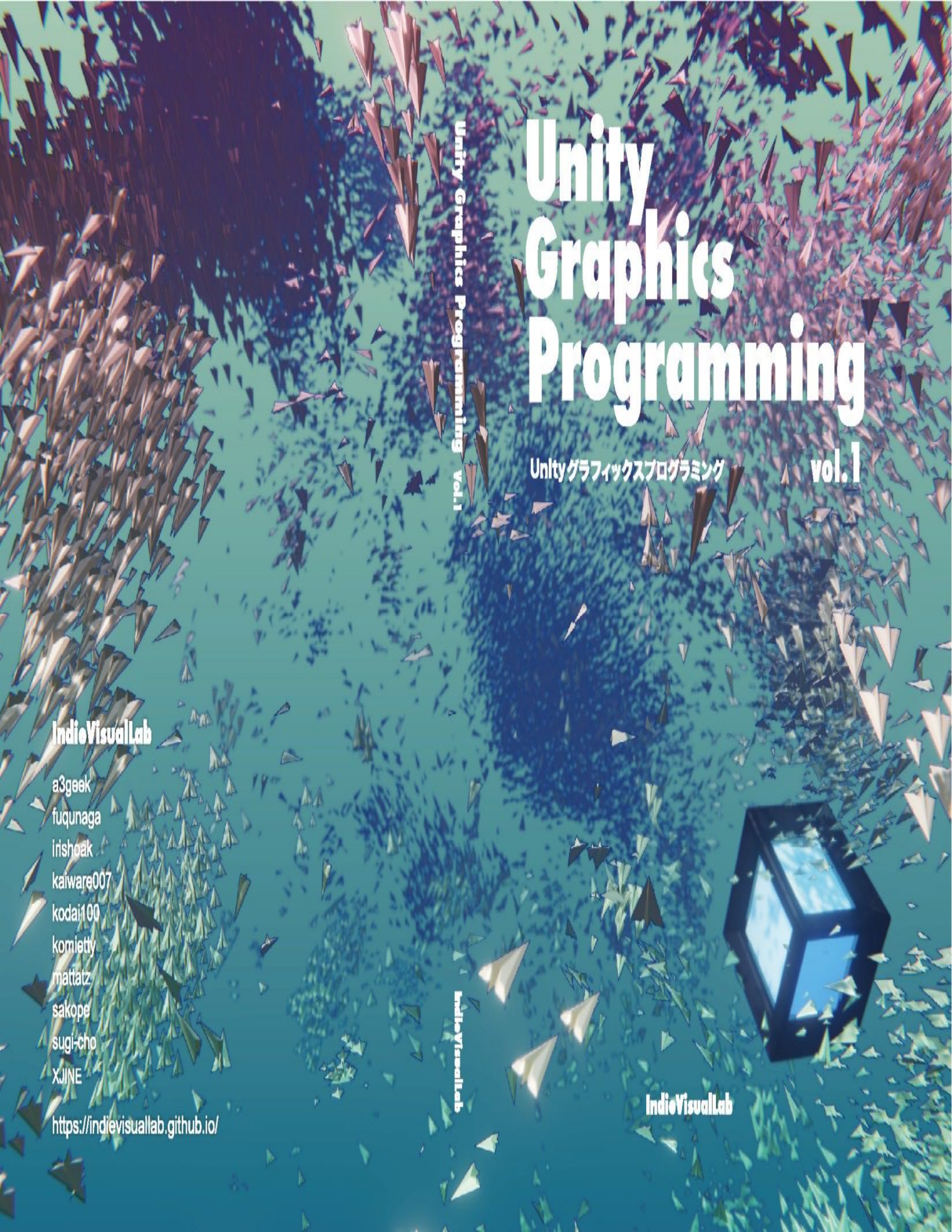


IndieVisualLab

a3geek
fuqunaga
irishoak
kaiware007
kodai100
komietty
mattatz
sakope
sugi-cho
XJINE

https://indievisuallab.github.io/

Unity Graphics Programming vol.1 IndieVisualLab

Unity Graphics Programming

Unityグラフィックスプログラミング

vol.1

IndieVisualLab

Preface

This book is mainly a book that explains the technology related to graphics programming by Unity. Graphics programming is broad in a nutshell, and many books have been published that cover only Shader techniques. This book also contains articles on various topics that the authors are interested in, but the visual results should be easy to see and useful for creating your own effects. In addition, the source code explained in each chapter is available at https://github.com/IndieVisualLab/UnityGraphicsProgramming, so you can read this manual while executing it at hand.

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

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

Requests and impressions about books

If you have any impressions, concerns, or other requests regarding this book (such as wanting to read the explanation about ○○), please feel free to use the Web form (https://docs.google.com/forms/d/e/1FAIpQLSdxeansJvQGTWfZTBN_2RT

uCK_kRqhA6QHTZKVXHCijQnC8zw/ Please let us know via viewform)
or email (lab.indievisual@gmail.com).

Chapter 1 Procedural Modeling Beginning with Unity

1.1 Introduction

Procedural Modeling is a technique for building 3D models using rules. Modeling generally refers to using modeling software such as Blender or 3ds Max to manually operate to obtain the target shape while moving the vertices and line segments. In contrast, the approach of writing rules and obtaining shape as a result of a series of automated processes is called procedural modeling.

Procedural modeling is applied in various fields. For example, in games, there are examples of being used for terrain generation, plant modeling, city construction, etc. By using this technology, each time you play, you will be staged. Content design such as changing the structure becomes possible.

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

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

[*2] http://www.grasshopper3d.com/

With procedural modeling, you can:

- Can create parametric structures
- Flexible models can be incorporated into the content

1.1.1 Can create parametric structures

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

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

1.1.2 Flexible models can be incorporated into content

As mentioned above, in fields such as games, there are many examples where procedural modeling is used to generate terrain and trees, and it is generated in real time in the content instead of incorporating what was once written as a model. Sometimes. Using procedural modeling techniques for real-time content, for example, you can create a tree that grows toward the sun at any position, or build a city where buildings line up from the clicked position. It can be realized.

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

If you learn procedural modeling techniques and build models programmatically, you will be able to develop your own modeling tools.

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, and one triangle is defined by three vertices. The official Unity documentation explains how to manage the vertex and triangle data of the model in the Mesh class as follows.

In the Mesh class, all vertices are stored in one array, and each triangle is specified by three integers that are the indexes of the vertex array. The triangles are further collected as an array of integers. This integer is grouped every three from the beginning of the array, so elements 0, 1, and 2 define the first triangle, followed by the second triangles 3, 4, 5. [*3]

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

The model has uv coordinates that represent the coordinates on the texture required for texture mapping to correspond to each vertex, and normal vectors (also called normal) required to calculate the influence of the light source during lighting. Can be included).

Sample repository

In this chapter, the following Assets / ProceduralModeling in the https://github.com/IndieVisualLab/UnityGraphicsProgramming repository are prepared as sample programs.

Since model generation by C # script is the main content of the explanation, we will proceed with the explanation while referring to the C # script under Assets / ProceduralModeling / Scripts.

Execution environment

The sample code in this chapter has been confirmed to work with Unity 5.0 and above.

1.2.1 Quad

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

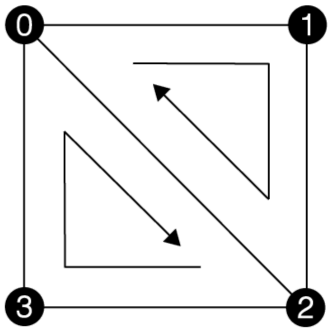


Figure 1.1: Quad model structure Black circles represent the vertices of the model, and the numbers 0 to 3 in the black circles indicate the index of the vertices. Triangles specified in the order of 1,2, lower left is triangles specified in the order of 2,3,0)

Sample program Quad.cs

First, create an instance of the Mesh class.

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

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

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

// Quad vertex data
var vertices = new Vector3[] {
     new Vector3 (-hsize, hsize, 0f), // Upper left position of
the first vertex Quad
     new Vector3 (hsize, hsize, 0f), // Upper right position of
the second vertex Quad
     new Vector3 (hsize, -hsize, 0f), // Lower right position of
the third vertex Quad
     new Vector3 (-hsize, -hsize, 0f) // Lower left position of
the 4th vertex Quad
};

// Quad uv coordinate data
```

```
var uv = new Vector2[] {
    new Vector2 (0f, 0f), // uv coordinates of the first vertex
    new Vector2 (1f, 0f), // uv coordinates of the second vertex
    new Vector2 (1f, 1f), // uv coordinates of the third vertex
    new Vector2 (0f, 1f) // uv coordinates of the 4th vertex
};

// Quad normal data
var normals = new Vector3[] {
    new Vector3 (0f, 0f, -1f), // normal of the first vertex
    new Vector3 (0f, 0f, -1f), // Normal of the second vertex
    new Vector3 (0f, 0f, -1f), // normal of the third vertex
    new Vector3 (0f, 0f, -1f) // Normal of the 4th vertex
};
```

Next, generate triangular data that represents the faces of the model. The triangle data is specified by an array of integers, and each integer corresponds to the index of the vertex array.

```
// Quad face data Recognize as one face (triangle) by arranging
three indexes of vertices
var triangles = new int[] {
    0, 1, 2, // 1st 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, every time you change a model parameter (for example, size that represents width and height in Quad), a new Mesh instance is created and applied to MeshFilter to check the change result immediately. I can. (This function is realized by using the Editor script. ProceduralModelingEditor.cs)

You can also visualize the UV coordinates and normal direction of the model by changing the enum type parameter called ProceduralModelingMaterial.

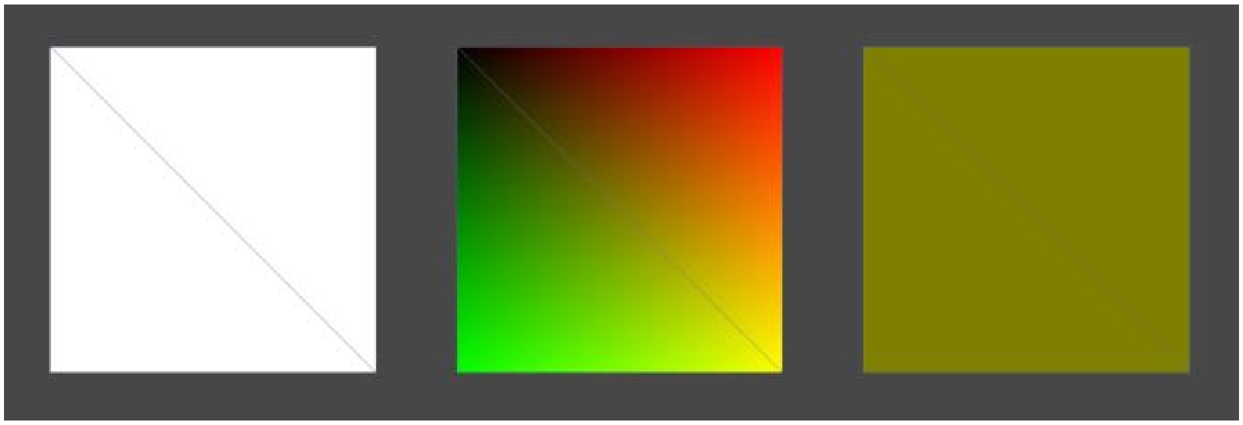


Figure 1.2: From the left, the model to which ProcedureModelingMaterial.Standard, ProcedureModelingMaterial.UV, and ProcedureModelingMaterial.Normal are applied.

1.3 Primitive shape

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

1.3.1 Plane

Plane is shaped like a grid of Quads.

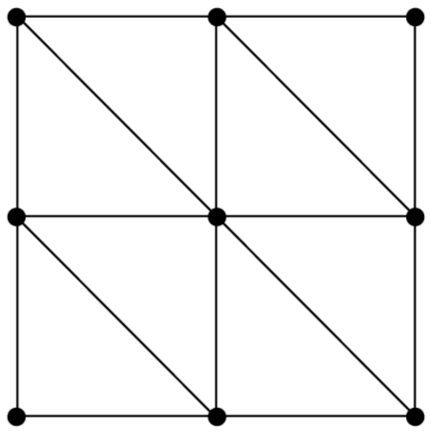


Figure 1.3: Plane model

Determine the number of rows and columns of the grid, place vertices at the intersections of each grid, build a Quad to fill each cell of the grid, and combine them to generate one Plane model.

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

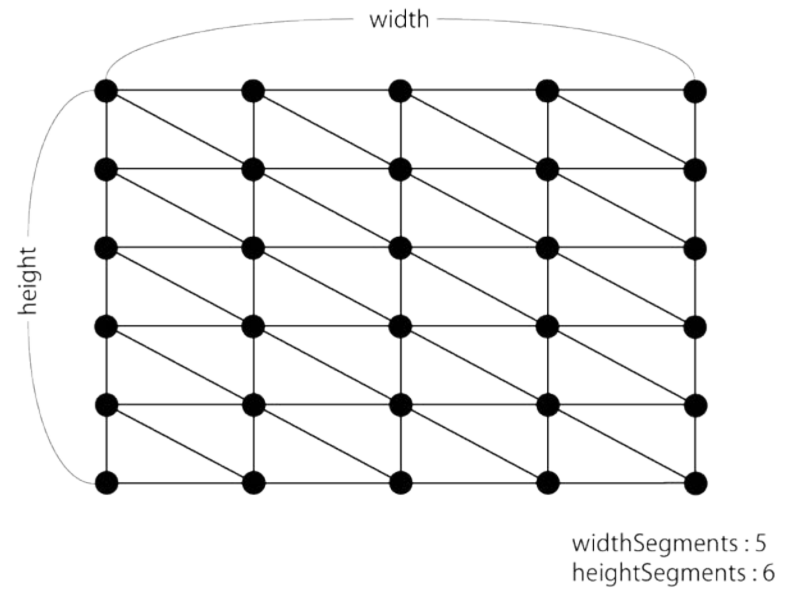


Figure 1.4: Plane parameters

Sample program Plane.cs

First, we will generate vertex data to be placed at the intersections 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
percentage of vertices on the grid (0.0 to 1.0)
var winv = 1f / (widthSegments - 1);
var hinv = 1f / (heightSegments - 1);

for(int y = 0; y < heightSegments; y++) {
    // Row position percentage (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;

        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 shown below in the loop that follows the 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 height (y coordinate) value of each vertex of Plane was set to 0, but by manipulating this height, it is not just a horizontal surface, but an uneven terrain or a shape like a small mountain. Can be obtained.

The ParametricPlaneBase class inherits from 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 to find the height by inputting the uv coordinates of each vertex, and reset the height to flexibly shape it. 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
percentage of vertices on the grid (0.0 to 1.0)
    var winv = 1f / (widthSegments - 1);
    var hinv = 1f / (heightSegments - 1);

    for(int y = 0; y < heightSegments; y++) {
        // Row position percentage (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 normal direction
    mesh.RecalculateNormals();

    return mesh;
}
```

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

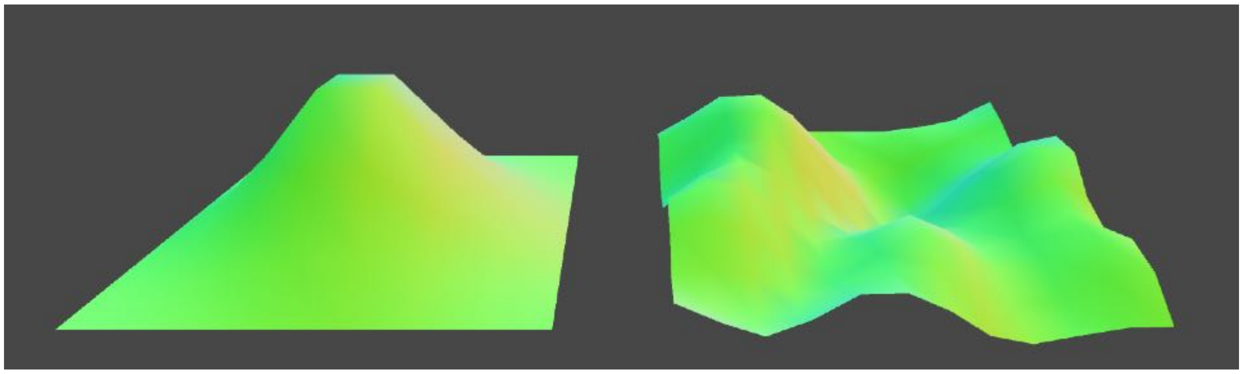


Figure 1.5: ParametricPlane.scene Model generated by the MountainPlane class on the left and the TerrainPlane class on the right

1.3.2 Cylinder

The Cylinder is a cylindrical model that looks like the following figure.

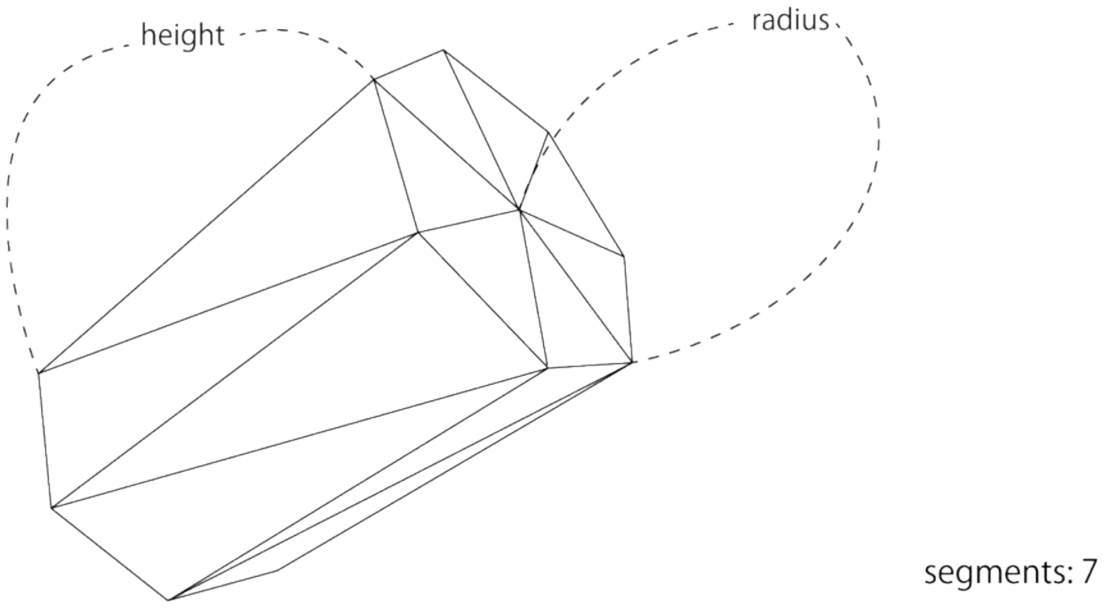


Figure 1.6: Structure of Cylinder

The smoothness of the cylindrical circle can be controlled by the segments, and the vertical length and thickness can be controlled by the height and radius parameters, respectively. As shown in the example above, if you specify 7 for segments, the cylinder will look like a regular heptagon stretched vertically, and the larger the value of segments, the closer it will be to a circle.

Vertices evenly aligned along the circumference

The vertices of the Cylinder should be evenly aligned around the circle located at the end of the cylinder.

Use trigonometric functions (Mathf.Sin, Mathf.Cos) to place evenly aligned vertices along the circumference. The details of trigonometric functions are omitted here, but these functions can be used to obtain the position on the circumference based on the angle.

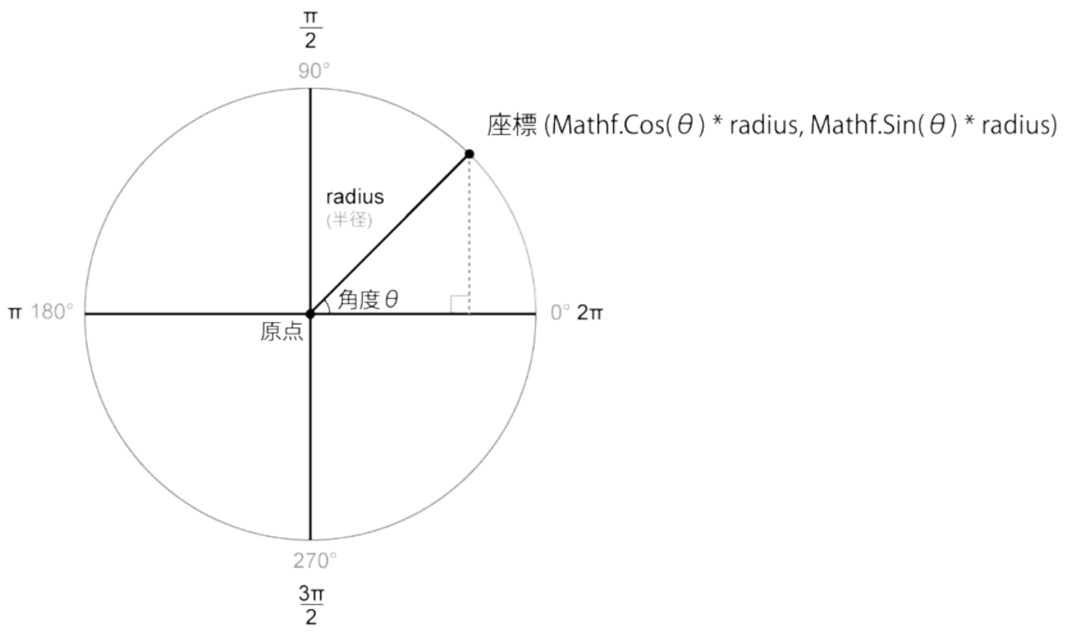


Figure 1.7: Obtaining the position of a point on the circumference from a trigonometric function

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

Based on this, perform the following processing to obtain the vertex positions of segments evenly arranged on the circumference of the radius radius.

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

    // Convert [0.0 ~ 1.0] to [0.0 ~ 2π]
    float rad = ratio * PI2;

    // Get a 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 placed along the circumference of the end of the cylinder, and the vertices are joined together to form a side surface. For each side, just as you would build a Quad, take two corresponding vertices from the top and bottom and place the triangles facing each other to build one side, a rectangle. The sides of the Cylinder can be imagined as the Quads arranged along a circle.

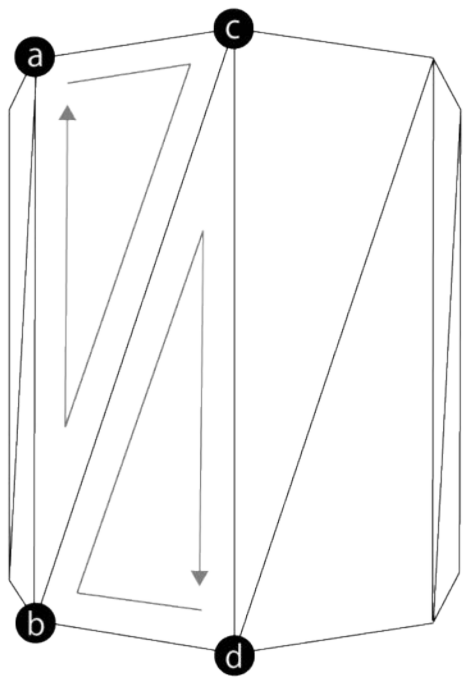


Figure 1.8: Modeling the sides of a cylinder Black circles are evenly distributed vertices along the circumference at the edges a to d in the vertices are index variables assigned to the vertices when constructing a triangle in the Cylinder.cs program.

Sample program Cylinder.cs

First of all, we will build the side, but in the Cylinder class, we have prepared a function GenerateCap to generate the data of the vertices arranged around the circumference located at the upper end and the lower end.

```
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;

// Generate vertex data that makes up the side
GenerateCap(segments + 1, top, bottom, radius, vertices, uvs, 
normals, true);

// To refer to the vertices on the circle when constructing the 
side triangles
// Divine for index to go around the circle
var len = (segments + 1) * 2;

// Build the sides by connecting the top and bottom
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);
}
```

In the GenerateCap function, the vertex and normal data are set in the variable passed as 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 ~ 2π
        float rad = ratio * PI2;

        // Place vertices evenly at the top and bottom along the
circumference
        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));

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

        if(side) {
            // Normal to 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)); // Normals
pointing up the lid
             normals.Add (new Vector3 (0f, -1f, 0f)); // Normals
pointing down the lid
        }
    }
}
```

In the Cylinder class, you can set with the openEnded flag whether to make the model with the top and bottom closed. If you want to close the top and bottom, form 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 side, but create a new vertex at the same position as the side. This is to separate the normals on the sides and the lid for natural lighting. (When constructing the vertex data of the side, specify true in the side variable of the argument of GenerateCap, and when constructing the lid, specify false so that the appropriate normal direction is set.)

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

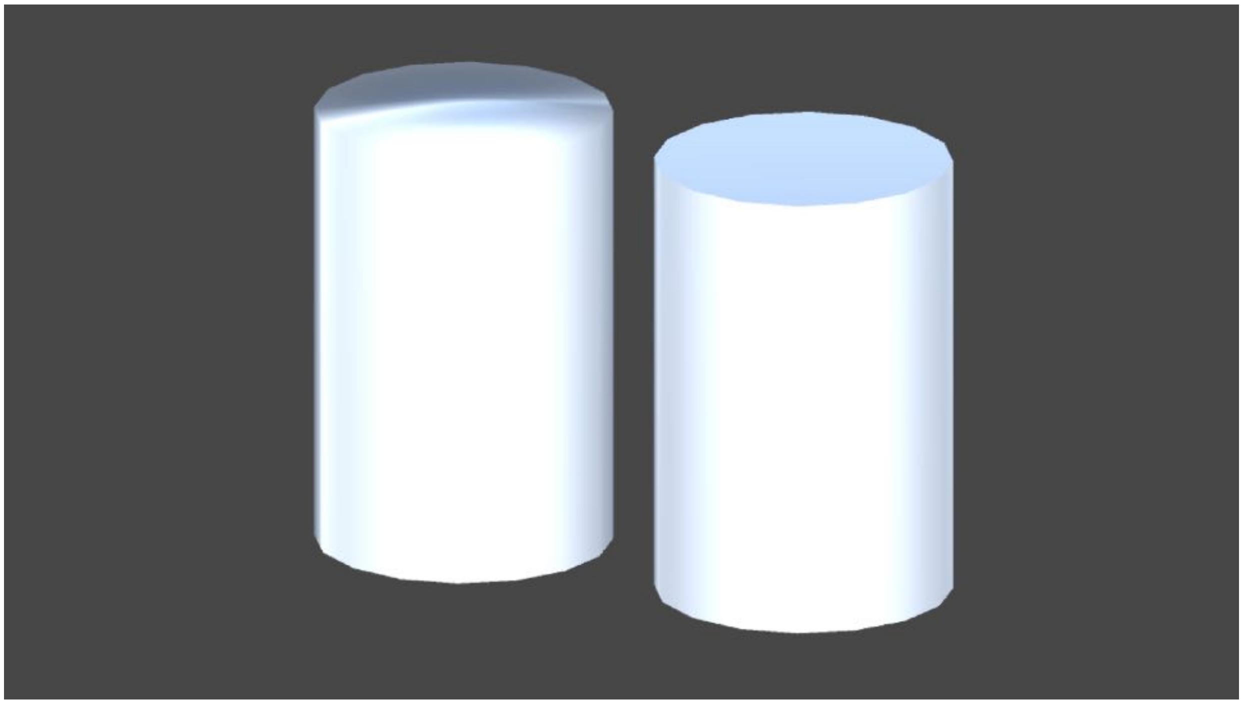


Figure 1.9: When the side of Cylinder and the top of the lid are shared (left: BadCylinder.cs) and when another vertex is prepared as in the sample program (right: Cylinder.cs) The lighting on the left becomes unnatural. ing

To model a circular lid, prepare vertices that are evenly arranged on the circumference (generated from the GenerateCap function) and vertices that are located in the middle of the circle, and the vertices along the circumference from the middle vertex. Join together to form a circular lid by building a triangle that resembles an evenly divided pizza.

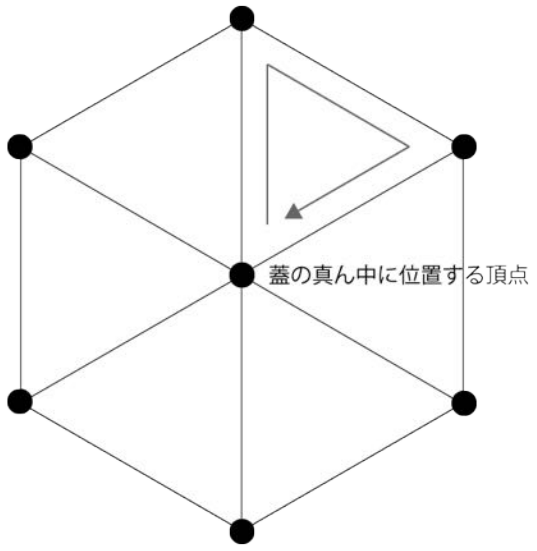


Figure 1.10: Cylinder lid modeling example with segments parameter of 6.

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

    // The 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));

    // The apex in the middle of the 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 to avoid referencing the vertex index for 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.

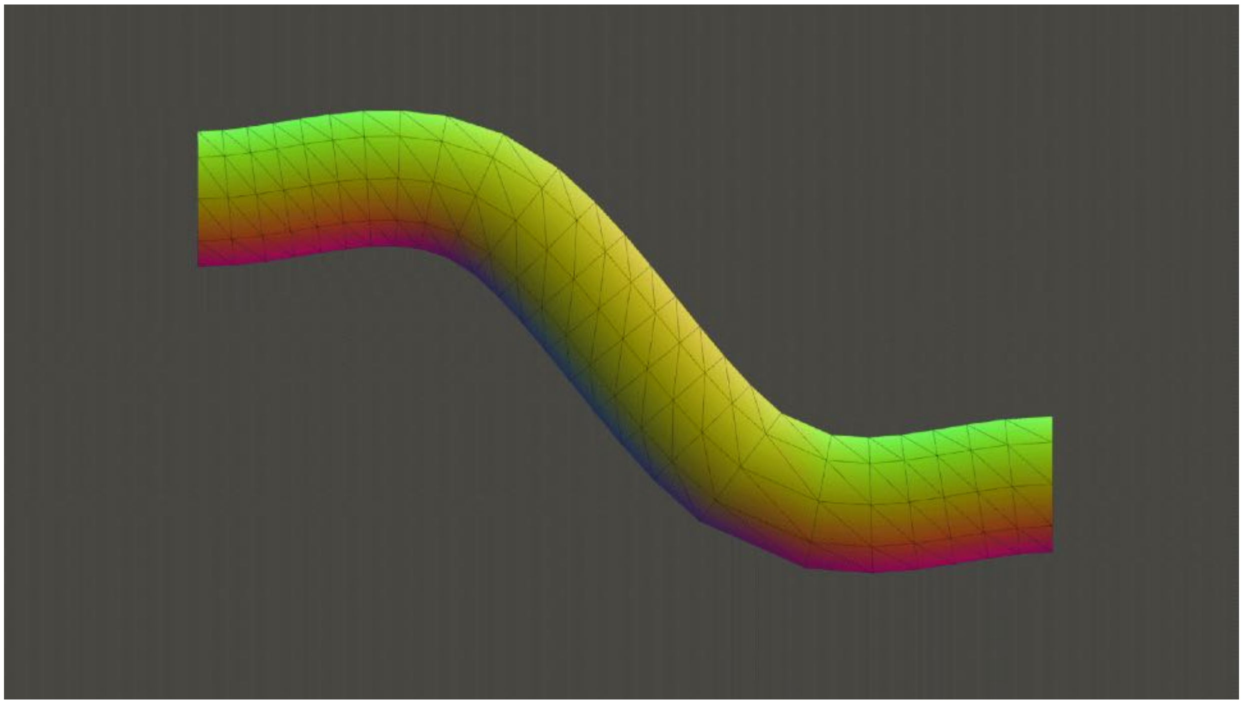


Figure 1.11: Tubular model

The Cylinder model has a straight cylindrical shape, while the Tubular has a curved, untwisted cylinder. In the example of the tree model described later, one branch is represented by Tubular, and a method of constructing one tree by combining them is adopted, but Tubular is used in situations where a tubular shape that bends smoothly is required. I will play an active part.

Cylindrical structure

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

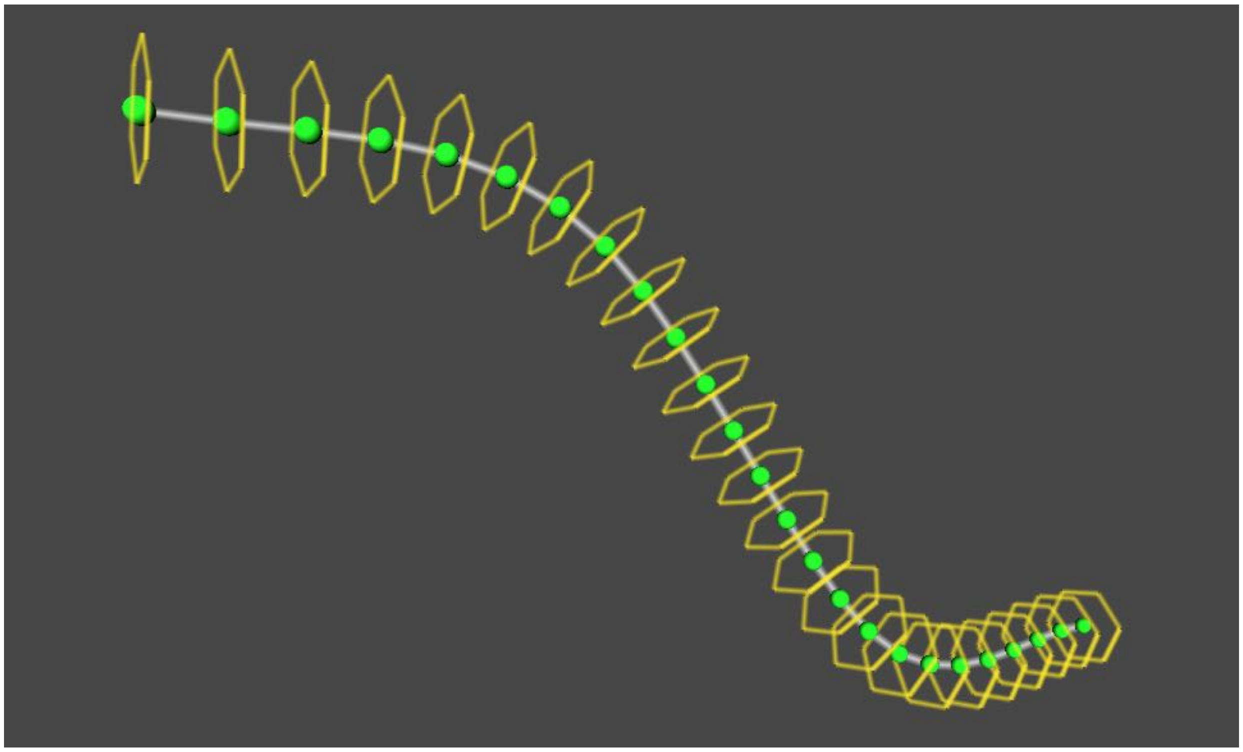


Figure 1.12: Cylindrical structure Tubular visualizes the points that divide the curve along with a sphere and the nodes that make up the sides with a hexagon.

Divide the curve, build sides for each node separated by the division points, and combine them to generate one Tubular model.

The sides of each node are similar to the sides of a cylinder, with the top and bottom vertices of the sides evenly arranged along a circle, and the cylinders are connected along a curve to build them together. You can think of things as Tubular types.

