Unity Graphics Programming

Unityグラフィックスプログラミング vol.3

IndieVisualLab

fuqunaga
irishoak
kaiware007
kodai100
mattatz
sakope
sugi-cho
XJINE

https://indievisuallab.github.io/

Unity Graphics Programming vol.3

IndieVisualLab

IndieVisualLab

Preface

This book is the third 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/UnityGraphicsProgramming3), so you can read this manual while executing it at hand.

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

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

Recommended execution environment

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

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

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

Requests and impressions about books

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

第1章　Baking Skinned Animation to Texture

1.1　Introduction

Hello, I'm Sugino! This chapter displays thousands and tens of thousands of skinned animated objects.

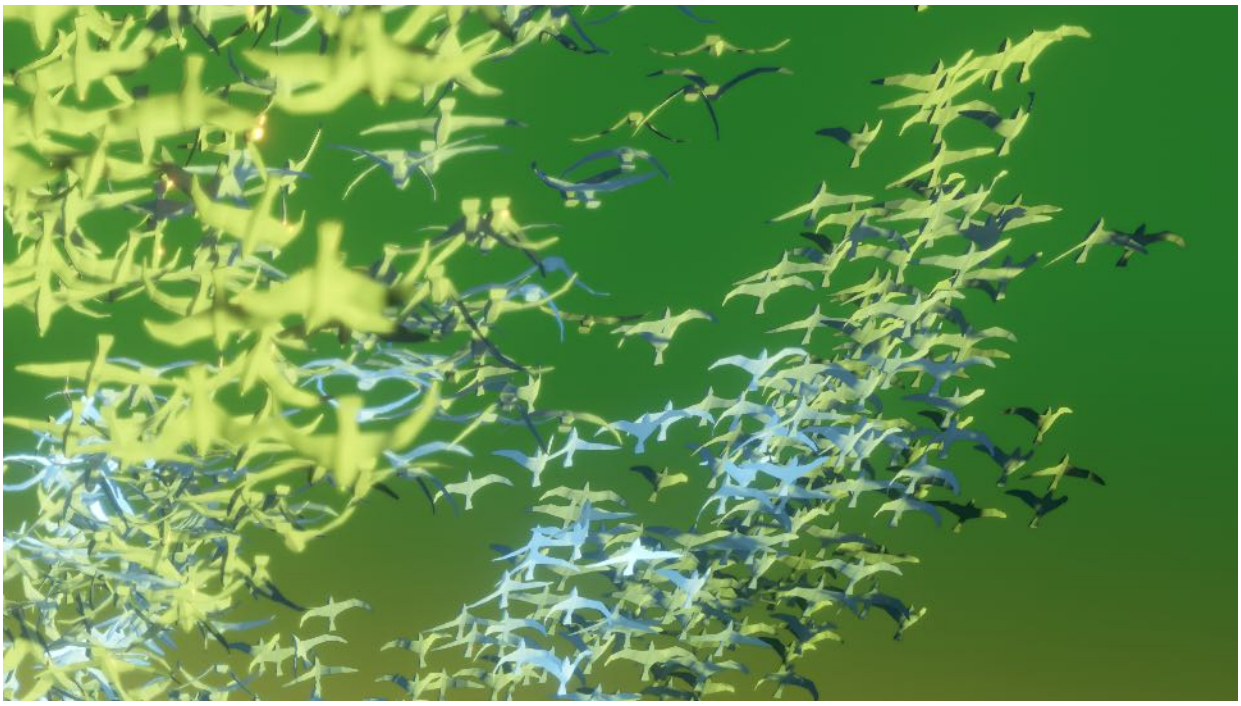

Figure 1.1: A flock of birds flapping their wings

In Unity, I think you'll be using the Animator and SkinnedMeshRenderer components to achieve character animation.

For example, what if you want to represent a flock or crowd of birds? Would you like to use Animator and SkinnedMeshRenderer for thousands or tens of thousands of character objects? Generally, when displaying a large number of objects on the screen, GPU instancing is used to render a large number of objects at once. However, SkinnedMeshRenderer does not support

instancing, which renders individual objects one by one, which is very heavy.

As a solution to solve this, there is a method to save the animated vertex position information as a texture, but in this chapter we will explain how to actually do it, the way of thinking and application until implementation, and points to be noted. I will.

Please feel free to ask questions on Twitter (to @sugi_cho) as some explanations may be omitted or some parts may be difficult to understand. If there is something wrong, I would appreciate it if you could point it out (._.)

1.2 Place 5000 Skined Mesh Renderers to animate

First of all, I would like to see how heavy the processing would be if a large number (5000 objects) of normally animated objects were placed. This time, we have prepared a simple animated horse 3D object with 1890 vertices.

Figure 1.2: Horse model used

When I actually moved it, I can see that the FPS is 8.8, which is considerably heavier. Figure 1.3

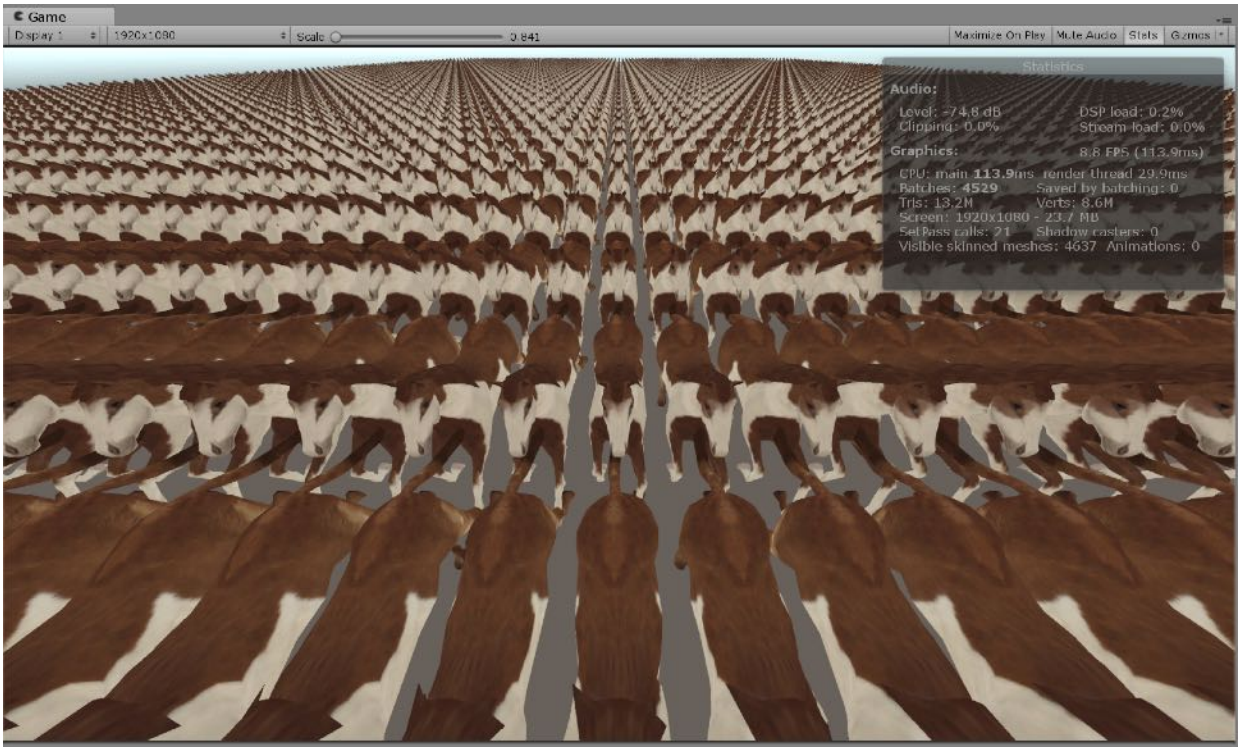

Figure 1.3: 5000 animated horses

Now, let's look at Unity's profiler to find out what is getting heavier in this process. Display Profiler (shortcut key: Ctr + 7) from the Window menu. You can get more detailed information by selecting GPU from the Add Profiler pull-down and viewing the GPU Usage profiler. Obtaining GPU Usage information itself is an overhead, so it is better not to display it when it is not needed, but this time GPU Usage will be important, so we will actively use it.

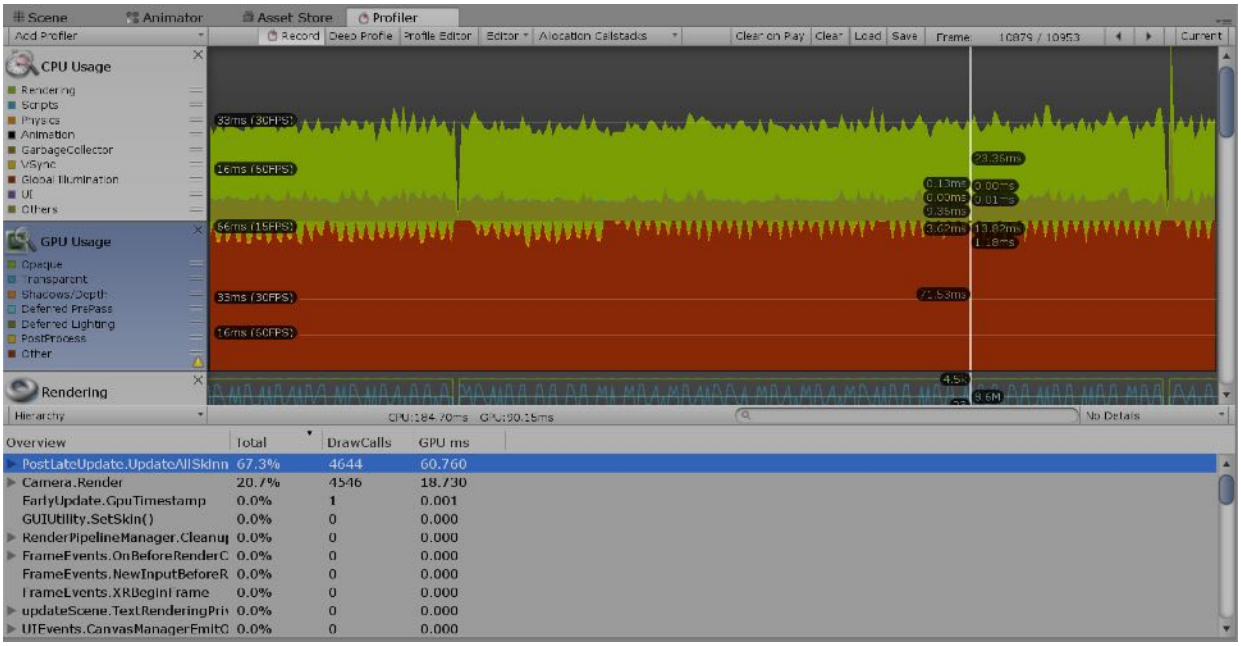

図 1.4: Profiler Window (GPU Usage)

Looking at the profiler, you can see that the GPU processing time is longer than the CPU processing time, and the CPU is waiting for the GPU processing to complete. Figure 1.4 And `PostLateUpdate.UpdateAllSkinnedMeshes`you can see that about 70% of GPU processing is occupied. Also, since there are as many horse objects as you can see, it `Camera.Renderer`seems that you can reduce the number of GPU rendering processes by batching the objects or performing GPU instancing. It is the way, but in the same way as any CPU Usage `PostLateUpdate.UpdateAllSkinnedMeshes`and `Camera.Render`processing of account for most of the time.

In this test scene, the Player Settings are set to use GPU Skinning. If you were skinning on the CPU instead of the GPU, the CPU processing rate would increase and the FPS would be lower than it is now. At the time of GPU skinning, the CPU side calculates the bone matrix, passes the matrix information to the GPU, and performs the skinning process on the GPU. At the time of CPU skinning, matrix calculation and skinning processing are performed on the CPU side, and the skinned vertex data is passed to the GPU side.

In this way, in order to optimize processing, it is important to first determine where the processing bottleneck is.

1.3 Pre-calculate the positions of the animated vertices of SkinnedMeshRenderer

As a result of profiling, the mesh skinning process seems to be heavy. Now that I know that, I would like to consider a method of calculating in advance instead of performing the skinning process itself in real time.

`SkinnedMeshRendererSkinnedMeshRenderer.BakeMesh(Mesh)`There is a function called as a method to acquire the vertex information after the skinning process of . It takes a snapshot of the skinned mesh and stores it in the specified mesh. It takes a little time to process, but it can be selected if it is used to store skinned vertex information in advance.

Listing 1.1: SkinnedMeshRenderer.BakeMesh () Example

```
 1: animator animator;
 2: SkinnedMeshRenderer skinedMesh;
 3: List<Mesh> meshList;
 4:
 5: void Start(){
 6:     animator = GetComponent<Animator>();
       7:                                          skinnedMesh       =
GetComponentInChildren<SkinnedMeshRenderer>();
 8:     meshList = new List<Mesh>();
 9:     animator.Play("Run");
10: }
11:
12: void Update(){
13: var mesh = new Mesh ();
14: skinnedMesh.BakeMesh (mesh);
15: // A snapshot of the skinned mesh is stored in mesh
16:     meshList.Add(mesh);
17: }
```

`SkinnedMeshRenderer`Now, the mesh of the snapshot of each frame of the animation of the Animator's Run state will be stored in the meshList. Listing 1.1

If `meshList`you use this saved file `MeshFilter.sharedMesh`and switch the mesh ( ) in the same way as switching pictures in a flip book, `SkinnedMeshRenderer`you can display the animation of the mesh without using, so the skinning process that was a bottleneck as a result of profiling It seems that you can omit.

1.4 Examining how to save vertex information

However, if this implementation saves multiple Mesh data for each frame, mesh information (Mesh.indeces, Mesh.uv, etc.) that is not changed by animation will also be saved, resulting in a lot of waste. In the case of skinning animation, the only data to be updated is the vertex position information and normal information, so you only need to save and update these.

1.4.1 How to save vertex information as a Vector3 array

One possible method is to have the vertex position and normal data for each frame in an array of Vector3, and update the mesh position and normal for each frame. Listing 1.2

Listing 1.2: Update Mesh

```
 1: Mesh objMesh;
 2: List<Vector3>[] vertecesLists;
 3: List<Vector3>[] normalsLists;
 4: // Saved vertex information
 5: // For use with Mesh.SetVertices (List <Vector3>)
 6:
 7: void Start(){
 8:     objMesh = GetComponent<MeshFilter>().mesh;
 9:     objMesh.MarkDynamic();
10: }
11:
12: void Update(){
13:     var frame = xx;
14: // Calculate the frame at the current time
15:
16: objMesh.SetVertices (vertecesLists [frame]);
17:     objMesh.SetNormals(normalsLists[frame]);
18: }
```

However, this method puts a heavy CPU load on the mesh update itself for the purpose of displaying the thousands of animation objects that we are trying to solve.

So, as the answer is written from the beginning of this chapter, we store the position information and normal information in the texture, and use VertexTextureFetch to update the vertex position and normal information of the mesh in the vertex shader. This eliminates the need to update the original mesh data itself, making it possible to realize vertex animation without the processing load of the CPU.

1.4.2 Write location information to texture

Now, let's briefly explain how to save the position information of mesh vertices in a texture.

Unity `Mesh`objects are classes that store data such as vertex positions, normals, and UV values of 3D models displayed in Unity. In the vertex position information (`Mesh.vertices`), the position information for all the vertices of the mesh is `Vector3`saved as an array. Table 1.1

And Unity `Texture2D`objects are saved as an array of color information ( ) for the number of pixels of texture width (`texture.width`) x height (`texture.height`) `Color`. Table 1.2

Table 1.1: Location Information (Vector3)

x float x direction component
y float y direction component
z float z direction component

Table 1.2: Pornography (Color)

r float red component
g float green component
b float cyan component
a float opacity component

Positions of the vertices, Mesh.Vertices Table 1.1 x the, y, and z values respectively, the color information of Texture2D Table 1.2 contains r of, g, a b, when stored as TextureAsset in EditorScript, texture vertex position information It will be saved as. This is a sample script that saves the positions and normals of mesh vertices as texture colors. Listing 1.3

Listing 1.3: Saving vertex information to texture

```
 1: public void CreateTex(Mesh sourceMesh)
 2: {
 3:     var vertCount = sourceMesh.vertexCount;
 4:     var width = Mathf.FloorToInt(Mathf.Sqrt(vertCount));
 5:     var height = Mathf.CeilToInt((float)vertCount / width);
 6: // Find the width and height where the number of vertices
<width x height
 7:
  8:              posTex   =   new   Texture2D(width,   height,
TextureFormat.RGBAFloat, false);
  9:             normTex   =   new   Texture2D(width,   height,
TextureFormat.RGBAFloat, false);
10: // Texture2D to store Color []
11: // By specifying TextureFormat.RGBAFloat, you can have color
information with each element Float value.
12:
13:     var vertices = sourceMesh.vertices;
14: var normals = sourceMesh.normals;
15:     var posColors = new Color[width * height];
16:     var normColors = new Color[width * height];
17: // Color information array for the number of vertices
18:
19:     for (var i = 0; i < vertCount; i++)
20:     {
21:         posColors[i] = new Color(
22:             vertices[i].x,
23: vertices [i] .y,
24:             vertices[i].z
25:         );
26:         normColors[i] = new Color(
27:             normals[i].x,
28:             normals[i].y,
29:             normals[i].z
30:         );
31:     }
32: // At each vertex, Color.rgb = Vector3.xyz,
33: // Generate a color array (Color []) such that position →
color, normal → color.
```

```
34:
35:     posTex.SetPixels(posColors);
36:     normTex.SetPixels(normColors);
37:     posTex.Apply();
38:     normTex.Apply();
39: // Set the color array to the texture and apply
40: }
```

Now, `Mesh`the position of the vertex of, we were able to normal information position texture, stamped in the normal texture.

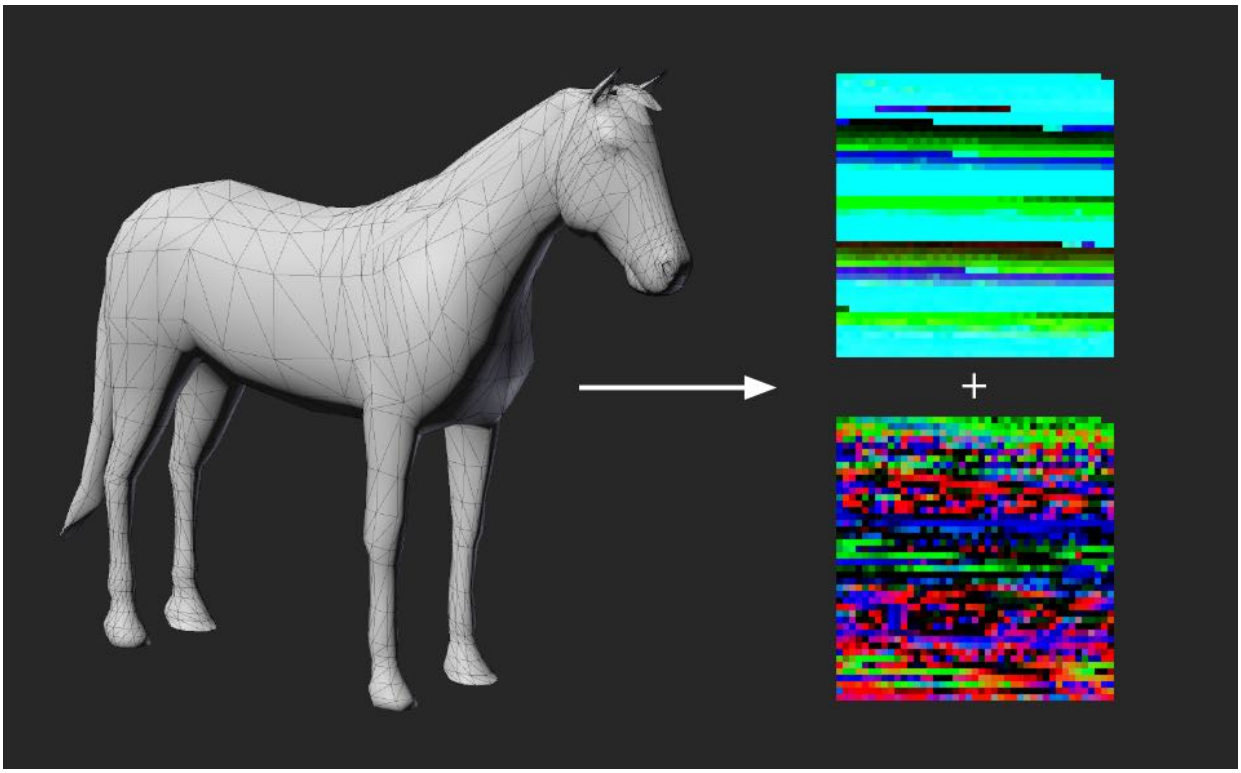

Figure 1.5: Write Mesh vertex positions and normals to Texture

Actually, since there is no index data for making polygons, it is not possible to reproduce the shape of the mesh with only the position texture and normal texture, but it is possible to write the mesh information to the texture. It's done. Figure 1.5

In the Unity of the official manual, `Texture2D.SetPixels(Color[])`it is `ColorFormat.RGBA32,ARGB32,RGB24,Alpha8`to work if only. is what it reads. This is only for fixed-point and fixed-precision texture formats, but apparently `RGBAHalf, RGBAFloat`it works with floating-point values, even if

you assign a negative value or a value of 1 or more to each element of the color. , `Clamp`it seems to us to hold the value without being. `Color`Substituting a fixed precision texture limits the RGB value to a value between 0 and 1 and the precision to 1/256.

In the method of burning the vertex information of this animation into a texture, the animation is sampled at regular intervals, the vertex information of the mesh of each frame is arranged, and a series of animation information is burned into one texture. A total of two textures, a position information texture and a normal information texture, are generated.

1.4.3 AnimationClip.SampleAnimation()

This time, `AnimationClip.SampleAnimation(gameObject, time);`we will use the function to sample the animation . For the specified GameObject, set it to the state of the specified time of AnimationClip. So `Animation`, `Animator`it supports both legacy and legacy . (Rather, it's a way to play an animation without using Animation or Animator components.)

Now, I will explain the actual implementation of specifying a frame from AnimationClip and acquiring the vertex position.

1.5 Implementation

This program consists of the following three elements.

- AnimationClipTextureBaker.cs
- MeshInfoTextureGen.compute
- TextureAnimPlayer.shader

With AnimationClipTextureBaker, get AnimationClip from Animation or Animator, and create ComputeBuffer of mesh vertex data while sampling AnimationClip to each frame. And it is ComputeShader that converts ComputeBuffer of vertex animation information created from AnimationClip and Mesh data into position information texture and normal information texture with MeshInfoTextureGen.compute.

TextureAnimPlayer.shader is a Shader for animating the mesh with the created location and normal textures.

図 1.6: AnimationClipTextureBaker Inspector

`AnimationClipTextureBaker`Inspector. Sets to `ComputeShader`play the animated texture for generating the animated texture `Shader`. Then, set what you want `AnimationClip`to texture to `Clips`. Figure 1.6

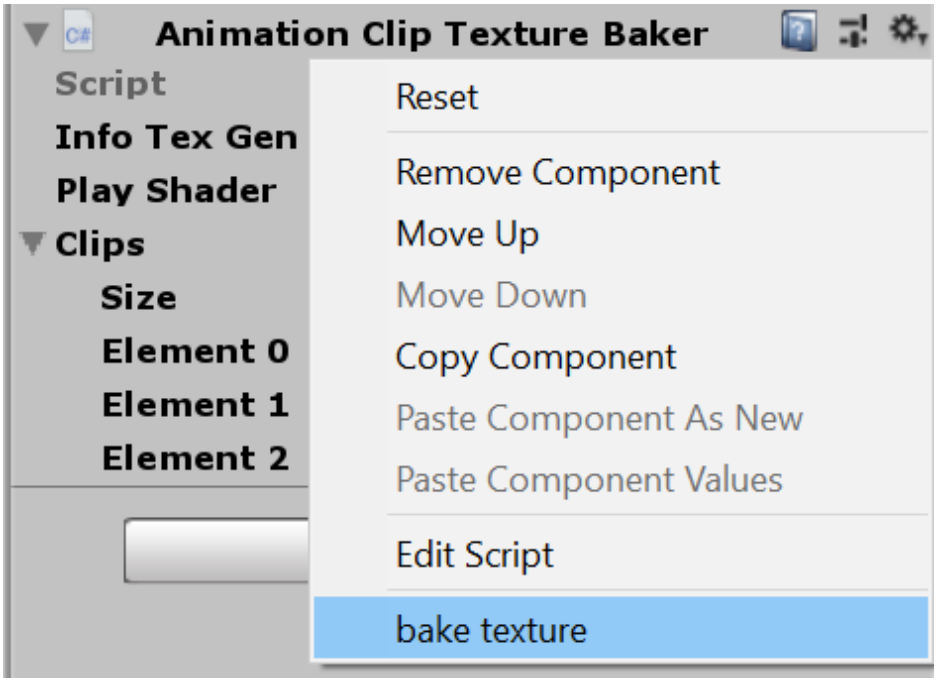

Figure 1.7: Texture writing can be done from the context menu in the Inspector

`ContextMenuAttribute`Allows you to call methods in your script from the context menu in Unity's Inspector. It is convenient because it can be executed without creating an editor extension. In this case, `Bake`you can call the script from "bake texture" in the context menu . Figure 1.6

Now let's look at the actual code.

Listing 1.4: AnimationClipTextureBaker.cs

```
1: using System.Collections.Generic;
2: using System.Linq;
3: using UnityEngine;
4:
5: #if UNITY_EDITOR
```

```
 6: using UnityEditor;
 7: using System.IO;
 8: #endif
 9:
10: public class AnimationClipTextureBaker : MonoBehaviour
11: {
12:
13:     public ComputeShader infoTexGen;
14:     public Shader playShader;
15:     public AnimationClip[] clips;
16:
17: // Vertex information is a structure of position and normal
18: public struct VertInfo
19:     {
20:         public Vector3 position;
21:         public Vector3 normal;
22:     }
23:
24: // Reset () is called when scripting a GameObject in the
editor
25:     private void Reset()
26:     {
27: var animation = GetComponent <Animation> ();
28: var animator = GetComponent <Animator> ();
29:
30:         if (animation != null)
31:         {
32:                                                 clips    =    new
AnimationClip[animation.GetClipCount()];
33: var i = 0;
34:             foreach (AnimationState state in animation)
35:                 clips[i++] = state.clip;
36:         }
37:         else if (animator != null)
38:                                                         clips    =
animator.runtimeAnimatorController.animationClips;
39: // Automatically set AnimationClip if there is an Animation
or Animator component
40:     }
41:
42:     [ContextMenu("bake texture")]
43:     void Bake()
44:     {
45: var skin = GetComponentInChildren <SkinnedMeshRenderer> ();
46:         var vCount = skin.sharedMesh.vertexCount;
47:         var texWidth = Mathf.NextPowerOfTwo(vCount);
48: var mesh = new Mesh ();
49:
```

```
50:         foreach (var clip in clips)
51:         {
52:                     var  frames  =  Mathf.NextPowerOfTwo((int)
(clip.length / 0.05f));
53:             var dt = clip.length / frames;
54:             var infoList = new List<VertInfo>();
55:
56:             var pRt = new RenderTexture(texWidth, frames,
57:                 0, RenderTextureFormat.ARGBHalf);
58:             pRt.name = string.Format("{0}.{1}.posTex", name,
clip.name);
59:             var nRt = new RenderTexture(texWidth, frames,
60:                 0, RenderTextureFormat.ARGBHalf);
61:                 nRt.name  =  string.Format("{0}.{1}.normTex",
name, clip.name);
62:             foreach (var rt in new[] { pRt, nRt })
63:             {
64:                 rt.enableRandomWrite = true;
65:                 rt.Create();
66:                 RenderTexture.active = rt;
67:                 GL.Clear(true, true, Color.clear);
68:             }
69: // Texture initialization
70:
71:             for (var i = 0; i < frames; i++)
72:             {
73:                 clip.SampleAnimation(gameObject, dt * i);
74:   //   Sampling   GameObject   at   the   specified   time   of
AnimationClip
75:                 skin.BakeMesh(mesh);
76: // Call BakeMesh () to get the mesh data in the skinned
state
77:
78:                     infoList.AddRange(Enumerable.Range(0,
vCount)
79:                 .Select(idx => new VertInfo()
80:                 {
81:                     position = mesh.vertices[idx],
82:                     normal = mesh.normals[idx]
83:                 })
84:                 );
85: // Store the animation frame in the list first
86:             }
87:             var buffer = new ComputeBuffer(
88:                 infoList.Count,
89:
System.Runtime.InteropServices.Marshal.SizeOf(
90: typeof (VertInfo)
```

```
91:                 )
92:             );
93:             buffer.SetData(infoList.ToArray());
94: // Set vertex information in ComputeBuffer
95:
96:             var kernel = infoTexGen.FindKernel("CSMain");
97: uint x, y, z;
98:             infoTexGen.GetKernelThreadGroupSizes(
99:                 kernel,
100:                 out x,
101: out y,
102:                 out z
103:             );
104:
105:             infoTexGen.SetInt("VertCount", vCount);
106:             infoTexGen.SetBuffer(kernel, "Info", buffer);
107:                infoTexGen.SetTexture(kernel, "OutPosition",
pRt);
108:                 infoTexGen.SetTexture(kernel, "OutNormal",
nRt);
109:             infoTexGen.Dispatch(
110:                 kernel,
111:                 vCount / (int)x + 1,
112:                 frames / (int)y + 1,
113:                 1
114:             );
115: // Set up Compute Shader and generate textures
116:
117:             buffer.Release();
118:
119: // Editor script to save the generated texture
120: #if UNITY_EDITOR
121:             var folderName = "BakedAnimationTex";
122:                    var  folderPath  =  Path.Combine("Assets",
folderName);
123:             if (!AssetDatabase.IsValidFolder(folderPath))
124:                        AssetDatabase.CreateFolder("Assets",
folderName);
125:
126:             var subFolder = name;
127:                var subFolderPath = Path.Combine(folderPath,
subFolder);
128:                                                          if
(!AssetDatabase.IsValidFolder(subFolderPath))
129:                       AssetDatabase.CreateFolder(folderPath,
subFolder);
130:
131:                                          var   posTex   =
```

```
RenderTextureToTexture2D.Convert(pRt);
132:                                                var   normTex   =
RenderTextureToTexture2D.Convert(nRt);
133:             Graphics.CopyTexture(pRt, posTex);
134:             Graphics.CopyTexture(nRt, normTex);
135:
136: var mat = new Material (playShader);
137:                                       mat.SetTexture("_MainTex",
skin.sharedMaterial.mainTexture);
138:             mat.SetTexture("_PosTex", posTex);
139:             mat.SetTexture("_NmlTex", normTex);
140:             mat.SetFloat("_Length", clip.length);
141:             if (clip.wrapMode == WrapMode.Loop)
142:             {
143:                 mat.SetFloat("_Loop", 1f);
144:                 mat.EnableKeyword("ANIM_LOOP");
145:             }
146:
147:                    var  go  =  new  GameObject(name  +  "."  +
clip.name);
148:              go.AddComponent<MeshRenderer>().sharedMaterial
= mat;
149:                  go.AddComponent<MeshFilter>().sharedMesh =
skin.sharedMesh;
150: // Set the generated texture as a material, set the mesh
and make a Prefab
151:
152:             AssetDatabase.CreateAsset(posTex,
153:                     Path.Combine(subFolderPath, pRt.name +
".asset"));
154:             AssetDatabase.CreateAsset(normTex,
155:                     Path.Combine(subFolderPath, nRt.name +
".asset"));
156:             AssetDatabase.CreateAsset(mat,
157:                 Path.Combine(subFolderPath,
158:                      string.Format("{0}.{1}.animTex.asset",
name, clip.name)));
159:             PrefabUtility.CreatePrefab(
160:                         Path.Combine(folderPath, go.name +
".prefab")
161:                 .Replace("\\", "/"), go);
162:             AssetDatabase.SaveAssets();
163:             AssetDatabase.Refresh();
164: #endif
165:         }
166:     }
167: }
```

If `RenderTexture`you generate it once , process it on the GPU, copy it to, `Graphics.CopyTexture(rt,tex2d);`and `Texture2D`save it as a Unity Asset with an editor script, it will be an asset that can be used without recalculation from now on, so I think it is a versatile technique. Listing 1.4 (lines 119,120)

In this implementation, it is implemented by writing to the texture `ComputeShader`. When doing a lot of processing, using the GPU is faster, so it is a useful technique, so please try to master it. As for the processing content, the position buffer and normal buffer of the vertex animation generated by the script are simply placed in each pixel as they are. Listing 1.5

Listing 1.5: MeshInfoTextureGen.compute

```
 1: #pragma kernel CSMain
 2:
 3: struct MeshInfo{
 4:     float3 position;
 5:     float3 normal;
 6: };
 7:
 8: RWTexture2D<float4> OutPosition;
 9: RWTexture2D<float4> OutNormal;
10: StructuredBuffer<MeshInfo> Info;
11: int VertCount;
12:
13: [numthreads(8,8,1)]
14: void CSMain (uint3 id : SV_DispatchThreadID)
15: {
16:     int index = id.y * VertCount + id.x;
17:     MeshInfo info = Info[index];
18:
19:     OutPosition[id.xy] = float4(info.position, 1.0);
20:     OutNormal[id.xy] = float4(info.normal, 1.0);
21: // Arrange the vertex information so that the x-axis of the
texture is the vertex ID and the y-axis direction is time.
22: }
```

Here is the texture generated from the script. Figure 1.8

Figure 1.8: Generated texture

This texture stores the vertices of the mesh in each sampled frame, one column in the x-axis direction. And the `uv.y`texture is designed so that the y-axis direction is time and you can specify the animation time by changing when sampling the texture.

What I would like you to pay attention to `Texture.FilterMode = Bilinear`is Tokoro. When sampling a texture, each pixel is interpolated with adjacent pixels, which causes the Shader to play an animated texture with a halfway time between the frame sampled by the script during texture generation and the next frame. When sampled, the frame-by-frame positions and normals will be automatically interpolated, resulting in smooth playback of the animation. The explanation is a little complicated!

And in this case, the Run animation is a loop animation `WrapMode = Repeat`. This interpolates the last and first pixels of the animation texture, resulting in a smooth looped animation. Of course, if you `WrapMode = Clamp`want to generate a texture from a non-looping animation, you need to set it to.

Next is the Shader for playing the generated animation texture. Listing 1.6

Listing 1.6: TextureAnimPlayer.shaer

```
1: Shader "Unlit/TextureAnimPlayer"
2: {
3:     Properties
4:     {
5:             _MainTex ("Texture", 2D) = "white" {}
6:             _PosTex("position texture", 2D) = "black"{}
7:             _NmlTex("normal texture", 2D) = "white"{}
8:             _DT ("delta time", float) = 0e
9:
```

```
10:             _Length ("animation length", Float) = 1
11:             [Toggle(ANIM_LOOP)] _Loop("loop", Float) = 0
12:     }
13:     SubShader
14:     {
15:             Tags { "RenderType"="Opaque" }
16: LOD 100 Cull Off
17:
18:             Pass
19:             {
20:                     CGPROGRAM
21:                     #pragma vertex vert
22:                     #pragma fragment frag
23:                     #pragma multi_compile ___ ANIM_LOOP
24: // It is convenient to make a multi-compile for the loop
25:
26:                     #include "UnityCG.cginc"
27:
28:                     #define ts _PosTex_TexelSize
29:
30:                     struct appdata
31:                     {
32:                             float2 uv : TEXCOORD0;
33:                     };
34:
35:                     struct v2f
36:                     {
37:                             float2 uv : TEXCOORD0;
38:                             float3 normal : TEXCOORD1;
39:                             float4 vertex : SV_POSITION;
40:                     };
41:
42:                     sampler2D _MainTex, _PosTex, _NmlTex;
43:                     float4 _PosTex_TexelSize;
44:                     float _Length, _DT;
45:
46:                            v2f vert (appdata v, uint vid :
SV_VertexID)
47:  //  You  can  get  the  vertex  ID  with  the  semantic  of
SV_VertexID
48:                     {
49:                                 float t = (_Time.y - _DT) /
_Length;
50: #if ANIM_LOOP
51:                             t = fmod(t, 1.0);
52: #else
53:                             t = saturate(t);
54: #endif
```

```
55:
56:                             float x = (vid + 0.5) * ts.x;
57:                             float y = t;
58: // uv.x is specified based on the vertex ID
59: // Set the time (t) to sample the animation in uv.y
60:
61:                             float4 pos = tex2Dlod(
62: _PosTex,
63: float4 (x, y, 0, 0)
64:                 );
65:                             float3 normal = tex2Dlod(
66:                         _NmlTex,
67: float4 (x, y, 0, 0)
68:                 );
69: // Sampling location and normal information from textures
70:
71:                             v2f o;
72:                                                     o.vertex  =
UnityObjectToClipPos(pos);
73: o.normal = UnityObjectToWorldNormal (normal);
74:                             o.uv = v.uv;
75:                             return o;
76:                     }
77:
78:                     half4 frag (v2f i) : SV_Target
79:                     {
80:                 half diff = dot(
81:                     i.normal,
82:                     float3(0, 1, 0)
83:                 ) * 0.5 + 0.5;
84:                                  half4 col = tex2D(_MainTex,
i.uv);
85:                             return diff * col;
86:                     }
87:                     ENDCG
88:             }
89:     }
90: }
```

Shaders that play animated textures use a technique called VertexTextureFetch (VTF). Simply put, the texture is sampled in the vertex shader and used to calculate the position of the vertices and each value. This method is often used for displacement mapping, etc.

I'm using the vertex ID to sample the texture, which `SV_VertexID`can be obtained semantically. Since the vertex information obtains both the position

information and the normal information from the texture, the part where there is only uv in the app data is also noteworthy. (`appdata`To `POSITION,NORMAL`not be a particularly error to define a semantic)

The UV when sampling the texture, `uv.y`is the normalized time of the animation (the value when the start of the animation is 0 and the end is 1.0). `uv.x`Is the vertex index (vid), `uv.x = (vid + 0.5) * _TexelSize.x`and what is this 0.5? You may think that this is the position `Bilinear`when sampling the texture with , `(n + 0.5) / テクスチャサイズ`because you can get the value in the uninterpolated texture, so add the value of 0.5 to the vertex ID and mesh We are getting the uninterpolated positions and normals between the vertices inside.

