#pragma kernel CSInit
#pragma kernel CSSimulation

#define NUM_THREADS_XY 32 // Number of kernel threads

// For reading position data (previous time step)
Texture2D<float4>   _PositionPrevBufferRO;
// For reading position data
Texture2D<float4>   _PositionBufferRO;
// For writing position data (previous time step)
RWTexture2D<float4> _PositionPrevBufferRW;
// For writing position data
RWTexture2D<float4> _PositionBufferRW;
// For writing normal data
RWTexture2D<float4> _NormalBufferRW;

int2 _ClothResolution; // Cloth resolution (number of particles)
(horizontal, vertical)
float2 _TotalClothLength; // Overall length of the cloth

float _RestLength; // Natural length of the spring

float3 _Gravity; // Gravity
float  _Stiffness;  //  Constant  that  determines  the  degree  of
expansion and contraction of the cloth
float _Damp; // Attenuation rate of cloth speed
```

```
float _InverseMass; // 1.0/quality

float _TimeStep; // Size of time step

bool _EnableCollideSphere; // Flag for collision handling
float4 _CollideSphereParams; // Collision handling parameters
(pos.xyz, radius)

// Array of ID offsets (x, y) of nearby particles
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
};
// Returns the offset of the ID of nearby particles
int2 NextNeigh(int n)
{
    return m_Directions[n];
}

// Kernel that initializes the simulation buffer
[numthreads(NUM_THREADS_XY, NUM_THREADS_XY, 1)]
void CSInit(uint3 DTid : SV_DispatchThreadID)
{
    uint2 idx = DTid.xy;

    // location
        float3 pos = float3(idx.x * _RestLength, idx.y *
_RestLength, 0);
    pos.xy -= _TotalClothLength.xy * 0.5;
    // normal
    float3 nrm = float3 (0, 0, -1);
    // write to buffer
    _PositionPrevBufferRW[idx] = float4(pos.xyz, 1.0);
    _PositionBufferRW[idx]     = float4(pos.xyz, 1.0);
    _NormalBufferRW[idx]       = float4(nrm.xyz, 1.0);
}
```

```
// Kernel to simulate
[numthreads(NUM_THREADS_XY, NUM_THREADS_XY, 1)]
void CSSimulation(uint2 DTid : SV_DispatchThreadID)
{
    int2 idx = (int2)DTid.xy;
      // Cloth resolution (number of particles) (horizontal,
vertical)
    int2   res = _ClothResolution.xy;
    // read position
    float3 pos = _PositionBufferRO[idx.xy].xyz;
    // Read position (previous time step)
    float3 posPrev = _PositionPrevBufferRO[idx.xy].xyz;
    // Calculate the speed from the position and the position of
the previous time step
    float3 vel = (pos - posPrev) / _TimeStep;

    float3 normal   = (float3)0; // 法線
      float3 lastDiff = (float3) 0; // Variable for storing
direction vector used when calculating normal
     float iters = 0.0; // Variable for adding the number of
iterations when calculating normals

     // Substitute the force applied to the particles and the
value of gravity as the initial value
    float3 force = _Gravity.xyz;
    // 1.0 / quality
    float  invMass = _InverseMass;

    // If it is the top side of the cloth, do not calculate to
fix the position
    if (idx.y == _ClothResolution.y - 1)
        return;

    // Calculate for nearby particles (12)
    [unroll]
    for (int k = 0; k < 12; k++)
    {
        // Offset of ID (coordinates) of neighboring particles
        int2 neighCoord = NextNeigh(k);
        // Do not calculate for X-axis, edge particles
         if (((idx.x+neighCoord.x) < 0) || ((idx.x+neighCoord.x)
> (res.x-1)))
            continue;
        // Do not calculate for Y-axis, edge particles
              if (((idx.y + neighCoord.y) <0) || ((idx.y +
neighCoord.y)> (res.y-1)))
            continue;
        // ID of nearby particles
```

```
          int2    idxNeigh = int2(idx.x + neighCoord.x, idx.y +
neighCoord.y);
        // Position of nearby particles
        float3 posNeigh = _PositionBufferRO[idxNeigh].xyz;
        // Difference in the position of nearby particles
        float3 posDiff = posNeigh - pos;

        // Normal calculation
            // Direction vector from the base point to nearby
particles
        float3 currDiff = normalize(posDiff);
        if ((iters > 0.0) && (k < 8))
        {
              // With the direction vector with the neighboring
particles examined one time ago
            // If the current angle is obtuse
            float a = dot(currDiff, lastDiff);
            if (a > 0.0) {
                    // Find and add orthogonal vectors by cross
product
                normal += cross(lastDiff, currDiff);
            }
        }
         lastDiff = currDiff; // Keep for calculation with next
neighbor particles

           // Calculate the natural length of the spring with
neighboring particles
        float  restLength = length(neighCoord * _RestLength);
        // Calculate the force of the spring
                        force   +=   (currDiff*(length(posDiff)-
restLength))*_Stiffness-vel*_Damp;
        // Addition
        if (k < 8) iters += 1.0;
    }
    // Calculate normal vector
    normal = normalize (normal / - (iters - 1.0));

    // acceleration
    float3 acc = (float3)0.0;
    // Apply the law of motion (the magnitude of acceleration is
proportional   to   the   magnitude   of   force   and   inversely
proportional to mass)
    acc = force * invMass;

    // Position calculation by Verlet method
    float3 tmp = pos;
    pos = pos * 2.0 - posPrev + acc * (_TimeStep * _TimeStep);
```

```
    posPrev = tmp; // Position of the previous time step

    // Calculate collision
    if (_EnableCollideSphere)
    {
            float3 center = _CollideSphereParams.xyz; // Center
position
        float  radius = _CollideSphereParams.w;   // 半径

        if (length(pos - center) < radius)
        {
             // Calculate the unit vector from the center of the
collision sphere to the position of the particles on the cloth
            float3 collDir = normalize(pos - center);
             // Move the position of particles to the surface of
the collision sphere
            pos = center + collDir * radius;
        }
    }

    // write
    _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

In this component

1. Get or create MeshRenderer
2. Generate a Mesh object that matches the resolution of the cloth simulation
3. Substitute simulation results (position, normal) for the material that draws the mesh

to hold.

Please check the sample code for details.

ClothSurface.shader

In this shader, the position and normal data obtained by simulation are acquired in the vertex shader, and the shape of the mesh is changed by rewriting the vertices.

Please check the sample code for details

3.4.2 Execution result

When you run it, you'll see an object that behaves like a cloth that collides with the sphere. If you change the various parameters used in the simulation, you can see the change in the movement.

TimeStep is the time of the simulation that progresses when the Update function is executed once. If you increase it, the change in movement will be large, but if you set a value that is too large, the simulation will become unstable and the value will diverge.

VerletIterationNum is the number of CSSimulation kernels to execute in the Simulation function. Even if the value of the same time step is increased, increasing this value will make the simulation easier to stabilize, but will increase the calculation load.

ClothResolution is the resolution of the cloth. If you increase it, you will see many details such as wrinkles, but if you increase it too much, the simulation will become unstable. Since the thread size is set to 32 on ComputeShader, it is desirable to be a multiple of 32.

RestLength is the natural length of the spring. Since it is the distance between the springs, the length of the cloth is Cloth Resolution x Rest Length.

Stiffness is the hardness of the spring. Increasing this value will reduce the stretch of the fabric, but increasing it too much will make the simulation unstable.

Damp is the attenuation value of the moving speed of the spring. Increasing this value will make the mass point stagnant faster and less likely to vibrate, but will reduce the change.

Mass is the mass of the mass point. Increasing this value will cause the cloth to move as if it were heavy.

Gravity is the gravity on the cloth. The acceleration of the cloth is the combination of this gravity and the force of the spring.

If you check **EnableDebugOnGUI** , a texture that stores the position that is the result of the simulation, the position in the previous time step, and the normal data is drawn in the upper left of the screen for confirmation.

When you press the **R** key, the cloth returns to its initial state. If the simulation becomes unstable and the values diverge, please return to the initial state.

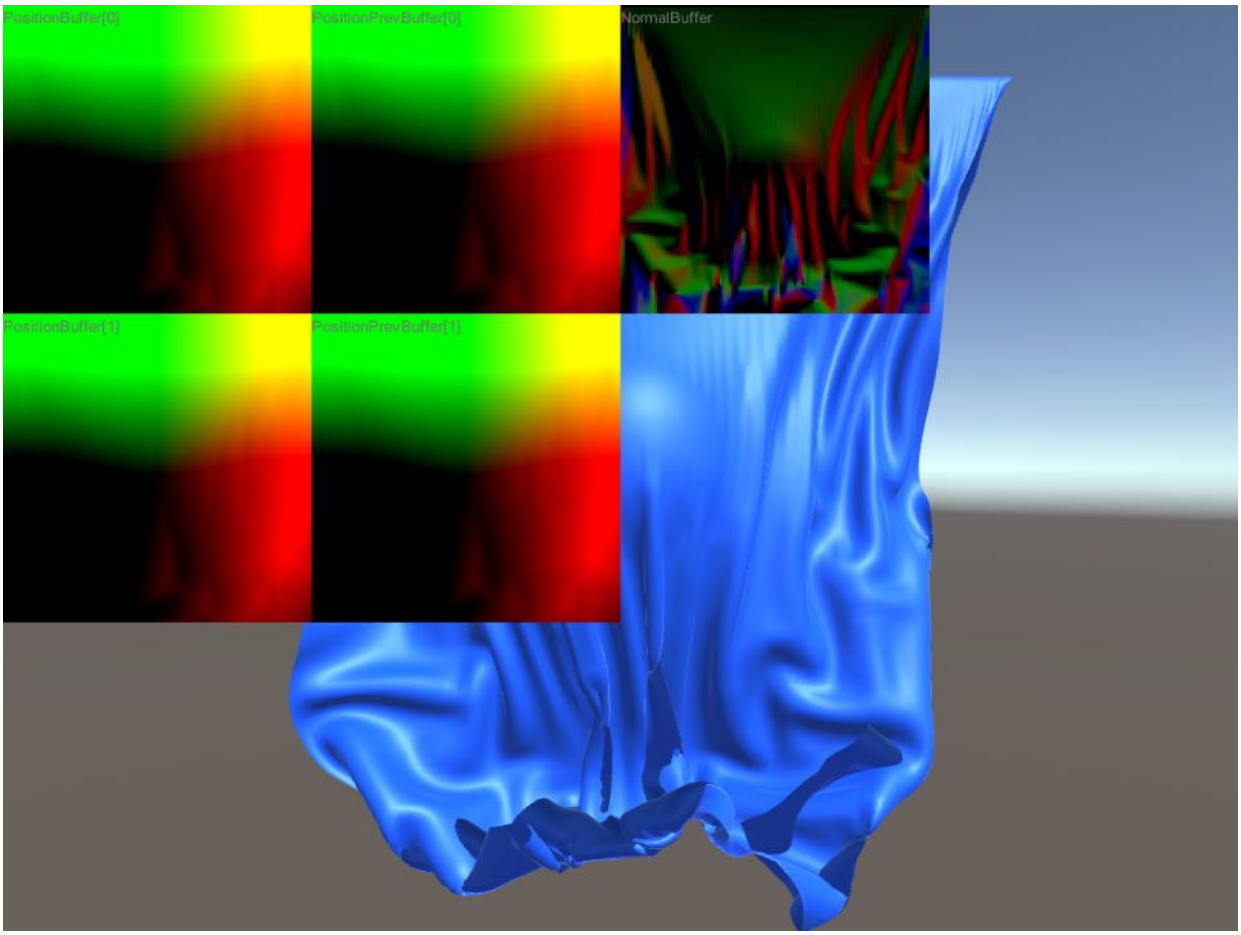

Figure 3.8: Execution result (calculation result RenderTexture is drawn in screen space)

Draw springs as debug

If you open Assets / GPUClothSimulation / Debug / **GPUClothSimulationDebugRender.unity** , you can see the spring that connects the mass **points** with particles and lines.

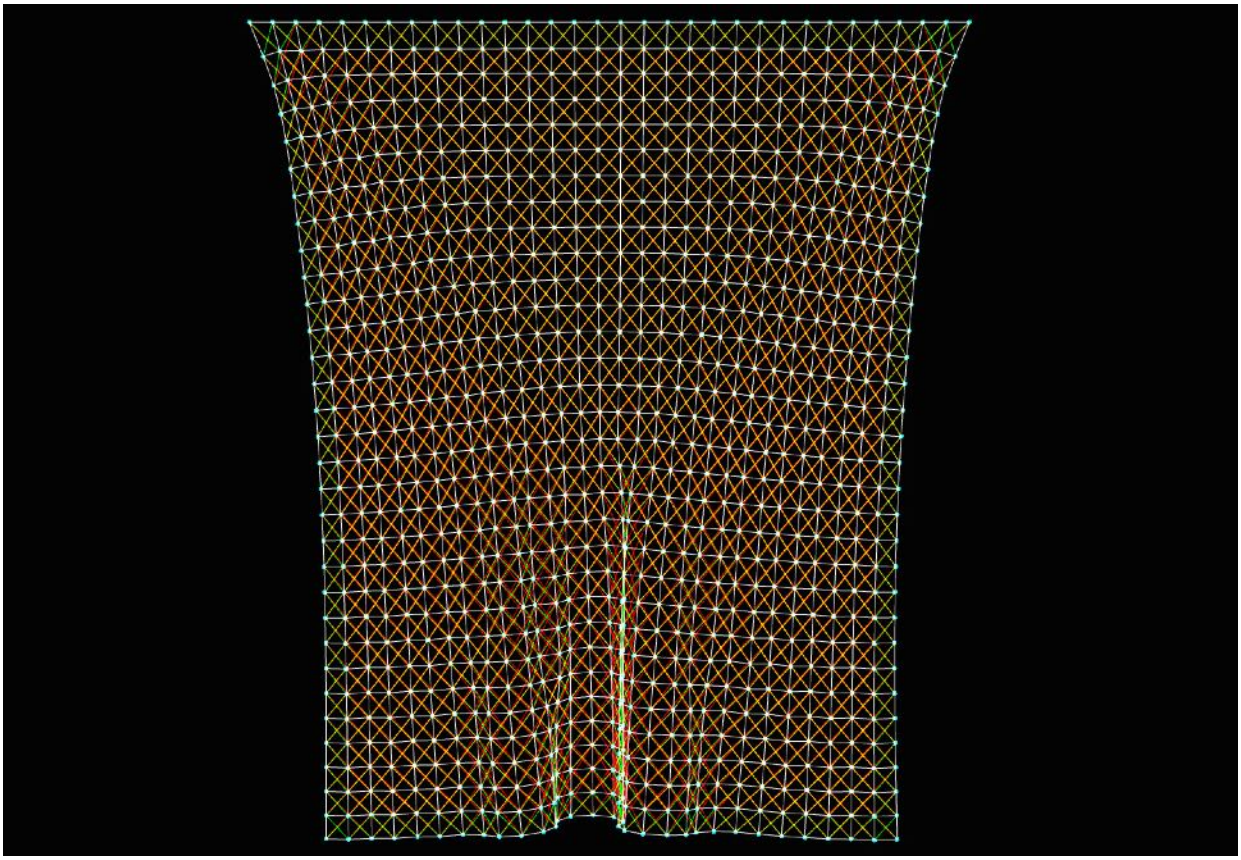

Figure 3.9: Draw mass points and springs with particles and lines

3.5 Summary

The movement obtained by the mass point-spring system simulation changes in a complicated manner depending on how the force is applied, and an interesting shape can be created. The cloth simulation presented in this chapter is very simple. Challenges such as collisions with objects of complex geometry rather than simple things like spheres, collisions between cloths, friction, consideration of the fiber structure of cloths, stability of simulations

when taking large time steps, etc. Much research has been done to overcome it. If you are interested in physics engine, why not pursue it?

3.6 Reference

- [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] Makoto Fujisawa, Basics of Physical Simulation for CG, Mynavi Corporation, 2013
- [4] Yukiichi Sakai, Physical Simulation by WebGL, Engineering Co., Ltd., 2014
- [5] Koichi Sakai, Introduction to Dynamic Animation Made with OpenGL, Morikita Publishing Co., Ltd., 2005

Chapter 4 StarGlow

Figure 4.1: Rays extending from bright areas

LightLeak, LightStreak, or StarGlow, which stretches when a strong light is reflected, let's express this with a post effect. Here, for convenience, it is called StarGlow.

This post-effect presented here was presented by Masaki Kawase at GDC 2003.

The sample in this chapter is "Star Glow" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming4
.

4.1 STEP 1: Generate a brightness image

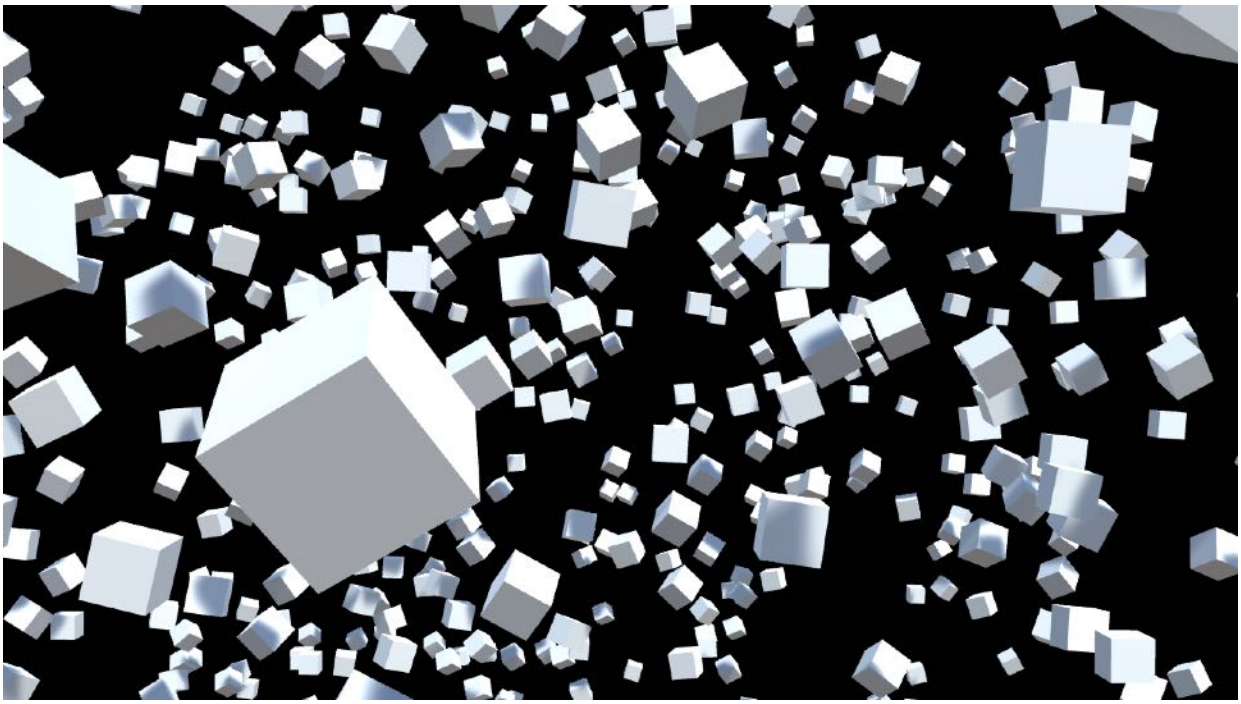

Figure 4.2: Original image

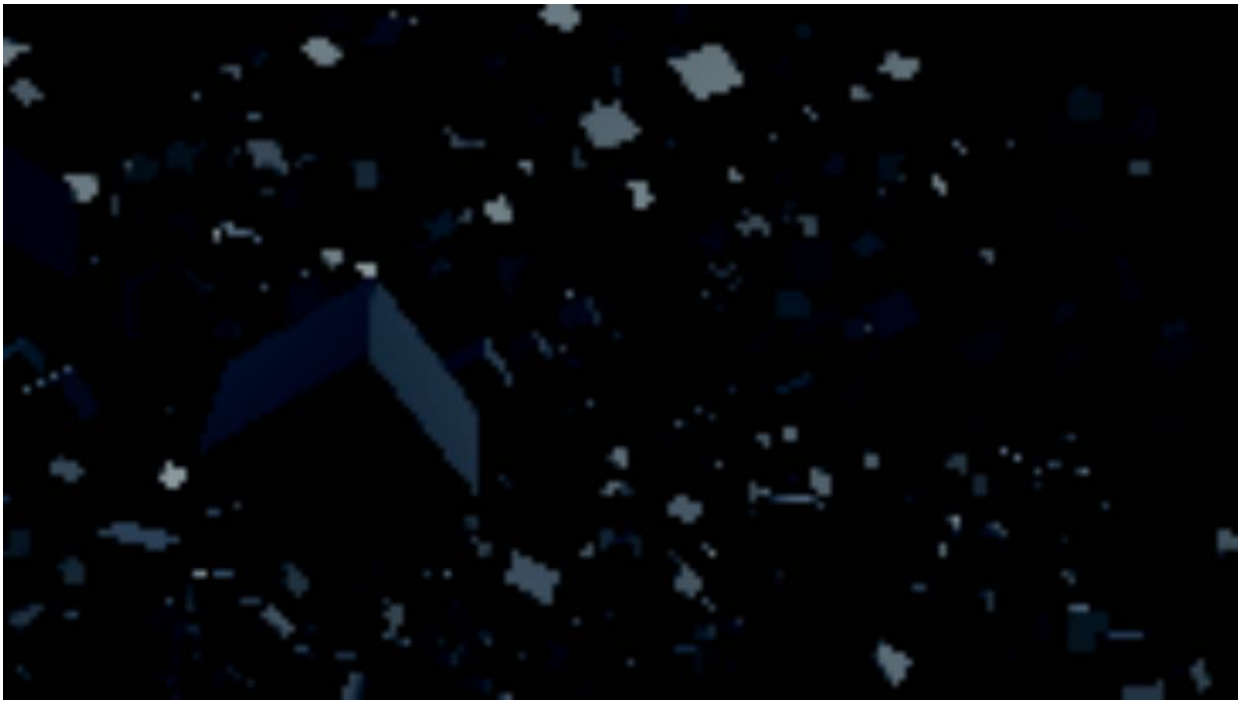

Figure 4.3: Image of detecting only high-brightness pixels

First, let's create an image (brightness image) that detects only bright areas. The same process is required for general glow. The source code of the shader and script for creating a luminance image is as follows. Note that the shader path is 1.

