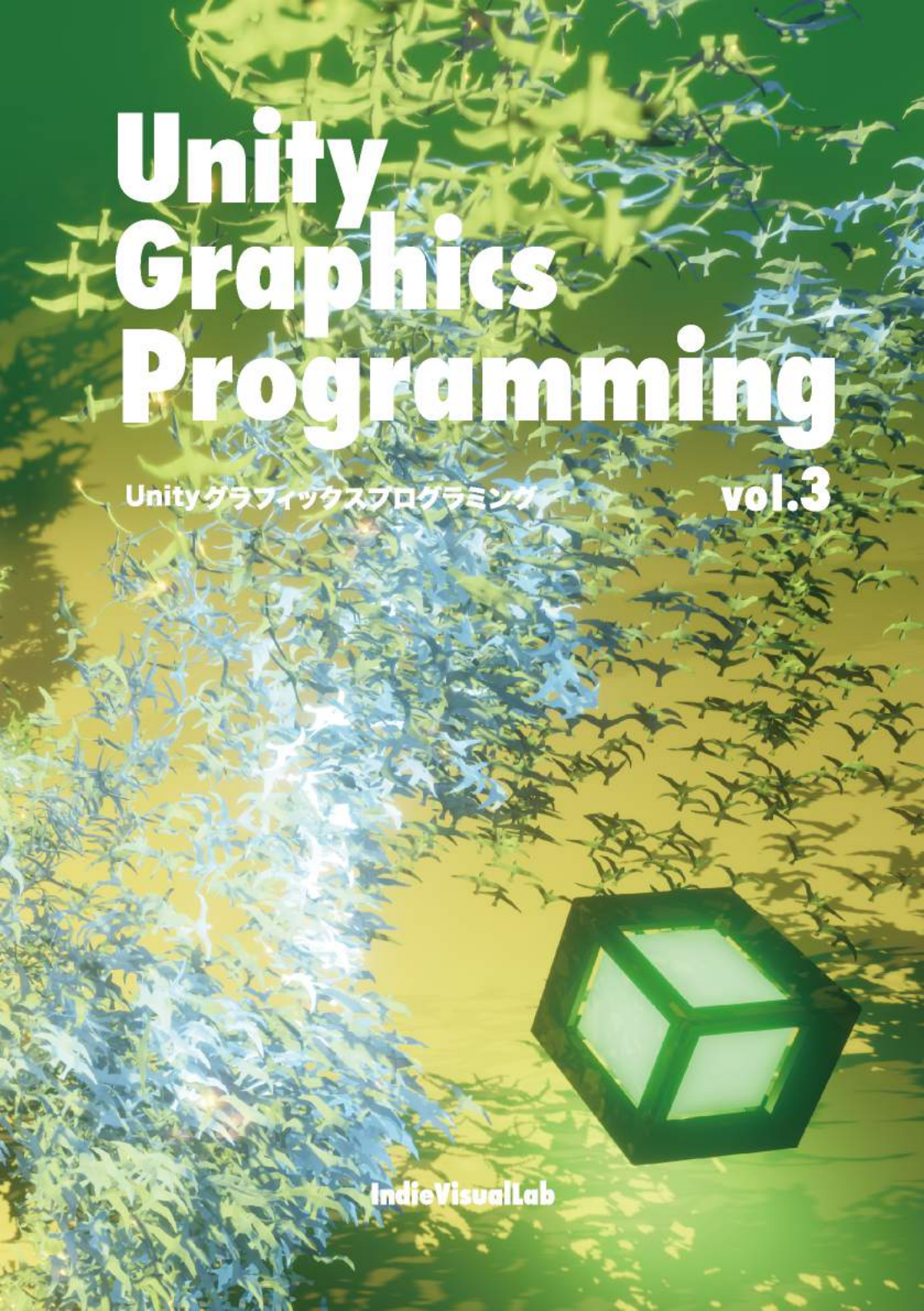

Unity
Graphics
Programming
Unityグラフィックスプログラミング
vol.3
IndieVisualLab

Unity Graphics Programming vol.3

IndieVisualLab　著

2018-10-08 版　　**IndieVisualLab** 発行

まえがき

本書は Unity によるグラフィクスプログラミングに関する技術を解説する「Unity グラフィックスプログラミング」シリーズの第三巻です。本シリーズでは、執筆者たちの興味の赴くままに取り上げられた様々なトピックについて、初心者向けの入門的な内容と応用を解説したり、中級者以上向けの tips を掲載しています。

各章で解説されているソースコードについては github リポジトリ (https://github.com/IndieVisualLab/UnityGraphicsProgramming3) にて公開していますので、手元で実行しながら本書を読み進めることができます。

記事によって難易度は様々で、読者の知識量によっては、物足りなかったり、難しすぎる内容のものがあるかと思います。自分の知識量に応じて、気になったトピックの記事を読むのが良いでしょう。普段仕事でグラフィクスプログラミングを行っている人にとって、エフェクトの引き出しを増やすことにつながれば幸いですし、学生の方でビジュアルコーディングに興味があり、Processing や openFrameworks などは触ったことはあるが、まだまだ 3DCG に高い敷居を感じている方にとっては、Unity を導入として 3DCG での表現力の高さや開発の取っ掛かりを知る機会になれば嬉しいです。

IndieVisualLab は、会社の同僚 (&元同僚) たちによって立ち上げられたサークルです。社内では Unity を使って、一般的にメディアアートと呼ばれる部類の展示作品のコンテンツプログラミングをやっており、ゲーム系とはまた一味違った Unity の活用をしています。本書の中にも節々に展示作品の中で Unity を活用する際に役立つ知識が散りばめられているかもしれません。

推奨する実行環境

本書で解説する内容の中には Compute Shader や Geometry Shader 等を使ったものがあり、DirectX11 が動作する実行環境を推奨しますが、CPU 側のプログラム (C#) で内容が完結する章もあります。

環境の違いによって公開しているサンプルコードの挙動が正しくならない、といったことが起こり得るかと思いますが、github リポジトリへの issue 報告、適宜読み替える、等の対処をお願いします。

本についての要望や感想

本書についての感想や気になった点、その他要望（○○についての解説が読みたい等）がありましたら、ぜひ Web フォーム（https://docs.google.com/forms/d/e/1FAIpQLSdxeansJvQGTWfZTBN_2RTuCK_kRqhA6QHTZKVXHCijQnC8zw/viewform）、またはメール（lab.indievisual@gmail.com）よりお知らせください。

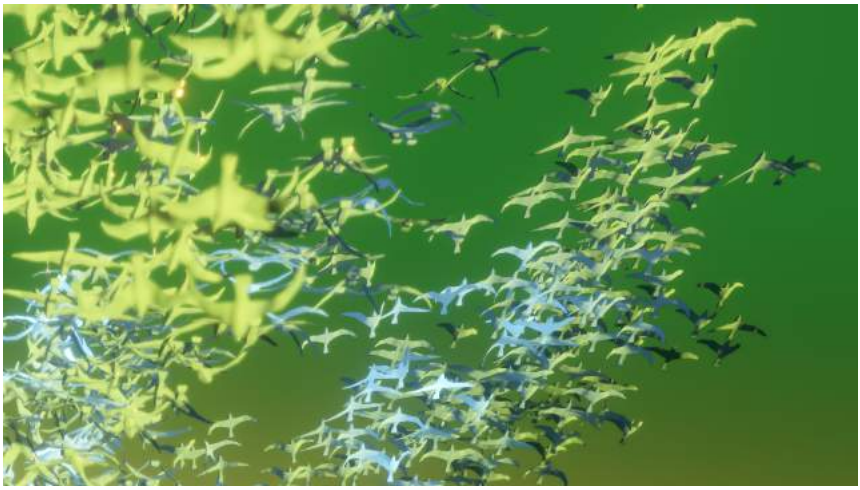
目次

第 1 章

Baking Skinned Animation to Texture

1.1　はじめに

　こんにちは、すぎのです！ この章では、数千、数万のスキニングアニメーションされたオブジェクトを表示します。



▲図 1.1　羽ばたきアニメーションをする鳥の群れ

　Unity では、キャラクターアニメーションを実現するとき、Animator コンポーネントと SkinnedMeshRenderer コンポーネントを使うことになると思います。

たとえば、鳥の群れや群衆を表現したいとき、どうするでしょう？ 数千、数万のキャラクターオブジェクトに対して Animator と SkinnedMeshRenderer を使用するでしょうか。一般的に、大量のオブジェクトを画面上に表示するとき、GPU インスタンシングを使用し、一度にまとめて大量のオブジェクトをレンダリングします。しかし、SkinnedMeshRenderer はインスタンシングをサポートしておらず、個別のオブジェクトを 1 つづつレンダリング処理することになり、処理が非常に重たくなってしまいます。

これを解決するソリューションとして、アニメーションされた頂点位置情報をテクスチャとして保存する方法があるのですが、実際にどういう方法か、実装までの考え方と応用、注意すべき点についてこの章では解説していきます。

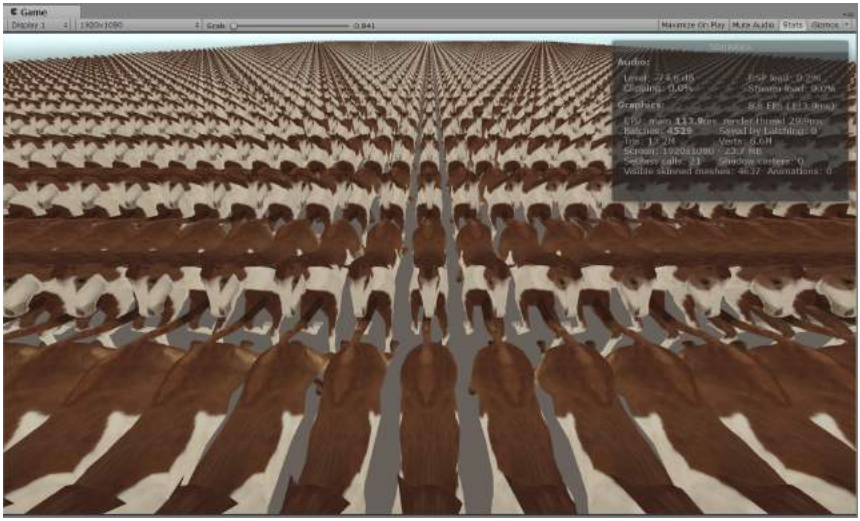
説明を省いていたり、分かりにくいところもあるかと思いますので、お気軽に Twitter（@sugi_cho 宛）でご質問ください。もし、間違っているところとかあったら、ご指摘いただけると助かります (._.)

1.2 アニメーションする SkinedMeshRenderer を 5000 体置いてみる

まずは、普通にアニメーションするオブジェクトを大量に（5000 体）置いたらどれくらい処理が重いのか、見ていきたいと思います。今回は、頂点数 1890 の、シンプルなアニメーション付き馬の 3D オブジェクトを用意しました。

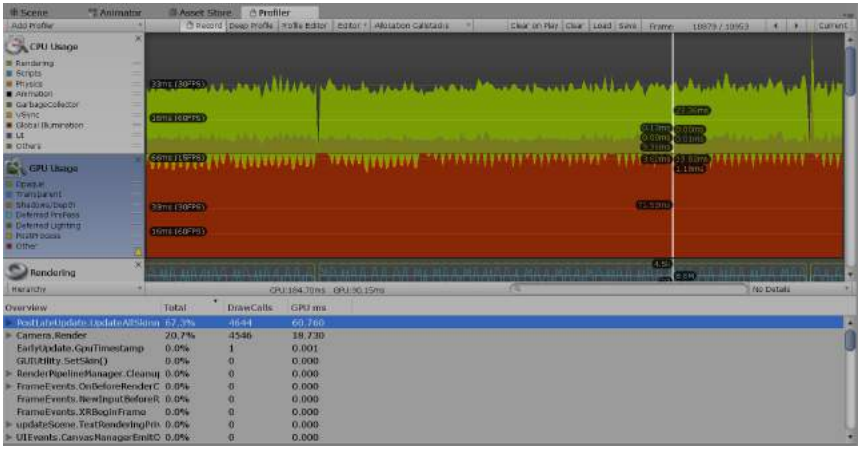
▲図 1.2　使用した馬のモデル

実際に動かしてみたところ、FPS は 8.8、かなり処理が重くなっていることがわかります。図 1.3



▲図 1.3　5000 頭のアニメーションする馬

では、この処理の中でどこが重くなっているのか、Unity のプロファイラを見て探っていくことにします。Window メニューから Profiler（ショートカットキー: Ctr+7）を表示します。Add Profiler のプルダウンから、GPU を選択し、GPU Usage のプロファイラを表示するとより詳細な情報を得られます。GPU Usage の情報を取得すること自体がオーバーヘッドになるので、必要ない時は表示しない方がいいのですが、今回は GPU Usage が重要になってくるので積極的に使用します。



▲図 1.4　Profiler Window (GPU Usage)

プロファイラを見ると、CPU の処理時間よりも GPU の処理時間の方が多く、CPU では GPU の処理完了待ちが発生していることが分かります。図 1.4 そして、GPU 処理の約 7 割を `PostLateUpdate.UpdateAllSkinnedMeshes`が占めていることが分かります。また、見えている馬のオブジェクト数ぶん、`Camera.Renderer`が走っているので、オブジェクトのバッチングか GPU インスタンシングを行うことにより、GPU のレンダリング処理数を抑えることができそうです。ちなみにですが、CPU Usage でも同じように `PostLateUpdate.UpdateAllSkinnedMeshes`と `Camera.Render`の処理がほとんどを占めています。

このテストシーンでは、PlayerSettings で GPU Skinning を使用する設定になっています。もしも GPU でなく CPU でスキニングを行っていた場合、CPU の処理の割合が増大し、今よりも FPS が落ちることになります。GPU スキニング時は、CPU 側でボーンの行列の計算を行い、行列情報を GPU に渡し、GPU でスキニング処理を行います。CPU スキニング時は、行列の計算とスキニングの処理まで CPU 側で行い、スキニングされた頂点データを GPU 側に渡しています。

このように、処理の最適化には最初にどこが処理のボトルネックになっているか、見極めることが重要になってきます。

1.3 SkinnedMeshRenderer のアニメーションする頂点の位置をあらかじめ計算しておく

プロファイリングの結果、メッシュのスキニングの処理が重そうだ。ということが分かったので、スキニングの処理自体はリアルタイムでは行わずに、あらかじめ計算しておく方法を検討してみようと思います。

SkinnedMeshRendererのスキニング処理された後の頂点情報を取得する方法としては、SkinnedMeshRenderer.BakeMesh(Mesh)という関数があります。これは、スキニングされた状態のメッシュのスナップショットを作成し、指定したメッシュに格納します。少々、処理に時間がかかるのですが、スキニングされた頂点情報を事前に格納しておくという使い方のためなら、選択可能な方法といえます。

▼リスト 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:     //mesh 内に、スキニングされたメッシュのスナップショットが格納される
16:     meshList.Add(mesh);
17: }
```

これで、Animator の Run ステートのアニメーションの SkinnedMeshRenderer の毎フレームのスナップショットの Mesh が meshList に格納されていきます。リスト 1.1

この保存された meshListを使い、パラパラ漫画で絵を切り替える要領でにメッシュ（MeshFilter.sharedMesh）を切り替えていくと、SkinnedMeshRendererを使用せずにメッシュのアニメーションが表示できるので、プロファイリングした結果ボトルネックになっていたスキニングの処理を省くことができそうです。

1.4 頂点情報を保存する方法の検討

しかし、この Mesh データをフレームごとに複数保存しておく実装だとアニメーションによって変更しないメッシュの情報（Mesh.indeces, Mesh.uv 等）も保存することになり、無駄が多くなります。スキニングアニメーションの場合、更新されるデータは頂点位置の情報と法線情報だけなので、これらのみを保存し、更新していけばいいのです。

1.4.1 頂点情報を Vector3 配列で保存しておく方法

そこで考えられるのが、フレームごとの頂点位置と法線データを Vector3 の配列で持っておいて、フレームごとにメッシュの位置と法線を更新していく方法です。リスト 1.2

▼リスト 1.2　Update Mesh

```
 1: Mesh objMesh;
 2: List<Vector3>[] vertecesLists;
 3: List<Vector3>[] normalsLists;
 4: //保存しておいた頂点の情報
 5: //Mesh.SetVertices(List<Vector3>) で使うため
 6:
 7: void Start(){
 8:     objMesh = GetComponent<MeshFilter>().mesh;
 9:     objMesh.MarkDynamic();
10: }
11:
12: void Update(){
13:     var frame = xx;
14:     //現在時刻でのフレームを計算する
15:
16:     objMesh.SetVertices(vertecesLists[frame]);
17:     objMesh.SetNormals(normalsLists[frame]);
18: }
```

しかし、この方法だと今解決しようとしている数千のアニメーションオブジェクトを表示するという目的に対して、メッシュの更新自体の CPU の処理負荷が大きくなってしまいます。

そこで、この章の冒頭から答えは書いてあるのですが、テクスチャに位置情報と法線情報を格納し、VertexTextureFetch を用いて頂点シェーダにおいてメッシュの頂点位置と法線情報を更新します。これにより元のメッシュデータ自体の更新は行う必要がなくなるので CPU の処理負荷無しで頂点アニメーションを実現可能になります。

1.4.2 位置情報をテクスチャに書き込む

それでは、メッシュ頂点の位置情報をテクスチャに保存する手法について、軽く説明しようと思います。

Unity の Meshオブジェクトは、Unity で表示する 3D モデルの頂点の位置、法線、UV 値などのデータが格納されたクラスになっています。頂点位置情報（Mesh.vertices）には、メッシュの全ての頂点数分の位置情報が Vector3の配列で保存されています。表 1.1

そして、Unity の Texture2Dオブジェクトは、テクスチャの幅 (texture.width) ×高さ（ texture.height）のピクセル数分、色情報 (Color) の配列で保存されています。表 1.2

▼表 1.1　位置情報（Vector3）

x float x 方向成分
y float y 方向成分
z float z 方向成分

▼表 1.2　色情報（Color）

r float 赤色成分
g float 緑色成分
b float 青色成分
a float 不透明度成分

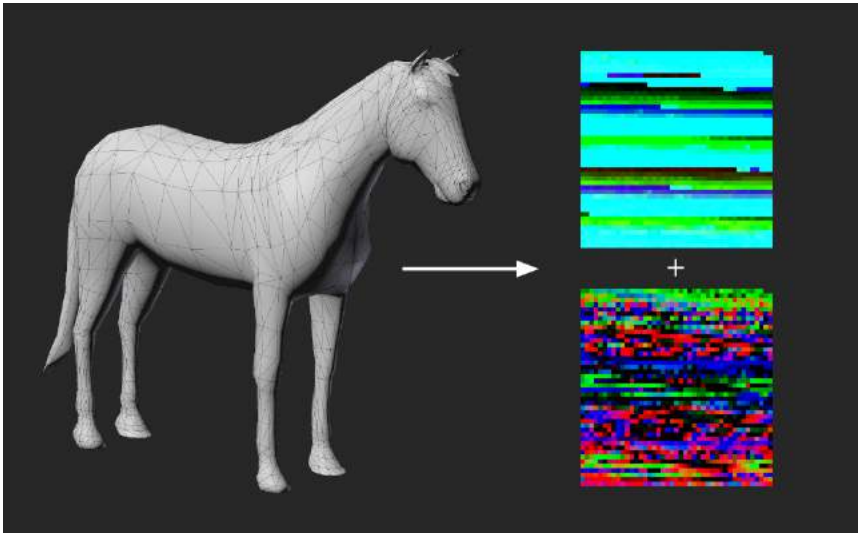
頂点の位置、Mesh.vertices 表 1.1 の x,y,z の値をそれぞれ、Texture2D の色情報 表 1.2 の r,g,b に格納し、EditorScript で TextureAsset として保存すれば、頂点の位置情報をテクスチャとして保存することになります。メッシュ頂点の位置と法線をテクスチャの色として保存する、サンプルスクリプトです。リスト 1.3

▼リスト 1.3　頂点情報をテクスチャに保存する

```
 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:     //頂点数<幅×高さになる width,height を求める
 7:
 8:     posTex = new Texture2D(width, height, TextureFormat.RGBAFloat, false);
 9:     normTex = new Texture2D(width, height, TextureFormat.RGBAFloat, false);
10:     //Color[] を格納する Texture2D
```

```
11:     //TextureFormat.RGBAFloat を指定することで、色情報を各要素 Float 値で持てる
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:     //頂点数分の色情報配列
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:     //各頂点において、Color.rgb = Vector3.xyz となるように、
33:     //位置→色、法線→色となるような色配列 (Color[]) を生成する。
34:
35:     posTex.SetPixels(posColors);
36:     normTex.SetPixels(normColors);
37:     posTex.Apply();
38:     normTex.Apply();
39:     //色配列をテクスチャにセットし、適用する
40: }
```

これで、Meshの頂点の位置、法線情報を位置テクスチャ、法線テクスチャに焼きこむことができました。



▲図 1.5　Mesh の頂点位置と法線を Texture に書き込み

実際には、ポリゴンを作る時の頂点の並び順（Index）データが無いので位置テクスチャと法線テクスチャのみではメッシュの形を再現することはできないのですが、メッシュの情報をテクスチャに書き込むことができました。図 1.5

Unity の公式マニュアルでは、`Texture2D.SetPixels(Color[])`は、`ColorFormat.RGBA32,ARGB32,RGB24,Alpha8`の場合のみ動作する。と書いてあります。これは、固定小数点値、Fixed 精度のテクスチャフォーマット時のみということなのですが、どうやら `RGBAHalf, RGBAFloat`の浮動小数点値でも、動作するようで、色の各要素に負の値や 1 以上の値を代入しても、`Clamp`されずに値を保持してくれるようです。固定精度のテクスチャに `Color`を代入すると、RGB 値は 0 ～ 1 の値、精度は 1/256 に制限されます。

今回のアニメーションの頂点情報をテクスチャに焼きこむ手法では、アニメーションを一定間隔ごとにサンプリングし、各フレームのメッシュの頂点情報を並べて、一連のアニメーション情報をまとめて 1 枚のテクスチャに焼きこみます。テクスチャは、位置情報テクスチャと法線情報テクスチャの計 2 枚、生成します。

1.4.3　AnimationClip.SampleAnimation()

今回、アニメーションのサンプリングには、`AnimationClip.SampleAnimation(gameObject, time);`という関数を使用します。指定した GameObject に対して、

AnimationClip の指定した時間の状態にする。というもので、レガシー `Animation` にも、`Animator`にも対応しています。（というより、Animation や Animator のコンポーネントを使用せずにアニメーションを再生する方法です。）

それでは、実際の、AnimationClip からフレームを指定し頂点位置を取得する実装を解説していきます。

1.5 実装

今回のプログラムは次の 3 つの要素から成り立っています。

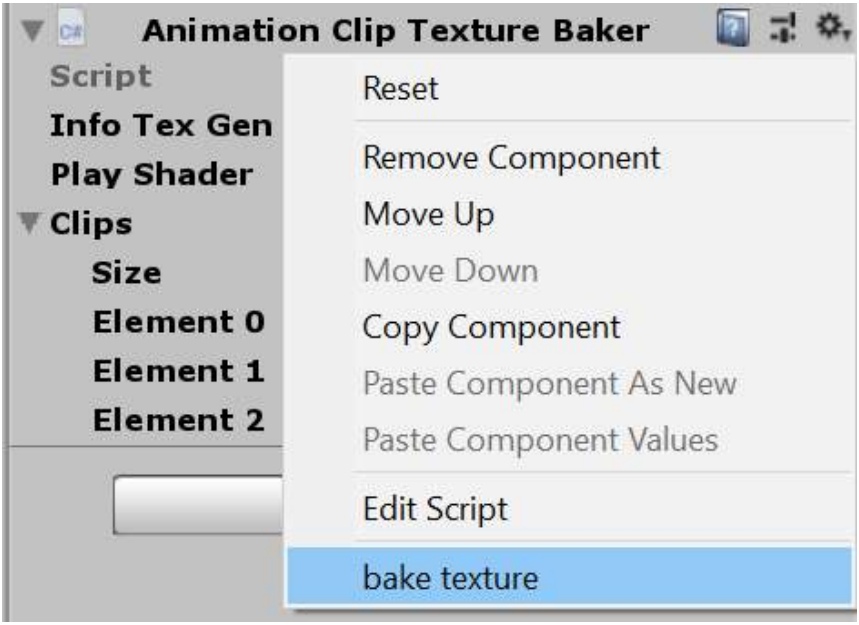
- AnimationClipTextureBaker.cs
- MeshInfoTextureGen.compute
- TextureAnimPlayer.shader

AnimationClipTextureBaker で、Animation、もしくは Animator から AnimationClip を取得し、AnimationClip を各フレームにサンプリングしながらのメッシュの頂点データの ComputeBuffer を作成します。そして、AnimationClip と Mesh のデータから作られた頂点アニメーション情報の ComputeBuffer を MeshInfoTextureGen.compute にて、位置情報テクスチャと法線情報テクスチャに変換する ComputeShader です。

TextureAnimPlayer.shader は、作られた位置情報テクスチャと法線情報テクスチャを使ってメッシュをアニメーションさせるための Shader になります。

▲図 1.6　AnimationClipTextureBaker Inspector

AnimationClipTextureBakerのインスペクタです。アニメーションテクスチャを生成するための ComputeShader、アニメーションテクスチャを再生するための Shaderを設定します。そして、テクスチャ化したい AnimationClipを Clipsに設定しておきます。図 1.6



▲図 1.7　インスペクタのコンテキストメニューからテクスチャの書き込みを実行できる

ContextMenuAttributeを使用すると、スクリプト内のメソッドを Unity のインスペクタのコンテキストメニューから呼べるようになります。エディター拡張を作らなくても実行できるので便利です。今回の場合、コンテキストメニューの『bake texture』から、スクリプトの Bakeを呼び出せます。図 1.6

それでは、実際のコードを見ていきましょう。

▼リスト 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: //頂点情報は位置と法線の構造体
18:     public struct VertInfo
19:     {
20:         public Vector3 position;
21:         public Vector3 normal;
22:     }
23:
24: //Reset() は、エディタ上で GameObject にスクリプトを付けるときに呼ばれる
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: //Animation、Animator のコンポーネントがあったら自動的に AnimationClip を設定す
る
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:             //テクスチャの初期化
70:
71:             for (var i = 0; i < frames; i++)
72:             {
73:                 clip.SampleAnimation(gameObject, dt * i);
74: //AnimationClip の指定した時間の状態で GameObject をサンプリング
75:                 skin.BakeMesh(mesh);
76: //BakeMesh() を呼んで Skinning された状態のメッシュデータを取得
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: //アニメーションのフレームを先にリストに格納しておく
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: //頂点情報を ComputeBuffer にセット
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: //ComputeShader をセットアップし、テクスチャ生成する
116:
117:             buffer.Release();
118:
119: //生成したテクスチャを保存するためのエディタースクリプト
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: //生成したテクスチャをマテリアルに設定して、メッシュを設定し Prefab を作っている
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: }
```

いったん RenderTextureを生成して GPU で処理し、Graphics.CopyTexture(rt,tex2d);で Texture2Dにコピーし、エディタースクリプトで UnityAsset として保存しておけば、次からは再計算無しで使えるアセットとなるので、使いどころの多いテクニックかと思います。リスト 1.4（119,120 行目）

今回の実装では、テクスチャへの書き込みを ComputeShaderを用いて実装しています。大量の処理を行う場合、GPU を使用した方が高速になるので、有用なテクニックですので、ぜひマスターしてみてください。処理内容としては、スクリプトで生成

