Unity®

UNITY 2022 LTS 版 → EBOOK

Unity の 上級開発者向け Data-Oriented Technology Stack (DOTS)

目次

はじめに

このガイドでは、Unity の Data-Oriented Technology Stack（DOTS）の潜在的なパフォーマンス上の利点について説明します。スタックに含まれる各パッケージと機能の概要を説明するとともに、デザイン指向設計（DOD）に関連する主要なコンセプトと、DOD による影響について解説します。API の詳細には触れませんが、DOTS の新しいチュートリアルやその他の学習用リソースへのリンクが全体を通して数多く提供されており、詳しく学習することができます。

この eBook の目標は、ご利用の Unity プロジェクトに DOTS の機能を部分的または全面的に導入することが有益かどうか判断するために必要な知識を提供することです。

著者と専門家の貢献者

こちらの eBook は、Unity DOTS のエンジニアと外部専門家の協力のもとで作成されました。主な著者は Unity のシニアソフトウェアエンジニア、Brian Will です。本ガイドの作成にご協力いただいた他の専門家は以下の通りです。

— リアルタイム 3D と Unity のエデュケーター、Nik Lever 氏

— ソフトウェアエンジニア、Steve McGreal 氏

— Unity のソフトウェアエンジニア、Daniel Kierkegaard Andersen

— Unity の製品マネジメントディレクター、Laurent Gilbert

パフォーマンスについて

経験豊富なゲーム開発者であれば、ターゲットプラットフォームにおけるパフォーマンスの最適化が 開発サイクル全体に関わる課題であることを知っているでしょう。あなたのゲームはハイエンドの PC ではうまく動作するかもしれませんが、同じくターゲットにしているローエンドのモバイルプラットフォームではどうでしょうか？一部のフレームは他のものより時間がかかり、顕著な不具合が生じることはありませんか？ロード時間が長くてイライラしたり、プレイヤーがドアをくぐるたびにゲームが数秒間フリーズしたりはしませんか？このような状況では、現在の体験が劣るだけでなく、機能を追加することも事実上不可能になります。環境の細かなディテールやスケール、メカニクス、キャラクターや行動、物理演算、プラットフォームなど、すべて追加できなくなってしまうのです。

このような現象の原因は何でしょうか？多くのプロジェクトにおいてはレンダリングが原因です。テクスチャが大きすぎたり、メッシュが複雑すぎたり、シェーダーが高コストすぎたり、バッチ処理、カリング、LOD が効果的に使われていなかったりします。

もう一つのよくある落とし穴は、複雑なメッシュコライダーを使いすぎて物理シミュレーションのコストが大幅に増加していることです。あるいは、ゲームシミュレーションそのものが遅いのかもしれません。ゲームの独自性をもたせるために記述している C# コードで、1 フレームあたりの CPU 時間が長すぎる可能性もあります。

では、どうすれば高速な、少なくとも遅くはないゲームコードを書けるのでしょうか？

過去数十年間、PC ゲーム開発者は、ただ待つことでこの問題を解決できることが多くありました。1970 年代から 21 世紀にかけて、CPU のシングルスレッド性能は一般に数年ごとに 2 倍になった（ムーアの法則と呼ばれる現象）ため、PC ゲームはそのライフサイクルを通じて「魔法のように」高速化しました。しかし、過去 20 年間における CPU のシングルスレッド性能の向上は比較的緩やかでした。その代わり、CPU のコア数は増加の一途をたどっており、現在ではスマートフォンのような小型ハンドヘルドデバイスにさえ複数のコアが搭載されています。さらに、ハイエンドとローエンドのゲームデバイス間の格差が広がりましたが、プレイヤーの大部分は数年前のハードウェアを使用しています。より高速なハードウェアの登場を待つことは、もはや実行可能な戦略とは言えません。

よって、ここで問われるべきは「そもそもこの CPU コードはどうして遅いのか?」ということです。よくある落とし穴はいくつかあります。

— **ガベージコレクションが顕著なオーバーヘッドや一時停止の原因になる：**これは、ガベージコレクターがアプリケーションのメモリの割り当てと解放を管理する自動メモリマネージャーとして機能するためです。ガベージコレクションは CPU とメモリのオーバーヘッドを伴うだけでなく、コードの実行を何ミリ秒も一時停止させることがあります。このような一時停止は、小さな不具合、あるいはユーザー体験を損なうスタッターとして現れるかもしれません。

— **コンパイラーが生成するマシンコードが最適ではない：**コンパイラーによっては、生成するコードの最適化が他のコンパイラーよりはるかに劣っているものもあり、その結果はプラットフォームによって異なります。

— **CPU コアが十分に利用されていない：**今日では最もローエンドなデバイスにもマルチコア CPU が搭載されていますが、マルチスレッドコードを書くのは難しくエラーも発生しやすいため、多くのゲームではロジックのほとんどをメインスレッドで実行しています。

— **データがキャッシュフレンドリーではない：**キャッシュのデータへアクセスする方が、メインメモリからデータをフェッチするよりもはるかに高速です。しかし、システムメモリにアクセスすると、CPU が何百 CPU サイクルも待機しなければならない場合があります。そのため、可能な限り CPU がキャッシュからデータを読み書きするようにするのが良いでしょう。

  これを可能にする最も単純な方法は、メモリの読み書きを順番に実行することです。隙間のない連続した配列としてデータを格納することで、最もキャッシュフレンドリーな読み書きを実現できます。逆に、データがメモリ全体に非連続的に散らばっている場合、データへのアクセス時に負荷の高いキャッシュミスが大量に発生しやすくなります。CPU が要求するデータがキャッシュメモリに存在せず、代わりに低速のメインメモリからデータをフェッチする必要が生じます。

— **コードがキャッシュフレンドリーではない：**コードがキャッシュに存在しない場合、そのコードを実行する際に、システムメモリからロードする必要があります。推奨される戦略の一つとして、システムメモリからのロード回数を減らすために、関数を呼び出す場所をできるだけ少なくすることが挙げられます。例えば、特定の関数をフレーム内のさまざまな場所で呼び出すよりも、1 つのループ内で呼び出す方が、1 フレームにつき最大 1 回のロードで済むため効率的です。

— **コードが過度に抽象化されている：**その他の問題の中でも、抽象化はデータとコードの両方に複雑さをもたらすため、前述の問題が悪化しやすくなります。ガベージコレクションなしで割り当てを管理するのが難しくなる、コンパイラーの最適化効率が下がる可能性がある、安全で効率的なマルチスレッドが難しくなる、データとコードがキャッシュフレンドリーでなくなる傾向がある、などの問題が生じます。そのうえ、抽象化によってパフォーマンスコストが分散され、コード全体が遅くなり、最適化すべき明確なボトルネックがなくなってしまう傾向があります。

上記の問題はすべて、Unity プロジェクトでよく見られるものです。もう少し具体的に見てみましょう。

- C# では手動で割り当てられたオブジェクト（ガベージコレクションされないオブジェクト）を作成できますが、C# やほとんどの Unity プロジェクトでは、ガベージコレクションされる **C# のクラスインスタンス**を使用するのが一般的です。実際には、Unity ユーザーは長い間、プーリングと呼ばれる手法でこの問題を軽減してきました（プーリングを使用すると、そもそもガベージコレクション機能をもつ言語を使用する目的が部分的に無効になってしまうにもかかわらず、です）。オブジェクトプーリングの主な利点は、事前に割り当てられたプールからオブジェクトを効率的に再利用することで、オブジェクトの頻繁な生成と割り当て解除を不要にすることです。
- Unity エディターでは、C# コードは通常、**Mono コンパイラーでマシンコードにコンパイルされます。**スタンドアロンビルドの場合、一般的に IL2CPP（C# 中間言語を C++ にクロスコンパイルしたもの）を使えばより良い結果が得られますが、ビルドの時間が長くなったり、MOD のサポート が難しくなるなどのデメリットが存在します。
- Unity では、簡単に**すべてのコードをメインスレッドで実行**できるため、Unity プロジェクトではこの手法を取るのが一般的です。
    - MonoBehaviour の Update() メソッドなど、Unity のイベント関数はすべてメインスレッドで実行されます。
    - ほとんどの Unity API は、メインスレッドからしか安全に呼び出すことができません。
- 一般的な Unity プロジェクトのデータは、**ランダムなオブジェクトがメモリ全体に散在する**構造になる傾向があり、キャッシュの利用効率が悪くなります。繰り返しますが、これは Unity によって簡単にきてしまうためでもあります。
    - ゲームオブジェクトとそのコンポーネントはすべて個別に割り当てられるため、メモリ上の別々の場所に保存されることが多くなります。
- **一般的な Unity プロジェクトのコードは、キャッシュフレンドリーではない傾向があります。**
    - 従来の C# と Unity の API は、オブジェクト指向のコードスタイルを推奨しており、多数の細分化されたメソッドや複雑な呼び出しチェーンが生じやすくなります。データ指向のアプローチとは異なり、ハードウェアにはあまり優しくありません。
    - 各 MonoBehaviour のイベント関数は個別に呼び出されますが、その呼び出しは必ずしも MonoBehaviour のタイプごとにグループ化されているわけではありません。例えば、**Monster** の MonoBehaviour が 1000 個あった場合、各 Monster は別々に更新され、必ずしも他の Monster と一緒に更新されるわけではありません。
- 従来の C# や多くの Unity API に見られるオブジェクト指向のスタイルは、一般的に**抽象度が高いソリューション**につながります。この結果として生成されたコードには非効率的な部分が散見され、それを切り離すことは難しいでしょう。

上記の問題のさらなる背景については、以下の概念を網羅した本ガイド巻末の付録をご参照ください。

— メモリの割り当てとガベージコレクション

— マルチスレッドプログラミング

— メモリと CPU キャッシュ

— オブジェクト指向プログラミングと抽象化

— データ指向設計

DOTS と Entity Component システム

Unity の Entity Component System（ECS）は、DOTS のパッケージとテクノロジーを支えるデータ指向アーキテクチャです。ECS は、Unity において、メモリ上のデータとランタイムプロセススケジューリングに対する大まかなコントロールと決定性を提供します。

ECS for Unity 2022 LTS には、2 つの互換性のある物理エンジン、高レベルの Netcode パッケージ、Unity のスクリプタブルレンダーパイプライン（SRP）（ユニバーサルレンダーパイプライン（URP）や HD レンダーパイプライン（HDRP）を含む）に大量の ECS データをレンダリングするためのレンダリングフレームワークが付属しています。ゲームオブジェクトデータと互換性があるため、アニメーション、ナビゲーション、入力、Terrain（地形）など、Unity 2022 LTS の時点では ECS をネイティブサポートしていないシステムであっても活用することができます。

このセクションでは、DOTS の機能、および前のセクションで概説した CPU パフォーマンスの落とし穴を回避するコードの記述を容易にする方法に焦点を当てています。

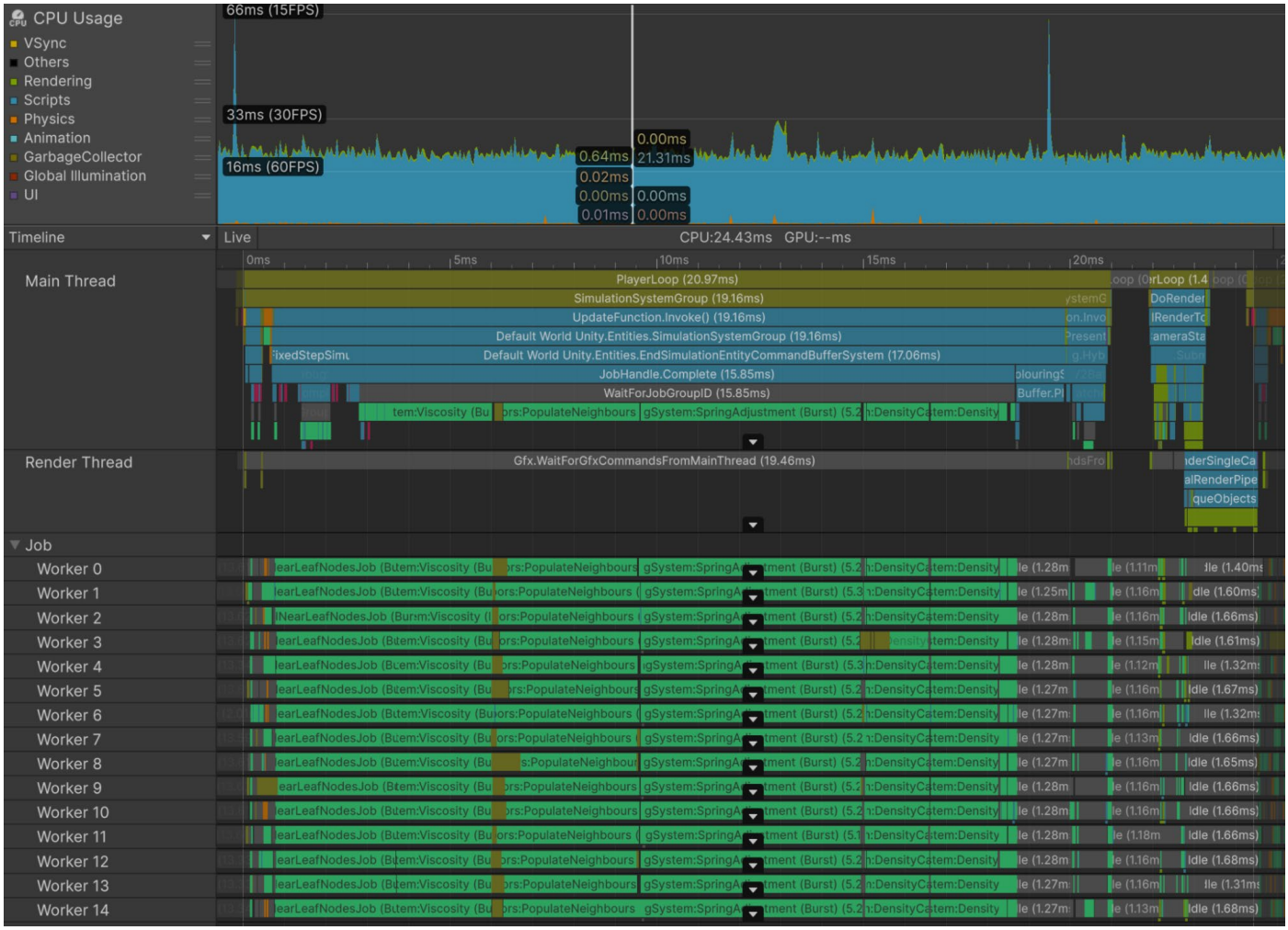
特定の DOTS パッケージについて学ぶための最良の出発点は EntityComponentSystemSamples の Github で、ここには説明文、ビデオ、そして多くのサンプルが掲載されています。

しかし、サンプルを詳しく見る前に、スタックを形成する機能やパッケージを確認してみましょう。

C# ジョブシステム

C# ジョブシステムは、アプリケーションが利用可能なすべての CPU コアを活用できるように、簡単かつ効率的にマルチスレッドコードを書く方法を提供します。

DOTS の他の機能とは異なり、このジョブシステムはパッケージではなく、Unity のコアモジュールに含まれています。



Burst コンパイルされたジョブが CPU の潜在能力を活用し、多くのワーカースレッドで実行されていることを示すプロフィール。

MonoBehaviour の更新はメインスレッドでのみ実行されるため、多くの Unity ゲームは、1 つの CPU コア上ですべてのゲームロジックを実行することになります。追加コアを活用するために手動でスレッドの追加や管理を行うことも可能ですが、安全かつ効率的にそれを行うのは非常に難しい場合があります。

