

UnityのGPGPU処理機能である ComputeShaderの紹介・検証

419M508 坂田康輔

Unity

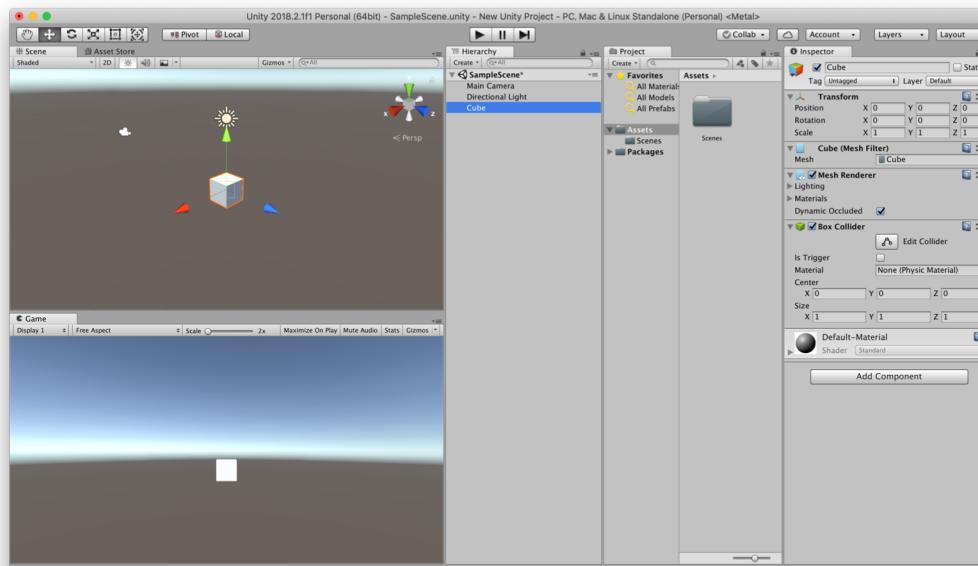
- Unity Technologies社が提供するゲーム開発用プラットフォーム
- 個人や企業で多く利用



Unity Technologies

Unity

- マウス操作型のUIとC#スクリプトで簡単にプロジェクトの作成が可能



Unityの作業画面

Unity

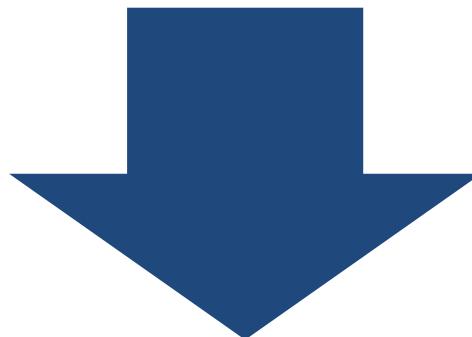
- オブジェクトの動作は基本CPUの処理
- 100000個とかのオブジェクトの処理をするのはつらい
(そもそもそんなにオブジェクトを配置したくない)

Unity

- たくさんオブジェクトが必要
そうな処理
 - ◆ 弾幕処理
 - ◆ 群衆の処理
 - ◆ 流体シミュレーション

Unity

- たくさんのオブジェクトを動かしたい



GPUにオブジェクトの処理をさせたい

Compute Shader

Compute Shaderの説明の前に
Shaderの説明

Shaderとは？

“シェーダー（英: shader）とは、
3次元コンピュータグラフィックスにおいて、
シェーディング（陰影処理）を行う
コンピュータプログラムのこと。”(Wikipediaより)