Listing 1.7: {TextureName} _TexelSize Float4 properties with texture size information (from Unity official manual)

```
x contains 1.0 / width
y contains 1.0 / height
z contains width
w contains height
```

1.6 Place 5000 animated horses

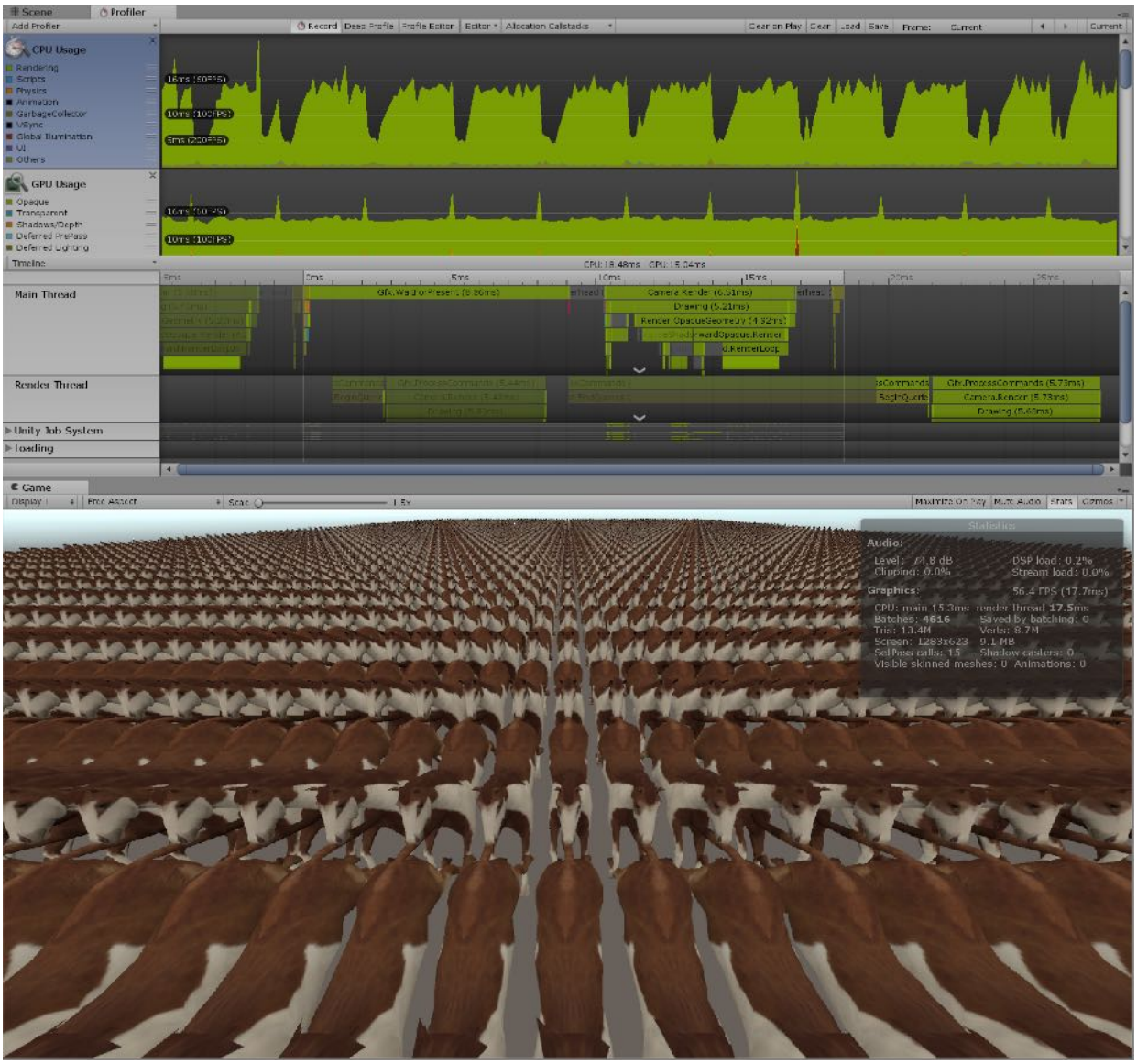

Figure 1.9: 5000 horses animated by texture

`SkinnedMeshRenderer`The `Renderer`animation is being played without using and with animation textures. FPS is greatly improved from 8 to 56.4 compared to when using skinning animation. Figure 1.9

* The GPU of the PC currently being written is GeForce MX150, which is the weakest of the NVIDIA Pascal GPUs. The rendering resolution was a bit smaller because we captured the profiler and the game window at the same time, but that shouldn't be that much as most of the processing load was mesh skinning. ..!!

Also, I would like you to pay attention to the fact that other optimization processing such as instancing support is not performed. `SkinnedMeshRenderer`Since we no longer use, it is now possible to draw with GPU instancing. It means that it is possible to pursue further performance by supporting Shader's instancing.

Although not explained here, the bird on the cover uses texture-animated `Graphics.DrawMeshInstancedIndirect()`birds to draw about 4000 birds at once. For Shader instancing support and other applications, check out my GitHub and other articles.

1.7 Limitations and consideration of application destinations

There are some restrictions on the technique using this texture. The memory for holding the texture is consumed depending on the number of vertices of the mesh and the length of the animation. You need to write a Shader to blend the animation. The state machine of AnimatorController cannot be used. Etc.

The biggest limitation is the maximum size of textures that can be used for each hardware. It can be 4K, 8K, 16K. In other words, in this method, the vertices of each frame of the mesh are arranged in a horizontal row, so the number of vertices of the mesh is limited by the texture size.

However, when you output a large number of objects, you should not output those with such a large number of vertices, so it is a good idea to accept the limit on the number of vertices as it is and make sure that the number of vertices of the mesh does not exceed the maximum size of the texture. maybe. If you want to use baking animation textures beyond this limit on the number of vertices, you can consider using multiple textures.

Alternatively, you can pre-calculate the matrix for each bone in the skeleton instead of the vertices of the mesh and save it in a texture or buffer. Since the skinning process itself is performed by Vertex Shader at the time of execution, the skinning process that was performed during normal mesh skinning `PostLateUpdate.UpdateAllSkinnedMeshes`is performed

`Camera.Render`collectively at the time of rendering, so the processing load is considerably lightened. Please, try it.

Since AnimatorController and Unity's state machine cannot be used, it is difficult to control the animation, so it is better to apply it to some deception such as mobs that repeat loop animation and swarms of flying birds and butterflies instead of the main character. I think it's good.

1.8 Summary

- When optimizing processing, it is important to profile properly and determine which processing is heavy.
- Skinning animation has a high processing load
- By saving the pre-skinned vertex coordinates in the texture, the execution processing load at the time can be reduced.
- There is room for further optimization, such as GPU instancing and GPU movement of characters.
- In real time, heavy processing such as skeleton matrices and simulation results can be saved in textures in advance to reduce the processing load at runtime, and there seems to be room for various applications.

Chapter 2 Gravitational N-Body Simulation

2.1 Introduction

In this chapter, we will explain the GPU implementation method of Gravitational N-Body Simulation, which is a method of simulating the movement of celestial bodies existing in outer space.

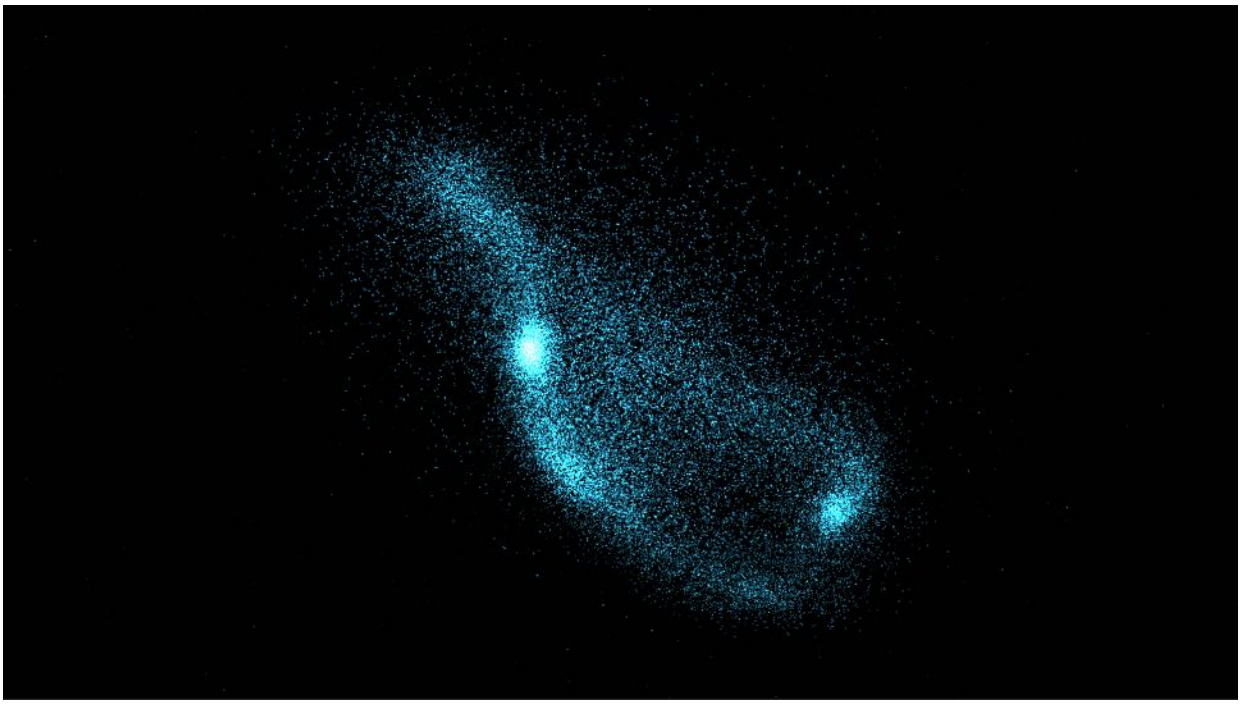

Figure 2.1: Result

The corresponding sample program is
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
"Assets / NBodySimulation".

2.2 What is N-Body simulation?

Simulations that calculate the interaction of N physical objects are collectively called N-Body simulations. There are many types of problems using N-Body simulation, and in particular, the problem of dealing with a system in which celestial bodies scattered in outer space attract each other by gravity to form a unit is called a **gravity multisystem problem.** I will. The algorithm explained in this chapter corresponds to this, and it means to solve the equation of motion of the gravitational multisystem using N-Body simulation.

In addition to the gravity multisystem problem, N-Body simulation is also available.

- Calculation of intermolecular force
- Analysis of dark matter
- Analysis of cluster collisions

It is applied in a wide range of fields, from small to magnificent.

2.3 Algorithm

Let's see what kind of mathematical formulas we will solve immediately.

The gravitational multisystem problem can be simulated by calculating the **universal gravitational** equation, which is a familiar equation for those who are taking high school physics, for all celestial bodies in space. However, in high school physics, I think that it was learned with such a description because it deals only with objects that are in a straight line.

Where f is the universal gravitational force, G is the gravitational constant, M and m are the masses of the two objects, and r is the distance between the objects. Of course, this can only determine **the magnitude** of the **force** (scalar amount) between two objects .

In this implementation, it is necessary to consider the movement in the 3D space inside Unity, so a **vector quantity** indicating the direction is required. Therefore, in order to find the vector of the force generated between two celestial bodies (i, j), the equation of universal gravitational force is described as follows.

Here, \ mbox {\ boldmath $ f $} _ {ij} is the vector of the force that the object i receives from the object j, m_i and m_j are the masses of the two objects, and r_ {ij} is the object i from the object j. Direction vector to. On the left side of the right side, the magnitude of the force is calculated in the same way as the equation of universal gravitational force that first appeared, and it is vectorized by multiplying the unit vector in the direction of receiving the force on the right side of the right side.

missing image: takao / vec

Figure: Meaning of formula

Furthermore, if the magnitude of the force that one object (i) receives from all the surrounding objects, not between two objects, is \ mbox {\ boldmath $ F $} _ {i} , it can be calculated as follows. ..

As shown in the formula, the force received from the surrounding celestial bodies can be calculated by taking the sum of all universal gravitational forces.

Also, to simplify the simulation, the equation can be rewritten using the Softening factor \ varepsilon as follows.

This makes it possible to ignore collisions even if the celestial bodies come to the same position (even if they calculate themselves, the result in Sigma will be 0).

Next, using the second law of motion m \ mbox {\ boldmath $ a $} = \ mbox {\ boldmath $ f $}, we will convert the force vector into the acceleration vector. First, we transform the second law of motion into the following equation.

Next, by substituting the above transformation equation into the equation of motion of the gravitational multisystem and rewriting it, the acceleration received by the celestial body can be calculated.

The simulation is now ready. Now that we were able to express the gravitational multisystem problem with mathematical formulas, how can we

incorporate these mathematical formulas into our program? I would like to explain it firmly in the next section.

2.4 Difference method

The above equation (\ ref {eq: newton}) is classified as a **differential equation** among the equations . Because, in the physical world, the relationship between **position, velocity, and acceleration** is as shown in the following image, and since acceleration is the second derivative of the position function, it is called a differential equation as it is.

Figure 2.2: Relationship between position, velocity, and acceleration

There are various methods for solving differential equations with a computer, but the most common one is the algorithm called the **difference method** . Let's start with a review of differentiation.

2.4.1 Differentiation

First, let's review the definition of mathematical differentiation. The derivative of the function f (t) is defined by the following equation.

Since it is difficult to understand what is shown by the formula alone, it becomes as follows when replaced with a graph.

Figure 2.3: Forward difference

You already know that the differential value of the function is the slope of the graph at t_n . After all, this graph shows the state that \ Delta t is made infinitely small to calculate the slope, and you can see that it represents the formula (\ ref {eq: delta}) itself. think.

2.4.2 Difference

You cannot handle "infinity" as a numerical value on a computer. Therefore, we will approximate it with a finite \ Delta t that is as small as possible .

With that in mind, if we rewrite the definition of differentiation to difference, we get the following equation.

It's a shape that has taken the limit as it is. I think it's okay to recognize that "an infinitesimal \ Delta t cannot be represented on a computer, so stop at a certain size \ Delta t to approximate it."

Here, the physical representation of Figure 2.2 above is as follows.

Figure 2.4: Relationship between position, velocity, and acceleration (mathematical formula)

That is, the formula (\ ref {eq: diffuse}) is compared to Figure 2.4 as follows:

Furthermore, when the formulas (\ ref {eq: x}) and (\ ref {eq: v}) are combined, the result is as follows.

This formula means that " the position coordinates after \ Delta t seconds from the current time t can be calculated if the acceleration and velocity of the current time are known." This is the basic idea when simulating with the finite difference method. In addition, the expression of such a differential equation using the difference method is called a **difference equation (recurrence formula)** .

When actually performing real-time simulation by the difference method, it is common to set the minute time \ Delta t (time step) to the drawing time of one frame (1/60 second at 60 fps).

2.5 implementation

Now, let's get into the implementation. The corresponding scene will be "SimpleNBodySimulation.unity".

2.5.1 CPU side program

Data structure of celestial bodies

First of all, we define the data structure of celestial particles. Looking at the equation (\ ref {eq: delta}), we can see that the physical quantities that one celestial body should have are "position, velocity, mass". Therefore, it seems good to define the following structure.

Body.cs

```
public struct Body
{
  public Vector3 position;
  public Vector3 velocity;
  public float mass;
}
```

Preparing the buffer

Next, set the number of particles you want to generate from the inspector, and secure a buffer for that number. Separate buffers for reading and writing to prevent data race conditions.

In addition, the number of bytes per structure can be obtained with the "Marshal.SizeOf (Type t)" function in the "System.Runtime.InteropServices" namespace.

SimpleNBodySimulation.cs

```
void InitBuffer()
{
  // Create buffer (for Read / Write) → Conflict prevention
  bodyBuffers = new ComputeBuffer[2];

     // Each element creates a buffer of Body structure for the
number of particles
  bodyBuffers[READ] = new ComputeBuffer(numBodies,
    Marshal.SizeOf(typeof(Body)));

  bodyBuffers[WRITE] = new ComputeBuffer(numBodies,
    Marshal.SizeOf(typeof(Body)));
}
```

Initial placement of celestial bodies

Then place the particles in space. First, create an array for the particles and give each element an initial value of the physical quantity. In the sample, the inside of the sphere was randomly sampled as the initial position, the velocity was 0, and the mass was randomly given.

Finally, set the buffer for the created array and you are ready * 1 .

[* 1] * For look adjustment, we have prepared a variable that can scale the position coordinates, but you do not have to worry because it has already been adjusted.

SimpleNBodySimulation.cs

```
void DistributeBodies()
{
  Random.InitState(seed);

  // For look adjustment
  float scale = positionScale
                                    *  Mathf.Max(1,  numBodies  /
DEFAULT_PARTICLE_NUM);

  // Prepare an array to set in the buffer
  Body[] bodies = new Body[numBodies];

  int i = 0;
  while (i < numBodies)
  {
    // Sampling inside the sphere
    Vector3 pos = Random.insideUnitSphere;

    // set in an array
    bodies[i].position = pos * scale;
    bodies[i].velocity = Vector3.zero;
    bodies[i].mass = Random.Range(0.1f, 1.0f);


    i++;
  }

  // Set the array in the buffer
  bodyBuffers[READ].SetData(bodies);
  bodyBuffers[WRITE].SetData(bodies);

}
```

Simulation routine

Finally, we will actually move the simulation. The following code is the part that is executed every frame.

First, set the value in the constant buffer of ComputeShader. For \ Delta t of the difference equation, use "Time.deltaTime" provided by Unity. In addition, due to the implementation of GPU, the number of threads and the number of thread blocks are also transferred.

After the calculation is completed, the calculation result of the simulation is stored in the buffer for writing, so the buffer is replaced in the last line so that it can be used as the buffer for reading in the next frame.

SimpleNBodySimulation.cs

```
void Update()
{
  // Transfer constants to compute shader
  // Δt
  NBodyCS.SetFloat ("_ DeltaTime", Time.deltaTime);
  // Speed attenuation rate
  NBodyCS.SetFloat("_Damping", damping);
  // Softening factor
  NBodyCS.SetFloat("_SofteningSquared", softeningSquared);
  // Number of particles
  NBodyCS.SetInt("_NumBodies", numBodies);

  // Number of threads per block
  NBodyCS.SetVector("_ThreadDim",
    new Vector4(SIMULATION_BLOCK_SIZE, 1, 1, 0));

  // Number of blocks
  NBodyCS.SetVector("_GroupDim",
              new       Vector4(Mathf.CeilToInt(numBodies       /
SIMULATION_BLOCK_SIZE), 1, 1, 0));

  // Transfer the buffer address
  NBodyCS.SetBuffer(0, "_BodiesBufferRead", bodyBuffers[READ]);
               NBodyCS.SetBuffer(0,        "_BodiesBufferWrite",
bodyBuffers[WRITE]);

  // Compute shader execution
  NBodyCS.Dispatch(0,
```

```
    Mathf.CeilToInt(numBodies / SIMULATION_BLOCK_SIZE), 1, 1);


  // Swap Read / Write (conflict prevention)
  Swap(bodyBuffers);
}
```

rendering

After the simulation calculation, issue an instance drawing instruction to the material that renders the particles. When rendering a particle, it is necessary to give the position coordinates of the particle to the shader, so transfer the calculated buffer to the shader for rendering.

ParticleRenderer.cs

```
void OnRenderObject ()
{
  particleRenderMat.SetPass (0);
  particleRenderMat.SetBuffer("_Particles"", bodyBuffers[READ]);

  Graphics.DrawProcedural(MeshTopology.Points, numBodies);
}
```

2.5.2 GPU side program

In N-Body simulation, it is necessary to calculate the interaction with all particles, so if it is calculated simply, the execution time will be O (n ^ 2) and performance cannot be achieved. Therefore, I will utilize the usage of Shared Memory described in Chapter 3 of UnityGraphicsProgramming Vol1.

Way of thinking

Data in the same block is stored in shared memory, which speeds up I / O. The following is a conceptual diagram that resembles a thread block as a tile.

Figure 2.5: Tile concept

Here, the row is the global thread (DispatchThreadID) that is running, and the column is the particle that is being exhausted in the thread. It's like the

columns you're running are shifting one by one to the right over time.

Also, the total number of tiles executed at the same time is (number of particles / number of threads in the group). In the sample, the number of threads in the block is 256 (SIMULATION_BLOCK_SIZE), so it is recognized that the tile content is actually 256x256 instead of 5x5.

All rows are running in parallel, but because they share data within the tile, they wait until all the columns running in the tile reach Sync (do not go to the right of the Sync layer). After reaching the Sync layer, the data of the next tile is reloaded into the shared memory.

Preparing a constant buffer

Describe the constant buffer for receiving the input from the CPU in the Compute Shader.

Also, prepare a buffer for storing particle data. This time, the Body structure is summarized in "Body.cginc". It is convenient to put together cginc for things that are likely to be reused later.

Finally, make a declaration to use shared memory.

SimpleNBodySimulation.compute

```
#include "Body.cginc"

// constant
cbuffer cb {
        float   _SofteningSquared, _DeltaTime, _Damping;
        uint    _NumBodies;
        float4  _GroupDim, _ThreadDim;
};


// Particle buffer
StructuredBuffer<Body> _BodiesBufferRead;
RWStructuredBuffer<Body> _BodiesBufferWrite;

// Shared memory (shared within the block)
groupshared Body sharedBody[SIMULATION_BLOCK_SIZE];
```

Tile implementation

Next, implement the tile.

SimpleNBodySimulation.compute

```
float3 computeBodyForce(Body body, uint3 groupID, uint3
threadID)
{

  uint start = 0; // start
  uint finish = _NumBodies;

  float3 acc = (float3)0;
  int currentTile = 0;

  // Execute for the number of tiles (number of blocks)
  for (uint i = start; i < finish; i += SIMULATION_BLOCK_SIZE)
  {
    // Store in shared memory
    // sharedBody [thread ID in block]
    // = _BodiesBufferRead [tile ID * total number of threads in
block + thread ID]
    sharedBody[threadID.x]
            =  _BodiesBufferRead[wrap(groupID.x  +  currentTile,
_GroupDim.x)
        * SIMULATION_BLOCK_SIZE + threadID.x];

    // Group sync
    GroupMemoryBarrierWithGroupSync();

    // Calculate the effect of gravity from the surroundings
    acc = gravitation(body, acc, threadID);

    GroupMemoryBarrierWithGroupSync();

    currentTile ++; // Go to the next tile
  }

  return acc;

}
```

Put the image of the for loop in the code in the next image.

Figure 2.6: Tile ID for loop

Interaction calculation

Since we controlled the movement of the tile in the previous loop, we will implement **the for loop in the tile** this time .

SimpleNBodySimulation.compute

```
float3 gravitation(Body body, float3 accel, uint3 threadID)
{

  // 100% survey
  // Execute for the number of threads in the block
  for (uint i = 0; i < SIMULATION_BLOCK_SIZE;)
  {
    accel = bodyBodyInteraction(accel, sharedBody[i], body);
    i++;
  }

  return accel;
}
```

This completes the 100% survey in the tile. Also, at the timing after the return of this function, it waits until all the threads in the tile complete the process.

Then implement the expression (\ ref {eq: first}) as follows:

SimpleNBodySimulation.compute

```
// Calculation in Sigma
float3 bodyBodyInteraction(float3 acc, Body b_i, Body b_j)
{
  float3 r = b_i.position - b_j.position;

  // distSqr = dot(r_ij, r_ij) + EPS^2
  float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
  distSqr += _SofteningSquared;

  // invDistCube = 1/distSqr^(3/2)
  float distSixth = distSqr * distSqr * distSqr;
  float invDistCube = 1.0f / sqrt(distSixth);

  // s = m_j * invDistCube
  float s = b_j.mass * invDistCube;
```

```
  // a_i =  a_i + s * r_ij
  acc += r * s;

  return acc;
}
```

This completes the program for calculating the total acceleration.

Update position with finite difference method

Next, the coordinates and velocity of the particles in the next frame are calculated using the acceleration calculated so far.

SimpleNBodySimulation.compute

```
[numthreads(SIMULATION_BLOCK_SIZE,1,1)]
void CSMain (
        uint3 groupID: SV_GroupID, // Group ID
          uint3 threadID: SV_GroupThreadID, // Thread ID in the
group
        uint3 DTid: SV_DispatchThreadID // Global Thread ID
) {

        // Current global thread index
        uint index = DTid.x;

        // Read particles from buffer
        Body body = _BodiesBufferRead[index];

                float3  force  =  computeBodyForce(body,  groupID,
threadID);

        body.velocity + = force * _DeltaTime;
        body.velocity *= _Damping;

        // Difference
        body.position + = body.velocity * _DeltaTime;

        _BodiesBufferWrite[index] = body;

}
```

The position coordinates of the celestial body have been updated. This completes the simulation of the movement of the celestial body!

2.6 How to render grains

In this section, I would like to supplement the drawing method of GPU particles, which was insufficiently explained in the previous article * 2 .

[* 2] The same particle rendering is performed in "Unity Graphics Programming Vol.1-Chapter 5 Fluid Simulation by SPH Method".

2.6.1 Billboard

A billboard is a simple planar object that always faces the camera. It is no exaggeration to say that most particle systems in the world are implemented by billboards. In order to implement the billboard simply, we need to make good use of the view transformation matrix.

The view transformation matrix contains numerical information that returns the camera position and rotation to the origin. In other words, by multiplying all the objects in space by the view transformation matrix, it is transformed into a coordinate system with the camera as the origin.

Therefore, for a billboard that has the characteristic **of facing the direction of** the **camera,** it seems that the **inverse matrix of the** view transformation matrix containing only the rotation information should be multiplied as the model transformation matrix. (Although it will be described later, it is difficult to understand, so the illustration is shown in Figure 2.8 .)

Billboard implementation

First of all, create a Quad mesh to draw the particles. This can be easily achieved by extending one vertex to a Quad parallel to the xy plane with a geometry shader * 3 .

[* 3] For a detailed explanation of geometry shaders, see UnityGraphicsProgramming Vol.1 "Growing grass with geometry shaders".

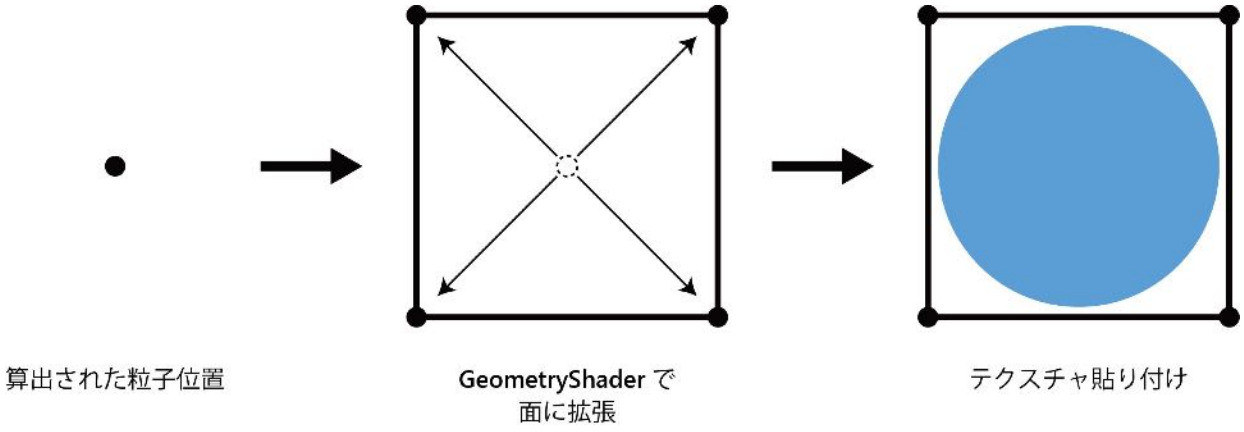

Figure 2.7: Quad extension with Geometry Shader

If you give this quad an inverse matrix * 4 that cancels the translation component of the view transformation matrix as a model transformation matrix, you can create a quad that faces the camera on the spot.

[* 4] Since it is known that the inverse matrix of the view transformation matrix can be simply transposed, it is implemented here by transposing.

I don't know what you're talking about, so here's an explanatory diagram.

Figure 2.8: How the billboard works

Furthermore, by applying a view and projection matrix to the billboard facing the camera, it can be converted to the coordinates on the screen. The shader that implements these is shown below.

ParticleRenderer.shader

```
[maxvertexcount(4)]
void geom(point v2g input[1], inout TriangleStream<g2f>
outStream) {
  g2f o;

  float4 pos = input[0].pos;

  float4x4 billboardMatrix = UNITY_MATRIX_V;

  // Take out only the rotating component
  billboardMatrix._m03 = billboardMatrix._m13 =
    billboardMatrix._m23 = billboardMatrix._m33 = 0;
```

```
  for (int x = 0; x < 2; x++) {
    for (int y = 0; y < 2; y++) {
      float2 uv = float2(x, y);
      o.uv = uv;

      o.pos = pos
            + mul(transpose(billboardMatrix), float4((uv * 2 -
float2(1, 1))
        * _Scale, 0, 1));

      o.pos = mul(UNITY_MATRIX_VP, o.pos);

      o.id = input[0].id;

      outStream.Append(o);
    }
  }

  outStream.RestartStrip();
}
```

2.7 Results

Let's see the result of the above simulation.

Figure 2.9: Simulation results (feeling of korejanai ...)

Looking at the movement, all the particles are gathered in the center, which is not visually interesting. Therefore, we will add some ideas in the next section.

2.8 Ingenuity to make it visually interesting

The corresponding scene is "NBodySimulation.unity". Even if it is a device, there is only one line to change, and the tile calculation part will be cut off in the middle as follows.

NBodySimulation.compute

```
float3 computeBodyForce(Body body, uint3 groupID, uint3
threadID)
{
   . . .

  uint finish = _NumBodies / div; // Cut in the middle

   . . .

}
```

As a result, the calculation of the interaction is discarded in the middle without being exhausted, but as a result, it is not affected by all the particles, so some particle lumps are generated and aggregated into one point. Will never happen.

Then, the interaction between the parts of multiple masses produces a more dynamic movement as shown in Figure 2.1 at the beginning .

2.9 Summary

In this chapter, we explained the GPU implementation method of Gravitational N-Body Simulation. From small atoms to the large universe, the potential of N-Body simulation is infinite. Why don't you try to create your own universe on Unity? I hope you can help me even a little!

2.10 Reference

- GPU Gems 3 - Chapter 31. Fast N-Body Simulation with CUDA
- What can be learned from N-body simulation? --Michiko Fujii, Kagoshima University http://www.astro-wakate.org/ss2011/web/ss11_proceedings/proceeding/galaxy_fujii.pdf
- Quaternion and Billboard-wgld.org https://wgld.org/d/webgl/w035.html

Chapter 3 ScreenSpaceFluidRendering

3.1 Introduction

This chapter introduces **Screen Space Fluid Rendering** by **Deferred Shading** as one of the particle rendering methods .

3.2 Screen Space Fluid Renderingとは

Traditionally, the Martin Cubes method is used to render fluid-like continuums, but it is relatively computationally intensive and not suitable for detailed drawing in real-time applications. Therefore, a method called **Screen Space Fluid Rendering** was devised as a method for drawing particle-based fluids at high speed .

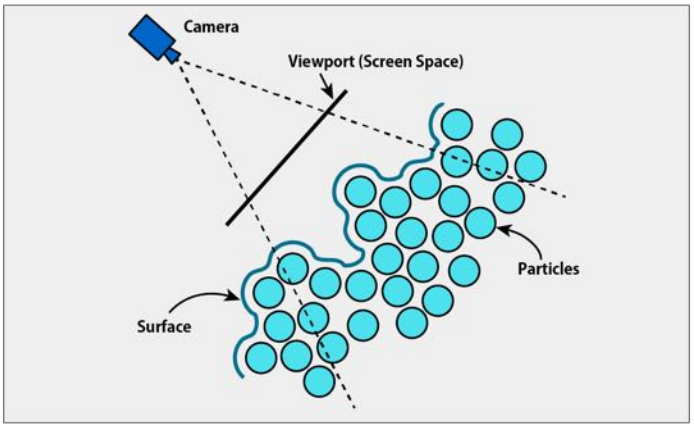

Figure 3.1: Schematic diagram of Screen Space Fluid Rendering

This creates a surface from the depth of the surface of the particles in the screen space visible to the camera, as shown in Figure 3.1.

A technique called **Deferred Rendering** is used to generate this surface geometry .

3.3 What is Deferred Rendering?

2-dimensional screen space (screen space) in the **shading (shadow calculation)** is a technology to perform. For the sake of distinction, the traditional type of technique is called **Forward Rendering** .

Figure 3.2 outlines the traditional **Forward Rendering** and **Deferred Rendering** rendering pipelines.

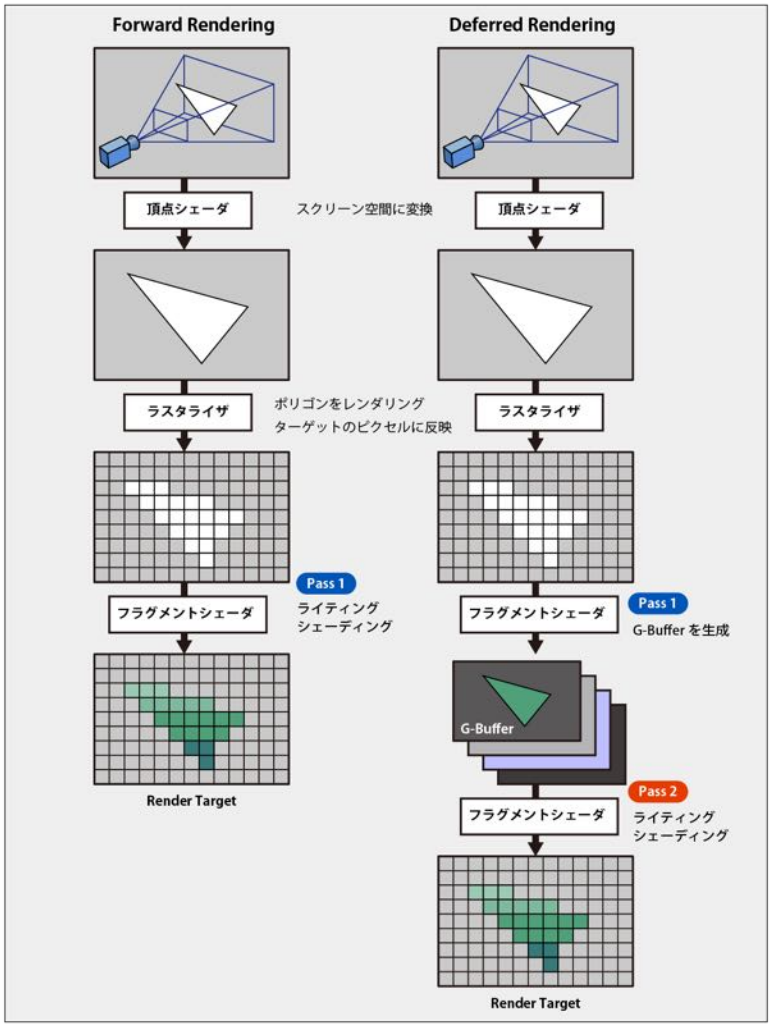

Figure 3.2: Comparison of Foward Rendering and Deferred Rendering pipelines

In the case of **Forward Rendering** , lighting and shading processing are performed in **the first pass** of the shader, but in **Deferred Rendering** , 2D image information such as **normal** , **position** , **depth** , **diffuse color** required for shading is generated, and **G-Buffer** Store in a buffer called. In the second pass, that information is used to perform lighting and shading to obtain the final rendering result. This delays **the** actual rendering to **the second pass (and beyond)** , hence the name **"Deferred" Rendering** .

The advantage of **Deferred Rendering** is

- Many light sources can be used
- When shading, you only need to calculate the displayed area, so you can minimize the number of times the fragment shader is executed.
- Geometry can be transformed
- G-Buffer information can be used in PostEffect etc. (SSAO etc.)

The downside is

- Weak in translucent expression
- Some algorithms such as MSAA may not be effective enough for antialiasing.
- Difficult to use multiple materials
- Not supported on Orthographic cameras

There are some restrictions such as trade-offs, so it is necessary to consider them before making a decision.

Terms of use for Deferred Rendering in Unity

Deferred Rendering has the following usage conditions, and this sample program may not work depending on the environment. ..

- Only available in Unity Pro
- Multi-render target (MRT) is enabled
- Shader model 3.0 and above

- Requires a graphics card that supports depth slender textures and two-sided stencil buffers

Also, **Deferred Rendering** is not supported when using **Orthographic** projection , and **Forward Rendering** is used when the camera's projection mode is set to **Orthographic** .

3.4 G-Buffer (geometry buffer)

Information about (2D texture) in screen space, such as **normals** , **positions** , and **diffuse colors** used for shading and lining calculations, is called **G-Buffer** . In the **G-Buffer path** of Unity's rendering pipeline , each object is rendered once and rendered into a **G-Buffer texture** , generating the following information by default:

Form 3.1:

render target	**format**	**data type**
RT0	ARGB32	Diffuse color (RGB), Occulusion (A)
RT1	ARGB32	Specular color (RGB), Roughness (A)
RT2	ARGB2101010	World space normal (RGB)
RT3	ARGB2101010	Emission + (Ambient + Reflections + Lightmaps)
Z-buffer		Depth + Stencil

These **G-Buffer textures** are set as global properties and can be retrieved within the shader.

Table 3.2:

shader property name	**data type**
_CameraGBufferTexture0	Diffuse color (RGB), occulusion (A)
_CameraGBufferTexture1	Specular color (RGB)
_CameraGBufferTexture2	World space normal (RGB)

shader property name	data type
_CameraGBufferTexture3	Emission + (Ambient + Reflections + Lightmaps)
_CameraDepthTexture	Depth + Stencil

If you open **Assets / ScreenSpaceFluidRendering / Scenes / ShowGBufferTest** in the sample code, you can see how this **G-Buffer** is acquired and displayed on the screen.

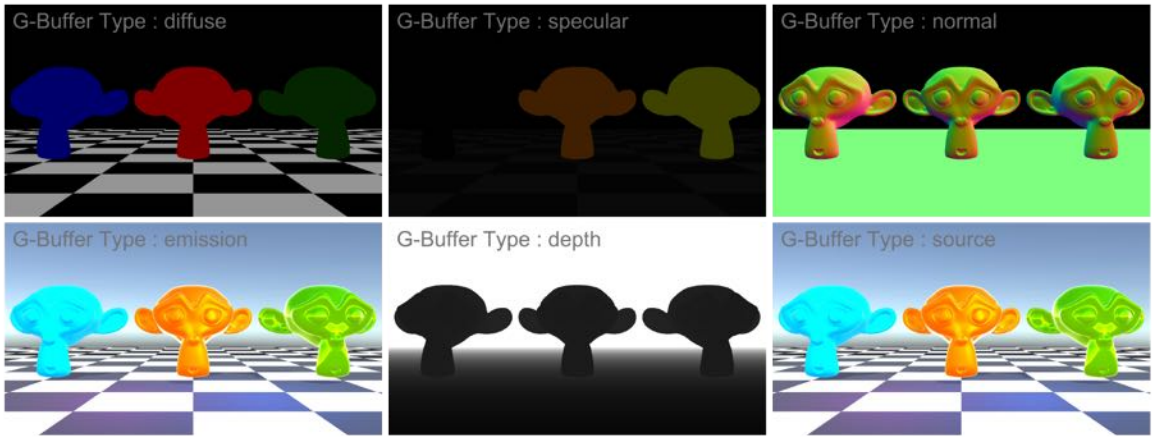

Figure 3.3: G-Buffer generated by default

3.5 About Command Buffer

The sample program introduced in this chapter uses Unity's API called **CommandBuffer** .

The drawing process itself is not performed in the method written in the script executed by the **CPU** . Instead, it is added to the **GPU** 's understandable list of **rendering commands** , called the **graphics command buffer,** and the generated **command buffer** is read directly by the **GPU** and executed to actually draw the object.

Rendering commands provided by Unity are , for example, methods such as **Graphics.DrawMesh ()** and **Graphics.DrawProcedural ()** .

Of Unity of API **CommandBuffer** By using, Unity of the **rendering pipeline** to a specific point in the **command buffer (a list of the rendering commands)** by inserting a, Unity of the **rendering pipeline** can be extended to.

You can see some sample projects using **CommandBuffer** here.

https://docs.unity3d.com/ja/current/Manual/GraphicsCommandBuffers.html

3.6 Coordinate system and coordinate transformation

In the following, we will briefly explain the 3DCG graphics pipeline and coordinate system to understand the contents of the calculations performed on the screen space.

Homogeneous Coordinates

When considering a three-dimensional position vector (x, y, z), it is sometimes treated as a four-dimensional one such as (x, y, z, w), which is called Homogeneous Coordinates. .. By thinking in four dimensions in this way, you can effectively multiply 4x4 Matrix. The calculation of the coordinate transformation is basically done by multiplying the 4x4 Matrix, so the position vector is expressed in 4 dimensions like this.

The conversion between homogeneous coordinates and non-homogeneous coordinates is done in this way. (x / w, y / w, z / w, 1) = (x, y, z, w)

Object Space (object coordinate system, local coordinate system, model coordinate system)

The coordinate system around which the object itself is central.

