→ EBOOK

Unity におけるモバイル、XR、ウェブのパフォーマンス最適化

Unity®

Contents

はじめに

本ガイドでは、Unity 6 のためのモバイル、XR、Unity ウェブのパフォーマンスを最適化するための最適かつ最新のヒントをまとめています。本ガイドは、2 つの最適化ガイドのうちの 1 つで、もう 1 つは *Unity におけるコンソールおよび PC のパフォーマンス最適化*です。

モバイル、XR、Unity ウェブ アプリケーションの最適化は、ゲームの開発サイクル全体を支える不可欠なプロセスです。ハードウェアが進化し続けるなかで、ゲームの最適化は、アート、ゲームデザイン、オーディオ、および収益化戦略と並んで、プレイヤーのエクスペリエンスを形成する上での重要な役割を果たします。

モバイル、XR、ウェブゲームは数十億もの活発なユーザー基盤があります。モバイルの場合、ゲームが高度に最適化されていれば、そのゲームが各プラットフォームのストアでの認証へ合格するチャンスも向上します。ローンチ時、そしてその後に、成功の機会を最大限に引き出すために、最多のデバイスで高パフォーマンスのアプリケーションを目指します。

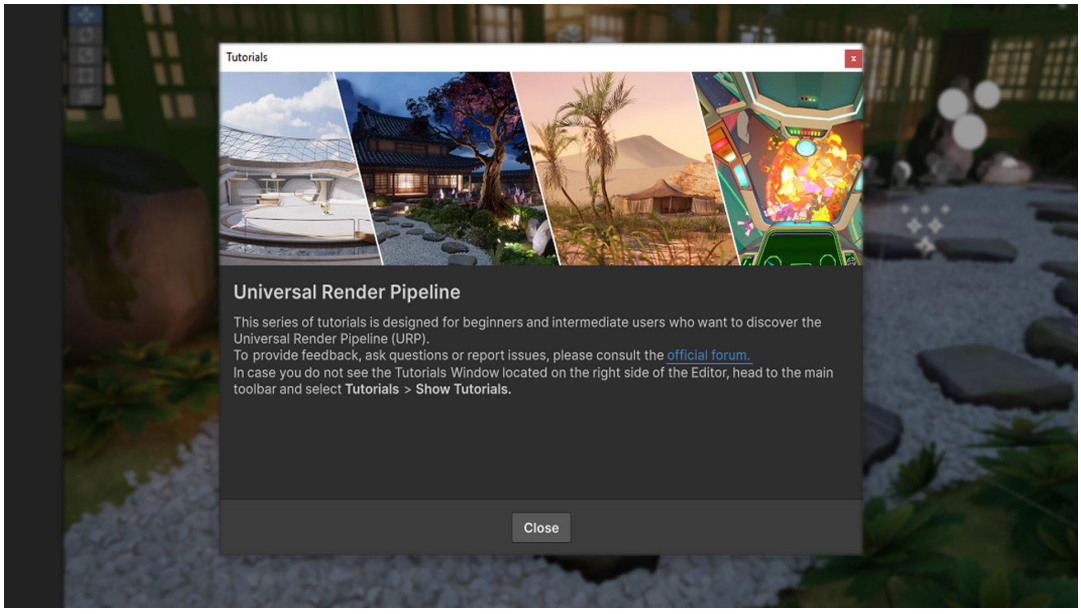
この eBook では、業界のさまざまな開発者と連携し、最高品質のゲームをローンチできるよう支援してきた Unity のエンジニアの知識とアドバイスをまとめています。

Unity チームのアドバイスを参考に、効果的な最適化を実現しましょう。[1]

[1] なお、ここで説明している最適化の多くは、プログラムをより複雑なものにする可能性があることに注意してください。これにより、追加のメンテナンスや潜在的なバグが発生する可能性があります。これらのベストプラクティスを実装するにあたっては、パフォーマンスの向上と、時間や人件費とのバランスを考慮するようにしてください。

パフォーマンスと ビジュアルの品質のために URP を選択する

XR（extended reality）、ウェブ、モバイルのゲーム、およびアプリケーションの開発において、Unity は ユニバーサルレンダーパイプライン（URP）を推奨します。URP は高パフォーマンスおよびスケーラビリティのために設計されており、広範囲のハードウェアに適応することができる効率的なレンダリングを提供します。より良いビジュアル品質を達成可能にしつつスムーズなパフォーマンスを維持するので、WebGL やモバイルデバイスなどのリソース効率が重要なプラットホームにとっては最適です。加えて、URP によってカスタマイズしやすくなるため、さまざまな環境下においてアプリケーションが最適に動作します。

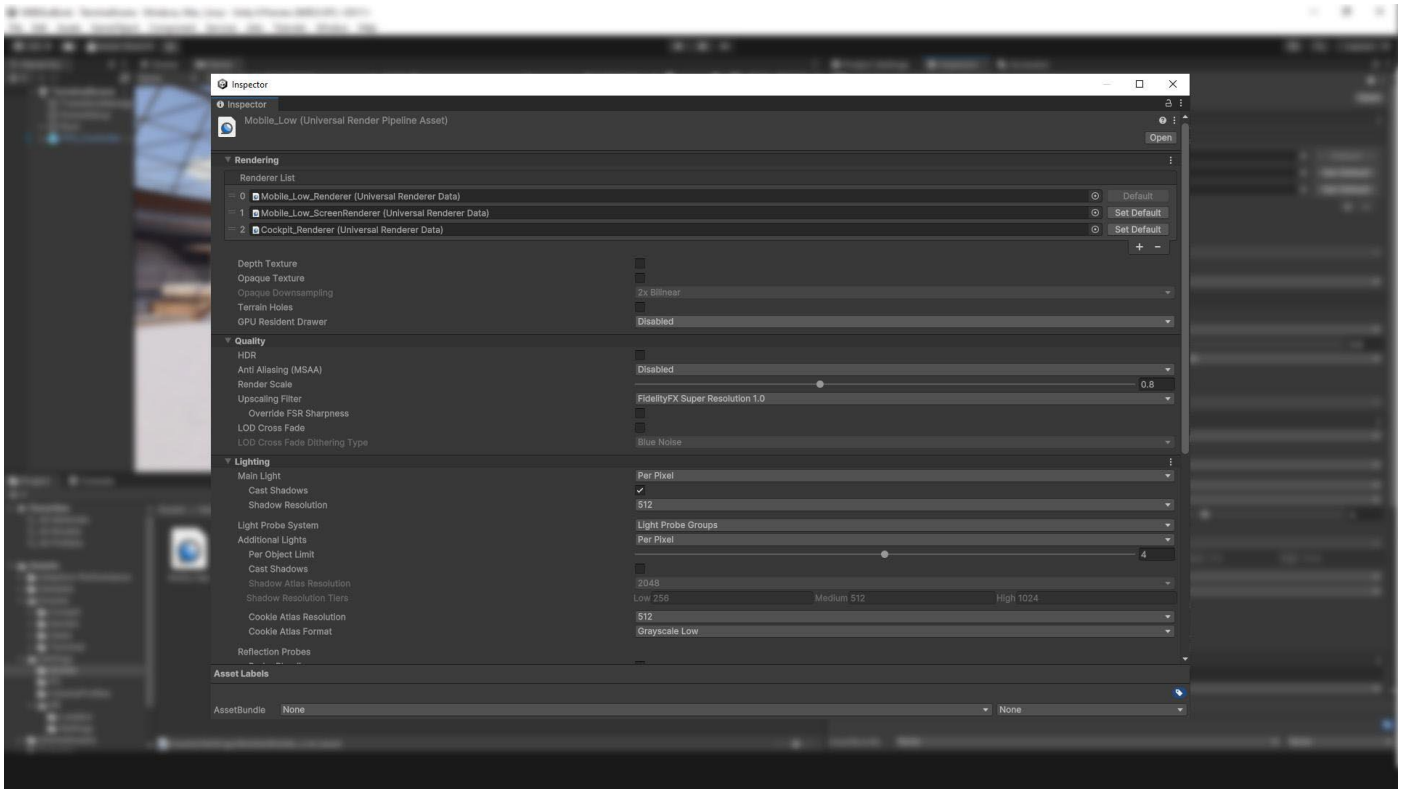


Unity モバイル、XR、またはウェブのゲームを開発する場合は、URP をレンダーパイプラインに選ぶ。

URP の選択に加えて、レンダ―パイプラインアセットを調節してさらに設定をカスタマイズできます。

レンダリング最適化

モバイルデバイスにおけるテザーレスな VR 体験や AR アプリケーションのために、URP は品質およびパフォーマンスに合わせたプリセットを提供します。適切なレンダリングの設定を選択することによってモバイルハードウェアのアプリケーションが最適になり、効率的なレンダリングとスムーズなパフォーマンスが得られます。最適化された URP の設定は、テクスチャ品質、影の解像度、ライトを効率的に管理し、モバイルやテザーレス XR デバイスのコンストレイントに適したビジュアルの忠実度とパフォーマンスのバランスをとります。



レンダーパイプラインアセット

プロファイリングのヒント

早期段階から頻繁に対象となるデバイスで プロファイリングを行う

プロファイリングは、ランタイム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。

モバイル、XR、ウェブプロジェクトについては、ローンチの直前だけではなく、開発サイクルの初期から全体を通じてアプリケーションのプロファイリングを行うことが大切です。不具合や急激な悪化などのパフォーマンスの問題が現れたらすぐに対応し、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。プロジェクトの明確な「パフォーマンスプロファイリング」を開発することによって、新たな問題をさらに簡単に特定し解決することができ、あらゆるターゲットプラットフォームにおいて最適なパフォーマンスが得られます。

エディターでのプロファイリングでは、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、各デバイスでのプロファイリングでは、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のプロファイルを作成するようにしましょう。サポートする予定の最高スペックのデバイスと最低スペックのデバイスの両方でプロファイルを作成し、忘れずに最適化を行ってください。

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。以下の通り、各ハードウェアのさらなるパフォーマンステストのための iOS や Android のネイティブツールもあります。

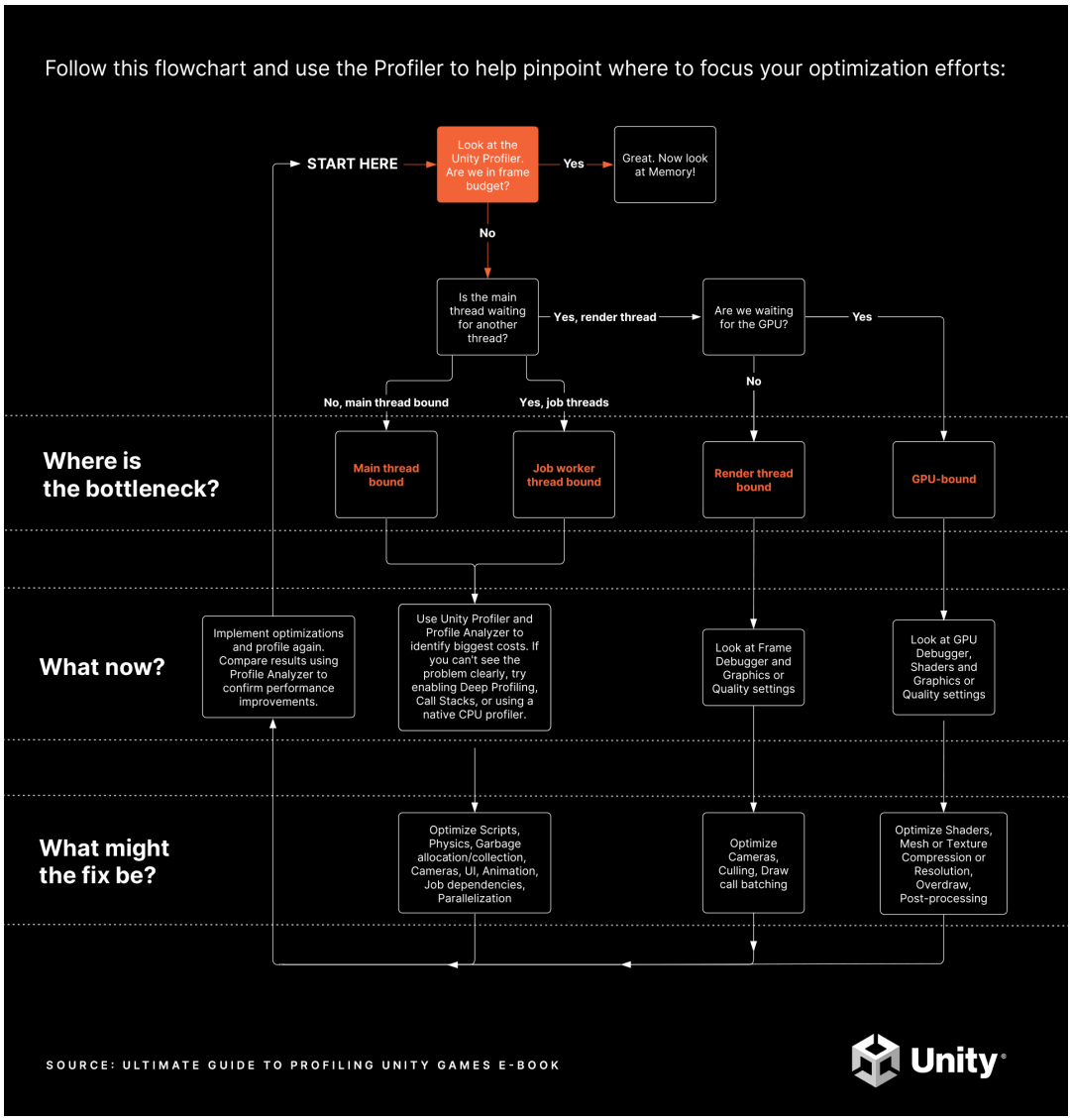
- iOS では、Xcode および Instruments を使用
- Android / Arm では、以下を使用：
    - Android Studio：最新の Android Studio には以前の Android Monitor ツールに代わる新しい Android Profiler が含まれています。Android デバイスのハードウェアリソースに関するリアルタイムデータを収集するために使用します。
    - Arm Mobile Studio：Arm ハードウェアを実行するデバイス向けに、ゲームの詳細なプロファイル作成とデバッグに役立つツール スイートです。
    - Snapdragon Profiler：Snapdragon チップセットデバイス専用です。CPU、GPU、DSP、メモリ、電力、熱、ネットワーク データを分析して、パフォーマンスのボトルネックを見つけて修正します。
    - Meta Quest のための開発者ツール：Meta の開発者ツールのウェブサイトにある Meta Quest ヘッドセット用のアプリケーション開発に関する情報を参照してください。

特定のハードウェアは Intel VTune も利用することができ、Intel プラットフォームにおけるパフォーマンスのボトルネックを見つけて修正するのに役立ちます。（Intel プロセッサーのみ）。

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ます。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、ご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。



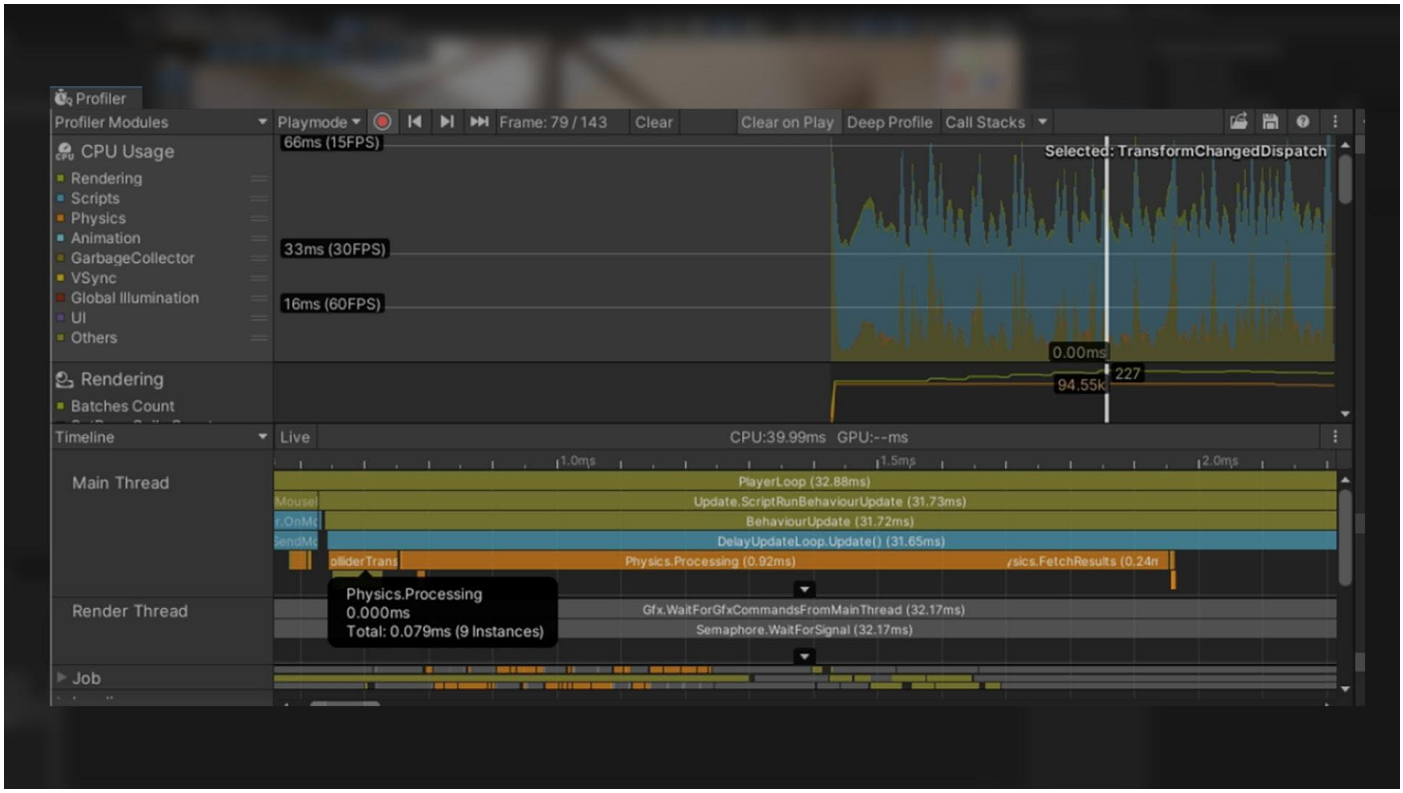
プロファイリング eBook からのチャートで、Unity プロジェクトを効率的にプロファイリングするためのワークフローを示している

Unity Profiler の仕組みを理解する

Unity Profiler は、ランタイムのラグ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。

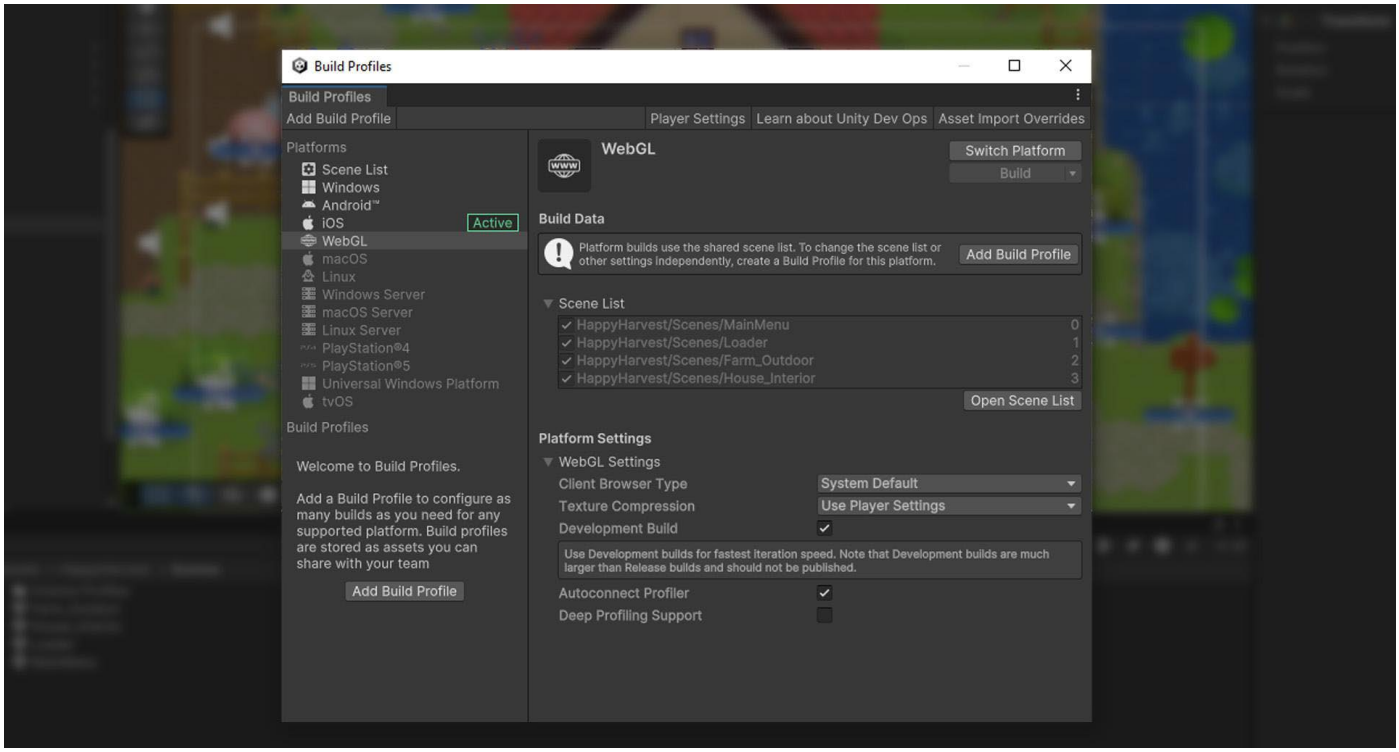
このプロファイラーはインストルメンテーションをベースとしています。ゲームおよびエンジンの自動でマークアップされたコード（MonoBehaviour の Start または Update メソッド、特定の API 呼び出しなど）や、ProfilerMarker の API を使用して明示的にラップされたコードのタイミングをプロファイルします。

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）。



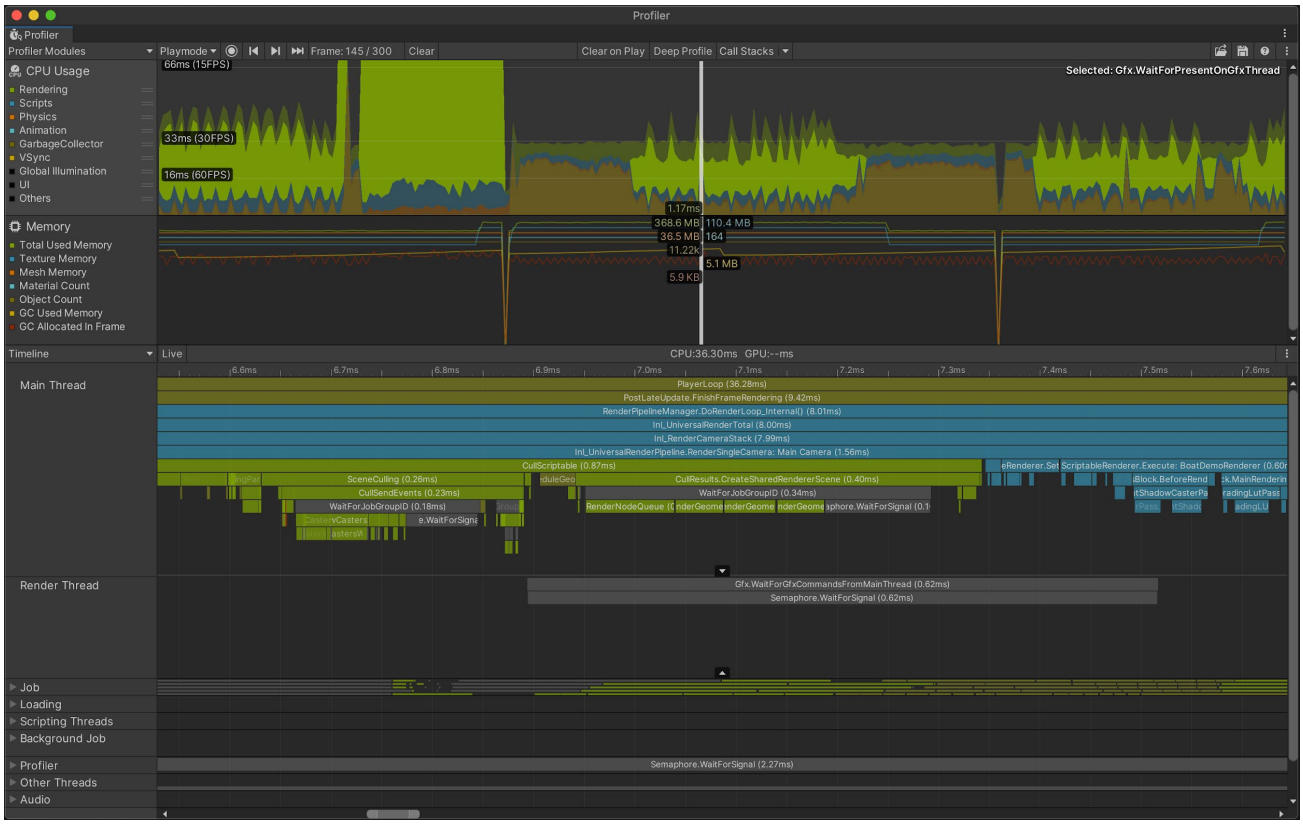
Unity Profiler を使ってアプリケーションのパフォーマンスとリソースの割り当てをテストする。

選択したプラットフォーム内の実際のモバイル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** および **Autoconnect Profiler** のボックスをチェックしてください。または、プロファイリングとは別にアプリケーションを開始したい場合は、**Autoconnect Profiler** のボックスのチェックを外し、アプリケーションが実行されたら手動で接続することができます。



ビルド設定は、プロファイリングを行う前に調整する。

プロファイルするプラットフォームのターゲットを選択します。**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。これは Unity エディターがさらに CPU に負荷をかけてメモリを消費する必要があることを意味しますが、特定のシナリオによっては有用です。



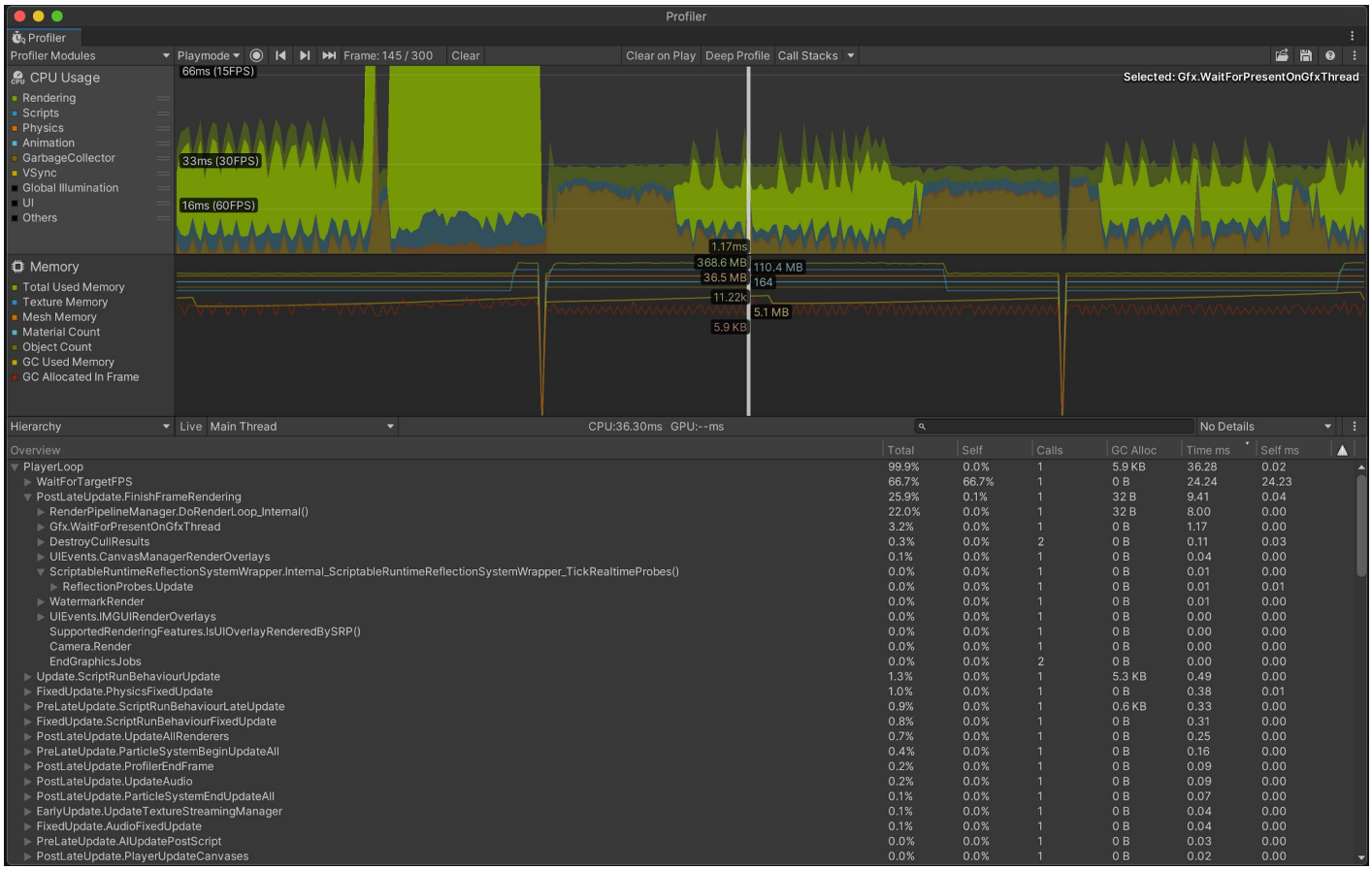
Timeline ビューを使用して、CPU 依存か GPU 依存かを確認する。

アプリケーションについて詳細な情報を捉えた詳しい分析が必要な場合は、Deep Profiling 設定も使用できます。これによって 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、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、それにつれてプロファイリングセッション中にゲームの実行速度が遅くなります。