About curves

In the sample program, the base class CurveBase that represents a curve is prepared. Various algorithms have been devised for drawing curves in 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, which inherits the CurveBase class, is used.

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

The 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 a value of [0.0 to 1.0] as an argument. By doing so, you can get the position and slope of the point between the start point (0.0) and the end point (1.0).

Frenet frame

To create a twist-free cylinder along a curve, three orthogonal vectors "tangent vector, normal vector, binormal vector" that change smoothly along the curve You will need an array. The tangent vector is a unit vector that represents the slope at one point on the curve, and the normal vector and the normal vector are obtained as vectors that are orthogonal to each other.

With these orthogonal vectors, you can get "coordinates on the circumference orthogonal to the curve" at a point on the curve.

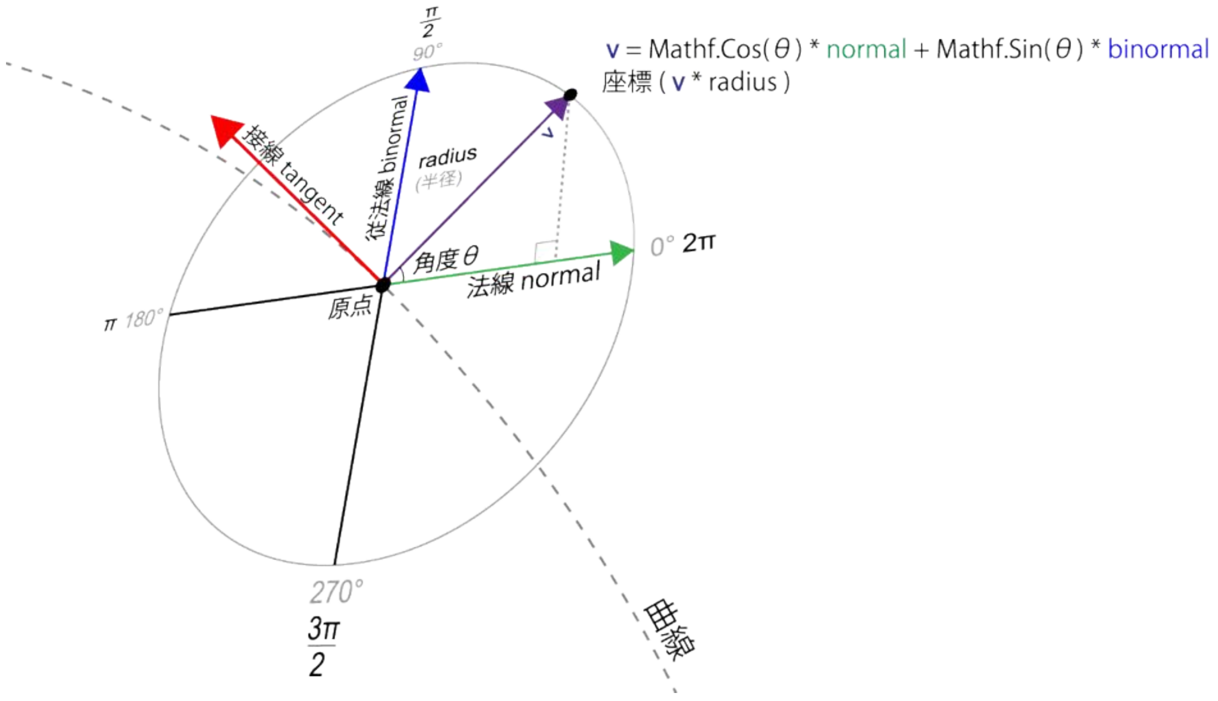


Figure 1.13: Find the unit vector (v) that points to the coordinates on the circumference from the normal and binormal. Multiply this unit vector (v) by the radius radius to make it orthogonal to the curve. You can get the coordinates on the circumference of the radius radius

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

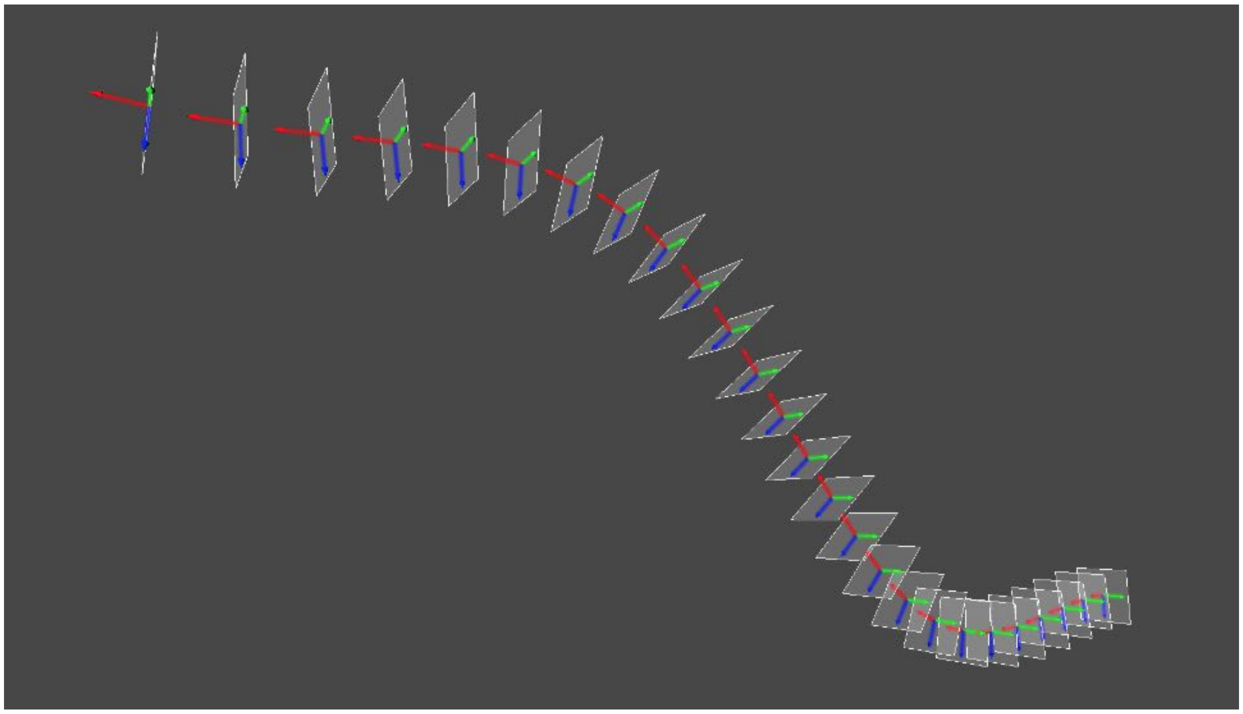


Figure 1.14: Visualization of the Frenet frame array that makes up Tubular The frame represents one Frenet frame, and the three arrows indicate the tangent vector, the normal vector, and the binormal vector.

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

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 drawn by this CatmullRomCurve.

The CatmullRomCurve class requires four or more control points, and when you manipulate the control points, 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 the Frenet frame from the curve
var frames = curve.ComputeFrenetFrames(tubularSegments, closed);

// Generate Tubular vertex data
for(int i = 0; i < tubularSegments; i++) {
     GenerateSegment(curve, frames, vertices, normals, tangents,
i);
}
// Place the last vertex at the start of the curve if you want
to generate a closed cylinder, or at the end of the curve if it
is not closed
GenerateSegment(
    curve,
    frames,
    vertices,
    normals,
    tangents,
    (!closed) ? tubularSegments : 0
);

// Set the uv coordinates from the start point of the curve to
the end point
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 the 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 clause based on the normal and binormal extracted from the Frenet frame mentioned 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 ~ 2π
        float rad = 1f * j / radialSegments * PI2;

        // Arrange the 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 Plants

Plant modeling is often mentioned as an application of the Procedural Modeling technique. The Tree API * 4 for modeling trees in the Editor is also provided in Unity, and there is software dedicated to plant modeling called Speed Tree * 5 .

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

[*5] http://www.speedtree.com/

In this section, we will focus on modeling trees, which are relatively simple modeling methods among plants.

1.4.2 L-System

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

L-System can be used to express the self-similarity found in the shape of plants.

Self-similarity means that when you magnify the shape of the details of an object, it matches the shape of the object as seen on a large scale. For example, when observing the branching of a tree, the branching of the part near the trunk And, there is a similarity in the way the branches are divided near the tip.

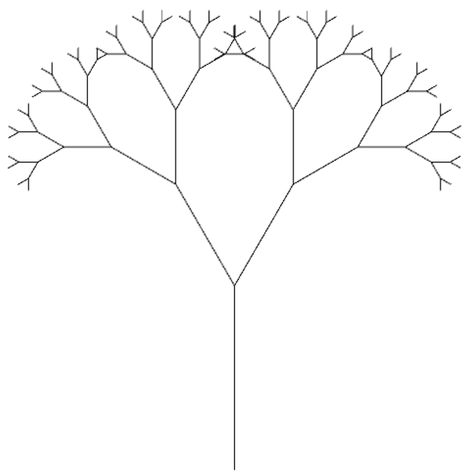


Figure 1.15: A figure in which each branch is branched by changing by 30 degrees. It can be seen that the root part and the branch tip part are similar, but even such a simple figure looks like a tree (Sample program LSystem.scene)

The L-System provides a mechanism for developing complex sequences of symbols by representing elements with symbols, defining rules to replace the symbols, and repeatedly applying the rules to the symbols.

For example, to give a simple example

- Initial character string: a

To

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

If you rewrite according to

a -> ab -> aba -> organize -> organize -> ...

Each step produces complex results.

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

In the LSystem class, the following operations

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

Is 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 this, the rule is applied repeatedly a fixed number of times.

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

1.4.3 Sample program ProceduralTree.cs

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

In ProceduralTree, like the LSystem class explained in the previous section, the tree shape is generated by recursively calling the routine "advance branches, branch, and advance branches".

In the LSystem class in the previous section, the simple rule for branching was "branch in two directions, left and right at a fixed angle", but in

ProceduralTree, random numbers are used, and the number of branches and the branching direction have randomness. However, we have set rules so that the branches branch in a complicated manner.

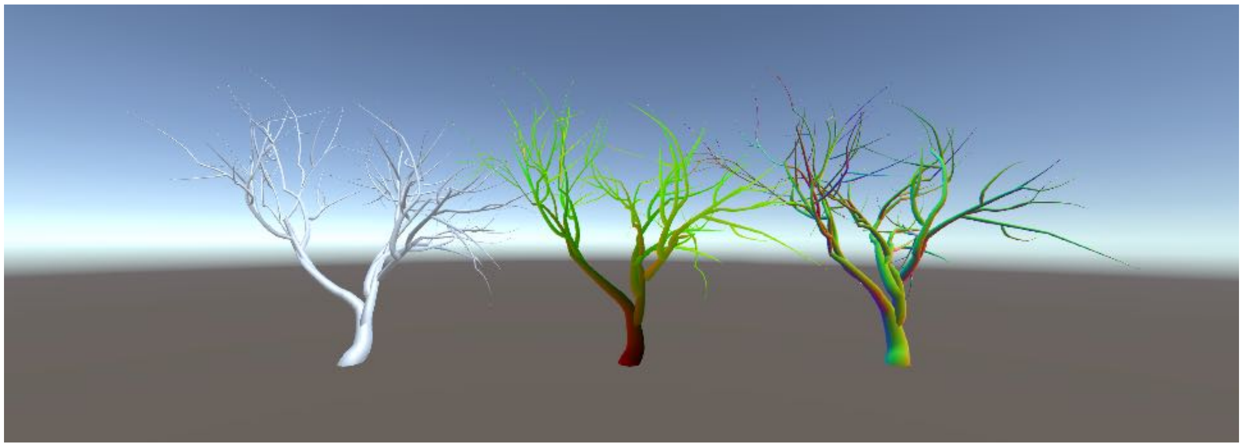


図 1.16: ProceduralTree.scene

TreeData class

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

Branching

Use some parameters in the TreeData class to adjust the degree of branching.

branchesMin, branchesMax

The number of branches branching from one branch is adjusted by the branchesMin / branchesMax parameters. branchesMin represents the minimum number of branches, branchesMax represents the maximum number of branches, and 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 in the branching direction, and growthAngleMax represents the maximum angle. The number between growthAngleMin and growthAngleMax is randomly selected to determine the branching direction.

Each branch has a tangent vector that represents the direction of extension, and a normal vector and a binormal vector as vectors that are orthogonal to it.

The value randomly obtained from the growthAngleMin / growAngleMax parameters is rotated in the direction of the normal vector and the direction of the binormal vector with respect to the tangent vector in the direction extending from the branch point.

By applying a random rotation to the tangent vector in the direction extending from the branch point, the direction in which the branch at the branch destination grows is changed, and the branching is changed in a complicated manner.

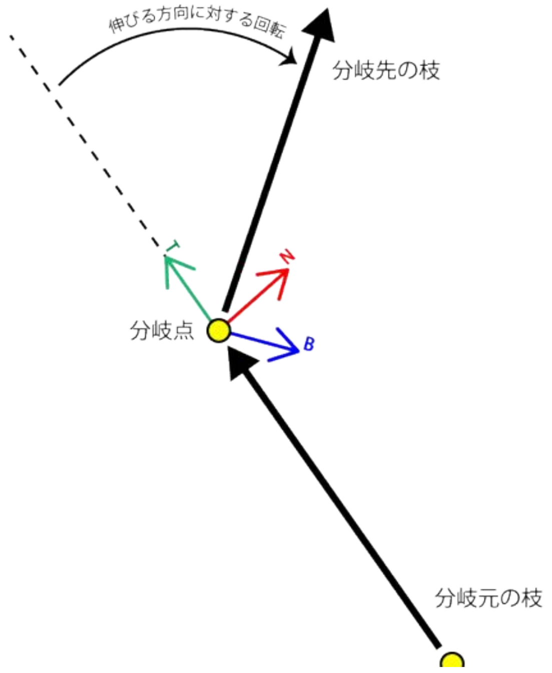


Figure 1.17: Random rotation applied in the direction extending from the branch point The T arrow at the branch point is the extending direction (tangent vector), the N arrow is the normal vector, and the B arrow is the normal line (normal vector). Binormal vector), and random rotation is applied in the direction of the normal and the normal with respect to the extending direction.

The growthAngleScale parameter is provided so that the angle of rotation randomly applied in the direction in which the branch grows increases toward the tip of the branch. This growthAngleScale parameter has a stronger effect on the rotation angle and increases the rotation angle as the generation parameter representing the generation of the branch instance approaches 0, that is, as it approaches the tip of the branch.

```
// The branching angle increases as the branch tip increases
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 the binormal direction
var        qb        =        Quaternion.AngleAxis(scale        *
data.GetRandomGrowthAngle(), binormal);

// Determine the position of the branch tip while rotating qn *
qb in the tangent direction where the branch tip is facing
this.to = from + (qn * qb) * tangent * length;
```

TreeBranch class

Branches are represented by the TreeBranch class.

If you call the constructor with TreeData for setting the branch pattern as an argument in addition to the parameters of the number of generations (generations) and the basic length (length) and thickness (radius), it will recursively internally. An instance of TreeBranch will be created.

A TreeBranch that branches from one TreeBranch is stored in a children variable of type List <TreeBranch> in the original TreeBranch so that all branches can be traced from the root TreeBranch.

TreeSegment class

Like Tubular, the model of one branch divides one curve, models the divided nodes as one Cylinder, and builds them so that they are connected.

The TreeSegment class is a class that expresses a clause that divides a single 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, which Tree Segment is facing,
     // FrenetFrame with vectors normal and binormal orthogonal
to it
    FrenetFrame frame;

    // Position of Tree Segment
    Vector3 position;

    // Tree Segment width (radius)
    float radius;

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

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

Procedural Tree model generation

The model generation logic of Procedural Tree is an application of Tubular, which generates a Tubular model from the array of Tree Segments of one branch Tree Branch and aggregates them into one model to form the whole tree. Modeling with an approach.

```
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 total length of the tree
// Divide the length of the branch by the total length to get
the height of the uv coordinates (uv.y)
// Set to change from the root to the tip of the branch with
[0.0 ~ 1.0]
float maxLength = TraverseMaxLength(root);

// Recursively follow all branches and generate 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 a single 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 ~ 2π
            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));
        }
    }

    // Build a one-branch triangle
    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();
```

Procedural modeling of plants is deep even with trees alone, and methods such as obtaining a model of a natural tree by branching so that the irradiation rate of sunlight is high have been devised.

If you are interested in modeling such plants, please refer to The Algorithmic Beauty of Plants [*6] , which was written by Aristid Lindenmayer, who invented the L-System, for various methods.

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

1.5 Application example of procedural modeling

From the procedural modeling examples introduced so far, we have learned the advantages of the technique of "dynamically generating a model while changing it according to parameters". You may get the impression that it is a technology for improving the efficiency of content development because you can efficiently create models of various variations.

However, like modeling tools and sculpting tools out there, procedural modeling techniques can also be applied to "interactively generate models in response to user input."

As an application example, we will introduce "Teddy," a technology that generates a three-dimensional model from contour lines created by handwritten sketches, devised by Takeo Igarashi of the Department of Computer Science, the University of Tokyo.

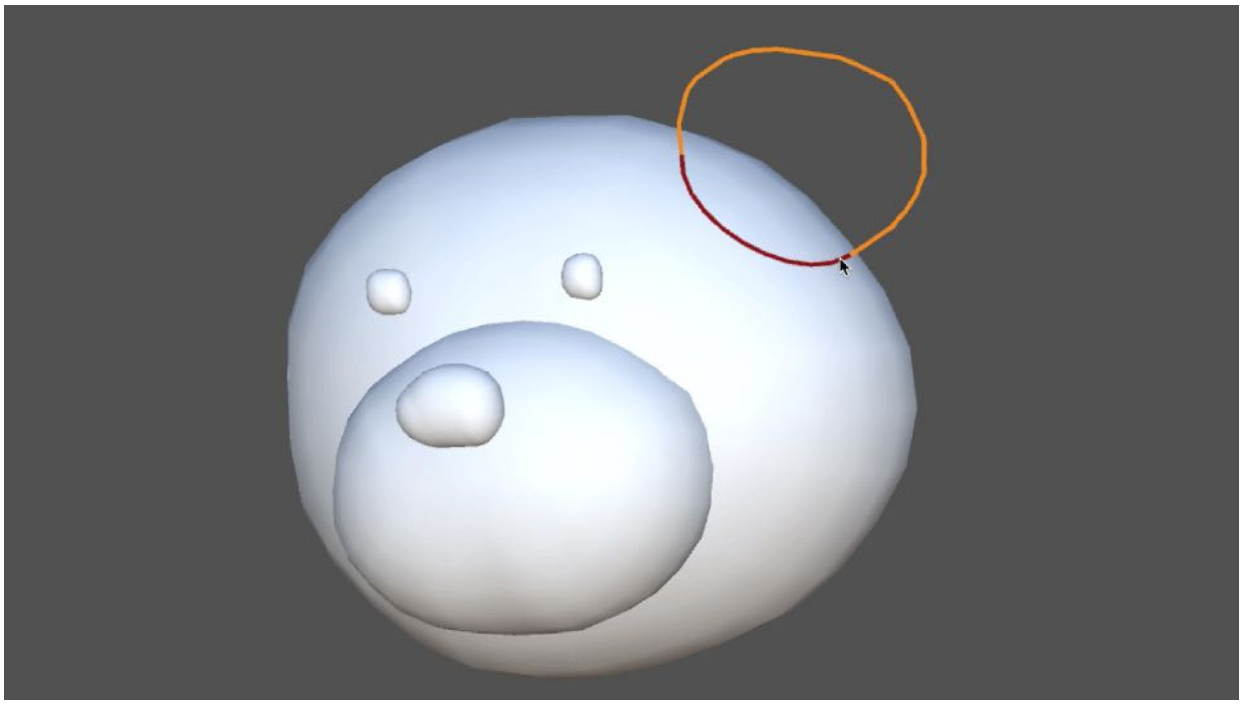


Figure 1.18: Unity assets of "Teddy", a technology for 3D modeling by hand-drawn sketches http://uniteddy.info/ja

This technology was actually used in the game "Junk Masterpiece Theater Rakugaki Kingdom" [* 7], which was released as software for PlayStation 2 in 2002, and it is said that "the picture you drew is converted to 3D and moved as an in-game character". The application has been realized.

[* 7] https://ja.wikipedia.org/wiki/Kingdom of Rakugaki

With this technology

- Define a line drawn on a two-dimensional plane as an outline
- A meshing process called Delaunay Triangulation [* 8] is applied to the point array that constitutes the contour line.
- Apply the algorithm to inflate the mesh on the obtained 2D plane into a solid.

[*8] https://en.wikipedia.org/wiki/Delaunay_triangulation

The 3D model is generated by the procedure. Regarding the details of the algorithm, a paper presented at SIGGRAPH, an international conference dealing with computer graphics, has been published. * 9

[*9] http://www-ui.is.s.u-tokyo.ac.jp/~takeo/papers/siggraph99.pdf

The version of Teddy ported to Unity is published in the Asset Store, so anyone can incorporate this technology into their content. *Ten

[*10] http://uniteddy.info/ja/

By using procedural modeling techniques in this way, it is possible to develop unique modeling tools and create content that develops according to the user's creation.

1.6 Summary

With procedural modeling techniques

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

I have seen that can be achieved.

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

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

In this way, if you think about the fields in which you will use the designed shapes from a broad perspective, you may find various situations where you can apply procedural modeling techniques.

1.7 Reference

- CEDEC2008 Computer automatically generates content with intelligence --What is procedural technology?- Http://news.mynavi.jp/articles/2008/10/08/cedec03/
- The Algorithmic Beauty of Plants - http://algorithmicbotany.org/papers
- nervous system - http://n-e-r-v-o-u-s.com/

Chapter 2 Getting Started with ComputeShader

Here's a simple explanation of how to use ComputeShader (hereafter "Compute Shader" if needed) in Unity. Compute shaders are used to parallelize simple operations using the GPU and perform large numbers of operations at high speed. It also delegates processing to the GPU, but it is different from the normal rendering pipeline. In CG, it is often used to express the movement of a large number of particles.

Some of the content that follows from this chapter uses compute shaders, and knowledge of compute shaders is required to read them.

Here, we use two simple samples to explain what will be the first step in learning a compute shader. These don't cover everything in compute shaders, so be sure to supplement the information as needed.

Although it is called ComputeShader in Unity, similar technologies include OpenCL, DirectCompute, and CUDA. The basic concepts are similar, especially with DirectCompute (DirectX). If you need more detailed information about the concept around the architecture, it is a good idea to collect information about these as well.

The sample in this chapter is "Simple Compute Shader" from https://github.com/IndieVisualLab/UnityGraphicsProgramming .

2.1 Concept of kernel, thread and group

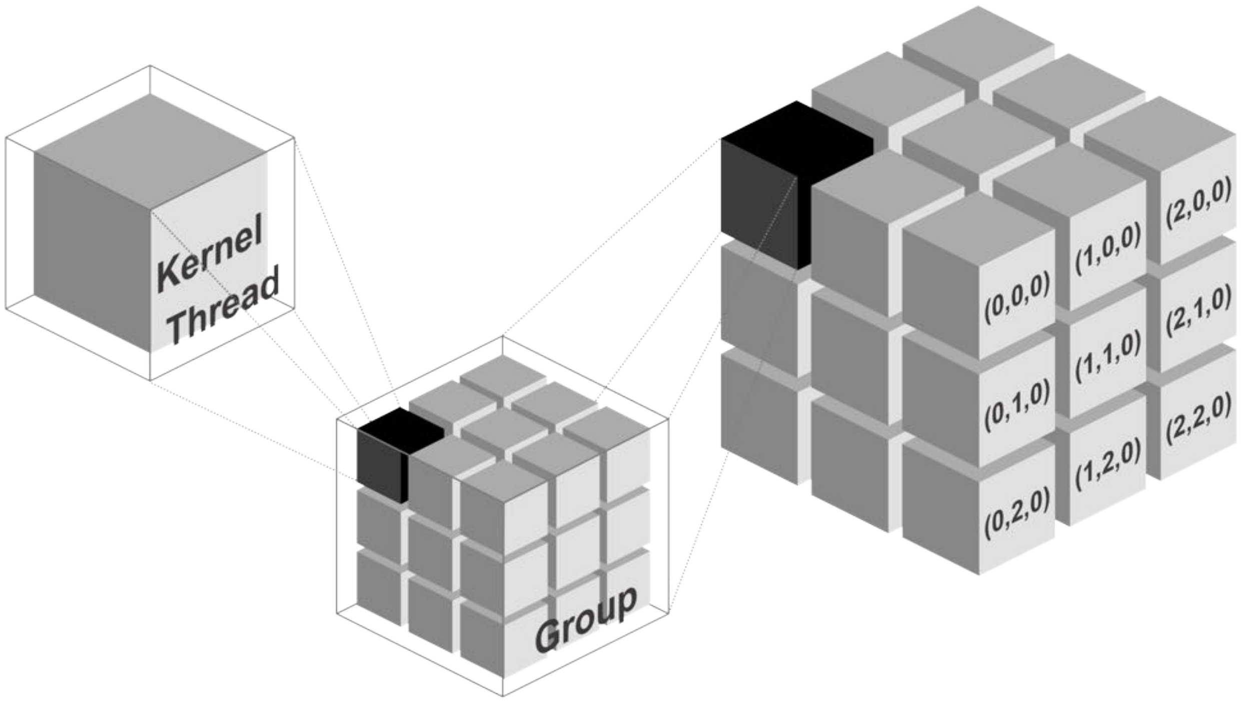


Figure 2.1: Kernel, thread, group image

Before explaining the concrete implementation, it is necessary to explain the concept of **kernel (Kernel)** , **thread (Thread)** , and **group (Group)** handled by the compute shader .

A kernel is a process performed on the GPU and is treated as a function in your code (corresponding to the kernel in general system terms).

A thread is a unit that runs the kernel. One thread runs one kernel. Compute shaders allow the kernel to run in parallel on multiple threads at the same time. Threads are specified in three dimensions (x, y, z).

For example, (4, 1, 1) will execute 4 * 1 * 1 = 4 threads at the same time. If (2, 2, 1), 2 * 2 * 1 = 4 threads will be executed at the same time. The same four threads run, but in some situations it may be more efficient to specify the threads in two dimensions, such as the latter. This will be explained later. For the time being, it is necessary to recognize that the number of threads is specified in three dimensions.

Finally, a **group** is a unit that executes a thread. Also, the threads that a **group** runs are called **group threads** . For example, suppose a group has (4, 1, 1) threads per unit. When there are two of these groups, each group has (4, 1, 1) threads.

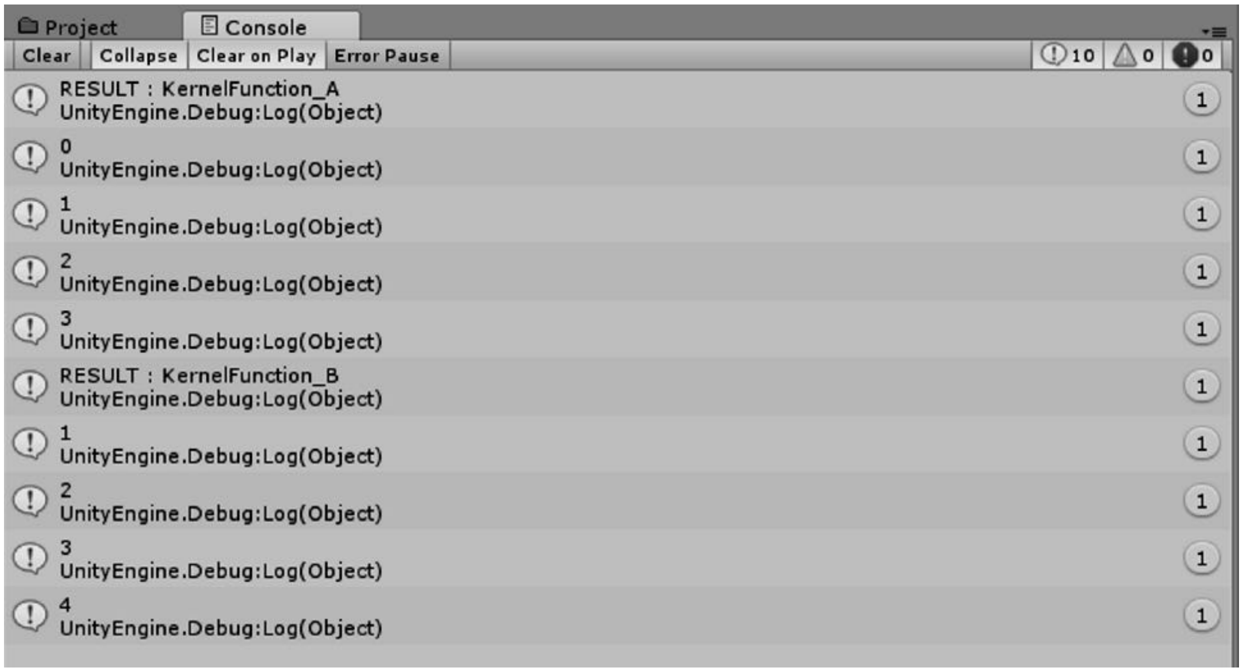
Groups are specified in three dimensions, just like threads. For example, when a (2, 1, 1) group runs a kernel running on (4, 4, 1) threads, the number of groups is 2 * 1 * 1 = 2. Each of these two groups will have 4 * 4 * 1 = 16 threads. Therefore, the total number of threads is 2 * 16 = 32.

2.2 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 results.
- Implement multiple functions in the compute shader and use them properly.
- Pass the value from the script (CPU) to the compute shader (GPU).

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



```
RESULT : KernelFunction_A
UnityEngine.Debug:Log(Object)
0
UnityEngine.Debug:Log(Object)
1
UnityEngine.Debug:Log(Object)
2
UnityEngine.Debug:Log(Object)
3
UnityEngine.Debug:Log(Object)
RESULT : KernelFunction_B
UnityEngine.Debug:Log(Object)
1
UnityEngine.Debug:Log(Object)
2
UnityEngine.Debug:Log(Object)
3
UnityEngine.Debug:Log(Object)
4
UnityEngine.Debug:Log(Object)
```

Figure 2.2: Execution result of sample (1)

2.2.1 Compute shader implementation

From here, I will explain using a sample as an example. It's very short, so it's a good idea to take a look at the compute shader implementation first. The basic configuration is a function definition, a function implementation, a buffer, and variables as needed.

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;
}
```

Features include the **numthreads** attribute and **SV_GroupID** semantics, which will be discussed later.

2.2.2 Kernel definition

As mentioned earlier, aside from the exact definition, the **kernel refers to a single operation performed on the GPU and is treated as a single function in the code.** Multiple kernels can be implemented in one compute shader.

In this example, `KernelFunction_A`there is no kernel and the `KernelFunction_B`function corresponds to the kernel. Also, the function `#pragma kernel`to be treated as a kernel is defined using. This distinguishes it from the kernel and other functions.

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

2.2.3 Preparation of buffers and variables

Create a **buffer area** to store the result of execution by the compute shader . The sample variable `RWStructuredBuffer<int> intBuffer`} corresponds to this.

If you want to give an arbitrary value from the script (CPU) side, prepare a variable in the same way as general CPU programming. In this example, the variable `intValue`corresponds to this, and the value is passed from the script.

2.2.4 Specifying the number of execution threads by numthreads

The numthreads attribute (Attribute) specifies the number of threads that execute the kernel (function). The number of threads is specified by (x, y, z). For example, (4, 1, 1), 4 * 1 * 1 = 4 threads execute the kernel. Besides, (2, 2, 1) runs the kernel in 2 * 2 * 1 = 4 threads. Both are executed in 4 threads, but the difference and proper use will be described later.

2.2.5 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 following the argument value セマンティクスis referred to as, in this example `groupID : SV_GroupID`city `groupThreadID :`

`SV_GroupThreadID`sets the. Semantics are meant to indicate what the value of the argument is and cannot be renamed to any other name.

The argument name (variable name) can be defined freely, but one of the semantics defined when using the compute shader must be set. In other words, it is not possible to implement an argument of any type and refer to it in the kernel, and the arguments that can be referenced in the kernel are selected from the specified limited ones.

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

For example, in a group of (4, 4, 1), when running a thread of (2, 2, 1) `SV_GroupID`, returns a value of (0 ~ 3, 0 ~ 3, 0). `SV_GroupThreadID`Returns a value of (0 ~ 1, 0 ~ 1, 0).

In addition to the semantics set in the sample, there are other `SV_~`semantics that start with and can be used, but I will omit the explanation here. I think it's better to read it after understanding the movement of the compute shader.

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

2.2.6 Kernel (function) processing contents

In the sample, the thread numbers are assigned to the prepared buffers in order. `groupThreadID`Is given the thread number to run in a group. This kernel runs in (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 groups (1, 1, 1) (from the script below). That is, it runs only one group, which contains 4 * 1 * 1 threads. As a result `groupThreadID.x`Please make sure that the value of 0 to 3 is applied to.

* Although is `groupID`not used in this example, the number of groups specified in 3D is given as in the case of threads. Try substituting it and use it to see how the compute shader works.

2.2.7 Run the compute shader from a script

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

- References to Compute Shaders | `comuteShader`
- Kernel index to run | `kernelIndex_KernelFunction_A, B`
- A buffer that stores the execution results of the compute shader | `intComputeBuffer`

SimpleComputeShader_Array.cs

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

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

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

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

2.2.8 Get the index of the kernel to run

In order to run a kernel, you need index information to specify that kernel. The indexes are `#pragma kernel`given as 0, 1… from the top in the order defined by `FindKernel`, but it is better to use the function from the script side .

SimpleComputeShader_Array.cs

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

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

2.2.9 Generate a buffer to save the operation result

Prepare a buffer area to save the calculation result by the compute shader (GPU) on the CPU side. It is `ComputeBuffer`defined as in Unity .

SimpleComputeShader_Array.cs

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

`ComputeBuffer`Initialize by specifying (1) the size of the area to be saved and (2) the size of the data to be saved per unit. Spaces for four int sizes are provided here. This is because the execution result of the compute shader is saved as an int [4]. Resize as needed.

Then, implemented in the compute shader, (1) specify which kernel runs, (2) specify which buffer to use on which GPU, and (3) specify which buffer on the CPU corresponds to. To do.

In this example, the buffer area `KernelFunction_A`(2) referenced when (1) is executed is specified to correspond to `intBuffer`(3) `intComputeBuffer`.

2.2.10 Passing values from scripts to compute shaders

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 and refer to it. Most types of values `ComputeShader.Set~`can be set to variables in the compute shader using. 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, `intValue`we are passing 1.

2.2.11 Running Compute Shader

The kernel implemented (defined) in the compute shader is `ComputeShader.Dispatch`executed by the 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 `ComputeBuffer.GetData`obtained by.

2.2.12 Confirmation of execution result (A)

Check the implementation on the compute shader side again. In this sample, the following kernels are running in 1 * 1 * 1 = 1 groups. The threads are 4 * 1 * 1 = 4 threads. It also `intValue`gives 1 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)`Will contain a value that indicates which thread in the group the kernel is currently running on, so in this example (0 ~ 3, 0, 0) will be entered. Therefore, `groupThreadID.x`is 0 to 3. In other words, `intBuffer[0] = 0` ~ `intBuffer[3] = 3`will be until are executed in parallel.

2.2.13 Run a different kernel (B)

When running different kernels implemented in one compute shader, specify the index of another kernel. In this example, `KernelFunction_A`run after `KernelFunction_B`. Furthermore `KernelFunction_A`, the buffer area used in is `KernelFunction_B`also used.

SimpleComputeShader_Array.cs

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

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

this.intComputeBuffer.GetData(result);

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

2.2.14 Confirmation of execution result (B)

`KernelFunction_B`Executes code similar to the following. This time `intBuffer`is `KernelFunction_A`Please note that specifies continue what we 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`a 20-3 it has been given in the order. Therefore, `KernelFunction_B`after running, make sure the value is between 1 and 4.

2.2.15 Discard the buffer

ComputeBuffers that are no longer in use must be explicitly destroyed.

SimpleComputeShader_Array.cs

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

2.2.16 Problems not solved by sample (1)

The intent of specifying multidimensional threads or groups is not covered in this sample. For example, (4, 1, 1) thread and (2, 2, 1) thread both run 4 threads, but it makes sense to use the two properly. This will be explained in the sample (2) that follows.

2.3 Sample (2): Texture the GPU calculation result

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

- Write information to the texture using a compute shader.

- Make effective use of multidimensional (two-dimensional) threads.

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

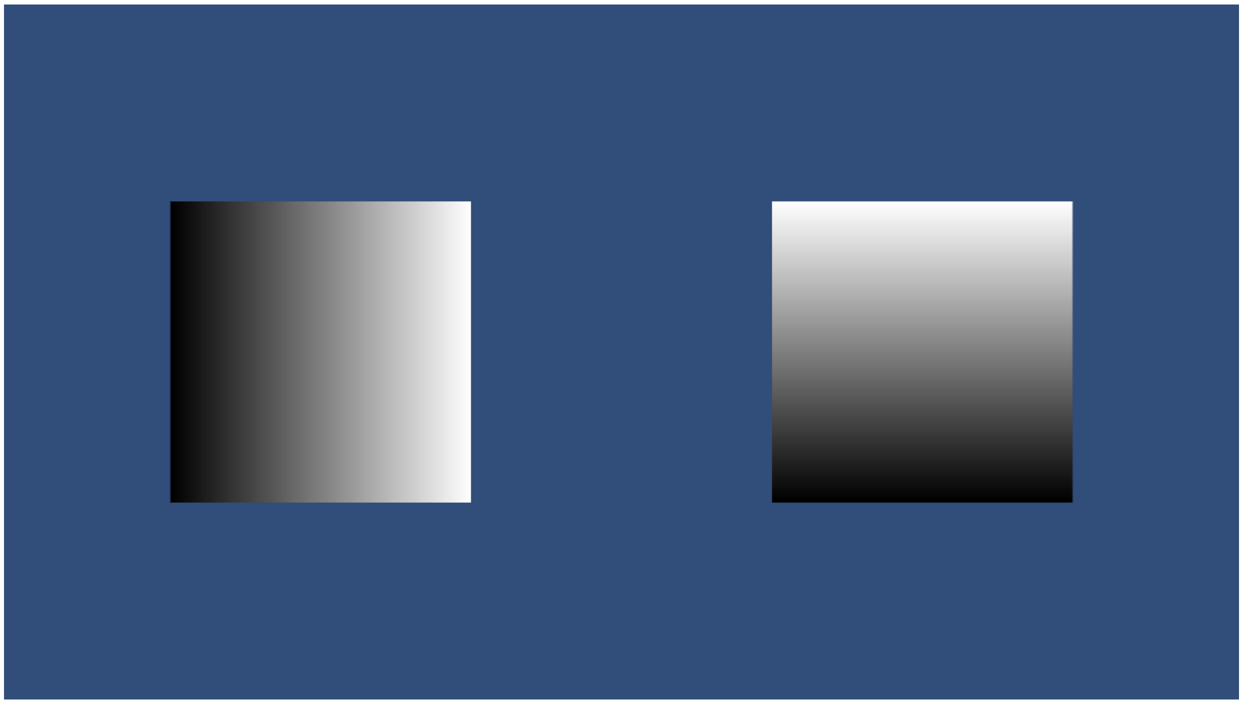


Figure 2.3: Execution result of sample (2)

2.3.1 Kernel implementation

See the sample for the overall implementation. In this sample, the following code is roughly executed in the compute shader. Notice that the kernel runs in multidimensional threads. Since it is (8, 8, 1), it will be executed in 8 * 8 * 1 = 64 threads per group. Another `RWTexture2D<float4>`major change is that the calculation result is saved in.

SimpleComputeShader_Texture.compute

```
RWTexture2D<float4> textureBuffer;

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

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

2.3.2 Special argument SV_DispatchThreadID

`SV_DispatchThradID`No semantics were used in sample (1) . It's a bit complicated, but it shows " **where the thread running a kernel is in all threads (x, y, z)**" .

