→ E ブック

入門編

上級 Unity クリエイター向けのユニバーサルレンダーパイプライン（URP）

Unity®

Contents

はじめに

このガイドは、経験豊富な Unity 開発者やテクニカルアーティストが Unity 6 のユニバーサル レンダーパイプライン（URP）を活用して可能な限り効率よく開発を進めるための助けとなります。

URP と HD レンダーパイプライン（HDRP）は、Unity が提供する 2 つのレンダリングソリューションで、Scriptable Render Pipeline（SRP）フレームワークの上に構築されています。これらの SRP を使用すると、C++ のような低水準プログラミング言語を使用することなく、オブジェクトのカリング、描画、およびフレームのポストプロセスをカスタマイズできます。また、独自の完全にカスタマイズされた SRP を作成することもできます。

URP は、モバイル、XR、コードレスハードウェア向け 2D および 3D ゲーム用の Unity のデフォルトレンダラーです。これはビルトインレンダーパイプラインの後継で、学習やカスタマイズがしやすく、Unity がサポートするすべてのプラットフォームに効率よく拡張できるよう設計されています。Unity 6 では、ビルトインレンダーパイプラインと同等の機能を提供し、多くの面でその品質レベルやパフォーマンスを上回っています。

この e ブックでは、以下のような分野について、専門的なガイダンスとベストプラクティスを提供しています。

- 新規プロジェクトにおいて URP を設定、または既存のビルトインレンダーパイプラインベースのプロジェクトを URP に変換する。
- URP クオリティ設定の使用。
- URP で利用可能なすべてのライティングツールを使用し、アダプティブプローブボリューム（APV）などの新機能を活用して、リアルタイムのグローバルイルミネーションや昼夜のライティングシナリオを実現する。

- URP のシェーダーを使ってライティングされたシーンを作成し、URP シェーダーとビルトインレンダーパイプラインシェーダーの違いを理解する。
- カスタムシェーダー、インクルード、HLSL インクルードを使用する。
- URP のポストプロセッシングフレームワークを活用し、ローカルボリュームの追加やコードによるポストプロセスの制御を行う。
- レンダリングレイヤーの使用。
- 新しくリリースされた GPU Resident Drawer や GPU オクルージョンカリングなど、URP のツールを使ってさまざまな方法でパフォーマンスを最適化する。
- Renderer Features（レンダラー機能）やレンダーグラフシステムを利用して、レンダーパイプラインをカスタマイズする。

マルチプラットフォームでのデプロイは、多くのゲームにとって成功の重要な要因です。プレイヤーは、コンソールとモバイルなど、異なるデバイスで同じゲームをプレイすることが多く、Unity の開発者は、ステップや複雑さをできる限り少なくしつつ、多数のデバイス用にスケールアップおよびスケールダウンするレンダリングオプションを必要としています。

URP はスケーラビリティやカスタマイズ性、豊富な機能セットにより、様式化されたビジュアルから物理ベースのレンダリングまで、あらゆるタイプのプロジェクトで創造の自由を提供します。

URP で作成されたシーン

著者と貢献者

この e ブックの著者である **Nik Lever** は、90 年代半ばからリアルタイム 3D コンテンツを制作しており、2006 年から Unity を使用しています。30 年以上にわたり、中小開発会社 Catalyst Pictures を率い、急速に進化する業界におけるゲーム開発者の知識を広げることを目的に、2018 年からコースを提供しています。

Unity の貢献者

Steven Cannavan は、Accelerate Solutions Games チームのシニア開発コンサルタントで、スクリプタブルレンダーパイプラインを専門としています。Steven はゲーム開発業界で 16 年以上の経験を持っています。

Maxime Grange は 8 年の経験を持つシニアテクニカルアーティストで、VR インディーゲームからキャリアをスタートし、Unity でライトテクニカルアーティストとして活躍しています。レンダリング技術やシェーダー、アーティスト向けツールの開発に情熱を注ぎ、最適なランタイムパフォーマンスを維持しながら魅力的なビジュアルを実現することに注力しています。

Felipe Lira はゲーム業界のソフトウェアエンジニアとして 14 年以上の経験を持ち、グラフィックプログラミングとマルチプラットフォームゲーム開発を専門としています。

Ali Mohebali はゲーム業界で 21 年の経験を持ち、Halfbrick Studios の『*Fruit Ninja*』や『*Jetpack Joyride*』などのヒットタイトルに貢献しています。

Adrien Moulin は Unity のレンダーパイプラインチームでシニアグラフィック開発者を務めています。シミュレーションやリアルタイムソフトウェア業界で 8 年以上の経験があり、現在は、スクリプタブルレンダーパイプラインのユーザーに最適な基盤と API を提供することに注力しています。

Mathieu Muller は Unity のグラフィックス担当リードプロダクトマネージャーで、グラフィックス製品管理チームを率い、グラフィックスのロードマップと製品ビジョンを監督しています。

Damian Nachman は Unity のグラフィックスチームでシニアテクニカルプロダクトマネージャーを務め、低レベルグラフィックスの開発と最適化を専門としています。リアルタイムグラフィックスエンジンや複数業界のベンチマークで 10 年の経験を持ちます。

Oliver Schnabel は Unity のグラフィックスチームでシニアテクニカルプロダクトマネージャーを務め、顧客インサイトを統合し、グローバルスタジオと協力して高性能で統一されたスケーラブルなレンダリングスタックの開発に尽力しています。コンピューターグラフィックスとリアルタイム開発における豊富な経験を持ちます。

URP：ビルトインレンダーパイプラインの後継

Unity の最大の強みの 1 つは、プラットフォーム対応力です。すべてのゲームスタジオにとって理想的なのは、一度ゲームを制作し、それをハイエンド PC からローエンドモバイルまで、希望のプラットフォームに効率的に展開することです。

ビルトインレンダーパイプラインは、Unity がサポートするすべてのプラットフォーム向けのオールインワンソリューションとして開発されました。さまざまなグラフィックス機能の組み合わせをサポートしており、フォワードおよびディファードレンダリングパイプラインで効率的に活用できます。

しかし、Unity がサポートするプラットフォームが増えるにつれ、ビルトインレンダーパイプラインの欠点も見えてきました。

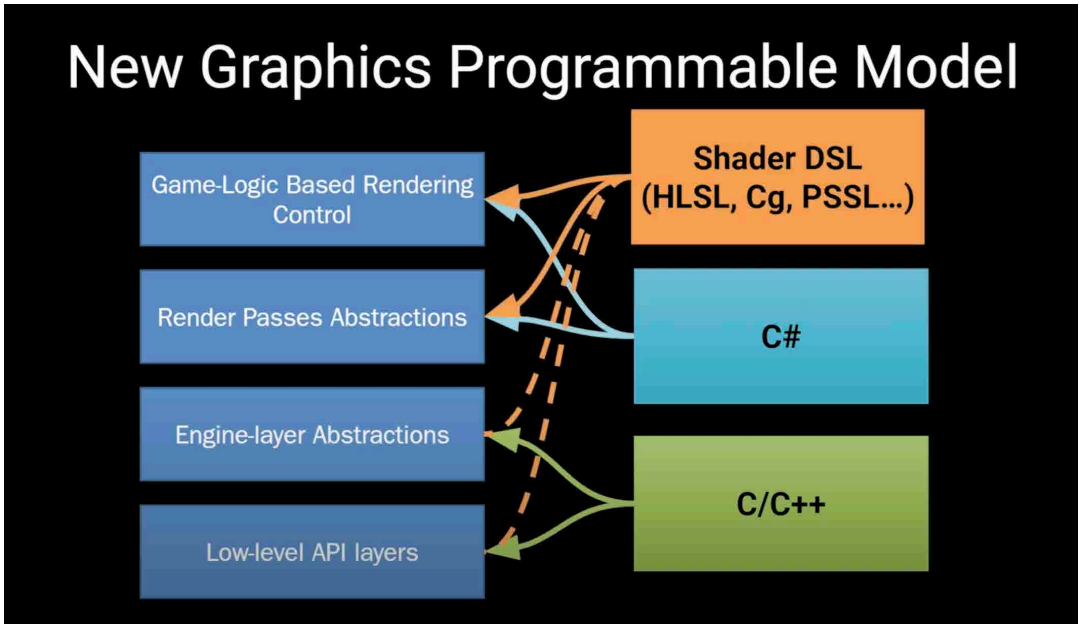
- コードの大部分が C++ で書かれ、開発者が変更できないため、ブラックボックスシステムとなっている。
- レンダーフローとレンダーパスが事前に構造化されている。
- レンダリングアルゴリズムがハードコードされている。
- カスタマイズの制約がないため、すべてのプラットフォームで高いパフォーマンスを実現するのが難しい。
- パイプラインの同期ポイントをトリガーするレンダリングコード内のコールバックが公開される。これらのコールバックにより、マルチスレッドレンダリングの最適化ができなくなり、C# の呼び出しによってフレーム内の任意の時点でステート注入による動的な変更が可能になる。
- ユーザー注入対策で永続状態を管理するためのデータのキャッシュが困難。

解決策：スクリプタブルレンダーパイプライン

SRP は、以下の提供を通して、効率的なマルチプラットフォームワークフローをサポートするために開発されました。

- ハイエンドデバイスからローエンドデバイスまで、最大数のハードウェアプラットフォームに対応するインテリジェントで信頼性の高いスケーリング。
- C++ ではなく、C# を使用してレンダリングプロセスをカスタマイズする機能。C# を使用するため、変更ごとに新しい実行ファイルをコンパイルする必要がない。
- アーキテクチャの進化をサポートする柔軟性。
- 多くのプラットフォームで高パフォーマンスとシャープなグラフィックスを実現できる柔軟性。

以下の図は、SRP の仕組みを示しています。SRP では、C# を使用してレンダーパスやレンダリングコントロールを制御・カスタマイズでき、また Shader Graph などのアーティストにとって使いやすいツールで作成可能な HLSL シェーダーを利用できます。シェーダーを使用すると、さらに低レベルの API やエンジンレイヤーの抽象化にもアクセスできます。



スクリプタブルレンダーパイプラインの新しいグラフィックスプログラマブルモデル

上級ユーザーは、ゼロから新しい SRP を作成したり、HDRP や URP に変更を加えたりできます。グラフィックススタックはオープンソースで、GitHub から入手可能です。

URP を選択する理由

— **幅広いユーザーがアクセス可能：**URP は、アーティストやテクニカルアーティストでも設定でき、プロトタイピングの柔軟性を高め、ゲーム全体の制作でレンダリング技術を洗練する助けとなります。

— **拡張およびカスタマイズ可能：**URP は、ユーザーが既存の機能を変更したり、新たな機能でパイプラインを拡張できる柔軟性を備えているため、アセットストアやサードパーティパッケージのクリエイター、経験豊富なスタジオ、上級者チームなどにとって信頼できる選択肢となっています。

低レベルのレンダリング API はパフォーマンス向上のために C++ で書かれていますが、URP 開発者はレンダーパイプライン内で呼び出されるシンプルな C# スクリプトを記述することで、パフォーマンスを犠牲にせずに高レベルなカスタマイズを行えます。

— **複数のレンダリングオプション：**URP は、フォワード、フォワード +、ディファードのレンダリングパスをサポートする Universal Renderer と、2D Renderer を提供します。

これらのレンダラーは、追加機能やスクリプタブルレンダーパスによる拡張が可能です。Render Objects 機能を使用して、レンダリングパイプラインの異なるイベントで、任意のレイヤーマスクのオブジェクトをレンダリングできます。さらに、オブジェクトのレンダリング時にマテリアルや他のレン

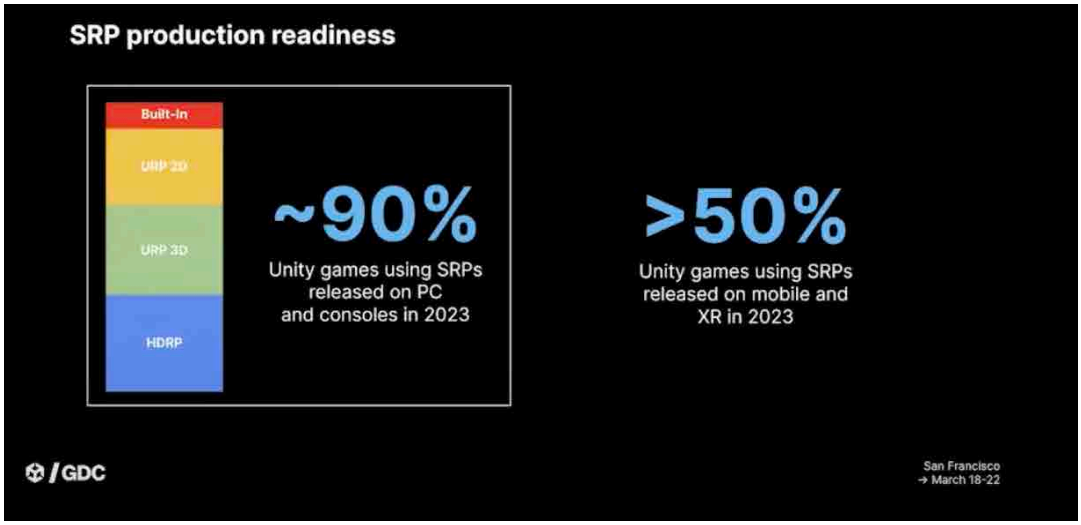
ダーステートをオーバーライドできるため、コードなしでレンダリングをカスタマイズできます。URP は、特定のニーズに合わせてカスタムレンダラーで拡張できます。

レンダー グラフ システムを使用すると、フレームのレンダリングに用いるさまざまなバッファにアクセスして操作できます。Renderer Features ワークフローを利用することで、レンダーパイプラインの任意の段階でこれを注入することが可能です。