Unity は、より簡単な手段として、C# ジョブシステムを提供しています。

— ジョブシステムは、ターゲットプラットフォームの追加コアごとに 1 つのワーカースレッドを持つスレッドプールを維持します。例えば、Unity が 8 コアで動作する場合、1 つのメインスレッドと 7 つのワーカースレッドが作成されます。

— その後、ワーカースレッドは、ジョブと呼ばれる作業単位を実行します。ワーカースレッドがアイドル状態になると、次に実行可能なジョブをジョブキューから取り出して実行します。

— ワーカースレッド上でジョブの実行が開始されると、完了するまで実行され続けます（つまり、ジョブがプリエンプトされることはありません）。

```
// 2 つの配列の要素を
// 乗算する簡単な例。
// IJob を実装すると、この構造体はジョブ型になります。
struct MyJob : IJob
{
    // NativeArray は「アンマネージ」型であり、
    // ガベージコレクションされません。
    public NativeArray<float> Input;
    public NativeArray<float> Output;
    // Execute メソッドは、
    // ジョブシステムがこのジョブを実行するときに呼び出されます。
    public void Execute()
    {
        // Output のすべての値を
        // 入力配列の対応する値で掛け算します。
        for (int i = 0; i < Input.Length; i++)
        {
            Output[i] *= Input[i];
        }
    }
```

ジョブのスケジューリングと完了

— ジョブのスケジューリング（ジョブキューへの追加）は、メインスレッドからのみ実行可能で、他のジョブからはできません。

— メインスレッドがスケジュールされたジョブの `Complete()` メソッドを呼び出すと、ジョブの実行が終了するのを待ちます（まだ終了していない場合）。

— `Complete()` の呼び出しを行えるのはメインスレッドのみです。

— `Complete()` の呼び出しから戻った後は、ジョブが使用するデータが再びメインスレッド上で安全にアクセスできるようになり、その後にスケジュールされるジョブに安全に渡すことができます。

ジョブのセーフティチェックと依存関係

マルチスレッドプログラミングでは、スレッド間の安全性を確保し、依存関係を管理することが、競合状態、データ破損、その他の並行性の問題を回避するためにも重要です。これらの落とし穴の詳細について説明することは、このガイドの範囲外となります。ここで大切なのは、ジョブシステムがセーフティチェックと依存関係をどのように処理するかを理解することです。

— 独立性を保つため、それぞれのジョブには、メインスレッドや他のジョブがアクセスできない独自のプライベートデータが存在します。

- しかし、ジョブ同士、またはジョブとメインスレッド間でデータ共有が必要な場合もあります。競合状態が発生するため、同じデータを共有するジョブは同時に実行するべきではありません。そのため、他のジョブと競合する可能性のあるジョブをスケジュールすると、ジョブシステムの「セーフティチェック」がエラーを返します。
- ジョブをスケジュールする際、そのジョブが先行してスケジュールされたジョブに依存することを宣言できます。ワーカースレッドは、その依存関係がすべて終了するまでジョブの実行を開始しないので、そうでなければ競合してしまうようなジョブであっても安全にスケジューリング可能です。
    - 例えば、ジョブ A とジョブ B が同じ配列にアクセスする場合、ジョブ B をジョブ A に依存させることができます。これにより、ジョブ A が完了するまでジョブ B が実行されなくなるため、競合の可能性を回避できます。
- ジョブの完了は、そのジョブが直接的、または間接的に依存しているすべてのジョブの完了も意味します。

Unity の多くの機能は内部でジョブシステムを使用しているため、プロファイラーでは、ワーカースレッドで実行されている自分のスケジュール済みのジョブ以外のものも表示されます。

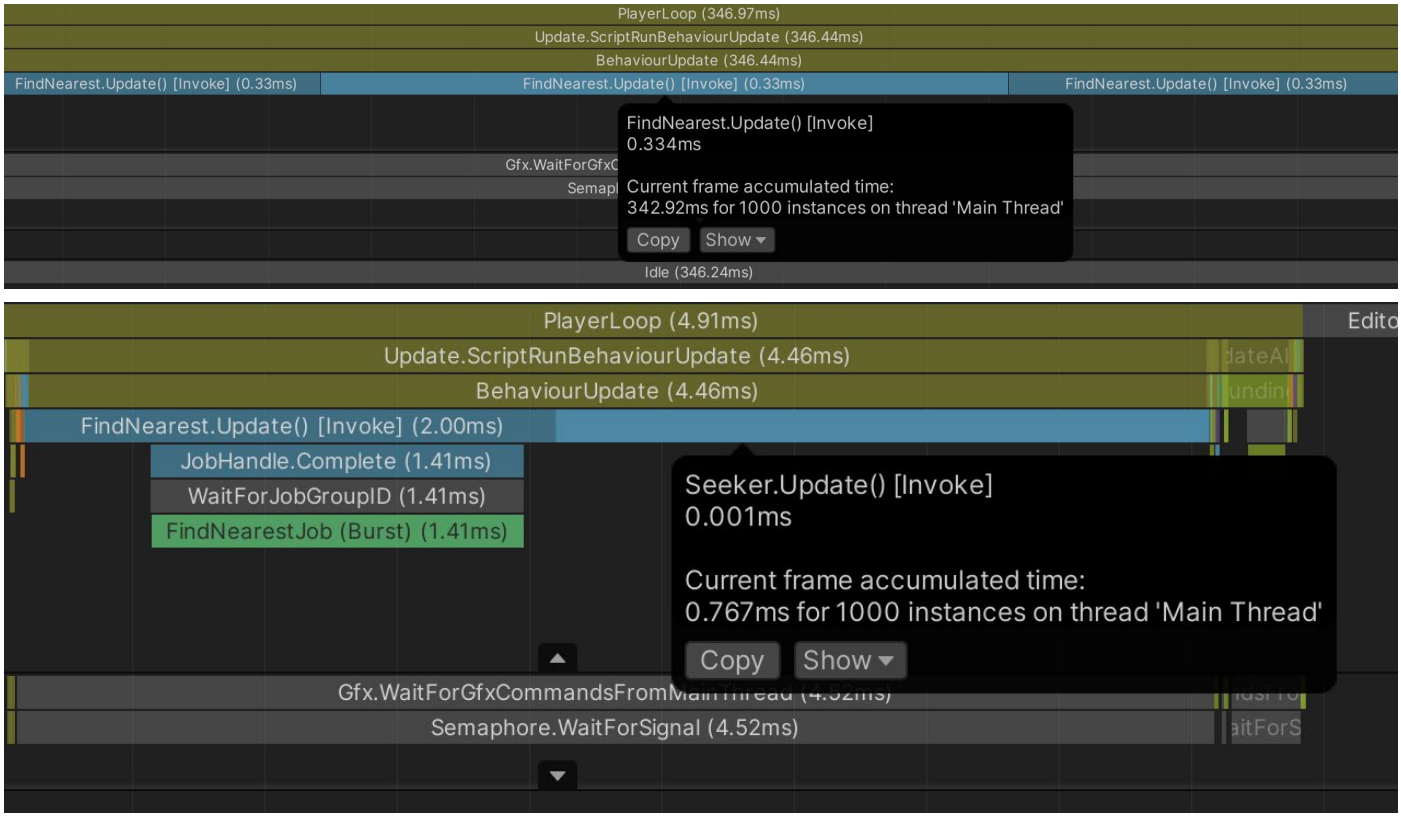
ジョブはメモリ上のデータ処理のみを目的としたものであり、ファイルの読み書きやネットワーク接続を介したデータの送受信といった I/O（入出力）操作の実行を目的としたものではないことに注意してください。一部の I/O 操作は呼び出しスレッドをブロックする可能性があるため、ジョブの中で実行してしまうと、CPU コアの利用率を最大化しようとする目標が達成できなくなります。マルチスレッドの I/O 処理を実行したい場合、メインスレッドから非同期 API を呼び出すか、従来の C# マルチスレッドを使用する必要があります。

ジョブについて学ぶには、最初にサンプルリポジトリのジョブのチュートリアルをご覧ください（Unity Learn にもチュートリアルがあります）。

Burst コンパイラー

前述したように、Unity の C# コードはデフォルトで JIT（ジャストインタイム）コンパイラーである Mono でコンパイルされるか、または代わりに AOT（先読み）コンパイラーである IL2CPP でコンパイルされます。一般的には AOT コンパイラーの方がランタイムパフォーマンスが優れており、ターゲットプラットフォームによってはより互換性が高くなっています。

Burst パッケージは、実質的な最適化を行い、多くの場合、Mono や IL2CPP よりも優れたパフォーマンスをもたらす第 3 のコンパイラを提供します。以下の画像が示すとおり、Burst を使用することで、重い計算作業のパフォーマンスやスケーラビリティを大幅に向上させることができます。



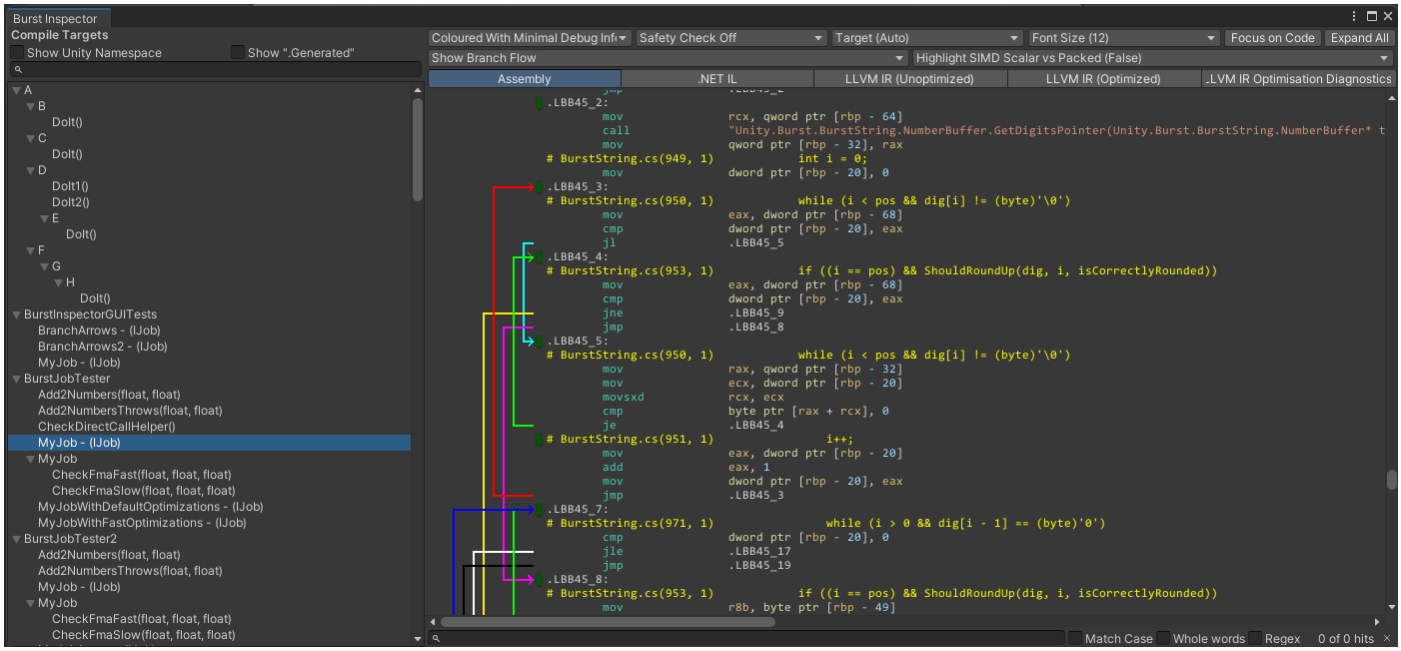
画像（上）：Jobs チュートリアルの、Mono でコンパイルされた FindNearest の更新には 342.9 ms かかります。
画像（下）：同じ Jobs チュートリアルの、Burst でコンパイルされた FindNearestJob の更新には 1.4 ms かかります。

ただし、Burst がコンパイルできるのは C# のサブセットのみであり、一般的な C# コードの多くは Burst ではコンパイルできないことをご理解ください。主な制限として、Burst コンパイルされたコードは、すべてのクラスインスタンスを含むマネージオブジェクトにアクセスできないことが挙げられます。従来の C# コードの大半は除外されるため、Burst コンパイルはジョブなどの指定されたコード部分にのみ選択的に適用されます。

```
// BurstCompile 属性は、このジョブがバーストコンパイルされることを示します。
[BurstCompile]
struct MyJob : IJob
{
    public NativeArray<float> Input;
    public NativeArray<float> Output;

    public void Execute()
    {
        for (int i = 0; i < Input.Length; i++)
        {
            Output[i] *= Input[i];
        }
    }
}
```

この動画で説明されているように、Burst の性能向上は、SIMD（複数のデータ要素に対して同時に同じ処理を実行するために使用される技術）の使用と、エイリアシング（2 つ以上のポインターまたは参照が同じメモリ位置を参照する場合）の考慮、およびその他の技術によりもたらされています。



Burst は、intrinsic や、生成されたアセンブリコードを表示する Burst Inspector（画像上）など、いくつかの上級者向けの機能を提供しています。

Collections

Collections パッケージは、ジョブや Burst コンパイルされたコードで使用するために最適化されたリストやハッシュマップのような、アンマネージコレクションタイプを提供します。

ここで言う「アンマネージ」とは、これらのコレクションが C# ランタイムやガベージコレクターによって管理されていないことを指します。作成したアンマネージコレクションが不要になった際は、`Dispose()` メソッドを呼び出して明示的に破棄しなくてはなりません。

これらのコレクションはアンマネージであるため、ガベージコレクションの圧力が発生せず、ジョブや Burst コンパイルされたコードで安全に使用することができます。

コレクションの型は、いくつかのカテゴリーに分類されます。

- 名前が **Native** で始まる型はセーフティチェックを行います。これらのセーフティチェックは、以下の場合、エラーを発行します。
  - コレクションが正しく破棄されていない場合。
  - コレクションがスレッドセーフではない形でジョブと共に使用された場合。

— 名前が **Unsafe** で始まる型はセーフティチェックを行いません。

— **Native** でも **Unsafe** でもない残りの型は、ポインターを持たない小さな構造体であるため、割り当ては一切行われません。その結果、破棄の必要もなく、スレッドセーフティの潜在的な問題もありません。

いくつかの **Native** 型に相当するものは、**Unsafe** にも存在します。例えば、**NativeList** と **UnsafeList**、**NativeHashMap** と **UnsafeHashMap** など、さまざまなペアがあります。安全のため、**Unsafe** の同等の型よりも、**Native** のコレクションの使用を優先することをおすすめします。

Mathematics

Mathematics パッケージは C# の数学ライブラリで、Collections と同様、Burst とジョブシステムが、C#/IL コードを非常に効率的なネイティブコードにコンパイルできるよう作成されています。このパッケージは、以下を提供しています。

