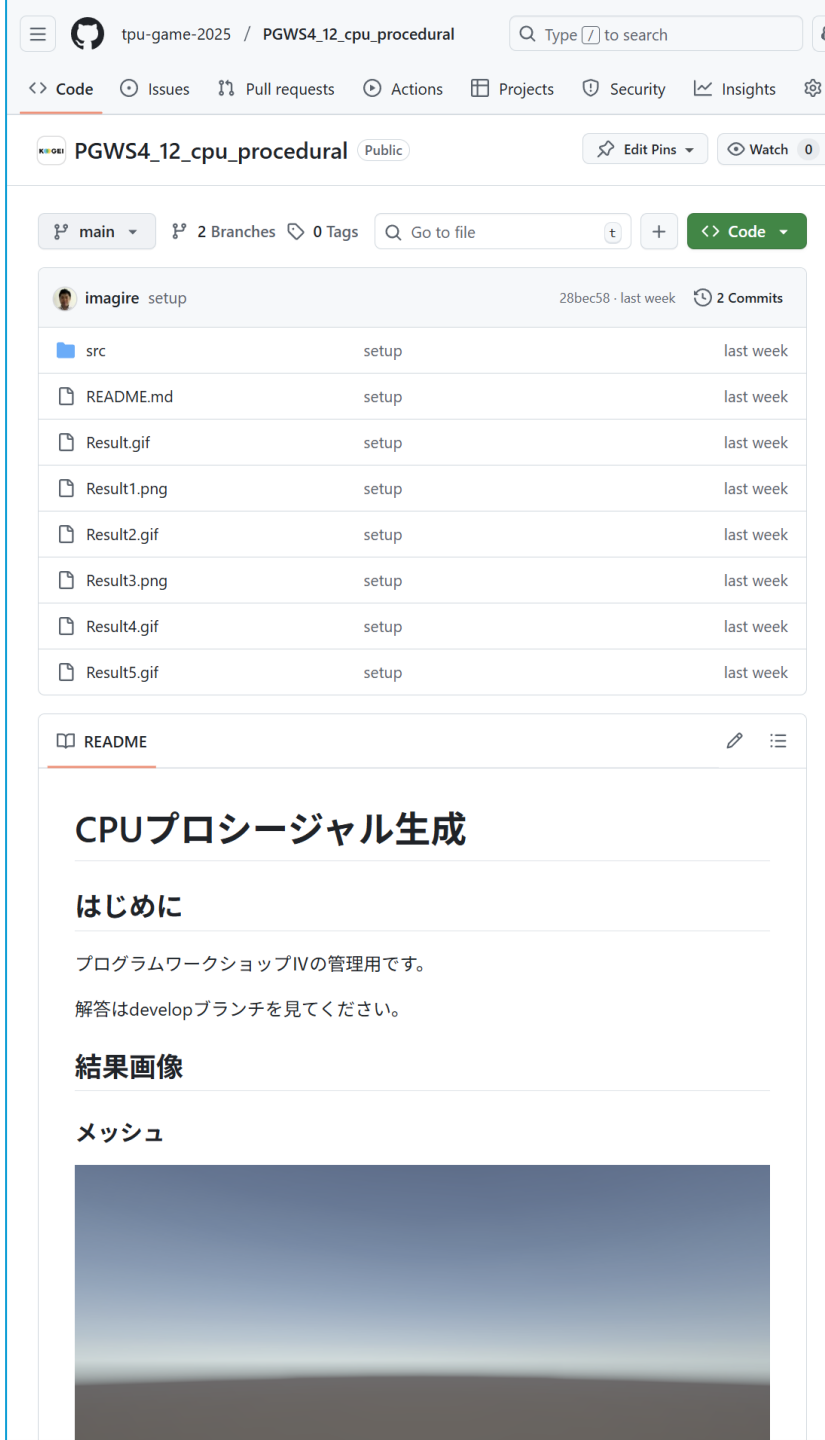
CPUプロシージャル生成

2025年度 プログラムワークショップIV (12)

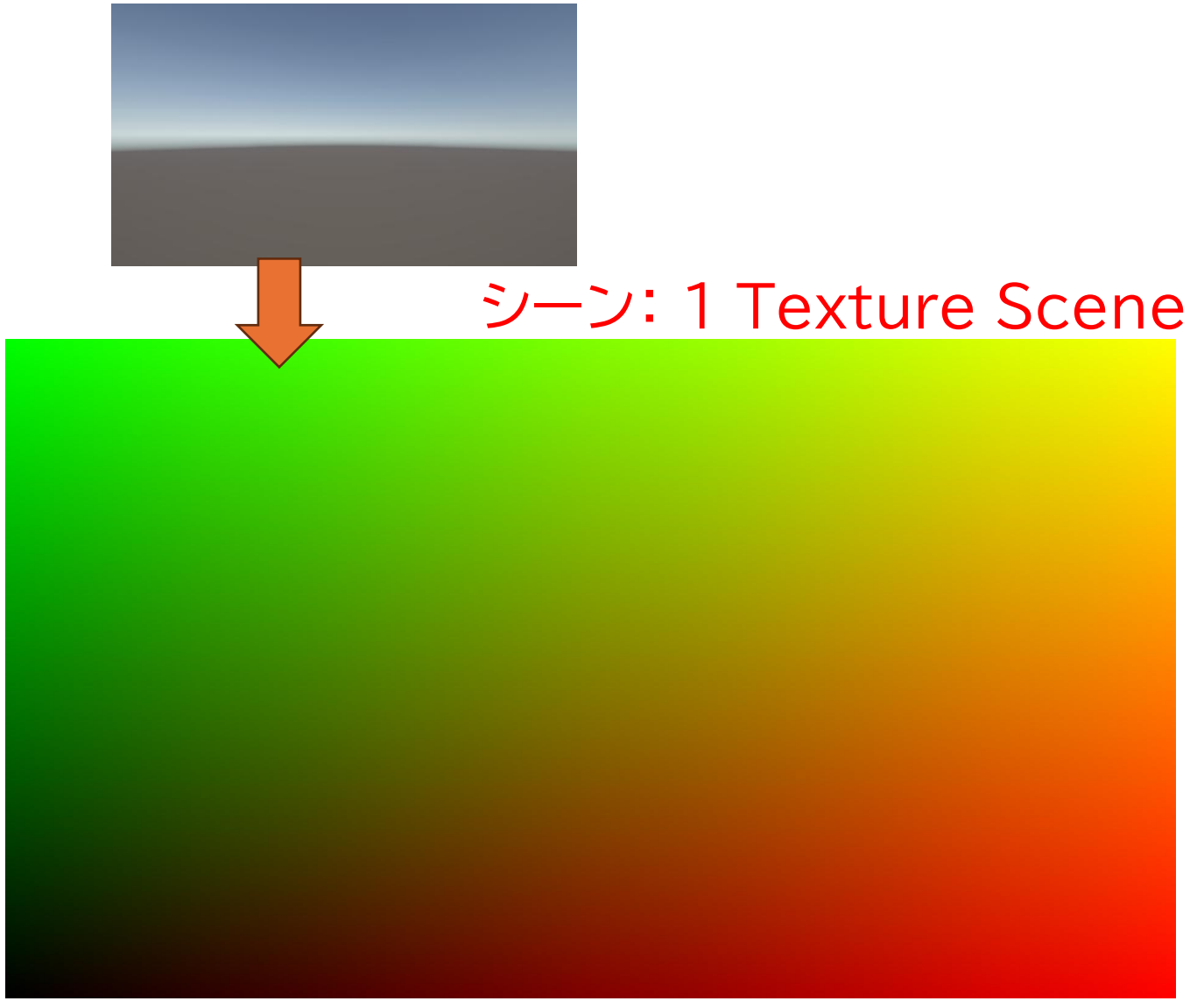
今回のリポジトリ

- https://github.com/tpu-game-2025/PGWS4_12_cpu_procedural



本日の内容

- CPUでのリソース生成
  - テクスチャへの描画
  - リヒテンベルク図形
  - ポリゴンの描画
  - 雷



CPUからのテクスチャ描画

- CPUとGPUのやり取りは原則としてCPUからGPUに転送
  - GPUの結果を受け取るのは計算終了の把握が難しいため難しい
    - GPU内で結果を保存しておいて後のフレームでCPUで読み込む
      - 古い情報しか取れない
    - GPUの描画後にイベントを発生させてコールバック処理
      - いつ結果を受け取れるかわからない

GetPixelData

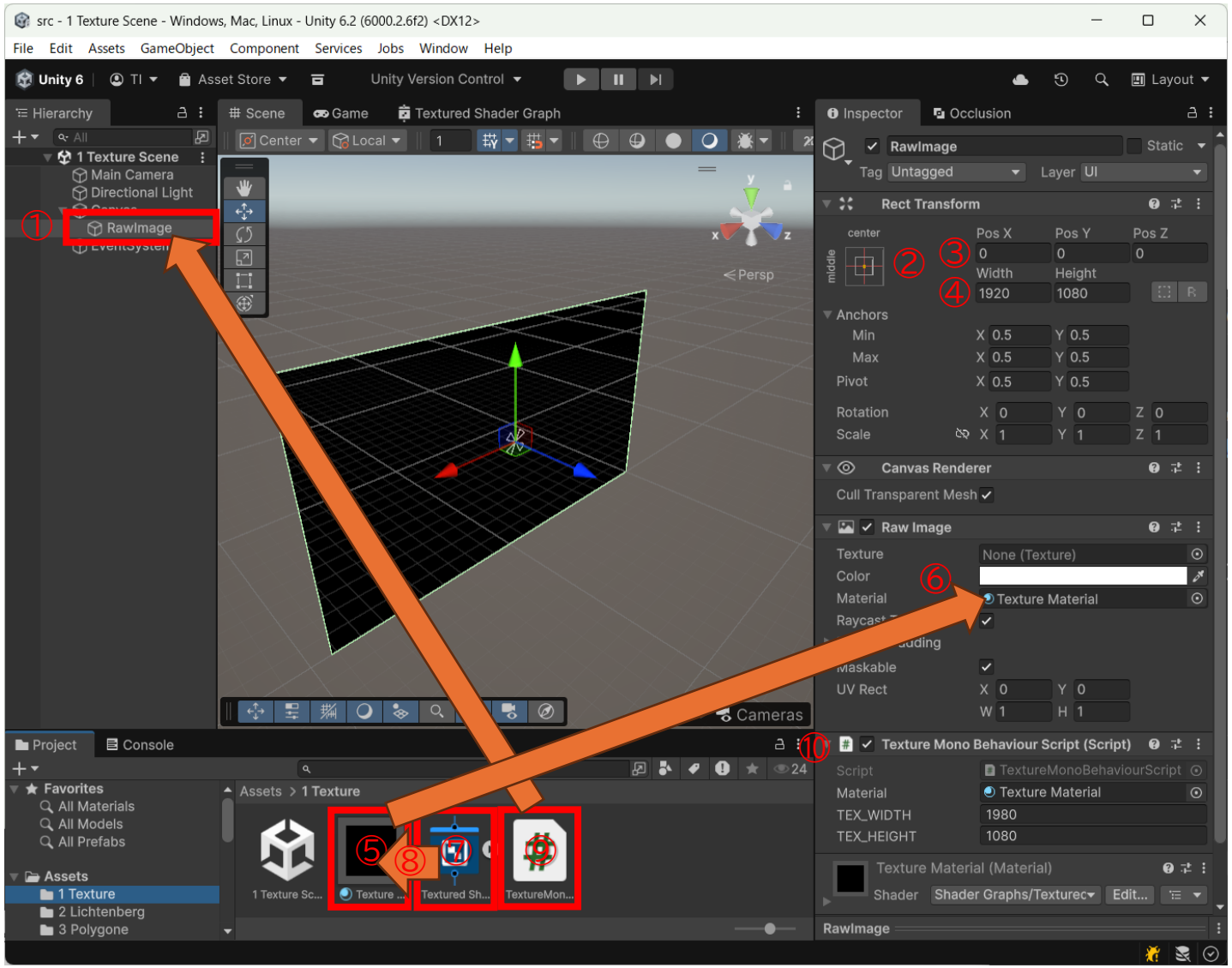
- テクスチャにデータを転送するためのバッファを確保
  - Applyで転送
  - メインメモリに永続的にバッファが確保されるのではないので、値を埋めてすぐに転送する

1 Texture/TextureMonoBehaviourScript

```
using UnityEngine;

Unity スクリプト (1 件のアセット参照)|0 個の参照
public class TextureMonoBehaviourScript : MonoBehaviour
{
    [SerializeField] Material material = default!;
    Texture2D texture = null;
    [SerializeField] int TEX_WIDTH = 1980;
    [SerializeField] int TEX_HEIGHT = 1080;

    // Start is called once before the first execution of Update after the MonoBehavio
    Unity メッセージ|0 個の参照
    void Start()
    {
        texture = new Texture2D(TEX_WIDTH, TEX_HEIGHT, TextureFormat.RGBA32, false);
        material.SetTexture("_Texture2D", texture);

        UpdateTexture();
    }

    1 個の参照
    void UpdateTexture()
    {
        var pixelData = texture.GetPixelData<Color32>(0);

        for (int y = 0; y < TEX_HEIGHT; y++)
        {
            // 縦方向は緑のグラデーション: [0,256)
            byte g = (byte)((256.0 / (double)(TEX_HEIGHT    )) * (double)y);

            for (int x = 0; x < TEX_WIDTH; x++)
            {
                // 横方向は赤のグラデーション: [0,256)
                byte r = (byte)((256.0 / (double)(TEX_WIDTH    )) * (double)x);

                pixelData[y * TEX_WIDTH + x] = new Color32(r, g, 0, 255);// 青は0固定
            }
        }

        texture.Apply();
    }
}
```

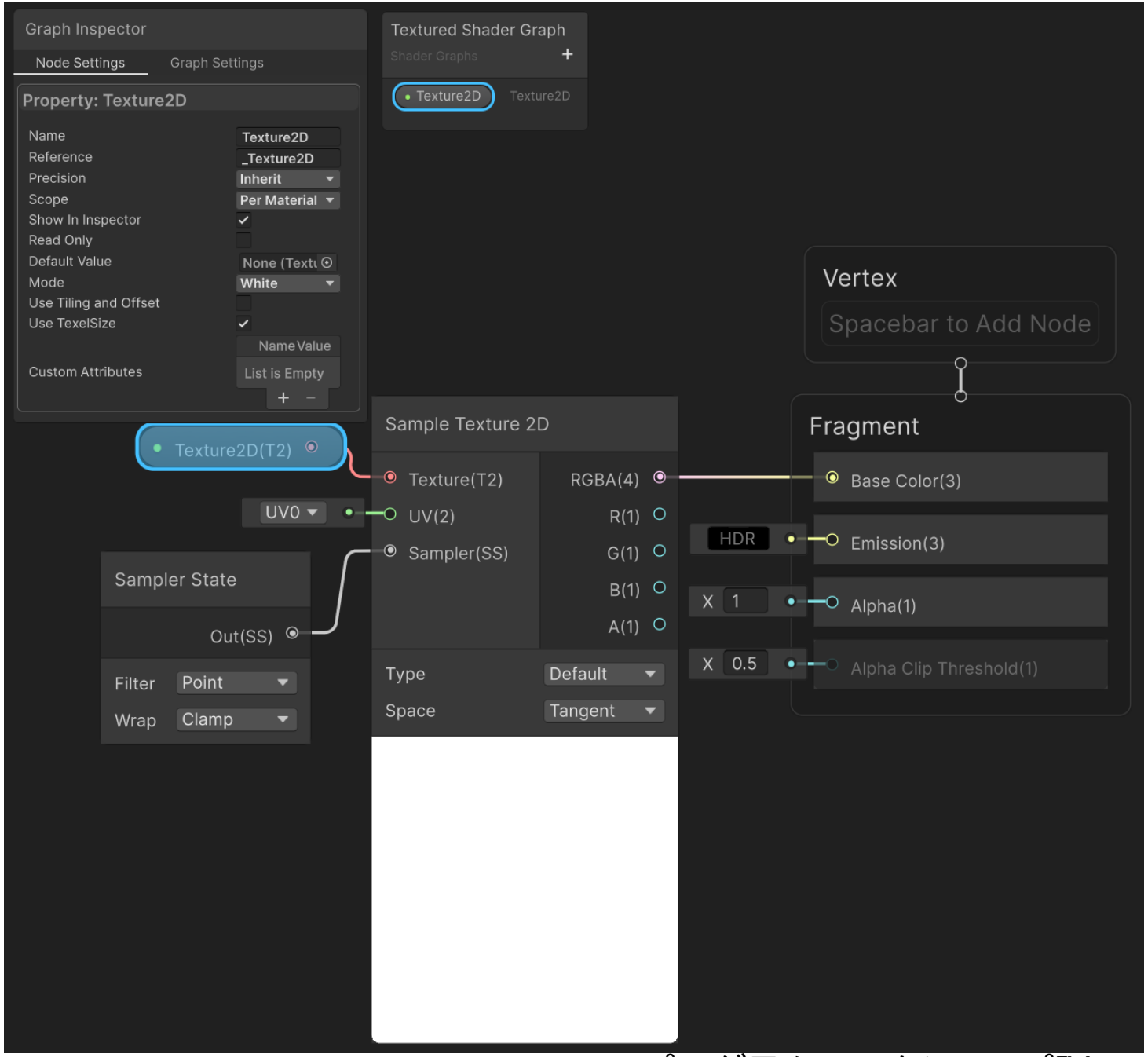
やってみよう

- 「UI/Raw Image」オブジェクトの生成
  - ①名称例：RawImage
  - ②アンカー：中心
  - ③位置：原点
  - ④幅、高さ：1920x1080（画面に合わせる）
- マテリアルの作成
  - ⑤名称例：Texture Material
  - ⑥RawImageオブジェクトに設定
    - 「Raw Image」 Material
- シェーダグラフの作成
  - Canvas Shader Graph
  - ⑦名称例：Textured Shader Graph
  - ⑧Texture Materialに設定
- MonoBehaviour Scriptの生成
  - ⑨名称例：TextureMonoBehaviourScript
  - ⑩RawImageオブジェクトにバインド



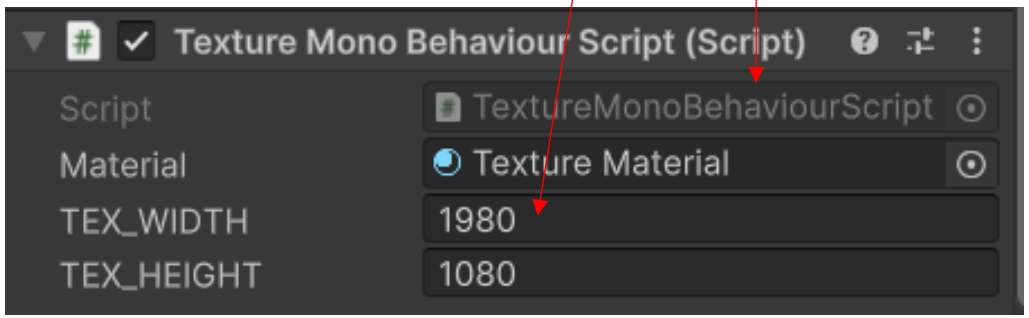
シェーダグラフ

- Texture2Dプロパティを追加
  - Reference「_Texture2D」を通してCPUから認識する
- シェーダグラフはテクスチャを表示
  - サンプラー
    - はっきりするようにポイントサンプリング
    - 上下、左右に回り込まないようにWrapはPoint



スクリプト

- マテリアルを通してシェーダのテクスチャ設定
  - テクスチャ自体も生成
    - Texture2Dオブジェクト
- Texture Mono Behavior Scriptを設定
  - マテリアル:Texture Material
  - サイズ(幅と高さ)



1 Texture/TextureMonoBehaviourScript

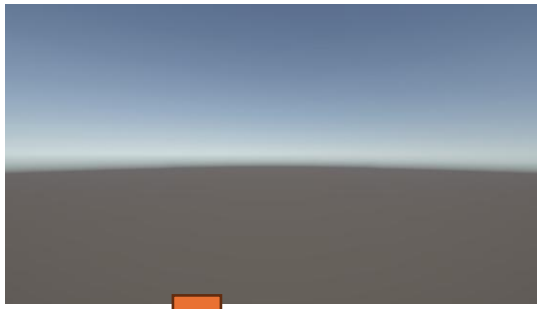
```
1   using UnityEngine;
2
    Unity スクリプト (1 件のアセット参照)|0 個の参照
3   public class TextureMonoBehaviourScript : MonoBehaviour
4   {
5       [SerializeField] Material material = default!;
6       Texture2D texture = null;
7       [SerializeField] int TEX_WIDTH = 1980;
8       [SerializeField] int TEX_HEIGHT = 1080;
9
10      // Start is called once before the first execution of Update after the MonoBehavio
        Unity メッセージ|0 個の参照
11      void Start()
12      {
13          texture = new Texture2D(TEX_WIDTH, TEX_HEIGHT, TextureFormat.RGBA32, false);
14          material.SetTexture("_Texture2D", texture);
15
16          UpdateTexture();
17      }
18
        1 個の参照
19      void UpdateTexture()
20      {
21          var pixelData = texture.GetPixelData<Color32>(0);
22
23          for (int y = 0; y < TEX_HEIGHT; y++)
24          {
25              // 縦方向は緑のグラデーション: [0,256)
26              byte g = (byte)((256.0 / (double)(TEX_HEIGHT    )) * (double)y);
27
28              for (int x = 0; x < TEX_WIDTH; x++)
29              {
30                  // 横方向は赤のグラデーション: [0,256)
31                  byte r = (byte)((256.0 / (double)(TEX_WIDTH    )) * (double)x);
32
33                  pixelData[y * TEX_WIDTH + x] = new Color32(r, g, 0, 255);// 青は0固定
34              }
35          }
36
37          texture.Apply();
38      }
39  }
```