World Space (world coordinate system, global coordinate system)

World Space is a coordinate system that shows how multiple objects are spatially related in a scene, centered on the scene. The World Space is transformed from the **Object Space** by a **Modeling Transform** that moves, rotates, and scales the object .

Eye (View) Space (viewpoint coordinate system, camera coordinate system)

Eye Space is a coordinate system centered on the drawing camera and with its viewpoint as the origin. Orientation on the position and the camera of the camera, was to define the information, such as a camera focus direction of the orientation of the **View Matrix** According to the **View Transform** by making a **World Space** will be converted from.

Clip Space (Clipping coordinate system, Clip coordinate system)

Clip Space , the above **View Matrix** other parameters of the camera defined by and defines a field of view (FOV) · aspect ratio · near clip · _far clip **Projection Matrix** and **View Space** coordinate system obtained by converting multiplying the is. This transformation is called the **Projection Transform** , which clips the space drawn by the camera.

Normalized Device Coordinates

Coordinate value obtained by **Clip Space** xyz By dividing each element by w, the range is -1 <= x <= 1, -1 <= y <= 1, 0 <= z <= 1. All position coordinates are normalized. The coordinate system obtained by this is called **Normalized Device Coordinates (NDC)** . This transformation is called **Persepective Devide,** and the objects in the foreground are drawn larger and the ones in the back are drawn smaller.

Screen (Window) Space (screen coordinate system, window coordinate system)

A coordinate system in which the normalized values obtained by **Normalized Device Coordinates** are converted to match the screen

resolution. In the case of Direct3D, the origin is the upper left.

Deferred Rendering calculates based on the image in this screen space, but if necessary, it calculates and uses the information of any coordinate system by multiplying it by the inverse matrix of each transformation, so this rendering pipeline is used. It's important to understand.

Figure 3.3 illustrates the relationship between the 3DCG graphics pipeline, the coordinate system, and coordinate transformation.

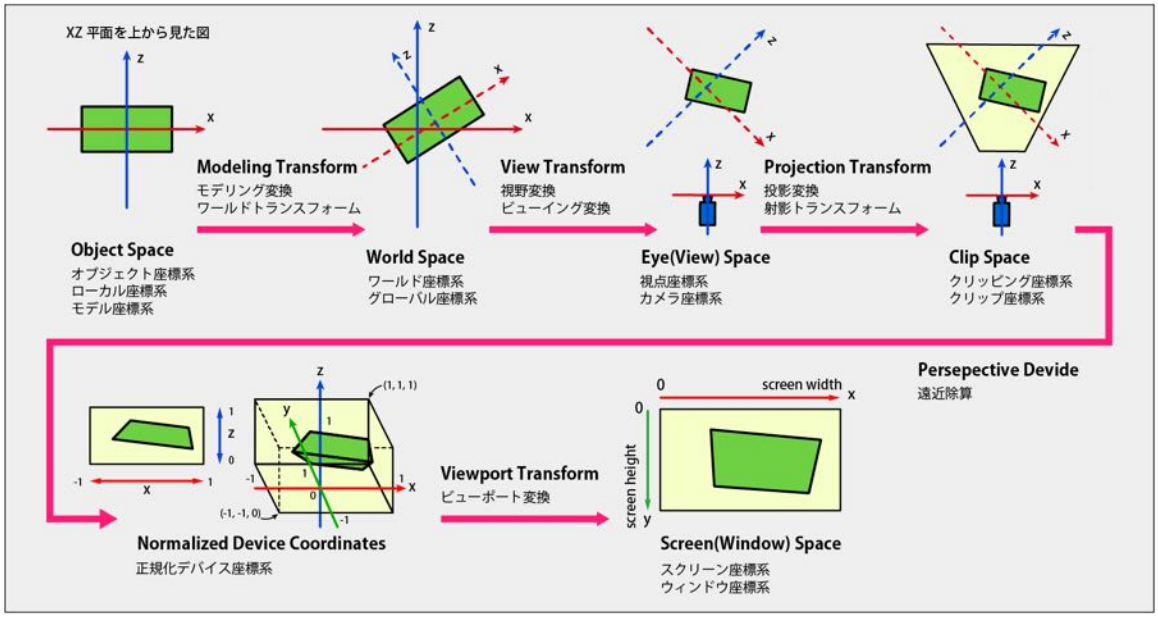

Figure 3.4: Coordinate system, flow of coordinate transformation

3.7 Explanation of implementation

Of the sample code

Assets/ScreenSpaceFluidRendering/Scenes/ScreenSpaceFluidRendering

Please open the scene.

3.7.1 Algorithm Overview

The general algorithm for **Screen Space Fluid Rendering** is as follows.

1. Draw particles to generate a depth image of screen space
2. Apply a blur effect to the depth image to make it smooth
3. Calculate surface normals from depth
4. Calculate surface shading

* In this sample code, even the creation of surface geometry is performed. We do not perform transparent expressions.

3.7.2 Script configuration

Table 3.3:

Script name	**function**
ScreenSpaceFluidRenderer.cs	Main script
RenderParticleDepth.shader	Find the depth of the particle's screen space
BilateralFilterBlur.shader	Blur effect that attenuates with depth
CalcNormal.shader	Find the normal from the depth information of the screen space
RenderGBuffer.shader	Write depth, normals, color information, etc. to G-Buffer

3.7.3 Create CommandBuffer and register it in the camera

ScreenSpaceFluidRendering.cs of **OnWillRenderObject** Within the function, **CommandBuffer** to create a, at any point of the camera of rendering path **CommandBuffer** make the process of registering.

Below is an excerpt of the code

ScreenSpaceFluidRendering.cs

```
// Called when the attached mesh renderer is in the camera
void OnWillRenderObject ()
{
  // If it is not active, release it and do nothing after that
  var act = gameObject.activeInHierarchy && enabled;
```

```
  if (!act)
  {
    CleanUp();
    return;
  }
  // If there is no camera currently rendering, do nothing after
that
  var cam = Camera.current;
  if (!cam)
  {
    return;
  }

  // For the camera currently rendering
  // If CommandBuffer is not attached
  if (!_cameras.ContainsKey(cam))
  {
    // Create Command Buffer information
    var buf = new CmdBufferInfo ();
    buf.pass   = CameraEvent.BeforeGBuffer;
    buf.buffer = new CommandBuffer();
    buf.name = "Screen Space Fluid Renderer";
    // In the path on the camera's rendering pipeline before the
G-Buffer was generated,
    // Add the created CommandBuffer
    cam.AddCommandBuffer(buf.pass, buf.buffer);

      // Add a camera to the list that manages cameras with
CommandBuffer added
    _cameras.Add(cam, buf);
  }
```

The Camera.AddCommandBuffer (CameraEvent evt, Rendering.CommandBuffer buffer) method adds a **command buffer** to the camera that runs at any path . Here, **CameraEvent.BeforeGBuffer** in **immediately before the G-Buffer is generated** and specifies the location of where any **command buffer** by inserting the, you can generate a calculated geometry on the screen space. The added **command buffer** is deleted by using the **RemoveCommandBuffer** method when the application is executed or the object is disabled . The process of deleting the **command buffer** from the camera is implemented in the **Cleanup** function.

Then, **CommandBuffer** to the **rendering command** will continue to register a. At that time, at the beginning of frame update, delete all buffer

commands by **CommandBuffer.Clear** method.

3.7.4 Generate a depth image of particles

Generates a **point sprite** based on **the data** of the **vertices of** a given particle and calculates **the depth texture** in that **screen space** .

The code is excerpted below.

ScreenSpaceFluidRendering.cs

```
// ---------------------------------------------------------------
-------
// 1. Draw particles as point sprites to get depth and color
data
// ---------------------------------------------------------------
-------
// Get the shader property ID of the depth buffer
int depthBufferId = Shader.PropertyToID("_DepthBuffer");
// Get a temporary RenderTexture
buf.GetTemporaryRT (depthBufferId, -1, -1, 24,
  FilterMode.Point, RenderTextureFormat.RFloat);

// Specify color buffer and depth buffer as render targets
buf.SetRenderTarget
(
  new RenderTargetIdentifier(depthBufferId),  // デプス
    new  RenderTargetIdentifier  (depthBufferId)  //  For  depth
writing
);
// Clear color buffer and depth buffer
buf.ClearRenderTarget(true, true, Color.clear);

// Set the particle size
_renderParticleDepthMaterial.SetFloat          ("_ParticleSize",
_particleSize);
// Set particle data (ComputeBuffer)
_renderParticleDepthMaterial.SetBuffer("_ParticleDataBuffer",
_particleControllerScript.GetParticleDataBuffer());

// Draw particles as point sprites to get a depth image
buf.DrawProcedural
(
  Matrix4x4.identity,
  _renderParticleDepthMaterial,
```

```
  0,
  MeshTopology.Points,
  _particleControllerScript.GetMaxParticleNum()
);
```

RenderParticleDepth.shader

```
// ------------------------------------------------------------
-------
// Vertex Shader
// ------------------------------------------------------------
-------
v2g vert(uint id : SV_VertexID)
{
  v2g or = (v2g) 0;
  FluidParticle fp = _ParticleDataBuffer[id];
  o.position = float4(fp.position, 1.0);
  return o;
}

// ------------------------------------------------------------
-------
// Geometry Shader
// ------------------------------------------------------------
-------
// Position of each vertex of the point sprite
static const float3 g_positions[4] =
{
  float3(-1, 1, 0),
  float3( 1, 1, 0),
  float3(-1,-1, 0),
  float3( 1,-1, 0),
};
// UV coordinates of each vertex
static const float2 g_texcoords[4] =
{
  float2(0, 1),
  float2(1, 1),
  float2(0, 0),
  float2(1, 0),
};

[maxvertexcount(4)]
void   geom(point   v2g   In[1],   inout   TriangleStream<g2f>
SpriteStream)
{
  g2f o = (g2f) 0;
  // Position of the center vertex of the point sprite
```

```
  float3 vertpos = In[0].position.xyz;
  // 4 point sprites
  [unroll]
  for (int i = 0; i < 4; i++)
  {
     // Find and substitute the position of the point sprite in
the clip coordinate system
    float3 pos = g_positions[i] * _ParticleSize;
    pos = mul(unity_CameraToWorld, pos) + vertpos;
    o.position = UnityObjectToClipPos(float4(pos, 1.0));
       // Substitute the UV coordinates of the point sprite
vertices
    o.uv       = g_texcoords[i];
     // Find and substitute the position of the point sprite in
the viewpoint coordinate system
     o.vpos      = UnityObjectToViewPos(float4(pos, 1.0)).xyz *
float3(1, 1, 1);
    // Substitute the size of the point sprite
    o.size     = _ParticleSize;

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

// ------------------------------------------------------------------
-------
// Fragment Shader
// ------------------------------------------------------------------
-------
struct fragmentOut
{
  float  depthBuffer  : SV_Target0;
  float  depthStencil : SV_Depth;
};

fragmentOut frag(g2f i)
{
  // Calculate normal
  float3 N = (float3) 0;
  N.xy = i.uv.xy * 2.0 - 1.0;
  float radius_sq = dot(N.xy, N.xy);
  if (radius_sq > 1.0) discard;
  N.z = sqrt(1.0 - radius_sq);

  // Pixel position in clip space
  float4 pixelPos     = float4(i.vpos.xyz + N * i.size, 1.0);
  float4 clipSpacePos = mul(UNITY_MATRIX_P, pixelPos);
```

```
  // depth
      float depth = clipSpacePos.z / clipSpacePos.w; //
normalization

  fragmentOut o = (fragmentOut) 0;
  o.depthBuffer  = depth;
  o.depthStencil = depth;

  return o;
}
```

The C # script first generates a **temporary RenderTexture** for calculations in screen space . In the command buffer, use the **CommandBuffer.GetTemporaryRT** method to create **temporary RenderTexture** data and use it. **In** the first argument of the **GetTemporaryRT** method, pass the **unique ID** of the shader property of the buffer you want to create . In shader **unique ID** and is, in order to access the properties of the shader that is generated each time the game scene of Unity is executed **int type unique ID** in, **Shader.PropertyToID** the method **property name** can be generated by passing the I can. (Since this unique ID is different in game scenes where the execution timing is different, its value cannot be retained or shared with other applications over the network.)

The second and third arguments of the **GetTemporaryRT** method specify the resolution. **If -1** is specified, the resolution (Camera pixel width, height) of the camera currently rendering in the game scene will be passed.

The fourth argument specifies the number of bits in the depth buffer. **In _DepthBuffer** , we also want to write the depth + stencil value, so specify a value of 0 or more.

The resulting RenderTexture is, **CommandBuffer.SetRenderTarget** in the method, **the render target** specified, **ClearRenderTarget** in the method, leave the clear. If this is not done, it will be overwritten every frame and will not be drawn properly.

The CommandBuffer.DrawProcedural method draws the particle data and calculates **the color and depth textures** in **screen space** . The figure below shows this calculation.

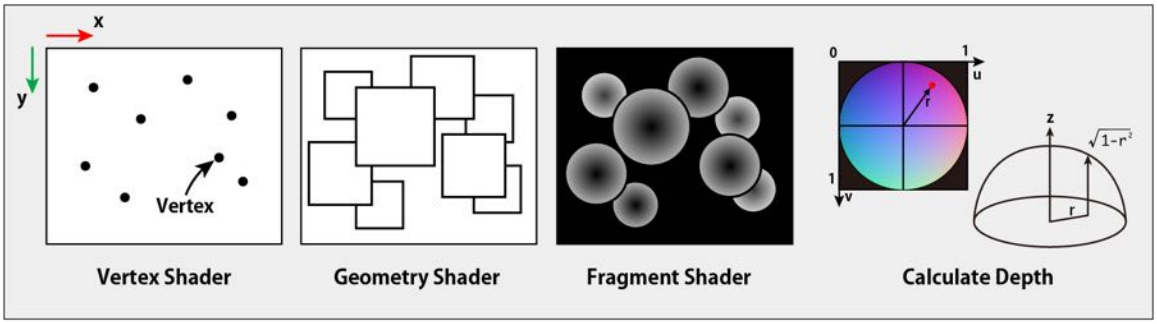

Figure 3.5: Depth image calculation

The Vertex shader and **Geometry** shader generate **point sprites** that billboard in the viewpoint direction from the given particle data . **The Fragment** shader calculates the hemispherical normal from the UV coordinates of the **point sprite** and uses this to get **a depth image in screen space** .

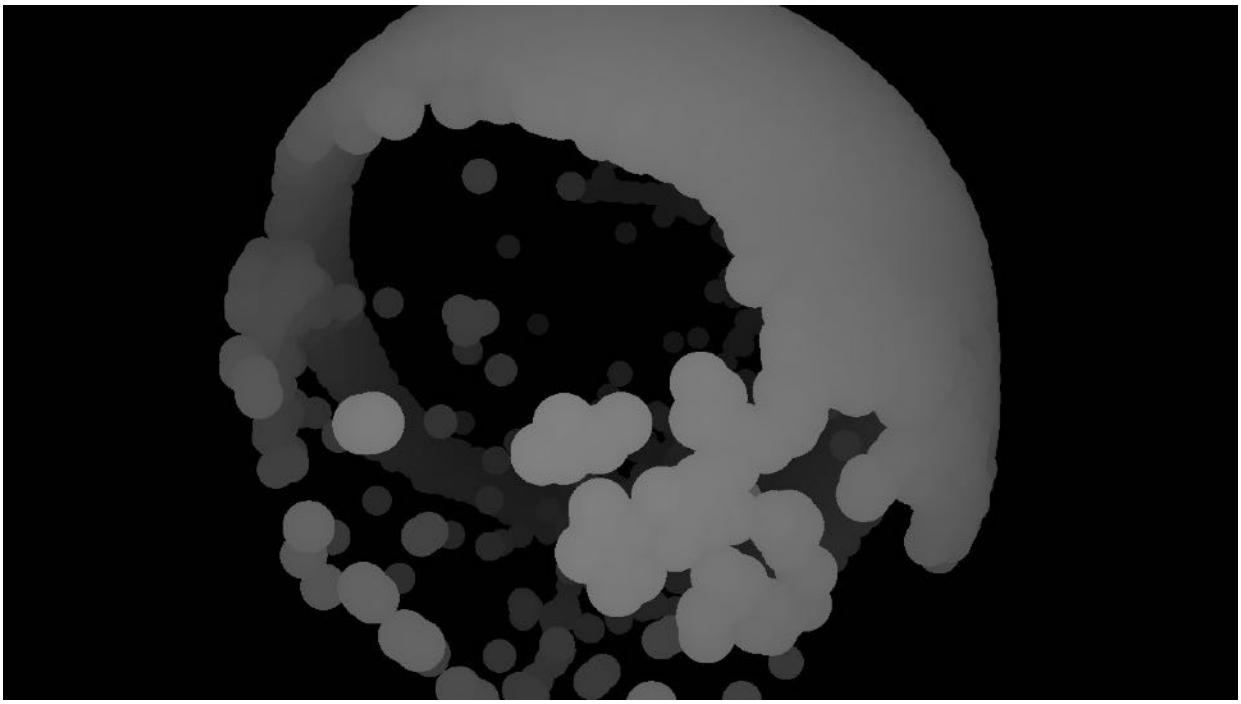

Figure 3.6: Depth image

3.7.5 Apply a blur effect to the depth image to make it smooth

By applying a blur effect to smooth the **obtained depth image** , it is possible to draw the image as if it is connected by blurring the boundary with adjacent particles. Here, a filter is used so that the offset amount of the blur effect is attenuated according to the depth.

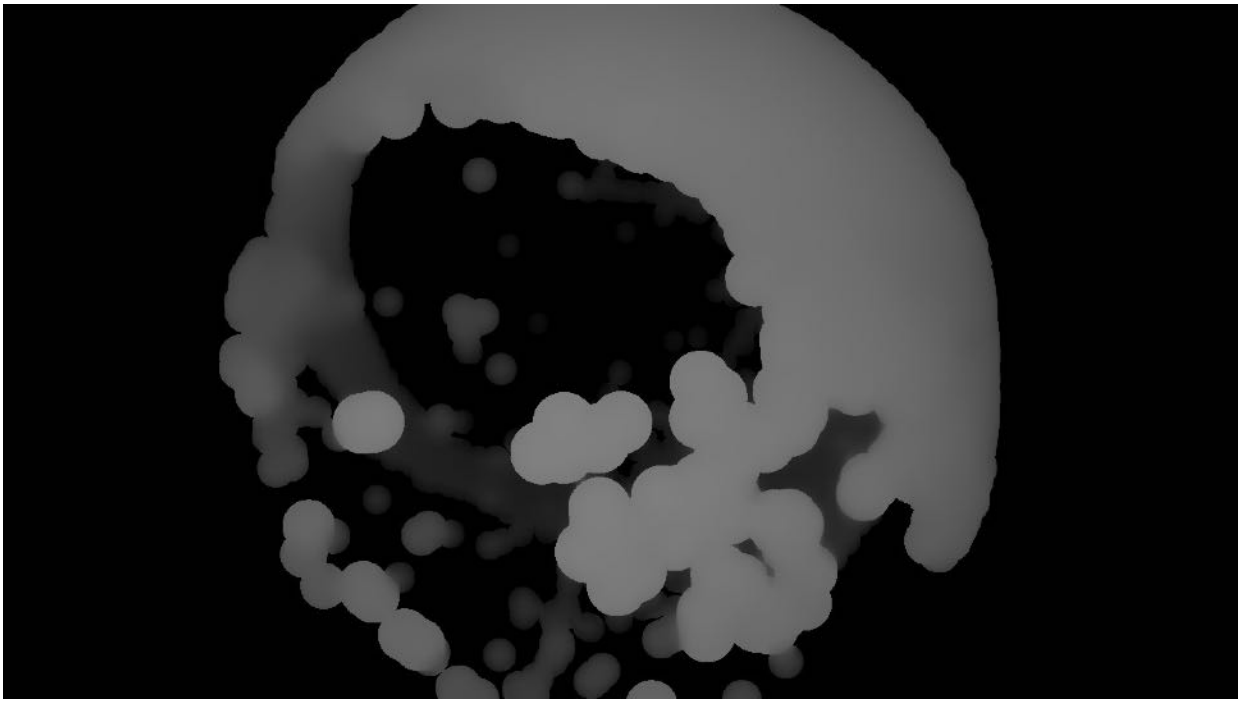

Figure 3.7: Blurred depth image

3.7.6 Calculate surface normals from depth

Calculate the **normal** from the **blurred depth image** . The normal is calculated by performing **partial differentiation** in the X and Y directions .

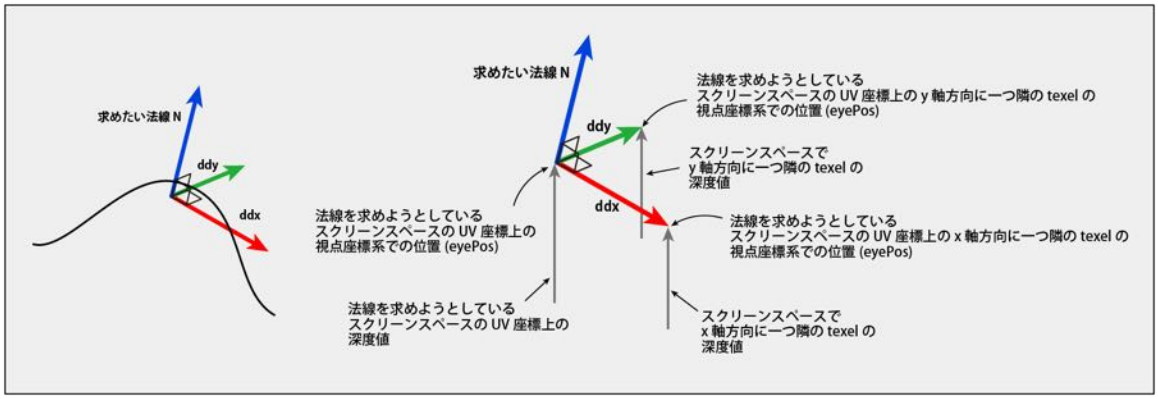

Figure 3.8: Normal calculation

Below is an excerpt of the code

CalcNormal.shader

```
// ------------------------------------------------------------
-------
// Fragment Shader
// ------------------------------------------------------------
-------
// Find the position in the viewpoint coordinate system from the
UV of the screen
float3 uvToEye(float2 uv, float z)
{
  float2 xyPos = uv * 2.0 - 1.0;
  // Position in the clip coordinate system
  float4 clipPos = float4(xyPos.xy, z, 1.0);
  // Position in the viewpoint coordinate system
  float4 viewPos = mul(unity_CameraInvProjection, clipPos);
  // normalization
  viewPos.xyz = viewPos.xyz / viewPos.w;

  return viewPos.xyz;
}

// Get the depth value from the depth buffer
float sampleDepth(float2 uv)
{
#if UNITY_REVERSED_Z
  return 1.0 - tex2D(_DepthBuffer, uv).r;
#else
```

```
  return tex2D(_DepthBuffer, uv).r;
#endif
}

// Get the position in the viewpoint coordinate system
float3 getEyePos(float2 uv)
{
  return uvToEye(uv, sampleDepth(uv));
}

float4 frag(v2f_img i) : SV_Target
{
  // Convert screen coordinates to texture UV coordinates
  float2 uv = i.uv.xy;
  // get depth
  float depth = tex2D(_DepthBuffer, uv);

  // Discard pixels if depth is not written
#if UNITY_REVERSED_Z
  if (Linear01Depth(depth) > 1.0 - 1e-3)
    discard;
#else
  if (Linear01Depth(depth) < 1e-3)
    discard;
#endif
  // Store texel size
  float2 ts = _DepthBuffer_TexelSize.xy;

   // Find the position of the viewpoint coordinate system (as
seen from the camera) from the uv coordinates of the screen
  float3 posEye = getEyePos(uv);

  // Partial differential with respect to x
  float3 ddx  = getEyePos(uv + float2(ts.x, 0.0)) - posEye;
  float3 ddx2 = posEye - getEyePos(uv - float2(ts.x, 0.0));
  ddx = abs(ddx.z) > abs(ddx2.z) ? ddx2 : ddx;

  // Partial differential with respect to y
  float3 ddy = getEyePos(uv + float2(0.0, ts.y)) - posEye;
  float3 ddy2 = posEye - getEyePos(uv - float2(0.0, ts.y));
  Dy = abs (Dy.z)> abs (Dy2.z)? dy2: should;

   // Find the normal orthogonal to the vector found above from
the cross product
  float3 N = normalize(cross(ddx, ddy));

  // Change the normal in relation to the camera position
  float4x4 vm = _ViewMatrix;
```

```
  N = normalize(mul(vm, float4(N, 0.0)));

  // Convert (-1.0 to 1.0) to (0.0 to 1.0)
  float4 col = float4(N * 0.5 + 0.5, 1.0);

  return col;
}
```

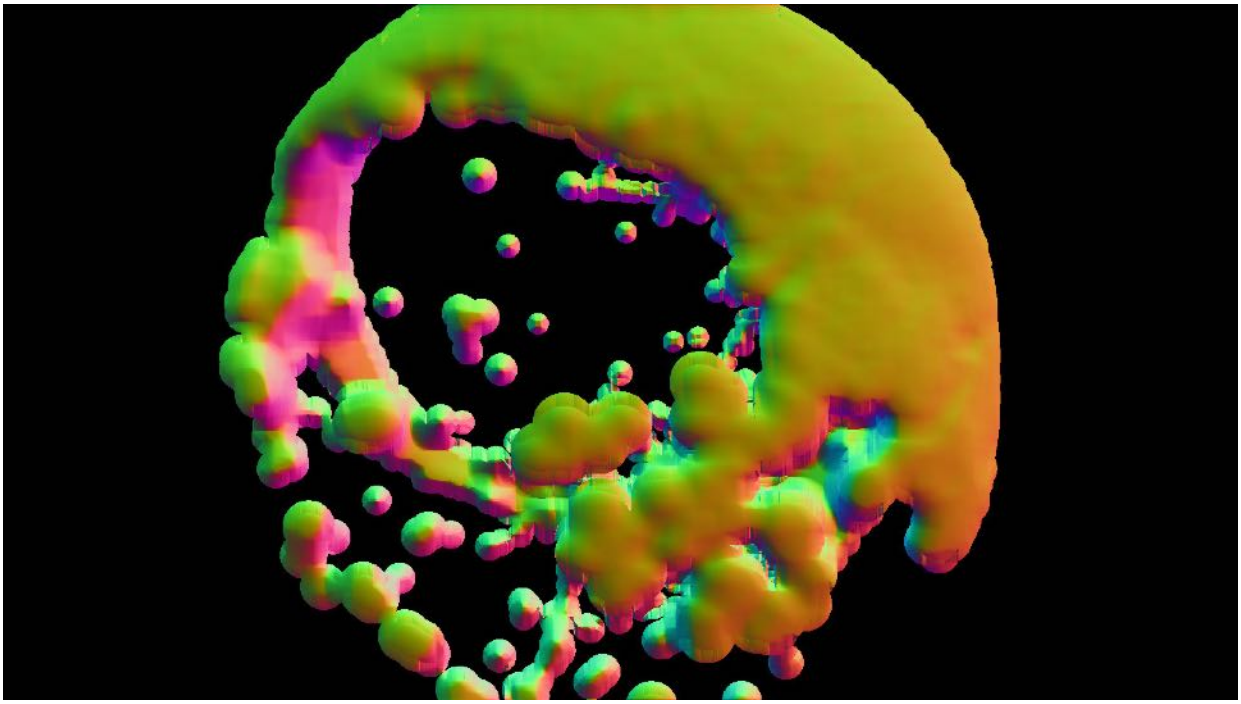

Figure 3.9: Normal image

3.7.7 Calculate surface shading

The depth image and normal image obtained by the calculation so far are written to **G-Buffer** . **By** writing just before the **rendering pass** where the **G-Buffer** is generated, the geometry based on the calculation result is generated, and shading and lighting are applied.

Excerpt from the code

ScreenSpaceFluidRendering.cs

```
// ------------------------------------------------------------
-------
// 4. Write the calculation result to G-Buffer and draw the
```

```
particles
// ------------------------------------------------------------
-------
buf.SetGlobalTexture ("_NormalBuffer", normalBufferId); // Set
the normal buffer
buf.SetGlobalTexture ("_DepthBuffer", depthBufferId); // Set the
depth buffer

// set properties
_renderGBufferMaterial.SetColor("_Diffuse", _diffuse );
_renderGBufferMaterial.SetColor("_Specular",
    new Vector4(_specular.r, _specular.g, _specular.b, 1.0f -
_roughness));
_renderGBufferMaterial.SetColor("_Emission", _emission);

// Set G-Buffer to render target
buf.SetRenderTarget
(
  new RenderTargetIdentifier[4]
  {
    BuiltinRenderTextureType.GBuffer0, // Diffuse
    BuiltinRenderTextureType.GBuffer1, // Specular + Roughness
    BuiltinRenderTextureType.GBuffer2, // World Normal
    BuiltinRenderTextureType.GBuffer3  // Emission
  },
  BuiltinRenderTextureType.CameraTarget  // Depth
);
// Write to G-Buffer
buf.DrawMesh(quad, Matrix4x4.identity, _renderGBufferMaterial);
```

RenderGBuffer.shader

```
// GBuffer structure
struct gbufferOut
{
  half4 diffuse: SV_Target0; // Diffuse reflection
  half4 specular: SV_Target1; // specular reflection
  half4 normal: SV_Target2; // normal
  half4 emission: SV_Target3; // emission light
  float depth: SV_Depth; // depth
};

sampler2D _DepthBuffer; // depth
sampler2D _NormalBuffer;// 法線

fixed4 _Diffuse; // Diffuse color
fixed4 _Specular; // Color of specular light
```

```
float4 _Emission; // Color of synchrotron radiation

void frag(v2f i, out gbufferOut o)
{
  float2 uv = i.screenPos.xy * 0.5 + 0.5;

  float  d = tex2D(_DepthBuffer,  uv).r;
  float3 n = tex2D(_NormalBuffer, uv).xyz;

#if UNITY_REVERSED_Z
  if (Linear01Depth(d) > 1.0 - 1e-3)
    discard;
#else
  if (Linear01Depth(d) < 1e-3)
    discard;
#endif

  o.diffuse  = _Diffuse;
  o.specular = _Specular;
  o.normal   = float4(n.xyz , 1.0);

  o.emission = _Emission;
#ifndef UNITY_HDR_ON
  o.emission = exp2(-o.emission);
#endif

  o.depth    = d;
}
```

In the SetRenderTarget method, specify the **G-Buffer** to be **written** to the render target . First in the color buffer to be an argument to the target **BuiltinRenderTextureType** enumeration **GBuffer0** , **GBuffer1** , **GBuffer2** , **GBuffer3** specify a **RenderTargetIdentifier** sequences, also in the depth buffer to be the second argument of the target **CameraTarget** by specifying a **default G- A set of Buffers** and **depth information** can be specified as the **render target** .

In order to perform calculations on the screen space with a shader that specifies multiple **render targets** in the **command buffer** , this is **achieved** here by using the **DrawMesh** method to draw a rectangular mesh that covers the screen.

3.7.8 Temporary release of RenderTexture

Don't forget to release the temporary **RenderTexture** generated by the **GetTemporaryRT** method with the **ReleaseTemporaryRT** method. If this is not done, memory will be allocated every frame and memory overflow will occur.

3.7.9 Rendering result

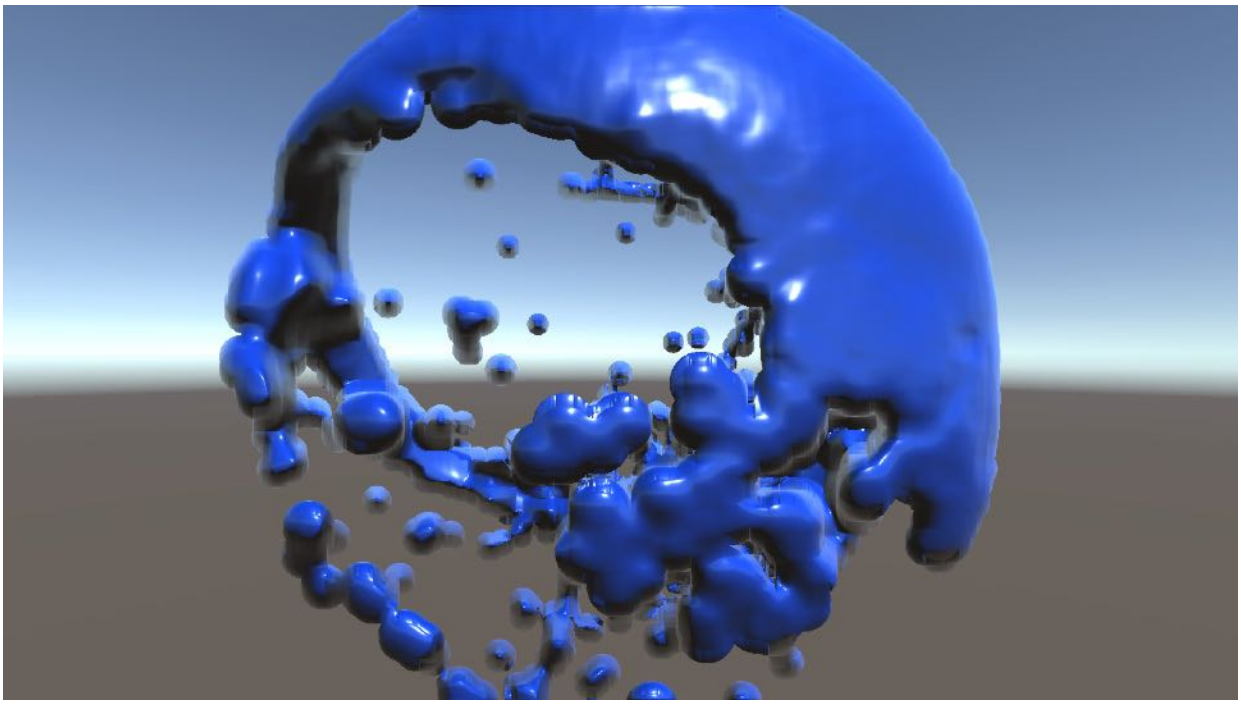

Figure 3.10: Rendering result

3.8 Summary

This chapter focused on manipulating geometry with **Deferred Shading** . In this sample, as a translucent object, elements such as light absorption according to the thickness, background transmission considering internal refraction, and condensing phenomenon are not implemented. If you want to express it as a liquid, you should implement these elements as well. **In** order to utilize **Deferred Rendering** , it is necessary to understand the calculation in 3DCG rendering such as coordinate system, coordinate transformation, shading, lighting, etc. that you do not usually need to be aware of when using Unity. In the range of observation, there are not many sample codes

and references for learning using **Deferred Rendering** in Unity, and I am still not fully understood, but I feel that it is a technology that can expand the range of CG expression. I am. We hope that this chapter will help those who have the purpose of expressing with CG that cannot be realized by conventional **Forward Rendering** .

See 3.9

- GDC Screen Space Fluid Rendering for Games, Simon Green, NVIDIA

http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf

- Why real-time rendering, Satoshi Kodaira

https://www.slideshare.net/SatoshiKodaira/ss-69311865

第4章　GPU-Based Cellular Growth Simulation

4.1　Introduction

Based on the algorithm of "Cell Division and Growth Algorithm 1" * 3 , which is introduced in the tutorial of iGeo * 2 , a library for procedural modeling in the field of construction by Processing * 1 , the GPU is used for cells. Develop a program that expresses division and growth.

The sample in this chapter is "Cellular Growth" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
.

In this chapter, through the cell division and growth program on the GPU

- How to dynamically control the number of objects on GPU using Append / ConsumeStructuredBuffer
- Representation of network structure on GPU
- Sequential processing by Atomic operation

I will explain about.

[*1] https://processing.org/

[*2] http://igeo.jp

[*3] http://igeo.jp/tutorial/55.html

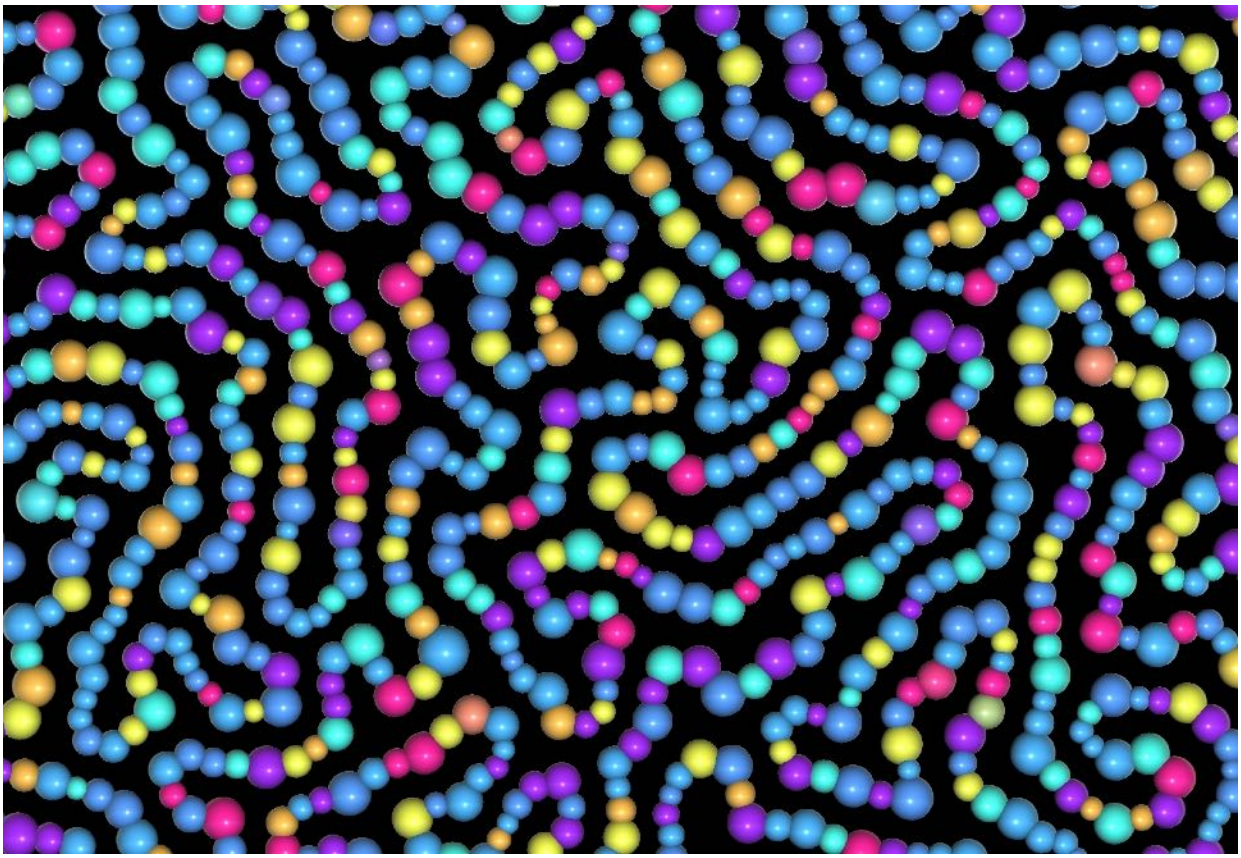

図 4.1: CellularGrowthSphere.scene

First, I will introduce a simple implementation of only Particles, and then explain how to introduce Edge and express the network structure that grows and becomes complicated.

4.2 Cell division and growth simulation

In the simulation program, we prepare two structures, Particle and Edge, to imitate the behavior of the cell.

One particle represents one cell and behaves as follows.