`SV_DispathThreadID`Is the `SV_Group_ID * numthreads + SV_GroupThreadID`value calculated by. `SV_Group_ID`Indicates a group with (x, y, z), and indicates the `SV_GroupThreadID`threads contained in a group with (x, y, z).

Specific calculation example (1)

For example, suppose you run a kernel in a (2, 2, 1) group that runs on (4, 1, 1) threads. One of the kernels runs on the (2, 0, 0) th thread in the (0, 1, 0) th group. In this case `SV_DispatchThreadID`, (0, 1, 0) * (4, 1, 1) + (2, 0, 0) = (0, 1, 0) + (2, 0, 0) = (2, 1, 0)).

Specific calculation example (2)

Now let's consider the maximum value. In the (2, 2, 1) group, when the kernel runs on the (4, 1, 1) thread, the (3, 0, 0) th thread in the (1, 1, 0) th group Is the last thread. In this case `SV_DispatchThreadID`, (1, 1, 0) * (4, 1, 1) + (3, 0, 0) = (4, 1, 0) + (3, 0, 0) = (7, 1, 0)).

2.3.3 Write a value to a texture (pixel)

After that, it is difficult to explain in chronological order, so please check while reading the entire sample.

Sample (2) `dispatchThreadID.xy`sets groups and threads to show all the pixels on the texture. Since it is the script side that sets the group, we need to look 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, we have prepared a texture of 512x512, but when `dispatchThreadID.x`is 0 ~ 511, it `dispatchThreadID / width`is 0 ~ 0.998…. In other words, as `dispatchThreadID.xy`the value (= pixel coordinates) increases, it will be filled from black to white.

Textures consist of RGBA channels, each set from 0 to 1. When all 0s, it is completely black, and when all 1s, it is completely white.

2.3.4 Preparing textures

The following is an explanation of the implementation on the script side. In sample (1), we prepared an array buffer to store the calculation results of the compute shader. In sample (2), we will 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 by specifying the resolution and format. Note `RenderTexture.enableRandomWrite`that this is enabled to enable writing to the texture.

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

2.3.5 Get the number of threads

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

SimpleComputeShader_Texture.cs

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

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

2.3.6 Kernel execution

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

When thinking about the horizontal direction, the texture resolution is 512 and the number of threads is 8, so the number of horizontal groups 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 we were able to process just all the pixels in parallel.

Execution of different kernels

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

2.3.7 Multidimensional threads, advantages of groups

Multidimensional threads and groups work well when you need multidimensional results, or when you need multidimensional operations, as in sample (2). If sample (2) is to be processed in a one-dimensional thread, the vertical pixel coordinates will need to be calculated arbitrarily.

You can confirm it when you actually implement it, but when you have a stride in image processing, for example, a 512x512 image, the 513th pixel is the (0, 1) coordinate, and so on. ..

It is better to reduce the number of operations, and the complexity increases as the advanced processing is performed. When designing processing with compute shaders, it's a good idea to consider whether you can take advantage of multidimensionality.

2.4 Supplementary information for further learning

In this chapter, we have provided introductory information in the form of explaining samples of compute shaders, but from now on, we will supplement some information necessary for learning.

2.4.1 GPU architecture / basic structure

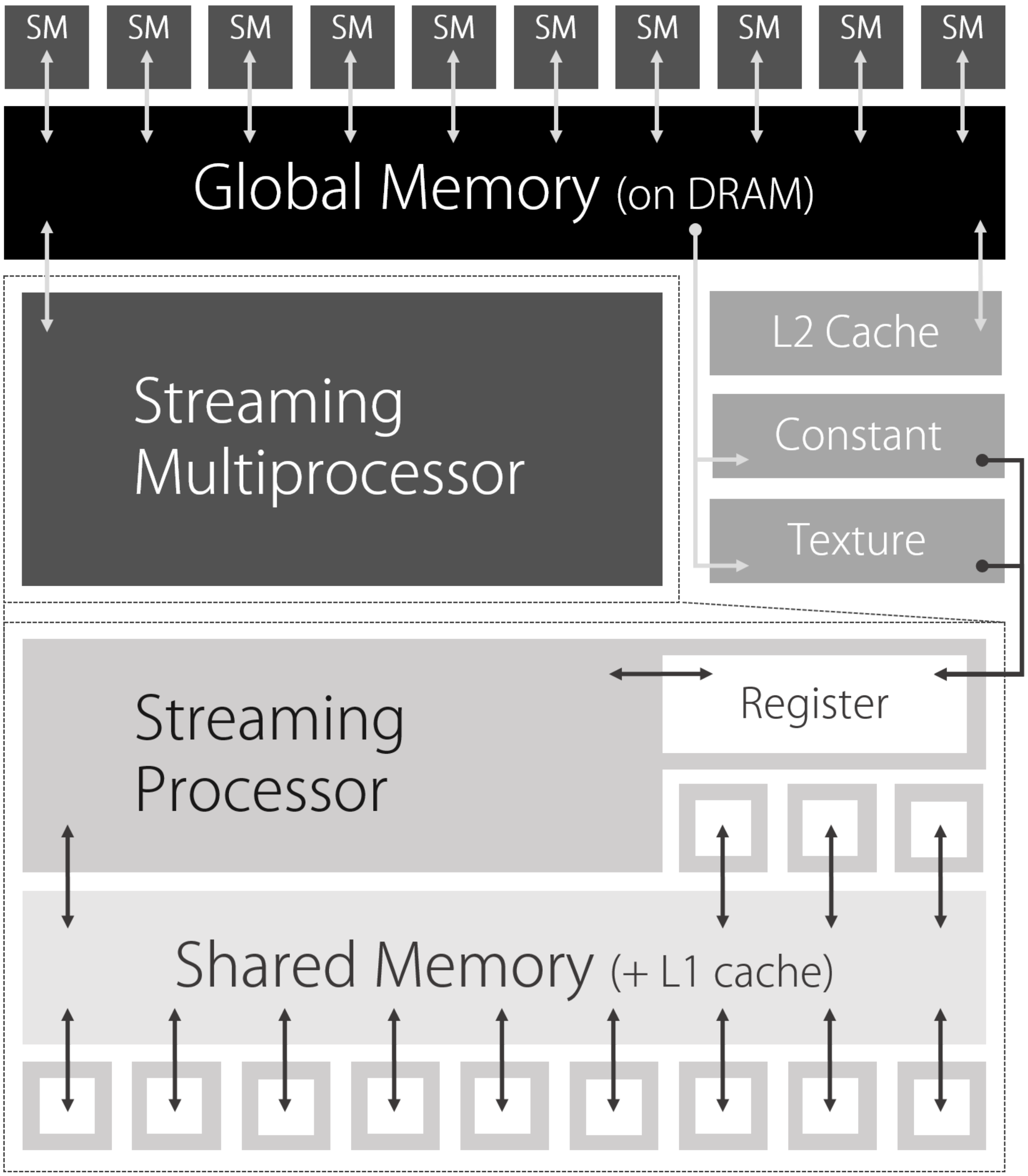


Figure 2.4: Image of GPU architecture

If you have a basic knowledge of GPU architecture and structure, it will be useful for optimizing it when implementing processing using compute shaders, so I will introduce it here a little.

The GPU is equipped with a large number of **Streaming Multiprocessors (SM)** , which are shared and parallelized to execute the given processing.

The SM has multiple smaller **Streaming Processors (SPs)** , and the SP calculates the processing assigned to the SM.

The SM has **registers** and **shared memory,** which allows it to read and write faster than **global memory (memory on DRAM)** . Registers are used for local variables that are referenced only within the function, and shared memory can be referenced and written by all SPs that belong to the same SM.

In other words, it is ideal to know the maximum size and scope of each memory and realize an optimal implementation that can read and write memory at high speed without waste.

For example, shared memory, which you may need to consider most, is `groupshared`defined using storage-class modifiers . Since this is an introduction, I will omit a concrete introduction example, but please remember it as a technique and terminology necessary for optimization and use it for future learning.

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

register

The fastest accessible memory area located closest to the SP. It consists of 4 bytes and contains kernel (function) scope variables. Since each thread is independent, it cannot be shared.

Shared memory

A memory area located in the SM, which is managed together 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 the DRAM, not on the GPU. References are slow because they are far from the processor on the GPU. On the other hand, it has a large capacity and can read and write data from all threads.

Local memory

The memory area on the DRAM, not the GPU, stores data that does not fit in the registers. References are slow because they are far from the processor on the GPU.

Texture memory

This memory is dedicated to texture data and handles global memory exclusively for textures.

Constant memory

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

2.4.2 Tips for specifying the number of threads efficiently

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

2.4.3 Limits on current specifications

Introducing the upper limit of the current specifications at the time of writing. Please note that it may not be the latest version. However, it is required to implement it while considering these restrictions.

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

Number of threads and groups

The limits on the number of threads and groups were not mentioned in the discussion. This is because it changes depending on the shader model (version). It is expected that the number that can be paralleled will continue to increase in the future.

- ShaderModel cs_4_x
  - Maximum value of Z is 1
  - The maximum value of X * Y * Z is 768

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

The group limit is (x, y, z), 65535 each.

Memory area

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

2.5 Reference

Other references in this chapter are:

- 5th GPU Structure --Japan GPU Computing Partnership --http: //www.gdep.jp/page/view/252
- Getting Started with CUDA Starting with Windows --Nvidia Japan --http: //on-demand.gputechconf.com/gtc/2013/jp/sessions/8001.pdf

Chapter 3 GPU implementation of the simulation of the group

3.1 Introduction

In this chapter, we will explain the implementation of group simulation using the Boids algorithm using Compute Shader. Birds, fish and other terrestrial animals sometimes flock. The movements of this group show regularity and complexity, and have a certain beauty and have attracted people. In computer graphics, it is not realistic to control the behavior of each individual by hand, and an algorithm for forming a group called Boids was devised. This simulation algorithm consists 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 as the number of individuals increases, it becomes squared. The amount of calculation will increase proportionally. If you want to control many individuals, it is very difficult to implement with CPU. Therefore, we will take advantage of the powerful parallel computing power of the GPU. Unity provides a shader program called Compute Shader to perform such general purpose computing (GPGPU) by GPU. The GPU has a special storage area called shared memory, which can be used effectively by using Compute Shader. In addition, Unity has an advanced rendering function called GPU instancing, which allows you to draw a large number 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 computing power of these Unity GPUs.

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".

In Reynolds, a herd produces complex behavior as a result of each individual modifying its own behavior based on the position and direction of movement of other individuals around it, through perceptions such as sight and hearing. Pay attention to the fact that there is.

Each individual follows three simple rules of conduct:

1. Separation

Move to avoid crowding with individuals within a certain distance

2. Alignment

Individuals within a certain distance move toward the average in the direction they are facing

3. Cohesion

Move to the average position of an individual within a certain distance

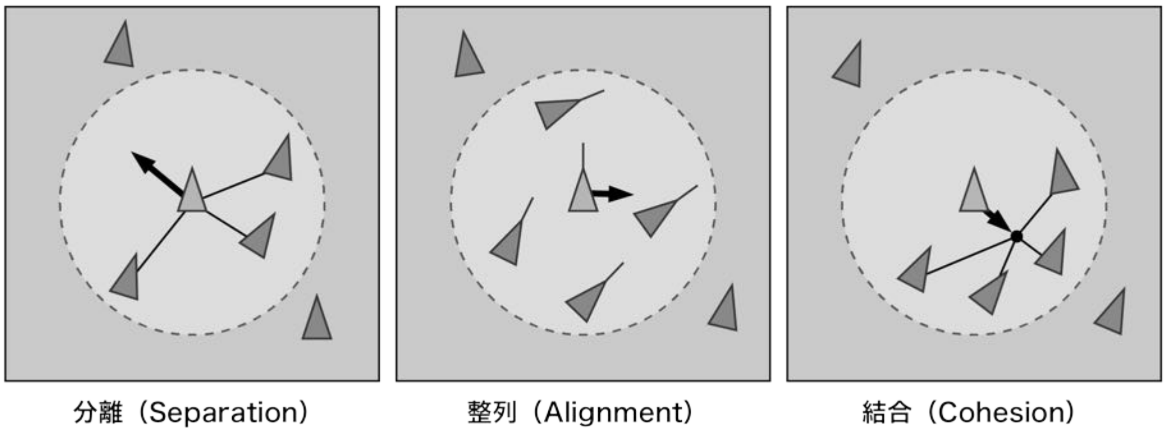


Figure 3.1: Basic rules for Boids

You can program the movement of the herd by controlling the individual movements according to these rules.

3.3 Sample program

3.3.1 Repository

https://github.com/IndieVisualLab/UnityGraphicsProgramming

Open the **BoidsSimulationOnGPU.unity** scene data in the Assets / **BoidsSimulationOnGPU** folder in the sample Unity project in this document .

3.3.2 Execution conditions

The programs introduced in this chapter use Compute Shader and GPU instancing.

ComputeShader runs on the following platforms or APIs:

- Windows and Windows Store apps with DirectX 11 or DirectX 12 graphics API and shader model 5.0 GPUs
- IOS with Mac OS and Metal Graphics API
- Android, Linux and 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 Mac OS and iOS
- Vulkan for Windows and Android
- Playstation 4 and Xbox One
- WebGL (requires WebGL 2.0 API)

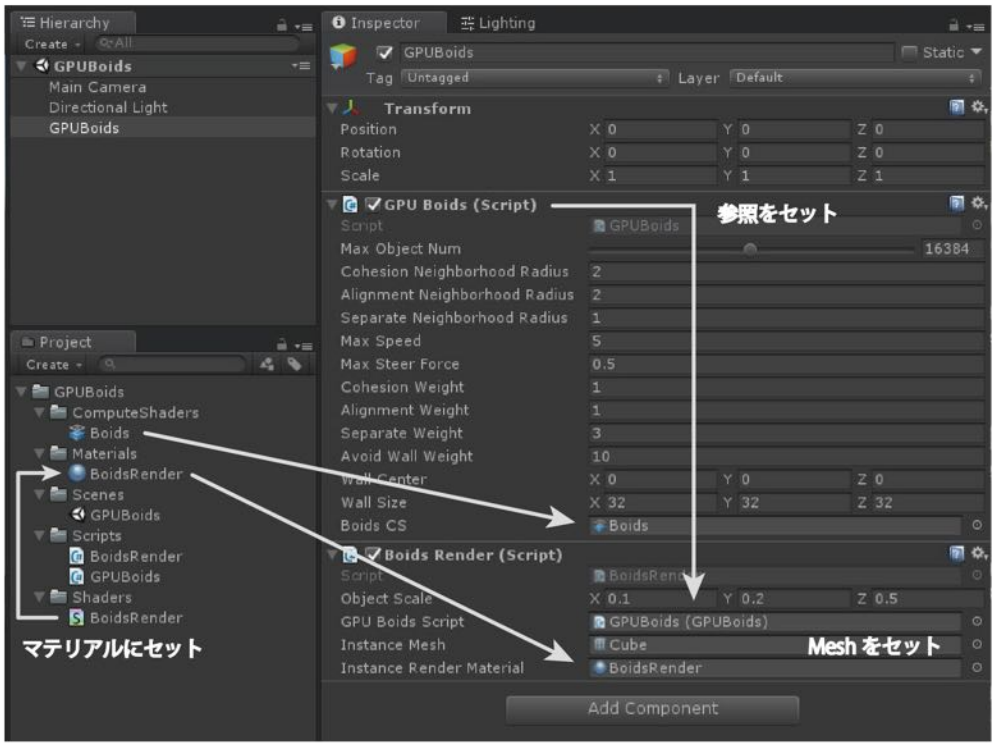
In this sample program, Graphics.DrawMeshInstacedIndirect method is used. Therefore, the Unity version must be 5.6 or later.

3.4 Explanation of implementation code

This sample program consists of the following code.

- GPUBoids.cs --Script that controls the Compute Shader that simulates Boids
- Boids.compute --ComputeShader that simulates Boids
- BoidsRender.cs --C # script that controls the shader that draws Boids
- BoidsRender.shader-Shader for drawing objects with GPU instancing

Scripts, material resources, etc. are set like this



```
Hierarchy
GPUBoids
  Main Camera
  Directional Light
  GPUBoids

Project
GPUBoids
  ComputeShaders
    Boids
  Materials
    BoidsRender
  Scenes
    GPUBoids
  Scripts
    BoidsRender
    GPUBoids
  Shaders
    BoidsRender

Inspector
GPUBoids                                  Static
Tag Untagged                 Layer Default
Transform
Position                        X 0     Y 0     Z 0
Rotation                        X 0     Y 0     Z 0
Scale                           X 1     Y 1     Z 1
GPU Boids (Script)
Script                          GPUBoids
Max Object Num                  16384
Cohesion Neighborhood Radius    2
Alignment Neighborhood Radius   2
Separate Neighborhood Radius    1
Max Speed                       5
Max Steer Force                 0.5
Cohesion Weight                 1
Alignment Weight                1
Separate Weight                 3
Avoid Wall Weight               10
Wall Center                     X 0     Y 0     Z 0
Wall Size                       X 32    Y 32    Z 32
Boids CS                        Boids
Boids Render (Script)
Script                          BoidsRender
Object Scale                    X 0.1   Y 0.2   Z 0.5
GPU Boids Script                GPUBoids (GPUBoids)
Instance Mesh                   Cube
Instance Render Material        BoidsRender
```

Figure 3.2: Settings on Unity Editor

3.4.1 GPUBoids.cs

This code manages Boids simulation parameters, Compute Shader that describes buffers and calculation instructions required for calculations on the GPU, and so on.

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; // Velocity
        public Vector3 Position; // position
    }
    // Thread size of thread group
    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 applies
    public float CohesionNeighborhoodRadius  = 2.0f;
      // Radius with other individuals to which alignment is
applied
    public float AlignmentNeighborhoodRadius = 2.0f;
      // Radius with 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 binding force
    public float CohesionWeight  = 1.0f;
    // Weight of aligning force
    public float AlignmentWeight = 1.0f;
    // Weight of separating force
    public float SeparateWeight  = 3.0f;

    // Weight of force to avoid walls
    public float AvoidWallWeight = 10.0f;
    // Center coordinates of the 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 to Compute Shader for Boids simulation
    public ComputeShader BoidsCS;
    #endregion

    #region Private Resources
    // Buffer that stores the steering force (Force) of the Boid
    ComputeBuffer _boidForceBuffer;
    // Buffer containing basic Boid data (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 the 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 the buffer
        InitBuffer();
    }

    void Update()
    {
        // simulation
        Simulation();
    }

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

    void OnDrawGizmos()
    {
        // Draw the simulation area in wireframe as a debug
        Gizmos.color = Color.cyan;
        Gizmos.DrawWireCube (WallCenter, WallSize);
    }
    #endregion

    #region Private Functions
    // Initialize the buffer
    void InitBuffer()
    {
        // Initialize the buffer
        _boidDataBuffer  = new ComputeBuffer(MaxObjectNum,
            Marshal.SizeOf(typeof(BoidData)));
        _boidForceBuffer = new ComputeBuffer(MaxObjectNum,
            Marshal.SizeOf(typeof(Vector3)));

        // Initialize Boid 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"); // Get the kernel ID
        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); // Run Compute
Shader

        // Calculate speed and position from steering force
        id = cs.FindKernel ("IntegrateCS"); // Get the kernel ID
        cs.SetFloat("_DeltaTime", Time.deltaTime);
                       cs.SetBuffer(id,  "_BoidForceBufferRead",
_boidForceBuffer);
                       cs.SetBuffer(id,  "_BoidDataBufferWrite",
_boidDataBuffer);
         cs.Dispatch (id, threadGroupSize, 1, 1); // Run Compute
Shader
    }
    // Free the buffer
    void ReleaseBuffer()
    {
        if (_boidDataBuffer != null)
        {
            _boidDataBuffer.Release();
            _boidDataBuffer = null;
        }

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

Initialization of Compute Buffer

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

Execution of the function described in ComputeShader

The Simulation function passes the required parameters to ComputeShader and issues a calculation instruction.

The function written 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 computing processing according to the GPU architecture, any number is 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 objects.

The kernel is specified in the ComputeShader script using the #pragma kernel directive. An ID is assigned to each of them, and you can get this ID from the C # script by using the FindKernel method.

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

By executing the Dispatch method, an instruction is issued to calculate the kernel defined in Compute Shader on the GPU. In the arguments, specify the kernel ID and the number of thread groups.

3.4.2 Boids.compute

Describe the calculation instruction to GPU. There are two kernels, one that calculates the steering force and the other that applies that force to update speed and position.

Boids.compute

```
// Specify kernel function
#pragma kernel ForceCS // Calculate steering force
#pragma kernel IntegrateCS // Calculate speed and position

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

// Thread size of thread group
#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; // Number of Boid objects

float _DeltaTime; // Time elapsed from the previous frame

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

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; // Wall center coordinates
float4 _WallSize; // Wall size
float _AvoidWallWeight; // Weight of strength to avoid walls


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

// Return the opposite force when hitting 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;

    // with
    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 for Boid data storage
groupshared BoidData boid_data[SIMULATION_BLOCK_SIZE];

// Kernel function for calculating steering force
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void ForceCS
(
    uint3 DTid: SV_DispatchThreadID, // ID unique to the entire
thread
    uint3 Gid: SV_GroupID, // Group ID
    uint3 GTid: SV_GroupThreadID, // Thread ID in the group
    uint GI: SV_GroupIndex // SV_GroupThreadID in one dimension
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 calculation
     int sepCount = 0; // Variable for counting the number of
other individuals calculated for separation

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

     float3 cohPosSum = float3 (0, 0, 0); // Position addition
```

```
variable for join calculation
     int cohCount = 0; // Variable for counting the number of
other individuals calculated for binding

    // Execution for each SIMULATION_BLOCK_SIZE (number of group
threads) (execution for the number of groups)
    [loop]
            for   (uint   N_block_ID   =   0;   N_block_ID   <
(uint)_MaxBoidObjectNum;
        N_block_ID += SIMULATION_BLOCK_SIZE)
    {
         // Store Boid data for SIMULATION_BLOCK_SIZE in shared
memory
        boid_data[GI] = _BoidDataBufferRead[N_block_ID + GI];

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

        // Calculation with other individuals
                    for   (int   N_tile_ID   =   0;   N_tile_ID   <
SIMULATION_BLOCK_SIZE;
            N_tile_ID++)
        {
            // Position of other individuals
            float3 N_position = boid_data[N_tile_ID].position;
            // 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 the position of
other individuals
            float  dist = sqrt(dot(diff, diff));

            // --- Separation ---
                              if   (dist   >   0.0   &&   dist   <=
_SeparateNeighborhoodRadius)
            {
                       //  Vector  from  the  position  of  another
individual to itself
                       float3  repulse  =  normalize(P_position  -
N_position);
                  // Divide by the distance between yourself and
the position of another individual (the longer the distance, the
smaller the effect)
```

```
                repulse /= dist;
                sepPosSum + = repulse; // Add
                sepCount ++; // Population count
            }

            // --- Alignment ---
                              if   (dist   >   0.0   &&   dist   <=
_AlignmentNeighborhoodRadius)
            {
                aliVelSum + = N_velocity; // Add
                aliCount ++; // Population count
            }

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

    // steering force (separated)
    float3 sepSteer = (float3)0.0;
    if (sepCount > 0)
    {
          sepSteer = sepPosSum / (float) sepCount; // Calculate
the average
         sepSteer = normalize (sepSteer) * _MaxSpeed; // Adjust
to maximum speed
         sepSteer = sepSteer --P_velocity; // Calculate steering
force
          sepSteer = limit (sepSteer, _MaxSteerForce); // Limit
steering force
    }

    // Steering force (alignment)
    float3 aliSteer = (float3)0.0;
    if (aliCount > 0)
    {
          aliSteer = aliVelSum / (float) aliCount; // Calculate
the average velocity of close individuals
         aliSteer = normalize (aliSteer) * _MaxSpeed; // Adjust
to maximum speed
         aliSteer = aliSteer --P_velocity; // Calculate steering
force
```

```
          aliSteer = limit (aliSteer, _MaxSteerForce); // Limit
steering force
    }
    // steering force (combined)
    float3 cohSteer = (float3)0.0;
    if (cohCount > 0)
    {
          // / Calculate the average of the positions of close
individuals
        cohPosSum = cohPosSum / (float)cohCount;
        cohSteer = cohPosSum --P_position; // Find the vector in
the average position direction
         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 a force to
align with the steering force
      force + = cohSteer * _CohesionWeight; // Add force to
combine with steering force
     force + = sepSteer * _SeparateWeight; // Add a separating
force to the steering force

    _BoidForceBufferWrite [P_ID] = force; // Write
}

// Kernel function for speed 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 the
current Boid data
      float3 force = _BoidForceBufferRead [P_ID]; // Read the
steering force

    // Give repulsive force when approaching the wall
    force += avoidWall(b.position) * _AvoidWallWeight;

     b.velocity + = force * _DeltaTime; // Apply steering force
```

```
to speed
    b.velocity = limit (b.velocity, _MaxSpeed); // Limit speed
    b.position + = b.velocity * _DeltaTime; // Update position

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

Calculation of steering force

The ForceCS kernel calculates the steering force.

Utilization of shared memory

Variables with the storage qualifier groupshared will now be written to shared memory. Shared memory cannot write large amounts of data, but it is located close to registers and can be accessed very quickly. This shared memory can be shared within the thread group. By writing the information of other individuals for SIMULATION_BLOCK_SIZE together in the shared memory so that it can be read at high speed within the same thread group, the calculation considering the positional relationship with other individuals is efficient. I will go to the target.

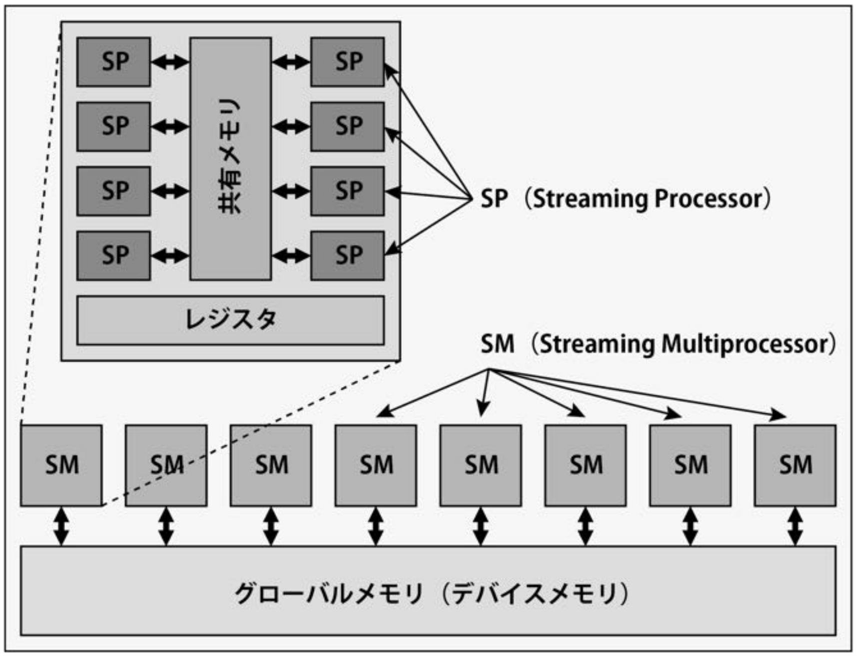

SP
共有メモリ
レジスタ
SP（Streaming Processor）
SM（Streaming Multiprocessor）
SM
グローバルメモリ（デバイスメモリ）

Figure 3.3: Basic GPU architecture

GroupMemoryBarrierWithGroupSync()

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

Steering force is calculated based on the distance to other individuals

Separation

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

Alignment

If there is an individual closer than the specified distance, the velocity (Velocity) of that individual is added up, the number of the individual is counted at the same time, and the velocity of the close individual (that is, the direction in which it is facing) is calculated by those values. Calculate the average of. After the calculation with all the individuals is completed, the steering force is calculated from the relationship with the current speed using the value.

Cohesion

If there is an individual closer than the specified distance, the position of that individual is added, the number of the individual is counted at the same time, and the average (center of gravity) of the position of the close individual is calculated from those values. Furthermore, the vector toward that point is found, and the steering force is found in relation to the current speed.

Update the speed and position of individual Boids

The IntegrateCS kernel updates the speed and position of the Boid based on the steering force obtained by ForceCS (). In AvoidWall, when you try to go out of the specified area, it applies a reverse force to stay inside the area.

3.4.3 BoidsRender.cs

This script draws the results obtained from the Boids simulation on the specified mesh.

BoidsRender.cs

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

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

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

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

    #region Private Variables
```

```
         //  Arguments  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 ()
    {
        // Instancing the mesh
        RenderInstancedMesh();
    }

    void OnDisable()
    {
        // Release the 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 not process
         if (InstanceRenderMaterial == null || GPUBoidsScript ==
null ||
            !SystemInfo.supportsInstancing)
            return;

        // Get the number of indexes of the specified mesh
        uint numIndices = (InstanceMesh != null) ?
            (uint)InstanceMesh.GetIndexCount(0) : 0;
        // Set the number of indexes of the mesh
        args[0] = numIndices;
        // Set the number of instances
        args[1] = (uint)GPUBoidsScript.GetMaxObjectNum();
        argsBuffer.SetData (args); // Set in buffer

        // Set the buffer containing Boid data to the material
        InstanceRenderMaterial.SetBuffer("_BoidDataBuffer",
            GPUBoidsScript.GetBoidDataBuffer());
        // Set the Boid object scale
                InstanceRenderMaterial.SetVector("_ObjectScale",
ObjectScale);
        // define the boundary area
        var bounds = new Bounds
        (
            GPUBoidsScript.GetSimulationAreaCenter(), // 中心
            GPUBoidsScript.GetSimulationAreaSize()    // サイズ
        );
        // GPU instantiate and draw mesh
        Graphics.DrawMeshInstancedIndirect
        (
            InstanceMesh, // Instancing mesh
            0, // submesh index
            InstanceRenderMaterial, // Material to draw
            bounds, // realm domain
            argsBuffer // Argument buffer for GPU instancing
        );
    }
    #endregion
}
```

GPU instancing

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

Graphics.DrawMeshInstancedIndirect () method

This script uses the Graphics.DrawMeshInstancedIndirect method to draw a mesh with GPU instancing. This method allows you to pass the number of mesh indexes and instances as a ComputeBuffer. This is useful if you want to read all instance data from the GPU.

Start () initializes the argument buffer for this GPU instancing. Specify **ComputeBufferType.IndirectArguments** as the third argument of the constructor at initialization .

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

The Graphics.DrawMeshInstancedIndrect method is passed as an argument a buffer that stores data such as the mesh to be instantiated, the index of the submesh, the drawing material, the boundary data, and the number of instances.

This method should normally be called within Update ().