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;
}
```

There are various methods for calculating the brightness, but the calculation method used in the classical glow implementation was used as it is. I also see shaders that perform processing such as grayscale once and then comparing the brightness.

`BRIGHTNESS_THRESHOLD`Is the threshold for determining brightness, and `INTENSITY`is the bias to multiply brightness. `color`Make sure that the larger the value given to, that is, the brighter the color, the easier it is to return a large value. The higher the threshold, the less likely it is that a value greater than 0 will be returned. Also, the larger the bias, the stronger the brightness image can be obtained.

`ATTENUATION`Is not used at this point. Since the overhead required for exchanging values between CPU and GPU is smaller if they are passed as parameters at once, `Vector3`they are passed together here .

The most important thing at this point is that we are getting the luminance image as a small RenderTexture.

In general, the higher the resolution of a post effect, the greater the load on the Fragment shader, which increases the number of calls and calculations. Furthermore, with regard to the glow effect, the processing load becomes even greater due to repeated processing. Star Glow is no exception to this example. Therefore, the load is reduced by reducing the resolution of the effect to the required level.

The iterative process will be described later.

4.2 STEP 2: Apply directional blur to the luminance image

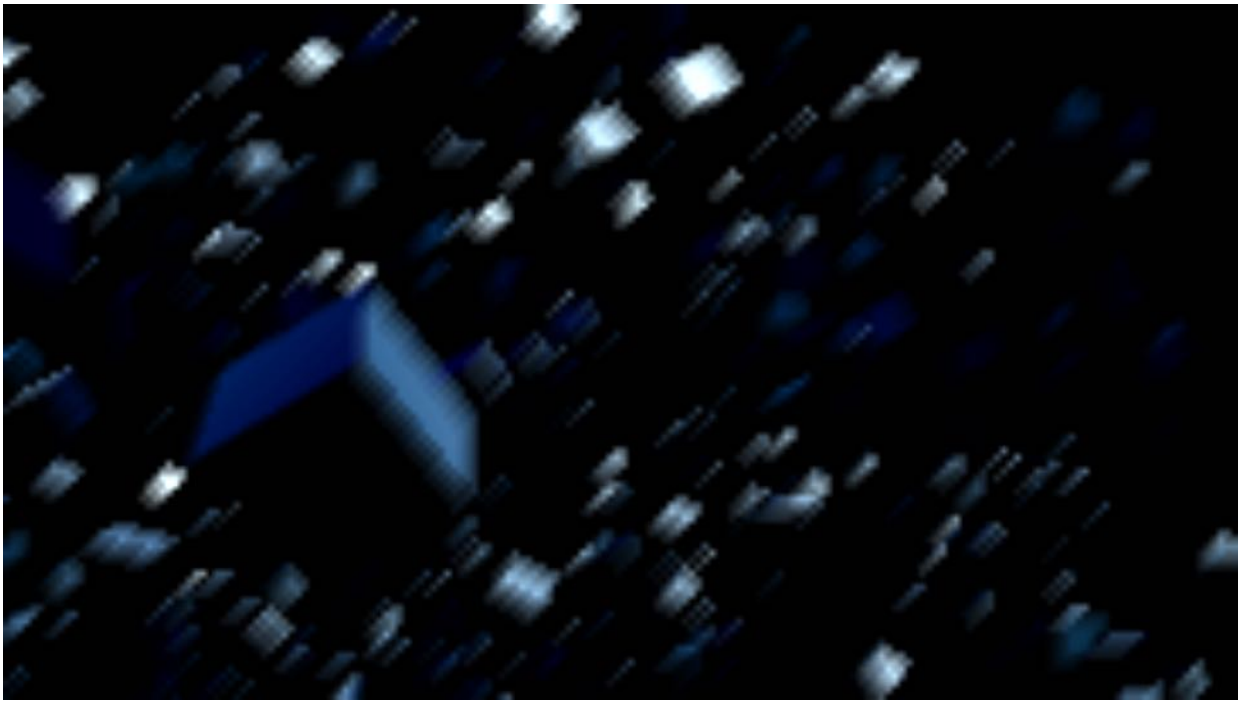

Figure 4.4: Diagonally stretched luminance image

Blur the brightness image obtained in STEP1 and stretch it. By devising this stretching method, it expresses a sharply extending ray that is different from general glow.

In the case of general glow, it is stretched by the Gaussian function in all directions, but in the case of star glow, it is stretched in a directional manner.

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++)
{
    Repeated drawing
}
```

It is different from the actual processing, but `offset = (1, 1)`let's just explain it here . Also note that we are passing `offset`and `iteration`to the shader.

Next, on the script side, drawing is repeated with shader path 2, but for the sake of simplicity, let's move on to the shader once. Notice that we are drawing in shader path 2.

StarGlow.shader

```
int    _Iteration;
float2 _Offset;

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

v2f_starglow vert(appdata_img v)
{
    v2f_starglow or;
    …
    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;
}
```

First, check from the Vertex shader. Indicates `power`the force with which the brightness attenuates when stretched, `offset`and indicates the direction in which the brightness is stretched by the blur. It will be referenced in the Fragment shader described later.

These are calculated within the Vertex shader to refer to common values within the Fragment shader. It is not good to calculate sequentially in the Fragment shader because it increases the number of operations.

Here it `_Iteration = 1`is. Therefore `power = 4^0 = 1`. Then `offset = 画素の大きさ * (1, 1)`you will get.

Now you are ready to sample pixels that are offset by exactly one pixel.

Next is the Fragment shader. To see `uv`wo `offset`one by only one minute to move the reference 4 times while, we are adding up the value of the pixel. However, the pixel value is `pow(ATTENUATION, input.power * j)`multiplied by.

`ATTENUATION`Is a value that indicates how much the value of that pixel is attenuated. It affects the degree of blurring and attenuation when stretched.

If `ATTENUATION = 0.7`so, the first pixel to be sampled would be * 0.7, and the second pixel to be sampled would be 0.7 ^ 2 = * 0.49. It is easy to get an image when you look at the figure.

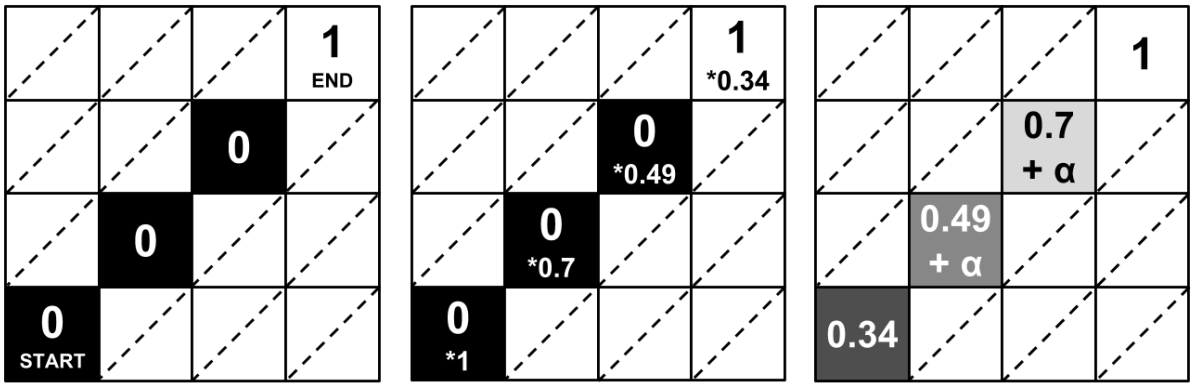

Figure 4.5: A diagram showing the process of blurring

The figure on the left is the original brightness image before attenuation. `_MainTex`Corresponds to. `uv`Let's say the pixel now given to the Fragment shader but referenced is START at the bottom left. `offset = (1, 1)`Therefore, the pixels referenced in the four iterations are up to END in the upper right.

The value in a pixel is the brightness value of that pixel. Three from START are 0s and only END is 1. The attenuation factor of the above source code increases with each iteration, so the image looks exactly like the one in the middle. When this is added up, the final value obtained for the START pixel is `color = 0.34`.

If the Fragment shader processes each pixel in the same way, you will see the result shown on the right. You get a gradation like a blur. Also `offset`it is described in a certain and earlier in the parameter that indicates the direction to stretch. However, as the effect on the appearance, it will extend in the opposite direction to the specified value.

4.2.1 Repeat and stretch further

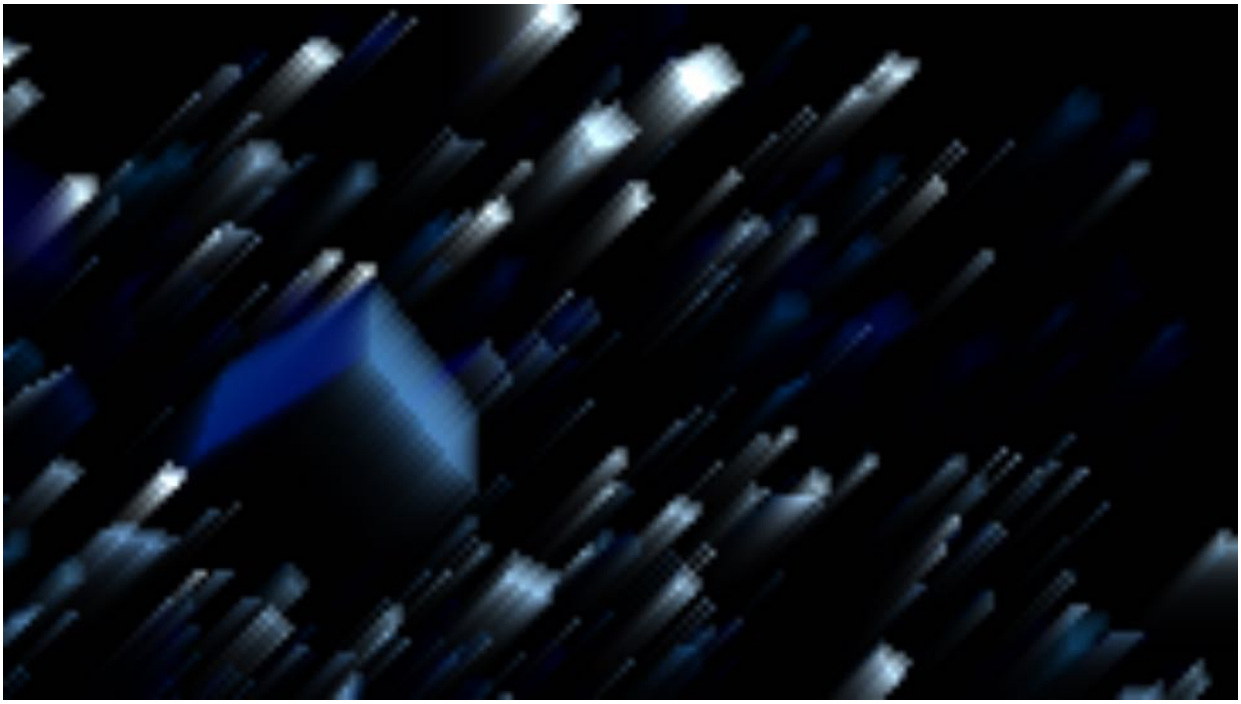

Figure 4.6: Further stretched luminance image

Let's get back to the script a little bit. There is `this.iteration`no explanation so far, `_Iteration`but it was said to be 1. Actually, the same process is repeated any number of times while replacing the RenderTexture.

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);

== The explanation from here to above corresponds to the
explanation ==

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;
}
```

Since the same process is repeated using the same path, the effect obtained does not change. However, `_Iteration`the value of the shader parameter will be higher, which will increase the attenuation in the shader described earlier. Also, the input image will be a blurred image that has already been stretched.

Simply put, this iteration `blurredTex1`results in a blurry image that is even more stretched than the first .

This process is costly, so in reality I think it will be repeated at most 3 times. Also, there are four iterations in the shader, but this value was suggested in Kawase's announcement.

4.3 STEP 2.5: Combine blur images that extend in multiple directions

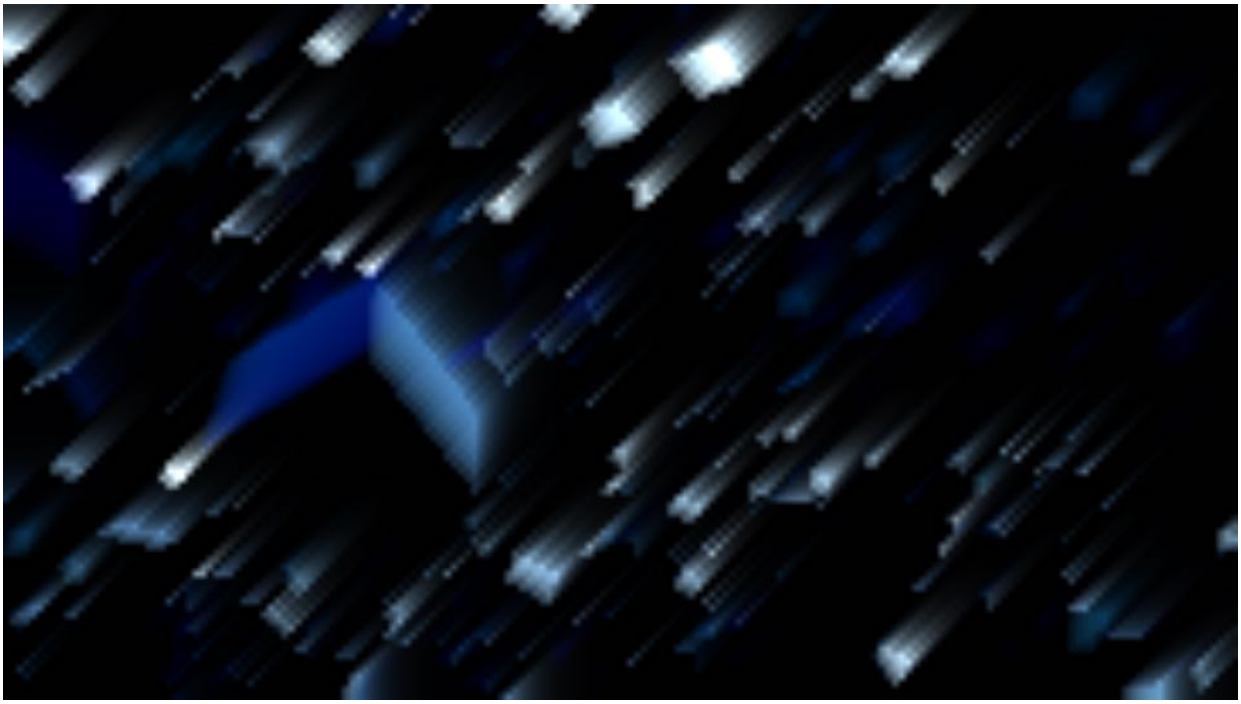

Figure 4.7: Luminance image stretched in another direction

I didn't comment on STEP2.5 in the source code, but I chose 2.5 for the sake of explanation. As mentioned earlier, let's rotate and blur again `offset = (1, 1)`to create a ray that extends in multiple directions `offset`.

Suppose `offset = (1, 1)`we define a ray that extends in the opposite direction `offset = (-1, -1)`. In the actual source code, only the number of rays `offset`is rotated, but for the sake of explanation `offset = (-1, -1)`.

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 is stretched by iterative processing
    }
```

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

The finally obtained blur image `blurredTex1`is `compositeTex`output to the image for compositing . `compositeTex`Is a composite image of all blur images that extend in multiple directions.

At this time, the shader path used to combine the blur images is 3.

StarGlow.shader

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

No special processing is done in this path, but the `Blend`syntax is used to synthesize the images. I think that the composition method may be remade depending on the production, but I decided here `OneMinusDstColor One`. This is a soft composition method.

4.4 STEP 3: Combine the blur image with the original image

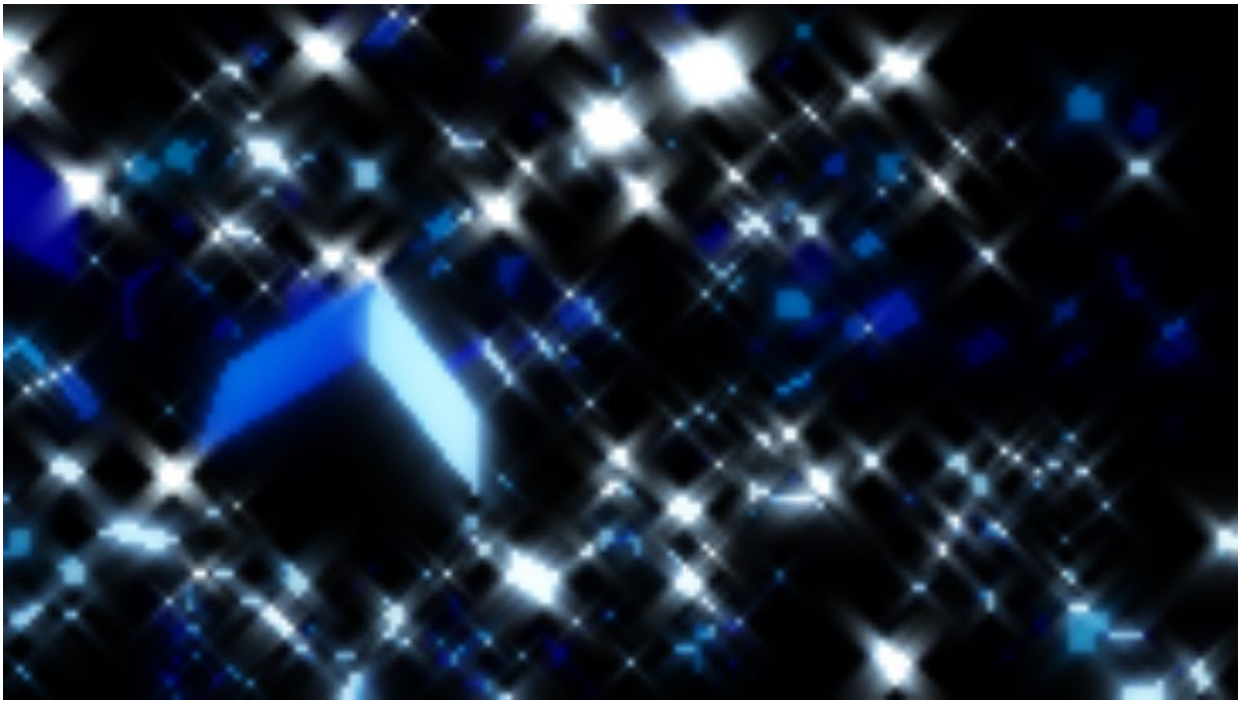

Figure 4.8: Final Blur image

After obtaining a blur image that extends in multiple directions, the blur image is combined with the original image and output in the same way as a general glow. You `Blend`can use the same syntax as in STEP 2.5 above to synthesize and output, but here, `Blit`to reduce the number of times and to make the synthesis method flexible, `Pass 4`we use the one for synthesis. ..

