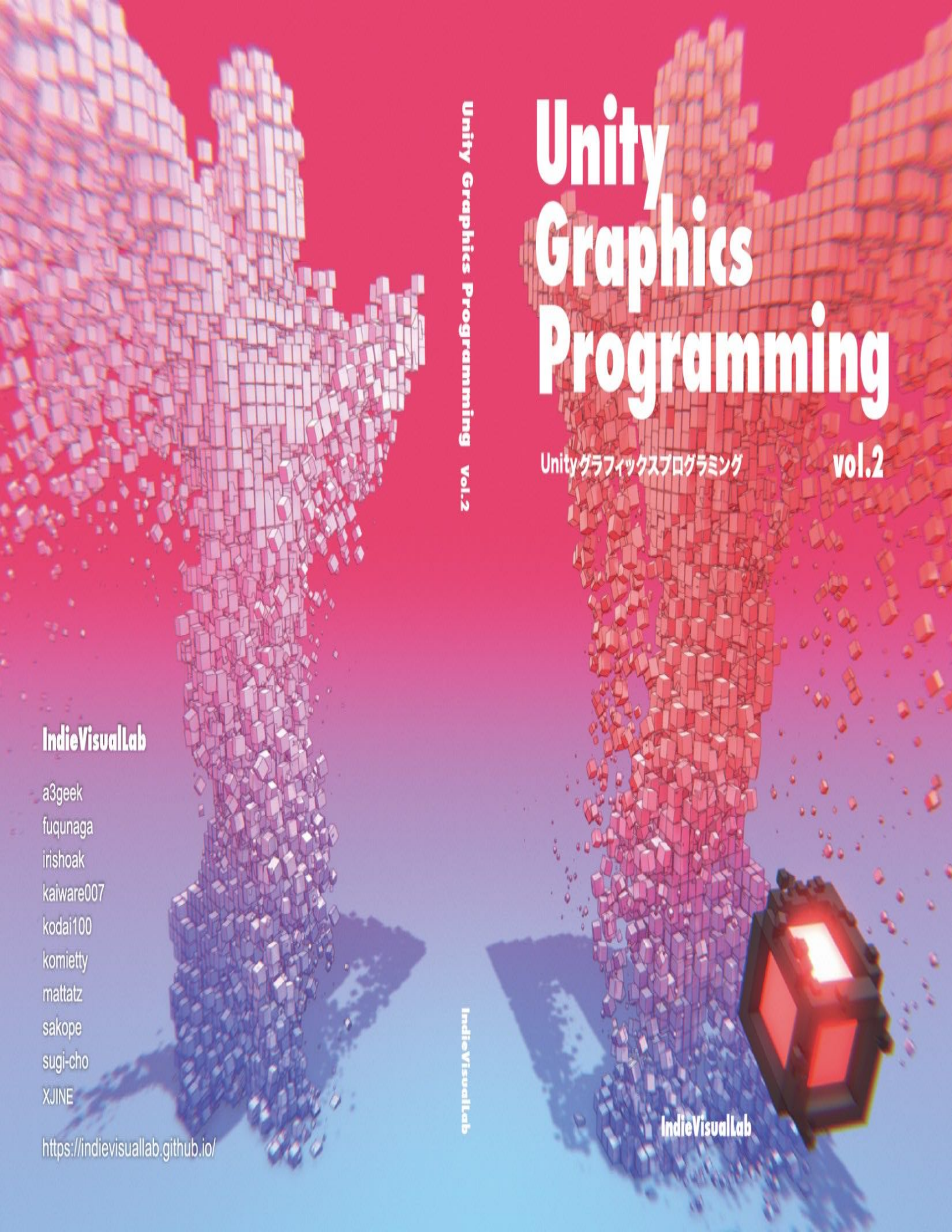
Unity Graphics Programming
Unityグラフィックスプログラミング
vol.2
IndieVisualLab
a3geek
fuqunaga
irishoak
kaiware007
kodai100
komietty
mattatz
sakope
sugi-cho
XJINE
https://indievisuallab.github.io/
Unity Graphics Programming vol.2
IndieVisualLab
IndieVisualLab

Preface

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

The source code explained in each chapter is published in the github repository (https://github.com/IndieVisualLab/UnityGraphicsProgramming2), so you can read this manual while executing it at hand.

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

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

Recommended execution environment

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

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

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

Requests and impressions about books

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

Figure 1: Web form QR code

第1章　Real-Time GPU-Based Voxelizer

1.1　Introduction

In this chapter, we will develop GPU Voxelizer, a program that uses GPU to make voxels of meshes in real time.

The sample in this chapter is "RealTime GPUBasedVoxelizer" from https://github.com/IndieVisualLab/UnityGraphicsProgramming2
.

First, after confirming the voxelization procedure and the obtained results based on the implementation on the CPU, we will explain the implementation method on the GPU and introduce an example of effects that apply high-speed voxelization.

1.1.1 What is a　voxel?

A voxel represents a basic unit in a three-dimensional reciprocal lattice space. It can be imagined as an increase in the dimension of the pixel (Pixel) used as the basic unit of the two-dimensional normal lattice space, and it is named Voxel in the sense of Pixel with Volume. Voxels can express volume, and each voxel may have a data format that stores values such as concentration, which may be used for visualization and analysis of medical and scientific data.

Also, in the game, Minecraft * 1 is listed as using voxels.

It takes time to create a detailed model and stage, but if it is a voxel model, it can be created with relatively little effort, and even if it is free, there are excellent editors such as MagicaVoxel * 2, and the model looks like a 3D pixel art. Can be created.

[*1] https://minecraft.net

[*2] http://ephtracy.github.io/

1.2 Voxelization algorithm

I will explain the voxelization algorithm based on the implementation on the CPU. The CPU implementation is described in CPUVoxelizer.cs.

1.2.1 Rough flow of voxelization

The general flow of voxelization is as follows.

1. Set voxel resolution
2. Set the range for voxelization
3. Generate 3D array data to store voxel data
4. Generate voxels located on the surface of the mesh
5. Fill the voxels located inside the mesh from the voxel data representing the surface of the mesh

Voxelization in CPU is a static function of CPUVoxelizer class

CPUVoxelizer.cs

```
public class CPUVoxelizer
{
    public static void Voxelize (
        Mesh mesh,
        int resolution,
        out List<Vector3> voxels,
        out float unit,
        bool surfaceOnly = false
    ) {
    ...
    }
    ...
}
```

Execute by calling. If you specify the mesh and resolution you want to voxel in the argument and execute it, the voxel array voxels and the unit

representing the size of one voxel are returned via the reference argument.

In the following, I will explain what is done inside the Voxelize function along the general flow.

1.2.2 Set the voxel resolution

To make voxels, first set the voxel resolution. The finer the resolution, the smaller the cube will be built, so a detailed voxel model can be generated, but it requires more calculation time.

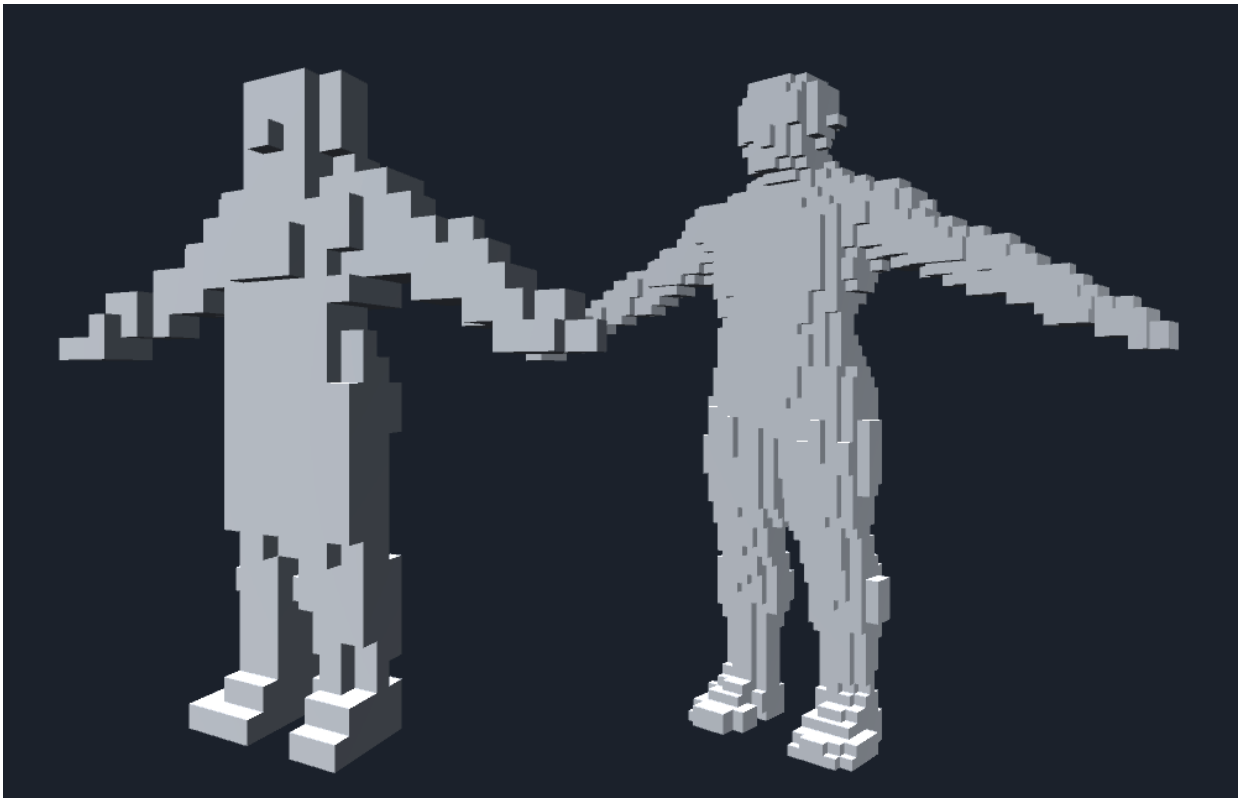

Figure 1.1: Differences in voxel resolution

1.2.3 Set the range for voxelization

Specifies the range to voxelize the target mesh model. If you specify the BoundingBox (the smallest rectangular parallelepiped that fits all the vertices of the model) of the mesh model as the voxelization range, you can voxelize the entire mesh model.

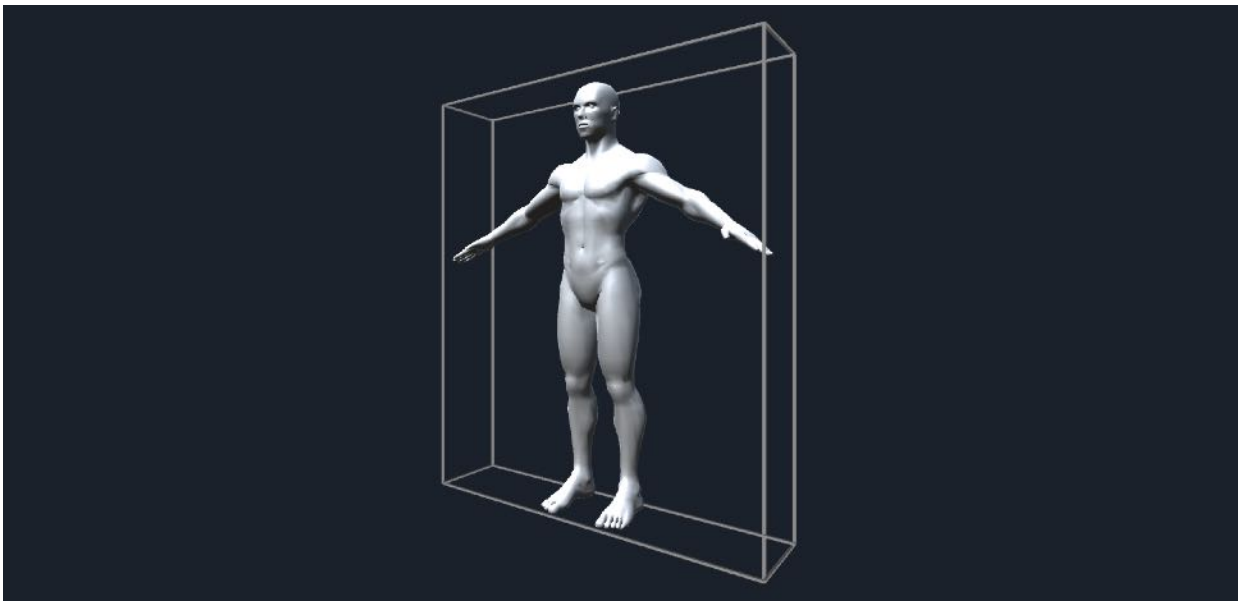

Figure 1.2: Mesh BoundingBox

It should be noted here that if the BoundingBox of the mesh model is used as it is as the voxelization range, problems will occur when voxelizing a mesh that has a surface that exactly overlaps the BoundingBox, such as a Cube mesh. ..

As will be described in detail later, when voxels are created, the intersection of the triangles and voxels that make up the mesh is determined. However, if the triangles and voxel surfaces overlap exactly, the intersection may not be determined correctly.

Therefore, the range in which the BoundingBox of the mesh model is expanded by "half the length of the unit length that constitutes one voxel" is specified as the voxelization range.

CPUVoxelizer.cs

```
mesh.RecalculateBounds();
var bounds = mesh.bounds;

// Calculate the unit lengths that make up one voxel from the
specified resolution
float maxLength = Mathf.Max(
    bounds.size.x,
    Mathf.Max(bounds.size.y, bounds.size.z)
```

```
);
unit = maxLength / resolution;

// Half the unit length
var hunit = unit * 0.5f;

// "Half the length of the unit length that makes up one voxel"
Expanded range
// Scope of voxels

// Minimum value of bounds to voxelize
var start = bounds.min - new Vector3 (unit, unit, unit);

// Maximum value of bounds to voxelize
var end = bounds.max + new Vector3 (unit, unit, unit);

// Size of bounds to voxelize
var size = end - start;
```

1.2.4 Generate 3D array data to store voxel data

The sample code provides a Voxel_t structure as a structure that represents voxels.

Voxel.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Voxel_t {
    public Vector3 position; // Voxel position
    public uint fill; // Flag whether voxels should be filled
       public uint front; // Flag whether the triangle that
intersects the voxel is the front when viewed from the
determined direction
    ...
}
```

A three-dimensional array of this Voxel_t is generated, and voxel data is stored in it.

CPUVoxelizer.cs

```
// Determine the size of 3D voxel data based on the unit length
of voxels and the range of voxelization
var width = Mathf.CeilToInt(size.x / unit);
var height = Mathf.CeilToInt(size.y / unit);
```

```
var depth = Mathf.CeilToInt(size.z / unit);
var volume = new Voxel_t[width, height, depth];
```

Also, in order to refer to the position and size of each voxel in the subsequent processing, generate an AABB array that matches the 3D voxel data in advance.

CPUVoxelizer.cs

```
var boxes = new Bounds[width, height, depth];
var voxelUnitSize = Vector3.one * unit;
for(int x = 0; x < width; x++)
{
    for(int y = 0; y < height; y++)
    {
        for(int z = 0; z < depth; z++)
        {
            var p = new Vector3(x, y, z) * unit + start;
            var aabb = new Bounds(p, voxelUnitSize);
            boxes[x, y, z] = aabb;
        }
    }
}
```

AABB

AABB (Axis-Aligned Bounding Box) is a rectangular parallelepiped boundary figure whose sides are parallel to the XYZ axes in 3D space.

AABB is often used for collision detection, and in some cases it is used for collision detection between two meshes or for simple collision detection between a certain mesh and a light beam.

If you want to make a strict collision detection for a mesh, you have to make a judgment for all the triangles

that make up the mesh, but if you only use AABB including the mesh, you can calculate at high speed, which is convenient.

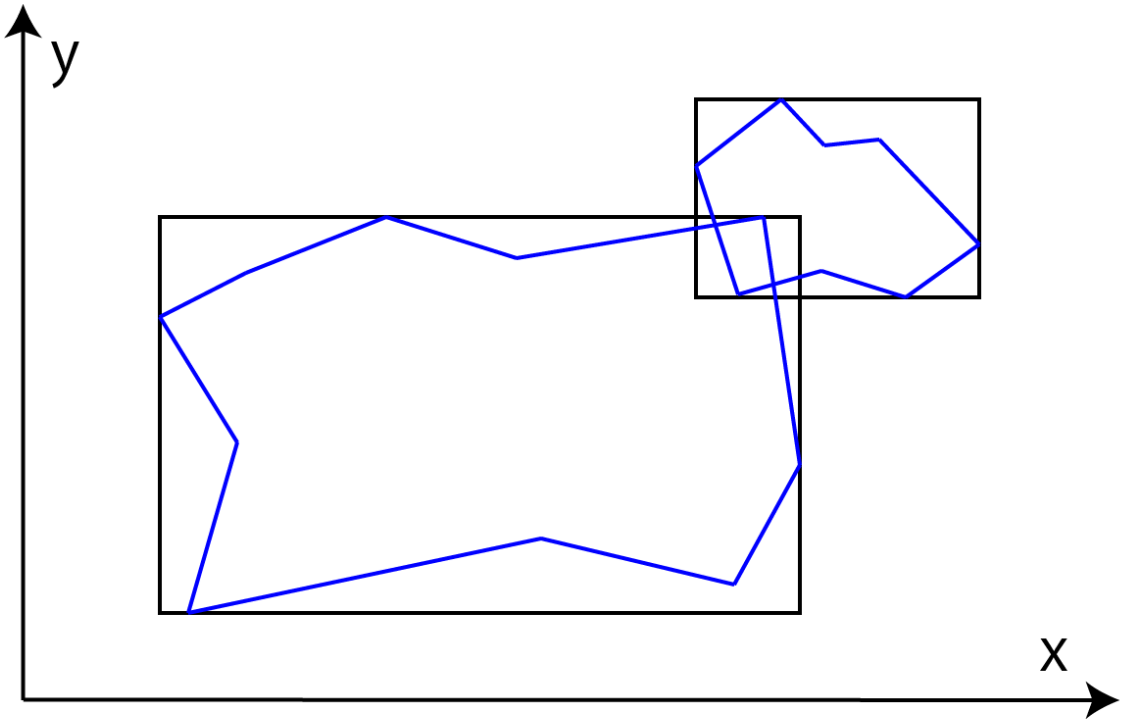

Figure 1.3: Collision detection between AABBs of two polygonal objects

1.2.5 Generate voxels located on the surface of the mesh

Generate voxels located on the surface of the mesh as shown in the figure below.

Figure 1.4: First, generate voxels located on the surface of the mesh, and then generate voxels based on it so as to fill the contents of the mesh.

To find the voxels located on the surface of the mesh, it is necessary to determine the intersection of each of the triangles that make up the mesh and the voxels.

Crossing judgment of triangle and voxel (crossing judgment algorithm using SAT)

SAT (Separating Axis Theorem) is used to determine the intersection of a triangle and a voxel. The intersection determination algorithm using SAT is not limited to triangles and voxels, but can be used for general purposes as intersection determination between convex surfaces.

The SAT has proved that:

A brief explanation of SAT (Separating Axis Theorem)

If you want a straight line with the entire object A on one side and the entire object B on the other, object A and object B will not intersect. Such a straight line that separates two objects is called a separation straight line, and the separation line is always orthogonal to the separation axis.

If the SAT finds an axis (separation axis) where the projections of two convex surfaces do not overlap, it can be derived that the two convex surfaces do not intersect because there is a straight line that separates the two convex surfaces. Conversely, if no separation axis is found, it can be determined that the two convex surfaces intersect. (If the shape is concave, it may not intersect even if the separation axis is not found.)

When a convex shape is projected onto an axis, the shadow of that shape appears as if it were projected onto a line that represents that axis. This can be represented as a line segment on the axis and can be represented by the range interval [min, max].

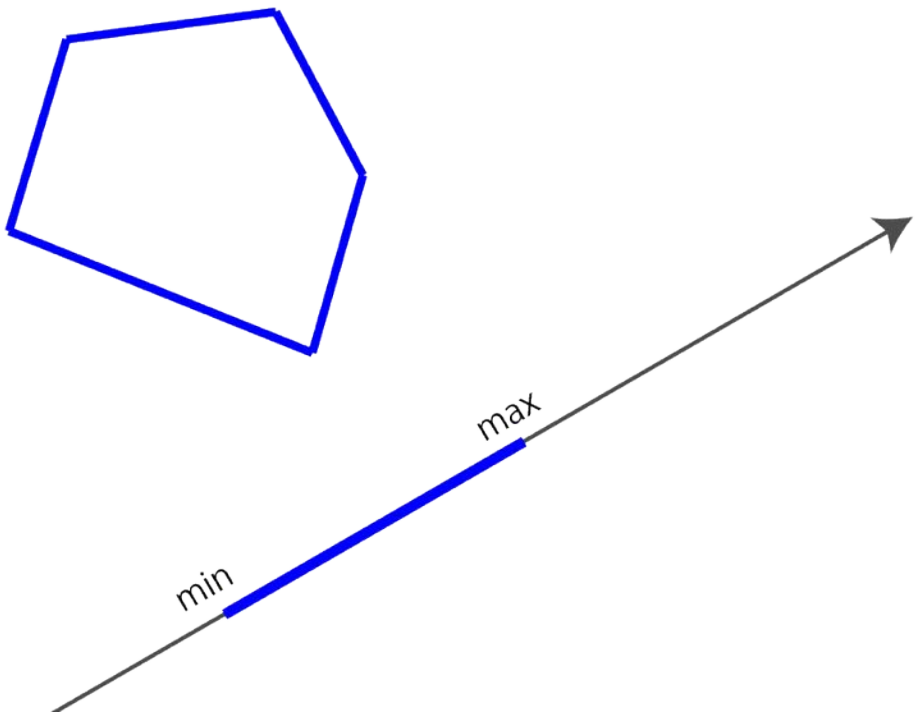

Figure 1.5: Convex shape projected onto a certain axis and the range of the convex shape projected on the axis (min, max)

As shown in the figure below, when there are two convex separation straight lines, the projection sections of the convex shape with respect to the separation axis orthogonal to the straight lines do not overlap.

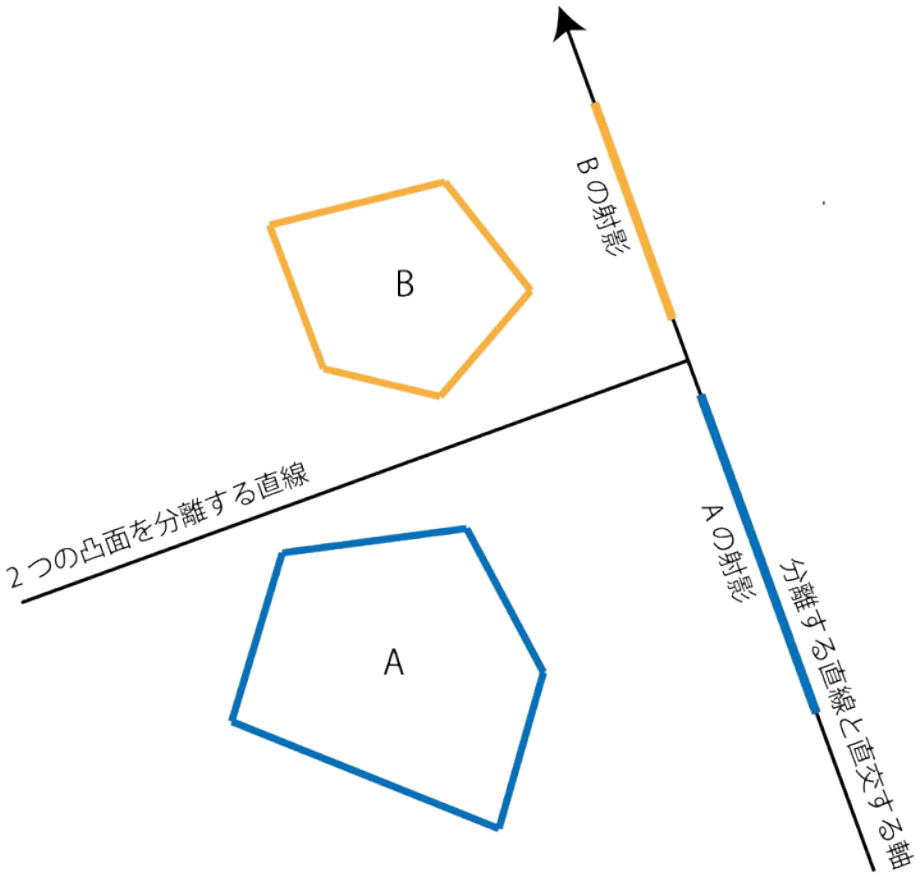

Figure 1.6: If there is a straight line that separates the two convex shapes, the projection sections on the axes orthogonal to the straight line do not overlap.

However, even with the same two convex surfaces, projections on other non-separable axes may overlap, as shown in the figure below.

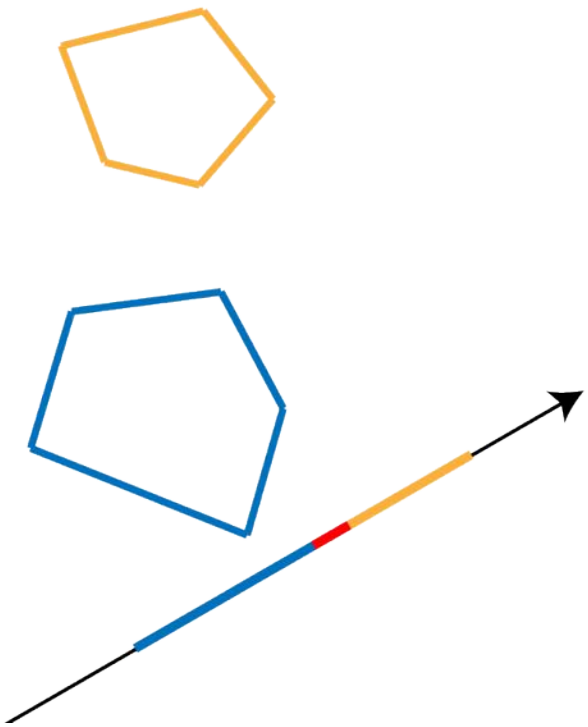

Figure 1.7: When projecting on an axis orthogonal to a straight line that does not separate the two convex shapes, the projections may overlap.

For some shapes, the possible axes of separation are obvious, and to determine the intersection of two such shapes A and B, two for each of the possible axes of separation. Crossing can be determined by projecting the shape and checking whether the two projection sections [Amin, Amax] and [Bmin, Bmax] overlap each other. Expressed in the formula, if Amax <Bmin or Bmax <Amin, the two intervals do not overlap.

The axis that can be the separation axis between the convex surfaces is

- Cross product of the side of convex surface 1 and the side of convex surface 2
- Convex 1 normal
- Convex 2 normal

From this, the axis that can be the separation axis between the triangle and the voxel (AABB) is

- Nine cross products obtained from the combination of the three sides of the triangle and the three orthogonal sides of AABB.
- AABB's three normals
- Triangle normal

Therefore, for each of these 13 axes, the intersection of the triangle and the voxel is determined by determining whether or not the projections overlap.

Since it may be useless to judge the intersection of one triangle and all voxel data, calculate the AABB including the triangle and judge the intersection with the voxels contained in it. ..

CPUVoxelizer.cs

```
// Calculate the triangle AABB
var min = tri.bounds.min - start;
var max = tri.bounds.max - start;
int iminX = Mathf.RoundToInt(min.x / unit);
int iminY = Mathf.RoundToInt(min.y / unit);
int iminZ = Mathf.RoundToInt(min.z / unit);
int imaxX = Mathf.RoundToInt(max.x / unit);
int imaxY = Mathf.RoundToInt(max.y / unit);
int imaxZ = Mathf.RoundToInt(max.z / unit);
iminX = Mathf.Clamp(iminX, 0, width - 1);
iminY = Mathf.Clamp(iminY, 0, height - 1);
iminZ = Mathf.Clamp(iminZ, 0, depth - 1);
imaxX = Mathf.Clamp(imaxX, 0, width - 1);
imaxY = Mathf.Clamp(imaxY, 0, height - 1);
imaxZ = Mathf.Clamp(imaxZ, 0, depth - 1);

// Judge the intersection with voxels in the triangular AABB
for(int x = iminX; x <= imaxX; x++) {
    for(int y = iminY; y <= imaxY; y++) {
        for(int z = iminZ; z <= imaxZ; z++) {
            if(Intersects(tri, boxes[x, y, z])) {
                ...
            }
        }
    }
}
```

The Intersects (Triangle, Bounds) function is used to determine the intersection of a triangle and a voxel.

CPUVoxelizer.cs

```
public static bool Intersects(Triangle tri, Bounds aabb)
{
    ...
}
```

In this function, the intersection judgment is performed for the above 13 axes, but the three normals of AABB are known (since they have sides along the XYZ axes, they are simply the X axis (1, 0, 0), Y-axis (0, 1, 0), Z-axis (0, 0, 1) normals), or so that the center of AABB is at the origin (0, 0, 0) The intersection judgment is optimized by translating the coordinates of the triangle and AABB.

CPUVoxelizer.cs

```
// Get the center coordinates of AABB and the extents of each
side
Vector3 center = aabb.center, extents = aabb.max - center;

// Translate the coordinates of the triangle so that the center
of AABB is at the origin (0, 0, 0)
Vector3 v0 = tri.a - center,
        v1 = tri.b - center,
        v2 = tri.c - center;

// Get the vector representing the three sides of the triangle
Vector3 f0 = v1 - v0,
        f1 = v2 - v1,
        f2 = v0 - v2;
```

First, we will make an intersection judgment based on the nine cross products obtained from the combination of the three sides of the triangle and the three orthogonal sides of AABB, but the direction of the three sides of AABB will be the XYZ axes. You can take advantage of the parallelism and omit the calculation to get the cross product.

CPUVoxelizer.cs

```
// Since the sides of AABB are the direction vectors x (1, 0,
0), y (0, 1, 0), z (0, 0, 1), respectively,
// You can get 9 different cross products without doing any
calculations
```

```
Vector3
     a00 = new Vector3 (0, -f0.z, f0.y), // Cross product of X
axis and f0
    a01 = new Vector3 (0, -f1.z, f1.y), // X and f1
    a02 = new Vector3(0, -f2.z, f2.y), // Xとf2
    a10 = new Vector3(f0.z, 0, -f0.x), // Yとf0
    a11 = new Vector3(f1.z, 0, -f1.x), // Yとf1
    a12 = new Vector3(f2.z, 0, -f2.x), // Yとf2
    a20 = new Vector3(-f0.y, f0.x, 0), // Zとf0
    a21 = new Vector3(-f1.y, f1.x, 0), // Zとf1
    a22 = new Vector3(-f2.y, f2.x, 0); // Zとf2

// Perform intersection judgment for 9 axes (described later)
// (If any one of the axes does not intersect, the triangle and
AABB do not intersect, so false is returned)
if (
    !Intersects(v0, v1, v2, extents, a00) ||
    !Intersects(v0, v1, v2, extents, a01) ||
    !Intersects(v0, v1, v2, extents, a02) ||
    !Intersects(v0, v1, v2, extents, a10) ||
    !Intersects(v0, v1, v2, extents, a11) ||
    !Intersects(v0, v1, v2, extents, a12) ||
    !Intersects(v0, v1, v2, extents, a20) ||
    !Intersects(v0, v1, v2, extents, a21) ||
    !Intersects(v0, v1, v2, extents, a22)
)
{
    return false;
}
```

The following function projects the triangle and AABB on these axes to determine the intersection.

CPUVoxelizer.cs

```
protected static bool Intersects(
    Vector3 v0,
    Vector3 v1,
    Vector3 v2,
    Vector3 extents,
    Vector3 axis
)
{
    ...
}
```

The point to note here is that the optimization is performed by bringing the center of AABB to the origin. It is not necessary to project all the vertices of AABB on the axis, and the interval on the axis can be obtained simply by projecting the vertex with the maximum value for the XYZ axes of AABB, that is, the extents of half the length of each side. I will.

The value r obtained by projecting extents represents the interval [-r, r] on the projection axis of AABB, which means that the projection can be calculated only once for AABB.

CPUVoxelizer.cs

```
// Project the vertices of the triangle on the axis
float p0 = Vector3.Dot(v0, axis);
float p1 = Vector3.Dot(v1, axis);
float p2 = Vector3.Dot(v2, axis);

// Project the extents with the maximum value for the XYZ axes
of AABB on the axis to get the value r
// Since the section of AABB is [-r, r], it is not necessary to
project all vertices for AABB.
float r =
    extents.x * Mathf.Abs(axis.x) +
    extents.y * Mathf.Abs(axis.y) +
    extents.z * Mathf.Abs(axis.z);

// Triangular projection section
float minP = Mathf.Min(p0, p1, p2);
float maxP = Mathf.Max(p0, p1, p2);

// Determine if the triangular section and the AABB section
overlap
return !((maxP < -r) || (r < minP));
```

Next to the discrimination based on the nine cross products, the discrimination is performed based on the three normals of AABB.

Using the characteristic that the normal of AABB is parallel to the XYZ axes, the coordinate values are translated so as to bring the center of AABB to the origin, so the minimum value for the XYZ component of each vertex of the triangle is simply. Crossing judgment can be performed simply by comparing the maximum value and extends.

CPUVoxelizer.cs

```
// X axis
if (
    Mathf.Max(v0.x, v1.x, v2.x) < -extents.x ||
    Mathf.Min(v0.x, v1.x, v2.x) > extents.x
)
{
    return false;
}

// Y axis
if (
    Mathf.Max (v0.y, v1.y, v2.y) <-extents.y ||
    Mathf.Min (v0.y, v1.y, v2.y)> extents.y
)
{
    return false;
}

// Z axis
if (
    Mathf.Max(v0.z, v1.z, v2.z) < -extents.z ||
    Mathf.Min(v0.z, v1.z, v2.z) > extents.z
)
{
    return false;
}
```

Lastly, regarding the triangular normal, we are making a judgment about the intersection of the Plane with the triangular normal and AABB.

CPUVoxelizer.cs

```
var normal = Vector3.Cross(f1, f0).normalized;
var pl = new Plane(normal, Vector3.Dot(normal, tri.a));
return Intersects(pl, aabb);
```

The Intersects (Plane, Bounds) function determines the intersection of Plane and AABB.

CPUVoxelizer.cs

```
public static bool Intersects(Plane pl, Bounds aabb)
{
```

```
    Vector3 center = aabb.center;
    var extents = aabb.max - center;

    // Project the extents on the normal of Plane
    var r =
        extents.x * Mathf.Abs(pl.normal.x) +
        extents.y * Mathf.Abs (pl.normal.y) +
        extents.z * Mathf.Abs(pl.normal.z);

     // Calculate the distance between Plane and the center of
AABB
    var s = Vector3.Dot(pl.normal, center) - pl.distance;

    // Determine if s is in the range [-r, r]
    return Mathf.Abs(s) <= r;
}
```

Write intersected voxels to array data

If the intersecting voxels can be determined for one triangle, the fill flag of the voxel data is set, and the front flag is set to indicate whether the triangle is front or back when viewed from the determined direction. (The front flag will be described later)

Some voxels may intersect both front-facing and back-facing triangles, in which case the front flag should prioritize the back.

CPUVoxelizer.cs

```
if(Intersects(tri, boxes[x, y, z])) {
    // Get voxels at intersecting (x, y, z)
    var voxel = volume[x, y, z];

    // Set the voxel position
    voxel.position = boxes[x, y, z].center;

    if(voxel.fill & 1 == 0) {
        // If the voxels are not yet filled
         // Flag the triangle that intersects the voxel for the
front
        voxel.front = front;
    } else {
        // If the voxel is already filled with other triangles
        // Give priority to the flag on the back
```

```
        voxel.front = voxel.front & front;
    }

    // Flag to fill voxels
    voxel.fill = 1;
    volume[x, y, z] = voxel;
}
```

The front flag is required for the "process to fill the contents of the mesh" described later, and sets whether it is the front or the back when viewed from the "direction to fill the contents".

In the sample code, the contents of the mesh are filled in the forward (0, 0, 1) direction, so it is determined whether the triangle is in front when viewed from forward (0, 0, 1).

If the inner product of the normal of the triangle and the direction to fill the voxels is 0 or less, it means that the triangle is the front when viewed from that direction.

CPUVoxelizer.cs

```
public class Triangle {
    public Vector3 a, b, c; // 3 points that make up a triangle
     public bool frontFacing; // Flag whether the triangle is a
surface when viewed from the direction of filling the voxels
    public Bounds bounds; // Triangular AABB

     public Triangle (Vector3 a, Vector3 b, Vector3 c, Vector3
dir) {
        this.a = a;
        this.b = b;
        this.c = c;

         // Determine if the triangle is front when viewed from
the direction of filling the voxels
        var normal = Vector3.Cross(b - a, c - a);
        this.frontFacing = (Vector3.Dot(normal, dir) <= 0f);

        ...
    }
}
```

1.2.6 Fill the voxels located inside the mesh from the voxel data representing the surface of the mesh

Now that we have calculated the voxel data located on the mesh surface, we will fill in the inside.

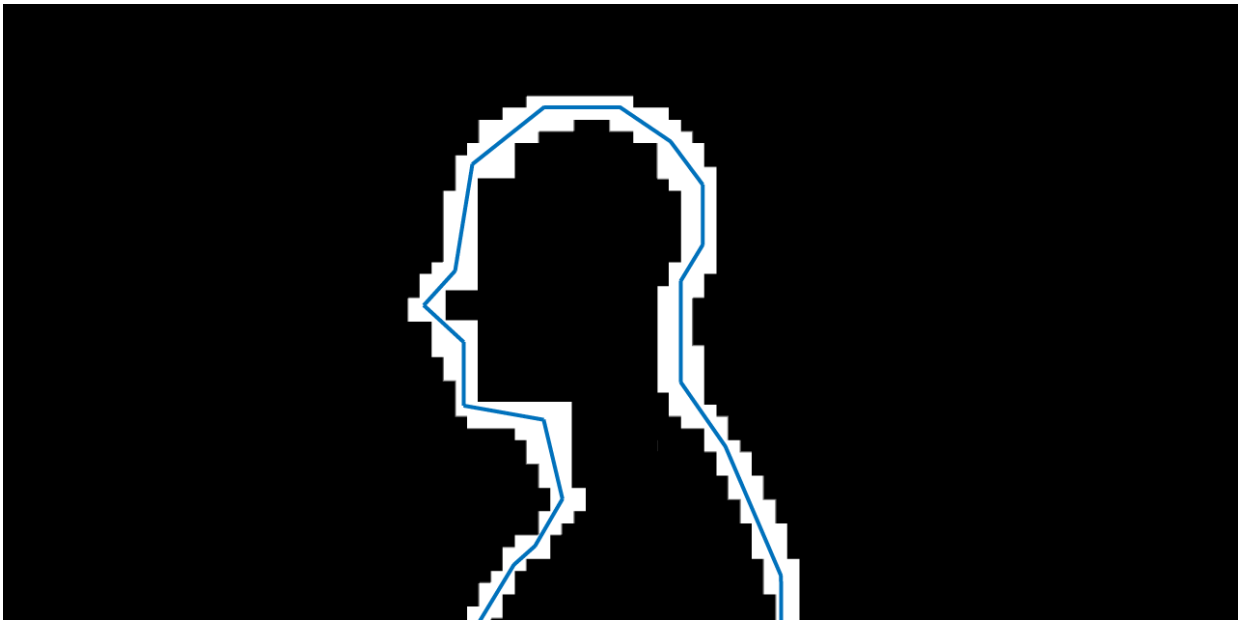

Figure 1.8: State after generating voxel data located on the mesh surface

Flow to fill voxels

Search for voxels that are facing forward when viewed from the direction of filling the voxels.

Empty voxels will pass through as shown in the figure below.

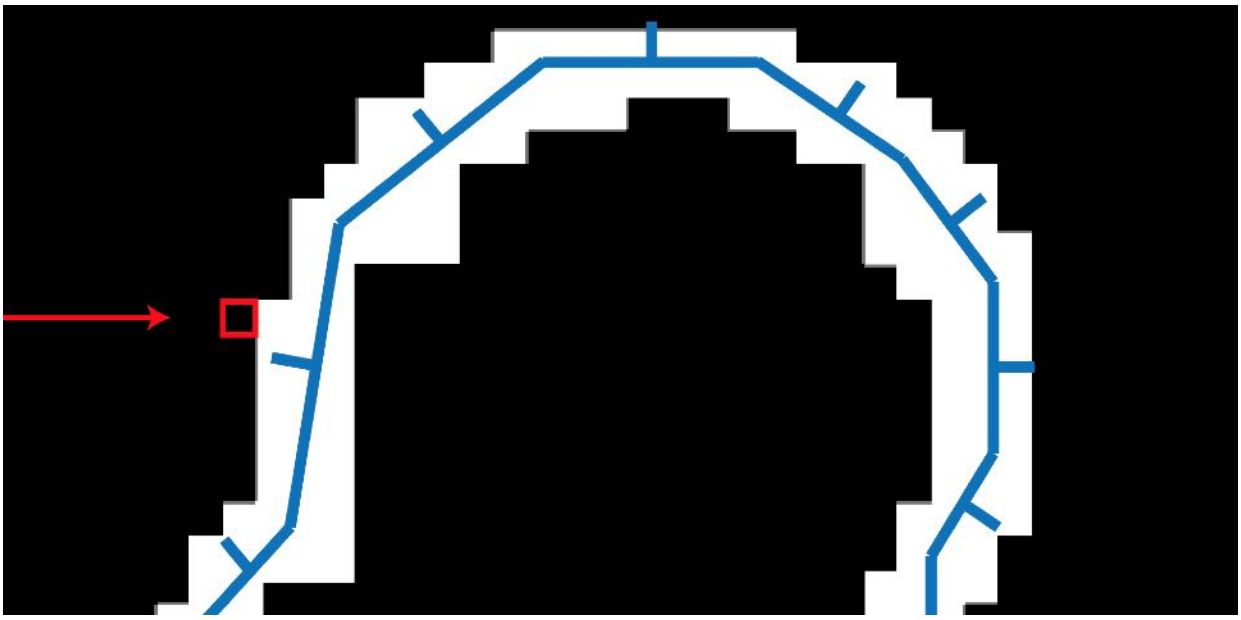

Figure 1.9: Search for voxels facing forward when viewed from the voxel-filling direction Empty voxels pass through (arrows fill voxels and frames represent voxel positions being searched)

Once you find a voxel that is facing the front, proceed through the voxel that is facing the front.

Figure 1.10: Finding a voxel facing the front (the line coming out of the mesh surface is the mesh normal, and in the figure the mesh normal and the

voxel filling direction are opposite, so the position of the frame You can see that the voxel is located in the front)

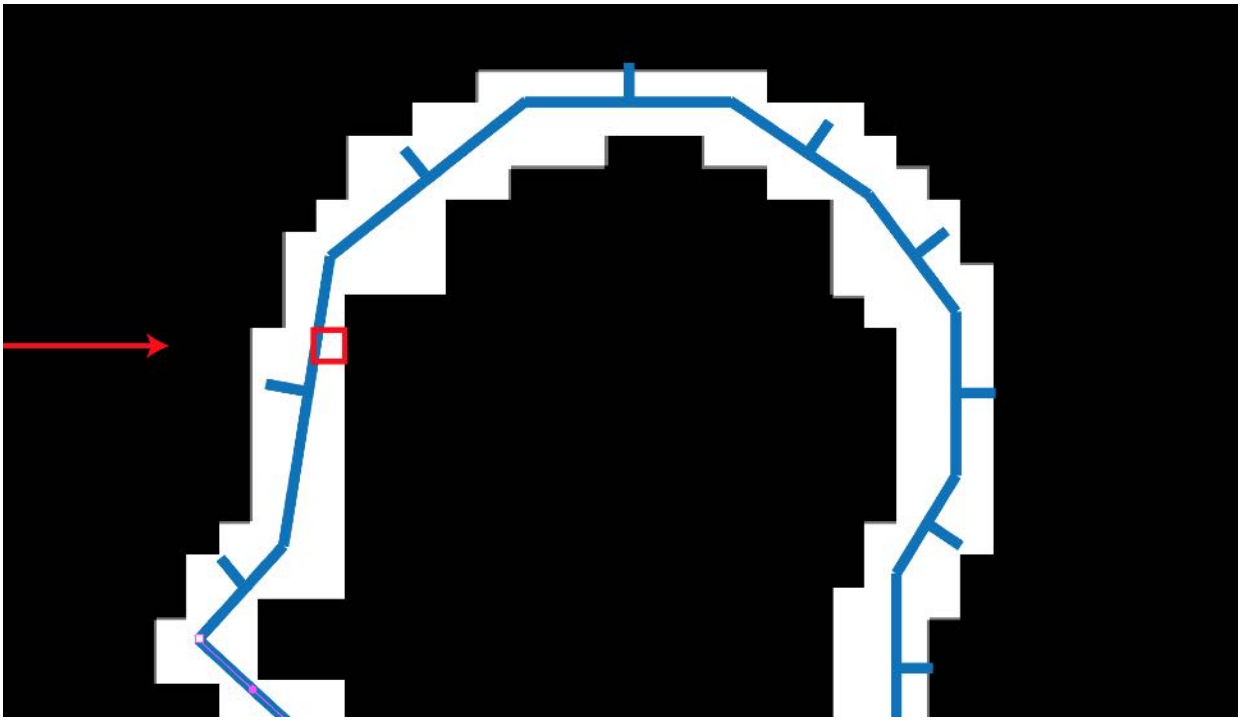

Figure 1.11: Go through a voxel facing the front

After passing through the voxels facing the front, you will reach the inside of the mesh.

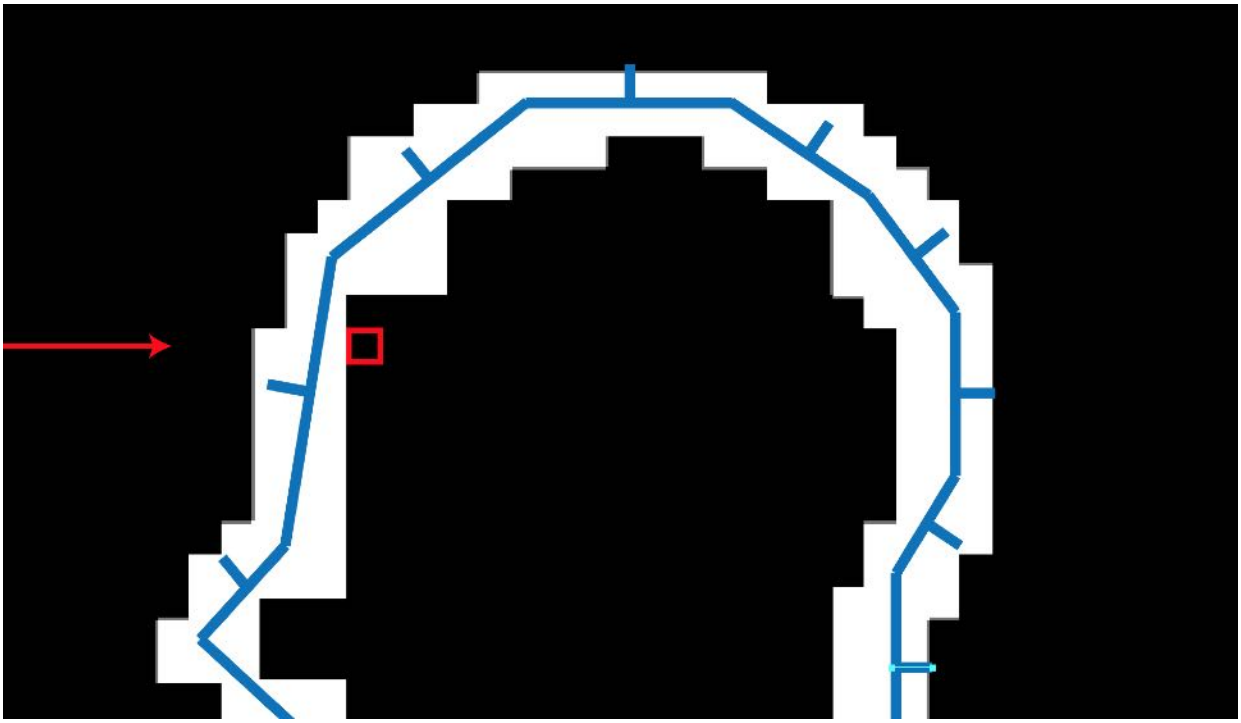

Figure 1.12: Passing through a voxel facing the front and reaching the inside of the mesh

Proceed through the inside of the mesh and fill the voxels that have arrived.

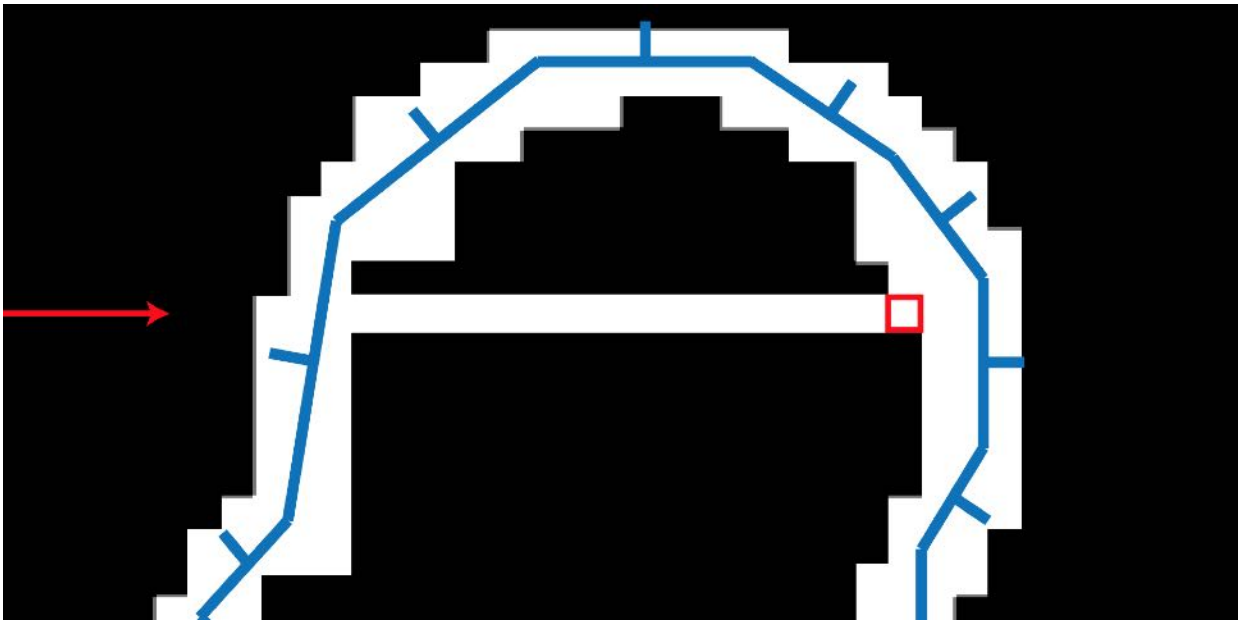

Figure 1.13: Fill the reached voxels as they fill the inside of the mesh

Then, when you reach the voxel facing the back when viewed from the direction of filling the voxel, you can see that the inside of the mesh has been filled. Go through the voxels facing the back, and when you reach the outside of the mesh, you will start searching for the voxels facing the front again.

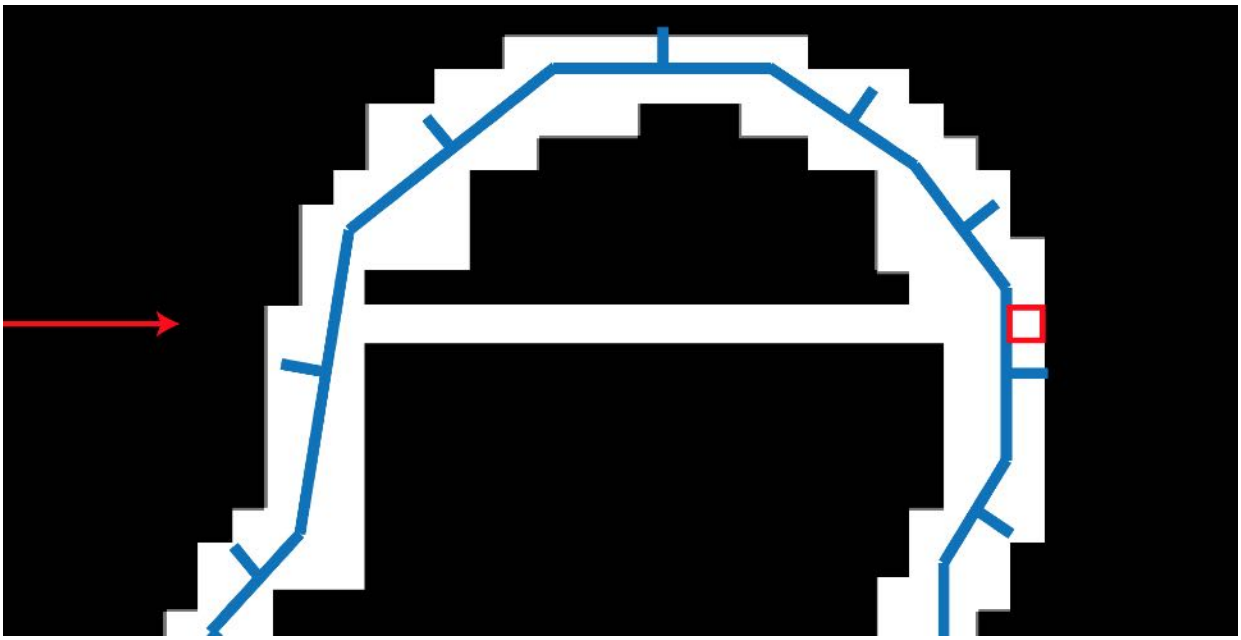

Figure 1.14: Proceed through the voxel facing backwards from the direction of filling the voxel and then out of the mesh

Implementation to fill voxels

As determined in the previous section, the inside is filled in the forward (0, 0, 1) direction, so in a 3D voxel array, the inside is filled in the z direction.

The process of filling the contents starts from volume [x, y, 0] on the front side in the z direction and proceeds to volume [x, y, depth -1].

CPUVoxelizer.cs

```
// Fill the inside of the mesh
for(int x = 0; x < width; x++)
{
    for(int y = 0; y < height; y++)
    {
```

```
          // Fill the inside of the mesh from the front side in
the z direction to the back side
        for(int z = 0; z < depth; z++)
        {
            ...
        }
    }
}
```

Based on the front flag (front or back in the z direction) already written in the voxel data, the process proceeds according to the above-mentioned flow of filling voxels.

CPUVoxelizer.cs

```
...
// Fill the inside of the mesh from the front side in the z
direction to the back side
for(int z = 0; z < depth; z++)
{
    // Ignore if (x, y, z) is empty
    if (volume[x, y, z].IsEmpty()) continue;

    // Go through the voxels located in front
    int ifront = z;
     for(; ifront < depth && volume[x, y, ifront].IsFrontFace();
ifront++) {}

    // If you go to the end, it's over
    if(ifront >= depth) break;

    // Find the voxels located on the back
    int iback = ifront;

    // Go inside the mesh
      for (; iback < depth && volume[x, y, iback].IsEmpty();
iback++) {}

    // If you go to the end, it's over
    if (iback >= depth) break;

    // Determine if (x, y, iback) is on the back
    if(volume[x, y, iback].IsBackFace()) {
        // Follow the voxels located on the back
                  for (; iback < depth && volume[x, y,
iback].IsBackFace(); iback++) {}
    }
```

```
    // Fill voxels from (x, y, ifront) to (x, y, iback)
    for(int z2 = ifront; z2 < iback; z2++)
    {
        var p = boxes [x, y, z2] .center;
        var voxel = volume[x, y, z2];
        voxel.position = p;
        voxel.fill = 1;
        volume[x, y, z2] = voxel;
    }

    // Proceed through the loop until it finishes processing (x,
y, iback)
    z = iback;
}
```

Up to this point, we have obtained voxel data filled with the contents of the mesh.

Since the processed 3D voxel data contains empty voxels, CPUVoxelizer.Voxelize returns only the voxels that make up the surface of the mesh and the filled contents.

CPUVoxelizer.cs

```
// Get non-empty voxels
voxels = new List<Voxel_t>();
for(int x = 0; x < width; x++) {
    for(int y = 0; y < height; y++) {
        for(int z = 0; z < depth; z++) {
            if(!volume[x, y, z].IsEmpty())
            {
                voxels.Add(volume[x, y, z]);
            }
        }
    }
}
```

In CPUVoxelizerTest.cs, a mesh is constructed using the voxel data obtained by CPUVoxelizer, and the voxels are visualized.

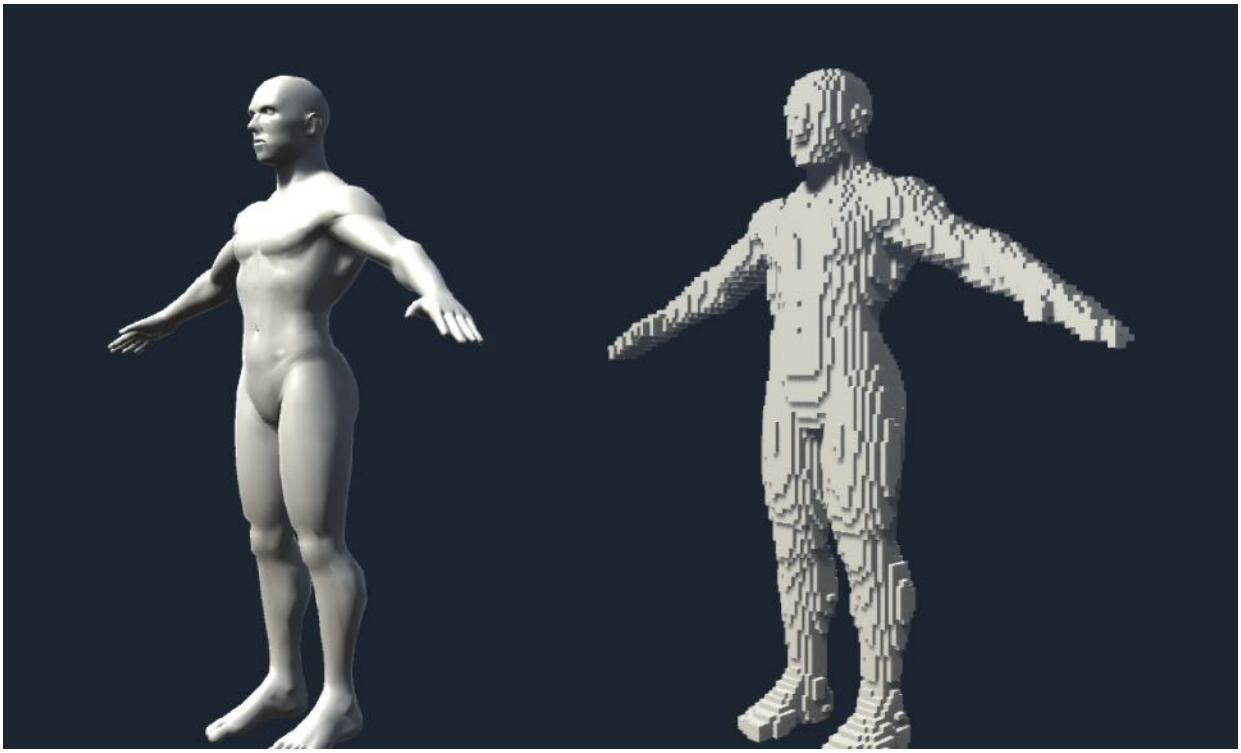

Figure 1.15: Demo of voxel data obtained by CPUVoxelizer.Voxelize visualized as a mesh (CPUVoxelizerTest.scene)

1.3 Voxel mesh representation

The VoxelMesh class describes the process of constructing a mesh based on the voxel data array Voxel_t [] and the unit length information of one voxel.

CPUVoxelizerTest.cs in the previous section uses this class to generate voxel mesh.

VoxelMesh.cs