完成

テクスチャ座標：左下が原点
左右：左から右
上下：下から上

本日の内容

- CPUからのリソース生成
  - テクスチャの描画
  - リヒテンベルク図形
    - テクスチャ書き込みの応用
  - ポリゴンの描画
  - 雷



シーン：2 Lichtenberg Scene

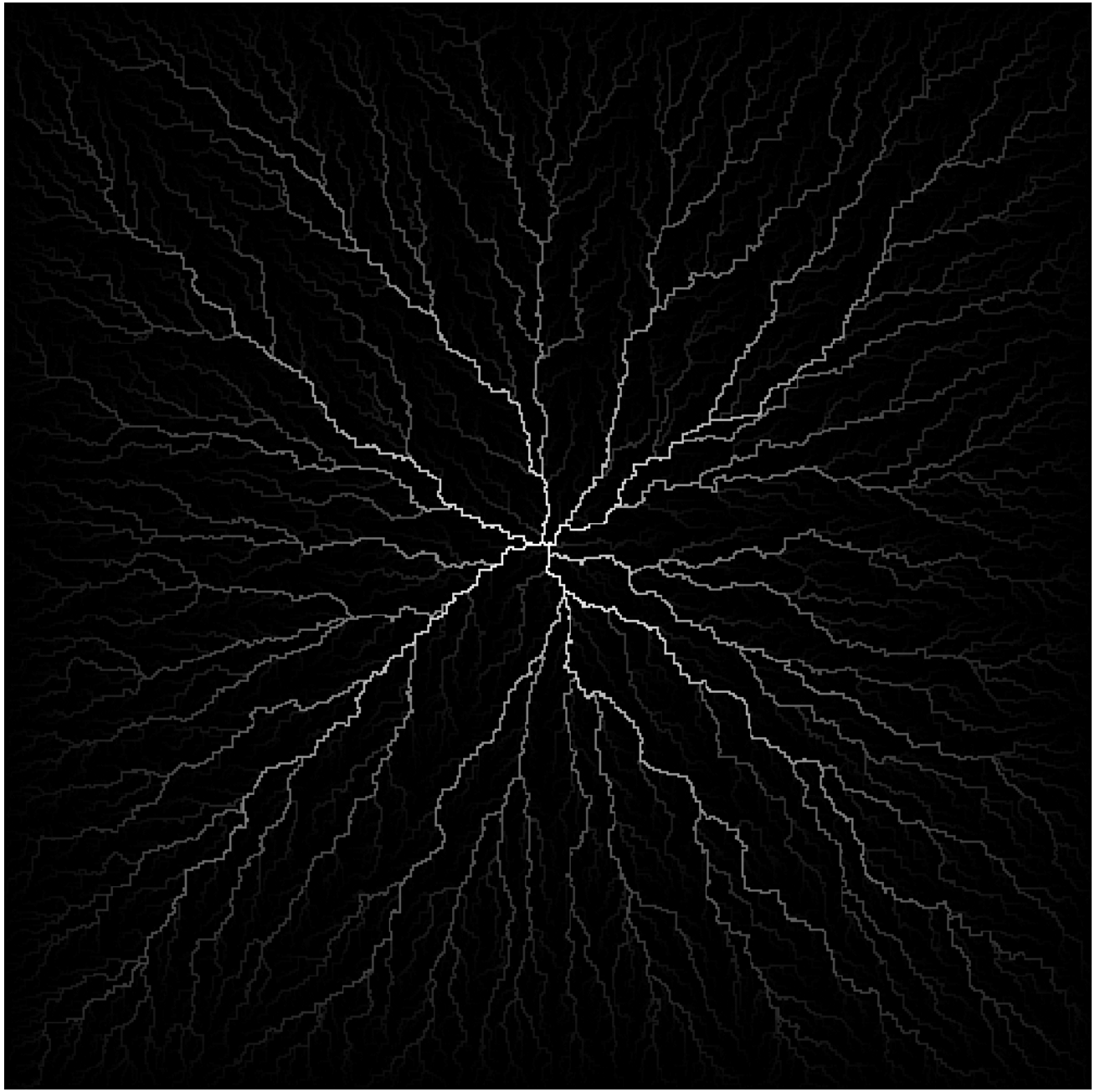
雷

- 高さは1，2km
- 30-50mほどのステップで曲がりながら進む

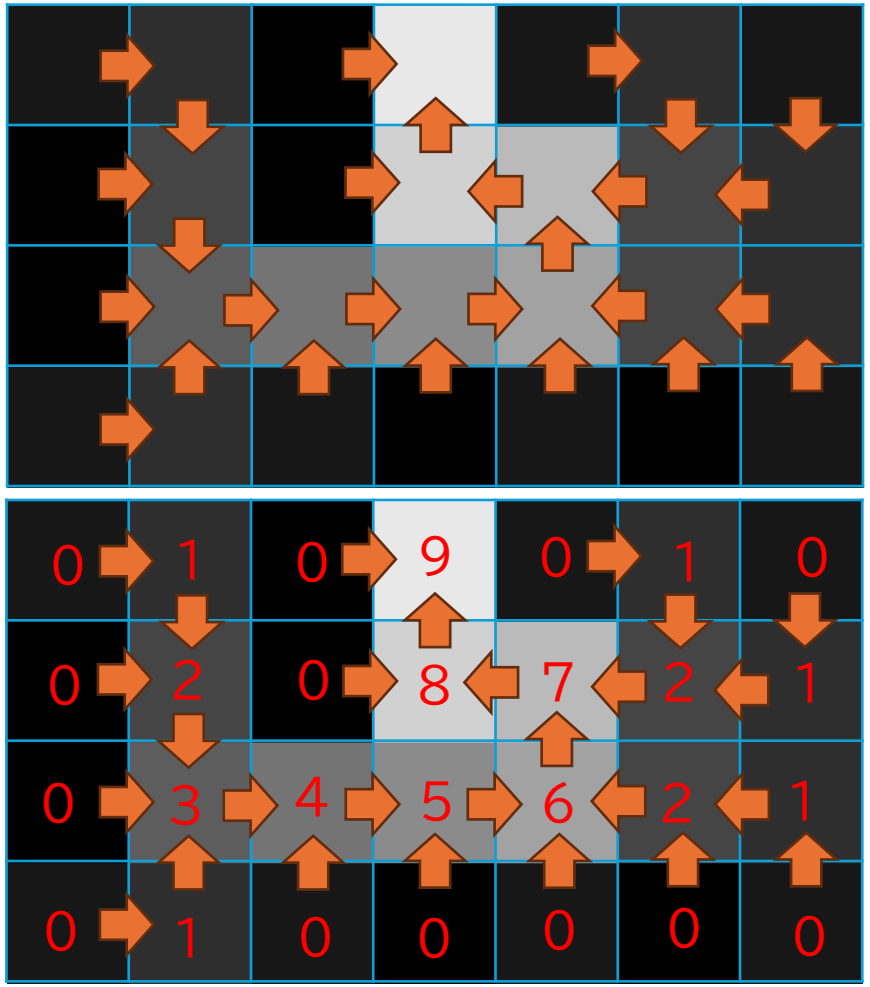
https://www.publicdomainpictures.net/jp/view-image.php?image=215233

雷の表現

- 確率的に伸びていく
  - リヒテンベルク図形
    - 絶縁材料の表面または内部にときに現れる分岐放電の図形



リヒテンベルク図形の作り方



- 始点から確率的に伸ばしていく
- 全てが埋まったら末端から親をたどる
  - 末端からの距離の最大値を記録する
- 末端からの距離の大きさを明るさとする

- 雷の場合はいずれかが地面に着いた時点で光るのが良いかも
  - 地面に着いたことを検索の終了条件として親をたどればよい

やってみよう

- リヒテンベルク図形クラスの作成
- リヒテンベルク図形クラスを呼び出すMono Behaviourの作成
- 表示用のUIの組み込み

やってみよう

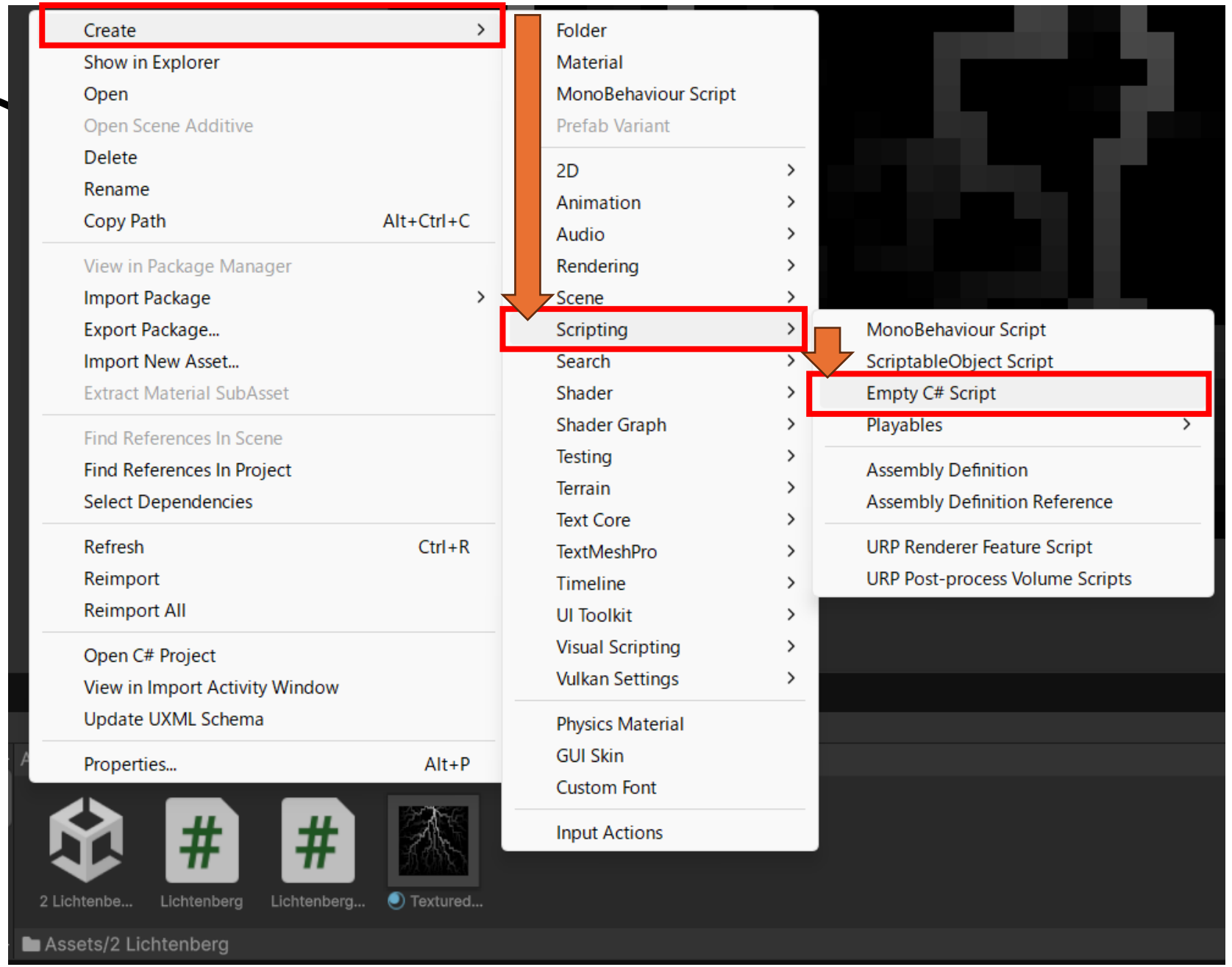
- リヒテンベルク図形クラスの作成
- リヒテンベルク図形クラスを呼び出すMono Behaviourの作成
- 表示用のUIの組み込み

C#スクリプトの追加

- リヒテンベルク図形はUnityと無関係
  - MonoBehaviourではないクラスとして作る
  - 一般的に、なるべく依存性をなくすのが好ましい
    - 移植性
    - 他の修正による不具合が起きにくい

Lichtenberg.cs

```
1   using System.Collections.Generic;
2
        8 個の参照
3   public class Lichtenberg...
```



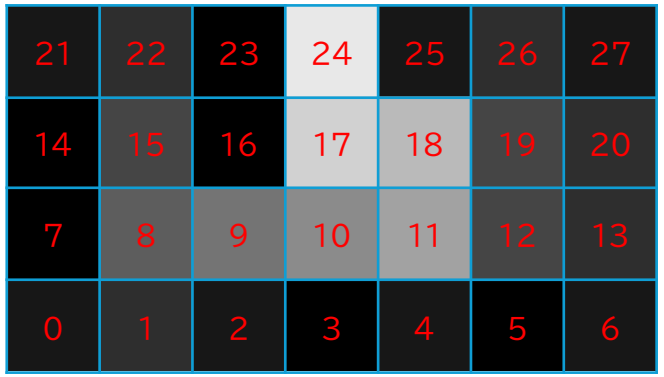
基本情報

- モード
  - 全て探索
  - 下についたら終了
- 状態
  - 構築中
  - 完了
    - 全て検索
    - 下についたら終了
- サイズ
  - 幅
  - 高さ
- 現在計算しているモード

```
5       public enum MODE
6       {
7           ALL,                        // 全ての範囲を埋める
8           FINISH_AT_FIRST_ARRIVE,     // 最初に端に到達したら終了
9       }
10
        6 個の参照
11      public enum State
12      {
13          Running,                    // 構築中
14          FinishedAll,                // 全て埋まった
15          FinishedAtFirstArrive,      // 最初に端に到達して終了
16      }
17
        9 個の参照
18      public int Width { get; private set; } = 96;
        3 個の参照
19      public int Height { get; private set; } = 54;
20
        2 個の参照
21      public MODE Mode { get; private set; } = MODE.ALL;
```

メンバー変数

- parent: 親のインデックス
  - 位置を線型化: index = y * Width + x
- edge: 次にいける場所を記録
  - Parent(親)→Child(子)
  - List: 可変長配列
    - ランダムな選択が必要なので、可変個数かつ、途中の要素も高速にアクセスできるコンテナ
- Value
  - 末端からの距離の最大値
  - キャッシュに載りやすいようにushortと小さく
- StartIndex
  - 始点を記録
- ArriveIndex
  - 端に着いた最初の点を記録



インデックス

Lichtenberg.cs

```
23          // 出力値
24          int StartIndex = -1;// 開始インデックス
            10 個の参照
25          public ushort[] Value { get; private set; } = null;// 各セルの値
            2 個の参照
26          public ushort ValueMax { get { return StartIndex < 0 ? (ushort)0 : Value[StartIndex]; } }
            3 個の参照
27          public int ArriveIndex { get; private set; } = -1;// 最初に端に到達したインデックス
28
29          // 内部管理用の親インデックス
            12 個の参照
30          public int[] Parent { get; private set; } = null;
31
32          // 拡張候補のエッジ
            6 個の参照
33          struct Edge
34          {
35              public int Parent;
36              public int Child;
37          }
38          List<Edge> edge = new List<Edge>();
```

コンストラクタ

- サイズを指定
  - メモリを確保
- 始点を後からも変更できるようにリセット関数を用意
- インデックスは-1を不定値とした

Lichtenberg.cs

```
40      public Lichtenberg(int height, int width, MODE mode = MODE.ALL)
41      {
42          Height = height;
43          Width = width;
44          Mode = mode;
45
46          int N = width * height;
47          Parent = new int[N];
48          Value = new ushort[N];
49
50          ResetInternalState(); // 初期クリアもコンストラクタで1回行っておく
51      }
52
        2 個の参照
53      void ResetInternalState()
54      {
55          // Parent, Value をクリア
56          for (int i = 0; i < Width * Height; i++)
57          {
58              Parent[i] = -1;
59              Value[i] = 0;
60          }
61
62          edge.Clear();
63          ArriveIndex = -1;
64          StartIndex = -1;
65      }
```

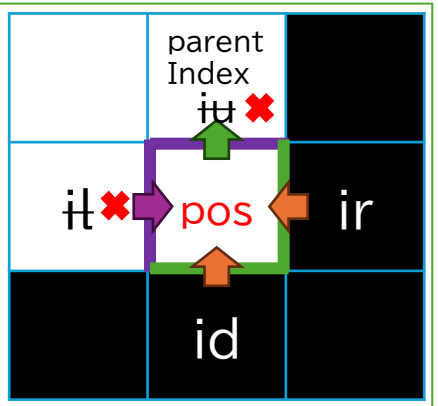
初期化

Lichtenberg.cs

- 念のため、変数をリセット
- 始点は親が自分自身であることで判定する
  - 変数にも保存しているが…
- 続くセルをedgeに登録
  - 検索済み(親が-1でない)なら登録しない

```
87      public void Initialize(int y, int x)
88      {
89          ResetInternalState();
90
91          int idx = y * Width + x;
92          StartIndex = idx;
93          AddEdge(idx, idx);// 親が自分自身なのを終了条件とする
94      }
```

```
67      int AddEdge(int pos, int parentIndex)
68      {
69          Parent[pos] = parentIndex;
70
71          int num = 0;
72          int y = pos / Width;
73          int x = pos % Width;
74
75          int iu = pos - Width;
76          int id = pos + Width;
77          int il = pos - 1;
78          int ir = pos + 1;
79          if (    0 <      y && Parent[iu] == -1) { num++; edge.Add(new Edge { Parent = pos, Child = iu }); }
80          if (y + 1 < Height && Parent[id] == -1) { num++; edge.Add(new Edge { Parent = pos, Child = id }); }
81          if (    0 <      x && Parent[il] == -1) { num++; edge.Add(new Edge { Parent = pos, Child = il }); }
82          if (x + 1 <  Width && Parent[ir] == -1) { num++; edge.Add(new Edge { Parent = pos, Child = ir }); }
83
84          return num;
85      }
```



いける場所を選択

- Listに登録がなければ終了
  - A*と同じ考え方
- 無作為に一つ選択
  - シェーダでは実施しずらいのでCPU処理となる
- 選んだら削除
- 選んだ場所が別の方向から先に選ばれている可能性がある。先に選ばれていないか確認
  - 親が登録されているかどうか
  - すでに選ばれていたら再トライ
- 選択したらさらに先を追加

```
113     public State Update()
114     {
115         // 行先を選ぶ
116         int q;// edgeのインデックス
117         int pos;// 行先インデックス
118         int pa;// 親インデックス
119
120         do
121         {
122             // 行ける場所がないなら終了
123             int c = edge.Count;
124             if (c == 0)
125             {
126                 return State.FinishedAll;
127             }
128
129             // 行先をランダムに選ぶ
130             q = UnityEngine.Random.Range(0, c);
131             pos = edge[q].Child;
132             pa = edge[q].Parent;
133
134             // 選ばれたエッジは削除
135             edge[q] = edge[c - 1];   C#Cのリストは末尾の削除が高速
136             edge.RemoveAt(c - 1);    (末尾のデータを移した後、末尾を削除)
137
138         }while (Parent[pos] != -1); // 既に埋まっているなら繰り返し
139
140         int num = AddEdge(pos, pa);// 新たなエッジを追加
```

到達判定

- いける場所がなければ末端
  - 末端からの距離を計算
- 通常は「継続(Running)」で1ループを終了

```
96          // 親をさかのぼって最大距離を埋める
            1 個の参照
97          private void searchRoot(int idx)
98          {
99              ushort v = 1;
100             while (Parent[idx] != idx)
101             {
102                 // 既に大きな値が設定されていれば譲る
103                 if (v <= Value[idx]) return;
104
105                 Value[idx] = v++;
106                 idx = Parent[idx];
107             }
108
109             // ルートの処理
110             Value[idx] = v < Value[idx] ? Value[idx] : v;
111         }
```

Updateの続き

```
154                 //既に周囲は埋まっていた
155                 if (num == 0)
156                 {
157                     searchRoot(pos);// 値の更新
158                 }
159
160                 return State.Running;
161             }
```

最初に端に到達したら終了

- 終了した際に高さ(pos/Width)を見て、一番下(0)にたどり着いたら終了とする
  - 後で追えるように位置を保存
  - 終了と同様に親への経路の最大値の更新を挑戦してみる
  - 探索用のエッジをクリアしてこれ以上検索できないようにする
  - 返り値をRunningではない値にする

Lichtenberg.cs

```
27      public int ArriveIndex { get; private set; } = -1;// 最初に端に到達したインデックス
140             int num = AddEdge(pos, pa);// 新たなエッジを追加
141
142             // 最初に端に到達したら終了
143             if (Mode == MODE.FINISH_AT_FIRST_ARRIVE)
144             {
145                 if (pos / Width == 0)
146                 {
147                     ArriveIndex = pos;
148                     searchRoot(pos);// 値の更新
149                     edge.Clear(); // 空にして強制終了
150                     return State.FinishedAtFirstArrive;
151                 }
152             }
```

やってみよう

- リヒテンベルク図形クラスの作成
- リヒテンベルク図形クラスを呼び出すMono Behaviourの作成
- 表示用のUIの組み込み

呼び出し

- 更新用のMonoBehaviourを追加
  - 名称例：LichtenbergMonoBehaviour

図形の作成

- コルーチンで少しづつ更新
  - yield return nullで各フレームセルを1つずつ処理
- 始点は、上部中央

LichtenbergMonoBehaviourScript.cs

```
using System.Collections;
using UnityEngine;

Unity スクリプト (1 件のアセット参照)|0 個の参照
public class LichtenbergMonoBehaviourScript : MonoBehaviour
{
    [SerializeField] Material material = default!;
    Texture2D texture = null;
    [SerializeField] int TEX_WIDTH = 128;
    [SerializeField] int TEX_HEIGHT = 64;

    Lichtenberg lichtenberg = null;

    // Start is called once before the first execution of Update after the MonoBehavi
    Unity メッセージ|0 個の参照
    void Start()
    {
        texture = new Texture2D(TEX_WIDTH, TEX_HEIGHT, TextureFormat.RGBA32, false);
        material.SetTexture("_Texture2D", texture);

        lichtenberg = new Lichtenberg(TEX_HEIGHT, TEX_WIDTH);

        // 検索の初期化
        int x = TEX_WIDTH / 2;
        int y = TEX_HEIGHT - 1;// yが大きい方が上
        lichtenberg.Initialize(y, x);

        StartCoroutine(Simulate());
    }

    1 個の参照
    private IEnumerator Simulate()
    {
        while(lichtenberg.Update() == Lichtenberg.State.Running)
        {
            UpdateTexture();
            yield return null;
        }
    }
```