- Growth: grows over time
- Repulsion: Collides with other particles and repels each other
- Division: Divides under certain conditions and grows into two particles

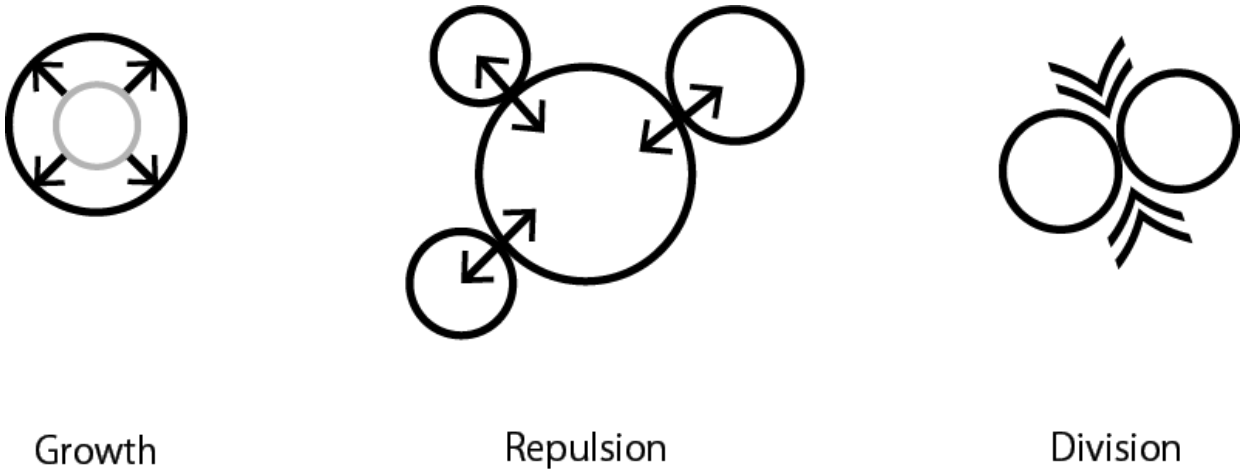

Figure 4.2: Cell behavior

Edge expresses how cells stick to each other. By connecting the divided particles with Edge and attracting them like a spring, the particles are attached to each other to express the network structure of cells.

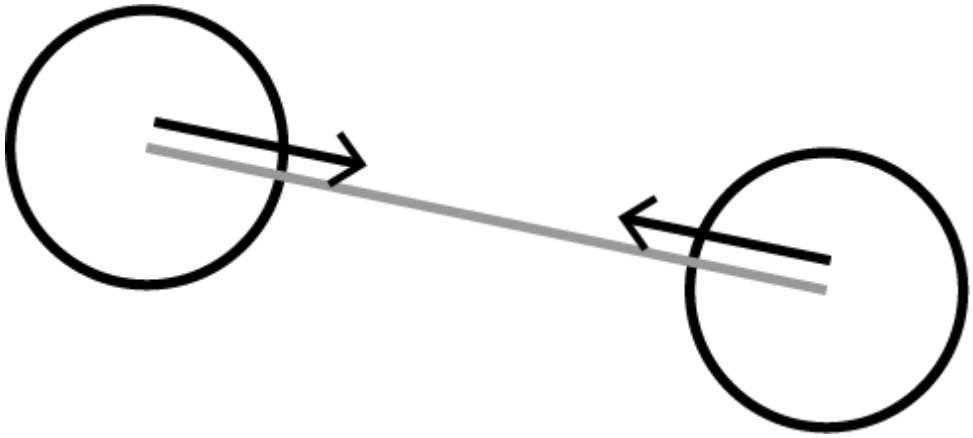

Figure 4.3: Edge sticks connected particles together

4.3 Implementation

In this section, we will explain by gradually implementing the necessary functions.

4.3.1 Particle Implementation (CellularGrowthParticleOnly.cs)

First, we will explain the behavior and implementation of particles through the sample CellularGrowthParticleOnly.cs that implements only the behavior of particles.

The structure of Particles is defined as follows.

Particle.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Particle_t {
    public Vector2 position; // position
    public Vector2 velocity; // velocity
    float radius; // size
    float threshold; // maximum size
     int links; // Number of connected Edges (used in the scene
below)
    uint alive; // activation flag
}
```

In this project, Particles are increased or decreased at any time, so the object pool is managed by Append / ConsumeStructuredBuffer so that the number of objects can be controlled on the GPU.

About Append / ConsumeStructuredBuffer

Append / ConsumeStructuredBuffer * 4 * 5 is a container for performing LIFO (Last In First Out) on the GPU made available from Direct3D11. AppendStructuredBuffer is responsible for adding data, and ConsumeStructuredBuffer is responsible for retrieving data.

By using this container, you can dynamically control the number on the GPU and express the increase or decrease of objects.

[*4] https://docs.microsoft.com/ja-jp/windows/desktop/direct3dhlsl/sm5-object-appendstructuredbuffer

[*5] https://docs.microsoft.com/ja-jp/windows/desktop/direct3dhlsl/sm5-object-consumestructuredbuffer

Buffer initialization

First, initialize the particle buffer and the object pool buffer.

CellularGrowthParticleOnly.cs

```
protected void Start () {
    // Particle initialization
                particleBuffer    =    new    PingPongBuffer(count,
typeof(Particle_t));

    // Initialize the object pool
    poolBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(int)),
        ComputeBufferType.Append
    );
    poolBuffer.SetCounterValue(0);
    countBuffer = new ComputeBuffer(
        4,
        Marshal.SizeOf(typeof(int)),
        ComputeBufferType.IndirectArguments
    );
    countBuffer.SetData(countArgs);

    // Object pool that manages divisible objects
    dividablePoolBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(int)),
        ComputeBufferType.Append
    );
    dividablePoolBuffer.SetCounterValue (0);

     // Particle and object pool initialization Kernel execution
(see below)
    InitParticlesKernel();

    ...
}
```

The PingPongBuffer class used as particleBuffer prepares two buffers, one for reading and the other for writing, and it is used in the scene of calculating the interaction of Particles described later.

poolBuffer and divideablePoolBuffer are Append / ConsumeStructuredBuffer, and ComputeBufferType.Append is specified in the argument ComputeBufferType at the time of initialization. Append /

ConsumeStructuredBuffer can handle variable length data, but as you can see from the initialization code, the upper limit of the number of data must be set when creating the buffer.

The poolBuffer created as an int type Append / ConsumeStructuredBuffer is

1. Store the index of particles that are inactive at initialization in poolBuffer (Push to Stack)
2. When adding a particle, take out the index from the poolBuffer (Pop from the Stack) and turn on the alive flag of the Particle in the particleBuffer associated with that index.

It functions as an object pool according to the flow. In other words, the int buffer of poolBuffer always points to the index of inactive Particles, and can be made to function as an object pool by fetching it as needed. (Fig. 4.4)

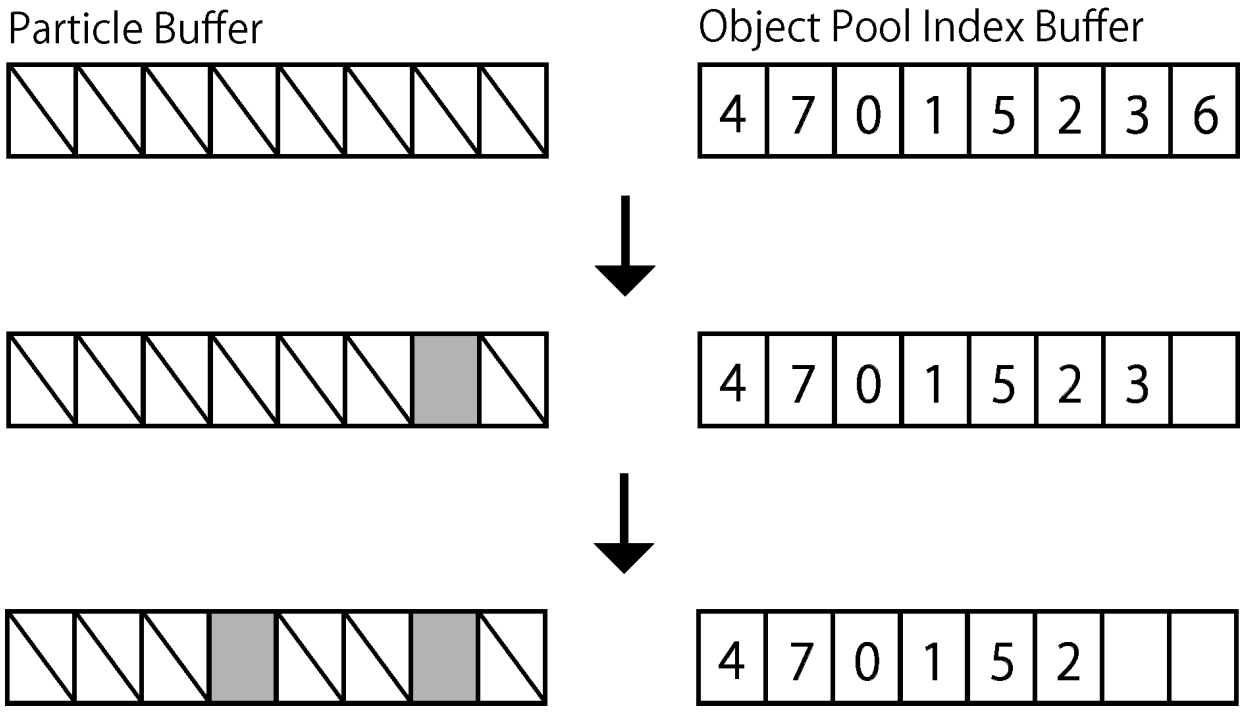

Figure 4.4: The array on the left represents particleBuffer and the right represents poolBuffer.In the initial state, all particles in particleBuffer are inactive, but when particles appear, the index of the inactive Particle is taken out from poolBuffer and the corresponding index is displayed. Activate the particles in the area

countBuffer is an int type buffer and is used to manage the number of object pools.

The InitParticlesKernel called at the end of Start runs the GPU kernel that initializes the Particles and object pool.

CellularGrowthParticleOnly.cs

```
protected void InitParticlesKernel()
{
    var kernel = compute.FindKernel("InitParticles");
                    compute.SetBuffer(kernel,     "_Particles",
particleBuffer.Read);

    // Specify the object pool as AppendStructuredBuffer
              compute.SetBuffer(kernel,   "_ParticlePoolAppend",
poolBuffer);

    Dispatch1D(kernel, count);
}
```

The following is the kernel to be initialized.

CellularGrowth.compute

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

  uint count, strides;
  _Particles.GetDimensions(count, strides);
  if (idx >= count)
    return;

  // Particle initialization
  Particle p = create();
  p.alive = false; // Inactivate all Particles
  _Particles[idx] = p;

  // Add particle index to object pool
  _ParticlePoolAppend.Append(idx);
}
```

By executing the above kernel, all the particles in the particleBuffer will be initialized and inactive, and the poolBuffer will store the indexes of all the particles in the inactive state.

Appearance of Particles

Now that we have initialized the particles, let's make them appear. In CellularGrowthParticleOnly.cs, particles are generated at the position where the mouse is clicked.

CellularGrowthParticleOnly.cs

```
protected void Update() {
    ...
    if(Input.GetMouseButton(0))
    {
        EmitParticlesKernel(GetMousePoint());
    }
    ...
}
```

When the mouse is clicked, it runs the EmitParticlesKernel to spawn particles.

CellularGrowthParticleOnly.cs

```
protected void EmitParticlesKernel(Vector2 point, int emitCount
= 32)
{
    // Compare the number of object pools with emitCount,
    // Prevent _ParticlePoolConsume.Consume () from running when
the object pool is empty
    emitCount = Mathf.Max(
        0,
        Mathf.Min (emitCount, CopyPoolSize (poolBuffer))
    );
    if (emitCount <= 0) return;

    var kernel = compute.FindKernel("EmitParticles");
                    compute.SetBuffer(kernel,     "_Particles",
particleBuffer.Read);

    // Specify the object pool as ConsumeStructuredBuffer
```

```
            compute.SetBuffer(kernel,   "_ParticlePoolConsume",
poolBuffer);

    compute.SetVector("_Point", point);
    compute.SetInt("_EmitCount", emitCount);

    Dispatch1D(kernel, emitCount);
}
```

As you can see from the fact that the poolBuffer specified in the _ParticlePoolAppend parameter in InitParticlesKernel is specified in the _ParticlePoolConsume parameter in EmitParticlesKernel, the same buffer is specified in Append / ConsumeStructuredBuffer.

Depending on the purpose of processing on the GPU, just changing the setting of whether to add a buffer (AppendStructuredBuffer) or to retrieve (ConsumeStructuredBuffer), the same buffer is sent to the GPU side from the CPU side. Become.

At the beginning of EmitParticlesKernel, we compare the size of the object pool obtained by emitCount and GetPoolSize, but this is to prevent index retrieval from the pool when the object pool is empty, if it is an empty object. Attempting to retrieve more indexes from the pool (running _ParticlePoolConsume.Consume inside the GPU kernel) results in unexpected behavior.

CellularGrowth.compute

```
THREAD
void EmitParticles(uint3 id : SV_DispatchThreadID)
{
  // Avoid adding more Particles than _EmitCount
  if (id.x >= (uint) _EmitCount)
    return;

  // Extract the index of the inactive Particle from the object
pool
  uint idx = _ParticlePoolConsume.Consume();

  Particle c = create();

  // Place the Particle at a position slightly offset from the
mouse position
```

```
    float2 offset = random_point_on_circle(id.xx + float2(0,
_Time));
  c.position = _Point.xy + offset;
  c.radius = nrand(id.xx + float2(_Time, 0));

  // Set the activated Particle to the inactive index location
  _Particles[idx] = c;
}
```

In Emit Particles, the index of the inactive particle is taken out from the object pool, and the activated particle is set at the position of the corresponding index in the particle Buffer.

By the above kernel processing, particles can be spawned while considering the number of object pools.

Particle behavior

Now that we have managed the appearance of particles, it's time to program the behavior of particles.

The cells of the simulator developed in this chapter behave as follows, as shown in Figure 4.2 .

- Growth: Particles grow gradually until they reach a certain size
- Repulsion: Forces are applied to repel each other when they come into contact with other particles.
- Division: Particles split under certain conditions

GrowthとRepulsion

Growth and Repulsion are executed every frame in Update.

CellularGrowthParticleOnly.cs

```
protected void Update() {
    ...
    UpdateParticlesKernel();
    ...
}
```

```
...
protected void UpdateParticlesKernel()
{
    var kernel = compute.FindKernel("UpdateParticles");

    // Set the read buffer
                  compute.SetBuffer(kernel,    "_ParticlesRead",
particleBuffer.Read);

    // Set a buffer for writing
                    compute.SetBuffer(kernel,     "_Particles",
particleBuffer.Write);

    compute.SetFloat ("_Drag", drag); // Speed attenuation
    compute.SetFloat ("_Limit", limit); // Speed limit
     compute.SetFloat ("_Repulsion", repulsion); // Coefficient
over repulsive distance
    compute.SetFloat("_Grow", grow); // growth rate

    Dispatch1D(kernel, count);

    // Swap read and write buffers (Ping Pong)
    particleBuffer.Swap();
}
```

The reason for setting the read and write buffers and swapping the buffers after processing will be described later.

Below is the Update Particles kernel.

CelluarGrowth.compute

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

  uint count, strides;
  _ParticlesRead.GetDimensions(count, strides);
  if (idx >= count)
    return;

  Particle p = _ParticlesRead[idx];

  // Process only activated Particles
  if (p.alive)
```

```
  {
    // Grow: Particle growth
    p.radius = min(p.threshold, p.radius + _DT * _Grow);

    // Repulsion: Collisions between Particles
    for (uint i = 0; i < count; i++)
    {
      Particle other = _ParticlesRead[i];
      if(i == idx || !other.alive) continue;

      // Calculate the distance between particles
      float2 dir = p.position - other.position;
      float l = length(dir);

       // The distance between the particles is greater than the
sum of their radii * _Repulsion
      // If they are close, they are in conflict
      float r = (p.radius + other.radius) * _Repulsion;
      if (l < r)
      {
        p.velocity += normalize(dir) * (r - l);
      }
    }

    float2 vel = p.velocity * _DT;
    float vl = length(vel);
     // check if velocity length over than zero to avoid NaN
position
    if (vl > 0)
    {
      p.position + = normalize (vel) * min (vl, _Limit);

      // Attenuate velocity according to _Drag parameter
      p.velocity =
        normalize(p.velocity) *
        min (
          length(p.velocity) * _Drag,
          _Limit
        );
    }
    else
    {
      p.velocity = float2(0, 0);
    }
  }

  _Particles[idx] = p;
}
```

The UpdateParticles kernel uses a read buffer (_ParticlesRead) and a write buffer (_Particles) to calculate collisions between particles.

If the same buffer is used for both reading and writing here, there is a possibility that another thread will use the particle information after being updated by another thread for particle position calculation due to GPU parallel processing. Will appear, and a problem (data race) will occur in which the calculation is inconsistent.

If one thread does not refer to the information updated by another thread, it is not necessary to prepare separate buffers for reading and writing, but if the thread refers to the buffer updated by another thread. Like the UpdateParticles kernel, it needs to have separate read and write buffers, which alternate with each update. (It is called Ping Pong buffer because it alternates buffers after each process.)

Division

Particle splitting is performed by coroutines at regular intervals.

Particle splitting process

1. Get the index of divisible particles and store it in the dividablePoolBuffer
2. Take out the particles you want to split from the dividablePoolBuffer and split them.

It is done in the flow.

CellularGrowthParticleOnly.cs

```
protected void Start() {
    ...
    StartCoroutine(IDivider());
}

...

protected IEnumerator IDivider()
{
```

```
    yield return 0;
    while(true)
    {
        yield return new WaitForSeconds(divideInterval);
        Divide();
    }
}

protected void Divide() {
    GetDividableParticlesKernel();
    DivideParticlesKernel(maxDivideCount);
}

...

// Store divisible particle candidates in dividablePoolBuffer
protected void GetDividableParticlesKernel()
{
    // Reset dividablePoolBuffer
    dividablePoolBuffer.SetCounterValue (0);

    var kernel = compute.FindKernel("GetDividableParticles");
                    compute.SetBuffer(kernel,     "_Particles",
particleBuffer.Read);
             compute.SetBuffer(kernel,   "_DividablePoolAppend",
dividablePoolBuffer);

    Dispatch1D(kernel, count);
}

protected void DivideParticlesKernel(int maxDivideCount = 16)
{
    // With the number you want to split (maxDivideCount)
     // Compare the number of particles that can be split (the
size of the dividable PoolBuffer)
    maxDivideCount = Mathf.Min (
        CopyPoolSize (dividablePoolBuffer),
        maxDivideCount
    );

    // With the number you want to split (maxDivideCount)
     // Compare the number of particles remaining in the object
pool (poolBuffer size)
       maxDivideCount  =  Mathf.Min  (CopyPoolSize  (poolBuffer),
maxDivideCount);

    if (maxDivideCount <= 0) return;
```

```
    var kernel = compute.FindKernel("DivideParticles");
                    compute.SetBuffer(kernel,     "_Particles",
particleBuffer.Read);
             compute.SetBuffer(kernel,   "_ParticlePoolConsume",
poolBuffer);
            compute.SetBuffer(kernel,   "_DividablePoolConsume",
dividablePoolBuffer);
    compute.SetInt("_DivideCount", maxDivideCount);

    Dispatch1D(kernel, count);
}
```

The GetDividableParticles kernel adds divisible particles (active particles) to the dividablePoolBuffer, and uses that buffer to determine the number of times to execute the DivideParticles kernel that actually performs the split processing.

How to find the number of splits is as shown at the beginning of the DivideParticlesKernel function.

- maxDivideCount
- The number of particles that can be divided by the dividablePoolBuffer,
- Number of inactive particles remaining in the object pool of poolBuffer

Compare with. Comparing these numbers prevents the split process from running beyond the limit of the number of splits that can be split.

The following is the contents of the kernel.

CellularGrowth.compute

```
// Function that determines the candidate particles that can be
split
// You can adjust the split pattern by changing the conditions
here
bool dividable_particle(Particle p, uint idx)
{
  // Split according to growth rate
  float rate = (p.radius / p.threshold);
  return rate >= 0.95;

  // Randomly split
  // return nrand(float2(idx, _Time)) < 0.1;
}
```

```
// Function that splits particles
uint divide_particle(uint idx, float2 offset)
{
  Particle parent = _Particles[idx];
  Particle child = create();

  // Set the size in half
  float rh = parent.radius * 0.5;
  rh = max(rh, 0.1);
  parent.radius = child.radius = rh;

  // Shift the position of parent and child
  float2 center = parent.position;
  parent.position = center - offset;
  child.position = center + offset;

  // Randomly set the maximum size of the child
  float x = nrand(float2(_Time, idx));
  child.threshold = rh * lerp(1.25, 2.0, x);

  // Get the child index from the object pool and set the child
particle in the buffer
  uint cidx = _ParticlePoolConsume.Consume();
  _Particles[cidx] = child;

  // Update parent particle
  _Particles[idx] = parent;

  return cidx;
}

uint divide_particle(uint idx)
{
  Particle parent = _Particles[idx];

  // Randomly shift the position
  float2 offset =
    random_point_on_circle(float2(idx, _Time)) *
    parent.radius * 0.25;

  return divide_particle(idx, offset);
}

...

THREAD
void GetDividableParticles(uint3 id : SV_DispatchThreadID)
```

```
{
  uint idx = id.x;
  uint count, strides;
  _Particles.GetDimensions(count, strides);
  if (idx >= count)
    return;

  Particle p = _Particles[idx];
  if (p.alive && dividable_particle(p, idx))
  {
    _DividablePoolAppend.Append(idx);
  }
}

THREAD
void DivideParticles(uint3 id : SV_DispatchThreadID)
{
  if (id.x >= _DivideCount)
    return;

  uint idx = _DividablePoolConsume.Consume();
  divide_particle(idx);
}
```

The results of cell division achieved by these processes are as follows.

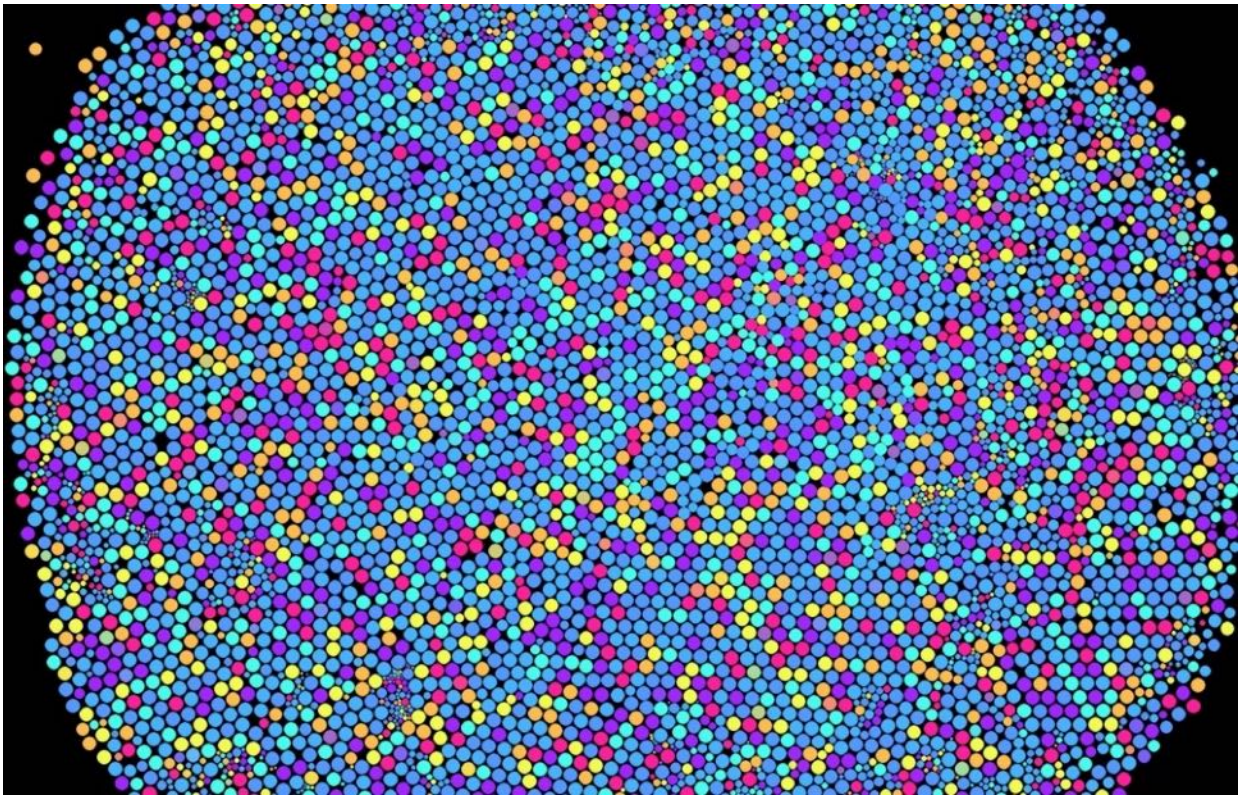

図 4.5: CellularGrowthParticleOnly.scene

4.3.2 Representation of network structure (CellularGrowth.cs)

In order to realize how cells stick to each other, we will introduce Edge that connects particles and express cells in a network structure.

From here, we will proceed through the implementation of CellularGrowth.cs.

Edges are added when the particles split, connecting the split particles together.

The structure of Edge is defined as follows.

Edge.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Edge_t
```

```
{
    public int a, b; // Index of two Particles connected by Edge
     public Vector2 force; // The force to attach two Particles
together
    uint alive; // activation flag
}
```

Edge also increases or decreases like Particle, so manage it with Append / ConsumeStructuredBuffer.

Division

The network structure is divided according to the following flow.

1. Get divisible Edge candidates and store in dividablePoolBuffer
2. If the splittable Edge is empty, split the Particle with 0 connected Edges (Particle with 0 links) and connect the two Particles with Edge.
3. If there is a splittable Edge, remove the Edge from the divideablePoolBuffer and split it

It is the Particle that actually splits, but the term "splittable Edge" here is convenient when processing the Edge that is connected to the Particle that splits from the split pattern that will be introduced later. For good reason, the network structure is split in Edge units.

The above-mentioned flow of division allows one particle to repeat division and generate a large network structure.

Edge splitting is performed by coroutines at regular intervals, similar to CellularGrowthParticleOnly.cs in the previous section.

CellularGrowth.cs

```
protected IEnumerator IDivider()
{
    yield return 0;
    while(true)
    {
        yield return new WaitForSeconds(divideInterval);
        Divide();
    }
```

```
}

protected void Divide()
{
     // 1. Get divisible Edge candidates and store them in the
divideablePoolBuffer
    GetDividableEdgesKernel();

                int    dividableEdgesCount    =    CopyPoolSize
(dividablePoolBuffer);
    if(dividableEdgesCount == 0)
    {
        // 2. If the splittable Edge is empty,
          // Split a Particle with 0 connected Edges (Particle
with 0 links) and split it.
        // Connect two Particles with Edge
        DivideUnconnectedParticles();
    } else
    {
         // 3. If there is a splittable Edge, take the Edge from
the divideablePoolBuffer and split it.
            // Execute Edge split according to split pattern
(described later)
        switch(pattern)
        {
            case DividePattern.Closed:
                      // Patterns that generate closed network
structures
                DivideEdgesClosedKernel(
                    dividableEdgesCount,
                    maxDivideCount
                );
                break;
            case DividePattern.Branch:
                // Branching pattern
                DivideEdgesBranchKernel(
                    dividableEdgesCount,
                    maxDivideCount
                );
                break;
        }
    }
}

...

protected void GetDividableEdgesKernel()
{
```

```
    // Reset the buffer that stores the splittable Edge
    dividablePoolBuffer.SetCounterValue (0);

    var kernel = compute.FindKernel("GetDividableEdges");
    compute.SetBuffer(
        kernel, "_Particles",
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);
             compute.SetBuffer(kernel,   "_DividablePoolAppend",
dividablePoolBuffer);

    // Maximum number of particle connections
    compute.SetInt("_MaxLink", maxLink);

    Dispatch1D(kernel, count);
}

...

protected void DivideUnconnectedParticles()
{
                                        var         kernel         =
compute.FindKernel("DivideUnconnectedParticles");
    compute.SetBuffer(
        kernel, "_Particles",
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(
        kernel, "_ParticlePoolConsume",
        particlePool.PoolBuffer
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);
                compute.SetBuffer(kernel,    "_EdgePoolConsume",
edgePool.PoolBuffer);

    Dispatch1D(kernel, count);
}
```

The kernels (GetDividableEdges) for getting divisible edges are:

CellularGrowth.compute

```
// Determine if it can be split
bool dividable_edge(Edge e, uint idx)
{
  Particle pa = _Particles[e.a];
```

```
  Particle pb = _Particles[e.b];

   // The number of particle connections does not exceed the
maximum number of connections (_MaxLink)
   // Allow splitting if the splitting conditions defined in
dividable_particle are met
  return
    !(pa.links >= _MaxLink && pb.links >= _MaxLink) &&
        (dividable_particle(pa,  e.a)  &&  dividable_particle(pb,
e.b));
}

...

// Get a splittable Edge
THREAD
void GetDividableEdges(uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, strides;
  _Edges.GetDimensions(count, strides);
  if (idx >= count)
    return;

  Edge e = _Edges[idx];
  if (e.alive && dividable_edge(e, idx))
  {
    _DividablePoolAppend.Append(idx);
  }
}
```

If there is no splittable Edge, run the following kernel (Divide Disconnected Particles) that splits the following connected Edgeless Particles.

CellularGrowth.compute

```
// A function that creates an Edge that connects Particles with
index a and Particles b
void connect(int a, int b)
{
  // Fetch the inactive Edge index from the Edge object pool
  uint eidx = _EdgePoolConsume.Consume();

  // Using Atomic operation (described later)
  // Increment the number of connections for each particle
  InterlockedAdd(_Particles[a].links, 1);
  InterlockedAdd(_Particles[b].links, 1);
```

```
  Edge e;
  ea = a;
  e.b = b;
  e.force = float2(0, 0);
  e.alive = true;
  _Edges[eidx] = e;
}

...

// Split a Particle that does not have a connected Edge
THREAD
void DivideUnconnectedParticles(uint3 id : SV_DispatchThreadID)
{
  uint count, stride;
  _Particles.GetDimensions(count, stride);
  if (id.x >= count)
    return;

  uint idx = id.x;
  Particle parent = _Particles[idx];
  if (!parent.alive || parent.links > 0)
    return;

  // Generate a split child Particle from a parent Particle
  uint cidx = divide_particle(idx);

  // Connect parent and child particles with Edge
  connect(idx, cidx);
}
```

The connect function, which creates an Edge that connects split particles, uses a technique called Atomic operation to increment the number of particle connections.

Atomic operation (about InterlockedAdd function)

When a thread performs a series of processes of reading, modifying, and writing data in global memory or shared memory, the value changes due to writing from other threads to the

to writing from other threads to the memory area during the process. You may want to prevent it from happening. (A phenomenon called data race (data race), in which the result changes depending on the order in which threads access memory, which is peculiar to parallel processing)

Atomic arithmetic guarantees this, preventing interference from other threads during resource arithmetic operations (four arithmetic operations and comparisons), and safely realizing sequential processing on the GPU.

In HLSL, the functions * 6 that perform these operations have a prefix called Interlocked, and the examples in this chapter use InterlockedAdd.

The InterlockedAdd function is the process of adding the integer specified in the second argument to the resource specified in the first argument, and increments the number of connections by adding 1 to _Particles [index] .links.

This allows you to manage the number of connections consistently between threads, and you can increase or decrease the number of connections consistently.

[*6] https://docs.microsoft.com/ja-jp/windows/desktop/direct3d11/direct3d-11-advanced-stages-cs-atomic-

functions

If there is a splittable Edge, remove the Edge from the divideablePoolBuffer and split it. As you can see from the enum parameter called DividePattern, various patterns can be applied to the division.

Here, we introduce a split pattern (DividePattern.Closed) that creates a closed network structure.

Closed network structure (DividePattern.Closed)

The pattern that creates a closed network structure splits as shown in the figure below.

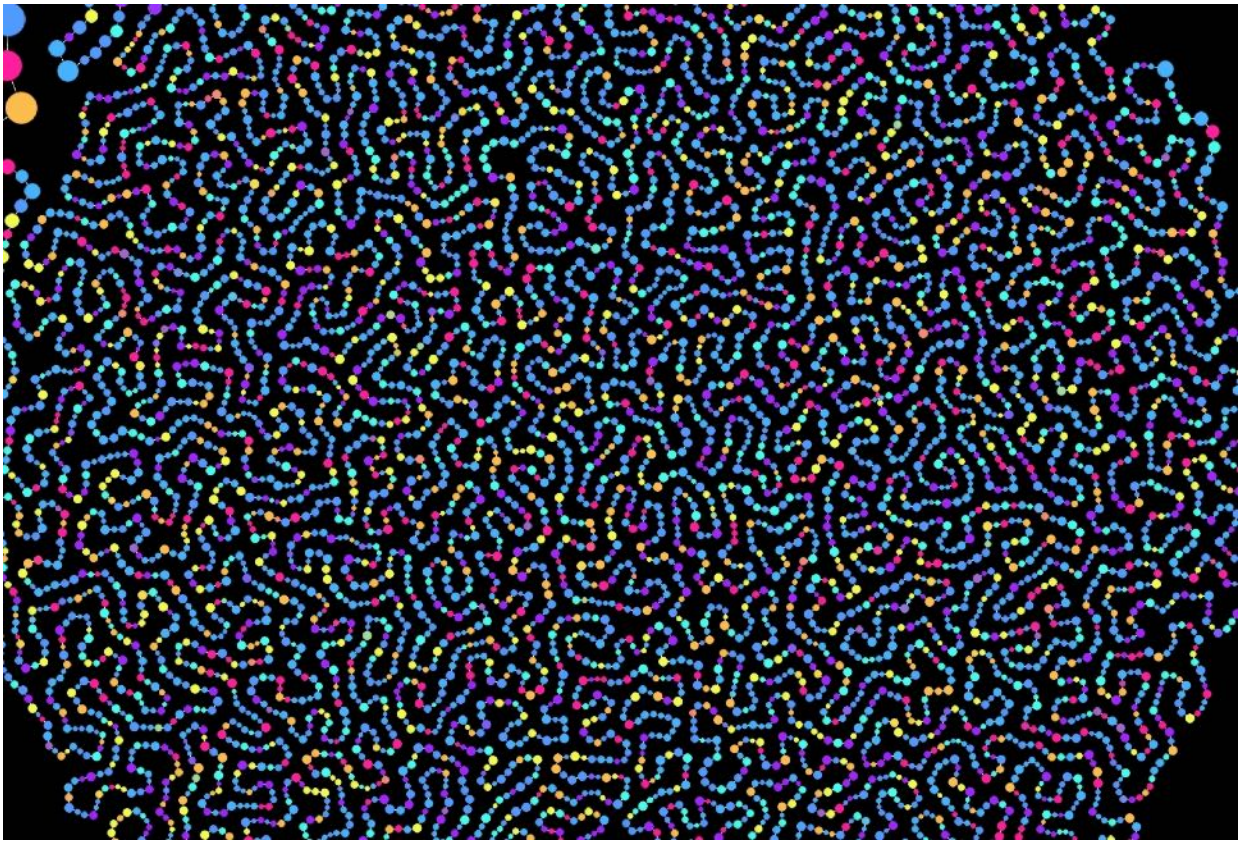

Figure 4.6: Pattern that creates a closed network structure (DividePattern.Closed)

CellularGrowth.cs

```
protected void DivideEdgesClosedKernel(
    int dividableEdgesCount,
    int maxDivideCount = 16
)
{
    // Pattern that splits into a closed network structure
    var kernel = compute.FindKernel("DivideEdgesClosed");
                DivideEdgesKernel(kernel,    dividableEdgesCount,
maxDivideCount);
}

// Common processing in split patterns
protected void DivideEdgesKernel(
    int kernel,
    int dividableEdgesCount,
    int maxDivideCount
)
{
    // Prevent Consume from being called when the object pool is
empty
    // Compare maxDivideCount with the size of each object pool
              maxDivideCount   =   Mathf.Min(dividableEdgesCount,
maxDivideCount);
          maxDivideCount  =  Mathf.Min(particlePool.CopyPoolSize(),
maxDivideCount);
            maxDivideCount   =   Mathf.Min(edgePool.CopyPoolSize(),
maxDivideCount);
    if (maxDivideCount <= 0) return;

    compute.SetBuffer(
        kernel, "_Particles",
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(
        kernel, "_ParticlePoolConsume",
        particlePool.PoolBuffer
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);
                compute.SetBuffer(kernel,    "_EdgePoolConsume",
edgePool.PoolBuffer);

              compute.SetBuffer(kernel,   "_DividablePoolConsume",
dividablePoolBuffer);
    compute.SetInt("_DivideCount", maxDivideCount);

    Dispatch1D(kernel, maxDivideCount);
}
```

The function divide_edge_closed used in the GPU kernel (DivideEdgesClosed) that generates a closed network structure changes the processing according to the number of Edges that the Particle has.

If the number of connections of one of the particles is 1, connect them with Edge so as to draw a triangle with 3 particles added to the split particles. (Fig. 4.7)

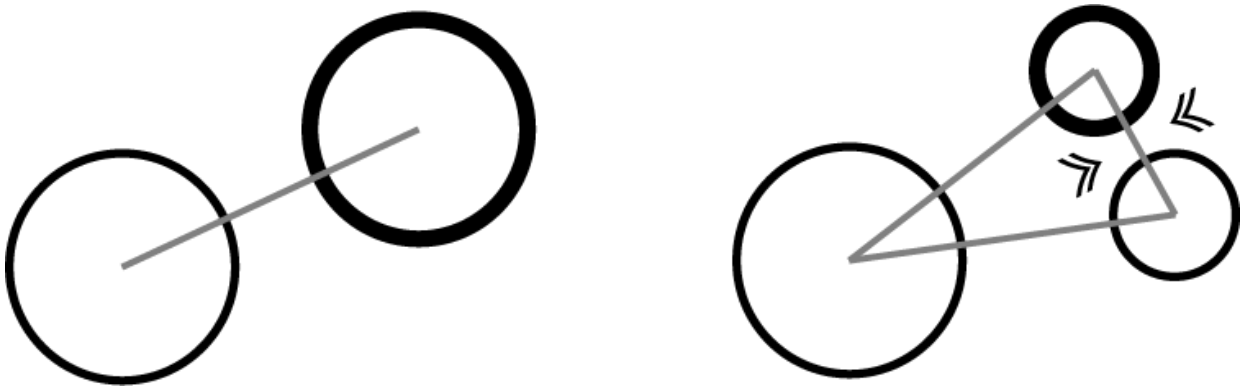

Figure 4.7: Two particles and split particles form a closed network in a triangular shape.

In other cases, the Edge is connected so that the split particle is inserted between the two existing particles, and the Edge that was connected to the split source particle is converted to maintain a closed network. I will. (Fig. 4.8)

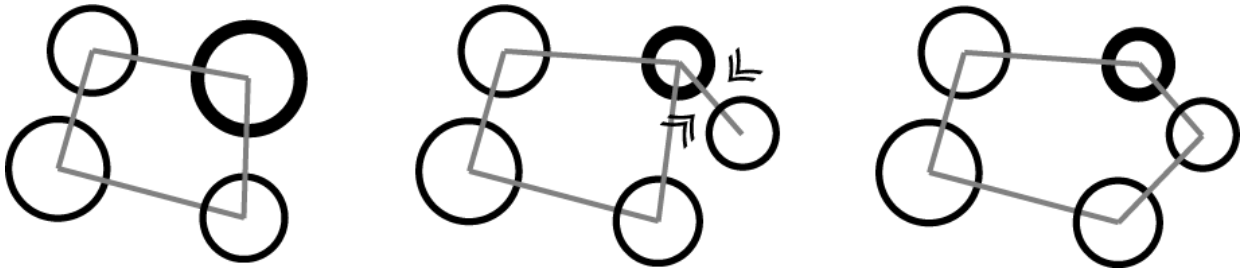

Figure 4.8: Insert a split particle between two existing particles and adjust the Edge connectivity to maintain a closed network.