3.4.4 BoidsRender.shader

A shader for drawing that supports 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; // velocity
            float3 position; // position
        };

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

        sampler2D _MainTex; // Texture

        half _Glossiness; // Gloss
        half _Metallic; // Metal characteristics
        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  the
position of Boid
            float3 scl = _ObjectScale; // Get the Boid scale

                   //  Define  a  matrix  to  convert  from  object
coordinates to world coordinates
            float4x4 object2world = (float4x4)0;
            // Substitute scale value
            object2world._11_22_33_44 = float4(scl.xyz, 1.0);
             // Calculate the rotation about the Y axis from the
velocity
            float rotY =
                atan2(boidData.velocity.x, boidData.velocity.z);
             // Calculate the rotation about the X axis from the
velocity
            float rotX =
                                        -asin(boidData.velocity.y   /
(length(boidData.velocity.xyz)
                + 1e-8)); // 0 division prevention
                 //  Find  the  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 transformation of vertices
            v.vertex = mul(object2world, v.vertex);
            // Convert normals 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 using the Graphics.DrawMeshInstancedIndirect method by writing procedural: FunctionName in the #pragma instancing_options directive, specified by FunctionName at the beginning of the vertex shader stage. The function will be called. If you look at the official sample (https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html) etc., in this function, based on the position, rotation and scale of each instance, of the unity_ObjectToWorld matrix, unity_WorldToObject matrix I am rewriting, but in this sample program, I receive Boids data in the vertex shader and perform coordinate conversion of vertices and normals (I do not know if it is good ...). Therefore, nothing is described in the specified setup function.

Get Boid data for each instance with vertex shader and perform coordinate conversion

Describe the processing to be performed on the vertices of the mesh passed to the shader in the vertex shader (Vertex Shader).

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

Ask for rotation

From the Boid's velocity data, calculate the value of rotation that points in the direction of travel. For the sake of intuitive handling, we will use Euler angles for rotation. If you think of a Boid as a flying object, the three-axis rotations of the coordinates relative to the object are called pitch, yaw, and roll, respectively.

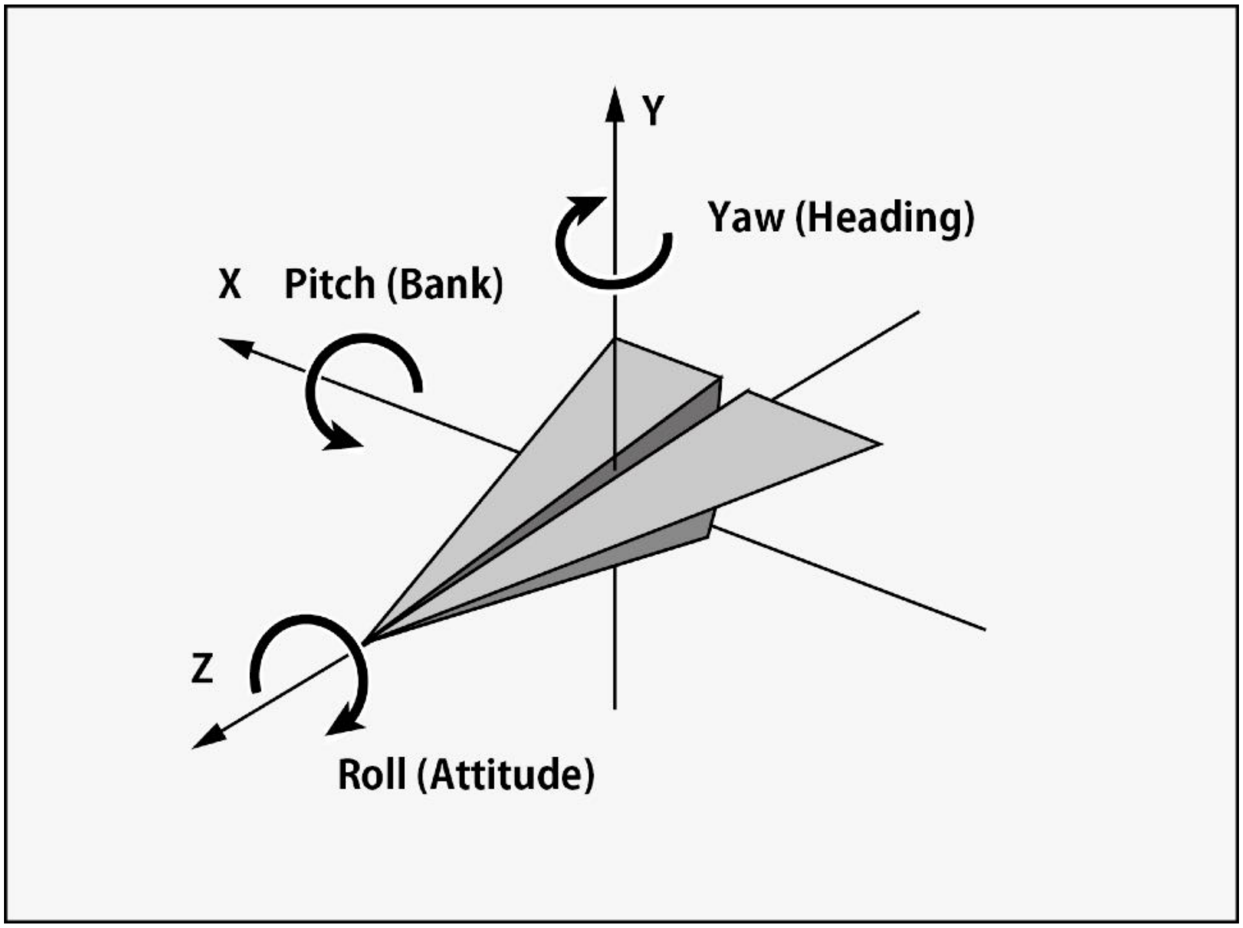


Figure 3.4: Axis and Rotation Names

First, from the velocity about the Z axis and the velocity about the X axis, find the yaw (which direction is facing the horizontal plane) using the atan2 method that returns the arctangent.

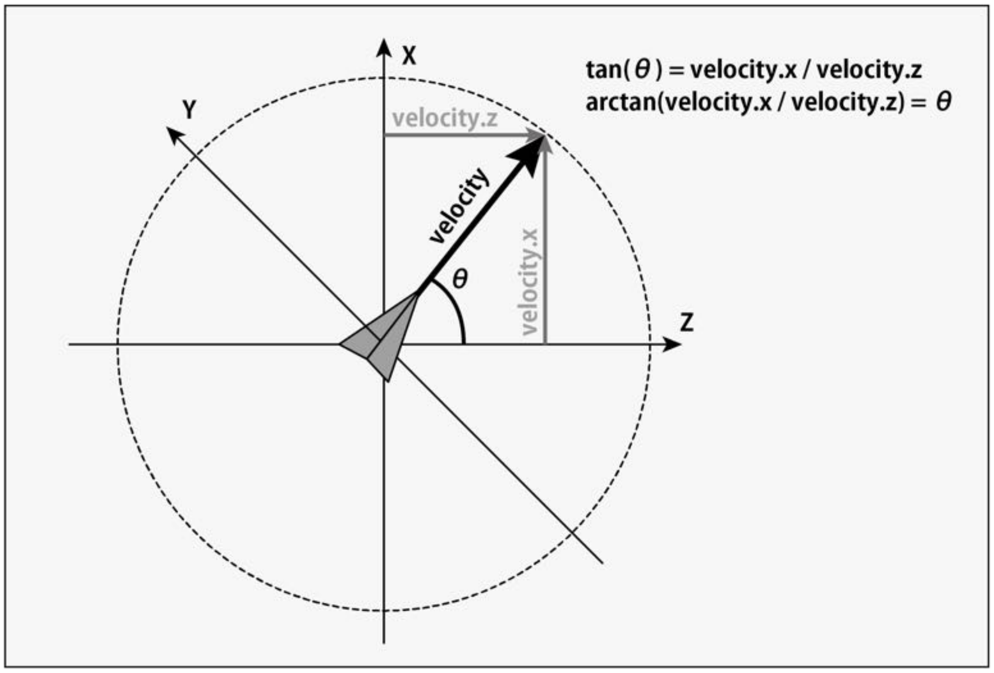

X
Y
Z
velocity
velocity.z
velocity.x
θ
tan(θ) = velocity.x / velocity.z
arctan(velocity.x / velocity.z) = θ

Figure 3.5: Relationship between speed and angle (yaw)

Next, from the magnitude of the velocity and the ratio of the velocity with respect to the Y axis, the pitch (slope up and down) is calculated using the asin method that returns an inverse sine (arc sine). If the speed of the Y axis is small among the speeds of each axis, the amount of rotation is weighted so that there is little change and the speed remains horizontal.

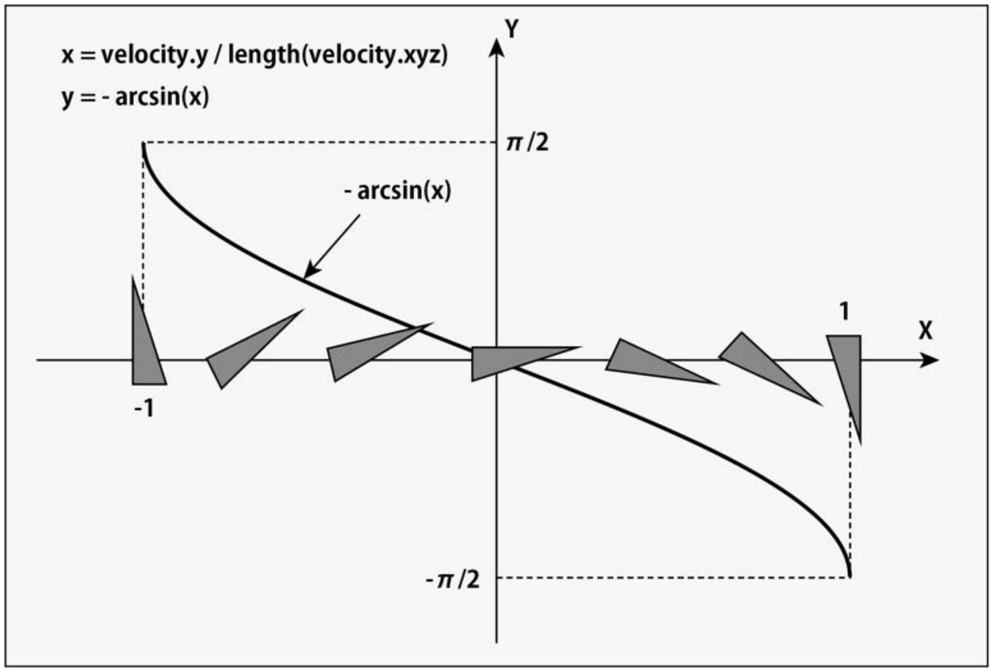


Figure 3.6: Relationship between velocity and angle (pitch)

Calculate the matrix to apply the Boids transform

Coordinate transformation processes such as movement, rotation, and scaling can be collectively represented by a single matrix. Defines a 4x4 matrix object2world.

Scale

First, substitute the scale value. The matrix S that scales by \ rm S_x S_y S_z {} on each of the XYZ axes is expressed as follows.

```
\rm
S=
\left(
\begin{array}{cccc}
\rm S_x & 0 & 0 & 0 \\
0 & \rm S_y & 0 & 0 \\
0 & 0 & \rm S_z & 0 \\
0 & 0 & 0 & 1
\end{array}
\right)
```

Variables of type float4x4 in HLSL can specify specific elements of the matrix using a swizzle such as ._11_22_33_44. By default, the components are arranged as follows:

Form 3.1:

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

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

rotation

Then apply the rotation. If the rotation \ rm R_x R_y R_z {} for each of the XYZ axes is represented by a matrix,

```
\rm
R_x(\phi)=
\left(
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & \rm cos(\phi) & \rm -sin(\phi) & 0 \\
0 & \rm sin(\phi) & \rm cos(\phi) & 0 \\
0 & 0 & 0 & 1
\end{array}
\right)

\rm
R_y(\theta)=
\left(
\begin{array}{cccc}
\rm cos(\theta) & 0 & \rm sin(\theta) & 0 \\
0 & 1 & 0 & 0 \\
\rm -sin(\theta) & 0 & \rm cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{array}
\right)

\rm
R_z (\ psi) =
\left(
\begin{array}{cccc}
\rm cos(\psi) & \rm -sin(\psi) & 0 & 0 \\
\rm sin(\psi) & \rm cos(\psi) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{array}
\right)
```

Combine this into a matrix. At this time, the behavior at the time of rotation changes depending on the order of the axes of rotation to be combined, but if you combine in this order, it should be similar to the standard rotation of Unity.

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

Figure 3.7: Synthesis of rotation matrix

The rotation is applied by finding the product of the rotation matrix thus obtained and the matrix to which the above scale is applied.

Translation

Then apply translation. Assuming that \ rm T_x T_y T_z {} translates to each axis , the matrix is expressed as follows.

```
\ rm T =
\left(
\begin{array}{cccc}
1 & 0 & 0 & \rm T_x \\
0 & 1 & 0 & \rm T_y \\
0 & 0 & 1 & \rm T_z \\
0 & 0 & 0 & 1
\end{array}
\right)
```

This translation can be applied by adding the Position data for each of 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 objects that move like a group like this are drawn.

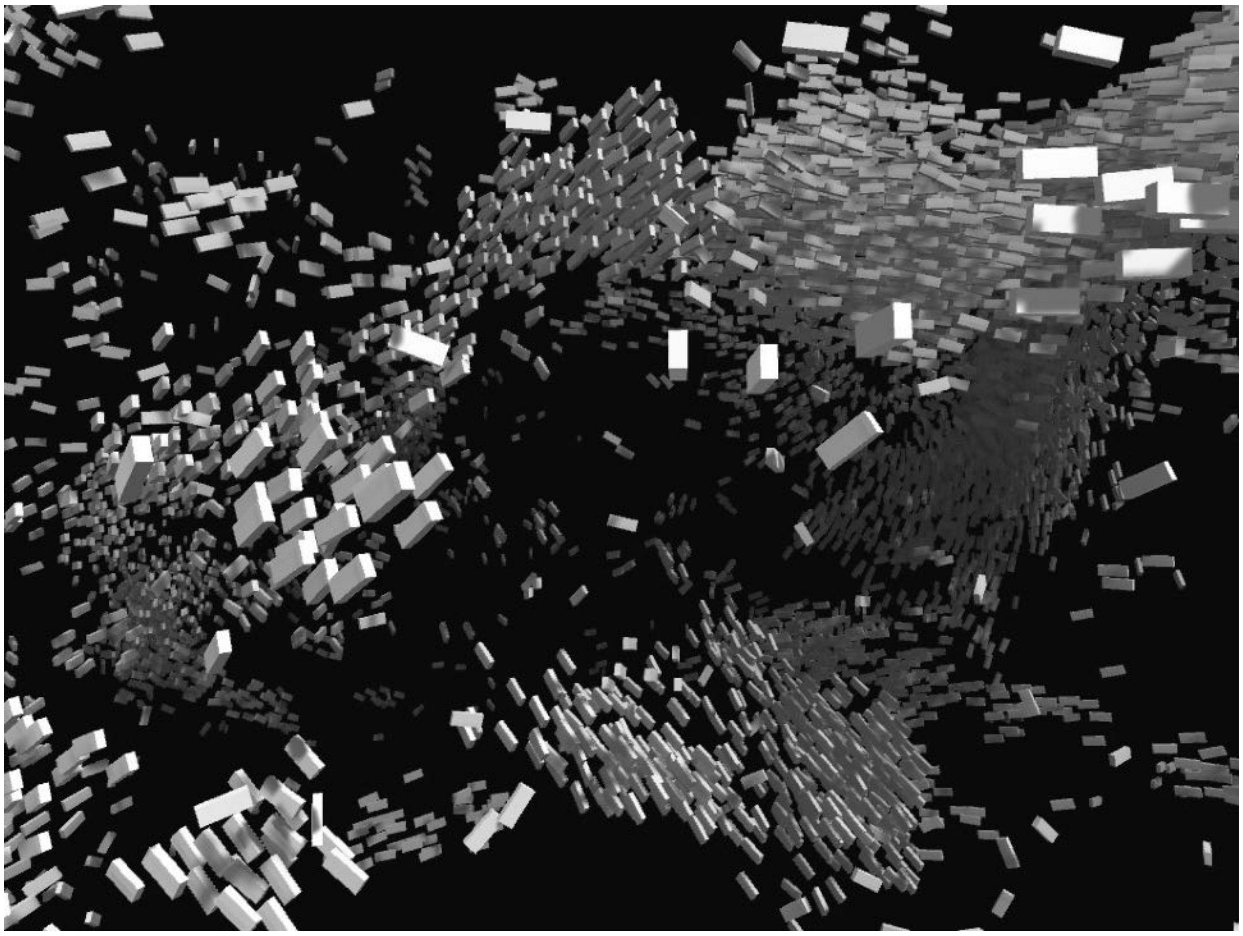


Figure 3.8: Execution result

3.5 Summary

The implementation introduced in this chapter uses the minimum Boids algorithm, but it has different characteristics such as a large group or a number of small colonies even by adjusting the parameters. I think it will move. In addition to the basic rules of conduct shown here, there are other rules to consider. For example, if this is a school of fish and foreign enemies that prey on them appear, they will naturally move away, and if there are obstacles such as terrain, the fish will avoid hitting them. When thinking about 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, it will be closer to the actual one. The characteristics of movement also change depending on the environment such as whether it flies in the sky, moves in water, or moves 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 more individuals than calculation by CPU, but basically the calculation with other individuals is done by brute force, and the calculation efficiency is not very good. To do this, the calculation cost is improved by improving the efficiency of searching for nearby individuals, such as registering individuals in an area divided by a grid or block according to their position and performing calculation processing only for individuals existing in adjacent areas. Can be suppressed.

There is still plenty of room for improvement, and by applying appropriate implementation and behavioral rules, we will be able to express even more beautiful, powerful, dense and tasty group movements. I want to be able to do it.

See 3.6

- 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-TimeParticleSystemsOnThe_GPU.pdf
- Practical Rendering and Computation with Direct3D 11 - https://dl.acm.org/citation.cfm?id=2050039
- Introduction to GPU Parallel Graphic Processing -- http://gihyo.jp/book/2014/978-4-7741-6304-8

Chapter 4 Fluid Simulation by Lattice Method

4.1 About this chapter

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

4.2 Sample data

4.2.1 Code

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

It is stored in Assets / StabeFluids of.

4.2.2 Execution environment

- Shader model 5.0 compatible environment where ComputeShader can be executed
- Operation confirmed in writing environment, Unity5.6.2, Unity2017.1.1

4.3 Introduction

In this chapter, we will explain the fluid simulation by the lattice method and the calculation method and understanding of mathematical formulas necessary for realizing them. First of all, what is the grid method? In order to explore its meaning, let's take a closer look at how to analyze "flow" in fluid mechanics.

4.3.1 How to understand in fluid mechanics

Fluid mechanics is characterized by formulating a natural phenomenon, "flow," and making it computable. How can this "flow" be quantified and analyzed?
If you go straight to it, you can quantify it by deriving the "flow velocity when the time advances for a moment". To put it a little mathematically, it can be rephrased as an analysis of the amount of change in the flow velocity vector when differentiating with time.
However, there are two possible methods for analyzing this flow.
One is to measure the flow velocity vector of each fixed lattice space by dividing the hot water in the bath into a grid when imagining the hot water in the bath.
And the other is to float a duck in the bath and analyze the movement of the duck itself. Of these two methods, the former is called the "Euler's method" and the latter is called the "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 roughly divided into the following three types.

- Lattice method (eg Stable Fluid)
- Particle Method (eg SPH)
- Lattice method + particle method (eg FLIP)

As you can imagine from the meaning of Chinese characters, the grid method creates a grid-like "field" when simulating the flow, like the "Euler's method", and when it is differentiated with time, it is It is a method of simulating the speed of each grid. In addition, the particle method is a method of simulating the advection of the particles themselves, focusing on the particles, such as the "Lagrange method".
Along with the lattice method and particle method, there are areas of strength and weakness in 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. (You can imagine
these strengths and weaknesses when you think of how to analyze Euler's

method and Lagrange's method.) To supplement these, the lattice method + particle method represented by the FLIP method. There are also methods that complement each other's areas of expertise.

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

4.4 About the Navier-Stokes equation

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

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

```
\dfrac {\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 represents the density field. The third is the "continuity equation (conservation of mass)". Let's unravel these three formulas one by one.

4.5 Continuity equation (conservation of mass)

First, let's unravel the "continuity equation (conservation of mass)", which is short as an equation and works as a condition when simulating an "incompressible" fluid.
When simulating a fluid, it is necessary to make a clear distinction between compressible and incompressible objects. For example, if the target is a gas whose density changes with pressure, it will be a compressible fluid. Conversely, objects with a constant density, such as water, are incompressible fluids.
Since this chapter deals with incompressible fluid simulations, the divergence of each cell in the velocity field should be kept at zero. That is, it offsets the inflow and outflow of the velocity field and keeps it at zero. 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 continuity equation (conservation of mass).

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

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

4.5.1 Divergence

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

\ nabla (nabla operator) is called the vector differential operator. For example, assuming that the vector field is two-dimensional, the partial differential of \ left (\ dfrac {\ partial} {\ partial x} _, \ dfrac {\ partial} {\ partial y} \ right) is calculated as shown in the figure. It acts as an operator that simplifies the notation of partial differential when taking. Since the \ nabla operator is an operator, it has no meaning by itself, but the operation content changes depending on whether the expression to be combined together is an inner product, an outer product, or just a function such as \ nabla f .
This time, let's explain "divergence" which takes the inner product of partial differential. First, let's see why this formula means "divergence".

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

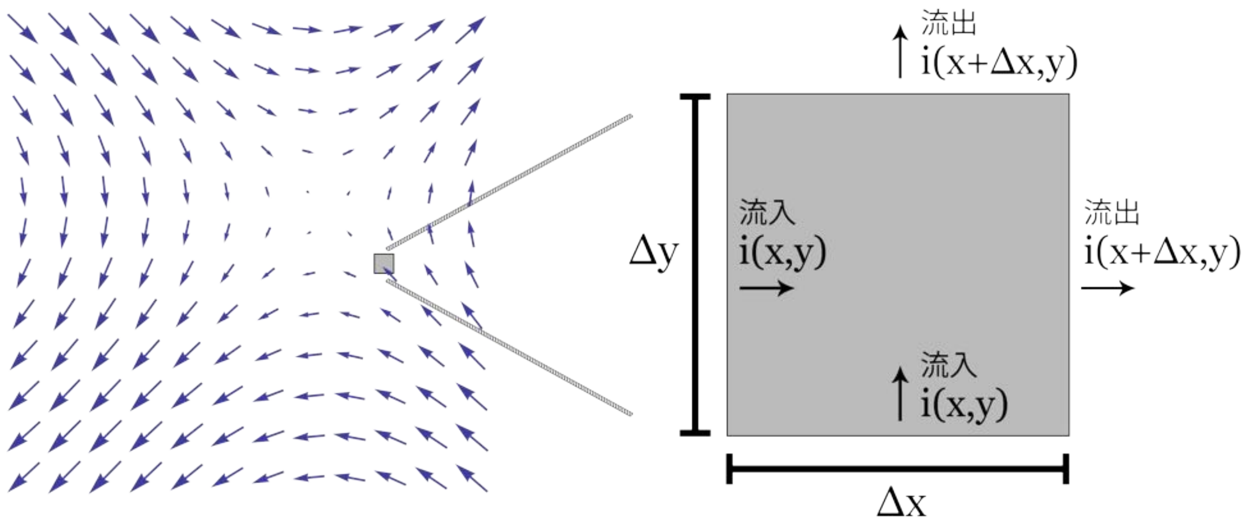


Figure 4.1: Extracting cells in the differential interval (Δx, Δy) from the vector field

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

As mentioned above, the divergence is the amount of change between the specific point x in the x direction and the slightly advanced \ Delta x when looking at the partial differential when the cell of the vector field is cut out, and the specific amount in the y direction. It can be calculated by the inner product of the amount of change between the point y and the slightly advanced \ Delta y . The reason why the outflow can be obtained by the inner product with the partial differential can be proved by performing a differential operation on 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}
```

Taking the limit from the above formula,

```
\ 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} = \ dfrac {\ partial i} {\
partial x} + \ dfrac {\ partial j} {\ partial y }
```

By doing so, we can see that the final equation is 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 body of the lattice method. Before that, in implementing the Navier-Stokes equation of the velocity field, let's confirm the gradient and the Laplacian in addition to the divergence confirmed earlier.

4.6.1 Gradient

```
\nabla f(x, y) = \left( \dfrac{\partial f}{\partial
x}_,\dfrac{\partial f}{\partial y}\right)
```

\ nabla f (grad \ f) is the formula for finding the gradient. The meaning is which vector is finally directed 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. I will. In other words, it is possible to calculate a vector that points in the direction of the larger value when partially differentiated.

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}
```

The Laplacian is represented by a symbol with the nabla turned upside down. (Same as delta, but read from the context and make sure you don't make a mistake.) Also write
\ nabla ^ 2 f or \ nabla \ cdot \ nabla f and it is calculated as the second derivative.
Also, if you think about it by disassembling it, you can take the form of finding the divergence by taking the gradient of the function.
In terms of meaning, in the vector field, the part concentrated in the gradient

direction has a lot of inflow, so when the divergence is taken, the part with a low gradient has a lot of springing out, so when the divergence is taken, it becomes +. I can imagine that.
Laplacian operators include scalar Laplacian and vector Laplacian, and when acting on a vector field, gradient, divergence, and rotation (cross product of ∇ and vector) are used.

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

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

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

This completes the confirmation of the mathematical formulas required to solve the Navier-Stokes equation in the grid 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

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

Of the above, \ overrightarrow {u} is the flow velocity, \ nu is the kinematic viscosity, and \ overrightarrow {f} is the external force (force).
You can see that the left side is the flow velocity when the partial differential is taken with respect to 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 collectively at the time of calculation, it is necessary to implement them in steps at the time of implementation.
First of all, as a step, if you do not receive an external force, you cannot make a change under the initial conditions, so I would like to start with the external force term in the fourth term.

4.6.4 External force term of velocity field

This is simply the part that adds the vectors 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 as the starting point of the vector or some event.
The kernel of the external force term of the compute shader is implemented as follows. Also, describe the definitions of each coefficient and buffer that will be used in the compute shader.

```
float visc; //Dynamic viscosity coefficient
float dt; // delta time
float  velocityCoef;  //external  force  coefficient  of  velocity
field
float densityCoef; //Density outside pressure coefficient

//  xy  =  velocity,  z  =  density,  fluid  solver  to  pass  to  drawing
shader
RWTexture2D<float4> solver;
//density field, density field
RWTexture2D<float>  density;
//velocity field, velocity field
RWTexture2D<float2> velocity;
// xy = for vel, z = for dens. when project, x = p, y = div
//  Save  the  buffer  one  step  before  and  the  temporary  buffer  when
saving the mass
RWTexture2D<float3> prev;
//  xy  =  velocity  source,  z  =  density  source  External  force  input
buffer
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 diffusion viscosity term.

4.6.5 Dispersion viscosity term of velocity field

```
\nu \nabla ^{2} \overrightarrow {u}
```

When there are values on the left and right of the \ nabla operator and \ Delta operator, there is a rule that "acts only on the right element", so in this case, leave the kinematic viscosity coefficient once and leave the vector Laplacian part. Think first.
With vector Laplacian for the flow velocity \ overright arrow {u} , the gradient and divergence of each component of the vector are taken and combined, and the flow velocity is diffused adjacently. By multiplying it by the kinematic viscosity coefficient, the momentum of diffusion is adjusted.
Here, since the gradient of each component of the flow velocity is taken and diffused, inflow from the adjacency and outflow to the adjacency occur, and the phenomenon that the vector received in step 1 affects the adjacency can be understood. think.
In terms of mounting, some ingenuity is required. If implemented according to the formula, if the diffusivity obtained by multiplying the viscosity coefficient by the differential time / number of lattices becomes high, vibration will occur, convergence will not be achieved, and the simulation itself will eventually diverge.
Iterative methods such as the Gauss-Seidel method, Jacobi method, and SOR method are used here to make the diffusion stable. Here, let's simulate with the Gauss-Seidel method.
The Gauss-Seidel method is a method of converting a formula into a linear equation consisting of unknowns for its own cell, using the calculated value immediately at the next iteration, and chaining it to converge to an

approximate answer. The higher the number of iterations, the more accurate the values will converge, but graphics in real-time rendering require better frame rates and aesthetics rather than accurate results, so iterations are machine. Adjust for 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);
        }
    }
}
```

The SetBoundaryVelocity function above is a method for boundaries. Please refer to the repository for details.

4.6.6 Quality preservation

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

Let's go back to the conservation of mass side before proceeding with the section. In the process so far, the force received in the external force term is diffused in the velocity field, but at present, the mass of each cell is not preserved, and the mass is in the place where it keeps springing out and the place where there is a lot of inflow. Is in an unsaved state.
As in the equation above, you must save the mass and bring the divergence

of each cell to 0, so let's save the mass here.
In addition, when performing the mass conservation step with Compute Shader, the field must be fixed because the partial differential operation with the adjacent thread is performed. It was expected to speed up if the partial differential operation could be performed in the group shared memory, but when the partial differential was taken from another group thread, the value could not be obtained and the result was dirty, so here is a buffer. While confirming, proceed in 3 steps.
Divergence calculation from velocity field> Poisson's equation is calculated by Gauss-Seidel method> Subtract to velocity field, divide the
kernel into 3 steps of conservation of mass, and bring it to conservation of mass while determining the field. In addition, SetBound ~ system is a method call for the boundary.

```
//Quality preservation Step1.
// In step1, 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 Step2.
// In step2, the Poisson equation is solved by the Gauss-Seidel
```

```
method from the divergence obtained in step1.
[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 Step3.
// In step3, set ∇ · u = 0.
[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);
    }
}
```

The velocity field is now in a state of conservation of mass. Since the inflow occurs at the place where the outflow occurs and the outflow occurs from the place where there is a lot of inflow, it is expressed like a fluid.

4.6.7 Advection term

```
-\left( \overrightarrow {u} \cdot \nabla \right) \overrightarrow
{u}
```

Lagrange's method is used for the advection term, but the work of back tracing the velocity field one step before and moving the value of the place where the velocity vector is subtracted from the corresponding cell to the current location Do this for each cell. When backtraced, it does not go back to the place where it fits exactly in the grid, so when 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;

        // Back trace point index.
        x = (float)id.x - dfdt * prev[id].x;
        y = (float)id.y - dfdt * prev[id].y;
          // Clamp so that the points are within the simulation
range.
        clamp(x, 0.5, w + 0.5);
        clamp(y, 0.5, h + 0.5);
        // Determining cells near the back trace point.
        ddx0 = floor(x);
        ddx1 = ddx0 + 1;
        ddy0 = floor(y);
        Dy1 = Dy0 + 1;
          // Save the difference for linear interpolation with
neighboring cells.
        s1 = x - ddx0;
        s0 = 1.0 - s1;
        t1 = y - ddy0;
        t0 = 1.0 - t1;

         // Backtrace, take the value one step before by linear
interpolation with the neighborhood, and substitute it for the
current velocity field.
        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 look at the density field equation.

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

Of the above, \ overrightarrow {u} is the flow velocity, \ kappa is the diffusion coefficient, ρ is the density, and S is the external pressure.
The density field is not always necessary, but by placing the pixels on the screen diffused by the density field on each vector when the velocity field is calculated, it becomes possible to express a more fluid-like expression that flows while melting. I will.
As some of you may have noticed by looking at the formula of the density field, the flow is exactly the same as the velocity field, the difference is that the vector is a scalar and the kinematic viscosity coefficient \ nu is the diffusion coefficient. There are only three points, one that is \ kappa and the other that does not use the law of conservation of mass.
Since the density field is a field of change in density, it does not need to be

incompressible and does not need to be conserved by mass. In addition, the kinematic viscosity coefficient and the diffusion coefficient have the same usage as coefficients.
Therefore, it is possible to implement the density field by making a kernel other than the mass conservation law of the kernel used in the velocity field earlier by lowering the dimension. I will not explain the density field on paper, but please refer to the density field as it is also implemented in the repository.

4.8 Each item step of the simulation

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

- Generates an external force event and inputs it to the external force term of the velocity field and density field.
- Update the speed field in the following steps
    - Nisan viscosity term
    - Quality preservation
    - Advection term
    - Quality preservation
- Then update the density field in the following steps
    - Stray term
    - Advection density using velocity in velocity field

The above is the simulation step of StableFluid.

4.9 Results

By executing and dragging on the screen with the mouse, it is possible to cause the following fluid simulation.

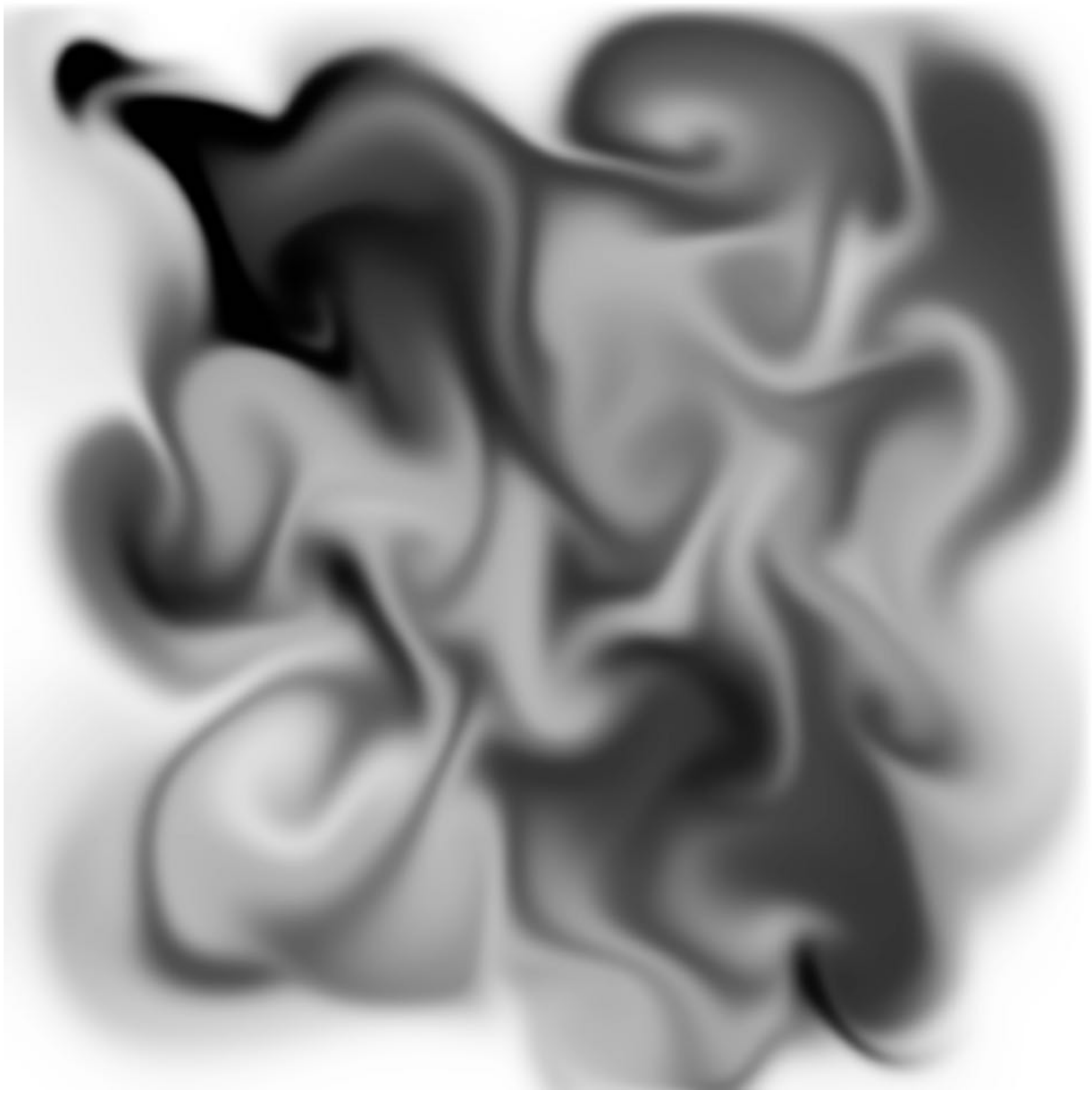


Figure 4.2: Execution example

4.10 Summary

Fluid simulation, unlike pre-rendering, is a heavy field for real-time game engines like Unity. However, due to the improvement of GPU computing power, it has become possible to produce FPS that can withstand even a certain resolution if it is two-dimensional. Also, if you try to implement the Gauss-Seidel iterative method, which is a heavy load for the GPU that came out on the way, with another process, or substitute the speed field itself with curl noise, etc. It will be possible to express fluids with lighter calculations.

If you have read this chapter and are interested in fluids even a little, please try "Fluid simulation by particle method" in the next chapter. Since you can approach the fluid from a different angle than the grid method, you can experience the depth of fluid simulation and the fun of mounting.

4.11 Reference

- Jos Stam. SIGGRAPH 1999. Stable Fluids

Chapter 5 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 the particle method, which is another fluid simulation method, especially the SPH method, to express the movement of the fluid. Please note that there are some inadequate expressions as the explanation is given in a slightly chewed manner.

5.1 Basic knowledge

5.1.1 Euler' s perspective and Lagrange' s perspective

There are Euler's viewpoint and Lagrange's viewpoint as the method of observing the movement of fluid. Euler's viewpoint is to **fix** observation points on the fluid at equal intervals and analyze the movement of the fluid at the observation points. On the other hand, the Lagrange viewpoint is to **float** an observation point that moves along the flow of fluid and observe the movement of the fluid at that observation point (see Fig. 5.1). Basically, the fluid simulation method using Euler's viewpoint is called the lattice method, and the fluid simulation method using Lagrange's viewpoint is called the particle method.

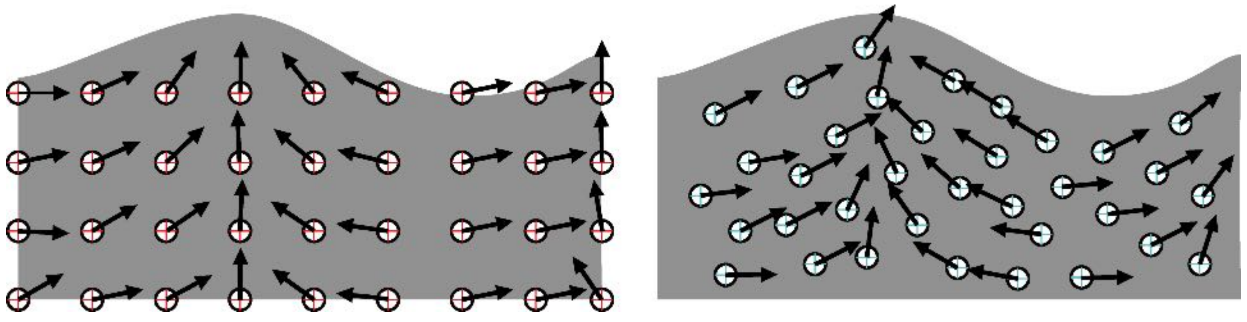


Figure 5.1: Left: Euler-like, Right: Lagrange-like

5.1.2 Lagrange derivative (material derivative)

The method of calculating the derivative differs between the Euler perspective and the Lagrange perspective. First , the physical quantity * 1 expressed from Euler's point of view is shown below.

[* 1] Physical quantities refer to observable velocities and masses. In short, you can think of it as something that has a unit.

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

This means the physical quantity \phi at the position \overrightarrow {x} at time t . The time derivative of this physical quantity is

```
\frac{\partial \phi}{\partial t}
```

Can be expressed as. Of course, this is a derivative from Euler's point of view because the position of the physical quantity is fixed by \overridearrow {x} .

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

On the other hand, from the Lagrange perspective , the observation point itself is a function of time because it moves * 2 along the flow . Therefore, the observation point at \overrightarrow {x} _0 in the initial state is at time t .

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

Exists in. Therefore, the notation of physical quantities is also

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

It will be. Looking at the current physical quantity and the amount of change in the physical quantity after \Delta t seconds according to the definition of differentiation

```
        \displaystyle     \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}

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

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

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

It 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

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

It can be shortened by introducing the operator. A series of operations that take into account the movement of observation points is called Lagrange differentiation. At first glance, it may seem complicated, but in the particle method where the observation points move, it is more convenient to express the equation from a Lagrangian point of view.