— **float3**、**クォータニオン**、**float3x3** などのベクトル型や行列型。

— HLSL のようなシェーダーの規則に従った、多くの数学メソッドや演算子。

— 多くのメソッドと演算子に対する、特別な Burst コンパイラーの最適化フック。

詳しくは Unity.Mathematics のチートシートをご覧ください。

以前の **UnityEngine.Mathf** ライブラリのほとんどの型とメソッドは、Burst コンパイルされたコードでも使用できますが、場合によっては **Unity.Mathematics** で提供されているものの方がより良いパフォーマンスを発揮することにご注意ください。

Entities（ECS）

Entities パッケージは ECS の実装を提供します。これはデータ用の**エンティティ**と**コンポーネント**、コード用の**システム**で構成されるアーキテクチャパターンです。

簡単に言うと、エンティティはコンポーネントで構成され、各コンポーネントは通常、C# の構造体です。ゲームオブジェクトと同様に、エンティティのコンポーネントは、その生存期間中に追加したり削除したりできます。

ただ、ゲームオブジェクトとは異なり、エンティティのコンポーネントは通常、独自のメソッドを持ちません。その代わりに、ECS では、各「システム」には通常 1 フレームに 1 回呼び出される更新メソッドが備えられており、これらの更新はいくつかのエンティティのコンポーネントの読み込みと変更を行います。例えば、モンスターが登場するゲームでは MonsterMoveSystem があり、その更新メソッドが各モンスターのエンティティの Transform コンポーネントを変更するかもしれません。

アーキタイプ

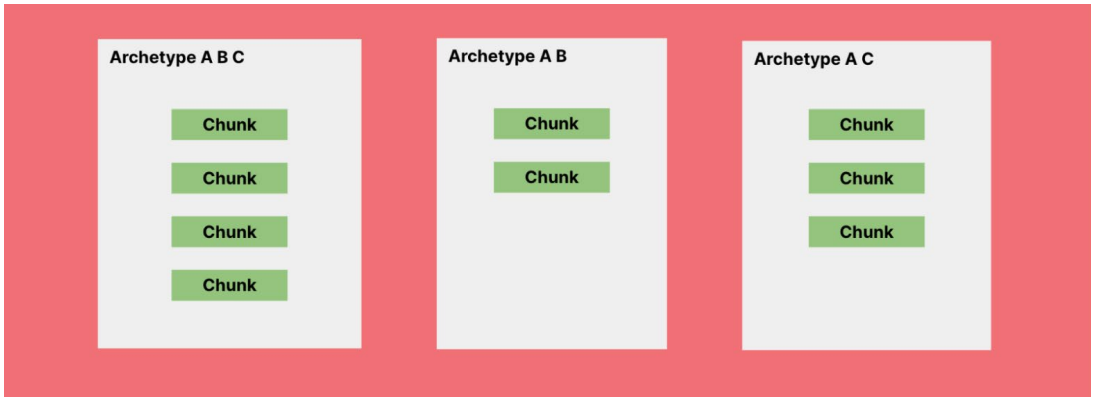
Unity の ECS では、同じコンポーネントタイプのセットを持つエンティティはすべて同じ「アーキタイプ」にまとめて保存されます。例えば、A、B、C の 3 つのコンポーネントタイプがあるとしましょう。各コンポーネントタイプの一意の組み合わせは、それぞれ別のアーキタイプになります。

— 例えば、コンポーネントタイプ A、B、C を持つエンティティは、それぞれ 1 つのアーキタイプに保存されます。

— コンポーネントタイプ A、B を持つエンティティは、共に 2 つ目のアーキタイプに保存されます。

— コンポーネントタイプ A、C を持つエンティティは、3 つ目のアーキタイプに保存されます。

エンティティにコンポーネントを追加したり、エンティティからコンポーネントを削除したりすると、エンティティは別のアーキタイプに移されます。



Unity の ECS では、同じコンポーネントタイプのセットを持つエンティティはすべて同じ「アーキタイプ」にまとめて保存されます。

チャンク

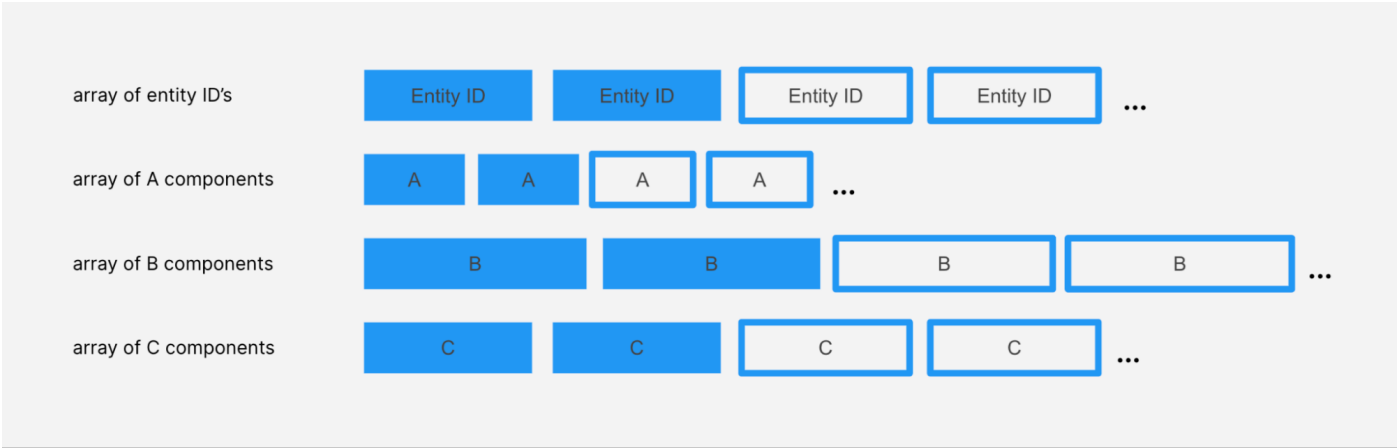
アーキタイプの中で、エンティティとそのコンポーネントは、チャンクと呼ばれるメモリのブロックに格納されます。それぞれのチャンクには最大 128 個のエンティティが格納され、各タイプのコンポーネントはチャンク内の独自の配列に格納されます。例えば、コンポーネントタイプ A と B を持つエンティティのアーキタイプでは、各チャンクに 3 つの配列が格納されます。

— エンティティ ID 用の配列 1 つ

— A コンポーネント用の配列 1 つ

— B コンポーネント用の配列 1 つ

チャンク内の最初のエンティティの ID とコンポーネントは、これらの配列のインデックス 0 に、2 番目のエンティティはインデックス 1 に、3 番目のエンティティはインデックス 2 に、といった具合に格納されます。



Unity の ECS アーキテクチャにおけるチャンクの仕組み

チャンクの配列は常に密に保たれています。

— 新しいエンティティがチャンクに追加されると、そのエンティティは配列の最初の空きインデックスに格納されます。

— エンティティがチャンクから削除されると、チャンク内の最後のエンティティが空いた箇所に移されます（エンティティがチャンクから削除されるのは、破壊される時、または他のアーキタイプに移される時です）。

クエリ

アーキタイプとチャンクベースのデータレイアウトの主な利点は、エンティティの効率的なクエリとイテレーションを可能にすることです。

特定のコンポーネントタイプのセットを持つすべてのエンティティをループ処理するために、エンティティクエリはまず、その条件に一致するすべてのアーキタイプを見つけ、次にアーキタイプのチャンク内のエンティティを反復処理します。

— チャンク内のコンポーネントは、要素間に隙間のない配列に格納されているため、コンポーネントの値をループすることで、多くのキャッシュミスを回避できます。

— アーキタイプのセットはプログラムの大部分を通して安定している傾向があるため、通常、クエリにマッチするアーキタイプのセットをキャッシュすることで、クエリを高速化することが可能です。

```
// シンプルなシステム例。
public partial struct MonsterMoveSystem : ISystem
{
    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // LocalTransform、Velocity、Monster コンポーネントを持つ
        // すべてのエンティティをループするクエリ
        foreach (var (transform, velocity) in
            SystemAPI.Query<RefRW<LocalTransform>, RefRO<Velocity>>()
                .WithAll<Monster>())
        {
            // 速度から変換位置を更新します
            //（デルタ時間を考慮）
            transform.ValueRW.Position +=
                    velocity.ValueRO.Value * SystemAPI.Time.deltaTime;
        }
    }
}
```

ジョブシステムインテグレーション

エンティティコンポーネントは、アンマネージタイプである限り、Burst コンパイルされたジョブでアクセスできます。エンティティにアクセスするために、IJobChunk と IJobEntity という 2 つの特別なジョブタイプが用意されています。

```
// IJobEntity をスケジュールするシンプルな例システムです。
public partial struct MonsterMoveSystem : ISystem
{
    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // ジョブの作成とスケジュールを行います。
        var job = new MonsterMoveJob {
            DeltaTime = SystemAPI.Time.DeltaTime
        };
        job.ScheduleParallel();
    }
}
// LocalTransform、Velocity、Monster コンポーネントを持つ
// すべてのエンティティを処理するバーストコンパイルされたジョブです。
[WithAll(typeof(Monster))]
[BurstCompile]
public partial struct MonsterMoveJob : IJobEntity
{
    public float DeltaTime;
    // LocalTransform を変更したいので、「ref」を使用します。
    // Velocity のみを読み込みたいので、「in」を使用します。
    public void Execute(ref LocalTransform, in Velocity)
    {
        transform.Position += velocity.Value * DeltaTime;
    }
}
```

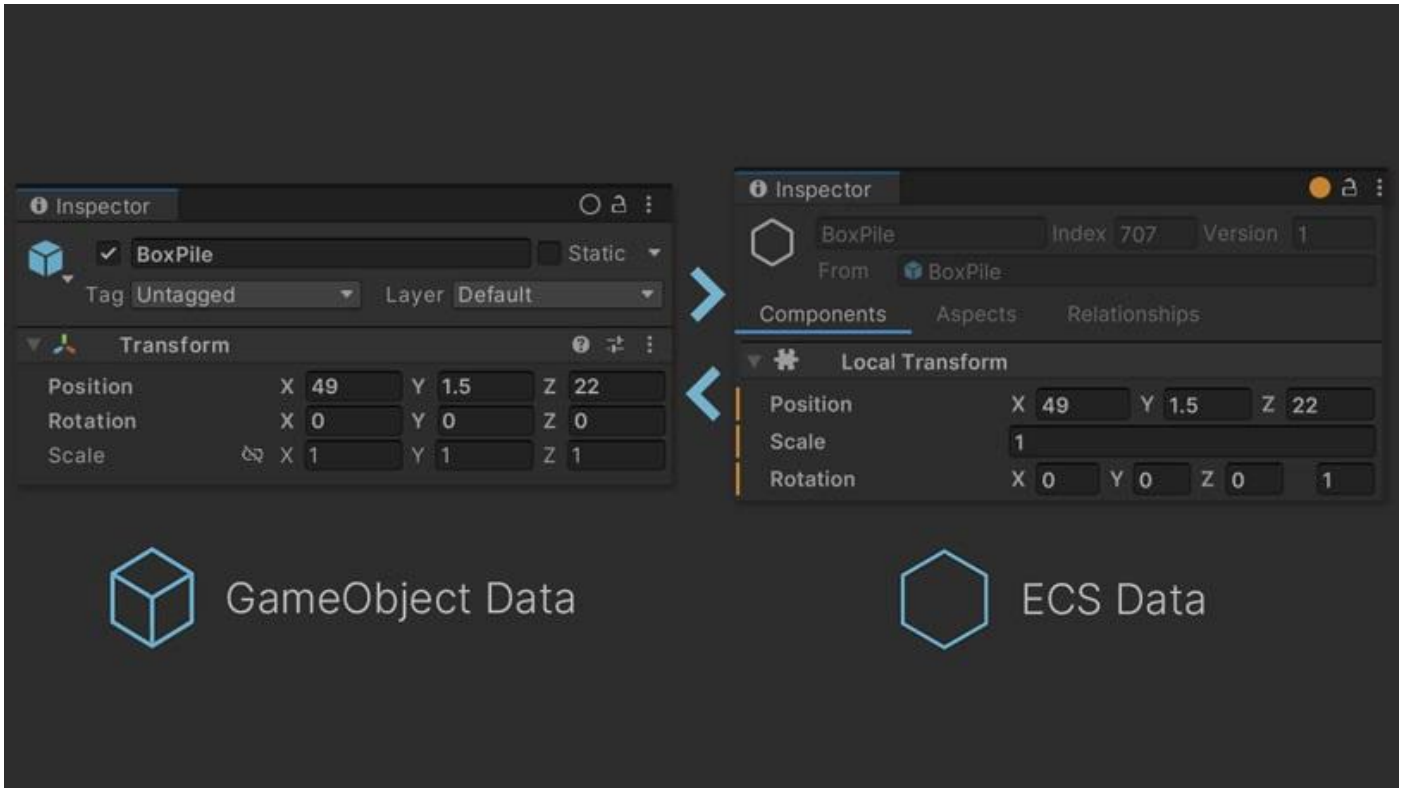
使いやすさ向上のため、システムはシステム間のジョブの依存関係やジョブの完了を自動的に処理することができるようになっています。

サブシーンとベイキング

Unity ECS は、アプリケーションのコンテンツを管理するのに、シーンではなくサブシーンを使用します。理由としては、Unity のコアシーンシステムに ECS との互換性がないためです。

Unity のシーンにエンティティを直接含めることはできませんが、ベイキングと呼ばれる機能を使用してシーンからエンティティを読み込み、ゲームオブジェクトや MonoBehaviour コンポーネントをエンティティと ECS コンポーネントに変換することは可能です。

サブシーンは、他のシーンの中にネストされ、編集されるたびにベイキング処理が再実行されるシーンとして考えてください。ベイキングは、サブシーン内の各ゲームオブジェクトに対するエンティティを作成し、これらのエンティティはファイルにシリアライズされます。サブシーンがロードされた場合、ランタイムにロードされるのはこれらのエンティティで、ゲームオブジェクトそのものではありません。



左：ゲームオブジェクト。右：ゲームオブジェクトからベイクされたエンティティ。

どのコンポーネントがベイクされたエンティティに追加されるかは、ゲームオブジェクトコンポーネントに関連付けられた「ベイカー」によって決定されます。例えば、MeshRenderer のような標準的なグラフィックスコンポーネントに関連するベイカーは、グラフィックス関連のコンポーネントをエンティティに追加します。独自の MonoBehaviour タイプでは、ベイカーを定義して、どのコンポーネントがベイクするエンティティに追加されるかを制御することができます。

```
// このエンティティコンポーネントタイプは、ヒットポイント、
// 最大ヒットポイント、リチャージ遅延、リチャージ速度を持つエネルギーシールドを表します。
public struct EnergyShield : IComponentData
{
    public int HitPoints;
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;
}
// シンプルなコンポーネント作成例。
// オーサリングコンポーネントは、定義済みの Baker クラスを持つ通常の MonoBehaviour
// です。
public class EnergyShieldAuthoring : MonoBehaviour
{
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;
    // EnergyShield オーサリングコンポーネントのベイカーです。
    // このベイカーは、サブシーン内の任意のゲームオブジェクトにアタッチされた
    // EnergyShieldAuthoring インスタンスごとに 1 回実行されます。
    class Baker : Baker<EnergyShieldAuthoring>
    {
        public override void Bake(EnergyShieldAuthoring authoring)
        {
            // TransformUsageFlags は、
            // エンティティが持つべきトランスフォームコンポーネントを指定します。
            // None フラグは、トランスフォームを必要としないことを意味します。
            var entity = GetEntity(TransformUsageFlags.None);
             // このシンプルなベイカーは、エンティティにコンポーネントを 1 つだけ
                追加します。
             AddComponent(entity, new EnergyShield
             {
                 HitPoints = authoring.MaxHitPoints,
                 MaxHitPoints = authoring.MaxHitPoints,
                 RechargeDelay = authoring.RechargeDelay,
                 RechargeRate = authoring.RechargeRate,
             });
         }
     }
}
```