StarGlow.cs

```
base.material.EnableKeyword(StarGlow.CompositeTypes[this.composi
teType]);
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`Although the syntax is not used, the screen composition and additive composition are reproduced as they are. Furthermore, here, by adding a color that is arbitrarily multiplied, it is possible to express a star glow with a strong color.

4.5 STEP 4: Release resources

Release all the resources you have used. There is no special explanation, but just in case it is described in the sample on the source code. If the implementation environment is limited, it may be possible to reuse the reserved resources, but here we will simply release it.

StarGlow.cs

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

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

4.6 Summary

I explained the basic (as announced by Mr. Kawase) implementation method of Star Glow, but if you are not particular about real-time performance, you can express various rays by switching the calculation method and parameters of the brightness image multiple times. And so on.

Even within the range described here, if you change the parameters at the timing of the iteration, for example, you will be able to create heterogeneous, more "like" and "tasteful" rays. Or you can use noise to change the parameters over time.

It is not a physically correct ray, and if you need a more dramatic and advanced expression of rays, it will be realized by a method other than the post effect, but this effect that can be made gorgeous with a relatively simple structure is also available. It's very interesting so please give it a try.

… It's a little heavy.

See 4.7

- 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　Introduction

In this chapter, we will explain one of the methods of dividing polygons into triangles, the "ear clipping method", hereinafter referred to as the "ear clipping method". In addition to the usual simple polygon triangulation, we will also explain the triangular division of polygons with holes and polygons that have a hierarchical structure.

The sample in this chapter is "TriangulationByEarClipping" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming4
.

5.1.1　How to operate the sample

Run the sample DrawTest scene. Left-click on GameView to make a dot on the screen. Continue left-clicking on another point to connect it with the first point with a line. If you repeat it, you will get a polygon. When drawing lines, be careful not to cross the lines. Right-click to split the polygon into triangles to generate a mesh. If you generate a polygon in the generated mesh, you will get a polygon with holes.

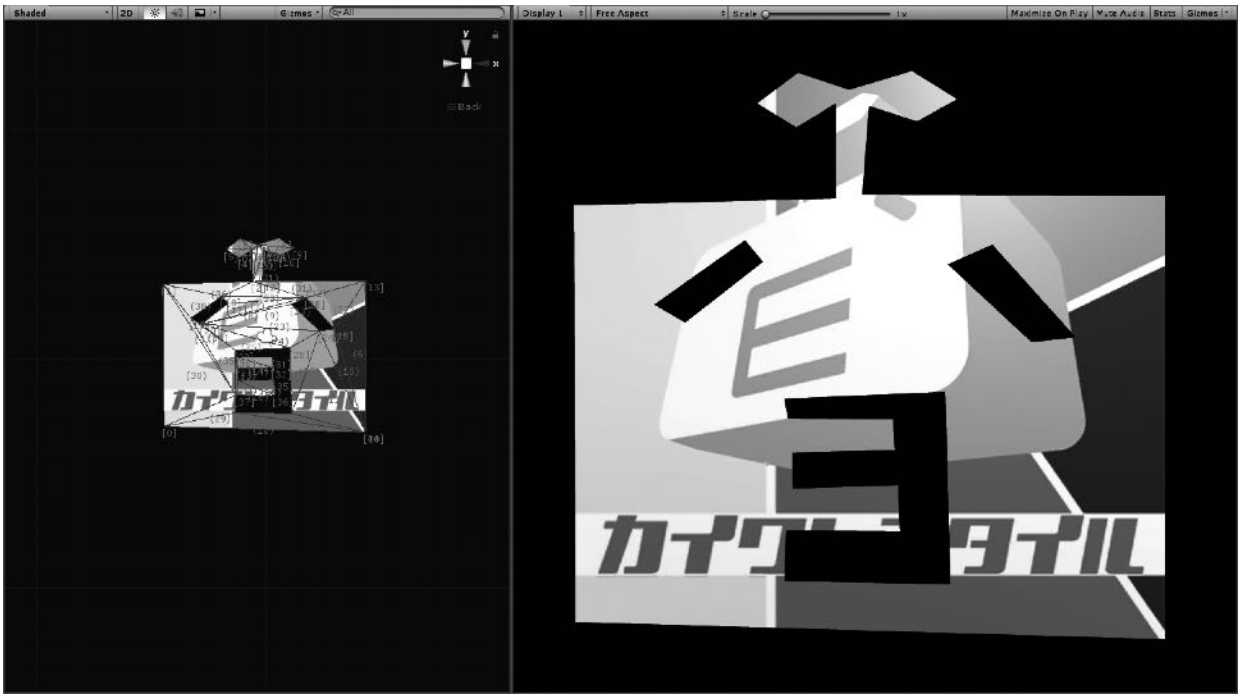

Figure 5.1: Screen of running the sample

5.2 Simple polygon triangulation

A simple polygon is a closed polygon that does not intersect at its own line segment.

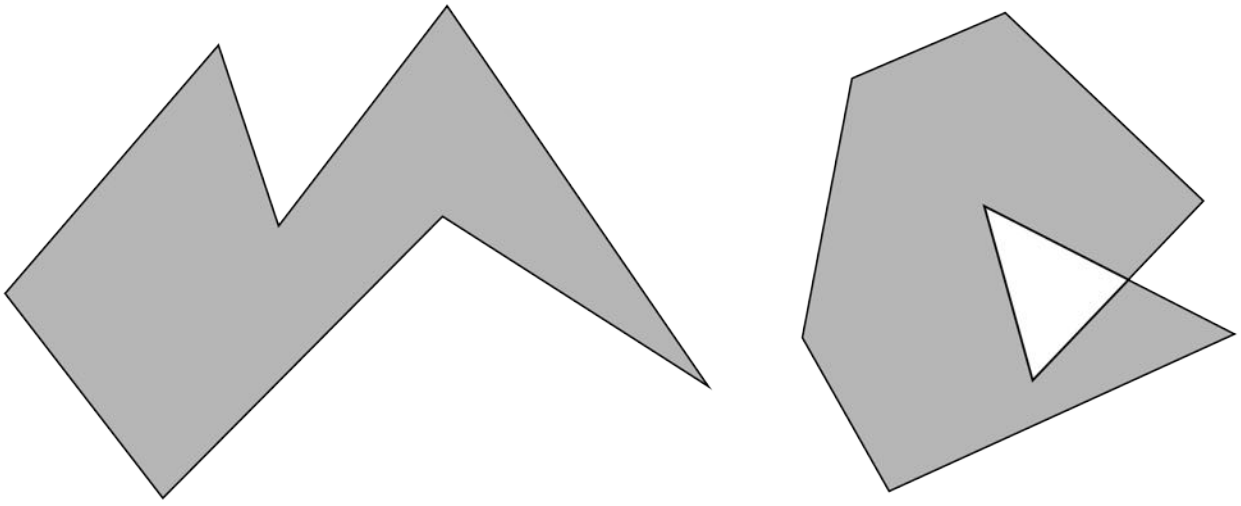

Figure 5.2: Left: Simple polygon, Right: Non-simple polygon

Any simple polygon can be triangulated. Dividing a simple polygon with n vertices into triangles creates n-2 triangles.

5.3 Ear cutting method (EarClipping method)

There are many methods for dividing polygons into triangles, but this time we will explain the "ear cutting method", which is simple to implement. The "ear-cutting method" is divided based on the theorem "Two ears the orem". This "Ear" refers to "a triangle whose two sides are polygonal sides and the remaining one side exists inside the polygon", and this theorem states that "four or more sides". A simple polygon without a hole with has at least two ears. "

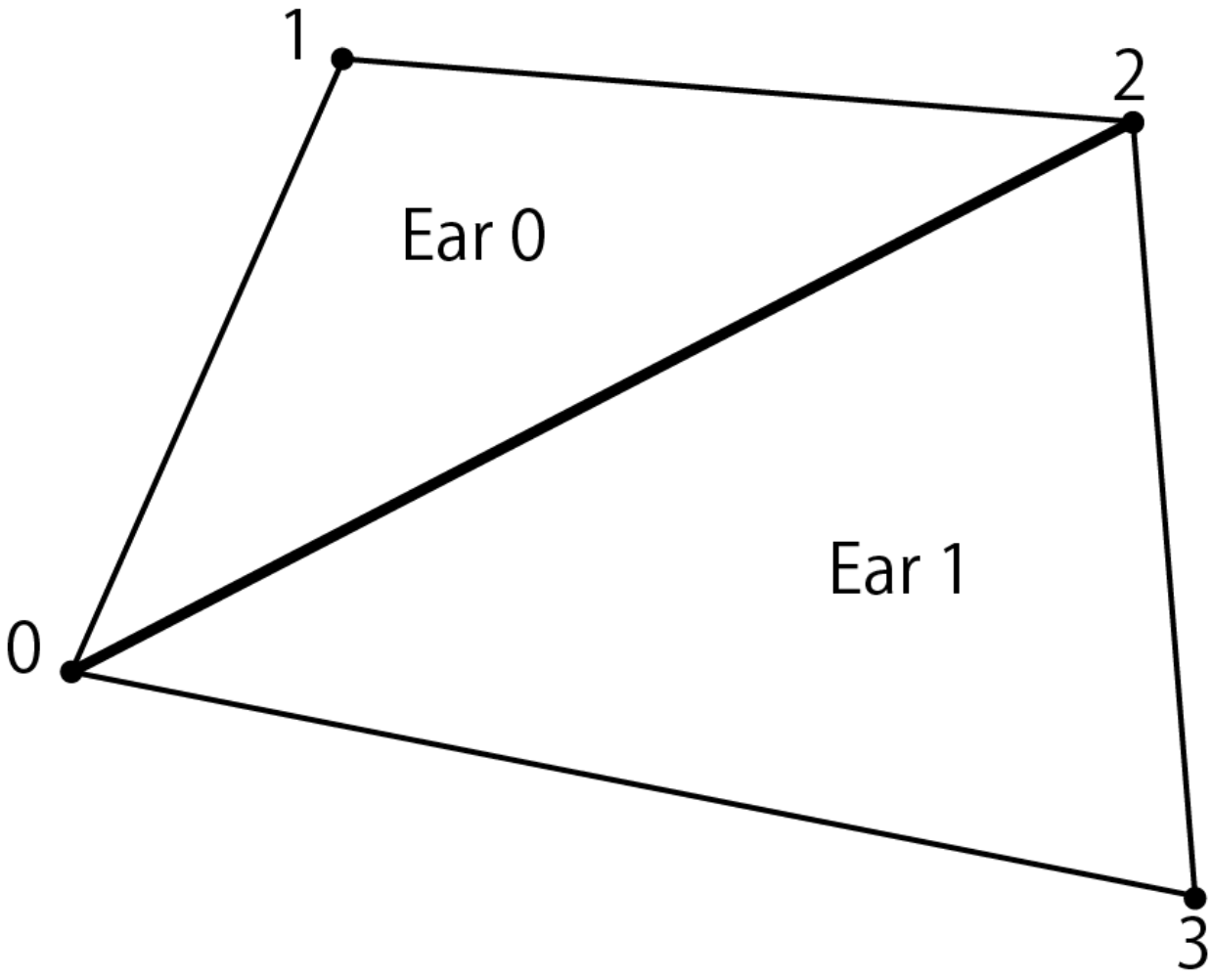

Figure 5.3: Ears

The "ear trimming method" is an algorithm that searches for this "ear" triangle and removes it from the polygon. This "ear cutting method" is simpler than other division algorithms, but it is slow, so I don't think it can be used very much in situations where speed is required.

5.3.1 Flow of triangle division

First, look for "ears" in the given array of polygon vertices. The conditions for "ears" are the following two points.

- The angle (internal angle) of the line segment with the vertices (vi-1, vi + 1) before and after the vertex vi of the polygon is within 180 degrees (called a convex vertex).
- No other vertices are included in the triangle consisting of polygon vertices vi-1, vi, vi + 1.

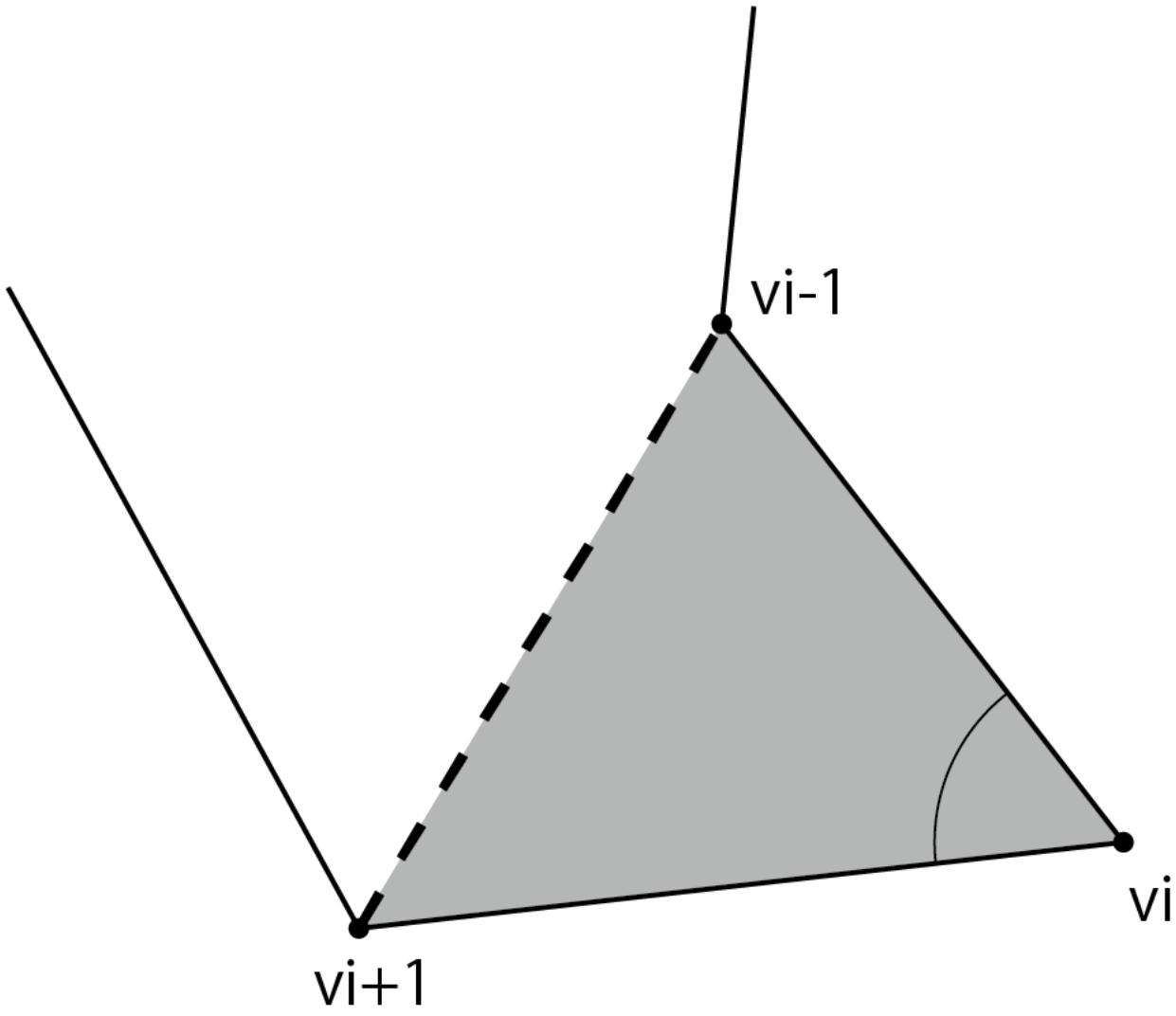

Figure 5.4: Ear conditions (within 180 degrees, no other vertices in the triangle)

Add the vertex vi that meets the above conditions to the ear list. This is done by the InitializeVertices function in the sample Triangulation.cs. Then create the triangles that make up the ear from the top of the ear list and remove the vertex vi from the vertex array.
Removing the vertex vi changes the shape of the polygon. For the remaining vertices vi-1, vi + 1, perform the above ear judgment again. If vertices vi-1, vi + 1 meet the ear criteria, they will be added to the end of the ear list, but they may also be removed from the ear list. This process corresponds to the CheckVertex function and EarClipping function of the sample Triangulation.cs.

