unity

ベストプラクティスガイド

ゲーム開発を最適化する 9 つの方法

Unity エキスパートによるリリースに向けてのヒント

Explorer: 3D Game Kit — Made with Unity

はじめに

生み出されるゲームのほぼすべてには、創造的なビジョンと、できる限り質の高いゲームをリリースしたいというスタジオの思いが込められています。ゲーム制作は素晴らしい、非常にやりがいのある仕事です。しかし、パフォーマンスの高いゲームを完成させてリリースするまでの道のりには無数の課題があります。

このようなことを念頭に、Unity の Integrated Success Services（ISS）エキスパートがこのガイドを制作しました。9 つの重要な開発領域を中心に構成されており、一般的なメモリ、パフォーマンス、およびプラットフォームの問題をより深く理解し、それを*回避*するために役立つでしょう。

Integrated Success Services について

ゲームスタジオは Unity の ISS プログラムを利用することで、専任のデベロッパーリレーションエンジニアやその他の専門家の協力を得て、プロジェクトを最適化して円滑に進めることができます。Unity の ISS エキスパートの豊富な経験と洞察は、スタジオの生産性の向上、タイトなスケジュールへの対応、新しい領域への拡大に役立ちます。

このガイドで具体的に解説されている ISS エキスパートの戦略的分析と最適化に関する推奨事項を参考にすることで、リリースの遅延につながる問題を*発生前*に防止し、リリース時には可能な限り最高のエクスペリエンスをプレーヤーに提供することができます。これらのベストプラクティスが皆さまのお役に立てば幸いです。

提供されているスクリーンショットは Unity のさまざまなバージョンから取得されており、ご使用の Unity バージョンとは異なる可能性があることにご注意ください。

1. 計画

機能要件とターゲットプラットフォームを調査する

プロジェクトを開始したり、プロジェクトで大量の処理を行ったりする前に、機能要件とターゲットプラットフォームを徹底的に調査してください。意図したすべてのプラットフォームが実際に必要な機能をサポートしていることを確認します（たとえば、低レベルのモバイルデバイスではインスタンストレンダリングはサポートされていません）。制限および考えられる回避策や妥協策についても必ず検討してください。

また、各ターゲットプラットフォームの最低スペックを特定し、開発チームと QA チームの両方に複数のハードウェアユニットを調達します。これは、開発中に幅広いターゲットハードウェアをテストする必要があるためです。これにより、現実的なパフォーマンスバジェットとフレームバジェットをすばやく測定して調整し、開発全体を通してこれらを監視できるようになります。

メモリバジェットとパフォーマンスバジェットを定義する

ターゲットの仕様と機能のサポートを明確にしたら、メモリバジェットとパフォーマンスバジェットを定義します。これを適切に定義することは難しく、開発中に修正や調整が必要になることもあります。しかし、計画を立てずに単に何でもプロジェクトに組み込むより妥当な計画から始めた方がはるかに効果的です。

まずは、ターゲットフレームレートと理想の CPU パフォーマンスバジェットを決定します。モバイルプラットフォームの場合、サーマルスロットリングが作動して CPU と GPU のクロック速度がどちらも低下する可能性があるため、計画ではそのオーバーヘッドを忘れずに考慮してください。

CPU バジェットから、レンダリング、エフェクト、コアロジックなど、必要な各種のシステムにどれだけの時間を費やすかを決定してみましょう。

メモリバジェットの決定はかなり難しい場合があります。メモリを最も消費するのはアセットであり、これは開発者がコントロールできるメインの領域です。たとえば、全体でどのくらいの量のメモリをアセットに割り当てるかを決定します。テクスチャ、メッシュ、サウンドはいずれも大量にメモリを消費するため、注意しないと制御不能に陥る可能性があります。過剰に大きいテクスチャは避け、画面に表示されるサイズとターゲットプラットフォームの画面解像度に適したサイズを維持します。同様に、メッシュの頂点とポリゴンの数もユースケースに適したものにする必要があります。遠方にのみ表示されるオブジェクトに高精細なメッシュは不要です。

最後に、プロジェクト内のシステムにどれだけのメモリを割り当てられるかを考えてください。たとえば、更新ごとの CPU 計算の量を減らすために、大量のデータを事前計算するシステムを実装するのは理にかなっているでしょうか。そのシステムが不釣り合いに大量のメモリを消費する可能性もあります。パフォーマンスの観点で最も重要なシステムとなり、このトレードオフにそれだけの価値がある場合もあります。こうした問題に対処できるよう計画する必要があります。

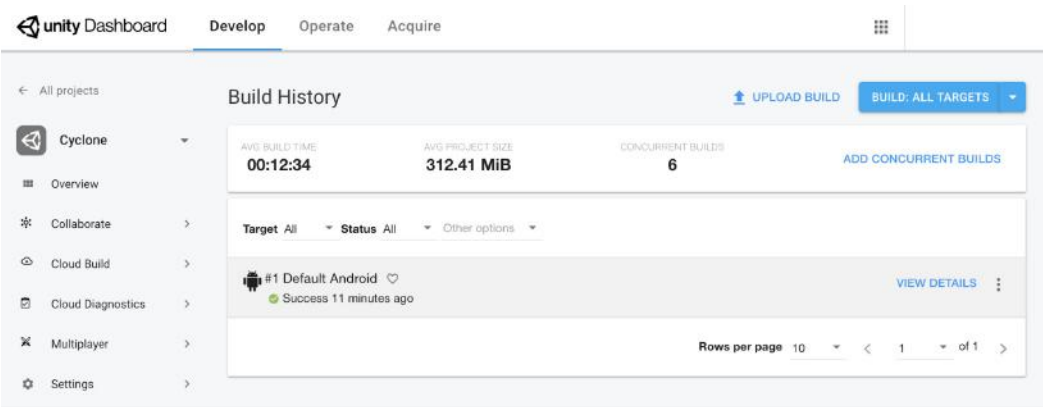
ビルドと QA のプロセスを組み込む

ビルドと QA のプロセスを組み込むことはとても大切です。ローカルでの構築とテストは、ある時点までは有用であるものの、時間がかかり、エラーが発生しがちです。候補となるソリューションは多数あります（たとえば、Jenkins は非常に広く使用されています）。ビルド専用のマシンを独自に構成することもできますが、クラウドベースのサービスが多数あり、そのいずれかを使用すれば独自のマシンを維持するためのオーバーヘッドやコストを削減できます。Unity Teams の Cloud Build 機能も検討してください。時間をかけてソリューションを評価し、ニーズに合うものを選びましょう。

リリースビルドに機能を公開する方法を計画するとよいでしょう。バージョン管理と連携し、開発ブランチをどのようにビルドおよび検証するかを検討します。ビルドプロセスの一部として実行される自動テストでは、すべてではないものの多くの問題を検出できます。自動テストを実行する場合は、複数ビルド間でのテストパフォーマンスの履歴を取得できるように、メトリックを収集すると役に立ちます。これにより、リグレッションをより迅速に見つけることができます。また、場合によってはプロセスに手動の品質保証と承認を含めることが必要になります。ビルドプロセスでは、ブランチが検証された後でそれをリリースビルドにマージできます。

ゼロからの制作開始を検討する

プロジェクトのプロトタイプを作成した後、制作フェーズをゼロから開始することを強くお勧めします。通常、プロトタイプの作成中に行われた決定はスピードを優先しており、プロトタイププロジェクトは多くの「ハック」で構成されている可能性が高く、制作の開始に使用できるしっかりとした基盤ではありません。



Unity Cloud Build を使用してプロジェクトの自動ビルドを作成

2. 開発とワークフロー

ほとんどの開発者は、ワークフローの問題と常に戦うより、創造的かつ生産的な作業に時間を使いたいと考えています。しかし、開発中に適切でない選択をすると、非効率やチームのミスにつながり、最終製品の品質にまで影響が及ぶ可能性があります。このような事態を避けるために、一般的なタスクを合理化し、チームの障壁となる可能性のある誤りを最小限に抑えることができる手法を採用しましょう。

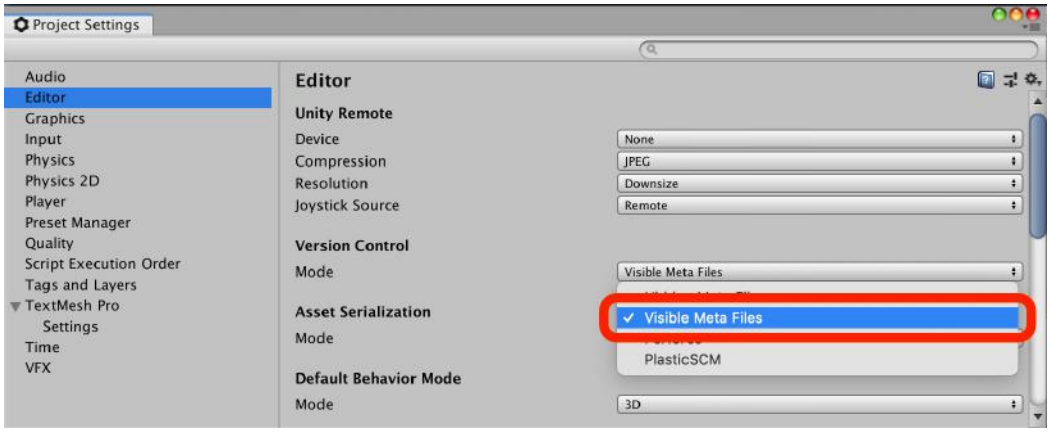
繰り返しの手動タスクを自動化する

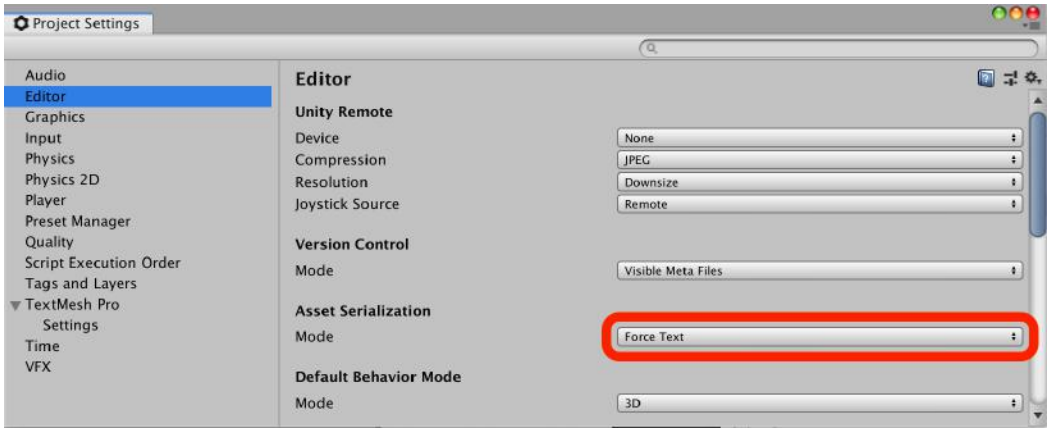
頻繁に繰り返される手動タスクは、（たとえば、カスタムスクリプトやエディターツールによる）自動化の最有力候補です。自動化にかける時間と労力が比較的少なくても、プロジェクトの期間中、チーム全体では大きな時間の節約につながる可能性があります。タスクの自動化によってユーザーエラーのリスクも排除されます。

