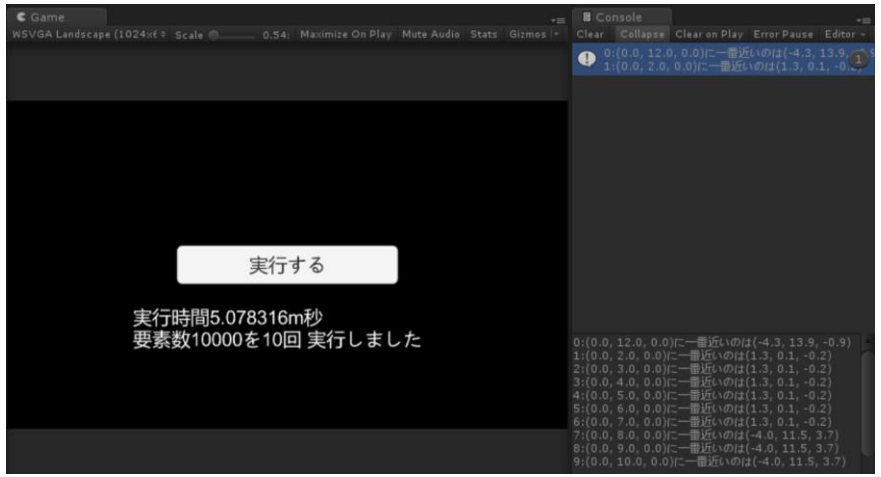
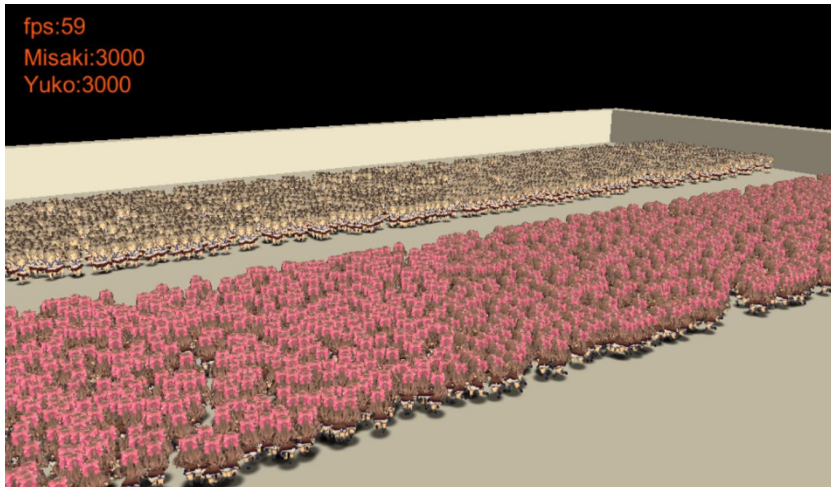
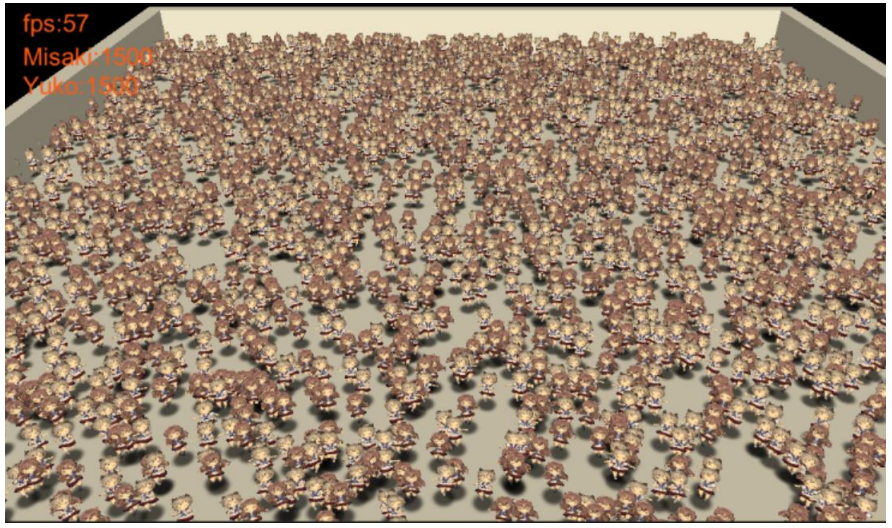
本日のお品書き