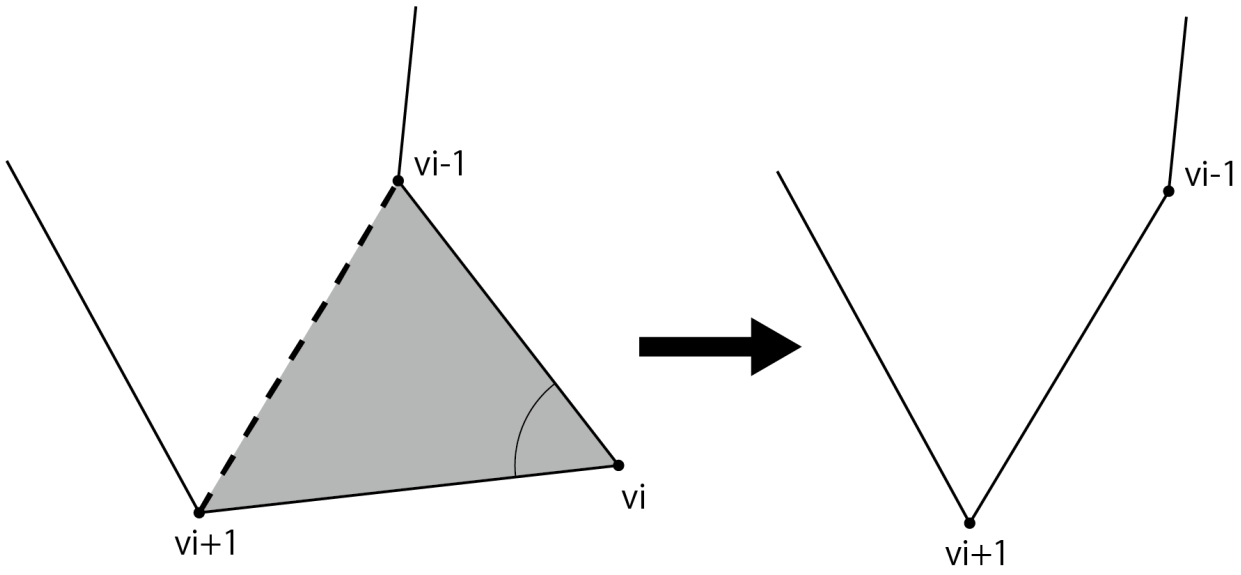

Figure 5.5: Polygon before and after deleting vertex vi

Let's illustrate a series of flows using a simple polygon as an example.

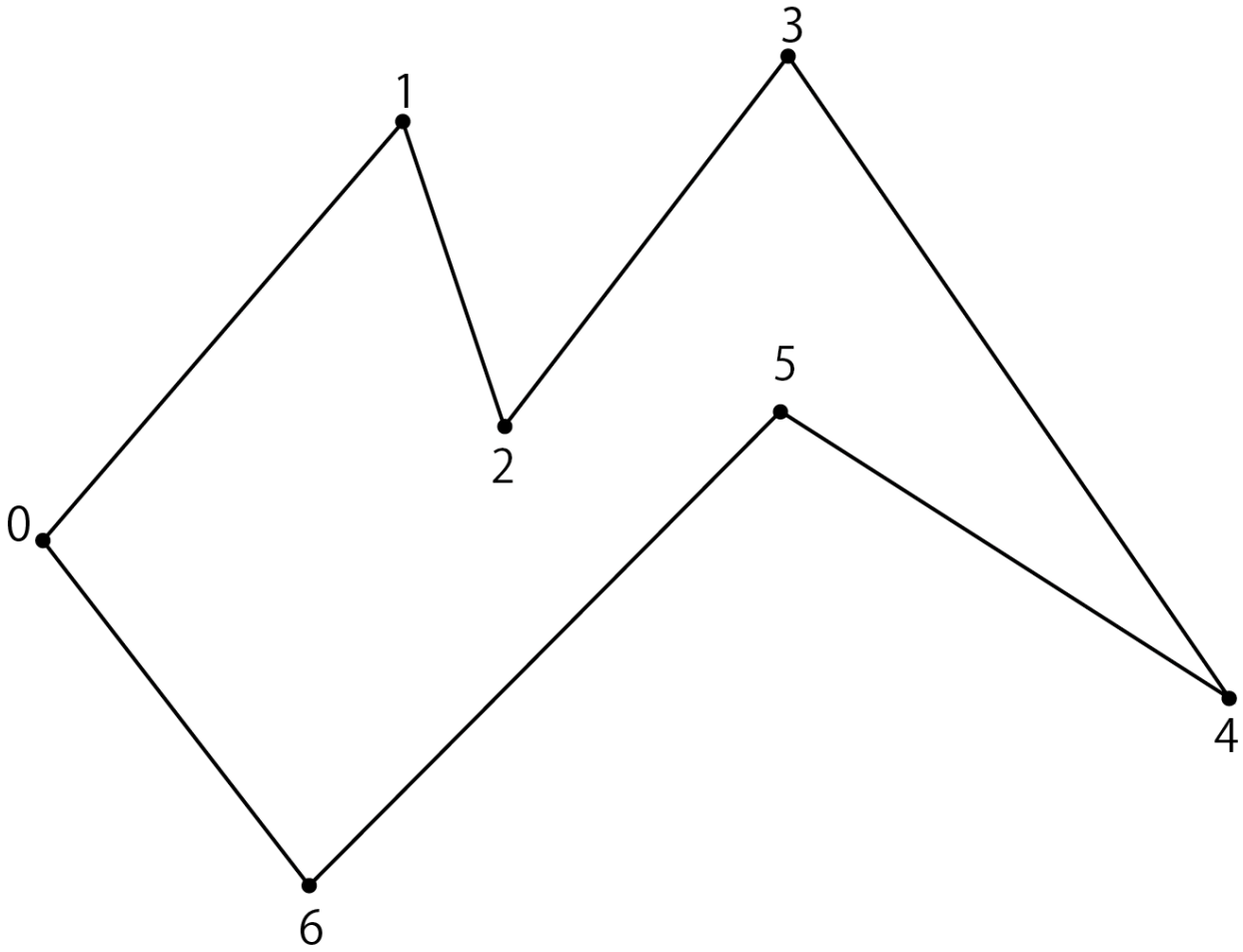

Figure 5.6: A simple polygon

First, look for your ears. In this case, the ear list contains vertices 0,1,4,6. Vertices 2 and 5 are excluded because they are not convex vertices, and vertices 3 are excluded because they are contained in triangles 2, 3 and 4.
First, take out the first vertex 0 of the ear list. Make a triangle with vertices 1 and 6 before and after vertex 0. Remove vertex 0 from the vertex array and connect the previous and next vertices 1 and 6 to form a new polygon. Then, the ears are judged for vertices 1 and 6. Originally both were ears, but they remain ears even after the ear judgment. The ear list at this time is 1,4,6.

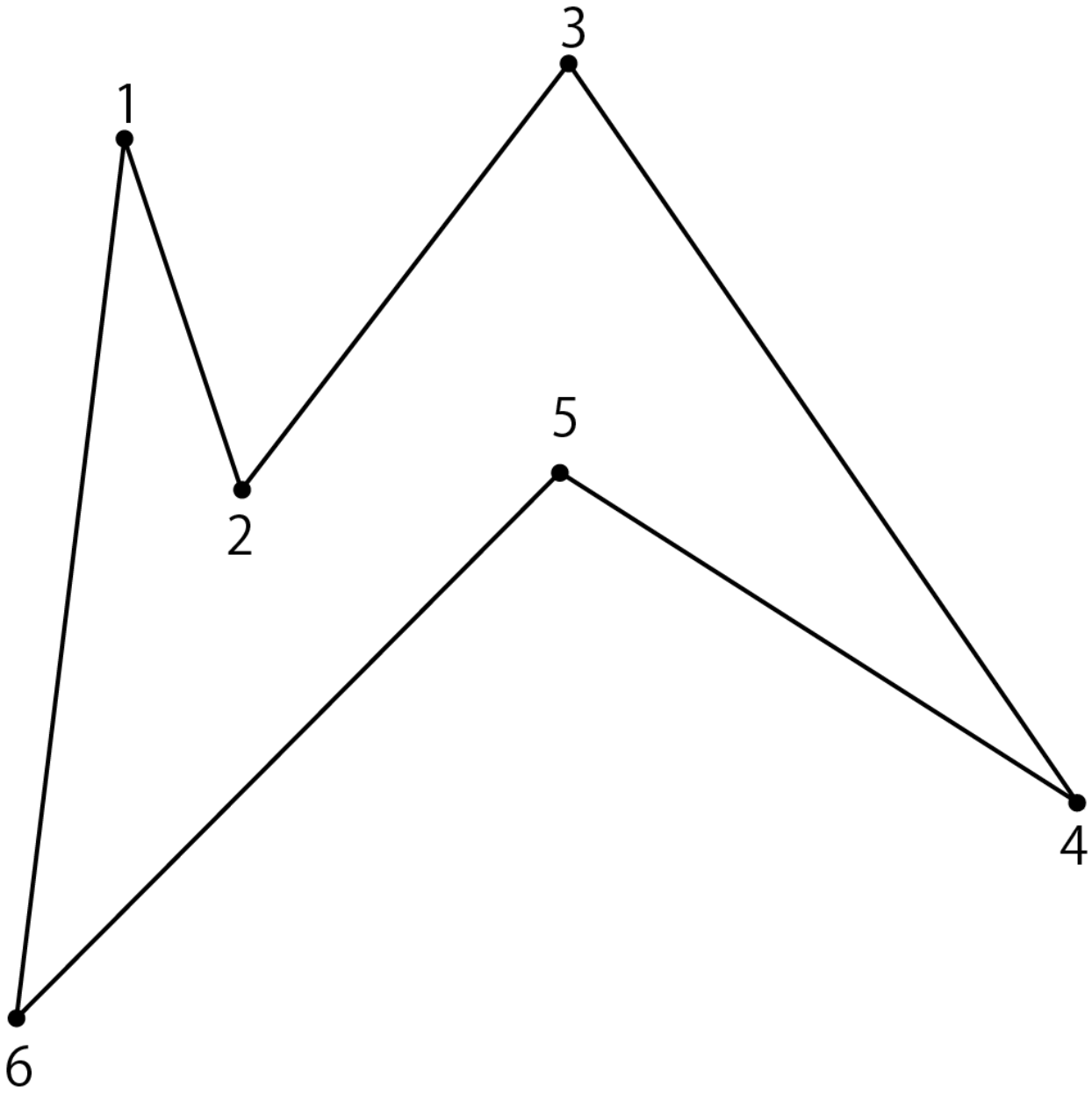

Figure 5.7: Polygon with 0 vertices removed

Then take vertex 1 from the beginning of the ear list. Make a triangle with vertices 2 and 6 before and after vertex 1. Remove vertex 1 from the vertex array and connect the previous and next vertices 2 and 6 to form a new polygon. Then, the ears are judged for vertices 2 and 6. Since the vertex 1 is gone, the vertex 2 becomes a convex vertex and the ear condition is satisfied, so add it to the ear list. Vertex 6 remains in the ear. The ear list at this time is 4,6,2.

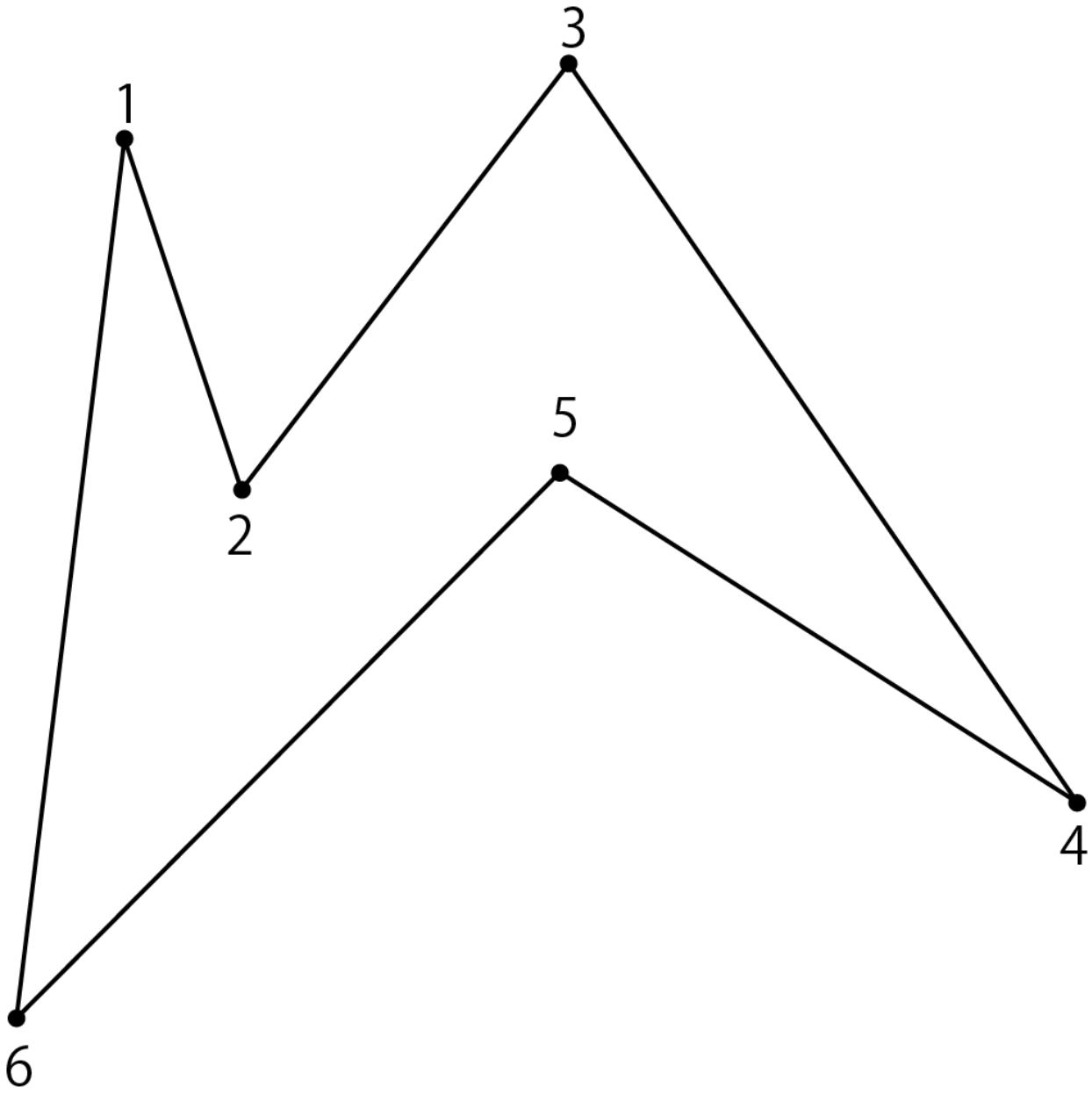

Figure 5.8: Polygon with vertex 1 removed

Then take vertex 4 from the top of the ear list. Make a triangle with vertices 3 and 5 before and after vertex 4. Remove vertex 4 from the vertex array and connect the previous and next vertices 3 and 5 to form a new polygon. Then, the ears are judged for vertices 3 and 5. With the disappearance of vertex 4, the triangle created by vertices 2 and 5 before and after vertex 3 no longer contains other vertices, so add vertex 3 to the ear list. Also, since the internal angle of vertex 5 is 180 degrees or less, it becomes a convex vertex and the

ear condition is satisfied, so add it to the ear list. The ear list at this time is 6,2,3,5.

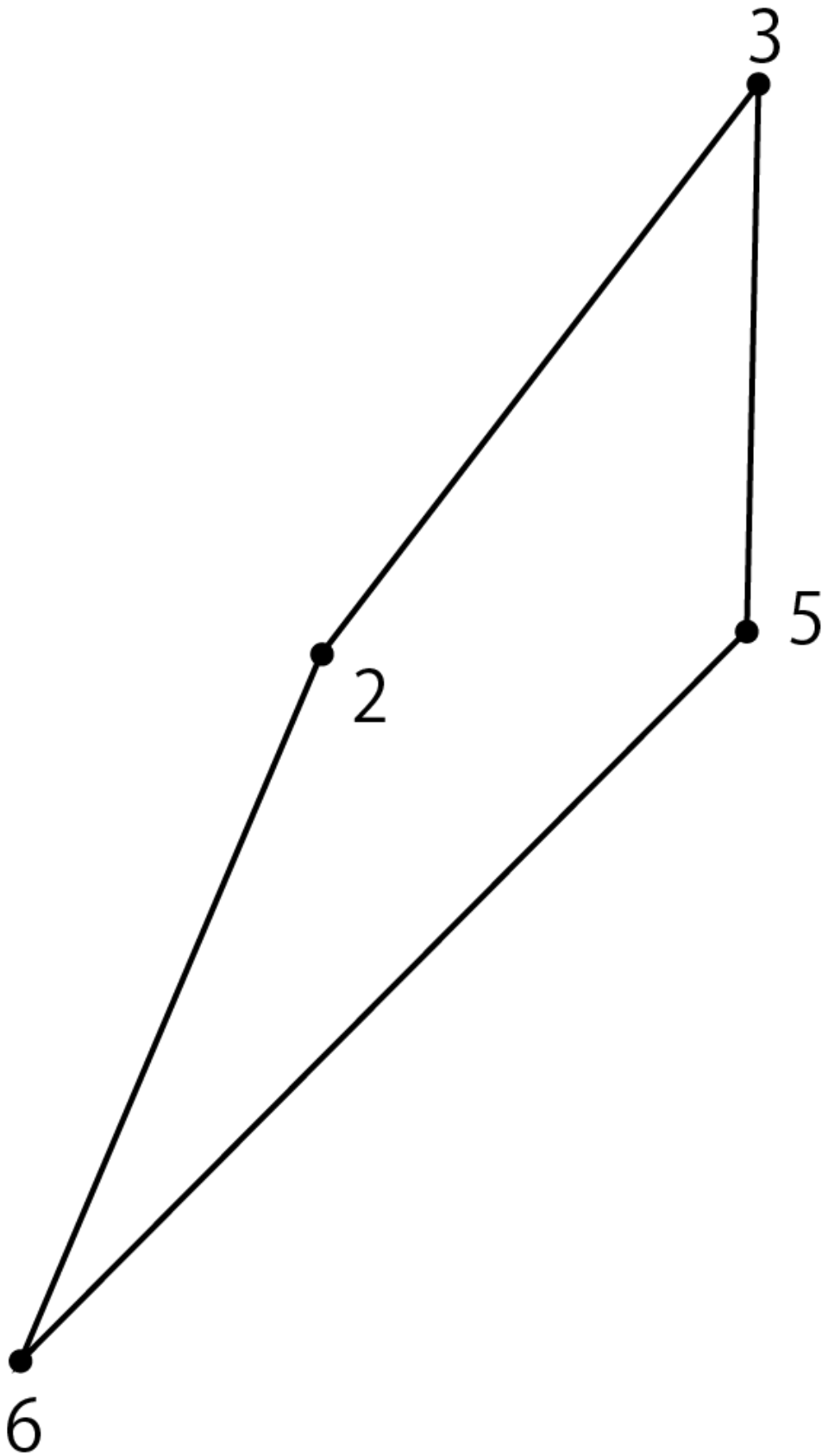
3
5
2
6

Figure 5.9: Polygon with vertex 4 removed

Then take vertex 6 from the top of the ear list. Make a triangle with vertices 2 and 5 before and after vertex 6. Remove vertex 6 from the vertex array and connect the previous and next vertices 2 and 5 to form a new polygon. Then, the ears are judged for vertices 2 and 5. Originally both were ears, but they remain ears even after the ear judgment. The ear list at this time is 2,3,5.

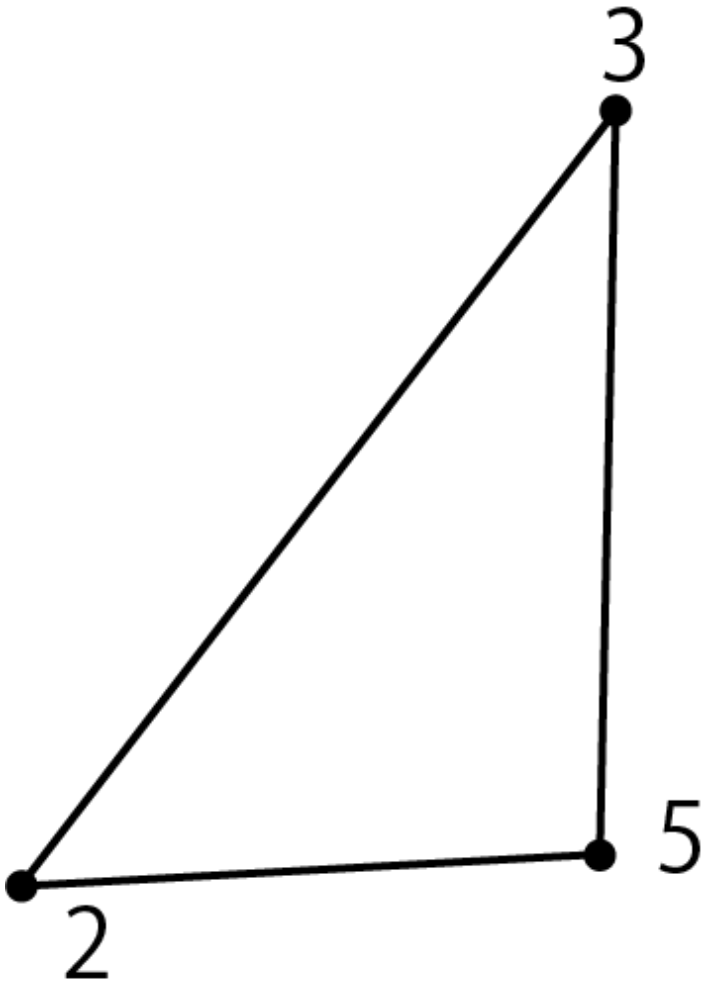

Figure 5.10: Polygon with vertex 6 removed

Next, I took out vertex 2 from the top of the ear list ... I thought, but since there are only 3 vertices of the polygon left, I made a triangle as it is and the triangle division is finished. The final result of the triangle split is:

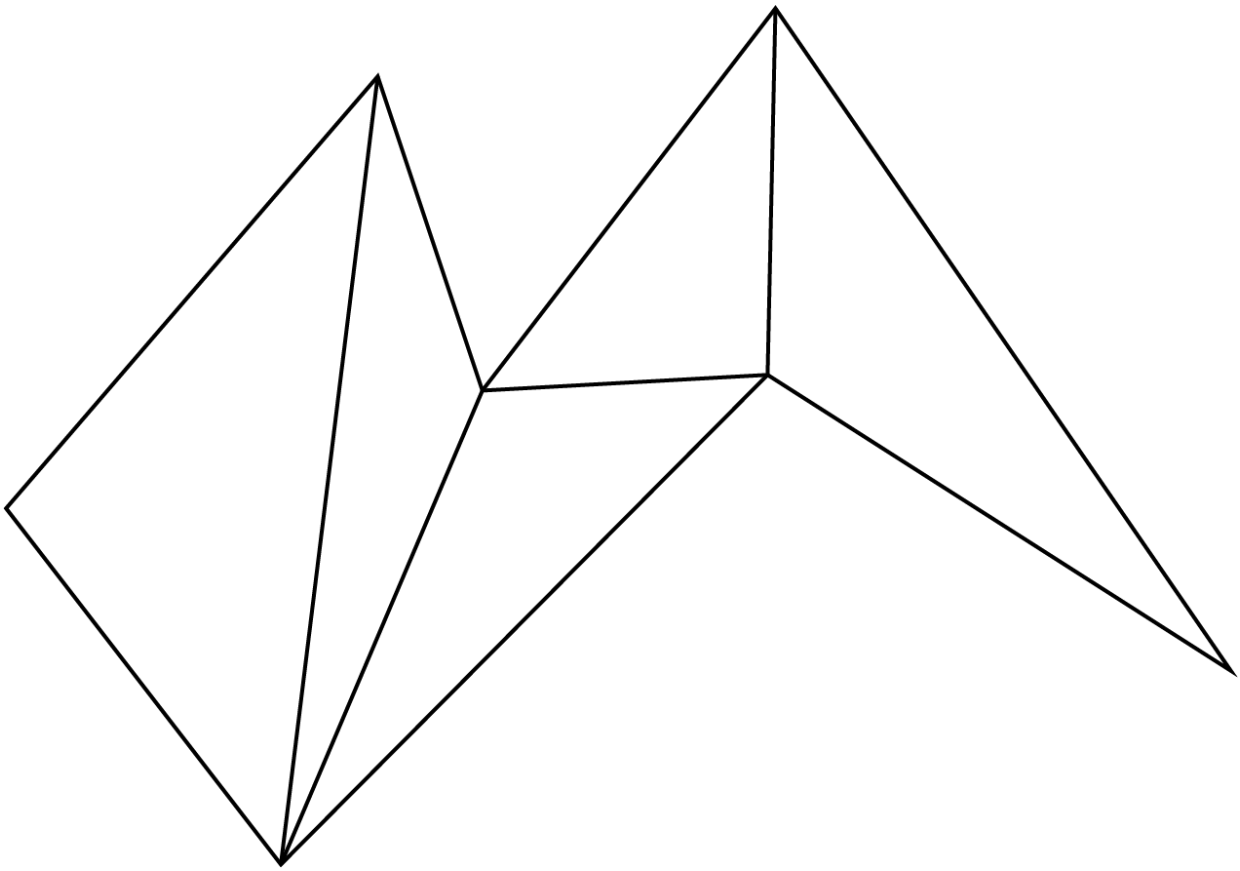

Figure 5.11: Triangular division result

5.4 Perforated polygonal triangle division

Next, I will explain the triangular division of a polygon with holes. Originally, the "ear cutting method" cannot be applied to polygons with holes, but if you make a notch in the outer polygon and connect it to the inner polygon as shown in the figure, the inner polygon will be the outer polygon. It will be part of the and you will be able to apply the ear-cutting method. This method is also possible for polygons with multiple holes.

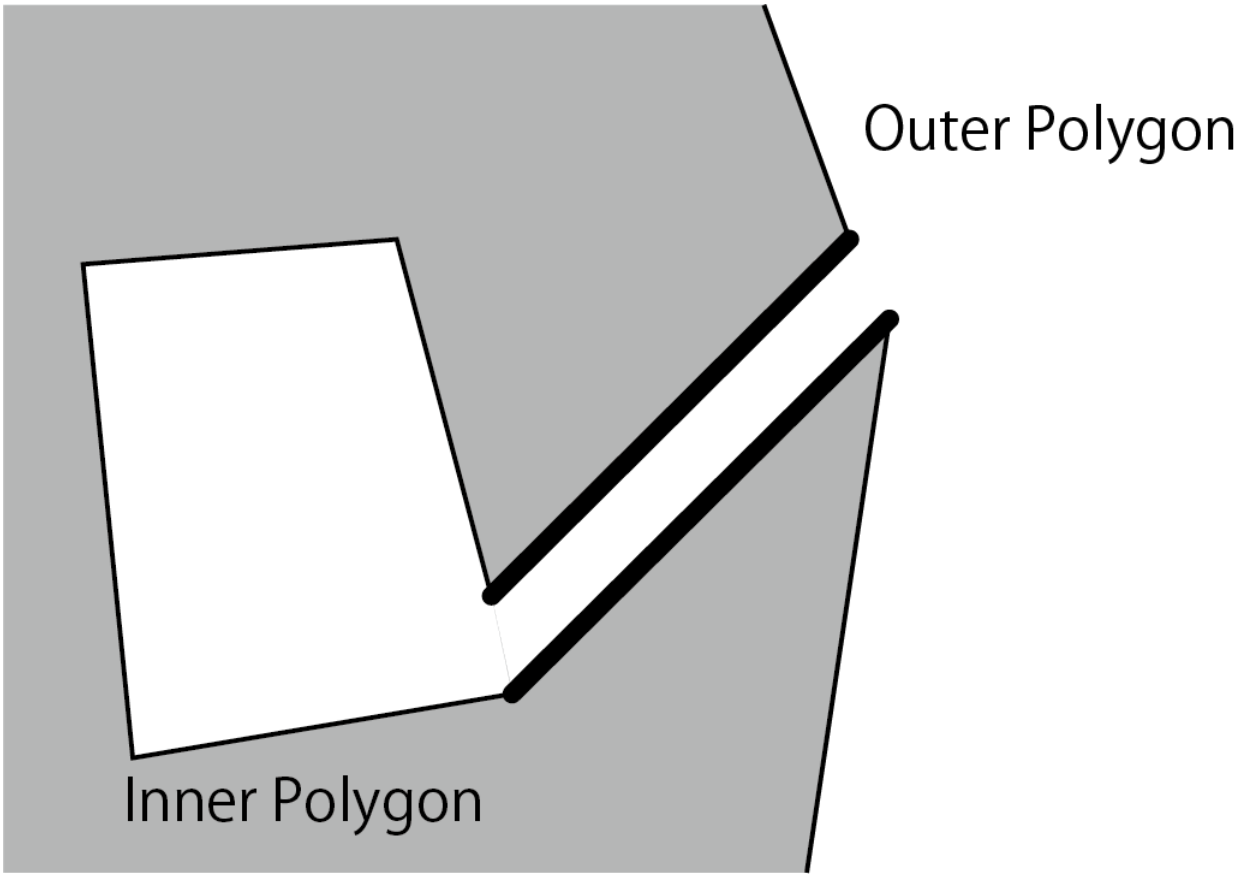

Figure 5.12: Joining inner and outer polygons (figure is a fairly exaggerated representation)

5.4.1 Flow of joining outer polygon and inner polygon

As a premise, the order of the vertices of the outer and inner polygons must be reversed. For example, if the outer polygon has vertices aligned clockwise, the inner polygon must align counterclockwise. The flow of joining is explained using the following polygon as an example.

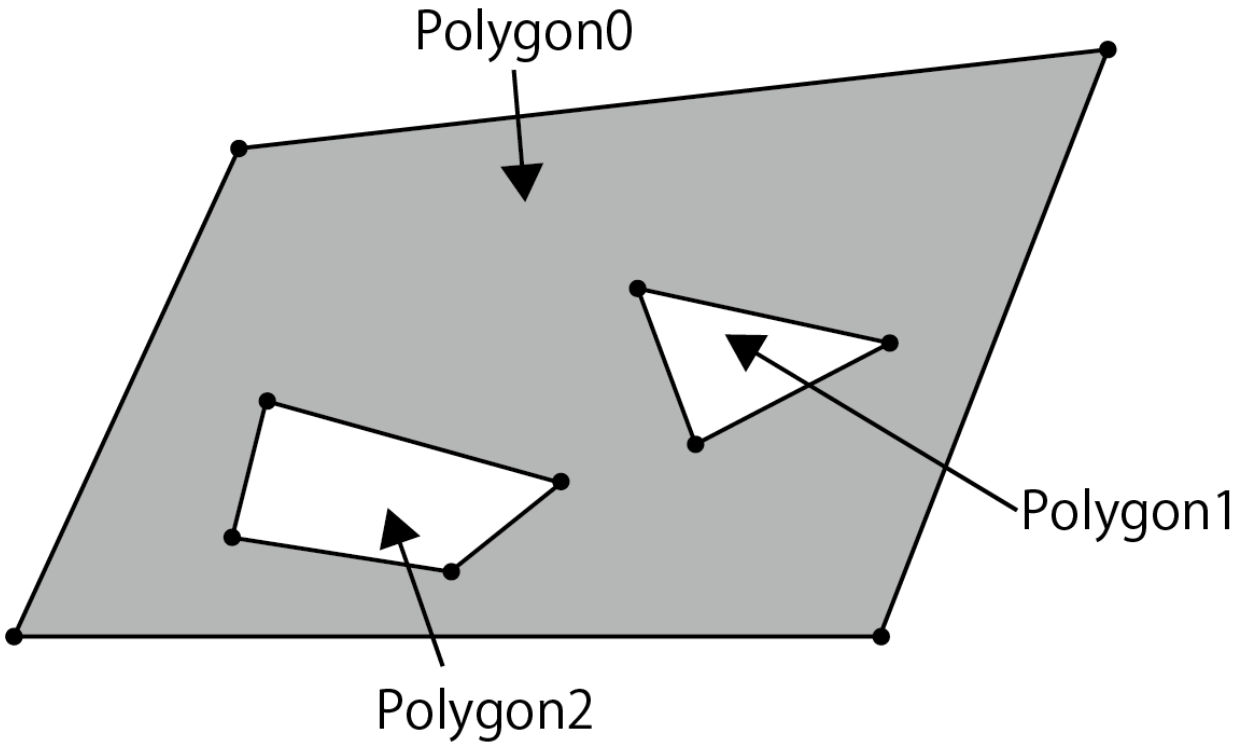

Figure 5.13: Polygon with holes

1. If there are multiple holes (inner polygons), look for the polygon with the largest X coordinate (on the right) and its vertices among the inner polygons.

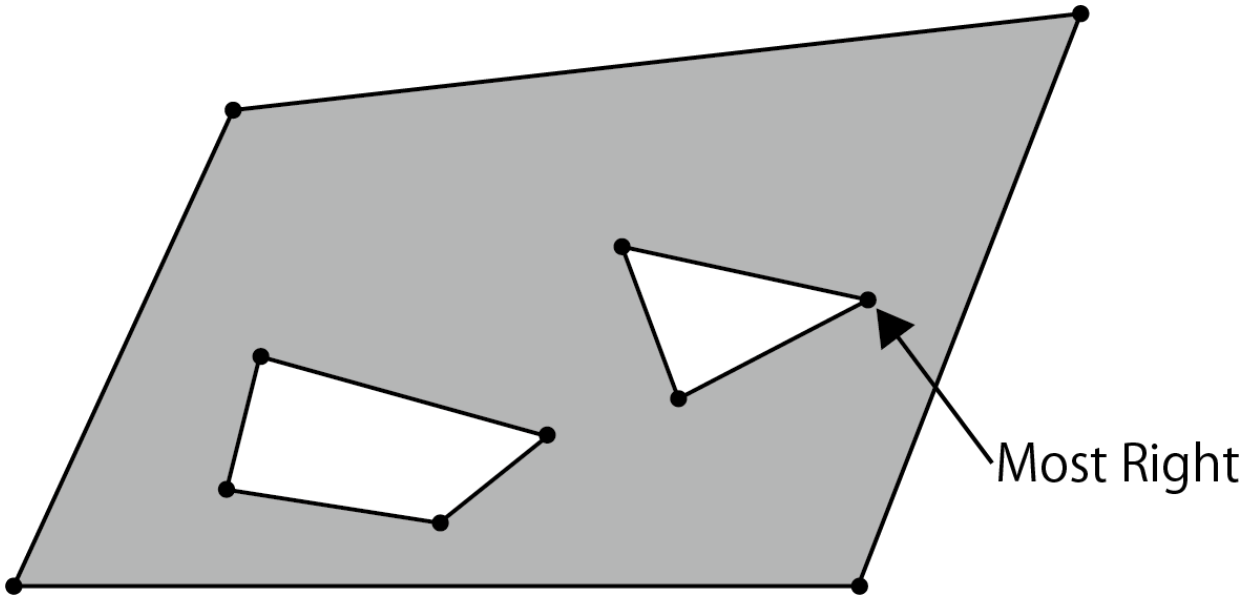

Figure 5.14: Vertex with the highest X coordinate

2. Let M be the vertex with the largest X coordinate. Draw a straight line from M to the right.

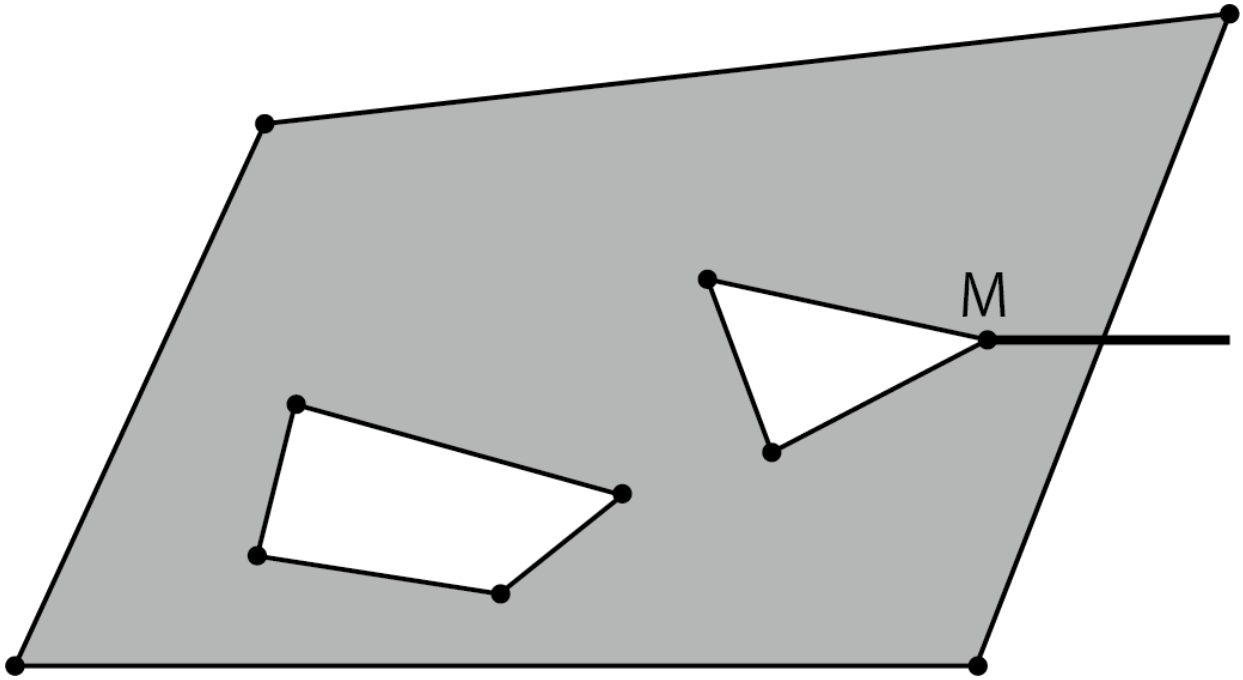

Figure 5.15: Draw a line to the right from vertex M

3. Find the edge and intersection I of the outer polygon that intersects the line extending to the right from vertex M. If it intersects multiple sides, select the side of the intersection closest to vertex M.

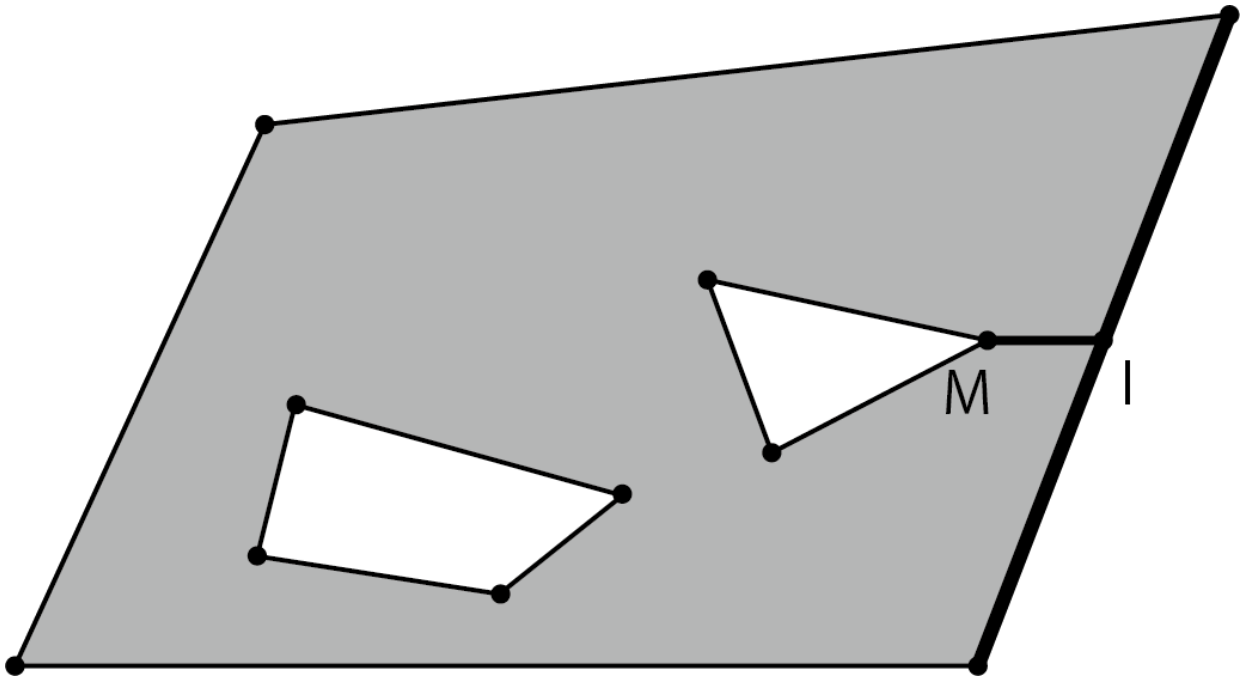

Figure 5.16: Vertex M and intersection I

