

コンピュータシエーダ

2025年度 プログラムワークショップⅣ (13)

今回のリポジトリ

- https://github.com/tpu-game-2025/PGWS4_13_compute_shader

The screenshot shows the GitHub interface for the repository `tpu-game-2025 / PGWS4_13_compute_shader`. The repository is private and has a recent push to the `develop` branch 10 seconds ago. The file list shows a `src` directory and several files: `README.md`, `Result.gif`, `Result1.png`, and `Result2.gif`, all initialized 1 minute ago. The `README` file is selected, showing the title `コンピュータシェーダー` and the section `はじめに` (Introduction). The introduction text states that the repository is for managing the program workshop IV and that the solution is in the `develop` branch. The section `結果画像` (Result Images) is also visible, with a sub-section `コンピュータシェーダー入門` (Introduction to Computer Shaders) and a large blue image placeholder below it.

tpu-game-2025 / PGWS4_13_compute_shader

Code Issues Pull requests Actions Projects Security Insights Settings

PGWS4 13 compute shader Private

develop had recent pushes 10 seconds ago Compare & pull request

main 2 Branches 0 Tags Go to file + Code

imagire initialize a5bd035 · 1 minute ago 2 Commits

src	initialize	1 minute ago
README.md	initialize	1 minute ago
Result.gif	initialize	1 minute ago
Result1.png	initialize	1 minute ago
Result2.gif	initialize	1 minute ago

README

コンピュータシェーダー

はじめに

プログラムワークショップIVの管理用です。

解答はdevelopブランチを見てください。

結果画像

コンピュータシェーダー入門

もくじ

- コンピュートシェーダ概要
- 簡単なコンピュートシェーダ
- GPUパーティクル

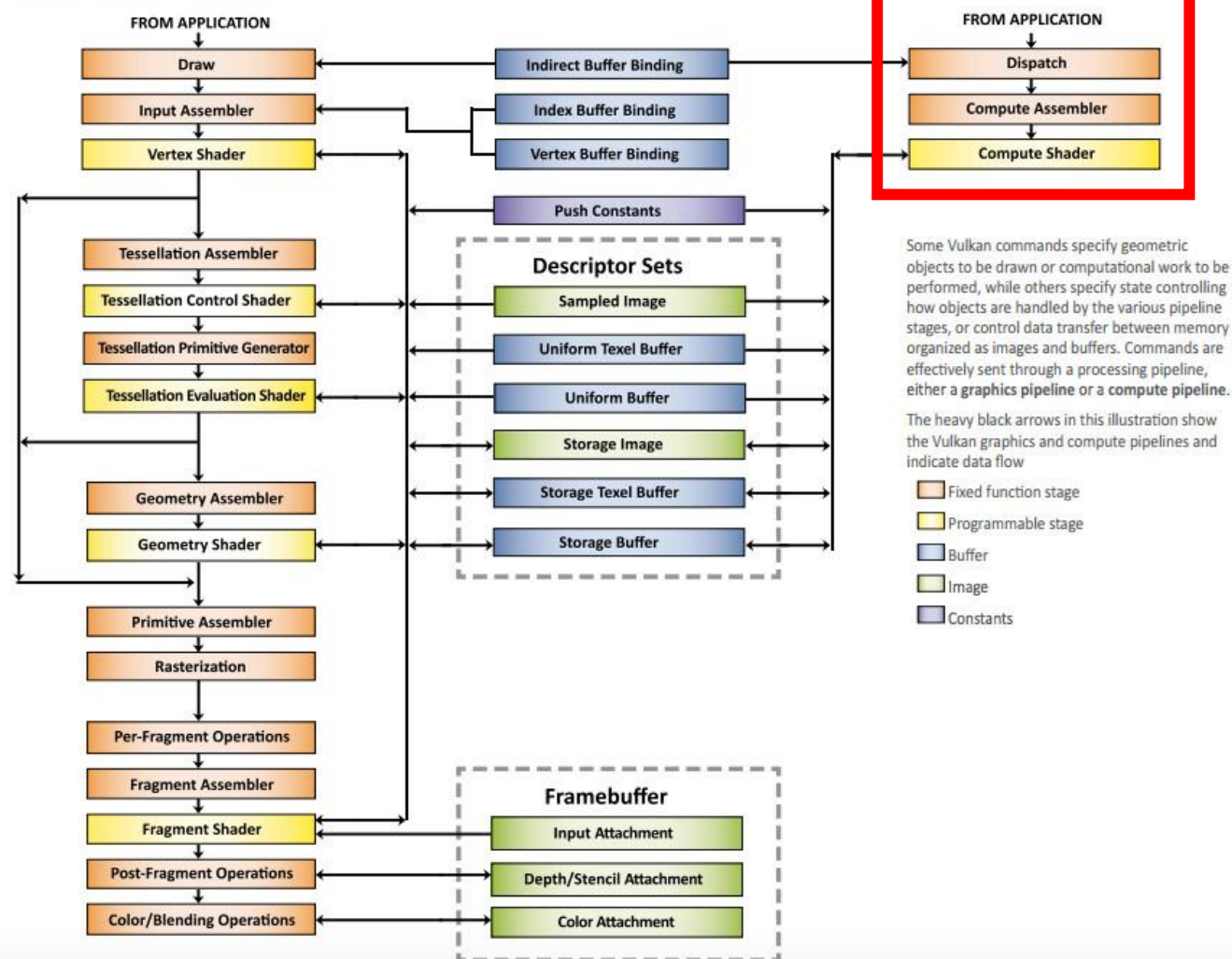
もくじ

- コンピュートシェーダ概要
- 簡単なコンピュートシェーダ
- GPUパーティクル

コンピュータシェーダ

- 現在、自由度が高い処理に用いられているシェーダ
- 描画目的以外にもGPUの並列性を使えるようにしよう

Vulkan Pipeline Diagram [9]