5.1.3 Fluid uncompressed conditions

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

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

This indicates that there is no gushing or disappearance in the fluid. Since the derivation of this equation involves a slightly complicated integral, the explanation is omitted [* 3] . Think of it as "do not compress the fluid!"

[* 3] It is explained in detail in "Fluid Simulation for Computer Graphics -- Robert Bridson".

5.2 Particle method simulation

In the particle method, a fluid is divided into small **particles** and the movement of the fluid is observed from a Lagrangian perspective. This particle corresponds to the observation point in the previous section. Even if it is called the "particle method" in one word, many methods have been proposed at present, and it is famous

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

And so on.

5.2.1 Derivation of Navier-Stokes equation in particle method

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

```
  \dfrac{D \overrightarrow{u}}{Dt} = -\dfrac{1}{\rho}\nabla p +
\nu \nabla \cdot \nabla \overrightarrow{u} + \overrightarrow{g}
  \label{eq:navier}
```

The shape is a little different from the NS equation that came out in the grid method in the previous chapter. The advection term is completely omitted, 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 there is no need to consider the advection term when calculating the NS equation. The calculation of advection can be completed by directly updating the particle position based on the acceleration calculated by the NS equation.

A real fluid is a collection of molecules, so it can be said to be 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 computable size. Each grain (* 4) shown in Figure 5.2 represents a portion of the fluid divided by a computable size. Each of these grains can be thought of as having a mass m , a position vector \ overridearrow {x} , a velocity vector \ overridearrow {u} , and a volume V , respectively.

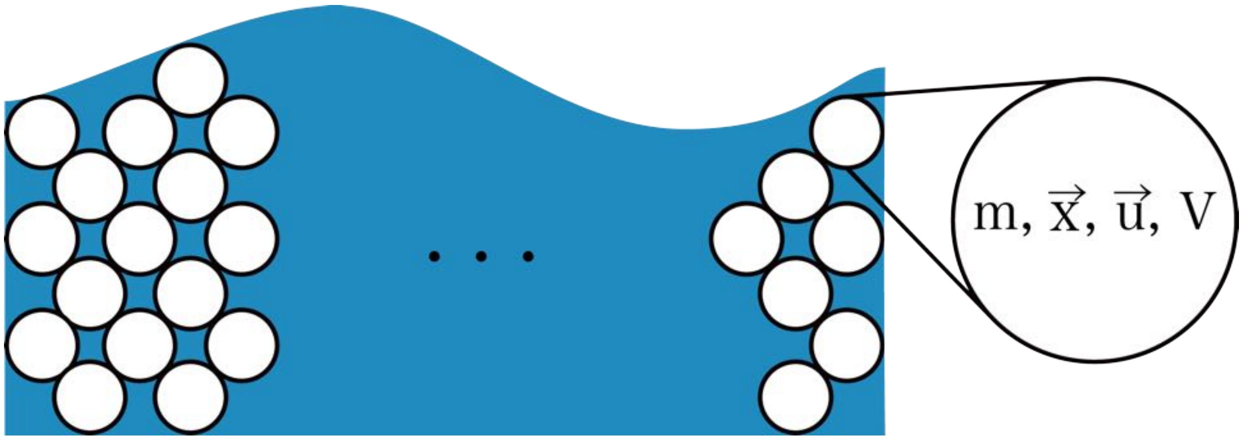


Figure 5.2: Fluid particle approximation

For each of these grains , the acceleration is calculated by calculating the external force \ overridearrow {f} and solving the equation of motion m \ overridearrow {a} = \ overridearrow {f} , and how in the next time step. You can decide whether to move to.

[* 4] Called'Blob'in English

As mentioned above, each particle moves by receiving some force from the surroundings, but what is that "force"? A simple example is gravity m \ overrightarrow {g}, but it should also receive some force from surrounding particles. These forces are explained below.

pressure

The first force exerted on a fluid particle is pressure. The fluid always flows from the higher pressure side to the lower pressure side. If the same amount of pressure is applied from all directions, the force will be canceled and the movement will stop, so consider the case where the pressure balance is uneven. As mentioned in the previous chapter, by taking the gradient of the pressure scalar field, it is possible to calculate the direction with the highest pressure rise rate from the viewpoint of your own particle position. The direction in which the particles receive the force is from the one with the highest pressure to the one with the lowest pressure, so take the minus and become-\ nabla p . Also, since particles have a volume, the pressure applied to the particles is calculated by multiplying-\ nabla p by the volume of the particles [* 5] . Finally, the result --V \ nabla p is derived.

[* 5] Due to the uncompressed condition of the fluid, the integral of the pressure applied to the particles can be expressed simply by multiplying the volume.

Viscous force

The second force applied to fluid particles is viscous force. A viscous fluid is a fluid that is hard to deform, such as honey and melted chocolate. Applying the word viscous to the expression of the particle method, **the velocity of a particle is easy to average the velocity of the surrounding particles** . As mentioned in the previous chapter, the operation of averaging the surroundings can be performed using the Laplacian.

Expressing the degree of **viscosity** using the **kinematic viscosity coefficient** \ mu , it can be expressed as \ mu \ nabla \ cdot \ nabla \ overridearrow {u} .

Integration of pressure, viscous force and external force

Applying these forces to the equation of motion m \ overrightarrow {a} = \ overrightarrow {f} ,

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

Here, since m is \ rho V, it is transformed (V is canceled).

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

Divide by both sides \ rho ,

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

Finally, the coefficient of viscosity term \ dfrac {\ mu} {\ rho} in \ nu by introducing,

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

Therefore, we were 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 by simply moving the particle position. In the actual calculation of the time derivative, infinitely small time is used, but since infinity cannot be expressed by computer calculation, the differentiation is expressed using sufficiently small time \ Delta t . This is called the difference, and the smaller \ Delta t , the more accurate the calculation.

Introducing the expression of difference for acceleration,

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

It will be. So the velocity increment \ Delta \ overrightarrow {u} is

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

And also for the position increment,

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

Than,

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

It will be.

By using this result, the velocity vector and position vector in the next frame can be calculated. Assuming that the particle velocity in the current frame is \ overrightarrow {u} _n , the particle velocity in the next frame is \ overrightarrow {u} _ {n + 1} .

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

Can be expressed as.

Assuming that the particle position in the current frame is \ overridearrow {x} _n , the particle position in the next frame is \ overridearrow {x} _ {n + 1} .

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

Can be expressed as.

This technique is called the forward Euler method. By repeating this every frame, it is possible to express the movement of particles at each time.

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 as they are on a computer, so some kind of approximation needs to be made. As a method, I will explain the **SPH method** that is often used in the CG field .

The SPH method was originally used for collision simulation between celestial bodies in astrophysics , but it was also applied to fluid simulation in CG by Desbrun et al. * 6 in 1996 . In addition, parallelization is easy, and the current GPU can calculate a large number of particles in real time. In computer simulation, it is necessary to discretize continuous physical quantities and perform calculations, and the method of performing this discretization using a function called a **weight function** 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 an influence range, and the closer the particle is to other particles, the greater the influence of that particle. Figure 5.3 shows the extent of this effect .

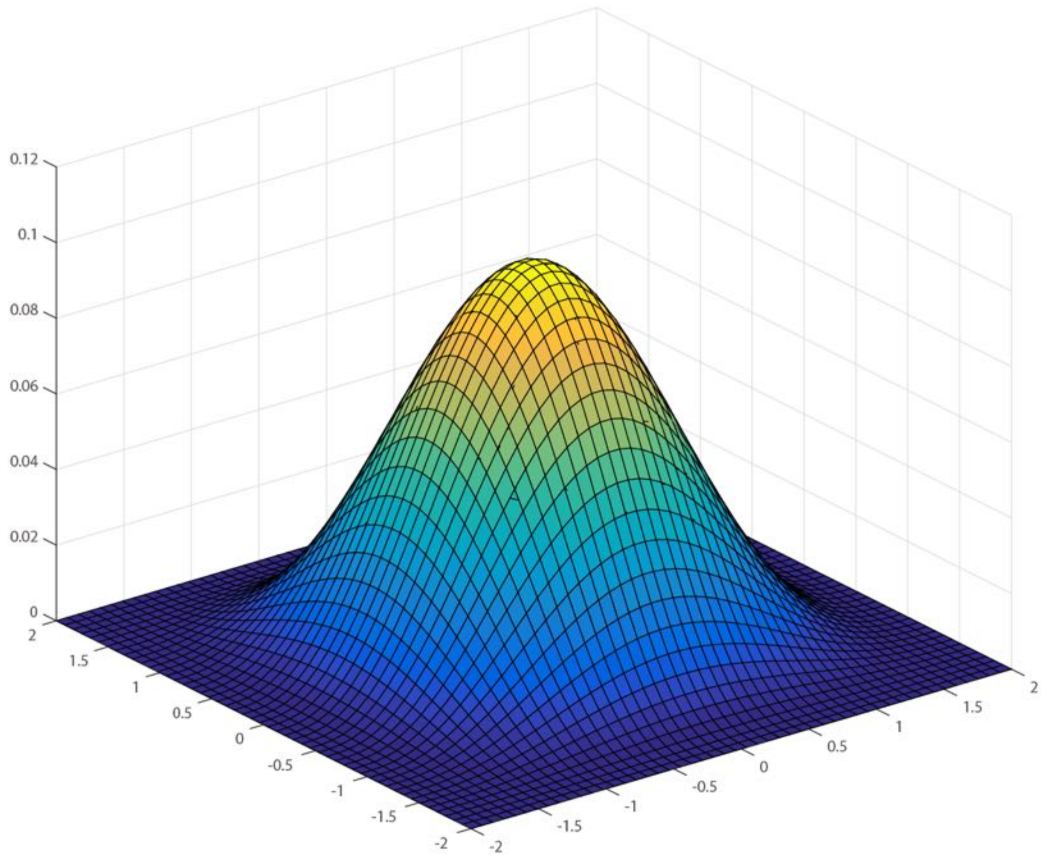


Figure 5.3: Two-dimensional weighting function

This function is called the **weight function** * 7 .

[* 7] Normally, this function is also called a kernel function, but it is called this function to distinguish it from the kernel function in Compute Shader.

Assuming that the physical quantity in the SPH method is ϕ , it is discretized as follows using a weighting function.

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

N, m, ρ, and h are the set of neighboring particles, the mass of the particles, the density of the particles, and the 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, and 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

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

Can be expressed as. As you can see from the equation, the physical quantity gradient and the Laplacian are images that apply only to the weighting function. The weight function W is different depending on the physical quantity to be obtained, but the explanation of the reason is omitted * 8 .

[* 8] It is explained in detail in "Basics of Physical Simulation for CG-Makoto Fujisawa".

5.3.2 Discretization of density

The density of fluid particles is determined by using the formula of the physical quantity discretized by the weighting function.

```
     \rho(\overrightarrow{x})   =   \sum_{j   \in   N}m_jW_{poly6}
(\overrightarrow{x_j} - \overrightarrow{x}, h)
```

Is given. Here, the weight function W to be used is given below.

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

Figure 5.4: Poly6 weight function

5.3.3 Discretization of viscous terms

Discretizing the viscosity term also uses 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)
```

Is expressed as. Where the weighting function Laplacian \ nabla ^ 2 W_ {visc} is given below.

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

Figure 5.5: Laplacian of Viscosity weighting function

5.3.4 Discretization of pressure terms

Similarly, the pressure term is discretized.

```
   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 below.

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

Figure 5.6: Gradient of Spiky weight function

At this time, the pressure of the particles is called the 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, Poisson's equation must be solved, but it is not suitable for real-time calculation. Instead, it is said that the SPH method [* 9] is not as good at calculating the pressure term as the lattice method in terms of ensuring incompressibility approximately.

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

5.4 Implementation of SPH method

Samples can be found under Assets / SPHFluid in this repository (https://github.com/IndieVisualLab/UnityGraphicsProgramming). Please note that in this implementation, speedup and numerical stability are not considered in order to explain the SPH method as simply as possible.

5.4.1 Parameters

A description of the various parameters used in the simulation can be found in the comments in the code.

Listing 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; // rest density
float particleMass = 0.0002f; // particle mass
float viscosity = 0.1f; // Viscosity coefficient
float maxAllowableTimestep = 0.005f; // Time step width
float wallStiffness = 3000.0f; // Penalty wall power
int iterations = 4; // number of iterations
Vector2 gravity = new Vector2(0.0f, -0.5f);     // 重力
Vector2 range = new Vector2 (1, 1); // Simulation space
bool simulate = true; // Run or pause

int numParticles; // Number of particles
float timeStep; // Time step width
float densityCoef; // Poly6 kernel density coefficient
float gradPressureCoef; // Pressure coefficient of Spiky kernel
float lapViscosityCoef; // Laplacian kernel viscosity coefficient
```

Please note that in this demoscene, the inspector sets a value that is different from the parameter initialization value described in the code.

5.4.2 Calculation of coefficients of SPH weighting function

Since the coefficient of the weight function does not change during simulation, it is calculated on the CPU side at the time of initialization. (However, it is updated in the Update function in consideration of the possibility of editing the parameter during execution)

Since the mass of each particle is constant this time, the mass m in the formula of the physical quantity goes out of the sigma and becomes as follows.

```
\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 weight function, calculate the coefficient for each.

Listing 5.2: Pre-calculation of weight function coefficients (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 on the CPU side are stored in the constant buffer on the GPU side.

Listing 5.3: Transferring a Value to the Compute Shader's Constant Buffer (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);
```

Listing 5.4: Compute Shader constant buffer (SPH2D.compute)

```
int _NumParticles; // Number of particles
float _TimeStep; // Time step width (dt)
float _Smoothlen; // particle radius
float _PressureStiffness; // Becker's coefficient
float _RestDensity; // Rest density
float _DensityCoef; // Coefficient when calculating density
float _GradPressureCoef; // Coefficient when calculating pressure
float _LapViscosityCoef; // Coefficient when calculating viscosity
float _WallStiffness; // Pushing back force of the penalty method
float _Viscosity; // Viscosity coefficient
float2 _Gravity; // Gravity
float2 _Range; // Simulation space

float3 _MousePos; // Mouse position
float _MouseRadius; // Radius of mouse interaction
bool _MouseDown; // Is the mouse pressed?
```

5.4.3 Density calculation

Listing 5.5: Kernel function for calculating density (SPH2D.compute)

```
[numthreads(THREAD_SIZE_X, 1, 1)]
void DensityCS(uint3 DTid : SV_DispatchThreadID) {
uint P_ID = DTid.x; // Particle ID currently being processed

    float h_sq = _Smoothlen * _Smoothlen;
    float2 P_position = _ParticlesBufferRead[P_ID].position;

// Proximity exploration (O(n^2))
    float density = 0;
    for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
if (N_ID == P_ID) continue; // Avoid referencing itself

        float2 N_position = _ParticlesBufferRead[N_ID].position;

float2 diff = N_position-P_position; // particle distance
float r_sq = dot (diff, diff); // Particle distance squared

// Exclude particles that do not fit within the radius
            if (r_sq < h_sq) {
// No need to take a route as the calculation only includes the square
                    density += CalculateDensity(r_sq);
            }
    }

// Update density buffer
26:     _ParticlesDensityBufferWrite[P_ID].density = density;
27: }
```

Normally, it is necessary to search for nearby particles using an appropriate neighborhood search algorithm without searching for all particles, but in this implementation, 100% survey is performed for simplicity (for loop on line 10). .. Also, since the distance between you and the other particle is calculated, you avoid calculating between your own particles in the 11th line.

The case classification by the effective radius h of the weight function is realized by the if statement on the 19th line. Density addition (sigma calculation) is realized by adding the calculation result inside sigma to the variable initialized with 0 in the 9th line. Here is the formula for calculating the density again.

```
     \rho(\overrightarrow{x}) = \sum_{j \in N}m_jW_{poly6}(\overrightarrow{x_j} - \overrightarrow{x}, h)
```

The density is calculated using the Poly6 weighting function as shown in the above equation. The Poly6 weighting function is calculated in Listing 5.6 .

Listing 5.6: Density Calculation (SPH2D.compute)

```
inline float CalculateDensity(float r_sq) {
    const float h_sq = _Smoothlen * _Smoothlen;
     return _DensityCoef * (h_sq - r_sq) * (h_sq - r_sq) * (h_sq - r_sq);
}
```

Finally, line 25 of Listing 5.5 writes to the write buffer.

5.4.4 Calculation of pressure per particle

Listing 5.7: Weighting function (SPH2D.compute) to calculate pressure per particle

```
[numthreads(THREAD_SIZE_X, 1, 1)]
void PressureCS(uint3 DTid : SV_DispatchThreadID) {
uint P_ID = DTid.x; // Particle ID currently being processed

```

```
    float P_density = _ParticlesDensityBufferRead[P_ID].density;
    float  P_pressure = CalculatePressure(P_density);

// Update pressure buffer
    _ParticlesPressureBufferWrite[P_ID].pressure = P_pressure;
}
```

Before solving the pressure term, calculate the pressure for each particle to reduce the calculation cost of the pressure term later. As I mentioned earlier, pressure calculation originally requires solving an equation called Poisson's equation, such as the following equation.

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

However, the operation of solving Poisson's equation accurately with a computer is very expensive, so it is calculated approximately using the Tait equation below.

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

Listing 5.8: Implementation of the Tait equation (SPH2D.compute)

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

5.4.5 Calculation of pressure term and viscosity term

Listing 5.9: Kernel function to calculate pressure and viscosity terms (SPH2D.compute)

```
[numthreads(THREAD_SIZE_X, 1, 1)]
void ForceCS(uint3 DTid : SV_DispatchThreadID) {
uint P_ID = DTid.x; // Particle ID currently being processed

    float2 P_position = _ParticlesBufferRead[P_ID].position;
    float2 P_velocity = _ParticlesBufferRead[P_ID].velocity;
    float P_density = _ParticlesDensityBufferRead[P_ID].density;
    float P_pressure = _ParticlesPressureBufferRead[P_ID].pressure;

    const float h_sq = _Smoothlen * _Smoothlen;

// Proximity exploration (O(n^2))
float2 press = float2 (0, 0);
    float2 visco = float2(0, 0);
    for (uint N_ID = 0; N_ID < _NumParticles; N_ID++) {
if (N_ID == P_ID) continue; // Skip if targeting itself

        float2 N_position = _ParticlesBufferRead[N_ID].position;

        float2 diff = N_position - P_position;
        float r_sq = dot(diff, diff);

// Exclude particles that do not fit within the radius
        if (r_sq < h_sq) {
            float  N_density
= _ParticlesDensityBufferRead[N_ID].density;
            float  N_pressure
= _ParticlesPressureBufferRead[N_ID].pressure;
            float2 N_velocity
            = _ParticlesBufferRead[N_ID].velocity;
            float  r = sqrt(r_sq);

// Pressure item
            press += CalculateGradPressure(...);

// Sticky items
            visco += CalculateLapVelocity(...);
        }
    }

// Integration
    float2 force = press + _Viscosity * visco;

// Acceleration buffer update
     _ParticlesForceBufferWrite[P_ID].acceleration = force / P_density;
}
```

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

First, the force is calculated by the following pressure term on the 31st line.

```
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 calculation of the contents of Sigma is performed by the following function.

Listing 5.10: Calculation of Pressure Term Elements (SPH2D.compute)

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

Next, the force is calculated by the following viscosity term 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 calculation of the contents of Sigma is performed by the following function.

Listing 5.11: Calculation of Viscosity Term Elements (SPH2D.compute)

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

Finally, on line 39 of Listing 5.9 , the forces calculated by the pressure and viscosity terms are added and written to the buffer as the final output.

5.4.6 Collision detection and position update

Listing 5.12: Kernel function for collision detection and position update (SPH2D.compute)

```
[numthreads(THREAD_SIZE_X, 1, 1)]
void IntegrateCS(uint3 DTid : SV_DispatchThreadID) {
const unsigned int P_ID = DTid.x; // Particle ID currently being processed

// Position and speed before update
    float2 position = _ParticlesBufferRead[P_ID].position;
    float2 velocity = _ParticlesBufferRead[P_ID].velocity;
    float2 acceleration = _ParticlesForceBufferRead[P_ID].acceleration;

// Mouse interaction
    if (distance(position, _MousePos.xy) < _MouseRadius && _MouseDown) {
        float2 dir = position - _MousePos.xy;
        float pushBack = _MouseRadius-length(dir);
        acceleration += 100 * pushBack * normalize(dir);
    }

// Here to write collision detection -----

// Wall boundary (penalty method)
    float dist = dot(float3(position, 1), float3(1, 0, 0));
    acceleration += min(dist, 0) * -_WallStiffness * float2(1, 0);

    dist = dot(float3(position, 1), float3(0, 1, 0));
    acceleration += min(dist, 0) * -_WallStiffness * float2(0, 1);

    dist = dot(float3(position, 1), float3(-1, 0, _Range.x));
    acceleration += min(dist, 0) * -_WallStiffness * float2(-1, 0);

    dist = dot(float3(position, 1), float3(0, -1, _Range.y));
    acceleration += min(dist, 0) * -_WallStiffness * float2(0, -1);

// Addition of gravity
    acceleration += _Gravity;

// 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: }
```

Collision detection with a wall is performed using the penalty method (lines 19-30). The penalty method is a method of pushing back with a strong force as much as it protrudes from the boundary position.

Originally, collision detection with obstacles is also performed before collision detection with a wall, but in this implementation, interaction with the mouse is performed (lines 213-218). If the mouse is pressed, the specified force is applied to move it away from the mouse position.

Gravity, which is an external force, is added on the 33rd line. Setting the gravity value to zero results in weightlessness and interesting visual effects. Also, update the position using the forward Euler method described above (lines 36-37) and write the final result to the buffer.

5.4.7 Simulation main routine

Listing 5.13: Simulation key functions (FluidBase.cs)

```
private void RunFluidSolver() {

  int kernelID = -1;
  int threadGroupsX = numParticles / THREAD_SIZE_X;

  // Density
  kernelID = fluidCS.FindKernel("DensityCS");
  fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
  fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferWrite", ...);
  fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);

  // Pressure
  kernelID = fluidCS.FindKernel("PressureCS");
  fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferRead", ...);
```

```
    fluidCS.SetBuffer(kernelID, "_ParticlesPressureBufferWrite", ...);
    fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);

    // Force
    kernelID = fluidCS.FindKernel("ForceCS");
    fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
    fluidCS.SetBuffer(kernelID, "_ParticlesDensityBufferRead", ...);
    fluidCS.SetBuffer(kernelID, "_ParticlesPressureBufferRead", ...);
    fluidCS.SetBuffer(kernelID, "_ParticlesForceBufferWrite", ...);
    fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);

    // Integrate
    kernelID = fluidCS.FindKernel("IntegrateCS");
    fluidCS.SetBuffer(kernelID, "_ParticlesBufferRead", ...);
    fluidCS.SetBuffer(kernelID, "_ParticlesForceBufferRead", ...);
    fluidCS.SetBuffer(kernelID, "_ParticlesBufferWrite", ...);
    fluidCS.Dispatch(kernelID, threadGroupsX, 1, 1);

    SwapComputeBuffer(ref particlesBufferRead, ref particlesBufferWrite);
}
```

This is the part that calls the Compute Shader kernel function described so far every frame. Give the appropriate ComputeBuffer for each kernel function.

Now, remember that the smaller the time step width \ Delta t , the less error the simulation will have. When running at 60FPS, \ Delta t = 1/60 , but this causes a large error and the particles explode. Furthermore, if the time step width is smaller than \ Delta t = 1/60, the time per frame will advance slower than the real time, resulting in slow motion. To avoid this, set \ Delta t = 1 / (60 \ times {iterarion}) and run the main routine iterarion times per frame.

Listing 5.14: Major Function Iteration (FluidBase.cs)

```
// Reduce the time step width and iterate multiple times to improve the calculation accuracy.
for (int i = 0; i<iterations; i++) {
    RunFluidSolver();
}
```

This allows you to perform real-time simulations with a small time step width.

5.4.8 How to use the buffer

Unlike a normal single-access particle system, particles interact with each other, so it is a problem if other data is rewritten during the calculation. To avoid this, prepare two buffers, a read buffer and a write buffer, which do not rewrite the value when performing calculations on the GPU. By swapping these buffers every frame, you can update the data without conflict.

Listing 5.15: Buffer Swapping Function (FluidBase.cs)

```
void SwapComputeBuffer(ref ComputeBuffer ping, ref ComputeBuffer pong) {
    ComputeBuffer temp = ping;
    ping = pong;
    pong = temp;
}
```

5.4.9 Particle rendering

Listing 5.16: Rendering Particles (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 containing the position calculation result of the fluid particle in the material and transfer it to the shader. On the 11th line, we are instructing to draw instances for the number of particles.