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。
    - CPU のフレーム時間が GPU のフレーム時間よりも著しく多い場合、ゲームは CPU 依存です。これは CPU のゲームロジック、物理演算、もしくは他の計算の処理が長引いており、CPU がタスクを終えるのを GPU が待っているということを意味しています。
    - 同様に、GPU のフレーム時間が CPU のフレーム時間よりも多い場合、ゲームは GPU 依存です。これは GPU のグラフィックスのレンダリングが長引いており、GPU がレンダリングを終えるのを CPU が待っているということを示しています。

— Timeline 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、関数へのコール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。Time ms および Self ms をソートすることによって、機能単独で、または呼び出した機能に起因して最も時間がかかっている機能を特定できます。これは、最もパフォーマンスが向上する箇所の最適化に力を集中させることに役立ちます。



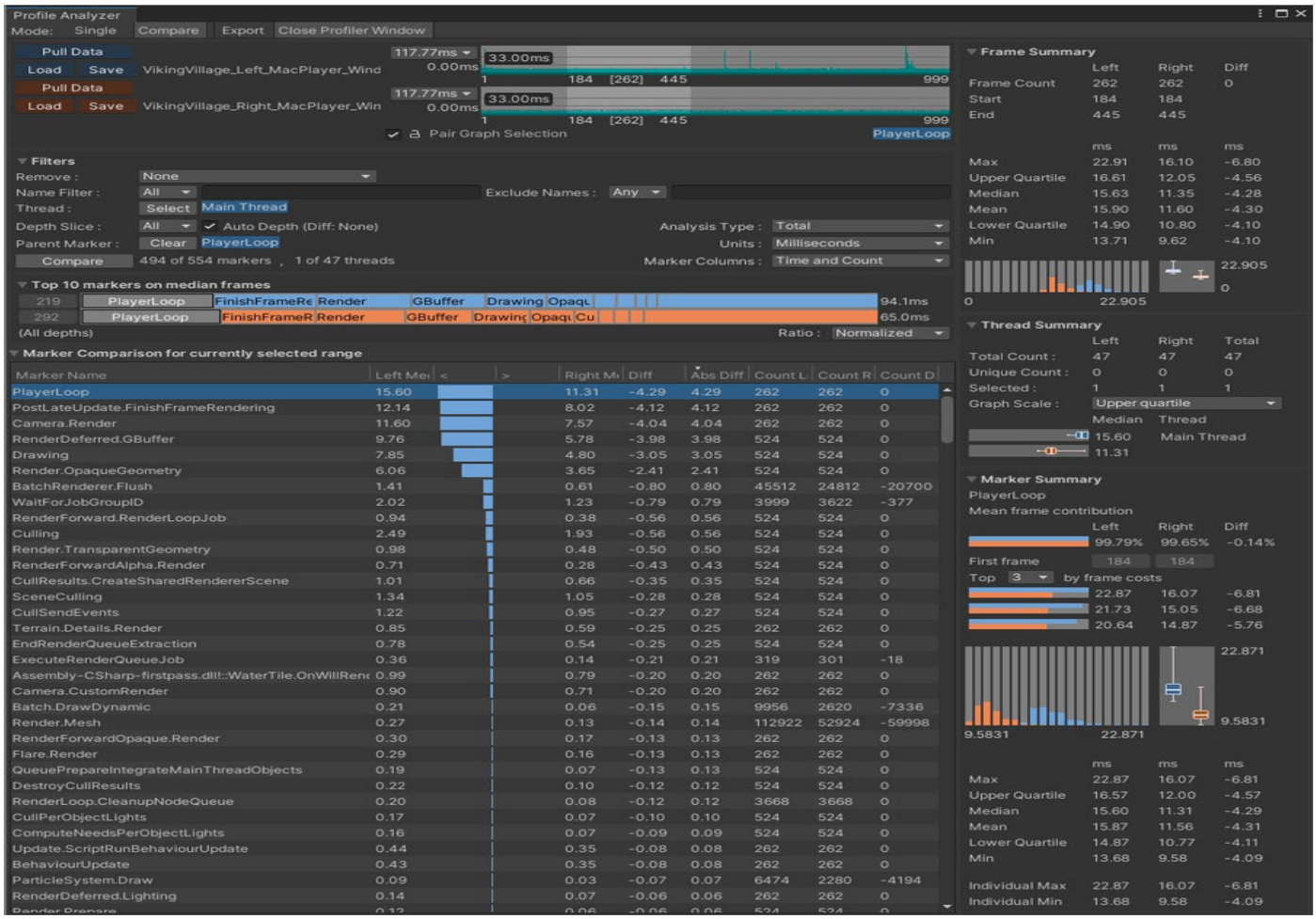
Hierarchy ビューでは、時間コストごとに ProfileMarkers を並べ替えることができる。

Unity Profiler の完全な概要については、こちらで確認できます。プロファイリングにまだ慣れていない場合は、「Unity プロファイリング入門」もご覧ください。

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。そして、繰り返し行ってください。

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 がどうなるかを見たいとします。**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、変更をテストしてその結果を改善することができます。Profile Analyzer は Unity の Package Manager からアクセス可能です。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。

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 30 FPS 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 33.33 ミリ秒（1000 ミリ秒 /30 FPS）を超えることはできません。同様に、60 FPS の場合は 1 フレームあたり 16.66 ミリ秒となります（1000 ミリ秒 / 60 FPS）。

仮想現実（VR）アプリケーションを開発する場合、高く安定したフレームレートを維持することは、スムーズな没入型体験を確保しモーションシックネスを防ぐためにはなおさら重要となります。VR アプリケーションの

最も一般的な目標は 90 FPS で、1 フレーム当たりたった 11.11 ミリ秒 という厳しいフレームバジェットとなります（1000 ミリ秒 / 90 FPS）。VR は各フレームを 2 回ずつ（一つの目に対して 1 回）レンダリングする必要があり、わずかなタイミングのずれでもユーザーにはすぐに気づかれてしまう可能性があるので、このような高い要件が必要です。

一貫性のある高いフレームレートは Unity Web ビルドにとっても不可欠で、そのパフォーマンスはブラウザーの効率とハードウェアの性能によるところが大きいです。フレームあたりの厳しいタイムバジェットは重要な要素であり続けます。例えば、Unity WebGL ビルドで目標が 60 FPS の場合でも、フレーム当たりたった 16.66 ミリ秒しかありません。このバジェットは、レンダリングのすべての側面、物理演算、ゲームロジックを含んでおり、このことはアプリケーションの各部を最適化することの重要性を意味しています。アセットの効率的な管理、シーンの複雑さの抑制、シェーダーやスクリプトの最適化。これらはすべてアプリケーションがパフォーマンスの目標を達成するために必要なステップです。

Unity がブラウザーでコードをコンパイルして実行するために使う WebAssembly（Wasm）のパフォーマンスの影響を考慮することも重要です。Wasm が従来の JavaScript と比べてパフォーマンスを大幅に改善できる一方で、利用可能なフレーム時間を最大限に活用するためにプロファイリングを行いコードを最適化することはいまなお重要です。

デバイスの温度を考慮に入れる

ただしモバイルについては、一貫して最大時間を使うのは一般的に推奨できません。デバイスがオーバーヒートして OS が CPU や GPU をサーマルスロットリングさせる可能性があるからです。一般的な経験則としては、利用可能な時間の約 65% だけを使って、フレーム間でクールダウンさせることです。通常のフレームバジェットは 30 FPS では 1 フレーム当たり約 22 ミリ秒、60 FPS では 1 フレーム当たり 11 ミリ秒です。

デバイスは短時間（カットシーンやローディングシーケンスなど）であればこのバジェットを超えることができますが、長時間に対してはできません。

大半のモバイルデバイスには、デスクトップデバイスのようなアクティブクーリングがありません。物理的な熱の高さはパフォーマンスに直接影響を与える可能性があります。

デバイスが発熱している場合、それが長期にわたる懸念の原因にならなくとも、プロファイラーが検知してパフォーマンスの低下をレポートするかもしれません。プロファイリングによるオーバーヒートに対処するため、短時間のプロファイリングを繰り返すようにします。これにより、デバイスが冷却されてリアルワールドのコンディションがシミュレーションされます。全般的には、デバイスの冷却状態を 10 〜 15 分間維持してから次のプロファイリングを行うことを推奨します。

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。レンダリングのパフォーマンスに関わる問題の原因が、CPU がフレームのレンダリングに時間がかかりすぎることである場合、そのゲームは CPU 依存となります。レンダリングのパフォーマンスに関わる問題の原因が、GPU がフレームのレンダリングに時間がかかりすぎることである場合、それは GPU 依存となります。

プロファイラーを使えば、CPU が割り当てられたフレームバジェット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 を先頭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.WaitForPresent** 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、GPU がフレームを提示するのを待機している可能性があることを意味します。

最低スペックのデバイスおよび最高スペックのデバイスの両方でテストする

世界には iOS や Android のデバイスが幅広くあります。アプリケーションのサポートを望むデバイスの最低および最高スペックで、プロジェクトのテストを可能な限り行うことの重要性を繰り返しお伝えしたいと思います。

XR、ウェブ、モバイルゲームのメモリ管理

効果的なメモリ管理はスムーズなパフォーマンスを確保するためには重要です。Unity は、スクリプトやユーザーが作成したコードの自動メモリ管理を扱い、小さい一時データをスタックに割り当て、より大きい長期データをマネージヒープまたはネイティブヒープに割り当てます。しかしながら、XR、ウェブ、モバイルのアプリケーションが求めるのは、メモリ使用量に対するさらに慎重なアプローチです。非効率的なメモリ管理は、遅いフレームレート、ロード時間の増加、さらにはアプリケーションのクラッシュなどのパフォーマンスの問題につながる可能性があるからです。このセクションでは、これらのプラットフォームでメモリ使用量を最適化する戦略を見ていき、応答性が高く安定したアプリケーションを提供するお手伝いをします。

効率的なメモリ管理

すべてのプラットフォームにおいてスムーズで応答性の高い体験を提供するには、オブジェクトのライフサイクルの慎重な管理、ガベージコレクションのオーバーヘッドの最小化、アセットローディング戦略の最適化が不可欠です。

オブジェクトのライフサイクルを管理：オブジェクトの作成と破棄を適切に管理して、メモリリークやリソースの不必要な使用を防ぎます。**Destroy()** を使って未使用のオブジェクトを取り除き、もう必要ない場合はそれらを無効にするためにリファレンスを設定します。これによってメモリを解放します。

オブジェクトプール：銃弾、敵、UI 要素といった頻繁に使うオブジェクトは作成や破棄を繰り返すのではなく再利用しましょう。オブジェクトプールを実装することによって、オブジェクトのインスタンス化や破棄に伴うオーバーヘッドを大幅に減らし、メモリリソースを節約できます。詳細については、「Unity プロジェクトのオブジェクトプール」をお読みください。

ガベージコレクションの影響を減らす：割り当てを最小化して、パフォーマンスの低下を引き起こす可能性のあるガベージコレクションの頻度と影響を減らします。可能な限り配列やリストを事前に割り当てることによって、更新ループにおける頻繁な割り当てを回避します。適切であれば参照型（クラス）の代わりに値型（構造体）を使います。構造体はスタックに割り当てられガベージコレクションのオーバーヘッドの一因とはならないからです。

以下のテクニックでアセットローディングを最適化します。

— **遅延読み込み**：実際に必要になるまでリソースの読み込みを遅らせます。これによって初期のロード時間を短縮し、より効率的にリソース管理をすることが容易になります。

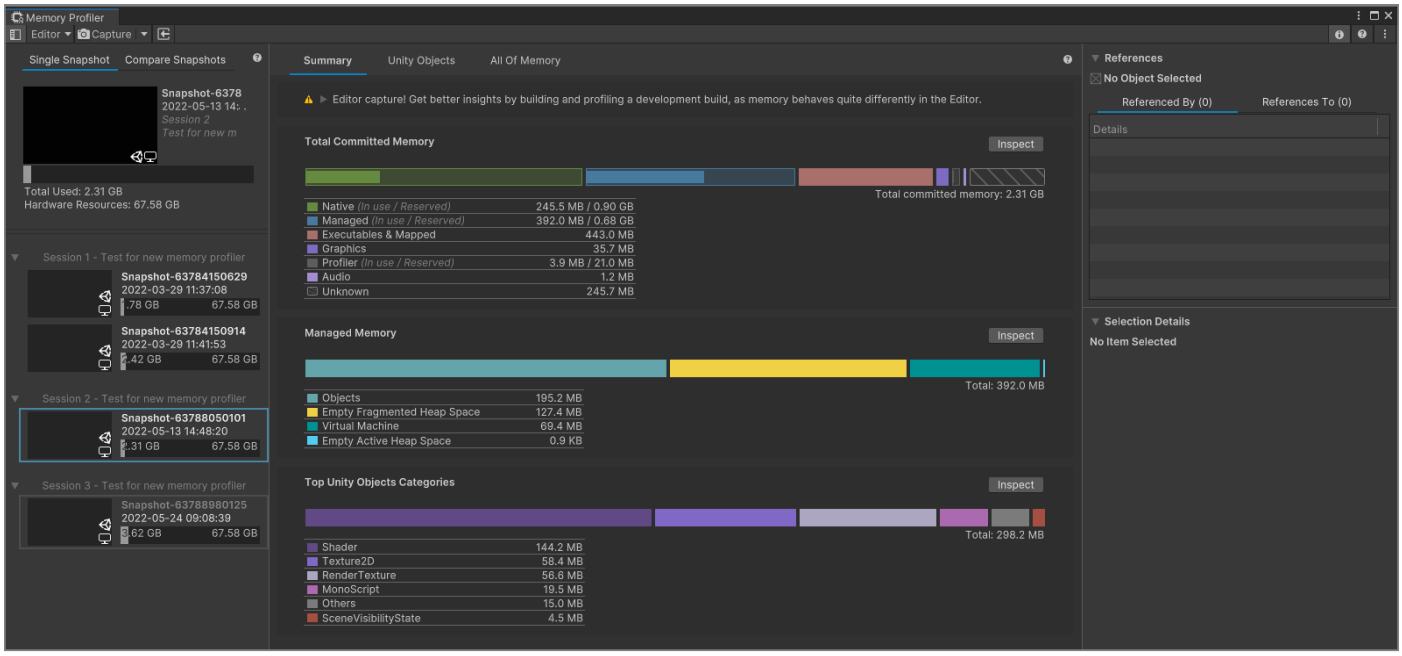
— **未使用のアセットをアンロードする**：Resources.UnloadUnusedAssets() を使って、もう必要のないアセットの占めるメモリを解放します。

— **Addressable Asset System を使用**：Addressable Asset System を活用してランタイム時に非同期でアセットを管理します。このシステムはウェブやモバイルのプラットフォームにおいては特に有益で、リモートアセットのホスト、動的コンテンツの更新、遅延読み込みをサポートしています。

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、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、Unity のこちらの動画をご覧ください。

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。大規模に使用する場合は、不要な文字列の作成や操作は減らしましょう。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しょう。ランタイムに文字列を構築する必要がある場合は、StringBuilder クラスを使用しましょう。
- **Unity 関数呼び出し**：いくつかの関数はヒープ割り当てを作成します。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ましょう。例えば、手動で文字列と `GameObject.tag` を比較するのではなく、`GameObject.CompareTag` を使用しましょう（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ましょう。これは一時的なオブジェクトを作成し、それに付随する潜在的なガベージは暗示的に値型を型オブジェクトに変換するためです（例：int i = 123、object o = i）。代わりに、渡したい値型に具体的なオーバーライドを提供するようにしましょう。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、リストを使いましょう。

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。

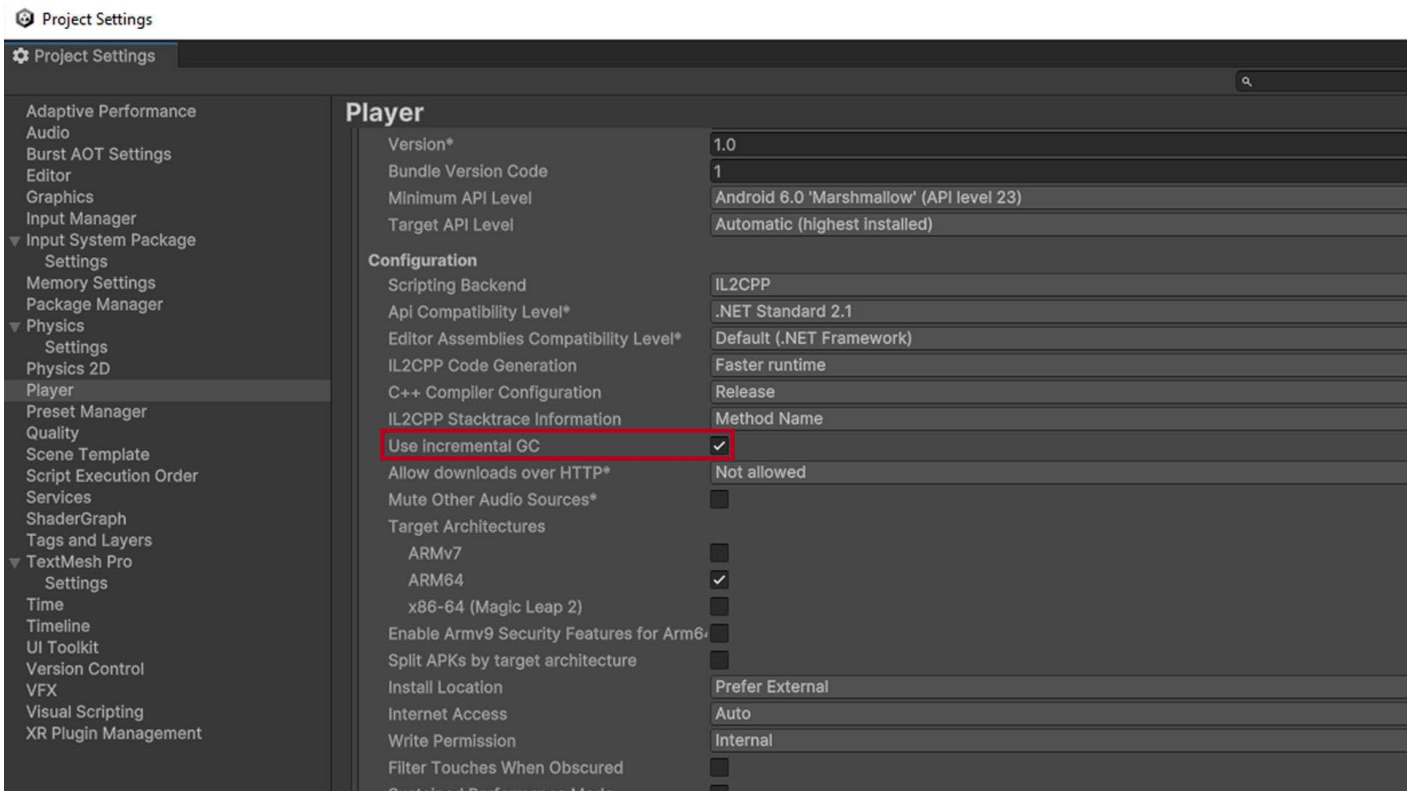
ガベージコレクションを使用できるタイミングを特定する

ガベージコレクション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。

インクリメンタルガベージコレクターを使って GC のワークロードを分割する

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、複数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よう。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。



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。

Adaptive Performance

Unity と Samsung の Adaptive Performance を使うことによってデバイスの熱や電源の状態を監視でき、適切に反応する準備を整えられます。ユーザーが長時間プレーする場合、level of detail（LOD）バイアスを動的に減らして、ゲームをスムーズに動作させ続けることができます。Adaptive Performance によって、開発者は管理する方法でパフォーマンスを向上させつつ、グラフィックスの忠実度を維持できます。

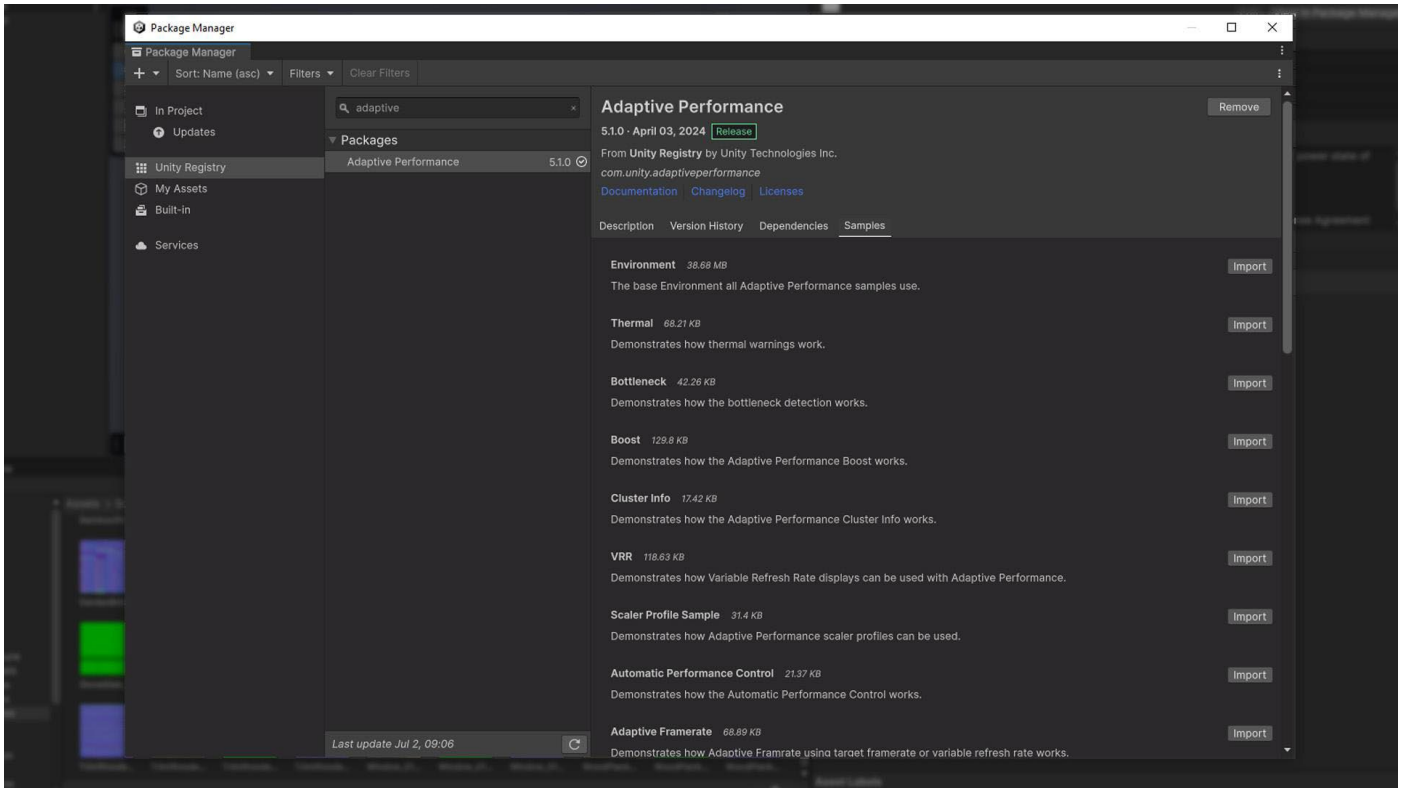
Adaptive Performance API を使ってアプリケーションを微調整できる一方、Adaptive Performance はいくつかの自動モードも提供します。これらのモードでは、Adaptive Performance はいくつかの重要な指標に沿ってゲームの設定を決定します。

- 以前のフレームを基にした必要なフレームレート
- デバイスの温度レベル
- 熱事象に対するデバイスの近接性
- CPU 依存または GPU 依存のデバイス

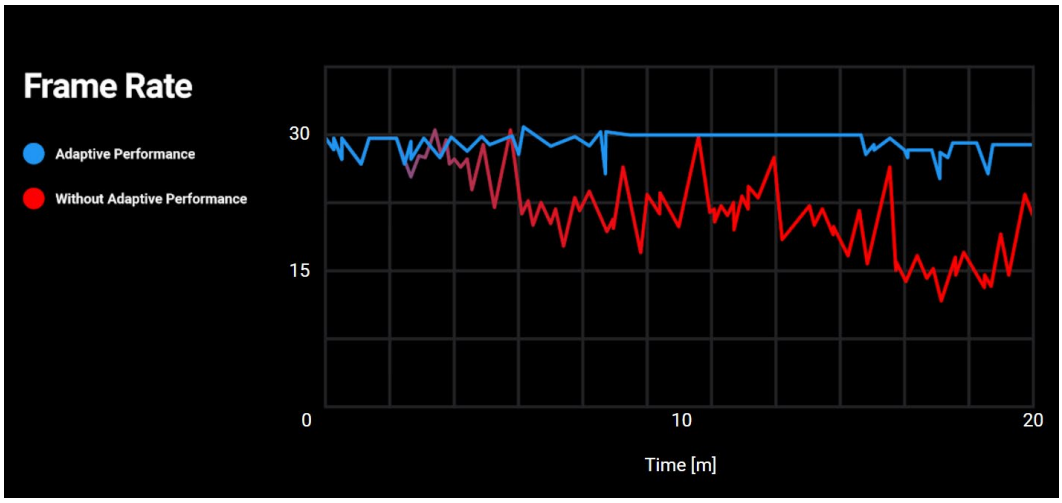
この 4 つの指標はデバイスの状態を決定し、Adaptive Performance はボトルネックを緩和するために適応された設定を調整します。これはインデクサーとして知られる整数値を供給することによって実現でき、デバイスの状態を表します。

Adaptive Performance についての詳細は、**「Package Manager」＞「Adaptive Performance」＞「Samples」**を選択し、**「Package Manager」**内に供給したサンプルをご覧ください。各サンプルは特定のスケーラーと相互作用するので、異なるスケーラーがどのようにゲームに影響を与えるかが分かります。また、エンドユーザードキュメントを見直して Adaptive Performance の設定の詳細、および API と直接相互作用する方法を学ぶことを推奨します。

Adaptive Performance は Samsung のデバイスのみ機能することに注意してください。



Adaptive Performance パッケージ



アセット

よく最適化されたアセットパイプラインはロード時間を短縮でき、メモリ使用量を削減し、ランタイムパフォーマンスを改善できます。経験豊富なテクニカルアーティストとともに働くことによって、チームはアセットフォーマット、仕様、インポート設定を定義して実施でき、効率的で合理的なワークフローを確保します。

デフォルトの設定のみ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活用してテクスチャ、メッシュジオメトリ、オーディオファイル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、メモリ使用量が低下する可能性があります。

Presets を使用して、特定のプロジェクトのニーズに合わせたベースライン設定の構築を検討してください。この積極的なアプローチによって、最初からアセットが最適化されるようになり、あらゆるプラットフォームでよりよいパフォーマンスやより一貫した経験を得ることにつながります。

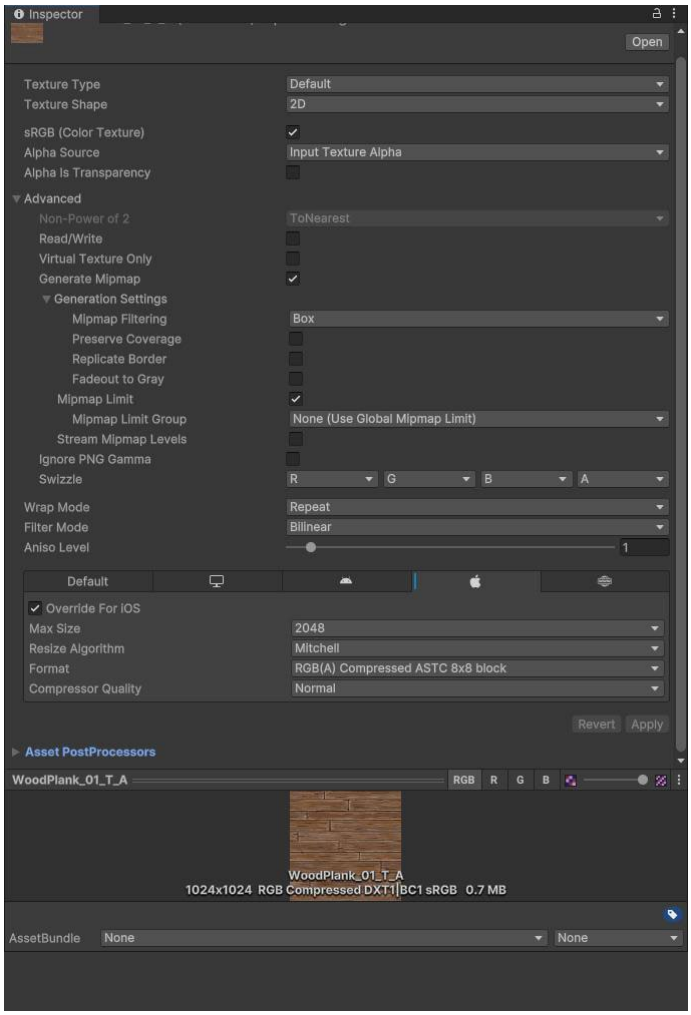
さらなるガイダンスは、Unity Learn の「アートアセットのベストプラクティス」または「モバイルアプリケーションのための 3D アート最適化」をご覧ください。これらのリソースは貴重な知見を提供し、Unity ウェブビルド、モバイル、XR のアプリケーションのためのアセット最適化について、情報に基づいた決定を下すことに役立ちます。

適切にテクスチャをインポートする

しばしばテクスチャが最もメモリを消費するので、インポート設定は重要です。テクスチャを最適化するために、以下のガイドラインをご検討ください。

— **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すぐに減らすことができます。

— **2 のべき乗を使用する**：Unity では、モバイルテクスチャ圧縮形式（PVRCT または ETC）に POT テクスチャ寸法が必要です。



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。単一のテクスチャに複数のテクスチャを配置することによって、ドローコールを減らしレンダリングをスピードアップできます。Unity Sprite Atlas またはサードパーティ製の TexturePacker を使用して、テクスチャをアトラス化してください。

— **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する場合は、**Texture2D.Apply** で、**true** に設定された **makeNoLongerReadable** を渡すことで有効にしてください。

— **不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できますが、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）。

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。左側の設定は、右側のものと比べてメモリを約 26 倍消費し、ビジュアルの品質はさほど向上しません。

非圧縮テクスチャはより多くのメモリを必要とする。

モバイル、XR、ウェブのために Adaptive Scalable Texture Compression（ATSC）を使用します。開発中のほとんどのゲームが、ATSC をサポートする最低スペックのデバイスを目標とする傾向があります。

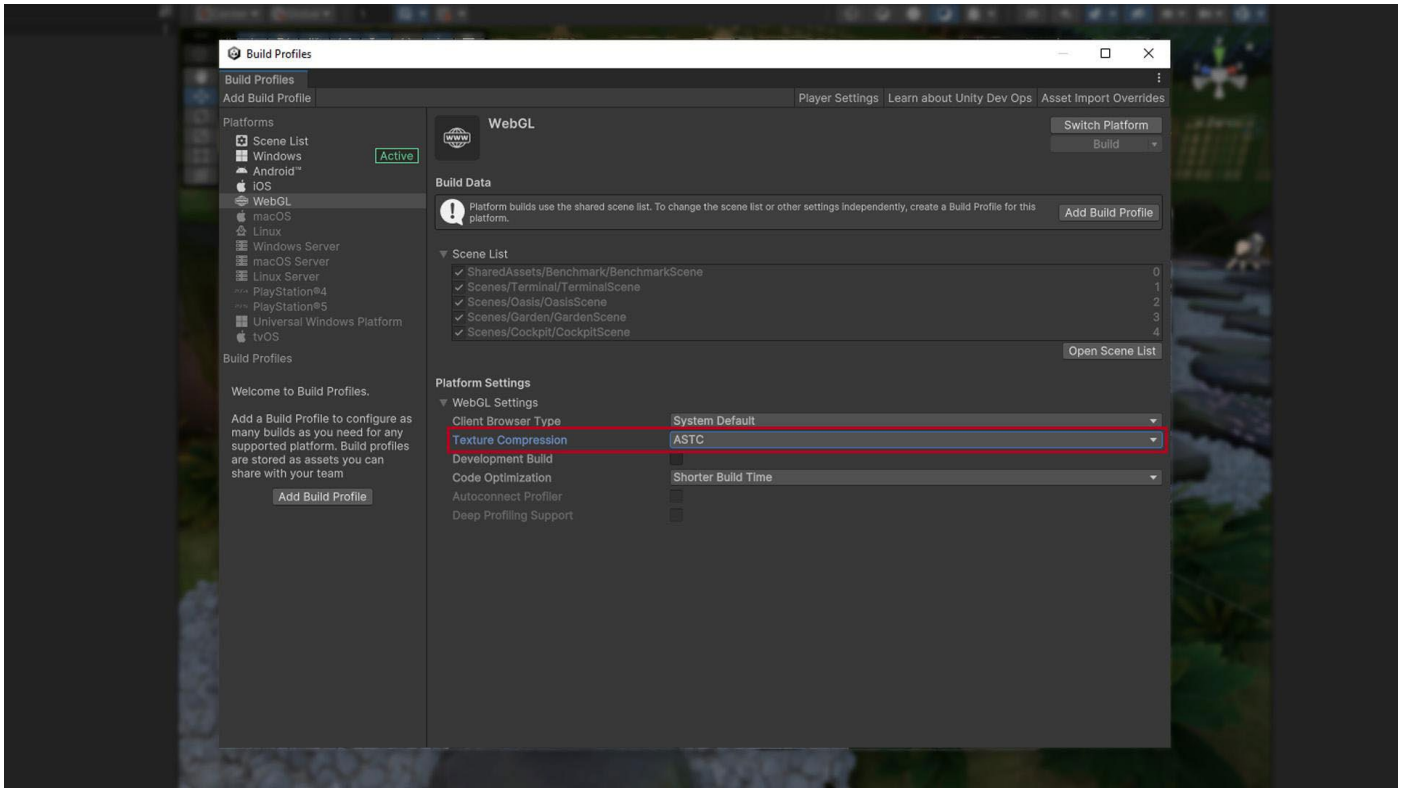
数少ない例外は以下の通りです。

— A7 以下のデバイスをターゲットとする iOS ゲーム（iPhone 5、5S など）。PVRTC を使用。

— 2016 年以前のデバイスをターゲットとする Android ゲーム。ETC2（Ericsson Texture Compression）を使用。

PVRTC や ETC などの圧縮形式が十分に高品質ではない場合、そして ASTC が目標のプラットフォームで十分にサポートされていない場合は、32 ビットのテクスチャの代わりに 16 ビットのテクスチャを使ってみてください。

詳細については、マニュアル「プラットフォーム別に推奨されるテクスチャ圧縮形式」をご覧ください。

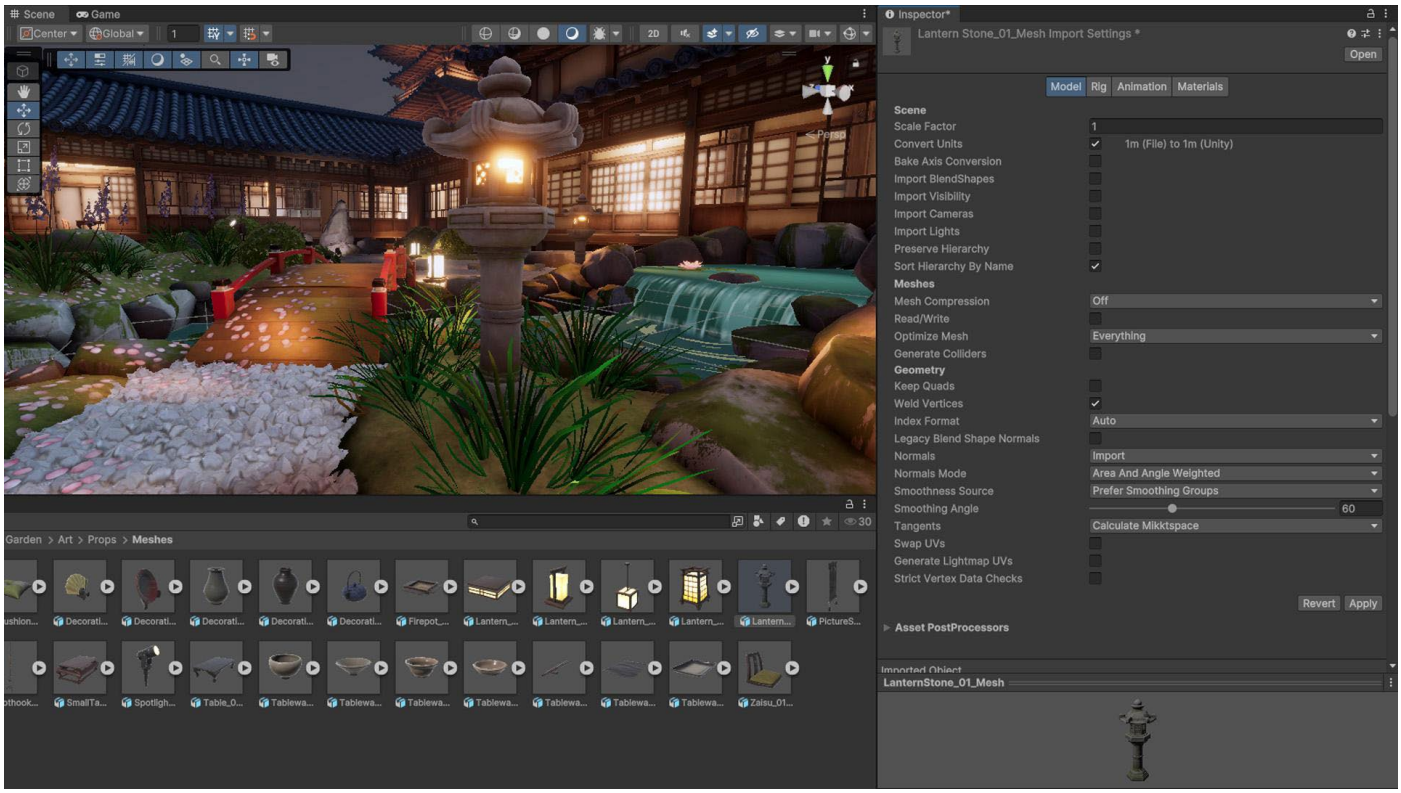


ビルド設定のもとでテクスチャ圧縮のための ASTC を選択する

メッシュのインポート設定を調整する

テクスチャと同様に、メッシュも大幅にメモリを消費する可能性があるので、それに対して最適なインポート設定を選んでください。以下のプラクティスによってメッシュのメモリフットプリントを削減します。

- **メッシュを圧縮する**：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、実行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、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。

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、50 万 もポリゴン数が必要ですか?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、高密度のメッシュの代わりに細かいディテールのためのテクスチャや法線マップを使います。

AssetPostprocessor を使用して インポート設定を自動化する

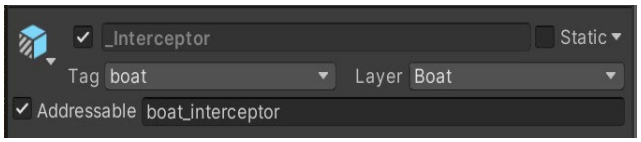
AssetPostprocessor を使用すると、インポートパイプラインにフックして、アセットのインポート前またはインポート時にスクリプトを実行できます。これは、モデル、テクスチャ、オーディオなどをインポートする前またはインポートした後に、設定をカスタマイズするよう促します。プリセットに似た方法ですが、コードを介して行います。このプロセスの詳細については、GDC 2023 トーク「ゲーム制作の各段階におけるテクニカルなヒント」を確認してください。

Unity DataTools

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。

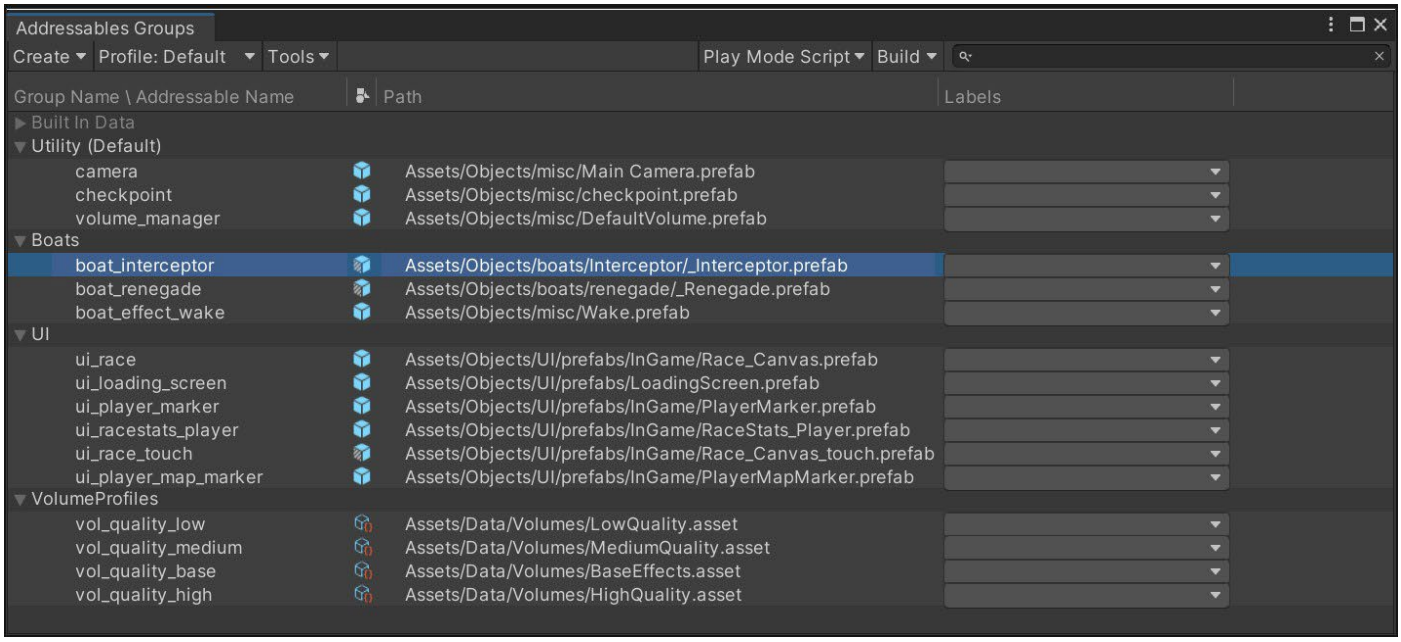
Addressable Asset System を使用する

Addressable Asset System は、コンテンツを管理するためのシンプルな方法を提供します。この統一されたシステムでは、アセットバンドルを「アドレス」またはエイリアスごとに、ローカルのパスまたはリモートのコンテンツデリバリーネットワーク（CDN）から非同期的に読み込みます。



コード以外のアセット（モデル、テクスチャ、プレハブ、オーディオ、さらにはシーン全体）をアセットバンドルに分割すれば、ダウンロードコンテンツ（DLC）として分離することができます。

その後、Addressables を使用して、モバイルアプリケーションの小規模な初期ビルドを作成します。Cloud Content Delivery を使えば、ゲームのコンテンツをホストし、ゲームの進行に合わせてプレイヤーに配信することができます。



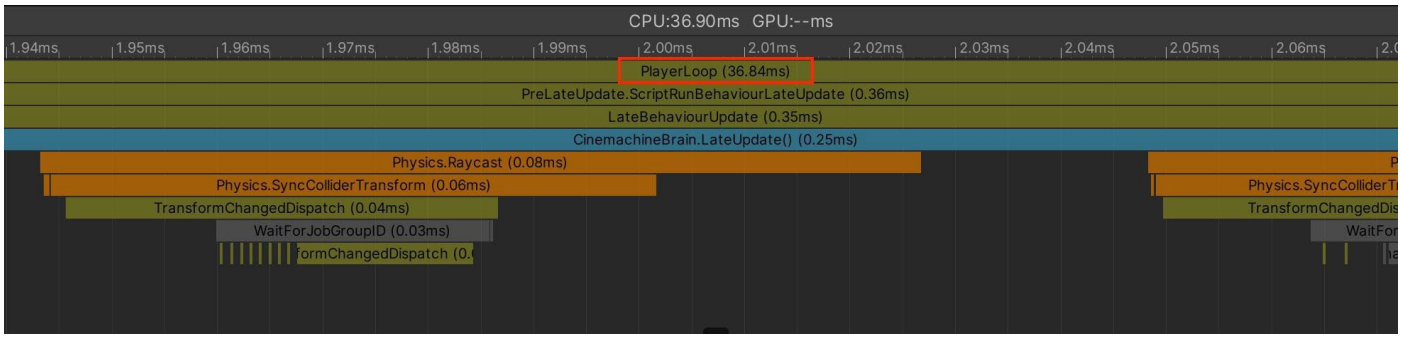
Addressable Asset System を使って、「アドレス」ごとにアセットを読み込む。

こちらをクリックして、Addressable Asset System を使って煩わしいアセット管理を単純化する方法をご覧ください。

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。この構造には、初期化とフレームごとの更新を処理するいくつかのシステムが含まれています。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）。

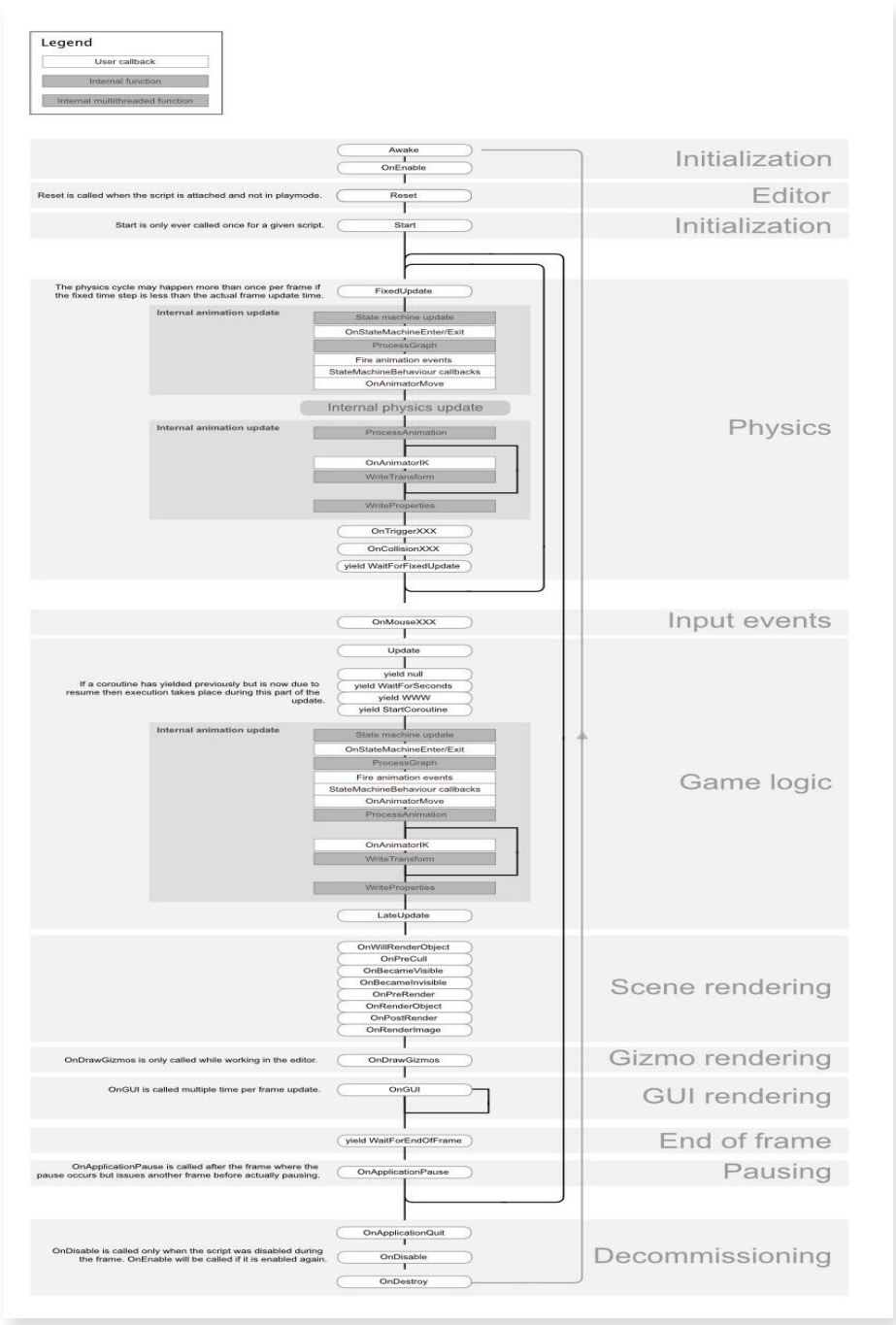


Profiler は、エンジン全体の実行コンテキスト内で、カスタムスクリプト、設定、グラフィックスを表示する。

以下のヒントやテクニックを使ってスクリプトを最適化してください。

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について学ぶ。

毎フレーム実行されるコードの最小化

コードが毎フレーム実行されるべきか考えます。そして、**Update**、**LateUpdate**、**FixedUpdate** から不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ましょう。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 1 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。

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。

詳細については、「イベント関数の実行順序」を参照してください。

空の Unity イベントを避ける

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
```

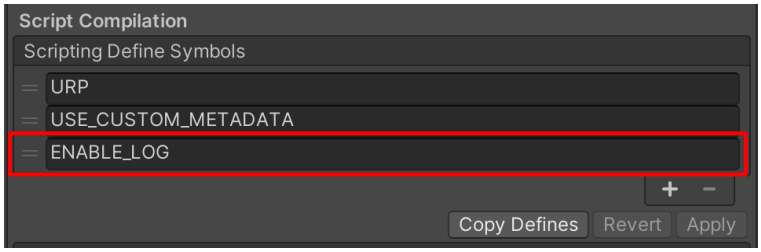
ここでは、不必要なオーバーヘッドがビルドに入り込むことなく、**Update** in Editor を自由にテストに使うことができます。

デバッグログステートメントをなくす

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** で **ENABLE_LOG** プリプロセッサを無効にすると、**Log** ステートメントを一挙に消去できます。

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。

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。

適切なデータ構造を選択する

データ構造の選択は、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？適切な構造を選択する際の参考として、C# のデータ構造に関する MSDN ガイド に従いましょう。

コンポーネントを実行時に追加することは避ける

ランタイム中に **AddComponent** を呼び出すには、それなりの負荷がかかります。Unity は、ランタイム中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セットアップ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。

ゲームオブジェクトとコンポーネントをキャッシュする

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。

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);
}
```

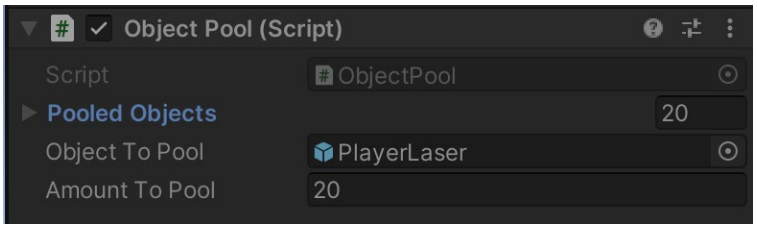
関数の結果がキャッシュされるので、**GetComponent** を一度だけ呼び出すようにした方が効率的です。キャッシュされた結果は、それ以上 **GetComponent** を呼び出すことなく、**Update** で再利用することができます。

```
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しょう。

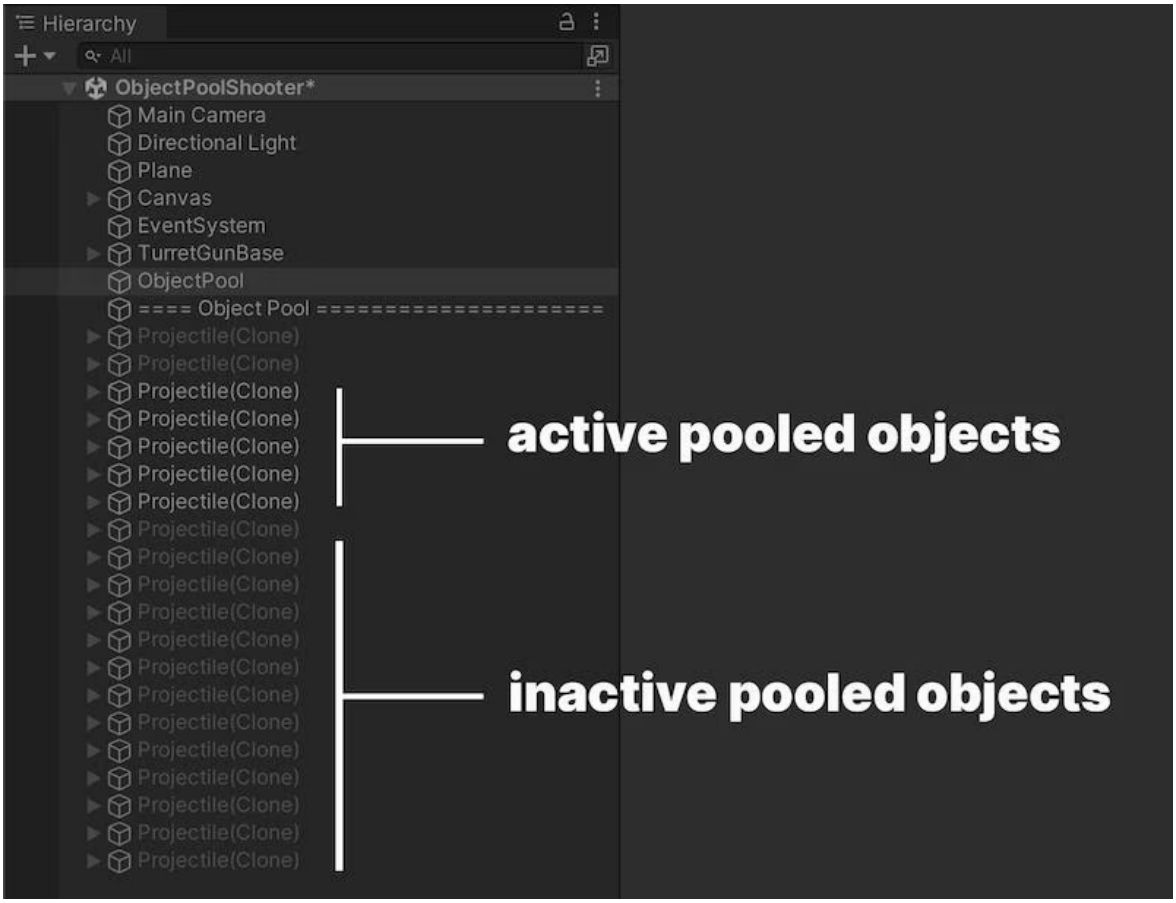
オブジェクトプールを使用する

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のベストプラクティスは、CPU スパイクが目立たない時（例えば、メニュー画面にいる間）に再利用可能なインスタンスを作成することです。その後、このオブジェクトの「プール」をコレクションで追跡します。ゲームプレイ中は、必要に応じて次の利用可能なインスタンスを有効にし、オブジェクトを破壊するのではなく無効にし、プールに戻します。



非アクティブで発射準備ができている発射体オブジェクトのプールの例。

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、オブジェクトプーリング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。

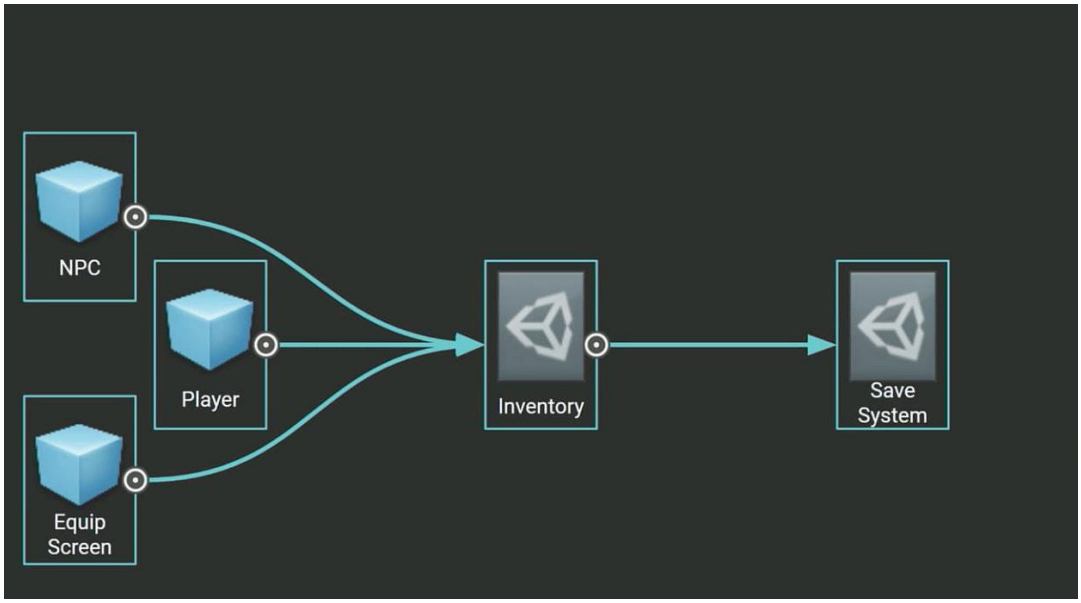
ScriptableObject を使用する

MonoBehaviour ではなく、**ScriptableObject** に静的な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、ランタイム中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、データそのものをコピーするのではなく、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*」をお読みください。

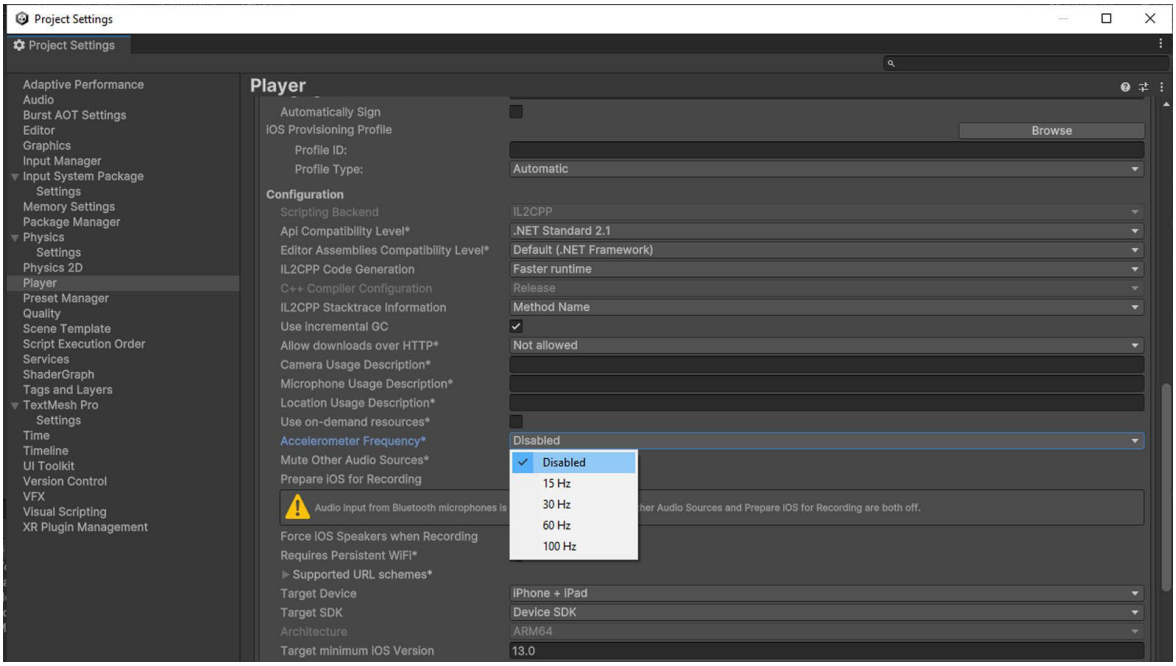
「Introduction to ScriptableObjects(日本語キャプション)」デブログを視聴して、ScriptableObjects があなたのプロジェクトにどのように役立つかを見つけましょう。こちらの Unity ドキュメントとテクニカルガイド「*Unity で ScriptableObject を使用してモジュラーゲームアーキテクチャを作成する*」も参照してください。

プロジェクト設定

いくつかの設定は、モバイルパフォーマンスに影響を与える可能性があります。

加速度センサーの頻度を減らすか、無効にする

Unity はモバイルの加速度センサーを毎秒数回プールします。アプリケーションでこれが使われていない場合は無効にするか、全体のより良いパフォーマンスのために頻度を減らします。



モバイルゲームで「Accelerometer Frequency」を利用していない場合は、確実に無効にする。

不要な「Player」設定や「Quality」設定を無効にする

Player 設定において、サポートされていないプラットフォームでは「**Auto Graphics API**」を無効にし、過剰なシェーダーバリアントの生成を防ぎます。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しょう。

不要な物理演算を無効にする

ゲームが物理演算を使用していない場合、「**Auto Simulation**」および「**Auto Sync Transforms**」のチェックを外します。これらはただアプリケーションをスピードダウンさせて、これといったメリットはありません。

適切なフレームレートを選択する

モバイルプロジェクトでは、フレームレートとバッテリー寿命やサーマルスロットリングのバランスを取る必要がある。60 FPS でデバイスの限界に挑戦するのではなく、妥協案として 30 FPS での実行を検討してください。Unity のモバイル向けデフォルトは 30 FPS になっています。

XR プラットフォームがターゲットである場合、フレームレートを検討することはなおさら重要です。没入感を維持しモーションシックネスを防ぐためには、多くの場合、72 FPS、90 FPS、さらには 120 FPS のフレームレートが必要です。より高いフレームレートはスムーズで応答性の高い体験の確保に役立ち、VR 環境を心地良くするためには重要です。しかしながら、これには電力消費や熱管理の点において課題があり、スタンドアロンの VR ヘッドセットについては特にそうです。

モバイルデバイス、スタンドアロンの VR ヘッドセット、AR デバイスのいずれにおいても、適したフレームレートを選ぶことは、すなわちターゲットプラットフォーム固有の要求や制約を理解するということです。適切なフレームレートを慎重に選択することによって、異なるプラットフォーム間でパフォーマンスおよびユーザー体験の両方を最適化できます。

Application.targetFrameRate でランタイム時にフレームレートを動的に調整することもできます。例えば、ゆっくりとしたシーンや比較的静かなシーンでは 30 FPS 未満に落とし、ゲームプレイではより高い FPS に設定するよう指定することができます。

大規模な階層を避ける

階層は分割しましょう。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。

トランスフォームを 2 回ではなく 1 回にする

加えて、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。

ランタイム中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。

XR、ウェブ、モバイルの開発における Vsync

XR、ウェブ、モバイルのプラットフォームを開発する場合、Vsync（Vertical Synchronization）は、Unity エディター（「Project Settings」＞「Quality」）で無効にしても有効になっていると考えてください。Vsync は多くの場合これらのプラットフォームのハードウェアレベルで実行され、スクリーンのティアリングを防ぎスムーズなビジュアル出力を確保します。GPU が十分な速さでフレームをレンダリングできずディスプレイのリフレッシュレートに合わせられない場合、その時点のフレームを保持して再表示することによって効果的に FPS を削減します。プラットフォームごとにその仕組みを見ていきましょう。

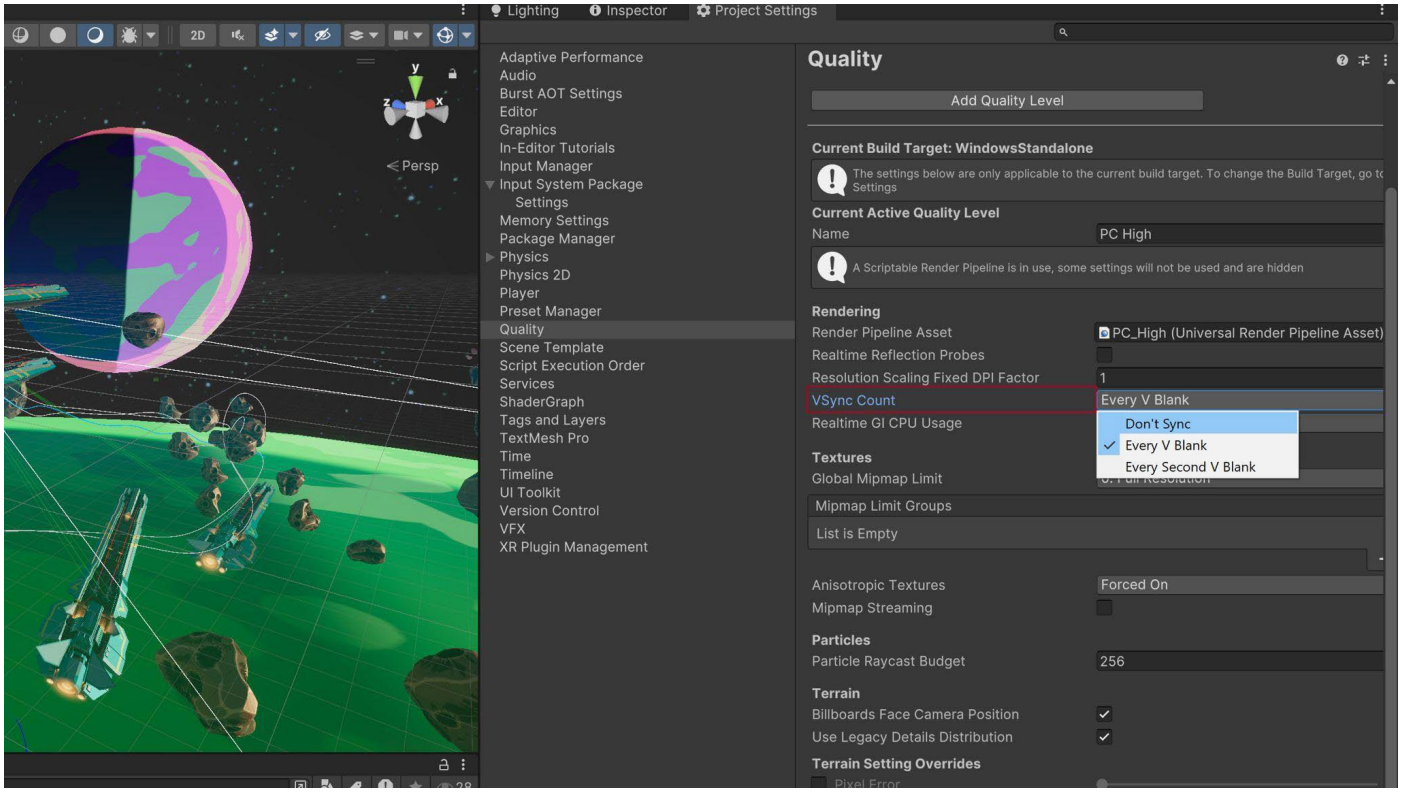
- モバイルプラットフォーム：モバイルデバイスは、ディスプレイのリフレッシュレート（新しいデバイスは多く場合 60Hz 以上）に合うように通常は Vsync を実行します。アプリケーションのフレームレートがこの目標を下回る場合、デバイスは前のフレームを保持し、スタッターや入力ラグが顕著に発生します。安定したフレームレートを維持するためにはレンダリングのパフォーマンスを最適化することが重要であり、異なる性能をもつ多様なモバイルデバイスにおいてスムーズな操作性を確保します。
- ウェブプラットフォーム：特に Unity Web などのウェブブラウザーにおいても Vsync を実行することが多く、ディスプレイのリフレッシュレートとの同期を確実にします。ブラウザー内で実行される追加のオーバーヘッドを考慮すると、一貫したフレームレートを維持するためのアプリケーションの最適化は、視覚的なパフォーマンス低下を回避するためには不可欠です。ウェブプラットフォームの性能はさまざまなので、あらゆる異なるブラウザーやデバイスをテストしてください。
- XR プラットフォーム：XR 環境においては、その体験が没入感の高い特性を持つことから、高く安定したフレームレートを維持することはなおさら重要です。多くの XR デバイスは 90Hz 以上で Vsync を実行し、いかなるフレームレートの低下もユーザーの不快感やモーションシックネスにつながる可能性があります。GPU がこれらの高い要求を満たすためには、レンダリングから物理演算まで、アプリケーションのあらゆる側面を最適化することが不可欠です。

XR、ウェブ、モバイルのプラットフォームにおける Vsync の管理方法を理解し、一貫したフレームレートを維持するためにアプリケーションを最適化することによって、多様なプラットフォーム上でユーザーの期待に応えるスムーズでより応答性の高い体験を提供することができます。

Vsync Count

Unity の Quality 設定内にある「**Vsync Count**」設定は、フレームのレンダリングがディスプレイのリフレッシュレートと同期される方法を決定します。「**Every V Blank**」（Vsync Count 1 に相当）に設定した場合、Unity はフレームのレンダリングを各垂直ブランクと同期し、ディスプレイのリフレッシュレートと合うようにフレームレートを効果的に制限します（例えば、60Hz = 60 FPS）。これはスクリーンのティアリングを防ぎスムーズなビジュアル出力の確保に役立ちます。

または、「**Every Second V Blank**」（Vsync Count 2 に相当）に設定した場合、フレームレートは半分になり、アプリケーションがリフレッシュレートのパフォーマンスを十分に維持することが難しい状況の時に役立つ可能性があります。Vsync 無効（Vsync しない）にすると、最大 FPS が得られますがスクリーンのティアリングが起こりえます。いくつかのプラットフォームでは、設定に関わらずハードウェアレベルで Vsync が実行されることがあります。



Quality 設定内にある VSync Count

グラフィックスと GPU の最適化

各フレームにおいて、Unity はレンダリングする必要のあるオブジェクトを決定し、ドローコールを作成します。ドローコールとはグラフィックス API にオブジェクト（三角形など）の描画を指示するもので、一方バッチとはドローコールのグループであり、まとめて実行されます。

プロジェクトの複雑さが増すにつれ、GPU の作業負荷を最適化するためのパイプラインが必要になります。**ユニバーサルレンダーパイプライン (URP)** は、レンダリングの選択肢としてフォワード、フォワード +、ディファードの 3 つをサポートしています。

フォワードレンダリングは、すべてのライティングをシングルパスで評価し、一般的にモバイルゲームではデフォルトとして推奨されています。フォワード + は Unity 2022 LTS とともに導入され、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大幅に増加します。ディファードモードは、動的ライトソースが多数あるゲームなど、特定のケースにおいては良い選択肢です。同じ物理ベースのライティングと、コンソールや PC からのマテリアルは、電話やタブレットにも拡張できます。

以下の表は、URP におけるレンダリングの 3 のオプションを比較しています。

機能	フォワード	フォワード +	ディファード
オブジェクトごとのリアルタイムライトの最大数	9	無制限、カメラごとの制限が適用	無制限
ピクセル単位の法線エンコーディング	エンコーディングなし (正確な法線値)	エンコーディングなし (正確な法線値)	2 つのオプション： G バッファでの法線の量子化（精度低下、パフォーマンス向上） 八面体エンコーディング（正確な法線。モバイル GPU では、パフォーマンスに大きな影響を与える可能性あり） 詳細については、G バッファへの法線のエンコーディングを参照してください。
MSAA	はい	はい	いいえ
頂点ライティング	はい	いいえ	いいえ
Camera Stacking	はい	はい	制限付きでサポート：Unity は、ディファードパスを使用して、ベースカメラのみをレンダリングし、フォワードレンダリングパスを使用して、すべてのオーバーレイカメラをレンダリングします。

Unity プロジェクトで URP を使用する場合の詳細については、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入門*」をご覧ください。

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、VR 、モバイル、ウェブのいずれにおいても、ターゲットハードウェアの制限を理解する必要があります。また、GPU の効率的なプロファイリング方法を理解する必要があります。プロファイリングによって最適化が望ましい影響をもたらしているかチェックや確認をすることができます。

- **VR**：VR ハードウェアは、スムーズで没入感のある体験を維持するために、高いフレームレート（通常 90 FPS 以上）と待ち時間の短さが要求されます。GPU は複雑なシーンを 2 回（一つの目に対して 1 回）レンダリングする必要があり、パフォーマンスとビジュアル忠実度の両方の最適化を慎重に行うことが要求されます。
- **モバイル**：デスクトップやコンソールと比べて、モバイルデバイスの処理能力とメモリは限られています。最適化においては、ドローコールの最小化、テクスチャサイズの削減、シンプルなシェーダーの使用に注目し、これらによってバッテリーの消耗やデバイスのオーバーヒートをすることなくスムーズなパフォーマンスを確保します。
- **ウェブ**：ウェブプラットフォームについては、特に Unity Web を使用する場合、ブラウザー環境で実行する際の制約とパフォーマンスとのバランスを取らなくてはなりません。最適化においては、ビルドサイズの削減、ロード時間の最小化、あらゆる異なるブラウザーやハードウェアの設定に対する互換性の確保を優先する必要があります。

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

プロファイリングの際は、ベンチマークから始めるのが有効です。ベンチマークを行うと特定の GPU からどのようなプロファイリング結果を得るべきかが分かります。

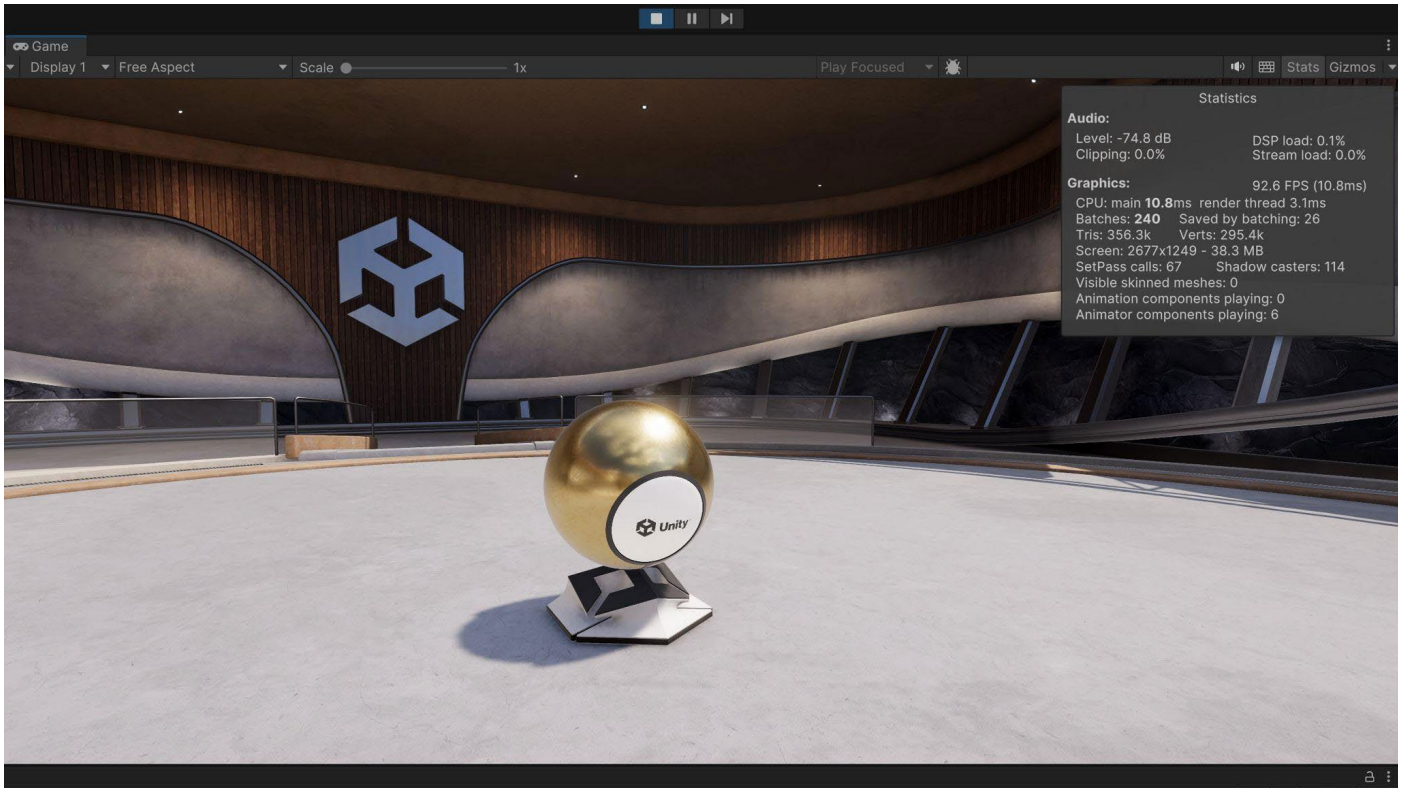
GPU とグラフィックスカードのさまざまな業界標準ベンチマークの一覧については、GFXBench を参照してください。このウェブサイトでは、現在利用可能な GPU の概要と、各 GPU の位置づけについて紹介しています。

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 のレンダースレッド**：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
- **バッチ**：同時に描画されるドローコールのグループ
- **Tris（三角形）と Verts（頂点）**：メッシュのジオメトリの複雑さ