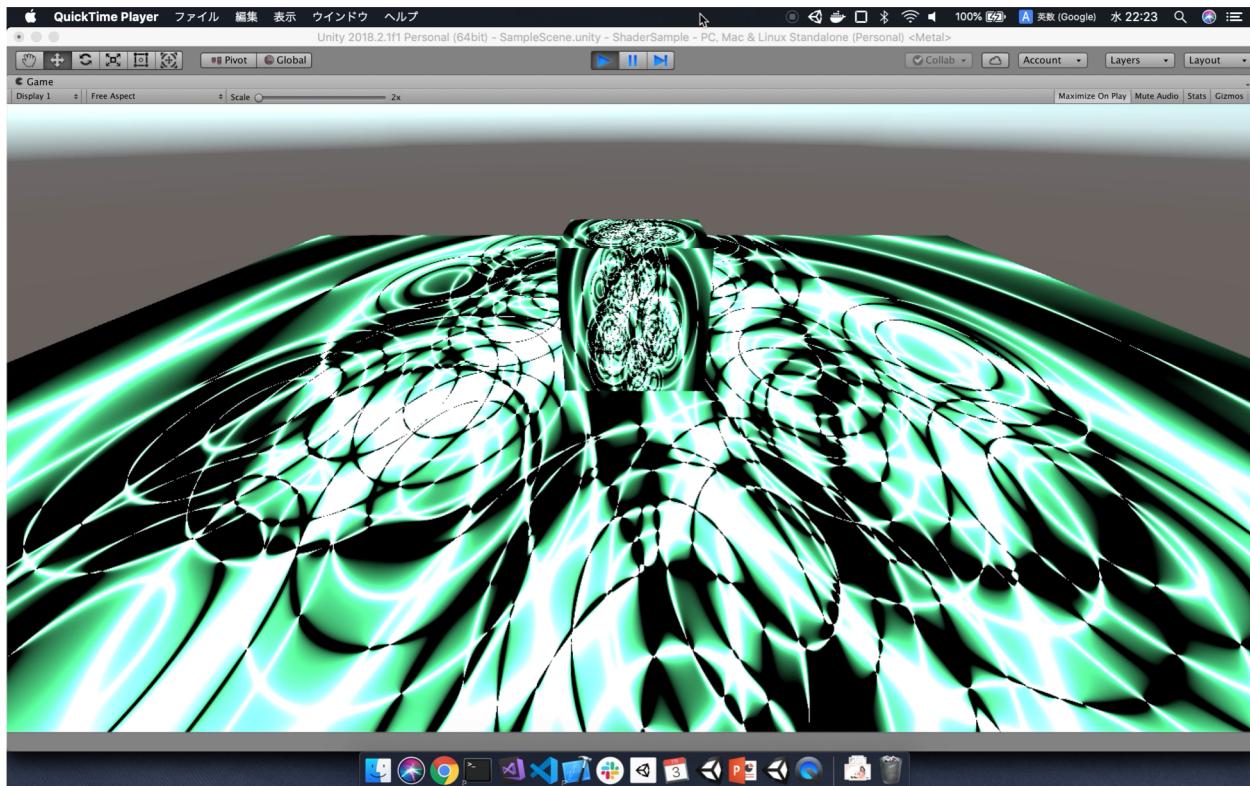
オブジェクトをピクセル単位で色付けし、質感
などを描画する
Shaderの処理にはGPUを使用
(1ピクセル1スレッド)

Shaderとは？

- Shader
 - ◆ Vertex Shader
 - ◆ Fragment Shader
 - ◆ Geometry Shader
- Unityで用いられるShader
 - ◆ **Standard Surface Shader**
 - ◆ Unlit Shader
 - ◆ Image Effect Shader
 - ◆ **Compute Shader**

Shaderとは？

- テクスチャやマテリアルの操作にShaderを適用することでより複雑な描画が可能



Compute Shader

- GPGPUの処理計算を行える
Unityが提供する機能
- Unity上でGPGPU処理を明示的に
行える

Compute Shader

- 使用言語はHLSL(High Level Shading Language)
 - ◆ 基本的なシェーダー言語
 - ◆ GLSLでもかける。らしい。
- 拡張子は.compute
→ shaderに自動コンパイル
- アイコンは→

Compute Shader

● 最低限の記述(ComputeShader側)

```
#pragma kernel CSMain  
  
[numthreads(1,1,1)]  
void CSMain(uint3 id: SV_DispatchThreadID){  
}
```