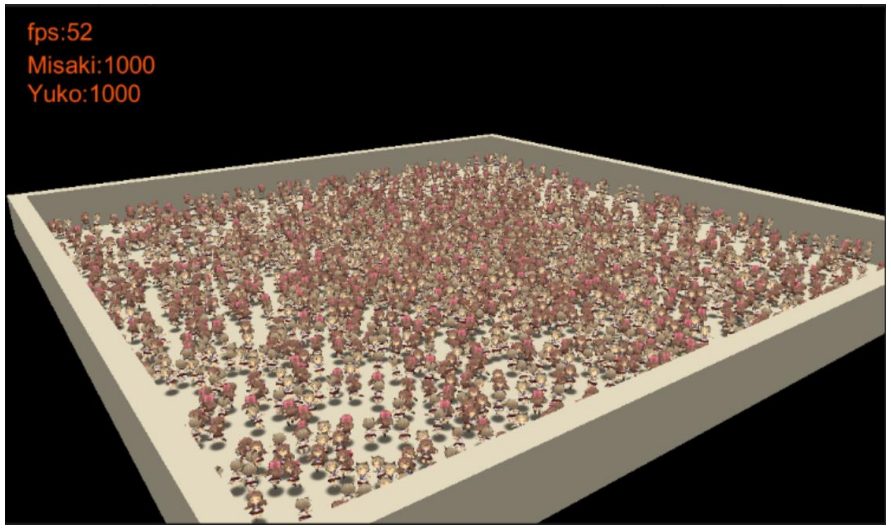
Work1.IJob



Work2.IJobParallelFor



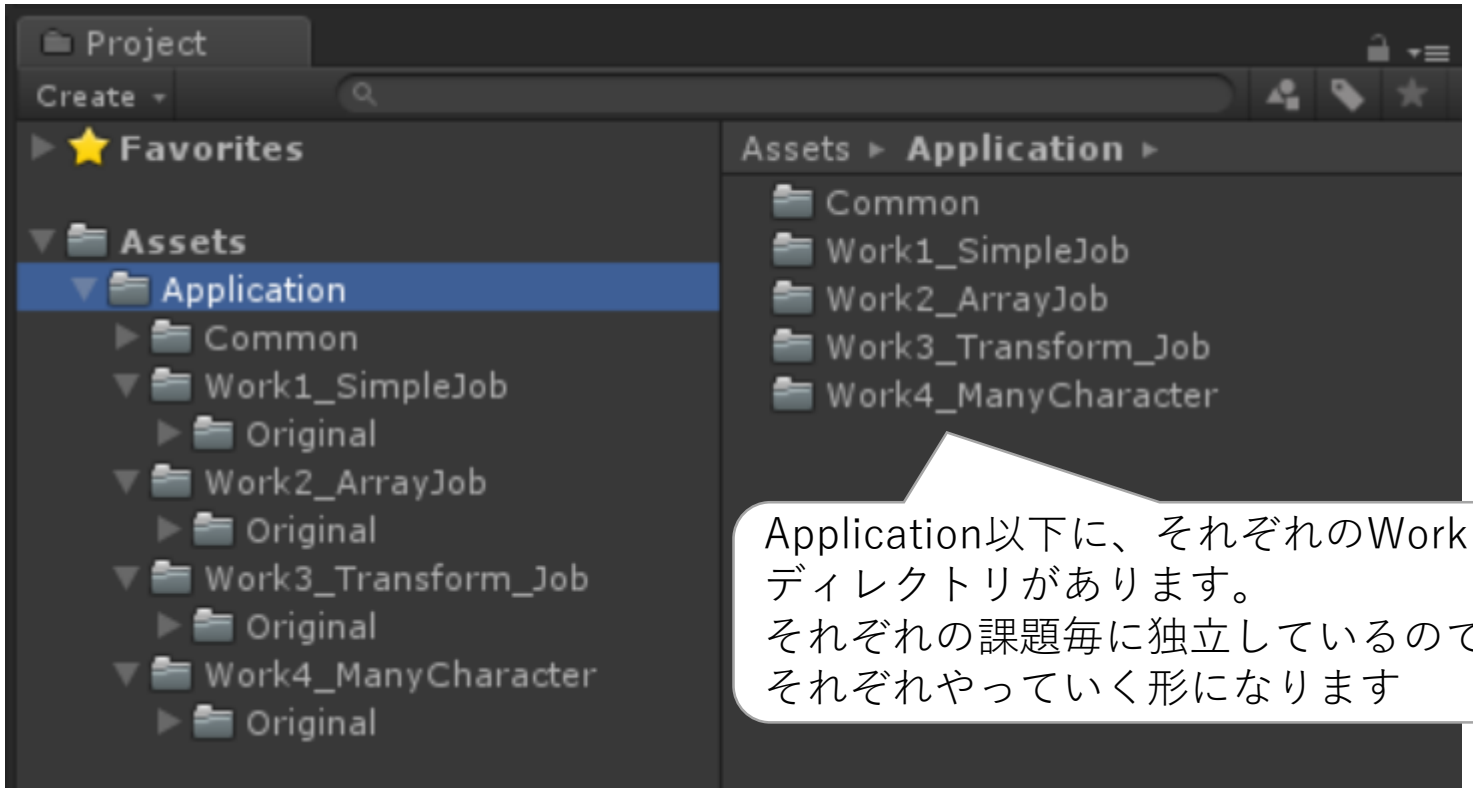
Work3.IJobParallelForTransform



Work4.RayCastCommand

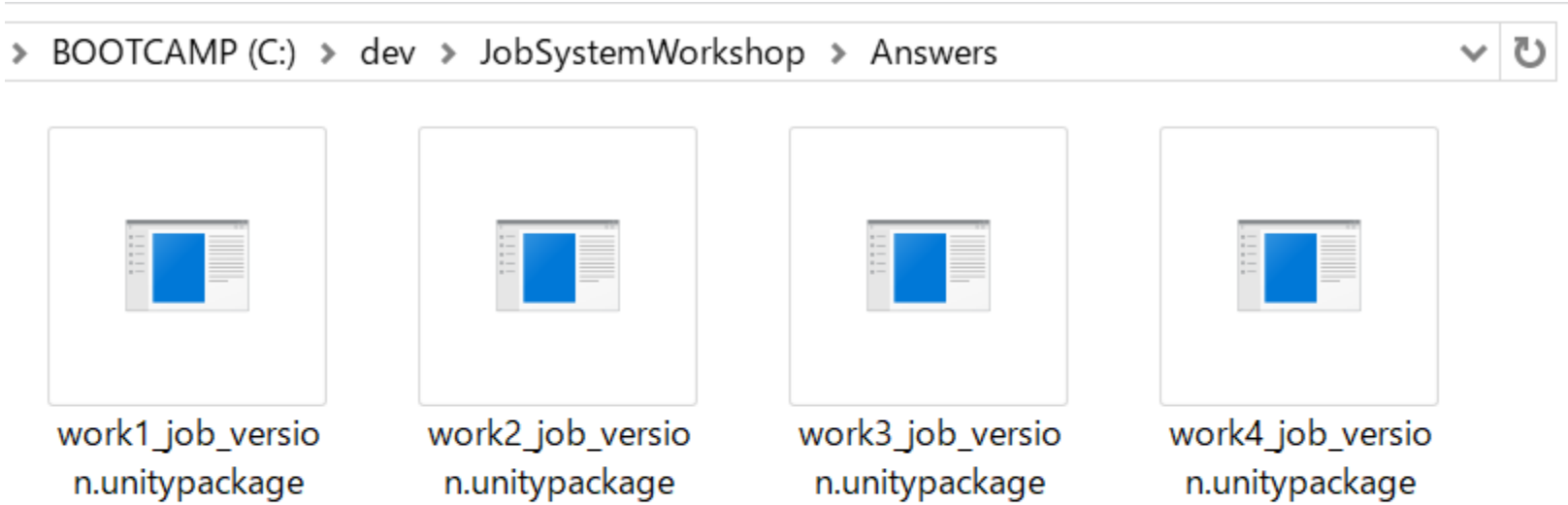
まず初めに…

プロジェクトについて



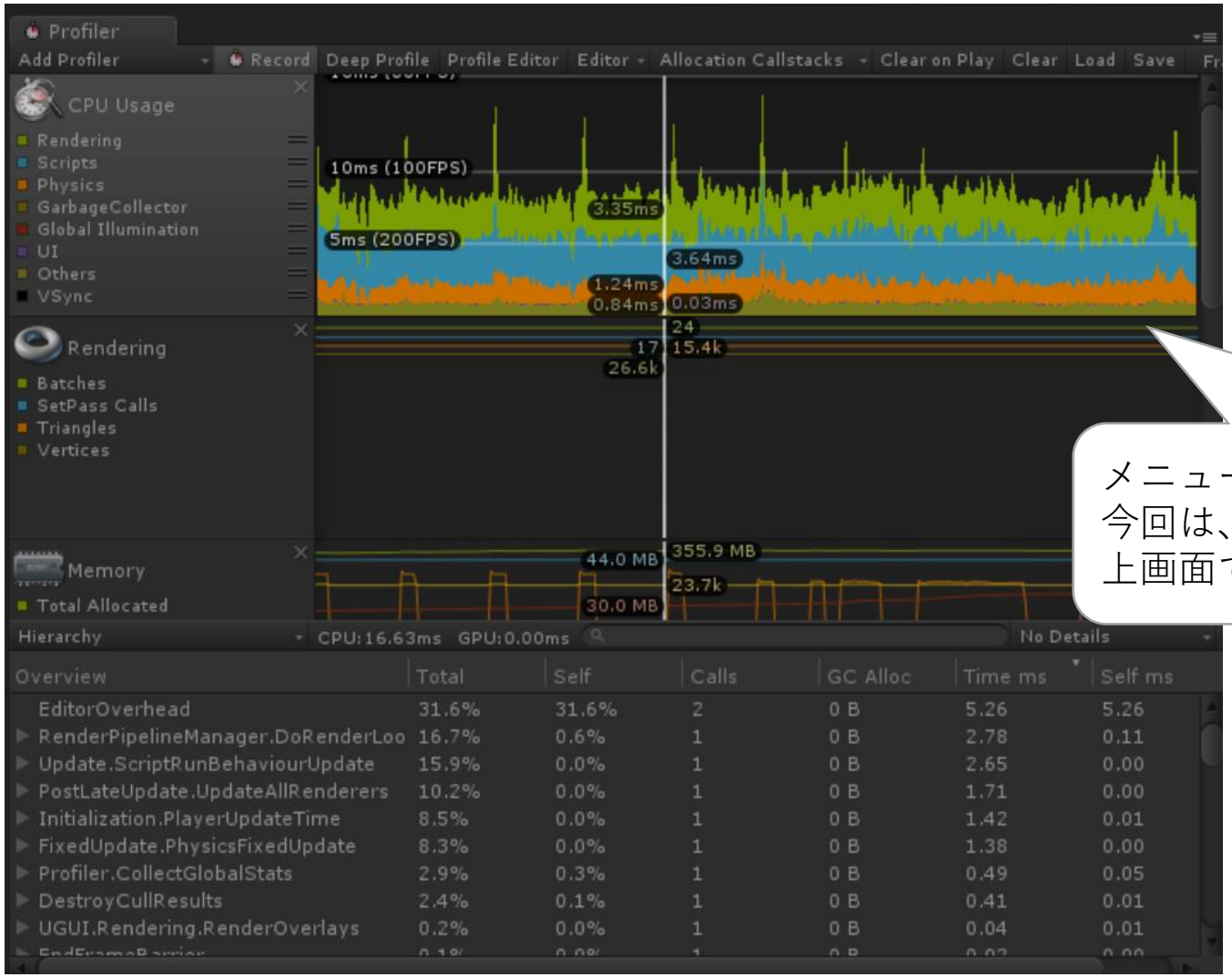
Application以下に、それぞれのWorkに応じて
ディレクトリがあります。
それぞれの課題毎に独立しているので、こちらを
それぞれやっていく形になります

各課題の解答について



こちらで、それぞれの課題の解答を別シーン・別スクリプトの形で用意しました。
Answersディレクトリにある、それぞれのUnityPackageをプロジェクトへインポートすると、
解答のスクリプトを確認できます。

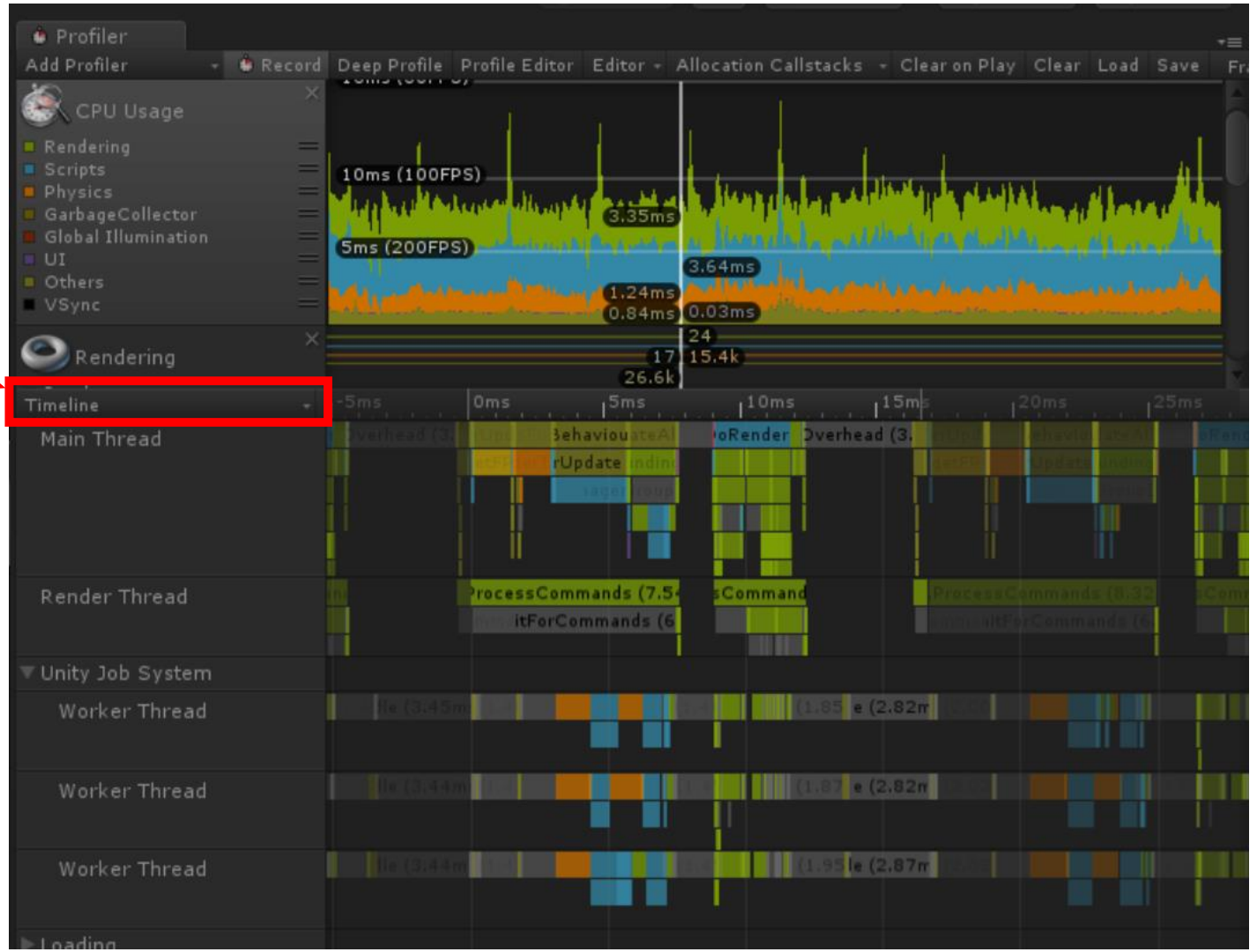
始める前にProfilerの使用について紹介(1)