4. Select the vertex P with the largest X coordinate among the vertices of the intersecting sides. Check if the triangle connecting the vertices M, I, P contains other vertices.

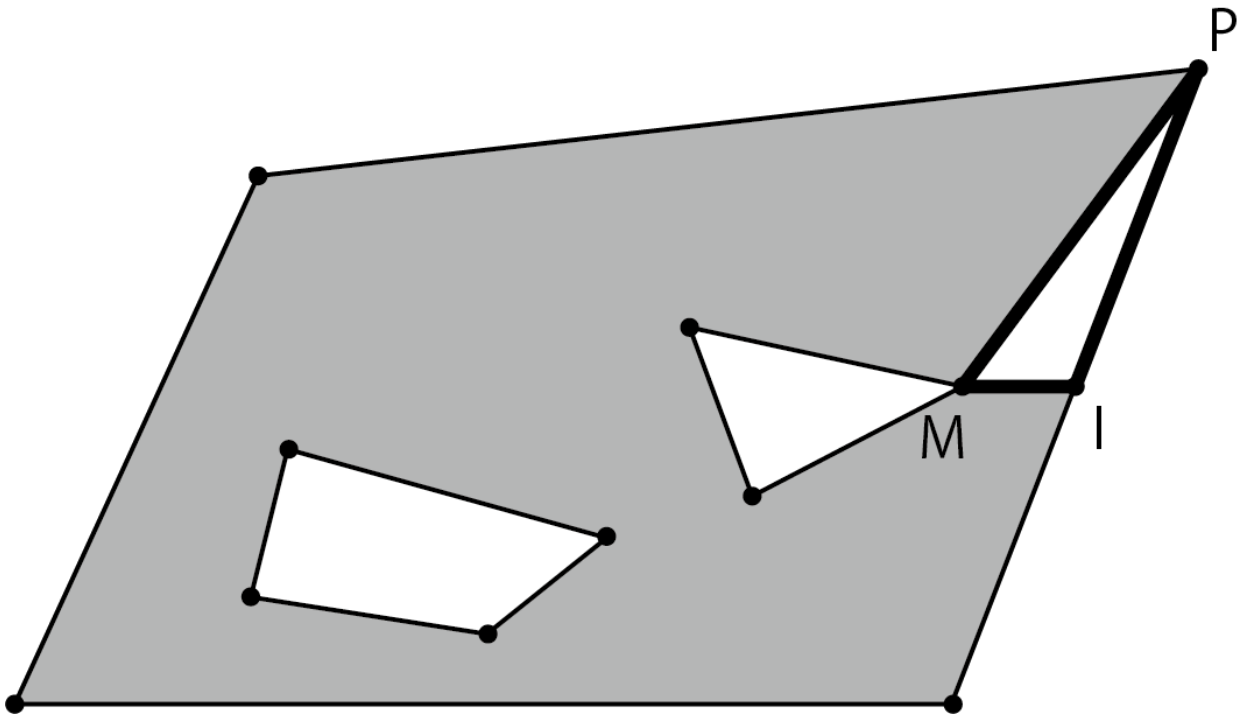

Figure 5.17: Triangle M, I, P

5. If the triangles M, I, P do not contain other vertices, it can be divided, so connect the vertices P of the outer polygon to the vertices M of the inner polygon, and turn the inner polygon counterclockwise. I will go around. When connecting from M to the vertex P of the outer polygon again, the vertex M and the vertex P are duplicated to make another vertex (vertex M', P'). By separating the incoming line and the outgoing line, the lines seem to overlap, but the order of the vertices is a simple polygon that does not intersect.

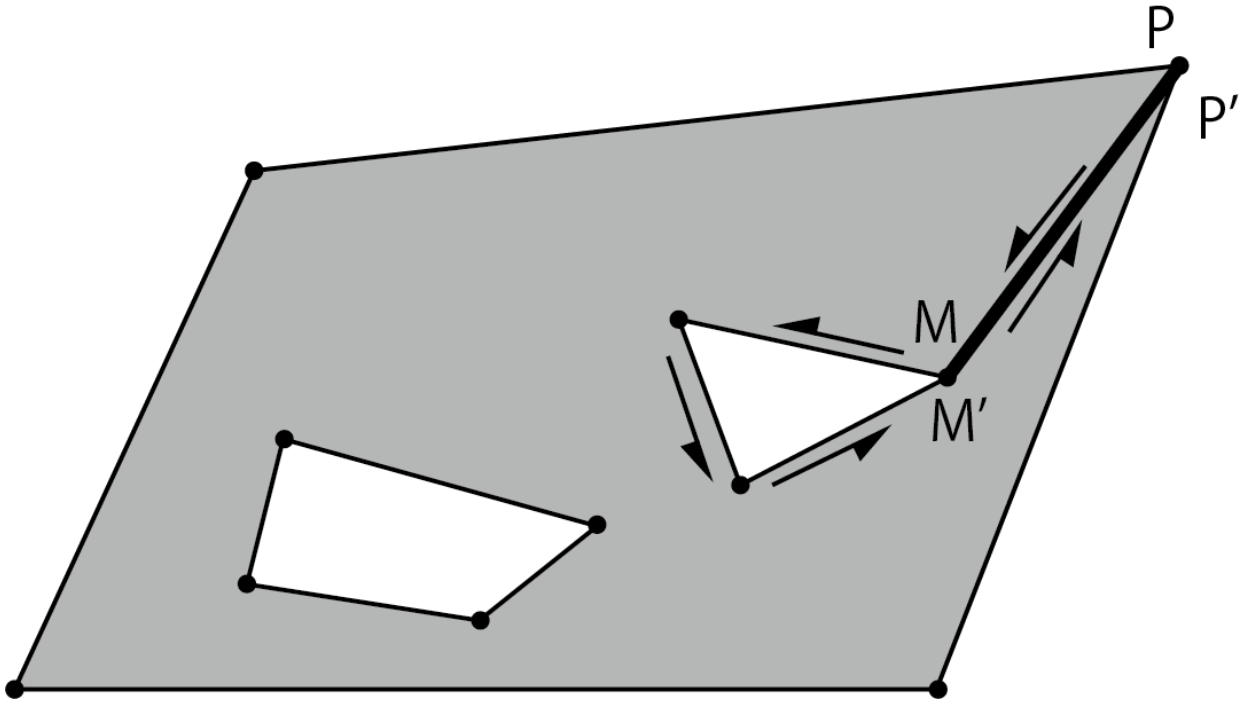

Figure 5.18: A diagram connecting the outer polygon and the inner polygon

6. If the triangles M, I, P contain other vertices R, select that vertex R, but if multiple vertices are included, line segments M, I and line segment M Select the vertex R with the smallest angle θ formed by, R, and perform processing 5.

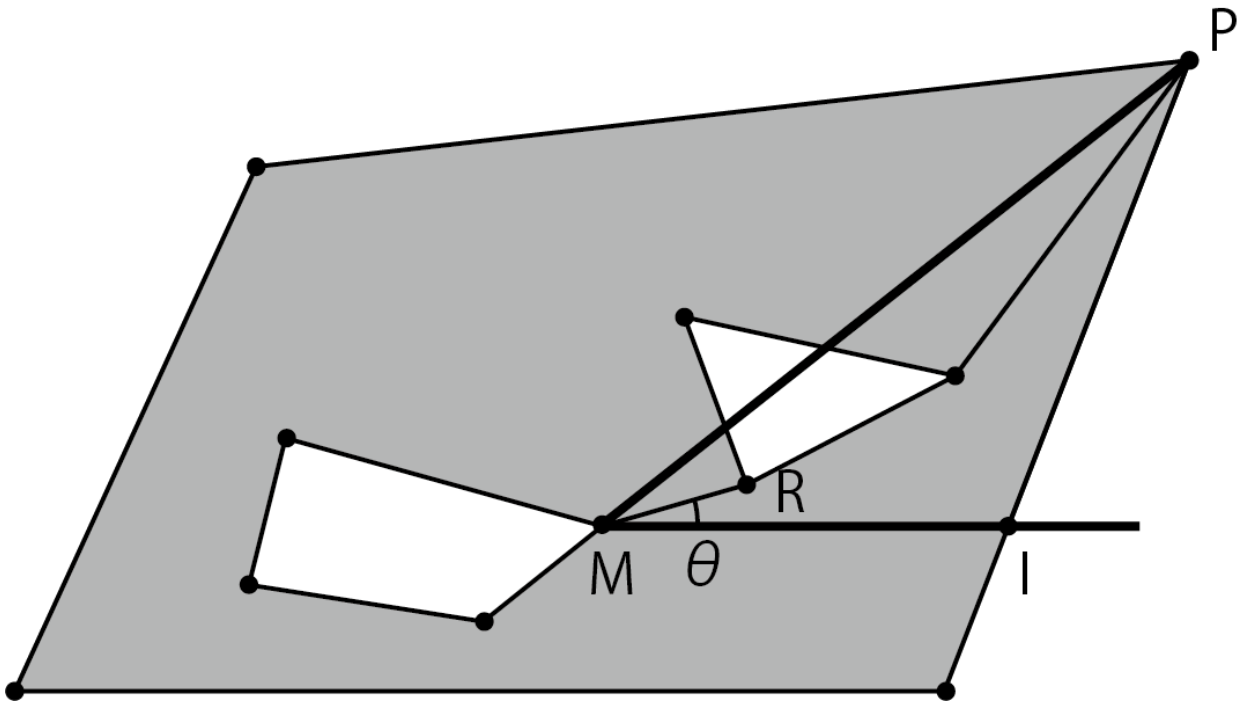

Figure 5.19: Vertex R with the smallest angle θ formed by line segments MI and MR

7. Go back to 1 and join with the other inner polygons.

5.5 Nested polygon triangulation

Next, I will explain the triangular division of a nested polygon. Since the process of joining polygons with holes and the process of dividing triangles were explained in the previous section, here we will mainly explain the procedure for building a tree of parent-child relationships of polygons.

1. Sorts in descending order of area when the polygon is a rectangular area. The area of the rectangular area created by the vertices of the minimum / maximum coordinates of the polygon.
2. Recursively determines whether other polygons include all vertices in a polygon with a large area, and creates a tree of parent-child relationships. At this time, the top-level route is an empty polygon (so-called dummy) and is not used for the subsequent combination processing of polygons. The reason for making the top level a dummy is that if multiple sets of completely uncovered polygons are passed, the

top level will not be one. You can simplify the process by hanging multiple polygons that do not overlap at all below the top of the dummy. Also, when the hierarchy is an even-numbered floor, it becomes an inner polygon, so the vertex array of the corresponding polygon is rearranged counterclockwise.

3. Once you have a parent-child relationship tree, take out one polygon from the top. It becomes the outer polygon.
4. Extracts the polygon one level below (child) of the outer polygon. Then, as an inner polygon, it is combined with the outer polygon to perform triangular division. If there are no children, divide it into triangles as it is.
5. Go back to 3 and repeat the join and split. In 4, the outer polygon and the inner polygon group are processed as one combination, so the next triangle to be extracted will be the outer polygon again.

Let's take the following set of polygons as an example.

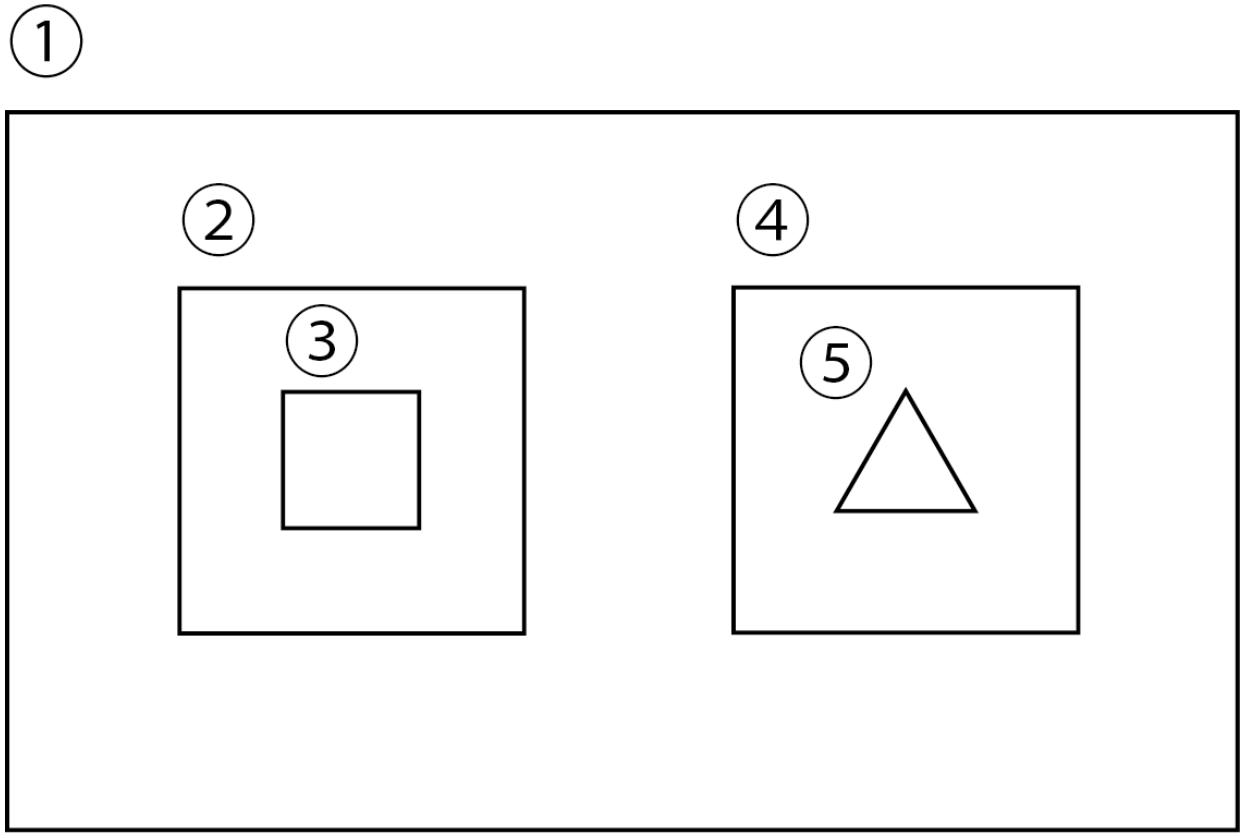

Figure 5.20: Nested polygons

When you create a polygonal parent-child relationship, you get the following tree.

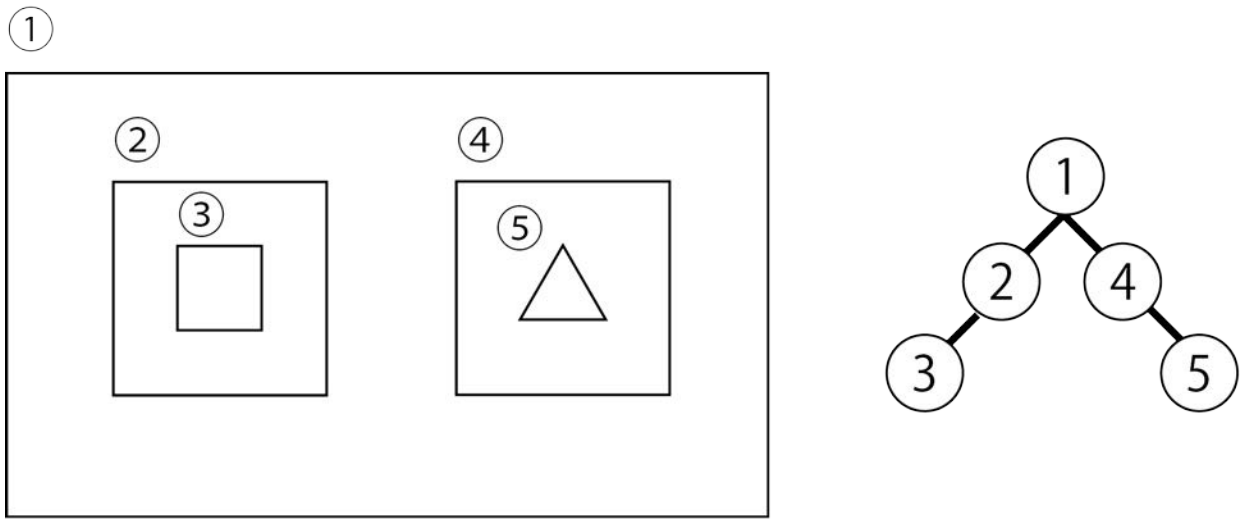

Figure 5.21: Left: Nested polygon Right: Parent-child relationship

Extract polygon 1 at the top of the tree (excluding dummies) and polygons 2 and 4 of its children.

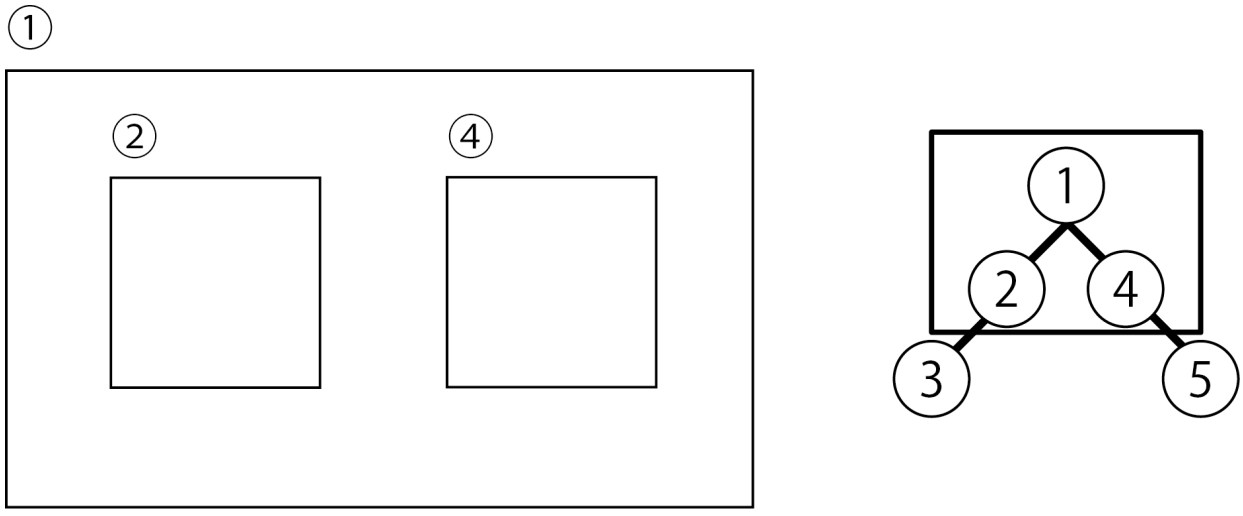

Figure 5.22: Extract polygons 1, 2 and 4

Join polygons 1, 2 and 4 in order from the right.

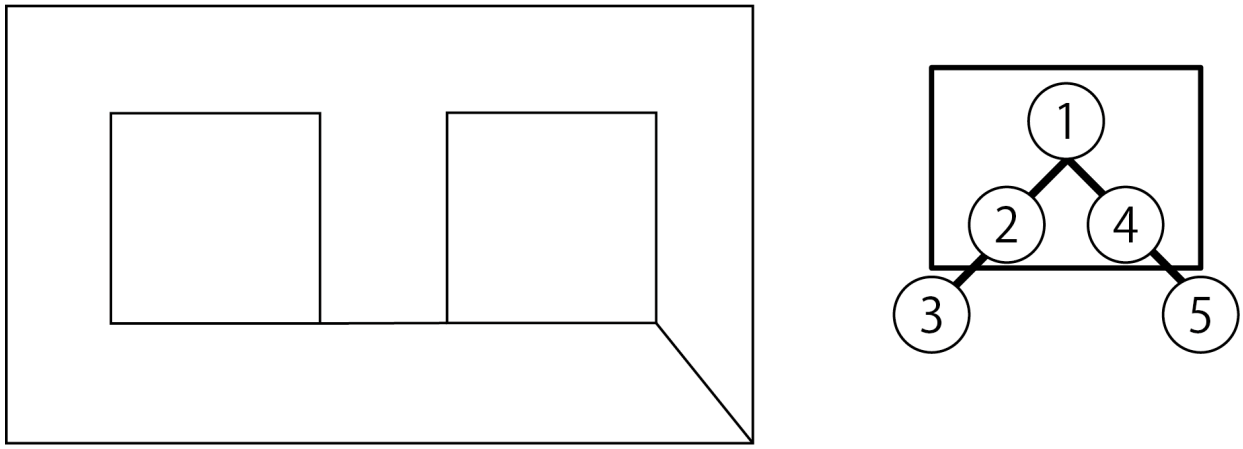

Figure 5.23: Joining polygons 1, 2 and 4

Divide the combined polygon into triangles.

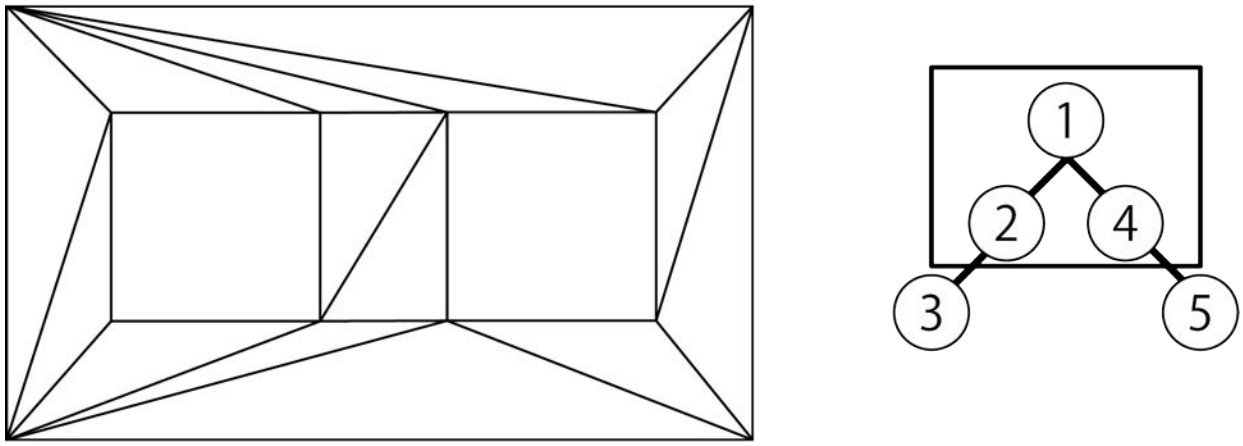

Figure 5.24: Triangulation of combined polygons

Remove the polygon that is divided into triangles. The rest of the parent-child relationship tree is 3 and 5. First, take it out from 3.

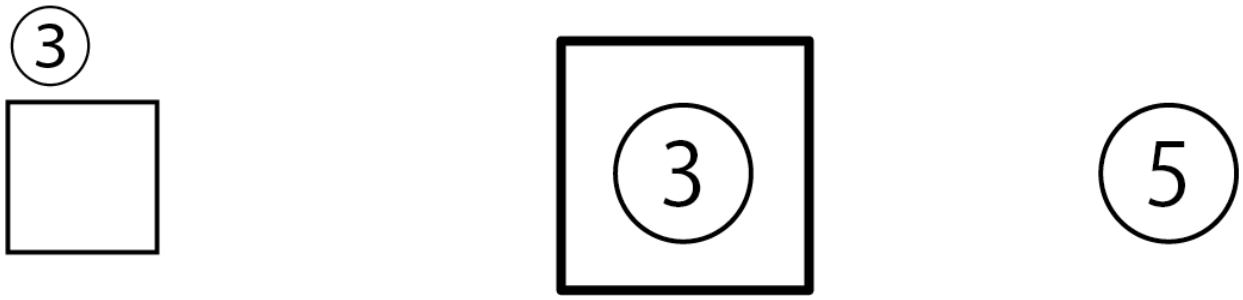

Figure 5.25: Polygon 3

Since 3 has no children, it is divided into triangles as it is.

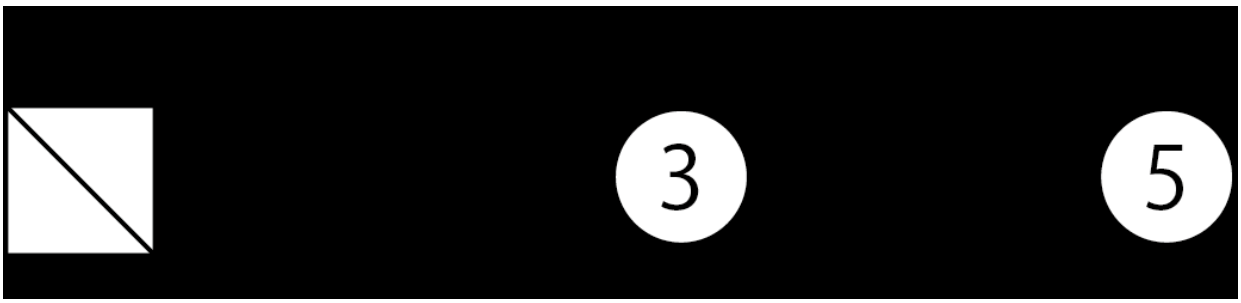

Figure 5.26: Polygon 3 divided into triangles

Remove the polygon that is divided into triangles. There are only 5 left in the parent-child relationship tree. 5 is a triangle and has no children, so it ends as it is. This completes the triangular division of the nested polygon.

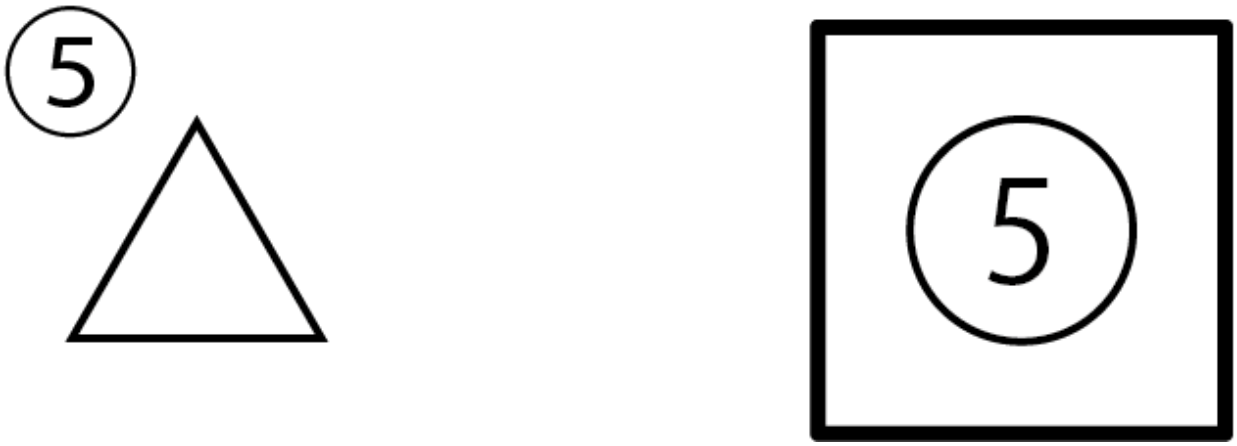

Figure 5.27: Last polygon 5

5.6 Implementation

Let's move on to the explanation of the sample source code that implements all three algorithms explained so far.

5.6.1 Polygon class

First, define a Polygon class that manages an array of polygon vertices. The Polygon class holds information such as the array of vertex coordinates and the direction of the loop, and determines whether a polygon is included in the polygon.

Polygon.cs