特に、ビルドプロセスを完全に自動化し、ローカルでも継続的インテグレーションサーバーでも 1 回のアクションで完全にビルドできるようにしてください。

バージョン管理を実装する

すべての開発者が何らかのバージョン管理を使用する必要があります。Unity では複数のソリューションがビルトインサポートされています。





Unity Editor 内でのバージョン管理設定

まず、プロジェクトの「Editor Settings」で、「Asset Serialization」の「Mode」がデフォルトの「Force Text」に設定されていることを確認します。

Unity には、シーンとプレハブをマージするための YAML（人間が読めるデータシリアル化言語）ツールも組み込まれています。このツールが設定されていることも確認してください。詳細については、「SmartMerge」を参照してください。

バージョン管理ソリューションがコミットフック（Git など）をサポートしている場合は、そのコミットフックを利用して特定の標準を適用できます。詳細については、この「Unity Git Hooks」のページを参照してください。

すべてのアクティブな開発作業をメインブランチとは切り離し、常にプロジェクトの確実な動作バージョンを維持できるようにします。ブランチとタグを使用して、マイルストーンとリリースを管理します。

チーム内に適切なコミットメッセージのポリシーを適用します。明確でわかりやすいメッセージは、開発の後工程で問題を追跡するのに役立ちます。空のメッセージや意味のないメッセージは混乱を招きます。

最後に、優れたバージョン管理は、問題が発生した場所をすばやく特定するのに役立ちます。たとえば Git には、既知の問題がないリビジョンと既知の問題があるリビジョンを分けてマークできる機能があるため、「分割統治」のアプローチを使用して、中間のリビジョンをチェックアウトしてテストし、既知の問題の有無を示すフラグを立てることができます。バージョン管理システムにどのような機能があるかを調べ、開発中に役立つ可能性のある機能を特定しておきましょう。

キャッシュサーバーを利用する

Unity Editor 内でのターゲットプラットフォームの切り替えは、特に大規模なプロジェクトでは、非常に時間がかかる可能性があります。このため、Unity キャッシュサーバーを利用することをお勧めします。Unity の複数のプロジェクトまたはバージョンをサポートする必要がある場合は、オプションとして異なるポートを使用し、複数のキャッシュサーバーを実行できます。

キャッシュサーバーを使用すると、ローカルユーザーのプラットフォームの切り替えもずっと高速になるため、必ず利用しましょう。

プラグインに注意する

多くのプラグインにはテストアセットやスクリプトが組み込まれています。このため、プロジェクトに多数のプラグインとサードパーティライブラリが含まれている場合には、未使用のアセットがゲームに組み込まれている可能性があります。

アセットストアのアセットを使用している場合、そのアセットによってプロジェクトに発生する依存関係を確認してください。たとえば、複数の異なる JSON ライブラリを発見して驚くかもしれません。

最後に、プロトタイプ作成フェーズから残っている可能性のある古いアセットやスクリプトなど、プラグインの不要なリソースをすべて取り除きます。

シーンやプレハブでのコラボレーションに注目する

開発チームがコンテンツに関してどのように共同作業するかを考えることが重要です。大きな単一の Unity シーンは、コラボレーションには適していません。レベルを多数の小さなシーンに分割することをお勧めします。そうすれば、競合のリスクを最小限に抑えながら、アーティストとデザイナーが 1 つのレベルでよりうまくコラボレーションできるようになります。ランタイムで、SceneManager.LoadSceneAsync() API を使用してパラメーターモード LoadSceneMode.Additive を渡すことで、シーンを追加的にロードできます。

Unity 2018.3 以降には、プレハブのネストをサポートする改良版プレハブが含まれています。これにより、プレハブコンテンツをより小さな個別のアイテムに分割でき、さまざまなチームメンバーが競合のリスクなしに個別に作業できるようになります。それでも、チームメンバーが同じアセットにアクセスして作業する必要が生じる場合があります。これにどう対処するか、チーム内で合意をとりましょう。これには、チームメンバーが Slack チャンネルや電子メールなどを介して警告し合うための通信ポリシーが必要な場合があります。ワークフローでそのような通信ポリシーがサポートされていれば、バージョン管理ソリューションのファイルチェックイン / チェックアウトメカニズムを使用してそれを処理することもできます。

3. プロファイリング

プロジェクトを頻繁にプロファイリングする

プロジェクトの問題が積み重ならないようにしましょう。スケジュールの後半や、プロジェクトでパフォーマンスの問題が発生し始めたときだけでなく、頻繁にプロファイリングしてください。プロジェクトの典型的な「パフォーマンス兆候」をよく理解しておくことで、新しいパフォーマンスの問題をより簡単に見つけることができます。新しいグリッチやスパイクを見つけた場合は、できるだけ早く調査してください。

重要なプロファイリングのヒント：

- 仮定に基づいてプロジェクトを最適化しないようにしてください。必ずプロファイリング中に特定したことに基づいて最適化します。

- Unity とプラットフォーム / ベンダー固有のプロファイリングツールの両方を使用して、プロジェクト全体で何が起きているかを明確に把握してください。

- Unity Editor 内でプロジェクトをプロファイリングするのは便利ですが、常にターゲットプラットフォームそのものでプロファイリングしてください。

- リグレッションを特定してさらに調査するための自動テストを検討してください。同様に、Profile Analyzer などのツールを使用して手動で比較するためのプロファイリングキャプチャを保存することをお勧めします。

Unity ツール

Unity 独自のプロファイリングツールは通常、プロジェクトの開発ビルドを必要とします。このため、パフォーマンスは最終的な非開発ビルドと同じではありませんが、デバッグビルド構成ではなくリリースビルド構成を選択していれば、かなり適切な概要が得られるはずです。

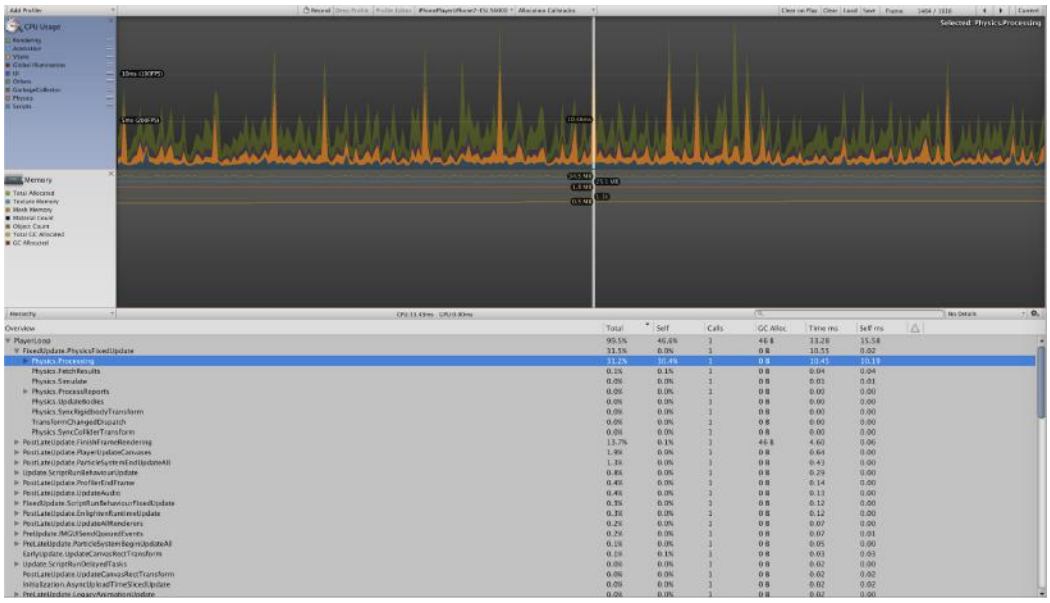
Unity プロファイラーを使用して主要なプロジェクト領域を調査する

Unity プロファイラーを頻繁に使用し、主要なプロジェクトの領域を調査してください。

意味のある Unity プロファイラーのサンプラーをスクリプトに追加すると、詳細プロファイリングに頼ることなく、より意味のある情報を提供できます。詳細プロファイリングは、パフォーマンスに大きな影響を与え、Unity Editor および Just-In-Time（JIT）Mono コンパイルをサポートするプラットフォーム（つまり Windows、macOS、および Android）でのみ機能します。

個々のプロファイラーツールの追跡（CPU、GPU、物理演算など）は並べ替えができることに加えて、追加および削除できることを忘れないでください。Unity 2018.3 以降では、現在は不要であるこれらの追跡を削除することで、ターゲットプラットフォーム上でのランタイムの CPU オーバーヘッドを軽減できます。以前の Unity バージョンでは、ランタイムによってすべてのプロファイリングデータが収集されていたため、プロファイラーのオーバーヘッドが増加していました。