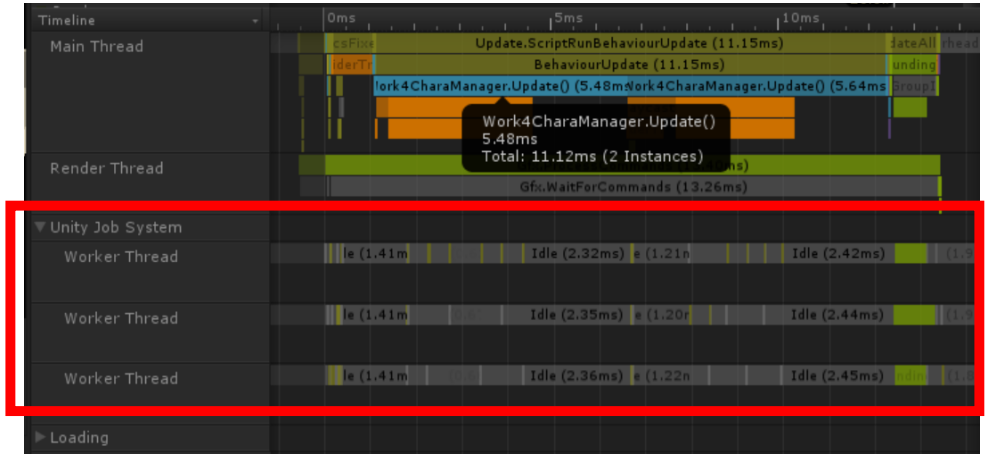
メニューの Window/Profiler で呼び出します。
今回は、CPU Usageの項目しか見ません。
上画面で、CPU Usageを選択しておきましょう

その前にProfilerの使用について紹介(2)

Hierarchy表示だと、別スレッドの様子を確認できないので、ココをクリックしてTimelineにしましょう



今回の課題進行で注目するところ



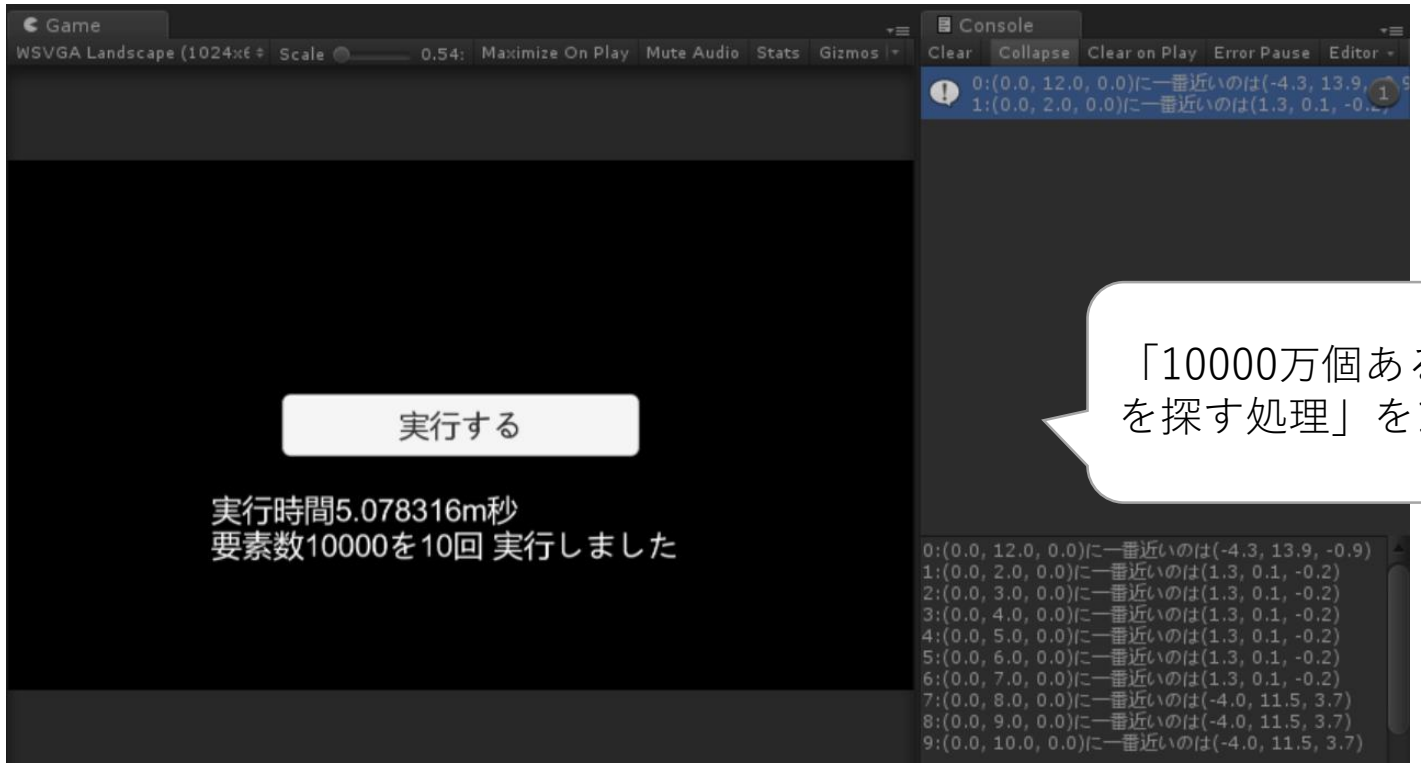
Job対応前

Job対応後

Job対応前は、スクリプトのUpdate処理中は、MainThreadのみが頑張って働き、WorkerThreadがIdle状態で、何も仕事していない状態です。
Jobの対応を行う事で、Update処理の一部をWorkerThreadにも分担してもらう事で早く処理が終わります。

それでは、
それぞれの課題紹介へ

Work1.IJob



「10000万個ある位置情報の中から一番近い場所を探す処理」を10か所について行います。

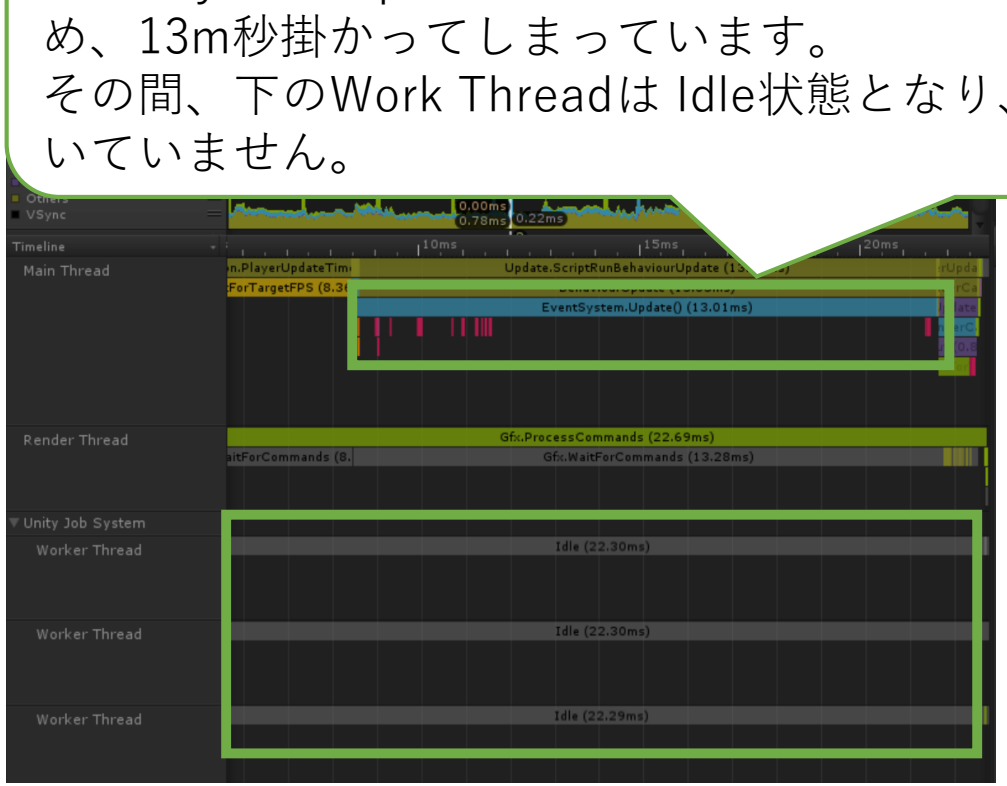
「10000個の要素を探す処理」×10回をしています。
「10000個の要素を探す処理」をIJobを利用してJob化して、10個のJobにして並行処理するようにしましょう。