した頂点アニメーションの位置バッファ、法線バッファを各ピクセルにそのまま配置しているだけです。リスト 1.5

▼リスト 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: //テクスチャの x 軸が頂点 ID、y 軸方向が時間になるように頂点情報を並べる
22: }
```

スクリプトから生成されたテクスチャがこちらになります。図 1.8

▲図 1.8　生成されたテクスチャ

このテクスチャは x 軸方向に 1 列ずつ、サンプリングした各フレームにおけるメッシュの頂点が格納されています。そして、y 軸方向が時間になっていて、テクスチャをサンプリングするときの uv.yを変化させることでアニメーションの時間を指定することができるよう、テクスチャを設計しています。

注目していただきたいことは Texture.FilterMode = Bilinearになっているトコロです。テクスチャをサンプリングするとき各ピクセルが隣接するピクセルと補間されるのですが、これにより、テクスチャ生成時にスクリプトでサンプリングしたフ

レームと次のフレームの間の中途半端な時刻のアニメーションテクスチャをアニメーション再生時に Shader でサンプリングしたとき、フレーム毎の位置と法線が自動的に補間されることになるので、アニメーションが滑らかに再生されることになります。ちょっと説明がややこしいですね！

そしてこの場合、走行（Run）のアニメーションはループアニメーションなので `WrapMode = Repeat`にしています。これにより、アニメーションテクスチャの最後のピクセルと最初のピクセルが補間されるので、滑らかにループが繋がったアニメーションになります。もちろん、ループしないアニメーションからテクスチャを生成する場合、`WrapMode = Clamp`に設定する必要があります。

次に、生成したアニメーションテクスチャを再生するための Shader になります。リスト 1.6

▼リスト 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: //ループ用のマルチコンパイルを作っておくと便利
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: //SV_VertexID のセマンティックで頂点 ID を取得できる
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 は頂点 ID を元に指定する
59: //uv.y にアニメーションをサンプリングする時間 (t) を設定する
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: //テクスチャから位置情報と法線情報をサンプリング
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: }
```

アニメーションテクスチャを再生する Shader では、VertexTextureFetch（VTF）という手法を使用しています。簡単にいうと、頂点シェーダ内でテクスチャをサンプリングし、頂点の位置や各値の計算に使用する。という方法で、ディスプレイスメン

トマッピング等に良く利用されます。

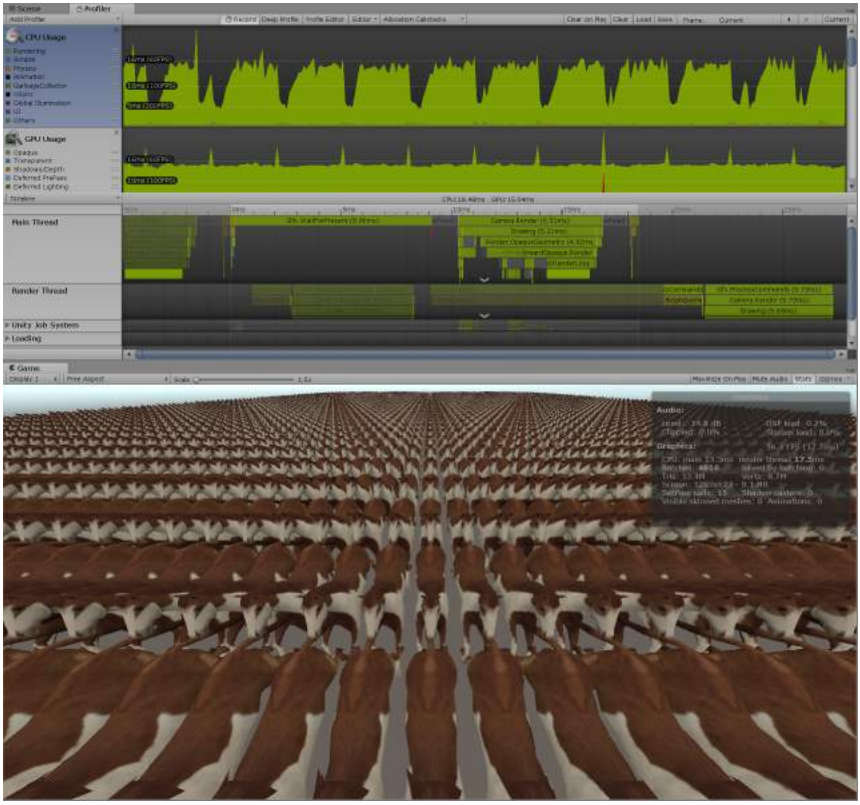
テクスチャのサンプリングには頂点 ID を使用しているのですが、これは、SV_VertexIDのセマンティックで取得できます。頂点情報は位置情報も法線情報もテクスチャから取得するので、appdata 内には uv しか無い部分も注目です。（appdataに POSITION,NORMALセマンティックを定義しても特にエラーにはなりません）

テクスチャをサンプリングするときの UV ですが、uv.yがアニメーションの正規化した時間（アニメーションの始まりを 0、終わりを 1.0 にしたときの値）になっています。uv.xは、頂点インデックス（vid）、uv.x = (vid + 0.5) * _TexelSize.xとなっていて、この 0.5 は何なのか？ と思うかも知れないのですが、これはテクスチャを Bilinearでサンプリングしたとき、(n + 0.5) / テクスチャサイズの位置だと、補間されていないテクスチャに入った値を得ることができるので、頂点 ID に 0.5 の値を足して、メッシュ内の頂点同士の補間されていない位置や法線を取得しています。

▼リスト 1.7 {TextureName}_TexelSize テクスチャサイズの情報を含む float4 のプロパティ (Unity 公式マニュアルより)

```
x には 1.0/width が含まれます
y には 1.0/height が含まれます
z には width が含まれます
w には height が含まれます
```

1.6 アニメーションする馬を 5000 頭置いてみる



▲図 1.9　テクスチャによりアニメーションする 5000 頭の馬

SkinnedMeshRendererを使わず、Rendererとアニメーションテクスチャにより、アニメーションを再生しています。FPS はスキニングアニメーションを使用していたときと比べると 8 → 56.4 と、大きく改善しています。図 1.9

※現在執筆中の PC の GPU は、GeForce MX150 で、NVIDIA Pascal GPU の中でも最弱のものとなっています。プロファイラとゲームウィンドウを同時にキャプチャするため、レンダリング解像度が少し小さくなりましたが、処理負荷のほとんどがメッシュのスキニングの処理だったので、そこまでは影響ないはずです。。！

また、注目してほしいのはインスタンシング対応等、他の最適化処理はしてないというトコロです。SkinnedMeshRendererを使わなくなったので、GPU インスタンシングによる描画が可能になりました。Shader のインスタンシング対応などにより、さらにパフォーマンスを追及することが可能だということです。

ここでは解説しませんが、表紙の鳥はテクスチャアニメーションさせた鳥を Graphics.DrawMeshInstancedIndirect()を用いて約 4000 羽の鳥を一気に描画しています。Shader のインスタンシング対応や他の応用については、ぼくの GitHub や他の記事を参考にしてみてください。

1.7 制限と応用先の検討

このテクスチャを使った手法にはいくつか制限があります。メッシュの頂点数やアニメーションの長さによりテクスチャを保持しておくメモリが消費される。アニメーションのブレンドには Shader を書く必要がある。AnimatorController のステートマシンを使用できない。などです。

その中で、一番大きな制限としては、ハードウェアごとに使用できるテクスチャの最大サイズがあります。それは、4K だったり 8K、16K だったりします。つまり、今回の手法ではメッシュの各フレームの頂点を横 1 列に並べるのでメッシュの頂点数がテクスチャサイズによって制限されるということです。

しかし、大量にオブジェクトを出す場合、そこまで頂点数の多いものを出すべきではないので、頂点数の制限は、そのまま受け入れ、メッシュの頂点数がテクスチャの最大サイズを超えないようにすることが得策かも知れません。この頂点数の制限を超えてベイキングアニメーションテクスチャを使いたい場合、複数テクスチャを使用する方法が考えられます。

もしくは、メッシュの頂点ではなく、スケルトンの各ボーンの行列を事前に計算しておき、テクスチャやバッファに保存しておく方法もあります。スキニングの処理自体は実行時に VertexShader で行うことになるので、通常のメッシュスキニング時にPostLateUpdate.UpdateAllSkinnedMeshesで行われていたスキニングの処理を Camera.Renderのレンダリング時にまとめて行うことになるので、処理負荷としてもかなり軽くなります。ぜひ、試してみてください。

AnimatorController や Unity のステートマシンを使用できないので、アニメーションの制御が難しくなるので、メインキャラクターではなく、ループアニメーションを繰り返すモブキャラや大量に飛ぶ鳥や蝶の群れなど、ある程度ごまかしが効くもに応用するのが良いかと思います。

1.8 まとめ

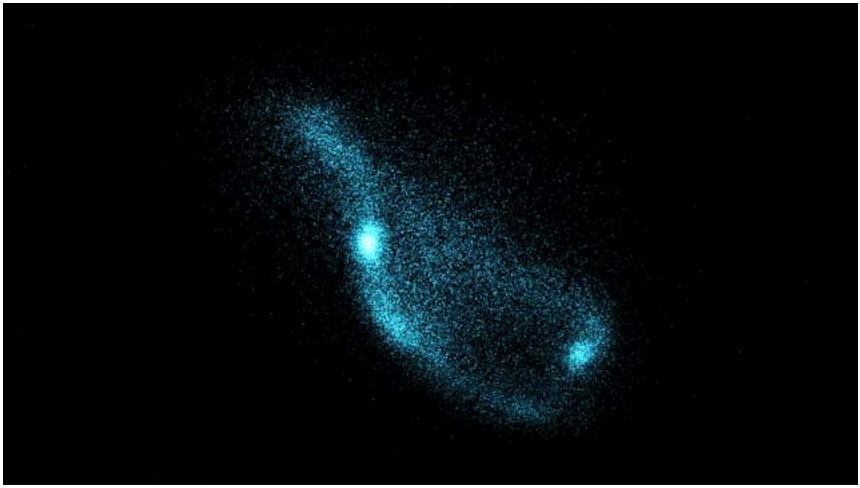
- 処理の最適化を進めるときは適切にプロファイリングし、どこの処理が重いか見極めることが重要
- スキニングアニメーションは処理負荷が高い
- 事前にスキニングした頂点座標をテクスチャに保存しておくことにより、時の実行処理負荷を減らすことができる
- GPU インスタンシングや GPU によるキャラクターの移動など、更なる最適化をする余地がある
- スケルトンの行列やシミュレーション結果など、リアルタイムだと重い処理を事前にテクスチャに保存しておき、実行時の処理負荷を軽くする方法がいろいろな応用の余地がありそう

第 2 章

Gravitational N-Body Simulation

2.1 はじめに

本章では、宇宙空間に存在している天体の動きをシミュレートする方法である、Gravitational N-Body Simulation の、GPU 実装手法を解説します。



▲図 2.1　Result

該当サンプルプログラムは、
`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`

「Assets/NBodySimulation」になります。

2.2 N-Body シミュレーションとは

N 個の物理的物体の相互作用を計算するシミュレーションのことを、N-Body シミュレーションと総称します。N-Body シミュレーションを用いる問題には多数の種類があり、特に、宇宙空間に点在している天体同士が重力によって引き合い、まとまりを形成する系を扱う問題のことを、**重力多体系問題**と呼びます。本章で解説されるアルゴリズムはこれに該当し、N-Body シミュレーションを用いて重力多体系の運動方程式を解いていく、ということになります。

また、N-Body シミュレーションは、重力多体系問題以外に、

- 分子間力の計算
- ダークマターの解析
- 銀河団の衝突の解析

など、小さいものから壮大なものまで、幅広い分野に応用されています。

2.3 アルゴリズム

早速、どのような数式を解いていくのか見ていきましょう。

重力多体系問題は、高校物理を履修している方にとっては馴染み深い方程式である、**万有引力の方程式**を、空間内の全天体に対して計算してあげることで、シミュレートが可能です。ただし、高校物理では一直線上にある物体のみを扱う都合上、このような記述で学習されていたのではないかと思います。

$$f = G\frac{Mm}{r^2} \tag{2.1}$$

ここで、 f は万有引力、 G は万有引力定数、 M, m は 2 天体のそれぞれの質量、 r は天体間の距離です。当然ながら、これでは 2 天体間の**力の大きさ** (スカラー量) しか求めることはできません。

今回の実装では、Unity 内部の 3 次元空間上での動きを考える必要があるため、方向を示す**ベクトル量**が必要になります。そこで、2 天体 (i, j) 間で発生する力のベクトルを求めるために、万有引力の方程式を次のように記述します。

$$\boldsymbol{f}_{ij} = G\frac{m_i m_j}{\|\boldsymbol{r}_{ij}\|^2} \cdot \frac{\boldsymbol{r}_{ij}}{\|\boldsymbol{r}_{ij}\|} \tag{2.2}$$

ここで、 $\boldsymbol{f}_{ij}$ は、天体 i が天体 j から受ける力のベクトル、 m_i, m_j は 2 天体のそれぞれの質量、 r_{ij} は天体 j から天体 i への方向ベクトルです。右辺左側は、最初に

出てきた万有引力の方程式と同じく、力の大きさを算出しており、右辺右側部分で力を受ける方向の単位ベクトルをかけることで、ベクトル化しています。

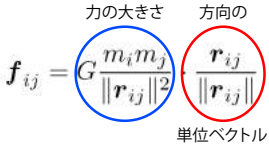


図: 数式の意味

さらに、2 天体間ではなく、1 つの天体 (i) が周囲のすべての天体から受ける力の大きさを $\boldsymbol{F}_i$ とすると、次のように計算できます。

$$\boldsymbol{F}_i = \sum_{j \in N} \boldsymbol{f}_{ij} = Gm_i \cdot \sum_{j \in N} \frac{m_j \boldsymbol{r}_{ij}}{\|\boldsymbol{r}_{ij}\|^3} \tag{2.3}$$

式にあるように、すべての万有引力の総和をとることで、周囲の天体から受ける力を算出することができます。

また、シミュレーションを簡単にするために、Softening 因子 ε を用いて方程式を書き換えると、次のようになります。

$$\boldsymbol{F}_i \simeq Gm_i \cdot \sum_{j \in N} \frac{m_j \boldsymbol{r}_{ij}}{(\|\boldsymbol{r}_{ij}\|^2 + \varepsilon)^{\frac{3}{2}}} \tag{2.4}$$

これにより、天体が同一の位置に来てしまっても衝突を無視することが可能になります (自身同士で計算をしてしまってもシグマ内の結果は 0 となります)。

次に、運動の第二法則 $m\boldsymbol{a} = \boldsymbol{f}$ を利用して、力のベクトルを加速度のベクトルに変換していきます。まず、運動の第二法則を変形して、次式のようにします。

$$m_i \boldsymbol{a}_i = \boldsymbol{F}_i \Longrightarrow \boldsymbol{a}_i = \frac{\boldsymbol{F}_i}{m_i} \tag{2.5}$$

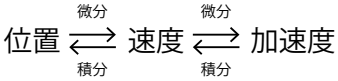
続いて、重力多体系の運動方程式に、先ほどの変形式を代入して書き換えると、天体が受ける加速度を算出することができます。

$$\boldsymbol{a}_i \simeq G \cdot \sum_{j \in N} \frac{m_j \boldsymbol{r}_{ij}}{(\|\boldsymbol{r}_{ij}\|^2 + \varepsilon)^{\frac{3}{2}}} \tag{2.6}$$

これで、シミュレーションの下準備は整いました。さて、数式では重力多体系問題を表現することができましたが、これらの数式をプログラムに落とし込むにはどのようにすればよいのでしょうか。次節でしっかりと解説していきたいと思います。

2.4 差分法

先述の式 (2.6) は、方程式の中でも**微分方程式**というものに分類されます。というのも、物理世界では**位置**、**速度**、**加速度**の関係は次の画像のようになっており、加速度が位置関数の 2 階微分であることから、そのまま微分方程式と呼ばれます。



▲図 2.2　位置・速度・加速度の関係

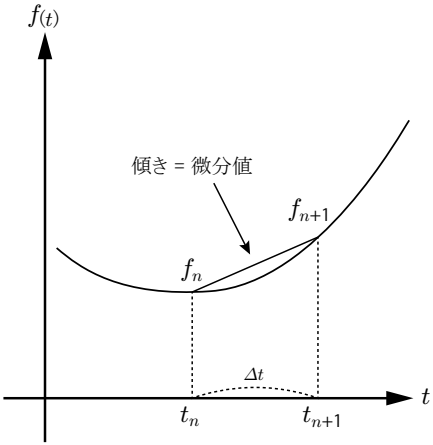
コンピュータで微分方程式を解く方法は様々存在しますが、中でも一般的なのが**差分法**と呼ばれるアルゴリズムです。手始めに、微分のおさらいから解説をはじめていきましょう。

2.4.1 微分

まずはじめに、数学的な微分の定義を確認します。関数 $f(t)$ の微分は次式で定義されます。

$$\frac{df}{dt} = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{2.7}$$

数式だけでは何を示しているのかがわかりにくいので、グラフに置き換えると次のようになります。



▲図 2.3　前進差分

関数の微分値は、 t_n 時点でのグラフの傾きになることはすでにご存じかとは思います。つまるところこのグラフは、傾きを算出するために Δt を無限に小さくしていきますよ、という状態を表しており、式 (2.7) そのものを表していることがわかるかと思います。

2.4.2　差分

コンピュータ上では数値としての「無限」を扱うことができません。そのため、できる限り小さい有限の Δt で近似してあげることになります。それを考慮して、先ほどの微分の定義を差分に書き換えると、次式のようになります。

$$\frac{df}{dt} \simeq \frac{f(t+\Delta t) - f(t)}{\Delta t} \tag{2.8}$$

そのまま極限が取れた形ですね。「無限小の Δt はコンピュータ上で表現できないので、ある程度の大きさの Δt で止めて近似します」という認識で大丈夫だと思います。

ここで、先ほどの図 2.2 を物理っぽく表してみると、次のようになります。

$$x(t) \xrightarrow{\frac{dx}{dt}} v(t) \xrightarrow{\frac{dx}{dt}} a(t)$$

▲図 2.4　位置・速度・加速度の関係 (数式)

つまり、式 (2.8) を、図 2.4 に照らし合わせると、次のようになります。

$$\frac{dx}{dt} \simeq \frac{x(t+\Delta t) - x(t)}{\Delta t} = v(t) \tag{2.9}$$

$$\frac{dv}{dt} \simeq \frac{v(t+\Delta t) - v(t)}{\Delta t} = a(t) \tag{2.10}$$

さらに、式 (2.9), (2.10) を合成すると、次のようになります。

$$x(t+\Delta t) = x(t) + v(t+\Delta t)\Delta t = x(t) + (v(t) + a(t)\Delta t)\Delta t \tag{2.11}$$

この式は、現在時刻 t から「 Δt 秒後の位置座標は、現在時刻の加速度と速度がわかっていれば算出が可能である」ということを意味しています。これが、差分法でシミュレーションを行う際の基礎的な考え方となります。また、このような微分方程式を差分法を用いて表現したものを、**差分方程式 (漸化式)** と呼びます。

実際に差分法でリアルタイムシミュレーションを行う場合は、微小時間 Δt (タイムステップ) を、1 フレームの描画時間 (60fps であれば、1/60 秒) にするのが一般的です。

2.5　実装

では、いよいよ実装に入っていきます。該当シーンは、「SimpleNBodySimulation.unity」になります。

2.5.1　CPU 側のプログラム

天体のデータ構造

まず初めに、天体粒子のデータ構造を定義します。式 (2.7) を見ると、1 つの天体がもつべき物理量は、「位置、速度、質量」であることがわかります。よって、次の構造体を定義してあげればよさそうです。

▼ Body.cs

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

バッファの用意

次に、生成したい粒子数をインスペクタから設定し、その個数分バッファを確保します。読み込み用と書き込み用でバッファを分けることにより、データの書き込み競合が起きないようにします。

また、構造体ひとつあたりの Byte 数は、「System.Runtime.InteropServices」名前空間の、「Marshal.SizeOf(Type t)」関数で取得することが可能です。

▼ SimpleNBodySimulation.cs

```
void InitBuffer()
{
  // バッファの作成 (Read/Write 用) → 競合防止
  bodyBuffers = new ComputeBuffer[2];

    // 各要素が Body 構造体のバッファを粒子の個数分作成
  bodyBuffers[READ] = new ComputeBuffer(numBodies,
    Marshal.SizeOf(typeof(Body)));

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

天体の初期配置

続いて、空間に粒子を配置します。初めに、粒子用の配列を作成し、それぞれの要素に物理量の初期値を与えます。サンプルでは、球体内をランダムサンプリングして初期位置とし、速度を 0、質量をランダムに与えました。

最後に、作成した配列をバッファをセットして、準備は完了です[*1]。

▼ SimpleNBodySimulation.cs

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

  // ルック調整用
```

[*1] ※ ルック調整のために、位置座標をスケーリングできる変数を用意していますが、すでに調整済みですので皆さんは気にする必要はありません。

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

    // バッファにセットするための配列を用意
    Body[] bodies = new Body[numBodies];

    int i = 0;
    while (i < numBodies)
    {
      // 球体内でサンプリング
      Vector3 pos = Random.insideUnitSphere;

      // 配列にセット
      bodies[i].position = pos * scale;
      bodies[i].velocity = Vector3.zero;
      bodies[i].mass = Random.Range(0.1f, 1.0f);


      i++;
    }

    // バッファに配列をセット
    bodyBuffers[READ].SetData(bodies);
    bodyBuffers[WRITE].SetData(bodies);

  }
```

シミュレーションルーチン

いよいよ、シミュレーションを実際に動かしていきます。次のコードは、毎フレーム実行される部分になります。

まずは、ComputeShader の定数バッファに値をセットします。差分方程式の Δt には、Unity に用意されている「Time.deltaTime」を使用します。また、GPU の実装の都合上、スレッド数・スレッドブロック数もあわせて転送しています。

計算終了後、シミュレーションの計算結果は、書き込み用のバッファに格納されていますので、次のフレームで読み込み用バッファとして利用するために最後の行でバッファを入れ替えています。

▼ SimpleNBodySimulation.cs

```
  void Update()
  {
    // コンピュートシェーダに定数を転送
    // Δ t
    NBodyCS.SetFloat("_DeltaTime", Time.deltaTime);
    // 速度減衰率
    NBodyCS.SetFloat("_Damping", damping);
    // Softening 因子
    NBodyCS.SetFloat("_SofteningSquared", softeningSquared);
    // 粒子数
    NBodyCS.SetInt("_NumBodies", numBodies);
```

```
    // ブロック当たりのスレッド数
    NBodyCS.SetVector("_ThreadDim",
      new Vector4(SIMULATION_BLOCK_SIZE, 1, 1, 0));

    // ブロック数
    NBodyCS.SetVector("_GroupDim",
    new Vector4(Mathf.CeilToInt(numBodies / SIMULATION_BLOCK_SIZE), 1, 1, 0));

    // バッファアドレスを転送
    NBodyCS.SetBuffer(0, "_BodiesBufferRead", bodyBuffers[READ]);
    NBodyCS.SetBuffer(0, "_BodiesBufferWrite", bodyBuffers[WRITE]);

    // コンピュートシェーダ実行
    NBodyCS.Dispatch(0,
      Mathf.CeilToInt(numBodies / SIMULATION_BLOCK_SIZE), 1, 1);


    // Read/Write を入れ替え (競合防止)
    Swap(bodyBuffers);
  }
```

レンダリング

シミュレーション計算後に、粒子をレンダリングするマテリアルに対してインスタンス描画指示を出します。粒子をレンダリングする際、粒子の位置座標をシェーダに与えてあげる必要があるため、計算後のバッファをレンダリング用シェーダに転送します。

▼ ParticleRenderer.cs

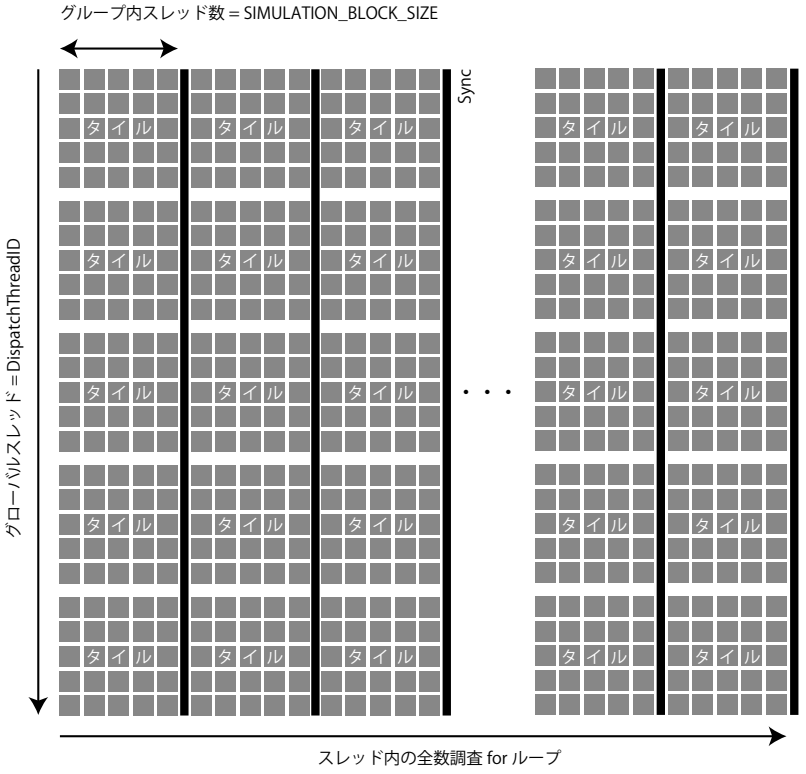
```
void OnRenderObject()
{
  particleRenderMat.SetPass(0);
  particleRenderMat.SetBuffer("_Particles"", bodyBuffers[READ]);

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

2.5.2 GPU 側のプログラム

N-Body シミュレーションでは、すべての粒子との相互作用を計算する必要がありますので、シンプルに計算していては実行時間が $O(n^2)$ となってしまい、パフォーマンスを出すことができません。そこで、UnityGraphicsProgramming Vol1、第 3 章に掲載されている SharedMemory(共有メモリ) の使い方を活用することにします。

考え方

同一ブロック内にあるデータは、シェアードメモリに格納してしまい、I/O を高速化します。スレッドブロックをタイルに見立てた概念図を次に示します。



▲図 2.5　タイルの概念

ここで、行は実行されているグローバルスレッド (DispatchThreadID)、列はスレッド内で全数調査されている対象の粒子です。時間が経過するにつれて、実行している列が右に 1 個ずつずれていくような認識です。

また、同時に実行されるタイルの総数は、(粒子の個数 / グループ内スレッド数) になります。サンプルでは、ブロック内スレッド数を 256 個 (SIMULA-

TION_BLOCK_SIZE) としているので、実際にはタイルの中身は 5x5 ではなく、256x256 あるという認識です。

すべての行は並列に動いていますが、タイル内でデータを共有するため、タイル内で実行されている列がすべて Sync に到達するまで同期待ちを行います (Sync 層より右に行かない)。Sync 層に到達したのち、次のタイルのデータをシェアードメモリに読み込みなおす、という形です。

定数バッファの用意

CPU からの入力を受けるための定数バッファを、ComputeShader 内に記述します。

また、粒子データを保存するためのバッファも用意しておきます。今回、Body 構造体は「Body.cginc」にまとめておきました。後々使いまわしそうなものは、cginc にまとめておくと便利です。

最後に、シェアードメモリを使用するための宣言もしておきます。

▼ SimpleNBodySimulation.compute

```
#include "Body.cginc"

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


// 粒子のバッファ
StructuredBuffer<Body> _BodiesBufferRead;
RWStructuredBuffer<Body> _BodiesBufferWrite;

// 共有メモリ（ブロック内で共有される）
groupshared Body sharedBody[SIMULATION_BLOCK_SIZE];
```