- **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。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。



リアルタイムでレンダリング情報を表示する統計ウィンドウ

ドローコールを減らす

ゲームオブジェクトを描画するために、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。CPU は必要なデータを準備して GPU へ送信する必要があり、その後 GPU はコマンドを処理してオブジェクトをレンダリングするので、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、ドローコール間の頻繁な状態変更は、CPU オーバーヘッドをさらに増加させる可能性があります。

PC やコンソールハードウェアは大量のドローコールを処理できる一方で、オーバーヘッドは顕著で最適化が必要なほどです。モバイルデバイス、VR ヘッドセット、ウェブブラウザーにおいては、ドローコールの最適化はパフォーマンスを維持するために不可欠です。ドローコール数を減らすことで、特にリソースに制限があるプラットフォームにおいては、よりスムーズで効率的にレンダリングができるようになります。

特にウェブ、VR、モバイルのプラットフォームのパフォーマンスを最適化するために、ドローコールを減らすことは必須です。これを達成するための重要な戦略は以下の通りです。

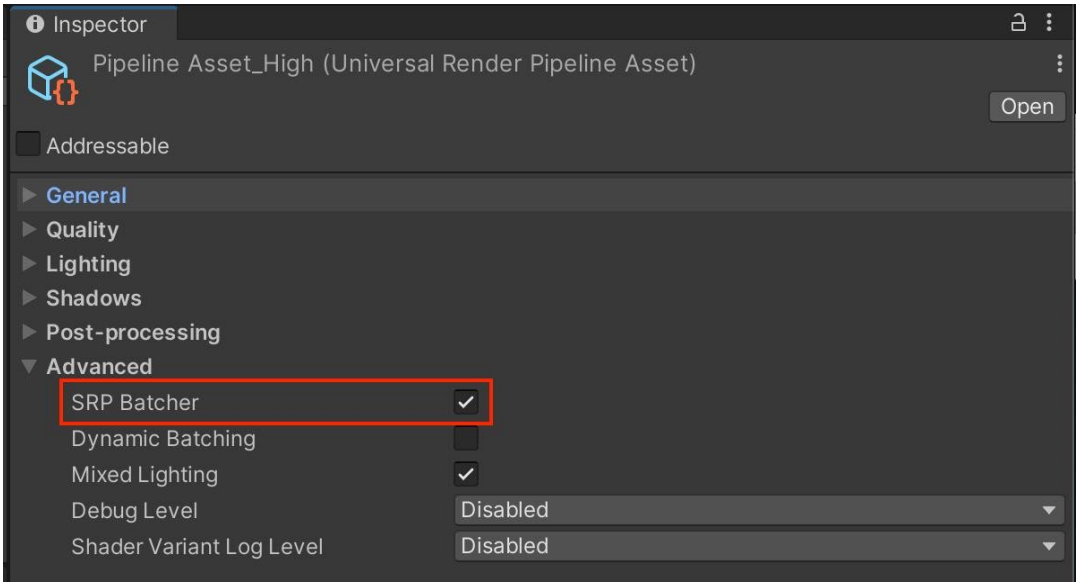
1. **テクスチャアトラスを使用する**：複数のテクスチャを単一のテクスチャアトラスにまとめ、テクスチャバインド数およびドローコール数を最小にします。これはウェブやモバイルの環境においては特に重要で、削減状態の変化がレンダリング効率の向上につながります。
2. **マテリアルを最適化する**：プロジェクトのマテリアル数やシェーダー数を制限してください。共有されたマテリアルのバッチがさらに容易になり、ドローコールオーバーヘッドが減ります。
3. **LOD（Level of Detail）を実行する**：LOD 技術を使って離れているオブジェクトの複雑性を減少させ、カメラから遠いオブジェクトのドローコール数を減らします。高いフレームレートを維持することがモーションシックネスを防ぐために不可欠な VR や、処理能力に制限があるモバイルプラットフォームにとっては、このアプローチは重要です。
4. **カリング技術を適用する**：錐台カリングおよびオクルージョンカリングを行い、可視オブジェクトのみレンダリングします。カメラのビューの外側にあるオブジェクトや他のジオメトリによってわかりにくいオブジェクトの描画をしないことで、ドローコール数を削減できます。これによってあらゆるプラットフォーム、特にリソースに制限があるウェブやモバイル環境にあるプラットフォームのパフォーマンスが向上します。

ドローコールをバッチ処理する

ドローコールのバッチ処理は、メッシュを結合して、Unity がより少ないドローコールでレンダリングできるようにする最適化の方法です。

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 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 設定の **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います。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、ランタイム中に（例えば、非可動部品のプロシージャルレベルを生成した後で）このような静的バッチを自分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、バッチ処理の効果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。さらに、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、ドローコールバッチ処理のドキュメントを参照してください。

GPU Resident Drawer

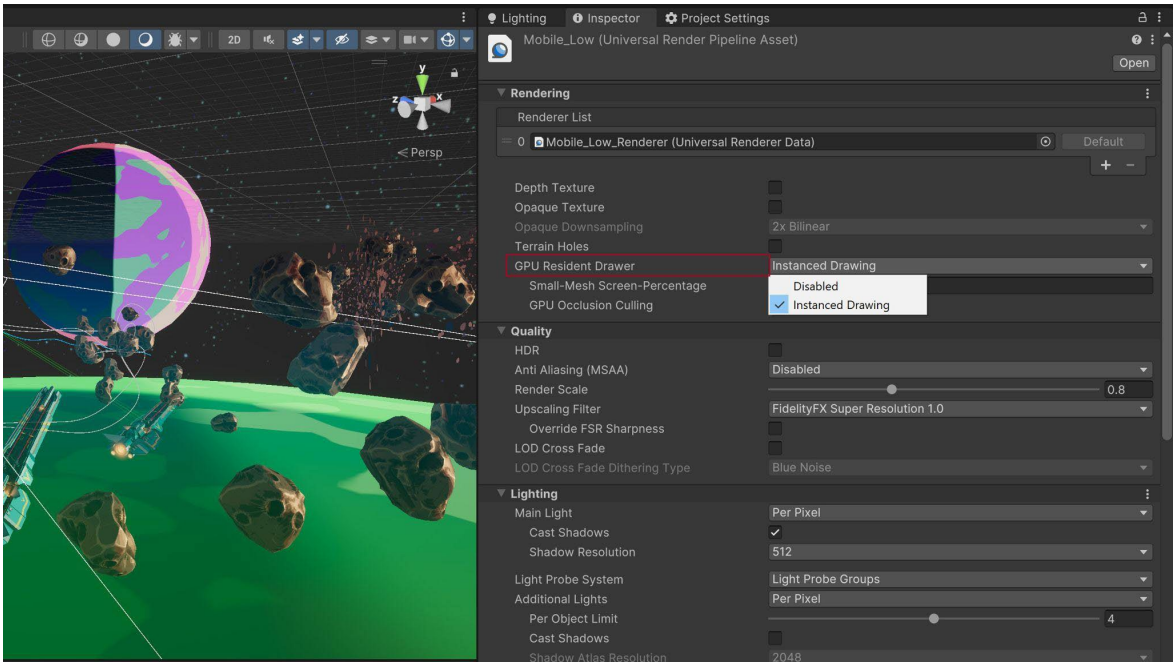
GPU Resident Drawer（URP および HDRP で利用可能）は、CPU 時間を最適化するように設計された GPU 駆動のレンダリングシステムで、パフォーマンスを向上させます。それはクロスプラットフォームレンダリングをサポートし、Vulkan や Metal を使った高性能モバイルプラットフォームも含まれており、既存のプロジェクトでそのまま使用できるように設計されています。

GPU Resident Drawer は BatchRendererGroup API を使って GPU インスタンス化とともにゲームオブジェクトを描画し、ドローコール数を減らして CPU の処理時間を短縮します。GPU Resident Drawer は以下の場合のみ可能です。

— フォワード + レンダリングパス

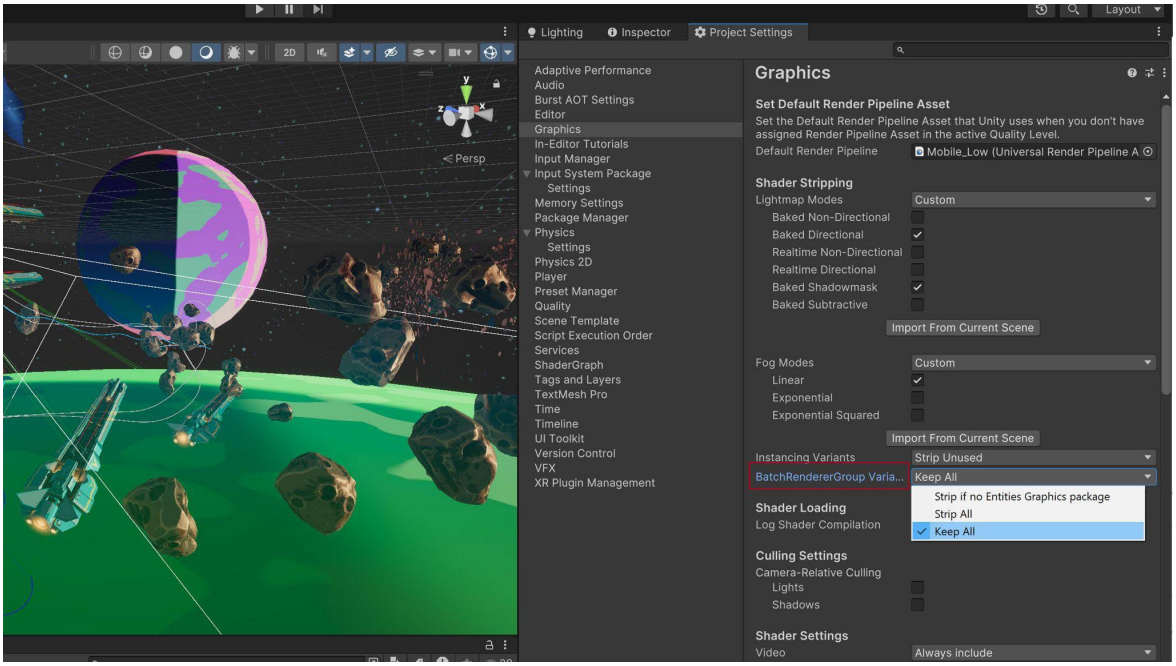
— OpenGL ES を除く、コンピュートシェーダーをサポートするグラフィックス API およびプラットフォーム

— Mesh Renderer コンポーネントを持つゲームオブジェクト



GPU Resident Drawer：レンダーパイプラインアセットで「**Instanced Drawing**」を選択

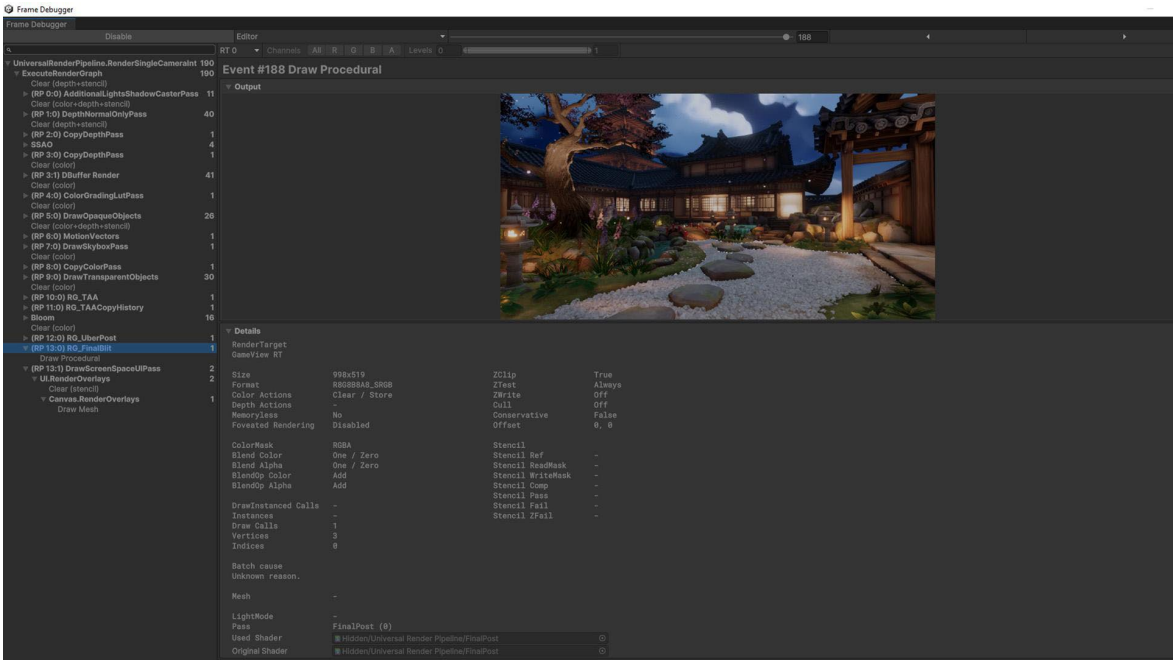
Instanced Drawing オプションを選択すると、「BatchRenderGroup Variants の設定は『Keep All』でなければならない」という旨の UI 警告メッセージが表示されることがあります。グラフィックス設定でこのオプションを調整すると、GPU Resident Drawer の設定が完了します。



グラフィックス設定で、BatchRenderGroup Varient の設定を「**Keep All**」にする。

フレームデバッガーを使う

フレームデバッガーは、各ドローコールからどのようにそれぞれのフレームが作成されたのかを示しています。これはシェーダーのプロパティのトラブルシューティングをするのに非常に貴重なツールであり、ゲームのレンダリング方法を分析するのに役立ちます。



フレームデバッガーは各フレームを分割して別々のステップに分ける。

フレームデバッガは初めてですか？こちらの入門用チュートリアルをご覧ください。

Graphics Jobs を分割する

このスレッドモードは、マルチデスクトップやコンソールプラットフォームでサポートされており、CPU マルチスレッドパフォーマンスを向上させるのが目的です。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。多くのオブジェクトやテクスチャを含む複雑なシーンなど、ドローコールがさらに多いシーンでは、パフォーマンスが大幅に向上します。

ダイナミックライトを増やしすぎないようにする

XR、モバイル、ウェブの開発の場合、動的ライトの使用を制限することが重要で、特にフォワードレンダリングを使用するときは重要です。動的ライトはパフォーマンスに大きな影響を及ぼす可能性があり、フレームレート低下や電力消費増加につながり、リソースに制限がある環境においては特に重要です。

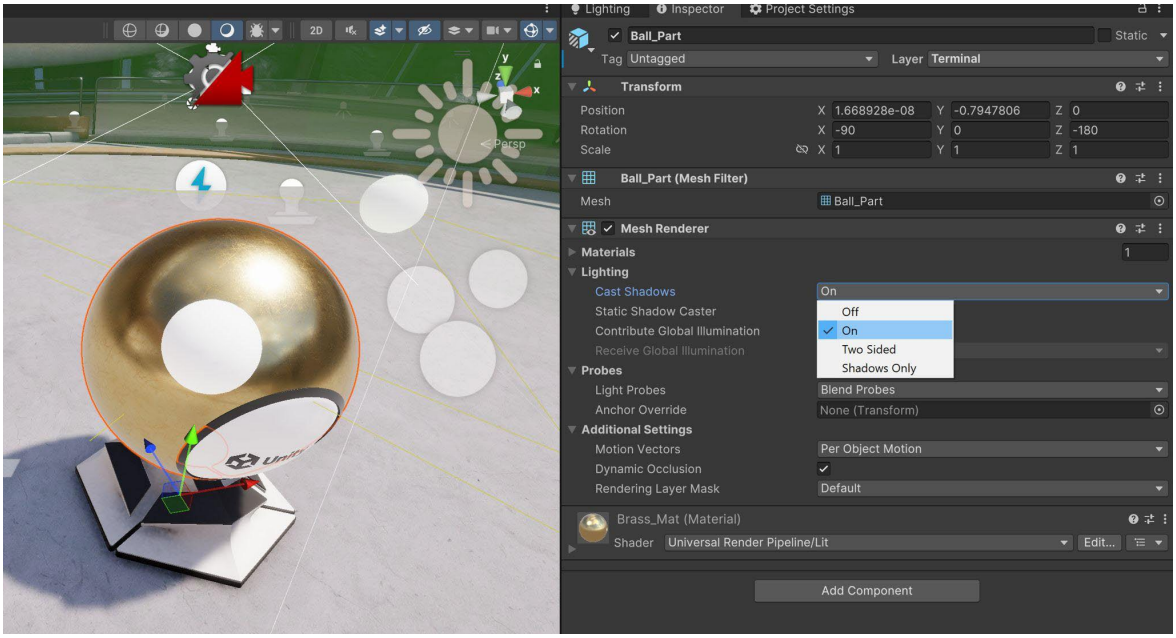
その代わりに、動的オブジェクトにはカスタムシェーダーエフェクトやライトプローブなどを使用する選択肢を検討しましょう。パフォーマンスに高負荷をかけずにライティングのシミュレーションができます。静的オブジェクトにはベイクライティングがより効率的なオプションであり、ランタイムオーバーヘッドを発生させることなく高品質のライティングを提供します。ライティングを慎重に管理することによって、XR、モバイル、ウェブのアプリケーションにおいてビジュアル品質を維持しつつパフォーマンスを最適化できます。

UPR の特定の制限やビルトインレンダーパイプラインのリアルタイムのライトについては、この feature comparison table を参照してください。

シャドウを無効にする

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可能な限り影を無効にしましょう。

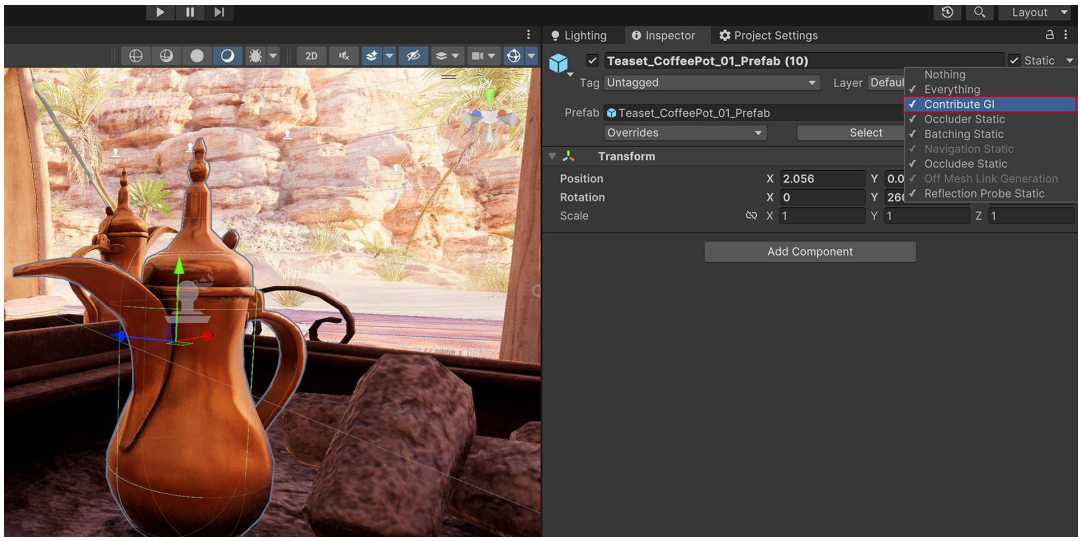
また、シンプルなメッシュや四角形をキャラクターの下に配置し、ぼかしたテクスチャを適用することで、偽の影を作るという方法もあります。あるいは、カスタムシェーダーを使ってブロブシャドウを作ることができます。



影のキャストを無効にしてドローコールを抑える。

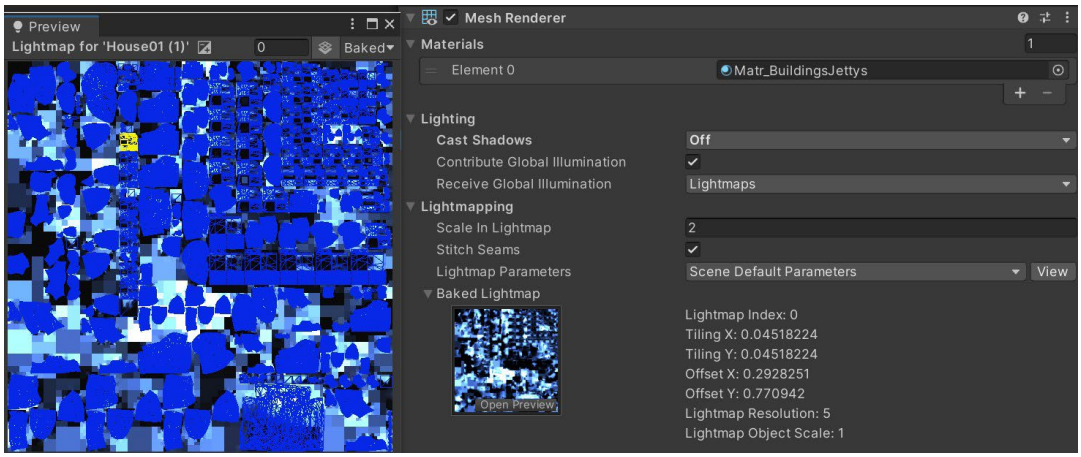
ライティングをライトマップにベイクする

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。



Contribute GI 設定を有効にする。

それによってベイクシャドウやライティングは、ランタイム時にパフォーマンスヒットが発生することなくレンダリングされます。プログレッシブな CPU ライトマッパーや GPU ライトマッパーはグローバルイルミネーションのベイクを高速化できます。

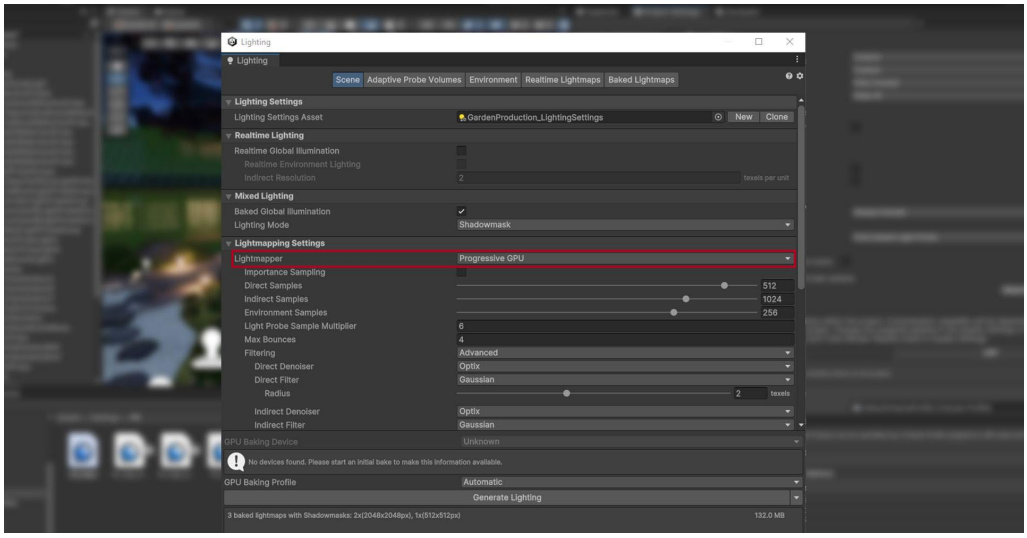


Lightmapping Settings（「**Windows**」＞「**Rendering**」＞「**Lighting Settings**」）と**ライトマップのサイズ**を調整してメモリ使用量を抑える。

マニュアルガイドおよびライトの最適化についてのこの動画(日本語キャプション)に従って Unity でライトマッピングを始めましょう。

GPU ライトベイク

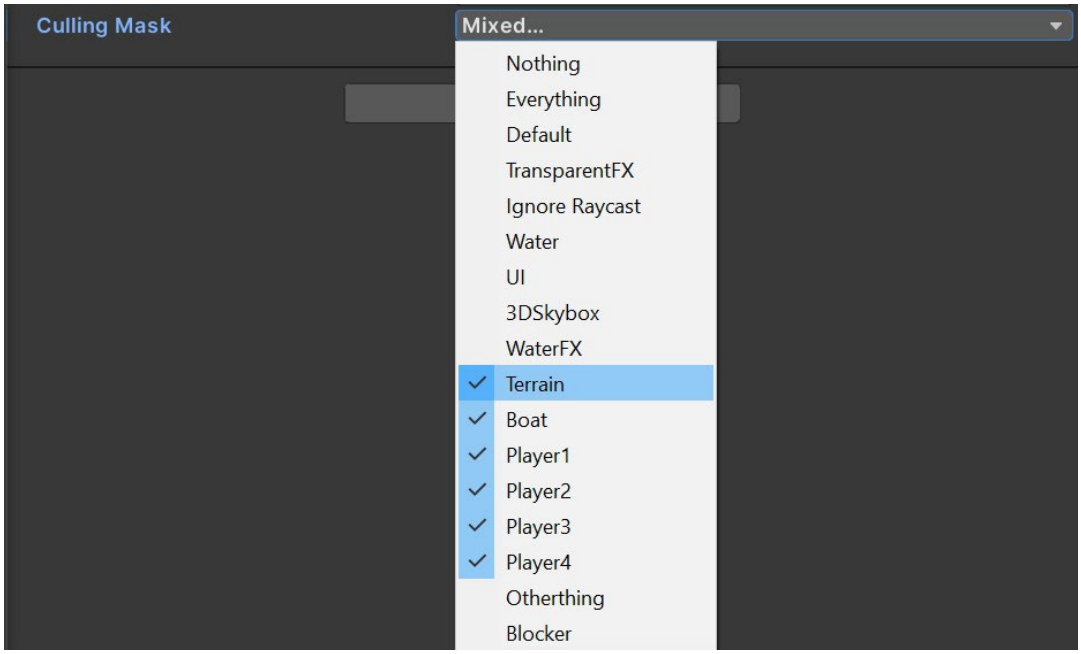
「Progressive GPU Lightmapper」は Unity 6 で製品利用が可能です。GPU のパワーを活用することで、ライティングデータの生成を劇的に高速化するように設計されており、従来の CPU によるライトマッピングに比べてベイク時間を短縮できます。このシステムは、コードベースを簡素化し、より予測可能な結果をもたらす新しいライトベイクのバックエンドを導入しています。加えて、GPU の最小要件が 2GB まで縮小されため、幅広い開発者がこの機能を利用しやすくなりました。ランタイム中にライトプローブの位置を
移動できる新しい API も追加されました。これは、プロシージャル生成されたコンテンツで特に有用で、さまざまな利便性の改善も含まれています。



「Progressive GPU Lightmapper」を選択する。

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することができる。

アダプティブプローブボリューム

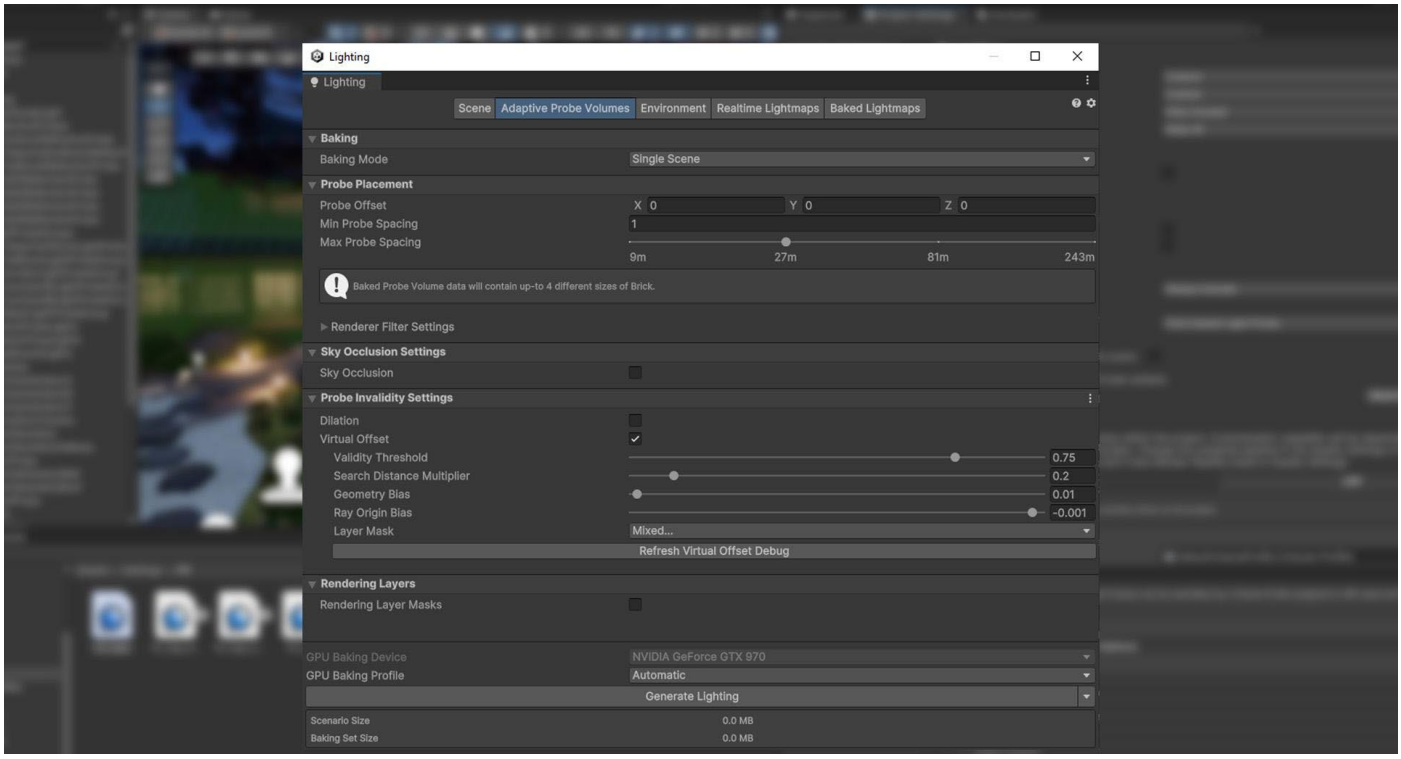
Unity 6 では、アダプティブプローブボリューム（APV）が導入され、Unity でグローバルイルミネーションを処理するための洗練されたソリューションが提供され、複雑なシーンで動的かつ効率的なライティングを実現できるようになりました。APV は、特にモバイルやローエンドのデバイスにおいて、パフォーマンスとビジュアル品質の両方を最適化することができると同時に、ハイエンドのプラットフォームに対しては高度な機能を提供することができます。

Unity のアダプティブプローブボリューム（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。今では URP は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、VFX パーティクル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ています。

URP 3D サンプルのオアシス環境に APV を配置する

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。開発者は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ができます。またこのシステムは、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 はワークフローをさらに改良し、ライトマップまたはリフレクションプローブからの APV の独立したベイクを可能にします。

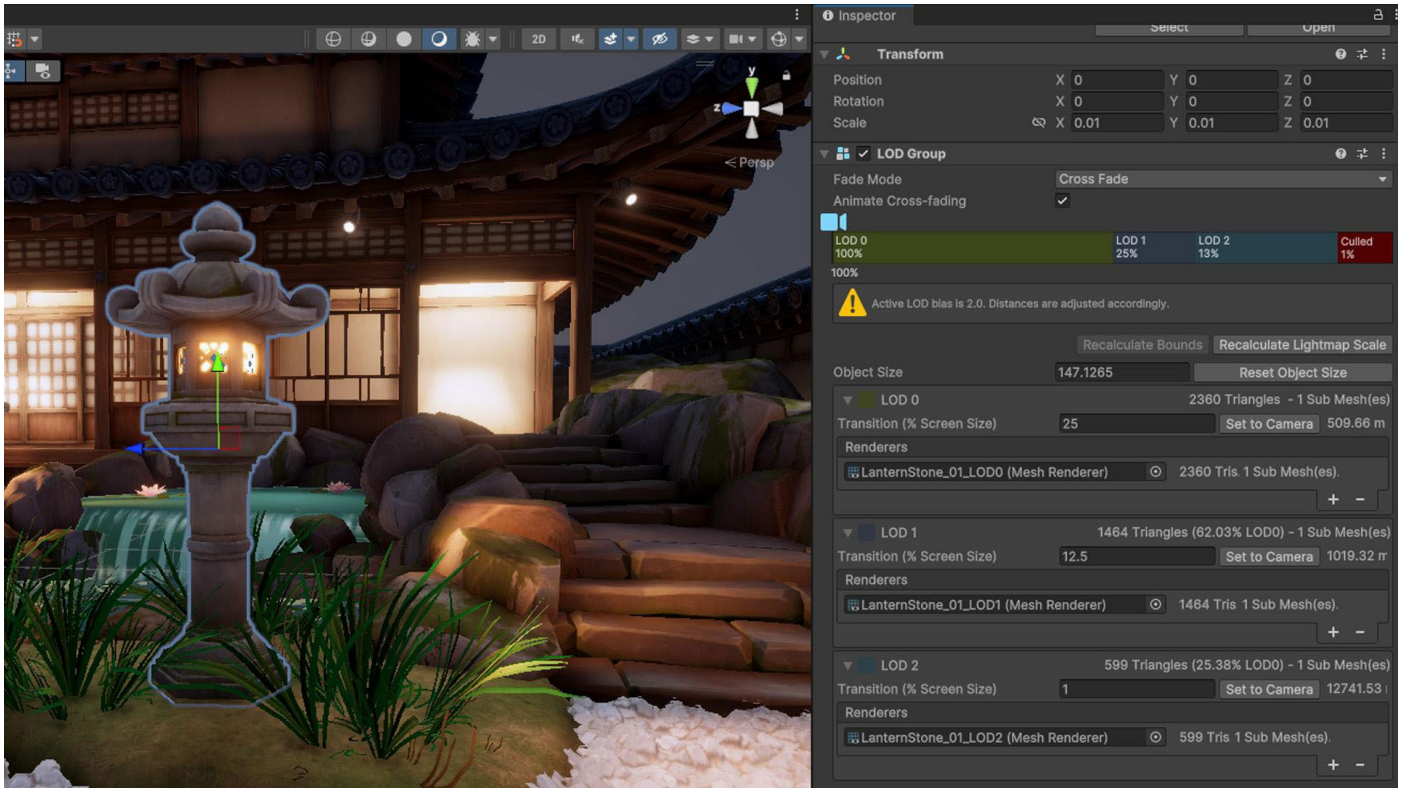
開始するには、GDC 2023 の「Efficient and impactful lighting with Adaptive Probe Volumes」をご覧ください。



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

詳細レベル（LOD）を使用する

オブジェクトがカメラから遠ざかるに従って、Level of Detail（LOD）は GPU パフォーマンスを向上させるために、より単純なマテリアルとシェーダーでより単純な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。



LOD グループを使ったメッシュの例

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。

オクルージョンカリングを使って隠れたオブジェクトを削除する

他のオブジェクトの後ろに隠れているオブジェクトは、レンダリングされリソースが費やされてしまう可能性がまだあります。オクルージョンカリングを使ってそれらを破棄しましょう。

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 Occluders** または **Occludees** とマークし、**「Window」＞「Rendering」＞「Occlusion Culling」**ダイアログからベイクします。各シーンに必要ではないものの、カリングは特定のケースにおいてはパフォーマンスを向上させることができるので、オクルージョンカリングを有効にする前後でプロファイリングを必ず行いパフォーマンスの向上をチェックしてください。

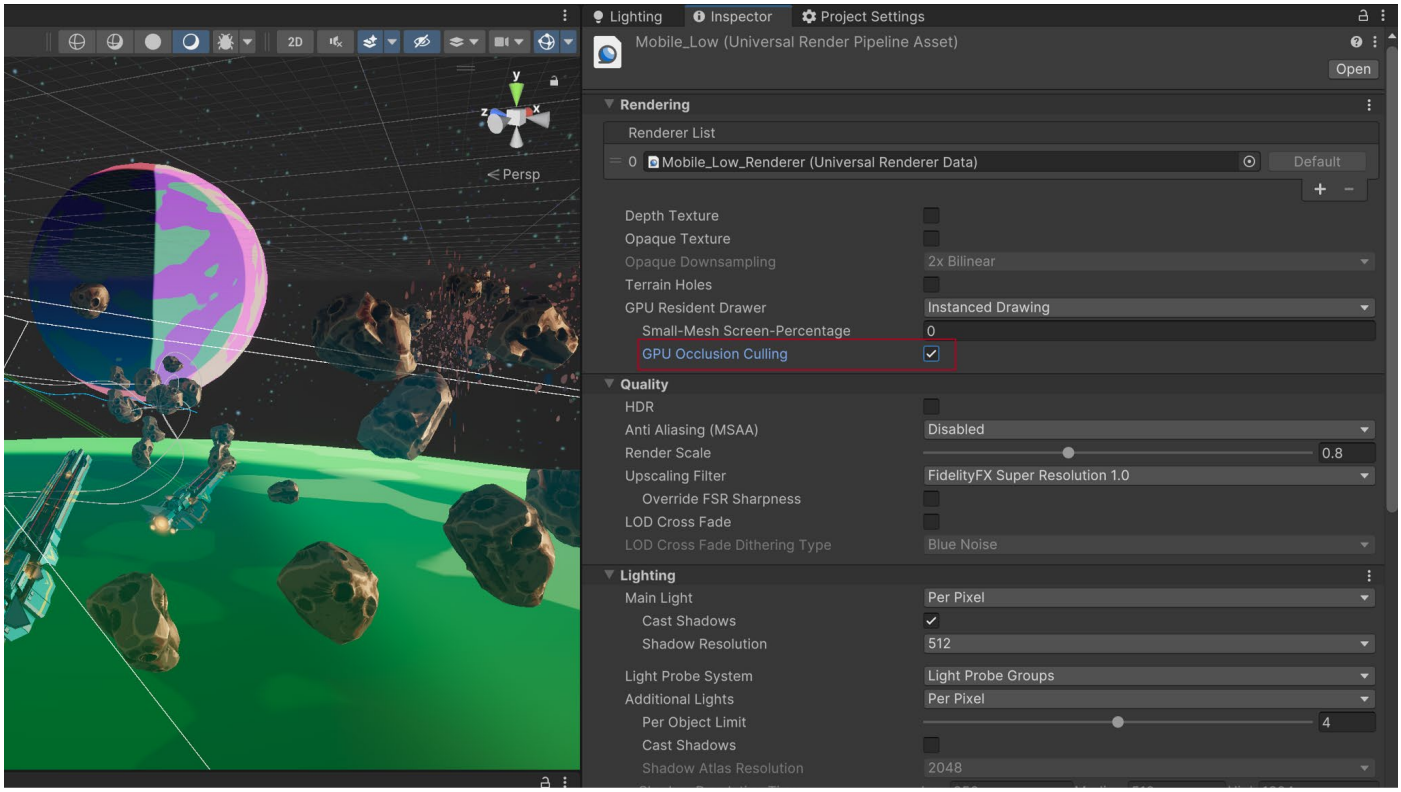
詳細については、チュートリアル「オクルージョンカリングの使用方法」をご覧ください。