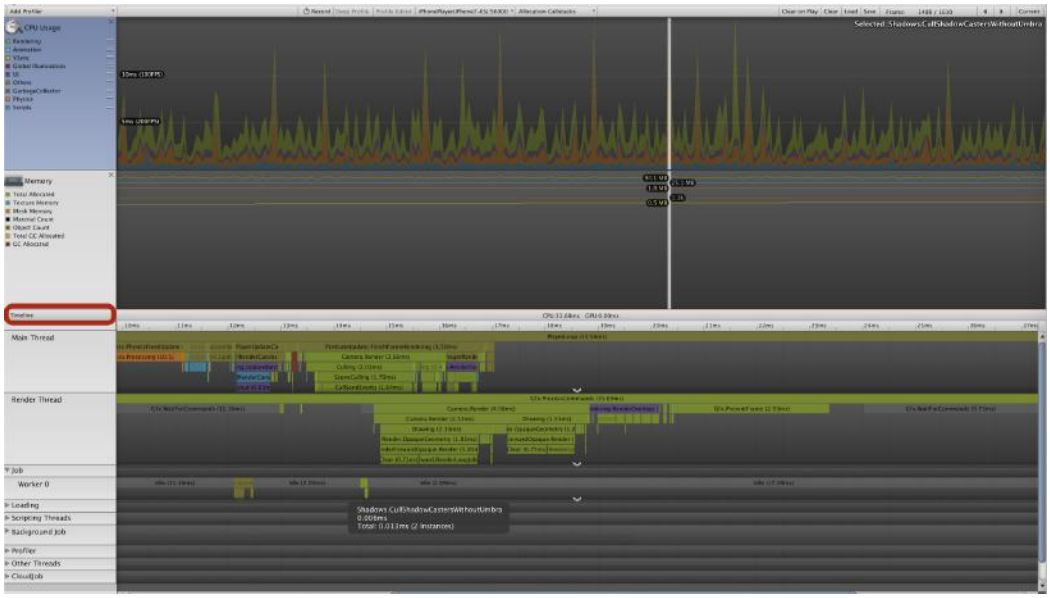
プロファイラーの「Hierarchy」ビュー内の「Total」列でソートして、CPU コストの観点で最も高コストの領域を特定します。これは調査が必要な領域を絞り込むのに役立ちます。



Unity プロファイラーの「Hierarchy」ビューを使用した最もコストが高いコード関数のプロファイリング

同様に、「GC Alloc」列でソートして、マネージアロケーションを生成している領域を特定します。特に定期的またはすべてのフレームで発生するアロケーションを可能な限り減らすことを目指し、マネージヒープの急速な上昇を招いてガベージコレクションをトリガーする可能性があるアロケーションスパイクに注目します。使用するメモリが多いほど断片化が進み、高コストなガベージコレクションが生じるため、マネージヒープのサイズ（メモリプロファイラートラックに「Mono」として報告されます）をできるだけ小さくします。

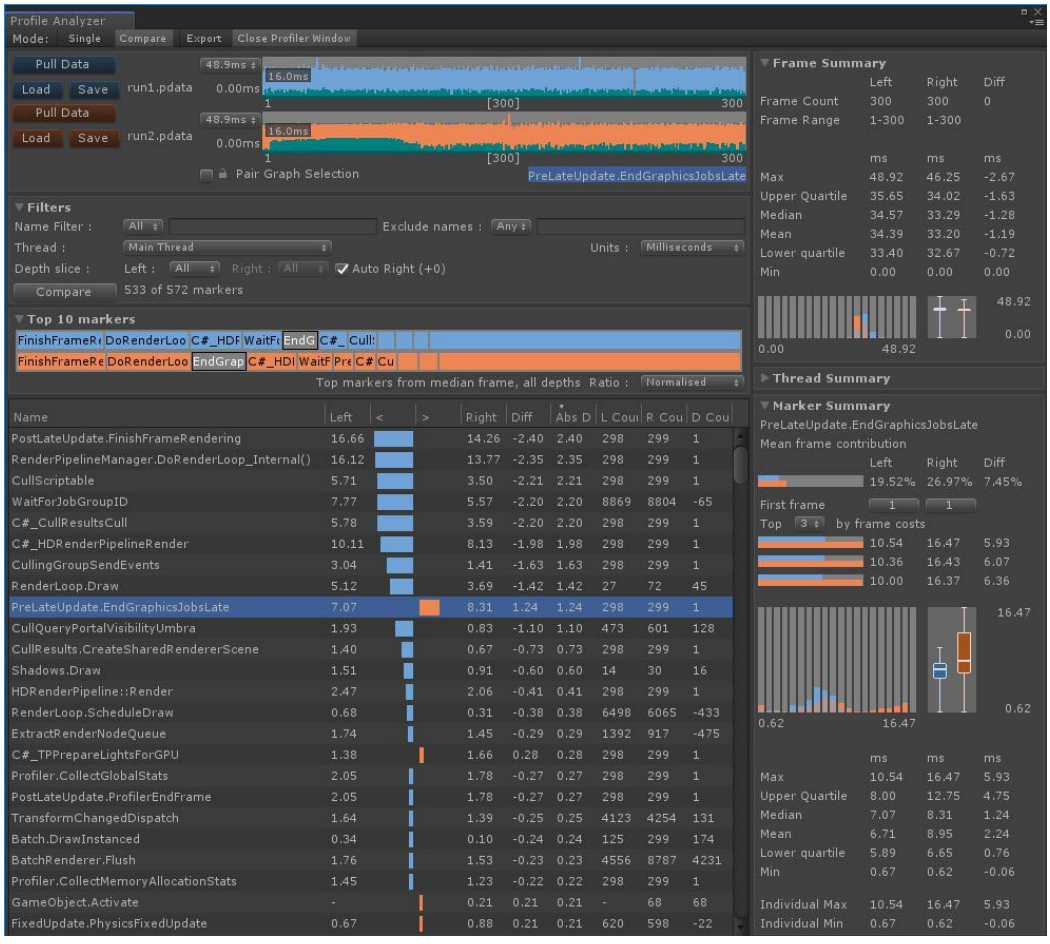
Unity メインスレッドのみに注目する「Hierarchy」ビューとは異なり、プロファイラーの「Timeline」ビューでは、Unity メインスレッドとレンダリングスレッド、Job System スレッド、ユーザースレッド（適切なプロファイラーサンプラーが追加されている場合）など、複数のスレッドにわたる有益な概要を見ることができます。



Unity プロファイラーの「Timeline」ビューを使用した複数スレッドにわたるプロファイリング

Profile Analyzer をリグレッションテストに使用する

Profile Analyzer では、Unity プロファイラーから現在キャプチャしているデータまたは以前にディスクに保存したキャプチャファイルのデータをインポートし、さまざまな分析を実行できます。平均、中央値、およびピークコストで分類されたプロファイリングマーカーを確認でき、加えてデータセットを比較することもできます。これは、たとえばリグレッションテストで変更の前後にキャプチャしたデータを比較する際に役立ちます。



変更前後のプロファイラーデータを比較して改善点やリグレッションを確認

メモリ消費量をキャプチャして視覚化する

メモリプロファイラー（プレビューパッケージを介してパッケージマネージャーで提供されます）は、アプリケーションが使用する一部のメモリ（すべてではありません）をキャプチャして視覚化できる強力なツールです。メモリプロファイラーの「Tree Map」ビューは、テクスチャやメッシュなどの一般的なアセットタイプ、および文字列やプロジェクト固有のタイプなどの他の管理対象のタイプのメモリ消費を即座に視覚化するのに特に役立ちます。

これらのアセット消費の領域に潜在的な問題がないかを確認します。たとえば、異常に大きく見えるテクスチャは、インポート設定が正しくないか、解像度が高すぎることを示している場合があります。重複アセットもここで確認できます。異なるアセットが同じ名前を持つ可能性もありますが、名前もサイズも同じアセットは重複の可能性があるため調査する必要があります。AssetBundle の割り当てが間違っていると、そのような問題が発生することがあります。

常にではありませんが、一般的にはメッシュメモリと比較してテクスチャメモリの割合が大きくなる傾向があります。そのため、テクスチャメモリの使用量と比べてメッシュメモリの使用量が大きくなっていないか、調べてください。

レンダリングとバッチ処理のフローを調べる

フレームデバッガーツールは、レンダリングやバッチ処理などの領域のフローを調べるのに非常に役に立ちます。このツールにより、たとえばバッチが壊れている理由がわかり、コンテンツを最適化できる場合があります。

フレームがどのように構築されているかを見ると、同じコンテンツが複数回レンダリングされている（たとえば現実世界のプロジェクトで重複したカメラが観察されている）、完全に隠されたコンテンツがレンダリングされている（たとえばフルスクリーン 2D UI の背後にある 3D シーンのレンダリングが継続されている）など、他の問題が明らかになる可能性があります。

プラットフォーム / ベンダー固有のツール

Unity のプロファイリングツールは多くの機能を備えていますが、ターゲットプラットフォーム用のネイティブなプロファイリングツールを効果的に使用する方法を学習することも非常に有用です。例を次に示します。

プラットフォーム	ツール
iOS と macOS	Xcode と Instruments
Android	Android Studio
Windows	VTune
Windows、Xbox One	PIX
Windows	NVidia NSight
PlayStation 4	Razor

これらのツールセットは、サンプルベースおよび計測ベースの CPU プロファイリング、完全ネイティブのメモリプロファイリング、シェーダーデバッグを含む GPU プロファイリングなど、より強力なプロファイリングオプションを提供します。通常、これらのツールはプロジェクトの非開発ビルドで使用します。

Unity プロファイラーマーカーを持つものだけでなく、すべての CPU コア、スレッド、および関数のパフォーマンスを測定できます。

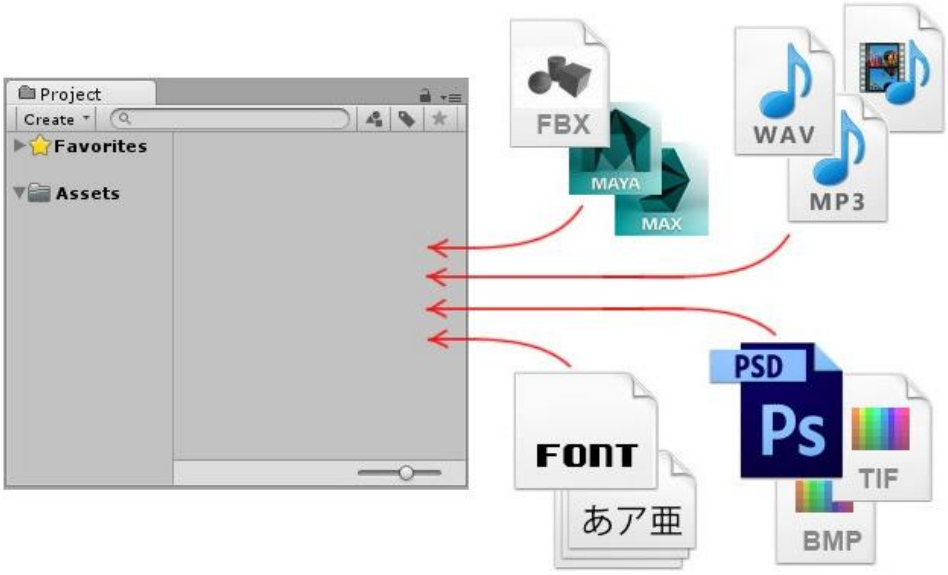
メモリプロファイリング（iOS の Instruments の Allocations ツールや VM Tracker ツールなど）を使用すると、サードパーティのプラグインが行ったネイティブ割り当てなど、Unity のメモリプロファイラーには表示されない、メモリを大量に消費する項目を明らかにできます。