タイルの実装

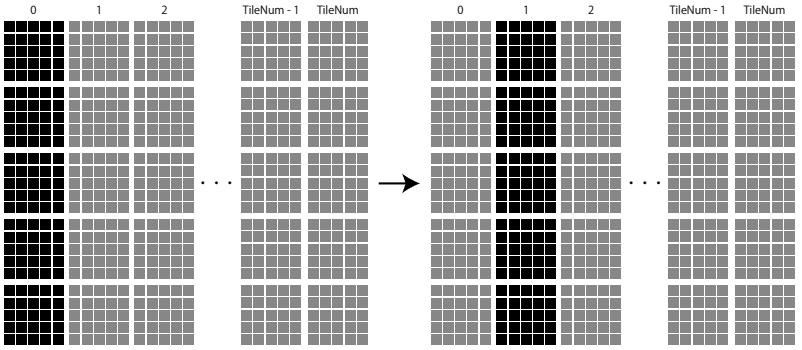
次に、タイルを実装します。

▼ SimpleNBodySimulation.compute

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

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

  float3 acc = (float3)0;
```

```
    int currentTile = 0;

    // タイル数（ブロック数）分実行
    for (uint i = start; i < finish; i += SIMULATION_BLOCK_SIZE)
    {
      // 共有メモリに格納
      // sharedBody[ブロック内スレッド ID]
      // = _BodiesBufferRead[タイル ID * ブロック内総スレッド数 + スレッド ID]
      sharedBody[threadID.x]
        = _BodiesBufferRead[wrap(groupID.x + currentTile, _GroupDim.x)
          * SIMULATION_BLOCK_SIZE + threadID.x];

      // グループ同期
      GroupMemoryBarrierWithGroupSync();

      // 周囲からの重力の影響を計算
      acc = gravitation(body, acc, threadID);

      GroupMemoryBarrierWithGroupSync();

      currentTile++;      // 次のタイルへ
    }

    return acc;

  }
```

コード内にある for ループのイメージ図を次の画像に置いておきます。



▲図 2.6　タイル ID の for ループ

相互作用の計算

先ほどのループでタイルの移動を制御していましたので、今度は**タイルの中での for ループ**を実装します。

▼ SimpleNBodySimulation.compute

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

  // 全数調査
  // ブロック内のスレッド数分実行
  for (uint i = 0; i < SIMULATION_BLOCK_SIZE;)
  {
    accel = bodyBodyInteraction(accel, sharedBody[i], body);
    i++;
  }

  return accel;
}
```

これで、タイル内の全数調査が完了します。また、この関数の return 後のタイミングで、タイル内のすべてのスレッドが処理を完了するまで待機します。

次に、式 (2.2) を、次のように実装します。

▼ SimpleNBodySimulation.compute

```
// シグマ内の計算
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;
}
```

これで、加速度の総和を求めるプログラムが完成しました。

差分法で位置を更新

続いて、次のフレームでの粒子の座標・速度を、これまでの計算で算出された加速度を使って、算出します。

▼ SimpleNBodySimulation.compute

```
[numthreads(SIMULATION_BLOCK_SIZE,1,1)]
void CSMain (
        uint3 groupID : SV_GroupID,     // グループ ID
        uint3 threadID : SV_GroupThreadID, // グループ内スレッド ID
        uint3 DTid : SV_DispatchThreadID // グローバルスレッド ID
) {

        // 現在のグローバルスレッドインデックス
        uint index = DTid.x;

        // 粒子をバッファから読み込み
        Body body = _BodiesBufferRead[index];

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

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

        // 差分法
        body.position += body.velocity * _DeltaTime;

        _BodiesBufferWrite[index] = body;

}
```

天体の位置座標が更新できました。これで、天体の動きのシミュレーションは完成です！

2.6 粒のレンダリング方法

本節では、前回[*2]の記事で説明が不十分だった、GPU パーティクルの描画方法について補足しておきたいと思います。

2.6.1 ビルボード

ビルボードとは、常にカメラの方向を向く、シンプルな平面オブジェクトのことを指します。世の中の大半のパーティクルシステムは、ビルボードによって実装されているといっても過言ではありません。そのビルボードをシンプルに実装するために

[*2] 「Unity Graphics Programming Vol.1 - 第 5 章 SPH 法による流体シミュレーション」でも、同様の粒子のレンダリングを行っています。

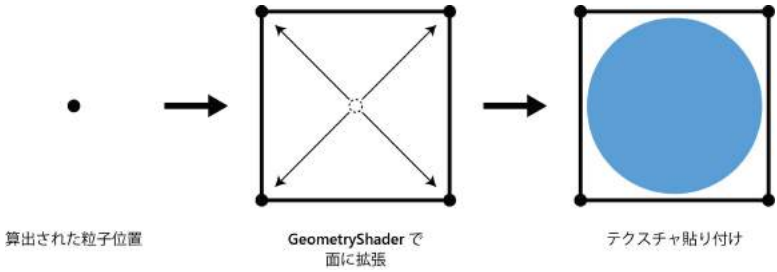
は、ビュー変換行列をうまく使ってあげる必要があります。

ビュー変換行列には、カメラ位置と回転を、原点に戻すような数値の情報が含まれています。つまり、ビュー変換行列を空間内のすべてのオブジェクトに掛けることで、カメラを原点とした座標系に変換されるわけです。

よって、**カメラの方向を向く**という特徴を持ったビルボードには、回転情報のみを含んだビュー変換行列の**逆行列**を、モデル変換行列としてかけてあげればよさそうです。(後述しますが、わかりにくいので図 2.8 に図解を掲載しています)

ビルボードの実装

まず初めに、パーティクルを描画するための Quad メッシュを作成します。これは、ジオメトリシェーダで 1 つの頂点を、xy 平面に平行な Quad に拡張することで簡単に実現できます[*3]。



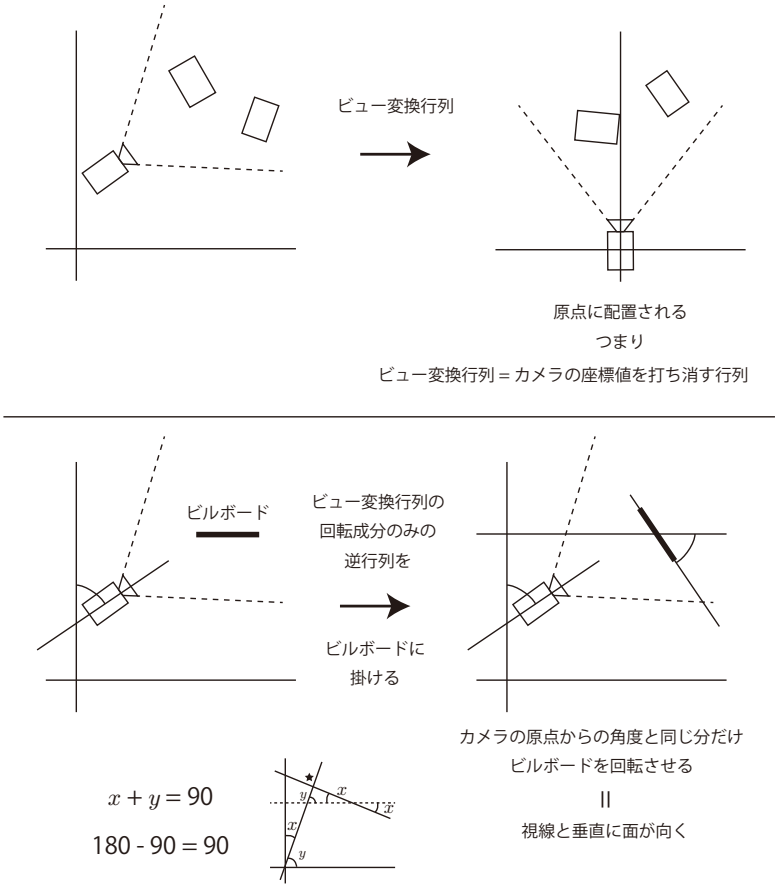
▲図 2.7　GeometryShader による Quad 拡張

この Quad に対して、ビュー変換行列の平行移動成分を打ち消した逆行列[*4]を、モデル変換行列として与えてあげると、その場でカメラの方向を向く Quad が作成できます。

何を言っているのかわからないと思いますので、次に解説図を示します。

[*3] ジオメトリシェーダの詳しい解説は、UnityGraphicsProgramming Vol.1 「ジオメトリシェーダで草を生やす」をご覧ください。

[*4] ビュー変換行列の逆行列は、単純に転置すればよいことが知られているので、ここでは転置で実装しています。



▲図 2.8　ビルボードの仕組み

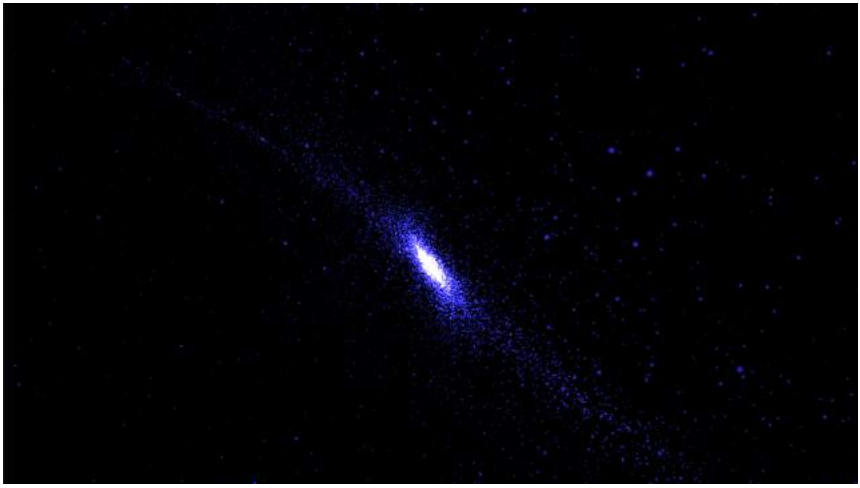
さらに、カメラの方向に向いたビルボードに対してビュー、プロジェクション行列をかけることにより、画面上の座標に変換することができます。これらを実装したシェーダを次に示します。

▼　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;

  // 回転成分だけ取り出す
  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 結果

以上のシミュレーションの結果を見てみましょう。



▲図 2.9　シミュレーション結果 (コレジャナイ感...)

動きを見てみると、すべての粒子が中心に集まってしまっていて、視覚的に面白くありません。そこで、次節にてひと工夫加えていきます。

2.8　視覚的に面白くするための工夫

該当シーンは、「NBodySimulation.unity」です。工夫といっても、変更点は 1 行だけで、次のようにタイル計算する部分を途中で打ち切ってしまいます。

▼ NBodySimulation.compute

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

  uint finish = _NumBodies / div;       // 途中で切る

   . . .

}
```

これにより、相互作用の計算が全数調査されることなく途中で破棄されますが、結果的にすべての粒子の影響を受けなかったことで、粒子の塊がいくつか発生し、1 点に集約されることはなくなります。

そして、複数の塊の部分同士が相互作用することで、冒頭の図 2.1 ような、よりダイナミックな動きを生みます。

2.9 まとめ

本章では、Gravitational N-Body Simulation の GPU 実装手法を解説しました。小さな原子から、大きな宇宙まで、N-Body シミュレーションの持つポテンシャルは無限大です。ぜひ皆さんも、オリジナルの宇宙を Unity 上で作ってみてはいかがでしょうか。少しでもお力になれましたら、幸いです！

2.10 参考

- GPU Gems 3 - Chapter 31. Fast N-Body Simulation with CUDA
- N 体シミュレーションで何がわかるか？ - 鹿児島大学 藤井通子 http://www.astro-wakate.org/ss2011/web/ss11_proceedings/proceeding/galaxy_fujii.pdf
- クォータニオンとビルボード - wgld.org https://wgld.org/d/webgl/w035.html

第 3 章

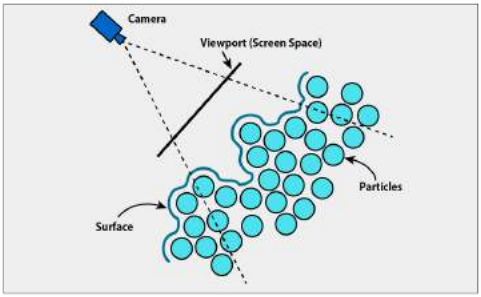
ScreenSpaceFluidRendering

3.1 はじめに

本章では、パーティクルのレンダリング手法の一つとして、**Deferred Shading** による **Screen Space Fluid Rendering** を紹介します。

3.2 Screen Space Fluid Rendering とは

流体のような連続体をレンダリングする手法としては、伝統的にはマーチンキューブス法が用いられますが、比較的計算負荷が高く、リアルタイムアプリケーションにおいての細部を追求した描画には向いていません。そこで、高速にパーティクルベースの流体を描画する手法として **Screen Space Fluid Rendering** という手法が考案されました。



▲図 3.1　Screen Space Fluid Rendering の概略図

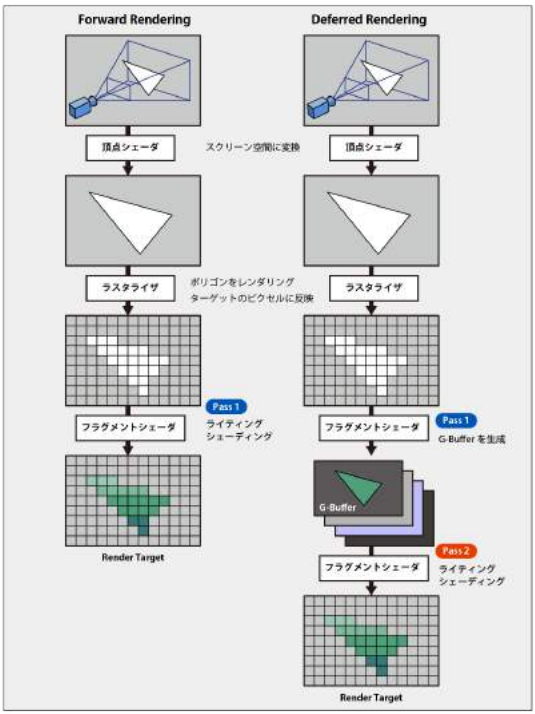
これは図 3.1 のように、カメラから見えるスクリーンスペース上のパーティクルの表面の深度からサーフェスを生成するというものです。

このサーフェスジオメトリの生成を行うために、**Deferred Rendering** という技術を用います。

3.3 Deferred Rendering（遅延シェーディング, 遅延レンダリング） とは

2 次元のスクリーンスペース（画面空間）でシェーディング（陰影計算）を行う技術です。区別のため、従来のタイプの手法は、**Forward Rendering** と呼ばれます。

図 3.2 は、従来の **Forward Rendering** と **Deferred Rendering** のレンダリングパイプラインの概略を描いた図です。



▲図 3.2　Foward Rendering と Deferred Rendering のパイプラインの比較

Forward Rendering の場合、シェーダの第 **1** パスでライティングやシェーディング処理を行いますが、**Deferred Rendering** では、シェーディングに必要な法線、位置、深度、拡散色などの 2 次元画像情報を生成し、**G-Buffer** と呼ばれるバッファに格納します。第 2 パスではそれらの情報を用い、ライティング、シェーディングを行い、最終的なレンダリング結果を得ます。このように実際のレンダリングが第 **2** パス（以降）に遅延されるので、**"Deferred**（遅延）**" Rendering** と呼ばれます。

Deferred Rendering の利点としては、

- 光源を多く使用できる
- シェーディングの際に、表示されている領域だけを計算するだけで済むので、フラグメントシェーダが実行される回数を最小限に抑えることができる
- ジオメトリの変形が可能
- G − Buffer の情報を PostEffect などで使用できる（SSAO など）

欠点としては、

- 半透明の表現に弱い
- MSAA などアルゴリズムによってはアンチエイリアシングの十分な効果が得られなくなる
- 複数のマテリアルを使うのが困難
- Orthographic カメラでのサポートをしていない

などがあり、トレードオフとなるような制約もできてしまうため、使用にはそれらを考慮した上で判断を行う必要があります。

Deferred Rendering の Unity での使用条件

Deferred Rendering には以下のような使用条件があり、環境によっては、本サンプルプログラムは動作いたしません。。

- Unity Pro でのみ使用可能
- マルチレンダーターゲット（MRT）が有効である
- シェーダーモデル 3.0 以上
- デプスレンダーテクスチャと two-sided ステンシルバッファをサポートするグラフィックスカードが必要

また、**Deferred Rendering** は、**Orthographic** プロジェクションを使用している場合はサポートされず、カメラのプロジェクションモードが **Orthographic** に設定されている場合は、**Forward Rendering** が使用されます。

3.4 G-Buffer（ジオメトリバッファ）

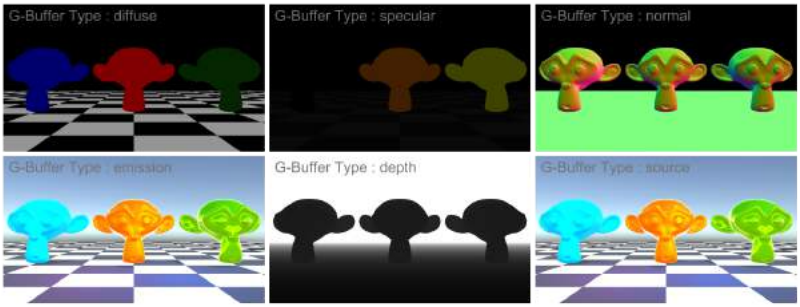
陰影計算、ラインティングの計算に使用する法線、位置、拡散反射色などのスクリーンスペースでの（2 次元テクスチャ）の情報は、**G-Buffer** と呼ばれます。Unity のレンダリングパイプラインの **G-Buffer** パスでは、それぞれのオブジェクトは一度レンダリングされ、**G-Buffer** テクスチャにレンダリングされ、デフォルトで以下の情報が生成されます。

render target	format	datat 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

これらの **G-Buffer** テクスチャは、グローバルなプロパティとして設定されており、シェーダ内で取得することができます。

shader property name	data type
_CameraGBufferTexture0	Diffuse color (RGB), occulusion (A)
_CameraGBufferTexture1	Specular color (RGB)
_CameraGBufferTexture2	World space normal (RGB)
_CameraGBufferTexture3	Emission + (Ambient + Reflections + Lightmaps)
_CameraDepthTexture	Depth + Stencil

サンプルコードの、**Assets/ScreenSpaceFluidRendering/Scenes/ShowGBufferTest** を開くと、この **G-Buffer** を取得して画面に表示する様子を確認することができます。



▲図 3.3　デフォルトで生成される G-Buffer

3.5　CommandBuffer について

この章で紹介するサンプルプログラムは、**CommandBuffer** という Unity の API を使用します。

CPU が実行するスクリプトに書かれたメソッドの中では、描画処理自体は行われません。代わりに、グラフィックスコマンドバッファと言われる **GPU** が理解できるレンダリングコマンドのリストに追加され、生成されたコマンドバッファは **GPU** によって直接読み込まれ、実行されることによって実際にオブジェクトを描画します。

Unity が用意するレンダリングコマンドは、例えば、**Graphics.DrawMesh()**, **Graphics.DrawProcedural()** などのメソッドです。

Unity の API の **CommandBuffer** を用いることで、Unity のレンダリングパイプラインの特定のポイントにコマンドバッファ（レンダリングコマンドのリスト）を差し込んで、Unity のレンダリングパイプラインを拡張することができます。

CommandBuffer を使ったサンプルプロジェクトは、ここでいくつか確認することができます。

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

3.6　座標系、座標変換について

以降、スクリーンスペース上で行われる計算の内容の理解のために、簡単に、3DCG のグラフィクスパイプラインと座標系について説明いたします。

Homogeneous Coordinates（同次座標系）

3 次元の位置ベクトル (x,y,z) を考える時に、(x,y,z,w) というように 4 次元のものとして扱うことがあり、これを Homogeneous Coordinates（同次座標）と呼びます。このように 4 次元で考えることによって 4x4 の Matrix（行列）を効果的に掛け合わせることができます。座標変換の計算は、基本的に 4x4 の Matrix を掛け合わせることで行われるので、位置ベクトルは、このように 4 次元で表現します。

同時座標と、非同次座標の変換はこのように行われます。(x/w, y/w, z/w, 1) = (x, y, z, w)

Object Space (オブジェクト座標系, ローカル座標系, モデル座標系)

オブジェクトそれ自身が中心となる座標系です。

World Space（ワールド座標系, グローバル座標系）

World Space は、シーンを中心として、シーンの中で複数のオブジェクトが空間的にどのような関係にあるのかを示す座標系です。World Space には、オブジェクトの移動・回転・スケールを行う **Modeling Transform** によって、**Object Space** から変換されます。

Eye(View) Space（視点座標系, カメラ座標系）

Eye Space は、描画するカメラを中心とし、その視点を原点とする座標系です。カメラの位置・カメラの上方向の向き、カメラのフォーカス方向の向きなどの情報を定義した **View Matrix** による **View Transform** を行うことによって **World Space** から変換されます。

Clip Space（クリッピング座標系, クリップ座標系）

Clip Space は、上記の **View Matrix** で定義されたカメラの他のパラメータ、field of view(FOV)・アスペクト比・near clip・far clip を定義した **Projection Matrix** を **View Space** に掛け合わせる変換によって得られる座標系です。この変換を **Projection Transform** と言い、これによって、カメラで描画される空間のクリッピングを行います。

Normalized Device Coordinates（正規化デバイス座標系）

Clip Space によって得られた座標値 xyz 各要素に対して w で除算することによって、-1<=x<=1、-1<=y<=1、0<=z<=1 にの範囲にすべての位置座標が正規化されます。これによって得られる座標系を **Normalized Device Coordinates**（**NDC**）と言います。この変換は **Persepective Devide** と呼ばれ、手前のオブ

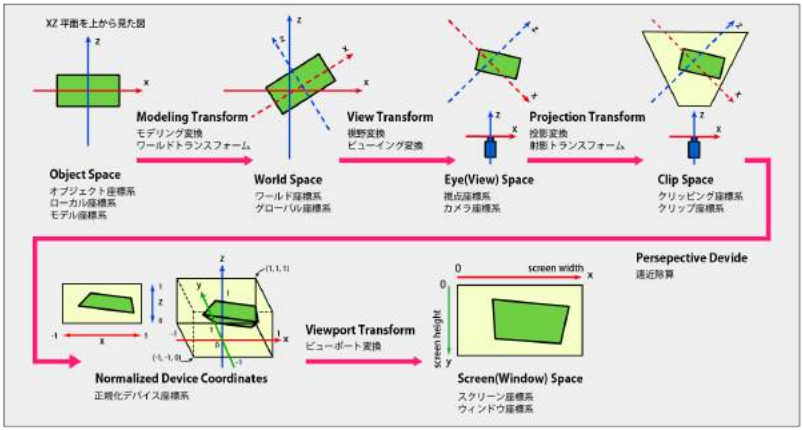
ジェクトは大きく、奥のものは小さく描画されるようになります。

Screen(Window) Space （スクリーン座標系、ウィンドウ座標系）

Normalized Device Coordinates で得られた正規化された値を、スクリーンの解像度に合うように変換した座標系です。Direct3D の場合、左上を原点とします。

Deferred Rendering では、このスクリーン空間での画像を元に計算しますが、必要によって、それぞれの変換の逆行列を掛けることによって、任意の座標系の情報を算出し使用するので、このレンダリングパイプラインを理解しておく事は重要です。

図 3.3 は 3DCG のグラフィックスパイプラインと座標系、座標変換の関係について説明したものです。



▲図 3.4　座標系、座標変換のフロー

3.7　実装についての解説

サンプルコードの、
Assets/ScreenSpaceFluidRendering/Scenes/ScreenSpaceFluidRendering
シーンを開いてください。

3.7.1 アルゴリズムの概要

Screen Space Fluid Rendering の大まかなアルゴリズムは以下の通りです。

1. パーティクルを描画しスクリーンスペースの深度画像を生成する
2. 深度画像にブラーエフェクトをかけ滑らかにする
3. 深度から表面の法線を計算する
4. 表面の陰影を計算する

※このサンプルコードでは、サーフェスジオメトリの作成までを行います。透過表現などは行っておりません。

3.7.2 スクリプトの構成

スクリプト名	機能
ScreenSpaceFluidRenderer.cs	メインのスクリプト
RenderParticleDepth.shader	パーティクルのスクリーンスペースの深度を求める
BilateralFilterBlur.shader	深度に応じて減衰するブラー エフェクト
CalcNormal.shader	スクリーンスペースのデプス情報から法線を求める
RenderGBuffer.shader	G-Buffer に深度、法線、カラー情報などを書き込む

3.7.3 CommandBuffer を作成し、カメラに登録

ScreenSpaceFluidRendering.cs の **OnWillRenderObject** 関数内では、**CommandBuffer** を作成し、カメラのレンダリングパスの任意の箇所に **CommandBuffer** を登録する処理を行います。

以下、コードを抜粋します

▼ ScreenSpaceFluidRendering.cs

```
// アタッチされたレンダラー (MeshRenderer) がカメラに映っているときに呼び出される
void OnWillRenderObject()
{
  // 自身がアクティブでなければ解放処理をして、以降は何もしない
  var act = gameObject.activeInHierarchy && enabled;
  if (!act)
  {
    CleanUp();
    return;
  }
  // 現在レンダリング処理をしているカメラがなければ、以降は何もしない
  var cam = Camera.current;
  if (!cam)
  {
    return;
```

```
  }

  // 現在レンダリング処理をしているカメラに、
  // CommandBuffer がアタッチされていなければ
  if (!_cameras.ContainsKey(cam))
  {
    // CommandBuffer の情報作成
    var buf = new CmdBufferInfo();
    buf.pass   = CameraEvent.BeforeGBuffer;
    buf.buffer = new CommandBuffer();
    buf.name = "Screen Space Fluid Renderer";
    // G-Buffer が生成される前のカメラのレンダリングパイプライン上のパスに、
    // 作成した CommandBuffer を追加
    cam.AddCommandBuffer(buf.pass, buf.buffer);

    // CommandBuffer を追加したカメラを管理するリストにカメラを追加
    _cameras.Add(cam, buf);
  }
```

Camera.AddCommandBuffer(CameraEvent evt, Rendering.CommandBuffer buffer) メソッドはカメラに、任意のパスで実行されるコマンドバッファを追加します。ここでは、**CameraEvent.BeforeGBuffer** で **G-Buffer** が生成される直前の位置を指定しており、ここに任意のコマンドバッファを差し込むことで、スクリーンスペース上で計算されたジオメトリを生成させることができます。追加したコマンドバッファは、アプリケーションの実行や、オブジェクトを Disable にしたタイミングで **RemoveCommandBuffer** メソッドを用いて削除します。カメラからコマンドバッファを削除する処理は、**Cleanup** 関数内に実装しています。

続いて、**CommandBuffer** にレンダリングコマンドを登録していきます。その際、フレームの更新の頭で、**CommandBuffer.Clear** メソッドによって、すべてのバッファのコマンドを削除しておきます。

3.7.4 パーティクルの深度画像を生成する

与えられたパーティクルの頂点のデータをもとにポイントスプライトを生成し、そのスクリーンスペースでの深度テクスチャを計算します。

以下、コードを抜粋します。

▼ ScreenSpaceFluidRendering.cs

```
// ------------------------------------------------------------------------
// 1. パーティクルをポイントスプライトとして描画し、深度とカラーのデータを得る
// ------------------------------------------------------------------------
// デプスバッファのシェーダプロパティ ID を取得
int depthBufferId = Shader.PropertyToID("_DepthBuffer");
// 一時的な RenderTexture を取得
```

```
buf.GetTemporaryRT(depthBufferId, -1, -1, 24,
  FilterMode.Point, RenderTextureFormat.RFloat);

// カラーバッファとデプスバッファをレンダーターゲットに指定
buf.SetRenderTarget
(
  new RenderTargetIdentifier(depthBufferId),  // デプス
  new RenderTargetIdentifier(depthBufferId)   // デプス書き込み用
);
// カラーバッファとデプスバッファをクリア
buf.ClearRenderTarget(true, true, Color.clear);