テクスチャ更新

- リヒテンベルク図形クラスの距離のデータを読み込んで正規化

LichtenbergMonoBehaviourScript.cs

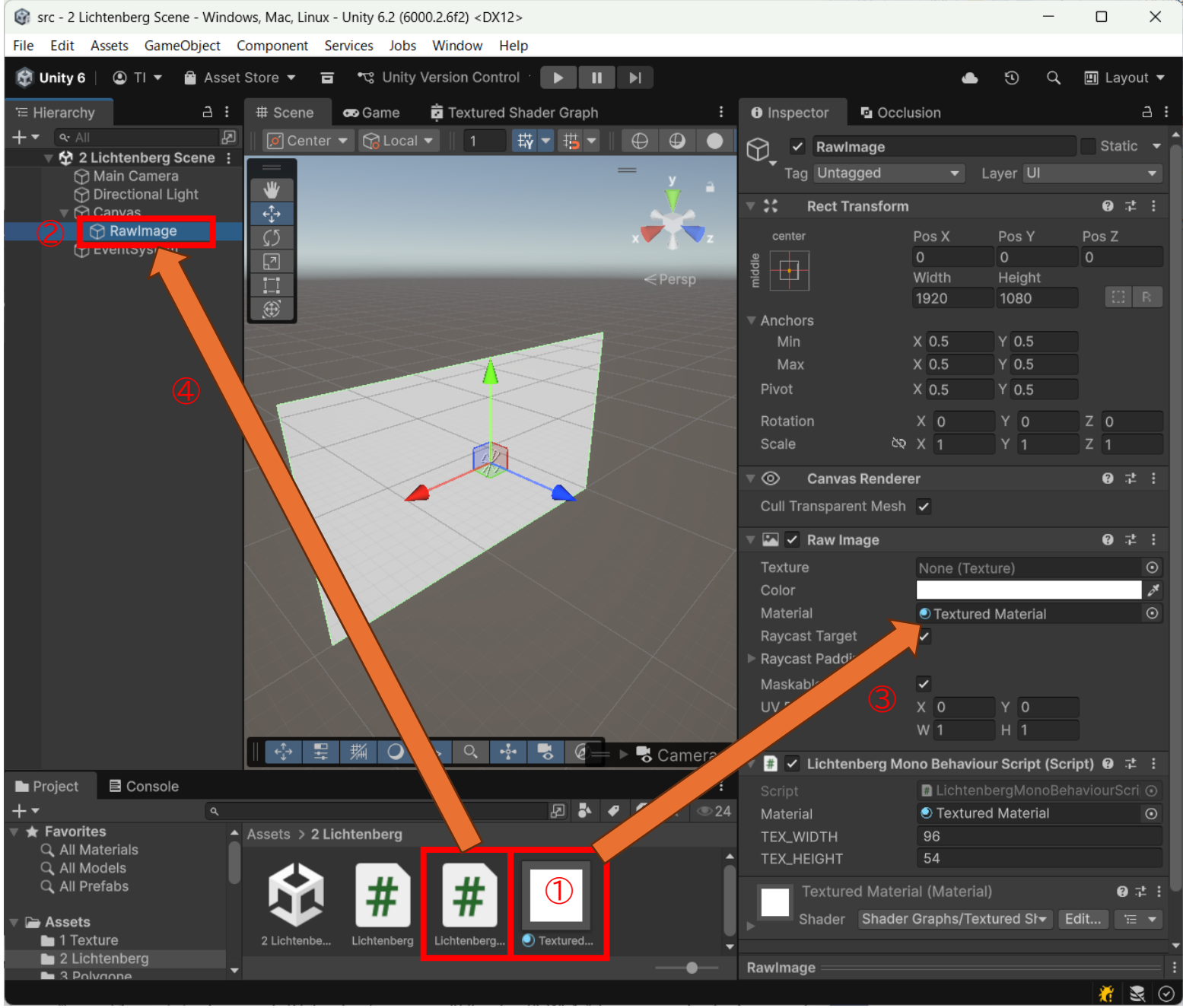
```
38      void UpdateTexture()
39      {
40          var pixelData = texture.GetPixelData<Color32>(0);
41
42          ushort[] value = lichtenberg.Value;
43          double vMax = (double)lichtenberg.ValueMax;
44
45          int idx = 0;
46          for (int y = 0; y < TEX_HEIGHT; y++)
47          {
48              for (int x = 0; x < TEX_WIDTH; x++)
49              {
50                  byte c = (byte)(256.0 * (double)value[idx] / (vMax+1));// 256は超えないようにする
51                  pixelData[idx] = new Color32(c, c, c, 255);
52                  idx++;
53              }
54          }
55
56          texture.Apply();
57      }
```

やってみよう

- リヒテンベルク図形クラスの作成
- リヒテンベルク図形クラスを呼び出すMono Behaviourの作成
- 表示用のUIの組み込み

可視化

1. マテリアルの追加
   - 名称例:Textured Material
   - すでに作成した、「1 Texture/Textured Shader Graph」を設定
2. UI/Raw Image オブジェクトの追加
   - 名称例:Raw Image
   - アンカー:中央
   - 幅・高さ:表示サイズに合わせる
   - 「Raw Image」の「Material」に「Textured Material」を追加
   - LichtenbergMonoBehaviourスクリプトの追加
     - プロパティのマテリアルに「Textured Material」を設定
     - プロパティの幅と高さを設定
       - アスペクト比をオブジェクトのアスペクト比と合わせるとセルが正方形で表示される



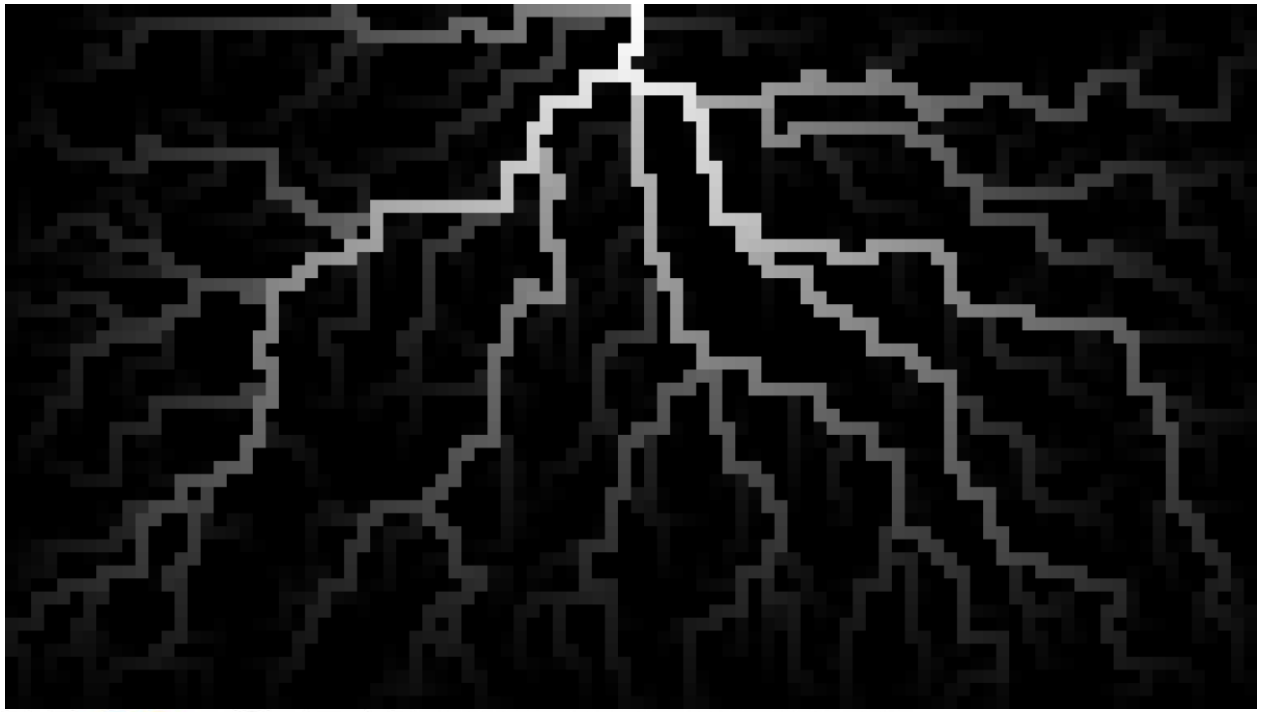
完成

- テクスチャの
  解像度を
  変更して
  みよう

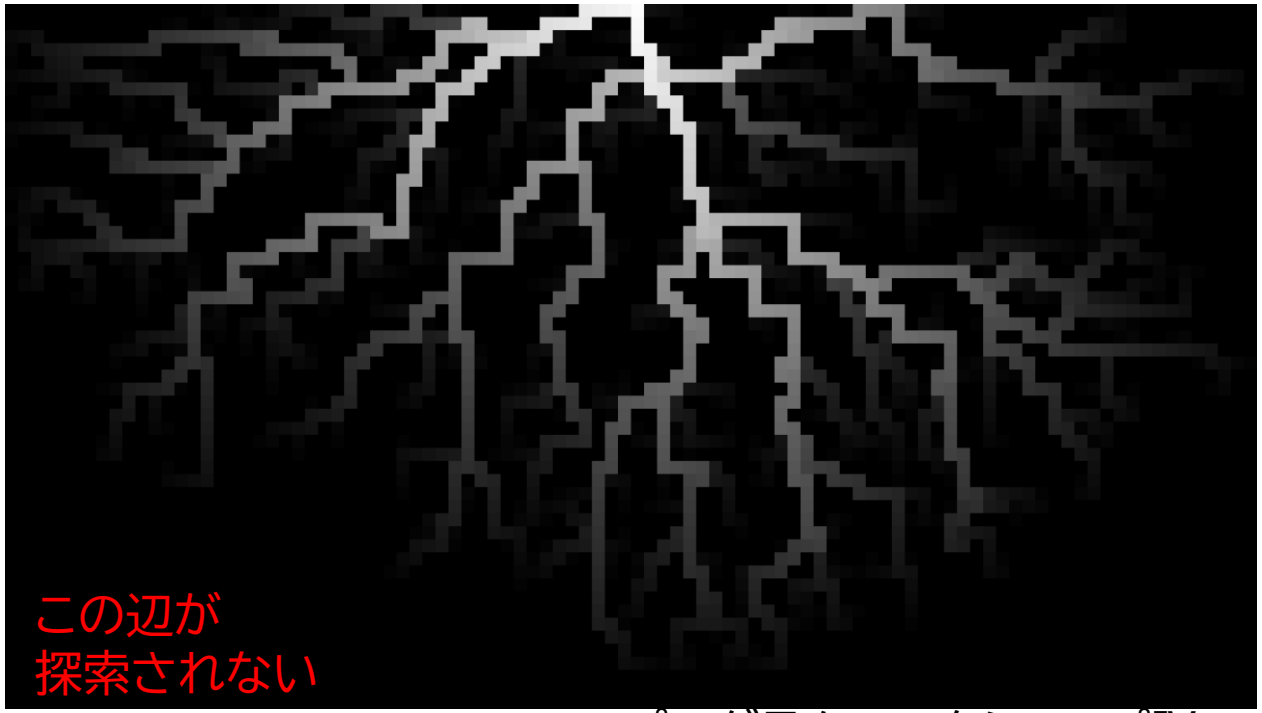
引数の違い

- おおよそ円形に広がる
- すべてをたどると斜めが伸びがち

MODE.ALL

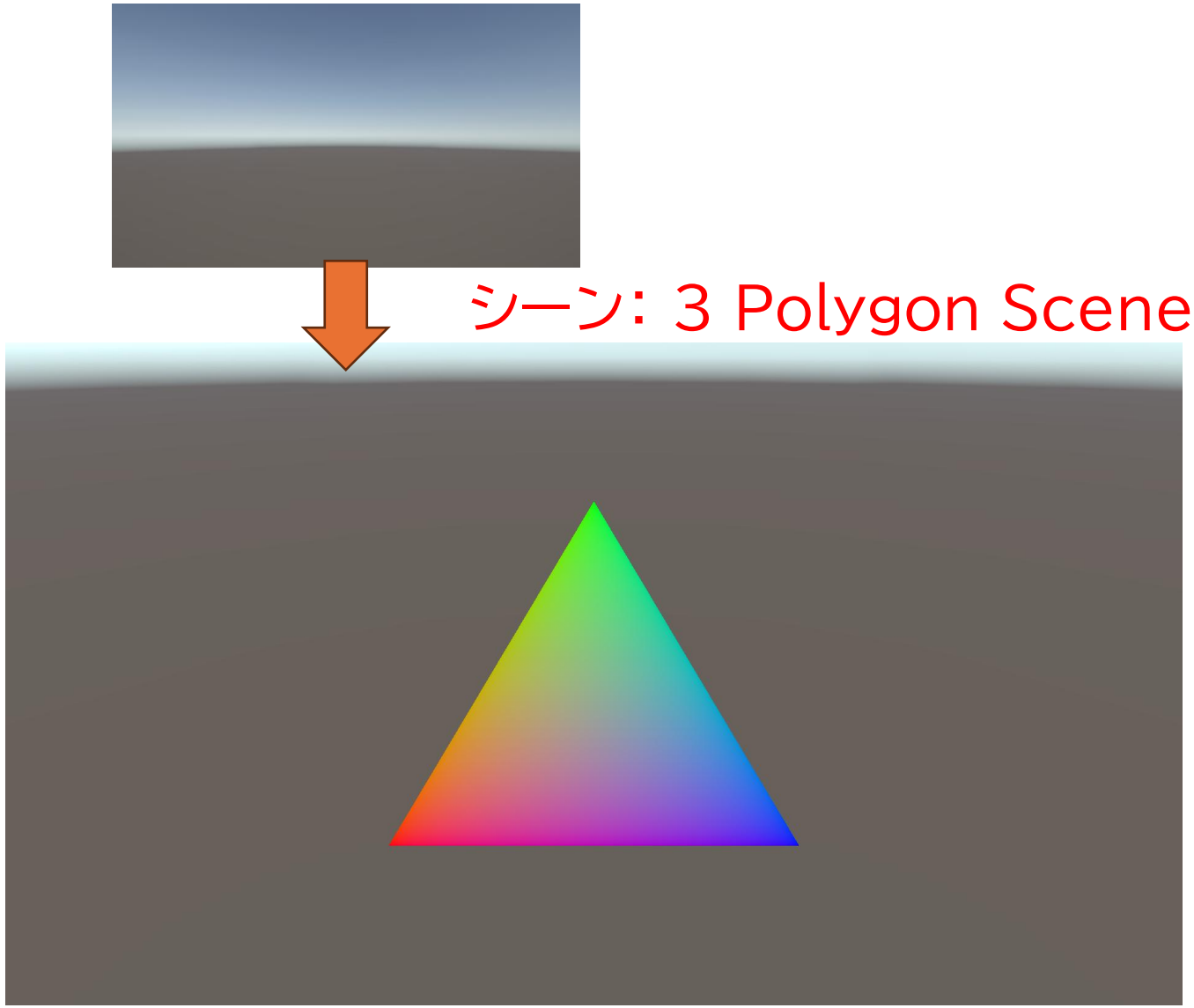


MODE.FINISH_AT_FIRST_ARRIVE



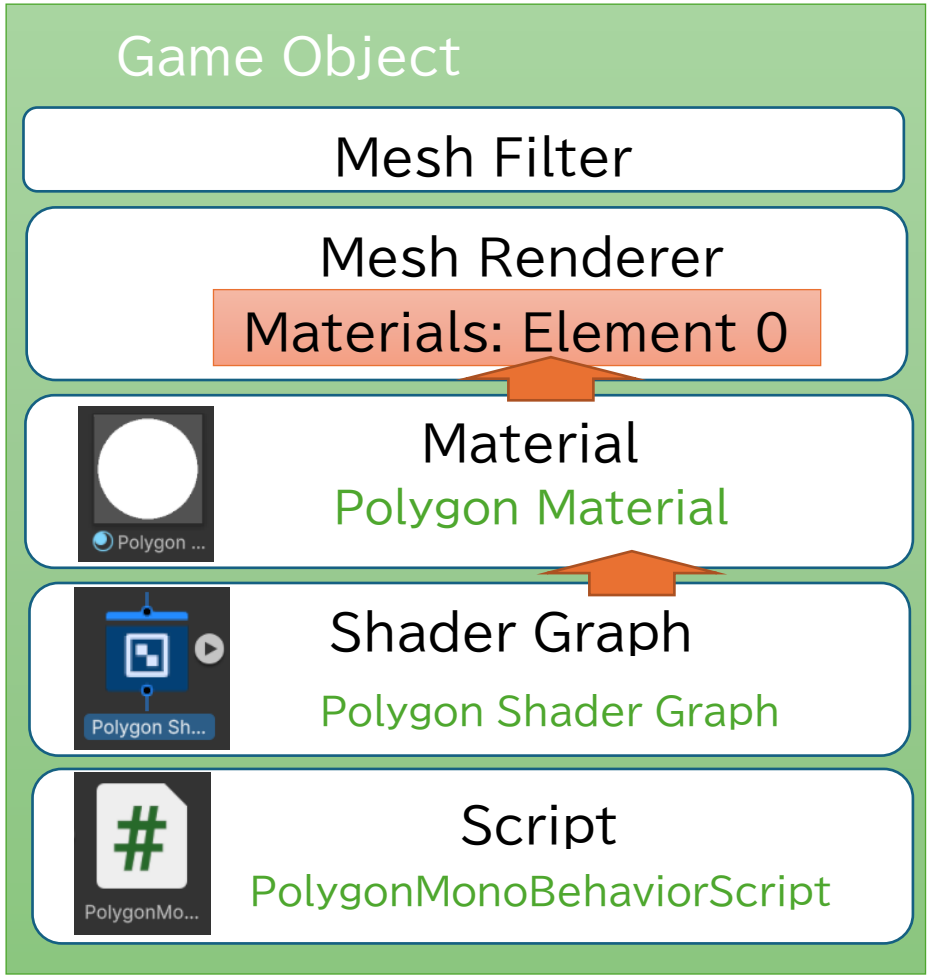
本日の内容

- CPUからのリソース生成
  - テクスチャの描画
  - リヒテンベルク図形
  - ポリゴンの描画
  - 雷



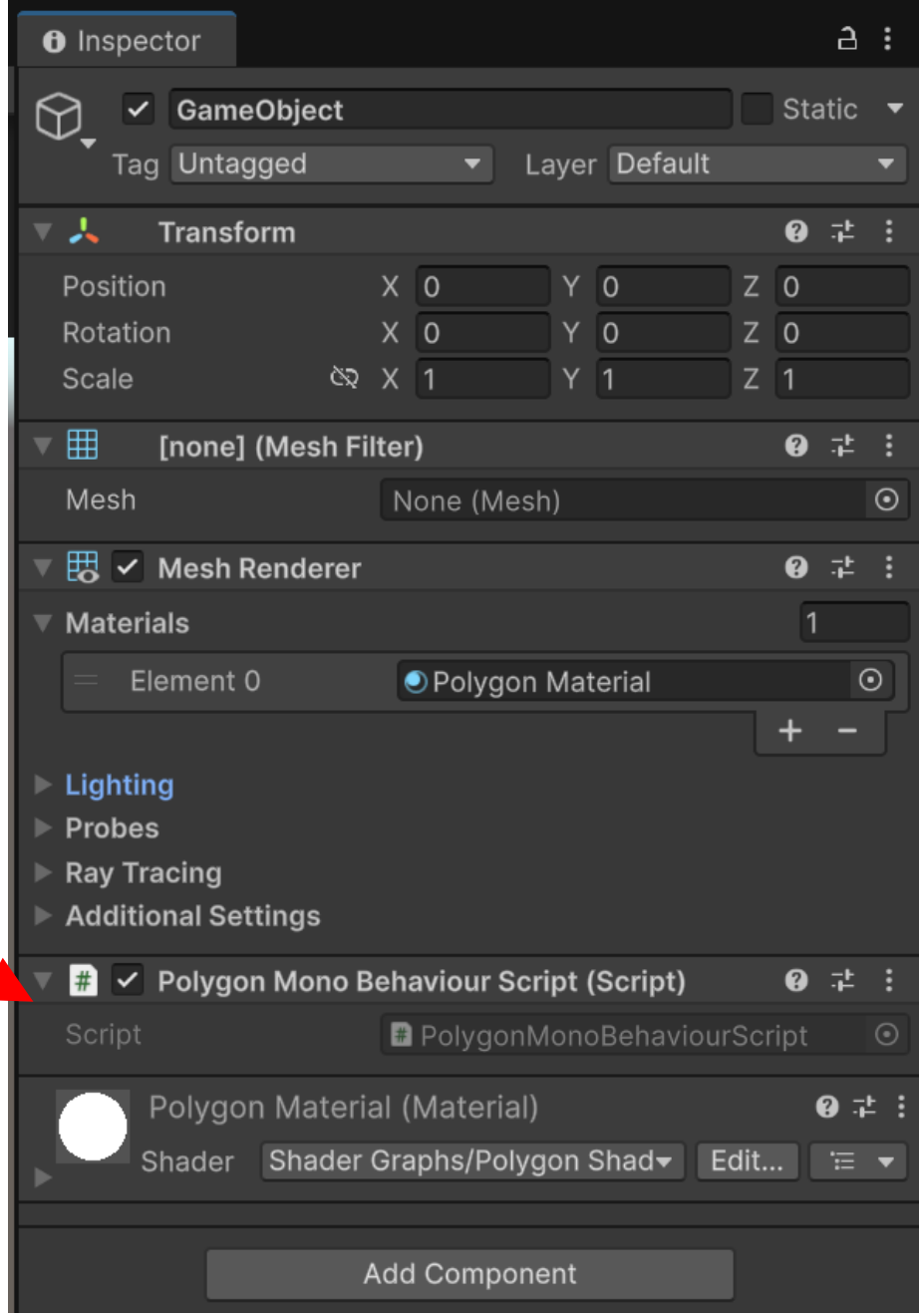
ポリゴンの描画

- コンポーネントの追加
  - メッシュフィルター
    - レンダリングされるメッシュを保持する
  - メッシュレンダラー
    - メッシュの描画設定を保持する
    - Materialsにマテリアルを設定する
  - マテリアル
    - シェーダを設定する
  - Shader Graph
    - シェーダを記載する
- スクリプトの追加
  - ポリゴンを生成