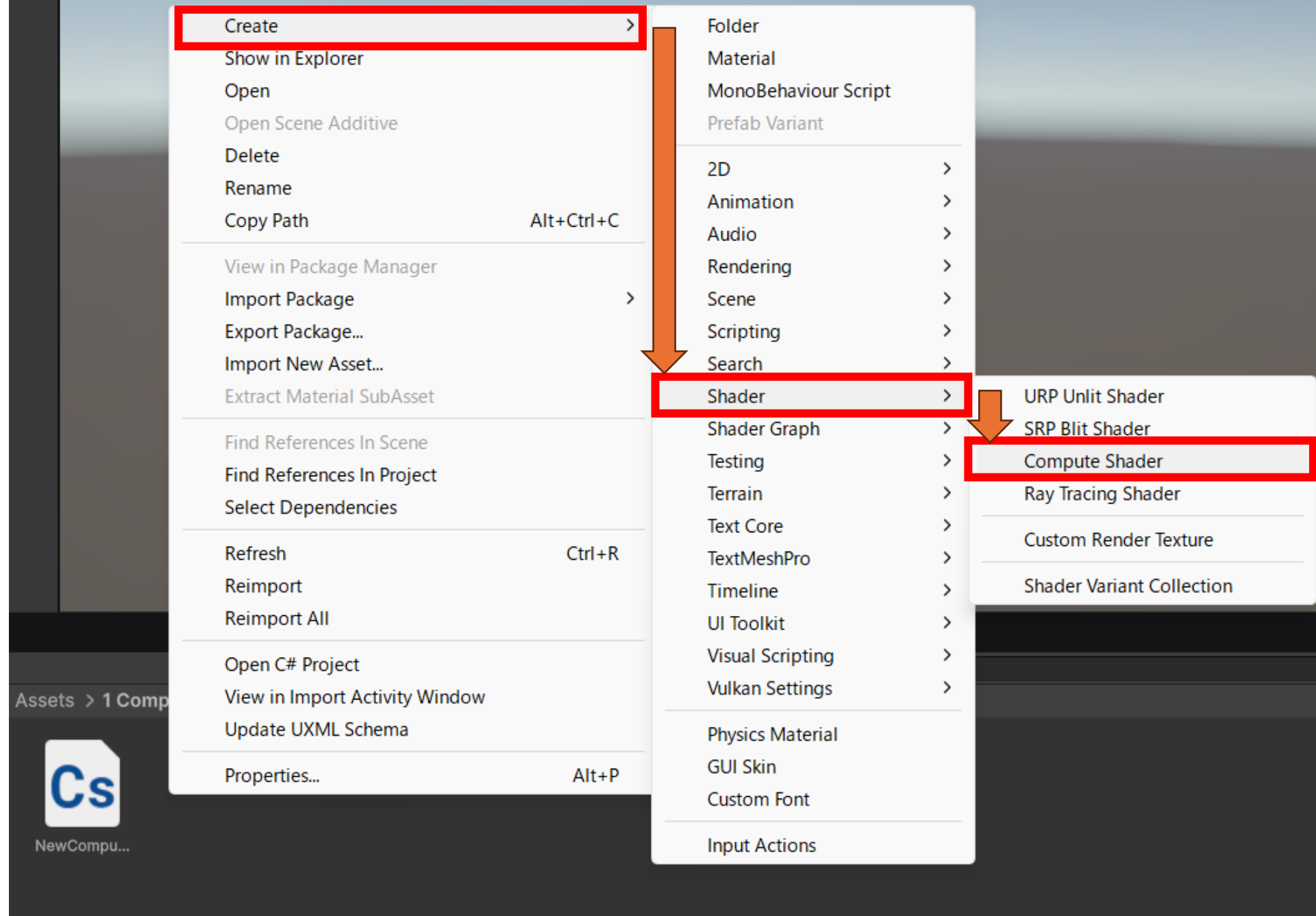
GPUに向けた計算

- 同じプログラムをたくさんのデータに対して適応する
 - 数千から数万の計算コアを有するので、何万以上の個数が向く
 - なるべく独立して、他との依存性がない計算が良い



使い道

- GPUパーティクル
 - 細かなパーティクル
 - 破片の物理演算
- Cluster Shading
 - 空間をブロックに分け、ブロックごとに描画するオブジェクト・ライトを洗い出して、大量のライトのシーンを高速に描画する



自動で追加されるソースコードの中身

```
1 // それぞれの#kernelは、どの関数をコンパイルするかを指示します; 多くのカーネルを持てます
2 // Each #kernel tells which function to compile; you can have many kernels
3 #pragma kernel CSMain
4 // enableRandomWriteフラグを持つRenderTextureを作成し、cs.SetTextureで設定せよ
5 // Create a RenderTexture with enableRandomWrite flag and set it
6 // with cs.SetTexture
7 RWTexture2D<float4> Result; 4成分floatの読み書きできるテクスチャ
8 [numthreads(8, 8, 1)] 8x8x1の単位で実行数(64単位で実行される)
9 戻り値なし void CSMain (uint3 id : SV_DispatchThreadID) 全スレッドを通してユニークなIDを受けとる
10 { // やること:実際のコードをここに挿入せよ!
11   // TODO: insert actual code here!
12   Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0.0);
13   // ビット計算例 X軸で16個の周期計算 Y軸で16個の周期計算
14 }
```

てきとうな計算

簡単なコンピュートシェーダ

#pragma kernel 関数名

RWTexture3D<float4> tex;

[numthreads(2, 8, 4)]

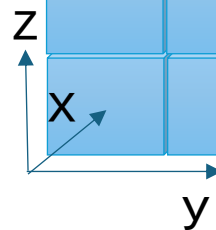
これらの積が32や64の倍数でないと
GPUで暇になるコアが発生する

void 関数名(uint3 id : SV_DispatchThreadID)

{

tex[id] = float4(0, 0, 0, 1);

}

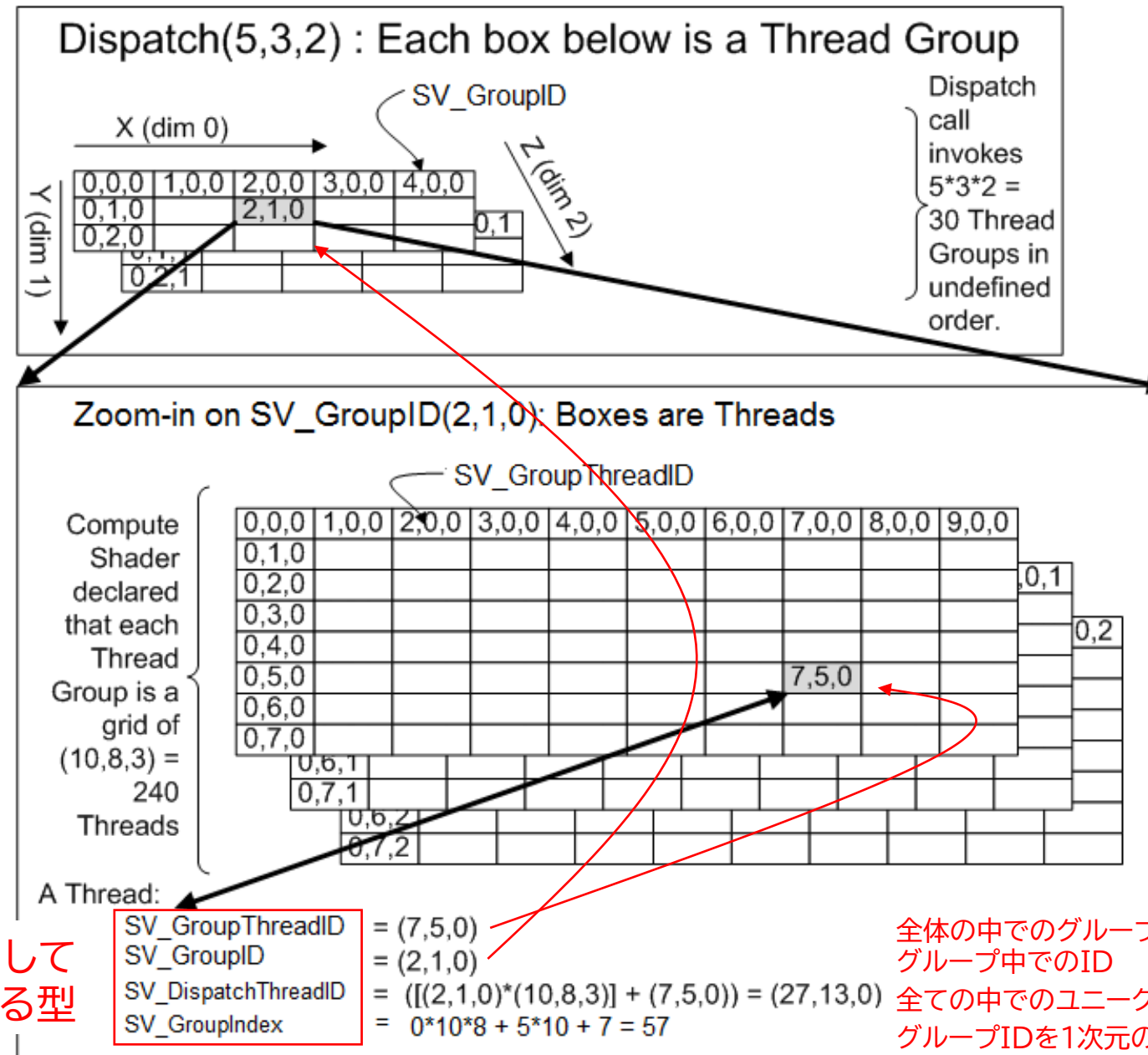


C#側

int kernel = computeshader.FindKernel("関数名");

computeshader.Dispatch(kernel, 3, 5, 7); ➡ (3*2, 5*8, 7*4) = (6, 40, 28) = 6720個の処理が走る