// パーティクルのサイズをセット
_renderParticleDepthMaterial.SetFloat ("_ParticleSize", _particleSize);
// パーティクルのデータ（ComputeBuffer）をセット
_renderParticleDepthMaterial.SetBuffer("_ParticleDataBuffer",
_particleControllerScript.GetParticleDataBuffer());

// パーティクルをポイントスプライトとして描画し、深度画像を得る
buf.DrawProcedural
(
  Matrix4x4.identity,
  _renderParticleDepthMaterial,
  0,
  MeshTopology.Points,
  _particleControllerScript.GetMaxParticleNum()
);
```

▼ RenderParticleDepth.shader

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

// ------------------------------------------------------------------
// Geometry Shader
// ------------------------------------------------------------------
// ポイントスプライトの各頂点の位置
static const float3 g_positions[4] =
{
  float3(-1, 1, 0),
  float3( 1, 1, 0),
  float3(-1,-1, 0),
  float3( 1,-1, 0),
};
// 各頂点の UV 座標値
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;
  // ポイントスプライトの中心の頂点の位置
  float3 vertpos = In[0].position.xyz;
  // ポイントスプライト 4 点
  [unroll]
  for (int i = 0; i < 4; i++)
  {
    // クリップ座標系でのポイントスプライトの位置を求め代入
    float3 pos = g_positions[i] * _ParticleSize;
    pos = mul(unity_CameraToWorld, pos) + vertpos;
    o.position = UnityObjectToClipPos(float4(pos, 1.0));
    // ポイントスプライトの頂点の UV 座標を代入
    o.uv       = g_texcoords[i];
    // 視点座標系でのポイントスプライトの位置を求め代入
    o.vpos     = UnityObjectToViewPos(float4(pos, 1.0)).xyz * float3(1, 1, 1);
    // ポイントスプライトのサイズを代入
    o.size     = _ParticleSize;

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

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

fragmentOut frag(g2f i)
{
  // 法線を計算
  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);

  // クリップ空間でのピクセルの位置
  float4 pixelPos     = float4(i.vpos.xyz + N * i.size, 1.0);
  float4 clipSpacePos = mul(UNITY_MATRIX_P, pixelPos);
  // 深度
  float  depth = clipSpacePos.z / clipSpacePos.w; // 正規化

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

```
    return o;
}
```

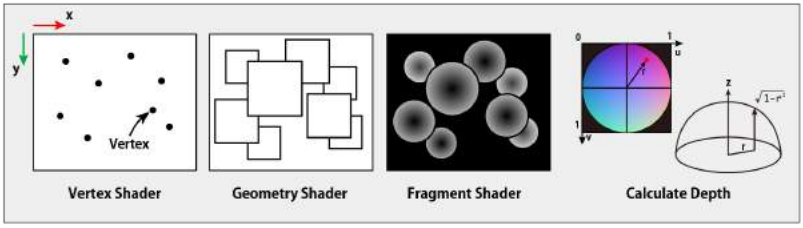
C#スクリプトでは、まず、スクリーンスペースでの計算のための一時的な **RenderTexture** を生成します。コマンドバッファでは、**CommandBuffer.GetTemporaryRT** メソッドによって、一時的な **RenderTexture** データを作り、それを利用します。**GetTemporaryRT** メソッドの第一引数には、作成したいバッファのシェーダプロパティのユニーク **ID** を渡します。シェーダにおけるユニーク **ID** とは、Unity のゲームシーンが実行される度に生成されるシェーダ内のプロパティにアクセスするための **int** 型の固有の **ID** で、**Shader.PropertyToID** メソッドでプロパティ名を渡して生成することができます。（この固有 ID は、実行されたタイミングが異なるゲームシーンでは異なるため、その値を保持したり、ネットワークを通じて他のアプリケーション共有する事はできません）

GetTemporaryRT メソッドの第 2,3 引数では、解像度を指定します。**-1** を指定すると、現在ゲームシーンでレンダリングしているカメラの解像度（Camera pixel width, height）が渡されます。

第 4 引数では、デプスバッファのビット数を指定します。**__DepthBuffer** では、デプス + ステンシルの値も書き込みたいため、0 以上の値を指定します。

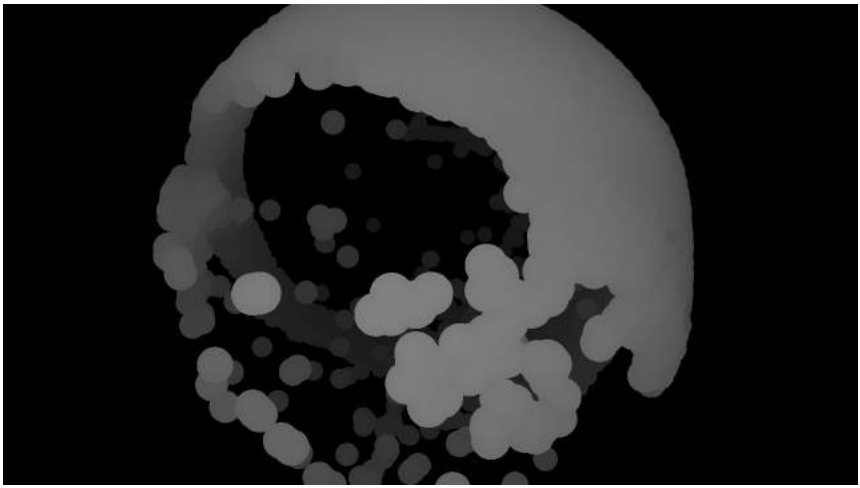
生成した RenderTexture は、**CommandBuffer.SetRenderTarget** メソッドで、レンダーターゲットに指定し、**ClearRenderTarget** メソッドで、クリアをしておきます。これを行わないと毎フレームごとに上書きされていくため、適切に描画されません。

CommandBuffer.DrawProcedural メソッドで、パーティクルのデータを描画し、スクリーンスペース上でのカラーと深度のテクスチャを計算します。この計算を図示すると以下のようになります。



▲図 3.5　深度画像の計算

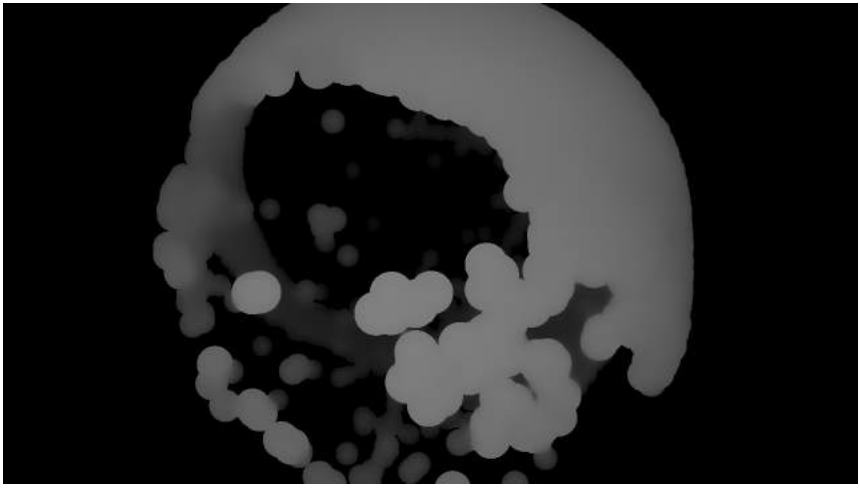
Vertex シェーダ、**Geometry** シェーダでは、与えられたパーティクルのデータから視点方向にビルボードするポイントスプライトを生成します。**Fragment** シェーダでは、ポイントスプライトの UV 座標値から半球体の法線を計算し、これを元に、スクリーンスペースでの深度画像を得ます。



▲図 3.6　深度画像

3.7.5　深度画像にブラーエフェクトをかけ滑らかにする

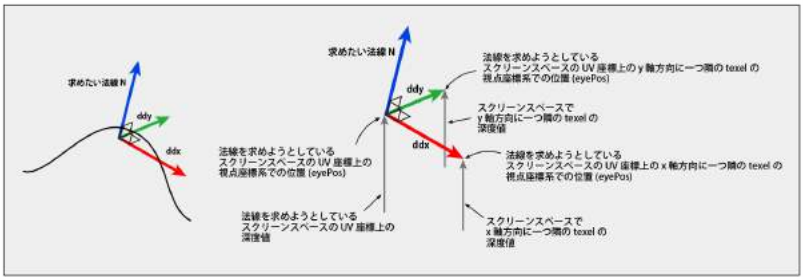
得られた深度画像をブラーエフェクトをかけ滑らかにすることで、隣接するパーティクルとの境界を曖昧にし連結しているように描画することができます。ここでは、深度に応じてブラーエフェクトのオフセット量の減衰がなされるようなフィルタを用いています。



▲図 3.7　ブラーをかけた深度画像

3.7.6　深度から表面の法線を計算する

ブラーを施した深度画像から法線を計算します。法線の計算には、X と Y 方向に、偏微分を行うことによって求めます。



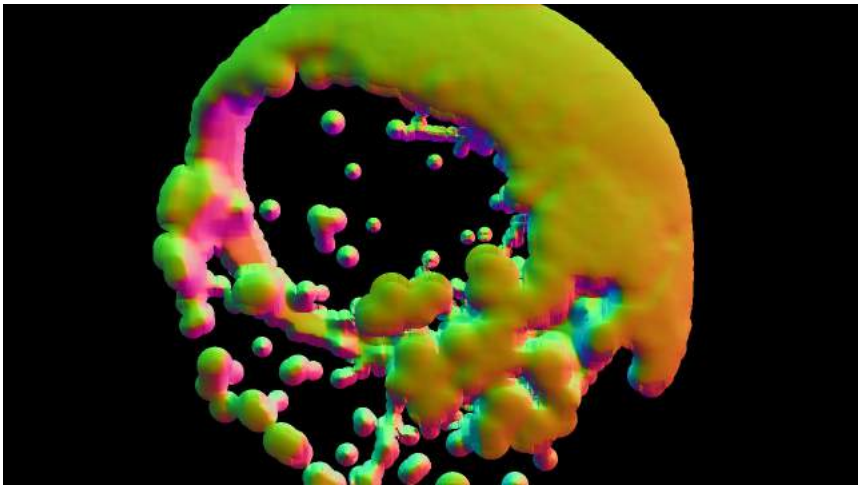
▲図 3.8　法線の計算

以下、コードを抜粋します

▼　CalcNormal.shader

```
// ------------------------------------------------------------------
// Fragment Shader
// ------------------------------------------------------------------
// スクリーンの UV から視点座標系での位置を求める
float3 uvToEye(float2 uv, float z)
{
  float2 xyPos = uv * 2.0 - 1.0;
  // クリップ座標系での位置
  float4 clipPos = float4(xyPos.xy, z, 1.0);
  // 視点座標系での位置
  float4 viewPos = mul(unity_CameraInvProjection, clipPos);
  // 正規化
  viewPos.xyz = viewPos.xyz / viewPos.w;

  return viewPos.xyz;
}

// 深度の値を深度バッファから得る
float sampleDepth(float2 uv)
{
#if UNITY_REVERSED_Z
  return 1.0 - tex2D(_DepthBuffer, uv).r;
#else
  return tex2D(_DepthBuffer, uv).r;
#endif
}

// 視点座標系での位置を得る
float3 getEyePos(float2 uv)
{
  return uvToEye(uv, sampleDepth(uv));
}

float4 frag(v2f_img i) : SV_Target
{
  // スクリーン座標からテクスチャの UV 座標に変換
  float2 uv = i.uv.xy;
  // 深度を取得
  float depth = tex2D(_DepthBuffer, uv);

  // 深度が書き込まれていなければピクセルを破棄
#if UNITY_REVERSED_Z
  if (Linear01Depth(depth) > 1.0 - 1e-3)
    discard;
#else
  if (Linear01Depth(depth) < 1e-3)
    discard;
#endif
  // テクセルサイズを格納
  float2 ts = _DepthBuffer_TexelSize.xy;

  // 視点座標系（カメラから見た）位置をスクリーンの uv 座標から求める
  float3 posEye = getEyePos(uv);

  // x について偏微分
  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;
```

```
    // y について偏微分
    float3 ddy = getEyePos(uv + float2(0.0, ts.y)) - posEye;
    float3 ddy2 = posEye - getEyePos(uv - float2(0.0, ts.y));
    ddy = abs(ddy.z) > abs(ddy2.z) ? ddy2 : ddy;

    // 外積から上で求めたベクトルと直交する法線を求める
    float3 N = normalize(cross(ddx, ddy));

    // 法線をカメラの位置との関係で変更する
    float4x4 vm = _ViewMatrix;
    N = normalize(mul(vm, float4(N, 0.0)));

    // (-1.0〜1.0) を (0.0〜1.0) に変換
    float4 col = float4(N * 0.5 + 0.5, 1.0);

    return col;
}
```



▲図 3.9　法線画像

3.7.7　表面の陰影を計算する

これまでの計算で求めた深度画像と法線画像を、**G-Buffer** に書き込みを行います。**G-Buffer** が生成されるレンダリングパスの直前に書き込むことによって、計算結果を元にしたジオメトリが生成され、シェーディングやライティングが施されます。

コードを抜粋します

▼ ScreenSpaceFluidRendering.cs

```
// ------------------------------------------------------------------------
// 4. 計算結果を G-Buffer に書き込みパーティクルを描画
// ------------------------------------------------------------------------
buf.SetGlobalTexture("_NormalBuffer", normalBufferId); // 法線バッファをセット
buf.SetGlobalTexture("_DepthBuffer",  depthBufferId);  // デプスバッファをセッ
ト

// プロパティをセット
_renderGBufferMaterial.SetColor("_Diffuse",  _diffuse );
_renderGBufferMaterial.SetColor("_Specular",
  new Vector4(_specular.r, _specular.g, _specular.b, 1.0f - _roughness));
_renderGBufferMaterial.SetColor("_Emission", _emission);

// G-Buffer を レンダーターゲットにセット
buf.SetRenderTarget
(
  new RenderTargetIdentifier[4]
  {
    BuiltinRenderTextureType.GBuffer0, // Diffuse
    BuiltinRenderTextureType.GBuffer1, // Specular + Roughness
    BuiltinRenderTextureType.GBuffer2, // World Normal
    BuiltinRenderTextureType.GBuffer3  // Emission
  },
  BuiltinRenderTextureType.CameraTarget  // Depth
);
// G-Buffer に書き込み
buf.DrawMesh(quad, Matrix4x4.identity, _renderGBufferMaterial);
```

▼ RenderGBuffer.shader

```
// GBuffer の構造体
struct gbufferOut
{
  half4 diffuse  : SV_Target0; // 拡散反射
  half4 specular : SV_Target1; // 鏡面反射
  half4 normal   : SV_Target2; // 法線
  half4 emission : SV_Target3; // 放射光
  float depth    : SV_Depth;   // 深度
};

sampler2D _DepthBuffer; // 深度
sampler2D _NormalBuffer;// 法線

fixed4 _Diffuse;  // 拡散反射光の色
fixed4 _Specular; // 鏡面反射光の色
float4 _Emission; // 放射光の色

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;
  }
```

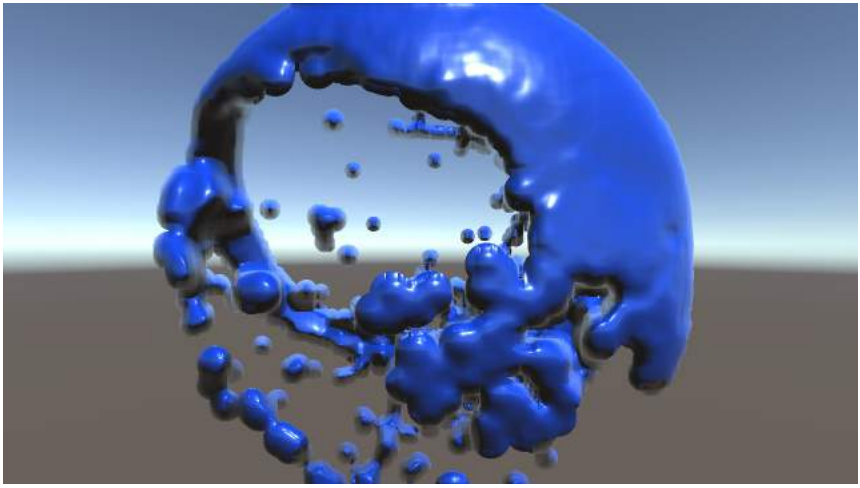
SetRenderTarget メソッドで、レンダーターゲットに書き込み対象の **G-Buffer** を指定します。第 1 引数のターゲットとしたいカラーバッファに **BuiltinRenderTextureType** 列挙型の **GBuffer0**、**GBuffer1**、**GBuffer2**、**GBuffer3** を指定した **RenderTargetIdentifier** の配列、また第 2 引数のターゲットとしたいデプスバッファに **CameraTarget** を指定することで、デフォルトの **G-Buffer** 一式、深度情報をレンダーターゲットに指定することができます。

コマンドバッファで複数のレンダーターゲットを指定したシェーダによるスクリーンスペース上の計算を行うために、ここでは、**DrawMesh** メソッドを用いて、画面を覆う矩形のメッシュを描画することで、これを実現しています。

3.7.8 一時的な RenderTexture の解放

GetTemporaryRT メソッドで生成した一時的な RenderTexture は、**ReleaseTemporaryRT** メソッドで忘れずに解放します。これを行わないと、毎フレームメモリが確保され、メモリのオーバーフローが起きてしまいます。

3.7.9 レンダリング結果



▲図 3.10　レンダリング結果

3.8 まとめ

この章は、**Deferred Shading** によるジオメトリの操作という点にフォーカスを当てた解説でした。今回のサンプルでは、半透明のオブジェクトとして、厚みに応じた光の吸収や内部での屈折を考慮した背景の透過、集光現象などの要素は実装しておりません。液体としての表現を追求するならば、これらの要素も実装していくと良いでしょう。**Deferred Rendering** を活用していくためには、Unity を使っていて普段は意識しなくても済むような座標系や座標変換、シェーディング、ライティングなどの 3DCG のレンダリングにおける計算の理解が求められます。観測の範囲では、まだまだ Unity での **Deferred Rendering** を使用したサンプルコードや学習のためのリファレンスが多くなく、自分自身まだ理解が不十分ですが、CG 表現の幅を広げることができる技術であると感じています。本章を通して、従来の **Forward Rendering** では実現できないような CG による表現を行いたいという目的を持った方への一助となれば幸いです。

3.9 参照

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

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

- なぜなにリアルタイムレンダリング, Satoshi Kodaira

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

第 4 章

GPU-Based Cellular Growth Simulation

4.1 はじめに

Processing[*1]による建築分野でのプロシージャルなモデリングを行うライブラリ、iGeo[*2]のチュートリアルに紹介されている「Cell Division and Growth Algorithm 1」[*3]のアルゴリズムをもとに、GPU を活用して細胞の分裂と成長の表現を行うプログラムを開発します。

本章のサンプルは

`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`

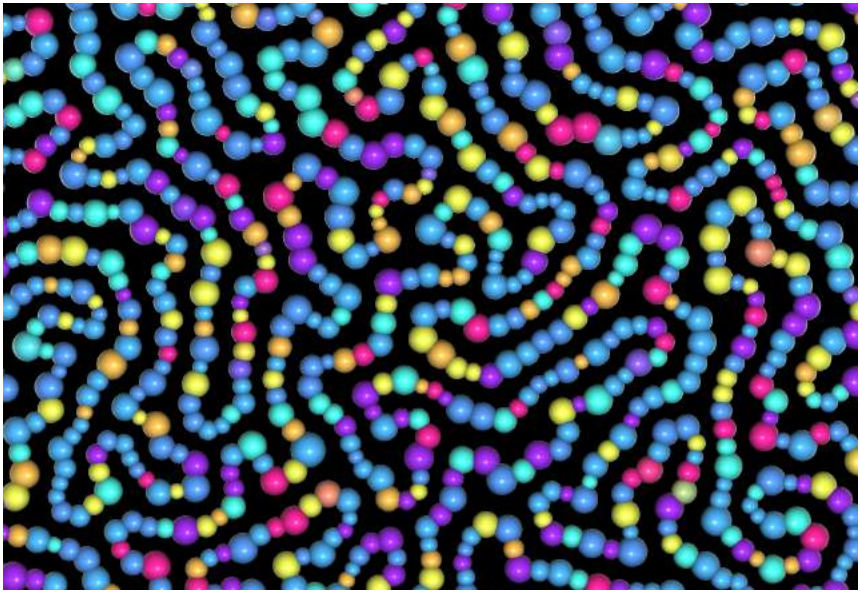
の「CellularGrowth」です。

本章では、GPU での細胞の分裂と成長プログラムを通して、

- Append/ConsumeStructuredBuffer を用いた GPU で動的にオブジェクト数を制御する方法
- GPU 上でのネットワーク構造の表現
- Atomic 演算による逐次的な処理

について解説を行います。

[*1] https://processing.org/
[*2] http://igeo.jp
[*3] http://igeo.jp/tutorial/55.html



▲図 4.1　CellularGrowthSphere.scene

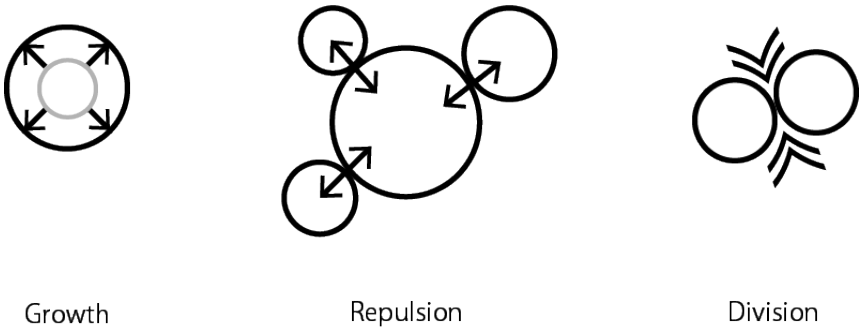
まずは Particle のみのシンプルな実装を紹介した後、Edge を導入し、成長しながら複雑化していくネットワーク構造を表現する手法を解説します。

4.2　細胞の分裂と成長シミュレーション

シミュレーションプログラムでは細胞のふるまいを模倣するため、Particle と Edge という 2 つの構造体を用意します。

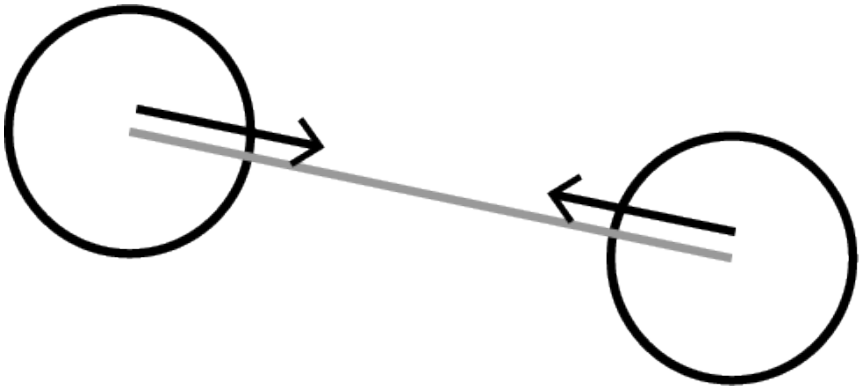
1 つの Particle は 1 つの細胞を表し、以下のようなふるまいをします。

- Growth(成長)：時間とともに大きくなる
- Repulsion(反発)：ほかの Particle と衝突して反発しあう
- Division(分裂)：特定の条件で分裂して 2 つの Particle に増える



▲図 4.2　細胞のふるまい

Edge は細胞同士がくっつきあう様子を表現します。分裂した Particle 同士を Edge で繋ぎ、バネのように引き合うことで Particle 同士をくっつけ、細胞のネットワーク構造を表現します。



▲図 4.3　Edge は繋いだ Particle 同士をくっつけ合う

4.3　実装

本節では必要な機能を段階的に実装することを通じて解説を進めます。

4.3.1 Particle の実装 (CellularGrowthParticleOnly.cs)

まずは Particle の挙動のみを実装したサンプル CellularGrowthParticleOnly.cs を通して、Particle の挙動と実装を解説します。

Particle の構造は以下のように定義します。

▼ Particle.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Particle_t {
    public Vector2 position;    // 位置
    public Vector2 velocity;    // 速度
    float radius;               // サイズ
    float threshold;            // 最大サイズ
    int links;                  // 繋がっている Edge の数 (後述の scene で利用)
    uint alive;                 // 活性化フラグ
}
```

本プロジェクトでは Particle を任意のタイミングで増減させるため、オブジェクトプールを Append/ConsumeStructuredBuffer によって管理し、GPU 上でオブジェクトの数を制御できるようにします。

Append/ConsumeStructuredBuffer について

Append/ConsumeStructuredBuffer[*4][*5]は、Direct3D11 から利用可能になった GPU 上で LIFO(Last In First Out : 後入れ先出し) を行うためのコンテナです。AppendStructuredBuffer はデータの追加を行い、ConsumeStructuredBuffer はデータの取り出しを行う役割を持ちます。

このコンテナを用いることで GPU 上で動的に数を制御でき、オブジェクトの増減を表現できるようになります。

バッファの初期化

まずは Particle のバッファとオブジェクトプールのバッファの初期化を行います。

▼ CellularGrowthParticleOnly.cs

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

[*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

```
    // オブジェクトプールの初期化
    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);

    // 分裂可能なオブジェクトを管理するオブジェクトプール
    dividablePoolBuffer = new ComputeBuffer(
        count,
        Marshal.SizeOf(typeof(int)),
        ComputeBufferType.Append
    );
    dividablePoolBuffer.SetCounterValue(0);

    // Particle とオブジェクトプールの初期化カーネルの実行 (後述)
    InitParticlesKernel();

    ...
}
```

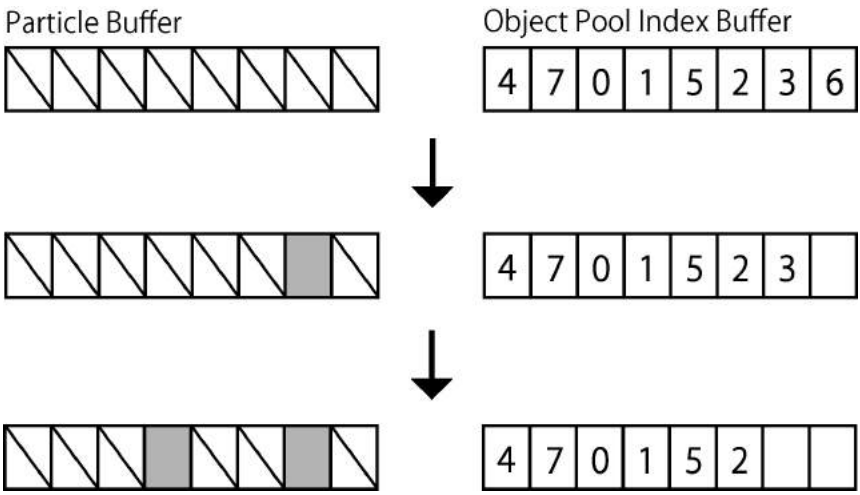
particleBuffer として利用している PingPongBuffer クラスはバッファを読み込み用と書き込み用の 2 つを用意するもので、後述の Particle の相互作用を計算する場面において活用します。

poolBuffer と dividablePoolBuffer が Append/ConsumeStructuredBuffer であり、初期化時の引数 ComputeBufferType に ComputeBufferType.Append を指定しています。Append/ConsumeStructuredBuffer は可変長のデータを扱えるのですが、初期化コードを見てわかるように、データ数の上限はバッファの作成時に設定しなければなりません。

int 型の Append/ConsumeStructuredBuffer として作成した poolBuffer は、

1. 初期化時に非活性な Particle の index を poolBuffer に貯める (Stack への Push)
2. Particle を追加する際に poolBuffer から index を取り出し (Stack からの Pop)、その index に紐づく particleBuffer 内の Particle の alive フラグを on にする