単純なケースにおいてはシーンにエンティティを直接追加できないのは不便ですが、より高度なケースではベイク処理が有用な場合があります。ベイクは、オーサリングデータ（エディターで編集するゲームオブジェクト）とランタイムデータ（ベイクされたエンティティ）を実質的に分離するため、直接編集するデータとランタイムで読み込まれるデータが 1 対 1 で一致する必要もなくなります。例えば、ベイク中にデータを連続的に生成するコードを書けば、ランタイムでのコストがなくなります。

ストリーミング

特に大規模で詳細な環境においては、プレイヤーやカメラが環境を動き回るのに合わせて、多くの要素を非同期で効率的にロードおよびアンロードできることが重要です。例えば、大きなオープンワールドでは、多くの要素が視界に入るときにロードされたり、視界から外れるときにアンロードされなくてはなりません。この手法はストリーミングとも呼ばれます。

ゲームオブジェクトよりもエンティティの方がストリーミングに遥かに向いています。何故なら、エンティティはメモリ使用量や処理のオーバーヘッドが少なく、シリアライズやデシリアライズも効率的に行えるからです。

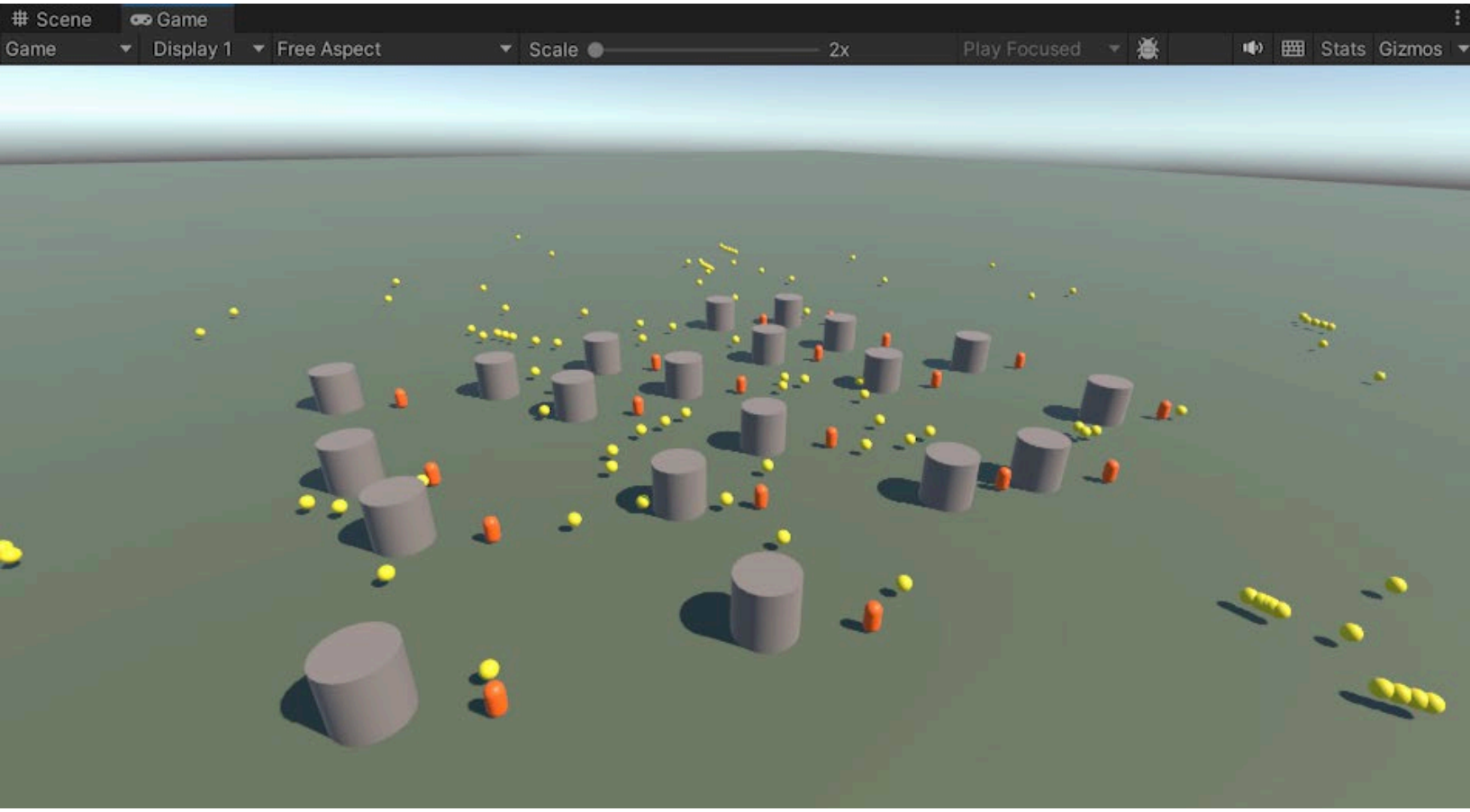
EntityComponentSystemSamples Github

「EntityComponentSystemSamples」Github リポジトリには、基本的な DOTS 機能と高度な DOTS 機能の両方を紹介する多くのサンプルが含まれています。詳細は各サンプルコレクションの Readme ファイルに記載されていますが、ここではいくつかの厳選されたサンプルについて簡単に説明します。

Github リポジトリにあるサンプルの一部は、Unity Learn の DOTS に関する新しいコース「Get acquainted with DOTS」で再現されています。各サンプルの下にある、Unity Learn のチュートリアルへのリンクを参照してください（該当する場合）。

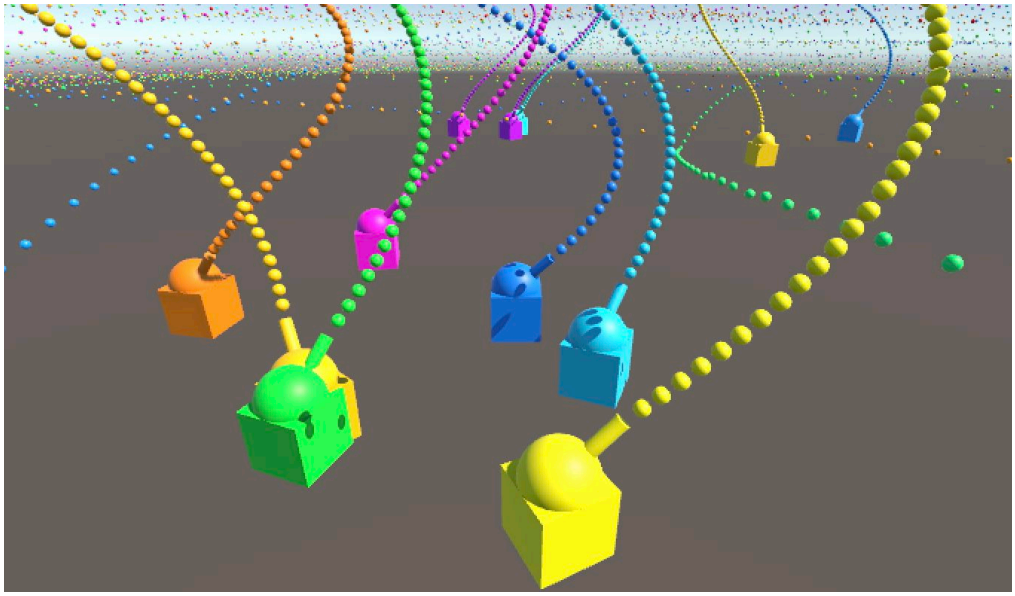
HelloCube サンプルは、DOTS 初心者に最適な入門であり、エンティティの作成と破棄、コンポーネントの追加と削除、エンティティにアクセスするシステムなど、Entities API の最も基本的な概念を学ぶことができます。サンプルの内容の詳細については、HelloCube のウォークスルービデオをご覧いただくか、Unity Learn のステップバイステップチュートリアル、HelloCube チュートリアルをご覧ください。

ベイキングサンプル は、ビルド時にエンティティデータをシリアライズし、ランタイムでシーンからロードされるベイキングのプロセスを紹介しています。



『Kickball』チュートリアルのスクリーンショット

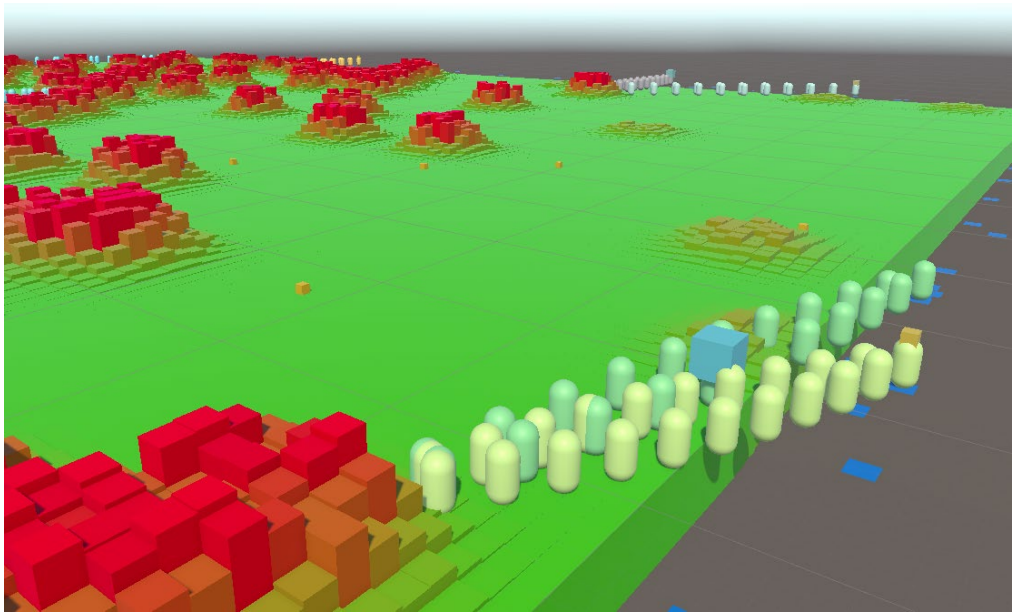
Kickball サンプルは、HelloCube よりも多く Entities の基本的な使い方を示しており、さらなるゲーム性があります。コントロールを使用することで、オレンジ色のカプセルを動かしたり、黄色いボールをスポーンしたり、ボールを蹴ってカプセルから遠ざけたりできます。灰色のシリンダーは、カプセルとボールの動きを妨げる障害物です。Kickball のウォークスルービデオもご覧ください。



『Tanks』チュートリアルのスクリーンショット

Tanks サンプルでは、ジョブシステムとエンティティの両方が活用されており、戦車が平面上を移動し、砲塔を回転させ、色のついた弾丸を発射します。プレイヤーの目的は、回転する砲塔から砲弾を発射する、動く戦車をスポーンさせることです。戦車は砲弾がヒットすると破壊され、プレイヤーは 1 両の戦車の動きをコントロールします。Tanks のウォークスルービデオをチェックするか、Unity Learn の Tanks チュートリアルを観ながら一緒に取り組んでみてください。

Firefighters サンプルでは、野原に火が燃え広がっており、ボットがバケツリレーを組んで消火活動を行います。このチュートリアルは上級者向けのもので、多くのコンセプトを応用しているので、上記のチュートリアルを先に行うことをおすすめします。このプロジェクトは、DOTS ブートキャンプの一環として 4 つのセッションで取り上げられました。

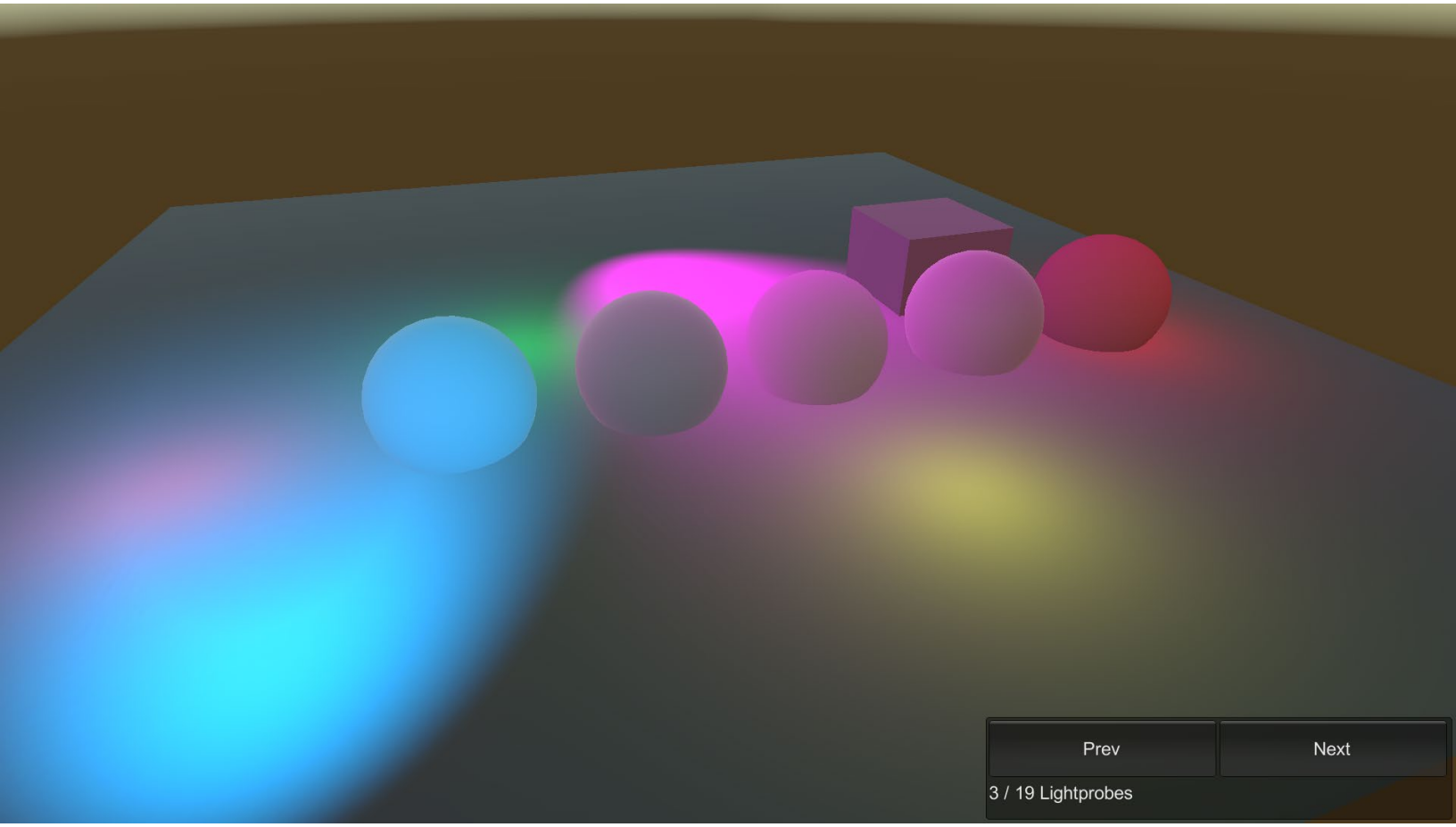


『Firefighters』サンプルのスクリーンショット

Entities Graphics

Entities Graphics パッケージは、ユニバーサルレンダーパイプライン（URP）または HD レンダーパイプライン（HDRP）を介してエンティティをレンダリングするためのコンポーネントとシステムを提供します。Entities Graphics は、BatchRendererGroup API を中心に構築されています。