- **優れたパフォーマンス**：URP は堅牢なパフォーマンスを提供し、多くのデバイスで忠実度とフレームレートのバランスを調整するためのツールを多数提供しています。特に、以下の点で効果を発揮します。
  - URP はリアルタイムライティングを非常に効率よく評価します。フォワードレンダリングでは、すべてのライティングを 1 つのパスで評価します。フォワード + は、オブジェクトごとではなく空間的にライトをカリングすることで、標準のフォワードレンダリングを改善します。これにより、フレームのレンダリング時に利用できるライト全体の数が増加します。ディファードレンダリングでは、Native RenderPass API をサポートしており、G バッファとライティングパスを 1 つのレンダーパスにまとめられます。
  - メッシュ描画時の CPU と GPU の効率が改善しました。SRP Batcher はドローコールの削減と深度処理の改善に役立ちます。また、GPU Resident Drawer およびオクルージョンカリングにより、一部のシーンでドローコールを大幅に削減できます。
  - URP は、モバイルデバイス上でタイルメモリを効率的に活用することで、消費電力の削減やバッテリー寿命の延長、長時間のプレイセッションも可能にします。
  - URP には、ビルトインレンダリングパイプラインと比較してより優れたパフォーマンスを可能にするポストプロセススタックが組み込まれています。Volume フレームワークを使用すると、コードを書かずに、カメラ位置に依存するポストプロセスエフェクトを作成することができます。
- **他の主要ツールとの互換性：**URP は、Shader Graph、VFX Graph、レンダリングデバッガーなど、アーティストやテクニカルアーティストにとって使いやすいツールをサポートします。
- **2D レンダリング：**URP は高度な 2D ライティング、影、ポストプロセス効果をサポートしており、パフォーマンスを維持しつつ 2D ゲームのビジュアル品質を向上させます。

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

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

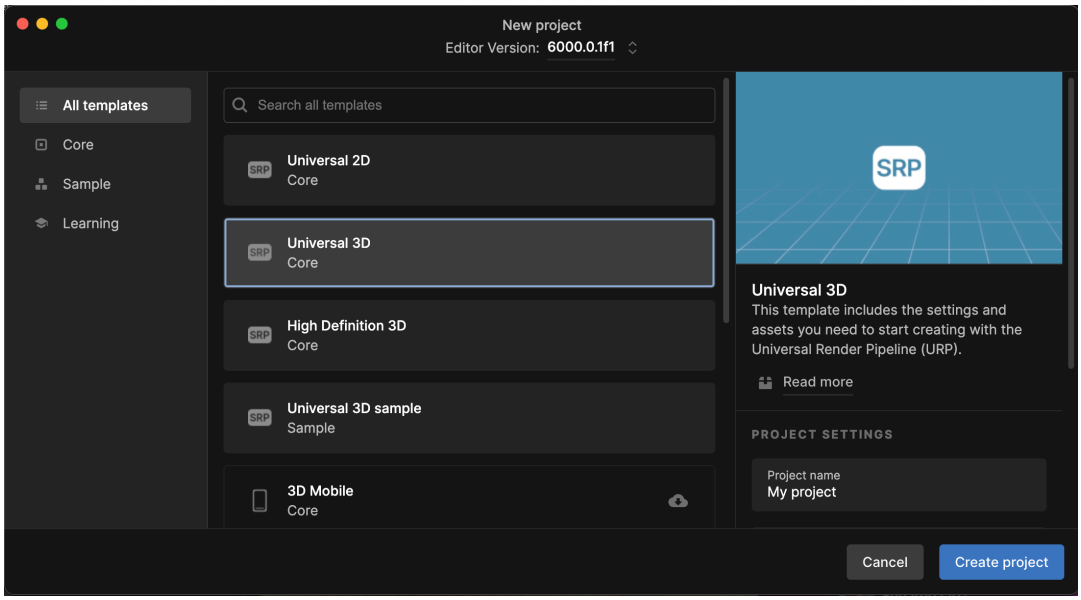


Unity で利用可能な各パイプラインが使用されているゲームの割合

現在、ほとんどのプロジェクトが URP または HDRP で構築されていますが、ビルトインレンダーパイプラインは、Unity 6 でも利用可能なオプションとして残る予定です。

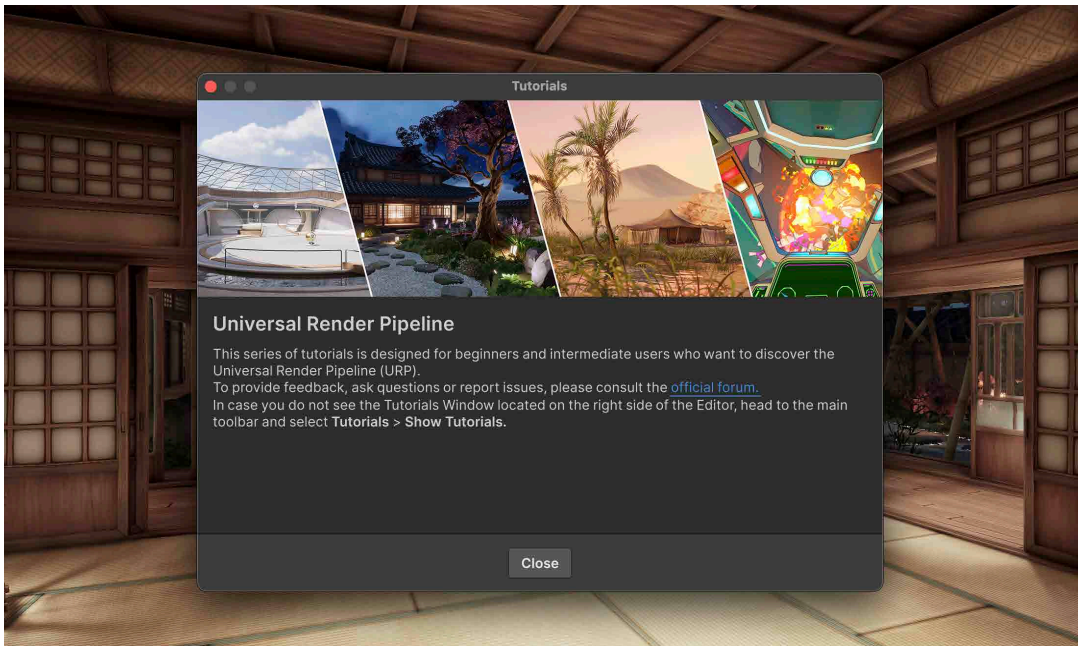
新しい URP プロジェクトを開く方法

URP を使って新しいプロジェクトを開くには、Unity Hub を使用します。「**New**」をクリックし、ウィンドウ上部で選択されている Unity バージョンが Unity 6 以降であることを確認します。プロジェクトの名前と場所を選択し、**Universal 2D** または **Universal 3D** テンプレートを選択して、「**Create**」をクリックします。



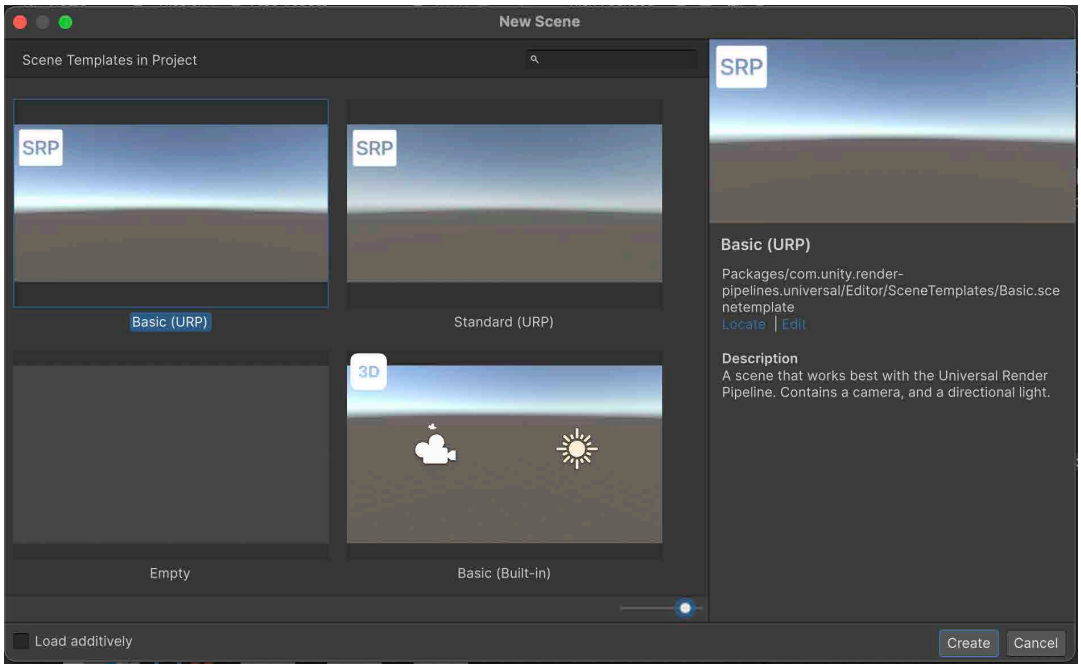
URP テンプレートを使って新しいプロジェクトを作成する際、最初にテンプレートをダウンロードする必要があるかもしれません。

注：テンプレートを使用すると、プロジェクトはライティングを正しく計算するために必要なリニア色空間を使用するように設定されます。



本書の最後で紹介する URP 3D サンプルシーンは、ダウンロード可能なテンプレートとして利用可能です。

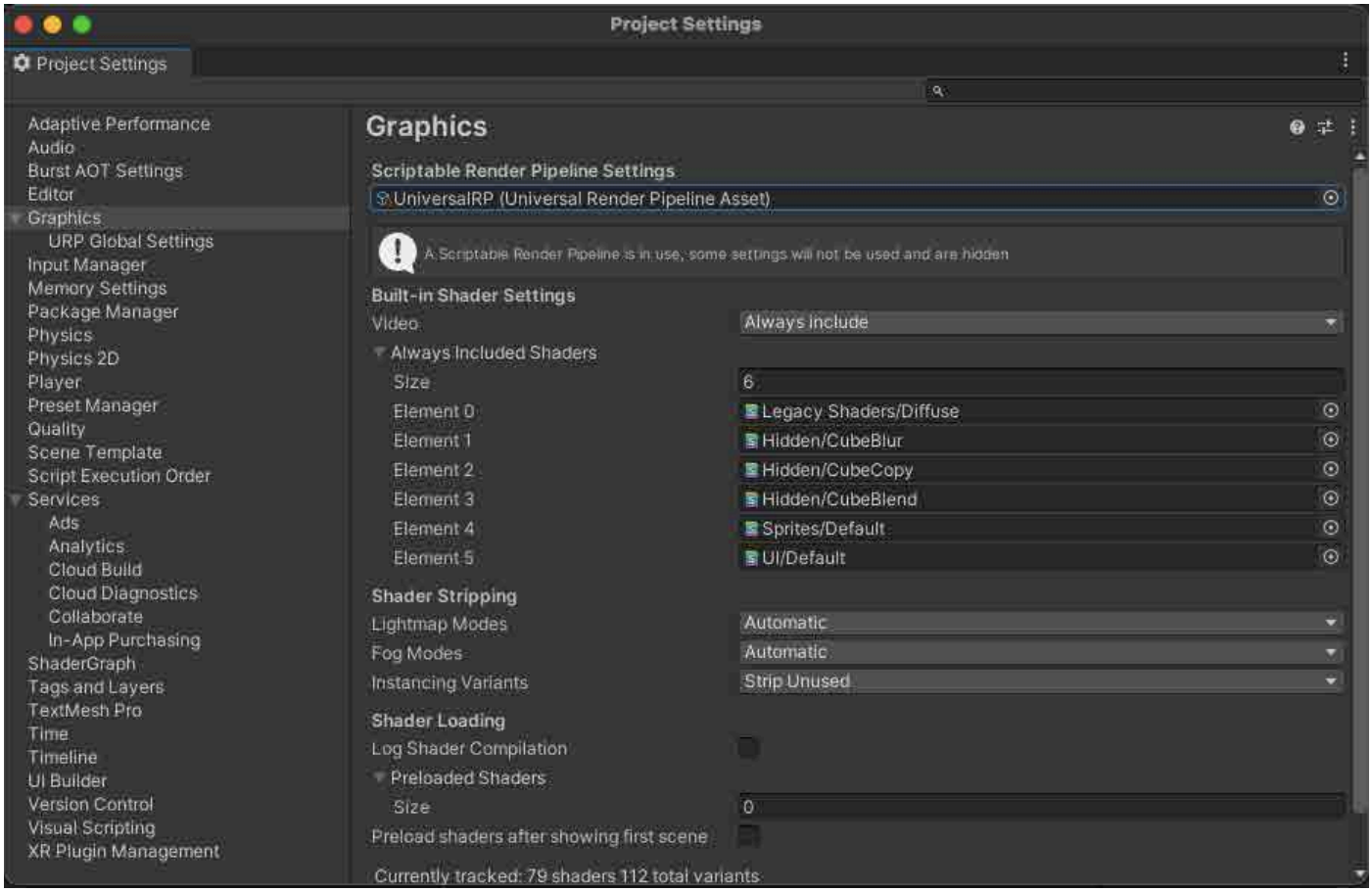
カメラやディレクショナルライトなどの基本ゲームオブジェクトを使用して、**「File」>「New Scene」**で新しいシーンを作成したり、あらかじめオブジェクトが配置された独自のシーンテンプレートを作成したりすることもできます。詳しくは URP Scene Templates ドキュメントをご覧ください。



シーンテンプレートを表示する新しいシーンのダイアログ

「Edit」 > 「Project Settings」 に移動し、「**Graphics**」パネルを開きます。エディター内で URP を使用するには、「**Scriptable Render Pipeline Settings**」から URP アセットを選択する必要があります。URP アセットは、プロジェクトのグローバルなレンダリング設定と品質設定を制御し、レンダーパイプラインのインスタンスを作成します。一方、レンダリングパイプラインのインスタンスには、中間リソースとレンダーパイプラインの実装が含まれています。

デスクトップで選択されるデフォルトの URP アセットは **PC_RPAsset** ですが、**Mobile_RPAsset** に切り替えることもできます。



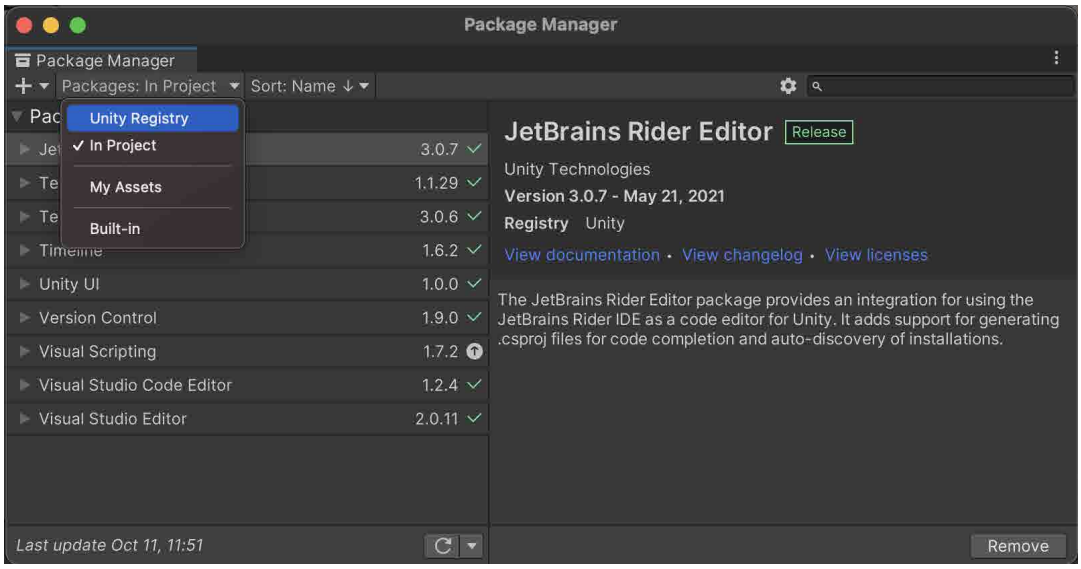
Project Settings の「Graphics」パネル

このガイドの後半のセクションでは、URP アセットの調整方法について詳しく説明しています。

既存のビルトインレンダーパイプラインプロジェクトに URP を追加する方法

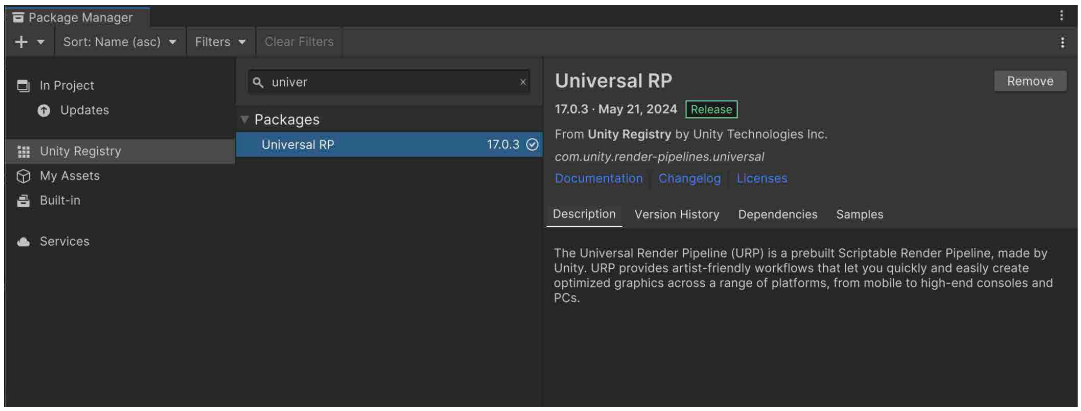
重要：このセクションの手順を実行する前に、ソースコントロールを使用してプロジェクトのバックアップを取るようにしましょう。このプロセスではアセットの変換を行いますが、Unity ではこの変更に対する取り消しオプションは提供されていません。ソースコントロールを使用していれば、必要に応じてアセットを変更前のバージョンに戻すことができます。

既存のビルトインレンダーパイプラインプロジェクトをアップグレードする場合、Unity 6 より前のバージョンには **URP パッケージ**が含まれていないため、プロジェクトに追加する必要があります。



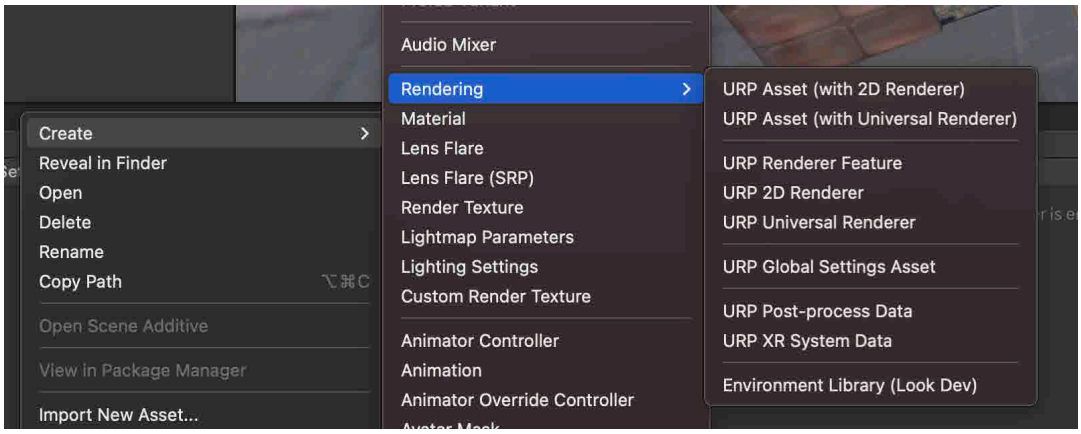
Package Manager に Unity Registry パッケージが表示されている

「Window」>「Package Manager」に移動し、**「Packages」**のドロップダウンをクリックして、URP をプロジェクトに追加します。**「Unity Registry」**を選択した後、**「Universal RP」**を選択します。URP パッケージが開発用コンピューターにインストールされていない場合、ウィンドウの右下隅にある**「Download」**をクリックします。ダウンロードが完了したら、**「Install」**をクリックします。



Package Manager を介して URP をインストール

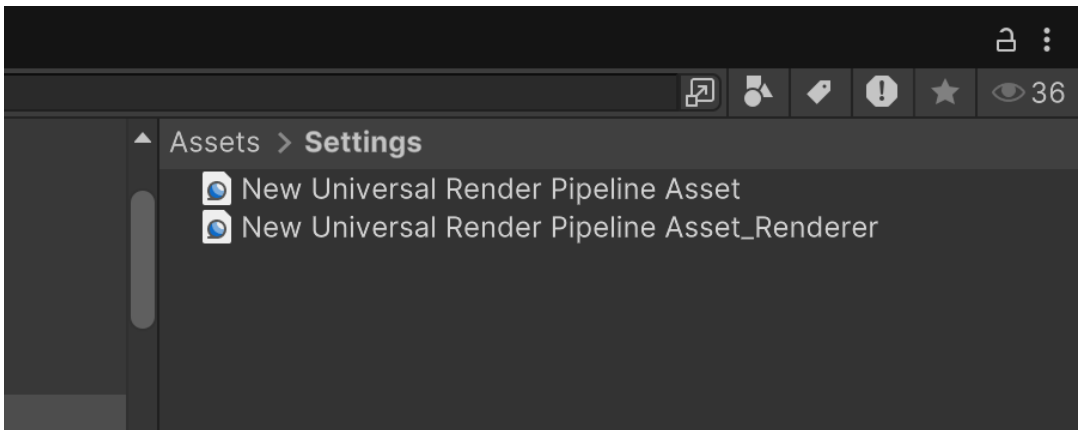
URP アセットを作成するには、**Project** ウィンドウを右クリックし、**「Create」 > 「Rendering」 > 「URP Asset (with Universal Renderer)」**を選択します。アセットに名前を付けます。



URP アセットの作成

覚えておくべきポイント：ユニバーサルレンダーパイプライン（URP）または 3D（URP）テンプレートを使用して新しいプロジェクトを作成した場合、これらの URP アセットはすでにプロジェクト内で利用可能な状態になっています。

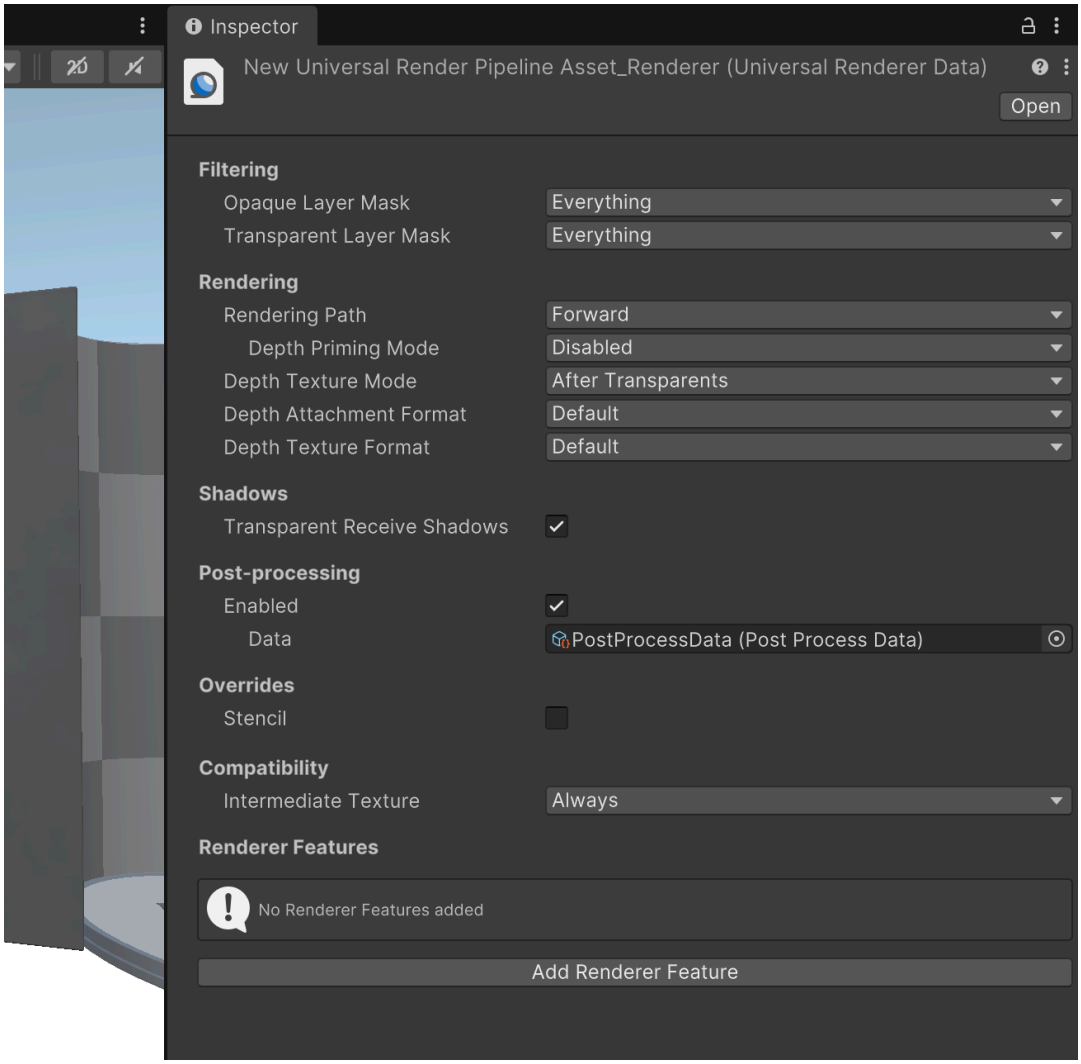
URP は、単一の URP アセットを作成するのではなく、アセット拡張子を持つ 2 つのファイルを使用します。



URP の 2 つのアセット。1 つは URP 設定用、もう 1 つはレンダラーデータ用

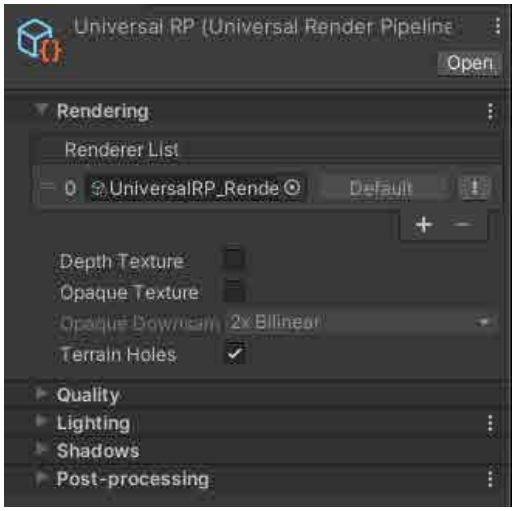
1 つは **UniversalRP_Renderer** と呼ばれる**レンダラーデータアセット**で、レンダラーが動作するレイヤーをフィルタリングしたり、レンダーパイプラインをインターセプトしてシーンのレンダリング方法をカスタマイズするために使用できます。これにより、高品質エフェクトの作成が容易になります。詳細については、パイプラインコールバックのセクションをご確認ください。

さらに、UniversalRP_Renderer は、URP のハイレベルなレンダーロジックとパスを制御します。フォワードとディファードパス、そして 2D ライト、2D シャドウ、ライト ブレンド スタイルなどの機能を使用できる 2D レンダラーをサポートしています。URP を拡張して、独自のレンダラーを作成することもできます。