3次元配列の入れ子構造

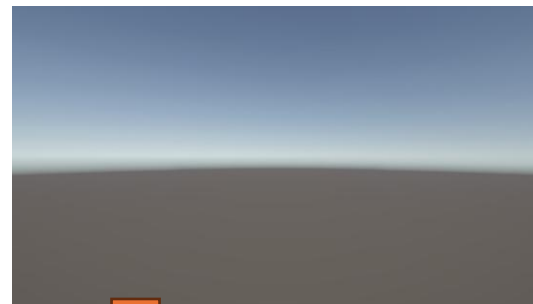


グローバル変数に使えるデータ型

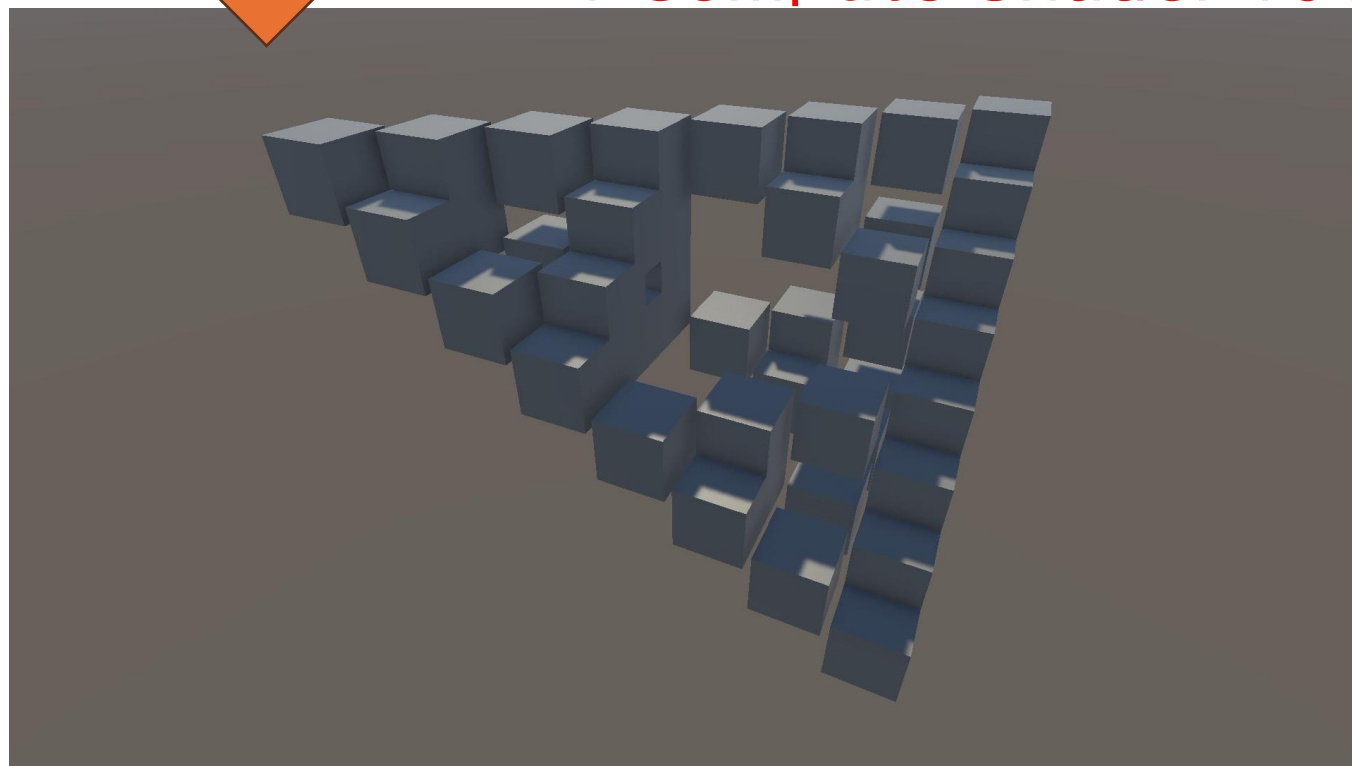
- 読み書き可能(非圧縮、遅い)
 - RWByteAddressBuffer:配列
 - RWStructuredBuffer:構造体配列
 - RWTexture1D
 - RWTexture1DArray:配列
 - RWTexture2D
 - RWTexture2DArray
 - RWTexture3D
- 読み込み専用(圧縮されているかも?速い)
 - StructuredBuffer
 - Texture1D
 - Texture1DArray
 - Texture2D
 - Texture2DArray
 - Texture3D
 - Texture2DMS:マルチサンプリング
 - Texture2DMSArray
 - TextureCube:キューブマップ
 - TextureCubeArray

本日の内容

- コンピュートシェーダ概要
- 簡単なコンピュートシェーダ
 - GPUの計算結果をCPUで読み込んで、オブジェクトを配置してみる
- GPUパーティクル



シーン: 1 Compute Shader 101



最低限のコード変更

- RWTexture2DをCPUで取得するのが面倒なので型を変更

```
1 // Each #kernel tells which function to compile; you can have many kernels
2 #pragma kernel CSMain
3
4 // Create a RenderTexture with enableRandomWrite flag and set it
5 // with cs.SetTexture
6 // ↓ RWTexture2D<float4> Result;
7 RWStructuredBuffer<float4> Result;
8
9 [numthreads(8, 8, 1)]
10 void CSMain (uint3 id : SV_DispatchThreadID)
11 {
12     // TODO: insert actual code here!
13
14     // ↓ Result[id.xy] = float4(id.x & id.y, (id.x & 15) / 15.0, (id.y & 15) / 15.0, 0.0);
15     Result[id.y*8+id.x] = float4(id.x & id.y, (id.x & 15) / 15.0, (id.y & 15) / 15.0, 0.0);
16 } インデックスを1次元に変更
```

オブジェクト 配置

ComputeShader
MonoBehaviour
Scriptを追加して編集

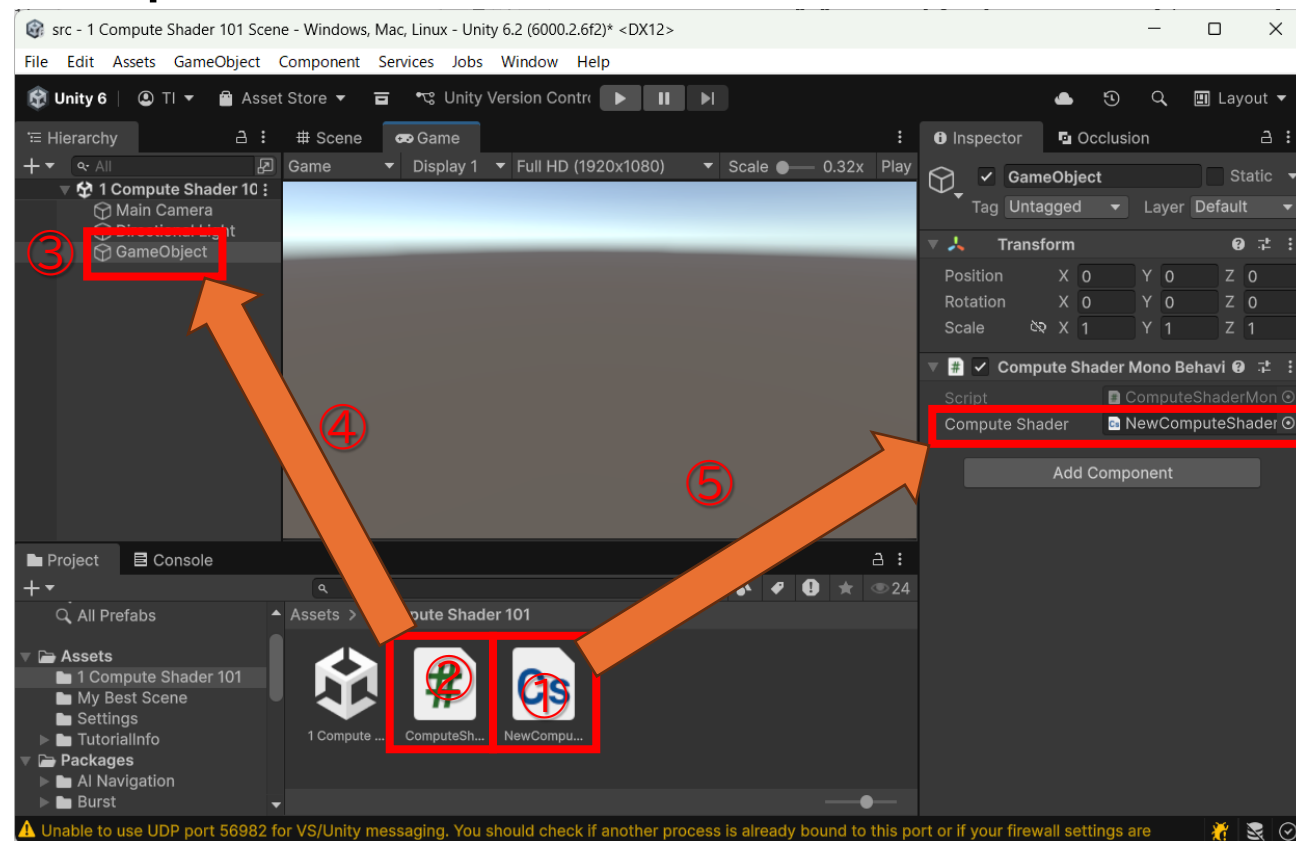
- 今回は、8x8個実行
- ComputeBuffer
 - RWStructuredBuffer(シェーダ側)のコンテナ
- Dispatch:実行
- GetData
 - CPUが読めるメモリにコピー
- CreatePrimitive
 - インスタンス生成

```
1 using UnityEngine;
2
3 Unity スクリプト (1 件のアセット参照) 0 個の参照
4 public class ComputeShaderMonoBehaviourScript : MonoBehaviour
5 {
6     [SerializeField] ComputeShader computeShader = default!;
7
8     // Start is called once before the first execution of Update after the MonoBehaviour is created
9     Unity メッセージ 0 個の参照
10    void Start()
11    {
12        int x = 8;
13        int y = 8;
14        ComputeBuffer computeBuffer = new ComputeBuffer(4 * x * y, sizeof(float));
15
16        int kernelHandle = computeShader.FindKernel("CSMain"); // カーネル関数のハンドルを取得
17        computeShader.SetBuffer(kernelHandle, "Result", computeBuffer); // バッファをセット
18        computeShader.Dispatch(kernelHandle, x / 8, y / 8, 1); // カーネル関数を実行
19
20        float[] result = new float[4 * x * y];
21        computeBuffer.GetData(result); // CPUからGPUへデータを転送
22        computeBuffer.Release(); // バッファの解放
23
24        // 確認用に立方体を表示してみる
25        for (int i = 0; i < x; i++)
26        {
27            for (int j = 0; j < y; j++)
28            {
29                float cs_x = result[4 * (i + j * x) + 0];
30                float cs_y = result[4 * (i + j * x) + 1] * 10.0f;
31                float cs_z = result[4 * (i + j * x) + 2] * 10.0f;
32                GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
33                cube.transform.position = new Vector3(cs_x, cs_y, cs_z);
34                cube.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f); // 隙間をあける
35            }
36        }
37    }
38 }
```