やってみよう

1. Shader Graphを追加
   - 種類:Unlit Shader Graph
   - 名称例:Polygon Shader Graph
2. マテリアルを追加
   - 名称例:Polygon Material
   - 「Polygon Shader Graph」を設定
3. Mono Behaviour Scriptを追加
   - 名称例: PolygonMonoBehaviourScript
4. 「Empty Object」を追加
   - コンポーネントを追加
     - Mesh Filter
     - Mesh Renderer
       - MaterialsのElement0に「Polygon Material」を追加
     - PolygonMonoBehaviourScript
       - ドラッグアンドドロップ



スクリプト

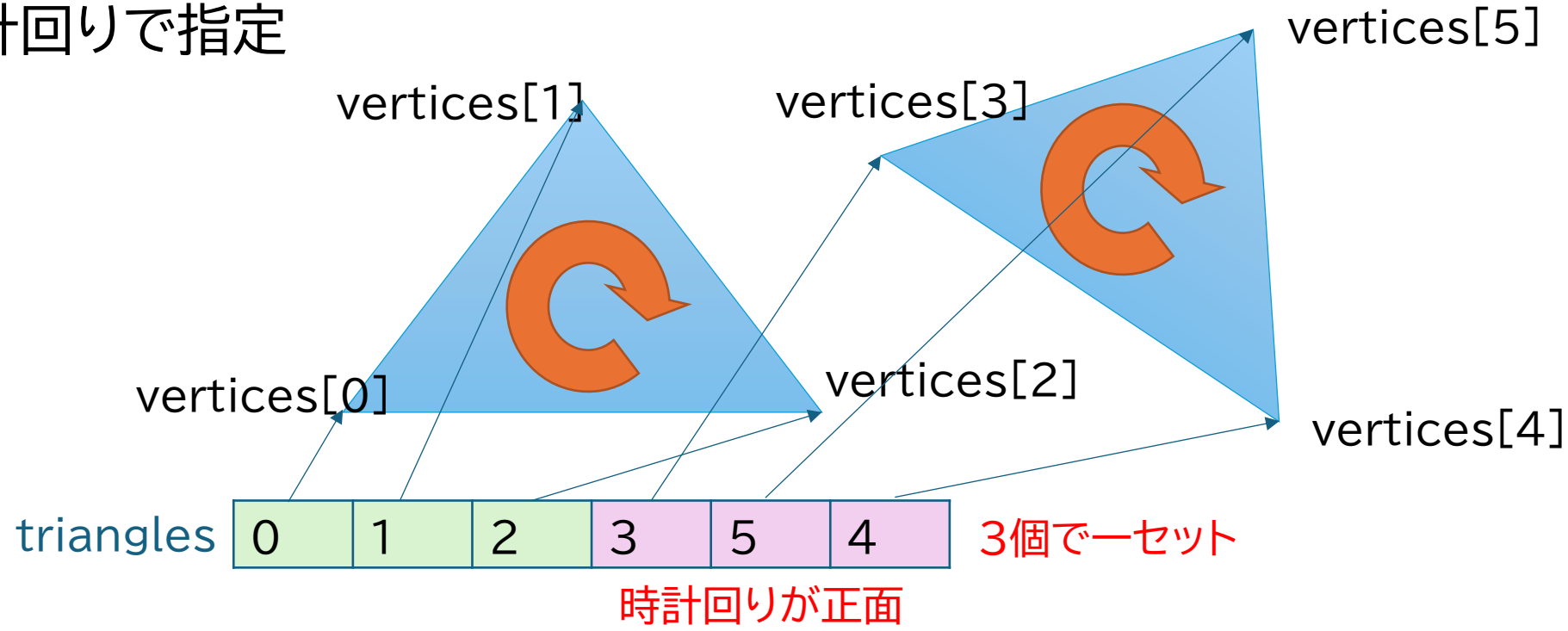
- メッシュオブジェクトを取得
- リセット
- データを設定
  - 頂点位置
  - 頂点色
  - 頂点インデックス
- 重要なデータの計算
  - バウンディングボックス
  - 法線

  今回は利用しないが忘れないように

```
1   using UnityEngine;
2
    Unity スクリプト (1 件のアセット参照)|0 個の参照
3   public class PolygonMonoBehaviourScript : MonoBehaviour
4   {
5       // Start is called once before the first execution of Update
        Unity メッセージ|0 個の参照
6       void Start()
7       {
8           Vector3[] vertices =
9           {
10              new Vector3(1.0f, 0.0f, 0.0f),
11              new Vector3(0.0f, 1.0f, 0.0f),
12              new Vector3(0.0f, 0.0f, 1.0f),
13          };
14          Color[] colors = { Color.red, Color.green, Color.blue };
15          int[] triangles = {0, 1, 2};
16
17          Mesh Mesh = GetComponent<MeshFilter>().mesh;
18
19          Mesh.Clear();
20          Mesh.vertices = vertices;
21          Mesh.colors = colors;
22          Mesh.triangles = triangles;
23
24          Mesh.RecalculateBounds();
25          Mesh.RecalculateNormals();
26      }
27  }
```

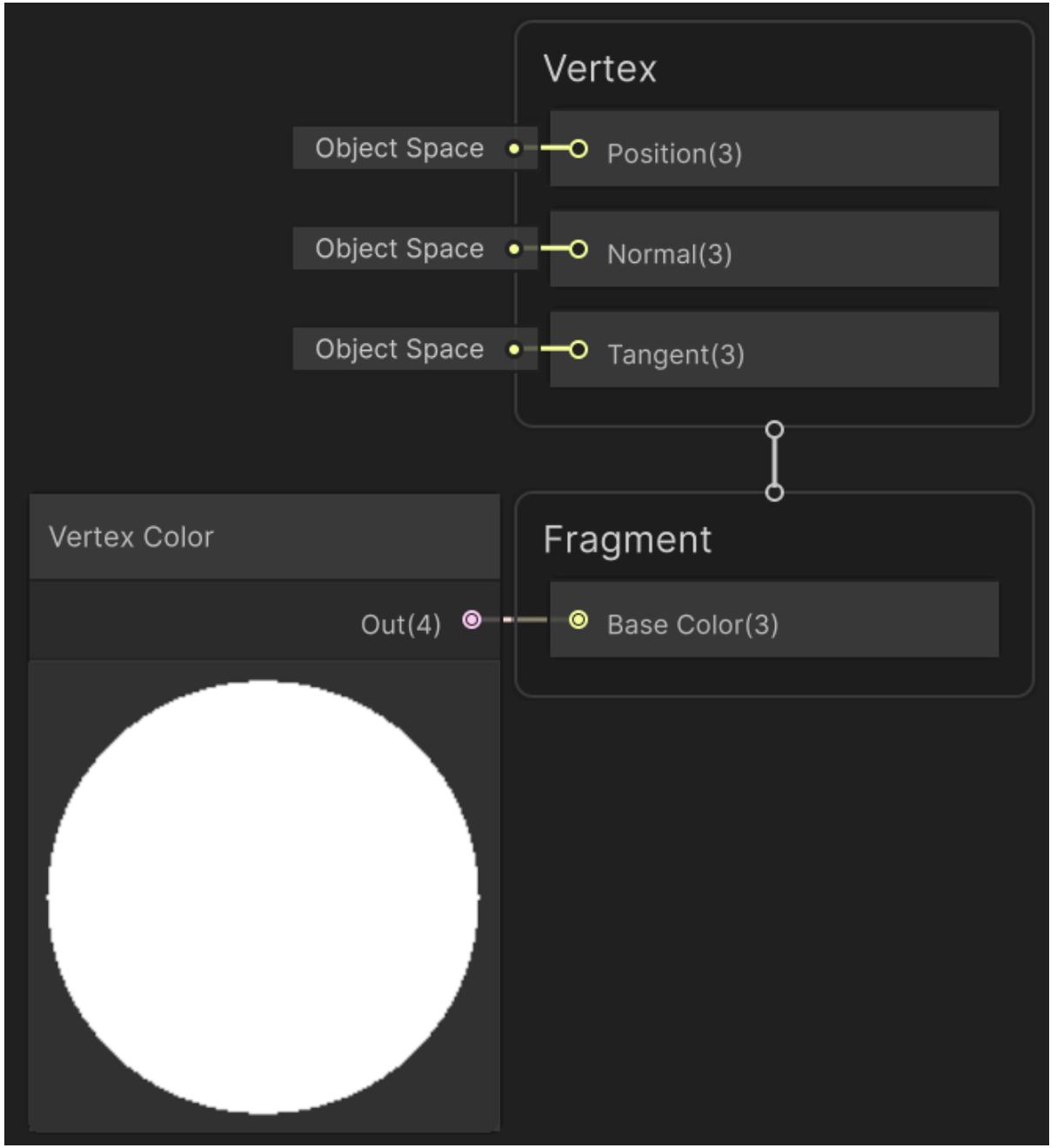
メッシュデータ

- トライアングルリスト
- 三角形のインデックスで頂点データを指示
  - 時計回りで指定



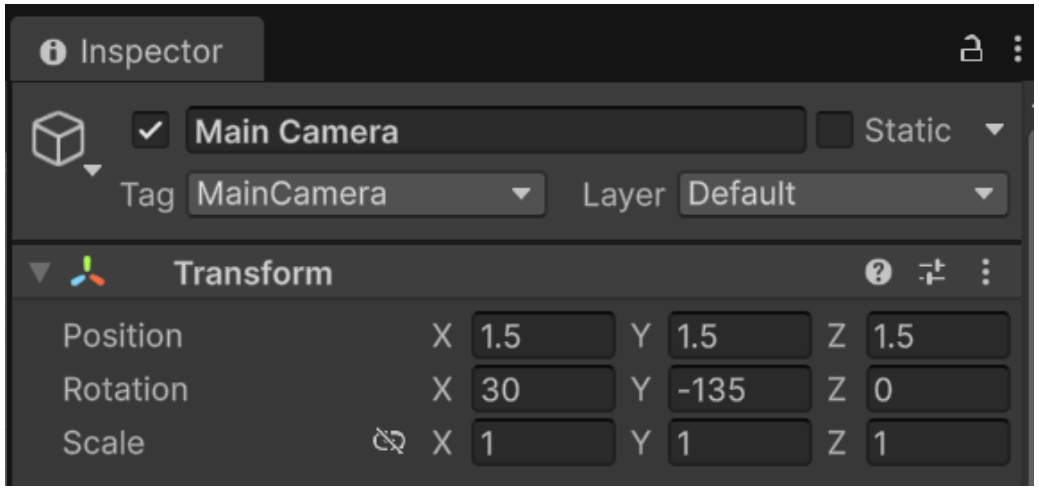
シェーダ

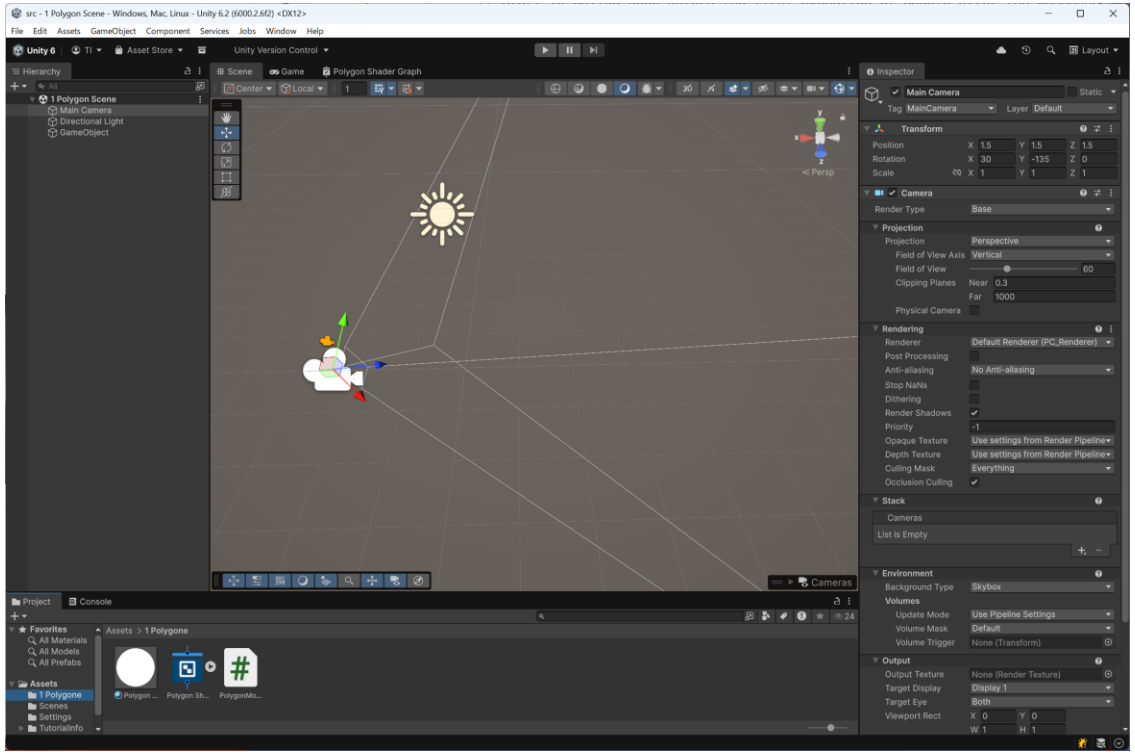
- 頂点色を出力


Vertex
Object Space
Position(3)
Object Space
Normal(3)
Object Space
Tangent(3)
Vertex Color
Fragment
Out(4)
Base Color(3)

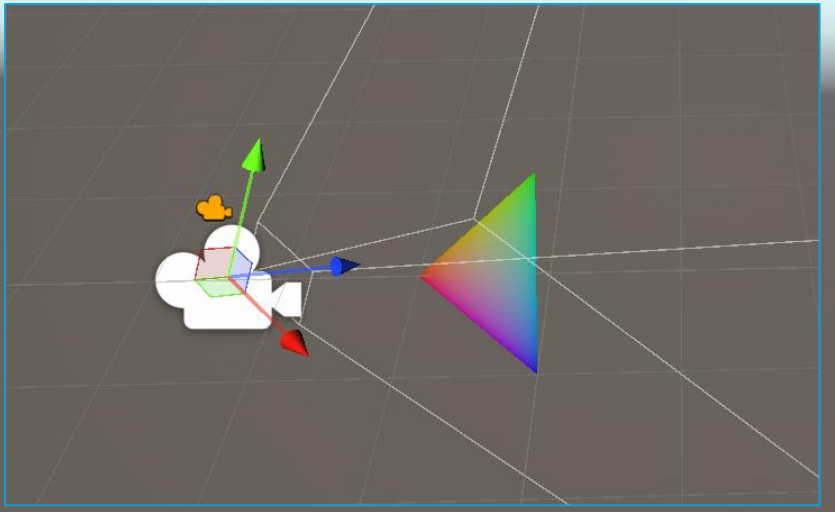
カメラレイアウト

- ゲームオブジェクトは原点に配置
- カメラを斜め上（1.5, 1.5, 1.5）から見下ろす
  - Rot: (30, -135, 0)