Listing 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 or = (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: }
```

Lines 1-6 define the information for receiving 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 by referring to the buffer element with id: SV_VertexID as shown in the 14th line.

After that, as with a normal particle system , create a billboard [* 10] centered on the position data of the calculation result with the geometry shader as shown in Fig. 5.7 , and attach and render the particle image.

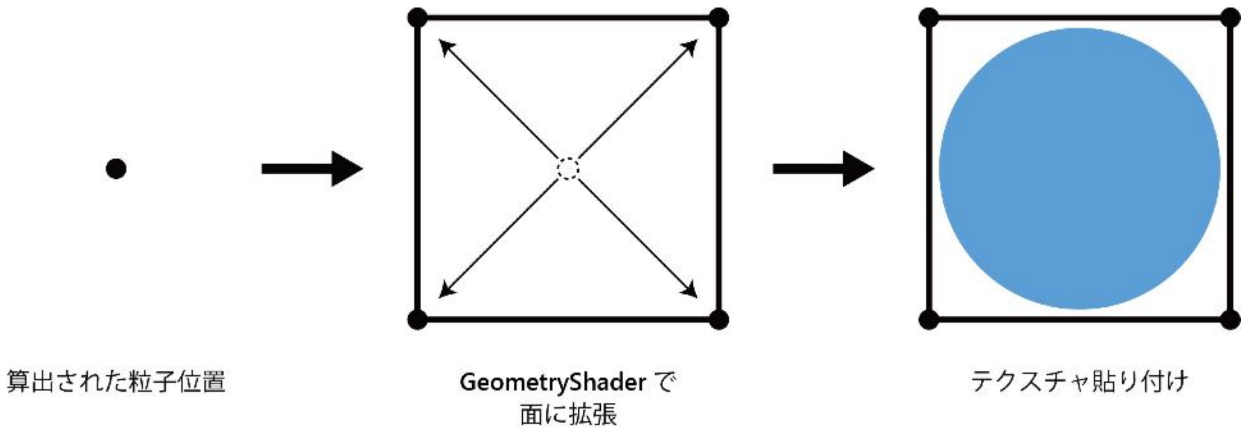


Figure 5.7: Creating a billboard

[* 10] A Plane whose table always faces the viewpoint.

5.5 Results

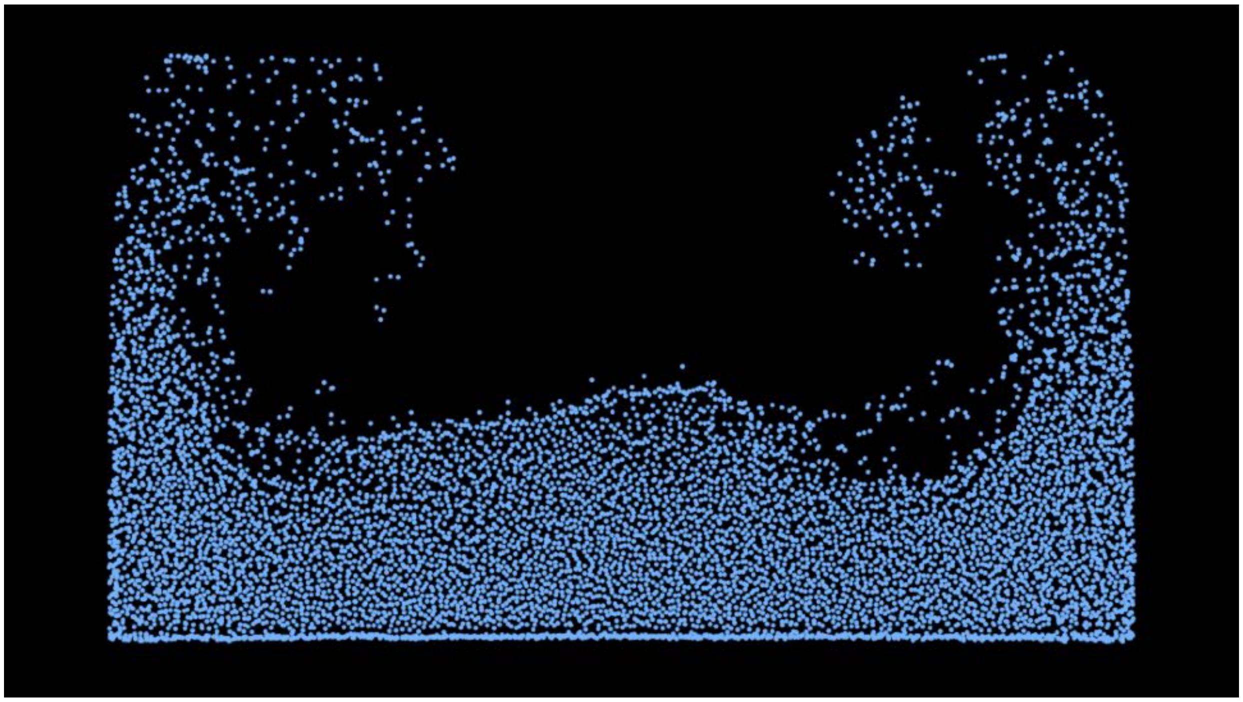


Figure 5.8: Rendering result

The video is posted here (https://youtu.be/KJVu26zeK2w).

5.6 Summary

In this chapter, the method of fluid simulation using the SPH method is shown. By using the SPH method, it has become possible to handle the movement of fluid as a general purpose like a particle system.

As mentioned earlier, there are many types of fluid simulation methods other than the SPH method. Through this chapter, we hope that you will be interested in other physics simulations themselves in addition to other fluid simulation methods, and expand the range of expressions.

Chapter 6 Growing grass with geometry shaders

6.1 Introduction

This chapter mainly describes Geometry Shader, which is one of the stages of the rendering pipeline, and explains the dynamic grass-generating shader (commonly known as Grass Shader) using Geometry Shader.

I've used some technical terms to describe the Geometry Shader, but if you're just trying to use the Geometry Shader, it's a good idea 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とは?

Geometry Shader is one of the programmable shaders that can dynamically convert, generate, and delete primitives (basic shapes that make up a mesh) on the GPU.

Until now, if you try to change the mesh shape dynamically, such as by converting primitives, you need to take measures such as processing on the CPU or giving meta information to the vertices in advance and converting with Vertex Shader. did. However, Vertex Shader cannot acquire information about adjacent vertices, and there are strong restrictions such as not being able to create new vertices based on the vertices being processed and vice versa. .. However, processing with a CPU would take an unrealistically huge amount of time from the perspective of real-time processing. As you can see, there have been some problems with changing the shape of the mesh in real time.

Therefore, Geometry Shader is installed as standard in DirectX 10 and OpenGL 3.2 as a function to solve these problems and enable free conversion processing within weak constraints. In OpenGL, it is also called Primitive Shader.

6.3 Features of Geometry Shader

6.3.1 Rendering pipeline

It is located on the rendering pipeline after Vertex Shader and before Fragment Shader and rasterization. In other words, within the Fragment Shader, the vertices dynamically generated by the Geometry Shader and the original vertices passed to the Vertex Shader are processed without distinction.

6.3.2 Input to Geometry Shader

Normally, the input information to Vertex Shader is in units of vertices, and conversion processing is performed for those vertices. However, the input information to the Geometry Shader is a user-defined input primitive unit.

The actual program will be 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 line, two vertex information will be passed, and if 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 vertex shader, and enables a wide range of calculations.

One thing to note is that Vertex Shader processes on a vertex-by-vertex basis and passes information about the vertices it processes, but Geometry Shader is a primitive assembly topology regardless of the input primitive type. Processing is performed in units of primitives determined by. In other words, if you run the Geometry Shader on a Quad mesh with a topology of Triangles, as shown in Figure 6.1, the Geometry Shader will be run twice for triangles ① and ②. At this time, when the primitive type for input is Line, the information passed to the input is the vertices of two vertices 0,1,2 in the case of triangle ①, and the vertices 0,2,3 in the case of ②. It will be the apex of the two points.

Quadメッシュ

トポロジ・・・三角形

三角形①、②

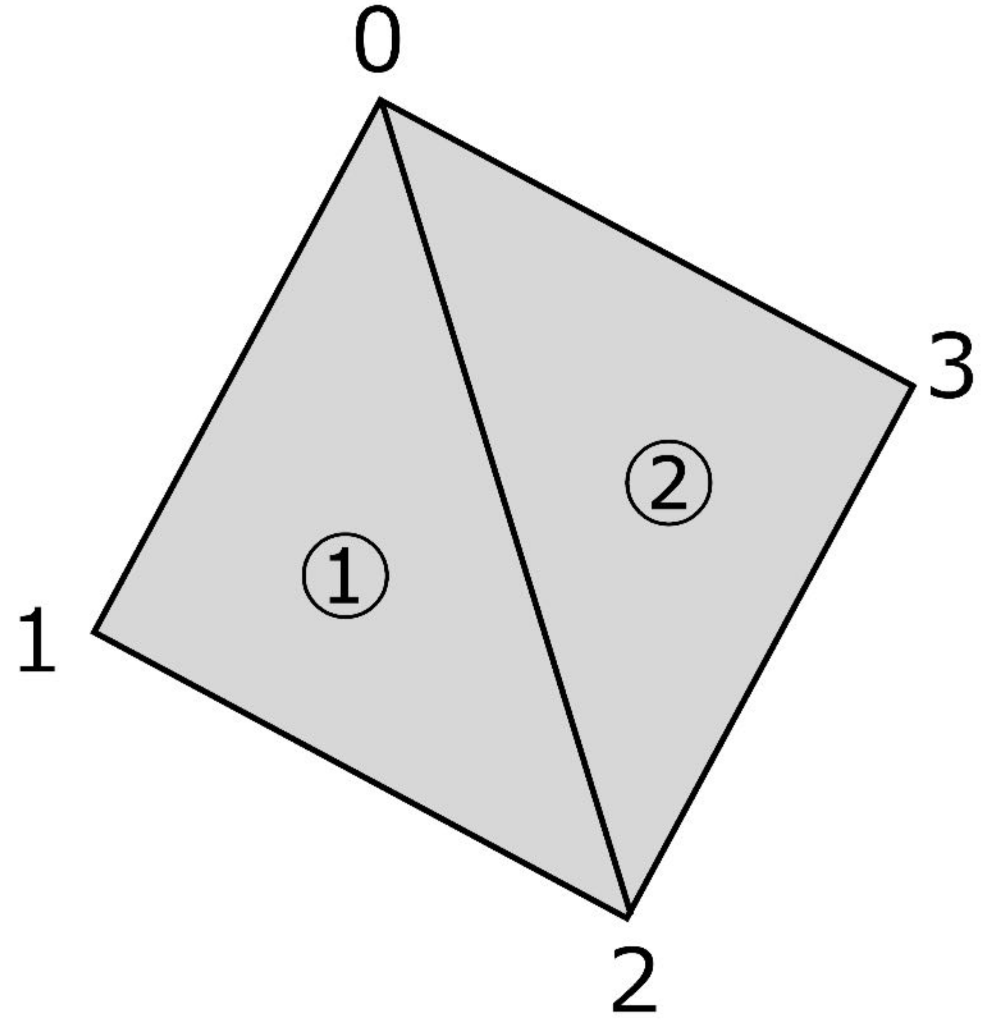


Figure 6.1: Quad mesh

6.3.3 Output from Geometry Shader

The output of Geometry Shader is a set of vertex information for user-defined output primitive types. In Vertex Shader, it was 1 input and 1 output, but Geometry Shader will output multiple information, and there is no problem even if there is one or more primitives generated by the output information.

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

In addition, it is necessary to set in advance the maximum number of vertices to be output in one process called MaxVertexCount in Geometry Shader. For example, if MaxVertexCount is set to 9, Geometry Shader will be able to output the number of vertices from 0 to 9 points. Due to the "Geometry Shader Limits" described later, 1024 is generally the maximum value for this value.

In addition, as a point to be careful when outputting vertex information, when adding a new vertex while maintaining the original mesh shape, the vertex information sent from Vertex Shader is also sent to Geometry Shader. Must be output. The Geometry Shader does not have 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 dynamically reduce the number of vertices by setting the output of the Geometry Shader to 0.

6.3.4 Geometry Shader Limits

The Geometry Shader has a maximum number of output vertices and a 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 depends on the GPU, 1024 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 have, such as coordinates and colors. Generally, the position elements of (x, y, z, w) and (r, g, b, a) There are a total of 8 color elements. The maximum number of outputs of this element also depends on the GPU, but since 1024 is also common, the output will be limited to 128 (1024/8) at the maximum.

Since both of these restrictions must be met, even if the number of vertices can be output at 1024 points, the actual output of the Geometry Shader is limited to 128 points due to restrictions on the number of elements. So, for example, if you use Geometry Shader for a mesh with 2 primitives (Quad mesh, etc.), you can handle only up to 256 vertices (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 Simple Geometry Shader

Below is a Geometry Shader program with simple behavior. I will explain the explanation up to the previous section again by comparing it with the actual program.

In addition to Geometry Shader, the explanation about ShaderLab syntax etc. 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 and moved further upward, and each vertex of the passed triangle is connected to the calculated new coordinates. In other words, we are generating a simple triangular pyramid from a flat triangle.

So if you apply this shader to a Quad mesh (consisting of two triangles), it will look like Figures 6.2 through 6.3.

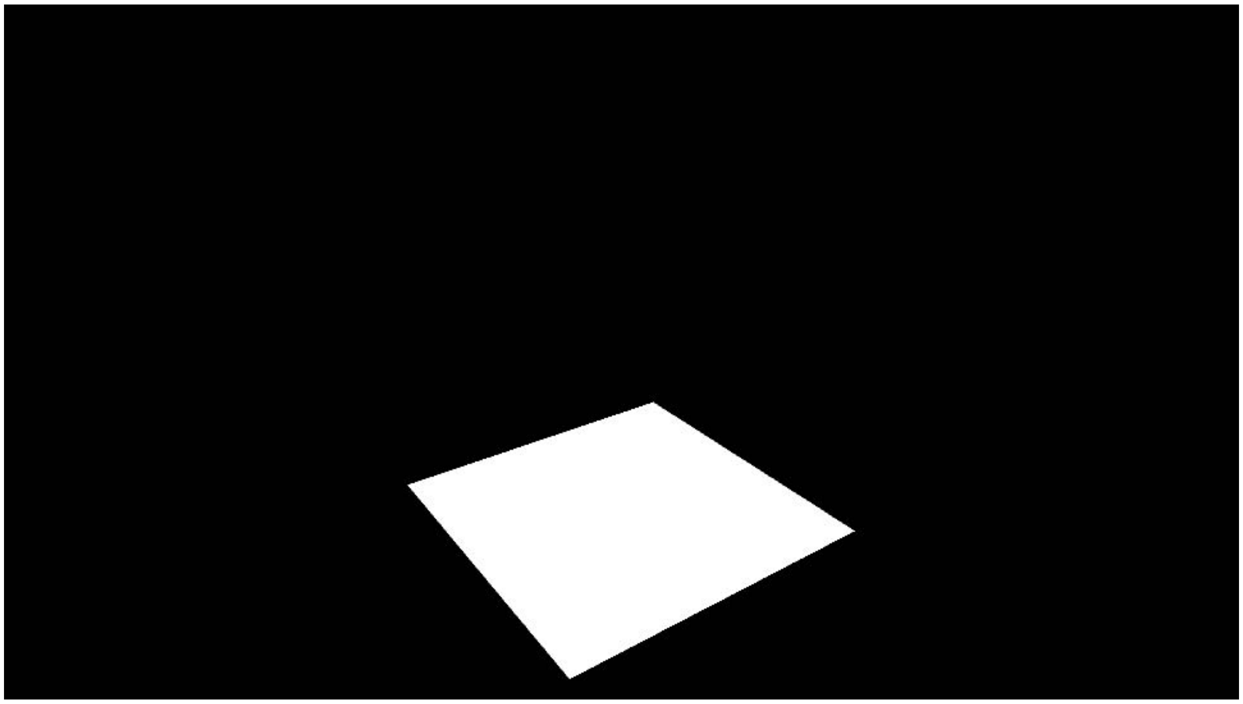


Figure 6.2: From a flat plate like this

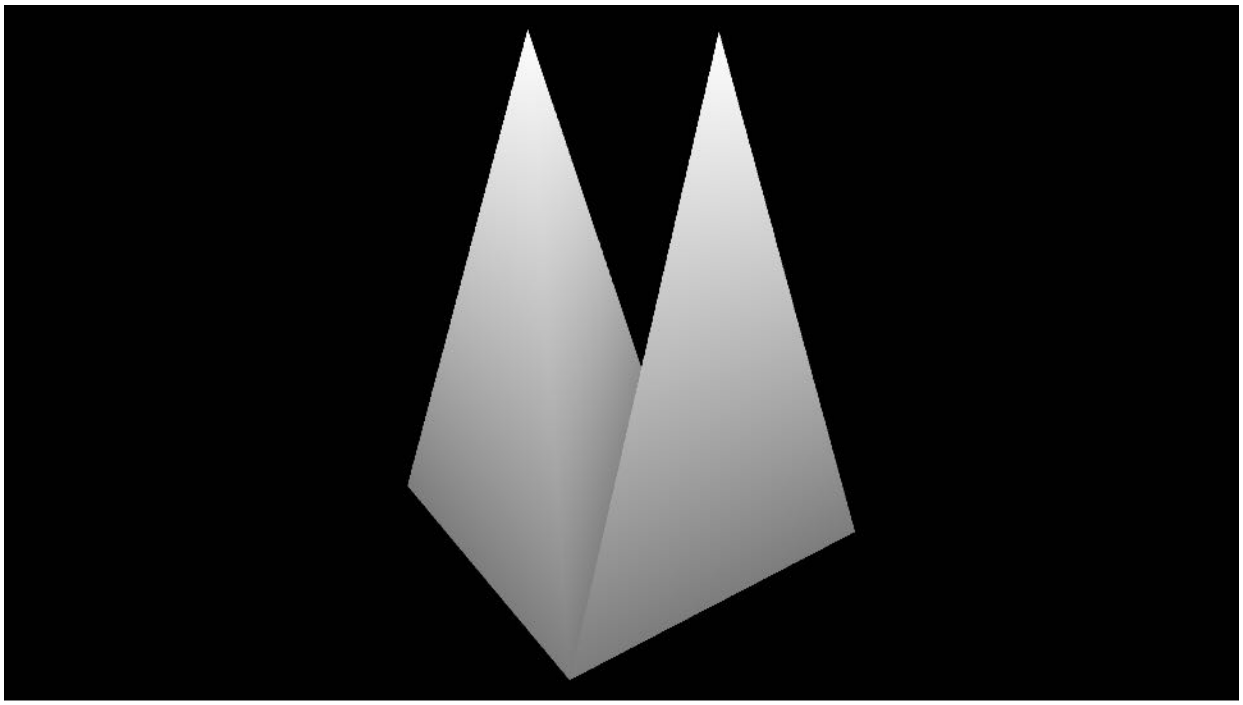


Figure 6.3: Two three-dimensional triangular pyramids are now displayed

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

```
#pragma target 5.0
#pragma vertex vert

// Declare the use of Geometry Shader
#pragma geometry geom

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

In the above declaration part, `geom`we declare that the function named is a function for Geometry Shader. This `geom`will cause the function to be called when the Geometry Shader stage is reached .

```
[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.

This time, I want to generate a triangular pyramid based on the triangle, so I input `triangle`it. As a result, the information of each vertex of the triangle, which is the unit primitive, is input, and since the triangle is composed of three vertices, the received formal argument is an array of length 3. So, if the input `triangle`is not input , `point`only one vertex will be composed, so `geom(point v2f input[1])`it will be received as an array of length 1 like.

6.4.2 Output

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

This is the part related to output.

Since we want to make the primitive of the mesh generated this time a triangle, `TriangleStream`we declare it with a type. `TriangleStrema`Since the type means that the output is a triangle strip, it will generate a triangle based on each output vertex information. There are other `PointStream`types and `LineStream`types, so you need to select the output primitive type according to your purpose.

In addition, `[maxvertexcount(12)]`the maximum number of outputs is set to 12 in the part. This is because the number of triangles that make up the triangular pyramid is one at the base and three at the side, for a total of four, and three vertices are required for each triangle, so 12 vertices are output with 3 * 4. 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, a g2f type variable for output is declared, and vertex coordinates and color information are stored. At this time, it is necessary to convert from the object space to the clip space of the camera in the same way as Vertex Shader.

After that, the vertex information is output while being aware of the order of the vertices that make up the mesh. `outStream`Of the variable `Append`will be added to the current stream by passing the output variable to the function, `RestartStrip`to end the current primitive strip by calling the function, you have to start a new stream.

Since this is a `TriangleStream`triangle strip, the more `Append`vertices you add in the function, the more connected triangles will be generated based on all the vertices added to the stream. So, `Append`if `RestartStrip`you don't want to be connected based on the order in which the triangles are connected like this time, you need to call once to start a new stream. Of course, it is possible to reduce the number `Append`of `RestartStrip`function calls by devising the order .

6.5 Grass Shader

In this section, we will explain Grass Shader, which is a little development from the previous section "Simple Geometry Shader", and uses Geometry Shader to generate grass in real time.

The following is the Grass Shader program described.

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

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

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

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

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

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

        // Noise texture that gives randomness to grass height
        _HeightMap("Height Map", 2D) = "white"
              //  Noise  texture  that  gives  randomness  to  the
orientation of the 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 hey: 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;
                or [6] .color = _TopColor;

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

                o [2] .pos + =
                            * (_WindPower * wind * _BottomBend)
                            * sin(_Time);
                o [3] .pos + =
                            * (_WindPower * wind * _BottomBend)
                            * sin(_Time);
                o [4] .pos + =
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o [5] .pos + =
                            * (_WindPower * wind * _MiddleBend)
                            * sin(_Time);
                o [6] .pos + =
                            * (_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
        }
    }
}
```

If you apply this shader to a Plane mesh with multiple vertical and horizontal arrangements, it will look like Figure 6.4.

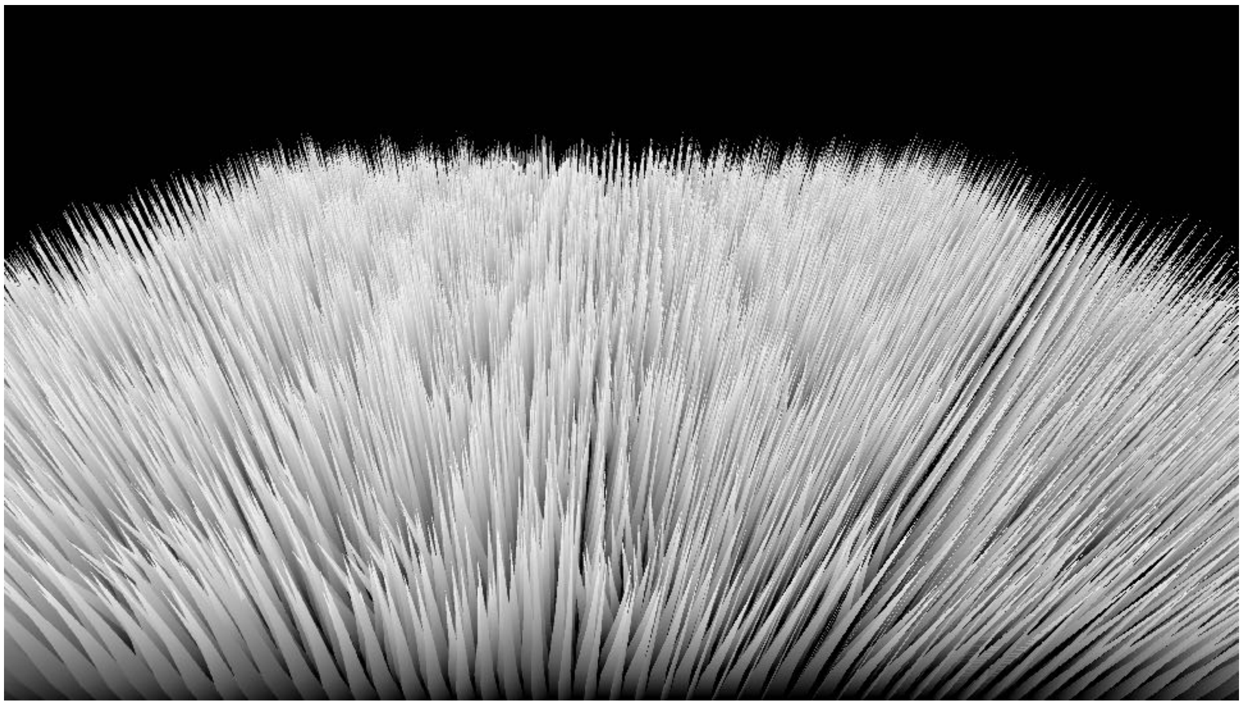


Figure 6.4: Grass Shader results

I will explain the process of growing grass from this.

6.5.1 Basic policy

This time, we will generate one grass for each primitive. As shown in Fig. 6.5, the shape of the grass is divided into the lower part, the middle part, and the upper part, and a total of 7 vertices are generated. I will.

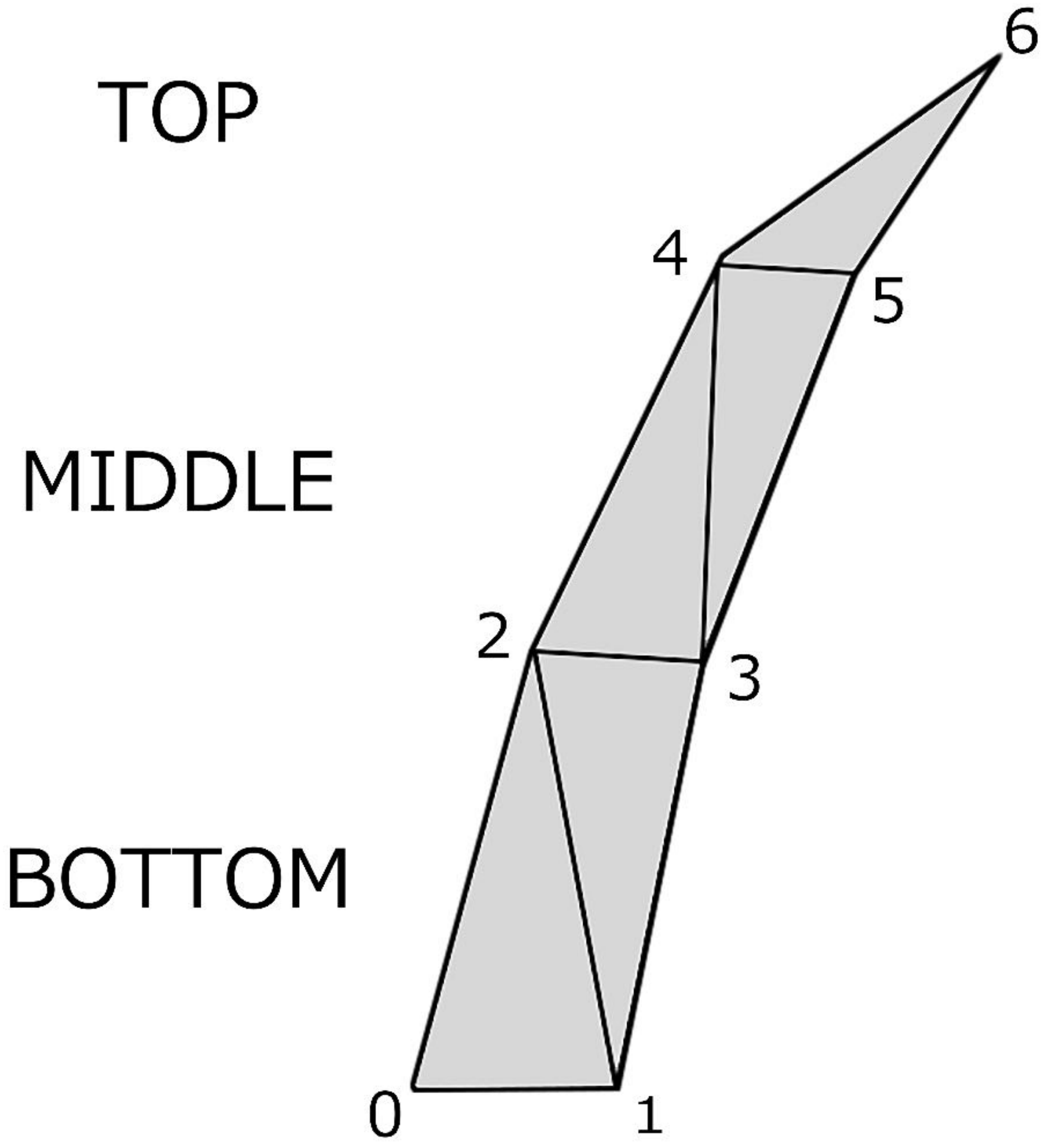


Figure 6.5: How to make a grass shape

6.5.2 Parameters

Details are described in the comments, but the coefficient that controls the width and height of each part (lower part, middle part, upper part) in one grass, and the coefficient that controls the width and height of the whole grass It is prepared as the main parameter. Also, it doesn't look good if each grass has the same shape, so we 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, the height and direction of the grass and the numerical values that are the standard of the strength of the wind are calculated. You can calculate in Geometry Shader, but if you give the vertices meta information, you can treat it like the initial value when performing calculation on Geometry Shader, so calculate with Vertex Shader. I am.

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

Here, the central part of the grass and the direction in which the grass grows are calculated. If you decide this part by noise texture etc., you can give randomness in the direction of grass growth.

```
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]);
}
```

There are 7 vertices calculated in this part `Append`. This time, there is no problem even if the triangles are generated while being connected, so I have not done so `RestartStrip`.

In addition, the attribute called is applied to the `for`statement `[unroll]`. This is an attribute that expands the processing in the loop as many times as the number of loops at compile time, and although it has the disadvantage of increasing the memory size, it has the advantage of operating at high speed.

6.6 Summary

So far, we have explained from the explanation of Geometry Shader to the basic and applied programs. There are some features that are slightly different from writing a program that runs on the CPU, but you should be able to utilize it if you suppress the basic part.

In fact, it is generally said that Geometry Shader is slow. I haven't really felt it, but it may be difficult when the range of use is large. If you are going to use Geometry Shader on a large scale, please take a benchmark etc. once.

Still, being able to dynamically and freely create and delete new meshes on the GPU will greatly expand 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 that you will learn about and learn about one tool called Geometry Shader in this chapter, and feel some new possibilities.

6.7 Reference

- Tutorial 13: Geometry Shader-https: //msdn.microsoft.com/en-us/library/bb172497
- Geometry shader object in MSDN-https: //msdn.microsoft.com/en-us/library/ee418313
- Rendering technique for transparent geometry by cutting geometry in geometry shader-http: //t-pot.com/program/147_CGGONG2008/index.html

Chapter 7 Introduction to the Marching Cubes Method Starting with 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 expired in 2005, it is now free to use.

7.1.2 Explanation of simple mechanism

First, divide the volume data space with a 3D grid.

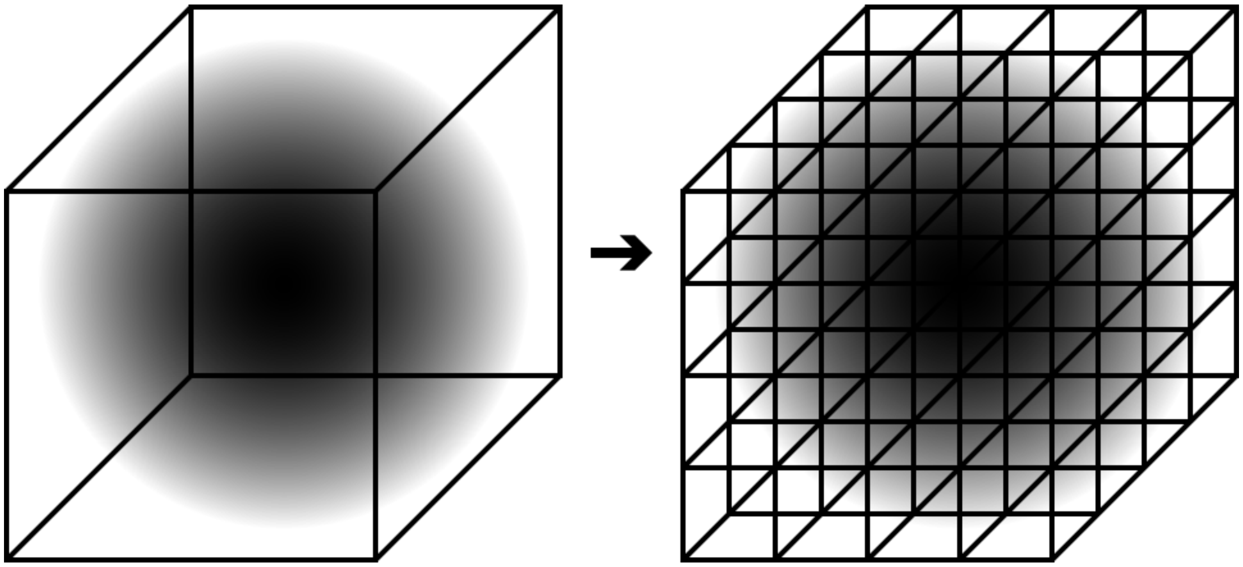


Figure 7.1: 3D volume data and grid partitioning

Next, let's take out one of the divided grids. The boundaries of the eight vertices are calculated as 1 if the values of the eight corners of the grid are above the threshold and 0 if they are below the threshold.
The figure below shows the flow when the threshold is set to 0.5.

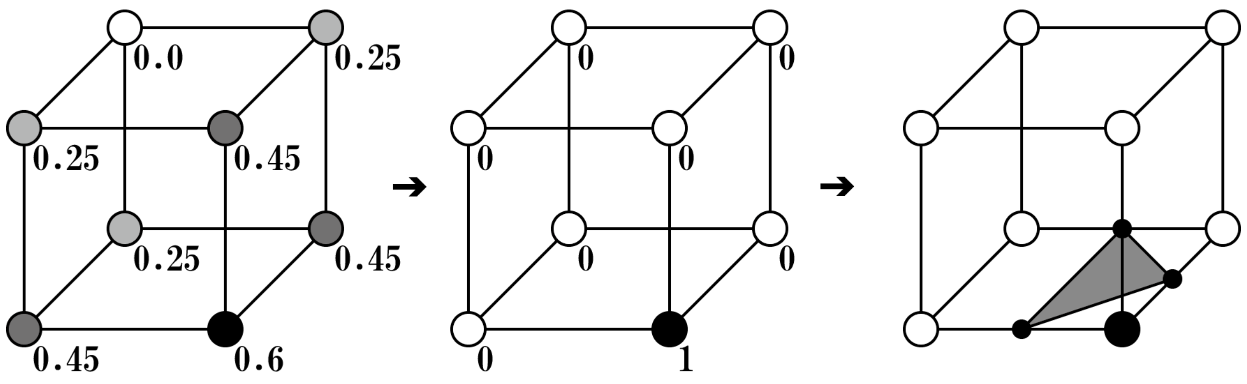


Figure 7.2: Determining the boundary according to the value of the angle

There are 256 types of combinations of the eight corners, but if you make full use of rotation and inversion, it will fit in 15 types. Assign triangular polygon patterns corresponding to the 15 types of combinations.

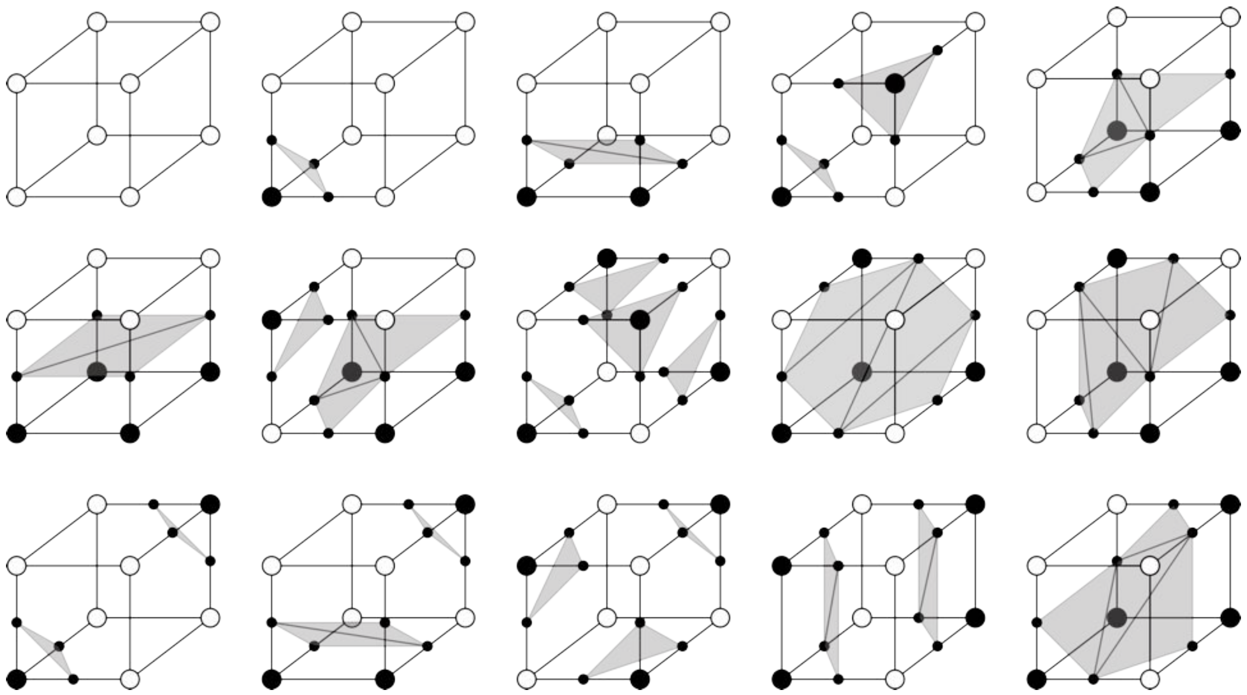


Figure 7.3: Combination of corners

7.2 Sample repository

The sample projects described in this chapter can be found in Assets / GPU ArchingCubes under Unity Graphics Programming's Unity Project https://github.com/IndieVisualLab/UnityGraphicsProgramming.

For implementation, I ported it to Unity by referring to Paul Bourke's Polygonising a scalar field site [* 1] .

[* 1] Polygonising a scalar field http://paulbourke.net/geometry/polygonise/

This time, I will explain along with this sample project.

There are three main implementations.

- Initialization of mesh, drawing registration process for each frame (C # script part)
- Initialization of Compute Buffer
- Actual drawing process (shader part)

First, create from the **GPU ArchingCubesDrawMesh** class that initializes the mesh and registers the drawing .

7.2.1 Make a mesh for Geometry Shader

As explained in the previous section, the Marching cubes method is an algorithm that generates polygons by combining the eight corners of the grid. To do that in real time, you need to dynamically create polygons.
However, it is inefficient to generate a mesh vertex array on the CPU side (C # side) every frame.
So we use Geometry Shader. GeometryShader is a Shader located between VertexShader and FragmentShader, which can increase or decrease the number of vertices processed by VertexShader.
For example, you can add 6 vertices around one vertex to generate a plate polygon.
Furthermore, it is very fast because it is processed on the Shader side (GPU

side).
This time, I will use Geometry Shader to generate and display Marching Cubes polygons.

First, define the variables used in the **GPUMarchingCubesDrawMesh** class.

Listing 7.1: Definition of variables

```
using UnityEngine;

public class GPUMarchingCubesDrawMesh : MonoBehaviour {

    #region public
     public int segmentNum = 32; // Number of divisions on one side of the grid

    [Range(0,1)]
     public float threashold = 0.5f; // Threshold for the scalar value to mesh
    public Material mat; // Material for rendering

    public Color DiffuseColor = Color.green; // Diffuse color
    public Color EmissionColor = Color.black; // Emission color
    public float EmissionIntensity = 0; // Emission intensity

    [Range(0,1)]
    public float metallic = 0; // metallic feeling
    [Range(0, 1)]
    public float glossiness = 0.5f; // Glossiness
    #endregion

     #region private
    int vertexMax = 0; // number of vertices
    Mesh[] meshs = null;                         // Mesh配列
     Material [] materials = null; // Material array for each mesh
    float renderScale = 1f / 32f; // Display scale
     MarchingCubesDefines mcDefines = null; // Constant array group for MarchingCubes
    #endregion

}
```

Next, create a mesh to pass to the Geometry Shader. The vertices of the mesh should be placed one by one in the divided 3D grid. For example, if the number of divisions on one side is 64, 64 * 64 * 64 = 262,144 vertices are required.

However, in Unity2017.1.1f1, 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.

Listing 7.2: Meshing part

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log(" VertexMax " + vertexMax);

   // Divide the size of 1Cube by segmentNum to determine the
size at the time of rendering
  renderScale = 1f / segmentNum;

  CreateMesh();

  // Initialize constant array for Marching Cubes used in shader
  mcDefines = new MarchingCubesDefines();
}

void CreateMesh()
{
   // Since the maximum number of vertices of Mesh is 65535,
divide 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];

  // Mesh bounce calculation
  Bounds bounds = new Bounds(
    transform.position,
         new Vector3(segmentNum, segmentNum, segmentNum) *
renderScale
  );
  int id = 0;
  for (int i = 0; i < meshNum; i++)
  {
    // Vertex creation
    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 creation
    meshs[i] = new Mesh();
    meshs[i].vertices = vertices;
     // Mesh Topology can be 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 Initialization of Compute Buffer

The source **MarchingCubesDefinces.cs** defines a constant array used for rendering the Marching Cubes method and a ComputeBuffer for passing the constant array to the shader. ComputeBuffer is a buffer that stores data used by shaders. Since the data is stored in the memory on the GPU side, it is quickly accessible from the shader.