やってみよう：

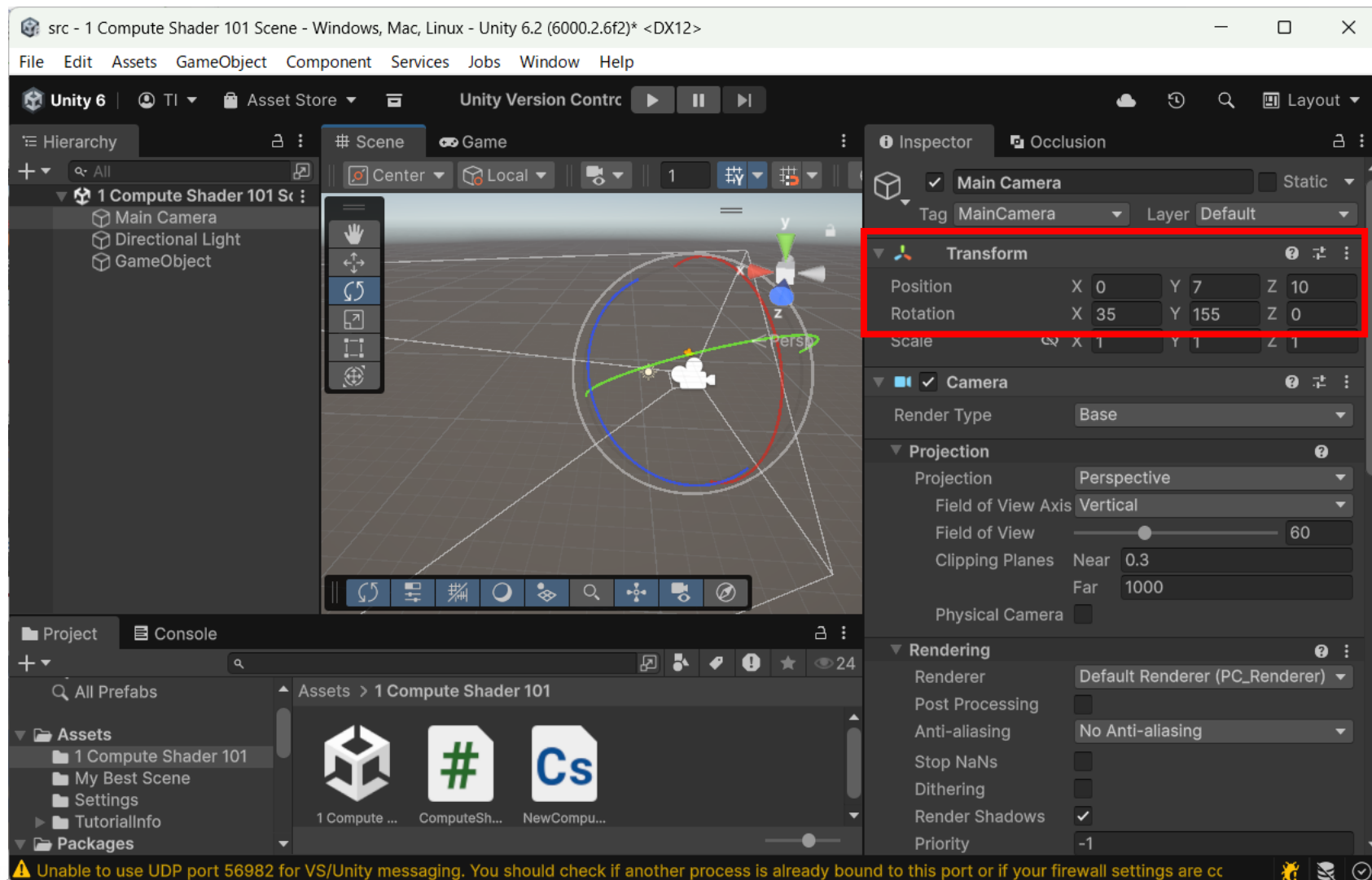
1 Compute Shader 101/1 Compute Shader 101 Scene

1. Compute Shader を追加
 - 名称例: NewComputeShader
2. C# Scriptを追加
 - 名称例: ComputeShaderMonoBehaviourScript
 - C# Scriptを書き換え
 - Compute Shader を書き換え
3. EmptyなGameObjectを追加
 - 名称例: GameObject
4. GameObject にCompute ShaderMonoBehaviour Scriptをバインド
5. ComputeShaderMonoBehaviourScriptに NewComputeShaderを設定