```
public class VoxelMesh {

    public static Mesh Build (Voxel_t[] voxels, float size)
    {
        var hsize = size * 0.5f;
        var forward = Vector3.forward * hsize;
        var back = -forward;
        var up = Vector3.up * hsize;
        var down = -up;
```

```
        var right = Vector3.right * hsize;
        var left = -right;

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

        for(int i = 0, n = voxels.Length; i < n; i++)
        {
            if(voxel[i].fill == 0) continue;

            var p = voxels[i].position;

              // The vertices of the eight corners that make up
the Cube that represents one voxel
            var corners = new Vector3[8] {
                p + forward + left + up,
                p + back + left + up,
                p + back + right + up,
                p + forward + right + up,

                p + forward + left + down,
                p + back + left + down,
                p + back + right + down,
                p + forward + right + down,
            };

            // Build the 6 faces that make up the Cube

            // up
            AddTriangle(
                corners[0], corners[3], corners[1],
                up, vertices, normals, triangles
            );
            AddTriangle(
                corners[2], corners[1], corners[3],
                up, vertices, normals, triangles
            );

            // down
            AddTriangle(
                corners[4], corners[5], corners[7],
                down, vertices, normals, triangles
            );
            AddTriangle(
                corners[6], corners[7], corners[5],
                down, vertices, normals, triangles
            );
```

```
            // right
            AddTriangle(
                corners[7], corners[6], corners[3],
                right, vertices, normals, triangles
            );
            AddTriangle(
                corners[2], corners[3], corners[6],
                right, vertices, normals, triangles
            );

            // left
            AddTriangle(
                corners[5], corners[4], corners[1],
                left, vertices, normals, triangles
            );
            AddTriangle(
                corners[0], corners[1], corners[4],
                left, vertices, normals, triangles
            );

            // forward
            AddTriangle(
                corners[4], corners[7], corners[0],
                forward, vertices, normals, triangles
            );
            AddTriangle(
                corners[3], corners[0], corners[7],
                forward, vertices, normals, triangles
            );

            // back
            AddTriangle(
                corners[6], corners[5], corners[2],
                forward, vertices, normals, triangles
            );
            AddTriangle(
                corners[1], corners[2], corners[5],
                forward, vertices, normals, triangles
            );
        }

        var mesh = new Mesh ();
        mesh.SetVertices (vertices);

          // Apply 32bit index format if the number of vertices
exceeds the number that can be supported by 16bit
        mesh.indexFormat =
```

```
            (vertices.Count <= 65535)
            ? IndexFormat.UInt16 : IndexFormat.UInt32;
        mesh.SetNormals(normals);
                            mesh.SetIndices(triangles.ToArray(),
MeshTopology.Triangles, 0);
        mesh.RecalculateBounds();
        return mesh;
    }
}
```

1.4 Implementation on GPU

From here, I will explain how to execute voxelization implemented by CPU Voxelizer faster using GPU.

The voxelization algorithm implemented by CPUVoxelizer can be parallelized for each coordinate in the lattice space separated by the unit length of voxels on the XY plane.

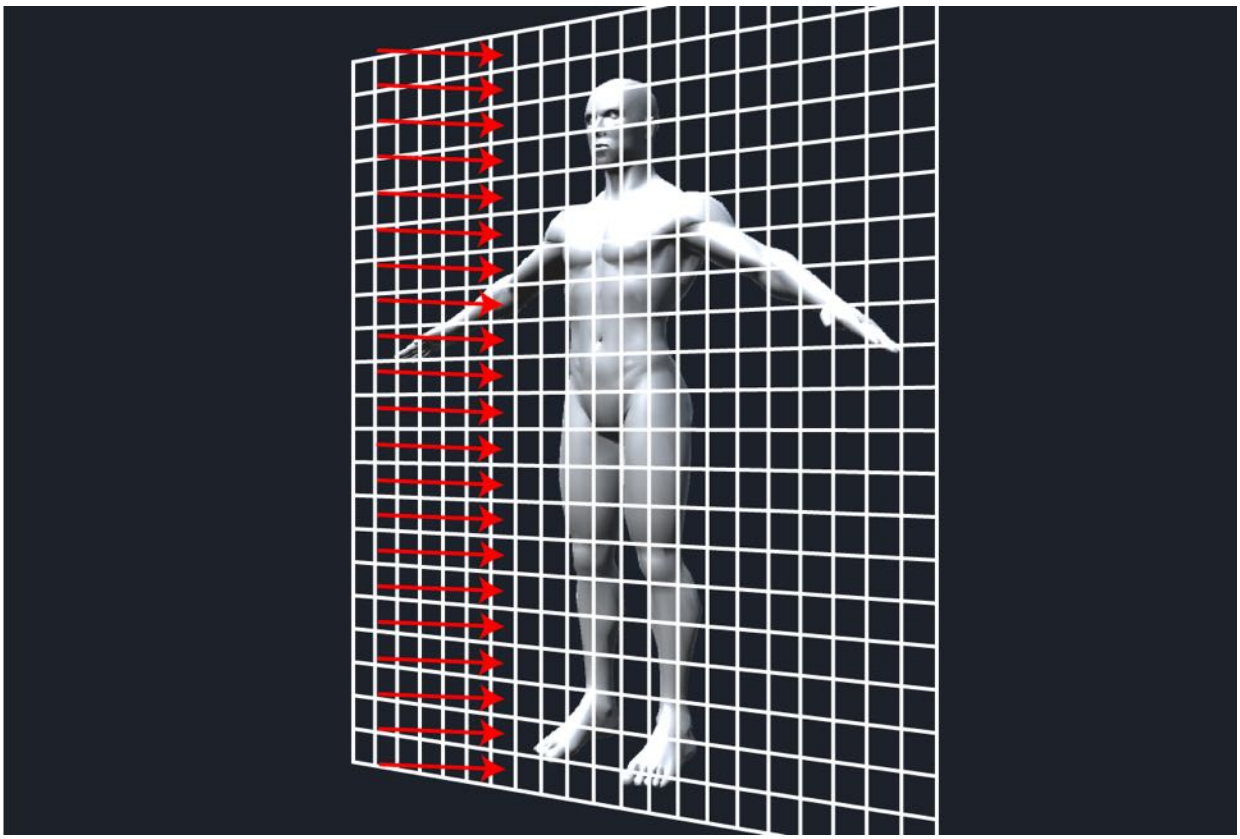

Figure 1.16: Lattice space delimited by unit length of voxels on the XY plane. Voxelization can be parallelized for each lattice, so GPU implementation is possible.

If you allocate each process that can be parallelized to the GPU thread, you can execute the process at high speed thanks to the high-speed parallel computing of the GPU.

The implementation of voxelization on GPU is described in GPU Boxelizer.cs and Voxelizer.compute.

(The basics of Compute Shader, which appears in this section and is indispensable for GPGPU programming in Unity, are explained in Unity Graphics Programming vol.1 "Introduction to Compute Shader")

Voxelization on GPU is a static function of GPUVoxelizer class

GPUVoxelizer.cs

```
public class GPUVoxelizer
{
    public static GPUVoxelData Voxelize (
        ComputeShader voxelizer,
        Mesh mesh,
        int resolution
    ) {
    ...
    }
}
```

Execute by calling. When executed with Voxelizer.compute, the mesh to be voxelized, and the resolution specified as arguments, GPUVoxelData indicating voxel data is returned.

1.4.1 Setting up the data required for voxel generation on the GPU

Follow the general flow of voxelization (1) to (3) to set up the data required for voxel generation.

GPUVoxelizer.cs

```
public static GPUVoxelData Voxelize (
    ComputeShader voxelizer,
    Mesh mesh,
    int resolution
) {
    // Same process as CPUVoxelizer.Voxelize -------
    mesh.RecalculateBounds();
    var bounds = mesh.bounds;

    float maxLength = Mathf.Max(
        bounds.size.x,
        Mathf.Max(bounds.size.y, bounds.size.z)
    );
    var unit = maxLength / resolution;

    var hunit = unit * 0.5f;

    var start = bounds.min - new Vector3 (unit, unit, unit);
    var end = bounds.max + new Vector3 (unit, unit, unit);
    var size = end - start;

    int width = Mathf.CeilToInt(size.x / unit);
    int height = Mathf.CeilToInt(size.y / unit);
    int depth = Mathf.CeilToInt(size.z / unit);
    // ------- So far the same as CPUVoxelizer.Voxelize
    ...
}
```

The array of Voxel_t is defined as ComputeBuffer so that it can be handled on the GPU. The point to note here is that the Voxel_t array generated as a 3D array is defined as a 1D array in the CPU implementation.

This is because it is difficult for GPUs to handle multidimensional arrays, so define it as a one-dimensional array and get the index on the one-dimensional array from the three-dimensional position (x, y, z) in Compute Shader. By doing so, the one-dimensional array is processed like a three-dimensional array.

GPUVoxelizer.cs

```
// Generate a ComputeBuffer representing a Voxel_t array
var voxelBuffer = new ComputeBuffer(
    width * height * depth,
```

```
    Marshal.SizeOf(typeof(Voxel_t))
);
var voxels = new Voxel_t[voxelBuffer.count];
voxelBuffer.SetData (voxels); // Initialize
```

Transfer these set up data to the GPU side.

GPUVoxelizer.cs

```
// Transfer voxel data to GPU side
voxelizer.SetVector("_Start", start);
voxelizer.SetVector("_End", end);
voxelizer.SetVector("_Size", size);

voxelizer.SetFloat("_Unit", unit);
voxelizer.SetFloat("_InvUnit", 1f / unit);
voxelizer.SetFloat ("_ HalfUnit", unit);
voxelizer.SetInt("_Width", width);
voxelizer.SetInt("_Height", height);
voxelizer.SetInt("_Depth", depth);
```

Generate a Compute Buffer that represents the mesh in order to determine the intersection of the triangles and voxels that make up the mesh.

GPUVoxelizer.cs

```
// Generate a ComputeBuffer that represents the vertex array of
the mesh
var vertices = mesh.vertices;
var vertBuffer = new ComputeBuffer(
    vertices.Length,
    Marshal.SizeOf(typeof(Vector3))
);
vertBuffer.SetData(vertices);

// Generate a ComputeBuffer that represents a triangular array
of meshes
var triangles = mesh.triangles;
var triBuffer = new ComputeBuffer(
    triangles.Length,
    Marshal.SizeOf(typeof(int))
);
triBuffer.SetData(triangles);
```

1.4.2 Generate voxels located on the surface of the mesh on the GPU

In the process of generating voxels located on the surface of the mesh on the GPU, after generating voxels that intersect the front-facing triangle, the voxels that intersect the back-facing triangle are generated.

This is because the value of the front flag written to the voxel may not be uniquely determined when multiple triangles intersect for the voxel at the same position.

One thing to keep in mind when using GPU parallel computing is the indefiniteness of the results due to multiple threads accessing the same data at the same time.

In the process of generating this surface, priority is given to the value of the front flag being the back (false), and voxel generation is executed in the order of front → back to prevent indeterminacy of the result.

Transfer the mesh data you just generated to the GPU kernel SurfaceFront, which creates voxels that intersect the front-facing triangles.

GPUVoxelizer.cs

```
// Transfer mesh data to GPU kernel SurfaceFront
var surfaceFrontKer = new Kernel(voxelizer, "SurfaceFront");
voxelizer.SetBuffer(surfaceFrontKer.Index,       "_VoxelBuffer",
voxelBuffer);
voxelizer.SetBuffer(surfaceFrontKer.Index,        "_VertBuffer",
vertBuffer);
voxelizer.SetBuffer(surfaceFrontKer.Index,         "_TriBuffer",
triBuffer);

// Set the number of triangles that make up the mesh
var triangleCount = triBuffer.count / 3; // (the number of
vertex indexes that make up the triangle / 3) is the number of
triangles
voxelizer.SetInt("_TriangleCount", triangleCount);
```

This process is performed in parallel for each triangle that makes up the mesh. Set the kernel thread group to (triangleCount / number of kernel

threads + 1, 1, 1) so that all triangles are processed and run the kernel.

GPUVoxelizer.cs

```
// Build a voxel that intersects a front-facing triangle
voxelizer.Dispatch(
    surfaceFrontKer.Index,
    triangleCount / (int)surfaceFrontKer.ThreadX + 1,
    (int)surfaceFrontKer.ThreadY,
    (int)surfaceFrontKer.ThreadZ
);
```

Since the SurfaceFront kernel only processes triangles that are facing the front, it checks the front and back of the triangle, returns to finish the process if it is the back, and builds the mesh surface if it is the front. Is running.

Voxelizer.compute

```
[numthreads(8, 1, 1)]
void SurfaceFront (uint3 id : SV_DispatchThreadID)
{
    // return if the number of triangles is exceeded
    int idx = (int)id.x;
    if(idx >= _TriangleCount) return;

    // Get the vertex position of the triangle and the front and
back flags
    float3 va, vb, vc;
    bool front;
    get_triangle(idx, va, vb, vc, front);

    // return if it is on the back
    if (!front) return;

    // Build a mesh surface
    surface(va, vb, vc, front);
}
```

The get_triangle function gets the vertex position and front / back flag of the triangle based on the mesh data (_TriBuffer representing the vertex index that constitutes the triangle and _VertBuffer representing the vertex) passed from the CPU to the GPU side.

Voxelizer.compute

```
void get_triangle(
    int idx,
    out float3 va, out float3 vb, out float3 vc,
    out bool front
)
{
    int ia = _TriBuffer[idx * 3];
    int ib = _TriBuffer[idx * 3 + 1];
    int ic = _TriBuffer[idx * 3 + 2];

    va = _VertBuffer [ia];
    vb = _VertBuffer[ib];
    vc = _VertBuffer[ic];

     // Determine if the triangle is front or back when viewed
from the forward (0, 0, 1) direction
    float3 normal = cross ((vb - va), (vc - vb));
    front = dot(normal, float3(0, 0, 1)) < 0;
}
```

The surface function that determines the intersection of a voxel and a triangle and writes the result to the voxel data takes time to acquire the index of the voxel data generated as a one-dimensional array, but the content of the process is implemented on the CPU Voxelizer. It will be almost the same as.

Voxelizer.compute

```
void surface (float3 va, float3 vb, float3 vc, bool front)
{
    // Calculate the triangle AABB
    float3 tbmin = min (min (va, vb), vc);
    float3 tbmax = max(max(va, vb), vc);

    float3 bmin = tbmin - _Start;
    float3 bmax = tbmax - _Start;
    int iminX = round(bmin.x / _Unit);
    int iminY = round(bmin.y / _Unit);
    int iminZ = round(bmin.z / _Unit);
    int imaxX = round(bmax.x / _Unit);
    int imaxY = round(bmax.y / _Unit);
    int imaxZ = round(bmax.z / _Unit);
    iminX = clamp(iminX, 0, _Width - 1);
    iminY = clamp(iminY, 0, _Height - 1);
    iminZ = clamp(iminZ, 0, _Depth - 1);
    imaxX = clamp(imaxX, 0, _Width - 1);
    imaxY = clamp(imaxY, 0, _Height - 1);
```

```
    imaxZ = clamp(imaxZ, 0, _Depth - 1);

    // Judge the intersection with voxels in the triangular AABB
    for(int x = iminX; x <= imaxX; x++) {
        for(int y = iminY; y <= imaxY; y++) {
            for(int z = iminZ; z <= imaxZ; z++) {
                // Generate AABB for voxels located at (x, y, z)
                     float3 center = float3(x, y, z) * _Unit +
_Start;
                AABB aabb;
                aabb.min = center - _HalfUnit;
                aabb.center = center;
                aabb.max = center + _HalfUnit;
                if(intersects_tri_aabb(va, vb, vc, aabb))
                {
                     // Get the index of a one-dimensional voxel
array from the position of (x, y, z)
                    uint vid = get_voxel_index(x, y, z);
                    Voxel voxel = _VoxelBuffer[vid];
                       voxel.position = get_voxel_position(x, y,
z);
                    voxel.front = front;
                    voxel.fill = true;
                    _VoxelBuffer[vid] = voxel;
                }
            }
        }
    }
}
```

Now that we have generated voxels for the front-facing triangles, let's move on to the back-facing triangles.

Transfer the mesh data to the GPU kernel SurfaceBack, which generates voxels that intersect the triangle facing the back, and execute it as before.

GPUVoxelizer.cs

```
var surfaceBackKer = new Kernel(voxelizer, "SurfaceBack");
voxelizer.SetBuffer(surfaceBackKer.Index,        "_VoxelBuffer",
voxelBuffer);
voxelizer.SetBuffer(surfaceBackKer.Index,         "_VertBuffer",
vertBuffer);
voxelizer.SetBuffer(surfaceBackKer.Index,          "_TriBuffer",
triBuffer);
voxelizer.Dispatch(
```

```
    surfaceBackKer.Index,
    triangleCount / (int)surfaceBackKer.ThreadX + 1,
    (int)surfaceBackKer.ThreadY,
    (int)surfaceBackKer.ThreadZ
);
```

The processing of SurfaceBack is the same as SurfaceFront except that it returns a return when the triangle is facing the front. By running SurfaceBack after SurfaceFront, the voxel's front flag will be overridden by SurfaceBack, even if there are voxels that intersect both the front-facing triangle and the back-facing triangle. It will be prioritized to face the back.

Voxelizer.compute

```
[numthreads(8, 1, 1)]
void SurfaceBack (uint3 id : SV_DispatchThreadID)
{
    int idx = (int)id.x;
    if(idx >= _TriangleCount) return;

    float3 va, vb, vc;
    bool front;
    get_triangle(idx, va, vb, vc, front);

    // return if front
    if (front) return;

    surface(va, vb, vc, front);
}
```

1.4.3 Fill the voxels located inside the mesh from the voxel data representing the surface of the mesh on the GPU.

The volume kernel is used to fill the inside of the mesh.

The Volume kernel prepares and executes a thread for each coordinate in the lattice space separated by the unit length of voxels on the XY plane. In other words, in the case of CPU implementation, the place where the double loop was executed for XY coordinates is parallelized by GPU and speeded up.

GPUVoxelizer.cs

```
// Transfer voxel data to Volume kernel
var volumeKer = new Kernel(voxelizer, "Volume");
voxelizer.SetBuffer(volumeKer.Index,              "_VoxelBuffer",
voxelBuffer);

// Fill the inside of the mesh
voxelizer.Dispatch(
    volumeKer.Index,
    width / (int)volumeKer.ThreadX + 1,
    height / (int)volumeKer.ThreadY + 1,
    (int)volumeKer.ThreadZ
);
```

The Volume kernel implementation is similar to the one implemented in GPU Foxelizer.

Voxelizer.compute

```
[numthreads(8, 8, 1)]
void Volume (uint3 id : SV_DispatchThreadID)
{
    int x = (int)id.x;
    int y = (int)id.y;
    if(x >= _Width) return;
    if(y >= _Height) return;

    for (int z = 0; z < _Depth; z++)
    {
        Voxel voxel = _VoxelBuffer[get_voxel_index(x, y, z)];
                    //  Almost  the  same  processing  as  in
CPUVoxelizer.Voxelize continues
        ...
    }
}
```

Once the voxel data is obtained in this way, it discards the mesh data that is no longer needed and generates GPUVoxel Data with the data needed to create the voxel visual representation.

GPUVoxelizer.cs

```
// Discard mesh data that is no longer needed
vertBuffer.Release();
triBuffer.Release();
```

```
return new GPUVoxelData(voxelBuffer, width, height, depth,
unit);
```

This completes voxelization by GPU implementation. Voxel data is actually visualized using GPU VoxelData in GPUVoxelizerTest.cs.

1.5 Speed difference between CPU implementation and GPU implementation

In the test scene, Voxelizer is executed at the time of Play, so it is difficult to understand the speed difference between CPU implementation and GPU implementation, but GPU implementation has achieved considerable speedup.

Performance depends greatly on the execution environment, the number of polygons in the mesh to be voxelized, and the resolution of voxelization.

- Execution environment OS: Windows10, CPU: Core i7, Memory: 32GB, GPU: GeForce GTX 980
- Mesh with 5319 vertices and 9761 triangles
- Voxelized resolution 256

Under these conditions, the GPU implementation is running 50 times faster than the CPU implementation.

1.6 Application example

Introducing an application example (GPUVoxelParticleSystem) using the GPU-implemented ParticleSystem.

The GPU Voxel Particle System uses the Compute Buffer, which represents the voxel data obtained from the GPU Voxelizer, to calculate the position of particles in the Compute Shader.

1. Voxelize animation model every frame with GPU Voxelizer
2. Pass the ComputeBuffer of GPUVoxelData to the ComputeShader that calculates the position of particles.

3. Draw particles with GPU instancing

I am creating an effect in the flow.

Figure 1.17: Application example using ParticleSystem of GPU implementation (GPUVoxelParticleSystem)

By making a large number of particles appear from the voxel position, a visual like an animation model composed of particles is realized.

Voxels can be applied to the animation model frame by frame only because of the speedup by implementing the GPU, and in order to expand the range of visual expressions that can be used in real time, such speedup on the GPU is indispensable. It has become a thing.

1.7 Summary

In this chapter, we introduced the algorithm for voxelizing the mesh model using CPU implementation as an example, and even speeded up voxelization by GPU implementation.

We took the approach of generating voxels using the intersection judgment of triangles and voxels, but there is also a method of constructing voxel data by rendering the model from the XYZ directions into a 3D texture by parallel projection.

The method introduced in this chapter has a problem in how to apply a texture to the voxelized model, but if the method renders the model to a 3D texture, coloring the voxels is easier and more accurate. You may be able to do it.

1.8 Reference

- http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization
- http://www.dyn4j.org/2010/01/sat/
- https://gdbooks.gitbooks.io/3dcollisions/content/Chapter4/aabb-triangle.html
- Game Engine Architecture 2nd Edition Chapter 12
- https://developer.nvidia.com/content/basics-gpu-voxelization

Chapter 2 GPU-Based Trail

2.1 Introduction

In this chapter, we will show you how to use GPU to create a trail. The sample in this chapter is "GPU Based Trail" from https://github.com/IndieVisualLab/UnityGraphicsProgramming2 .

2.1.1 What is Trail?

The trajectory of a moving object is called a trail. In a broad sense, it includes car ruts, ship tracks, ski spurs, etc., but what is impressive in CG is a light trail expression that draws a curve like a car tail lamp or a homing laser in a shooting game. ..

2.1.2 Unity standard Trail

Two types of trails are provided as standard in Unity.

- **TrailRenderer** * 1 Used to draw the trajectory of GameObject
- **Trails module** * 2 Used to draw the trajectory of Particles

[*1] https://docs.unity3d.com/ja/current/Manual/class-TrailRenderer.html

[*2] https://docs.unity3d.com/Manual/PartSysTrailsModule.html

Since this chapter focuses on how to create the Trail itself, we will not use these functions, and by implementing it on the GPU, it will be possible to express more than the Trails module.

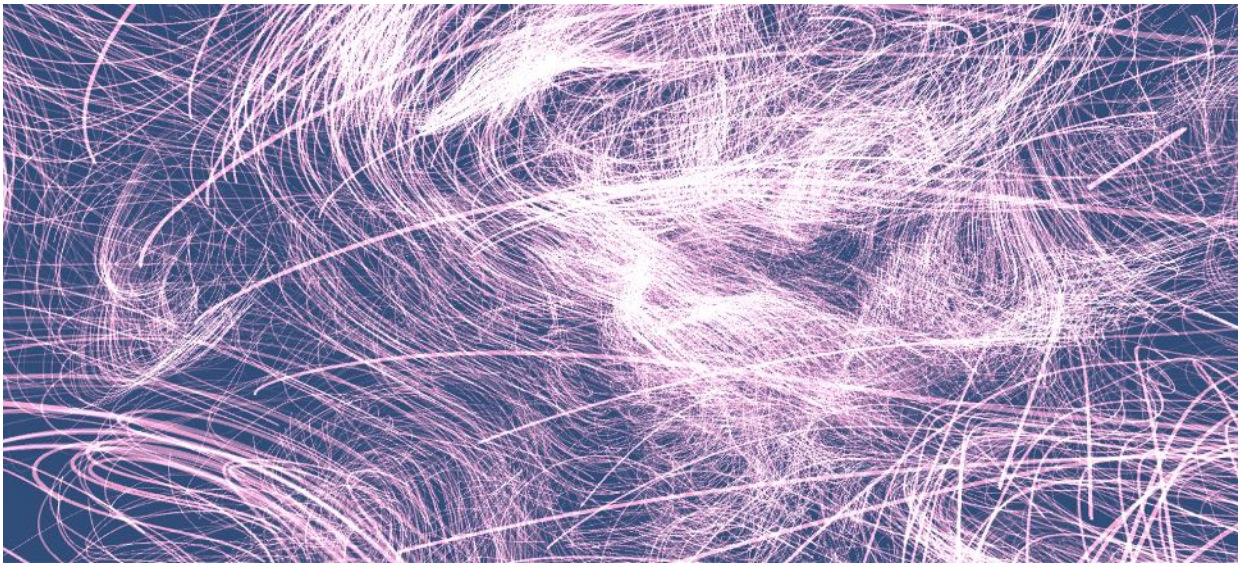

Figure 2.1: Sample code execution screen. Show 10000 Trails

2.2 Creating data

Now let's create a Trail.

2.2.1 Data definition

There are three main structures used.

GPUTrails.cs

```
public struct Trail
{
  public int currentNodeIdx;
}
```

Each Trail structure corresponds to one Trail. currentNodeIdx Stores the index of the last written Node buffer.

GPUTrails.cs

```
public struct Node
{
  public float time;
```

```
  public Vector3 pos;
}
```

Node structures are control points in the Trail. It stores the location of the Node and the time it was updated.

GPUTrails.cs

```
public struct Input
{
  public Vector3 pos;
}
```

The Input structure is the input for one frame from the emitter (the one that leaves the trajectory). Here, it's just the position, but I think it would be interesting to add colors and so on.

2.2.2 Initialization

Initialize the buffer used by GPUTrails.Start ()

GPUTrails.cs

```
trailBuffer = new ComputeBuffer(trailNum,
Marshal.SizeOf(typeof(Trail)));
nodeBuffer = new ComputeBuffer(totalNodeNum,
Marshal.SizeOf(typeof(Node)));
inputBuffer = new ComputeBuffer(trailNum,
Marshal.SizeOf(typeof(Input)));
```

Initializing trailBuffers for trailNum. In other words, this program processes multiple Trails at once. In nodeBuffer, Nodes for all Trails are handled together in one buffer. Indexes 0 to nodeNum-1 are the first, nodeNum to 2 * nodeNum-1 are the second, and so on. The inputBuffer also holds trailNums and manages the input of all trails.

GPUTrails.cs

```
var initTrail = new Trail() { currentNodeIdx = -1 };
var initNode = new Node() { time = -1 };

trailBuffer.SetData(Enumerable.Repeat(initTrail,
```

```
trailNum).ToArray());
nodeBuffer.SetData(Enumerable.Repeat(initNode,
totalNodeNum).ToArray());
```

The initial value is put in each buffer. Set Trail.currentNodeIdx and Node.time to negative numbers, and use them later to determine whether they are unused. Since all values of inputBuffer are written in the first update, there is no need to initialize and there is no touch.

2.2.3 How to use Node buffer

Here's how to use the Node buffer.

initial state

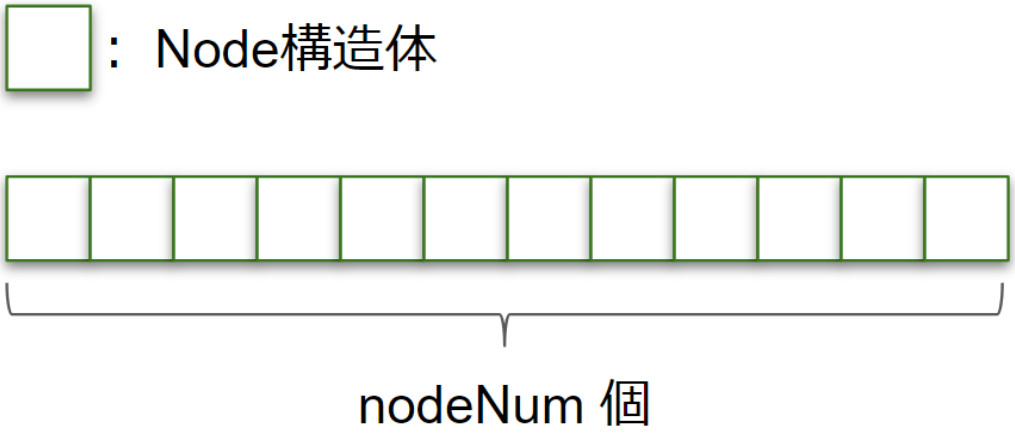

Figure 2.2: Initial state

Nothing has been entered yet.

Entering

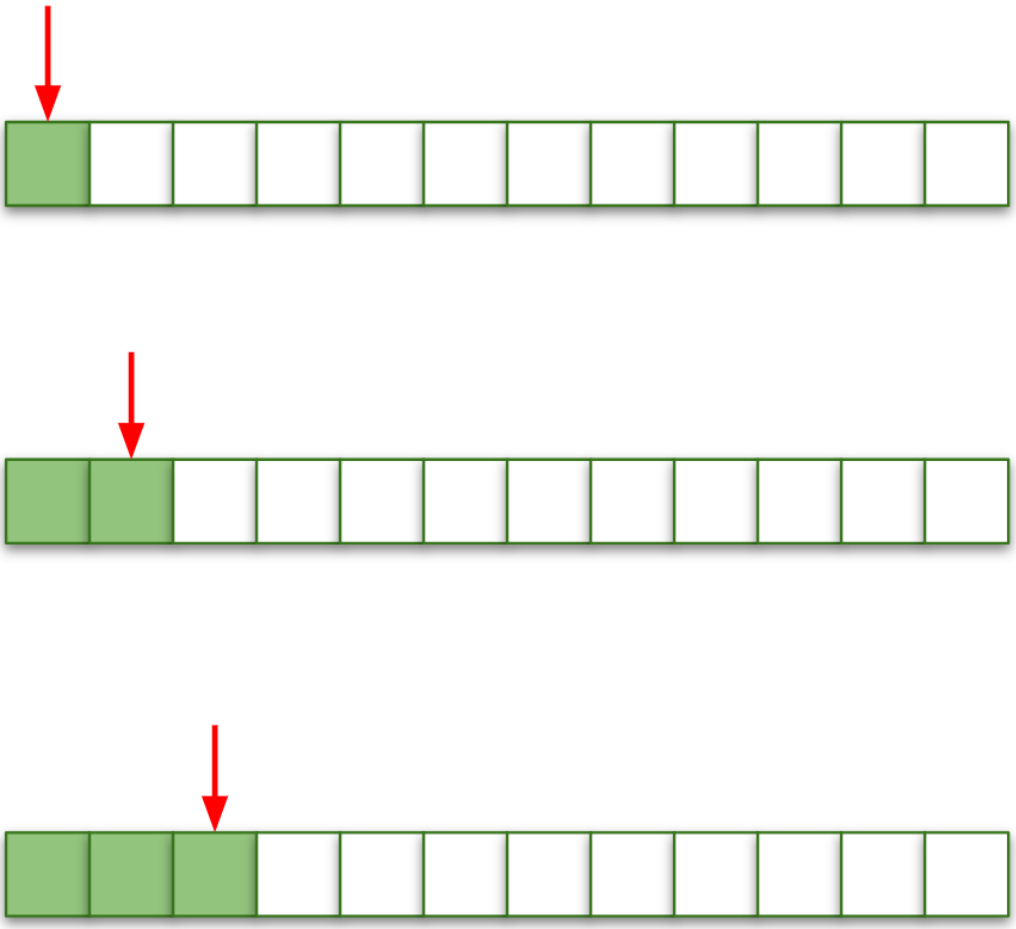

Figure 2.3: Input

It will be input one node at a time. I have an unused Node.

loop

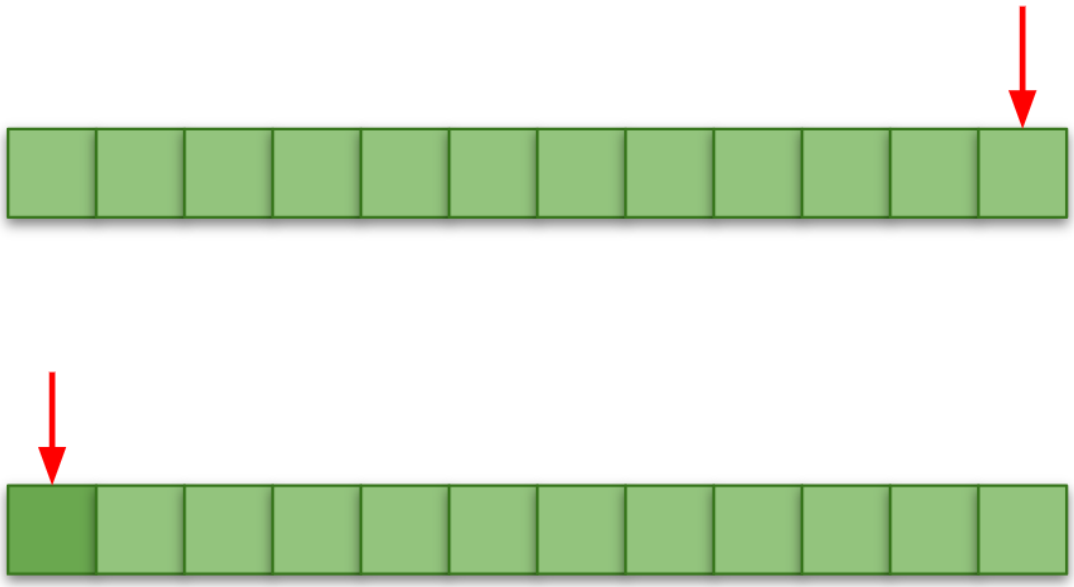

Figure 2.4: Loop

When all the Nodes are exhausted, the returning Nodes will be overwritten at the beginning. It is used like a ring buffer.

2.2.4 Input

From here, it will be called every frame. Enter the position of the emitter to add and update Nodes.

First, update the inputBuffer externally. This can be any process. At first `ComputeBuffer.SetData()`, it may be easier and better to calculate with the CPU . The sample code moves particles in a simple GPU implementation and treats them as emitters.

Curl Noise

The particles in the sample code move in search of the force received by Curl Noise. As you can see, Curl Noise is very convenient because you can easily create pseudo-fluid-like movements. Of this book - "Neuss' s Arco 's squirrel' s-time commentary for the pseudo-fluid Curl Noise" Chapter 6 in **@sakope** See all means because I have been described in detail.

Emitter update

GPUTrailParticles.cs

```
void Update()
{
 cs.SetInt(CSPARAM.PARTICLE_NUM, particleNum);
 cs.SetFloat(CSPARAM.TIME, Time.time);
 cs.SetFloat(CSPARAM.TIME_SCALE, _timeScale);
 cs.SetFloat(CSPARAM.POSITION_SCALE, _positionScale);
 cs.SetFloat(CSPARAM.NOISE_SCALE, _noiseScale);

 var kernelUpdate = cs.FindKernel(CSPARAM.UPDATE);
    cs.SetBuffer(kernelUpdate,    CSPARAM.PARTICLE_BUFFER_WRITE,
_particleBuffer);

 var updateThureadNum = new Vector3(particleNum, 1f, 1f);
 ComputeShaderUtil.Dispatch(cs, kernelUpdate, updateThureadNum);


 var kernelInput = cs.FindKernel(CSPARAM.WRITE_TO_INPUT);
     cs.SetBuffer(kernelInput,     CSPARAM.PARTICLE_BUFFER_READ,
_particleBuffer);
         cs.SetBuffer(kernelInput,         CSPARAM.INPUT_BUFFER,
trails.inputBuffer);
```

```
 var inputThreadNum = new Vector3(particleNum, 1f, 1f);
 ComputeShaderUtil.Dispatch(cs, kernelInput, inputThreadNum);
}
```

I'm running two kernels.

CSPARAM.UPDATE
: I'm updating the particles used as emitters.

CSPARAM.WRITE_TO_INPUT
: The current position of the emitter is written to inputBuffer. Use this as a Trail entry.

Input to Trail

Now, let's update nodeBuffer by referring to inputBuffer.

GPUTrailParticles.cs

```
void LateUpdate()
{
  cs.SetFloat(CSPARAM.TIME, Time.time);
  cs.SetFloat(CSPARAM.UPDATE_DISTANCE_MIN, updateDistaceMin);
  cs.SetInt(CSPARAM.TRAIL_NUM, trailNum);
  cs.SetInt(CSPARAM.NODE_NUM_PER_TRAIL, nodeNum);

  var kernel = cs.FindKernel(CSPARAM.CALC_INPUT);
  cs.SetBuffer(kernel, CSPARAM.TRAIL_BUFFER, trailBuffer);
  cs.SetBuffer(kernel, CSPARAM.NODE_BUFFER, nodeBuffer);
  cs.SetBuffer(kernel, CSPARAM.INPUT_BUFFER, inputBuffer);

   ComputeShaderUtil.Dispatch(cs, kernel, new Vector3(trailNum,
1f, 1f));
}
```

On the CPU side, all you have to do is Dispatch () ComputeShader, passing the required parameters. The processing on the main ComputeShader side is as follows.

GPUTrail.compute

```
[numthreads(256,1,1)]
void CalcInput (uint3 id : SV_DispatchThreadID)
{
```

```
  uint trailIdx = id.x;
  if ( trailIdx < _TrailNum)
  {
        Trail trail = _TrailBuffer[trailIdx];
        Input input = _InputBuffer[trailIdx];
        int currentNodeIdx = trail.currentNodeIdx;

        bool update = true;
        if ( trail.currentNodeIdx >= 0 )
        {
          Node node = GetNode(trailIdx, currentNodeIdx);
          float dist = distance(input.position, node.position);
          update = dist > _UpdateDistanceMin;
        }

        if ( update )
        {
          Node node;
          node.time = _Time;
          node.position = input.position;

          currentNodeIdx++;
          currentNodeIdx %= _NodeNumPerTrail;

          // write new node
          SetNode(node, trailIdx, currentNodeIdx);

          // update trail
          trail.currentNodeIdx = currentNodeIdx;
          _TrailBuffer[trailIdx] = trail;
        }
  }
}
```

Let's take a closer look.

```
uint trailIdx = id.x;
if ( trailIdx < _TrailNum)
```

First, I'm using the argument id as the Trail index. Due to the number of threads, it may be called with ids equal to or greater than the number of Trails, so I play something outside the range with an if statement.

```
int currentNodeIdx = trail.currentNodeIdx;

bool update = true;
```

```
if ( trail.currentNodeIdx >= 0 )
{
    Node node = GetNode(trailIdx, currentNodeIdx);
        update = distance(input.position, node.position) >
_UpdateDistanceMin;
}
```

`Trail.currentNodeIdx`I am checking next . If it is negative, it is an unused Trail.

`GetNode()` Is a function that gets the specified Node from _NodeBuffer. Since the index calculation is the source of mistakes, it is functionalized.

The Trail, which is already in use, compares the distance between the latest Node and the input position and states that `_UpdateDistanceMin`it will be updated if it is farther away and will not be updated if it is closer. Although it depends on the behavior of the emitter, the input at almost the same position as the previous Node is usually in a state of being almost stopped and moving with a slight error, so if you try to generate a Trail by converting these into Nodes in a lawful manner, between consecutive Nodes The direction is very different and it is often quite dirty. Therefore, at a very short distance, I dare to skip without adding Node.

GPUTrail.compute

```
if ( update )
{
  Node node;
  node.time = _Time;
  node.position = input.position;

  currentNodeIdx++;
  currentNodeIdx %= _NodeNumPerTrail;

  // write new node
  SetNode(node, trailIdx, currentNodeIdx);

  // update trail
  trail.currentNodeIdx = currentNodeIdx;
  _TrailBuffer[trailIdx] = trail;
}
```

Finally, I'm updating _NodeBuffer and _TrailBuffer. The Trail stores the index of the entered Node as currentNodeIdx. When the number of Nodes per Trail is exceeded, it is returned to zero so that it becomes a ring buffer. Node stores the time and position of the input.

Well, this completes the logical processing of Trail. Next, let's look at the process of drawing from this information.

2.3 Drawing

Drawing a Trail is basically a process of connecting Nodes with a line. Here, I will try to keep the individual trails as simple as possible and focus on quantity. Therefore, we want to reduce the number of polygons as much as possible, so we will generate the line as a plate polygon facing the camera.

2.3.1 Generation of plate polygons facing the camera

The method to generate the plate polygon facing the camera is as follows.

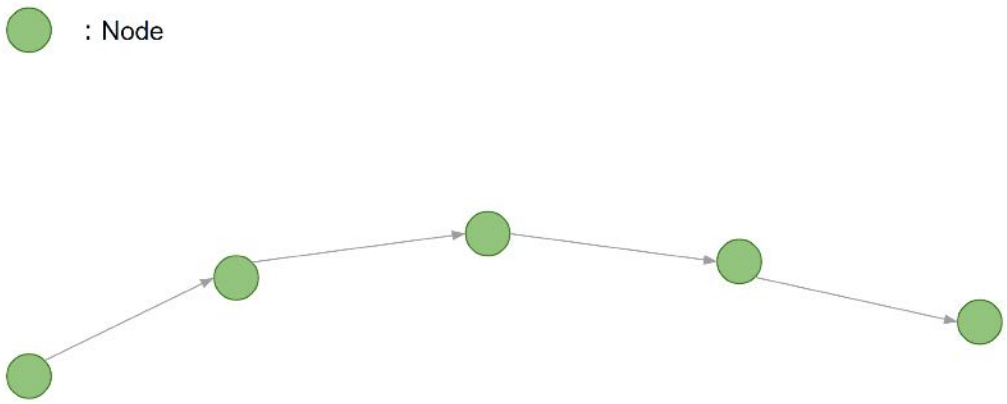

Figure 2.5: Node column

From a Node column like this

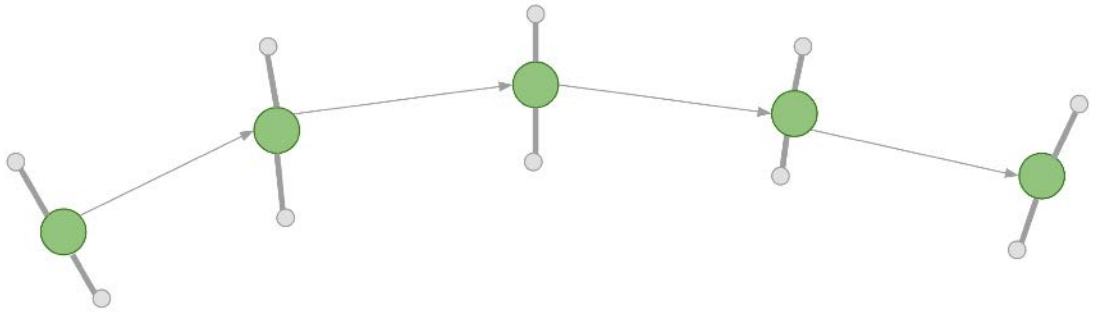

Figure 2.6: Vertices generated from Node

Finds the vertices that are moved from each node by the specified width in the direction perpendicular to the line of sight.

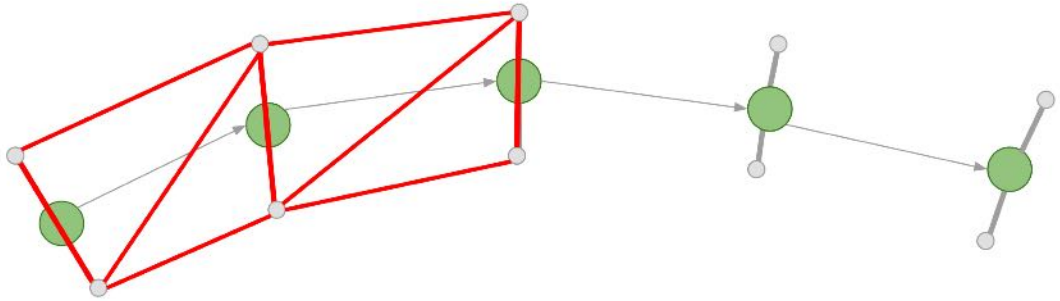

Figure 2.7: Polygonization

Connect the generated vertices to make a polygon. Let's take a look at the actual code.

2.3.2 CPU side

On the CPU side, the process is simply to pass the parameters to the material and perform DrawProcedual ().

GPUTrailRenderer.cs

```
void OnRenderObject ()
{
          _material.SetInt(GPUTrails.CSPARAM.NODE_NUM_PER_TRAIL,
trails.nodeNum);
  _material.SetFloat(GPUTrails.CSPARAM.LIFE, trails._life);
             _material.SetBuffer(GPUTrails.CSPARAM.TRAIL_BUFFER,
trails.trailBuffer);
              _material.SetBuffer(GPUTrails.CSPARAM.NODE_BUFFER,
trails.nodeBuffer);
  _material.SetPass(0);

  var nodeNum = trails.nodeNum;
  var trailNum = trails.trailNum;
        Graphics.DrawProcedural(MeshTopology.Points,    nodeNum,
trailNum);
}
```

Parameters `trails._life`that have not appeared until now have appeared. This is used for processing that compares the lifetime of the Node with the generation time that the Node itself has, and makes it transparent after this amount of time. By doing this, you can express that the end of the trail disappears smoothly.

Since there are no meshes or polygons to input, `Graphics.DrawProcedural()`we issue a command to draw a model with trails.nodeNum vertices in batches of trails.trailNum instances.

2.3.3 GPU side

vertex shader

GPUTrails.shader

```
vs_out vert (uint id : SV_VertexID, uint instanceId :
SV_InstanceID)
{
  vs_out Out;
```

```
  Trail trail = _TrailBuffer[instanceId];
  int currentNodeIdx = trail.currentNodeIdx;

  Node node0 = GetNode(instanceId, id-1);
  Node node1 = GetNode(instanceId, id); // current
  Node node2 = GetNode(instanceId, id+1);
  Node node3 = GetNode(instanceId, id+2);

  bool isLastNode = (currentNodeIdx == (int)id);

  if ( isLastNode || !IsValid(node1))
  {
          node0 = node1 = node2 = node3 = GetNode(instanceId,
currentNodeIdx);
  }

  float3 pos1 = node1.position;
  float3 pos0 = IsValid(node0) ? node0.position : pos1;
  float3 pos2 = IsValid(node2) ? node2.position : pos1;
  float3 pos3 = IsValid(node3) ? node3.position : pos2;

  Out.pos = float4(pos1, 1);
  Out.posNext = float4(pos2, 1);

  Out.dir = normalize(pos2 - pos0);
  Out.dirNext = normalize(pos3 - pos1);

  float ageRate = saturate((_Time.y - node1.time) / _Life);
  float ageRateNext = saturate((_Time.y - node2.time) / _Life);
  Out.col = lerp(_StartColor, _EndColor, ageRate);
  Out.colNext = lerp(_StartColor, _EndColor, ageRateNext);

  return Out;
}
```

First is the processing of vertex shader. Outputs information about the current Node and the next Node corresponding to this thread.

GPUTrails.shader

```
Node node0 = GetNode(instanceId, id-1);
Node node1 = GetNode(instanceId, id); // current
Node node2 = GetNode(instanceId, id+1);
Node node3 = GetNode(instanceId, id+2);
```

The current node is set to node1, and a total of four nodes are referenced, including the previous node0, the previous node2, and the second node3.

GPUTrails.shader

```
bool isLastNode = (currentNodeIdx == (int)id);

if ( isLastNode || !IsValid(node1))
{
    node0 = node1 = node2 = node3 = GetNode(instanceId,
currentNodeIdx);
}
```

If the current Node is terminal or has not yet been entered, treat nodes 0-3 as a copy of the terminal Node. In other words, all Nodes beyond the end that have no information yet are treated as "folded" to the end. By doing this, it can be sent as it is to the subsequent polygon generation processing.

GPUTrails.shader

```
float3 pos1 = node1.position;
float3 pos0 = IsValid(node0) ? node0.position : pos1;
float3 pos2 = IsValid(node2) ? node2.position : pos1;
float3 pos3 = IsValid(node3) ? node3.position : pos2;

Out.pos = float4(pos1, 1);
Out.posNext = float4(pos2, 1);
```

Now, extract the location information from the four Nodes. Please note that all but the current Node (node1) may be blank. It may be a little surprising that node0 is not entered, but this is possible because node0 points to the last node in the buffer going back in the ring buffer when currentNodeIdx == 0. Again, copy the location of node1 to fold it to the same location. The same applies to nodes2 and 3. Of these, pos1 and pos2 are output toward the geometry shader.

GPUTrails.shader

```
Out.dir = normalize(pos2 - pos0);
Out.dirNext = normalize(pos3 - pos1);
```

Furthermore, the direction vector of pos0 → pos2 is output as the tangent at pos1, and the direction vector of pos1 → pos3 is output as the tangent at pos2.

GPUTrails.shader

```
float ageRate = saturate((_Time.y - node1.time) / _Life);
float ageRateNext = saturate((_Time.y - node2.time) / _Life);
Out.col = lerp(_StartColor, _EndColor, ageRate);
Out.colNext = lerp(_StartColor, _EndColor, ageRateNext);
```

Finally, the color is calculated by comparing the write time of node1 and node2 with the current time.

geometry shader

GPUTrails.shader

```
[maxvertexcount(4)]
void geom (point vs_out input[1], inout TriangleStream<gs_out>
outStream)
{
  gs_out output0, output1, output2, output3;
  float3 pos = input[0].pos;
  float3 dir = input [0] .dir;
  float3 posNext = input[0].posNext;
  float3 dirNext = input[0].dirNext;

  float3 camPos = _WorldSpaceCameraPos;
  float3 toCamDir = normalize(camPos - pos);
  float3 sideDir = normalize(cross(toCamDir, dir));

  float3 toCamDirNext = normalize(camPos - posNext);
  float3 sideDirNext = normalize(cross(toCamDirNext, dirNext));
  float width = _Width * 0.5;

  output0.pos = UnityWorldToClipPos(pos + (sideDir * width));
  output1.pos = UnityWorldToClipPos(pos - (sideDir * width));
   output2.pos = UnityWorldToClipPos(posNext + (sideDirNext *
width));
   output3.pos = UnityWorldToClipPos(posNext - (sideDirNext *
width));

  output0.col =
```

```
  output1.col = input[0].col;
  output2.col =
  output3.col = input[0].colNext;

  outStream.Append (output0);
  outStream.Append (output1);
  outStream.Append (output2);
  outStream.Append (output3);

  outStream.RestartStrip();
}
```

Next is the processing of geometry shader. The polygon is finally generated from the information for two Nodes passed from the vertex shader. From 2 pos and dir, 4 positions = quadrangle are obtained and output as TriangleStream.

GPUTrails.shader

```
float3 camPos = _WorldSpaceCameraPos;
float3 toCamDir = normalize(camPos - pos);
float3 sideDir = normalize(cross(toCamDir, dir));
```

The outer product of the direction vector (toCameraDir) from pos to the camera and the tangent vector (dir) is obtained, and this is set as the width of the line (sideDir).

GPUTrails.shader

```
output0.pos = UnityWorldToClipPos(pos + (sideDir * width));
output1.pos = UnityWorldToClipPos(pos - (sideDir * width));
```

Find the vertices that have moved in the positive and negative sideDir directions. Here, we have completed the coordinate transformation to make it a Clip coordinate system and pass it to the fragment shader. By performing the same processing for posNext, a total of four vertices were obtained.

GPUTrails.shader

```
output0.col =
output1.col = input[0].col;
output2.col =
output3.col = input[0].colNext;
```

Add color to each vertex to complete.

fragment shader

GPUTrails.shader

```
fixed4 frag (gs_out In) : COLOR
{
  return In.col;
}
```

Finally, the fragment shader. It's as simple as it gets. It just outputs the color (laughs)

2.4 Application

I think that the Trail has been generated. This time, the processing was only for colors, but I think that it can be applied in various ways, such as adding textures and changing the width. Also, as the source code is separated from GPUTrails.cs and GPUTRailsRenderer.cs, the GPUTrails.shader side is just a process of drawing by looking at the buffer, so if you prepare _TrailBuffer and _NodeBuffer, it is not limited to Trail but actually line-shaped. It can be used for display. This time it was just a trail added to _NodeBuffer, but I think that by updating all Nodes every frame, it is possible to express something like a tentacle.

2.5 Summary

This chapter has provided the simplest possible example of Trail's GPU implementation. While debugging becomes difficult with the GPU, it enables overwhelming physical expression that cannot be done with the CPU. I hope that as many people as possible can experience that "Uhyo!" Feeling through this book. Also, I think Trail is an expression of an interesting area with a wide range of applications, such as "displaying a model" and "drawing with an algorithm in screen space". I think that the understanding gained in this process will be useful when programming various video expressions, not limited to Trail.

Chapter 3 Application of Geometry Shader for Line Representation

3.1 Introduction

This year, I participated in a hackathon called Art Hack Day 2018 * 1 where I personally created a visual work using Unity.

Figure 3.1: Visual part of Already There

In my work, I used the technique of drawing a wireframe polygon using the Geometry Shader. In this chapter, we will explain the method. The sample in this chapter is "Geometry Wireframe" from https://github.com/IndieVisualLab/UnityGraphicsProgramming2 .

[*1] Art Hack Day 2018 http://arthackday.jp/

3.2 Try to draw a line for the time being

I think that LineRenderer and GL are often used to draw lines in Unity, but this time I will use Graphics.DrawProcedural assuming that the amount of drawing will increase later.
First of all, let's draw a simple sine wave. Take a **look at the** sample **SampleWaveLine scene** .

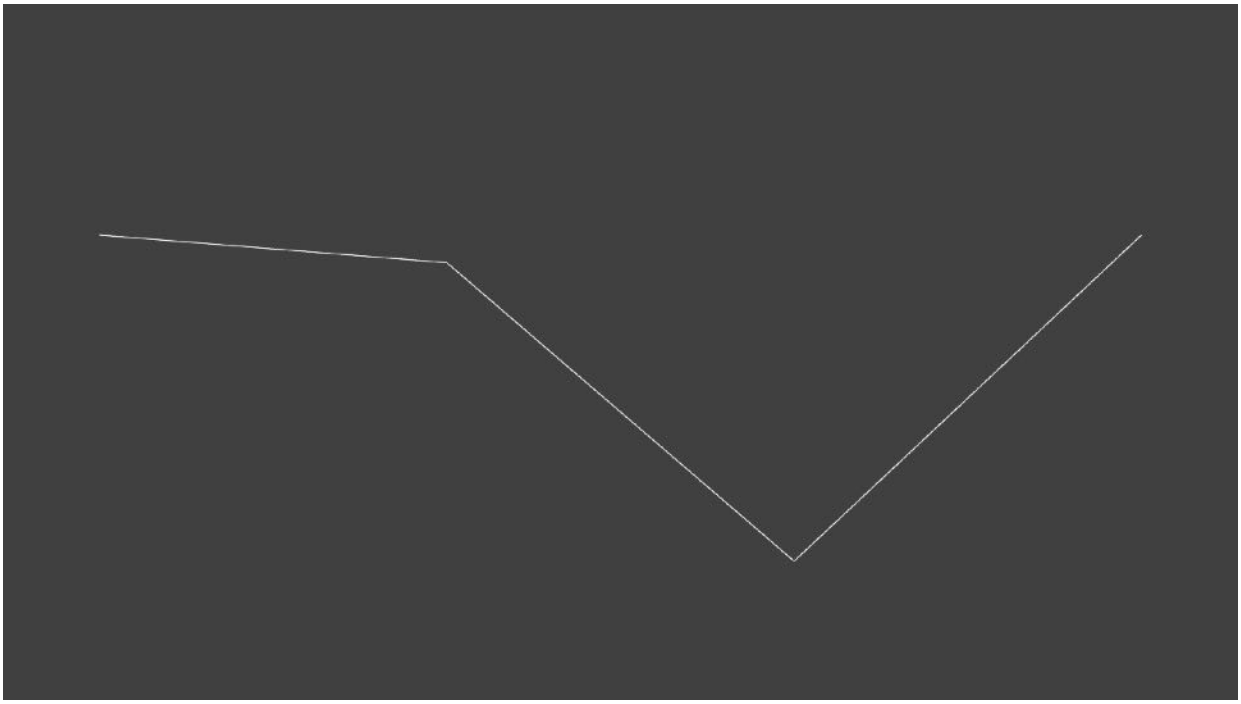

Figure 3.2: SampleWaveLine scene

For now, press the play button and run it, and you should see an orange sine wave in the Game view. Select the WabeLine object in the Hierarchy window and move the Vertex Num slider on the RenderWaveLine component in the Inspector window to change the smoothness of the sine wave. The implementation of the RenderWaveLine class looks like this:

Listing 3.1: RenderWaveLine.cs

```
using UnityEngine;

[ExecuteInEditMode]
public class RenderWaveLine : MonoBehaviour {
    [Range(2,50)]
    public int vertexNum = 4;

    public Material material;

    private void OnRenderObject ()
    {
        material.SetInt("_VertexNum", vertexNum - 1);
        material.SetPass(0);
                Graphics.DrawProcedural(MeshTopology.LineStrip,
vertexNum);
```

```
    }
}
```

Graphics.DrawProcedural runs immediately after the call, so it must be called inside the OnRenderObject. OnRenderObject is called after all cameras have rendered the scene. The first argument of Graphics.DrawProcedural is **MeshTopology** . MeshTopology is a specification of how to configure the mesh. There are six configurations that can be specified: Triangles (triangle polygon), Quads (square polygon), Lines (line connecting two points), LineStrip (connecting all points continuously), and Points (independent points). The second argument is the **number** of **vertices** .
This time, I want to place the vertices on the line of the **sine** wave and connect the lines, so I use **MeshTopology.LineStrip** . The second argument, vertexNum, specifies the number of vertices used to draw the sine wave. As you may have noticed here, I haven't passed an array of vertex coordinates to Shader anywhere. The vertex coordinates are calculated in the following Shader Vertex Shader (vertex shader). Next is WaveLine.shader.

Listing 3.2: WaveLine.shader

```
Shader "Custom/WaveLine"
{
  Properties
  {
    _Color ("Color", Color) = (1,1,1,1)
    _ScaleX ("Scale X", Float) = 1
    _ScaleY ("Scale Y", Float) = 1
    _Speed ("Speed",Float) = 1
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #pragma target 3.5

      #include "UnityCG.cginc"
```

```
      #define PI  3.14159265359

      struct v2f
      {
        float4 vertex : SV_POSITION;
      };

      float4 _Color;
      int _VertexNum;
      float _ScaleX;
      float _ScaleY;
      float _Speed;

      v2f vert (uint id : SV_VertexID)
      {
        float div = (float)id / _VertexNum;
        float4 pos = float4((div - 0.5) * _ScaleX,
          sin(div * 2 * PI + _Time.y * _Speed) * _ScaleY, 0, 1);

        v2f o;
        o.vertex = UnityObjectToClipPos(pos);
        return o;
      }

      fixed4 frag (v2f i) : SV_Target
      {
        return _Color;
      }
      ENDCG
    }
  }
}
```

SV_VertexID (vertex ID) is passed to the argument of the Vertex Shader function vert. The vertex ID is a serial number unique to the vertex. Feeling that if you pass the number of vertices to be used as the second argument of Graphics.DrawProcedural, Vertex Shader will be called for the number of vertices, and the vertex ID of the argument will be a value from 0 to -1. is.
In Vertex Shader, the ratio from 0 to 1 is calculated by dividing the vertex ID by the number of vertices. The vertex coordinates (pos) are calculated based on the calculated ratio. The coordinates on the sine wave are obtained by giving the ratio obtained earlier in the calculation of the Y coordinate to the sin function. By adding _Time.y, we also animate the change in height as time progresses. Since the vertex coordinates are calculated in Vertex

Shader, there is no need to pass the vertex coordinates from the C # side. Then, UnityObjectToClipPos is passing the coordinates converted from the object space to the clip space of the camera to the Fragment Shader.

3.3 Dynamically draw a two-dimensional polygon with Geometry Shader

3.3.1 Increase vertices with Geometry Shader

Next, let's draw a polygon. To draw a polygon, you need vertices for each corner. It can be done by connecting vertices and closing as in the previous section, but this time I will draw a polygon from one vertex using Geometry Shader. For details on Geometry Shader, refer to "Chapter 6 Growing Grass with Geometry Shader" in UnityGraphicsProgramming vol.1 * 2 . Roughly speaking, the Geometry Shader is a shader that can increase the number of vertices, located between the Vertex Shader and the Fragment Shader.

Take a **look at the** sample **SamplePolygonLine scene** .

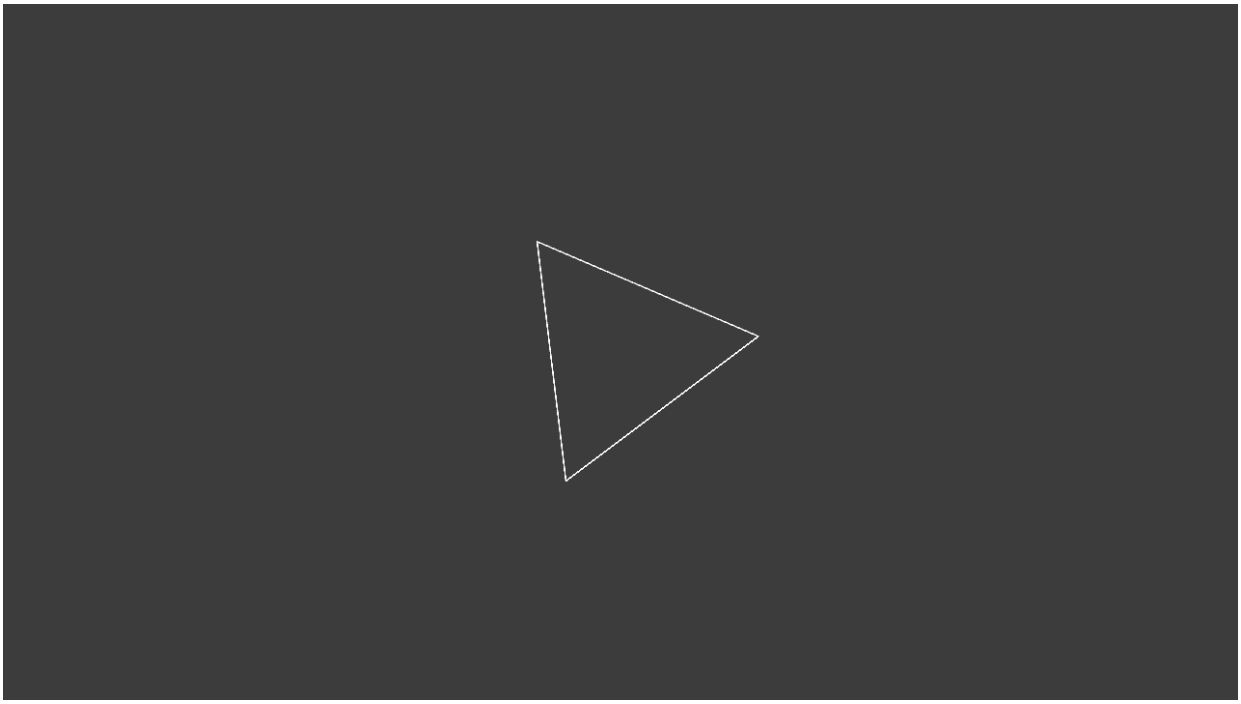

Figure 3.3: SamplePolygonLine scene

When you press the play button and run it, the triangle should rotate in the Game view. You can increase or decrease the number of triangle angles by selecting the PolygonLine object in the Hierarchy window and moving the Vertex Num slider on the SinglePolygon2D component in the Inspector window. The implementation of the SimglePolygon2D class looks like this:

Listing 3.3: SinglePolygon2D.cs

```
  using UnityEngine;

[ExecuteInEditMode]
public class SinglePolygon2D : MonoBehaviour {

    [Range(2, 64)]
    public int vertexNum = 3;

    public Material material;

    private void OnRenderObject ()
    {
        material.SetInt("_VertexNum", vertexNum);
                                        material.SetMatrix("_TRS",
transform.localToWorldMatrix);
        material.SetPass(0);
        Graphics.DrawProcedural(MeshTopology.Points, 1);
    }
}
```

It has almost the same implementation as the RenderWaveLine class.
There are two major differences. The first is that the first argument of Graphics.DrawProcedural is **changed** from **MeshTopology.LineStrip** to **MeshTopology.Points** . The other is that the second argument of Graphics.DrawProcedural is fixed at **1** . In the RenderWaveLine class in the previous section, **MeshTopology.LineStrip** was specified because the lines were drawn by connecting the vertices, but this time I want to pass only one vertex and draw a polygon, so **MeshTopology.Points**Is specified. This is because the minimum number of vertices required for drawing changes depending on the MeshTopology specification, and if it is less than that, nothing is drawn. MeshTopology.Lines and MeshTopology.LineStrip are 2 because they are lines, MeshTopology.Triangles are 3 because they are triangles, and MeshTopology.Points are 1 because they are points. By the way, in the part of material.SetMatrix ("_TRS",

transform.localToWorldMatrix) ;, the matrix converted from the local coordinate system of the GameObject to which the SinglePolygon2D component is assigned to the world coordinate system is passed to the shader. By multiplying this by the vertex coordinates in the shader, the transform of the GameObject, that is, the coordinates (position), orientation (rotation), and size (scale) will be reflected in the drawn figure.

Next, let's take a look at the implementation of SinglePolygonLine.Shader.

Listing 3.4: SinglePolygonLine.shader

```
Shader "Custom/Single Polygon Line"
{
  Properties
  {
    _Color ("Color", Color) = (1,1,1,1)
    _Scale ("Scale", Float) = 1
    _Speed ("Speed",Float) = 1
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma geometry geom // Declaration of Geometry Shader
      #pragma fragment frag
      #pragma target 4.0

      #include "UnityCG.cginc"

      #define PI  3.14159265359

      // Output structure
      struct Output
      {
        float4 pos : SV_POSITION;
      };

      float4 _Color;
      int _VertexNum;
      float _Scale;
```

```
      float _Speed;
      float4x4 _TRS;

      Output vert (uint id : SV_VertexID)
      {
        Output o;
        o.pos = mul (_TRS, float4 (0, 0, 0, 1));
        return o;
      }

      // Geometry shader
      [maxvertexcount(65)]
       void geom(point Output input[1], inout LineStream<Output>
outStream)
      {
        Output o;
        float rad = 2.0 * PI / (float)_VertexNum;
        float time = _Time.y * _Speed;

        float4 pos;

        for (int i = 0; i <= _VertexNum; i++) {
          pos.x = cos(i * rad + time) * _Scale;
          pos.y = sin (i * rad + time) * _Scale;
          pos.z = 0;
          pos.w = 1;
          o.pos = UnityObjectToClipPos (pos);

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

      fixed4 frag (Output i) : SV_Target
      {
        return _Color;
      }
      ENDCG
    }
  }
}
```

A new **#pragma geometry geom** declaration has been added between the **#pragma vertex vert** and the **#pragma fragment frag** . This means declaring a Geometry Shader function named geom. Vertex Shader's vert sets the coordinates of the vertices to the origin (0,0,0,1) for the time being, and multiplies it by the _TRS matrix (the matrix that converts from the local

coordinate system to the world coordinate system) passed from C #. It has become like. The coordinates of each vertex of the polygon are calculated in the following Geometry Shader.

Definition of Geometry Shader

```
  // Geometry shader
  [maxvertexcount(65)]
   void geom(point Output input[1], inout LineStream<Output>
outStream)
```

maxvertexcount

The maximum number of vertices output from the Geometry Shader. This time, VertexNum of the SinglePolygonLine class is used to increase the number to 64 vertices, but since a line connecting the 64th vertex to the 0th vertex is required, 65 is specified.

point Output input[1]

Represents the input information from Vertex Shader. point is a primitive type and means that one vertex is received, Output is a structure name, and input [1] is an array of length 1. Since only one vertex is used this time, I specified point and input [1], but when I want to mess with the vertices of a triangular polygon such as a mesh, I use triangle and input [3].

inout LineStream<Output> outStream

Represents the output information from the Geometry Shader. LineStream <Output> means to output the line of the Output structure. There are also PointStream and TriangleStream. Next is the explanation inside the function.

Implementation in function

```
Output o;
float rad = 2.0 * PI / (float)_VertexNum;
float time = _Time.y * _Speed;

float4 pos;
```

```
for (int i = 0; i <= _VertexNum; i++) {
  pos.x = cos(i * rad + time) * _Scale;
  pos.y = sin (i * rad + time) * _Scale;
  pos.z = 0;
  pos.w = 1;
  o.pos = UnityObjectToClipPos (pos);

  outStream.Append(o);
}

outStream.RestartStrip();
```

In order to calculate the coordinates of each vertex of the polygon, 2π (360 degrees) is divided by the number of vertices to obtain the angle of one corner. The vertex coordinates are calculated using trigonometric functions (sin, cos) in the loop. Output the calculated coordinates as vertices with outStream.Append (o). After looping as many times as _VertexNum to output the vertices, outStream.RestartStrip () ends the current strip and starts the next strip. As long as you add it with Append (), the lines will be connected as LineStream. Execute RestartStrip () to end the current line once. The next time Append () is called, it will not connect to the previous line and a new line will start.

[* 2] UnityGraphicsProgramming vol.1 https://indievisuallab.stores.jp/items/59edf11ac8f22c0152002588

3.4 Try to make Octahedron Sphere

3.4.1 Octahedron Sphereとは?

A regular octahedron is a polyhedron composed of eight equilateral triangles , as shown in Fig. 3.4 . Octahedron Sphere is a sphere created by dividing the three vertices of an equilateral triangle that make up a regular octahedron by spherical linear interpolation * 3 . Whereas normal linear interpolation interpolates so that two points are connected by a straight line, spherical linear interpolation interpolates so that two points pass on a spherical surface as shown in Fig. 3.5 .

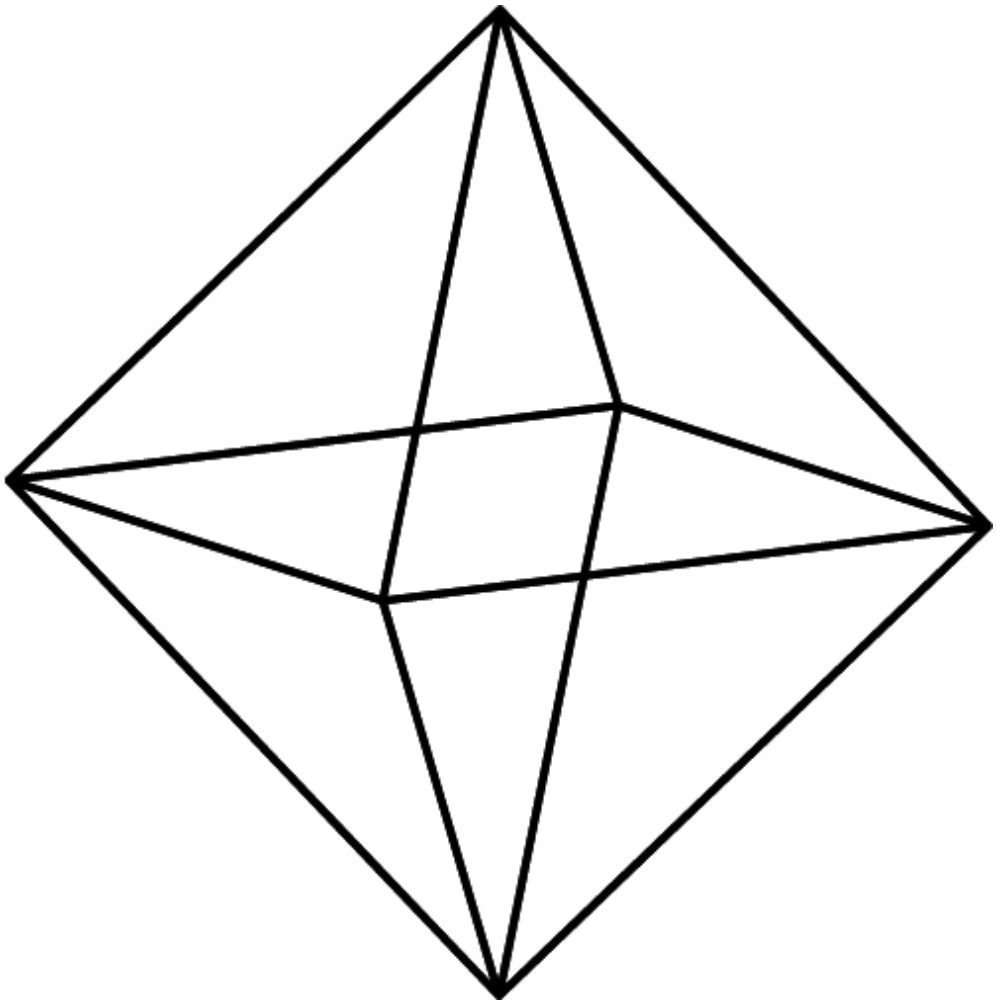

Figure 3.4: Octahedron

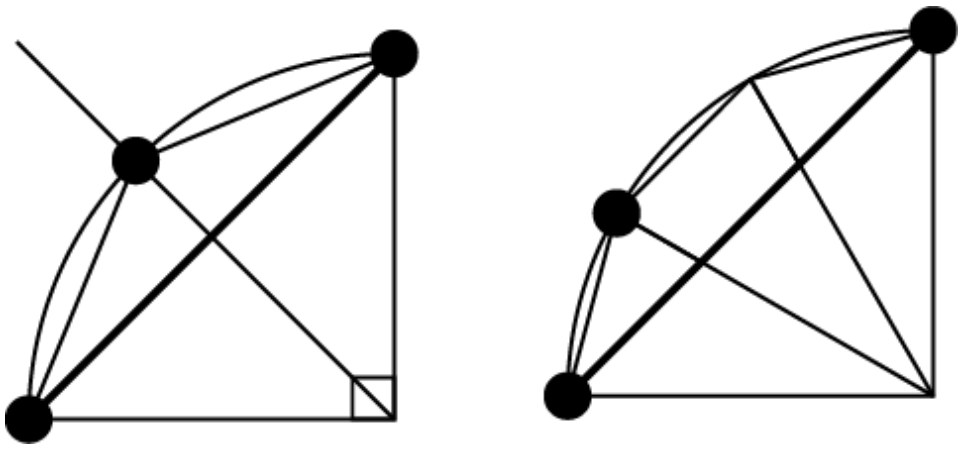

Figure 3.5: Octahedron

Take a **look at the** Sample **Octahedron** Sample **scene** .

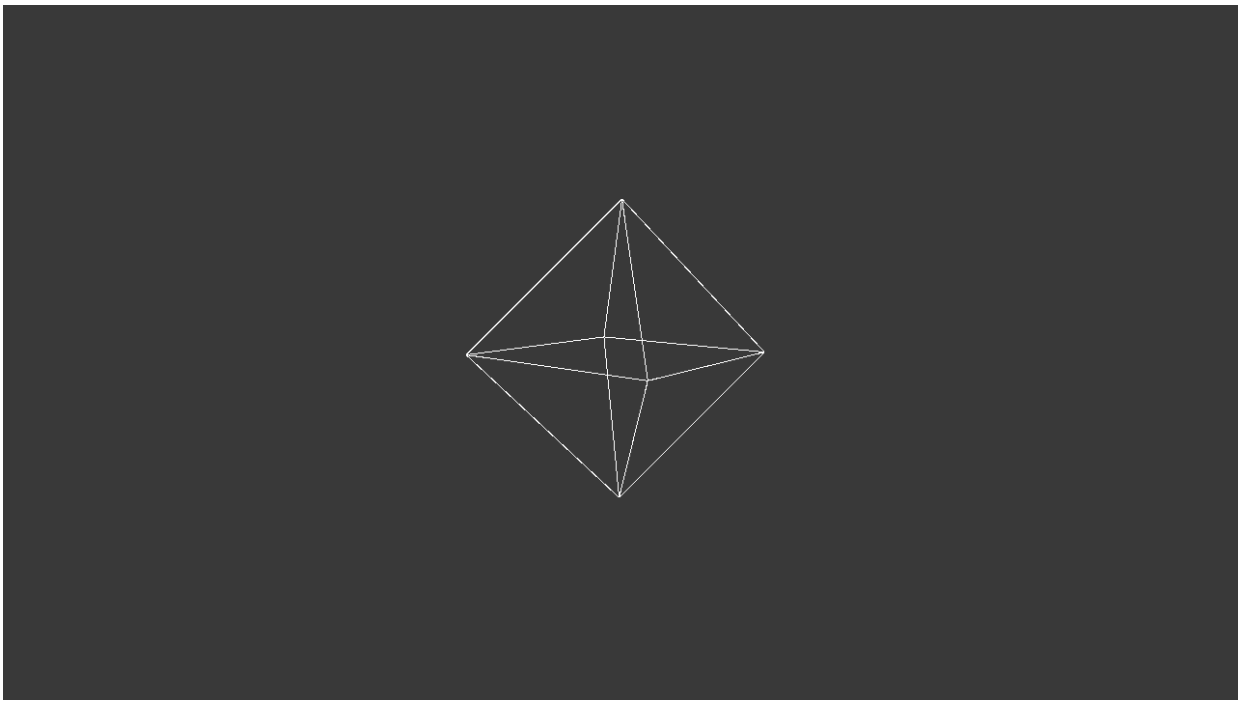

Figure 3.6: SampleWaveLine scene

When you press the run button, you should see a slowly rotating octahedron in the center of the Game view. Also, if you change the Level slider of the Geometry Octahedron Sphere component of the Single Octahedron Sphere object in the Hierarchy window, the sides of the octahedron will be split and gradually approach the sphere.

[* 3] spherical linear interpolation, slerp for short

3.4.2 Dividing the octahedron in the Geometry Shader

Next, let's take a look at the implementation. The implementation on the C # side is almost the same as SinplePolygon2D.cs in the previous section, so it will be omitted. OctahedronSphere.shader has a long source, so I will explain only in Geometry Shader.

Listing 3.5: The beginning of the Gometry Shader in OctahedronSphere.shader

```
// Geometry shader
  float4 init_vectors[24];
  // 0 : the triangle vertical to (1,1,1)
  init_vectors[0] = float4(0, 1, 0, 0);
  init_vectors[1] = float4(0, 0, 1, 0);
  init_vectors[2] = float4(1, 0, 0, 0);
  // 1 : to (1,-1,1)
  init_vectors[3] = float4(0, -1, 0, 0);
  init_vectors[4] = float4(1, 0, 0, 0);
  init_vectors[5] = float4(0, 0, 1, 0);
  // 2 : to (-1,1,1)
  init_vectors[6] = float4(0, 1, 0, 0);
  init_vectors[7] = float4(-1, 0, 0, 0);
  init_vectors[8] = float4(0, 0, 1, 0);
  // 3 : to (-1,-1,1)
  init_vectors[9] = float4(0, -1, 0, 0);
  init_vectors[10] = float4(0, 0, 1, 0);
  init_vectors[11] = float4(-1, 0, 0, 0);
  // 4 : to (1,1,-1)
  init_vectors[12] = float4(0, 1, 0, 0);
  init_vectors[13] = float4(1, 0, 0, 0);
  init_vectors[14] = float4(0, 0, -1, 0);
  // 5 : to (-1,1,-1)
  init_vectors[15] = float4(0, 1, 0, 0);
  init_vectors[16] = float4(0, 0, -1, 0);
  init_vectors[17] = float4(-1, 0, 0, 0);
  // 6 : to (-1,-1,-1)
  init_vectors[18] = float4(0, -1, 0, 0);
  init_vectors[19] = float4(-1, 0, 0, 0);
  init_vectors[20] = float4(0, 0, -1, 0);
  // 7 : to (1,-1,-1)
  init_vectors[21] = float4(0, -1, 0, 0);
  init_vectors[22] = float4(0, 0, -1, 0);
  init_vectors[23] = float4(1, 0, 0, 0);
```

First, as shown in Fig. 3.7 , we define a "normalized" octahedron triangular system that is the initial value.

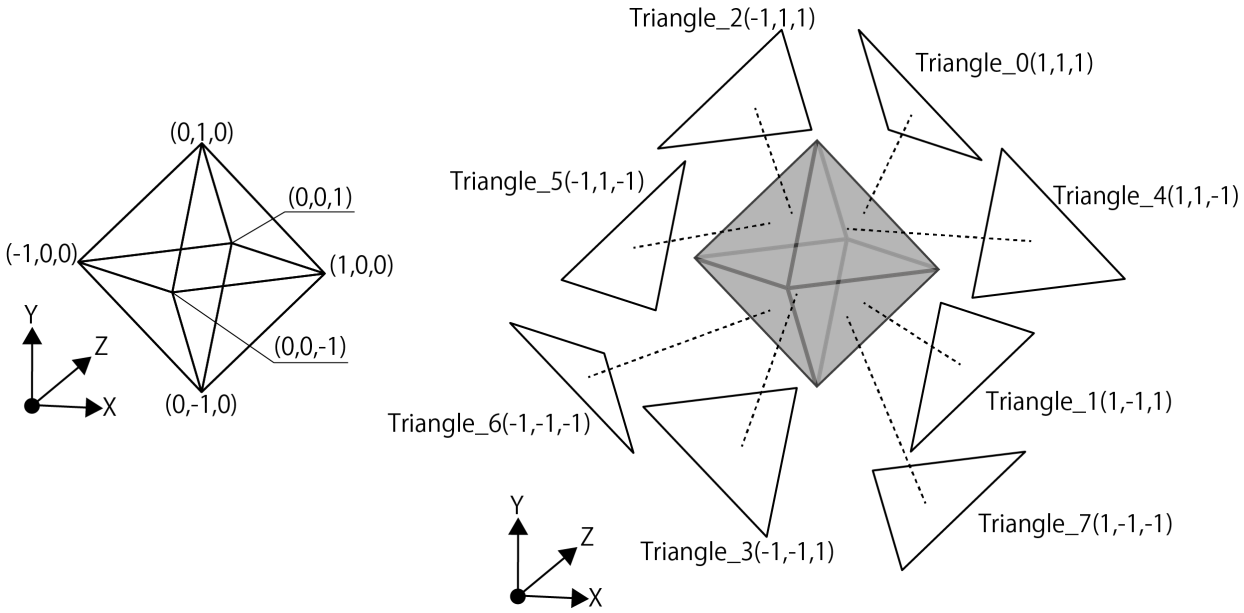

Figure 3.7: Octahedron vertex coordinates and triangles

It is defined in float4 because it is defined as a quaternion.

Listing 3.6: OctahedronSphere.shader Triangle Spherical Linear Interpolation Split Processing Part

```
for (int i = 0; i < 24; i += 3)
{
  for (int p = 0; p < n; p++)
  {
    // edge index 1
    float4 edge_p1 = qslerp(init_vectors[i],
      init_vectors[i + 2], (float)p / n);
    float4 edge_p2 = qslerp(init_vectors[i + 1],
      init_vectors[i + 2], (float)p / n);
    float4 edge_p3 = qslerp(init_vectors[i],
      init_vectors[i + 2], (float)(p + 1) / n);
    float4 edge_p4 = qslerp(init_vectors[i + 1],
      init_vectors[i + 2], (float)(p + 1) / n);

    for (int q = 0; q < (n - p); q++)
    {
      // edge index 2
      float4 a = qslerp(edge_p1, edge_p2, (float)q / (n - p));
       float4 b = qslerp(edge_p1, edge_p2, (float)(q + 1) / (n -
p));
      float4 c, d;
```

```
      if(distance(edge_p3, edge_p4) < 0.00001)
      {
        c = edge_p3;
        d = edge_p3;
      }
      else {
        c = qslerp(edge_p3, edge_p4, (float)q / (n - p - 1));
         d = qslerp(edge_p3, edge_p4, (float)(q + 1) / (n - p -
1));
      }

            output1.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, a));
            output2.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, b));
            output3.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, c));

      outStream.Append(output1);
      outStream.Append(output2);
      outStream.Append(output3);
      outStream.RestartStrip();

      if (q < (n - p - 1))
      {
             output1.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, c));
            output2.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, b));
            output3.pos  =  UnityObjectToClipPos(input[0].pos  +
mul(_TRS, d));

        outStream.Append(output1);
        outStream.Append(output2);
        outStream.Append(output3);
        outStream.RestartStrip();
      }
    }
  }
}
```

This is the part where the triangle is divided by spherical linear interpolation. n is the number of triangle divisions. edge_p1 and edge_p2 find the starting point of the triangle, and edge_p3 and ege_p4 find the midpoint of the split edge. The qslerp function is a function that finds spherical linear interpolation. The definition of qslerp is as follows:

Listing 3.7: Definition of qslerp in Quaternion.cginc

```
// a: start Quaternion b: target Quaternion t: ratio
float4 qslerp(float4 a, float4 b, float t)
{
  float4 r;
  float t_ = 1 - t;
  float wa, wb;
   float theta = acos(a.x * b.x + a.y * b.y + a.z * b.z + a.w *
b.w);
  float sn = sin(theta);
  wa = sin (t_ * theta) / sn;
  wb = sin(t * theta) / sn;
  rx = wa * ax + wb * bx;
  ry = wa * ay + wb * by;
  rz = wa * az + wb * bz;
  rw = wa * aw + wb * bw;
  normalize(r);
  return r;
}
```

Flow of triangle division 1

Next, I will explain the flow of the triangle division process. As an example, it is the flow when the number of divisions is 2 (n = 2).

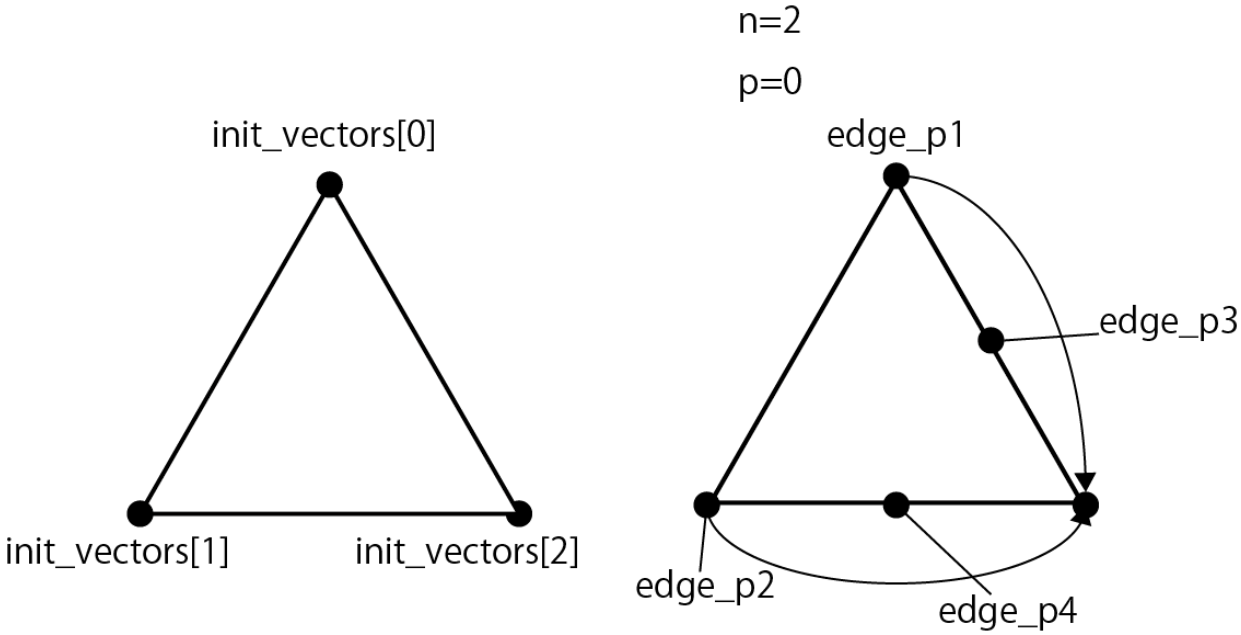

Figure 3.8: Triangle division process flow 1, calculation of edges_p1 to p4

Figure 3.8 shows the following code.

Listing 3.8: Calculation of edge_p1 to p4

```
for (int p = 0; p < n; p++)
{
  // edge index 1
  float4 edge_p1 = qslerp(init_vectors[i],
    init_vectors[i + 2], (float)p / n);
  float4 edge_p2 = qslerp(init_vectors[i + 1],
    init_vectors[i + 2], (float)p / n);
  float4 edge_p3 = qslerp(init_vectors[i],
    init_vectors[i + 2], (float)(p + 1) / n);
  float4 edge_p4 = qslerp(init_vectors[i + 1],
    init_vectors[i + 2], (float)(p + 1) / n);
```

The coordinates of edge_p1 to edge_p4 are obtained from the three points in the **init_vectors** array. When p = 0, p / n = 0/2 = 0 and edge_p1 = init_vectors [0], edge_p2 = init_vectors [1]. edge_p3 and edge_p4 are between init_vectors [0] and init_vectors [2] and between init_vectors [1] and init_vectors [2] at (p + 1) / n = (0 + 1) / 2 = 0.5, respectively. .. It is a flow that mainly divides the right side of the triangle.

Flow of triangle division 2

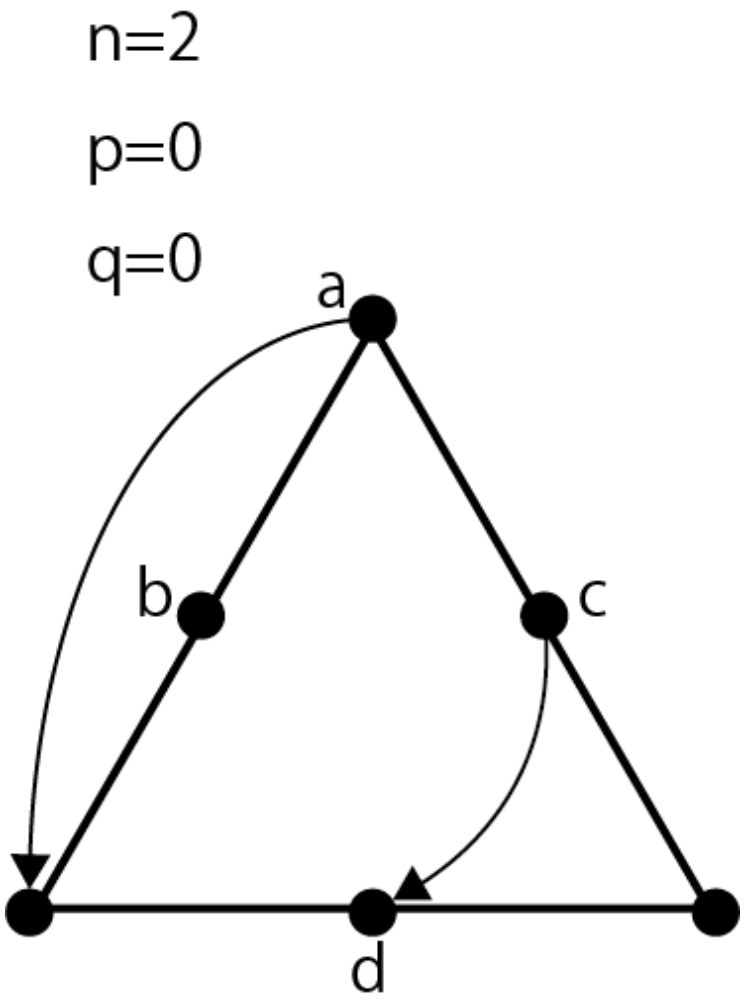

Figure 3.9: Triangle division process flow 2, abcd calculation

Figure 3.9 shows the following code.

Listing 3.9: Calculation of coordinates a, b, c, d

```
for (int q = 0; q < (n - p); q++)
{
  // edge index 2
  float4 a = qslerp(edge_p1, edge_p2, (float)q / (n - p));
  float4 b = qslerp(edge_p1, edge_p2, (float)(q + 1) / (n - p));
  float4 c, d;

  if(distance(edge_p3, edge_p4) < 0.00001)
  {
    c = edge_p3;
    d = edge_p3;
```

```
}
else {
  c = qslerp(edge_p3, edge_p4, (float)q / (n - p - 1));
  d = qslerp(edge_p3, edge_p4, (float)(q + 1) / (n - p - 1));
}
```

The coordinates of the vertex abcd are calculated using edge_p1 to p4 obtained in the previous section. It is a flow that mainly divides the left side of the triangle. Depending on the conditions, the coordinates of edge_p3 and edge_p4 will be the same. This happens when the right side of the triangle reaches a stage where it can no longer be divided. In that case, both c and d take the lower right coordinates of the triangle.

Flow of triangle division 3

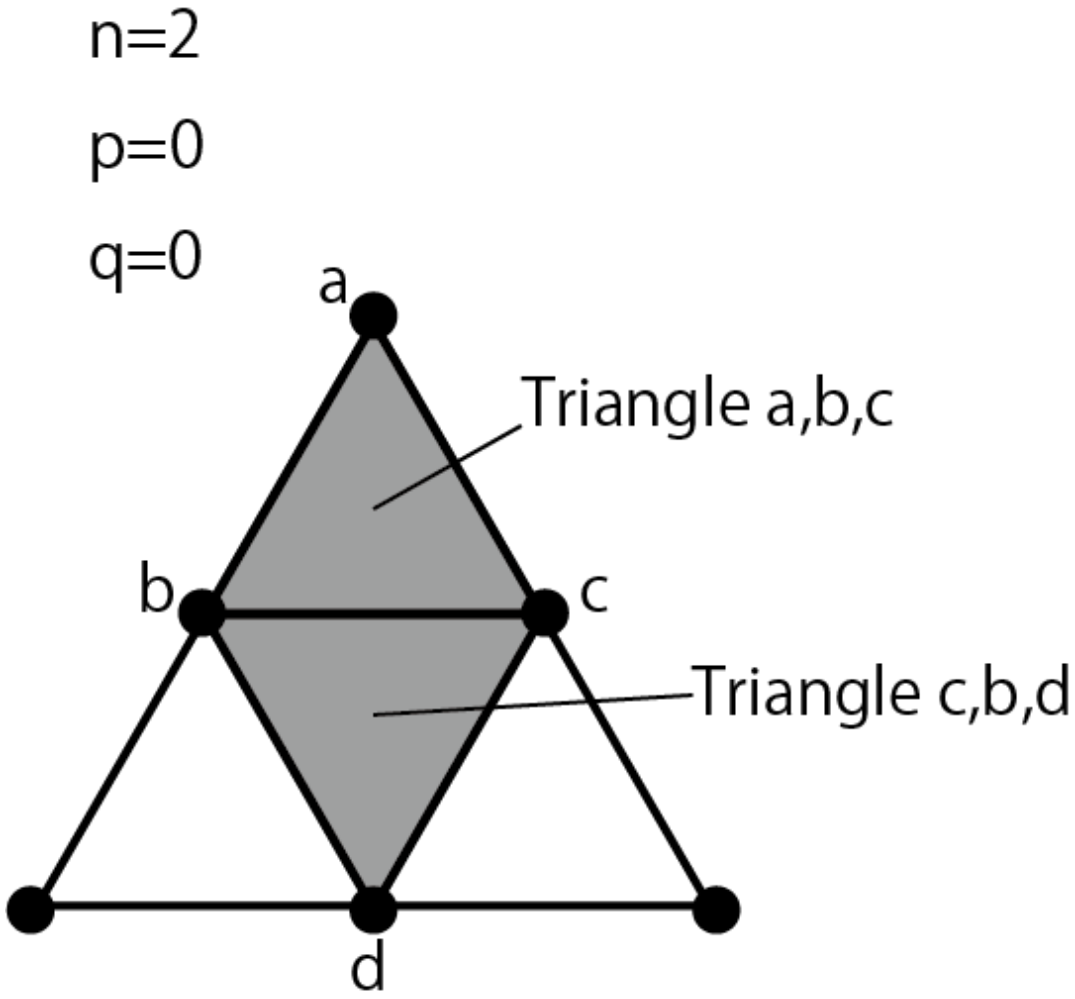

Figure 3.10: Flow of triangle division processing 3, output triangle abc, triangle cbd

Figure 3.10 shows the following code.

Listing 3.10: Output the triangle connecting the coordinates a, b, c & the triangle connecting the coordinates c, b, d

```
output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, a));
output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, b));
output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS, c));

outStream.Append(output1);
outStream.Append(output2);
outStream.Append(output3);
outStream.RestartStrip();

if (q < (n - p - 1))
{
   output1.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS,
c));
   output2.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS,
b));
   output3.pos = UnityObjectToClipPos(input[0].pos + mul(_TRS,
d));
  outStream.Append(output1);
  outStream.Append(output2);
  outStream.Append(output3);
  outStream.RestartStrip();
}
```

Convert the calculated coordinates of a, b, c, d to the coordinates for the screen by multiplying by UnityObjectToClipPos or the world coordinate transformation matrix. After that, outStream.Append and outStream.RestartStrip output two triangles connecting a, b, c and c, b, d.

Flow of triangle division 4

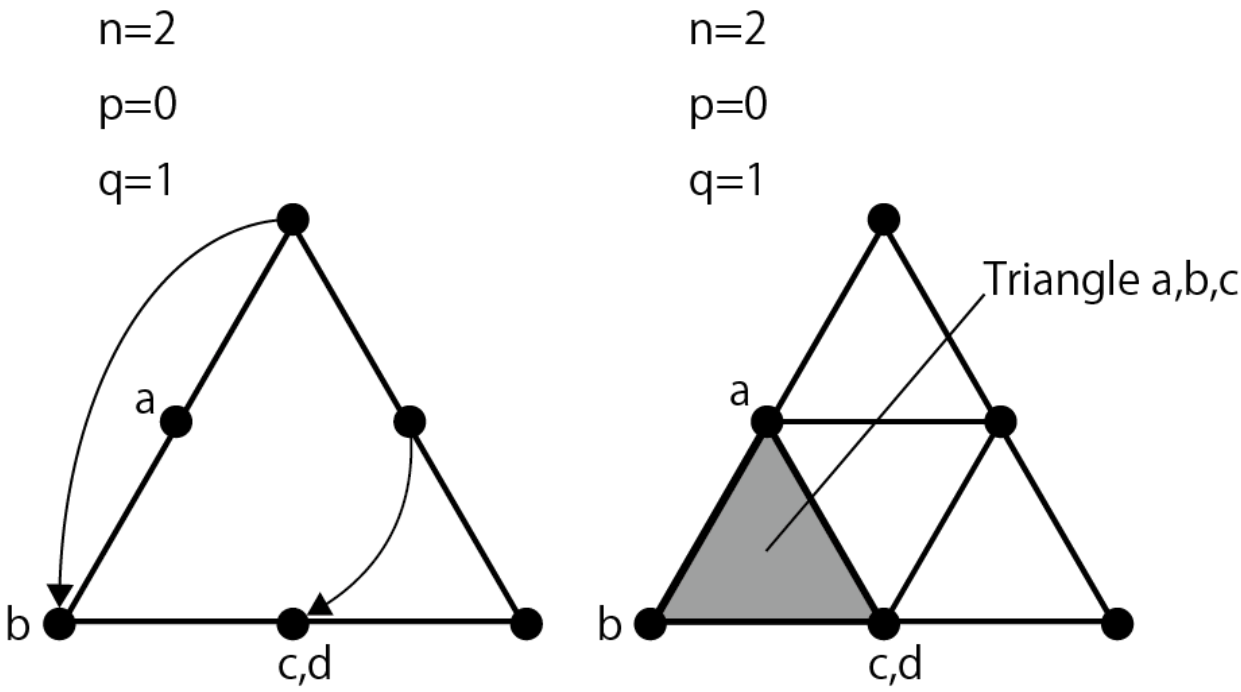

Figure 3.11: Flow of triangle division processing 4, when q = 1

When q = 1, a is 1/2 = 0.5, so it is in the middle of edge_p1 and edge_p2, and b is 1/1 = 1, so it is in the position of edge_p2. Since c is 1/1 = 1, edge_p4 is calculated, and d is calculated for the time being, but it is not used because it does not fall under the condition of if (q <(n --p -1)). Outputs a triangle connecting a, b, and c.

Flow of triangle division 5

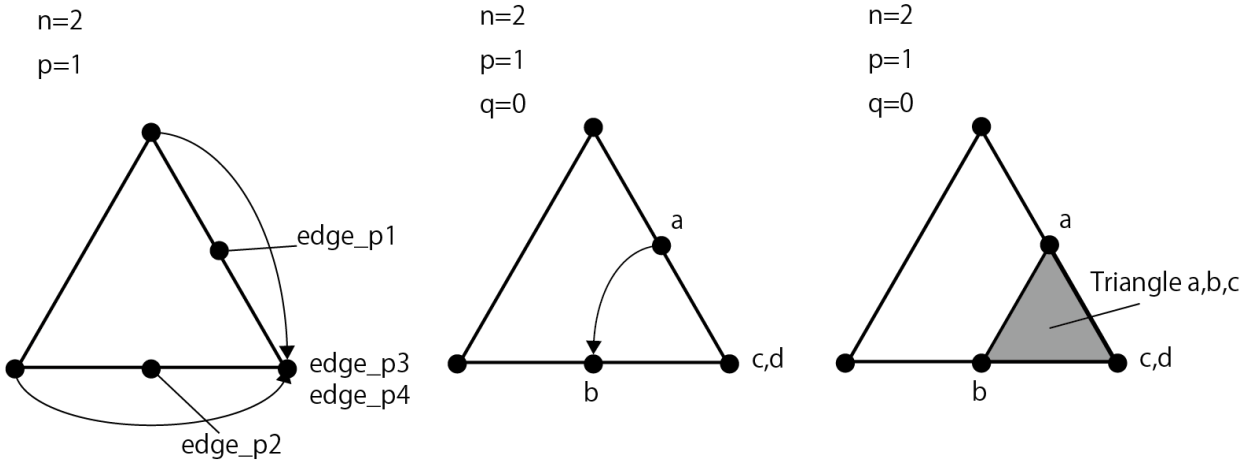

Figure 3.12: Triangle division process flow 5, when p = 1

This is the flow when the for statement of q ends and p = 1. Since p / n = 1/2 = 0.5, edge_p1 is between init_vectors [0] and init_vectors [2], and edge_p2 is between init_vectors [1] and init_vectors [2]. The subsequent coordinate calculation of a, b, c, d and the output of the triangles a, b, c are the same as the above processing. You have now divided one triangle into four. All the triangles of the octahedron are processed up to the above.

3.4.3 Bonus

In addition to this, we have prepared three samples that cannot be introduced due to space limitations, so please take a look if you are interested.

- OctahedronSphere の GPUInstancing 版（SampleOctahedronSphereInstancing シーン）
- OctahedronSphere standalone version that can be divided into 9 stages (SampleOctahedronSphereMultiVertex scene)
- GPU Instancing version of OctahedronSphere that can be divided up to 9 stages (SampleOctahedronSphereMultiVertexInstancing scene)

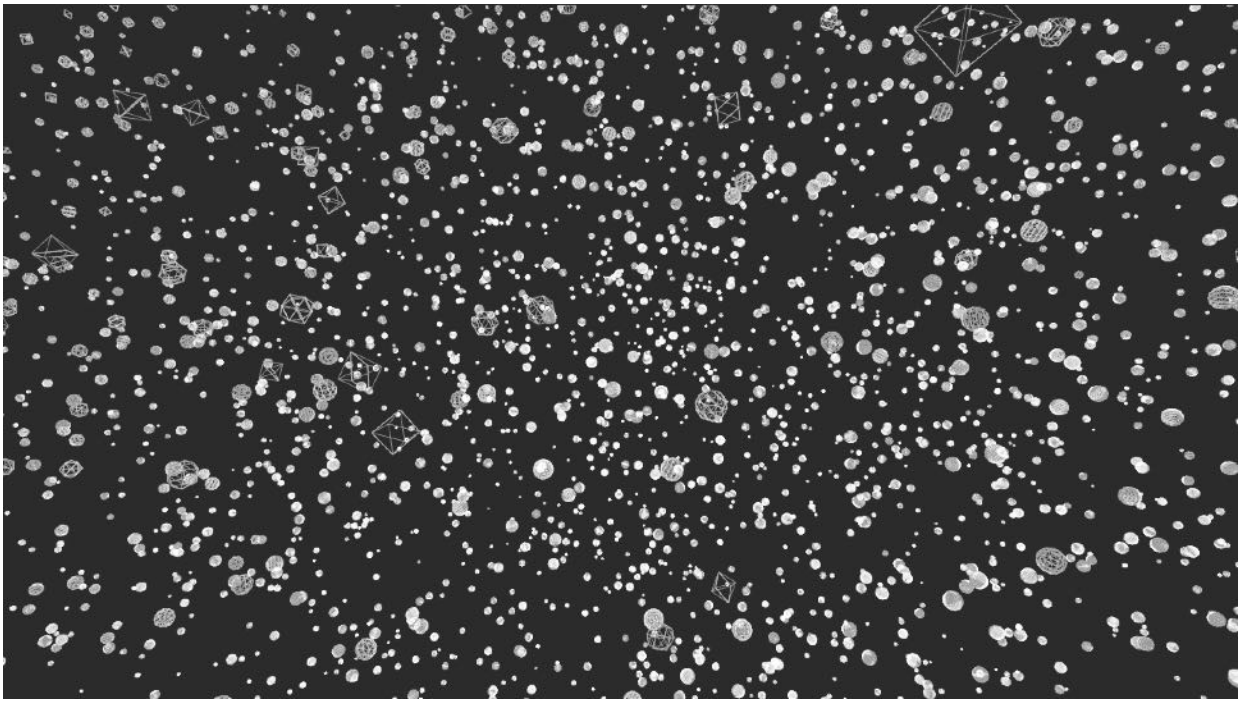

図3.13: SampleOctahedronSphereMultiVertexInstancingシーン

3.5 Summary

In this chapter, we explained the application of Geometry Shader for line representation. Geometry Shader usually divides polygons and creates plate polygons of particles, but you should also try to find interesting expressions by using the property of dynamically increasing the number of vertices.

Chapter 4 Projection Spray

4.1 Introduction

Hello! It's Sugihiro Nori! I'm sorry I couldn't write an article in the previous "UnityGraphicsPrograming Vol1"! Thanks to Oishi-kun for writing on behalf of me (._.)

Here, I would like to explain the spray program that I could not write last time. The code of Unity is a little better than the one at the time of Vol1!

First, let's implement a simple lighting effect by ourselves without using Unity's Built-in lights. Then, as an application, I will explain the development of the process of painting 3D objects with a spray. The concept of this chapter is to follow the flow of creating your own functions by referring to UnityCG.cginc and built-in processing and applying them to new functions. I think.

4.1.1 Sample repository

The samples in this chapter are in the ProjectionSpray folder at https://github.com/IndieVisualLab/UnityGraphicsProgramming2 .

4.2 Implementation of Light Component

Light in Unity is extremely useful. Just install a light object and the world will be brighter. When you select a shadow from the Inspector, a shadow map is automatically created and the shadow is cast from the object.

First of all, we will implement the light independently while watching how Unity implements the light.

4.2.1 Unity Built-in Shader

In Unity, you can download the material shaders included by default and the internally used CGINC files from the Unity Download Archive.

It will be helpful when writing your own shader, and you can learn more about CG depiction, so I recommend you to download and see it!

Figure 4.1: https://unity3d.com/jp/get-unity/download/archive

The writing-related files that may be relevant in this chapter are:

- CGIncludes/UnityCG.cginc
- CGIncludes / AutoLight.cginc
- CGIncludes/Lighting.cginc
- CGIncludes/UnityLightingCommon.cginc

Let's take a look at the basic Lambert Lighting. (Listing 4.1) Mr. Lambert thought.

Listing 4.1: Lighting.cginc

```
1: struct SurfaceOutput {
2:     fixed3 Albedo;
3:     fixed3 Normal;
4:     fixed3 Emission;
```

```
 5:     half Specular;
 6:     fixed Gloss;
 7:     fixed Alpha;
 8: };
 9: ~~
10: inline fixed4 UnityLambertLight (SurfaceOutput s, UnityLight
light)
11: {
12:     fixed diff = max (0, dot (s.Normal, light.dir));
13:
14:     fixed4 c;
15:     c.rgb = s.Albedo * light.color * diff;
16:     c.a = s.Alpha;
17:     return c;
18: }
```

In the actual lighting calculation, the diffuse value is calculated from the inner product of the direction from the light to the mesh and the normal direction of the mesh. Listing 4.1

```
fixed diff = max (0, dot (s.Normal, light.dir));
```

For Unity Lights that are undefined in Lighting.cginc, they are defined in UnityLightingCommon.cginc and contain information about the color and direction of the lights. Listing 4.2

Listing 4.2: UnityLightingCommon.cginc

```
 1: struct UnityLight
 2: {
 3:     half3 color;
 4: half3 you;
 5:
 6:     // Deprecated: Ndotl is now calculated on the fly
 7:     // and is no longer stored. Do not used it.
 8:     half  ndotl;
 9: };
```

4.2.2 Displaying mesh normals

Looking at the actual lighting process, I found that the calculation of lighting requires mesh normal information, so let's take a quick look at how to display mesh normal information in Shader. ..

See the scene in **00_viewNormal.unity** in the sample project .

The object has a material that outputs normal information as a color, and its shader is shown in Listing 4.3 .

Listing 4.3: simple-showNormal.shader

```
 1: struct appdata
 2: {
 3:     float4 vertex : POSITION;
 4: float3 normal: NORMAL;
 5: };
 6:
 7: struct v2f
 8: {
 9:     float3 worldPos : TEXCOORD0;
10:     float3 normal : TEXCOORD1;
11:     float4 vertex : SV_POSITION;
12: };
13:
14: v2f vert (appdata v)
15: {
16:     v2f o;
17:     o.vertex = UnityObjectToClipPos(v.vertex);
18:     o.normal = UnityObjectToWorldNormal(v.normal);
19:     return o;
20: }
21:
22: half4 frag (v2f i) : SV_Target
23: {
24:     fixed4 col = half4(i.normal,1);
25:     return col;
26: }
```

The vertex shader (v2f vert) calculates the normal direction of the mesh in the world coordinate system and passes it to the fragment shader (half4 frag). In the fragment shader, the passed normal information is converted to color with the x component as R, the y component as G, and the z component as B, and output as it is. Listing 4.3

Even if the part looks black on the image, the normal x may actually have a negative value. Figure 4.2

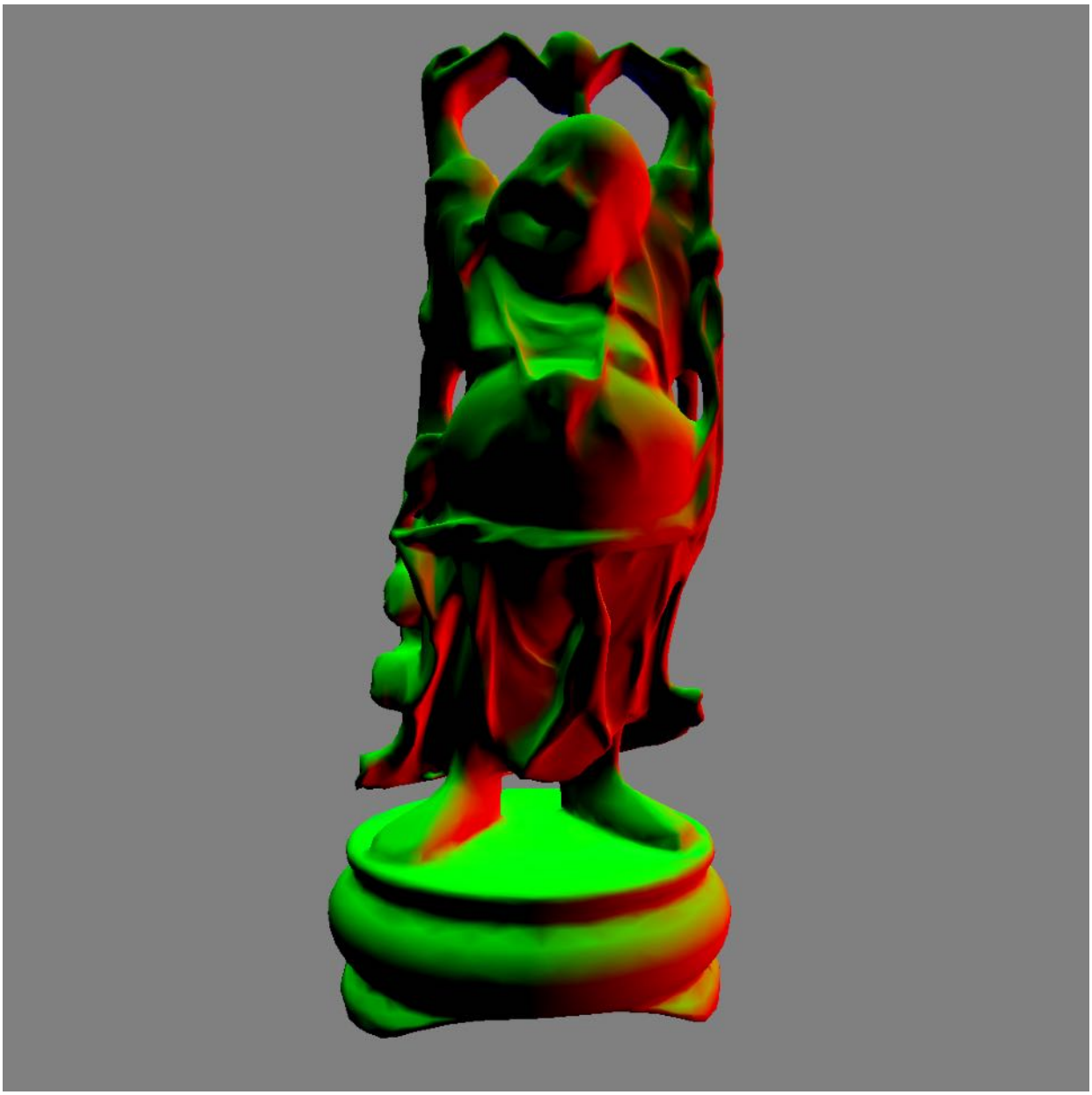

図4.2: 00_viewNormal.unity

The mesh is now ready for lighting.

Built-in shader helper function

There is a built-in utility function in UnityCG.cginc that makes it easy to write shaders. For example, the vertex position used in Listing 4.3`UnityObjectToClipPos` transforms from the object (local) coordinate system to the clip coordinate system. Also, `UnityObjectToWorldNormal`the function of is converting the normal direction from the object coordinate system to the world coordinate system.

For other functions, please refer to UnityCG.cginc or the official manual as it is convenient for writing shaders. https://docs.unity3d.com/ja/current/Manual/SL-BuiltinFunctions.html

Also, if you want to know more about coordinate transformation and each coordinate system, you may be able to learn more by referring to Unity Graphics Programming vol.1 and Mr. Fukunaga's "Chapter 9 Multi Plane Perspective Projection".

4.2.3 Point Light

See the scene at **01_pointLight.unity** in the sample project .

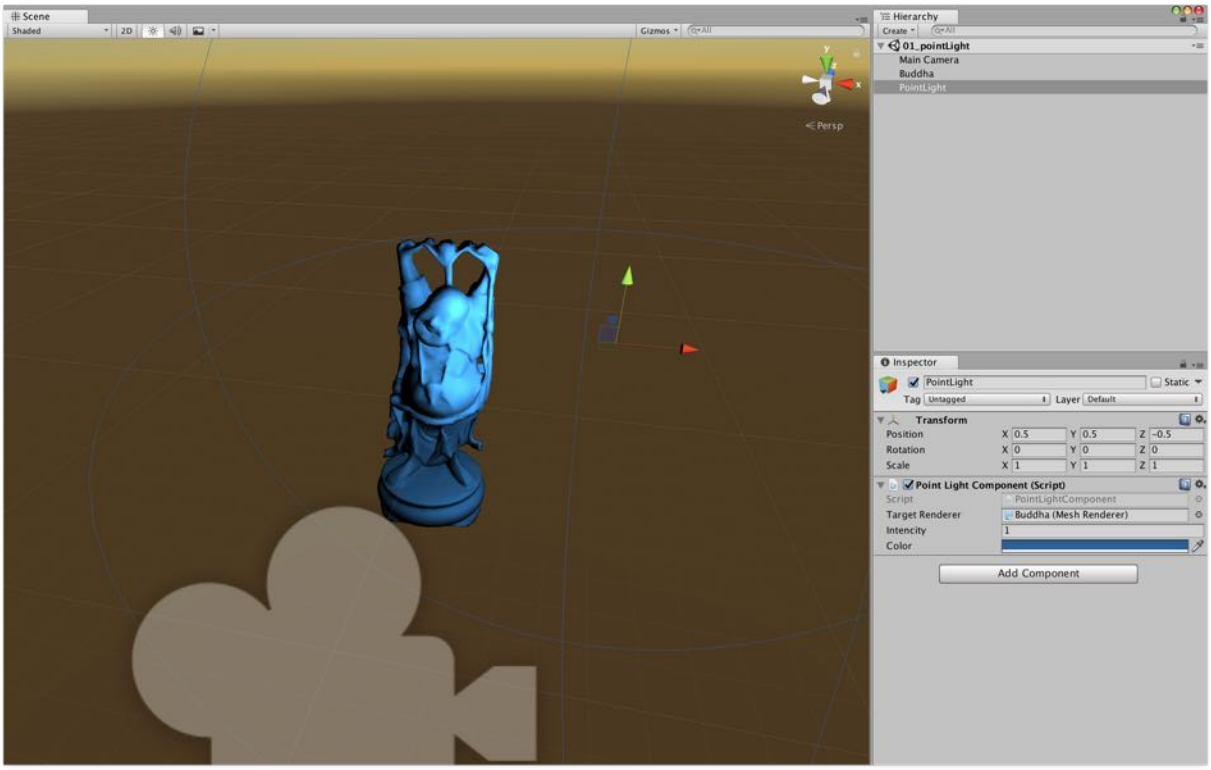

図4.3: 01_pointLight.unity

A point light source is a light source that illuminates all directions from a certain point. The scene has a Buddha mesh object and a PointLight object. The PointLight object has a script (Listing 4.4) for sending light information to the mesh, and based on that light information, the lighting result is displayed as a material (Listing 4.5).

Listing 4.4: PointLightComponent.cs

```
 1: using UnityEngine;
 2:
 3: [ExecuteInEditMode]
 4: public class PointLightComponent : MonoBehaviour
 5: {
 6:     static MaterialPropertyBlock mpb;
 7:
 8:     public Renderer targetRenderer;
 9:     public float intensity = 1f;
10:     public Color color = Color.white;
11:
12:     void Update()
```

```
13:     {
14:         if (targetRenderer == null)
15:             return;
16:         if (mpb == null)
17:             mpb = new MaterialPropertyBlock();
18:
19:         targetRenderer.GetPropertyBlock(mpb);
20:         mpb.SetVector("_LitPos", transform.position);
21:         mpb.SetFloat("_Intensity", intensity);
22:         mpb.SetColor("_LitCol", color);
23:         targetRenderer.SetPropertyBlock(mpb);
24:     }
25:
26:     private void OnDrawGizmos()
27:     {
28:         Gizmos.color = color;
29:                     Gizmos.DrawWireSphere(transform.position,
intensity);
30:     }
31: }
```

This component passes the position, intensity and color of the light to the target mesh. And, the material to perform the writing process is set based on the received information.

Values are set from CSharp for each property of `"_LitPos"`, `"_LitCol"`and `"_Intensity"`of the material .

Listing 4.5: simple-pointLight.shader

```
 1: Shader "Unlit/Simple/PointLight-Reciever"
 2: {
 3:     Properties
 4:     {
 5:         _LitPos("light position", Vector) = (0,0,0,0)
 6:         _LitCol("light color", Color) = (1,1,1,1)
 7:         _Intensity("light intensity", Float) = 1
 8:     }
 9:     SubShader
10:     {
11:         Tags { "RenderType"="Opaque" }
12: LOD 100
13:
14:         Pass
15:         {
16:             CGPROGRAM
```

```
17:             #pragma vertex vert
18:             #pragma fragment frag
19:
20:             #include "UnityCG.cginc"
21:
22:             struct appdata
23:             {
24:                 float4 vertex : POSITION;
25: float3 normal: NORMAL;
26:             };
27:
28:             struct v2f
29:             {
30:                 float3 worldPos : TEXCOORD0;
31:                 float3 normal : TEXCOORD1;
32:                 float4 vertex : SV_POSITION;
33:             };
34:
35:             half4 _LitPos, _LitCol;
36:             half _Intensity;
37:
38:             v2f vert (appdata v)
39:             {
40:                 v2f o;
41:                 o.vertex = UnityObjectToClipPos(v.vertex);
42:                       o.worldPos = mul(unity_ObjectToWorld,
v.vertex).xyz;
43: // Pass the position of the mesh in the world coordinate
system to the fragment shader
44:                                                 o.normal   =
UnityObjectToWorldNormal(v.normal);
45:                 return o;
46:             }
47:
48:             fixed4 frag (v2f i) : SV_Target
49:             {
50:                 half3 to = i.worldPos - _LitPos;
51: // Vector from light position to mesh position
52:                 half3 lightDir = normalize(to);
53:                 half dist = length(to);
54:                 half atten =
55:                      _Intensity * dot(-lightDir, i.normal) /
(dist * dist);
56:
57:                 half4 col = max(0.0, atten) * _LitCol;
58:                 return col;
59:             }
60:             ENDCG
```

```
61:          }
62:      }
63: }
```

The lighting calculation is based on the basic Lambert Lighting (Listing 4.1) calculation, and the intensity is attenuated in inverse proportion to the square of the distance. Listing 4.5

```
half atten = _Intensity * dot(-lightDir, i.normal) / (dist * dist);
```

It's a simple system of one light for one model, but I was able to implement the lighting process.

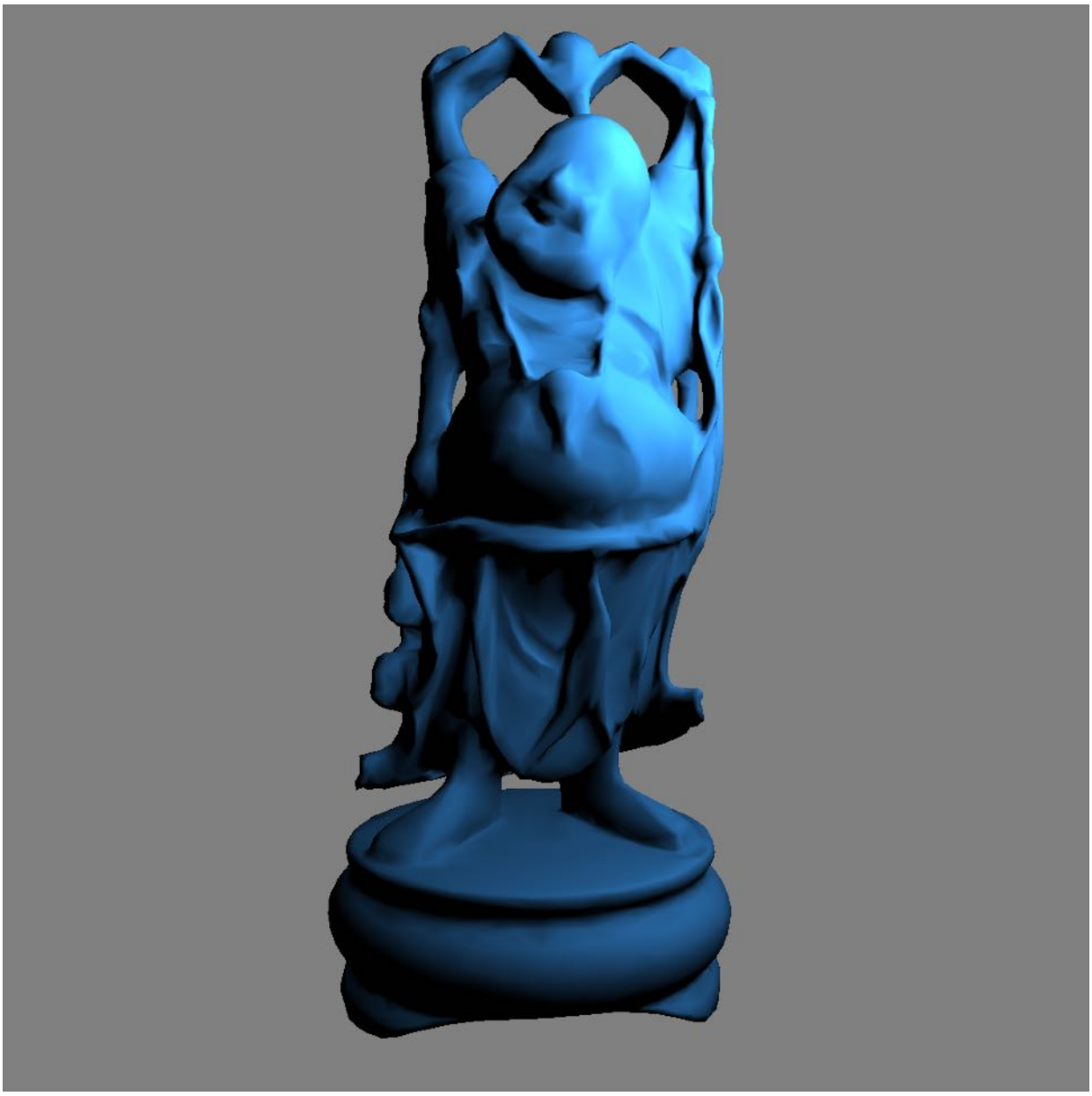

図4.4: 01_pointLight.unity

4.2.4 SpotLight

Next, let's implement the spotlight. Unlike point lights, spotlights are directional lights that emit light in one direction.

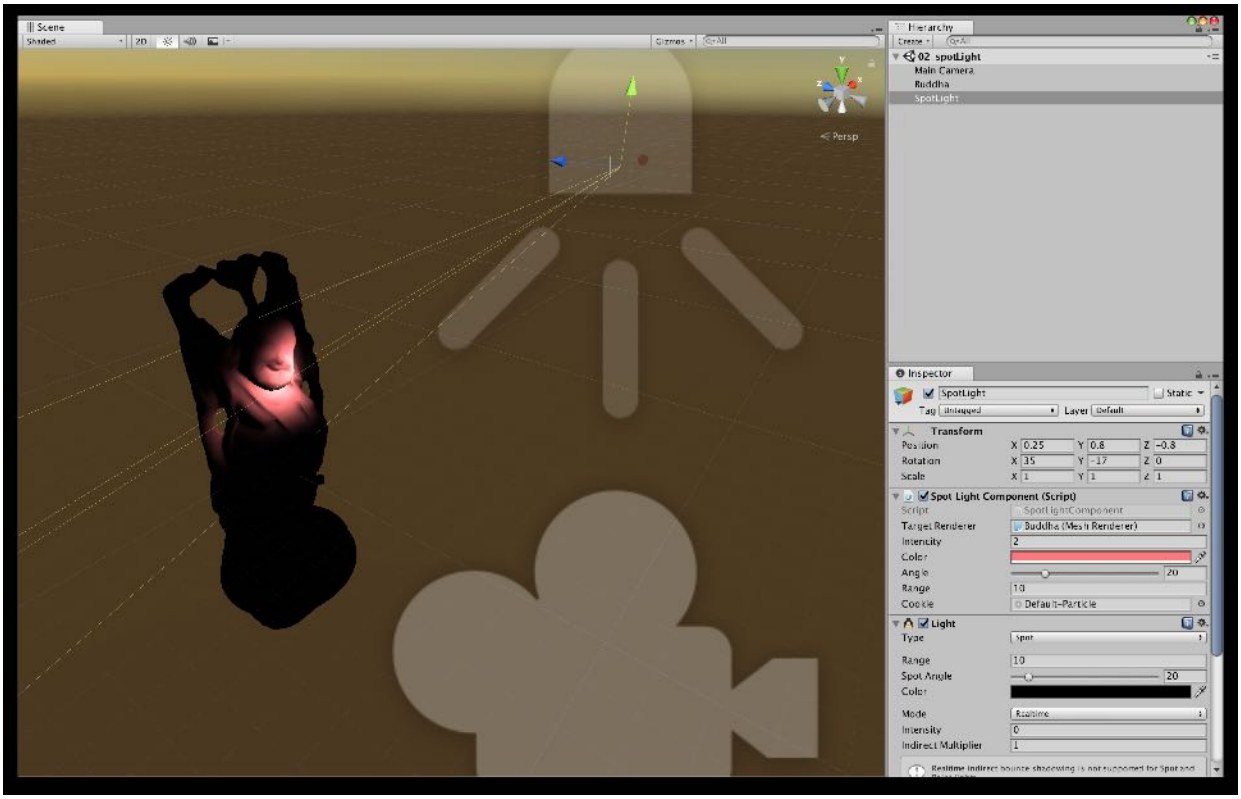

図4.5: 02_spotLight.unity

We are using standard Unity lights here just for the Gizmo display of the spotlights. Figure 4.5

Since the spotlight is directional, the direction of the light and the spot angle (angle) information will be added to the point light in addition to the position information. This information is of light `worldToLightMatrix`, `projectionMatrix`respectively, as `Matrix4x4`(that's Shader `float4x4`Passes in the properties of).

In addition, the spotlight can also set a Light Cookie. (Unity has a default LightCookie, but I couldn't select it from the editor, so I'm using the Default-Particle texture)

Listing 4.6: SpotLightComponent.cs

```
1: using UnityEngine;
2:
3: [ExecuteInEditMode]
```

```
 4: public class SpotLightComponent : MonoBehaviour
 5: {
 6:     static MaterialPropertyBlock mpb;
 7:
 8:     public Renderer targetRenderer;
 9:     public float intensity = 1f;
10:     public Color color = Color.white;
11:     [Range(0.01f, 90f)] public float angle = 30f;
12:     public float range = 10f;
13:     public Texture cookie;
14:
15:     void Update()
16:     {
17:         if (targetRenderer == null)
18:             return;
19:         if (mpb == null)
20:             mpb = new MaterialPropertyBlock();
21:
22: // Calculating projectionMatrix
23:           var projMatrix = Matrix4x4.Perspective(angle, 1f,
0f, range);
24:                                  var   worldToLightMatrix   =
transform.worldToLocalMatrix;
25:
26:         targetRenderer.GetPropertyBlock(mpb);
27:         mpb.SetVector("_LitPos", transform.position);
28:         mpb.SetFloat("_Intensity", intensity);
29:         mpb.SetColor("_LitCol", color);
30:                             mpb.SetMatrix("_WorldToLitMatrix",
worldToLightMatrix);
31:         mpb.SetMatrix("_ProjMatrix", projMatrix);
32:         mpb.SetTexture("_Cookie", cookie);
33:         targetRenderer.SetPropertyBlock(mpb);
34:     }
35: }
```

projectionMatrix is `Matrix4x4.Perspective(angle, 1f, 0f, range)`calculated by.

Shader calculates and displays the lighting process based on the parameter information received from the spotlight. Listing 4.7

Listing 4.7: simple-spotLight.shader

```
 1: uniform float4x4 _ProjMatrix, _WorldToLitMatrix;
 2:
```

```
 3: sampler2D _Cookie;
 4: half4 _LitPos, _LitCol;
 5: half _Intensity;
 6:
 7: ~~
 8:
 9: fixed4 frag (v2f i) : SV_Target
10: {
11:     half3 to = i.worldPos - _LitPos.xyz;
12:     half3 lightDir = normalize(to);
13:     half dist = length(to);
14:       half atten = _Intensity * dot(-lightDir, i.normal) /
(dist * dist);
15:
16:             half4  lightSpacePos  =  mul(_WorldToLitMatrix,
half4(i.worldPos, 1.0));
17:     half4 projPos = mul(_ProjMatrix, lightSpacePos);
18: projPos.z * = -1;
19: half2 litUv = projPos.xy / projPos.z;
20: litUv = litUv * 0.5 + 0.5;
21:     half lightCookie = tex2D(_Cookie, litUv);
22:     lightCookie *=
23:         0<litUv.x && litUv.x<1 && 0<litUv.y && litUv.y<1 &&
0<projPos.z;
24:
25:     half4 col = max(0.0, atten) * _LitCol * lightCookie;
26:     return col;
27: }
28:
```

You can see that it's basically the same as a point light except for the fragment shader. Listing 4.7

From the 16th line to the 22nd line, the intensity at each point of the spotlight is calculated. When viewed from the position of the light, the light cookie at that point is sampled to determine whether the point is within the range of the light and the intensity of the light.

図4.6: 02_spotLight.unity

Handling of built-in cookies

For the spotlight cookie processing, `UnitySpotCookie()`the part in the built-in CGINC, AutoLight.cginc will be helpful.

Listing 4.8: AutoLight.cginc

```
 1: #ifdef SPOT
 2: sampler2D _LightTexture0;
      3:      unityShadowCoord4x4
unity_WorldToLight;
 4: sampler2D _LightTextureB0;
      5:       inline      fixed
UnitySpotCookie(unityShadowCoord
4 LightCoord)
 6: {
 7:     return tex2D(
 8:         _LightTexture0,
 9:              LightCoord.xy  /
LightCoord.w + 0.5
10:     ).w;
11: }
12:          inline         fixed
UnitySpotAttenuate(unityShadowCo
ord3 LightCoord)
13: {
14:     return tex2D(
15:         _LightTextureB0,
16:              dot(LightCoord,
LightCoord).xx
17:     ).UNITY_ATTEN_CHANNEL;
18: }
19:                         #define
UNITY_LIGHT_ATTENUATION(destName
, input, worldPos) \
20:             unityShadowCoord4
lightCoord = mul( \
21:          unity_WorldToLight,
\
22:
unityShadowCoord4(worldPos, 1) \
23:     ); \
24:             fixed   shadow   =
UNITY_SHADOW_ATTENUATION(input,
worldPos); \
25:     fixed destName = \
26:         (lightCoord.z > 0) *
\
27:
UnitySpotCookie(lightCoord) * \
28:
UnitySpotAttenuate(lightCoord.xy
z) * shadow;
```

```
29: #endif
```

4.2.5 Shadow Shadow implementation

Finally, as a lighting implementation, let's implement a shadow.

Light comes out of the light, the mesh that is directly exposed to light becomes brighter, there is something else between the light and the mesh, and the mesh that is blocked by light becomes darker. This is the shadow.

As a procedure, roughly

- Create a depth texture as seen from the light.
- When rendering an object, if the depth from the light at that point is greater than the depth texture, the point will be shadowed because it is blocked by other objects.

It will be in the form of. This time we need a depth texture from the position of the light, so we'll add a Camera component to SpotLight to create the depth texture as seen from the light.

図4.7: 03_spotLight-withShadow.unity

Camera (built-in) is attached to SpotLightComponent (self-made). Figure 4.7

Listing 4.9: SpotLightWithShadow.cs

```
 1: Shader depthRenderShader {
 2:     get { return Shader.Find("Unlit/depthRender"); }
 3: }
 4:
 5: new Camera camera
 6: {
 7:     get
 8:     {
 9:         if (_c == null)
10:         {
11:             _c = GetComponent<Camera>();
12:             if (_c == null)
13:                 _c = gameObject.AddComponent<Camera>();
```

```
14:             depthOutput = new RenderTexture(
15:                 shadowMapResolution,
16:                 shadowMapResolution,
17:                 16,
18:                 RenderTextureFormat.RFloat
19:             );
20:             depthOutput.wrapMode = TextureWrapMode.Clamp;
21:             depthOutput.Create();
22:             _c.targetTexture = depthOutput;
23:                   _c.SetReplacementShader(depthRenderShader,
"RenderType");
24:             _c.clearFlags = CameraClearFlags.Nothing;
25:             _c.nearClipPlane = 0.01f;
26:             _c.enabled = false;
27:         }
28:         return _c;
29:     }
30: }
31: Room _c;
32: RenderTexture depthOutput;
33:
34: void Update()
35: {
36:     if (mpb == null)
37:         mpb = new MaterialPropertyBlock();
38:
39:     var currentRt = RenderTexture.active;
40:     RenderTexture.active = depthOutput;
41:     GL.Clear(true, true, Color.white * camera.farClipPlane);
42:     camera.fieldOfView = angle;
43:     camera.nearClipPlane = 0.01f;
44:     camera.farClipPlane = range;
45:     camera.Render();
46: // Camera rendering is done manually in the script
47:     RenderTexture.active = currentRt;
48:
49:     var projMatrix = camera.projectionMatrix;
50: // Use the camera's projection matrix
51:     var worldToLightMatrix = transform.worldToLocalMatrix;
52:
53:     ~~
54: }
```

The C # script is almost the same as the shadowless version, but with the camera set to render the depth texture and the ReplacementShader to render the depth. Also, since we have a camera this time `Matrix4x4.Perspective`, `Camera.projectionMatrix`we will use the projection matrix instead of .

The shader for depth texture generation looks like this:

Listing 4.10: depthRender.shader

```
 1:      v2f vert (float4 pos : POSITION)
 2:      {
 3:          v2f o;
 4:          o.vertex = UnityObjectToClipPos(pos);
 5:          o.depth = abs(UnityObjectToViewPos(pos).z);
 6:          return o;
 7:      }
 8:
 9:      float frag (v2f i) : SV_Target
10:      {
11:          return i.depth;
12:      }
```

The generated depth texture (Fig. 4.8) outputs the z-coordinate value of the position of the object in the light coordinate system (camera coordinate system).

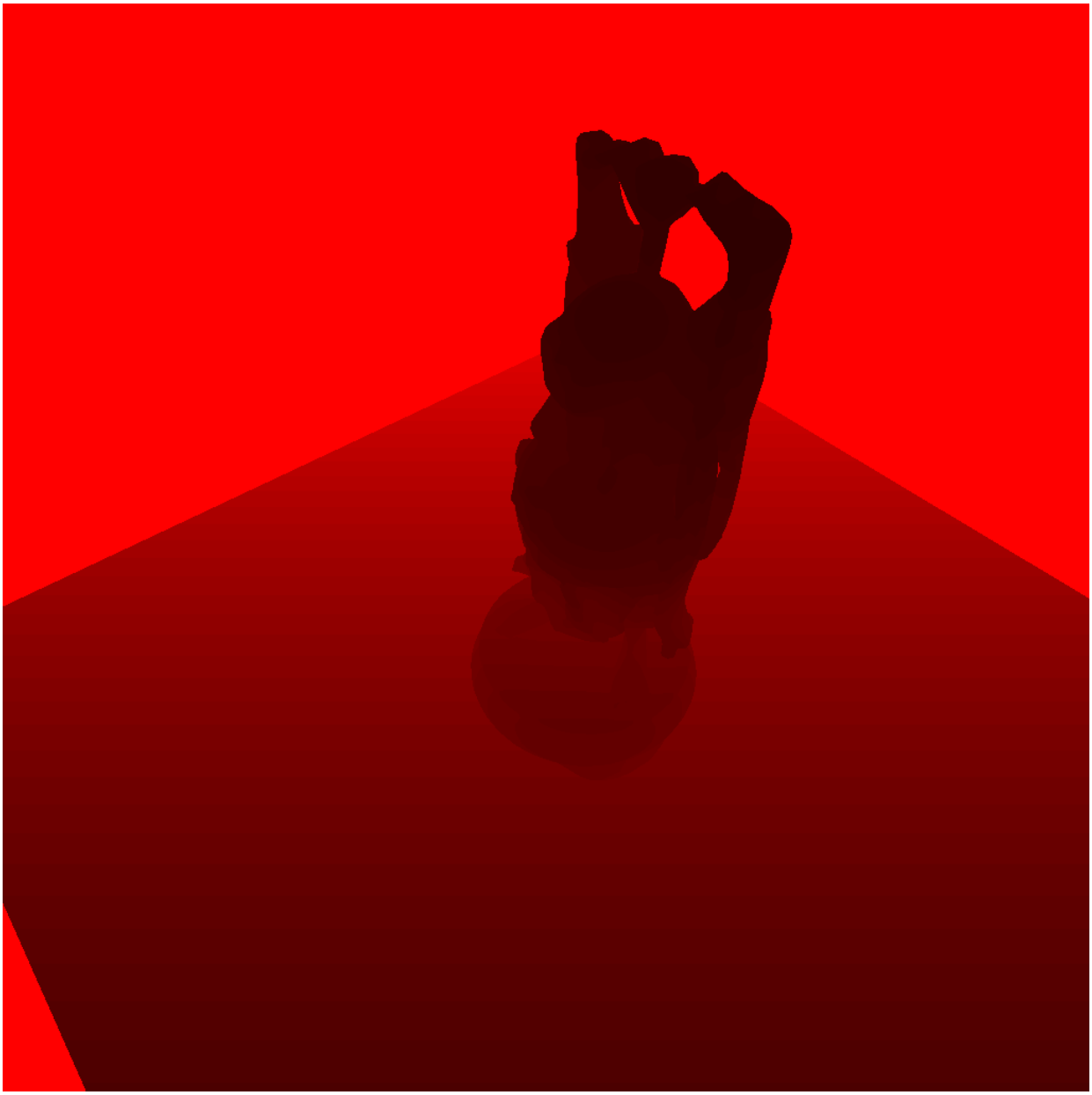

図4.8: light depthTexture

`depthOutput`Pass the generated depth texture ( ) to the mesh object and render the object. The calculated part of the shadow of the object looks like this:

Listing 4.11: simple-spotLight-withShadow.shader

```
1: fixed4 frag (v2f i) : SV_Target
2: {
```

```
 3:     ///diffuse lighting
 4:     half3 to = i.worldPos - _LitPos.xyz;
 5:     half3 lightDir = normalize(to);
 6:     half dist = length(to);
 7:       half atten = _Intensity * dot(-lightDir, i.normal) /
(dist * dist);
 8:
 9:     ///spot-light cookie
10:             half4  lightSpacePos  =  mul(_WorldToLitMatrix,
half4(i.worldPos, 1.0));
11:     half4 projPos = mul(_ProjMatrix, lightSpacePos);
12: projPos.z * = -1;
13: half2 litUv = projPos.xy / projPos.z;
14: litUv = litUv * 0.5 + 0.5;
15:     half lightCookie = tex2D(_Cookie, litUv);
16:     lightCookie *=
17:          0<litUv.x && litUv.x<1 && 0<litUv.y && litUv.y<1 &&
0<projPos.z;
18:
19:     ///shadow
20:     half lightDepth = tex2D(_LitDepth, litUv).r;
21: // _LitDepth is passed the depth texture seen from the light
22:         atten  *=  1.0  -  saturate(10*abs(lightSpacePos.z)  -
10*lightDepth);
23:
24:     half4 col = max(0.0, atten) * _LitCol * lightCookie;
25:     return col;
26: }
```

Made the depth texture by the camera `tex2D(_LitTexture, litUv).r`and `lightSpacePos.z`is, z value of the vertex position of both viewed from the light object is stored. Since the texture is `_LitTexture`the information of the surface seen from the light = the surface exposed to the light, it is judged whether it is a shadow or not by comparing it with the value (`lightDepth`) sampled from the depth texture `lightSpacePos.z`.

```
atten    *=    1.0    -    saturate(10*abs(lightSpacePos.z)    -
10*lightDepth);
```

In the code here, the `lightDepth`higher `lightSpacePos.z`the value, the darker the surface.

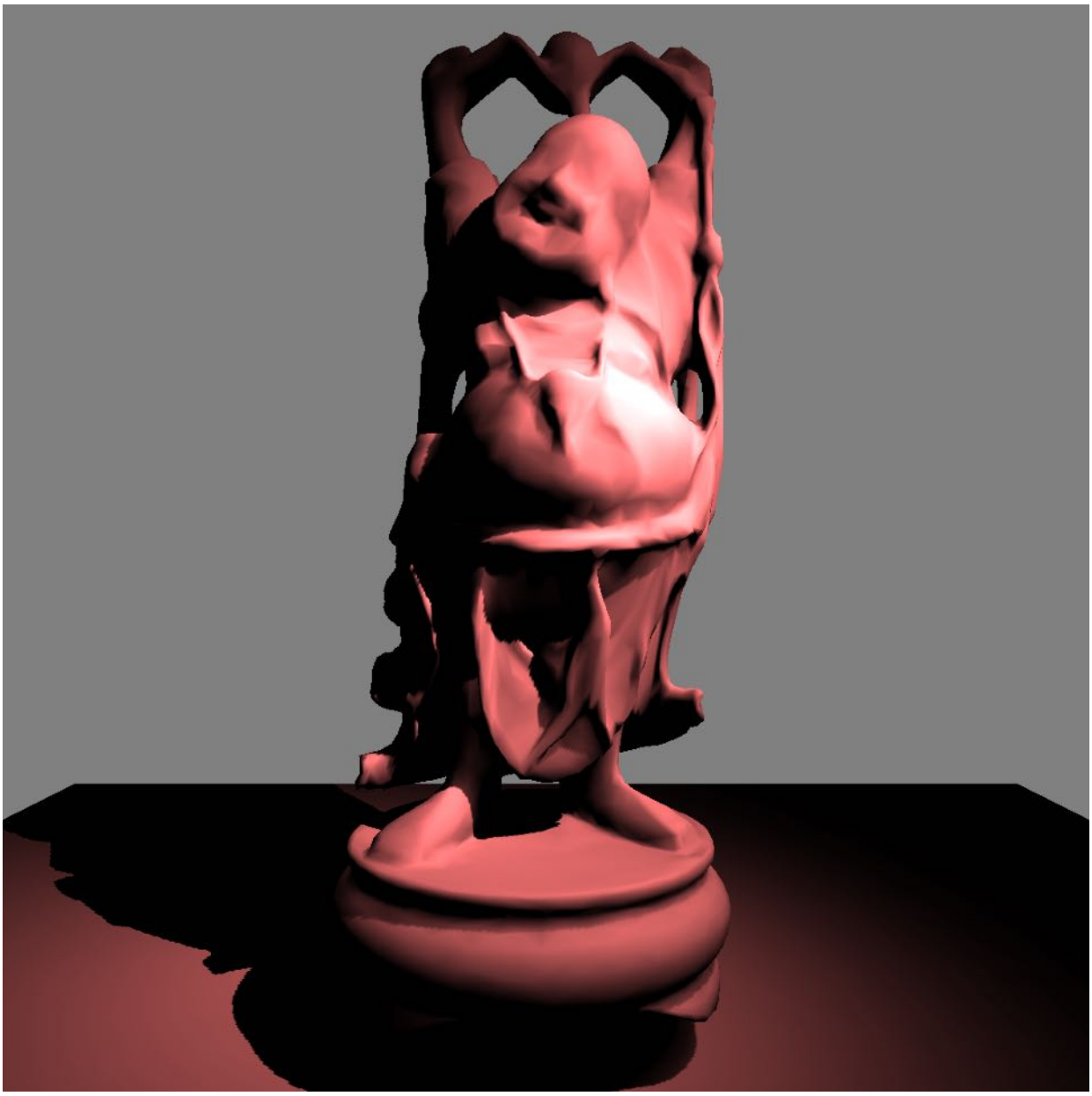

図4.9: 03_spotLight-withShadow.unity

You can now see the shadow of the object in the spotlight.

Using this spotlight and shadow implementation, we will implement a spray function that colors objects in real time.

Camera.projectionMatrix and Matrix4x4.Perspective are the same matrix

Unity scene: in Example, **compareMatrix.unity**

Listing 4.12: CompareMatrix.cs

```
 1:     float fov = 30f;
 2:     float near = 0.01f;
 3:     float far = 1000f;
 4:
 5:        camera.fieldOfView  =
fov;
 6:       camera.nearClipPlane =
near;
 7:        camera.farClipPlane  =
far;
 8:
 9:     Matrix4x4 cameraMatrix =
camera.projectionMatrix;
10:      Matrix4x4 perseMatrix =
Matrix4x4.Perspective(
11:         fov,
12:         1f,
13:         near,
14:         far
15:     );
```

4.3 Implementation of Projection Spray

From here, we will apply our own SpotLightComponent to implement the spray function that allows you to paint objects.

Basically, it draws on the texture of the object based on the value of the lighting intensity. Since the Buddha object used this time does not have uv data, it is not possible to paste the texture as it is, but Unity has a function to generate UV for LightMap.

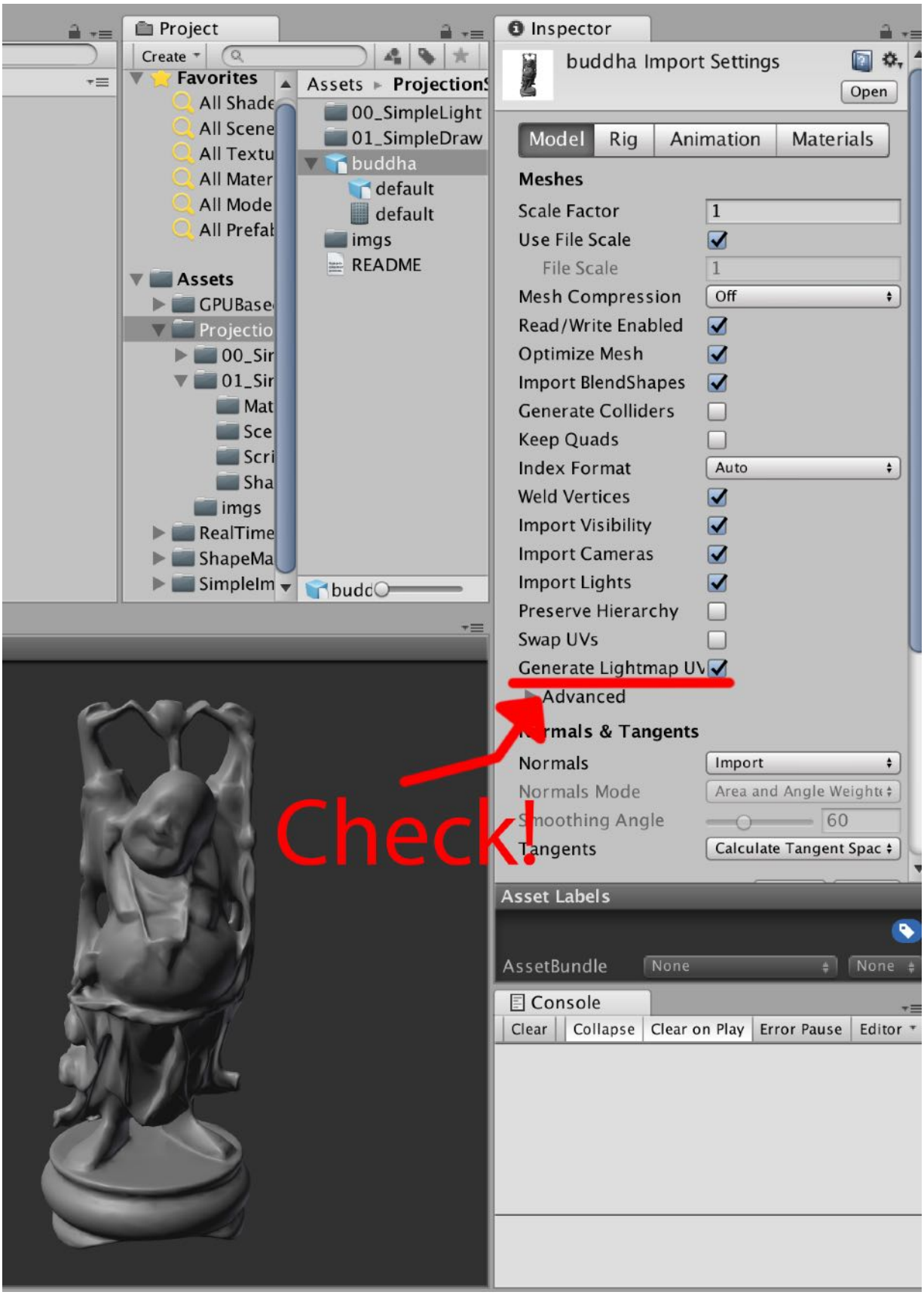
Project
Create
Favorites
All Shade
All Scene
All Textu
All Mater
All Mode
All Prefab
Assets
GPUBase
Projectio
00_Sir
01_Sir
Mat
Sce
Scri
Sha
imgs
RealTime
ShapeMa
SimpleIm
Assets ▸ Projection
00_SimpleLight
01_SimpleDraw
buddha
default
default
imgs
README
budd
Inspector
buddha Import Settings
Open
Model
Rig
Animation
Materials
Meshes
Scale Factor
1
Use File Scale
File Scale
1
Mesh Compression
Off
Read/Write Enabled
Optimize Mesh
Import BlendShapes
Generate Colliders
Keep Quads
Index Format
Auto
Weld Vertices
Import Visibility
Import Cameras
Import Lights
Preserve Hierarchy
Swap UVs
Generate Lightmap UV
Advanced
rmals & Tangents
Normals
Import
Normals Mode
Area and Angle Weighte
Smoothing Angle
60
Tangents
Calculate Tangent Spac
Check!
Asset Labels
AssetBundle
None
None
Console
Clear
Collapse
Clear on Play
Error Pause
Editor

図4.10: buddha Import Setting

If you check the "Generate Lightmap UVs" item in the model Import Setting, UVs for the lightmap will be generated. (`v.uv2 : TEXCOORD1`) Create a drawable RenderTexture for this Uv2 and draw it.

4.3.1 showUv2

See **00_showUv2.unity** for a sample scene .

In order to write to the texture that maps to mesh.uv2, we need to generate a texture that is expanded from the mesh to UV2. First, let's create a shader that expands the vertices of the mesh to the coordinates of UV2.

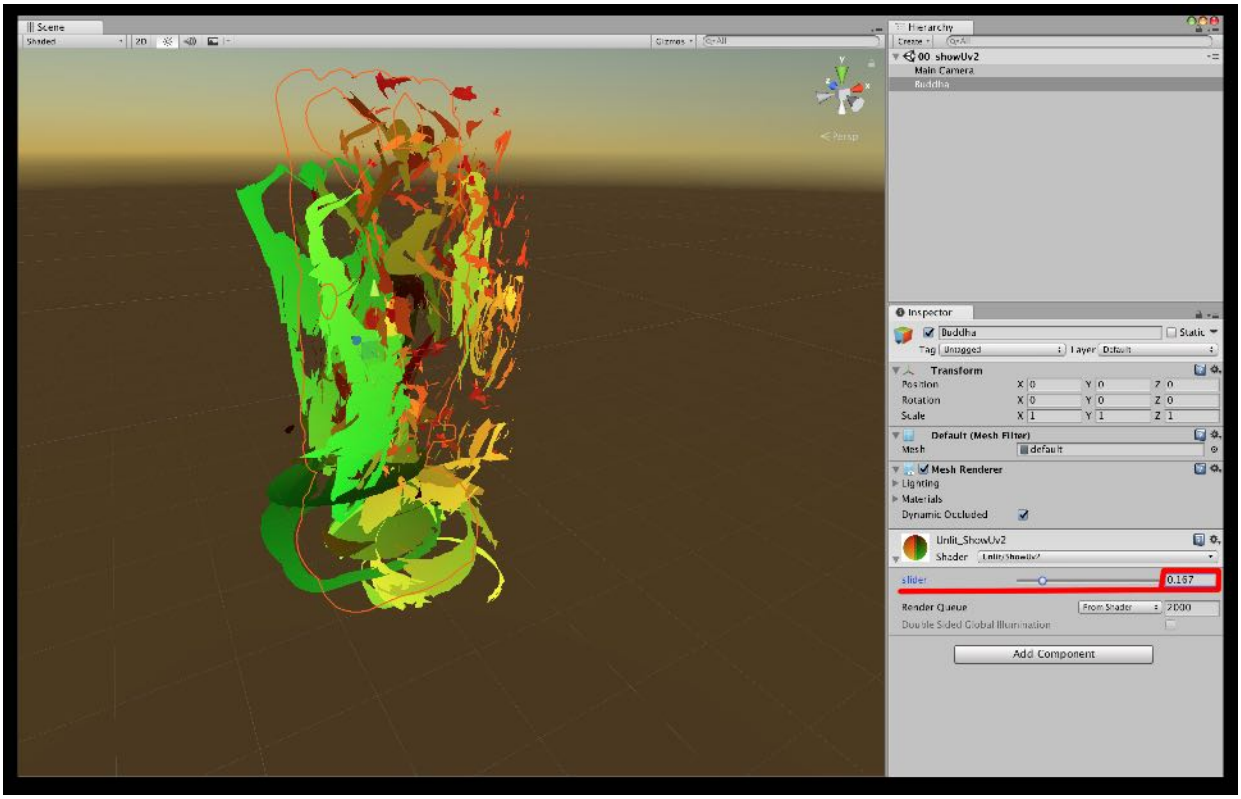

Figure 4.11: 00_showUv2.unity

Selecting a Buddha object in the scene and manipulating the material's "slider" parameter will change the object from its original shape to its Uv2 expanded shape. Coloring is, `uv2.xy`wo `color.rg`has been assigned to.

Listing 4.13: showUv2.shader

```
 1: float _T;
 2:
 3: v2f vert(appdata v)
 4: {
 5: #if UNITY_UV_STARTS_AT_TOP
 6: v.uv2.y = 1.0 - v.uv2.y;
 7: #endif
 8:     float4 pos0 = UnityObjectToClipPos(v.vertex);
 9: float4 pos1 = float4 (v.uv2 * 2.0 - 1.0, 0.0, 1.0);
10:
11:     v2f o;
12:     o.vertex = lerp(pos0, pos1, _T);
13: o.uv2 = v.uv2;
14:     o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
15:     o.normal = UnityObjectToWorldNormal(v.normal);
16:     return o;
17: }
18:
19: half4 frag(v2f i) : SV_Target
20: {
21:     return half4(i.uv2,0,1);
22: }
```

`float4 pos1 = float4(v.uv2*2.0 - 1.0, 0.0, 1.0);`The value of is the position expanded to Uv2 in the clip coordinate system. Listing 4.13

Since we are passing the value of `worldPos`and to `normal`the fragment shader, we will use this value to handle the lighting in the spotlight calculation.

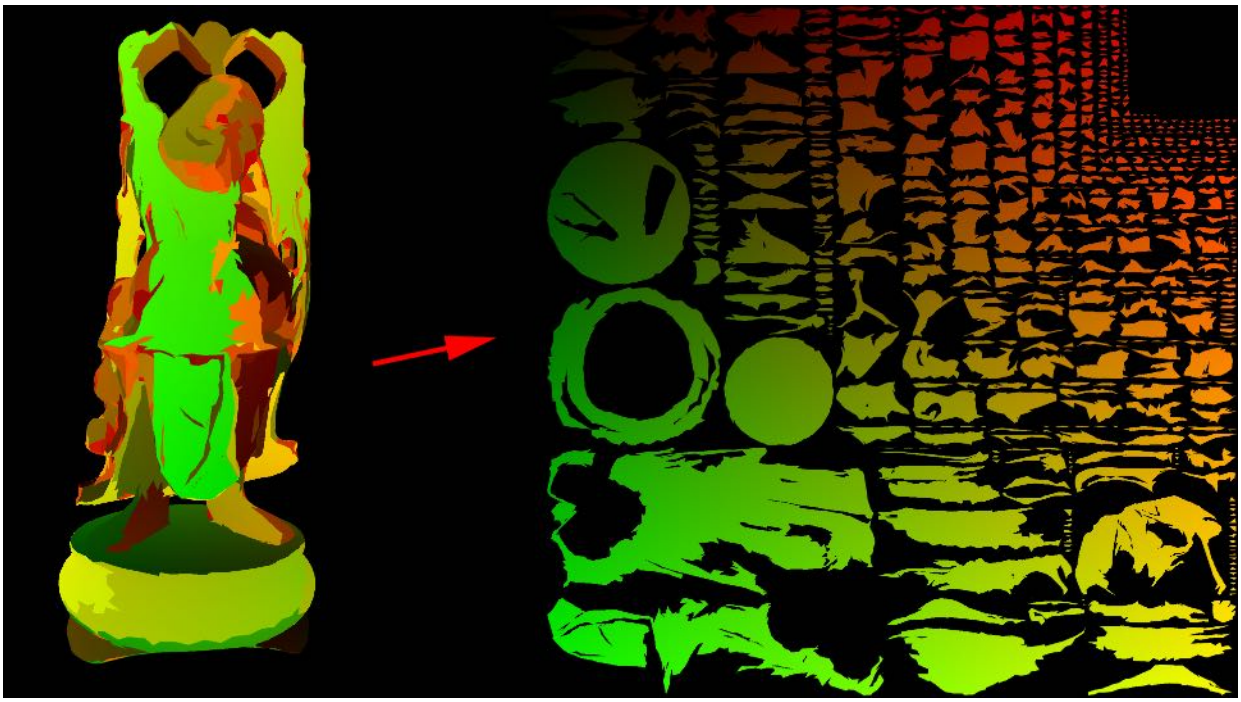

Figure 4.12: 00_showUv2.unity

You can generate textures expanded from mesh to Uv2!

4.3.2 ProjectionSpray

Now that we're ready, we'll implement the spray functionality. **See** the scene at **01_projectionSpray.unity** .

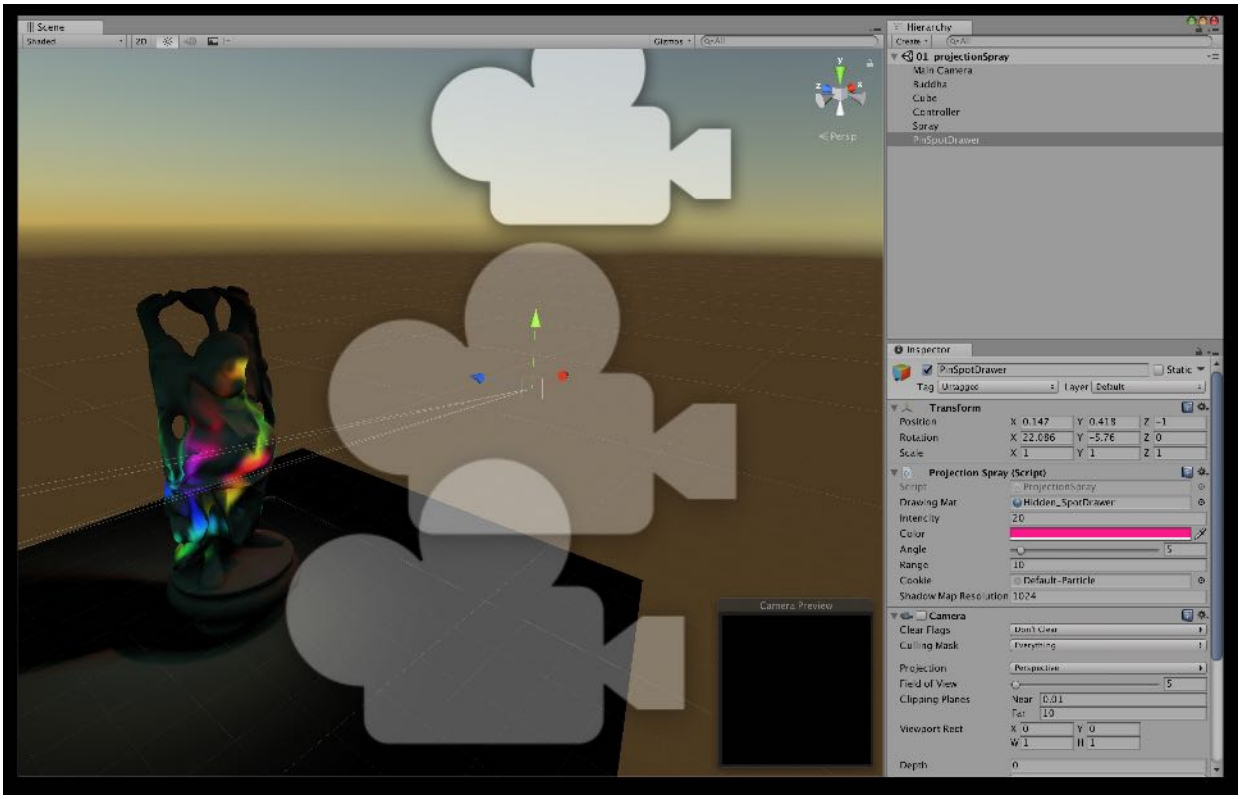

図4.13: 01_projectionSpray.unity

As you run this scene, the black Buddha objects will gradually become colored. Then, when you click on the screen, that part will be sprayed with a colorful color.

In terms of implementation content, it is an application of the self-made spotlight that has been implemented so far. The spotlight lighting calculation is not used for lighting `RenderTexture`as it is, but is used for updating. In this example, the texture you wrote is mapped to the one generated for the lightmap `mesh.uv2`.

`Drawable`Is a component attached to the object to be sprayed and is being drawn into the texture. `ProjectionSpray`The component `Material`sets properties for drawing on the texture, such as the position of the spray . As the processing flow , we call in `DrawableController`the `Update`function of `projectionSpray.Draw(drawable)`and draw on the texture.

ProjectionSpray.cs

- `Material drawMat`: Material for drawing
- `UpdateDrawingMat()`: Update material settings before drawing
- `Draw(Drawable drawable)`Pass : `drawMat`to `drawable.Draw(Material mat)`and draw.

Listing 4.14: projectionSpray.cs

```
 1: public class ProjectionSpray : MonoBehaviour {
 2:
 3:     public Material drawingMat;
 4:
 5:     public float intensity = 1f;
 6:     public Color color = Color.white;
 7:     [Range(0.01f, 90f)] public float angle = 30f;
 8:     public float range = 10f;
 9:     public Texture cookie;
10:     public int shadowMapResolution = 1024;
11:
12:     Shader depthRenderShader {
13:         get { return Shader.Find("Unlit/depthRender"); }
14:     }
15:
16:     new Camera camera{get{~~}}
17: Room _c;
18:     RenderTexture depthOutput;
19:
20:     public void UpdateDrawingMat()
21:     {
22:         var currentRt = RenderTexture.active;
23:         RenderTexture.active = depthOutput;
24:                         GL.Clear(true,  true,  Color.white  *
camera.farClipPlane);
25:         camera.fieldOfView = angle;
26:         camera.nearClipPlane = 0.01f;
27:         camera.farClipPlane = range;
28:         camera.Render();
29: // Update depth texture
30:         RenderTexture.active = currentRt;
31:
32:         var projMatrix = camera.projectionMatrix;
33:                                 var   worldToDrawerMatrix   =
transform.worldToLocalMatrix;
34:
35:                             drawingMat.SetVector("_DrawerPos",
transform.position);
36:                 drawingMat.SetFloat("_Emission",  intensity  *
```

```
Time.smoothDeltaTime);
37:         drawingMat.SetColor("_Color", color);
38:                 drawingMat.SetMatrix("_WorldToDrawerMatrix",
worldToDrawerMatrix);
39:         drawingMat.SetMatrix("_ProjMatrix", projMatrix);
40:         drawingMat.SetTexture("_Cookie", cookie);
41:         drawingMat.SetTexture("_DrawerDepth", depthOutput);
42: // The property name is different, but the information
passed is the same as the spotlight.
43:     }
44:
45:     public void Draw(Drawable drawable)
46:     {
47:         drawable.Draw(drawingMat);
48: // The drawing process itself is done with Drawable.
49: // Projection Spray has the Material to draw.
50:     }
51: }
```

Drawable.cs

The object to be drawn by spraying. It has a texture for drawing. `Start()`In the function `RenderTexture`creates a. It uses the classic Ping-pong Buffer.

Let's see the processing of the part that draws on the texture

Listing 4.15: Drawable.cs

```
 1: // This function is called from projectionSpray.Draw
(Drawable drawable)
 2:     public void Draw(Material drawingMat)
 3:     {
 4:         drawingMat.SetTexture("_MainTex", pingPongRts[0]);
 5: // Set the current state of the texture to be drawn as the
material.
 6:
 7:         var currentActive = RenderTexture.active;
 8:         RenderTexture.active = pingPongRts[1];
 9: // Set the texture to be drawn.
10:         GL.Clear(true, true, Color.clear);
11: // Clear the texture to be drawn.
12:         drawingMat.SetPass(0);
13:                                 Graphics.DrawMeshNow(mesh,
transform.localToWorldMatrix);
14: // Updated texture with target mesh and transform values to
```

```
draw.
15:         RenderTexture.active = currentActive;
16:
17:         Swap(pingPongRts);
18:
19:         if(fillCrack!=null)
20:         {
21: // This is a process to prevent cracks from forming at the
joints of Uv.
22:                 Graphics.Blit(pingPongRts[0], pingPongRts[1],
fillCrack);
23:             Swap(pingPongRts);
24:         }
25:
26:         Graphics.CopyTexture(pingPongRts[0], output);
27: // Copy the updated texture to output
28:     }
```

The point here is that we are updating `Graphics.DrawMeshNow(mesh, matrix)`using `RenderTexture`. Since the vertex shader of (Listing 4.15) expands `drawingMat`the vertices of `mesh.uv2`the mesh into the shape of, it is possible to update the texture after passing the vertex position, normal, and transform information of the mesh to the fragment shader. .. (Listing 4.16)

Listing 4.16: ProjectionSpray.shader

```
 1: v2f vert (appdata v)
 2: {
 3: v.uv2.y = 1.0 - v.uv2.y;
 4: // Invert and!
 5:
 6:     v2f o;
 7:     o.vertex = float4(v.uv2*2.0 - 1.0, 0.0, 1.0);
 8: // Same process as showUv2!
 9:     o.uv = v.uv2;
10:     o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
11:     o.normal = UnityObjectToWorldNormal(v.normal);
12:     return o;
13: }
14:
15: sampler2D _MainTex;
16:
17: uniform float4x4 _ProjMatrix, _WorldToDrawerMatrix;
18:
19: sampler2D _Cookie, _DrawerDepth;
20: half4 _DrawerPos, _Color;
```

```
21: half _Emission;
22:
23: half4 frag (v2f i) : SV_Target
24: {
25:     ///diffuse
26:     half3 to = i.worldPos - _DrawerPos.xyz;
27:     half3 dir = normalize(to);
28:     half dist = length(to);
29:      half atten = _Emission * dot(-dir, i.normal) / (dist *
dist);
30:
31:     ///spot cookie
32:     half4 drawerSpacePos = mul(
33:         _WorldToDrawerMatrix,
34:         half4(i.worldPos, 1.0)
35:     );
36:     half4 projPos = mul(_ProjMatrix, drawerSpacePos);
37: projPos.z * = -1;
38:     half2 drawerUv = projPos.xy / projPos.z;
39:     drawerUv = drawerUv * 0.5 + 0.5;
40:     half cookie = tex2D(_Cookie, drawerUv);
41:     cookie *=
42:         0<drawerUv.x && drawerUv.x<1 &&
43:         0<drawerUv.y && drawerUv.y<1 && 0<projPos.z;
44:
45:     ///shadow
46:     half drawerDepth = tex2D(_DrawerDepth, drawerUv).r;
47:      atten *= 1.0 - saturate(10 * abs(drawerSpacePos.z) - 10
* drawerDepth);
48: // So far, it's the same as spotlight processing!
49:
50:     i.uv.y = 1 - i.uv.y;
51:     half4 col = tex2D(_MainTex, i.uv);
52: // _MainTex is assigned drawable.pingPongRts [0]
53:     col.rgb = lerp(
54:         col.rgb,
55:         _Color.rgb,
56:         saturate(col.a * _Emission * atten * cookie)
57:     );
58: // This is the process of drawing!
59:  // Complementing the original texture to the drawn color
according to the calculated lighting intensity.
60:
61:     col.a = 1;
62:     return col;
63: // The value is output to drawable.pingPongRts [1]
64: }
```

You can now spray the 3D model. (Fig. 4.14)

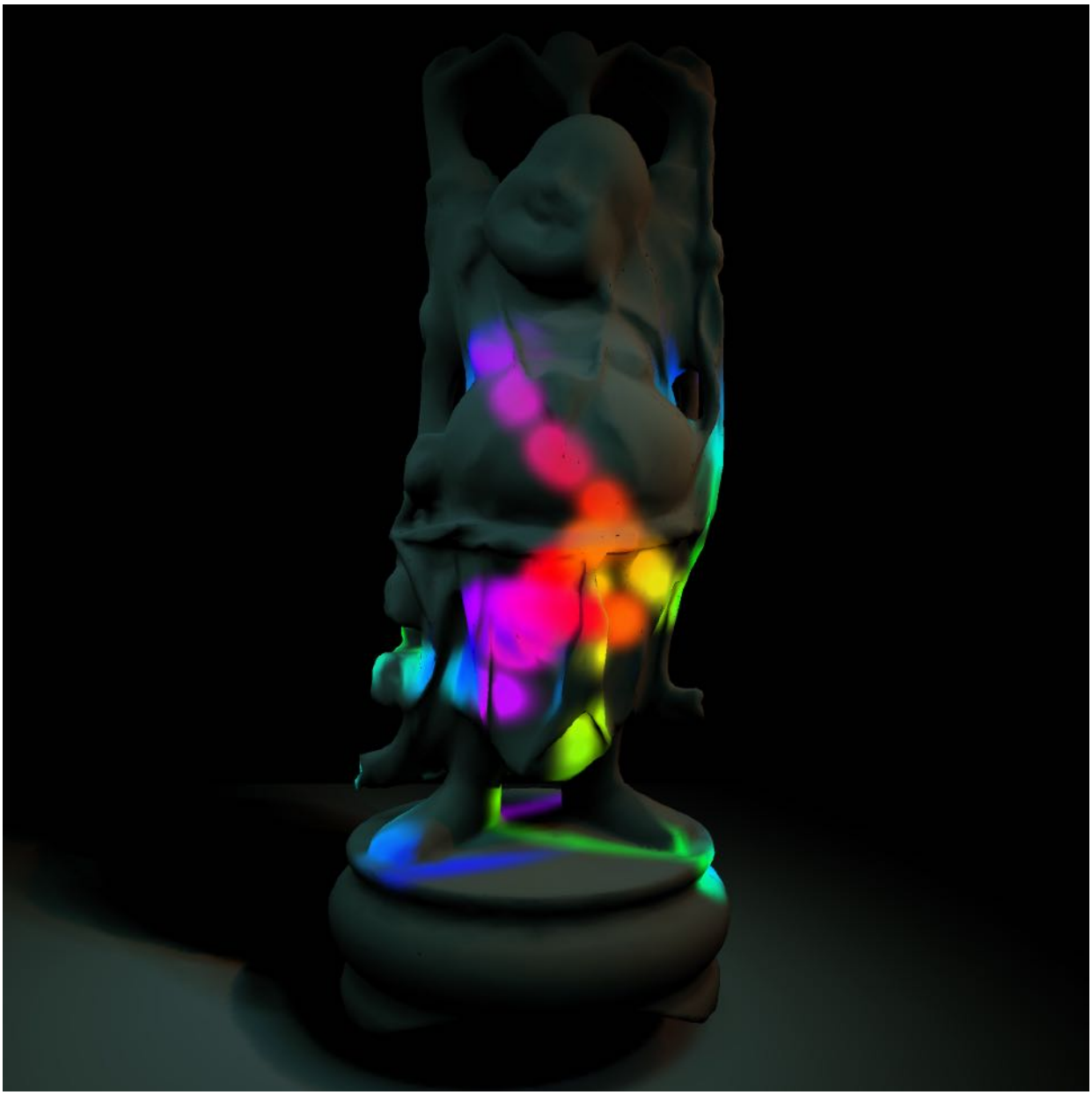

図4.14: 01_projectionSpray.unity

4.4 Summary

If you look at UnityCG.cginc, Lighting.cginc, etc., the built-in processing is written, and it will be a reference to implement various processing, so it is good to see it!

Chapter 5 Introduction to Procedural Noise

5.1 Introduction

In this chapter, we will explain the noise used in computer graphics. Noise was developed in the 1980s as a new method of image generation for texture mapping. Texture mapping, which attaches an image to an object to create its complexity, is a well-known technique in today's CG, but computers at that time had very limited storage space. Using image data for texture mapping was not compatible with the hardware. Therefore, a method for procedurally generating this noise pattern was devised. Naturally occurring substances and phenomena such as mountains, desert-like terrain, clouds, water surfaces, flames, marble, grain, rocks, crystals, and foam films have visual complexity and regular patterns. .. Noise can generate the best texture patterns for expressing such naturally occurring substances and phenomena, and has become an indispensable technique when procedurally wanting to generate graphics. Typical noise algorithms are **Ken Perlin** 's achievements, **Perlin Noise** and **Simplex Noise** . Here, as a stepping stone to many applications of noise, I would like to explain mainly the algorithms of these noises and the implementation by shaders.

The sample data in this chapter is from the Common Unity Sample Project.

Assets/**TheStudyOfProceduralNoise**

It is in. Please also refer to it.

5.2 What is noise?

The word noise means a noisy sound that can be translated as noise in the field of audio, and also in the field of video, it usually refers to general unnecessary information for the content to be processed or to show image

roughness. Also used. Noise in computer graphics is a function that takes an N-dimensional vector as an input and returns a scalar value (one-dimensional value) of a random pattern with the following characteristics.

- Continuously changing with respect to adjacent areas
- Features are statistically invariant with respect to rotation (even if a specific area is cut out and rotated, the features do not change) (= isotropic)
- Features are statistically invariant to movement (even if a specific area is cut out and moved, the features do not change)
- When viewed as a signal, the frequency band is limited (most energy is concentrated in a particular frequency spectrum)

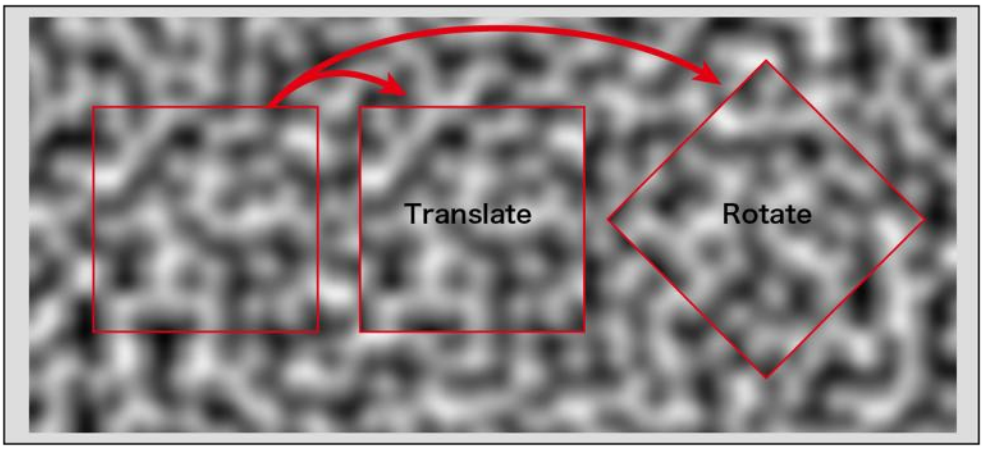

Figure 5.1: Noise characteristics

Noise can be used for the following purposes by receiving an N-dimensional vector as an input.

- Animation ・ ・ ・ 1D (time)
- Texture ・ ・ ・ 2D (UV coordinates of object)
- Animated texture ・ ・ ・ 3D (UV coordinates of object + time)
- Solid (3D) texture ・ ・ ・ 3D (local coordinates of object)
- Animated solid texture ・ ・ ・ 4D (local coordinates of object + time)

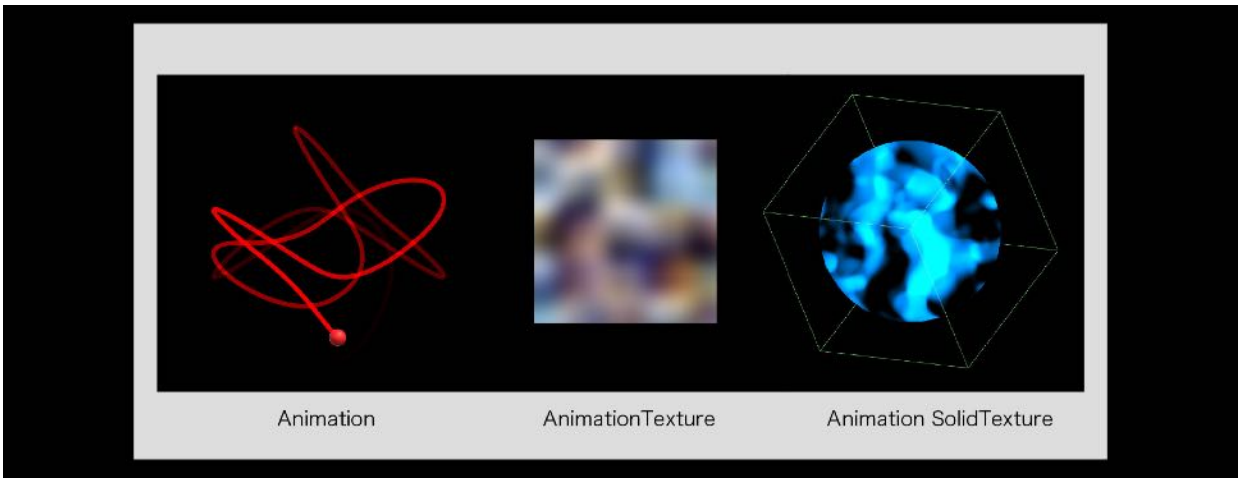

Figure 5.2: Noise application

5.3 Explanation of noise algorithm

We will explain the algorithms for **Value Noise** , **Perlin Noise** , **Improved Perlin Noise** , and **Simplex Noise** .

5.3.1 Value Noise

Although it does not strictly meet the conditions and accuracy of a noise function, we will introduce a noise algorithm called **Value Noise** , which is the easiest to implement and helps you understand noise.

algorithm

1. Define grid points at regular intervals on each axis in space
2. Find the value of a pseudo-random number for each grid point
3. Find the value of the point between each grid point by interpolation

Define grid

For two dimensions, define an evenly spaced grid on each of the x and y axes. The grid has a square shape, and at each of these grid points, the value of the pseudo-random number is calculated with reference to the coordinate

values of the grid points. In the case of 3D, a grid is defined at equal intervals on each of the x, y, and z axes, and the shape of the grid is a cube.

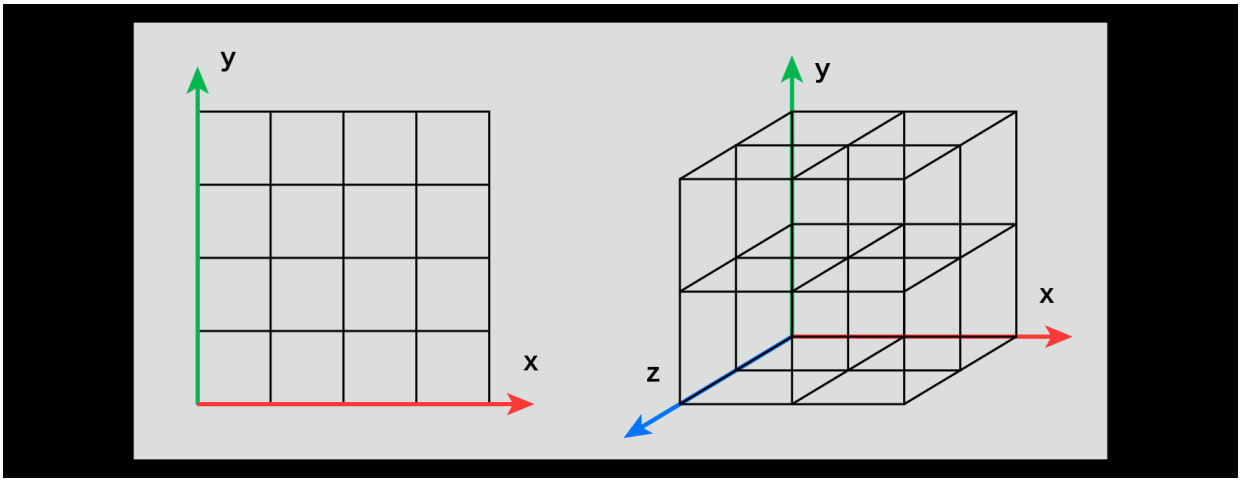

Figure 5.3: Lattice (2D), Lattice (3D)

Generation of pseudo-random numbers (Pseudo Random)

A random number is a sequence of numbers that are randomly arranged so that they have the same probability of appearing. There are also random numbers called true random numbers and pseudo-random numbers. For example, when rolling a dice, it is impossible to predict the next roll from the previous roll, and such a random number is a true random number. Is called. On the other hand, those with regularity and reproducibility are called **pseudo-random numbers (Pseudo Random)** . (When a computer generates a random number sequence, it is calculated by a deterministic calculation, so most of the generated random numbers can be said to be pseudo-random numbers.) When calculating noise, the same result can be obtained by using common parameters. Use the pseudo-random number that gives.

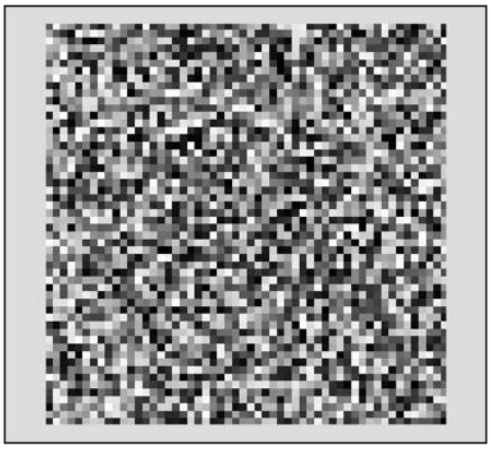

Figure 5.4: Pseudo-random numbers

By giving the coordinate values of each grid point to the argument of the function that generates this pseudo-random number, the value of the pseudo-random number unique to each grid point can be obtained.

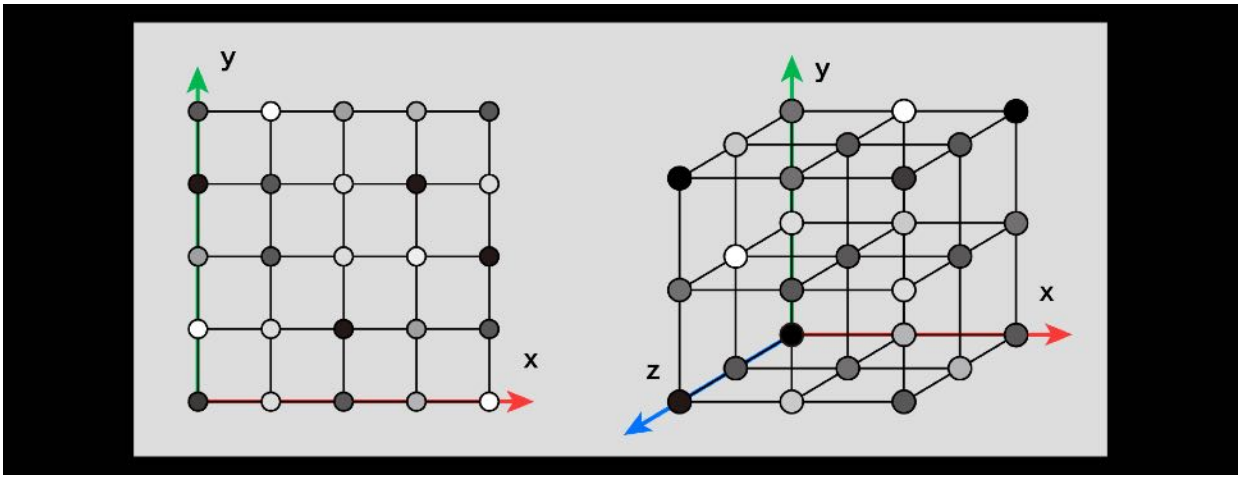

Figure 5.5: Pseudo-random numbers on each grid point

Interpolation

There are values A and B, and the value of P between them changes linearly from A to B, and finding that value approximately is called **linear interpolation** . This is the simplest interpolation method, but if you use it to find the value between the grid points, the change in the value will be sharp at the start and end points of the interpolation (near the grid point).

Therefore, we use a **cubic Hermitian curve** as the interpolation factor so that the values change smoothly .

```
f\left( t\right) =3t^{2}-2t^{3}
```

When this is changed `t=0`from `t=1`to, the value will be as shown in the lower right figure.

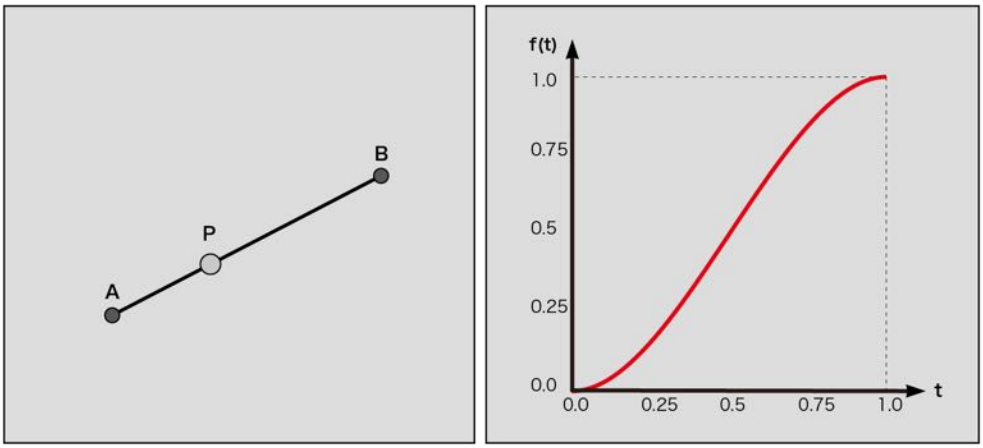

Figure 5.6: Linear interpolation in a two-dimensional plane (left), cubic Hermitian curve

* The cubic Hermitian curve is implemented as a **smoothstep** function in GLSL and HLSL .

This interpolation function is used to interpolate the values obtained at each grid point on each axis. In the case of 2D, first interpolate for x at both ends of the grid, then interpolate those values for the y-axis, and perform a total of 3 calculations. In the case of 3D, as shown in the figure below, 4 interpolations are performed for the z-axis, 2 for the y-axis, and 1 for the x-axis, for a total of 7 interpolations.

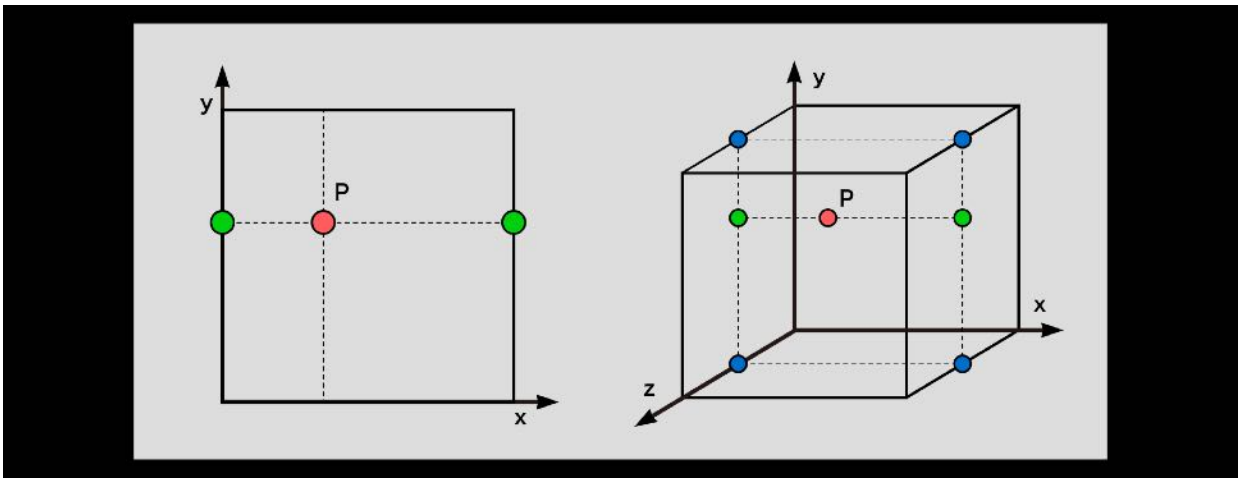

Figure 5.7: Interpolation (2D space), Interpolation (3D space)

Implementation

I will explain about 2D. Find the coordinates of each grid point.

`floor()`

The integer part is `floor()`calculated using a function. `floor()`Is a function that returns the smallest integer less than or equal to the input real number. When a real number of 1.0 or more is given to the input value, the values 1, 2, 3 ... are obtained and the same values are obtained at equal intervals, so this can be used as the coordinate value of the grid.

Use a `frac()`function to find the decimal part .

`frac()`

`frac()`Returns the decimal value of the given real number and takes a value greater than or equal to 0 and less than 1. This allows you to get the coordinate values inside each grid.

```
// Coordinate values of grid points
float2 i00 = i;
float2 i10 = i + float2 (1.0, 0.0);
float2 i01 = i + float2(0.0, 1.0);
float2 i11 = i + float2 (1.0, 1.0);
```

If you assign the coordinate values obtained above to the fragment colors R and G, you will get the following image. (For the integer part, since it can take a value of 1 or more, it is scaled so that the result does not exceed 1 for visualization.)

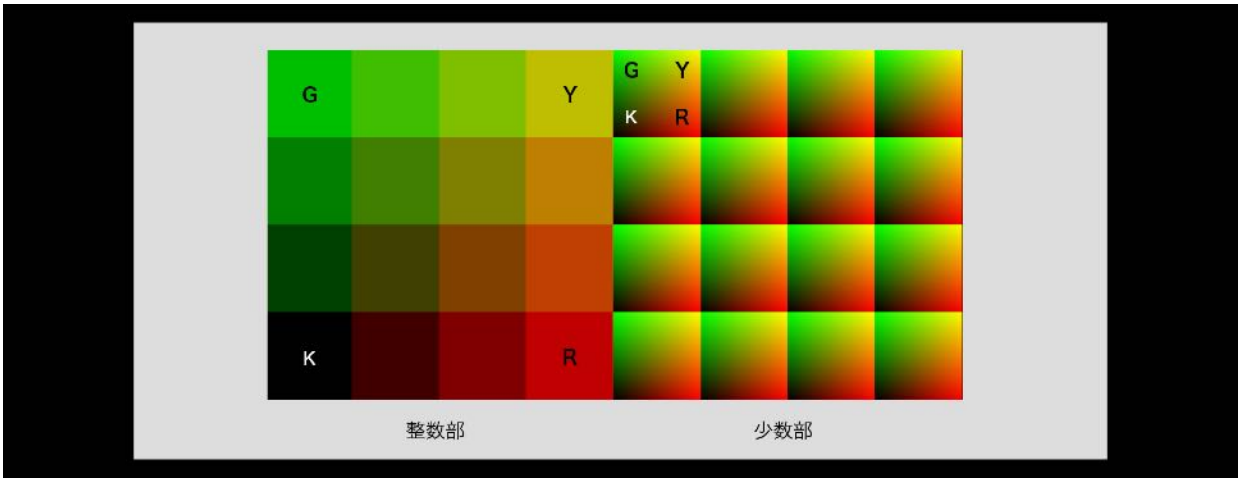

Figure 5.8: Integer and minority parts drawn as RG

Pseudo-random number generator

Searching the internet for the random function often returns this function as a result.

```
float rand(float2 co)
{
        return  frac(sin(dot(co.xy,  float2(12.9898,78.233)))  *
43758.5453);
}
```

Looking at the processing one by one, first, the input two-dimensional vector is rounded to one dimension by the inner product to make it easier to handle, and it is given as an argument of the sin function, multiplied by a large number, and the decimal part is obtained. So, this gives us regular and reproducible, but chaotically continuous values.

The origin of this function is uncertain,

https://stackoverflow.com/questions/12964279/whats-the-origin-of-this-glsl-rand-one-liner

According to the report, it originated from a treatise called "**On generating random numbers, with help of y = [(a + x) sin (bx)] mod 1**" published in 1998 .

Although it is simple and easy to handle, the cycle in which the same random number sequence appears is short, and if the texture has a large resolution, a pattern that can be visually confirmed occurs, so it is not a very good pseudo-random number.

```
// Pseudo-random value on the coordinates of the grid points
float n00 = pseudoRandom(i00);
float n10 = pseudoRandom(i10);
float n01 = pseudoRandom(i01);
float n11 = pseudoRandom(i11);
```

By giving the coordinate value (integer) of each grid point to the argument of the pseudo-random number, the noise value on each grid point is obtained.

Interpolation

```
// Interpolation function (3rd order Hermitian curve) = smoothstep
float2 interpolate(float2 t)
{
    return t * t * (3.0 - 2.0 * t);
}

// Find the interpolation factor
float2 u = interpolate(f);
// Interpolation of 2D grid
return  lerp(lerp(n00, n10, u.x), lerp(n01, n11, u.x), u.y);
```

`interpolate()`Calculate the interpolation factor with a predefined function. By using the decimal part of the grid as an argument, you can obtain a curve that changes smoothly near the start and end points of the grid.

`lerp()`Is a function that performs linear interpolation and stands for **Linear Interpolate** . It is possible to calculate the linearly interpolated value of the values given to the first and second arguments, and by substituting **u** obtained as the interpolation coefficient into the third argument, the values between the grids can be connected smoothly.

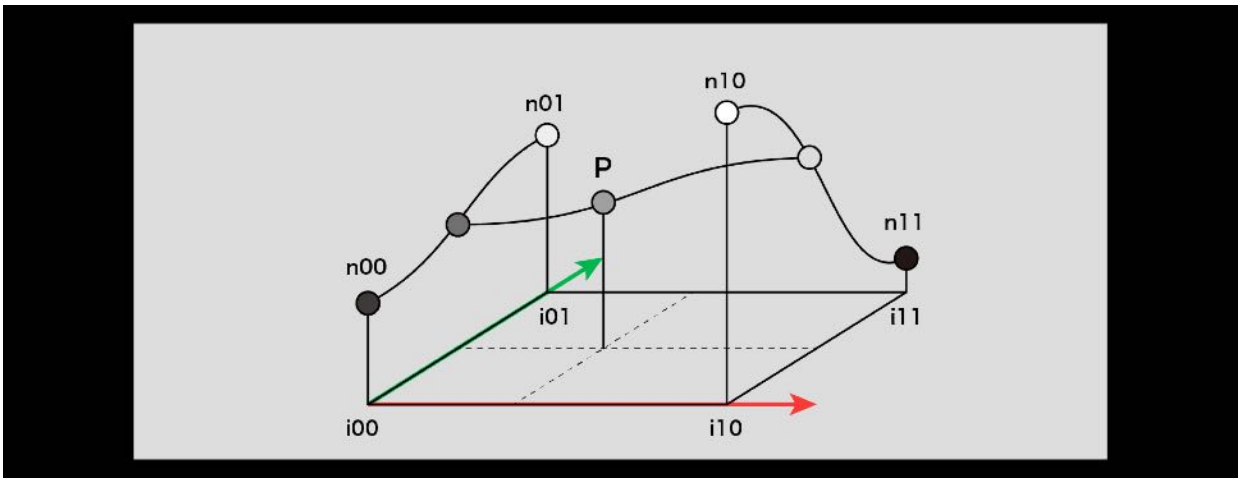

Figure 5.9: Interpolation of grid points (two-dimensional space)

result

In the sample project

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

When you open the scene, you can see the implementation result of **Value Noise** . For the code,

- Shaders/ProceduralNoise/**ValueNoise2D.cginc**
- Shaders/ProceduralNoise/**ValueNoise3D.cginc**
- Shaders/ProceduralNoise/**ValueNoise4D.cginc**

There is an implementation in.

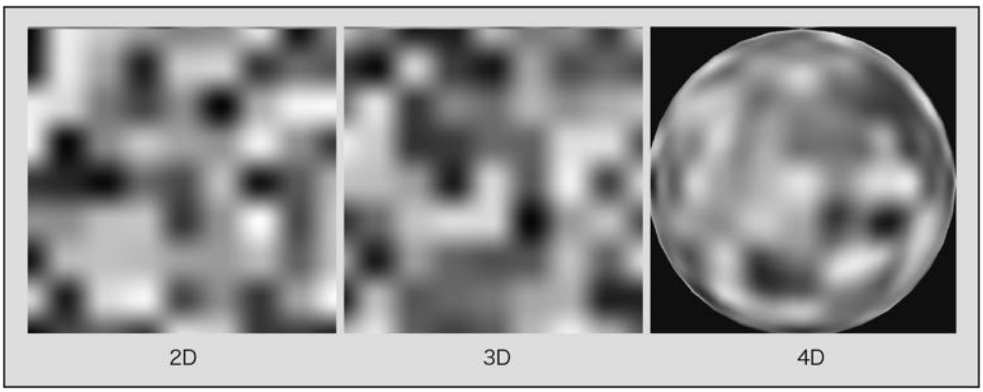

Figure 5.10: Value Noise (2D, 3D, 4D) Drawing result

If you look at the result image, you can see that the shape of the grid can be seen to some extent. As you can see, **Value Noise** is easy to implement, but its isotropic property that its characteristics are invariant when a certain area is rotated is not guaranteed, and it is not enough to be called noise. However, the process of "**interpolating the values of pseudo-random numbers obtained from regularly arranged grid points to obtain continuous and smooth values of all points in space**" performed in the implementation of **Value Noise** is , Has the basic algorithmic structure of the noise function.

5.3.2 Perlin Noise

Perlin Noise is a traditional and representative method of procedural noise and was developed by its name, **Ken Perlin** . Originally, it was produced in the experiment of texture generation for visual expression of the American science fiction movie "Tron" produced in 1982, which is known as the world's first movie that fully introduced computer graphics, and the result. Was published in a 1985 SIGGRAPH paper entitled "**An Image Synthesizer**" .

algorithm

1. Find the coordinates of the lattice
2. Find the Gradient on the grid points
3. Find the vector from each grid point to the point P in the grid
4. Calculate the inner product of the gradient obtained in 2 and the vector obtained in 3, and calculate the noise value on each grid point.
5. Interpolate the noise value of each grid point obtained in 4 with a cubic Hermitian curve.

Gradient

The difference from Value Noise is that the value of the grid point noise is not defined as a one-dimensional value, but as a **gradient** with a **slope (Gradient)** . Define a 2D gradient for 2D and a 3D gradient for 3D.

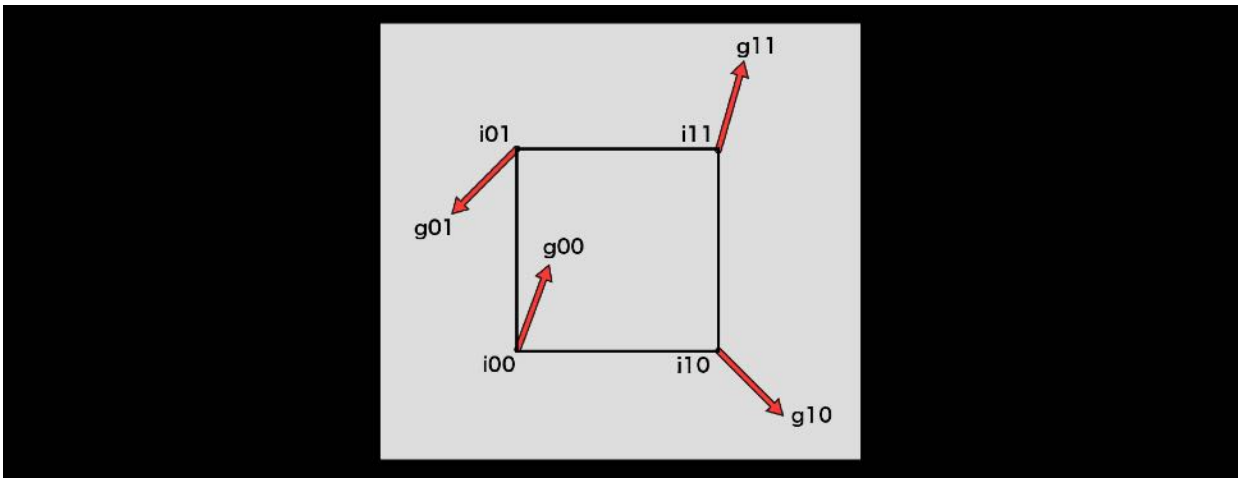

Figure 5.11: Perlin Noise Gradient Vector

Dot Product

Inner product is

```
\overrightarrow {a}\cdot \overrightarrow {b}=\left| a\right|
\left| b\right| \cos \theta
= \left( a.x\ast b.x\right) +\left( a.y\ast b.y\right)
```

In the vector operation defined in, the geometric meaning is the ratio of how much the two vectors are oriented in the same direction, and the values taken by the inner product are the **same direction → 1** , **orthogonal → 0** , and **vice versa. Orientation → -1** . In other words, finding the inner product of the gradient and the vector from each grid point toward the point P where you want to find the noise value in the grid means that if those vectors point in the same direction, the high noise value will be different. If you are facing the direction, a small value will be returned.

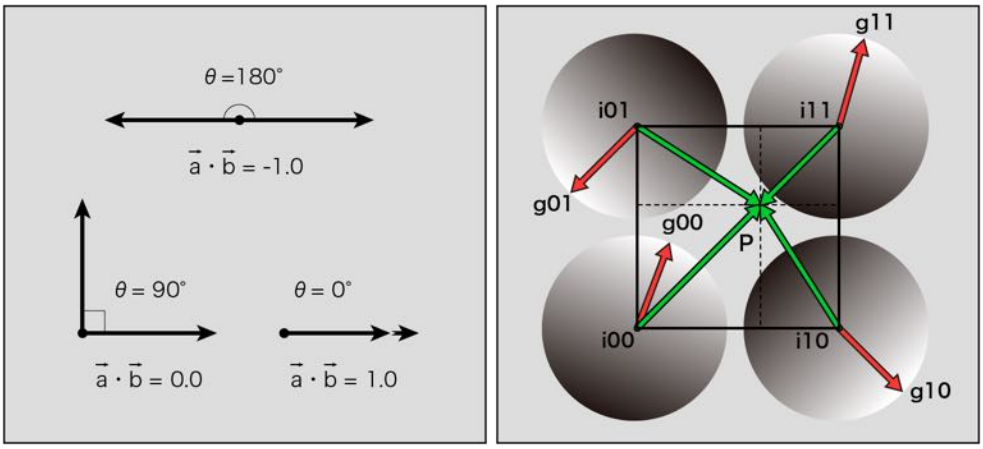

Figure 5.12: Dot Product (Left) Perlin Noise Gradient and Interpolation Vector (Right)

Interpolation

Here, the cubic Hermitian curve is used as a function for interpolation, but Ken Perlin later modified it to a cubic Hermitian curve. We 'll talk about that in the **Improved Perlin Noise** section.

Implementation

In the sample project

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

If you open the scene, you can see the implementation result of **Perlin Noise** . For the code,

- Shaders/ProceduralNoise/**OriginalPerlinNoise2D.cginc**
- Shaders/ProceduralNoise/**OriginalPerlinNoise3D.cginc**
- Shaders/ProceduralNoise/**OriginalPerlinNoise4D.cginc**

There is an implementation in.

I will post the implementation for 2D.

```
// Original Perlin Noise 2D
float originalPerlinNoise(float2 v)
{
    // Coordinates of the integer part of the grid
    float2 i = floor (v);
    // Coordinates of the decimal part of the grid
    float2 f = frac(v);

    // Coordinate values of the four corners of the grid
    float2 i00 = i;
    float2 i10 = i + float2 (1.0, 0.0);
    float2 i01 = i + float2(0.0, 1.0);
    float2 i11 = i + float2 (1.0, 1.0);

    // Vectors from each grid point inside the grid
    float2 p00 = f;
    float2 p10 = f - float2(1.0, 0.0);
    float2 p01 = f - float2(0.0, 1.0);
    float2 p11 = f - float2(1.0, 1.0);

    // Gradient of each grid point
    float2 g00 = pseudoRandom(i00);
    float2 g10 = pseudoRandom(i10);
    float2 g01 = pseudoRandom(i01);
    float2 g11 = pseudoRandom(i11);

    // Normalization (set the magnitude of the vector to 1)
    g00 = normalize(g00);
    g10 = normalize(g10);
    g01 = normalize(g01);
    g11 = normalize(g11);

    // Calculate the noise value at each grid point
    float n00 = dot(g00, p00);
    float n10 = dot(g10, p10);
    float n01 = dot(g01, p01);
    float n11 = dot(g11, p11);

    // Interpolation
    float2 u_xy = interpolate(f.xy);
      float2  n_x   =  lerp(float2(n00,  n01),  float2(n10,  n11),
u_xy.x);
    float n_xy = lerp(n_x.x, n_x.y, u_xy.y);
    return n_xy;
}
```

result

There is no unnatural grid shape as seen in **Value Noise** , and isotropic noise is obtained. **Perlin Noise** is also called **Gradient Noise** because it uses a gradient as **opposed** to **Value Noise** .

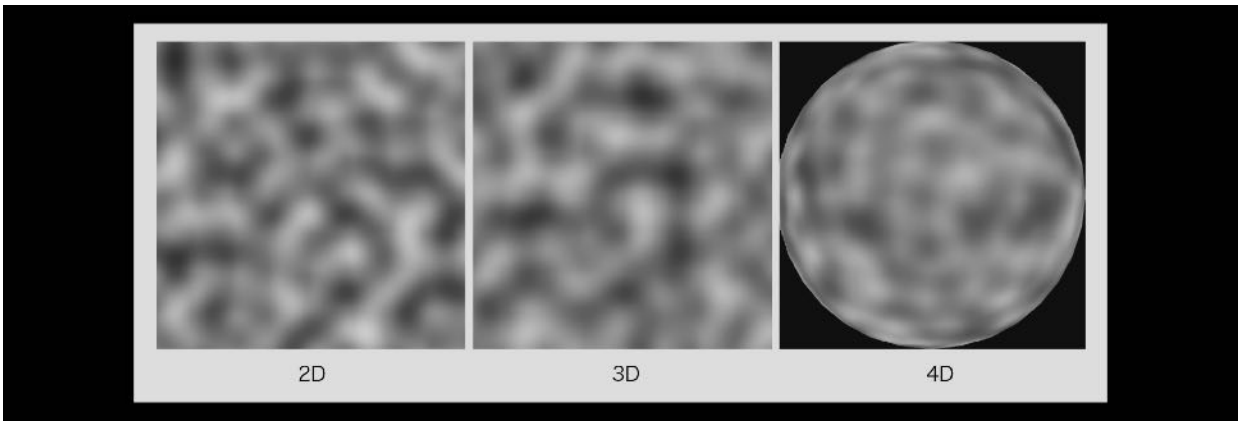

Figure 5.13: Perlin Noise (2D, 3D, 4D) results

5.3.3 Improved Perlin Noise

Improved Perlin Noise was announced in 2001 by Ken Perlin as an improvement over the shortcomings of **Perlin Noise** . More details can be found here.

http://mrl.nyu.edu/~perlin/paper445.pdf

Currently, most **Perlin Noise** is implemented based on this **Improved Perlin Noise** .

There are two main improvements Ken Perlin has made:

1. Interpolation function for interpolating gradients between grids
2. Gradient calculation method

Interpolation function for interpolating gradients between grids

For Hermite curve interpolation, the original of the **Perlin Noise** in **cubic Hermite curve** was used. However, if there is in this third-order equation, (when the differential to a result obtained can be further differentiated, that differentiates this) second-order differential `6-12t`is `t=0`, `t=1`when `0`you do

not take. Differentiating the curve gives the slope of the tangent. Another derivative gives that curvature, which is non-zero means there is a slight change. As a result, when used as a normal for bump mapping, adjacent grids and values are not exactly continuous, resulting in visual artifacts.

It is a comparison figure.

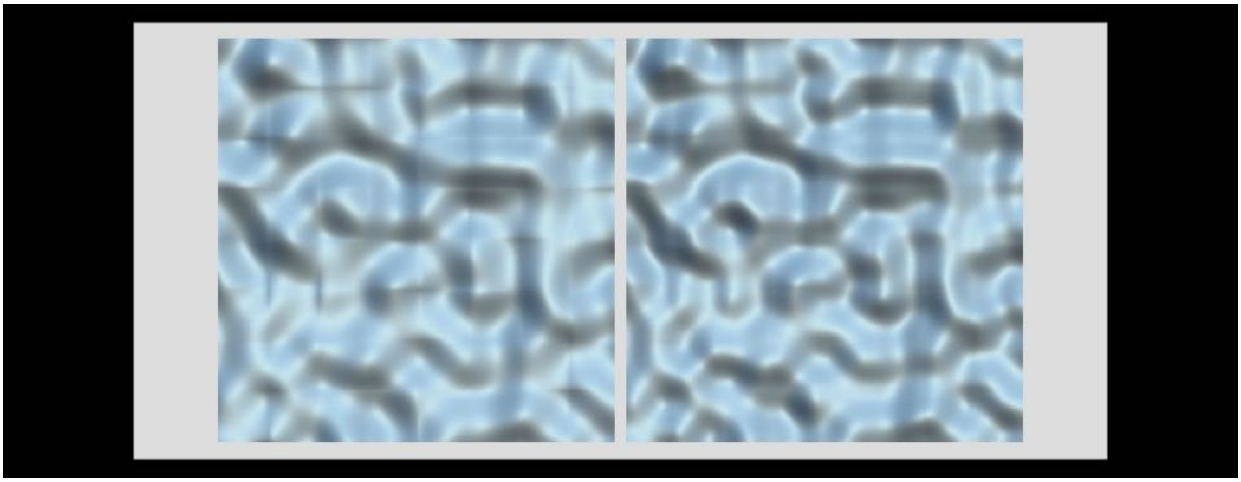

Figure 5.14: Interpolation with a cubic Hermitian curve (left) Interpolation with a fifth-order Hermitian curve (right)

Sample project

TheStudyOfProceduralNoise/Scenes/**CompareBumpmap**

You can see this by opening the scene.

Looking at the figure, the person who interpolated by the **cubic Hermitian curve on the** left shows a visually unnatural normal discontinuity at the boundary of the lattice. To avoid this, use the following **fifth-order Hermitian curve** .

```
f\left( t\right) =6t^{5}-15t^{4}+10t^{3}
```

Each curve diagram is shown. ① is a **cubic Hermitian curve** and ② is a **5th order Hermitian curve** .

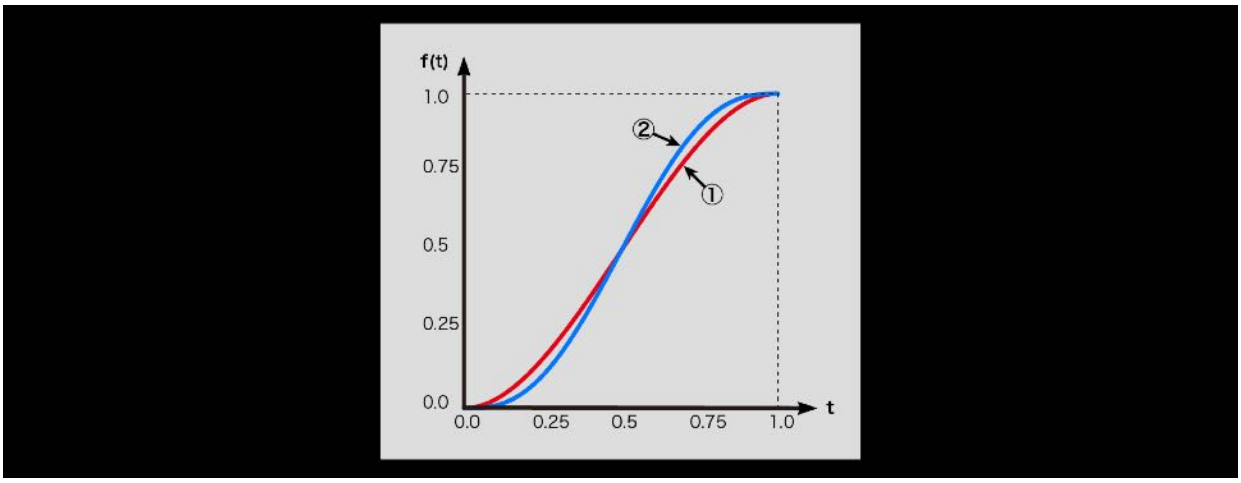

Figure 5.15: 3rd and 5th order Hermitian curves

`t=0`, `t=1`You can see that you have a smooth change around. Since both the 1st derivative and the 2nd derivative are at `t=0`or `t=1`at times `0`, continuity is maintained.

Gradient calculation

Think about 3D. The gradient G is evenly distributed in a spherical shape, but the cubic lattice is short about its axis, long about its diagonal, and has a directional bias in itself. If the gradient is close to parallel to the axis, aligning it with the ones in close proximity can result in unusually high values in those areas due to the close distance, which can result in a spotty noise distribution. In order to remove this gradient bias, we will limit it to the following 12 vectors, with those parallel to the axes and those on the diagonal removed.

```
(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),
(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)
```

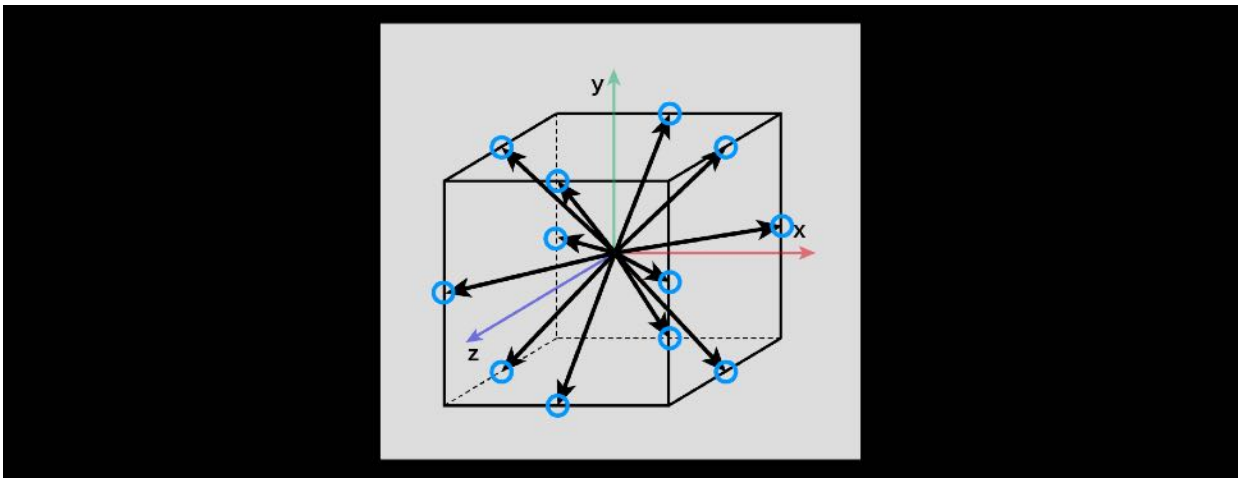

Figure 5.16: Improved Perlin Noise Gradient (3D)

From a cognitive psychological point of view, Ken Perlin states that in reality, the point P in the grid gives enough randomness, and the gradient G does not have to be random in all directions. .. In addition, for example, `(1, 1, 0)`the `(x, y, z)`inner product of, simply `x + y`can be calculated as, to simplify the inner product calculation to be performed later, you can avoid a lot of multiplication. This removes 24 multiplications from the calculation and keeps the calculation cost down.

Implementation and results

In the sample project

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

If you open the scene, you can see the implementation result of **Improved Perlin Noise** . For the code,

- Shaders/ProceduralNoise/**ClassicPerlinNoise2D.cginc**
- Shaders/ProceduralNoise/**ClassicPerlinNoise3D.cginc**
- Shaders/ProceduralNoise/**ClassicPerlinNoise4D.cginc**

This **improved Perlin Noise** implementation is based on the one published **in the** paper **"Effecient computational noise in GLSL"** , which will also be introduced in the next **Simplex Noise** . (Here it's called **Classic Perlin**

Noise , so it's a bit confusing, but I'm using that name.) This implementation is different from what Ken Perlin described in the paper for gradient calculations, but it gives quite similar results.

You can check the original implementation of Ken Perlin from the URL below.

http://mrl.nyu.edu/~perlin/noise/

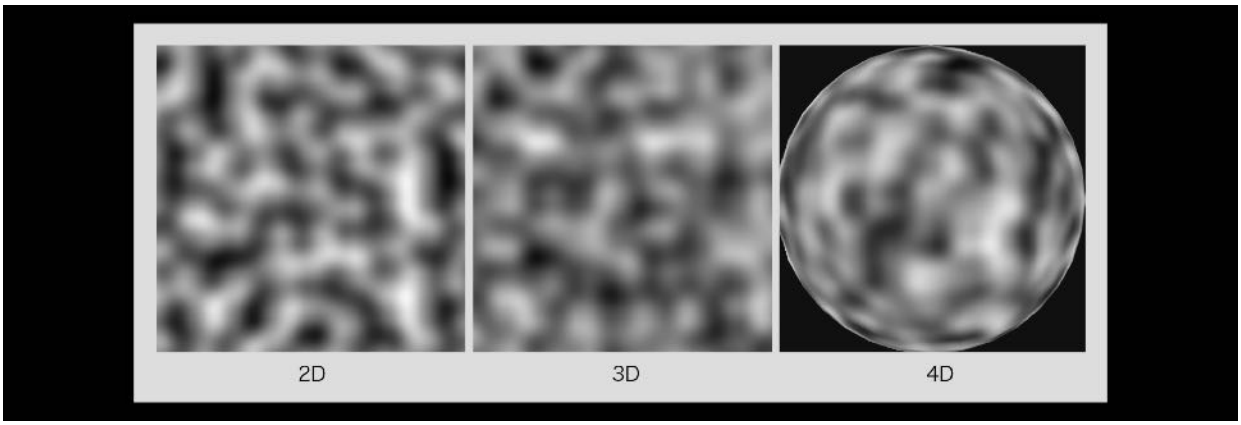

図5.17: Improved Perlin Noise（2D, 3D, 4D）

The figure below compares the noise gradient with the results. The left is the original **Perlin Noise** and the right is the **Improved Perlin Noise** .

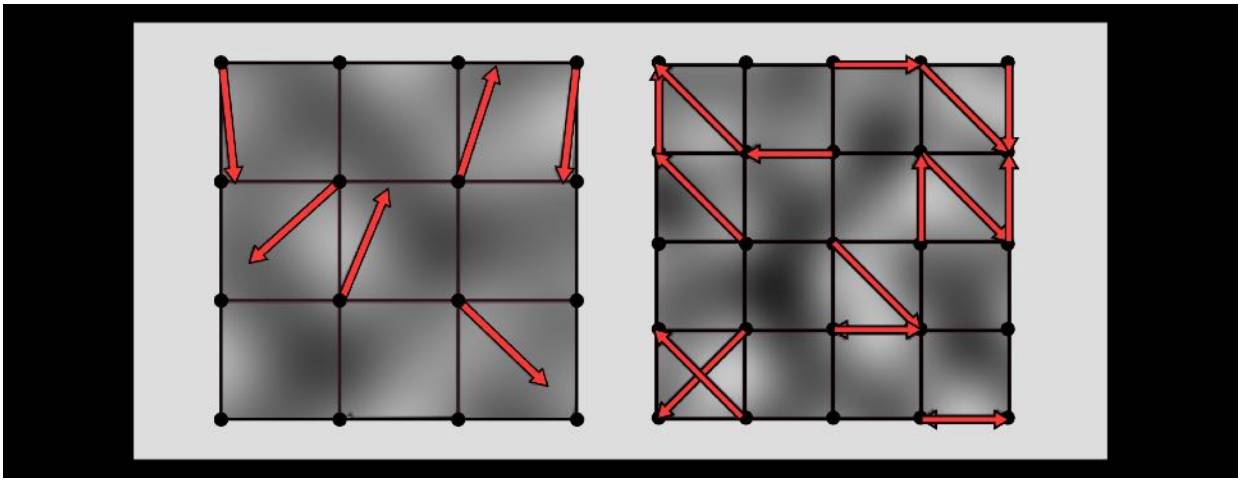

Figure 5.18: Perlin Noise, Improved Perlin Noise Gradients and Results Comparison

5.3.4 Simplex Noise

Simplex Noise was introduced by Ken Perlin in 2001 as a better algorithm than traditional **Perlin Noise** .

Simplex Noise has the following advantages over traditional **Perlin Noise** .

- The calculation complexity is low and the number of multiplications is small.
- **Simplex Noise** is O (N) where the computational load increases less and **Perlin Noise** is in the computational order of O (2 ^ {N}) when the noise dimension is increased to 4D, 5D or higher. ^ {2}) is enough
- No visual artifacts caused by directional bias of the gradient vector
- There is a continuous gradient with less computational load
- Easy to implement with hardware (shader)

Here, **"Simplex Noise Demystify"**

http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

I will explain based on the contents of.

algorithm

1. Define a grid by simplex
2. Calculate which unit has the point P for which the noise value is to be calculated.
3. Calculate the gradient at a single corner
4. Calculate the noise value at point P from the gradient values at the corners around each unit

Simplex grid

Simplex is called a simple substance in the topology of mathematics. A simple substance is the smallest unit that makes a figure. A 0-dimensional simplex is a **point** , a 1-dimensional simplex is a **line segment** , a 2-dimensional simplex is a **triangle** , a 3-dimensional simplex is a **tetrahedron** , and a 4-dimensional simplex is a **5-cell** .

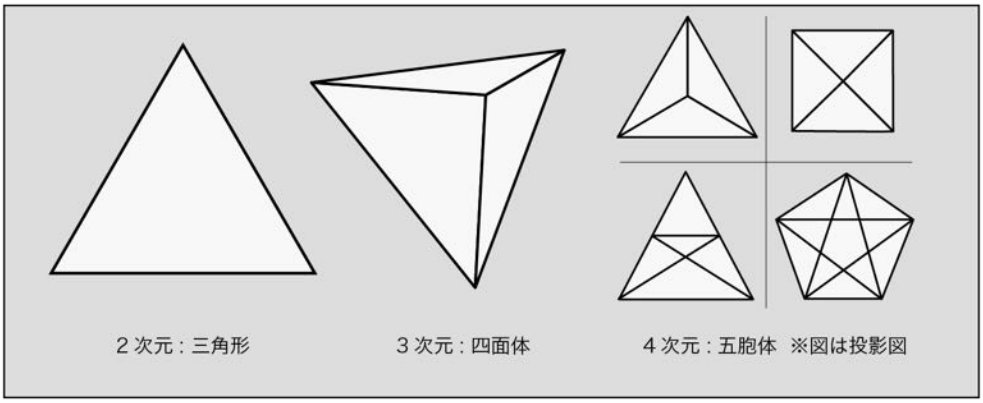

Figure 5.19: Simple substance in each dimension

Perlin Noise used a square grid for 2D and a cubic grid for 3D, but **Simplex Noise** uses this simple substance for the grid.

In one dimension, the simplest shape that fills the space is evenly spaced lines. In two dimensions, the simplest shape that fills the space is a triangle.

Two of the tiles made up of these triangles can be thought of as crushed squares along their main diagonal.

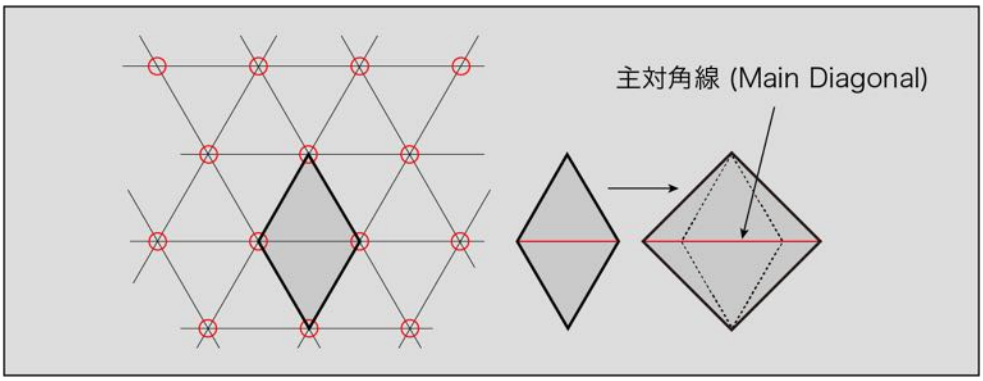

Figure 5.20: Two-dimensional simple substance grid

In three dimensions, the single shape is a slightly distorted tetrahedron. These six tetrahedra form a cube that is crushed along the main diagonal.

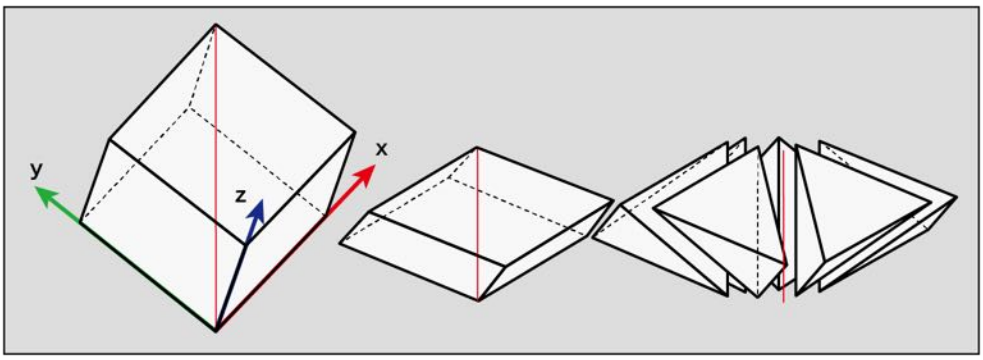

Figure 5.21: Three-dimensional simple substance grid

In 4D, single shapes are very difficult to visualize. Its single shape has five corners, and these 24 shapes form a four-dimensional hypercube that collapses along the main diagonal.

N-dimensional elemental shapes have **N + 1** corners, and **N! (3! Is 3 × 2 × 1 = 6)** shapes fill the N-dimensional hypercube collapsed along the main diagonal. It can be said that.

The advantage of using a simple substance shape for a grid is that you can define a grid with as few angles as possible with respect to the dimension, so when finding the values of points inside the grid, you will interpolate from the values of the surrounding grid points. It is in a place where the number of calculations can be suppressed. The N-dimensional hypercube has 2 ^ {N} corners, while the N-dimensional elemental shape has only N + 1 corners.

When trying to find higher dimensional noise values, traditional **Perlin Noise** requires O (2 ^ {N}) the complexity of the calculations at each corner of the hypercube and the amount of interpolation for each principal axis .) It's a problem and quickly becomes awkward. On the other hand, with **Simplex Noise** , the number of vertices of the simplex shape with respect to the dimension is small, so the amount of calculation is limited to O (N ^ {2}) .

Determining which unit has the point P for which the noise value is to be calculated.

With Perlin Noise, the integer part of the coordinates `floor()`could be used to calculate which grid the point P you want to find is in. For Simplex Noise,

follow the two steps below.

1. By distorting the input coordinate space along the main diagonal and looking at the integer part of the coordinates of each axis, it is possible to determine which single unit it belongs to.
2. By comparing the magnitude of the distance from the origin of a single unit to the point P in each dimension, it is possible to determine which single unit belongs to.

For a visual understanding, let's look at a diagram of the two-dimensional case.

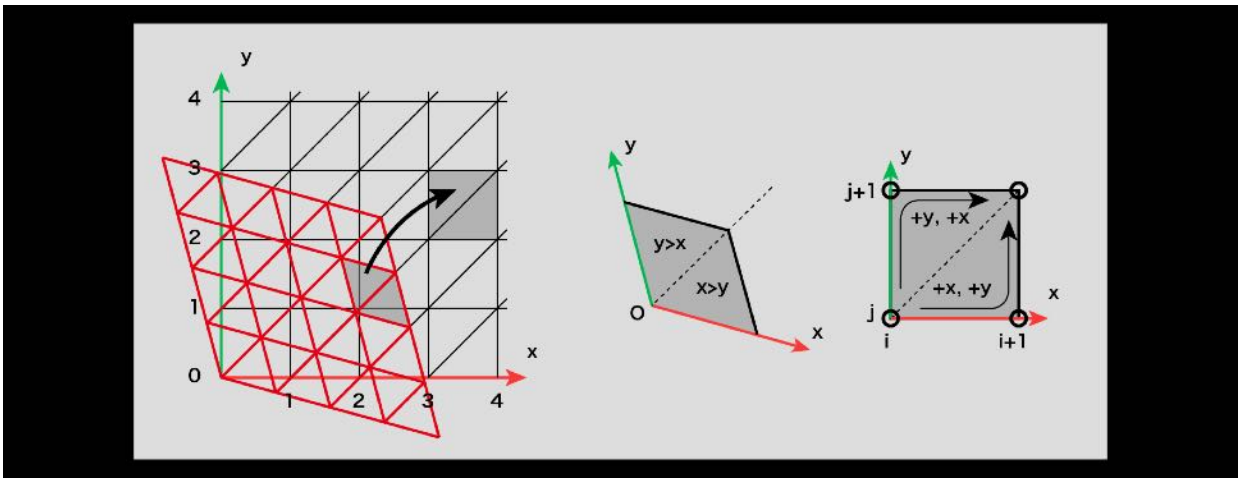

Figure 5.22: Deformation of a single grid in two dimensions

A single grid of two-dimensional triangles can be distorted into a grid of isosceles triangles by scaling. Two isosceles triangles form a quadrangle with one side length (a single unit refers to this quadrangle). `(x, y)`By looking at the integer part of the coordinates after moving , you can determine which single unit square the point P for which you want to find the noise value is. Also, by comparing the sizes of x and y from the origin of a single unit, it is possible to know which of the units is the single unit including the point P, and the coordinates of the three single points surrounding the point P are determined.

In the case of 3D, the 3D single lattice is regularly arranged by scaling along its main diagonal so that the 2D equilateral triangle single lattice can be transformed into an isosceles triangular lattice. It can be transformed into a

cubic grid. As in the case of two dimensions, you can determine which six units belong to a single unit by looking at the integer part of the coordinates of the moved point P. Furthermore, which unit of the unit belongs to can be determined by comparing the relative size of each axis from the origin of the unit.

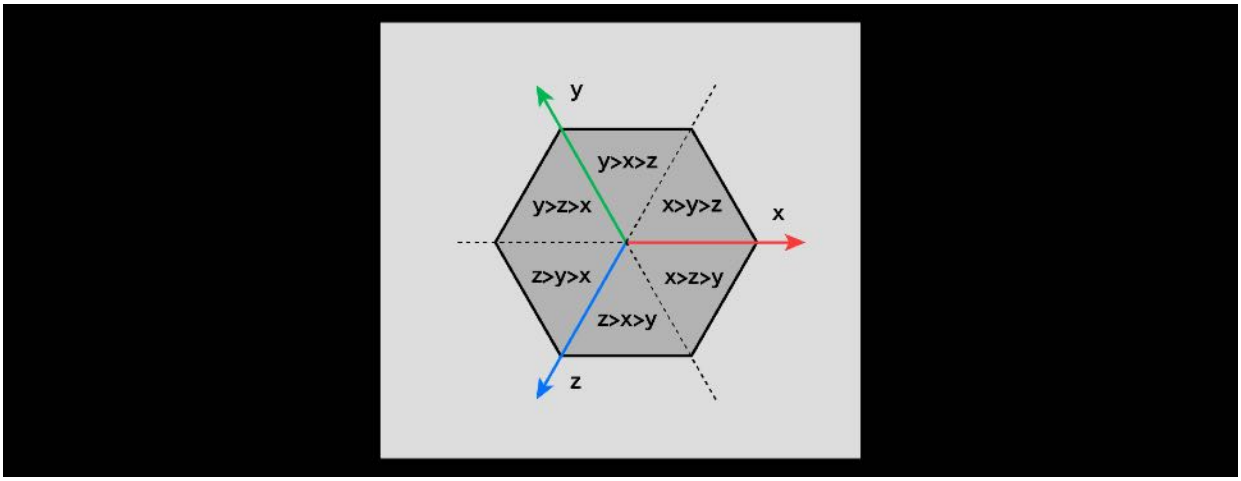

Figure 5.23: Rules for determining which single unit the point P belongs to in the 3D case

The figure above shows a cube formed by a three-dimensional unit along the main diagonal, and belongs to which unit depending on the size of the coordinate values of point P on the x, y, and z axes. It shows the rules of.

In the case of 4D, it is difficult to visualize, but it can be thought of as a rule in 2D and 3D. Coordinates of a four-dimensional hypercube that fills space There are `(x, y, z, w)`4! = 24 combinations of sizes for each axis, which are unique to each of the 24 units in the hypercube, and the point P belongs to which unit. Can be determined.

The figure below is a two-dimensional single grid visualized in fragment color.

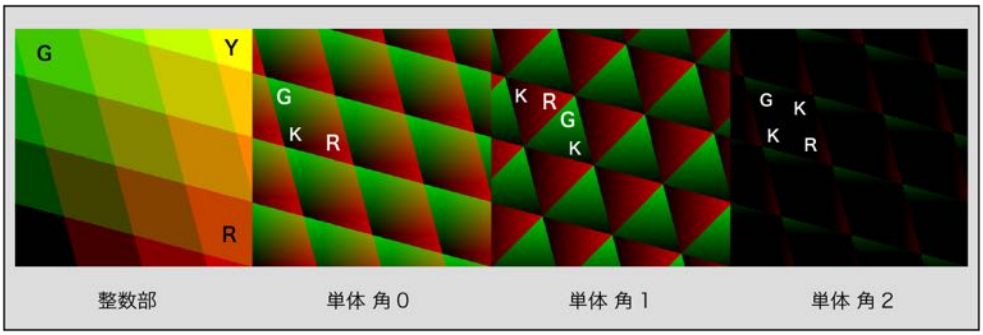

Figure 5.24: Single (2D) integer and minority parts

Transition from interpolation to sum

In conventional **Perlin Noise** , the values of points inside the grid are **calculated from** the values of the surrounding grid points by interpolation. However, with **Simplex Noise** , instead, the degree of influence of the values of the vertices of each simple substance is calculated by a simple sum calculation. Specifically, the **extrapolation** of the **slope of** each corner of a single unit and the product of **the functions that decay in a radial circle depending on the distance from each vertex** are added.

Think about two dimensions.

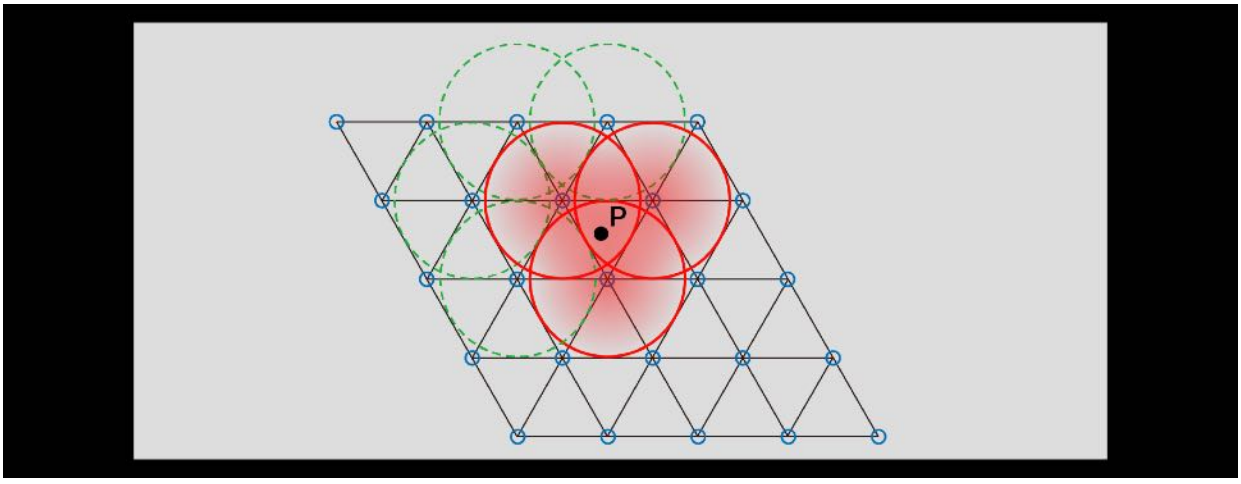

Figure 5.25: Radial circular decay function and its range of influence

The value of the point P inside a single unit only affects the values from each of the three vertices of the single unit that surrounds it. The values of the distant vertices have no effect because they decay to 0 before crossing the single boundary containing the point P. In this way, the noise value at point P can be calculated as the sum of the values of the three vertices and their degree of influence.

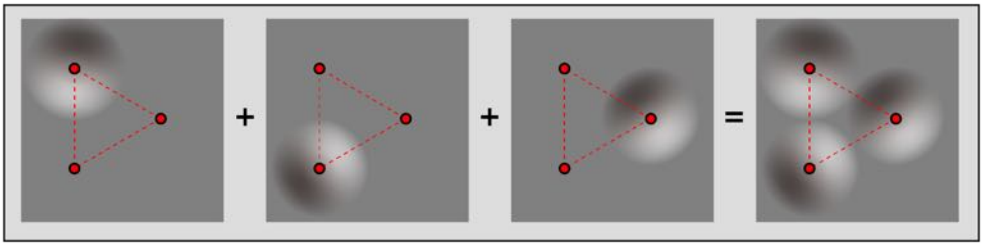

Figure 5.26: Contribution rate and sum of each vertex

Implementation

The implementation is **"Effecient computational noise in GLSL"** published by Ian McEwan, David Sheets, Stefan Gustavson and Mark Richardson in 2012.

https://pdfs.semanticscholar.org/8e58/ad9f2cc98d87d978f2bd85713d6c909c8a85.pdf

It is shown in the manner according to.

Currently, if you want to implement noise with a shader, it is an easy-to-use algorithm that is less hardware-dependent, efficient in calculation, and does not require reference to textures. (Probably)

As of April 2018, the source code is managed at https://github.com/stegu/webgl-noise/ . The original was here (https://github.com/ashima/webgl-noise), but Ashima Arts, which currently manages it, doesn't seem to be functioning as a company, so it was cloned by Stefan Gustavson.

There are three features of the implementation:

- Randomly arranged indexes for gradient vector calculation are calculated by polynomial instead of referring to the table.
- Use the cross-polytope geometry for gradient vector calculations
- Replace the single selection condition with Rank Ordering

Polynomial for index sorting of gradients

Previously announced noise implementations used tables containing pre-computed index values or bit-swapped hashes for index generation during gradient calculations, but both approaches are shaders. It cannot be said that it is suitable for implementation by. So, for index sorting,

```
\left( Ax^{2}+Bx\right) mod\ M
```

We are proposing a method to use a polynomial with a simple form. (Mod = modulo The number of remainders when a certain number is divided (remainder)) For example, \ left (6x ^ {2} + x \ right) mod \ 9 is `(0 1 2 3 4 5 6 7 8)` for `(0 7 8 3 1 2 6 4 5)` 0 to 8 inputs. Returns 9 unique numbers from 0 to 8.

To generate an index to distribute the gradient well enough, we need to sort at least hundreds of numbers, so we will choose \ left (34x ^ {2} + x \ right) mod \ 289 .

This permutation polynomial is a problem of the precision of variables in the shader language , and truncation occurs when 34x ^ {2} + x> 2 ^ {24} , or | x |> 702 in the integer region . .. So, in order to calculate the polynomial for sorting without the risk of overflow, we do a modulo 289 of x before doing the polynomial calculation to limit x to the range 0-288.

Specifically, it is implemented as follows.

```
// Find the remainder of 289
float3 mod289(float3 x)
{
    return x - floor(x * (1.0 / 289.0)) * 289.0;
}

// Sort by permutation polynomial
float3 trade-ins (float3 x)
```

```
{
    return fmod(((x * 34.0) + 1.0) * x, 289.0);
}
```

The treatise admits that in 2D and 3D, there is no problem, but in 4D, this polynomial has generated visual artifacts. For 4 dimensions, an index of 289 seems to be inadequate.

Use the cross-polytope geometry for gradient vector calculations

Traditional implementations used pseudo-random numbers for gradient calculations, referencing the table containing the indexes and performing bit operations to calculate the pre-calculated gradient indexes. Here, we use a **cross-polytope** for gradient calculations to get a more efficiently distributed gradient in different dimensions, which is more suitable for shader implementation . A cross-polytope is a generalized shape of a two-dimensional **square** , a three-dimensional **regular octahedron** , and a four-dimensional **regular six-cell body** in each dimension. Each dimension takes a geometric shape as shown in the figure below.

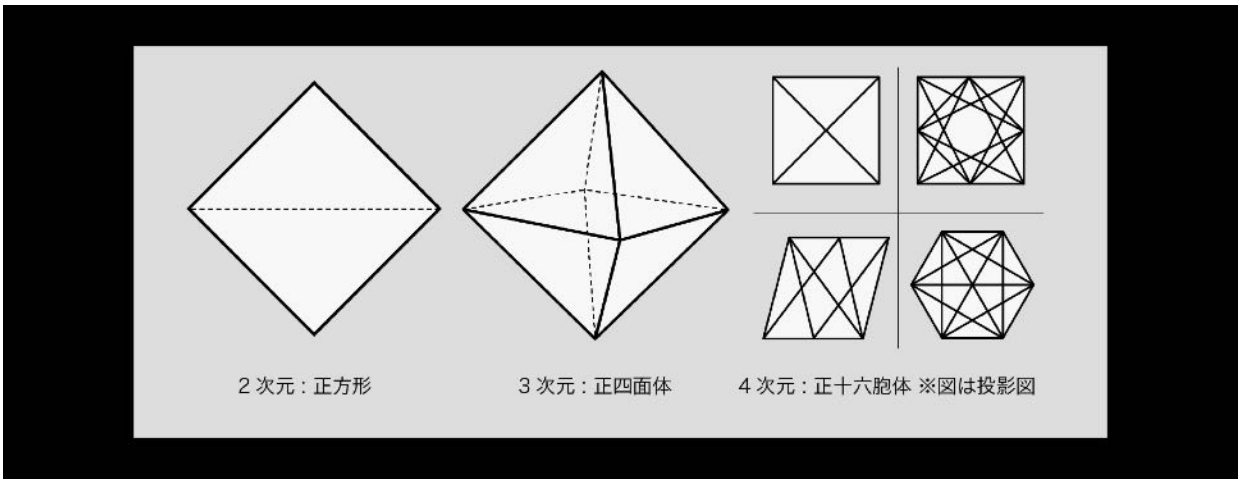

Figure 5.27: Cross-polytope in each dimension

Gradient vector at each dimension, if the two-dimensional **square** , if a three-dimensional **regular octahedral** , if four-dimensional (truncated portion) **16-cell** of **the surface** and distributed.

Each dimension and equation are as follows.

```
2-D: x0 ∈ [−2, 2], y = 1 − |x0|
if y > 0 then x = x0 else x = x0 − sign(x0)

3-D: x0, y0 ∈ [−1, 1], z = 1 − |x0| − |y0|
if z > 0 then x = x0, y = y0
else x = x0 − sign(x0), y = y0 − sign(y0)

4-D: x0, y0, z0 ∈ [−1, 1], w = 1.5 - | x0 | - | y0 | - | z0 |
if w > 0 then x = x0, y = y0, z = z0
else x = x0 − sign(x0), y = y0 − sign(y0), z = z0 − sign(z0)
```

Gradient normalization

Most **Perlin Noise** implementations used gradient vectors of equal magnitude. However, there is a difference in length between the shortest and longest vectors on the surface of the N-dimensional cross-polytope by the factor of \ sqrt {N} . This does not cause strong artifacts, but at higher dimensions the noise pattern becomes less isotropic without explicit normalization of this vector. Normalization is the process of aligning a vector to 1 by dividing the vector by the size of the vector. Assuming that the magnitude of the gradient vector is r , normalization can be achieved by multiplying the gradient vector by the inverse square root of r \ dfrac {1} {\ sqrt {r}} . Here, to improve performance, this inverse square root is approximately calculated using the Taylor expansion. The Taylor expansion is that in an infinitely differentiable function, if x is in the vicinity of a, it can be approximately calculated by the following formula.

```
\sum ^{\infty }_{n=0}\dfrac {f^{\left( n\right) }\left( a\right) }{n!}\left( x-a\right) ^{n}
```

Finding the first derivative of \ dfrac {1} {\ sqrt {a}}

```
\begin{array}{l}
f\left( a\right) =\dfrac {1}{\sqrt {a}}=a^{-\frac{1}{2}}\\
f'\left( a\right) =-\dfrac {1}{2}a^{-\frac{3}{2}}\\
\end{array}
```

Therefore, the approximate expression in the vicinity of a by Taylor expansion is as follows.

```
\sum ^{\infty }_{n=0}\dfrac {f^{\left( n\right) }\left( a\right) }{n!}\left( x-a\right) ^{n}
```

```
\begin{array}{l}
=a^{-\frac{1}{2}}-\frac{1}{2}a^{-\frac{3}{2}}\left( x-a\right)\\
=\frac{3}{2}a^{-\frac{1}{2}}-\frac{1}{2}a^{-\frac{3}{2}}x\\
\end{array}
```

Here, if a = 0.7 (I think it is because the length range of the gradient vector is 0.5 to 1.0), 1.79284291400159 --0.85373472095314 * x is obtained.

This is what the implementation looks like.

```
float3 taylorInvSqrt(float3 r)
{
    return 1.79284291400159 - 0.85373472095314 * r;
}
```

5.3.5 Implementation and results

In the sample project

TheStudyOfProceduralNoise/Scenes/**ShaderExampleList**

When you open the scene, you can see the implementation result of **Simplex Noise** . The implemented code is

- Shaders/ProceduralNoise/**SimplexNoise2D.cginc**
- Shaders/ProceduralNoise/**SimplexNoise3D.cginc**
- Shaders/ProceduralNoise/**SimplexNoise4D.cginc**

It is in.

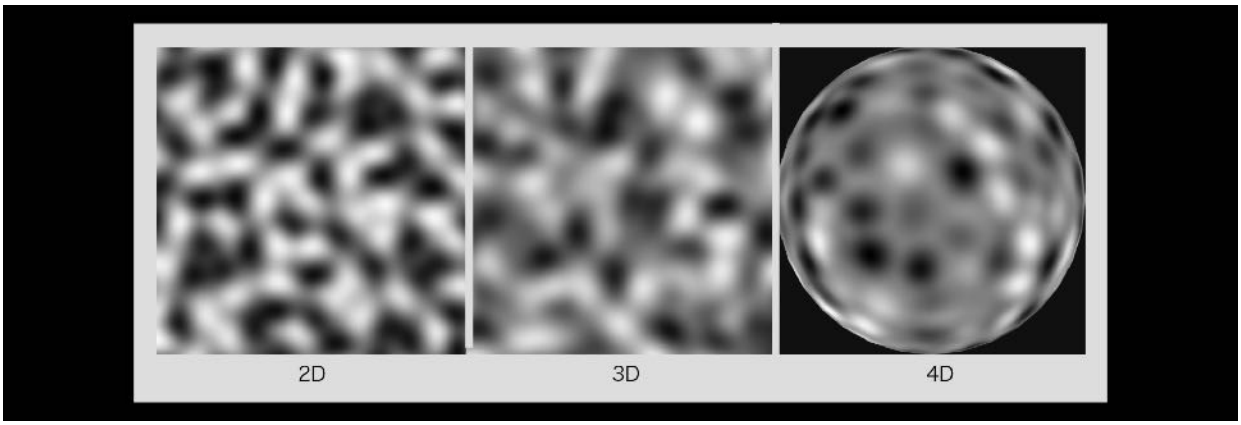

Figure 5.28: Simplex Noise (2D, 3D, 4D) results

Simplex Noise gives a slightly **grainier** result when compared to **Perlin Noise** .

5.4 Summary

We have looked at the algorithms and implementations of typical procedural noise methods in detail, but you can see that there are differences in the characteristics of the noise patterns obtained and the calculation costs. When noise is used in a real-time application, when it becomes high resolution, the calculation is performed for each pixel, so this calculation load cannot be ignored, and what kind of calculation is performed. Should be kept in mind to some extent. Nowadays, many noise functions are built into the development environment from the beginning, but it is important to understand the noise algorithm in order to make full use of it. I couldn't explain its application here, but in graphics generation, the application of noise is extremely diverse and has a great effect. (The next chapter will show one example.) We hope this article provides a foothold for countless applications. Finally, I would like to pay tribute to the wisdom that our predecessors have accumulated and primarily to Ken Perlin's outstanding achievements.

See 5.5

- [1] An Image Synthesizer, Ken Perlin, SIGGRAPH 1985
- [2] Improving Noise, Ken Perlin —http://mrl.nyu.edu/~perlin/paper445.pdf
- [3] Noise hardware. In Real-Time Shading SIGGRAPH Course Notes, Ken Perlin, 2001 — https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf
- [4] Improved Noise reference implementation, Ken Perlin, SIGGRAPH 2002http://mrl.nyu.edu/~perlin/noise/
- [5] GPU Gems Chapter 5. Implementing Improved Perlin Noise, Ken Perlin

—http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch05.html
- [6] Simplex noise demystified. Technical Report, Stefan Gustavson, 2005 —http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
- [7] Efficient computational noise in GLSL, Ian McEwan, David Sheets, Stefan Gustavson and Mark Richardson, 6 Apr 2012 —http://webstaff.itn.liu.se/~stegu/jgt2012/article.pdf
- [8] Direct computational noise in GLSL Supplementary material, Ian McEwan, David Sheets, Stefan Gustavson and Mark Richardson, 2012 — http://weber.itn.liu.se/~stegu/jgt2011/supplement.pdf
- [9] Texturing and Modeling; A Procedural Approach, Second Edition —
- [10] The Book of Shaers Noise, Patricio Gonzalez Vivo & Jen Lowe — https://thebookofshaders.com/11/
- [11] Building Up Perlin Noise — http://eastfarthing.com/blog/2015-04-21-noise/
- [12] Let's go with Z! I checked Extension for 3ds Max 2015 Part 24 3dsmax 2015 — http://blog.livedoor.jp/takezultima/archives/2015-05.html

Chapter 6 Curl Noise- Explanation of Noise Algorithms for Pseudo-Fluids

6.1 Introduction

In this chapter, we will explain the GPU implementation of Curl Noise, which is a pseudo-fluid algorithm.
The sample in this chapter is "Curl Noise" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming2
.

6.1.1 What is Curl Noise?

Curl Noise is a pseudo-fluid noise algorithm announced in 2007 by Professor Robert Bridson of the University of British Columbia, who is also known as a developer of fluid algorithms such as the FLIP method.
In the previous work "Unity Graphics Programming vol.1", I explained the fluid simulation using the Navier-Stokes equation, but Curl Noise is a pseudo but light load compared to those fluid simulations. It is possible to express fluid.
In particular, with the recent advances in display and projector technology, there is an increasing need for real-time rendering at high resolutions such as 4K and 8K, so low-load algorithms such as Curl Noise can express fluids in high resolution and low resolution. It is a useful option for expressing with machine specifications.

6.2 Curl Noise algorithm

In fluid simulation, the first thing you need is a vector field called the "velocity field". First, let's imagine what a velocity field is like.

Below is an image of the velocity field in two dimensions. You can see that the vector is defined at each point on the plane.

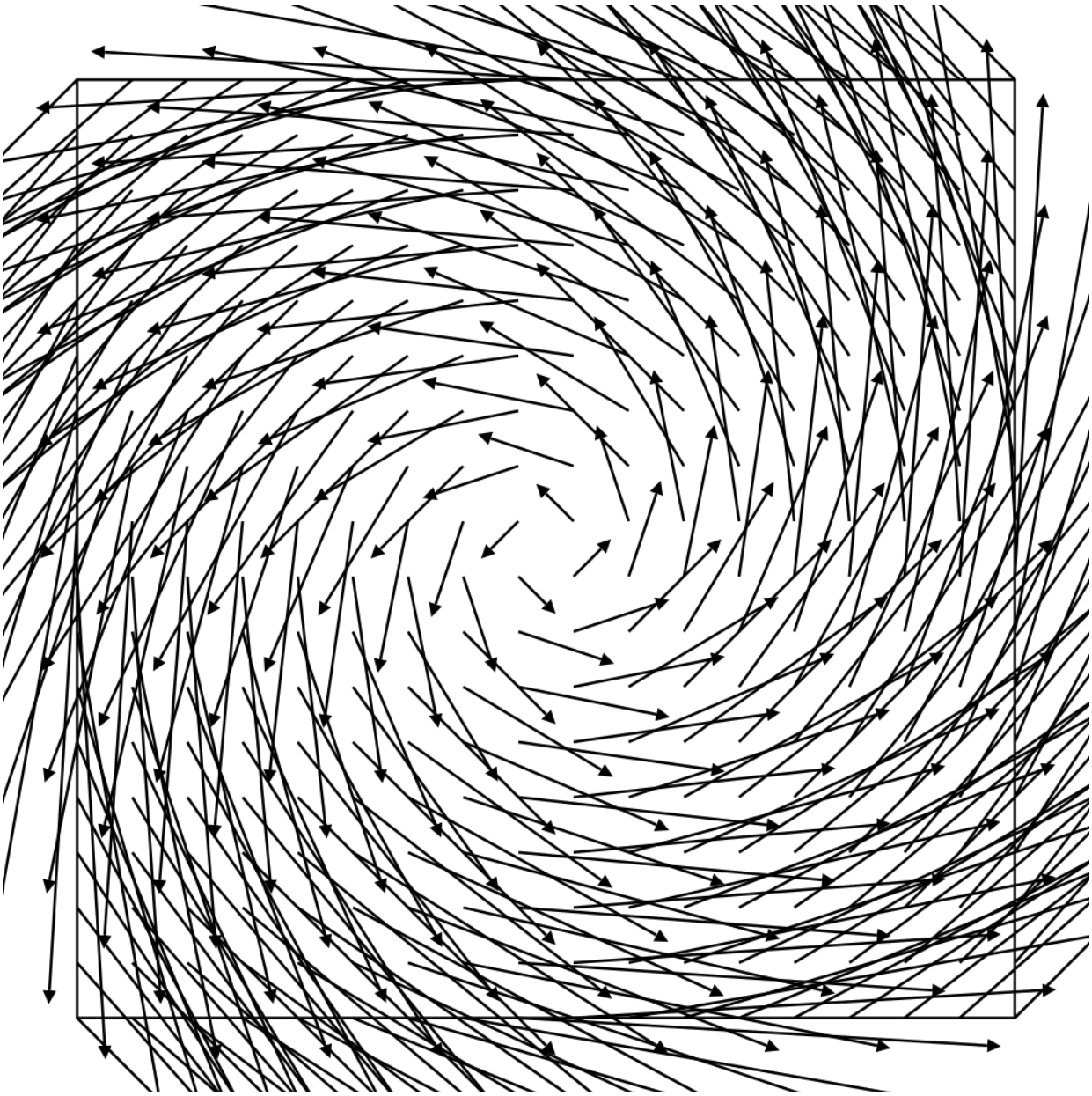

Figure 6.1: Two-dimensional velocity field observations

As shown in the above figure, the state in which each vector is individually defined in each differential interval on the plane is called a vector field, and the one in which each vector is a velocity is called a velocity field.
Even if these are three-dimensional, it is easy to understand if you can

imagine that the vector is defined in each differential block in the cube.
Now let's see how Curl Noise derives this velocity field.

The interesting thing about Curl Noise is that it uses gradient noise such as Perlin Noise and Simplex Noise, which was explained in the previous chapter "Introduction to Procedural Noise", as a potential field, and derives the velocity field of the fluid from it.
In this chapter, we will use 3D Simplex Noise as a potential field.

Below, I would like to first unravel the algorithm from the Curl Noise formula.

```
\overrightarrow{u} = \nabla \times \psi
```

The above is the Curl Noise algorithm.
The left side \ overright arrow {u} is the derived velocity vector, the right side \ nabla is the vector differential operator (read as nabla, which acts as an operator of partial differential), and \ psi is the potential field. (3D Simplex Noise in this chapter)
Curl Noise can be expressed as the cross product of the two terms on the right side.

In other words, Curl Noise is Simple x Noise and partial differential of each vector element \ left (\ dfrac {\ partial} {\ partial x} _, \ dfrac {\ partial} {\ partial y} _, \ dfrac {\ partial} { It is the outer product of \ partial z} \ right) , and for those who have learned vector analysis in the past, you can see that it is the shape of rotA itself.
Now let's calculate the outer product of 3D Simplex Noise and partial derivative

```
\overrightarrow{u} = \left( \dfrac {\partial \psi _3} {\partial y} - \dfrac {\partial \psi _2} {\partial z}_, \dfrac {\partial \psi _1} {\partial z} - \dfrac {\partial \psi _3} {\partial x}_, \dfrac {\partial \psi _2} {\partial x} - \dfrac {\partial \psi _1} {\partial y} \right)
```

In general, the outer product is characterized by the fact that the two vectors are oriented vertically to each other and their length is the same as the area of the surface stretched by both vectors, but rotA (rotation) in vector analysis.) Is a simple way to grasp the image of the cross product operation

from the above formula, saying, "Look up the vector of the potential field in each twisted partial differential element direction, and pull the terms together, so rotation occurs." It may be easier to grasp the image if you capture it in.
The implementation itself is very simple, looking up the vector from each point of \ psi above , that is, 3D SimplexNoise, while slightly shifting the lookup point in the direction of each element of partial differentiation, and performing the outer product operation like the above formula. Just do.

6.2.1 Conservation of mass

If you have read the fluid simulation chapter of the previous work "Unity Graphics Programming vol.1", you may be wondering what the law of conservation of mass is.
The law of conservation of mass is that at each point in the velocity field, the inflow and outflow are always balanced, the inflow is outflowed, the outflow is inflowed, and finally the divergence is zero (divergence free). It was a rule.

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

This is also mentioned in the paper, but since the gradient noise itself changes gently in the first place (when imagining with a two-dimensional gradation, if the pixel on the left side is thin, the pixel on the right side is dark (As you can see), divergence-free was guaranteed at the time of the potential field. Considering the characteristics of Perlin noise, it is quite natural.

6.2.2 Implementation of Curl Noise

Now, let's implement the CurlNoise function on the GPU with Compute shader or Shader based on the formula.

```
#define EPSILON 1e-3

float3 CurlNoise (float3 coord)
{
    float3 dx = float3(EPSILON, 0.0, 0.0);
    float3 dy = float3 (0.0, EPSILON, 0.0);
```

```
    float3 dz = float3(0.0, 0.0, EPSILON);

    float3 dpdx0 = snoise(coord - dx);
    float3 dpdx1 = snoise(coord + dx);
    float3 dpdy0 = snoise(coord - dy);
    float3 dpdy1 = snoise(coord + dy);
    float3 dpdz0 = snoise(coord - dz);
    float3 dpdz1 = snoise(coord + dz);

    float x = dpdy1.z - dpdy0.z + dpdz1.y - dpdz0.y;
    float y = dpdz1.x - dpdz0.x + dpdx1.z - dpdx0.z;
    float z = dpdx1.y - dpdx0.y + dpdy1.x - dpdy0.x;

    return float3(x, y, z) / EPSILON * 2.0;
}
```

As mentioned above, this algorithm can be reduced to a simple four arithmetic operation, so the implementation itself is very easy, and it can be implemented with just this number of lines.
Below is a sample of Curl Noise implemented in the compute shader this time. It is possible to advect particles of particles, add a rising vector to make it look like a flame, and bring out various expressions depending on the idea.

Figure 6.2:

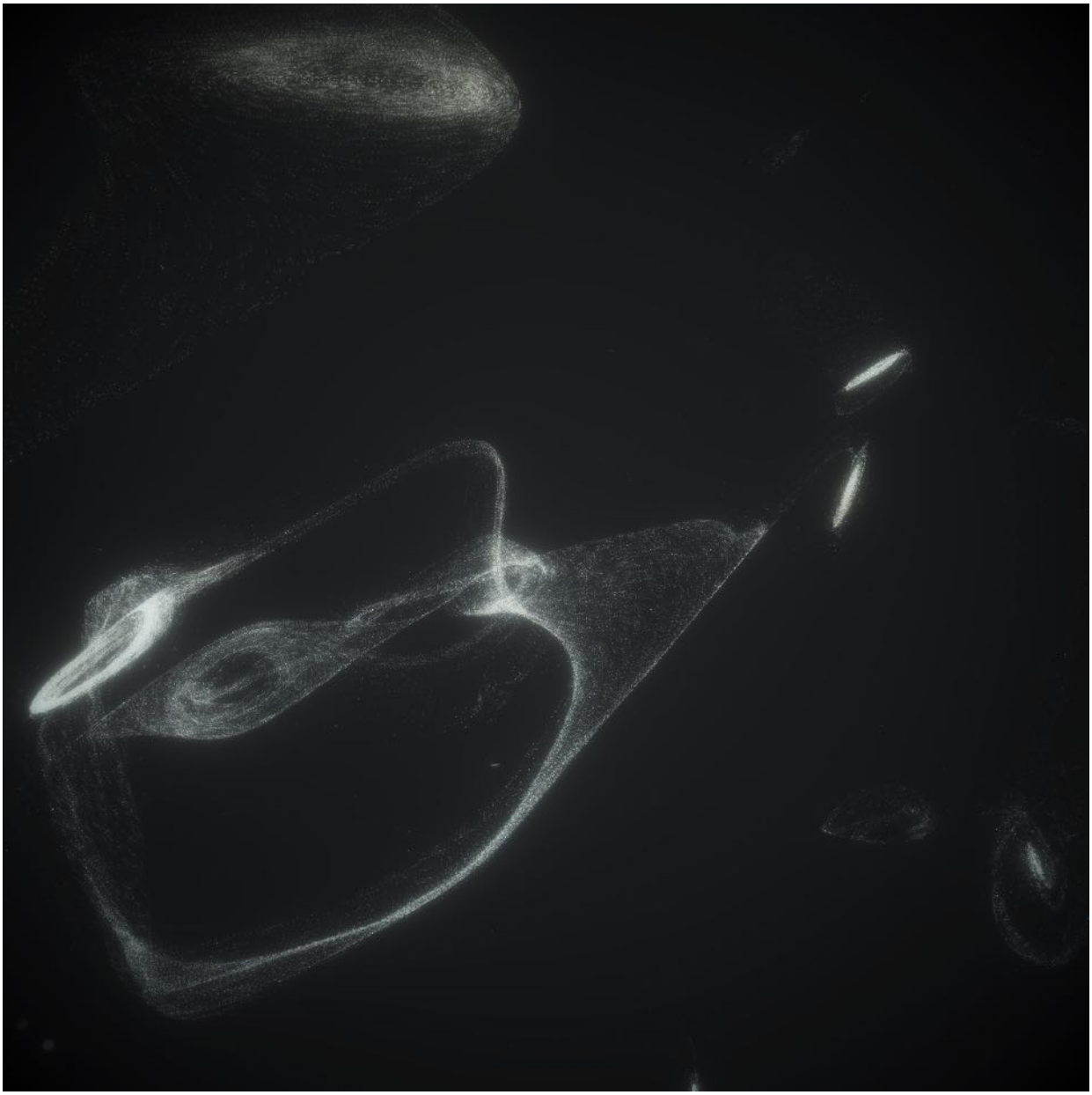

Figure 6.3:

Figure 6.4:

6.3 Summary

In this chapter, we explained the implementation of pseudo-fluid by Curl Noise.
Since it is possible to reproduce a 3D pseudo-fluid with a small load and implementation, it is an algorithm that works especially useful for real-time rendering at high resolution.

In summary, I would like to conclude this chapter with the utmost thanks to Professor Robert Bridson, who is still discovering various techniques, including the Curl Noise algorithm.
I think there were some points that could not be explained and some parts were difficult to understand, but I hope that readers will enjoy graphics programming as well.

6.4 References

- Robert Bridson, Jim Hourihan, Marcus Nordenstam. 2007, Curl-noise for procedural fluid flow. In proc, ACM SIGGRAPH 46.

Chapter 7 Shape Matching-Application of Linear Algebra to CG

7.1 Introduction

In this chapter, we will introduce learning of singular value decomposition and applications using singular value decomposition from the basics and applications of linear algebra. Although I learned linear algebra in high school students and university students, I think that there are many people who do not know how it is used, so I wrote this time. In this chapter, in order to prioritize ease of understanding , explanations are given in **two dimensions and within the range of real numbers.** Therefore, there are some differences from the actual definition of linear algebra, but we would appreciate it if you could read it as appropriate.

7.2 What is a matrix?

Most readers may have heard the word matrix once (currently, they don't learn matrices in high school ...). Matrix is a number like this: Refers to those arranged vertically and horizontally.

The horizontal direction is the **row** , the vertical direction is the **column** , the diagonal direction is the **diagonal** , and each number is called a matrix **element** .

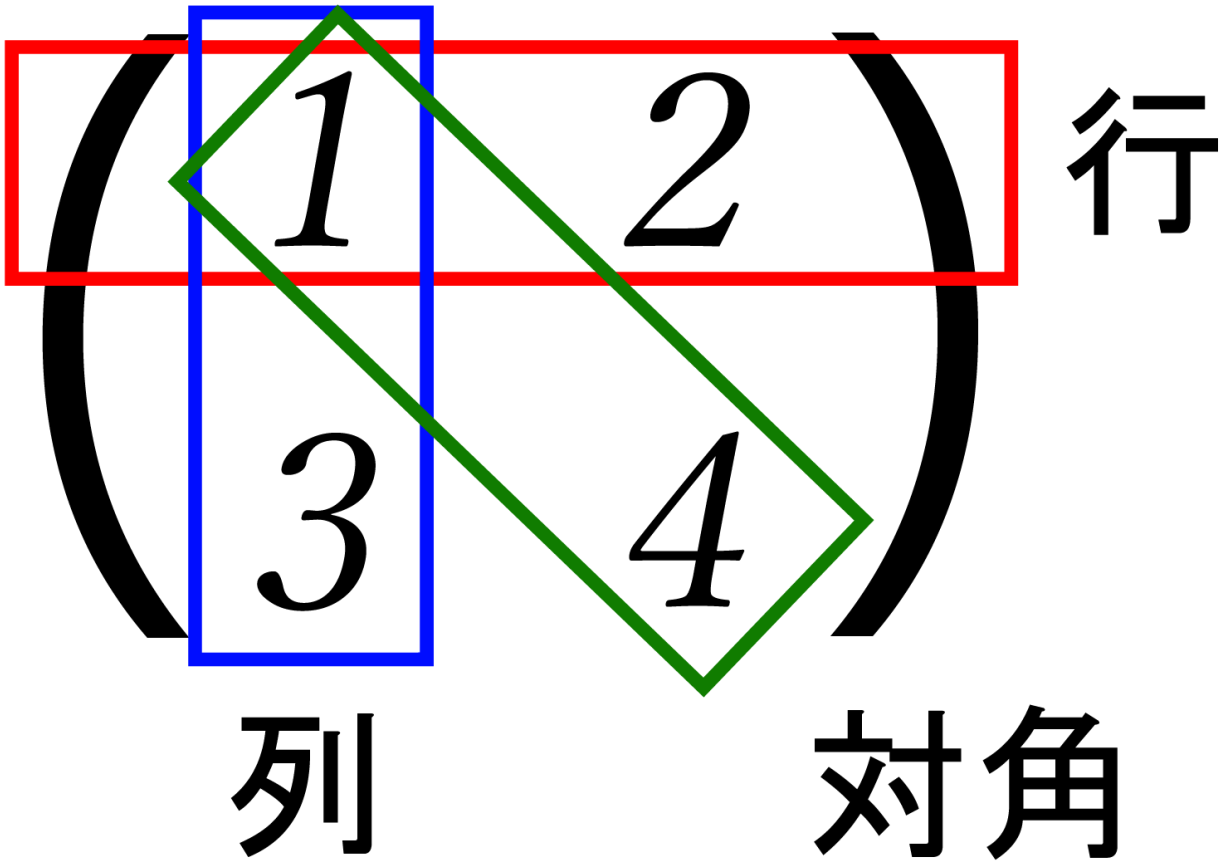

Figure 7.1: Matrix

By the way, the matrix is called Matrix in English.

7.3 Review of matrix operations

Let's take a quick look at the basics of matrix operations. If you are already familiar with it, you can skip this section.

7.3.1 Addition, subtraction, multiplication and division

Similar to the four arithmetic operations of scalars, there are additions, subtractions, multiplications and divisions in matrices. For simplicity, quadratic square matrices * 1 \ mbox {\ boldmath $ A $} and \ mbox {\

boldmath $ B $} , The 2D vector \ mbox {\ boldmath $ c $} is defined below.

[* 1] Square matrix \ cdots A matrix with the same number of rows and columns.

Addition

Matrix addition calculates the sum for each element, as in the formula (\ ref {plus}).

Subtraction

Matrix subtraction calculates the difference element by element, as in the formula (\ ref {minus}).

Multiply

Matrix multiplication is a bit more complicated, like an expression (\ ref {times}).

Please note that if you reverse the order of multiplication, the calculation result will also change.

Divide (reverse column)

Matrix division uses a concept called the **inverse matrix** , which is a little different from division in scalars . First of all, scalars have the property that when multiplied by their own reciprocal, they always become 1.

In other words, the act of division is equivalent to the operation of "multiplying the reciprocal".

Replacing this with a matrix, we can say that what produces the identity matrix over the matrix is the matrix that represents division. In the matrix, the one corresponding to 1 in the scalar is called the identity matrix and is defined below. ..

As with the scalar 1 , the value does not change when the identity matrix is applied to any matrix.

With these in mind, let's consider matrix division. If the inverse matrix is \ mbox {\ boldmath $ M $} ^ {-1} , the definition of the inverse matrix is as follows.

Derivation is omitted, but the elements of the inverse matrix of the matrix \ mbox {\ boldmath $ A $} are defined below.

At this time, a_ {00} a_ {11} --a_ {01} a_ {10} is called a determinant (Determinant) and is expressed as det (\ mbox {\ boldmath $ A $}) .

7.3.2 Matrix action on vector

A matrix can transform the coordinates pointed to by a vector by multiplying it by a vector. As you know, in CG, it is mostly used as a coordinate transformation matrix (world, projection, view transformation matrix). The product of and the vector is defined below.

7.4 Application of matrix operation

From this section, I will explain the concept of the matrix in the range learned at the university. I think that there are many parts that seem a little difficult, but to understand Shape Matching, this concept is necessary, so do your best. However, since the story in this section is also absorbed inside the matrix operation library, there is no problem in implementing it even if you skip to section \ ref {shape matching}.

7.4.1 Transpose matrix

The transposed matrix is the swapped rows and columns of the elements and is defined below.

7.4.2 Symmetric matrix

A matrix that satisfies \ mbox {\ boldmath $ A $} ^ {T} = \ mbox {\ boldmath $ A $} is called a symmetric matrix.

7.4.3 Eigenvalues and eigenvectors

Given the square matrix \ mbox {\ boldmath $ A $} ,

Meet the \ lambda a, \ Mbox {\ Boldmath $ A $} of **eigenvalues** , \ Mbox {\ Boldmath $ V $} the **eigenvectors** is called.

The calculation method of eigenvalues and eigenvectors is shown below. First, the formula (\ ref {eigen}) is transformed.

Here, using the condition \ mbox {\ boldmath $ v $} \ neq 0 , the expression (\ ref {eigen2}) becomes:

Becomes. Expanding this expression, \ lambda to become a quadratic equation for, by solving this \ lambda can be calculated. Moreover, each of the calculated \ lambda expressions (\ ref { By substituting into eigen2}), you can calculate the eigenvector \ mbox {\ boldmath $ v $} .

Since the concept of eigenvalues and eigenvectors is difficult to understand from mathematical formulas alone, I think you should also read the Qiita article (described at the end of the chapter) where eigenvalues were visualized by @ kenmatsu4.

7.4.4 Eigenvalue decomposition

The eigenvalues and eigenvectors in the square matrix \ mbox {\ boldmath $ A $} can be used to represent the matrix \ mbox {\ boldmath $ A $} in different ways. First, the eigenvalues \ lambda are sorted by size . , Create a matrix \ mbox {\ boldmath $ \ Lambda $} with it as a diagonal element . Next, a matrix \ mbox {\ boldmath $ V $} in which the eigenvectors corresponding to each eigenvalue are arranged in order from the left . Then, the expression (\ ref {eigen}) can be rewritten using these matrices as follows.

Furthermore, multiply this by \ mbox {\ boldmath $ V $} ^ {-1} from the right of both sides so that the matrix \ mbox {\ boldmath $ A $} remains on the left side .

Will be.

Decomposing a matrix into a form like an expression (\ ref {eigendecomp}) in this way is called eigenvalue decomposition of a matrix.

7.4.5 Orthonormal Basis

A set of vectors that are perpendicular to each other and each is a unit vector is called an orthonormal basis. Any vector can be represented using a set of orthonormal bases * 2. In the case of two dimensions, There are two vectors that can be orthonormal bases. For example, the commonly used x-axis and y-axis are \ mbox {\ boldmath $ x $} = (1, 0), \ mbox {\ boldmath $ y $} = Since the set of (0, 1) forms an orthonormal basis, any vector can be represented by this \ mbox {\ boldmath $ x $}, \ mbox {\ boldmath $ y $} . \ Expressing mbox {\ boldmath $ v $} = (4, 13) using orthonormal basis \ mbox {\ boldmath $ x $}, \ mbox {\ boldmath $ y $} , \ mbox {\ boldmath $ v $} = 4 \ mbox {\ boldmath $ x $} + 13 \ mbox {\ boldmath $ y $} .

[* 2] Formally, it is called a linear combination of vectors.

7.4.6 Hermitian matrix

The Hermitian matrix is defined in the range of complex numbers, which is beyond the scope of this chapter, so I will briefly explain it in the range of real numbers. In the range of real numbers, the Hermitian of the matrix \ mbox {\ boldmath $ A $} The matrix \ mbox {\ boldmath $ A $} ^ {*} simply means that it is a symmetric matrix.

Will be.

7.4.7 Orthogonal matrix

Square matrix \ mbox {\ boldmath $ Q $} column vector Q = (\ mbox {\ boldmath $ q $} _1, \ mbox {\ boldmath $ q $} _2, \ cdots, \ mbox {\

boldmath $ q $ When decomposed into } _n) , the set of these vectors forms an orthonormal system, that is,

When is true, \ mbox {\ boldmath $ Q $} is said to be an orthogonal matrix. Also, even if the orthogonal matrix is decomposed into row vectors, it has the characteristic of forming an orthonormal system.

7.4.8 Unitary matrix

A matrix that satisfies is called a unitary matrix. If all the elements of the unitary matrix \ mbox {\ boldmath $ U $} are real (execution columns), \ mbox {\ boldmath $ U $} ^ {*} = \ mbox Since it becomes {\ boldmath $ U $} ^ {T} , we can see that the real unitary matrix \ mbox {\ boldmath $ U $} is an orthogonal matrix.

7.5 Singular value decomposition

Decomposing an arbitrary m \ times n matrix \ mbox {\ boldmath $ A $} into the following form is called singular value decomposition of the matrix.

Note that \ mbox {\ boldmath $ U $} and \ mbox {\ boldmath $ V $} ^ {T} are orthogonal matrices of m \ times m , and \ mbox {\ boldmath $ \ Sigma $} are m \ times. It is a diagonal matrix of n (diagonal elements are non-negative and arranged in order of magnitude).

The word "arbitrary" is important, and the eigenvalue decomposition of a matrix is defined only for a square matrix, but the singular value decomposition can also be performed for a square matrix. In the CG world, the matrix to be handled is a square matrix. In most cases, the calculation method is not so different from the eigenvalue decomposition. Also, when \ mbox {\ boldmath $ A $} is a symmetric matrix, the eigenvalues and singular values of \ mbox {\ boldmath $ A $} matches. in addition, \ mbox {\ boldmath $ a $ } ^ {T} \ mbox {\ boldmath $ a $ } of 0 is a positive eigenvalues is not the square root of \ mbox {\ boldmath $ a $ } of singular value is.

7.5.1 Singular value decomposition algorithm

The dropped eigenvalue decomposition to program the formula is helpful to the (\ ref {svd}) Formula deformed. Matrices \ mbox {\ boldmath $ A $ } transpose of the left from the matrix of \ mbox {\ boldmath $ A $ } Multiply by ^ {T} to get:

You will notice that the form is the same as the eigenvalue decomposition. In fact, it is known that the square of the singular value matrix becomes the eigenvalue matrix. Therefore, the calculation of the singular value decomposes the matrix into eigenvalues. This can be done by taking the square root of the eigenvalues. This leads to the incorporation of eigenvalue decomposition into the algorithm, but fortunately it is necessary to solve a quadratic equation to find the eigenvalues. Since the solution formula of the quadratic equation is simple, it is easy to drop it into the program * 3 .

[* 3] Although there are solution formulas for cubic and quartic equations, they are generally calculated using Newton's method.

\ mbox {\ boldmath $ A $} ^ {T} \ mbox {\ boldmath $ A $} By eigenvalue decomposition, \ mbox {\ boldmath $ \ Sigma $} and \ mbox {\ boldmath $ V $} ^ { Now that T} has been calculated, the remaining \ mbox {\ boldmath $ U $} can be calculated by transforming the formula (\ ref {svd}) as follows.

Since V is an orthogonal matrix, the transpose and the inverse matrix match.

This can be expressed programmatically as follows.

Listing 7.1: Singular Value Decomposition Algorithm (Matrix2x2.cs)

```
 1: /// <summary>
 2: /// Singular value decomposition
 3: /// </summary>
 4: /// <param name="u">Returns rotation matrix u</param>
 5: /// <param name="s">Returns sigma matrix</param>
   6:   ///   <param   name="v">Returns   rotation   matrix   v(not
transposed)</param>
  7:  public  void  SVD(ref  Matrix2x2  u,  ref  Matrix2x2  s,  ref
Matrix2x2 v)
 8: {
  9:  //  If  it  was  a  diagonal  matrix,  the  singular  value
decomposition is simply given below.
```

```
10:     if (Mathf.Abs(this[1, 0] - this[0, 1]) < MATRIX_EPSILON
11:         && Mathf.Abs(this[1, 0]) < MATRIX_EPSILON)
12:     {
13:         u.SetValue(this[0, 0] < 0 ? -1 : 1, 0,
14:                     0, this[1, 1] < 0 ? -1 : 1);
15:          s.SetValue(Mathf.Abs(this[0, 0]), Mathf.Abs(this[1,
1]));
16: v.LoadIdentity ();
17:     }
18:
19: // Calculate A ^ T * A if it is not a diagonal matrix.
20:     else
21:     {
22: // 0 Column vector length (non-root)
23:          float i   = this[0, 0] * this[0, 0] + this[1, 0] *
this[1, 0];
24: // Length of 1 column vector (non-root)
25:          float j   = this[0, 1] * this[0, 1] + this[1, 1] *
this[1, 1];
26: // Inner product of column vectors
27:         float i_dot_j = this[0, 0] * this[0, 1]
28:                         + this[1, 0] * this[1, 1];
29:
30: // If A ^ T * A is an orthogonal matrix
31:         if (Mathf.Abs(i_dot_j) < MATRIX_EPSILON)
32:         {
33: // Calculation of diagonal elements of the singular value
matrix
34:             float s1 = Mathf.Sqrt(i);
35:             float s2 = Mathf.Abs(i - j) <
36:                         MATRIX_EPSILON ? s1 : Mathf.Sqrt(j);
37:
38:             u.SetValue(this[0, 0] / s1, this[0, 1] / s2,
39:                         this[1, 0] / s1, this[1, 1] / s2);
40:             s.SetValue(s1, s2);
41: v.LoadIdentity ();
42:         }
43: // If A ^ T * A is not an orthogonal matrix, solve the
quadratic equation to find the eigenvalues.
44:         else
45:         {
46: // Calculation of eigenvalues / eigenvectors
47: float i_minus_j = i --j; // Difference in column vector
length
48: float i_plus_j = i + j; // sum of column vector lengths
49:
50: // Formula for solving quadratic equations
51:             float root = Mathf.Sqrt(i_minus_j * i_minus_j
```

```
52:                                                 + 4 * i_dot_j *
i_dot_j);
53:             float eig = (i_plus_j + root) * 0.5f;
54:             float s1 = Mathf.Sqrt(eig);
55:             float s2 = Mathf.Abs(root) <
56:                         MATRIX_EPSILON ? s1 :
57:                           Mathf.Sqrt((i_plus_j - root) / 2);
58:
59:             s.SetValue(s1, s2);
60:
61: // Use the eigenvector of A ^ T * A as V.
62:             float v_s = eig - i;
63:                float len = Mathf.Sqrt(v_s * v_s + i_dot_j *
i_dot_j);
64: i_dot_j / = len;
65: v_s / = only;
66:             v.SetValue(i_dot_j, -v_s, v_s, i_dot_j);
67:
68: // Since v and s have already been calculated, the rotation
matrix u is calculated by Av / s.
69:             u.SetValue(
70:                 (this[0, 0] * i_dot_j + this[0, 1] * v_s) /
s1,
71:                 (this[0, 1] * i_dot_j - this[0, 0] * v_s) /
s2,
72:                 (this[1, 0] * i_dot_j + this[1, 1] * v_s) /
s1,
73:                 (this[1, 1] * i_dot_j - this[1, 0] * v_s) /
s2
74:             );
75:         }
76:     }
77: }
```

7.6 Algorithm using singular value decomposition

Singular value decomposition is active in a wide variety of fields, and seems to be used mainly in principal component analysis (PCA) in statistics. There are many cases where it is used in CG.

- Shape Matching*4
- Anisotropic Kernel*5
- Material Point Method*6

And so on.

This time, we will focus on Shape Matching and explain the basic idea.

[*4] Meshless deformations based on shape matching, Matthias Muller et al., SIGGRAPH 2005

[*5] Reconstructing surfaces of particle-based fluids using anisotropic kernels, Jihun Yu et al., ACM Transaction on Graphics 2013

[*6] A material point method for snow simulation, Alexey Stomakhin et al., SIGGRAPH 2013

7.7 Shape Matching

\label{shapematching}

7.7.1 Overview

Shape Matching is a technique for aligning two different shapes within the range where there is as little error as possible. Currently, a method for simulating an elastic body using Shape Matching is being developed. ..

This section describes the algorithm for aligning the unicorn object placement to the lion object placement , as shown in Figure 7.2 and Figure 7.3 .

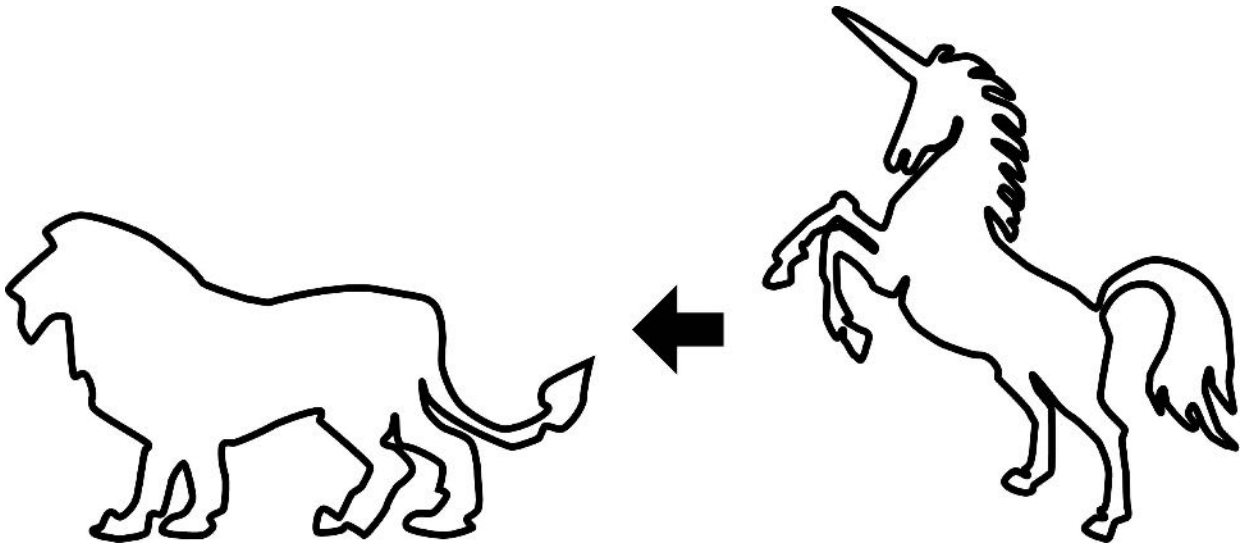

Figure 7.2: Two objects

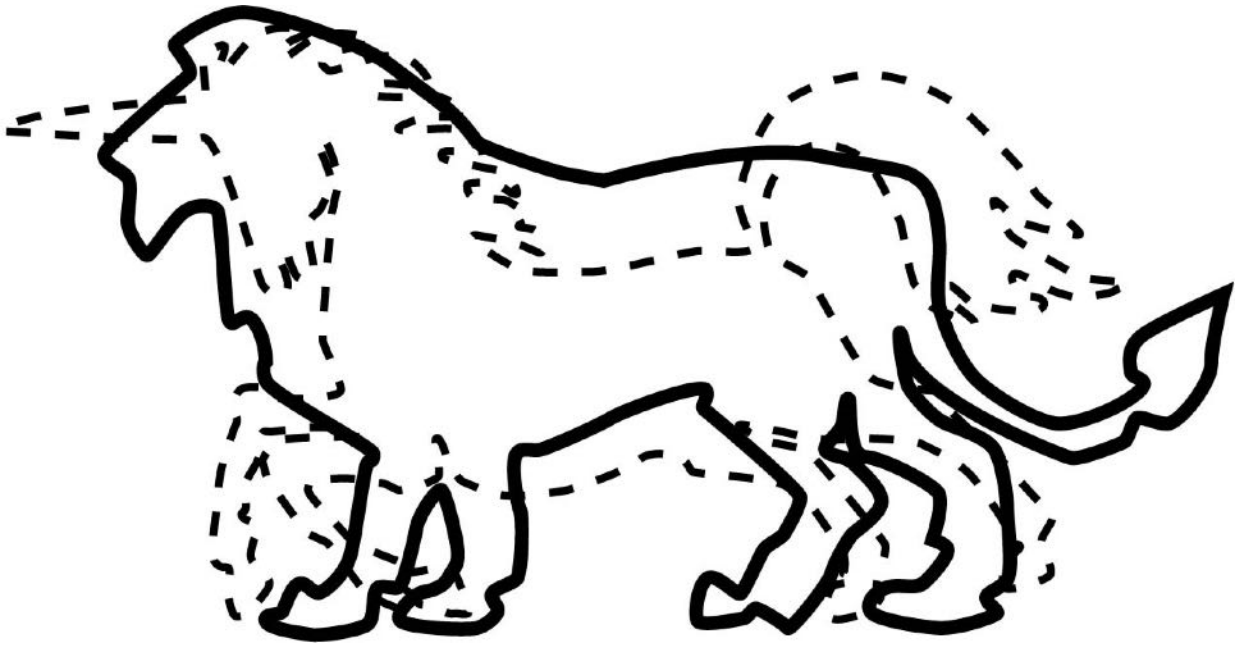

Figure 7.3: Aligned results

7.7.2 Algorithm

First, define a set of the same number of points on each shape. (Lion's point set is P, Unicorn's point set is Q.)

At this time, note that those with the same subscript are in the geometrically corresponding positions as shown in Fig. 7.4 .

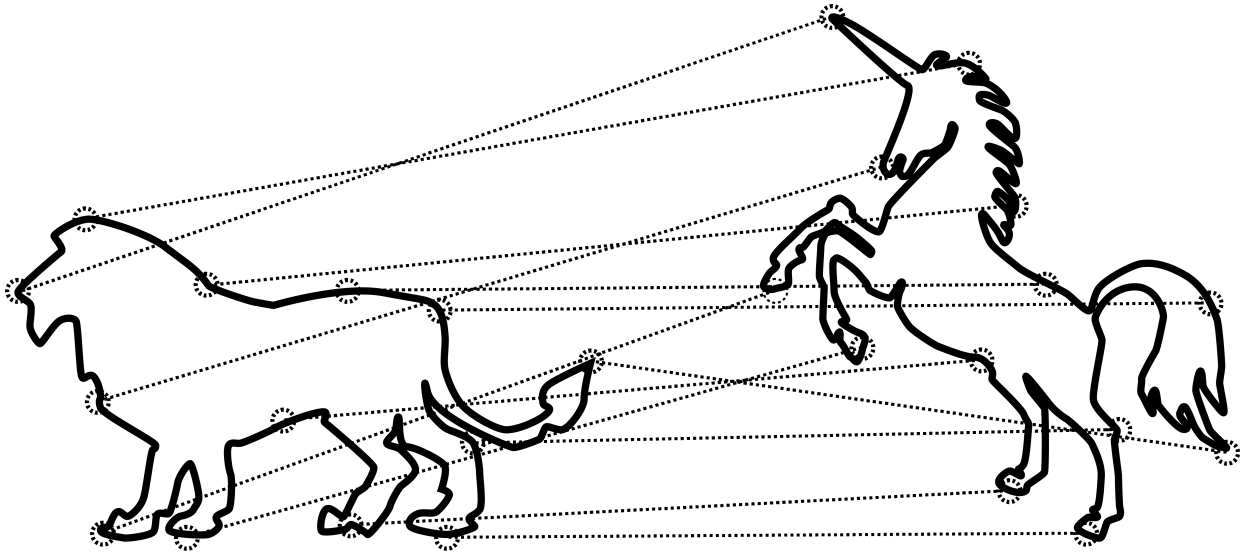

Figure 7.4: Correspondence of point sets

Next, calculate the centroid of each point set.

Assuming that the center of gravity of the unicorn point set is at the same position as the center of gravity of the lion point set, the rotation matrix \ mbox {\ boldmath $ R $} is applied to the unicorn point set, and the vector \ mbox {\ boldmath $ t Since the result of the $} translation is equal to the center of gravity of the lion, the following equation can be derived.

When this is transformed,

And further deformed,

Will be.

Therefore, from this equation, if the rotation matrix \ mbox {\ boldmath $ R $} is obtained, the translation vector \ mbox {\ boldmath $ t $} is automatically obtained. Here, the original point Define a set of points from the position of, minus the center of gravity of each.

This makes it possible to perform calculations with local coordinates with the center of gravity of each point set as the origin.

Next, the variance-covariance matrix \ mbox {\ boldmath of \ mbox {\ boldmath $ p $} _ {i} ^ {\ prime}, \ mbox {\ boldmath $ q $} _ {i} ^ {\ prime} Calculate $ H $} .

This variance-covariance matrix \ mbox {\ boldmath $ H $} stores information such as the variability of the two point sets. Here the product of the vectors \ mbox {\ boldmath $ q $} _ {i } ^ {\ prime} {\ mbox {\ boldmath $ p $} _ {i} ^ {\ prime}} ^ {T} is an operation called direct product (outer product), unlike the normal vector internal product operation. The direct product of the vectors produces a matrix. The direct product of the two-dimensional vectors is defined below.

In addition, the covariance matrix \ mbox {\ boldmath $ H $} is singularly decomposed.

In the result of singular value decomposition, \ mbox {\ boldmath $ \ sum $} is a matrix representing expansion and contraction, so the desired rotation matrix \ mbox {\ boldmath $ R $} is

(The detailed derivation method is a little advanced, so I will omit it here.)

Finally, the translation vector \ mbox {\ boldmath $ t $} can be calculated from the obtained rotation matrix and equation (\ ref {trans}) .

7.7.3 Implementation

In this implementation, the algorithm in the previous section is just dropped into the code, so detailed explanation is omitted. In addition, all the processing is completed in the Start function in ShapeMaching.cs.

Listing 7.2: ShapeMatching (ShapeMaching.cs)

```
1: // Set p, q
2: p = new Vector2[n];
3: q = new Vector2[n];
4: centerP = Vector2.zero;
```

```
 5: centerQ = Vector2.zero;
 6:
 7: for(int i = 0; i < n; i++)
 8: {
      9:                                  Vector2       pos       =
_destination.transform.GetChild(i).position;
10:     p[i] = pos;
11:     centerP += pos;
12:
13:     pos = _target.transform.GetChild(i).position;
14:     q[i] = pos;
15:     centerQ += pos;
16: }
17: centerP /= n;
18: centerQ /= n;
19:
20: // Calc, p, q!
21: Matrix2x2 H = new Matrix2x2(0, 0, 0, 0);
22: for (int i = 0; i < n; i++)
23: {
24:     p[i] = p[i] - centerP;
25:     q[i] = q[i] - centerQ;
26:
27:     H += Matrix2x2.OuterProduct(q[i], p[i]);
28: }
29:
30: Matrix2x2 u = new Matrix2x2();
31: Matrix2x2 s = new Matrix2x2();
32: Matrix2x2 v = new Matrix2x2();
33: H.SVD(ref u, ref s, ref v);
34:
35: R = v * u.Transpose();
36: Debug.Log(Mathf.Rad2Deg * Mathf.Acos(R.m00));
37: t = centerP - R * centerQ;
```

7.8 Results

I was able to safely align the shape of the unicorn with the shape of the lion.

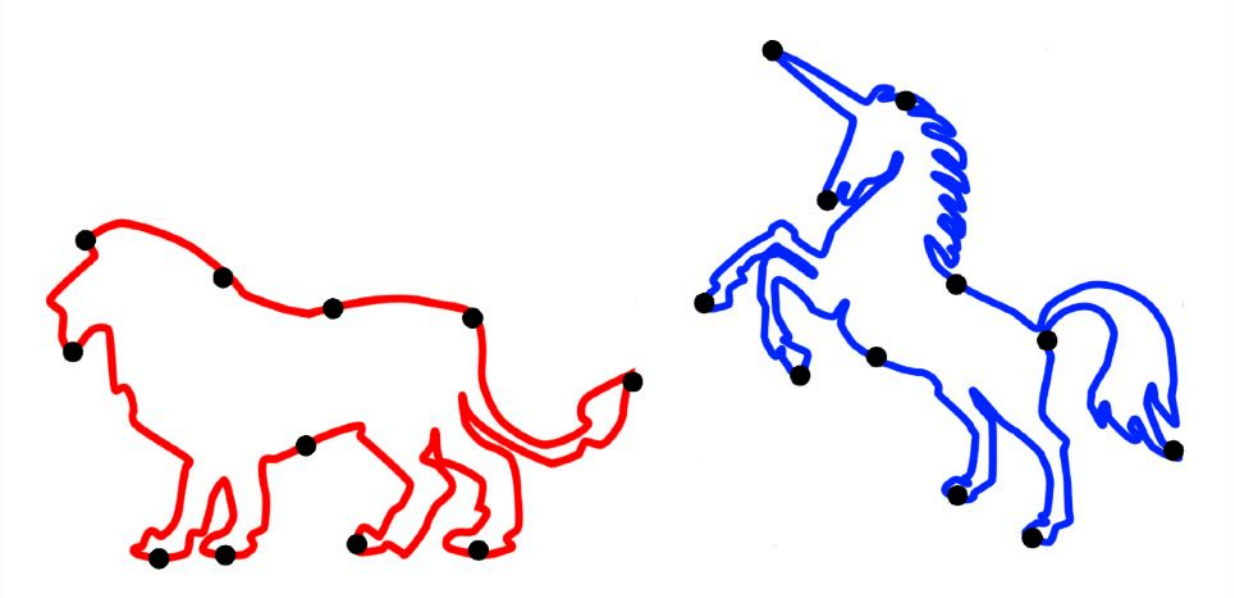

Figure 7.5: Before execution

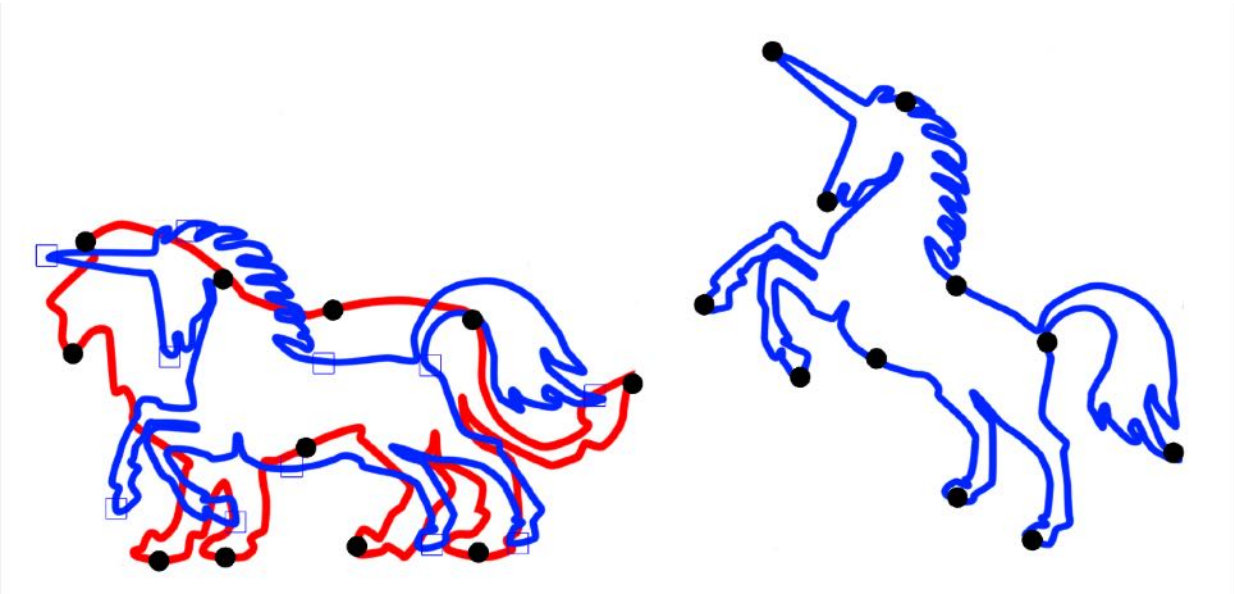

Figure 7.6: After execution

7.9 Summary

In this section, we explained the implementation of the Shape Matching method using singular value decomposition. This time, it was implemented in 2D, but it can also be implemented in 3D with the same algorithm. I think

that there were many, but I hope that you will take this opportunity to become interested in the application method of matrix arithmetic in the CG field and deepen your learning.

7.10 References

- 3D Geometry for Computer Graphics (https://igl.ethz.ch/teaching/tau/cg/cg2005/svd.ppt)
- Mathematical linear algebra of science and engineering (by Toshitaka Nagai, Atsushi Nagai) Shokabo
- Singular Value Decomposition (the SVD) : MIT OpenCourseWare (https://www.youtube.com/watch?v=mBcLRGuAFUk)
- Lecture: The Singular Value Decomposition (SVD) : AMATH 301 (https://www.youtube.com/watch?v=EokL7E6o1AE)
- Visualize what eigenvalues and eigenvectors are @ kenmatsu4 (https://qiita.com/kenmatsu4/items/2a8573e3c878fc2da306)

Chapter 8 Space Filling

8.1 Introduction

This chapter focuses on the **Space filling problem** * 1 and explains **Apollonian Gasket** , which is one of the methods to solve it .

Since this chapter focuses on the algorithm explanation of Apollonius Gasket, it deviates a little from the story of graphic programming.

8.2 Space filling problem

The space filling problem is the problem of finding a method to fill the inside of one closed plane as much as possible with a certain shape without overlapping. This problem is an area that has been studied for a long time, especially in the fields of geometry and combinatorial optimization. Since there are innumerable combinations of what kind of plane to fill with what kind of shape, various methods have been proposed for each combination.

To give a few examples

- Rectangular packing * 2 : O-Tree method
- Polygonal packing * 3 : Bottom-Left method
- Circle packing * 4 : Apollonian Gasket
- Triangular packing * 5 : Sierpinski Gasket

[* 1] Other names such as "tessellation", "packing problem", and "packing problem" are used.

[* 2] Rectangular packing \ cdots Fill the rectangular plane with a rectangle

[* 3] Polygon packing \ cdots Fill the rectangular plane with polygons

[* 4] Circle packing \ cdots Fill the inside of a circular plane with a circle

[* 5] Triangular packing \cdots Fill the triangular plane with triangles

And so on, there are many other techniques. In this chapter, we will explain about Apollonian Gasket among the above.

The Space filling problem is known to be NP-hard, and it is currently difficult to always fill the plane 100% with any of the above algorithms. The same is true for the Apollonian Gasket, which cannot completely fill the inside of a circle.

8.3 Apollonian Gasket

The Apollonian Gasket is a type of fractal figure generated from three circles that touch each other. This is a kind of the earliest fractal figure, and it is said that one of the research results of plane geometry could be one solution of the Space filling problem, not the algorithm proposed to solve the Space filling problem. is. The name is named after Apollonius of Perga, a Greek mortar in BC.

First, assuming that the three circles that touch each other are C1, C2, and C3, respectively, Apollonius discovered that there are two non-intersecting circles C4 and C5 that touch all of C1, C2, and C3. These C4 and C5 are **Apollonius circles** for C1, C2 and C3 **(details will be described later)** .

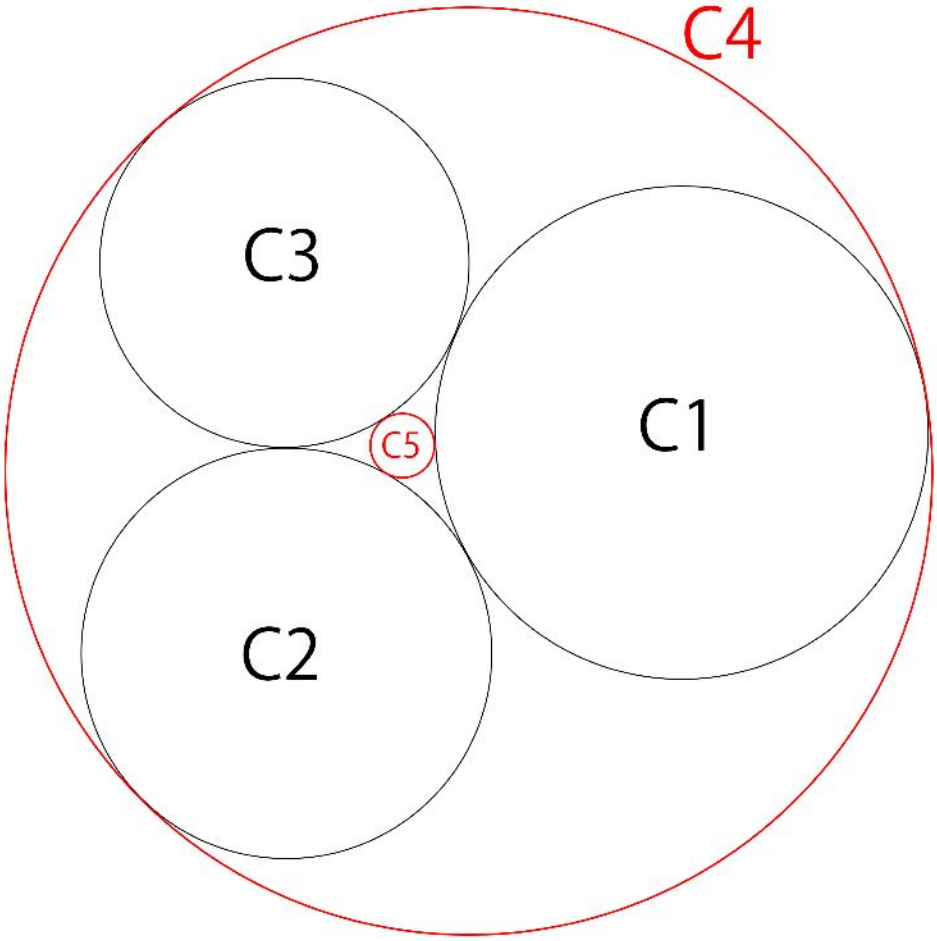

Figure 8.1: C1, C2, C3 and the two circles C4, C5 in contact with it

Now, if we consider C4 as opposed to C1, C2, we can get two new Apollonius circles for C1, C2, and C4. Of these two circles, one will be C3 and the other will be the new circle C6.

Considering the Apollonius circles for all combinations, such as (C1, C2, C5), (C2, C3, C4), (C1, C3, C4), you can get at least one new circle for each. I can do it. By repeating this infinitely, a set of circles that touch each other is created. This set of circles is the Apollonian Gasket.

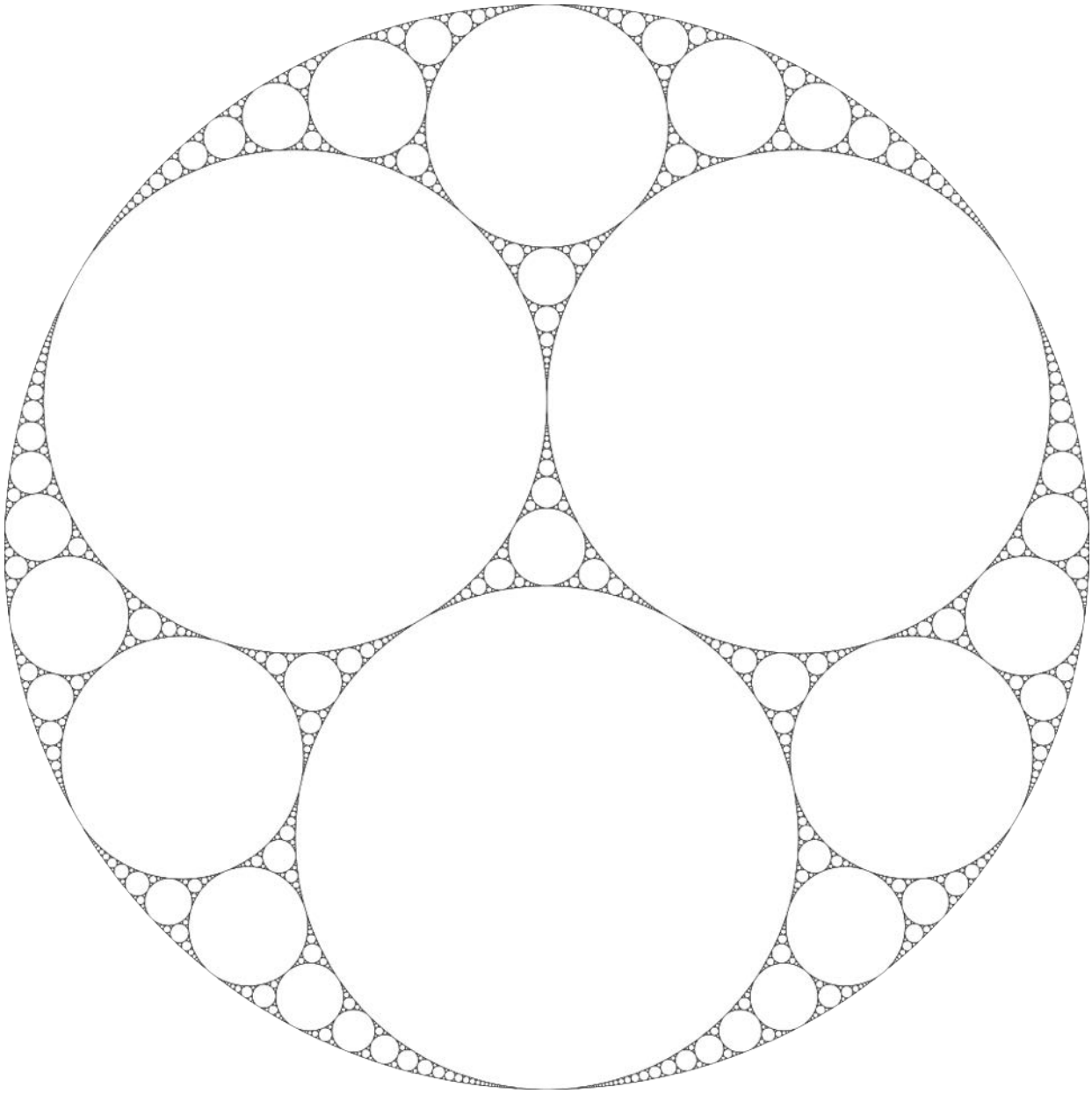

Figure 8.2: Apollonian Gasket

https://upload.wikimedia.org/wikipedia/commons/e/e6/Apollonian_gasket.svg

Circles of Apollonius

It is the locus of the point P when the two fixed points A and B are taken and the point P is taken so that AP: BP = constant . However, apart from this, it

refers to the solution to the Apollonius problem and is sometimes called the Apollonius circle, which has a stronger meaning in the Apollonius Gasket.

Problem of Apollonius

In Euclidean geometry, the problem is to draw a fourth circle tangent to the given three circles. It is said that there are a maximum of eight solutions for this fourth circle, two of which are always circumscribed at 3 yen, and two circles are always inscribed at 3 yen.

By the way, the three circles given as a condition do not have to touch each other, and the problem is to draw a fourth circle that touches the three circles.

8.4 Apollonian Gasket Calculation

From here, I will explain the calculation method of Apollonian Gasket in order while looking at the actual program. A sample program is available on Github, so please download it from there if necessary.

URL: https://github.com/IndieVisualLab/UnityGraphicsProgramming2

8.4.1 Preparation

In programming the Apollonian Gasket, this time we have prepared our own class to represent the circle and a structure to handle complex numbers.

Circle.cs

```
using UnityEngine;

public class Circle
{
    public float Curvature
    {
        get { return 1f / this.radius; }
    }
```

```
    public Complex Complex
    {
        get; private set;
    }
    public float Radius
    {
        get { return Mathf.Abs(this.radius); }
    }
    public Vector2 Position
    {
        get { return this.Complex.Vec2; }
    }

    private float radius = 0f;


    public Circle(Complex complex, float radius)
    {
        this.radius = radius;
        this.Complex = complex;
    }

    /// ...
      /// Below, a function to check the relationship between
circles is implemented.
    /// Whether they are in contact, intersect, include, etc.
    /// ...
}
```

Part of the implementation of a class that represents a circle.

If you have basic programming knowledge, there should be nothing difficult. In addition, `Complex`is a complex number structure prepared by myself this time , and is `Curvature`called curvature, both of which are necessary values for calculating the Apollonian Gasket.

Complex.cs

```
using UnityEngine;
using System;
using System.Globalization;

public struct Complex
{
```

```
    public static readonly Complex Zero = new Complex(0f, 0f);
    public static readonly Complex One = new Complex(1f, 0f);
        public static readonly Complex ImaginaryOne = new
Complex(0f, 1f);

    public float Real
    {
        get { return this.real; }
    }
    public float Imaginary
    {
        get { return this.imaginary; }
    }
    public float Magnitude
    {
        get { return Abs(this); }
    }
    public float SqrMagnitude
    {
        get { return SqrAbs(this); }
    }
    public float Phase
    {
        get { return Mathf.Atan2(this.imaginary, this.real); }
    }
    public Vector2 Vec2
    {
        get { return new Vector2(this.real, this.imaginary); }
    }

    [SerializeField]
    private float real;
    [SerializeField]
    private float imaginary;


    public Complex(Vector2 vec2) : this(vec2.x, vec2.y) { }

          public Complex(Complex other) : this(other.real,
other.imaginary) { }

    public Complex(float real, float imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    /// ...
```

```
     /// Below, the function to calculate the complex number is
implemented.
     /// Four arithmetic operations, absolute value calculation,
etc.
    /// ...
}
```

A structure for handling complex numbers.

C # has a `Complex`structure, but it has been included since .Net 4.0. At the time of writing this chapter, Unity's .Net 4.6 support was in the Experimental stage, so I decided to prepare it myself.

8.4.2 Calculation of the first three circles

As a prerequisite for calculating the Apollonian Gasket, there must be three circles tangent to each other. Therefore, in this program, three circles with randomly determined radii are generated, and the coordinates are calculated and arranged so that they touch each other.

ApollonianGaskets.cs

```
private void CreateFirstCircles(
    out Circle c1, out Circle c2, out Circle c3)
{
    var r1 = Random.Range (
        this.firstRadiusMin, this.firstRadiusMax
    );
    var r2 = Random.Range (
        this.firstRadiusMin, this.firstRadiusMax
    );
    var r3 = Random.Range(
        this.firstRadiusMin, this.firstRadiusMax
    );

    // Get random coordinates
    var p1 = this.GetRandPosInCircle(
        this.fieldRadiusMin,
        this.fieldRadiusMax
    );
    c1 = new Circle(new Complex(p1), r1);

     // Calculate the center coordinates of the tangent circle
based on p1
```

```
    var p2 = -p1.normalized * ((r1 - p1.magnitude) + r2);
    c2 = new Circle(new Complex(p2), r2);

     // Calculate the center coordinates of a circle tangent to
two circles
    var p3 = this.GetThirdVertex(p1, p2, r1 + r2, r2 + r3, r1 +
r3);
    c3 = new Circle(new Complex(p3), r3);
}

private   Vector2   GetRandPosInCircle(float   fieldMin,   float
fieldMax)
{
    // Get the right angle
    var theta = Random.Range (0f, Mathf.PI * 2f);

    // Calculate the appropriate distance
    var radius = Mathf.Sqrt(
        2f * Random.Range(
            0.5f * fieldMin * fieldMin,
            0.5f * fieldMax * fieldMax
        )
    );

    // Convert from polar coordinate system to Euclidean plane
    return new Vector2(
        radius * Mathf.Cos(theta),
        radius * Mathf.Sin(theta)
    );
}

private Vector2 GetThirdVertex(
    Vector2 p1, Vector2 p2, float rab, float rbc, float rca)
{
    var p21 = p2 - p1;

    // Calculate the angle by the cosine theorem
    var theta = Mathf.Acos(
        (rab * rab + rca * rca - rbc * rbc) / (2f * rca * rab)
    );

    // Calculate and add the angle of the starting point
     // theta is just an angle in the triangle, not an angle in
the plane
    theta += Mathf.Atan2(p21.y, p21.x);

     // Add the coordinates converted from the polar coordinate
system to the Euclidean plane to the starting coordinates.
```

```
    return p1 + new Vector2(
        rca * Mathf.Cos(theta),
        rca * Mathf.Sin(theta)
    );
}
```

`CreateFirstCircles`By calling the function, the initial condition of 3 yen is generated.

Randomly into three radial first `r1,r2,r3`decided, then `GetRandPosInCircle`the function `r1`to determine the center coordinates of a circle having a radius (hereinafter C1). This function returns random coordinates inside a circle that is `fieldMin`greater `fieldMax`than or equal to the radius of the origin center .

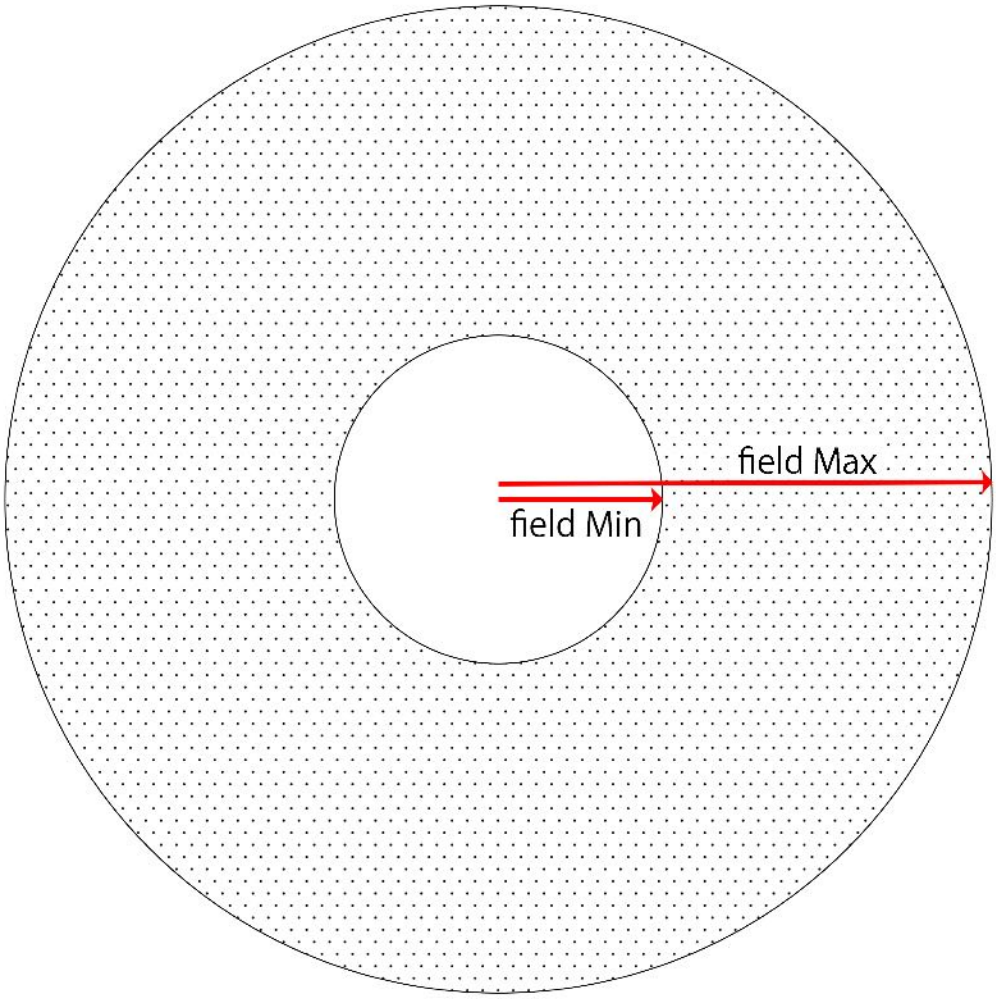

Figure 8.3: Area where random coordinates are generated

Next, `r2`the center coordinates of a circle with a radius (below C2) to calculate. First , calculate the distance from the origin to the center of C2 by (r1-p1.magnitude) + r2 . By multiplying this by the normalized coordinates of C1 which is sign-inverted, `r2`the center coordinates of a circle with a radius adjacent to C1 can be obtained.

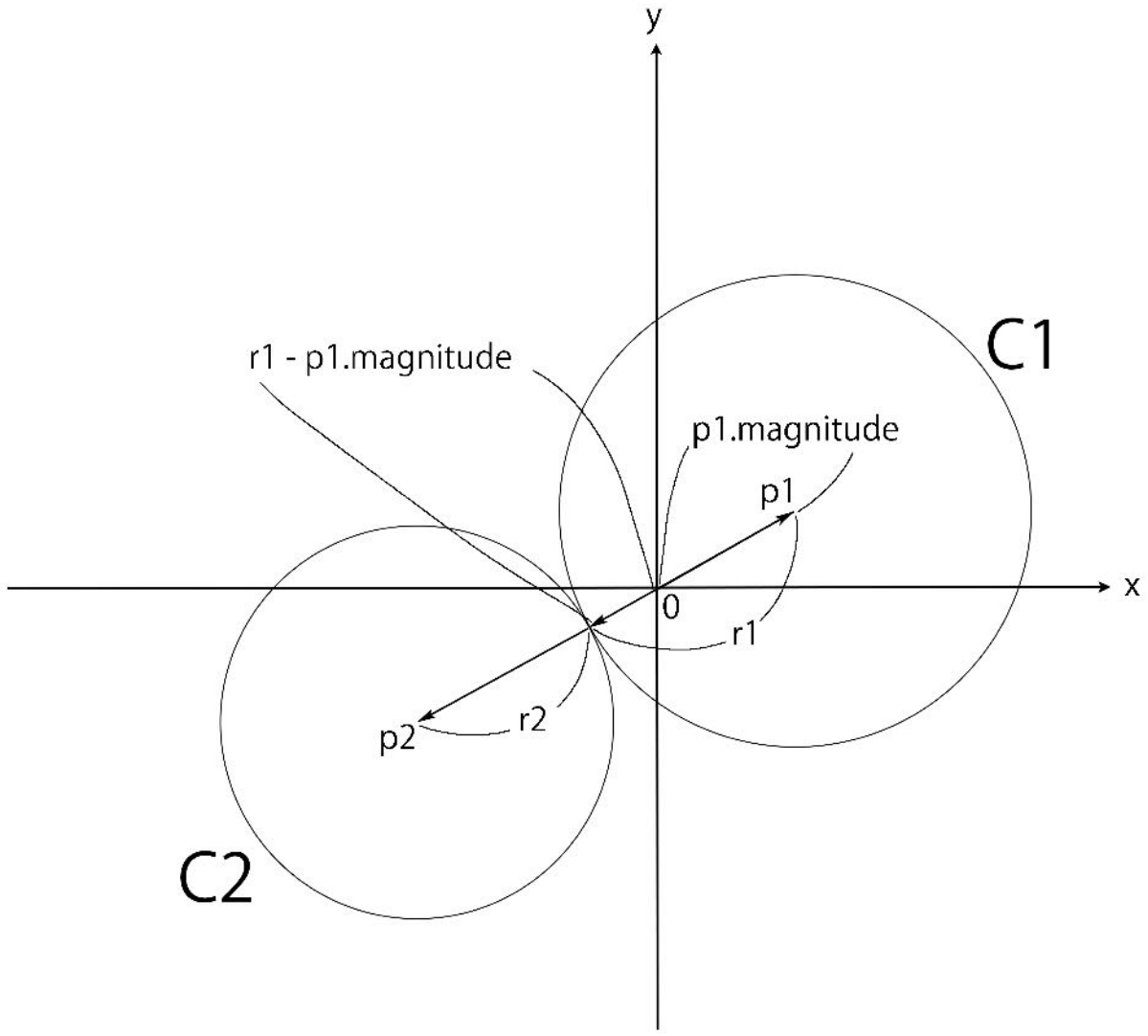

Figure 8.4: Position represented by the p2 vector

`r3`The center coordinates of the circle with the last radius (hereinafter C3) are `GetThirdVertex`calculated by the function, but this calculation uses the **cosine theorem** . As most readers may have learned in high school, the cosine theorem is a theorem that holds between the length of the sides of a triangle and the cosine of an internal angle (cos). △ in ABC, a = BC, b = CA, c = AB, alpha = ∠CAB When

```
a^2 = c^2 + b^2 - 2cbcosα
```

Is the law of cosines.

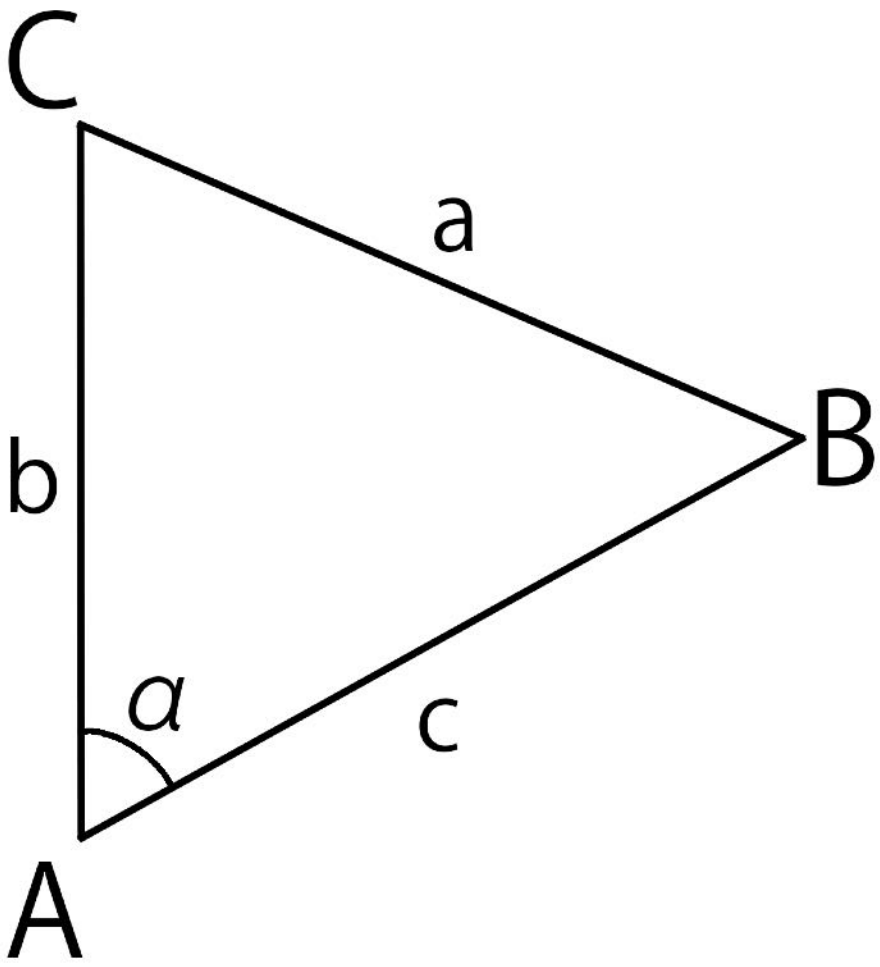

Figure 8.5: Triangle ABC

You may be wondering why you need a triangle to think about the center of a circle, but in fact, the relationship between the three circles makes it possible to think of a very easy-to-use triangle. Considering a triangle whose apex is the center of C1, C2, and C3, since these three circles are in contact with each other, the length of each side of the triangle can be known from the radius of the circle.

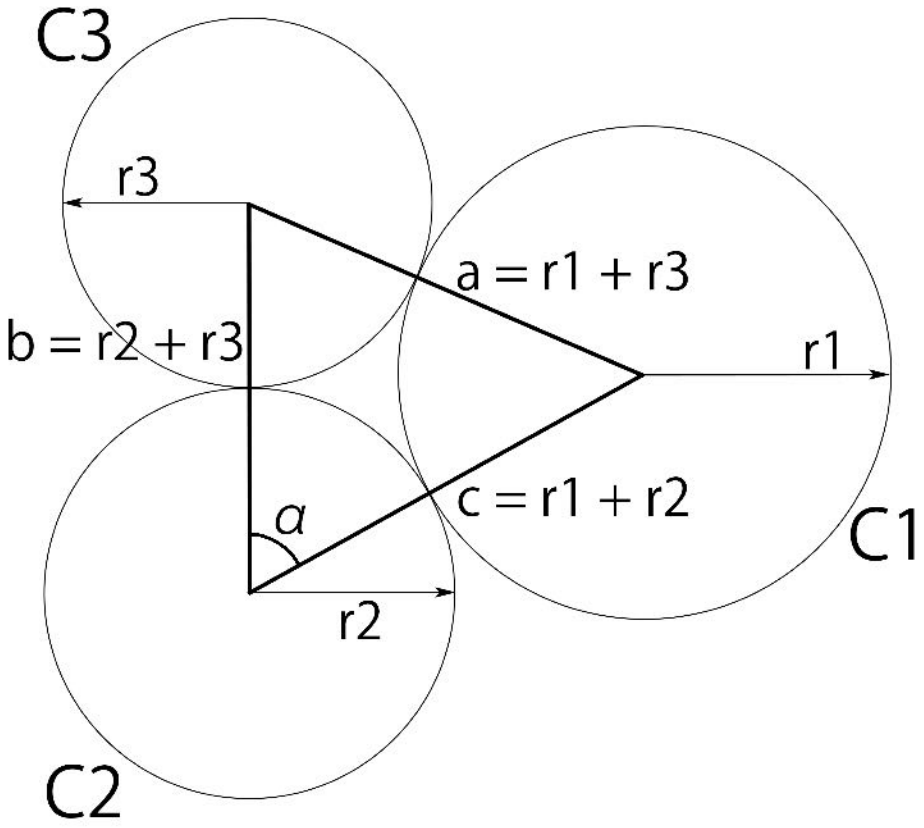

Figure 8.6: Triangle ABC and Circles C1, C2, C3

When the law of cosines is transformed

```
cosα = \frac{c^2 + b^2 - a^2}{2cb}
```

Therefore, we can solve the cosine from the lengths of the three sides. Once the angle and distance between the two sides are found, the center coordinates of C3 can be found based on the center coordinates of C1. ..

You have now generated the three tangent circles you need as an initial condition.

8.4.3 Calculation of the circle tangent to C1, C2, C3

Based on the three circles C1, C2, and C3 generated in the previous section, the circles tangent to them are calculated. Two parameters, radius and center coordinates, are required to create a new circle, so each is calculated.

radius

We first calculate from the radius, which can be calculated by **Descartes's circle theorem** . Descartes's circle theorem is that for four circles C1, C2, C3, C4 that touch each other, the curvature * 6 is k1, k2, k3, k4, respectively.

```
(k_1 + k_2 + k_3 + k_4) ^ 2 = 2 ({k_1} ^ 2 + {k_2} ^ 2 + {k_3} ^ 2 + {k_4} ^ 2)
```

Is true. This is a quadratic equation for the radii of four circles, but if you organize this equation

```
k_4 = k_1 + k_2 + k_3 \ pm 2 \ sqrt {k_1k_2 + k_2k_3 + k_3k_1}
```

[* 6] The reciprocal of the radius , defined by k = \ pm \ frac {1} {r}

If the three circles C1, C2, and C3 are known, the curvature of the fourth circle C4 can be obtained. Since the curvature is the reciprocal of the radius, the radius of the circle can be known by taking the reciprocal of the curvature.

Here, the curvature of C4 is obtained by compounding two, but one solution is always positive and the other is either positive or negative. When the curvature of C4 is positive, it is circumscribed to C1, C2, C3, and when it is negative, it is inscribed to C1, C2, C3 (including 3 circles). In other words, the fourth circle C4 can be considered in two patterns, and there is a possibility that both can be considered.

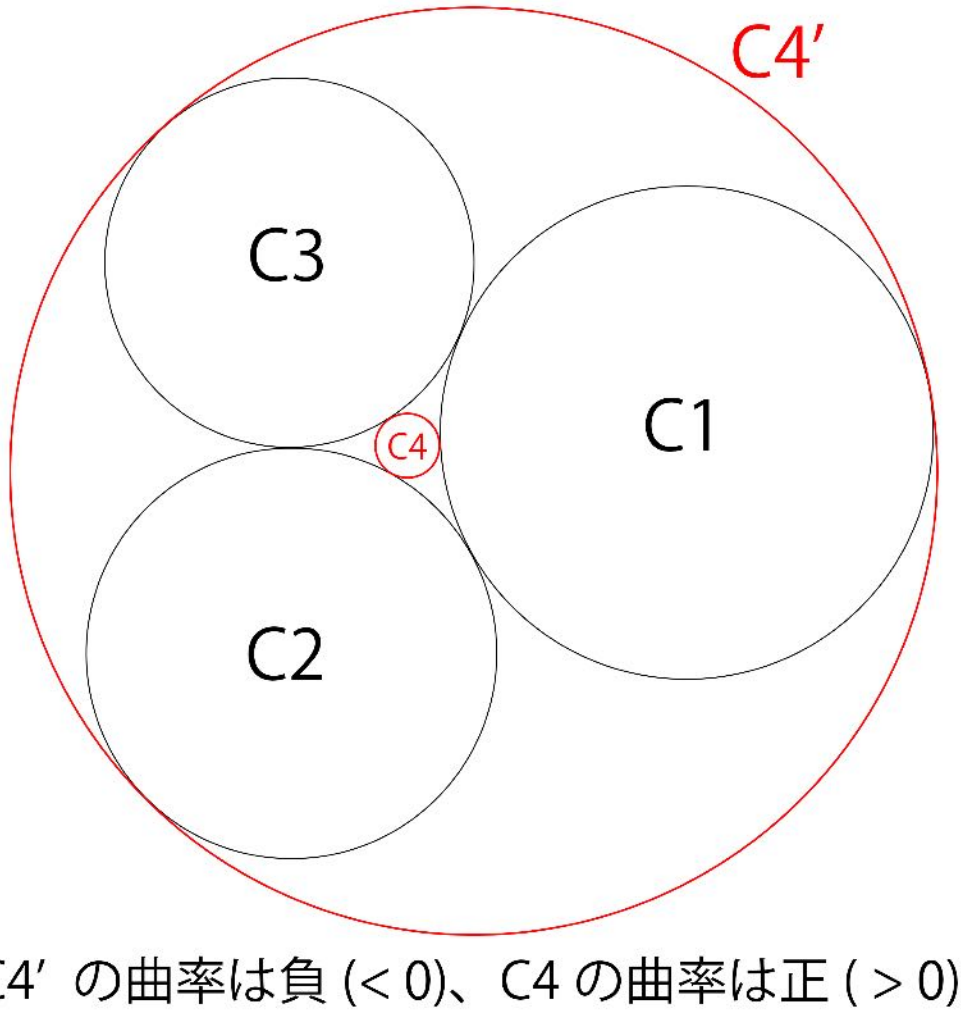

Figure 8.7: Positive and negative curvature

The following part is programming this.

SoddyCircles.cs

```
// Curvature calculation
var k1 = this.Circle1.Curvature;
var k2 = this.Circle2.Curvature;
var k3 = this.Circle3.Curvature;

var plusK = k1 + k2 + k3 + 2f * Mathf.Sqrt (k1 * k2 + k2 * k3 +
k3 * k1);
```

```
var minusK = k1 + k2 + k3 - 2f * Mathf.Sqrt (k1 * k2 + k2 * k3 +
k3 * k1);
```

This Cartesian circle theorem was later rediscovered by a chemist named Sodi, and the C1, C2, C3, and C4 circles are called Sodi's circles.

Sodi's Circle and Apollonius' Circle

In the previous section, we talked about the Circle of Apollonius, but I think some of you may have wondered what this is different from the Circle of Sodi.

The Apollonius circle is a general term for circles that solve the problems of Apollonius. Sodi's circle is a term that refers to four circles that satisfy Descartes' circle theorem.

In other words, since Sodi's circle is one of the solutions to Apollonius' problem, it is also Apollonius's circle.

Center coordinates

Next is the calculation of the center coordinates, which is calculated by the **Cartesian complex number theorem** , which has a shape similar to the **Cartesian** circle theorem . The Cartesian complex number theorem is that the center coordinates of the circles C1, C2, C3, C4 that touch each other on the complex plane are z1, z2, z3, z4, and the curvature is k1, k2, k3, k4.

```
(k_1z_1 + k_2z_2 + k_3z_3 + k_4z_4) ^ 2 = 2 ({k_1} ^ 2 {z_1} ^ 2
+ {k_2} ^ 2 {z_2} ^ 2 + {z_3} ^ 2 {z_3} ^ 2 + {k_4} ^ 2 {z_4} ^
2)
```

Is true. To organize this formula for z4

```
z4 = \frac{z_1k_1 + z_2k_2 + z_3k_3 \pm 2\sqrt{k_1k_2z_1z_2 +
k_2k_3z_2z_3 + k_3k_1z_3z_1}}{k4}
```

Since it can be transformed into, the center coordinates of the circle C4 can be obtained with this.

Here, two curvatures were obtained when calculating the radius, but two can also be obtained by compounding the Cartesian complex number theorem. However, unlike the curvature calculation, one of the two is the correct Sodi circle, so you need to determine which is correct.

The following part is programming this.

SoddyCircles.cs

```
/// Calculation of center coordinates
var ck1 = Complex.Multiply(this.Circle1.Complex, k1);
var ck2 = Complex.Multiply(this.Circle2.Complex, k2);
var ck3 = Complex.Multiply(this.Circle3.Complex, k3);

var plusZ = ck1 + ck2 + ck3
    + Complex.Multiply(Complex.Sqrt(ck1 * ck2 + ck2 * ck3 + ck3
* ck1), 2f);
var minusZ = ck1 + ck2 + ck3
    - Complex.Multiply(Complex.Sqrt(ck1 * ck2 + ck2 * ck3 + ck3
* ck1), 2f);

var recPlusK = 1f / plusK;
var recMinusK = 1f / minusK;

// Judgment of Sodi's circle
this.GetGasket(
    new Circle(Complex.Divide(plusZ, plusK), recPlusK),
    new Circle(Complex.Divide(minusZ, plusK), recPlusK),
    out c4
);

this.GetGasket(
    new Circle(Complex.Divide(plusZ, minusK), recMinusK),
    new Circle(Complex.Divide(minusZ, minusK), recMinusK),
    out c5
);
```

SoddyCircles.cs

```
/// Judgment of Sodi's circle
(c1.IsCircumscribed(c4, CalculationAccuracy)
    || c1.IsInscribed(c4, CalculationAccuracy)) &&
(c2.IsCircumscribed(c4, CalculationAccuracy)
    || c2.IsInscribed(c4, CalculationAccuracy)) &&
(c3.IsCircumscribed(c4, CalculationAccuracy)
    || c3.IsInscribed(c4, CalculationAccuracy))
```

Circle.cs

```
public bool IsCircumscribed(Circle c, float accuracy)
{
    var d = (this.Position - c.Position).sqrMagnitude;
     var abs = Mathf.Abs(d - Mathf.Pow(this.Radius + c.Radius,
2));

    return abs <= accuracy * accuracy;
}

public bool IsInscribed(Circle c, float accuracy)
{
    var d = (this.Position - c.Position).sqrMagnitude;
     var abs = Mathf.Abs(d - Mathf.Pow(this.Radius - c.Radius,
2));

    return abs <= accuracy * accuracy;
}
```

Now, based on the initial conditions C1, C2, C3, we have obtained two circles tangent to them (hereinafter C4, C5).

8.4.4 Apollonian Gasket Calculation

At this point, you can easily calculate the Apollonian Gasket. Simply repeat the calculation performed in "8.4.3 Calculation of the circle tangent to C1, C2, C3" .

In the previous section, we found the circles C4 and C5 that are tangent to C1, C2 and C3 found in "8.4.2 Calculation of the first three circles" . Next, find the circles that touch (C1, C2, C4) (C1, C2, C5) (C2, C3, C4) (C2, C3, C5) (C3, C1, C4) (C3, C1, C5). I will continue.

Here, even if the circles are in contact with each other on the combination, they may actually overlap with other circles. Therefore, after determining whether it is the correct Sodi circle, it is also necessary to confirm that it does not overlap all the circles that have been requested so far.

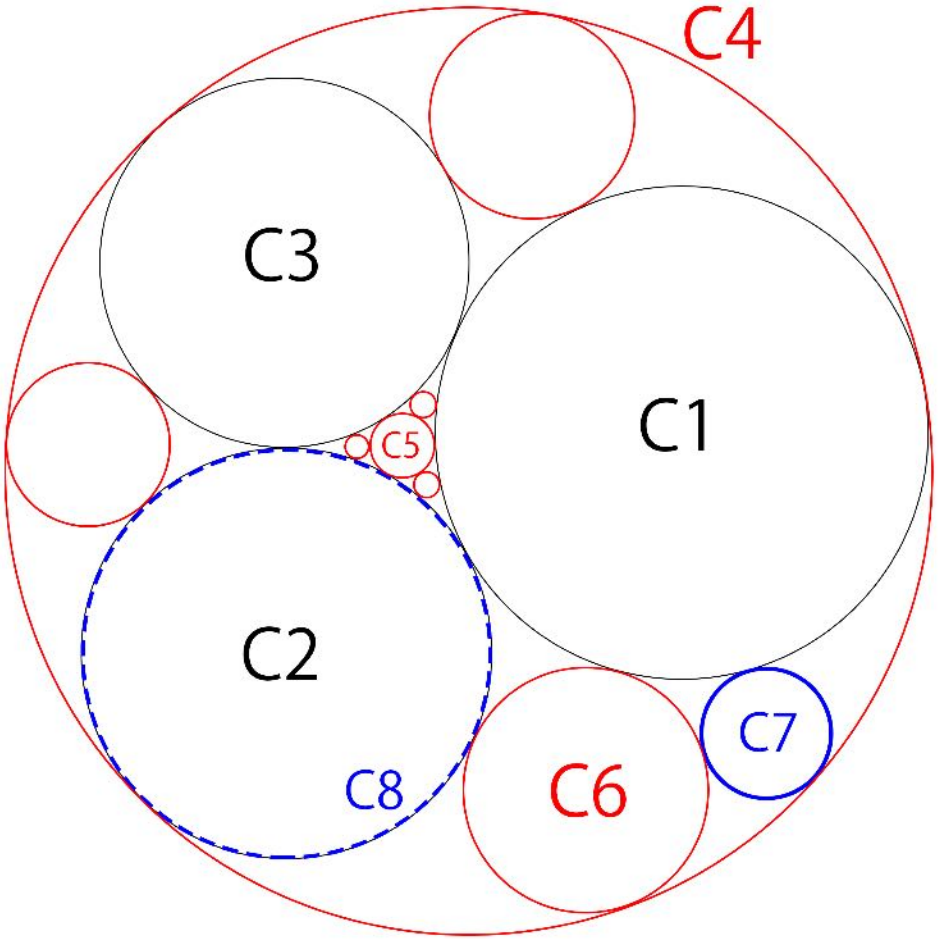

Figure 8.8: Of the circles C7 and C8 tangent to C1, C4 and C6, C8 overlaps C2 and is not included in the Apollonian Gasket.

Then you will get a new circle that touches each one. After that, in the same way, we will continue to find new tangent circles for all combinations of the original circle to find the tangent circle and the newly found tangent circle.

Mathematically, the set of circles obtained by repeating this procedure infinitely is the Apollonian Gasket, but it is not possible to handle the program infinitely. Therefore, in this program, if the radius of the newly

obtained tangent circle is less than a certain value, the condition that the processing is completed is given for the combination.

The following part is programming this.

ApollonianGaskets.cs

```
private void Awake()
{
    // Generate the initial condition of 3 yen
    Circle c1, c2, c3;
    this.CreateFirstCircles(out c1, out c2, out c3);
    this.circles.Add(c1);
    this.circles.Add(c2);
    this.circles.Add(c3);

    this.soddys.Enqueue(new SoddyCircles(c1, c2, c3));

    while(this.soddys.Count > 0)
    {
        // Calculate Sodi's circle
        var soddy = this.soddys.Dequeue();

        Circle c4, c5;
        soddy.GetApollonianGaskets(out c4, out c5);

        this.AddCircle(c4, soddy);
        this.AddCircle(c5, soddy);
    }
}

private void AddCircle(Circle c, SoddyCircles soddy)
{
    if(c == null || c.Radius <= MinimumRadius)
    {
        return;
    }
      // If the curvature is negative, no questions asked and
added
    // Circles with negative curvature appear only once
    else if(c.Curvature < 0f)
    {
        this.circles.Add(c);
                        soddy.GetSoddyCircles(c).ForEach(s   =>
this.soddys.Enqueue(s));

        return;
```

```
    }

    // Check if it covers other circles
    for(var i = 0; i < this.circles.Count; i++)
    {
        var o = this.circles[i];

        if(o.Curvature < 0f)
        {
            continue;
        }
        else if(o.IsMatch(c, CalculationAccuracy) == true)
        {
            return;
        }
    }

    this.circles.Add(c);
                        soddy.GetSoddyCircles(c).ForEach(s      =>
this.soddys.Enqueue(s));
}
```

You have now successfully requested the Apollonian Gasket.

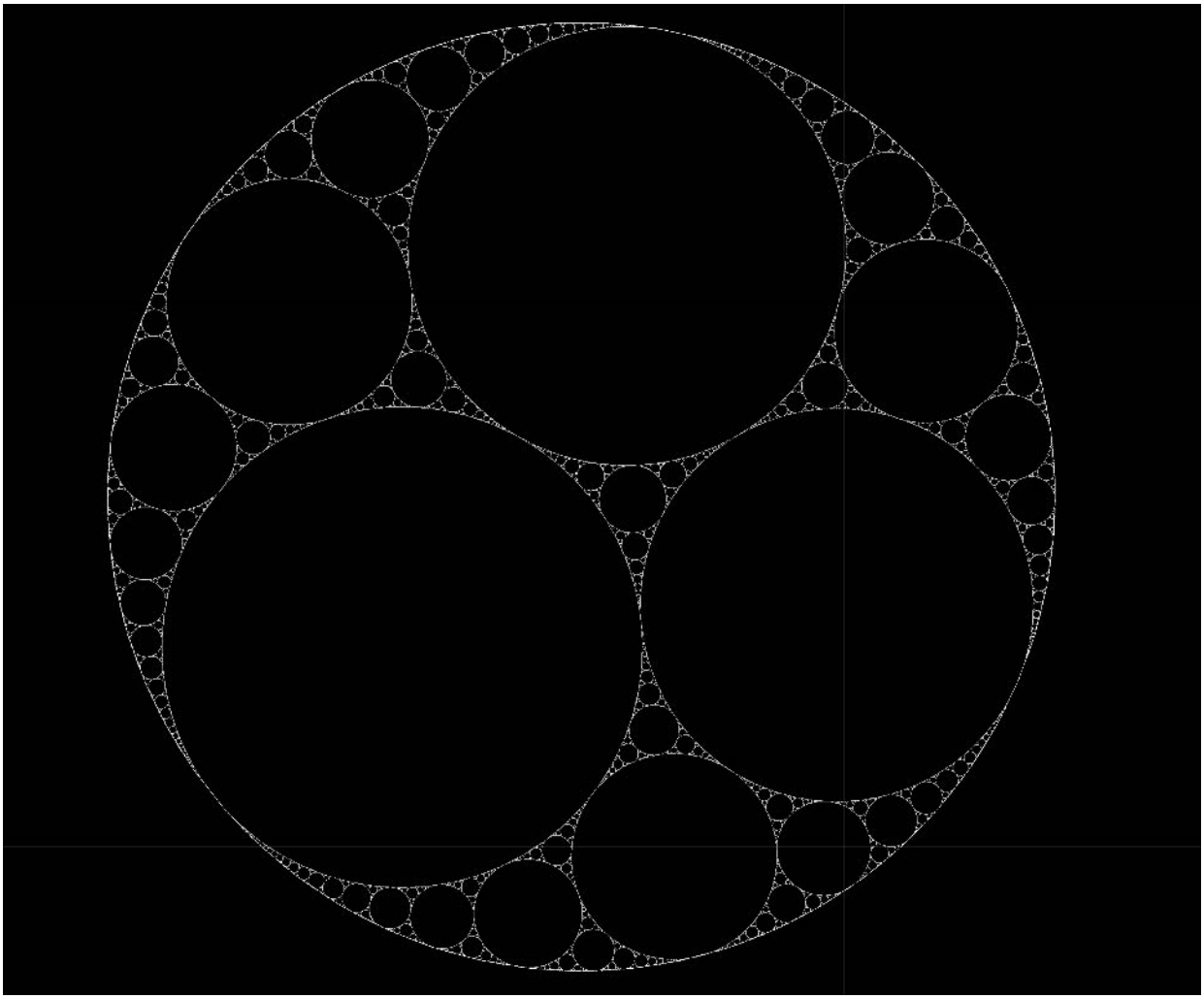

Figure 8.9: Execution result on Unity

8.5 Summary

So far, we have walked through the steps required to calculate the Apollonian Gasket. As I explained at the beginning, the Apollonian Gasket has a stronger meaning as a fractal figure.

However, if we remove the limitation of the plane this time and jump out into the world of space, it will become difficult to talk about it, and the meaning of filling (packing) from the fractal figure will become stronger. The proposition of sphere-packing space is a field that has been controversial for hundreds of years, including the existence of famous mathematical conjectures such as the Kepler conjecture.

The Space filling problem is also useful in practical terms. It is applied in a wide range of fields such as optimization of VLSI layout design, optimization of cutting out parts such as cloth, and automation and optimization of UV development.

This time, I chose the Apollonian Gasket, which is relatively easy to understand and interesting. If you are interested in packing itself, check out the algorithms introduced at the beginning.

Fill the inside of the object with the object. I think it can be used as a new expression method in unexpected places.

8.6 Reference

- https://ja.wikipedia.org/wiki/ Apollonian Gasket
- https://ja.wikipedia.org/wiki/ Descartes' Circle Theorem
- http://paulbourke.net/fractals/randomtile/

Chapter 9 ImageEffect Getting Started

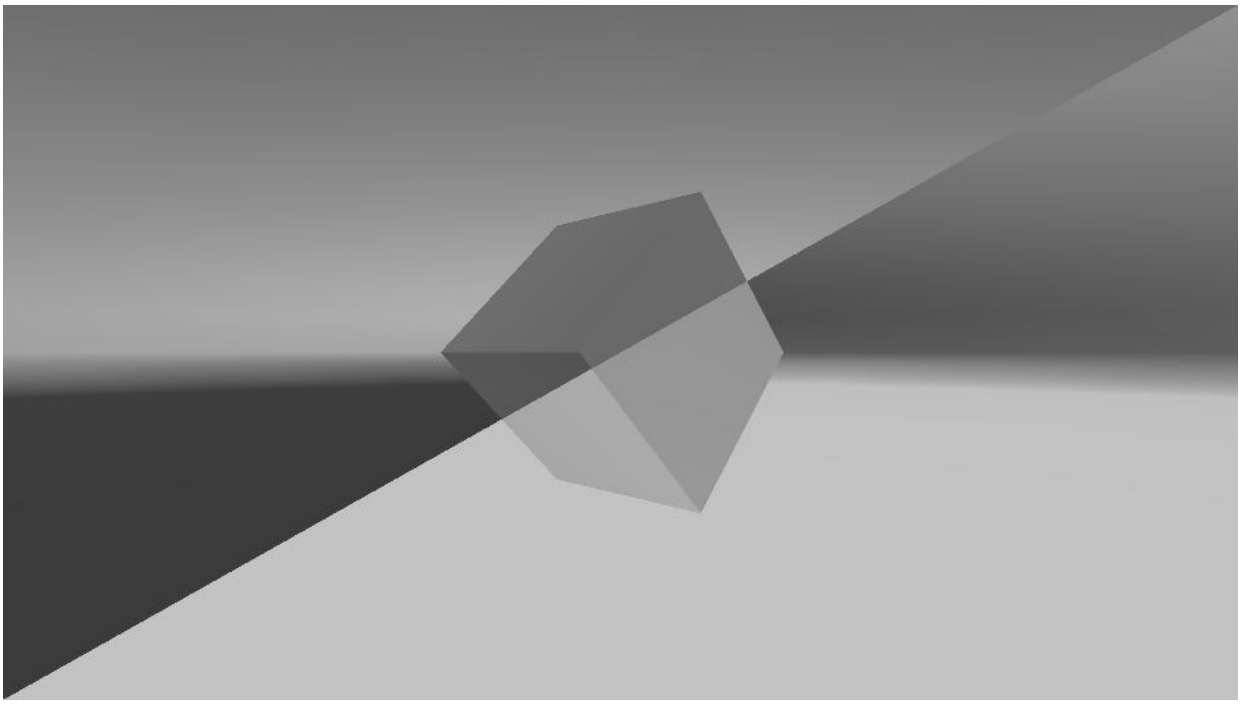

Figure 9.1: Negative-Positive Inversion with ImageEffect

A simple explanation of how to implement ImageEffect, a technology that applies effects to the output video using a shader (GPU), in Unity. The technology is also known as PostEffect.

ImageEffect is used for glow effects that express light, anti-aliasing that reduces jaggies, depth of field DOF, and much more. The simplest example would be a color change or modification that also deals with the sample presented here.

This chapter is written on the assumption that you have some prerequisite knowledge about the basic knowledge and usage of Unlit shader and Surface shader, but since it is the shader with the simplest configuration, even if you do not have the prerequisite knowledge, I think you can read on and use it.

The sample in this chapter is "Simple Image Effect" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming2
.

9.1 How ImageEffect works

The way ImageEffect achieves various effects is, in a nutshell, image processing, that is, by manipulating the screen pixel by pixel, various effects are achieved.

Speaking of processing pixels by shaders, it is a fragment shader. In essence, implementing an ImageEffect is equivalent to implementing a fragment shader.

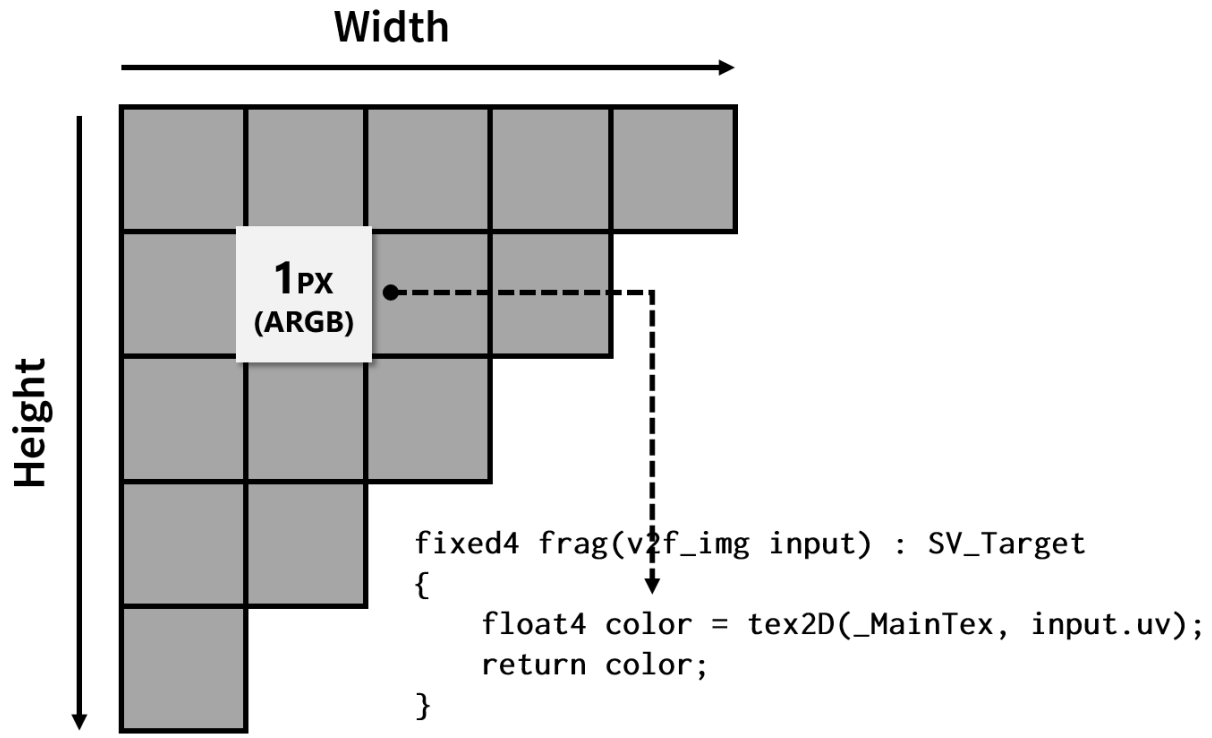

Figure 9.2: ImageEffect implementation implements fragment shader

9.2 Simple flow of ImageEffect in Unity

In Unity, the processing order of ImageEffect is roughly as follows.

1. The camera draws the scene.
2. The drawing contents of the camera are input to the OnRenderImage method.
3. In the shader for ImageEffect, modify the input drawing contents.
4. The OnRenderImage method outputs the modified drawing content.

9.3 Check the configuration from the sample scene

We have prepared the simplest sample scene. Open the sample "ImageEffectBase" scene to see it. The associated script and other resources have the same name.

A similar sample has a resource with the same name as the ImageEffect scene, but be aware that it will be discussed later.

When you open the sample, the image projected by the camera in the scene will be negatively and positively inverted by ImageEffect. This is equivalent to the shader for ImageEffect that Unity generates by default, but the actual source code is slightly different.

Make sure the "ImageEffectBase" script is attached to the "Main Camera" in the sample scene. In addition, "ImageEffectBase" references a material with the same name, and that material has a shader with the same name.

9.4 Script implementation

First of all, I will explain the process from calling the Shader of ImageEffect from the script.

9.4.1 OnRenderImage method

When you want to make changes to the video that Unity outputs, you almost always need to implement the OnRenderImage method. OnRenderImage is a method defined in Unity's standard workflow, like Start and Update.

ImageEffectBase.cs

```
[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
public class ImageEffectBase : MonoBehaviour
{
…
protected virtual void OnRenderImage
    (RenderTexture source, RenderTexture destination)
{
    Graphics.Blit(source, destination, this.material);
}
```

OnRenderImage is only called when it is added to a GameObject that has a Camera component. Therefore, the ImageEffect class `[RequireComponent(typeof(Camera))]`defines.

The `ExcludeInEditMode`attributes are also defined because the result of applying ImageEffect should be visible before running Scene . Disable the ImageEffect script when you want to switch between multiple ImageEffects and check when they are disabled.

About the arguments source and destination

OnRenderImage is given an input in the first argument (source) and an output destination in the second (destination). Both are of type RenderTexture, but unless otherwise specified, source is given the drawing result of the camera and destination is given null.

ImageEffect modifies the picture entered in source and writes it to destination, but when destination is null, the modified picture is output to the framebuffer, the area visible to the display.

Also, when RenderTexture is set to the output destination of the Camera, the source is equivalent to that RenderTexture.

Graphics.Blit

`Graphics.Blit`The method is the process of drawing the input RenderTexture to the output RenderTexture using the specified material and

shader. The inputs and outputs here are the source and destination of the OnRenderImage. Also, the material will be the one with the shader set for ImageEffect.

As a general rule, the OnRenderImage method must always pass some image data to the destination argument. Therefore, in most cases Graphics.Blit is called within OnRenderImage.

It `Graphics.Blit`may also be used as an application, for example, when creating a texture for use in another effect, or when duplicating a texture . Alternatively, you may use another method to pass the data to the destination, but I'll omit those application examples here for the sake of getting started.

The following items are a little different from the process of applying ImageEffect, so if you are reading for the first time, it is recommended that you skip to the shader description.

9.4.2 Verify if ImageEffect is available

I don't think it is necessary to implement or explain this item when explaining ImageEffect, but I decided to explain it so that it would not be an obstacle when reading materials with more practical implementations. Equivalent functionality is implemented in the ImageEffect documentation provided by Unity.

ImageEffect is a process that is calculated for each pixel. Therefore, in an execution environment without an advanced GPU, ImageEffect may not be welcomed due to the large number of operations. Therefore, it is helpful to verify at the start whether ImageEffect is available in the execution environment and disable it if it is not available.

ImageEffectBase.cs

```
protected virtual void Start()
{
    if (!SystemInfo.supportsImageEffects
     || !this.material
     || !this.material.shader.isSupported)
```

```
    {
        base.enabled = false;
    }
}
```

Verification `SystemInfo.supportsImageEffects`can be easily achieved by providing Unity .

This implementation will be useful in most cases, but you may need a different implementation, for example when using the fallback feature implemented on the shader side. Please refer to it to the last.

The only `this.material`thing you need to be aware of is when to validate the reference. The example validates with the Start method, but when this is Awake or OnEnable `this.material`, Unity will show null (and `base.enabled = false`will invalidate the script) , even if a reference is given to , for example . Details are omitted, but `ExcludeInEditMode`it depends on the specifications (it is hard to say that it is harmful).

9.5 Simplest ImageEffect Shader Implementation

Next, I will explain about the ImageEffect shader. The most basic sample presented here implements the effect of just flipping the output colors, similar to what Unity creates as standard.

ImageEffectBase.shader

```
Shader "ImageEffectBase"
{
    Properties
    {
        _MainTex("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            CGPROGRAM
```

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

            sampler2D _MainTex;

            fixed4 frag(v2f_img input) : SV_Target
            {
                float4 color = tex2D(_MainTex, input.uv);
                color.rgb = 1 - color.rgb;

                return color;
            }

            ENDCG
        }
    }
}
```

As a rough process flow, `_MainTex`the image drawn by the camera is input to, and the fragment shader determines the final color to be displayed on the pixel.

Here `_MainTex`texture information given to `OnRenderImage`the `source`, `Graphics.Blit`the `source`is equal to.

`_MainTex`Tooth `Graphics.Blit`Please note that has been reserved by the Unity for input. If you change to a different other name, `Graphics.Blit`the `source`is not entered correctly shader.

9.5.1 Differences from the standard shaders generated by Unity

The ImageEffect that Unity generates by default is a bit long and complex (excerpt): ImageEffect is also a shader, so you get the final output through a standard rendering pipeline. Therefore, a vertex shader that does not seem to affect the effect that ImageEffect achieves must also be defined in the ImageEffect shader.

NewImageEffectShader.shader

```
SubShader
{
    Cull Off ZWrite Off ZTest Always

    Pass
    {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.uv;
            return o;
        }

        sampler2D _MainTex;

        fixed4 frag (v2f i) : SV_Target
        {
            fixed4 col = tex2D(_MainTex, i.uv);
            col.rgb = 1 - col.rgb;
            return col;
        }
        ENDCG
    }
}
```

The vertex shader in ImageEffect simply faces the camera and passes a rectangular mesh that fills the entire surface and its UV coordinates to the

fragment shader. There are some benefits that can be achieved by modifying this vertex shader, but most ImageEffects do not.

That's why Unity provides a standard vertex shader and a structure to define its inputs. They are defined in "UnityCG.cginc". Here, in the source code of the shader is not a prepared standard, defined in the UnityCg.cginc `vertex vert_img`Ya `appdata`, `v2f_img`by making use of, and to simplify the entire source code.

9.5.2 Cull, ZWrite, ZTest

At first glance, standard values seem to be fine for culling, writing and referencing the Z-buffer. However, Unity `Cull Off ZWrite Off ZTest Always`recommends defining to prevent inadvertent writing to the Z-buffer .

9.6 The easiest practice

Let's practice ImageEffect easily. The sample simply flips the full screen negatively and positively, but try applying negative and positive flipping "only to the diagonal half" of the entire image, as shown in the figure at the beginning of this chapter.

`input.uv`Is given coordinates that indicate one pixel of the entire image, so take advantage of this. Each pixel in the entire image is represented by the x * y coordinates normalized by 0 to 1.

An example code that works is included in the sample "Prtactice" folder and will be explained later, but if you are new to it, try implementing it yourself first. I recommend that.

9.6.1 Easy up, down, left and right halves

It's very easy to change the color in the upper and lower halves. This is a good way to see the origin of the ImageEffect's coordinates. For example, the following two lines of code invert colors when the x and y coordinates are less than half, respectively.

Practice/ImageEffectShader_01.shader

```
color.rgb = input.uv.x < 0.5 ? 1 - color.rgb : color.rgb;
color.rgb = input.uv.y < 0.5 ? 1 - color.rgb : color.rgb;
```

Did you confirm from the color change that the origin of the coordinates given to ImageEffect is the lower left?

9.6.2 Easy diagonal half

I mentioned earlier that the top, bottom, left, and right halves are easy, but in reality, the diagonal halves are also easy. You can apply the effect (invert the color) diagonally in half with the following source code.

Practice/ImageEffectShader_02.shader

```
color.rgb = input.uv.y < input.uv.x ? 1 - color.rgb : color.rgb;
```

9.7 Convenient definition values for coordinates

That was introduced as UnityCg.cginc The `vertex vert_img`killing `appdata`of such useful functions and structures have been defined, convenient value in implementing the ImageEffect In addition to these have been defined.

9.7.1 _ScreenParams

`_ScreenParams`Is `float4`the type of value, `x, y`to the pixel width and height of the image to be output, respectively, `w, z`the `1 + 1 / x`, `1 + 1 / y`we are given.

For example, when you run the rendering of 640x480 size, `x = 640`, `y = 480`, `z = 1 + 1 / 640`, `z = 1 + 1 / 480`and will be. As a matter of fact, `w`and `z`it would not have to use so much.

On the other hand `x`, `y`the value of is often used to calculate, for example, how many pixels on an image it corresponds to, or to calculate the aspect ratio. These are important for creating elaborate effects, but it would be

helpful if Unity provided them without giving any values from the script. If you put it in the corner of your head, it may help you to read other shaders.

9.7.2 _TexelSize

One of the similar `<sampler2Dの変数名>_TexelSize`definition values is. Here it `_MainTex_TexelSize`will be.

`_ScreenParams`When the same `float4`, but the type of values, `x = 1 / width`, `y = 1 / height`, `z = width`, `y = height`and, different value given to each element. Another `sampler2D`feature is that the values differ depending on the corresponding type. `_MainTex`Regardless, `~_TexlSize`if you define a corresponding , Unity will give you a value.

`_ScreenParams`There are many ImageEffects that `_MainTex_TexelSize`use, but I think it 's easier to use.

9.7.3 Refer to one pixel

For example, it is often the case in image processing that you want to refer to the color (value) of the next pixel, but you can refer to the value of the next pixel with the following code.

Practice/ImageEffectShader_03.shader

```
sampler2D _MainTex;
float4    _MainTex_TexelSize;

fixed4 frag(v2f_img input) : SV_Target
{
    float4 color = tex2D(_MainTex, input.uv);

                color    +=    tex2D(_MainTex,    input.uv    +
float2(_MainTex_TexelSize.x, 0));
                color    +=    tex2D(_MainTex,    input.uv    -
float2(_MainTex_TexelSize.x, 0));
          color   +=   tex2D(_MainTex,   input.uv   +   float2(0,
_MainTex_TexelSize.y));
          color   +=   tex2D(_MainTex,   input.uv   -   float2(0,
_MainTex_TexelSize.y));
```

```
    color = color / 5;

    return color;
}
```

This code references the four surrounding pixels and returns the average value. In image processing, it is literally called a smoothing filter. In addition, a higher quality noise reduction filter may be implemented by referring to the surrounding pixels in the same way, and it is also used in edge / contour detection filters, for example.

9.8 Obtaining depth and normals

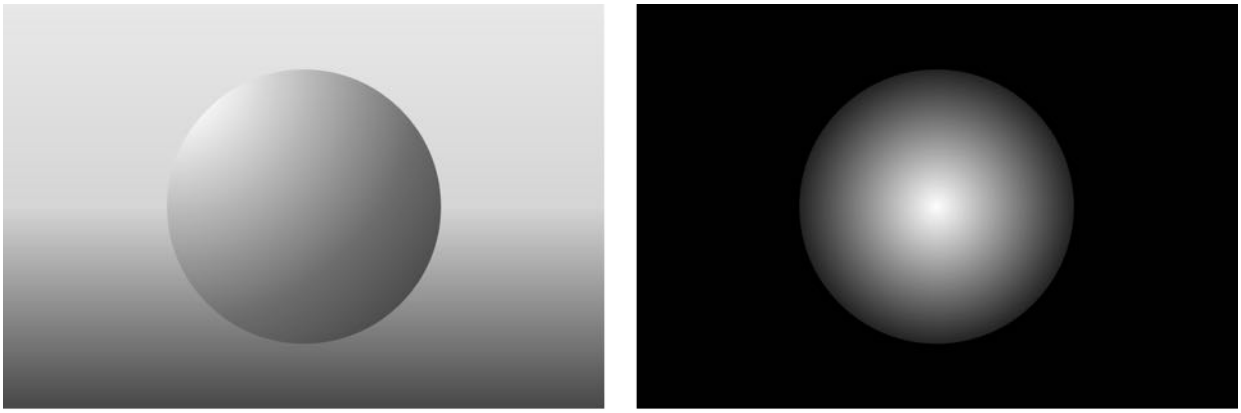

Figure 9.3: Image of G-Buffer

When implementing a material (shader) to apply to a model, you will often refer to the model's depth and normal information. ImageEffect, which manipulates two-dimensional image information, does not seem to be able to acquire depth and normal information, but there is a method to acquire the depth and normal information of an object projected on a certain pixel on the image. I have.

To explain the technical details, it is necessary to explain the rendering pipeline, which will be a little long, so let me omit it. Briefly, depth information and normal information corresponding to a pixel on the image to be drawn can be buffered. Those buffers are called G-Buffers. Some G-

Buffers store colors and depths. (By the way, the original paper shows that the reading of G-Buffer is "game buffer".)

When drawing an object, the depth and normal information is also written in the buffer, and it is referenced by ImageEffect, which is executed at the end of drawing. This technique plays an important role in Deffered rendering, but it can also be used in Forward rendering.

These discussions use a sample "ImageEffect" scene and a resource with the same name.

9.8.1 Settings for getting depth and normal information

A little setting is required to refer to the depth and normal information in ImageEffect. Since the basic functions are common, here we will set it in ImageEffect.cs, which inherits ImageEffectBase.cs.

ImageEffect.cs

```
public class ImageEffect : ImageEffectBase
{
    protected new Camera camera;
    public DepthTextureMode depthTextureMode;

    protected override void Start()
    {
        base.Start();

        this.camera = base.GetComponent<Camera>();
        this.camera.depthTextureMode = this.depthTextureMode;
    }

    protected virtual void OnValidate()
    {
        if (this.camera != null)
        {
                                    this.camera.depthTextureMode   =
this.depthTextureMode;
        }
    }
}
```

To get the depth and normal information, `DepthTextureMode`you need to set the camera . This is a setting to control how information such as depth and normal is written. The initial value is `None`.

Unfortunately, it `DepthTextureMode`'s a parameter that doesn't appear in the camera's Inspector, so you'll need to optionally get a camera reference from the script and set it.

`OnValidate` For those who haven't used the method very often, it is the method that is called when the parameter is updated on the Inspector.

9.8.2 DepthTextureMode value

Use the code presented here `DepthTextureMode`to change the value of inspector. There are some values, but `DepthNormals`note that we use here .

`Depth`If is set, it will be the setting to acquire only the depth information. However `Depth`theft `DepthNormals`door, the slightly different procedure to obtain the depth information from the shader. Also `MotionVectors`by setting the, How can a lot of fun can get the information of the motion corresponding to each pixel, a little because the longer and all commentary, please let omitted in this place.

9.9 Getting depth and normals on shaders

`DepthTextureMode`Here's how to get depth and normal information from the shader when set to camera :

`_CameraDepthNormalsTexture`Is `_MainTex`given depth and normal information, just as the image to draw is given `sampler2D`. Therefore `input.uv`, you can use to get the depth and normal information for a pixel with an image to draw.

ImageEffect.shader

```
sampler2D _MainTex;
sampler2D _CameraDepthNormalsTexture;
```

```
fixed4 frag(v2f_img input) : SV_Target
{
    float4 color = tex2D(_MainTex, input.uv);
    float3 normal;
    float  depth;

    DecodeDepthNormal
         (tex2D(_CameraDepthNormalsTexture,  input.uv),  depth,
normal);

    depth = Linear01Depth(depth);
    return fixed4(depth, depth, depth, 1);

    return fixed4(normal.xyz, 1);
}
```

`_CameraDepthNormalsTexture`The values that can be obtained from are the sum of the depth and normal values, so we need to decompose them into their respective values. The function for decomposing is the one provided by Unity. `DecodeDepthNormal`Give the function a variable to assign the value you want to decompose and the result.

9.9.1 Acquisition and visualization of depth information

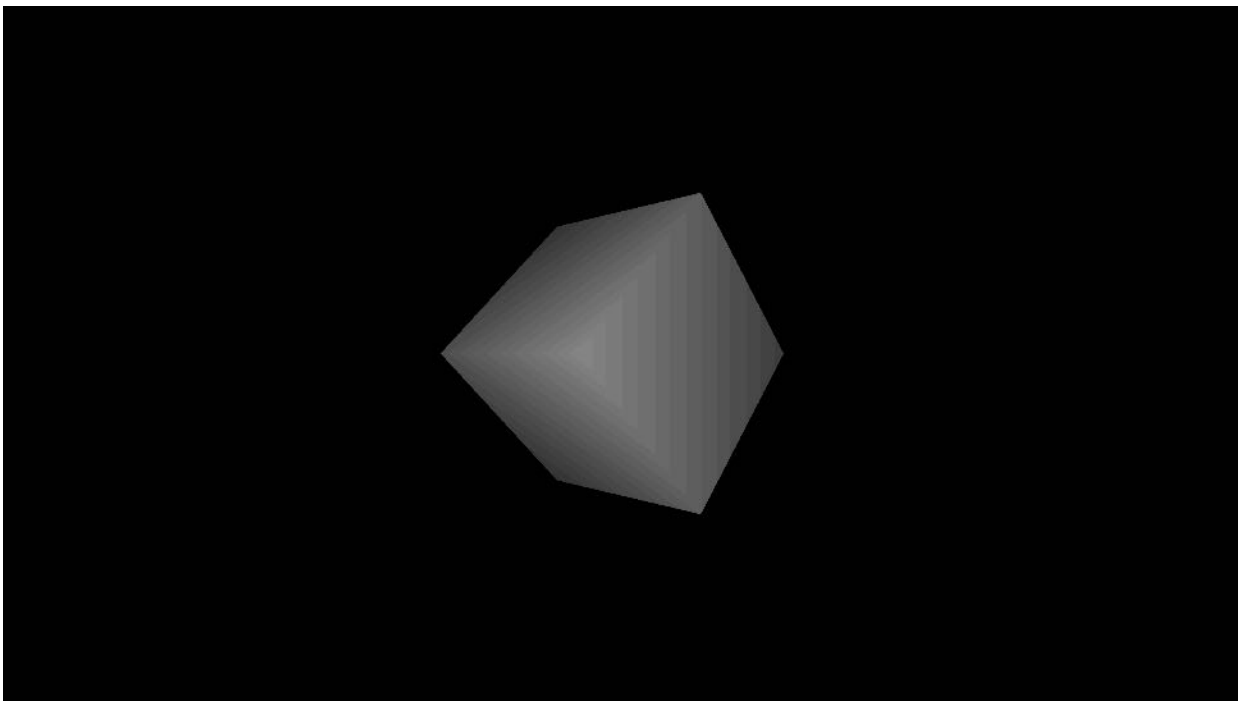

Figure 9.4: Depth visualization with ImageEffect

I will explain the depth information first. Depth information is actually handled differently depending on the platform. Unity provides some mechanisms to absorb the difference `Linear01Depth`, but I think it's better to use a function when implementing ImageEffect . `Linear01Depth`Is a function to normalize the obtained depth value from 0 to 1.

In the sample, the acquired depth value is given to R, G, and B to visualize the depth value. `Clipping Planes`It is recommended to move the camera in the scene or change the value from the Inspector to see how it changes.

9.9.2 Visualization of normal information

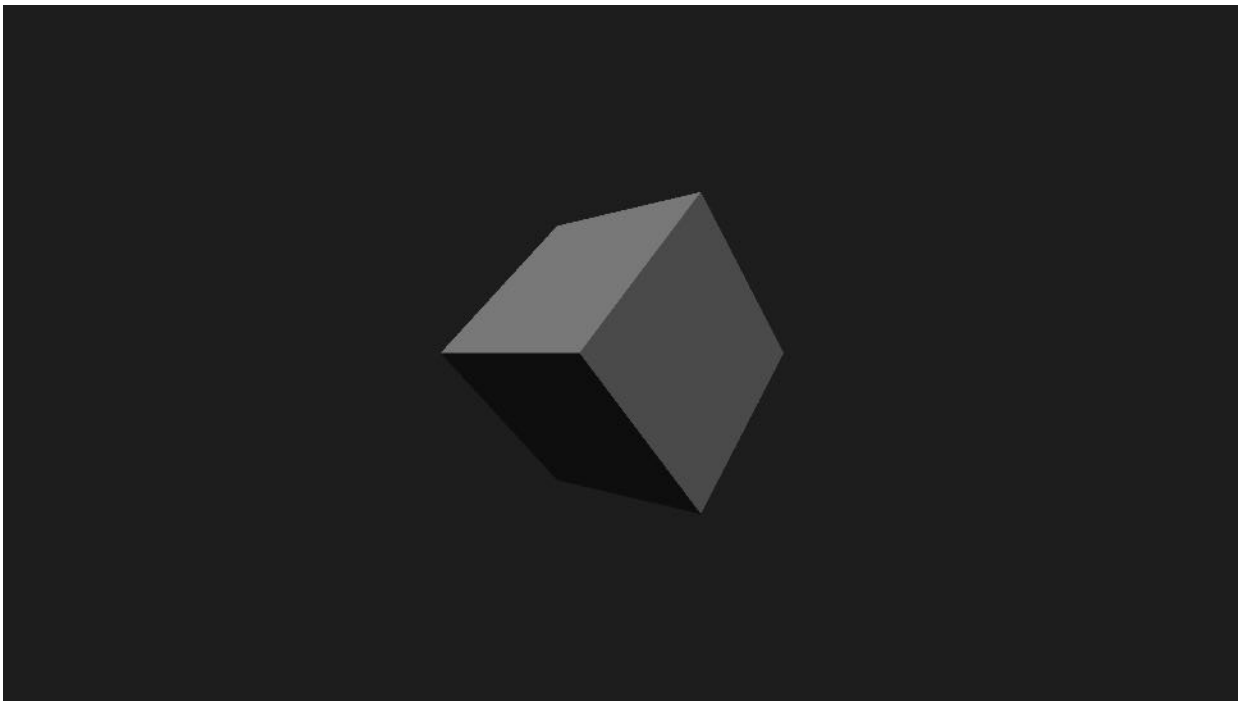

Figure 9.5: ImageEffect Visualization of Normals

Visualization of normal information is not as complicated as depth information. The normal information is equivalent to that referenced by scripts and common shaders. X, YZ information indicating the direction of the surface projected on a pixel is given in a format normalized to 0 to 1.

If you just want to check if the normals are obtained correctly, you can output the values of X, Y, Z as R, G, B as they are. In other words, the face facing to the right has a larger value of X = R and becomes more red, and the face facing upward has a value of Y = G and becomes greener.

9.10 Reference

The main references in this chapter are: Both are official Unity.

- Writing Image Effects- https://docs.unity3d.com/540/Documentation/Manual/WritingImageEffects.html
- Accessing shader properties in Cg/HLSL - https://docs.unity3d.com/Manual/SL-PropertiesInPrograms.html
- Using Depth Textures - https://docs.unity3d.com/ja/current/Manual/SL-DepthTextures.html

Chapter 10 Application of ImageEffect (SSR)

10.1 Introduction

This chapter describes the theory and implementation of Screen Space Reflection as an application of ImageEffect. When constructing a three-dimensional space, reflections and reflections are useful for expressing reality along with shadows. However, despite the simplicity of the phenomena we see in our daily lives, reflections and reflections are enormous calculations when trying to faithfully reproduce physical phenomena using ray tracing (described later) in the world of 3DCG. It is also an expression that requires quantity. Recently, Octan Renderer has become available in Unity, and when producing as a video work, it has become possible to produce quite photorealistic effects in Unity, but in real-time rendering it is still necessary to devise a pseudo reproduction. There is.

There are several techniques for expressing reflections with real-time rendering, but in this chapter we will introduce a technique called Screen Space Reflection (SSR) that belongs to the post-effects.

As for the structure of this chapter, we will first explain the blur processing used in the sample program in advance as a shoulder break-in for post effects. After that, I will explain SSR while breaking it down into the smallest possible processing units.

In addition, the sample of this chapter is in "SSR" of
https://github.com/IndieVisualLab/UnityGraphicsProgramming2
.

10.2 Blur

In this section, we will explain the blur processing. If you include anti-aliasing, you need to understand the procedure that blurring is very complicated, but this time it is a basic process because it is a shoulder break-in. The basis of blur processing is to homogenize the color of texels by multiplying each texel (pixels after rasterization * 4) of the image to be processed by a matrix that refers to the texels around it. I will continue. The matrix that references the texels around this is called the kernel. The kernel is a matrix that determines the proportion of texel colors mixed.

Gaussian blur is the most commonly used blur treatment. As the name implies, this refers to the process of using a Gaussian distribution in the kernel. Read the Gaussian Blur implementation diagonally to get a feel for how it works in post-effects.

The Gaussian kernel mixes the brightness around the pixel to be processed at a rate that follows a Gaussian distribution. By doing this, it is possible to suppress the blurring of the contour part where the brightness changes non-linearly.

As a review of mathematics, the Gaussian distribution can be expressed by the following formula.

```
G\left( x\right) =\dfrac {1}{\sqrt {2 \pi \sigma ^{2}}}\exp \left( -\dfrac {x^{2}}{2\sigma ^{2}}\right)
```

Since the Gaussian distribution can be approximated to the binomial distribution here, the Gaussian distribution can be substituted by the combination of weighting according to the binomial distribution as shown below (see footnote * 2 for the approximation of the Gaussian and binomial distributions).

GaussianBlur.shader

```
float4 x_blur (v2f i) : SV_Target
{
    float weight [5] = { 0.2270270, 0.1945945, 0.1216216, 0.0540540, 0.0162162 };
  float offset [5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };
  float2 size = _MainTex_TexelSize;
  fixed4 col = tex2D(_MainTex, i.uv) * weight[0];
  for(int j=1; j<5; j++)
```

```
  {
    col += tex2D(_MainTex, i.uv + float2(offset[j], 0) * size) *
weight[j];
    col += tex2D(_MainTex, i.uv - float2(offset[j], 0) * size) *
weight[j];
  }
  return col;
}
```

The above code is only in the x direction, but the processing is almost the same in the y direction. Here, the blur in the x and y directions is divided into two directions, and the number of brightness acquisitions is reduced from n * n = n ^ 2 times to n * 2 + 1 = 2n + 1 times. Because you can.

Figure 10.1: Confirmation that Blur composition in each direction correctly blurs

On the script side, `OnRenderImage`Blit alternately between src and temporary RenderTexture in each direction of xy, and finally Blit from src to dst and output. On MacOS, Blitt was possible only with src, but on Windows, the result was not output, so `RenderTexture.GetTemporary`I am using. (For OnRenderImage and Blit, refer to the introduction to ImageEffect in the previous chapter.)

GaussianBlur.cs

```
void OnRenderImage (RenderTexture src, RenderTexture dst)
{
  var rt = RenderTexture.GetTemporary(src.width, src.height, 0,
src.format);

  for (int i = 0; i < blurNum; i++)
```

```
  {
    Graphics.Blit(src, rt, mat, 0);
    Graphics.Blit(rt, src, mat, 1);
  }
  Graphics.Blit(src, dst);

  RenderTexture.ReleaseTemporary(rt);
}
```

This is the end of the explanation of Gaussian blur. Now that you have a sense of how post-effects are performed, I will explain SSR from the next section.

10.3 SSR

SSR is a technique that attempts to reproduce reflections and reflections within the range of post effects. All that is required for SSR is the image itself taken by the camera, the depth buffer in which the depth information is written, and the normal buffer in which the normal information is written. Depth buffer and normal buffer are collectively called G-buffer and are indispensable for Deferred rendering such as SSR. (For Deferred Rendering, there is a great explanation in the introduction to ImageEffect in the previous chapter, so please refer to that.)

This is a premise when reading this section, but in this section, we will proceed with the explanation on the premise of basic knowledge about ray tracing. Ray tracing is a big theme that I can write another chapter even at the introductory level, so unfortunately I will omit the explanation here. However, if you do not understand what ray tracing is, you can not understand the following contents, so if you do not understand it, there is a good introduction book "Ray Tracing in One Weekend" * 3 by Peter Shirley, so it is recommended that you read that first. I will.

In addition, kode80's "Screen Space Reflections in Unity 5 * 6 " is famous as a commentary text for the Unity implementation of SSR . Also, in Japanese text, there is "I tried to implement Screen Space Reflection in Unity * 8 ". In this section, what is explained in the above text is simplified as much as possible, and explanation of branch and leaf techniques is omitted. If you read the source code and find any questions, try to hit them.

10.3.1 Overview of Theory

The basic idea of SSR is to use ray tracing techniques to simulate the relationship between a camera, a reflective surface, and an object (light source).

Unlike ordinary optics, SSR reproduces reflections on the reflecting surface by fetching the color on the reflecting surface after identifying the light source by calculating back from the path of light incident on the camera.

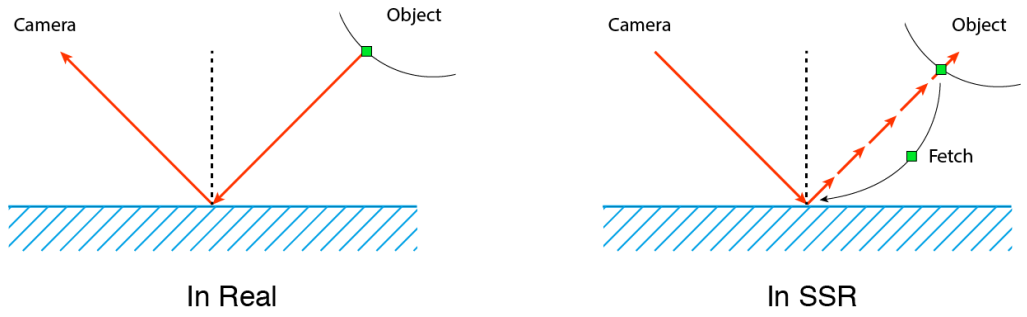

Figure 10.2: Differences between real-life optics and SSR light thinking

SSR does this for each pixel of the camera.

The outline of the process can be summarized as follows.

1. Return the screen coordinate system to the world coordinate system by using depth information
2. Find the reflection vector from the line-of-sight vector and normal information
3. Extend the reflection vector a little and return the position of its tip (= ray) to the screen coordinate system again.
4. Compare the depth of the ray with the depth written in the depth buffer for the position of the ray in the screen coordinate system.
5. If the ray is less deep, the ray is still wandering in the air. Go back to 3 and move Ray a little further
6. If the ray is deeper, it means that the ray has passed some object and you can get the reflected color.

7. Return to the original pixel and reflect the acquired color

The procedure is difficult to explain in the figure, but it is complicated to explain in words. Let's disassemble it.

10.3.2 Coordinate transformation

First, pass the matrix for converting the screen coordinate system and the world coordinate system to the shader. `_ViewProj`Is the transformation matrix from the world coordinate system to the screen coordinate system, and `_InvViewProj`is the inverse matrix.

SSR.cs

```
void OnRenderImage (RenderTexture src, RenderTexture dst)
{

  ....

  // world <-> screen matrix
  var view = cam.worldToCameraMatrix;
    var  proj  =  GL.GetGPUProjectionMatrix(cam.projectionMatrix,
false);
  var viewProj = proj * view;
  mat.SetMatrix("_ViewProj", viewProj);
  mat.SetMatrix("_InvViewProj", viewProj.inverse);

  ....

}
```

Now, using the transformation matrix passed, the normal vector and the reflection vector can be obtained. Let's take a look at the processing of the corresponding shader.

SSR.shader

```
float4 reflection (v2f i) : SV_Target
{
  float2 uv = i.screen.xy / i.screen.w;
  float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);

  ...
```

```
float2 screenpos = 2.0 * uv - 1.0;
float4 pos = mul(_InvViewProj, float4(screenpos, depth, 1.0));
pos /= pos.w;
float3 camDir = normalize(pos - _WorldSpaceCameraPos);
float3 normal = tex2D(_CameraGBufferTexture2, uv) * 2.0 - 1.0;
float3 refDir = reflect(camDir, normal);

....

if (_ViewMode == 1) col = float4((normal.xyz * 0.5 + 0.5), 1);
if (_ViewMode == 2) col = float4((refDir.xyz * 0.5 + 0.5), 1);

....

return col;
}
```

First, the depth of the corresponding pixel is `_CameraDepthTexture`written in, and this is used. Next, from the position information and seismic intensity information on the screen, the position of the polygon in the corresponding pixel in the world coordinate system can be found, so `pos`hold it in. `pos`Then, `_WorldSpaceCameraPos`since the vector toward the camera is known, the reflection vector can be known from this and the normal information.

From the script attached to the main camera, you can see where the normal and reflection vectors are facing. Since each vector is standardized between -1 and 1, color information with a value less than or equal to 0 is not displayed. When the x-axis component is large, the vector is displayed in reddish, when the y-axis component is large, it is displayed in greenish, and when the z-axis component is large, it is displayed in bluish. Please set ViewMode to `Normal`or `Reflection`and check.

10.3.3 Ray tracing

Now let's look at the process of performing ray tracing.

SSR.shader

```
float4 reflection(v2f i) : SV_Target
{
```

```
  ...

  [loop]
  for (int n = 1; n <= _MaxLoop; n++)
  {
    float3 step = refDir * _RayLenCoeff * (lod + 1);
    ray += step * (1 + rand(uv + _Time.x) * (1 - smooth));

    float4 rayScreen = mul (_ViewProj, float4 (ray, 1.0));
    float2 rayUV      = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
    float  rayDepth   = ComputeDepth(rayScreen);
    float  worldDepth = (lod == 0)?
           SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, rayUV) :
           tex2Dlod (_CameraDepthMipmap, float4 (rayUV, 0, lod))
           + _BaseRaise * lod;

    ...

    if(rayDepth < worldDepth)
    {

      ....

      return outcol;
    }
  }
}
```

Variables related to the processing explained later are also mixed, but please read it without worrying about it. Inside the loop, first stretch the ray by a step and then put it back in the screen coordinate system. We will compare the depth of the ray in the screen coordinate system with the depth written to the depth buffer and return the color if the ray is deeper. (1.0 when the depth is the closest, because the smaller the further away, `rayDepth`but `worldDepth`is smaller than the will to a determination that Ray is in the back.)

Also, if the number of loops is undecided, HLSL will throw an error, so if you want to pass the number of loops from the script `[loop]`, you need to write the attribute at the beginning.

The skeleton of ray tracing is now complete. The basic processing is not so difficult once you have an image. However, in order to reproduce beautiful

reflections, it is necessary to add some processing from now on. Are the following four points that need to be improved?

1. Due to the limited number of loops, the distance traveled by the ray is not large enough to reproduce the reflection of distant objects.
2. If the number of steps of the ray is large, it will pass through the reflected object and sample the wrong color.
3. Since the loop is performed many times, the processing is simply heavy.
4. Not taking into account material differences

For post-effects, including antialiasing, techniques for efficient processing are rather essential. Now that you understand the gist of the process, let's look at the technique for establishing SSR as a video.

10.3.4 Mipmap

Below, we will explain how to improve processing efficiency by using Mipmap, referring to the article * 7 of Chalmers University of Technology . (See footnote for what Mipmap is * 9) Ray tracing basically determines the step width of the ray and gradually advances the ray, but by using Mipmap, the step of the ray until the intersection with the object is judged. The width can be variable. By doing this, you will be able to fly rays far away even with a limited number of loops, and processing efficiency will also increase.

We have prepared a demoscene that uses Mipmap from RenderTexture, so let's check it from there.

Mipmap.cs

```
public class Mipmap : MonoBehaviour
{
  Material mat;
  RenderTexture rt;
  [SerializeField] Shader shader;
  [SerializeField] int lod;

  void OnEnable()
  {
    mat = new Material(shader);
```

```
    rt = new RenderTexture(Screen.width, Screen.height, 24);
    rt.useMipMap = true;
  }

  void OnDisable()
  {
    Destroy(mat);
    rt.Release();
  }

  void OnRenderImage (RenderTexture src, RenderTexture dst)
  {
    mat.SetInt("_LOD", lod);
    Graphics.Blit(src, rt);
    Graphics.Blit(rt, dst, mat);
  }
}
```

Since mipmap cannot be set for ready-made RenderTexture, here, `src`after creating a new RenderTexture and copying it, processing is added.

Mipmap.shader

```
sampler2D _MainTex;
float4 _MainTex_ST;
int _LOD;

....

fixed4 frag (v2f i) : SV_Target
{
        return tex2Dlod(_MainTex, float4(i.uv, 0, _LOD));
}
```

`tex2Dlod(_MainTex, float4(i.uv, 0, _LOD))`You can get the Mipmap according to the LOD with.

If you raise the LOD from the script attached to the camera on the scene, you can see that the image becomes grainy.

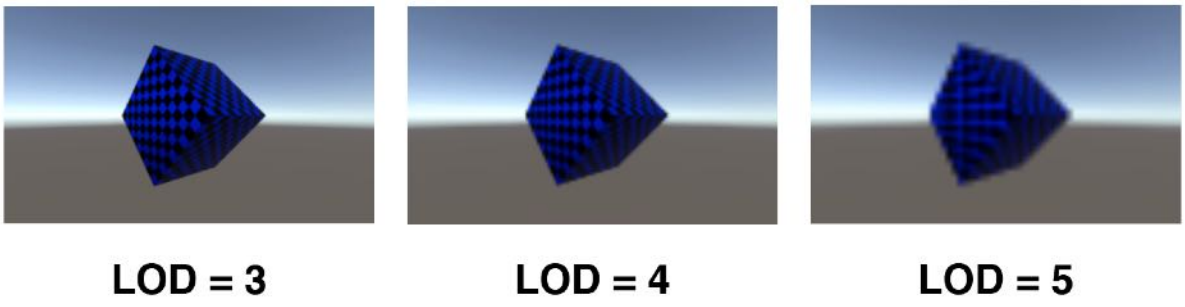

Figure 10.3: Comparison of increased LOD and Mipmap image quality

Now that you have confirmed how to use Mipmap, let's see how Mipmap is used in the SSR scene.

SSR.shader

```
[loop]
for (int n = 1; n <= _MaxLoop; n++)
{
  float3 step = refDir * _RayLenCoeff * (lod + 1);
  ray += step;

  ....

  if(rayDepth < worldDepth)
  {
    if(lod == 0)
    {
      if (rayDepth + _Thickness > worldDepth)
      {
        float sign = -1.0;
        for (int m = 1; m <= 8; ++m)
        {
          ray += sign * pow(0.5, m) * step;
          rayScreen = mul (_ViewProj, float4 (ray, 1.0));
          rayUV = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
          rayDepth = ComputeDepth(rayScreen);
          worldDepth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture,
rayUV);
          sign = (rayDepth < worldDepth) ? -1 : 1;
        }
        refcol = tex2D(_MainTex, rayUV);
      }
      break;
```

```
    }
    else
    {
      ray -= step;
      lod--;
    }
  }
  else if(n <= _MaxLOD)
  {
    lod ++;
  }
  calcTimes = n;
}
if (_ViewMode == 3) return float4(1, 1, 1, 1) * calc / _MaxLoop;

....
```

I will proceed with the explanation using the figure in the article of Chalmers.

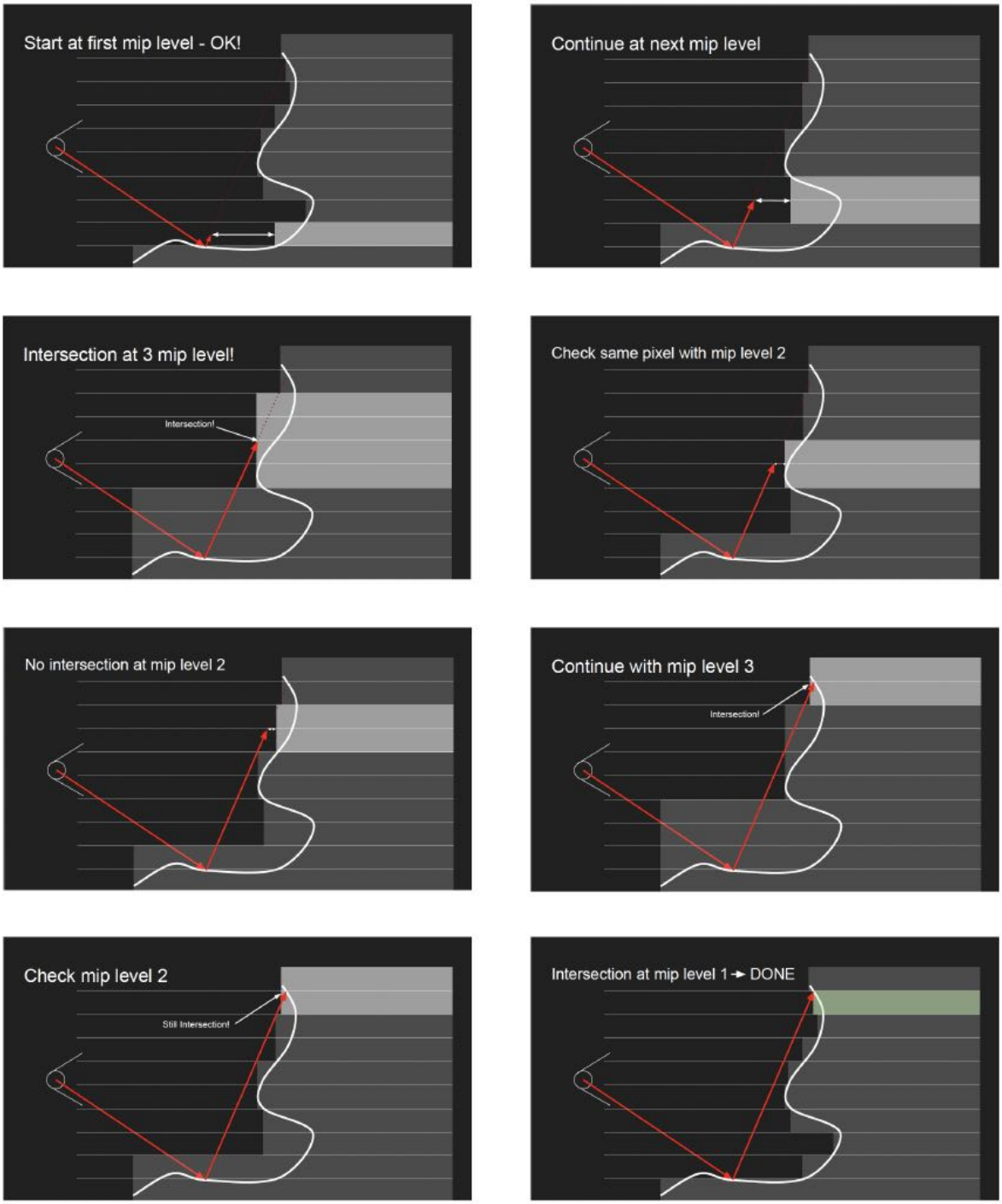

Figure 10.4: Calculation method using Mipmap

As shown in the figure, the LOD is raised while carefully judging the intersection for the first few times. As long as there is no intersection with other meshes, we will proceed with large steps. If there is an intersection,

Unity's MipMap will roughen the pixels while taking the average value, so unlike the case of the article, the ray may go too far. Therefore, move back by one unit step and advance the ray again with one smaller LOD. Finally, by making an intersection judgment on the image with LOD = 0, the moving distance of the ray can be extended and the processing can be made more efficient.

From the script attached to the main camera, you can see how much the amount of calculation changes when you raise the LOD. The larger the amount of calculation, the whiter it looks, and the smaller the amount of calculation, the darker it looks. Set ViewMode and `CalcCount`change the LOD to check the change in the amount of calculation.

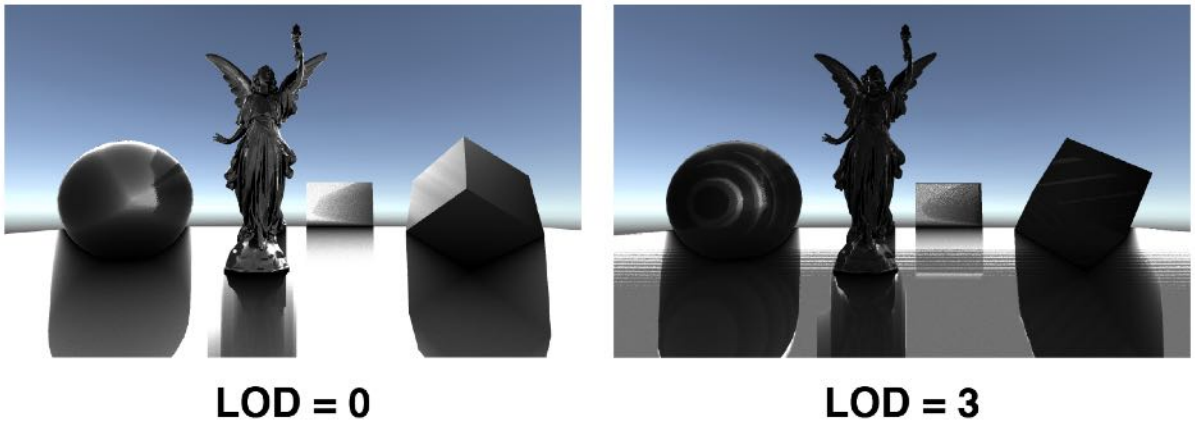

Figure 10.5: Difference in computational complexity due to changes in LOD (closer to black, smaller computational complexity)

10.3.5 Dichotomous exploration

Let's see how to improve the accuracy near the intersection by binary tree search. Check from the code immediately.

SSR.shader

```
if (lod == 0)
{
  if (rayDepth + _Thickness > worldDepth)
  {
    float sign = -1.0;
```

```
    for (int m = 1; m <= 8; ++m)
    {
      ray += sign * pow(0.5, m) * step;
      rayScreen = mul (_ViewProj, float4 (ray, 1.0));
      rayUV = rayScreen.xy / rayScreen.w * 0.5 + 0.5;
      rayDepth = ComputeDepth(rayScreen);
         worldDepth  =  SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture,
rayUV);
      sign = (rayDepth < worldDepth) ? -1 : 1;
    }
    refcol = tex2D(_MainTex, rayUV);
  }
  break;
}
```

Immediately after the intersection, it is behind the intersected object, so first retract the ray. After that, while checking the context of the ray and the mesh, change the direction of travel of the ray either forward or backward. At the same time, by shortening the step width of the ray, it is possible to identify the intersection with the mesh with less error.

10.3.6 Reflects material differences

The methods so far have not taken into account the differences in the materials of the objects in the screen. Therefore, there is a problem that all objects are reflected to the same extent. Therefore, use G-buffer again. `_CameraGBufferTexture1.w`Since the smoothness of the material is stored in, use this.

SSR.shader

```
if (_ViewMode == 8)
   return  float4(1,  1,  1,  1)  *  tex2D(_CameraGBufferTexture1,
uv).w;

....

return
  (col * (1 - smooth) + refcol * smooth) * _ReflectionRate
  + col * (1 - _ReflectionRate);
```

If you change the smoothness value of the material attached to an object in the scene, you can see that only that object changes the degree of reflection.

SmoothnessYou can also list the smoothness in the scene by setting the ViewMode of the script attached to the main camera . The whitish, the greater the smoothness.

10.3.7 Blur processing

This is the part using the Gaussian blur explained in the first section. If the step width of the ray is not small enough, you may not be able to get the reflection well even if you perform a binary tree search. If the step width of the ray is reduced, the total length of the ray will be shortened and the amount of calculation will increase, so it is not enough to just reduce the step width, but it should be kept to an appropriate size. The part where the reflection could not be obtained well is blurred to make it look like it.

SSR.shader

```
float4 xblur(v2f i) : SV_Target
{
  float2 uv = i.screen.xy / i.screen.w;
  float2 size = _ReflectionTexture_TexelSize;
  float smooth = tex2D(_CameraGBufferTexture1, uv).w;

  // compare depth
  float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv);
  float depthR =
        SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv + float2(1,
0) * size);
  float depthL =
        SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv - float2(1,
0) * size);

  if (depth <= 0) return tex2D(_ReflectionTexture, uv);

     float  weight[5]  =  {  0.2270270,  0.1945945,  0.1216216,
0.0540540, 0.0162162 };
  float offset[5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };

  float4 originalColor = tex2D(_ReflectionTexture, uv);
     float4  blurredColor  =  tex2D(_ReflectionTexture,  uv)  *
weight[0];

  for (int j = 1; j < 5; ++j)
  {
    blurredColor
```

```
        += tex2D(_ReflectionTexture, uv + float2(offset[j], 0) *
size)
           * weight[j];

    blurredColor
        += tex2D(_ReflectionTexture, uv - float2(offset[j], 0) *
size)
           * weight[j];
  }

    float4 o = (abs(depthR - depthL) > _BlurThreshold) ?
originalColor
                : blurredColor * smooth + originalColor * (1 -
smooth);
  return o;
}
```

Again, from the reason described above `xblur`and `yblur`have divided the processing out. Also, since we only want to blur the contours within the same reflective surface, we try not to blur the contours. If the difference between the left and right depths is large, it is judged to be the contour part. (Then `yblur`evaluates the difference between the top and bottom.)

The result of adding the processing up to this point is as follows.

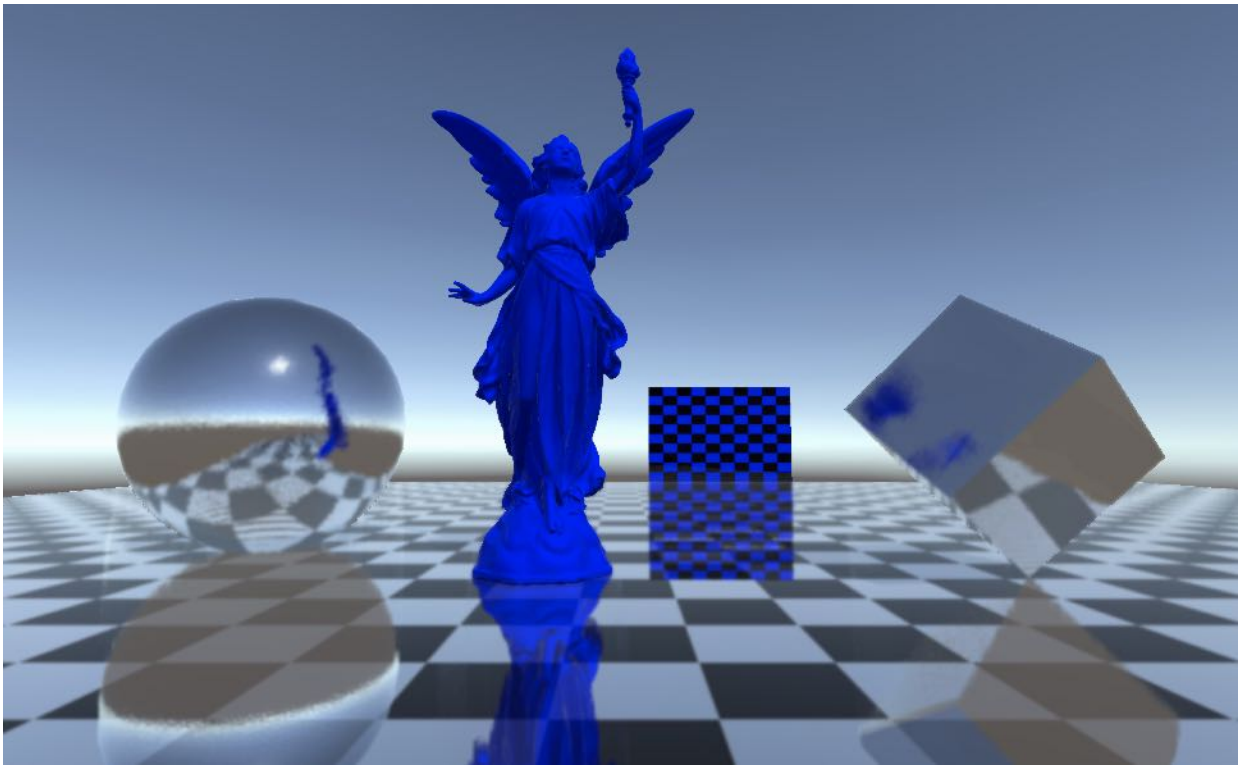

Figure 10.6: Results

10.3.8 Bonus

As a bonus, I will introduce a technique that makes it look as if a non-existent object is reflected using two cameras, a main camera and a sub camera.

SSRMainCamera.shader

```
float4 reflection(v2f i) : SV_Target
{

  ....

  for (int n = 1; n <= 100; ++n)
  {
    float3 ray = n * step;
    float3 rayPos = pos + ray;
    float4 vpPos = mul (_ViewProj, float4 (rayPos, 1.0));
    float2 rayUv = vpPos.xy / vpPos.w * 0.5 + 0.5;
    float rayDepth = vpPos.z / vpPos.w;
                              float       subCameraDepth       =
```

```
SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv);

      if (rayDepth < subCameraDepth && rayDepth + thickness >
subCameraDepth)
    {
      float sign = -1.0;
      for (int m = 1; m <= 4; ++m)
      {
        rayPos += sign * pow(0.5, m) * step;
        vpPos = mul (_ViewProj, float4 (rayPos, 1.0));
        rayUv = vpPos.xy / vpPos.w * 0.5 + 0.5;
        rayDepth = vpPos.z / vpPos.w;
                                              subCameraDepth      =
SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv);
        sign = rayDepth - subCameraDepth < 0 ? -1 : 1;
      }
      col = tex2D (_SubCameraMainTex, rayUv);
    }
  }
  return col * smooth + tex2D(_MainTex, uv) * (1 - smooth);
}
```

It is made simple with as little extra processing as possible. The point is that `SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, uv)` it `SAMPLE_DEPTH_TEXTURE(_SubCameraDepthTex, rayUv)` is used instead of for depth evaluation, and the object information to be referenced `_SubCameraMainTex` is also obtained from. `_CameraDepthTexture`, `_SubCameraDepthTex` Is set as a global texture from the sub camera.

The downside is that each camera casts shadows on objects that shouldn't be visible. It may not be very practical, but it's a little interesting effect.

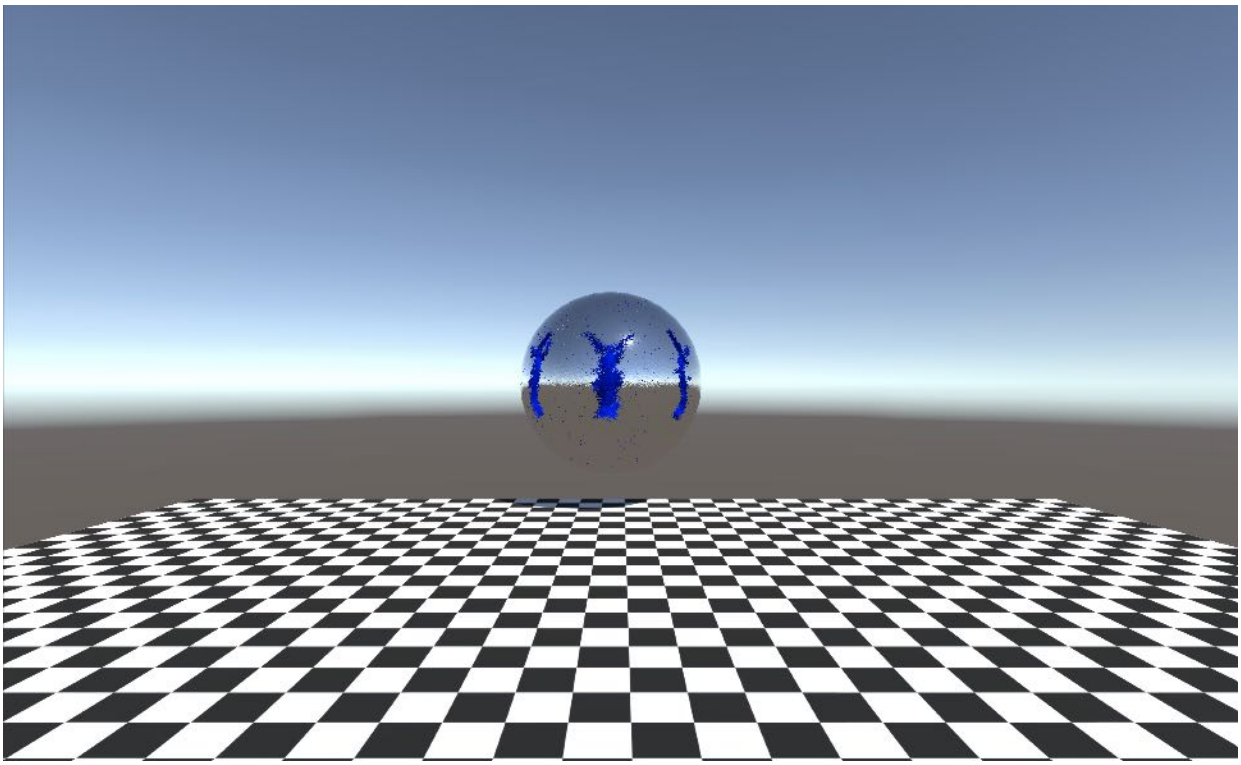

Figure 10.7: Method using two cameras

10.4 Summary

This is the end of the explanation of SSR.

Since SSR is a technique that requires a large amount of processing capacity, it is not realistic to reflect objects in all positions cleanly. Therefore, the point is to improve the appearance of the reflection of the object of interest and to make the trivial reflection look like it with less processing. In addition, the screen size to be rendered is directly linked to the amount of calculation, so it is important to search for the points that will be established as an image while considering the expected screen size and GPU performance. Check the role and trade-offs of each parameter by adjusting the parameters while moving the objects in the scene.

In addition, the Mipmap, binary tree search, how to use the camera buffer, and many other detailed techniques mentioned above can be applied not only to SSR but also to various places. I would be happy if there is some content that is helpful to the readers.

[*1] http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

[*2] https://ja.wikipedia.org/wiki/%E4%BA%8C%E9%A0%85%E5%88%86%E5%B8%83

[*3] https://www.amazon.co.jp/gp/product/B01B5AODD8

[*4] https://msdn.microsoft.com/ja-jp/library/bb219690(v=vs.85).aspx

[*5] https://www.sciencelearn.org.nz/resources/48-reflection-of-light

[*6] http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/

[*7] http://www.cse.chalmers.se/edu/year/2017/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf

[*8] http://tips.hecomi.com/entry/2016/04/04/022550

[*9] https://answers.unity.com/questions/441984/what-is-mip-maps-pictures.html

[*10] https://docs.unity3d.com/Manual/RenderTech-DeferredShading.html

About the author

Chapter 1 Real-Time GPU-Based Voxelizer-Madam Nakamura/@mattatz

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

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

Chapter 2 GPU-Based Trail-Fu Naga Hidekazu / @fuqunaga

Former game developer, programmer making interactive art. I like the design and development of moderately complicated mechanisms and libraries. Night Type.

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

Chapter 3 Application of Geometry Shader for Line Representation-@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/
- https://kaiware007.github.io/

Chapter 4 Projection Spray-Hironori Sugi / @sugi_cho

A person who makes interactive art in Unity. Freelance. We look forward to your work => hi@ sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc

Chapter 5 Introduction to Procedural Noise-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 6 Curl Noise-Explanation of Noise Algorithm for Pseudo-Fluids-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 7 Shape Matching --Application of Linear Algebra to CG-- Kodai Takao / @ kodai100

Former VFX production technical artist. Current interactive artist / engineer. I'm still a student.

- https://twitter.com/kodai100_tw
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

Chapter 8 Space Filling-@a3geek

Interaction engineer. I am interested in visualization of simulations by CG, and I would like to make visualizations that shake people's emotions more, rather than visualizing them accurately. I like to make it, but I find it more fun to know more than that. My favorite school classroom is the drawing room or the library.

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

Chapter 9 Introduction to ImageEffect-@XJINE

Continuing from the previous introduction to ComputeShader, this time it was more loose and fluffy than Graphics Programming :-) I hope you can reach out to those who haven't been able to keep up with the advanced content written by others.

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

Chapter 10 ImageEffect Application (SSR)-@komietty

After working in physics and the web, he is an interactive engineer. New art school 3rd term. Interested in stage production.

- https://github.com/komietty
- https://www.instagram.com/komietty/