By repeating this division process, a closed network structure grows.

CellularGrowth.compute

```
// A function that performs a split into a closed network
structure
void divide_edge_closed(uint idx)
{
  Edge e = _Edges[idx];

  Particle pa = _Particles[e.a];
  Particle pb = _Particles[e.b];

  if ((pa.links == 1) || (pb.links == 1))
  {
    // Divide into a triangle with 3 particles and connect them
with Edge
    uint cidx = divide_particle(e.a);
    connect(e.a, cidx);
    connect(cidx, e.b);
  }
  else
  {
    // Generate a Particle between two Particles and
    // Connect Edges so that they are connected
    float2 dir = pb.position - pa.position;
    float2 offset = normalize(dir) * pa.radius * 0.25;
    uint cidx = divide_particle(e.a, offset);

    // Connect the parent particle and the split child particle
    connect(e.a, cidx);

    // Edge that connected the original two Particles,
    // Convert to Edge connecting split child Particles
    InterlockedAdd(_Particles[e.a].links, -1);
    InterlockedAdd(_Particles[cidx].links, 1);
    ea = cidx;
  }

  _Edges[idx] = e;
}

...

// Pattern that splits into a closed network structure
THREAD
void DivideEdgesClosed(uint3 id : SV_DispatchThreadID)
{
  if (id.x >= _DivideCount)
    return;

  // Get the index of the splittable Edge
```

```
  uint idx = _DividablePoolConsume.Consume();
  divide_edge_closed(idx);
}
```

Inquiries about Edge

Many naturally occurring cells have the property of sticking to other cells. To mimic these properties, Edge pulls two connected particles together like a spring.

1. Calculate the force of the spring that attracts two particles for each edge
2. Add the power of Edge connected for each particle

Inquiries about springs are realized.

CellularGrowth.cs

```
protected void Update() {
    ...
    UpdateEdgesKernel();
    SpringEdgesKernel ();
    ...
}

...

protected void UpdateEdgesKernel()
{
     // Calculate the force that the spring attracts for each
Edge
    var kernel = compute.FindKernel("UpdateEdges");
    compute.SetBuffer(
        kernel, "_Particles",
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);
    compute.SetFloat("_Spring", spring);

    Dispatch1D(kernel, count);
}

protected void SpringEdgesKernel()
{
    // Apply the spring force of Edge for each particle
```

```
    var kernel = compute.FindKernel("SpringEdges");
    compute.SetBuffer(
        kernel, "_Particles",
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);

    Dispatch1D(kernel, count);
}
```

The following is the contents of the kernel.

CellularGrowth.compute

```
THREAD
void UpdateEdges(uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, strides;
  _Edges.GetDimensions(count, strides);
  if (idx >= count)
    return;

  Edge e = _Edges[idx];

  // Initialize the attractive force
  e.force = float2(0, 0);

  if (!e.alive)
  {
    _Edges[idx] = e;
    return;
  }

  Particle pa = _Particles[e.a];
  Particle pb = _Particles[e.b];
  if (!pa.alive || !pb.alive)
  {
    _Edges[idx] = e;
    return;
  }

  // Measure the distance between the two Particles,
  // Apply force to attract if you are too far away or too close
  float2 dir = pa.position - pb.position;
  float r = pa.radius + pb.radius;
  float len = length(dir);
```

```
  if (abs(len - r) > 0)
  {
    // Apply force to the proper distance (sum of radii of each
other)
    float l = ((len - r) / r);
    float2 f = normalize(dir) * l * _Spring;
    e.force = f;
  }

  _Edges[idx] = e;
}

THREAD
void SpringEdges(uint3 id : SV_DispatchThreadID)
{
  uint idx = id.x;
  uint count, strides;
  _Particles.GetDimensions(count, strides);
  if (idx >= count)
    return;

  Particle p = _Particles[idx];
  if (!p.alive || p.links <= 0)
    return;

  // The more connections you have, the weaker your attraction
  float dif = 1.0 / p.links;

  int iidx = (int)idx;

  _Edges.GetDimensions(count, strides);

  // Find the Particles that are connected to you from all Edges
  for (uint i = 0; i < count; i++)
  {
    Edge e = _Edges[i];
    if (!e.alive)
      continue;

    // Apply force when you find a connected Edge
    if (e.a == iidx)
    {
      p.velocity -= e.force * dif;
    }
    else if (e.b == iidx)
    {
      p.velocity += e.force * dif;
    }
```

```
  }

  _Particles[idx] = p;
}
```

By the above processing, it is possible to express how the cells composed of the network grow.

4.3.3 Variation of division pattern

Various division patterns can be designed by adjusting the judgment of the edge to be divided (dividable_edge function) and the division logic.

In the sample project CellularGrowth.cs, the split pattern can be switched by the enum parameter.

Dividing pattern (Divide Pattern.Branch)

In the branching pattern, the division is performed as shown in Figure 4.9 below .

Split child Particles connect only to the parent Particle. A branched network grows just by repeating this.

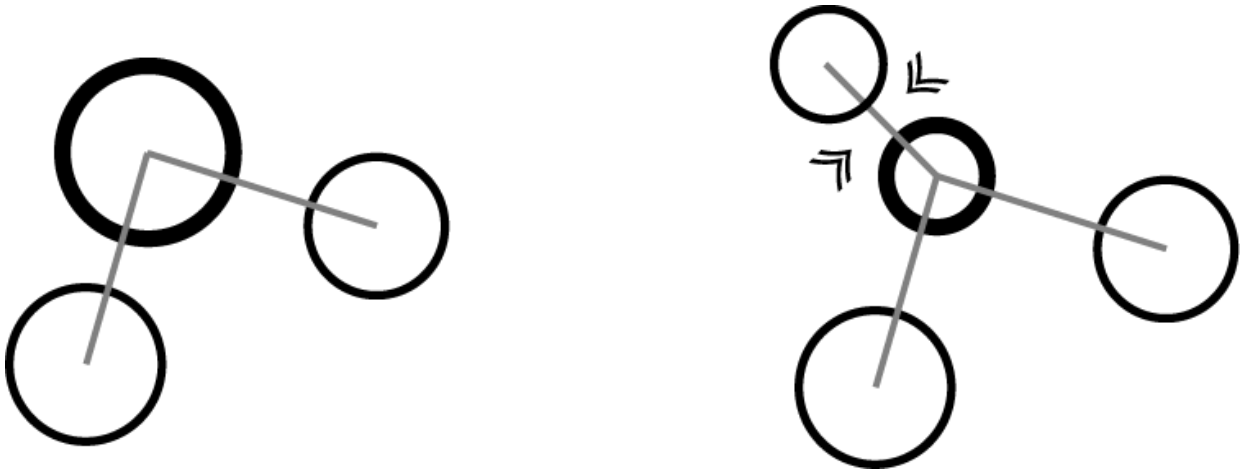

Figure 4.9: Branching split pattern

CellularGrowth.cs

```
protected void DivideEdgesBranchKernel(
    int dividableEdgesCount,
    int maxDivideCount = 16
)
{
    // Execute a branching split pattern
    var kernel = compute.FindKernel("DivideEdgesBranch");
                DivideEdgesKernel(kernel,    dividableEdgesCount,
maxDivideCount);
}
```

CellularGrowth.compute

```
// Function that performs branching
void divide_edge_branch(uint idx)
{
  Edge e = _Edges[idx];
  Particle pa = _Particles[e.a];
  Particle pb = _Particles[e.b];

    // Get the Particle index with the smaller number of
connections
  uint i = lerp(e.b, e.a, step(pa.links, pb.links));

  uint cidx = divide_particle(i);
  connect(i, cidx);
}

...

// Branching split pattern
THREAD
void DivideEdgesBranch(uint3 id : SV_DispatchThreadID)
{
  if (id.x >= _DivideCount)
    return;

  // Get the index of the splittable Edge
  uint idx = _DividablePoolConsume.Consume();
  divide_edge_branch(idx);
}
```

In a branching pattern, the logic that determines which edges are split has a significant visual impact. You can control the degree of branching by changing the value of the maximum number of connections (_MaxLink) of the Particles referenced in the dividable_edge function.

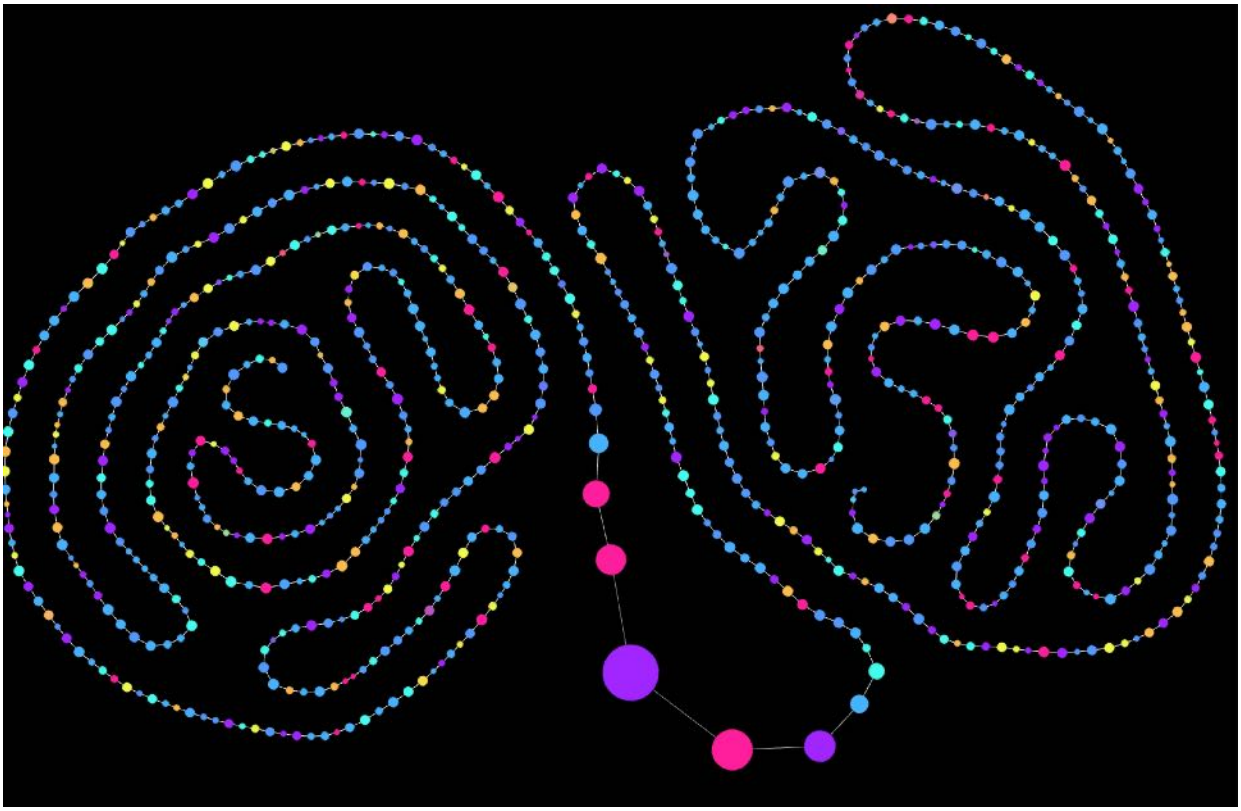

Figure 4.10: Pattern with _MaxLink set to 2 (DividePattern.Branch)

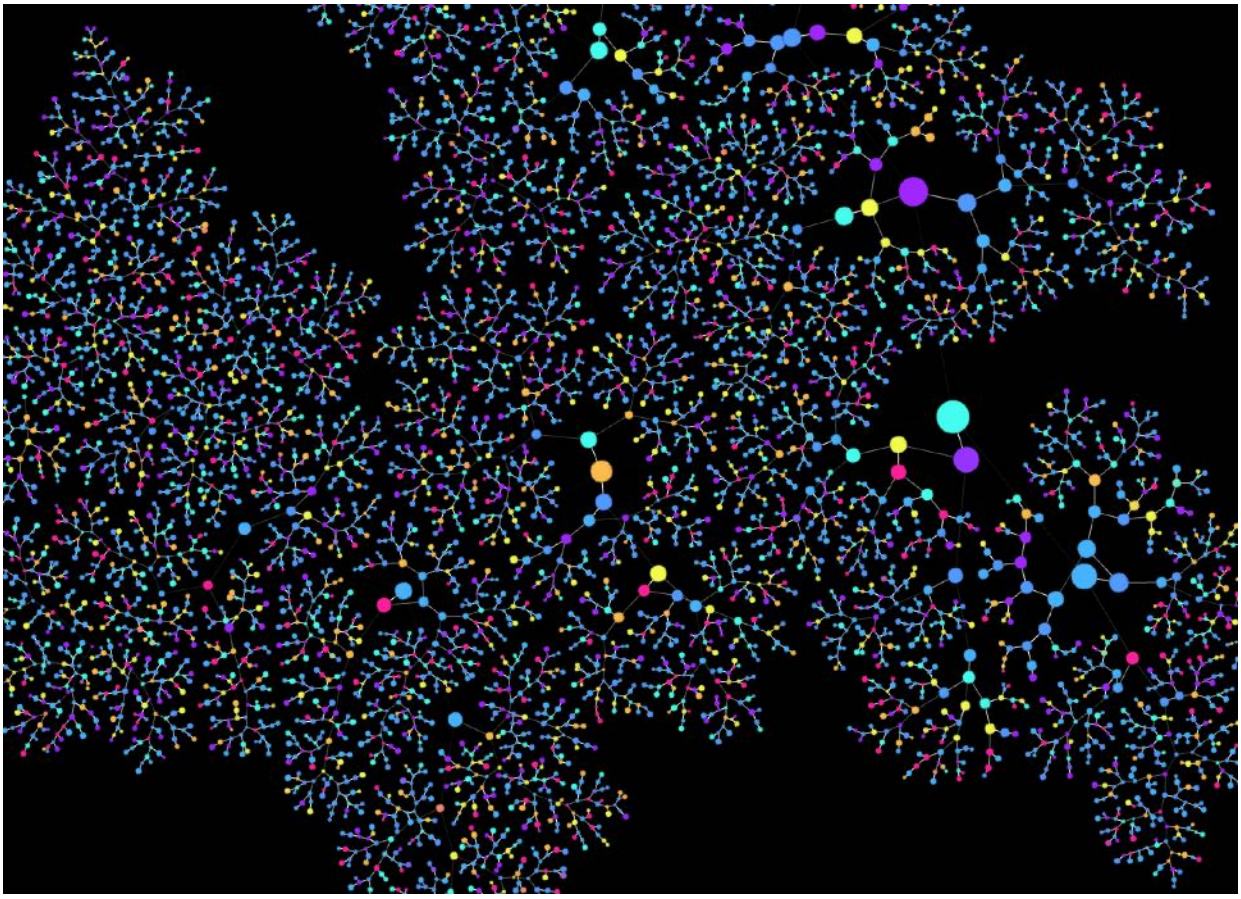

Figure 4.11: Pattern with _MaxLink set to 3 (Divide Pattern.Branch)

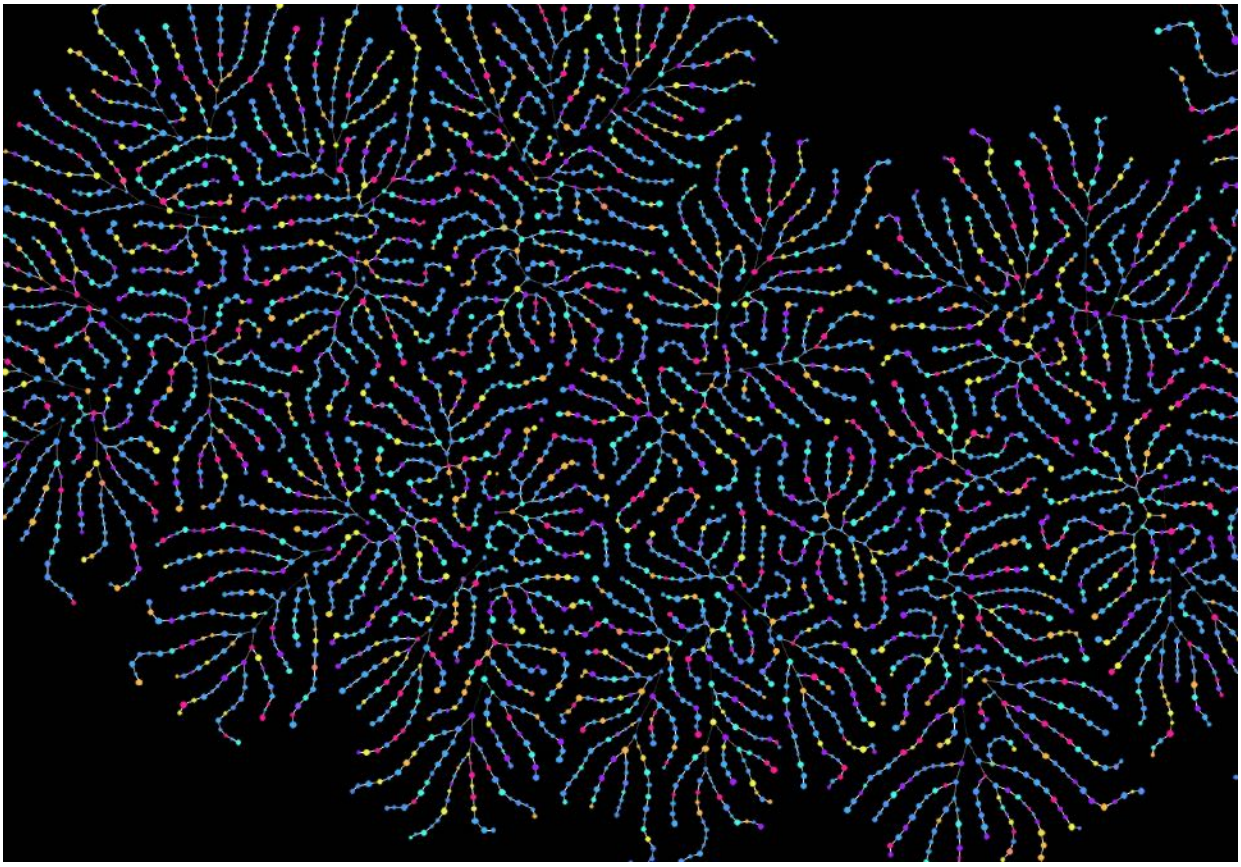

Figure 4.12: A pattern in which _MaxLink was set to 3 to grow to some extent and then set to 2 to continue growing (Divide Pattern.Branch).

4.4 Summary

In this chapter, we introduced a program that simulates cell division and growth on the GPU.

Other attempts to generate CG with such cells as motifs include the Morphogenetic Creations project by Andy Lomas * 7 and the Computational Biology project by JAKaandorp * 8 for academic purposes , especially the latter in biology. We are doing a more realistic simulation based on it.

Also, Max Cooper's music video * 10 by Maxime Causeret * 9 is an example of a wonderful video work using organic motifs such as cells. (Houdini is used for the simulation part in this video work)

This time, it was limited to those that split and grow in two dimensions, but as shown in the original iGeo tutorial * 12 , this program can also be extended in three dimensions.

In the extension to three dimensions, it is also possible to realize a mesh that grows organically with Gniguni by using a cell network that consists of three cells and grows and spreads. Samples of 3D extensions are available at https://github.com/mattatz/CellularGrowth, so if you are interested, please refer to them.

[*7] http://www.andylomas.com/

[*8] https://staff.fnwi.uva.nl/j.a.kaandorp/research.html

[*9] http://teresuac.fr/

[*10] https://vimeo.com/196269431

[*11] https://www.sidefx.com/

[*12] http://igeo.jp/tutorial/56.html

4.5 Reference

- http://igeo.jp/tutorial/55.html
- https://msdn.microsoft.com/ja-jp/library/ee422322(v=vs.85).aspx

Chapter 5 Reaction Diffusion

5.1 Introduction

In nature, there are various patterns such as horizontal stripes of tropical fish and wrinkles like a maze of coral. The genius mathematician Alan Turing expressed the occurrence of these patterns that exist in nature with mathematical formulas. The pattern generated by the mathematical formulas he derived is called the "Turing pattern". This equation is commonly referred to as the reaction-diffusion equation. Based on this reaction-diffusion equation, we will develop a program to create a picture like a pattern of a living thing using Compute Shader on Unity. At first, we will create a program that operates on a two-dimensional plane, but at the end, we will also introduce a program that operates on a three-dimensional space. For details on ComputeShader, refer to "Chapter 2 Introduction to ComputeShader" in UnityGraphicsProgramming vol.1.

The sample in this chapter is "Reaction Diffusion" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
.

5.2 Reaction Diffusionとは

As the name suggests, Reaction Diffusion is a local chemical reaction in which the concentrations of one or more substances distributed in a space change with each other, and space. It is a mathematical model of how diffusion, which spreads throughout the whole, changes due to the influence of two processes. This time, we will use the "Gray-Scott model" as the reaction-diffusion equation. The Gray-Scott model was published in a treatise by P. Gray and SKScott in 1983. Roughly speaking, when two virtual substances, U and V, are filled in the grid, they react with each other to increase or decrease or diffuse, and the concentration in the space changes over time. Various patterns will appear as you go.

Figure 5.1 is a schematic diagram of the “Reaction” of the Gray-Scott model.

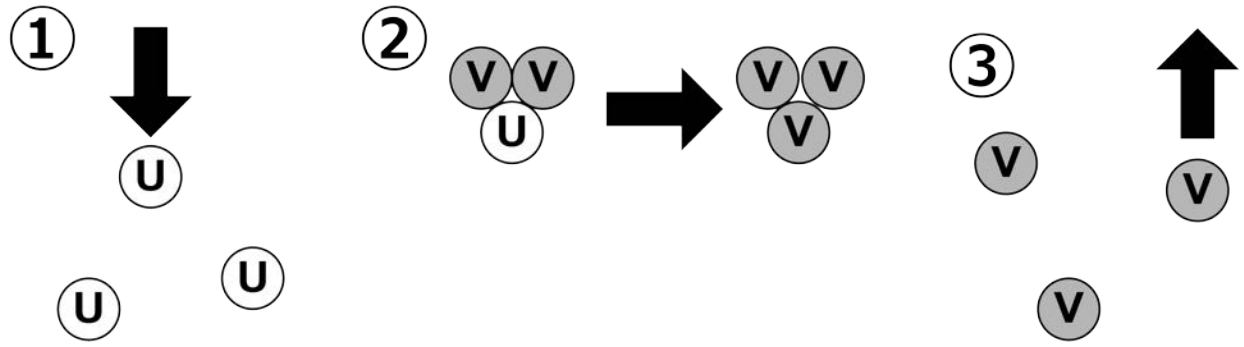

Figure 5.1: Schematic diagram of the "Reaction" of the Gray-Scott model

1. U is fed into the space at a constant rate
2. When there are two Vs, it reacts with U to create another V.
3. Since V will continue to increase as it is, V will be deleted (Kill) at a certain rate.

Also, as shown in Figure 5.2 , U and V spread to the adjacent grid at different speeds.

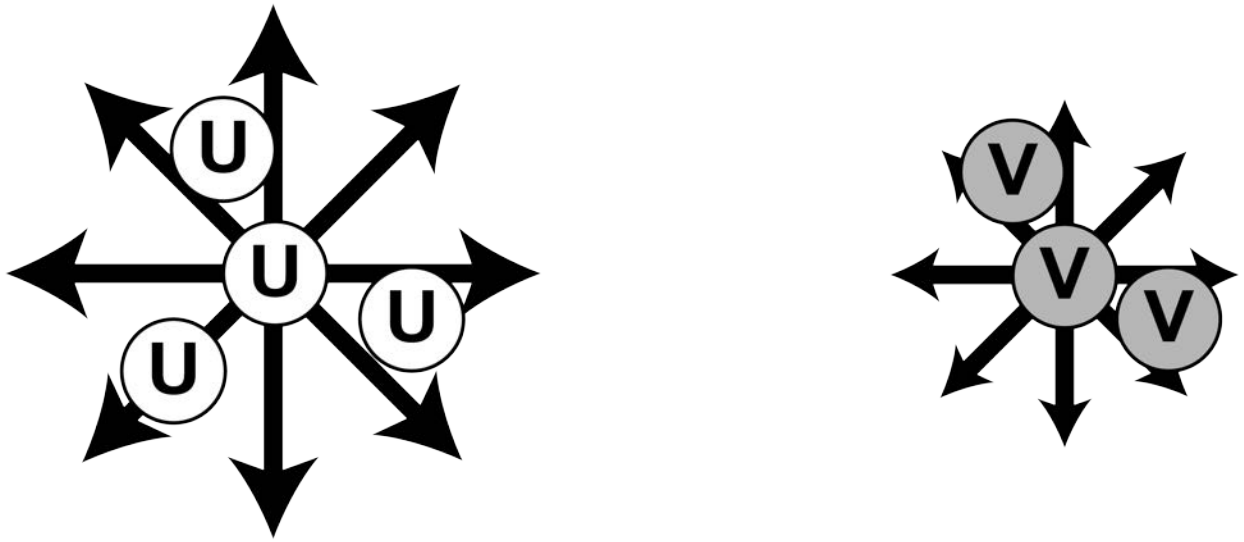

Figure 5.2: Schematic diagram of "Diffusion" in the Gray-Scott model

This difference in diffusion rate creates a difference in U and V concentrations, creating a pattern. The reaction and diffusion of these U and

V are expressed by the following equations.

```
\frac{\partial u}{\partial t} = Du \Delta u - uv^2 + f_{(1-u)}

\frac{\partial v}{\partial t} = Dv \Delta v + uv^2 - (f_{}+k)
```

In this formula, U is represented by u and V is represented by v . The formula is roughly divided into three.
The first Du \ Delta u and Dv \ Delta v are called the diffusion terms, and the first Du and Dv are constants of the diffusion rate of u and v . The latter half, \ Delta u and \ Delta v, are called Laplacian, and represent the process of diffusion in the direction of eliminating the concentration difference between the surroundings of U and V.
The second is called the reaction term, and uv ^ 2 indicates that U decreases and V increases by reacting with one U and two V.
The third + f_ {(1-u)} is called the inflow term and represents the amount to be replenished (Feed) when U decreases. The closer it is to 0, the more it is replenished, and the closer it is to 1, the more it is replenished. lose. -(f_ {} + k) is called the outflow term, which means that the increased V is reduced by a certain number (Kill).

To summarize a little more simply, U decreases and V increases in response to one U and two Vs. At this rate, U will continue to decrease and V will continue to increase, so U will be replenished by + f_ {(1-u)} and V will be forcibly decreased by- (f_ {} + k). It has become. Then, U and V are diffused to the surroundings by Du \ Delta u and Dv \ Delta v .

5.3 Implementation in Unity

Now that I have somehow understood the atmosphere of the equation, I will move on to the explanation of the implementation in Unity. The sample scene whose operation can be confirmed is **ReactionDiffusion2D_1** .

5.3.1 Definition of grid structure

Suppose you have U and V density values in a two-dimensional planar grid. This time, we will use Compute Shader to process in parallel, so we will manage the grid with Compute Buffer. First, define the structure in one grid.

ReactionDiffusion2D.cs

```
public struct RDData
{
    public float u; // U concentration
    public float v; // V concentration
}
```

5.3.2 Initialization

ReactionDiffusion2D.cs

```
/// <summary>
/// Initialization
/// </summary>
void Initialize()
{
    ...

    int wh = texWidth * texHeight; // Buffer size
    buffers = new ComputeBuffer [2]; // Array initialization of
ComputeBuffer for double buffering

    for (int i = 0; i < buffers.Length; i++)
    {
        // Grid initialization
                        buffers[i]   =   new   ComputeBuffer(wh,
Marshal.SizeOf(typeof(RDData)));
    }

    // Grid array for reset
    bufData = new RDData[wh];
    bufData2 = new RDData[wh];

    // Buffer initialization
    ResetBuffer();

    // Initialize the Seed addition buffer
    inputData = new Vector2[inputMax];
    inputIndex = 0;
    inputBuffer = new ComputeBuffer(
        inputMax, Marshal.SizeOf(typeof(Vector2))
      );
}
```

The **buffers** of ComputeBuffer for updating are two-dimensional arrays, but there are two for reading and writing. Because the Compute Shader is multi-threaded and processed in parallel. When processing that changes the calculation result by referring to the surrounding grid like this time, if it is one buffer, the calculation result will be referred to the value of the grid that has been calculated earlier depending on the order of the threads to be processed. It will change. In order to prevent it, it is divided into two parts, one for reading and the other for writing.

5.3.3 Update process

ReactionDiffusion2D.cs

```
// Update process
void UpdateBuffer()
{
    cs.SetInt("_TexWidth", texWidth);
    cs.SetInt("_TexHeight", texHeight);
    cs.SetFloat("_DU", du);
    cs.SetFloat("_DV", dv);

    cs.SetFloat("_Feed", feed);
    cs.SetFloat("_K", kill);

    cs.SetBuffer(kernelUpdate, "_BufferRead", buffers[0]);
    cs.SetBuffer(kernelUpdate, "_BufferWrite", buffers[1]);
    cs.Dispatch(kernelUpdate,
      Mathf.CeilToInt((float)texWidth / THREAD_NUM_X),
      Mathf.CeilToInt((float)texHeight / THREAD_NUM_X),
      1);

    SwapBuffer();
}
```

In the source on the C # side, the parameters that were also in the above equation are passed to the Compute Shader for update processing. Next, the update process in Compute Shader is explained.

ReactionDiffusion2D.compute

```
// Update process
[numthreads(THREAD_NUM_X, THREAD_NUM_X, 1)]
void Update(uint3 id : SV_DispatchThreadID)
```

```
{

  int idx = GetIndex(id.x, id.y);
  float u = _BufferRead[idx].u;
  float v = _BufferRead[idx].v;
  float uvv = u * v * v;
  float f, k;

  f = _Feed;
  k = _K;

  _BufferWrite[idx].u = saturate(
      u + (_DU * LaplaceU(id.x, id.y) - uvv + f * (1.0 - u))
    );
  _BufferWrite[idx].v = saturate(
      v + (_DV * LaplaceV (id.x, id.y) + uvv - (k + f) * v)
    );
}
```

The calculation is exactly the same as the above equation. GetIndex () is a function for associating 2D grid coordinates with 1D ComputeBuffer index.

ReactionDiffusion2D.compute

```
// Buffer index calculation
int GetIndex(int x, int y) {
  x = (x < 0) ? x + _TexWidth : x;
  x = (x >= _TexWidth) ? x - _TexWidth : x;

  y = (y <0)? y + _TexHeight: y;
  y = (y> = _TexHeight)? y - _TexHeight: y;

  return y * _TexWidth + x;
}
```

_BufferRead contains the calculation result one frame before. Extract u and v from there. LaplaceU and LaplaceV are Laplacian functions that collect the U and V concentrations of 8 squares around your grid. This will average the surrounding grid and density. The diagonal grid is adjusted to have a low degree of influence.

ReactionDiffusion2D.compute

```
// Laplacian function of U
float LaplaceU(int x, int y) {
```

```
  float sumU = 0;

  for (int i = 0; i < 9; i++) {
    int2 pos = laplaceIndex[i];
    int idx = GetIndex(x + pos.x, y + pos.y);
    sumU += _BufferRead[idx].u * laplacePower[i];
  }

  return sumU;
}

// Laplacian function of V
float LaplaceV(int x, int y) {
  float sumV = 0;

  for (int i = 0; i < 9; i++) {
    int2 pos = laplaceIndex[i];
    int idx = GetIndex(x + pos.x, y + pos.y);
    sumV += _BufferRead[idx].v * laplacePower[i];
  }

  return sumV;
}
```

After calculating u and v, write to _BufferWrite. saturate is insurance for clipping between 0 and 1.

5.3.4 Input processing

By pressing the A key and C key, the function to intentionally add the density difference between U and V to the grid is provided. Press the A key to place SeedNum points (Seeds) at random positions. Press the C key to place one point in the center.

ReactionDiffusion2D.cs

```
/// <summary>
/// Add Seed
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
void AddSeed(int x, int y)
{
  if (inputIndex < inputMax)
  {
```

```
    inputData[inputIndex].x = x;
    inputData [inputIndex] .y = y;
    inputIndex++;
  }
}
```

The inputData array stores the coordinates of the points on the grid.

ReactionDiffusion2D.cs

```
/// <summary>
/// Pass the Seed array to the Compute Shader
/// </summary>
void AddSeedBuffer()
{
  if (inputIndex > 0)
  {
    inputBuffer.SetData(inputData);
    cs.SetInt("_InputNum", inputIndex);
    cs.SetInt("_TexWidth", texWidth);
    cs.SetInt("_TexHeight", texHeight);
    cs.SetInt("_SeedSize", seedSize);
              cs.SetBuffer(kernelAddSeed,    "_InputBufferRead",
inputBuffer);
       cs.SetBuffer(kernelAddSeed,  "_BufferWrite",  buffers[0]);
// update前なので0
    cs.Dispatch(kernelAddSeed,
      Mathf.CeilToInt((float)inputIndex / (float)THREAD_NUM_X),
      1,
      1);
    inputIndex = 0;
  }
}
```

inputBuffer in, set the inputeData array coordinates entered in the previous point, you pass to ComputeShader.

ReactionDiffusion2D.compute

```
// Add seed
[numthreads(THREAD_NUM_X, 1, 1)]
void AddSeed(uint id : SV_DispatchThreadID)
{
  if (_InputNum <= id) return;

  int w = _SeedSize;
```

```
  int h = _SeedSize;
  float radius = _SeedSize * 0.5;

  int centerX = _InputBufferRead[id].x;
  int centerY = _InputBufferRead[id].y;
  int startX = _InputBufferRead[id].x - w / 2;
  int startY = _InputBufferRead[id].y - h / 2;
  for (int x = 0; x < w; x++)
  {
    for (int y = 0; y < h; y++)
    {
      float dis = distance(
        float2(centerX, centerY),
        float2(startX + x, startY + y)
      );
      if (dis <= radius) {
        _BufferWrite[GetIndex((centerX + x), (centerY + y))].v =
1;
      }
    }
  }
}
```

The value of v is set to 1 so that it becomes a circle around the coordinates of the inputBuffer passed from C #.

5.3.5 Write the result to RenderTexture

Since the updated grid is just an array, write it to RenderTexture for visualization and make it an image. Write the density difference between u and v in RenderTexture.
First, create a Render Texture. Since the only information to be written to one pixel is the density difference, set RenderTextureFormat to RFloat. RenderTextureFormat.RFloat is a RenderTexture format that can write information for one float per pixel.

ReactionDiffusion2D.cs

```
/// <summary>
/// Create Render Texture
/// </summary>
/// <param name="width"></param>
/// <param name="height"></param>
/// <returns></returns>
```

```
RenderTexture CreateRenderTexture(int width, int height)
{
  RenderTexture tex = new RenderTexture(width, height, 0,
    RenderTextureFormat.RFloat,
    RenderTextureReadWrite.Linear);
  tex.enableRandomWrite = true;
  tex.filterMode = FilterMode.Bilinear;
  tex.wrapMode = TextureWrapMode.Repeat;
  tex.Create();

  return tex;
}
```

Next, it is the process on the C # side that passes RenderTexture to ComputeShader and writes it.

ReactionDiffusion2D.cs

```
/// <summary>
/// Write the result of Reaction Diffusion to the texture
/// </summary>
void DrawTexture()
{
  cs.SetInt("_TexWidth", texWidth);
  cs.SetInt("_TexHeight", texHeight);
  cs.SetBuffer(kernelDraw, "_BufferRead", buffers[0]);
  cs.SetTexture(kernelDraw, "_HeightMap", resultTexture);
  cs.Dispatch(kernelDraw,
    Mathf.CeilToInt((float)texWidth / THREAD_NUM_X),
    Mathf.CeilToInt((float)texHeight / THREAD_NUM_X),
    1);
}
```

This is the processing on the Compute Shader side, the density difference between u and v is obtained from the buffer of the grid and written to the texture.

ReactionDiffusion2D.compute

```
// Value calculation for texture writing
float GetValue(int x, int y) {
  int idx = GetIndex(x, y);
  float u = _BufferRead[idx].u;
  float v = _BufferRead[idx].v;
  return 1 - clamp(u - v, 0, 1);
```

```
}

...

// Draw on texture
[numthreads(THREAD_NUM_X, THREAD_NUM_X, 1)]
void Draw(uint3 id : SV_DispatchThreadID)
{
  float c = GetValue(id.x, id.y);

  // height map
  _HeightMap[id.xy] = c;

}
```

5.3.6 Drawing

The normal Unlit Shader is modified and the two colors are interpolated based on the brightness of the texture created in the previous section.

ReactionDiffusion2D.cs

```
/// <summary>
/// Material update
/// </summary>
void UpdateMaterial()
{
  material.SetTexture("_MainTex", resultTexture);

  material.SetColor("_Color0", bottomColor);
  material.SetColor("_Color1", topColor);
}
```

ReactionDiffusion2D.shader

```
fixed4 frag (v2f i) : SV_Target
{
  // sample the texture
  fixed4 col = lerp(_Color0, _Color1, tex2D(_MainTex, i.uv).r);
  return col;
}
```

When executed, a creature-like pattern should spread on the screen.

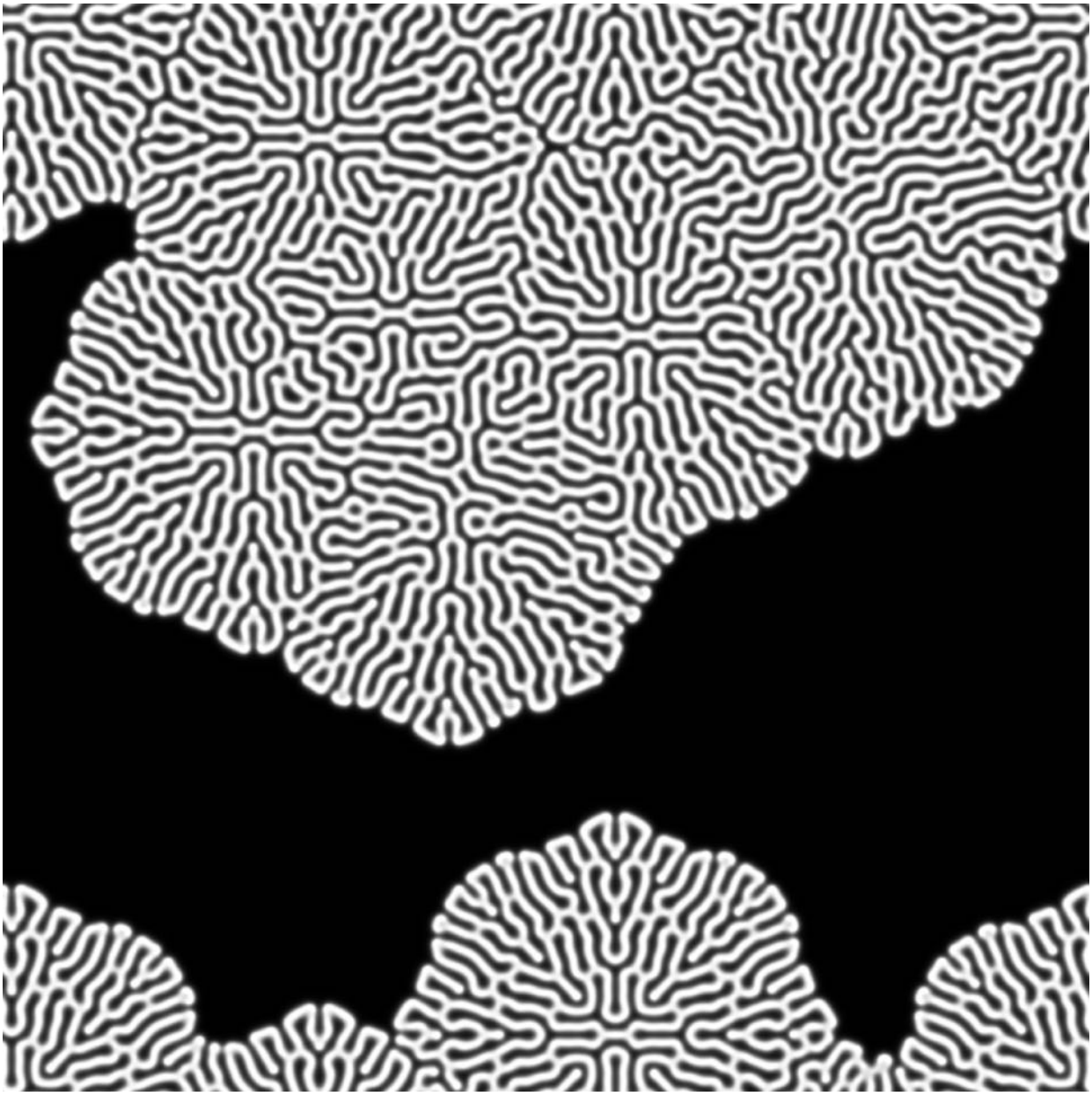

Figure 5.3: Simulation

5.4 Try changing the parameters

Just by changing the Feed and Kill parameters a little, various patterns emerge. Here are some parameter combinations.