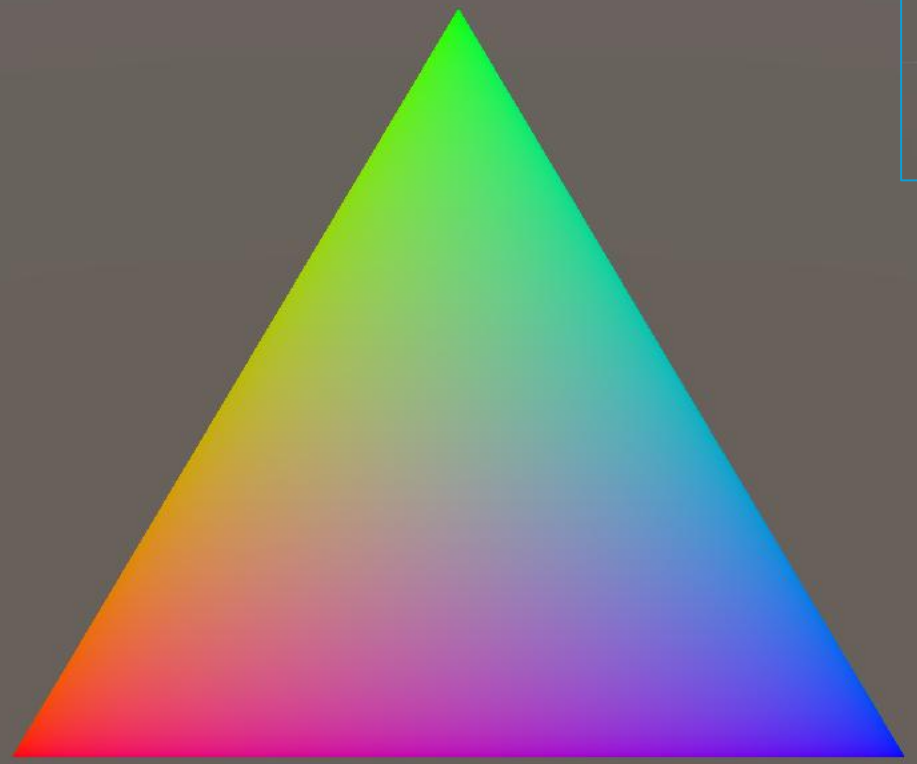




完成

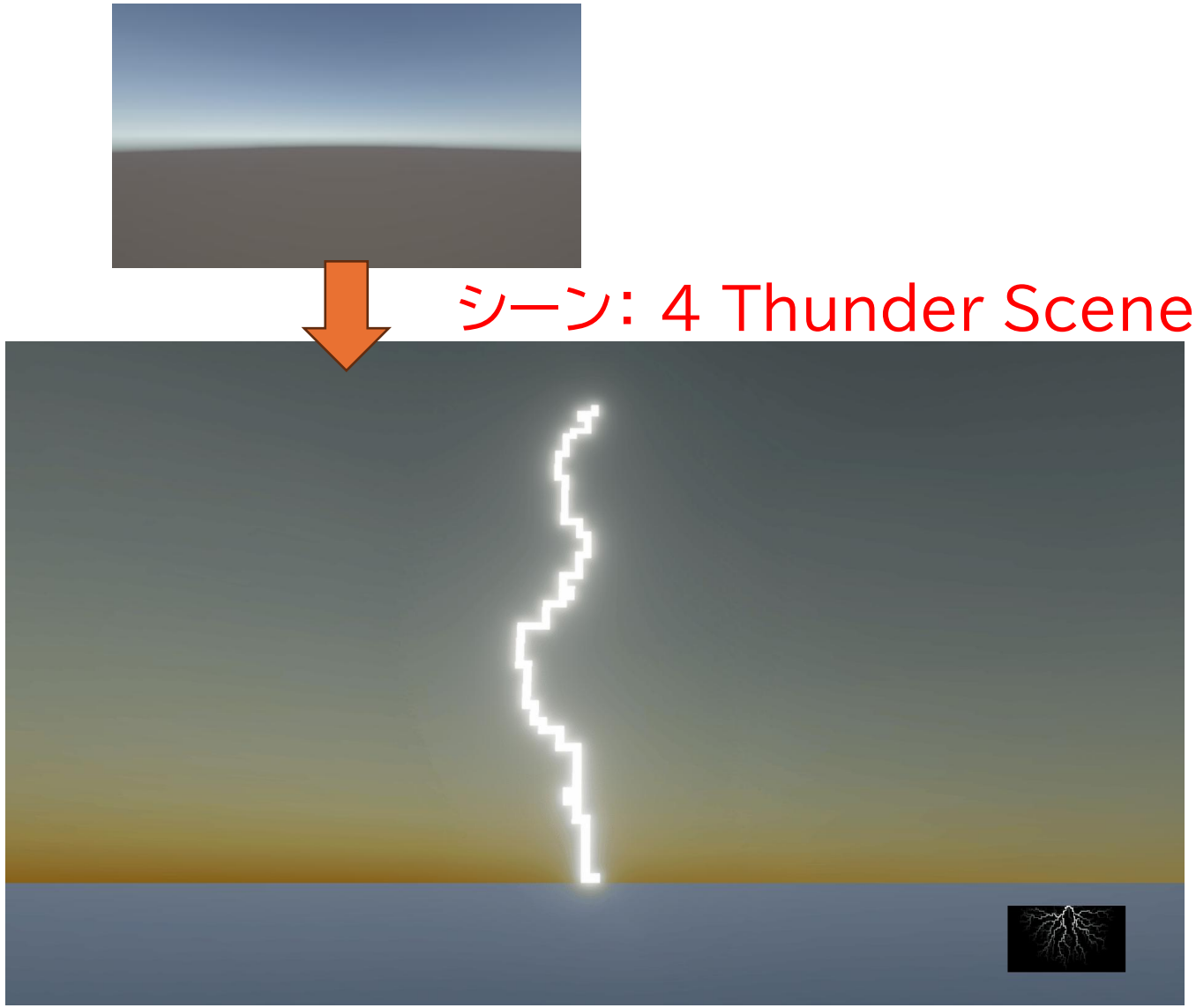


「シーン」タブでは実行したら現れる



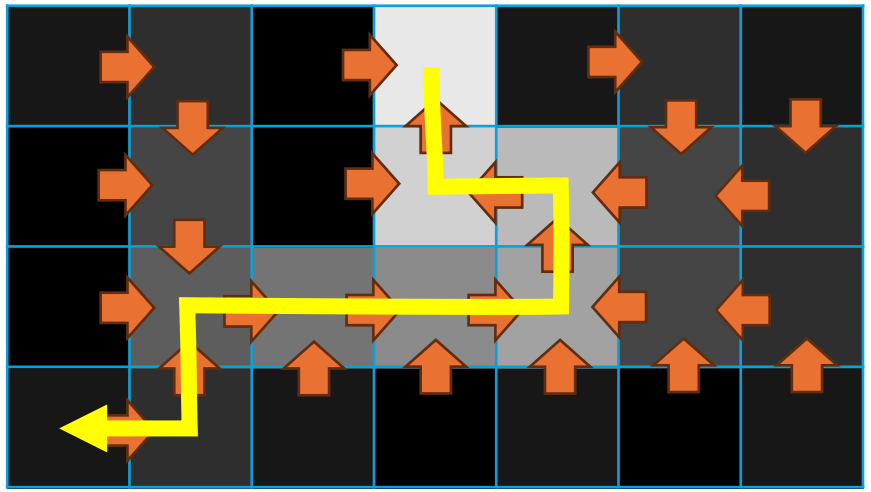
本日の内容

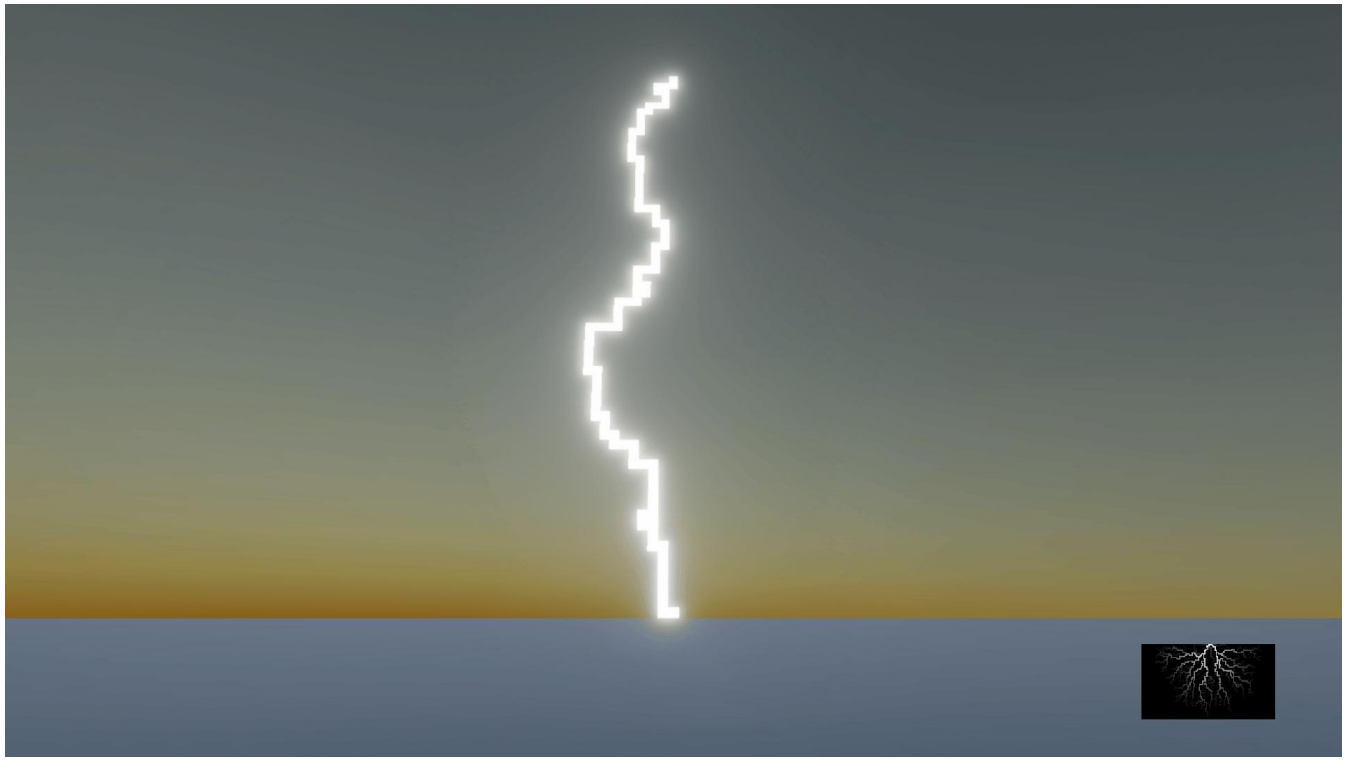
- CPUからのリソース生成
  - テクスチャの描画
  - リヒテンベルク図形
  - ポリゴンの描画
  - 雷
    - リヒテンベルク図形
    - ランダムウォーク



ポリゴンでの雷

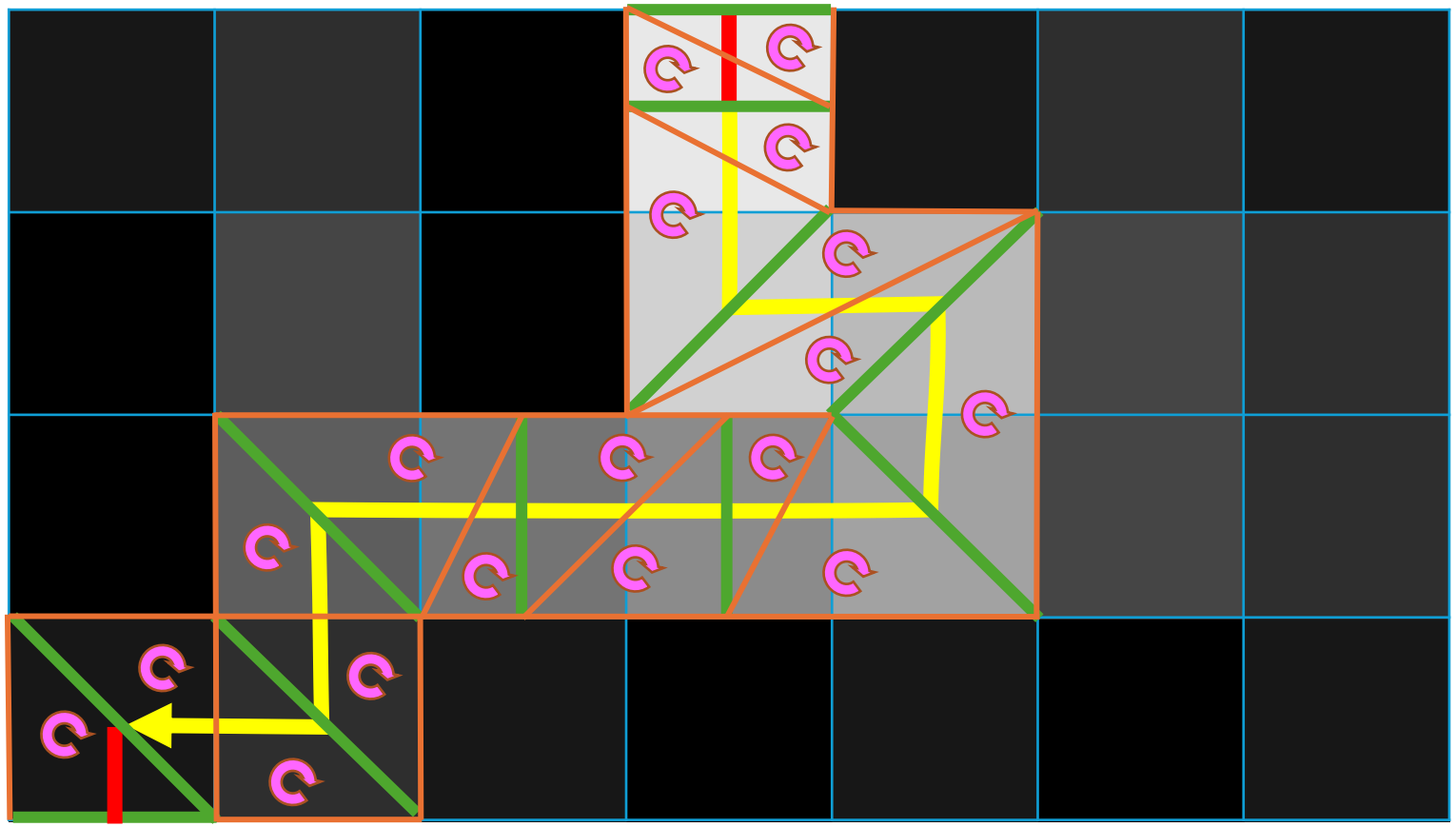
- リヒテンベルク図形で作ったパターンにおいて、一番最初に地面に到達した経路をポリゴンで表示する





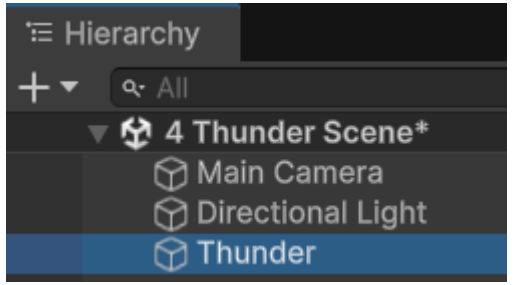
経路からポリゴンの生成

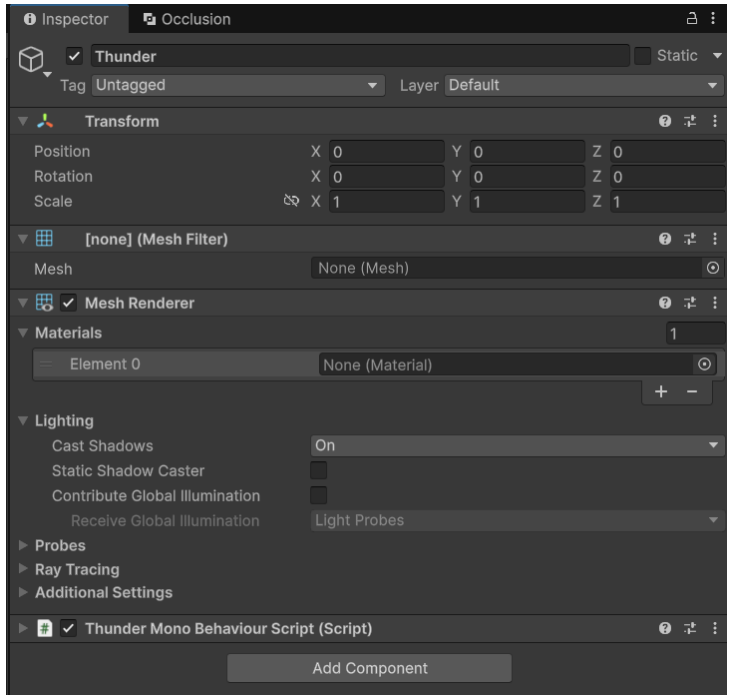
- 各セルの中心から幅となる線分の計算
  - 頂点で曲がる方向ー
    - 進行方向→を計算
      - $dir = (p_{i+1} - p_i) + (p_i - p_{i-1}) = p_{i+1} - p_{i-1}$
        - 前後の進行方向の平均（和）
      - $p[i+1] - p[i-1]$
      - 始点と終点は特殊処理
        - 鉛直方向ーに進む
    - 進行方向を90度回転
- 幅となる線分の頂点をつないでポリゴンを作る
  - 前後の幅によって作られる四角形を2つの三角形に分割



やってみよう

- Mono Behabiour Scriptの追加
  - 名称例:ThunderMonoBehaviourScript
- GameObjectの追加
  - Empty Objectでよい
  - 名称例:Thunder
  - コンポーネントの追加
    - Mesh Filter
    - Mesh Renderer
    - ThunderMonoBehaviourScript





メンバー変数

- メッシュ構築用
  - メッシュ
    - メッシュフィルターとして
      - meshFilter
  - マテリアル: material
- リヒテンベルク図形
  - lichtenberg
  - 大きさ
    - TEX_WIDTH
    - TEX_HEIGHT
- テクスチャ
  - 可視化用
- 更新時間: time
  - 1～3秒間隔で更新

ThunderMonoBehaviourScript.cs

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

Unity スクリプト (1 件のアセット参照)|0 個の参照
public class ThunderMonoBehaviourScript : MonoBehaviour
{
    [SerializeField] MeshFilter meshFilter = default!;
    [SerializeField] Material material = default!;
    Texture2D texture = null;
    [SerializeField] int TEX_WIDTH = 128;
    [SerializeField] int TEX_HEIGHT = 64;

    Lichtenberg lichtenberg = null;

    float time = 0.0f;

    // Start is called once before the first execution of Update after the MonoBehavi
    Unity メッセージ|0 個の参照
    void Start()
    {
        texture = new Texture2D(TEX_WIDTH, TEX_HEIGHT, TextureFormat.RGBA32, false);
        material.SetTexture("_Texture2D", texture);

        // テクスチャと関連するもの初期化
        lichtenberg = new Lichtenberg(TEX_HEIGHT, TEX_WIDTH,
            Lichtenberg.MODE.FINISH_AT_FIRST_ARRIVE);
        UpdateTexture();
    }

    Unity メッセージ|0 個の参照
    void Update()
    {
        time -= Time.deltaTime;
        if (time <= 0.0f)
        {
            Generate();
            time = Random.Range(1.0f, 3.0f);// 1秒から3秒の間隔で再生成
        }
    }
```

更新処理

- リヒテンベルク図形はwhileループで一気に生成
- 接地場所から親を辿ったリストとして経路生成
  - メッシュ生成に渡す

```
38      void Generate()
39      {
40          // 検索の初期化
41          int x = TEX_WIDTH / 2;
42          int y = TEX_HEIGHT - 1;// yが大きい方が上
43          lichtenberg.Initialize(y, x);
44
45          // 雷テクスチャの生成
46          while (lichtenberg.Update() == Lichtenberg.State.Running) ;
47          UpdateTexture();
48
49          // 到達経路の取得
50          List<int> path = new List<int>();
51
52          int p = lichtenberg.ArriveIndex;
53          while (lichtenberg.Parent[p] != p)
54          {
55              path.Add(p);
56              p = lichtenberg.Parent[p];
57          }
58          path.Add(p);// 最後に根元を追加
59
60          // メッシュの構築
61          UpdateMesh(path);
62      }
```

テクスチャ生成

- 可視化用:リヒテンベルク図形の末端からの距離　ThunderMonoBehaviourScript.cs

```
64      void UpdateTexture()
65      {
66          var pixelData = texture.GetPixelData<Color32>(0);
67
68          ushort[] value = lichtenberg.Value;
69          double vMax = (double)lichtenberg.ValueMax;
70
71          int idx = 0;
72          for (int y = 0; y < TEX_HEIGHT; y++)
73          {
74              for (int x = 0; x < TEX_WIDTH; x++)
75              {
76                  byte c = (byte)(256.0 * (double)value[idx] / (vMax + 1));// 256にはならないようにする
77                  pixelData[idx] = new Color32(c, c, c, 255);
78                  idx++;
79              }
80          }
```

ThunderMonoBehaviourScript.cs

メッシュ生成

- 基本的にはポリゴン数が増えただけ

```
84      void UpdateMesh(List<int> path)
85      {
86          int n = path.Count;
87
88          // 頂点の生成
89          int vertexCount = 2 * (n + 2);// 始点と終点に2頂点ずつ追加
90          Vector3[] vertices = new Vector3[vertexCount];
91          float SIZE = 1000.0f / (float)TEX_HEIGHT;
92          float halfWidth = SIZE * 0.5f;
93
94          int vtx = 0;
95          // 始点
96          ...
104         for (int i = 0; i < n; i++)...
149         // 終点
150         ...
159
160         Color[] colors = new Color[vertexCount];
161         for(int i = 0; i < vertexCount; i++)...
165
166         // インデックスの生成
167         int polygonCount = 2 * (n + 1);// 四角形の数
168         int[] triangles = new int[3 * polygonCount];
169         int idx = 0;
170         vtx = 0;
171         while(idx < 3 * polygonCount)...
184
185         //  メッシュの更新
186         Mesh Mesh = meshFilter.mesh;
187
188         Mesh.Clear();
189         Mesh.vertices = vertices;
190         Mesh.colors = colors;
191         Mesh.triangles = triangles;
192
193         Mesh.RecalculateBounds();
194         Mesh.RecalculateNormals();
195     }
196 }
```

頂点カラー

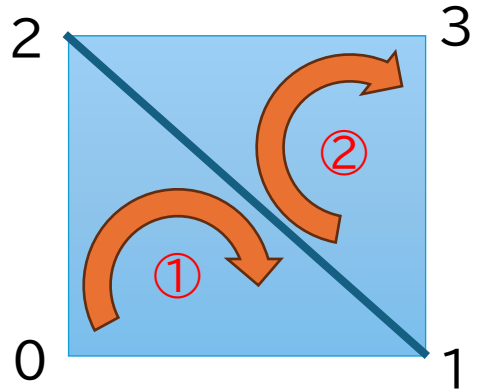
- 白のみ

ThunderMonoBehaviourScript.cs

```
160         Color[] colors = new Color[vertexCount];
161         for(int i = 0; i < vertexCount; i++)
162         {
163             colors[i] = Color.white;
164         }
```

頂点インデックス

- 四角形を3角形ポリゴン2枚で表現
- n(経路の頂点数)＋2(両端に追加)個



ThunderMonoBehaviourScript.cs

```
166         // インデックスの生成
167         int polygonCount = 2 * (n + 1);// 四角形の数
168         int[] triangles = new int[3 * polygonCount];
169         int idx = 0;
170         vtx = 0;
171         while(idx < 3 * polygonCount)
172         {
173             triangles[idx + 0] = vtx + 0;
174 ①           triangles[idx + 1] = vtx + 2;
175             triangles[idx + 2] = vtx + 1;
176
177             triangles[idx + 3] = vtx + 1;
178 ②           triangles[idx + 4] = vtx + 2;
179             triangles[idx + 5] = vtx + 3;
180
181             idx += 6;
182             vtx += 2;
183         }
```

端点処理

- 下端と上端に水平に頂点を追加
  - Vector3.left方向
- 下端の点の真下と上端の点の真上
  - 高さ範囲:[0,1000]
  - 幅はセルが正方形で表示されるように調整
- X軸の原点はリヒテンベルク図形の横方向の中心に合わせる

ThunderMonoBehaviourScript.cs

```
84      void UpdateMesh(List<int> path)
85      {
86          int n = path.Count;
87
88          // 頂点の生成
89          int vertexCount = 2 * (n + 2);// 始点と終点に2頂点ずつ追加
90          Vector3[] vertices = new Vector3[vertexCount];
91          float SIZE = 1000.0f / (float)TEX_HEIGHT;
92          float halfWidth = SIZE * 0.5f;
93
94          int vtx = 0;
95          // 始点
96          {
97              int index = path[0];
98              int x = index % TEX_WIDTH - TEX_WIDTH / 2;
99              Vector3 center = new Vector3(halfWidth * 2.0f * (float)x, 0.0f, 0.0f);
100             vertices[vtx + 0] = center + Vector3.left * halfWidth;
101             vertices[vtx + 1] = center - Vector3.left * halfWidth;
102             vtx += 2;
103         }
104         for (int i = 0; i < n; i++)...
149         // 終点
150         {
151             int index = path[n - 1];
152             int x = index % TEX_WIDTH - TEX_WIDTH / 2;
153             int y = TEX_HEIGHT;
154             Vector3 center = new Vector3(halfWidth * 2.0f * (float)x, SIZE * (float)TEX_HEIGHT, 0.0f);
155             vertices[vtx + 0] = center + Vector3.left * halfWidth;
156             vertices[vtx + 1] = center - Vector3.left * halfWidth;
157             vtx += 2;
158         }
```

経路の頂点

- 前後の頂点の差から向きを計算
  - xyを入れ替え、片方の符号を変えてて回転
- 横方向の向きが45度の際に長さが√2倍になるよう補正
  - 結果として差をDotで2乗した大きさを割るのが正しい

ThunderMonoBehaviourScript.cs