UniversalRP_Renderer データアセットの Inspector

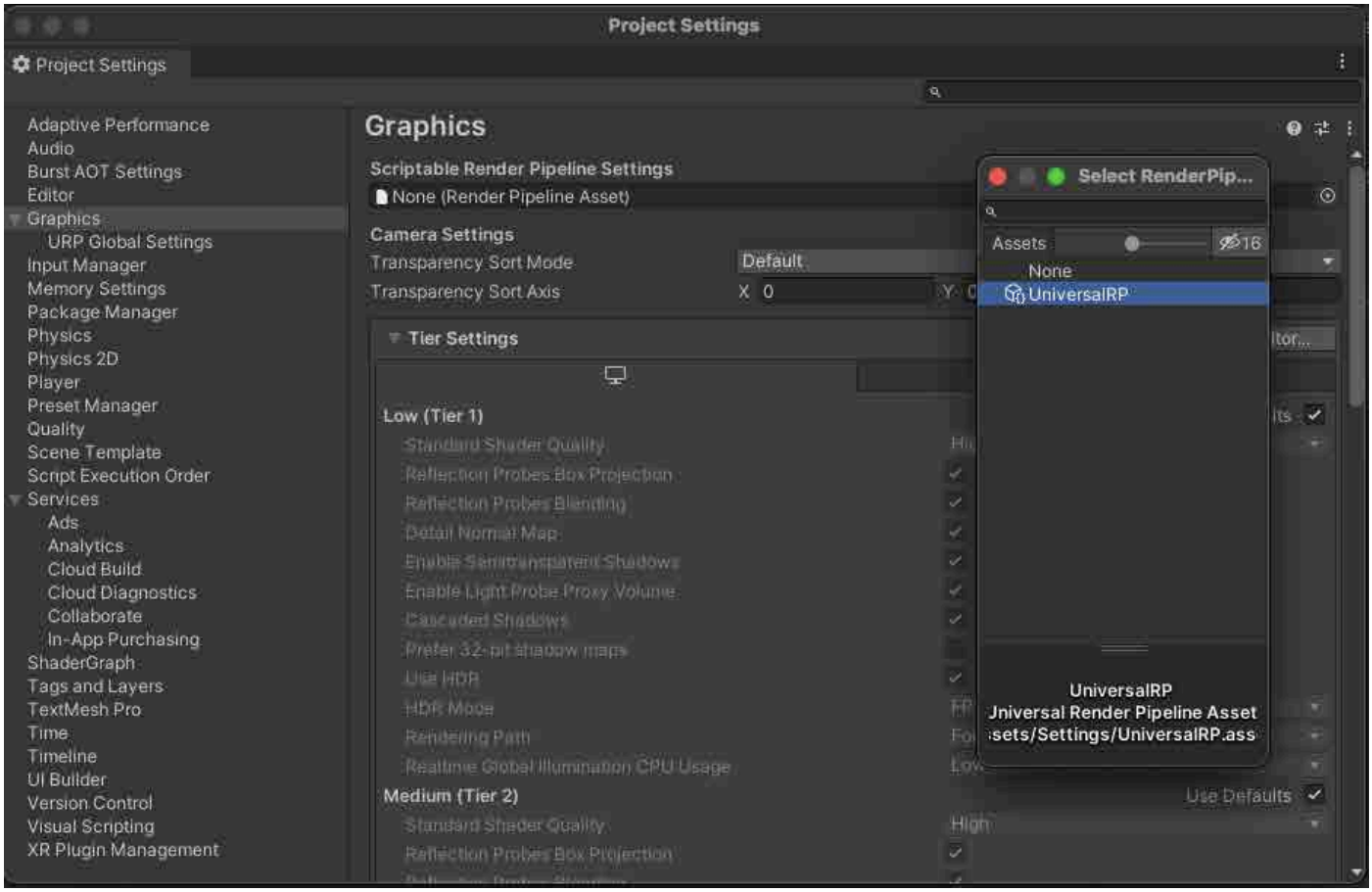
もう 1 つの URP アセットは、品質、ライティング、影、ポストプロセス設定を制御するオプションを提供します。このセクションの後半でプロセスを説明していますが、異なる URP アセットを使用して品質設定を制御できます。この設定アセットは、レンダラーリストを介してレンダラーデータアセットにリンクされています。新しい URP アセットを作成すると、設定アセット内に、1 つのアイテム（同時に作成されるレンダラーデータアセットで、デフォルトとして設定される）を含むレンダラーリストが作成されます。このリストには、代わりのレンダラーデータアセットを追加することができます。



Inspector 内の URP アセット

デフォルトのレンダラーは、シーンビューを含むすべてのカメラに使用されます。カメラは、レンダラーリストから別のレンダラーを選択することで、デフォルトのレンダラーをオーバーライドできます。必要に応じて、スクリプトを使ってこの操作を行うことも可能です。

これらのステップに従って URP アセットを作成しても、シーンビューやゲームビューで開いているシーンはまだビルトインレンダーパイプラインを使用しています。URP に切り替えるには、最後のステップを完了する必要があります。**「Edit」>「Project Settings」**に移動し、**「Graphics」**パネルを開きます。**「None (Render Pipeline Asset)」**の横にある小さいドットをクリックします。開いたパネル上で**「UniversalRP」**を選択します。



スクリプタブルレンダーパイプラインアセットを選択

切り替えに関する警告メッセージが表示されるので、「**Continue**」を押します。

プロジェクト内にまだコンテンツがないため、レンダーパイプラインの変更はほぼ即座に完了します。これで URP を使用する準備が整いました。

既存プロジェクトのシーンの変換

上記のステップを完了すると、作成したシーンが全てマゼンタ色で表示されるようになってしまいます。これは、ビルトインレンダーパイプラインプロジェクトのマテリアルで使用されているシェーダーが、URP でサポートされていないためです。シーンの品質を元に戻す方法はありますので、安心してください。

シーン内のマテリアルがマゼンタ色で表示されるのは、ビルトインレンダーパイプラインベースのシェーダーを URP で使用できるように変換する必要があるためです。

「**Window**」>「**Rendering**」>「**Render Pipeline Converter**」の順に選択します。2D プロジェクトの場合は「**Convert Built-In to 2D (URP)**」、3D プロジェクトの場合は「**Built-In to URP**」を選択します。プロジェクトが 3D の場合、次の中から適切なコンバータを選択する必要があります。

- **Rendering Settings：**これを選択すると、ビルトインレンダーパイプラインの品質設定にできる限り近いレンダーパイプラインの設定アセットが複数作成されます。これにより、より効率的に異なる品質レベルをテストすることができます。詳細については、ビルトインレンダーパイプラインと URP 品質オプションの比較セクションをご確認ください。
- **Material Upgrade：**これを使用すると、マテリアルをビルトインレンダーパイプラインから URP に変換できます。
- **Animation Clip Converter:**これはアニメーションクリップを変換します。Material Upgrade コンバータの終了後に実行されます。
- **Read-only Material Converter：**これは、Unity 内に含まれる読み取り専用のビルド済みマテリアルを変換します。プロジェクトのインデックスを作成して、一時的な .index ファイルを作成します。処理の完了までに長時間かかる可能性があることに注意してください。

カスタムシェーダーの変換

カスタムシェーダーは、Material Upgrade コンバータを使用しても変換されません。シェーダーと Shader Graph のセクションでは、カスタムビルトインレンダーパイプラインシェーダーを URP に変換するステップを概説しています。多くの場合、Shader Graph の使用は、カスタムシェーダーを URP にアップデートするための最も手早い方法です。

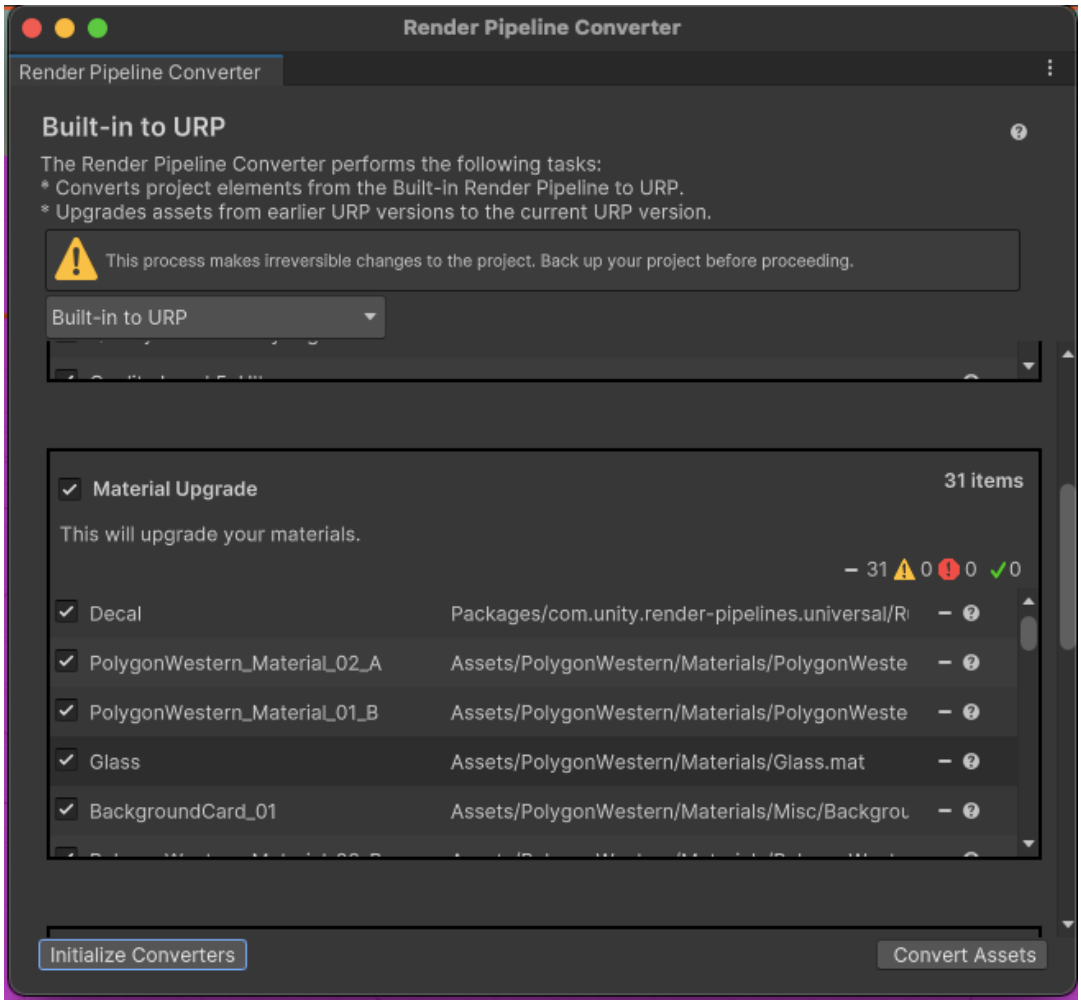
URP シェーダーには、いくつかの異なる種類があります。

- **Universal Render Pipeline/Lit**：この物理ベースのレンダー（PBR）シェーダーは、ビルトインスタンダードシェーダーに類似しており、ほとんどの現実世界のマテリアルを表現できます。メタリックとスペキュラーの両方のワークフローで、スタンダードシェーダーのすべての機能をサポートしています。
- **Universal Render Pipeline/Simple Lit**：Blinn-Phong モデルを使用しており、ローエンドモバイルデバイスや PBR ワークフローを使用していないゲームに最適です。
- **Universal Render Pipeline/Baked Lit**：様式化されたゲームや、ライトマップ、ライトプローブ、アダプティブ プローブ ボリューム（APV）を使ったベイクドライティングのみが必要なアプリに使用します。このシェーダーは物理ベースではなく、リアルタイムライティングがないため、リアルタイム関連のシェーダーキーワードやバリアントがすべてコードから削除されており、計算速度が向上しています。
- **Universal Render Pipeline/Complex Lit**：Complex Lit シェーダーには、Lit シェーダーのすべての機能が含まれ、高度なマテリアル機能が追加されています。このシェーダーの一部機能はリソース消費が多く、Unity Shader Model 4.5 ハードウェアを必要とする場合があります。
- **Universal Render Pipeline/Unlit**：ライティング方程式を使用しない、GPU 効率の高いシェーダーです。
- **Universal Render Pipeline/Terrain/Lit**：これは、Terrain Tools パッケージと共に使用するのが最適です。
- **Universal Render Pipeline/Particles/Lit**：このパーティクルシェーダーは、PBR ライティングモデルを使用しています。
- **Universal Render Pipeline/Particles/Unlit**：この Unlit パーティクルシェーダーは、GPU 負荷が少ないシェーダーです。

Simple Lit は多くの旧式 / モバイルシェーダーの代替として機能しますが、パフォーマンスは同じではありません。旧式 / モバイルシェーダーはライティングを部分的にしか評価しませんが、Simple Lit は URP アセットで定義されたすべてのライトを考慮します。

URP ドキュメント内のこの表を参照することで、各 URP シェーダーがビルトインレンダーパイプラインのどのシェーダーに対応しているかを確認できます。

上記のコンバータを 1 つ以上選択した後、「**Initialize Converters**」または「**Initialize And Convert**」をクリックします。どちらのオプションを選択しても、プロジェクトがスキャンされ、変換が必要なアセットが各コンバータパネルに追加されます。「**Initialize Converters**」を選択すると、各項目に用意されているチェックボックスを使用して項目の選択を解除することで変換を制限できます。ここで「**Convert Assets**」をクリックすると、変換プロセスが開始されます。「**Initialize And Convert**」を選択すると、コンバータの初期化後、変換が自動的に開始されます。完了した後、エディター内でアクティブなシーンを再度開くように求められる場合があります。



レンダーパイプラインコンバータ

ビルトインレンダーパイプラインと URP の品質オプションの比較

ビルトインレンダーパイプラインには、Very low から Ultra まで、いくつかのデフォルトの品質オプションがあります。品質設定は、テクスチャの解像度、ライティング、シャドウレンダリングなど、シーンの忠実度に影響を与えます。

「Edit」 > 「Project Settings」 に移動し、**「Quality」** パネルを選択します。ここでは、現在の品質を選択することで、品質オプションを切り替えられます。これにより、シーンとゲームビューで使用されるレンダー設定が変更されます。また、このパネルで各品質オプションを編集することも可能です。

レンダーパイプラインコンバータを使用してビルトインレンダーパイプラインから URP に切り替える際、**「Rendering Settings」** オプションを選択すると、ビルトインレンダーパイプラインの品質オプションに近付けた URP アセットのセットが作成されます。以下の最初の表は、ビルトインレンダーパイプラインと URP の Low 設定での対応を示し、2 つ目の表は High 設定での比較を示しています。ビルトインレンダーパイプラインと URP のどちらの場合も、設定は「Quality」パネルから行います。URP アセットを選択すると、その設定に Inspector からアクセスできます。詳細については URP ドキュメントを参照してください。

ビルトインレンダーパイプラインと URP の対応表：Low 設定

設定	ビルトインレンダーパイプライン	URP	URP アセットの設定
Rendering			
Pixel Light Count	0	該当なし（N/A）*	NA
アンチエイリアス	Disabled	NA	Disabled
Render Scale	NA	NA	1
リアルタイムリフレクションプローブ	なし	なし	NA
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Don’t sync	Don’t sync	NA
Depth Texture	NA	NA	いいえ
Opaque Texture	NA	NA	いいえ
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	はい
HDR	NA	NA	はい
Textures			
Texture Quality	Half res	Half res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	いいえ	いいえ	NA
Particles			
Soft Particles	いいえ	NA	NA
Particle Raycast Budget	16	16	NA
Terrain			
Billboards Face Camera Position	いいえ	いいえ	NA

Shadows			
Shadowmask Mode	Shadowmask	Shadowmask	NA
Shadows	Disabled	NA	NA
Shadow Resolution	Low resolution	NA	NA
Shadow Projection	Stable fit	NA	NA
Shadow Distance	20	NA	NA
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	No Cascades	NA	NA
カスケードの分割	NA	NA	NA
Working unit	NA	NA	NA
Depth Bias	NA	NA	NA
Normal Bias	NA	NA	NA
ソフトシャドウ	NA	NA	NA
Async Asset Upload			
Time Slice	2	2	NA
バッファサイズ	16	16	NA
永続バッファ	はい	はい	NA
Level of Detail			
LOD バイアス	0.4	0.4	NA
最大 LOD レベル	0	0	NA
メッシュ			
スキンウェイト	4 ボーン	4 ボーン	NA
Lighting			
メインライト	NA	NA	Per pixel
Cast Shadows	NA	NA	いいえ
Shadow Resolution	NA	NA	NA
追加ライト	NA	NA	Disabled
Per Object Limit	NA	NA	NA
Cast Shadows	NA	NA	NA
Shadow Atlas Resolution	NA	NA	NA
Shadow Resolutiontiers	NA	NA	NA
Cookie AtlasResolution	NA	NA	NA
Cookie AtlasFormat	NA	NA	NA
Reflection probes:	NA	NA	NA

Probe Blending	NA	NA	いいえ
Box Projection	NA	NA	いいえ
Post-processing			
Grading Mode	NA	NA	Low Dynamic Range
	NA	NA	16
Fast sRGB/Linear conversion	NA	NA	いいえ

* URP では、Pixel Light Count は**「Additional Lights」>「(Per pixel)」>「Per Object Limit」**で設定されます。

ビルトインレンダーパイプラインと URP の対応表：High 設定

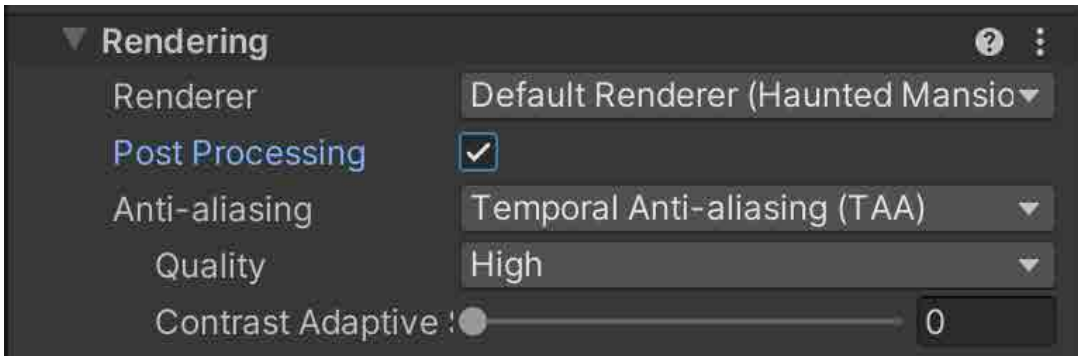
設定	ビルトインレンダーパイプライン	URP	URP アセットの設定
Rendering			
Pixel Light Count	2	該当なし（N/A）	NA
アンチエイリアス	Disabled	NA	2x
Render Scale	NA	NA	1
リアルタイムリフレクションプローブ	はい	はい	NA
Resolution Scaling Fixed DPI Factor	1	1	NA
VSync Count	Every V Blank	Every V Blank	NA
Depth Texture	NA	NA	いいえ
Opaque Texture	NA	NA	いいえ
Opaque Downsampling	NA	NA	NA
Terrain Holes	NA	NA	はい
HDR	NA	NA	はい
Textures			
Texture Quality	Full res	Full res	NA
Anisotropic Textures	Disabled	Disabled	NA
Texture Streaming	いいえ	いいえ	NA
Particles			
Soft Particles	いいえ	NA	NA
Particle Raycast Budget	256	256	NA

Terrain			
Billboards Face Camera Position	はい	はい	NA
Shadows			
Shadowmask Mode	Distance Shadowmask	Distance Shadowmask	NA
Shadows	Hard and Soft Shadows	NA	NA
Shadow Resolution	Medium resolution	NA	2048
Shadow Projection	Stable fit	NA	NA
Shadow Distance	40	NA	50
Shadow Near Plane Offset	3	NA	NA
Shadow Cascades	2 Cascades	NA	2
カスケードの分割	33/67	NA	12.5/33.8/3.8
Working unit	Percent	Percent	Metric
Depth Bias	NA	NA	1
Normal Bias	NA	NA	1
ソフトシャドウ	NA	NA	はい
Async Asset Upload			
Time Slice	2	2	NA
バッファサイズ	16	16	NA
永続バッファ	はい	はい	NA
Level of Detail			
LOD バイアス	1	1	NA
最大 LOD レベル	0	0	NA
メッシュ			
スキンウェイト	Unlimited	Unlimited	NA
Lighting			
メインライト	NA	NA	Per pixel
Cast Shadows	NA	NA	はい
Shadow Resolution	NA	NA	NA
追加ライト	NA	NA	Per pixel
Per Object Limit	NA	NA	4
Cast Shadows	NA	NA	はい
Shadow Atlas Resolution	NA	NA	2048
Shadow Resolution tiers	NA	NA	512/1024/2048
Cookie AtlasResolution	NA	NA	2048

Cookie AtlasFormat	NA	NA	Color high
Reflection probes:	NA	NA	NA
Probe Blending	NA	NA	はい
Box Projection	NA	NA	いいえ
Post-processing			
Grading Mode	NA	NA	ローダイナミックレンジ
LUT サイズ	NA	NA	32
Fast sRGB/Linear conversion	NA	NA	いいえ

アンチエイリアス

Unity 6 の URP では、**「Camera」 > 「Rendering」 > 「Anti-aliasing」**で、カメラのアンチエイリアスオプションとして Temporal Anti-aliasing (TAA) を選択できます。



TAA が選択されている

Unity 6 の URP は、パフォーマンスに影響を与えることなく、TAA の全体的な品質を向上させます。エッジのアンチエイリアスが強化され（以前の実装で見られた奇妙なエッジのアーティファクトを除去）、テクスチャの品質保持も改善しています。出力品質は低および中の SMAA プリセットと同等ですが、パフォーマンスはさらに向上しています。

品質設定の使用方法

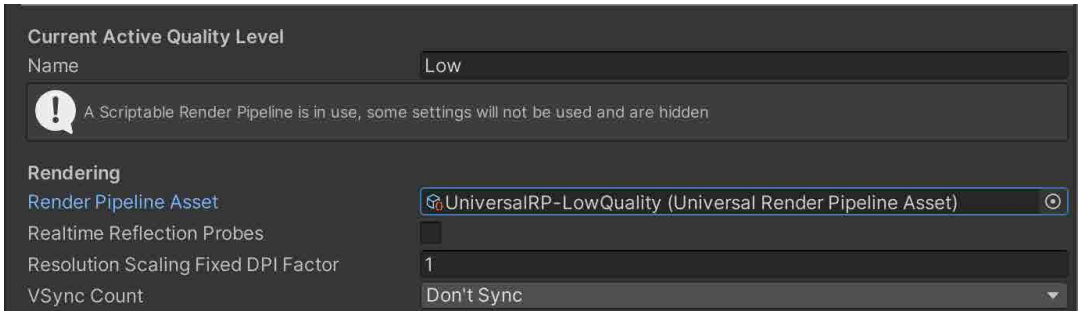
URP を使用する場合、品質設定は「Quality」パネルと各 URP アセットの設定に分かれています。次の表は、各設定の場所を示しています。

URP 使用時の品質設定

設定	「Quality」パネル	URP アセット
Rendering		
アンチエイリアス		✅
Render Scale		✅
Resolution Scaling Fixed DPI Factor	✅	
VSync Count	✅	
Depth Texture		✅
Opaque Texture		✅
Opaque Downsampling		✅
Terrain Holes		✅
HDR		✅
Textures		
Texture Quality	✅	
Anisotropic Textures	✅	
Texture Streaming	✅	
Particles		
Particle Raycast Budget	✅	
Terrain		
Billboards Face Camera Position	✅	
Shadows		
Shadowmask Mode	✅	
Shadow Resolution		✅
Shadow Distance		✅
Shadow Cascades		✅
Cascade splits		✅
Working unit		✅
Depth Bias		✅
Normal Bias		✅
Soft Shadows		✅

Async Asset Upload		
Time Slice	✓	
Buffer Size	✓	
Persistent Buffer	✓	
Level of Detail		
LOD Bias	✓	
Maximum LOD level	✓	
Meshes		
Skin Weights	✓	
Lighting		
Main Light:		✓
— Cast Shadows		✓
— Shadow Resolution		✓
Additional Lights:		✓
— Per Object Limit		✓
— Cast Shadows		✓
— Shadow Atlas Resolution		✓
— Shadow Resolutiontiers		✓
— Cookie AtlasResolution		✓
— Cookie AtlasFormat		✓
Reflection probes:	✓	
— Probe Blending		✓
— Box Projection		✓
Post-processing		
Grading Mode		✓
LUT size		✓
Fast sRGB/Linear conversion		✓

品質オプションを切り替える場合は「**Project Settings**」から「**Quality**」パネルでレンダーパイプラインアセットの「**品質レベル**」を選択します。品質レベルが設定されていない場合、「Graphics」パネルでスクリプタブルレンダーパイプラインアセットとして設定されているレンダーパイプラインアセットがデフォルトで使用されるため、ご注意ください。これにより、URP アセットの品質設定を調整する際に混乱が生じる場合があります。例えば、URP アセットで設定した品質レベルが、シーンビューとゲームビューで現在使用されているものだと誤解してしまう可能性があります。

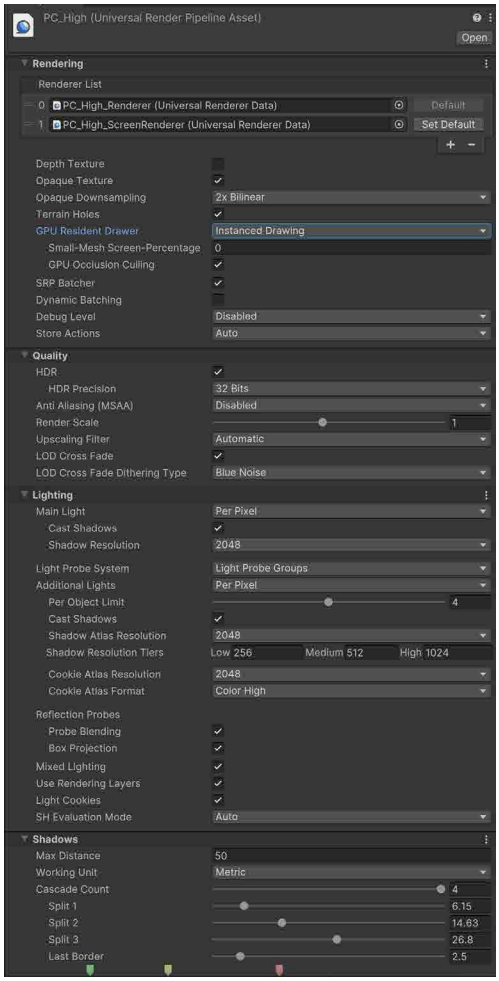


レンダーパイプラインアセットの品質レベルの設定

URP アセットの変更

以下の画像は、Inspector 内で URP アセットのすべての利用可能な設定を示したものです。各設定の詳細については、URP のドキュメントを参照してください。

注：URP 2D レンダラーを有効にしている場合、URP アセット内の 3D レンダリングに関連するオプションの一部は、最終的なアプリやゲームに影響を与えません。2D レンダラーアセットは、**「Edit」>「Project Settings」>「Graphics」**の**「Scriptable Render Pipeline Settings」**でアクセス可能です。