という流れによってオブジェクトプールとして機能させます。つまり、poolBuffer が持つ int バッファは常に非活性な Particle の index を指しており、必要に応じて取り出したりすることによってオブジェクトプールとして機能させることができるのです。(図 4.4)



▲図 4.4　左の配列が particleBuffer で右が poolBuffer を表している　初期状態では particleBuffer 内の Particle が全て非活性状態だが、Particle を出現させる際は poolBuffer から非活性の Particle の index を取り出し、該当 index の箇所の Particle を活性化させる

countBuffer は int 型のバッファで、オブジェクトプールの数を管理するために用います。

Start の最後に呼び出している InitParticlesKernel では、Particle とオブジェクトプールの初期化を行う GPU カーネルを実行しています。

▼ CellularGrowthParticleOnly.cs

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

    // オブジェクトプールを AppendStructuredBuffer として指定
    compute.SetBuffer(kernel, "_ParticlePoolAppend", poolBuffer);

    Dispatch1D(kernel, count);
}
```

以下が初期化を行うカーネルになります。

▼ 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 の初期化
  Particle p = create();
  p.alive = false; // 全 Particle を非活性に
  _Particles[idx] = p;

  // オブジェクトプールに Particle の index を追加
  _ParticlePoolAppend.Append(idx);
}
```

上記のカーネルを実行することで、particleBuffer 内のすべての Particle が初期化され非活性状態に、poolBuffer には非活性状態の全 Particle の index が格納されます。

Particle の出現

Particle を初期化できたので、次は Particle を出現させます。CellularGrowthParticleOnly.cs では、マウスをクリックした位置に Particle を発生させます。

▼ CellularGrowthParticleOnly.cs

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

マウスがクリックされていると、EmitParticlesKernel を実行して Particle を出現させます。

▼ CellularGrowthParticleOnly.cs

```
protected void EmitParticlesKernel(Vector2 point, int emitCount = 32)
{
    // オブジェクトプールの数と emitCount を比較して、
    // オブジェクトプールが空の状態で_ParticlePoolConsume.Consume() が実行されな
いようにする
    emitCount = Mathf.Max(
        0,
        Mathf.Min(emitCount, CopyPoolSize(poolBuffer))
```

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

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

    // オブジェクトプールを ConsumeStructuredBuffer として指定
    compute.SetBuffer(kernel, "_ParticlePoolConsume", poolBuffer);

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

    Dispatch1D(kernel, emitCount);
}
```

InitParticlesKernel では_ParticlePoolAppend パラメータに指定していた poolBuffer を、EmitParticlesKernel では_ParticlePoolConsume パラメータに指定していることからわかるように、Append/ConsumeStructuredBuffer にはそれぞれ同一のバッファを指定します。

GPU 上の処理での用途によって、バッファを追加用か (AppendStructuredBuffer)、取り出し用か (ConsumeStructuredBuffer) の設定を変えているだけで、CPU 側からみると同じバッファを GPU 側に送信していることになります。

EmitParticlesKernel の冒頭では、emitCount と GetPoolSize で取得したオブジェクトプールのサイズを比較していますが、これはオブジェクトプールが空の状態でプールからの index の取り出しが実行されないようにするためで、もし空のオブジェクトプールからさらに index を取り出そうとすると (GPU カーネル内で_ParticlePoolConsume.Consume を実行すると)、予期しない動作が発生してしまいます。

▼ CellularGrowth.compute

```
THREAD
void EmitParticles(uint3 id : SV_DispatchThreadID)
{
  // _EmitCount よりも多くの Particle を追加しないようにする
  if (id.x >= (uint) _EmitCount)
    return;

  // オブジェクトプールから非活性の Particle の index を取り出し
  uint idx = _ParticlePoolConsume.Consume();

  Particle c = create();

  // マウスの位置から少しずれた位置に Particle を配置する
  float2 offset = random_point_on_circle(id.xx + float2(0, _Time));
  c.position = _Point.xy + offset;
  c.radius = nrand(id.xx + float2(_Time, 0));

  // 活性化した Particle を非活性だった index の個所に設定
  _Particles[idx] = c;
```

```
}
```

EmitParticles では非活性な Particle の index をオブジェクトプールから取り出し、活性化した Particle を particleBuffer の該当 index の位置に設定しています。

上記のカーネルの処理によって、オブジェクトプールの数を考慮しつつ Particle を出現させることができます。

Particle のふるまい

これで Particle の出現を管理することができたので、次は Particle のふるまいをプログラムしていきます。

本章で開発するシミュレータの細胞は図 4.2 にもある通り、以下のふるまいをします。

- Growth：Particle は特定のサイズに達するまで徐々に大きくなる
- Repulsion：ほかの Particle と接触すると反発しあうように力が加わる
- Division：Particle は特定の条件で分裂する

Growth と Repulsion

Growth と Repulsion は Update 内で毎フレーム実行します。

▼ CellularGrowthParticleOnly.cs

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

    // 読み込み用のバッファを設定
    compute.SetBuffer(kernel, "_ParticlesRead", particleBuffer.Read);

    // 書き込み用のバッファを設定
    compute.SetBuffer(kernel, "_Particles", particleBuffer.Write);

    compute.SetFloat("_Drag", drag);            // 速度の減衰率
    compute.SetFloat("_Limit", limit);          // 速度の限界値
    compute.SetFloat("_Repulsion", repulsion);  // 反発する距離にかける係数
    compute.SetFloat("_Grow", grow);            // 成長速度

    Dispatch1D(kernel, count);

    // 読み込み用と書き込み用のバッファをスワップ (Ping Pong)
    particleBuffer.Swap();
```

```
}
```

読み込み用と書き込み用のバッファをそれぞれ設定し、処理の後にバッファをスワップしている理由は後述します。

以下が UpdateParticles カーネルになります。

▼ 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];

  // 活性化している Particle のみ処理する
  if (p.alive)
  {
    // Grow : Particle の成長
    p.radius = min(p.threshold, p.radius + _DT * _Grow);

    // Repulsion : Particle 同士の衝突
    for (uint i = 0; i < count; i++)
    {
      Particle other = _ParticlesRead[i];
      if(i == idx || !other.alive) continue;

      // Particle 同士の距離を計算
      float2 dir = p.position - other.position;
      float l = length(dir);

      // Particle 同士の距離が互いの半径の合計*_Repulsion よりも
      // 近ければ衝突している
      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);

      // _Drag パラメータに従って velocity を減衰させる
      p.velocity =
        normalize(p.velocity) *
        min(
```

```
            length(p.velocity) * _Drag,
            _Limit
          );
      }
      else
      {
        p.velocity = float2(0, 0);
      }
    }

    _Particles[idx] = p;
}
```

UpdateParticles カーネルでは、Particle 同士の衝突を計算するため、読み込み用のバッファ (_ParticlesRead) と書き込み用のバッファ (_Particles) とを利用しています。

もしここで読み込みも書き込みも同一のバッファを利用してしまった場合、GPU の並列処理により、別スレッドで更新された後の Particle の情報を、また別のスレッドが Particle の位置計算に用いる可能性が出てきてしまい、計算の整合性が取れない問題 (データレース) が発生してしまいます。

一つのスレッドが別スレッドで更新される情報を参照しなければ、読み込みと書き込み用で別々にバッファを用意する必要はありませんが、スレッドが別スレッドで更新されたバッファを参照してしまうような場合は、UpdateParticles カーネルのように読み込みと書き込み用のバッファを別々に用意し、更新を行うたびに交互に入れ替える必要があります。(処理が終わるたびに交互にバッファを入れ替えることから Ping Pong バッファと呼びます)

Division

Particle の分裂はコルーチンによって一定時間ごとに実行します。

Particle の分裂処理は

1. 分裂可能な Particle の index を取得し、dividablePoolBuffer に格納
2. dividablePoolBuffer から分裂させたい分の Particle を取り出し、分裂させる

という流れで行われます。

▼ 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);
}

...

// 分裂可能な Particle 候補を dividablePoolBuffer に格納
protected void GetDividableParticlesKernel()
{
    // dividablePoolBuffer をリセット
    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)
{
    // 分裂させたい数 (maxDivideCount) と
    // 分裂可能な Particle の数 (dividablePoolBuffer のサイズ) を比較
    maxDivideCount = Mathf.Min(
        CopyPoolSize(dividablePoolBuffer),
        maxDivideCount
    );

    // 分裂させたい数 (maxDivideCount) と
    // オブジェクトプールに残っている Particle の数 (poolBuffer のサイズ) を比較
    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);
}
```

GetDividableParticles カーネルによって、dividablePoolBuffer に分裂可能な Particle(active になっている Particle) を追加し、そのバッファを元に、実際に分裂処理を行う DivideParticles カーネルを実行する回数をもとめます。

分裂処理の回数のもとめ方は DivideParticlesKernel 関数の冒頭の通りで、

- maxDivideCount
- dividablePoolBuffer が持つ分裂可能な Particle 数、
- poolBuffer が持つオブジェクトプールに残っている非活性な Particle 数

とを比較します。これらの数値の比較によって、分裂可能な数の制限を超えて分裂処理が走ることを防いでいます。

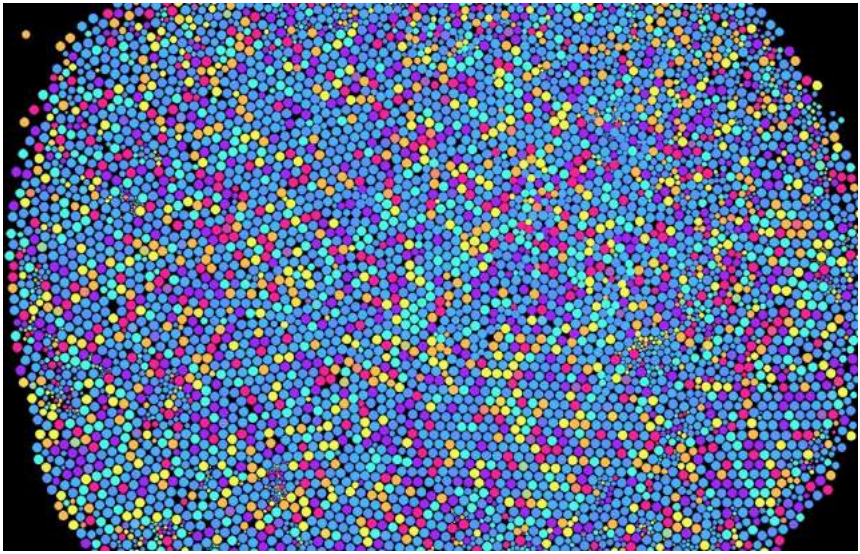
以下がカーネルの中身になります。

▼ CellularGrowth.compute

```
// 分裂できる Particle の候補を決定する関数
// ここの条件を変更することで分裂パターンを調整することができる
bool dividable_particle(Particle p, uint idx)
{
  // 成長率に応じて分裂
  float rate = (p.radius / p.threshold);
  return rate >= 0.95;

  // ランダムに分裂
  // return nrand(float2(idx, _Time)) < 0.1;
}

// Particle を分裂する関数
uint divide_particle(uint idx, float2 offset)
{
  Particle parent = _Particles[idx];
  Particle child = create();

  // サイズを半分に設定
  float rh = parent.radius * 0.5;
  rh = max(rh, 0.1);
  parent.radius = child.radius = rh;

  // 親と子の位置をずらす
  float2 center = parent.position;
  parent.position = center - offset;
  child.position = center + offset;

  // 子の最大サイズをランダムに設定
  float x = nrand(float2(_Time, idx));
  child.threshold = rh * lerp(1.25, 2.0, x);

  // 子の index をオブジェクトプールから取得し、子 Particle をバッファに設定
  uint cidx = _ParticlePoolConsume.Consume();
  _Particles[cidx] = child;

  // 親 Particle を更新
  _Particles[idx] = parent;

  return cidx;
}

uint divide_particle(uint idx)
```

```
{
  Particle parent = _Particles[idx];

  // ランダムに位置をずらす
  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);
}
```

これらの処理によって実現される細胞分裂の結果は以下のようになります。



▲図 4.5　CellularGrowthParticleOnly.scene

4.3.2　ネットワーク構造の表現 (CellularGrowth.cs)

細胞同士がくっつきあう様子を実現するため、Particle 同士を結ぶ Edge を導入し、細胞をネットワーク構造で表現します。

ここからは CellularGrowth.cs の実装を通して解説を進めます。

Edge は Particle が分裂するタイミングで追加され、分裂した Particle 同士をつなげます。

Edge の構造は以下のように定義します。

▼ Edge.cs

```
[StructLayout(LayoutKind.Sequential)]
public struct Edge_t
{
    public int a, b;        // Edge が結ぶ 2 つの Particle の index
    public Vector2 force;   // 2 つの Particle 同士をくっつけあわせる力
    uint alive;             // 活性化フラグ
}
```

Edge も Particle と同様に増減するため、Append/ConsumeStructuredBuffer で管理します。

Division

ネットワーク構造の分裂は以下のような流れで行います。

1. 分裂可能な Edge の候補を取得し、dividablePoolBuffer に格納
2. 分裂可能な Edge が空の場合は、接続 Edges 数が 0 の Particle(links が 0 の Particle) を分裂させ、2 つの Particle を Edge で接続する
3. 分裂可能な Edge がある場合は、dividablePoolBuffer から Edge を取り出し分裂させる

実際に分裂するのは Particle なのですが、ここで"分裂可能な Edge"と言っているのは、後ほど紹介する分裂パターンにおいて、分裂元の Particle と接続されている Edge を処理する際に都合が良いため、ネットワーク構造の分裂は Edge 単位で行っています。

上に挙げた分裂の流れによって、一つの Particle から分裂を繰り返し、大きなネットワーク構造を生成することができます。

Edge の分裂は前節の CellularGrowthParticleOnly.cs と同様、コルーチンによって一定時間ごとに実行されます。

▼ CellularGrowth.cs

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

protected void Divide()
{
    // 1. 分裂可能な Edge の候補を取得し、dividablePoolBuffer に格納
    GetDividableEdgesKernel();

    int dividableEdgesCount = CopyPoolSize(dividablePoolBuffer);
    if(dividableEdgesCount == 0)
    {
        // 2. 分裂可能な Edge が空の場合は、
        // 接続 Edges 数が 0 の Particle(links が 0 の Particle) を分裂させ、
        // 2 つの Particle を Edge で接続する
        DivideUnconnectedParticles();
    } else
    {
        // 3. 分裂可能な Edge がある場合は、dividablePoolBuffer から Edge を取り出
し分裂させる
        // 分裂パターン（後述）に応じて Edge の分裂を実行する
        switch(pattern)
```

```
        {
            case DividePattern.Closed:
                // 閉じたネットワーク構造を生成するパターン
                DivideEdgesClosedKernel(
                    dividableEdgesCount,
                    maxDivideCount
                );
                break;
            case DividePattern.Branch:
                // 枝分かれするパターン
                DivideEdgesBranchKernel(
                    dividableEdgesCount,
                    maxDivideCount
                );
                break;
        }
    }
}

...

protected void GetDividableEdgesKernel()
{
    // 分裂可能な Edge を格納するバッファをリセット
    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);

    // Particle の最大接続数
    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);
}
```

分裂可能な Edge を取得するカーネル (GetDividableEdges) は以下の通りです。

▼ CellularGrowth.compute

```
// 分裂可能かどうかの判断を行う
bool dividable_edge(Edge e, uint idx)
{
  Particle pa = _Particles[e.a];
  Particle pb = _Particles[e.b];

  // Particle の接続数が最大接続数 (_MaxLink) を超えず、
  // dividable_particle に定義された分裂条件を満たしていれば分裂可能とする
  return
    !(pa.links >= _MaxLink && pb.links >= _MaxLink) &&
    (dividable_particle(pa, e.a) && dividable_particle(pb, e.b));
}

...

// 分裂可能な Edge を取得する
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);
  }
}
```

分裂可能な Edge が存在しない場合は、以下の接続している Edge のない Particle を分裂させるカーネル (DivideUnconnectedParticles) を実行します。

▼ CellularGrowth.compute

```
// index が a の Particle と b の Particle をつなぐ Edge を生成する関数
void connect(int a, int b)
{
  // Edge のオブジェクトプールから非活性な Edge の index を取り出す
  uint eidx = _EdgePoolConsume.Consume();

  // Atomic 演算 (後述) を用いて
  // 各 Particle の接続数をインクリメントする
  InterlockedAdd(_Particles[a].links, 1);
  InterlockedAdd(_Particles[b].links, 1);

  Edge e;
  e.a = a;
  e.b = b;
  e.force = float2(0, 0);
```

```
  e.alive = true;
  _Edges[eidx] = e;
}

...

// 接続している Edge が存在しない Particle を分裂させる
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;

  // 親 Particle から分裂した子 Particle を生成
  uint cidx = divide_particle(idx);

  // 親 Particle と子 Particle を Edge で接続する
  connect(idx, cidx);
}
```

分裂した Particle 同士を接続する Edge を生成する connect 関数では、Atomic 演算というテクニックを用いて Particle の接続数をインクリメントしています。

Atomic 演算 (InterlockedAdd 関数について)

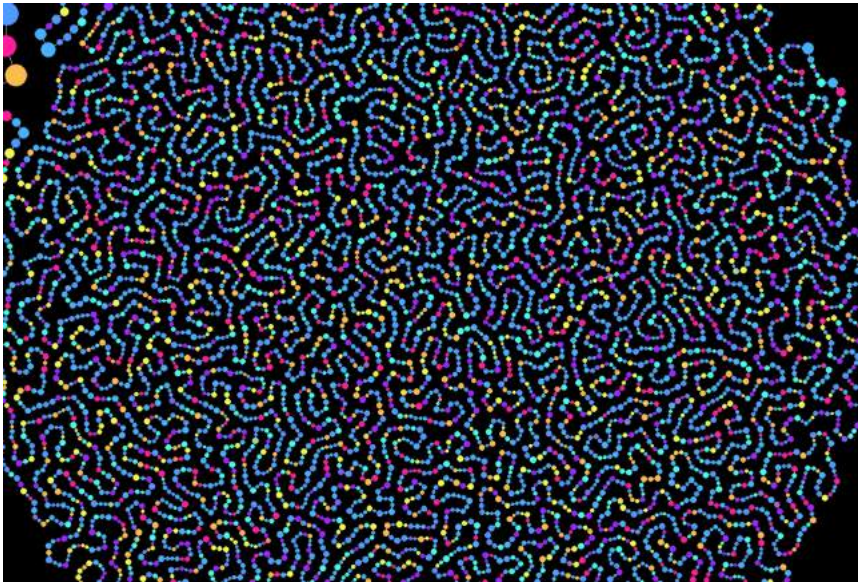
あるスレッドがグローバルメモリやシェアードメモリ上のデータを読み込み、修正し、書き込むという一連の処理を行うとき、その処理中にそのメモリ領域に他のスレッドからの書き込みが行われるなどして、値が変化するのを防ぎたい場合があります。(並列処理特有の、スレッドがメモリにアクセスする順番によって結果が変化してしまうデータレース (データの競合) と呼ばれる現象)
これを保証するのが Atomic 演算で、リソースの演算操作 (四則演算や比較) 中に他のスレッドからの干渉を防ぎ、GPU 上で安全に逐次的な処理を実現できます。
HLSL ではこれらの操作を行う関数[a]は Interlocked という prefix がついており、本章の例では InterlockedAdd を用いています。
InterlockedAdd 関数は、第一引数に指定されたリソースに第二引数に指定された整数を足し合わせる処理で、_Particles[index].links に 1 を足すことで接続数をインクリメントしています。
こうすることでスレッド間で一貫性のある接続数の管理が実現でき、矛盾なく接続数を増やしたり減らしたりすることが可能になります。

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

分裂可能な Edge がある場合は dividablePoolBuffer から Edge を取り出し分裂させます。DividePattern という enum パラメータを用意していることからわかるように、分裂には様々なパターンを適用することができます。

ここでは閉じたネットワーク構造を生成する分裂パターン (DividePattern.Closed) を紹介します。

■閉じたネットワーク構造 (DividePattern.Closed) 閉じたネットワーク構造を生成するパターンでは、以下の図のような分裂を行います。



▲図 4.6　閉じたネットワーク構造を生成するパターン (DividePattern.Closed)

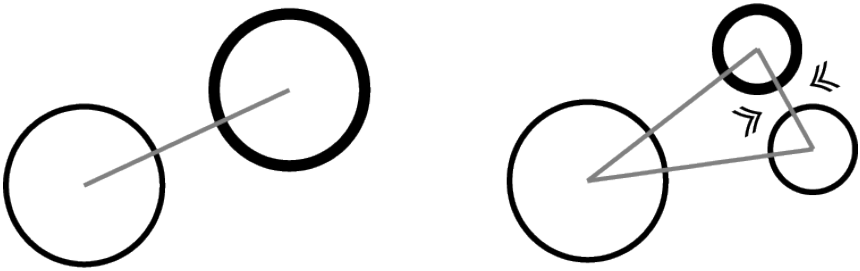
▼ CellularGrowth.cs

```
protected void DivideEdgesClosedKernel(
    int dividableEdgesCount,
    int maxDivideCount = 16
)
{
    // 閉じたネットワーク構造に分裂するパターン
    var kernel = compute.FindKernel("DivideEdgesClosed");
    DivideEdgesKernel(kernel, dividableEdgesCount, maxDivideCount);
}

// 分裂パターンで共通の処理
protected void DivideEdgesKernel(
    int kernel,
    int dividableEdgesCount,
    int maxDivideCount
)
{
    // オブジェクトプールが空の状態で Consume が呼ばれないように
    // maxDivideCount と各オブジェクトプールのサイズを比較
    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);
    }
```

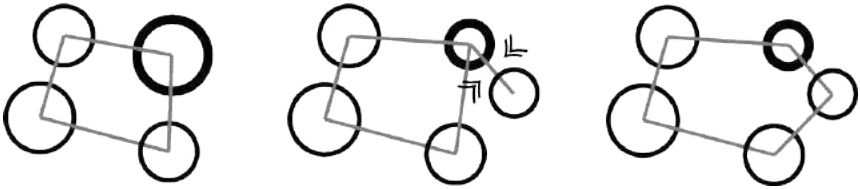
閉じたネットワーク構造を生成する GPU カーネル (DivideEdgesClosed) で用いている関数 divide_edge_closed は、Particle が持つ Edge の数に応じて処理を変えます。

いずれか一方の Particle の接続数が 1 の場合、分裂した Particle と足し合わせた 3 つの Particle で 3 角形を描くように Edge で繋ぎます。(図 4.7)



▲図 4.7　2 つの Particle と分裂した Particle で 3 角形を描くように閉じたネットワークを形成する

それ以外のケースでは、分裂した Particle を既存の 2 つの Particle の間に挿入するように Edge を繋ぎ、分裂元の Particle と繋がっていた Edge を変換して閉じたネットワークが維持されるように処理します。(図 4.8)



▲図 4.8　分裂した Particle を既存の 2 つの Particle の間に挿入し、閉じたネットワークが維持されるように Edge の接続関係を調整する

こうした分裂処理を繰り返すことによって、閉じたネットワーク構造が成長していきます。

▼ CellularGrowth.compute

```
// 閉じたネットワーク構造への分裂を実行する関数
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))
  {
    // 3 つの Particle で三角形を描くように分裂し、Edge で繋ぐ
    uint cidx = divide_particle(e.a);
    connect(e.a, cidx);
    connect(cidx, e.b);
  }
  else
  {
    // 2 つの Particle の間に Particle を生成し、
    // 一繋ぎになるように Edge を繋ぐ
    float2 dir = pb.position - pa.position;
    float2 offset = normalize(dir) * pa.radius * 0.25;
    uint cidx = divide_particle(e.a, offset);

    // 親 Particle と分裂した子 Particle を結ぶ
    connect(e.a, cidx);

    // 元の 2 つの Particle を結んでいた Edge を、
    // 分裂した子 Particle を結ぶ Edge に変換する
    InterlockedAdd(_Particles[e.a].links, -1);
    InterlockedAdd(_Particles[cidx].links, 1);
    e.a = cidx;
  }

  _Edges[idx] = e;
}

...
```

```
// 閉じたネットワーク構造に分裂するパターン
THREAD
void DivideEdgesClosed(uint3 id : SV_DispatchThreadID)
{
  if (id.x >= _DivideCount)
    return;

  // 分裂可能な Edge の index を取得
  uint idx = _DividablePoolConsume.Consume();
  divide_edge_closed(idx);
}
```

Edge の引き合い

自然に存在する多くの細胞は他の細胞とくっつきあう性質を持ちます。こうした性質を模倣するため、Edge は繋がった 2 つの Particle をバネのように引き合います。

1. Edge ごとに 2 つの Particle を引き付けるバネの力を計算
2. Particle ごとに接続した Edge が持つ力を加える

という流れでバネの引き合いを実現しています。

▼ CellularGrowth.cs

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

...

protected void UpdateEdgesKernel()
{
    // Edge ごとにバネが引き合う力を計算する
    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()
{
    // Particle ごとに Edge が持つバネの力を加える
    var kernel = compute.FindKernel("SpringEdges");
    compute.SetBuffer(
        kernel, "_Particles",
```

```
        particlePool.ObjectPingPong.Read
    );
    compute.SetBuffer(kernel, "_Edges", edgePool.ObjectBuffer);

    Dispatch1D(kernel, count);
}
```

以下がカーネルの中身になります。

▼ 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];

  // 引き合う力を初期化
  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;
  }

  // 2 つの Particle 間の距離を測り、
  // 離れていたり、近づきすぎていれば引き合う力を加える
  float2 dir = pa.position - pb.position;
  float r = pa.radius + pb.radius;
  float len = length(dir);
  if (abs(len - r) > 0)
  {
    // 適切な距離 (互いの半径の合計) になるように力を加える
    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;

  // 接続数が多いほど、引き合う力を弱める
  float dif = 1.0 / p.links;

  int iidx = (int)idx;

  _Edges.GetDimensions(count, strides);

  // すべての Edge から自身と接続している Particle を探す
  for (uint i = 0; i < count; i++)
  {
    Edge e = _Edges[i];
    if (!e.alive)
      continue;

    // 接続している Edge が見つかったら力を加える
    if (e.a == iidx)
    {
      p.velocity -= e.force * dif;
    }
    else if (e.b == iidx)
    {
      p.velocity += e.force * dif;
    }
  }