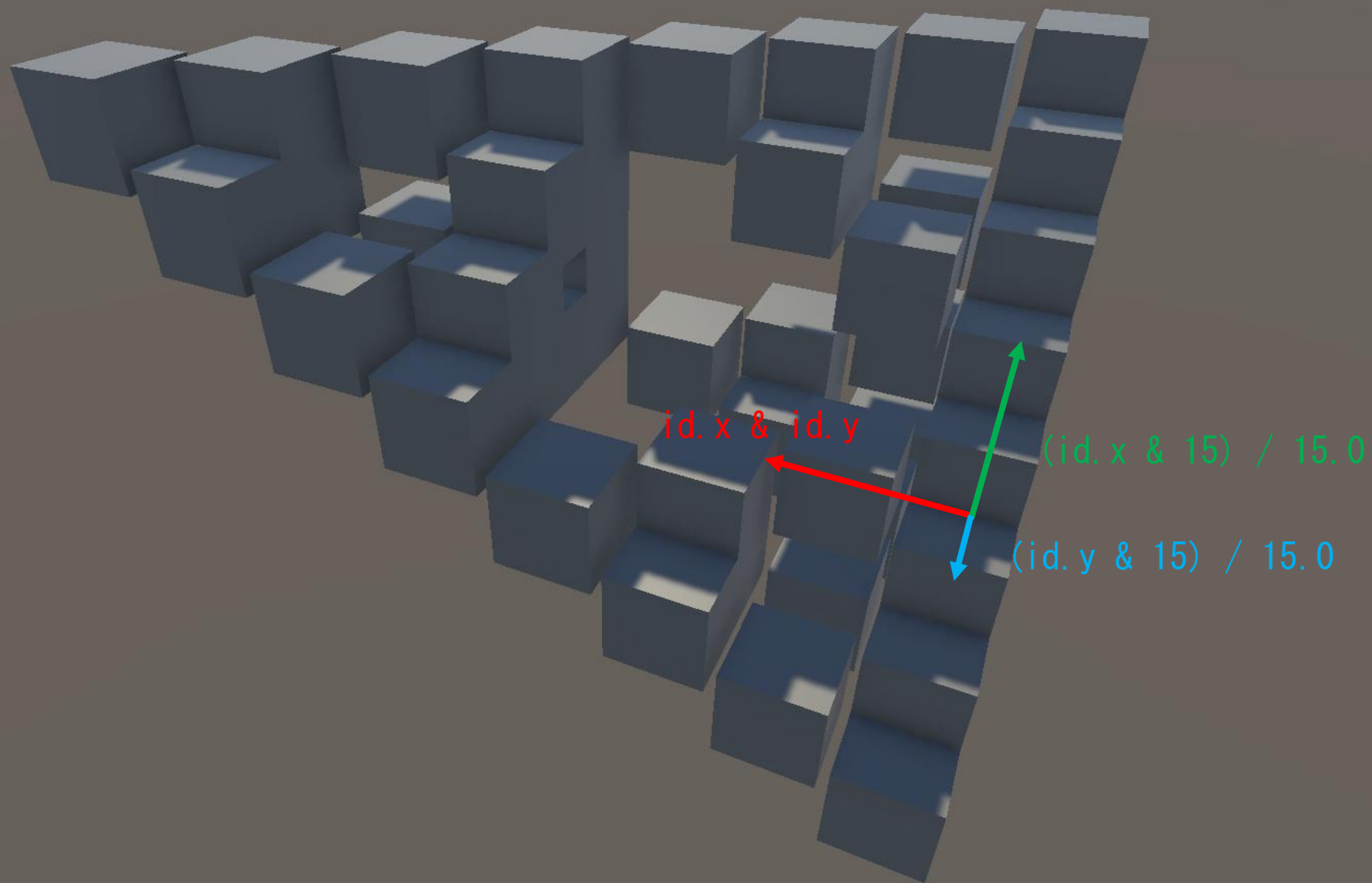


カメラを移動

- 何となく見やすい位置に

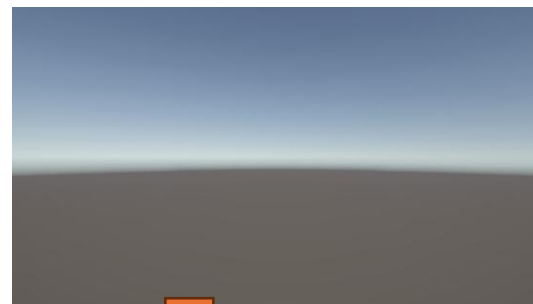


完成

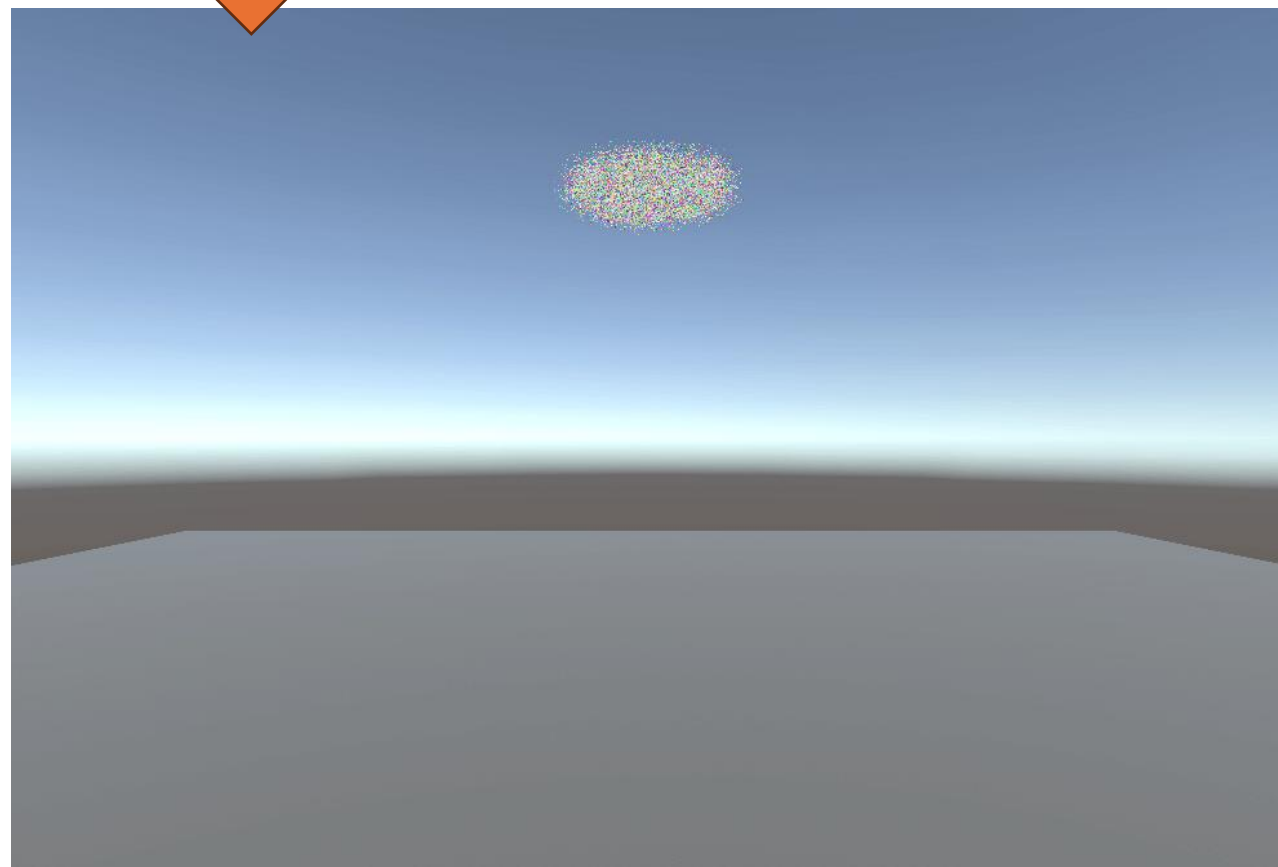


もくじ

- コンピュートシェーダ概要
- 簡単なコンピュートシェーダ
- GPUパーティクル

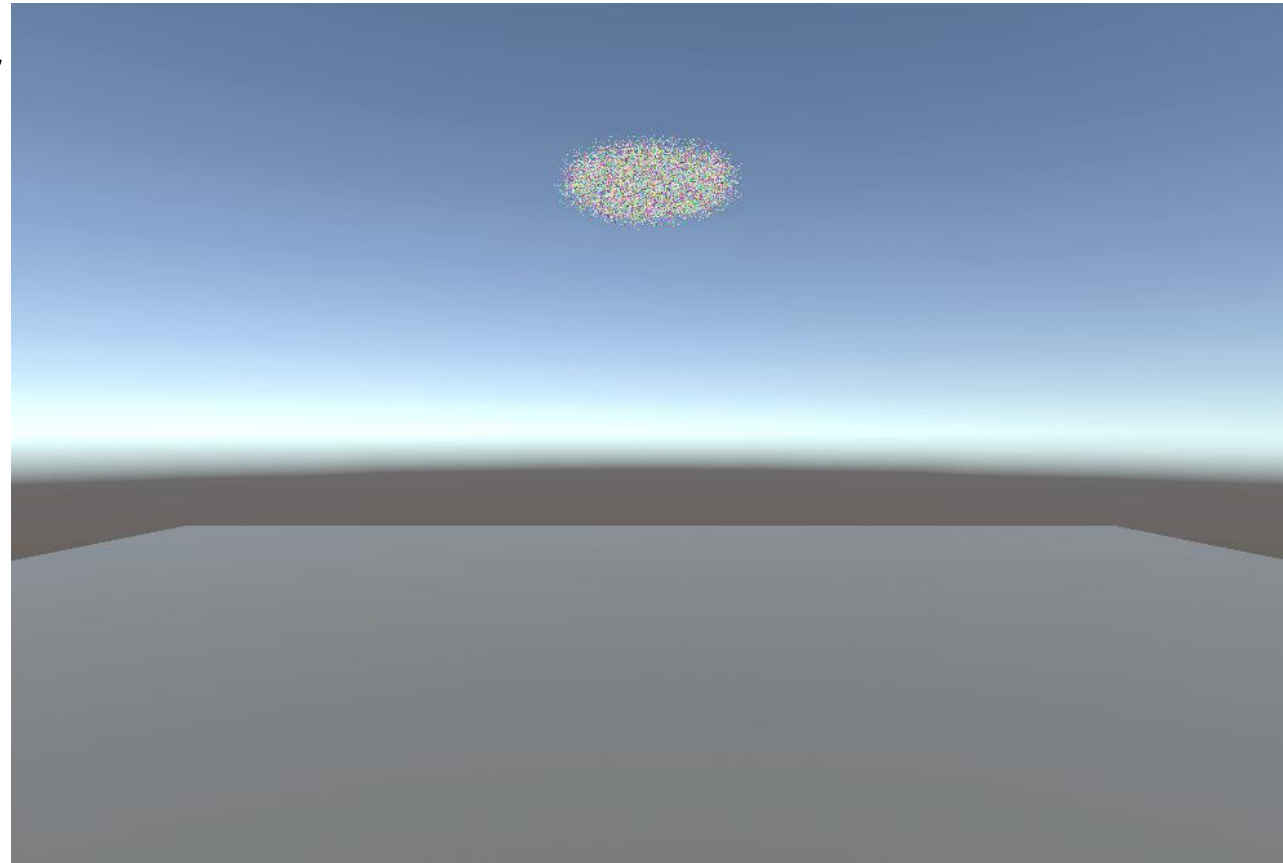


シーン: 2 GPU Particle Scene



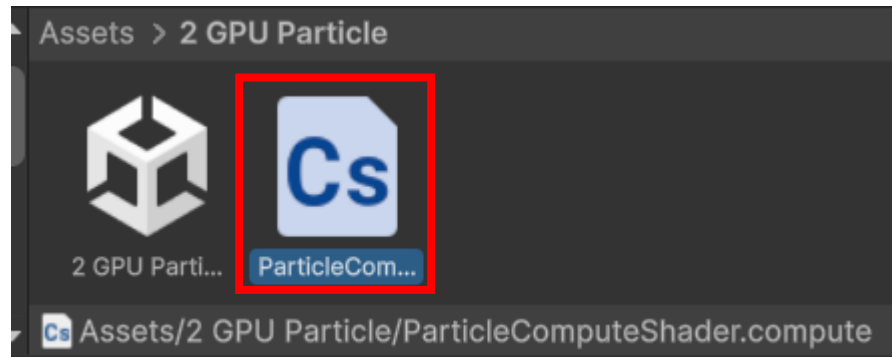
GPUパーティクルを試す

- 地面についたら反射
- 動きが止まったら吹き出し直し
- 噴水のように動く
- 色はランダム



コンピュートシェーダの追加

- 名称例: ParticleComputeShader



乱数生成

ParticleComputeShader.compute

```
15 // 乱数関数(see. https://andantesoft.hatenablog.com/entry/2024/12/19/193517)
16 float fihash_orig(float2 v)
17 {
18     uint2 u = asint(v * float2(141421356, 2718281828));
19     return float((u.x ^ u.y) * 3141592653) * 2.3283064365386962890625e-10;
20 }
21
22 float2 rand2(float2 st)
23 {
24     return float2(fihash_orig(st), fihash_orig(st + 1));
25 }
26
27 float3 rand3(float2 st)
28 {
29     return float3(fihash_orig(st), fihash_orig(st + 1), fihash_orig(st + 2));
30 }
```

構造体とオブジェクト

- 構造体配列を定義

ParticleComputeShader.compute

```
1 // Each #kernel tells which function to compile; you can have many kernels
2 #pragma kernel CSInitialize
3 #pragma kernel CSUpdate
```

関数の宣言

```
4
5 struct Particle
6 {
7     float3 position;
8     float3 velocity;
9     float3 color;
10 };
11
```

一つ一つの粒子のデータ

```
12 RWStructuredBuffer<Particle> Particles;
```

構造体配列

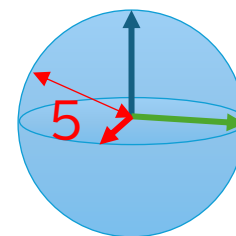
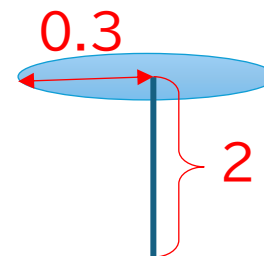
```
13 float deltaTime; // 更新時にCPUから受け取る前フレームからの経過時間
```

初期化

```
32 void reset(uint id)
33 {
34     float2 seed = float2(
35         (1.0 / 256.0) * (float) (id % 256),
36         (1.0 / 256.0) * (float) ((id / 256) % 256));
37
38     // 高さ2の半径0.3の円盤上に生成
39     float2 r2 = rand2(seed);
40     Particles[id].position = float3(
41         cos(r2.x * 2.0 * 3.14159265) * r2.y * 0.3,
42         2.0,
43         sin(r2.x * 2.0 * 3.14159265) * r2.y * 0.3
44     );
45
46     // 速度は、球的に広げる
47     float3 r3 = rand3(seed + 10);
48     Particles[id].velocity = float3(