Entities Graphics サンプルは、ライトプローブやライトマップ、マテリアルプロパティのオーバーライド、LOD など、さまざまなグラフィックス機能のデモンストレーションを行います。



EntityComponentSystemSamples リポジトリの Entites.Graphics サンプルシーン

物理演算

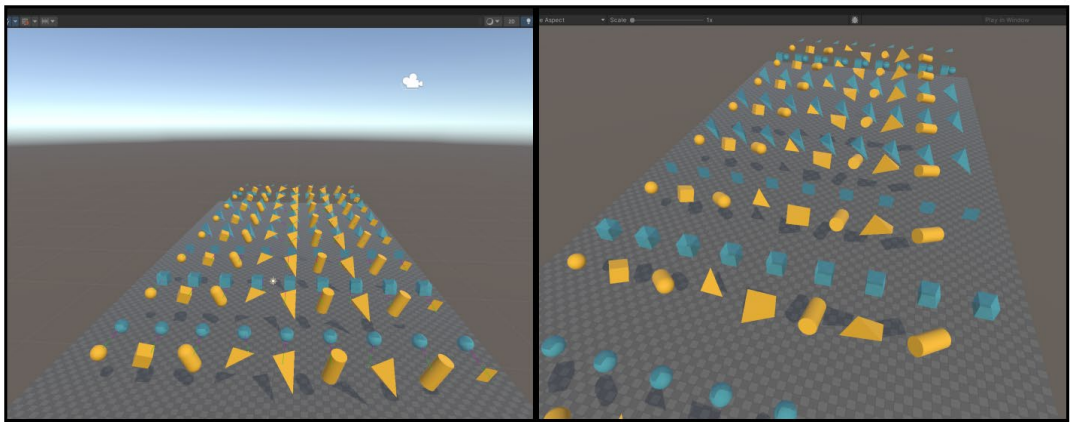
Unity Physics パッケージは、リジッドボディシミュレーションと衝突チェックを提供します。

Unity Physics では、同じサーフェスレベルの API を維持しながら、代替の「バックエンド」に変更でき、独自のコードやアセットを編集することなく物理演算の実装を変更することができます。

パッケージで提供されるデフォルトのバックエンドは決定論的であり、同じ初期条件と入力が与えられれば同じ結果が得られます。

Havok Physics パッケージは、業界をリードする AAA ゲームの多くに採用されているプロプライエタリ Havok Physics エンジンをベースとした代替バックエンドを提供します。

Physics サンプルは、コライダー、質量とモーションのプロパティ、マテリアルプロパティ、イベント、ジョイントやモーターなど、パッケージの多くの機能を説明しています。



EntityComponentSystemSamples リポジトリの Physics サンプルシーン

Netcode for Entities

Netcode for Entities パッケージは、Unity が提供する 2 つのネットコードソリューションのうちの 1 つです。もう 1 つのソリューションである Netcode for GameObjects とは異なり、Netcode for Entities は権威サーバーを使用しており、クライアントサイド予測をサポートしているため、ペースの速い対戦ゲームに適しています。

権威サーバー

ゲームの進行や状態の権限をプレイヤーのマシンに分散させるのではなく、権威サーバーがゲームシミュレーション全体を実行し、ゲーム内で起きていることを決定します。クライアントがプレイヤーの入力をサーバーに送り、サーバーがゲームシミュレーションを更新し、サーバーがゲーム状態の新しいスナップショットをクライアントに送り返します。これは、ネットワークゲームのロジックを実装する最も単純な方法であり、不正をする人にもっとも悪用されにくい方法でもあります。

クライアントサイド予測

サーバーから送信されたデータがクライアントに届くまでには時間がかかるため、クライアントが受け取る状態は常にサーバーのものより遅れています。ゲーム内の多くの要素に関しては、このラグは許容範囲かもしれませんが、プレイヤーキャラクターなどの一部の要素においては、このようなラグはゲーム体験を損ない、プレイを困難にすることがあります。

この問題はクライアントサイド予測によって解決できます。クライアントは、プレイヤーキャラクターなどの指定の要素について、1 秒先の状態を予測しようと試みます。これらの予測がサーバー上の状態と正確かつ十分に一致する場合、ゲームのユーザー体験はほぼラグがないものになるでしょう。

これら 2 つの重要な機能に加え、Netcode for Entities は Netcode for GameObjects よりも拡張性が高く、帯域幅を最適化するための優れた手段を提供します。

Netcode for Entities を初めて学ぶには、Netcode for Entities サンプルのリポジトリが便利です。これらのサンプルは、同期、接続フロー、Unity Physics とのインテグレーションなど、多くの基本的な機能および上級者向けの機能を紹介しています。まずは Networked Cube チュートリアルで以下を学びましょう。

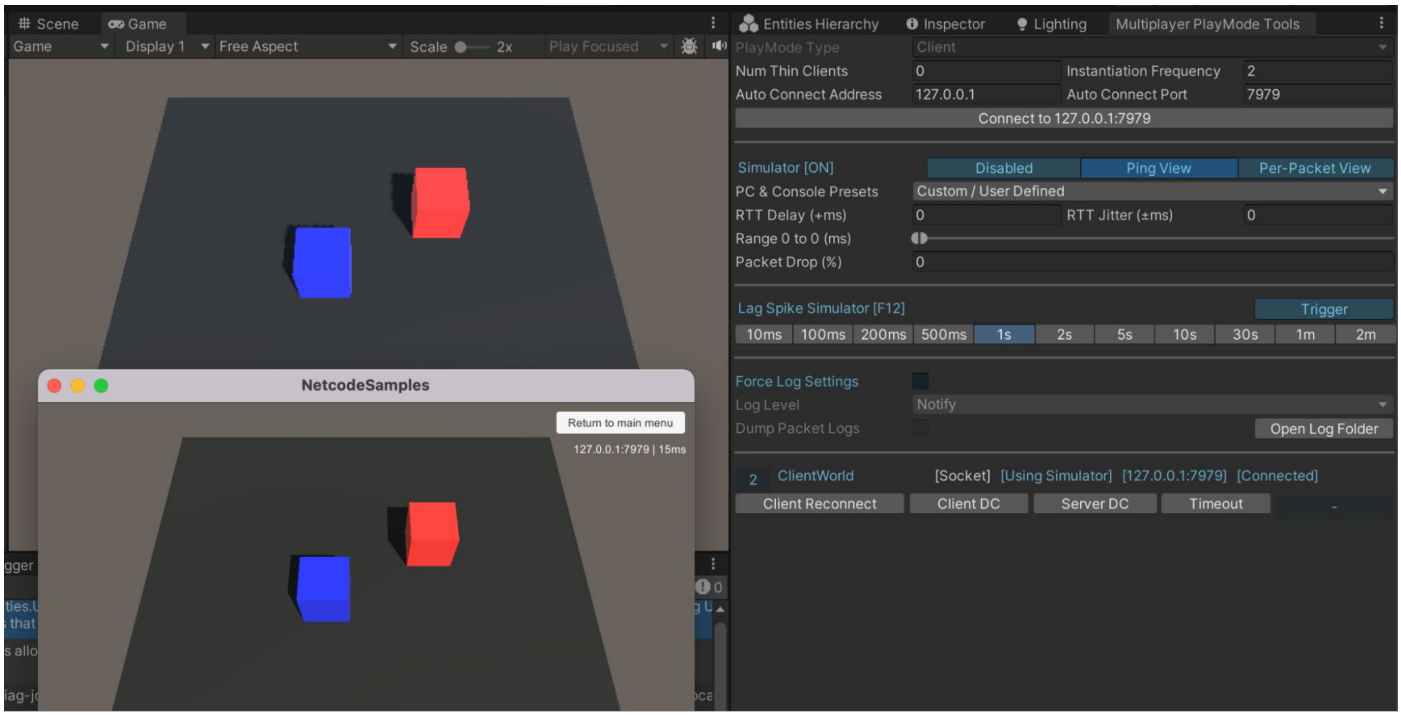
— サーバーとの接続の確立。

— サーバーとの通信。

— サーバーでの同期エンティティのスポーン。

— サーバーとクライアントのスタンドアロンビルドの作成。

— エディター内での再生モードでのサーバーとクライアントの実行。



エディター内、そしてスタンドアロンビルドとして起動中の『Networked Cube』チュートリアル

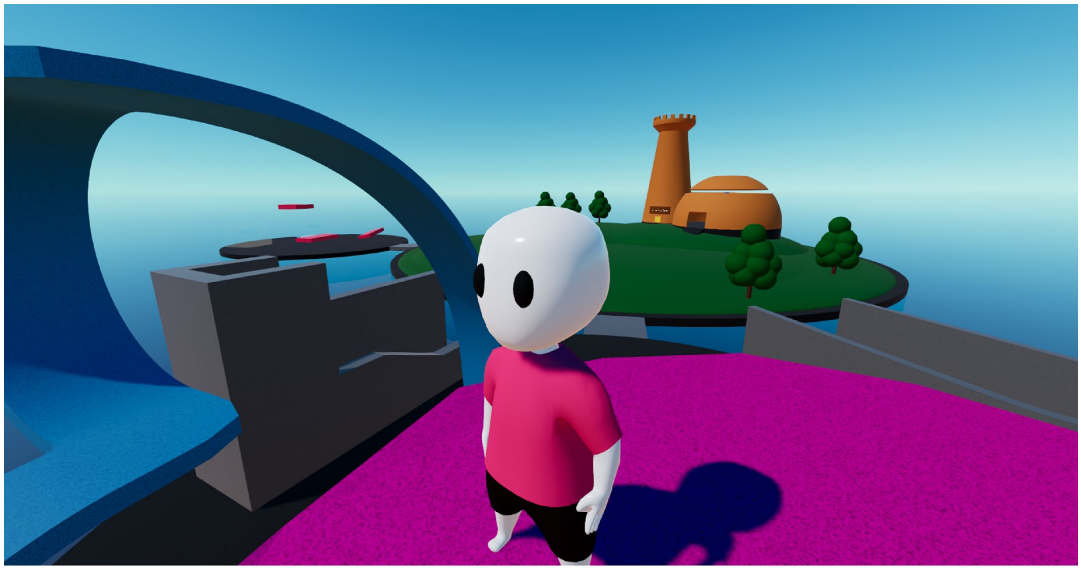
ECS Network Racing サンプル

カーレースのマルチプレイヤーメカニックを備えた『ECS Network Racing』サンプル

ECS Network Racing サンプルは、Unity Physics と Vivox ボイスチャットを使用したロビーベースのマルチプレイヤーカーレースサンプルです。

キャラクターコントローラー

CharacterController パッケージは、Unity Physics と Netcode for Entities で動作する一人称および三人称のキャラクターコントローラーの ECS ベースの実装を提供します。コントローラーは、スプリントやダブルジャンプなど、キャラクターの一般的な動作の多くをサポートしています。また、CharacterController のチュートリアルやサンプルを使用すると、より多くの例から学ぶことができます。



キャラクターコントローラーパッケージは Unity Asset Store よりアクセス可能です。

DOTS の今後のロードマップについて

DOTS の主要パッケージは本番環境で使用可能ですが、DOTS ベースの補完的なシステムの一部はまだ開発中です。本ガイドの執筆時点（2024 年春）における関連システムの開発状況は以下のとおりです。

アニメーション
Unity はエンティティと併用可能なスキンメッシュアニメーションシステムを開発していますが、本ガイドの執筆時点では、Unity の現バージョンでは未公開です。

現状、最も一般的な解決策は、アニメーションつきのキャラクターをゲームオブジェクトとしてレンダリングし、その Transform とアニメーションステートをエンティティから同期させることです。つまり、ゲームシミュレーションはエンティティで完全に実装されているものの、アニメーションつきのキャラクターの表示はゲームオブジェクトで行われます（このアイデアの簡単なデモは、サンプルリポジトリの「AnimateGameObject」サンプルをご覧ください）。この解決策は追加のコーディングを必要とし、オーバーヘッドも発生しますが、ほとんどのゲームで使用可能なはずです。

あるいは、独自のカスタムアニメーションソリューションを実装しているゲームメーカーやアセット開発者もいます。いくつかの異なるソリューションがコミュニティにより提供されており、その一部は Unity Asset Store で入手可能です。

ユーザーインターフェース

現在、ECS ベースの UI システムは存在しません。代わりに、ECS ベースのゲームでは、既存のゲームオブジェクトベースの UI ツールキットを使用することができます（UI ツールキットと ECS の簡単な連携例は、Firefighters チュートリアルにてご覧いただけます）。ゲームにおける UI Toolkit の使用についての詳細は、以下の上級者向けリソースよりご確認ください。

- Unity におけるユーザーインターフェースデザインと実装
- UI Toolkit サンプル – Dragon Crashers
- UI Toolkit サンプル – QuizU
- QuizU についての Discussions の記事シリーズ

Unity ECS と DOTS の最新計画については、DOTS のロードマップをご参照ください。

プロジェクトへの DOTS 導入検討

コードが CPU ボトルネックの原因になっている場合、Burst コンパイルジョブとしての再実装を検討することもできます。Burst コンパイルされたコードは、Mono や IL2CPP でコンパイルされた同等のコードよりも数倍高速に実行されることが多いだけでなく、ジョブによって CPU のすべてのコアにワークロードを分割することができます。

幸い、Burst コンパイルされたジョブは、プロジェクトのそれ以外の部分で DOTS が使用されていない場合でも、既存のプロジェクトの大部分に比較的簡単に統合できます。アンマネージコレクションにデータをコピーしたり取り出したりしなければならない可能性があることを除けば、多くの場合、既存のコードを Burst コンパイルジョブとして書き換える際、コードの大幅な再構築は必要ありません。

ただ、これは Entities パッケージにはあまり当てはまりません。特定の機能を実装するためのエンティティを選択的に統合することが可能な場合もありますが、ECS アーキテクチャはプロジェクト全体に独自のコード構造を課す傾向があります。

ここでは、Entities を使った新しいプロジェクトの構築を検討したほうが良い 4 つの例をご紹介します。

— プロジェクトに、大規模で詳細な環境のレンダリングなど、静的な要素が多く含まれる場合。オリジナルの Megacity プロジェクトは、エンティティを使用して構築された複雑な環境の一例です。

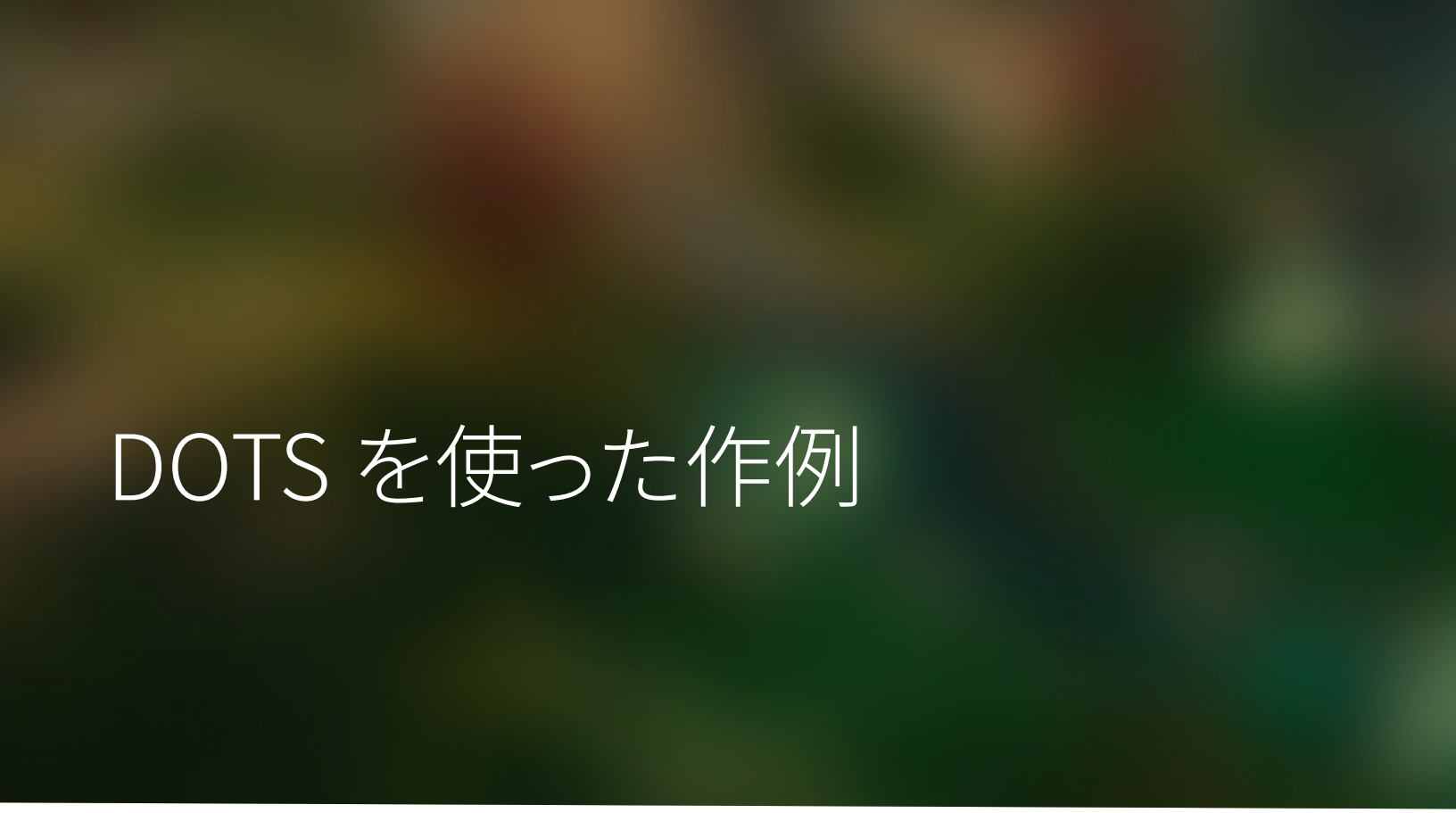
— プロジェクトに、計算負荷の高い挙動を持つ動的要素が多く含まれる場合。例えば、リアルタイムストラテジーゲームでは、多くの場合、何百、何千ものユニットの経路探索を計算する必要があります。

— より一般的なオブジェクト指向の手法よりも推論や保守が容易な、ECS でデータやコードを構造化する手法を好んでいる場合。少なくとも、ECS は、一般的にプロファイリングとボトルネックの特定を容易にします。

— プロジェクトが、シューティングゲームなどの素早いアクションを伴う対戦型マルチプレイヤーゲームであり、優れたプレイヤー体験のために権威サーバーとクライアントサイド予測が必要である場合（上記の通り、これらの機能は Netcode for Entities ではサポートされていますが、Netcode for GameObjects ではサポートされていません）。

一方、多くのゲームは主に GPU がボトルネックになっています。その場合、DOTS は CPU の効率を向上させるだけなので、Entities をはじめとした DOTS パッケージや関連技術はあまり役に立たないかもしれません。それでも、DOTS が同じ作業量をより少ない CPU 時間でこなすのに役立つのであれば、追加機能に余裕を与えることができ、将来的に低消費電力デバイスをターゲットにすることを決めた場合にも大いに役立ちます。

以下のセクションでは、Unity ECS と DOTS の技術を使用したゲームをいくつかご紹介します。



DOTS を使った作例

ここ数年、開発チームはマルチプラットフォームゲームに DOTS のパッケージやテクノロジーを使用するメリットを享受してきました。以下のカスタマーストーリーの抜粋からわかるように、各チームは DOTS の導入を決定する前に、自分たちのゲームが DOTS によってどのような恩恵を受けることができるかを慎重に考え抜いています。

Unity リソースハブでは、Unity のカスタマーストーリーやプロフィールをさらにご覧いただけます。

DOTS を使った作例：『Bare Butt Boxing』（開発元：Tuatara）

『Bare Butt Boxing』（Tuatara Games 作、Made with Unity、PC およびコンソールで配信中）

Tuatara Games は、開発の初期段階から Unity の DOTS を使用して『Bare Butt Boxing』を構築しました。ソフトウェアエンジニアの Hendrik du Toit 氏は、「これは、我々が新しいチームとして手がける最初のゲームだったので、デザインを正しい方向に転換することが可能な強固な基盤を持って早期アクセスに臨みたかったのです。DOTS のおかげで、何週間もかけてコードを書き換えなくてもゲームプレイのアイデアをテストできるように、システムをモジュール化することができました。」と述べています。

Tuatara のデータ指向設計手法は、イテレーションを単純化し、柔軟な最適化を可能にしました。「ECS があれば、シリアライズされたデータに影響を与えることなく、ランタイムのデータレイアウトを簡単に調整できます」と、ゲームプログラマーの Ewan Argouse 氏は言います。

「ECS のおかげで、問題なくゲームを複数のレイヤーに分けることができました。ゲームデザインはシンプルで、シミュレーションに直接関連させることができ、その上にシステムを作ってうまく見せることができます。おかげで、表示を複雑なものにしながら、CPU への負担を減らしつつクライアント予測が可能なシミュレーションを実現できました。」

- Tuatara Games ゲームプログラマー、Ewan Argouse 氏

DOTS を使った作例：『Histera』（開発元：StickyLock Games）

『Histera』（StickyLock Games 作、Made with Unity、Steam Early Access を通して PC で配信中）

『Histera 』は、FPS というジャンル全体を革新するという、並外れたアイデアとして始まりました。それをどのように実現したかというと「グリッチ」です。グリッチを主な USP として活用したのです。つまり、地図の 1 つの区間をまったく新しい時代のものに変更するのです。先史時代から未来の時代まで移動できるうえ、レイアウトも完全に変わります。

DOTS を選んだ理由は、当時、ネットワークソリューションを検討していたものの、利用可能な選択肢があまりなかったからです。一人称シューティングゲームを作っていたので、ピアツーピアは信頼できる選択肢ではないとわかっていました。私たちは、専用のゲームサーバーが欲しかったのです。Unity のブログで Unity Netcode と DOTS のリリースの詳細が紹介されており、いくつかのサンプルも公開されていました。その技術で FPS が作れるということがわかったので、とても興味をそそられたのです。

それらの（DOTS）パッケージを深く掘り下げたところ、開発者目線からしても非常に興味深いことがわかりました。まったく新しいパラダイムだったのです。オブジェクト指向ではなく、データ指向に基づいたものだったからです。何度も話し合い、議論を重ねた結果、私たちは DOTS と ECS に関する知識に投資することを決めました。」

- StickyLock Games プロデューサー、Jamel Ziaty 氏

DOTS を使った作例：『V Rising』（開発元：Stunlock Studios）

『V Rising』（Stunlock Studios 作、Made with Unity、PC で配信中）

Stunlock Studios が『V Rising』の制作に着手したとき、その構想のスケールの大きさから、これまでのタイトルとは異なるデザインパターンが必要であることがすぐ判明しました。共同設立者兼テクニカルディレクターの Rasmus Höök 氏は、「私たちは、破壊可能な要素やインタラクティブな要素がたくさんある、生き生きとした世界を作りたかったのです」と言います。

Höök 氏が DOTS の試験的な利用を始めたのは、「そのユースケースが、私たちが解決しようとしている問題と完璧に合致しているように思えたから」でした。DOTS と ECS を使用することで、Stunlock はサーバーの負担を減らし、クライアントの CPU リソースを最小限に抑えました。その結果、同時に遊べるプレイヤーの数が増え、システム要件が下がり、Stunlock Studios のクリエイティブなビジョンに合わせて拡張できる堅牢な技術スタックが実現しました。

「ECS で、エディターデータとランタイムデータが明確に分離されているのは、大きなアドバンテージです。エディターで作業するときは、実質的には MonoBehavior を持つ標準的なゲームオブジェクトである、オーサリング用のプレハブを作成します。ただし、これらのプレハブはあくまで編集用であり、ゲームそのものに直接使用されることはありません。その代わり、これらはベイキングと呼ばれるプロセスを経て、ランタイムコンポーネントに変換されます。オーサリング用のプレハブはエディターでのみ使用されるため、実際のゲームが煩雑になる心配なく、ワークフロー改善のために機能やデータを追加することができます。

これにより、エディターのデータに影響を与えることなく、ランタイムコンポーネントを自由に変更し、最適化することができます。この分離は『V Rising』のような複雑なプロジェクトの維持にはとても役に立ちました。」

- Stunlock Studios テクニカルディレクター、Rasmus Höök 氏

DOTS を使った作例：『Zenith: The Last City』（開発元：Ramen VR）

『Zenith:The Last City』（Ramen VR 作、Made with Unity、PC およびコンソール VR で配信中）

MMO はシステムベースのゲームであるため、強力でスケーラブルな技術基盤が必要です。開発当初、Ramen VR は MonoBehavior を使って『Zenith』のシステムを構成していましたが、何百個もの同一のゲームオブジェクトで何百回もロジックを実行するのは非効率的でした。

そこで、オブジェクト指向プログラミングの欠点を避けるために、Unity の ECS フレームワークを活用したのです。[CTO Lauren] Frazier 氏は、「MMO は ECS の素晴らしい活用例です。」と述べています。「『Zenith』では何千ものエンティティが同時に共存する必要がありますが、ECS はそれを可能にします。」

新しいワークフローでは、すべての「アクター」ゲームオブジェクト（プレイヤー、モブ、収集可能アイテム）に、対応する ECS エンティティがあります。ECS はゲームオブジェクトを調べ関連タグをチェックし、タグが見つかるたびにロジックをトリガーします。

「状況に応じてワークフローを選べるのは効率的でした。オブジェクトだけ、またはエンティティだけで対応することも可能でしたが、わざわざ統一する必要もないと思います。」

- Ramen VR CTO、Lauren Frazier 氏

DOTS を使った作例：『Megacity Metro』サンプル

『Megacity Metro』サンプル

Unity の『Megacity Metro』サンプルは、『Megacity』サンプルのオリジナル版をマルチプレイに対応させ、モバイルに特化したものです。『Megacity Metro』は、URP、Entities、Netcode for Entities、Unity Physics で構築されており、ローエンドのモバイルからハイエンドのコンソールプラットフォームまで、さまざまなデバイスで動作します。100 人以上のプレイヤーによる対戦型クロスプレイをサポートし、Authentication、Game Server Hosting、Matchmaker、Vivox などの Unity Gaming Services を紹介しています。

こちらから『Megacity Metro』の詳細を読み、プロジェクトをダウンロードすることができます。

付録：ECS に関連するコンセプト

DOTS のデータ指向の要素は、Monobehaviour プロジェクトのようなオブジェクト指向によるプログラミングアプローチより、ハードウェアへの負荷が少なくなります。Monobehaviour ベースのプロジェクトで使う C# プログラミングとは通常あまり関連性のない、DOD 関連のコンセプトや DOD の影響を受けるいくつかの重要なコンセプトについて、理解を深めることもできます。

メモリの割り当てとガベージコレクション

最近のオペレーティングシステムでは、プログラムは別々のプロセスとして実行され、各プロセスのメモリはオペレーティングシステムによって管理されます。プロセスがより多くのメモリを必要とする場合は、オペレーティングシステムにリクエストを行わなければならず、リクエストが行われた際はオペレーティングシステムがプロセスに連続したメモリブロックを与えます。これはメモリ割り当てと呼ばれています。

プロセスが終了すると、オペレーティングシステムは、メモリを他の場所で使えるように解放します。しかし、長時間実行されるプログラムにおいては、プログラムの方から使用しなくなったメモリブロックを返却する方が理にかなっていることが多いです。これはメモリの割り当て解除、または解放と呼ばれています。短時間のみ実行されるシンプルなプログラムであれば、多くの場合、メモリを解放せずに割り当てるだけで問題ありません。しかし、長時間実行されるプログラムが、新しいメモリを割り当て続けながら一度も解放しなかった場合、そのプログラムは必要以上に多くの量のメモリを使用する可能性があります。このような状況はメモリリークと呼ばれ、パフォーマンスの低下や不安定性の原因となります。

プログラムは多くの場合、以下のように独自の内部アロケーターを使用します。

1. プログラムがオペレーティングシステムから大きなメモリブロックを割り当てます。
2. プログラム内部のアロケーターが、ブロック内で現在使われている範囲をトラッキングします。
3. プログラムが追加のメモリを必要とする場合、オペレーティングシステムではなく、内部のアロケーターにリクエストします。
4. 内部で割り当てられたメモリブロックが不要になった場合、プログラムはアロケーターにメモリを解放するよう通知する必要があります。

内部アロケーターにはいくつかの利点があります。

— オペレーティングシステムからの割り当てや割り当て解除とは異なり、多くの場合、内部アロケーターからの割り当てや割り当て解除には高コストなシステムコールが発生しません。

— プログラムは複数のカスタムアロケーターを使用して、さまざまなユースケースに対応することができます。アロケーターの中には、期間の短い小さな割り当てに適しているものもあれば、長時間にわたる大きな割り当てに適しているものもあります。例えば、いわゆる「アリーナアロケーター」は、すべての割り当てを同時に解放するので、その内部ロジックと記録は非常にシンプルで低コストなものになります。

C# を含む一般的な言語の多くにおいて、ランタイムはガベージコレクターを使用し、メモリをスキャンして未使用の割り当てを見つけ、それらを解放します。この自動化された方法は手動での割り当てよりも便利ですし、メモリリークやその他のメモリ関連の問題も回避しやすくなります。欠点としては、ガベージコレクションにはオーバーヘッドが発生し、プログラムの実行を中断する必要があることが挙げられます。これにより、プレイヤーの体験に悪影響を与えるほどの一時停止が発生する可能性があります。