  _Particles[idx] = p;
}
```

以上までの処理でネットワークで構成された細胞が成長していく様子を表現することができます。

4.3.3 分割パターンのバリエーション

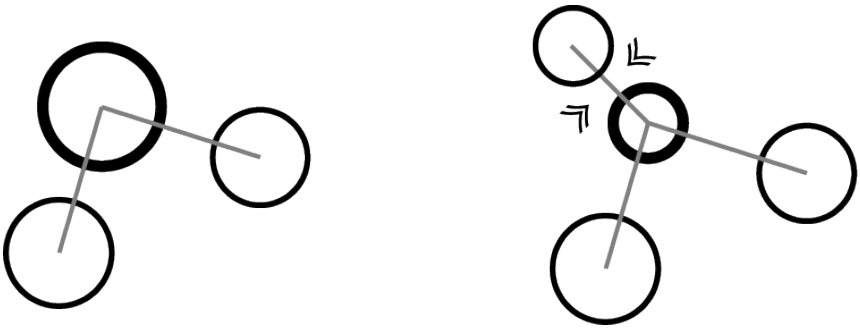
分割させる Edge の判定 (dividable_edge 関数) と分割ロジックを調整することで、様々な分割パターンをデザインすることができます。

サンプルプロジェクトの CellularGrowth.cs では、分裂パターンを enum パラメータによって切り替えられるようにしています。

枝分かれする分裂パターン (DividePattern.Branch)

枝分かれするパターンでは、以下の図 4.9 のような分裂を行います。

分裂した子 Particle は親 Particle とのみ接続します。これを繰り返すだけで枝分かれしたネットワークが成長します。



▲図 4.9　枝分かれする分裂パターン

▼ CellularGrowth.cs

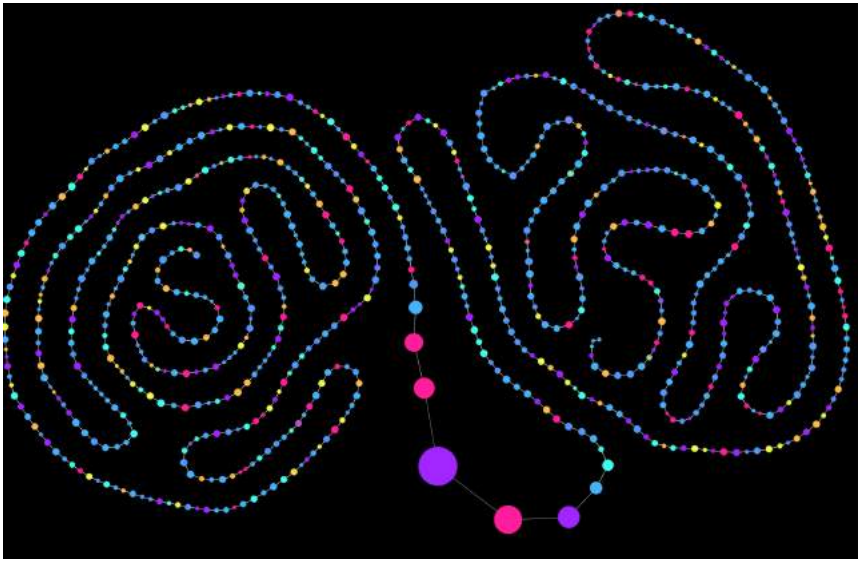
```
protected void DivideEdgesBranchKernel(
    int dividableEdgesCount,
    int maxDivideCount = 16
)
{
    // 枝分かれする分裂パターンを実行
    var kernel = compute.FindKernel("DivideEdgesBranch");
    DivideEdgesKernel(kernel, dividableEdgesCount, maxDivideCount);
}
```

▼ CellularGrowth.compute

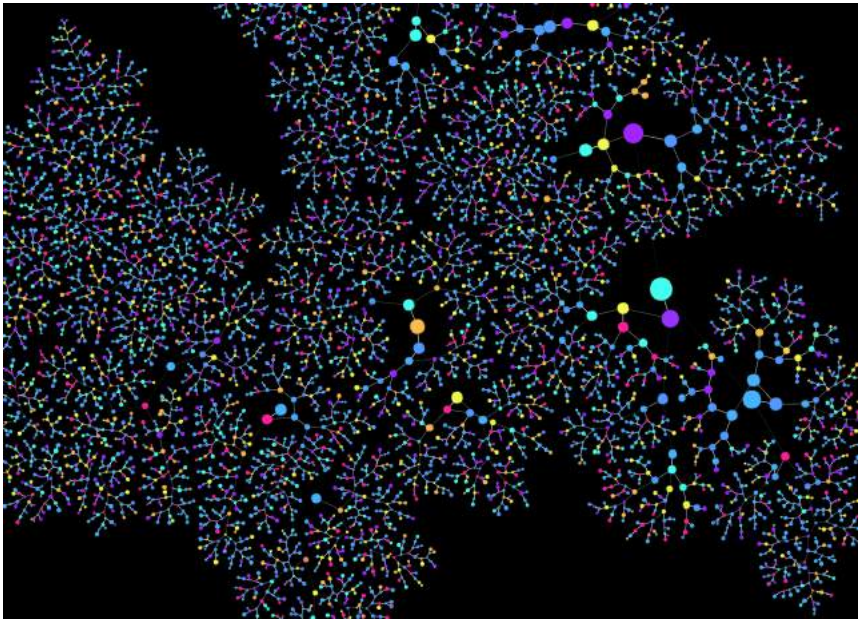
```
// 枝分かれ分裂を実行する関数
void divide_edge_branch(uint idx)
{
  Edge e = _Edges[idx];
  Particle pa = _Particles[e.a];
  Particle pb = _Particles[e.b];

  // 接続数の少ない方の Particleindex を取得
  uint i = lerp(e.b, e.a, step(pa.links, pb.links));

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

...

// 枝分かれ分裂パターン
THREAD
```

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

  // 分裂可能な Edge の index を取得
  uint idx = _DividablePoolConsume.Consume();
  divide_edge_branch(idx);
}
```

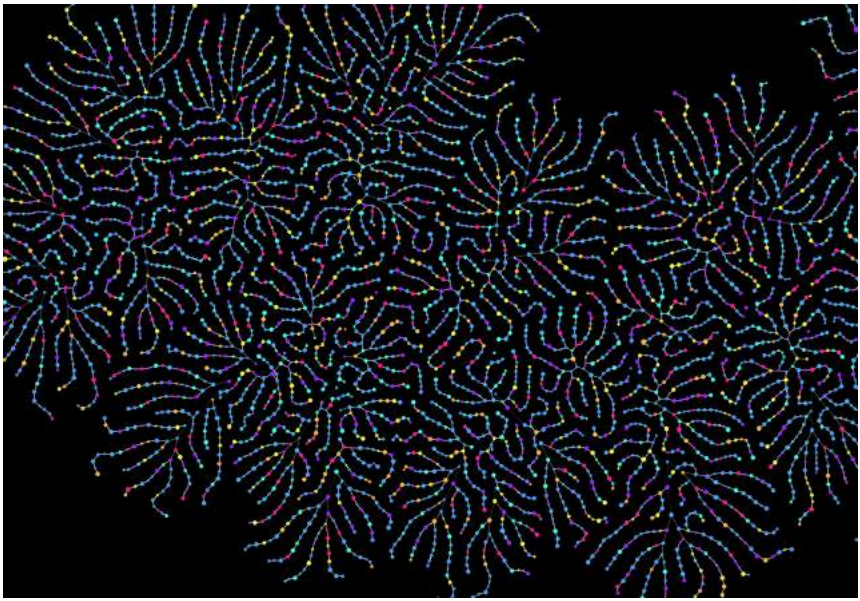
枝分かれするパターンにおいては、分裂する Edge を判定するロジックがビジュアルに大きく影響します。dividable_edge 関数内で参照している Particle の最大接続数 (_MaxLink) の値を変化させることで、枝分かれ具合をコントロールすることができます。



▲図 4.10　_MaxLink に 2 を設定したパターン (DividePattern.Branch)



▲図 4.11　_MaxLink に 3 を設定したパターン (DividePattern.Branch)



▲図 4.12　_MaxLink を 3 に設定してある程度成長させた後、2 に設定して成長を続けさせたパターン (DividePattern.Branch)

4.4　まとめ

本章では、GPU 上で細胞の分裂と成長をシミュレーションするプログラムを紹介しました。

こうした細胞をモチーフとした CG を生成する試みは他にも、Andy Lomas[*6]による Morphogenetic Creations プロジェクトや、学術的なものだと J.A.Kaandorp[*7]による Computational Biology プロジェクトがあり、特に後者のものは生物学に基づいたよりリアルなシミュレーションを行っています。

また、Maxime Causeret[*8]による Max Cooper のミュージックビデオ[*9]が細胞などの有機的なモチーフを使用した素晴らしい映像作品の例として挙げられます。(この映像作品内のシミュレーション部分には Houdini が使われています)

[*6] http://www.andylomas.com/
[*7] https://staff.fnwi.uva.nl/j.a.kaandorp/research.html
[*8] http://teresuac.fr/
[*9] https://vimeo.com/196269431

今回は 2 次元上に分裂・成長するものに留まりましたが、元の iGeo のチュートリアル[*10]にもあるように、本プログラムは 3 次元上に拡張することも可能です。

3 次元への拡張では、3 つの細胞から面を構成し、成長して広がる細胞ネットワークを用いて、グニグニと有機的に成長するメッシュを実現することもできます。3 次元への拡張を行っているサンプルは https://github.com/mattatz/CellularGrowth に上げているので、興味のある方は参考にしてみてください。

4.5 参考

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

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

第 5 章

Reaction Diffusion

5.1 はじめに

自然界には、熱帯魚の横縞模様やサンゴの迷路のようなシワなどさまざまな模様があります。それらの自然界に存在する模様の発生を数式で表したのが、かの天才数学者アラン・チューリングです。彼が導き出した数式で生成される模様のことを「チューリング・パターン」といいます。一般的にこの数式は反応拡散方程式と呼ばれています。この反応拡散方程式をもとに、Unity 上で ComputeShader を使って生物の模様のような絵を作るプログラムを開発します。最初は 2 次元平面上で動作するプログラムを作りますが、最後におまけとして 3 次元空間上で動作するプログラムも紹介します。ComputeShader については、UnityGraphicsProgramming vol.1 の「第 2 章 ComputeShader 入門」を参照してください。

本章のサンプルは

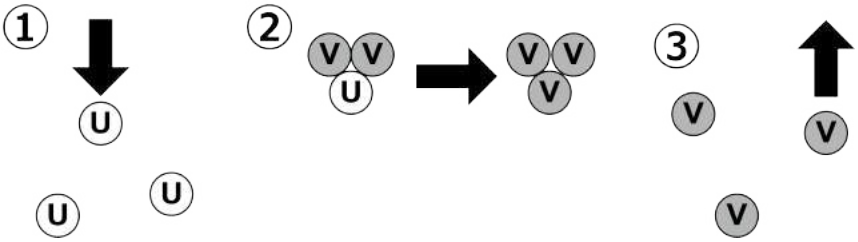
`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`

の「ReactionDiffusion」です。

5.2 Reaction Diffusion とは

Reaction Diffusion（反応拡散系）とは、その名のとおり、空間内に分布された一種あるいは複数種の物質の濃度が、物質が互いに変化し合うような局所的な化学反応（Reaction）と、空間全体に広がる拡散（Diffusion）の、2 つのプロセスの影響によって変化する様子を数理モデル化したものです。今回反応拡散方程式として「Gray-Scott モデル」を採用します。Gray-Scott モデルは、1983 年に P.Gray と S.K.Scott に論文で発表されました。ざっくり説明すると、U と V の 2 つの仮想物質がグリッドの中に満たされた状態で、お互いに反応しあって増減したり、拡散したりすることで、時間とともに空間内の濃度が変化していくことでさまざまなパターンが現れます。

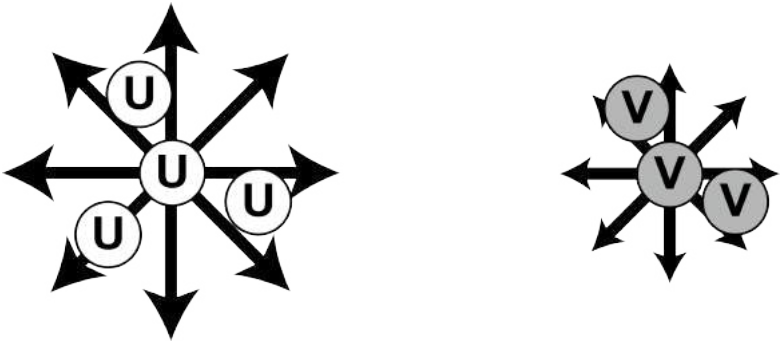
図 5.1 は、Gray-Scott モデルの「反応（Reaction）」の概要図です。



▲図 5.1　Gray-Scott モデルの「反応（Reaction）」の概要図

1. U は、一定の割合で空間内に補充（Feed）されます
2. V が 2 つある時、U と反応（Reaction）してもう 1 つ V を作り出します
3. このままでは V が増え続けてしまうので、一定の割合で V を削除（Kill）します

また、 図 5.2 のように、U と V はそれぞれ異なる速さで隣のグリッドに拡散していきます。



▲図 5.2　Gray-Scott モデルの「拡散（Diffusion）」の概要図

この拡散の速度の差によって U と V の濃度の差が生まれ、パターンが生成されます。これらの U と V の反応と拡散は、次の方程式で表されます。

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

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

この式では、U は u 、V は v で表しています。式は大きく 3 つに分かれています。最初の $Du\Delta u$ と $Dv\Delta v$ は拡散項といい、前半の Du と Dv は、 u と v の拡散の速度の定数です。後半の Δu と Δv はラプラシアンといって、U と V の周囲との濃度差を無くす方向に拡散（Diffusion）する過程を表しています。
2 番目は反応項といい、 uv^2 は U 1 つと V 2 つで反応（Reaction）することで、U が減り、V が増えることを表しています。
3 番目の $+f_{(1-u)}$ は流入項といい、U が減った場合に補充（Feed）される量を表しており、0 に近いほど多く補充され、1 に近いほど補充されなくなります。 $-(f+k)$ は流出項といい、増えた V を一定数減らす（Kill）ことを表しています。
もうちょっと簡単にまとめると、U 1 つと V 2 つで反応して U は減り、V は増えていきます。このままでは U は減り続け、V は増え続ける一方なので、U は $+f_{(1-u)}$ の分だけ補充され、V は $-(f+k)$ の分だけ強制的に減るようになっています。そして、U と V は $Du\Delta u$ と $Dv\Delta v$ によって周囲に拡散していきます。

5.3 Unity での実装

なんとなく方程式の雰囲気がわかったところで Unity での実装の説明に移ります。動作確認できるサンプルシーンは、**ReactionDiffusion2D_1** です。

5.3.1 グリッド構造体の定義

2 次元の平面空間のグリッドの中に、U と V のそれぞれの濃度の値が入っていると仮定します。今回は ComputeShader を使って並列に処理するため、ComputeBuffer でグリッドを管理します。まず、1 グリッドの中の構造体を定義します。

▼ ReactionDiffusion2D.cs

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

5.3.2 初期化

▼ ReactionDiffusion2D.cs

```
/// <summary>
/// 初期化
/// </summary>
void Initialize()
```

```
{
    ...

    int wh = texWidth * texHeight;  // バッファのサイズ
    buffers = new ComputeBuffer[2]; // ダブルバッファリング用の ComputeBuffer の
配列初期化

    for (int i = 0; i < buffers.Length; i++)
    {
        // グリッドの初期化
        buffers[i] = new ComputeBuffer(wh, Marshal.SizeOf(typeof(RDData)));
    }

    // リセット用のグリッド配列
    bufData = new RDData[wh];
    bufData2 = new RDData[wh];

    // バッファの初期化
    ResetBuffer();

    // Seed 追加用バッファの初期化
    inputData = new Vector2[inputMax];
    inputIndex = 0;
    inputBuffer = new ComputeBuffer(
        inputMax, Marshal.SizeOf(typeof(Vector2))
      );
}
```

更新用である ComputeBuffer の **buffers** は 2 次元配列ですが、これは読み込み用と書き込み用に分けるために 2 つ用意しています。というのも、ComputeShader はマルチスレッドで並列に処理されています。今回のように周囲のグリッドを参照して計算結果が変わる処理をする場合、1 つのバッファだと、処理するスレッドの順番によって先に計算し終わったグリッドの値を参照したりして計算結果がかわってきてしまいます。それを防ぐために、読み込み用と書き込み用の 2 つに分けています。

5.3.3 更新処理

▼ ReactionDiffusion2D.cs

```
// 更新処理
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();
}
```

C#側のソースでは、前述の方程式にもあったパラメータを ComputeShader に渡して更新処理を行っています。次に、ComputeShader 内の更新処理について説明します。

▼ ReactionDiffusion2D.compute

```
// 更新処理
[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)
    );
}
```

まさに前述の方程式と同様の計算を行っています。GetIndex() は、2 次元のグリッド座標と 1 次元の ComputeBuffer のインデックスを紐付けるための関数です。

▼ ReactionDiffusion2D.compute

```
// バッファのインデックス計算
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 には 1 フレーム前の計算結果が入っています。そこから u と v を取り出します。LaplaceU と LaplaceV は、自分のグリッドの周囲 8 マスの U と V の濃度を集めるラプラシアン関数です。これによって周囲のグリッドと濃度が平均化されていきます。斜めのグリッドは影響度は低くなるように調整しています。

▼ ReactionDiffusion2D.compute

```
// U のラプラシアン関数
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;
}

// V のラプラシアン関数
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;
}
```

u と v を計算したら、_BufferWrite に書き込みます。saturate は 0〜1 の間でクリップするための保険です。

5.3.4 入力処理

A キーや C キーを押すことで、グリッドに意図的に U と V の濃度差を追加する機能を用意しています。A キーを押すことでランダムな位置に SeedNum 個の点（Seed）を配置します。C キーを押すことで中心に 1 つ点を配置します。

▼ ReactionDiffusion2D.cs

```
/// <summary>
/// Seed の追加
/// </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++;
    }
  }
```

inputData 配列にグリッド上の点の座標を格納しています。

▼ ReactionDiffusion2D.cs

```
/// <summary>
/// Seed 配列を ComputeShader にわたす
/// </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 に、さきほどの点の座標が入った inputeData 配列をセットして、ComputeShader に渡しています。

▼ ReactionDiffusion2D.compute

```
// シードの追加
[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;
      }
    }
  }
}
```

C#から渡された inputBuffer の座標を中心に円形になるように v の値を 1 にしています。

5.3.5 RenderTexture に結果を書き込み

更新したグリッドはただの配列なので、可視化のために RenderTexture に書き込んで画像にします。RenderTexture には u と v の濃度差を書き込みます。
まずは RenderTexture を作成します。1 ピクセルに書き込む情報は濃度差だけなので、RenderTextureFormat は RFloat にしておきます。RenderTextureFormat.RFloat は 1 ピクセルに付き float 1 つ分の情報が書き込める RenderTexture の形式です。

▼ ReactionDiffusion2D.cs

```
/// <summary>
/// RenderTexture 作成
/// </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;
}
```

続いて ComputeShader に RenderTexture を渡して書き込む C#側の処理です。

▼ ReactionDiffusion2D.cs

```
/// <summary>
/// ReactionDiffusion の結果をテクスチャに書き込み
/// </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);
}
```

こちらは、ComputeShader 側の処理です、グリッドのバッファから u と v の濃度差を求めて、テクスチャに書き込んでいます。

▼ ReactionDiffusion2D.compute

```
// テクスチャ書き込み用の値計算
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);
}

...

// テクスチャに描画
[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 描画

通常の Unlit Shader を改修して、前項で作ったテクスチャの明度をもとに、2 色を間を補間しています。

▼ ReactionDiffusion2D.cs

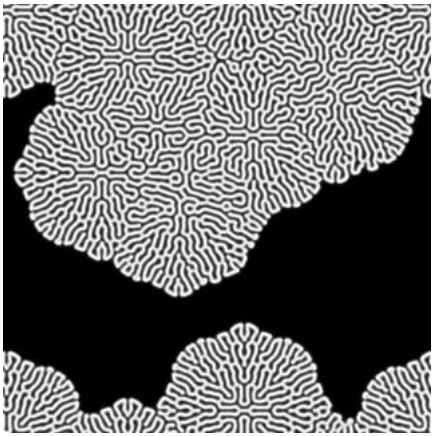
```
/// <summary>
/// マテリアルの更新
/// </summary>
void UpdateMaterial()
```

```
{
  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;
}
```

実行すると画面上に生物のような模様が広がっていくはずです。



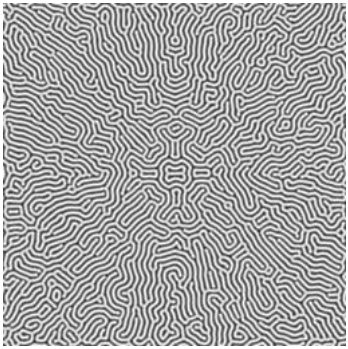
▲図 5.3　シミュレーションの様子

5.4　パラメータを変えてみる

Feed と Kill のパラメータを少し変えてみるだけで、さまざまな模様が浮かび上がります。ここでいくつかパラメータの組み合わせを紹介します。

5.4.1 サンゴのような模様

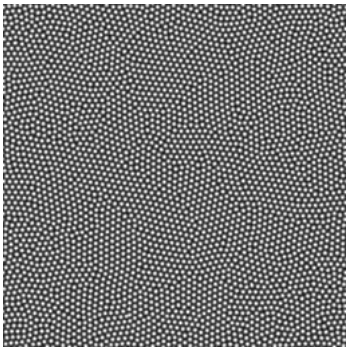
Feed:0.037 / Kill:0.06



▲図 5.4　サンゴのような模様

5.4.2 つぶつぶ模様

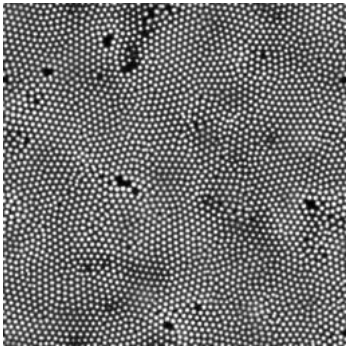
Feed:0.03 / Kill:0.062



▲図 5.5　つぶつぶ模様

5.4.3 つぶつぶが消滅と分裂を繰り返す模様

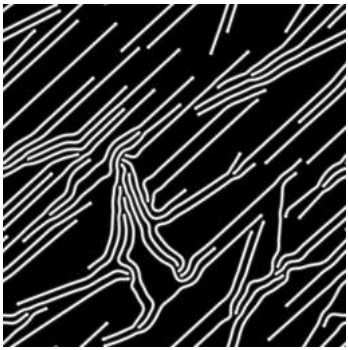
Feed:0.0263 / Kill:0.06



▲図 5.6 つぶつぶが消滅と分裂を繰り返す模様

5.4.4 まっすぐ伸びて、ぶつからないように避ける模様

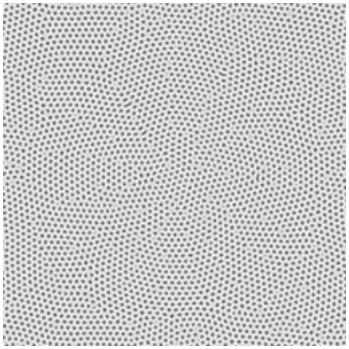
Feed:0.077 / Kill:0.0615



▲図 5.7 まっすぐ伸びて、ぶつからないように避ける模様

5.4.5 ぷつぷつ穴模様

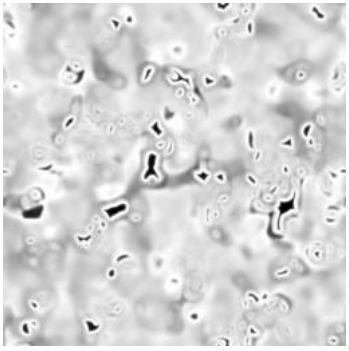
Feed:0.039 / Kill:0.058



▲図 5.8　ぷつぷつ穴模様

5.4.6 常にうねうねして安定しない模様

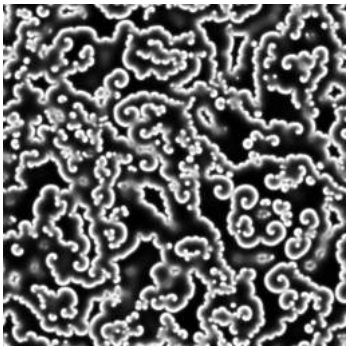
Feed:0.026 / Kill:0.051



▲図 5.9　常にうねうねして安定しない模様

5.4.7 波紋のように広がり続ける模様

Feed:0.014 / Kill:0.0477



▲図 5.10 波紋のように広がり続ける模様

5.5 おまけ：Surface Shader 対応版

ここで、Surface Shader を利用した Unity ならではのきれいな質感表現をしたサンプルを紹介します。動作確認できるサンプルシーンは、**ReactionDiffusion2D_2** です。

5.5.1 通常版との変更点

ReactionDiffusion の処理自体は通常版と同じですが、描画用のテクスチャの作成時に、立体感を出すためのノーマルマップも作成しています。また、結果のテクスチャは RenderTextureFormat.RFloat でしたが、ノーマルマップは XYZ 方向のノーマルベクトルを格納するため、RenderTextureFormat.ARGBFloat で作成しています。

▼ ReactionDiffusion2DForStandard.cs

```
void Initialize()
{
  ...
  heightMapTexture = CreateRenderTexture(texWidth, texHeight,
    RenderTextureFormat.RFloat);        // 高さマップ用テクスチャ作成
  normalMapTexture = CreateRenderTexture(texWidth, texHeight,
```

```
    RenderTextureFormat.ARGBFloat);     // ノーマルマップマップ用テクスチャ作成
  ...
}

/// <summary>
/// RenderTexture 作成
/// </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);  // ノーマルマッ
プ用テクスチャセット
  cs.Dispatch(kernelDraw,
    Mathf.CeilToInt((float)texWidth / THREAD_NUM_X),
    Mathf.CeilToInt((float)texHeight / THREAD_NUM_X),
    1);
}
```

ComputeShader 内では、周囲のグリッドとの濃度差から傾きを求めてノーマルマップ用のテクスチャに書き込んでいます。

▼ 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;
}

...
```

```
// テクスチャに描画
[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);
}
```

作成した 2 枚のテクスチャを Surface Shader に渡して模様を描画します。Surface Shader は、Unity の物理ベースレンダリングを簡単に使えるようにラッピングされたシェーダーで、surf 関数の中で **SurfaceOutputStandard** 構造体に必要なデータを代入して出力するだけで、自動的にライティングしてくれます。

▼ SurfaceOutputStandard 構造体の定義

```
struct SurfaceOutputStandard
{
    fixed3 Albedo;      // ベース（ディフューズかスペキュラー）カラー
    fixed3 Normal;      // 法線
    half3 Emission;     // 発光色
    half Metallic;      // 0=非メタル, 1=メタル
    half Smoothness;    // 0=粗い, 1=滑らか
    half Occlusion;     // オクルージョン（デフォルト 1）
    fixed Alpha;        // 透明度のアルファ
};
```

▼ ReactionDiffusion2DStandard.shader

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

  float2 uv = IN.uv_MainTex;

  // 濃度取得
  half v0 = tex2D(_MainTex, uv).x;

  // 法線取得
  float3 norm = UnpackNormal(tex2D(_NormalTex, uv));

  // A と B の境界の値を出す
  half p = smoothstep(_Threshold, _Threshold + _Fading, v0);

  o.Albedo = lerp(_Color0.rgb, _Color1.rgb, p);         // ベース色
  o.Alpha = lerp(_Color0.a, _Color1.a, p);              // アルファ値
  o.Smoothness = lerp(_Smoothness0, _Smoothness1, p);   // スムースネス
  o.Metallic = lerp(_Metallic0, _Metallic1, p);         // メタリック
  o.Normal = normalize(float3(norm.x, norm.y, 1 - _NormalStrength));  // 法
線
```

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

Unity のビルトイン関数の **unpackNormal** 関数を使ってノーマルマップから法線を取得します。また、濃度差の割合から **SurfaceOutputStandard** の各種色や質感を設定しています。

実行すると次のような模様ができ上がるはずです。

▲図 5.11　SurfaceShader 版

ノーマルマップによって、立体感が生まれています。また、シーン上の RGB 3 色

のポイントライトの光沢も表現されています。

5.6 3次元への拡張

今まで2次元の平面上でのシミュレーションだった Reaction Diffusion を3次元に拡張してみましょう。基本的な流れは2次元のときと同じですが、次元が1つ増えているので RenderTexture や ComputeBuffer の作り方、ラプラス演算の仕方が少し変わっています。動作確認できるサンプルシーンは、**ReactionDiffusion3D** です。

5.6.1 バッファの初期化部分

濃度差の書き込み先の RenderTexture を2次元から3次元にするため、初期化処理をいくつか追加しています。

▼ 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;
}
```

