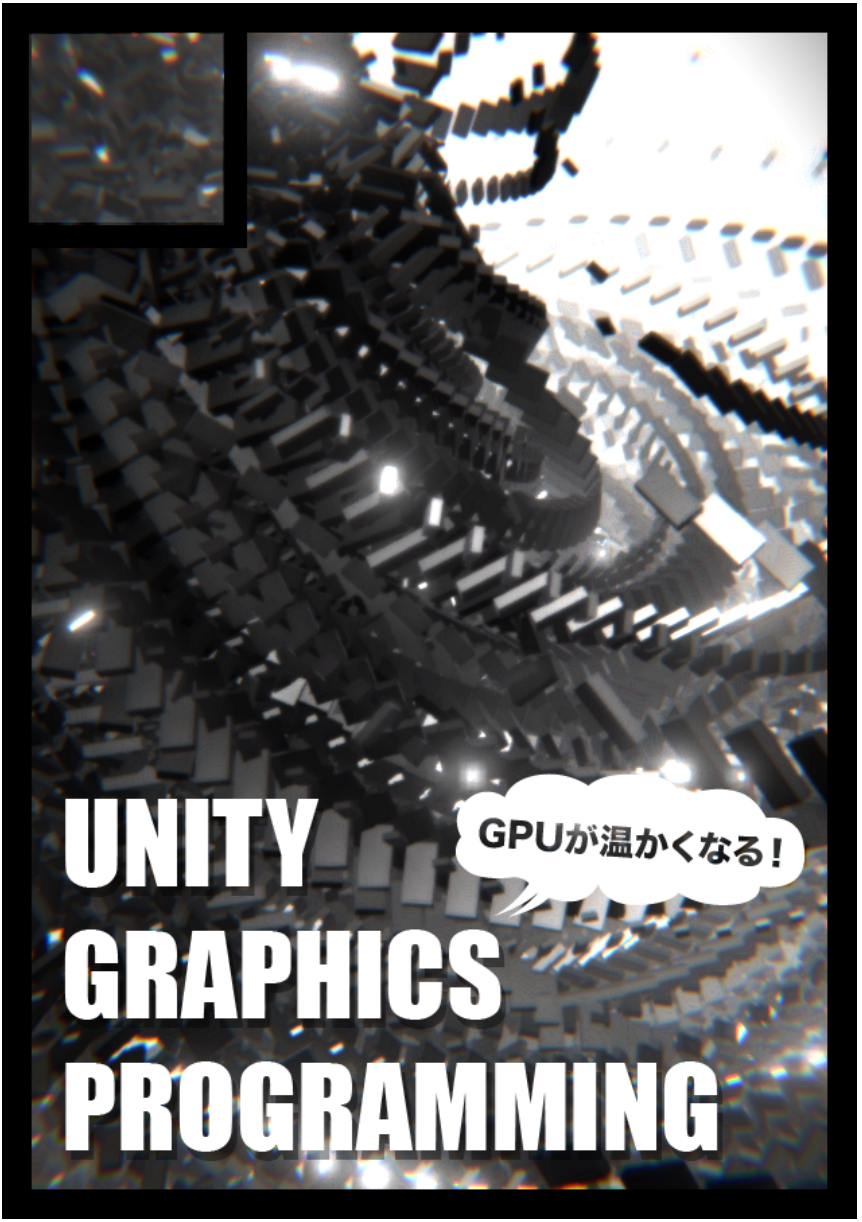

UNITY
GRAPHICS
PROGRAMMING
GPUが温かくなる！

Unity Graphics Programming vol.1

Author: IndieVisualLab

2018-04-22 edition　　IndieVisualLab

Preface

This book mainly describes the technology related to graphics programming by Unity. Graphics programming is broad in one word, and many books have been published even if only Shader technique is taken up. This book also contains articles on various topics that were taken up by the authors' interests, but the visual results should be easy to see and should be useful for creating your own effects. The source code explained in each chapter is available at https://github.com/IndieVisualLab/UnityGraphicsProgramming. You can read this book while you run it.

The degree of difficulty varies depending on the article, and depending on the amount of knowledge of the reader, there may be something that is insufficient or too difficult. Depending on how much you know, it's a good idea to read articles on topics of interest. For those who do graphics programming on a regular basis, I hope it will lead to more effects. For students who are interested in visual coding and have touched Processing, openFrameworks, etc., but for those who still feel a high threshold in 3DCG, introducing Unity as a high expression power and development of 3DCG I would be happy if it was an opportunity to get to know the details.

IndieVisualLab is a circle set up by current (and former) colleagues of IndieVisualLab. We use Unity in-house to program the content of exhibition works that are generally called media art, and we are using Unity, which is a bit different from game-based games. This book may be populated with knowledge useful in utilizing Unity in the exhibited works.

Requests and impressions about the book

If you have any thoughts or concerns about this book, or if you have any other requests (such as wanting to read the explanation about ○○), please feel free to use the web form (`https://docs.google.com/forms/dPleaseletusknowvia/e/1FAIpQLSdxeansJvQGTWfZTBN_2RTuCK_kRqhA6QHTZKVXHCijQnC8zw/viewform`)

or email (lab.indievisual@gmail.com).

目次

第 1 章

Procedural Modeling with Unity

1.1 Introduction

Procedural Modeling is a technique for building 3D models using rules. Modeling generally means using the modeling software Blender or 3ds Max to move the vertices and line segments and manipulating them by hand to sculpt the desired shape. By contrast, the approach of writing rules and obtaining the shape as a result of a series of automated processes is called procedural modeling.

Procedural modeling has been applied in various fields, for example in games it is used for generation of terrain, modeling of plants, construction of cities, etc. It enables content design such as changing the structure.

In the field of architecture and product design the method of procedurally designing shapes is actively used using Grasshopper[*1], which is a CAD software plug-in called Rhinoceros[*2].

With procedural modeling, you can:

- Can create parametric structures
- A flexible, changeable and dynamic model can be incorporated into the content.

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

1.1.1 Creating Parametric Structures

A parametric structure is a structure in which the elements of the structure can be deformed according to certain parameters. For example, in the case of a sphere model, the radius representing the size and the smoothness of the sphere Parameters such as the number of segments to represent can be defined, and by changing those values, a sphere with the desired size and smoothness can be obtained.

Once you have implemented the program that defines the parametric structure, you can get a model with a specific structure in various situations, which is convenient.

1.1.2 A flexible, dynamic model

As mentioned above, in fields such as games, there are many cases where procedural modeling is used to generate terrain and trees. Instead of incorporating what was once exported as a model, it is generated in real time in the content. There are also cases. By using procedural modeling techniques for real-time content, you can create trees that grow toward the sun at any position, or build a city so that buildings line up from the clicked position. These are just illustrations of the type of content that can be realized.

In addition, if you incorporate various patterns of models into the content, the data size will swell, but you can reduce the data size by using procedural modeling to increase the variation of the model.

By learning procedural modeling techniques and building models programmatically, it will be possible to develop the modeling tools themselves.

1.2 Model representation in Unity

In Unity, the geometry data that represents the shape of the model is managed by the Mesh class.

The shape of the model consists of triangles arranged in 3D space, where one triangle is defined by three vertices. The official Unity document explains how to manage the vertex and triangle data of the model in the Mesh class as follows.

> A mesh consists of triangles arranged in 3D space to create the impression of a solid object. A triangle is defined by its three corner points or vertices. In the Mesh class, the vertices are all stored in a single array and each triangle is specified using three integers that correspond to indices of the vertex array. The triangles are also collected together into a single array of integers; the integers are taken in groups of three from the start of this array, so elements 0, 1 and 2 define the first triangle, 3, 4 and 5 define the second, and so on. Any given vertex can be reused in as many triangles as desired but there are reasons why you may not want to do this, as explained below.[a]
>
> [a] https://docs.unity3d.com/Manual/AnatomyofaMesh.html

The model has uv coordinates that represent the coordinates on the texture that are necessary for texture mapping so as to correspond to each vertex, and a normal vector (also called normal) that is needed to calculate the influence of the light source during lighting. Will be included.

Sample repository

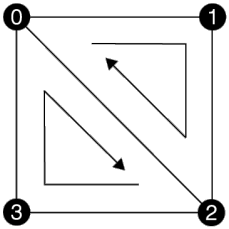
In this chapter, the following Assets/Procedural Modeling in https://github.com/IndieVisualLab/UnityGraphicsProgramming repository is prepared as a sample program.

Since the model description by C# script is the main explanation content, We will continue to explain while referring to the C# scripts under Assets/ProceduralModeling/Scripts.

■Execution environment The sample code in this chapter has been confirmed to work with Unity 5.0 or higher.

1.2.1 Quad

Using the basic model Quad as an example, I will explain how to build a model programmatically. Quad is a square model that combines two triangles with four vertices, and is provided as a Primitive Mesh by default in Unity, but since it is the most basic shape, it is an example to understand the structure of the model. Useful



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

Sample program Quad.cs

First, create an instance of the Mesh class.

▼

```
// Create an instance of Mesh
var mesh = new Mesh();
```

Next, create a Vector3 array that represents the four vertices at the four corners of the Quad. Also, prepare uv coordinate and normal data so that they correspond to each of the four vertices.

▼

```
// Quad: Find half the length so that the width and height of are each size
var hsize = size * 0.5f;

// Quad: Vertex data of
var vertices = new Vector3[] {
    new Vector3(-hsize,  hsize, 0f), // The top left position of the first vertex Quad
    new Vector3( hsize,  hsize, 0f), // Upper right position of second vertex Quad
    new Vector3( hsize, -hsize, 0f), // Lower right position of third vertex Quad
    new Vector3(-hsize, -hsize, 0f)  // Lower left position of fourth vertex Quad
};

// Quad uv coordinate data
var uv = new Vector2[] {
    new Vector2(0f, 0f), // Uv coordinate of the 1st vertex
    new Vector2(1f, 0f), // Uv coordinate of the 2nd vertex
    new Vector2(1f, 1f), // Uv coordinate of the 3rd vertex
    new Vector2(0f, 1f)  // Uv coordinate of the 4th vertex
};
```

```
// Quad normal data
var normals = new Vector3[] {
    new Vector3(0f, 0f, -1f), // 1st vertex normal
    new Vector3(0f, 0f, -1f), // 2nd vertex normal
    new Vector3(0f, 0f, -1f), // 3rd vertex normal
    new Vector3(0f, 0f, -1f)  // 4th vertex normal
};
```

Next, generate triangle data representing the faces of the model. Triangle data is specified by an integer array, and each integer corresponds to the index of the vertex array.

```
// Quad face data, categorize three face vertices as one triangle face
var triangles = new int[] {
    0, 1, 2, // First triangle
    2, 3, 0  // Second triangle
};
```

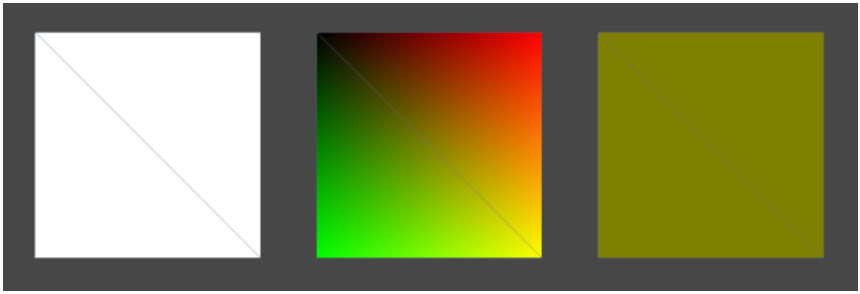
Set the last generated data to the Mesh instance.

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

// Calculate the boundary area occupied by Mesh (required for culling)
mesh.RecalculateBounds();

return mesh;
```

1.2.2 ProceduralModelingBase

The sample code used in this chapter uses a base class called ProceduralModelingBase. In the inherited class of this class, a new Mesh instance is created each time the model parameter (for example, the size representing the width and height in Quad) is changed, and it is applied to the MeshFilter, so that the change result can be immediately confirmed. I can. (This function is realized by using Editor script. ProceduralModelingEditor.cs)

In addition, by changing the enum type parameter called ProceduralModelingMaterial, you can visualize the UV coordinates and normal direction of the model.



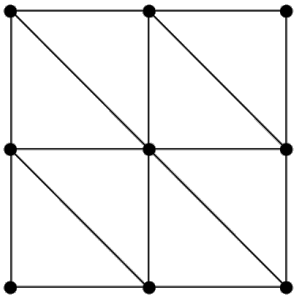
▲図 1.2 From the left, ProceduralModelingMaterial.Standard、ProceduralModeling-Material.UV、ProceduralModelingMaterial.Normal Applied model

1.3 Primitive shape

Now that you understand the model structure, let's create some primitive shapes.

1.3.1 Plane

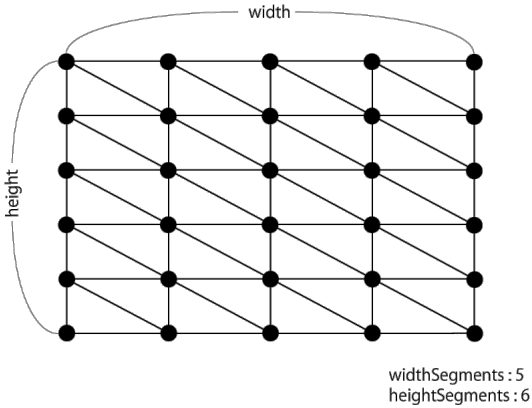
Plane is like quads arranged on a grid.



▲図 1.3 Plane model

Decide the number of rows and columns in the grid, place vertices at the intersections of the grids, construct a Quad so as to fill each grid grid, and combine them to generate one Plane model.

In the sample program Plane.cs, the number of vertical vertices of the Plane heightSegments, the number of vertical vertices widthSegments, and the parameters of the vertical length height and the horizontal length width are prepared. Each parameter affects the shape of the plane as shown in the figure below.



▲図 1.4　Plane parameter

Sample program Plane.cs

First, create the vertex data to be placed at the intersection points of the grid.

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

// The reciprocal of the number of matrices to calculate the ratio of vertices
// on the grid (0.0 to 1.0)
// LW:TODO (reciprocal?)
var winv = 1f / (widthSegments - 1);
var hinv = 1f / (heightSegments - 1);

for(int y = 0; y < heightSegments; y++) {
```

```
    // Percentage of row positions (0.0 to 1.0)
    var ry = y * hinv;

    for(int x = 0; x < widthSegments; x++) {
        // Column position percentage (0.0 to 1.0)
        var rx = x * winv;

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

Next, regarding triangle data, the vertex index set for each triangle is referenced as follows in a loop that follows rows and columns.

▼

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

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

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

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

ParametricPlaneBase

The value of the height (y coordinate) of each vertex of Plane was set to 0, but by operating this height, not only a horizontal surface but also a shape such as uneven terrain and small mountains Can get

The ParametricPlaneBase class inherits the Plane class and overrides the Build function that creates the Mesh. First, generate the original Plane model, call the Depth(float u, float v) function that finds the height with the uv coordinates of each vertex as input, and set the height flexibly by resetting the height.

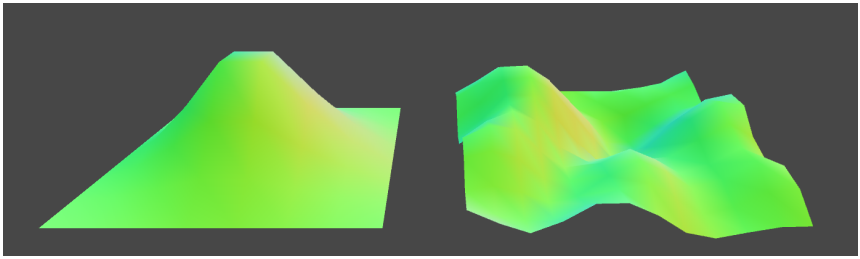
Transforms.

By implementing a class that inherits this ParametricPlaneBase class, you can generate a Plane model whose height changes depending on the vertices.

Sample Program ParametricPlaneBase.cs

```
protected override Mesh Build() {
    // Generate the original Plane model
    var mesh = base.Build();

    // Reset the height of the vertices of the Plane model
    var vertices = mesh.vertices;

    // The reciprocal of the number of matrices to calculate the ratio of vertices on the gr
    var winv = 1f / (widthSegments - 1);
    var hinv = 1f / (heightSegments - 1);

    for(int y = 0; y < heightSegments; y++) {
        // Percentage of row position (0.0 ~ 1.0)
        var ry = y * hinv;
        for(int x = 0; x < widthSegments; x++) {
            // Percentage of column positions (0.0-1.0)
            var rx = x * winv;

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

    // reset the vertex position
    mesh.vertices = vertices;
    mesh.RecalculateBounds();

    // Automatically calculate the normal direction
    mesh.RecalculateNormals();

    return mesh;
}
```

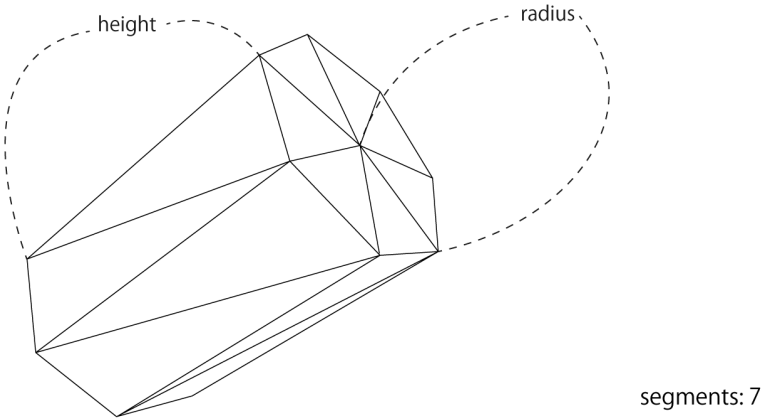
In the sample scene ParametricPlane.scene, the GameObject that uses the class (MountainPlane, TerrainPlane class) that inherits this ParametricPlaneBase is located. Try changing each parameter and see how the shape changes.



▲図 1.5　ParametricPlane.scene　Model generated by MountainPlane class on the left and TerrainPlane class on the right

1.3.2 Cylinder

Cylinder is a cylindrical model with the shape shown in the following figure.



▲図 1.6　Cylinder の構造

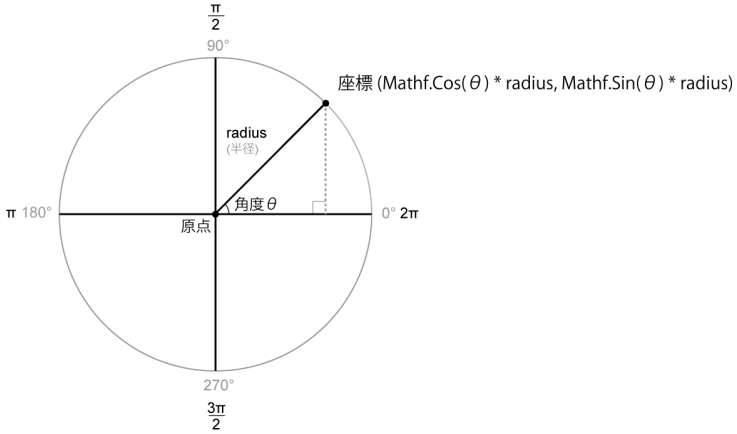
The smoothness of a cylindrical circle can be controlled by segments, and the vertical length and thickness can be controlled by the height and radius parameters, respectively. As shown in the example in the above figure, when 7 is specified for segments, Cylinder becomes a shape in which a regular heptagon is stretched

vertically, and it becomes closer to a circle as the value of segments is increased.

vertices evenly arranged along the circumference

The Cylinder vertices should be evenly spaced around the circle at the end of the tube.

Use trigonometric functions (Mathf.Sin, Mathf.Cos) to place the vertices evenly distributed along the circumference. I will omit the details of trigonometric functions here, but by using these functions, the position on the circumference can be obtained based on the angle.



▲図 1.7 Get the position of a point on the circumference from a trigonometric function

As shown in this figure, the points located on the circle with radius radius from the angle θ (theta) are obtained by (x, y) = (Mathf.Cos(θ) * radius, Mathf.Sin(θ) * radius) can do.

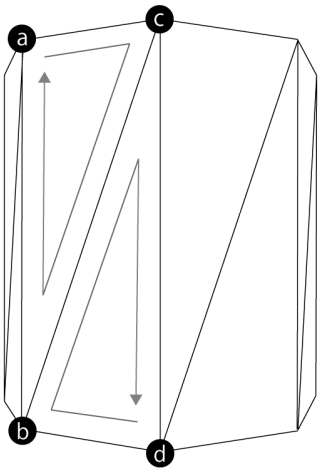
Based on this, the following process is performed to obtain the positions of segments vertices evenly arranged on the circumference of radius radius.

▼

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

```
    // [0.0 ~ 1.0]To[0.0 ~ 2 π]Conversion to radians
    float rad = ratio * PI2;

    // Get the position on the circumference
    float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
    float x = cos * radius, y = sin * radius;
}
```

In Cylinder modeling, vertices are evenly distributed along the circumference of the cylinder at the ends and the vertices are joined together to form the sides. Just like building a Quad on each side, we take two corresponding vertices from the top and bottom and place the triangles face-to-face to build one side, a rectangle. The side of Cylinder can be imagined as Quads arranged in a circle.



▲図 1.8 Modeling the side of Cylinder Black circles are vertices that are evenly arranged along the circumference of the edge. a to d inside the vertices are index variables that are assigned to the vertices when constructing triangles in the Cylinder.cs program.

Sample program Cylinder.cs

First of all, we will build the side surface, but in the Cylinder class, we have a function GenerateCap for generating data of vertices arranged on the circumference located at the top and bottom.

▼

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

// Top height and bottom height
float top = height * 0.5f, bottom = -height * 0.5f;

// Generates vertex data that constitutes the side surface
GenerateCap(segments + 1, top, bottom, radius, vertices, uvs, normals, true);

// To refer to the vertices on the circle when building the side triangle,
// index is the divisor by which the circle goes around
var len = (segments + 1) * 2;

// Build a side by joining the top and bottom edges
for (int i = 0; i < segments + 1; i++) {
    int idx = i * 2;
    int a = idx, b = idx + 1, c = (idx + 2) % len, d = (idx + 3) % len;
    triangles.Add(a);
    triangles.Add(c);
    triangles.Add(b);

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

Generate CapIn the function, set the vertex and normal data to the variables passed in List type.

▼

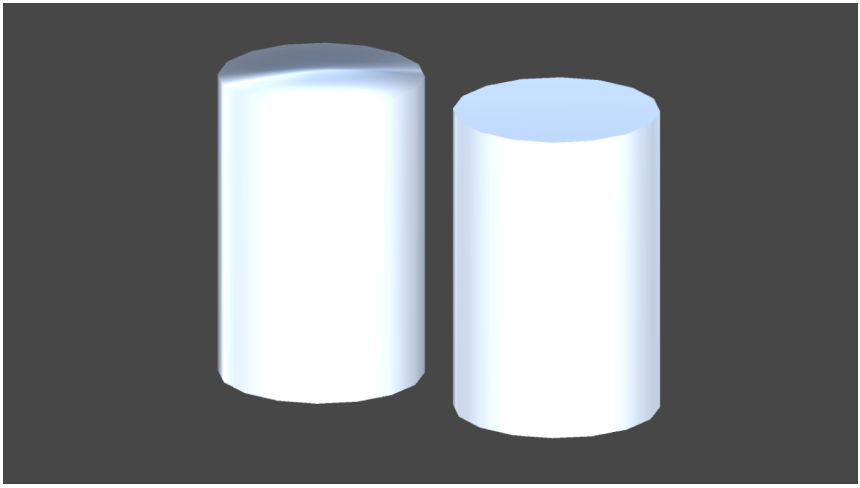
```
void GenerateCap(
    int segments,
    float top,
    float bottom,
    float radius,
    List<Vector3> vertices,
    List<Vector2> uvs,
    List<Vector3> normals,
    bool side
) {
    for (int i = 0; i < segments; i++) {
        // 0.0 ~ 1.0
        float ratio = (float)i / (segments - 1);

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

        // Place vertices evenly along the circumference at the top and bottom
        float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
        float x = cos * radius, z = sin * radius;
        Vector3 tp = new Vector3(x, top, z), bp = new Vector3(x, bottom, z);

        // Upper end
```

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

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

            if(side) {
                // Normals that face the outside of the side
                var normal = new Vector3(cos, 0f, sin);
                normals.Add(normal);
                normals.Add(normal);
            } else {
                normals.Add(new Vector3(0f, 1f, 0f)); // Normal facing up the lid
                normals.Add(new Vector3(0f, -1f, 0f)); // Normal facing down the lid
            }
        }
    }
```

In Cylinder class, you can set with openEnded flag whether to make the model with the top and bottom closed. If you want to close the top and bottom, shape a circular "lid" and plug the ends.

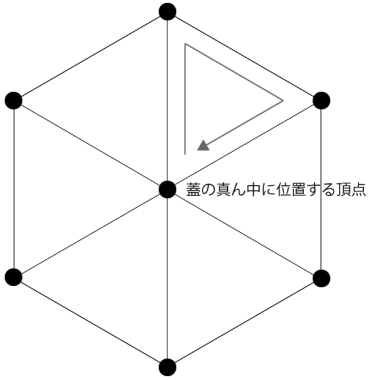
The vertices that make up the surface of the lid do not use the vertices that make up the sides, but a new vertex is created at the same position as the sides. This is to separate the normals between the sides and the lid to give natural lighting. (When constructing the side vertex data, true is set to the side variable of the argument of GenerateCap, false is specified when constructing the lid, and the appropriate normal direction is set.)

If the side and the lid share the same vertex, the side and lid will refer to the same normal, which makes the lighting unnatural.



▲図 1.9 When the side of Cylinder and the top of the lid are shared (left: BadCylinder.cs), and when another vertex is prepared as in the sample program (right: Cylinder.cs), the lighting is unnatural on the left

To model a circular lid, (GenerateCap Generated from a function) Prepare vertices evenly arranged on the circumference and vertices located in the middle of the circle, connect the vertices along the circumference from the middle vertices, and build a triangle like an evenly divided pizza To form a circular lid.



▲図 1.10　CylinderModeling the lid of the case where the segments parameter is 6

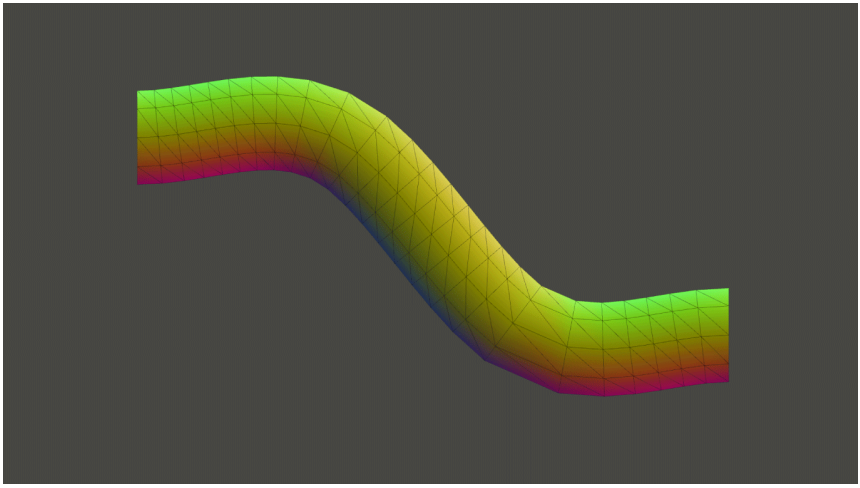
▼

```
// // create top and bottom lids
if(openEnded) {
    // New vertices for lid model, not shared with sides, to use different normals for lighti
    GenerateCap(
        segments + 1,
        top,
        bottom,
        radius,
        vertices,
        uvs,
        normals,
        false
    );

    // The top apex in the middle of the top lid
    vertices.Add(new Vector3(0f, top, 0f));
    uvs.Add(new Vector2(0.5f, 1f));
    normals.Add(new Vector3(0f, 1f, 0f));

    // Middle apex of bottom lid
    vertices.Add(new Vector3(0f, bottom, 0f)); // bottom
    uvs.Add(new Vector2(0.5f, 0f));
    normals.Add(new Vector3(0f, -1f, 0f));

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

    // Offset for not referencing the vertex index of the side
    var offset = len;
```

```
        // Top lid surface
        for (int i = 0; i < len; i += 2) {
            triangles.Add(it);
            triangles.Add((i + 2) % len + offset);
            triangles.Add(i + offset);
        }

        // Bottom lid surface
        for (int i = 1; i < len; i += 2) {
            triangles.Add(ib);
            triangles.Add(i + offset);
            triangles.Add((i + 2) % len + offset);
        }
    }
```

1.3.3 Tubular

Tubular is a tubular model that looks like the following figure.

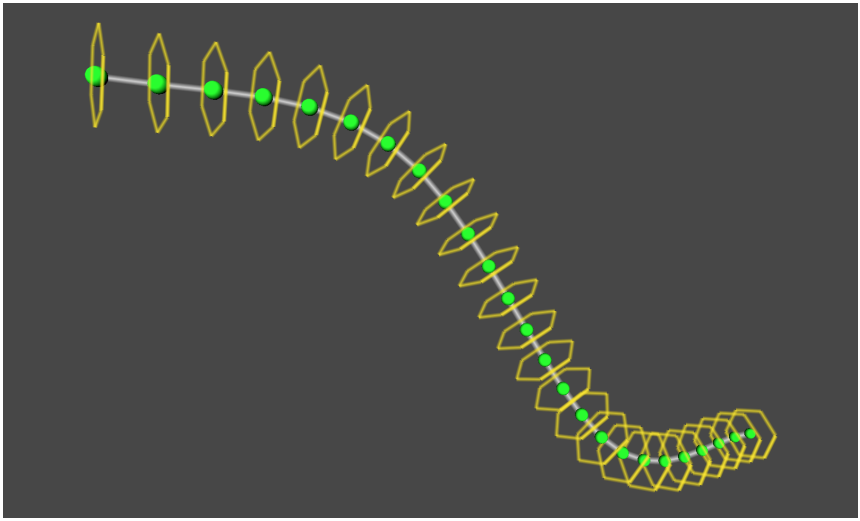


▲図 1.11　Tubular model

The Cylinder model has a straight, cylindrical shape, while the Tubular has a cylindrical shape that does not twist along a curve. In the example of the tree model described below, we use a method of expressing one branch with Tubular and constructing a single tree with that combination, but in the scene where a smoothly bending tubular type is required, Tubular I will play an active role.