5.4.1 Coral-like pattern

Feed:0.037 / Kill:0.06

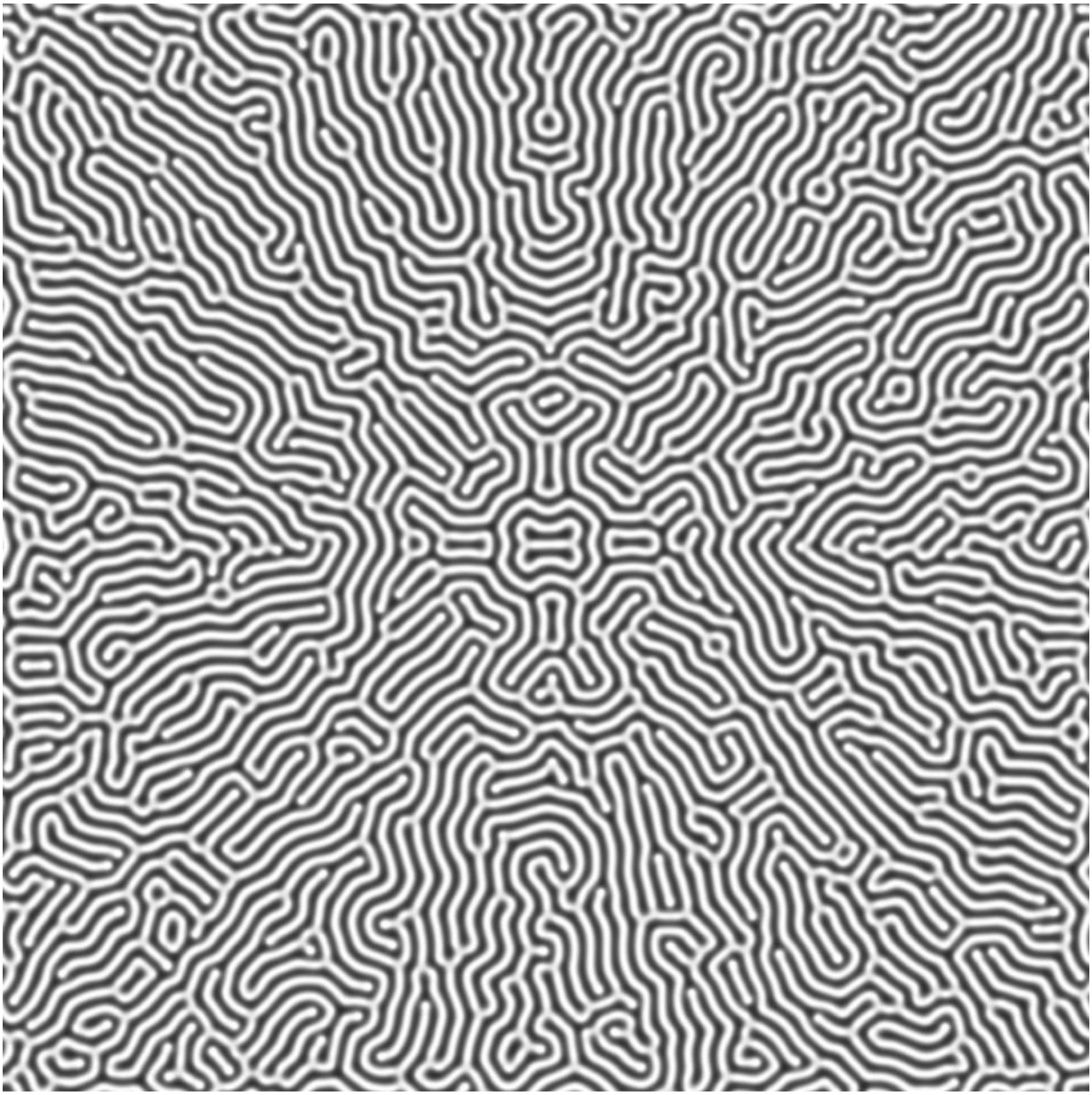

Figure 5.4: Coral-like pattern

5.4.2 Crushed pattern

Feed:0.03 / Kill:0.062

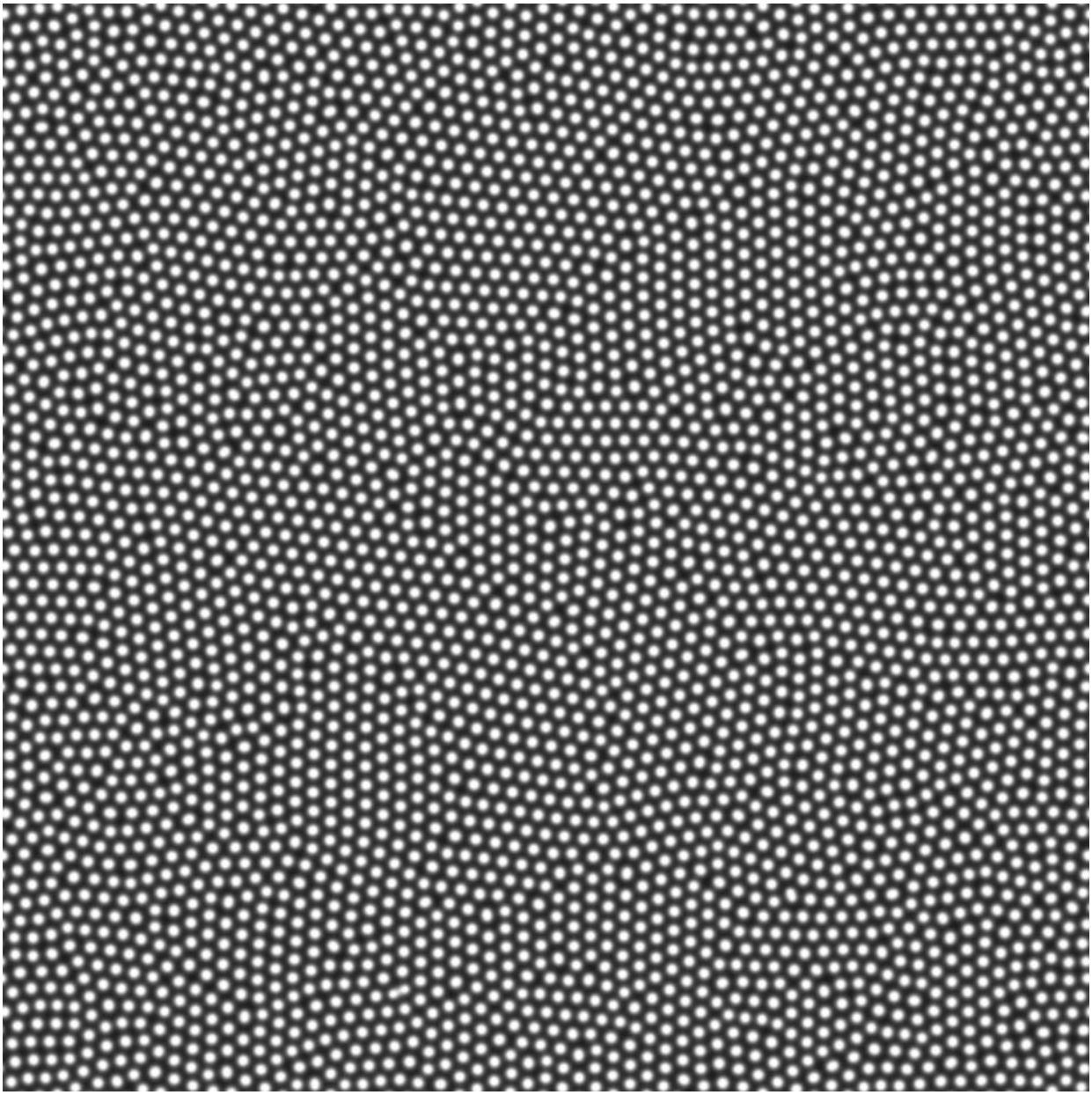

Figure 5.5: Crushed pattern

5.4.3 Crushing seems to repeat disappearance and division

Feed:0.0263 / Kill:0.06

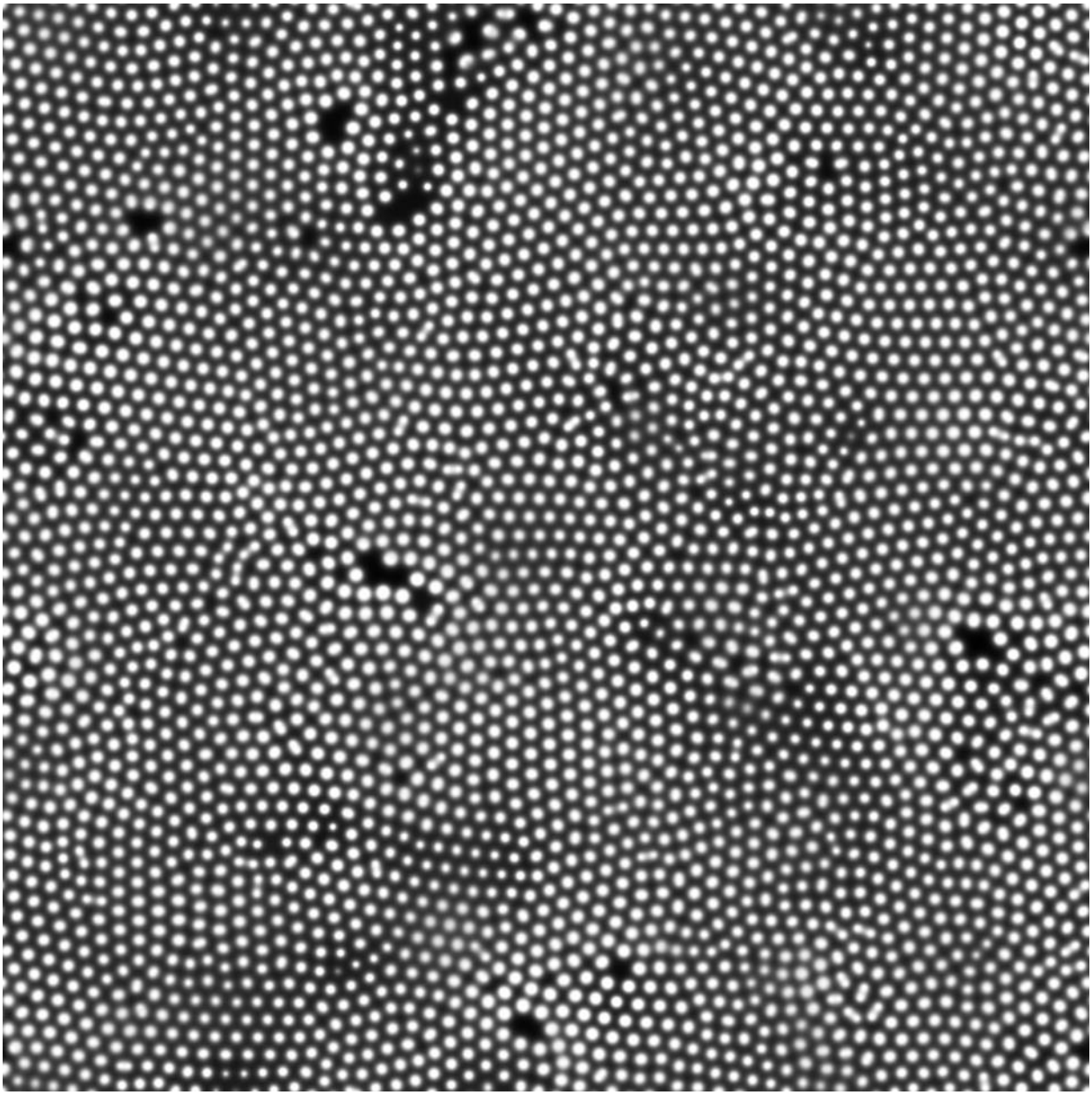

Figure 5.6: Crushing seems to repeat disappearance and division

5.4.4 It seems to stretch straight and avoid hitting

Feed:0.077 / Kill:0.0615

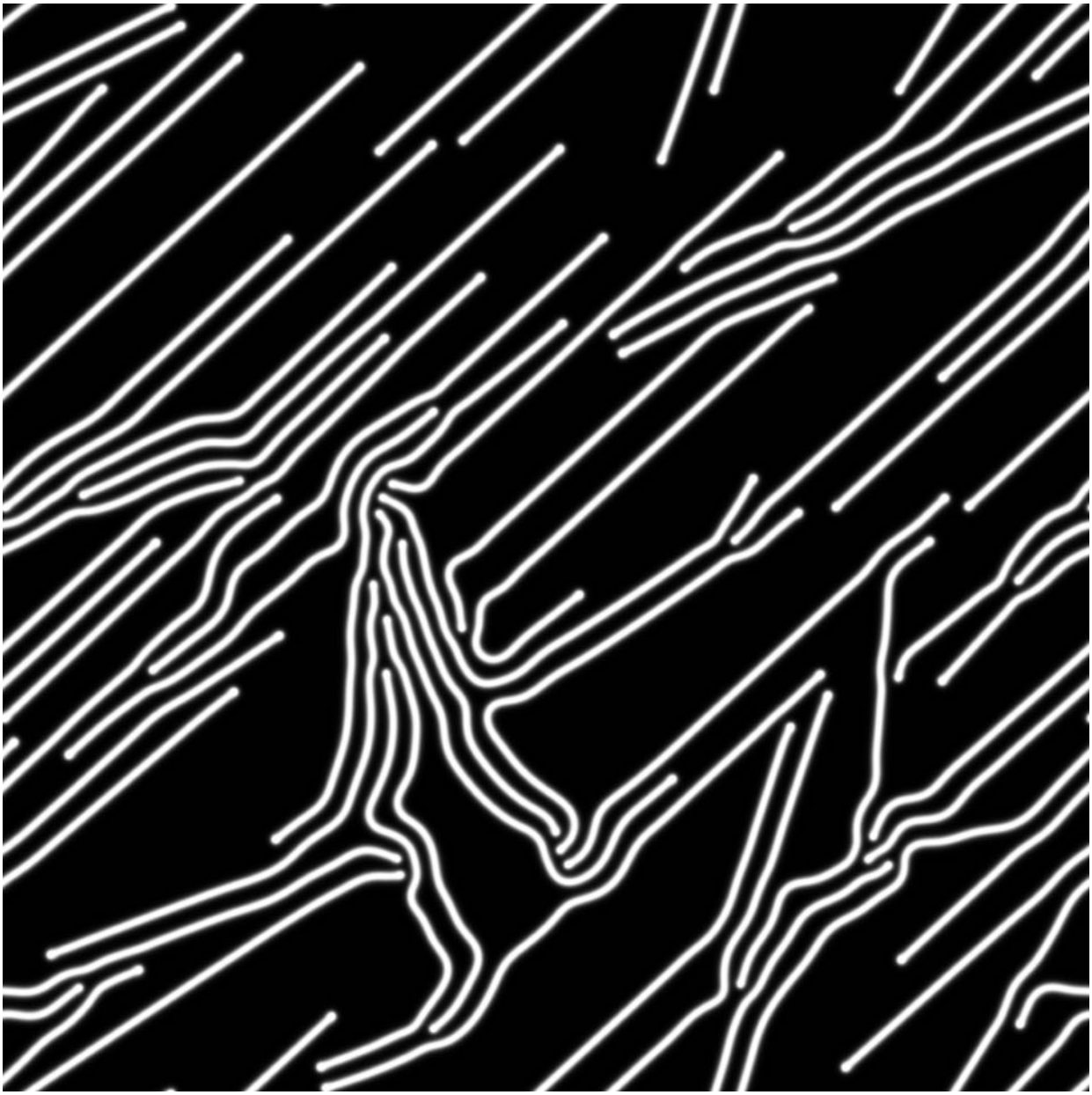

Figure 5.7: A pattern that stretches straight and avoids collision

5.4.5 Plump hole pattern

Feed:0.039 / Kill:0.058

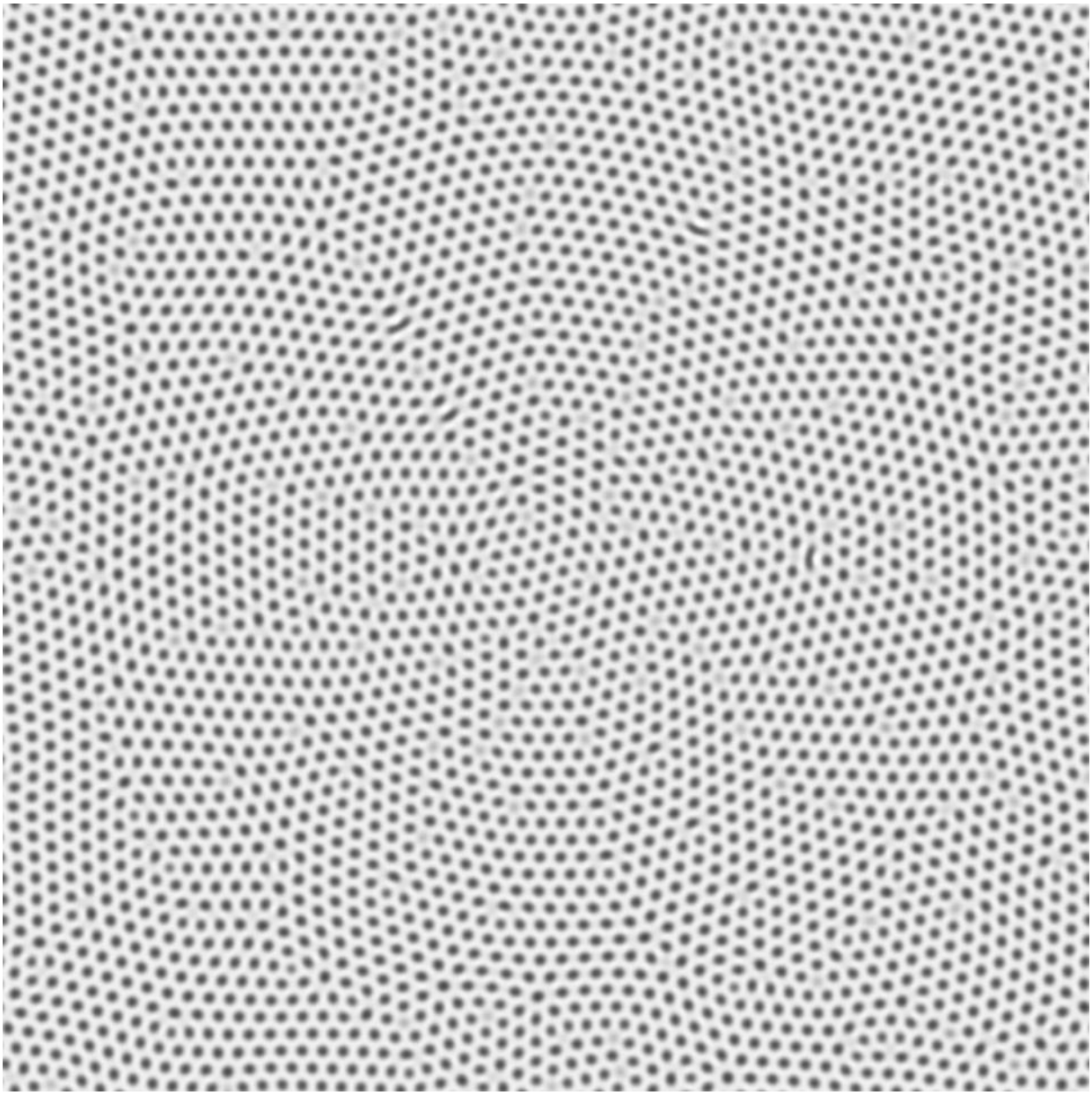

Figure 5.8: Plump hole pattern

5.4.6 It seems that it is always undulating and unstable

Feed:0.026 / Kill:0.051

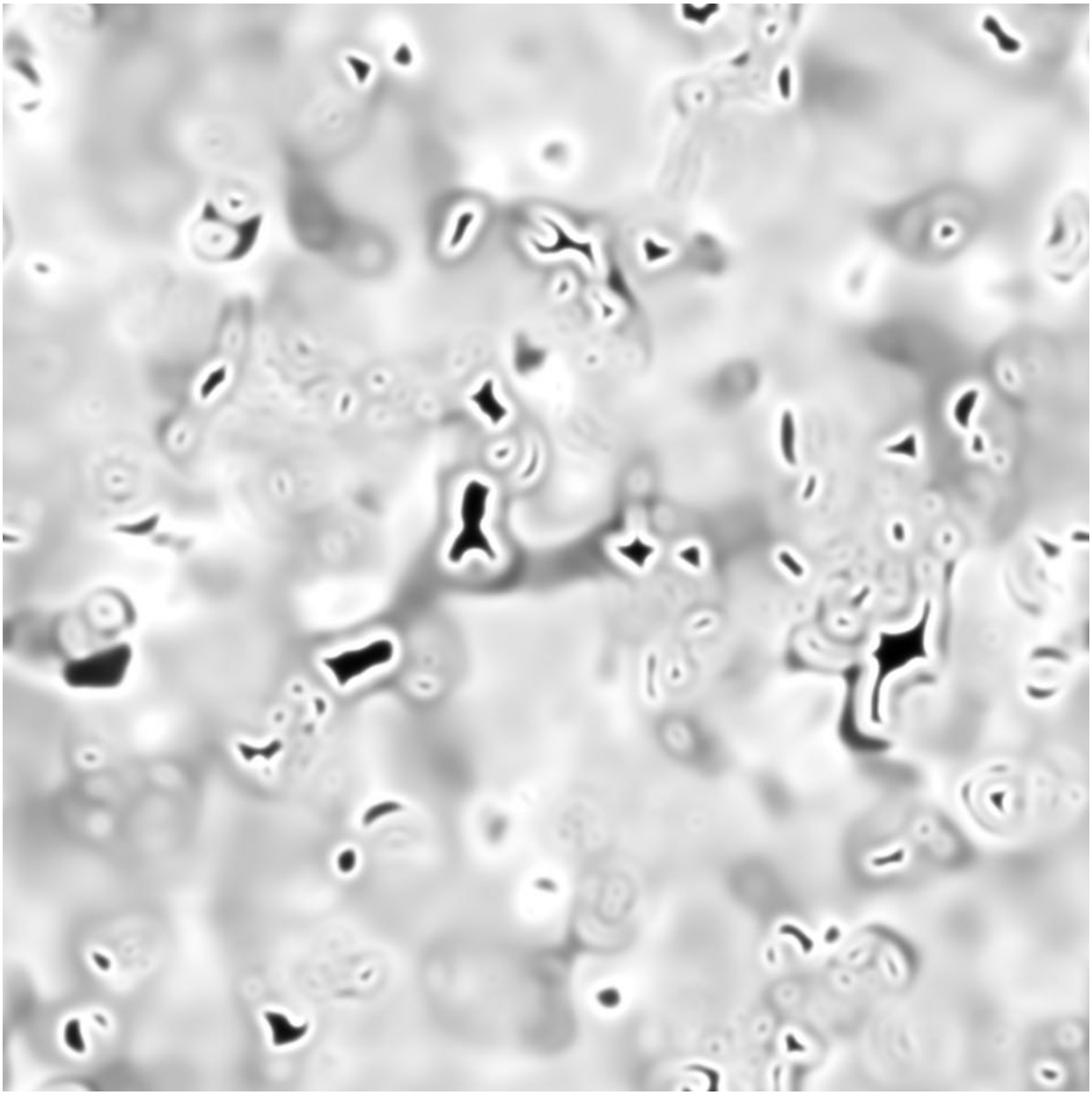

Figure 5.9: It seems that it is always undulating and unstable

5.4.7 A pattern that continues to spread like ripples

Feed:0.014 / Kill:0.0477

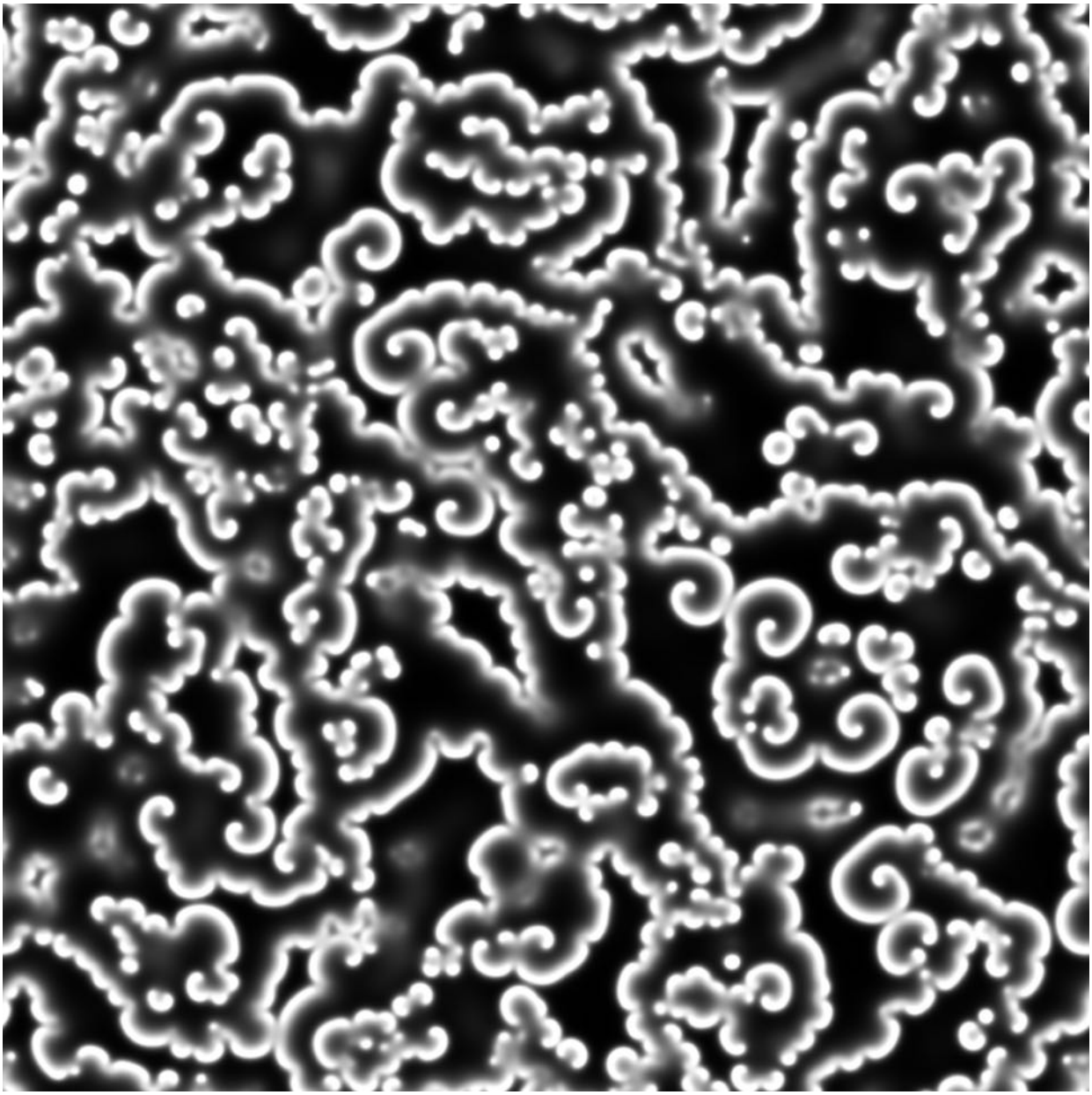

Figure 5.10: Pattern that continues to spread like ripples

5.5 Bonus: Surface Shader compatible version

Here, I will introduce a sample that expresses the beautiful texture unique to Unity using Surface Shader. The sample scene whose operation can be confirmed is **ReactionDiffusion2D_2** .

5.5.1 Changes from the regular version

The process of ReactionDiffusion itself is the same as the normal version, but when creating the texture for drawing, a normal map is also created to give a three-dimensional effect. Also, the resulting texture was RenderTextureFormat.RFloat, but since the normal map stores the normal vector in the XYZ directions, it is created with RenderTextureFormat.ARGBFloat.

ReactionDiffusion2DForStandard.cs

```
void Initialize()
{
  ...
  heightMapTexture = CreateRenderTexture(texWidth, texHeight,
      RenderTextureFormat.RFloat); // Create texture for height
map
  normalMapTexture = CreateRenderTexture(texWidth, texHeight,
    RenderTextureFormat.ARGBFloat); // Create texture for normal
map map
  ...
}

/// <summary>
/// Create RenderTexture
/// </summary>
/// <param name="width"></param>
/// <param name="height"></param>
/// <param name="texFormat"></param>
/// <returns></returns>
RenderTexture CreateRenderTexture(
  int width,
  int height,
  RenderTextureFormat texFormat)
{
    RenderTexture tex = new RenderTexture(width, height, 0,
      texFormat, RenderTextureReadWrite.Linear);
    tex.enableRandomWrite = true;
    tex.filterMode = FilterMode.Bilinear;
    tex.wrapMode = TextureWrapMode.Repeat;
    tex.Create();

    return tex;
}

...

void DrawTexture()
```

```
{
  ...
  cs.SetTexture(kernelDraw, "_HeightMap", heightMapTexture);
  cs.SetTexture (kernelDraw, "_NormalMap", normalMapTexture); //
Texture set for normal map
  cs.Dispatch(kernelDraw,
    Mathf.CeilToInt((float)texWidth / THREAD_NUM_X),
    Mathf.CeilToInt((float)texHeight / THREAD_NUM_X),
    1);
}
```

In ComputeShader, the slope is calculated from the density difference with the surrounding grid and written to the texture for the normal map.

ReactionDiffusion2DStandard.compute

```
float3 GetNormal(int x, int y) {
  float3 normal = float3(0, 0, 0);
  float c = GetValue(x, y);
  normal.x = ((GetValue(x - 1, y) - c) - (GetValue(x + 1, y) -
c));
  normal.y = ((GetValue(x, y - 1) - c) - (GetValue(x, y + 1) -
c));
  normal.z = 1;
  normal = normalize(normal) * 0.5 + 0.5;
  return normal;
}

...

// Draw on texture
[numthreads(THREAD_NUM_X, THREAD_NUM_X, 1)]
void Draw(uint3 id : SV_DispatchThreadID)
{
  float c = GetValue(id.x, id.y);

  // height map
  _HeightMap[id.xy] = c;

  // normal map
  _NormalMap[id.xy] = float4(GetNormal(id.x, id.y), 1);
}
```

Pass the two created textures to the Surface Shader and draw the pattern. Surface Shader is a shader wrapped for easy use of Unity's physics-based rendering, just assign the necessary data to the **SurfaceOutputStandard**

structure in the surf function and output it, and it will automatically **light** it. ..

Definition of SurfaceOutputStandard structure

```
struct SurfaceOutputStandard
{
    fixed3 Albedo; // Base (diffuse or specular) color
    fixed3 Normal; // normal
    half3 Emission; // Emission color
    half Metallic; // 0 = non-metal, 1 = metal
    half Smoothness; // 0 = coarse, 1 = smooth
    half Occlusion; // Occlusion (default 1)
    fixed Alpha; // Transparency alpha
};
```

ReactionDiffusion2DStandard.shader

```
void surf(Input IN, inout SurfaceOutputStandard o) {

  float2 uv = IN.uv_MainTex;

  // Get concentration
  half v0 = tex2D(_MainTex, uv).x;

  // Normal acquisition
  float3 norm = UnpackNormal(tex2D(_NormalTex, uv));

  // Get the value of the boundary between A and B
  half p = smoothstep(_Threshold, _Threshold + _Fading, v0);

  o.Albedo = lerp (_Color0.rgb, _Color1.rgb, p); // Base color
  o.Alpha = lerp (_Color0.a, _Color1.a, p); // Alpha value
    o.Smoothness = lerp (_Smoothness0, _Smoothness1, p); //
Smoothness
  o.Metallic = lerp (_Metallic0, _Metallic1, p); // Metallic
      o.Normal   =   normalize(float3(norm.x,   norm.y,   1   -
_NormalStrength));  // 法線

  o.Emission = lerp (_Emit0 * _EmitInt0, _Emit1 * _EmitInt1, p)
.rgb; //??
}
```

Use Unity's built-in function **unpackNormal** to get the normals from the normal map. In addition, various colors and textures of **Surface Output**

Standard are set from the ratio of density difference .

When you run it, you should see something like the following.

Figure 5.11: Surface Shader version

The normal map creates a three-dimensional effect. In addition, although it is not known in monochrome, the gloss of the RGB 3-color point light in the scene is also expressed.

5.6 Extension to 3D

Let's extend Reaction Diffusion, which used to be a simulation on a two-dimensional plane, to three dimensions. The basic flow is the same as for 2D, but since the dimension is increased by 1, the method of creating RenderTexture and ComputeBuffer and the method of Laplace operation are slightly different. The sample scene whose operation can be confirmed is **ReactionDiffusion3D** .

5.6.1 Buffer initialization part

Some initialization processing is added to change the Render Texture to which the density difference is written from 2D to 3D.

ReactionDiffusion3D.cs

```
RenderTexture CreateTexture(int width, int height, int depth)
{
  RenderTexture tex = new RenderTexture(width, height, 0,
    RenderTextureFormat.RFloat, RenderTextureReadWrite.Linear);
  tex.volumeDepth = depth;
  tex.enableRandomWrite = true;
  tex.dimension = UnityEngine.Rendering.TextureDimension.Tex3D;
  tex.filterMode = FilterMode.Bilinear;
  tex.wrapMode = TextureWrapMode.Repeat;
  tex.Create();

  return tex;
}
```

First, put the depth in the Z direction in tex.volumeDepth. Then I put UnityEngine.Rendering.TextureDimension.Tex3D in tex.dimension. This is a setting to specify that RenderTexture is a 3D volume texture. The Render Texture is now a 3D volume texture. Similarly, the Compute Buffer that stores the Reaction Diffusion simulation results is also made three-dimensional. This simply secures the size of width x height x depth.

ReactionDiffusion3D.cs

```
void Initialize()
{
```

```
  ...
  int whd = texWidth * texHeight * texDepth;
  buffers = new ComputeBuffer[2];
  ...
  for (int i = 0; i < buffers.Length; i++)
  {
                    buffers[i]    =    new    ComputeBuffer(whd,
Marshal.SizeOf(typeof(RDData)));
  }
  ...
}
```

5.6.2 Three-dimensional simulation

Next is the change on the Compute Shader side. First, in order to RenderTexture for the writing of the result is a three-dimensional, ComputeShader of the side of the texture definition **RWTexture2D <float>** from **RWTexture3D <float>** will change to.

ReactionDiffusion3D.compute

```
  RWTexture3D <float> _HeightMap; // Heightmap
```

Next is the three-dimensionalization of the Laplacian function. It has been changed to refer to a total of 27 squares of 3x3x3. By the way, the degree of influence of laplacePower is a value that was somehow calculated.

ReactionDiffusion3D.compute

```
// Surrounding index calculation table referenced by the
Laplacian function
static const int3 laplaceIndex[27] = {
  int3 (-1, -1, -1), int3 (0, -1, -1), int3 (1, -1, -1),
  int3 (-1, 0, -1), int3 (0, 0, -1), int3 (1, 0, -1),
  int3 (-1, 1, -1), int3 (0, 1, -1), int3 (1, 1, -1),

  int3 (-1, -1, 0), int3 (0, -1, 0), int3 (1, -1, 0),
  int3 (-1, 0, 0), int3 (0, 0, 0), int3 (1, 0, 0),
  int3 (-1, 1, 0), int3 (0, 1, 0), int3 (1, 1, 0),

  int3 (-1, -1, 1), int3 (0, -1, 1), int3 (1, -1, 1),
  int3 (-1, 0, 1), int3 (0, 0, 1), int3 (1, 0, 1),
  int3 (-1, 1, 1), int3 (0, 1, 1), int3 (1, 1, 1),
};
```

```
// Impact of the grid around the Laplacian
static const float laplacePower[27] = {
  0.02, 0.02, 0.02,
  0.02, 0.1,  0.02,
  0.02, 0.02, 0.02,

  0.02, 0.1,  0.02,
  0.1, -1.0,  0.1,
  0.02, 0.1,  0.02,

  0.02, 0.02, 0.02,
  0.02, 0.1,  0.02,
  0.02, 0.02, 0.02
};

// Buffer index calculation
int GetIndex(int x, int y, int z) {
  x = (x < 0) ? x + _TexWidth : x;
  x = (x >= _TexWidth) ? x - _TexWidth : x;

  y = (y <0)? y + _TexHeight: y;
  y = (y> = _TexHeight)? y - _TexHeight: y;

  z = (z <0)? z + _TexDepth: z;
  z = (z> = _TexDepth)? z - _TexDepth: z;

  return z * _TexWidth * _TexHeight + y * _TexWidth + x;
}

// Laplacian function of U
float LaplaceU(int x, int y, int z) {
  float sumU = 0;

  for (int i = 0; i < 27; i++) {
    int3 pos = laplaceIndex[i];

    int idx = GetIndex(x + pos.x, y + pos.y, z + pos.z);
    sumU += _BufferRead[idx].u * laplacePower[i];
  }
  return sumU;
}

// Laplacian function of V
float LaplaceV(int x, int y, int z) {
  float sumV = 0;

  for (int i = 0; i < 27; i++) {
```

```
    int3 pos = laplaceIndex[i];
    int idx = GetIndex(x + pos.x, y + pos.y, z + pos.z);
    sumV += _BufferRead[idx].v * laplacePower[i];
  }
  return sumV;
}
```

5.6.3 Drawing process

Since the Render Texture of the simulation result is a 3D volume texture, even if you paste the texture on the Unlit Shader or Surface Shader as before, it will not be displayed normally. In the sample, polygons are generated and drawn using a method called the Marching cubes method, but due to space limitations, the explanation of implementation will be omitted. For an explanation of the Marching Cubes method, please refer to "Chapter 7 Introduction to the Marching Cubes Method in an Atmosphere" in Unity Graphics Programming Vol.1. Another method is to draw with volume rendering using ray marching. A very easy-to-understand implementation [* 1] is introduced on Mr. Utsu's blog, so please refer to it.

[* 1] Dent Tips http://tips.hecomi.com/entry/2018/01/05/192332

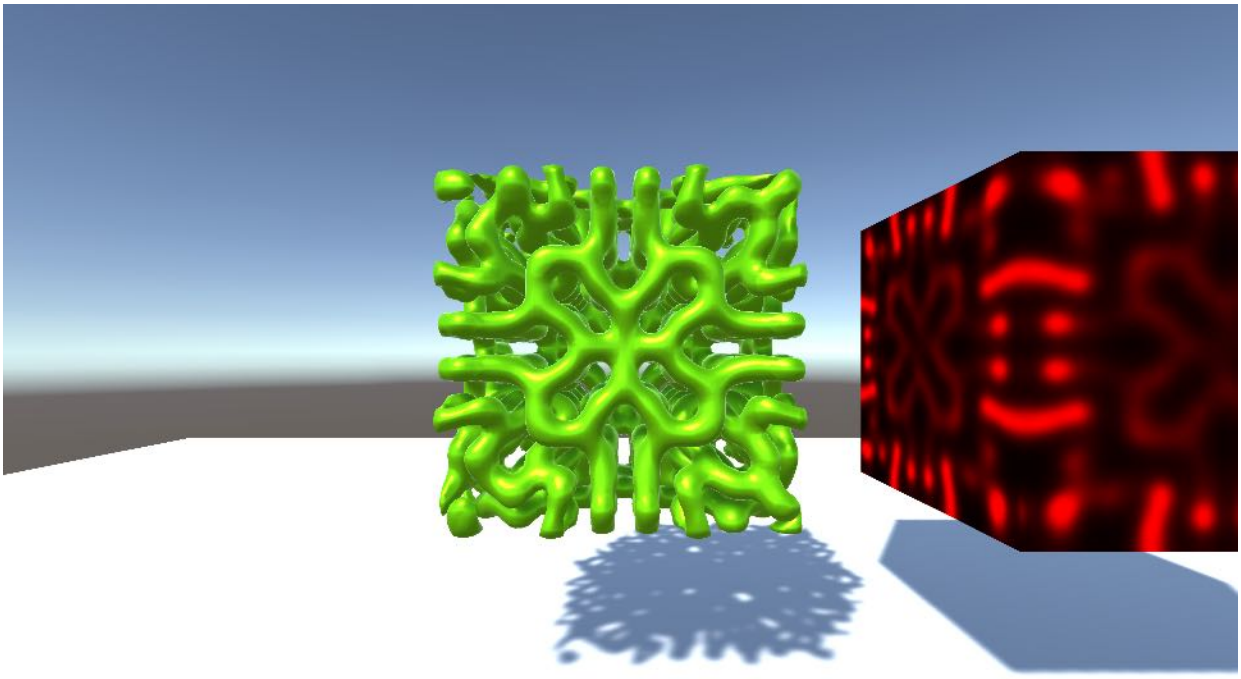

Figure 5.12: 3D Reaction Diffusion

5.7 Summary

I introduced how to make a creature-like pattern using the Gray-Scott model. You can create a completely different pattern by just changing the parameters of Feed and Kill, so be careful as the time will pass quickly if you get absorbed in it (* There are individual differences)
Also, for works using Reaction Diffusion , Nakama Kouhei's "DIFFUSION" * 2 and Kitahara Nobutaka's "Reaction-Diffusion" * 3 . Would you like to be obsessed with the mysterious charm of Reaction Diffusion?

[*2] DIFFUSION https://vimeo.com/145251635

[*3] Reaction-Diffusion https://vimeo.com/176261480

5.8 Reference

- Reaction-Diffusion Tutorial http://www.karlsims.com/rd.html
- Reaction diffusion system(Gray-Scott model)
  https://pmneila.github.io/jsexp/grayscott/

Chapter 6 Strange Attractor

6.1 Introduction

In this chapter, we will develop a visualization of a phenomenon called "Strange Attractor" that shows non-linear chaotic behavior by a differential equation or difference equation with a specific state using Unity and GPU arithmetic.
The sample in this chapter is "Strange Attractors" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
.

6.1.1 Execution environment

- Shader model 5.0 compatible environment where ComputeShader can be executed
- Confirmed operation with Unity 2018.2.9f1 at the time of writing

6.2 Strange Attractorとは

A state in which the motion of a dissipative system (energy non-conservation, an unbalanced system with a specific input and opening) maintains a stable orbit over time is called an "Attractor".
Among them, the one that shows a chaotic behavior by amplifying the slight difference in the initial state with the passage of time is called "Strange Attractor".

In this chapter, I would like to take up " * 1 Lorenz Attractor" and " * 2 Thomas' Cyclically Symmetric Attractor" as the subjects .

6.3 Lorenz Attractor

Do you know the phenomenon called the butterfly effect? This is a word derived from the title of a lecture given by meteorologist Edward N Lorenz

at the American Association for the Advancement of Science in 1972, " * 3 Does the flapping of a single Brazilian butterfly cause a tornado in Texas?" This term describes a phenomenon in which slight differences in initial values do not always produce mathematically similar results, but are amplified chaotically and behave unpredictably.

Lorenz, who pointed out this mathematical property, published "Lorenz Attractor" in 1963.

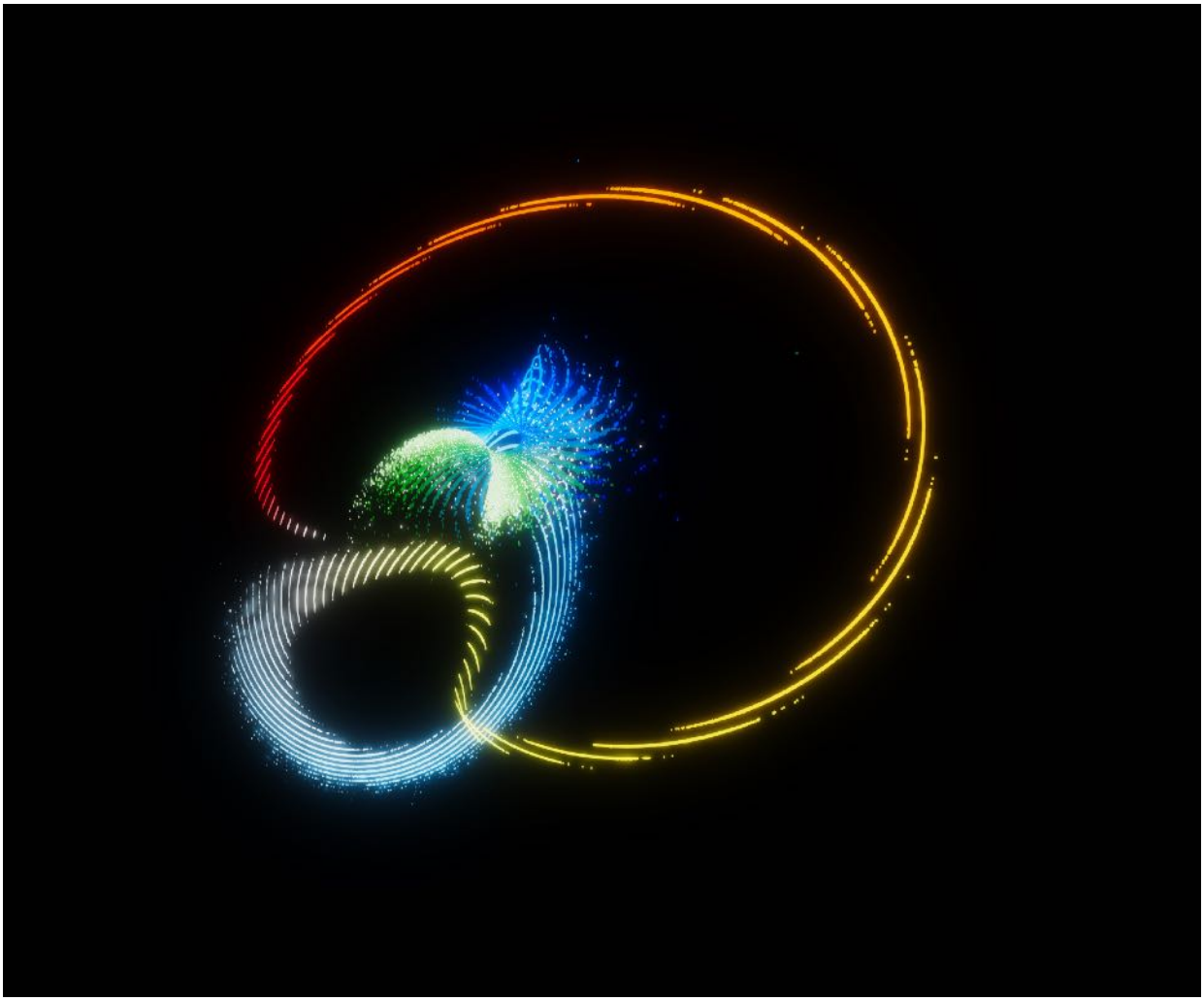

Figure 6.1: Initial state of Lorenz attractor

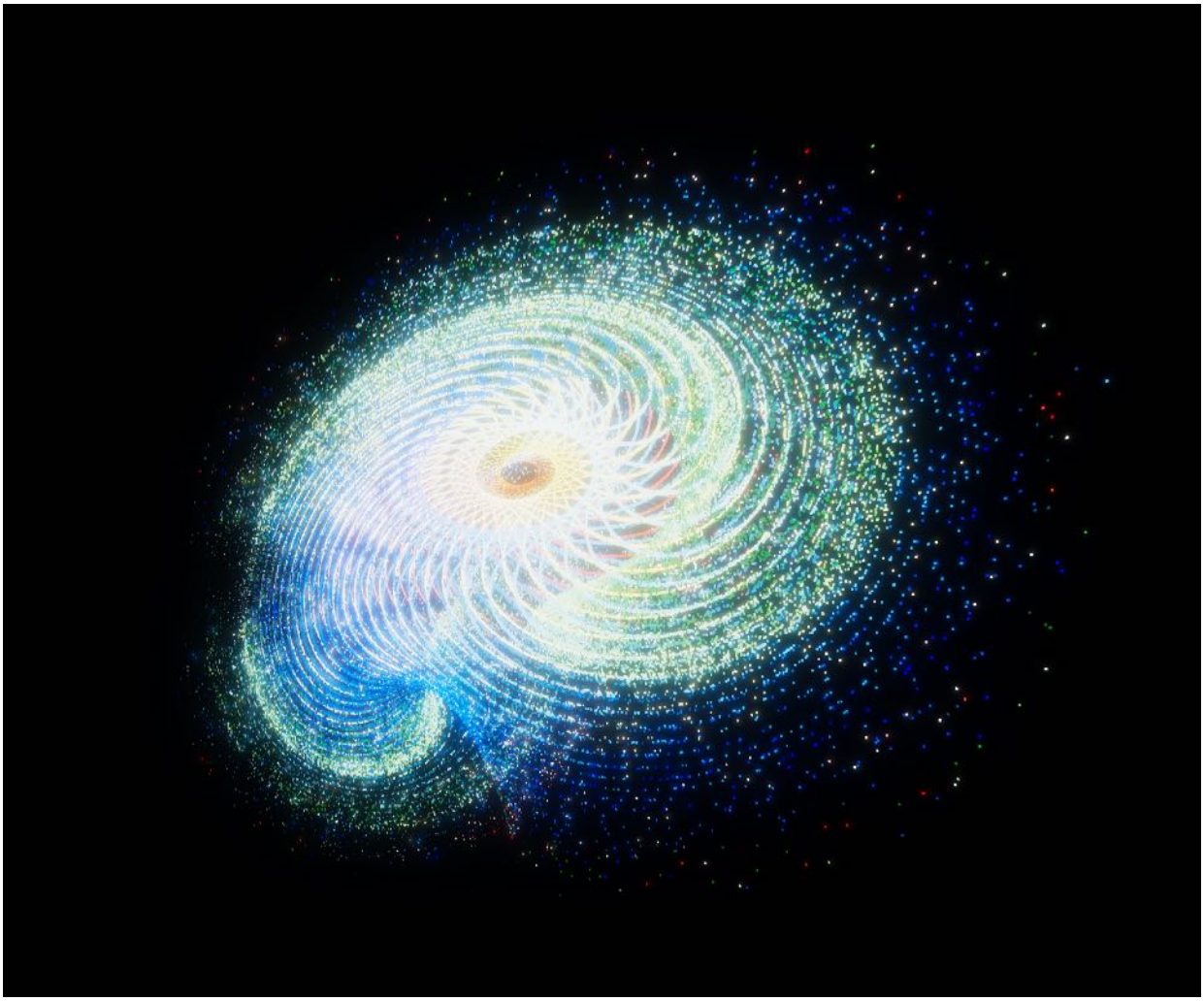

Figure 6.2: Mid-term Lorenz attractor

[*1] Lorenz, E. N.: Deterministic Nonperiodic Flow, Journal of Atmospheric Sciences, Vol.20, pp.130-141, 1963.

[*2] Thomas, René（ 1999） . "Deterministic chaos seen in terms of feedback circuits: Analysis, synthesis, 'labyrinth chaos'". Int. J. Bifurcation and Chaos. 9（ 10）: 1889–1905.

[*3] http://eaps4.mit.edu/research/Lorenz/Butterfly_1972.pdf

6.3.1 Lorenz equation

The Lorenz equation is represented by the following nonlinear ODE.

By setting p = 10, r = 28, b = 8/3 in each variable of p, r, and b in the above equation, it will behave chaotically as "Strange Attractor".

6.3.2 Implementation of Lorenz Attractor

Now let's implement the Lorenz equation with a compute shader. First, define the structure you want to calculate in the compute shader.

StrangeAttractor.cs

```
protected struct Params
{
    Vector3 emitPos;
    Vector3 position;
    Vector3 velocity; // xyz = velocity, w = velocity coef;
    float   life;
    Vector2 size;     // x = current size, y = target size.
    Vector4 color;

    public Params(Vector3 emitPos, float size, Color color)
    {
        this.emitPos = emitPos;
        this.position = Vector3.zero;
        this.velocity = Vector3.zero;
        this.life = 0;
        this.size = new Vector2(0, size);
        this.color = color;
    }
}
```

Since this structure will be used universally in multiple Strange Attractors in the future, it is defined in the abstract class StrangeAttractor.cs.

Next, the Compute Buffer is initialized.

LorenzAttrator.cs

```
protected sealed override void InitializeComputeBuffer()
{
    if (cBuffer != null)  cBuffer.Release();
```

```
            cBuffer = new ComputeBuffer(instanceCount,
Marshal.SizeOf(typeof(Params)));
    Params[] parameters = new Params[cBuffer.count];
    for (int i = 0; i < instanceCount; i++)
    {
        var normalize = (float)i / instanceCount;
        var color = gradient.Evaluate(normalize);
        parameters[i] = new Params(Random.insideUnitSphere *
            emitterSize * normalize, particleSize, color);
    }
    cBuffer.SetData(parameters);
}
```

The abstract method InitializeComputeBuffer defined in the abstract class StrangeAttractor.cs is implemented in LorenzAttrator.cs.
Since I want to adjust the gradation, emitter size, and particle size in Unity's inspector, initialize the Params structure with the gradient, emitterSize, and particleSize exposed in the inspector, and setData to the ComputeBuffer variable, cBuffer.
This time, I want to apply the velocity vector by delaying it little by little in the order of particle id, so I add the gradation color in order of particle id.
In "Strange Attractor", the initial position is greatly related to the subsequent behavior depending on the thing, so I would like you to try various initial positions, but in this sample, the sphere is the initial shape.

Then pass the LorenzAttrator variables p, r, b to the compute shader.

LorenzAttrator.cs

```
[SerializeField, Tooltip("Default is 10")]
float p = 10f;
[SerializeField, Tooltip("Default is 28")]
float r = 28f;
[SerializeField, Tooltip("Default is 8/3")]
float b = 2.666667f;

private int pId, rId, bId;
private string pProp = "p", rProp = "r", bProp = "b";

protected override void InitializeShaderUniforms()
{
    pId = Shader.PropertyToID(pProp);
    rId = Shader.PropertyToID(rProp);
    bId = Shader.PropertyToID(bProp);
```

```
}

protected override void UpdateShaderUniforms()
{
    computeShaderInstance.SetFloat(pId, p);
    computeShaderInstance.SetFloat(rId, r);
    computeShaderInstance.SetFloat(bId, b);
}
```

Next, initialize the state of particles at the time of emission on the compute shader side.

LorenzAttractor.compute

```
#pragma kernel Emit
#pragma kernel Iterator

#define THREAD_X 128
#define THREAD_Y 1
#define THREAD_Z 1
#define DT 0.022

struct Params
{
    float3 emitPos;
    float3 position;
    float3 velocity; //xyz = velocity
    float  life;
    float2 size;     // x = current size, y = target size.
    float4 color;
};

RWStructuredBuffer<Params> buf;

[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void Emit(uint id : SV_DispatchThreadID)
{
    Params p = buf[id];
    p.life = (float)id * -1e-05;
    p.position = p.emitPos;
    p.size.x = 0.0;
    buf[id] = p;
}
```

Initialization is performed by the Emit method. p.life manages the time since the generation of particles, and provides a small delay for each id at the time

of the initial value.
This is to easily prevent the particles from drawing the same trajectory all at once. Also, since the gradation color is set for each id, it is useful for making the color look beautiful.
Here, the particle size p.size is set to 0 at the initial stage, but this is to make the particles invisible at the moment of occurrence to make the balloon natural.

Next, let's look at the iteration part.

LorenzAttractor.compute

```
#define DT 0.022

// Lorenz Attractor parameters
float p;
float r;
float b;

// The arithmetic part of the Lorenz equation.
float3 LorenzAttractor(float3 pos)
{
    float dxdt = (p * (pos.y - pos.x));
    float dydt = (pos.x * (r - pos.z) - pos.y);
    float dzdt = (pos.x * pos.y - b * pos.z);
    return float3(dxdt, dydt, dzdt) * DT;
}

[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void Iterator(uint id : SV_DispatchThreadID)
{
    Params p = buf[id];
    p.life.x += DT;
    // Clamp the vector length of the velocity vector from 0 to
1 and multiply it by the size to make the start look natural.
    p.size.x = p.size.y * saturate(length(p.velocity));
    if (p.life.x > 0)
    {
        p.velocity = LorenzAttractor(p.position);
        p.position += p.velocity;
    }
    buf[id] = p;
}
```

The LorenzAttractor method above is the arithmetic part of the "Lorenz equation". The velocity vector of x, y, z with a small amount of delta time is calculated, and finally the delta time is multiplied to derive the amount of movement.
From experience, when performing derivative operations related to the shape in the compute shader, it is better to use a fixed value delta time independent of the frame rate difference instead of sending the delta time from Unity to maintain a stable shape.
This is because if the frame rate drops too much, the value of Unity's Time.deltaTime may become too large for differential operations. The larger the delta width, the rougher the calculation result will be compared to the previous one.
Another reason is that, depending on the equation, the "Strange Attractor" may completely converge or diverge infinitely depending on how the delta time is taken.
For these two reasons, DT is using the predefined ones this time.

6.4 Thomas' Cyclically Symmetric Attractor

Next, I would like to implement the "Thomas' Cyclically Symmetric Attractor" announced by biologist René Thomas.
It is not affected by the initial value, it becomes stable over time, and the shape is very unique.

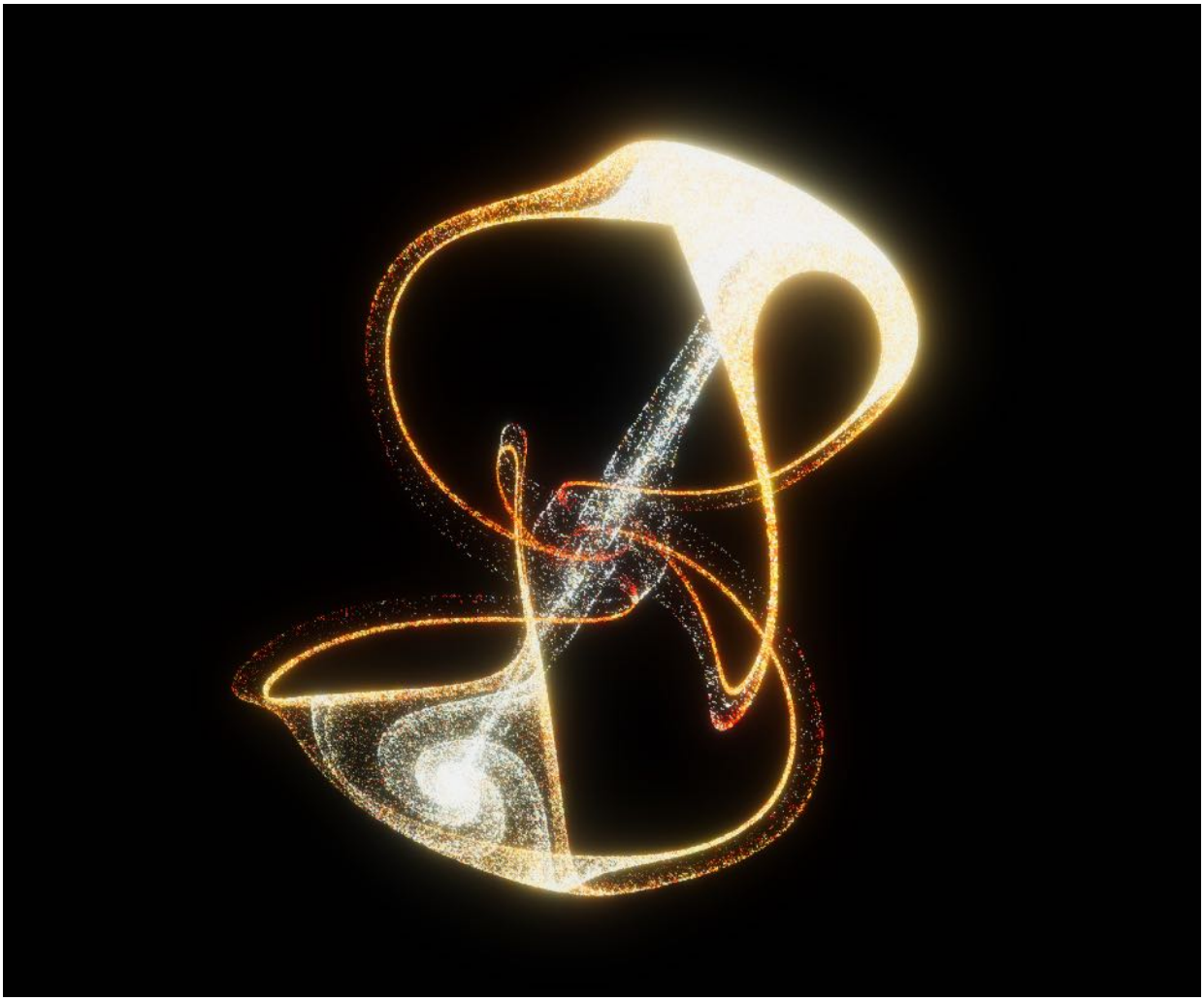

Figure 6.3: Thomas' Cyclically Symmetric Attractor Stable Period

6.4.1 Thomas' Cyclically Symmetric equation

The equation is represented by the following nonlinear ODE.

In the variable b of the above equation, if it is set as b \ simeq 0.208186, it behaves chaotically as "Strange Attractor", and if it is set as b \ simeq 0, it floats in space.

6.4.2 Implementation of Thomas' Cyclically Symmetric Attractor

Now let's implement the "Thomas' Cyclically Symmetric equation" with a compute shader.
Since there is a part in common with the implementation of "Lorenz Attractor" mentioned above, the parameter structure and procedural part are inherited and only the necessary part is taken up.
First, override the color and initial position on the CPU side.

ThomasAttractor.cs

```
protected sealed override void InitializeComputeBuffer()
{
    if (cBuffer != null)  cBuffer.Release();

                cBuffer    =    new    ComputeBuffer(instanceCount,
Marshal.SizeOf(typeof(Params)));
    Params[] parameters = new Params[cBuffer.count];
    for (int i = 0; i < instanceCount; i++)
    {
        var normalize = (float)i / instanceCount;
        var color = gradient.Evaluate(normalize);
        parameters[i] = new Params(Random.insideUnitSphere *
            emitterSize * normalize, particleSize, color);
    }
    cBuffer.SetData(parameters);
}
```

This time, in order to make the colors look beautiful, the initial position is a sphere with a mantle-like gradation color from the inside to the outside.

Next, let's look at the compute shader methods at the time of emission and iteration.

ThomasAttractor.compute

```
//Thomas Attractor parameters
float b;

float3 ThomasAttractor(float3 pos)
{
    float dxdt = -b * pos.x + sin(pos.y);
    float dydt = -b * pos.y + sin(pos.z);
    float dzdt = -b * pos.z + sin(pos.x);
    return float3(dxdt, dydt, dzdt) * DT;
}
```

```
[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void Emit(uint id : SV_DispatchThreadID)
{
    Params p = buf[id];
    p.life = (float)id * -1e-05;
    p.position = p.emitPos;
    p.size.x = p.size.y;
    buf[id] = p;
}

[numthreads(THREAD_X, THREAD_Y, THREAD_Z)]
void Iterator(uint id : SV_DispatchThreadID)
{
    Params p = buf[id];
    p.life.x += DT;
    if (p.life.x > 0)
    {
        p.velocity = ThomasAttractor(p.position);
        p.position += p.velocity;
    }
    buf[id] = p;
}
```

The ThomasAttractor method becomes the arithmetic part of the "Thomas' Cyclically Symmetric equation".
Also, unlike LorenzAttrator, the implementation at Emit is set from the initial size to the target size because I want to show the initial position this time.
Others have almost the same implementation.

6.5 Summary

In this chapter, we introduced an example of implementing "Strange Attractor" on GPU using a compute shader.
There are various types of "Strange Attractor", and even in the implementation, it shows chaotic behavior with relatively few operations, so it may be a useful accent in graphics as well.
There are many other types, such as a two-dimensional motion called " * 4 Ueda Attractor" and a spin motion like " * 5 Aizawa Attractor", so if you are interested, please try it.

[*4] http://www-lab23.kuee.kyoto-u.ac.jp/ueda/Kambe-Bishop_ver3-1.pdf

[*5] http://www.algosome.com/articles/aizawa-attractor-chaos.html

6.6 Reference

- http://paulbourke.net/fractals/lorenz/
- https://en.wikipedia.org/wiki/Thomas%27_cyclically_symmetric_attractor
- Lorenz, E. N. : Deterministic Nonperiodic Flow, Journal of Atmospheric Sciences, Vol.20, pp.130-141, 1963.
- Thomas, René (1999) . "Deterministic chaos seen in terms of feedback circuits: Analysis, synthesis, 'labyrinth chaos'". Int. J. Bifurcation and Chaos. 9 (10) : 1889–1905.

I tried to implement Chapter 7 Portal in Unity

7.1 Introduction

Do you know the game Portal * 1 ? A puzzle action game released by Valve in 2007. It is a god game. The feature is a hole called a portal, and you can see the scenery on the other side as if the two holes are connected by a worm home, and you can warp through objects and your character. The point is Anywhere Door. You can set up a portal on a flat surface with a gun called a portal gun, and use this to advance the game. This chapter is an article that tried to implement the function of this portal in Unity while being simple. The sample is "PortalGate System" of
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
.

[*1]
https://ja.wikipedia.org/wiki/Portal_(%E3%82%B2%E3%83%BC%E3%83%A0)

7.2 Project overview

Think about the necessary elements as a place to play on the portal.

- Own character that can move
- field
- Portal gun (function to make a portal appear at a specified position)

I want a hit. Your character may be visible on the other side of the portal, so it's a first-person perspective, but you'll need a full-body model. This time, I used the **Adam** * 2 model distributed by Unity . Also, I want to see the objects warping other than my character, so I can launch a red ball with the E button.

[*2] https://assetstore.unity.com/packages/essentials/tutorial-projects/adam-character-pack-adam-guard-lu-74842

The operation method is as follows.

- Move: WASD key or arrow key (hold down shift and dash)
- Viewpoint movement: Mouse movement
- Portal launch: Mouse left and right click
- Ball launch: E key
- Jump: Spacebar

From now on, the hole to warp is called a gate. The class name is **PortalGate in the** source code .

7.2.1 Own character

My character was created by modifying Unity's Standard Assets * 3 . It's like using the animation of ThirdPersonChracter while modifying the control of FirstPersonCharacter. Field of view because if (the main camera) to the user's character itself will be reflected polygon is not clean Dari sinks **Player** in the main camera provided a layer **Player** we try to not displayed to set the layer to the camera of CullingMask.

[*3] https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351

7.2.2 Field

For the field, I used **unity3d-jp** 's level design asset **playGROWnd** * 4 . Somehow the atmosphere is like Portal. This time, the field is a rectangular parallelepiped room to simplify the rest of the process. Collisions are not used as they are, but transparent collisions are placed on each side of the rectangular parallelepiped. The floor collision is wider than the room to prevent it from falling, as the post-warp object may be partially outside the room. This is the **Stage Coll** layer.

[*4] https://github.com/unity3d-jp/playgrownd

7.3 Gate generation

Now let's implement the gate. The gate this time is oriented to spread on the XY plane in the local coordinate system and pass through the Z + direction.

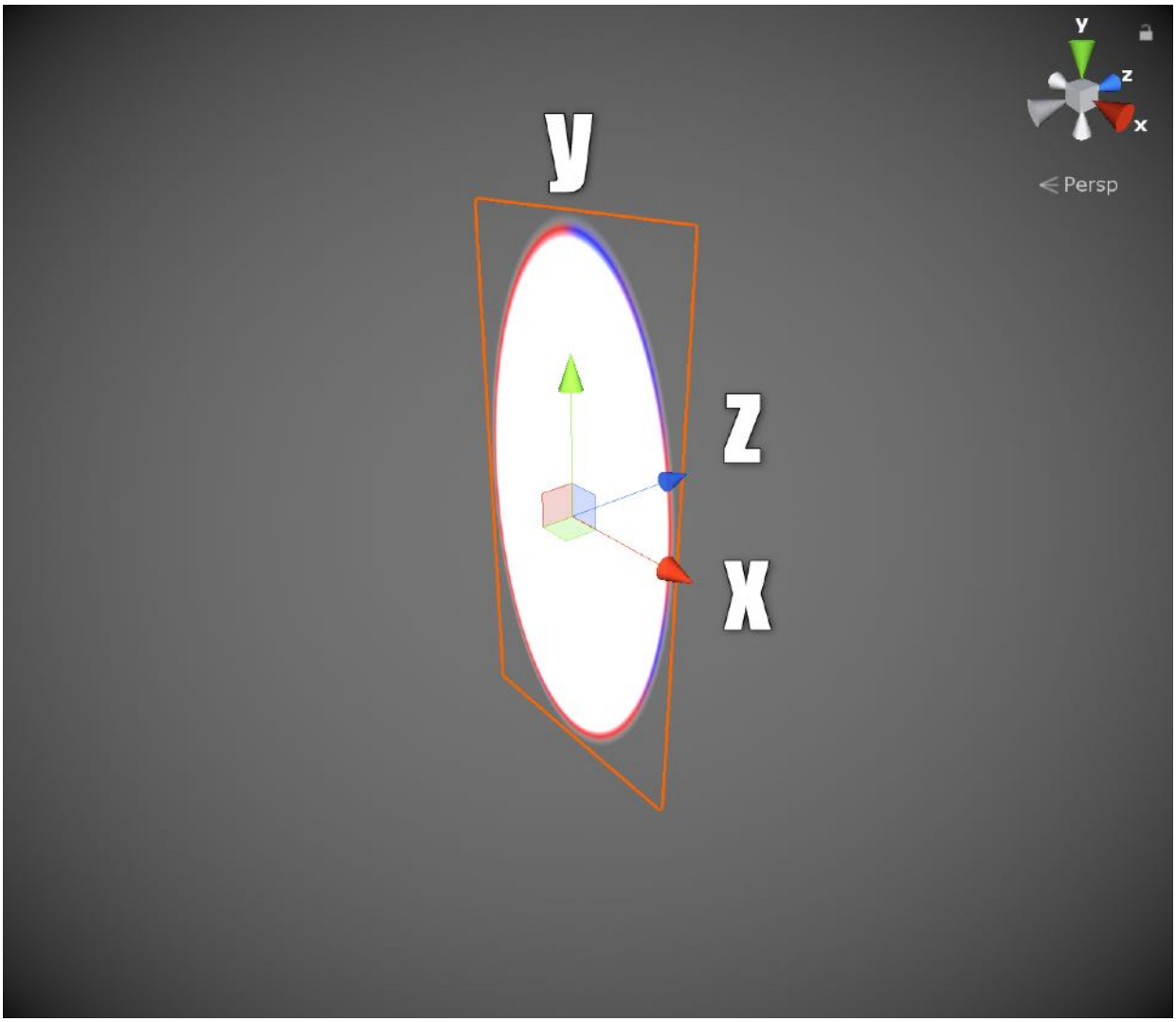

Figure 7.1: Gate coordinate system

The outbreak follows the original portal, so that when you click the mouse, the gate will appear on the plane of the viewpoint, and left-click and right-click will connect each other as a pair. If there is already a gate, the old gate will disappear on the spot and a new gate will open. Internally, the old gate has been moved to a new location and is the earliest.

PortalGun.cs

```
void Shot(int idx)
{
    RaycastHit hit;
    if (Physics.Raycast(transform.position,
        transform.forward,
        out hit,
        float.MaxValue,
        LayerMask.GetMask(new[] { "StageColl" })))
    {
        var gate = gatePair [idx];
        if (gate == null)
        {
            var go = Instantiate(gatePrefab);
              gate = gatePair[idx] = go.GetComponent<PortalGate>
();

            var pair = gatePair[(idx + 1) % 2];
            if (pair != null)
            {
                gate.SetPair(pair);
                pair.SetPair(gate);
            }
        }

        gate.hitColl = hit.collider;

        var trans = gate.transform;
        var normal = hit.normal;
                var up = normal.y >= 0f ? transform.up :
transform.forward;

        trans.position = hit.point + normal * gatePosOffset;
        trans.rotation = Quaternion.LookRotation(-normal, up);

        gate.Open();
    }
}
```

By specifying only the **StageColl** layer `transform.forward`, the ray is skipped in the direction and the hit is confirmed. If there is a hit, the gate operation is processed. First, check if there is an existing gate, and if not, generate it. Pairing is also done here. `PortalGate.hitColl`Set the collider that Ray collided with for later use , and ask for the position and orientation. The position is slightly lifted in the normal direction from the plane where it

collided, and Z-fighting measures are taken. Did you notice that the way to find the orientation is a little strange? The specification of the up vector of Quaternion.LookRotation () is changed by the positive or negative of normal.y. Normally, transform.up is fine, but when the gate is put out on the ceiling, the front and back (Y direction of PortalGate) will be reversed and it will feel strange, so I did it like this. I think the original Portal also behaved like this.

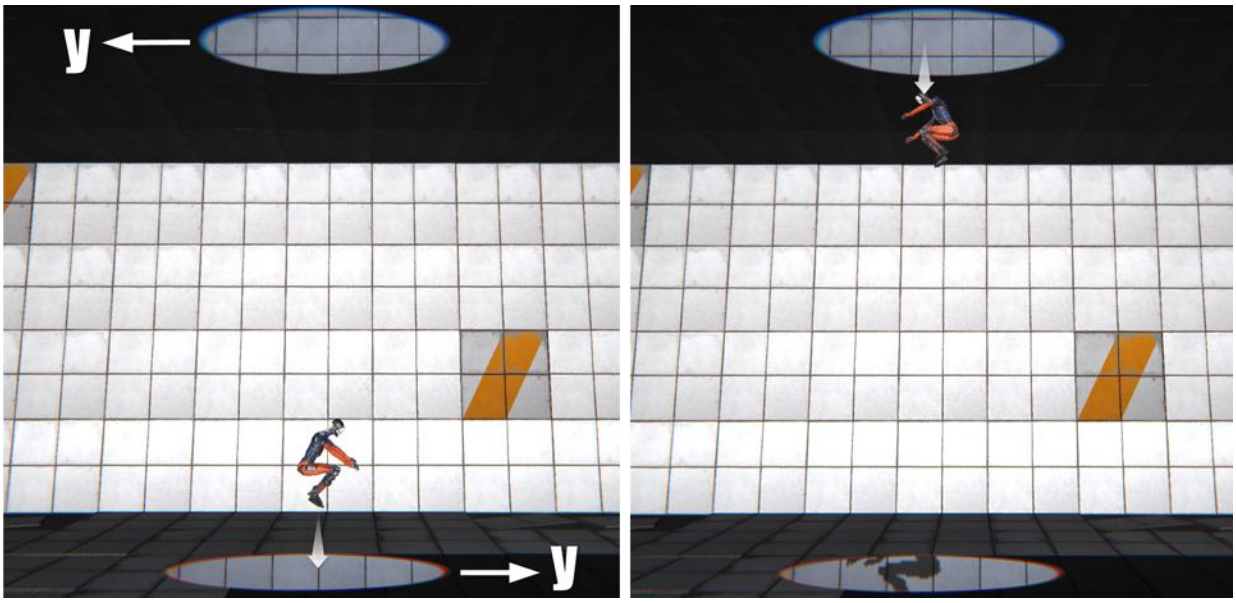

Figure 7.2: Without up-vector processing

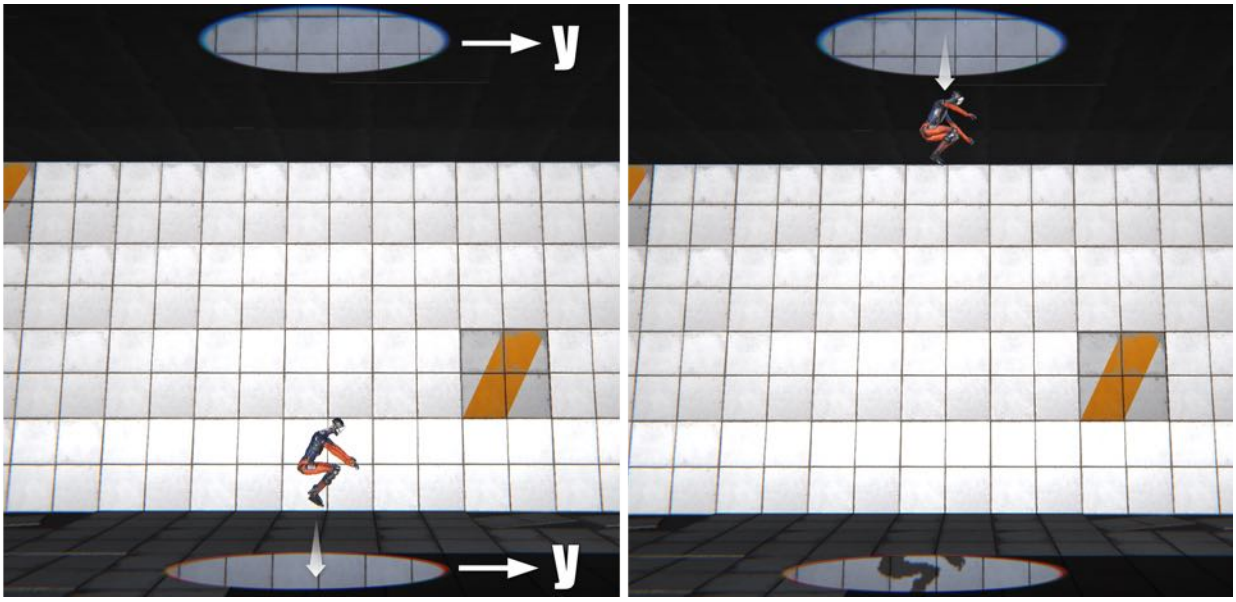

Figure 7.3: Up-vector processing

7.4 VirtualCamera

7.4.1 Initialization

When the gate opens, you can see the other side of another paired gate (hereinafter referred to as pair gate), so you need to implement this drawing somehow. I took the approach of "**preparing another camera (Virtual Camera), capturing it on the Render Texture, pasting it on the Portal Gate, and drawing with the main camera** " to draw the "other side" .

Virtual Camera is a camera that captures pictures on the other side of the gate.