まず、tex.volumeDepth に Z 方向の深さを入れています。それから、tex.dimension に UnityEngine.Rendering.TextureDimension.Tex3D を入れています。これは、RenderTexture が3次元のボリュームテクスチャであることを指定するための設定です。これで RenderTexture が3次元のボリュームテクスチャになりました。同様に Reaction Diffusion のシミュレーション結果を格納する ComputeBuffer も3次元化します。こちらは単純に幅×高さ×深さのサイズを確保するだけです。

▼ 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 シミュレーションの３次元化

続いて ComputeShader 側の変更点です。まず、結果の書き込み用の RenderTexture が３次元になったため、ComputeShader 側のテクスチャの定義が **RWTexture2D<float>**から **RWTexture3D<float>**に変わります。

▼ ReactionDiffusion3D.compute

```
RWTexture3D<float> _HeightMap; // ハイトマップ
```

次にラプラシアン関数の３次元化です。３×３×３の合計２７マスを参照するように変更しています。ちなみに laplacePower の影響度はなんとなくで割り出した値です。

▼ ReactionDiffusion3D.compute

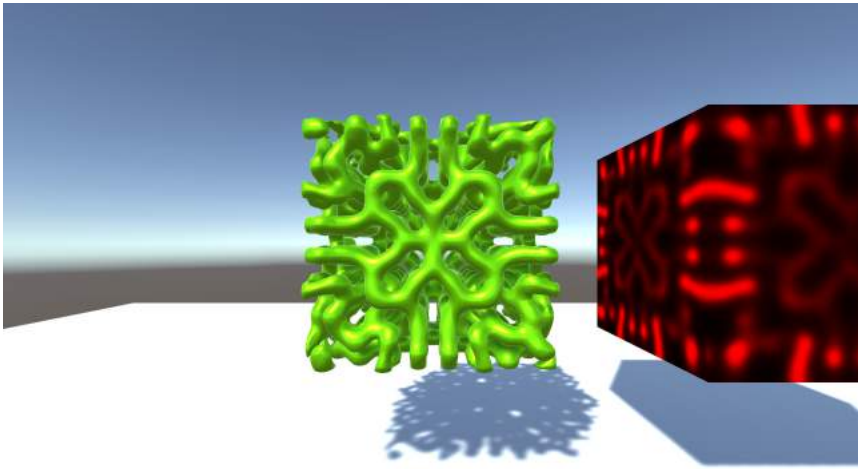
```
// ラプラシアン関数で参照する周囲のインデックス計算用テーブル
static const int3 laplaceIndex[27] = {
  int3(-1,-1,-1), int3(0,-1,-1), int3( 1,-1,-1),
  int3(-1, 0,-1), int3(0, 0,-1), int3(1, 0,-1),
  int3(-1, 1,-1), int3(0, 1,-1), int3(1, 1,-1),

  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),
};

// ラプラシアンの周囲のグリッドの影響度
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
};
```

```
// バッファのインデックス計算
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;
}

// U のラプラシアン関数
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;
}

// V のラプラシアン関数
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 描画処理

シミュレーション結果の RenderTexture が 3 次元のボリュームテクスチャになっているので、今までのように Unlit Shader や Surface Shader にテクスチャを貼り付けても正常に表示されません。サンプルではマーチングキューブス法という手法を使ってポリゴンを生成して描画していますが、紙面の都合上、実装についての説明は省略させていただきます。マーチングキューブス法の解説については、申し訳ありませんが Unity Graphics Programming Vol.1 の「第 7 章 雰囲気で始めるマーチングキューブス法入門」を参照してください。他にも、レイマーチングを使ったボリュームレンダリングで描画する方法もあります。凹さんのブログに非常にわかりやすい実

装[*1]が紹介されているので、ぜひとも参考にしてください。



▲図 5.12　3 次元版 Reaction Diffusion

5.7　まとめ

Gray-Scott モデルを使って生物のような模様を作る方法を紹介しました。Feed と Kill のパラメータを少し変えるだけで全然違う模様ができるので、夢中になるとあっという間に時間が過ぎてしまうので注意しましょう（※個人差があります）
また、Reaction Diffusion を使った作品には、Nakama Kouhei さんの「DIFFUSION」[*2]や Kitahara Nobutaka さんの「Reaction-Diffusion」[*3]があります。みなさんも Reaction Diffusion の不思議な魅力に取り憑かれてみませんか？

5.8　参考

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

[*1] 凹み Tips http://tips.hecomi.com/entry/2018/01/05/192332
[*2] DIFFUSION https://vimeo.com/145251635
[*3] Reaction-Diffusion https://vimeo.com/176261480

第 6 章

Strange Attractor

6.1 はじめに

本章では「Strange Attractor」と呼ばれる、状態が特定の微分方程式や差分方程式によって、非線形のカオス的な振る舞いを見せる現象の可視化を Unity と GPU 演算を使って開発していきます。
本章のサンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`
の「StrangeAttractors」です。

6.1.1 実行環境

- ComputeShader が実行できる、シェーダーモデル 5.0 対応環境
- 執筆時環境、Unity2018.2.9f1 で動作確認済み

6.2 Strange Attractor とは

散逸系（エネルギー非保存。特定の入力と開放がある非均衡系）の運動でありながら、時間経過に伴って安定した軌道を保つ状態を「Attractor」と言います。
その中でも初期状態の僅かな差が時間経過にしたがって増幅され、カオス的な振る舞いを見せるものを「Strange Attractor」と呼びます。

本章ではその中から題材として、「[*1]Lorenz Attractor」と「[*2]Thomas' Cyclically Symmetric Attractor」を取り上げていきたいと思います。

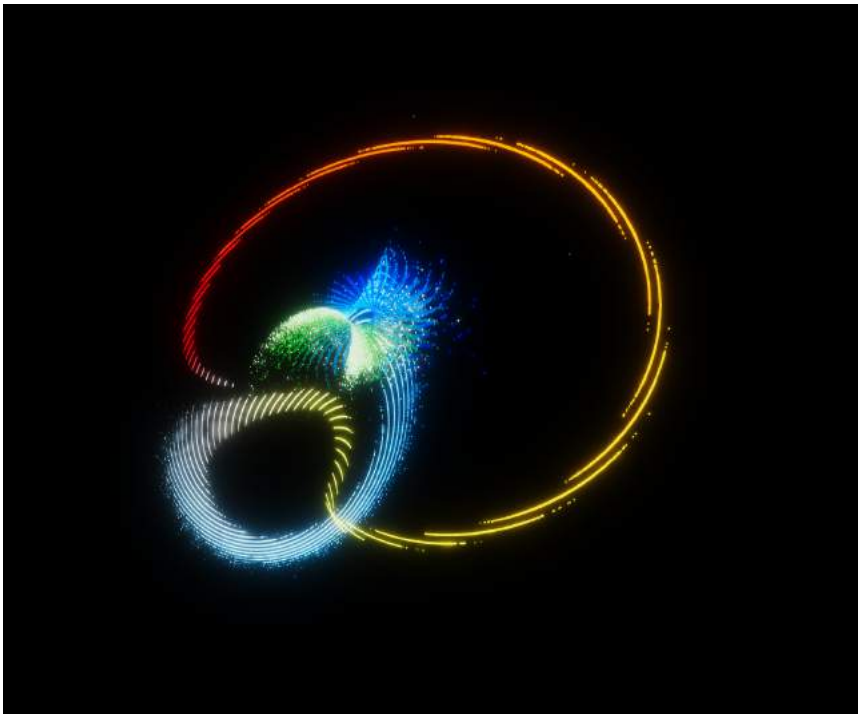
[*1] Lorenz, E. N.: Deterministic Nonperiodic Flow, Journal of Atmospheric Sciences, Vol.20, pp.130-141, 1963.

[*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.

6.3 Lorenz Attractor

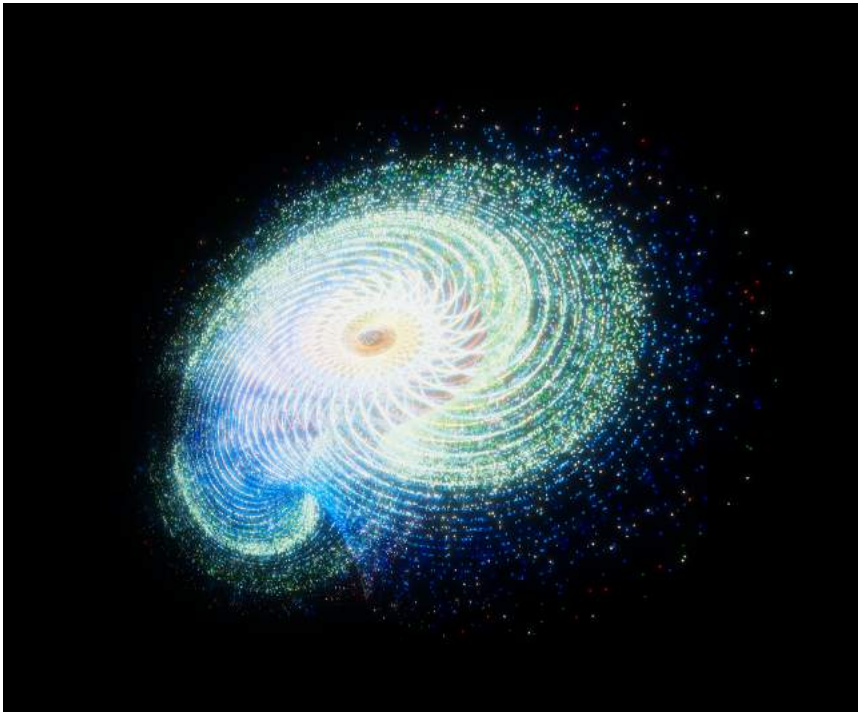
バタフライ効果という現象をご存知でしょうか。これは気象学者の Edward N Lorenz が 1972 年にアメリカ科学振興協会で行った講演のタイトル「[*3]ブラジルの 1 匹の蝶の羽ばたきはテキサスで竜巻を引き起こすか？」に由来する言葉です。
この言葉は、初期値の僅かな差が数学的に必ずしも似た結果をもたらさず、カオス的に増幅され予測困難な振る舞いを見せる現象をあらわしています。

この数学的性質を指摘した Lorenz が 1963 年に発表したのが「Lorenz Attractor」です。



▲図 6.1　Lorenz attractor の初期状態

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



▲図 6.2　Lorenz attractor の中期

6.3.1　Lorenz 方程式

Lorenz 方程式は次の非線形常微分方程式で表されます。

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= -px + py \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= -xz + rx - y \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - bz\end{aligned}$$

上記の方程式の p、r、b の各変数において、p=10、r=28、b=8/3 と定めることにより、「Strange Attractor」としてカオス的に振る舞う状態になります。

6.3.2 Lorenz Attractor の実装

それでは Lorenz 方程式をコンピュートシェーダーによって実装してみましょう。まずはコンピュートシェーダー内で演算したい構造体を定義します。

▼ 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;
    }
}
```

なお、この構造体は今後複数の Strange Attractor で汎用的に使う予定ですので、抽象クラスの StrangeAttractor.cs 内に定義しています。

次に ComputeBuffer の初期化処理を行います。

▼ 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);
}
```

抽象クラスの StrangeAttractor.cs で定義した抽象メソッド InitializeComputeBuffer を LorenzAttrator.cs にて実装しています。
Unity のインスペクターにてグラデーションやエミッターサイズ、粒子サイズを調整したいので、インスペクターに露出した gradient や emitterSize、particleSize で Params 構造体を初期化し、ComputeBuffer の変数、cBuffer に SetData します。
今回はパーティクルの id 順で、少しずつ遅延させて速度ベクトルを適用していきたいと思っていますので、パーティクル id 順でグラーデーションカラーをつけています。
「Strange Attractor」はものによって初期ポジションがその後の振る舞いに大きく関係してきますので、色々な初期ポジションを試していただきたいのですが、本サンプルでは球体を初期形状とします。

次に、LorenzAttrator の変数 p、r、b をコンピュートシェーダーに渡します。

▼ 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);
}
```

次にコンピュートシェーダー側でエミット時のパーティクルの状態を初期化します。

▼ 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;
}
```

Emit メソッドで初期化を行なっています。p.life はパーティクルの発生してからの時間を管理しており、初期値の時点で id 毎に微量の遅延を設けています。
これはパーティクルが一斉に同じ軌道を描くのを簡易に防ぐためです。また、id 毎のグラデーションカラーを設定していますので、カラーをきれいに見せるためにも役立ちます。
ここでは、パーティクルサイズの p.size を初期段階で 0 にしていますが、これは発生した瞬間のパーティクルを不可視にすることによって、自然な吹き出しにする為です。

次にイテレーション部分を見ていきます。

▼ LorenzAttractor.compute

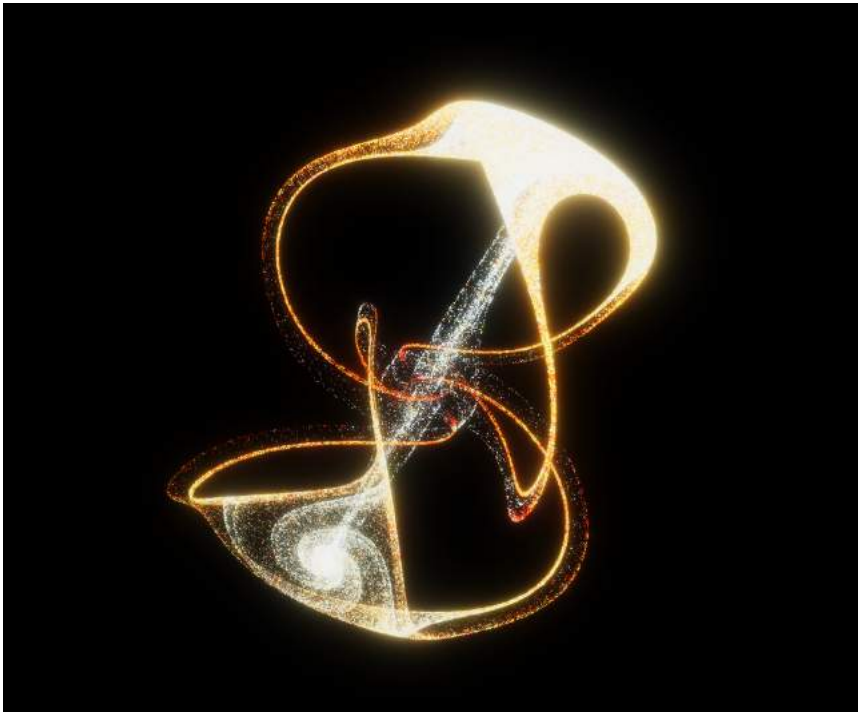
```
#define DT 0.022

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

//Lorenz 方程式の演算部分です。
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;
    //速度ベクトルのベクトル長を 0 から 1 でクランプしつつ、サイズに掛ける事によってス
タートを自然に見せています。
    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;
}
```

上記の LorenzAttractor メソッドが「Lorenz 方程式」の演算部分になります。微量なデルタタイムでの x,y,z の速度ベクトルを演算し、最後にデルタタイムを掛け、移動量を導き出しています。
経験上、コンピュートシェーダー内で形状に関わる微分演算を行う際は、Unity からのデルタタイムを送らずに、フレームレート差から独立した固定値のデルタタイムを使用した方が安定した形状を保ちます。
これはフレームレートが落ちすぎた場合に、Unity の Time.deltaTime の数値が微分演算を行うにしては大きくなりすぎることがある為です。デルタ幅が大きくなれば、それだけ演算結果が前回の物と比べ荒い大雑把な形状になってしまいます。
また、別の理由として、「Strange Attractor」はその方程式によっては、デルタタイムの取り方によって完全な収束もしくは無限大の発散をしてしまう場合がある為です。
これら 2 つの理由から、今回 DT は定義済みの物を使用しています。

6.4 Thomas' Cyclically Symmetric Attractor

次に生物学者の Ren é Thomas 氏により発表された「Thomas' Cyclically Symmetric Attractor」の実装をしていきたいと思います。
初期値に左右されず、時間経過にしたがって安定状態になり、形状も非常にユニークな物になっています。



▲図 6.3　Thomas' Cyclically Symmetric Attractor の安定期

6.4.1　Thomas' Cyclically Symmetric 方程式

方程式は次の非線形常微分方程式で表されます。

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sin y - bx$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \sin z - by$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = \sin x - bz$$

上記の方程式の変数 b において、 $b \simeq 0.208186$ として定めた場合、カオス的に「Strange Attractor」として振る舞い、 $b \simeq 0$ と定めた場合は空間を漂う形となります。

6.4.2 Thomas' Cyclically Symmetric Attractor の実装

それでは「Thomas' Cyclically Symmetric 方程式」をコンピュートシェーダーによって実装してみましょう。
前述の「Lorenz Attractor」の実装と共通する部分がある為、パラメーターの構造体や手続き的な部分は継承し必要な部分のみ取り上げます。
まずは、CPU 側でカラーと初期ポジションをオーバーライドし定めます。

▼ 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);
}
```

今回は色をきれいに見せる為に、内側から外側にいくにつれてマントル的にグラデーションカラーをつけた球体を初期ポジションとして定めています。

次にエミット時とイテレーション時のコンピュートシェーダーのメソッドを見ていきます。

▼ 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;
}
```

ThomasAttractor メソッドが「Thomas' Cyclically Symmetric 方程式」の演算部分になります。
また、Emit 時の実装は、LorenzAttrator と違い、今回は初期ポジションをあえて見せたいので、初期サイズからターゲットサイズに設定しています。
その他はほぼ同じ実装になります。

6.5 まとめ

本章では、「Strange Attractor」をコンピュートシェーダーを用いて GPU 実装する例をご紹介しました。
「Strange Attractor」にはさまざまな種類があり、実装においても比較的少ない演算でカオス的な振る舞いを見せる為、グラフィックスにおいても有用なアクセントになるのではないでしょうか。
他にも、「[*4]UedaAttractor」と呼ばれる 2 次元運動のものや、「[*5]AizawaAttractor」のようなスピン運動を見せるもの等多種多様ありますので、もし興味がございましたらぜひ挑戦してみてください。

6.6 参考

- http://paulbourke.net/fractals/lorenz/

[*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

- 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, Ren é（ 1999）. "Deterministic chaos seen in terms of feedback circuits: Analysis, synthesis, 'labyrinth chaos'". Int. J. Bifurcation and Chaos. 9 （10）: 1889 - 1905.

第 7 章

Portal を Unity で実装してみた

7.1 はじめに

Portal[*1]というゲームをご存知でしょうか？ 2007 年に Valve 社から発売されパズルアクションゲームです。神ゲーです。特徴はポータルと呼ばれる穴で、２箇所の穴がワームホームでつながっているかのように向こう側の景色を見ることができオブジェクトや自キャラがくぐってワープすることができます。要はどこでもドアです。ポータルガンと呼ばれる銃で平面にポータルを設置することができ、これを駆使してゲームを進めていきます。本章はこのポータルの機能を簡易的ながら Unity で実装してみた記事になります。サンプルは
`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`
の「PortalGateSystem」になります。

7.2 プロジェクトの概要

ポータルで遊ぶ場として必要な要素を考えます。

- 移動できる自キャラ
- フィールド
- ポータルガン（指定位置にポータルを出現させる機能）

あたりは欲しいです。自キャラはポータルの向こう側に見えることがあるので一人称視点ですが全身のモデルが必要になります。今回は Unity 社から配布されている **Adam**[*2]のモデルを使わせてもらいました。また自キャラ以外にも物体がワープするさまを見たいので E ボタンで赤いボールを打ち出せるようにしています。

[*1] https://ja.wikipedia.org/wiki/Portal_(%E3%82%B2%E3%83%BC%E3%83%A0)
[*2] https://assetstore.unity.com/packages/essentials/tutorial-projects/adam-character-pack-adam-guard-lu-74842

操作方法としては次のようにしました。

- 移動： WASD キー、もしくは矢印キー（シフト押しながらでダッシュ）
- 視点移動： マウス移動
- ポータル発射： マウス左右クリック
- ボール発射： E キー
- ジャンプ： スペースキー

なお以降ではワープする穴をゲートと呼んでいます。ソースコード上では **PortalGate** というクラス名になっています。

7.2.1 自キャラ

自キャラは Unity の Standard Assets [*3]を改造する形で作りました。FirstPersonCharacter の制御を改造して使いつつ、ThirdPersonChracter のアニメーションを使用する感じです。視界（メインカメラ）に自キャラ自身が写ってしまうとポリゴンがめり込んだり綺麗ではないので **Player** レイヤーを設けてメインカメラでは **Player** レイヤーをカメラの CullingMask に設定して映らないようにしています。

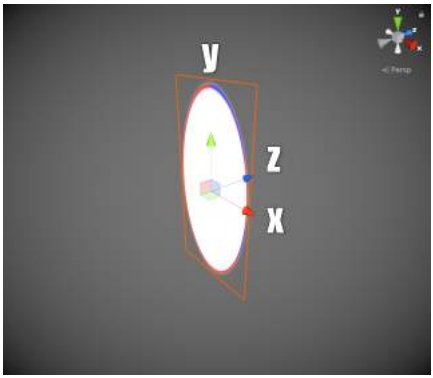
7.2.2 フィールド

フィールドは **unity3d-jp** さんのレベルデザイン用アセット **playGROWnd**[*4]を使わせてもらいました。なんとなく雰囲気も Portal っぽいです。今回はあとの処理を簡単にするためにフィールドは直方体の部屋としました。コリジョンもオブジェクトのものはそのまま使わず直方体の各面ごとに透明なコリジョンを置いています。ワープ後オブジェクトが部分的に部屋の外側に存在することもあるので落下防止用に床のコリジョンは部屋よりも広げています。こちらは **StageColl** レイヤーにしています。

7.3 ゲートの生成

それではゲートを実装していきましょう。今回のゲートはローカル座標系で XY 平面上に広がり Z+ 方向を通過する向きとしています。

[*3] https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351
[*4] https://github.com/unity3d-jp/playgrownd



▲図 7.1　ゲートの座標系

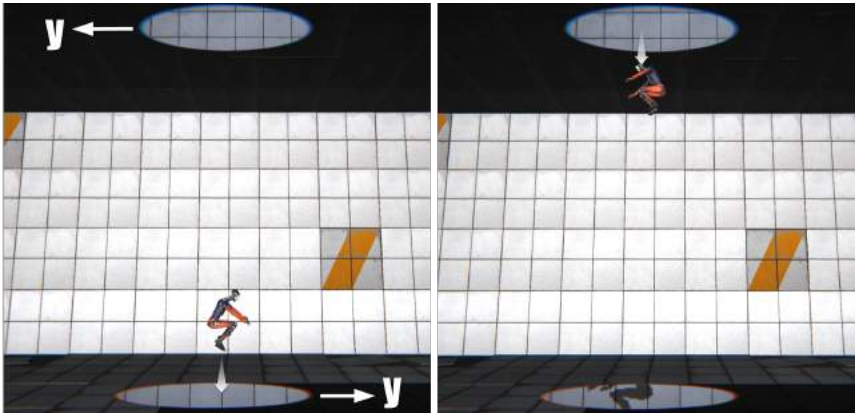
発生はオリジナル Portal を踏襲してマウスをクリックすると視点先の平面にゲートが出現するようにし、左クリックと右クリックでお互いがペアとして繋がります。すでにゲートがある場合は古いゲートはその場で消滅し新しいゲートが開きます。内部的には古いゲートを新しい場所に移動して最初期化しています。

▼ PortalGun.cs

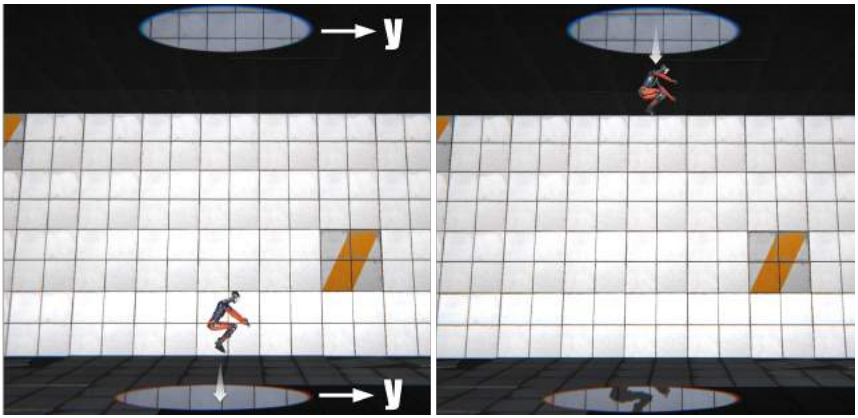
```
void Shot(int idx)
{
    RaycastHit hit;
    if (Physics.Raycast(transform.position,
        transform.forward,
        out hit,
        float.MaxValue,
        LayerMask.GetMask(new[] { "StageColl" })))
    {
        var gate = gatePair[idx];
        if (gate == null)
        {
            var go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

StageColl レイヤーのみの指定で、`transform.forward`方向にレイを飛ばしてヒット確認しています。ヒットがあればゲート操作の処理を行います。まずは既存のゲートがあるのか確認し、なければ生成します。ペアリングもここで行います。あとで使用するためにレイがぶつかったコライダーを `PortalGate.hitColl`にセットし、位置と向きを求めます。位置はぶつかった平面から少し法線方向に浮かせて Z ファイティング対策をしています。向きの求め方が少し変なのにお気づきでしょうか？ Quaternion.LookRotation() のアップベクトルの指定を normal.y の正負で変えています。通常は transform.up でよいのですが、天井にゲートを出したときこのままだと前後（PortalGate の Y 方向）が反転して違和感があるのでこのようにしました。たしかオリジナルの Portal でもこのような挙動だったと思います。



▲図 7.2　アップベクトル処理をしない場合



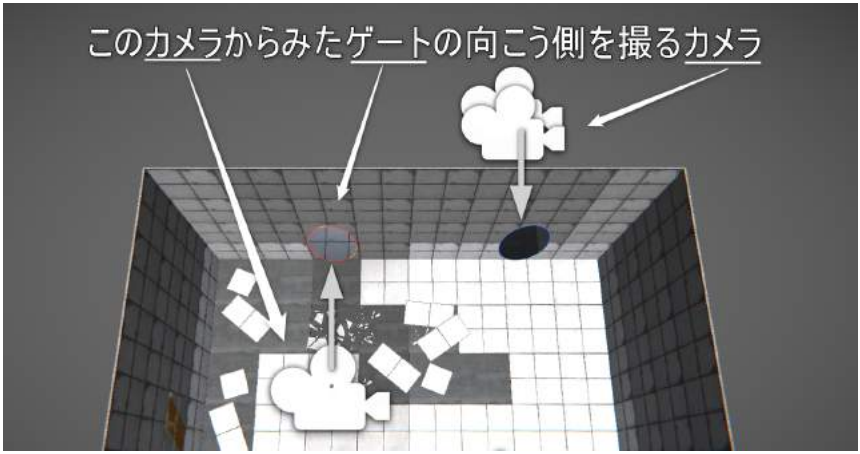
▲図 7.3　アップベクトル処理をした場合

7.4　VirtualCamera

7.4.1　初期化

ゲートが開くとペアになった別のゲート（以降ペアゲートと呼びます）の向こう側が見えるのでこの描画をなんとか実装する必要があります。「向こう側」を描画するための「別のカメラ **(VirtualCamera)** を用意し **RenderTexture** にキャプチャ、それを **PortalGate** に貼り付けてメインカメラで描画」というアプローチにしました。

VirtualCamera はゲートの向こう側の絵をキャプチャするカメラです。



▲図 7.4　VirtualCamera

PortalGate.OnWillRenderObject()が各カメラごとに呼ばれるのでそのタイミングで VirtualCamera が必要であれば生成します。