Cylindrical structure

The structure of the tubular model is as shown in the figure below.



▲図 1.12 Cylindrical structure: The points that divide the curve along which the Tubular follows are visualized as spheres, and the nodes that make up the side surface are visualized as hexagons.

The curve is divided, the sides are constructed for each node separated by the dividing points, and these are combined to generate one Tubular model.

The side of each knot is the same as the side of Cylinder, because the top and bottom vertices of the side are evenly arranged along the circle and they are connected together to build. You can think of the Tubular type as connecting Cylinders along a curve.

About curves

In the sample program, the base class CurveBase that represents the curve is prepared. Various algorithms have been devised for how to draw a curve in a three dimensional space, and it is necessary to select an easy-to-use method according to the application. In the sample program, the class CatmullRomCurve that inherits the CurveBase class is used.

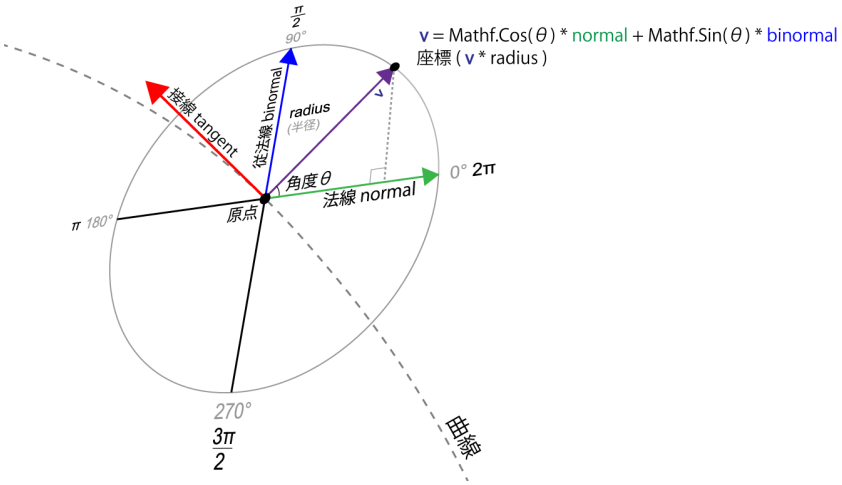
I will omit the details here, but CatmullRomCurve has the feature that it forms a curve while interpolating between points so that it passes through all the control points passed, and it is easy to use because you can specify the point you want to pass through the curve. It has a good reputation for its performance and ease of use.

CurveBase class that represents a curve provides GetPointAt(float) and GetTangentAt(float) functions to obtain the position and slope (tangent vector) of a point on the curve, and specify the value of [0.0 to 1.0] as an argument. By doing so, the position and inclination of the point between the start point (0.0) and the end point (1.0) can be acquired.

Frenet frame

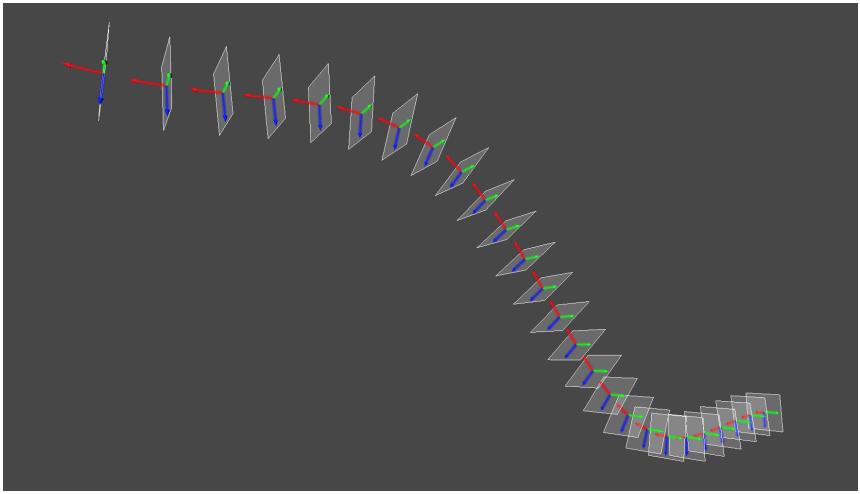
In order to create a twist-free cylindrical shape along a curved line, three orthogonal vectors that change smoothly along the curved line: a tangent vector, a normal vector, and a binormal vector. An array is required. The tangent vector is a unit vector that represents the slope at one point on the curve. The normal vector and the binormal vector are obtained as mutually orthogonal vectors.

With these orthogonal vectors, it is possible to obtain "coordinates on the circle orthogonal to the curve" at a certain point on the curve.



▲図 1.13 Normal (normal) And the binormal (binormal) From this, find the unit vector (v) that points to the coordinates on the circumference. By multiplying this unit vector (v) by the radius radius, you can obtain the coordinates on the circumference of the radius radius orthogonal to the curve.

A set of three orthogonal vectors at one point on this curve is called a Frenet frame.



▲図 1.14 Visualization of Frenet frame array that composes Tubular Represents a frame, the three arrows are tangent Vector, normal Vector, showing binormal vector

Tubular modeling is performed by obtaining vertex data for each node based on the normals and binormals obtained from this Frenet frame, and connecting them.

In the sample program, the CurveBase class has a function ComputeFrenetFrames to generate this Frenet frame array.

Sample program Tubular.cs

The Tubular class has a CatmullRomCurve class that represents a curve, and forms a cylinder along the curve that this CatmullRomCurve draws.The Tubular class has a CatmullRomCurve class that represents a curve, and forms a cylinder along the curve that this CatmullRomCurve draws.

The CatmullRomCurve class requires four or more control points, and when the control points are manipulated, the shape of the curve changes and the shape of the Tubular model changes accordingly.

▼

```
var vertices = new List<Vector3>();
var normals = new List<Vector3>();
var tangents = new List<Vector4>();
```

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

// Get Frenet frame from the curve
var frames = curve.ComputeFrenetFrames(tubularSegments, closed);

// Tubular の頂点データを生成
for(int i = 0; i < tubularSegments; i++) {
    GenerateSegment(curve, frames, vertices, normals, tangents, i);
}
// If you want to create a closed cylinder, place the last vertex at the beginning of the cu
GenerateSegment(
    curve,
    frames,
    vertices,
    normals,
    tangents,
    (!closed) ? tubularSegments : 0
);

// Set uv coordinates from the start point to the end point of the curve
for (int i = 0; i <= tubularSegments; i++) {
    for (int j = 0; j <= radialSegments; j++) {
        float u = 1f * j / radialSegments;
        float v = 1f * i / tubularSegments;
        uvs.Add(new Vector2(u, v));
    }
}

// Build side
for (int j = 1; j <= tubularSegments; j++) {
    for (int i = 1; i <= radialSegments; i++) {
        int a = (radialSegments + 1) * (j - 1) + (i - 1);
        int b = (radialSegments + 1) * j + (i - 1);
        int c = (radialSegments + 1) * j + i;
        int d = (radialSegments + 1) * (j - 1) + i;

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

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

The function GenerateSegment calculates the vertex data of the specified node based on the normals and binormals extracted from the Frenet frame described above and sets it in the variable passed in List type.

```
void GenerateSegment(
    CurveBase curve,
    List<FrenetFrame> frames,
    List<Vector3> vertices,
    List<Vector3> normals,
    List<Vector4> tangents,
    int index
) {
    // 0.0 ~ 1.0
    var u = 1f * index / tubularSegments;

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

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

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

        // Distribute vertices evenly along the circumference
        float cos = Mathf.Cos(rad), sin = Mathf.Sin(rad);
        var v = (cos * N + sin * B).normalized;
        vertices.Add(p + radius * v);
        normals.Add(v);

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

1.4 Complex shape

This section introduces techniques for generating more complex models using the procedural modeling techniques described so far.

1.4.1 plant

Plant modeling is often cited as an application of Procedural Modeling techniques. Tree API [*3] is available for modeling trees in Editor in Unity. There is software dedicated to plant modeling called Speed Tree[*4].

This section deals with modeling trees, which are relatively simple modeling methods among plants.

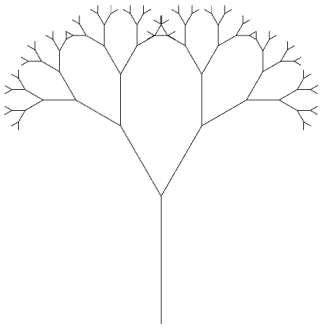
[*3] https://docs.unity3d.com/ja/540/Manual/tree-FirstTree.html
[*4] http://www.speedtree.com/

1.4.2 L-System

L-System is an algorithm that can describe and express the structure of plants. The L-System was advocated by the botanist Aristid Lindenmayer in 1968, and the L for the L-System comes from his name.

L-System can be used to express the self-similarity found in plant shapes.

Self-similarity means that when the shape of a detail of an object is enlarged, it matches the shape of the object seen on a large scale. For example, when observing the branching of trees, there is similarity in the way of branching near the trunk and the way of branching near the tip.



▲図 1.15 Figure that each branch is branched by 30 degrees change It can be seen that the root part and the branch part are similar, but even such a simple figure looks like a tree (sample program LSystem .scene)

The L-System provides a mechanism to develop a complex sequence of symbols by expressing elements by symbols, defining rules for replacing symbols, and applying the rules repeatedly to symbols.

A simple example

- Initial character string: a

To

- Rewrite rule 1: a -> ab
- Rewrite rule 2: b -> a

When rewriting according to
a -> ab -> aba -> abaab -> abaababa -> ...
It produces complicated results with each step.

An example of using this L-System for graphic generation is the LSystem class of the sample program.

In LSystem class, the following operations

- Draw: Draw a line in the direction you are facing
- Turn Left: Rotate θ degrees to the left
- Turn Right: Turn θ degrees to the right

Are available,

- Initial operation: Draw

To

- Rewrite Rule 1: Draw -> Turn Left | Turn Right
- Rewrite Rule 2: Turn Left -> Draw
- Rewrite Rule 3: Turn Right -> Draw

According to, the rules are applied a fixed number of times.

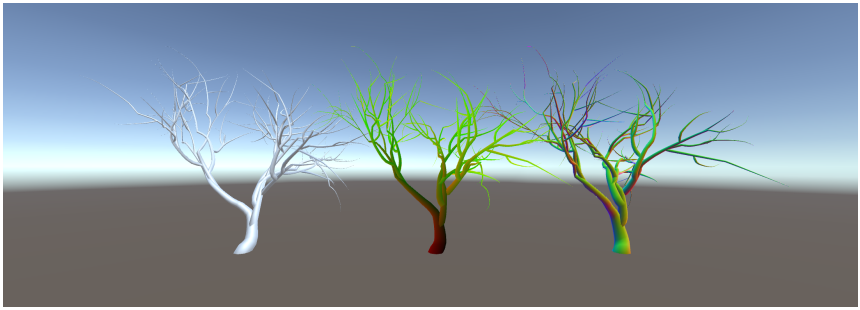
As a result, you can draw a diagram with self-similarity, as shown in the sample LSystem.scene. The property of "recursively rewriting state" of this L-System creates self-similarity. Self-similarity, also called Fractal, is a research area.

1.4.3 Sample program ProceduralTree.cs

As an example of actually applying L-System to a program for generating a tree model, we have prepared a class called ProceduralTree.

In ProceduralTree, the tree shape is generated by recursively calling the routine "advance a branch, then a branch, and then an additional branch" as in the LSystem class described in the previous section.

In the LSystem class of the previous section, regarding the branching of the branch, it was a simple rule to "branch in two directions, left and right, at a fixed angle", Procedural Tree uses random numbers to give randomness to the number of branches and the direction of branching, and sets rules such that branches branch in a complicated manner.



▲図 1.16　ProceduralTree.scene

TreeData class

TreeData class is a class that includes parameters that determine the branching condition of branches, parameters that determine the size of the tree and the fineness of the model mesh. You can design a tree shape by adjusting the parameters of the instance of this class.

Branching

Adjust the branching using some parameters in the TreeData class.

■branchesMin, branchesMax　The number of branches that branch from one branch is adjusted by the branchesMin/branchesMax parameter. branchesMin represents the minimum number of branches and branchesMax represents the maximum number of branches. The number between branchesMin and branchesMax is randomly selected to determine the number of branches.

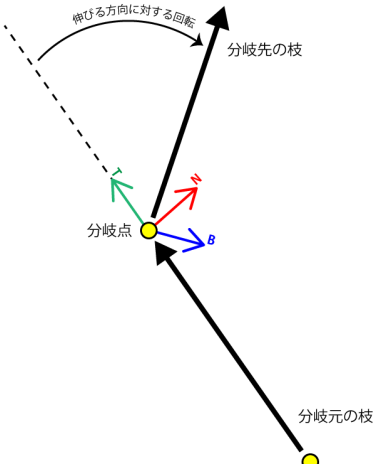
■growthAngleMin, growthAngleMax, growthAngleScale　The direction in which the branching branches grow is adjusted with the growthAngleMin and growthAngleMax parameters. growthAngleMin represents the minimum angle of the branching direction, growthAngleMax represents the maximum angle, and the number between growthAngleMin and growthAngleMax is randomly selected to determine the branching direction.

Each branch has a tangent vector that represents the direction in which it extends, and a normal vector and a binormal vector that are orthogonal to it.

The values randomly obtained from the growthAngleMin and growAngleMax parameters are rotated in the normal vector direction and the binormal vector

direction with respect to the tangent vector extending from the branch point.

By adding a random rotation to the direction tangent vector extending from the branch point, the direction in which the branch destination branch grows is changed, and the branching is complicatedly changed.



▲図 1.17 Random rotation applied to the direction extending from the branch point T arrow at the branch point is the extending direction (tangent vector), N arrow is the normal line (normal vector), B arrow is the binormal line (binormal vector) Represents a random rotation in the normal and binormal directions with respect to the extending direction.

The growthAngleScale parameter is prepared so that the angle of rotation that is randomly applied in the direction in which the branch grows becomes larger as it goes to the tip of the branch. This growthAngleScale parameter strongly influences the angle of rotation as the generation parameter representing the generation of the branch instance approaches 0, that is, the edge of the branch, and increases the angle of rotation.

▼

```
// The branching angle increases as the end of the branch branches
var scale = Mathf.Lerp(
    1f,
    data.growthAngleScale,
    1f - 1f * generation / generations
);
```

```
// rotation in the normal direction
var qn = Quaternion.AngleAxis(scale * data.GetRandomGrowthAngle(), normal);

// rotation in binormal direction
var qb = Quaternion.AngleAxis(scale * data.GetRandomGrowthAngle(), binormal);

// Determine the position of the branch tip by applying qn * qb rotation in the tangent direc
this.to = from + (qn * qb) * tangent * length;
```

TreeBranch class

Branches are represented by the TreeBranch class.

In addition to the parameters of the number of generations (generations) and the basic length (length) and thickness (radius), when you call the constructor by specifying the TreeData for setting the branching pattern as an argument, recursively inside TreeBranch instances will be created.

The TreeBranch branched from one TreeBranch is stored in the child variable of List<TreeBranch> type in the original TreeBranch, and all branches can be traced from the root TreeBranch.

TreeSegment class

Similar to Tubular, the model of one branch is constructed by dividing one curve, modeling the divided nodes as one Cylinder, and connecting them together.

TreeSegment class is a class that represents a segment (Segment) that divides one curve.

▼

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

    // Direction vector tangent that TreeSegment is facing,
    // FrenetFrame with vectors normal and binormal orthogonal to it
    FrenetFrame frame;

    // TreeSegment の位置
    Vector3 position;

    // TreeSegment の幅 (半径)
    float radius;

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

```
    }
}
```

One TreeSegment has FrenetFrame which is a set of vector of the direction in which the node is facing and orthogonal vector, variables that represent position and width, and holds the information required at the top and bottom when building the Cylinder.

Procedural Tree model generation

The model generation logic of ProceduralTree is an application of Tubular. It generates a Tubular model from the TreeSegment array of one branch TreeBranch. Modeling is done by aggregating them into a single model to form an entire tree.


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

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

// Get the full length of a tree
// By dividing the length of the branch by the total length, the height of the uv coordinate
// Set to change from [0.0 to 1.0] from root to branch
float maxLength = TraverseMaxLength(root);

//Recursively traverses all branches and creates a mesh corresponding to each branch
Traverse(root, (branch) => {
    var offset = vertices.Count;

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

    // Generate vertex data from one branch
    for(int i = 0, n = branch.Segments.Count; i < n; i++) {
        var t = 1f * i / (n - 1);
        var v = vOffset + vLength * t;

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

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

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

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

    // Construct a triangle with one branch
    for (int j = 1; j <= data.heightSegments; j++) {
        for (int i = 1; i <= data.radialSegments; i++) {
            int a = (data.radialSegments + 1) * (j - 1) + (i - 1);
            int b = (data.radialSegments + 1) * j + (i - 1);
            int c = (data.radialSegments + 1) * j + i;
            int d = (data.radialSegments + 1) * (j - 1) + i;

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

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

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

For procedural modeling of plants, methods have been devised such as deep in trees alone, and by branching so that the irradiation rate of sunlight is high, a natural tree model is obtained.

For those who are interested in modeling such plants, various methods are introduced in The Algorithmic Beauty of Plants[*5] written by Aristid Lindenmayer who invented L-System, so please refer to it. ..

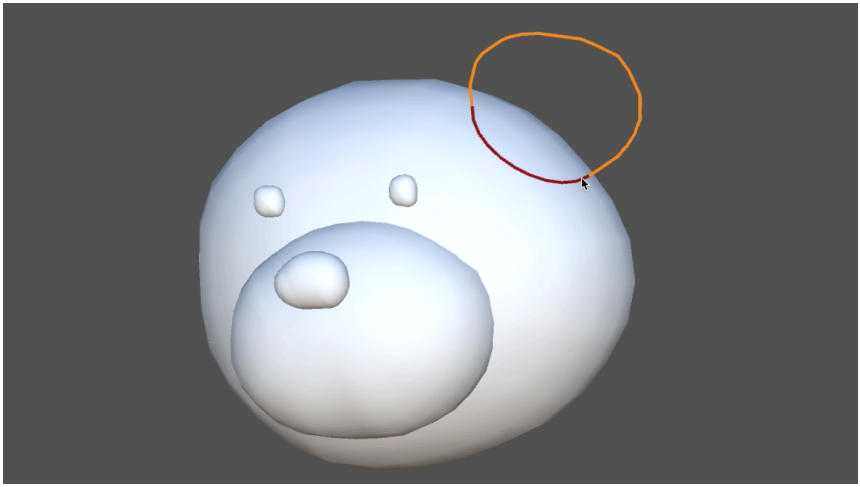
[*5] http://algorithmicbotany.org/papers/#abop

1.5 Application example of procedural modeling

From the examples of procedural modeling introduced so far, you can see the advantage of the technique that "models can be dynamically generated while changing with parameters". Since you can efficiently create various variations of models, you may get the impression that it is a technology for streamlining content development.

However, like the modeling tools and sculpting tools in the world, the technique of procedural modeling can be applied to "generate a model interactively according to the user's input".

As an application example, we would like to introduce "Teddy," a technology that was invented by Takeo Igarashi of the University of Tokyo's Graduate School of Information Engineering to generate a three-dimensional model from contour lines created by handwriting sketches.



▲図 1.18 Unity asset of "Teddy" technology for 3D modeling by handwriting sketch
http://uniteddy.info/ja

This technology was actually used in a game called "Garagaku Masterpiece

Theater: Rakugaki Kingdom" [*6], which was released as software for PlayStation 2 in 2002. The application of "moving as a character" has been realized.

With this technology,

- Define a line drawn on a 2D plane as a contour
- Perform a meshing process called Delaunay Triangulation [*7] on the array of points that make up the contour line
- Apply an algorithm to inflate the mesh on the obtained 2D plane

The 3D model is generated by the procedure. For details of the algorithm, a paper presented at SIGGRAPH, an international conference dealing with computer graphics, has been published. [*8]

A version of Teddy that has been ported to Unity is available on the Asset Store, so anyone can incorporate this technology into their content. [*9]

By using procedural modeling techniques like this, you can develop your own modeling tool, It is also possible to create content that will evolve as the user creates it.

1.6 Summary

With procedural modeling techniques,

- Efficient model generation (under certain conditions)
- Development of tools and contents that interactively generate models according to user operations

We have seen that can be realized.

Unity itself is a game engine, so you can imagine its application in games and video content from the examples presented in this chapter.

However, just as the computer graphics technology itself has a wide range of applications, it can be considered that the technology of model generation has a wide range of applications. As I mentioned at the beginning, procedural modeling methods are used in the fields of architecture and product design, With the development of digital fabrication such as 3D printer technology, opportunities to use designed shapes in real life are increasing at the individual level.

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

In this way, considering in which field the designed shape is used, There may be many places where you can apply procedural modeling techniques.

1.7 reference

- Computers intelligently generate content--what is procedural technology?
- The Algorithmic Beauty of Plants - http://algorithmicbotany.org/papers
- nervous system - http://n-e-r-v-o-u-s.com/

第 2 章

ComputeShader: Getting Started

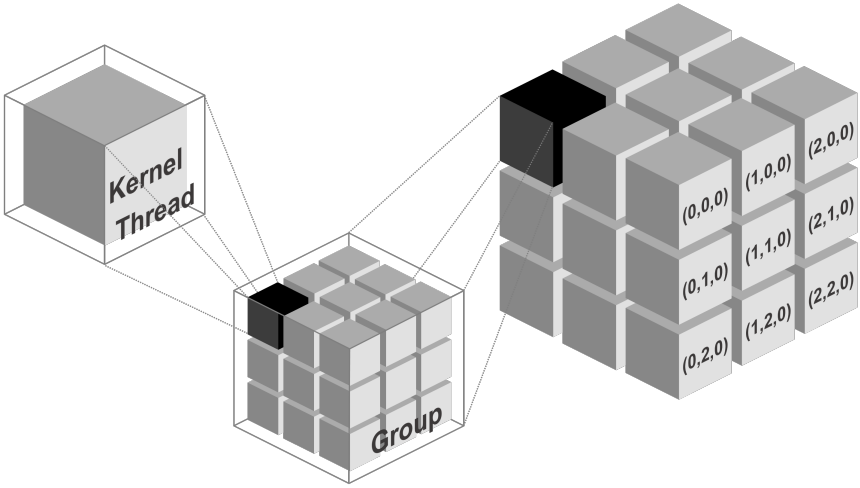
A simple explanation of how to use ComputeShader (hereafter "compute shader" if needed) in Unity. Compute shader is used to execute a large number of operations at high speed by parallelizing simple processing using GPU. It also delegates processing to the GPU, but is characterized by a difference from the normal rendering pipeline. It is often used in CG to represent the movement of a large number of particles.

Some of the content that follows this chapter also uses compute shaders, Knowledge of compute shaders is required to read through them.

Here, in learning Compute Shader, We'll use two simple samples to get you started with the very first step. These do not cover all aspects of compute shaders, so make sure to supplement the information as needed.

In Unity it's called ComputeShader, Similar technologies include OpenCL, DirectCompute, and CUDA. The basic concepts are similar and are very closely related to DirectCompute(DirectX). If you need concepts around the architecture or more details, I think it would be good to collect information on these items as well.

The sample in this chapter is "SimpleComputeShader" from `https://github.com/IndieVisualLab/UnityGraphicsProgramming`. == Kernel, thread, group concept



▲図 2.1　Images of kernels, threads and groups

Handled by Compute Shaders before describing the specific implementation **kernel(Kernel)、thread(Thread)、Group(Group)** It is necessary to explain the concept of.

kernel What is、GPU Refers to a single operation performed by Treated as a function in code (equivalent to the kernel in the sense of general system terminology).

threadIs the unit that executes the kernel. One thread runs one kernel. Compute shaders allow the kernel to run concurrently in multiple threads simultaneously. Threads are specified in three dimensions (x, y, z).

For example,(4, 1, 1) Nara 4 * 1 * 1 = 4 One thread runs at the same time. (2, 2, 1) Then2 * 2 * 1 = 4 One thread runs at the same time. The same four threads are executed, but in some cases it may be more efficient to specify threads in two dimensions like the latter. This will be explained later. For the time being, it is necessary to realize that the number of threads is specified in three dimensions.

Finally**group**Is the unit of execution of a thread. Also, the thread that a group executes is**Group thread**Is called. For example, a group has(4, 1, 1) Suppose you have a thread. This group 2 When there are two groups,(4, 1, 1) Has a thread of.

Groups, like threads, are specified in three dimensions. For example, when a (2, 1, 1) group runs a kernel that runs with (4, 4, 1) threads, The number of

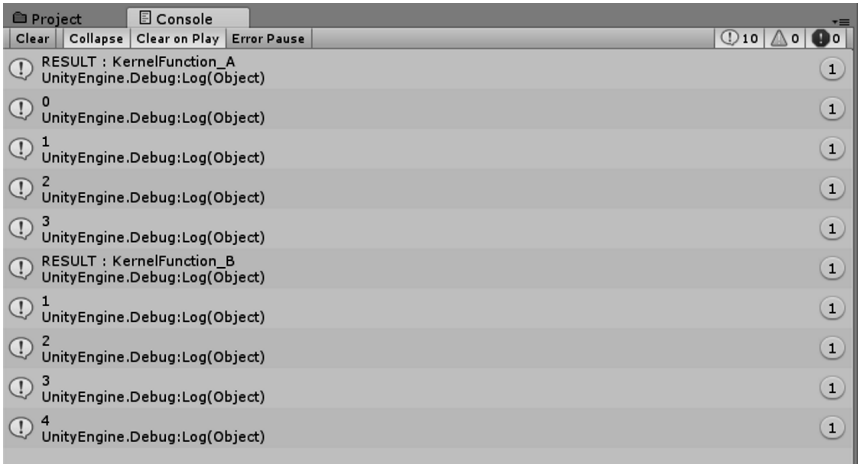
groups is 2 * 1 * 1 = 2. The two groups will each have 4 * 4 * 1 = 16 threads. Therefore, the total number of threads is 2 * 16 = 32.

2.1 Sample (1): Get the result calculated by GPU

Sample (1) "SampleScene_Array" deals with how to execute an appropriate calculation with a compute shader and get the result as an array. The sample includes the following operations:

- Use the compute shader to process multiple data and get the result.
- Implement multiple functions in the compute shader and use them properly.
- Pass values ☒☒ from the script (CPU) to the compute shader (GPU).

The execution result of sample (1) is as follows. Check the operation while reading the source code, since it is only the debug output.



▲図 2.2　sample (1) Execution result of

2.1.1 Compute shader implementation

From here, we will proceed with the explanation using a sample as an example. It's very short, so it's a good idea to go through the Compute Shader implementation first. The basic configuration consists of function definitions, function

implementations, buffers, and optionally variables.

▼ SimpleComputeShader_Array.compute

```
#pragma kernel KernelFunction_A
#pragma kernel KernelFunction_B

RWStructuredBuffer<int> intBuffer;
float floatValue;

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

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

as a feature,**numthreads** Attributes,**SV_GroupID** There are semantics etc., This will be discussed later.

2.1.2 Kernel definition

As I explained earlier, aside from the exact definition,**The kernel is a piece of work performed on the GPU and is treated as a function in code.** Multiple kernels can be implemented in one compute shader.

In this example, the kernel is `KernelFunction_A` No `KernelFunction_B` The function corresponds to the kernel. Also, the function treated as a kernel is `#pragma kernel` Use to define. This distinguishes it from the kernel and other functions.

A unique index is given to the kernel to identify any one of the defined kernels. Index is `#pragma kernel` They are given as 0, 1 …from the top in the order defined by.

2.1.3 Preparing buffers and variables

Create **buffer area** to save the result executed by Compute Shader. Sample variables `RWStructuredBuffer<int> intBuffer`} Is equivalent to this.

Also script (CPU) If you want to give any value from the side、一 Prepare variables as in general CPU programming. この例では変数 `intValue` Is equivalent

to this, and passes the value from the script.

2.1.4 numthreads The number of execution threads by

numthreads Attributes (Attribute) Is the kernel (function) Specifies the number of threads to execute. To specify the number of threads,(x, y, z) Specify with, for example (4, 1, 1) Will run the kernel with 4 * 1 * 1 = 4 threads. (2, 2, 1) Nara 2 * 2 * 1 = 4 Run the kernel in a thread. Both are executed with 4 threads, but the difference and usage will be described later.

2.1.5 Kernel (function) arguments

There are restrictions on the arguments that can be set in the kernel, and the degree of freedom is extremely low compared to general CPU programming.

The value following the argument is called `Semantics`, and in this example `groupID :SV_GroupID` and `groupThreadID :SV_GroupThreadID` are set. Semantics are just to show what kind of value the argument is, and cannot be changed to any other name.

The argument name (variable name) can be freely defined, but it is necessary to set one of the semantics defined when using the compute shader. In other words, it is not possible to implement it by defining an argument of any type and referencing it in the kernel. The argument that can be referred to in the kernel is to select from the defined limited ones.

`SV_GroupID` Indicates in which group the thread executing the kernel is running (x, y, z). `SV_GroupThreadID` Is the (x, y, z) number of the thread that runs the kernel in the group.

例えば (4, 4, 1) In a group of(2, 2, 1) When executing the thread of `SV_GroupID` Is (0 ~ 3, 0 ~ 3, 0) Returns the value of. `SV_GroupThreadID` Is (0 ~ 1, 0 ~ 1, 0) Returns the value of.

In addition to the semantics set in the sample, there are other semantics starting from `SV_~`, which you can use, I will omit the explanation here. I think it's better to read it once you understand the behavior of the compute shader.

- SV_GroupID - Microsoft Developer Network
  - `https://msdn.microsoft.com/ja-jp/library/ee422449(v=vs.85).aspx`
  - You can see the different SV~ semantics and their values.

2.1.6 Contents of kernel (function) processing

In the sample, the thread number is sequentially assigned to the prepared buffer. `groupThreadID` Is given the thread number to run in a group. This kernel runs with (4, 1, 1) threads, so `groupThreadID` is given (0 ~ 3, 0, 0).

▼ SimpleComputeShader_Array.compute

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

This sample runs this thread in a group of (1, 1, 1) (from the script below). That is, run only one group, which contains 4 * 1 * 1 threads. Make sure `group ThreadID.x` is given a value between 0 and 3 as a result.

*In this example, `groupID` is not used, but like threads, the number of groups specified in 3 dimensions is given. Please try using it to check the behavior of the compute shader, such as by substituting it.

2.1.7 Run Compute Shader from script

Execute the implemented compute shader from the script. The items required on the script side are as follows.