```
104         for (int i = 0; i < n; i++)
105         {
106             int index = path[i];
107             int x = index % TEX_WIDTH - TEX_WIDTH / 2;
108             int y = index / TEX_WIDTH;
109             Vector3 center = new Vector3(halfWidth * 2.0f * (float)x, SIZE * (0.5f + (float)y), 0.0f);
110             // 幅方向のベクトルを計算
111             Vector3 right;
112             if (i == 0)...
123             else if(i < n - 1)
124             {
125                 // 端点以外は前後の差分で方向を決定
126                 int nextIndex = path[i + 1];
127                 int nextX = nextIndex % TEX_WIDTH;
128                 int nextY = nextIndex / TEX_WIDTH;
129                 int prevIndex = path[i - 1];
130                 int prevX = prevIndex % TEX_WIDTH;
131                 int prevY = prevIndex / TEX_WIDTH;
132                 right = new Vector3(-(float)(nextY - prevY), (float)(nextX - prevX), 0.0f);
133             }
134             else...
145             vertices[vtx + 0] = center + right * 2.0f * halfWidth / Vector3.Dot(right, right);
146             vertices[vtx + 1] = center - right * 2.0f * halfWidth / Vector3.Dot(right, right);
147             vtx += 2;
148         }
```

$$\vec{r} = \begin{bmatrix} \cos\frac{\pi}{2} & \sin\frac{\pi}{2} \\ \sin\frac{\pi}{2} & \cos\frac{\pi}{2} \end{bmatrix} \overrightarrow{dir} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \overrightarrow{dir} = \begin{bmatrix} -dir_y \\ dir_x \end{bmatrix}$$

始点と終点

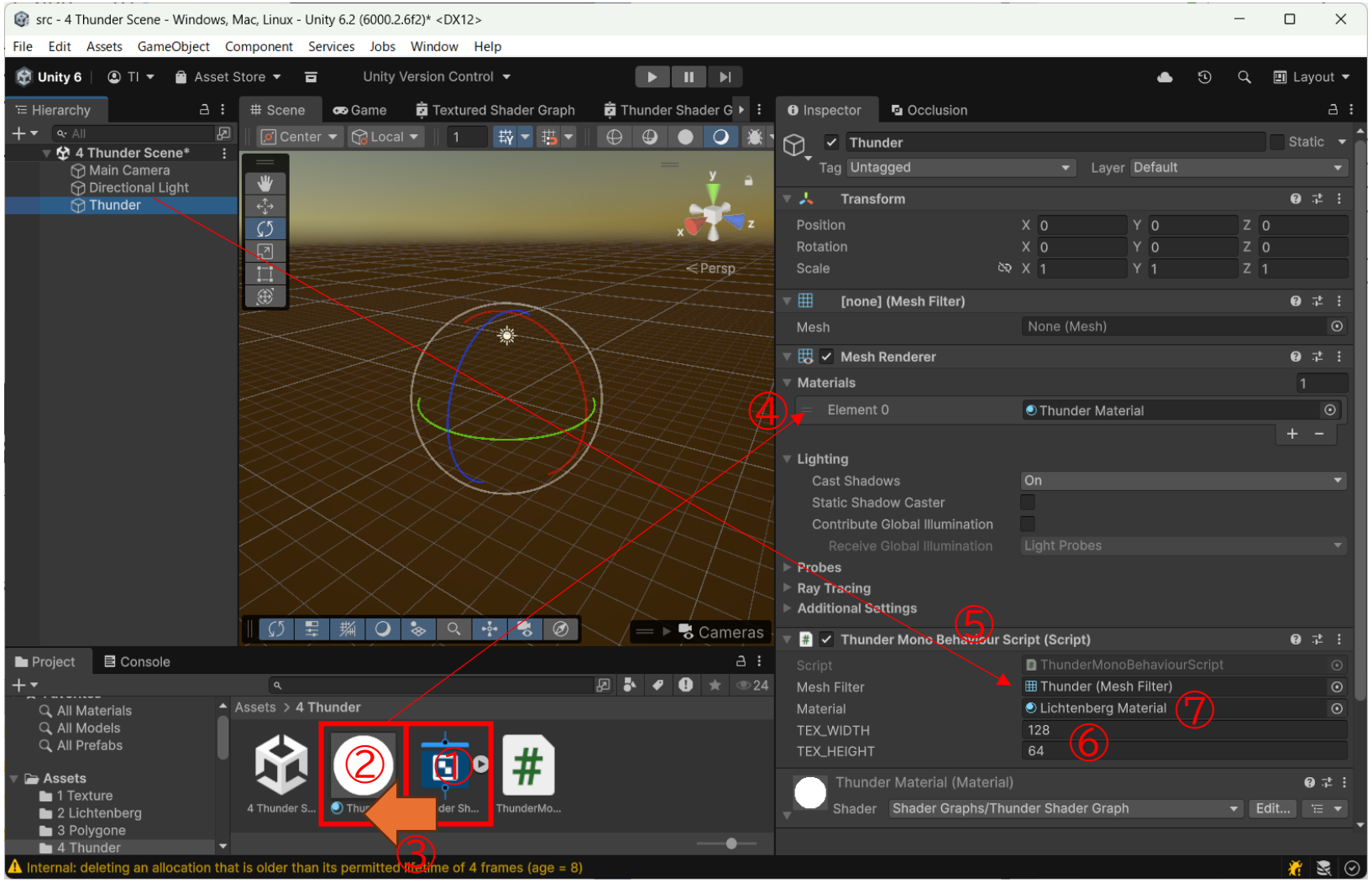
- 前か後の頂点がないので、端点の真下と真上の点との差から方向を決定

ThunderMonoBehaviourScript.cs