In fact, the constant array used in the rendering of the Marching Cubes method can be defined on the shader side. However, the reason why the constant array used in 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 only be registered up to 4096. If you define a huge array of constants in your shader, you will quickly reach the upper limit of 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. Therefore, the number of processes increases a little, but on the C # side, the constant array is stored in ComputeBuffer and passed to the shader.

Listing 7.3: ComputeBuffer initialization part

```
void Initialize()
{
  vertexMax = segmentNum * segmentNum * segmentNum;

  Debug.Log(" VertexMax "  + vertexMax);

   // Divide the size of 1Cube by segmentNum to determine the
size at the time of rendering
  renderScale = 1f / segmentNum;

  CreateMesh();

  // Initialize constant array for Marching Cubes used in shader
  mcDefines = new MarchingCubesDefines();
}
```

In the Initialize () function mentioned earlier, MarchingCubesDefines is initialized.

7.2.3 Rendering

Next is the function that calls the rendering process.
This time, I'll use Graphics.DrawMesh () to render multiple meshes at once and to be affected by Unity's lighting. The meaning of DiffuseColor etc. defined in the public variable will be explained in the explanation on the shader side.

The ComputeBuffers of the MarchingCubesDefines class in the previous section are passed to the shader with material.setBuffer.

Listing 7.4: Rendered 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

Listing 7.5: Calling Part

```
// Use this for initialization
void Start ()
{
  Initialize();
}

void Update()
{
  RenderMesh ();
}
```

Start () calls Initialize () to generate a mesh, and the Update () function calls RenderMesh () to render.
The reason for calling RenderMesh () with Update () is that Graphics.DrawMesh () does not draw immediately, but it feels like "registering for rendering process once".
By registering, Unity will adapt the lights and shadows. A similar function is Graphics.DrawMeshNow (), but it draws instantly, so Unity lights and shadows are not applied. Also, you need to call it with OnRenderObject () or OnPostRender () instead of Update ().

7.4 Shader side implementation

The shader this time is roughly divided into two **parts, the " rendering part of the entity"** and the "**rendering part of the shadow**" . In addition, three shader functions are executed within each, the vertex shader, the geometry shader, and the fragment shader.

Since the shader source is long, I will have the sample project look at the entire implementation, and I will explain only the important points. The shader file described is GPU ArchingCubesRenderMesh.shader.

7.4.1 Variable declaration

At the top of the shader, we define the structure used for rendering.

Listing 7.6: Structure Definition Part

```
// 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
```

```
when rendering the entity
struct g2f_light
{
  float4 pos: SV_POSITION; // Local coordinates
  float3 normal: NORMAL; // normal
  float4 worldPos: TEXCOORD0; // World coordinates
  half3 sh        : TEXCOORD3;    // SH
};

// Data passed from the geometry shader to the fragment shader
when rendering shadows
struct g2f_shadow
{
  float4 pos: SV_POSITION; // coordinate
  float4 hpos     : TEXCOORD1;
};
```

Next, we are defining variables.

Listing 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 in the MarchingCubesDefines class have changed their type names to StructuredBuffer <○○>.

7.4.2 Vertex shader

The vertex shader is very simple, as most of the work is done by the geometry shader. It simply passes the vertex data passed from the mesh to the geometry shader as is.

Listing 7.8: Vertex shader implementation

```
// 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; // coordinate
};

// Vertex shader
v2g vert(appdata v)
{
  v2g or = (v2g) 0;
  o.pos = v.vertex;
  return o;
}
```

By the way, the vertex shader is common to the entity and the shadow.

7.4.3 Entity Geometry Shader

Since it is long, I will explain it while dividing it.

Listing 7.9: Function declaration part of the geometry shader

```
// Entity geometry shader
[maxvertexcount (15)] // Definition of the maximum number of
vertices output from the shader
void geom_light(point v2g input[1],
                inout TriangleStream<g2f_light> outStream)
```

First is the declaration part of the geometry shader.

`[maxvertexcount(15)]`Is the definition of the maximum number of vertices output from the shader. With the algorithm of the Marching Cubes method this time, a maximum of 5 triangular polygons can be created per grid, so a total of 15 vertices are output in 3 * 5.
Therefore, write 15 in () of maxvertexcount.

Listing 7.10: Scalar value acquisition part of the eight corners of the grid

```
float cubeValue [8]; // Array for getting scalar values at the
eight corners of the grid

// Get the scalar values for 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 the grid space when creating the mesh. As the name implies, vertexOffset is an array of offset coordinates added to pos.

This loop gets the scalar values in the volume data of the coordinates of the eight corners of one vertex = one grid. vertexOffset points to the order of the corners of the grid.

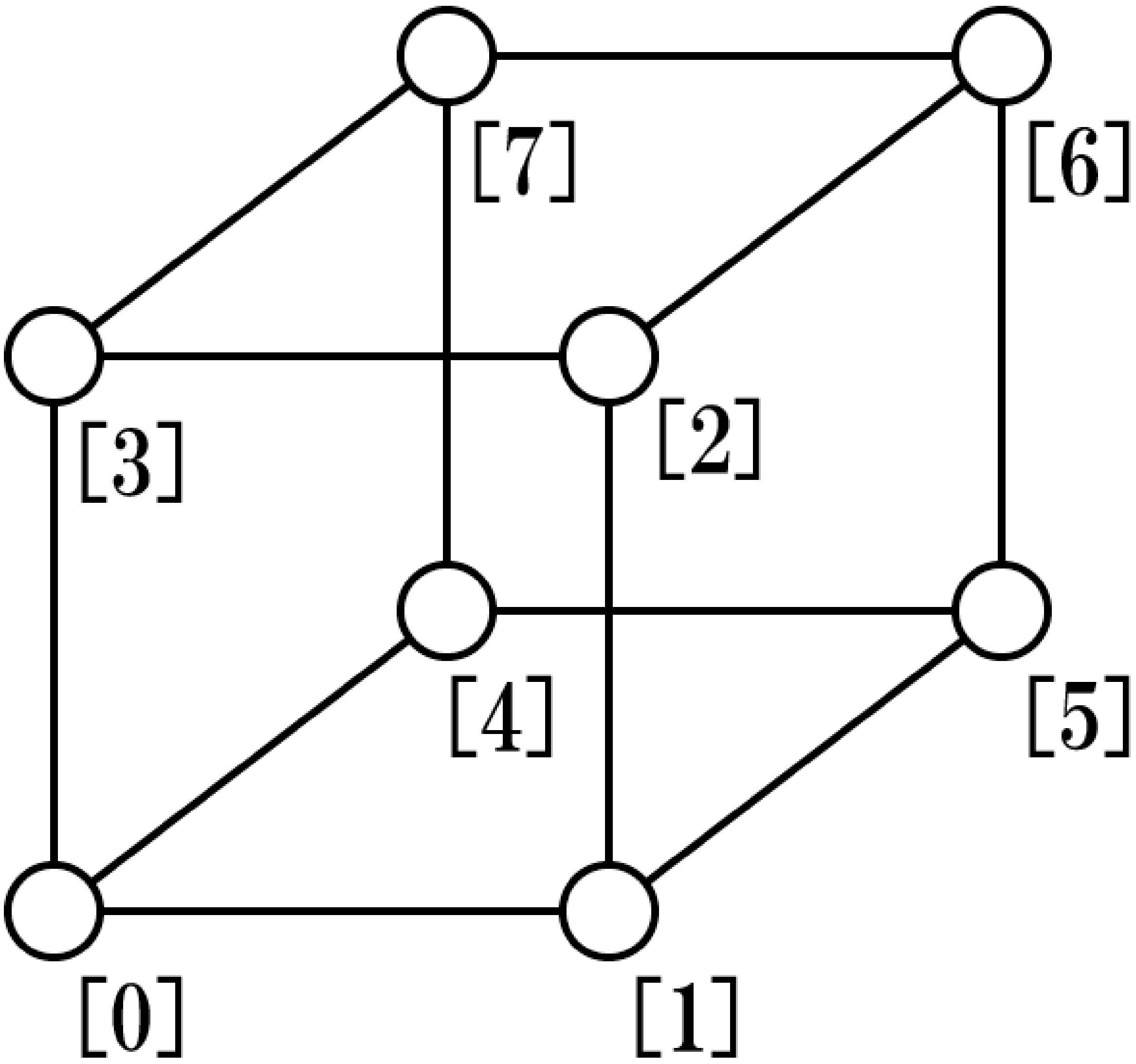


Figure 7.4: Order of grid corner coordinates

Listing 7.11: Sampling function part

```
// sampling function
float Sample(float x, float y, float z) {

  // Are the coordinates out of 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 3 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;
}
```

This function fetches the scalar value of the specified coordinates from the volume data. This time, instead of using a huge amount of 3D volume data, we will calculate the scalar value using a simple algorithm that uses a distance function.

About the distance function

The 3D shape drawn by the Marching Cubes method this time is defined using what is called a " **distance function**" .

The distance function here is, roughly speaking, a "function that satisfies the distance condition".

For example, the distance function of a sphere is:

Listing 7.12: Sphere Distance Function

Function

```
inline float sphere(float3 pos,
float radius)
{
    return length(pos) - radius;
}
```

Coordinates are entered in pos, but consider the case where the center coordinates of the sphere are the origin (0,0,0). radius is the radius.

The length is calculated by length (pos), but this is the distance between the origin and pos, and it is subtracted by the radius radius, so if the length is less than the radius, it is a natural but negative value.

In other words, if you pass the coordinates pos and a negative value is returned, you can judge that the coordinates are inside the sphere.

The advantage of the distance function is that it is easy to make the program small because the figure can be expressed with a simple calculation formula of several 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

Listing 7.13: A composite of the distance functions of three spheres

```
// Distance function of 3 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, 8 corners (vertices) of 1 square of the grid are used as pos. The distance from the center of the sphere is treated as it is as the density of the volume data.

As will be described later, the sign is inverted because it is polygonized when the threshold value is 0.5 or more. In addition, the coordinates are slightly shifted to obtain the distances to the three spheres.

Listing 7.14: smoothMax function

```
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 the three spheres like a metaball.

Listing 7.15: Threshold Check

```
// Check if the values at 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];

// Do not draw anything 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 the flagIndex as an index, the information for generating polygons is extracted from the cubeEdgeFlags array and stored in edgeFlags. If all corners of the grid are below or above the threshold, it is completely inside or outside and no polygons are generated.

Listing 7.16: Polygon Vertex Coordinate Calculation

```
float offset = 0.5;
float3 vertex;
for (i = 0; i < 12; i++) {
  if ((edgeFlags & (1 << i)) != 0) {
    // Get the threshold offset between the 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 (requires vertex coordinates before
scaling to resample)
    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. Looking at the bit of edgeFlags earlier, we are calculating the vertex coordinates of the polygon to be placed on the edge of the grid.

getOffset gives the ratio (offset) from the current corner to the next corner from the scalar values and thresholds of the two corners of the grid. By offsetting the coordinates of the current corner toward the next corner by offset, the polygon will eventually become smooth.

In getNormal, the normal is calculated by re-sampling and calculating the gradient.

Listing 7.17: Concatenate vertices to make a polygon

```
// Concatenate vertices to create polygons
int vindex = 0;
int findex = 0;
// Create up to 5 triangles
for (i = 0; i < 5; i++) {
  findx = 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); // Append vertices to strip
  }
   outStream.RestartStrip (); // Break once and start the next
primitive strip
}
```

This is the place where the polygon is made by connecting the vertex coordinate groups obtained earlier. triangleConnectionTable Contains the indexes of the vertices that connect to the array. Multiply the vertex coordinates by the Transform matrix to convert to world coordinates, and then use UnityObjectToClipPos () to convert to screen coordinates.

Also, UnityObjectToWorldNormal () converts the normals to the world coordinate system. These vertices and normals will be used for lighting in the next 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 on one strip.

7.4.4 Entity Fragment Shader

In order to reflect the lighting such as GI (Global Illumination) of Unity, the lighting processing part of Surface Shader after Generate code is ported.

Listing 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)
```

There are 4 outputs (SV_Target) to output to G-Buffer.

Listing 7.19: Initializing the 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 to the SurfaceOutputStandard structure that will be used later.

Listing 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);
```

GI related processing. I put the initial value in UnityGIInput and write the result of GI calculated by LightnintStandard_GI () to UnityGI.

Listing 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 calculation results to LightingStandard_Deferred () to calculate the degree of light reflection and write it to the Emission buffer. In the case of HDR, write after sandwiching the part compressed by exp.

7.4.5 Shadow Geometry Shader

It's almost the same as the actual geometry shader. I will explain only where there are differences.

Listing 7.22: Shadow Geometry Shader

```
int vindex = 0;
int findex = 0;
for (i = 0; i < 5; i++) {
  findx = 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();
}
```

Convert the vertex coordinates to the coordinates of the shadow projection destination with UnityClipSpaceShadowCasterPos () and UnityApplyLinearShadowBias ().

7.4.6 Shadow Fragment Shader

Listing 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, the shadow is drawn normally even with return 0 ;. Is Unity doing a good job inside?

7.5 Finish

When you run it, you should see a picture like this.

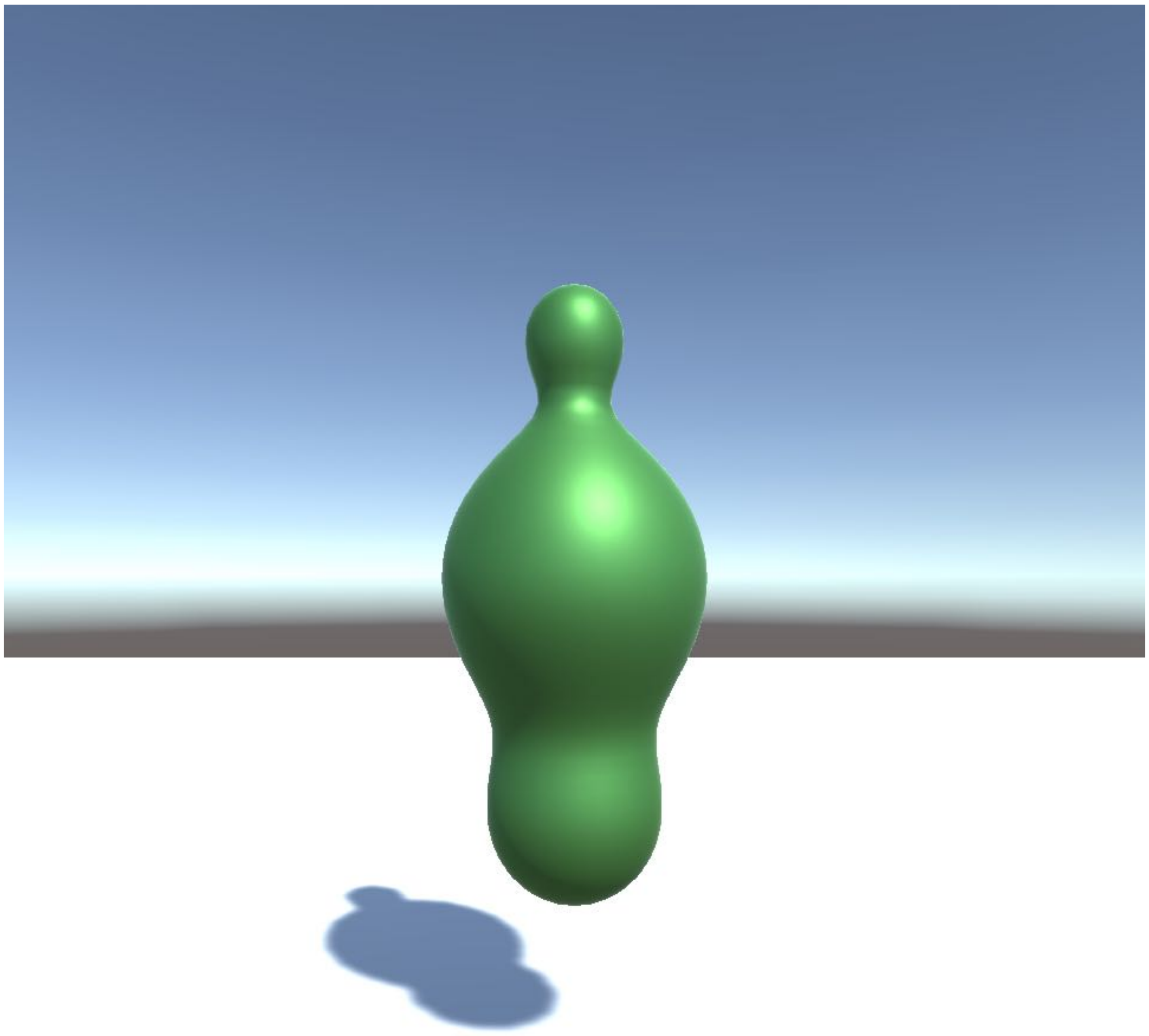


Figure 7.5: undulation

Also, various shapes can be created by combining distance functions.

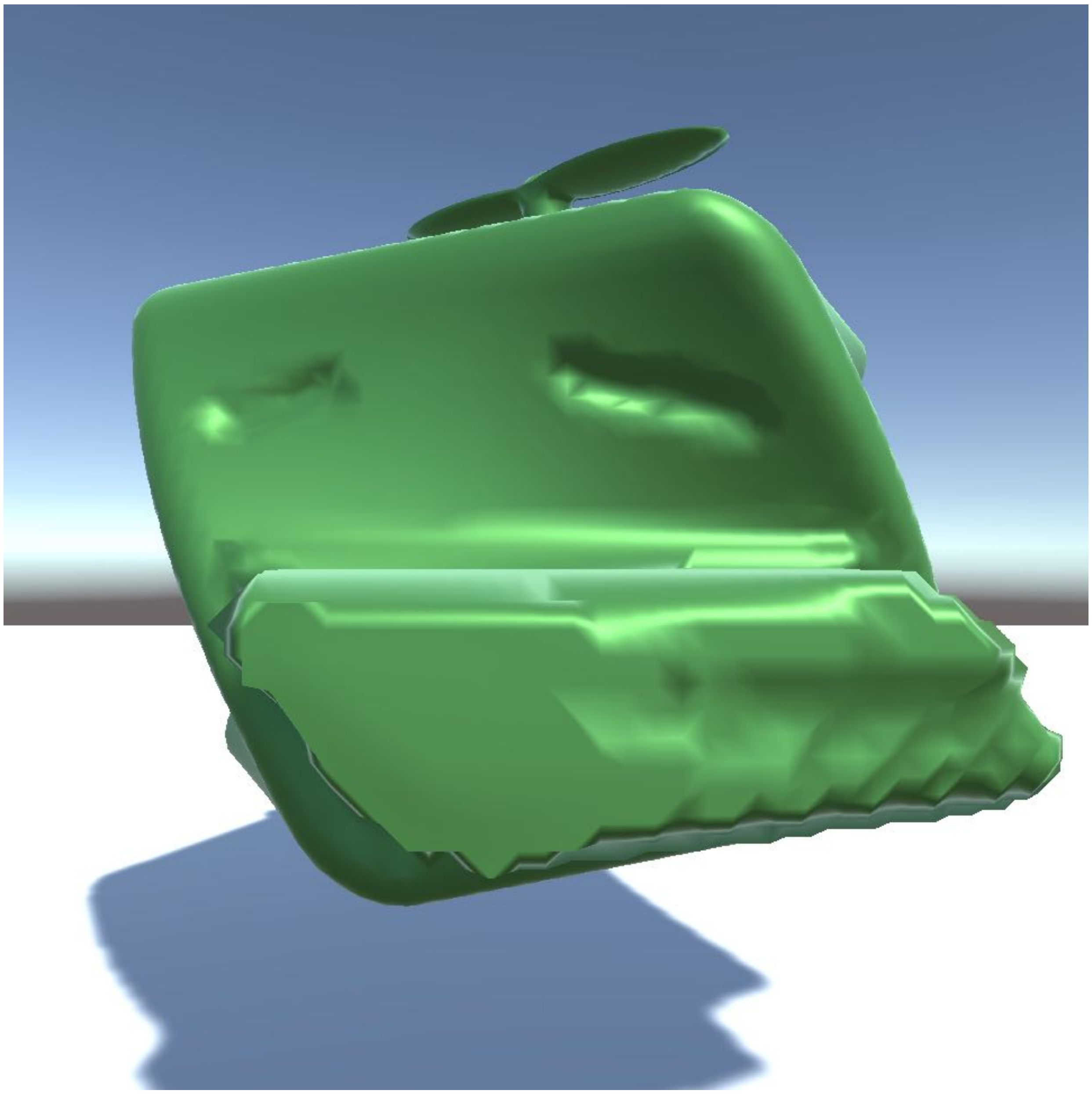


Figure 7.6: Kaiware daikon

7.6 Summary

This time I used the distance function for simplification, but I think that the Marching cubes method can also be used to use 3D textures with volume data written in them and to visualize various 3D data. ..
For game use, you may be able to create games like ASTORONEER * 2 , which allows you to dig and build terrain .

Everyone, please try to find various expressions with 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

Chapter 8 3D Spatial Sampling with 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), which samples multiple appropriate values from a certain probability distribution.

The simplest method for sampling from a certain probability distribution is the rejection method, but sampling in a three-dimensional space has a large rejected area and cannot withstand actual operation. Therefore, the content of this chapter is that by using MCMC, sampling can be performed efficiently even in high dimensions.

As for the information about MCMC, on the one hand, systematic information such as books is for statisticians, and there is no guide to implementation for programmers, although it is redundant, and on the other hand, the information on the net has more than 10 lines of sample code. The reality is that there is no content that allows you to quickly and comprehensively understand the theory and implementation, as it is only described and there is no care for the theoretical background. I tried to make the concrete explanations in the following sections as such as possible.

The explanation of the probability that is the background of MCMC is enough to write one book if it is strictly speaking. This time, with the motto of explaining the minimum theoretical background that can be implemented with peace of mind, we aimed for an intuitive expression with moderate strictness of definition. I think that if you have used mathematics in the first year of university or even a little at work, you can read the program without difficulty.

8.2 Sample repository

In this chapter, the Unity project of Unity Graphics Programming https://github.com/IndieVisualLab/Assets/ProceduralModeling in UnityGraphicsProgramming is prepared as a sample program.

8.3 Basic knowledge about probability

To understand the theory of MCMC, we first need to understand the basics of probability. However, there are few concepts to keep in mind in order to understand MCMC this time, only the following four. No likelihood or probability density function required!

- Random variable
- Probability distribution
- Stochastic process
- Stationary distribution

Let's look at them in order.

8.3.1 Random variables

When an event occurs at establishment P (X), this real number X is called a random variable. For example, when "the probability of getting a 5 on a dice is 1/6", "5" is a random variable and "1/6" is a probability. In general, the previous sentence can be rephrased as "the probability that an X on the dice will appear is P (X)".

By the way, if you write it a little like a definition, the random variable X is a mapping X that returns a real number X for the element ω (= one event that happened) selected from the sample space Ω (= all the events that can occur). You can write = X (ω).

8.3.2 Stochastic process

I added a slightly confusing definition in the latter half of the random variable because it makes it easier to understand the stochastic process on the assumption that the random variable X is represented by the notation X = X (ω). The stochastic process is the one that can be expressed as X = X (ω, t) by adding the time condition to X. In other words, the stochastic process can be thought of as a kind of random variable with a time condition.

8.3.3 Probability distribution

The probability distribution shows the correspondence between the random variable X and the probability P (X). It is often represented by 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 does not change even if it transitions. For a transition matrix π with a distribution P, P that satisfies πP = P is called a stationary distribution. This definition alone is confusing, but it is clear from the figure below.

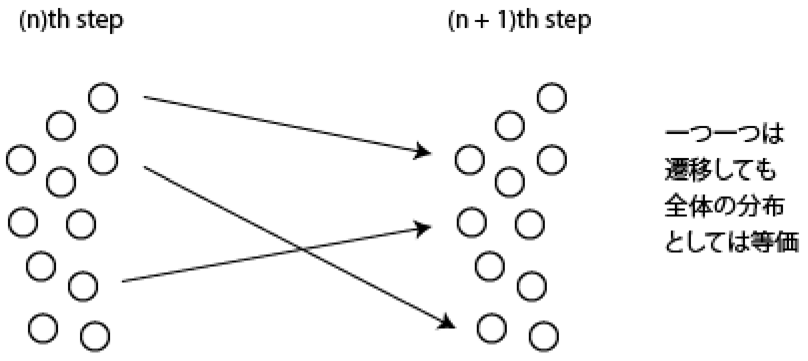


図 8.1: stationaryDistribution

8.4 MCMC concept

Now, in this section, we 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 specifically, the Monte Carlo method under the condition that the given distribution is a steady distribution. (Monte Carlo) and Markov chain (Markov chain) sampling method. Below, we will explain the Monte Carlo method, Markov chain, and stationary distribution in that order.

8.4.1 Monte Carlo method

The Monte Carlo method is a general term for numerical calculations and simulations that use pseudo-random numbers.
An example that is often used to introduce numerical calculations using the Monte Carlo method is the following calculation of pi.

```
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 Markov properties, in which states can be described discretely.
Markov property is the property that the probability distribution of the future state of a stochastic process depends only on the current state and not on the past state.

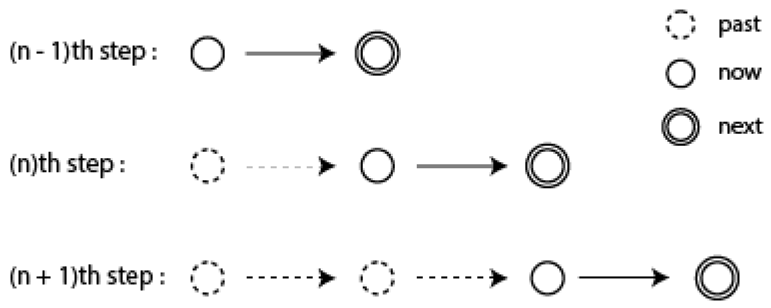


Figure 8.2: Markov Chain

As shown in the above figure, in the Markov chain, the future state depends only on the current state and does not directly affect the past state.

8.4.3 Stationary distribution

MCMC needs to use pseudo-random numbers to converge from an arbitrary distribution to a given stationary distribution. This is because if you do not converge to a given distribution, you will sample from a different distribution each time, and if you do not have a stationary distribution, you will not be able to sample well in a chain. In order for an arbitrary distribution to converge to a given distribution, the following two conditions must be met.

- Irreducibility: A 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 point

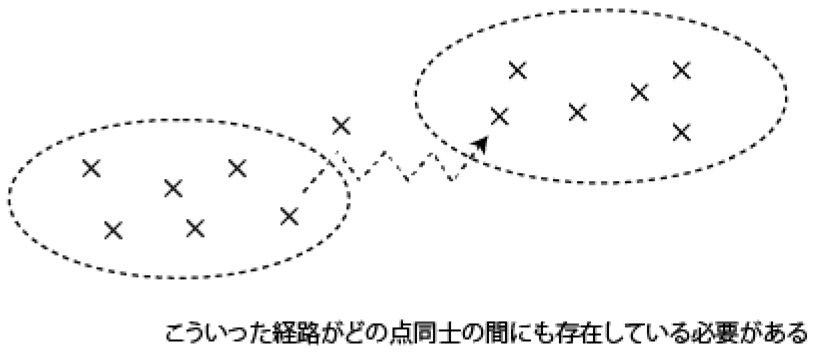


図 8.3: Irreducibility

- Aperiodicity: The condition that any n can be returned to the original place in n times. For example, in the distribution arranged on the circumference, there must be no condition that the transition can be made only by skipping one.

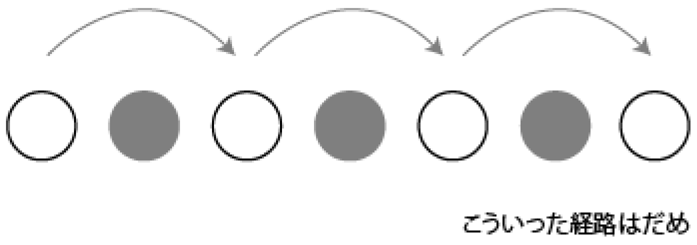


Figure 8.4: Aperiodicity

Any distribution that meets these two conditions can converge to a given stationary distribution. This is called the ergodic nature of the Markov process.

8.4.4 Metropolis method

Now, it is difficult to check whether the given distribution satisfies the ergonomics mentioned earlier, so in many cases, we will strengthen the conditions and investigate within the range that satisfies the condition of "detailed balance". One of the Markov chain methods that achieves detailed balance is called the metropolis method.

The metropolis method samples by taking the following two steps.

1. Select the 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 destination) Transitions to the transition candidate destination if P (x') / P (x)> r is satisfied.

The merit of the metropolis method is that even after the transition to the maximum value of the probability distribution is completed, if the value of r is small, the probability value transitions to the smaller one, so sampling proportional to the probability value can be performed around the maximum value.

By the way, the Metropolis method is a kind of Metropolis-Hastings method (MH method). The Metropolis method uses a symmetrical distribution for the proposed distribution, but the MH method does not.

8.5 3D sampling

Let's take a look at the actual code excerpt and see how to implement MCMC.

First, prepare a three-dimensional probability distribution. This is called the target distribution. This is the "target" distribution because it is the distribution you actually want to 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 adopted 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. Since MCMC starts processing from a completely different place when one Markov chain ends, it can be conceptually considered as parallel processing. This time, I use the Reset function to run another process after a series of processes. By doing this, you will be able to sample well even if there are many maxima of the probability distribution.

Since the first part of the transition is likely to be a point away from the target distribution, this section is burn-in without sampling. When the target distribution is sufficiently approached, sampling and transition set are performed a certain number of times, and when finished, another series of processing is started.

Finally, it is the process of determining the transition.
Since it is three-dimensional, the proposed distribution uses a 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;
}
```

In the Metropolis method, the distribution must be symmetrical, so the mean value is not set to anything other than 0, but if the variance is set to something other than 1, it is derived as follows using the Cholesky decomposition. 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);
}
```

To determine the transition destination, take the ratio of the probabilities of the proposed distribution (one point above) next and the immediately preceding point_curr on the target distribution, and if it is larger than a uniform random number, it will transition, otherwise it will not transition. I will.
Since the process of finding the probability value corresponding to the coordinates of the transition destination is heavy (the amount of processing of O (n ^ 3)), the probability value is approximated. Since we are using a distribution in which the target distribution changes continuously this time, the established 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 3D rejection method (a simple Monte Carlo method as shown in the circle example), so it is a good idea to compare them. With the rejection method, sampling cannot be done well if the rejection standard value is set stronger, whereas with MCMC, similar sampling results can be presented more smoothly. Also, in MCMC, if the width of the random walk for each step is reduced, sampling is performed from a close space in a series of chains, so it is possible to easily reproduce a cluster of plants and flowers.

8.7 References

- Takuya Kubo (2012) Introduction to Statistical Modeling for Data Analysis: Generalized Linear Model, Hierarchical Bayes Model, MCMC (Science of Probability and Information) Iwanami Shoten
- Olle Haggstrom, Kentaro Nomaguchi (2017) Introduction to Easy MCMC: Limited Markov Chain and Algorithm Kyoritsu Shuppan

Chapter 9 MultiPlane Perspective Projection

In this chapter, we will introduce a video projection method that allows you to experience the experience of being in the CG world by projecting images on multiple surfaces such as the walls and floor of a rectangular parallelepiped room. In addition, as the background, we will explain camera processing in CG and its application examples. The sample project can be found in Assets / Room Projection in Unity Project [* 1] of Unity Graphics Programming, so please have a look. In addition, this content has been significantly revised and revised based on the content contributed to the "Mathematics Seminar December 2016" [* 2] .

[* 1] Sample project https://github.com/IndieVisualLab/UnityGraphicsProgramming

[*2] https://www.nippyo.co.jp/shop/magazine/7292.html

9.1 How the camera works in CG

Camera processing in general CG is processing that projects a 3D model of the visible range onto a 2D image using perspective projection conversion. The perspective projection conversion is a local coordinate system with the center of each model as the origin, a world coordinate system with the origin at 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 4-dimensional coordinate system in which w also has meaning, and the 3-dimensional version is called **NDC (Normalized Device Coordinates**), which is a screen coordinate system that represents the 2-dimensional position of the output screen. The coordinates of the vertices are projected in order.

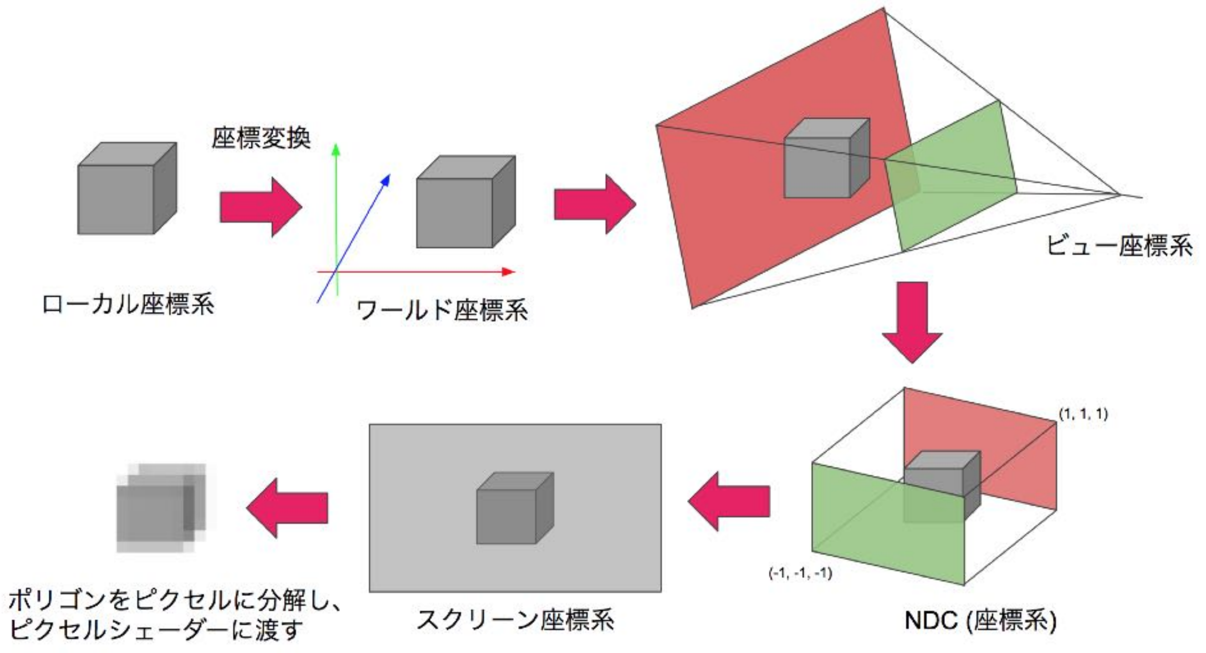


Figure 9.1: Flow of coordinate transformation

In addition, since each of these transformations can be represented by one matrix, it is common practice to multiply the matrices in advance so that some coordinate transformations can be done by multiplying the matrix and the vector once.