- Reference to Compute Shader | `comuteShader`
- Index of kernel to execute | `kernelIndex_KernelFunction_A, B`
- A buffer to save the execution result of compute shader | `intComputeBuff er`

▼ SimpleComputeShader_Array.cs

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

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

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

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

2.1.8 Get the index of the kernel to execute

In order to execute a certain kernel, index information for specifying the kernel is required. Index is `#pragma kernel` It is given as 0, 1… from the top in the order defined by From the script side `FindKernel` It's better to use a function.

▼ SimpleComputeShader_Array.cs

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

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

2.1.9 Create a buffer to save the calculation result

Prepare the buffer area to save the calculation result by Compute Shader (GPU) on the CPU side. Unity Then `ComputeBuffer` Is defined as

▼ SimpleComputeShader_Array.cs

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

`ComputeBuffer` To (1) The size of the area to save, (2) Specify the size per unit of the data to be saved and initialize it. There are 4 areas of size int provided here. This is because the compute shader execution result is saved as int[4]. Resize as needed.

Then, when (1) which kernel implemented in the compute shader executes, (2) Specify which GPU's buffer to use, and (3) specify which CPU's buffer. In this example,(1) `KernelFunction_A` Referenced when is executed, (2) `intBuffer` The buffer area is (3) `intComputeBuffer` Equivalent to.

2.1.10 Pass value from script to compute shader

▼ SimpleComputeShader_Array.cs

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

Depending on what you want to process, you may want to pass a value from the script (CPU) side to the compute shader (GPU) side for reference. Values ☒☒ of most types can be set to variables inside the compute shader using `ComputeShader.Set~`. At this time, the variable name of the argument set in the argument and the variable name defined in the compute shader must match. In this example we are passing 1 for `intValue`.

2.1.11 Run Compute Shader

The kernel implemented (defined) in the compute shader executes with the `ComputeShader.Dispatch` method. Runs the kernel with the specified index in the specified number of groups. The number of groups is specified by X * Y * Z. In this sample, 1 * 1 * 1 = 1 group.

▼ SimpleComputeShader_Array.cs

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

int[] result = new int[4];

this.intComputeBuffer.GetData(result);

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

The execution result of the compute shader (kernel) is obtained with `ComputeBuffer.GetData`.

2.1.12 Confirmation of execution result (A)

Check the implementation on the compute shader side again. This sample runs the following kernels in a 1 * 1 * 1 = 1 group. The threads are 4 * 1 * 1 = 4 threads. Also, 1 is given to `intValue` from the script.

▼ SimpleComputeShader_Array.compute

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

`groupThreadID(SV_GroupThreadID)` Is Now it has a value that tells how many threads this group is running in the group, so In this example, (0 ~ 3, 0, 0) will be entered. So `groupThreadID.x` is 0-3. In other words, `intBuffer[0] = 0` to `intBuffer[3] = 3` will be executed in parallel.

2.1.13 Run a different kernel (B)

When running different kernels implemented in one compute shader, specify the index of another kernel. This example executes `KernelFunction_A` and then `KernelFunction_B`. Furthermore, the buffer area used by `KernelFunction_A` is also used by `KernelFunction_B`.

▼ SimpleComputeShader_Array.cs

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

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

this.intComputeBuffer.GetData(result);

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

2.1.14 Confirmation of execution result (B)

`KernelFunction_B` Executes code similar to the following: このとき `intBuffer` は `KernelFunction_A` Note that we still specify the one used in.

▼ SimpleComputeShader_Array.compute

```
RWStructuredBuffer<int> intBuffer;

[numthreads(4, 1, 1)]
void KernelFunction_B
```

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

In this sample, `KernelFunction_A` By `intBuffer` 0 to 3 are given in order. So after executing `KernelFunction_B`, make sure the value is between 1 and 4.

2.1.15 Discard buffer

You need to explicitly destroy the ComputeBuffer when you are done using it.

▼ SimpleComputeShader_Array.cs

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

2.1.16 Problems not resolved in sample (1)

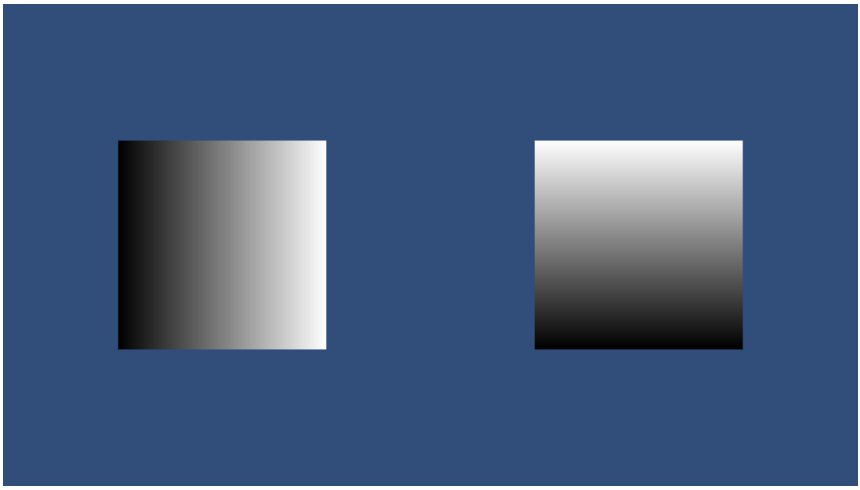
The intention of specifying multidimensional threads or groups is not covered in this sample. For example, a (4, 1, 1) thread and a (2, 2, 1) thread both run 4 threads, These two have the meaning to use properly. This is demonstrated in the sample (2) that follows.

2.2 Sample (2): Make GPU operation result texture

Sample (2) In "SampleScene_Texture", the calculation result of the compute shader is obtained as a texture. The sample includes the following operations:

- Write information to texture using compute shader.
- Effectively utilize multi-dimensional (two-dimensional) threads.

The execution result of sample (2) is as follows. Generates a horizontal and vertical gradient texture.



▲図 2.3　Result of sample (2)

2.2.1 Kernel implementation

See sample for full implementation. In this sample, the following code is executed by the compute shader. Notice that the kernel runs in a multidimensional thread. Since it is (8, 8, 1), there are 8 * 8 * 1 = 64 threads per group. Another big change is that the calculation result is saved in `RWTexture2D<float4>`.

▼ SimpleComputeShader_Texture.compute

```
RWTexture2D<float4> textureBuffer;

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

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

2.2.2 Special arguments SV_DispatchThreadID

sample (1) では `SV_DispatchThradID` I didn't use semantics. It's a little complicated,**"Where is the thread that runs a kernel located among all threads?" (x,y,z)** 」Is shown.

`SV_DispathThreadID` は、`SV_Group_ID * numthreads + SV_GroupThreadID` The value calculated by. `SV_Group_ID` A certain group (x, y, z) Indicated by `SV_GroupThreadID` The threads contained in a group (x, y, z) Indicate.

Specific calculation example (1)

For example, suppose you want to run a kernel that runs in (2, 2, 1) groups with (4, 1, 1) threads. One of them runs in the (2, 0, 0)th thread of the (0, 1, 0)th group. At this time `SV_DispatchThreadID` Is (0, 1, 0) * (4, 1, 1) + (2, 0, 0) = (0, 1, 0) + (2, 0, 0) = (2, 1, 0) Will be.

Specific calculation example (2)

Now let's consider the maximum value. In the (2, 2, 1) group, when the kernel runs with (4, 1, 1) threads, (1, 1, 0) In the second group, (3, 0, 0) The second thread is the last thread. At this time `SV_DispatchThreadID` Is (1, 1, 0) * (4, 1, 1) + (3, 0, 0) = (4, 1, 0) + (3, 0, 0) = (7, 1, 0) Will be.

2.2.3 Write value to texture (pixel)

Since it is difficult to explain in chronological order, check the entire sample while checking.

sample (2) of `dispatchThreadID.xy` Is I've set up a group and a thread to show all the pixels on the texture. Since it is the script side that sets up the group, we need to see it across the script and the compute shader.

▼ SimpleComputeShader_Texture.compute

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

In this sample 512x512 Although the texture of is prepared, `dispatchThreadID.x` But 0 ~ 511 When `dispatchThreadID / width` Is 0 ~ 0.998··· Indicates.

That is `dispatchThreadID.xy` As the value of increases (= pixel coordinates), it will fill from black to white.

The texture consists of RGBA channels and is set between 0 and 1.When all 0s, it becomes completely black, when all 1s, it becomes completely white.

2.2.4 Prepare the texture

Below is the explanation of the implementation on the script side. In sample (1), an array buffer is prepared to store the calculation result of the compute shader. For sample (2), prepare a texture instead.

▼ SimpleComputeShader_Texture.cs

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

Initialize RenderTexture with resolution and format. At this time `RenderTexture.enableRandomWrite` Enable Note that we have enabled writing to the texture.

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

2.2.5 Get the number of threads

You can also get how many threads the kernel can run (thread size) just as you can get the index of the kernel.

▼ SimpleComputeShader_Texture.cs

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

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

2.2.6 Kernel execution

`Dispath` The process is executed by the method. At this time, pay attention to the method of specifying the number of groups. In this example, the number of groups is calculated as "resolution of texture in horizontal (vertical) direction / number of threads in horizontal (vertical) direction".

When considering the horizontal direction, the texture resolution is 512 and the number of threads is 8, so The number of groups in the horizontal direction is 512/8 = 64. Similarly, the vertical direction is 64. Therefore, the total number of groups is 64 * 64 = 4096.

▼ SimpleComputeShader_Texture.cs

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

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

In other words, each group will process 8 * 8 * 1 = 64 (= number of threads) pixels. Since there are 4096 groups, we will process 4096 * 64 = 262,144 pixels. The image is 512 * 512 = 262,144 pixels, which means that all pixels could be processed in parallel.

Running different kernels

The other kernel fills using the y coordinate instead of x. Note that at this time, a value close to 0, a black color appears at the bottom. You may need to consider the origin when working with textures.

2.2.7 Advantages of multidimensional threads, groups

Multidimensional threads and groups work well when you need multidimensional results or multidimensional operations, like in sample (2). If you try to process sample (2) in a one-dimensional thread, you need to compute the vertical pixel coordinates arbitrarily.

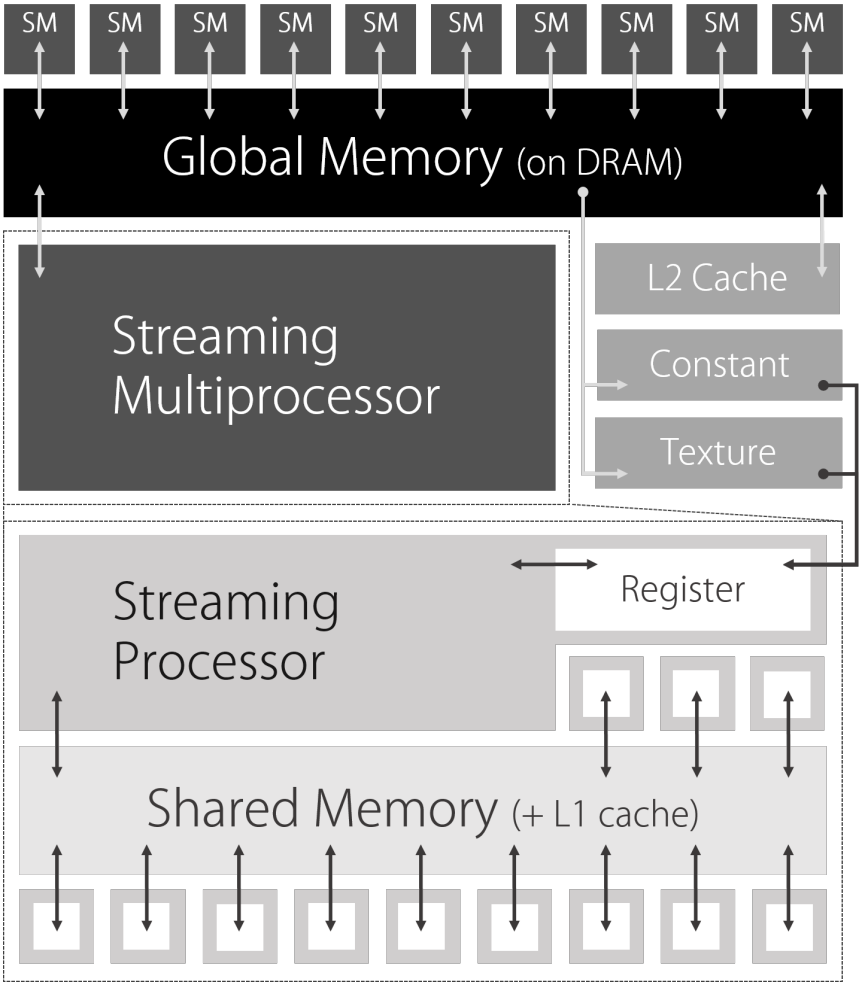
> You can confirm it by actually implementing it, but when there is a stride in the image processing, for example, a 512x512 image,The 513th pixel needs to be calculated as (0, 1) coordinates.

It is better to reduce the number of operations, and the complexity increases with advanced processing. When designing a process that uses a compute shader, it is a good idea to consider whether you can utilize multidimensionality well.

2.3 Additional information for further learning

In this chapter, we used introductory information in the form of explaining a sample about the compute shader. In the future, I will supplement some information that will be necessary for further learning.

2.3.1 GPU Architecture/basic structure



▲図 2.4　GPU Image of architecture

If you have a basic knowledge of GPU architecture and structure, I will introduce it here a little because it helps to optimize it when implementing processing

using Compute Shader.

GPU Are numerous **Streaming Multiprocessor(SM)** Is installed, They share and parallelize and execute the given process.

SM is even smaller **Streaming Processor(SP)** Is installed, SM The process assigned to SP Is calculated by.

SM has **registers** and **shared memory**, You can read and write faster than **Global memory (memory on DRAM)**. Registers are used for local variables that are only referenced inside functions, The shared memory can be referenced and written by all SPs that belong to the same SM.

In other words, grasp the maximum size and scope of each memory, Ideally, you should be able to implement the optimum implementation that can read and write memory without waste and at high speed.

For example, the shared memory that you need to consider most is Storage-class modifiers Defined using `groupshared`. Since this section is an introduction, I will omit specific examples of introduction, but please remember them as techniques and terms necessary for optimization, and use them for subsequent learning.

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

register

It is the memory area that is located closest to the SP and has the fastest access. It consists of 4 bytes, and kernel (function) scope variables are allocated. Since each thread is independent, it cannot be shared.

Shared memory

A memory area that resides on the SM and is managed along with the L1 cache. It can be shared by SPs (= threads) in the same SM, and can be accessed fast enough.

Global memory

A memory area on DRAM, not on the GPU. The reference is slow because it is far from the processor on the GPU. On the other hand, it has a large capacity and data can be read and written from all threads.

Local memory

The memory area on DRAM, not on the GPU, stores data that does not fit in the registers. The reference is slow because it is far from the processor on the GPU.

Texture memory

This is a memory dedicated to texture data, and the global memory is used only for textures.

Constant memory

This is a read-only memory and is used to store the arguments and constants of the kernel (function). It has its own cache and can be referenced faster than global memory.

2.3.2 Tips for efficiently specifying the number of threads

If the total number of threads is larger than the number of data you actually want to process, This is inefficient because it results in threads that are meaninglessly executed (or not processed). Design the total number of threads to match the number of data you want to process as much as possible.

2.3.3 Limits on current specifications

I will introduce the upper limit of the current specifications at the time of writing. Please be aware that it may not be the latest version. However, it is required to implement it while considering such restrictions.

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

Number of threads and groups

I did not mention the limit of the number of threads or groups in the explanation. This is because it changes depending on the shader model (version). It seems that the number that can be paralleled will increase in the future.

- ShaderModel cs_4_x

  - Z The maximum value of 1
  - X * Y * Z The maximum value of 768

- ShaderModel cs_5_0
  - Z The maximum value of 64
  - X * Y * Z The maximum value of 1024

Also, the group limit is (x, y, z) And 65535 respectively.

Memory area

The upper limit of shared memory is per unit group 16 KB, The size of shared memory that a thread can write is limited to 256 bytes per unit.

2.4 reference

Other references in this chapter are:

- #5 GPU structure-Japan GPU Computing Partnership-`http://www.gdep.jp/page/view/252`
- Windows Start with CUDA getting Started - Nvidia Japan - `http://on-demand.gputechconf.com/gtc/2013/jp/sessions/8001.pdf`

第 3 章

GPU implementation of flocking / swarm simulation

3.1 Introduction

In this chapter, we will explain the implementation of group simulation using the Boids algorithm with ComputeShader. Birds, fish and other terrestrial animals sometimes swarm. The movement of this group has regularity and complexity, and it has a certain beauty and has attracted people. In computer graphics, it is not realistic to control the behavior of each individual one by one, and an algorithm for creating groups called Boids was devised. This simulation algorithm is composed of some simple rules and is easy to implement, but in a simple implementation, it is necessary to check the positional relationship with all individuals, and if the number of individuals increases, it becomes the square. The amount of calculation will increase in proportion. If you want to control many individuals, it is very difficult to implement with CPU. Therefore, take advantage of the powerful parallel computing power of the GPU. A shader program called ComputeShader is provided in Unity to perform such general-purpose calculation (GPGPU) by GPU. The GPU has a special storage area called shared memory that can be used effectively by using ComputeShader. In addition, Unity has an advanced rendering function called GPU instancing, and it is possible to draw large numbers of arbitrary meshes. We will introduce a program that controls and draws a large number of Boid objects using the functions that make use of the GPU's computing power.

3.2 Boids algorithm

A group of simulation algorithms called Boids was developed by Craig Reynolds in 1986 and published the following year in 1987 at ACM SIGGRAPH as a paper entitled "Flocks, Herds, and Schools: A Distributed Behavioral Model".

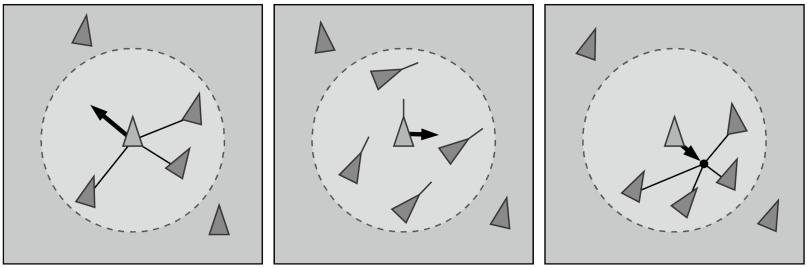
Reynolds is a herd that each individual modifies its own behavior based on the position and moving direction of other individuals around it by perception such as sight and hearing, resulting in complicated behavior. I will focus on that.

Each individual follows three simple rules of behavior:

■1. 分離（Separation） Move to avoid crowding with individuals within a certain distance

■2. 整列（Alignment） An individual within a certain distance moves toward the average of the directions they are facing

■3. 結合（Cohesion） Move to the average position of individuals within a certain distance



▲図 3.1　Boids basic rules

Following these rules, you can program herd movements by controlling individual movements.

3.3 Sample program

3.3.1 Repository

https://github.com/IndieVisualLab/UnityGraphicsProgramming

Assets in the sample Unity project in this book/**BoidsSimulationOnGPU**In a folder**BoidsSimulationOnGPU.unity**Please open the scene data.

3.3.2 Execution condition

The program introduced in this chapter uses ComputeShader, GPU instancing. ComputeShader Works on the following platforms or APIs:

- Windows and Windows Store apps with DirectX11 or DirectX12 graphics API and shader model 5.0 GPU
- IOS with MacOS and Metal Graphics API
- Android, Linux, Windows platforms with Vulkan API
- The latest OpenGL platform (OpenGL 4.3 on Linux or Windows, OpenGL ES 3.1 on Android). (Note that MacOSX does not support OpenGL 4.3)
- Console machines commonly used at this stage (Sony PS4, Microsoft Xbox One)

GPU instancing is available on the following platforms or APIs.

- DirectX 11 and DirectX 12 on Windows
- OpenGL Core 4.1+/ES3.0+ on Windows, MacOS, Linux, iOS, Android
- Metal on MacOS and iOS
- Vulkan for Windows and Android
- PlayStation 4 and Xbox One
- WebGL（WebGL 2.0 APIs necessary）

This sample program uses Graphics.DrawMeshInstacedIndirect method. Therefore, Unity version must be 5.6 or later.

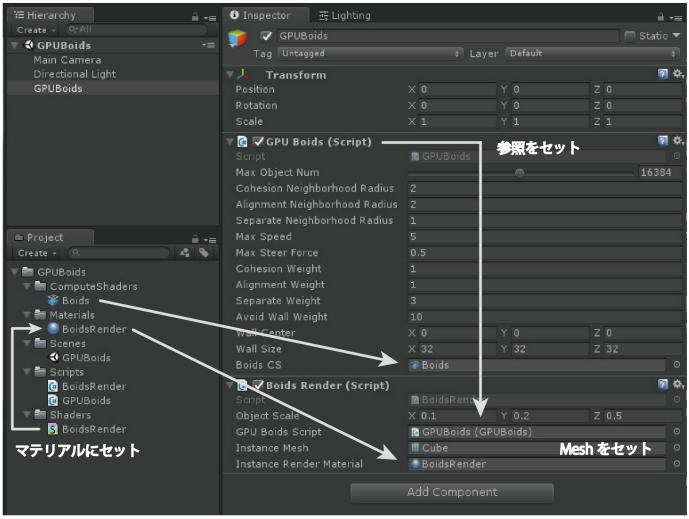
3.4 Description of implementation code

This sample program consists of the following code.

- GPUBoids.cs - Script that controls Compute Shader that simulates Boids

- Boids.compute - ComputeShader that simulates Boids
- BoidsRender.cs - C# script that controls the shader that draws the Boids
- BoidsRender.shader - A shader for drawing objects by GPU instancing

Scripts, material resources etc. are set like this



▲図 3.2　UnityEditor Settings on

3.4.1 GPUBoids.cs

This code manages the Boids simulation parameters, ComputeShader that describes the buffers and calculation instructions required for calculation on the GPU.

▼ GPUBoids.cs

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

public class GPUBoids : MonoBehaviour
{
```

```
// Boid data structure
[System.Serializable]
struct BoidData
{
    public Vector3 Velocity; // 速度
    public Vector3 Position; // 位置
}
// Thread group thread size
const int SIMULATION_BLOCK_SIZE = 256;

#region Boids Parameters
// Maximum number of objects
[Range(256, 32768)]
public int MaxObjectNum = 16384;

// Radius with other individuals to which the bond is applied
public float CohesionNeighborhoodRadius  = 2.0f;
// Radius of alignment with other individuals
public float AlignmentNeighborhoodRadius = 2.0f;
// Radius to other individuals to which separation is applied
public float SeparateNeighborhoodRadius  = 1.0f;

// Maximum speed
public float MaxSpeed        = 5.0f;
// Maximum steering force
public float MaxSteerForce   = 0.5f;

// Weight of force to combine
public float CohesionWeight  = 1.0f;
// Force weights to align
public float AlignmentWeight = 1.0f;
// Separating force weights
public float SeparateWeight  = 3.0f;

// Weight of power to avoid walls
public float AvoidWallWeight = 10.0f;

// Center coordinates of wall
public Vector3 WallCenter = Vector3.zero;
// Wall size
public Vector3 WallSize = new Vector3(32.0f, 32.0f, 32.0f);
#endregion

#region Built-in Resources
// Reference of Compute Shader for Boids simulation
public ComputeShader BoidsCS;
#endregion

#region Private Resources
// A buffer that stores the steering force (Force) of the Boid
ComputeBuffer _boidForceBuffer;
// Buffer that stores basic data of Boid (speed, position)
ComputeBuffer _boidDataBuffer;
#endregion

#region Accessors
// Get the buffer that stores the basic data of Boid
public ComputeBuffer GetBoidDataBuffer()
```

```
    {
        return this._boidDataBuffer != null ? this._boidDataBuffer : null;
    }

    // Get number of objects
    public int GetMaxObjectNum()
    {
        return this.MaxObjectNum;
    }

    // Returns the center coordinates of the simulation area
    public Vector3 GetSimulationAreaCenter()
    {
        return this.WallCenter;
    }

    // Returns the size of the box in the simulation area
    public Vector3 GetSimulationAreaSize()
    {
        return this.WallSize;
    }
    #endregion

    #region MonoBehaviour Functions
    void Start()
    {
        // Initialize buffer
        InitBuffer();
    }

    void Update()
    {
        // just calls run simulation
        Simulation();
    }

    void OnDestroy()
    {
        // Discard buffer
        ReleaseBuffer();
    }

    void OnDrawGizmos()
    {
        // Drawing the simulation area in wireframe for debugging
        Gizmos.color = Color.cyan;
        Gizmos.DrawWireCube(WallCenter, WallSize);
    }
    #endregion

    #region Private Functions
    // Initialize buffer
    void InitBuffer()
    {
        // Initialize buffer
        _boidDataBuffer  = new ComputeBuffer(MaxObjectNum,
            Marshal.SizeOf(typeof(BoidData)));
        _boidForceBuffer = new ComputeBuffer(MaxObjectNum,
```

```
            Marshal.SizeOf(typeof(Vector3)));

        // Boid Initialize data, Force buffer
        var forceArr = new Vector3[MaxObjectNum];
        var boidDataArr = new BoidData[MaxObjectNum];
        for (var i = 0; i < MaxObjectNum; i++)
        {
            forceArr[i] = Vector3.zero;
            boidDataArr[i].Position = Random.insideUnitSphere * 1.0f;
            boidDataArr[i].Velocity = Random.insideUnitSphere * 0.1f;
        }
        _boidForceBuffer.SetData(forceArr);
        _boidDataBuffer.SetData(boidDataArr);
        forceArr    = null;
        boidDataArr = null;
    }

    // simulation
    void Simulation()
    {
        ComputeShader cs = BoidsCS;
        int id = -1;

        // Find the number of thread groups
        int threadGroupSize = Mathf.CeilToInt(MaxObjectNum
            / SIMULATION_BLOCK_SIZE);

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

        // Calculate speed and position from steering force
        id = cs.FindKernel("IntegrateCS"); // カーネル ID を取得
        cs.SetFloat("_DeltaTime", Time.deltaTime);
        cs.SetBuffer(id, "_BoidForceBufferRead", _boidForceBuffer);
        cs.SetBuffer(id, "_BoidDataBufferWrite", _boidDataBuffer);
        cs.Dispatch(id, threadGroupSize, 1, 1); // ComputeShader を実行
    }

    // Free buffer
    void ReleaseBuffer()
```

```
    {
        if (_boidDataBuffer != null)
        {
            _boidDataBuffer.Release();
            _boidDataBuffer = null;
        }

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

■ComputeBuffer initialization The InitBuffer function declares the buffer used when performing calculations on the GPU. We will use a class called ComputeBuffer as a buffer to store data for calculation on the GPU. ComputeBuffer is a data buffer that stores data for ComputeShader. You can read and write to the memory buffer on GPU from C# script. Pass the number of elements in the buffer and the size (number of bytes) of each element in the arguments at initialization. You can get the size (number of bytes) of a type by using the Marshal.SizeOf() method. In ComputeBuffer, you can use SetData() to set the array value of an arbitrary structure.

■Execution of the function described in ComputeShader In the Simulation function, pass the necessary parameters to ComputeShader and issue the calculation instruction.

The function described in ComputeShader that actually causes the GPU to perform calculations is called the kernel. The execution unit of this kernel is called a thread, and in order to perform parallel calculation processing that conforms to the GPU architecture, any number of them are treated as a group, and they are called a thread group. Set the product of the number of threads and the number of thread groups to be equal to or greater than the number of Boid object individuals.

The kernel is specified in the ComputeShader script using the #pragma kernel directive. Each ID is assigned to this, and this ID can be obtained from the C# script by using the FindKernel method.

Use the SetFloat method, SetVector method, SetBuffer method, etc. to pass parameters and buffers required for simulation to ComputeShader. You need the kernel ID when setting buffers and textures.

By executing the Dispatch method, a command is issued so that the kernel defined in ComputeShader will be calculated by the GPU. In the argument, specify the kernel ID and the number of thread groups.

3.4.2 Boids.compute

Write the calculation instruction to the GPU. There are two kernels, one to calculate the steering force and the other to apply that force and update the speed and position.

▼ Boids.compute

```
// Specify kernel function
#pragma kernel ForceCS      // 操舵力を計算
#pragma kernel IntegrateCS  // 速度，位置を計算

// Boid data structure
struct BoidData
{
    float3 velocity; // 速度
    float3 position; // 位置
};

// Thread group thread size
#define SIMULATION_BLOCK_SIZE 256

// Boid data buffer (for reading)
StructuredBuffer<BoidData>   _BoidDataBufferRead;
// Boid data buffer (for reading and writing)
RWStructuredBuffer<BoidData> _BoidDataBufferWrite;
// Boid steering force buffer (for reading)
StructuredBuffer<float3>     _BoidForceBufferRead;
// Boid steering force buffer (for reading and writing)
RWStructuredBuffer<float3>   _BoidForceBufferWrite;

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

float _DeltaTime;      // Time elapsed from the previous frame

float _SeparateNeighborhoodRadius;  // Distance to other individuals to which the separation
float _AlignmentNeighborhoodRadius; // Distance to other individuals to which the alignment a
float _CohesionNeighborhoodRadius;  // Distance from other individuals to which the bond is a

float _MaxSpeed;        // Maximum speed
float _MaxSteerForce;   // Maximum steering force

float _SeparateWeight;  // Weight when applying separation
float _AlignmentWeight; // Weight when applying alignment
float _CohesionWeight;  // Weight when applying join

float4 _WallCenter;      // Center coordinates of wall
float4 _WallSize;        // Wall size
float  _AvoidWallWeight; // Weight of strength to avoid walls
```

```
// Limit vector magnitude
float3 limit(float3 vec, float max)
{
    float length = sqrt(dot(vec, vec)); // 大きさ
    return (length > max && length > 0) ? vec.xyz * (max / length) : vec.xyz;
}

// Returns the opposite force when it hits the wall
float3 avoidWall(float3 position)
{
    float3 wc = _WallCenter.xyz;
    float3 ws = _WallSize.xyz;
    float3 acc = float3(0, 0, 0);
    // x
    acc.x = (position.x < wc.x - ws.x * 0.5) ? acc.x + 1.0 : acc.x;
    acc.x = (position.x > wc.x + ws.x * 0.5) ? acc.x - 1.0 : acc.x;

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

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

    return acc;
}

// Shared memory Boid data storage
groupshared BoidData boid_data[SIMULATION_BLOCK_SIZE];

// Steering force calculation kernel function
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void ForceCS
(
    uint3 DTid : SV_DispatchThreadID, // Unique ID for the entire thread
    uint3 Gid : SV_GroupID, // Group ID
    uint3 GTid : SV_GroupThreadID, // Thread ID within group
    uint  GI : SV_GroupIndex // One-dimensional SV_GroupThreadID 0-255
)
{
    const unsigned int P_ID = DTid.x; // Own ID
    float3 P_position = _BoidDataBufferRead[P_ID].position; // Own position
    float3 P_velocity = _BoidDataBufferRead[P_ID].velocity; // Own speed

    float3 force = float3(0, 0, 0); // Initialize steering force

    float3 sepPosSum = float3(0, 0, 0); // Position addition variable for separation calcula
    int sepCount = 0; // Variable for counting the number of other individuals calculated fo

    float3 aliVelSum = float3(0, 0, 0); // Speed ☒                    ☒
addition variable for alignment calculation
    int aliCount = 0; // A variable for counting the number of other individuals calculated

    float3 cohPosSum = float3(0, 0, 0); // Position addition variable for join calculation
    int cohCount = 0; // A variable for counting the number of other individuals calculated
```

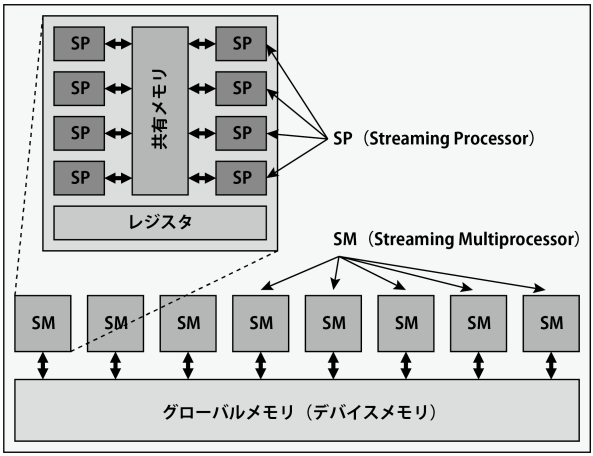
```
// SIMULATION_BLOCK_SIZE(Execution for each group thread number) (Run for the number of g
[loop]
for (uint N_block_ID = 0; N_block_ID < (uint)_MaxBoidObjectNum;
    N_block_ID += SIMULATION_BLOCK_SIZE)
{
    // Boid data for SIMULATION_BLOCK_SIZE is stored in shared memory
    boid_data[GI] = _BoidDataBufferRead[N_block_ID + GI];

    // All group share access is complete,
    // Until all threads in the group reach this call
    // Block execution of all threads in group
    GroupMemoryBarrierWithGroupSync();

    // Calculation with other individuals
    for (int N_tile_ID = 0; N_tile_ID < SIMULATION_BLOCK_SIZE;
        N_tile_ID++)
    {
        // Location of other individuals
        float3 N_position = boid_data[N_tile_ID].position;
        // The speed of other individuals
        float3 N_velocity = boid_data[N_tile_ID].velocity;

        // Difference in position between yourself and other individuals
        float3 diff = P_position - N_position;
        // Distance between yourself and other individuals
        float  dist = sqrt(dot(diff, diff));

        // --- Separate (Separation)  ---
        if (dist > 0.0 && dist <= _SeparateNeighborhoodRadius)
        {
            // Vector from other individuals' positions to themselves
            float3 repulse = normalize(P_position - N_position);
            // Divide by the distance between itself and the position of another individu
            repulse /= dist;
            sepPosSum += repulse; // Addition
            sepCount++;           // Population count
        }

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

        // --- Combine (Cohesion)  ---
        if (dist > 0.0 && dist <= _CohesionNeighborhoodRadius)
        {
            cohPosSum += N_position; // Addition
            cohCount++;              // Population count
        }
    }
    GroupMemoryBarrierWithGroupSync();
}

// Steering force (separation)
float3 sepSteer = (float3)0.0;
if (sepCount > 0)
```

```
    {
        sepSteer = sepPosSum / (float)sepCount;     // Find the average
        sepSteer = normalize(sepSteer) * _MaxSpeed; // Adjust to maximum speed
        sepSteer = sepSteer - P_velocity;           // Calculate steering force
        sepSteer = limit(sepSteer, _MaxSteerForce); // Limit steering force
    }

    // 操舵力（整列）
    float3 aliSteer = (float3)0.0;
    if (aliCount > 0)
    {
        aliSteer = aliVelSum / (float)aliCount; // Find the average velocity of close indivi
        aliSteer = normalize(aliSteer) * _MaxSpeed; // Adjust to maximum speed
        aliSteer = aliSteer - P_velocity;           // Calculate steering force
        aliSteer = limit(aliSteer, _MaxSteerForce); // Limit steering force
    }
    // Steering power (combined)
    float3 cohSteer = (float3)0.0;
    if (cohCount > 0)
    {
        // / Find the average of the positions of close individuals
        cohPosSum = cohPosSum / (float)cohCount;
        cohSteer = cohPosSum - P_position; // Find vector toward average position
        cohSteer = normalize(cohSteer) * _MaxSpeed; // Adjust to maximum speed
        cohSteer = cohSteer - P_velocity;           // Calculate steering force
        cohSteer = limit(cohSteer, _MaxSteerForce); // Limit steering force
    }
    force += aliSteer * _AlignmentWeight; // Add force to align with steering force
    force += cohSteer * _CohesionWeight;  // Add force to the steering force
    force += sepSteer * _SeparateWeight;  // Add a separating force to the steering force

    _BoidForceBufferWrite[P_ID] = force; // writing
}

// Kernel function for velocity and position calculation
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void IntegrateCS
(
    uint3 DTid : SV_DispatchThreadID // Unique ID for the entire thread
)
{
    const unsigned int P_ID = DTid.x; // Get index

    BoidData b = _BoidDataBufferWrite[P_ID]; // Read current Boid data
    float3 force = _BoidForceBufferRead[P_ID]; // Read steering force

    // Gives power to repel when approaching a wall
    force += avoidWall(b.position) * _AvoidWallWeight;

    b.velocity += force * _DeltaTime; // Apply steering force to speed
    b.velocity = limit(b.velocity, _MaxSpeed); // Speed □□ limit
    b.position += b.velocity * _DeltaTime; // Update position

    _BoidDataBufferWrite[P_ID] = b; // Write the calculation result
}
```

Steering force calculation

The ForceCS kernel calculates steering force.

■Utilization of shared memory Variables with the storage modifier groupshared will now be written to shared memory. Although the shared memory cannot write a large amount of data, it is located close to the registers and can be accessed very fast. This shared memory can be shared within a thread group. SIMULATION_BLOCK_SIZE's worth of information about other individuals can be collectively written to the shared memory so that they can be read at high speed within the same thread group, making efficient calculation considering the positional relationship with other individuals. I will go on a regular basis.



▲図 3.3　GPU basic architecture

GroupMemoryBarrierWithGroupSync() When accessing the data written in the shared memory, it is necessary to write the GroupMemoryBarrierWithGroupSync() method to synchronize the processing of all threads in the thread group. GroupMemoryBarrierWithGroupSync() blocks execution of all threads in the thread group until all threads in the group reach this call. This will ensure that all threads in the thread group have properly initialized the boid_data array.

■Calculate steering force by distance from other individuals

Separation If there is an individual closer than the specified distance, the vector from that individual's position to its own position is calculated and normalized. By dividing the vector by the distance value, it is weighted so that it is more avoided when it is close and smaller when it is far, and it is added as a force to prevent collision with other individuals. When the calculation with all the individuals is completed, use that value to calculate the steering force from the relationship with the current speed.

Alignment If there is an individual closer than the specified distance, the velocity (Velocity) of that individual is added together, and at the same time, the number of individuals is counted, and with those values, the velocity of the closer individual (that is, the direction in which it is facing) Find the average of. When the calculation with all the individuals is completed, use that value to calculate the steering force from the relationship with the current speed.

Cohesion If there is an individual closer than the specified distance, the position of that individual is added, and at the same time, the number of that individual is counted, and the average (center of gravity) of the positions of close individuals is calculated from these values. In addition, the vector going to it is calculated, and the steering force is calculated from the relationship with the current speed.

■Update velocity and position of individual Boids The IntegrateCS kernel updates the speed and position of the Boid based on the steering force obtained by ForceCS(). AvoidWall tries to stay outside the specified area by applying a reverse force when trying to get out of the specified area.

3.4.3 BoidsRender.cs

In this script, the result obtained by the Boids simulation is drawn with the specified mesh.

▼ BoidsRender.cs

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

// Guaranteed that the GPU Boids component is attached to the same GameObject
[RequireComponent(typeof(GPUBoids))]
public class BoidsRender : MonoBehaviour
{
    #region Paremeters
    // Scale of Boids object to draw
    public Vector3 ObjectScale = new Vector3(0.1f, 0.2f, 0.5f);
```

```
#endregion

#region Script References
// GPUBoids script reference
public GPUBoids GPUBoidsScript;
#endregion

#region Built-in Resources
// Reference to the mesh to draw
public Mesh InstanceMesh;
// Material reference for drawing
public Material InstanceRenderMaterial;
#endregion

#region Private Variables
// Argument for GPU instancing (for transfer to ComputeBuffer)
// Number of indexes per instance, number of instances,
// Start index position, base vertex position, instance start position
uint[] args = new uint[5] { 0, 0, 0, 0, 0 };
// Argument buffer for GPU instancing
ComputeBuffer argsBuffer;
#endregion

#region MonoBehaviour Functions
void Start ()
{
    // Initialize the argument buffer
    argsBuffer = new ComputeBuffer(1, args.Length * sizeof(uint),
        ComputeBufferType.IndirectArguments);
}

void Update ()
{
    // Instantiating mesh
    RenderInstancedMesh();
}

void OnDisable()
{
    // Free argument buffer
    if (argsBuffer != null)
        argsBuffer.Release();
    argsBuffer = null;
}
#endregion

#region Private Functions
void RenderInstancedMesh()
{
    // The drawing material is Null, or the GPUBoids script is Null,
    // Or if GPU instancing is not supported, do nothing
    if (InstanceRenderMaterial == null || GPUBoidsScript == null ||
        !SystemInfo.supportsInstancing)
        return;

    // Get the index number of the specified mesh
    uint numIndices = (InstanceMesh != null) ?
        (uint)InstanceMesh.GetIndexCount(0) : 0;
```

```
        // Set the number of mesh indexes
        args[0] = numIndices;
        // Set the number of instances
        args[1] = (uint)GPUBoidsScript.GetMaxObjectNum();
        argsBuffer.SetData(args); // Set in buffer

        // Set the buffer that stores the Boid data in the material
        InstanceRenderMaterial.SetBuffer("_BoidDataBuffer",
            GPUBoidsScript.GetBoidDataBuffer());
        // Boid object scale set
        InstanceRenderMaterial.SetVector("_ObjectScale", ObjectScale);
        // Bounding area defined
        var bounds = new Bounds
        (
            GPUBoidsScript.GetSimulationAreaCenter(), // 中心
            GPUBoidsScript.GetSimulationAreaSize()    // サイズ
        );
        // GPU instancing and drawing mesh
        Graphics.DrawMeshInstancedIndirect
        (
            InstanceMesh,           // The mesh to instantiate
            0,                      // submesh index
            InstanceRenderMaterial, // Material to draw
            bounds,                 // Realm
            argsBuffer              // Argument buffer for GPU instancing
        );
    }
    #endregion
}
```

GPU instancing

If you want to draw a large number of identical meshes, creating a GameObject one by one will increase the draw call and the drawing load will increase. In addition, the cost of transferring the calculation result of ComputeShader to CPU memory is high, and if you want to perform high-speed processing, it is necessary to pass the calculation result of GPU to the shader for drawing as it is and perform drawing processing. If you use Unity's GPU instancing, you can draw a large number of same meshes at high speed with few draw calls without generating unnecessary GameObjects.

Graphics.DrawMeshInstancedIndirect() This script uses the Graphics.DrawMeshInstancedIndirect method to draw the mesh by GPU instancing. In this method, you can pass the number of mesh indexes and the number of instances as ComputeBuffer. This is useful if you want to read all instance data from the GPU.

In Start(), the argument buffer for this GPU instancing is initialized. Specify **ComputeBufferType.IndirectArguments** as the third argument of the

constructor at initialization.

RenderInstancedMesh() executes mesh drawing by GPU instancing. Boid data (velocity, position array) obtained by the Boids simulation is passed to the material for rendering InstanceRenderMaterial by the SetBuffer method.

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

This method should normally be called within Update().

3.4.4 BoidsRender.shader

A shader for drawing corresponding to the Graphics.DrawMeshInstancedIndrect method.

▼ BoidsRender.shader

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

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

        struct Input
        {
            float2 uv_MainTex;
        };
        // Boid structure
        struct BoidData
        {
            float3 velocity; // 速度
            float3 position; // 位置
        };

        #ifdef UNITY_PROCEDURAL_INSTANCING_ENABLED
        // Boid data structure buffer
        StructuredBuffer<BoidData> _BoidDataBuffer;
        #endif

        sampler2D _MainTex; // texture
```

```
        half   _Glossiness; // Gloss
        half   _Metallic;   // Metal properties
        fixed4 _Color;      // Color

        float3 _ObjectScale; // Boid Object scale

        // Convert Euler angles (radians) to rotation matrix
        float4x4 eulerAnglesToRotationMatrix(float3 angles)
        {
            float ch = cos(angles.y); float sh = sin(angles.y); // heading
            float ca = cos(angles.z); float sa = sin(angles.z); // attitude
            float cb = cos(angles.x); float sb = sin(angles.x); // bank

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

        // Vertex shader
        void vert(inout appdata_full v)
        {
            #ifdef UNITY_PROCEDURAL_INSTANCING_ENABLED

            // Get Boid data from instance ID
            BoidData boidData = _BoidDataBuffer[unity_InstanceID];

            float3 pos = boidData.position.xyz; // Get Boid position
            float3 scl = _ObjectScale;          // Get Boid Scale

            // Define a matrix to transform from object coordinates to world coordinates
            float4x4 object2world = (float4x4)0;
            // Substitute scale value
            object2world._11_22_33_44 = float4(scl.xyz, 1.0);
            // Calculate rotation about Y axis from speed
            float rotY =
                atan2(boidData.velocity.x, boidData.velocity.z);
            // Calculate rotation about X axis from speed
            float rotX =
                -asin(boidData.velocity.y / (length(boidData.velocity.xyz)
                + 1e-8)); // 0 division prevention
            // Find rotation matrix from Euler angles (radians)
            float4x4 rotMatrix =
                eulerAnglesToRotationMatrix(float3(rotX, rotY, 0));
            // Apply rotation to matrix
            object2world = mul(rotMatrix, object2world);
            // Apply position (translation) to matrix
            object2world._14_24_34 += pos.xyz;

            // Coordinate conversion of vertices
            v.vertex = mul(object2world, v.vertex);
            // Convert normal to coordinates
            v.normal = normalize(mul(object2world, v.normal));
            #endif
        }
```

```
        void setup()
        {
        }

        // Surface shader
        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

#pragma surface surf Standard vertex:vert addshadow In this part, surf() is specified as the surface shader, Standard is specified as the lighting model, and vert() is specified as the custom vertex shader.

You can tell Unity to generate an additional variant for when you use the Graphics.DrawMeshInstancedIndirect method by writing procedural:FunctionName in the #pragma instancing_options directive, and specify it in FunctionName at the beginning of the vertex shader stage. The called function will be called. Official sample (https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html) Looking at etc., in this function, unity_ObjectToWorld matrix, unity_WorldToObject matrix is ☒☒ rewritten based on the position, rotation and scale of each instance, but in this sample program, data of Boids is received in the vertex shader, I am converting the coordinates of vertices and normals (I am not sure if it is good ...). Therefore, nothing is written in the specified setup function.

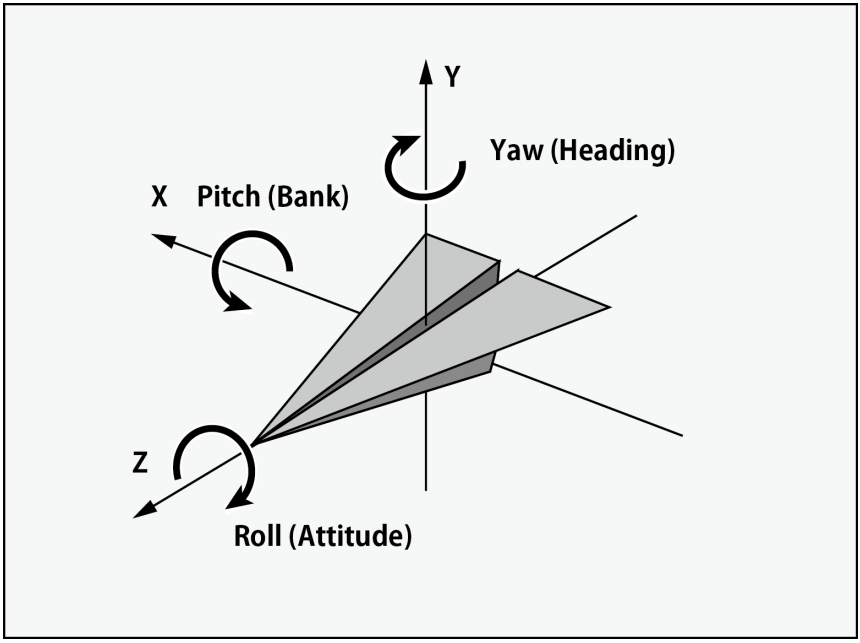
Get the Boid data for each instance with the vertex shader and convert the coordinates

In Vertex Shader, describe the processing to be performed on the vertices of the mesh passed to the shader.

You can get the unique ID for each instance by unity_InstanceID. By specifying this ID as the index of the array of StructuredBuffer that is declared as a buffer of Boid data, you can get Boid data unique to each instance.

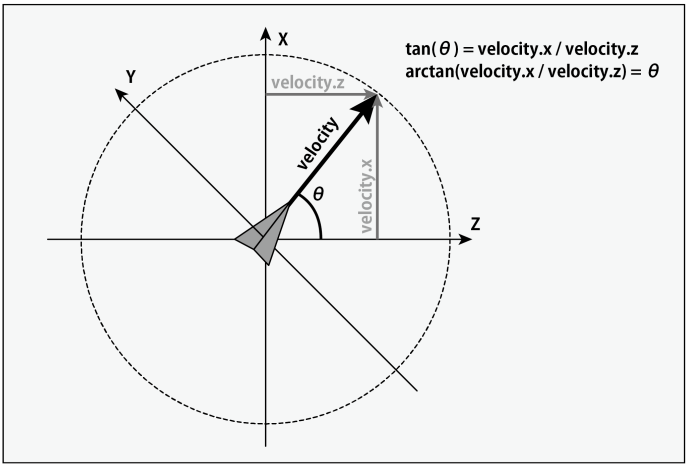
Ask for rotation

From the velocity data of the Boid, calculate the rotation value so as to face the traveling direction. In order to handle it intuitively, rotation is expressed by Euler angles. When the Boid is regarded as a flying object, the rotations of the three axes with respect to the object are called pitch, yaw, and roll, respectively.



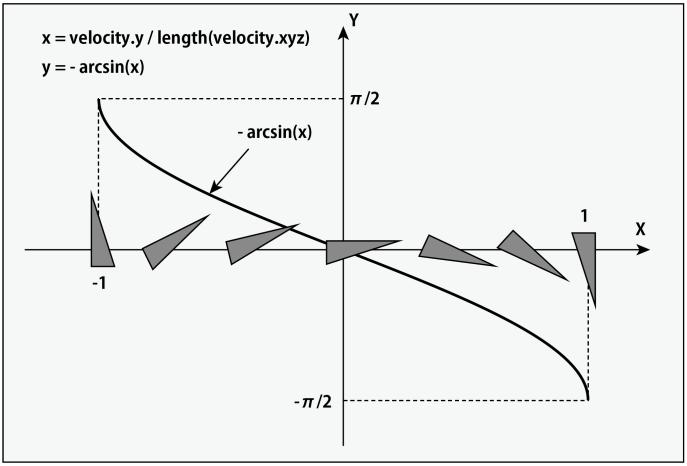
▲図 3.4　Axis and rotation designation

First, from the velocity about the Z-axis and the velocity about the X-axis, yaw (which direction it is facing with respect to the horizontal plane) is obtained using the atan2 method that returns the arctangent.



▲図 3.5　Relationship between speed and angle (yaw)

Next, the asin method that returns the arc sine (arc sine) is used to find the pitch (upward and downward inclination) from the ratio of the speed and the speed about the Y axis. When the Y-axis speed is small among the speeds for each axis, the amount of rotation is weighted so that the Y-axis speed is small and the level is kept horizontal.



▲図 3.6　Relationship between speed and angle (pitch)

Compute matrix applying Boid transform

Coordinate conversion processing such as movement, rotation, and scaling can be expressed together in a single matrix. Define a 4x4 matrix object2world.

■Scale　First, substitute the scale value. For each XYZ axis $S_xS_yS_z$ The matrix S that scales only by is expressed as follows.

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

HLSL float4x4 type variables can specify a particular element of the matrix using swizzles like ._11_22_33_44. By default, the components are arranged as follows.

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

Here, substitute the values ☒☒ of the scale of XYZ for 11, 22, 33, and 1 for 44.

■rotation Then apply rotation. Rotation about each XYZ axis $\mathrm{R_x R_y R_z}$ is expressed as a matrix,

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

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

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

This is combined into a matrix. At this time, the behavior at the time of rotation changes depending on the order of the rotation axis to be combined, but if combined in this order, it should be similar to Unity's standard rotation.

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

▲図 3.7 Composition of rotation matrix

Apply rotation by finding the product of the resulting rotation matrix and the scaled matrix above.

■Translation Then apply the translation. On each axis, $T_xT_yT_z$ When translated, the matrix is ☒☒ represented as

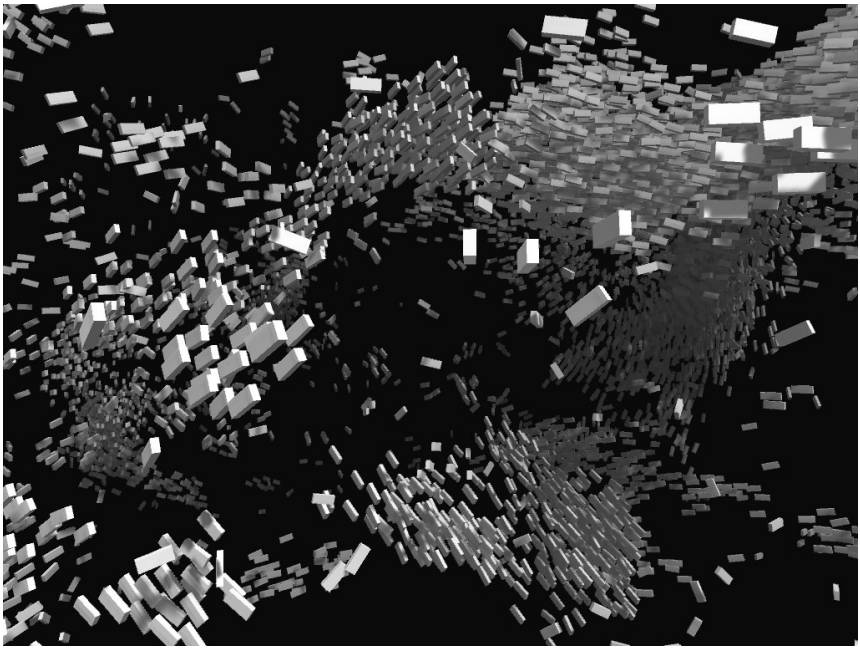
$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This translation can be applied by adding the position data for the XYZ axes to the 14, 24 and 34 components.

By applying the matrix obtained by these calculations to the vertices and normals, the Boid transform data is reflected.

3.4.5 Drawing result

I think that an object that moves like a flock like this is drawn.



▲図 3.8 Execution result

3.5 Summary

The implementation introduced in this chapter uses the minimum Boids algorithm, but it also has different characteristics such as a large group or several small colonies even if the parameters are adjusted. I think it will show a movement. In addition to the basic rules of behavior shown here, there are other rules to consider. For example, if this is a school of fish, it will naturally escape if a foreign enemy preying on them appears, and if there are obstacles such as terrain, the fish will avoid hitting it. In terms of vision, the field of view and accuracy differ depending on the species of animal, and I think that if you exclude other individuals outside the field of view from the calculation process, you will get closer to the actual one. The characteristics of movement also change depending on the environment such as whether you fly in the sky, move in water, or move on land, and the characteristics of the motor organs for locomotion. You should also pay attention to individual differences.

Parallel processing by GPU can calculate many individuals compared to the calculation by CPU, but basically, it does brute force calculation with other individuals, so it cannot be said that the calculation efficiency is very good. To do this, the cost of computation is increased by increasing the efficiency of the neighboring individual search, such as registering individuals in regions divided by grids or blocks according to their positions and performing calculation processing only for the individuals that exist in adjacent regions. Can be suppressed.

There is plenty of room for improvement in this way, and by applying proper implementation and behavior rules, it is possible to express more beautiful, powerful, dense and tasting group movements. I want to be able to do it.

3.6 reference

- Boids Background and Update - https://www.red3d.com/cwr/boids/
- THE NATURE OF CODE - http://natureofcode.com/
- Real-Time Particle Systems on the GPU in Dynamic Environments - http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/02/Chapter7-Drone-Real-Time**Particle**Systems**On**The_GPU.pdf
- Practical Rendering and Computation with Direct3D 11 - https://dl.acm.org/citation.cfm?id=2050039

- GPU 並列図形処理入門 - http://gihyo.jp/book/2014/978-4-7741-6304-8

第 4 章

Fluid simulation by grid method

4.1 About this chapter

In this chapter, we will explain the fluid simulation by the grid method using Compute Shader.

4.2 Sample data

4.2.1 code

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

4.2.2 Execution environment

- Shader model 5.0 compatible environment that ComputeShader can execute
- Environment confirmed at the time of writing, Unity5.6.2, Unity2017.1.1

4.3 Introduction

In this chapter, we will explain the fluid simulation by the grid method and the calculation method and the way of understanding the mathematical formulas that are necessary to realize them. First of all, what is the lattice method? In order to find out its meaning, let's take a closer look at the method of analyzing

“flow” in hydrodynamics.

4.3.1 How to think in fluid mechanics

Fluid mechanics is characterized by formulating a natural phenomenon, "flow", and making it computable. How can this "flow" be digitized and analyzed?
If you go straight, it can be quantified by guiding "the flow velocity when the time advances momentarily". Mathematically speaking, it can be translated into the analysis of the amount of change in the velocity vector when differentiated with respect to time.
However, there are two possible methods to analyze this flow.
One is to divide the hot water in the bath into grids and measure the flow velocity vector of each fixed grid space when you imagine the hot water in the bath.
And the other is to float the duck in the bath and analyze the duck movement itself. Of these two methods, the former is called "Euler's method" and the latter is called "Lagrange's method".

4.3.2 Various fluid simulations

Now let's get back to computer graphics. There are several simulation methods for fluid simulation, such as "Euler's method" and "Lagrange's method", but they can be broadly divided into the following three types.

- Lattice method (e.g. Stable Fluid)
- Particle method (e.g. SPH)
- Lattice method + particle method (e.g. FLIP)

It may be a little imaginable from the meaning of the kanji, but the grid method, like the Euler method, creates a grid-shaped "field" when simulating a flow, and differentiates it by time. It is a method of simulating the speed of each grid.
The particle method is a method of simulating the advection of the particle itself, focusing on the particle, as in the "Lagrange method".
Along with the lattice method and the particle method, there are areas where we are good and bad at each other.
The lattice method is good at calculating pressure, viscosity, diffusion, etc. in fluid simulation, but not good at advection calculation.
On the contrary, the particle method is good at calculating advection. (These strengths and weaknesses can be imagined when you think of how to analyze Euler's method and Lagrange's method.)

In order to supplement these, methods such as the lattice method + particle method, which are typified by the FLIP method, have been born to complement each other's specialty fields.

In this paper, based on Jon Stam's Stable Fluids, which is a paper on incompressible viscous fluid simulation in the lattice method presented at SIGGRAPH 1999., we will explain the implementation method of fluid simulation and the mathematical formulas required for simulation. ..

4.4 On the Navier-Stokes equation

First, let's look at the Navier-Stokes equation in the lattice method.

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

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

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

Of the above, the first equation represents the velocity field and the second the density field. The third is "Continuity formula (mass conservation law)". Let's try unraveling these three expressions one by one.

4.5 Formula of continuity (mass conservation law)

First, let's unravel from the "continuous equation (mass conservation law)", which is a short equation and serves as a condition when simulating an "incompressible" fluid.

When simulating a fluid, you need to make a clear distinction between what is compressible and what is incompressible. For example, if the target is a substance whose density such as gas changes with pressure, it will be a compressible fluid. On the other hand, if the density of water is constant at any place, it is an incompressible fluid.

Since this chapter deals with the simulation of incompressible fluids, the divergence of each cell in the velocity field should be kept at zero. In other words, it cancels the inflow and outflow of the velocity field and keeps it at 0. If there is an inflow, it will flow out, so the flow velocity will propagate. This condition can be

expressed by the following equation as a continuous equation (mass conservation law).

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

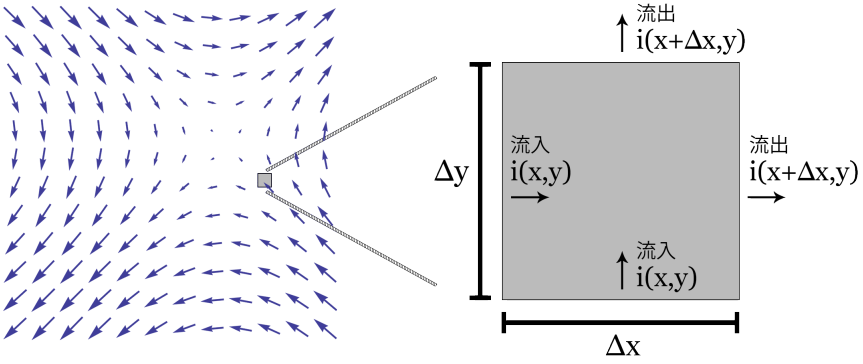
The above means that "divergence is 0". First, let's check the formula for "divergence".

4.5.1 Divergence

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

∇ (Nabla operator) is called vector differential operator. For example, assuming that the vector field is two-dimensional, $\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$ の偏微分を取る際の、偏微分の表記を簡略化した演算子として作用します。 ∇ 演算子は演算子ですので、それだけでは意味を持ちませんが、一緒に組み合わせる式が内積なのか、外積なのか、それとも単に ∇f といった関数なのかで演算内容が変わってきます。
This time, let's talk about "divergence," which is the inner product of partial derivatives. First, let's see why this expression means "divergence".