49         5 * (r3.x + 0.01) * sin(r3.y * 3.14159265) * cos(r3.z * 2.0 * 3.14159265),
50         5 * (r3.x + 0.01) * sin(r3.y * 3.14159265) * sin(r3.z * 2.0 * 3.14159265),
51         5 * (r3.x + 0.01) * cos(r3.y * 3.14159265)
52     );
53
54     // 色はランダム
55     Particles[id].color = rand3(seed + 20);
56 }
57
58 #define THREAD_NUM 64
59
60 [numthreads(THREAD_NUM, 1, 1)]
61 void CSInitialize(uint3 id : SV_DispatchThreadID)
62 {
63     reset(id.x);
64 }
```

ParticleComputeShader.compute

IDごとに異なる2次元の値を生成



運動

- 速度から位置を更新
- 加速度(重力)で速度を更新
 - 空気抵抗で少しずつ遅くする
- 当たり判定
 - 床より下に来たら上向きにする
 - 跳ね返り係数を付けて勢いを落とす
- 終了判定
 - 速度が一定以下になったら再初期化

```
66 [numthreads(THREAD_NUM, 1, 1)]
67 void CSUpdate(uint3 id : SV_DispatchThreadID)
68 {
69     float3 position = Particles[id.x].position;
70     float3 velocity = Particles[id.x].velocity;
71
72     // 動かなくなったらリセット
73     if (dot(velocity, velocity) < 0.001)
74     {
75         reset(id.x);
76         return;
77     }
78
79     // 下に落ちたら跳ね返る
80     if (position.y < 0.0)
81     {
82         velocity.y = -0.6 * velocity.y;
83     }
84
85     // 陽オイラー
86     position += velocity * deltaTime;
87     velocity.y -= 9.8 * deltaTime;
88     velocity *= 0.995; // 簡易空気抵抗
89
90     // 値の格納
91     Particles[id.x].position = position;
92     Particles[id.x].velocity = velocity;
93 }
```

描画用のアセットを追加

1. マテリアル

- 名称例: 2 Particle Material

2. シェーダ

- 「Shader」-「URP Unlit Shader」
- 2 Particle Materialに設定
- 名称例: ParticleUnlitUniversalRenderPipelineShader



シェーダ

- URP向けのシェーダとして、ある程度自動的に記述される

ParticleUnlitUniversalRenderPipelineShader.shader

```
1 Shader "Custom/ParticleUnlitUniversalRenderPipelineShader"
2 {
3     SubShader
4     {
5         Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }
6
7         Pass
8         {
9             HLSLPROGRAM
10
11             #pragma vertex vert
12             #pragma fragment frag
13
14             #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"
```