4. アセット

アセットパイプラインは非常に重要です。プロジェクトのメモリフットプリントの大部分が、テクスチャ、メッシュ、サウンドなどのアセットによって使用されるため、設定が最適でない場合には多大なコストが発生する可能性があります。

これを避けるためには、適切な仕様で作成されたアートコンテンツのフローを適切に設定することが必要です。可能であれば、このプロセスの定義を支援してもらうために、最初から経験豊かなテクニカルアーティストに関与してもらいましょう。

最初に、使用するフォーマットと仕様について明確なガイドラインを定義します。



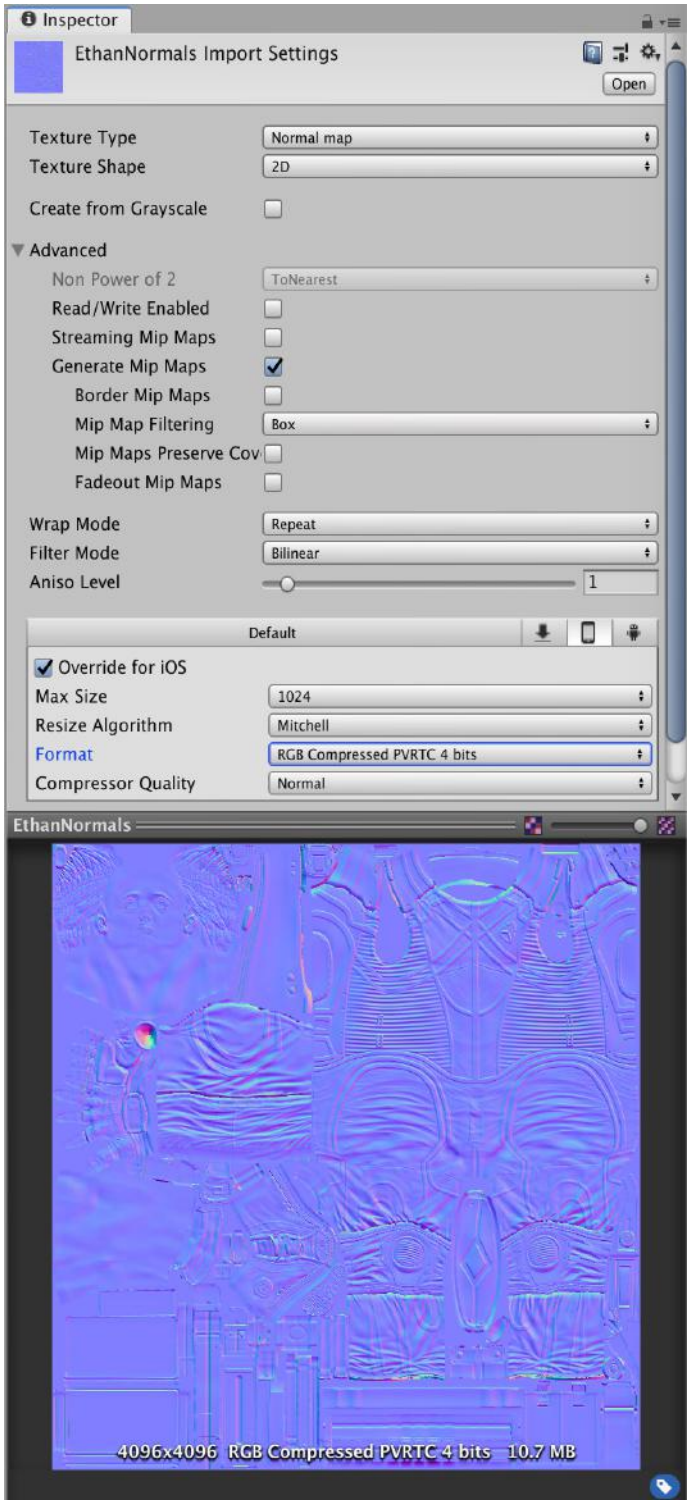
Unity は多くの種類のアセットファイルをインポートできる

プロジェクト内のすべてのアセットでインポート設定を正しく行うことが非常に重要です。チームが作業中のアセットタイプの設定を理解していることを確認し、すべてのアセットの設定が一貫していることを確認するためのルールを適用します。

また、単純にすべてのプラットフォームにデフォルト設定を使用するのは避けてください。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アセットを最適化します（さまざまな最大テクスチャサイズやオーディオ品質を設定するなど）。

このプロジェクトに示すように、AssetPostprocessor API を使用してインポート設定を新規アセットに適用する、自動化された方法をセットアップすることをお勧めします。

関連項目：アートアセットベストプラクティスガイド



テクスチャインポートを正しく設定する

テクスチャは通常、すべてのアセットタイプの中で最も多くのメモリを消費するため、インポート設定が正しいことを必ず確認します。

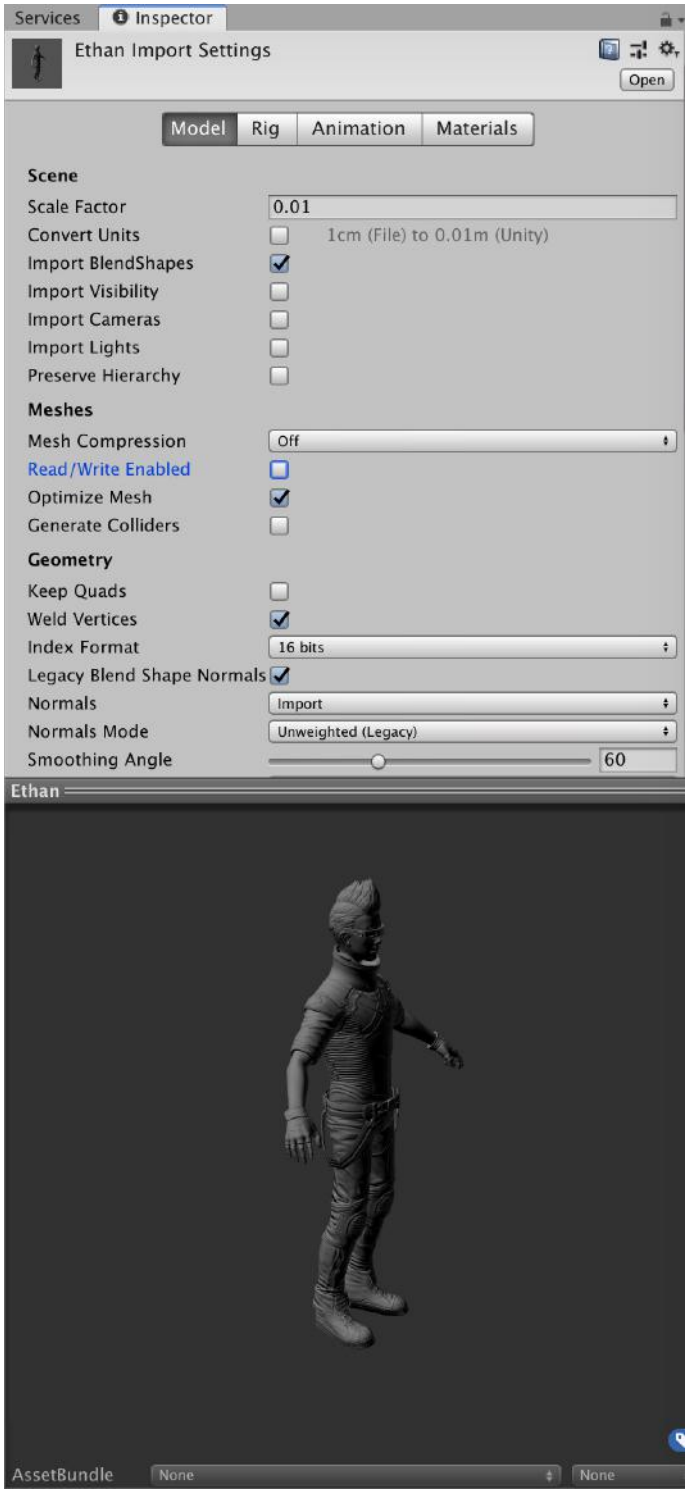
「Read/Write Enabled」オプションは、スクリプトからピクセルデータへのアクセスが絶対に必要である場合を除き、有効にしないでください。このオプションを有効にした場合、CPU アドレス可能メモリ内のピクセルデータのコピーと、GPU アドレス可能メモリにアップロードされたコピーが保持されるため、全体のメモリフットプリントが 2 倍になります。幸いなことに、ランタイムで Texture2D.Apply() API を使用し、パラメーター *makeNoLongerReadable=false*; を渡すことで、テクスチャアセットに CPU アドレス可能コピーを強制的に破棄させることができます。

ミップマップは、必要な場合にのみ有効にします。一般に、これは 3D シーンで使用されるテクスチャに使用し、2D では通常は必要ありません。不要な場合にこのオプションを有効なままにしておくと、テクスチャのメモリフットプリントが 1/3 ほど増加します。