In order to understand the divergence, let's first consider cutting out one cell in the lattice space as shown below.



▲図 4.1 Extract cells of differential interval (Δ x, Δ y) from vector field

Divergence is the calculation of how many vectors flow into and out of one cell in the vector field. Outflow is + and inflow is -.

The divergence is the amount of change between a specific point x in the x direction and a slight amount of Δx when looking at the partial derivative when the cell of the vector field is cut off as described above, or , It can be calculated by the inner product of the change amount of the specific point y in the y direction and the slightly advanced Δy . The reason why the outflow is obtained by the inner product with the partial derivative can be proved by differentiating the above figure.

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

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

上記の式から極限をとり、

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

By doing, you can finally find the equation and the equation of the inner product with the partial derivative.

4.6 Velocity field

Next, I will explain the velocity field, which is the main point of the lattice method. Before that, let's confirm the gradient and Laplacian in addition to the divergence confirmed earlier when implementing the Navier-Stokes equation of the velocity field.

4.6.1 Gradient

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

$\nabla f(grad\ f)$ Is the formula for the gradient. The meaning is that by sampling the coordinates slightly advanced in each partial differential direction with the function f and synthesizing the obtained values ☒☒ in each partial differential direction, which vector is finally determined. It means facing. In other words, it

is possible to calculate the vector that is oriented in the direction with the larger value when the partial differentiation is performed.

4.6.2 Laplacian

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

Laplacian is represented by the symbol of Nabla inverted upside down. (Same as Delta, but read from the context, don't make a mistake.)
$\nabla^2 f$ Or $\nabla \cdot \nabla f$ It is also written as and is calculated as the second partial derivative.
Also, when it is disassembled and considered, it can be taken as a form in which the gradient of the function is taken and the divergence is obtained.
In terms of meaning, there are many inflows at the points concentrated in the gradient direction in the vector field, so there is a divergence. I can imagine that.
There are two types of Laplacian operators, scalar Laplacian and vector Laplacian. When acting on a vector field, gradient, divergence, and rotation (the outer product of ∇ and the vector) are used.

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

However, only in the case of Cartesian coordinate system, the gradient and divergence can be obtained for each vector component, and can be obtained by combining them.

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

This completes the confirmation of the mathematical formulas required to solve the Navier-Stokes equations in the lattice method. From here, let's look at the velocity field equation for each term.

4.6.3 Confirmation of velocity field from Navier-Stokes equation

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

Of the above, $\vec{u}$ Is the flow velocity, ν Is the kinematic viscosity coefficient (kinematic viscosity)、 $\vec{f}$ Is an external force.
You can see that the left side is the flow velocity when partial differentiation is taken with time. On the right side, the first term is the advection term, the second term is the diffusion viscosity term, the third term is the pressure term, and the fourth term is the external force term.

Even if these can be done in a batch at the time of calculation, it is necessary to implement them in steps when implementing them.
First of all, as a step, if you do not receive an external force, you cannot make changes under the initial conditions, so I would like to consider from the external force term of the fourth term.

4.6.4 Force item outside the velocity field

This is the part that simply adds the vector from the outside. In other words, when the velocity field is 0 in the initial condition, the vector is added to the corresponding ID of RWTexture2D from the UI or some event as the starting point of the vector.
The kernel of the external force term of compute shader is implemented as follows. Also, describe the definition of each coefficient and buffer that will be used in the compute shader.

```
float visc;                   //Kinematic viscosity coefficient
float dt;                     //Delta time
float velocityCoef;           //Velocity external force coefficient
float densityCoef;            //Pressure coefficient outside density field

//xy = velocity, z = density, Fluid solver to pass to the drawing shader
RWTexture2D<float4> solver;
//density field, Density field
RWTexture2D<float>  density;
//velocity field, Velocity field
RWTexture2D<float2> velocity;
//xy = pre vel, z = pre dens. when project, x = p, y = div
//Save buffer one step before and temporary buffer when saving mass
RWTexture2D<float3> prev;
//xy = velocity source, z = density source 外力入力バッファ
Texture2D source;

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

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

The next step is to implement the second term, the diffusive viscosity term.

4.6.5 Velocity field

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

∇ 演算子や Δ When there is a value on the left and right of the operator, there is a rule that "acts only on the right element", so in this case, leave the kinematic viscosity coefficient once and consider the vector Laplacian part first.
The vector Laplacian is used for the flow velocity $\vec{u}$ to synthesize the gradient and divergence of each component of the vector, and diffuse the flow velocity to the adjacent. By multiplying it by the kinematic viscosity coefficient, the diffusion momentum is adjusted.
Here, since the gradient of each component of the flow velocity is taken and then diffused, it may be possible to understand the phenomenon that inflow from and outflow from adjacent neighbors occur, and the vector received in step 1 affects adjacent neighbors. think.
On the mounting side, some ingenuity is required. If implemented according to the formula, vibration will occur when the diffusivity obtained by multiplying the viscosity coefficient by the differential time and the number of grids becomes high, and convergence will not be achieved and the simulation itself will eventually diverge.
In order to make the diffusion stable, the iterative methods such as Gauss-Seidel method, Jacobi method and SOR method are used here. Here, let's simulate the Gauss-Seidel method.
The Gauss-Seidel method is a method in which an equation is converted into a linear equation consisting of unknowns for its own cell, the calculated value is immediately used in the next iteration, and a chain is made to converge to an approximate answer. More iterations will converge to more accurate values, but the number of iterations is machine Adjust it considering performance and appearance.

```
#define GS_ITERATE 4

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

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

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

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

4.6.6 Quality preservation

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

Now let's go back to the mass storage side before proceeding with the section. In the process up to this point, the force received by the external force term was diffused into the velocity field, but at present, the mass of each cell is not preserved, and the mass of the place where there is a lot of inflow and the place where there are many inflows Is not saved.
As in the above equation, the mass must be saved and the divergence of each cell must be set to 0, so save the mass here.
In addition, when performing the mass preservation step with ComputeShader, the field must be fixed because it performs a partial differential operation with the adjacent thread. If partial differential operation could be performed in the group shared memory, speedup could be expected, but when partial differential was taken from another group thread, the value could not be obtained and the result was dirty, so buffer here. While confirming, proceed in 3 steps.
Calculate divergence from velocity field> Calculate Poisson equation by Gauss-

Seidel method> Subtract velocity field and save mass
Divide the kernel into the three steps of, and bring it to the mass conservation while establishing the field. The SetBound~ system is a method call for the boundary.

```
//Quality preservation Step1.
//In step 1, calculate the divergence from the velocity field
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep1(uint2 id : SV_DispatchThreadID)
{
    uint w, h;
    velocity.GetDimensions(w, h);

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

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

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

//Quality preservation Step 2.
//In step2, the Poisson equation is solved by the Gauss-Seidel method from the divergence ob
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep2(uint2 id : SV_DispatchThreadID)
{
    uint w, h;

    velocity.GetDimensions(w, h);

    if (id.x < w && id.y < h)
    {
        for (int k = 0; k < GS_ITERATE; k++)
        {
            prev[id] = float3(
                        (prev[id].y + prev[uint2(id.x - 1, id.y)].x +
                                      prev[uint2(id.x + 1, id.y)].x +
                                      prev[uint2(id.x, id.y - 1)].x +
                                      prev[uint2(id.x, id.y + 1)].x) / 4,
                        prev[id].yz);
            SetBoundaryDivPositive(id, w, h);
        }
    }
}
```

```
//Quality preservation Step 3.
//step3 so、∇・u = 0 To
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void ProjectStep3(uint2 id : SV_DispatchThreadID)
{
    uint w, h;

    velocity.GetDimensions(w, h);

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

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

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

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

With this, the velocity field is in the state of mass conservation. Inflow occurs at the place where it flows out, and outflow occurs at the place where there is a large amount of inflow.

4.6.7 Advection term

$$-(\overrightarrow{u} \cdot \nabla)\overrightarrow{u}$$

For the advection term, Lagrange's method is used, but it is necessary to perform backtracking of the velocity field one step before and move the value of the location where the velocity vector is subtracted from the relevant cell to the current location. Do this for each cell. When backtracing, it does not go back to a place that fits exactly on the grid, so at the time of advection, linear interpolation with the neighboring 4 cells is performed and the correct value is advected.

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

```
    density.GetDimensions(w, h);

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

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

        //Backtrace point calculation.
        x = (float)id.x - dfdt * prev[id].x;
        y = (float)id.y - dfdt * prev[id].y;
        //Clamp the points so that they are within the simulation range.
        clamp(x, 0.5, w + 0.5);
        clamp(y, 0.5, h + 0.5);
        //Near cell index of back trace point.
        ddx0 = floor(x);
        ddx1 = ddx0 + 1;
        ddy0 = floor(y);
        ddy1 = ddy0 + 1;
        //Save the difference for linear interpolation with neighboring cells.
        s1 = x - ddx0;
        s0 = 1.0 - s1;
        t1 = y - ddy0;
        t0 = 1.0 - t1;

        //Back trace, take the value of 1 step before with linear interpolation and substitu
        velocity[id] = s0 * (t0 * prev[int2(ddx0, ddy0)].xy +
                             t1 * prev[int2(ddx0, ddy1)].xy) +
                       s1 * (t0 * prev[int2(ddx1, ddy0)].xy +
                             t1 * prev[int2(ddx1, ddy1)].xy);
        SetBoundaryVelocity(id, w, h);
    }
}
```

4.7 Density field

Next, let's see the density field equation.

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

上記の内、$\overrightarrow{u}$ Is the flow velocity, κ Is the diffusion coefficient, ρ is the density, and S is the external pressure.

The density field is not always necessary, but by adding the pixels on the screen diffused by the density field to each vector when the velocity field is obtained, it becomes possible to express a more fluid-like fluid like melting. I will.

As you may have noticed by looking at the density field formula, the flow is exactly the same as the velocity field, the difference is that the vector is a scalar and the kinematic viscosity coefficient $Thereareonlythreepoints : thepointwhere\nu$ is the

diffusion coefficient κ and the point where the law of conservation of mass is not used.
Since the density field is the field of change in density, it does not need to be incompressible and does not require mass conservation. The kinematic viscosity coefficient and the diffusion coefficient have the same usage as coefficients.
Therefore, it is possible to implement the density field by reducing the dimension of the kernel other than the mass conservation law of the kernel used in the velocity field. I will not explain the density field on paper, but please refer to that as well because the density field is implemented in the repository.

4.8 Simulation term steps

A fluid can be simulated by using the above velocity field, density field, and mass conservation law, but let's look at the simulation steps at the end.

- Generate an external force event and input it into the external force terms of the velocity and density fields
- Update the velocity field in the following steps
  - Loose sticky term
  - Quality preservation
  - Advection term
  - Quality preservation
- Then update the density field in the following steps
  - Loose item
  - Advancing density using velocity in velocity field

The above is the simulation step of StableFluid.

4.9 result

By executing and dragging on the screen with the mouse, the following fluid simulation can be triggered.

▲図 4.2　Execution example

4.10 Summary

Unlike pre-rendering, fluid simulation is a heavy field for real-time game engines like Unity. However, due to improvements in GPU computing power, it has become possible to produce FPS that can withstand a certain level of resolution in two dimensions. In addition, if you try to implement the Gauss-Seidel iterative method, which is a heavy load on the GPU that came out on the way, with another process, or substitute the curl noise for the velocity field itself, It will be possible to express fluids with lighter calculations.

If you have read this chapter and are interested in fluids, please try the next chapter, "Fluid Simulation by Particle Method". Since you can approach the fluid from a different angle from the grid method, I think that you can experience the depth of fluid simulation and the fun of mounting.

4.11 reference

- Jos Stam. SIGGRAPH 1999. Stable Fluids

第 5 章

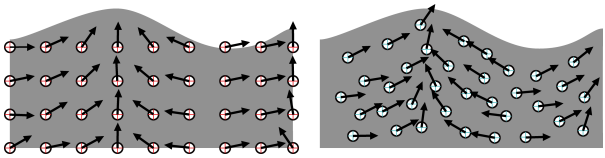
Fluid simulation by SPH method

In the previous chapter, we explained how to create a fluid simulation using the grid method. In this chapter, we will use another particle simulation method, the SPH method in particular, to represent the fluid motion. Please be aware that some explanations may be inadequate, as the explanation is made with a little bit of clutter.

5.1 Basic knowledge

5.1.1 Euler and Lagrangian perspectives

There are Euler's point of view and Lagrange's point of view as a method of observing the movement of a fluid. The Euler's point of view is to fix the observation points at fixed intervals on the fluid **fixed** and analyze the movement of the fluid at the observation points. On the other hand, the Lagrangian point of view is that the observation point that moves along the fluid flow is **floating** and the movement of the fluid at that observation point is observed (see Figure 5.1). .. Basically, the fluid simulation method using the Euler's viewpoint is called the grid method, and the fluid simulation method using the Lagrangian viewpoint is called the particle method.



▲図 5.1　Left: Euler-like, Right: Lagrange-like

5.1.2 Lagrange differentiation (material differentiation)

The Euler and Lagrangian viewpoints differ in the way they are calculated. First, let us show the physical quantity [*1] expressed from the Euler point of view.

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

This is the time t Positioned at $\overrightarrow{x}$ Physical quantity ϕ It means that $\cdots$ The time derivative of this physical quantity is

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

Can be expressed as Of course, this is because the position of the physical quantity $\overrightarrow{x}$ Since it is fixed at, it is the derivative from the Euler's point of view.

On the other hand, from the Lagrangian perspective, the observation point moves along the flow.[*2]Therefore, the observation point itself is also a function of time. Therefore, in the initial state $\overrightarrow{x}_0$ The observation point in t ぐ

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

Exists in Therefore, the notation of physical quantity

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

Will be. According to the definition of derivative, Δt Looking at the amount of change in physical quantity after a second

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

[*1] Physical quantity refers to observable speed and mass. In short, you can think of it as having a unit.

[*2] The movement of the observation point along the flow is called advection.

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

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

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

Will be. This is the time derivative of the physical quantity considering the movement of the observation point. However, using this notation complicates the formula, so

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

It can be shortened by introducing the operator. A series of operations that consider the movement of observation points is called Lagrange differentiation. It may seem complicated at first glance, but in the particle method in which the observation points move, it is more convenient to express the formula from a Lagrangian perspective.

5.1.3 Fluid incompressible condition

A fluid can be considered to have no volume change if the velocity of the fluid is well below the speed of sound. This is called the fluid incompressible condition and is expressed by the following formula.

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

This indicates that there is no upwelling or disappearance in the fluid. Derivation of this formula involves a little complicated integration, so the explanation is omitted.[*3]To do. Please keep in mind about "the fluid does not compress!"

5.2 Particle method simulation

Particle method, small fluid **粒子** Divide by and observe the fluid movement from a Lagrangian perspective. These particles are the observation points in the

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

previous section. Even though the “particle method” is used to describe a lot, many methods have been proposed at the present time, and as a famous one,

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

And so on.

5.2.1 Derivation of Navier-Stokes equations in the particle method

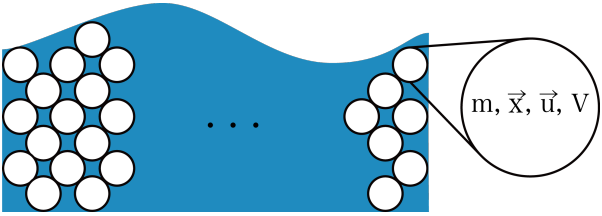
First, the Navier-Stokes equation (hereinafter NS equation) in the particle method is described as follows.

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

The shape is a little different from the NS equation that appeared in the lattice method in the previous chapter. The advection term is completely gone, but if you look at the relationship between the Euler derivative and the Lagrange derivative, you can see that it can be transformed into this shape well. In the particle method, the observation point is moved along the flow, so it is not necessary to consider the advection term when calculating the NS equation. The advection calculation can be done by directly updating the particle position based on the acceleration calculated by the NS equation.

Since a real fluid is a collection of molecules, it can be said that it is a kind of particle system. However, it is impossible to calculate the actual number of molecules with a computer, so it is necessary to adjust it to a size that can be calculated. Figure 5.2Each grain shown in([*4])Represents a portion of the fluid divided into calculable sizes. Each of these grains has a mass m , Position vector $\overrightarrow{x}$ 、 Speed ☒☒ vector $\overrightarrow{u}$,volume V Can be thought of as having.

[*4] Called'Blob' in English



▲図 5.2　Particle approximation of fluid

External force on each of these grains $\overrightarrow{f}$ To calculate the equation of motion $m\overrightarrow{a} = \overrightarrow{f}$ The acceleration is calculated by solving You can decide how to move in the next time step.

As mentioned above, each particle moves by receiving some force from the surroundings. What is that "force"? As a simple example, gravity $m\overrightarrow{g}$ Other than that, some other force should also be exerted by surrounding particles. These forces are explained below.

pressure

The first force on a fluid particle is pressure. The fluid always flows from higher pressure to lower pressure. If the pressure is the same from all directions, the force will be canceled and the movement will stop, so consider the case where the pressure is not balanced. As mentioned in the previous section, by taking the gradient of the pressure scalar field, it is possible to calculate the direction with the highest rate of pressure rise from the viewpoint of one's own particle position. 粒子が力を受ける方向は、圧力の高い方から低い方ですので、マイナスを取って $-\nabla p$ となります。Also, since particles have a volume, the pressure applied to them is $-\nabla p$ It is calculated by multiplying the particle volume by[*5]。Finally, $-V\nabla p$ The result is derived.

Viscous force

The second force on fluid particles is viscous force. A viscous (sticky) fluid is a fluid that is difficult to deform, such as honey and melted chocolate. Applying the word viscous to the expression of the particle method, **The particle velocity is easy to average the surrounding particle velocity.**It means that. As

[*5] The incompressible condition of a fluid makes it possible to express the integral of the pressure exerted on a particle simply by multiplying it by volume.

mentioned in the previous chapter, the operation of taking the average of the surroundings can be done using Laplacian.

The degree of viscosity**Kinematic viscosity coefficient** μ When expressed using $\mu\nabla\cdot\nabla\overrightarrow{u}$ Can be expressed as

Integration of pressure, viscous force, and external force

Equations of motion for these forces $m\overrightarrow{a}=\overrightarrow{f}$ If you apply it to

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

here, m は ρV Since it is(V Will be canceled)

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

Both sides ρ Divide by

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

Finally, the coefficient of the viscosity term $\frac{\mu}{\rho}$ に ν Introduced

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

Then, I was able to derive the NS equation mentioned at the beginning.

5.2.2 Representation of advection in the particle method.

In the particle method, the particles themselves represent the observation points of the fluid, so the calculation of the advection term is completed simply by moving the particle position. In the actual calculation of time derivative, infinitely small time is used, Infinite time cannot be expressed by computer calculation, so the time is small enough. Δt Is used to express the derivative. This is called difference, Δt The smaller is, the more accurate the calculation can be made.

Introducing the difference expression for acceleration,

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

Will be. Therefore speed increment $\Delta\overrightarrow{u}$ Is

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

And for position increments as well,

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

Than,

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

Will be.

You can use this result to calculate the velocity and position vectors for the next frame. The particle velocity at the current frame $\overrightarrow{u}_n$, Then The particle velocity in the next frame is $\overrightarrow{u}_{n+1}$ so,

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

Can be expressed as

The particle position in the current frame $\overrightarrow{x}_n$, Then The particle position in the next frame is $\overrightarrow{x}_{n+1}$ so,

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

Can be expressed as

This method is called the forward Euler method. By repeating this for each frame, the movement of particles at each time can be expressed.

5.3 Fluid simulation by SPH method

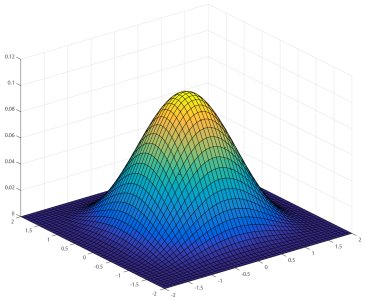
In the previous section, we explained how to derive the NS equation in the particle method. Of course, these differential equations cannot be solved directly by computer, so some kind of approximation is needed. As a method, I will explain **SPH method** which is often used in the CG field.

The SPH method was originally used for the simulation of collisions between celestial bodies in astrophysics, but was also applied to fluid simulation in CG by Desbrun et al. [*6] in 1996. In addition, parallelization is easy, and with the current GPU, it is possible to calculate large numbers of particles in real time. In computer simulation, it is necessary to discretize continuous physical quantities for calculation, This discretization is**Weight function**The method that uses a function called is called the SPH method.

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

5.3.1 Discretization of physical quantities

In the SPH method, each particle has a range of influence, and the closer the distance to other particles, the greater the influence of that particle. When this influence range is illustratedFigure 5.3It looks like.



▲図 5.3　2-D weight function

This function**Weight function**[*7]I call it.

Physical quantity in SPH method ϕ Then, it is discretized using the weight function as follows.

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

N, m, ρ, h Are the collection of neighboring particles, particle mass, particle density, and radius of influence of the weighting function, respectively. Also, the function W is the weighting function mentioned earlier.

Furthermore, partial differential operations such as gradient and Laplacian can be applied to this physical quantity, The gradient is

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

Laplacian is

[*7] Usually, this function is also called a kernel function, but this name is used to distinguish it from the kernel function in ComputeShader.

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

Can be expressed as As you can see from the formula, the gradient of the physical quantity and the Laplacian are images that are applied only to the weighting function. Weight function W Uses different values ☒☒ depending on the physical quantity you want to find, but I will omit the explanation for this reason.[*8]To do.

5.3.2 Discretization of density

The density of fluid particles can be calculated using the equation of physical quantity discretized by the weighting function.

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

Is given. Where the weighting function to use W Is given by

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

▲図 5.4　Poly6 weight function

5.3.3 Discretization of viscosity term

The viscous term is discretized using the weighting function as in the case of density,

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

It is expressed as. Where the Laplacian of the weighting function $\nabla^2 W_{visc}$ Is given by

[*8] "Basics of physical simulation for CG: Makoto Fujisawa".

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

▲図 5.5　Viscosity Laplacian of weighting function

5.3.4 Discretization of pressure term

Similarly, discretize the pressure term.

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

Where the gradient of the weighting function W_{spiky} Is given by

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

▲図 5.6　Spiky weight function gradient

At this time, the particle pressure is called Tait equation in advance,

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

It is calculated by. Where B is the gas constant. In order to guarantee incompressibility, it is necessary to solve the Poisson equation, but it is not suitable for real-time calculation. Instead, SPH method[*9]Then, it is said that the calculation of the pressure term is weaker than the lattice method in terms of securing incompressibility approximately.

[*9] The SPH method that calculates pressure using the Tait equation is called the WCSPH method.

5.4 Implementation of SPH method

Sample here(`https://github.com/IndieVisualLab/UnityGraphicsProgramming`)Assets/SPH Fluid is listed below. Please note that this implementation does not consider speedup or numerical stability in order to explain the SPH method as simply as possible.

5.4.1 The parameter

The comments in the code describe the various parameters used for the simulation.

▼ List 5.1 Parameters used for simulation(FluidBase.cs)

```
NumParticleEnum particleNum = NumParticleEnum.NUM_8K;    // Number of particles
float smoothlen = 0.012f;               // Particle radius
float pressureStiffness = 200.0f;       // Pressure term coefficient
float restDensity = 1000.0f;            // Resting density
float particleMass = 0.0002f;           // Particle mass
float viscosity = 0.1f;                 // Viscosity coefficient
float maxAllowableTimestep = 0.005f;    // Step size
float wallStiffness = 3000.0f;          // Power of Penalty Law Wall
int iterations = 4;                     // Number of iterations
Vector2 gravity = new Vector2(0.0f, -0.5f);     // gravity
Vector2 range = new Vector2(1, 1);              // Simulation space
bool simulate = true;                           // Execute or pause

int numParticles;             // Number of particles
float timeStep;               // Step size
float densityCoef;            // Density coefficient of Poly6 kernel
float gradPressureCoef;       // Pressure coefficient of Spiky kernel
float lapViscosityCoef;       // Laplacian kernel viscosity coefficient
```

Please note that in this demo scene, the inspector is set to a value different from the initialization value of the parameter described in the code.

5.4.2 Compute SPH weight function coefficients

The coefficient of the weight function does not change during the simulation, so it should be calculated on the CPU side at initialization. (However, it is updated in the Update function considering the possibility of editing the parameter during execution)

This time, the mass of each particle is kept constant, so the mass m in the physical quantity formula goes out of the sigma and becomes the following.

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

Therefore, the mass can be included in the coefficient calculation.

Since the coefficient changes depending on the type of weighting function, calculate the coefficient for each.

▼ List 5.2 Precompute coefficients for weighting function(FluidBase.cs)

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

Finally, the coefficients (and various parameters) calculated by these CPUs To GPU Store in the constant buffer on the side.

▼ List 5.3 ComputeShader Transfer the value to the constant buffer of(FluidBase.cs)

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

▼ List 5.4 ComputeShader Constant buffer(SPH2D.compute)

```
int   _NumParticles;      // Number of particles
float _TimeStep;          // Step size(dt)
float _Smoothlen;         // Particle radius
float _PressureStiffness; // Becker Coefficient of
float _RestDensity;       // Resting density
float _DensityCoef;       // Coefficient for calculating density
float _GradPressureCoef;  // Coefficient when calculating pressure
float _LapViscosityCoef;  // Coefficient when calculating viscosity
float _WallStiffness;     // Force of pushing back the penalty method
float _Viscosity;         // Viscosity coefficient
float2 _Gravity;          // gravity
float2 _Range;            // Simulation space

float3 _MousePos;         // Mouse position
```

```
15: float _MouseRadius;       // Radius of mouse interaction
16: bool _MouseDown;          // Is the mouse pressed?
```

5.4.3 Density calculation

▼ List 5.5 Kernel function for calculating density(SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void DensityCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x;     // Particle ID currently being processed
 4:
 5:     float h_sq = _Smoothlen * _Smoothlen;
 6:     float2 P_position = _ParticlesBufferRead[P_ID].position;
 7:
 8:     // Neighborhood search(O(n^2))
 9:     float density = 0;
10:     for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
11:             if (N_ID == P_ID) continue;     // Avoid referencing yourself
12:
13:             float2 N_position = _ParticlesBufferRead[N_ID].position;
14:
15:             float2 diff = N_position - P_position;    // 粒子距離
16:             float r_sq = dot(diff, diff);             // 粒子距離の 2 乗
17:
18:             // Exclude particles that are not within the radius
19:             if (r_sq < h_sq) {
20:             // No need to take a route as the calculation only includes squares
21:                     density += CalculateDensity(r_sq);
22:             }
23:     }
24:
25:     // Update density buffer
26:     _ParticlesDensityBufferWrite[P_ID].density = density;
27: }
```

Originally, it is necessary to search for neighboring particles using an appropriate neighborhood search algorithm without exhaustive examination of all particles. For the sake of simplicity, this implementation implements a 100% survey (for loop on line 10). Also, since the distance between yourself and the other particle is calculated, you avoid doing calculations between your own particles on line 11.

Effective radius of weight function h It is realized by the if statement on the 19th line. Addition of densities (calculation of sigma) is realized by adding the calculation result inside sigma to the variable initialized with 0 in the 9th line. Here is the density formula again.

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

The density calculation uses the Poly6 weighting function as shown in the above formula. Poly6 weighting function isList 5.6Calculate with.

▼ List 5.6　密度の計算 (SPH2D.compute)

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

FinallyList 5.5In the 25th line of, write to the write buffer.

5.4.4 Calculation of pressure in particles

▼ List 5.7　Weight function to calculate the pressure for each particle(SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void PressureCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x;     // Particle ID currently being processed
 4:
 5:     float  P_density = _ParticlesDensityBufferRead[P_ID].density;
 6:     float  P_pressure = CalculatePressure(P_density);
 7:
 8:     // Update pressure buffer
 9:     _ParticlesPressureBufferWrite[P_ID].pressure = P_pressure;
10: }
```

Before solving the pressure term, calculate the pressure in particle units, and reduce the calculation cost of the pressure term after that. As I mentioned earlier, it is necessary to solve the equation called Poisson's equation like the following in the calculation of pressure.

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

However, the operation to solve the Poisson equation accurately with a computer is very expensive, so it is approximately calculated using the following Tait equation.

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

▼ List 5.8 Implementation of Tait equation(SPH2D.compute)

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

5.4.5 Calculation of pressure and viscosity terms

▼ List 5.9 Kernel function that calculates pressure and viscosity terms(SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void ForceCS(uint3 DTid : SV_DispatchThreadID) {
 3:     uint P_ID = DTid.x; // Particle ID currently being processed
 4:
 5:     float2 P_position = _ParticlesBufferRead[P_ID].position;
 6:     float2 P_velocity = _ParticlesBufferRead[P_ID].velocity;
 7:     float  P_density = _ParticlesDensityBufferRead[P_ID].density;
 8:     float  P_pressure = _ParticlesPressureBufferRead[P_ID].pressure;
 9:
10:     const float h_sq = _Smoothlen * _Smoothlen;
11:
12:     // Neighborhood search(O(n^2))
13:     float2 press = float2(0, 0);
14:     float2 visco = float2(0, 0);
15:     for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
16:             if (N_ID == P_ID) continue;     // Skip if you target yourself
17:
18:             float2 N_position = _ParticlesBufferRead[N_ID].position;
19:
20:             float2 diff = N_position - P_position;
21:             float r_sq = dot(diff, diff);
22:
23:             // Exclude particles that are not within the radius
24:             if (r_sq < h_sq) {
25:                     float  N_density
26:                     = _ParticlesDensityBufferRead[N_ID].density;
27:                     float  N_pressure
28:                     = _ParticlesPressureBufferRead[N_ID].pressure;
29:                     float2 N_velocity
30:                     = _ParticlesBufferRead[N_ID].velocity;
31:                     float  r = sqrt(r_sq);
32:
33:                     // Pressure item
34:                     press += CalculateGradPressure(...);
35:
36:                     // Sticky item
37:                     visco += CalculateLapVelocity(...);
38:             }
39:     }
40:
41:     // Integration
42:     float2 force = press + _Viscosity * visco;
```

```
43:
44:     // Acceleration buffer update
45:     _ParticlesForceBufferWrite[P_ID].acceleration = force / P_density;
46: }
```

The pressure and viscosity terms are calculated in the same way as the density calculation method.