```
public class Polygon
{
    // Loop direction
    public enum LoopType
    {
        CW, // clockwise
        CCW, // counterclockwise
        ERR, // Indefinite (no orientation)
    }

    public Vector3 [] vertices; // Vertex array
    public LoopType loopType; // Loop direction

    // ~ omitted ~
}
```

5.6.2 Triangulation class

This is a Triangulation class that actually divides a polygon into triangles. The main is the Triangulate function of the Triangulation class.

Data structure used for triangle division

Data structure definition in Triangulation.cs

```
// Vertex array
List<Vector3> vertices = new List<Vector3>();

// List of vertex numbers (let' s end and start connected)
LinkedList<int> indices = new LinkedList<int>();

// Ear vertex list
List<int> earTipList = new List<int>();
```

It defines vertices that stores the array of vertex coordinates of the polygon to be processed, indices that stores the number (index) of the vertices of the

polygon, and earTipList that stores the ears. Since indices need to refer to the vertices before and after, we use LinkedList, which has the property of a bidirectional list.

Create a hierarchical structure

First, if you are given an array of vertices that make up a polygon from the outside, store it in the list as a Polygon class.

Polygon list

```
// Polygon list
List<Polygon> polygonList = new List<Polygon>();

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

At the beginning of the Triangulate function, sort the Polygon list with polygonal data added in descending order of area of the rectangular area.

Sorted part of Polygon list

```
// Sort in descending order of area of rectangular area in
polygon list
polygonList.Sort((a, b) => Mathf.FloorToInt(
      (b.rect.width   *   b.rect.height)   -   (a.rect.width   *
a.rect.height)
  ));
```

Next, we will pack the sorted Polygon list into the TreeNode class that creates the tree structure.

The part that packs the Polygon list into a TreeNode

```
// Create route (empty)
polygonTree = new TreeNode<Polygon>();

// Create a polygonal hierarchy
foreach (Polygon polygon in polygonList)
{
   TreeNode<Polygon> tree = polygonTree;
```

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

The TreeNode looks like this: I think it's a common tree structure, but for an empty top-level node, it defines a flag isValue for the existence of its contents.

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;
        }
```

```
    }
}
```

Returning to Triangulation.cs, it is the contents of the CheckInPolygonTree function that creates a hierarchical structure of polygons. It checks whether the passed polygon fits inside its own polygon, and recursively determines whether it fits inside its own children. Makes the passed polygon its own child if it is included in itself but not in its children, or if there are no children.

CheckInPolygonTree関数

```
bool CheckInPolygonTree(TreeNode<Polygon> tree, Polygon polygon,
int lv)
{
    // Does it have a polygon?
    bool isInChild = false;
    if (tree.isValue)
    {
        if (tree.Value.IsPointInPolygon(polygon))
        {
            // If it is included in itself, search if it is also
included in the child
            for(int i = 0; i < tree.children.Count; i++)
            {
                isInChild |= CheckInPolygonTree(
                    tree.children[i], polygon, lv + 1);
            }

             // Make it your own child if it is not included in
the child
            if (!isInChild)
            {
                // Invert the order of the vertices if necessary
                // CW because it is Inner when even nesting
                // CCW because it is Outer when odd nesting
                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
        {
            // not included
            return false;
        }
    }
    else
    {
        // Search only for children if they have no value
        for (int i = 0; i < tree.children.Count; i++)
        {
            isInChild |= CheckInPolygonTree(
                tree.children[i], polygon, lv + 1);
        }

         // Make it your own child if it is not included in the
child
        if (!isInChild)
        {
            // Invert the order of the vertices if necessary
            // CW because it is Inner when even nesting
            // CCW because it is Outer when odd nesting
            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;
}
```

Processing of polygons with holes (combination of inner and outer polygons)

If there are multiple inner polygons, select the vertex with the largest X coordinate among the inner polygons and that polygon. At that time, define a class that collects the X coordinate, vertex number, and polygon number information for judgment.

XMaxData structure

```
/// <summary>
/// X coordinate maximum value and polygon information
/// </summary>
struct XMaxData
{
    public float xmax; // x coordinate maximum value
    public int no; // Polygon number
    public int index; // vertex number of xmax

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

Next, the actual joining process is divided into two processes: sorting multiple polygons in descending order of X coordinate, and joining. The first is the process of sorting multiple polygons in descending order of X coordinate.

CombineOuterAndInners関数

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

    // Find the inner polygon with the vertex with the largest X
coordinate
    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));
    }

    // Sort to the right (in descending order of xmax)
    pairs.Sort((a, b) => Mathf.FloorToInt(b.xmax - a.xmax));

    // Combine from right
    for (int i = 0; i < pairs.Count; i++)
    {
            outer = CombinePolygon(outer, inners[pairs[i].no],
pairs[i].index);
    }

    return outer;
}
```

Next is the join processing part. In the CombinePolygon function, draw a horizontal line from the vertex M with the largest X coordinate of the inner polygon and find the line segment of the outer polygon that intersects that line.

Early stage of CombinePolygon function

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

    // Find the intersection
    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))
    {
      ~ Omitted ~
```

GeometryUtil.GetIntersectionPoint, a function that finds the intersection of line segments M and I and the line segment of the outer polygon, is as follows. The point is that the outer polygon is clockwise, so we only look for those whose starting point is above the line segments M and I and whose end point is below. Doing so will prevent the vertices from getting out of order if you select a line segment that connects the outer polygon to the inner polygon in the already joined inner and outer polygons.

GetIntersectionPoint function

```
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; // Next vertex

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

        // The starting point of the intersecting polygonal line
segment must be at least the line segment M, I
        Vector3 diff0 = vstart - start;
        if (diff0.y < 0f)
        {
            continue;
        }

           // The end point of the intersecting polygonal line
```

```
segment is below the line segment M, I
        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;
}
```

After finding the intersection, check if the triangle created from the vertex with the largest X coordinate of the intersecting line segment, vertex M, and intersection I contains other vertices. To determine if a triangle contains vertices, a two-dimensional cross product is used to determine which side of the triangle's line segment the vertices are on. If the vertices are to the right of all lines, they are inside the triangle.

IsTriangle and CheckLine functions in GeometryUtil.cs

```
/// <summary>
/// Returns the positional relationship between the line and the
vertex
/// </summary>
/// <param name="o"></param>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <returns> +1: Right of line -1: Left of line 0: On line </
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 (it> 0D? +1: (it <0D? -1: 0));
}

/// <summary>
/// Triangle (clockwise) and point inside / outside judgment
/// </summary>
/// <param name="o"></param>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <param name="p3"></param>
/// <returns> +1: outside -1: inside 0: online</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);
}
```

Now, the continuation of Combine Polygon. After finding the intersection, it is judged whether there are other vertices in the triangle, but the direction of the connection of the triangle is clockwise because the inside and outside judgment is made using the outer product.

Middle 1 of CombinePolygon function

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

    // Get the rightmost vertex of the intersecting line segment
    int pindex;
    Vector3[] triangle = new Vector3[3];
    if (outer[index0].x > outer[index1].x)
    {
        pindex = index0;
        // The triangle will be reversed depending on the vertex
of the selected line segment, so adjust it so that it is
clockwise.
        triangle[0] = M;
        triangle[1] = outer[pindex];
        triangle[2] = intersectionPoint;
    }
    else
    {
        pindex = index1;
        triangle[0] = M;
        triangle[1] = intersectionPoint;
        triangle[2] = outer[pindex];
    }
```

If the intersection I and the vertex with the largest X coordinate of the line segment are the same, there is nothing to block from the vertex M, so it is not checked whether the triangle contains other vertices. If they are not the same, it is checked whether other vertices are included, but since the vertices included in the triangle are recessed vertices, the inclusion judgment is performed while satisfying that condition. If the triangle contains multiple vertices, the vertex with the smallest angle between the line segments M and I and the line segment M and the corresponding vertex is selected and stored in the finalIndex.

Middle 2 of CombinePolygon function

```
Vector3 P = outer[pindex];

int finalIndex = pindex;

// If the intersection and P are the same, there is nothing to
block, so do not check the triangle
if((Vector3.Distance(intersectionPoint, P) > float.Epsilon))
{
    float angleMin = 361f;

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

        // Convex vertex / Reflective vertex check
         int prevIndex = (i == 0)? outer.Length --1: i --1; //
Previous vertex
        int nextIndex = (i + 1)% outer.Length; // Next vertex
        int nowIndex = i;

        if (nowIndex == pindex) continue;

        Vector3 outerP = outer[nowIndex];

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

         // Ignore if the coordinates are the same duplicated at
the time of division
             if (Vector3.Distance(outerP, P) <= float.Epsilon)
continue;

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

        // Is it a reflection vertex?
        bool isReflex =! GeomUtil.IsAngleLessPI (nowVertex,
                                                 prevVertex,
                                                 nextVertex);

        // Does the triangle contain "reflection vertices"?
        if ((GeomUtil.IsInTriangle(outerP,
                                   triangle[0],
                                   triangle[1],
                                            triangle[2]) <= 0)&&
(isReflex))
        {
```

```
                // Invisible because the vertices are included in
the triangle

                // Find the angle between the M, I and M, outerP
line segments (select the vertex with the shallowest angle)
               float angle = Vector3.Angle(intersectionPoint - M,
outerP - M);
            if (angle < angleMin)
            {
                angleMin = angle;
                finalIndex = nowIndex;
            }
        }
    }
}
```

After finding the vertices (finalIndex) to join, join the vertex arrays of the inner and outer polygons.

1. Create a list of new vertex arrays.
2. Add up to the vertices (finalIndex) to join the outer polygons to the list.
3. Add the inner polygonal vertex array to be joined to the list so that it goes around in order from vertex M.
4. In order to connect the inner polygon to the outer polygon again, duplicate the vertex (finalIndex) to be combined with vertex M and add it to the list.
5. Add the remaining outer polygon vertices to the list and you're done.

Second half of Combine Polygon

```
Vector3 FinalP = outer[finalIndex];

// Join (create a new polygon)
List<Vector3> newOuterVertices = new List<Vector3>();

// Add up to Index that divides outer
for (int i = 0; i <= finalIndex; i++)
{
    newOuterVertices.Add(outer[i]);
}

// Add all inner
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]);
}

// Increase two vertices to split
newOuterVertices.Add(M);
newOuterVertices.Add(FinalP);

// Add the index of the remaining outer
for (int i = finalIndex + 1; i < outer.Length; i++)
{
    newOuterVertices.Add(outer[i]);
}

outer = newOuterVertices.ToArray();
```

Triangular division

When the inner and outer polygons become one polygon, it is finally divided into triangles. First, initialize the index array of vertices and create an ear list.

InitializeVertices function

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

    // Create index array
    resultTriangulationOffset = resultVertices.Count;
    for (int i = 0; i < points.Length; i++)
    {
        Vector3 nowVertex = points[i];
        vertices.Add(nowVertex);

        indices.AddLast(i);

        resultVertices.Add(nowVertex);
```

```
    }

    // Search for convex triangles and ears
    LinkedListNode<int> node = indices.First;
    while (node != null)
    {
        CheckVertex(node);
        node = node.Next;
    }
}
```

The CheckVertex function that determines if a vertex is an ear looks like this:

CheckVertex function

```
void CheckVertex(LinkedListNode<int> node)
{
    // Convex vertex / Reflective vertex check
    int prevIndex = (node.Previous == null) ?
                      indices.Last.Value :
                      node.Previous.Value; // Previous vertex
    int nextIndex = (node.Next == null) ?
                      indices.First.Value :
                      node.Next.Value; // Next vertex
    int nowIndex = node.Value;

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

    bool isEar = false;

    // Is the internal angle within 180 degrees?
            if   (GeomUtil.IsAngleLessPI(nowVertex,   prevVertex,
nextVertex))
    {
        // Ear check
          // Within 180 degrees, the triangle does not contain
other vertices
        isEar = true;
        foreach(int i in indices)
        {
              if ((i == prevIndex) || (i == nowIndex) || (i ==
nextIndex))
                continue;
```

```
            Vector3 p = vertices[i];

            // Ignore if the coordinates are the same duplicated
at the time of division
                        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))
            {
                // Add ears
                earTipList.Add(nowIndex);
            }
        }
        else
        {
               // Exclude if it is no longer an ear when it is
already an ear
            if (earTipList.Contains(nowIndex))
            {
                // Ear removal
                earTipList.Remove(nowIndex);
            }
        }

    }

}
```

The actual triangulation is done in the following EarClipping function. As mentioned above, the vertices are taken out from the top of the ear list and the triangle connected to the front and back vertices is output. Then, the

procedure of deleting the vertices of the ear from the vertex index array and determining whether the vertices before and after are the ears is repeated.