Work1NearestFind.cs でここをJob化

```
/// <summary>
/// 計算のコア部分です。
/// [課題] 今は 全てシングルコアでしていますが、これを並行処理できるようにしてみましょう
/// </summary>
private void CalculateCore()
{
    for (int i = 0; i < sourcePositions.Length; ++i)
    {
        // これを一つのJobにして並行処理できるようにしましょう
        var result = FindNearestPosition(sourcePositions[i], targets);
        Debug.Log( sourcePositions[i] + "に一番近いのは" + result );
    }
}
```

Work1.Job化後のBefore / After

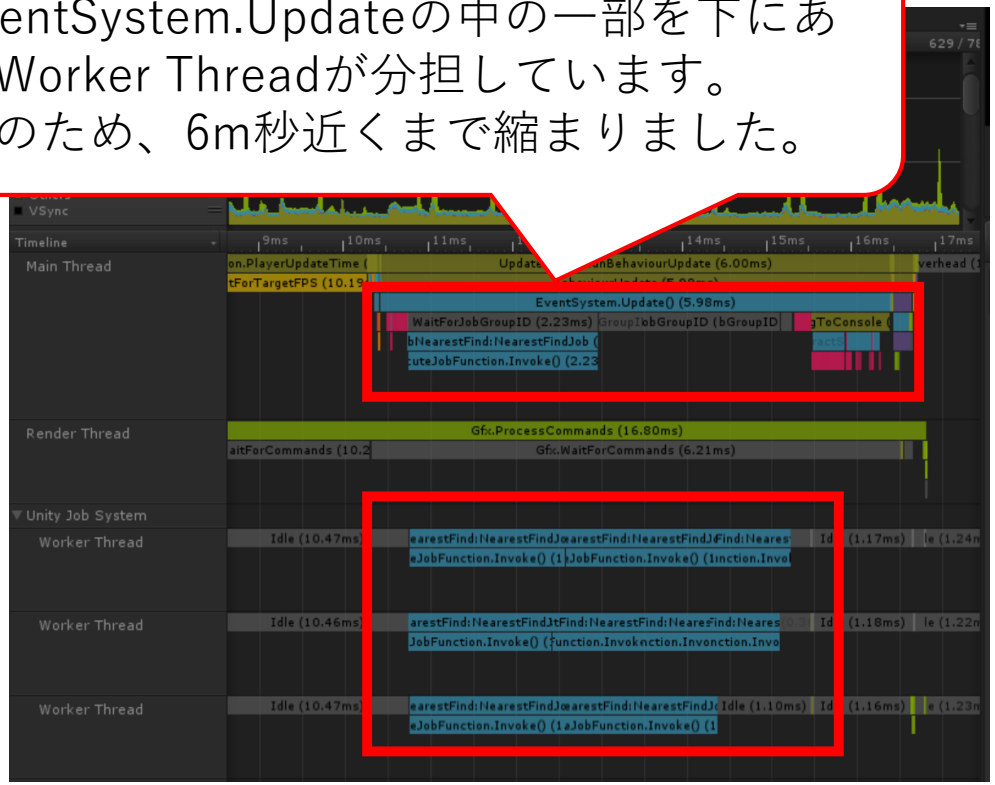
ボタンを押したときのフレームを確認します。
EventSystem.Updateの中で全て処理しているため、13m秒掛かってしまっています。
その間、下のWork Threadは Idle状態となり、働いていません。