First, we calculate the force by the following pressure term in line 31.

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

The following function calculates the contents of Sigma.

▼ List 5.10　圧力項の要素の計算 (SPH2D.compute)

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

Next, the calculation of the force by the following viscous term is performed on the 34th line.

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

The following function calculates the contents of Sigma.

▼ List 5.11　Calculate elements of viscosity term(SPH2D.compute)

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

Finally,List 5.9At line 39, the forces calculated by the pressure term and the viscosity term are added together and written in the buffer as the final output.

5.4.6 Collision judgment and position update

▼ List 5.12　Kernel function for collision detection and position update(SPH2D.compute)

```
 1: [numthreads(THREAD_SIZE_X, 1, 1)]
 2: void IntegrateCS(uint3 DTid : SV_DispatchThreadID) {
 3:     const unsigned int P_ID = DTid.x; // Particle ID currently being processed
 4:
 5:     // Position and speed before update
 6:     float2 position = _ParticlesBufferRead[P_ID].position;
 7:     float2 velocity = _ParticlesBufferRead[P_ID].velocity;
 8:     float2 acceleration = _ParticlesForceBufferRead[P_ID].acceleration;
 9:
10:     // Mouse interaction
11:     if (distance(position, _MousePos.xy) < _MouseRadius && _MouseDown) {
12:             float2 dir = position - _MousePos.xy;
13:             float pushBack = _MouseRadius-length(dir);
14:             acceleration += 100 * pushBack * normalize(dir);
15:     }
16:
17:     // If you want to write collision judgment, here -----
18:
19:     // Wall boundary (penalty method)
20:     float dist = dot(float3(position, 1), float3(1, 0, 0));
21:     acceleration += min(dist, 0) * -_WallStiffness * float2(1, 0);
22:
23:     dist = dot(float3(position, 1), float3(0, 1, 0));
24:     acceleration += min(dist, 0) * -_WallStiffness * float2(0, 1);
25:
26:     dist = dot(float3(position, 1), float3(-1, 0, _Range.x));
27:     acceleration += min(dist, 0) * -_WallStiffness * float2(-1, 0);
28:
29:     dist = dot(float3(position, 1), float3(0, -1, _Range.y));
30:     acceleration += min(dist, 0) * -_WallStiffness * float2(0, -1);
31:
32:     // Addition of gravity
33:     acceleration += _Gravity;
34:
35:     // Update the next particle position with the forward Euler method
36:     velocity += _TimeStep * acceleration;
37:     position += _TimeStep * velocity;
38:
39:     // Particle buffer update
40:     _ParticlesBufferWrite[P_ID].position = position;
41:     _ParticlesBufferWrite[P_ID].velocity = velocity;
42: }
```

Use the penalty method to judge collision with a wall(19-30 Line)。The penalty method is a method of pushing back with a strong force as much as it sticks out from the boundary position.

Originally, the collision judgment with the obstacle is also performed before the collision judgment with the wall, but in this implementation, the interaction with the mouse is performed.(213-218 Line)。If the mouse is pressed, the specified force is applied to move away from the mouse position.

In line 33, the external force of gravity is added. Setting the gravity value to

zero creates weightlessness, which is an interesting visual effect. In addition, the position is updated by the forward Euler method described above.(36-37 Line)、Write the final result to the buffer.

5.4.7 Simulation main routine

▼ List 5.13 Simulation main functions(FluidBase.cs)

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

This is the part that calls the kernel function of ComputeShader described so far every frame. Give an appropriate ComputeBuffer for each kernel function.

Where time step width Δt Recall that the smaller the, the less error in the simulation. When running at 60FPS, $\Delta t = 1/60$ However, this will cause a large error and the particles will explode. further, $\Delta t = 1/60$ If the time step width is smaller, the progress of time per frame will be slower than the actual time, resulting in slow motion. To avoid this, $\Delta t = 1/(60 \times iterarion)$ As for, iterarion

turns the main routine once per frame.

▼ List 5.14 Iteration of major functions(FluidBase.cs)

```
1: // Iterate multiple times with smaller time step to improve calculation accuracy
2: for (int i = 0; i<iterations; i++) {
3:     RunFluidSolver();
4: }
```

By doing this, you can perform real-time simulation with a small time step width.

5.4.8 How to use the buffer

Unlike the normal single access particle system, Since particles interact with each other, it would be a problem if other data were rewritten during the calculation. In order to avoid this, prepare two buffers, a read buffer and a write buffer, that do not rewrite the values ☒☒ when performing calculations on the GPU. By exchanging these buffers every frame, data can be updated without conflict.

▼ List 5.15 Buffer swapping function(FluidBase.cs)

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

5.4.9 Particle rendering

▼ List 5.16 Particle rendering(FluidRenderer.cs)

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

On the 10th line, set the buffer that stores the position calculation results of the fluid particles in the material and transfer it to the shader. In the 11th line, we are instructing to draw instances for the number of particles.

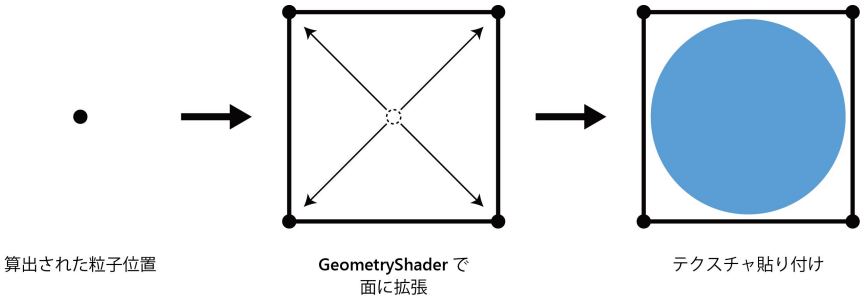
▼ List 5.17 Particle rendering(Particle.shader)

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

1-6 In the line, define the information to receive the fluid particle information. At this time, it is necessary to match the definition with the structure of the buffer transferred from the script to the material. The position data is received as in line 14. id : SV_VertexID This is done by referring to the buffer element with.

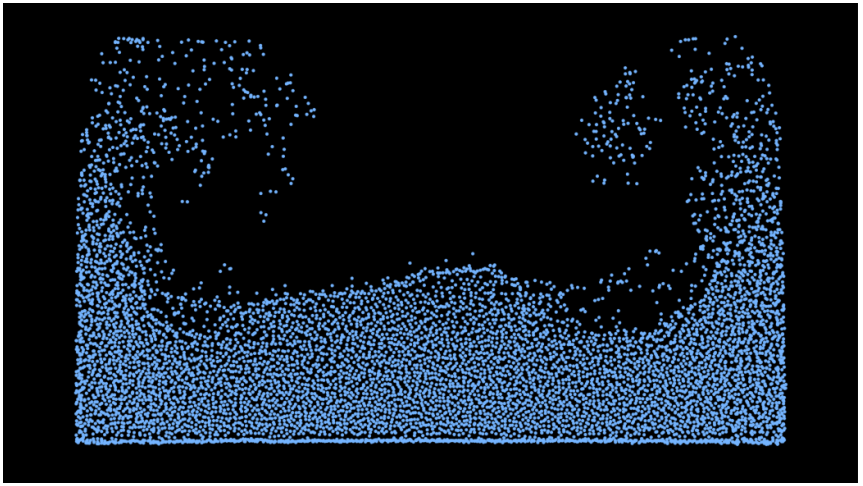
After that, just like a normal particle systemFigure 5.7like Billboard centered on the position data of the calculation result with the geometry shader*10Create Attach the particle image and render.

*10 Plane where the table always faces the viewpoint.



▲図 5.7　Billboard Creation

5.5　result



▲図 5.8　Rendering result

Click here for video(`https://youtu.be/KJVu26zeK2w`)It is posted in.

5.6 Summary

In this chapter, we showed the method of fluid simulation using SPH method. By using the SPH method, it has become possible to handle fluid movements in a general-purpose manner like a particle system.

As mentioned above, there are many types of fluid simulation methods other than the SPH method. Throughout this chapter, you will be interested in other physical simulations in addition to other fluid simulation methods. We would appreciate if you could expand the range of expressions.

第 6 章

Grow grass with geometry shader

6.1 Introduction

In this chapter, we focus on the Geometry Shader, which is one of the stages of the rendering pipeline, and explain the dynamic grass generation shader using the Geometry Shader (so-called Grass Shader).

I'm using some jargon to describe the Geometry Shader, but if you want to use the Geometry Shader for now, it's quick to take a look at the sample code.

The Unity project in this chapter has been uploaded to the following Github repository.

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

6.2 Geometry Shader What is

Geometry Shader Is a programmable shader that can dynamically convert, create, and delete primitives (basic shapes that make up a mesh) on the GPU.

Up to now, when trying to change the mesh shape dynamically, such as converting primitives, it is necessary to perform processing on the CPU, or to add meta information to the vertices in advance and convert with Vertex Shader. did. However, Vertex Shader could not get information about adjacent vertices, and there was a strong constraint that new vertices could not be created based on the vertices being processed or vice versa. .. Also, even if that is done by CPU, it takes an unrealistically huge amount of time from the viewpoint of real-time processing. As described above, there have been some problems with changing the shape of the mesh in real time.

Therefore, as a function to solve these problems and allow conversion processing freely under weak constraints, Geometry Shader is standardly installed in DirectX 10 and OpenGL 3.2. In OpenGL, it is also called the Primitive Shader.

6.3 Geometry Shader Features of

6.3.1 Rendering pipeline

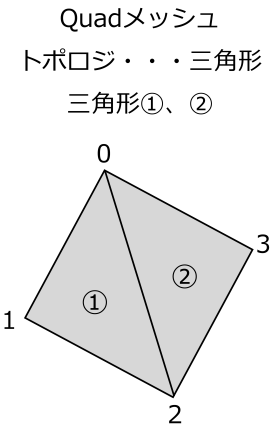
On the rendering pipeline Vertex Shader Next to Fragment Shader It is located before the rasterization process. In other words, in the Fragment Shader, it is processed without distinguishing between the vertices dynamically generated by the Geometry Shader and the original vertices passed to the Vertex Shader.

6.3.2 Geometry Shader Input to

Normally, the input information to the Vertex Shader is in units of vertices, and the conversion process for that vertex is performed. However, the input information to the Geometry Shader will be the primitive unit for input defined by the user.

The actual program is described later, but the vertex information group processed by Vertex Shader will be divided and input based on the input primitive type. For example, if the input primitive type is triangle, three vertex information will be passed, if it is line, two vertex information will be passed, and if it is point, one vertex information will be passed. This makes it possible to perform processing while referring to other vertex information, which was not possible with the vertex shader, and it becomes possible to perform a wide range of calculations.

One point to note is that the Vertex Shader performs processing on a per-vertex basis, and information about the vertices to be processed is passed. Processing is performed in units of the primitive determined by. In other words, when Geometry Shader is executed on a Quad mesh whose topology is Triangles as shown in Figure 6.1, Geometry Shader is executed twice for triangles ① and ②. At this time, if the input primitive type is Line, the information passed to the input is the triangle of ①, two vertices of vertices 0, 1 and 2, and ② of vertices of 0, 2, 3 It is the apex of two points.



▲図 6.1　Quad mesh

6.3.3 Geometry Shader Output from

Geometry Shader The output of is the vertex information group for the user-defined output primitive type. Vertex Shader With 1 input and 1 output, the Geometry Shader outputs multiple pieces of information, and there is no problem even if there is more than one primitive generated by the output information.

For example, if you define the output primitive type as triangle and output a total of 9 vertices newly calculated, 3 triangles are generated by the Geometry Shader. Since this processing is performed in primitive units as described above, it is possible that the number of triangles that was originally one has increased to three.

Also, Geometry Shader For MaxVertexCount, it is necessary to set in advance the maximum number of vertices to be output in one process. For example, if MaxVertexCount is set to 9, the Geometry Shader will be able to output the number of vertices from 0 to 9. This number is generally the maximum value of 1024 due to the "Geometry Shader limit" described later.

It is important to note that when outputting vertex information, when adding a new vertex while maintaining the original mesh shape, the vertex information sent from the Vertex Shader should also be added to the Geometry Shader. Should be output. The Geometry Shader is not the behavior of adding to the output

of the Vertex Shader, but the output of the Geometry Shader is rasterized and passed to the Fragment Shader. Paradoxically, you can also reduce the number of vertices dynamically by setting the output of the Geometry Shader to 0.

6.3.4 Geometry Shader Limits

The Geometry Shader has the limitation of the maximum number of output vertices and the maximum number of output elements for one output. The maximum number of output vertices is literally the limit value of the number of vertices, and although it is a numerical value that depends on the GPU, 1024 etc. is common, so you can increase the number of vertices from one triangle to a maximum of 1024 points. The elements in the maximum number of output elements are the information that the vertices such as coordinates and colors have, and generally the position element of (x, y, z, w) and (r, g, b, a) There are a total of 8 color elements. The maximum output number of this element also depends on the GPU, but 1024 is also common, so the output will be limited to 128 (1024/8) at the maximum.

Since these two restrictions need to be satisfied, even if it is possible to output 1024 points in terms of the number of vertices, due to the constraint on the number of elements side, the actual Geometry Shader The output of is limited to 128 points. So, for example, if you use Geometry Shader for a mesh with 2 primitives (Quad mesh etc.), you can only handle vertices up to 256 points (128 points * 2 primitives). ..

This number of 128 points is the limit value of the value that can be set in MaxVertexCount in the previous section.

6.4 Easy Geometry Shader

Below is a simple Geometry Shader program. The explanation up to the previous section will be explained again with reference to the actual program.

In addition to the Geometry Shader, the explanation of the ShaderLab syntax required when writing shaders in Unity is omitted in this chapter, so if you have any questions, please refer to the official document below.

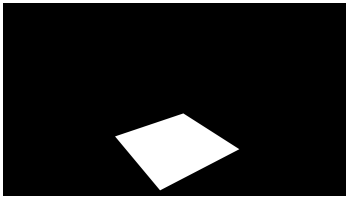
https://docs.unity3d.com/ja/current/Manual/SL-Reference.html

```
Shader "Custom/SimpleGeometryShader"
{
    Properties
    {
        _Height("Height", float) = 5.0
        _TopColor("Top Color", Color) = (0.0, 0.0, 1.0, 1.0)
        _BottomColor("Bottom Color", Color) = (1.0, 0.0, 0.0, 1.0)
    }
    SubShader
    {
        Tags { "RenderType" = "Opaque"}
        LOD 100

        Cull Off
        Lighting Off

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

            uniform float _Height;
            uniform float4 _TopColor, _BottomColor;

            struct v2g
            {
                float4 pos : SV_POSITION;
            };

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

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

                return o;
            }

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

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

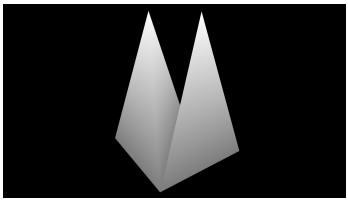
```
                g2f out0;
                out0.pos = UnityObjectToClipPos(p0);
                out0.col = _BottomColor;

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

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

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

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

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

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

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

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

In this shader, the center coordinates of the passed triangle are calculated, moved further upward, and each vertex of the passed triangle is connected to the new calculated coordinates. In other words, you are generating a simple triangular pyramid from a flat triangle.

So when you apply this shader to a Quad mesh (consisting of two triangles), you get something like Figure 6.2 to Figure 6.3.



▲図 6.2　From a flat plate like this



▲図 6.3　Two three-dimensional pyramids will be displayed

Of this shader, I will extract and explain only the part related to Geometry Shader.

▼

```
#pragma target 5.0
#pragma vertex vert

// Geometry Shader Declare use of
#pragma geometry geom

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

In the declaration section above, `geom`I declare that the function named is for Geometry Shader. By doing this, when it comes to the Geometry Shader stage`geom`The function will be called.

▼

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

Here is the function declaration for the Geometry Shader.

6.4.1 input

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

This is the part related to input.

Since I want to generate a triangular pyramid from a triangle this time, input is`triangle`I am. As a result, each vertex information of the triangle which is a unit primitive is input, and since the triangle consists of three vertices, the received dummy argument is an array of length 3. So if you type`triangle`not`po int`If you set to, there is only one vertex to configure, so`geom(point v2f input [1])`You will receive it as an array of length 1 like.

6.4.2 output

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

This is the output part.

Since we want to make the primitive of the mesh generated this time a triangle,`T riangleStream` It is declared with a type. `TriangleStrema`Since the type means that the output is a triangle strip, it will generate a triangle based on the output vertex information. Other`PointStream` Type`LineStream`Since there are types etc., it is necessary to select the output primitive type according to the purpose.

Also,`[maxvertexcount(12)]`The maximum output number is set to 12 in the part. This is because the number of triangles that make up the triangular pyramid is one on the bottom and three on the side, and there are a total of four vertices per triangle, so 3 * 4 outputs 12 vertices. It is set to 12 because it will be different.

6.4.3 processing

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

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

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

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

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

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

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

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

This is the part of the process that outputs the actual vertices.

First of all, g2f type variable for output is declared and the vertex coordinates and color information are stored. At this time, it is necessary to convert from the object space to the camera clip space in the same way as the Vertex Shader.

After that, the vertex information is output while being aware of the order of the vertices that make up the mesh. `outStream`Variable`Append`By passing an output variable to the function, it will be added to the current stream,`RestartStrip`Calling the function ends the current primitive strip and starts a new stream.

this is,`TriangleStream`Is a triangle strip, so`Append`As more vertices are added by the function, multiple connected triangles will be created based on all vertices added to the stream. So, like this time, triangles`Append`If you have trouble connecting based on the order`RestartStrip`You need to call to start a new stream. of course`Append`By devising the order`RestartStrip`It is possible to reduce the number of function calls.

6.5 Grass Shader

This section explains the Grass Shader that uses the Geometry Shader to generate grass in real time, which is a slight development of the "Simple Geometry Shader" in the previous section.

The following is the program of Grass Shader to be described.

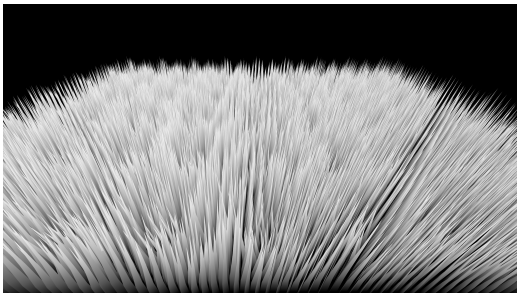
```
Shader "Custom/Grass" {
    Properties
    {
        // Grass height
        _Height("Height", float) = 80
        // Width of grass
        _Width("Width", float) = 2.5

        // Height of bottom of grass
        _BottomHeight("Bottom Height", float) = 0.3
        // Height of middle part of grass
        _MiddleHeight("Middle Height", float) = 0.4
        // Top of grass
        _TopHeight("Top Height", float) = 0.5

        // Width of bottom of grass
        _BottomWidth("Bottom Width", float) = 0.5
        // Width of middle part of grass
        _MiddleWidth("Middle Width", float) = 0.4
        // Width of the top of the grass
        _TopWidth("Top Width", float) = 0.2

        // Bending of the bottom of the grass
        _BottomBend("Bottom Bend", float) = 1.0
        // Bending of the middle part of the grass
        _MiddleBend("Middle Bend", float) = 1.0
        // Bending of the upper part of the grass
        _TopBend("Top Bend", float) = 2.0

        // Wind strength
        _WindPower("Wind Power", float) = 1.0

        // Color of the top of the grass
        _TopColor("Top Color", Color) = (1.0, 1.0, 1.0, 1.0)
        // Color of the bottom of the grass
        _BottomColor("Bottom Color", Color) = (0.0, 0.0, 0.0, 1.0)

        // A noise texture that gives randomness to the height of the grass
        _HeightMap("Height Map", 2D) = "white"
        // Noise texture that gives randomness to the direction of grass
        _RotationMap("Rotation Map", 2D) = "black"
        // Noise texture that gives randomness to wind strength
        _WindMap("Wind Map", 2D) = "black"
    }
    SubShader
```

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

        LOD 100
        Cull Off

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

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

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

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

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

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

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

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

                return o;
            }

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

```
                float4 p1 = i[1].pos;
                float4 p2 = i[2].pos;

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

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

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

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

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

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

                g2f o[7];

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

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

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

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

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

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

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

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

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

```
                o[3].pos += dir
                            * (_WindPower * wind * _BottomBend)
                            * sin(_Time);
                o[4].pos += dir
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o[5].pos += dir
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o[6].pos += dir
                            * (_WindPower * wind * _TopBend)
                            * sin(_Time);

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

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

When this shader is applied to a Plane mesh with multiple rows and columns, it becomes as shown in Figure 6.4.

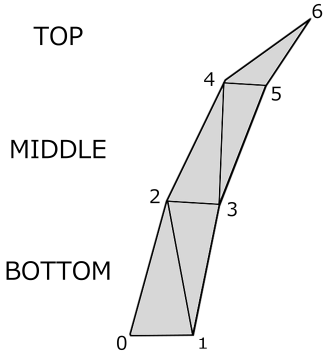


▲図 6.4　Grass Shader の結果

I will explain the process of generating grass from this.

6.5.1 Basic policy

This time, we will generate one grass for each primitive. Regarding the generation of the grass shape, as shown in Figure 6.5, a total of 7 vertices are generated by dividing into the lower part, the middle part, and the upper part. I will.



▲図 6.5　How to make a grass shape

6.5.2 parameter

Although the details are described in the comments, the coefficient that controls the width and height of each part (lower part, middle part, upper part) in one grass, the coefficient that controls the width and height of the entire grass It is prepared as the main parameter. Also, it doesn't look good if each grass has the same shape, so use a noise texture to give it randomness.

6.5.3 processing

▼

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

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

In this part, we calculate the height and direction of the grass, and the numerical value that serves as the standard for the strength of the wind. It may be calculated in the Geometry Shader, but it is possible to calculate it with the Vertex Shader because it is possible to handle it like the initial value when performing calculation on the Geometry Shader if the vertex has meta-information. I am.

▼

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

Here, the center of the grass and the direction of growing grass are calculated. If you decide the part here with noise texture, you can give randomness to the direction in which the grass grows.

▼

```
float bottomHeight = height * _Height * _BottomHeight;

...

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

The program is abbreviated because it is long. In this part, the height and width of the lower part, middle part and upper part are calculated respectively, and the coordinates are calculated based on that.

▼

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

The 7 vertices calculated in this part`Append`doing. This time it doesn't matter if the triangles are connected and generated, so`RestartStrip`I haven't.

In addition,`for`For statements`[unroll]`Is applied. This is an attribute that expands the processing in the loop by the number of loops at compile time, and it has the disadvantage of increasing the memory size, but it has the advantage of operating at high speed.

6.6 Summary

So far, I have explained from the description of Geometry Shader to the basic and applied programs. There are some differences in characteristics from writing a program that runs on a CPU, but it should be possible to use it by suppressing the basic parts.

In general, it is said that Geometry Shader is slow. I myself haven't felt that much, but it may be difficult when the scope of use becomes large. If you think you will be using the Geometry Shader on a large scale, please try benchmarking once.

Still, I think that being able to dynamically create new meshes and delete them on the GPU will broaden the range of ideas. Personally, I think the most important thing is not what technology was used but what is created and expressed by it. We hope you will learn and learn about a tool called Geometry Shader in this chapter, and feel new possibilities.

6.7 reference

- Tutorial 13: Geometry Shader - https://msdn.microsoft.com/ja-jp/library/bb172497
- Geometry shader object in MSDN - https://msdn.microsoft.com/ja-jp/library/ee418313
- Rendering Method for Transparent Geometry by Cutting Geometry Shader Geometry - http://t-pot.com/program/147_CGGONG2008/index.html

第 7 章

Introduction to the Marching Cubes method starting with the atmosphere

7.1 What is the Marching Cubes method?

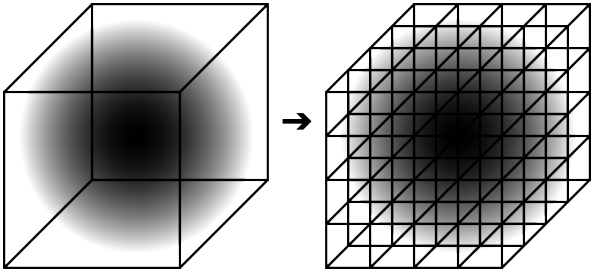
7.1.1 History and overview

The marching cubes method is one of the volume rendering methods and is an algorithm that converts 3D voxel data filled with scalar data into polygon data. The first paper was published in 1987 by William E. Lorensen and Harvey E. Cline.

The Marching Cubes method was patented, but since it was expired in 2005, it is now freely available.

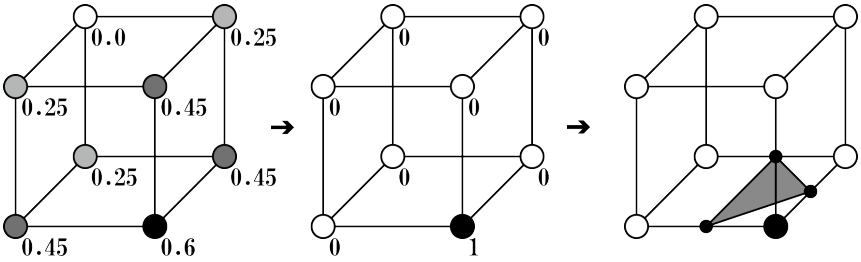
7.1.2 Explanation of simple mechanism

First, the volume data space is divided into three-dimensional grids.



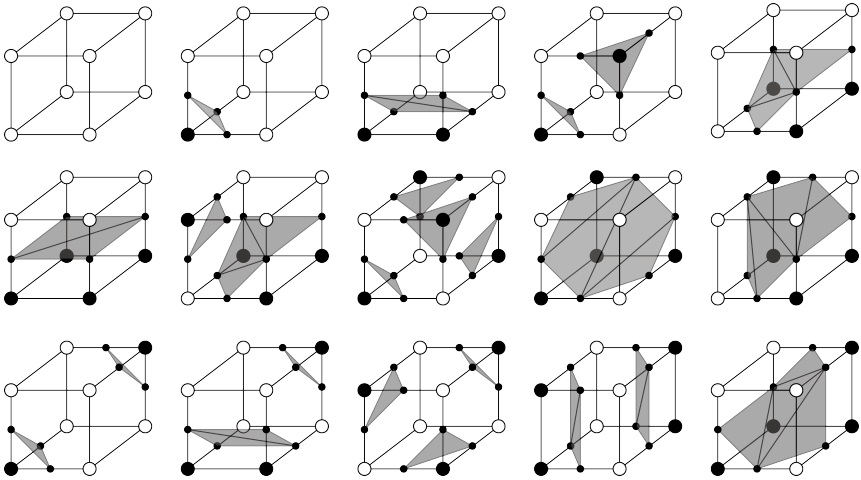
▲図 7.1　3D volume data and grid division

Next, let's take out one of the divided grids. The boundary of 8 vertices is calculated as 1 if the values ☒☒ of the 8 corners of the grid are greater than or equal to the threshold, and 0 if they are less than the threshold.
The figure below shows the flow when the threshold is set to 0.5.



▲図 7.2　Determine boundaries according to corner values

There are 256 combinations of the eight corners, but if you make full use of rotation and inversion, it will fit in 15 types. A triangular polygon pattern corresponding to the 15 combinations is assigned.



▲図 7.3　Combination of corners

7.2 Sample repository

The sample project explained in this chapter is Unity Graphics Programming's Unity project. https://github.com/IndieVisualLab/UnityGraphicsProgramming 内にある Assets/GPUMarchingCubes にあります。

For implementation, Paul Bourke's site of Polygonising a scalar field[*1]I ported it to Unity with reference to.

This time, I will explain along this sample project.

There are three major implementations.

- Initialization of mesh, drawing registration process for each frame (C# script part)
- ComputeBuffer initialization
- Actual drawing process (shader part)

First, initialize the mesh and register the drawing. **GPUMarchingCubes-DrawMesh** I will make it from the class.

[*1] Polygonising a scalar field http://paulbourke.net/geometry/polygonise/

7.2.1 GeometryShader Make a mesh for

As explained in the previous section, the marching cubes method is an algorithm that generates polygons by combining the eight corners of a grid. To do that in real time, you need to dynamically create polygons.
However, it is inefficient to generate the vertex array of the mesh on the CPU side (C# side) every frame.
So we use Geometry Shader. GeometryShader is roughly a Shader located between VertexShader and FragmentShader, and can increase or decrease the vertices processed by VertexShader.
For example, a plate polygon can be generated by adding 6 vertices around one vertex.
Furthermore, it is very fast because it is processed on the Shader side (GPU side).
This time I will use the Geometry Shader to generate and display the Marching Cubes polygons.

First, **GPUMarchingCubesDrawMesh**Define the variables used in the class.

▼ List 7.1 Definition part of variable group

```
using UnityEngine;

public class GPUMarchingCubesDrawMesh : MonoBehaviour {

    #region public
    public int segmentNum = 32;                 // The number of divisions on one side of the

    [Range(0,1)]
    public float threashold = 0.5f;             // Threshold of scalar value to mesh
    public Material mat;                        // Material for rendering

    public Color DiffuseColor = Color.green;    // Diffuse color
    public Color EmissionColor = Color.black;   // Emitting color
    public float EmissionIntensity = 0;         // Luminous intensity

    [Range(0,1)]
    public float metallic = 0;                  // Metallic feeling
    [Range(0, 1)]
    public float glossiness = 0.5f;             // Gloss
    #endregion

     #region private
    int vertexMax = 0;                          // Number of vertices
    Mesh[] meshs = null;                        // Mesh array
    Material[] materials = null;                // Material array for each mesh
    float renderScale = 1f / 32f;               // Display scale
    MarchingCubesDefines mcDefines = null;      // MarchingCubes Constant array group
    #endregion
```

```
}
```

Next, create a mesh to pass to Geometry Shader. The vertices of the mesh should be placed one by one in the divided 3D grid. For example, when the number of divisions on one side is 64, 64*64*64=262,144 vertices are required.

However, in Unity 2017.1.1f1, the maximum number of vertices in one mesh is 65,535. Therefore, each mesh is divided so that the number of vertices is within 65,535.