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レンダリングのパフォーマンスを大幅に向上させ、特に複雑なジオメトリや多数のオクルージョンされたオブジェクトを含むシーンにおいてパフォーマンスを向上させます。GPU オクルージョンカリングは、各フレームのオーバードローの量を削減することによってゲームオブジェクトのパフォーマンスを向上させます。すなわち、レンダラーは不可視のものを描画するリソースを無駄にしていないということです。従来これは 3D 環境におけるパフォーマンスの大きなボトルネックとなる課題でした。GPU オクルージョンカリングの重要機能は以下の通りです。

5. **GPU の高速化**：オクルージョンカリングを CPU に大きく依存していた以前のバージョンとは異なり、Unity 6 は GPU の高速化を活用しています。この転換によってさらに効率的にリアルタイムで計算ができ、CPU のオーバーヘッドを減らし、パフォーマンスを犠牲にすることなくより複雑なシーンが可能になります。
6. **GPU Resident Drawer との統合**：GPU オクルージョンカリングは、大きなオブジェクト群とその可視性を扱う GPU Resident Drawer と連携して、静的オブジェクトおよび動的オブジェクトの両方のレンダーパイプラインをいっそう最適化します。
7. **動的オブジェクトおよび静的オブジェクトのカリング**：Unity 6 のオクルージョンカリングシステムは、静的オブジェクトおよび動的オブジェクトをさらに効果的に管理できます。動的オブジェクトのカリングはポータルをベースとしたシステムを使って行い、可視オブジェクトの処理のみ行うことができ、それらがシーンの中で動いていても処理が可能です。
8. **ベイクおよびリアルタイムの調整：**開発者はエディターでオクルージョンデータをベイクでき、その後ランタイム時に使用します。このプロセスでは、シーンはセルに分割されてセル間の可視性が計算され、それによってカメラが動くにつれてリアルタイムでの調整が可能です。そのシステムはエディターのオクルージョンカリングの可視化もサポートしており、開発者がシーンをさらに最適化するのに役立ちます。
9. **メモリ管理：**Unity 6 はオクルージョンデータのメモリフットプリントを管理するツールを提供し、オクルージョンカリングのプロセスの微調整を可能にして、パフォーマンスとメモリ使用量のバランスを取ります。

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。



Render Pipeline Asset の GPU Occlusion Culling オプション

モバイルネイティブの解像度を使用しない

電話やタブレットの高度化が進む中、新しいデバイスは高解像度を売りにする傾向があります。

Screen.SetResolution(width, height, false) を使用して出力解像度を下げ、パフォーマンスをいくらか回復させましょう。複数の解像度でプロファイルを取ることで、画質と速度の最適なバランスを見つけることができます。

カメラの使用を制限する

有益な処理かどうかにかかわらず、有効な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。ローエンド寄りのモバイルプラットフォームでは、カメラ 1 つあたり最大で 1 ミリ秒の CPU 時間を消費することもあります。

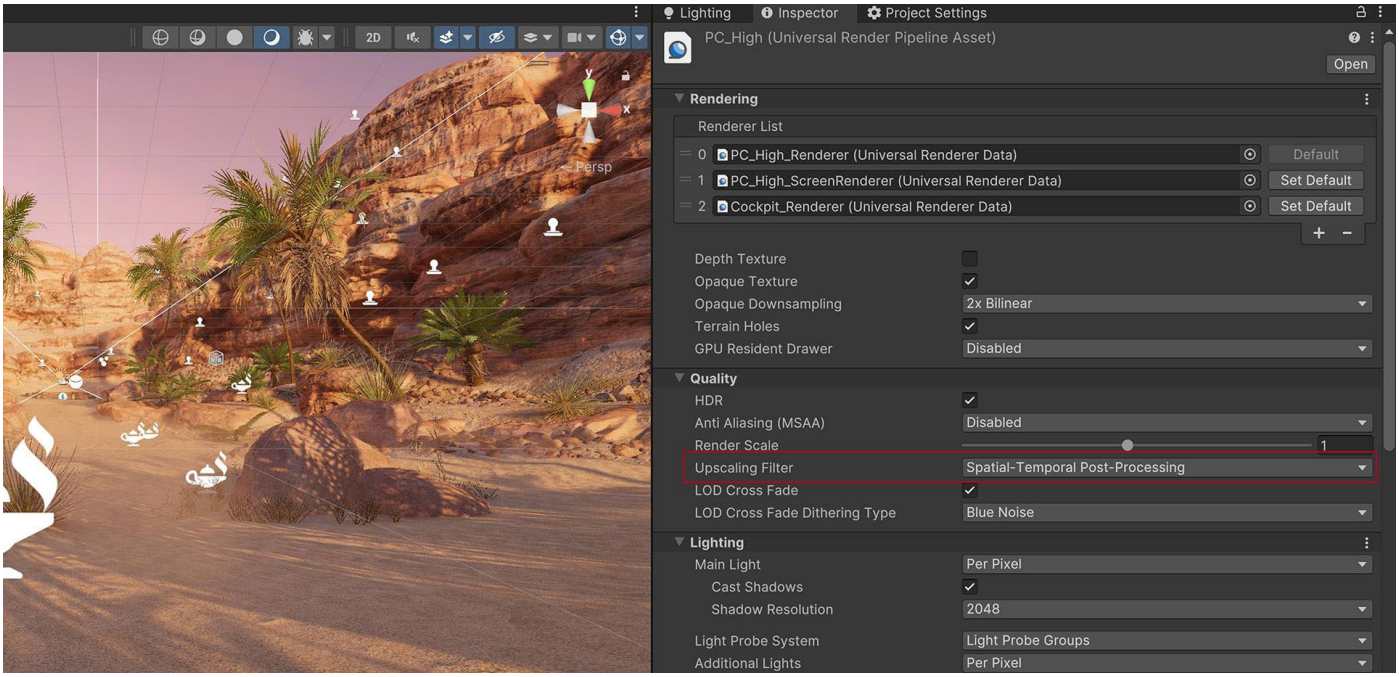
Spatial-Temporal Post-Processing

Spatial-Temporal Post-Processing (STP) は、モバイルデバイスからコンソールや PC まで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 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いパフォーマンスを達成することを容易にします。

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

シェーダー

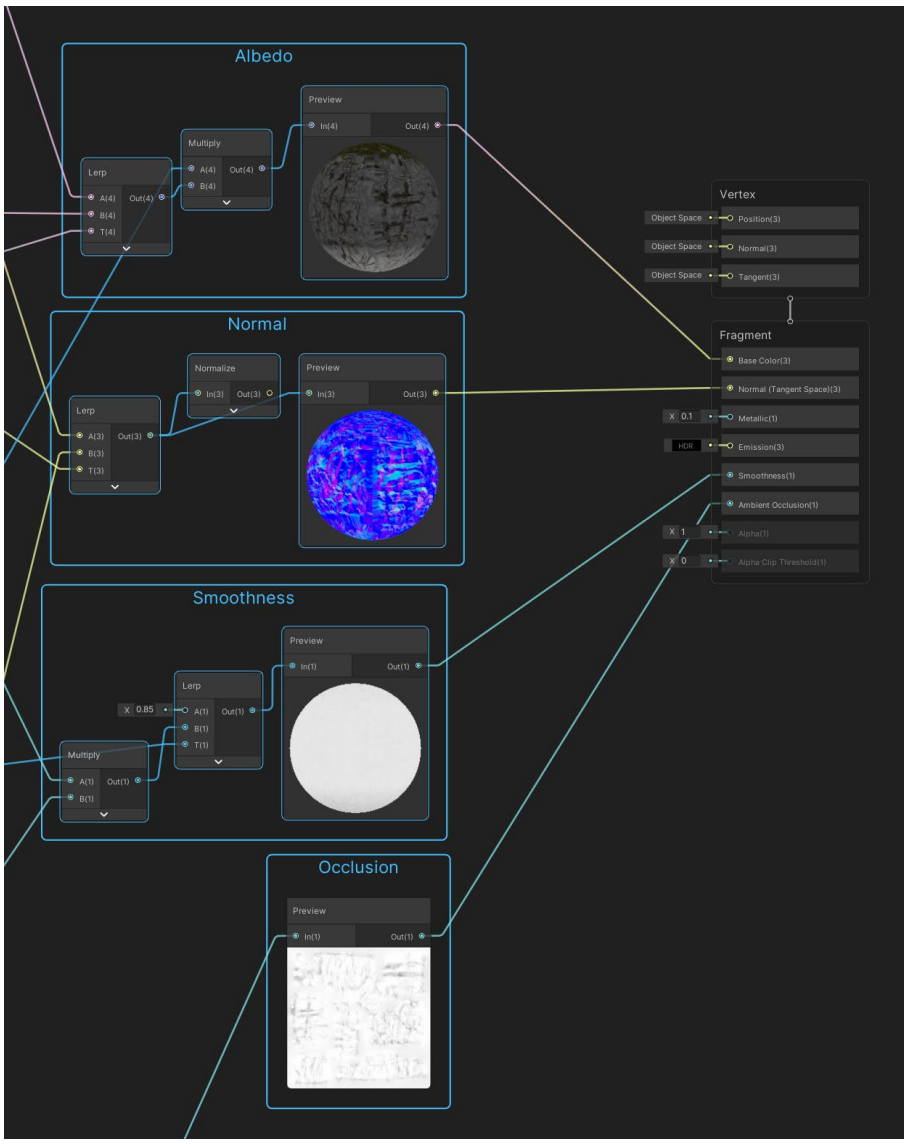
シェーダーはシンプルかつ最適にしておく

URP はモバイルプラットフォームに最適なさまざまな軽量の Lit シェーダーと Unlit シェーダーを提供し、これらはウェブ、モバイル、XR のプロジェクトにとっては最高の出発点となります。パフォーマンスを最大化するために、シェーダーのバリエーションは最小限にしてください。複数のバリアントはランタイム時のメモリ使用量に影響を及ぼすことがあり、リソースに制限があるデバイスにおいては特にその可能性があるからです。

デフォルトの URP シェーダーが特定のニーズを満たさない場合は、Shader Graph を使ってカスタマイズでき、プロジェクトのためにシェーダーを視覚的に設計し最適化することができます。以下にシェーダー最適化のヒントをいくつか挙げます。

- **計算を最小限にする**：操作数を減らしてシェーダーをシンプルにします。特に各ピクセルの計算が必要なフラグメントシェーダーにおいてはそのようにします。複雑な数学演算や高負荷な分岐ロジック（if ステートメントなど）を避けます。これらによって特にモバイルや XR のアプリケーションにおいては GPU に対する負荷が高くなるからです。
- **組み合わせたテクスチャを使う**：オクルージョン、ラフネス、メタリック（ORM）マップのような組み合わせたテクスチャを活用してテクスチャルックアップ数を削減します。このアプローチによって複数のマップが単一のテクスチャにまとめられ GPU の作業負荷が軽減されるので、モバイル、ウェブ、XR のプラットフォームにおけるパフォーマンスの維持には重要です。
- **Shader Graph を最適化する**：シェーダーグラフを使用する場合、シェーダーロジックの合理化に注目してパフォーマンスを向上させましょう。各シェーダーの効率が全体のパフォーマンスに直接影響を及ぼすモバイルや XR のアプリケーションにおいて、これは特に重要です。

— **定期的にプロファイルする**：ウェブ、モバイル、XR のいずれでも、ターゲットデバイスのシェーダーに対して継続的にテストおよびプロファイリングを行い、パフォーマンス要件を満たすようにします。定期的なプロファイリングは、潜在的な課題を早期に捉え、それに応じて各プラットフォームの特定のニーズのために最適化するのに役立ちます。



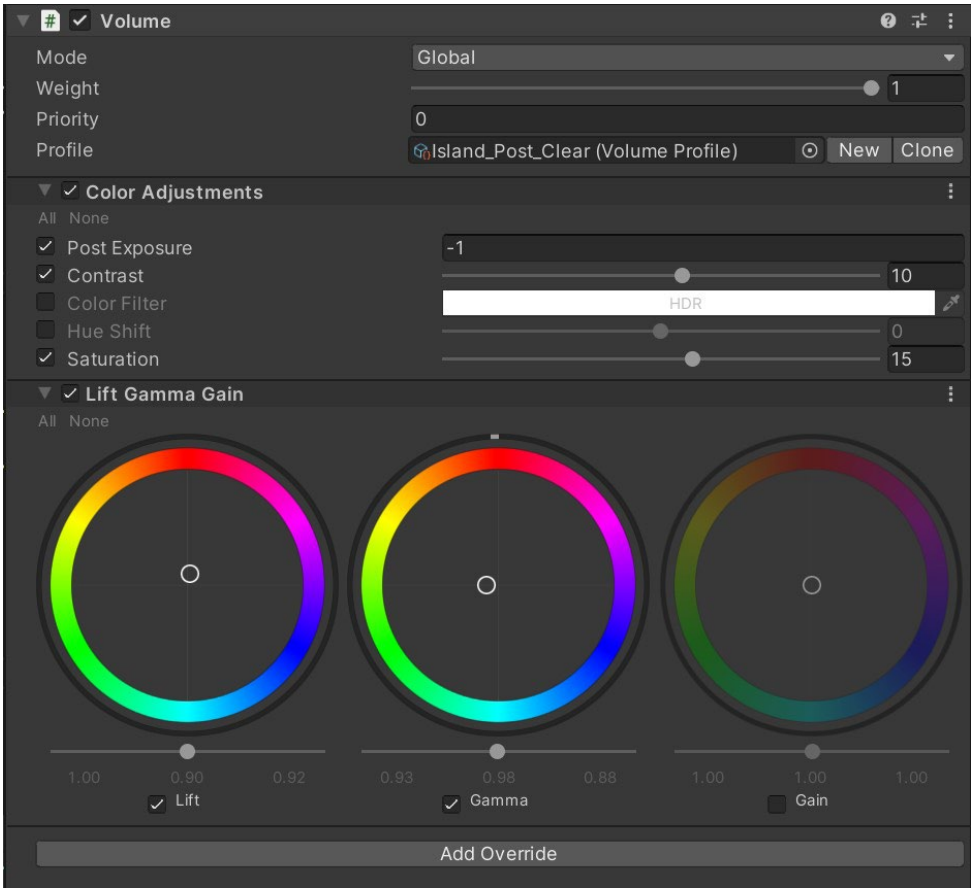
Shader Graph でカスタムシェーダーを作成する。

オーバードローとアルファブレンドを最小限に抑える

不要な透明イメージや半透明イメージの描画を避け、ほとんど不可視のイメージやエフェクトを重ねないでください。モバイルプラットフォームは結果的に生じるオーバードローやアルファブレンドの影響を受けます。オーバードローは RenderDoc グラフィックスデバッガーを使ってチェックできます。さまざまなライティング、レンダリング、マテリアルのプロパティを可視化するレンダリングデバッガーを活用することもできます。可視化することでレンダリングの課題を特定でき、シーンやレンダリングの構成を最適化できます。

ポストプロセスエフェクトを制限する

輝きのような全画面のポストプロセスエフェクトは、パフォーマンスを劇的に低下させることがあります。タイトルのアートの方向性に従って慎重に使用してください。モバイル、XR、ウェブにおいて、ポストプロセスはパフォーマンスに対するボトルネックのよくある原因になりうるので、特に慎重にベンチマークを行い、それに応じてアートの観点からの選択をしてください。



モバイルアプリケーションではポストプロセスエフェクトはシンプルに保つ。

Renderer.material には注意を払う

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。

SkinnedMeshRenderer を最適化する

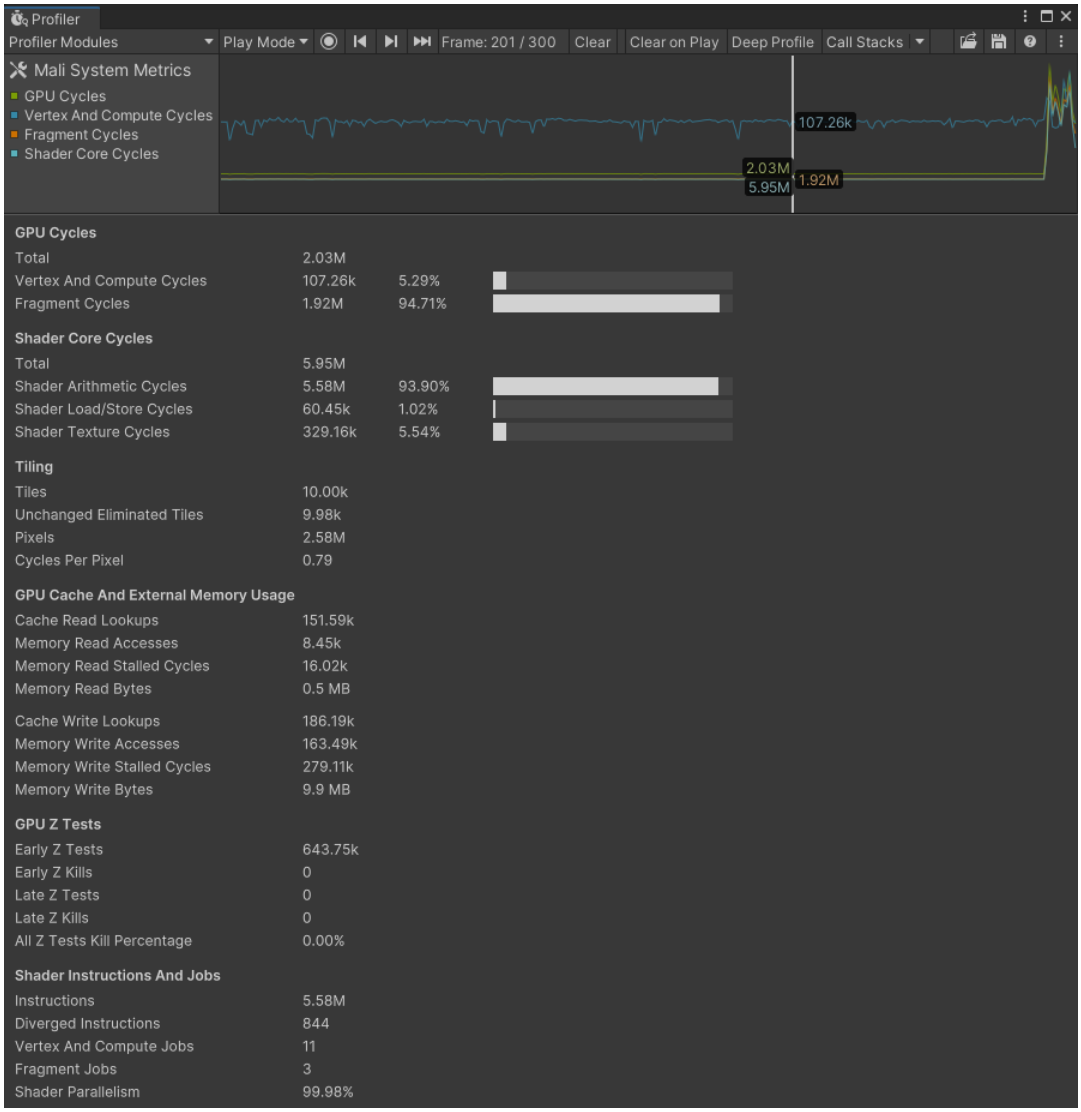
スキンメッシュのレンダリングは高い負荷がかかります。**SkinnedMeshRenderer** を使用しているすべてのオブジェクトが、本当にそれを必要としているか確認しましょう。時々アニメーションする程度のゲームオブジェクトの場合は、**BakeMesh** 関数を使ってスキンメッシュを静的ポーズに固定し、ランタイム時にはよりシンプルな **MeshRenderer** に切り替えます。

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。

System Metrics Mali

System Metrics Mali パッケージを活用して、ARM GPU を使うデバイスの低レベルのシステムやハードウェアパフォーマンスメトリクスにアクセスすることもできます。これには、Unity Profiler の低レベルの GPU メトリクスを監視できること、ランタイム時に低レベルの GPU メトリクスにアクセスするために Recorder API を使用できること、継続的な統合テストラン付きのパフォーマンステストを自動化できることも含まれています。



Mali System Metrics Profiler Module

Unity のライティングワークフローの詳細については以下のガイドをご覧ください。

- 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入門
- URP の 2D ライトおよびシャドウテクニック
- 2D ゲームのためのライティングおよび AI テクニック(日本語キャプション)
- ユニバーサルレンダーパイプラインクックブック：シェーダーと視覚エフェクトのレシピ

ユーザーインターフェース

Unity は従来の Unity UI と新しい UI Toolkit の 2 つの UI システムを提供しています。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。

Unity UI パフォーマンスの最適化のヒント

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

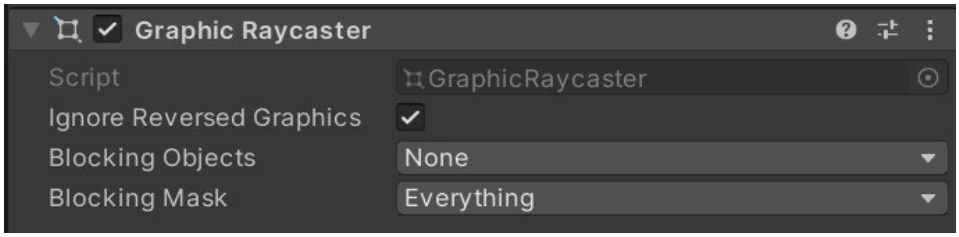
ゲーム中に散発的にしか表示されない UI 要素（例えば、キャラクターがダメージを受けたときに表示されるヘルスバー）があるとします。不可視の UI 要素がアクティブであれば、まだドローコールを使っているかもしれません。不可視の UI コンポーネントを明示的に無効にし、必要に応じて再び有効にしてください。

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再構築する必要がありません。

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

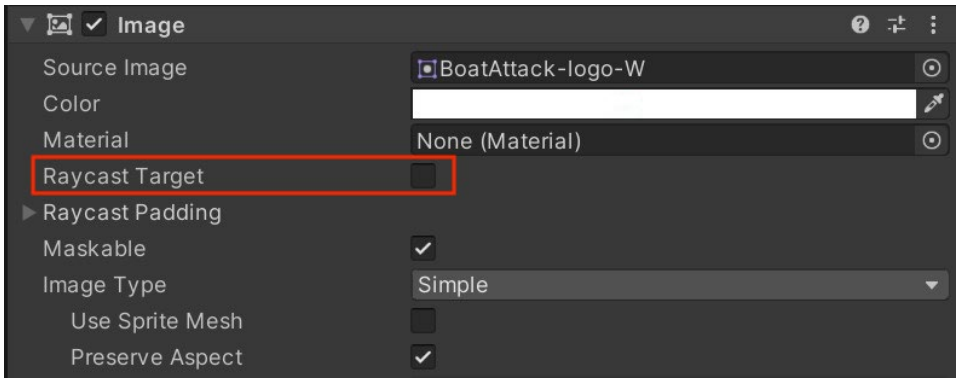
オンスクリーンタッチやクリックなどの入力イベントには、**GraphicRaycaster** コンポーネントが必要です。これは単純に、画面上の各入力点をループし、それが UI の RectTransform 内にあるかどうかを確認します。

デフォルトの **GraphicRaycaster** を階層内のトップの Canvas から取り除きます。その代わりに専用の **GraphicRaycaster** を相互作用が必要な各要素（ボタン、スクロールレクトなど）に加えます。



デフォルトで有効になっている Reversed Graphics を無効する。

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。

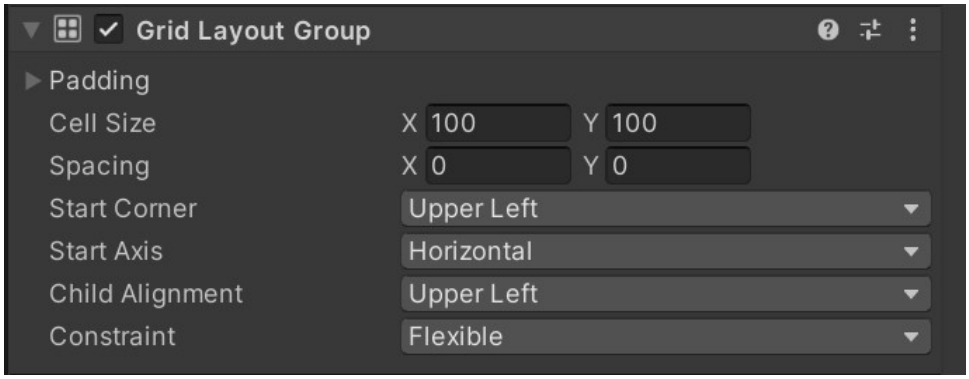


Raycast Target は可能な限り無効にする。

レイアウトグループを避ける

Layout Group の更新は非効率的なので、使用は控えます。しかしながら、コンテンツが動的でない場合は使用を避けるのが一般的には最適です。代わりにプロポーショナルレイアウト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ります。

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（例えば、数百のアイテムがあるインベントリ画面）、すべてのアイテムに対して UI エレメントを作成するのではなく、より小さな UI エレメント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。

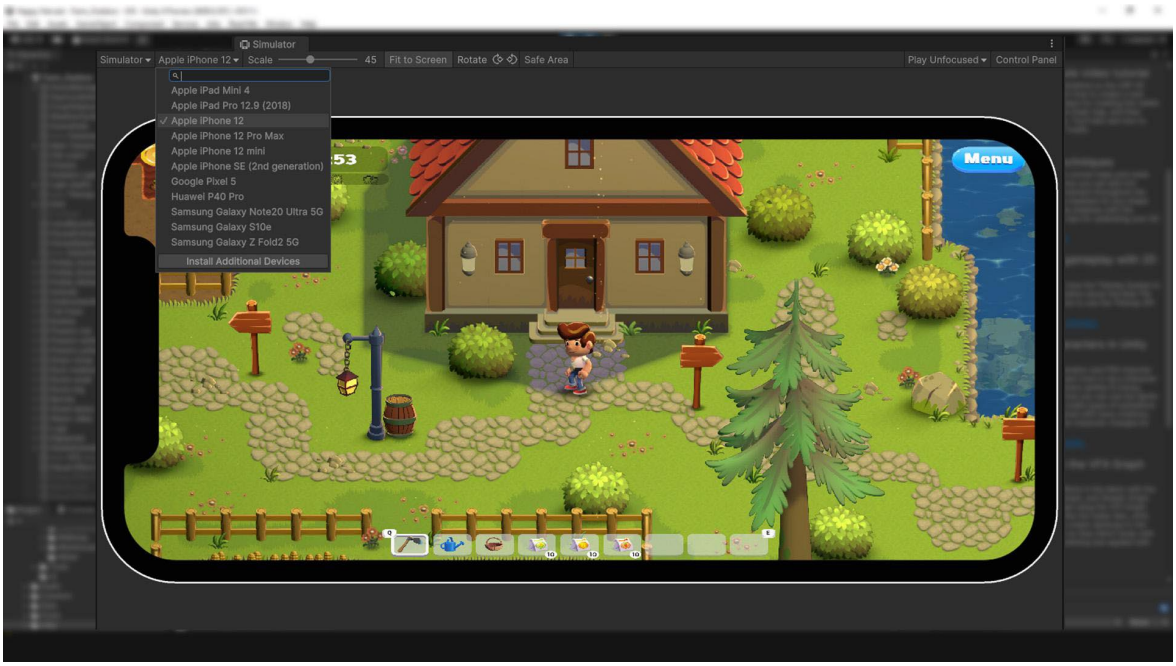
多数のオーバーレイ要素を避ける

UI 要素をたくさん重ねる（例えば、カードバトルゲームでカードを重ねる）と、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。

複数の解像度とアスペクト比を使用する

今日それぞれのモバイルデバイスがまったく異なる解像度やスクリーンサイズを扱っているのに対して、UI の代替バージョンを作成して各デバイスで最高の体験を提供します。

デバイスシミュレーターを使って、サポートされている幅広いデバイスにおいて UI をプレビューします。XCode および Android Studio で仮想デバイスを作成することもできます。



デバイスシミュレーターを使って多様なスクリーンフォーマットをプレビューする。

全画面の UI を使用する場合、その他のものをすべて非表示にする

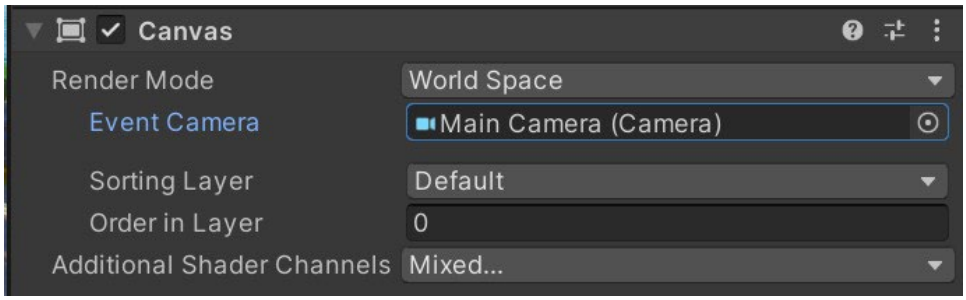
一時停止や開始の画面がシーンにある他のすべての要素を覆う場合は、その 3D シーンをレンダリングしているカメラを無効にします。同様に、一番上のキャンバスの後ろに隠れている、背景のキャンバス要素も無効にします。

全画面 UI の間は、60 FPS 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 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。

Canvas 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 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。

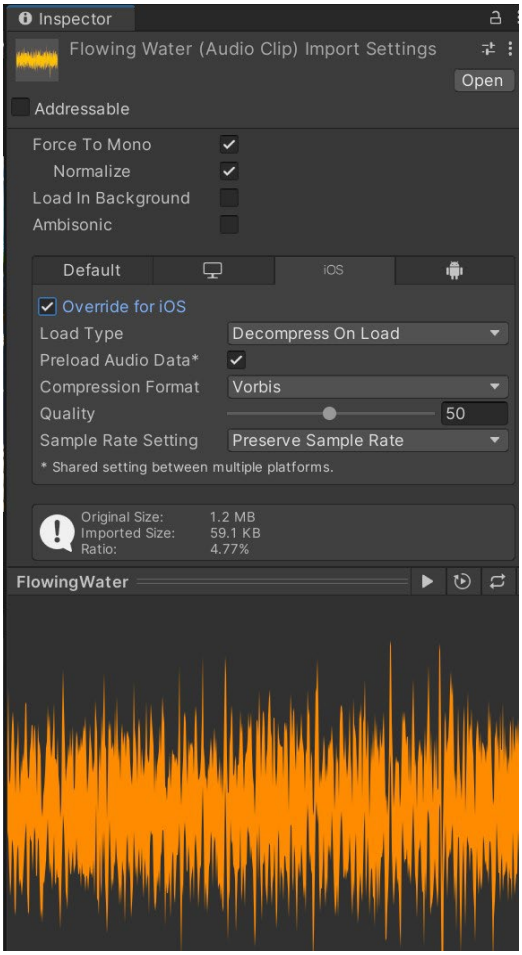
ターゲットプラットフォームでテストする

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 t 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、メモリを節約するために最適化することはできます。



オーディオクリップのインポート設定を最適化する。

可能な場合はサウンドクリップをモノラルにする