```
110             // 幅方向のベクトルを計算
111             Vector3 right;
112             if (i == 0)
113             {
114                 // 始点は次の点と始点の一つ下の点の差分で方向を決定
115                 int nextIndex = path[1];
116                 int nextX = nextIndex % TEX_WIDTH;
117                 int nextY = nextIndex / TEX_WIDTH;
118                 int prevIndex = path[0];
119                 int prevX = prevIndex % TEX_WIDTH;
120                 int prevY = prevIndex / TEX_WIDTH - 1;
121                 right = new Vector3(-(float)(nextY - prevY), (float)(nextX - prevX), 0.0f);
122             }
123             else if(i < n - 1)
124             {
125                 // 端点以外は前後の差分で方向を決定
126                 int nextIndex = path[i + 1];
127                 int nextX = nextIndex % TEX_WIDTH;
128                 int nextY = nextIndex / TEX_WIDTH;
129                 int prevIndex = path[i - 1];
130                 int prevX = prevIndex % TEX_WIDTH;
131                 int prevY = prevIndex / TEX_WIDTH;
132                 right = new Vector3(-(float)(nextY - prevY), (float)(nextX - prevX), 0.0f);
133             }
134             else
135             {
136                 // 終点は前の点と終点の一つ上の点の差分で方向を決定
137                 int nextIndex = path[n - 1];
138                 int nextX = nextIndex % TEX_WIDTH;
139                 int nextY = nextIndex / TEX_WIDTH + 1;
140                 int prevIndex = path[n - 2];
141                 int prevX = prevIndex % TEX_WIDTH;
142                 int prevY = prevIndex / TEX_WIDTH;
143                 right = new Vector3(-(float)(nextY - prevY), (float)(nextX - prevX), 0.0f);
144             }
```

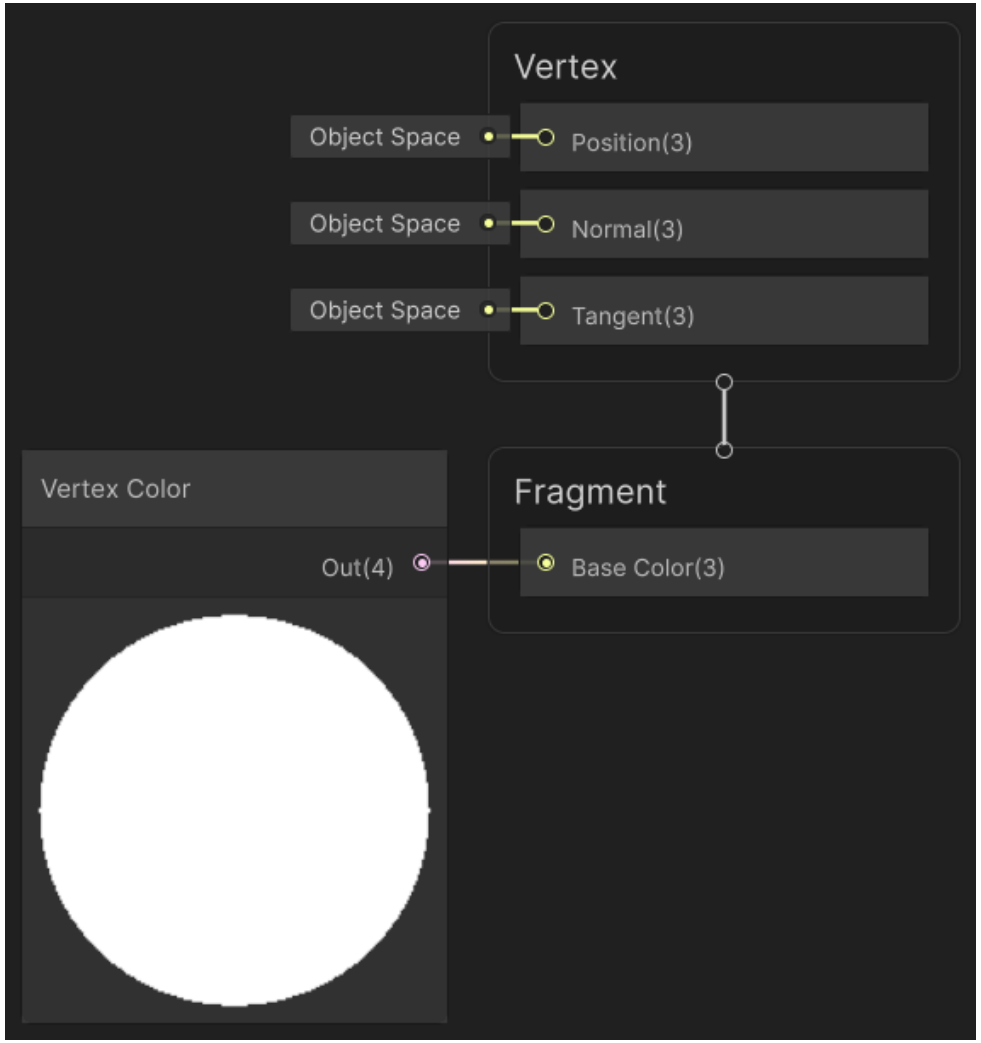
マテリアル

1. シェーダグラフを追加
   - Unlit Shader Graph
   - 名称例：Thunder Shader Graph
2. マテリアルを追加
   - 名称例：Thunder Material
   3. Thunder Shader Graphを設定
   4. ThunderオブジェクトのMaterialに設定
- ThunderMonoBehaviour Scriptの設定
   5. Mesh FilterにThunderオブジェクト(自分自身)を設定
   6. 幅や高さは適当に設定
   7. Materialは後で…



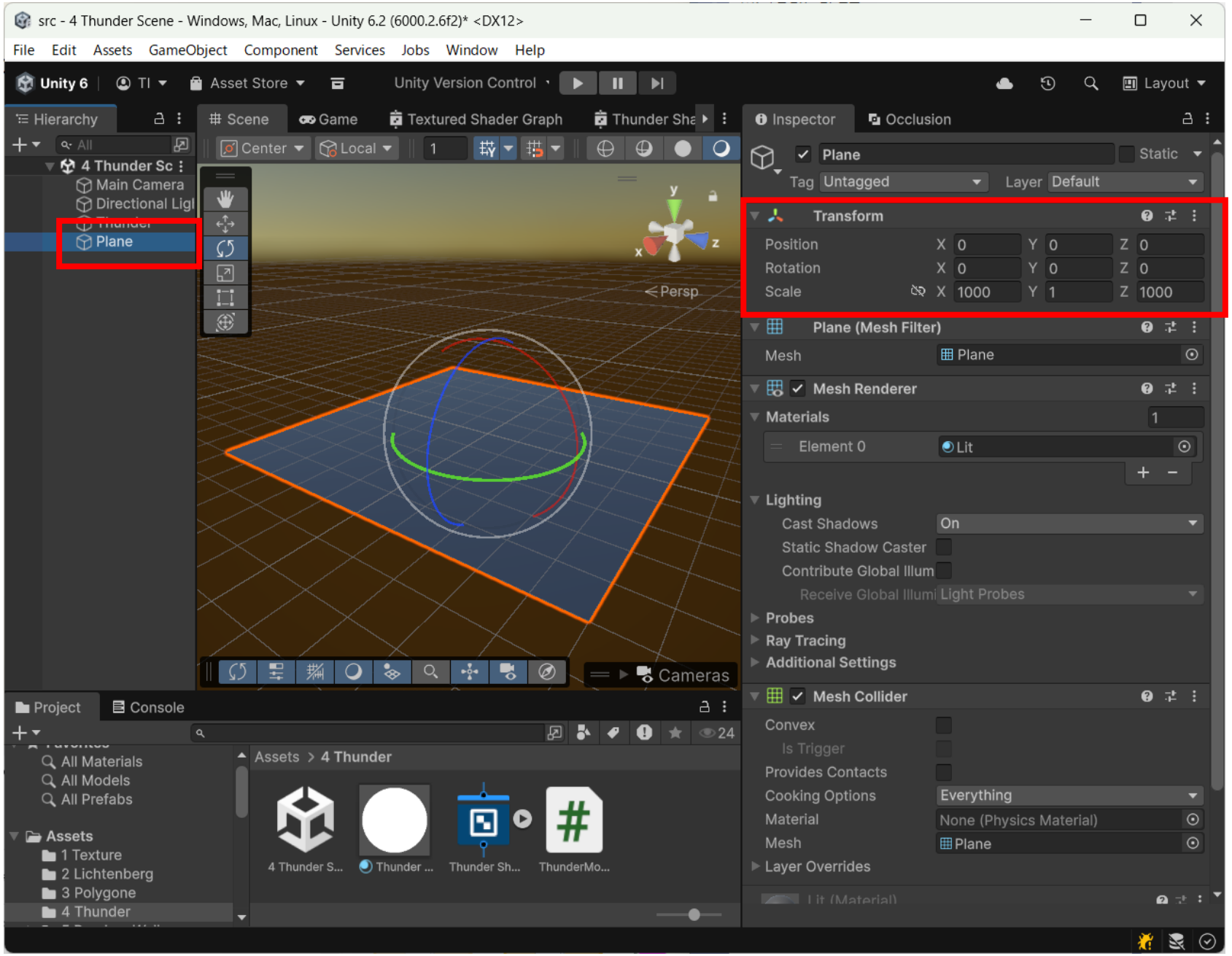
Thunder Shader Graph

- 頂点色を出力



床

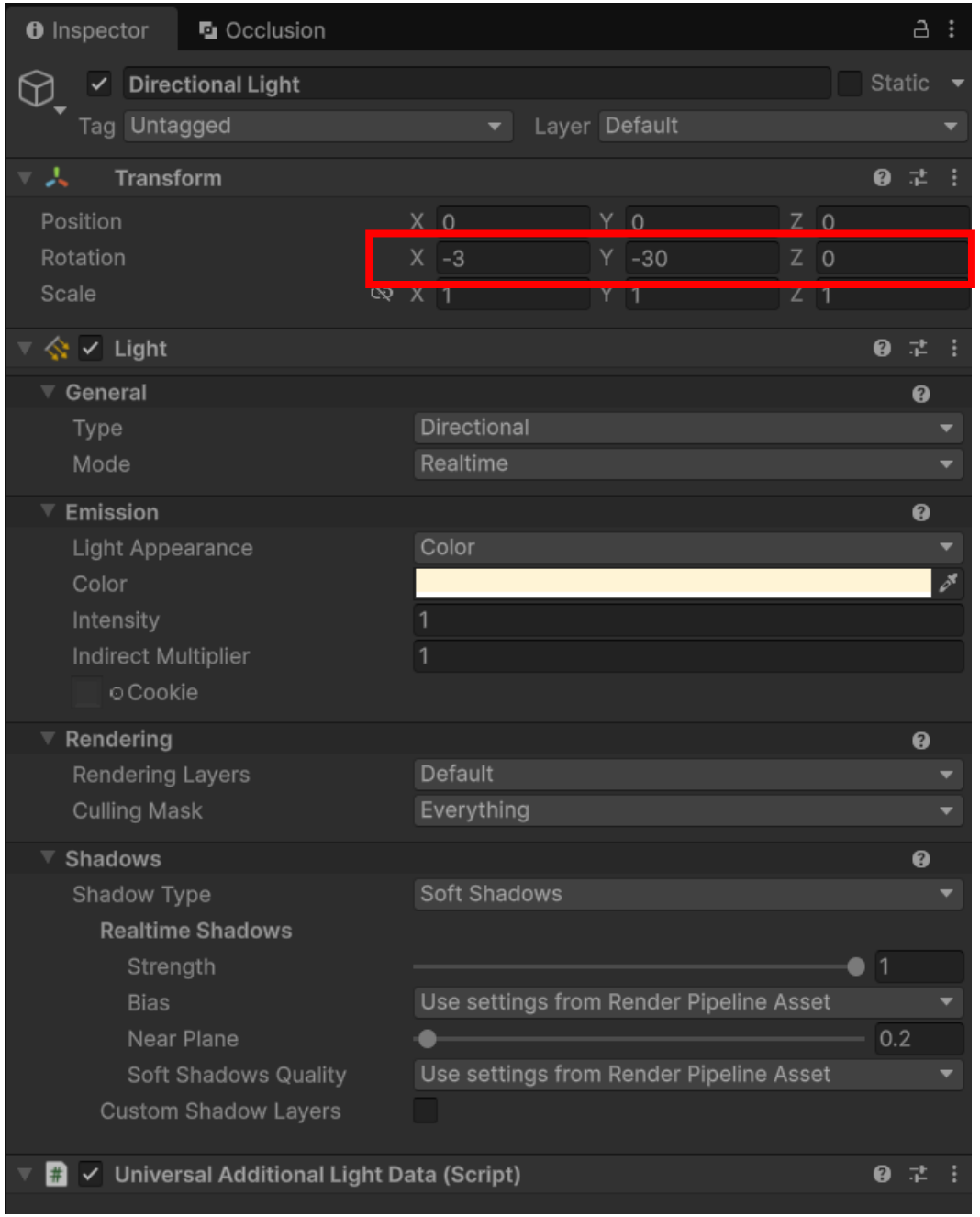
- Planeオブジェクトを追加
- 原点中心
- 大きさは1000倍



Directional Light

- 光の向きを浅くして夕方を表現
  - 雷が見やすいように

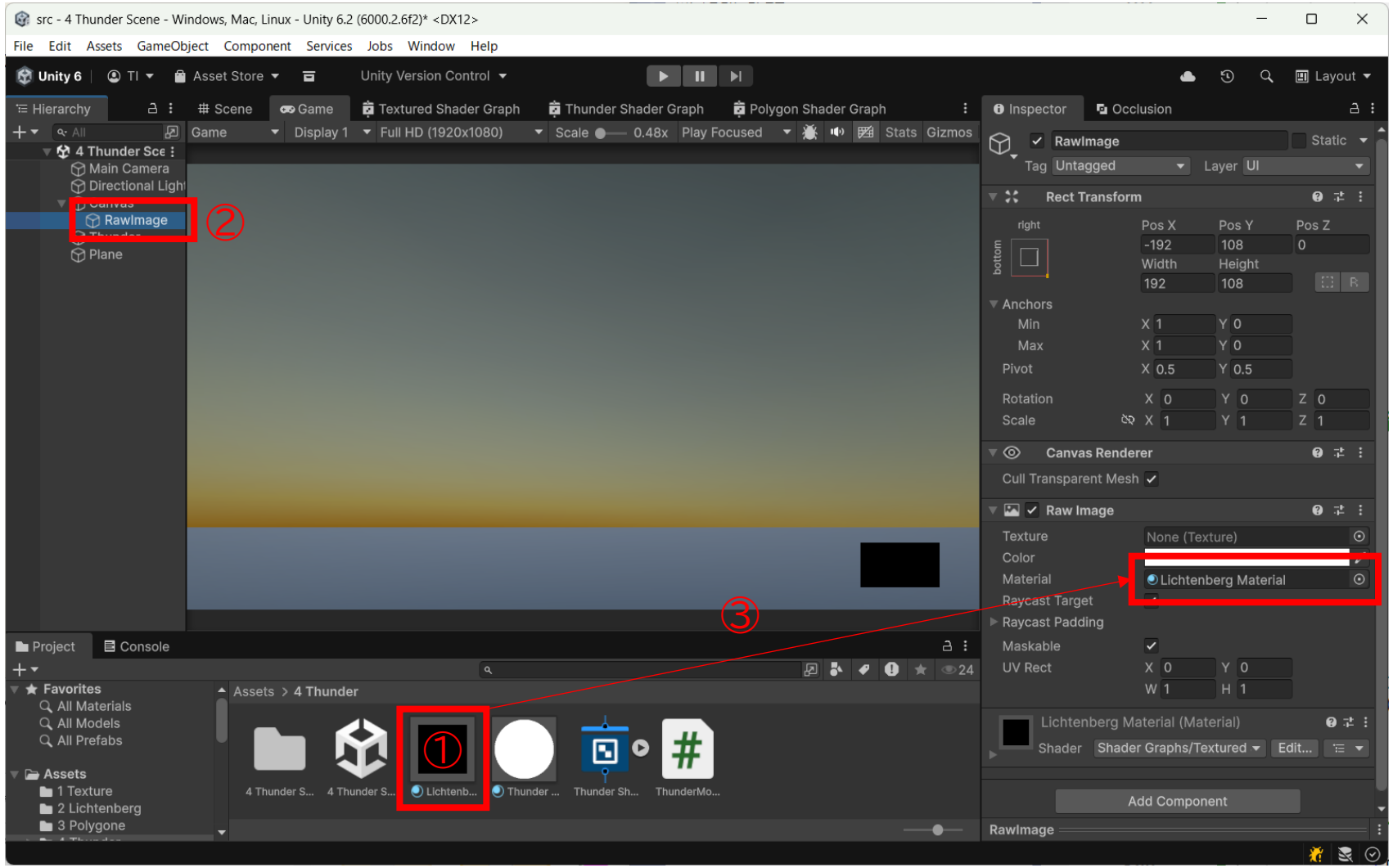
重要ではないので、お好みで



リヒテンベルク図形の可視化（1/2）

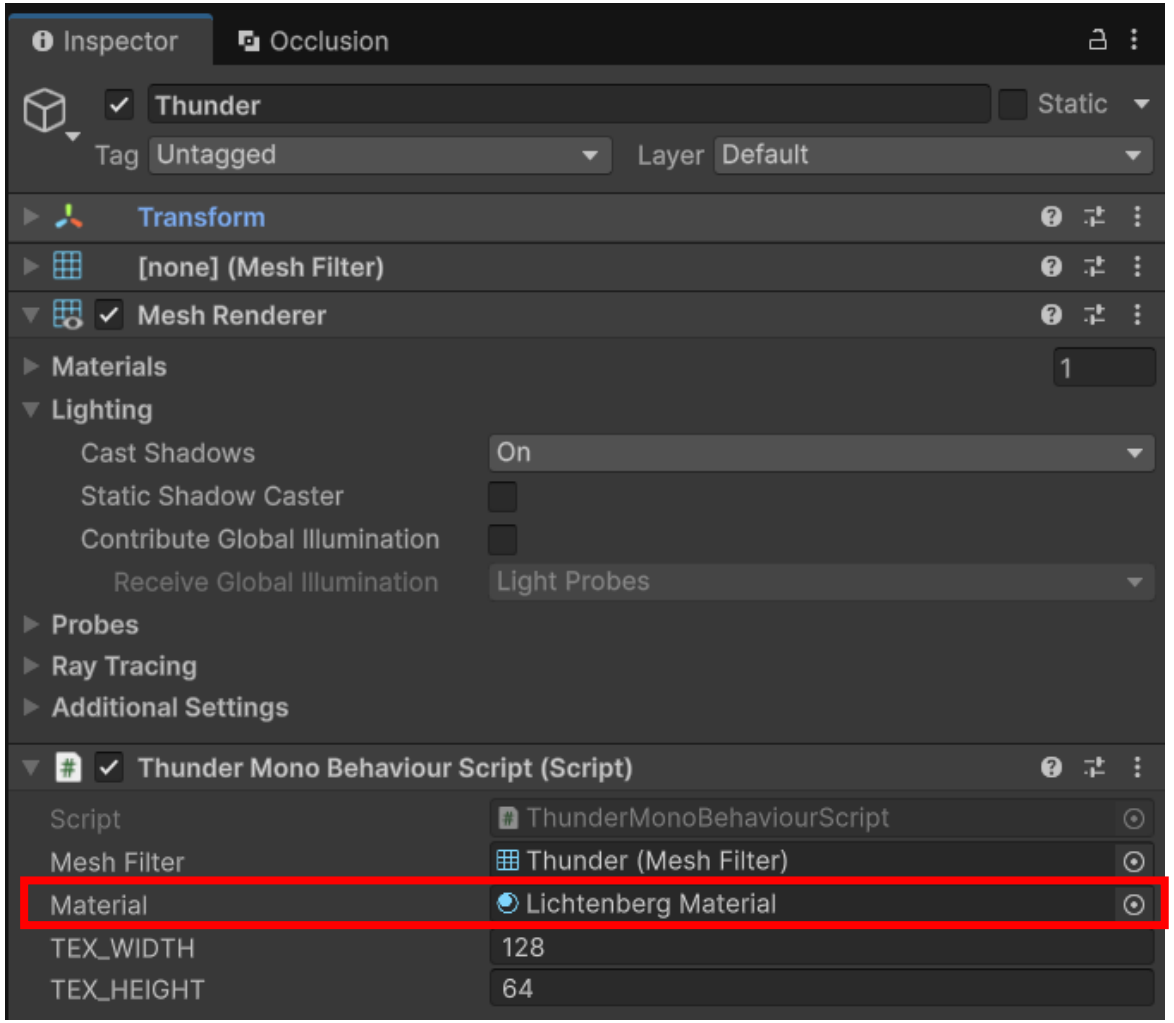
左下に表示

1. マテリアルの追加
   - 名称例：Lichtenberg Material
   - 「1 Texture/Textured Shader Graph」を設定
2. Raw Imageオブジェクトの追加
   3. Lichtenberg Materialを設定



リヒテンベルク図形の可視化（2/2）

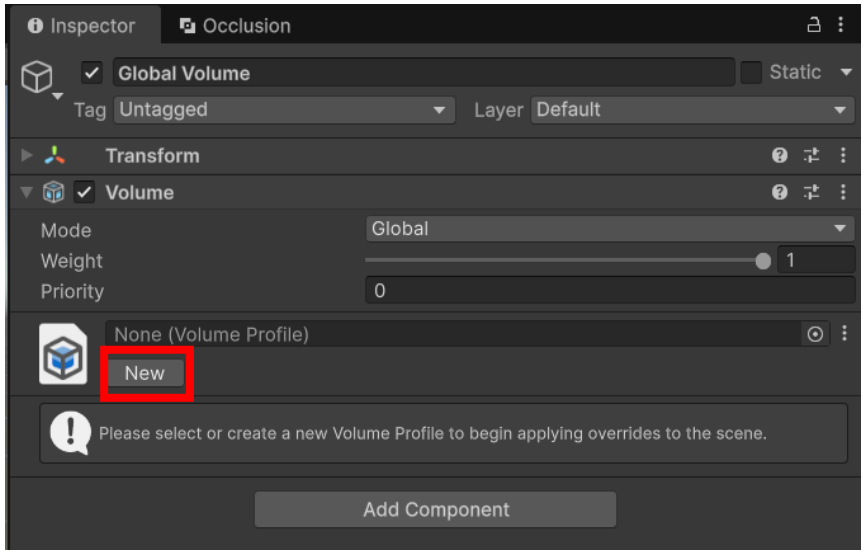
- ThunderMonoBehaviourScriptの設定
  - Materialに「 Lichtenberg Material」を追加

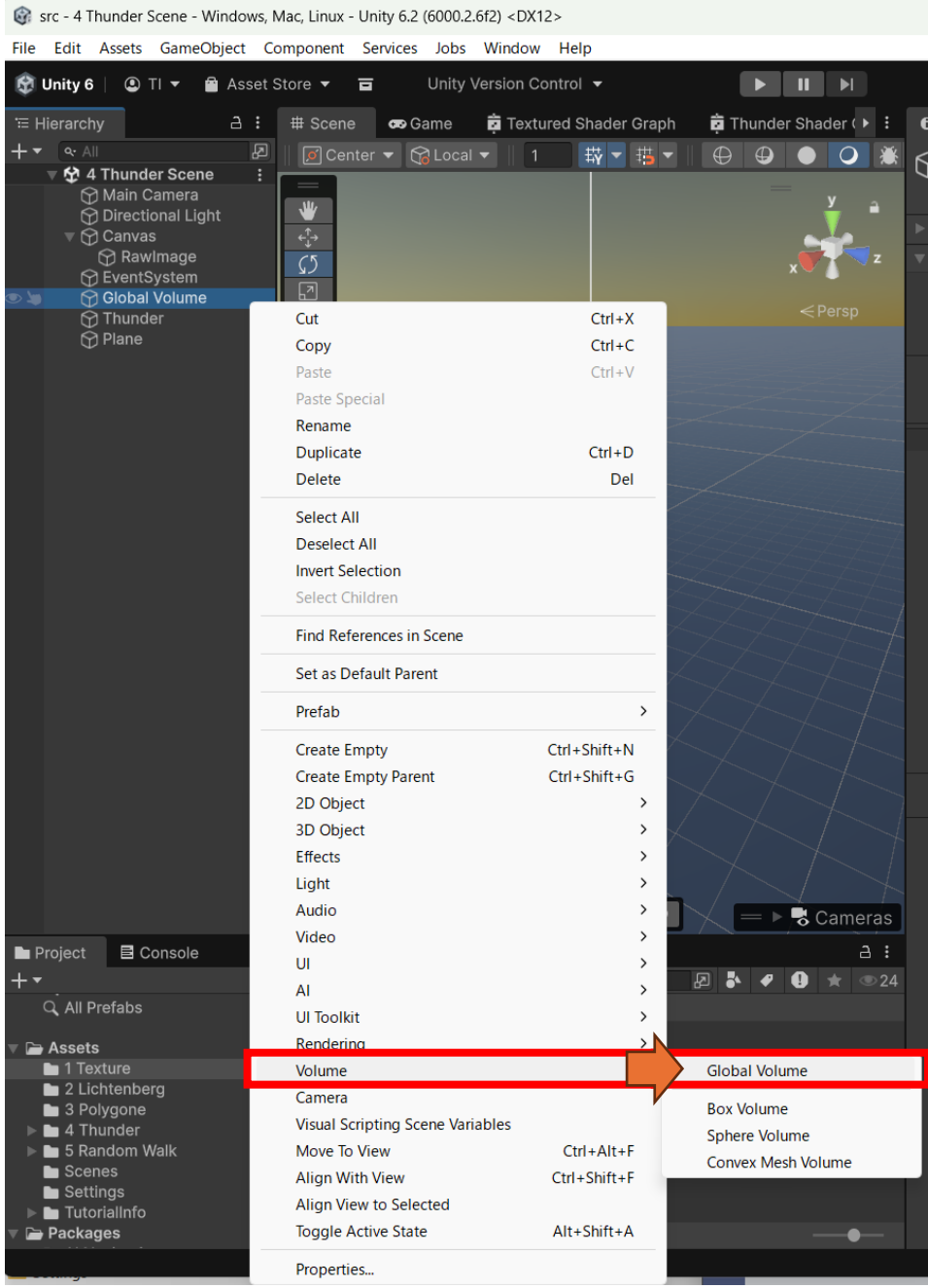


Bloom: ぼんやり光らせる

組み込み済みのポストエフェクト

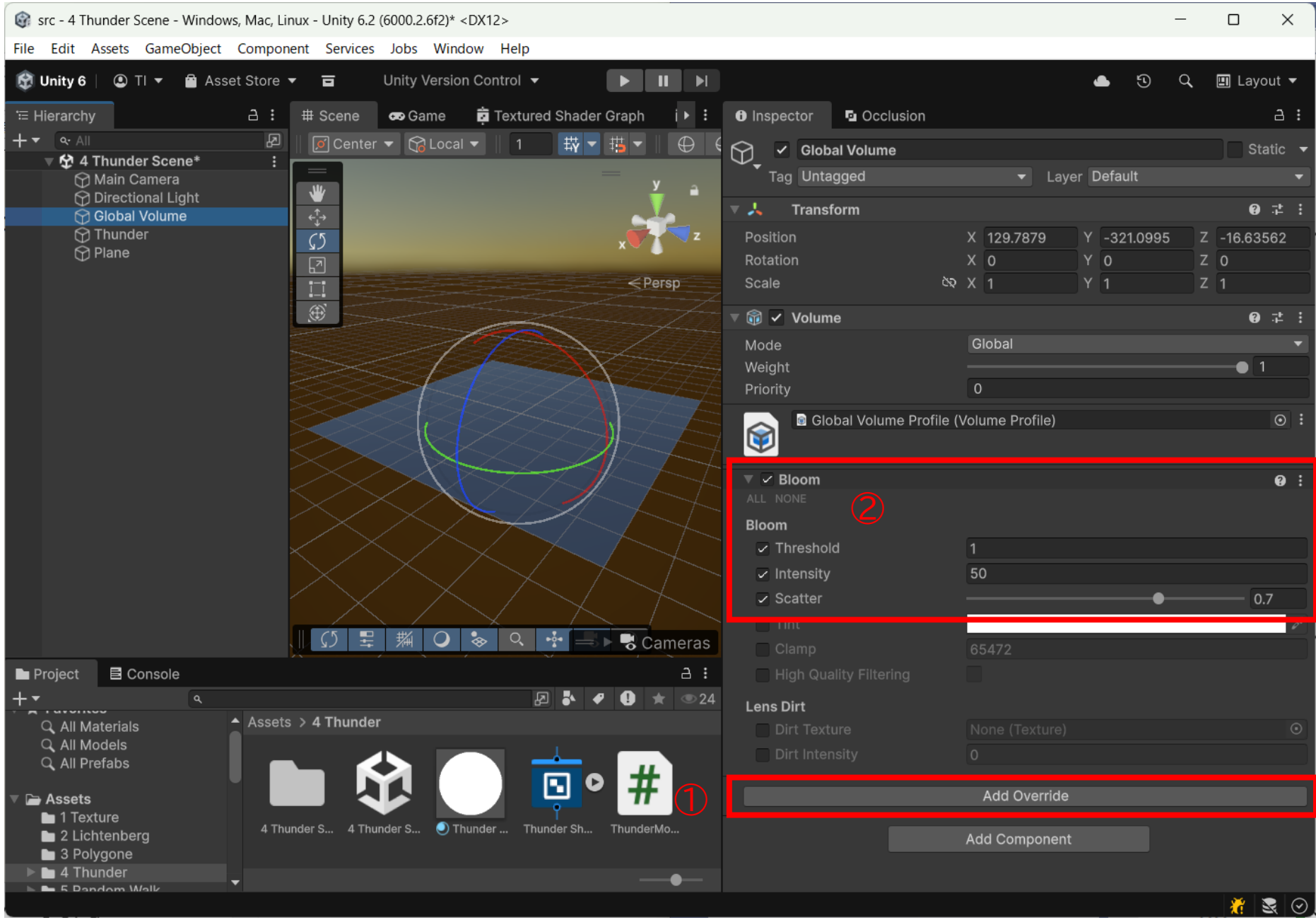
- 範囲を指定するために「Global Volume」を追加
- Volumeプロファイルを追加





Bloomの追加

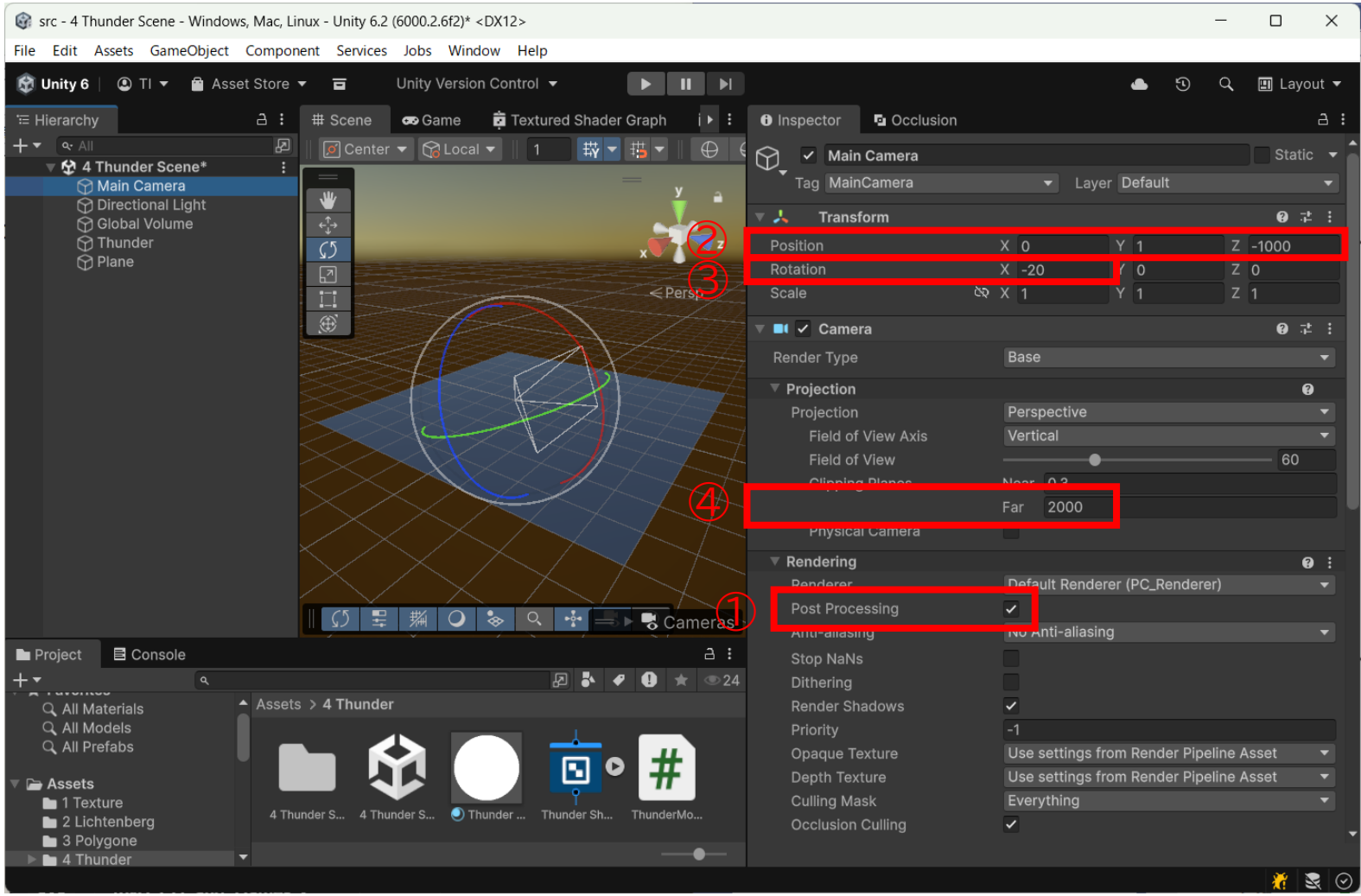
1. 「Add Override」からBloomを追加
2. Threshold、IntensityやScatterを有効にして値を設定
   - 後でみていい感じに



カメラの設定

1. 「Post Processing」を有効にしないとBloomがかからない

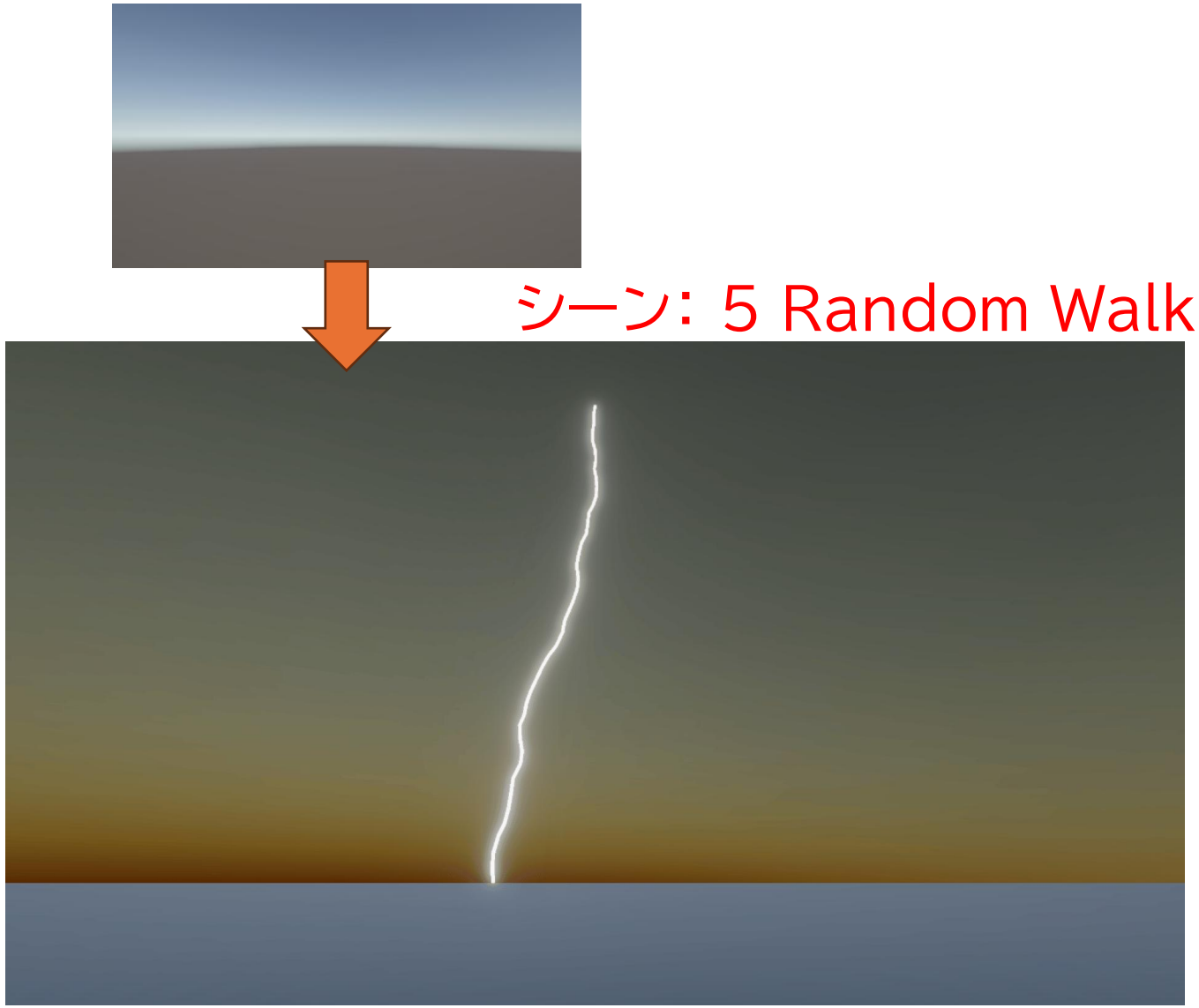
- その他、カメラの設定を調整
  2. 遠く(0,1,-1000)から見る
  3. 少し上を見る(-20度)
  4. 遠方クリッピング距離を遠くにする
     - 2000



完成

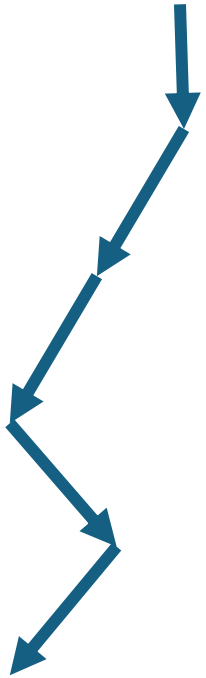
本日の内容

- CPUからのリソース生成
  - テクスチャの描画
  - リヒテンベルク図形
  - ポリゴンの描画
  - 雷
    - リヒテンベルク図形
    - ランダムウォーク



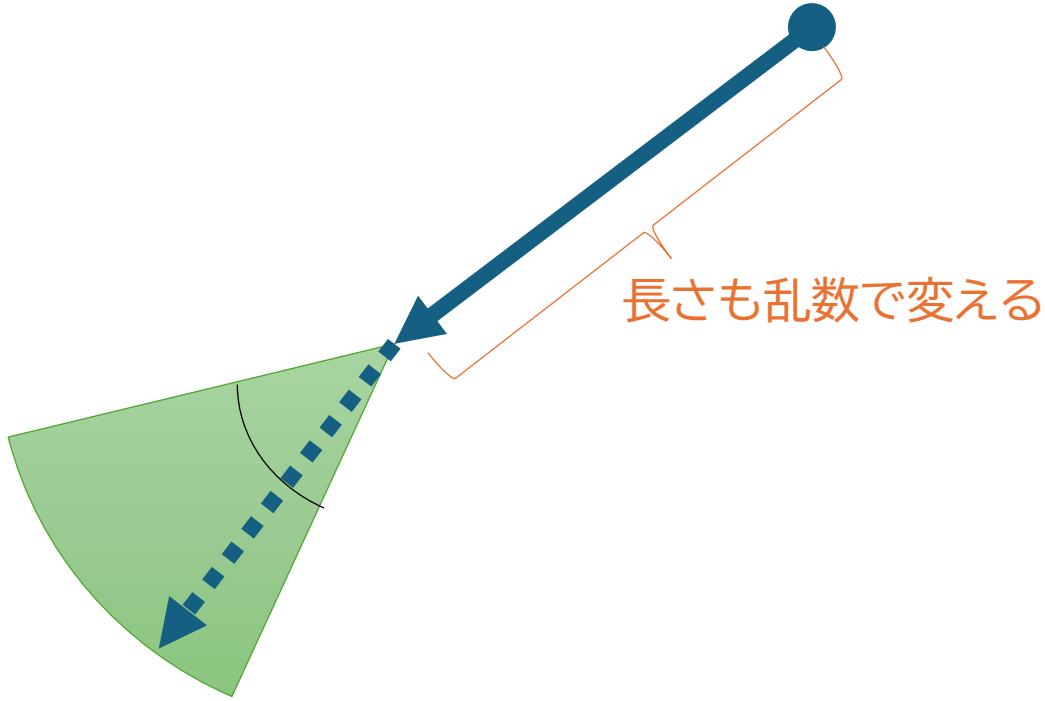
酔歩(ランダムウォーク)

- 次に進む向きがその時の状態によって確率的に決まる



今回のランダムウォーク：乱雑なジグザグ

- 次の向きを今の向きから一定範囲内のランダムで変える
  - 下向きから始める
  - y=1000から始めてy<0になったら終了
    - 長いステップ地面につかなかったら強制終了



スクリプトを追加

- Mono Behaviour Script
  - 名称例:RandomWalkMonoBehaviourScript
  - ThunderMonoBehaviourScript の中身をコピーして修正が手早い
  - 調整用のパラメータを導入

RandomWalkMonoBehaviourScript.cs