#pragma kernel : CPU側が呼び出す関数(複数可)
[numthreads(1,1,1)] : GPUの使用するスレッド数(後述)
SV_DispatchThreadID : スレッドID

Compute Shader

● 最低限の記述(Script側)

```
public ComputeShader computeShader;  
public void Function(){  
    int kernel = computeShader.FindKernel("CSMain");  
    computeShader.Dispatch(kernel,1,1,1);  
}
```

computeShader.FindKernel("CSMain");

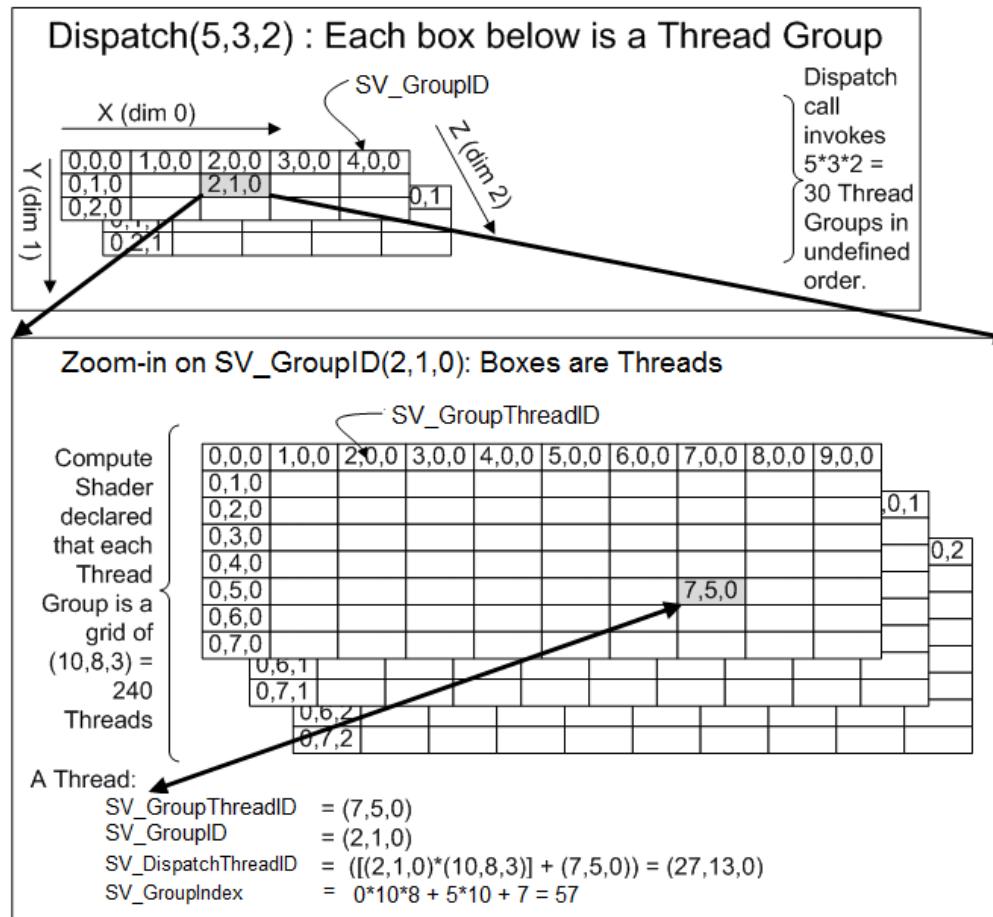
カーネルインデックス(0始まり)を取得

computeShader.Dispatch(kernel,1,1,1);

カーネルを実行、後ろの引数はスレッドグループ数

Compute Shaderの実行

● Compute Shaderの実行単位



Compute Shaderの実行

- Compute Shader側
 - ◆ スレッド・グリッド・グループの3次元
 - ◆ [numthreads(8,1,1)]なら8スレッド
- スクリプト側
 - ◆ 3次元のグループ
 - ◆ Dispatch(kernel,4,4,4)なら
64グループ
- 3 × 3次元のスレッド
 - ◆ 上記の例なら512スレッド