3D 空間オーディオを使用している場合、サウンドクリップをモノラル（シングルチャンネル）として作成するか、「Force To Mono」設定を有効にします。そうでない場合、ランタイム時に位置的に使用されるマルチチャンネルサウンドがモノラルソースに平坦化されて、これによって CPU の負荷が増加しメモリの無駄が発生します。

元の非圧縮 WAV ファイルをソースアセットとして使用する

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。

クリップを圧縮し、圧縮のビットレートを下げる

圧縮することによってクリップのサイズやメモリ使用量を削減してください。

- 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。これは非圧縮の PCM と比べてファイルが小さくなりますが、プレイバック中に素早くデコードされます。

モバイルデバイスのサウンドエフェクトは最大 22,050 Hz としてください。これよりも低い設定が通常最終品質に与える影響は軽微です。自身の耳が判断に役立つことでしょう。

適切なロードタイプを選択する

クリップサイズによって設定は変化します。詳しくは以下の表をご覧ください。

クリップサイズ	使用例	ロードタイプ設定
小（200 KB 未満）	騒々しい効果音（足音、銃声）、UI 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凍できます。
中（200 KB 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 Load Type 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 ～ 400 KB 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。

ミュートされた AudioSource をメモリからアンロードする

ミュートボタンを実装する場合、単純にボリュームを 0 に設定しないでください。AudioSource コンポーネントを破棄して、メモリからアンロードすることができます。ただし、プレイヤーがミュート状態を頻繁に切り替える必要がない場合に限ります。

Sample Rate Setting を使用する

Sample Rate Setting を Optimize Sample Rate または Override Sample Rate に設定します。

モバイルプラットフォームでは、22050 Hz 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。

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でのアニメーション制作の完全ガイド*」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。
- ジェネリックリグでは、ルートモーションを使うことは使わないことに比べて負荷が高いです。アニメーションがルートモーションを使わない場合、ルートボーンを指定しないでください。

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。

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

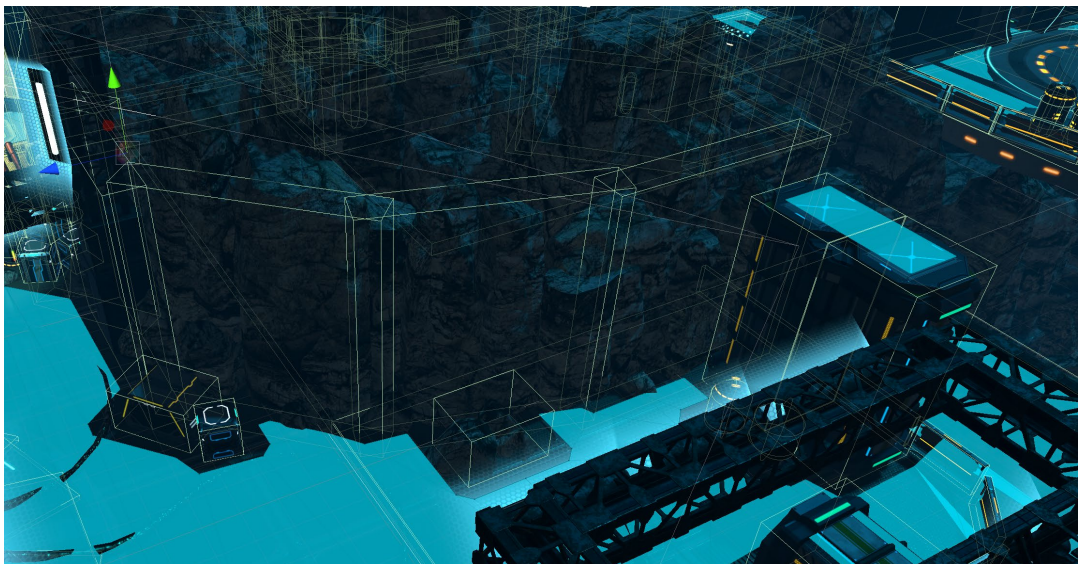
アニメーションのリギングを使用する場合は、各コンストレイントによって生じるパフォーマンスのオーバーヘッドに注意してください。ヒューマノイドモデルを扱う場合は、これを考慮することが重要です。可能な限り、パフォーマンスを向上させるために、ヒューマノイドリグのビルトイン IK（インバースキネマティクス）パスを活用してください。

物理演算

物理演算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、Unity の built-in physics（NVIDIA PhysX）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

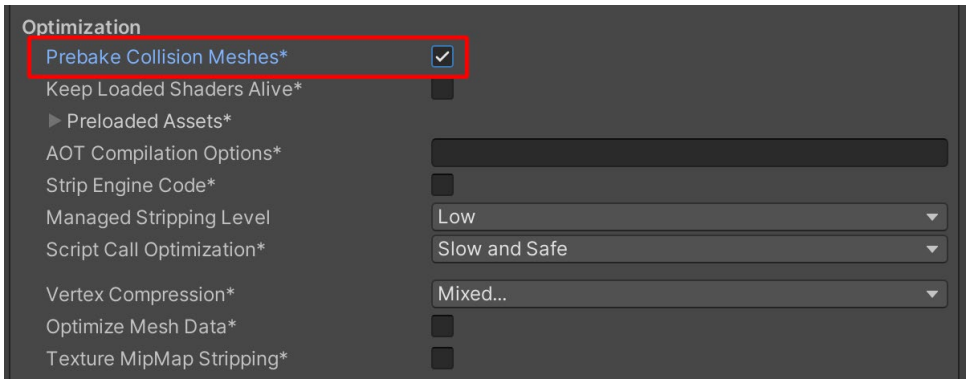
メッシュコライダーの負荷は高くなりえます。より複雑なメッシュコライダーをプリミティブまたは簡略化されたメッシュコライダーで代用し、元の形状に近づけてください。



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。

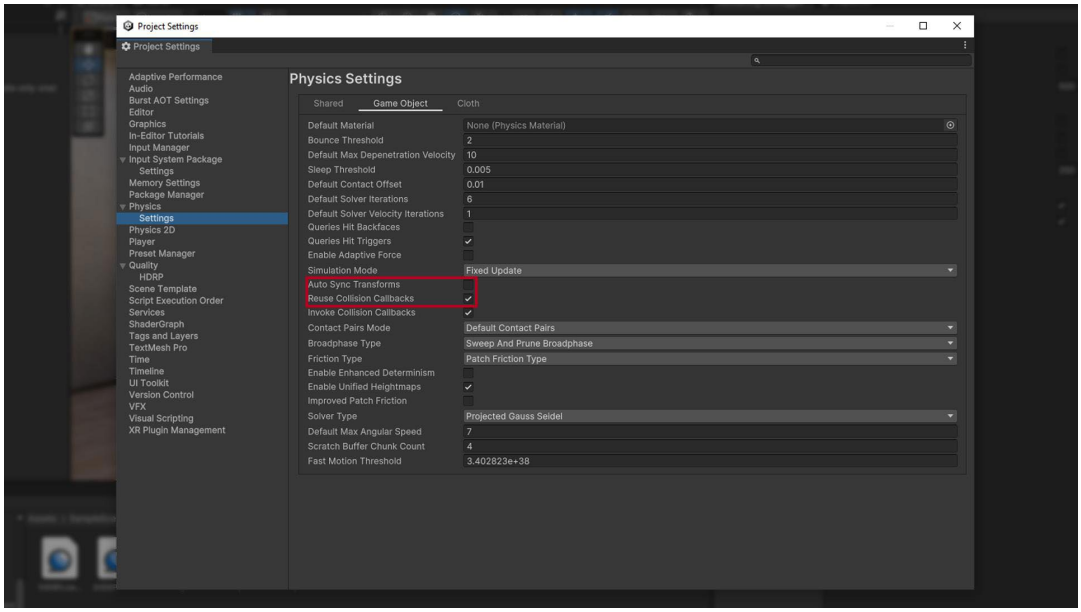
設定を最適化する

PlayerSettings 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 settings（**「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。

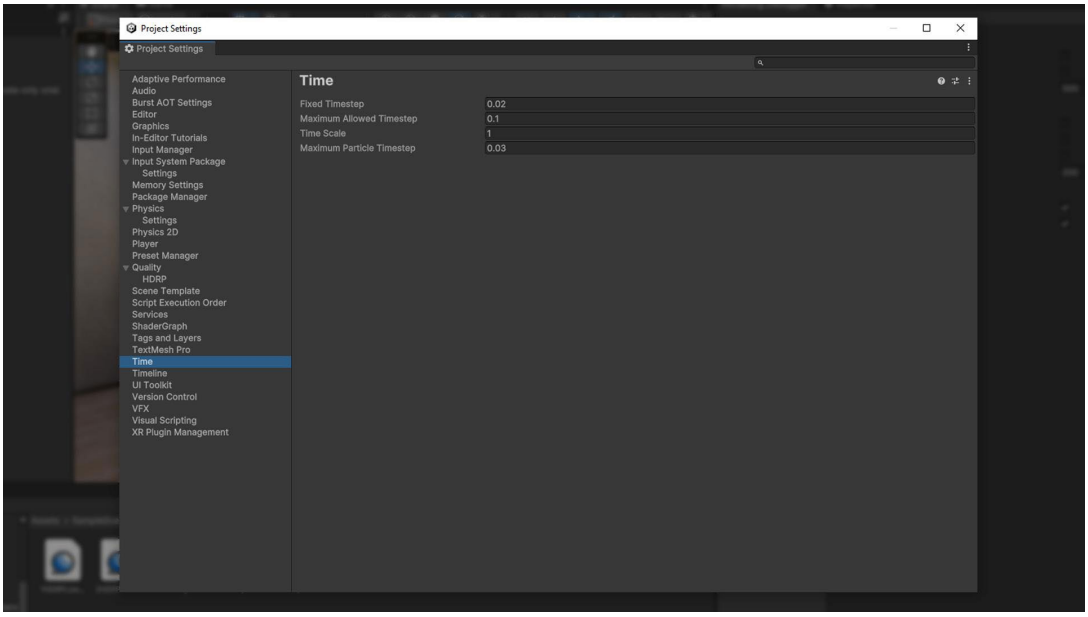


プロジェクトの物理演算設定を変更し、より高いパフォーマンスを引き出す。

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。プロジェクトが実行されている固定レートは、**「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（20 ミリ秒）は 50 FPS（50 Hz）に相当します。



Project Settings のデフォルトの Fixed Timestep は 0.02 秒（1 秒あたり 50 フレーム）。

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 40 ミリ秒以上かかることがあります。デフォルトの 20 ミリ秒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 つの物理演算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、これは性能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。

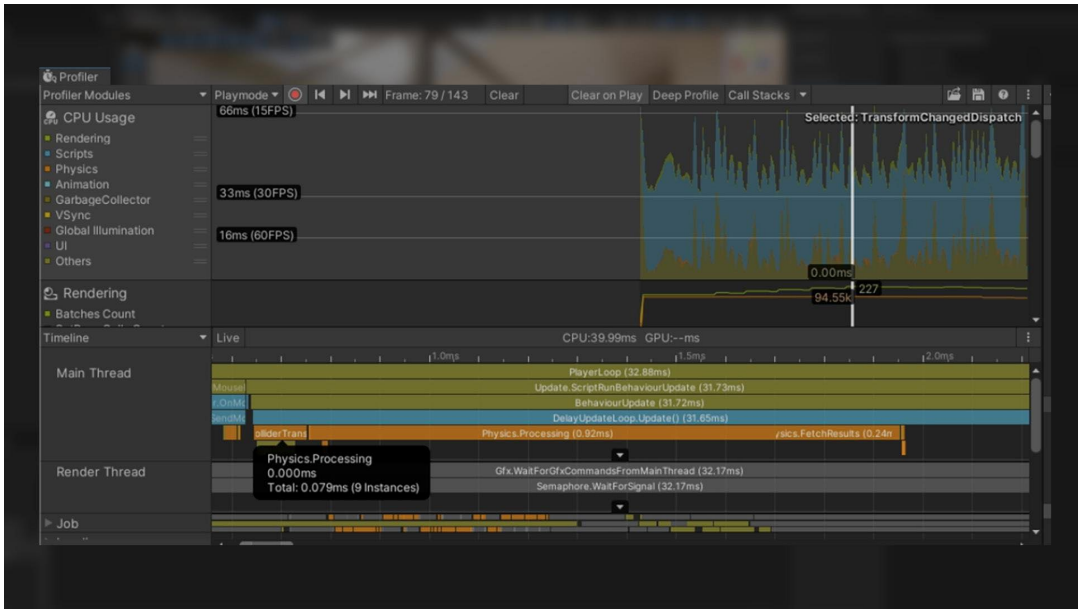
物理演算によるパフォーマンス問題を回避するには、以下を行ってください。

— シミュレーション頻度を下げる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 を下げるより小さな値（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に大きな変化があるシーンでは、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

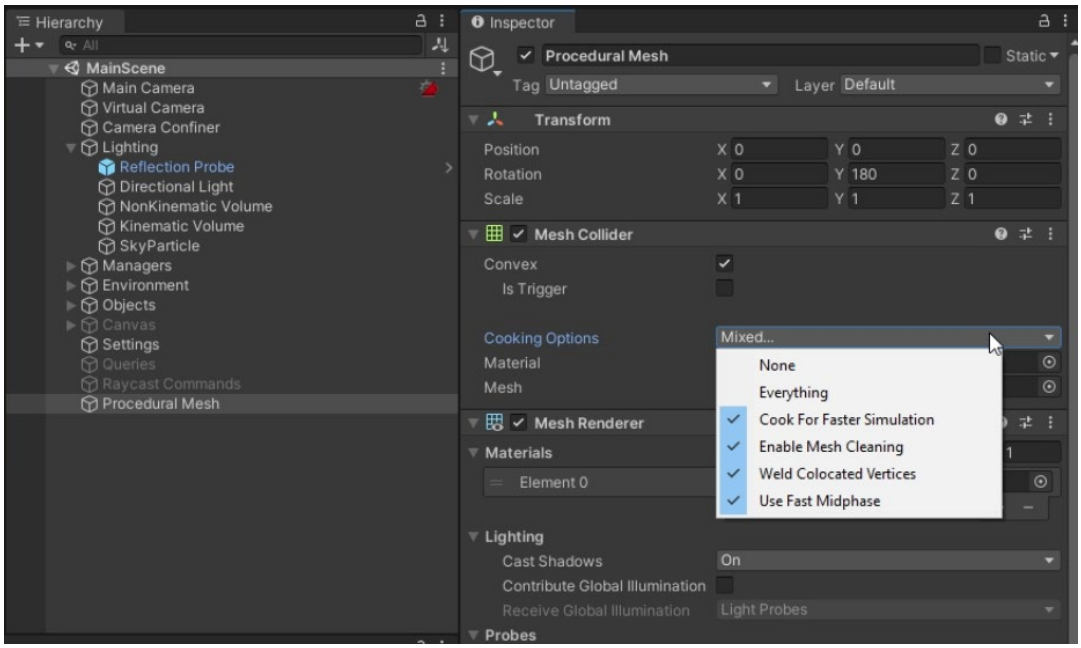
MeshCollider 向けに CookingOptions を変更する

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、物理演算のためのメッシュを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）。



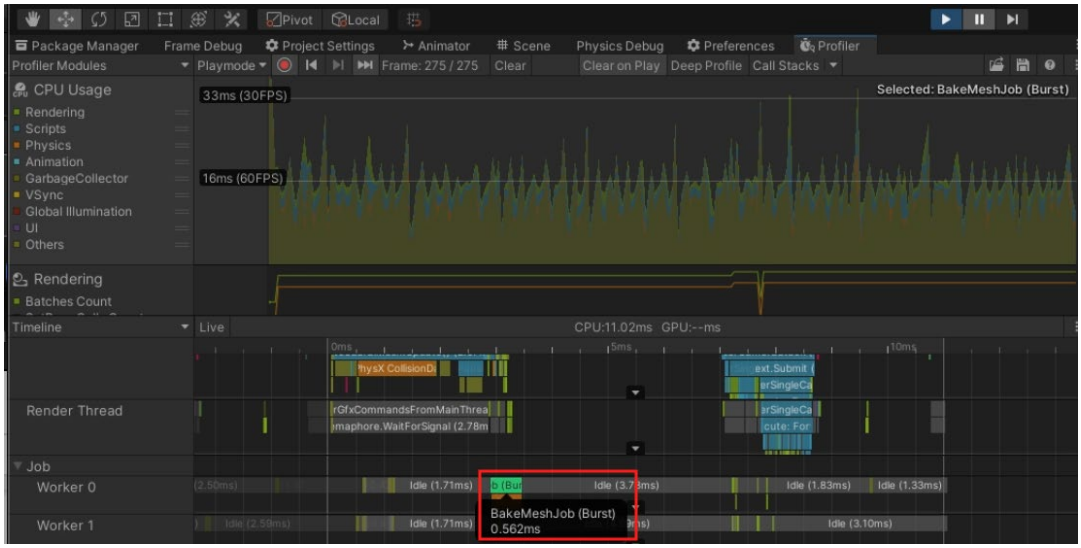
メッシュのクッキングオプション

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。これは後に、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

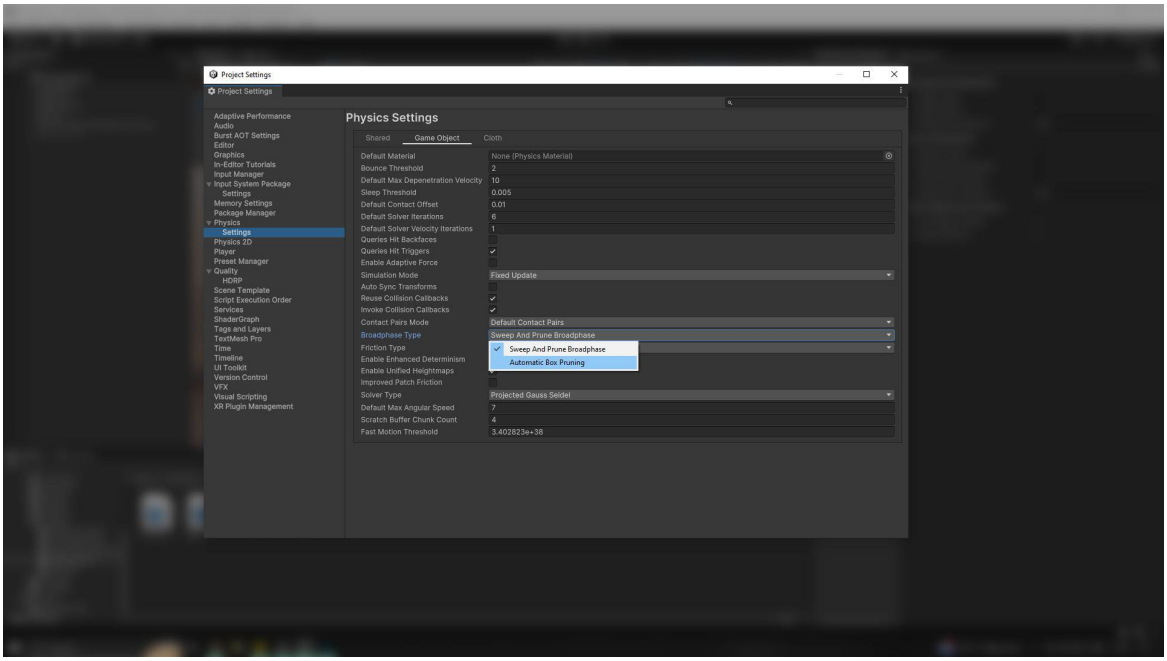
Unity の 物理演算エンジンは 2 つのステップで実行されます。

- **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。

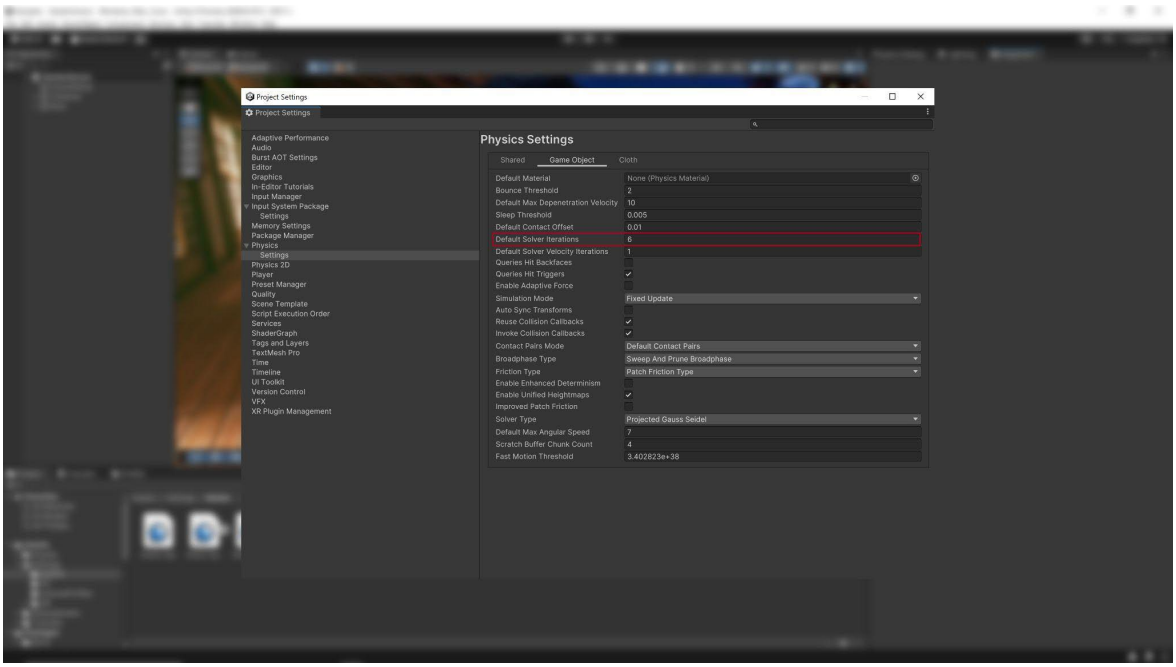
Multibox Pruning Broadphase では、ワールドの境界とグリッドセルの数を手動で指定できますが、Automatic Box Pruning ではそれを計算してくれます。



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

特定の物理演算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 が物理演算エンジンと自動的に同期されます。

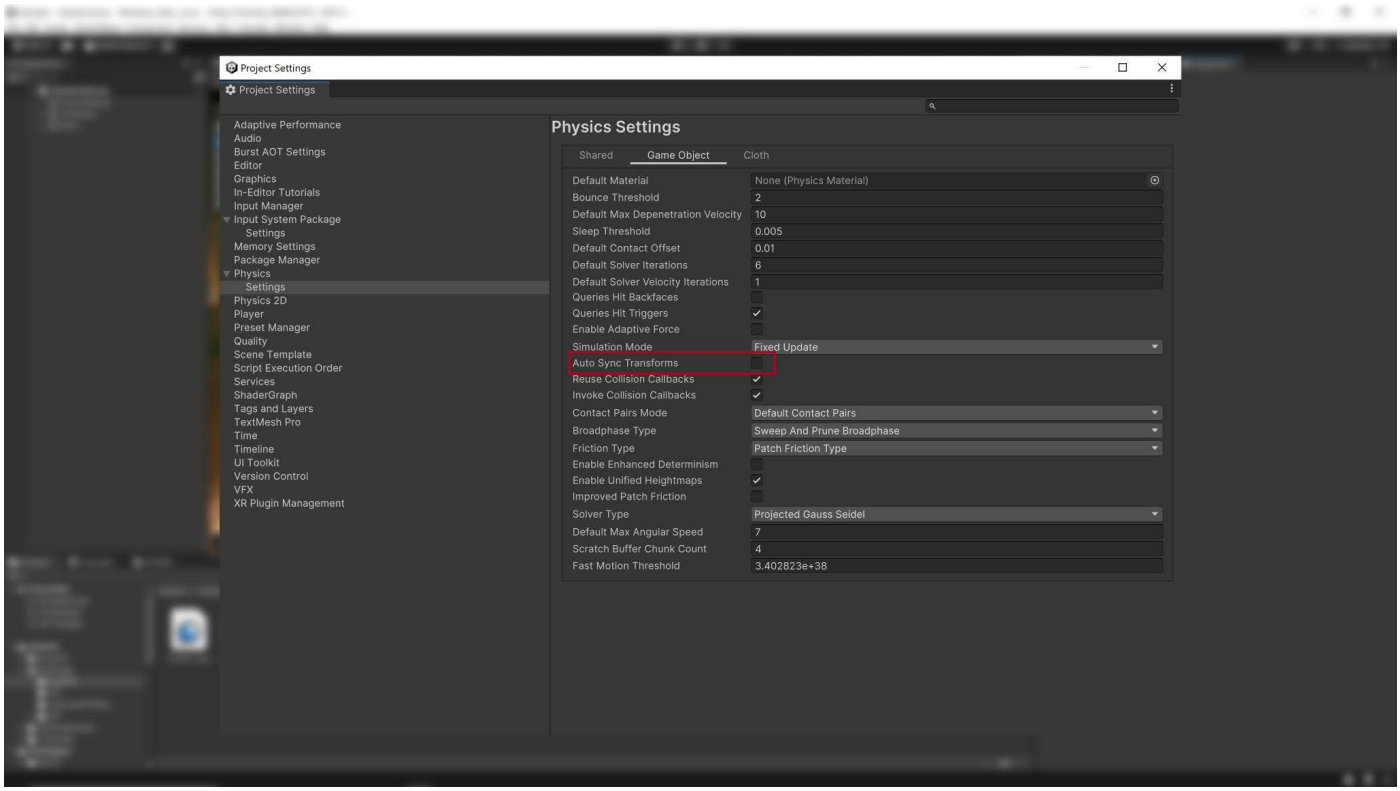
手動で同期すべきタイミング

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。

パ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。

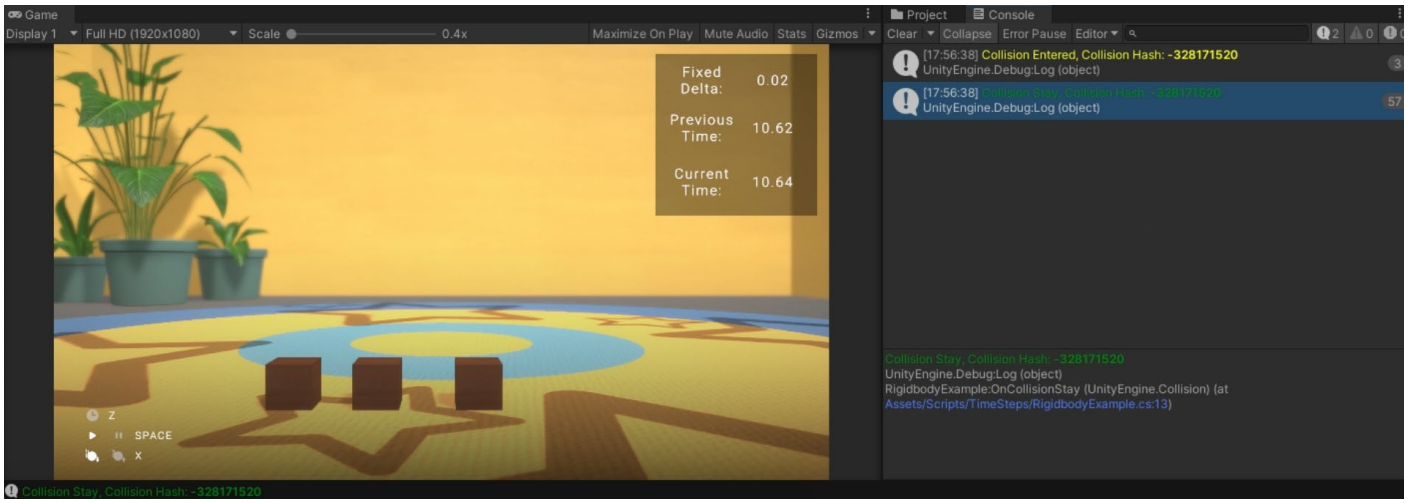
Reuse Collision Callbacks

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。

ガベージの発生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。これを行うには、物理演算ボディの位置を変更するだけです。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、再設定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れ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

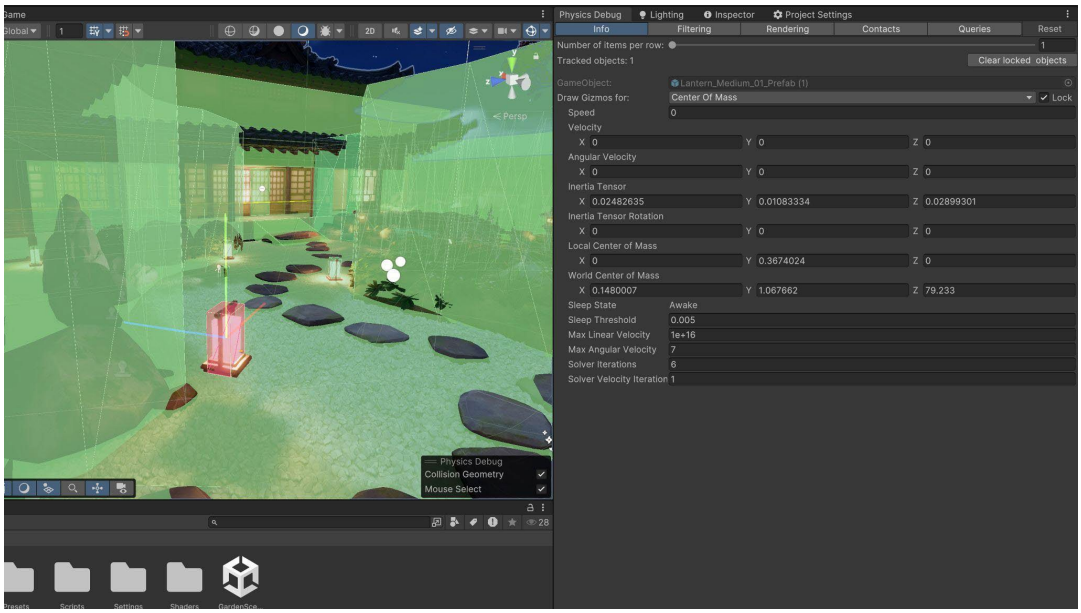
Physics.Raycast を用いてレイキャストクエリを実行できます。しかし、レイキャスト操作の数が多い場合（エージェント 10,000 体の視線の計算など）、CPU 時間が大幅に消費される可能性があります。

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。

ワークフローとコラボレーション

バージョン管理を使うべき理由

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。チームがコラボレーションできるようにプロジェクトが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することができるようになります。

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。

Unity Version Control

Unity Version Control（UVCS）は、プログラマーやアーティストをサポートするユニークなインターフェースを備えた柔軟な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。
- アセットを以前のイテレーションに戻す。
- 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。

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。

Version Control 設定で、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、**Mode** が **Visible Meta Files** に設定されていることを確認してください。

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 ドキュメントの スマートマージを参照してください。

Unity VCS、一般的なバージョン管理、プロジェクト編成のベストプラクティスについての詳細は、eBook 「*ゲーム開発者のためのバージョン管理とプロジェクト編成のベストプラクティス*」をご覧ください。

大規模なシーンを分割する

大規模な単一の Unity 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、レベルを複数の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** パラメーターモードを渡すことで、プロジェクトにシーンを加算的にロードできます。

使用されていないリソースを削除する

サードパーティ製のプラグインやライブラリとバンドルになった未使用のアセットに注意してください。その多くは埋め込まれたテストアセットやスクリプトを含み、それらを削除しないとビルドの一部になります。プロトタイピングから残っている不要なリソースを取り除きます。

この eBook に示されている最適化のためのすべてのヒントは、プラットフォームに関わらずゲームにとって有益となります。次のセクションでは、XR およびウェブの適性化のための具体的なヒントに注目します。

Unity Web ビルドのためのプラットフォーム特有のヒント

Unity Web ビルドの最適化は、特有の制限とウェブブラウザー内でアプリケーションを実行する上での課題があることより、固有のアプローチを要します。Web ビルドは、パフォーマンス、ロード時間、幅広いデバイスやブラウザに対する互換性のバランスを取る必要があります。このセクションでは、Unity Web プロジェクト向けに具体的に合わせた重要戦略やベストプラクティスを見ていきます。アセット管理やメモリの最適化から、ビルドサイズ削減やユーザー体験の向上まで、これらのヒントは高パフォーマンスのウェブアプリケーションの作成に役立ち、あらゆる異なる環境下でシームレスな体験を提供します。

Framerate

— Unity Web ビルドでは、**Application.targetFrameRate** をデフォルトのプロジェクト設定値 -1 にしておきます。targetFrameRate を -1 にしておくことによって、フレームレートをブラウザの「アニメーションレート」に関連付けるようにブラウザに指示します（すなわち、これによって可能な限り高速のレンダリングレートが提供されます）。少なくとも Firefox や Chrome においては、通常、アニメーションレートはディスプレイのネイティブフレッシュレートと同じです。しかしながら、Safari においては、フレッシュレートは常に 60 FPS に制限されています。

— プロジェクトでデフォルト設定を変更した場合、新しい C# スクリプトを作成することや、既存のスクリプトを開いてターゲットフレームレートを設定することもできます。通常、これはゲームを初期設定するスクリプトの中やセントラルゲーム管理スクリプトの中にあります。