パーティクルの描画

- Id値から構造体配列を読み込み位置や色として使用する

```
16 struct Particle ParticleUnlitUniversalRenderPipelineShader.shader
17 {
18     float3 Position;
19     float3 Velocity;
20     float3 Color;
21 };
22
23 struct Varyings
24 {
25     float4 positionHCS : SV_POSITION;
26     half3 color : COLOR0;
27 };
28
29 uniform StructuredBuffer<Particle> Particles;
30
31 Varyings vert(uint id : SV_VertexID) IDを入力として受け取ることにする
32 {
33     Varyings OUT;
34
35     Particle particle = Particles[id];
36     OUT.positionHCS = TransformObjectToHClip(particle.Position);
37
38     OUT.color = particle.Color;
39     return OUT;
40 }
41
42 half4 frag(Varyings IN) : SV_Target
43 {
44     return half4(IN.color, 1);
45 }
46 ENDHLSL
```

Compute shaderと
同じ構造体を定義する

構造体配列へのアクセス

CPU側の処理

- MonoBehaviourスクリプトの追加
 - 名称例: ParticleMonoBehaviourScript



```

1 using UnityEngine;
2 using System.Runtime.InteropServices;
3

```

1 個の参照

```

4 struct Particle
5 {
6
7
8
9
10

```

```

6     public Vector3 Position;
7     public Vector3 Velocity;
8     public Vector3 Color;
9 }

```

Compute Shaderと同じメモリレイアウトの構造体
(型名は異なるが実質的に同じ型)

☞ Unity スクリプト (1 件のアセット参照) | 0 個の参照

```

11 public class ParticleMonoBehaviourScript : MonoBehaviour
12 {
13
14
15
16
17
18
19
20

```

```

13     [SerializeField] Material material = default!;
14     [SerializeField] ComputeShader computeShader = default!;
15
16     private int updateKernel;
17     private ComputeBuffer buffer;
18
19     private const int THREAD_NUM = 64;
20     private const int PARTICLE_NUM = ((65536 + THREAD_NUM - 1) / THREAD_NUM) * THREAD_NUM;

```

表示するシェーダへのアクセス

[SerializeField] ComputeShader computeShader = default!; コンピュートシェーダへのアクセス

private int updateKernel; コンピュートシェーダの関数を呼び出すための対応付け

private ComputeBuffer buffer; 構造体配列の実体を保持

private const int THREAD_NUM = 64; // 1つのグループあたりのスレッド数

private const int PARTICLE_NUM = ((65536 + THREAD_NUM - 1) / THREAD_NUM) * THREAD_NUM;

粒子数(THREAD_NUMの倍数に増やす調整)

```

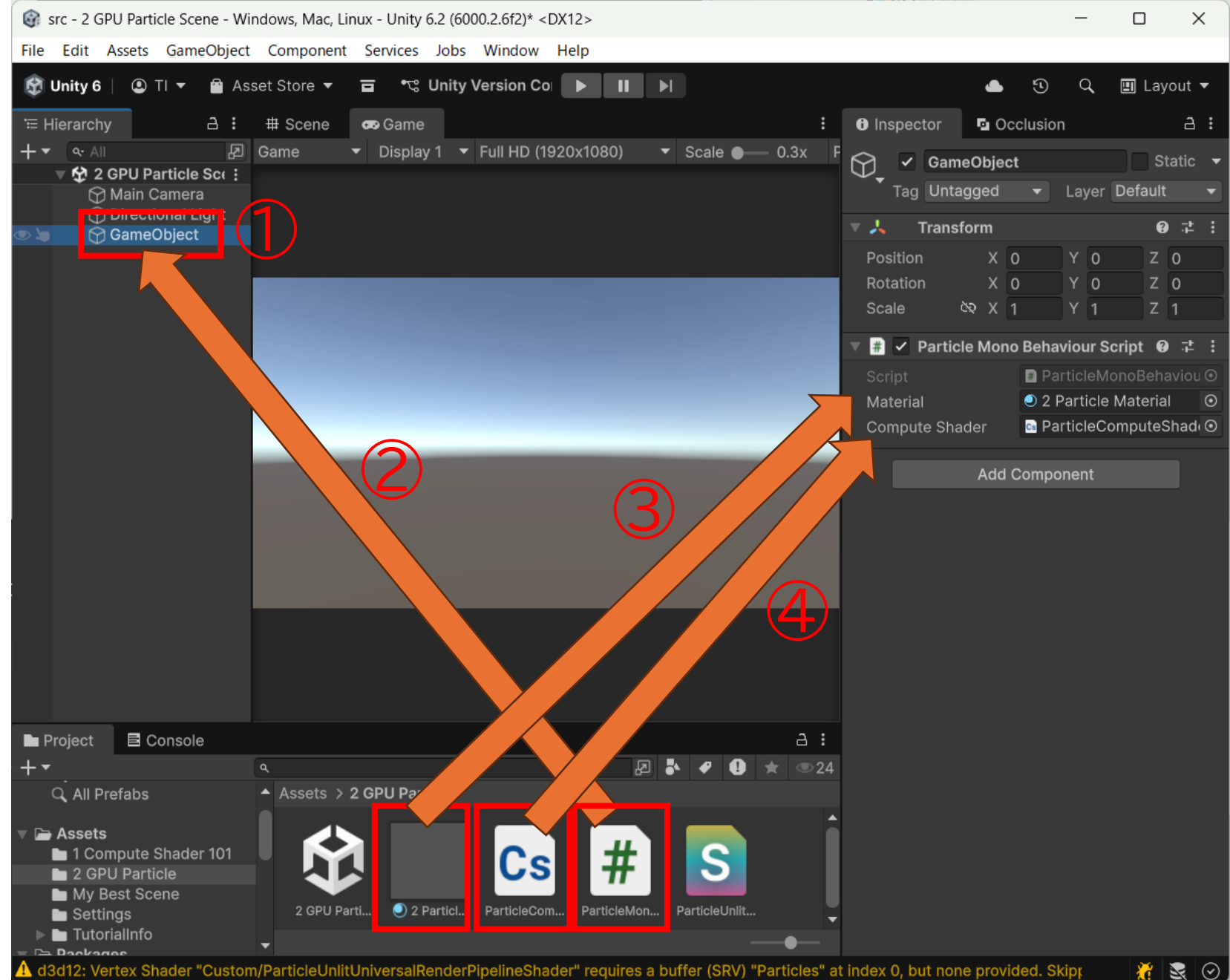
22 // オブジェクトが有効になった際に呼ばれる ParticleMonoBehaviourScript.cs
   ☐ Unity メッセージ 0 個の参照
23 private void OnEnable()
24 {
25     // パーティクルの情報を格納するバッファ
26     buffer = new ComputeBuffer(
27         PARTICLE_NUM,
28         Marshal.SizeOf(typeof(Particle)),
29         ComputeBufferType.Default);
30
31     // 初期化 文字列で関数を検索
32     int initKernel = computeShader.FindKernel("CSInitialize");
33     computeShader.SetBuffer(initKernel, "Particles", buffer);
34     初期化関数の実行 computeShader.Dispatch(initKernel, PARTICLE_NUM / THREAD_NUM, 1, 1);
35
36     // 更新後の設定
37     updateKernel = computeShader.FindKernel("CSUpdate");
38     関数ごとにバッファを  
設定する必要あり computeShader.SetBuffer(updateKernel, "Particles", buffer);
39
40     // 描画用のマテリアルの設定
41     material.SetBuffer("Particles", buffer);
42 }

```

```
44 // プロジェクトが無効になった際に呼ばれる
   ☞ Unity メッセージ 0 個の参照
45 private void OnDisable()    OnEnableの対となる解放処理
46 {
47     buffer.Release();
48 }
49
50 // 動かす
   ☞ Unity メッセージ 0 個の参照
51 void Update()
52 {
53     computeShader.SetFloat("deltaTime", Time.deltaTime);
54     computeShader.Dispatch(updateKernel, PARTICLE_NUM / THREAD_NUM, 1, 1);
55     更新関数の実行
56 }
57
   ☞ Unity メッセージ 0 個の参照
58 void OnRenderObject()
59 {
60     material.SetPass(0);
61     描画処理は直接呼び出す Graphics.DrawProceduralNow(MeshTopology.Points, PARTICLE_NUM);
62 }
```