EarClipping function

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

    while (earTipList.Count > 0)
    {
        int nowIndex = earTipList [0]; // Extract top

        LinkedListNode<int> indexNode = indices.Find(nowIndex);
        if (indexNode != null)
        {
            int prevIndex = (indexNode.Previous == null) ?
                              indices.Last.Value :
                                    indexNode.Previous.Value; //
Previous vertex
            int nextIndex = (indexNode.Next == null) ?
                              indices.First.Value :
                                   indexNode.Next.Value; // Next
vertex

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

            // Triangle creation
            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)
            {
                // End because it is the last triangle
```

```
                break;
            }

            // Delete ear vertices
            earTipList.RemoveAt (0); // Remove top
            indices.Remove(nowIndex);

            // Check the vertices before and after
            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 calculation
    for (int i = 0; i < vertices.Count; i++)
    {
        Vector2 uv2 = CalcUV(vertices[i], uvRect);
        resultUVs.Add(uv2);
    }
}
```

Mesh generation

Make the result of triangle division into Mesh. In the EarClipping function, we prepare the necessary vertex array and index array (resultVertices and resultTriangulation) and pour them into Mesh.

MakeMesh function

```
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;
    }
}
```

By the way, I also set the UV coordinates. UV coordinates are assigned within the rectangular area of the polygon.

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 Summary

I have explained the polygonal triangle division by the ear cutting method. It is important to use it, but if you draw a figure with the mouse, it will become a mesh in real time, you can make the outline data of the font a mesh, etc., but since it is not such a fast algorithm, if the number of vertices increases, it will be speedy. There will be problems (calculated in order by the CPU), but I find it interesting that complex polygons can be divided into triangles with a simple algorithm.

See 5.8

- Triangulation by Ear Clipping

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

- Polygon triangulation

https://ja.wikipedia.org/wiki/ Polygon triangulation

Chapter 6 Tessellation & Displacement

6.1 Introduction

In this chapter, we will explain the function called "Tessellation" that divides polygons on the GPU and how to displace the divided vertices by Displacement map.
The sample in this chapter is "Tessellation" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming4
.

6.1.1 Execution environment

- Supported environment of shader model 5.0 or higher that can execute ComputeShader
- If only Tessellation Shader is used, shader model 4.6 or higher compatible environment
- Confirmed operation with Unity 2018.3.9 at the time of writing

6.2 What is Tessellation?

Tessellation is a function that divides polygons on the GPU, which is installed as standard in rendering pipelines such as DirectX, OpenGL, and Metal.
Normally, vertices, normals, tangents, UV information, etc. are transferred from the CPU to the GPU and flow to the rendering pipeline, but when processing high polygons, the transfer band between the CPU and GPU is overloaded, and the drawing speed bottle It will be a neck.
Since Tessellation provides the function to divide the mesh on the GPU, it is possible to process polygons that have been reduced to some extent on the CPU, subdivide them on the GPU, and restore them to fine displacement by

Displacement map lookup. Will be.
In this book, I will mainly explain the Tessellation function in Unity.

6.2.1 Each stage of Tessellation

Tessellation adds three stages to the drawing pipeline: "Hull Shader", "Tessellation", and "Domain Shader". Three stages will be added, but there are only two programmable stages, "Hull Shader" and "Domain Shader".

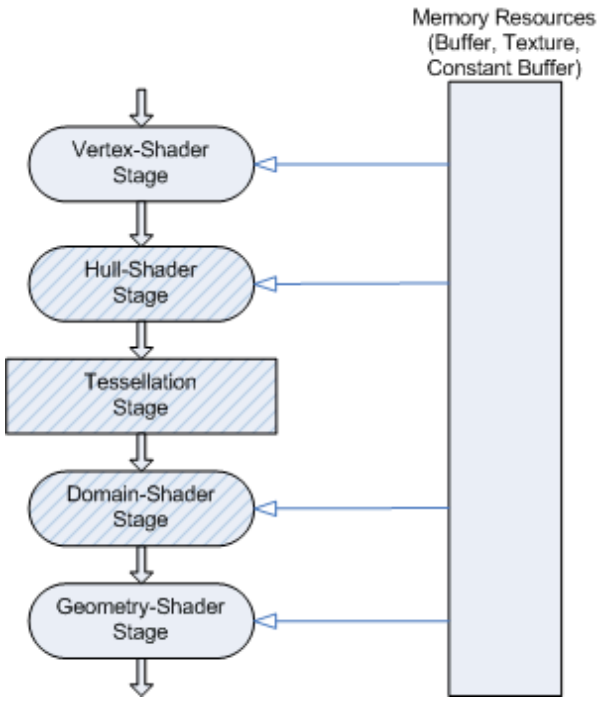

Figure 6.1: Tessellation pipeline Source: Microsoft

[*1] https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-tessellation

Understanding the details of each stage here and implementing Hull Shader and Domain Shader is one way to deepen your understanding of Tessellation, but in Unity, Wrapper, which is very convenient, is Surface Shader. It is available in a form that can be incorporated into.
First, let's perform Tessellation and Displacement based on this Surface Shader.

6.3 Surface Shader とTessellation

I will explain about Tessellation supported by Surface Shader with comments in the comments.

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: Specify a function that defines the
number of patch divisions and method as tessEdge
         // As vertex: disp, specify disp for the function that
performs displacement.
        // Called inside the Domain Shader inside the Wrapper
                  #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;

        // A function that specifies the number of divisions and
the division method
        // This function is called per patch, not per vertex
          // Specify the number of edge divisions of the patch
consisting of 3 vertices in xyz,
        // Specify the number of divisions inside the patch in w
and return it
        float4 tessEdge (appdata v0, appdata v1, appdata v2)
        {
               //Tessellation.cginc has a function that defines
three types of splitting methods

             // Tessellation according to the distance from the
camera
            //UnityDistanceBasedTess

             // Tessellation according to the edge length of the
mesh
            //UnityEdgeLengthBasedTess

                // Culling function in UnityEdgeLengthBasedTess
function
            //UnityEdgeLengthBasedTessCull

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

              // This is the disp function specified in the
Displacement processing function.
        // This function is in Wrapper after Tessellator
        // Called in the Domain Shader.
         // All the elements defined in appdata in this function
are accessible, so
           // Displacement and other processing such as vertex
modulation are performed here.
        void disp (inout appdata v)
        {
             // Here, we are performing vertex modulation in the
```

```
normal direction using the Displacement map.
            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"
}
```

Displacement processing using Surface Shader is realized with the above Shader. You can get great benefits with a very cheap implementation.

6.4 Vertex/Fragment Shaderと Tessellation

Implementation when writing each Tessellation stage in Vertex / Fragment Shader.

6.4.1 Hull Shader Stage

The Hull Shader is a programmable stage, called immediately after the Vertex Shader. Here, we mainly define "division method" and "how many

divisions".
The Hull Shader consists of two functions, a "control point function" and a "patch constant function", which are processed in parallel by the GPU. The control point is the control point of the division source, and the patch has the topology to divide. For example, if you want to form a patch for each triangular polygon and divide it with a Tessellator, there are 3 control points and 1 patch.
The control point function works per control point, and the patch constant function works per patch.

Tessellation.Shader

```
#pragma hull hull_shader

// Structure used as input of hull shader system
struct InternalTessInterp_appdata
{
    float4 vertex : INTERNALTESSPOS;
    float4 tangent : TANGENT;
    float3 normal: NORMAL;
    float2 texcoord : TEXCOORD0;
};

// Tessellation coefficient structure defined and returned by
the patch constant function
struct TessellationFactors
{
    float edge[3] : SV_TessFactor;
    float inside : SV_InsideTessFactor;
};

// hull constant shader (patch constant function)
TessellationFactors                                    hull_const
(InputPatch<InternalTessInterp_appdata, 3> v)
{
    TessellationFactors o;
    float4 tf;

     // Split Utility function explained in the comment at the
time of Tessellation on Surface shader
    tf = UnityEdgeLengthBasedTessCull(
        v[0].vertex, v[1].vertex, v[2].vertex,
        _EdgeLength, _Displacement * 1.5f
    );
```

```
    // Set the number of edge divisions
    o.edge [0] = tf.x;
    o.edge[1] = tf.y;
    o.edge [2] = tf.z;
    // Set the number of divisions in the center
    o.inside = tf.w;
    return o;
}

// hull shader (control point function)

// Triangular polygon with split primitive type tri
[UNITY_domain("tri")]
// Select the division ratio from integer, fractional_odd,
fractional_even
[UNITY_partitioning("fractional_odd")]
// Topology after division triangle_cw is a clockwise triangle
polygon Counterclockwise is triangle_ccw
[UNITY_outputtopology("triangle_cw")]
// Specify the patch constant function name
[UNITY_patchconstantfunc("hull_const")]
// Output control point. 3 outputs for triangular polygons
[UNITY_outputcontrolpoints(3)]
InternalTessInterp_appdata hull_shader (
    InputPatch<InternalTessInterp_appdata,3> v,
    uint id : SV_OutputControlPointID
)
{
    return v[id];
}
```

6.4.2 Tessellation Stage

Here, the patch is divided according to the tessellation factor (Tessellation Factors structure) returned by the Hull shader.
The Tessellation Stage is not programmable, so you cannot write a function.

6.4.3 Domain Shader Stage

Domain Shader is a programmable stage that reflects positions such as vertices, normals, tangents, and UVs based on the processing results of the Tessellation Stage.
A semantic parameter called SV_DomainLocation is input to the Domain Shader, so this parameter will be used to reflect the coordinates.

Also, if you want to perform displacement processing, describe it in Domain Shader. After Domain Shader, the process flows to Fragment Shader and the final drawing process is performed, but if the Geometry Shader function is specified in #pragma, it can also be sent to Geometry Shader.

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;
}

// Domain shader function
[UNITY_domain("tri")]
v2f domain_shader (
    TessellationFactors tessFactors,
    const OutputPatch<InternalTessInterp_appdata, 3> vi,
    float3 bars: SV_DomainLocation
)
{
    appdata v;
    UNITY_INITIALIZE_OUTPUT(appdata,v);
       // Set each coordinate based on the SV_DomainLocation
semantics calculated in the Tessellation stage.
    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;

    // This is the best place to do Displacement processing.
    disp (v);

     // Finally, describe the process just before passing to the
fragment shader.
    v2f o = vert_process (v);
```

```
    return o;
}
```

The above is the process when incorporating Tessellation into Vertex / Fragment Shader.

Finally, I will attach an example. In this example, the fluid RenderTexture output of the grid method described in "Unity Graphics Programming vol.1" is used as the Height map, and the Plane mesh originally included in Unity is subjected to Tessellation and Displacement processing.
Originally it is a Plane mesh with a limited number of vertices, but you can see that the mesh follows with a high particle size without breaking.

Figure 6.2: Fluid Displacement

6.5 Summary

In this chapter, we introduced "Tessellation".
Tessellation is a technology that has withered to some extent, but I think that it is easy to use from performance optimization to creative use, so I hope you will use it where you need it.

6.6 Reference

- https://docs.unity3d.com/jp/current/Manual/SL-SurfaceShaderTessellation.html
- https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-tessellation

Chapter 7 Poisson Disk Sampling

7.1 Introduction

This chapter provides an overview of Poisson Disk Sampling (hereinafter referred to as "PDS") and an explanation of the CPU implementation algorithm.

Fast Poisson Disk Sampling in Arbitary Dimensions (high-speed Poisson disk sampling in any dimension: hereinafter, "FPDS" is used) is adopted for the implementation in the CPU. This algorithm was proposed by Robert Bridson of the University of British Columbia in a 2007 paper submitted to SIGGRAPH.

7.2 Overview

Some people may not know what PDS is in the first place, so this section will explain PDS.

First, plot a large number of points in a suitable space. Next, consider a distance d greater than 0 . At this time, as shown in Fig . 7.1 , the distribution in which all the points are at random positions but all the points are separated by at least d or more is called the Poisson-disk distribution. In other words, even if two points are randomly selected from Figure 7.1 , the distance between them will not always be less than d in any combination . Sampling such a Poisson-disk distribution, in other words, generating a Poisson-disk distribution by calculation, is called PDS.

With this PDS, you can get a random point cloud with uniformity. Therefore, it is used for sampling in filtering processing such as antialiasing, and sampling for determining composite pixels in texture composition processing.

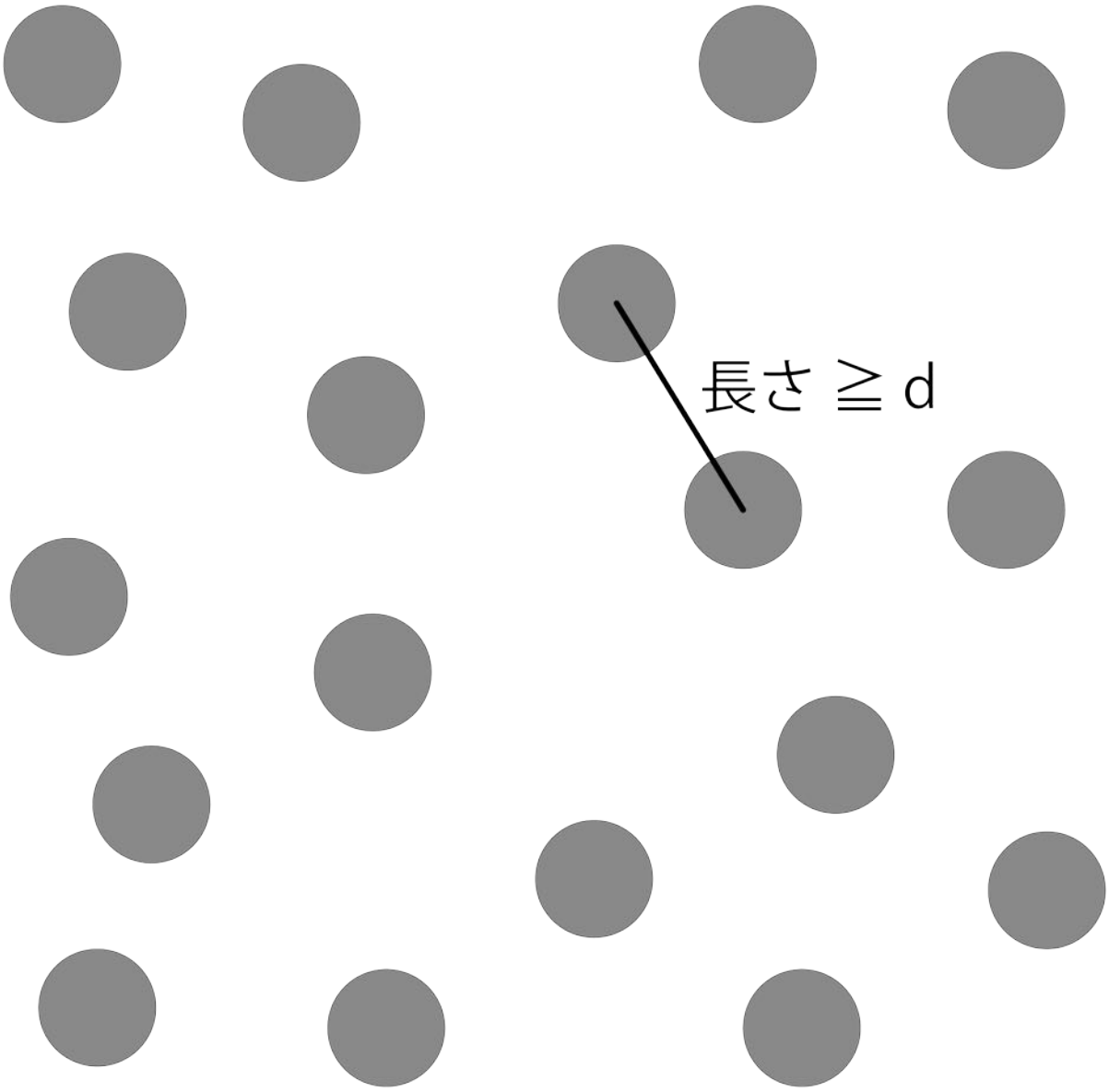

Figure 7.1: Plot random points

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

The biggest feature of FPDS is that it is dimension-independent. Most of the methods proposed so far are based on the assumption of two-dimensional calculation, and it has not been possible to efficiently perform three-

dimensional sampling. Therefore, FPDS was proposed to perform high-speed calculations in any dimension.

FPDS can be calculated in O (N) and can be implemented based on the same algorithm in any dimension. This section describes the FPDS process step by step.

7.3.1 Introduction of FPDS

Three parameters are used in the calculation of FPDS.

- Sampling space range: Rn
- Minimum distance between points: r
- Sampling limit: k

These parameters can be freely entered by the user. The above parameters are also used in the following explanations.

7.3.2 Divide the sampling space into Grids

Divide the sampling space into Grids to speed up the calculation of the distance between points. Here, the number of dimensions of the sampling space is n , and the size of each divided space (hereinafter referred to as "cell") is \ frac {r} {\ sqrt {n}} . Figure 7.2 As shown in \ sqrt {n} is, n the length of each axis in the dimension represents the absolute value of the first vector.

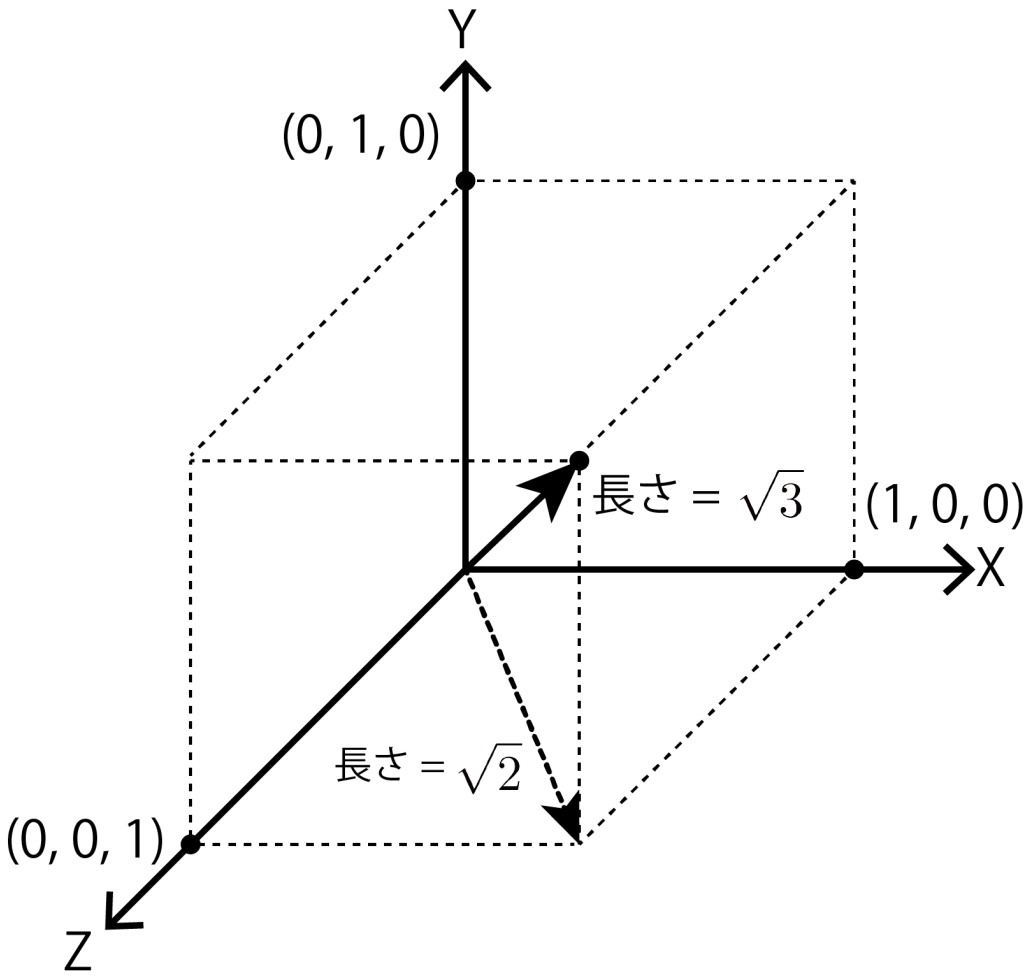

Figure 7.2: When n = 3

Then, the sampled points belong to the cell that contains the coordinates. Since the size of each cell is \ frac {r} {\ sqrt {n}} , the center distance between adjacent cells is also \ frac {r} {\ sqrt {n}} . In other words, by dividing the space by \ frac {r} {\ sqrt {n}} , as you can see from Figure 7.3 , at most one point belongs to each cell. Furthermore, when searching the neighborhood of a certain cell, by searching the surrounding cells by \ pm {n} , all the cells within the minimum distance r can be searched.

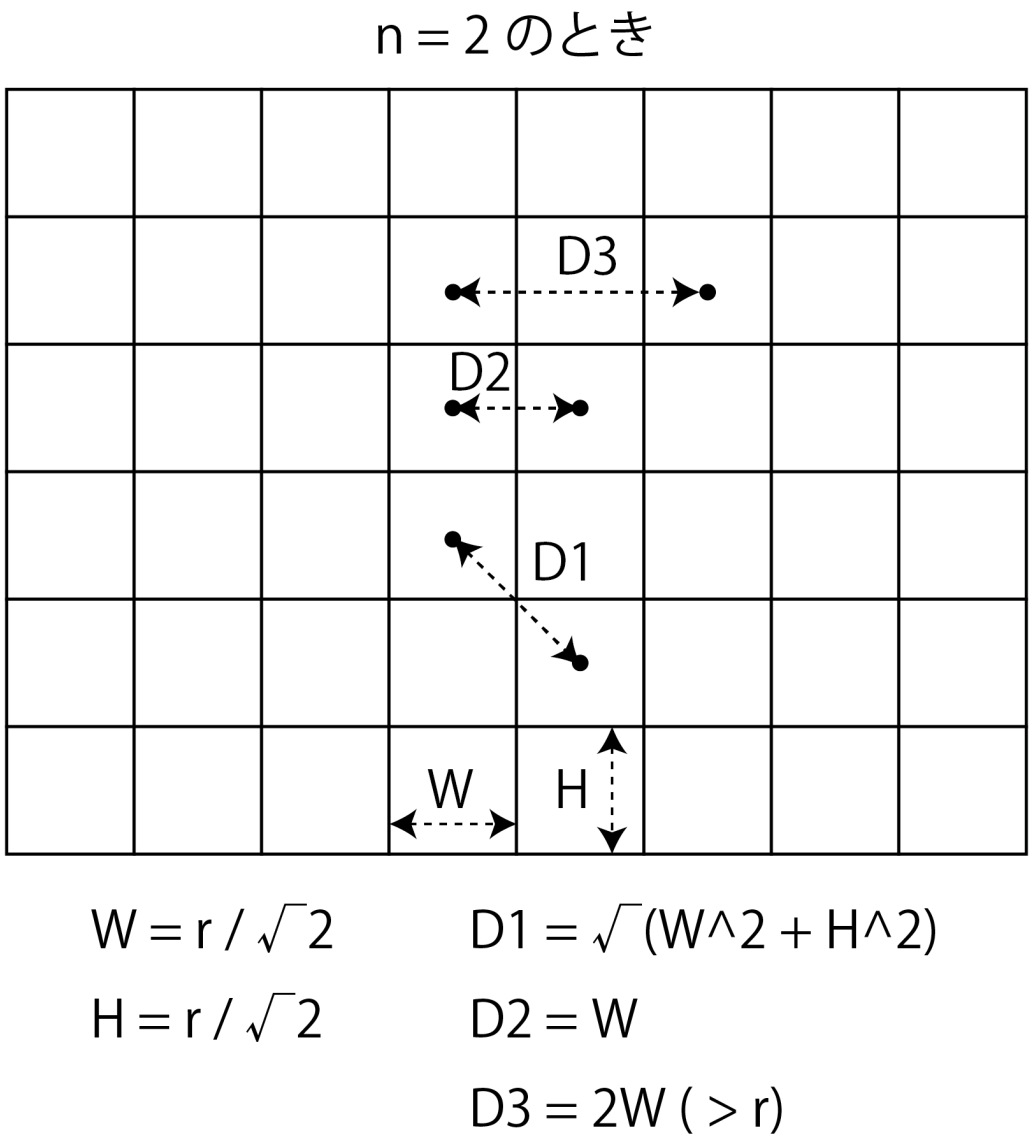

Figure 7.3: Partition of a set at n = 2

Also, this Grid can be represented by an n-dimensional array, so it is very easy to implement. By assigning the sampled point to the cell corresponding to its coordinates, it is possible to easily search for other points that exist nearby.