Before

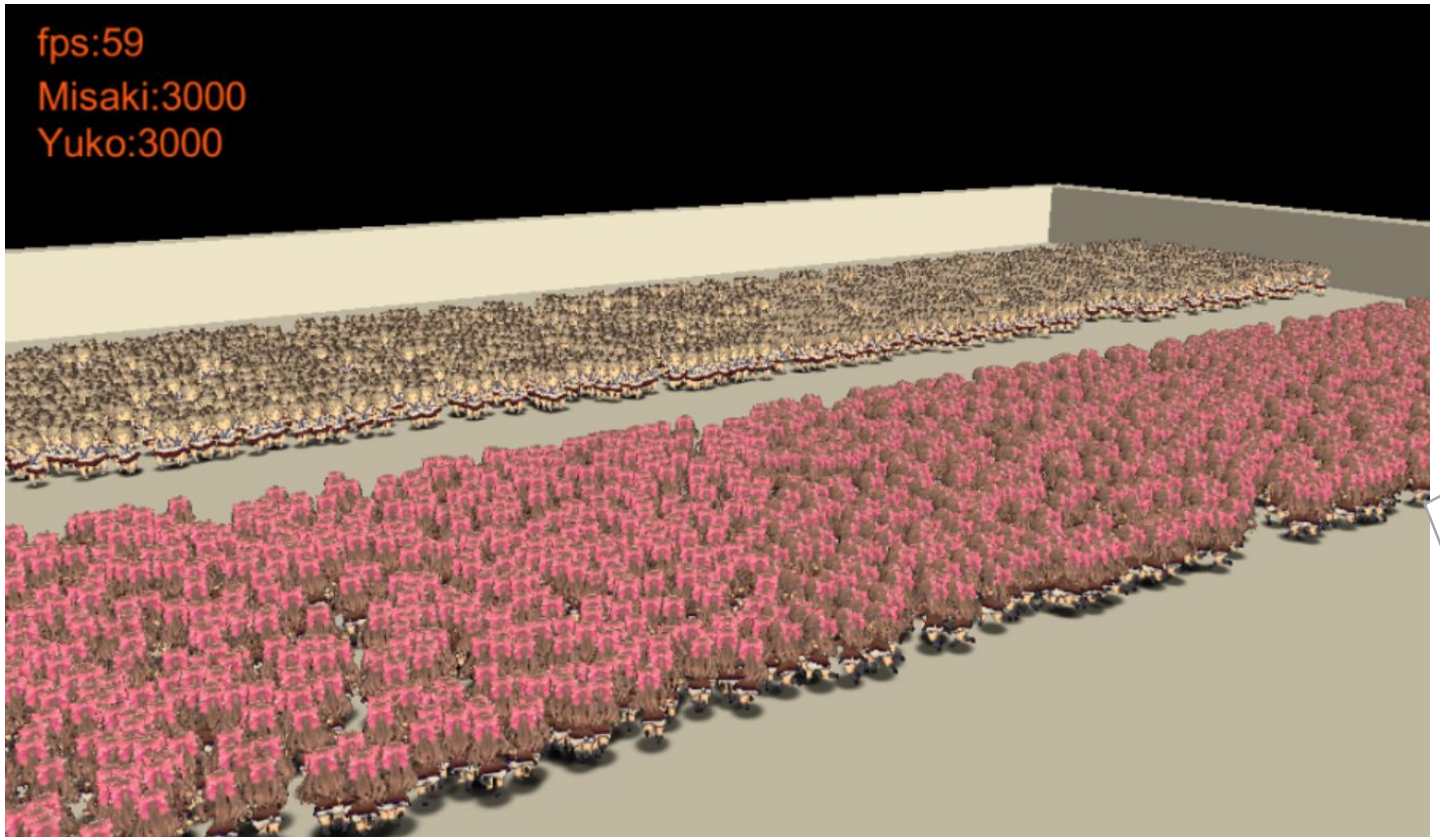
Job対応

EventSystem.Updateの中の一部を下にあるWorker Threadが分担しています。
そのため、6m秒近くまで縮まりました。



After

Work2. IJobParallelFor



群 VS 群の雰囲気を楽しめます。

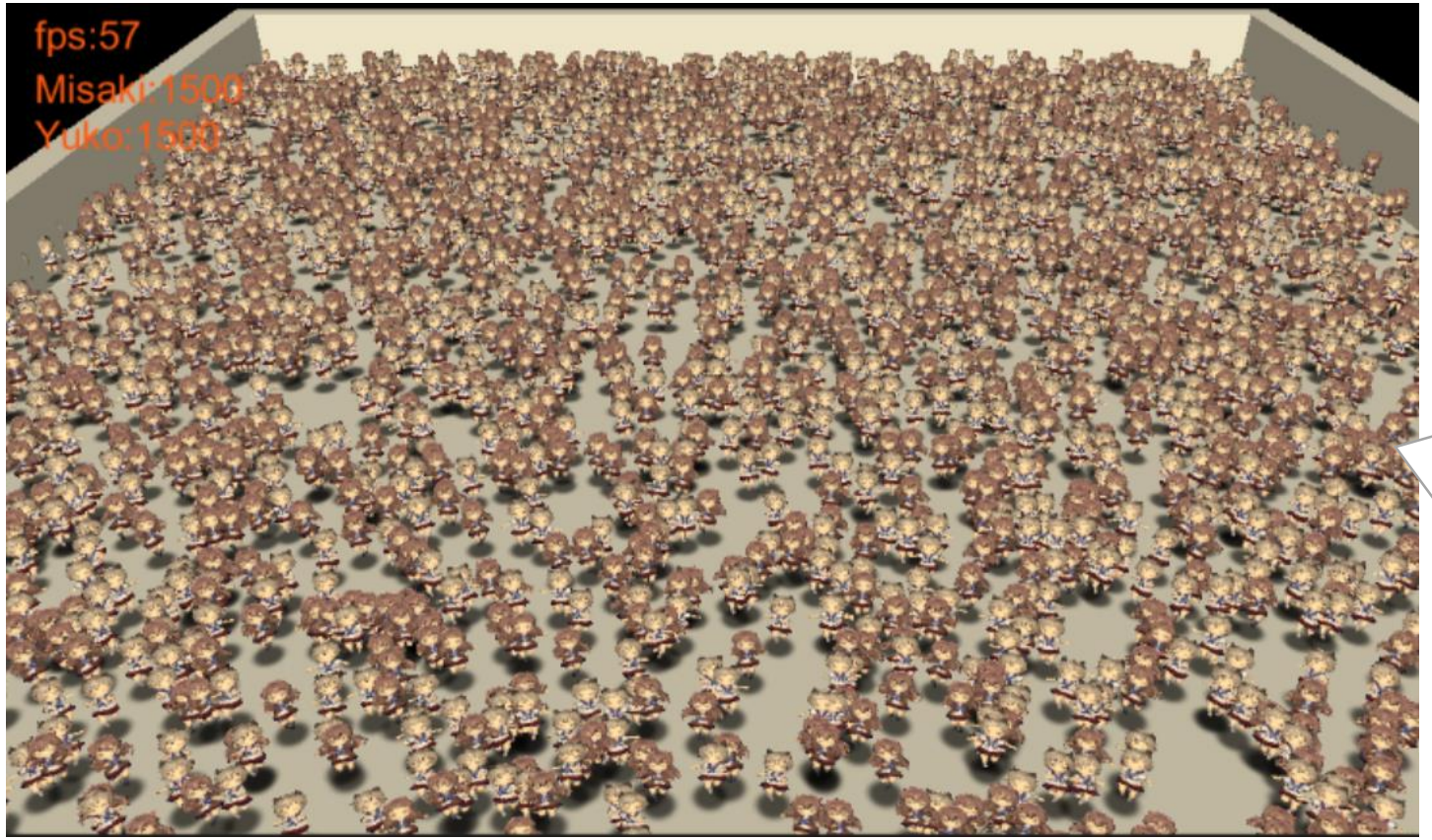
Job化すればAndroid端末(GalaxyS6)でも60FPSで動きます。
※当たり判定がないので、すれ違っていくだけです。

キャラクターの位置等がNativeArrayで保持しており、forループで移動処理・アニメーション更新を行っています。
この辺りを IJobParallelForを使ってJob化しましょう。
※ちなみに描画は全てCommandBufferで行っています

Work2CharaManager.cs のここをJob化

```
// void Update() 関数内にて
// ここを IJobParallelForを利用して並行処理にします
for (int i = 0; i < characterNum;++i )
{
    // 移動処理
    characterPosition[i] = characterPosition[i] + characterVelocity * deltaTime;
    // はみ出し処理対応
    //.....  以下省略…
}
```

Work3. IJobParallelForTransform



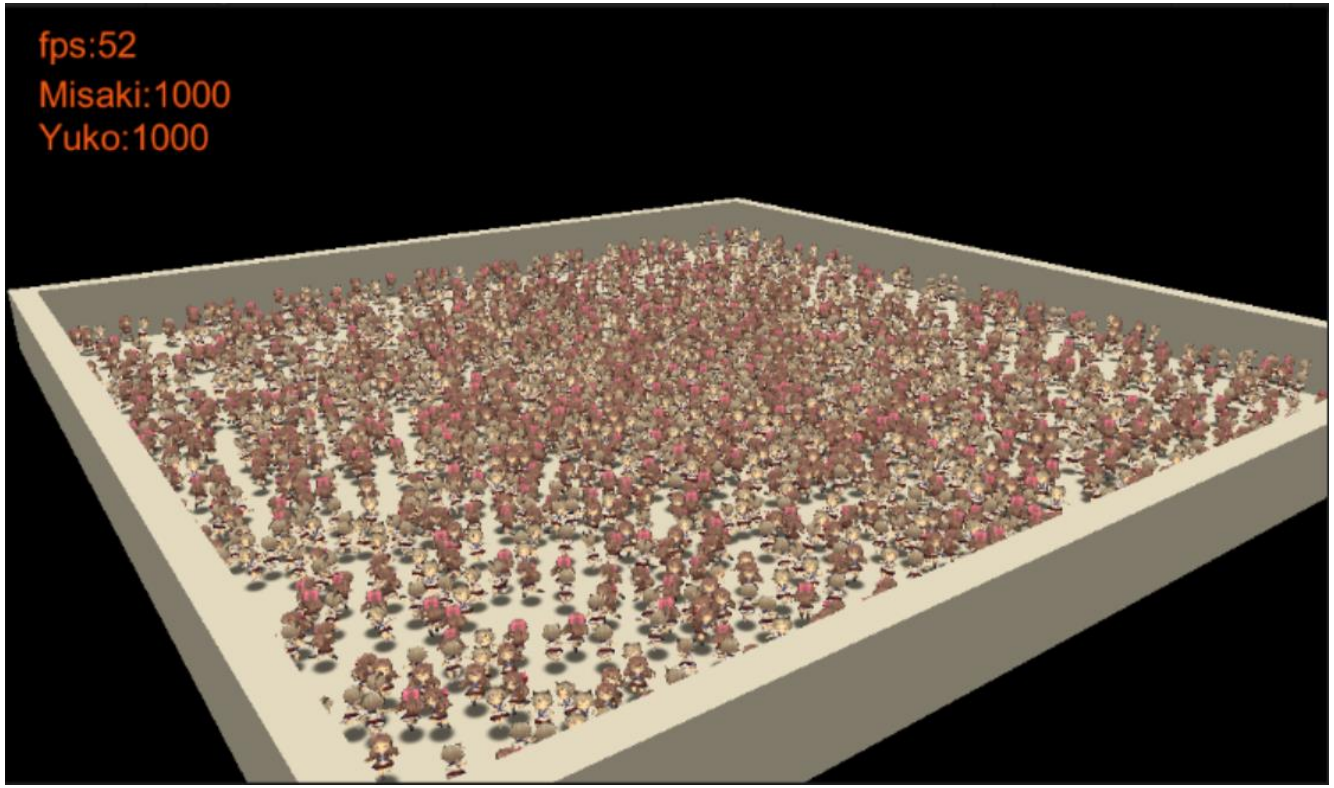
大量のキャラクターが押し寄せてくるだけです。
Job化すれば、Galaxy S6で 30FPS前後で動作します

キャラクターはそれぞれ GameObjectです。
Transform更新をIJobParallelForTransformを利用して、Job化しましょう

Work3CharaManager.cs のここをJob化

```
// void Update()関数内にて
// 全キャラクターの移動処理を行います
// [課題]ここもうまく計算処理を並行して出来るように
Vector3 cameraPosition = Camera.main.transform.position;
cameraPosition.y = 0.5f;
for (int i = 0; i < characterNum; ++i)
{
    Vector3 delta = new Vector3( Mathf.Sin(Time.realtimeSinceStartup + i ) * 0.8f ,0.0f, -1);
    characterTransforms[i].position = characterTransforms[i].position + delta * Time.deltaTime;
    if (characterTransforms[i].position.z < -15.0f)
    {
        characterTransforms[i].position = ...長くて改行してしまったので省略
    }
    characterTransforms[i].rotation = ...長くて改行してしまったので省略
}
```

Work4. RaycastCommand



キャラクターは互いにColliderを持ち、ぶつかると向きを反転します。
Job化すると GalaxyS6で 30FPS前後で動作します

キャラクターは、それぞれ進行方向にRayを飛ばして衝突検知しています。
まずはRaycastCommandを利用して、処理のJob化をしましょう。
また他にもJob化できるところはJob化してみましょう