可能な場合は圧縮を使用してください。ただし、圧縮時にすべてのコンテンツが現実感のあるビジュアルを十分に維持できるわけではありません。これは特に UI テクスチャに当てはまります。ターゲットプラットフォームに適用できる圧縮形式の長所と短所を理解して、適切な形式を選択してください。たとえば、通常は iOS で PVRT ではなく ASTC を使用すると、より忠実度の高い結果が得られますが、この形式は、Apple A8 チップ以降を搭載していないローエンドデバイスではサポートされていません。一部の開発者は、アセットの圧縮バージョンを複数含めることで、特定のデバイスがサポートする最高品質のバージョンを使用できるようにしています。

スプライトをグループ化するため、可能な限りスプライトアトラスを使用します。これにより、バッチ処理が向上し、ドローコールの数が減少します。これには、使用している Unity のバージョンに基づいて Unity SpritePacker または Unity SpriteAtlas を使用するか、Texture Packer など多数のサードパーティ製ツールの 1 つを使用します。

テクスチャアセットのインポート設定



モデルアセットのインポート設定

このメッシュオプションを無効にしてメモリを再利用する

前にも述べましたが、「Read/Write Enabled」オプションは、スクリプトからメッシュデータを読み取る必要がある場合にのみ有効にしてください。このオプションは、Unity 2019.3 ではデフォルトでオフになっていますが、これより前のすべてのバージョンの Unity ではデフォルトで有効になっており、通常はそのままにします。このオプションを無効にすると、多くのプロジェクトでメモリを再利用できます。

プロジェクトでさまざまなモデルソースファイル（FBX など）を使用して可視のメッシュとアニメーションをインポートする場合は、そのソースファイルからメッシュを確実に削除しておいてください。そうしないと、メッシュデータが最終出力に組み込まれたままとなり、メモリが無駄になります。

オーディオクリップに最適な圧縮形式を選択する

ほとんどのオーディオクリップでは、「Load Type」設定を「Compressed in Memory」（デフォルト）にするのが適しています。プラットフォームごとに適切な圧縮形式を選択してください。コンソールには独自のカスタム形式があり、その他のすべてには Vorbis が適しています。モバイルでは、Unity はハードウェア解凍を使用しないため、iOS に MP3 を選択するメリットはありません。

ターゲットプラットフォームに適したサンプルレートを選択します。たとえば、モバイルデバイスでの短いサウンドエフェクトは最大 22050 Hz にします。その多くは、これよりはるかに低くしても最終的な品質にはほとんど影響しません。

長い楽曲や環境音の「Load Type」は「Streaming」に設定します。そうでないと、アセット全体が一度にメモリにロードされます。

「Load Type」を「Decompress On Load」に設定すると、サウンドが raw 16 ビット PCM オーディオデータに解凍されるため、CPU コストとメモリ消費が発生します。これは通常、足音や剣がぶつかり合う音など、多くの場合、複数のインスタンスが同時に発生する短いサウンドにのみ適しています。

可能であれば元の（初期状態の）非圧縮 WAV ファイルをソースアセットとして使用します。圧縮形式（MP3 や Vorbis など）を使用した場合、Unity はこれを解凍してビルド時に再圧縮するため、不可逆パスが 2 回発生し、最終品質が低下します。

プロジェクトの 3D 空間内にサウンドを配置する場合、これらのサウンドをモノラル（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。これは、ランタイムで位置的に使用されるマルチチャンネルサウンドがリアルタイムでモノソースに平坦化されるため、CPU コストが増加しメモリの無駄が発生するためです。

Resource フォルダーではなく AssetBundle を使用する

アセットを Resource フォルダー内に配置する代わりに AssetBundle を使用します。Resource フォルダー内のアセットが内部の AssetBundle に組み込まれ、起動時にそのヘッダー情報がロードされます。このため、多数のアセットがこの方法で保存されていると、起動時間が長くなります。

重複アセットを避けるため、すべてのアセット、特に依存関係を持つアセットを明示的に AssetBundle に割り当てます。たとえば、同じテクスチャを使用する 2 つのマテリアルアセットがあるとしましょう。テクスチャが AssetBundle に割り当てられておらず、2 つのマテリアルが別々の AssetBundle に割り当てられている場合、テクスチャは参照マテリアルとともにそれらの各 AssetBundle に複製されます。これはテクスチャを明示的に AssetBundle に割り当てることで回避できます。

AssetBundle ブラウザーツールを使用して、複数の AssetBundle のアセットと依存関係を設定および追跡できます。また、AssetBundle Analyzer ツールを使用して、重複アセットに加えて設定が最適ではない可能性のあるアセットも強調表示できます。

何にでも対応できる万能のソリューションは存在しませんから、コンテンツを AssetBundle に組み込むための戦略を計画する必要があります。多くの開発者は AssetBundle を論理コンテンツ別にグループ化します。たとえば、レーシングゲームでは、各車種に必要なすべてのアセットをそれぞれ個別の AssetBundle に入れる方法があります。

ただし、パッチをサポートするプラットフォームの場合は、この戦略に注意が必要です。使用する圧縮アルゴリズムにより、1 つまたは複数のアセットに対する比較的小さな変更によって、AssetBundle のバイナリデータの多くが変更される可能性があります。これは、効率的なパッチ差分を生成できないことを意味します。このため、少数の大きな AssetBundle を使用している場合、これは大きな問題になる可能性があります。パッチを適用する際には、大きな AssetBundle を少数使用するのではなく、小さな AssetBundle を多数使用してください。

5. プログラミングとコードアーキテクチャ

以下のベストプラクティスに従い、アーキテクチャについての決定をスマートに行うことで、チームの生産性を高め、ゲームリリース後のユーザー体験を向上させることができます。

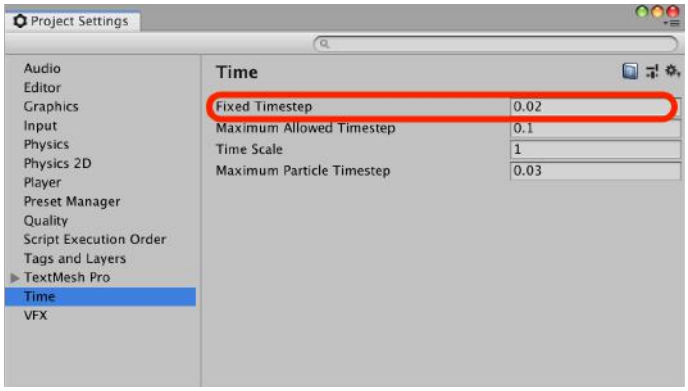
関連項目：スクリプティング体験を改善する方法

抽象化したコードは避ける

設計が過剰で抽象化されたコードは、特にチームの新しく入ったメンバーにとっては追跡が非常に難しくなる場合があり、一般に変更の分析や説明が困難になります。さらに、このタイプのコードでは生成されるコードが増え、ビルド時間が長くなり（トランスパイルする IL が多くなる IL2CPP ビルドを含む）、最終的なコードのパフォーマンスが低下する可能性があります。

Unity プレイヤーループを理解する

Unity フレーム（またはプレーヤー）ループを十分に理解してください。たとえば、Awake、OnEnable、Update などのメソッドがいつ呼び出されるかを知っていることは非常に大切です。詳細については Unity のドキュメントを参照してください。



デフォルトの Fixed Timestep 設定は余分な CPU 処理の原因になることがよくある

Update と FixedUpdate の違いは特に重要です。FixedUpdate メソッドは、プロジェクトの Fixed Timestep 値によって制御されます。これはデフォルトで 0.02（50 Hz）に設定されています。この値は、Unity での FixedUpdate メソッドの呼び出しが 1 秒間に 50 回行われることを意味します。このため、1 つのフレームで複数回呼び出される場合があります。フレームレートが低下すると、FixedUpdate 呼び出しの回数が増えるため、この問題はさらに悪化します。場合によっては深刻なグリッチが発生し、「死のスパイラル」と呼ばれるサイクルが発生する可能性があります。

物理演算システムの更新は FixedUpdate フェーズの一部であるため、物理コンテンツが多いゲームには大きな悪影響が及ぶ可能性があります。FixedUpdate を介してさまざまなゲームシステムを実行するプロジェクトもあるため、このようなパフォーマンスの問題については入念にチェックし回避するようにしてください。これらの理由から、Fixed Timestep 値をターゲットフレームレートに厳密に一致するよう設定することを強くお勧めします。

適切なフレームレートを選択する

適切なターゲットフレームレートを選択してください。たとえば、モバイルプロジェクトでは一般に、フレームレートの高さとバッテリ寿命およびサーマルスロットリングのバランスをとる必要があります。60 FPS でゲームを実行し、フレーム時間の大部分が CPU または GPU の負荷で占められている場合、バッテリーの寿命が短くなり、デバイス自体によって早期に CPU と GPU のクロック速度がスロットリングされます。多くのプロジェクトでは、30 FPS を目指すのが最適な妥協策です。また、Application.targetFrameRate プロパティーを使用してランタイムにフレームレートを動的に調整することも検討してください。

Unity 2019.3 ベータ版で提供された新しいオンデマンドレンダリング機能を使用すると、Input System などの他のシステムに影響を与えることなく、レンダリングの頻度を減らすことができます。

多くの場合、スクリプトロジックやアニメーションでは、フレームレートは考慮されません。更新に定数値を想定しないでください。代わりに Update メソッドでは Time.deltaTime、FixedUpdate メソッドでは Time.fixedDeltaTime を使用してください。

同期ロードを回避する

Unity メインスレッドでロードが同期的に実行されることが原因で、シーンまたはアセットのロード時にパフォーマンスのスパイクが発生することがよくあります。そのため、非同期のロードを使用してこうしたスパイクを最小限に抑え、堅牢なタイトルを設計しましょう。AssetBundle.LoadFromFile() および AssetBundle.LoadAsset() の代わりに、AssetBundle.LoadFromFileAsync() および AssetBundle.LoadAssetAsync() API を使用してください。

プロジェクトを非同期に設定すると、他にも役立ちます。なめらかなユーザーインタラクションが常に維持されます。サーバーの認証とやり取りを、シーンやアセットのロードと並行して簡単に実行できるため、全体的な起動時間およびロード時間を短縮できます。また、完全に非同期に設計することで、将来、新しい Addressable Asset System への移行が容易になります。

事前に割り当てられたオブジェクトのプールを使用する

オブジェクトのインスタンス化と破棄が頻繁に行われる場合（たとえば弾丸や NPC など）、事前に割り当てられたオブジェクトのプールを使用し、オブジェクトをリサイクルして再利用しましょう。このようなオブジェクトの特定のコンポーネントを再利用ごとにリセットすると CPU コストが発生する場合がありますが、完全にインスタンス化するより低コストです。このアプローチにより、プロジェクト内のマネージアロケーションの数も大幅に減少します。

標準の動作メソッドの使用を可能な限り減らす

すべてのカスタム動作は Update()、Awake()、Start() などのメソッドを定義する抽象クラスから継承されます。ある動作がこれらのメソッドのいずれかを必要としない場合（Unity ドキュメントにリストの全容があります）、空のままにせず削除してください。削除しないと、空のメソッドが Unity によって呼び出されます。これらのメソッドでは、ネイティブコード（C++）からマネージコード（C#）を呼び出すコストが原因で、わずかな CPU オーバーヘッドが発生します。

これは、Update() メソッドで特に問題になる可能性があります。Update() メソッドを持つオブジェクトが多数ある場合、この CPU コストが大きくなり、無視できない問題になります。1 つ以上のマネージャークラスで Update() メソッドを実装した後、そのマネージャークラスですべての個別オブジェクトを更新するという、「マネージャーパターン」を検討してください。この手法によって、ネイティブコードとマネージコードの間の遷移の回数が大きく減少します。詳細については、このブログ記事を参照してください。

また、プロジェクト内の各種のシステムをフレームごとに更新する必要があるかどうかを判断します。たいていの場合、システムはこれより低い頻度で実行しても問題ありません。つまり、システムを順番に呼び出すことができ、簡単な例としては 2 つのシステムを各フレームで交互に更新することができます。

また、システムが多数のアイテムを処理する一方で、フレームごとに特定の数のアイテムのみを処理することで負荷を複数のフレームに分散するという、タイムスライスの手法をお勧めします。これは、CPU のピーク負荷を抑えるのにも役立ちます。

これをより高度な形にしたのが、包括的な「予算型更新マネージャー」の実装です。この手法では、プロジェクトの各システムにフレームごとの最大時間予算を割り当てた後、割り当てられた時間内にできるだけ多くの作業を実行する標準インターフェースに基づいたマネージャーを各システムに実装します。このアプローチは、プロジェクトの生存期間全体でピーク CPU 負荷を管理するのに非常に役立ちます。このパターンに従ってシステムを設計することで、柔軟性を高めることができます。

コストの高い API の結果は必ずキャッシュする

データはできる限りキャッシュします。GameObject.Find()、GameObject.GetComponent()、Camera.main へのアクセスなど、よく見られる API 呼び出しはコストが高くなる可能性があるため、Update() メソッドで呼び出すことは避けてください。代わりに、Start() でこれらを呼び出し、結果をキャッシュします。

ランタイムでの文字列操作は避ける

ランタイムで連結などの文字列操作を多量に行うことは避けてください。多くのマネージアロケーションが生成され、ガベージコレクションのスパイクにつながる可能性があります。文字列（プレーヤーのスコアなど）は、更新ごとにではなく、実際に変更されたときにのみ再生成してください。また、StringBuilder クラスを使用することで、割り当ての回数を大幅に減らすことができます。

意図しないデバッグログを回避する

意図しないデバッグログによってプロジェクトでスパイクが発生することがよくあります。Debug.Log() のような API は、非開発ビルドでもログを継続します。このことに開発者が気付かないことがよくあります。これを防ぐために、**[Conditional(“ ENABLE_LOGS”)]** のように、メソッドの Conditional 属性を使用して、Debug.Log() などの API への呼び出しを独自のクラスにラップすることを検討してください。Conditional 属性に使用される定義が存在しない場合、メソッドとすべての呼び出し側が削除されます。

重要なパスで LINQ クエリを使用しない

LINQ クエリは強力で使いやすいため非常に魅力的ですが、マネージアロケーションが多量に生成され、CPU コストの点で高価になる可能性があるため、重要なパス（定期的な更新）で使用することは避けてください。LINQ クエリを使用する必要がある場合、不要に大きな CPU スパイクの原因にならないよう注意を払い、レベル初期化などの不定期な処理にのみ使用してください。

メモリ割り当てのない API を使用する

Unity には、Component.GetComponents() などのようにマネージアロケーションを生成する API があります。このような配列を返す API は内部的に割り当てを行います。代わりに割り当てのない API を使用できる場合は、常にそちらを使用してください。Physics API の多くには、代わりに使用できる割り当てなしの新しい API があります。たとえば、Physics.RaycastAll() の代わりに Physics.RaycastNonAlloc() を使用できます。

静的データ解析を避ける

プロジェクトでは多くの場合、JSON や XML などの読み取り可能な形式で保存されたデータを処理します。これはサーバーからダウンロードされたデータへのレスポンスで頻繁に行われますが、埋め込みの静的データでも一般的に発生します。この種の解析には時間がかかる可能性があり、通常はマネージアロケーションが大量に生成されます。代わりに、タイトルに組み込む静的データには、カスタムエディターツールで ScriptableObjects を使用してください。

関連項目：スクリプタブルオブジェクトを使ってゲームを構築するための 3 つの方法

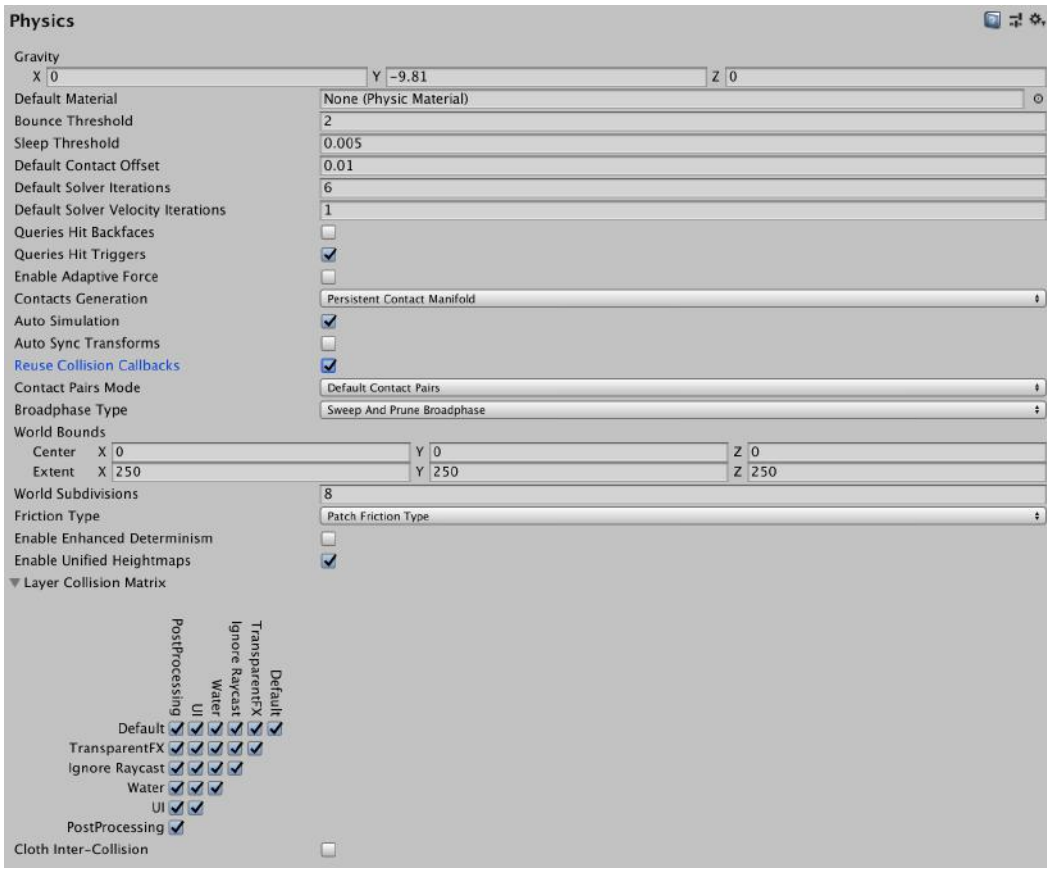
6. 物理演算

FixedUpdate でフレームごとに複数回実行される可能性があることが主な原因で、タイトルでの物理演算はコストが高すぎると見なされることがよくあります。ただし、パフォーマンスに影響する可能性がある、注意が必要な問題は他にもあります。

可能であれば「Player Settings」で「Prebake Collision Meshes」オプションを有効にして、ビルド時にランタイム物理メッシュデータを生成します。そうしないと、このデータはランタイム中のアセットのロード時に生成されます。コンテンツに多数の物理メッシュが含まれる場合、生成中に Unity のメインスレッドで高い CPU コストが発生する可能性があります。

メッシュコライダーはコストが高くなる可能性があるため、必要に応じて、比較的複雑なメッシュコライダーは 1 つまたは複数のよりシンプルなプリミティブタイプに置き換えて、元の形状に近づけます。

設定



Unity Editor 内での物理演算設定

後方互換性を維持するには、Unity 2017 で導入された後デフォルトで有効になっていた「Auto Sync Transforms」オプションは無効にすることを検討してください。このオプションが有効な場合、トランスフォームの変更が、基礎となる物理演算オブジェクトに自動的に即時に同期されます。しかし、物理演算シミュレーションがステップ実行される時点まで同期する必要がない多くの場合には、フレーム中に顕著な CPU 時間が発生する可能性があります。このオプションを無効にすると、同期が後の時点まで遅れますが、全体の CPU 時間を節約できる可能性があります。

衝突コールバックを使用している場合は、「Reuse Collision Callbacks」オプションを有効にします。これにより、コールバックごとに新しいオブジェクトを作成するのではなく、コールバック中に 1 つの内部 Collision オブジェクトを再利用することで、各コールバックでのマネージアロケーションを回避できます。また、コールバックの使用時は、コールバックで複雑な処理を行っていないか注意してください。衝突イベントが多い重いシーンではこの処理が大量になることがあります。Unity プロファイラーマーカーをコールバックに追加すると、パフォーマンスの問題が発生した場合に、コールバックがマーカーで示され追跡しやすくなります。