```
// Get a 3D array that represents a 3D grid
Vector3?[,    ,]    GetGrid(Vector3    bottomLeftBack,    Vector3
topRightForward
```

```
    , float min, int iteration)
{
    // Sampling space
    var dimension = (topRightForward - bottomLeftBack);
    // Multiply the minimum distance by the reciprocal of √3 (I
want to avoid division)
    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
    ];
}

// Get the Index of the cell corresponding to the coordinates
Vector3Int GetGridIndex(Vector3 point, Settings set)
{
      // Calculate Index by dividing the distance from the
reference point by the cell size
    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 Calculate initial sample points

Calculate the first sample point that will be the starting point for the calculation. At this point, no matter which coordinates are sampled, there is no other point closer than the distance r, so one completely random coordinate is determined.

Based on the coordinates of this calculated sample point, it belongs to the corresponding cell and is added to the active list and sampling list. This active list is a list that stores the starting points for sampling. Sampling will be performed sequentially based on the points saved in this active list.

```
// Find one random coordinate
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;
    // Sampling list, eventually returning this List as a result
    bags.SamplePoints.Add(first);
    // Active list, sampling around this List
    bags.ActivePoints.Add(first);
}
```

7.3.4 Select a reference point from the active list

Randomly select Index i from the active list and let the coordinates stored in i be x_i . (Of course, at the very beginning , i is 0 because it is only the point generated in "7.3.3 Calculate the initial sample points" .)

With this x_i as the center, other points will be sampled with respect to nearby coordinates. By repeating this, the entire space can be sampled.

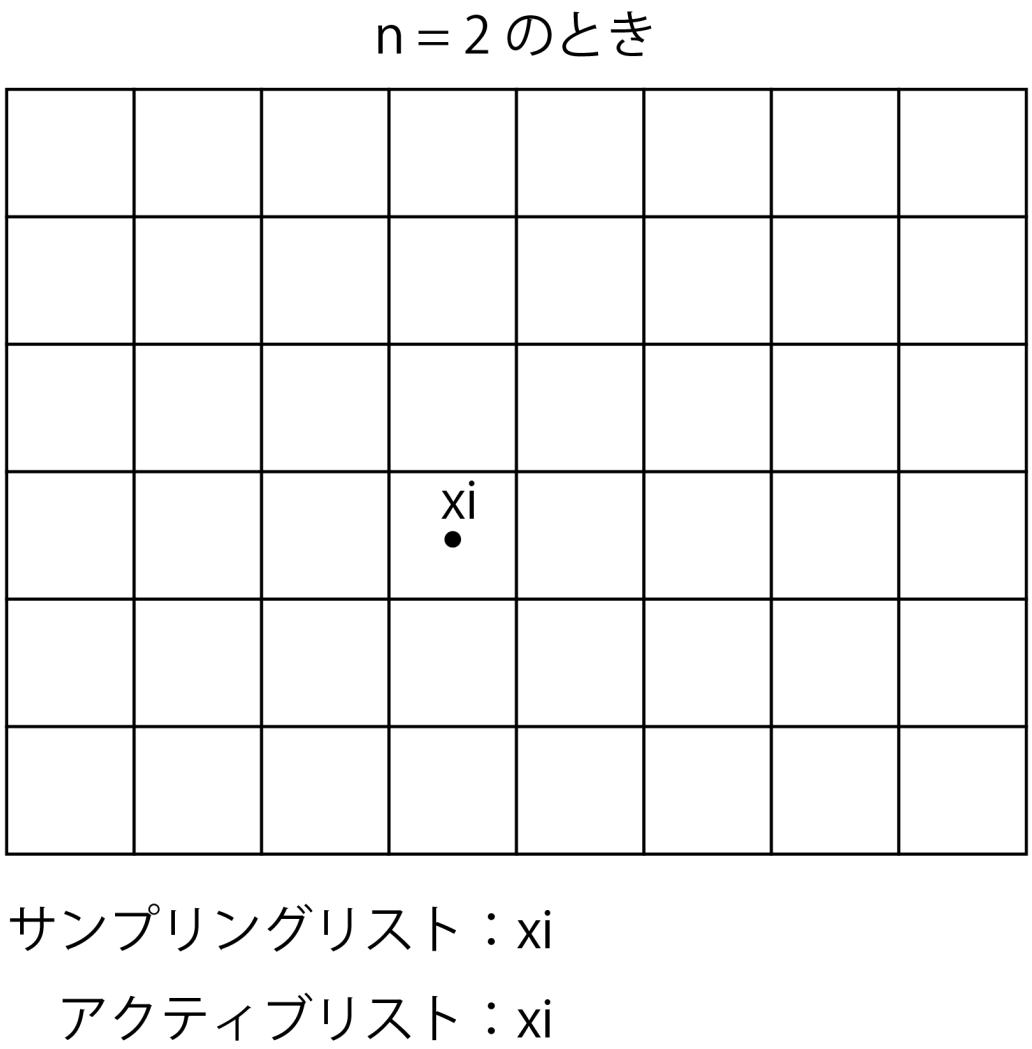

Figure 7.4: Select initial sampling point

```
// Randomly select points from the active list
var index = Random.Range(0, bags.ActivePoints.Count);
var point = bags.ActivePoints[index];
```

7.3.5 Sampling

Random coordinates x ^ {\ prime} _i are calculated within an n-dimensional sphere (a circle for 2D and a sphere for 3D) with a radius of r or more and 2r

or less centered on x_i . Next, check if there are other points with a distance closer than r around the calculated x ^ {\ prime} _i .

Here, the distance calculation for all other points is a process with a higher load as the number of other points increases. Therefore, in order to solve this problem, the Grid generated in "7.3.2 Dividing the sampling space into Grid" is used, and the calculation is performed by examining only the cells around the cell to which x ^ {\ prime} _i belongs. Reduce the amount. If there are other points in the surrounding cells, x ^ {\ prime} _i is discarded, and if there are no other points, x ^ {\ prime} _i belongs to the corresponding cell, and the active list and sampling list Add to.

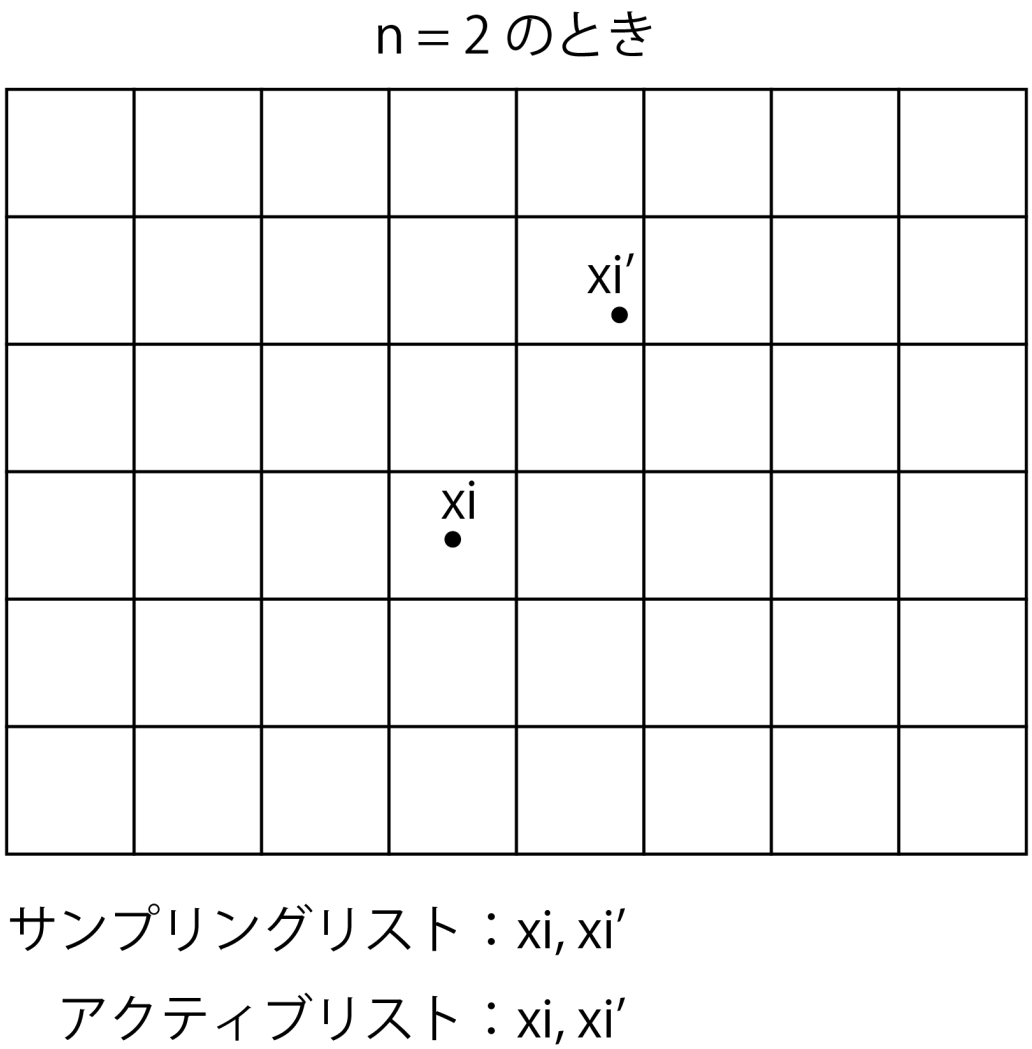

Figure 7.5: Sampling other points relative to the initial sampling point

```
// Find the next sampling point based on the point coordinates
private static bool GetNextPoint(Vector3 point, Settings set,
Bags bags)
{
    // Find a random point in the range r ~ 2r around the point
coordinates
    var p = point +
                GetRandPosInSphere(set.MinimumDistance,  2f  *
set.MinimumDistance);
```

```
    // If it is out of the sampling space, it will be treated as
sampling failure.
    if(set.Dimension.Contains(p) == false) { return false; }

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

    // Calculate the range of Grid to search
    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)
    );

    // Check if there are other points in the surrounding Grid
     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; }

    // Adopted because there are no other points in the vicinity
    bags.SamplePoints.Add(p);
    bags.ActivePoints.Add(p);
    bags.Grid[index.x, index.y, index.z] = p;
```

```
    return true;
}
```

7.3.6 Repeat sampling

With x_i at the center, only one point was sampled in "7.3.5 Sampling" , but this is repeated k times. If x_i for k repeated times, if you were not able to sample even one point, x_i to remove from the active list.

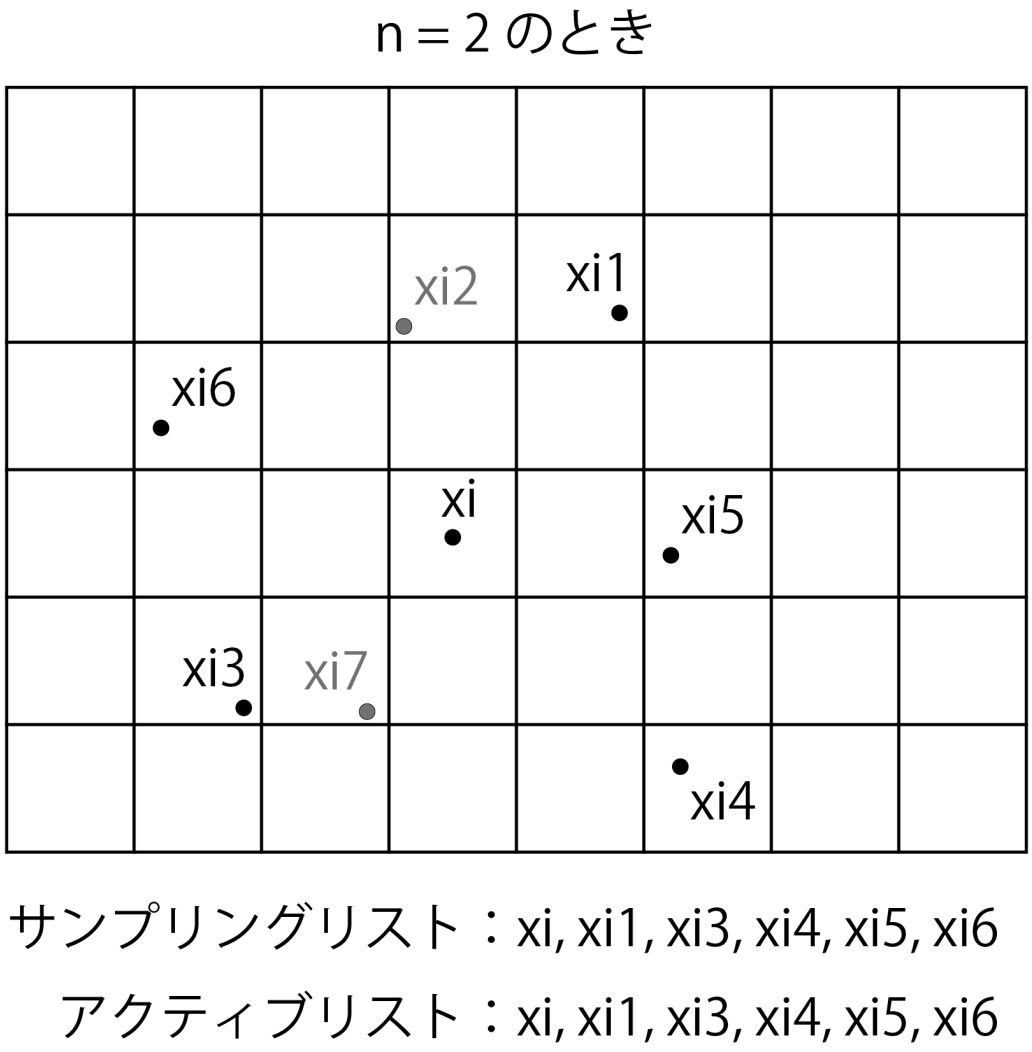

Figure 7.6: K = 7 when the

Then, when the repetition of k is finished, it returns to "7.3.4 Select a reference point from the active list" . You can sample the entire space by repeating this until the active list reaches zero.

```
// Repeat sampling
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;
}
```

In other words, if you briefly explain the overall flow

- Determine the space and divide it into grids
- Determine the initial point appropriately
- Sampling around the initial point
- For the sampled points, sample the periphery in the same way.
- For all points, finish when the surroundings can be sampled

It means that. Since parallelization is not proposed in this algorithm, if the sampling space Rn is wide or the minimum distance r is small, a certain amount of calculation time is required, but it is easy to sample in any dimension. , A very attractive advantage.

7.4 Summary

By the processing up to this point , the sampling result can be obtained as shown in Fig. 7.7 . This image is a circle created and placed by Geometry Shader at the sampled coordinates. You can see that the circles do not overlap and are filled to the full extent.

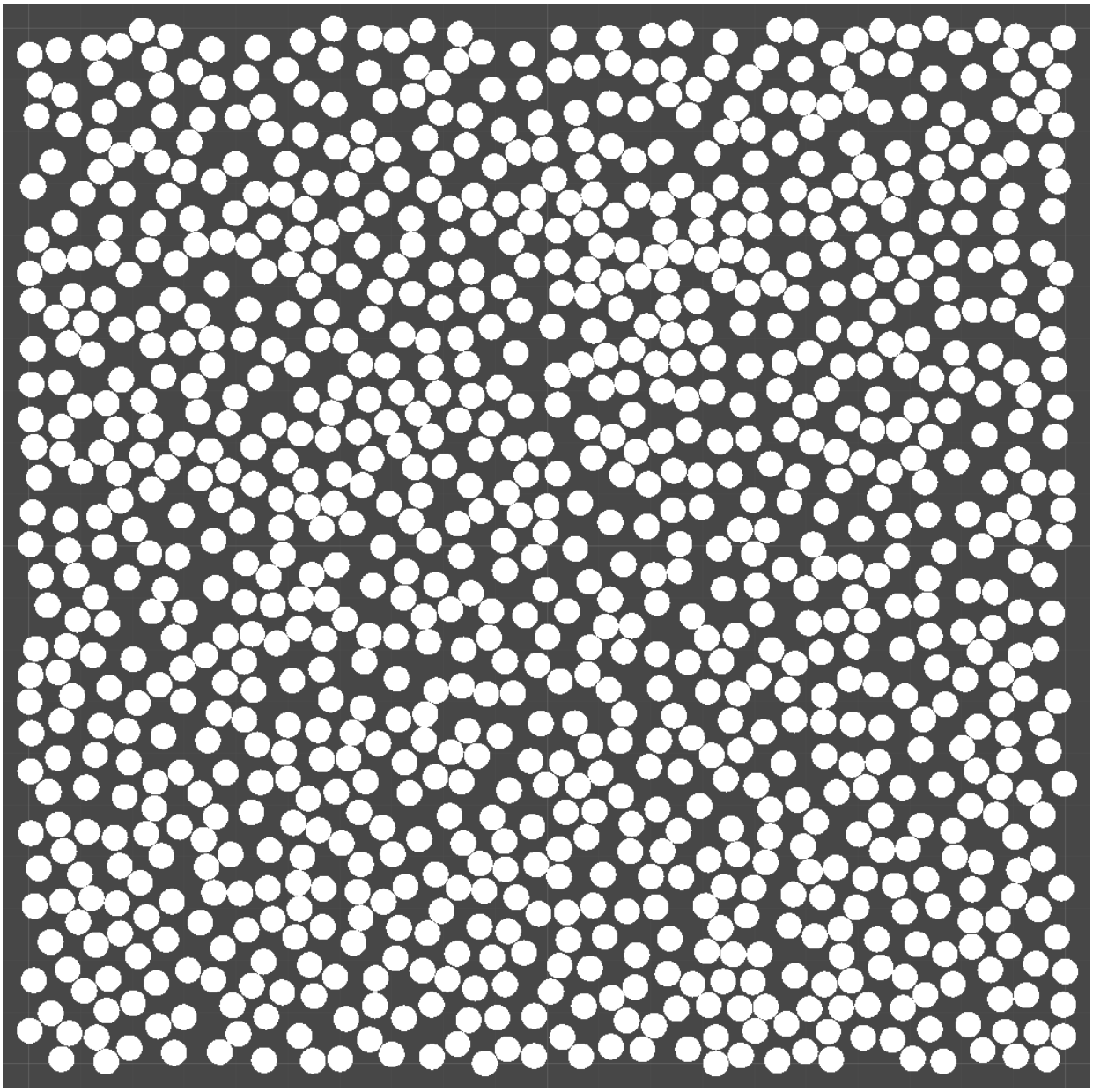

Figure 7.7: Visualization of sampling results

As I mentioned at the beginning, Poisson disc sampling is used in a wide range of places, from antialiasing and image composition to image effects such as Blur and evenly spaced objects. It doesn't give you a clear visual result on its own, but it's often used behind the high-quality visuals we usually see. It can be said that it is one of the algorithms that is worth knowing when doing visual programming.

7.5 Reference

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

About the author

第1章 GPU-Based Space Colonization Algorithm

-Nakamura will reach / @mattatz

A programmer who creates installations, signage, the Web (front-end / back-end), smartphone apps, etc. I am interested in video expression and design tool development.

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

Chapter 2 Limit sets of Kleinian groups-Fu Yong Xiuhe / @fuqunaga

Former game developer, programmer making interactive art. I like the design and development of moderately complicated mechanisms and libraries. I've been sleeping well lately.

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

Chapter 3 GPU-Based Cloth Simulation --Hiroaki Oishi / @irishoak

Interaction engineer. In the field of video expression such as installation, signage, stage production, music video, concert video, VJ, etc., we are producing content that makes use of real-time and procedural characteristics. I have been active several times in a unit called Aqueduct with sugi-cho and mattatz.

- https://twitter.com/_irishoak
- https://github.com/hiroakioishi

- http://irishoak.tumblr.com/
- https://a9ueduct.github.io/

Chapter 4 StarGlow-@XJINE

It is inevitable to keep up with it, and I am living somehow while becoming tattered. Please also use "Unity Shader Programming" for getting started with shaders.

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

Chapter 5 Triangulation by Ear Clipping-@ kaiware007

An interactive engineer who works in an atmosphere. I often post Gene videos on Twitter. I do VJ once in a while. Recently I'm interested in VR.

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

Chapter 6 Tessellation & Displacement-Yoshiaki Sada / @sakope

Former technical artist of a game development company. I like art, design and music, so I turned to interactive art. My hobbies are samplers, synths, musical instruments, records, and equipment. I started Twitter.

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

Chapter 7 Poisson Disk Sampling-@a3geek

Interaction engineer. I am interested in visualization of simulations by CG, and I would like to make visualizations that shake people's emotions more, rather than visualizing them accurately. I like to make it, but I find it more

fun to know more than that. My favorite school classroom is the drawing room or the library.

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