Work4CharaManager.cs をJob化

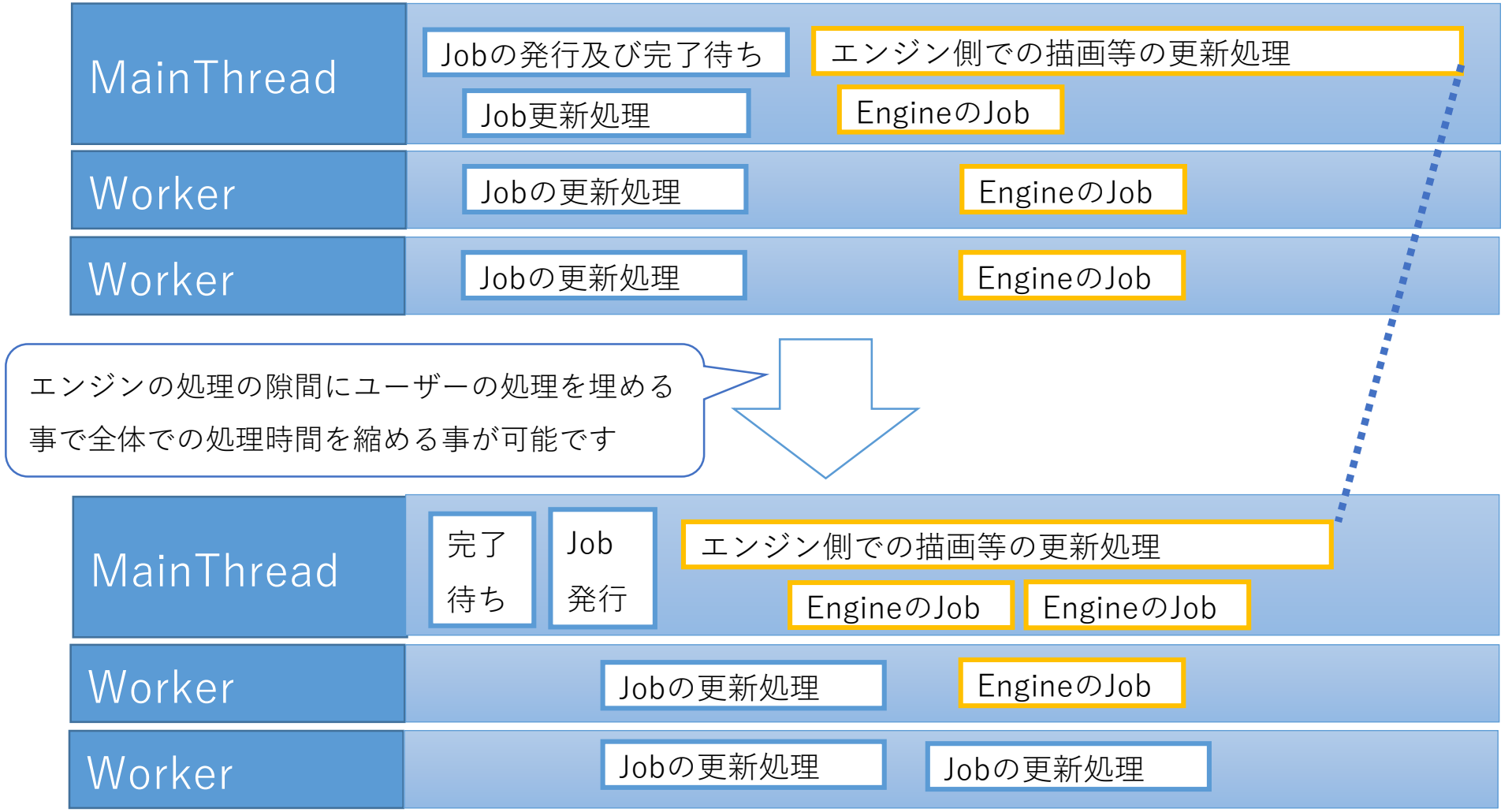
```
// void Update関数内にて
// 全キャラクター分 進行方向にRayを飛ばします。
// 何かにぶつかったら180° 進行方向を変えます
// [課題]ここを RaycastCommandを使って並行して出来るように
for (int i = 0; i < characterNum; ++i)
{
    if (Physics.Raycast(new Ray(characterTransforms[i].position, velocities[i]), Time.deltaTime * 3.0f))
    {
        velocities[i] = -velocities[i];
    }
}
```

C# JobSystemでのテクニックについて

次Updateの先頭で完了待ちをすることでより高速化します

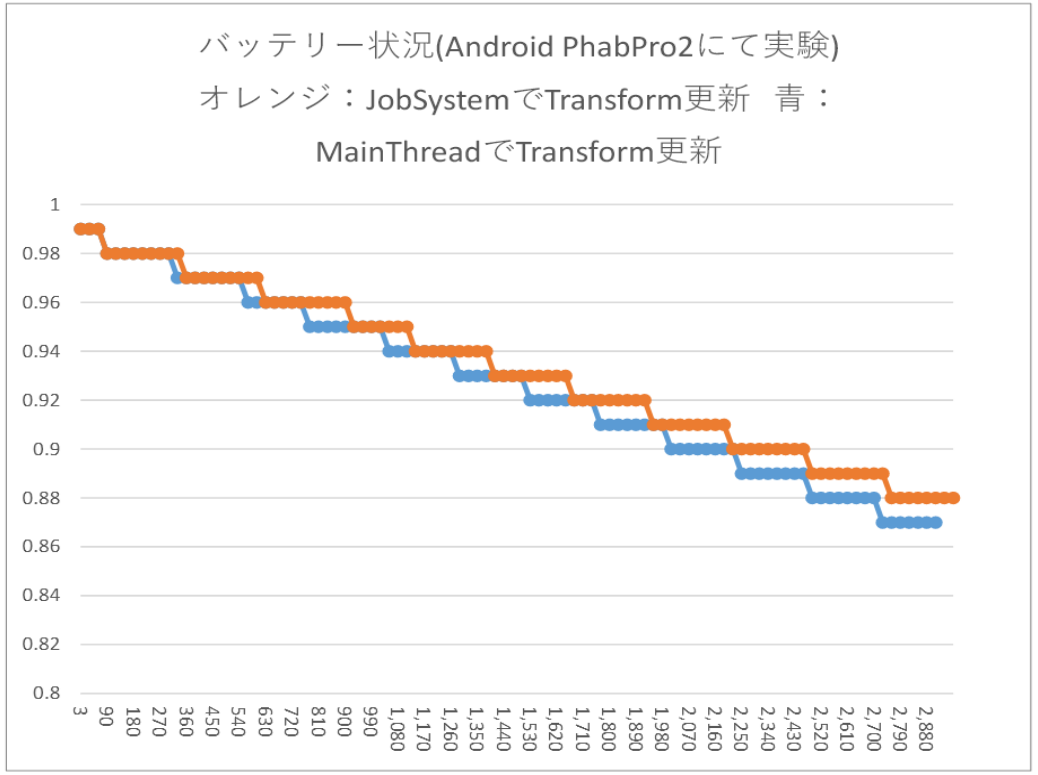
```
class JobTest : MonoBehaviour{
  private JobHandle handle;
  public Transform[] transformArray;

  void Update(){
    //前のフレームで発行したJobが完了するのを待ちます
    jobHandle.Complete();
    // 並行して処理をしたいTransformのリストを定義します
    var transformAccessArray = new TransformAccessArray(transformArray);
      // Jobを作成します
      var myTransformUpdateJob = new MyTransformUpdateJob(){
      time = Time.deltaTime ,  objNum = transformArray.Length
    };
    // Jobを発行してます。この時点では、Todo用のQueueに積まれるだけです
    JobHandle handle = myTransformUpdateJob.Schedule( transformAccessArray );
    // [重要] 実際に積まれたJobの実行を促します
    JobHandle.ScheduleBatchedJobs();
  }
}
```

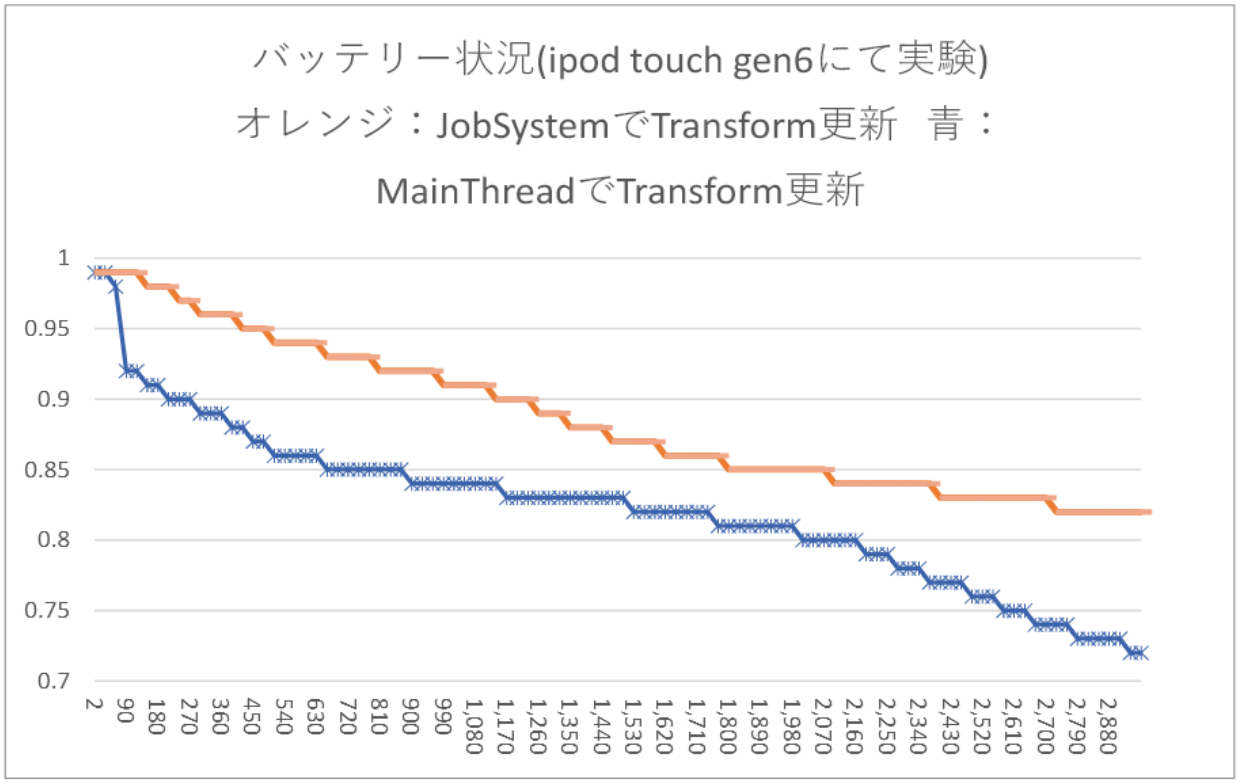
次Updateの先頭で完了待ちをすることでより高速化します



C# JobSystemを使って、
処理の効率化をすると、バッテリー消費も
減ります

実験結果





https://github.com/wotakuro/UnityJobSystemTest

こちらで計測。GPU・通信の影響を抑えるため、機内モード/画面輝度最低固定、ゲームのFPSを10固定にし、CPUでのバッテリー消費が顕著に出る環境にして実験。

Android / iOS共に、JobSystemにして処理時間が減った結果結果、CPUの休む時間が増え、バッテリー消費が下がったと思われる。

これ以下は おまけコンテンツです
C# JobSystemとは関係のない部分での
高速化話です

おまけ・・・

- PlayerLoopSystemについて
- ビルボードでの描画について
- GPUインスタンシングでの描画
- Scriptable RenderPipeline(SRP)について
- Culling周りについて

