→ EBOOK

Unity によるコンソールと PC 向けゲームのパフォーマンス最適化

Unity®

Contents

はじめに

本ガイドは、Unity 6 で利用可能な PC とコンソールのパフォーマンスを最適化するための最善かつ最新のヒントをまとめたものです。本ガイドは、Unity 6 向けの 2 つの最適化ガイドのうちの 1 つで、もう 1 つは「*Unity におけるモバイル、XR、ウェブゲームのパフォーマンス最適化 (Unity 6 版)*」です。

パフォーマンスの最適化は、細心の注意を要する膨大なテーマです。設計要件を満たす効率的なソリューションを見つけるには、Unity のクラスとコンポーネント、アルゴリズムとデータ構造体、そしてプラットフォームのプロファイリングツールを使いこなす必要があります。

コンソールおよび PC 向けアプリケーションの最適化は、ゲーム開発サイクル全体を支える不可欠なプロセスです。オーディエンスは、ゲームが秒間 60 フレーム（fps）以上で滑らかに動作することを当然だと思っているかもしれませんが、複数のプラットフォームでパフォーマンス目標を達成することは容易ではありません。効果的に最適化するには、コードアーキテクチャとアートアセットの両方がより効率的になるように注力し、ターゲットハードウェアがその制限の中でどのように動作するかを理解する必要があります。

本ガイドは、Unity のエキスパートソフトウェアエンジニアが、業界パートナーとともに実際のシナリオでテストした、ベストプラクティスの知識とアドバイスをまとめています。

これらの手順に従って、PC およびコンソール向けゲームから最高のパフォーマンスを引き出しましょう。

プロファイリング

早期から頻繁にターゲットデバイスでプロファイルする

プロファイリングは、ランタイム時にゲームのパフォーマンスの側面を測定するプロセスであり、したがって、あらゆる最適化ワークフローの中核をなすものです。Unity とターゲットハードウェアのメーカーの両方が提供する一連のプロファイリングツールを理解することが重要です。

ターゲットプラットフォームでゲームがどのように動作するかを測定し、この情報を使用してパフォーマンスの問題の原因を突き止めることが不可欠です。変更を加えながらプロファイリングツールを観察することで、その変更が実際にパフォーマンスの問題を解決しているかどうかを測ることができます。

Unity Profiler はアプリケーションのパフォーマンス情報を提供してくれますが、使わなければ意味がありません。

プロファイリングは、ランタイム時にゲームのパフォーマンスの側面を測定し、パフォーマンス問題の原因を突き止めるプロセスです。変更を加えながらプロファイリングツールを観察することで、その変更が実際にパフォーマンスの問題を解決しているかどうかを測ることができます。

プロジェクトのプロファイリングは、リリース直前だけなく、開発サイクルの初期から全体を通じて行いましょう。不具合や急激な悪化が現れたらすぐに調査し、プロジェクトにおける大きな変更の前後でパフォーマンスのベンチマークを取るようにしてください。プロジェクトの「パフォーマンスシグネチャ」を開発すれば、新しい問題をより簡単に発見できるようになります。

エディターでのプロファイリングでもゲーム内の異なるシステムの相対的なパフォーマンスを知ることができますが、各デバイスでプロファイルすることで、より正確な洞察を得られます。可能な限り、ターゲットデバイス上で開発ビルドをプロファイルしてください。サポート予定の最高スペックと最低スペックの両方の

デバイスでプロファイルし、忘れずに最適化を行いましょう。

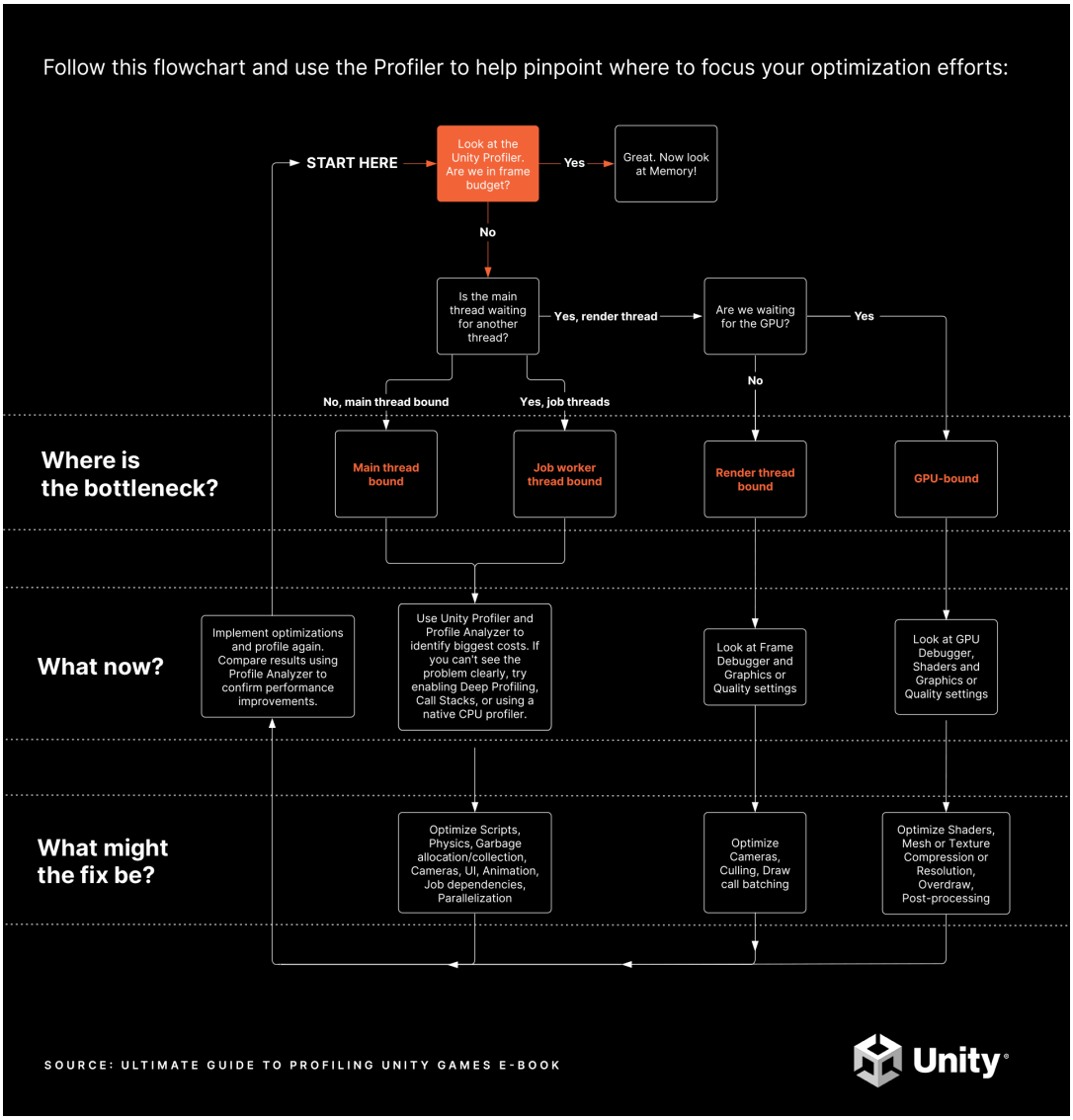
Unity は、Unity Profiler や Memory Profiler、Profile Analyzer といった、ボトルネックを特定するための一連のプロファイリングツールを提供しています。

適切な箇所の最適化に集中する

ゲームのパフォーマンスを低下させている原因を憶測したり、決めつけたりしてはいけません。Unity Profiler とプラットフォーム固有のツールを使用して、ラグの正確な原因を特定してください。プロファイリングツールとは、端的にいうと Unity プロジェクトの内部で何が起きているのかを理解する手助けをしてくれるものです。しかし、重大なパフォーマンス問題が表面化するのを待つのではなく、早めに分析ツールを活用しましょう。

もちろん、ここで説明されているすべての最適化があなたのアプリケーションに当てはまるとは限りません。あるプロジェクトでうまくいったことが、あなたのプロジェクトではうまくいかないかもしれません。真のボトルネックを特定し、自身の作業に役立つ部分に注力しましょう。

プロファイリングのワークフローを計画する方法については、「*Unity ゲームのプロファイリングに関する決定版ガイド*」を参照してください。



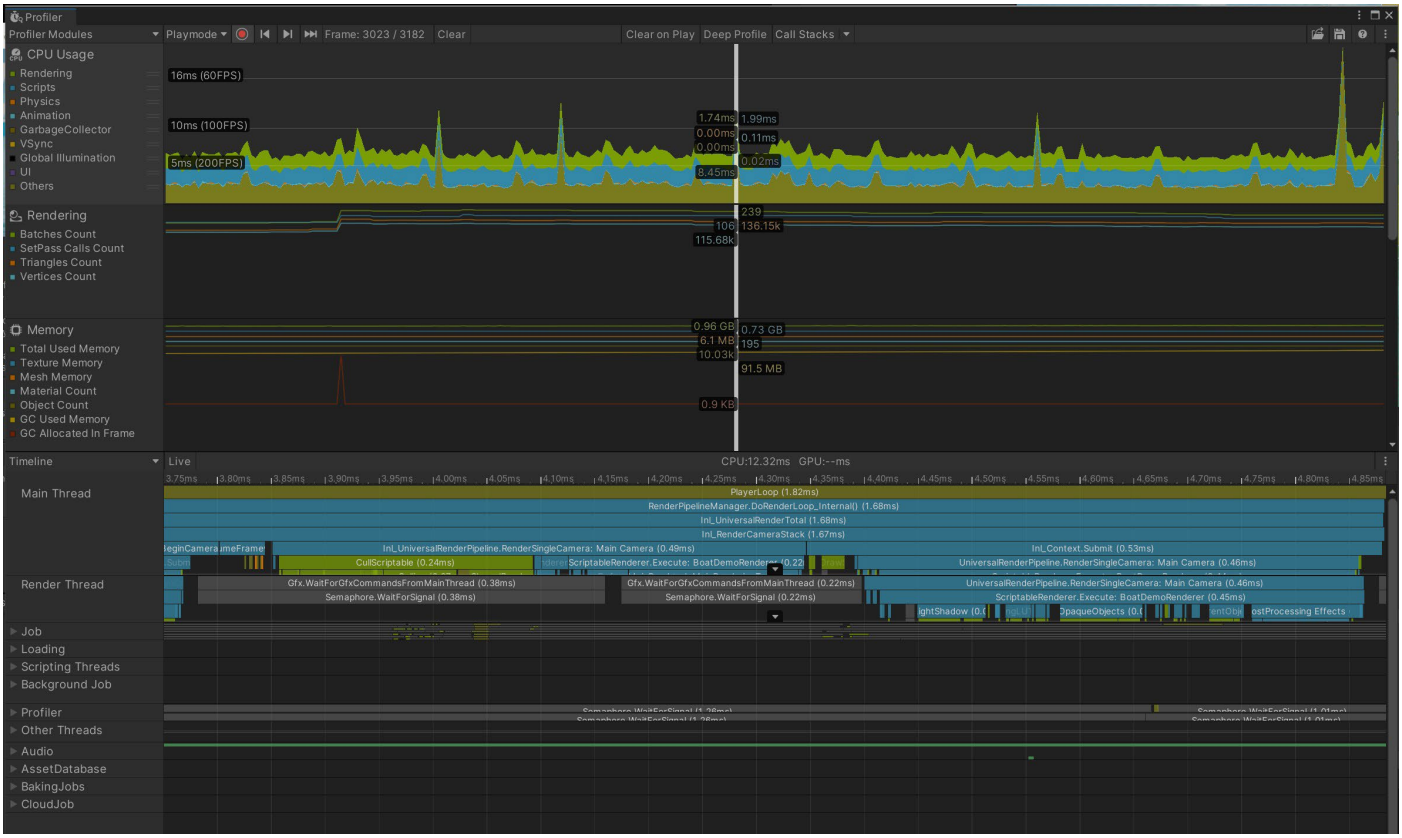
上記のワークフローに従って、効率的に Unity プロジェクトプロファイルする。

Unity Profiler の仕組みを理解する

Unity Profiler 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。

このプロファイラーは、インスツルメンテーションをベースとしています。ゲームやエンジンの自動でマークアップされたコード（MonoBehaviour の Start または Update メソッド、特定の API 呼び出しなど）や、ProfilerMarker の API を使用して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ルします。

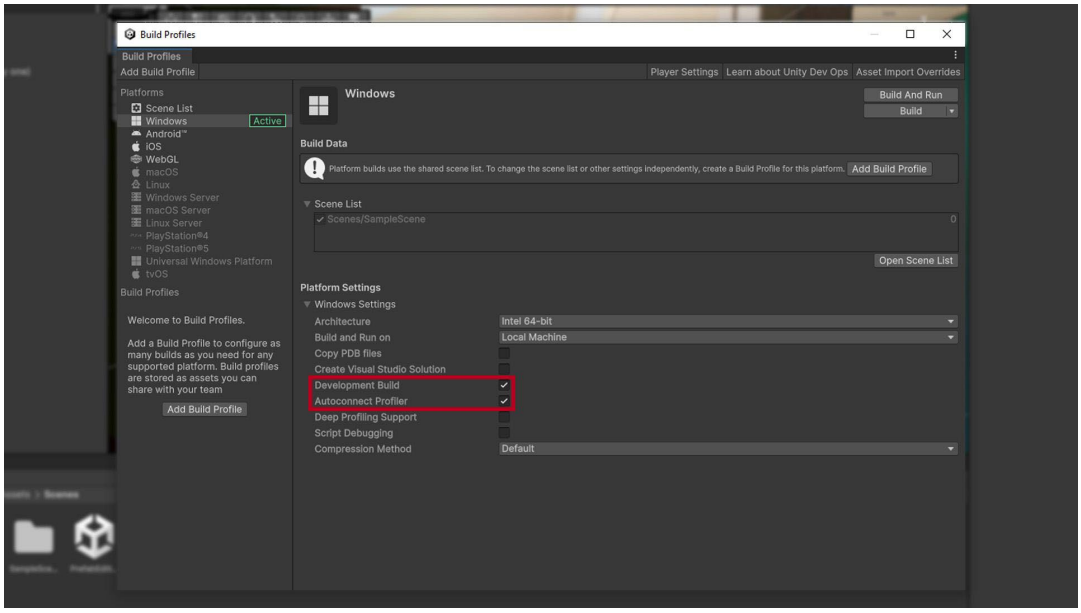
CPU とメモリーのトラックをデフォルトで有効にすることから始めましょう。ゲームに応じて、Renderer、Audio、Physics など、補足のプロファイラーモジュールを観察することも可能です（例えば、物理演算を多用するゲームプレイや音楽をベースとするゲームプレイに必要なプロファイリングを行います）。ただし、必要なものだけを有効にして、パフォーマンスに影響を与えたり、結果を歪ませたりしないようにしてください。



Unity Profiler を使用してパフォーマンスとリソースの割り当てをテストする。

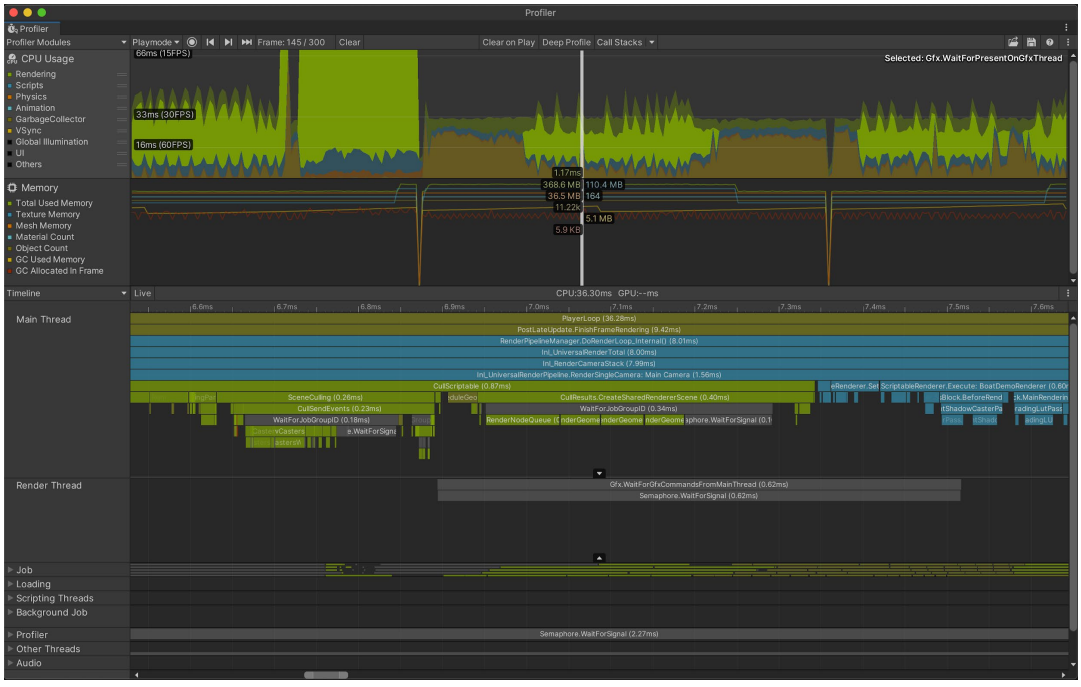
選択したプラットフォーム内の実際のデバイスからプロファイリングデータを取得するには、「Build And Run」をクリックする前に、「**Development Build**」をチェックします。そして、アプリケーションの実行を開始したら、プロファイラーを手動でアプリケーションに接続します。

任意で、Build Profiles ウィンドウの「**Platform Settings**」の下にある「**Autoconnect Profiler**」をチェックすることもできます。これは、特にアプリケーションの最初の数フレームをキャプチャしたい場合に便利です。ただし、これにより起動時間が 5 ～ 10 秒長くなる可能性があるので、必要な場合のみ使用してください。



プラットフォーム設定は、プロファイリング前に調整する。

プロファイルするターゲットプラットフォームを選択します。Record ボタンは、アプリケーションの再生を数秒間（デフォルトでは 300 フレーム）記録します。記録時間を伸ばしたい場合は、**「Unity」> 「Preferences」> 「Analysis」> 「Profiler」> 「Frame Count」**から、最高で 2000 フレームまで設定できます。CPU とメモリーのリソース消費量は増えますが、特定のシナリオによっては有用です。

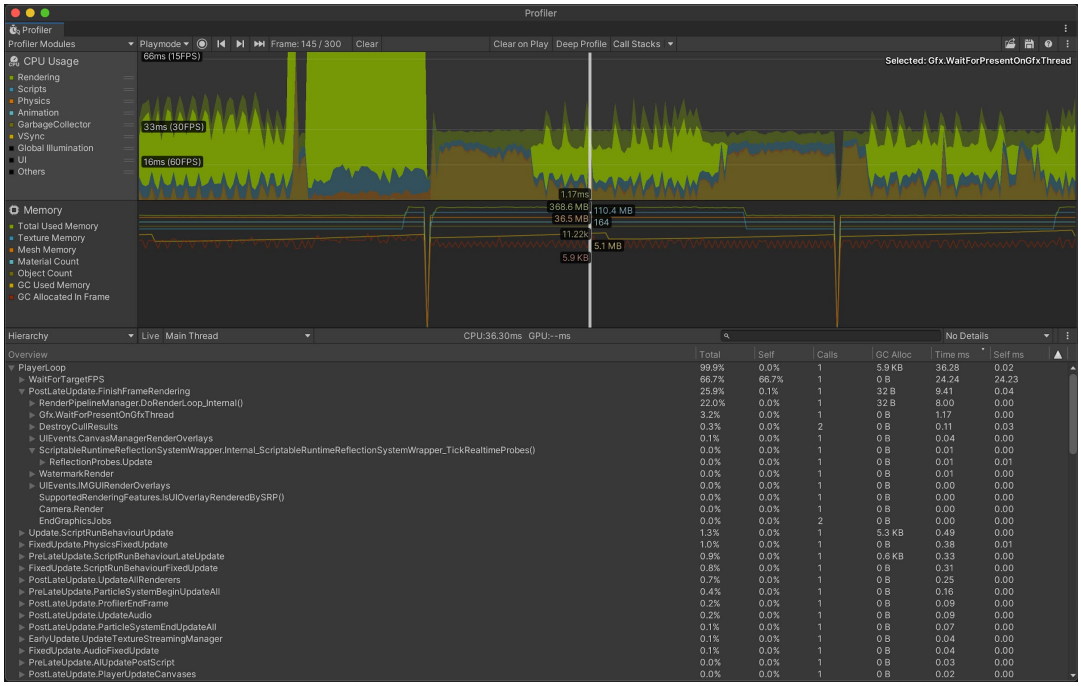


Timeline ビューを使用して、CPU 依存か GPU 依存かを確認する。

Deep Profiling 設定を使用すると、Unity はスクリプトコード内のすべての関数呼び出しの最初と最後をプロファイルできるため、アプリケーションのどの部分が実行され、どの部分で遅延が発生する可能性があるかを正確に知ることができます。しかし、Deep Profiling はすべてのメソッド呼び出しにオーバーヘッドを追加するため、パフォーマンス分析を歪める可能性があります。

特定のフレームを分析するには、ウィンドウ内をクリックします。次に、Timeline または Hierarchy ビューを使用して以下を行います。

- **Timeline** は、特定のフレームのタイミングにおける視覚的な内訳を示します。これにより、アクティビティが互いに、また異なるスレッド間でどのように関連しているかを視覚化できます。このオプションを使用して、CPU 依存か GPU 依存かを確認してください。
- **Hierarchy** は ProfileMarkers の階層をグループ化して表示します。これにより、ミリ秒単位の時間コスト（Time ms と Self ms）に基づいてサンプルを並べ替えることができます。また、関数呼び出し数やフレーム上のマネージヒープメモリ（GC Alloc）をカウントすることもできます。



Hierarchy ビューでは、時間コストごとに ProfileMarkers を並べ替えることができる。

プロファイリングにまだ慣れていない場合は、以下の短いチュートリアル動画をご覧ください。

- Unity Profiler ウォークスルー
- Profile Analyzer ウォークスルー
- Memory Profiler ウォークスルー

Unity Profiler の完全な概要については、こちらのプロファイリングに関する eBook をダウンロードしてください。

プロジェクトで何かを最適化する前に、Profiler .data ファイルを保存してください。変更を実行し、保存した .data を変更前と変更後で比較してください。パフォーマンスを改善するために、プロファイリング、最適化、比較のサイクルに依拠しましょう。

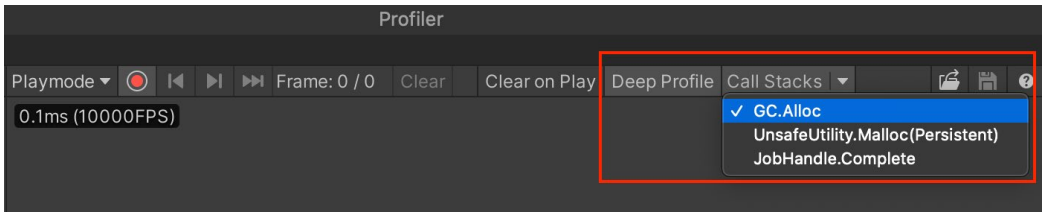
Deep Profiling

「Build Setting」から「Deep Profiling Support」を有効にすることもできます。ビルドされた Player が起動すると、Deep Profiler は、ProfilerMarkers で明示的にラップされたコードのタイミングだけでなく、コードのすべての部分をプロファイルします。

Deep Profiling を有効にすると、Unity はスクリプトコード内のすべての関数呼び出しの最初と最後をプロファイルできます。これは、アプリケーションのどの部分が速度低下を引き起こしているかを正確に特定するのに役立ちます。

しかし、Deep Profiling はリソースを大幅に消費し、大量のメモリを使用します。各 ProfilerMarker は、わずかながらオーバーヘッド（プラットフォームにもよりますが、10ns 程度）を追加するため、測定ポイントが増えるごとにアプリケーションの動作が遅くなります。また、関数呼び出しが多い場合、Deep Profiling によってそのオーバーヘッドも増幅されることに注意してください。

GC.Alloc や JobHandle.Complete などのマーカーが付いたサンプルの詳細を見たい場合は、Profiler ウィンドウのツールバーに移動し、「**Call Stacks**」設定を有効にします。これは、サンプルの完全なコールスタックを提供し、Deep Profiling のオーバーヘッドを発生させることなく、必要な情報を得ることができます。

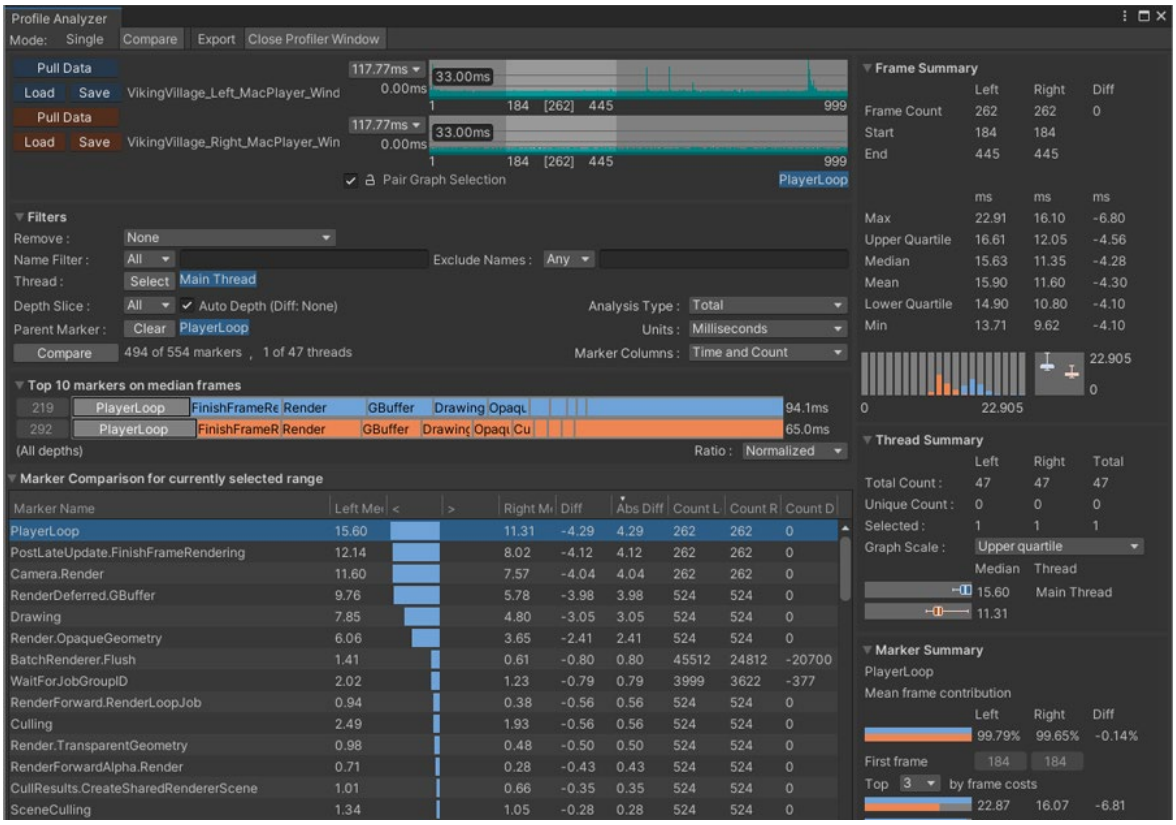


可能な限り、Deep Profiling ではなく Call Stacks を使用する。

一般的に、Deep Profiling が使用されると、アプリケーションの動作が著しく遅くなるため、Deep Profiling は必要な場合にのみ使用してください。

Profile Analyzer を使用する

Profile Analyzer は、Profiler データの複数のフレームを集約し、対象のフレームを見つけることができます。プロジェクトに変更を加えた後、Profiler がどうなるか見たいと思うことがあるでしょう。**Compare** ビューでは、2 つのデータセットをロードして区別することができるので、変更をテストしてからその結果を改善できます。Profile Analyzer は Unity の Package Manager からアクセス可能です。



フレームとマーカーデータの詳細は、既存の Profiler を補完する Profile Analyzer から確認できる。

この機能の詳細については、こちらの Profile Analyzer チュートリアルをご覧ください。

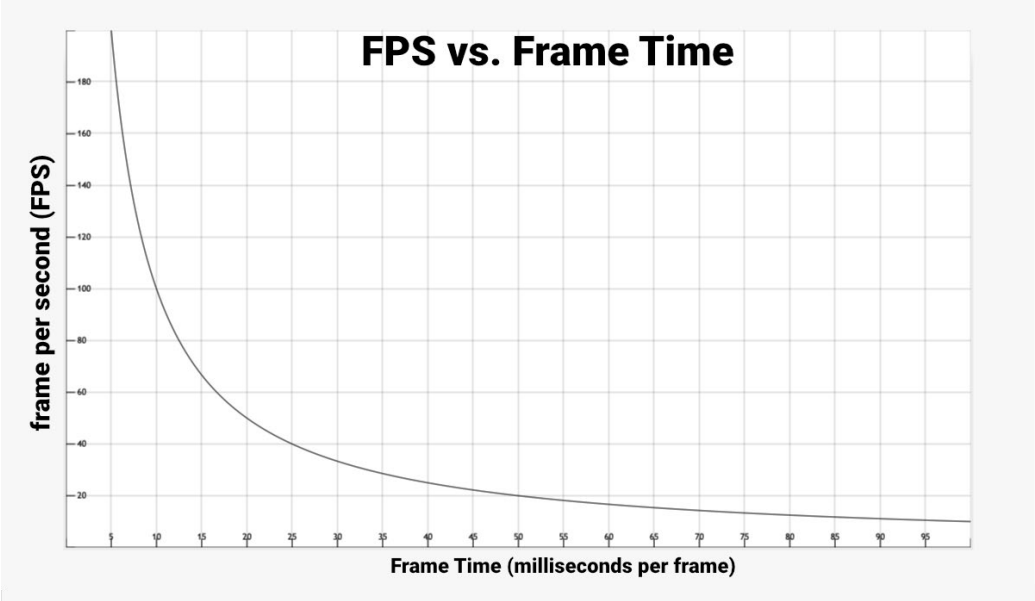
フレームあたりの明確なタイムバジェットで作業する

各フレームには、目標とする秒間フレーム数（fps）に基づいたタイムバジェットが設定されます。アプリケーションを 30fps で実行するには、フレームバジェットが 1 フレームあたり 33.33ms（1000ms ÷ 30fps）を超えることはできません。同様に、60fps の場合は 1 フレームあたり 16.66ms となります。

1 秒あたりのフレーム数：まやかしの指標

ゲーマーがパフォーマンスを測る一般的な方法は、フレームレート（1 秒あたりのフレーム数）です。しかし、これはアプリケーションのパフォーマンスを測る上では、まやかしの指標となりえます。

代わりに、ミリ秒単位のフレーム時間を使用することをお勧めします。その理由を理解するには、以下の **fps 対フレーム時間**のグラフをご覧ください。



fps 対フレーム時間

これらの数字を考えてみましょう。

1000 ms（1 秒）÷ 900fps = 1 フレームあたり 1.111ms
1000 ms（1 秒）÷ 450fps = 1 フレームあたり 2.222ms
1000 ms（1 秒）÷ 60fps = 1 フレームあたり 16.666ms
1000 ms（1 秒）÷ 56.25fps = 1 フレームあたり 17.777ms

アプリケーションが 900fps 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 ミリ秒ということになります。450fps の場合は、1 フレームあたり 2.222 ミリ秒です。**フレームレートが半分になったように見えますが、これは 1 フレームあたりの差がわずか 1.111 ミリ秒であることを示しています。**

60fps と 56.25fps の違いを見ると、それぞれ 1 フレームあたり 16.666 ミリ秒と 17.777 ミリ秒になります。これはまた、1 フレームあたり 1.111 ミリ秒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タイムを使う理由です。

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、フレーム予算内に収まるようにしてください。

詳しくは、Robert Dunlop 氏による元記事「FPS Versus Frame Time」をご覧ください。

GPU 依存か CPU 依存かを確認する

中央演算処理装置 (CPU) は描画すべきものを決定し、グラフィックス処理装置 (GPU) は描画を担当します。

CPU が GPU のためにフレームを処理し準備するのに時間がかかりすぎると、全体のフレームレートは CPU パフォーマンスによって制限されます。つまり、CPU 依存となります。

同様に、CPU がフレームを準備した後、GPU がレンダリングするのに時間がかかりすぎる場合は、GPU 依存となります。

Unity Profiler を使えば、CPU が割り当てられたフレーム予算より長くかかっているのか、または GPU が原因なのかを知ることができます。これは、次のように Gfx をプレフィックスに持つマーカーを出力することによって行われます。

- **Gfx.WaitForCommands** マーカーが表示された場合、レンダースレッドの準備はできているが、メインスレッドでボトルネックが発生している可能性があることを意味します。
- 頻繁に **Gfx.WaitForPresentOnGfxThread** に遭遇する場合は、メインスレッドの準備ができているが、レンダースレッドは待機状態であることを意味します。これは、アプリケーションが GPU 依存であることを示している可能性があります。CPU Profiler モジュールの Timeline ビューでレンダースレッドのアクティビティを確認してください。

  レンダースレッドが **Camera.Render** に多くの時間を費やしている場合、アプリケーションは CPU 依存の状態で、ドローコールやテクスチャを GPU に送るのに多くの時間を費やしている可能性があります。

  レンダースレッドが **Gfx.PresentFrame** に多くの時間を費やしている場合、アプリケーションが GPU 依存の状態にあるか、GPU で VSync を待機している可能性があります。

マーカーの一覧については、Common Profiler マーカーのドキュメントを参照してください。また、フレームパイプラインに関する詳細は、ブログ記事「より安定したゲームプレイを実現する Unity 2020.2 の Time.deltaTime 修正」をご覧ください。

ネイティブのプロファイリングおよびデバッグツールを使用する

Unity ツールでプロファイルした後、より詳細な情報が必要となった場合は、ターゲットプラットフォーム向けに利用可能なネイティブのプロファイリングおよびデバッグツールを使用しましょう。

ネイティブのプロファイリングツール

インテル

- インテル VTune：このインテルプロセッサー専用のツールスイートを使用すると、インテルプラットフォームでのパフォーマンスのボトルネックをすばやく検出して修正できます。
- インテル GPA スイート：グラフィックスに特化したこのツールスイートは、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ちます。

Xbox コンソールおよび Windows PC

- PIX：PIX は、DirectX 12 を使用する Windows および Xbox コンソール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。Windows 開発者の場合は、こちらから開始してください。Xbox 向けに PIX を活用するための詳細が知りたい場合は、Xbox 開発者登録が必要です。こちらから登録してください。

PC/Universal

- AMD μ Prof：AMD uProf は、AMD ハードウェア上で動作するアプリケーションパフォーマンスを理解し、プロファイルするためのパフォーマンス解析ツールです。
- NVIDIA NSight 開発者ツール：このツールにより、開発者は NVIDIA の最新のアクセラレーテッドコンピューティングハードウェアを使用して、クラスをリードする最先端のソフトウェアをビルド、デバッグ、プロファイル、開発できます。
- Superluminal：Superluminal は、C++、Rust、.NET で書かれた Windows、Xbox コンソール、PlayStation® で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、高性能かつ高頻度の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。

PlayStation®

- PlayStation の開発環境では、CPU プロファイラツールが利用できます。詳細を知るには、PlayStation 開発者登録が必要です。こちらから登録してください。

ウェブ

- Firefox Profiler：Firefox Profiler を使用すると、（他にも数ある中から特に）Unity Web 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：このウェブブラウザツールは、Unity Web ビルドのプロファイリングに使用できます。

GPU デバッグおよびプロファイリングツール

Unity フレームデバッガーは CPU から送信されるドローコールをキャプチャして描写しますが、以下のツールは、GPU がこれらのコマンドを受信したときに何をする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。さまざまな特定のプラットフォーム専用で利用可能なツールは以下の通りです。

- RenderDoc：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
- インテル GPA：インテル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
- 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
- Visual Studio グラフィックス診断：PIX の他に、DirectX ベースのプラットフォーム（Windows や Xbox など）向けに選択すべきツール
- NVIDIA Nsight Graphics：NVIDIA GPU 向けのグラフィックスプロファイラーおよびデバッガー
- 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
- Xcode フレームデバッガー：iOS および macOS 向け

Project Auditor

Project Auditor は、プロジェクトのスクリプトや設定を静的に解析する実験的なツールです。マネージメモリ割り当て、非効率的なプロジェクト構成、パフォーマンスのボトルネックの可能性などの原因を突き止める素晴らしい方法を提供します。

Project Auditor は、エディターで使用するための無料の非公式パッケージです。詳細については、Project Auditor ドキュメントを参照してください。

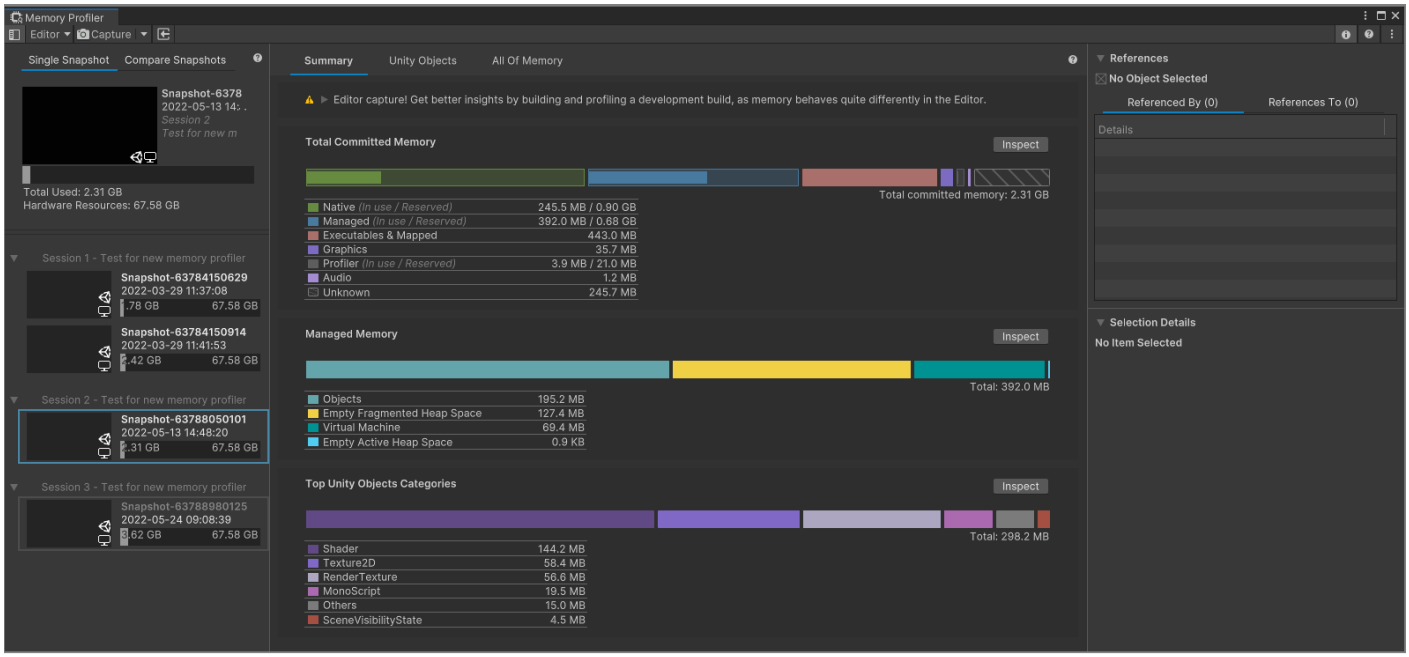
メモリ

Unity は、ユーザーが作成したコードやスクリプトの自動メモリ管理を採用しています。値型ローカル変数のような小さなデータは、スタックに割り当てられます。より大きなデータと長期ストレージは、マネージヒープまたはネイティブヒープに割り当てられます。

ガベージコレクターは、定期的に未使用のマネージヒープメモリを特定し、割り当てを解除します。アセットガベージコレクションは、オンデマンドまたは新しいシーンをロードした際に実行され、ネイティブオブジェクトとリソースの割り当てを解除します。これは自動的に実行されますが、ヒープ内のすべてのオブジェクトを調べる過程で、ゲームでスタッターが発生したり、動作が遅くなったりすることがあります。

メモリ使用量を最適化するということは、マネージヒープメモリをいつ割り当て、いつ割り当て解除するか、そしてガベージコレクションの影響をどのように最小化するかを意識することを意味します。

詳細については、「マネージヒープを理解する」をご覧ください。



Memory Profiler でスナップショットをキャプチャ、検査、比較する。

Memory Profiler を使用する

Memory Profiler パッケージは、断片化やメモリリークなどの問題を特定するために、マネージヒープメモリのスナップショットを取ります。簡単な紹介として、こちらの Memory Profiler チュートリアルをご覧ください。

Unity Objects タブを使用して、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見つけます。**All of Memory** タブは Unity が追跡しているスナップショット内のすべてのメモリの内訳を表示します。

Unity で Memory Profiler を活用してメモリ使用量を改善する方法を学びましょう。

ガベージコレクション（GC）の影響を減らす

Unity は Boehm-Demers-Weiser ガベージコレクターを使用しており、これによりプログラムコードの実行を停止し、作業が完了してから通常の実行が再開されます。

GC スパイクを引き起こす可能性のある、特定の不必要なヒープ割り当てに注意しましょう。

— **Strings：**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。大規模に使用する場合は、不必要な文字列の作成や操作は抑えてください。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 クラスを使用します。

- **Unity 関数呼び出し**：Unity API 関数の中には、特に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ます。また、ガベージの発生を避ける特定の関数も活用してください。例えば、手動で文字列と **GameObject.tag** を比較するのではなく、**GameObject.CompareTag** を使用します（新しい文字列を返すとガベージが発生するため）。
- **ボックス化**：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**：いずれの方法でも、裏でボックス化が起こり、ガベージを発生させます。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。新しい配列を作る代わりに、for ループを書き、リストを使用します。
- **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、リストを MonoBehaviour のメンバーにして、Start で初期化します。シンプルに、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。

詳細については、「ガベージコレクションのベストプラクティス」のマニュアルページをご覧ください。

可能な限りガーベジコレクションのタイミングを調整する

ガベージコレクションのフリーズがゲームの特定のポイントに影響しないことが確実な場合は、**System.GC.Collect** でガベージコレクションをトリガーすることができます。

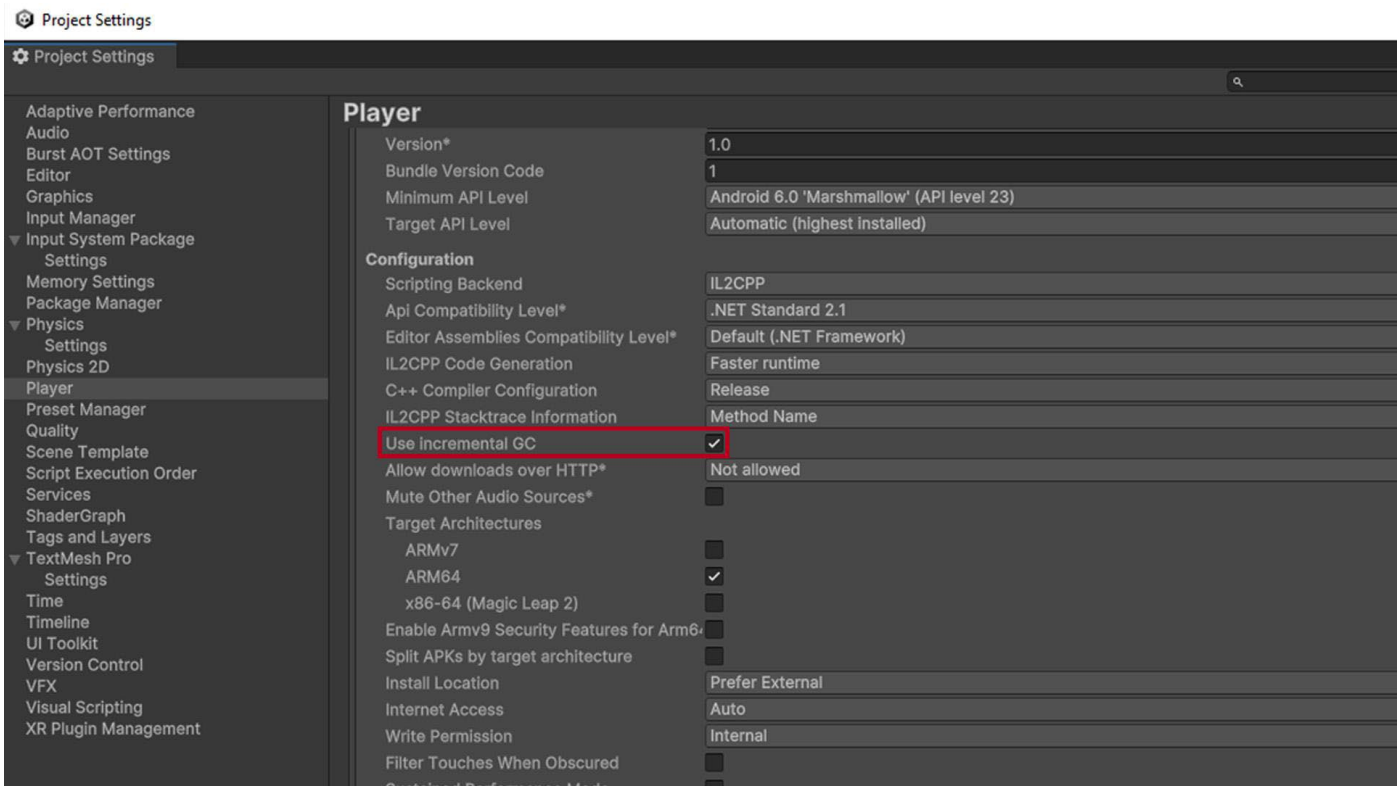
自動メモリ管理の活用例については、「自動メモリ管理を理解する」を参照してください。[1]

1　注意すべき点は、GC を使用すると、一部の C# 呼び出しに読み込みおよび書き込みのバリアが追加される可能性があり、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 1ms のわずかなオーバーヘッドが発生することがあるということです。最適なパフォーマンスを得るためには、メインのゲームプレイループでは GC の割り当てを行わず、GC.Collect をユーザーが気づかないような場所に隠すのが理想的です。

インクリメンタルガベージコレクターを使って GC の作業負荷を分割する

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 回の長い割り込みを発生させるのではなく、多くのフレームにわたって作業負荷を分散させる、複数のはるかに短い割り込みを使用します。ガベージコレクションがパフォーマンスに影響を及ぼしている場合、このオプションを有効にして GC スパイクの問題を軽減できるかどうか試してみてください。Profile Analyzer を使用して、あなたのアプリケーションにとってのメリットを確認してください。

注：インクリメンタル GC は一時的にガベージコレクションの問題を軽減するのに役立ちますが、長期的な最善策は、ガベージコレクションのトリガーとなる頻繁な割り当てを特定し、それを止めることです。



GC スパイクを減らすためにインクリメンタルガベージコレクターを使用する。

アセット

アセットパイプラインはアプリケーションのパフォーマンスに劇的な影響を与えます。経験豊富なテクニカルアーティストが、アセットフォーマット、仕様、インポート設定を定義し、実施することで、スムーズなプロセスを実現します。

デフォルトの設定に依存してはいけません。プラットフォーム固有のオーバーライドタブを使用して、テクスチャやメッシュジオメトリなどのアセットを最適化してください。設定が正しくない場合、ビルドサイズが大きくなり、ビルド時間が長くなり、GPU パフォーマンスが低下し、メモリ使用量が低下する可能性があります。Presets 機能を使用して、特定のプロジェクトを強化するベースライン設定をカスタマイズすることを検討してください。

こちらのアートアセットをインポートするためのベストプラクティスガイドをご覧ください。モバイルに特化したガイド（多くの一般的なヒントもあります）については、Unity Learn コース「モバイルアプリケーションのための 3D アートの最適化」をご覧ください。また、Presets を活用するための更なる詳細は、GDC 2023 セッション「ゲーム制作の各段階におけるテクニカルなヒント」をご視聴ください。

テクスチャを圧縮する

同じモデルとテクスチャを使った 2 つの例を考えてみましょう。上部の設定は、下部の設定に比べて 5 倍以上のメモリを消費しますが、ビジュアル品質において、大きなメリットはありません。

非圧縮テクスチャはより多くのメモリを要する。

テクスチャ圧縮は、正しく適用すればパフォーマンスに大きなメリットをもたらします。

その結果、ロード時間が短縮され、メモリフットプリントが小さくなり、レンダリングパフォーマンスが劇的に向上します。圧縮テクスチャは、非圧縮 32 ビット RGBA テクスチャに必要なメモリ帯域幅のほんの一部しか使用しません。

ターゲットプラットフォームのテクスチャ圧縮形式の推奨リストを参照してください。

- **iOS/Android：ASTC**
- **PC/XBox One/PlayStation®4：BC7**（高画質）または **DXT1**（低画質 / 通常画質）

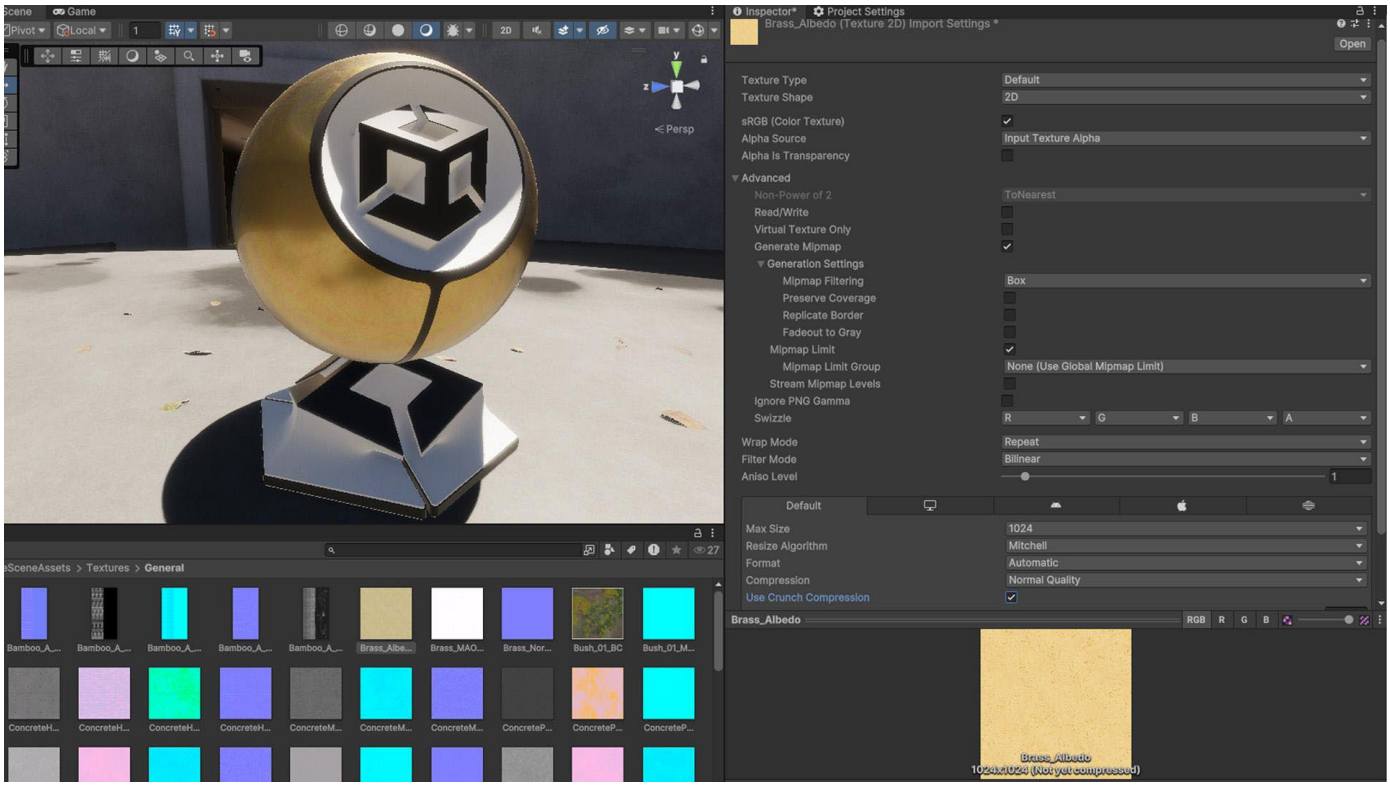
テキスチャインポート設定

テキスチャには多くのリソースを使用する可能性があります。そのため、ここでのインポート設定はとても重要です。以下のガイドラインに従いましょう。

- **最大サイズを抑える**：視覚的に許容できる結果が得られる最小限の設定を使用します。これは非破壊的で、テクスチャのメモリを簡単に削減できます。
- **2 のべき乗を使用する**：Unity では、テクスチャ圧縮形式に POT テクスチャ寸法が必要です。
- **Read/Write Enabled オプションをオフにする**：このオプションがオンの場合、CPU と GPU の両方でアドレス指定可能なメモリにコピーが作成され、テクスチャのメモリフットプリントが 2 倍になります。ほとんどの場合ではオフにしておきましょう（ランタイム時にテクスチャを生成し、それを上

書きする必要がある場合にのみオンにしてください）。このオプションは、**Texture2D.Apply** で、true に設定された **makeNoLongerReadable** を渡すことで有効にすることもできます。

— **不必要なミップマップを無効にする**：ミップマップは、カメラから異なる距離でレンダリングする必要があるディテールの量を減らすことでパフォーマンスを最適化するために使用されますが、常に必要というわけではありません。2D スプライトや UI グラフィックスなど、画面上で一定のサイズを保つテクスチャの場合は、ミップマップを無効にしても問題ありません（カメラからの距離が変化する 3D モデルの場合は有効にしてください）。



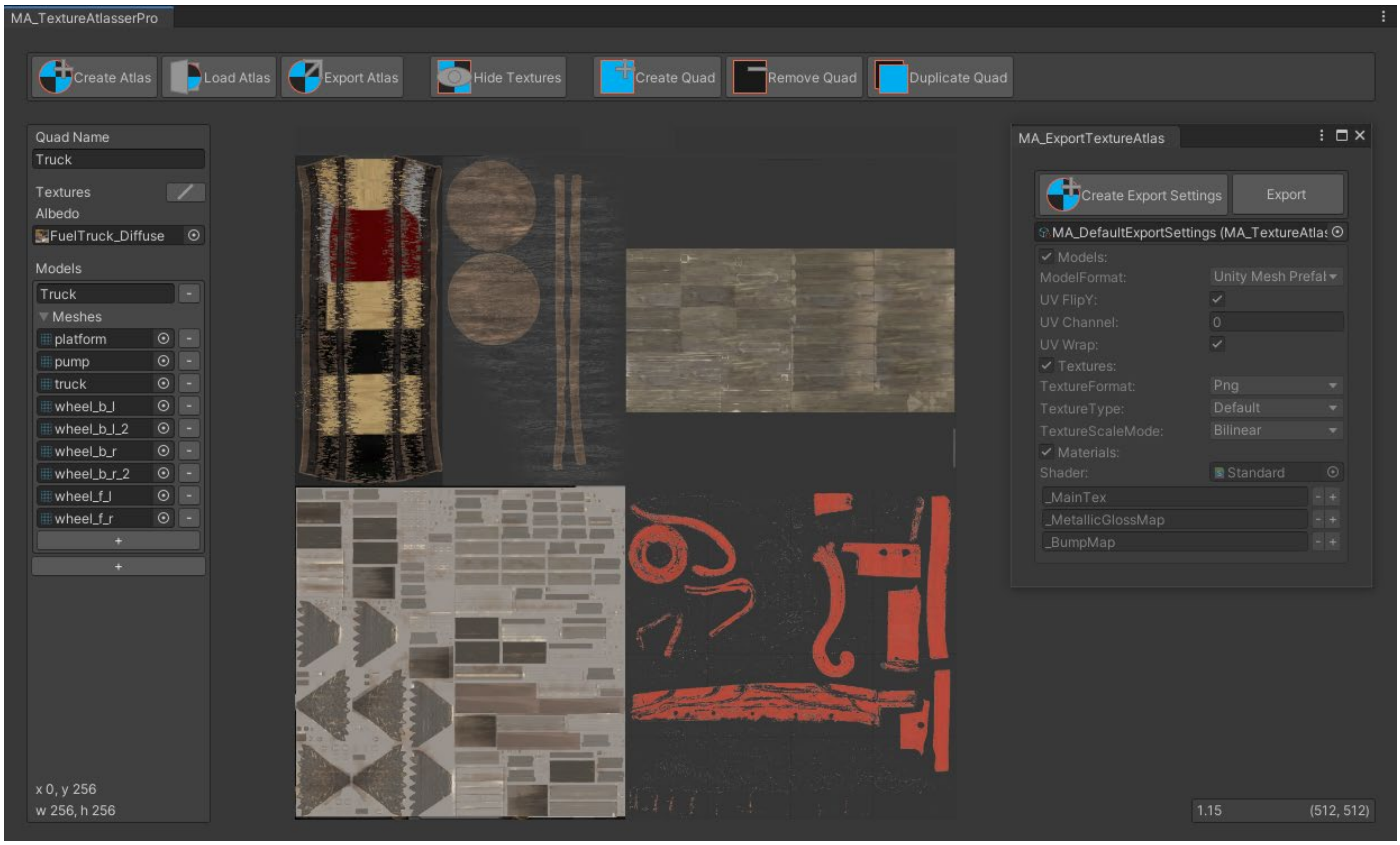
適切なテクスチャのインポート設定は、ビルドサイズを最適化するのに役立つ。

テクスチャをアトラス化する

アトラス化とは、複数の小さなテクスチャをグループ化し、単一の均一サイズの大きなテクスチャにするプロセスです。これにより、コンテンツの描画に必要な GPU の労力を減らし（ドローコールの回数を減らし）、メモリ使用量を削減できます。

2D プロジェクトの場合は、個別のスプライトやテクスチャをレンダリングするのではなく Sprite Atlas（**「Asset」>「Create」>「2D」>「Sprite Atlas」**）を使用することができます。

3D プロジェクトの場合は、お好みのデジタルコンテンツ生成（DCC）パッケージを使用することができます。MA_TextureAtlasser や TexturePacker といった、いくつかのサードパーティ製のツールでもテクスチャアトラスをビルドできます。



テクスチャアトラスを使用してドローコールを削減する。

高解像度マップを必要としないあらゆる 3D ジオメトリに対して、テクスチャを組み合わせて UV をリマップできます。ビジュアルエディターを使えば、テクスチャアトラスやスプライトシートのサイズや位置を設定し、優先順位をつけることができます。

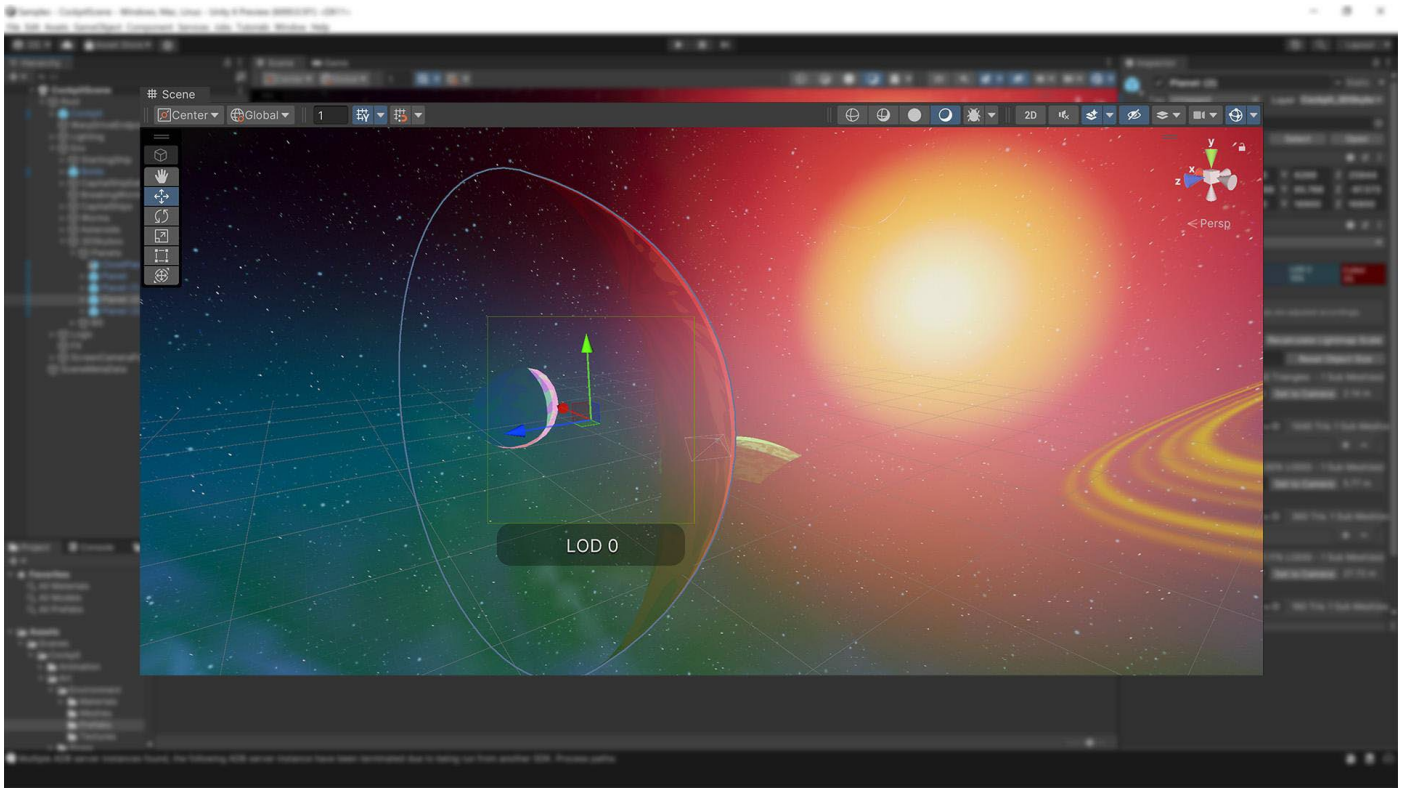
テクスチャパッカーは、個々のマップを 1 つの大きなテクスチャに統合します。そうすれば、パックされたテクスチャにアクセスする際に Unity が発行するドローコールは 1 回となり、パフォーマンスのオーバーヘッドが少なくなります。

ポリゴン数をチェックする

高解像度のモデルは、より多くのメモリを使用し、GPU 時間が長くなる可能性があります。背景のジオメトリには、本当に 100 万 もポリゴン数が必要ですか。

シーン内のゲームオブジェクトは、ジオメトリの複雑さが最小限になるように抑えてください。そうしなければ、Unity は大量の頂点データをグラフィックスカードにプッシュしなければなりません。

希望する DCC パッケージのモデルの削減を検討してください。カメラの視点から見えないポリゴンは削除しましょう。例えば、食器棚の背面が壁に面していて見えない場合、食器棚のモデルにその面は必要ありません。



見えない面を取り除いてモデルを最適化する。

ボトルネックは通常、最新の GPU ではポリゴン数ではなく、ポリゴン密度であることに注意してください。遠くのオブジェクトのポリゴン数を減らすために、すべてのアセットでアートパスを実行することをお勧めします。マイクロ三角形は GPU パフォーマンスの低下の重大な原因となります。

ターゲットプラットフォームによっては、ローポリジオメトリを補うために、高解像度のテクスチャを使用してディテールを追加することを検討してください。メッシュの密度を上げる代わりに、テクスチャと法線マップを使用すると良いでしょう。

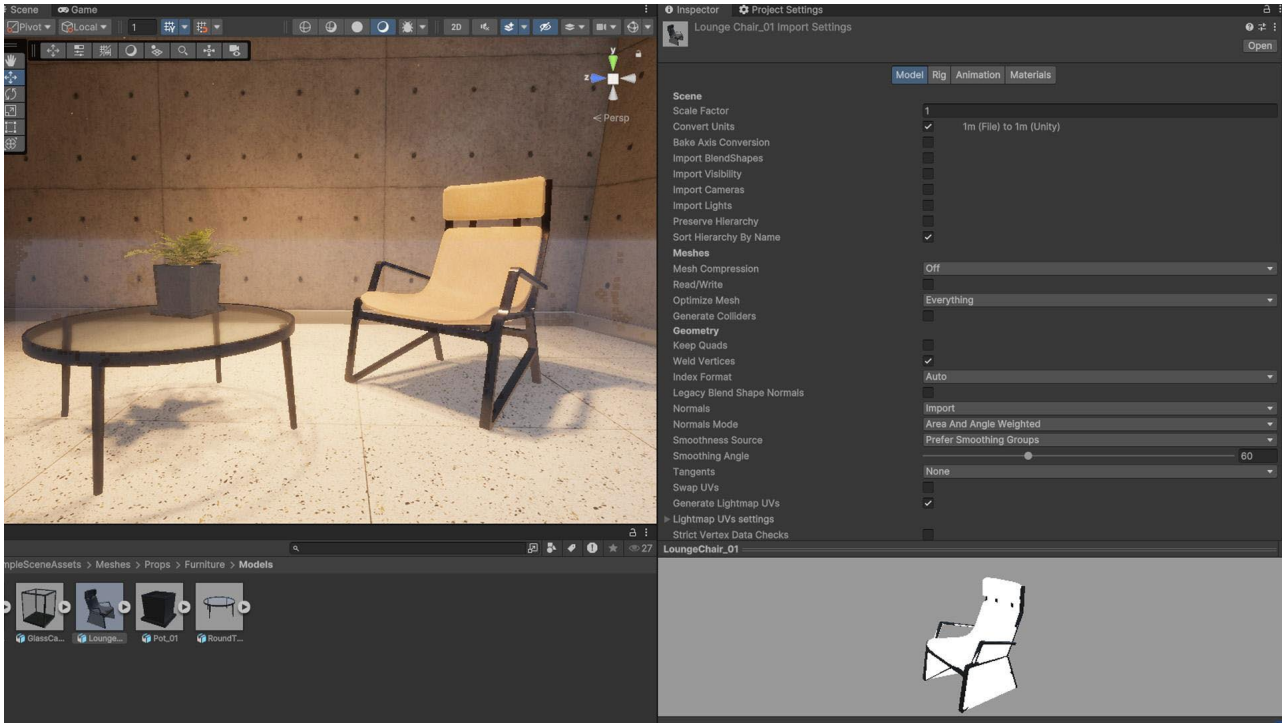
テクスチャに可能な限りディテールをベイクして、ピクセルを簡素化することもできます。例えば、フラグメントシェーダーでハイライトを計算しなくて済むように、テクスチャにスペキュラーハイライトを入れます。

これらのテクニックはパフォーマンスに影響を与える可能性があり、ターゲットプラットフォームには適していない場合もあるため、注意して行い、定期的にプロファイルしてください。

メッシュのインポート設定

テクスチャと同様、メッシュも注意深くインポートしなければ、メモリを過剰に消費してしまいます。メッシュのメモリ消費を最小限に抑えるためにできることは、以下の通りです。

- **メッシュ圧縮を使う**：メッシュを積極的に圧縮することで、ディスクスペースを削減できます（ただし、ランタイム時のメモリは影響を受けません）。メッシュの量子化では、正確な結果を得られるとは限らないので、圧縮レベルを試して、モデルに合うものを見つけてください。
- **読み込み / 書き込みを無効にする**：このオプションを有効にすると、メモリ上にメッシュが複製され、その 1 つがシステムメモリに、もう 1 つが GPU メモリに保持されます。ほとんどの場合では、このオプションを無効にしてください（Unity 2019.2 以前では、このオプションはデフォルトで有効に設定されています）。
- **リグとブレンドシェイプを無効にする**：メッシュにスケルタルやブレンドシェイプのアニメーションが不要の場合は、可能な限りこれらのオプションを無効にしてください。
- **法線と接線を無効にする**：メッシュのマテリアルに法線や接線が全く必要ない場合は、これらのオプションを無効にすると、無駄をさらに削減できます。



メッシュインポート設定を確認する。

その他のメッシュ最適化方法

Player 設定でも、メッシュにいくつかの最適化を適用できます。

- **Vertex Compression** はチャンネルごとの頂点圧縮を設定します。例えば、ポジションとライトマップ UV 以外のすべての圧縮を有効にすることができます。これにより、メッシュのランタイムメモリ使用量を減らすことができます。

各メッシュの「Import Settings」にある「Mesh Compression」が頂点圧縮設定をオーバーライドすることに注意してください。その場合、メッシュのランタイムコピーは圧縮されず、より多くのメモリを使用する可能性があります。

— **Optimize Mesh Data** は、メッシュに適用されているマテリアルが必要としないデータ（接線、法線、色、UV など）をメッシュから削除します。

アセットを監査する

アセット監査プロセスを自動化することで、誤ったアセット設定の変更を防ぐことができます。インポート設定を標準化したり、既存のアセットを分析したりするのに役立つツールを以下にいくつか紹介します。

AssetPostprocessor

AssetPostprocessor を使用すると、インポートパイプラインにフックして、アセットのインポート前またはインポート時にスクリプトを実行できます。これは、モデル、テクスチャ、オーディオなどをインポートする前またはインポートした後に、設定をカスタマイズするよう促します。プリセットに似た方法ですが、コードを介して行います。このプロセスの詳細については、GDC 2023 トーク「ゲーム制作の各段階におけるテクニカルなヒント」をご覧ください。

UnityDataTools

UnityDataTools は、Unity が提供するオープンソースツールのコレクションで、Unity プロジェクトのデータ管理とシリアル化機能を強化することを目的としています。未使用のアセットの特定、アセットの依存関係の検出、ビルドサイズの削減など、プロジェクトデータの分析と最適化のための機能が含まれています。

ツールの詳細についてはこちらを、Asset Auditing については、ベストプラクティスガイドの「Unity における最適化」のセクションをご覧ください。

非同期テクスチャバッファ

Unity はリングバッファを使ってテクスチャを GPU にプッシュします。この非同期テクスチャバッファは、QualitySettings.asyncUploadBufferSize で手動で調整できます。

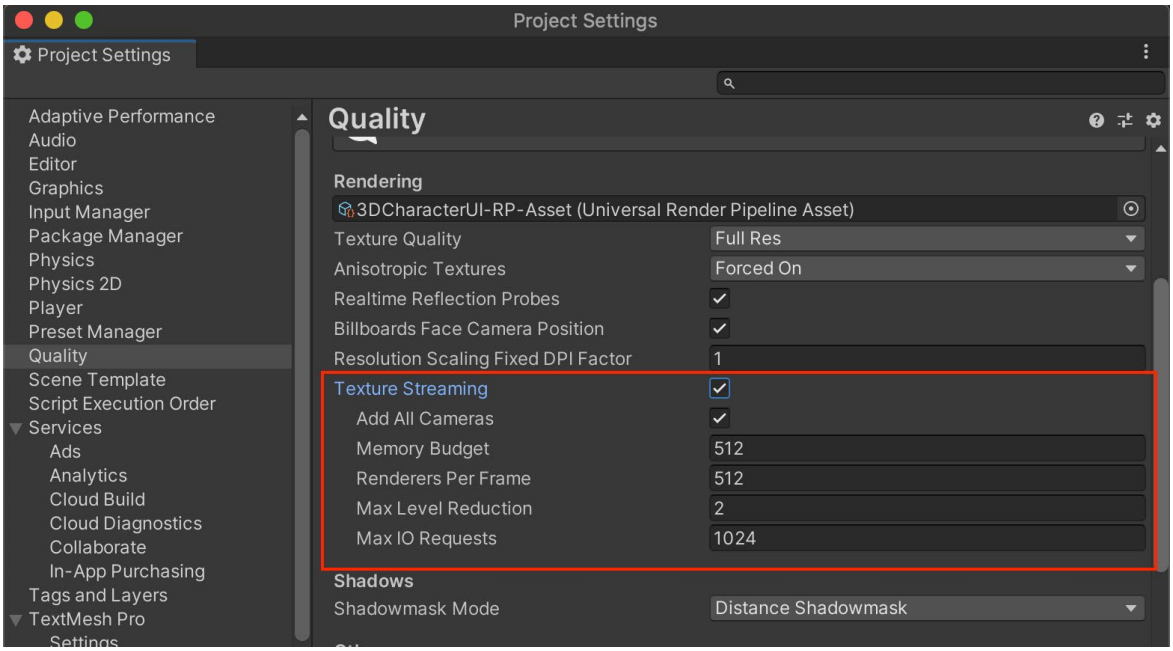
アップロード速度が遅すぎたり、一度に複数のテクスチャをロードしている間にメインスレッドがストールする場合は、テクスチャバッファを調整してください。通常は、シーンにロードする必要がある最大テクスチャのサイズに値（MB）を設定できます。

注：デフォルト値を変更すると、メモリを圧迫する可能性があります。また、Unity がリングバッファのメモリを割り当てた後は、それをシステムに戻すことはできません。GPU メモリがオーバーロードした場合、GPU は最も最近使われたテクスチャをアンロードし、次にカメラ錐台に入った際に CPU に再アップロードさせます。

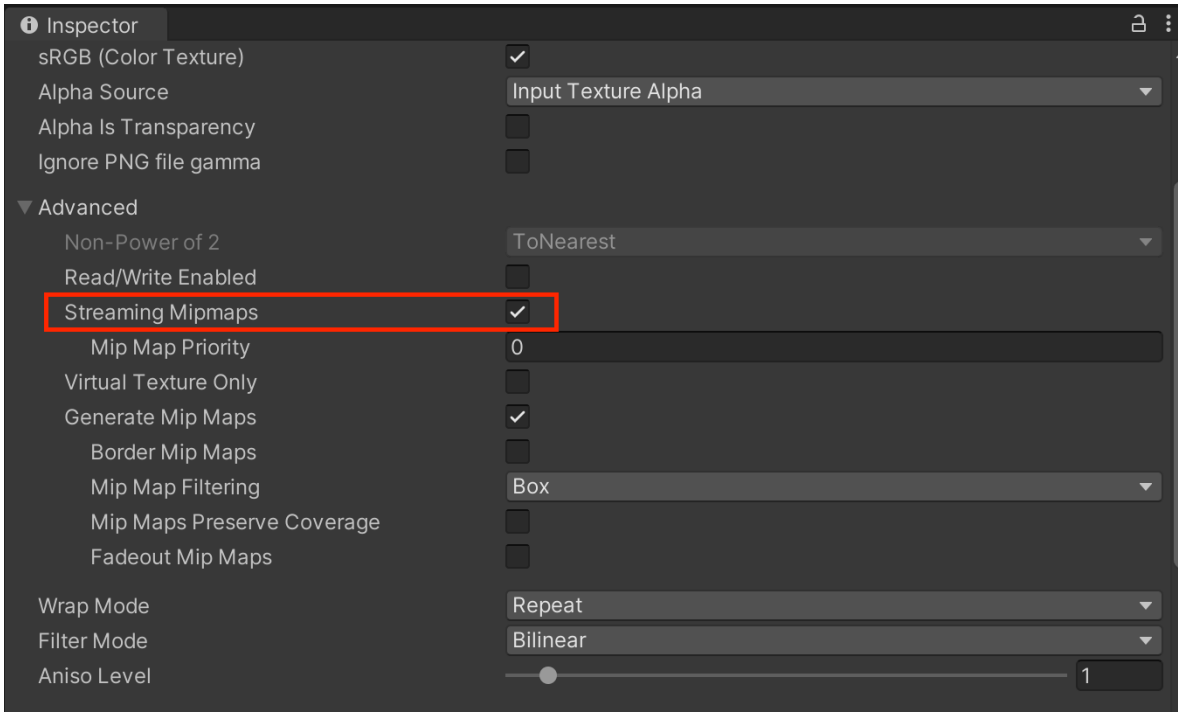
タイムスライスの Awake メソッド使用時におけるテクスチャバッファのメモリ制限については、「Unity のメモリ管理」ガイドを参照してください。

ミップマップとテクスチャをストリームする

ミニマップストリーミングシステムを使うと、どのミニマップレベルをメモリにロードさせるかをコントロールできます。これを有効にするには、Unity の Quality 設定（**「Edit」>「Project Settings」>「Quality」**）を開き、**「Texture Streaming」**にチェックを入れます。テクスチャの「Import Settings」の**「Advanced」**にある**「Streaming Mipmaps」**を有効にします。



Texture Streaming 設定



Streaming Mipmaps を有効にする。

このシステムは、現在のカメラ位置をレンダリングするのに必要なミップマップのみをロードするため、テクスチャに必要なメモリの総量を削減します。そうしなければ、Unity はデフォルトですべてのテクスチャをロードしてしまいます。テクスチャストリーミングは、大量の GPU メモリを大量に節約する代わりに、CPU リソースの消費はわずかです。

更なるコントロールを行いたい場合は、Mipmap Streaming API を使うことができます。テクスチャストリーミングは、ユーザー定義のメモリバジェット内に収まるように、ミップマップレベルを自動的に減らします。

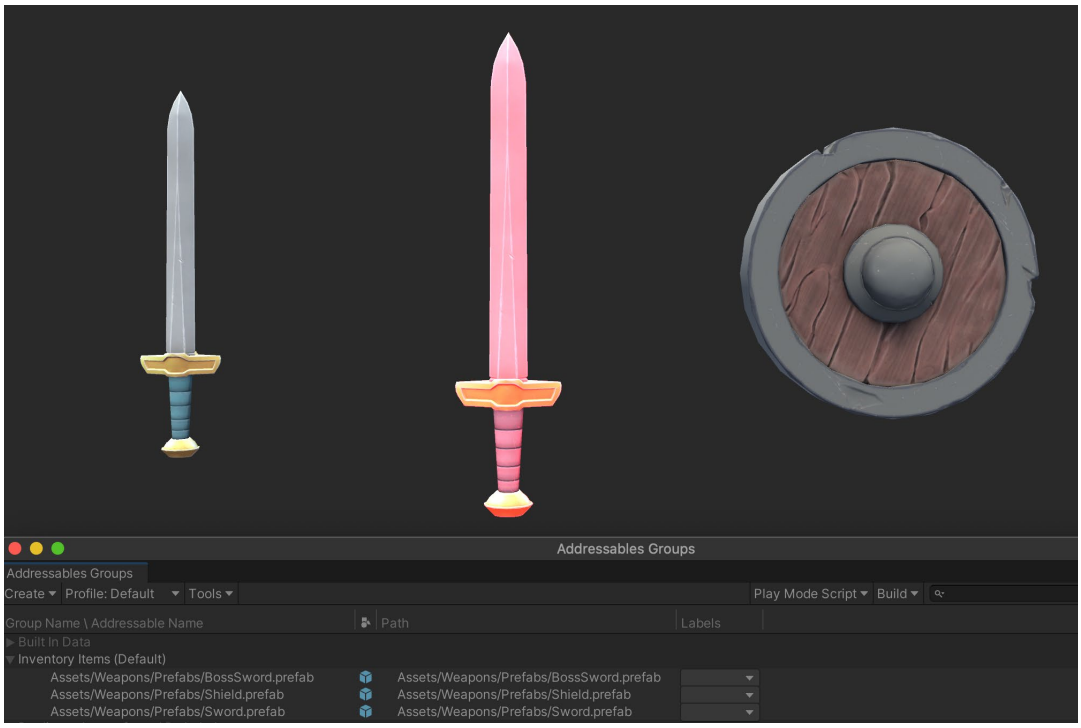
Addressable を使う

Addressable Asset System は、ゲームを構成するアセットの管理方法を簡素化します。シーン、プレハブ、テキストアセットなど、どのようなアセットでも「Addressable」とマークし、一意の名前を付けることができます。こうすることで、どこからでもこのエイリアスを呼び出せるようになります。

開発中のゲームがスタンドアロンやコンソールを含むプラットフォーム向けの場合は、以下の Addressable Asset Groups のデフォルト設定を見直すことを検討してください。

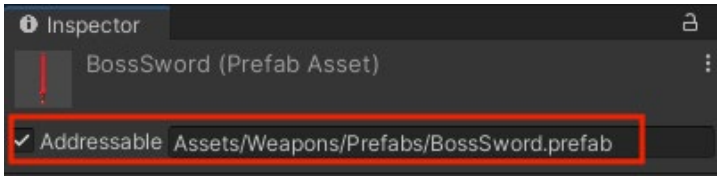
- コンソールでは「Asset Bundle Compression」を「LZ4」に設定する必要はなく（ファイルシステムには独自の優れた圧縮機能があるため）、また設定してしまうと、大きなバンドル内で小さな変更を効率的にパッチ適用することが難しくなります。コンソールプラットフォームでは「Uncompressed」に、Windows では「LZ4」に設定してください。
- 「Asset Bundle CRC」を「Enabled」に設定した場合、コンソールプラットフォームでのキャッシングを含めると、ロード時間が不必要に長くなります。ローカルのファイルシステムが信頼できるコンソールプラットフォームでは、この設定は「Disabled」に設定すべきです。ファイルの改ざんが問題視される場合は、Windows のキャッシュも含めて「Enabled」に設定することを検討してください。
- 「Bundle Naming Mode」設定が「Append Hash to Filename」の場合、コンソールのパッチシステムと相性が悪くなります。なぜなら、これらのシステムは「同じ」ファイルへの名前変更を認識しないからです。コンソールプラットフォームでは、この設定を「Filename」に設定する必要があります。

ゲームとアセットの間の抽象化レベルをこのように更に追加することで、別のダウンロード可能なコンテンツパックを作成するなど、特定の作業を効率化できます。Addressable を使えば、ローカルやリモートを問わず、アセットパックを簡単に参照できます。



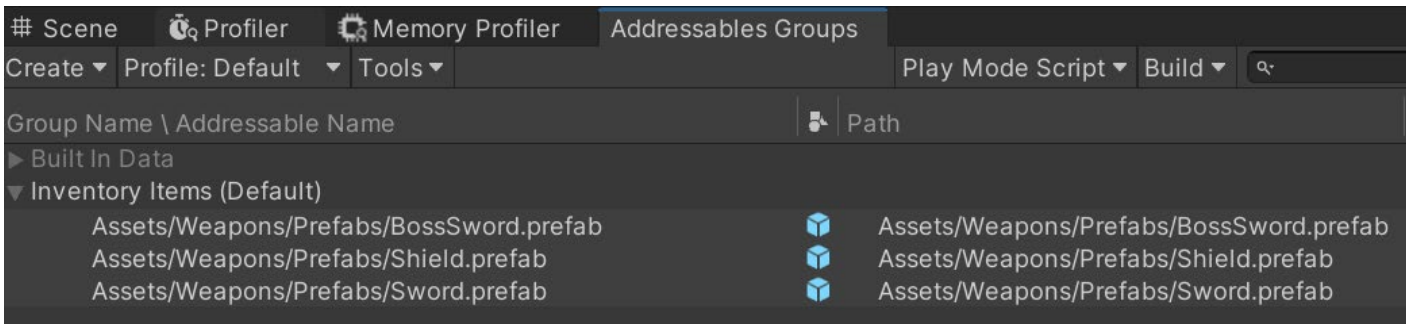
この例では、Addressable はプレハブのインベントリを追跡している。

Package Manager から Addressable パッケージをインストールします。すると、プロジェクトの各アセットやプレハブを「Addressable」にすることが可能となります。Inspector のアセット名の下のオプションにチェックを入れると、そのアセットにデフォルトの一意のアドレスが割り当てられます。



Addressable オプションがデフォルトの Addressable Name で有効になっている

一度マークされると、対応するアセットが**「Window」>「Asset Management」>「Addressables」>「Groups」**ウィンドウに表示されます。



Addressables Groups から、各アセットのカスタムアドレスとその位置を確認できる。

アセットが別の場所にホストされている場合でも、ローカルに保存されている場合でも、システムは Addressable Name の文字列を使用してアセットを検索します。Addressable プレハブは、必要になるまでメモリにロードされず、使用されなくなると関連するアセットが自動的にアンロードされます。

以下は、Addressable に関して役立つ 2 つのブログ記事です。

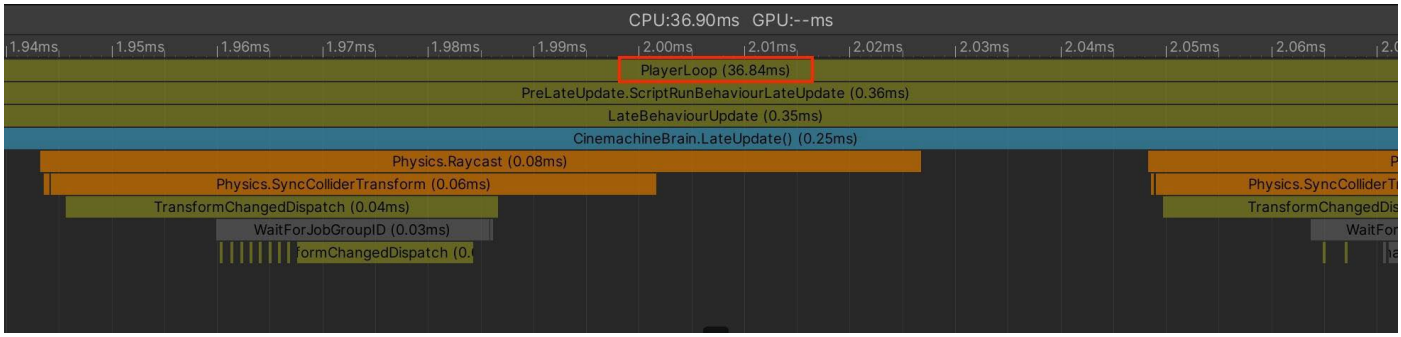
— 「最適化の最前線から：Addressables を使ってメモリを節約する」では、メモリをより効率的に使用するために、Addressable Groups を整理する方法の例を紹介しています。

— 「Addressables：プランニングとベストプラクティス」では、Addressable アセットの整理、ビルド、ロード、アンロードに関するヒントを提供しています。

プログラミングとコードアーキテクチャ

Unity PlayerLoop にはゲームエンジンのコアと相互作用するための関数が含まれています。この構造体には、初期化とフレームごとの更新を処理するシステムがいくつかあります。すべてのスクリプトは、この PlayerLoop に依存してゲームプレイを作成します。

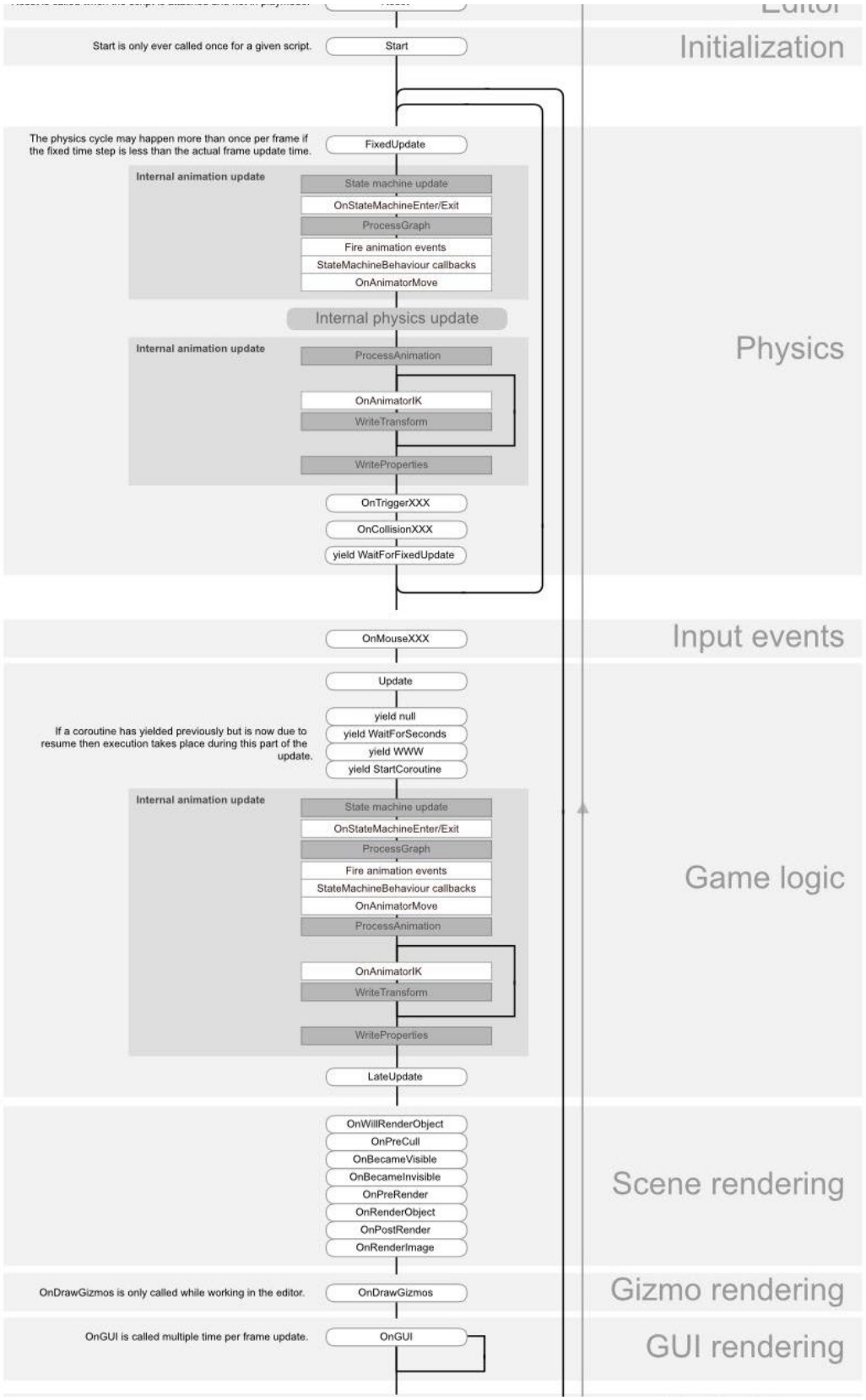
プロファイリングを行うと、プロジェクトのユーザーコードが PlayerLoop 下に表示されます（Editor コンポーネントは EditorLoop の下に表示されます）。



Profiler は、エンジン全体の実行コンテキスト内で、カスタムスクリプト、設定、グラフィックスを表示する。

Unity PlayerLoop を理解する

Unity のフレームループの実行順序を確実に理解しておくようにしてください。すべての Unity スクリプトは、いくつかのイベント関数を決められた順序で実行します。**Awake**、**Start**、**Update** といった、スクリプトのライフサイクルを作成する関数の違いを理解する必要があります。Low-Level API を使用して、プレーヤーの更新ループにカスタムロジックを追加できます。



PlayerLoop とスクリプトのライフサイクル

毎フレーム実行されるコードを最小化する

コードが毎フレーム実行されるべきか考えてください。そして、Update、LateUpdate、FixedUpdate から不必要なロジックを取り除きます。これらのイベント関数は、毎フレーム更新しなければならないコードを置くのに便利な場所であると同時に、その頻度で更新する必要のないロジックをあぶり出すことができます。可能な限り、状況が変化した場合にのみロジックを実行するようにしましょう。

Update を使用する必要がある場合は、***n*** フレームごとにコードを実行させることを検討してみてください。これは、重い作業負荷を複数のフレームに分散させる一般的な手法であるタイムスライシングを適用する方法のひとつです。以下の例では、3 フレームごとに **ExampleExpensiveFunction** を実行しています。

```
private int interval = 3;
void Update()
{
    if (Time.frameCount % interval == 0)
    {
        ExampleExpensiveFunction();
    }
}
```

さらに良い方法は、**ExampleExpensiveFunction** がデータセットに対して何らかの操作を行う場合、タイムスライシングを使用して、そのデータの異なるサブセットに対して毎フレーム操作を行うことです。n フレームごとにすべての作業を行うのではなく、1 フレームごとに 1/*n* の作業を行うことで、CPU が周期的にスパイクするのではなく、全体としてより安定し予測可能なパフォーマンスが得られます。

コツは、これを他のフレームで実行される作業と交互に実行することです。この例では、**Time.frameCount % interval == 1** または **Time.frameCount % interval == 2** の場合に、負荷が高い他の関数を「スケジュール」できます。

あるいは、カスタム UpdateManager クラス（下記）を使い、サブスクライブされたオブジェクトを n フレームごとに更新します。

Start/Awake では重いロジックを避ける

最初のシーンがロードされると、以下の関数が各オブジェクトに対して呼び出されます。

- **Awake**
- **OnEnable/OnDisable**
- **Start**

アプリケーションが最初のフレームをレンダリングするまでは、これらの関数では負荷が高いロジックは避けてください。そうしなければ、必要以上にロード時間が長くなる可能性があります。

最初のシーンをロードする上で詳細を知りたい場合は、「イベント関数の実行順序」を参照してください。

負荷が高い関数をキャッシュする

一般的には、Update メソッドでの呼び出しを避けるために、Awake か Start のどちらかで参照をキャッシュするのが最適です。

これらのメソッドを頻繁に呼び出すと、CPU スパイクの原因となります。可能な限り、負荷が高い関数（つまり MonoBehaviour.Awake と MonoBehaviour.Start）は初期化段階で実行してください。必要な参照をキャッシュし、後で再利用しましょう。

以下は、GetComponent 呼び出しが繰り返し非効率的に使用されている例です。

```
void Update()
{
    Renderer myRenderer = GetComponent<Renderer>();
    ExampleFunction(myRenderer);
}
```

代わりに、**GetComponent** を一度だけ呼び出し、その結果をキャッシュするようにします。キャッシュされた結果は、それ以上 **GetComponent** を呼び出すことなく、**Update** で再利用できます。

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}
void Update()
{
    ExampleFunction(myRenderer);
}
```

Unity 2020.2 より前のバージョンでは、**GameObject.Find**、**GameObject.GetComponent**、**Camera.main** は非常に負荷が高いものでしたが、現在はそうではありません。とは言え、**Update** メソッドでこれらを呼び出すのは避け、結果をキャッシュして上記のプラクティスに従うのが最適です。

空の Unity イベント関数を避ける

空の MonoBehaviours でもリソースが必要なので、空の Update メソッドや LateUpdate メソッドは削除してください。

テストにこれらのメソッドを使う場合は、プリプロセッサーディレクティブを使います。

```
#if UNITY_EDITOR
void Update()
{
}
#endif
```

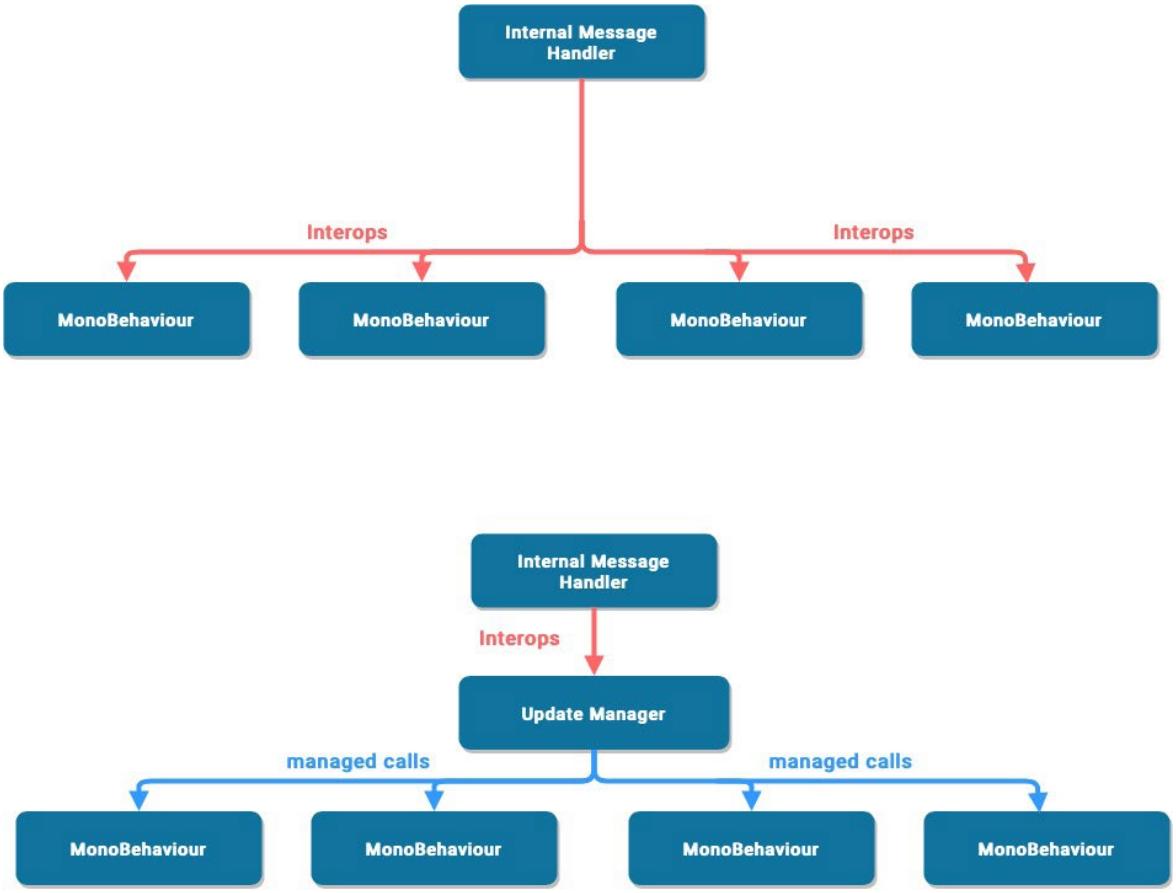
ここでは、不必要なオーバーヘッドがビルドに入り込むことなく、Update in Editor を自由にテストに使うことができます。

カスタム Update Manager をビルドする

Update または LateUpdate の一般的な使用パターンは、何らかの条件が満たされた際にのみロジックを実行することです。これにより、1 フレームごとに多くのコールバックが発生し、この条件をチェックする以外、事実上コードは実行されません。

Unity が Update や LateUpdate などの Message メソッドを呼び出すたびに、**interop 呼び出し**、つまり C/C++ 側からマネージ C# 側への呼び出しが行われます。オブジェクトの数が少ない場合、これは問題ありません。オブジェクト数が何千ともなると、このオーバーヘッドは大きくなります。

この方法で Update や LateUpdate を使用する大規模なプロジェクト（オープンワールドゲームなど）がある場合は、カスタム UpdateManager の作成を検討してください。コールバックが必要な際はアクティブオブジェクトをこの UpdateManager にサブスクライブし、不要な際はサブスクライブを解除してください。このパターンを使えば、Monobehaviour オブジェクトへの interop 呼び出しの多くを減らすことができます。



カスタム Update Manager をビルドすることで、interop 呼び出しを削減できる。

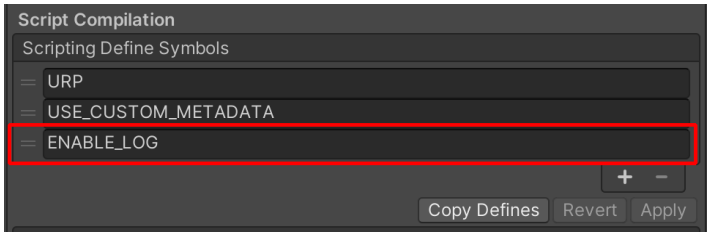
実装例と見込まれる成果については、「Unity のゲームエンジン固有の最適化テクニック」を参照してください。

Debug Log ステートメントを削除する

Log ステートメント（特に Update、LateUpdate、FixedUpdate）はパフォーマンスを低下させる可能性があります。ビルドを行う前に Log ステートメントを無効にしてください。

これをより簡単に行うには、プリプロセッサディレクティブと併せて Conditional 属性を作成することを検討してください。例えば、以下のようなカスタムクラスを作成します。

```
public static class Logging
{
    [System.Diagnostics.Conditional(“ENABLE_LOG”)]
    static public void Log(object message)
    {
        UnityEngine.Debug.Log(message);
    }
}
```



カスタムのプリプロセッサーディレクティブを追加すると、スクリプトを分割できる。

カスタムクラスでログメッセージを生成します。**「Player Settings」>「Scripting Define Symbols」**で**「ENABLE_LOG」**プリプロセッサーを無効にすると、Log ステートメントを一斉消去できます。

同じことが、Debug.DrawLine や Debug.DrawRay など、Debug クラスの他の使用例にも当てはまります。これらもまた、開発中のみの使用を意図したものであり、パフォーマンスに大きな影響を与える可能性があります。

Unity プロジェクトにおいて、文字列やテキストを扱うことはパフォーマンス問題を起こす一般的な原因です。Log ステートメントとその負荷の高い文字列形式を削除することは、大きな効率化につながります。

スタックトレースのログ出力を無効にする

「Player Settings」の「Stack Trace」オプションを使用すると、表示されるログメッセージのタイプをコントロールできます。

アプリケーションがリリースビルドでエラーや警告メッセージのログを出力している場合（クラッシュレポートを生成するためなど）、パフォーマンスを向上させるためにスタックトレースを無効にしてください。

Stack Trace オプション

文字列パラメーターの代わりにハッシュ値を使用する

Unity は、アニメーター、マテリアル、およびシェーダーのプロパティを参照するために、システム内部で文字列名を使用することはありません。素速く処理を行うため、すべてのプロパティ名はプロパティ ID にハッシュ化され、これらの ID が実際にプロパティを参照するために使用されます。

アニメーター、マテリアル、シェーダーで Set メソッドや Get メソッドを使用する場合は、文字列値メソッドではなく整数値メソッドを使用します。文字列メソッドは単に文字列をハッシュ化し、ハッシュされた ID を整数値メソッドに転送します。

Animator のプロパティ名には Animator.StringToHash を使用し、マテリアルとシェーダーのプロパティ名には Shader.PropertyToID を使用します。初期化時にこれらのハッシュ値を取得し、Get メソッドや Set メソッドに渡す必要がある場合のために変数にキャッシュしておきます。

適切なデータ構造体を選択する

データ構造体の選択は、1 フレームあたり何千回も反復するため、効率に影響します。コレクションにリスト、配列、ディクショナリのどれを使うべきか迷っていますか。適切な構造体を選択する際の参考として、C# のデータ構造体に関する MSDN ガイドに従いましょう。

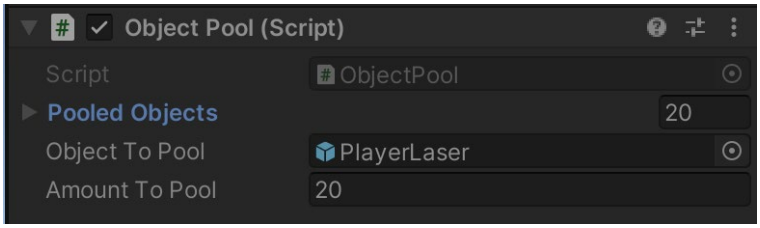
ランタイム時のコンポーネントの追加は避ける

ランタイム時に AddComponent を呼び出すには、それなりの負荷がかかります。Unity は、ランタイム時にコンポーネントを追加するたびに、重複するコンポーネントやその他の必要なコンポーネントをチェックする必要があります。

必要なコンポーネントがすでに設定されたプレハブをインスタンス化するほうが、一般的にパフォーマンスが高くなります。

オブジェクトプールを使用する

Instantiate と Destroy は、ガベージとガベージコレクション（GC）のスパイクを発生させる可能性があり、一般的に処理が遅いです。大量のオブジェクトをインスタンス化する必要がある場合は、オブジェクトプールのテクニックを適用して GC スパイクを回避してください。

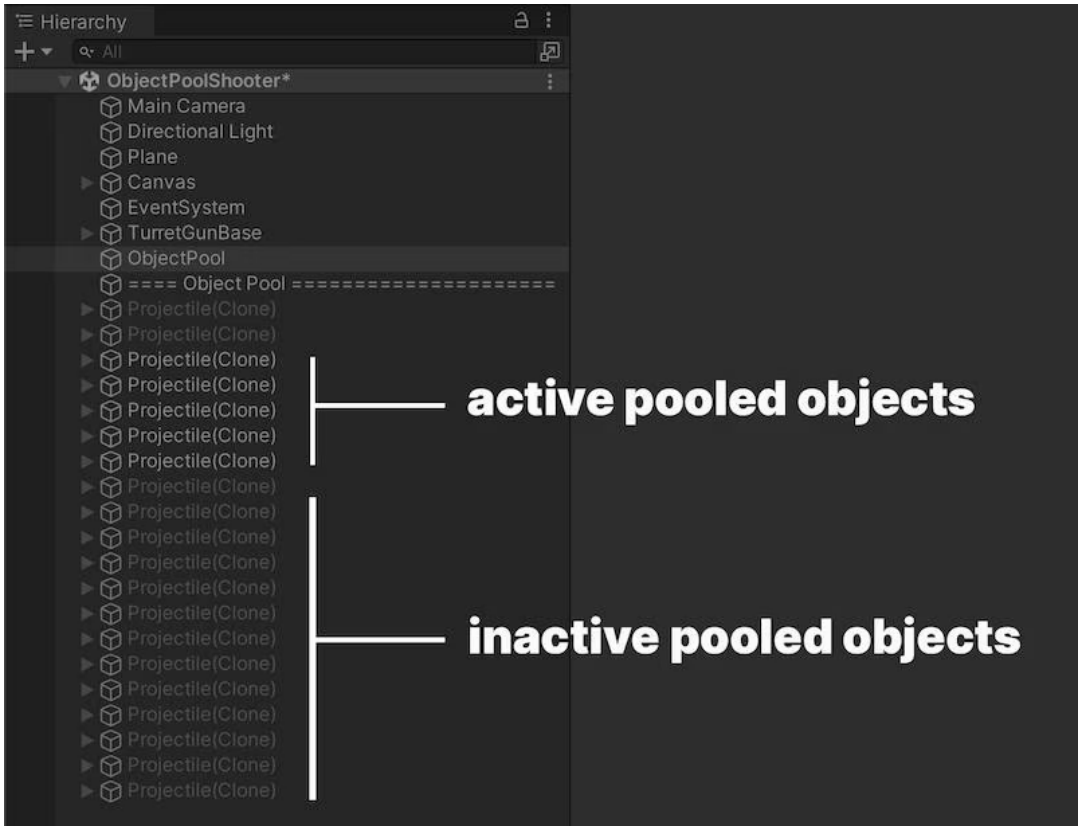


この例では、ObjectPool が再利用のために 20 個の PlayerLaser インスタンスを作成している。

オブジェクトプールは、繰り返し行われる生成と破棄の呼び出しに必要な CPU の処理能力を削減することで、パフォーマンスを最適化するデザインパターンです。これを使うと、既存のゲームオブジェクトを何度でも再利用できます。

オブジェクトプールの重要な機能は、オンデマンドでオブジェクトを作成および破棄するのではなく、事前にオブジェクトを作成し、プールに保存できることです。オブジェクトが必要となった際にプールから取り出して使用します。不要になると、破棄される代わりに、プールに戻されます。

ゲームオブジェクトを定期的にインスタンス化して破棄するのではなく（銃で弾を撃つなど）、再利用や再利用が可能な事前割り当てされたオブジェクトのプールを使用します。



こうすることで、プロジェクト内のマネージ割り当ての数を減らし、ガベージコレクションの問題を防ぐことができます。

Unity には、UnityEngine.Pool 名前空間からアクセス可能なビルトインのオブジェクトプーリング機能が含まれています。Unity 2021 LTS 以降で利用可能なこの名前空間は、オブジェクトプールの管理を容易にし、オブジェクトのライフサイクルやプールサイズのコントロールなどの作業を自動化します。

シンプルなオブジェクトプーリングシステムを Unity で作成する方法について詳しく知りたい場合は、こちらを確認してください。また、Unity Asset Store で入手可能なこのサンプルプロジェクトでは、オブジェクトプールパターンやその他多くのパターンを Unity シーンに実装して見ることができます。

Transform の更新を一度にする

Transform を動かす場合は、Transform.SetPositionAndRotation を使用して位置と回転を一度に更新します。これにより Transform を二度修正することにより発生するオーバーヘッドを回避できます。

ランタイム時にゲームオブジェクトを Instantiate する必要がある場合、単純な最適化としては、インスタンス化する際にペアレント化して再配置することです。

```
GameObject.Instantiate(prefab, parent);
GameObject.Instantiate(prefab, parent, position, rotation);
```

Object.Instantiate に関する詳細は、Scripting API を参照してください。

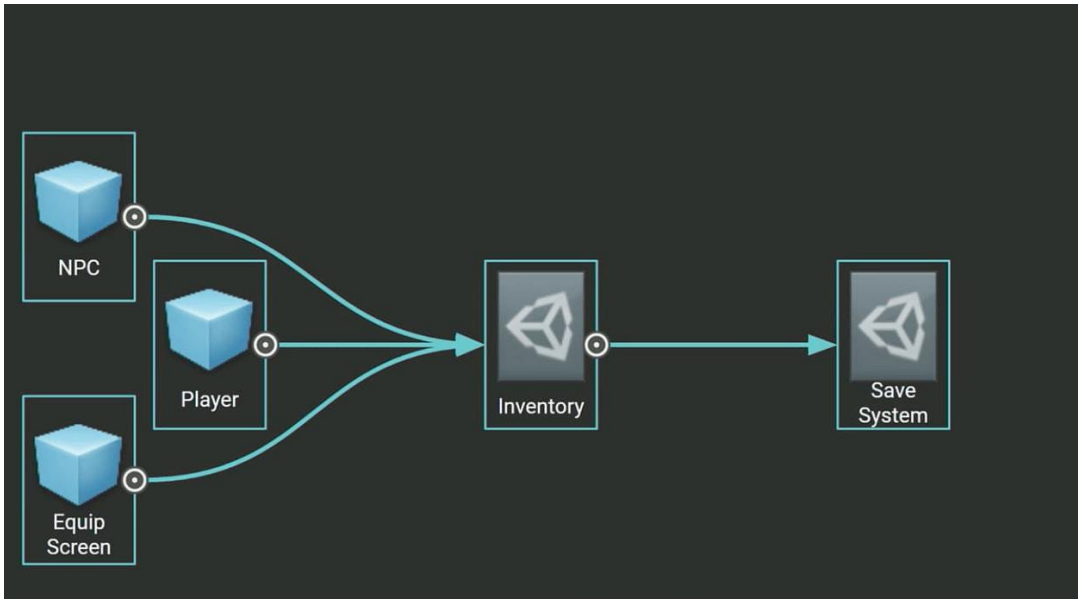
ScriptableObject を使用する

MonoBehaviour ではなく、ScriptableObject に静的な不変値や設定を保存します。ScriptableObject はプロジェクト内に存在するアセットで、一度だけ設定する必要があります。

MonoBehaviours は、ゲームオブジェクト（デフォルトでは併せて Transform）をホストとして動作させる必要があるため、余分なオーバーヘッドが発生します。つまり、1 つの値を保存する前に、多くの未使用データを作成する必要があります。ScriptableObject は、ゲームオブジェクトと Transform を削除することで、このメモリフットプリントをスリム化します。また、プロジェクトレベルでデータを保存するので、複数のシーンから同じデータにアクセスする必要がある場合に便利です。

よくある使用例は、ランタイム時に変更する必要のない、同じ重複データに依存する多くのゲームオブジェクトがある場合です。このようにゲームオブジェクトごとに重複したローカルデータを持つのではなく、ScriptableObject に流すことができます。そして、各オブジェクトは、データそのものを複製するのではなく、共有データアセットへの参照を保存します。これは、何千ものオブジェクトを扱うプロジェクトにおいて、大幅なパフォーマンス向上をもたらします。

ScriptableObject にフィールドを作成して値や設定を保存し、MonoBehaviours で ScriptableObject を参照します。



この例では、Inventory という ScriptableObject がさまざまなゲームオブジェクトの設定を保持している。

ScriptableObject のフィールドを使用することで、その MonoBehaviour でオブジェクトをインスタンス化するたびにデータが重複するのを防ぐことができます。

ソフトウェア設計では、これはフライウェイトパターンと呼ばれる最適化です。ScriptableObject を使ってこのようにコードを再構築すれば、多くの値をコピーする必要がなくなり、メモリフットプリントも削減できます。フライウェイトパターンをはじめとする多くのパターンや設計原則について詳しく学びたい場合は、eBook「*デザインパターンと SOLID でコードをレベルアップする*」をご覧ください。

「Introduction to ScriptableObjects」チュートリアルを視聴して、ScriptableObjects があなたのプロジェクトにどのように役立つかを見つけましょう。こちらの Unity ドキュメントとテクニカルガイド「*Unity で ScriptableObject を使用してモジュラーゲームアーキテクチャを作成する*」も参照してください。

ラムダ式を避ける

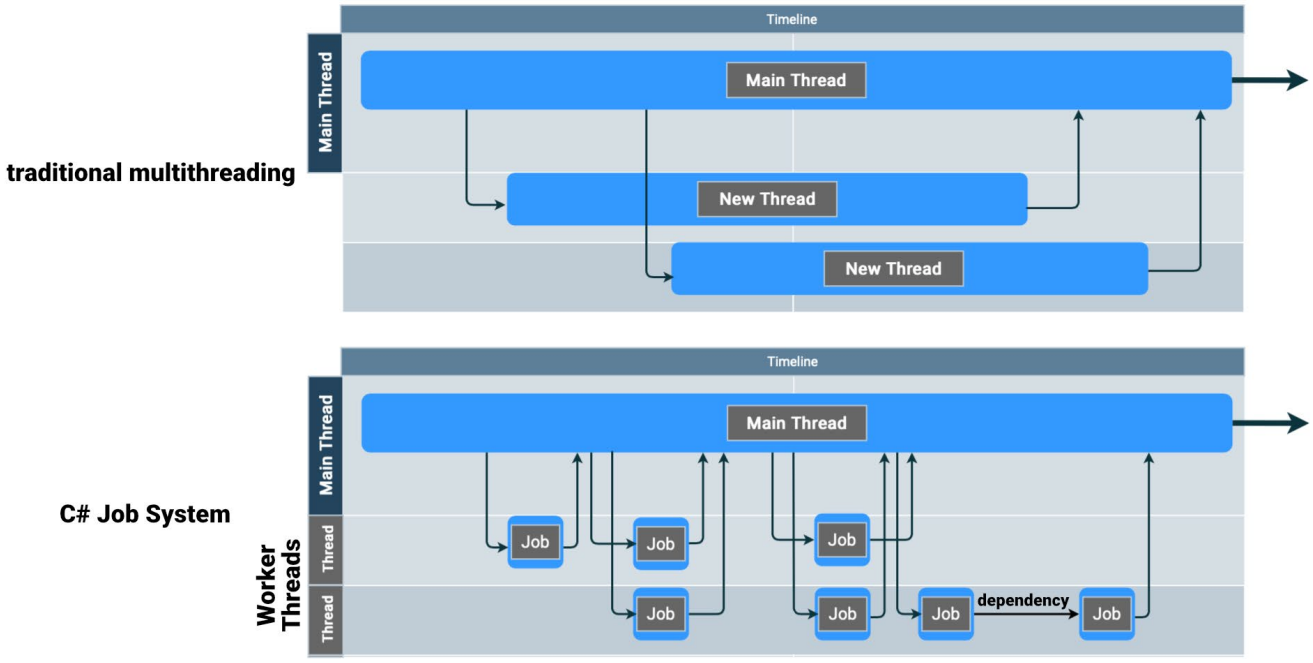
ラムダ式はコードを単純化しますが、これにはコストがかかります。ラムダを呼び出すと、デリゲートも生成されます。コンテキスト（this、インスタンスメンバ、ローカル変数など）をラムダに渡すと、デリゲートのキャッシュが無効になります。そのような場合、これを頻繁に呼び出すと、相当量のメモリートラフィックが発生する可能性があります。

ラムダ式の使用中にクロージャを含むメソッドをリファクタリングしてください。方法についてはこちらをご覧ください。

C# Job System

最近の CPU はマルチコアを備えていますが、アプリケーションがそれらを利用するためにはマルチスレッドコードが必要です。Unity のジョブシステムでは、大きなタスクを小さなチャンクに分け、余った CPU コアで並列に実行することで、パフォーマンスを大幅に向上させることができます。

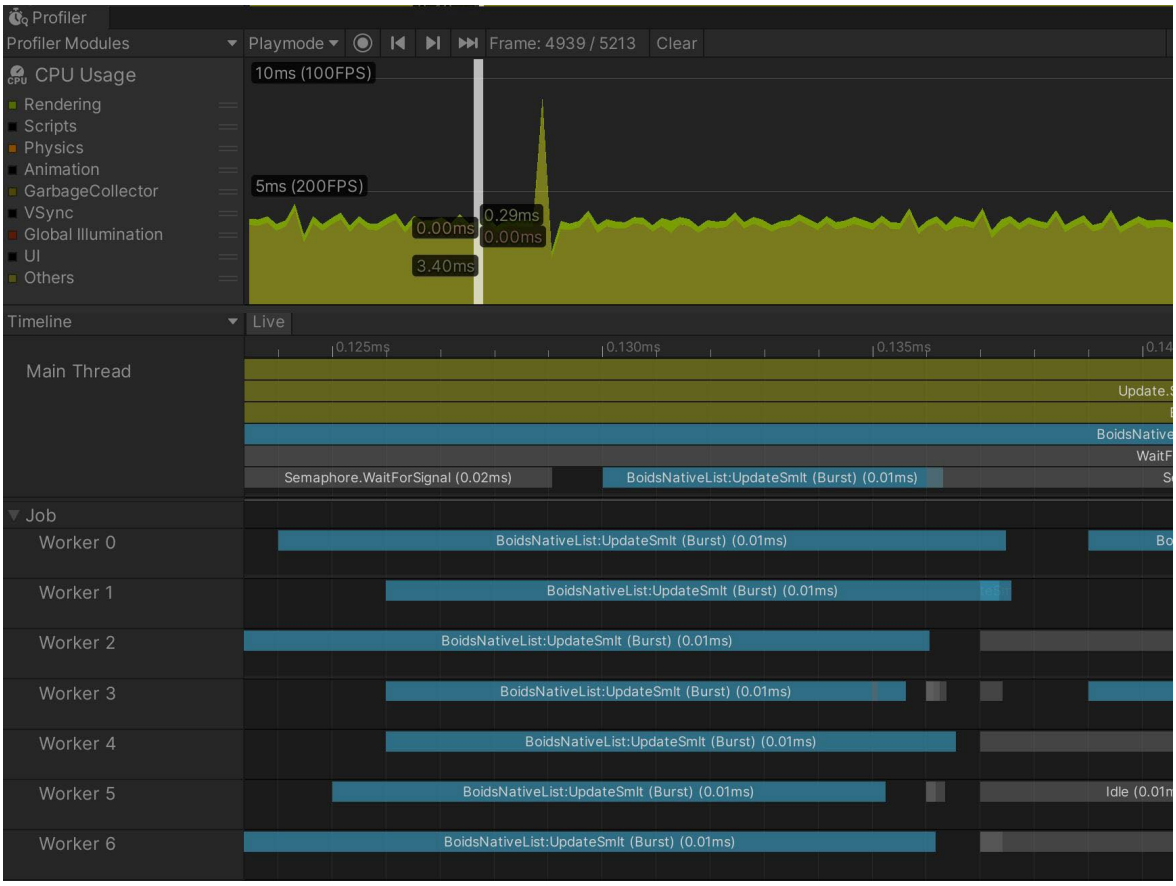
マルチスレッドプログラミングでは、メインスレッドである CPU の実行スレッドが、タスクを処理するために他のスレッドを作成することがよくあります。これらの追加のワーカースレッドは、作業が完了するとメインスレッドと同期します。



従来のマルチスレッドプログラミングでは、スレッドは生成と破棄を繰り返す。C# Job System では、小さなジョブはスレッドのプールで実行される。

長時間実行するタスクがいくつかある場合、このマルチスレッドアプローチは効果的です。しかし、1 秒間に 30 ～ 60 フレームで多くの短いタスクを処理しなければならないゲームアプリケーションにとっては非効率です。

そのため Unity では、C# Job System と呼ばれる少し異なるマルチスレッドのアプローチを採用しています。寿命の短いスレッドを多数生成するのではなく、作業を**ジョブ**と呼ばれる小さな単位に分割します。



Profiler の Timeline ビューには、ワーカースレッドで実行されているジョブが表示される。

これらのジョブはキューに入り、ワーカースレッドの共有プールで実行されるようにスケジュールされます。JobHandle は依存関係を作成するのに役立ち、ジョブが正しい順序で実行されるようにします。

マルチスレッドの潜在的な問題のひとつに競合状態というものがあり、これは 2 つのスレッドが同時に共有変数にアクセスしたときに発生します。これを防ぐために、Unity のマルチスレッドでは、ジョブの実行に必要なデータを分離する安全システムを使用しています。C# Job System は、各ジョブをジョブ構造体のコピーで起動するため、競合状態が発生しません。

Unity の C# Job System を使用するには、以下のガイドに従ってください。

— クラスを構造体に変更します。ジョブとは、IJob インターフェースを実装した構造体のことです。多数のオブジェクトに対して同じタスクを実行する場合、マルチコアで実行するために IJobParallelFor を使うこともできます。

— ジョブに渡されるデータは、blittable 型でなければなりません。参照型を削除し、blittable 型のデータのみをコピーとしてジョブに渡します。

— 各ジョブ内の作業は安全のために分離されたままなので、NativeContainer を使用してメインスレッドに結果を送り返します。Unity Collections パッケージの NativeContainer は、ネイティブメモリ用の C# ラッパーを提供します。そのサブタイプ（NativeArray、NativeList、NativeHashMap、NativeQueue など）は、同等の C# データ構造体と同様に動作します。

C# Job System を使用して、プロジェクトで CPU パフォーマンスを最適化する方法については、ドキュメントを参照してください。

Burst コンパイラー

Burst compiler は Job System を補完します。Burst は、LLVM を使用して IL/.NET バイトコードを最適化されたネイティブコードに変換します。これには、Package Manager から **com.unity.burst** を追加することで簡単にアクセスできます。

Burst により、Unity 開発者はパフォーマンスを向上させながら、利便性のために C# のサブセットを使い続けることができます。

Burst コンパイラーをスクリプトで有効にする方法は以下の通りです。

— 静的変数を取り除きます。リストに書き込む必要がある場合は、NativeDisableContainerSafetyRestriction 属性で装飾された NativeArray の使用を検討してください。これにより、並列ジョブは NativeArray に書き込むことができます。

— Mathf 関数の代わりに Unity.Mathematics 関数を使用します。

— ジョブ定義を BurstCompile 属性で装飾します。

```
  [BurstCompile]
public struct MyFirstJob :IJob
{
       public NativeArray<float3> ToNormalize;

       public void Execute()
         {
              for (int i = 0; i < ToNormalize.Length; i++)
              {
                     ToNormalize[i] = math.normalize(ToNormalize[i]);
              }
       }
}
```

これは、float3 の配列に対して実行し、ベクトルを正規化する Burst ジョブの例です。上記の通り、これは Unity Mathematics パッケージを使用します。

C# Job system と Burst コンパイラーは、両方とも Unity の DOTS（Data-Oriented Tech Stack）の一部を構成しています。しかし、これらは「従来の」Unity ゲームオブジェクトでも、Entity Component System でも使用できます。

最新のドキュメントを参照し、eBook「*上級 Unity 開発者向け Data-Oriented Technology Stack 入門*」をダウンロードして、C# Job System と併用した際に、Burst がどのようにワークフローを加速させるかについて学びましょう。

プロジェクト設定

いくつかの設定は、パフォーマンスに影響を与える可能性があります。

不必要な Player および Quality の設定を無効にする

Player 設定で、「Auto Graphics API」を無効にし、各ターゲットプラットフォームでサポートする予定のないグラフィックス API を削除します。これにより、過剰なシェーダーバリアントの生成を防ぐことができます。アプリケーションが古い CPU をサポートしていない場合は、「Target Architectures」を無効にしましょう。

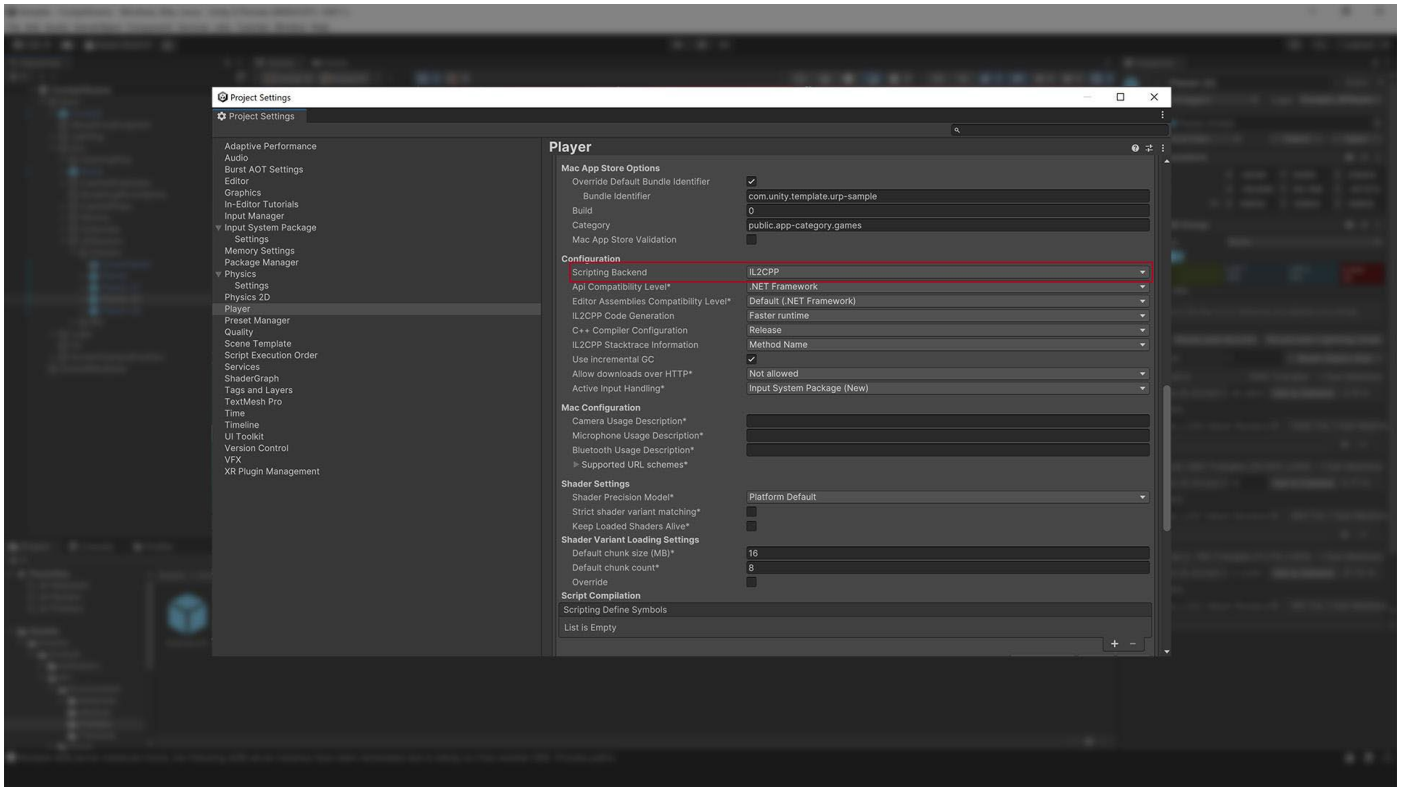
Quality 設定で、不必要な「Quality」レベルを無効にします。

IL2CPP に切り替える

スクリプティングバックエンドを Mono から IL2CPP（Intermediate Language to C++）に切り替えることをお勧めします。これにより、全体的にランタイムパフォーマンスが向上します。

ただし、ビルト時間は増加するので注意してください。中には、より速いイテレーションのためにローカルでは Mono を使用し、ビルドマシンやリリース候補には IL2CPP に切り替えることを好む開発者もいるでしょう。ビルド時間を抑えるには、ドキュメント「IL2CPP のビルド時間を最適化する」を参照してください。

IL2CPP が唯一のオプションである PlayStation プラットフォームでは、**「Player Settings」 > 「Other Settings」 > 「IL2CPP optimization level」**設定を確認してください。開発中はビルド時間を短縮するために、最適でないオプションを使用してください。プロファイリングや最終リリース段階では、**Optimized Compile、Remove Unused Code、Optimized リンク**を選択してください。



スクリプティングバックエンドを IL2CPP に切り替える

このオプションを使用すると、Unity はスクリプトとアセンブリの IL コードを C++ に変換してから、ターゲットプラットフォーム向けにネイティブバイナリファイル（.exe、.apk、.xap など）を作成します。

ビルド時間を最適化する方法を紹介したこちらのドキュメントを参照してください。

大規模な階層を避ける

階層を分割します。ゲームオブジェクトを階層にネストさせる必要がない場合は、ペアレンティングをシンプルにしてください。階層が小さいと、シーン内の Transform を更新する際、マルチスレッドを活用することがメリットとなります。階層が複雑だと、不必要な Transform の計算を引き起こし、ガベージコレクションのコストが増加してしまいます。

Transforms のベストプラクティスについては、Unite トークをご覧ください。

グラフィクス

ライティングとエフェクトは非常に複雑なため、最適化を試みる前にレンダーパイプラインドキュメントを確認することをお勧めします。

レンダーパイプラインに専念する

シーンのライティングを最適化することは、厳密な物理的正確さの追求よりも、試行錯誤を繰り返すプロセスです。そのプロセスは通常、アーティスティックな方向性とレンダーパイプラインに依存します。

シーンのライティングを始める前に、利用可能なレンダーパイプラインの 1 つを選択する必要があります。レンダーパイプラインは、シーンの内容を画面に表示するための一連の操作を実行します。

Unity は、異なる機能とパフォーマンス特性を持つ、3 つのプリビルトレンダーパイプラインを提供しています。または、自身で独自のパイプラインを作成することも可能です。

1. ユニバーサルレンダーパイプライン（URP）は、プリビルトのスクリプタブルレンダーパイプライン（SRP）です。URP は、モバイルからハイエンドコンソールや PC まで、さまざまなプラットフォームで最適化されたグラフィックスを作成するための、アーティストに優しいワークフローを提供します。URP はビルトインレンダーパイプラインの後継で、旧来のパイプラインでは利用できなかったグラフィックスとレンダリング機能を提供します。パフォーマンスを維持するために、ライティングとシェーディングの計算コストを削減するトレードオフを行います。モバイルや VR を含む、大抵のプラットフォームにリーチしたい場合は、URP を選択してください。

   URP 機能の完全な概要については、eBook「*上級 Unity クリエイター向けのユニバーサルレンダーパイプライン（URP）*」をご覧ください。

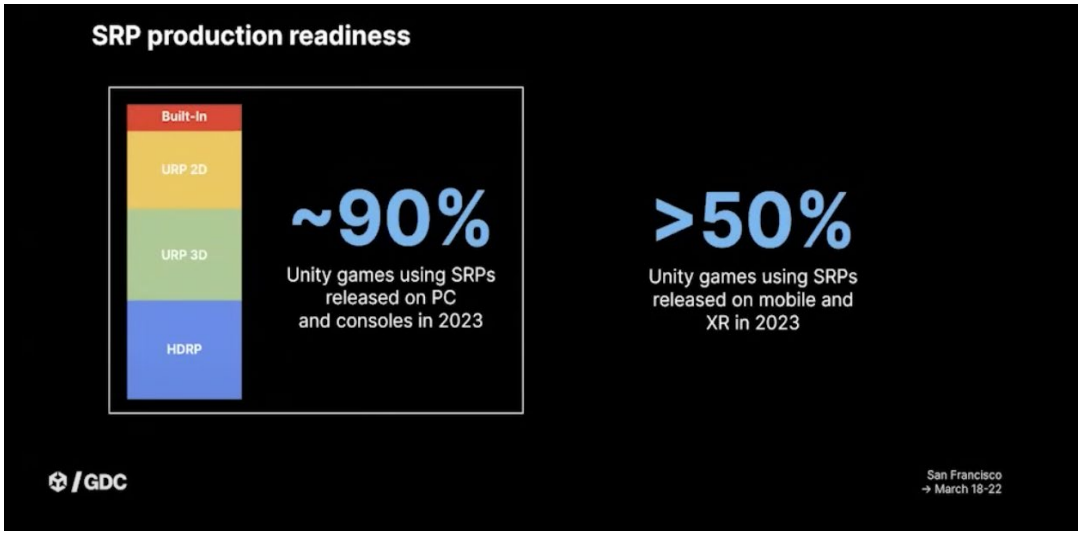
2. HD レンダーパイプライン（HDRP）は、最先端の高忠実度グラフィックスのために設計された、もう 1 つのプリビルト SRP です。HDRP は、PC、Xbox、PlayStation といったハイエンドのハードウェアをターゲットとしています。[2] 高度なライティング、リフレクション、シャドウを駆使し、フォトリアルなグラフィックでゲームに最高レベルのリアリズムを作成するために推奨されるレンダーパイプラインです。HDRP は物理ベースのライティングとマテリアルを使用し、改良されたデバッグツールをサポートしています。

   HDRP 機能の完全な概要については、eBook「*HD レンダーパイプライン（HDRP）におけるライティングと環境*」をご覧ください。

3. ビルトインレンダーパイプラインは Unity の旧型で汎用的なレンダーパイプラインであり、カスタマイズ性が限定されています。

PC/ コンソール向けの Unity ゲームの約 90% が SRP を使用

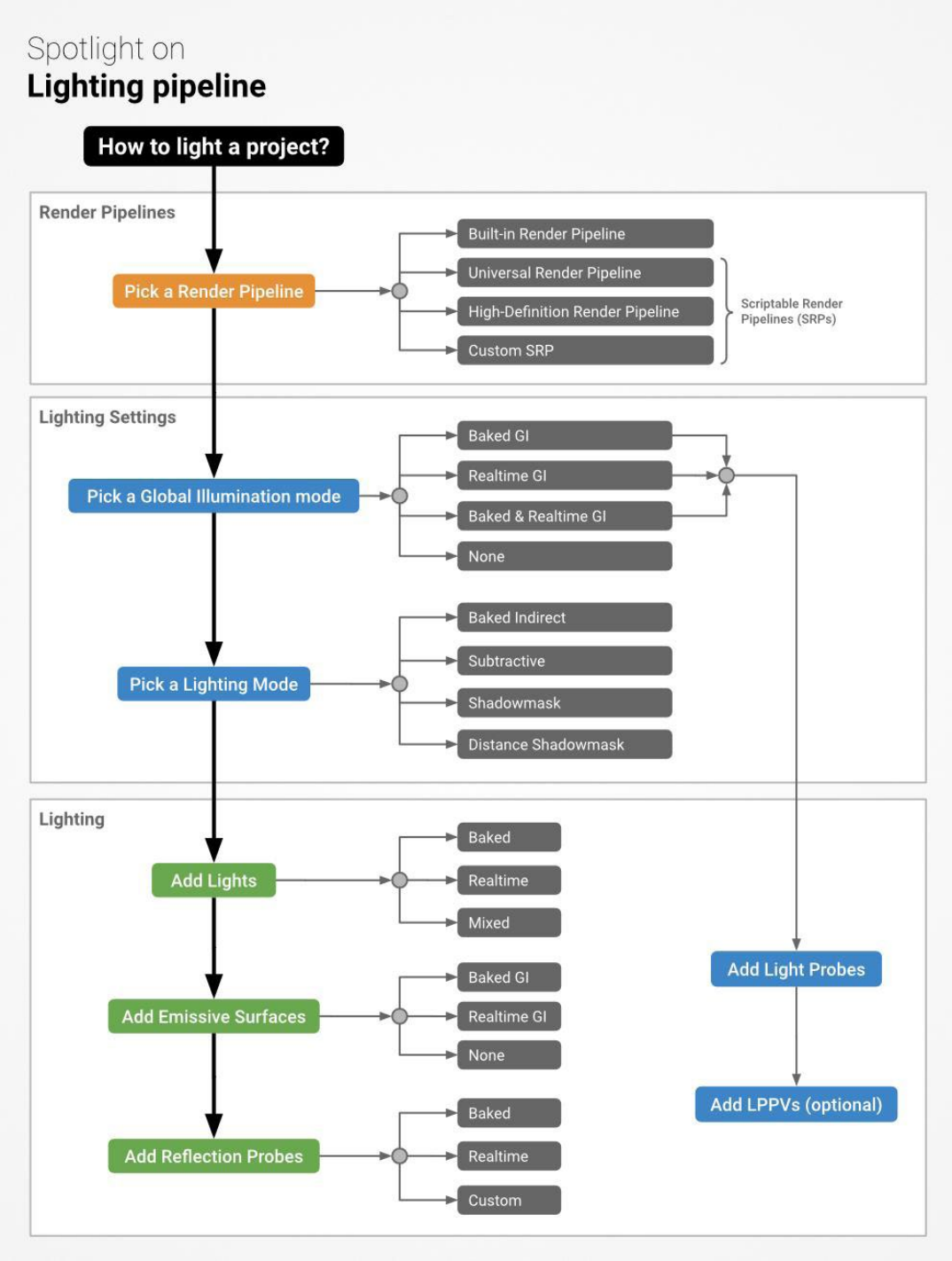
Unity の最新データによると、2023 年に PC とコンソール向けにリリースされた Unity 製ゲームでは、URP が最も選ばれていることがわかっています。以下のグラフは、ビルトインレンダーパイプラインが現在ごく一部の開発チームでのみ使用されていることを示しています。



Unity で利用可能な各パイプラインが使用されているゲームの割合

[2] HDRP は現在、モバイルプラットフォームではサポートされていません。詳細については、要件と互換性に関するページをご覧ください。

しかし、ほとんどの Unity プロジェクトが現在 URP または HDRP で構築されている一方で、ビルトインレンダーパイプライン（リストの 3 番目のオプションを参照）は、Unity 6 でも利用可能なオプションとして残る予定です。



プロジェクトを計画する際は、早めにレンダーパイプラインを決めておく。

Unity が作成したデモ『*Enemies*』は、HDRP のハイエンドのグラフィックス機能を披露してる。

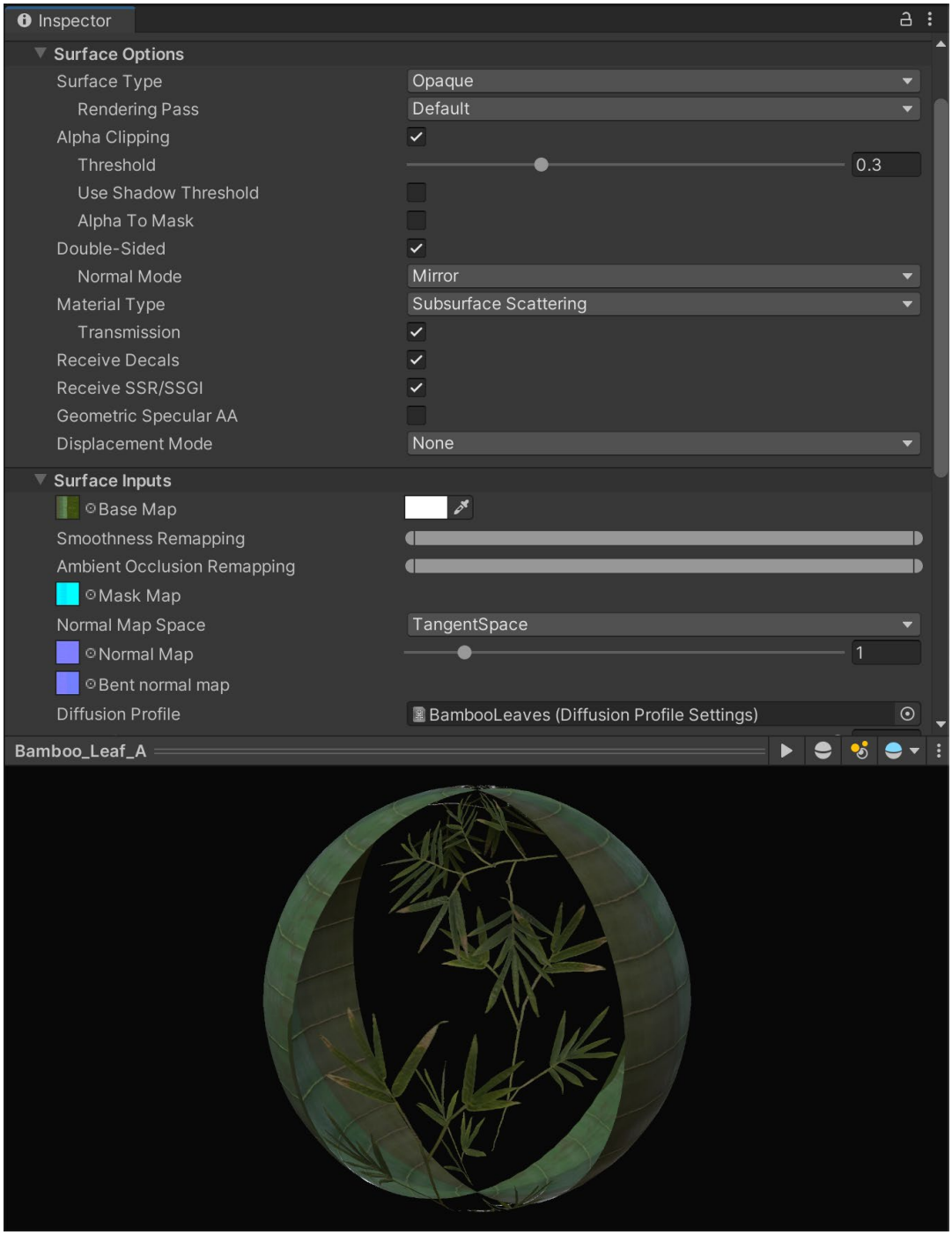
URP と HDRP は、スクリプタブルレンダーパイプライン（SRP）を時間として動作します。これは、C# スクリプトを使用してレンダリングコマンドをスケジュールおよび設定できる薄い API レイヤーです。この柔軟性により、パイプラインのほぼすべての部分をカスタマイズできます。また、SRP を基盤としてカスタムのレンダーパイプラインを作成することも可能です。

利用可能なパイプラインの詳細な比較については、「Unity のレンダーパイプライン」を参照してください。

コンソール向けのレンダーパイプラインパッケージ

PlayStation 4、**PlayStation 5**、**Game Core Xbox** 向けプロジェクトをビルドするには、サポートしたい各プラットフォーム向けに追加のパッケージをインストールする必要があります。各プラットフォーム向けパッケージは以下の通りです。

- **PlayStation 4：**com.unity.render-pipelines.ps4
- **PlayStation 5：**com.unity.render-pipelines.ps5
- **Xbox コンソール：**com.unity.render-pipelines.gamecore



肌や葉のようなマテリアルは、HDRP であらかじめ設定されている高度なライティングとシェーディング機能のメリットを享受できる。

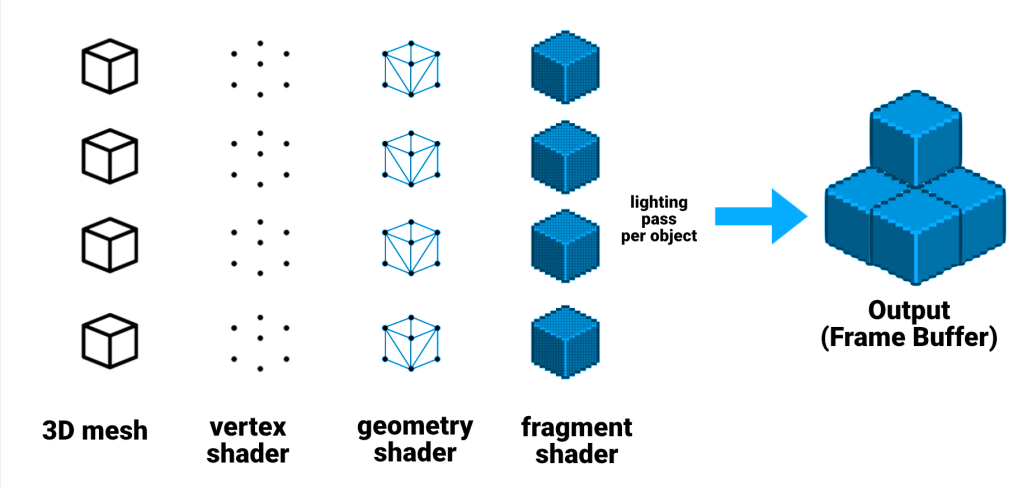
レンダリングパスを選択する

レンダーパイプラインを選択する際は、**レンダリングパス**についても考える必要があります。レンダリングパスは、ライティングとシェーディングに関連する、特定および一連の操作を表します。レンダリングパスの決定は、アプリケーションのニーズとターゲットハードウェアに依存します。

フォワード

フォワードレンダリングでは、グラフィックスカードがジオメトリを投影し、それを頂点に分割します。これらの頂点はさらにフラグメント（ピクセル）に分解され、最終的な画像を作成するために画面にレンダリングされます。

パイプラインは、各オブジェクトを 1 つずつグラフィックス API に渡します。フォワードレンダリングには、各ライトにコストがかかります。シーンにあるライトの数が多い程、レンダリング時間は長くなります。



フォワードレンダリングパス

ビルトインパイプラインのフォワードレンダラーは、オブジェクトごとに別々のパスで各ライトを描画します。同じゲームオブジェクトに複数のライトが当たっている場合、重複した領域が同じピクセルを 2 回以上描画する必要があり、大幅なオーバードローが発生する可能性があります。オーバードローを減らすには、リアルタイムライトの数を最小限に抑えてください。

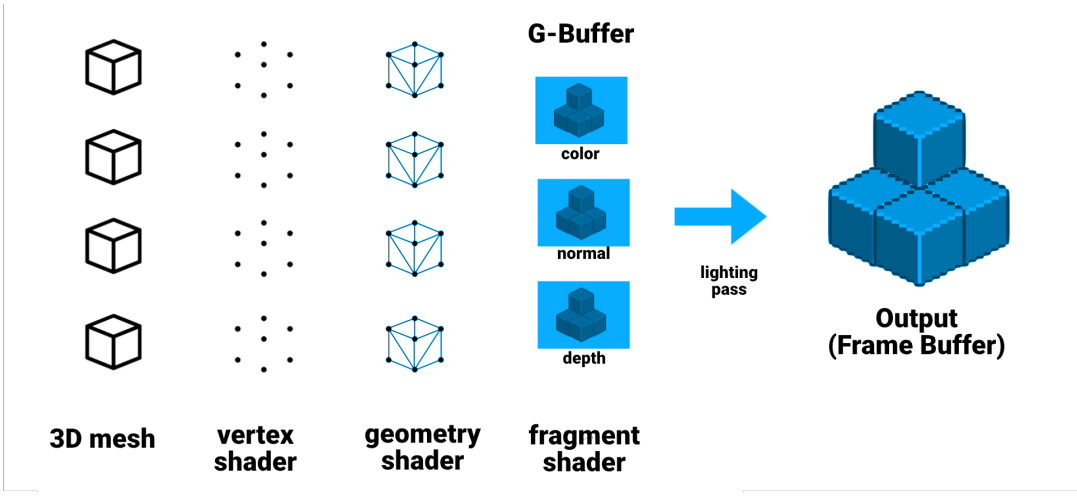
URP は、ライトごとに 1 つのパスをレンダリングするのではなく、オブジェクトごとにライトをカリングします。これにより、ライティングが 1 つのパスで計算されるため、ビルトインレンダーパイプラインのフォワードレンダラーに比べてドローコールが少なくなります。

フォワード +

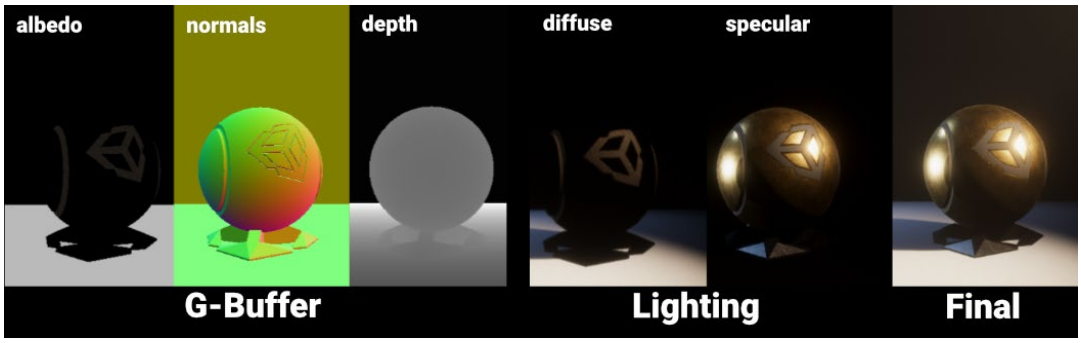
フォワード + レンダリングは、オブジェクトごとではなく空間的にライトをカリングすることで、標準のフォワードレンダリングを改善します。これにより、フレームのレンダリング時に利用できるライト全体の数が増加します。ディファードレンダリングでは、Native RenderPass API をサポートしており、G バッファとライティングパスが 1 つのレンダーパスにまとめられます。

ディファードシェーディング

ディファードシェーディングでは、ライティングはオブジェクトごとには計算されません。



ディファードシェーディングパス



ディファードシェーディングは、各オブジェクトの代わりにバッファにライティングを適用する。

その代わりに、ライティングの計算を後回しにします。ディファードシェーディングは 2 つのパスを使います。

1 つ目のパスである G バッファジオメトリパスでは、Unity がゲームオブジェクトをレンダリングします。このパスでは、数種類の幾何学的プロパティを取得し、テクスチャのセットに格納します。G バッファテクスチャには、以下が含まれます。

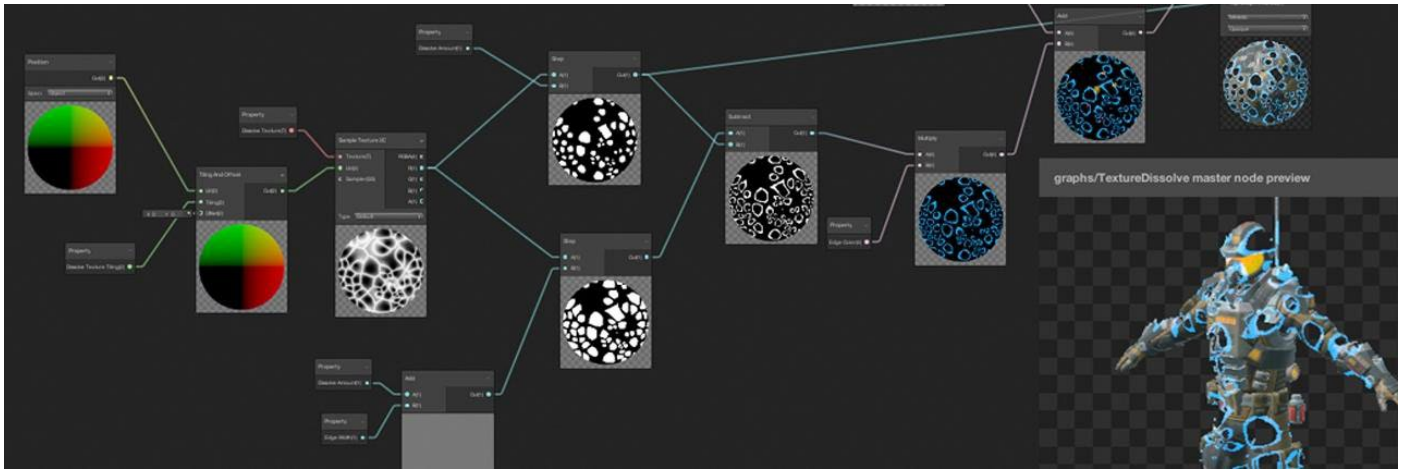
- ディフューズカラーとスペキュラカラー
- 表面の滑らかさ
- オクルージョン
- ワールド空間法線
- 放出 + アンビエント + リフレクション + ライトマップ

2 つ目のパスのライティングパスでは、Unity が G バッファに基づいてシーンのライティングをレンダリングします。各ピクセルを繰り返し処理し、個々のオブジェクトではなくバッファに基づいてライティング情報を計算する様子を想像してください。その結果、ディファードシェーディングで影を落とさないライトを追加しても、フォワードレンダリングと同じようなパフォーマンスヒットは発生しません。

レンダリングパスを選択すること自体は最適化ではありませんが、プロジェクト最適化の方法に影響を与える可能性があります。このセクションの他のテクニックとワークフローは、選択したレンダーパイプラインとレンダリングパスによって異なる場合があります。

Shader Graph を最適化する

HDRP と URP はともに、シェーダー制作を行うビジュアルインターフェースである Shader Graph をサポートしています。これにより、以前は一部のユーザーにとっては手が届かなかったような複雑なシェーディング効果を作り出せるようになりました。ビジュアルグラフシステムの 150 以上のノードを使用して、より多くのシェーダーを作成することが可能です。また、API を活用してカスタムノードを作成することもできます。



ビジュアルインターフェースを使用してシェーダーをビルドする。

各 Shader Graph は、グラフの出力を決定する互換性のあるマスターノードから始めてください。ビジュアルインターフェイスでノードと演算子を追加し、シェーダーロジックを構築します。

この Shader Graph は、その後レンダーパイプラインのバックエンドに渡されます。最終的な結果は ShaderLab シェーダーとなり、機能的には HLSL や Cg で書かれたものに似ています。

Shader Graph の最適化は、従来の HLSL/Cg シェーダーに適用されるものと同じルールの多くに従います。Shader Graph の処理が増えるほど、アプリケーションのパフォーマンスに影響します。

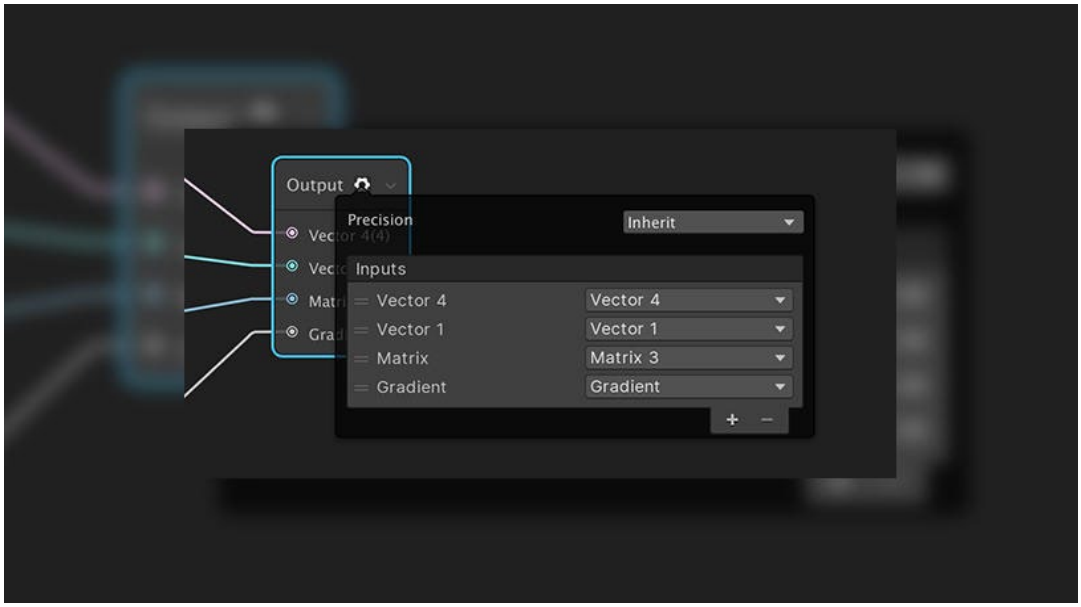
CPU 依存の場合、シェーダーを最適化してもフレームレートは改善しませんが、モバイルプラットフォームのバッテリー寿命は改善するかもしれません。

GPU 依存の場合に Shader Graphs を使用してパフォーマンスを向上させるには、以下のガイドライに従ってください。

— **ノードを減らす：**未使用のノードを削除します。必要な変更以外は、デフォルトを変更したり、ノードを接続したりしないでください。Shader Graph は、未使用の機能を自動的にコンパイルします。

可能な場合は、値をテクスチャにベイクしてください。例えば、ノードを使用してテクスチャを明るくする代わりに、テクスチャアセット自体に明るさを加えてください。

— **小さいデータ形式を使用する：**可能な場合は、より小さいデータ構造体に切り替えます。プロジェクトに影響がなければ、Vector3 の代わりに Vector2 を使うことを検討してください。状況によって可能な場合は、精度を下げることを試してください（float の代わりに half を使うなど）。



可能な場合は、Output ノードの Shader Graph の精度を下げる。

— **演算を減らす：**シェーダー演算は 1 秒間に何度も実行されるため、可能な限り演算子を最適化します。ロジックブランチを作成する代わりに、結果をブレンドすることを試してみてください。定数を使用し、ベクトルを適用する前にスカラー値を組み合わせてください。最後に、Inspector に表示する必要のないプロパティをインラインノードに変換します。これらの小さな速度の向上が、フレーム予算の改善に役立ちます。

— **プレビューを分岐する：**グラフが大きくなると、コンパイルが遅くなることがあります。ワークフローを簡素化するために、今プレビューに出したい操作だけを含む、独立した小さなブランチを作成します。そして、目的の結果が得られるまで、この小さなブランチで素早く反復作業を行います。

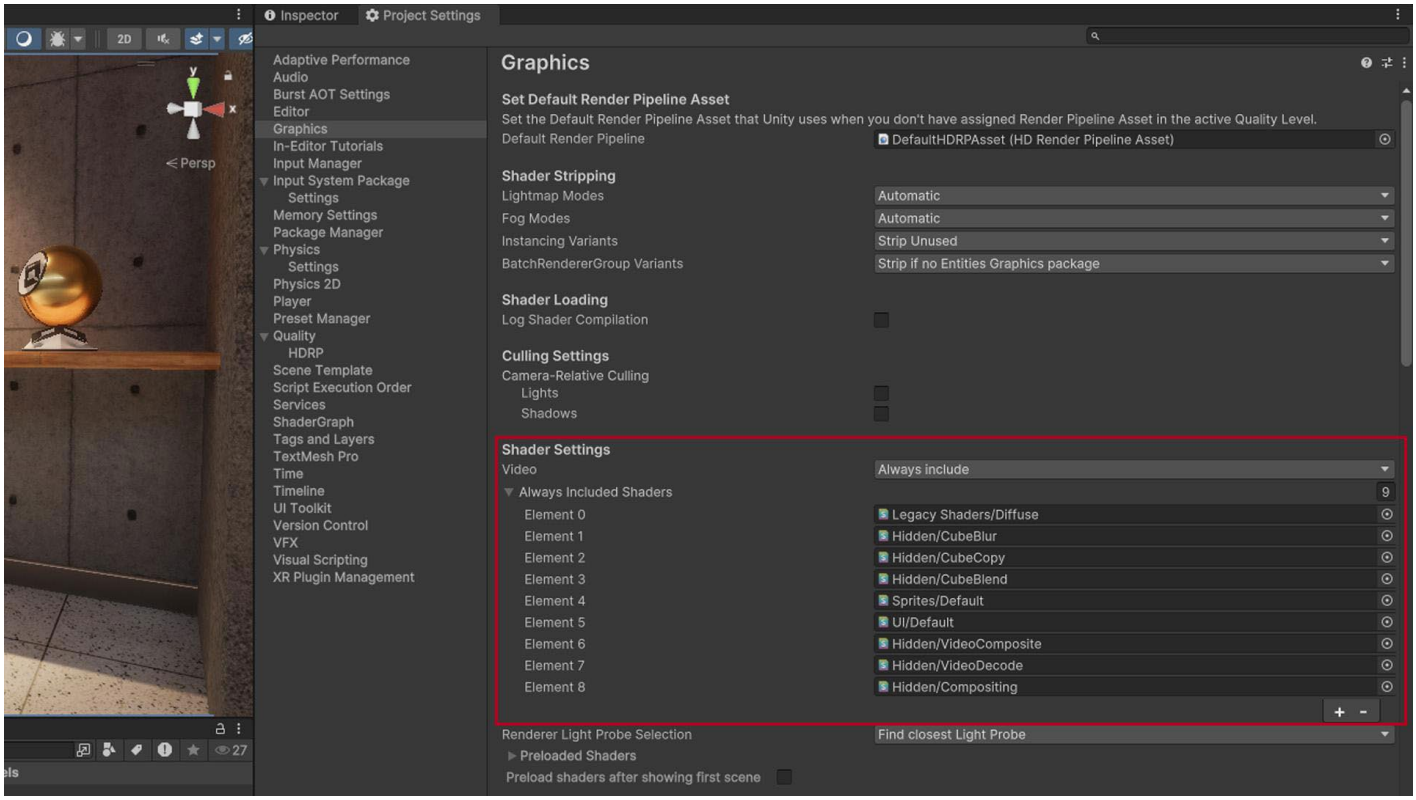
ブランチがマスターノードに接続されていない場合は、プレビューブランチをグラフに残しておくと安全です。Unity は、コンパイル時に最終出力に影響しないノードを削除します。

— **手動で最適化する:** どれだけ経験豊富なグラフィックスプログラマーでも、スクリプトベースのシェーダーの定型コードを Shader Graph で手軽に作成できるに越したことはないでしょう。Shader Graph アセットを選択し、コンテキストメニューから「Copy Shader」を選択します。

新しい HLSL/Cg シェーダーを作成し、コピーしたシェーダーグラフを貼り付けます。これは一方向の処理ですが、手動の最適化によってさらなるパフォーマンスを引き出すことができます。

ビルトインシェーダー設定を削除する

使用しないすべてのシェーダーを Graphics 設定（**「Edit > ProjectSettings」>「Graphics」**）にある「**Always Included**」のリストから削除します。アプリケーションのライフタイムで必要なシェーダーをここに追加してください。



Always Included Shaders

シェーダーバリアントを削る

シェーダーバリアントはプラットフォーム固有の機能によっては有用ですが、ビルド時間やファイルサイズを増加させます。

シェーダーコンパイルのプラグマディレクティブを使用することで、ターゲットプラットフォームごとに異なるシェーダーをコンパイルすることができます。その後、シェーダーキーワード（または Shader Graph Keyword ノード）を使用し、特定の機能が有効または無効のシェーダーバリアントを作成します。

シェーダーバリアントが必要ないとわかっている場合は、ビルドに含めないようにすることができます。

Editor.log を解析してシェーダー時間とサイズを確認します。「Compiled shader」と「Compressed shader」で始まる行を探します。

以下は、TEST シェーダーが表示するログの例です。

```
Compiled shader 'TEST Standard (Specular setup)' in 31.23s
   d3d9 (total internal programs:482, unique:474)
   d3d11 (total internal programs:482, unique:466)
   metal (total internal programs:482, unique:480)
   glcore (total internal programs:482, unique:454)
Compressed shader 'TEST Standard (Specular setup)' on d3d9 from 1.04MB to 0.14MB
Compressed shader 'TEST Standard (Specular setup)' on d3d11 from 1.39MB to 0.12MB
Compressed shader 'TEST Standard (Specular setup)' on metal from 2.56MB to 0.20MB
Compressed shader 'TEST Standard (Specular setup)' on glcore from 2.04MB to 0.15MB
```

このログは、このシェーダーに関して以下のことを示しています。

- このシェーダーは、#pragma multi_compile と shader_feature により 482 のバリアントに展開される。
- Unity は、ゲームデータに含まれるシェーダーを圧縮し、そのサイズはおおよそ圧縮後の各シェーダーのサイズの合計になる（0.14+0.12+0.20+0.15 = 0.61MB）。
- Unity は、ランタイム時、現在使用しているグラフィックス API のデータは非圧縮のままに、圧縮されたデータをメモリ（0.61MB）に保持する。例えば、現在の API が Metal の場合、2.56MB となる。

ビルド後、Project Auditor は Editor.log を解析し、プロジェクトにコンパイルされたすべてのシェーダー、シェーダーキーワード、シェーダーバリアントのリストを表示することができます。また、ゲームが実行されると Player.log も解析します。これにより、アプリケーションがラインタイム時に実際にコンパイルして使用したバリアントを確認できます。

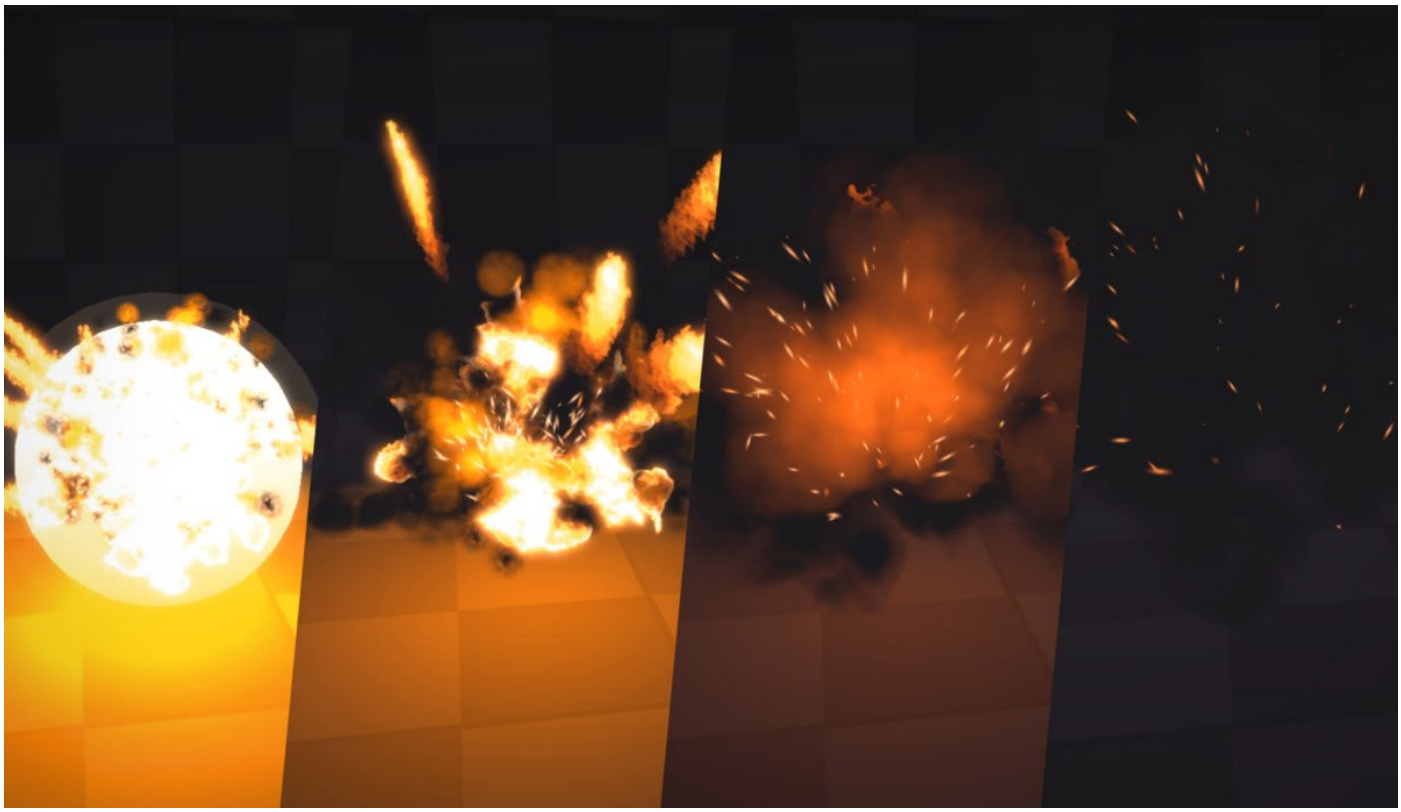
スクリプト可能なシェーダーストリッピングシステムをビルドし、バリアントの数を減らすために、この情報を使用してください。これにより、ビルド時間、ビルドサイズ、ランタイム中のメモリ使用量を改善することができます。

このプロセスに関する詳細は、ブログ記事「スクリプタブルシェーダーバリアントの除去」をご覧ください。

パーティクルシミュレーション：Particle System と VFX Graph

Unity 6 には、煙、液体、炎、その他のエフェクトを扱う 2 つのパーティクルシミュレーションソリューションが含まれています。

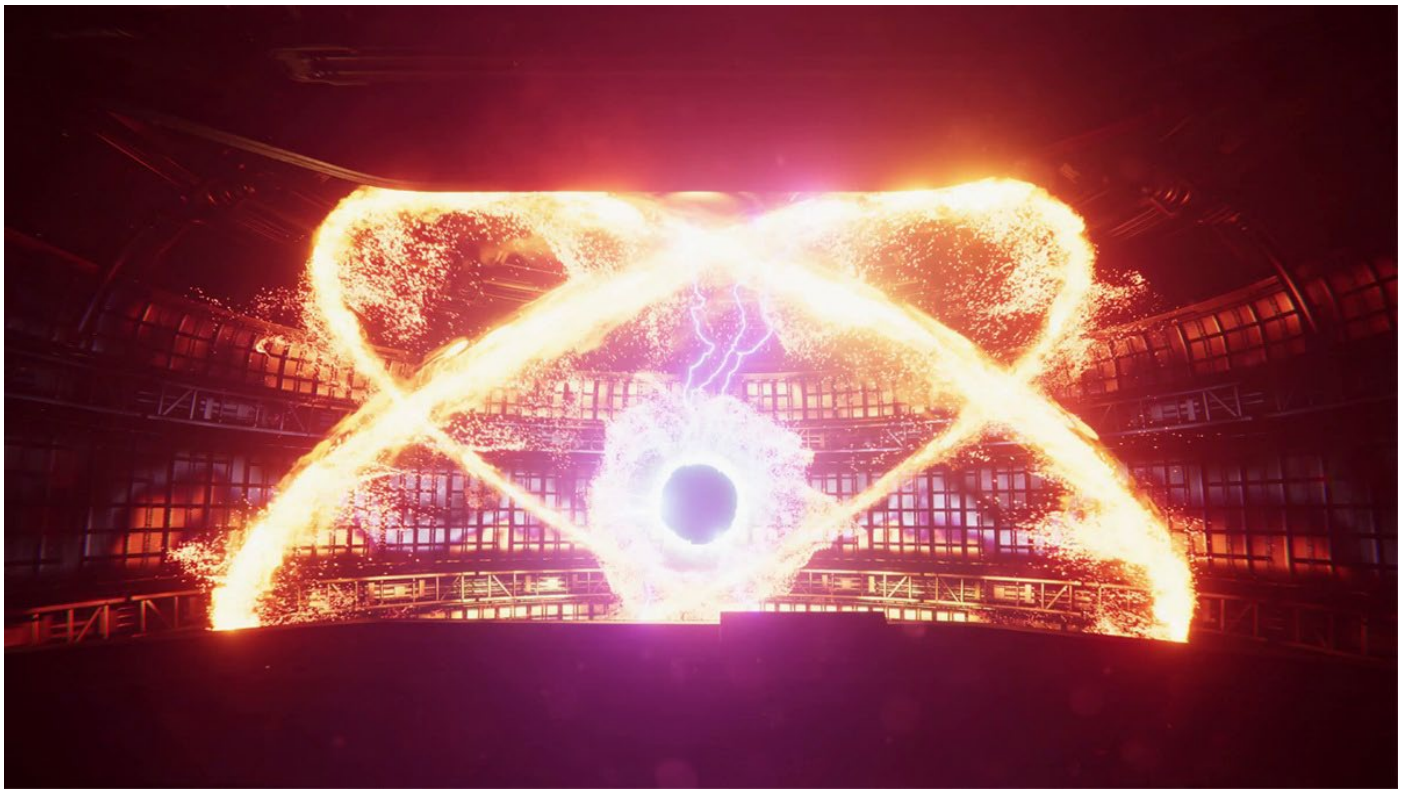
— ビルトインパーティクルシステムは、CPU で何千ものパーティクルを再現できます。C# スクリプトを使用してシステムと各パーティクルを定義します。パーティクルシステムは、Unity の物理システムやシーン内のコライダーと相互作用します。パーティクルシステムは最大限の互換性を提供し、Unity がサポートするいずれのビルドプラットフォームでも動作します。



ビルトインパーティクルシステムを使った簡単なエフェクトシミュレーション

— VFX Graph は、Unity で高度なビジュアルエフェクトを作成するための強化された機能を提供する新しいシステムで、特にハイエンドのグラフィックスとパフォーマンスをターゲットとするプロジェクト向けです。コンピュートシェーダーを使って GPU 上で計算を行い、大規模な視覚効果で何百万ものパーティクルを実現します。ワークフローには、カスタマイズ性が非常に高いグラフビューが含まれています。また、パーティクルは色および深度バッファと相互作用することができます。

VFX Graph は、基礎となる物理システムにはアクセスできませんが、Point Cache、Vector Field、Signed Distance Field などの複雑なアセットと相互作用できます。VFX Graph は HDRP と URP の両方と、コンピュートシェーダーをサポートするプラットフォームで動作します。



Visual Effect Graph で作成された画面上の数百万ものパーティクル

2 つのシステムのどちらかを選択する際には、デバイスの互換性を念頭に置いてください。ほとんどの PC やコンソールはコンピュートシェーダーをサポートしていますが、多くのモバイルデバイスはサポートしていません。ターゲットプラットフォームがコンピュートシェーダーをサポートしていない場合、Unity ではプロジェクトで両方のタイプのパーティクルシミュレーションを使用することが可能です。

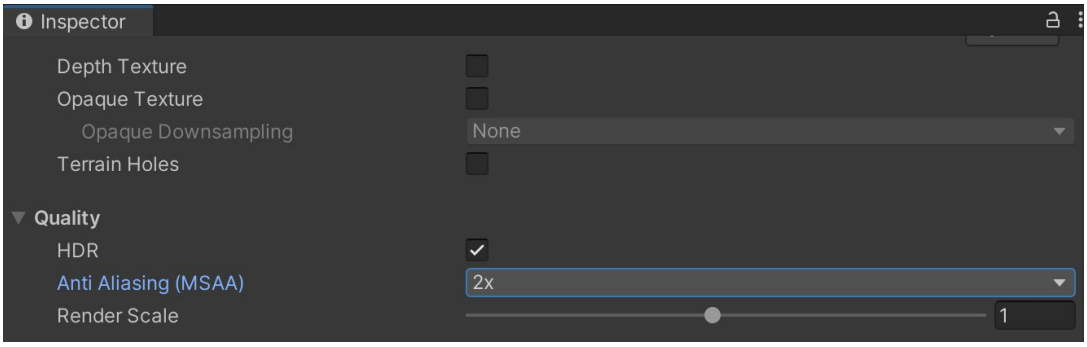
ハイエンドなビジュアルエフェクトを作成する方法についてさらに知りたい場合は、eBook「*Unity で高度なビジュアルエフェクトを作成するための決定版ガイド*」をご覧ください。

アンチエイリアスで滑らかにする

アンチエイリアスは、画像を滑らかにし、エッジのギザギザを減らし、スペキュラーエイリアスを最小限に抑えます。

ビルトインレンダーパイプラインでフォワードレンダリングを使用しているなら、Quality 設定で Multisample Anti-aliasing（MSAA）を利用することができます。MSAA は高品質のアンチエイリアシングを実現しますが、高い負荷がかかります。「**MSAA Sample Count**」のドロップダウンメニュー（None、2x、4x、8x）は、レンダラーがエフェクトを評価するために使用するサンプル数を定義します。

URP または HDRP でフォワードレンダリングを使用している場合、レンダーパイプラインアセットで MSAA を有効にできます。



URP で、Render Pipeline Asset の MSAA 設定を見つける。

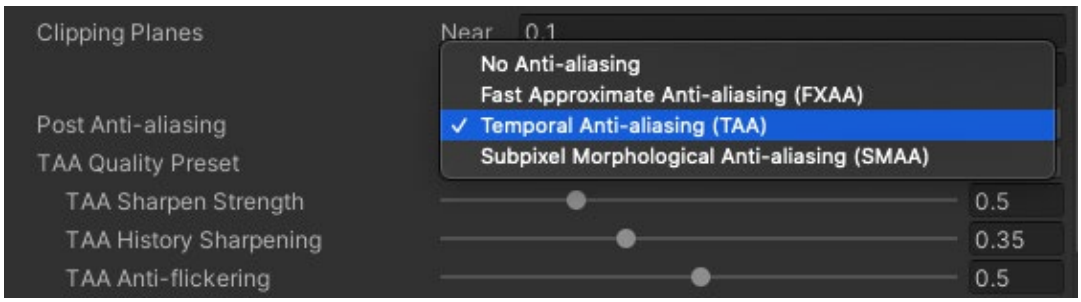
あるいは、後処理としてアンチエイリアスを加えることもできます。これは、Camera コンポーネントの「**Anti-aliasing**」に表示されます。

— **Fast Approximate Anti-aliasing（FXAA）**は、ピクセル単位でエッジを滑らかにします。これは最もリソースの消費を抑えることができるアンチエイリアス処理で、最終的な画像をわずかにぼかします。

— **Subpixel Morphological Anti-aliasing（SMAA）**は、画像の境界に基づいてピクセルをブレンドします。これは、FXAA よりもはるかにシャープな結果が得られ、フラットスタイルやカートゥーン調のスタイル、クリーンなアートスタイルに適しています。

HDRP では、カメラの「**Post Anti-aliasing**」設定で FXAA と SMAA を使用することもできます。URP と HDRP には追加のオプションも用意されています。

— **Temporal Anti-aliasing（TAA）**は、履歴バッファのフレームを使用してエッジを滑らかにします。これは、FXAA よりも効果的に機能しますが、そのためには、モーションベクトルが必要となります。TAA はアンビエントオクルージョンとボリューメトリックも改善できます。一般的に FXAA よりも高品質ですが、リソースを多く消費し、時折ゴーストアーティファクトが発生することがあります。



HDRP カメラのポストアンチエイリアス効果として機能する TAA

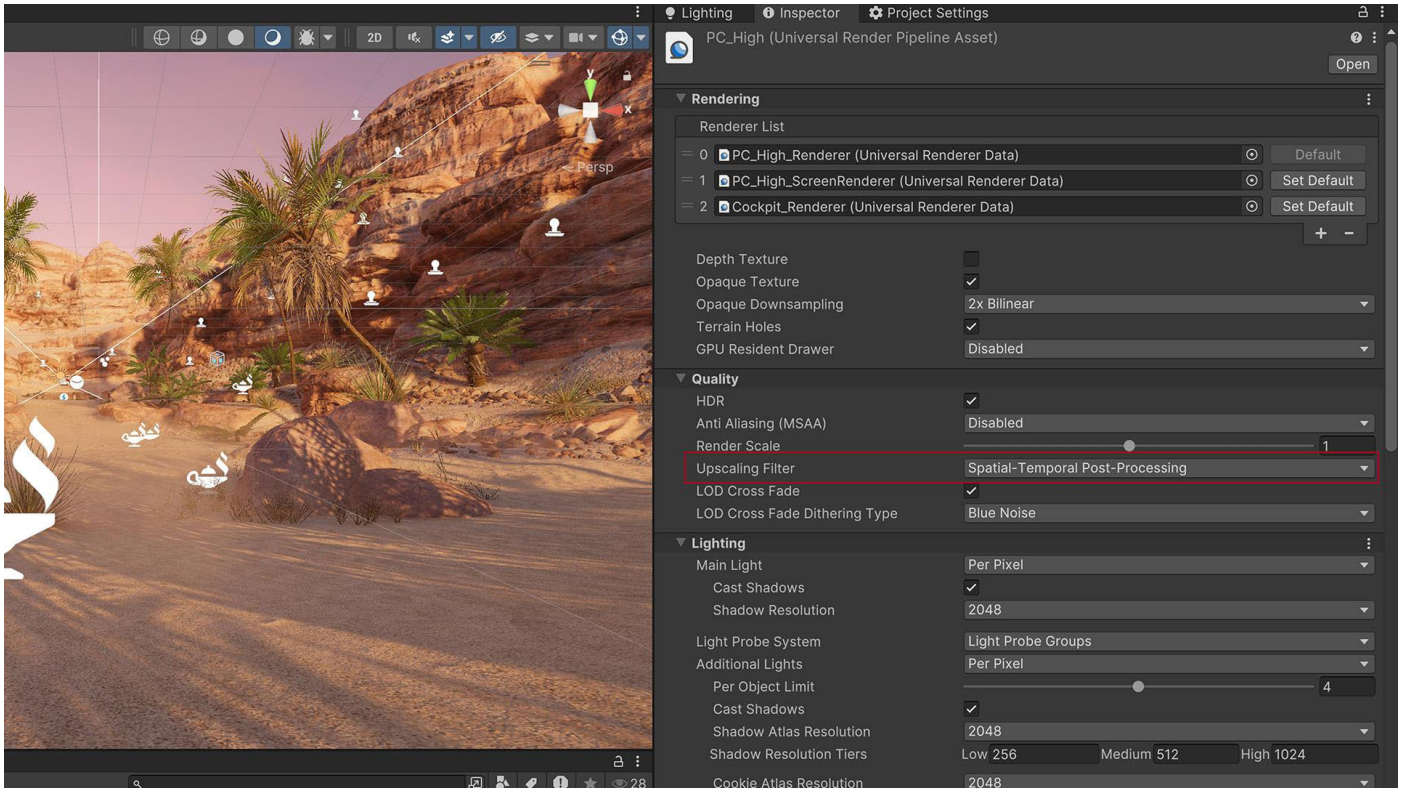
Spatial-Temporal Post-Processing（STP）

Spatial-Temporal Post-Processing（STP）は、モバイル、コンソール、PC などの幅広いプラットフォームにおいて、ビジュアル品質を向上させるように設計されています。STP は、HDRP と URP の両方のレンダーパイプラインで動作する、時空間的アンチエイリアシングアップスケーラーで、既存のコンテンツに変更を加えることなく、高品質のコンテンツスケーリングを提供します。このソリューションは特に GPU のパフォーマンスに最適化されており、レンダリング時間の短縮を保証し、ビジュアル品質を維持しながら高性能の実現を容易にします。

URP で STP を有効にする方法は、以下の通りです。

— Project ウィンドウで、アクティブな URP Asset を選択します。

— Inspector で「Quality」>「Upscaling Filter」に移動し、「Spatial-Temporal Post-Processing」を選択します。



URP Asset 内で STP を有効化

一般的なライティングの最適化

ライティングは非常に広範なテーマですが、これらの一般的なヒントはリソースの最適化に役立ちます。

ライトマップをベイクする

ライティングを作成する最速のオプションは、フレームごとに計算する必要がないものです。これをするには、静的ライティングをリアルタイムで計算する代わりに、Lightmapping を使用してを一度「ベイク」します。

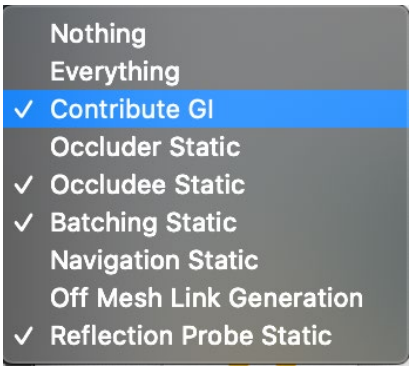
Global Illumination（GI）を使用してドラマティックライティングを静的ジオメトリに追加します。「**Contribute GI**」でオブジェクトをマークし、ライトマップの形式で高品質のライティングを保管できるようにします。

ライトマップ環境を生成するプロセスは、Unity でシーンにライトを配置するだけの場合より時間がかかりますが、以下のメリットがあります。

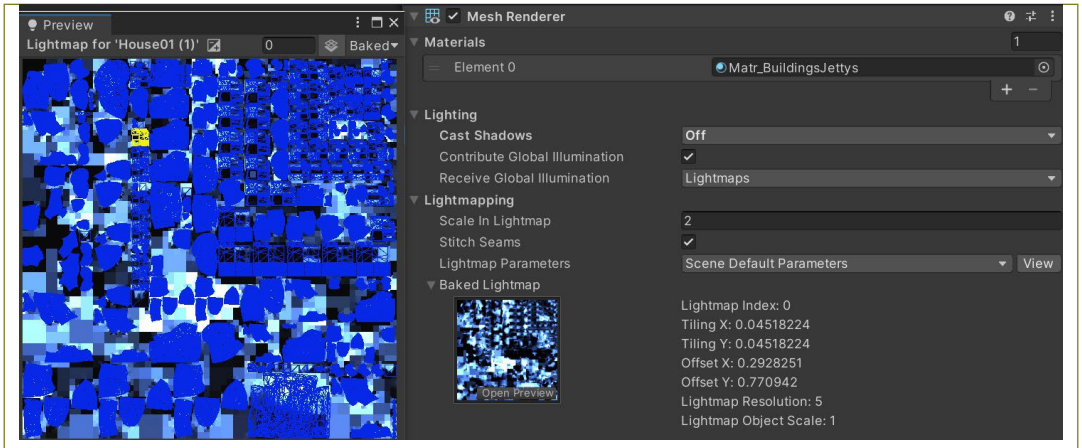
- 処理速度が速くなり、特に 1 つのピクセルにライトが 2 つある場合では 2 〜 3 倍速い。
- GI はリアルな直接光と間接光を計算するため、見栄えが良くなる。ライトマッパーは、得られたマップを平滑化し、ノイズ除去する。

ベイクした影とライティングは、リアルタイムで生成した場合に発生するパフォーマンス上の打撃を受けることなくレンダリングできます。

シーンが複雑な場合は、ベイク時間が長くなることがあります。お使いのハードウェアが**プログレッシブ GPU ライトマッパー**をサポートしている場合、このオプションを使うと、ライトマップの生成が劇的に速くなり、場合によっては 10 倍にもなります。



Contribute GI を有効にする。



ライトマッピングの設定（「**Windows**」>「**Rendering**」>「**Lighting Settings**」）とライトマップのサイズを調整してメモリ使用量を抑える。

マニュアルに従い、Unity でライトマップを活用する方法を学びましょう。

リフレクションプローブを最小限に抑える

Reflection Probe コンポーネントはリアルなリフレクションを作成しますが、バッチの面で大幅なコストがかかります。ランタイムパフォーマンスを向上させるために、低解像度のキューブマップ、カリングマスク、テクスチャ圧縮を使用してください。「Type」に「**Baked**」を使用してフレームごとの更新を避けてください。

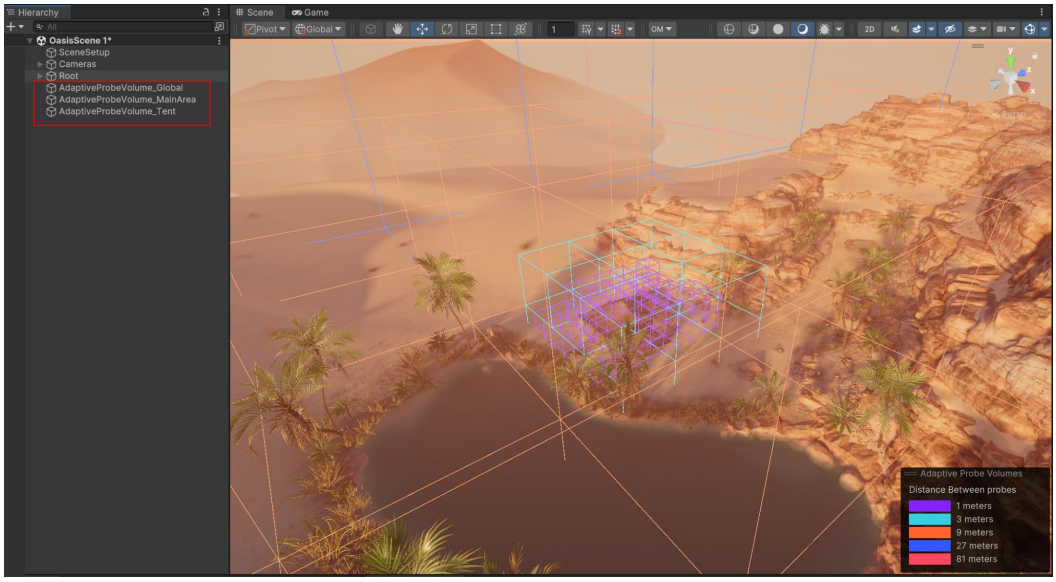
URP で「**Type**」に「**Realtime**」を使用することが必須の場合は、可能な限り「**Every Frame**」を避けるようにしてください。「Refresh Mode」と「Time Slicing」の設定を調整して更新レートを削減しましょう。また、「Via Scripting」オプションを使用してリフレッシュをコントロールし、カスタムスクリプトからプローブをレンダリングすることもできます。

HDRP で「**Type**」に「**Realtime**」を使用することが必須の場合は、「**On Demand**」モードを使用してください。また、**「Project Settings」>「HDRP Default Settings」**から「Frame Settings」を変更することも可能です。「Realtime Reflection」で品質と機能を落とすことで、パフォーマンスを向上させことができます。

アダプティブプローブボリューム

Unity 6 では、アダプティブプローブボリューム（APV）が導入され、Unity でグローバルイルミネーションを処理するための洗練されたソリューションが提供され、複雑なシーンで動的かつ効率的なライティングを実現できるようになりました。APV は、特にモバイルやローエンドのデバイスにおいて、パフォーマンスとビジュアル品質の両方を最適化することができると同時に、ハイエンドのプラットフォームに対しては高度な機能を提供することができます。

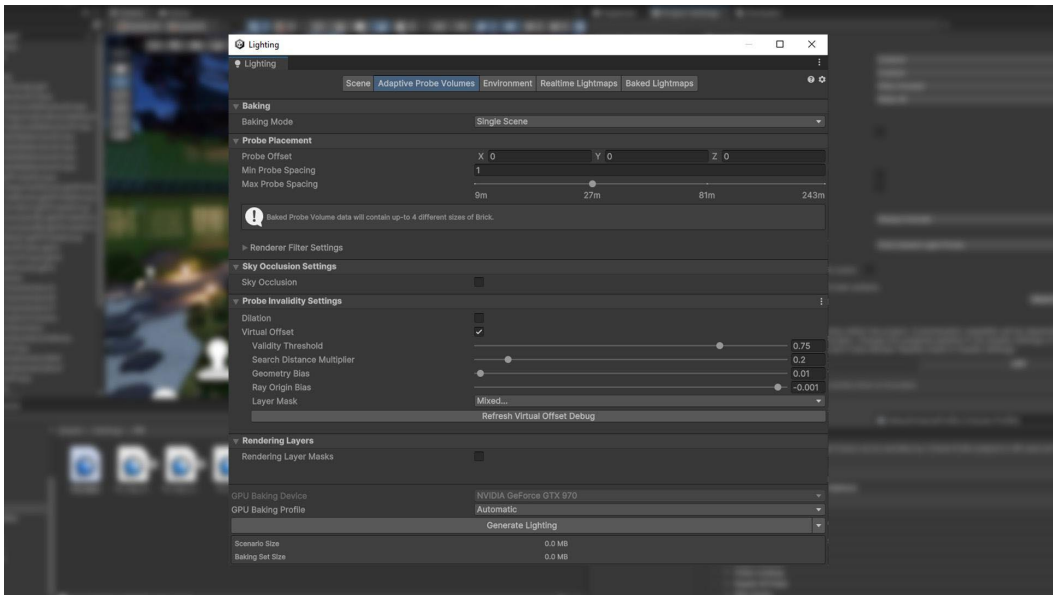
APV は、特に規模が大きい動的なシーンにおいて、グローバルイルミネーションを強化するためのさまざまな機能を提供します。URP は、プローブボリュームにベイクされた間接光のメリットを享受しがら、ローエンドデバイスでのパフォーマンスを向上させるために、頂点ごとのサンプリングをサポートするようになりました。



複数の APV を持つことは、レベル内のプローブ密度をよりコントロールするのに役立つ。

APV データはディスクから CPU や GPU にストリーミングすることができ、大規模な環境のライティング情報を最適化できます。複数のライティングシナリオをベイクしてブレンドし、昼夜のサイクルのようなリアルタイムの光の遷移を可能にします。さらに、このシステムはスカイオクルージョンをサポートし、Ray Intersector API と統合することでプローブ計算をより効率化し、また、ライトプローブのサンプル密度をコントロールすることで、光漏れを抑え、反復作業の速度を向上させることができます。新しい C# ベイク API は、ライトマップまたはリフレクションプローブからの APV の独立したベイクも可能にします。

APV の活用方法については、GDC 2023 の「Efficient and Impactful Lighting with Adaptive Probe Volumes」をご覧ください。

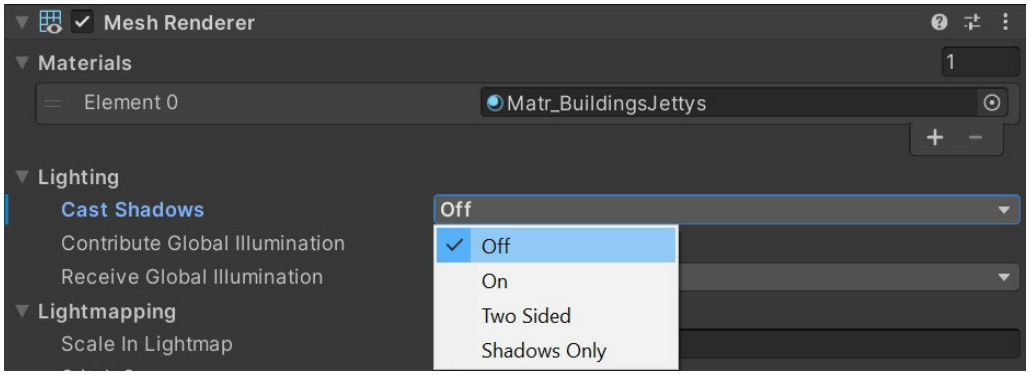


Lighting 設定にある Adaptive Probe Volumes ウィンドウ

影を無効にする

影のキャストは、MeshRenderer とライトごとに無効にできます。ドローコールを削減するためには、可能な限り影を無効にしてください。

また、シンプルなメッシュや四角形をキャラクターの下に配置し、ぼかしたテクスチャを適用することで、偽の影を作るという方法もあります。あるいは、カスタムシェーダーを使ってブロブシャドウを作ることができます。



影のキャストを無効にしてドローコールを削減する。

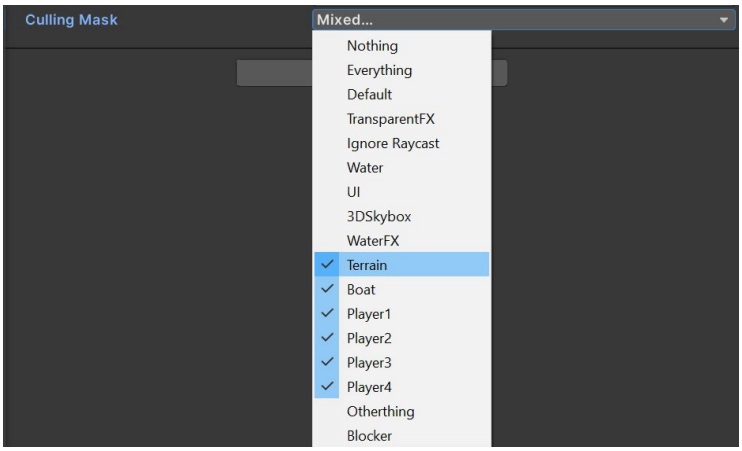
特に、ポイントライトの影を有効にするのは避けてください。影のあるポイントライトは、1 つのライトにつき 6 つのシャドウマップパスを必要とします。これを、スポットライトの 1 つのシャドウマップパスと比べてみてください。どうしても動的な影が必要な場合は、ポイントライトをスポットライトに置き換えることを検討してください。動的な影を使用しないで済む場合は、代わりにポイントライトと併せて Light.cookie としてキューブマップを使用しましょう。

シェーダーエフェクトを代替する

場合によっては、複数のライトを追加するのではなく、少しのコツで解決できることがあります。例えば、リムライティング効果を与えるためにカメラにまっすぐ光を当てるライトを作成する代わりに、リムライティングを模したシェーダーを使用します（HLSL でこれを導入する方法については、サーフェスシェーダーの例を参照してください）。

ライトレイヤーを使用する

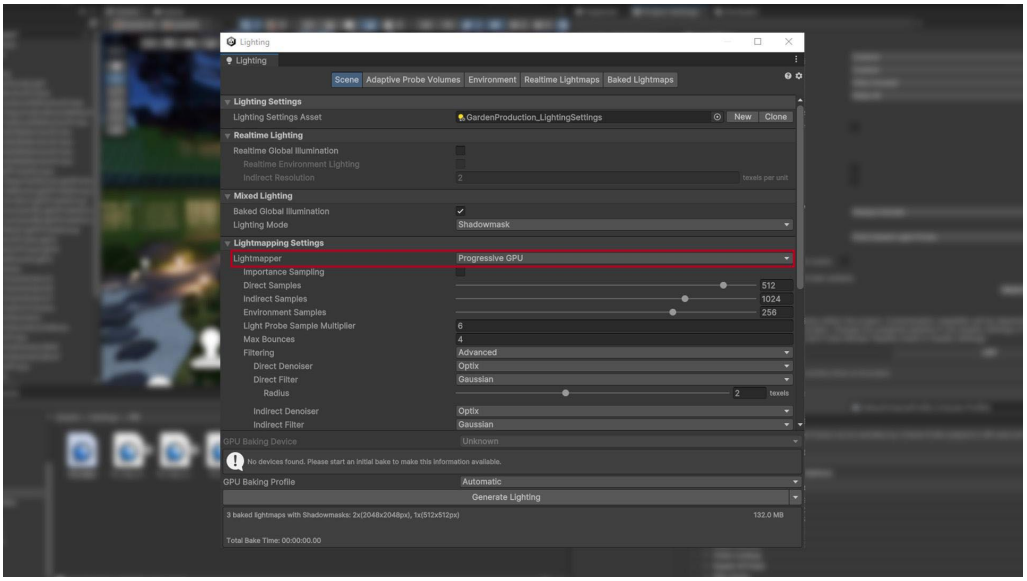
ライトが複数ある複雑なシーンでは、オブジェクトをレイヤー分けし、各ライトの影響を特定のカリングマスクに限定します。



レイヤーは、光の影響を特定のカリングマスクに制限できる。

GPU ライトマッパー

GPU ライトマッパーは Unity 6 で製品利用が可能です。GPU を活用することで、ライティングデータの生成を劇的に高速化し、従来の CPU によるライトマッピングに比べてベイク時間を大幅に短縮できます。コードベースを簡素化し、より予測可能な結果をもたらす新しいライトベイクのバックエンドを導入しています。さらに、GPU の最小要件が 2GB に引き下げられたほか、ランタイム時にライトプローブの位置を移動できる新しい API も追加されました。これは、プロシージャル生成されたコンテンツで特に有用で、さまざまな利便性向上の改善も含まれています。



GPU ライトマッパーを選択する様子

GPU の最適化

グラフィックスレンダリングを最適化するには、ターゲットハードウェアの制限と GPU のプロファイリング方法を理解する必要があります。プロファイリングは、あなたが行っている最適化が効果的であるかどうかを検証するのに役立ちます。

GPU のレンダリング負荷軽減のため、以下のベストプラクティスを活用しましょう。

GPU をベンチマークする

プロファイリングを行う場合、特定の GPU からどのようなプロファイリング結果が期待できるかを教えてくれるベンチマークから始めると良いでしょう。

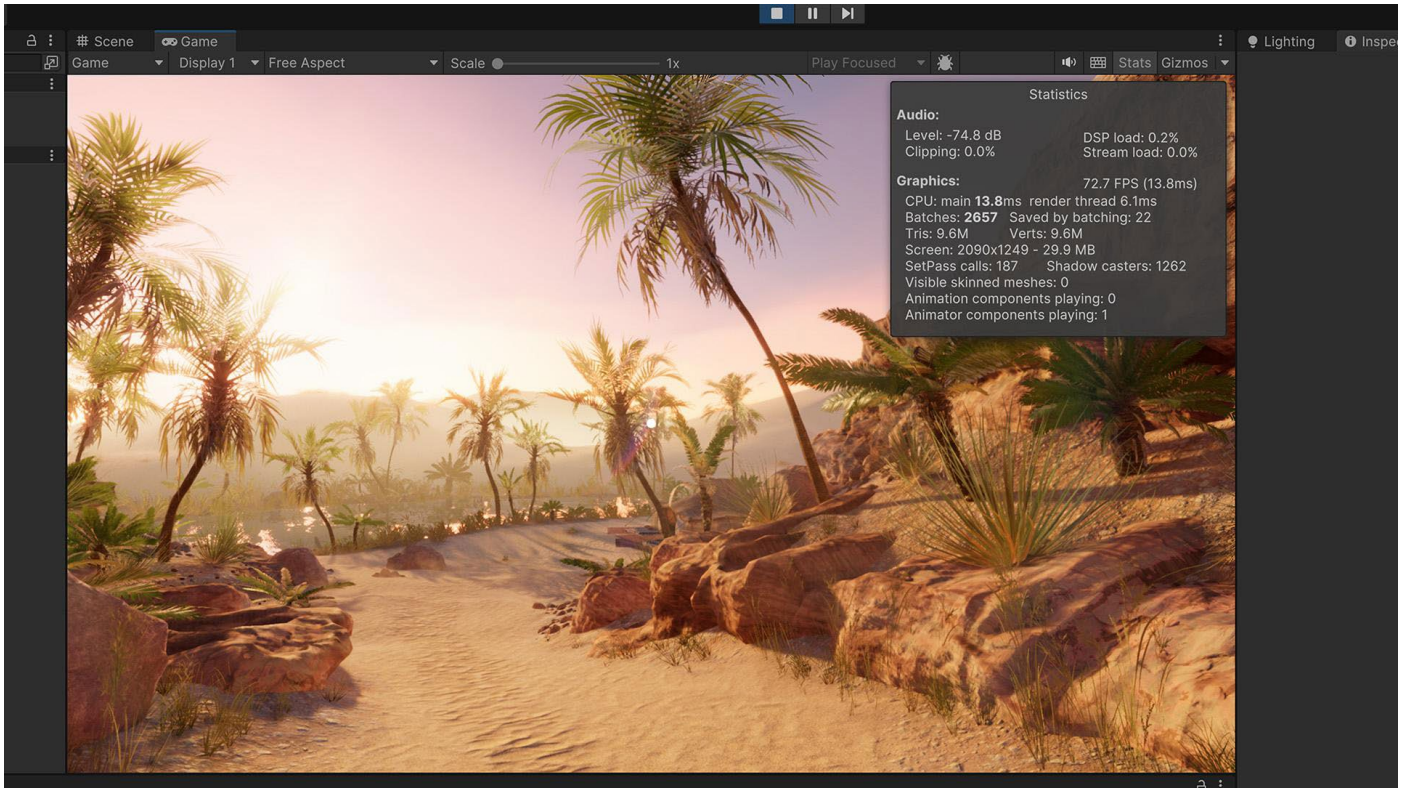
GPU とグラフィックスカードのさまざまな業界標準ベンチマークの一覧については、GFXBench を参照してください。このウェブサイトでは、現在利用可能な GPU の概要と、各 GPU の位置づけについて紹介しています。

レンダリング統計を確認する

ゲームビューの右上にある「**Stats**」ボタンをクリックします。すると、再生モード中のアプリケーションに関するリアルタイムのレンダリング情報が確認できるウィンドウが表示されます。このデータを用いて、次のパフォーマンスを最適化しましょう。

- **fps**：1 秒あたりのフレーム数
- **CPU Main**：1 フレームのレンダリング（および全ウィンドウのエディターの更新）に要する時間
- **CPU Render**：ゲームビューの 1 フレームのレンダリングに要する時間
- **Batches**：同時に描画されるドローコールのバッチ数
- **Tris（三角形）と Verts（頂点）**：メッシュのジオメトリ
- **SetPass calls**：Unity が画面上のゲームオブジェクトをレンダリングするためにシェーダーパスを切り替える回数（各パスは余分な CPU オーバーヘッドをもたらす可能性がある）

注：エディター内の fps が必ずしもビルドのパフォーマンスにつながるとは限りません。最も正確な結果を得るために、ビルドのプロファイリングを行うことをお勧めします。「1 秒あたりのフレーム数：まやかしの指標」セクションで前述した通り、ベンチマークを行う場合、ミリ秒単位のフレームタイムが 1 秒あたりのフレーム数よりも正確な指標となります。



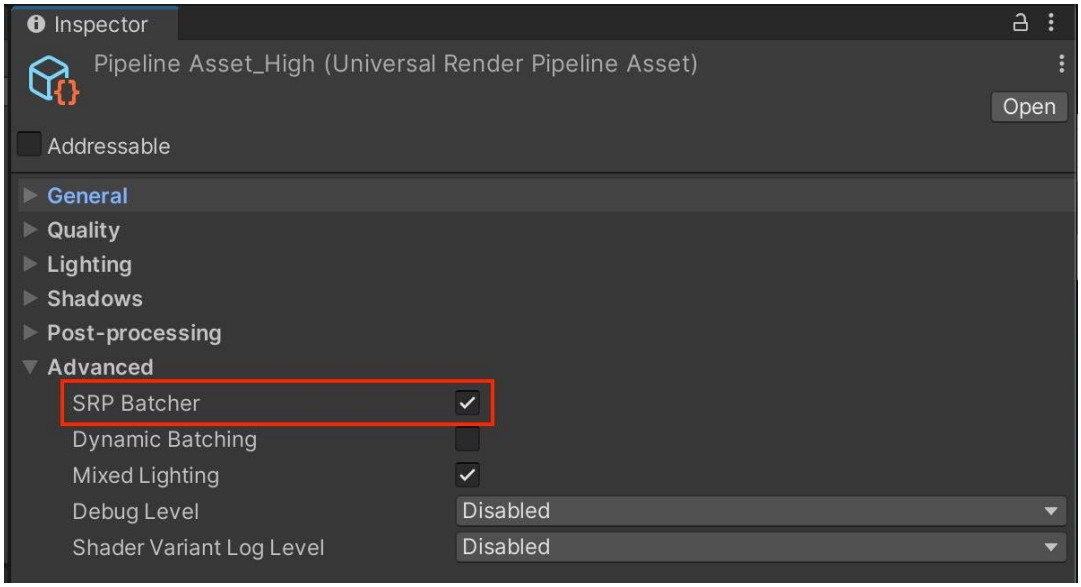
ドローコールをバッチ処理する

ゲームオブジェクトを描画するために、Unity はグラフィックス API（OpenGL、Vulkan、Direct3D など）にドローコールを発行します。各ドローコールはリソースを大量に消費します。マテリアルの切り替えなど、ドローコール間の状態変更は、CPU 側でパフォーマンスのオーバーヘッドを引き起こす可能性があります。

PC や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することができますが、各ドローコールのオーバーヘッドは、削減する努力が必要とされるほどまだ高いと言えます。モバイルデバイスでは、ドローコールの最適化が不可欠です。これは、ドローコールバッチ処理によって実現できます。

ドローコールバッチ処理は、このような状態変更を最小限に抑え、オブジェクトのレンダリングにかかる CPU コストを削減します。Unity では、いくつかのテクニックを使って、複数のオブジェクトをより少ないバッチにまとめることができます。

— **SRP バッチ処理**：HDRP または URP を使用している場合は、Pipeline Asset の「**Advanced**」にある「SRP Batcher」を有効にします。互換性のあるシェーダーを使用する場合、SRP Batcher はドローコール間の GPU セットアップを削減し、マテリアルデータを GPU メモリに永続的に保持します。これにより、CPU のレンダリング時間を大幅に抑えることができます。SRP バッチ処理をさらに改善するには、最小限のキーワードでより少ないシェーダーバリアントを使用します。このレンダリングワークフローをプロジェクトに活用する方法については、こちらの SRP ドキュメントを参照してください。



SRP Batcher はドローコールのバッチ処理に役立つ。

— **GPU インスタンシング**：同じオブジェクト（同じメッシュとマテリアルを持つ建物、木、草など）が多数ある場合は、GPU インスタンシングを使用します。このテクニックは、グラフィックスハードウェアを使ってバッチ処理を行うます。GPU インスタンシングを有効にするには、Project ウィンドウでマテリアルを選択し、Inspector で「**Enable Instancing**」をチェックします。

— **静的バッチ処理**：移動しないジオメトリの場合、Unity は同じマテリアルを共有するメッシュのドローコールを削減できます。動的バッチ処理よりも効率的ですが、より多くのメモリを使用します。

確実に動くことのないメッシュは、すべて Inspector で「**Batching Static**」とマークします。Unity は、ビルド時にすべての静的メッシュを 1 つの大きなメッシュに結合します。また、StaticBatchingUtility を使用すると、ランタイム時に（例えば、非可動部品のプロシージャレベルを生成した後で）このような静的バッチを自身で作成することもできます。

— **動的バッチ処理**：小さなメッシュの場合、Unity は CPU 上で頂点をグループ化して変換し、それらを一度に描画することができます。注：十分な数のローポリメッシュ（各メッシュの頂点が 300 以下、頂点アトリビュートの合計が 900 以下）がない限り、これを使用しないでください。そうでない場合は、バッチ処理を行うための小さなメッシュを探すことに CPU 時間を浪費することになります。

いくつかの簡単なルールで、バッチ処理を最大限に活用することができます。

— シーンで使用するテクスチャの数を可能な限り抑えます。テクスチャの数が少なければ、必要な固有のマテリアルも少なくなり、バッチ処理が容易になります。また、可能な限りテクスチャアトラスを使用してください。

— ライトマップは可能な限り常に最大のアトラスサイズでベイクしてください。ライトマップの数が少なければマテリアルの状態変更も少なくて済みますが、メモリフットプリントに注意してください。

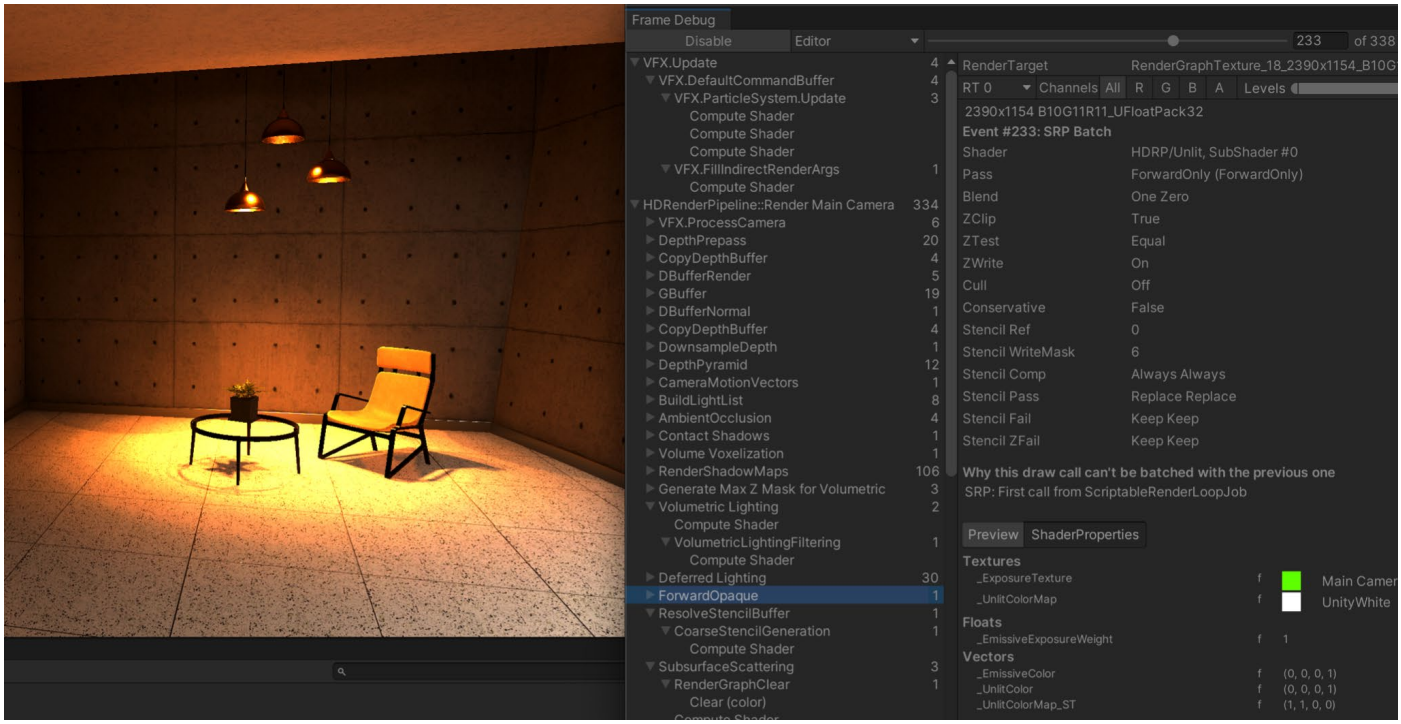
— 意図せずにマテリアルをインスタンス化しないように注意してください。スクリプトの Renderer.material にアクセスするとマテリアルを複製し、新しいコピーへのリファレンスを返します。これにより、すでにそのマテリアルを含む既存のバッチは破棄されます。バッチオブジェクトのマテリアルにアクセスしたい場合は、代わりに Renderer.sharedMaterial を使用してください。

— Profiler や最適化中のレンダリング統計を使って、静的および動的バッチカウント数とドローコールの総数を常に監視してください。

詳細については、ドキュメント「ドローコールバッチ処理」を参照してください。

フレームデバッガーを確認する

フレームデバッガーを使用すると、1 つのフレームで再生をフリーズし、Unity がシーンをどのように構築するかをステップごとに確認して、最適化の機会を特定できます。不必要にレンダリングしているゲームオブジェクトを探し、それらを無効にしてフレームごとのドローコールを減らしてください。



フレームデバッガーは、レンダリングされた各フレームを詳細に分析する。

注：フレームデバッガーは、個々のドローコールや状態変更を表示するものではありません。ネイティブの GPU プロファイラーだけが、詳細なドローコールとタイミング情報を提供します。しかし、フレームデバッガーは、パイプラインの問題やバッチ処理の問題をデバッグするのに非常に役立ちます。

Unity フレームデバッガーの利点のひとつは、ドローコールをシーン内の特定のゲームオブジェクトに関連付けできることです。これにより、外部フレームデバッガーでは不可能な特定の問題を調査しやすくなります。

詳細については、Frame Debugger のドキュメントをご覧ください。プラットフォーム固有のデバッグツールの一覧については、「ネイティブのプロファイリングおよびデバッグツールを使用する」のセクションをご覧ください。

フィルレートを最適化し、オーバードローを削減する

フィルレートとは、GPU が 1 秒間に画面にレンダリングできるピクセル数のことです。

あなたのゲームがフィルレートによって制限されている場合、これは GPU が処理できるよりも多くのピクセルをフレームごとに描画しようとしていることを意味します。

同じピクセルの上に何度も描画することをオーバードローと呼びます。オーバードローはフィルレートを低下させ、余分なメモリ帯域幅を消費します。オーバードローが発生するよくある要因は以下の通りです。

- 不透明または透明なジオメトリのオーバーラップ
- 複雑なシェーダー（多くの場合、複数のレンダーパスを伴う）
- 最適化されていないパーティクル
- オーバーラップする UI 要素

その影響を最小限に抑えることは重要ですが、オーバードローの問題を解決するための万能のアプローチはありません。まずは、上記の要因の影響を減らすことを試みましょう。

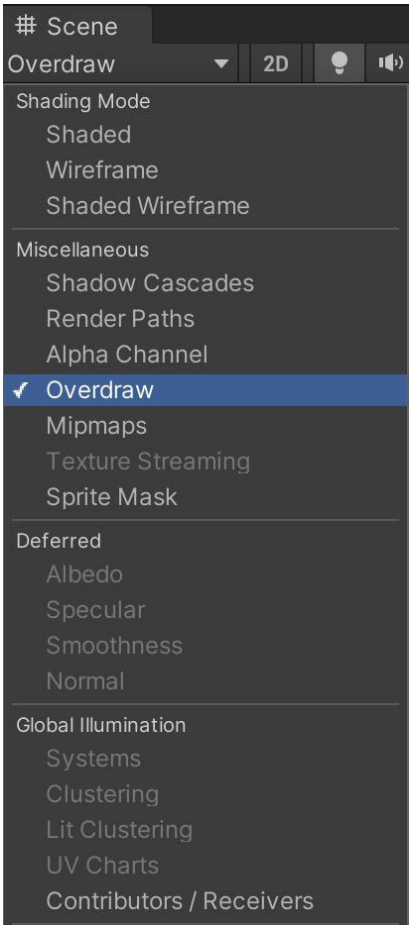
描画順とレンダーキュー

オーバードローに対処するには、Unity がオブジェクトをソートしてからレンダリングする方法を理解する必要があります。

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 renderQueue 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。

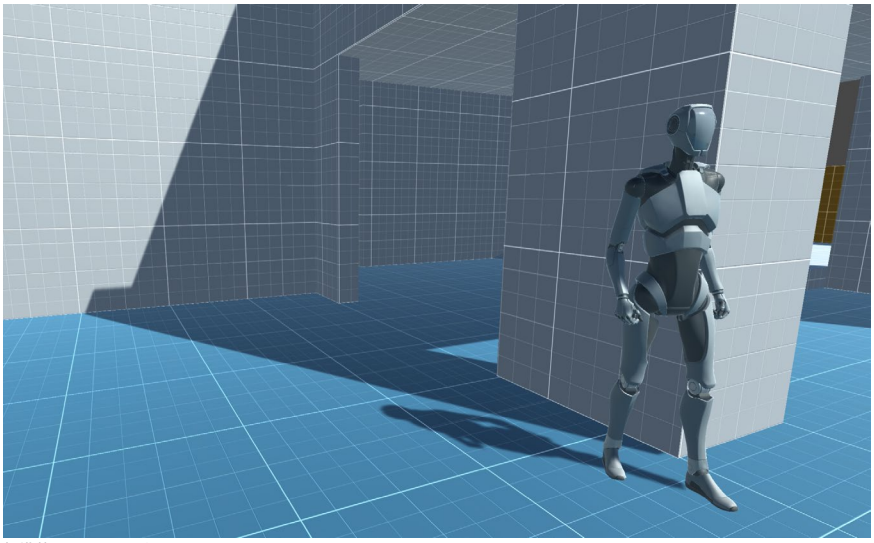
Unity が実際にオブジェクトを画面に描画する前のソートについては、レンダーキューごとに異なるルールに従うことがあります。例えば、Unity は Opaque Geometry キューを前から後ろにソートしますが、Transparent キューは前から後ろにソートします。

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。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、Scene 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。

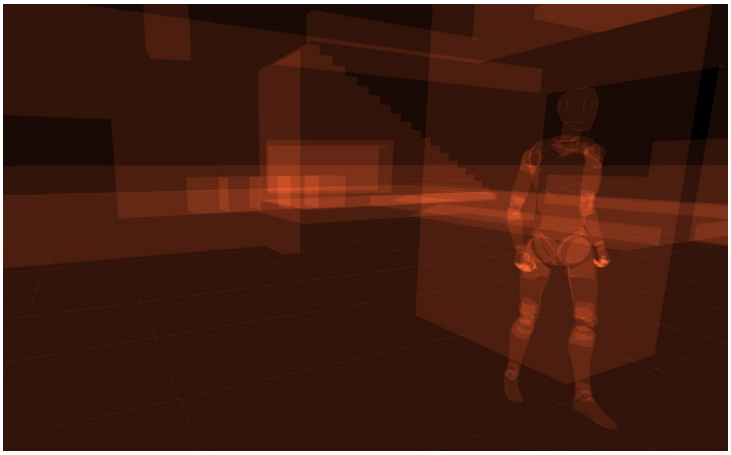


Scene ビューのコントロールバーにある Overdraw

明るいピクセルは、オブジェクトが重なって描画されていることを示し、暗いピクセルはオーバードローが少ないことを意味します。



標準的な Shaded ビューのシーン



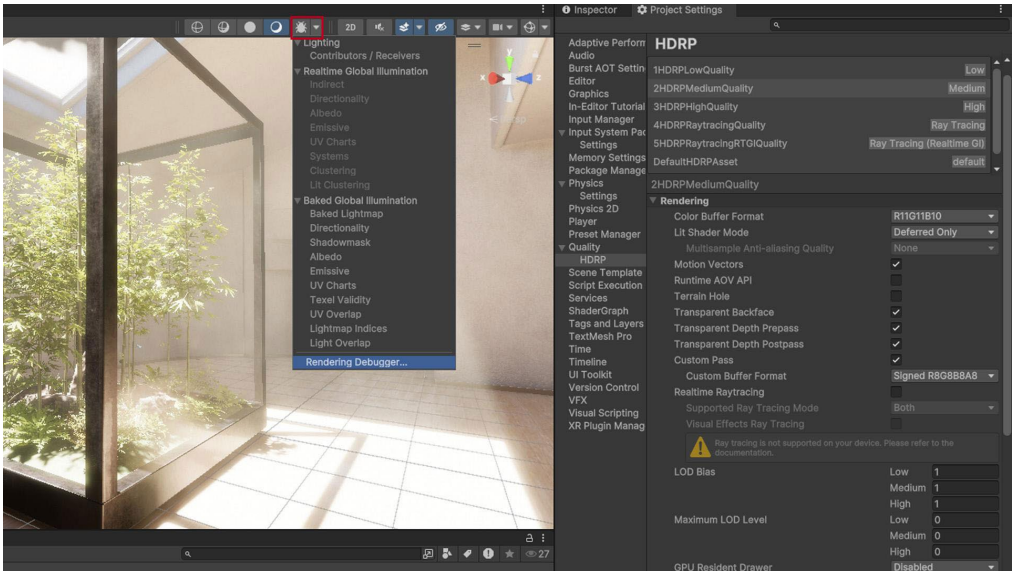
同じシーンの Overdraw ビュー。バーラップするジオメトリは、しばしばオーバードローの原因となる。

HDRP ではレンダーキューのコントロール方法が多少異なります。HDRP は、以下を行ってレンダーキューの順番を計算します。

- 共有マテリアルごとにメッシュをグループ化
- マテリアルの Priority に基づいて、それらのグループのレンダリング順序を計算
- 各 Mesh Renderer の Priority プロパティを使用して各グループをソート

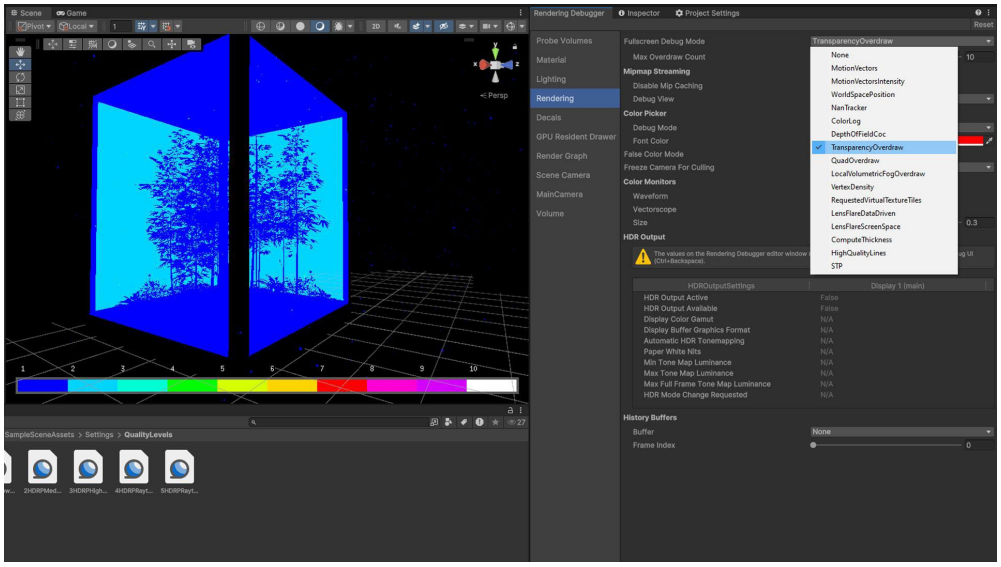
その結果キューには、最初にマテリアルの Priority で、そして次に各 Mesh Renderer の Priority でソートされた、ゲームオブジェクトのリストが反映されます。詳細については、「Renderer と Material Priority」ページで説明されています。

HDRP で透明度オーバードローを視覚化するには、Render Pipeline Debug ウィンドウ（**「Window」 > 「Render Pipeline」 > 「Render Pipeline Debug」**）を使用して、「TransparencyOverdraw」を選択します。

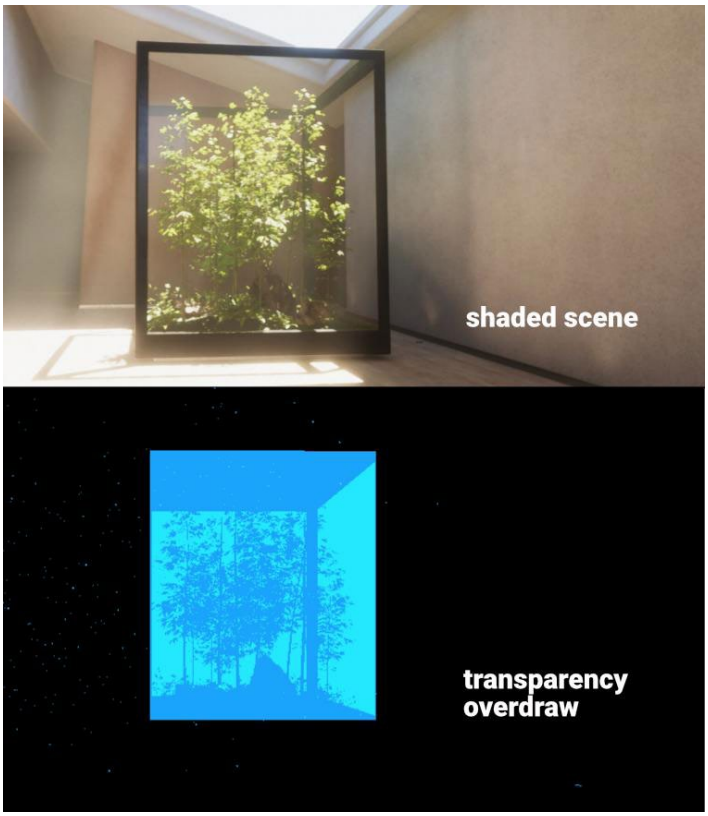


Rendering Debugger にアクセスする様子

透明度オーバードローを表示するには、「Rendering」を選択し、「Fullscreen Debug Mode」のドロップダウンから「TransparencyOverdraw」を選択します。



透明度オーバードローを可視化する様子



HDRP Render Pipeline Debug ウィンドウでは、透明マテリアルからのオーバードローを視覚化できる。

ヒートマップは、シーンでレンダリングされる透明度レイヤーを青から白に視覚的に表現します。青はオーバードローが少ないことを示し、白はパフォーマンスに重大な影響を与えるほど透明度のレイヤーが多いことを示します。

透明度オーバードローの可視化を使用することで、より良いパフォーマンスのためにシーンを最適化する方法について、情報に基づいた決定を下すことができます。

これらの診断ツールは、オーバードローを修正する際に、最適化のバロメーターを視覚的に提供してくれます。

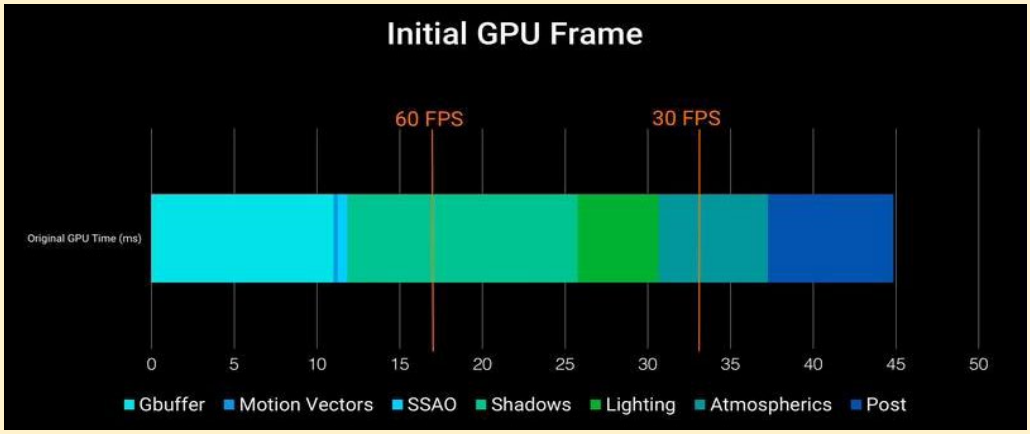
コンソール向けにグラフィックスを最適化する

Xbox コンソールや PlayStation® 向けの開発は、PC 向けの開発に似ていますが、これらのプラットフォームには独自のテクニックが必要です。スムーズなフレームレートを実現するためには、GPU の最適化に注力する必要があります。

パフォーマンスのボトルネックを特定する

PIX for Xbox と PlayStation のプロファイラーツールを、これらのプラットフォームでの最適化のためのツールボックスの一部として使用してください。

まずは、GPU ロードが高いフレームを見つけます。Microsoft と Sony は CPU と GPU の両方においてプロジェクトのパフォーマンスを分析するための優れたツールを提供しています。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。



ビューが PlayStation®4 Pro で 1 フレームあたり約 45 ミリ秒の GPU 依存であることを示している。

バッチカウントを削減する

他のプラットフォームと同様に、コンソールでの最適化は、しばしばドローコールのバッチを削減することを指します。これには、以下のようなテクニックが使えます。

- オクルージョンカリングを使用すると、前景オブジェクトに隠れたオブジェクトを削除してオーバードローを削減できます。ただし、これには追加の CPU 処理が必要となるため、Unity Profiler を使用して、GPU から CPU への作業移動が有益であることを確認してください。
- GPU インスタンシングもまた、同じメッシュやマテリアルを共有するオブジェクトが多くある場合に、バッチを削減できます。シーン内のモデル数を制限することで、パフォーマンスを向上させることができます。巧妙に行えば、複雑なシーンも反復的に見せることなくビルドできます。

— 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。必要なだけマテリアルを使用することで、この SRP バッチ処理の恩恵を受けることができますが、互換性のある少数のシェーダー（URP と HDRP で Lit シェーダーと Unlit シェーダーなど）に限定してください。

Graphics Jobs をアクティベートする

「Player Settings」>「Other Settings」でこのオプションを有効にすると、PlayStation または Xbox コンソールのマルチコアプロセッサーを活用できます。**Graphics Jobs** は、Unity が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軽減を可能にします。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。

ポストプロセスをプロファイルする

必ずコンソール用に最適化されたポストプロセスアセットを使用してください。元々 PC 用にオーサリングされた Asset Store のツールは、Xbox コンソールや PlayStation では必要以上にリソースを消費する場合があります。念のため、ネイティブのプロファイラーを使ってプロファイルしてください。

テッセレーションシェーダーの使用を避ける

テッセレーションは、シェイプを細分化してより小さくします。これにより、ジオメトリが細かくなり、ディテールの向上が期待できます。リアルな樹皮を表現したい場合など、テッセレーションが良い効果をもたらすこともありますが、一般的には、コンソール向けにはテッセレーションの使用を控えてください。GPU に高い負荷がかかってしまいます。

ジオメトリシェーダーをコンピュートシェーダーに置き換える

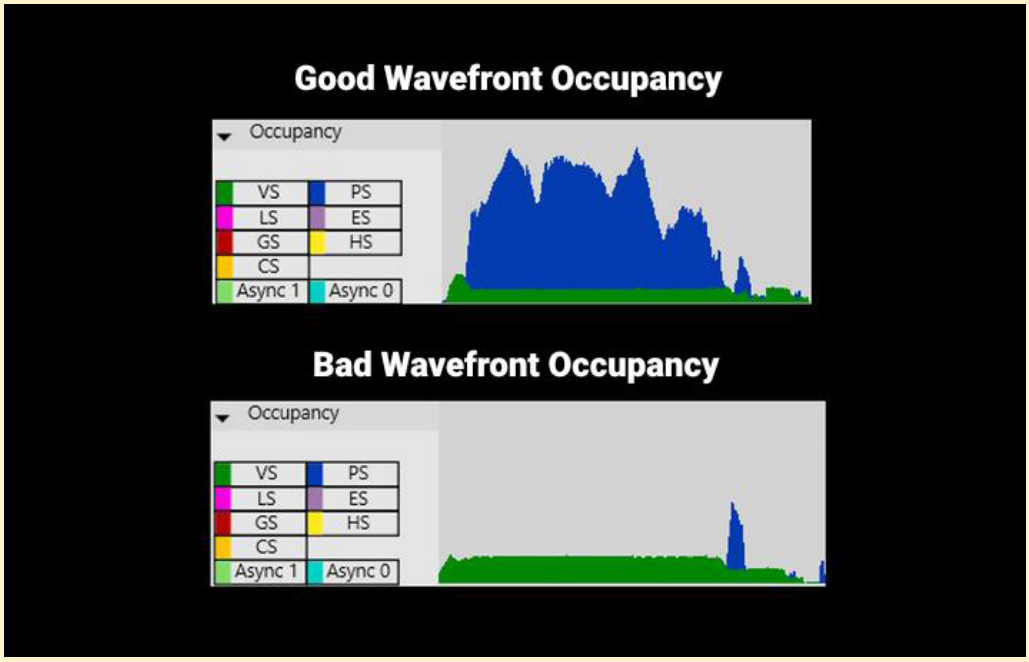
テッセレーションシェーダーと同様に、ジオメトリシェーダーと頂点シェーダーは、GPU 上で 1 フレームにつき 2 回実行されることがあります。

GPU 上で頂点データを生成したり修正したりしたい場合は、ジオメトリシェーダーよりもコンピュートシェーダーの方が良い選択となることがあります。コンピュートシェーダーで作業を行うことで、実際にジオメトリをレンダリングする頂点シェーダーを比較的高速かつシンプルなものにすることができます。

良好なウェーブフロントの占有率を目指す

ドローコールを GPU に送ると、その作業は多くのウェーブフロントに分割され、Unity はそれを GPU 内の利用可能な SIMD に分配します。

各 SIMD の一度に実行可能なウェーブフロントの最大数は決まっています。ウェーブフロントの占有率とは、現在使用されているウェーブフロントの数の最大数に対する比率のことです。これは、GPU のポテンシャルをどれだけ活用できているかを測るものです。コンソール固有のパフォーマンス分析ツールは、ウェーブフロントの占有率を詳細に示します。



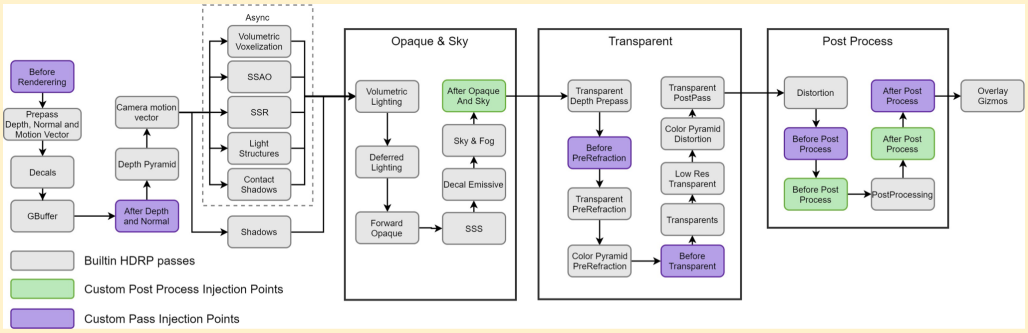
ウェーブフロントの良好な占有率と不良な占有率

上記の例では、頂点シェーダーのウェーブフロントが緑色で表示されています。ピクセルシェーダーのウェーブフロントは青く表示されています。下のグラフでは、ピクセルシェーダーがあまり動作していないのに、頂点シェーダーのウェーブフロントがたくさん現れています。これは、GPU のポテンシャルを十分に活用できていないということです。

もし、ピクセルの描画と関係ない頂点シェーダーの作業をたくさんしているなら、非効率な描画を行っているということかもしれません。ウェーブフロントの占有率が低いことは必ずしも悪いことではありませんが、シェーダーの最適化や、他のボトルネックのチェックを検討するための指標となります。例えば、メモリや演算処理が原因でストールが発生している場合、占有率を上げることでパフォーマンスが向上する可能性があります。一方で、処理中のウェーブフロントが多すぎると、キャッシュスラッシングが発生し、パフォーマンスが低下します。

HDRP ビルトインパスおよびカスタムパスを使用する

プロジェクトが HDRP を使っている場合、そのビルトインパスおよびカスタムパスを活用してください。これらは、シーンのレンダリングに役立ちますビルトインパスは、シェーダーの最適化に役立ちます。HDRP には、シェーダーにカスタムパスを追加できる注入ポイントがいくつか用意されています。



HDRP 注入ポイントを使用してパイプラインをカスタマイズする。

透明なマテリアルの動作の最適化については、こちらのレンダラーとマテリアルのプロパティに関するページを参照してください。

シャドウマッピングのレンダーターゲットのサイズを縮小する

HDRP の High Quality 設定では、デフォルトで 4K シャドウマップを使用します。シャドウマップの解像度を下げ、フレームのコストへの影響を測定します。ただし、ライトの設定でビジュアル品質の変化を補う必要があるかもしれないことにはご注意ください。

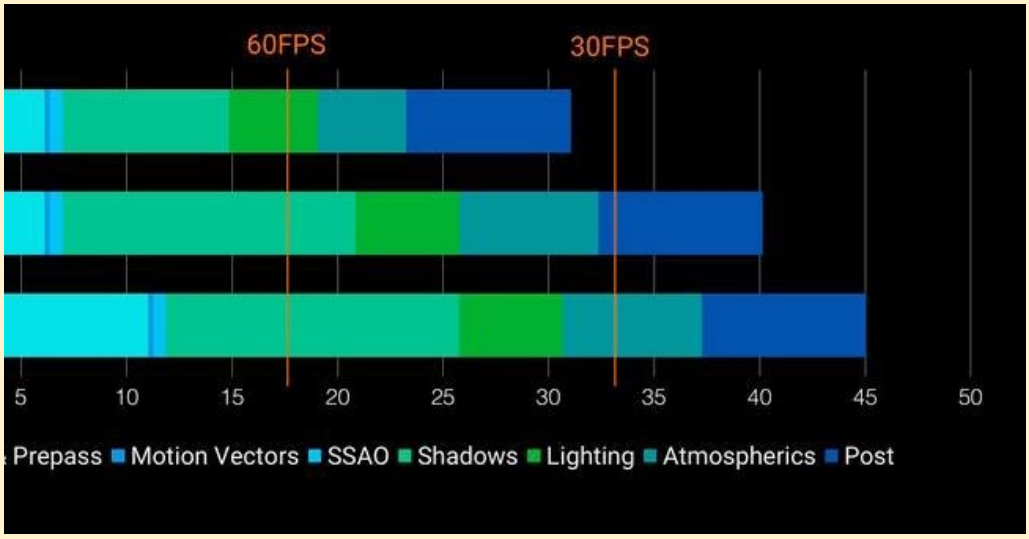
Async Compute を活用する

GPU を十分に活用できていない区間がある場合、Async Compute を使用することで、意味のあるコンピュートシェーダーの作業をグラフィックキューに移動させて、並列化することができます。これにより、GPU のリソースを有効に活用できます。

例えば、シャドウマップ生成時には、GPU は深度のみのレンダリングを行います。この時点ではピクセルシェーダーの作業はほとんど行われず、多くのウェーブフロントが占有されないままとなります。

Async Compute は、コンピュートシェーダーの作業をグラフィックキューに移動させて、並列化する。

コンピュートシェーダーの作業を深度のみのレンダリングと同期させることができれば、GPU の全体的な使用率を高めることができます。未使用のウェーブフロントは、スクリーンスペースアンビエントオクルージョンや、現在の作業を補完するタスクに使うことができます。



30fps で最適化されたレンダー

上記の例では、いくつかの最適化により、シャドウマッピング、ライティングパス、大気の表現の処理にかかる時間を数ミリ秒短縮しています。その結果、フレームのコストを下げ、PlayStation®4 Pro で 30fps で動作するようになりました。

カリング

オクルージョンカリングは、他のゲームオブジェクトによって完全に隠されている（オクルードされている）ゲームオブジェクトを無効にします。これにより、CPU と GPU が、カメラに映ることのないオブジェクトのレンダリングに時間を費やすのを防ぐことができます。

カリングはカメラごとに起こります。そのため、特に複数のカメラを同時に有効にした場合、パフォーマンスに大きな影響を与える可能性があります。Unity は錐台カリングとオクルージョンカリングの 2 種類のカリングを使用します。

錐台カリングは各カメラで自動的に実行されます。これは、View Frustum の外にあるゲームオブジェクトがレンダリングされることを防ぐため、パフォーマンスの最適化に役立ちます。

レイヤーごとのカリング距離は、Camera.layerCullDistances で手動で設定できます。これにより、デフォルトの farClipPlane よりも短い距離で小さなゲームオブジェクトをカリングすることができます。

そのためには、ゲームオブジェクトをレイヤーに整理します。layerCullDistances 配列を使用して、32 個のレイヤーのそれぞれに farClipPlane より小さい値を割り当てます (または、0 を使用してデフォルトとして farClipPlane を適用します)。

Unity は、まずレイヤーごとにカリングを行い、ゲームオブジェクトはカメラが使用するレイヤーにのみ残ります。その後、錐台カリングによって、カメラ錐台の外にあるゲームオブジェクトが削除されます。錐台カリングは、利用可能なワーカースレッドを利用する一連のジョブとして実行されます。

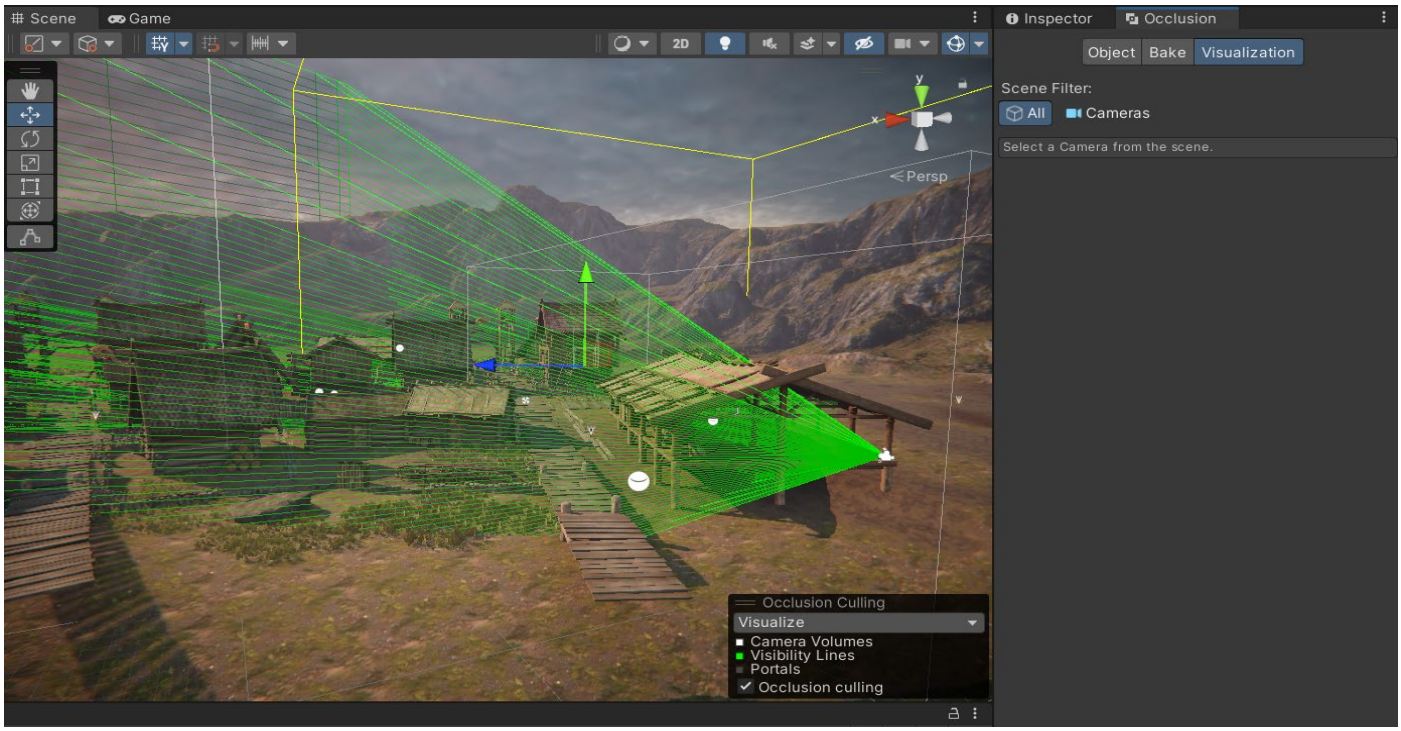
各レイヤーのカリングテストは短時間で済みます（実質的にはビットマスク操作のみのため）。しかし、ゲームオブジェクトの数が多ければ、このコストはかさみます。これプロジェクトにおいて問題になる場合、Unity のレイヤーまたは錐台カリングシステムへの負担を軽減するために、ワールドを「セクター」に分割し、カメラ錐台の外側にあるセクターを無効にするシステムを実装する必要があるかもしれません。

オクルージョンカリングは、ゲームオブジェクトがカメラに映らない場合、ゲームビューから削除します。他のオブジェクトの後ろに隠れているオブジェクトも、レンダリングされリソースが費やされてしまうため、この機能を使用してレンダリングされないようにします。例えば、ドアが閉まっていて、ドアの向こう側の部屋の中がカメラに映らない場合は、その部屋をレンダリングする必要はありません。

オクルージョンカリングを有効にすると、パフォーマンスは大幅に向上しますが、必要なディスク容量、CPU 時間、RAM も増加します。Unity はビルド中にオクルージョンデータをベイクし、シーンのロード中にディスクから RAM にロードする必要があります。

カメラビュー外の錐台カリングは自動で行われますが、オクルージョンカリングにはベイクプロセスが必要です。オブジェクトを Static.Occluder または Occludee としてマークし、**「Window」>「Rendering」>「Occlusion Culling」**ダイアログからベイクします。

詳細については、チュートリアル「オクルージョンカリングの使用方法」をご覧ください。



オクルージョンカリングの例

動的解像度

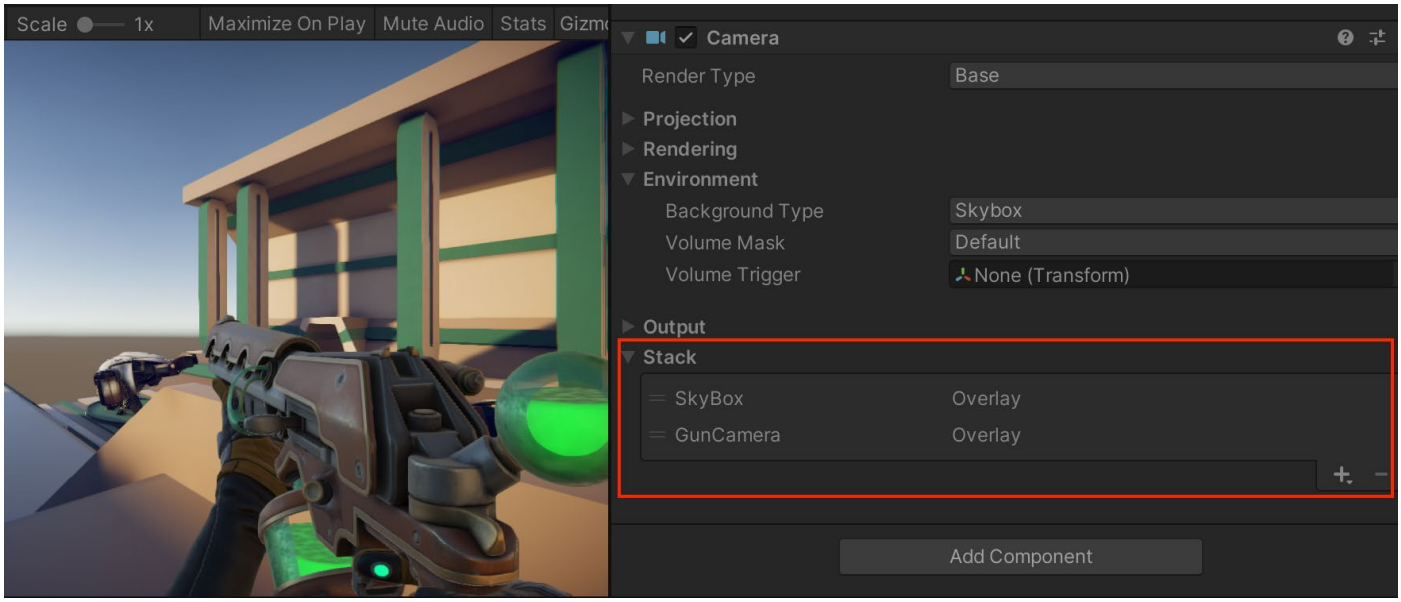
Allow Dynamic Resolution は、各レンダーターゲットを動的に拡大縮小して GPU の作業負荷を軽減する Camera 設定です。アプリケーションのフレームレートが低下した場合は、解像度のスケールを徐々に下げてフレームレートを一定に保つことができます。

Unity は、パフォーマンスデータが GPU 依存の結果としてフレームレートが低下しそうであることを示唆した場合、このスケーリングをトリガーします。このスケーリングは、スクリプトを使用して、手動で事前にトリガーすることもできます。これは、アプリケーションの GPU を多用するセクションに近づいている場合に便利です。徐々にスケールしていけば、動的解像度はほとんど気にならなくなります。

追加の情報やサポートされているプラットフォームの一覧については、動的解像度に関するマニュアルページを参照してください。

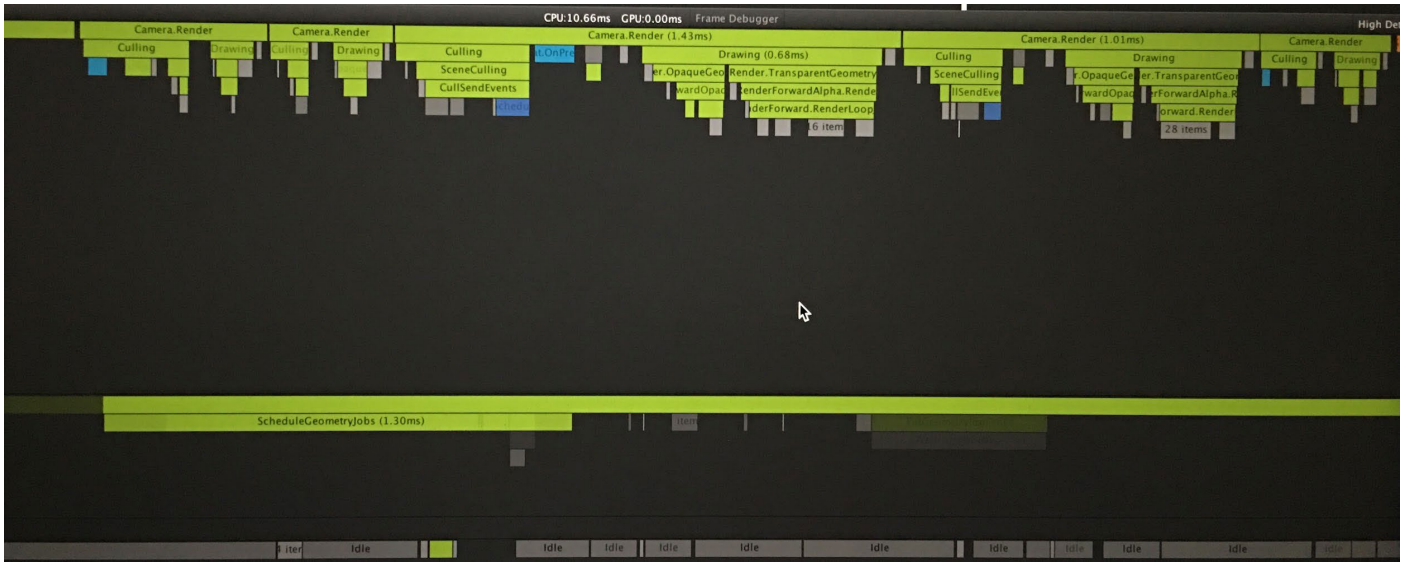
複数のカメラビュー

ゲームでは、複数の視点からレンダリングする必要がある場合があります。例えば、FPS ゲームでは、プレイヤーの武器と環境を異なる有効視野（FOV）で別々に描くのが一般的です。これにより、背景向けの広角 FOV から見た際に、前景にあるオブジェクトが極端に歪んで見えてしまう状態を防ぐことができます。



URP の Camera Stacking。銃と背景が異なるカメラ設定でレンダリングされる。

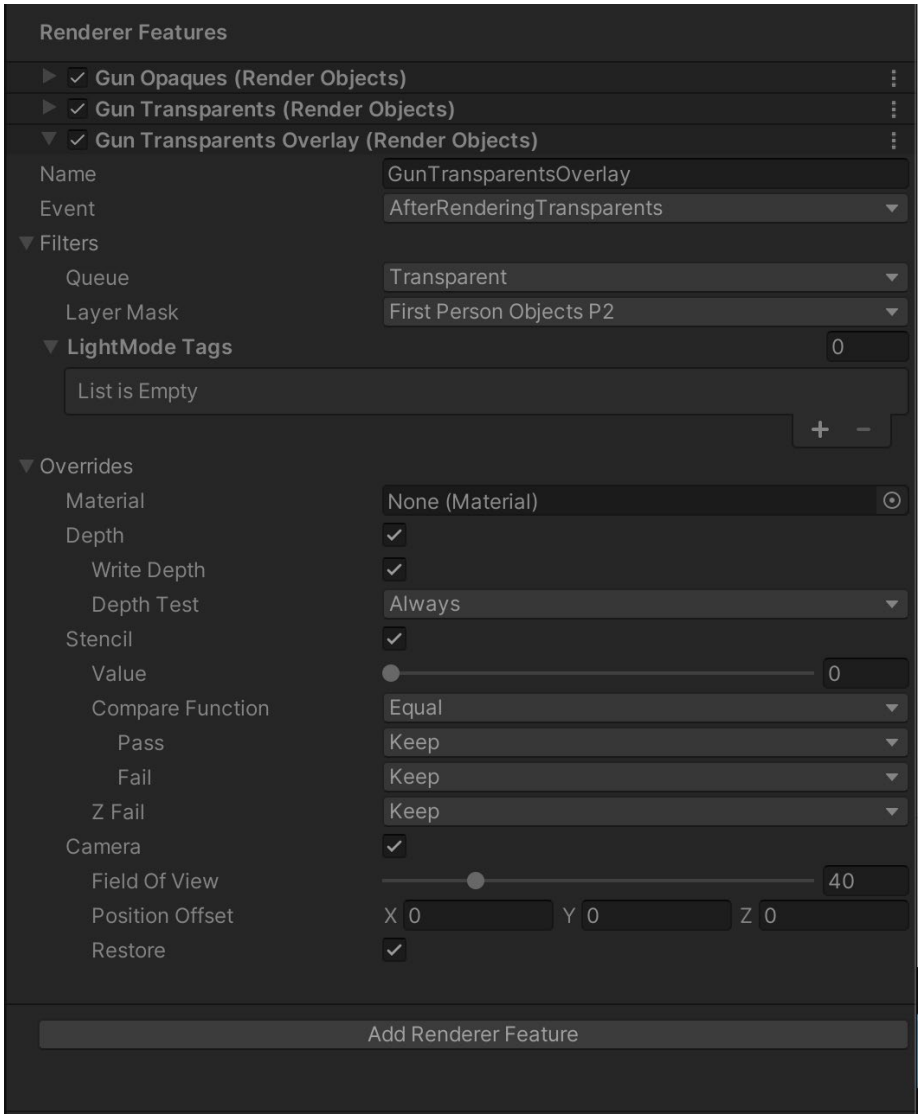
URP の Camera Stacking を使用して、2 つ以上のカメラビューをレンダリングすることができます。しかし、各カメラには依然として相当なカリングとレンダリング処理が行われます。有益な処理かどうかにかかわらず、各カメラは何らかのオーバーヘッドを発生させます。レンダリングに必要な Camera コンポーネントのみを使用するようにしてください。モバイルプラットフォームでは、何もレンダリングしていないときでも、各アクティブカメラは最大 1ms の CPU 時間を使用する可能性があります。



Unity CPU Profiler は、タイムラインビューにメインスレッドを表示し、複数のカメラがあることを示す。Unity は各カメラに対してカリングを実行する。

URP の Render Objects Renderer Feature

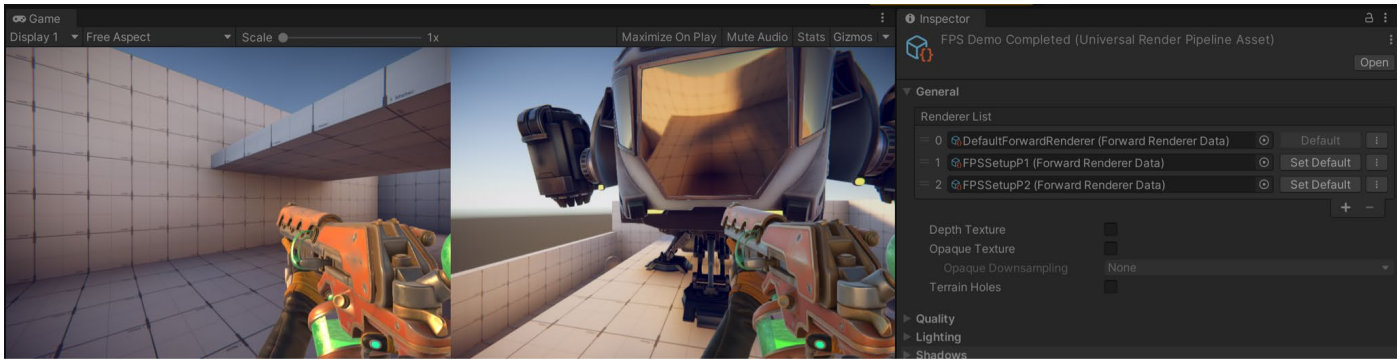
URP では、複数のカメラを使用する代わりに、カスタムの Render Objects Renderer Feature を使用してみてください。**Renderer Data** アセットで「**Add Renderer Feature**」をクリックし、「**Render Object**」を選択します。



カスタムレンダーオブジェクトを作成し、レンダー設定をオーバーライドする。

各レンダーオブジェクトをオーバーライドする際は、以下のことを行います。

- Event と関連付けしてレンダリングループの特定のタイミングに注入する
- Render Queue（Transparent または Opaque）と LayerMask でフィルター分けする
- Depth と Stencil の設定をオンにする
- Camera 設定（Field of View と Position Offset）を変更する



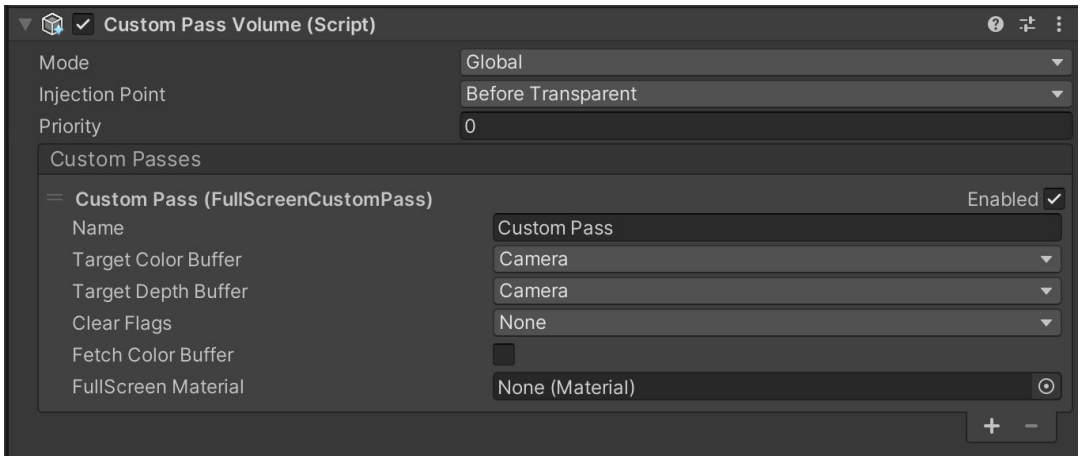
URP の Render Objects Renderer Feature 機能は、複数のレイヤーを 1 つのレンダリングビューに統合する。

HDRP の Custom Pass Volume

HDRP では、カスタムパスを使用して同様の効果を得ることができます。Custom Pass Volume を使用した Custom Pass の設定は、HDRP Volume を使用した場合と似ています。

Custom Pass は以下のことを可能にします。

- シーン内のマテリアルの外観を変更する
- Unity がゲームオブジェクトをレンダーする順番を変更する
- カメラバッファをシェーダーに読み込む

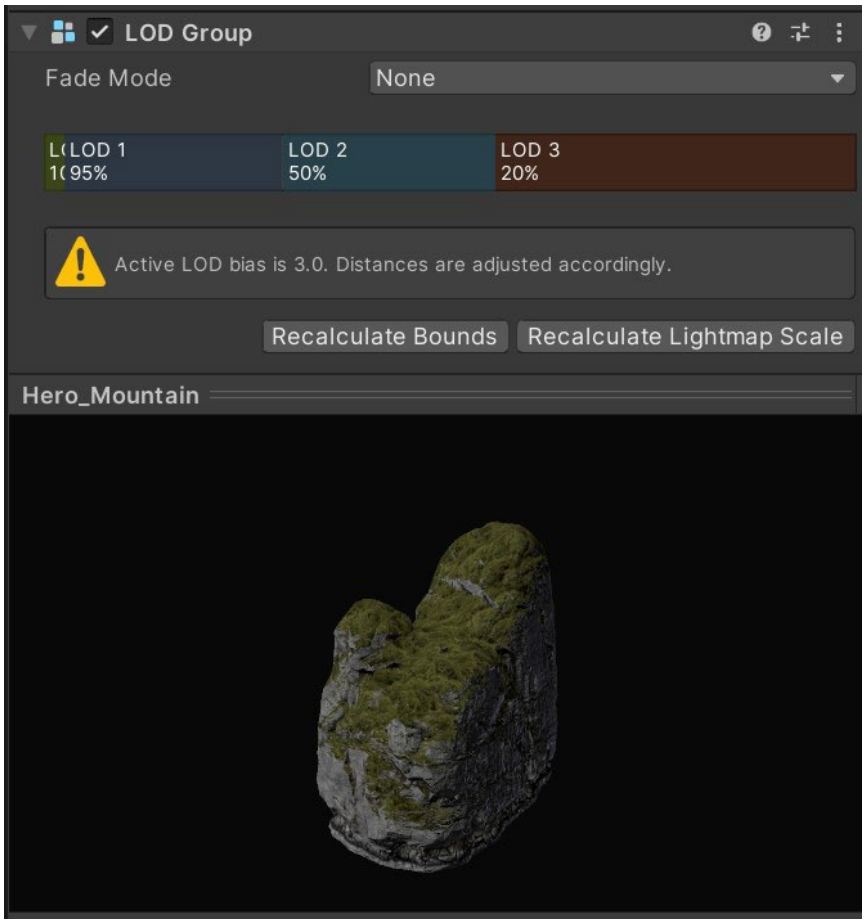


HDRP の Custom Pass Volume

HDRP の Custom Pass Volumes を使用することで、余分なカメラの使用とそれに伴って発生する追加のオーバーヘッドを防ぐことができます。カスタムパスには、シェーダーとの相互作用の仕方に多くの柔軟性があります。C# で Custom Pass クラスを拡張することもできます。

Level of Detail（LOD）を使用する

オブジェクトが遠くに移動するに従って、Level of Detail（LOD）は GPU パフォーマンスを助けるために、より単純なマテリアルとシェーダーで低解像度のメッシュを使用するように調整または切り替えを行います。



メッシュに Level of Detail Group が使用されている例

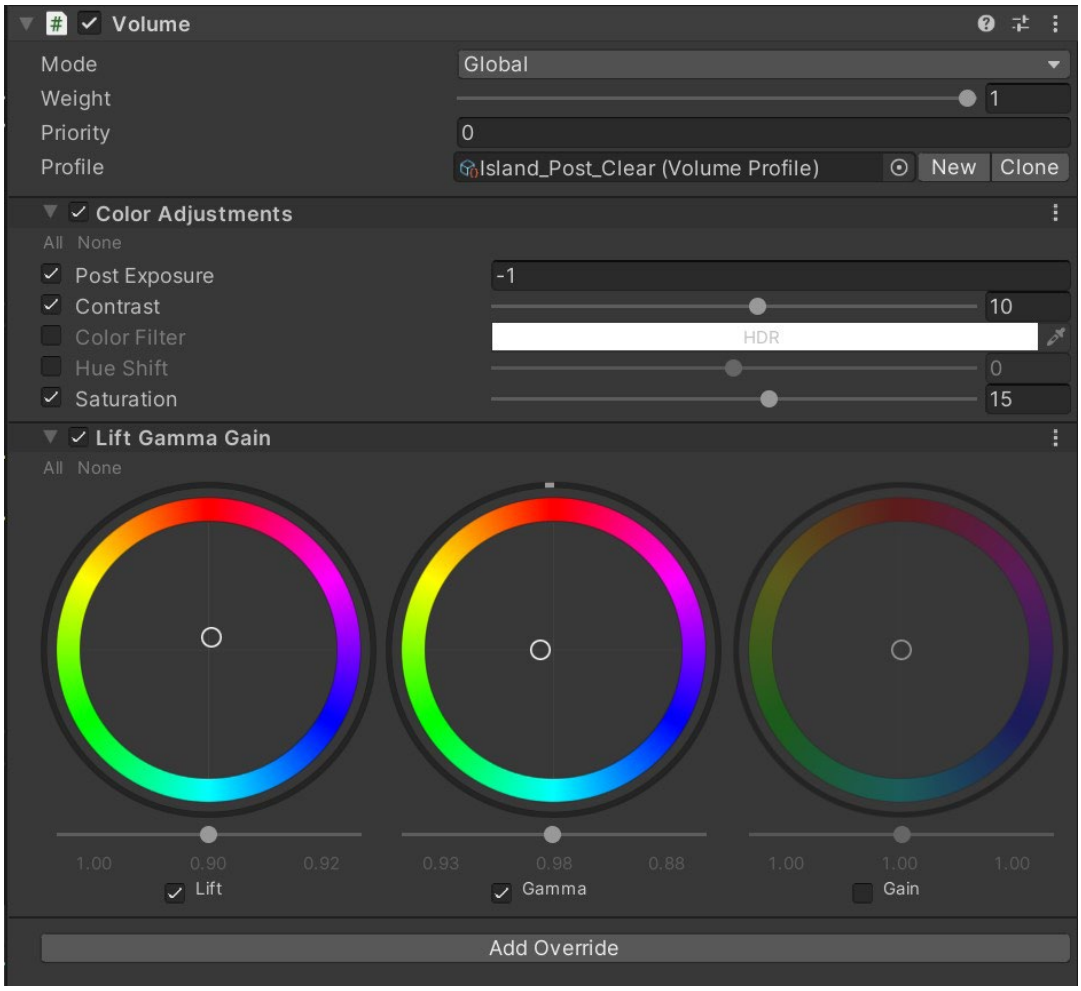
さまざまな解像度でモデリングされたソースメッシュ

詳細については、Unity Learn の「LOD の使い方、設定方法」をご覧ください。

ポストプロセスエフェクトをプロファイルする

ポストプロセスエフェクトをプロファイルして GPU のコストを確認しましょう。ブルームや被写界深度のようなフルスクリーン効果には負荷が高いものもありますが、ビジュアル品質とパフォーマンスのバランスが取れるまで試してみてください。

ポストプロセスエフェクトはランタイム時にあまり変動しない傾向があります。ボリュームオーバーライドを決定したら、ポストエフェクトに総フレーム予算の一部を固定的に割り当てます。



ポストプロセスエフェクトは可能な限りシンプルに保つ。

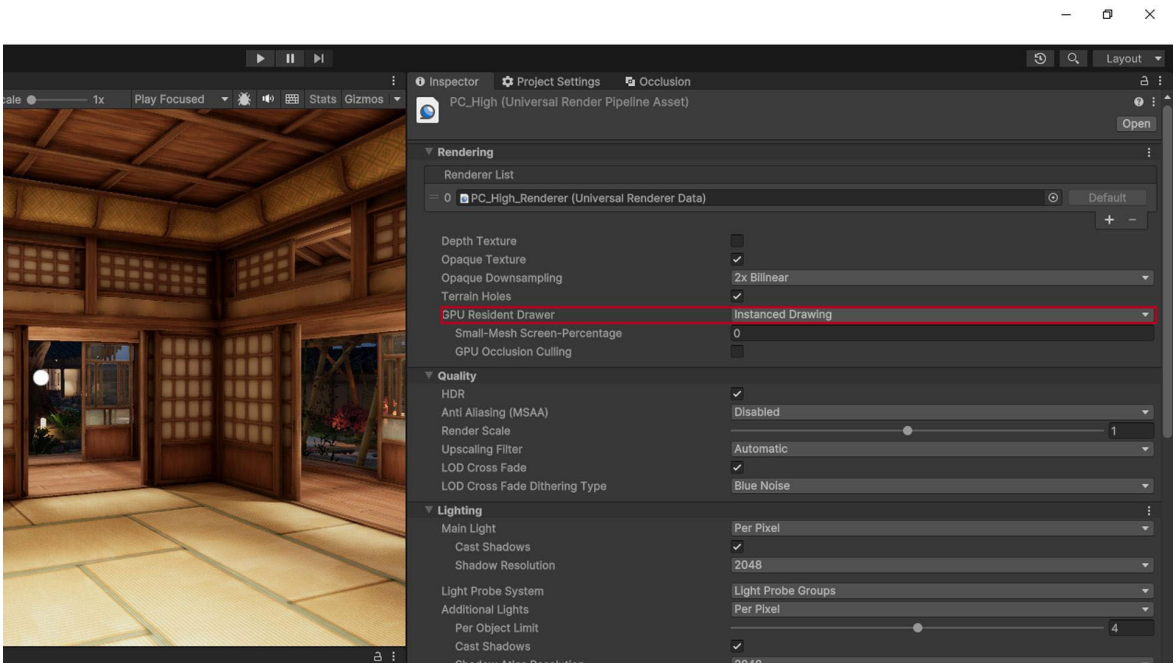
GPU Resident Drawer

GPU Resident Drawer（URP および HDRP で利用可能）は、CPU 時間を最適化するように設計された GPU 駆動のレンダリングシステムで、パフォーマンスに大きなメリットをもたらします。クロスプラットフォームのレンダリングをサポートし、既存のプロジェクトでそのまま使用できるように設計されています。

GPU Resident Drawer で実行されている URP 3D サンプルの庭園環境

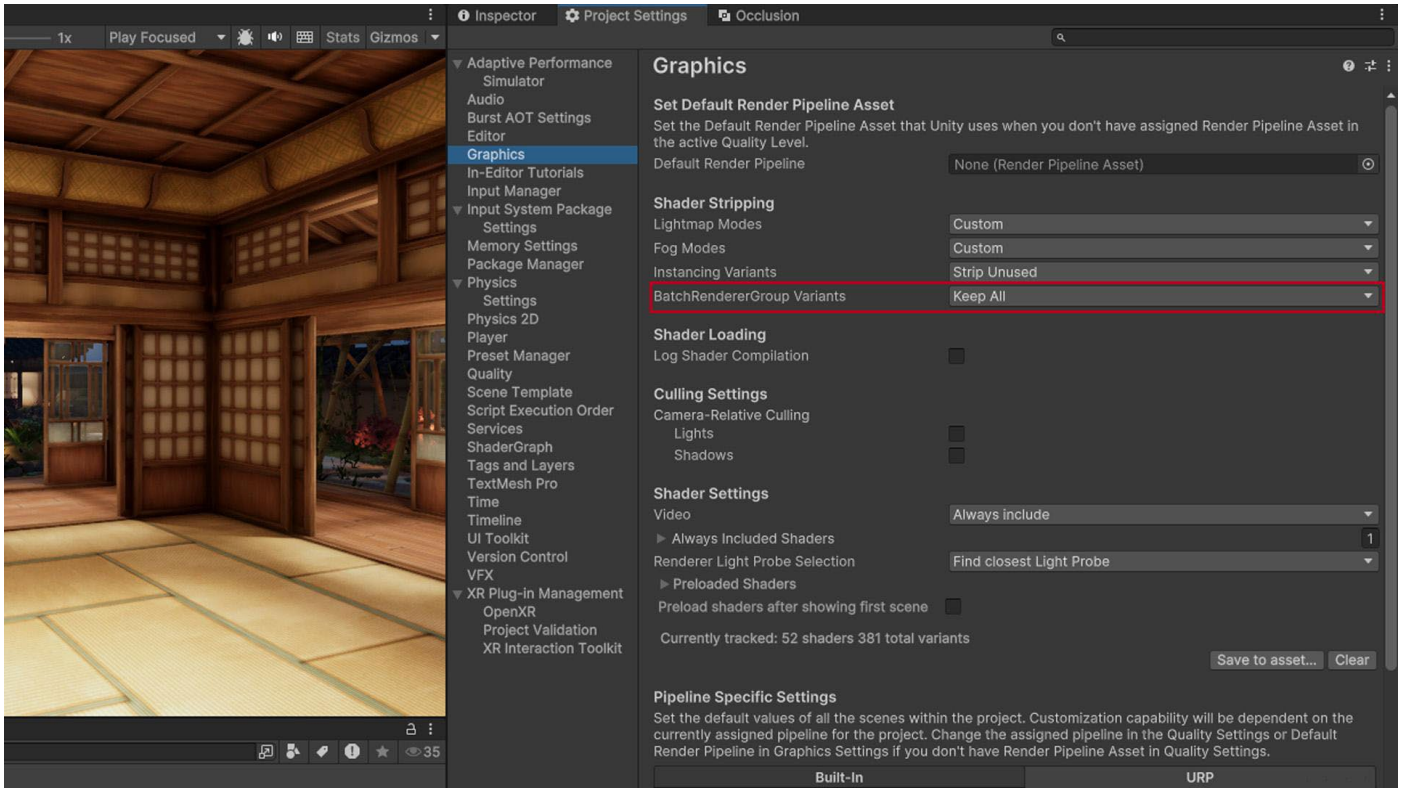
このシステムは、HDRP または URP Render Pipeline Asset 内で有効にできます。「**Instanced Drawing**」を選択して有効にします。また、再生モードのみで有効にするか、編集モードでも有効にするかを選択することもできます。

GPU Resident Drawer を有効にすると、ドローコールの回数が多いために CPU 依存となっているゲームは、ドローコールの量が減るためパフォーマンスが向上します。どの程度の改善が見られるかは、シーンの規模やインスタンシングの量によって異なります。レンダリングするインスタンス化可能なオブジェクトの数が多いほど、目に見えるメリットが大きくなります。



GPU Resident Drawer のドロップダウンから Instanced Drawing を選択

「Instanced Drawing」オプションを選択すると、で「BatchRenderGroup Variants の設定は『Keep All』でなければならない」という旨の UI 警告メッセージが表示されることがあります。グラフィックス設定でこのオプションを調整すると、GPU Resident Drawer の設定が完了します。



BatchRenderGroup バリアントを **Keep All** に設定

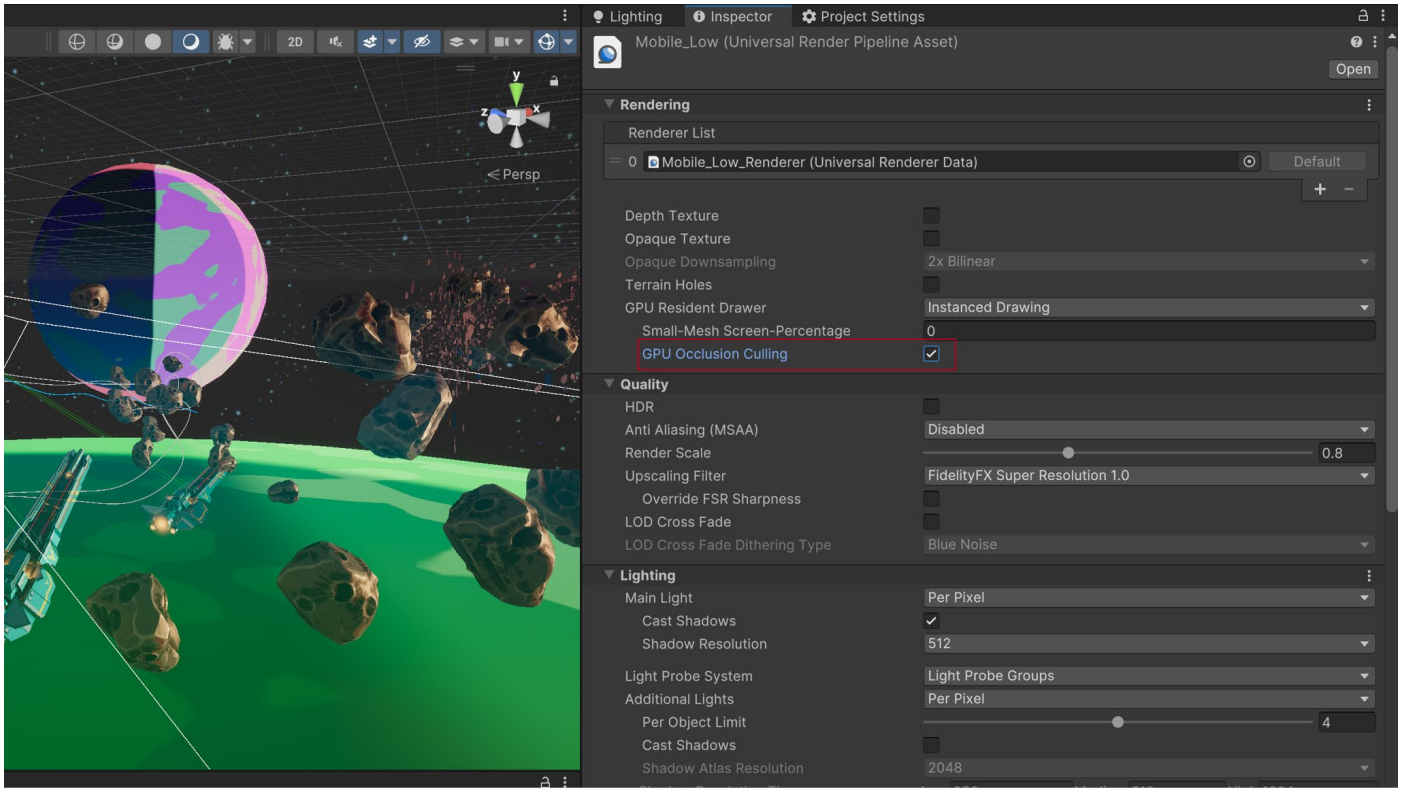
さらに学びたい場合は、こちらのディスカッションスレッドをご覧ください。

GPU オクルージョンカリング

GPU オクルージョンカリングは、URP と HDRP の両方で利用可能で、GPU Resident Drawer と連動して動作します。これは、各フレームのオーバードローの量を減らすことでパフォーマンスを大幅に向上させ、レンダラーが見えないものを描画するためにリソースを浪費することはありません。

GPU オクルージョンカリングを有効にするには、レンダーパイプラインアセットを探し、「GPU Occlusion」にチェックを入れます。

Unity 6 のデバッグモードで GPU オクルージョンカリングを有効にするには、**「Window」>「Rendering」>「Occlusion Culling」**から設定できます。ここでは、さまざまな視覚化オプションを切り替えることで、オブジェクトがどのようにカリングされているかを確認できます。



Render Pipeline Asset の GPU Occlusion Culling オプション

Graphics Jobs を分割する

Split Graphics Jobs は、レンダリングコマンドを複数の CPU コアでより効率的に実行し、レンダリングタスク間の並列性を高めることでパフォーマンスを向上させます。

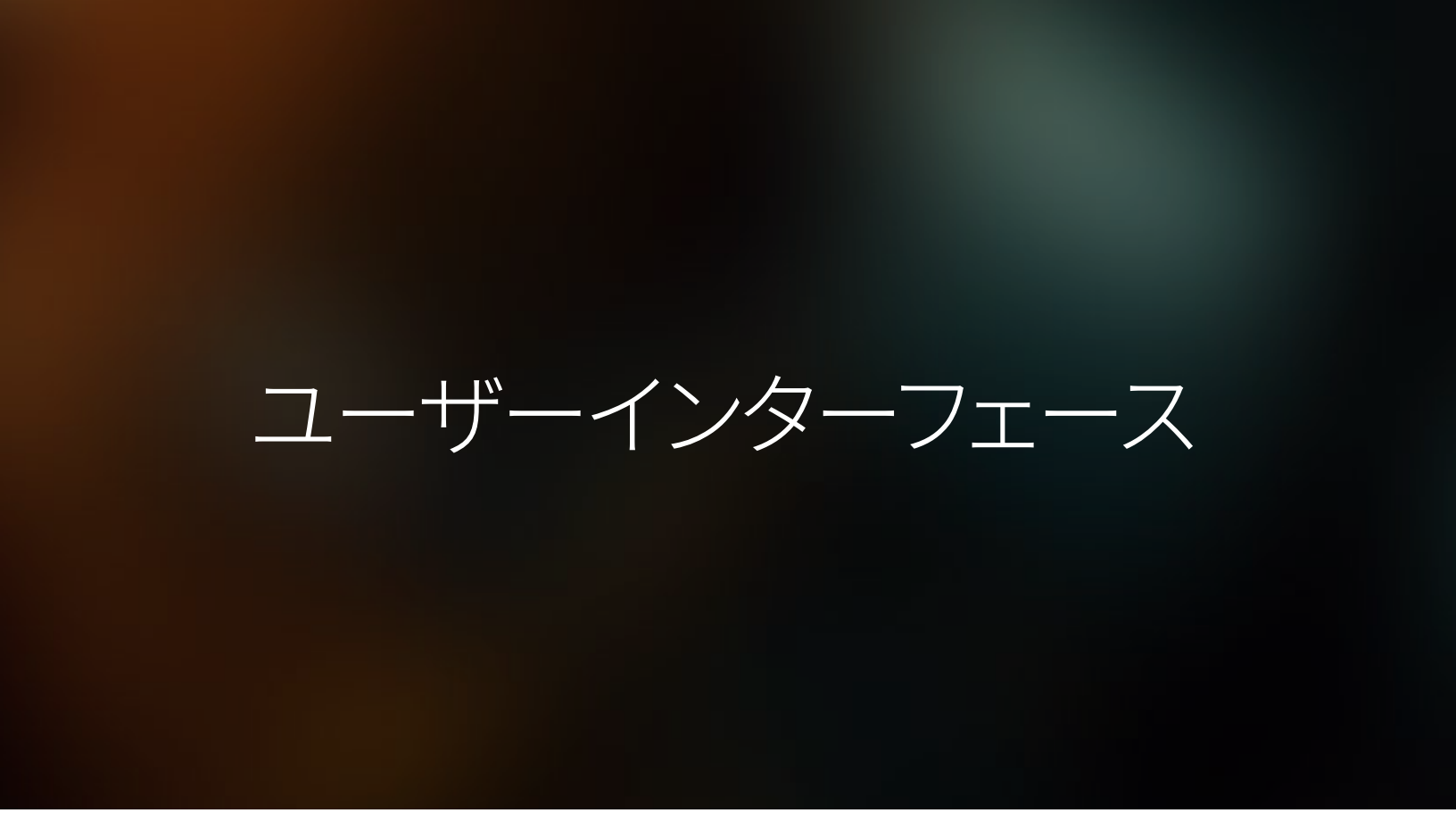
これは、複数のデスクトップおよびコンソールプラットフォームでサポートされるスレッドモードを提供します。主な改善点は、メインスレッド（一般的なゲームロジックとオーケストレーションを担当）とネイティブグラフィックスジョブスレッド（レンダリングタスクを担当）間の不必要な同期が削減されることです。

この新しいスレッドモードによるパフォーマンスの向上は、各フレームで送信されるドローコールの数に応じて変化します。ドローコールが多いほどこれらの最適化によるメリットは大きくなり、多くのオブジェクトやテクスチャを含む複雑なシーンでは、パフォーマンスが大幅に向上します。

Splits Graphics Jobs は、Windows 向けの DX12 を使用時に利用可能で、Vulkan Player でもサポートされています。

これは、**「Project Settings」>「Player」>「Other Settings」>「Graphics Jobs Mode」**で「Split」を選択することで有効にできます。

以前利用可能だったレガシーモードやネイティブモードではなく、常に Splits Graphics Jobs を使用することをお勧めします。



ユーザーインターフェース

ユーザーインターフェース

Unity には 2 つの UI システムがあります。従来の Unity UI と新しい UI Toolkit です。UI Toolkit は、推奨される UI システムになることを目的としています。標準的なウェブ技術にヒントを得たワークフローとオーサリングツールにより、最大限のパフォーマンスと再利用性を実現しています。そのため、すでにウェブページのデザイン経験がある UI デザイナーやアーティストにとって親しみやすいものとなっています。

しかし、Unity 6 の時点では、UI Toolkit には Unity UI や Immediate Mode GUI（IMGUI）ではサポートされている一部の機能がありません。Unity UI と IMGUI は、特定のユースケースにより適しており、レガシープロジェクトをサポートするために必要です。詳細については、「Unity の UI システムの比較」をご覧ください。

UGUI パフォーマンス最適化のヒント

Unity UI（UGUI）は、しばしばパフォーマンス問題の原因となることがあります。Canvas コンポーネントは、UI 要素のメッシュを生成および更新し、GPU にドローコールを発行します。その機能は負荷が高いため、扱う際には以下のことに注意してください。

キャンバスを分割する

1 つの大きなキャンバスに何千もの要素がある場合、1 つの UI 要素を更新するとキャンバス全体が更新されるため、CPU スパイクが発生する可能性があります。

そこで、複数のキャンバスをサポートする UGUI の機能を活用しましょう。更新頻度に応じて UI 要素を分割してください。静的な UI 要素は別のキャンバスに置き、同時に更新される動的な要素は小さなサブキャンバスに置きます。

各キャンバス内のすべての UI 要素が同じ Z 値、マテリアル、テクスチャを持つようにしてください。

見えない UI 要素を非表示にする

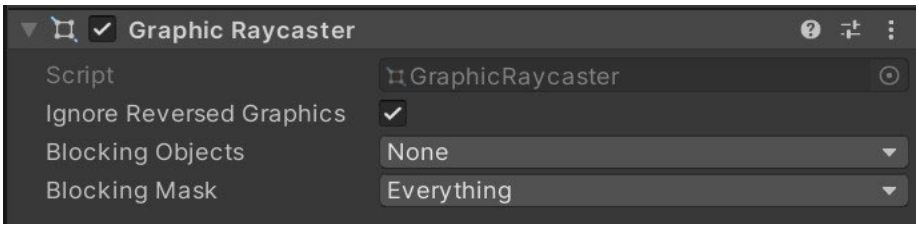
ゲーム中に散発的にしか表示されない UI 要素（キャラクターがダメージを受けたときに表示されるヘルスバーなど）があるとします。不可視の UI 要素がアクティブであれば、まだドローコールを使っているかもしれません。不可視の UI コンポーネントを明示的に無効にし、必要に応じて再び有効にしてください。

キャンバスの可視性をオフにしたいだけなら、ゲームオブジェクト全体ではなく、Canvas コンポーネントを無効にします。これにより、再び有効にした際にメッシュや頂点をリビルドする必要がありません。

GraphicRaycaster を制限し、Raycast Target を無効にする

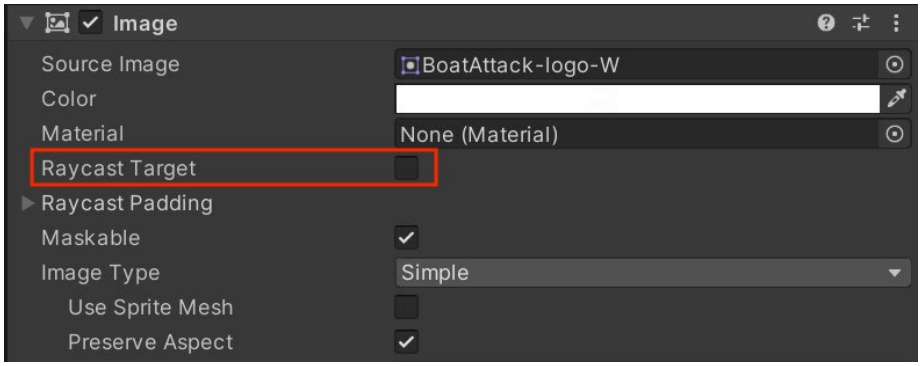
オンスクリーンタッチやクリックなどの入力イベントには、GraphicRaycaster コンポーネントが必要です。これは単純に、画面上の各入力点をループし、それが UI の RectTransform 内にあるかどうかを確認します。サブキャンバスを含め、入力が必要なすべてのキャンバスに Graphic Raycaster が必要です。

名前とは反対に、これは実際にはレイキャスターではありませんが、各交差判定にはそれなりの負荷がかかります。相互作用しない UI キャンバスには Graphic Raycaster を追加せず、数を最小限に抑えましょう。



相互作用しない UI キャンバスから GraphicRaycaster を削除する。

さらに、Raycast Target を必要としないすべての UI テキストと画像でこれを無効にします。UI が多くの要素で複雑な場合は、このような小さな変更で不必要な計算を減らすことができます。

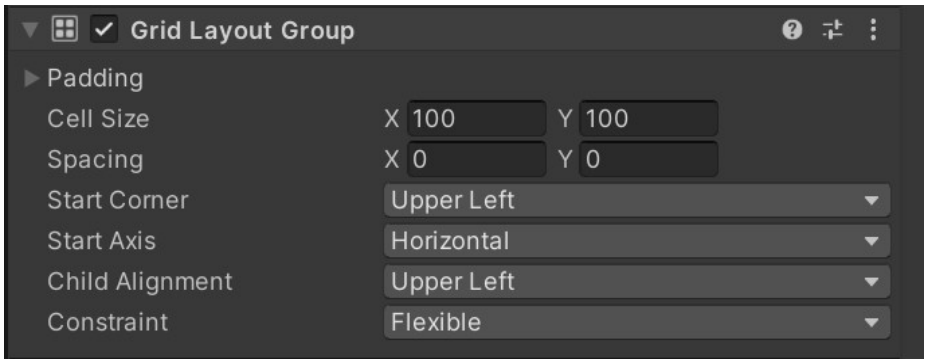


Raycast Target は可能な限り無効にする。

Layout Group の使用を避ける

Layout Group の更新は非効率的なので、使用は控えます。コンテンツが動的でない場合は完全に避け、代わりにプロポーショナルレイアウトの設計にアンカーを使います。そうでない場合は、UI を設定した後に Layout Group コンポーネントを無効にするカスタムコードを作成します。

動的要素に Layout Group（Horizontal、Vertical、Grid）を使用する必要がある場合は、パフォーマンスを向上させるため、ネストは避けてください。



Layout Group は、特にネストされている場合、パフォーマンスを低下させる可能性がある。

大仰なリストビューやグリッドビューは避ける

大仰なリストビューやグリッドビューは負荷が高くなります。大規模なリストビューやグリッドビューを作成する必要がある場合（数百のアイテムがあるインベントリ画面など）、すべてのアイテムに対して UI 要素を作成するのではなく、より小さな UI 要素のプールを再利用することを検討してください。こちらのサンプル GitHub プロジェクトで、実際の動作を確認してください。

多数のオーバーレイ要素を避ける

UI 要素をたくさん重ねると（カードバトルゲームでカードを重ねるなど）、オーバードローが発生します。コードをカスタマイズして、ランタイム時にレイヤー化された要素をより少ない要素とバッチにマージしてください。

全画面の UI を使用する場合に、その他のものをすべて非表示にする

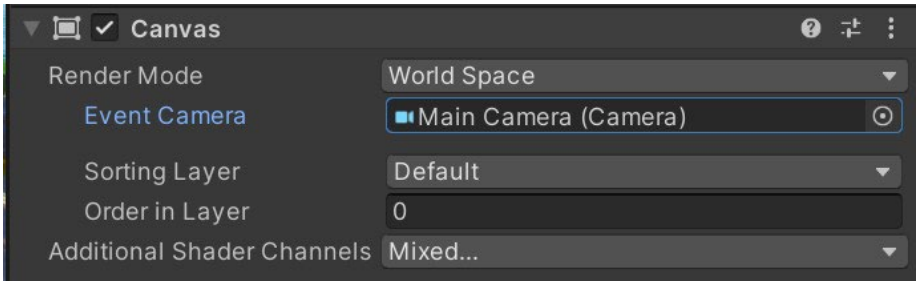
一時停止や開始の画面がシーンにある他のすべての要素を覆う場合は、その 3D シーンをレンダリングしているカメラを無効にします。同様に、一番上のキャンバスの後ろに隠れている、背景のキャンバス要素も無効にします。

全画面 UI の間は、60fps で更新する必要がないため、Application.targetFrameRate を下げることを検討してください。

ワールド空間とカメラ空間のキャンバスにカメラを割り当てる

「**Event Camera**」や「**Render Camera**」のフィールドを空白にしておくと、Unity は自動的に **Camera.main** を使用しますが、これには不要な負荷がかかります。

キャンバスの「**RenderMode**」には、カメラを必要としない「**Screen Space – Overlay**」を可能な限り使用してください。



Render Mode に World Space を使用する場合は、Event Camera を空白にしない。

UI Toolkit パフォーマンス最適化のヒント

UI を多用するゲームの場合は、一般的に Unity 6 で UI Toolkit を活用することをお勧めします。

UI Toolkit は、Unity UI よりもパフォーマンスが向上し、最大限のパフォーマンスと再利用性を実現するように調整されており、標準的なウェブテクノロジーからヒントを得たワークフローとオーサリングツールを提供します。その主な利点のひとつは、UI 要素のために特別に設計された、高度に最適化されたレンダリングパイプラインを使用していることです。

UI Toolkit を活用して UI のパフォーマンスを最適化するための一般的な推奨事項をいくつか紹介します。

効率的なレイアウトを使用する

効率的なレイアウトとは、手動で UI 要素の位置やサイズを調整するのではなく、UI Toolkit が提供する Flexbox などのレイアウトグループを使用することです。レイアウトグループはレイアウト計算を自動的に処理するため、パフォーマンスが大幅に向上します。指定されたレイアウトルールに基づき、UI 要素の位置やサイズが正しく調整されます。効率的なレイアウトを利用することで、手作業によるレイアウト計算のオーバーヘッドを回避し、一貫性のある最適化された UI レンダリングを実現します。

Update で負荷の高い操作を避ける

Update メソッドで実行される作業、特に UI 要素の作成、操作、計算のような重い操作を最小限に抑えます。Update メソッドは 1 フレームにつき 1 回呼び出されるため、これらの操作は控えめに、あるいは可能な限り初期化中に行うようにしてください。

イベント処理を最適化する

イベントのサブスクリプションに注意し、不要になったら登録を解除してください。過剰なイベント処理はパフォーマンスに影響するので、必要なイベントだけをサブスクライブしてください。

スタイルシートを最適化する

スタイルシートで使用されているスタイルクラスとセレクターの数に注意してください。多数のルールを持つ大規模スタイルシートはパフォーマンスに影響を与える可能性があります。スタイルシートは無駄を省き、不必要に複雑にすることは避けましょう。

プロファイリングと最適化を行う

Unity のプロファイリングツールを使用して、UI のパフォーマンスボトルネックを特定し、非効率なレイアウト計算や過剰な再描画など、さらに最適化できる領域を見つけ出しましょう。

ターゲットプラットフォームに焦点を当てる

ターゲットプラットフォームで UI のパフォーマンスをテストし、さまざまなデバイスで最適なパフォーマンスが得られるようにします。パフォーマンスはハードウェアの性能によって異なるため、UI を最適化する際はターゲットプラットフォームを考慮してください。

パフォーマンスの最適化は反復プロセスです。UI コードを継続的にプロファイリング、測定、最適化し、スムーズかつ効率的に実行できるようにしましょう。

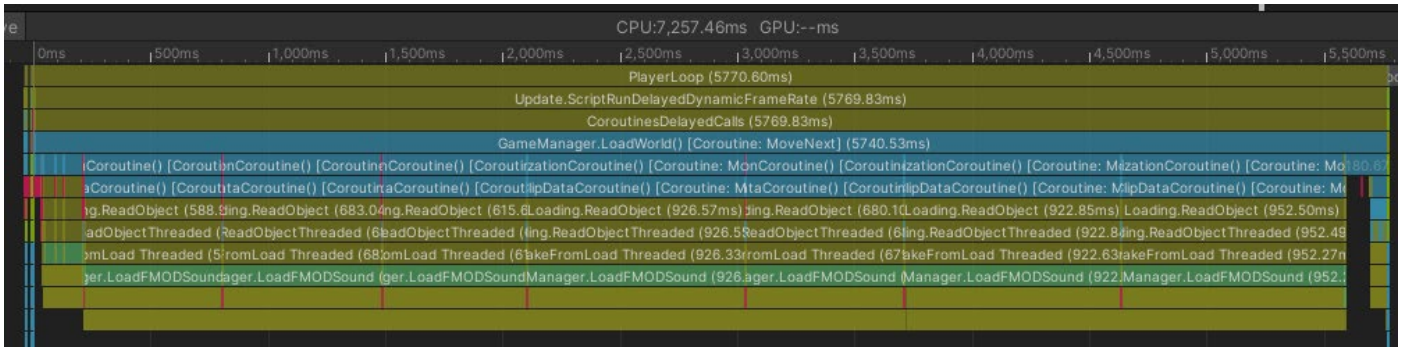
オーディオ

オーディオは、通常パフォーマンスのボトルネックにはありませんが、メモリ、ディスクスペース、CPU 使用率を節約するために最適化することはできます。

ロスレスファイルをソースとして使用する

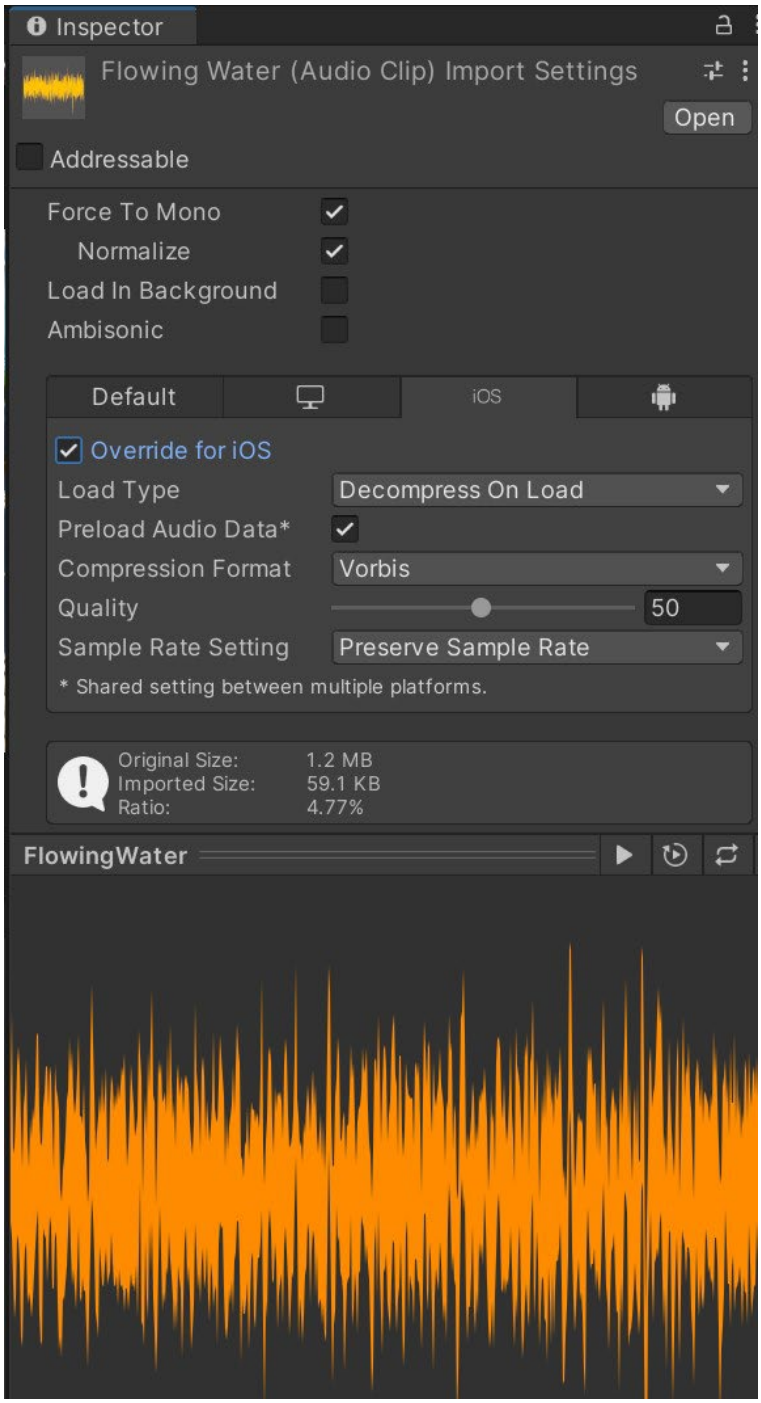
WAV や AIFF といった、ロスレスのファイル形式のサウンドアセットを使用することから始めましょう。

圧縮フォーマット（MP3 や Vorbis など）を使用する場合、Unity はビルド時にそれを解凍し、再圧縮します。この結果、不可逆なパスが 2 回発生し、最終的な品質が劣化してしまいます。



Unity Profiler でのオーディオ読み込みの CPU 使用率

オーディオクリップを縮小する



オーディオクリップのインポート設定

オーディオクリップのインポート設定により、ランタイム時のメモリ使用量を節約し、CPU パフォーマンスを最適化できます。

— ステレオサウンドを必要としない場合は、ステレオオーディオファイルの「**Force To Mono**」オプションを有効にすることで、ランタイム時のメモリとディスクの容量を節約できます。

  空間化された Audio Source には、モノラルでオーサリングされているか、インポート設定で「Force To Mono」が有効になっているオーディオクリップを使用してください。空間化された Audio Source でステレオサウンドを使用する場合、オーディオデータは 2 倍のディスクスペースとメモリを消費します。これにより、Unity はオーディオミキシング処理中にサウンドをモノラルに変換する必要があり、これにも余分な CPU 処理時間が必要となります。

— 「**Preload Audio Data**」は、Unity がシーンを初期化する前に、参照されているオーディオクリップをロードすることを確実にします。しかし、これによりシーンの読み込み回数が増えます。

— サウンドクリップがすぐに必要ではない場合は、非同期で読み込んでください。「**Load in Background**」にチェックを入れます。これは、メインスレッドをブロックすることなく、別スレッドでサウンドを遅延読み込みします。

— 「**Sample Rate Setting**」を「Optimize Sample Rate」または「Override Sample Rate」に設定します。

  モバイルプラットーフォームでは、22050Hz で十分です。44100Hz（CD 品質）は必要最小限の使用にとどめてください。48000Hz は過剰です。

  PC や コンソールのプラットフォームでは、44100Hz が理想です。通常、48000Hz は必要ありません。

— オーディオクリップを圧縮し、圧縮のビットレートを下げます。

  モバイルプラットーフォームの場合、大抵のサウンドには Vorbis（ループすることを想定していないサウンドには MP3）を使用します。短く、頻繁に使用されるサウンド（銃声や足音など）には ADPCM を使用します。

  PC と Xbox® では、Vorbis や MP3 の代わりに Microsoft XMA コーデックを使用してください。Microsoft は 8：1 から 15：1 までの圧縮率を推奨しています。

  Playstation では、ATRAC9 形式を使用してください。これは、Vorbis や MP3 よりも CPU オーバーヘッドが少なく済みます。

— 適切なロードタイプはクリップの長さによって異なります。

クリップサイズ	使用例	ロードタイプ設定
小 （200KB 未満）	騒々しい効果音（足音、銃声）、U サウンド	「**Decompress on Load**」を使用します。これは、サウンドを生の 16-bit PCM オーディオデータに解凍する際、わずかな CPU コストがかかりますが、ランタイム時に最も高いパフォーマンスを発揮します。 または、「**Compressed In Memory**」を使用し、「**Compression Format**」を「**ADPCM**」に設定します。これは、3.5：1 の固定圧縮率を提供し、リアルタイムかつ少ない負荷で解答できます。

中 （200KB 以上）	ダイアログ、短い音楽、中程度の音量または騒音の少ない効果音	最適なロードタイプは、プロジェクトの優先順位によって決まります。 メモリ使用量の節約が第一優先の場合は、「**Compressed In Memory**」を選択します。 CPU 使用量が心配なら、クリップを「**Decompress On Load**」に設定します。
大 （350 ～ 400KB 超）	BGM、周囲の雑音、長いダイアログ	「**Streaming**」に設定します。ストリーミングには 200KB のオーバーヘッドがあるため、十分に大きなオーディオクリップにのみ適しています。

AudioMixer を最適化する

オーディオクリップの設定に加え、AudioMixer における以下の問題にも注意してください。

— SFX Reverb Effect は、AudioMixer で最も負荷が高いオーディオ効果のひとつです。SFX Reverb を持つミキサーグループ（およびそれに送信するミキサーグループ）を追加すると、CPU コストが増加します。

これは、実際にグループに信号を送っている AudioSource がない場合でも起こります。Unity のオーディオシステムは、null 信号を受信しているかどうかを区別していません。

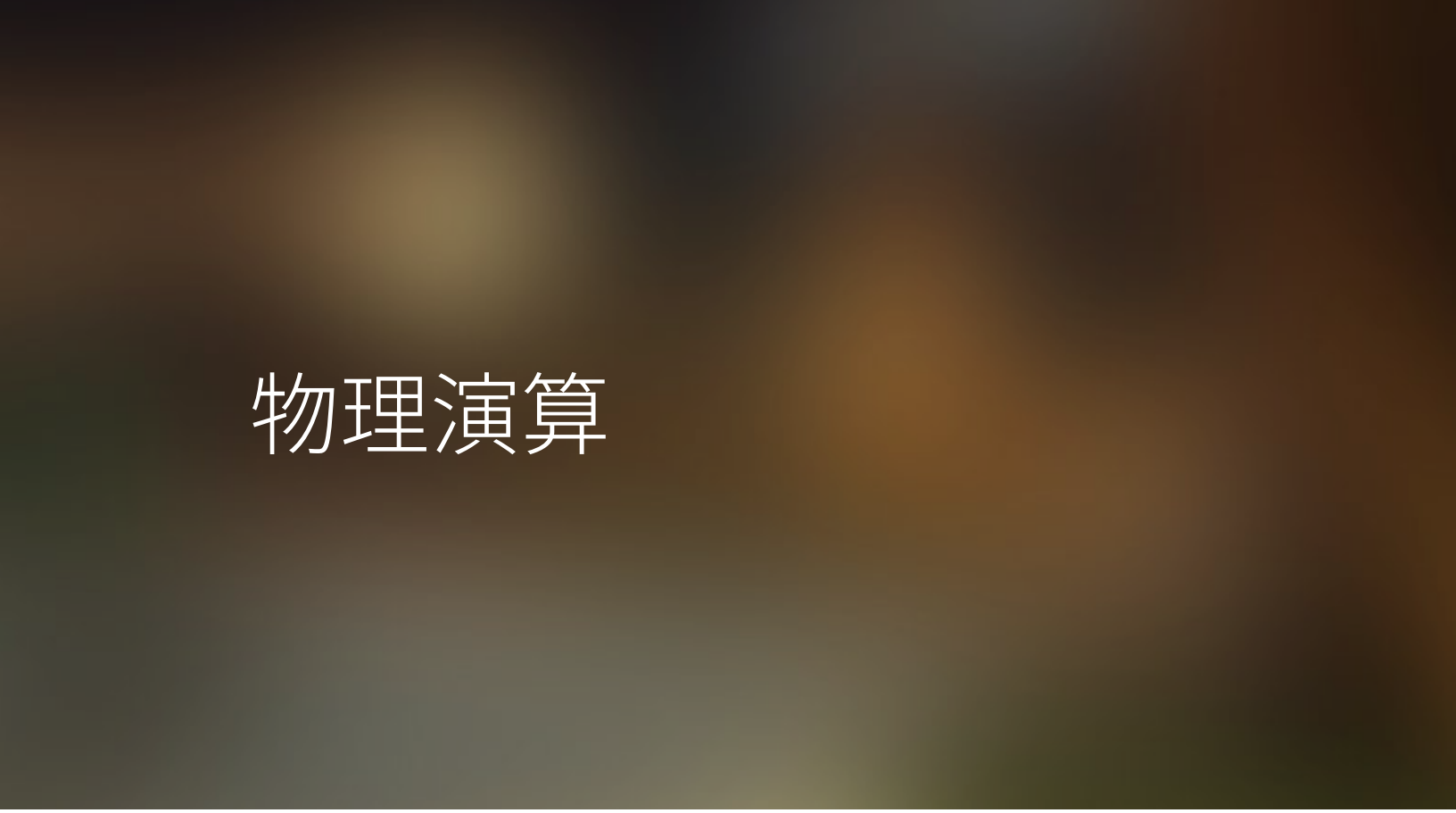
Reverb グループとそれに送信するグループを追加すると、たとえ AudioSource から書き込みがなくても、コストがかかる。

— AudioMixer のパフォーマンスを向上させるために、ミキサーグループの数を減らしてください。1 つの親グループの下に多数の子グループを追加すると、オーディオの CPU コストが大幅に増加します。Unity の DSP は null 信号を区別しないため、すべての AudioSource が Master に直接出力されていても、この現象は発生します。

子グループの数が多すぎる AudioMixer グループ

— 親グループに子グループが 1 つしかない状態をを回避してください。可能な限り、2 つのミキサーグループを 1 つにまとめます。

子グループを 1 つだけ持つ AudioMixer

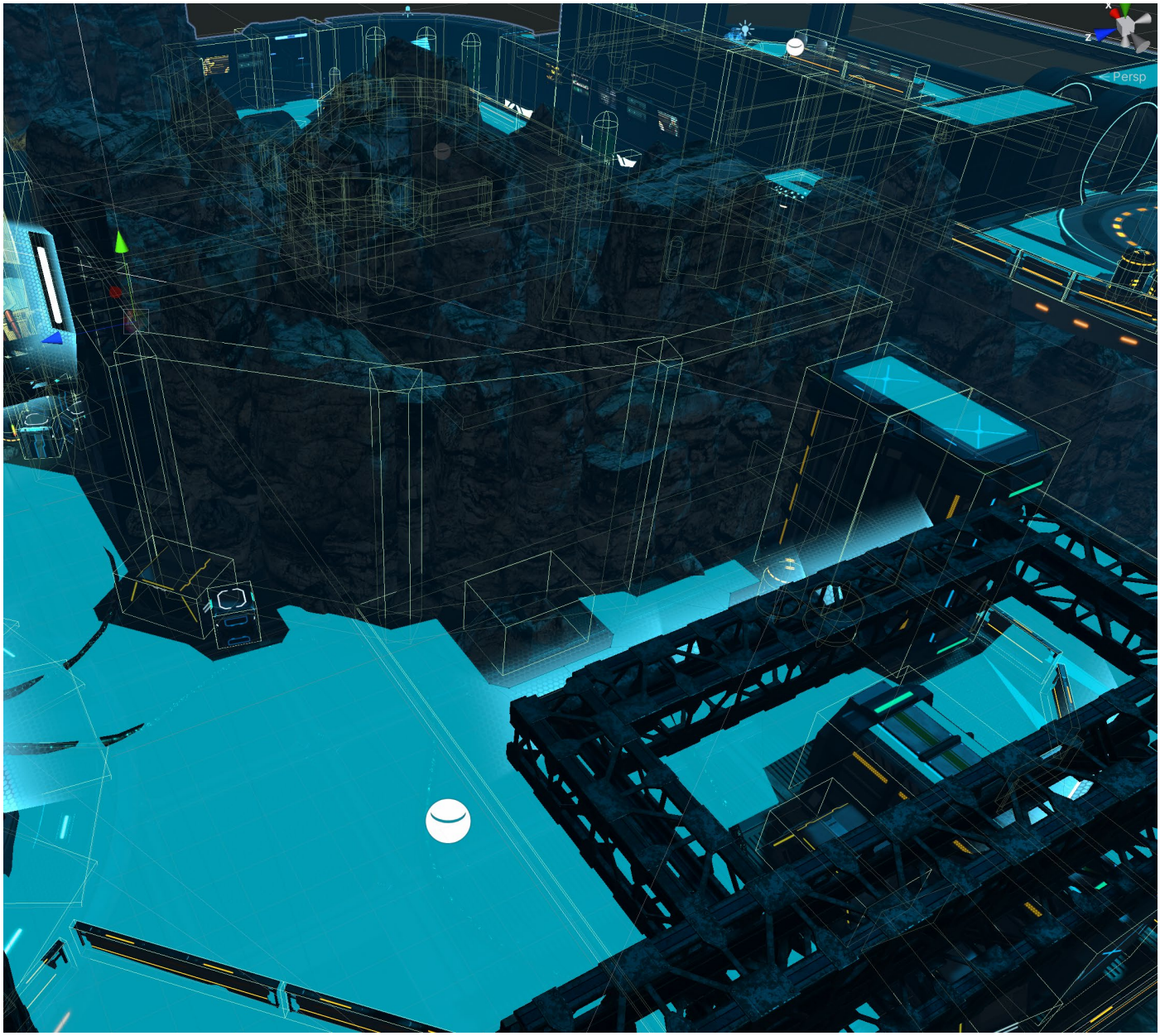


物理演算

物理学は複雑なゲームプレイを作り出すことが可能ですが、これにはパフォーマンス上のコストがかかります。これらのコストの正体がわかれば、シミュレーションを微調整して適切に管理することができます。以下のヒントを活用すると、ターゲットフレームレートを維持しながら、NVIDIA PhysX エンジンを統合した Unity の built-in 3D physics を使用してゲームを滑らかに再生できます。

コライダーを単純化する

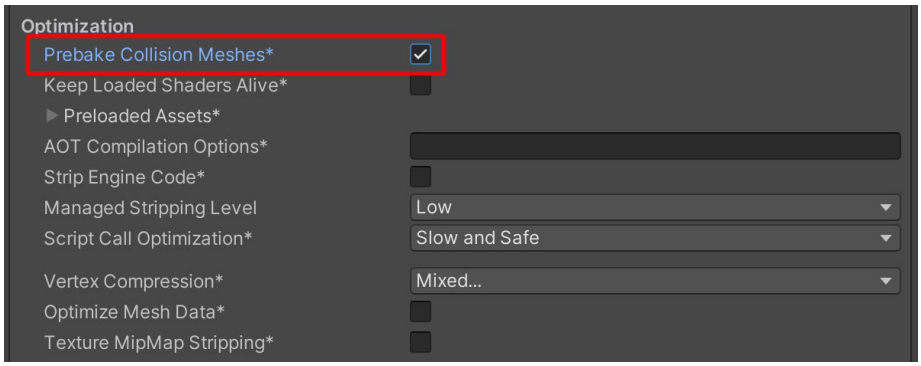
メッシュコライダーの負荷は高くなりえます。より複雑なメッシュコライダーをプリミティブまたは簡略化されたメッシュコライダーで代用し、元の形状に近づけてください。



コライダーにはプリミティブまたは簡略化したメッシュを使用する。

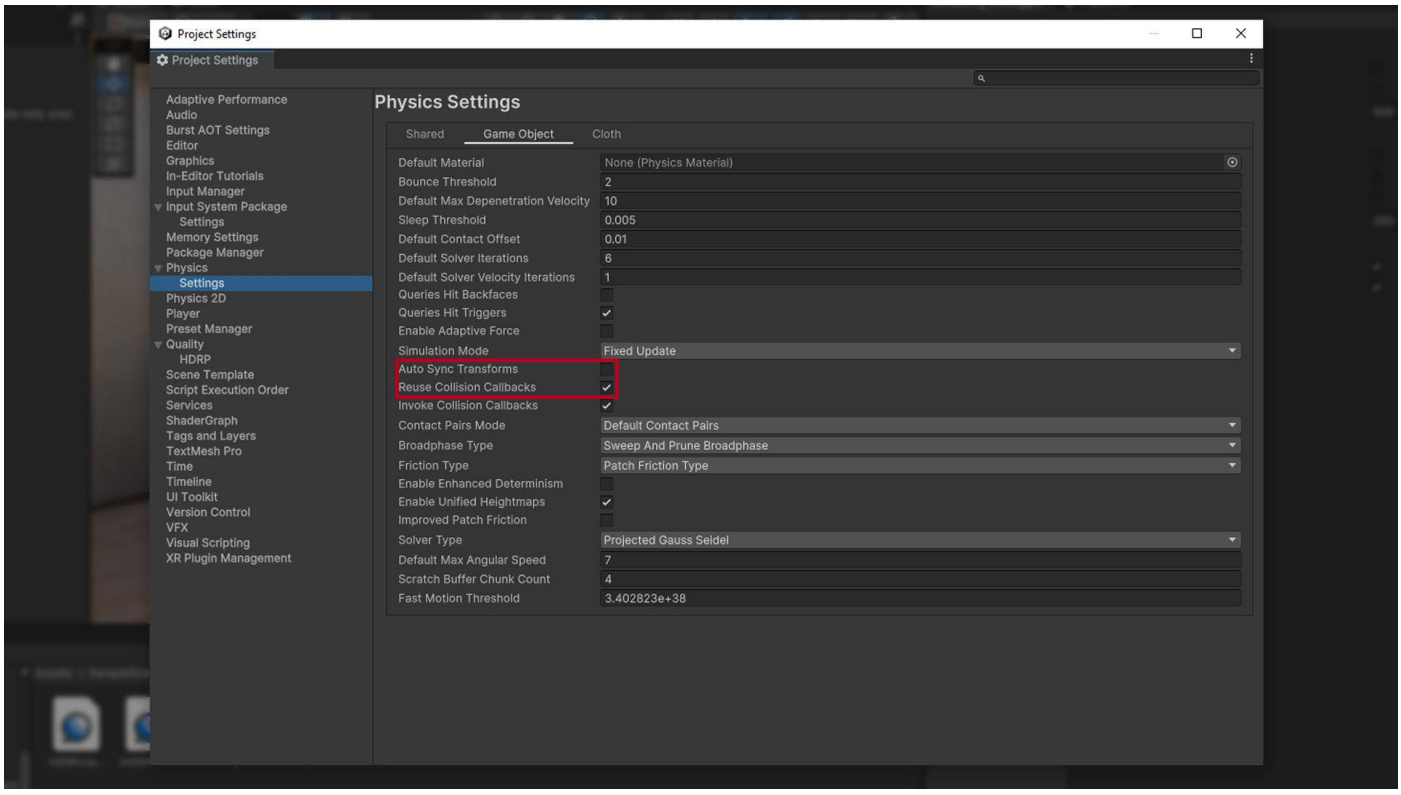
設定を最適化する

Player 設定で、「**Prebake Collision Meshes**」には可能な限りチェックを入れておきましょう。



Prebake Collision Meshes を有効にする。

Physics 設定（「**Project Settings**」>「**Physics**」）も忘れずに編集してください。「Layer Collision Matrix」を可能な限り簡素化します。

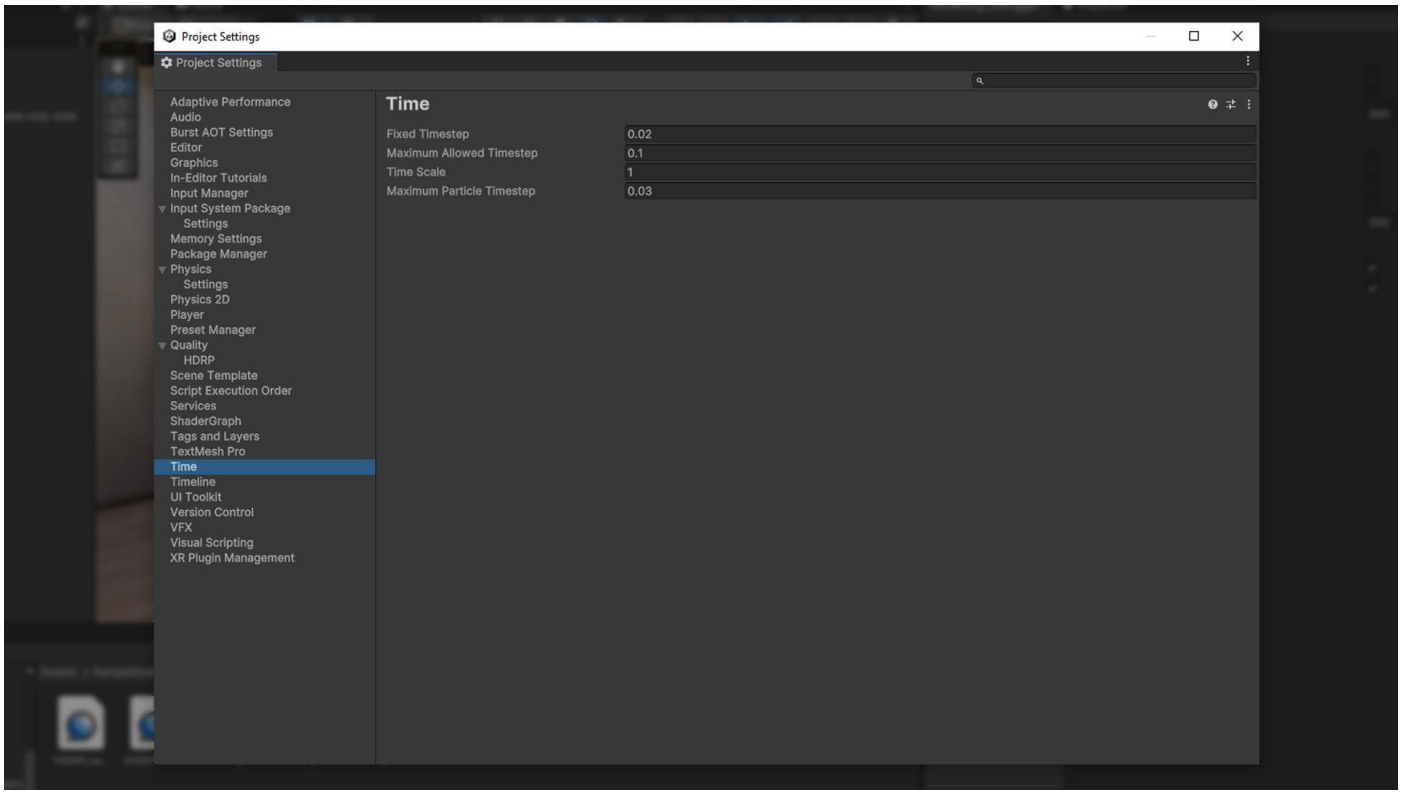


プロジェクトの物理演算設定を変更し、より高いパフォーマンスを引き出す。

シミュレーション頻度を調整する

物理演算エンジンは固定の時間ステップで動作します。プロジェクトの固定レートは、**「Edit」 > 「Project Settings」 > 「Time」**から確認できます。

「**Fixed Timestep**」フィールドは、各物理演算ステップで使用される時間の間隔を定義します。例えば、デフォルト値の 0.02 秒（20ms）は 50fps（50Hz）に相当します。



「Project Settings」の「Fixed Timestep」はデフォルトで 0.02 秒（1 秒あたり 50 フレーム）に設定されている。

Unity の各フレームにかかる時間はそれぞれ異なるため、物理演算シミュレーションと完全に同期しているわけではありません。エンジンは次の物理演算時間ステップまでカウントします。フレームがわずかに遅くなったり速くなったりした場合、Unity は経過時間を使用して、適切な時間ステップで物理演算シミュレーションを実行するタイミングを特定します。

フレームの準備に時間がかかる場合、パフォーマンス問題につながる可能性があります。例えば、ゲームにスパイクが発生した場合（多数のゲームオブジェクトのインスタンス化やディスクからのファイルのロードなど）、フレームの実行に 40ms 以上かかることがあります。デフォルトの 20ms の固定時間ステップでは、可変時間ステップに「追いつく 」ために、次のフレームで 2 つの物理シミュレーションが実行されることになります。

余分な物理演算シミュレーションは、フレームの処理時間を増やすことになります。ローエンド寄りのプラットフォームでは、これはパフォーマンスの下降スパイラルにつながる可能性があります。

次のフレームの準備に時間がかかると、物理演算シミュレーションのバックログも長くなります。このため、フレームがさらに遅くなり、1 フレームあたりに実行するシミュレーションの数がも増加します。その結果、パフォーマンスはますます低下します。

最終的には、物理演算の更新間隔が最大許容時間ステップを超える可能性があります。この制限を超えると、Unity は物理演算の更新を中断するようになり、ゲームにスタッターが発生してしまいます。

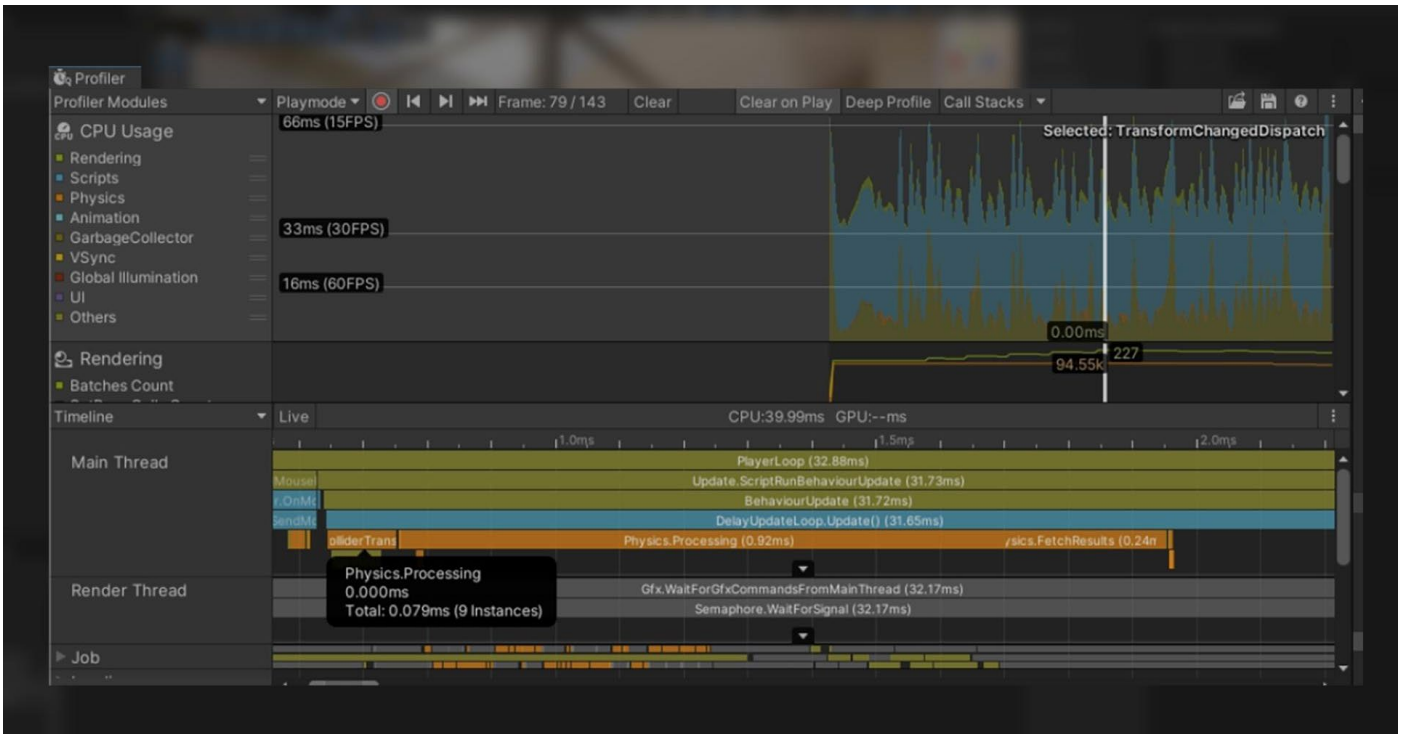
物理演算によるパフォーマンス問題を回避するには、以下を行ってください。

— シミュレーション頻度を下げます。ローエンド寄りのプラットフォームでは、**Fixed Timestep** をターゲットフレームレートよりもやや高めに設定してください。例えば、モバイルで 30fps の場合は 0.035 秒を使用します。これは、パフォーマンスの下降スパイラルを防ぐのに役立ちます。

— **Maximum Allowed Timestep** を下げます。より小さな値（0.1 秒など）を使用すると、物理演算シミュレーションの精度は多少落ちますが、1 フレームに起こる物理演算更新の回数も制限されます。異なる値を試して、プロジェクトの要件に合うものを見つけてください。

— 必要に応じて、物理演算ステップを手動で行いたい場合は、フレームの更新段階で SimulationMode を選択します。これにより、物理演算ステップを実行するタイミングをコントロールできます。Time.deltaTime を Physics.Simulate に渡すことで、物理演算をシミュレーション時間と同期させることができます。

この方法は、複雑な物理演算やフレーム時間が大きく変化するシーンでは、物理演算シミュレーションが不安定になる可能性があるため、使用には注意が必要です。



手動シミュレーションによる Unity シーンのプロファイリング

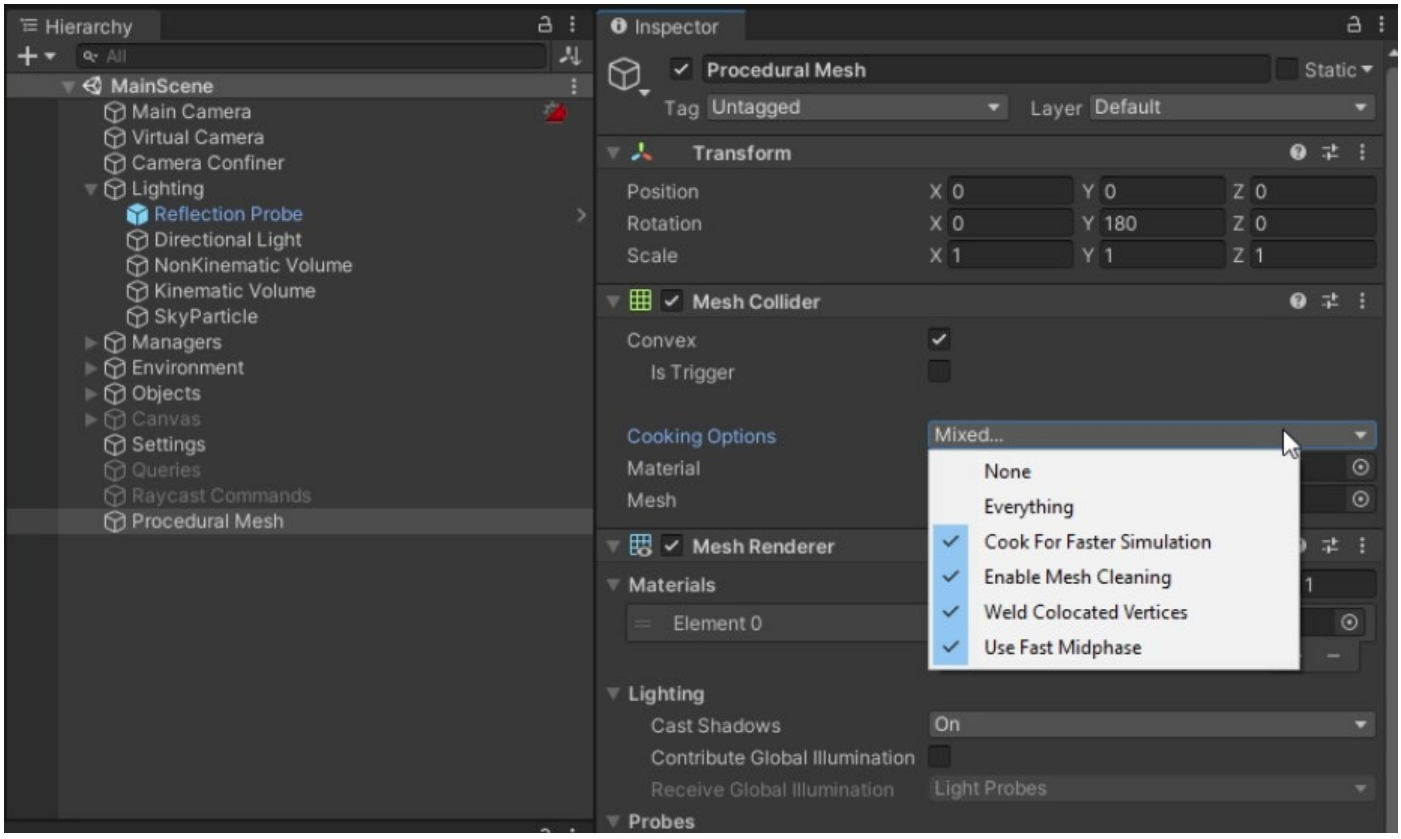
MeshCollider 向けに CookingOptions を変更する

物理演算で使用されるメッシュは、クッキングと呼ばれるプロセスを経ます。これは、レイキャストや接触などの物理演算クエリで動作するようにメッシュを準備します。

MeshCollider には、メッシュを物理的に検証するのに役立ついくつかの CookingOptions があります。メッシュにこれらの検証が必要ないと断定できる場合は、これらを無効にしてクッキング時間を短縮できます。

この場合は、各 MeshCollider の「CookingOptions」で、「EnableMeshCleaning」「WeldColocatedVertices」「CookForFasterSimulation」 のチェックを外してください。これらのオプションは、ランタイム時にプロシージャル生成されたメッシュには有効ですが、メッシュがすでに適切な三角形を持っている場合は無効にできます。

また、PC をターゲットにしている場合は、「Use Fast Midphase」 を有効にしておいてください。これは、シミュレーションの中間段階で PhysX 4.1 の高速アルゴリズムに切り替わります（物理演算クエリのために交差する可能性のある三角形の小さなセットを絞り込むのに役立ちます）。



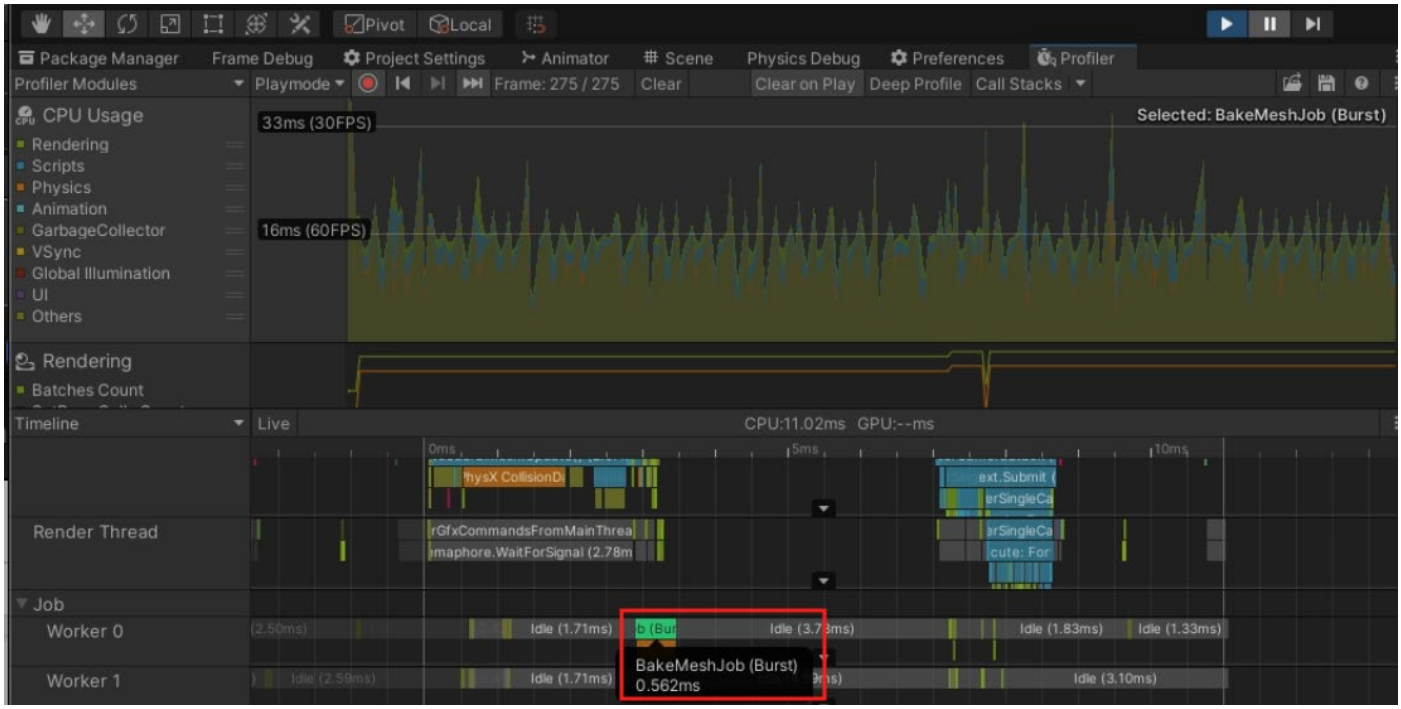
メッシュのクッキングオプション

Physics.BakeMesh を使用する

ゲームプレイ中にメッシュをプロシージャル生成する場合、ランタイム時に MeshCollider を作成することができます。しかし、MeshCollider コンポーネントを直接メッシュに追加すると、メインスレッドで物理演算をクック / ベイクします。これは相当な CPU 時間を消費する可能性があります。

Physics.BakeMesh を使用して MeshCollider と併せて使用するメッシュを準備し、ベイクデータをメッシュ自体に保存してください。このメッシュを参照する新しい MeshCollider は、（メッシュを再度ベイクするのではなく）このベイク済みデータを再利用します。これは、シーンのロード時間やインスタンス化の時間を短縮するのに役立ちます。

パフォーマンスを最適化するには、C# Job System を使用してメッシュのクッキングを別のスレッドにオフロードすることができます。複数のスレッドにまたがってメッシュをベイクする方法の詳細については、こちらの例を参照してください。



Profiler の BakeMeshJob

広大なシーンには Box Pruning を使用する

Unity の 物理演算エンジンは 2 つのステップで実行されます。

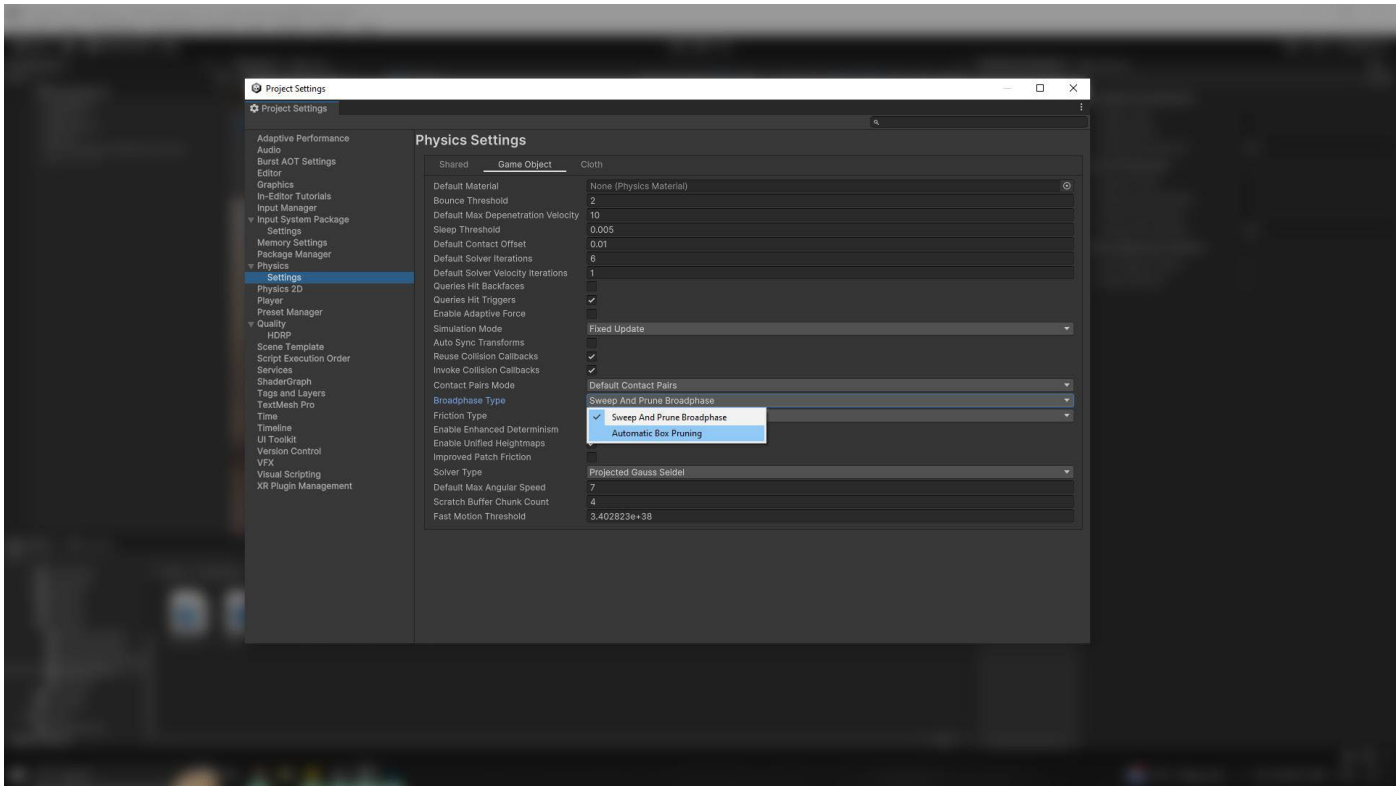
— **ブロードフェーズ**：スイープ&プルーンアルゴリズムを使用して潜在的な衝突を収集

— **ナローフェーズ**：エンジンが実際に衝突を計算

ブロードフェーズのデフォルト設定である「Sweep and Prune BroadPhase」（**「Edit」 > 「Project Settings」 > 「Physics」 > 「BroadPhase Type」**）は、一般的に平坦で、多くのコライダーがあるワールドに対して誤検出を発生させる可能性があります。

シーンが広大でほとんど平坦の場合は、この問題を避けて、「**Automatic Box Pruning**」または「**Multibox Pruning Broadphase**」に切り替えてください。これらのオプションはワールドをグリッドに分割し、各グリッドセルがスイープ&プルーンを実行します。

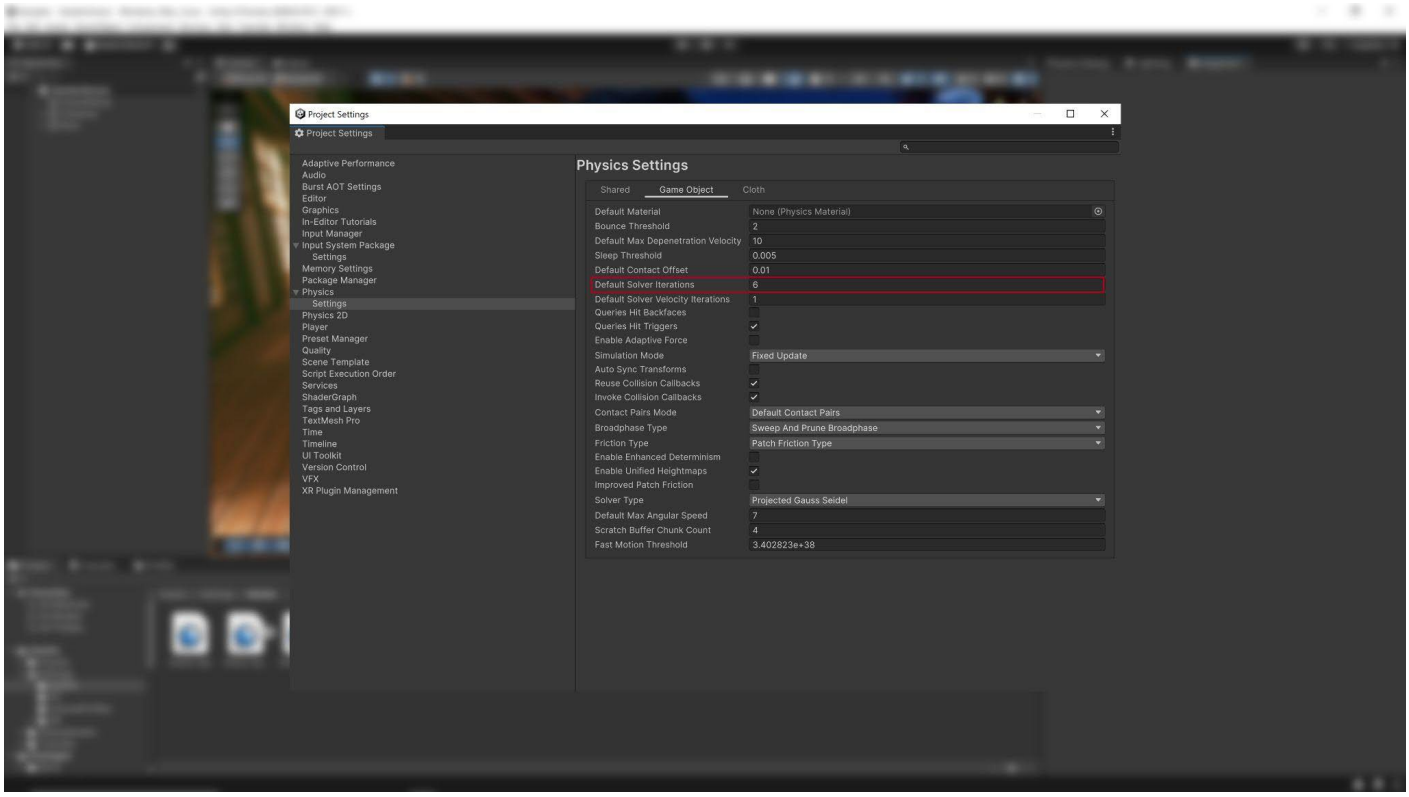
「Multibox Pruning Broadphase」では、ワールドの境界とグリッドセルの数を手動で指定できますが、「Automatic Box Pruning」ではそれを計算してくれます。



Physics オプションの Broadphase Type

ソルバーのイテレーションを変更する

特定の物理ボディのシミュレーションをより正確に行いたい場合は、そのボディの Rigidbody.solverIterations を増やします。



リジッドボディごとに Default Solver Iterations をオーバーライドする

これは Physics.defaultSolverIterations（**「Edit」>「Project Settings」>「Physics」>「Default Solver Iterations」**からもアクセス可能）をオーバーライドします。

物理演算シミュレーションを最適化するには、プロジェクトの defaultSolveIterations に比較的低い値を設定します。そして、より詳細な情報が必要な個々のインスタンスに、より高いカスタム Rigidbody.solverIterations 値を適用します。

自動トランスフォーム同期を無効にする

デフォルトでは、Unity は Transform の変更を物理演算エンジンと自動的に同期しません。代わりに、次の物理演算が更新されるまで、または手動で Physics.SyncTransforms が呼ばれるまで待機します。これを有効にすると、その Transform またはその子オブジェクトにある Rigidbody や Collider が物理エンジンと自動的に同期されます。

手動で同期すべきタイミング

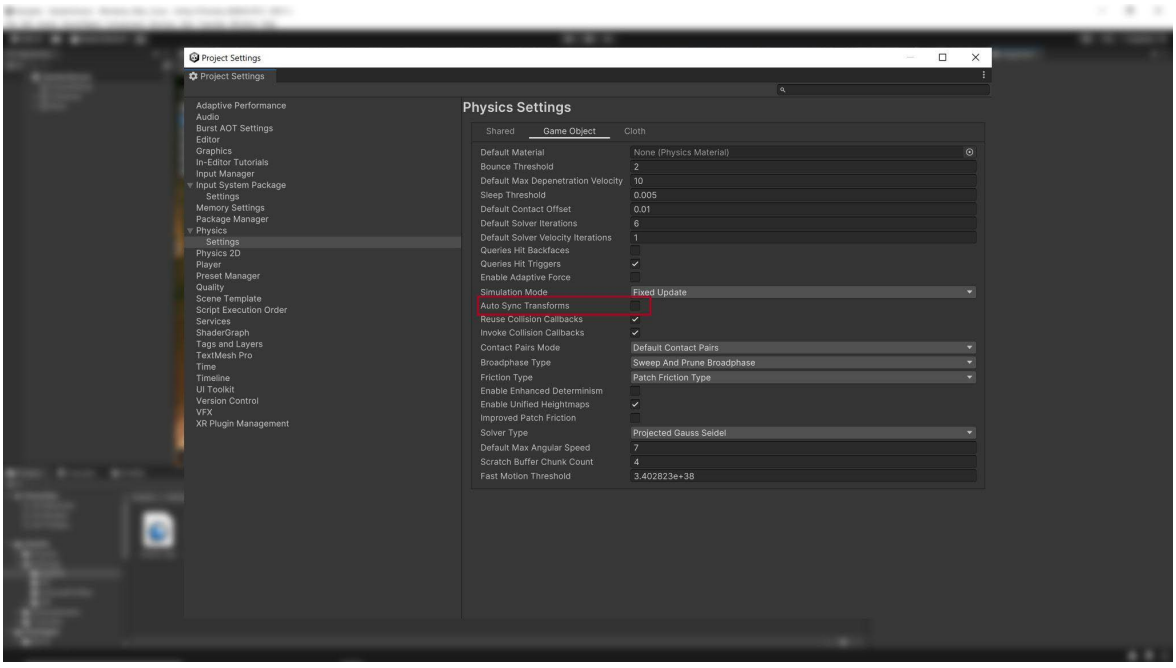
autoSyncTransforms が無効の場合、Unity は FixedUpdate で物理演算シミュレーションが行われる前、または Physics.Simulate で明示的に要求された場合にのみトランスフォームを同期します。Transform の変更と物理演算の更新の間に、物理演算エンジンから直接読み込む API を使用する場合は、さらに同期を行う必要があるかもしれません。この例として、**Rigidbody.position** にアクセスすることや、**Physics.Raycast** を実行することが挙げられます。

フォーマンスのベストプラクティス

autoSyncTransforms は最新の物理演算クエリを保証しますが、パフォーマンス上のコストは発生します。物理演算関連の API を呼び出すたびに同期が強制されるため、特に複数のクエリを連続して実行する場合は、パフォーマンスが低下する可能性があります。以下のベストプラクティスに従ってください。

— **必要な場合以外は autoSyncTransforms を無効にする**：ゲームメカニクス上、正確で継続的な同期が不可欠な場合にのみ有効にしてください。

— **手動で同期する**：パフォーマンスを向上させるには、最新の Transform データを必要とする呼び出しの前に、Physics.SyncTransforms() を使用して Transform を手動で同期させます。この方法は、グローバルに autoSyncTransforms を有効にするよりも効率的です。



Unity で Auto Sync Transform を無効にしてシーンをプロファイルする。

Contact Array を使用する

Contact Array は、衝突データ（接触）を配列形式で格納および管理する方法を提供します。つまり、衝突イベントごとに接触点の配列が生成され、それにアクセスして処理することができます。配列であるため、連続したメモリブロックを提供し、衝突データの処理時のアクセス速度を向上させます。また、バッチ処理に適した形でデータを準備でき、パフォーマンスが求められる用途では C# Job System と組み合わせて使用することが可能です。

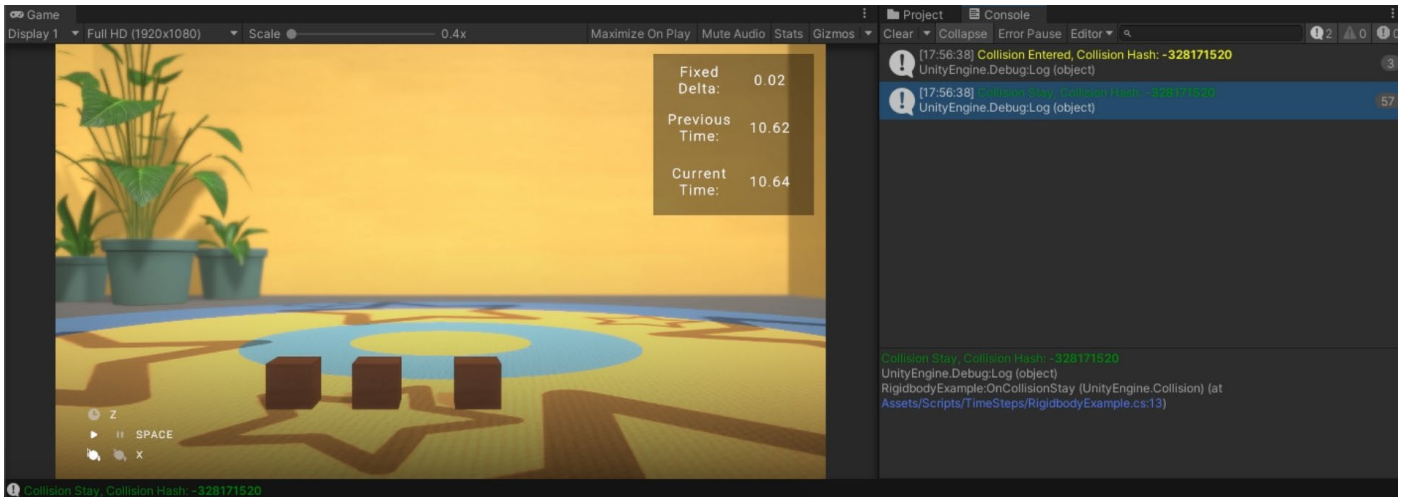
Reuse Collision Callbacks を有効にする

接触配列は一般的にかなり高速であるため、通常は衝突コールバックを再利用するよりも接触配列を使用することが推奨されます。ただし、特定の用途がある場合は以下の点を考慮してください。

MonoBehaviour.OnCollisionEnter、MonoBehaviour.OnCollisionStay、および MonoBehaviour.OnCollisionExit のコールバックはすべて、衝突インスタンスをパラメーターとして受け取ります。この衝突インスタンスはマネージヒープ上に割り当てられ、ガベージコレクトされなければなりません。

ガベージコレクトされる量を減少させたい場合は、Physics.reuseCollisionCallbacks（**「Projects Settings」>「Physics」>「Reuse Collision Callbacks」**からもアクセス可能）を有効にします。これを有効にすると、Unity は各コールバックに 1 つの衝突ペアインスタンスだけを割り当てます。これにより、ガベージコレクターへの無駄な負荷が減り、パフォーマンスが向上します。

パフォーマンス向上のため、一般的には「Reuse Collision Callbacks」を常に有効にすることが推奨されます。この機能を無効にするのは、コードが個々の Collision クラスのインスタンスに依存しており、個々のフィールドを保存するのが非現実的なレガシープロジェクトに限ります。



Unity コンソールでは、Collision Entered と Collision Stay に 1 つの衝突インスタンスがある。

静的コライダーを動かす

静的コライダーは、Collider コンポーネントを持ち、Rigidbody を持たないゲームオブジェクトです。

「静的」という言葉とは反対に、静的コライダーは動かすことができます。これは、物理演算ボディの位置を変更するだけで行えます。位置の変化を累積し、物理演算更新の前に同期します。静的コライダーを動かすためだけに、Rigidbody コンポーネントを追加する必要はありません。

しかし、静的コライダーを他の物理演算ボディともっと複雑な方法で相互作用させたい場合は、Kinematic Rigidbody を追加してください。Transform コンポーネントにアクセスする代わりに、Rigidbody.position と Rigidbody.rotation を使用して動かしてください。これにより、物理演算エンジンの動作が予測しやすくなります。

注：もし、ランタイム時に個々の Static Collider 2D を移動したり、再設定する必要がある場合は、Rigidbody 2D コンポーネントを追加し、「Body Type」を「Static」に設定します。これは、Rigidbody 2D がある場合に Collider 2D のシミュレーションをより速く行えるためです。ランタイム時に Collider 2D のグループを移動させたり、再構成する必要がある場合、それぞれのゲームオブジェクトを個別に移動させるよりも、非表示になっている 1 つの親 Rigidbody 2D の子にした方が速くなります。

非割り当てクエリを使用する

3D プロジェクトで特定の距離と方向にあるコライダーを検出し収集するには、レイキャストや BoxCast のような他の物理演算クエリを使用します。

OverlapSphere や OverlapBox のように、複数のコライダーを配列として返す Physics クエリは、それらのオブジェクトをマネージヒープ上に割り当てる必要があることに注意してください。これは、ガベージコレクターが最終的に割り当てられたオブジェクトを回収する必要があることを意味し、それが間違ったタイミングで起こるとパフォーマンスが低下する可能性があります。

このオーバーヘッドを減らすには、これらのクエリの **NonAlloc** バージョンを使用します。例えば、ある地点の周りにあるすべての考えらるコライダーを収集するために OverlapSphere を使用している場合は、代わりに OverlapSphereNonAlloc を使用してください。

これにより、バッファとして機能するコライダーの配列（results パラメーター）を渡すことができます。NonAlloc メソッドはガベージを発生させずに動作します。それ以外の点では、対応する割り当てメソッドと同じように機能します。

NonAlloc メソッドを使用する際は、十分なサイズの結果バッファを定義する必要があることに注意してください。バッファの容量が不足した場合、自動的に拡張されることはありません。

2D 物理演算

上記のヒントは 2D 物理演算クエリには当てはまりません。これは、Unity の 2D 物理演算システムでは、メソッドに「NonAlloc」というサフィックスがないためです。その代わりに、複数の結果を返すメソッドを含むすべての 2D 物理演算メソッドは、配列またはリストを受け付けるオーバーロードを提供します。

例えば、3D 物理演算システムには RaycastNonAlloc のようなメソッドがありますが、2D では、以下のように、単に配列または List<T> をパラメーターとして受け取る Raycast のオーバーロード版を使います。