▼ List 7.2 Mesh creation part

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log("VertexMax " + vertexMax);

  // Divide the size of 1 Cube by segmentNum to determine the size for rendering
  renderScale = 1f / segmentNum;

  CreateMesh();

  // Initialization of constant array for Marching Cubes used in shader
  mcDefines = new MarchingCubesDefines();
}

void CreateMesh()
{
  // Since the upper limit of the number of vertices of the mesh is 65535, divide the mesh
  int vertNum = 65535;
  int meshNum = Mathf.CeilToInt((float)vertexMax / vertNum);  // Number of meshes to split
  Debug.Log("meshNum " + meshNum );

  meshs = new Mesh[meshNum];
  materials = new Material[meshNum];

  // Bounce calculation for Mesh
  Bounds bounds = new Bounds(
    transform.position,
    new Vector3(segmentNum, segmentNum, segmentNum) * renderScale
  );

  int id = 0;
  for (int i = 0; i < meshNum; i++)
  {
    // 頂点作成
    Vector3[] vertices = new Vector3[vertNum];
    int[] indices = new int[vertNum];
    for(int j = 0; j < vertNum; j++)
    {
      vertices[j].x = id % segmentNum;
      vertices[j].y = (id / segmentNum) % segmentNum;
      vertices[j].z = (id / (segmentNum * segmentNum)) % segmentNum;
```

```
        indices[j] = j;
        id++;
      }

      // Mesh 作成
      meshs[i] = new Mesh();
      meshs[i].vertices = vertices;
      // Mesh Topology is Points because polygons are created with Geometry Shader
      meshs[i].SetIndices(indices, MeshTopology.Points, 0);
      meshs[i].bounds = bounds;

      materials[i] = new Material(mat);
    }
  }
```

7.2.2 ComputeBuffer の初期化

MarchingCubesDefinces.cs In the source, a constant array used in the rendering of the marching cubes method and a ComputeBuffer for passing the constant array to the shader are defined. ComputeBuffer is a buffer that stores the data used by the shader. Since the data is stored in the memory on the GPU side, access from the shader is fast.

Actually, it is possible to define the constant array used in the rendering of the Marching Cubes method on the shader side. However, the reason why the constant array used by the shader is initialized on the C# side is that the shader has a limitation that the number of literal values ☒☒ (directly written values) can be registered up to 4096. If you define a huge array of constants in your shader, you will quickly hit the upper limit on the number of literal values.

Therefore, by storing it in ComputeShader and passing it, it will not be a literal value, so it will not hit the upper limit. Because of this the processing will increase a little, but I am trying to store the constant array in ComputeBuffer on the C# side and pass it to the shader.

▼ List 7.3 ComputeBuffer Initialization part of

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log("VertexMax " + vertexMax);

  // 1Cube The size of segmentNum Divide by to determine the size for rendering
  renderScale = 1f / segmentNum;

  CreateMesh();
```

```
  // Initialization of constant array for Marching Cubes used in shader
  mcDefines = new MarchingCubesDefines();
}
```

In the Initialize() function above, the MarchingCubesDefines are initialized.

7.2.3 rendering

Next is a function that calls the rendering process.
This time we will use Graphics.DrawMesh() to render multiple meshes at once and be able to be influenced by Unity's lighting. The meaning of DiffuseColor etc. defined by public variables will be explained in the explanation on the shader side.

ComputeBuffers of the MarchingCubesDefines class in the previous section are passed to the shader with material.setBuffer.

▼ List 7.4 Rendering part

```
void RenderMesh()
{
  Vector3 halfSize = new Vector3(segmentNum, segmentNum, segmentNum)
                   * renderScale * 0.5f;
  Matrix4x4 trs = Matrix4x4.TRS(
                   transform.position,
                   transform.rotation,
                   transform.localScale
                 );

  for (int i = 0; i < meshs.Length; i++)
  {
    materials[i].SetPass(0);
    materials[i].SetInt("_SegmentNum", segmentNum);
    materials[i].SetFloat("_Scale", renderScale);
    materials[i].SetFloat("_Threashold", threashold);
    materials[i].SetFloat("_Metallic", metallic);
    materials[i].SetFloat("_Glossiness", glossiness);
    materials[i].SetFloat("_EmissionIntensity", EmissionIntensity);

    materials[i].SetVector("_HalfSize", halfSize);
    materials[i].SetColor("_DiffuseColor", DiffuseColor);
    materials[i].SetColor("_EmissionColor", EmissionColor);
    materials[i].SetMatrix("_Matrix", trs);

    Graphics.DrawMesh(meshs[i], Matrix4x4.identity, materials[i], 0);
  }
}
```

7.3 call

▼ List 7.5 Call part

```
// Use this for initialization
void Start ()
{
  Initialize();
}

void Update()
{
  RenderMesh();
}
```

Start() ↷ Initialize()To generate a mesh, Update()RenderMesh with a function()Call to render.
Update() ↷ RenderMesh()The reason for calling Graphics.DrawMesh()This is because it doesn't draw immediately, but it's like "register once for rendering".
By registering, Unity will adapt the lights and shadows. There's a similar function, Graphics.DrawMeshNow(), but it draws instantly, so Unity's lights and shadows don't apply. Also, it should be called in OnRenderObject(), OnPostRender(), etc. instead of Update().

7.4 Shader side implementation

The shader this time is roughly divided into two parts: **"Entity rendering part"** and **"Shadow rendering part"**. In addition, three shader functions are executed within each: vertex shader, geometry shader, and fragment shader.

Since the shader source is long, let's have a look at the sample project for the entire implementation, and explain only the essential points. The shader file to explain is GPU ArchingCubesRenderMesh.shader.

7.4.1 Variable declaration

The upper part of the shader defines the structure used for rendering.

▼ List 7.6 Definition part of structure

```
// Vertex data coming from the mesh
struct appdata
{
  float4 vertex : POSITION; // Vertex coordinates
};

//Data passed from the vertex shader to the geometry shader
struct v2g
{
  float4 pos : SV_POSITION; // Vertex coordinates
};

// Data passed from the geometry shader to the fragment shader during entity rendering
struct g2f_light
{
  float4 pos      : SV_POSITION;  // Local coordinates
  float3 normal   : NORMAL;       // Normal
  float4 worldPos : TEXCOORD0;    // World coordinates
  half3 sh        : TEXCOORD3;    // SH
};

// Data to pass from the geometry shader to the fragment shader when rendering shadows
struct g2f_shadow
{
  float4 pos      : SV_POSITION;  // 座標
  float4 hpos     : TEXCOORD1;
};
```

Next we define the variables.

▼ List 7.7 Variable definition part

```
int _SegmentNum;

float _Scale;
float _Threashold;

float4 _DiffuseColor;
float3 _HalfSize;
float4x4 _Matrix;

float _EmissionIntensity;
half3 _EmissionColor;

half _Glossiness;
half _Metallic;

StructuredBuffer<float3> vertexOffset;
StructuredBuffer<int> cubeEdgeFlags;
StructuredBuffer<int2> edgeConnection;
StructuredBuffer<float3> edgeDirection;
StructuredBuffer<int> triangleConnectionTable;
```

The contents of various variables defined here are passed by the material.Set

○○ function in the RenderMesh() function on the C# side. ComputeBuffers of MarchingCubesDefines class have changed the type name as StructuredBuffer< ○○>.

7.4.2 Vertex shader

The vertex shader is pretty simple, as most of the processing is done by the geometry shader. It simply passes the vertex data passed from the mesh directly to the geometry shader.

▼ List 7.8 Implementation part of vertex shader

```
// Vertex data coming from the mesh
struct appdata
{
  float4 vertex : POSITION; // 頂点座標
};

// Data passed from the vertex shader to the geometry shader
struct v2g
{
  float4 pos : SV_POSITION; // 座標
};

// Vertex shader
v2g vert(appdata v)
{
  v2g o = (v2g)0;
  o.pos = v.vertex;
  return o;
}
```

By the way, the vertex shader is common to the substance and the shadow.

7.4.3 Entity geometry shader

Since it is long, I will explain it while dividing it.

▼ List 7.9 Function declaration part of geometry shader

```
// Entity geometry shader
[maxvertexcount(15)]  // Definition of maximum number of vertices output from shader
void geom_light(point v2g input[1],
                inout TriangleStream<g2f_light> outStream)
```

First, the declarative part of the geometry shader.

`[maxvertexcount(15)]`Is the definition of the maximum number of vertices output from the shader. In the marching cubes algorithm this time, up to 5

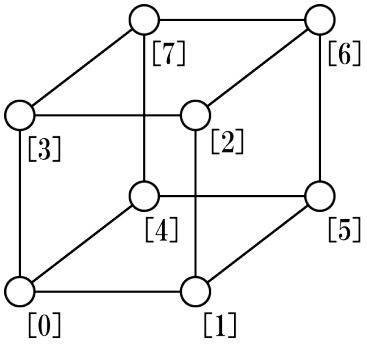
triangular polygons can be created per grid, so a total of 15 vertices will be output in 3*5.
Therefore, enter 15 in () of maxvertexcount.

▼ List 7.10　Scalar value acquisition part of the eight corners of the grid

```
float cubeValue[8]; // Array for getting scalar values □                □
of the eight corners of the grid

// Get the scalar values □□ of the eight corners of the grid
for (i = 0; i < 8; i++) {
  cubeValue[i] = Sample(
                  pos.x + vertexOffset[i].x,
                  pos.y + vertexOffset[i].y,
                  pos.z + vertexOffset[i].z
  );
}
```

pos contains the coordinates of the vertices placed in grid space when creating the mesh. vertexOffset is an array of offset coordinates to add to pos as the name implies.

This loop gets the scalar value in the volume data of the coordinates of 1 corner = 8 corners of 1 grid. vertexOffset refers to the order of the corners of the grid.



▲図 7.4　Order of the coordinates of the corners of the grid

▼ List 7.11　Sampling function part

```
// Sampling function
float Sample(float x, float y, float z) {

  // Are the coordinates outside the grid space?
```

```
    if ((x <= 1) ||
        (y <= 1) ||
        (z <= 1) ||
        (x >= (_SegmentNum - 1)) ||
        (y >= (_SegmentNum - 1)) ||
        (z >= (_SegmentNum - 1))
       )
      return 0;

    float3 size = float3(_SegmentNum, _SegmentNum, _SegmentNum);

    float3 pos = float3(x, y, z) / size;

    float3 spPos;
    float result = 0;

    // Distance function of three spheres
    for (int i = 0; i < 3; i++) {
      float sp = -sphere(
        pos - float3(0.5, 0.25 + 0.25 * i, 0.5),
        0.1 + (sin(_Time.y * 8.0 + i * 23.365) * 0.5 + 0.5) * 0.025) + 0.5;
      result = smoothMax(result, sp, 14);
    }

    return result;
  }
```

It is a function to get the scalar value of the specified coordinate from the volume data. This time, instead of enormous 3D volume data, a simple algorithm that uses the distance function is used to calculate the scalar value.

About distance function

The 3D shape drawn by the marching cubes method this time is「**距離関数**」It is defined using
Roughly speaking, the distance function here is a "function that satisfies the distance condition."
For example, the distance function for a sphere is

▼ List 7.12　Sphere distance function

```
inline float sphere(float3 pos, float radius)
{
    return length(pos) - radius;
}
```

pos The coordinates are entered in, but the center coordinates of the sphere are the origin.(0,0,0)I will think in that case. radius is the radius.
The length is calculated with length(pos), but this is the distance from the origin to pos, and it is subtracted with the radius radius, so if the length is less than or equal to the radius, it will be a natural but negative value.
In other words, if you pass the coordinate pos and a negative value is returned, you can judge that "the coordinate is inside the sphere".
The advantage of the distance function is that the program can be made smaller because the figure can be expressed with a simple calculation formula of a few lines. You can find a lot of information about other distance functions on Inigo Quilez's site.
http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

▼ List 7.13　Composite of distance functions of three spheres

```
// Distance function of three spheres
for (int i = 0; i < 3; i++) {
  float sp = -sphere(
    pos - float3(0.5, 0.25 + 0.25 * i, 0.5),
    0.1 + (sin(_Time.y * 8.0 + i * 23.365) * 0.5 + 0.5) * 0.025) + 0.5;
  result = smoothMax(result, sp, 14);
}
```

This time, I use 8 corners (vertices) of 1 grid as pos. The distance from the center of the sphere is directly treated as the density of volume data.

As will be described later, the sign is inverted because polygons are created when the threshold value is 0.5 or more. Also, the coordinates are delicately shifted to find the distances to the three spheres.

▼ List 7.14 smoothMax 関数

```
float smoothMax(float d1, float d2, float k)
{
  float h = exp(k * d1) + exp(k * d2);
  return log(h) / k;
}
```

smoothMax is a function that blends the results of distance functions nicely. You can use this to fuse three spheres like a metaball.

▼ List 7.15 Threshold check

```
// Check if the values ☒☒ of the eight corners of the grid exceed the threshold
for (i = 0; i < 8; i++) {
  if (cubeValue[i] <= _Threashold) {
    flagIndex |= (1 << i);
  }
}

int edgeFlags = cubeEdgeFlags[flagIndex];

// Draw nothing if empty or completely filled
if ((edgeFlags == 0) || (edgeFlags == 255)) {
  return;
}
```

If the scalar value at the corner of the grid exceeds the threshold, set a bit in flagIndex. Using that flagIndex as an index, the information for generating polygons is extracted from the cubeEdgeFlags array and stored in edgeFlags. If all the corners of the grid are below the threshold or above the threshold, the polygon is not generated because it is completely inside or outside.

▼ List 7.16 Calculation of polygon vertex coordinates

```
float offset = 0.5;
float3 vertex;
for (i = 0; i < 12; i++) {
  if ((edgeFlags & (1 << i)) != 0) {
    // Get threshold offset between corners
    offset = getOffset(
               cubeValue[edgeConnection[i].x],
               cubeValue[edgeConnection[i].y], _
               Threashold
             );
```

```
    // Complement the coordinates of the vertices based on the offset
    vertex = vertexOffset[edgeConnection[i].x]
             + offset * edgeDirection[i];

    edgeVertices[i].x = pos.x + vertex.x * _Scale;
    edgeVertices[i].y = pos.y + vertex.y * _Scale;
    edgeVertices[i].z = pos.z + vertex.z * _Scale;

    // Normal calculation (in order to resample, the vertex coordinates before scaling are re
    edgeNormals[i] = getNormal(
                       defpos.x + vertex.x,
                       defpos.y + vertex.y,
                       defpos.z + vertex.z
                     );
  }
}
```

This is where the vertex coordinates of the polygon are calculated. By looking at the bit of edgeFlags, we are calculating the vertex coordinates of the polygon placed on the side of the grid.

getOffset outputs the ratio (offset) from the current corner to the next corner from the scalar value and threshold of the two corners of the grid. By shifting from the coordinates of the current corner to the direction of the next corner by offset, the final polygon becomes a smooth polygon.

In getNormal, the normal line is calculated by re-sampling and giving the slope.

▼ List 7.17 Make polygons by connecting vertices

```
// Create polygon by connecting vertices
int vindex = 0;
int findex = 0;
// Up to 5 triangles
for (i = 0; i < 5; i++) {
  findex = flagIndex * 16 + 3 * i;
  if (triangleConnectionTable[findex] < 0)
    break;

  // Make a triangle
  for (j = 0; j < 3; j++) {
    vindex = triangleConnectionTable[findex + j];

    // Multiply the Transform matrix to convert to world coordinates
    float4 ppos = mul(_Matrix, float4(edgeVertices[vindex], 1));
    o.pos = UnityObjectToClipPos(ppos);

    float3 norm = UnityObjectToWorldNormal(
                    normalize(edgeNormals[vindex])
                  );
    o.normal = normalize(mul(_Matrix, float4(norm,0)));

    outStream.Append(o);  // Add vertices to strip
```

```
    }
    outStream.RestartStrip(); // Break and start the next primitive strip
  }
```

This is the place where the polygons are made by connecting the vertex coordinate groups obtained earlier. Contains the indices of the vertices that connect to the triangleConnectionTable array. It is converted to world coordinates by multiplying the vertex coordinates by the Transform matrix, and then converted to screen coordinates with UnityObjectToClipPos().

In addition, the normal line is also converted to the world coordinate system with UnityObjectToWorldNormal(). These vertices and normals will be used for lighting in the following fragment shader.

TriangleStream.Append() and RestartStrip() are special functions for geometry shaders. Append()Adds vertex data to the current strip. RestartStrip()Creates a new strip. Since it is a Triangle Stream, it is an image to append up to 3 to one strip.

7.4.4 Entity fragment shader

In order to reflect lighting such as GI (Global Illumination) of Unity, the lighting processing part of Surface Shader after Generate code is ported.

▼ List 7.18 Fragment shader definition

```
// Entity fragment shader
void frag_light(g2f_light IN,
    out half4 outDiffuse        : SV_Target0,
    out half4 outSpecSmoothness : SV_Target1,
    out half4 outNormal         : SV_Target2,
    out half4 outEmission       : SV_Target3)
```

Output to output to G-Buffer(SV_Target)There are four.

▼ List 7.19 Initialize SurfaceOutputStandard structure

```
#ifdef UNITY_COMPILER_HLSL
    SurfaceOutputStandard o = (SurfaceOutputStandard)0;
#else
    SurfaceOutputStandard o;
#endif
    o.Albedo = _DiffuseColor.rgb;
    o.Emission = _EmissionColor * _EmissionIntensity;
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
```

```
    o.Alpha = 1.0;
    o.Occlusion = 1.0;
    o.Normal = normal;
```

Set parameters such as color and gloss in the SurfaceOutputStandard structure that will be used later.

▼ List 7.20 GI related processing

```
// Setup lighting environment
UnityGI gi;
UNITY_INITIALIZE_OUTPUT(UnityGI, gi);
gi.indirect.diffuse = 0;
gi.indirect.specular = 0;
gi.light.color = 0;
gi.light.dir = half3(0, 1, 0);
gi.light.ndotl = LambertTerm(o.Normal, gi.light.dir);

// Call GI (lightmaps/SH/reflections) lighting function
UnityGIInput giInput;
UNITY_INITIALIZE_OUTPUT(UnityGIInput, giInput);
giInput.light = gi.light;
giInput.worldPos = worldPos;
giInput.worldViewDir = worldViewDir;
giInput.atten = 1.0;

giInput.ambient = IN.sh;

giInput.probeHDR[0] = unity_SpecCube0_HDR;
giInput.probeHDR[1] = unity_SpecCube1_HDR;

#if UNITY_SPECCUBE_BLENDING || UNITY_SPECCUBE_BOX_PROJECTION
// .w holds lerp value for blending
giInput.boxMin[0] = unity_SpecCube0_BoxMin;
#endif

#if UNITY_SPECCUBE_BOX_PROJECTION
giInput.boxMax[0] = unity_SpecCube0_BoxMax;
giInput.probePosition[0] = unity_SpecCube0_ProbePosition;
giInput.boxMax[1] = unity_SpecCube1_BoxMax;
giInput.boxMin[1] = unity_SpecCube1_BoxMin;
giInput.probePosition[1] = unity_SpecCube1_ProbePosition;
#endif

LightingStandard_GI(o, giInput, gi);
```

This is a GI-related process. Put the initial value in UnityGIInput and write the GI result calculated by LightnintStandard_GI() to UnityGI.

▼ List 7.21 Calculation of light reflection

```
// call lighting function to output g-buffer
outEmission = LightingStandard_Deferred(o, worldViewDir, gi,
                                        outDiffuse,
                                        outSpecSmoothness,
                                        outNormal);
outDiffuse.a = 1.0;

#ifndef UNITY_HDR_ON
outEmission.rgb = exp2(-outEmission.rgb);
#endif
```

Pass the various calculation results to LightingStandard_Deferred() to calculate the light reflection level and write it to the Emission buffer. In the case of HDR, write after inserting the part compressed by exp.

7.4.5 Shadow geometry shader

It is almost the same as the physical geometry shader. Only the differences will be explained.

▼ List 7.22 Shadow geometry shader

```
int vindex = 0;
int findex = 0;
for (i = 0; i < 5; i++) {
  findex = flagIndex * 16 + 3 * i;
  if (triangleConnectionTable[findex] < 0)
    break;

  for (j = 0; j < 3; j++) {
    vindex = triangleConnectionTable[findex + j];

    float4 ppos = mul(_Matrix, float4(edgeVertices[vindex], 1));

    float3 norm;
    norm = UnityObjectToWorldNormal(normalize(edgeNormals[vindex]));

    float4 lpos1 = mul(unity_WorldToObject, ppos);
    o.pos = UnityClipSpaceShadowCasterPos(lpos1,
                                          normalize(
                                            mul(_Matrix,
                                              float4(norm, 0)
                                            )
                                          )
                                         );
    o.pos = UnityApplyLinearShadowBias(o.pos);
    o.hpos = o.pos;

    outStream.Append(o);
  }
  outStream.RestartStrip();
}
```

UnityClipSpaceShadowCasterPos() と UnityApplyLinearShadowBias() Convert the vertex coordinates to the shadow projection coordinates with.

7.4.6 Shadow fragment shader

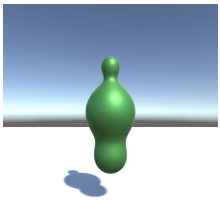
▼ List 7.23 Shadow fragment shader

```
// Shadow fragment shader
fixed4 frag_shadow(g2f_shadow i) : SV_Target
{
  return i.hpos.z / i.hpos.w;
}
```

It's too short to explain. Actually return 0; But the shadow is drawn normally. Is Unity doing a good job inside?

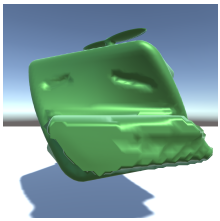
7.5 carry out

When you execute it, a picture like this should appear.



▲図 7.5 Swell

Also, various shapes can be created by combining distance functions.



▲図 7.6　Kaiwarei

7.6 Summary

I used the distance function for simplification this time, but I think that the marching cubes method can be used for other things such as writing 3D texture with volume data and visualizing various 3D data. ..
For gaming purposes, you might be able to make games like ASTORONEER[*2] where you can dig and dig the terrain.
Everyone, try exploring various expressions using the Marching Cubes method!

7.7 reference

- Polygonising a scalar field - http://paulbourke.net/geometry/polygonise/
- modeling with distance functions -

http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[*2] ASTRONEER http://store.steampowered.com/app/361420/ASTRONEER/?l=japanese

第 8 章

3D spatial sampling performed by MCMC

8.1 Introduction

In this chapter, we will explain the sampling method. This time, we will focus on a sampling method called MCMC (Markov chain Monte Carlo method) that samples appropriate values ☒☒ from a certain probability distribution.

The simplest method for sampling from a certain probability distribution is the rejection method. However, sampling in a three-dimensional space causes a large rejected area and cannot be used in actual operation. Therefore, it is the content of this chapter that MCMC can be used for efficient sampling even in high dimensions.

On the one hand, the information about MCMC is for books such as books, which is for statisticians, but it is redundant for programmers, but there is no guide to implementation. The fact is that there is no content to understand the theory and implementation quickly and in a comprehensive manner, as it is only described and there is no care for the theoretical background. I have tried to make the concrete explanations in the following sections as much as possible.

The explanation of the probability that is the background of MCMC is such that it is possible to write one book if strict. This time, with the motto of explaining the minimum theoretical background that can be implemented with peace of mind, the rigor of the definition was moderate, and the aim was to be as intuitive as possible. Mathematics is about the first year of university, and I think that the program can be read without difficulty by those who have used it a little for work.

8.2 Sample repository

In this chapter, Unity Project of Unity Graphics Programming https://github.com/IndieVisualLab/UnityGraphicsProgramming 内にあるAssets/ProceduralModeling 以下をサンプルプログラムとして用意しています。The translated code (comments in English can be found here https://github.com/LIAMPUTCODEHERE) TODO:

8.3 Basic knowledge about probability

To understand the theory of MCMC, it is first necessary to suppress the basic contents of probability. However, there are few concepts that should be held in order to understand MCMC, and there are only the following four. No likelihood or probability density function is needed!

- Random variable
- Probability distribution
- Stochastic process
- Stationary distribution

Let's look at them in order.

8.3.1 Random variable

This real number X when an event occurs at the probability P(X) is called a random variable. For example, when saying "the probability of getting a 5 on the die is 1/6", "5" is the random variable and "1/6" is the probability. In other words, the above sentence can be rephrased as follows: "The probability that an X on the die rolls is P(X)".

By the way, if we write it a bit like a definition, the random variable X is a map X that returns a real number X for the element ω (=one event that occurred) selected from the sample space Ω (= all the events that may occur). $= X(\omega)$ can be written.

8.3.2 Stochastic process

I added a slightly confusing definition in the latter half of the random variable because the assumption that the random variable X is expressed as $X = X(\omega)$

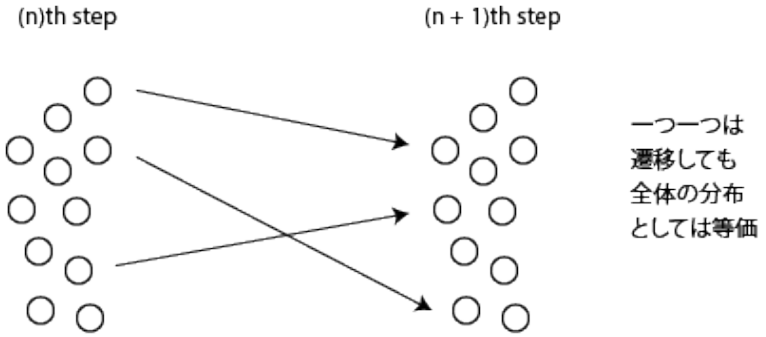
simplifies the understanding of the stochastic process. The stochastic process is the one obtained by adding the time condition to the previous X and can be expressed as X = X(ω, t). In other words, the stochastic process can be considered as a kind of random variable with the condition of time.

8.3.3 Probability distribution

The probability distribution shows the correspondence between the random variable X and the probability P(X). It is often expressed as a graph with probability P(X) on the vertical axis and X on the horizontal axis.

8.3.4 Stationary distribution

Each point is a distribution in which the overall distribution remains unchanged even after a transition. For a distribution P and some transition matrix π, P that satisfies π P = P is called a stationary distribution. This definition is hard to understand, but it's clear from the figure below.



▲図 8.1　stationaryDistribution

8.4 MCMC concept

In this section, I will touch on the concepts that make up MCMC.
As mentioned at the beginning, MCMC is a method of sampling an appropriate

value from a certain probability distribution, but more concretely, the Monte Carlo method It refers to the method of sampling by (Monte Carlo) and Markov chain. In the following, we will explain in order of Monte Carlo method, Markov chain, and stationary distribution.

8.4.1 Monte Carlo method

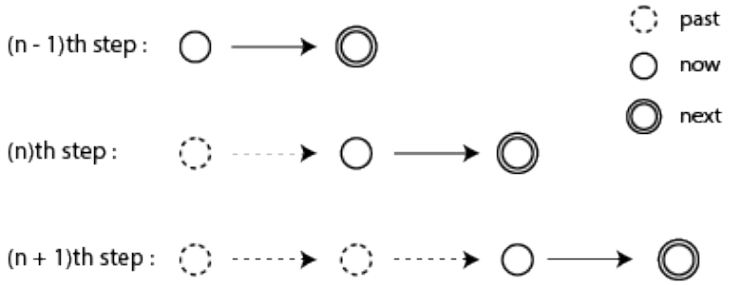
The Monte Carlo method is a general term for numerical calculation and simulation using pseudo random numbers.
An example that is often used to introduce numerical calculations by the Monte Carlo method is the calculation of pi as shown below.

```
float pi;
float trial = 10000;
float count = 0;

for(int i=0; i<trial; i++){
    float x = Random.value;
    float y = Random.value;
    if(x*x+y*y <= 1) count++;
}

pi = 4 * count / trial;
```

In short, the ratio of the number of trials in a fan-shaped circle in a 1 x 1 square to the total number of trials is the area ratio, so the pi can be calculated from that. As a simple example, this is also the Monte Carlo method.

8.4.2 Markov chain

A Markov chain is a stochastic process that satisfies Markovity and whose states can be described discretely.
Markov property is a property in which the probability distribution of future states of a stochastic process depends only on the current state and not on the past states.



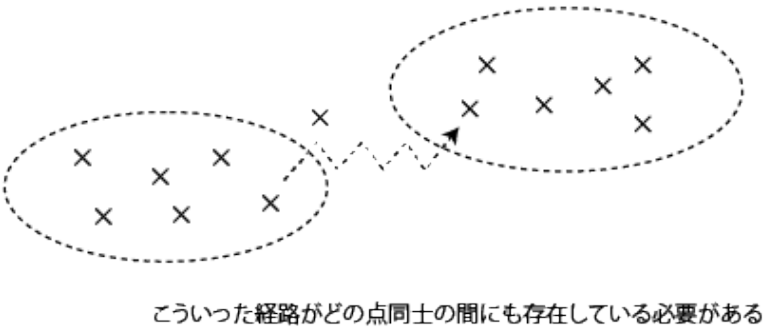
▲図 8.2　MarkovChain

In the Markov chain as shown above, the future state depends only on the present state, and does not directly affect the past state.

8.4.3 Stationary distribution

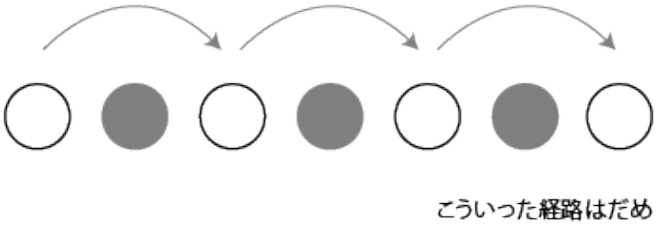
In MCMC, it is necessary to converge from a given distribution using pseudo-random numbers to a given stationary distribution. Because, if it does not converge to the given distribution, it will sample from a different distribution every time, and unless it is a stationary distribution, it will not be able to sample successfully in a chain. In order for an arbitrary distribution to converge to a given distribution, the following two conditions must be met.

- Irreducibility: the condition that the distribution must not be divided into multiple parts. When repeating the transition from a certain point on the probability distribution, there must be no unreachable points



▲図 8.3　Irreducibility

- Non-periodicity: The condition of returning to the original place n times for any n. For example, there should be no condition that only one skip can be made in the distribution lined up on the circumference.



▲図 8.4　Aperiodicity

As long as these two conditions are met, any given distribution can converge to the given stationary distribution. This is called the ergodic property of the

Markov process.

8.4.4 Metropolis method

Now, it is difficult to check whether or not the given distribution satisfies the ergot characteristics mentioned earlier, so in many cases, we will strengthen the condition and investigate within the range of "detailed balance". One of the Markov chain methods that achieves a detailed balance is called the metropolis method.