9.2 Perspective consistency across multiple cameras

In a camera in CG, a quadrangular pyramid with the apex at the camera position and the bottom surface at the camera orientation is called the viewing frustum, and can be illustrated as a 3D volume that represents the projection of the camera.

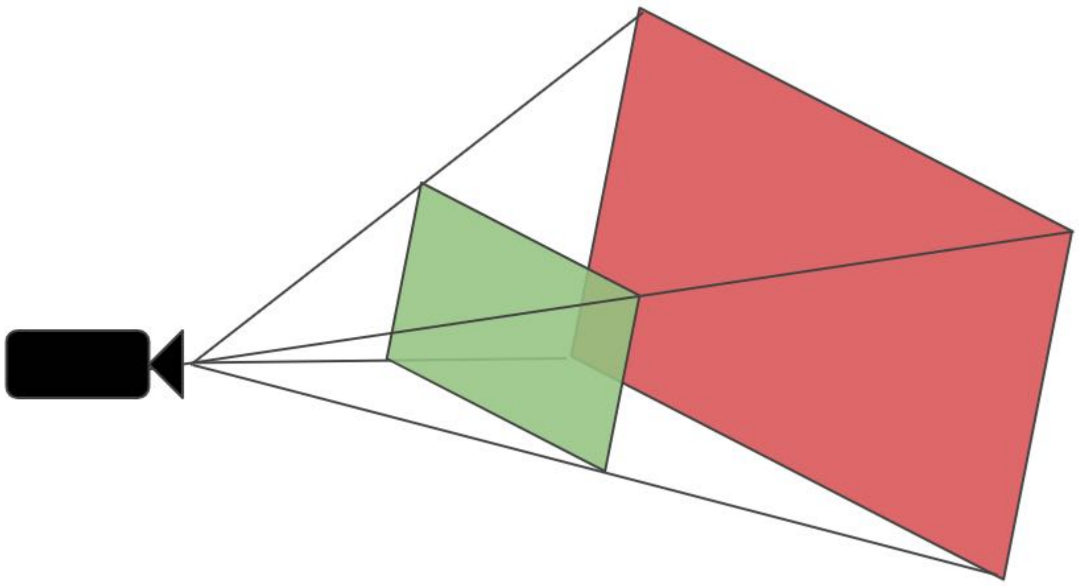


Figure 9.2: Frustum

If the viewing frustums of the two cameras share the apex and the sides are in contact, they will be connected visually even if the projection surfaces are facing different directions, and the perspectives when viewed from the apex will be the same. I will.

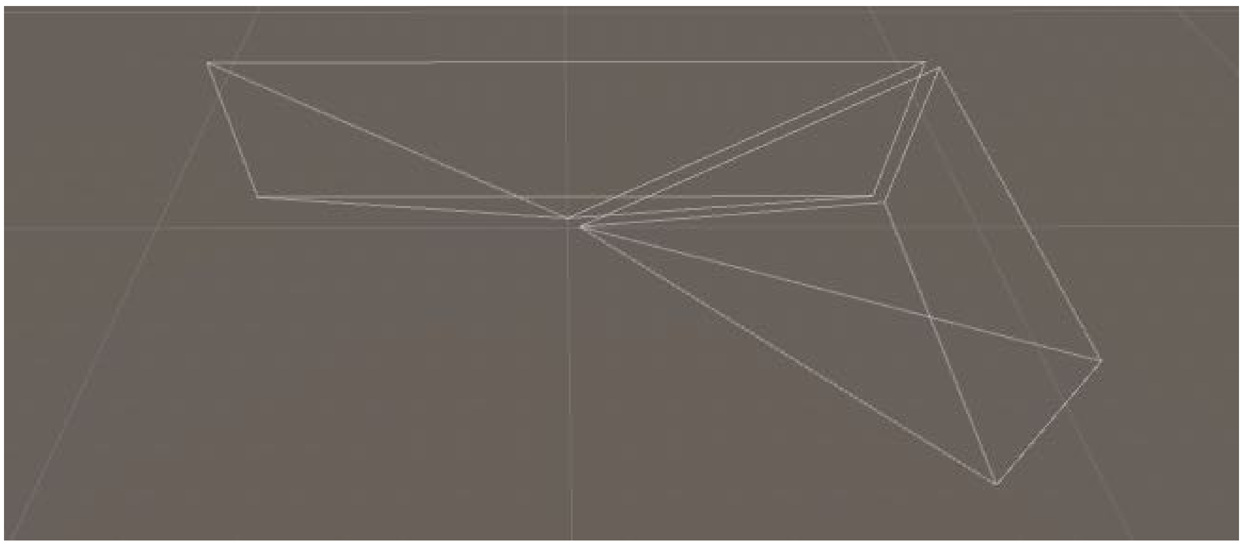


Figure 9.3: Touching frustum (placed slightly apart for clarity)

This can be understood by considering the view frustum as a set of innumerable lines of sight and thinking that the lines of sight are continuous (= images that are consistent in perspective can be projected). This idea was extended to five cameras, and the angle of view was adjusted so that the five view frustums shared the apex and were in contact with the adjacent view frustums, thereby corresponding to each surface of the room. You can generate a video. Theoretically, 6 faces including the ceiling are possible, but this time we consider it as a projector installation space and assume 5 faces excluding the ceiling.

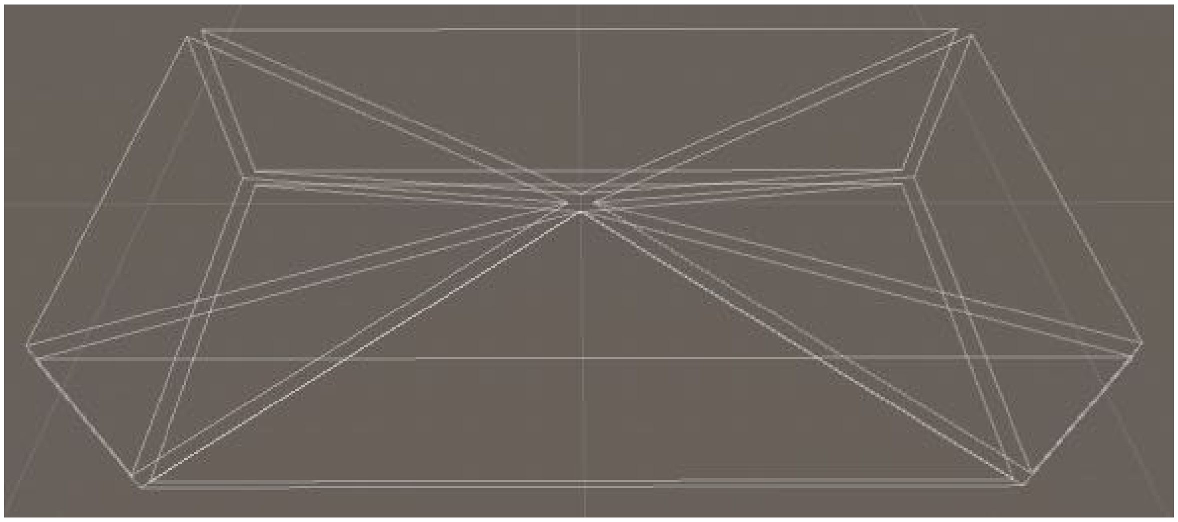


Figure 9.4: Five viewing frustums corresponding to the room

By viewing from this apex, that is, the location corresponding to all camera positions, you can view images that are consistent in perspective regardless of the direction of the room.

9.3 Derivation of projection matrix

The projection matrix (hereinafter referred to as Proj) is a matrix that transforms from the view coordinate system to the clip coordinate system.

- C : Position vector in the clip coordinate system
- V : is the position vector in the view coordinate system

It is expressed as follows by the formula.

```
C = Proj * V
```

Furthermore C each element of C_ {w} will position coordinates in NDC by dividing.

```
NDC    =    (\frac{C_{x}}{C_{w}},\frac{C_{y}}{C_{w}},\frac{C_{z}}
{C_{w}})
```

In addition, C_ {w} = -V_ {z} (make Proj so that). Since the front direction of the view coordinate system is the Z minus direction, it is minus. In NDC, the display range is -1 \ leq x, y, z \ leq 1, and this conversion scales V_ {x, y} according to V_ { z} to obtain a perspective expression.

Now let's think about how to make Proj . Let N be the coordinate of the upper right point of nearClipPlane and F be the coordinate of the upper right point of farClipPlane in the view coordinate system .

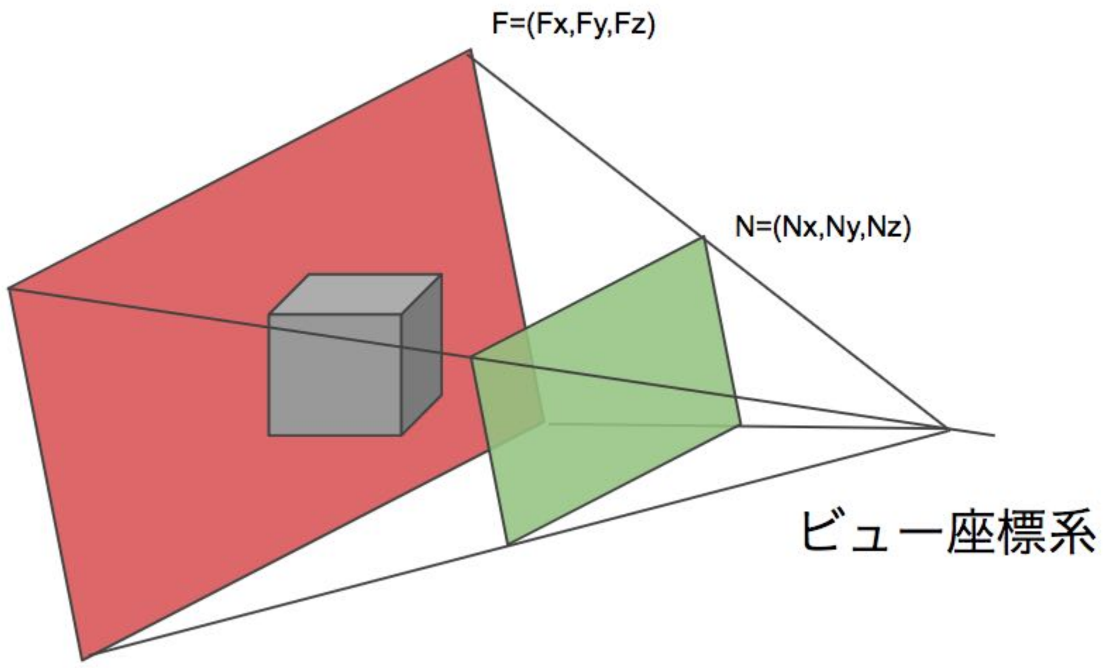


Figure 9.5: N, F

First of all, if you pay attention to x ,

- Projection range of -1 \leq x \leq 1 to become a
- Later divided by C_ {w} (= -V_ {z})

Considering

```
Proj [0,0] = \ frac {N_ {z}} {N_ {x}}
```

If so, it looks good. Since the ratio of x and z does not change, any x, z such as Proj [0] [0] = \frac {F_ {z}} {F_ {x}} can be used at the right end of the view frustum.

Similarly

```
Proj [1,1] = \ frac {N_ {z}} {N_ {y}}
```

Can also be obtained.

A little ingenuity is required for z . The calculation related to z in Proj * V is as follows.

```
C_ {z} = Proj [2,2] * V_ {z} + Proj [2,3] * V_ {w} (however, V_
{w} = 1)
```

```
NDC_ {z} = \ frac {C_ {z}} {C_ {w}} (however, C_ {w} = -V_ {z})
```

Here, I want to convert N_ {z} → -1, F_ {z} → 1 , so I put a = Proj [2,2], b = Proj [2,3].

```
-1 = \frac{1}{N_{z}} (aN_{z} +b),
1 = \frac{1}{F_{z}} (aF_{z} +b)
```

A solution can be obtained from this system of equations.

```
Proj [2,2] = a = \ frac {F_ {z} + N_ {z}} {F_ {z} -N_ {z}},
Proj [2,3] = b = \ frac {-2F_ {z} N_ {z}} {F_ {z} -N_ {z}}
```

Also, I want C_ {w} = -V_ {w}

```
Proj[3,2] = -1
```

will do.

Therefore, the required Proj has the following form.

```
Proj = \left(
\begin{array}{cccc}
    \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_{z}N_{z}}{F_{z}-N_{z}} \\
    0   &   0 & -1 & 0
\end{array}
\right)
```

9.3.1 Camera.projectionMatrix trap

Some of the people who have dealt with projection matrices in shaders may feel uncomfortable with the contents so far. Actually, the handling of the projection matrix of Unity is complicated, and the contents so far are the explanation about Camera.projectionMatrix. This value is OpenGL compliant regardless of platform [* 3] . This is why -1 \ leq NDC_ {z} \ leq 1 and C_ {w} = -V_ {w} .

[*3] https://docs.unity3d.com/ScriptReference/GL.GetGPUProjectionMatrix.html

However, Camera.projectionMatrix is not always used for perspective projection conversion as it is converted to a platform-dependent form when it is passed to the shader in Unity. In particular, the range and orientation of NDC_ {z} (that is, the handling of the Z buffer) are diverse and easy to get caught [* 4] .

[*4] https://docs.unity3d.com/Manual/SL-PlatformDifferences.html

9.4 Operation of the viewing frustum

9.4.1 Projection surface size adjustment

The shape of the bottom of the view frustum, or projection plane, depends on the camera's **fov (fieldOfView)** and **aspect (aspect ratio)** . In Unity's camera, the angle of view is published in the Inspector, but the aspect ratio is not published, so you need to edit it from the code. **The** code to calculate the angle of view and aspect ratio from **faceSize (the size of the surface of the room)** and **distance (distance from the viewpoint to the surface)** is as follows.

Listing 9.1: Finding the angle of view and aspect ratio

```
camera.aspect = faceSize.x / faceSize.y;
camera.fieldOfView  =  2f  *  Mathf.Atan2(faceSize.y  *  0.5f,
distance)
                    * Mathf.Rad2Deg;
```

Note that Mathf.Atan2 () is used to find half the angle of fov with radian, doubled, and corrected to degree for substitution in Camera.fieldOfView.

9.4.2 Movement of projection plane (lens shift)

Consider the case where the viewpoint is not in the center of the room. If the projection plane can be translated vertically and horizontally with respect to the viewpoint, the same effect as moving the viewpoint with respect to the projection plane can be obtained. In the real world, this corresponds to a function called **lens shift** that adjusts the projection position of the image with a projector or the like .

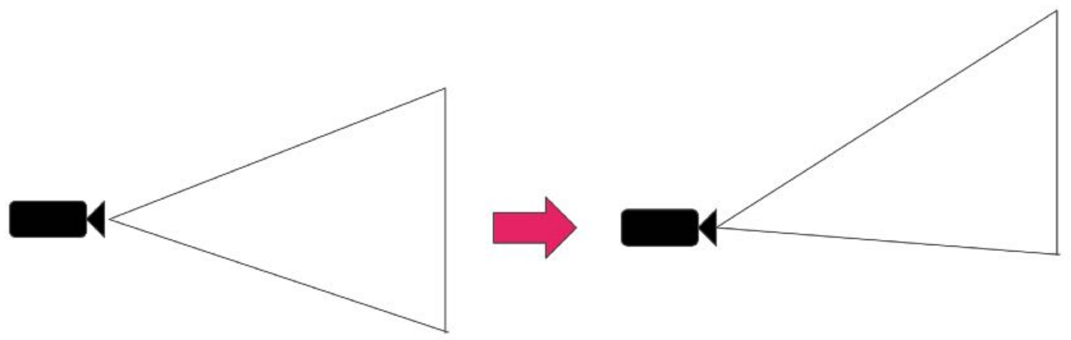


Figure 9.6: Lens shift

Looking back at the mechanism by which the camera performs perspective projection, what part of the lens shift is the process? When projecting to NDC with a projection matrix, it seems good to shift x and y. Let's look at the Projection matrix again.

```
Proj = \left(
\begin{array}{cccc}
    \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_{z}N_{z}}{F_{z}-N_{z}} \\
    0   &   0 & -1 & 0
\end{array}
\right)
```

As long as C_ {x} and C_ {y} are out of alignment , I want to put something in the translation components of the matrix, Proj [0,3] and Pro [1,3] , but later C_ {w} Considering that it is divided by, the correct answer is to put it in Proj [0,2], Pro [1,2] .

```
Proj = \left(
\begin{array}{cccc}
    \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_{z}N_{z}}{F_{z}-N_{z}} \\
    0   &   0 & -1 & 0
\end{array}
\right)
```

Since the unit of LensShift is NDC, the size of the projection plane is normalized to -1 to 1. The code looks like this:

Listing 9.2: Reflect lens shift in projection matrix

```
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;
```

Note that once set to Camera.projectionMatrix, subsequent changes to Camera.fieldOfView will not be reflected unless Camera.ResetProjectionMatrix () is called. *Five

[*5] https://docs.unity3d.com/ScriptReference/Camera-projectionMatrix.html

9.5 room projection

It is assumed that the viewer's viewpoint position can be tracked in a rectangular parallelepiped room. Since the size of the projection surface of the viewing frustum can be translated by the method in the previous section, the viewing frustum is moved so that the viewpoint position is the apex of the viewing frustum and the wall surface or floor surface is the projection surface. You can ask for it. By making each camera such a viewing frustum, it is possible to create an image for each projection plane. By projecting this image into an actual room, the viewer will be able to see the CG world with perspective.

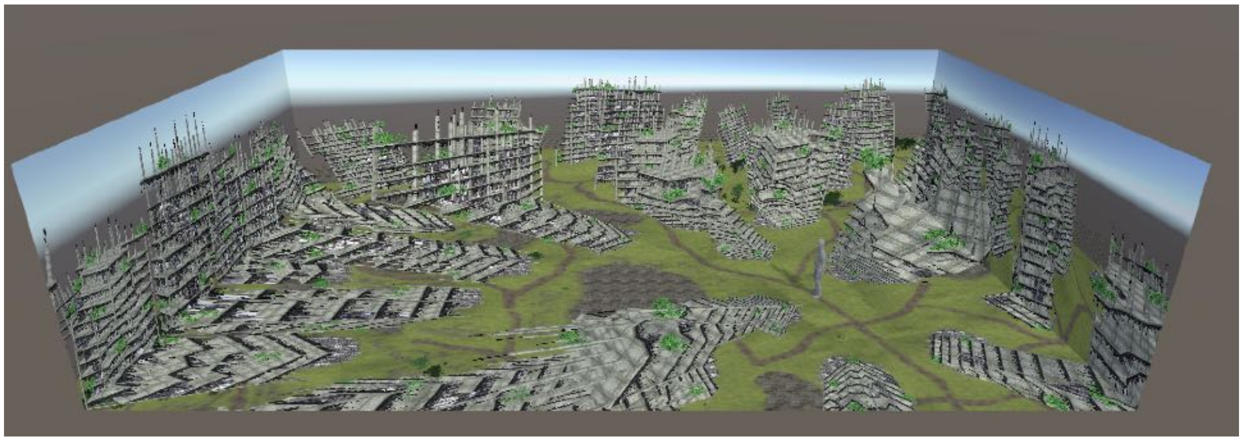


Figure 9.7: Room simulation (overhead view)

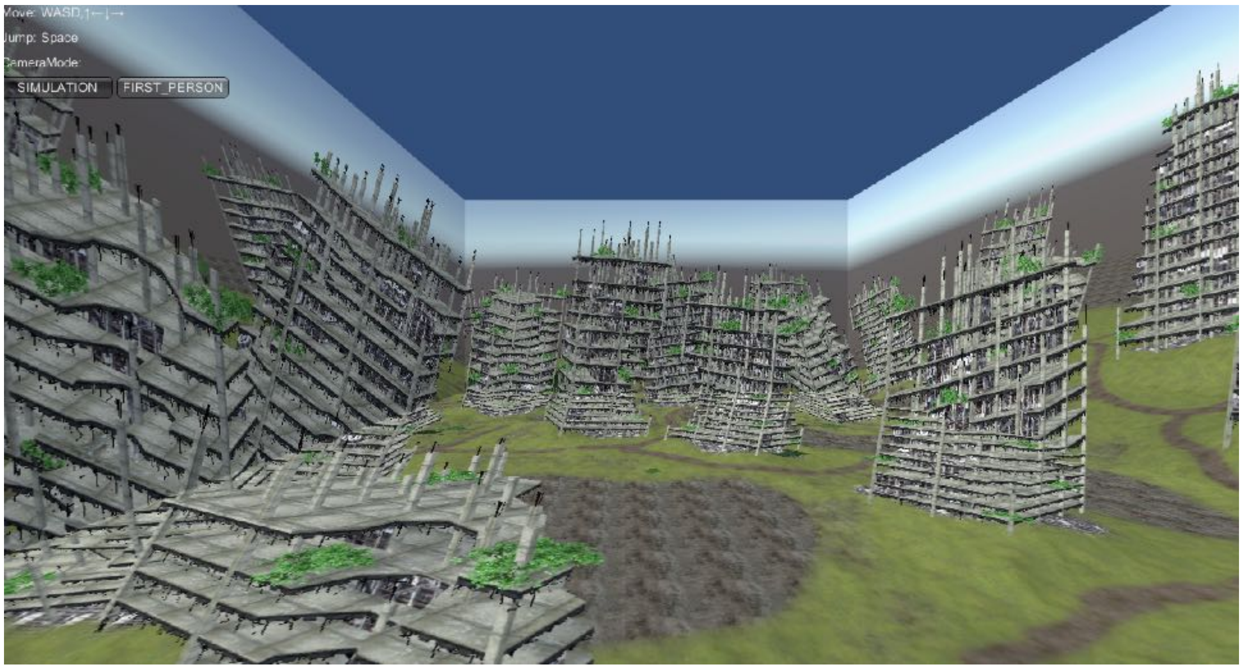


Figure 9.8: Room simulation (first person view)

9.6 Summary

In this chapter, we introduced a projection method that matches perspectives on multiple projection planes by applying a projection matrix. I think that it can be said that it is a VR with a non-approach similar to the recent HMD type in that it makes a wide range of the field of view a dynamically responsive image instead of placing the display in front of you. In addition, this method does not deceive binocular parallax or eye focus, so it may not be possible to see stereoscopically as it is, and it may look like a "moving picture projected on the wall". It seems that we need to devise a little more to increase the immersive feeling.

- Enlarge the room so that the binocular parallax is small, and increase the distance from the viewer to the projection surface.
- Prevents the plane of the projection surface from being conscious of reflected light as much as possible
    - Make a dark image
    - Make walls and floors as non-specular as possible

A mechanism called "CAVE" [* 6] that combines the same method with stereoscopic vision is known.

[*6] https://en.wikipedia.org/wiki/Cave_automatic_virtual_environment

Chapter 10 Introduction to Projection Spray

10.1 Introduction

Hello! It's Sugihiro Nori! Unfortunately not.

One day when the deadline was approaching, when I asked "Are you writing an article too much?", He just said "Ah!", And apparently he was completely forgotten. I've been busy lately, but I'd like to take this opportunity to introduce his achievements, so I'll send you a ghostwriter here.

10.1.1 ProjectionSpray

Sugicho is actively publishing his work on Github, and I personally found it interesting.

https://github.com/sugi-cho/ProjectionSpray

You can color the 3D model by spraying it.

Demo image

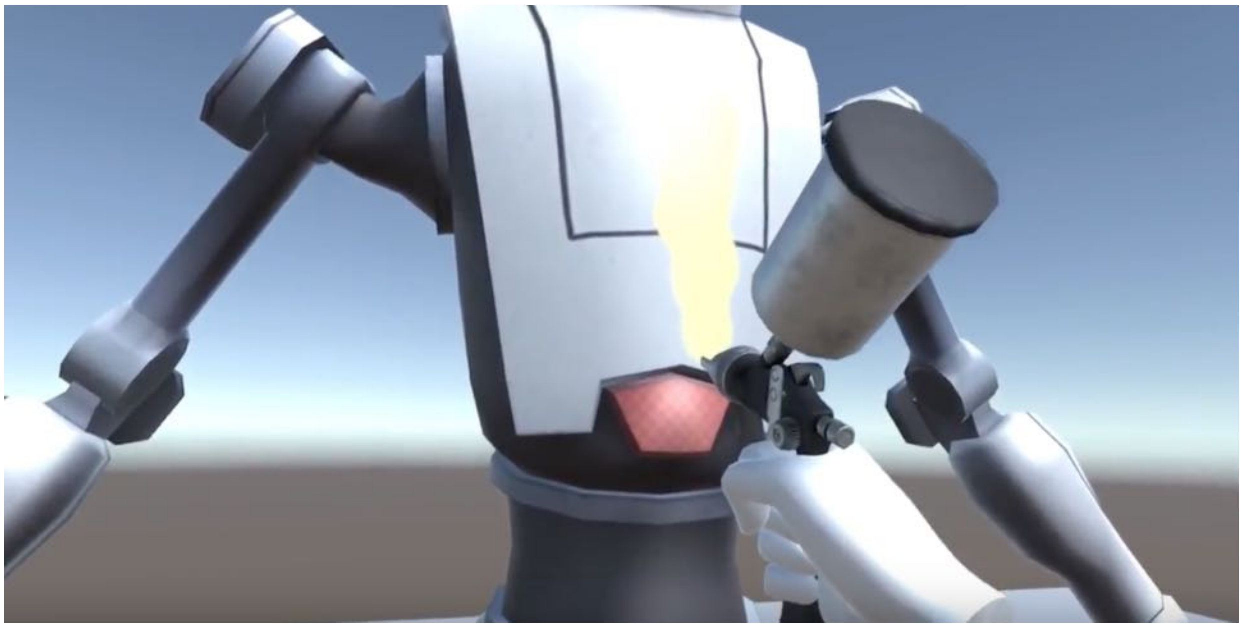


Figure 10.1: Demo image 1

A spray is ejected from the spray device to color the surface of the body.

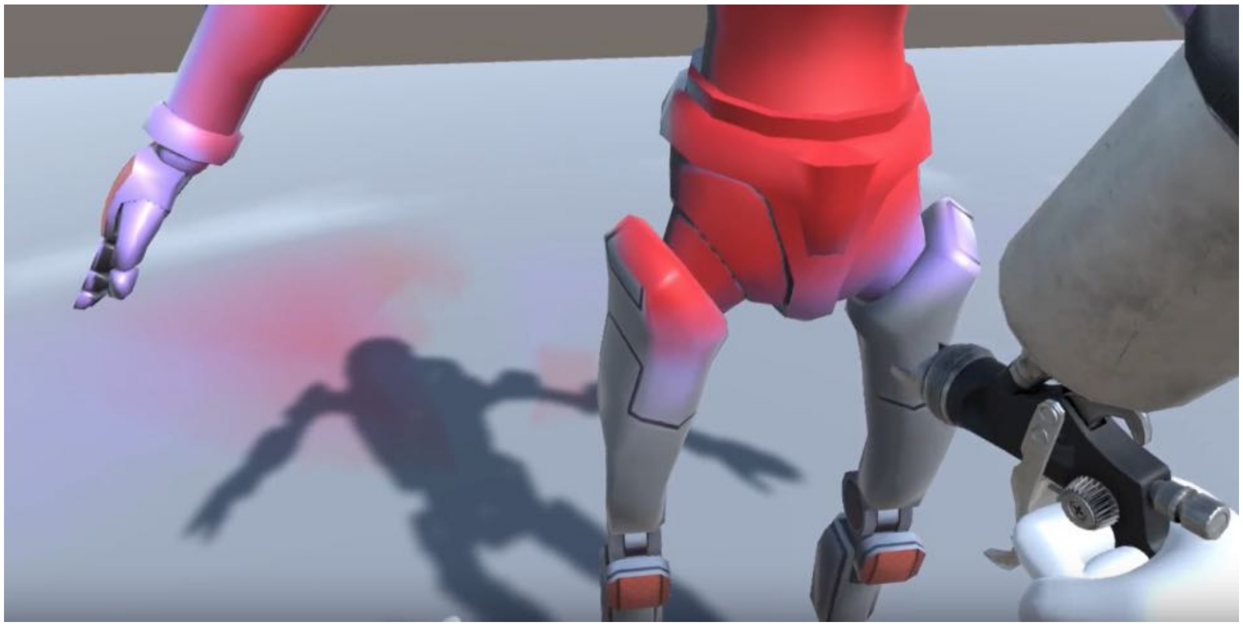


Figure 10.2: Demo image 2

I feel a mysterious fetishism.

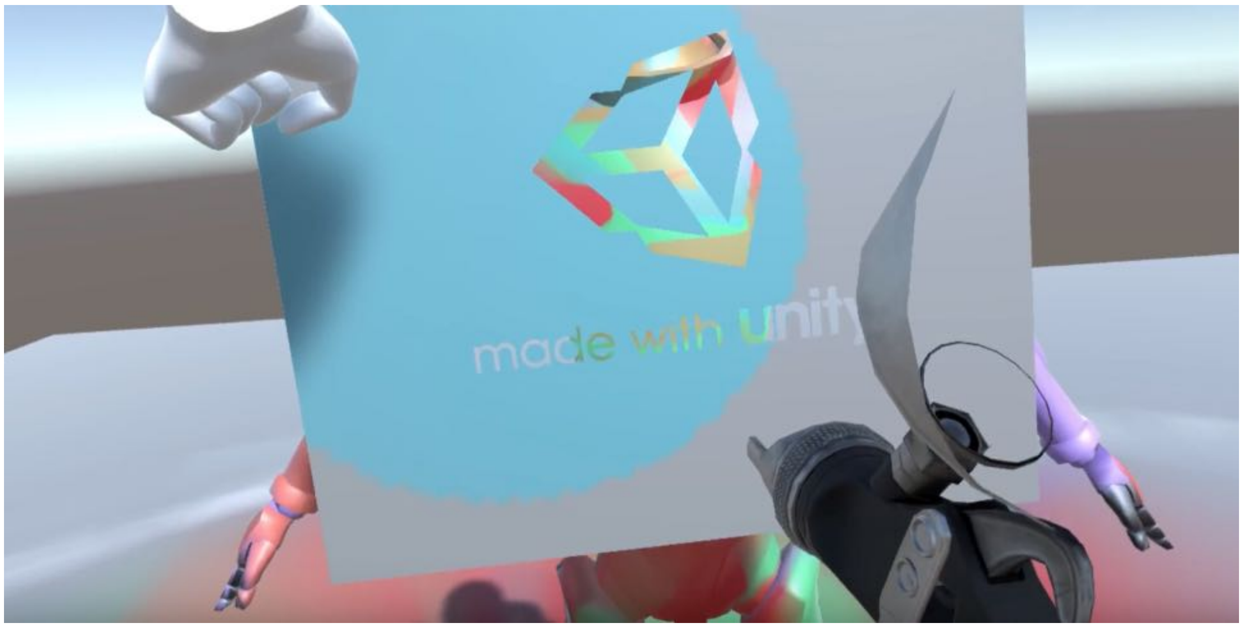


Figure 10.3: Demo image 3

Something like a stencil!

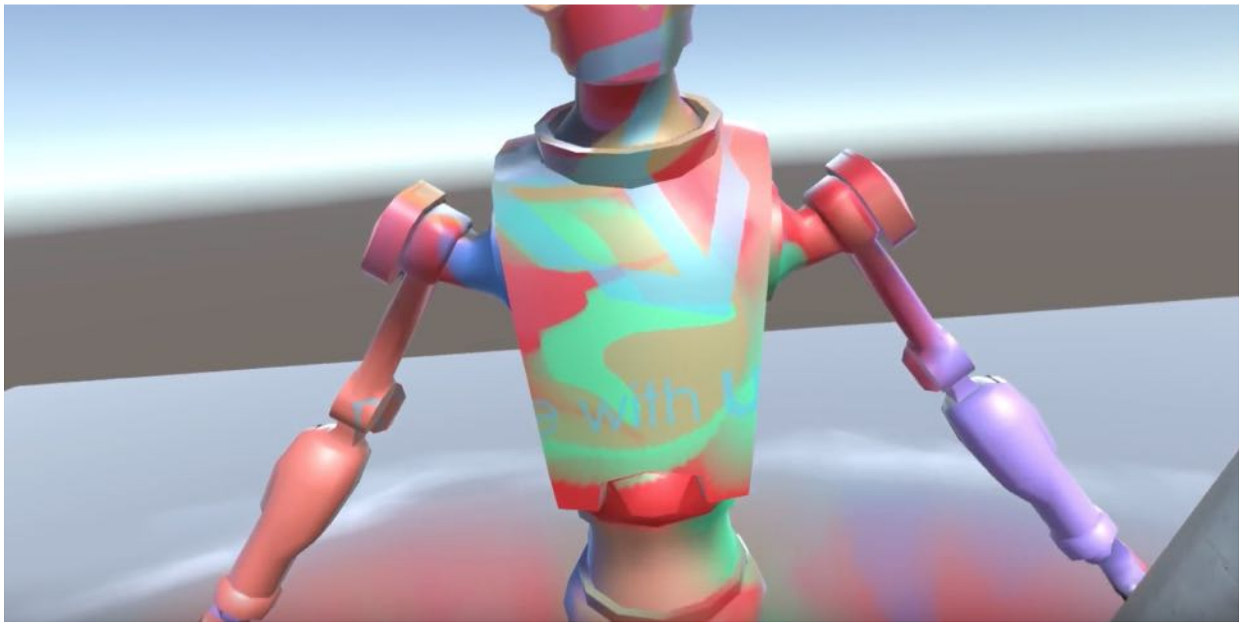


Figure 10.4: Demo image 4

Unity!

10.2 Summary

If there is next time, I would like to ask for a detailed explanation.

I also recommend the repository of my colleague Nakata-san, who said that Sugicho feels something similar to himself at the time of the interview, because he has a lot of useful and excellent code for Unity.

https://github.com/nobnak

excuse me.

(｡•ˇ_ˇ•｡)

About the author

Chapter 1 Procedural Modeling Beginning with Unity --Masatsu Nakamura / @mattatz

I try to hit all the balls that come in, such as installations, signage, the 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 / engineer, and it became very difficult. I manage to do it while studying with the help of the people around me.

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

Chapter 3 GPU Implementation of Group Simulation-Hiroaki Oishi / @irishoak

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

- https://twitter.com/_irishoak
- https://github.com/hiroakioishi
- http://irishoak.tumblr.com/

- https://a9ueduct.github.io/

Chapter 4 Fluid Simulation by Grid Method-Yoshiaki Sakoda / @sakope

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

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

Chapter 5 Fluid Simulation by SPH Method --Koudai Takao / @ kodai100

Interactive artist / engineer and student. While studying snow physics simulation at university, he also works in engineering. Recently I'm 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

Interaction engineer, programmer of small fish gale, loose fluffy force, anything shop that makes anything. My favorite school classroom is the drawing room or the library.

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

Chapter 7 Introduction to the Marching Cubes Method Beginning with Atmosphere-@ kaiware007

An interactive artist / engineer who works in an atmosphere. I like interactive content more than three meals. I like potatoes and don't eat radish sprouts. I often post Gene videos on Twitter. I do VJ once in a while.

- 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. Please contact twitter for production requests.

- https://github.com/komietty
- https://twitter.com/9_chinashi

Chapter 9 MultiPlanePerspectiveProjection --Hidekazu Fukunaga / @fuqunaga

Former game developer, current interactive artist / engineer. When I tried to eat breakfast to be careful about my health, I lost about 2 kg for some reason.

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

Chapter 10 Introducing Projection Spray-Hironori Sugi / @sugi_cho

A person who makes interactive art in Unity. Freelance. hi@sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc