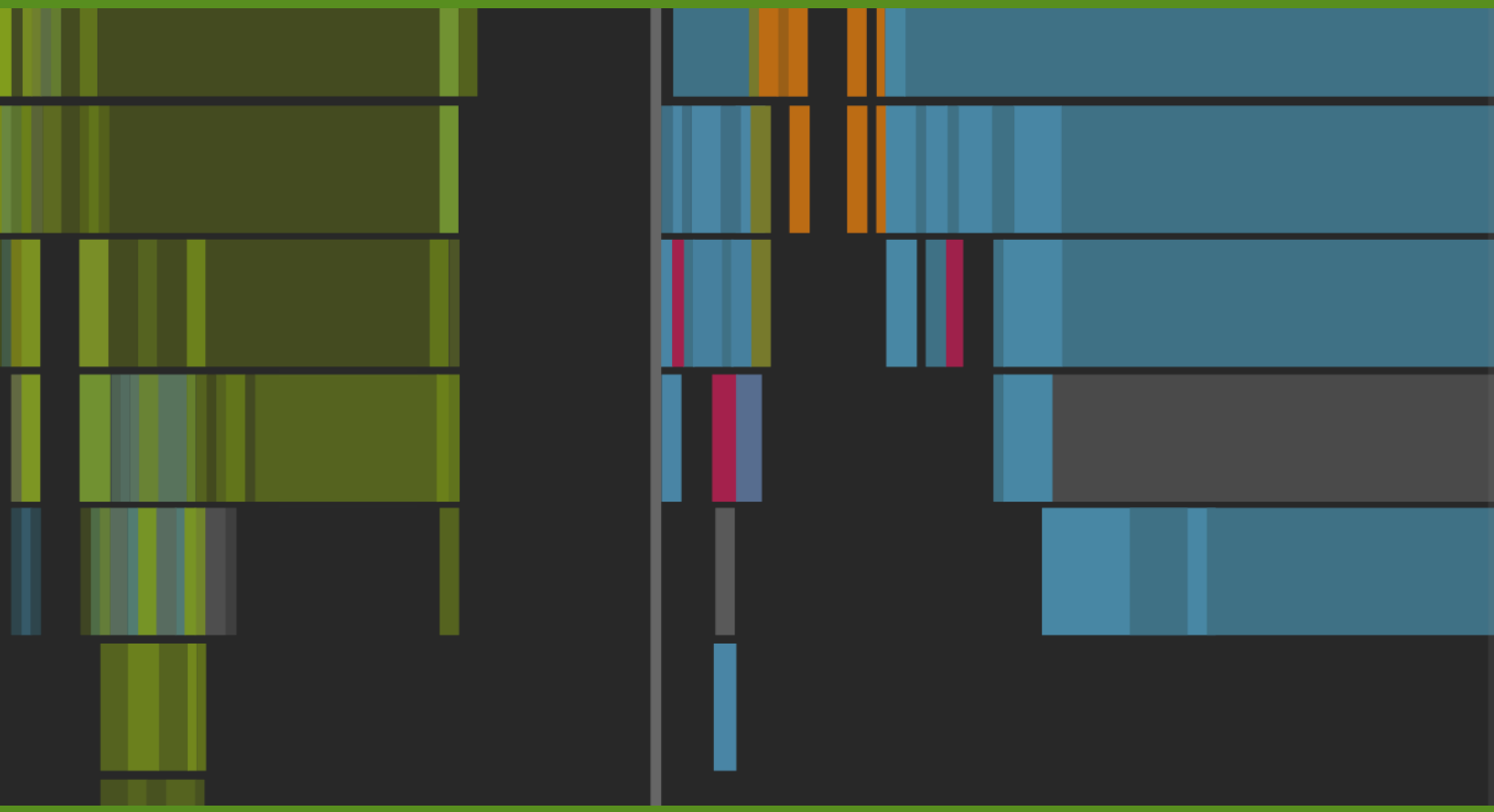
→ E-BOOK

Unity ゲームのプロファイリングに関する決定版ガイド

(Unity 6 版)



Unity®

Contents

はじめに

幅広いデバイスやプレイヤーにリーチする優れたゲーム体験を創造するには、スムーズなパフォーマンスが不可欠です。Unity は、Unity 開発者がターゲットプラットフォームで利用可能なネイティブプロファイリングツールと併用できる、プロファイリングツールとメモリ管理ツール一式を提供しています。

このガイドには、Unity におけるアプリケーションのプロファイリング、メモリの管理、消費電力の最適化についての実用的なアドバイスや知識がまとめられています。

この e-book の第 2 版は、Unity 6 の最新機能と、コミュニティからのフィードバックに基づく改善を反映して更新されました。

Unity のプロファイリングガイドは、以下の Unity 社内のエキスパートと外部のゲーム開発者によって共同作成されました。

— Steven Cannavan (シニアソフトウェア開発コンサルタント)

— Sean Duffy 氏 (ソフトウェアエンジニア兼ゲーム開発者)

— Peter Hall (ソフトウェアエンジニアリング担当シニアマネージャー)

— Thomas Krogh-Jacobsen (コンテンツマーケティング管理担当シニアマネージャー)

— Steve McGreal (ソフトウェアエンジニア)

— Martin Tilo Schmitz (シニアソフトウェアエンジニア)

— Peter Harris 氏、Arm

Unity 6 でのパフォーマンス最適化に関する追加ガイドには、コンソールと PC でのゲームパフォーマンス最適化 と Unity における XR、ウェブ、モバイルでのゲームパフォーマンス最適化 があります。

プロファイリング入門

Unity でゲームをプロファイリングする方法の詳細に入る前に、いくつかの重要な概念とプロファイリングの原則をまとめておきます。

無駄のない高パフォーマンスのコードとメモリ使用量の最適化は、ローエンドデバイスとハイエンドデバイスの両方において、ユーザー体験の向上に役立ちます。これは、熱やバッテリー消費に対処してより多くのローエンドデバイスユーザーへのリーチを可能にすることから、プレイヤーの快適性レベル、そして最終的にはゲームの購入とリテンションを高める各種要因まで、すべてに当てはまります。また、配信プラットフォームの仕様を満たすための要件となることもあります。

一貫したエンドツーエンドのプロファイリングワークフローは、効率的なゲーム開発に "必要不可欠" です。まずは、以下の 3 つの簡単な手順に従いましょう。

— 大きな変更を加える前にプロファイリングを行い、ベースラインを確立します。

— 開発中にプロファイリングを行い、変更がパフォーマンスやバジェットに影響を与えないように追跡および確認します。

— 開発後にプロファイリングを行い、変更が意図した効果をもたらしたかどうかを確認します。

プロファイラーは、開発者のツールキットにおいてもっとも有用なツールの 1 つで、コード内のメモリやパフォーマンスのボトルネックを特定するのに役立ちます。

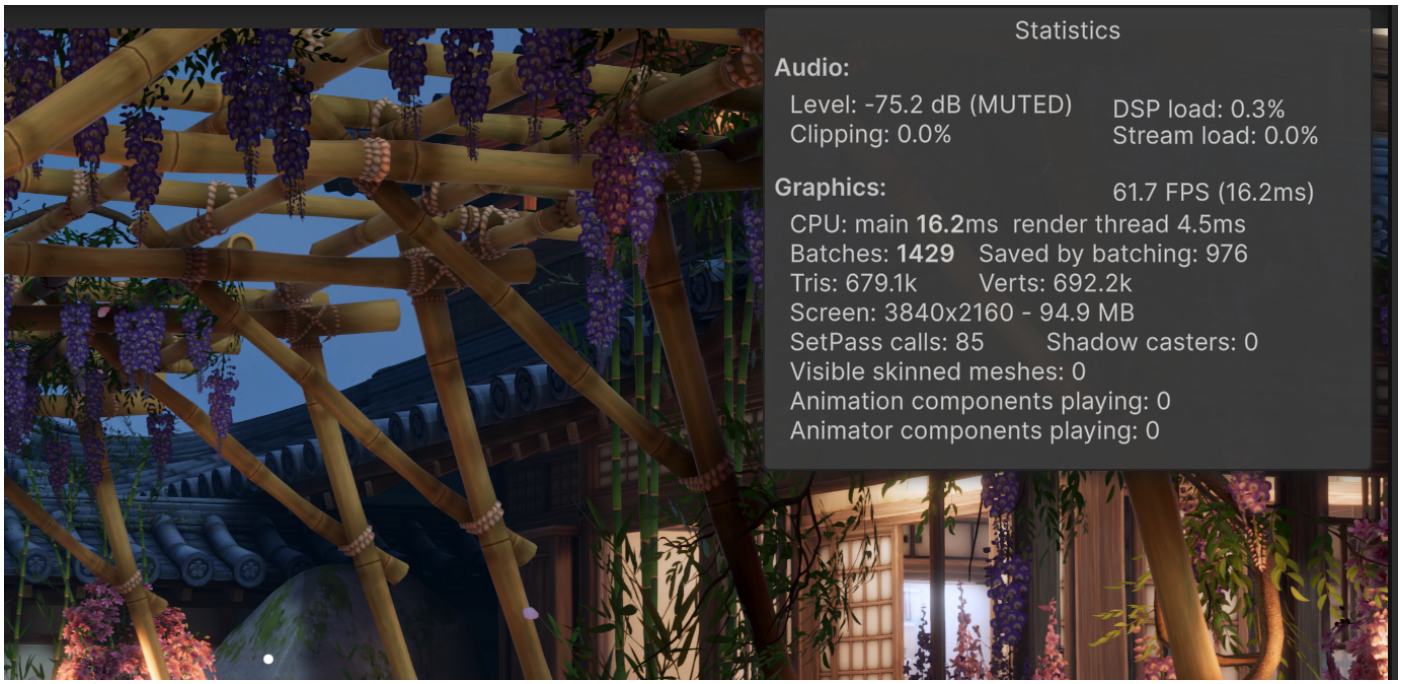
プロファイラーは、アプリケーションのパフォーマンスが低下している理由や、コードが過剰なメモリを割り当てている理由を解明するために役立つ調査用ツールだと考えることができます。内部で起こっていることへの理解を促します。

Unity には、エディターとハードウェアの両方でコードを分析および最適化するためのさまざまなプロファイリングツールが付属しています。これらについては e-book で詳しく説明しますが、モバイルやその他のコードレスデバイス、コンソール、PC など、ターゲットプラットフォームごとにネイティブのプロファイリングツールを使用することをおすすめします。

フレームバジェットを理解する

ゲーマーはフレームレート（1 秒あたりのフレーム数、fps）を使用してパフォーマンスを測定することが多いですが、開発者としては **フレーム時間 (ミリ秒、ms)** を使用することが、一般的に推奨されます。以下の簡略化されたシナリオについて考えてみましょう。

ランタイム中、ゲームが 0.75 秒で 59 フレームをレンダリングするものの、次のフレームのレンダリングには 0.25 秒かかるとします。平均配信フレームレート 60 fps とすれば聞こえは良いですが、実際には最後のフレームのレンダリングに 1/4 秒かかるため、プレイヤーはスタッター効果に気付くでしょう。



Unity には詳細かつ精密な分析を行うためのさまざまなプロファイリングツールが用意されていますが、ゲームビューの統計パネルを見るだけでも、パフォーマンスを簡単に確認できます。

これが、フレームごとに特定のタイムバジェットを設定することが重要である理由の 1 つです。これにより、ゲームのプロファイリングと最適化を行うときに目指すべき確固たる目標ができ、最終的にはプレイヤーにとってよりスムーズで一貫性のある体験が実現します。

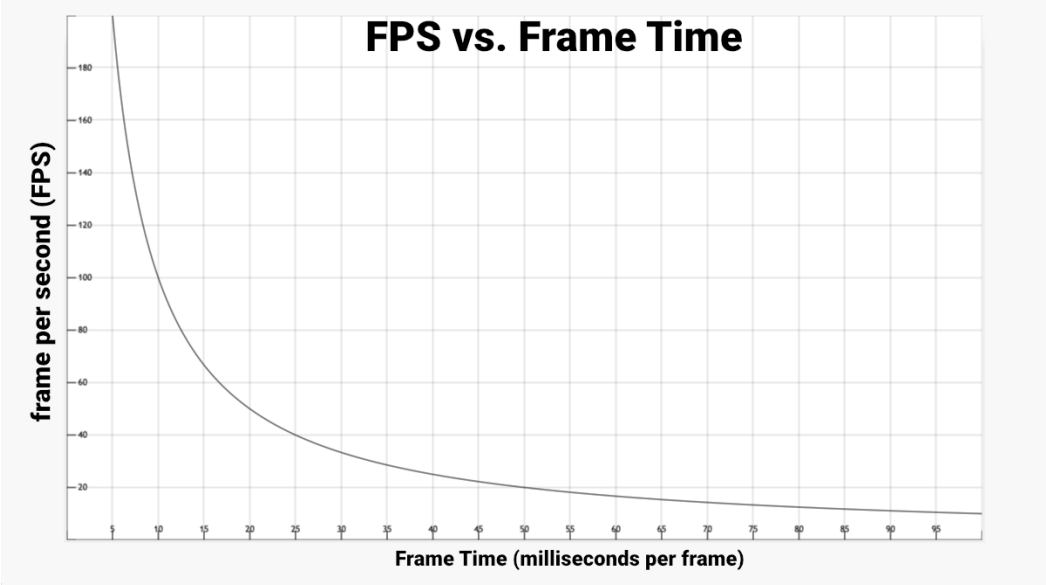
各フレームには、目標とする fps に基づいたタイムバジェットが設定されます。30 fps を目標とするアプリケーションでは、常に 1 フレームあたり 33.33 ms (1000 ms/30 fps) 未満である必要があります。同様に、60 fps を目標とする場合は 1 フレームあたり 16.66 ms 未満である必要があります。

UI メニューの表示やシーンのロードなど、没入感のあるゲーム体験の邪魔にならない非対話型のシーケンスの場合はこのバジェットを超えても問題ありませんが、ゲームプレイ中にはバジェットを超えないようにしてください。目標のフレームバジェットを 1 フレームでも超えると、フレーム落ちが発生します。

VR ゲームで一貫して高いフレームレートを維持することは、プレイヤーに吐き気や不快感を与えないために不可欠であり、多くの場合、ゲームがプラットフォームホルダーから認定を受けるために必要です。

1 秒あたりのフレーム数:誤解を招く指標

フレーム時間の測定に 1 秒あたりのフレーム数ではなくミリ秒が推奨されるのはなぜでしょうか。その理由を理解するには、次のグラフを参照してください。



fps とフレーム時間の比較

これらの数字を考えてみましょう。

1000 ms (1 秒) ÷ 900 fps = 1 フレームあたり 1.111 ms
1000 ms (1 秒) ÷ 450 fps = 1 フレームあたり 2.222 ms

1000 ms (1 秒) ÷ 60 fps = 1 フレームあたり 16.666 ms
1000 ms (1 秒) ÷ 56.25 fps = 1 フレームあたり 17.777 ms

アプリケーションが 900 fps で実行されている場合、フレーム時間は 1 フレームあたり 1.111 ms ということになります。450 fps の場合は、1 フレームあたり 2.222 ms です。フレームレートが半分に低下したように見えますが、1 フレームあたりの差は 1.111 ms でしかありません。

60 fps と 56.25 fps の違いを見ると、それぞれ 1 フレームあたり 16.666 ms と 17.777 ms になります。これもまた、1 フレームあたり 1.111 ms 長いことを示しますが、ここでのフレームレートの低下は、割合的にははるかに少なく感じられます。

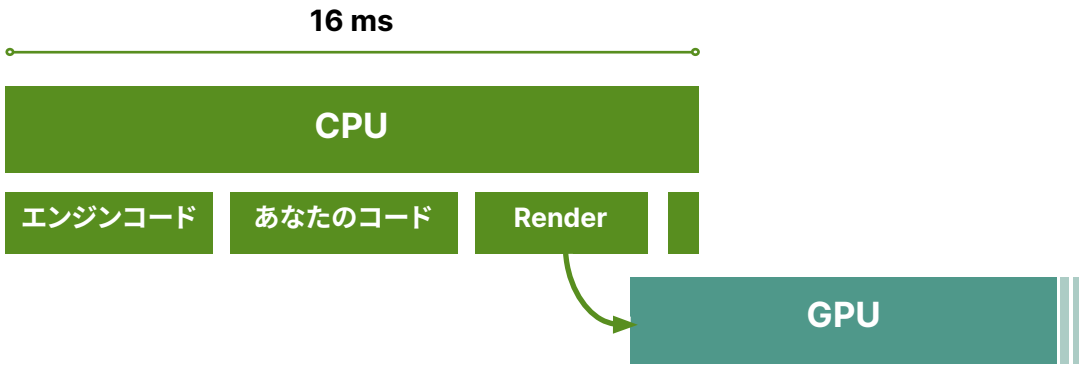
これこそが、開発者がゲーム速度のベンチマークに fps ではなく平均フレーム時間を使う理由です。

目標フレームレートを下回らない限り、fps を気にする必要はありません。フレーム時間に着目してゲームの実行速度を測定し、フレームバジェット内に収まるようにしてください。

詳しくは、Robert Dunlop 氏の記事 FPS versus Frame Time (fps とフレーム時間の比較) を参照してください。

フレームの構造

上記を踏まえた上で、Unity がどのようにフレームを構築し、このプロセスで CPU (中央処理装置) と GPU (画像処理装置) がどのように連携するかを探ってみましょう。60 fps を目標にすると 1 フレームあたり 16.66 ms になりますが、Unity では実際のところ、CPU と GPU によって異なるフレームが同時に処理されるパイプラインが維持されます。



CPU はレンダリング命令を準備し、GPU に渡します。

CPU 側では、実行は Unity の内部エンジンコード (ユーザーの制御外) から始まり、その後にカスタムゲームロジック (ユーザーのスクリプト) が続きます。その後、CPU がレンダリング命令を準備します。準備ができると GPU に渡します。GPU がフレーム N のレンダリングを開始する間、CPU はすでに次のフレーム (N+1 と表します) の処理を行っています。

Unity はデュアルスレッドシステムを使用して、このワークフローを効率化します。

- メインスレッドは、ゲームロジック、物理演算、アニメーション、入力を処理しつつ、レンダリングコマンドをキューに入れます。
- レンダースレッドは、これらのコマンドを GPU に適した命令に変換します。

レンダリング命令を受け取った GPU は、グラフィックスパイプラインを通じてこの命令を処理し、頂点シェーディング、フラグメントシェーディング、ポストプロセッシングなどのタスクを実行し、最後にフレームをディスプレイに出力します。

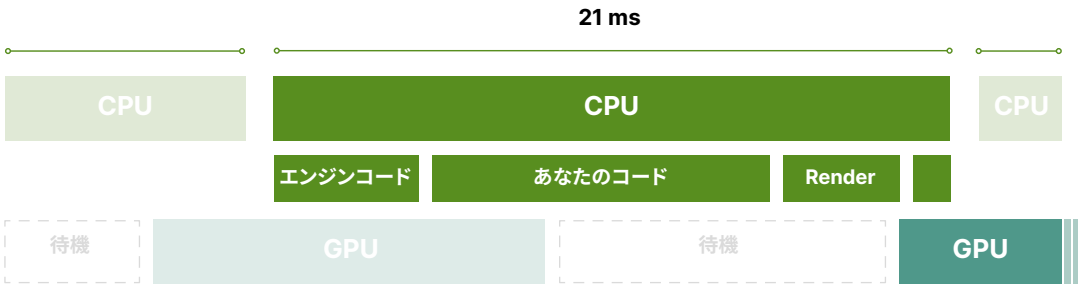
この並列化アプローチにより、GPU が現在のフレームをレンダリングしている間に、CPU が次のフレームの準備を開始できます。しかし、GPU は CPU がレンダリングデータの準備を終えるまで待たなければならず、適切なパフォーマンスを得るためにはこの 2 つの同期が不可欠です。

フレームの構造をより深く理解するには、Event function execution order (イベント関数実行順) を参照してください。このページでは、各フレームの処理中に発生する入力処理、物理演算、レンダリング、GUI更新などの一連の操作の概要を説明しています。

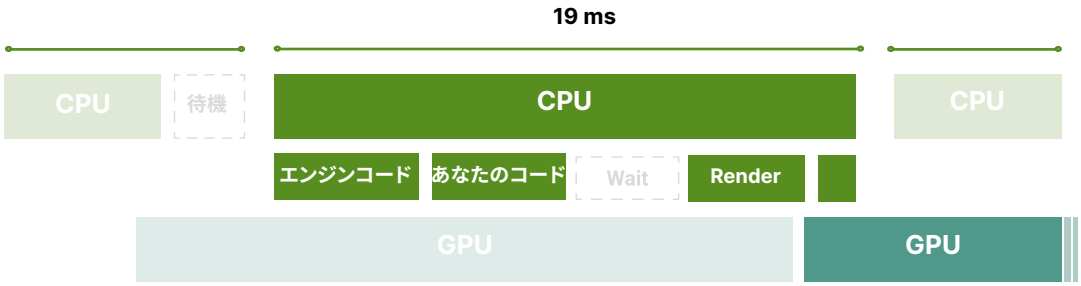
GPU 依存か CPU 依存かを理解する

CPU 依存とGPU 依存とは、システム内のどの部分がゲームのパフォーマンスを制限しているかを指し、最適化 のプロセスをどこから始めるべきかを理解する鍵となります。

CPU 依存 とは、CPU がボトルネックであることを意味します。スクリプト、物理演算、ドローコール管理などのタスクの処理に、GPU よりも時間がかかります。この場合、GPU 設定を最適化してもフレームレートは改善されません。ボトルネックを取り除くには、CPU の負荷を減らす必要があります。

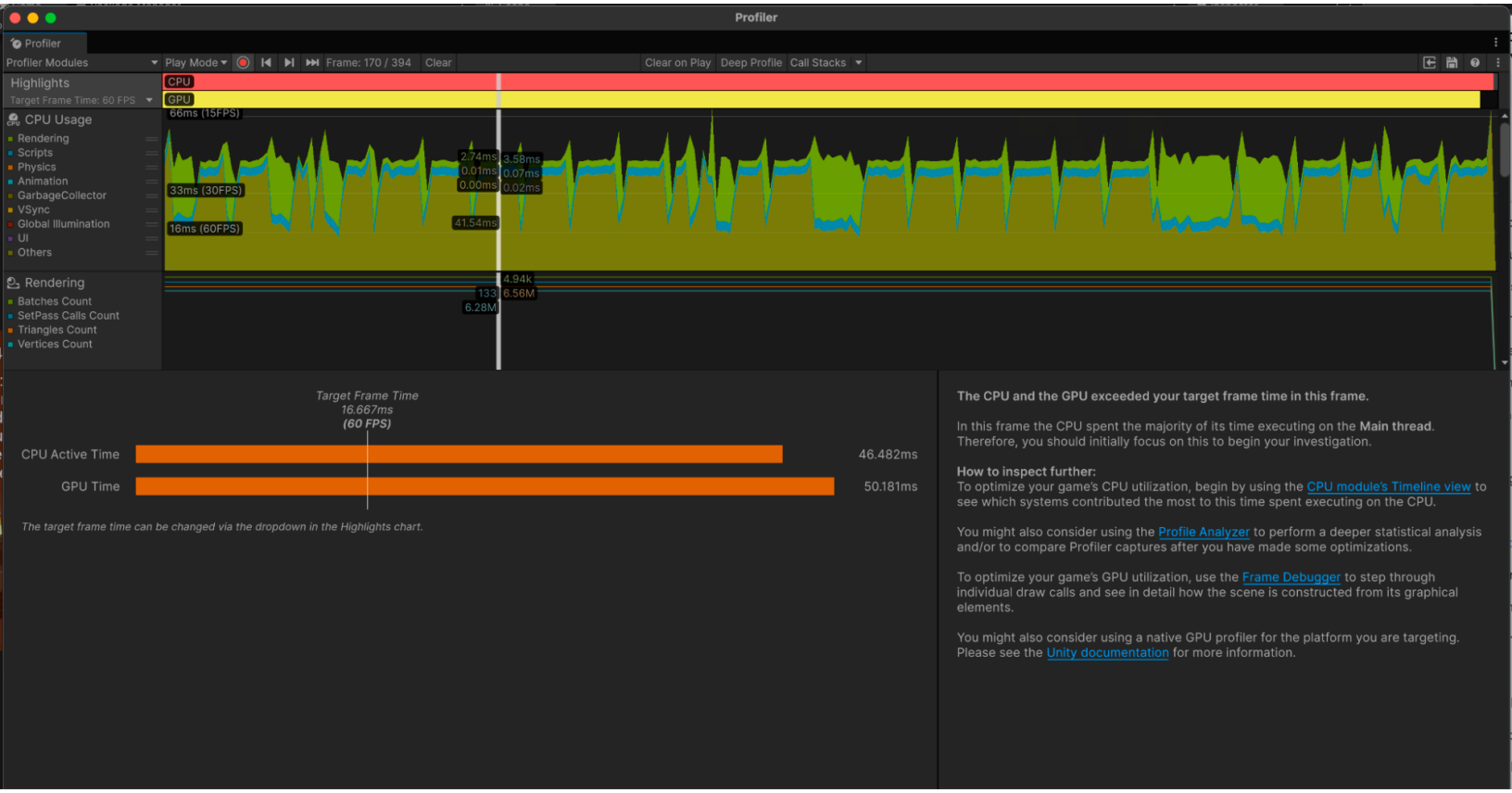


GPU 依存 とは、GPU がボトルネックであることを意味します。CPU がロジックに費やしている時間よりも、グラフィックスのレンダリングに時間がかかっています。ここでは、CPU のコードを最適化してもフレームレートは改善されないため、シェーダーを簡素化したり、ライティングの複雑さを軽減したり、解像度を下げてパフォーマンスを向上させたりする必要があります。



プロファイラーでは、この待機は `Gfx.WaitForPresent` として表示されます

Unity 6 の新しい Highlights モジュール を使用すると、すべてのプラットフォームでアプリケーションが CPU 依存か GPU 依存であるかを簡単に判断できます。



Highlights モジュールを使用することで、設定した目標フレーム時間に対するゲームのパフォーマンスを簡単に把握できます。この例では、目標の 60 fps を達成するために、CPU と GPU の両方で多くの最適化作業が必要です。

VSync とは

VSync は、アプリケーションのフレームレートとモニターのリフレッシュレートを同期します。つまり、モニターが 60 Hz で、ゲームが 16.66 ms のフレームバジェット内で実行される場合、より速く実行するのではなく、60 fps での実行が強制されます。1 秒あたりのフレーム数をモニターのリフレッシュレートと同期させることで、GPU の負荷が軽減され、画面ティアリング などの視覚的なアーティファクトがなくなります。Unity では、**VSync Count** を Quality settings (**Edit > Project Settings > Quality**) のプロパティとして設定できます。

Unity でのプロファイリングについて

Unity の プロファイリングツール は、エディターと Package Manager からアクセスできます。これらのツールと Unity フレームデバッガー については「Unity のプロファイリングツールおよびデバッグツール」のセクションで詳しく説明していますが、ここでも簡単に概要を説明します。

- Unity プロファイラー は、Unity エディターのパフォーマンスを測定し、デバイスに接続している間は再生モードまたは開発モードでのアプリケーションのパフォーマンスを測定します。
- Profiling Core パッケージ は、Unity プロファイラーのキャプチャにコンテキスト情報を加えるために使用できる API を提供します。
- Memory Profiler は、ゲームが使用しているメモリ量と、そのメモリを使用しているオブジェクトを詳細に分析します。
- Profile Analyzer を使用すると、2 つのプロファイリングデータセットを並べて比較し、変更がアプリケーションのパフォーマンスにどのように影響するかを分析できます。
- Project Auditor は、プロジェクト内のスクリプト、アセット、コードに関する情報や問題を報告します。その多くはパフォーマンスに関するものです。

また、Unity は一連のプロファイリングツールを補完するデバッグツールもいくつか提供しています。例えば、レンダリングデバッガー の Display Stats パネルでは、エディターを接続しなくても、開発ビルドのパフォーマンス数値とマーカー (CPU + GPU) を限定的に表示できます。

サンプルベースとインストルメンテーションベースのプロファイリングの比較

ゲームのパフォーマンスのプロファイリングには、一般的な方法が 2 つあります。

- サンプルベースプロファイリング
- インストルメンテーションプロファイリング

サンプルベースプロファイリング

サンプルベースプロファイリングは、コードが定期的に (通常はミリ秒単位で) 行っていることのスナップショットを一定の周期で取得することで機能します。これらのスナップショットを分析することで、コードのどの部分が最も頻繁に実行されており、最も時間がかかっているかを把握できます。一般的に、オーバーヘッドは低いものの、集約されたスナップショットをキャプチャするので、データは全体像を捉えています。サンプリング頻度を増やすと、より多くのデータを得ることができますが、インストルメンテーションベースのプロファイラーのような精度は得られません。

サンプリングプロファイラーは通常、プラットフォームインフラストラクチャを使用して、最小限のオーバーヘッドと最大のサンプリングレートを可能にしています。このようなプロファイラーの例としては、Windows Performance Analyzer と Event Tracing for Windows、Instruments、Android Studio の組み合わせが挙げられます。

インストルメンテーションベースのプロファイラー

インストルメンテーションベースのプロファイリングでは、プロファイラーマーカー を加えてコードを “インストルメント化” します。このマーカーは、各マーカー内のコードの実行時間の長さに関する詳細なタイミング情報を記録します。このプロファイラーは、各マーカーの開始イベントと終了イベントのタイムスタンプのストリームをキャプチャします。情報を失うことはありませんが、プロファイリングデータをキャプチャするには、マーカーが配置されている必要があります。

これにより、カスタムのプロファイラーマーカーを加えたり、ディーププロファイリング を使用したりして、コードのパフォーマンスを調査し、パフォーマンスの問題を簡単に特定できるほか、最適化のポイントをすぐに見つけることができます。これはまた、サンプルベースのプロファイリングに比べてオーバーヘッドが高くなることを意味しますが、具体的な問題を特定するための非常に正確な情報をキャプチャできることも意味します。

ディーププロファイリングでは、すべてのスクリプティングメソッドの呼び出しに自動的に開始マーカーと終了マーカーが挿入されます。C# Getter プロパティや Setter プロパティなどでも同様です。このシステムは、スクリプティング側ではプロファイリングの詳細を完全に提供しますが、キャプチャされたプロファイリングスコープ内の呼び出しの数に応じて関連するオーバーヘッドが生じ、これにより報告されるタイミングデータ量が増大する可能性があります。

インストルメンテーションベースとサンプリングベースの比較

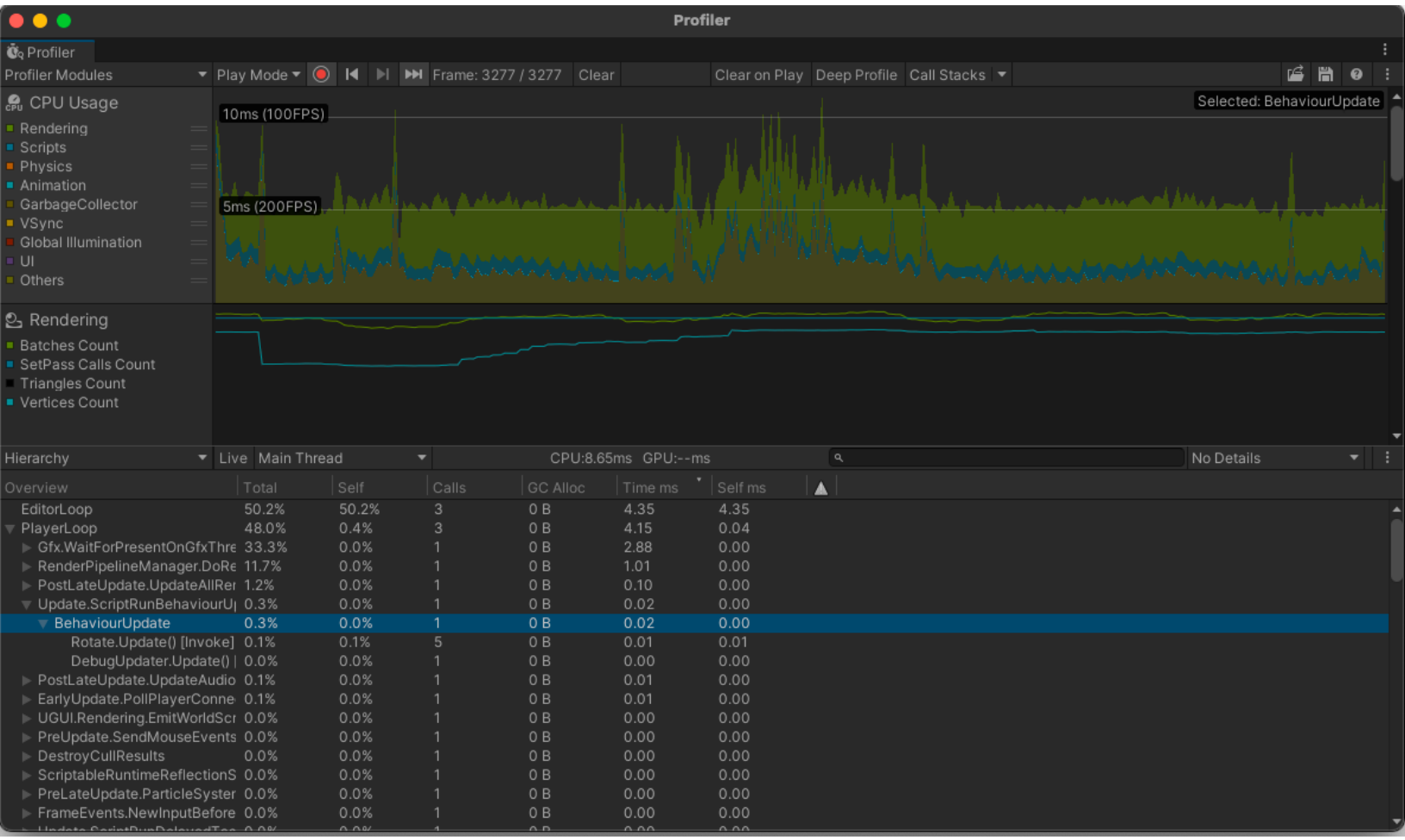
一般的に、サンプルベースプロファイリングはアプリケーションのパフォーマンスの全体像を分析します。一方でインストルメンテーションベースのプロファイリングは重要なパフォーマンスを特定しますが、オーバーヘッドが高くなります。

サンプリングベースのプロファイラーがもたらすオーバーヘッドは、プロファイリング中の CPU の処理に関係なく一定です。このオーバーヘッドに対応できるようにサンプルレートを変更することもできますが、一般的にはより低くなります。インストルメンテーションプロファイリングによって生じるオーバーヘッドは、マーカーの数によって異なります (つまり、マーカーを多数施すと、マーカー自体に時間がかかるため、キャプチャの負荷が高くなります)。インストルメント化プロファイラーを使用する際は、この点に注意してください。プロジェクト内で多くの関数を呼び出す場所は、実際よりもコストが高く見える可能性があるためです。これは特にディーププロファイリング時に現れ、プロファイラーのキャプチャのタイミングを歪める可能性があります。

Unity でのインストルメンテーションベースのプロファイリング

Unity プロファイラーは、使用しているモードに応じて、インストルメンテーションベース とサンプルベースのプロファイリングの両方を組み合わせます。詳細なプロファイリングとオーバーヘッドの適切なバランスは、Unity API サーフェスのほとんどに設定されているマーカーによって実現されています。重要なネイティブ機能とスクリプティングコードベースのメッセージ呼び出しは、大きなオーバーヘッドを発生させずに最も重要な "概要" を把握するためにインストルメント化されています。

上記のスクリプティングコードベースのメッセージ呼び出し (デフォルトでは明示的にインストルメント化されます) には通常、Unity ネイティブコードからマネージコードへの呼び出しの最初のコールスタック深度が含まれます。例えば、Start()、Update()、FixedUpdate() などの一般的な MonoBehaviour メソッドです。



サンプルスクリプトをプロファイリングすると、Update() メソッドの呼び出しが表示されます。

プロファイラーでは、Unity の API にコールバックするマネージスクリプティングコードの子サンプルを確認することもできます。ただし、1 つ注意点があります。この Unity API コードに、インストルメンテーションプロファイラーマーカー自体が含まれている必要がある点です。パフォーマンスのオーバーヘッドを伴うほとんどの Unity API はインストルメント化されます。例えば、**Camera.main** を使用すると、プロファイリングのキャプチャに **FindMainCamera** マーカーが表示されます。キャプチャされたプロファイリングデータセットを調べる際には、各種マーカーが何を意味するかを知っていると便利です。詳細については、一般的なプロファイラーマーカーの 一覧 を参照してください。

プロファイラーマーカーでプロファイリングの詳細度を上げる

デフォルトでは、Unity プロファイラーは、プロファイラーマーカー で明示的にラップされたコードのタイミングをプロファイリングします。プロファイラーマーカーをコード内の重要な関数に手動で挿入することで、プロファイリングの実行の詳細度を効率的に高めることができます。独自のマーカーを加えることで、ディーププロファイリング に伴うすべてのオーバーヘッドや、キャプチャ内の時間の不正確さの問題を回避できます。

ディーププロファイリングが有効になっている場合、Unity はインストルメンテーションベースのプロファイリングを使用します。これにより、すべての関数呼び出しに対して正確で詳細なデータを収集するための追加命令が挿入され、ランタイムオーバーヘッドが高くなる代わりに、実行時間と動作の測定が可能になります。

プロファイラーモジュール

プロファイラーはフレームごとのパフォーマンス指標をキャプチャし、ボトルネックを特定するのに役立ちます。**CPU 使用率**、**GPU**、**レンダリング**、**メモリ**、**物理演算** など、プロファイラーに含まれるモジュールを使用して詳細を掘り下げます。

メインの Profiler ウィンドウには、左側にモジュール、下部に詳細パネルが表示されます。

Profiler ウィンドウでは、現在選択されている プロファイラーモジュール でキャプチャされた詳細がビュー下部のパネルに一覧表示されます。例えば、CPU Usage Profiler モジュール は、CPU の処理の **Timeline** (タイムライン) または **Hierarchy** (階層) ビューを、具体的な時間とともに表示します。

CPU Usage モジュールで利用できる Timeline (タイムライン) ビューには、Main Thread (メインスレッド) と Render Thread (レンダースレッド) のマーカーの詳細が表示されます。

Unity プロファイラーを使用してアプリケーションのパフォーマンスを評価し、具体的な領域や問題を掘り下げましょう。デフォルトでは、プロファイラーは Unity エディターの Player インスタンスに接続します。

エディターでのプロファイリングとスタンドアロンビルドでのプロファイリングでは、パフォーマンスに大きな差が生じることに注意してください。プロファイラーの接続先には、ターゲットハードウェア上で直接実行されるスタンドアロンビルドが常に推奨されます。エディターのオーバーヘッドなしで非常に正確な結果が得られるためです。

プロファイリングのワークフロー

このセクションでは、プロファイリング時に便利な目標や、CPU 依存や GPU 依存などの一般的なパフォーマンスのボトルネックについて説明します。これらの状況を特定し、より詳細に調査する方法を学ぶことになります。また、メモリプロファイリングについても取り上げます。メモリプロファイリングはランタイムのパフォーマンスとはほとんど関係ありませんが、ゲームのクラッシュを防ぐことができるので、知っておくことが重要です。

全体的なファイリングから詳細のプロファイリングまで

プロファイリングを行う場合は、最も大きな影響を生みだせる領域に、時間と労力を集中させる必要があります。したがって、プロファイリングの際には上から下へのアプローチから始めることをおすすめします。つまり、レンダリング、スクリプト、物理演算、ガベージコレクション (GC) 割り当てなどのカテゴリの全体像をまず把握する方法です。対象とする領域を特定したら、さらに深く掘り下げることができます。この高レベルパスを使用してデータを収集し、コアゲームループで不要なマネージ割り当てや過剰な CPU 使用率を引き起こすシナリオなど、特に重要なパフォーマンスの問題について確認しましょう。

まず、GC.Alloc マーカーのコールスタックを収集する必要があります。このプロセスに馴染みのない方は、アプリケーションの生存期間にわたって繰り返し発生するメモリ割り当ての特定 というタイトルのセクションでヒントやコツを参照してください。

レポートされたコールスタックの詳細が十分でなく、割り当てやその他の速度低下の原因を突き止められない場合は、ディーププロファイリングを有効にして 2 回目のプロファイリングセッションを実行することで、割り当ての原因を探ることができます。ディーププロファイリング についてはこのガイドの後半で詳しく説明しますが、要約すると、すべての関数呼び出しの詳細なパフォーマンスデータをキャプチャするプロファイラーのモードであり、実行時間と動作を詳細に把握できますが、標準的なプロファイリングに比べてオーバーヘッドが大幅に高くなります。

フレーム時間の "超過原因" について情報を集めるときは、フレームの他の部分との比較に注目してください。ディーププロファイリングを有効にすると、すべてのメソッドコールがインストルメント化されることでオーバーヘッドが著しく増加するため、この相対的な影響が歪む可能性があります。

早期のプロファイリング

プロジェクトの開発サイクル全体を通して常にプロファイリングを行う必要がありますが、初期段階でプロファイリングを始めることで最も大きなメリットを得られます。

早い段階で頻繁にプロファイリングを行い、ベンチマークに使用できるプロジェクトの "パフォーマンスシグネチャ" をチーム全員が理解して、覚えてしまうくらいにしましょう。パフォーマンスが大幅に低下した場合に、問題が生じていることにすぐに気が付き、その問題を修正できるようになります。

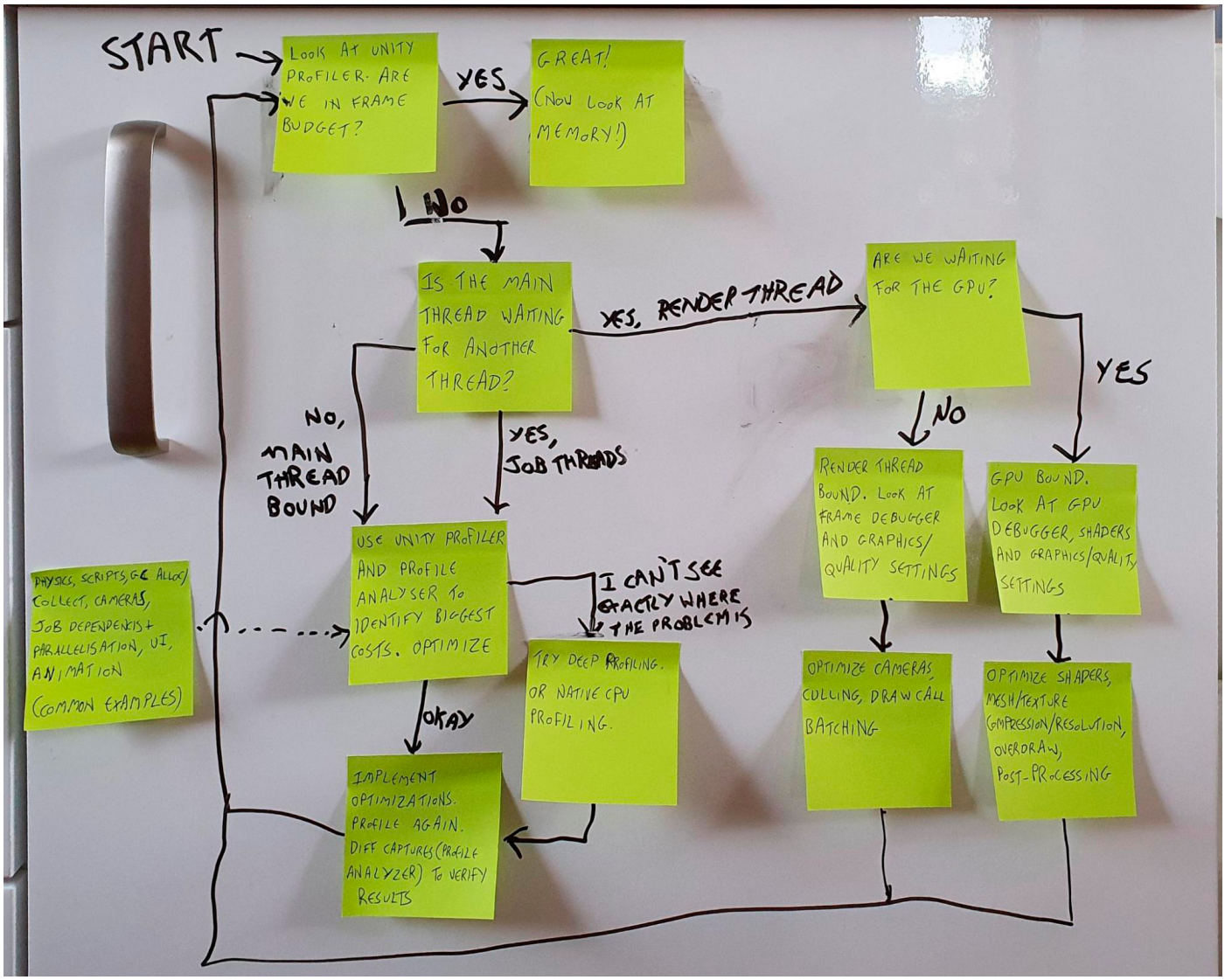
主な問題はエディターでのプロファイリングで簡単に特定できます。しかし、最も正確なプロファイリング結果は常に、ターゲットデバイスでビルドを実行してプロファイリングを行い、さらにプラットフォーム固有のツールを活用して各プラットフォームのハードウェア特性を掘り下げることで得られます。この組み合わせにより、すべてのターゲットデバイスでのアプリケーションパフォーマンスを包括的に把握できます。例えば、一部のモバイルデバイスでは GPU 依存であっても、別のモバイルデバイスでは CPU 依存である場合があり、これは該当するデバイスで測定しないとわかりません。

プロファイリング方法の確立

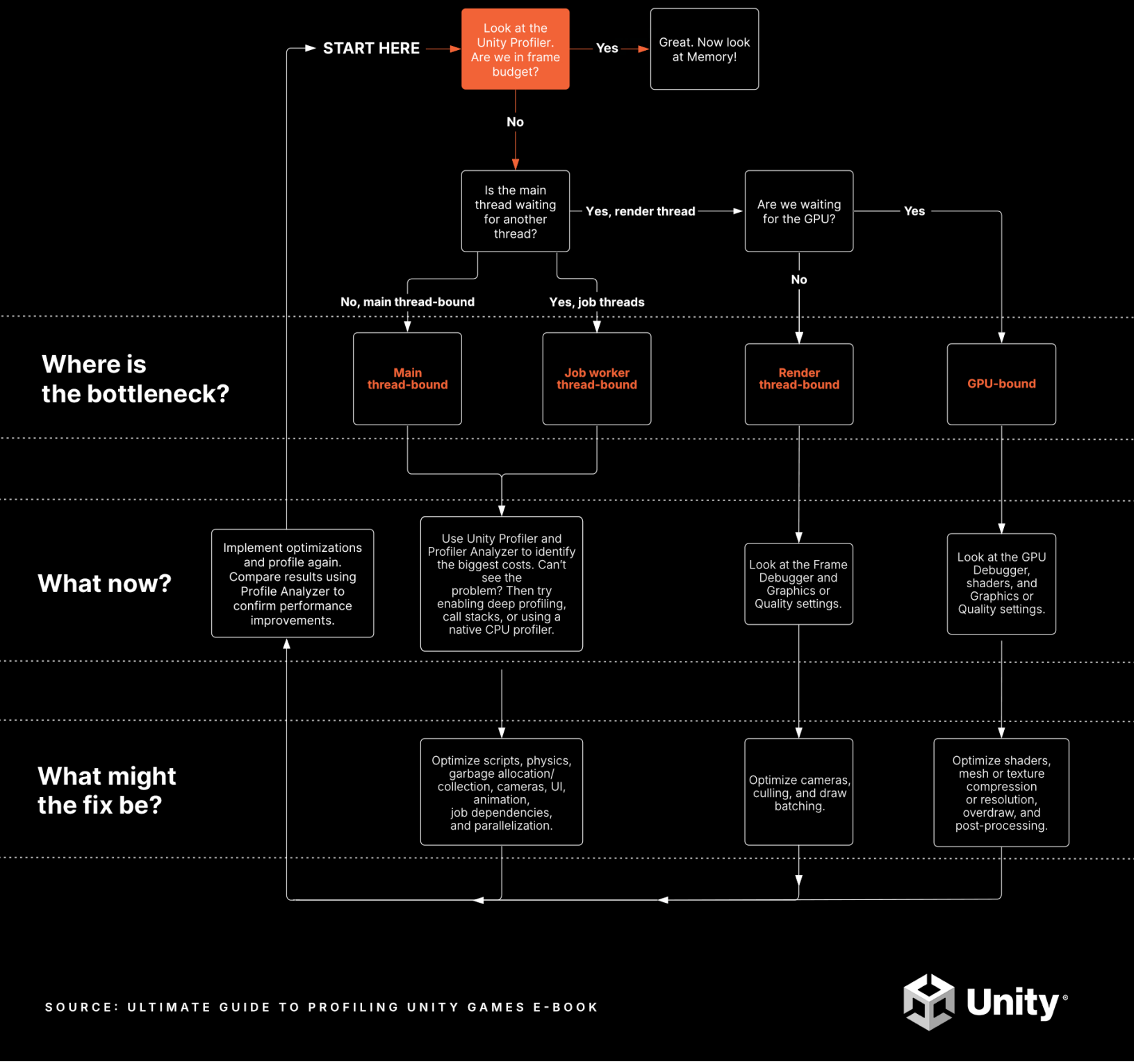
プロファイリングは構造化されたプロセスであるべきです。問題を無作為に把握するのはよくありません。そのため、プロファイリングの方法論を確立するのが良いでしょう。

プロファイリングの意義は、ボトルネックを最適化のターゲットとして特定することにあります。当て推量に頼ると、ボトルネックになっていない部分を最適化することになり、全体的なパフォーマンスはほとんど、またはまったく向上しません。"最適化" の中には、ゲーム全体のパフォーマンスを悪化させるものもあります。さらに、大きな労力を費やしても取るに足らない結果しか得られない場合もあります。重要なのは、集中的な時間投資の効果を最適化することです。

以下のフローチャートは最初のプロファイリングプロセスを示しており、その後のセクションで各ステップの詳細を説明しています。また、実際の Unity プロジェクトからのプロファイラーのキャプチャも紹介し、注目すべきポイントを説明しています。



Follow this flowchart and use the Profiler to help pinpoint where to focus your optimization efforts:



SOURCE: ULTIMATE GUIDE TO PROFILING UNITY GAMES E-BOOK

このフローチャートに従い、プロファイラーを使用して、最適化の労力を集中させるポイントを特定しましょう。

すべての CPU アクティビティの全体像 (GPU の処理を待つ時間を含む) をつかむには、プロファイラーの **CPU モジュール** の **Timeline ビュー** を使用します。キャプチャを正しく解釈するために、一般的なプロファイラーマーカー に慣れるようにしてください。プロファイラーマーカーの中には、ターゲットプラットフォームによって表示が異なるものもあるので、各ターゲットプラットフォームでゲームのキャプチャを調べ、プロジェクトにおける "通常の" キャプチャがどのように見えるかを確認してください。

プロジェクトのパフォーマンスは、最も時間がかかるチップおよび/またはスレッドに左右されます。最適化の取り組みは、そこに注力するべきです。例えば、目標フレーム時間バジェットが 33.33 ms で、VSync が有効になっているゲームについて、以下のシナリオを想像してみてください。

- CPU フレーム時間 (VSync を除く) が 25 ms、GPU 時間が 20 ms であれば問題ありません。CPU 依存ですが、すべてがバジェット内であり、最適化してもフレームレートは改善しません (CPU と GPU の両方が 16.66 ms 未満になり、60 fps まで跳ね上がる場合を除く)。
- CPU のフレーム時間が 40 ms、GPU が 20 ms の場合、CPU 依存であり、かつ CPU のパフォーマンスを最適化する必要があります。GPU のパフォーマンスを最適化しても効果はありません。実際、必要に応じて C# コードの代わりにコンピュートシェーダーを使用してバランスを取るなど、CPU の負荷の一部を GPU に移すことをおすすめします。
- CPU のフレーム時間が 20 ms、GPU が 40 ms の場合、GPU 依存であり、かつ GPU の処理を最適化する必要があります。
- CPU と GPU の両方が 40 ms の場合、両方に依存しており、30 fps に到達するには両方とも 33.33 ms 未満に最適化する必要があります。

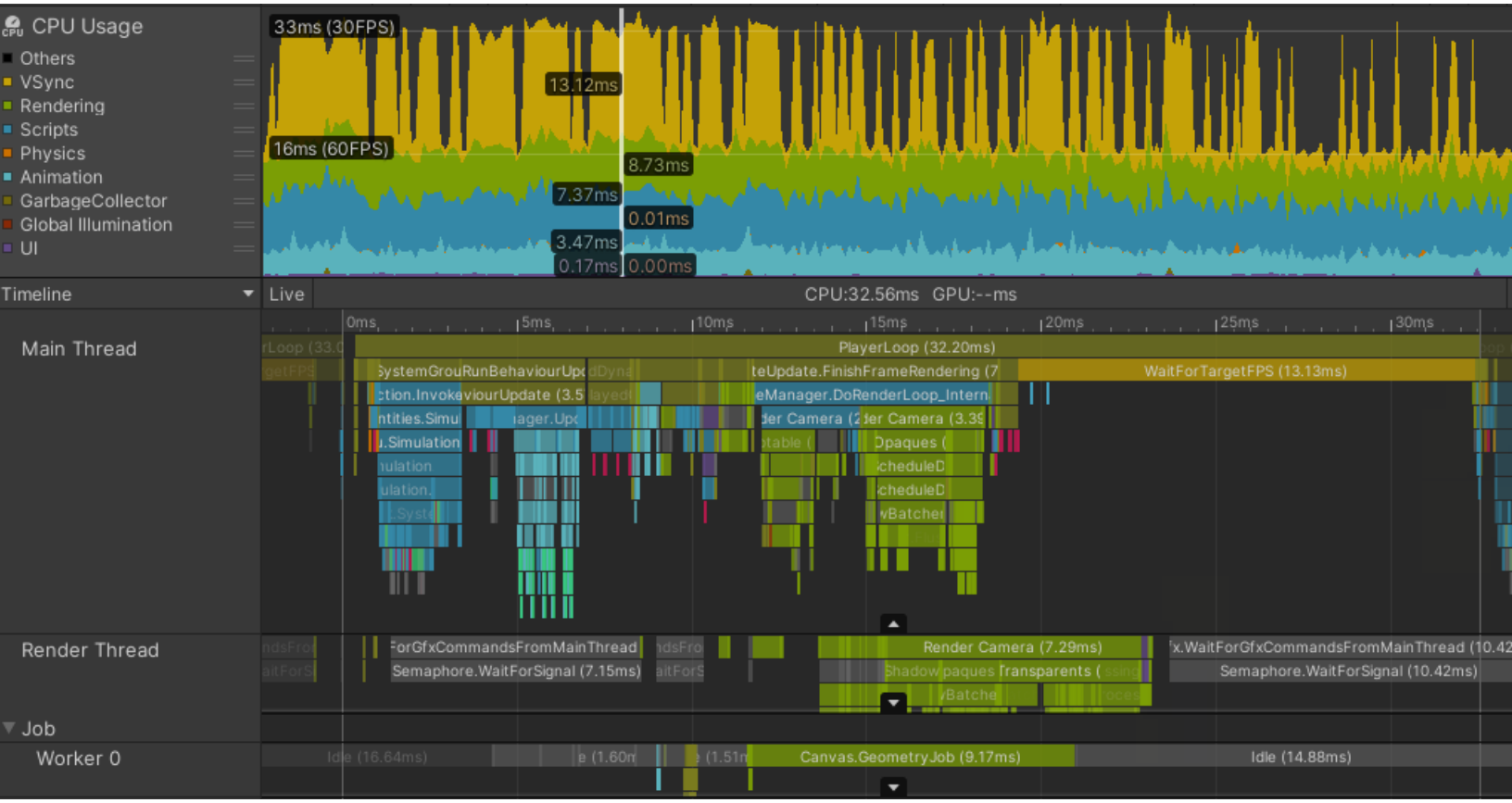
CPU 依存または GPU 依存であることについて、詳しくは以下のリソースを参照してください。
YouTube自動音声翻訳/字幕推奨

- フレームの構造、CPU と GPU
- Is your game draw call-bound? (ゲームがドローコードに依存していますか?)

フレームバジェット内かどうか

開発の初期段階から頻繁にプロジェクトのプロファイリングと最適化を行うことで、アプリケーションのすべての CPU スレッドと GPU フレーム時間を、より確実にフレームバジェット内に収められるようになります。このプロセスの指針になる問いは、フレームバジェット内に収まっているかどうかです。

下の画像は、継続的なプロファイリングと最適化を行ったチームが Unity を使用して開発したモバイルゲームのプロファイルキャプチャです。このゲームは、ハイスペックのスマートフォンでは 60 fps、このキャプチャのような中スペック/低スペックの場合では 30 fps を目標としています。



これは、30 fps に必要な最大 22 ms のフレームバジェット内で、過熱することなく快適に動作するゲームのプロファイルです。**WaitForTargetfps** が VSync までのメインスレッド時間と、レンダースレッドとワーカースレッドのグレーのアイドル時間をパディングしていることに注意してください。VBlank 間隔は、フレーム間で **Gfx.Present** の終了時間を見ることで確認できます。また、Timeline ビューまたは上部のタイムルーラーで時間スケールを作成して、ある時間スケールから次の時間スケールまでを測定できます。

選択したフレームの時間の半分近くが、黄色の **WaitForTargetFPS** プロファイラーマーカーで占められていることに注意してください。アプリケーションで **Application.targetFrameRate** が 30 fps に設定され、**VSync** が有効になっています。メインスレッドでの実際の処理は 19 ms 付近で終了し、残りの時間は 33.33 ms が経過して次のフレームを開始するまで待機となります。この時間はプロファイラーマーカーで表されますが、実質的にメイン CPU スレッドはアイドル状態であり、この間に CPU が冷却されます。また、最小限のバッテリー電力しか消費されません。

注意すべきマーカーは、プラットフォームが異なる場合、または VSync が無効になっている場合は異なる可能性があります。重要なのは、メインスレッドがフレームバジェット内で実行されているか、フレームバジェットちょうどで実行されているかを、アプリケーションが VSync を待機していることや他のスレッドにアイドル時間があるかどうかを示す何らかのマーカーで確認することです。

アイドル時間は、グレーまたは黄色のプロファイラーマーカーで表されます。上のスクリーンショットは、レンダースレッドが **Gfx.WaitForGfxCommandsFromMainThread** でアイドル状態にあることを示しています。これは、1 つのフレームで GPU へのドローコールの送信が終了し、次のフレームで CPU からの追加のドローコールリクエストを待機している時間を示しています。同様に、**Job Worker 0** スレッドは **Canvas.GeometryJob** をしばらく実行しますが、ほとんどの時間はアイドル状態です。これらはすべて、アプリケーションがフレームバジェット内に余裕を持って収まっていることを示す兆候です。

ゲームがフレームバジェット内である場合

フレームバジェット (バッテリー使用量とサーマルスロットリングを考慮して調節されたバジェットも対象) の範囲内であれば、主要なプロファイリングタスクは完了です。最後に **Memory Profiler** を実行して、アプリケーションがメモリバジェット内にも収まるようにします。

CPU 依存

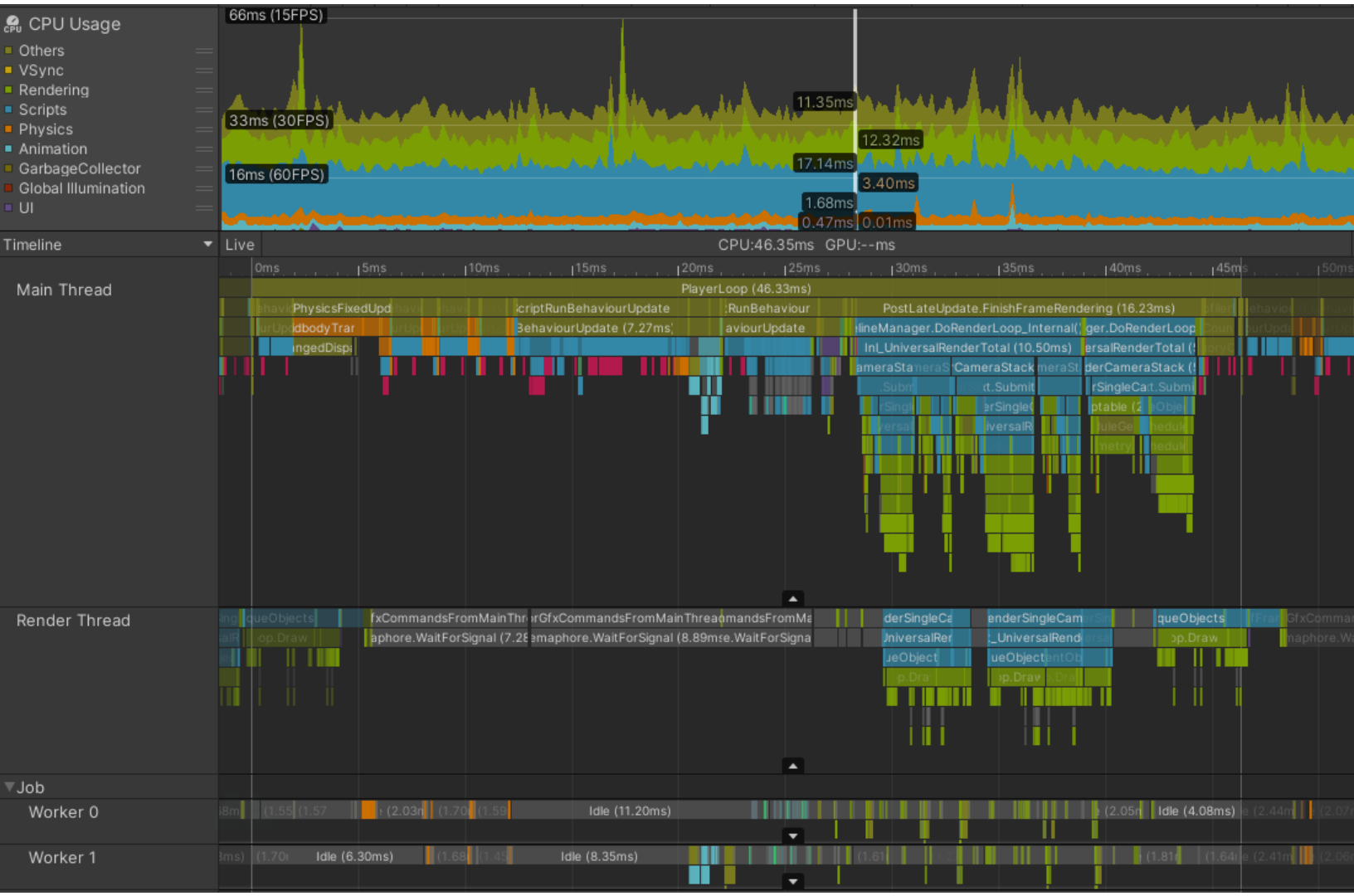
ゲームが CPU のフレームバジェット内に収まらない場合、次のステップは CPU のどの部分がボトルネックになっているか、つまりどのスレッドが最もビジー状態になっているかを調査することです。

CPU 負荷全体がボトルネックになることはあまりありません。最近の CPU には、独立して同時に処理を実行できるさまざまなコアが存在します。各 CPU コアで異なるスレッドを実行することが可能です。完全な Unity アプリケーションはさまざまな目的で異なるスレッドを使用しますが、最も一般的にパフォーマンスの問題が見つかるスレッドは、次のとおりです。

- **メインスレッド**:ゲームロジック/スクリプトの大部分が、デフォルトで、ここで作業を実行します。物理演算、アニメーション、UI、レンダリングの初期段階など、ほとんどの Unity システムはここで実行されます。
- **レンダースレッド**:レンダースレッドでは、GPU にレンダリング命令を送信する前に行う必要がある準備作業 (シーン内のどのオブジェクトがカメラから見えるか、または視錐台の外側にあるか、遮蔽されているか、他の基準によってカリングされているために除外されている/見えないかなど) が処理されます。
    - レンダリングプロセス中、メインスレッドはシーンを調査し、カメラカリング、深度ソート、ドローコールのバッチ処理を実行して、レンダリングする対象のリストを生成します。このリストはレンダースレッドに渡され、Unity 内部のプラットフォームに依存しない表現から、特定のプラットフォームで GPU に指示するために必要なグラフィックス API コール (DirectX、Vulkan、Metal など) に変換されます。
- **ジョブワーカースレッド**:開発者は、ジョブシステム を利用してワーカースレッドで実行される特定の種類の作業をスケジュールすることで、メインスレッドの負荷を軽減できます。Unity のシステムや機能の中には、物理演算、アニメーション、レンダリングなどのジョブシステムを利用するものもあります。

メインスレッドの最適化の実例

下の画像は、メインスレッド (Main Thread) 依存のプロジェクトの様子を示しています。このプロジェクトは Meta Quest 2 で実行されています。通常、このフレームバジェットは 13.88 ms (72 fps) または 8.33 ms (120 fps) を目標としています。これは、VR デバイスで乗り物酔いを防ぐためには高フレームレートが重要であるためです。しかし、このゲームが 30 fps を目指していたとしても、このプロジェクトに問題があるのは明らかです。



メインスレッド依存のプロジェクトからのキャプチャ

レンダースレッド (Render Thread)とワーカースレッド (Worker Thread)の状態はフレームバジェット内の例と似ていますが、メインスレッドはフレーム全体を通してビジー状態なのが明白です。フレーム終了時のわずかなプロファイラーのオーバーヘッドを考慮しても、メインスレッドは 45 ms 以上ビジー状態であり、このプロジェクトのフレームレートは 22 fps 未満です。メインスレッドが VSync を待機していることを示すマーカーはなく、フレーム全体にわたってビジー状態です。

調査の次の段階は、フレームの中で最も時間がかかる部分を特定し、その理由を理解することです。このフレームでは、**PostLateUpdate.FinishFrameRendering** に 16.23 ms かかり、フレーム全体のバジェットを超えています。よく見ると、**Inl_RenderCameraStack** というマーカーのインスタンスが 5 つあり、5 つのカメラがアクティブでシーンをレンダリングしていることがわかります。Unity のすべてのカメラは、カリング、ソート、バッチ処理を含むレンダーパイプライン全体を呼び出すため、このプロジェクトで最優先すべきタスクは、アクティブなカメラ を (可能なら 1 台にまで) 減らすことです。

すべての `MonoBehaviour.Update()` メソッドの実行を含むプロファイラーマーカー **BehaviourUpdate** は、このフレームで 7.27 ms かかります。

Timeline ビューのマゼンタ色の部分は、スクリプトがマネージヒープメモリを割り当てているポイントを示します。**Hierarchy** ビューに切り替え、検索バーに **GC.Alloc** と入力してフィルタリングすると、このメモリの割り当てにはこのフレームで約 0.33 ms かかることがわかります。ただし、これはメモリ割り当てが CPU パフォーマンスに与える影響の、不正確な測定です。

GC.Alloc マーカーでは、一般的なプロファイラーサンプルのように、開始点と終了点を記録して時間が計測されることはありません。オーバーヘッドを最小限に抑えるために、Unity では割り当てのタイムスタンプと割り当てサイズのみが記録されます。

プロファイラーは、プロファイラービューに表示されるようにするためだけに、GC.Alloc マーカーに小さな人工的なサンプル時間を割り当てます。

実際の割り当ては、特にシステムから新しい範囲のメモリをリクエストする必要がある場合、より時間がかかることがあります。影響をより明確に把握するために、割り当てを行うコードの周囲にプロファイラーマーカーを配置します。ディーププロファイリングでは、Timeline ビューのマゼンタ色の GC.Alloc サンプルの隙間が、処理にかかった時間の目安になります。

さらに、新しいメモリを割り当てると、パフォーマンスに悪影響を及ぼす可能性があります。これらを直接、測定し、原因として特定するのはさらに困難です。

- システムに新しいメモリをリクエストすると、モバイルデバイスの電力バジェットに影響し、CPU や GPU の速度が低下する可能性があります。
- 新しいメモリは、CPU の L1 キャッシュへのロードが必要である可能性が高く、そのために既存のキャッシュラインを押し出すことが多くなります。
- インクリメンタルガベージコレクションまたは同期ガベージコレクションは、マネージメモリの既存の空き領域を最終的に超えたときに、すぐに、または少し時間をおいてトリガーされます。

フレームの開始時に **Physics.FixedUpdate** のインスタンスが 4 つ追加されると、合計で 4.57 ms になります。この後、**LateBehaviourUpdate**（MonoBehaviour.LateUpdate() の呼び出し）にかかる時間は 4 ms、**Animator** にかかる時間は約 1 ms になります。このプロジェクトが目標のフレームバジェットとレートを達成するためには、これらのメインスレッドの問題をすべて調査し、適切な最適化を見つける必要があります。

メインスレッドのボトルネックに関するよくある落とし穴

最も時間がかかる部分を最適化することで、パフォーマンスの向上が最も大きくなります。メインスレッド依存のプロジェクトでは、多くの場合、以下の領域での最適化に効果が期待できます。

- 物理演算
- MonoBehaviour スクリプトの更新
- ガベージの割り当てとガベージコレクション
- カメラカリングとメインスレッドでのレンダリング
- 非効率的なドローコールのバッチ処理
- UI の更新、レイアウト、リビルド
- アニメーション

最適化ガイドでは、特に一般的な落とし穴のいくつかを最適化するための実用的なヒントを、豊富に提供しています。

e-book を入手する

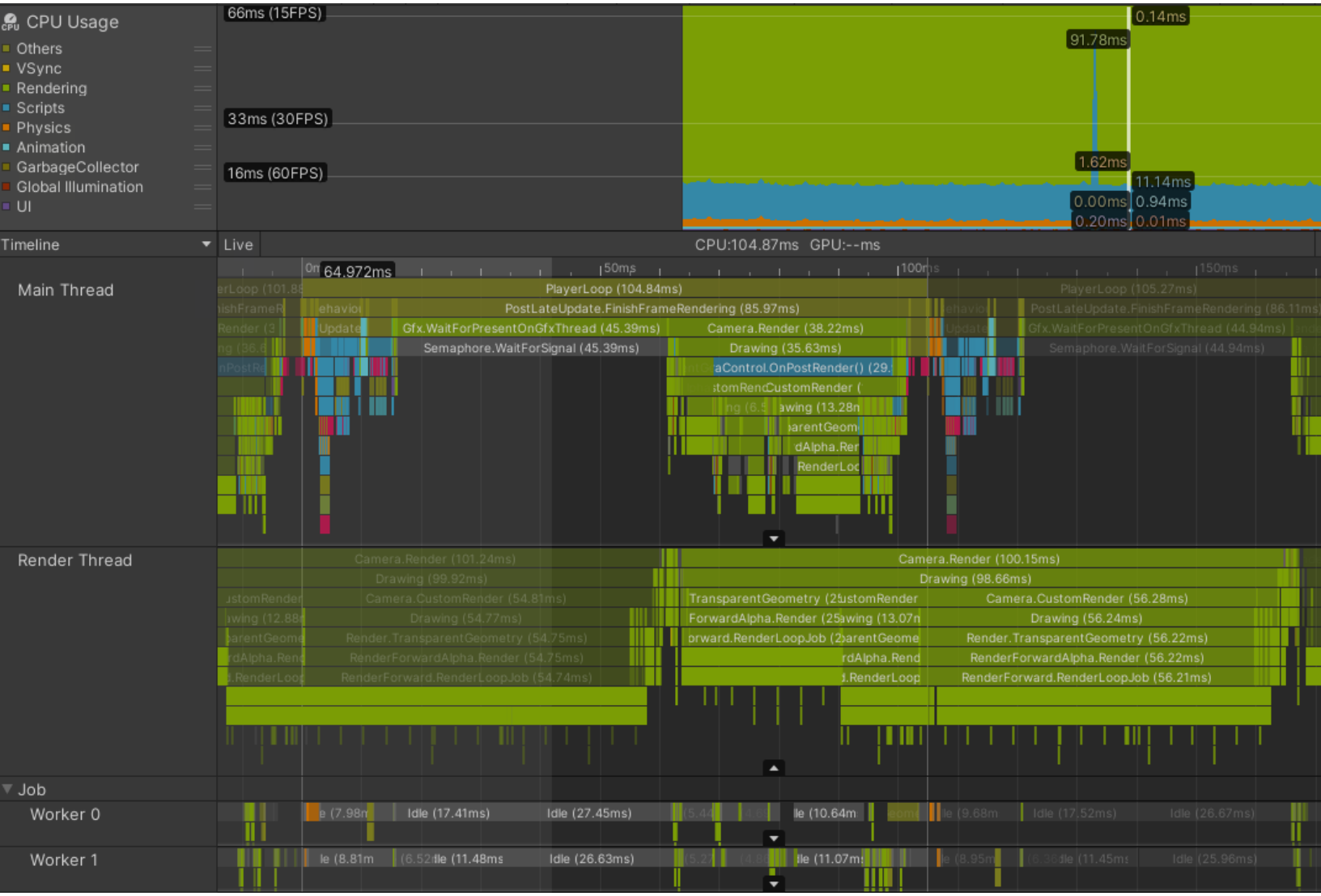
e-book を入手する

調査する問題によっては、他のツールも役立つことがあります。

- MonoBehaviour スクリプトで時間がかかっているのに、その理由がわからない場合は、コードにプロファイラーマーカー を加えるか、ディーププロファイリング を試してコールスタック全体を確認してください。
- マネージメモリを割り当てるスクリプトの場合、アロケーションコールスタック を有効にして、アロケーションがどこから来るかを正確に確認します。ディーププロファイリングを有効にするか、Project Auditor を使用することも可能です。これにより、コードの問題がメモリによってフィルタリングされ、マネージ割り当ての原因となるすべてのコード行を特定できます。
- フレームデバッガー を使用して、ドローコールのバッチ処理がうまくいかなかった原因を調査します。

レンダースレッドの最適化の実例

以下は、レンダースレッド (Render Thread)依存のプロジェクトの実例です。アイソメトリック視点のコンソールゲームで、目標フレームバジェットは 33.33 ms です。



レンダースレッド依存シナリオ

プロファイラーのキャプチャは、現在のフレームでレンダリングを開始する前に、メインスレッドがレンダースレッドを待機することを示しています。これは、**Gfx.WaitForPresentOnGfxThread** マーカーで示されています。レンダースレッドは、前のフレームからのドローコールコマンドをまだ送信しており、メインスレッドからの新しいドローコールを受け入れる準備ができていません。また、**Camera.Render** に時間を費やしています。

他のフレームのマーカーは暗く見えるため、現在のフレームに関連するマーカーと見分けることができます。また、メインスレッドが処理を続行してレンダースレッドによる処理のためにドローコールの発行を開始すると、レンダースレッドが現在のフレームを処理するのに 100 ms 以上かかり、このために次のフレームでもボトルネックが発生していることがわかります。

さらに調査を進めると、このゲームは 9 つの異なるカメラを使用し、シェーダーの交換によって多くの余分なパスが発生するなど、複雑なレンダリング設定になっていることがわかりました。また、このゲームは、フォワードレンダリングパスを使用して 130 を超えるポイントライトをレンダリングしていました。フォワードレンダリングパスでは、各ライトに対して複数の透明なドローコールが追加される場合があります。これらの問題が組み合わさって、1 フレームあたり 3,000 以上のドローコールが生成されていました。

レンダースレッドのボトルネックに関するよくある落とし穴

レンダースレッド依存のプロジェクトにおいて、一般的に調査すべき要因は以下の通りです。

- **不適切なドローコールのバッチ処理**:これは特に OpenGL や DirectX 11 などの古いグラフィックス API に当てはまります。
- **多すぎるカメラ**:画面分割型のマルチプレイヤーゲームを制作している場合を除き、アクティブなカメラは 1 つに絞ったほうが良いでしょう。
- **不適切なカリング**:これにより、描画される要素が多くなりすぎます。カメラの錐台寸法を調査して、レイヤーマスクをカリングしてください。

Rendering Profiler モジュール では、ドローコールバッチと SetPass コールの数がフレームごとに表示されます。レンダースレッドが GPU に発行しているドローコールバッチがどれかを調べるために最適なツールは、フレームデバッガー です。

特定したボトルネックを解決するためのツール

この e-book はパフォーマンスの問題を特定することに焦点を当てていますが、以前に取り上げた 2 つの補完的なパフォーマンス最適化ガイドでは、ターゲットプラットフォームが PC またはコンソール、もしくは モバイル のいずれであるかに応じて、ボトルネックを解決する方法のヒントを紹介しています。レンダースレッドのボトルネックにおいては、特定した問題によって、Unity で利用可能なバッチ処理システムやオプションが異なることを強調しておきます。ここでは、e-book で詳しく取り上げているいくつかのオプションについて簡単に説明します。

- **SRP バッチ処理** は、マテリアルデータを GPU メモリに永続的に保存することで、CPU オーバーヘッドを削減します。実際のドローコール数は減りませんが、各ドローコールのコストは低下します。
- **GPU インスタンシング** は、同じマテリアルを使用する同じメッシュの複数のインスタンスを 1 つのドローコールにまとめます。
- **静的バッチ処理** は、同じマテリアルを共有する静的 (非移動) メッシュを組み合わせます。そのため、静的要素の多いレベルデザインで作業する場合に効果的です。
- **GPU 常駐ドロワー** は、類似のゲームオブジェクトをグループ化することで、自動的に GPU インスタンシングを使用して CPU オーバーヘッドとドローコールを削減します。
- **動的バッチ処理** は、ランタイム時に小さなメッシュを組み合わせます。これはドローコールのコストが高い古いモバイルデバイスでは有利に働く場合があります。ただし、マイナス面は、頂点変換でリソースを大量に消費する可能性もあるという点です。
- **GPU オクルージョンカリング** は、コンピュートシェーダーを使用して、現在のフレームと前のフレームの深度バッファを比較することでオブジェクトの可視性を測り、事前にベイクされたデータを必要とせずに、オクルージョンされたオブジェクトの不要なレンダリングを削減します。

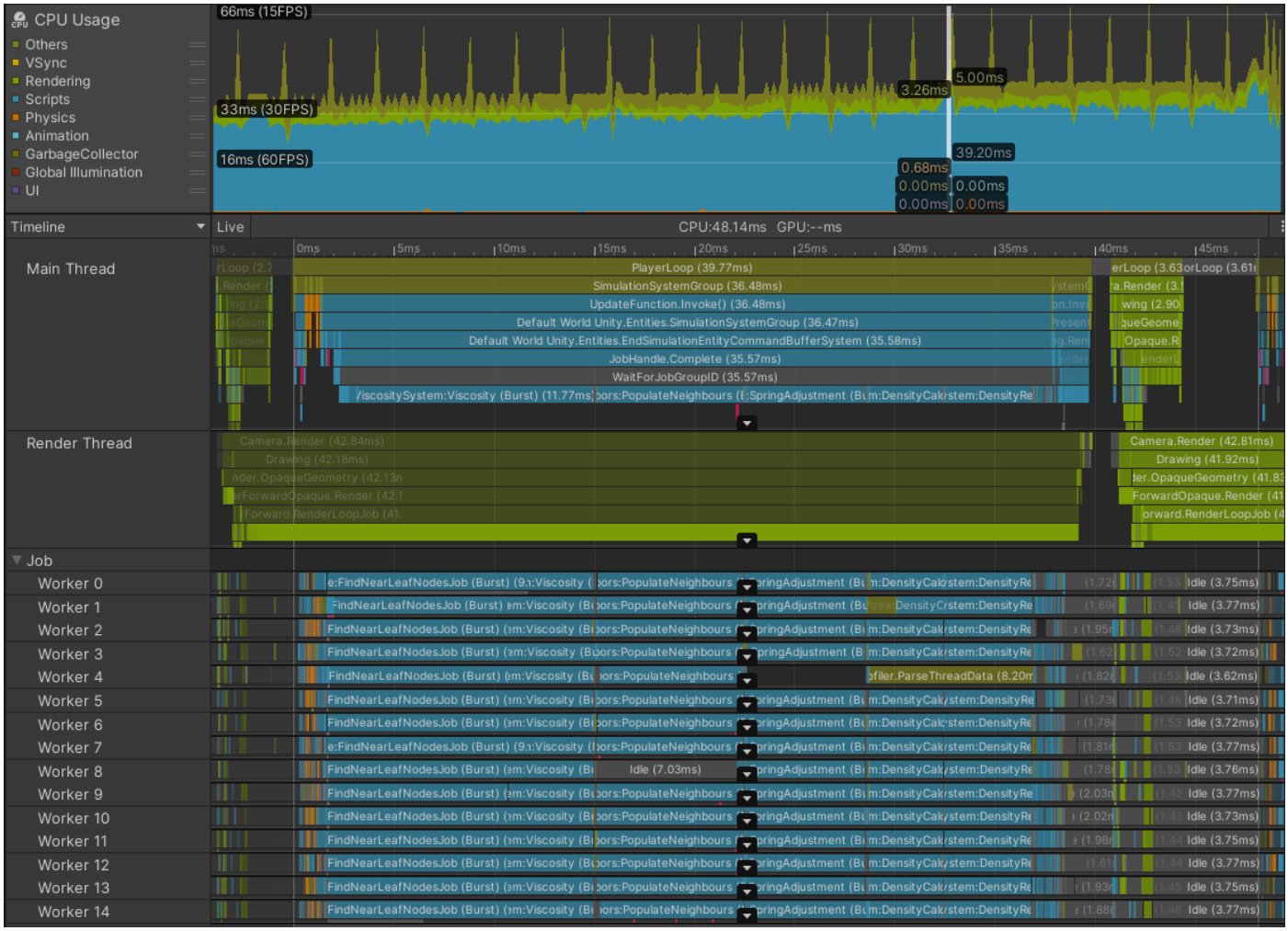
また、CPU 側では、**Camera.layerCullDistances** などの手法を使用してカメラからの距離に基づいてオブジェクトをカリングすることで、レンダースレッドに送信されるオブジェクトの数を減らし、カメラカリング時の CPU ボトルネックを軽減できます。

これらは利用可能なオプションの一部に過ぎません。それぞれには異なる長所と短所があります。一部のオプションは特定のプラットフォームに限定されます。プロジェクトでは、多くの場合、これらのシステムをいくつか組み合わせて使用する必要があります。そのためには、これらを最大限に活用する方法を理解しなくてはなりません。

ワーカースレッド

プロジェクトがメインスレッドやレンダースレッド以外の CPU スレッドに依存する状況はそれほど一般的ではありません。ただし、プロジェクトで Data-Oriented Technology Stack (DOTS) を使用している場合、特にジョブシステム を使用してメインスレッド (Main Thread)からワーカースレッド (Worker)に作業を移動する際に発生する可能性があります。

こちらはエディターの再生モードからのキャプチャです。DOTS プロジェクトが CPU 上でパーティクル流体シミュレーションを実行している様子を示しています。



ワーカースレッド依存の、シミュレーション負荷の高い DOTS ベースのプロジェクト

一見すると成功しているように見えます。ワーカースレッドには、Burst コンパイル されたジョブがぎっしりと詰まっており、大量の処理がメインスレッドから移動されたことを示しています。通常、これは適切な判断と言えます。

しかし、この場合、フレーム時間 48.14 ms とメインスレッドのグレーの **WaitForJobGroupID** マーカー 35.57 ms が、何かがおかしいことを示しています。WaitForJobGroupID は、メインスレッドがワーカースレッドで非同期的に実行されるジョブをスケジュールしていることを示しますが、ワーカースレッドがジョブの実行を終了する前に、それらのジョブの結果が必要になります。WaitForJobGroupID の下にある青いプロファイラーマーカーは、メインスレッドは待機中にジョブを実行し、ジョブがより早く終了するようにしていることを示しています。

ジョブは Burst コンパイルされていますが、まだ多くの処理が行われています。このプロジェクトでは、どのパーティクル同士が近くにあるかを素早く見つけるために使用されている空間クエリ構造を最適化するか、より効率的な構造に取り替えるべきかもしれません。または、空間クエリジョブをフレームの始めではなく終わりにスケジュールし、その結果が次のフレームの開始時まで不要になるようにすることができます。このプロジェクトでは、シミュレートしようとしているパーティクルが多すぎるのかもしれません。解決策を見つけるにはジョブのコードをさらに分析する必要があるため、より細かいプロファイラーマーカーを施すことで、最も遅い部分を特定しやすくできます。

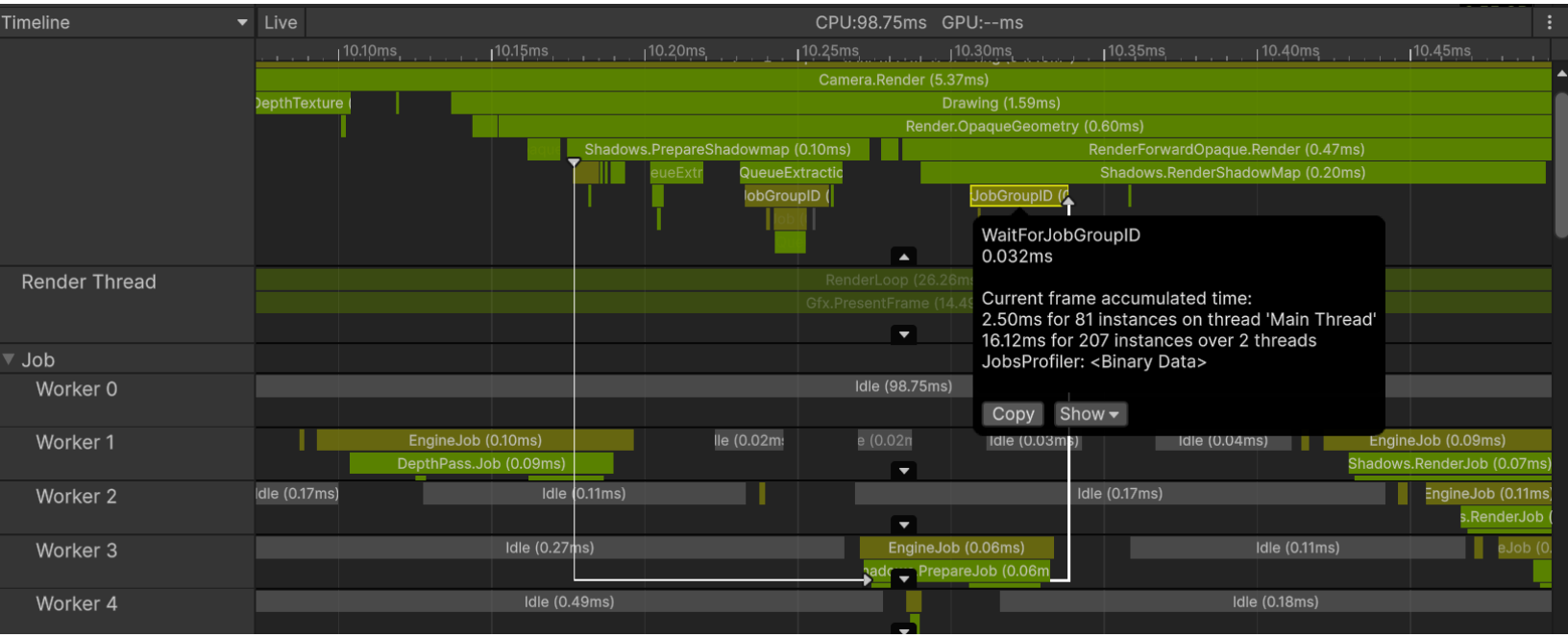
プロジェクト内のジョブは、この例ほど並列化されていない可能性があります。1 つのワーカースレッドで 1 つの長いジョブだけが実行されているのかもしれません。ジョブが予定されている時間から完了する必要のある時間までの間隔が、ジョブ実行に十分な長さであれば問題ありません。そうでない場合は、上のスクリーンショットのように、ジョブの完了を待っている間にメインスレッドが停止してしまいます。

ワーカースレッドのボトルネックに関するよくある落とし穴

同期ポイントとワーカースレッドのボトルネックの一般的な原因には、以下などが挙げられます。

- Burst コンパイラーによってコンパイルされていないジョブ
- 複数のワーカースレッド間で並列化するのではなく、1 つのワーカースレッドで長時間実行されるジョブ
- ジョブが予定されているフレーム内の時点から結果が必要な時点までの時間不足
- すべてのジョブを即座に完了する必要がある、フレーム内の複数の "同期ポイント"

CPU Usage Profiler モジュールで Timeline ビューの Flow Events 機能を使用すると、ジョブが予定されているタイミングと、その結果がメインスレッドで必要となるタイミングを調べることができます。



効率的な DOTS コードの記述について、詳しくは Unity Learn の DOTS に関するベストプラクティス ガイドを参照してください。

GPU 依存

メインスレッドが **Gfx.WaitForPresentOnGfxThread** などのプロファイラーマーカーに長時間費やされ、レンダースレッドが **Gfx.PresentFrame** や **<GraphicsAPIName>.WaitForLastPresent** などのマーカーを同時に表示する場合、アプリケーションは GPU 依存になります。

GPU フレーム時間を取得する最善の方法は、ターゲットプラットフォーム固有の GPU プロファイリングツールを使用することですが、すべてのデバイスで信頼性の高いデータを簡単にキャプチャできるわけではありません。

そのような場合、FrameTimingManager API は、CPU と GPU の両方で、低オーバーヘッドで高レベルのフレーム時間を実現できて便利です。

以下は、Vulkan グラフィックス API を使用して Android のスマートフォン上でキャプチャしたものです。この例の Gfx.PresentFrame で費やされた時間の一部は VSync のための待機に関連している可能性がありますが、このプロファイラーマーカーの極端な長さは、この時間の大部分が、GPU による前のフレームのレンダリングが終了するのを待つために費やされていることを示しています。

GPU 依存のモバイルゲームからのキャプチャ

このゲームでは、特定のゲームプレイイベントがトリガーとなって使用されたシェーダーの影響で、GPU によってレンダリングされるドローコールの数が 3 倍になっていました。GPU パフォーマンスをプロファイリングする際に調査すべき一般的な問題には、以下などがあります。

- アンビエントオクルージョンやブルームなどの、高コストな全画面のポストプロセスエフェクト
- 以下のような要因によってコストが高くなったフラグメントシェーダー:
  - シェーダーコード内の分岐ロジック
  - 特にモバイルにおける、半精度ではなく完全な Float 精度の使用
  - GPU のウェーブフロント占有率に影響を与えるレジスターの過剰な使用

- 以下のような要因によって引き起こされた、Transparent レンダーキューのオーバードロー:
  - 非効率的な UI レンダリング
  - パーティクルシステムの重複または過剰な使用
  - ポストプロセスエフェクト
- 以下のような極端に高い画面解像度:
  - 4K ディスプレイ
  - モバイルデバイスにおける Retina ディスプレイ
- 以下のような要因によって引き起こされた微小三角形:
  - 高密度なメッシュジオメトリ
  - LOD (詳細レベル) システムの欠如。モバイル GPU で特に問題になりやすく、PC やコンソール GPU にも影響を与える可能性があります
- 以下のような要因によって引き起こされた、キャッシュミスや GPU メモリ帯域の無駄遣い:
  - 非圧縮テクスチャ
  - ミップマップを使用しない高解像度テクスチャ
- ジオメトリシェーダーまたはテッセレーションシェーダー。動的シャドウが有効になっている場合は 1 フレームにつき複数回実行される可能性があります

アプリケーションが GPU 依存であると思われる場合は、GPU に送信されるドローコールバッチを簡単に理解する方法として、フレームデバッガーを使用できます。ただし、このツールは具体的な GPU タイミング情報を提示できず、シーン全体の構築方法のみを提示します。

GPU のボトルネックの原因を調査する最善の方法は、適切な GPU プロファイラーからの GPU キャプチャを調べることです。使用するツールは、ターゲットハードウェアと選択したグラフィックス API によって異なります。詳細については、このガイドの プロファイリングおよびデバッグツール のセクションを参照してください。

モバイルの課題:熱制御とバッテリー寿命

熱制御は、モバイルデバイス向けのアプリケーションを開発する際に最適化すべき最も重要な領域の 1 つです。コードが非効率的であるために CPU や GPU がフルスロットルで長時間の処理を行うことになると、これらのチップは熱くなります。オペレーティングシステムは過熱やチップの損傷を避けるためにデバイスのクロックスピードを下げて冷却し、フレームスタッターやユーザー体験の劣化を引き起こします。このパフォーマンスの低下は、サーマルスロットリングと呼ばれます。

フレームレートが高くなり、コード実行 (または DRAM アクセス操作) が増えると、バッテリー消耗と発熱が増加します。また、パフォーマンスが悪いと、ローエンドのモバイルデバイスのセグメント全体でゲームがプレイできなくなり、市場機会を逃してしまう可能性があります。

熱の問題に取り組む場合は、取り扱うバジェットをシステム全体のバジェットとして考えてください。

早期にプロファイリングを行うことでゲームを最初から最適化し、サーマルスロットリングやバッテリー消耗の問題に対処しましょう。熱とバッテリー消耗の問題には、ターゲットプラットフォームハードウェアのプロジェクト設定を調整することで対応します。

モバイルにおけるフレームバジェットの調整

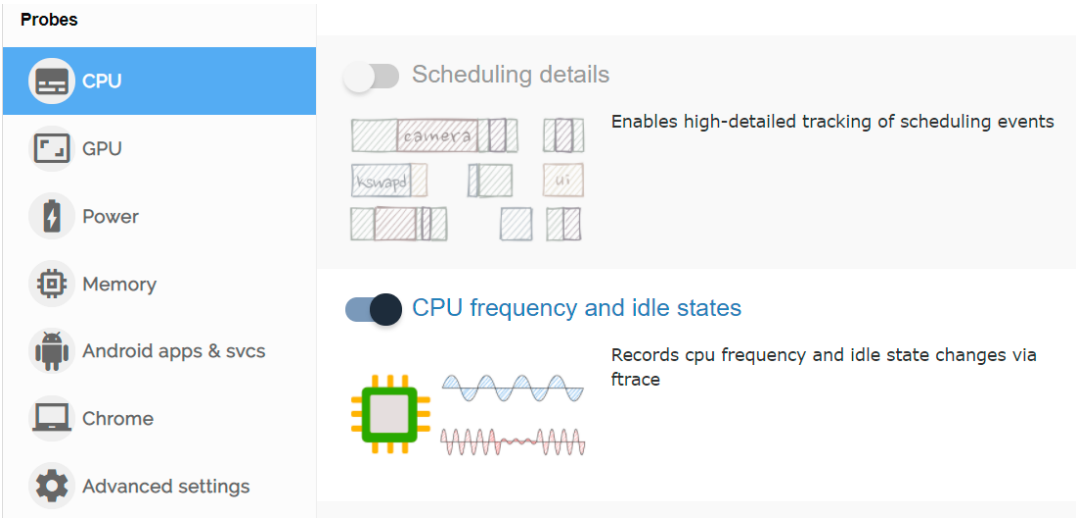
プレイ時間の延長に伴うデバイス温度の問題に対処するための一般的なヒントは、フレームのアイドル時間を約 35% にすることです。これにより、モバイルチップが冷却するための時間が得られ、バッテリーの過剰な消耗を防ぐことができます。目標フレーム時間を 1 フレームあたり 33.33 ms (30 fps の場合) とすると、モバイルデバイスのフレームバジェットは 1 フレームあたり約 22 ms になります。

計算は以下のようになります。**(1000 ms/30) * 0.65 = 21.66 ms**

同じ計算を使用してモバイルで 60 fps を達成するには、(1000 ms/60) * 0.65 = 10.83 ms の目標フレーム時間が必要です。これは多くのモバイルデバイスでは実現が難しく、30 fps を目標とするよりも 2 倍速くバッテリーを消費することになります。これらの理由から、多くのモバイルゲームは 60 fps ではなく 30 fps を目標としています。この設定を制御するには Application.targetFrameRate を使用します。フレーム時間の詳細については、フレームバジェットを設定する のセクションを参照してください。

モバイルチップの周波数スケーリングによって、プロファイリング時にフレームのアイドル時間バジェットの割り当てを特定するのが難しくなる可能性があります。改善や最適化は全体としてポジティブな効果をもたらしますが、モバイルデバイスのスケーリング周波数が下がっており、その結果としてより低温で動作している可能性があります。FTrace や Perfetto などのカスタムツールを使用して、モバイルチップの周波数、アイドル時間、最適化前後のスケーリングを監視しましょう。

目標 fps の総フレーム時間バジェット (30 fps の場合は 33.33 ms) 内に収まり、このフレームレートを維持するためにデバイスがより低い負荷で動作したり、より低い温度を記録したりしている場合は、正しい方向に進んでいます。



FTrace や Perfetto などのツールを使用して CPU 周波数とアイドル状態を監視し、フレームバジェット割り当ての最適化の結果を特定するために活用しましょう。

モバイルデバイスのフレームバジェットに余裕を持たせるべきもう 1 つの理由は、現実世界の温度変動を考慮する必要があるからです。暑い日には、モバイルデバイスが熱くなって放熱が難しくなり、サーマルスロットリングやゲームパフォーマンスの低下につながる可能性があります。このような状況を回避するには、フレームバジェットの一定割合を余分に取っておいてください。

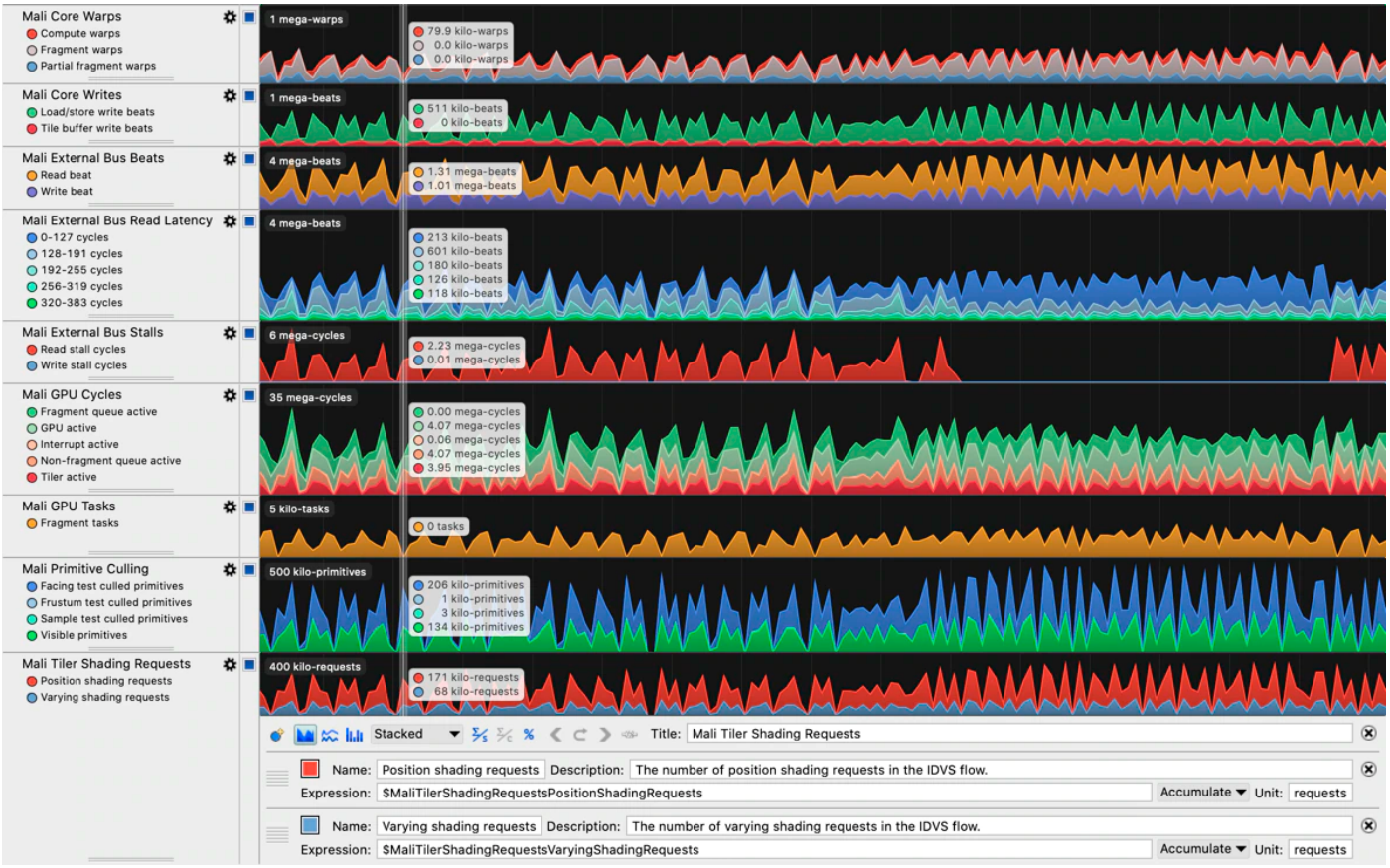
メモリアクセス操作の削減

DRAM アクセスは、モバイルデバイスでは通常、消費電力の高い操作です。Arm の モバイルデバイス向けグラフィックスコンテンツの最適化に関するアドバイス によると、LPDDR4 メモリアクセスのコストは 1 バイトあたり約 100 ピコジュールです。

フレームあたりのメモリアクセス操作の数を減らすには、以下のようにします。

- フレームレートを減らす
- 可能な場合、ディスプレイの解像度を下げる
- 頂点数と属性精度を抑えたシンプルなメッシュを使用する
- テクスチャ圧縮とミップマップを使用する

Arm の CPU または GPU ハードウェアを利用するデバイスに注力する必要がある場合、Arm Performance Studio ツール (具体的には Streamline Performance Analyzer) で、メモリ帯域幅の問題を特定するための優れたパフォーマンスカウンターを利用できます。各世代の Arm GPU で使用可能なカウンターについては、対応するユーザーガイド (Mali-G710 パフォーマンスカウンター参照ガイド など) にリストと説明があります。Arm Performance Studio の GPU プロファイリングには、Arm Immortalis または Mali GPU が必要です。



Arm の Streamline Performance Analyzer では、ターゲット Arm ハードウェアのライブプロファイリングセッション中にキャプチャできる豊富なパフォーマンスカウンター情報が利用できます。これは、オーバードローに起因するメモリ帯域幅の飽和など、パフォーマンスの問題を特定するのに最適です。

選択された一連の ARM ハードウェア指標は、Systemmetrics パッケージ を介して Unity プロファイラーおよびプレイヤーのビルドに公開されます。

ベンチマークのためのハードウェアティアの確立

プラットフォーム固有のプロファイリングツールを使用するだけでなく、サポートするプラットフォームと品質レベルごとにティアまたはデバイスの最低スペックを設定し、これらの仕様ごとにパフォーマンスをプロファイリングして最適化しましょう。

例えば、モバイルプラットフォームをターゲットにしている場合、ターゲットハードウェアに基づいて機能のオン/オフを切り替える品質管理を備えた 3 つのティアのサポートを決定するかもしれません。その後、各ティアで最もスペックの低いデバイスに合わせて最適化を行うことになります。もう 1 つの例として、コンソール向けのゲームを開発している場合は、古いバージョンと新しいバージョンの両方でプロファイリングを行うようにしてください。

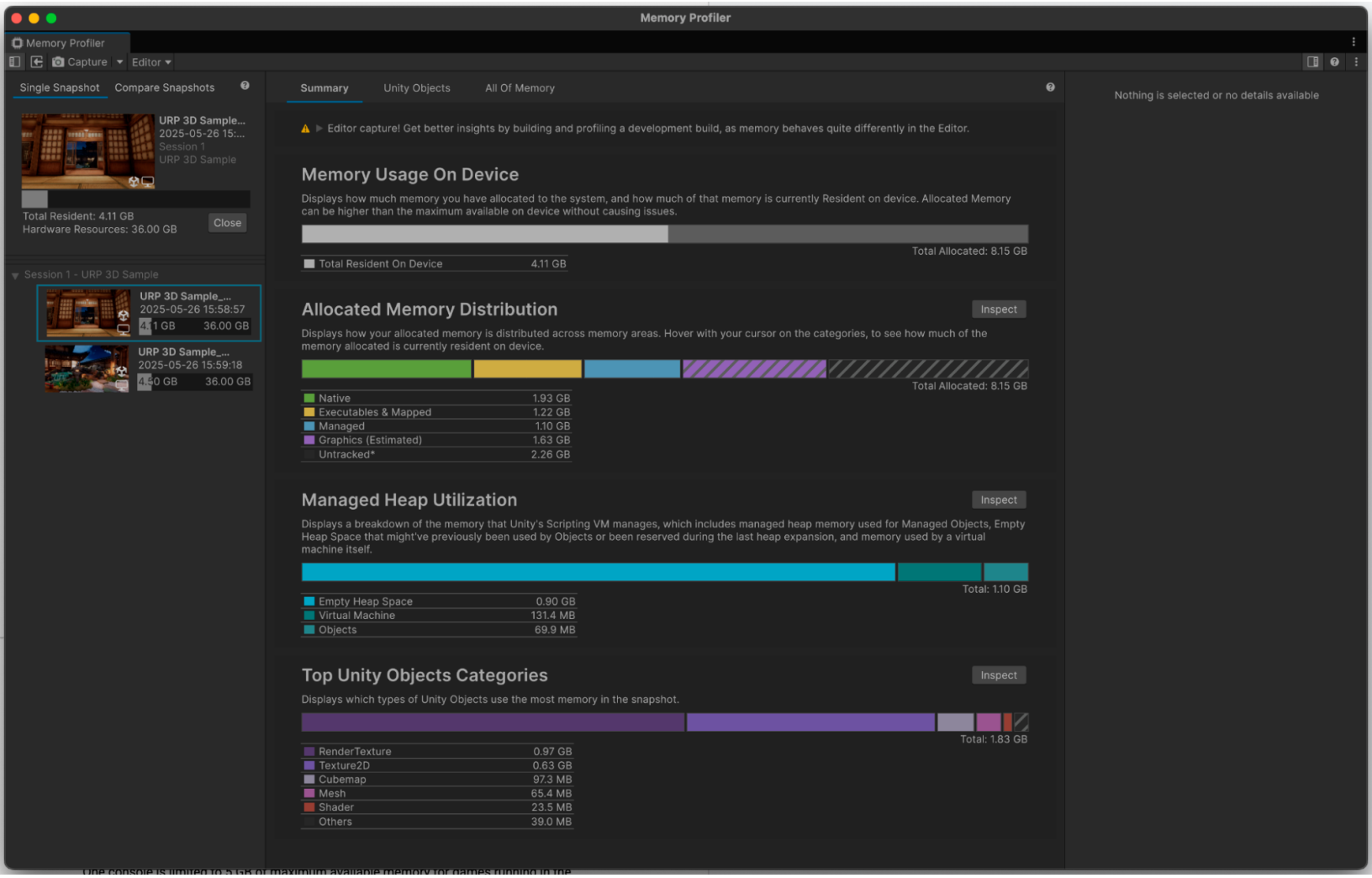
最新のモバイル最適化ガイド (このガイドと PC/コンソール最適化ガイドへのリンクは前のセクションにあります) には、ゲームを実行するモバイルデバイスのサーマルスロットリングを減らし、バッテリー寿命を延ばすためのヒントが多数、掲載されています。

メモリプロファイリング

メモリプロファイリングはランタイムのパフォーマンスとはほとんど無関係ですが、ハードウェアプラットフォームのメモリ制限に対してテストを行う場合や、ゲームがクラッシュしている場合に役立ちます。また、メモリ使用量を実際に増加させる変更を加えることで CPU/GPU のパフォーマンスを改善したい場合にも有用な場合があります。

Unity でアプリケーションのメモリ使用量を分析する方法は 2 つあります。

1. Memory Profiler モジュール:通常のプロファイラーで、メモリがアプリケーションによって何に使用されているかの基本的な情報を提供する、ビルトインのプロファイラーモジュールです。
2. Memory Profiler:プロジェクトに追加できる Unity パッケージとして利用できる専用ツールです。このパッケージにより、Unity エディターに Memory Profiler ウィンドウが追加されます。これを使用すれば、アプリケーションのメモリ使用量をより詳細に分析することが可能です。スナップショットを保存および比較してメモリリークを見つけたり、メモリレイアウトを確認してメモリ断片化の問題を発見したりできます。これについては、このガイドの後半で詳しく説明し、ここでは検討する必要がある一般的な考慮事項に焦点を当てます。



Memory Profiler パッケージは、Unity アプリケーションと Unity エディターのメモリ使用量を調べるために使用できるツールです。

これらのツールは両方とも、メモリ使用量の監視、アプリケーション内でメモリ使用量が想定より高い領域の特定、メモリ断片化の発見および改善を行うことができます。

メモリバジェットの把握および定義

マルチプラットフォーム開発では、ターゲットデバイスのメモリ制限について理解し、バジェットを設定することが重要です。シーンやレベルをデザインする際には、ターゲットデバイスごとに設定されたメモリバジェットを守る必要があります。制限やガイドラインを設定することで、各プラットフォームのハードウェア仕様の範囲内でアプリケーションが正常に動作することを保証できます。

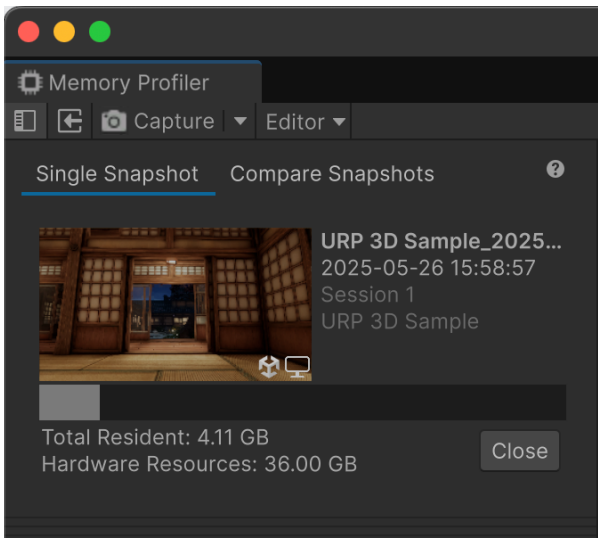
デバイスのメモリ仕様は 開発者ドキュメント に記載されています。

また、メッシュやシェーダーの複雑さに関するコンテンツバジェットの設定や、テクスチャ圧縮にも便利です。これらはすべて、割り当てられるメモリの量に影響します。これらのバジェット数値は、プロジェクトの開発サイクル中に参照できます。

物理敵な RAM の制限について判断する

各プラットフォームにはメモリ制限があるため、アプリケーションではターゲットデバイスごとにメモリバジェットが必要になります。Memory Profiler を使用して、メモリ使用量のキャプチャされたスナップショットを確認しましょう。**ハードウェアリソース** のスナップショット (下の画像を参照) には、**物理ランダムアクセスメモリ** (RAM) と **ビデオランダムアクセスメモリ** (VRAM) のサイズが示されています。この数字は、これらメモリの容量がすべて利用できるわけではないことは、考慮していません。しかし、これは作業を開始する際の参考となる概算値として有用です。

ここに表示されている数値は、必ずしも全体像を示しているとは限らないため、ターゲットプラットフォームのハードウェア仕様を併せて参照することをおすすめします。開発者キットのハードウェアにはより多くのメモリが搭載されている場合もあれば、統一されたメモリアーキテクチャを持つハードウェアを使用している場合もあります。



ハードウェアリソースのスナップショットには、スナップショットがキャプチャされたデバイスの RAM と VRAM の数値が表示されます。

各ターゲットプラットフォームに対応する最低スペックを決定する

サポートする各プラットフォームにおいて RAM のスペックが最も低いハードウェアを特定し、メモリバジェットの決定の参考にしてください。物理メモリがすべて使用できるわけではないことに注意しましょう。例えば、コンソールでハイパーバイザーを実行することで、総メモリの一部を使用する古いゲームをサポートすることができるでしょう。具体的なシナリオに応じて、チーム内で使用する割合 (全体の 80% など) を考えてください。また、モバイルプラットフォームでは、ハイエンドデバイス向けの高品質と機能をサポートできるように、複数の仕様ティアに分割することを検討しても良いでしょう。

大規模なチームの場合はチームごとのバジェットを検討する

メモリバジェットを定義したら、チームごとのメモリバジェットの設定を検討しましょう。例えば、環境アーティストにはロードされるレベルやシーンごとに使用できる一定量のメモリ、オーディオチームには音楽やサウンドエフェクト用のメモリなどを割り当てます。これは柔軟性に欠けるように思えるかもしれませんが、リソースのコストではなくクリエイティブな意思決定を導くための情報提供を重視したガイドラインと考えてください。

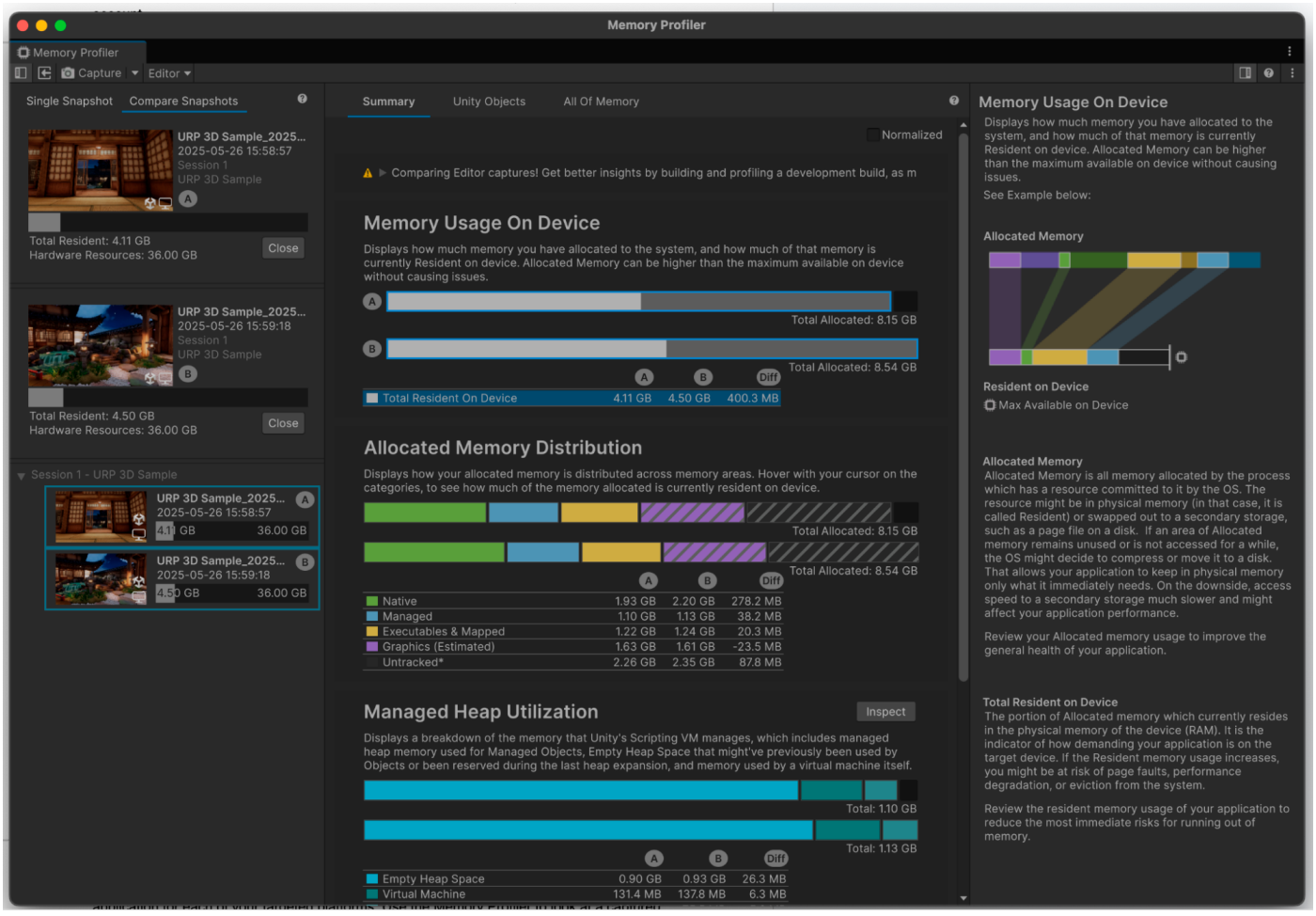
プロジェクトの進捗に合わせてバジェットを柔軟に設定することも重要です。あるチームの成果物がバジェット内に収まった場合、その余剰分が別のチームが開発しているゲームの領域を改善できるのなら、そのチームに割り当てるようにします。

ターゲットプラットフォームのメモリバジェットを決定して設定したら、次はプロファイリングツールを使用してゲーム内のメモリ使用量を監視および追跡し、情報に基づいた意思決定と必要に応じてのアクションを実行できるようにします。

Memory Profiler パッケージによる詳細な分析

Memory Profiler パッケージは、さらに詳細なメモリ分析に役立ちます。これを使用してスナップショットを保存および比較し、メモリリークを見つけたり、アプリケーションのメモリレイアウトを確認して最適化できる領域を見つけたりできます。

Memory Profiler パッケージの大きな利点の 1 つは、(Memory Profiler モジュールのように) ネイティブオブジェクトをキャプチャするだけでなく、マネージメモリ の表示、スナップショットの保存と比較、メモリ使用状況の詳細で視覚的な内訳によるメモリの内容の確認もできることです。



Memory Profiler パッケージを使用すると、スナップショットを比較できます。

Memory Profiler パッケージ の詳細については、「Unity のプロファイリングおよびデバッグツール」のセクションを参照してください。

メモリプロファイリング時に留意すべきヒント

メモリバジェットを設定する際には、ターゲットプラットフォーム全体においてスペックが最も低いデバイスでプロファイリングを行うことを忘れないでください。メモリ使用量を厳密に監視し、目標値に注意してください。

一般的に、プロファイリングを行う際には、大容量のメモリを搭載した高性能な開発システムを使用することをおすすめします (大容量のスナップショットを保存したり、それらのスナップショットを迅速にロードおよび保存するためのスペースが重要です)。

メモリプロファイリングは、CPU や GPU のプロファイリングとは異なり、メモリオーバーヘッドを増加させる可能性があるためです。ハイエンドデバイス (メモリ容量が大きいもの) ではメモリプロファイリングが必要になる場合がありますが、特にローエンドのターゲット仕様におけるメモリバジェット制限には注意が必要です。

品質レベル、グラフィックスのティア、アセットバンドルのバリアントなどの設定は、より高性能なデバイスではメモリ使用量が変化する場合があります。その点を踏まえ、メモリプロファイリングを最大限活用するためのポイントを紹介します。

— シャドウマップに使用されるレンダーテクスチャのサイズには、品質とグラフィックスの設定が影響することがあります。

— 解像度のスケーリングは、スクリーンバッファのサイズ、レンダーテクスチャ、ポストプロセスエフェクトに影響を与える可能性があります。

— テクスチャの設定は、すべてのテクスチャのサイズに影響を与える可能性があります。

— 最大 LOD はモデルなどに影響を与える可能性があります。

- HD (高解像度) 版と SD (標準解像度) バージョンなどのアセットバンドルのバリアントがある状況で、ターゲットデバイスの仕様に基づいて使用するアセットを選択すると、プロファイリングするデバイスによってアセットのサイズが異なる場合があります。
- ターゲットデバイスの画面解像度は、ポストプロセスエフェクトに使用されるレンダーテクスチャのサイズに影響します。
- デバイスでサポートされているグラフィックス API は、サポートするシェーダーのバリアントやサポートしているかどうかによって、シェーダーのサイズに影響する可能性があります。
- AssetBundle のバリアントだけでなく、品質やグラフィック設定も異なるティア型システムは、より幅広いデバイスをターゲットにする優れた方法です。
    - 例えば、4 GB のモバイルデバイスには HD バージョンのアセットバンドルをロードし、2 GB のデバイスには SD バージョンをロードできます。ただし、上記のメモリ使用量のばらつきを考慮して、両方のタイプのデバイスと、画面解像度やサポートされているグラフィックス API が異なるデバイスをテストしてください。

注意:Unity エディターは、エディターとプロファイラーからロードされる追加のオブジェクトのため、通常はより大きなメモリフットプリントを表示します。さらに、テクスチャメモリフットプリントは、すべてエディターで読み取り/書き込みが有効になるように強制されるため、大きくなります。

Unity のプロファイリングおよびデバッグツール

Unity は、パフォーマンスの問題を予防し、特定し、解決するための一連のツールを提供しています。このガイドでは、これらのうちのいくつかをすでにご紹介しました。それでは、次に、それぞれをいつ使用すべきか詳しく見ていきましょう。

このセクションで紹介するツールの中には、静的アナライザーやデバッグツールのカテゴリ (フレームデバッガーなど) に該当するものもあります。これらはプロファイラーではありませんが、Unity プロジェクトの分析と改善を行うにあたり、ツールキットに加えるべき重要なツールです。

プロファイリングツール、デバッグツール、静的分析ツールには、以下のような違いがあります。

プロファイリングツール は、コードの実行に関連するタイミングデータをインストルメント化および収集します。

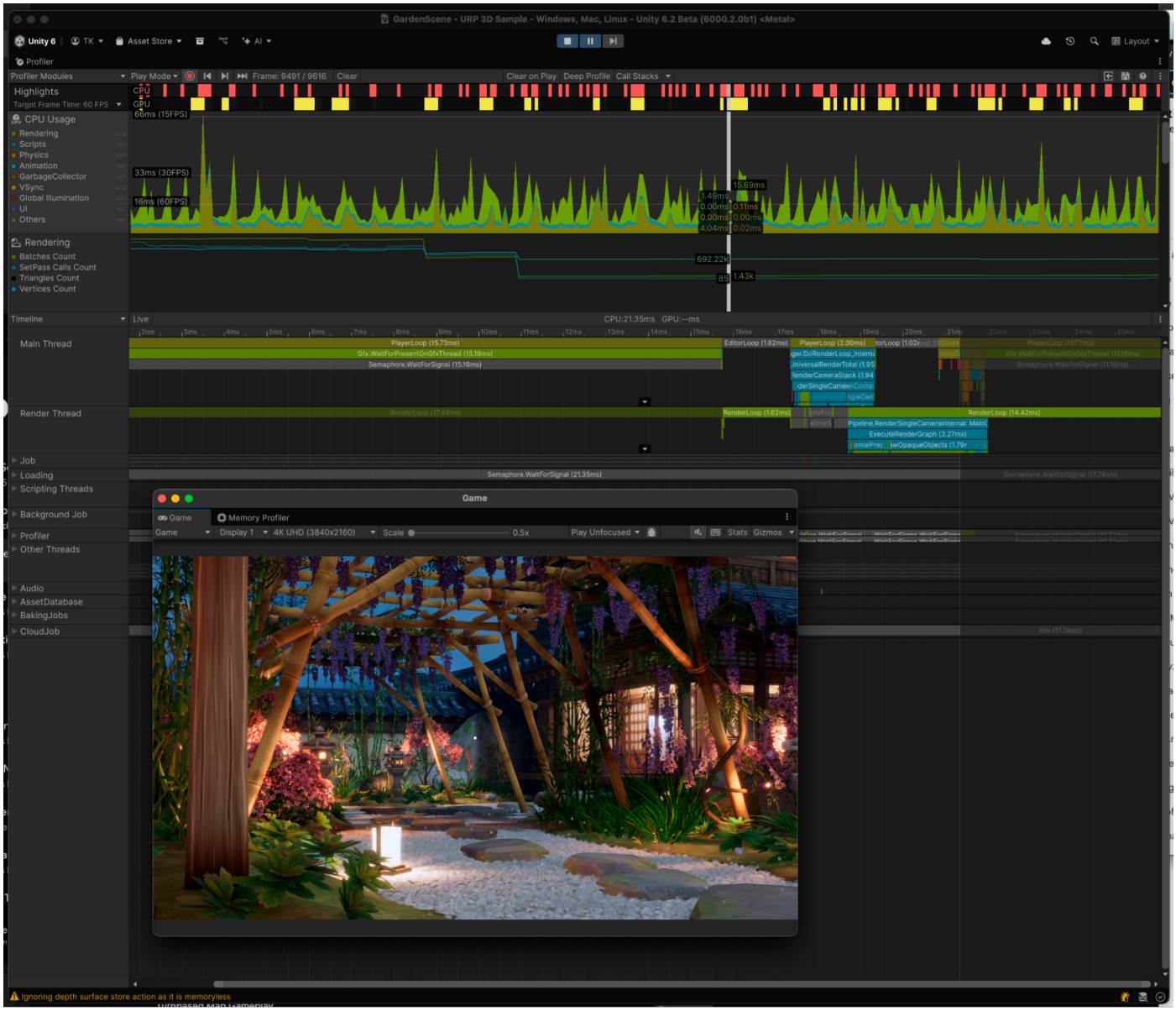
デバッグツール では、プログラムの実行をステップ実行したり、一時停止して値を調べたり、その他の多くの高度な機能を利用できます。例えば、フレームデバッガーを使用すると、フレームのレンダリングをステップ実行したり、シェーダーの値を調べたりできます。

静的アナライザー は、ソースコードやその他のアセットを入力として受け取り、ビルトインのルールを使用して分析し、プロジェクトを実行することなく、その入力の "正しさ" を推論できるプログラムです。

Unity プロファイラー

ビルトインの Unity プロファイラー は、ランタイム時のボトルネックやフリーズの原因を検出し、特定のフレームや時点で何が起こっているかをよりよく理解するのに役立ちます。このガイドで前述したように、プロファイラーでは、アプリケーションのビルドのプロファイリングを行うことも、エディターで直接プロ

ファイリングを行うこともできます。ただし、オーバーヘッドがいくらか増え、結果が歪む可能性があります。また、多くの場合は開発用マシンのほうがターゲットデバイスより性能が高いことも覚えておく必要があります。



URP 3D サンプル の庭のシーンをプロファイリングしている Unity プロファイラーの動作

プロファイラーに含まれている ディーププロファイリング 設定は、ランタイムに実行される特定のコードについて詳細に理解する必要がある場合に役立ちます。

さらに、Unity プロファイラーでは 異なるモジュール 間の比較が可能であるため、アプリケーションの特定の部分に集中できます。アプリケーションのパフォーマンスを包括的に把握するために、**CPU**、**メモリ**、および **Renderer** モジュールを常に有効にしておくことをおすすめします。調査している問題の種類に基づき、必要な場合は、オーディオや物理演算などの他のモジュールも有効にしてください。

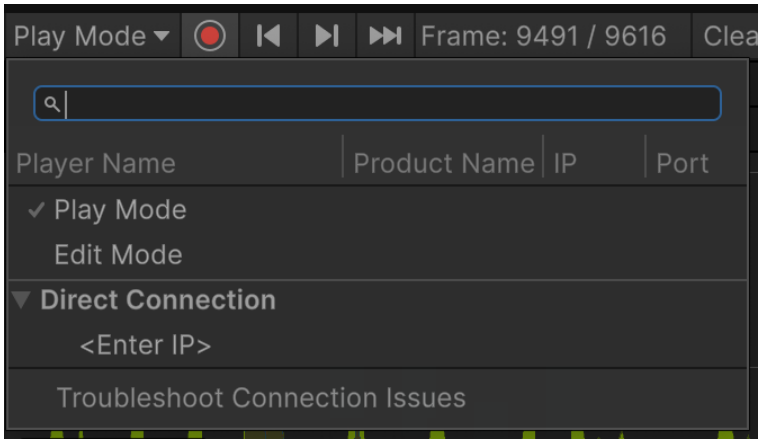
Unity でプロファイリングを始める

Unity プロファイラーを初めて使用する場合は、こちらの動画チュートリアルを参照してください (YouTube自動音声翻訳/字幕推奨)　または、以下の手順に従って開始することもできます。

- プロファイリング時には開発ビルドを使用する必要があります。そのためには、**File > Build Profiles** から、**Development Build** チェックボックスをオンにします。
- **Autoconnect Profiler** のチェックボックスをオンにします (任意)。
    - 注意:Autoconnect Profiler を使用すると、最初の起動時間が最大 10 秒増加するため、最初のシーンの初期化をプロファイリングしたい場合にのみ有効にしてください。Autoconnect Profiler を有効にしない場合、プロファイラーを実行中の開発ビルドに手動で接続できます。
- ターゲットプラットフォーム用にビルドします。
- **Window > Analysis > Profiler** から、プロファイラーを開きます。
- 不要なプロファイラーモジュールはすべて無効にします。有効化された各モジュールは、プレイヤーにパフォーマンスのオーバーヘッドを発生させます (このオーバーヘッドの一部は、Profiler.CollectGlobalStats マーカーを使用して確認できます)。
- **Preferences > Profiler** ウィンドウで **Frame Count** オプションを設定します。数値を高く設定すると、Profiler ウィンドウで分析できるフレーム数が増加しますが、エディターマシンで余分なメモリを使用することになります。
- デバイスのモバイルネットワークを無効にし、Wi-Fi を有効にしておきます。
- ターゲットデバイスでビルドを実行します。
    - Autoconnect Profiler を選択すると、ビルドにエディターマシンの IP アドレスがベイクされます。起動時に、アプリケーションは、この IP アドレスで Unity プロファイラーに直接接続を試みます。プロファイラーが自動的に接続され、フレームとプロファイリング情報が表示され始めます。
    - Autoconnect Profiler を選択しなかった場合は、**Target Selection** ドロップダウンを使用してプレイヤーに手動で接続する必要があります。

トップバーに新しく追加された Highlights モジュールにより、プロジェクトが目標フレームレートを達成できない箇所を特定しやすくなりました。

ビルド時間を短縮するには (精度が低下する代償として)、アプリケーションを Unity エディター内で直接実行してプロファイルしてください。Profiler ウィンドウの **Attach to Player** ドロップダウンメニューから再生モードを選択します。



プロファイラーを使用して再生モードで実行されているゲームを対象に分析している様子

Unity プロファイラーのヒント

CPU プロファイラーモジュールで VSync および Others マーカーを無効化

VSync マーカーは "デッドタイム" を表します。デッドタイムとは、CPU のメインスレッドが VSync を待っている間、アイドル状態であることを意味します。マーカーを非表示にすると、他のカテゴリの時間がどのようになっているのか、あるいは全体のフレーム時間がどのように形成されているのかさえ、理解しにくくなることがあります。これを踏まえて、VSync が先頭に来るようにリストを並べ替えることも可能です。こうすることで、VSync マーカーによって生じた "ノイズ" が軽減され、グラフの全体像がより明確になります。

Others マーカーはプロファイリングのオーバーヘッドを表し、プロジェクトの最終的なビルドには含まれないため無視して問題ありません。

ビルドで VSync を無効化

メインスレッド、レンダースレッド、GPU がどのように相互作用しているかを明確に把握するためのもう 1 つの選択肢は、VSync が完全に無効になっているビルドをプロファイリングすることです。これには、以下の手順に従います。

1. **Edit > Project Settings...** を開きます。
2. **Quality** を選択し、ターゲットデバイスで使用する品質レベルをクリックします。
3. **VSync Count** を **Don’t Sync** に設定します。
4. ゲームの開発ビルドを作成し、プロファイラーに接続します。

次の VBlank を待つ代わりに、ゲームは前のフレームが完了次第、すぐに次のフレームを開始します。VSync を無効にすると、一部のプラットフォームでティアリングなどの視覚的なアーティファクトが発生する可能性があります (その場合は、リリースビルドで再度 VSync を有効にしてください) が、人為的な待ち時間を削除すると、特にプロジェクトのボトルネックがある場所を調査する際は、プロファイラーのキャプチャが見やすくなります。

プロファイリングの実行に再生モードを使用すべき場合とエディターモードを使用すべき場合を把握

プロファイラーを使用する場合、プレイヤーターゲットとして再生モード、エディター、リモートまたは接続されたデバイスを選択できます。

再生モードを使用してゲーム/アプリケーションのプロファイリングを行い、エディターモードを使用してゲームを取り巻く Unity エディターの動作を確認しましょう。

エディターをプロファイリングのターゲットとして使用すると、プロファイリングの精度に大きく影響します。Profiler ウィンドウは、再帰的に自身をプロファイリングしています。ただし、エディターのパフォーマンスが低下した場合、そのプロファイリングを行うことは有用です。これにより、エディターの動作を遅くし、生産性を妨げているスクリプトや拡張機能を特定できます。

エディターのプロファイリングを行う場合の例を、以下に示します。

— 再生ボタンを押してから再生モードに入るまでに時間がかかる場合

— エディターの動作が遅くなったり反応しなくなったりした場合

— プロジェクトを開くのに時間がかかる場合

アセットデータベースをより効果的に活用するためのヒント のブログ記事では、**-profiler-enable** コマンドラインオプションを使用して、エディターの実行を開始した瞬間からプロファイリングを開始する方法について説明しています。

スタンドアロンプロファイラーの使用

スタンドアロンプロファイラー を使用すると、Unity エディターとは独立した専用プロセスでプロファイラーを起動し、再生モードまたはエディターでのプロファイリングを行うことができます。これにより、プロファイラーの UI やエディターが、測定されたタイミングに影響を与えることがなくなります。また、フィルタリングや操作に使用できる、よりクリーンなプロファイリングデータも手に入ります。

プロファイラーをスタンドアロンプロセスとして起動する

エディターでプロファイリングを行ってイテレーションを高速化

パフォーマンスの問題を修正するために素早くイテレーションを行いたい場合は、エディターでプロファイリングを行いましょう。例えば、ビルドでパフォーマンスの問題が見つかった場合は、エディターでプロファイリングを行い、ここでも同様にその問題が見つかることを確認します。問題が見つかった場合、再生モードでのプロファイリングを使用して、考えられる解決策に向けて変更のイテレーションを迅速に行います。問題が解決したら、ビルドを行い、その解決策がターゲットデバイスでも機能することを確認します。

このワークフローは、変更のビルドとデバイスへのデプロイに費やす時間が少なくて済むため、最適です。時間をかけずに、エディターで素早くイテレーションを行い、プロファイリングツールを使用して変更結果を検証できます。

Memory Profiler モジュールの使用

Memory Profiler モジュールの機能の多くは Memory Profiler パッケージに置き換えられていますが、今でもこのモジュールを使用してメモリ分析作業に役立てることが可能です。

Memory Profiler モジュールの **Detailed** ビューを使用して、メモリ使用量が最も大きいメモリツリーを掘り下げ、多くのメモリを使用している原因を特定しましょう。

Memory Profiler モジュールを使用すると、システムに割り当てられたメモリ量を簡単に確認できます。

以下に、Unity プロファイラーのその他のユースケースや機能を探るのに役立つリソースをいくつか紹介します。

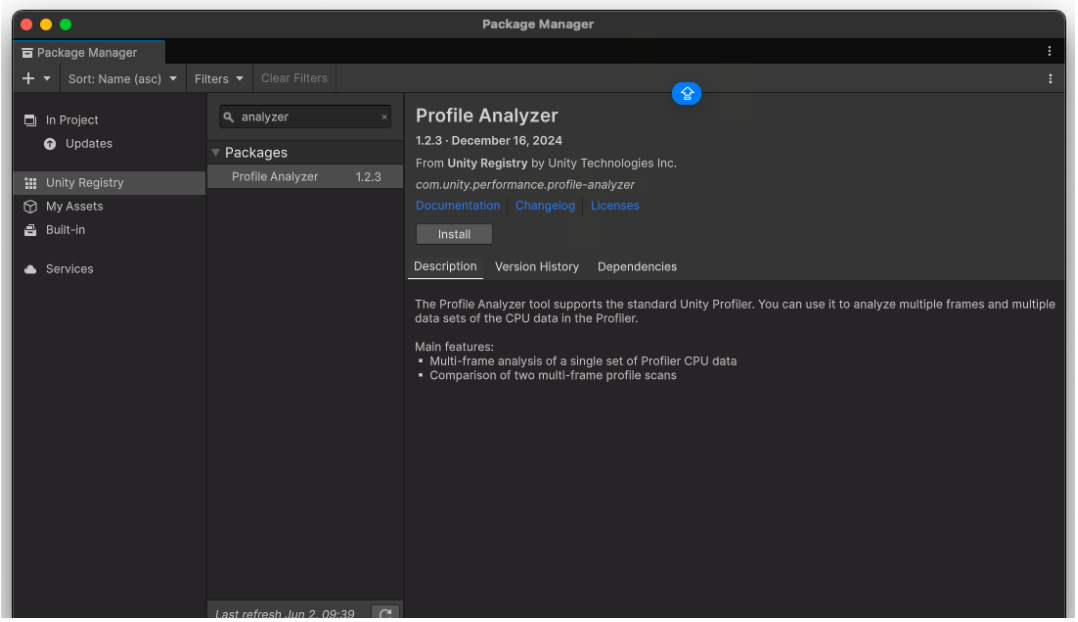
- Unity マニュアルのプロファイラーの概要
- Unity でのプロファイリング入門
- ゲームのプロファイリングと最適化の方法
- Unity プロファイラーのウォークスルーとチュートリアル

Profile Analyzer

標準の Unity プロファイラーでは個々のフレームの詳細な分析が可能ですが、Profile Analyzer では Unity プロファイラーの複数のフレームからキャプチャされたマーカーデータを集計して可視化し、より広範な "全体像" を提供します。これにより、複数のフレーム間または異なるプロファイリングセッション間でパフォーマンスデータを簡単に比較および分析できます。

Profile Analyzer の使用を開始するには、以下の手順を行います。

1. **Wind > Package Management > Package Manager** から、Profile Analyzer パッケージをインストールします。
2. Unity Registry に移動し、検索フィルターを参照または使用して Profile Analyzer パッケージを探します。

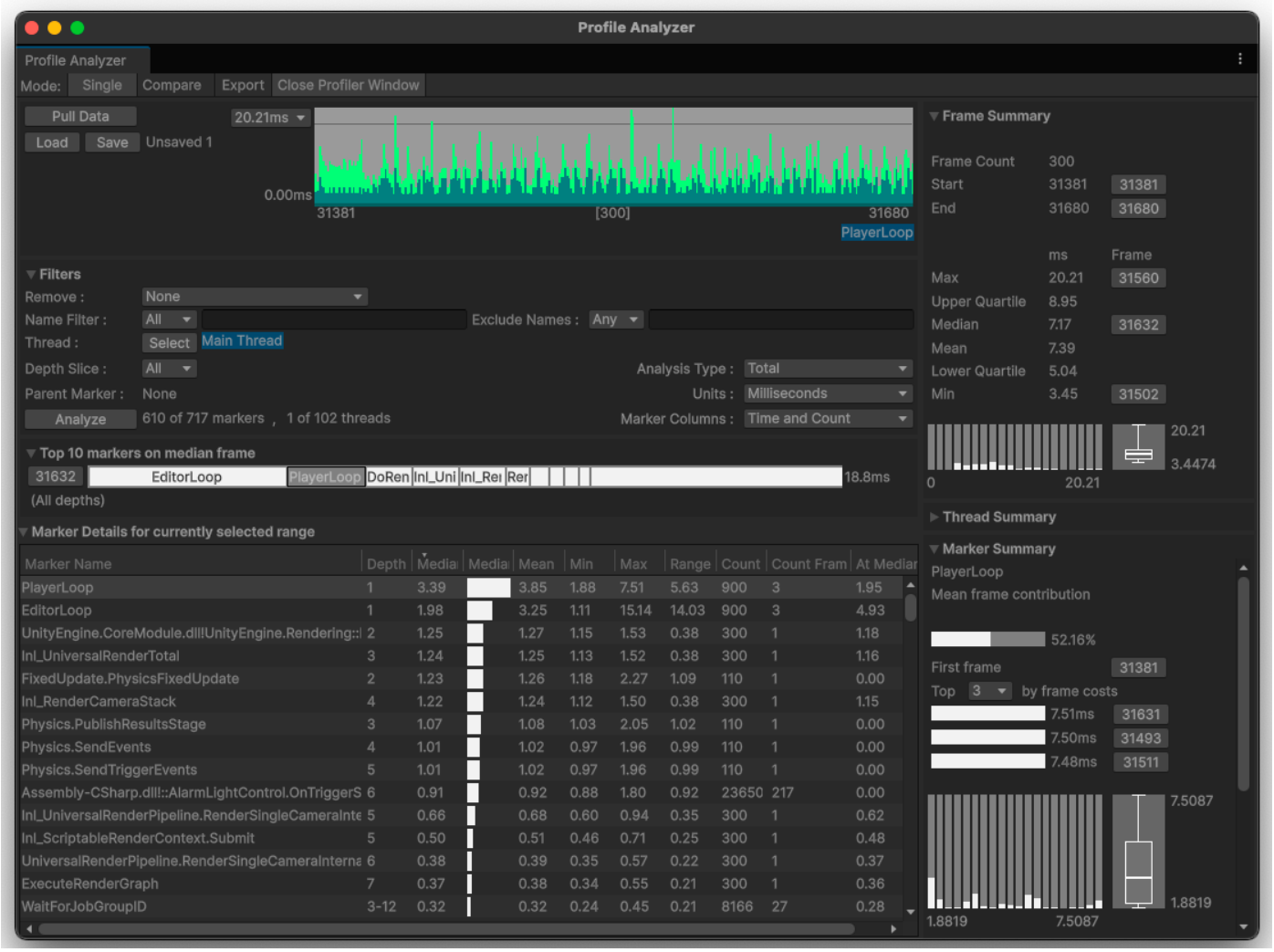


Package Manager から Profile Analyzer をインストールします。

Profile Analyzer は、Unity プロファイラーでキャプチャされた一連のフレームを取得し、それらに対して統計分析を実行します。表示されるデータは、タイミングの最小値、最大値、平均値、中央値など、各関数のパフォーマンスタイミングに関する有用な情報を提供します。

Profile Analyzer はデータセットの比較を行うのに適しているため、ゲーム開発の全工程で活用することで、パフォーマンスや最適化に関する課題の明確化に役立てることができます。また、このツールを使用して、ゲームシナリオのパフォーマンス差を検証するための A/B テストを実施したり、コードのリファクタリングや最適化、新機能の追加、または Unity のバージョンアップグレードなどについて、その前後でのプロファイリングデータを比較することも可能です。

Profile Analyzer を使用する際に便利なヒントの 1 つは、パフォーマンス最適化作業の前後を比較するためにプロファイリングセッションを保存することです。



Unity プロファイラーの優れた補完ツールである Profile Analyzer は、プロファイリングセッションでキャプチャされた複数のフレームを集計および比較します。上図は Single ビューのスクリーンショットです。

この集計と比較を始めるには、まずプロファイラーを使用してデータをキャプチャし、Profile Analyzer にそのデータを 入力 して分析を実行する必要があります。

集計データを使用することで、一度に 1 つのフレームだけを見るのではなく、ゲーム内で何が起きているかをより詳しく把握できます。例えば、300 フレーム（10 秒）のゲームプレイのキャプチャや 20 秒のロードシーケンスでは、次のような情報を確認する必要があるかもしれません。

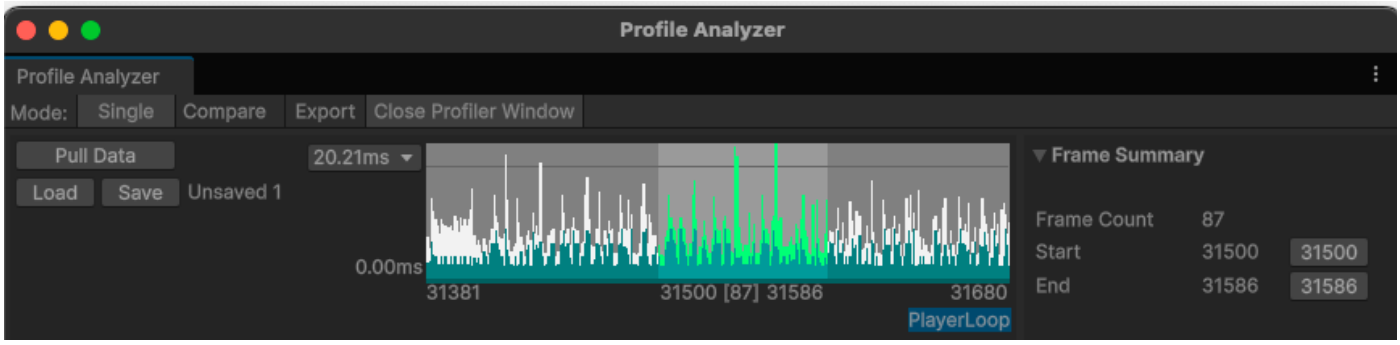
— メインスレッドとレンダースレッドで最も CPU コストが大きい要素は何か

— 各マーカーの平均/中央値/合計コスト

これらの重要な質問に答えることで、影響の大きい問題を特定し、その最適化に優先順位をつけることができます。

Profile Analyzer で利用できる統計や詳細により、コードのパフォーマンス特性について、複数のフレームにわたって実行している場合でもより深く掘り下げることができ、さらには以前のプロファイルキャプチャセッションとも比較可能です。

Frame Control パネル を使用して、1 つのフレーム、またはフレームの範囲を選択します。選択すると、Marker Details のペインが更新され、有用な統計情報を含むマーカーのソート可能なリストとともに、選択したフレームの集計データが表示されます。



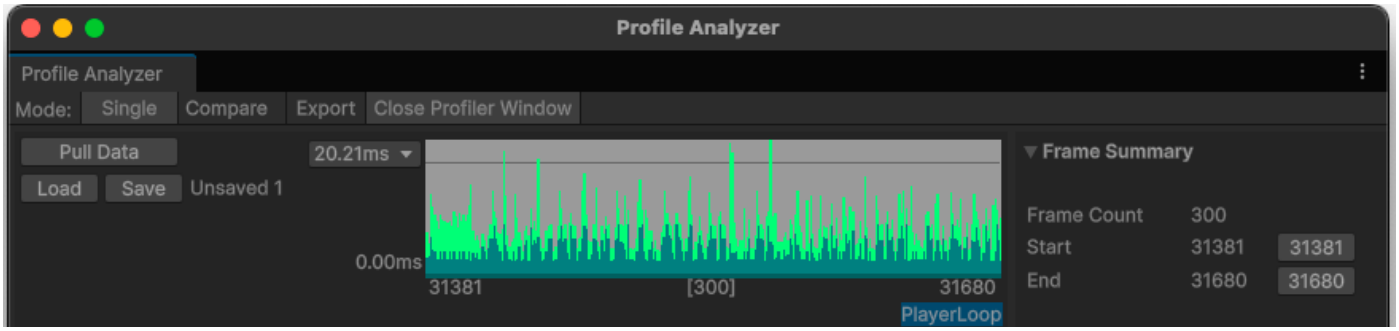
フレームコントロールパネルを使用して、詳細を見るフレームの範囲を選択します。

Marker Summary ペイン には、選択したマーカーの詳細情報が表示されます。リスト内の各マーカーは、選択したフレームの範囲内にあるフィルタリングされたすべてのスレッドについて、そのマーカーのすべてのインスタンスを集計したものです。

Marker Summary パネルには、Marker Details パネルで選択した各マーカー集計に関する詳細情報が表示されます。

Profile Analyzer ビュー

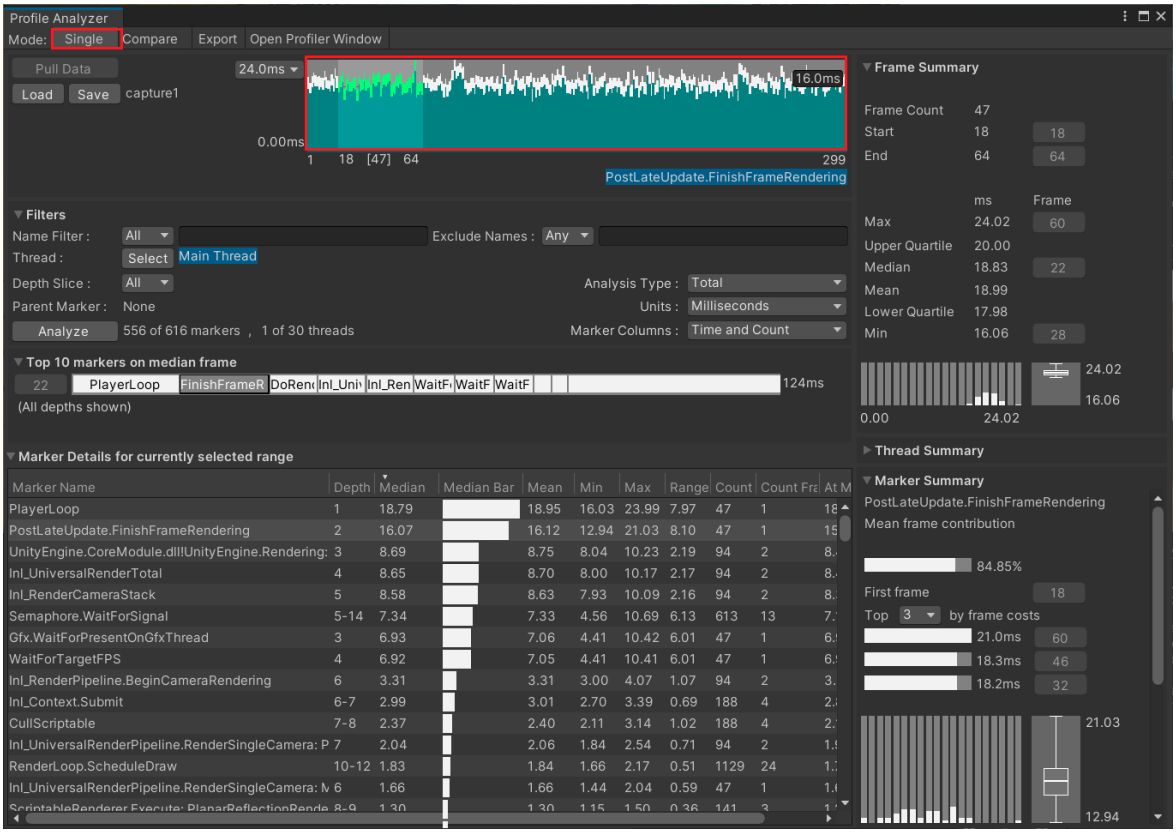
ウィンドウ上部での **Mode (モード)** の選択に注目してください。Profile Analyzer には、プロファイリングデータを分析するための複数のビューやアプローチがあります。さまざまなビューを使用して、プロファイリングデータのセットを選択、ソート、表示、比較します。



パネル上部でさまざまなモードを選択できます。

Single ビュー

Single ビュー は Profile Analyzer のデフォルトの開始点であり、時間経過に伴うパフォーマンスの全体像に関する分析結果を最初に提供します。Single ビューには、キャプチャされたプロファイルデータの単一のセットに関する情報が表示されます。これを使用して、プロファイルマーカーがフレーム間でどのように動作するかを分析しましょう。このビューはいくつかのパネルに分かれており、タイミングに関する情報のほか、フレーム、スレッド、マーカーの最小値、最大値、中央値、平均値、上下四分位数の値が含まれています。

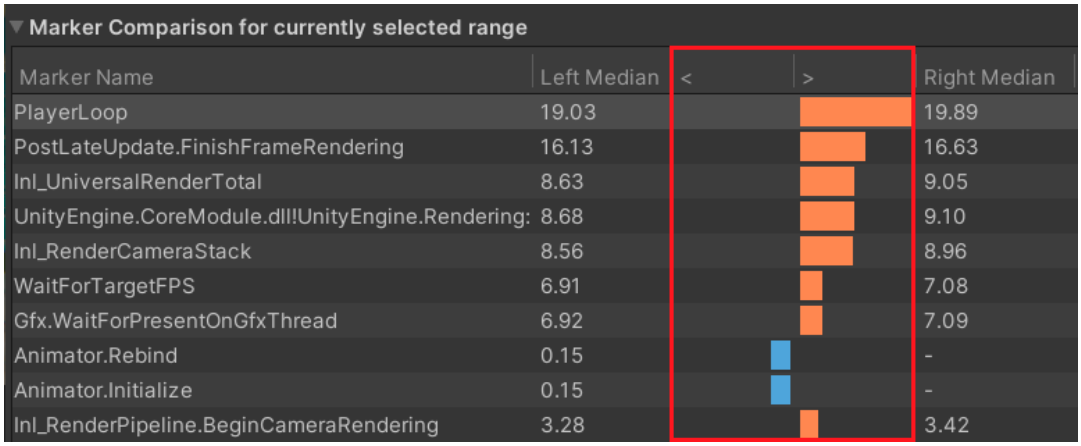


Single ビューには、単一または特定の範囲のフレームに対するプロファイルマーカーの統計情報とタイミングが表示されます。

Single ビューは、多くの有用な分析情報を提供できるため、プロファイリングツールキットの不可欠な要素となっています。

Compare ビュー

Compare ビュー は、2 つの異なるデータセットをロードし、異なる色で表示して、並べて比較できるためわかりやすく、パフォーマンスのばらつきを分析する場合に特に有用です。



データセットのマーカーのタイミングは、Compare ビューの Marker Comparison ペインとその色分けを使用して簡単に比較できます。

Profile Analyzer を使用してパフォーマンスの変化を比較するには、以下のステップを実行します。アクティブな Unity プロファイラーのキャプチャから **Pull Data** オプションを使用するか、保存されたセッションから **Load Data** オプションを使用できます。ロード時には、ファイルは Profile Analyzer の .pdata 形式である必要があります。Unity プロファイラーの .data ファイルの場合、まず Profiler ウィンドウで開き、Profile Analyzer で Pull Data を使用します。また、プロファイラーからオリジナルの .data ファイルを保存することをおすすめします。

1. **テストの準備:**ゲームの一貫したセクションを選択してプロファイリングを実行し、有意義なベンチマーク比較を行いましょう。スクリプト化された、または繰り返し可能な手動のプレイスルーが最も効果的です。これにより、パフォーマンスに影響を与えるランダムな副作用を最小限に抑えることができます。
2. **"適用前" データのキャプチャ:**
   - Profile Analyzer を開きます (**Window > Analysis > Profile Analyzer**)。
   - Unity プロファイラーで、選択したゲームプレイの最適化を行う前のプロファイリングセッションを記録します。
   - Analyzer の **Compare** タブで、最初の **Pull Data** ボタンをクリックします。これにより、プロファイラーから現在のキャプチャがロードされます。または、セッションを保存することもできます。
3. **最適化と "適用後" データのキャプチャ**:
   - コードやパフォーマンスの改善を適用します。
   - Unity プロファイラーの過去のデータをクリアし、同じゲームプレイの新しいプロファイリングセッションを記録します。
   - Profile Analyzer で 2 つ目の Pull Data ボタンをクリックして、この新しいセッションをロードします。

4. **違いの分析**:

   — **Marker Comparison** ペインには、"適用前" (左) と "適用後" (右) のキャプチャでマーカーのタイミングがどのように異なるかが示されます。

   — **<** または **>** のマークが付いた列は、その指標で値が大きいキャプチャがどちらかを示します。

   — 比較する指標は、**Marker Columns** フィルターを使用して変更できます。

Marker Comparison のそれぞれの列の詳細については、Compare ビューのエントリーページ を参照してください。

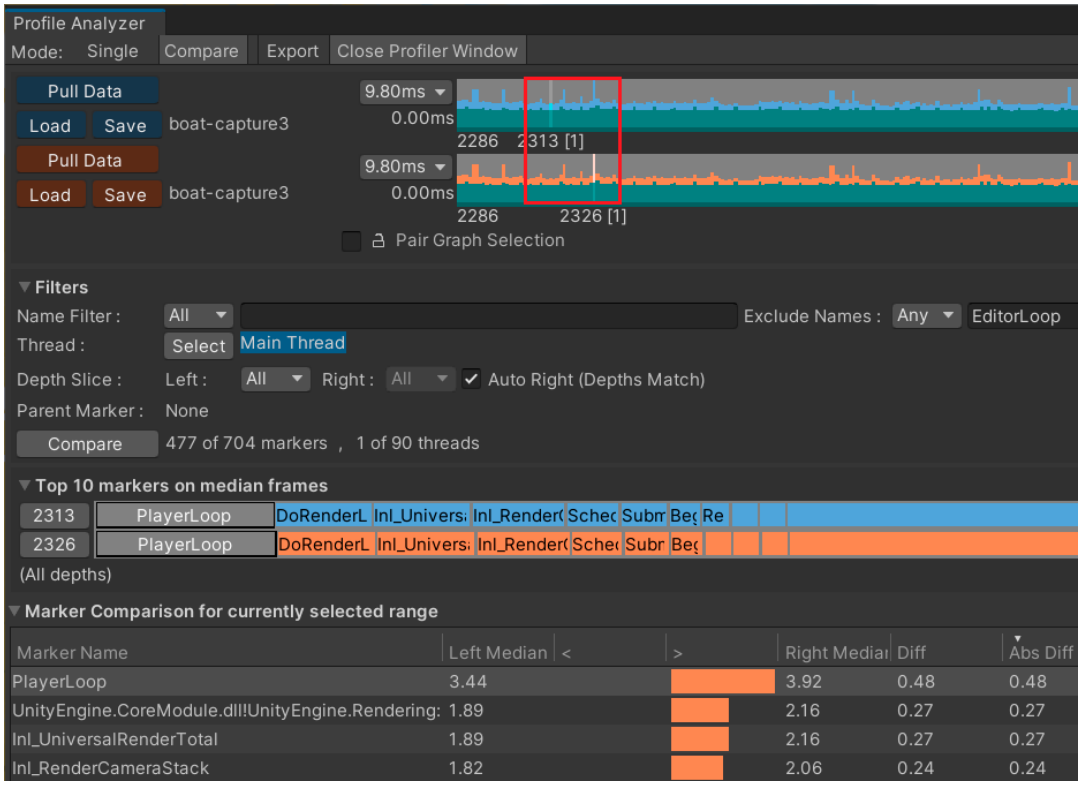
フレームの中央値と最長フレームの比較

プロファイラーの 1 つのキャプチャ内のフレームの中央値と最長フレームを比較して、後者では発生していて前者では発生していないものを特定したり、完了までに平均以上の時間がかかっているものは何かを確認したりできます。

Profile Analyzer の Compare ビューを開き、左側と右側の両方で同じデータセットをロードします。Single ビューでデータセットをロードしてから Compare に切り替えることもできます。

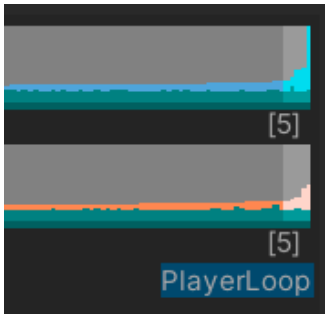
一番上の **Frame Control** グラフを右クリックし、**Select Median Frame** を選択します。一番下のグラフを右クリックし、**Select Longest Frame** を選択します。

Profile Analyzer の Marker Comparison パネルが更新され、差分が表示されます。



キャプチャから得たフレームの中央値と最長フレームの比較

データ比較に役立つもう 1 つのテクニックは、両方のグラフをフレームの継続時間順にソート (**右クリック > Order By Frame Duration**) し、外れ値のフレーム (不自然に長かったり短かったりするフレーム) に焦点を当てるかこれらを除外するかして、各セットの範囲を選択することです。



フレームを継続時間順に並べ、外れ値の範囲を選択

これにより、最も典型的なフレームと最も極端なフレームを比較できます。選択した範囲の Marker Comparison 表にデータが表示されるため、パフォーマンスのスパイクや不一致の原因を分析しやすくなります。

Profiler Analyzer の詳細については、以下のリソースを参照してください。

YouTube自動音声翻訳/字幕推奨

- Profile Analyzer ウォークスルーとチュートリアル
- Unity の Profile Analyzer による CPU パフォーマンス分析
- プロファイリング入門

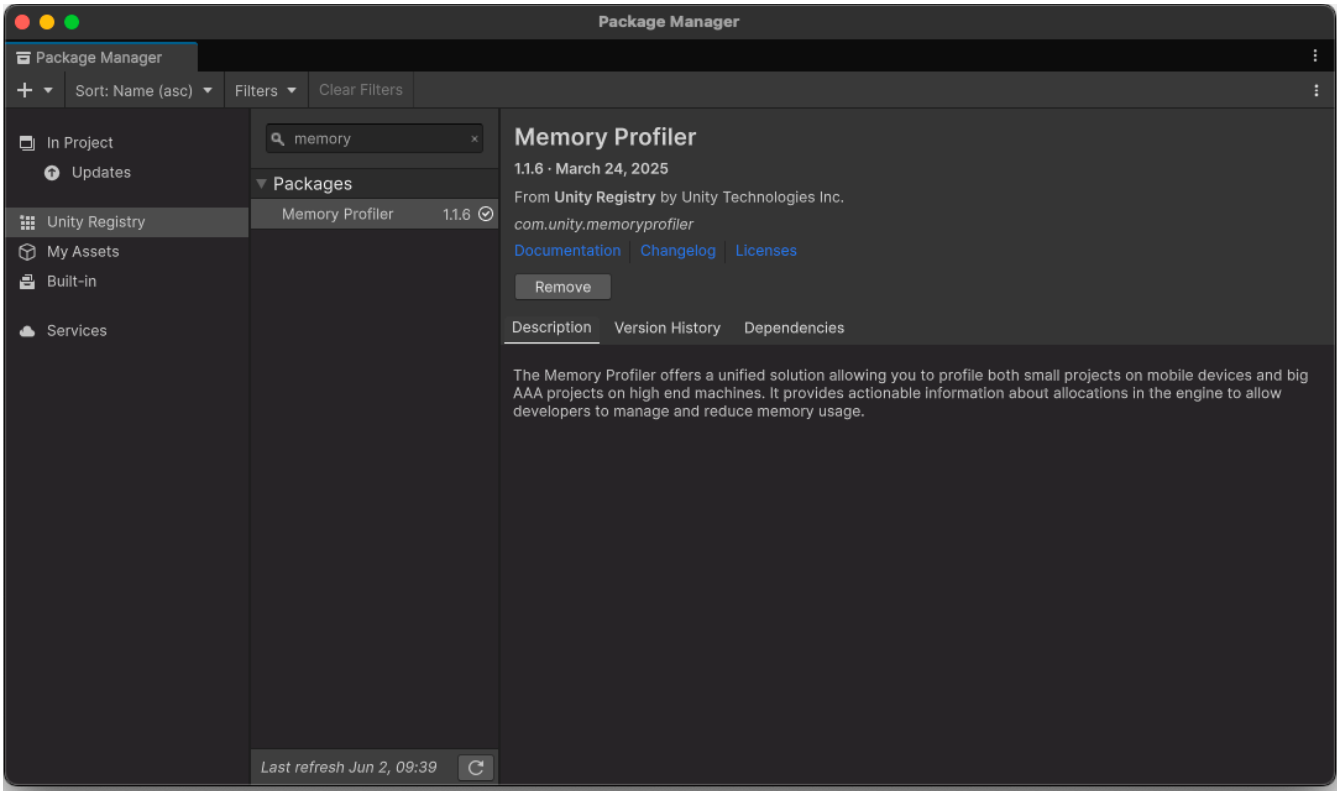
Profile Analyzer のヒント

- **Depth レベル** を **4** にして、ユーザースクリプトを深堀りしましょう (Unity エンジンの API レベルは無視してください)。このレベルにフィルタリングし、**Timeline モード** で Unity プロファイラーを確認すると、コールスタック深度を関連付けて選択できます。MonoBehaviour スクリプトは 4 つ下のレベルに青で表示されます。これは、他の "ノイズ" なしで、特定のロジックやゲームプレイスクリプトが単独で負荷をかけているかどうかを素早く確認する方法です。
- アニメーターやエンジンの物理演算など、Unity エンジンの他の領域でも同じようにデータをフィルタリングしましょう。
- **Frame Summary** セクションの右側には、強調表示されているメソッドのパフォーマンス範囲のヒストグラムが表示されます。**Max Frame** 番号 (時間がもっとも長かった特定のフレーム) にカーソルを合わせると、クリック可能なリンクが表示され、Unity プロファイラーでフレーム選択を確認できます。このビューを使用して、最大フレーム時間の高さに寄与する可能性のある他の要因を分析しましょう。
- ワイドスクリーンがある、またはモニターが 2 台ある場合は、Profile Analyzer と Unity プロファイラーを並べて開くと便利です。そうした場合、Profile Analyzer でフレームをダブルクリックすると、Unity プロファイラーで同じフレームが自動的に選択され、Timeline ビューまたは Hierarchy ビューを使用してさらに詳しく調べることができます。

Memory Profiler

Memory Profiler パッケージは、プロジェクトのメモリ使用量を把握および最適化するのに役立ちます。これにより、Unity エディター内、およびターゲットデバイス上で実行中のプレイヤービルドの両方で、特定の瞬間にアプリケーションのメモリの "スナップショット" をキャプチャできます。

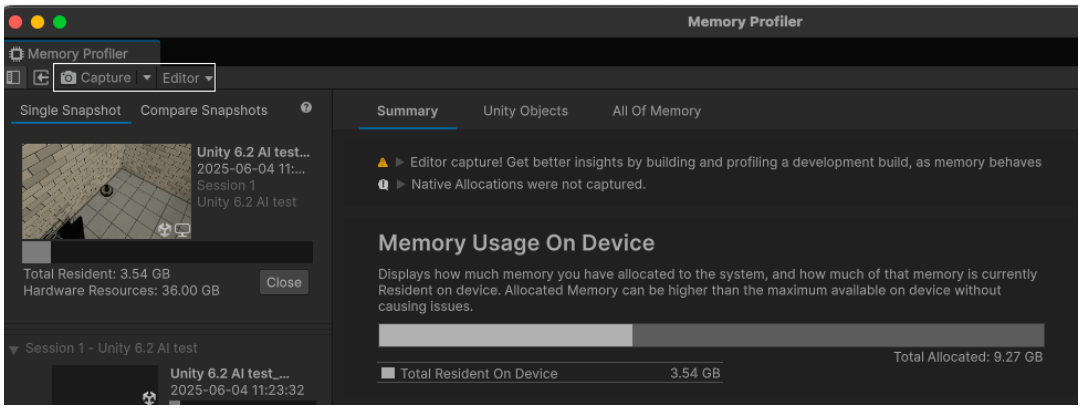
これらのスナップショットは、メモリの使用状況の包括的な内訳を提供し、エンジン全体における割り当てを示します。これにより、過剰または不必要なメモリ使用量の原因を特定し、メモリリークを追跡し、ヒープの断片化などの問題を調査できます。



Memory Profiler は Package Manager で利用可能なパッケージです。

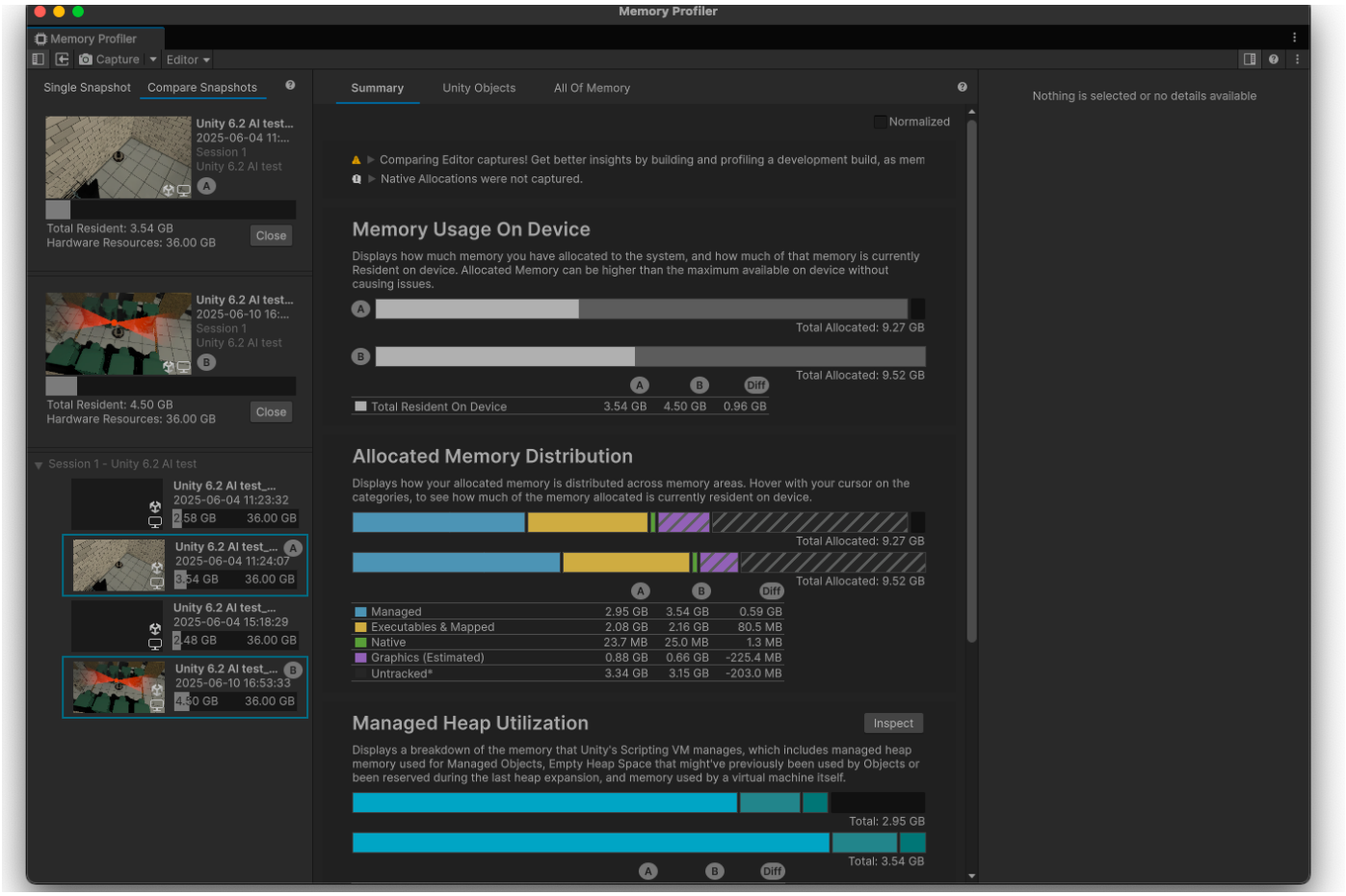
Memory Profiler パッケージをインストールしたら、**Window > Analysis > Memory Profiler** で開きます。

Memory Profiler 上部のメニューバーでは、プレイヤー選択ターゲットを変更したり、スナップショットをキャプチャまたはインポートしたりできます。左上隅のターゲット選択ドロップダウン (下の画像の場合、選択されたターゲットは "Editor") では、Memory Profiler をリモートデバイスに接続することで、ターゲットハードウェア上で直接メモリをプロファイリングできます。Unity エディターでプロファイリングを行うと、エディターやその他のツールによって増加したオーバーヘッドにより、不正確な数値が表示されることに注意してください。



プレイヤーの選択を変更し、メモリスナップショットをキャプチャまたはインポートします。

Memory Profiler ウィンドウの左側には Snapshots コンポーネント があります。これを使用して、保存されたメモリスナップショットを管理し、開いたり閉じたりできます。Snapshot コンポーネントには、Single Snapshot と Compare Snapshot の 2 つのビューがあります。



複数のメモリスナップショットを管理できます。

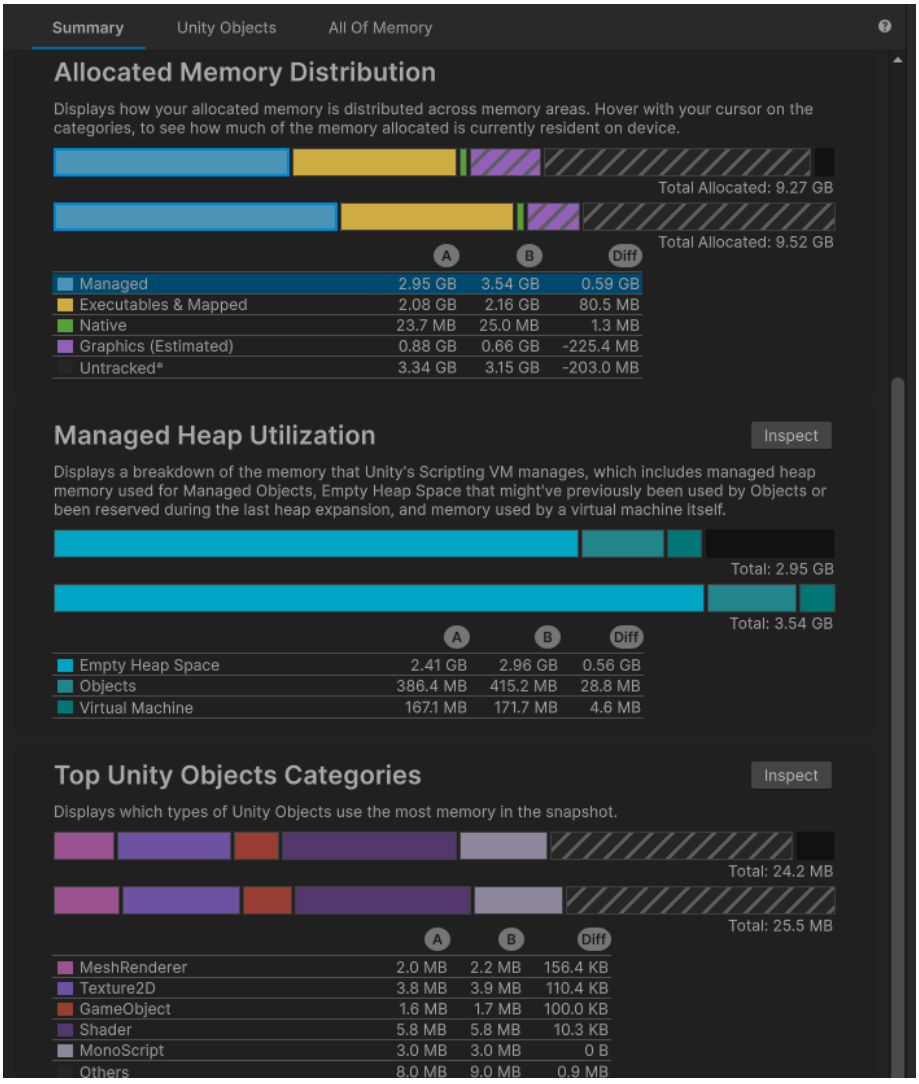
Profile Analyzer と同様に、Memory Profiler では 2 つのメモリスナップショットを並べてロードし、比較することが可能です。この比較は、時間の経過に伴うメモリ増加を追跡したり、シーン間の使用状況を分析したり、潜在的なメモリリークを特定したりするために使用します。

Memory Profiler のメインウィンドウには、メモリスナップショットの詳細な調査を可能にするタブがいくつかあります。主なタブは、**Summary**、**Unity Objects**、および **All of Memory** です。それぞれのオプションを詳しく見ていきましょう。

Summary タブ

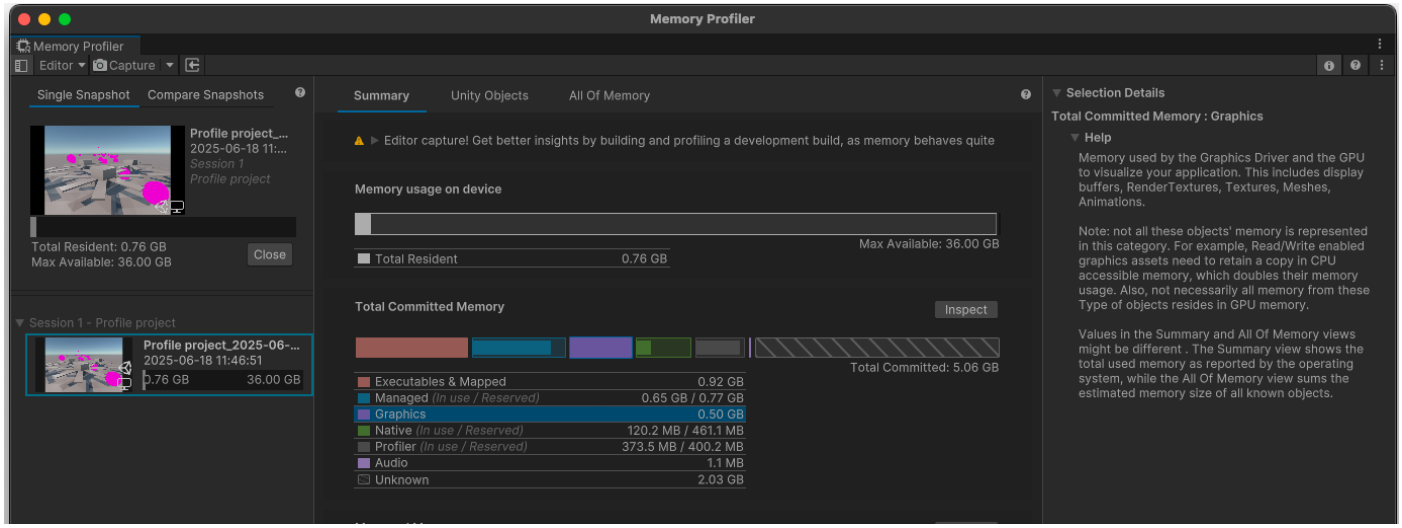
Summary タブ には、メモリキャプチャが取られた時点でのプロジェクトのメモリ使用状況の概要が表示されます。詳細な分析をすることなく、全体像を一目で把握しつつ十分な情報を得る必要がある場合に最適です。

このビューは、重要な指標をハイライトし、メモリの潜在的な問題や予期しない使用パターンを素早く見つけるのに役立ちます。これは特に、スナップショットを比較したり、メモリ使用量を経時的にデバッグしたりする場合に便利です。その主なセクションをいくつか見てみましょう。



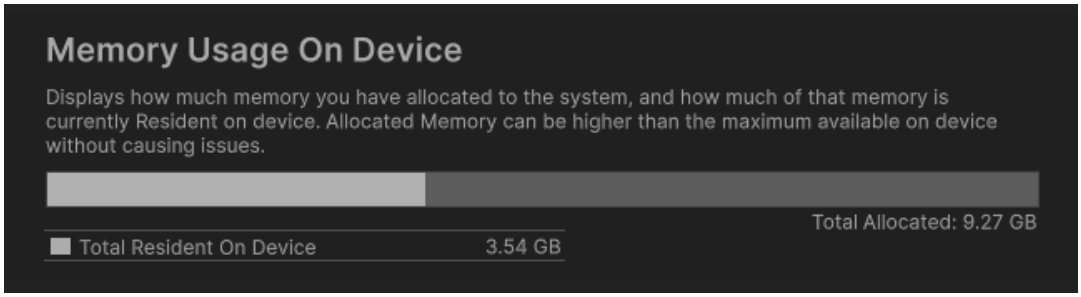
Summary タブには、スナップショットがキャプチャされた時点のメモリの概要が表示されます。

ヒント:右側のパネル（下の画像を参照）には、スナップショットに関する便利なコンテキスト情報が表示されます。これらは、潜在的な問題に対する警告や、結果の解釈のガイドとなります。

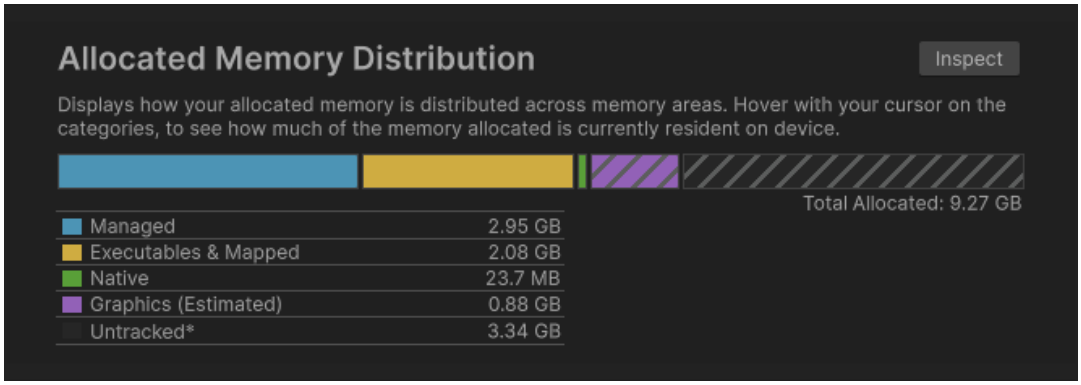


右側のパネルからは、スナップショットに関する便利なヒントが得られます。

Memory Usage on Device:物理メモリ内のアプリケーションのフットプリントを示します。これには、キャプチャ時にメモリ内に存在するすべての Unity の割り当て、および Unity 以外の割り当てが含まれます。

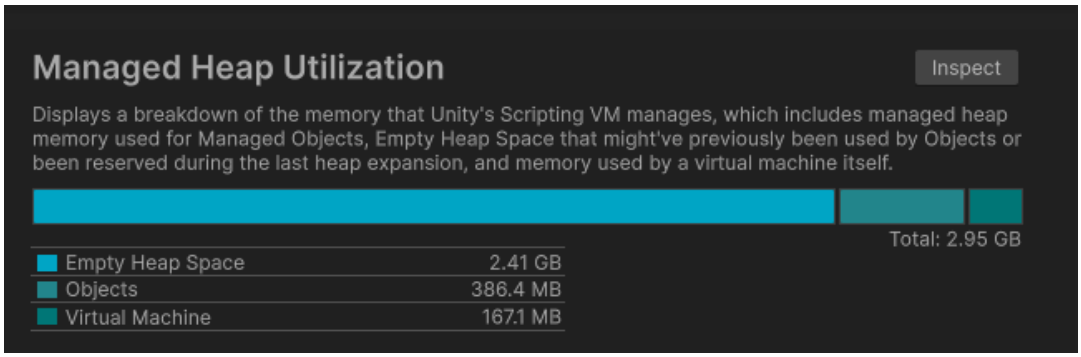


Allocated Memory Distribution:このビューは、割り当てられたメモリがさまざまなメモリカテゴリにどのように分散しているかを視覚化します。

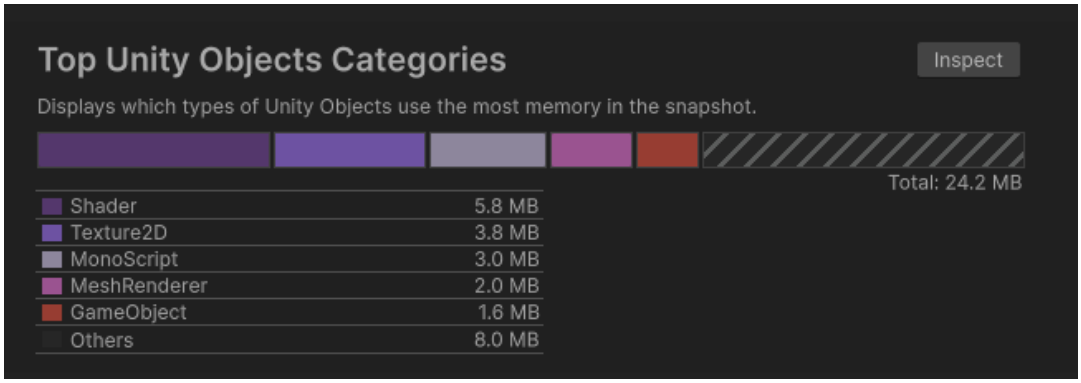


Untracked* メモリバーに注目してください。ここには、Unity がメモリ管理システムで追跡しないメモリが該当します。このような割り当ては、ネイティブのプラグインやドライバーに起因する場合があります。プラットフォーム固有のプロファイラーを使用して、ターゲットデバイスの Untracked メモリ使用量を分析しましょう。

Managed Heap Utilization:このビューでは、Unity のスクリプティング VM が管理するメモリの内訳を確認できます。これには、マネージオブジェクトに使用されるマネージヒープメモリ、オブジェクトによって以前に使用されたか最後のヒープ拡張時に予約された可能性のある空のヒープ領域、仮想マシン自体で使用されるメモリが含まれます。

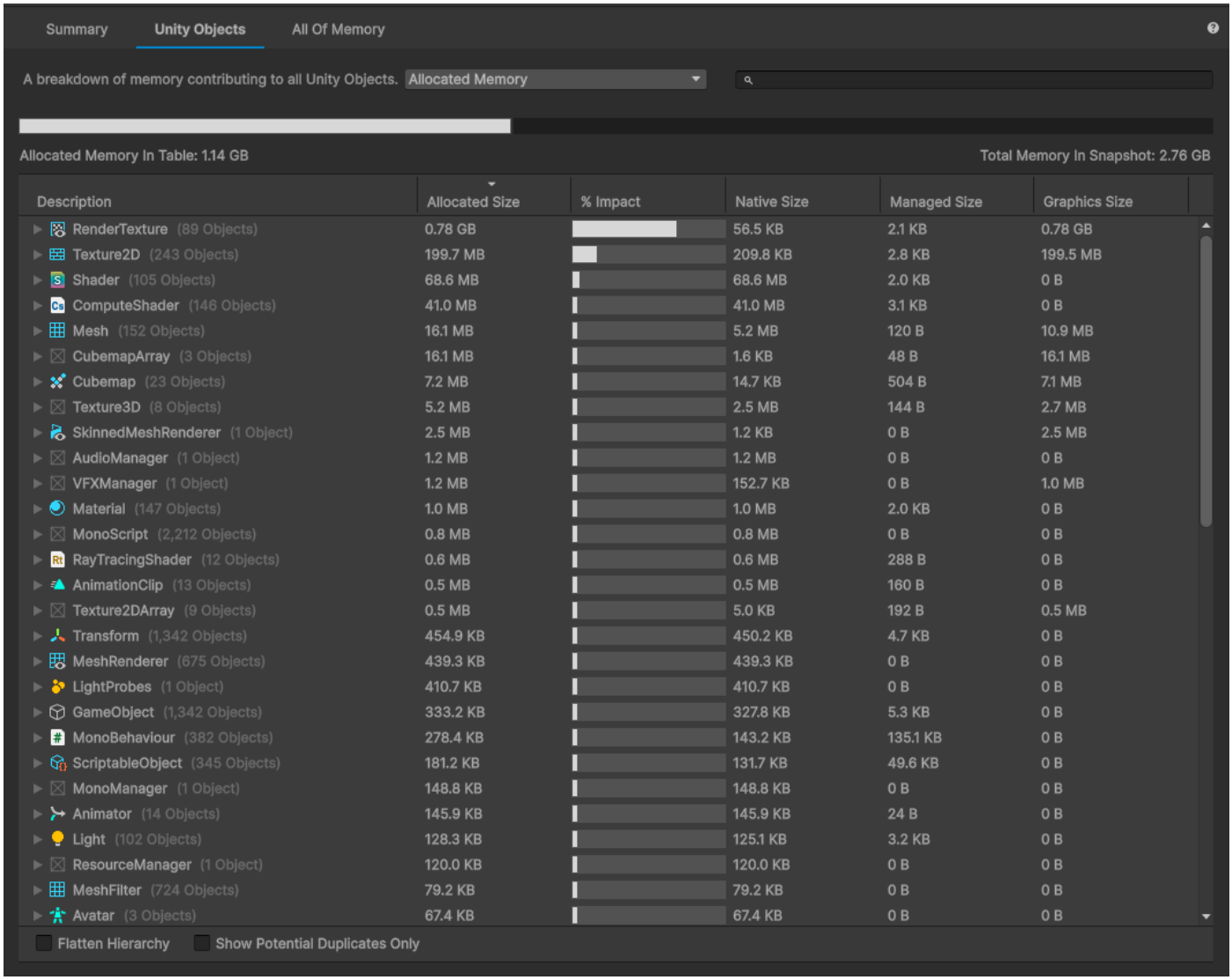


Top Unity Object Categories:スナップショットで最もメモリを使用している Unity オブジェクトのタイプが表示されます (Texture2D、メッシュ、ゲームオブジェクトなど)。



Unity Objects タブ

Unity Objects タブ には、メモリが割り当てられた Unity オブジェクト、オブジェクトが使用するネイティブメモリとマネージメモリの量、およびそれらの合計が表示されます。この情報を使用して、重複するメモリエントリを削除できる領域を特定したり、最もメモリを使用しているオブジェクトを特定したりできます。また、検索バーから、入力したテキストを含むテーブル内のエントリを見つけることもできます。



Unity Objects タブでは、キャプチャされたスナップショットのメモリ使用状況を高い粒度で詳細に分析できます。

デフォルトでは、この表には、関連するすべてのオブジェクトが割り当てサイズ順に一覧表示されます。列ヘッダー名をクリックすると、その列で表をソートしたり、列を昇順または降順でソートしたりできます。

メモリ使用量を最適化したり、メモリバジェットが限られているハードウェアプラットフォーム向けにメモリをより効率的に詰め込むことを目指す場合に、この方法を活用できます。

メモリプロファイリング手法とワークフロー

まず、Memory Profiler のスナップショットを分析して、メモリ使用率が高い領域を特定します。Memory Profiler のスナップショットをキャプチャまたはロードしたら、Unity Objects タブを使用して、メモリフットプリントのサイズが大きいものから小さいものの順にカテゴリを確認します。

プロジェクトアセットは、しばしばメモリの最大の消費源となります。Table モード を使用して、テクスチャ、メッシュ、オーディオクリップ、レンダーテクスチャ、シェーダーバリアント、および事前に割り当てられたバッファを見つけましょう。これらは、メモリ使用量を最適化する際に、最初に着手するのに適した対象となることが多いです。Project Auditor は、アセットのメモリ使用量を削減する方法に関する推奨事項を提供できるため、ここで非常に有用な補助ツールとなります (Inspector の**インポート設定**でアセットが正しく設定されていることを確認するのが良い出発点です)。

メモリリークの特定

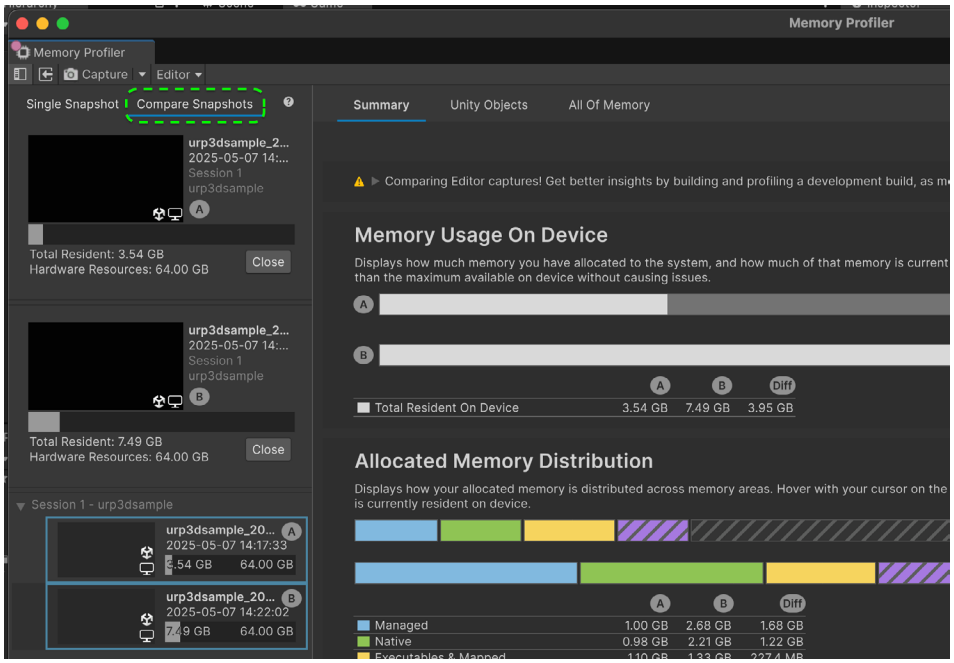
メモリリークとは、未使用のアセット、オブジェクト、またはリソースが適切にメモリからリリースされない状態のことです。これにより、メモリ使用量が徐々に増加し、パフォーマンスの問題やクラッシュが発生する可能性があります。

メモリリークは、一般的に次のような場合に発生します。

— オブジェクトがコードを通じてメモリから手動でリリースされない場合。

— 別のオブジェクトがそのオブジェクトへの参照を保持しているため、気付かないままオブジェクトがメモリに残っている場合。

Memory Profiler には、特定の期間における 2 つのスナップショットを比較して メモリリークを見つける のに役立つ Compare Snapshots (スナップショット比較) モードがあります。この比較により、割り当てが解除されるべきオブジェクトがメモリに留まり続けている状態を可視化できます。



2 つのスナップショットを比較して違いを確認します。

Unity ゲームにおいてメモリリークが頻出するのは、シーンをアンロードした後です。アンロードされたシーンのオブジェクトへの参照がまだ存在する場合、オブジェクトのガベージコレクションが正しく行われない場合があります。

アプリケーションの生存期間にわたって繰り返し発生するメモリ割り当ての特定

複数のメモリスナップショットの差分比較 を通じて、アプリケーションの生存期間中に継続的にメモリが割り当てられる原因を特定できます。

ここでは、プロジェクトでマネージヒープの割り当てを特定するためのヒントをいくつか紹介します。

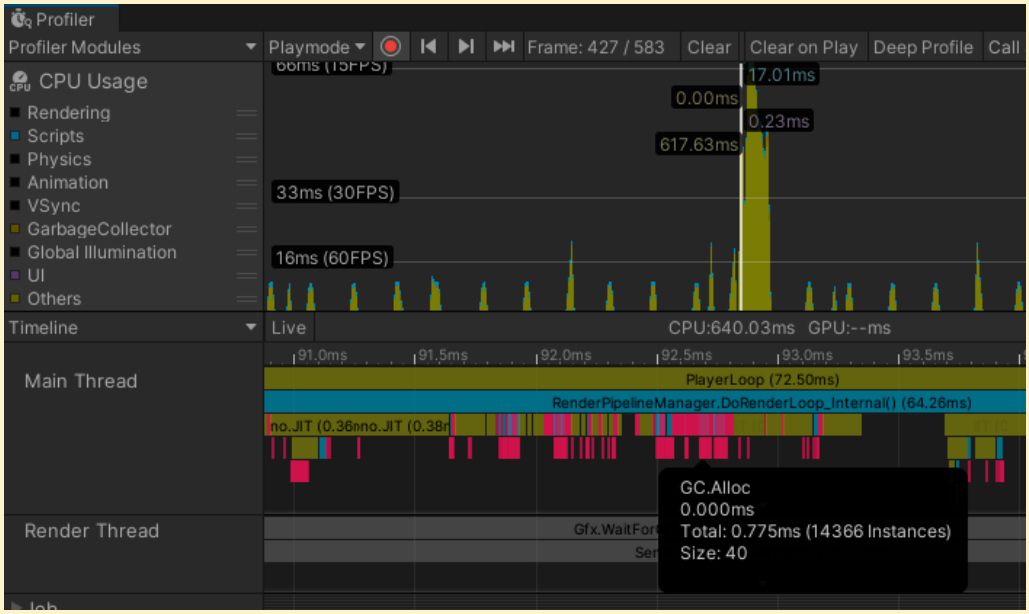
Unity プロファイラーの Memory Profiler モジュール

Unity プロファイラーの Memory Profiler モジュールは、フレームごとのマネージ割り当てを赤い線で表します。ほとんどの場合でこの値は 0 となるため、この線でのスパイクはマネージ割り当てを調査する必要があるフレームを示しています。

GC Allocated In Frame にスパイクがあれば、そのスパイクがマネージ割り当ての調査のヒントになります。

CPU Usage Profiler の Timeline ビュー

CPU Usage Profiler の Timeline ビューには、マネージ割り当てを含む割り当てがピンク色で表示されて見やすく、絞り込みやすくなっています。



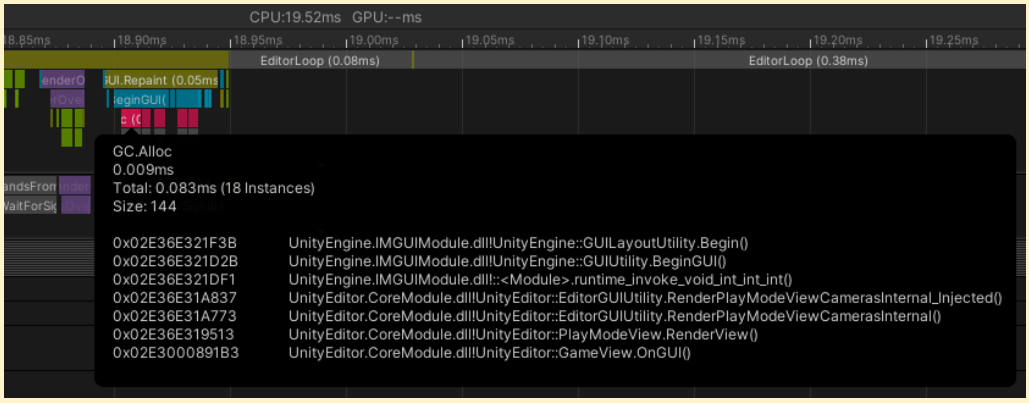
マネージ割り当ては、Timeline ビューにピンク色のマーカーとして表示されます。

アロケーションコールスタック

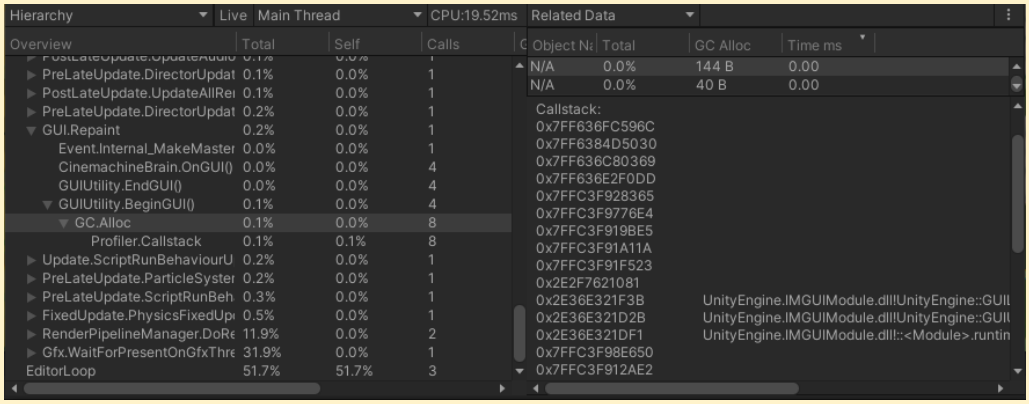
アロケーション (割り当て) コールスタック を使用すると、コード内のマネージメモリ割り当てを簡単に検出できます。ディーププロファイリングによって一般的に発生する量よりも少ないオーバーヘッドで必要なコールスタックの詳細を確認でき、標準のプロファイラーを使用してその場で有効化することも可能です。

アロケーションコールスタックは、プロファイラーのデフォルトでは無効になっています。有効にするには、Profiler ウィンドウのメインツールバーにある **Call Stacks** (コールスタック) ボタンをクリックし、**Details** ビューを **Related Data** に変更します。

注意:古いバージョンの Unity (アロケーションコールスタックがサポートされる前のもの) を使用している場合は、ディーププロファイリング でコールスタック全体を取得し、マネージ割り当てを確認するのがおすすめです。



プロファイラーでアロケーションコールスタックを有効にして、マネージ割り当てのソースまでコールスタックを追跡します。

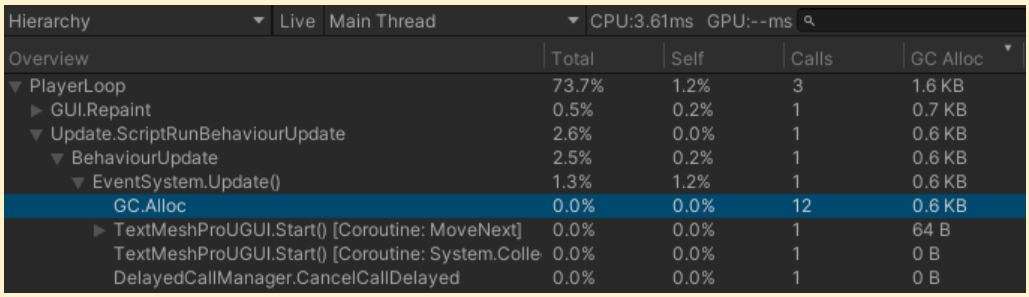


Hierarchy ビューの Related Data パネルには、アロケーションコールスタックの詳細も表示されます。

Hierarchy または Raw Hierarchy で選択した GC.Alloc サンプルに、そのコールスタックが含まれるようになります。また、Timeline の選択ツールチップで GC.Alloc サンプルのコールスタックを確認できます。

CPU Usage Profiler の Hierarchy ビュー

CPU Usage Profiler の Hierarchy ビューでは、列ヘッダーをクリックしてソート基準として使用できます。GC Alloc でソートするとマネージ割り当てに焦点を当てるのに便利です。



CPU Usage Profiler モジュールの Hierarchy ビューを使用して、マネージ割り当てをフィルタリングし、焦点を当てましょう。

メモリと GC の最適化

ガベージコレクション (GC) の影響を減らす

Unity では、Boehm-Demers-Weiser ガベージコレクター を使用して、アプリケーションで不要になったメモリが自動的にクリーンアップされます。GC はプログラムコードの実行を停止し、作業が完了した後にのみ通常の実行を再開します。

自動管理は便利ですが、ガベージコレクターは未使用のメモリをクリーンアップするためにゲームを一時停止する (GC スパイクとも呼ばれます) 必要があるため、不必要な、または頻繁な割り当てはパフォーマンスの低下につながります。以下に、覚えておくべきよくある落とし穴をいくつか紹介します。

— **文字列:**C# では、文字列は参照型であり、値型ではありません。つまり、新しい文字列は、たとえ一時的にしか使われないとしても、すべてマネージヒープ上に割り当てられることを意味します。不要な文字列の作成や操作は減らすようにしてください。JSON や XML のような文字列ベースのデータファイルの解析は避け、代わりに ScriptableObject や、MessagePack や Protobuf のような形式でデータを保存します。ランタイム時に文字列をビルドする必要がある場合は、StringBuilder クラスを使用しましょう。

— **Unity 関数の呼び出し:**Unity API 関数の中には、特に一時的なマネージオブジェクトの配列を返すものなど、ヒープ割り当てを行うものがあります。ループの途中で割り当てるのではなく、配列への参照をキャッシュしてください。また、ガベージの発生を避ける特定の関数も活用しましょう。例えば、**GameObject.tag** を使用して文字列を手動で比較するのではなく、**GameObject.CompareTag** を使用しましょう (新しい文字列を返すとガベージが発生するため)。

また、Project Auditor を使用して代替案を一覧表示することも可能です。これは可能な限りメモリ割り当てのないバージョンを使用するのに役立ちます。

— **ボックス化:**値型 (int、float、struct など) が参照型 (オブジェクトなど) に変換されると、ボックス化が発生します。参照型変数の代わりに値型変数を渡すことは避けてください。これは一時的なオブジェクトを作成し、それに付随する潜在的なガベージは暗示的に値型を型オブジェクトに変換するためです (例:**int i = 123、object o = i**)。代わりに、渡したい値型に

具体的なオーバーライドを提供するようにします。ジェネリックもこれらのオーバーライドに使用できます。

- **コルーチン:**yield 自体はガベージを発生させませんが、新しい WaitForSeconds オブジェクトを作成するとガベージが発生します。WaitForSeconds オブジェクトは、yield 文で作成するのではなく、キャッシュして再利用してください。
- **LINQ と正規表現:**いずれの場合も、水面下でのボックス化によってガベージが生成されます。パフォーマンスに問題がある場合は、LINQ と正規表現は避けてください。新しい配列を作成する代わりに、for ループやリストを使用しましょう。
- **ジェネリックコレクションとその他のマネージタイプ:**Update で毎フレーム List や コレクションを宣言して入力するのは避けてください (例えば、プレイヤーの一定半径内にいる敵のリストなど)。代わりに、List を MonoBehaviour のメンバーにして、Start で初期化します。コレクションを使用する前に、毎フレーム **Clear** を使用してコレクションを空にします。

ガベージコレクションを使用できるタイミングを特定する

ガベージコレクションによる停止がゲームの特定のポイントに影響しないことが確定している場合は、System.GC.Collect を使用してガベージコレクションをトリガーすることが可能です。典型的な例としては、ユーザーがメニュー画面を開いているときやゲームを一時停止したときなど、気付かれにくい状況が挙げられます。

自動メモリ管理の活用例については、自動メモリ管理を理解する を参照してください。

インクリメンタルガベージコレクターを使って GC の負荷を分散する

インクリメンタルガベージコレクションは、プログラムの実行中に 1 回の長い中断を発生させるのではなく、短い中断を複数回に分けて実行し、負荷を複数のフレームに分散させます。ガベージコレクションによってフレームレートが不規則になる場合は、このオプションを試すことで GC スパイクの問題が軽減するかどうか確認してください。また、Profile Analyzer を使用して、アプリケーションに対する効果を検証してください。

注意すべき点は、インクリメンタルモードで GC を使用すると、一部の C# 呼び出しに読み込み-書き込みのバリアが加えられる可能性があり、これにより、スクリプト呼び出しごとに 1 フレームあたり最大 1 ms のオーバーヘッドが発生する可能性があるということです。最適なパフォーマンスを得るには、メインのゲームプレイループに GC Alloc を使用しないのが理想的です。これにより、スムーズなフレームレートを実現するためにインクリメンタル GC が必要なくなり、メニューを開いたり新しいレベルをロードしたりする際など、ユーザーが気づかない場所に GC.Collect を隠すことができます。このような最適化されたシナリオでは、(System.GC.Collect() を使用して) 完全な非増分 (インクリメンタルでない) ガベージコレクションを実行できます。

Memory Profiler の詳細については、以下のリソースを参照してください。

- Memory Profiler のウォークスルーとチュートリアル
- Memory Profiler のドキュメント
- Unity の Memory Profiler によるメモリ使用量の改善
- Memory Profiler: メモリ関連の問題のトラブルシューティングツール
- Memory Profiler の操作

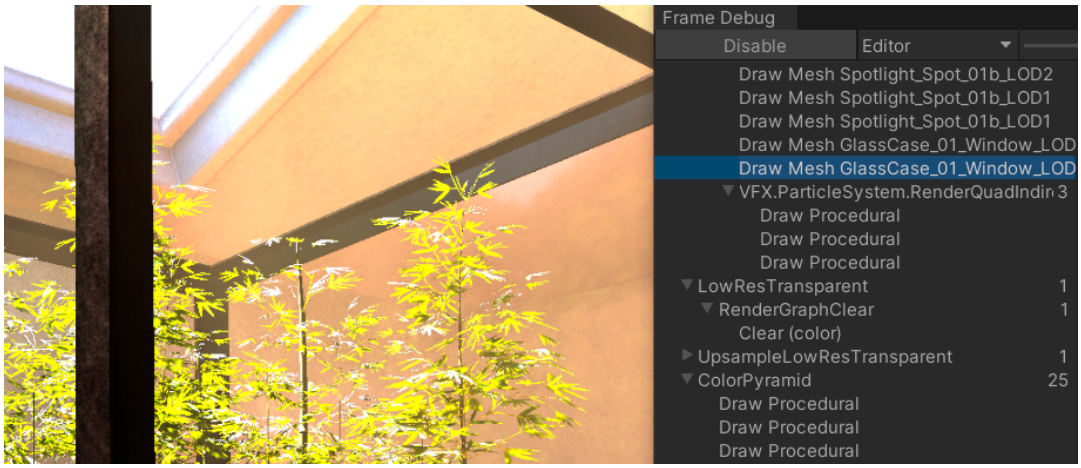
フレームデバッガー

フレームデバッガー は、実行中のゲームの再生を特定のフレームで停止し、そのレンダリングに使用された個々のドローコールを表示できるため、レンダリングのデバッグと最適化に役立ちます。ドローコールのリストを 1 つずつステップ実行することで、グラフィカルな要素からシーンを形成するためにフレームが構築される様子を確認できます。

フレームデバッガーを使用すると、あるドローコールが呼び出すゲームオブジェクトのジオメトリの場所を簡単に確認できます。フレームデバッガーにより、メインの Hierarchy パネルでゲームオブジェクトがハイライトされ、識別しやすくなります。

ドローコールを理解する:
Unity のドローコールとは、CPU から GPU に送信される、特定のマテリアルとシェーダーで特定のジオメトリのセット (メッシュ、スカイボックス、ユーザーインターフェースなど) をレンダリングするリクエストのことです。異なるオブジェクト、マテリアル、またはステートの変更をレンダリングする必要があるたびに、新しいドローコールが発行されます。

フレームデバッガー は、フレームごとにレンダリング順を分析することで、オーバードローのテストにも使用できます。詳細については、以下の 最適化のヒント を参照してください。

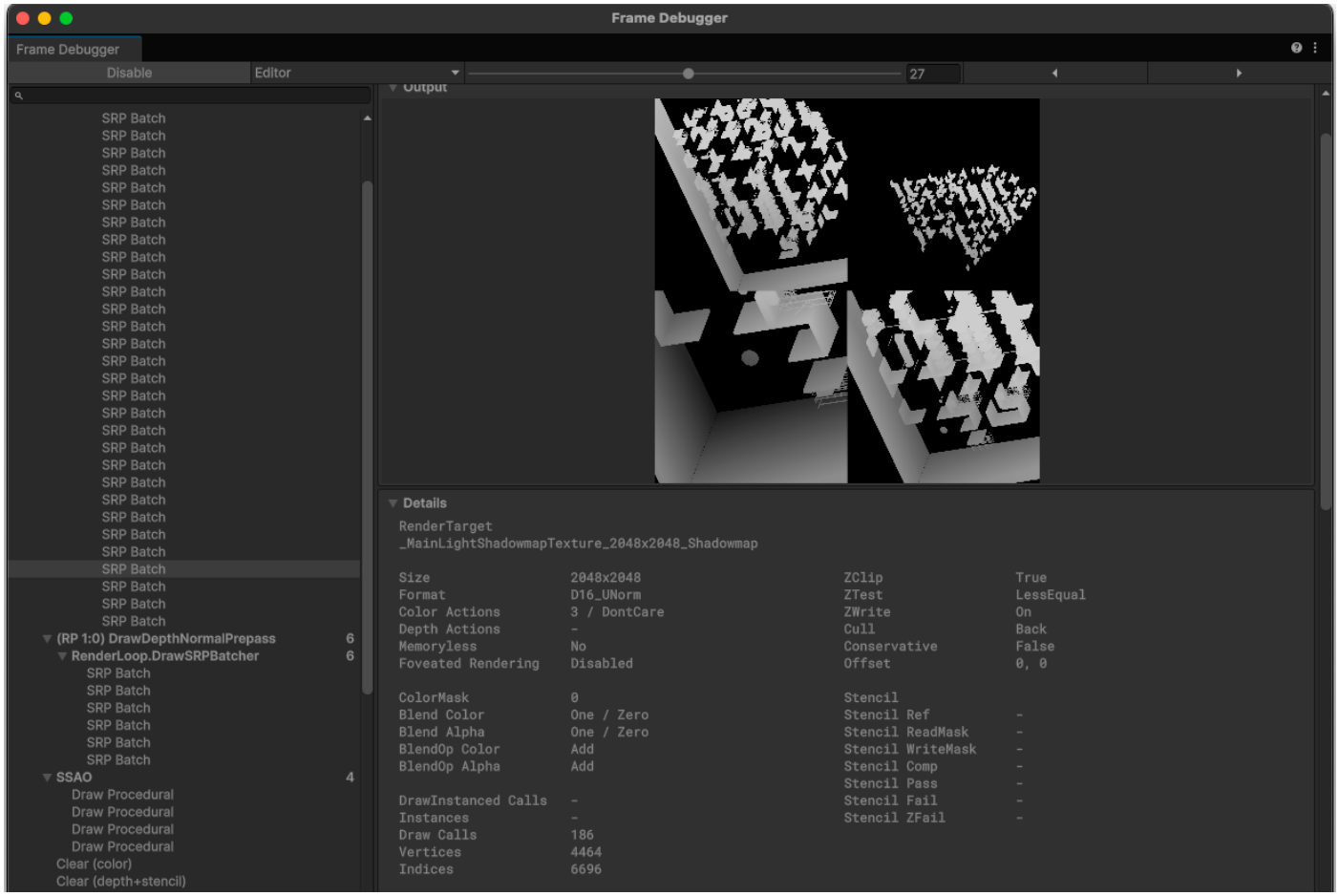


フレームデバッガーを使用して、特定されたオーバードローがどのように発生するかを分析しましょう。

Window > Analysis > Frame Debugger メニューから、フレームデバッガーを開きます。

エディターまたはデバイスでアプリケーションを実行している状態で、**Enable** をクリックします。これによりアプリケーションが一時停止し、現在のフレームのすべてのドローコールがキャプチャされ、Frame Debugger ウィンドウの左側に順番に一覧表示されます。キャプチャには、フレームバッファの消去イベントなど、さらに詳細が含まれるようになります。

Debugger ウィンドウの上部にあるスライダーを使用すると、ドローコールを通してすばやくスクラブし、関心のある項目をすばやく見つけることができます。



Frame Debugger ウィンドウには、ドローコールとイベントが左側に一覧表示され、それぞれを視覚的にステップ実行するためのスライダーが用意されています。

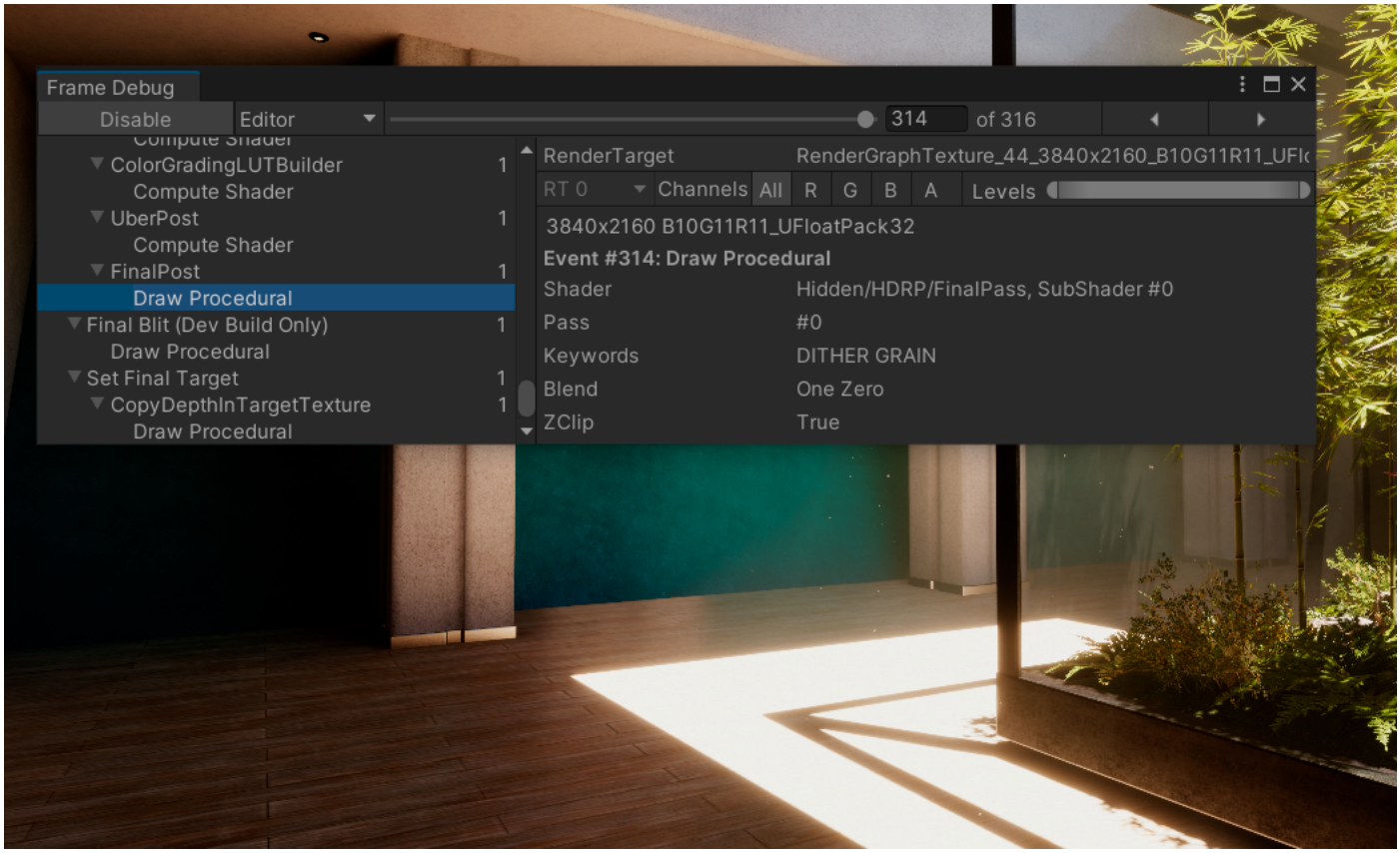
Unity は CPU からグラフィックス API にドローコールを発行し、画面にジオメトリを描画します。ドローコールは、何をどのように描画するかをグラフィックス API に指示します。各ドローコールには、テクスチャ、シェーダー、バッファに関する情報など、グラフィックス API が必要とするすべての情報が含まれています。多くの場合、ドローコールの準備はドローコール自体よりも多くのリソースを必要とします。

この準備プロセスはレンダーステートと呼ばれ、変更を最小限に抑えることでパフォーマンスを向上させることが可能です。

フレームデバッガーを使用してドローコールの発生場所を特定し、レンダリングプロセスを視覚化しましょう。これは、ドローコールをグループ化してレンダーステートの変更を減らす方法を決めるための情報として役立ちます。

フレームデバッガーのリスト階層を参照し、対象ドローコールがどこで発生しているかを特定します。リストから項目を選択すると、そのドローコールまでのシーンが Game ウィンドウに表示されます。

ドローコールを最小限に抑えることは、特にモバイルや VR プラットフォームでは、コールのたびに CPU 負荷が増加するため、パフォーマンス改善に必要不可欠です。バッチ処理、GPU インスタンシング、テクスチャアトラスなどの手法は、マテリアルやメッシュを共有するオブジェクトを組み合わせることで、ドローコールの数を減らすのに役立ちます。



Game ウィンドウには、フレームデバッガーで選択したドローコール (ポストプロセスエフェクトの適用終了付近) までで構成されたシーンフレームが表示されます。リスト階層の右側にあるパネルには、ジオメトリの詳細やレンダリングに使用されるシェーダーなど、各ドローコールに関する情報が表示されます。

他にも、ドローコールを以前のものと同じバッチで処理できなかった理由や、シェーダーに供給された正確なプロパティ値の内訳など、役立つ情報があります。

リモートフレームデバッグ

サポートされているプラットフォームで実行中の Unity Player にフレームデバッガーをアタッチすることで、フレームをリモートでデバッグできます (WebGL はサポートされていません)。デスクトッププラットフォームでは、ビルドの **Run In Background** を有効にします。

リモートフレームデバッグを設定するには、次の手順に従います。

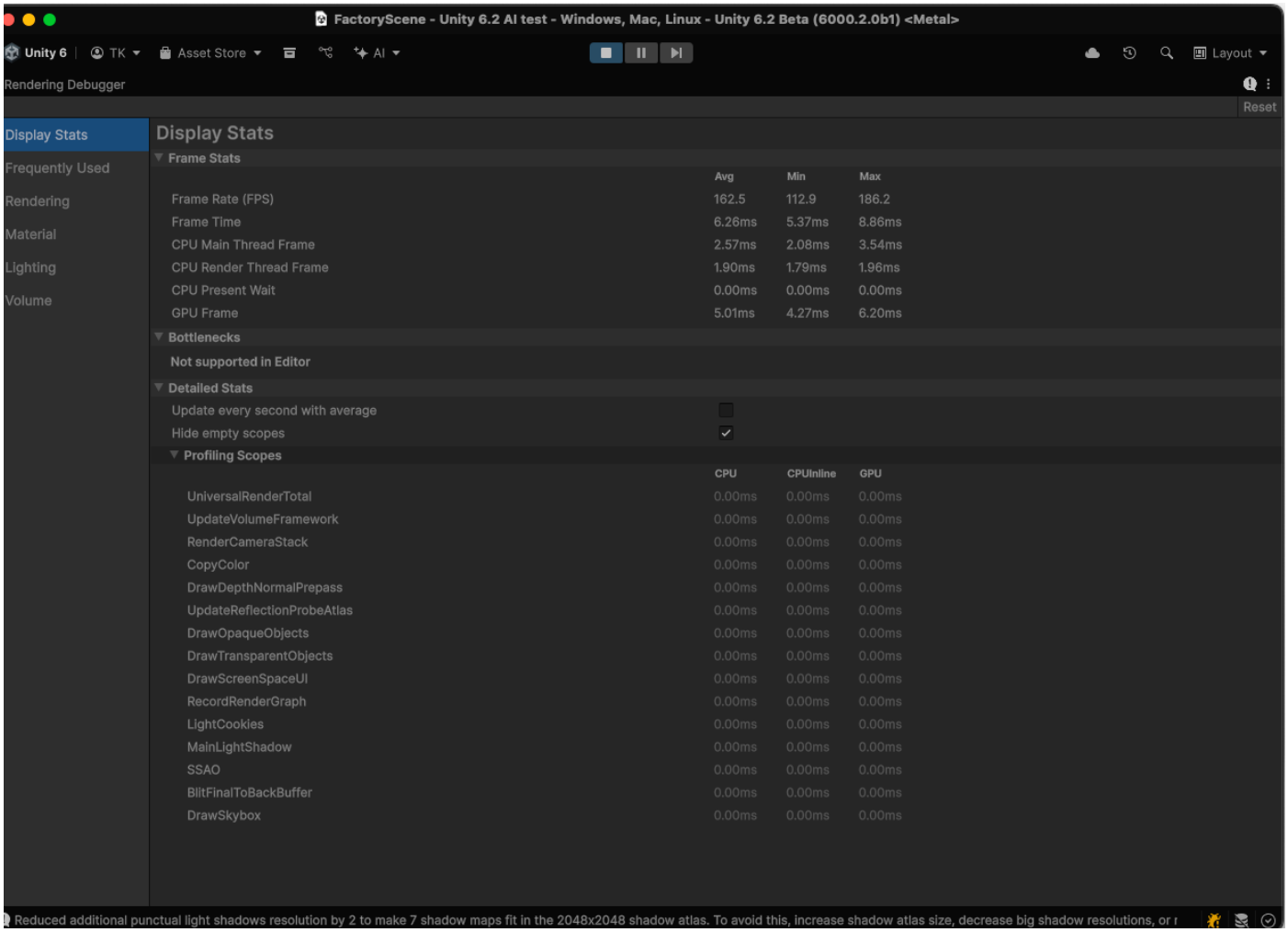
1. ターゲットプラットフォームに対してプロジェクトの標準ビルドを作成します (Development Build を選択します)。
2. プレイヤーを実行します。
3. エディターから Frame Debugger ウィンドウを開きます。
4. Player 選択のドロップダウンをクリックし、実行中のアクティブなプレイヤーを選択します。
5. Enable ボタンをクリックします。

これで、フレームデバッグリスト階層内のドローコールとイベントをステップ実行し、アクティブなプレイヤーで結果を観察できます。

レンダリングデバッガー

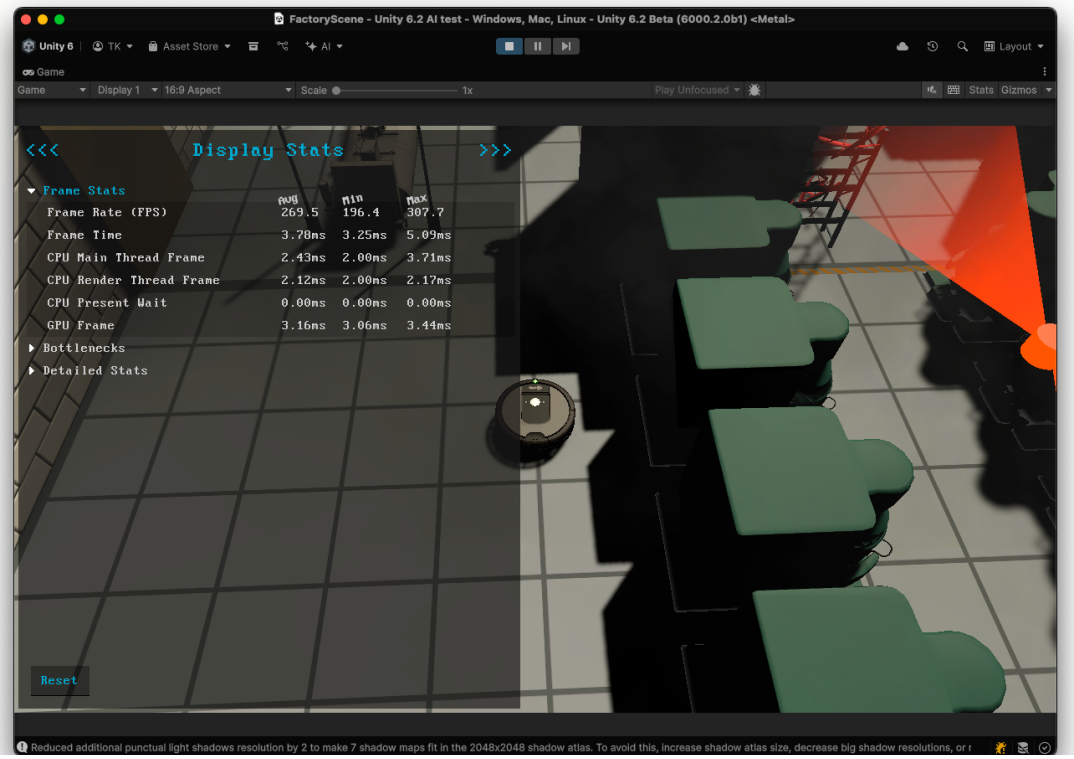
レンダリングデバッガー には、オーバードロー、ライティングの複雑さ、レンダリング、マテリアルのプロパティに関する情報を表示する複数のデバッグビューとモードが用意されており、レンダリングの問題を特定し、URP と HDRP のシーンを最適化できます。

このツールを開くには、エディターで **Window > Analysis > Rendering Debugger** を選択します。再生モードまたはデスクトッププレイヤーのビルドでは、**左 Ctrl + Back Space** (**macOS では 左 Ctrl + Delete**) のショートカットを使用することもできます。



レンダリングデバッガーを使用すると、ライティング、レンダリング、マテリアルのさまざまなプロパティを視覚化できるため、レンダリングの問題を特定してシーンを最適化できます。

これは、レンダリングに関連する視覚的なアーティファクトやパフォーマンスのボトルネックの診断に役立ちます。詳細な統計情報を含むウィンドウは開発ビルドでのみ利用可能で、パイプライン固有ではないシェーダーや外部レンダリングオブジェクトとの相互作用に制限があることに注意してください。



レンダリングデバッガーウィンドウは、エディターで開くか、ゲームビュー、再生モード、またはビルドしたアプリケーションでオーバーレイとして開くことができます。

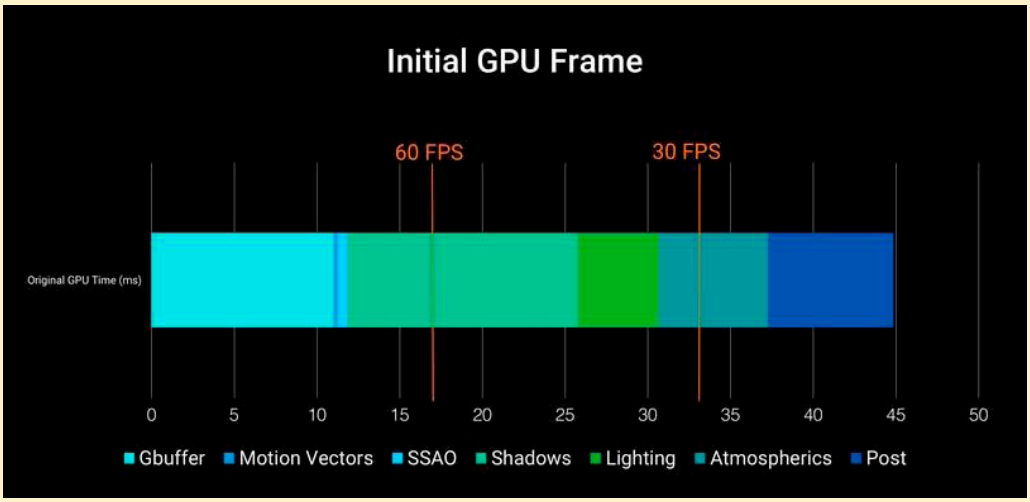
よくある落とし穴に対する 5 つのレンダリング最適化

フレームデバッガーやその他のレンダリングデバッグツールを使用して特定できる一般的なレンダリングパフォーマンスの問題を最適化するにあたり、これらのヒントとテクニックを活用してください。

最初にパフォーマンスのボトルネックを特定する

まずは、GPU ロードが高いフレームを見つけます。ほとんどのプラットフォームは、CPU と GPU の両方でプロジェクトのパフォーマンスを分析するための安定したツールを提供しています。例えば、Arm ハードウェア/Immortalis および Mali GPU 向けの Arm Performance Studio、Microsoft Xbox 向けの PIX、Sony PlayStation 向けの Razor、Apple iOS 向けの Xcode Instruments などです。

それぞれのネイティブプロファイラーを使って、フレームにかかっているコストを個別のパーツに分解します。これは、グラフィックスのパフォーマンスを向上させるための出発点となります。



この画面は GPU 依存の PS4 Pro のもので、1 フレームあたり約 45 ms でした。

ドローコールの最適化

PC や現世代のコンソールのハードウェアは、多くのドローコールをプッシュできますが、各ドローコールのオーバーヘッドはまだ高く、削減する努力が必要とされています。モバイルデバイスでは、多くの場合、ドローコールの最適化は不可欠です。ドローコールのバッチ処理 は、メッシュを結合して、Unity がより少ないドローコールでレンダリングできるようにする最適化手法です。

フレームデバッガーは、上で説明したように、最適なグループやバッチに再編成できるドローコールの特定に役立ちます。また、このツールは特定のドローコールをバッチ処理できない理由を突き止めるために利用することもできます。

ドローコールのバッチ数を減らすテクニックには、以下のようなものがあります。

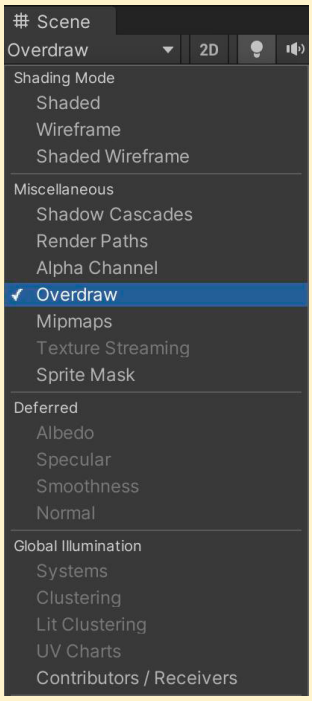
- **オクルージョンカリング** は、手前のオブジェクトの後ろに隠れたオブジェクトを削除し、見えない要素のオーバードロー (透明なオブジェクトが重なるために GPU が同じピクセルを複数回にわたって描画すること) を減らします。ただし、これには追加の CPU 処理が必要となるため、プロファイラーを使用して、GPU から CPU へ作業を移すことが有益かどうか、別のボトルネックが生じないかどうかを確認してください。
- **GPU インスタンシング** は、同じメッシュやマテリアルを共有する多くのオブジェクトをより少ないバッチでレンダリングすることで、ドローコールを減らします。パフォーマンスコストを抑え、ビジュアルの繰り返しを最小限に抑えながら、複雑なシーンを作成できます。
- **SRP Batcher** は Bind と Draw の GPU コマンドをバッチ処理することで、ドローコール間の GPU 設定を削減します。SRP バッチ処理の恩恵を受けるには、必要な数のマテリアルを使用しつつ、互換性のある少数のシェーダーバリアント (例: URP や HDRP の Lit シェーダーと Unlit シェーダー) に限定し、キーワードの組み合わせによる差異を可能な限り少なくしてください。
- **GPU Resident Drawer** は、GPU インスタンシングを使用して多くのゲームオブジェクトを描画するため、ドローコールの数を大幅に削減できます。これにより、より多くのレンダリング作業負荷を GPU にシフトすることで CPU 処理時間を解放し、特に類似オブジェクトが多数存在するシーンにおいてパフォーマンスが向上します。

オーバードローを減らし、フィルレートを最適化

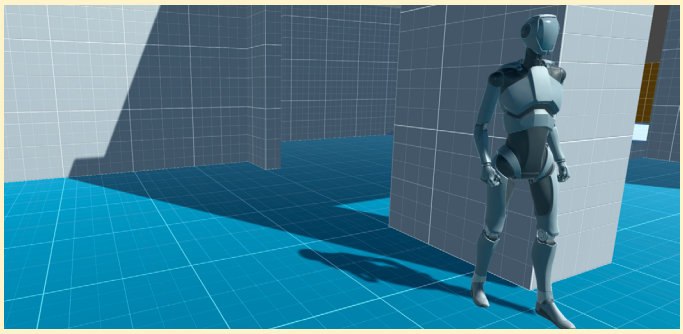
オブジェクトが重なってレンダリングされると、オーバードローとなります。オーバードローは、アプリケーションによって描画されようとしているフレームあたりのピクセル数が、GPU で処理しきれないほど多いことを示している可能性があります。パフォーマンスだけでなく、モバイルデバイスの温度やバッテリーの寿命も悪影響を受けます。オーバードローを防止するには、Unity でオブジェクトがレンダリングされる前にどのようにソートされるかを理解することが重要です。

ビルトインレンダーパイプラインは、ゲームオブジェクトを レンダリングモード と レンダーキュー に基づいてソートします。各オブジェクトのシェーダーは レンダーキュー に配置され、多くの場合、このキューが描画順を決定します。

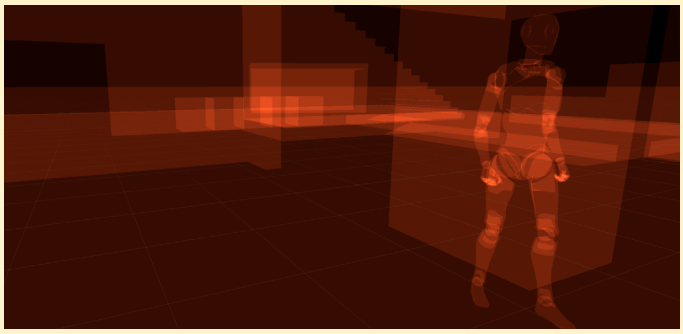
ビルトインレンダーパイプラインを使用している場合は、シーンビューのコントロールバー からオーバードローを可視化できます。描画モードを Overdraw に切り替えてください。



シーンビューのコントロールバーにある Overdraw



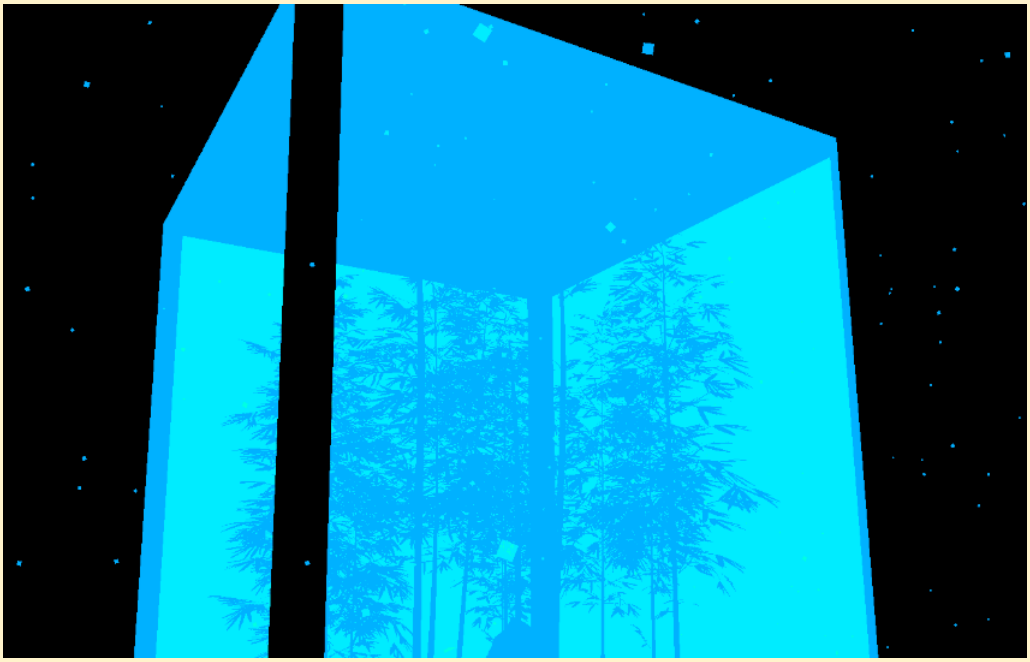
標準的な Shaded ビューのシーン



上の画像と同じシーンの Overdraw ビュー。
ジオメトリの重なりは、しばしばオーバードローの原因となります

HDRP では、レンダーキューの操作方法が多少異なります。このアプローチの詳細を理解するには、レンダラーとマテリアルプライオリティ のセクションを参照してください。

前述したように、URP と HDRP では、レンダリングデバッガーを使用してオーバードローを特定することが可能です。



HDRP とフルスクリーンデバッグモードによるオーバードローの視覚化

最も負荷の高いシェーダーを調べる

これは深い話題ですが、一般的には、可能な限りシェーダーの複雑さを減らすことを目指してください。ここでの簡単な改善策としては、可能な限り精度を下げることが挙げられます。つまり、可能であれば半精度浮動小数点数の変数を使用してください。また、ターゲットプラットフォームの ウェーブフロント占有率 や、GPU プロファイリングツールを使用して適切な占有率を得る方法についても学ぶことができます。

レンダリングのためのマルチコア最適化

Player Settings > Other Settings で Graphics Jobs を有効にすると、デスクトップとコンソールでマルチコアプロセッサーを利用できます。Graphics Jobs によって、Unity でレンダリング作業を複数の CPU コアに分散させ、レンダースレッドへの負担を軽減できるようになります。詳細については、Multithreaded Rendering & Graphics Jobs チュートリアルをご覧ください。

ポストプロセスエフェクトのプロファイリング

ポストプロセッシングアセットがターゲットプラットフォームに合わせて最適化されていることを確認しましょう。元々 PC ゲーム用にオーサリングされた Unity Asset Store のツールは、コンソールやモバイルデバイスでは必要以上にリソースを消費する場合があります。ネイティブのプロファイラーツールを使用して、ターゲットプラットフォームをプロファイリングしましょう。モバイルやコンソールのターゲット用に独自のポストプロセスエフェクトを作成するときは、できるだけシンプルにしましょう。

フレームのデバッグと分析に役立つツールは他にも多数あります。プロファイリングおよびデバッグツールのインデックス を見て、さらにアイディアを得ましょう。

Unity フレームデバッガーの詳細については、以下のリソースを参照してください。

— Unity フレームデバッガー のドキュメント

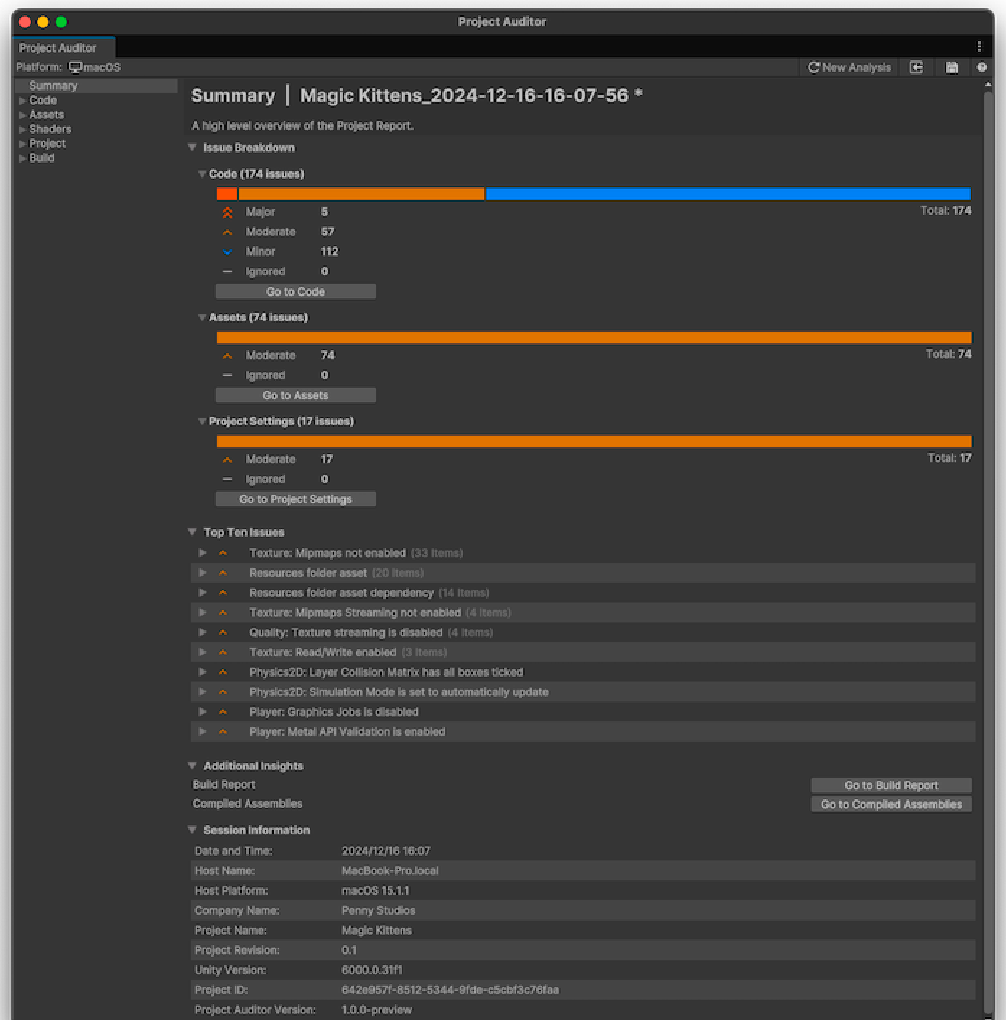
— フレームデバッガーの使い方

— レンダリングのプロファイリング[illegible]

Project Auditor

Project Auditor は、Unity 6.1 でパッケージとして導入された、Unity プロジェクト用の強力な分析ツールです。開発者がパフォーマンスを最適化し、ベストプラクティスを維持し、プロジェクトに潜む問題やボトルネックを特定できるよう設計されています。

Project Auditor は、プロジェクト全体をスキャンし、スクリプト呼び出しが重い、アセットが使用されていない、エンティティ数が多すぎるなどの非効率な点に関する詳細なレポートを提供します。



Project Auditor の Summary ビュー

Project Auditor はいくつかの異なる領域に対応しています。

- **パフォーマンスの最適化:**過剰なガベージ生成、不要なオブジェクト割り当て、高コストな関数呼び出しなど、プロジェクトのランタイムパフォーマンスに影響を与える可能性がある問題を特定します。
- **コードとアセットレビュー**:未使用のアセット、非効率的なコードパターン、リファクタリング可能な古い API などがハイライト表示されます。これにより、ビルドサイズが小さくなり、プロジェクト全体のメンテナンス性が向上し、メモリの使用が最適化されます。
- **診断とベストプラクティス**:Unity のベストプラクティスに基づいた推奨事項を提供し、参照漏れ、最適でないプレイヤー設定や品質設定など、プロジェクト設定に関連するエラーや警告をハイライトします。
- **カスタマイズ可能なレポート**:結果をカテゴリ別に整理し、最適化の優先順位付けを容易にします。また、カスタムルールを作成して、特定のプロジェクトやニーズに合わせて分析を調整することも可能です。

💡**ヒント**:

- 開発の重要な段階 (マイルストーン、ベータ版リリース、最終ビルドの前など) で Project Auditor を実行します。定期的に監査を行うことで、パフォーマンスのボトルネック、未使用のアセット、古いコードを早期に発見し、プロジェクトの規模が大きくなるにつれて問題が拡大するのを防ぐことができます。
- Project Auditor は、CI またはビルド設定の一部として自動化できます (マニュアルの こちら を参照)。また、レポートを使用して、新しい問題を生じさせるアセットやコードをチェックインしないようにすることも可能です (使用する API の詳細は こちら を参照)。
- テクスチャ設定、サイズ、より複雑なルールなど、ゲーム内で確実に捕捉したい事項がある場合は独自のルールを追加できます。この方法の詳細については、マニュアルのこちらのページ を参照してください。

Project Auditor によって生成されるレポートは、重大度 (Major、Moderate、Info) 別に分類されます。最も深刻な問題にまず焦点を当ててください。なぜなら、それらは、メモリの過剰割り当てや過剰なガベージコレクションなど、パフォーマンスに重大な影響を与える問題を浮き彫りにすることが多いからです。また、このような問題は Update などの頻繁に呼び出されるコードパスにも存在している可能性が高く、パフォーマンスへの影響がプレイヤーにより顕著に感じられるでしょう。

Project Auditor は **プレイヤー設定** や **品質設定** などの設定もチェックし、変更内容に関する推奨事項を提示します。これを使用して、ビルドターゲット、解像度、テキスト圧縮、その他のプロジェクト設定が意図したプラットフォームに合わせて確実に最適化されるようにしましょう。

Domain Reload

Unity エディターでは、再生モードの開始に関する設定を行うことができます。このページ で詳しく説明しますが、多くの場合、Domain Reload を無効にすることでエディターのイテレーション時間を短縮できます。ただし、これでは再生モードに入るたびに実施されていたスクリプティングステートのリセットが実施されなくなるため、コード内で手動で行う必要があります。

Project Auditor の Code エリアでは、プロジェクト内のスクリプトを分析し、スクリプト変数をリセットする必要がある箇所を見つけ出すことができます。ベストプラクティスは、Domain Reload ビューに表示されるすべての問題を修正してから Domain Reload を無効にすることです。このビューにデータを取り込むには、Preferences ウィンドウの Use Roslyn Analyzers 設定を有効にする必要があります。その後、マニュアルの手順 に従って問題のリストを確認し、問題を修正します。 これらがすべて解決されたら、再生モードに入るときに Domain Reload を無効にすることが可能です。

ディーププロファイリング

プロファイリング入門 のセクションで説明したように、デフォルトでは、Unity でプロファイリングされるのはプロファイラーマーカーで明示的にラップされたコードのみです。これには、エンジンのネイティブコードによって呼び出されるマネージコードの最初のコールスタック深度も含まれます。

ディーププロファイリングを有効にすると、各関数呼び出し の最初と最後にプロファイラーマーカーが挿入されます。これにより、非常に細かい部分までキャプチャすることが可能です。コールスタックが十分に表示されない長いプロファイラーマーカーの中で何が起こっているかを確認するには、Deep Profile 設定を使用してください。

ゲームのパフォーマンスを測定するには、スナップショットベースのプロファイリング (サンプルプロファイリング、細部をキャプチャできない可能性があります) よりも、このようなきめ細かなアプローチのほうが好ましい場合があります。

問題のあるコード領域を手動でインストルメント化する方法としては、ProfileMarker API を忘れずにチェックしてください。

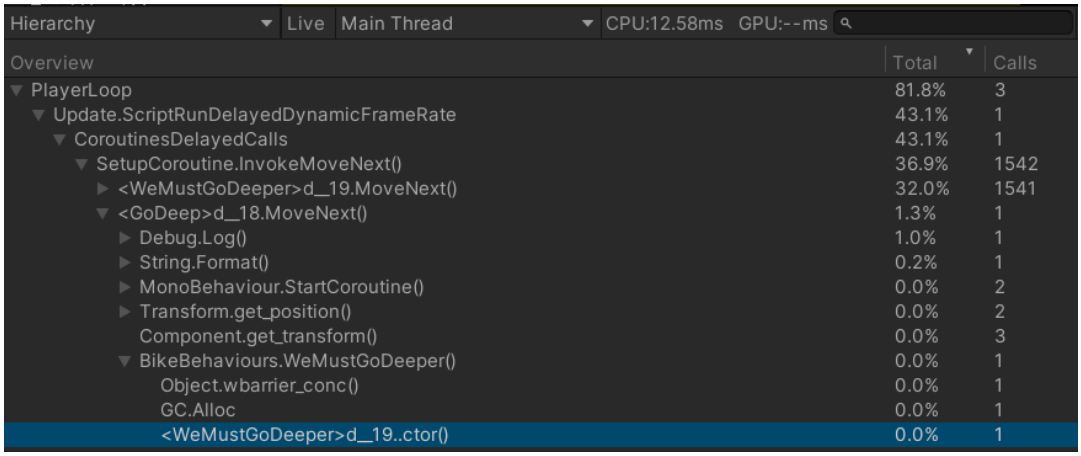
ディーププロファイリングよりもパフォーマンスへの影響をはるかに低くできます。場合によっては、ディーププロファイリングを有効にしてゲームのテストしたい部分に移動するよりも、ProfileMarker を追加してゲームをリビルドした方が早いこともあります。

デバイスビルドで完全なコールスタックを取得する別の方法は、ネイティブの CPU 使用率プロファイラーを実行することです。場合によっては、ディーププロファイリングよりも簡単で、パフォーマンスへの悪影響が少ないこともあります。

ディーププロファイリングを使用するタイミング

Deep Profile 設定は、アプリケーションまたはマネージコードのさらに詳細に調査する必要がある具体的な部分を特定した後にのみ有効にしてください。ディーププロファイリングはリソースを大幅に消費し、大量のメモリを使用します。有効にするとアプリケーションの実行速度が遅くなります。

ディーププロファイリングにより、コールツリーを詳しく調べ、コードの非効率性や問題を発見できます。



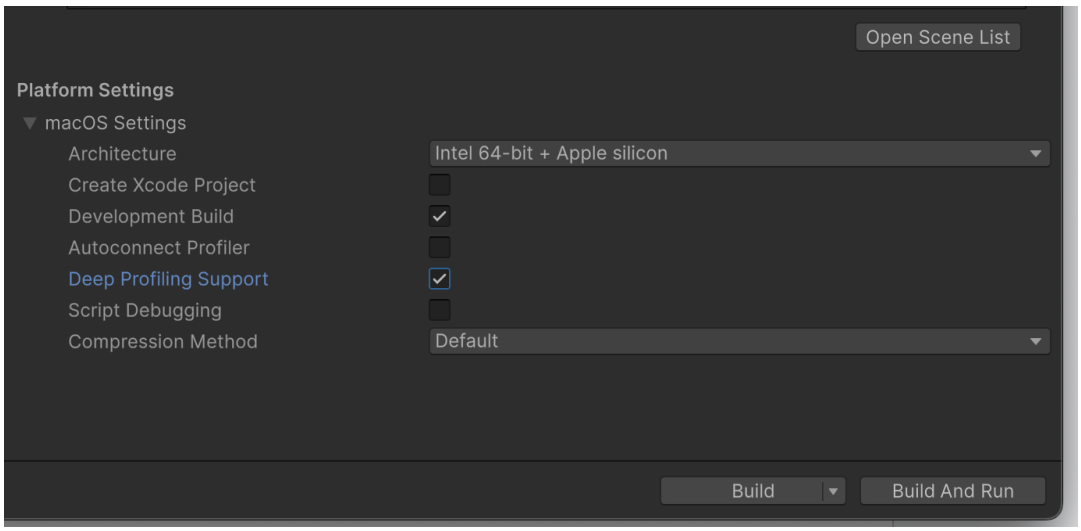
ディーププロファイリングは、特定の問題を追跡し理解する必要がある場合に、膨大な詳細情報を明らかにします。

ただし、ディーププロファイリングでは、すべての関数呼び出しの開始地点と終了地点にマーカーが施され、それぞれのマーカーがオーバーヘッドを増加させます。これにより、コードの深いコールスタックがある部分 (MyDeepFunction など) は、1 つの関数 (MySingleFunction) 内ですべての処理を行う場所よりもコストが高くなります。つまり、コードのこれら 2 つの部分の相対的なタイミングに頼ることはできません。MyDeepFunction は、ディーププロファイリングを有効にした MySingleFunction よりも高コストに見えるかもしれませんが、このコストはすべて余分に挿入されたマーカーによるものである可能性があります。

注意:Unity 2019.3 以降では、Mono バックエンドと IL2CPP バックエンドの両方でのディーププロファイリングのサポートが追加されました。これは、iOS などの IL2CPP が必須のプラットフォームにとっては朗報です。

ディーププロファイリングの使用

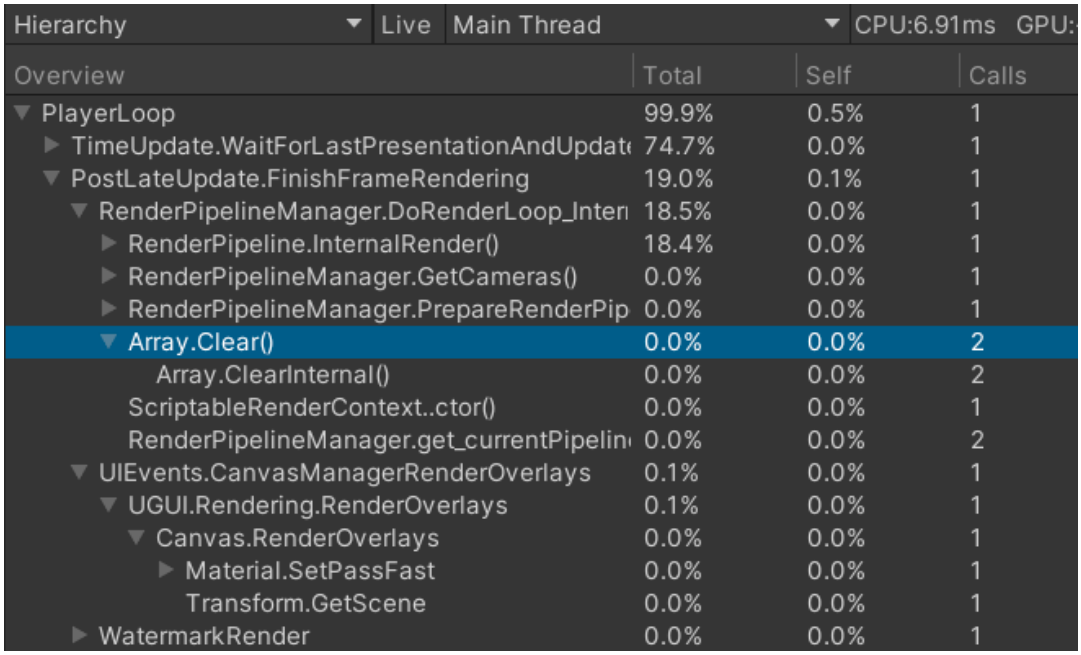
プレイヤービルドでディーププロファイリングを使用するには、**File > Build Settings > Enable Deep Profiling Support** でディーププロファイリングを有効にする必要があります。



Deep Profiling Support の有効化

サポートが有効になったら、必要に応じて、Profiler ウィンドウでビルドのディーププロファイリングのオンとオフを簡単に切り替えることができます。

プレイヤーにアタッチされたときに Deep Profile ボタンがフェードアウトする場合、ビルドで Deep Profiling Support が有効になっていません。



ディーププロファイリングにより、アプリケーションコードのパフォーマンスとタイミングについて、非常に多くの詳細な情報が明らかになります。メソッド呼び出しツリー全体が表示され、マネージ割り当てがどこで発生しているかを掘り下げるのに役立ちます。

ディーププロファイリングのヒント

上から下へのアプローチ

アプリケーションをプロファイリングするときは、まず概要レベルから始め、ディーププロファイリングを使わずにパフォーマンスを改善できる領域を見つけるようにしましょう。より詳細な情報が必要な場合は、プロファイラーでディーププロファイリングを有効にして、より細かい粒度で掘り下げることができます。このアプローチを使用すると、プロファイラーの Hierarchy に表示される情報のレベルを最小限に抑え、目の前の目標に集中できます。

必要な場合にのみディーププロファイリングを実施

一般的に、ディーププロファイリングは、コードのパフォーマンスについて詳細情報をより細かく取得する必要がある場合に使用するのが最適です。ビルドの Deep Profiling フラグを有効にしたままにしておいても、実際に機能を有効化しない場合はパフォーマンスに影響はありませんが、有効にするとアプリケーションの実行速度が遅くなります。

コード内のマネージ割り当てのソースのみを見つける必要がある場合、Unity 2019.3 以降のバージョンでは、ディーププロファイリングを有効化せずにこれを実行できることに留意してください。プロファイラーの Call Stacks トグルと Calls ドロップダウンを使用して、マネージ割り当てを探してください。

自動化されたプロセスにおけるディーププロファイリング

コマンド行からプロファイリング時にディーププロファイリングをオンに切り替えるには、ビルドの実行ファイルに ***-deepprofiling*** 引数を加えます。Android/Mono スクリプティングバックエンドビルドでは、次のように adb コマンドライン引数を使用します。

adb shell am start -n com.company.game/com.unity3d.player.UnityPlayerActivity -e ‘unity’ ‘-deepprofiling’

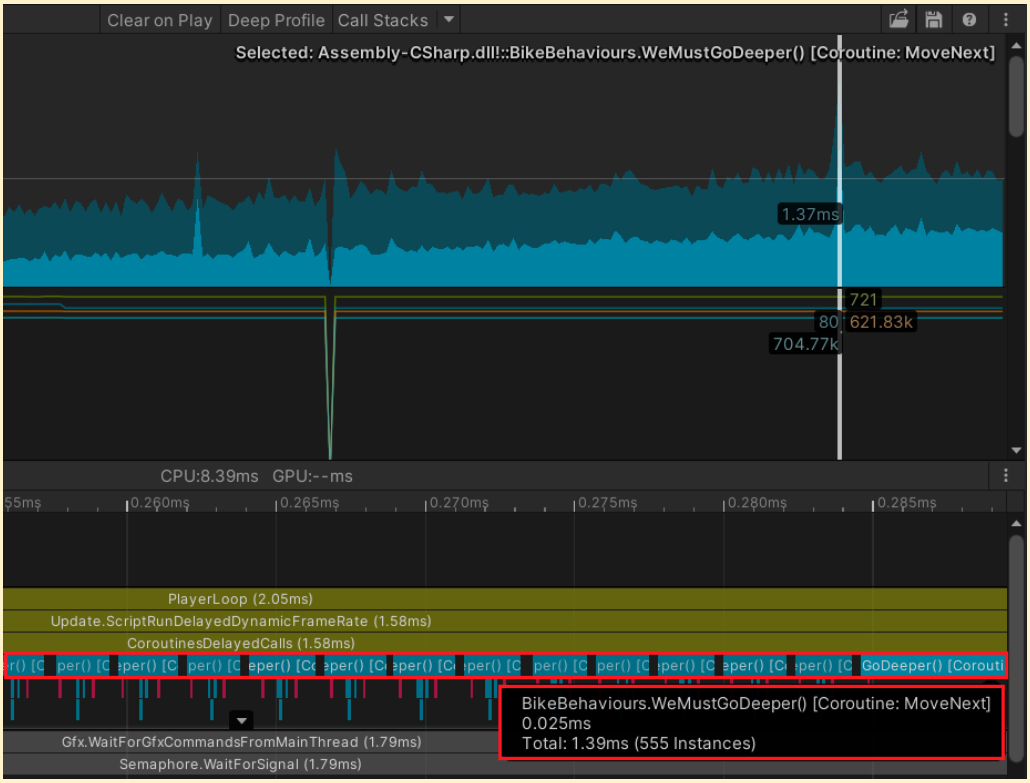
低スペックハードウェアでのディーププロファイリング

スペックの低いハードウェアでは、メモリとパフォーマンスが制限され、ディーププロファイリングの使用に影響を与える可能性があります。Unity のプロファイラーのサンプルを保存するリングバッファは、低速のデバイスで Deep Profile 設定を使用するといっぱいになることがあります。この場合、Unity 上でエラーメッセージが表示されます。

Profiler.maxUsedMemory の プロパティ (bytes) を設定することで、このバッファリングデータのプロファイラーにより多くのメモリを割り当てることが可能です。デフォルトは、プレイヤーの場合は 128 MB、エディターの場合は 512 MB です。ディーププロファイリング時に問題が発生した場合は、低速デバイスでのプレイヤービルドで必要に応じて割り当て量を増加してください。

ディーププロファイリングによるオーバーヘッドのせいで動作が遅すぎる (あるいは全く動作しない) ハードウェア上で、コードをより詳細にプロファイリングする必要がある場合、独自のマーカーを使用してより深いプロファイリングを行うことが可能です。

Deep Profile 設定を有効にするのではなく、コードの特定の領域に プロファイラーマーカー を追加してください。これらのマーカーは、CPU Usage モジュールを表示しているときに、プロファイラーの Timeline または Hierarchy に表示されます。



ディーププロファイリングによるオーバーヘッドが大きすぎる場合は、プロファイラーマーカーを加えて、より深いコードレイヤーのプロファイリングを行いましょう。

いつ、どのプロファイリングツールを使用するべきか

プロファイリングは、プロジェクトライフサイクルの初期段階で開始した場合に最大の効果を発揮します。早期に開始することにより、ゲームやアプリケーション開発のより多くのチェックポイントで、比較に役立つベースラインを確立できます。"プロファイリングのツールキット" のどのツールを、いつ選択すべきかを知ることが重要です。

各ツールの用途とメリットを理解しておくと、いつ使用するべきか把握しやすくなります。Unity が提供する各プロファイリングツールについては、Unity のプロファイリングおよびデバッグツール のセクションで確認してください。

"いつ使うべきか" に答えを出しやすくなるように、プロジェクトのライフサイクルにおけるチェックポイントのアイデアをまとめました。プロファイリング戦略を立てる際に参考にしてください。

— **プロトタイピング**:プロファイリングは、プロジェクトのプロトタイプ段階でリスクを軽減するために重要です。ゲームの設計ドキュメントで 10,000 体の敵が画面上に表示することが求められている場合、ターゲットプラットフォーム上でそれができることを証明するプロトタイプをビルドし、プロファイリングを行う必要があります。それができない場合、設計を変更する必要があります。

— **プロジェクトの初期段階**:選択したターゲットデバイスハードウェア全体で、プロジェクトパフォーマンスのベースラインを確立します。Memory Profiler を使用してメモリ使用量を大まかに把握し、プロジェクトの範囲の計画が、ターゲットハードウェアのメモリ制限によって後々問題が生じる水準に向かっていない

ことを確認してください。

— **スプリント終了**:アジャイルチームでスプリントで作業している場合、標準化された一連のプロファイリングツールを実行するのに最適なタイミングは、スプリント終了時のリリース候補版（RC）です。例えばデータベースやスプレッドシートなどに、結果や指標を記録するための標準フォーマットがあることを確認してください。Unity プロファイラーを使用して、以下のプロファイリングアクティビティとデータキャプチャを実行します。

  — CPU 使用率
  — GPU 使用率
  — メモリ使用量
  — レンダリング
  — 物理演算

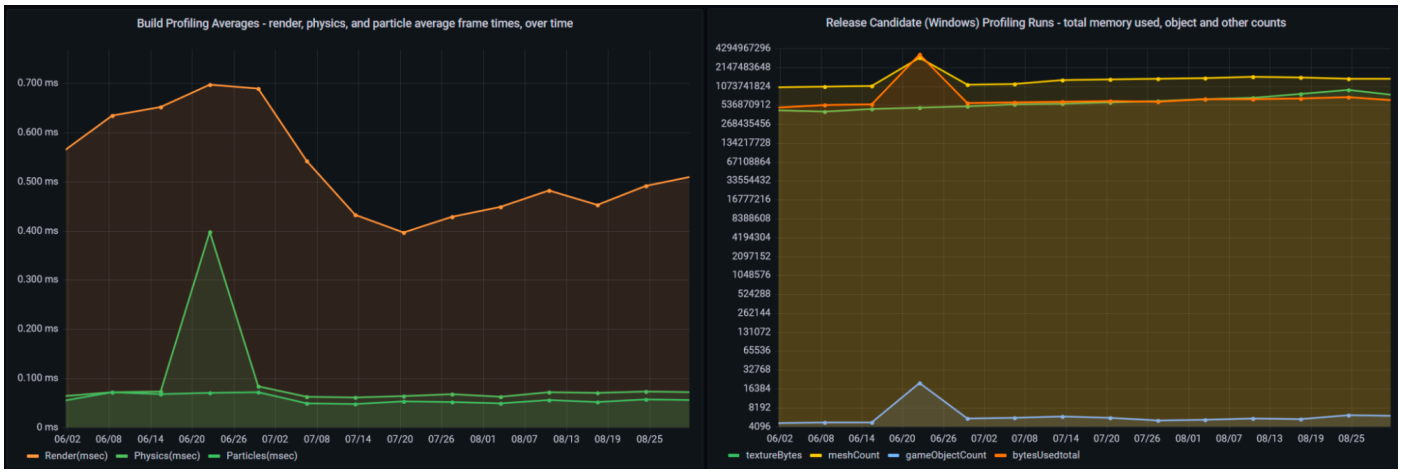
さらに深く掘り下げ、以下のツールを使用して、結果および大きな相違のあった指標（以前のスプリントリリースとの比較で）を記録します。

— **Profile Analyzer**:以前のリリースのプロファイリングデータをキャプチャしたものをロードし、差分を比較して記録します。
— **Memory Profiler**:以前のリリース候補ビルドのメモリスナップショットを比較し、メモリの増減の違いを記録します。

主要なパフォーマンス指標およびプロファイリング指標の自動化

一般的なタスクや繰り返し発生するタスクを自動化し、プロジェクトのプロファイリングとデータキャプチャをレベルアップさせましょう。これにより、時間を節約し、常に最新の指標を活用できます。

指標をグラフ化してプロジェクトダッシュボードに追加することで、パフォーマンスが急激に低下している部分（新しく追加された機能やバグなど）や、最適化とバグ修正のスプリント後に改善された部分を確認できます。

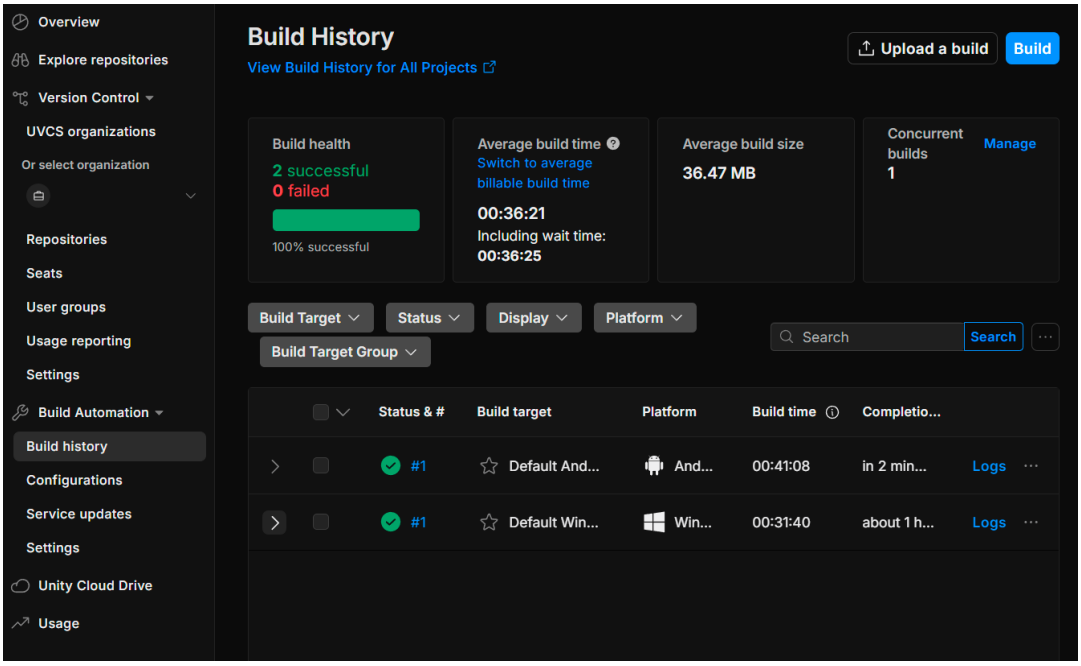


毎週のビルドプロファイリングデータのキャプチャを自動化し、Grafana ダッシュボードで可視化したもの

ゲームの全レベルにわたる、プロジェクトの全体的なメモリ使用プロファイルを、時間の経過とともに図示しましょう。Memory Profiler でメモリスナップショットをキャプチャし、すべてのレベルの数値を平均化することで、ターゲットデバイス/プラットフォーム、スプリント、リリースサイクルごとにメモリフットプリントを記録できます。

全体的なプロファイリングの統計を記録したい場合は、ProfilerRecorder を使用して予約済みメモリ合計やシステム使用メモリなどの指標を記録し、CI (継続的インテグレーション) に出力し、グラフや Grafana などのグラフツールに転送しましょう。

Unity DevOps ツール を使用してリリースビルドの作成を自動化し、このプロセスを自動化されたデバイスプロファイリングワークフローと統合しましょう。



ビルドの結果は、Unity Cloud ダッシュボードから利用できる **Build Automation > Build History** 機能で確認できます。いずれかのビルドが失敗した場合、ログを確認してトラブルシューティングを行うことができます。プロジェクトの整理と Unity Services の詳細については、e-book *プロジェクト整理とバージョン管理のベストプラクティス (Unity 6 版)* を参照してください。

自動化されたプロファイリングパイプラインの例

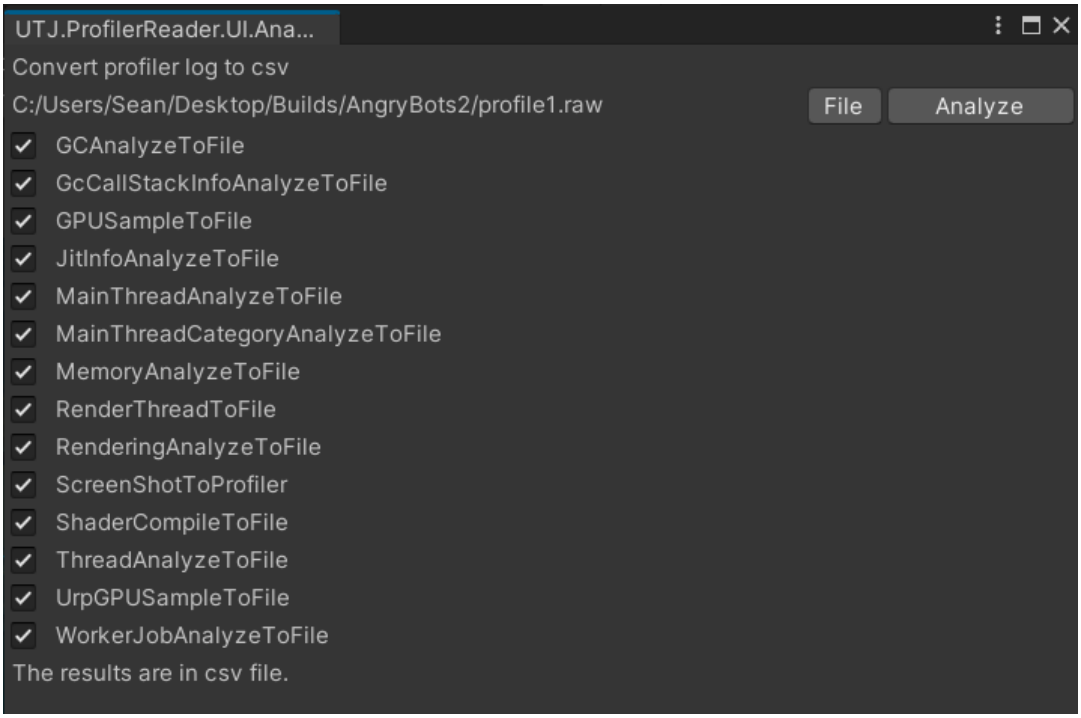
自動化により、時間的制約によってこのプロセスが後回しになる心配をすることなく、チームがビルドのプロファイリングのメリットを確実に享受できるようになります。

このワークフロー例では、ビルドプロファイリングを頻繁かつ正確に行うために自動化を使用する方法を紹介します。

- Unity Build Automation を使用して自動ビルドリリースを作成します。
- 各リリース後、以下のようなスクリプトを使用してビルドプレイヤーを起動し、2000 フレームを超える生のプロファイリングデータをキャプチャします。
    - **AngryBots2.exe -profiler-enable -profiler-log-file profile1.raw -profiler-capture-frame-count 2000**コマンドライン引数の詳細については、Unity の ドキュメント を参照してください。

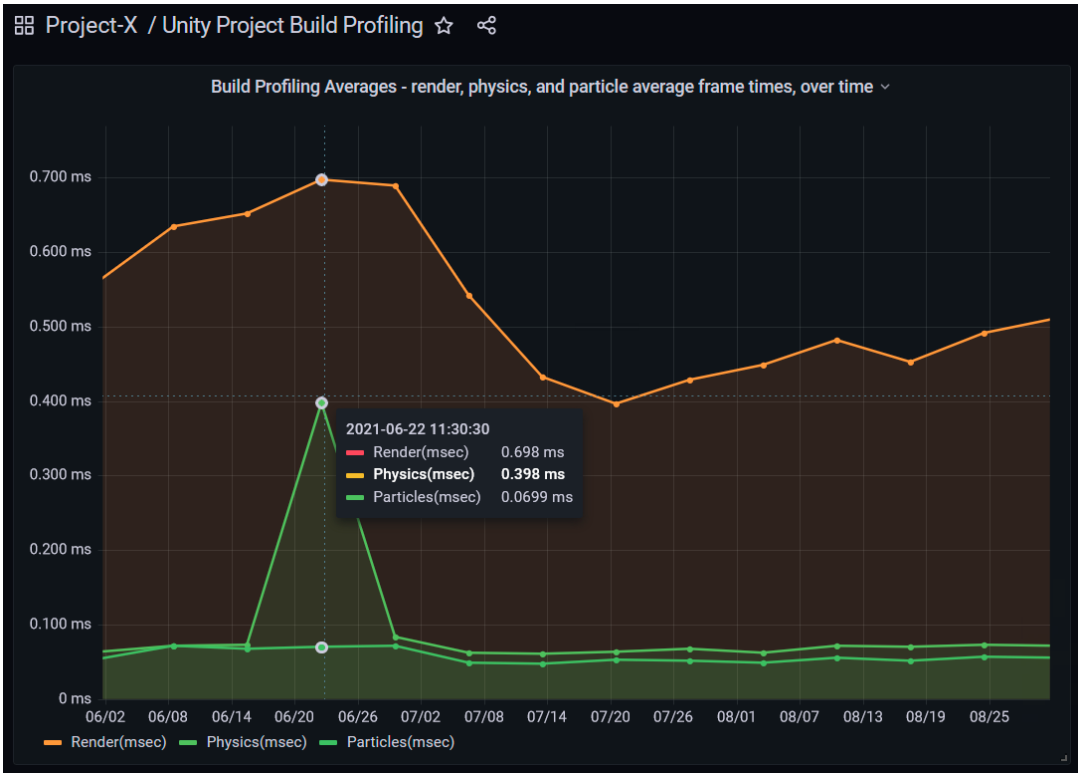
注意:もう 1 つの選択肢は、スクリプトを使用して、300 - 2000 フレームごとに 新しいログファイル (profile_<N+1>.raw など) に切り替えるか、アプリケーションのテストサイクルの重要なポイント (自動レベルプレイスルーのチェックポイント) のプロファイリングを行うことです。保存されたデータは、後からダッシュボードのグラフで問題箇所が見つかった場合に参照できます。

- プロファイリングデータは **profile1.raw** ファイルに取り込まれ、解析して関連のある指標を探すことができます。
- 次のステップでは、ProfileReader (生のプロファイルデータを解析して　CSV 形式に変換するためのツール) を使用して、生のプロファイルデータをより簡単に読み取れる CSV 形式に変換します。



ProfileReader のエディターインターフェース

- ProfileReader はコマンドラインで使用できるため、パイプラインのこの段階はそれを実行するスクリプトとなります。
    - Unity.exe -batchMode -projectPath “AngryBots2” -logFile .\Editor.log -executeMethod UTJ.ProfilerReader.CUIInterface.ProfilerToCsv -PH.inputFile “profile1.raw” -PH.timeout 2400 -PH.log
- CSV から解析されたデータを用いて、自動化されたパイプラインによって、夜間ごと、週次、スプリントごとのリリースデータが Grafana などの可視化ツールへアップロードされます。



自動化されたプロファイリングデータを可視化した Grafana ダッシュボードの画像。どうやら物理演算オブジェクト生成のバグがビルドに忍び込んでいるようです。

データが可視化され、自動的に更新されるため、グラフの急な変化を簡単に発見し、問題をより迅速に特定できます。また、パフォーマンス最適化タスクの結果や、ゲーム内のさまざまなレベルにわたってレベルデザインチームが行うメモリ最適化パスの結果も表示できます。

Unity Test Framework の Performance Testing パッケージ

Unity Performance Testing パッケージ は、パフォーマンステスト用のツールの追加によって Unity Test Framework を強化します。Unity プロファイラーのマーカーとカスタムパフォーマンス指標からサンプルをキャプチャできる API とテストデコレーターを導入します。これらはエディター内とビルド済みプレイヤーの両方で利用可能です。

このパッケージは、測定機能に加えて、ビルド設定やハードウェア詳細などの設定メタデータを収集するため、異なる環境間で結果を比較しやすくなります。

このパッケージは、Unity Test Framework と連携して動作するように設計されています。効果的に使用するには、Unity Test Framework のドキュメントに記載されているテストの作成と実行に精通している必要があります。

プロファイリングおよびデバッグツールのインデックス

まず Unity ツールでプロファイリングし、より詳細な情報が必要となった場合に、ターゲットプラットフォームで利用可能なネイティブのプロファイラーおよびデバッグツールを使用しましょう。そのようなツールのインデックスについて以下を参照してください。

ネイティブのプロファイリングツール

Android/Arm

- Android Studio: 最新の Android Studio には、以前の Android Monitor ツールに代わる新しい Android Profiler があります。これを使用して、Android デバイスのハードウェアリソースに関するリアルタイムデータを収集します。
- Arm Performance Studio:Arm ハードウェアを実行するデバイス向けの一連のツールで、ゲームの詳細なプロファイリングとデバッグに役立ちます。
- Snapdragon Profiler:Snapdragon チップセットを搭載したデバイス専用です。CPU、GPU、DSP、メモリ、電力、熱、ネットワークデータを分析して、パフォーマンスのボトルネックを見つけて修正します。

Intel

- Intel VTune:この一連のツールを使用すれば、Intel プラットフォームにおけるパフォーマンスのボトルネックを迅速に特定し、解決できます。Intel プロセッサー専用です。
- Intel GPA スイート:問題箇所を素早く特定することで、ゲームのパフォーマンス向上に役立つ、グラフィックスに特化したツールスイートです。

Xbox/PC

— PIX:PIX は、DirectX 12 を使用する Windows および Xbox ゲーム開発者向けのパフォーマンスチューニングおよびデバッグツールです。CPU と GPU のパフォーマンスを理解および分析するためのツールや、さまざまなリアルタイムパフォーマンスカウンターを観察するためのツールが含まれています。

PC/Universal

— AMD μProf:AMD uProf は、AMD ハードウェア上で動作するアプリケーションのパフォーマンスを理解し、プロファイリングを行うためのパフォーマンス解析ツールです。

— NVIDIA NSight:開発者が NVIDIA の最新ビジュアルコンピューティングハードウェアを活用し、業界をリードする最先端ソフトウェアを構築、デバッグ、プロファイリング、開発するためのツール群です。

— Samply:Samply は、Firefox プロファイラーを UI として使用するオープンソースのコマンドライン CPU プロファイラーです。 macOS、Linux、Windows で動作します。

— Superluminal:Superluminal は、C++、Rust、.NET で書かれた Windows、Xbox One、PlayStation 向けのアプリケーションのプロファイリングをサポートする、高性能かつ高頻度なプロファイラーです。ただし、有料であり、使用にはライセンスが必要です。始め方の簡単な紹介については、ディスカッション記事 をお読みください。

PlayStation

— PlayStation ハードウェア向けの CPU プロファイラーツールが利用可能です。詳細を見るには、PlayStation® 開発者登録が必要です。登録はこちらから行ってください。

iOS

— Xcode Instruments と XCode Frame Debugger:Instruments は、Xcode ツールセットに含まれる、強力で柔軟なパフォーマンス分析およびテストツールです。

WebGL

— Firefox Profiler:Firefox Profiler を使用すると、(他にも数ある中から特に) Unity WebGL ビルドのコールスタックを調べたり、フレームグラフを表示したりできます。また、プロファイリングのキャプチャを並べて確認できる比較ツールもあります。

— Chrome DevTools Performance:Unity WebGL ビルドのプロファイリングに使用できるもう 1 つのウェブブラウザーツールです。

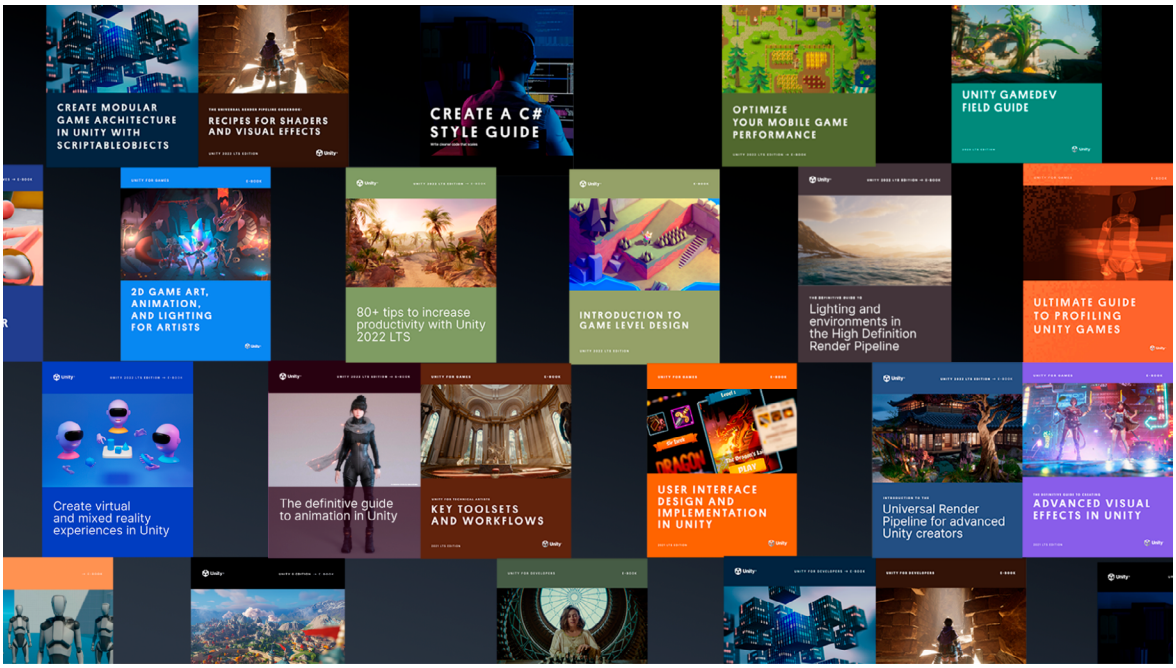
GPU デバッグおよびプロファイリングツール

Unity フレームデバッグツールは CPU から送信されるドローコールをキャプチャおよび可視化しますが、以下のツールは、GPU がこれらのコマンドを受信した際に何を行うかを示すのに役立ちます。

一部のツールは、特定のプラットフォーム専用で、より緊密なプラットフォームインテグレーションを提供します。関心のあるプラットフォームに関連するツールを参照してください。

- Arm Streamline:Arm の Performance Studio ソフトウェアスイートの一部で、CPU と GPU の低オーバーヘッドパフォーマンス測定に焦点を当てています。
- Arm Frame Advisor:Arm の Performance Studio ソフトウェアスイートの一部で、フレームベースの API プロファイリングに焦点を当てています。
- RenderDoc:フレームベースの API デバッグに重点を置いた、デスクトップおよびモバイルプラットフォーム向けの GPU デバッガーです。
- Intel GPA:Intel ベースのプラットフォーム向けのグラフィックスプロファイラーです。
- Apple フレームキャプチャデバッグツール:Apple プラットフォーム向けの GPU デバッガーです。
- Visual Studio グラフィックス診断:Windows や Xbox などの DirectX ベースのプラットフォームでは、このツールおよび/または PIX を使用してください。
- NVIDIA Nsight フレームデバッガー:NVIDIA GPU 用の OpenGL ベースのフレームデバッガーです。
- AMD Radeon 開発者ツールスイート:AMD GPU 向けの GPU プロファイラーです。
- Xcode フレームデバッガー:iOS および macOS 向けツールです。

上級開発者およびアーティスト向けのリソース



Unity のベストプラクティスハブ では、上級 Unity 開発者およびクリエイター向けのさまざまな e-Book をダウンロードできます。業界のエキスパートと Unity のエンジニアやテクニカルアーティストが作成した 30 以上のガイドから、必要なものを選べます。Unity のツールセットとシステムを活用してゲームを効率的に開発するのに役立つベストプラクティスを入手しましょう。

また、Unity ブログ、Unity Discussions、Unity Learn、そして **#unitytips** のハッシュタグからも、ヒント、ベストプラクティス、最新情報を見つけることができます。


Unity®