Compute Shaderの実行

- 処理の中で自分のスレッドの位置を把握して処理を行う
 - ◆ **SV_DispatchThreadID**
 - ・ スレッドの位置を特定するセマンティクス
 - ・ 自身がどのグループのどのスレッドかを一意に特定する
 - ・ 前述のスレッド数でスレッドID(2,0,0),スレッドグループID(2,3,1)のときSV_DispatchThreadIDは $((2,3,1)*(4,4,4)) + (2,0,0) = \mathbf{(10,12,4)}$
 - ・ 別のセマンティクスもある(SV_GroupIDなど)

Compute Shaderの実行

- CPUとデータを転送
 - ◆ intやfloat型変数はそのまま転送可能
 - ◆ 構造体や配列はBufferを用意
 - ◆ テクスチャも(もちろん)扱える

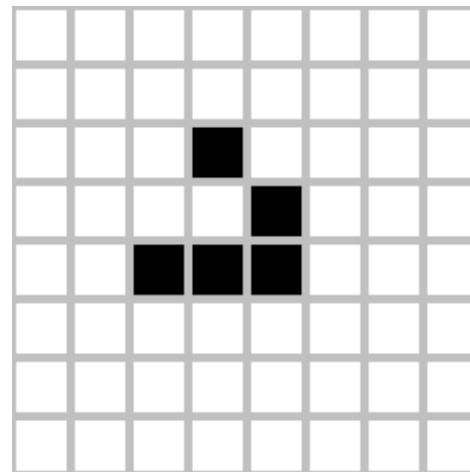
```
float time = Time.deltaTime;  
float[] data = new float[10];  
ComputeBuffer computeBuffer = new ComputeBuffer();  
computeBuffer.SetData(data);  
  
computeShader.SetFloat("Time",time);  
computeShader.SetBuffer(kernel,"Data",computeBuffer);
```

```
float Time;  
RWStructuredBuffer<float> Data;
```