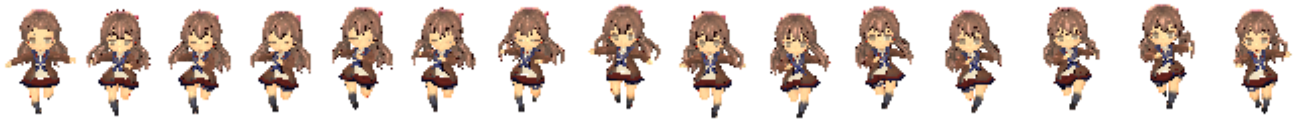
PlayerLoopSystemについて

- 2018.1から入った新機能です。
  - メインループから余計な処理を省く事が出来たり、システム側の呼び出しタイミングをフックして何か処理を挟むことが出来ます

- このプロジェクトではPhysicsの更新処理を止めるためにPlayerLoopSystemをカスタマイズしています。
  - プロジェクト内にある「CustomPlayerLoop.cs」で、実際に処理しております

ビルボードでの描画について(1)

今回のキャラクターは全てビルボードでのSprite切り替えアニメーションで実装しています。



下記ツールを利用し、事前にアニメーションしたキャラから8方向分のTextureを生成しています。カメラの角度を考慮して、適切な向きの画像を選ぶようになっています

https://github.com/wotakuro/BillBoardCreator

ビルボードでの描画について(2)

今回、こちらの描画をとった主な理由ですが…

Animator / SkinnedMeshRendererの組み合わせだった時に、先にそちら側が根を上げてしまい、十分なキャラクター数を表示できないためです。

※一応、下記のようなソリューションはあるのですが、今回は見送りました
https://blogs.unity3d.com/jp/2018/04/16/animation-instancing-instancing-for-skinnedmeshrenderer/

GPU Instancingの描画について(1)

今回、キャラクターを描く際には DynamicBatchingではなくGPU Instancingでの描画をしています。

Androidでは、OpenGL 3.1以上。iOSではMetal対応であればGPU Instancingが利用できます。市場の端末でも多くが対応しているため、今後のタイトルではGPU Instancingの採用は十分に選択肢に入ります。

※DynamicBatchingでは、CPU側でメッシュの結合をしています。これもJobの仕組みを使うのでWorkerThreadで結合しています。そのため、C# JobSystemと資源の奪い合いとなります。
そのため、その懸念がないGPU Instancingを今回は全面的に使用することにしました

GPU Instancingの描画について(2)

今回はSpriteRendererではなく、MeshRenderer+独自コンポーネント(BoardRenderer)にてGPU InstancingでのSprite描画を実現しています。
SpriteRendererで描画すると、uvが異なる状況ではうまくドローコールがまとまらなかったためです。

ShaderにInstancing毎の変数で描画用の矩形情報を渡し、
頂点Shaderでuvの調整を行っています。

SRPについて

今回のプロジェクトでは、独自に作成したScriptableRenderPipeline(SRP)も利用しています。

各キャラクターを描くとき前に Zバッファーの更新だけ行い、その後に通常の描画をするという形にして描画負荷を抑えたかったためです。

SRPについて 良いところ

・良い点

- 描画順等のRenderingに関する部分に手を入れやすくなりました。

  => 従来は各コンポーネントの設定を上手く利用して描画順の制御をしていました。

  これには各コンポーネントの特性を熟知している必要がありました。

  SRPの登場により、スクリプトでわかりやすく記述することが出来る用になりました。

- 今まで手を入れることが出来なかった細かい部分にも手を入れることが出来る用になりました。

  => 影処理の置き換え、描画グループごとのバッチング/インスタンシングの制御、
  描画順のソートの方法やCullingに関する設定の変更等々出来る用になりました。

  ※ Cullingに関する処理を全てオフにすることは出来ません。

  これはAnimator等で更新を判断するために利用されているからです。

SRPについて 注意するところ

・導入の際に気を付ける点

- SRPと既存のレンダリングシステムで全く同じShaderは利用できません。

  => SRPの対応をするためには、ShaderPassのLightModeを以下のように独自に定義する必要があるためです。

  「Tags { "LightMode" = "BasicPass" }」

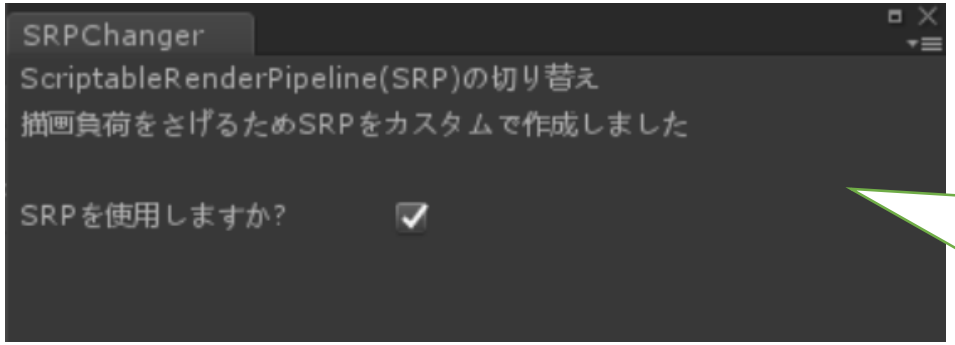
- Surface Shaderは利用不可能です。

  => Shader Graph Toolで作成されたShaderは、Scriptable Render Pipelineを考慮して動作します。

- Cameraのコールバックが効かなくなるため、その内容をSRPで実装しなおす必要があります。

  => ImageEffect等を実装しなおしが必要となります。

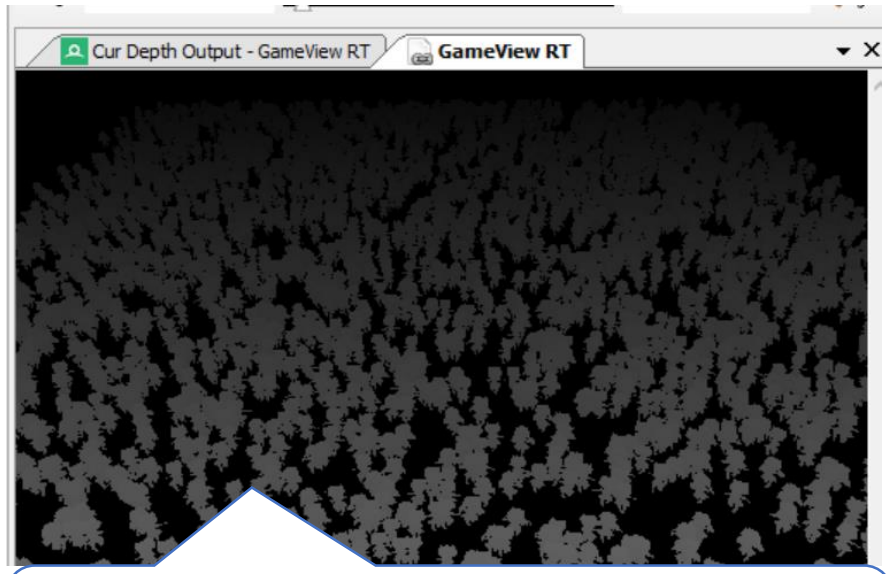
SRPについて ON/OFFで効果を確認



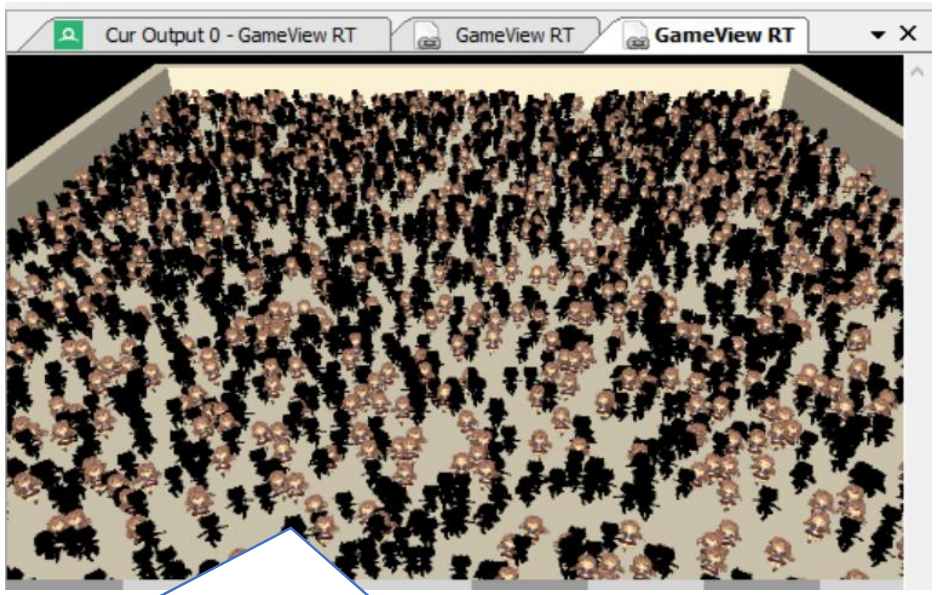
メニューの Tools/SRPChangerにて呼び出せます。
Work3 / Work4 では、チェックボックスでSRPの
ON / OFFを切り替えられます。
※Work2は対応していません