```
var results = new List<RaycastHit2D>();
int hitCount = Physics2D.Raycast(origin, direction, contactFilter, results);
```

オーバーロードを使用することで、特別な NonAlloc メソッドを必要とせずに、2D 物理演算システムで非割り当てクエリを実行することができます。

レイキャストのクエリをバッチ処理する

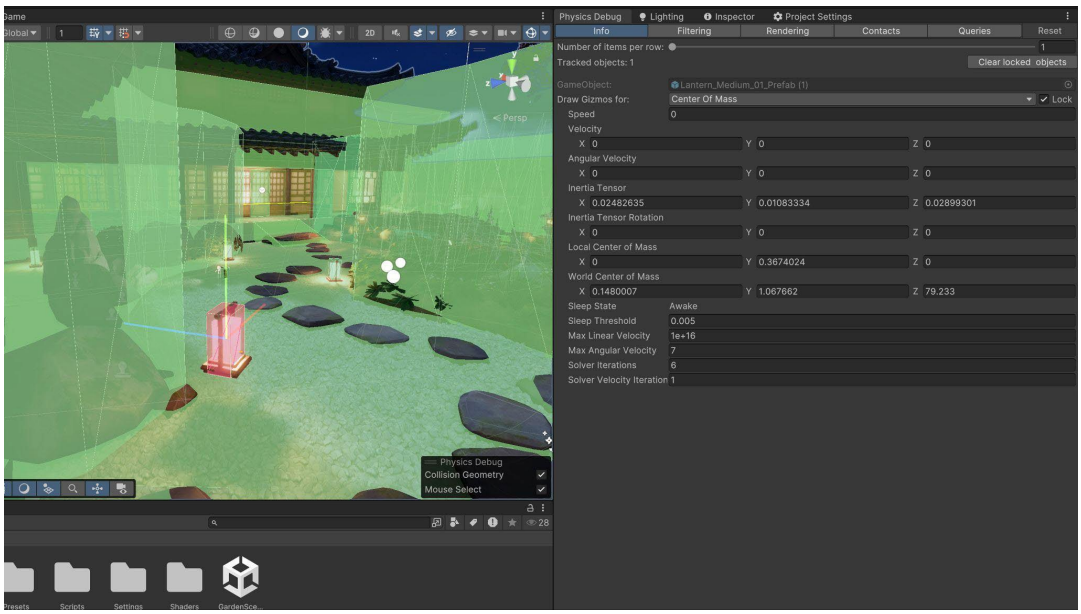
Physics.Raycast を用いてレイキャストクエリを実行できます。しかし、レイキャスト操作の数が多い場合（エージェント 10,000 体の視線の計算など）、CPU 時間が大幅に消費される可能性があります。

C# Job System で RaycastCommand を使用してクエリをバッチ処理します。これにより、メインスレッドから作業をオフロードし、レイキャストを非同期かつ並列に行うことができます。

RaycastCommands のドキュメントページから使用例をご覧ください。

Physics Debugger を使って視覚化する

Physics Debug ウィンドウ（**「Window」 > 「Analysis」 > 「Physics Debugger」**）を使用して問題のあるコライダーや不一致のトラブルシューティングに役立てることができます。これは、互いに衝突する可能性のあるゲームオブジェクトを色分けして表示します。



Physics Debugger では、物理演算オブジェクトがどのように相互作用するかを視覚的に確認できる。

詳細については、Physics Debugger のドキュメントをご覧ください。

アニメーション

以下のヒントは、Unity でアニメーションを扱う際に役立ちます。アニメーションシステムに関する包括的なガイドは、無料の eBook「*Unity におけるアニメーション決定版ガイド*」をダウンロードしてください。

ヒューマノイドリグではなく、ジェネリックリグを使用する

デフォルトでは、Unity はジェネリックリグでアニメーションが付いたモデルをインポートしますが、開発者はキャラクターにアニメーションを付ける際にヒューマノイドリグに切り替えることがよくあります。リグにおける問題を把握しておきましょう。

- 可能な限りジェネリックリグを使用します。ヒューマノイドリグは、使用されていないときも、フレームごとにインバースキネマティクスとアニメーションのリターゲティングを計算します。そのため、同等のジェネリックリグに比べ、30 ～ 50% 多くの CPU 時間を消費します。
- ヒューマノイドニメーションをインポートする際、IK ゴールや指のアニメーションが不要な場合は、アバターマスクを使って削除してください。
- 可能な限り低いボーンウェイト数を保ちます。

ジェネリックリグはヒューマノイドリグよりも CPU の使用時間が短い。

シンプルなアニメーションには代替手段を使用する

Animator の機能は、主にヒューマノイドキャラクターを対象としています。しかし、単一の値（UI 要素のアルファチャンネルなど）にアニメーションを付けるために再利用されることが多くあります。Animator の過剰な使用、特に、UI 要素との併用は避けてください。余分なオーバーヘッドが発生する原因となります。

現在のアニメーションシステムは、アニメーションのブレンディングや、より複雑な設定に最適化されています。ブレンディングに使用される一時的なバッファがあり、サンプリングされたカーブやその他のデータの複製が発生します。

静的なアニメーション、特に群衆のように重複が多いものについては、ベイク処理されたアニメーションを実装してみるとよいでしょう。一般的な手法は 2 つあります。頂点テクスチャアニメーション（法線と接線バッファを含む）と、頂点シェーダーで手動でスキニングするためにベイクされたボーン行列を使用する方法です。

アニメーションをベイクする方法についてさらに学びたい場合は、こちらの Unity Learn チュートリアルと Llam Academy によるこちらのチュートリアルをご覧ください。

また、可能ならば、アニメーションシステムをまったく使わないことも検討してみましょう。イージング関数を作成するか、可能であればサードパーティ製のトゥイーンライブラリ（DOTween など）を使用してみてください。これらを使うと、数学的な表現で非常に自然な補間を実現できます。

スケールカーブの使用を避ける

スケールカーブのアニメーションは、平行移動カーブや回転カーブのアニメーションよりも負荷がかかります。パフォーマンスを向上させるには、スケールのアニメーションを避けてください。

注：これは、定数カーブ（アニメーションクリップの長さにわたって同じ値を持つカーブ）には適用されません。定数カーブは最適化されており、通常のカーブよりも負荷がかかりません。

可視状態の場合のみ更新する

Animator の「Culling Mode」を「Based on Renderers」に設定し、「Skinned Mesh Renderer」の「Update When Offscreen」プロパティを無効にします。これにより、Unity が可視状態でないキャラクターのアニメーションを更新せずに済みます。

ワークフローを最適化する

その他の最適化はシーンレベルで行うことができます。

- Animator へのクエリの送信に、文字列の代わりにハッシュを使用する。
- 小規模の AI Layer を実装して Animator をコントロールする。OnStateChange、OnTransitionBegin、その他のイベントに対するシンプルなコールバックを提供させることが可能。
- State タグを使用して、AI のステートマシンを Unity のステートマシンに簡単に一致させる。
- イベントのシミュレーションを行うために追加のカーブを使用する。
- アニメーションをマークアップするために、追加カーブを使用する（例：ターゲットマッチングと組み合わせて使用する場合）。

アニメーションの階層を分ける

アニメーションを付ける階層が共通の親を共有しないようにしてください（その親がシーンのルートである場合を除く）。階層を分けることで、アニメーションの結果をゲームオブジェクトに書き戻す際に、重大なパフォーマンス低下を引き起こすスレッドの問題を防ぐことができます。

バインディングコストを最小化する

アニメーションシステムにおけるバインディング操作には高い負荷がかかります。パフォーマンスを最適化するには、頻繁にクリップを追加したり、ゲームオブジェクトやコンポーネントを追加または削除したり、ランタイム中にオブジェクトを有効または無効にしたりといった、一般的に再バインディングが必要となる操作は避けてください。これらの操作はすべて高い計算負荷がかかります。

深い階層でのコンポーネントベースのコンストレイントの使用を避ける

複雑な構造を持つキャラクターなど、深い階層にコンポーネントベースのコンストレイントを使用することは、パフォーマンスが低下する可能性があるため避けてください。

アニメーションのリギングがパフォーマンスに与える影響を考慮する

アニメーションのリギングを使用する場合は、各コンストレイントによって生じるパフォーマンスのオーバーヘッドに注意してください。ヒューマノイドモデルを扱う場合は、これを考慮することが重要です。可能な限り、パフォーマンスを向上させるために、ヒューマノイドリグのビルトイン IK（インバースキネマティクス）パスを活用してください。

ワークフローとコラボレーション

Unity でのアプリケーションの構築は、多くの開発者を巻き込む大掛かりな作業です。プロジェクトがチームにとって最適に設定されていることを確認してください。

バージョン管理を使う

バージョン管理システム（VCS）を活用すると、プロジェクト全体の履歴を保存できます。作業内容を整理し、チームが効率的にイテレーションを行うことも可能にします。

プロジェクトファイルは、リポジトリと呼ばれる共有データベースに保存されます。定期的にプロジェクトをリポジトリにバックアップすることで、何か問題が発生した際、プロジェクトを以前のバージョンに戻すことが可能になります。

VCS を用いると、個別の変更を複数行い、バージョニング用にそれらを 1 つのグループとしてコミットできます。このコミットは、プロジェクトのタイムライン上の一点として位置づけられます。以前のバージョンに戻す必要がある場合、そのコミット以降の変更はすべて元に戻され、プロジェクトがその時点の状態に復元されます。コミット内でグループ化された各変更をレビューして修正することも、コミットを完全に取り消すことも可能です。

プロジェクト全体の履歴にアクセスできるため、どの変更によってバグが発生したかを特定したり、過去に削除された機能を復元したり、ゲームや製品のリリース間の変更を簡単にドキュメントにまとめたりできます。

さらに、バージョン管理はクラウドまたは分散サーバーに保存されることが多いため、開発チームがどこで作業していてもコラボレーションをサポートすることができます。リモートワークが一般化している今、このメリットはより重要なものになっています。

バージョン管理のマージに役立つように、Editor 設定で **Asset Serialization の Mode** が「**Force Text**」に設定してあることを確認してください。これはスペース効率に劣りますが、Unity がシーンファイルをテキストベースで保存するようになります。

Asset Serialization の Mode

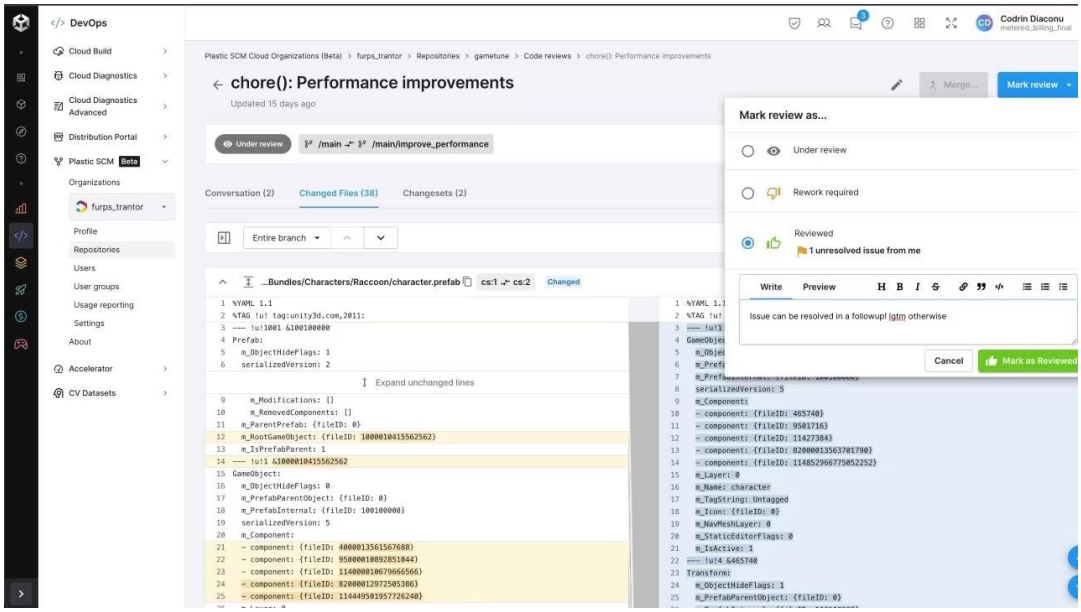
外部のバージョン管理システム（Git など）を使用している場合は、Version Control 設定で Mode が「Visible Meta Files」に設定されていることを確認してください。

Version Control の Mode

Unity にはシーンやプレハブをマージするための ビルトイン YAML（人間にも解読可能なデータシリアライズ言語）ツールが用意されています。詳しくは、Unity ドキュメント「スマートマージ」を参照してください。

Unity Version Control

ほとんどの Unity プロジェクトには、スクリプトコードに加えて、相当な量のアートアセットが含まれています。これらのアセットをバージョン管理で管理したい場合は、Unity Version Control（UVCS、旧称：Plastic SCM）への切り替えを検討してください。Git LFS を使っても、大きなバイナリファイル（500MB 超）を扱うときに優れた速度を発揮する Plastic SCM よりも、大きなリポジトリでは Git のパフォーマンスは劣ります。



Unity ダッシュボードの Unity Version Control のウェブインターフェース

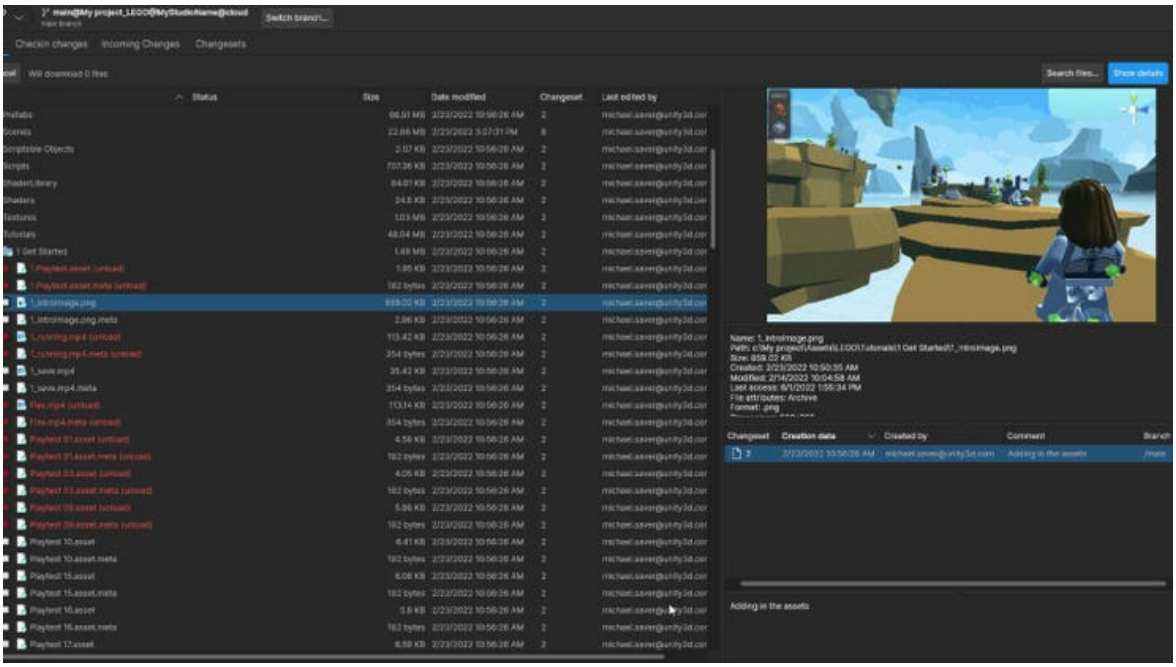
UVCS は、プログラマーやアーティストをサポートする独自のインターフェイスを備えた柔軟性の高いバージョン管理システムです。大規模なリポジトリやバイナリファイルの扱いに優れており、ファイルベースと変更セットベースのソリューションとして、プロジェクトビルド全体ではなく、作業中の特定のファイルのみをダウンロードできます。

UVCS にアクセスする方法は 3 つあります。UVCS デスクトップクライアントを使用して複数のアプリケーションやリポジトリにアクセスする、Unity Hub を通じてプロジェクトに追加する、またはウェブブラウザから Unity Cloud 上のリポジトリにアクセスする方法です。

UVCS は以下を実現します。

- アートアセットが安全にバックアップされていることを確認しながら作業する。
- 各アセットの所有権を追跡する。
- アセットを以前のバージョンに戻す。
- 単一の中央リポジトリで自動化されたプロセスを実行する。
- 複数のプラットフォームで迅速かつ安全にブランチを作成する。

さらに、UVCS は優れた可視化ツールで開発の一元化を支援します。特にアーティストは、開発チームとアートチームの統合をより密にすることを促進する Gluon アプリケーションのユーザーフレンドリーなワークフローを評価するでしょう。このアプリケーションを使用すると、プロジェクト全体のリポジトリを複雑に扱うことなく、必要なファイルだけを簡単に確認および管理できるようになります。簡素化されたワークフローを提供するだけでなく、アセットのバージョンの違いを視覚的に確認でき、統一されたバージョン管理環境への貢献を容易にするツールも提供しています。



UVCS は大規模なアセットでもアーティストにとって親しみやすい UI を提供する。

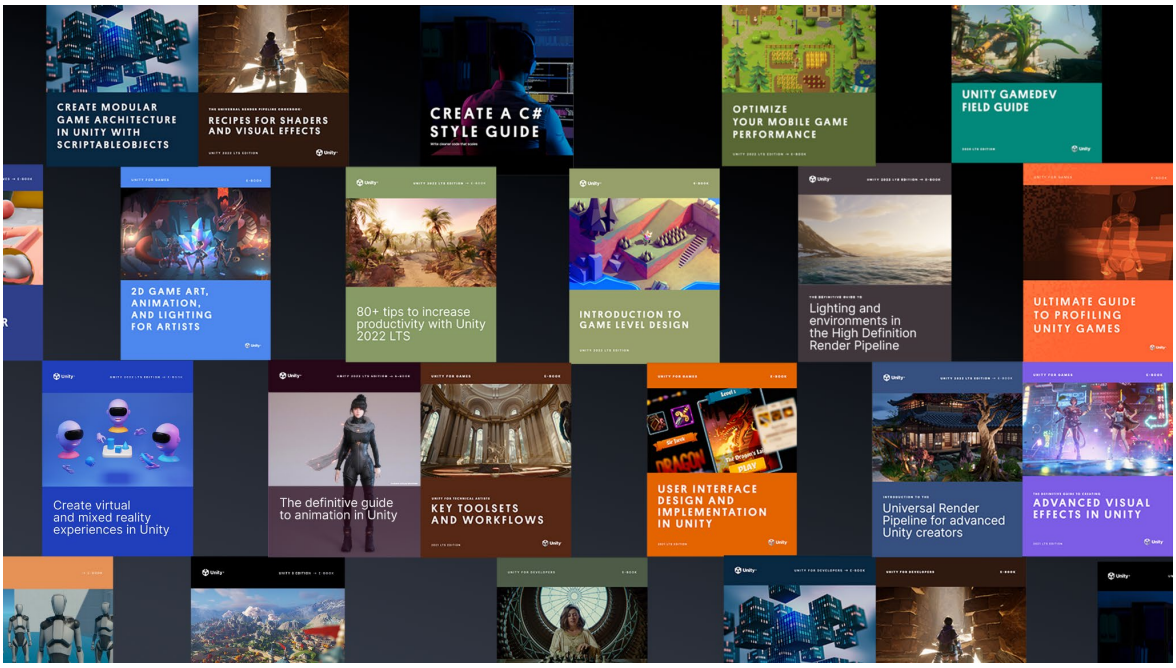
Unity Version Control を使い始めるにあたり、eBook「*プロジェクト整理とバージョン管理のベストプラクティス ー Unity 6版*」をご覧ください。

大きなシーンを分割する

Unity 単体の大きなシーンは、コラボレーションに適していません。アーティストやデザイナーが 1 つのレベルでより効果的に協力し、競合のリスクを最小限に抑えることができるよう、レベルをより小さなシーンに分割しましょう。

ランタイム時には、**SceneManager.LoadSceneAsync** を使用し、**LoadSceneMode.Additive** パラメーターを渡すことで、シーンを加算的にロードできます。

上級の開発者およびアーティスト向けのリソース



Unity ベストプラクティスハブでは、さらに多くの上級 Unity 開発者およびクリエイター向け eBook をダウンロードできます。業界の専門家や Unity のエンジニア、テクニカルアーティストによって作成された 30 以上のガイドの中からお選びください。ゲーム開発のベストプラクティスを紹介し、Unity のツールセットやシステムを使った効率的な開発に役立ちます。

また、Unity ブログ、Unity コミュニティフォーラム、Unity Learn、や **#unitytips** ハッシュタグから、ヒントやベストプラクティス、ニュースを見つけることができます。


Unity®