The Metropolis method performs sampling in the following two steps

1. Select a transition destination candidate x with a pseudo-random number. x is generated according to a distribution Q that satisfies Q(x|x') = Q(x'|x), and this distribution Q is called the proposed distribution. The Gaussian distribution is often chosen as the proposed distribution.
2. A random number independent of 1 is generated, and if a certain criterion is satisfied using the random number, the transition destination candidate is adopted. Specifically, for a uniform random number 0 <= r <1, ☒☒ the ratio P(x')/P(x of the probability value P(x) on the target distribution and the probability value P(x') of the transition candidate) Satisfies P(x')/P(x)> r, transitions to the transition candidate destination.

The merit of the Metropolis method is that even after the transition to the maximum value of the probability distribution, if the value of r is small, the transition is to the smaller probability value, so sampling can be performed in proportion to the maximum value around the maximum value.

By the way, the Metropolis method is a type of Metropolis-Hasting method (MH method). The Metropolis method uses a symmetrical distribution for the proposed distribution, but the MH method does not have this limitation.

8.5 Three-dimensional sampling

Now let's see how to implement MCMC while actually looking at the code excerpt.

First, prepare a three-dimensional probability distribution. This is called the target distribution. This is the “target” distribution because it is the distribution that you want to actually sample.

```
void Prepare()
{
    var sn = new SimplexNoiseGenerator();
    for (int x = 0; x < lEdge; x++)
        for (int y = 0; y < lEdge; y++)
            for (int z = 0; z < lEdge; z++)
            {
                var i = x + lEdge * y + lEdge * lEdge * z;
                var val = sn.noise(x, y, z);
                data[i] = new Vector4(x, y, z, val);
            }
}
```

This time, we used simplex noise as the target distribution.
Next, actually run MCMC.

```
public IEnumerable<Vector3> Sequence(int nInit, int limit, float th)
{
    Reset();

    for (var i = 0; i < nInit; i++)
        Next(th);

    for (var i = 0; i < limit; i++)
    {
        yield return _curr;
        Next(th);
    }
}
```

```
public void Reset()
{
     for (var i = 0; _currDensity <= 0f && i < limitResetLoopCount; i++)
     {
             _curr = new Vector3(
               Scale.x * Random.value,
               Scale.y * Random.value,
               Scale.z * Random.value
               );
             _currDensity = Density(_curr);
     }
}
```

Run the process using a coroutine. MCMC can be conceptually thought of as parallel processing, since processing starts at a completely different place when one Markov chain ends. This time, I use the Reset function to run another process after the series of processes is completed. By doing this, you will be able to perform good sampling even when there are many maximum values ☒☒ of the probability distribution.

Since the first point after the transition is likely to be a point away from the target distribution, this section is discarded without sampling (burn-in). When the target distribution is sufficiently approached, sampling and transition are set a certain number of times, and then another series of processing is started.

Finally, the process of determining the transition.
Since it is three-dimensional, the proposed distribution uses the trivariate standard normal distribution as follows.

```
public static Vector3 GenerateRandomPointStandard()
{
        var x = RandomGenerator.rand_gaussian(0f, 1f);
        var y = RandomGenerator.rand_gaussian(0f, 1f);
        var z = RandomGenerator.rand_gaussian(0f, 1f);
        return new Vector3(x, y, z);
}
```

```
public static float rand_gaussian(float mu, float sigma)
{
     float z = Mathf.Sqrt(-2.0f * Mathf.Log(Random.value))
              * Mathf.Sin(2.0f * Mathf.PI * Random.value);
     return mu + sigma * z;
}
```

The Metropolis method requires a symmetrical distribution, so there is no need to set the mean value to anything other than 0, but if the variance is to be other than 1, use Cholesky decomposition to derive it as follows. I will.

```
public static Vector3 GenerateRandomPoint(Matrix4x4 sigma)
{
    var c00 = sigma.m00 / Mathf.Sqrt(sigma.m00);
    var c10 = sigma.m10 / Mathf.Sqrt(sigma.m00);
    var c20 = sigma.m21 / Mathf.Sqrt(sigma.m00);
    var c11 = Mathf.Sqrt(sigma.m11 - c10 * c10);
    var c21 = (sigma.m21 - c20 * c10) / c11;
    var c22 = Mathf.Sqrt(sigma.m22 - (c20 * c20 + c21 * c21));
    var r1 = RandomGenerator.rand_gaussian(0f, 1f);
    var r2 = RandomGenerator.rand_gaussian(0f, 1f);
    var r3 = RandomGenerator.rand_gaussian(0f, 1f);
    var x = c00 * r1;
    var y = c10 * r1 + c11 * r2;
    var z = c20 * r1 + c21 * r2 + c22 * r3;
    return new Vector3(x, y, z);
}
```

The transition destination is determined by taking the ratio of the probabilities

of the proposed distribution (which is one point above) next and the immediately preceding point _curr on the target distribution, and transitioning if it is greater than a uniform random number, and not transitioning otherwise. I will.
The probability value is approximated because it takes a lot of processing to find the probability value corresponding to the transition destination coordinate (the processing amount of O(n^3)). Since the target distribution uses a distribution that changes continuously this time, the probability value is approximately derived by performing a weighted average that is inversely proportional to the distance.

```
void Next(float threshold)
{
        Vector3 next =
          GaussianDistributionCubic.GenerateRandomPointStandard()
          + _curr;

        var densityNext = Density(next);
        bool flag1 =
          _currDensity <= 0f ||
          Mathf.Min(1f, densityNext / _currDensity) >= Random.value;
        bool flag2 = densityNext > threshold;
        if (flag1 && flag2)
        {
                _curr = next;
                _currDensity = densityNext;
        }
}

float Density(Vector3 pos)
{
        float weight = 0f;
        for (int i = 0; i < weightReferenceloopCount; i++)
        {
                int id = (int)Mathf.Floor(Random.value * (Data.Length - 1));
                Vector3 posi = Data[id];
                float mag = Vector3.SqrMagnitude(pos - posi);
                weight += Mathf.Exp(-mag) * Data[id].w;
        }
        return weight;
}
```

8.6 Other

This time, the repository also contains a sample of the three-dimensional rejection method (a simple Monte Carlo method as shown in the circle example), so you can compare it. In the rejection method, if the reference value for rejection is set to be strong, the sampling cannot be done well, but MCMC can present

similar sampling results more smoothly. In MCMC, if you narrow the width of the random walk for each step, you can easily reproduce plant and flower colonies because it samples from close spaces in a series of chains.

8.7 references

- Kubo Takuya (2012) Introduction to statistical modeling for data analysis: generalized linear model, hierarchical Bayesian model, MCMC (science of probability and information) Iwanami Shoten
- Olle Haggstrom, Kentaro Nomaguchi (2017) Introduction to Easy MCMC: Finite Markov Chain and Algorithm Kyoritsu Publishing

第 9 章

MultiPlane PerspectiveProjection

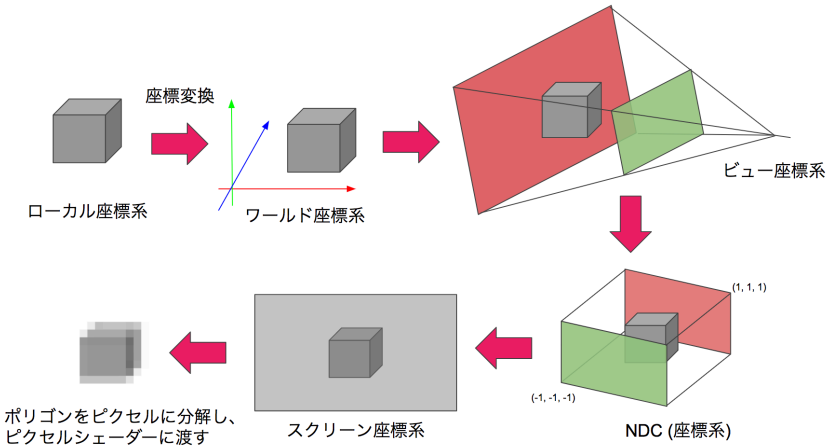
In this chapter, we will introduce an image projection method that allows you to project an image on multiple surfaces such as the walls and floors of a rectangular parallelepiped room with a projector and experience the experience of being in the CG world. In addition, I will explain the processing of the camera in CG and its application examples as the background. Please see the sample project in Assets/RoomProjection in Unity Project [*1] of Unity Graphics Programming. In addition, this content is "Math Seminar December 2016 Issue"[*2] Based on the content contributed to.

9.1 Mechanism of camera in CG

The camera process in general CG is a process of projecting a visible 3D model to a 2D image using perspective projection conversion. The perspective projection transformation is a local coordinate system whose origin is the center of each model, a world coordinate system whose origin is a uniquely determined location in the CG world, a view coordinate system centered on the camera, and a clip coordinate system for clipping (this Is a four-dimensional coordinate system in which w is also meaningful, and a three-dimensional one is called **NDC** (Normalized Device Coordinates) , which represents the two-dimensional position of the output screen. Project the coordinates of the vertices in the order of screen coordinate system.

[*1] LIAM PUT ENGLISH TRANSLATION PROJECT HERE

[*2] https://www.nippyo.co.jp/shop/magazine/7292.html

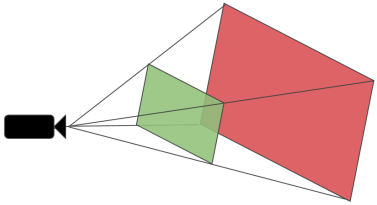


▲図 9.1　Flow of coordinate conversion

Also, since each of these transformations can be represented by one matrix, it is often practiced to multiply the matrices in advance so that several coordinate transformations can be done only once by multiplying the matrix and the vector.

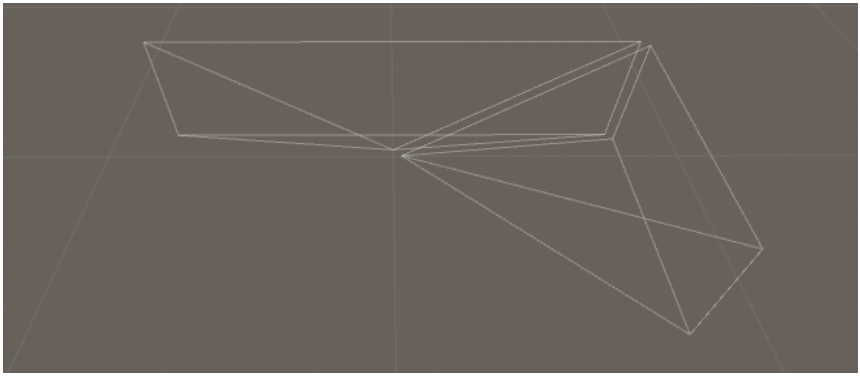
9.2 Perspective consistency with multiple cameras

With a camera in CG, a quadrangular pyramid whose head vertex is located at the camera position and whose bottom is aligned with the camera's orientation is called a viewing frustum, and can be illustrated as a 3D volume representing the projection of the camera.



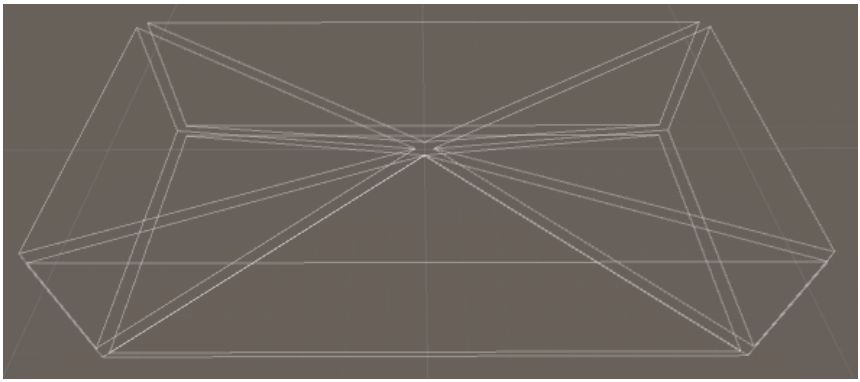
▲図 9.2　視錐台

２つのカメラの視錐台が頭頂点を共有し側面が接していれば、投影面が別々の方向を向いていても映像的には繋がり、かつ、頭頂点から見たときのパースペクティブが一致します。



▲図 9.3　接する視錐台（わかりやすいように少し離して配置しています）

これは視錐台を無数の視線の集合とみなすことで「視線同士が連続している（＝パースペクティブ上矛盾のない映像を投影することができる）」と考えることで理解できます。この考えを５つのカメラまで拡張し、５つの視錐台が頭頂点を共有しそれぞれ隣接する視錐台と接するような配置になるよう画角を調整することで、部屋の各面に対応した映像を生成することができます。理論上は天井も含めた６面も可能ですが今回はプロジェクタの設置スペースとして考え、天井を除く５面を想定しています。



▲図 9.4　部屋に対応した５つの視錐台

この頭頂点、つまり全てのカメラの位置に相当する場所から鑑賞することで、部屋どの方向を見てもパースペクティブ上矛盾のない映像を鑑賞することができます。

9.3　プロジェクション行列の導出

プロジェクション行列（以下 $Proj$ ）はビュー座標系からクリップ座標系へ変換する行列です。

- C：クリップ座標系おける位置ベクトル
- V：をビュー座標系における位置ベクトル

として式で表すと以下のようになります。

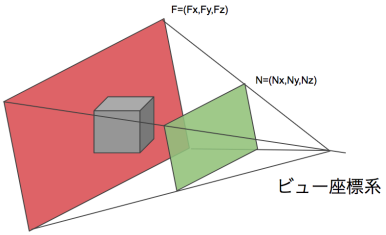
$$C = Proj * V$$

さらに C の各要素を C_w で除算することで NDC での位置座標となります。

$$NDC = (\frac{C_x}{C_w}, \frac{C_y}{C_w}, \frac{C_z}{C_w})$$

なお、 $C_w = -V_z$ とな（るように $Proj$ を作）ります。ビュー座標系の正面方向が Z マイナス方向なのでマイナスがかかっています。NDC では表示範囲を $-1 \leq x, y, z \leq 1$ とし、この変換で V_z に応じて $V_{x,y}$ が拡大縮小することにより遠近法の表現が得られます。

それでは、 $Proj$ をどのように作ればよいか考えてみましょう。ビュー座標系における nearClipPlane の右上の点の座標を N 、farClipPlane の右上の点の座標を F としておきます。



▲図 9.5　N,F

まずは x に注目してみると、

- 投影範囲が $-1 \leq x \leq 1$ になること
- あとで $C_w(=-V_z)$ で除算されること

を考慮すると

$$Proj[0,0] = \frac{N_z}{N_x}$$

とすれば良さそうです。x,z の比率は変わらないので $Proj[0][0] = \frac{F_z}{F_x}$ など視錐台の右端ならどの x,z でも構いません。

同様に

$$Proj[1,1] = \frac{N_z}{N_y}$$

も求まります。

z については少し工夫が必要です。$Proj * V$ で z に関わる計算は以下ようになります。

$$C_z = Proj[2,2] * V_z + Proj[2,3] * V_w \quad (\text{ただし、} V_w = 1)$$

$$NDC_z = \frac{C_z}{C_w} (\text{ただし、} C_w = -V_z)$$

ここで、 $N_z \rightarrow -1, F_z \rightarrow 1$ と変換したいので、 $a = Proj[2,2], b = Proj[2,3]$ と置いて

$$-1 = \frac{1}{N_z}(aN_z + b), \quad 1 = \frac{1}{F_z}(aF_z + b)$$

この連立方程式から解が得られます。

$$Proj[2,2] = a = \frac{F_z + N_z}{F_z - N_z}, \quad Proj[2,3] = b = \frac{-2F_zN_z}{F_z - N_z}$$

また、 $C_w = -V_w$ となるようにしたいので

$$Proj[3,2] = -1$$

とします。
したがって求める $Proj$ は以下の形になります。

$$Proj = \begin{pmatrix} \frac{N_z}{N_x} & 0 & 0 & 0 \\ 0 & \frac{N_z}{N_y} & 0 & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

9.3.1 Camera.projectionMatrix の罠

シェーダー内などでプロジェクション行列を扱ったことがある方の中にはもしかしたらここまでの内容に違和感を持つ方もいらっしゃるかもしれません。実は Unity のプロジェクション行列の扱いはややこしく、ここまでの内容は Camera.projectionMatrix についての解説になります。この値はプラットフォームによらず OpenGL に準拠しています[*3]。 $-1 \leq NDC_z \leq 1$ や $C_w = -V_w$ となるのもこのためです。

しかし Unity 内でシェーダーに渡す際にプラットフォームに依存した形に変換するため、Camera.projectionMatrix をそのまま透視投影変換に使っているとは限りません。とくに NDC_z の範囲や向き（つまり Z バッファの扱い）は多様でひっかかりやすいポイントになっています[*4]。

9.4 視錐台の操作

9.4.1 投影面のサイズ合わせ

視錐台の底面つまり投影面の形はカメラの **fov**（fieldOfView, 画角）と **aspect**（アスペクト比）に依存しています。Unity のカメラでは画角は Inspector で公開されているものの、アスペクト比は公開されていないのでコードから編集する必要がありま

[*3] https://docs.unity3d.com/ScriptReference/GL.GetGPUProjectionMatrix.html
[*4] https://docs.unity3d.com/Manual/SL-PlatformDifferences.html

す。**faceSize**（部屋の面のサイズ）、**distance**（視点から面までの距離）から画角とアスペクト比を求めるコードは以下のようになります。

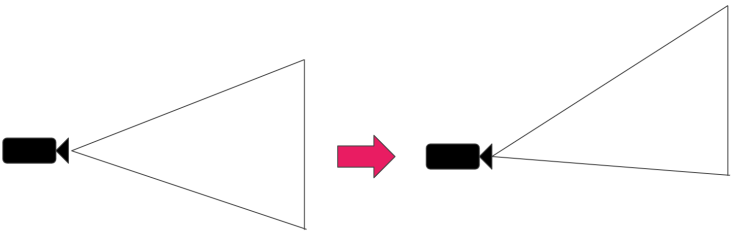
▼ List 9.1　画角とアスペクト比を求める

```
camera.aspect = faceSize.x / faceSize.y;
camera.fieldOfView = 2f * Mathf.Atan2(faceSize.y * 0.5f, distance)
                    * Mathf.Rad2Deg;
```

Mathf.Atan2() で fov の半分の角度を radian で求め、2 倍し、Camera.fieldOfView に代入するため degree に直している点に注意して下さい。

9.4.2　投影面の移動（レンズシフト）

視点が部屋の中心にない場合も考慮してみましょう。視点に対して投影面が上下左右に平行移動することができれば、投影面に対して視点が移動したことと同じ効果が得られます。これは現実世界ではプロジェクターなどで映像の投影位置を調整する**レンズシフト**という機能に相当します。



▲図 9.6　レンズシフト

あらためてカメラが透視投影する仕組みに立ち返ってみるとレンズシフトはどの部分で行う処理になるでしょうか？ プロジェクション行列で NDC に射影する際に、x,y をずらせば良さそうですもう一度 Projection 行列を見てみましょう。

$$Proj = \begin{pmatrix} \frac{N_z}{N_x} & 0 & 0 & 0 \\ 0 & \frac{N_z}{N_y} & 0 & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

C_x, C_y がずれればいいので、行列の平行移動成分である $Proj[0,3], Pro[1,3]$ になにか入れたくなりますが、あとで C_w で除算されることを考慮すると、$Proj[0,2], Pro[1,2]$ に入れるのが正解です。

$$Proj = \begin{pmatrix} \frac{N_z}{N_x} & 0 & LensShift_x & 0 \\ 0 & \frac{N_z}{N_y} & LensShift_y & 0 \\ 0 & 0 & \frac{F_z+N_z}{F_z-N_z} & \frac{-2F_zN_z}{F_z-N_z} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

LensShift の単位は NDC ですので投影面のサイズを-1〜1 に正規化したものになります。コードにすると以下のようになります。

▼ List 9.2　レンズシフトをプロジェクション行列に反映

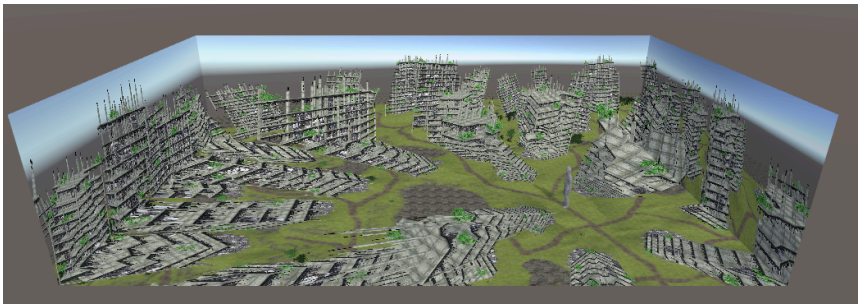
```
var shift = new Vector2(
    positionOffset.x / faceSize.x,
    positionOffset.y / faceSize.y
) * 2f;
var projectionMatrix = camera.projectionMatrix;
projectionMatrix[0,2] = shift.x;
projectionMatrix[1,2] = shift.y;
camera.projectionMatrix = projectionMatrix;
```

一度 Camera.projectionMatrix に set すると Camera.ResetProjectionMatrix() を呼ばない限りその後の Camera.fieldOfView の変更が反映されなくなる点に注意が必要です。[*5]

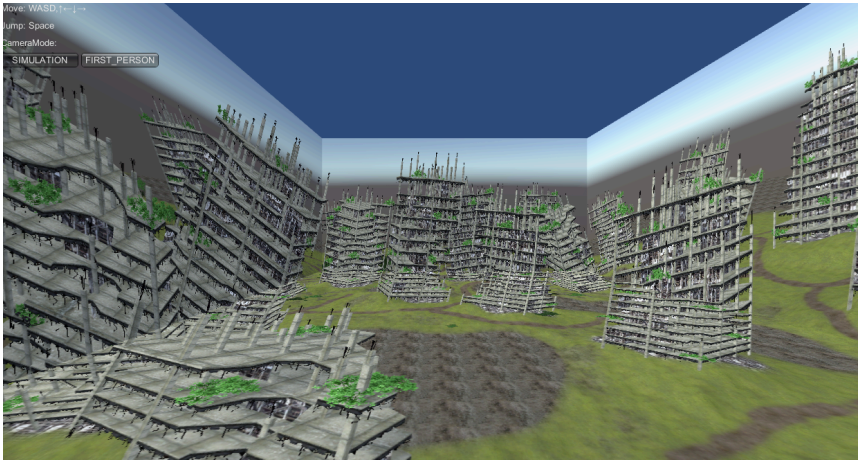
9.5　部屋プロジェクション

直方体の部屋で、鑑賞者の視点位置をトラッキングできているものとします。前節の方法で視錐台の投影面のサイズと平行移動ができるので、視点位置を視錐台の頭頂点、壁面や床面を投影面としたときその形状に合うような視錐台を動的に求める事ができます。各カメラをこのような視錐台にすることによって各投影面用の映像を作ることができます。この映像を実際の部屋に投影すれは鑑賞者からはパースのあった CG 世界が見えるようになります。

[*5] https://docs.unity3d.com/ScriptReference/Camera-projectionMatrix.html



▲図 9.7　部屋のシミュレーション（俯瞰視点）



▲図 9.8　部屋のシミュレーション（一人称視点）

9.6　まとめ

本章ではプロジェクション行列を応用することで複数の投影面でパースを合わせる投影方法を紹介しました。目の前にディスプレイを置くのではなく視界の広い範囲を動的に反応する映像にしてしまう点で、昨今の HMD 型と似て非なるアプローチの VR と言えるのではないかと思います。また、この方法では両眼視差や目のフォーカスを騙せるわけではないのでそのままでは立体視できずに「壁に投影された動く絵」に見えてしまう可能性があります。没入感を高めるためにはもう少し工夫する必要が

ありそうです。

- 両眼視差が小さくなるように部屋を大きくして鑑賞者から投影面までの距離を遠くする
- 反射光などで投影面の平面が意識されてしまうことをできるだけ防ぐ
  - 暗めの映像にする
  - 壁や床をできるだけ鏡面反射しない素材にする

なお、同様の手法を立体視と組み合わせる「CAVE」[*6]という仕組みが知られています。

[*6] https://en.wikipedia.org/wiki/Cave_automatic_virtual_environment

第 10 章

Introduction of Projection Spray

10.1 Introduction

Hello! It's Hirono Sugi! But unfortunately not.

The deadline was approaching one day, and when I asked, "Is there an article written in Sugicho?", he just said "Ah!", and apparently he was completely forgotten. I've been busy lately, but I'd like to take this opportunity to share his achievements, so I'll send you a copy of this easily.

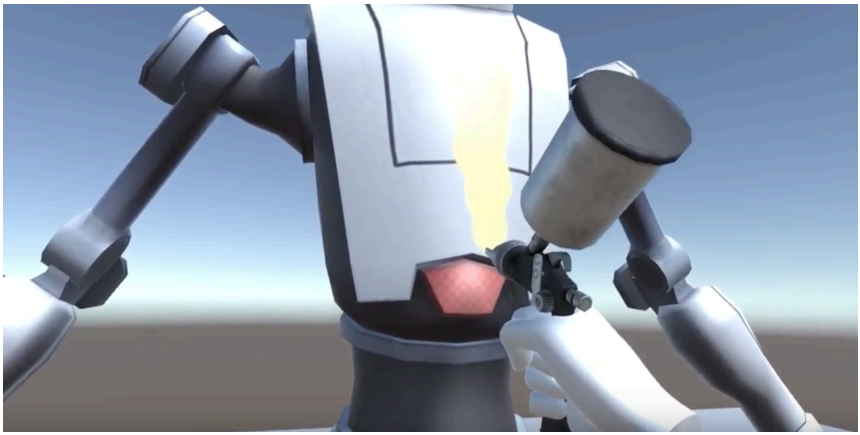
10.1.1 ProjectionSpray

Sugicchi actively publishes his work on Github, and here I personally find it interesting.

https://github.com/sugi-cho/ProjectionSpray

You can add color by spraying a 3D model.

Demo image



▲図 10.1　Demo image 1

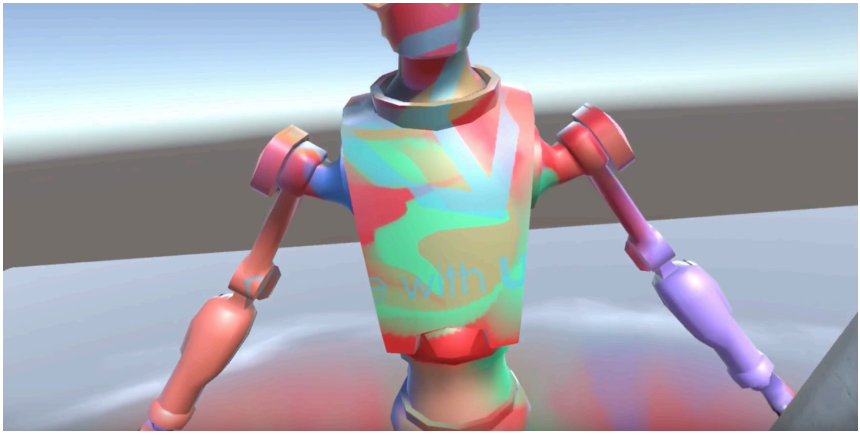
The spray is emitted from the spray device, and the surface of the body is colored.

▲図 10.2　Demo image 2

I feel a mysterious fetishism.

▲図 10.3　Demo image 3

Something like a stencil!



▲図 10.4　Demo image 4

Unity !

10.2 Summary

If there is next time, I would like to request detailed explanation.

The colleague Nakata-san's repository, where Sugichicho said that he felt something similar to himself at the time of the interview, is recommended because it has a lot of useful and excellent code for Unity.

https://github.com/nobnak

excuse me.

(。･ˇ_ˇ･。)

About the authors

Translation by Liam Walsh @evilliam who speaks absolutely no Japanese and is at best a mediocre graphics programmer.

I am Creative Technology Director at Nexus Studios where I have become reacquainted with Unity after many years with native iOS, openframeworks, Three.JS and Cinder

Chapter 1 Beginning with Procedural Modeling in Unity - Masatatsu Nakamura / @mattatz

I try to hit all the balls that come, such as installations, signage, Web (front end/back end), and smartphone apps.

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

Chapter 2 Getting Started with ComputeShader - @XJINE

While living only with the momentum and atmosphere, I suddenly became an interactive artist and engineer, and it became very difficult. I am doing it while being studied while being helped by the people around me.

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

Chapter 3 GPU Implementation of Swarm Simulation (Flocking) - 大石啓明 (Hiroaki Oishi) / @irishoak

Interaction engineer. In the field of video expression such as installation, signage, stage production, MV, concert video, VJ, etc., we are producing contents that take advantage of real-time and procedural characteristics. I have worked

several times with sugi-cho and mattatz as a unit called Aqueduct.

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

Chapter 4 Fluid Simulation by Lattice Method - Yoshiaki Sakoda / @sakope

Former technical developer of game development company. Loves art, design and music, and turned to interactive art. Hobbies are samplers, synths, musical instruments, records, and messing around with equipment. I started Twitter.

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

Chapter 5 Fluid Simulation by SPH Method - 高尾航大 / @kodai100

Interactive artist/engineer and student. He also works in engineering while studying physics simulation of snow at university. Recently, I am having an affair with Touch Designer. Let's talk on twitter.

- https://twitter.com/m1ke_wazowsk1
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

Chapter 6 Growing Grass with Geometry Shaders - @a3geek

An interaction engineer, a programmer of a small fish system, a fluffy gachi, and a shop that makes anything. My favorite classroom is a library or library.

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

Chapter 7 Introduction to the Marching Cubes Method Starting with Atmosphere - @kaiware007

An interactive artist engineer who works in an atmosphere. I like interactive contents more than three times. I like sweet potatoes and I don't eat kaiware. I often post Gene videos on Twitter. Sometimes I do VJ.

- https://twitter.com/kaiware007
- https://github.com/kaiware007
- https://www.instagram.com/kaiware007/

Chapter 8 3D spatial sampling with MCMC - @ komietty

Interactive engineer. I also do web production and graphic design work individually. For production requests, please visit twitter.

- https://github.com/komietty
- https://twitter.com/9_chinashi

Chapter 9 MultiPlanePerspectiveProjection - 福永秀和 / @fuqunaga

Former game developer, current interactive artist engineer. I tried to eat breakfast to keep my health, and for some reason I lost weight by about 2 kg.

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

Chapter 10 Introduction to Projection Spray - Hirono Sugi / @sugi_cho

A person who creates interactive art with Unity. Freelance. hi@sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc

Unity Graphics Programming vol.1

Apr. 22, 2018 技術書典 4 版 v1.1.0
Author IndieVisualLab
Editor IndieVisualLab
Publisher IndieVisualLab