SRPについて ZPrepassの実現

下記のような形で、「キャラクターを描く前にZBufferの更新のみを行う事で、実際に塗られる描画面積を減らして、描画負荷を下げる」というのを SRPで記述しています。



キャラクターの深度情報を事前にZBufferに
先に書いておきます



そのあとに、事前に書き込んだ情報を利用して、
キャラクターの描画処理を行います

※RenderDocにて
https://docs.unity3d.com/jp/current/Manual/RenderDocIntegration.html

SRPについて　描画時のソート

半透明オブジェクトを描く際に、

・通常であれば、描画の破綻をさけるため「奥→手前」の順番を優先して描画していきます。

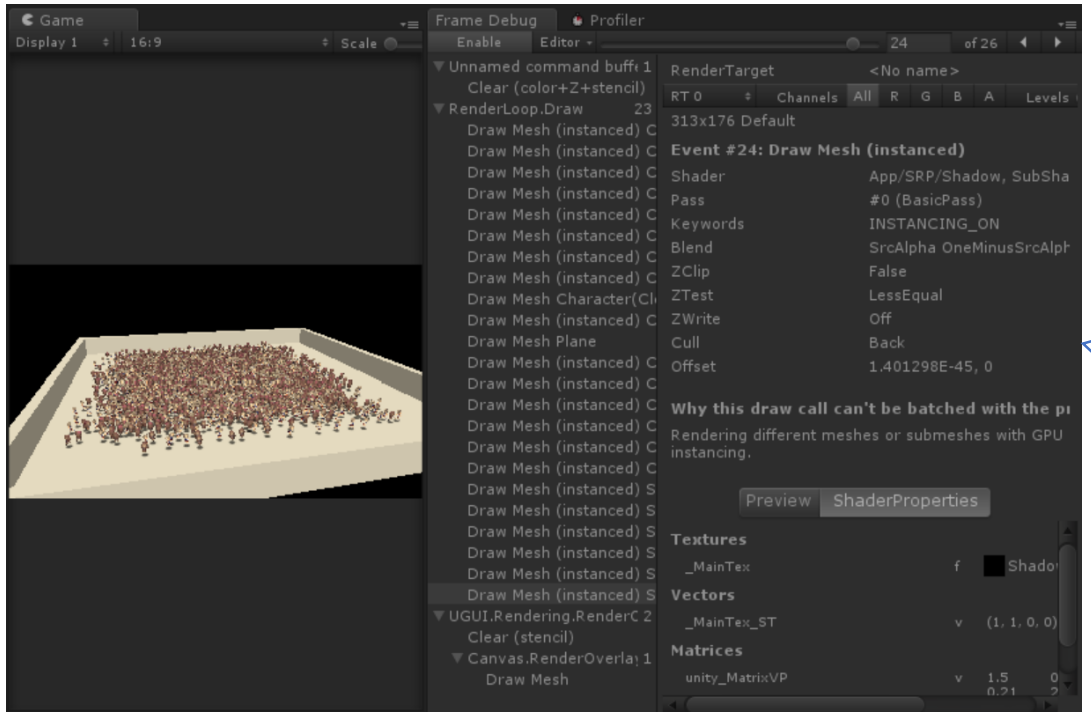
今回は「ゆうこ」と「みさき」の２キャラが交互に入り混じっていますので、都度Material切り替えを強要されます

・今回は ZBufferに事前に書き込んでいる情報を利用すれば、破綻なく描画出来る状態です。

そのため、半透明であっても奥→手前の順番で描かずともよくなります。なので、キャラクター描画時に「奥→手前」のソートをせず、Material切り替えを抑えるのを優先して描画させています

SRPについて

MyScriptableRenderPipeline.cs が独自に作成したSRPになります。
今回のプロジェクト専用の形で描画パスをしています。



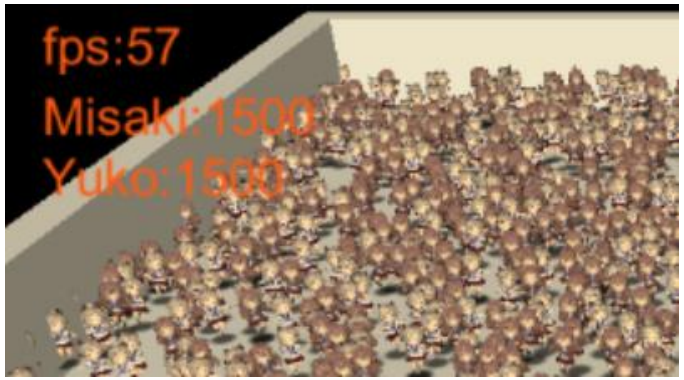
効果については、FrameDebuggerでご確認ください。

Cullingについて

皆様、お気づきでしょうか？

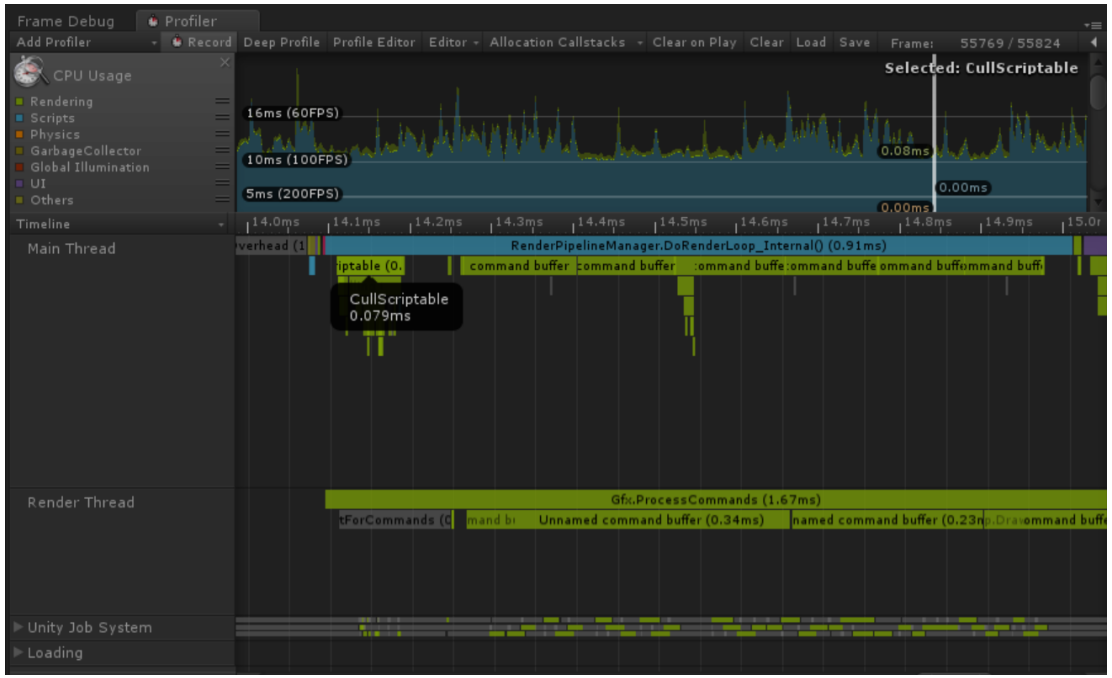
Work3よりも、Work2の方がより多くのキャラクターを出していることに！

※これが「おまけ」コンテンツの最後の話題になります。



左のWork2では6000体のキャラクターを60FPSを出していますが、Work3では3000体でイッパイです。

Cullingについて Profilerにて



左がWork2、右がWork3のProfilerの結果です。見ての通り、Work3の方が緑色のRenderingに関する処理のコストが高くなっています。Work3ではCullingだけでも1m秒以上かかってしまっています。

Cullingコストは描画対象(Renderer)が多くなればなるほど多くなります。

Work2ではCommandBufferを利用して直接描画しているので、エンジン側でのCullingをスキップしており、描画関係のCPU処理はかなり減っています。ただし、今回はWork2では何もCulling/距離に応じたソートをしていません。そのため、GPU側へ処理を少し負担してもらっています。

実は、C# JobSystemの効果を最大限活用させるために、描画周りを事前にいじっていたのでした！
そして、そちらの方が効果は大きいデス！