```
using UnityEngine;

public class FrameRateManager : MonoBehaviour
{
    void Start()
    {
        // Set the target frame rate to -1 to let the browser control the frame rate
        Application.targetFrameRate = -1;
    }
}
```

Unity Web の設定を公開する

圧縮

圧縮することによって、ユーザーのブラウザーにダウンロードされる必要のあるファイルのサイズを大幅に削減できます。より小さなファイルは、より小さく速くダウンロードされて帯域幅の消費量も少なくて済みます。

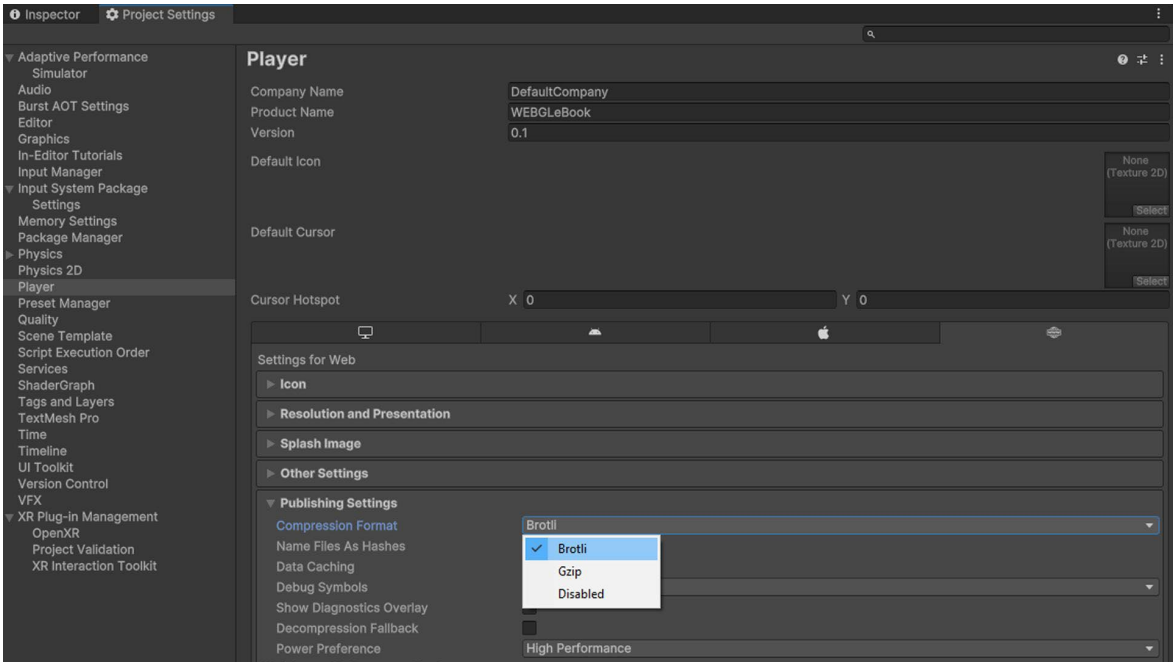
以下の圧縮方法が利用可能です。

Brotli：gzip に比べて高い圧縮率を提供し、ファイルサイズがより小さくなりロード時間が短縮されます。最新のブラウザにサポートされ、安全な https:// の URL または http://localhost/ のテスト URL からウェブサイトが提供されることを要求します。

Gzip：広くサポートされ、いまなお効果的な方法です。コンテンツが安全ではない http:// サーバー経由で提供される場合や、Brotli 圧縮コンテンツを提供するための設定が未整備のウェブサーバーにホストされている場合、または、Brotli との互換性がまだないさらに複雑な CDN ロードバランシングやキャッシュインフラストラクチャを使用している場合は、この選択肢にしてください。

非圧縮：ファイルサイズがかなり大きくロード時間が遅いので、一般的に製品に対しては推奨されません。ウェブサーバーの設定がオンザフライ圧縮キャッシュの使用のみになっており、Brotli や Gzip の圧縮済みコンテンツをサポートしていない場合でのみ、この設定で展開してください。

Unity Web ビルドを公開する場合、高い圧縮率、ブラウザのサポート性、パフォーマンスの点において、全般的に Brotli が最高の圧縮方法です。



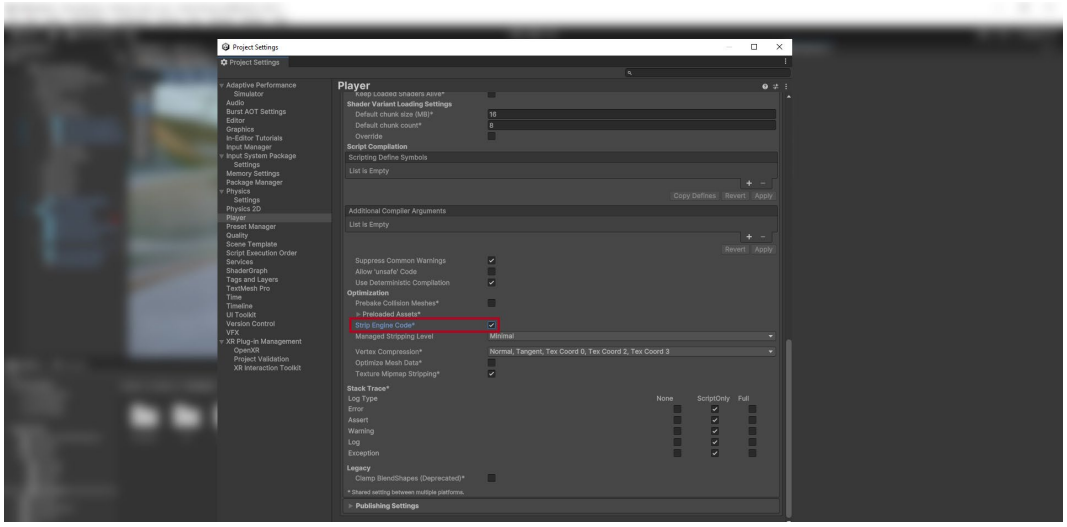
公開設定で Brotli を選択する

サイトの素早いスタートアップ時間を確保するために「**Decompression Fallback**」設定を **Disabled** にすることを推奨します。加えて、ページをホストするウェブサーバーは圧縮済みの Unity コンテンツを提供するように設定してください。

解凍フォールバックを有効にすると、モバイルブラウザーのバッテリー使用量に悪影響を及ぼし、ゲームのスタートアップ時間を遅くします。このドキュメントページにあるウェブサーバーの設定ガイドラインに従ってください。

Strip engine code

プレーヤー設定においてチェックするもう一つの設定は、**「Player settings」＞「Other Settings」**パネルにある **Strip Engine Code の有効化**です。これによって効率的なビルドを確保します。



Strip Engine Code を有効にする

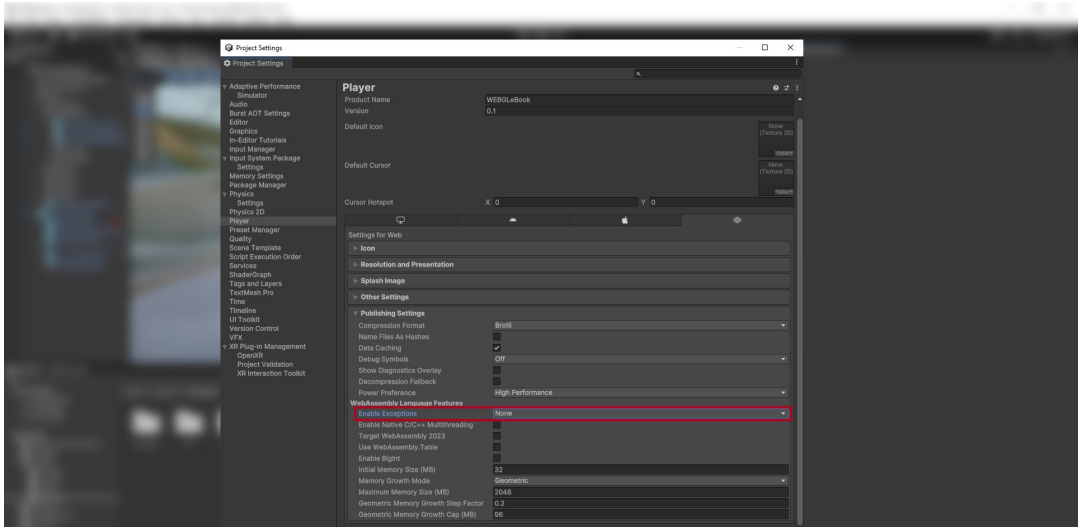
Enabling Strip Engine Code は、特に Unity Web プロジェクトにおいては効率的なビルドを確保するために推奨されるプラクティスです。この機能は未使用のエンジンコードを削除して、最終ビルドのサイズを大幅に縮小でき、さらに速いロード時間やより良いパフォーマンスにつながります。

Enable Exceptions 設定では「None」を選択する

例外のハンドリングには、ビルドに追加のコードを含むことが必要です。例外を無効にすることによって、ランタイムのチェックや例外をハンドリングするコードを削減します。もしプロジェクトが例外のハンドリングを制御フローとしない一方で、すべての例外を実行の終了のために扱えるならば、例外ハンドリングサポートを無効にすることはコードサイズの縮小につながります。

または、アプリケーションが例外ハンドリングサポートを不要とする場合、一般的に例外ハンドリングのコードサイズを最適化する WebAssembly 2023 を有効にすることを検討してください。

ビルドに例外が不要な場合は、Unity Web ビルドタブの下にある「**Player Settings**」ウィンドウの中で、「**Publishing Settings**」を広げて、「**Enable Exceptions**」を **None** に設定します。



Enable Exceptions を None に設定する

ターゲット WebAssembly 2023 機能セット

Firefox 89 以上、Chrome 91 以上、Safari 16.4 以上をゲームのターゲットとして利用できる場合、WebAssembly 2023 機能セットを有効にしてコードサイズを縮小させます。

WebAssembly 2023 は、JS BigInt、Bulk Memory Operations、Non-trapping float-to-int conversions、Sign-extension Operators、Fixed-width SIMD の各機能が利用可能です。各ブラウザーバージョンがサポートする WebAssembly の各機能を調べるには、WebAssembly Roadmap のページをご覧ください。https://webassembly.org/features/

Code Optimization 設定

最終リリースビルド向けに Project Build Platform Settings 中の Code Optimization を Disk Size with LTO に設定します（LTO ビルドの生成に長時間かかることがあります）。または、開発中の場合は、コードサイズが最適化されたビルド向けに Code Optimization を Disk Size に設定します。

Unity Web ビルドのプロファイリングを行う

Unity の一連のプロファイリングツールの使用に加えて、Chrome DevTools や Firefox Profiler のようなツールを活用してパフォーマンスのプロファイリングと分析ができます。

Chrome DevTools

Chrome DevTools は、Google Chrome ブラウザーに組み込まれた包括的なウェブ開発ツールのセットです。パフォーマンスのプロファイリング、JavaScript のデバッグ、ネットワークアクティビティの分析、レンダリングの検査をする機能を提供します。Chrome DevTools を有効化する基本的なステップは以下の通りです。

1. Chrome DevTools を開くためには F12、または **Ctrl+Shift+I**（Windows/Linux）、または **Cmd+Option+I**（Mac）を押します。ページ上で右クリックして「**Inspect**」を選択しても開けます。
2. 「**Performance**」タブへ進みます。「**Record**」ボタンをクリックし、Unity Web ゲームと相互作用してパフォーマンスのデータを記録します。「**Stop**」をクリックして記録を終了し、記録したデータを分析します。フレームレート、CPU 使用量、レンダリングパフォーマンスに注目します。
3. 「**Network**」タブへ移動して Unity Web ゲームをリロードし、すべてのネットワークリクエストを記録します。タイムライン、リクエストの詳細、ロード時間を調査し、ネットワーク関連のパフォーマンスのボトルネックを特定します。
4. 「**Sources**」タブを使って、JavaScript のコードにブレークポイントを設定し、実行を一時停止して変数を確認します。コールスタックやスコープ情報を使って、コードのデバッグや最適化を行います。
5. 「**Console**」タブを使って、Unity Web の状態を記録し、レンダリングの課題をデバッグします。Unity Web 特有のツールおよび、Unity Web Insight や Unity Web Debugging などの拡張機能を活用して、より深く分析します。

XR 最適化のヒント

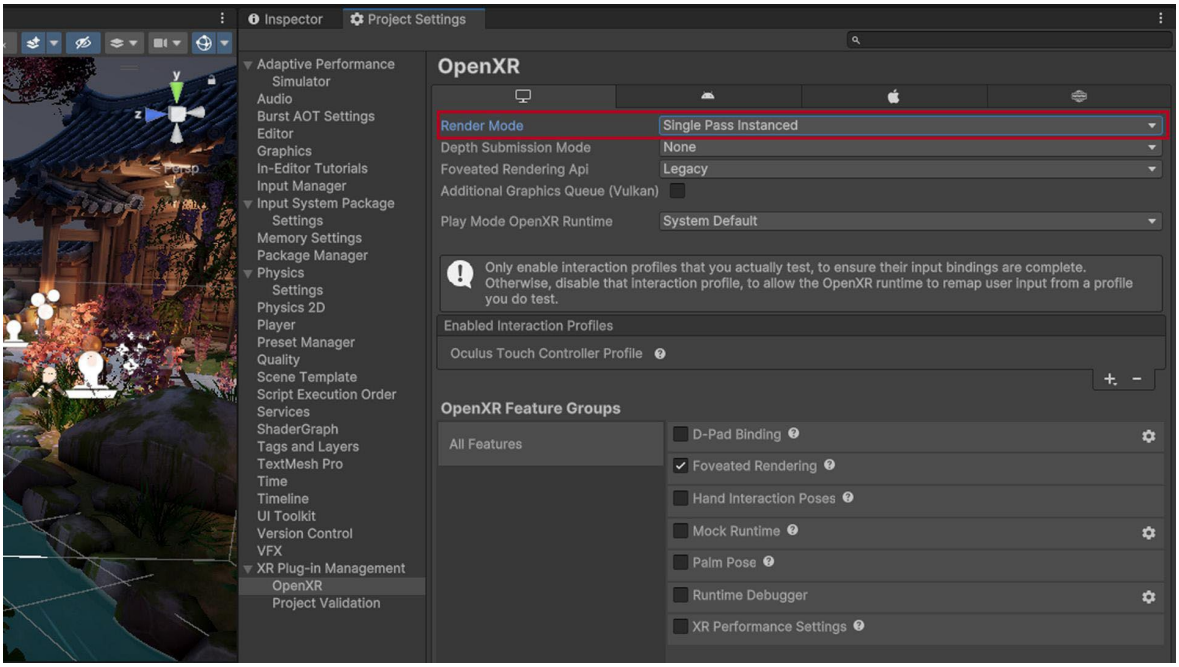
このセクションは、Unity を使って構築した VR、AR、MR（総称 XR ）のアプリケーションのための最適化のヒントを取り扱っています。これらの技術の多くについては、一般的にモバイルデバイスに適用されるため、このガイドの他の部分に記載されています。しかしながら、XR のアプリケーションに限定して着目する読者に向けて、まとめてここにも記載しました。

これらの技術を試して XR、特に VR のアプリケーションの効率的な実行に役立ててください。なぜなら XR の体験は、没入感を維持してモーションシックネスを防ぐために、高いパフォーマンスと短い待ち時間が要求されるからです。高解像度、3D レンダリング環境、応答性の高い相互作用を得るためには、スムーズで物理的にも快適な体験を確保するために最適化を必要とします。

Unity の XR アプリケーション開発に関する包括的なガイドについては、eBook「*Unity で仮想現実および総合現実の体験を創造する*」をダウンロードしてください。

Render Mode

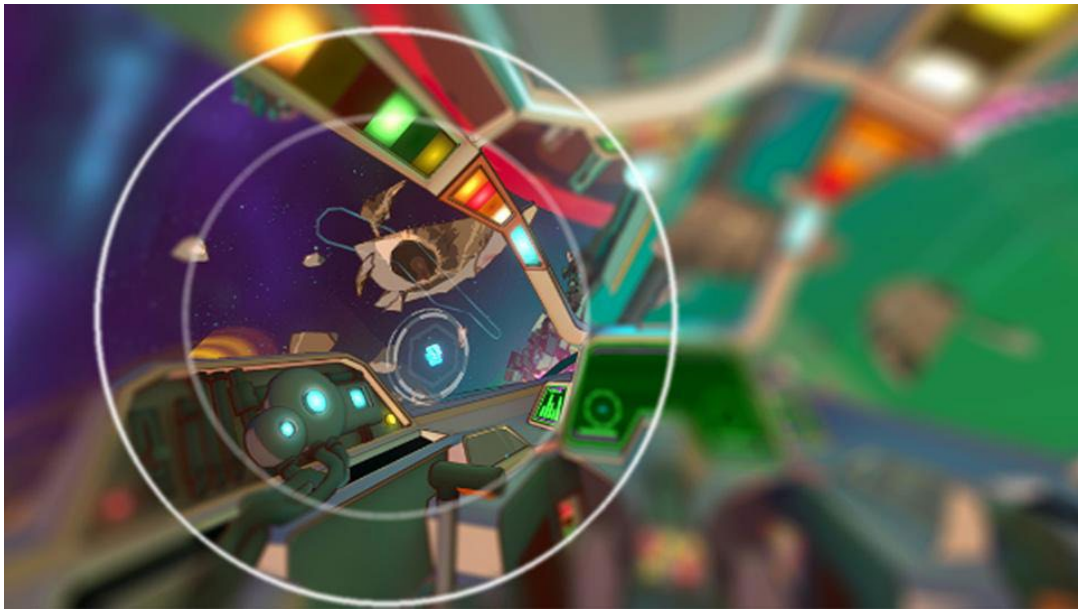
正しい Render Mode 設定は、VR ゲームのパフォーマンスに大きな差をもたらします。Unity の OpenXR plugin を使用している場合、「**Project Settings**」＞「**XR Plugin-management**」＞「**plugin provider**」の中に「**Render Mode**」のメニューオプションがあります。ドロップダウンから「**Single Pass Instanced**」を選択してください。このモードは、両眼のシングルパスインスタンスのレンダリングを行います。シーンはレンダリングが 1 回行われ、シェーダーは両眼同時に実行されます。



XR アプリケーションを開発する場合は、レンダリングモードとして「Single Pass Instanced」を選択する。

中心窩レンダリング

Unity 6 は、中心窩レンダリングを統合して、Oculus XR および OpenXR をサポートし、PlayStation VR2 のサポートも含みます。中心窩レンダリングは VR 向けの最適化技術であり、人間の眼が一度で焦点を合わせられるのは小範囲である性質を活用しています。焦点エリアを高解像度、周囲を低解像度でレンダリングすることによって、GPU の作業負荷を大幅に削減します。中心窩レンダリングを実装することによってパフォーマンスが向上し、より高いフレームレートが得られます。最も重要なのは、ビジュアル品質が改善されることです。



高解像度によって焦点エリアを示す中心窩レンダリング

XR Interaction Toolkit を活用する

Unity の XR Interaction Toolkit は、XR プロジェクトの最適化された入力をハンドリングするための素晴らしい選択肢です。VR や AR のハードウェアとともに効率的に動作するよう設計された事前に構築されたコンポーネントおよびインタラクションシステムのセットを提供します。開発者はこのツールキットを活用して以下のことができます。

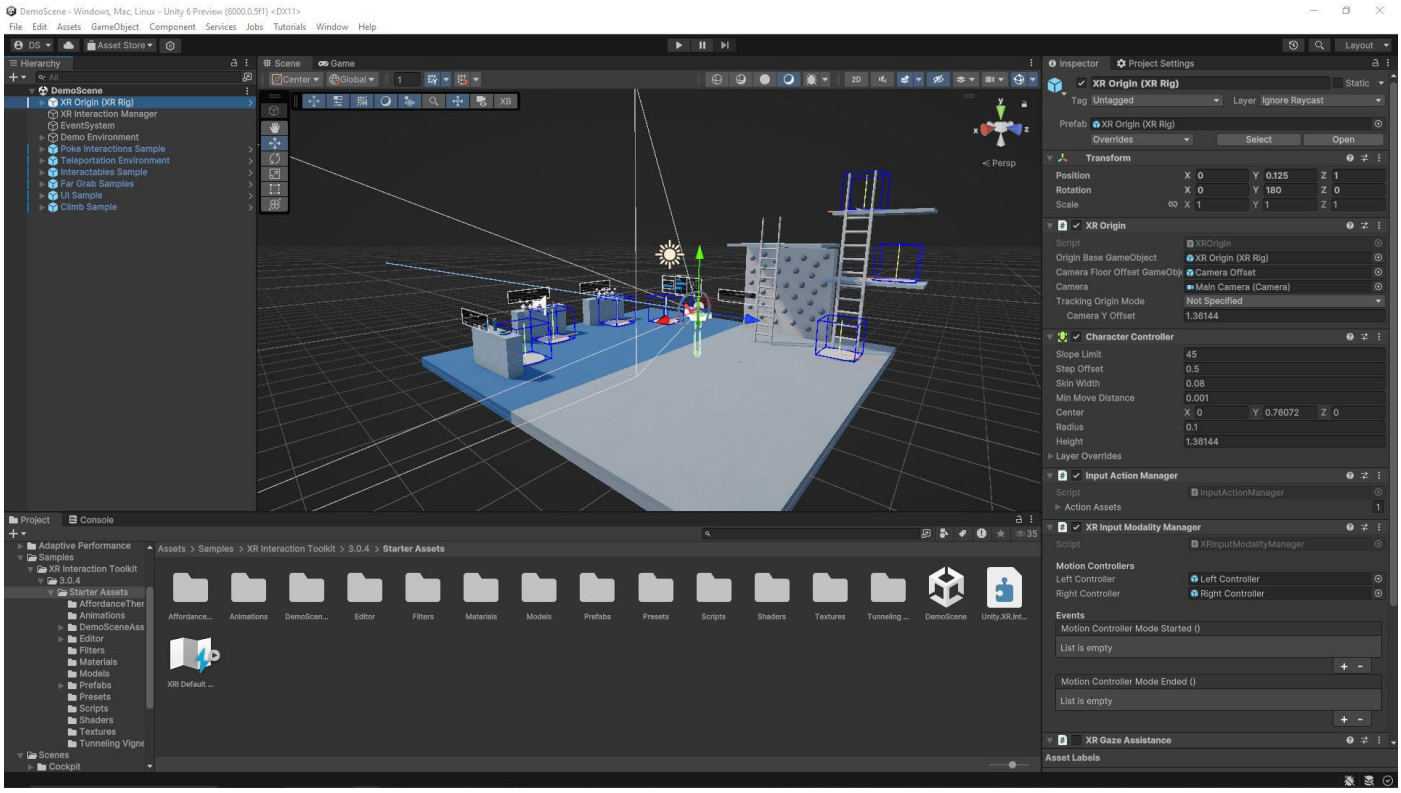
インタラクションの標準化：ビルトインインタラクションパターンを使ってカスタムコードを減らし一貫性を確保します。

イベントドリブンアーキテクチャの使用：イベントドリブンの入力ハンドリングを活用してポーリングを最小化しパフォーマンスを向上させます。

使いやすさの向上：すぐに使えるコンポーネントによって開発プロセスをシンプルにして、より速くイテレーションや最適化を行います。

全体として、XR Interaction Toolkit は入力ハンドリングの合理化と最適化に役立ち、XR アプリケーションにおける応答性やユーザー体験を向上させます。

これらの戦略を実行することによって、スムーズで応答性の高い入力ハンドリングを確保でき、XR アプリケーションにおける全体的なユーザー体験が向上します。



XR Interaction Toolkit のサンプルシーン

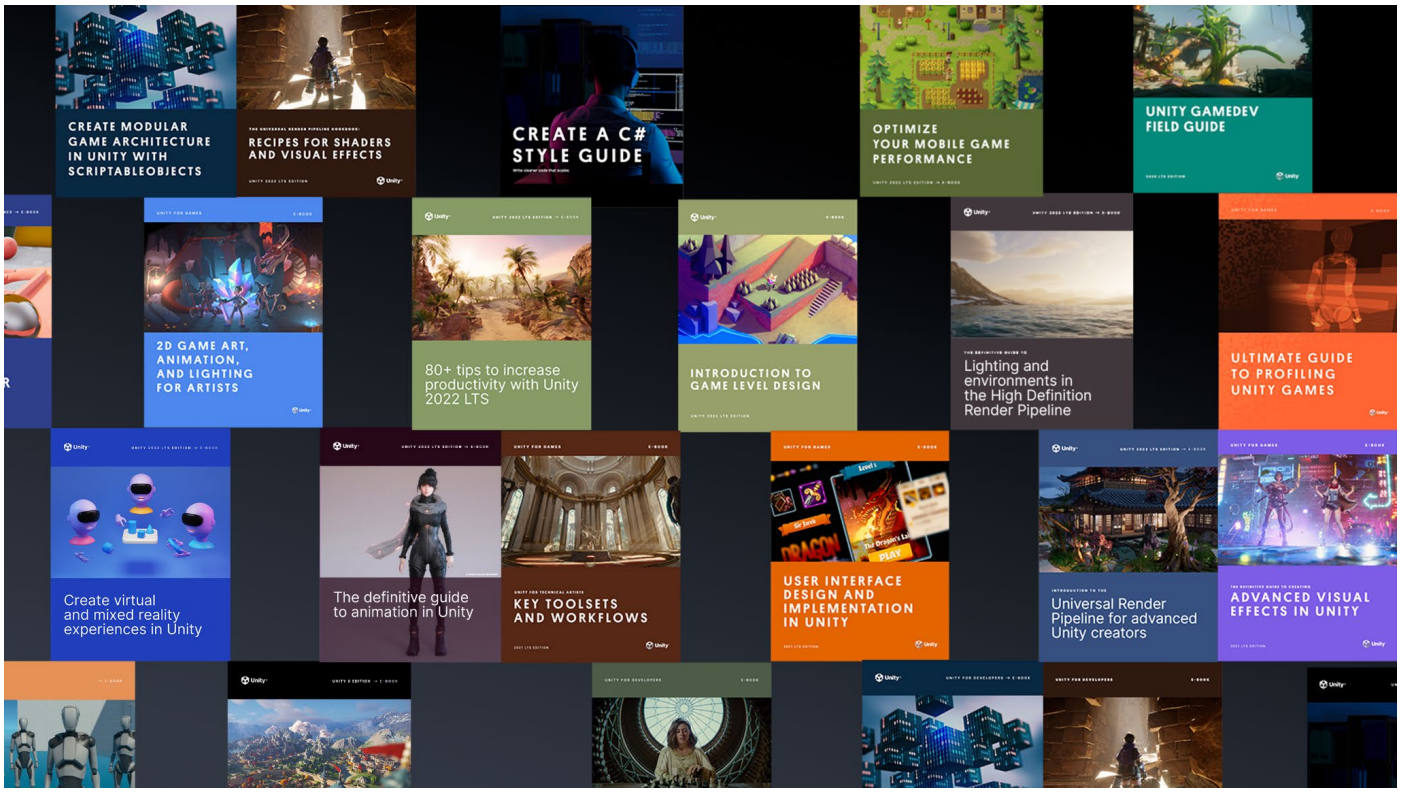
XR 最適化のためのパフォーマンステスト

パフォーマンステストは、XR アプリケーションのスムーズで没入感のある体験の提供を保証するためには不可欠です。

XR 固有のプロファイラーは以下の通りです。

- **Oculus Performance Head-Up Display（HUD）**：リアルタイムのパフォーマンスメトリクス
- **SteamVR Performance Tool**：VR アプリケーションパフォーマンスを分析

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®