▼ PortalGate.cs

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

～省略～
}
```

ゲート同士を向かい合わせにすると向こう側の景色にもまたゲートが映り、そのゲートの向こう側にもまたゲートが・・・と、あわせ鏡のように無限に続きます。

▲図 7.5　向かい合わせのゲート

この場合、

1. メインカメラに映るゲートの向こう側の絵を用意する VirtualCamera
2. その VirtualCamera に映るゲートの用の第二世代 VirtualCamera
3. さらにその VirtualCamera に映るゲート用の第三世代 VirtualCamera
4. さらに・・・

と愚直に実装すると VirtualCamera も無限に必要になってしまいます。さすがにそうはいかないので `PortalGate.maxGeneration`で世代数を制限し、それ以上は正確な絵ではないものの 1 フレーム前のテクスチャをゲートに貼り付けることで代用します。

▼ 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 vc;
}
```

VirtualCamera.rootCameraは VirtualCamera の世代をさかのぼって大元になるメインカメラです。他に、親のカメラ、対象のゲート、世代などを設定しています。

▼ 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()で親のカメラからパラメータを引き継いでいます。VirtualCamera には自キャラを映すので CullingMask から **Player** レイヤーを削除しています。また親のカメラより先に絵をキャプチャしたいので parentCamera.depth - 1しています。

当初 Camera.CopyFrom()を使っていたのですがどうも CommandBuffer もコピーしてしまうようで、ポストエフェクトに使っている PostProcessingStack[*5]との併用でエラーが出てしまったので各プロパティごとにコピーするようにしました。

7.4.2 更新

VirtualCamera は処理が軽いほど PortalGate.maxGenerationを多くできるのでできるだ無駄な処理をしないよう少しパフォーマンスに気を使っています。

▼ VirtualCamera.cs

```
private void LateUpdate()
{
    // PreviewCamera などはこのタイミングで null になっているようなのでチェック
    if (parentCamera == null)
    {
```

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

```
            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();
        }
    }
```

こちらのコードを詳しく追っていきます。

カメラの無効化

親のカメラにゲート映っていなければ向こう側の絵を用意する必要がないので VirtualCamera のカメラを無効にします。

▼ 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;
}
```

可視判定は次のようになっています。

1. 向きの判定。ゲートがカメラの方を向いているかをチェックしています。カメラ→ゲート方向とゲートの Z+ 方向の内積の符号で判定しています。
2. 視錐台とバウンディングボックスの可視判定。Unity にはまさのこのための関

数が用意されておりそのまま使えます。ありがたや。

位置と向きの更新

`parentGate.UpdateTransformOnPair()` で「親ゲートに対する親カメラの位置と向きから、親のペアのゲートに対する位置と向きを求めて transform を更新する」処理をしています。

▼ 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);
}
```

実装はこのようになっており、

1. ゲートのローカル座標系に直す
2. gateRot でゲートの手前→奥の向き変換
3. ペアのゲートのローカル座標として扱い
4. ワールド座標系に変換する

という手順になっています。gateRot は

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

と、Y 軸で 180 度回転にしてますが、Z 値が反転すればいいので

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

のような実装でも一応破綻はしないはずです。ただゲートの手前と奥で上方向が反転するのでゲートを通過すると自キャラの頭が地面側になるなど、やはり違和感が出てしまうので Y 軸回転がよさそうです。

カメラパラメータの更新

▼ 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
    // pairGate の奥しか描画しない = nearClipPlane を pairGate と一致させる
    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)
        );
}
```

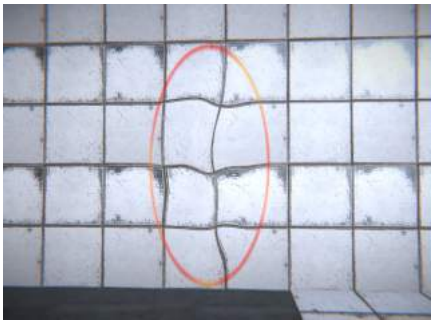
VirtualCamera はできるだけ処理を軽くしたいので視錐台もできるだけ狭くします。VirtualCamera 越しに見たペアゲートの範囲だけ描画できればいいので、ペアゲートのメッシュの頂点をワールド座標にし、`TargetCameraUtility.Update()`でその頂点群が収まるように視錐台と `Camera.rect`を変更しています。

また VirtualCamera とペアゲートの間のオブジェクトは描画しないのでカメラのニアクリップ面をペアゲートと同じ平面にします。この操作は `Camera.CalculateObliqueMatrix()`で行えます。あまりドキュメントがないのでサンプルコードなどからの判断になりますがニアクリップ面をビュー座標系で xyz に法線、w に距離をいれた Vector4 で渡すようです。

7.5 ゲートの描画

状態に合わせて描画するものが違うのですが単一のシェーダーでやりきっています。

- 枠と中身の表示がある
- ゲートが生成、移動された直後は円が広がるアニメーションをする
- まだペアゲートが無いときは背景（既存の壁や床）をモヤモヤさせる（図 7.6）
- ペアゲートができたらモヤモヤから VirtualCamera の絵にフェードインする
- `PortalGate.maxGeneration`に達して VirtualCamera が無い場合は 1 フレーム前の絵を PortalGate に貼り付ける



▲図 7.6　ペアゲートが無いときは背景をモヤモヤ

▼ PortalGate.shader

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

まずは GrabPass[*6]で背景をキャプチャしときます。

7.5.1 頂点シェーダー

▼ PortalGate.shader

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

```
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;
}
```

頂点シェーダーはこんな感じです。スクリーン座標系の位置を 2 つ求めていて、現在のカメラでの位置 clipPosと、メインカメラのもの clipPosOnMainがあります。前者は通常のレンダリングに用い、後者は VirtualCamera でキャプチャした RenderTexture を参照する際に使用します。また GrabPass を用いるときは専用のポジション計算関数がありますのでこれを使います。

7.5.2 フラグメントシェーダー

▼ 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（円の内側率）を求めています。円の中心が 1、円周上が 0、それより外はマイナスになります。_OpenRateで円の開き具合を変えれます。**PortalGate.Open()** で制御しています。

▼ 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);
```

モヤモヤした背景を生成しています。snoiseは include している Noise.cginc で定義されている関数で SimplexNoise です。uv 値と時間で grabUV を揺らしています。insideRate も乗算することで中心付近ほど揺らぎを大きくしています。

▼ PortalGate.shader

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

ゲートの向こう側の絵です。_MainTexには VirutualCamera がキャプチャしたテクスチャが入っており、メインカメラの UV 値で参照しています。

▼ PortalGate.shader

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

bgColor（壁や床）と sideColor（ゲートの向こう側）をミックスしています。_ConnectRateはペアゲートができると 0 から 1 に遷移して、その後はずっと 1 のままです。

▼ 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;
```

最後に枠を計算しています。insideRateの端っこを、_FrameColor0,_FrameColor1を適当にミックスして表示しています。

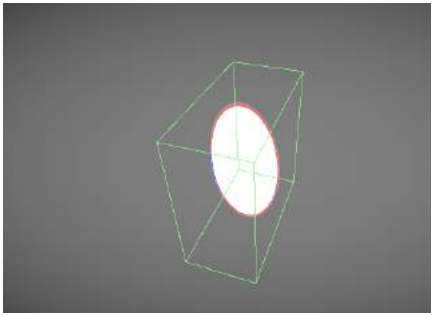
ここまでで見た目は完成しました。次は物理的な挙動のほうに焦点をあわせてみます。

7.6 オブジェクトのワープ

PortalObj コンポーネントでワープまわりの処理を行うようにしました。これがついている GameObject はワープできるようになります。

7.6.1 既存コリジョンの無効化

ゲートが設置された平面はもともととおり抜けできない、つまりコリジョンがあります。ゲートを通るときはこれを無効化しなくてはなりません。実はゲートには前後にわりと大きめに飛び出たコライダーをトリガーとして付けています。PortalObj はこのコライダーをトリガーとして平面とのコリジョン無効化を行っています。



▲図 7.7　ゲートのコライダー

▼ 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()ではなく OnTriggerStay()なのは、まだゲートがひとつでペアがない状態のときに Enter しその後ペアができたときにも対応するためです。まずはトリガーとなったゲートを tougingGatesに登録しておきます。前述の PortalGate.hitCollがやっと出てきました。これと自身のコライダーを Physics.IgnoreCollision()で衝突無視するようにセットしておきます。

OnTriggerExit()で衝突を有効に戻しています。お気づきの方も多いかと思いますが、PortalGate.hitCollは平面全体のコライダーなので実は PortalGate の枠外でも通り抜けできてしまいます。「OnTriggerStay() をキープしている限り」という条件はつくのであまり目立ちませんがちゃんとしたゲートの形でのコリジョンするにはもうちょっと複雑な処理が必要そうです。

7.6.2 ワープ処理

▼ 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はゲートを通過したかどうかの判定に使う Transform です。基本的には PortalObj コンポーネントのついている GameObject のものでいいのですが、自キャラはキャラの中心ではなくカメラが通過した時点でワープしたいので手動で設定できるようにしています。center.positionが z > 0f（ゲートの裏）になっているゲートがないか touchingGatesをチェックしています。もしそのようなゲートが見つかれば PassGate()（ワープ処理）を行います。

また、後述しますがゲート通過直後に PortalObj は重力を無効化しています。これは地面に落ちているオブジェクトの下に別の床に繋がるゲートを開くと、オブジェクトがゲート間を行き来して振動してしまうので通過後は少し慣性をもったような挙動にするために行っています。

▼ 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();
        }
    }
```

ワープ処理はこんな感じになっています。VirtualCamera の位置を求めるときにも使用した PortalGate.UpdateTransformOnPair()で Transform をワープさせます。RigidBodyを持っている場合は速度の向きも変えてやります。fpController（自キャラ操作のスクリプト）も同様です。この辺は大規模化するともっと対応が必要なオブジェクトが出てくるので各スクリプトコールバックを用意して通知するほうがいいかもしれません。

7.6.3 ワープの問題点

今回ワープを実装してみていくつかもうちょっと詰めていかないとなという点がありました。

PortalObj の速度がはやいと一度壁にぶつかる

物理エンジンが衝突判定をしたあと押出処理をする前になんらかの方法でコリジョンを無効化したかったのですがうまい方法が見つかりませんでした。OnTriggerEnter(),OnCollisionEnter()内で Physics.IgnoreCollision()を呼んでも一度衝突したあとから無効化されるようです。おそらく On~Enter()が押出処理後に呼ばれているか Physics.IgnoreCollision()の反映されるタイミングが少し遅いのだと思います。このためトリガーへ Enter するフレームと壁に衝突するフレームが別になるようにトリガーの範囲をかなり大きく飛び出させています。しかしこの方法では限界があり、より高速で移動する PortalObj には対応できていません。もし「こういう方法あるよ！」という方がいましたらぜひご連絡いただきたいです！

本当は途中にコピーを挟んだほうがよい

ワープを「オブジェクトの位置を書き換える」ことで実装しましたが厳密に考えればゲートをくぐってる最中は半分手前で半分向こう側という状態があるはずです。大きいオブジェクトなどを出す場合は目立つのでこのあたりも考える必要がありそうです。さらに手前と向こう両方の衝突物の影響も受ける必要があり、より厳密には物理エンジン内のソルバに介入しないといけなそうな気がします。Unity だと厳しそうなのでうまくごまかす方向が現実的かなーという気がしています。

7.7 まとめ

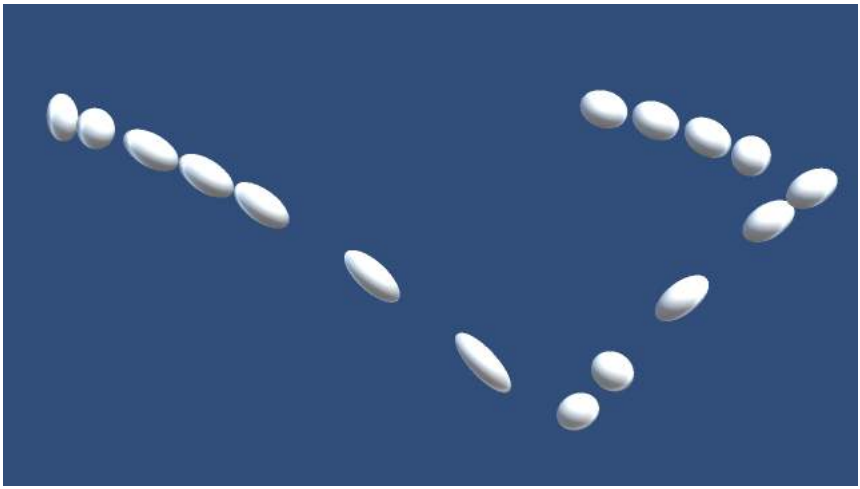
以前からやってみたかった Portal の再現を Unity で挑戦してみました。カメラ重ねればいけるっしょーと気楽にはじめてみたものの思いのほか細かいところで大変なことがわかりました。CG やゲーム技術のなかでもリアル寄りにするものは需要が高く定型化してどんどん手軽になっていっています。現実感が簡単に出せるようになると、どこでもドアのような「いままでありがちだったけど現実味がなく眠っていたアイデア」が今後は意外と新しい体験として活きてきたりするかもしれません。

7.8 参考

- 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`

第 8 章

柔らかな変形を簡単に表現する



▲図 8.1　球体が変形する様子

オブジェクトの柔らかさを表現するとき、バネを模したり、流体や軟体のシミュレーションを計算したりすることがありますが、ここではそれほど大げさな計算をするでもなく、オブジェクトの柔らかな変形を表現してみます。図にあるように手描きアニメのような変形です。

本章のサンプルは

`https://github.com/IndieVisualLab/UnityGraphicsProgramming3`

の「OverReaction」です。

8.1 サンプルシーンの動かし方

「OverReaction」のシーンでは、基本的な変形を確認することができます。マニピュレータや Inspector からオブジェクトを動かして変形するのを確認してください。

「PhysicsScene」のシーンでは、オブジェクトに対して上下左右のキーで力を加えることができます。シーン上にいくつかのオブジェクトを置くと、状況に応じて変形する様子を確認することができます。

8.2 運動エネルギーの算出

変形の規則はいくつも考えられますが、基本的なルールは 3 つ程度でしょう。

1. 変化が大きいほど大きく変形する。
2. 変化がないときは変形が徐々に戻る。
3. 変化の方向が反転するときは変形の方向も反転する。

ここでは特にオブジェクトが動くときを考慮しますから、まずは動きの方向とその大きさを検出していきます。物理法則で用いられる用語と異なりますが、便宜上このパラメータを「運動エネルギー」`moveEnergy` と呼びます。運動エネルギーは方向と大きさで表されるパラメータですから、ベクトルで表現することができます。

※正しい物理学用語としての運動エネルギーは「kinetic energy」です。ここでは特に移動のみを考慮したエネルギーのため「move energy」で命名しています。

ゲームプログラミングにおいては当たり前のように扱われるので今更ですが、オブジェクトの動きは単純に座標の変化を検出します。一点だけ注意したいのは、`Update` ではなく `FixedUpdate` を採用している点です。

▼ 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` は一定時間ごとに呼び出されるメソッドですから、秒間当たりに 2 度 3 度 ~ と呼び出される可能性のある `Update` とは明らかに性質が異なります。こ

れらの違いについて詳細を解説すると本題から外れるので割愛しますが、ここでは Unity の PhysX(物理挙動) を使ったオブジェクトの動きにも対応したいために FixedUpdate を採用しました。また Update のような頻度でメッシュを変形する必要性も特にないです。

ではオブジェクトの動きが座標の変化から算出できたところで、これから運動エネルギーを算出しましょう。運動エネルギーの算出は UpdateMoveEnergy メソッドに実装されています。

▼ 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),
    };
}
```

運動エネルギーは X, Y, Z 方向のそれぞれの成分に分解して算出します。以降に UpdateMoveEnergy の処理を順に追って解説していきます。

まずは現在の動きがない場合を考えます。動きがないとき、現存する運動エネルギーは減衰していきます。

▼ 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);
}
```

現在の動きと直前の動きとが反転しているときは、運動エネルギーを反転します。

▼ OverReaction.cs

```
if (crntMoveSign != prevMoveSign)
{
    return moveEnergy - crntMove;
}
```

現在の動きと運動エネルギーとが反転しているときは、運動エネルギーを小さくします。

▼ OverReaction.cs

```
if (crntMoveSign != moveEnergySign)
{
    return moveEnergy + crntMove;
}
```

上記以外のケース、現在の動きと運動エネルギーが同じ方向のときは、現在の動きと現存する運動エネルギーとを比較して、大きい方を採用します。

ただし運動エネルギーは減衰して小さくなっていきます。また、現在の動きが発生させる新たな運動エネルギーは、変形を演出しやすいように、任意のパラメータを乗算して増強します。

▼ OverReaction.cs

```
if (crntMoveSign < 0)
{
    return Mathf.Min(crntMove * this.deformPower,
                     moveEnergy * this.undeformPower);
}
else
{
    return Mathf.Max(crntMove * this.deformPower,
                     moveEnergy * this.undeformPower);
}
```

これで変形に利用するための運動エネルギーは算出することができました。

8.3 変形エネルギーの算出

次に、算出された運動エネルギーを変形を決定するパラメータに変換します。便宜上、このパラメータを「変形エネルギー」`deformEnergy` と呼びます。変形エネルギーは `UpdateDeformEnergy` メソッドで更新されます。

変形エネルギーの大きさは、そのまま運動エネルギーの大きさと定義することもできますが、変形エネルギーの方向と、オブジェクトが動いている方向とにズレが生じている場合、変形エネルギーは完全にはオブジェクトに伝わりません。また変形エネ

ルギーの方向と、オブジェクトが動いている方向とが反転してしまっているケースも考えられます。

そこで変形エネルギーと現在の動きとの内積から、変形エネルギーがどの程度伝わるのかを求めます。もし完全に方向が一致していれば単位ベクトル同士の内積は 1 となり、ズレの大きさによって徐々に 0 に近づいていきます。さらに反転するときは負の値となります。

▼ OverReaction.cs

```
protected void UpdateDeformEnergy()
{
    float deformEnergyVertical
    = this.moveEnergy.magnitude
    * Vector3.Dot(this.moveEnergy.normalized,
                  this.crntMove.normalized);
    …
```

オブジェクトを垂直方向に変形する力が算出できたので、垂直方向へ変化した分だけ、水平方向を変形します。つまり垂直方向にオブジェクトが伸びたなら水平方向に縮ませようということです。逆に、垂直方向に縮むときは水平方向に伸びるようにします。

垂直方向へどれだけ変形したのかは「垂直方向への変形の大きさ / 最大の変形の大きさ」で算出されます。あとは垂直方向へ変形したのと同じだけ、水平方向にも変形するように算出します。

仮に垂直方向に +0.8 変形したとして、水平方向へは -0.8 変形すればよいので、水平方向の変形エネルギーは 1 - 0.8 = 0.2 となります。また、実際に変形するときの係数として * 0.8 では小さくなりますから、1 を足して * 1.8 となるようにします。

▼ OverReaction.cs

```
protected void UpdateDeformEnergy()
{
…
    float deformEnergyHorizontalRatio
    = deformEnergyVertical / this.maxDeformScale;

    float deformEnergyHorizontal
    = 1 - deformEnergyHorizontalRatio;
…
    deformEnergyVertical = 1 + deformEnergyVertical;
}
```

最後にオブジェクトが進行方向に潰れるケースも考慮しましょう。オブジェクトが進行方向に潰れるケースとは、運動エネルギーと現在の動きが反転しているときで、つまり先の「運動エネルギーと現在の動きの内積」が負のときです。

内積の値が負のとき、垂直方向への変形エネルギーと、水平方向への変形エネルギーを逆転します。

場合分けを考えながら理解するために、次のコードはこれまでの手順を一続きにしました。内積の値が負のとき、deformEnergyHorizontal は 1 より大きい正の値になります。また deformEnergyVertical は、deformEnergyHorizontal の値と逆転して、1 より小さい正の値になります。

▼ 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;
…
```

最後に変形エネルギーが任意に設定される範囲に収まるように値を修正して、変形エネルギーの算出を完了します。

▼ 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 メッシュを変形する

ここでは説明と汎用化のためにスクリプトでメッシュを変形しています。メッシュの変形は DeformMesh メソッドに実装されます。

※行列演算ですから、実用的にはシェーダを使って GPU で処理する方が良いケースが多いでしょう。

得られた変形エネルギー deformEnergy は、運動エネルギー moveEnergy の方向を向くときの伸縮を表すベクトルです。したがって、変形するときは座標を合わせてから変形する必要があります。そのために必要となるパラメータを先に抑えておきます。現在のオブジェクトの回転行列とその逆行列、運動エネルギー moveEnergy の回転行列とその逆行列です。

▼ 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. 現在の回転行列を、回転していないメッシュの頂点に乗算して回転します。
2. 移動方向を示す回転行列の逆行列を、頂点に乗算して回転します。
3. 頂点を deformEnergy にしたがってスケーリングします。
4. 移動方向を示す回転行列を、頂点に乗算して回転を元に戻します。
5. 現在の回転行列の逆行列を、頂点に乗算して回転を元に戻します。

分かりやすく適当な変形エネルギー deformEnergy を仮に与え、順次ソースコードをコメントアウトすると、処理手順が分かりやすくなると思います。

▼ 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 まとめ

非常に簡単な実装だけでオブジェクトを変形することができました。これだけの実装な手軽な反面、見た目に与える印象は大きく変わります。

算出コストはさらに要しますが、発展形として、オブジェクトの回転や拡大縮小にも対応したり、変形の重心を移動したり、スキンメッシュアニメーションに対応することも考えられます。

著者紹介

第 1 章 Baking Skinned Animation to Texture - すぎのひろのり / @sugi_cho

Unity でインタラクティブアートを作る人間。フリーランス。お仕事お待ちしてます => hi@sugi.cc

- https://twitter.com/sugi_cho
- https://github.com/sugi-cho
- http://sugi.cc

第 2 章 Gravitational N-Body Simulation / @kodai100

物理が好きな学生フリーランスエンジニア。VFX プロダクションの TA 業務や、インタラクティブエンジニアリング、VR 関連事業にハマっています。ご連絡は Twitter の DM までお気軽によろしくお願い致します！

- https://twitter.com/kodai100_tw
- https://github.com/kodai100
- http://creativeuniverse.tokyo/portfolio/

第 3 章 Screen Space Fluid Rendering - 大石啓明 / @irishoak

インタラクションエンジニア。インスタレーション、サイネージ、舞台演出、MV、コンサート映像、VJ などの映像表現領域で、リアルタイム、プロシージャルの特性を生かしたコンテンツの制作を行っている。sugi-cho と mattatz とで Aqueduct というユニットを組んで数回活動したことがある。

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

第 4 章 GPU-Based Cellular Growth Simulation - 中村将達 / @mattatz

インスタレーション、サイネージ、Web（フロントエンド・バックエンド）、スマートフォンアプリ等をつくるプログラマー。映像表現やデザインツール開発に興味があります。

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

第 5 章 Reaction Diffusion - @kaiware007

雰囲気でやるインタラクティブ・エンジニア。ジェネ系の動画を Twitter によく上げている。たまに VJ をやる。

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

第 6 章 Strange Attractor - 迫田吉昭 / @sakope

元ゲーム開発会社テクニカルアーティスト。アート・デザイン・音楽が好きで、インタラクティブアートに転向。趣味はサンプラー・シンセ・楽器・レコード・機材いじり。Twitter はじめました。

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

第 7 章 Portal を Unity で実装してみた - 福永秀和 / @fuqunaga

元ゲーム開発者、インタラクティブアートを作ってるプログラマ。そこそこややこしい仕組みやライブラリの設計開発が好き。夜型。

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

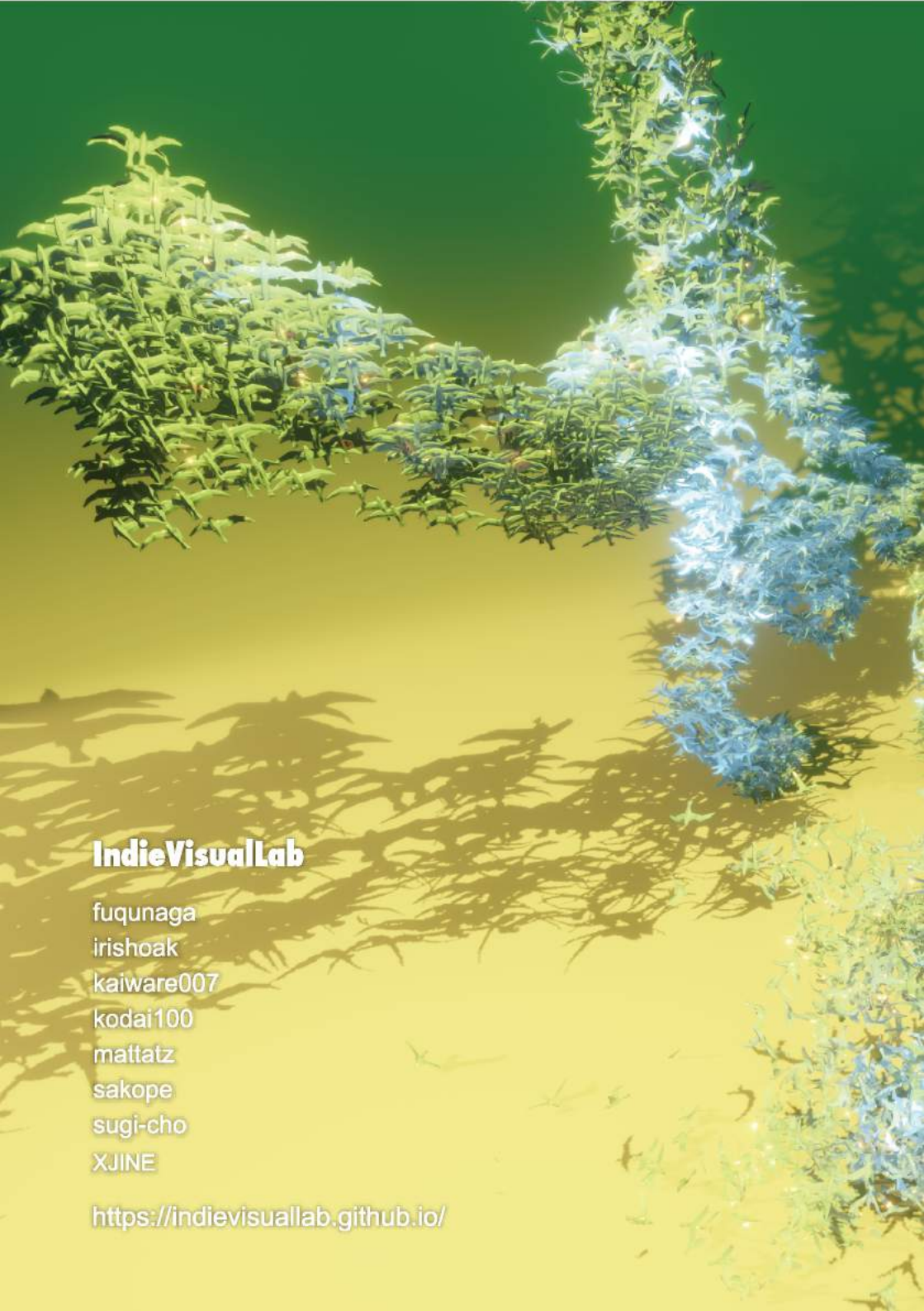
第 8 章 柔らかな変形を簡単に表現する - @XJINE

ついていくのも必至なレベルでボロボロになりながらどうにか生きています。シェーダ入門用の「UnityShaderProgramming」もよろしくお願いいたします。

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

Unity Graphics Programming vol.3

2018 年 10 月 8 日　技術書典 5 版 v1.0.0
著　者　IndieVisualLab
編　集　IndieVisualLab
発行所　IndieVisualLab


IndieVisualLab
fuqunaga
irishoak
kaiware007
kodai100
mattatz
sakope
sugi-cho
XJINE
https://indievisuallab.github.io/