最後に、必要なレイヤーの組み合わせのみをチェックすることで、「Layer Collision Matrix」を最適化できます。

7. アニメーション

開発者は Animator を過剰に使用する傾向があるため、アニメーションはプロジェクトで驚くほど高価な機能になることがよくあります。

Animator は主にヒューマノイドキャラクターを想定した機能ですが、単一の値（UI 要素のアルファチャンネルなど）をアニメーション化するために使用されることもよくあります。Animator はステートマシンフローで使用すると便利ですが、これは比較的非効率なユースケースです。内部ベンチマークでは、ローエンドの iPhone 4S などのデバイスで Animator のパフォーマンスが Legacy Animation のパフォーマンスを上回るのは、約 400 のカーブがアニメーション化されている場合のみであることが示されています。

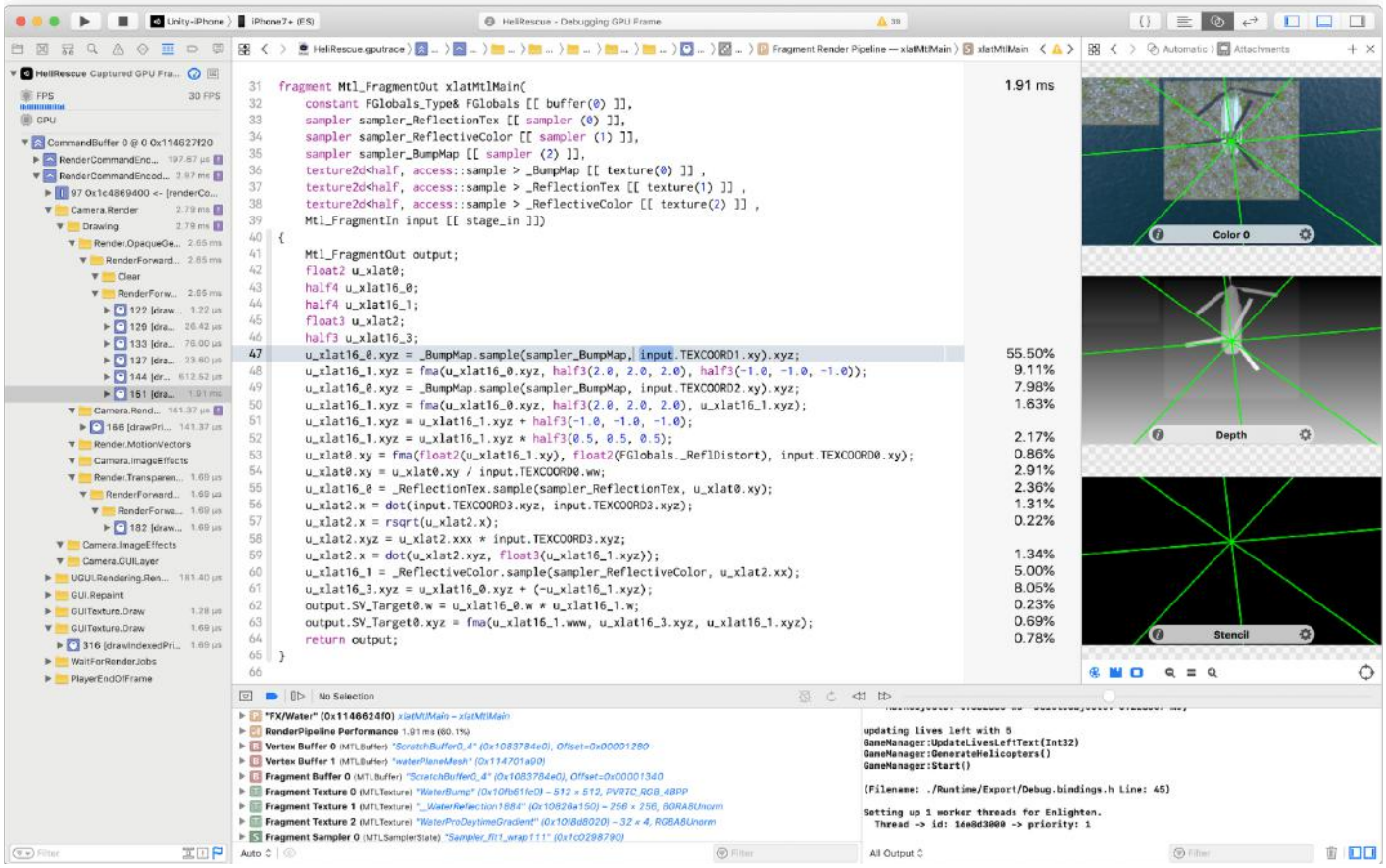
より単純なユースケース（アルファ値やサイズの変動など）については、独自のユーティリティスクリプトの作成など、より軽量な実装を検討するか、DOTween などのサードパーティ製のライブラリを使用してください。

最後に、Animator を手動で更新する場合は注意が必要です。Animator は通常、Unity の Job System を使用して並行処理を行いますが、手動で更新すると強制的にメインスレッドで処理が行われます。

8. GPU パフォーマンス

GPU プロファイリングを実行して、GPU パフォーマンスがどれだけ頂点シェーダー、フラグメントシェーダー、およびコンピュートシェーダーに消費されているかを明確にすることをお勧めします。これにより、最も高コストのドローコールを調査したり、最も高コストのシェーダーを特定したりでき、最適化が大幅に向上する可能性があります。

関連項目：
[グラフィックパフォーマンスの最適化]
[パフォーマンス最適化のためのキャラクターのモデリング]
[シェーダーのプロファイリングと最適化に関するヒント]



Xcode の GPU キャプチャツールを使った iOS でのシェーダーのデバッグとプロファイリング

オーバードローとアルファブレンドに注意する

アルファブレンドとオーバードローによって、特にモバイルプラットフォームは大きく影響を受けます。ゼロアルファピクセルを多量に含んだ複数レイヤーのアルファブレンドスプライトを持つ、大きくてほとんど見えないオーバーレイやエフェクトによって、GPU レンダリング時間が非常に長くなることも珍しくありません。

同様に、不要な透明画像の描画も避けてください。完全に透明である大きな領域が画像に含まれる場合（フルスクリーンのビネットオーバーレイなど）は、カスタムメッシュを作成して、これらのゼロアルファ領域のレンダリングを回避することをお勧めします。

シェーダーをシンプルに保ちバリエーションを極力減らす

モバイルでは、できる限りシェーダーをシンプルに保ちます。スタンダードシェーダーを使用せず、なるべく軽量化することが可能なカスタムシェーダーを使用します。ローエンドのターゲットデバイスでは簡略化したバージョンを使用するか、効果を無効にしてもよいでしょう。

シェーダーのバリエーションの数はできるだけ少なくします。これは、パフォーマンスに影響し、ランタイムのメモリ使用量にも大きな影響を与える可能性があるためです。

Camera コンポーネントの数を増やし過ぎない

レンダリングを実行するのに実際に必要である数を超えて Unity Camera コンポーネントを使用することは避けてください。たとえば、プロジェクトで UI レイヤーを構築するために複数のカメラが使用されていることは珍しくありません。各 Camera コンポーネントでは、意味のある動作を行うかどうかにかかわらずオーバーヘッドが発生します。これは、強力なターゲットプラットフォームでは無視できることもありますが、ローエンドまたはモバイルプラットフォームでは、それぞれ 1 ミリ秒の CPU 時間に達する可能性があります。

スタティックバッチングとダイナミックバッチングを検討する

同じマテリアルを共有する環境メッシュではスタティックバッチングを有効にします。これにより、Unity はオブジェクトカリングのメリットを受けながら、ドローコールとレンダリングの状態変化を大幅に減らすようにこれらのマテリアルをマージできます。

プロファイリングにより、ダイナミックバッチングがプロジェクトに適しているかどうかを確認します。ダイナミックバッチングは常に適しているわけではありません。オブジェクトを動的にバッチ処理するには、これらのオブジェクトが「類似」しており、非常に厳格で比較的単純な基準内にある必要があります。Unity のフレームデバッガーを使用して、特定のオブジェクトがバッチ処理されなかった理由を確認できます。

フォワードレンダリングと LOD を忘れない

フォワードレンダリングを使用する場合は、ダイナミックライトの数が多くなりすぎないようにしてください。すべてのダイナミックライトによって、照射されるすべてのオブジェクトに新しいレンダーパスが追加されます。

また、可能であれば詳細レベル（LOD）を使用してください。オブジェクトが遠方に移動するほど、単純なマテリアルとシェーダーを使用する単純なメッシュを使用することで GPU パフォーマンスを大幅に向上させることができます。