Compute Shaderの使用例

● ライフゲーム

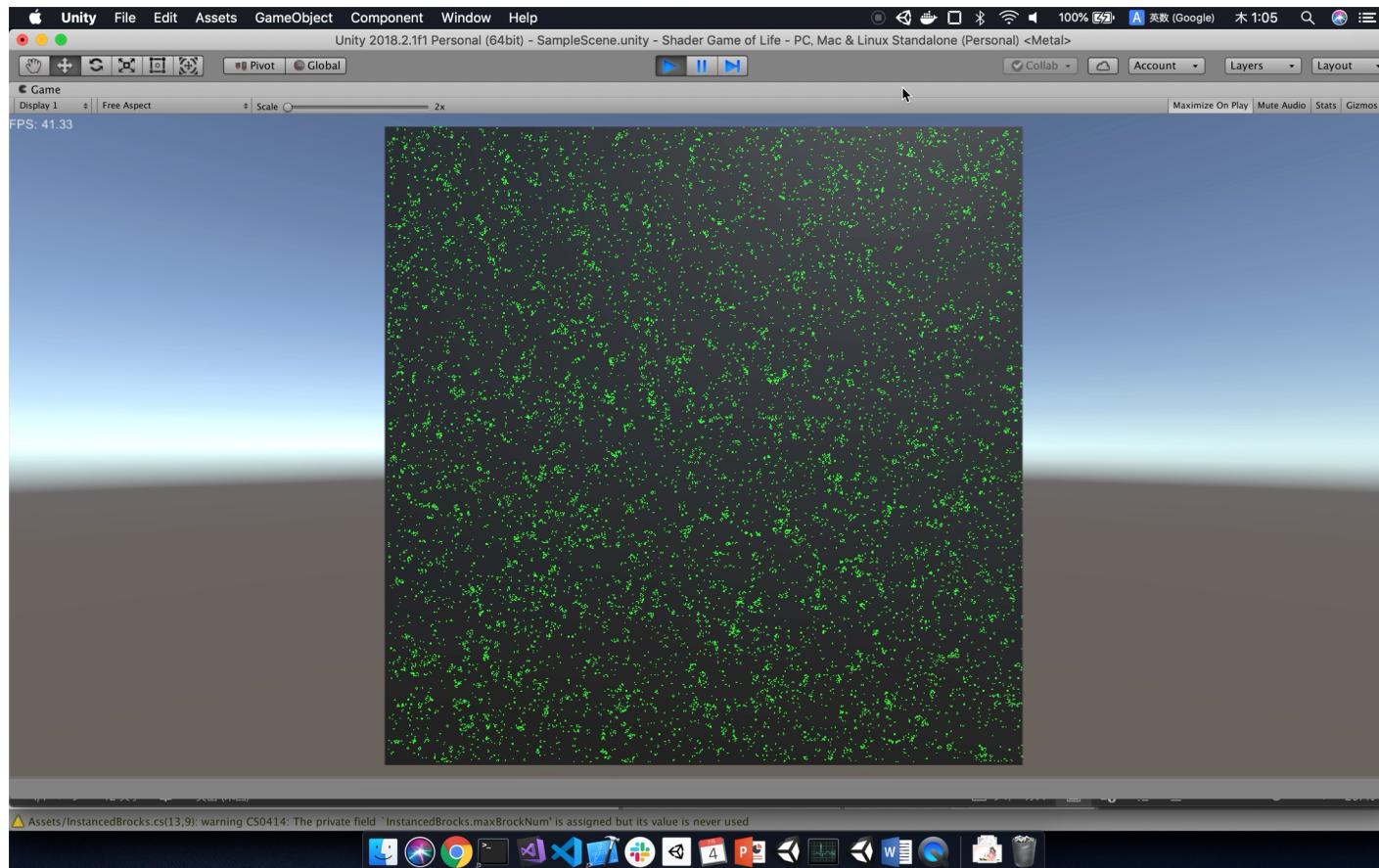
セルの生死状態をCompute Shaderで判定
テクスチャで描画することを考えると
ピクセルごとにスレッドを割り当てて処理
ピクセルの書き換え