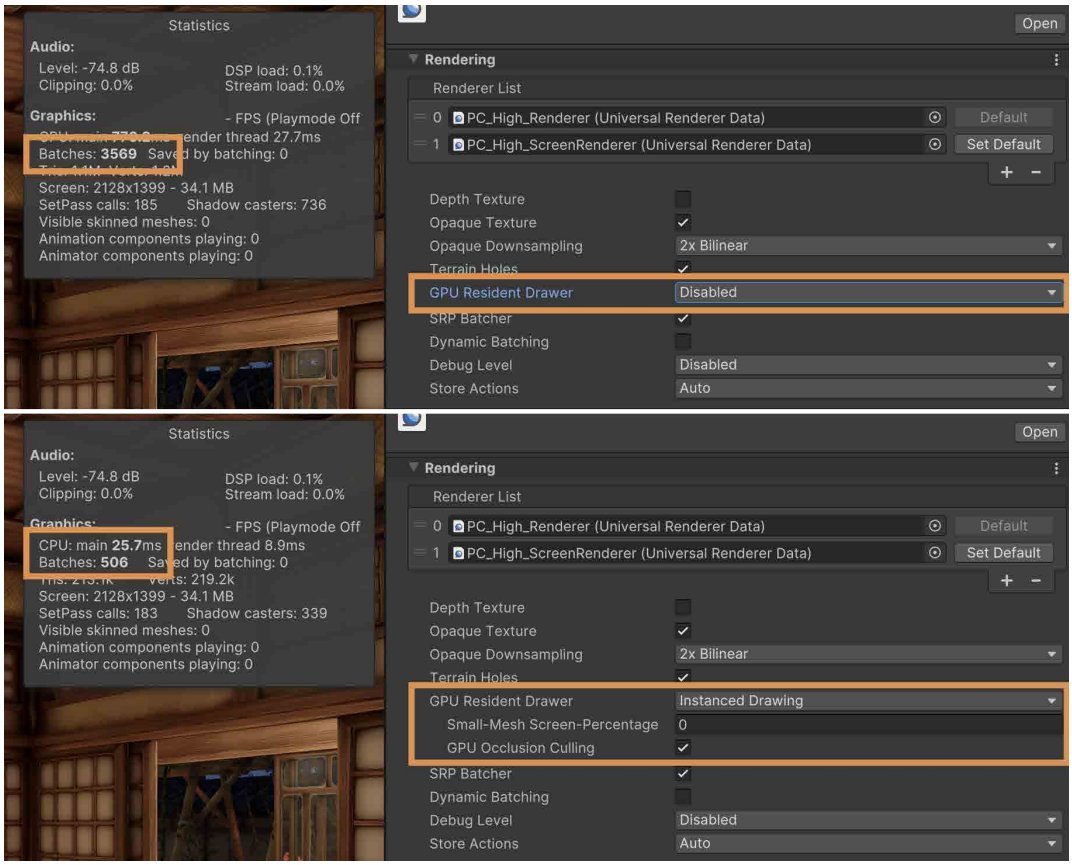
Inspector 内の URP アセット

URP アセットの「Quality」パネルでは、忠実度を高めるために HDR フォーマットを 64 ビットに設定できます。ただし、これによりパフォーマンスが低下し、追加のメモリも必要となるため、ローエンドハードウェアではこの設定を避けてください。

「Quality」パネルのもう 1 つの機能は、「LOD Cross Fade」を有効化するオプションです。LOD は、遠くにあるメッシュのレンダリングに要する GPU 負荷を減らす技法です。カメラの移動に伴い、LOD が切り替わり、適切な品質レベルを提供します。LOD Cross Fade により、異なる LOD ジオメトリ間の切り替えがスムーズになり、スワップ時に発生する急なスナップやポッピングを防ぎます。

GPU Resident Drawer と GPU オクルージョンカリング

GPU Resident Drawer は Unity 6 の新機能で、URP アセットの「**Rendering**」セクションから利用できます。



Unity 6 の URP アセットで利用できる GPU Resident Drawer と GPU オクルージョンカリングのオプション

上のスクリーンショットから、エディターモードで URP 3D サンプルシーンの庭をレンダリングするために必要なバッチ数が 3569 であることがわかります。GPU Resident Drawer を「**Instanced Drawing**」に設定すると、バッチ数は 506 まで減少します。

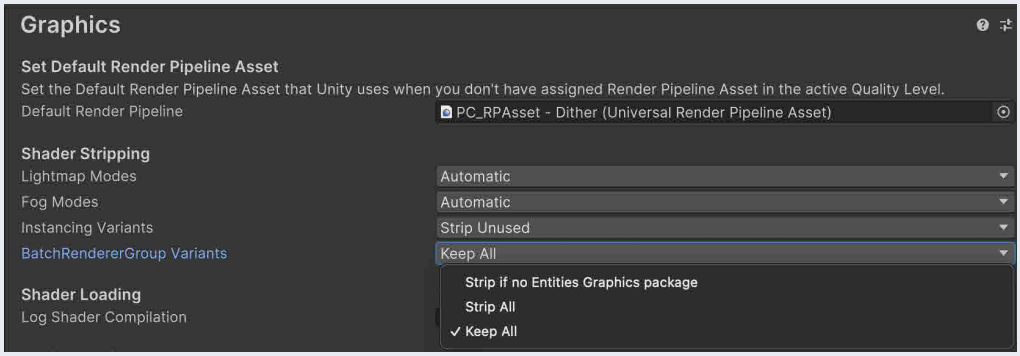
GPU Resident Drawer は、CPU 時間を最適化するために設計された GPU 駆動のレンダリングシステムです。これにより、ゲームオブジェクトが BatchRenderGroup API を活用でき、高速なバッチ処理と CPU パフォーマンスの向上が期待できます。

GPU Resident Drawer を使用すると、ゲームオブジェクトを使ってゲームを作成し、処理時により効率的なインスタンシングを行う特別な高速パスでレンダリングされます。この機能を有効にすると、多数のドローコールが発生する GPU 依存ゲームでは、ドローコールの数が減少するため、このボトルネックが緩和されます。

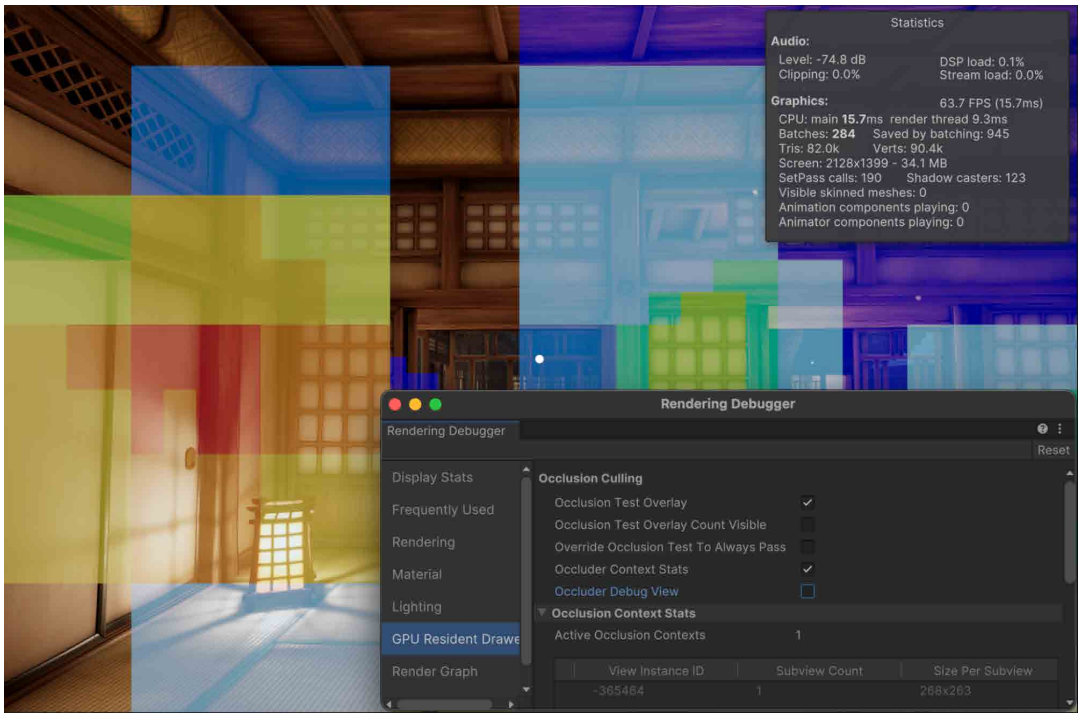
どの程度の改善が見られるかは、シーンの規模やインスタンシングの量によって異なります。レンダリングするインスタンス化可能なオブジェクトの数が多いほど、メリットが大きくなります。

GPU Resident Drawer は MeshRenderer を対象としており、スキンメッシュレンダラー、VFX Graph、パーティクルシステム、または同様のエフェクトレンダラーには対応していません。この機能を利用するために、既存のコンテンツを変更する必要はありません。

注：GPU Resident Drawer を使用するには、フォワード + レンダラーが必要で、**「Project Settings」> 「Graphics」> 「BatchRendererGroup Variants」**を**「Keep All」**に設定する必要があります。



GPU Resident Drawer を有効にすると、GPU オクルージョンカリングもオプションとして利用可能になります。これは GPU 駆動のアプローチを用いて、画面に表示されないものをレンダリングしないようにします。コンテンツによっては、CPU の負荷を大幅に削減できる場合があります。



レンダリングデバッガーを使ったオクルージョンテストの確認

シーンで GPU オクルージョンカリングが有効かどうかを確認するには、**「Window」> 「Analysis」> 「Rendering Debugger」**に進み、**「GPU Resident Drawer」> 「Occlusion Test Overlay」**を選択します。これにより、カリングされたインスタンスのヒートマップが表示されます。ヒートマップは、カリングされたインスタンスが少ない場合は青色で、数が多くなるにつれて赤色へと変化して表示されます。この設定を有効にすると、カリングが遅くなることがあります。

URP のライティング

このセクションでは、Unity 6 の URP におけるライティングの仕組みと、グラフィックスの忠実度とパフォーマンスのバランスを取るために使用できるテクニックを説明します。

Unity のライティングに触れるのが初めての場合、以下のリソースを活用してください。

- ライティングに関するドキュメント
- ゲーム環境でのライティング技術
- Real-time lighting in Unity
- URP と GPU ライトマッパーによるライトの活用

レンダラーの選択

URP にはさまざまなレンダリング手法があり、それぞれ長所と短所があります。どのレンダリング手法を選ぶかは、プロジェクトの具体的な要件と制約によります。URP のフォワード、フォワード＋、ディファードレンダリングの違いを詳しく見ていきましょう。

フォワードレンダリング

仕組み	長所	短所
フォワードレンダリングは、シーン内の各オブジェクトを個別にレンダリングし、各ピクセルを別々に計算する従来のレンダリング手法です。つまり、画面上の各ピクセルに対して、そのピクセルの最終的な色に影響を与えるシーン内の各オブジェクトのライティングとシェーディングを Unity が計算します。	このワークフローは比較的理解しやすいものです。ライトの数が少なく、マテリアルがシンプルなシーンでは効果的に動作します。	多数のライトや複雑なマテリアルを持つシーンのレンダリングには非効率的です。なぜなら、各ライトに対してシーンを複数回処理する必要があり、レンダリングのオーバーヘッドが増加するからです。

フォワード + レンダリング

仕組み	長所	短所
フォワード + レンダリングは、特に多数のライトを扱う場合に、フォワードレンダリングの制限を解決する拡張機能です。フォワード + レンダリングでは、ライトがクラスターごとにグループ化され、各クラスター内のオブジェクトだけがライティングの計算に含まれるため、シーン内のすべてのオブジェクトを各ライトごとに処理する必要がありません。	フォワードレンダリングと比較してパフォーマンスが向上し、特に多数のライトを使用するシーンで効果的です。各ライトで考慮すべきオブジェクトの数を減らすことで、ハードウェアリソースをより効率的に利用できます。	しかし、非常に多くのライトがあるシーンや複雑なシーンでは問題が発生する可能性があります。また、フォワード + レンダリングを実装するには、フォワードレンダリングよりも多くの労力と最適化が必要になる場合があります。

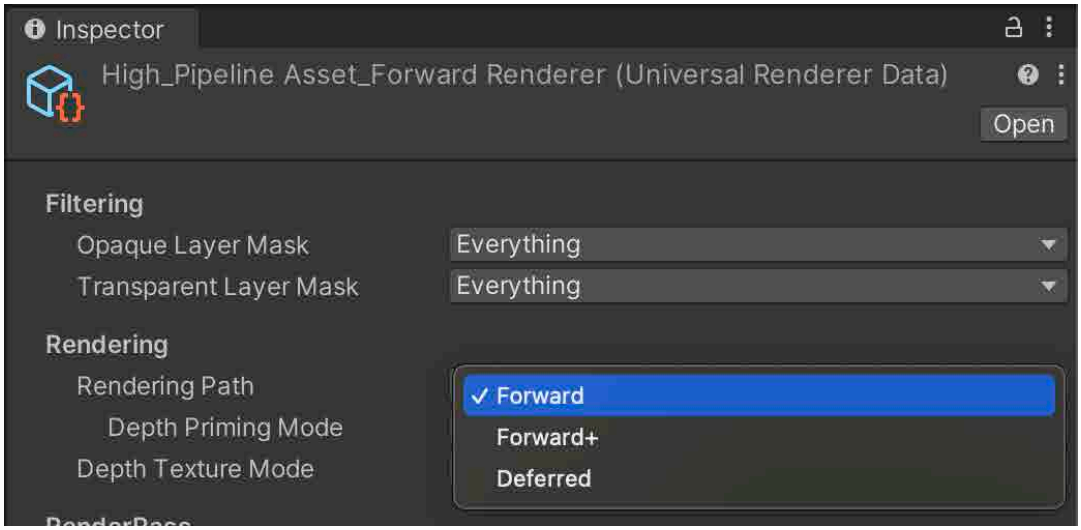
URP ディファードレンダリング

仕組み	長所	短所
ディファードレンダリングは、ライティングとシェーディングの計算をジオメトリのレンダリングプロセスから切り離します。ジオメトリはまず、各ピクセルの位置、法線、マテリアルプロパティの情報を格納するバッファ（G バッファなど）にレンダリングされます。その後、これらのバッファに保存された情報に基づいて、ピクセルごとにライティング計算が行われます。	ライティング計算がオブジェクト単位ではなくピクセル単位で行われるため、多数のライトや複雑なマテリアルを持つシーンにおいては非常に効率的です。これにより、パフォーマンスへの影響を最小限に抑えながら、多数のライトをレンダリングできます。	ディファードレンダリングの課題としては、追加のバッファを保存する必要があるためメモリ使用量が増加する点、透明なオブジェクトに制限がある点、ボリュメトリックライティングなど特定のライティング効果の処理が難しくなる点などが挙げられます。