DOTS では、エンティティおよびネイティブコレクションはアンマネージであり、ランタイムやそのガベージコレクターによって割り当てられたり管理されたりすることはありません。

— エンティティの場合、EntityManager によってメモリの割り当てと解放が行われるため、エンティティがメモリリークを起こすのは、不要になったときに破棄するのを怠った場合のみです。このようなケースは目立つことが多いので、簡単に発見および修正できます。

— ネイティブコレクションの場合、DOTS はトレードオフの異なるいくつかのアロケーターを提供しています。例えば、**Allocator.Temp** アロケーターは、非常に低コストな割り当てを提供し、フレームや割り当てられたジョブの終了時に自動的に破棄されます。これとは対照的に、**Allocator.Persistent** アロケーターは、手動で解放するまで無期限に持続する、より高コストな割り当てを提供します。Allocator.Temp からの割り当てとは異なり、Allocator.Persistent は大きなメモリ領域を割り当てることができ、ジョブに渡して使用することもできます。他にも、**Allocator.TempJob** や **WorldUpdateAllocator** などのアロケーターがあります。

詳しくは Unity のガベージコレクターについてのドキュメントをご覧ください。

マルチスレッドプログラミング

最近の CPU の多くは複数のコアを備えており、これらの追加コアを使用することで、CPU 負荷の高いゲームのパフォーマンスを大幅に向上させることができます。しかし、マルチスレッドは多くの C# 開発者にとって馴染みのない、手作業による低レベルのプログラミングを必要とするため、困難かつ危険な場合があります。DOTS では C# ジョブシステムにより、安全なマルチスレッドコードを簡単に書き、よくある落とし穴を避けることができますが、このセクションは根本的な問題を理解するのに役立つはずです。

オペレーティングシステムによってプロセスがスポーンされると、そのプロセスは 1 つの実行スレッドとして始まります。システムコールを通じて、プロセスは同じプロセスに属し、結果として同じメモリを共有する追加のスレッドを生成することができます。

CPU の各論理コアは一度に 1 つのスレッドを実行することができ、オペレーティングシステムは、どのスレッドがどのコアでいつ実行されるかを制御します。オペレーティングシステムは、いつでも実行中のスレッドに割り込んでサスペンドし、別のスレッドがそのコアを使えるようにできます。2 つのスレッドが同じリソース（つまりデータ）にアクセスする場合、一方のスレッドが、もう一方のスレッドが予期していないタイミングでリソースを変更してしまう危険があります。一般的にスレッドは、スレッドが共有リソースに排他的にアクセスできるコードスパンである「クリティカルセクション」でのみ、共有リソースの読み取りと変更を行うべきです。

スレッド間のアクセスを制御するために、共有リソースはミューテックスなどの同期プリミティブによって管理されます。しかし、これらの同期プリミティブは、一般にそれらを使用するすべてのスレッドが厳格なプロトコルに従うことを要求し、それを怠るとプリミティブが機能しなくなったり、プログラムのハングアップが発生します。

もう 1 つの問題は、特定のシステムコールが呼び出し元のスレッドをブロックし、実行をサスペンドする可能性があることです。例えば、スレッドがファイルを読み込むためにシステムコールを呼び出した際、データはおそらくまだメモリ上にないため、システムコールがリターンする前に、まずデバイスからメモリにコピーされる必要があります。データ待ちの時間が（CPU からすると）非常に長くなる可能性があるため、データがロードされている間、オペレーティングシステムが呼び出し元のスレッドをブロックし、その間に CPU コア上で別のスレッドが実行される場合もあります。オペレーティングシステムは、データの準備が整った段階で初めてスレッドのブロックを解除し、実行を再開させます。

マルチスレッドの 1 つの活用例として、「メインスレッド」で作業を続けながら、ファイルの読み込みや書き込みなど、より長い実行時間を要するブロック処理を「バックグラウンドスレッド」で行うことが挙げられます。例えば、インタラクティブなプログラムでは、メインスレッドがユーザー入力に応答して画面を再描画している間に、バックグラウンドでファイルをロードすることもできるでしょう。

また、単純にプログラムの CPU 負荷を複数のコアに分割して、作業を高速化できるようなケースもあります。例えば、データ圧縮は CPU に非常に負荷がかかるため、マルチスレッドが役に立つことが多いです。

注意しなければならないのは、CPU コアはストレージデバイス、システムメモリ、その他のシステムリソースの使用を巡って、互いに争う必要があることです。例えば、2 つのスレッドが同時にメモリにアクセスしようとした場合、同時にアクセスすることはできず、交互にアクセスする必要があります。幸いなことに、これらの競合はハードウェアレベルで解決されますが、各スレッドのメモリアクセスが他のすべてのスレッドの競合するメモリアクセスの速度が遅くなるという問題は残ります。特に、CPU の計算量と比べて大量のメモリアクセスを必要とするタスクでは、複数のスレッドに作業を分割することで得られるメリットは多くありません。

例えば、10 個のスレッドに分割されたタスクは、1 個のスレッドで実行される同じタスクよりも 10 倍速く実行されるはずだと考えられるかもしれませんが、メモリ競合の存在により、この理論的限界が実際に達成されることはほとんどありません。実際に、10 個のスレッドで現実的に得られるブーストは 5 〜 7 倍程度でしょう（ただし、これもまた具体的なシナリオによって変わります）。タスクが計算作業に比べてメモリアクセスの比率が高い場合、スレッド数が少ないほどメモリ競合が少なくなるため、スレッド数を減らすことで性能を向上できる可能性すらあります。

メモリと CPU キャッシュ

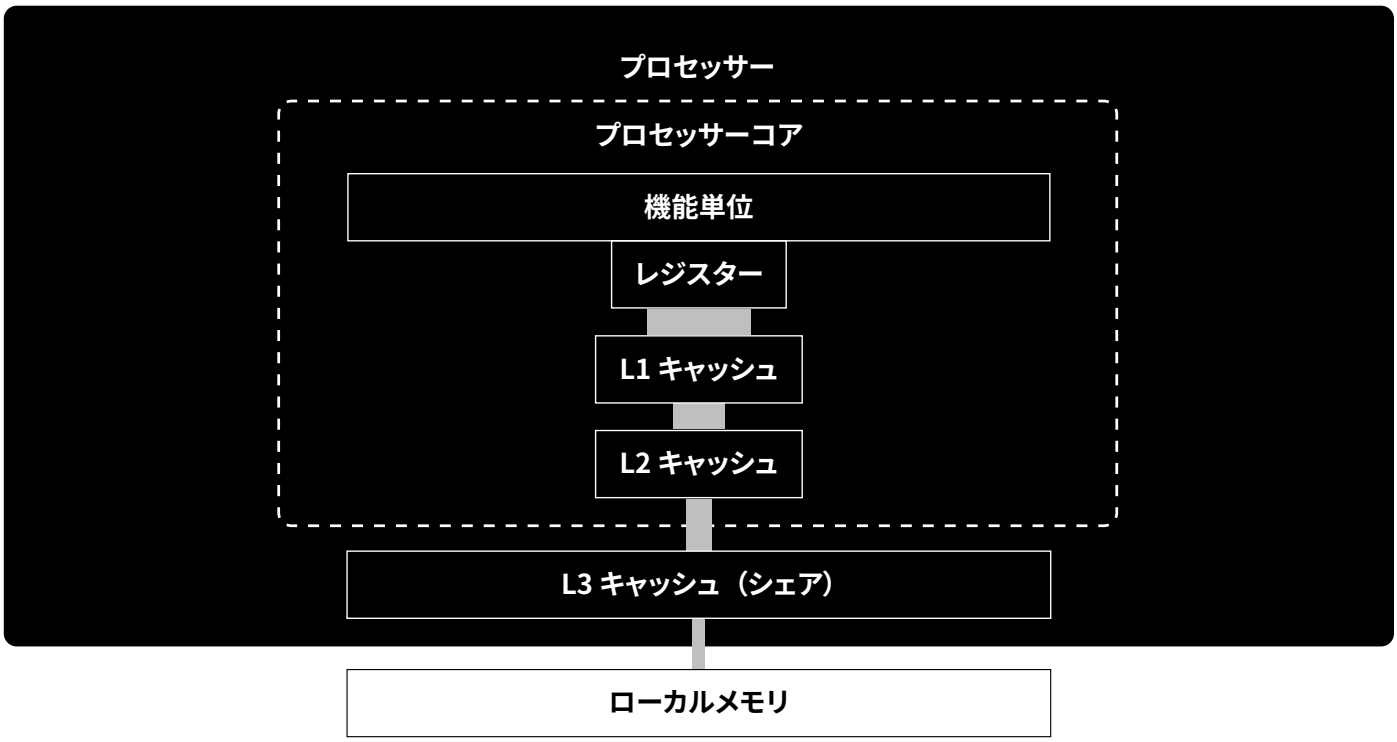
現在の CPU のほとんどは、通常 L1、L2、L3 と呼ばれる 1 〜 3 レベルのキャッシュを搭載しています。

CPU がシステムメモリのアドレスを読み出す命令を実行すると、ハードウェアはまず、そのアドレスのデータのコピーが L1 キャッシュにあるかどうかを確認します。なかった場合、ハードウェアは L2 キャッシュ（存在する場合）を確認し、その次に L3 キャッシュ（存在する場合）を確認します。コピーがどのキャッシュにも存在しない場合、ハードウェアはシステムメモリ自体を読み込みます。データを読み込むと、ハードウェアはそのデータを下位のキャッシュレベルの一部にコピーします。例えば、L3 から読み出されたデータは L1 と L2 に、システムメモリから読み出されたデータはすべてのキャッシュレベルにコピーされます。特定のアドレスが読み込まれると、その直後に同じアドレスが再び読み込まれる可能性が高いので、このキャッシュ戦略は理にかなっています。データをキャッシュにコピーすると、次にそのデータが必要になった時、キャッシュから直接読み出せる可能性が高くなります。

もちろん、システムメモリのすべてがキャッシュに収まるわけではありません。各レベルのキャッシュはその上のレベルよりも小さく、最大のキャッシュでもシステムメモリ全体よりもはるかに小さくなります。そのため、メモリの一部がキャッシュにコピーされる場合、以前にキャッシュされていたメモリの他の部分を上書きする必要があります。

「キャッシュヒット」とは、データがキャッシュから読み込まれることを指します。データのコピーが現在キャッシュされておらず、システムメモリ自体からデータを読み出さなければならない場合は「キャッシュミス」と呼ばれます。

キャッシュの正確なパフォーマンス特性はチップによって大きく異なりますが、おおよそ、L1 は L2 より少なくとも数倍速く、L2 は L3 より少なくとも 1 桁速く、L3 はシステムメモリより少なくとも 2 倍の速度を持っています。全体的には、CPU はシステムメモリのデータにアクセスするよりも、L1 キャッシュのデータにアクセスする方が、少なくとも 2 桁は速くなるでしょう。下位のキャッシュレベルとシステムメモリとの速度差は非常に大きいため、プログラムで発生するキャッシュミスの数を最小限に抑えることがパフォーマンスの鍵になります。



キャッシュのレベル（ソース：https://tech4gamers.com）

プリフェッチと呼ばれるハードウェア機能により、キャッシュミスを最小限に抑える最もシンプルで効果的な方法は、メモリのアドレスをランダムに飛び回るのではなく、順次アクセスすることになります。よって、データ構造の要素をメモリに密に、かつ連続的に格納することで、キャッシュ効率が上がります。つまり、データを配列に格納するのです。

メモリのアドレスを順番に読み始めるとハードウェアはこの法則に気づき、このまま読み続けることを想定して、先読みとメモリのキャッシュへのコピーを開始します。これは余分なデータが必要ない場合は無駄になるかもしれませんが、配列を読み込んでいる場合、このプリフェッチ動作により、CPU が必要とする直前にデータをキャッシュに入れることができます。つまり、配列の最初のバイトを読み込むときにキャッシュミスが発生する可能性を除けば、キャッシュミスを引き起こさずに配列を読み込むことが可能になるのです。CPU が列車を走らせるのと同時に、タイミングよく線路が列車の目の前に敷かれていくイメージです。

ゲームオブジェクトや MonoBehaviour のようなマネージ C# オブジェクトは、個別にインスタンス化されるため、メモリの異なる場所に保存される可能性があります。そのため、多くの場合、大量のマネージオブジェクトを走査するにはメモリ上のさまざまな場所を参照する必要があり、キャッシュミスが多く発生しやすくなります。

DOTS では、設計上、エンティティおよびそのコンポーネントは連続した配列に緊密に詰め込まれており、最小限のキャッシュミスで順次走査できるようになっています。

メモリとキャッシュについてのさらなる情報は、Scott Meyers 氏によるトーク、「CPU Caches and Why You Care」をご覧ください。

オブジェクト指向プログラミングのコスト

オブジェクト指向プログラミング（OOP）に共通する課題は、その定義の多さです。OOP は継承、ポリモーフィズム、カプセル化、あるいはその 3 つの組み合わせがすべてだと主張する人もいれば、あまり型にはまらない理論を提唱する人もいます。Wikipedia の定義は以下のとおりです。

「オブジェクト指向プログラミング（OOP）とは、「オブジェクト」という概念に基づいたプログラミングパラダイムで、フィールド（多くの場合、属性またはプロパティとして知られる）の形のデータと、プロシージャ（多くの場合、メソッドとして知られる）の形のコードを含む。**OOP において、コンピュータープログラムは、相互作用するオブジェクトから構成されることによって設計される。**」– Wikipedia

つまり、オブジェクト指向のプログラムは、相互作用する「オブジェクト」で構成されており、各オブジェクトはデータとコードのカプセル化された単位であり、ある程度の自律性と他からの独立性を保っています。オブジェクト指向プログラムのオブジェクトは、ネットワーク上のプログラムが互いにメッセージを送り合うことで協力し合うように、互いのメソッドを呼び出すことで協力し合います。実際のところ、オブジェクト指向プログラミングを定義するのは、個々のオブジェクトそのものではなく、オブジェクト同士の相互作用なのです。

理論上、OOP には以下のような利点があります。

- **コンポーザビリティ**：オブジェクトで構成されたプログラムは、段階的に組み立てたり変更したりできます。
- **再構成可能性**：オブジェクトの挿入、削除、入れ替えなどを行うことで、簡単に機能の追加、削除、変更を行えます。
- **コードの再利用**：オブジェクトはプログラム間で簡単に再利用できます。
- **直感性**：現実世界の物やプロセスは、自然な形でオブジェクトにも対応します。
- **抽象化**：オブジェクトは、プログラマーが低レベルの詳細に気を取られることなく、高レベルで問題を解決することを可能にします。