Compute Shaderの実行例

● 実行結果

1024x1024のセル



Compute Shaderの使用例

OS : macOS Mojave

CPU : 2.3 GHz Intel Core i5

RAM : 8 GB

GPU : Intel Iris Plus Graphics 640 1536 MB

2000x2000では60FPS程度

4000x4000では30FPS程度

8192x8192ではUnityが落ちる

Compute Shaderの使用例

OS : Windows 10 Home

CPU : 3.60 GHz Intel Core i7-7700

RAM : 16.0 GB

GPU : NVIDIA GeForce GTX 1060 6GB

14000x14000くらいまでなら30FPS程度で動作

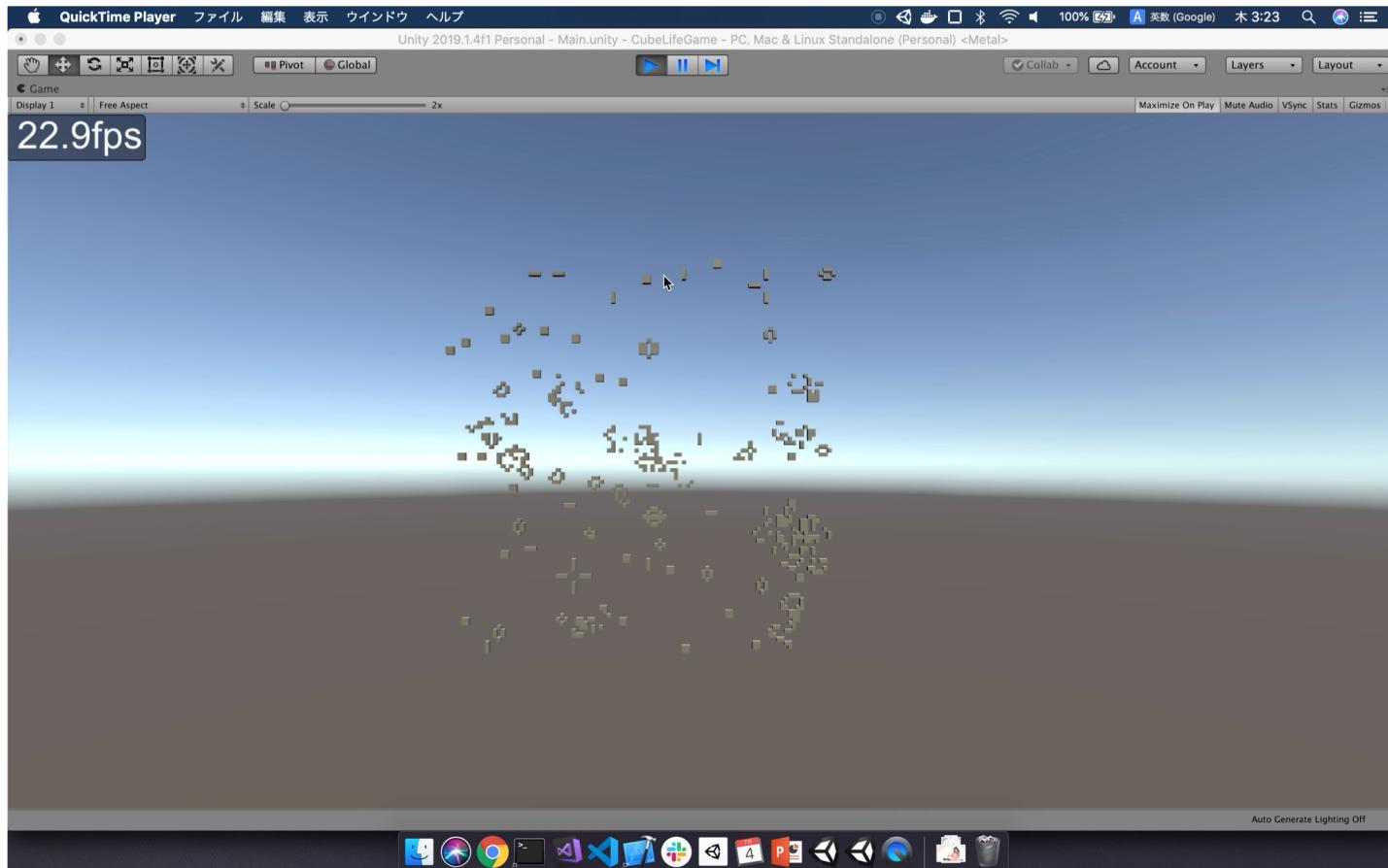
Compute Shaderの使用例

- オブジェクトの操作
 - ◆ ライフゲームの実装をテクスチャではなくオブジェクトで行う
 - ◆ セルの生死はオブジェクトの生死
 - 実際には処理が重いのでアクティブ/非アクティブ
 - ◆ セルの生死の評価のみを行い、更新はスクリプト側で行う
 - Standard Shaderを使えばできるらしいが理解できなかった

Compute Shaderの使用例

● 実行結果

100x100のセル



考察

- テクスチャに比べ実行性能が劣る
 - ◆ オブジェクトのアクティブ/非アクティブ化の処理が重い
 - Shaderを用いてオブジェクトを“見えなく”する
 - オブジェクトを一つのグループとしてみる

他Shaderとの組み合わせ

Compute Shaderの使用例

- 物理演算をさせる
 - ◆ オブジェクトを複数の粒子の集合と見て粒子の位置関係を計算する
 - ◆ Shader芸の合わせ技
- レイトレーシング

まとめ

- UnityのGPU処理機能であるCompute Shaderの紹介
- Compute Shaderを利用したGPGPUプログラムの実装

参考文献

Unity公式HP <https://unity.com/ja>

The Book of Shaders

<https://thebookofshaders.com>

使用したソースコード

<http://www.00jknight.com/blog/gpu-accelerated-voxel-physics-solver>

https://qiita.com/aa_debdeb/items/7c9c3d14b0264964a4e