以下の表は、これら 3 つのレンダリングオプションの詳細を示しています。

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 バッファで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
GPU Resident Drawer	いいえ	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、ベースカメラのみをディファードパスでレンダリングし、すべてのオーバーレイカメラはフォワードレンダリングパスでレンダリングします。

レンダリングパスを切り替えるには、ユニバーサルレンダラーデータアセットを使用します。



レンダリングパスの選択

フォワード + を使用する場合、多くの URP アセットのライティング設定がオーバーライドされます。

— **Main light**：このプロパティの値は、選択した値に関係なく、**Per Pixel** となります。

— **Additional lights**：このプロパティの値は、選択した値に関係なく、**Per Pixel** となります。

— **Additional Lights > Per Object Limit**：Unity はこのプロパティを無視します。

— **Reflection Probes > Probe Blending**：リフレクションプローブのブレンディングは常にオンです。

ライト設定

ライトのプロパティは、以下の 3 か所で設定します。

1. **Window > Rendering > Lighting：**このパネルでは、ライトマッピングや環境の設定を行い、リアルタイムおよびベイクしたライトマップを確認できます。ビルトインレンダーパイプラインから URP に変更しても変わりはありません。
2. **Light Inspector:**ビルトインレンダーパイプラインと URP の Inspector には大きな違いがあります。詳細については、Light Inspector のセクションを参照してください。
3. **URP アセットの Inspector：**これは、主に影の設定を行う場所です。URP のライティングは、このパネルで選択した設定に大きく依存します。

ビルトインレンダーパイプラインの場合、品質の設定は**「Edit」>「Project Settings」>「Quality」**で行います。URP では、URP アセット設定に依存し、「**Quality**」パネルを使って変更することができます（品質設定のセクションを参照）。

ここでの焦点はライティングであるため、この手法は、次の表にあるシェーダーを使用するマテリアルに適用されます。

ライトに照らされたシーンの URP シェーダー

シェーダー	説明
Complex Lit	このシェーダーには、Lit シェーダーのすべての機能が備わっています。たとえば、車にメタリックな光沢を持たせるために「Clear Coat」オプションを使用する際は、これを選択してください。鏡面反射は 2 回計算されます。1 回はベースレイヤーに対して、もう 1 回はベースレイヤーの上の透明な薄いレイヤーのシミュレーションを行うために実行されます。
Lit	Lit シェーダーを使用すると、石、木、ガラス、プラスチック、金属など、現実世界に存在する物の表面を写実的な品質でレンダリングできます。光量や反射は実物のように見え、眩しい日差しから暗澹とした洞窟まで、さまざまなライティング条件に対して反応します。 これは、ライティングを使用するほとんどのマテリアルでデフォルトの選択肢となります。ベイク、混合、リアルタイムのライティングをサポートし、フォワードまたはディファードレンダリングでも機能します。 これは物理ベースシェーディング（PBS）モデルです。シェーディングの計算は複雑なので、このシェーダーはローエンドのモバイルハードウェアでは使用しないようにすることをお勧めします。
Simple Lit	このシェーダーは物理ベースではありません。エネルギー保存則を満たさない Blinn-Phong シェーディングモデルを使用し、結果の写実性はやや控えめになります。それでもなお、優れたビジュアルを実現できます。ローエンドのモバイルデバイスをターゲットにする場合で、物理ベースではないプロジェクトで使用するのに適しています。
Baked Lit	このシェーダーは、リアルタイムライトが不要なオブジェクトのパフォーマンスを向上させます（動的オブジェクト、リアルタイムライト、動的シャドウなどの影響を受けない遠方の静的オブジェクトなどが対象）。

Lit か Simple Lit か?

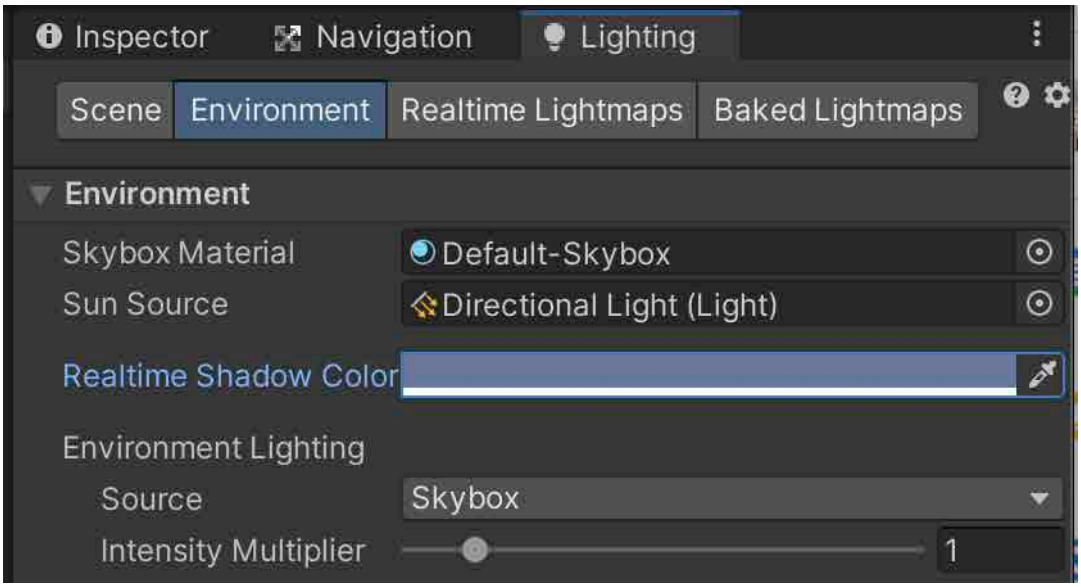
Lit シェーダーと Simple Lit シェーダーのどちらを選ぶかは、主にアートの観点から決定されます。アーティストがリアルなレンダリングを行うには Lit シェーダーの方が簡単ですが、より様式化されたレンダリングを望む場合は Simple Lit を使用する方がより良い結果を得られます。

異なるシェーダーを使用してレンダリングされたシーンの比較：左上の画像は Lit シェーダーを、右上の画像は Simple Lit シェーダーを、下の画像は Baked Lit シェーダーを使用したものです。

カスタムシェーダーを書くか、Shader Graph を使用することで、独自のライティングモデルを実装することが可能です（追加ツールの章を参照）。

ライティングの概要

URP では、ライトは Main Light と Additional Lights に分けられます。Main Light プロパティの設定は、Directional Light に影響します。これに該当するのは、最も明るいライト、もしくは**「Window」>「Rendering」>「Lighting」>「Environment」>「Sun Source」**で設定されたライトになります。



「Sun Source」の設定

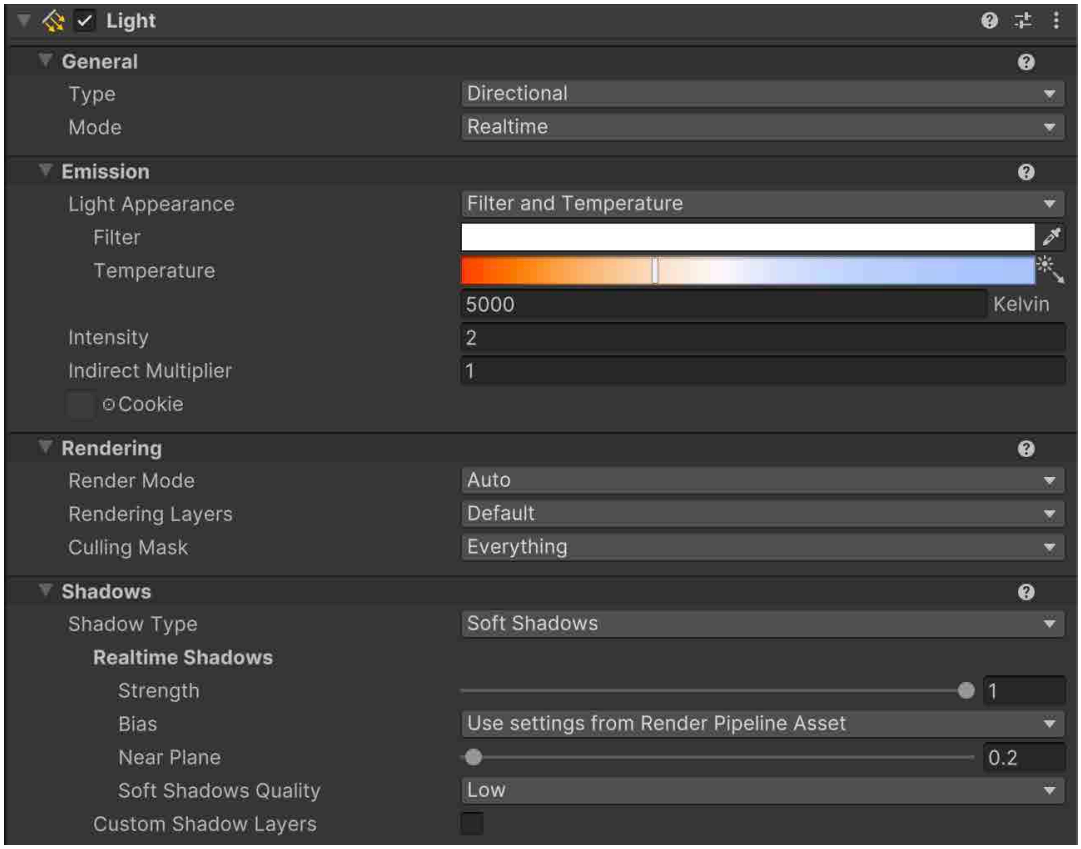
このガイドの後半では、URP アセットの設定を使用して、オブジェクトに影響を与える動的ライトの数を設定する方法を学びます。これは Object Per Light の制限を介して可能で、URP フォワードレンダラーでは最大 8 つに制限されていますが、フォワード + とディファードでは無制限です。カメラごとに使用できる動的ライトの数も、ハードウェアによって制限されます。

- **デスクトップおよびコンソールプラットフォーム**：メインライト 1 つ、追加ライト 256 個
- **モバイルプラットフォーム**：メインライト 1 つ、追加ライト 32 個
- **OpenGL ES 3.0 以前**：メインライト 1 つ、追加ライト 16 個

Light Inspector

Light Inspector は、ライティングの設定を行える 3 つの場所のうちの 1 つです。

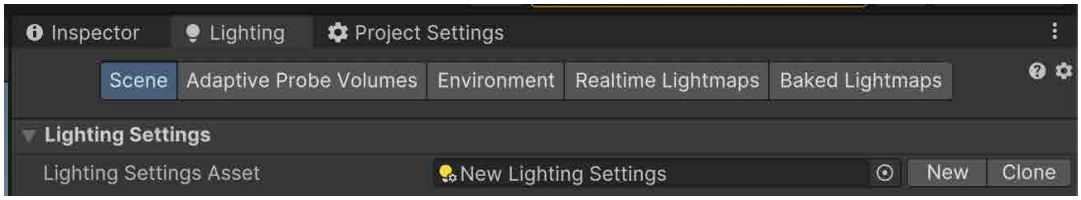
URP のライトで利用可能なプロパティは、Directional、Spot、Point、Area です。ただし、エリアライトは「Baked Indirect」モードでのみ機能します。詳細についてはライトモードのセクションを参照してください。



URP の「Light Inspector」パネル

上の画像は、Light Inspector でライトプロパティがどのように表示されるかを示しています。URP バージョンには、ライトが Directional か Point かによって分けられる 4 つの制御グループと、Spot と Area ライト用の追加の形状グループがあります。

新規シーンのライティング

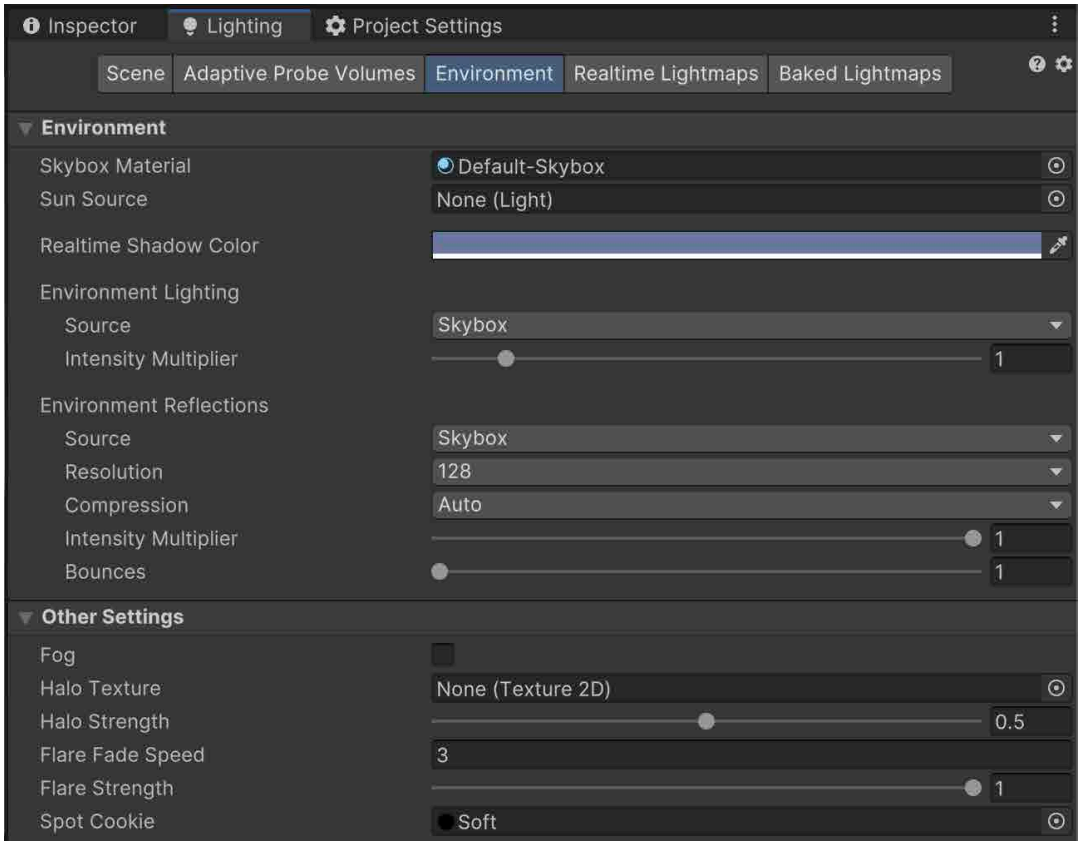


ライティング設定アセットの作成

URP 用の新規シーンをライティングするための最初のステップは、新しいライティング設定アセットを作成することです（以下の画像参照）。**「Window」>「Rendering」>「Lighting」**を開き、**「Scene」**タブで**「New Lighting Settings」**をクリックし、新しいアセットに名前を付けます。「Lighting」パネルで設定した内容が、このアセットに保存されます。ライティング設定アセットを切り替えることで、設定を変更できます。