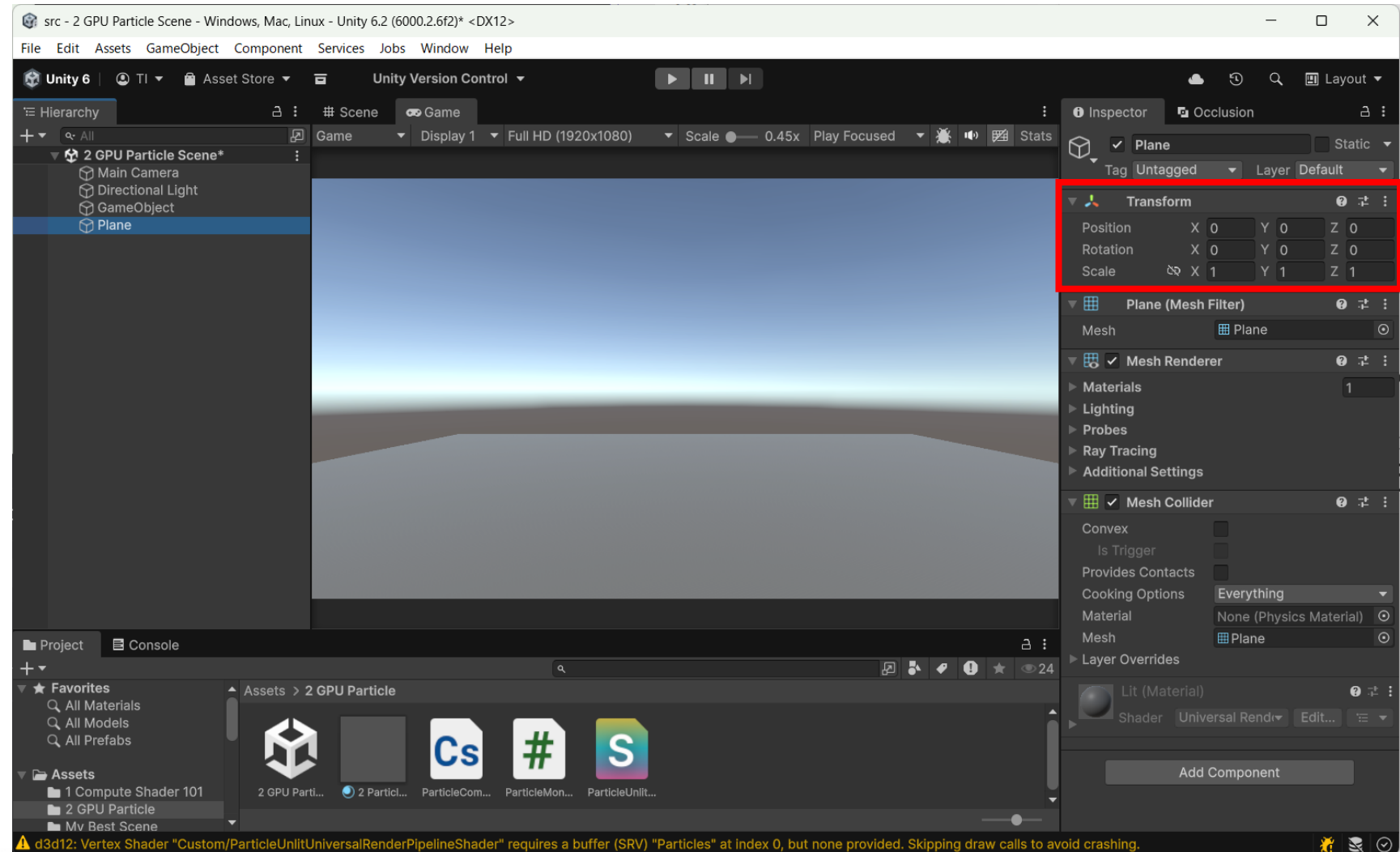
オブジェクト の追加

1. 「Create Empty」
で空のオブジェクトを
生成
 - 名称例:GameObject
2. ParticleMonoBehaviourScriptをオ
ブジェクトに追加
 - ParticleMonoBehaviourScriptのプロパ
ティを設定
3. Material: 2
Particle Material
4. Compute
Shader:ParticleC
omputeShader



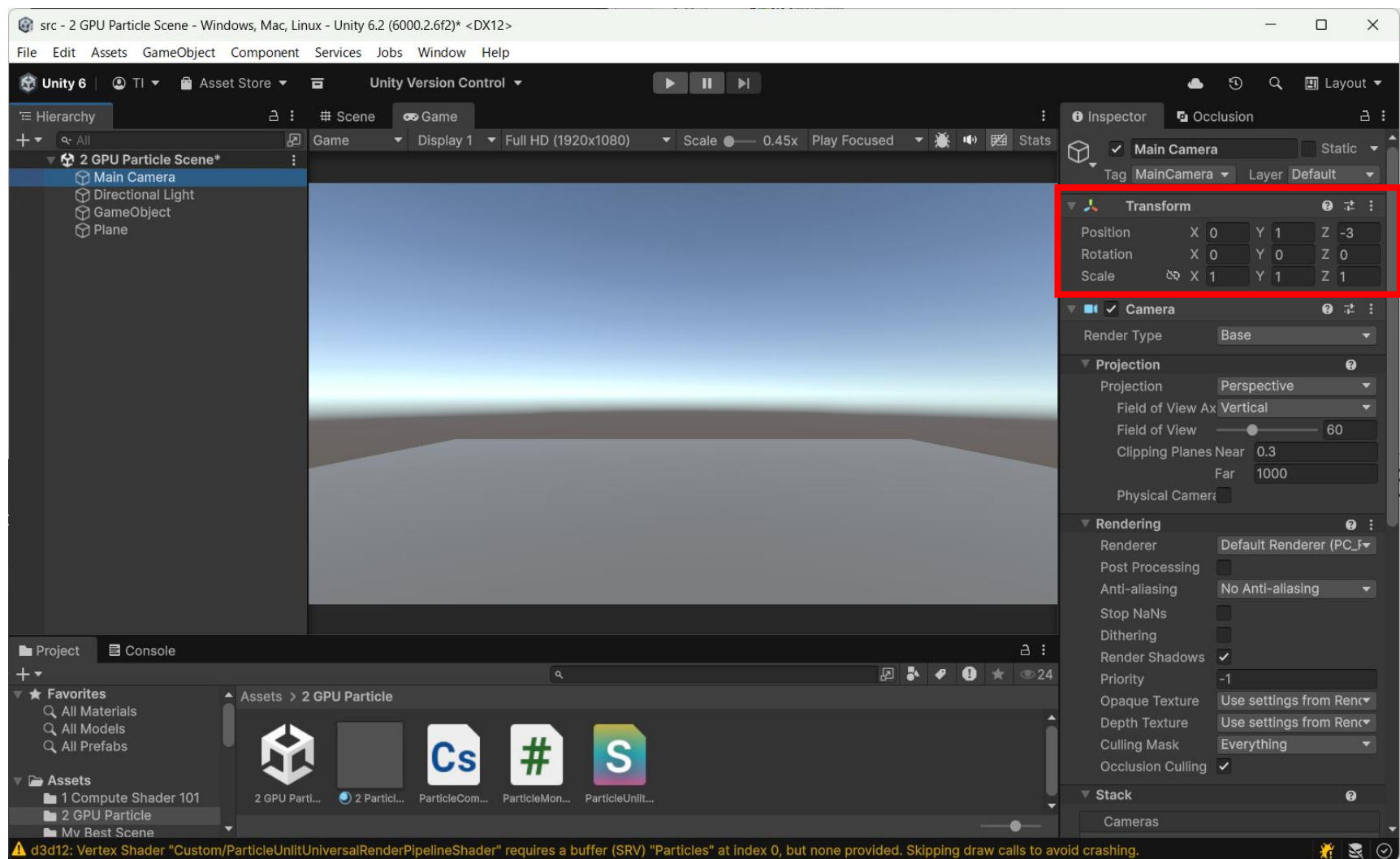
床の追加

- 何もない場所を跳ね返ると変なので床を追加
 - Planeを追加
 - 原点に配置
 - Scaleは1にしているがはみ出しているので、大きくするのも良い



カメラ

- 見やすい位置に調整
 - ここでは-10から-3に近づけた



やってみよう



まとめ

- コンピュートシェーダ概要
- 簡単なコンピュートシェーダ
 - デフォルトで生成されるコンピュートシェーダコードを素直に表示
- GPUパーティクル
 - 大量の計算を並列に実行する