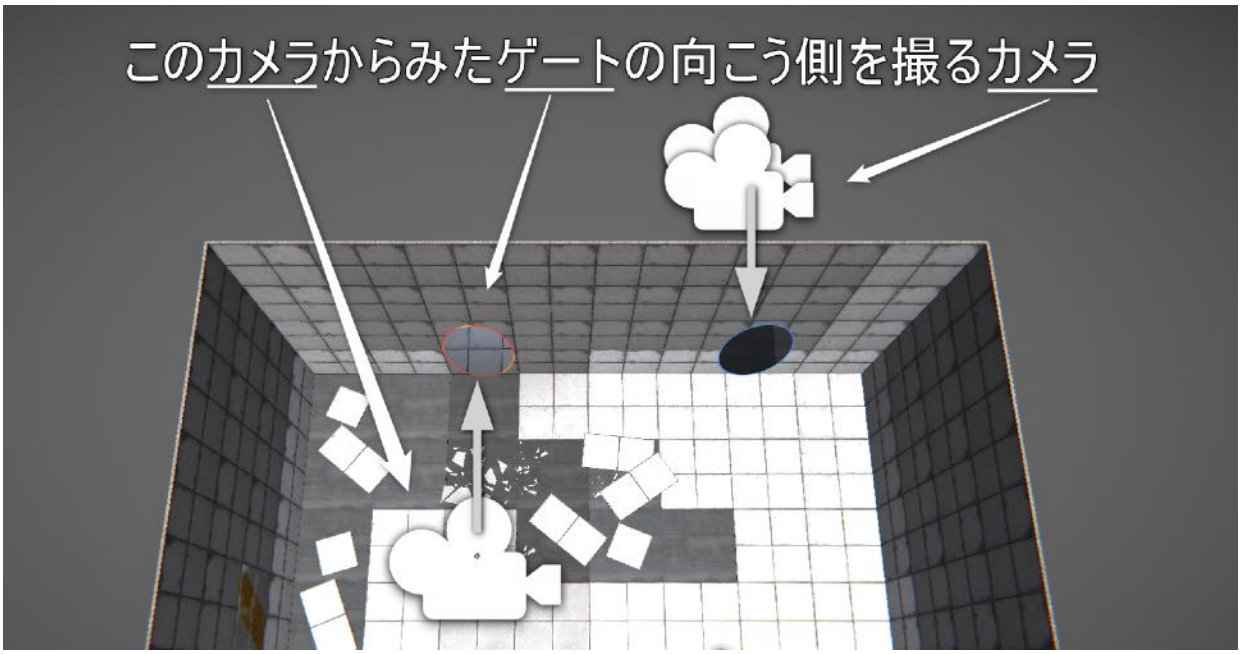

図 7.4: VirtualCamera

`PortalGate.OnWillRenderObject()`Is called for each camera, so if Virtual Camera is required at that timing, it will be generated.

PortalGate.cs

```
private void OnWillRenderObject ()
{
~ Omitted ~
    VirtualCamera pairVC;
    if (!pairVCTable.TryGetValue(cam, out pairVC))
    {
        if ((vc == null) || vc.generation < maxGeneration)
        {
            pairVC = pairVCTable[cam] = CreateVirtualCamera(cam,
vc);
            return;
        }
    }

~ Omitted ~
}
```

When the gates face each other, the gate is reflected in the scenery on the other side, and the gate is also on the other side of the gate.

Figure 7.5: Facing gates

in this case,

1. Prepare a picture of the other side of the gate reflected in the main camera Virtual Camera

2. Second generation Virtual Camera for the gate reflected in that Virtual Camera
3. Furthermore, the third generation Virtual Camera for the gate reflected in the Virtual Camera
4. further···

If you implement it honestly, you will need an infinite number of Virtual Cameras. This is not the case, so `PortalGate.maxGeneration`I will limit the number of generations, and although it is not an accurate picture, I will substitute it by pasting the texture one frame before to the gate.

PortalGate.cs

```
VirtualCamera        CreateVirtualCamera(Camera        parentCam,
VirtualCamera parentVC)
{
    var rootCam = parentVC?.rootCamera ?? parentCam;
    var generation = parentVC?.generation + 1 ?? 1;

    var go = Instantiate(virtualCameraPrefab);
    go.name = rootCam.name + "_virtual" + generation;
    go.transform.SetParent(transform);

    var vc = go.GetComponent<VirtualCamera>();
    vc.rootCamera = rootCam;
    vc.parentCamera = parentCam;
    vc.parentGate = this;
    vc.generation = generation;

    vc.Init ();

    return you;
}
```

`VirtualCamera.rootCamera`Is the main camera that dates back to the generation of Virtual Camera. In addition, the parent camera, target gate, generation, etc. are set.

VirtualCamera.cs

```
public void Init()
{
    camera_.aspect = rootCamera.aspect;
```

```
    camera_.fieldOfView = rootCamera.fieldOfView;
    camera_.nearClipPlane = rootCamera.nearClipPlane;
    camera_.farClipPlane = rootCamera.farClipPlane;
          camera_.cullingMask  |=  LayerMask.GetMask(new[]  {
PlayerLayerName });
    camera_.depth = parentCamera.depth - 1;

    camera_.targetTexture = tex0;
    currentTex0 = true;
}
```

`VirtualCamera.Init()`The parameters are inherited from the parent camera. Since my character is reflected in Virtual Camera, the **Player** layer is deleted from Culling Mask . Also, because you want to capture a picture earlier than the parent of the camera `parentCamera.depth - 1`has been.

`Camera.CopyFrom()`I used it at the beginning , but it seems that CommandBuffer is also copied, and an error occurred when using it together with PostProcessingStack * 5 used for post effect, so I copied it for each property.

[*5] https://github.com/Unity-Technologies/PostProcessing

7.4.2 update

VirtualCamera `PortalGate.maxGeneration`can do more as the processing is lighter, so I pay a little attention to performance so as not to waste processing.

VirtualCamera.cs

```
private void LateUpdate()
{
     // PreviewCamera etc. seems to be null at this timing, so
check
    if (parentCamera == null)
    {
        Destroy(gameObject);
        return;
    }

    camera_.enabled = parentGate.IsVisible(parentCamera);
    if (camera_.enabled)
```

```
    {
        var parentCamTrans = parentCamera.transform;
        var parentGateTrans = parentGate.transform;

        parentGate.UpdateTransformOnPair(
            transform,
            parentCamTrans.position,
            parentCamTrans.rotation
            );


        UpdateCamera();
    }
}
```

I will follow this code in detail.

Disable camera

If the parent camera does not show the gate, you do not need to prepare the picture on the other side, so disable the camera of Virtual Camera.

PortalGate.cs

```
public bool IsVisible(Camera camera)
{
    var ret = false;

    var pos = transform.position;
    var camPos = camera.transform.position;

    var camToGateDir = (pos - camPos).normalized;
    var dot = Vector3.Dot(camToGateDir, transform.forward);
    if (dot > 0f)
    {
                                                var     planes     =
GeometryUtility.CalculateFrustumPlanes(camera);
                  ret   =   GeometryUtility.TestPlanesAABB(planes,
coll.bounds);
    }

    return ret;
}
```

The visibility judgment is as follows.

1. Judgment of orientation. I'm checking if the gate is facing the camera. It is judged by the sign of the inner product of the camera → gate direction and the Z + direction of the gate.
2. Visibility judgment of the frustum and bounding box. Unity has a function for this, and you can use it as it is. Thank you.

Position and orientation updates

`parentGate.UpdateTransformOnPair()` In, "From the position and orientation of the parent camera with respect to the parent gate, find the position and orientation of the parent pair with respect to the gate and update the transform".

PortalGate.cs

```
public void UpdateTransformOnPair(
    Transform trans,
    Vector3 worldPos,
    Quaternion worldRot
    )
{
    var localPos = transform.InverseTransformPoint(worldPos);
       var  localRot  =  Quaternion.Inverse(transform.rotation)  *
worldRot;

    var pairGateTrans = pair.transform;
    var gateRot = pair.gateRot;
    var pos = pairGateTrans.TransformPoint(gateRot * localPos);
    var rot = pairGateTrans.rotation * gateRot * localRot;

    trans.SetPositionAndRotation(pos, rot);
}
```

The implementation looks like this,

1. Change to the local coordinate system of the gate
2. Change the direction from the front to the back of the gate with gateRot
3. Treated as local coordinates of paired gates
4. Convert to world coordinate system

It is the procedure. gateRot

```
public Quaternion gateRot { get; } = Quaternion.Euler(0f, 180f,
0f);
```

And, I rotate it 180 degrees on the Y axis, but since the Z value should be inverted

```
public Quaternion gateRot { get; } =  Quaterion.Euler(180f, 0f,
0f);
```

Even an implementation like this should not break down. However, since the upward direction is reversed between the front and the back of the gate, when you pass through the gate, your character's head will be on the ground side, which makes you feel uncomfortable, so Y-axis rotation seems to be good.

Update camera parameters

VirtualCamera.cs

```
void UpdateCamera()
{
    var pair = parentGate.pair;
    var pairTrans = pair.transform;
    var mesh = pair.GetComponent<MeshFilter>().sharedMesh;
    var vtxList = mesh.vertices
                                                .Select(vtx   =>
pairTrans.TransformPoint(vtx)).ToList();

    TargetCameraUtility.Update(camera_, vtxList);

    // Oblique
    // Draw only the back of pairGate = match nearClipPlane with
pairGate
    var pairGateTrans = parentGate.pair.transform;
    var clipPlane = CalcPlane(camera_,
                              pairGateTrans.position,
                              -pairGateTrans.forward);

                               camera_.projectionMatrix        =
camera_.CalculateObliqueMatrix(clipPlane);
}

Vector4 CalcPlane(Camera cam, Vector3 pos, Vector3 normal)
{
```

```
    var viewMat = cam.worldToCameraMatrix;

                                    var         normalOnView         =
viewMat.MultiplyVector(normal).normalized;
    var posOnView = viewMat.MultiplyPoint (pos);

    return new Vector4(
        normalOnView.x,
        normalOnView.y,
        normalOnView.z,
        -Vector3.Dot(normalOnView, posOnView)
        );
}
```

Virtual Camera wants to be as light as possible, so make the view frustum as narrow as possible. Since it is only necessary to draw the range of the pair gate seen through VirtualCamera, the vertices of the pair gate mesh are set to world coordinates, and the `TargetCameraUtility.Update()`view frustum is `Camera.rect`changed so that the vertices fit in .

Also, since the object between the Virtual Camera and the pair gate is not drawn, make the near clip surface of the camera the same plane as the pair gate. `Camera.CalculateObliqueMatrix()`You can do this with . Since there is not much documentation, it will be judged from the sample code etc., but it seems that the near clip plane is passed by Vector4 with the normal to xyz and the distance to w in the view coordinate system.

7.5 Gate drawing

What is drawn is different according to the state, but it is done with a single shader.

- There is a frame and a display of the contents
- Immediately after the gate is created and moved, the circle will expand.
- If there is no pair gate yet, the background (existing walls and floors) will be moody (Fig. 7.6).
- When the pair gate is completed, fade in to the picture of Virtual Camera from Moyamoya
- `PortalGate.maxGeneration`If you reach and there is no Virtual Camera, paste the picture one frame before to PortalGate.

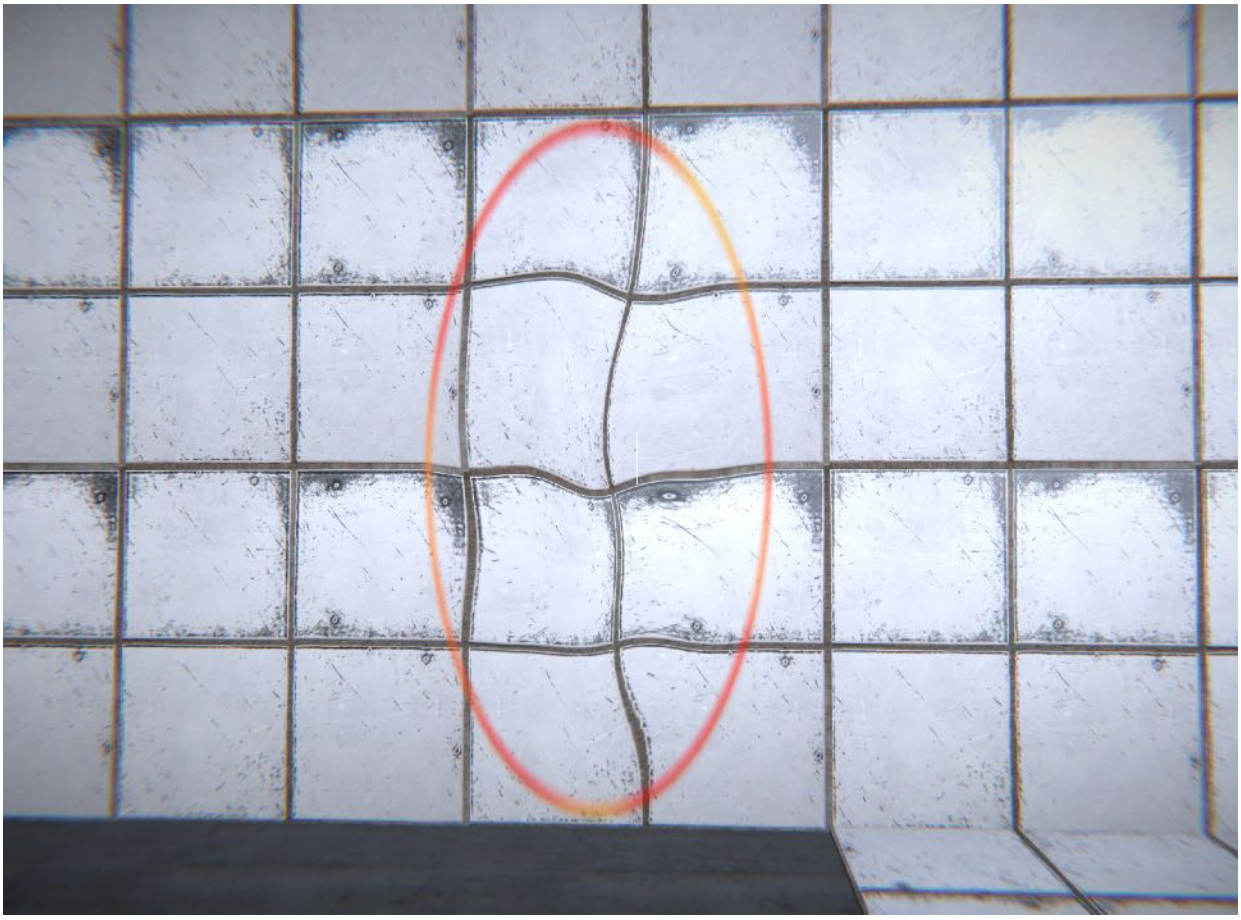

Figure 7.6: The background is moody when there is no pair gate

PortalGate.shader

```
GrabPass
{
    "_BackgroundTexture"
}
```

First , capture the background with GrabPass * 6 .

[*6] https://docs.unity3d.com/ja/current/Manual/SL-GrabPass.html

7.5.1 Vertex shader

PortalGate.shader

```
v2f vert(appdata_img In)
{
    v2f o;

                float3    posWorld    =    mul(unity_ObjectToWorld,
float4(In.vertex.xyz, 1)).xyz;
    float4 clipPos = mul(UNITY_MATRIX_VP, float4(posWorld, 1));
            float4   clipPosOnMain   =   mul(_MainCameraViewProj,
float4(posWorld, 1));

    o.pos = clipPos;
    o.uv = In.texcoord;
    o.sposOnMain = ComputeScreenPos(clipPosOnMain);
    o.grabPos = ComputeGrabScreenPos (o.pos);
    return o;
}
```

The vertex shader looks like this. We are looking for two positions in the screen coordinate system, one for the current camera and one for `clipPos`the main camera `clipPosOnMain`. The former is used for normal rendering, and the latter is used for referencing RenderTexture captured by Virtual Camera. Also, when using GrabPass, there is a dedicated position calculation function, so use this.

7.5.2 Fragment shader

PortalGate.shader

```
float2 uv = In.uv.xy;
uv = (uv - 0.5) * 2; // map 0~1 to -1~1
float insideRate = (1 - length(uv)) * _OpenRate;
```

`insideRate`(Inside ratio of the circle) is calculated. The center of the circle is 1, the circumference is 0, and the outside is negative. `_OpenRate`You can change the opening degree of the circle with. It is controlled by **PortalGate.Open ()** .

PortalGate.shader

```
// background
float4 grabUV = In.grabPos;
float2 grabOffset = float2(
    snoise(float3(uv, _Time.y     )),
```

```
    snoise(float3(uv, _Time.y + 10))
);
grabUV.xy += grabOffset * 0.3 * insideRate;
float4 bgColor = tex2Dproj(_BackgroundTexture, grabUV);
```

It is generating a moody background. `snoise`Is a function defined in the included Noise.cginc and is SimplexNoise. The grab UV is rocking with the uv value and time. By multiplying the insideRate, the fluctuation becomes larger toward the center.

PortalGate.shader

```
// portal other side
float2 sUV = In.sposOnMain.xy / In.sposOnMain.w;
float4 sideColor = tex2D(_MainTex, sUV);
```

It is a picture of the other side of the gate. `_MainTex`Contains the texture captured by the Virutual Camera and is referenced by the UV value of the main camera.

PortalGate.shader

```
// color
float4 col = lerp(bgColor, sideColor, _ConnectRate);
```

`bgColor``sideColor`I mix (walls and floors) and (beyond the gate). `_ConnectRate`Transitions from 0 to 1 when a pair gate is created and remains at 1 thereafter.

PortalGate.shader

```
// frame
float frame = smoothstep(0, 0.1, insideRate);
float frameColorRate = 1 - abs(frame - 0.5) * 2;
float mixRate = saturate(grabOffset.x + grabOffset.y);
float3 frameColor = lerp(_FrameColor0, _FrameColor1, mixRate);
col.xyz = lerp(col.xyz, frameColor, frameColorRate);

col.a = frame;
```

Finally, the frame is calculated. `insideRate`The edges of are `_FrameColor0`,`_FrameColor1`displayed by mixing them appropriately.

The appearance is completed so far. Next, let's focus on the physical behavior.

7.6 Object Warp

Changed to process around warp in **PortalObj component** . GameObjects with this will be able to warp.

7.6.1 Disable existing collisions

The plane on which the gate is installed cannot pass through, that is, there is a collision. This must be disabled when passing through the gate. Actually, the gate is equipped with a collider that pops out rather large in the front and back as a trigger. PortalObj uses this collider as a trigger to invalidate the collision with the plane.

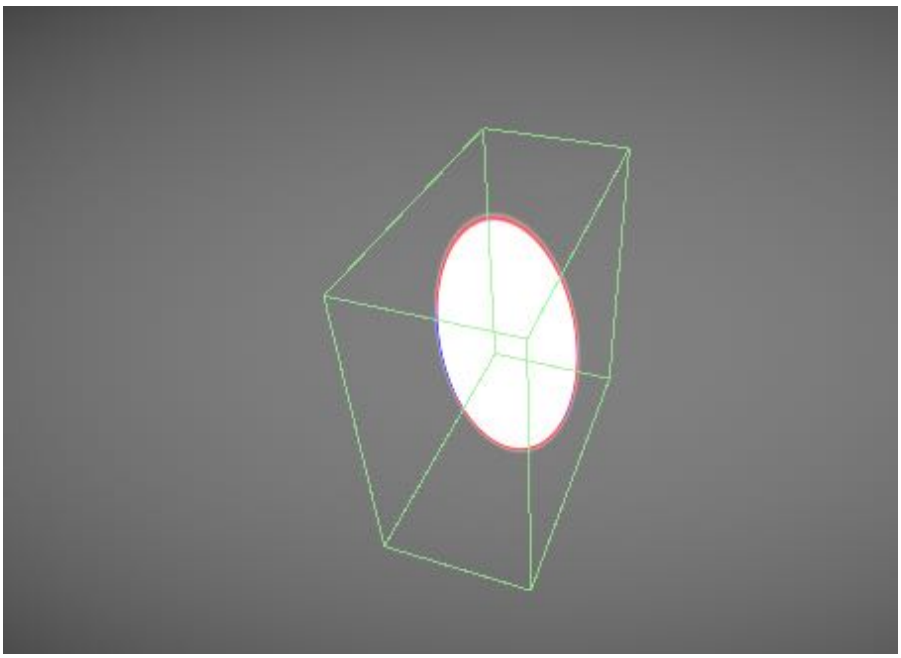

Figure 7.7: Gate Collider

PortalObj.cs

```
private void OnTriggerStay(Collider other)
{
    var gate = other.GetComponent<PortalGate>();
```

```
      if ((gate != null) && !touchingGates.Contains(gate) &&
(gate.pair != null))
    {
        touchingGates.Add(gate);
        Physics.IgnoreCollision(gate.hitColl, collider_, true);
    }
}

private void OnTriggerExit(Collider other)
{
    var gate = other.GetComponent<PortalGate>();
    if (gate != null)
    {
        touchingGates.Remove(gate);
        Physics.IgnoreCollision(gate.hitColl, collider_, false);
    }
}
```

`OnTriggerEnder()`The `OnTriggerStay()`reason for this is that if there is only one gate and there is no pair, Enter will be performed and then a pair will be created. First `tougingGates`, register the gate that triggered it in . The above `PortalGate.hitColl`is finally coming out. `Physics.IgnoreCollision()`Set this and your collider to ignore the collision with.

`OnTriggerExit()`The collision is enabled again with. As many of you may have noticed, since `PortalGate.hitColl`is a collider on the entire plane, it can actually pass through even outside the frame of the Portal Gate. The condition "as long as you keep OnTriggerStay ()" is attached, so it is not very noticeable, but it seems that a little more complicated processing is required to collide in the form of a proper gate.

7.6.2 Warp processing

PortalObj.cs

```
private void Update()
{
    var passedGate = touchingGates.FirstOrDefault(gate =>
    {
                                        var     posOnGate     =
gate.transform.InverseTransformPoint(center.position);
        return posOnGate.z > 0f;
```

```
    });


    if (passedGate != null)
    {
        PassGate(passedGate);
    }

    if ((rigidbody_ != null) && !rigidbody_.useGravity)
    {
                if  ((Time.time  -  ignoreGravityStartTime)    >
ignoreGravityTime)
        {
            rigidbody_.useGravity = true;
        }
    }
}
```

`center`Is a Transform used to determine if it has passed the gate. Basically, the GameObject with PortalObj component is fine, but I want to warp my character when the camera passes, not the center of the character, so I can set it manually. `center.position`We are checking `z > 0f`if there is a gate with (behind the gate) `touchingGates`. If such a gate is found `PassGate()`(warp processing).

Also, as will be described later, Portal Obj disables gravity immediately after passing through the gate. This is done to make the object behave with a little inertia after passing because if you open a gate that connects to another floor under the object that is falling on the ground, the object will vibrate back and forth between the gates. I have.

PortalObj.cs

```
void PassGate(PortalGate gate)
{
    gate.UpdateTransformOnPair(transform);

    if (rigidbody_ != null)
    {
                                          rigidbody_.velocity     =
gate.UpdateDirOnPair(rigidbody_.velocity);
        rigidbody_.useGravity = false;
        ignoreGravityStartTime = Time.time;
    }
```

```
    if (fpController != null)
    {
                                        fpController.m_MoveDir     =
gate.UpdateDirOnPair(fpController.m_MoveDir);
        fpController.InitMouseLook();
    }
}
```

The warp process looks like this. I also used it to find the position of the Virtual `PortalGate.UpdateTransformOnPair()`Camera and warp the Transform. `RigidBody`If you have, change the direction of speed as well. `fpController`The same applies to (script for own character operation). As this area becomes larger, there will be objects that need more support, so it may be better to prepare each script callback and notify it.

7.6.3 Warp issues

There was a point that I had to implement a warp this time and pack some more.

PortalObj hits the wall once when the speed is fast

I wanted to somehow nullify the collision after the physics engine made a collision detection and before extrusion, but I couldn't find a good way. `OnTriggerEnter()`, `OnCollisionEnter()`The inner `Physics.IgnoreCollision()`seems to be referred to are disabled from after a collision once. I think `On~Enter()`it `Physics.IgnoreCollision()`'s probably a little late to reflect what is called after extrusion . For this reason, the range of the trigger is made to protrude considerably so that the frame that enters the trigger and the frame that collides with the wall are different. However, this method has its limitations and is not compatible with Portal Obj, which moves at a higher speed. If anyone says "There is such a way! ", Please contact me!

Actually it is better to put a copy in the middle

I implemented the warp by **"rewriting the position of the object"** , but strictly speaking, there should be a state where it is half in front and half

behind while passing through the gate. If you want to put out a large object, it will be noticeable, so you need to think about this as well. In addition, it needs to be affected by collisions both in front and behind, and more strictly, I feel that we have to intervene in the solver in the physics engine. It seems to be strict with Unity, so I feel that it is realistic to cheat well.

7.7 Summary

I tried to reproduce Portal that I wanted to try from before with Unity. I tried it comfortably for the first time by stacking the cameras, but I found that it was more difficult than I expected. Among CG and game technologies, those that are closer to the real world are in high demand and are becoming more and more standardized. When it becomes easier to create a sense of reality, Anywhere Door-like "ideas that used to be common but unrealistic and slept" may come to life as a new experience.

7.8 Reference

- Portal http://www.thinkwithportals.com/
- Adam Character Pack https://assetstore.unity.com/packages/essentials/tutorial-projects/adam-character-pack-adam-guard-lu-74842
- playGROWnd https://github.com/unity3d-jp/playgrownd
- PostProcessingStack https://github.com/Unity-Technologies/PostProcessing

Chapter 8 Easily express soft deformation

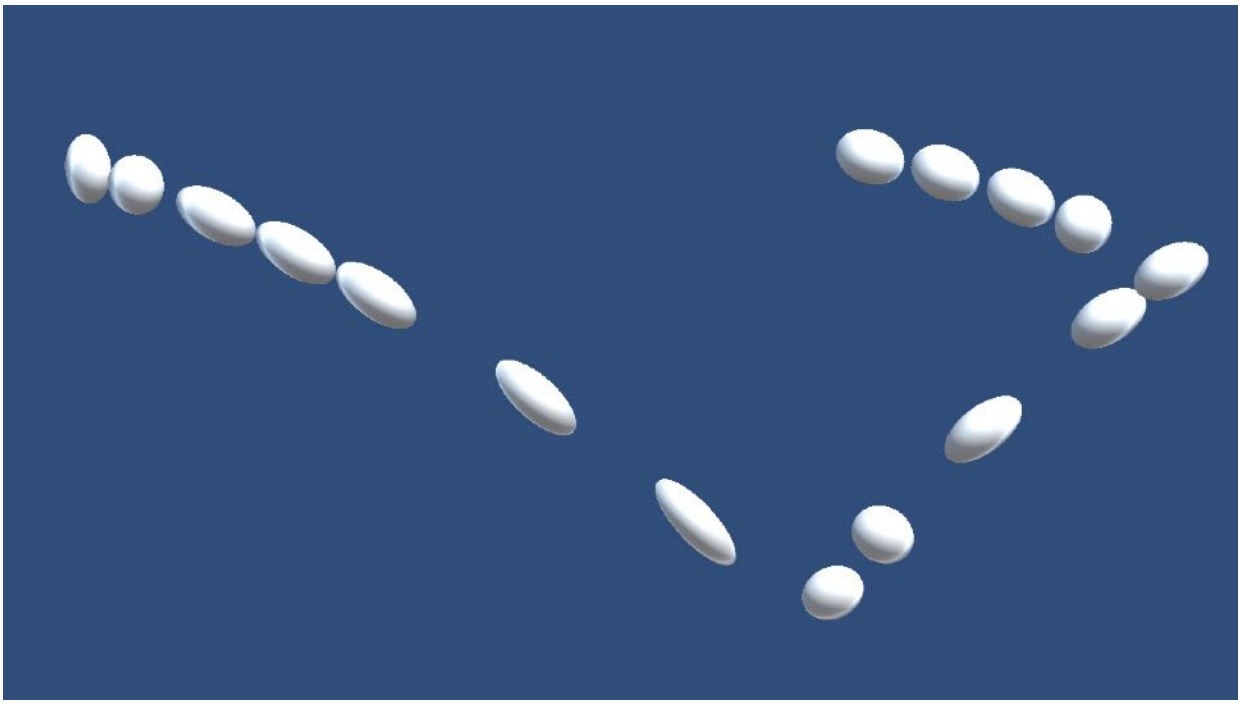

Figure 8.1: Sphere deforms

When expressing the softness of an object, we sometimes imitate a spring or calculate a simulation of a fluid or soft body, but here we do not make such an exaggerated calculation, but express the soft deformation of the object. to watch. As shown in the figure, it is a transformation like a hand-drawn animation.

The sample in this chapter is "Over Reaction" from
https://github.com/IndieVisualLab/UnityGraphicsProgramming3
.

8.1 How to move the sample scene

In the "Over Reaction" scene, you can see the basic transformation. Watch the object move and transform from the manipulator or Inspector.

In the "Physics Scene" scene, you can apply force to the object with the up, down, left, and right keys. If you place some objects on the scene, you can see how they transform depending on the situation.

8.2 Calculation of kinetic energy

There are many possible transformation rules, but there are probably only three basic rules.

1. The larger the change, the greater the deformation.
2. When there is no change, the deformation gradually returns.
3. When the direction of change is reversed, the direction of deformation is also reversed.

Here, we will especially consider when the object moves, so first we will detect the direction and magnitude of the movement. Although different from the term used in the laws of physics, this parameter `moveEnergy`is called "kinetic energy" for convenience . Since kinetic energy is a parameter expressed by direction and magnitude, it can be expressed by a vector.

* Kinetic energy is the correct physics term. Here, it is named "move energy" because it is an energy that considers only movement.

This is not the case as it is treated as a matter of course in game programming, but the movement of an object simply detects a change in coordinates. The only thing I want to note is that `Update`it uses instead `FixedUpdate`.

OverReaction.cs

```
protected void FixedUpdate()
{
    this.crntMove = this.transform.position - this.prevPosition;

    UpdateMoveEnergy();
    UpdateDeformEnergy();
```

```
    DeformMesh();

    this.prevPosition = this.transform.position;
    this.prevMove = this.crntMove;
}
```

`FixedUpdate`Is a method that is called at regular intervals, so it's `Update`clearly different in nature from being called twice or three times per second . I will omit the details of these differences because it is out of the main subject, but `FixedUpdate`I adopted it here because I want to support the movement of objects using PhysX (physical behavior) of Unity . There `Update`is no particular need to deform the mesh as often as.

Now that the movement of the object can be calculated from the change in coordinates, let's calculate the kinetic energy. The calculation of kinetic energy is `UpdateMoveEnergy`implemented in the method.

OverReaction.cs

```
protected void UpdateMoveEnergy()
{
    this.moveEnergy = new Vector3()
    {
        x = UpdateMoveEnergy
        (this.crntMove.x, this.prevMove.x, this.moveEnergy.x),

        y = UpdateMoveEnergy
        (this.crntMove.y, this.prevMove.y, this.moveEnergy.y),

        z = UpdateMoveEnergy
        (this.crntMove.z, this.prevMove.z, this.moveEnergy.z),
    };
}
```

Kinetic energy is calculated by decomposing into each component in the X, Y, and Z directions. The following `UpdateMoveEnergy`processes will be explained step by step.

First, consider the case where there is no current movement. When there is no movement, the existing kinetic energy decays.

OverReaction.cs

```
protected float UpdateMoveEnergy
(float crntMove, float prevMove, float moveEnergy)
{
    int crntMoveSign = Sign(crntMove);
    int prevMoveSign = Sign(prevMove);
    int moveEnergySign = Sign(moveEnergy);

    if (crntMoveSign == 0)
    {
        return moveEnergy * this.undeformPower;
    }
…
}

public static int Sign(float value)
{
    return value == 0 ? 0 : (value > 0 ? 1 : -1);
}
```

When the current movement and the previous movement are reversed, the kinetic energy is reversed.

OverReaction.cs

```
if (crntMoveSign != prevMoveSign)
{
    return moveEnergy - crntMove;
}
```

When the current movement and the kinetic energy are reversed, reduce the kinetic energy.

OverReaction.cs

```
if (crntMoveSign != moveEnergySign)
{
    return moveEnergy + crntMove;
}
```

In cases other than the above, when the current movement and the kinetic energy are in the same direction, the current movement and the existing kinetic energy are compared and the larger one is adopted.

However, the kinetic energy decays and becomes smaller. In addition, the new kinetic energy generated by the current movement is increased by multiplying it by any parameter so that it can easily produce deformation.

OverReaction.cs

```
if (crntMoveSign < 0)
{
    return Mathf.Min(crntMove * this.deformPower,
                     moveEnergy * this.undeformPower);
}
else
{
    return Mathf.Max(crntMove * this.deformPower,
                     moveEnergy * this.undeformPower);
}
```

With this, the kinetic energy to be used for deformation could be calculated.

8.3 Calculation of deformation energy

It then converts the calculated kinetic energy into parameters that determine the deformation. For convenience, this parameter `deformEnergy`is called "deformation energy" . The deformation energy is `UpdateDeformEnergy`updated by the method.

The magnitude of deformation energy can be defined as the magnitude of kinetic energy as it is, but if there is a discrepancy between the direction of deformation energy and the direction in which the object is moving, the deformation energy is completely in the object. I can't tell. It is also possible that the direction of deformation energy and the direction in which the object is moving are reversed.

Therefore, the amount of deformation energy transmitted is calculated from the inner product of the deformation energy and the current movement. If the directions are exactly the same, the inner product of the unit vectors will be 1, and will gradually approach 0 depending on the magnitude of the deviation. When it is further inverted, it becomes a negative value.

OverReaction.cs

```
protected void UpdateDeformEnergy()
{
    float deformEnergyVertical
    = this.moveEnergy.magnitude
    * Vector3.Dot(this.moveEnergy.normalized,
                  this.crntMove.normalized);
    …
```

Now that we have calculated the force that deforms the object in the vertical direction, it deforms in the horizontal direction by the amount of change in the vertical direction. In other words, if the object stretches vertically, it will shrink horizontally. On the contrary, when it shrinks in the vertical direction, it should stretch in the horizontal direction.

The amount of deformation in the vertical direction is calculated by "the magnitude of the deformation in the vertical direction / the maximum magnitude of the deformation". After that, it is calculated so that it deforms in the horizontal direction as much as it deforms in the vertical direction.

Assuming that the deformation is +0.8 in the vertical direction, the deformation should be -0.8 in the horizontal direction, so the deformation energy in the horizontal direction is 1-0.8 = 0.2. Also, as the coefficient for actual deformation, * 0.8 is small, so add 1 to make it * 1.8.

OverReaction.cs

```
protected void UpdateDeformEnergy()
{
…
    float deformEnergyHorizontalRatio
    = deformEnergyVertical / this.maxDeformScale;

    float deformEnergyHorizontal
    = 1 - deformEnergyHorizontalRatio;
…
    deformEnergyVertical = 1 + deformEnergyVertical;
}
```

Finally, consider the case where the object collapses in the direction of travel. The case where the object collapses in the direction of travel is when the kinetic energy and the current movement are reversed, that is, when the

previous "inner product of the kinetic energy and the current movement" is negative.

When the value of the inner product is negative, the deformation energy in the vertical direction and the deformation energy in the horizontal direction are reversed.

To help you understand the case, the following code is a continuation of the previous steps. When the dot product value is negative, `deformEnergyHorizontal`is a positive value greater than 1. Or `deformEnergyVertical`is `deformEnergyHorizontal`of value reversed with, it will be one less than the positive value.

OverReaction.cs

```
protected void UpdateDeformEnergy()
{
    float deformEnergyVertical
    = this.moveEnergy.magnitude
    * Vector3.Dot(this.moveEnergy.normalized,
                  this.crntMove.normalized);

    float deformEnergyHorizontalRatio
    = deformEnergyVertical / this.maxDeformScale;

    float deformEnergyHorizontal
    = 1 - deformEnergyHorizontalRatio;

    if (deformEnergyVertical < 0)
    {
        deformEnergyVertical = deformEnergyHorizontalRatio;
    }

    deformEnergyVertical = 1 + deformEnergyVertical;
…
```

Finally, correct the value so that the deformation energy falls within the arbitrarily set range, and complete the calculation of the deformation energy.

OverReaction.cs

```
deformEnergyVertical = Mathf.Clamp(deformEnergyVertical,
                                   this.minDeformScale,
```

```
                                        this.maxDeformScale);

deformEnergyHorizontal = Mathf.Clamp(deformEnergyHorizontal,
                                            this.minDeformScale,
                                            this.maxDeformScale);

this.deformEnergy = new Vector3(deformEnergyHorizontal,
                                    deformEnergyVertical,
                                    deformEnergyHorizontal);
```

8.4 Transform the mesh

Here, the mesh is transformed with a script for explanation and generalization. The transformation of the mesh is `DeformMesh`implemented in the method.

* Since it is a matrix operation, it is often better to process it on the GPU using a shader for practical purposes.

The obtained deformation energy `deformEnergy`is `moveEnergy`a vector representing expansion and contraction when facing the direction of kinetic energy . Therefore, when transforming, it is necessary to match the coordinates before transforming. First, suppress the parameters required for that purpose. The rotation matrix of the current object and its inverse matrix, the kinetic energy `moveEnergy`rotation matrix and its inverse matrix.

OverReaction.cs

```
protected void DeformMesh()
{
Vector3[]            deformedVertices            =            new
Vector3[this.baseVertices.Length];

Quaternion crntRotation  = this.transform.localRotation;
Quaternion crntRotationI = Quaternion.Inverse(crntRotation);

Quaternion moveEnergyRotation
=                          Quaternion.FromToRotation(Vector3.up,
this.moveEnergy.normalized);
Quaternion                 moveEnergyRotationI                  =
Quaternion.Inverse(moveEnergyRotation);
…
```

1. Multiplies the current rotation matrix by the vertices of the unrotated mesh to rotate.
2. Multiply the vertices by the inverse matrix of the rotation matrix that indicates the direction of movement and rotate.
3. Scales `deformEnergy`according to the vertices .
4. Multiplies the vertices by the rotation matrix that indicates the direction of movement to restore the rotation.
5. Multiplies the vertices by the inverse of the current rotation matrix to restore the rotation.

`deformEnergy`If you give an appropriate deformation energy in an easy-to-understand manner and comment out the source code in sequence, the processing procedure will be easier to understand.

OverReaction.cs

```
for (int i = 0; i < this.baseVertices.Length; i++)
{
    deformedVertices[i] = this.baseVertices[i];
    deformedVertices[i] = crntRotation * deformedVertices[i];
              deformedVertices[i]    =    moveEnergyRotationI    *
deformedVertices[i];
    deformedVertices[i] = new Vector3(
        deformedVertices[i].x * this.deformEnergy.x,
        deformedVertices[i].y * this.deformEnergy.y,
        deformedVertices[i].z * this.deformEnergy.z);
               deformedVertices[i]    =    moveEnergyRotation    *
deformedVertices[i];
    deformedVertices[i] = crntRotationI * deformedVertices[i];
}

this.baseMesh.vertices = deformedVertices;
```

8.5 Summary

I was able to transform the object with a very simple implementation. Although it is so easy to implement, the impression it gives to the appearance changes greatly.

Although the calculation cost is higher, it is possible to support the rotation and enlargement / reduction of the object, move the center of gravity of the

deformation, and support the skin mesh animation as an advanced form.

About the author

Chapter 1 Baking Skinned Animation to Texture-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 2 Gravitational N-Body Simulation / @ kodai100

A student freelance engineer who likes physics. I am addicted to the TA business of VFX production, interactive engineering, and VR related business. Please feel free to contact DM on Twitter!

- https://twitter.com/kodai100_tw
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

Chapter 3 Screen Space Fluid Rendering --Hiroaki Oishi / @irishoak

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

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

Chapter 4 GPU-Based Cellular Growth Simulation-Janda 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 5 Reaction Diffusion-@ kaiware007

An interactive engineer who works in an atmosphere. 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 6 Strange Attractor-Sada Yoshiaki / @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 I implemented Portal in Unity --Hidekazu Fukunaga / @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 8 Easily Expressing Soft Deformations-@XJINE

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

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