```
1   using UnityEngine;
2   using System.Collections.Generic;
3
        Unity スクリプト (1 件のアセット参照)|0 個の参照
4   public class RandomWalkMonoBehaviourScript : MonoBehaviour
5   {
6       [SerializeField] MeshFilter meshFilter = default!;
7
8       [SerializeField][Range(0.0f, 90.0f)] private float bendMax = 70.0f;
9       [SerializeField][Range(0.0f, 1.0f)] private float gravityRate = 0.3f;
10      [SerializeField][Range(0.0f, 100.0f)] private float distanceMin = 3.0f;
11      [SerializeField][Range(0.0f, 100.0f)] private float distanceMax = 20.0f;
```

更新

- 1-3秒に一回更新する

RandomWalkMonoBehaviourScript.cs

```
13          float time = 0.0f;
14
            Unity メッセージ | 0 個の参照
15          void Update()
16          {
17              time -= Time.deltaTime;
18              if (time <= 0.0f)
19              {
20                  Generate();
21                  time = Random.Range(1.0f, 3.0f);// 1秒から3秒の間隔で再生成
22              }
23          }
```

酔歩

- 3次元で処理
  - 回転がしやすい
- あらぬ方向に行くと困るので、最大ステップ数を決める
- 向きをランダムにquaternionで回転
- Lerpを使って下方向に向かわせる

RandomWalkMonoBehaviourScript.cs

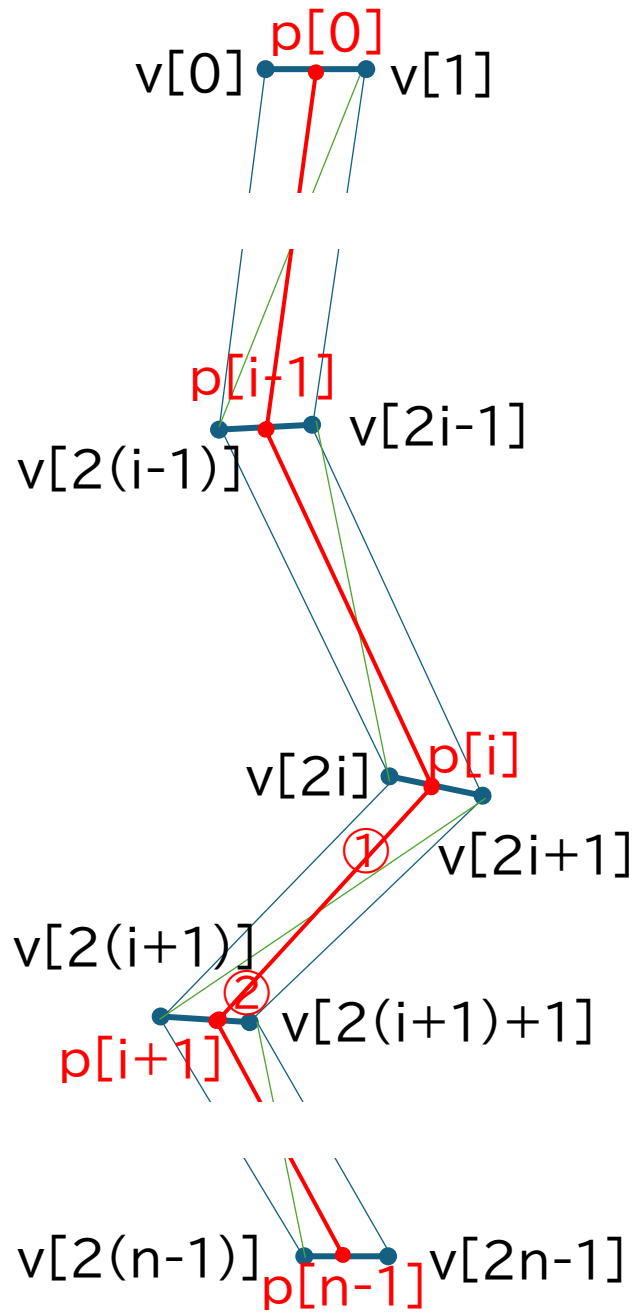
```
25      void Generate()
26      {
27          // 到達経路の生成
28          float HEIGHT = 1000.0f;
29          Vector3 pos = new Vector3(0.0f, HEIGHT, 0.0f);
30          Vector3 dir = Vector3.down;
31
32          List<Vector3> path = new List<Vector3>();
33          path.Add(pos);
34          for(int i = 0; i < 1000; i++)// 進む最大値に制限
35          {
36              float angle = Random.Range(-bendMax, +bendMax);// 曲がる角度
37              Quaternion q = Quaternion.AngleAxis(angle, Vector3.forward);
38              dir = q * dir;
39              dir = Vector3.Lerp(dir, Vector3.down, gravityRate);// 徐々に下方向に戻す
40
41              float distance = Random.Range(distanceMin, distanceMax);
42              pos += dir.normalized * distance;
43              path.Add(pos);
44
45              if(pos.y < 0.0f) break;// 地面に到達
46          }
47
48          // メッシュの構築
49          UpdateMesh(path);
50      }
```

RandomWalkMonoBehaviourScript.cs

メッシュの構築

- ほとんどの処理が先ほどと同じ
- 変更点
  端点の先にポリゴンを追加しない
  - 端点を上下に合わせて平らにする
  - 頂点数：経路の頂点の2倍
  - ポリゴン数：経路の頂点数-1の2倍

```
52      void UpdateMesh(List<Vector3> path)
53      {
54          int n = path.Count;
55
56          int vertexCount = 2 * n;
57          Vector3[] vertices = new Vector3[vertexCount];
58          float halfWidth = 3.0f;
59                                                  変更点(後述)
60          int vtx = 0;
61          // 始点
62          ...
68          for (int i = 1; i < n - 1; i++)...
78          // 終点
79          ...
85
86          Color[] colors = new Color[vertexCount];
87          for (int i = 0; i < vertexCount; i++)
88          {
89              colors[i] = Color.white;
90          }
91
92          int polygonCount = 2 * (n - 1);// 線分の数
93          int[] triangles = new int[3 * polygonCount];
94          int idx = 0;
95          vtx = 0;
96          while (idx < 3 * polygonCount)
97          {
98              triangles[idx + 0] = vtx + 0;
99              triangles[idx + 1] = vtx + 2;   ①
100             triangles[idx + 2] = vtx + 1;
101
102             triangles[idx + 3] = vtx + 1;
103             triangles[idx + 4] = vtx + 2;   ②
104             triangles[idx + 5] = vtx + 3;
105
106             idx += 6;
107             vtx += 2;
108         }
109
110         Mesh Mesh = meshFilter.mesh;
111
112         Mesh.Clear();
113         Mesh.vertices = vertices;
114         Mesh.colors = colors;
115         Mesh.triangles = triangles;
116
117         Mesh.RecalculateBounds();
118         Mesh.RecalculateNormals();
119     }
```



端点の処理

- 左右に一定の幅で広げる
  - 幅はプロパティにして変えられるようにしても良いですねね

RandomWalkMonoBehaviourScript.cs

```
54          int n = path.Count;
55
56          int vertexCount = 2 * n;
57          Vector3[] vertices = new Vector3[vertexCount];
58          float halfWidth = 3.0f;
59
60          int vtx = 0;
61          // 始点
62          {
63              Vector3 center = path[0];
64              vertices[vtx + 0] = center + Vector3.left * halfWidth;
65              vertices[vtx + 1] = center - Vector3.left * halfWidth;
66              vtx += 2;
67          }
68          for (int i = 1; i < n - 1; i++)...
78          // 終点
79          {
80              Vector3 center = path[n - 1];
81              vertices[vtx + 0] = center + Vector3.left * halfWidth;
82              vertices[vtx + 1] = center - Vector3.left * halfWidth;
83              vtx += 2;
84          }
```

途中のポリゴン

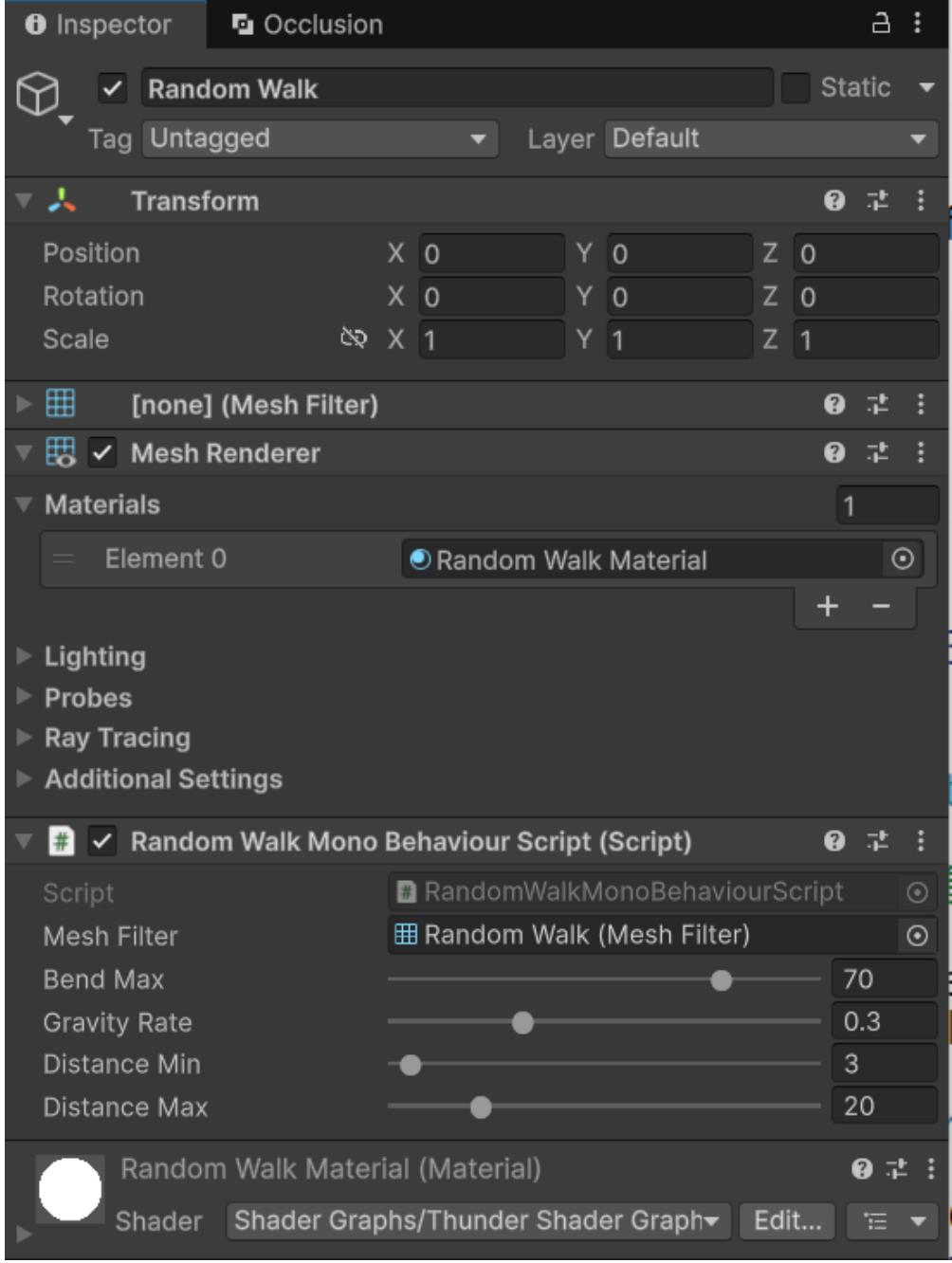
前後の位置の差から向きを定め、90度回転することで幅が広がる向きを決める

RandomWalkMonoBehaviourScript.cs

```
54          int n = path.Count;
55
56          int vertexCount = 2 * n;
57          Vector3[] vertices = new Vector3[vertexCount];
58          float halfWidth = 3.0f;
59
60          int vtx = 0;
61          // 始点
62          ...
68          for (int i = 1; i < n - 1; i++)
69          {
70              Vector3 center = path[i];
71              // 幅方向のベクトルを計算: 端点以外は前後の差分で方向を決定
72              Vector3 right = (path[i + 1] - path[i - 1]).normalized;
73              right = new Vector3(-right.y, right.x, right.z);// 90度回転
74              vertices[vtx + 0] = center + right * halfWidth;
75              vertices[vtx + 1] = center - right * halfWidth;
76              vtx += 2;
77          }
```

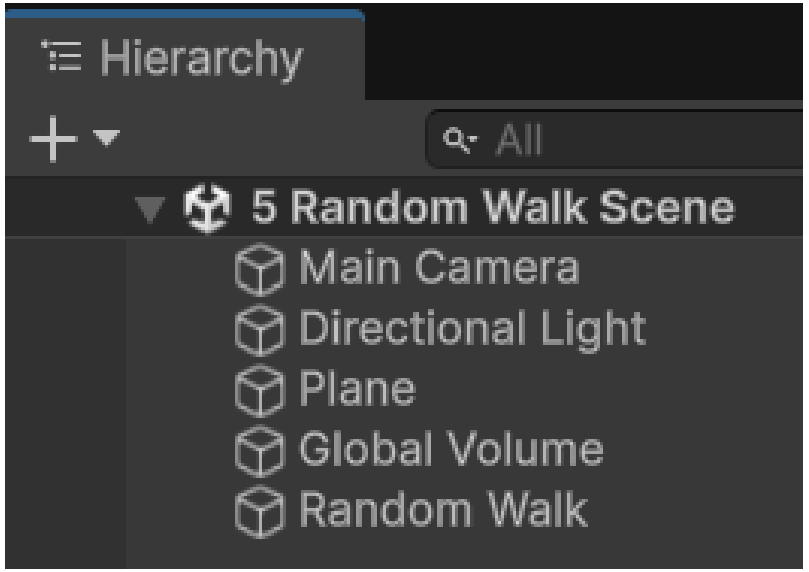
ゲームオブジェクト

- Materialの作成
  - Shader Graphを設定
    - 4 Thunder/Thunder Shader Graph
- 空のGameObjectを作成
  - 名称例:Random Walk
  - Mesh Filter を追加
  - Mesh Renderer を追加
    - Materialを追加
  - RandomWalkMonoBehaviourScriptを追加
    - パラメータをいい感じに設定



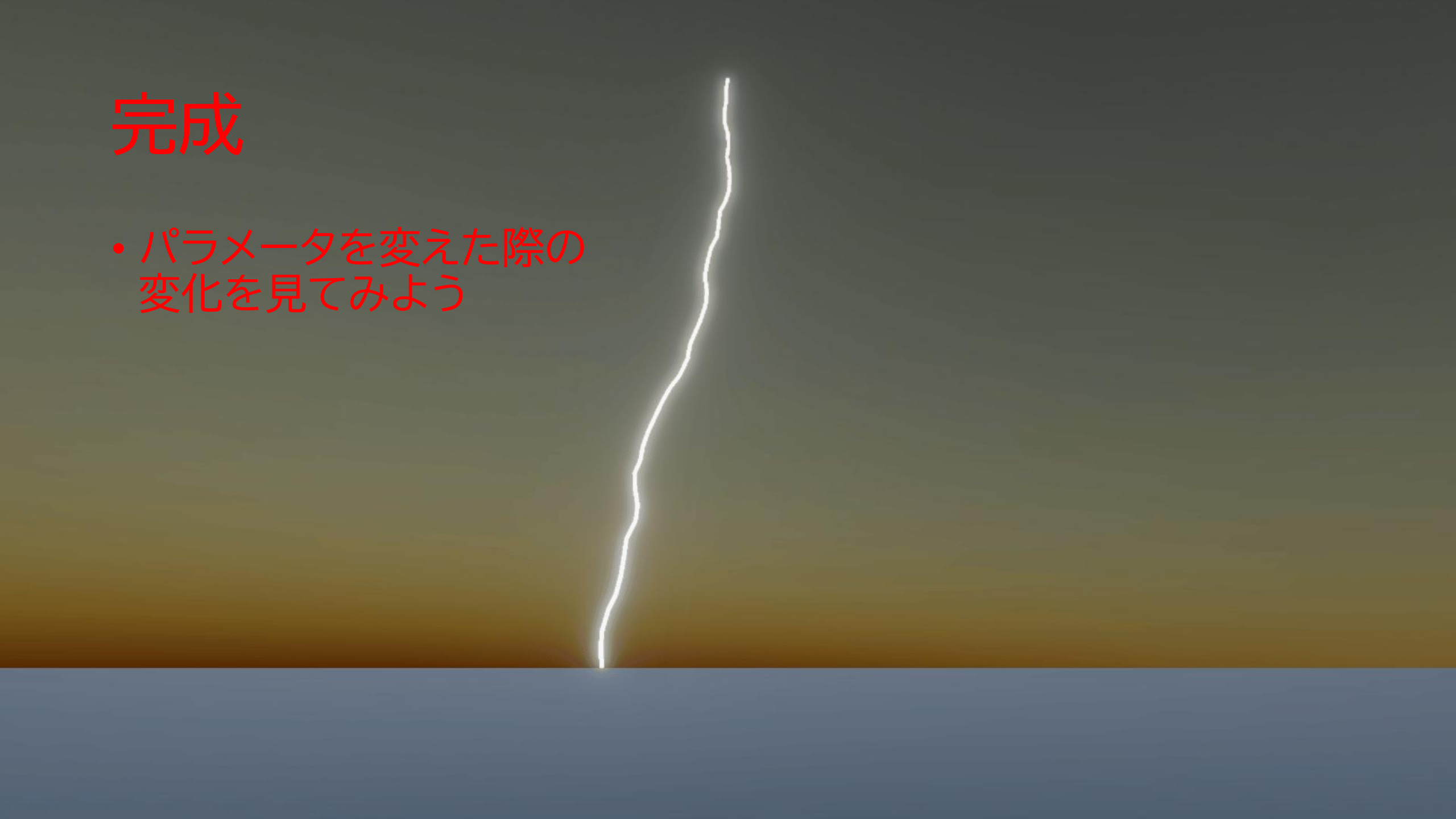
その他

- 先ほどと同じようにオブジェクトを追加・設定する
  - 床
  - ブルーム(グローバルボリューム)
  - カメラ
  - 平行光源(Directional Light)



完成

- パラメータを変えた際の変化を見てみよう



まとめ

- CPUでのリソース生成
  - テクスチャへの描画
    - テクスチャに簡単な値(テクスチャ座標)を書き込む
  - リヒテンベルク図形
    - テクスチャに書き込む応用として雷の模様を作成する
  - ポリゴンの描画
    - ポリゴンを動的に生成する
  - 雷
    - ポリゴンの動的な作成の応用として雷の模様を作成する
      - リヒテンベルク図形を基に
      - ランダムウォーク