『Rolling Stone』誌 1994 年 6 月 16 日号では、Steve Jobs 氏がこの最後のポイントについて詳しく話していました。

> 「オブジェクトは人間のようなものです。生きていて、呼吸をしていて、物事のやり方について知識を持ち、物事を記憶するためのメモリを保持しています。オブジェクトとは、非常に低いレベルで相互作用するのではなく、ここでやっているように、非常に高い抽象度で対話するのです。
>
> 例を挙げてみましょう。もし私が洗濯担当のオブジェクトだとしたら、汚れた服を渡して、「服を洗濯してください」とメッセージを送ってくれれば良いのです。私は、サンフランシスコで一番のランドリーがどこにあるのか知っています。英語も話せますし、ポケットにはお金が入っています。なので、外に出てタクシーを呼び、サンフランシスコの特定の住所に連れて行ってくれるようにドライバーにお願いします。そして服を洗い、またタクシーに乗り、ここに戻ってきます。その後、きれいになった服を渡し、「洗濯が終わりました」と告げます。

> あなたは、私がどうやってそれを成し遂げたのかわからないでしょう。何故なら、あなたはランドリーのことなど知らないのです。話す言語がフランス語で、タクシーを呼ぶことすらできないかもしれません。ポケットにお金がないので、タクシー代を払うこともできません。ですが、私がそれらすべてのやり方を知っていました。あなたが知っている必要は一切なかったのです。このような複雑性はすべて私の中にあり、私たちはとても高いレベルの抽象度で相互作用を行うことができました。オブジェクトとはそういうものです。オブジェクトは複雑さをカプセル化しており、その複雑さへのインターフェースはハイレベルなものです。」

OOP のパフォーマンスコスト

デメリットとして、OOP では多少のパフォーマンスコストが生じる傾向があります。

- **分散したデータレイアウト：**OOP コードは多くの小さなオブジェクトに分割されることが多く、データがメモリ全体に分散されてしまうことも多いです（前のセクションで説明したように、これはキャッシュの非効率性につながります）。
- **過度な抽象化：**オブジェクト指向設計では、多くの場合、上位のレベルが下位のレベルに実際の作業を委ねるようなデリゲートのレイヤーができやすくなり、結果として実際の作業をほとんど行わないオブジェクトやメソッドが多く発生します。
- **複雑な呼び出しチェーン：**抽象化の層が多く、小さな関数が好まれるため、呼び出しチェーンが非常に複雑になります。
- **仮想呼び出し**：仮想ディスパッチテーブルは通常の関数呼び出しに比べてオーバーヘッドが多く発生するだけでなく、仮想呼び出しは通常インライン化できません（JIT コンパイラーによっては実行時にインライン化する場合もあります）。
- **望ましくない割り当てパターン：**OOP で発生しやすい複雑なコードパスは、オブジェクトの生存期間の推論をしばしば困難にします。そのため、OOP コードは、より効率的な代替案ではなく、頻繁で小さな割り当てやガベージコレクションに依存する傾向があります。
- **1 回ずつの処理：**オブジェクトを直接操作するコードがオブジェクト自体の一部であるため、OOP ではオブジェクトを一括処理するのではなく、自然に 1 つ 1 つ処理する傾向があります。

OOP の構造上のコスト

パフォーマンスの最適化を犠牲にしても、プログラムの作成・保守を容易にしたいと思うかもしれませんが、OOP は、プログラムの作成・保守の点でも欠点があります。以下にいくつかの例を挙げます。

1. データとコードを一緒に扱うことで、両者に混乱が生じ、より複雑になります。

よく、OOP はコードよりもデータを優先すると言われています。

> 「オブジェクト指向プログラミング（OOP）とは、関数やロジックではなく、データやオブジェクトを中心にソフトウェア設計を行うコンピュータープログラミングモデルです。（中略）OOP は、オブジェクトを操作するために必要なロジックよりも、開発者が操作したいオブジェクトに焦点を当てる傾向があります。」
>
> – Alexander S. Gillis 氏、「What is Object-Oriented Programming」（TechTarget Network）

しかし実際は、OOP はデータとコードを一緒に扱います。オブジェクトの能力がそのデータに直接依存し、またその逆も同様であるとすれば、オブジェクトにできることはその定義と一体であり、オブジェクトのデータから切り離すことは不可能です。

このような依存関係により、以下のような、適切ではないかもしれない設計になることがあります。

— コードを持っているが、本当はデータのみを持つべきオブジェクト。

— データを持っているが、本当はコードのみを持つべきオブジェクト。

— コード中心で考え、データをグループ化しているオブジェクト。

— データ中心で考え、コードをグループ化しているオブジェクト。

— データ中心で考え、複数オブジェクト間に分割されているコード。

— コード中心で考え、複数オブジェクト間に分割されているデータ。

2. 複雑さを集中させるのではなく分散させると、全体の複雑さが増すことがあります。

オブジェクト指向設計のルールによると、「役割」が多すぎるオブジェクトは、より小さなオブジェクトに分割されるべきです。しかし、大きなものを小さく分割することで、全体的な複雑さを軽減するのではなく、複雑さを分散させるだけで終わる可能性があります。実際、小さな断片がたくさんあるコードベースでは、どのデータやコードがどのような目的で使われているのか判別しにくく、特定の機能に関連するコードの部分を見つけにくくなることが多いです。

そのため、オブジェクト指向設計では、適切に設計されたオブジェクト群の間で責任が正しくデリゲートされている限りはコードがわかりやすくなりますが、オブジェクト指向設計のプロセス自体がしばしば負担になり、憶測が飛び交い、結果として典型的なプログラム構造が過度に分断されたものになっています。

3. オブジェクトでは、どのコードがどのデータにアクセスしたか追跡することが困難になります。

プログラムを理解することは、最終的にはそのデータを理解し、そのデータがどのように変換されるか理解することに帰結します。データについての推論の簡単さに比例して、プログラムについての推論も簡単になります。プログラマーは、機能を追加したりバグを修正したりする際、どのコードが特定のデータの一部に影響を与えるのか、また別の視点から見れば、どのデータが特定のコードの一部に影響を受けるのか判断できなくてはなりません。

オブジェクト指向のプログラムでは、オブジェクトのつながりが増えるほど、こうした判断が難しくなっていきます。オブジェクトのカプセル化は、データの一部への直接的なアクセスがプライベートに保たれるかもしれませんが、間接的に接続されたオブジェクトには、public メソッドの呼び出しを通じて間接的にアクセスできる可能性があります。例えば、ある値が正しく設定されていない理由をデバッグする際、関連するコードのパスをすべて特定するのはかなりの作業を要する可能性があります。これとは対照的に、厳格な手続き的プログラムにおいて、データの一部に影響を与える可能性のあるコードのすべてのパスを特定するために考慮しなくてはならない可能性は、通常（プログラムが無謀にグローバル変数を使用しない限り）はるかに少なくなります。

データ指向設計

データ指向設計（DOD）という単語は、2000 年代に一部のゲームプログラマーや高パフォーマンスなソフトウェアに関心を持つ人たちの間で生まれた概念を表す造語です。データ指向設計の権威ある定義は存在しませんが、おそらく以下のリソースが最も近いでしょう。

- 「Data-Oriented Design and C++」および「Building a Data-Oriented Future」（Mike Acton 氏によるトーク 2 つ）
- 「Data-Oriented Design」Richard Fabian 氏による書籍
- データ指向設計のリソース：DOD についてのリンク集

ここでは、理論的な説明をするのではなく、DOD をいくつかの実践的なアドバイスに集約します。

コードの前にデータを設計する

DOD の大前提は、**データは少なくともコードと同じくらい大切だ**ということです。マクロレベルでもマイクロレベルでも、**プログラムとは結局のところ、データを変換し、生成することがすべてなのです**。

つまり、データの性質がコードの構造を決めるべきで、その逆ではありません。これはプロジェクトの初期だけでなくすべての段階で言えることで、機能を追加したり変更したりする場合は、コードを再構築する前に、まずデータの構造を再評価する必要があります。

これは、オブジェクトによりデータとコードを密接に結びつけるオブジェクト指向設計とは相反する考えです。データの設計とコードに関する懸念が混在すると、設計プロセスが複雑になり、最適とは言えない設計の選択につながることも多いです。反対に、コードの影響を気にすることなくデータを自由に変更できるようにすることで、設計プロセスが簡素化され、一般的によりシンプルで最適なデータが得られます。

シンプルなデータが好ましい

一般的な傾向として、シンプルなデータはシンプルで効率的なコードにつながります。特に、階層構造やグラフ構造よりも配列を優先すべきです。なぜなら配列は、多くのデータ要素を格納する最も単純な方法であり、一次元配列を順次ループすることは、メモリにアクセスする最も効率的な方法だからです。

また、データの要素間に（ポインターや配列のインデックスを介して）不必要なつながりを作ることにも注意が必要です。これらのつながりを正しく維持しようとするとコードが複雑になり、つながりを走査するためには最適とは言えないランダムなルックアップが必要になります。

コードをデータパイプラインとして捉える

大まかなデータ設計が完成したら、次はデータがどのような変換を受ける必要があるのか考えなくてはなりません。

- サーバーでは、クライアントのリクエストやデータベースのデータがサーバーレスポンスに変換されます。
- コンパイラーでは、ソースコードがマシンコードや何らかの中間コードに変換されます。
- オーディオエンコーダーでは、ある形式のオーディオデータが別の形式に変換されます。
- ビデオゲームでは、ある瞬間のユーザー入力とゲームの状態が新しいゲーム状態に変換され、それが新しいレンダリングフレームに変換されます。

もちろん、このようなマクロレベルの変換はいくつかのサブステップに分かれますが、目的自体は同じです。データのなんらかの開始状態から、想定される最終状態に到達するために点をつなぐことです。そして、コードは自然に、各ステップでパイプラインの後のステップに渡すデータを変換または生成する「データパイプライン」として構造化されます。

プログラミングをこのように説明するとあまりに単純で明白に聞こえるかもしれませんが、ソフトウェアの作り方に関する他の理論に比べると非常に明快です。始点と終点が明確に定義されていれば、A 点から B 点への行き方の正確な把握は、非常に具体的で扱いやすい問題であり、各変換は他の部分から独立して記述したり書き直したりできます。

このモデルは、機能するソリューションの作成だけでなく、最適化も容易にします。

まず、一連のステップにおけるボトルネックを特定するのは、すべてのステップをプロファイリングするのと同じくらい簡単です。多くの場合は少数のステップがコストの大半を占めるので、どこを最適化すれば最も効果的なのか明確に知ることができます。したがって、最適化における優先順位を決める際には、コストと利益を考慮し、パフォーマンス最適化プロセスにおいて現実的な考え方をすることが重要です。

次に、パイプラインモデルは最適化機会をより見つけやすくします。以下のようなケースが多く見られるでしょう。

- 特定のデータが、後のステップでより効率的に処理できるような中間形式に変換されるべきである。
- 以前のステップで一度だけキャッシュされるべきデータが、複数のステップで冗長に生成されている。
- 同じものに繰り返しアクセスする際のオーバーヘッドを減らすために、同じデータにアクセスする別々のステップの全部または一部を、より少ないステップに統合する余地がある。
- 個別に処理されるデータ要素の一部は、メモリアクセスの効率化、分岐の削減、関数呼び出しオーバーヘッドの削減、その他の効率化のために一括処理されるべきである。

最後に、データパイプラインは並列化に適しています。どのステップがどのデータに触れるかを明確に分けていれば、どのステップが安全に同時処理できるかを簡単に特定できます。

開発の全段階でパフォーマンスの測定、見積もり、予算化を行う

ゲーム開発でよくあるミスとして、プロジェクトの最後までパフォーマンス修正を後回しにすることが挙げられます。以下の理由から、遅い段階での最適化作業は高コストかつ危険です。

— 多くの最適化は、プロジェクトの後半になると難しくなる。

— 遅い段階での最適化にかかる時間や労力を予測するのは難しい。

— 最適化が遅れると、納得のいく結果が得られない可能性がある。

健全なやり方は、最後まで待つのではなく、プロジェクトの最初からパフォーマンスに気を配ることです。プロトタイピングやベータフェーズにおいて最適でないパフォーマンスを許容する場合でも、少なくともプロジェクトのニーズを継続的に再評価し、パフォーマンス予算を再設定すべきです。各機能とゲーム全体において、メモリ、CPU、GPU、ストレージ容量、ネットワーク帯域幅にはどれくらいの余裕がありますか？ターゲットの数字は、ターゲットのプラットフォームによって違いますか？これらの質問は、開発のすべての段階において検討すべきです。

プロファイリングの詳細については、70 ページ以上を誇る Unity ゲームのプロファイリングの究極のガイドよりご覧ください。このガイドでは、Unity でアプリケーションのプロファイリングを行い、メモリを管理し、消費電力の最適化を最初から最後まで行う方法についての、外部および社内の Unity エキスパートによる上級者向けの知識とアドバイスをまとめています。

抽象化よりも具体的なソリューションを

プログラミングにおいて「抽象化」とは、簡略化された外見の裏に内部の詳細を隠す、一般化されたソリューションのことです。抽象化には、関数、オブジェクト、ライブラリ、フレームワーク、プログラミング言語、さらにはゲームエンジンなど、さまざまな形態があります。

ある程度の抽象化は理にかなっているものの、過度な抽象化は問題を引き起こす可能性があります。

— ライブラリやフレームワーク、ゲームエンジンのような既製の抽象化を利用すべき大きな理由として、難しくて時間のかかる実装作業から解放されることが挙げられます。しかし、この便利さの代償として、提供されるソリューションと特定のニーズとの間に厄介なミスマッチがしばしば発生します。最終的に、既成の抽象化ソリューションを自分の目的に沿うように曲げることは、自分専用の解決策を一から作成するよりも大変な作業になってしまう可能性があります。

— 抽象化には、大きなメモリフットプリントや CPU オーバーヘッドなど、結局のところは使うことすらない可能性がある機能のために支払われる、多くの隠れたパフォーマンスコストが伴うことがあります。

— ご自身の実装において、プロジェクトの後半で役に立つかもしれないことを期待して、現在のニーズを超えて一般化する抽象的なソリューションを作ることは魅力的に思えるでしょう。しかし、多くの場合、このような推測的な作業は、解決するよりも多くの仕事を生み出すことになり、結果として得られるソリューションは最適ではない（あるいは少なくとも最適化しにくい）ものになる可能性が高いです。実際、抽象化は、後になって要件が抽象化にうまく当てはまらなくなったときに、変更を行うのをより難しくする可能性があります。

殆どの場合、要件が後でどのように変わるかを心配するよりも、現在理解している要件を解決する方が望ましいでしょう。イテレーションを受け入れてください。実際に問題を解決してみるまでは、その問題を完全に理解することはできませんし、実際に要件が変わるまでコード変更を後回しにすることも可能です。一般的な文章について言われていることは、コードを書くときにも当てはまります。よい文章を書くというのは、すなわち書き直しを行うことです。そして、コードの書き換えを何よりも簡単にするのは、シンプルさです。

抽象化したいと思ったら、一番のアドバイスは待ってみることです。まずは少なくともいくつかの具体的なケースを解決し、その後で解決策を組み合わせて抽象化することを検討しましょう。Richard Fabian 氏は、以下のように述べています。

> 「データ指向設計は今にあります。問題の歴史や今日までの解決策を表すものでもなければ、これから起こることに対処するために作られた一般的な解決策で未来を表すものでもありません。過去にしがみつくと柔軟性が損なわれますし、プログラマーは占い師ではないので、未来を見つめても得られるものはありません。将来性のあるシステムなどめったに存在しない、というのが著者の意見です。」

まとめると、早すぎる抽象化は諸悪の根源と言っても過言ではないでしょう。

unity.com