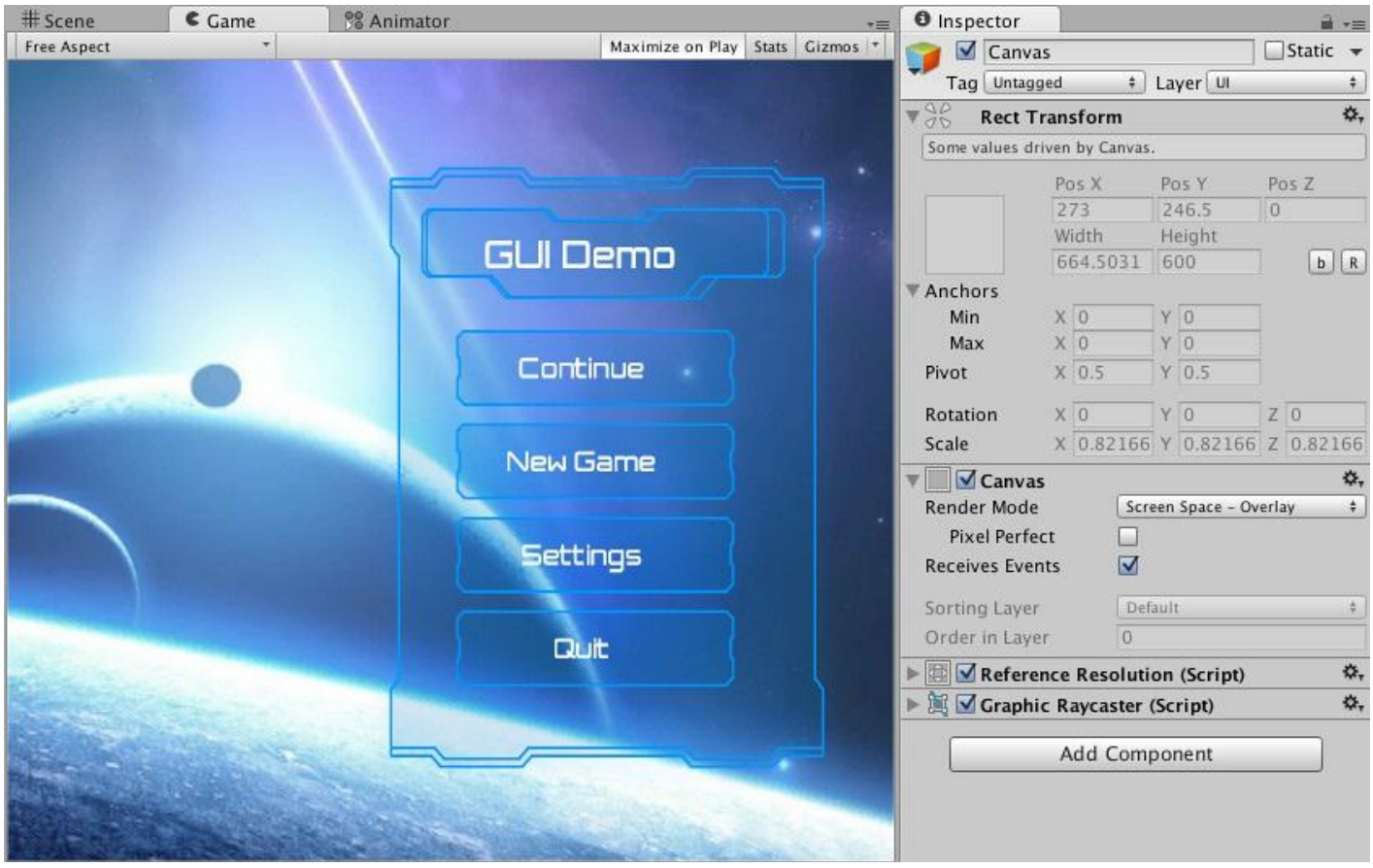
9. ユーザーインターフェース（UI）

Unity UI（UGUI とも呼ばれます）が原因でプロジェクトのパフォーマンスの問題が発生することがよくあります。詳細については、「Unity UI 最適化のヒント」を参照してください。

複数の解像度とアスペクト比を検討する

Unity UI を使用すると、さまざまな画面解像度とアスペクト比に対応するよう位置とスケールを調整できる UI を簡単に構築できます。ただし、単一のレイアウト / デザインがすべてのデバイスでうまく機能するとは限らないため、各種のデバイスで最高のエクスペリエンスを提供するには、複数のバージョンの UI（または UI の一部）を作成した方がよいこともあります。

サポートされている多様なデバイスで広範な UI テストを必ず実施し、ユーザー体験が良好で、すべてのデバイスで一貫していることを確認してください。



Unity UI キャンバスの設定

少数のキャンバスの使用は避ける

すべての UI コンテンツを単一または少数の「モノリシック」なキャンバスに配置することは避けてください。各キャンバスは、そのすべてのコンテンツのメッシュを維持しており、いずれか 1 つの要素が変更されると、このメッシュが再構築されます。

コンテンツを、できれば更新頻度によって個別のキャンバスに分割します。動的な要素を静的な要素から分離しておくことで、静的なメッシュデータを絶えず再構築する CPU コストを回避できます。

レイアウトグループに注意する

レイアウトグループも、特にネストされている場合、パフォーマンスの問題の原因になることがよくあります。レイアウトグループ内の UI グラフィックコンポーネントが変更された場合（たとえば、ScrollRect が移動された場合）、UGUI はシーン階層の上方に向けて親レイアウトグループを再帰的に検索し、レイアウトグループを持たない親を特定します。その後、その下にあるすべてのレイアウトが再構築されます。

特にコンテンツが真に動的でない場合、可能な限りレイアウトグループは避けるようにしてください。レイアウトグループがコンテンツの初期レイアウトを実行する目的にのみ使用されており、その後このコンテンツが変更されない場合は、コンテンツの初期化後にこれらのレイアウトグループコンポーネントを無効にするカスタムコードを追加することをお勧めします。

リストビューとグリッドビューはコストが高い可能性がある

リストビューとグリッドビューも一般的な UI パターンです（インベントリやショップ画面など）。これらのパターンで、数百のアイテムが存在する可能性があるのに対して一度に表示できるアイテム数が少ない場合、コストが非常に高くなるため、すべてのアイテムの UI 要素を作成することは避けてください。代わりに、要素を再利用するパターンを実装し、片側から要素が外に出たときに反対側にその要素を表示します。この GitHub プロジェクトに Unity エンジニアによる例が提供されています。

多数のオーバーレイ要素を避ける

多数のオーバーレイされた要素で構成された領域を持つ UI はよく使用されます。たとえば、カードバトラーゲームのカードプレハブがその良い例です。このアプローチでは、デザインに多くのカスタマイズを加えることができますが、ピクセルのオーバードローが多くなると、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。さらに、より多くのドローバッチが発生する可能性があります。レイヤー化された多数の要素を、より少数の（または 1 つの）要素にマージできるかどうかを判断してください。

Mask および RectMask2D コンポーネントの使用方法について考える

Mask コンポーネントおよび RectMask2D コンポーネントは UI で一般的に使用されています。Mask はレンダーターゲットのステンシルバッファを利用して描画中のピクセルを描画、または拒否し、GPU コストのほぼすべてを専有します。一方、RectMask2D は、CPU で境界チェックを実行してマスク外の要素を拒否します。多数の RectMask2D コンポーネントが含まれる複雑な UI では、特にネストされている場合、境界チェックを実行する際に大きな CPU コストが発生する可能性があります。過剰な数の RectMask2D コンポーネントを使用しないように注意してください。または、GPU の負荷が CPU の負荷より少ない場合には、Mask コンポーネントに切り替えて全体の負荷のバランスをとることが望ましいかどうかを検討してください。

UI テクスチャをアトラス化してバッチ処理を改善する

バッチ処理を改善するため、UI テクスチャを可能な限りアトラス化してください。テクスチャの論理グループに多数のアトラスを使用することは理にかなっていますが、アトラスのサイズが大きすぎたり、テクスチャが少なすぎたりしないように注意してください。メモリの無駄の原因となることがよくあります。

また、可能な場合はアトラスを圧縮してください。アーティファクトが生じるため、多くの場合 UI テクスチャの圧縮は望ましくありませんが、これはコンテンツそのものの性質によって異なります。

ほとんどの場合、UI テクスチャではミップマップは必要ありません。このため、特に必要な場合（たとえば、ワールドスペース UI の場合）を除いて、このインポート設定が無効になっていることを確認してください。

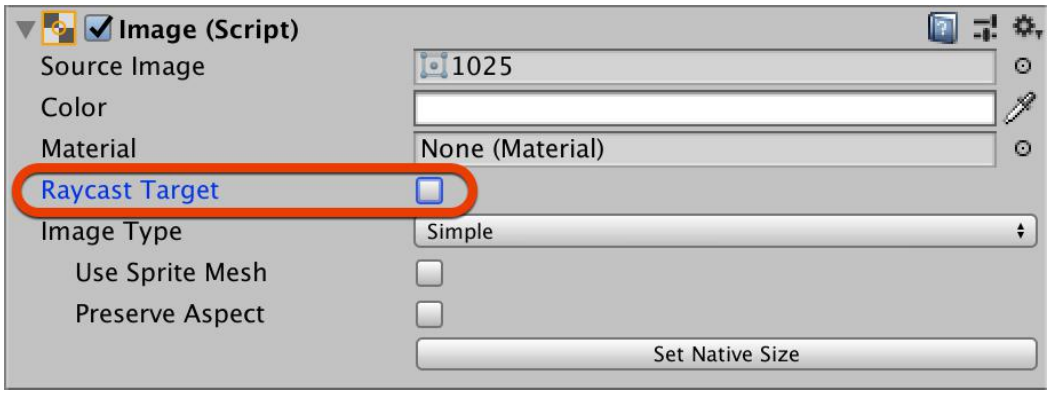
新しい UI ウィンドウまたは画面を追加するときは注意する

新しい UI ウィンドウや画面を導入すると、プロジェクトにグリッチが発生することがよくあります。これには多くの理由が考えられます（たとえば、アセットのオンデマンドロードや、多数の UI コンポーネントがある複雑な階層をインスタンス化する莫大なコストのため）。

UI 自体の複雑さを軽減することに加えて、特に頻繁に使用されている場合には、そのような UI コンポーネントをキャッシュすることを検討してください。毎回破棄して再インスタンス化するのではなく、無効にして再度有効にします。

Raycast Target を不要時に無効にする

入力イベントを受け取る必要がない UI グラフィック要素では「Raycast Target」オプションを必ず無効にしてください。ボタンのテキストや非インタラクティブ画像など、多くの UI 要素は入力イベントを受け取る必要がありません。しかし、UI グラフィックコンポーネントの「Raycast Target」オプションはデフォルトで有効になっています。複雑な UI には、多数の不要な Raycast Target ターゲットが含まれている可能性があります。これらを無効にすると、CPU 処理を大幅に減らすことができます。



入力イベントが不要な要素では Raycast Target を無効化

次のステップ

このガイドで説明したベストプラクティスとヒントを、プロジェクトの構造を堅牢にし、ゲームの設計、開発、テスト、リリースに適したツールとワークフローを使用するのに役立てていただければ幸いです。ニーズがより高度になり、スケジュールが厳しくなってきた場合には、その他の共有 Unity リソースを参照してください。

追加情報

Unity ブログ、Unity コミュニティフォーラムのハッシュタグ #unitytips、および Unity Learn では、さらなる最適化のヒント、ベストプラクティス、およびニュースをご覧いただけます。

Unity ISS へのお問い合わせ

個別のサポートをご希望ですか？ Unity Integrated Success Services をご検討ください。ISS は単なるサポートパッケージではありません。専任のデベロッパーリレーションマネージャー（DRM）が協力して、すぐに Unity エキスパートがお客様のチームの一員となりプロジェクトを支援します。DRM は、問題を事前に阻止し、プロジェクトをリリース以降までスムーズに進めるために必要な技術および運用に関する専門知識を提供します。お問い合わせには、ご連絡フォームをご利用ください。


unity