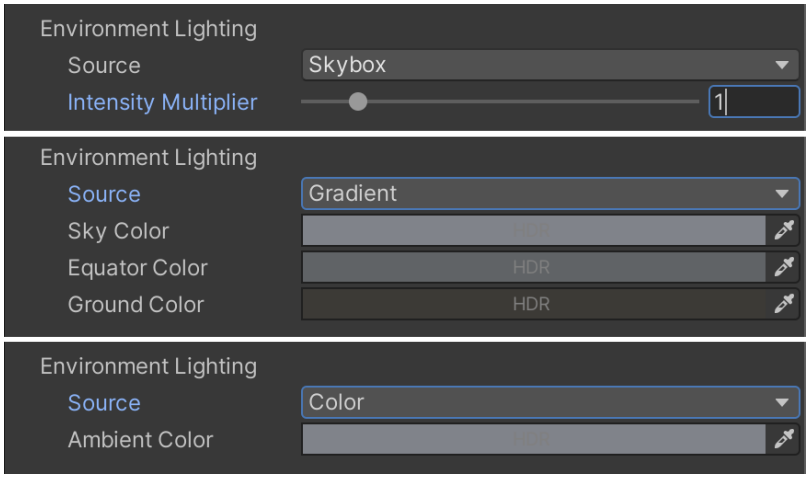
アンビエントまたは環境ライティング

主なアンビエントライトは、**「Window」** > **「Rendering」** > **「Lighting」** > **「Environment」** からアクセスできるパネルで計算されます。



「Environment」 パネルでアクセス可能なライティングの設定

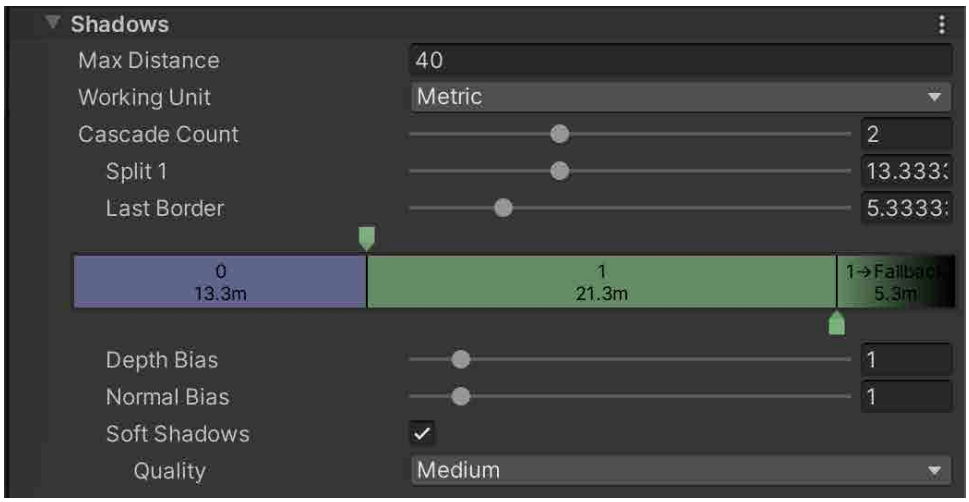
環境ライティングは、シーンのスカイボックスを使用するように設定でき、強度、グラデーション、または色を調整するオプションがあります。グラデーションとカラーモードのみリアルタイムで更新されます。スカイボックスモードでは、空からのアンビエントプローブを計算するためにオンデマンドベイクが必要です。



環境ライティングオプション

影：

前述したように、URP を使用する場合は、レンダラーデータオブジェクトとレンダーパイプラインアセットが必要です。URP プロジェクトのセットアップに関するセクションでは、レンダーパイプラインアセットを使ってシーンを表示する方法が説明されています。このアセットを使うと、影の忠実度を設定することができます。



URP アセット

メインライトの影の解像度

URP アセットのライティングとシャドウのグループは、シーンに影を設定するうえでの重要なポイントとなります。まず、「**Main Light Shadow**」を「**Disabled**」または「**Per Pixel**」に設定し、その後チェックボックスに移動して「**Cast Shadows**」を有効にします。最後に、シャドウマップの解像度を設定します。

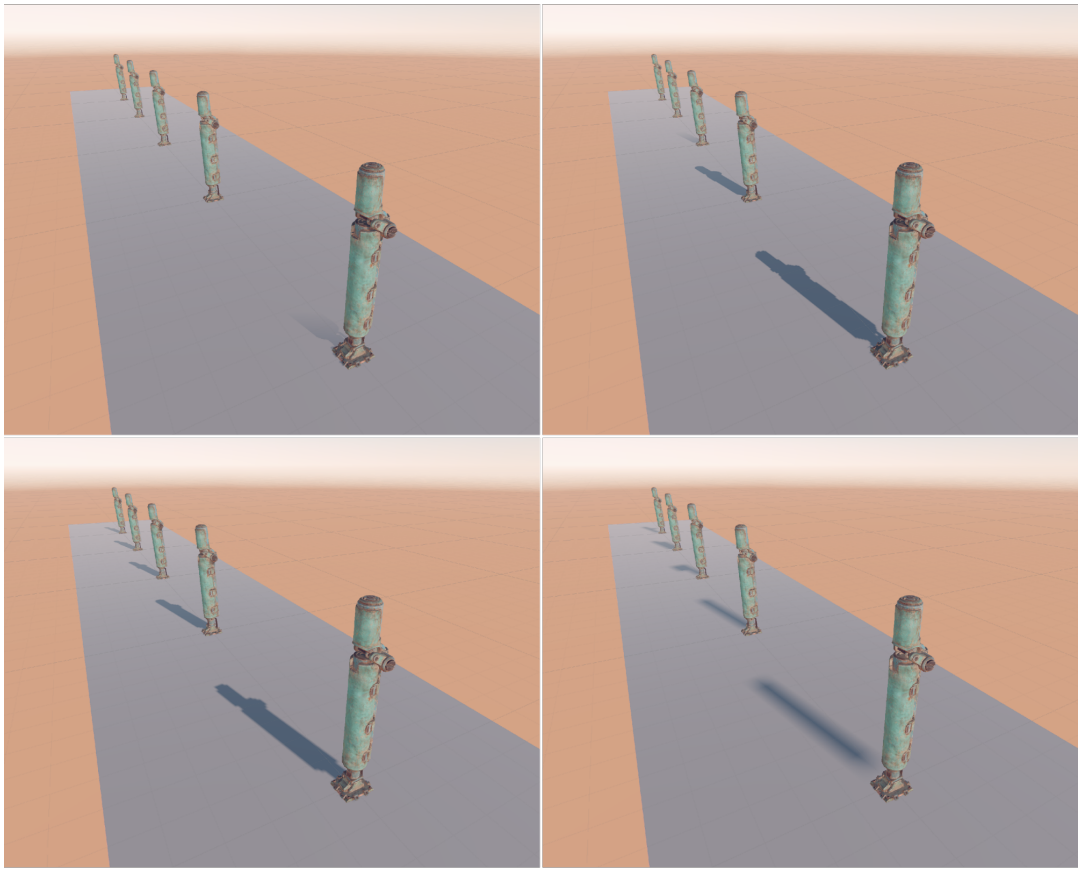
Unity で影を扱った経験がある方なら、リアルタイムシャドウでは、ライトの視点から見たオブジェクトの深度を含むシャドウマップをレンダリングする必要があることをご存知だと思います。このシャドウマップの解像度が高いほど、より現実感のある見た目になります。ただし、処理能力とメモリの両方で拡張が必要となります。影の処理が増える要因としては、以下のものがあります。

1. シャドウマップでレンダリングされるシャドウキャスターの数。メインライトの場合、この数はシャドウディスタンス（シャドウ錐台の遠平面）によって決まります。
2. 画面に表示されるシャドウレシーバー（すべて含める必要があります）
3. シャドウカスケードの分割
4. シャドウフィルタリング（ソフトシャドウ）

最高解像度が必ずしも理想的であるとは限りません。例えば、「**Soft Shadows**」オプションにはマップをぼかす効果があります。以下の幽霊屋敷風の部屋の画像では、手前の椅子が机の引き出しに影を落としていますが、解像度が 1024 を超えると影が鮮明になり過ぎてしまいます。

メインライトの影の解像度の設定：解像度は、左上の画像では 256、右上の画像では 512、中央左の画像では 1024、中央右の画像では 2048、下の画像では 4096 に設定されています。

メインライトシャドウの Max Distance



メインライトシャドウの様々な Max Distance：左上の画像 – 10、右上の画像 – 30、左下の画像 – 60、右下の画像 – 400

メインライトシャドウのもう 1 つの重要な設定は、「Max Distance」です。これはシーンユニットで設定されます。上の画像では、ポールの間隔が 10 ユニットになっています。Max Distance は、10 ユニットから 400 ユニットまで設定されています。左上の画像では、1 つ目のポールだけが影を落としていて、カメラの位置から 10 ユニットの距離で影が途切れています。60 ユニット（左下の画像）では、すべての影が表示されていて、影の忠実度が適切です。Max Distance が見えているアセットよりも遥かに大きいと、シャドウマップが広範囲に分散されてしまいます。これは、画面内の領域が必要以上に低い解像度になることを意味します。

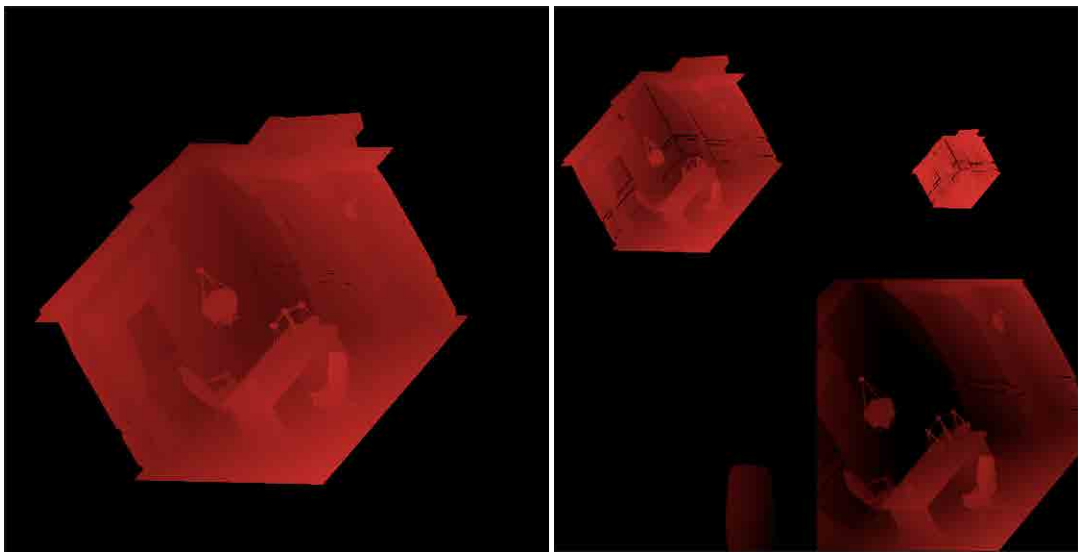
「Max Distance」プロパティは、ユーザーが見える範囲とシーンで使用される単位に直接関連付ける必要があります。影の品質が許容できる範囲で、最小の距離を設定するようにしてください（下記の注記を参照）。プレイヤーがカメラから 60 ユニット以内の動的オブジェクトの影しか見えないのであれば、「Max Distance」を 60 に設定します。混合ライトのライティングモードを「Shadowmask」に設定した場合、シャドウディスタンスを超えるオブジェクトの影はベイクされます。これが静的シーンであった場合は、すべてのオブジェクトに影が表示されますが、シャドウディスタンス内では動的シャドウのみが描画されます。

シャドウカスケード

遠くのアセットは遠近法によって見えなくなるため、影の解像度を下げ、シャドウマップの範囲をよりカメラに近い影に割り当てると便利です。シャドウカスケードは、こういった際に役に立ちます。

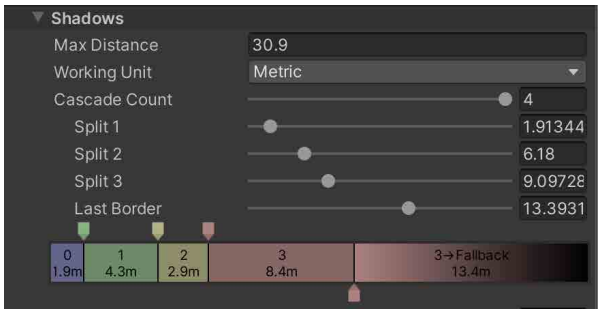
下の画像は、幽霊屋敷の部屋にある椅子と机が映ったシーンのシャドウマップです。左の画像では、カスケードカウントは 1 です。マップがエリア全体を占めています。右の画像では、カスケードカウントは 4 です。マップには 4 つの異なるマップが含まれており、各エリアにより解像度の低いマップが割り当てられていることがわかります。

このような小規模なシーンでは、多くの場合、カスケードカウントが 1 でも最適な結果が得られます。ただし、Max Distance が大きい場合は、カスケードカウントを 2 や 3 にした方が、割り当てられるシャドウマップの割合が大きくなるので、前景のオブジェクトに対してより良い影が得られます。左の画像の椅子の方がはるかに大きく、結果として影がより鮮明になっていることに注目してください。



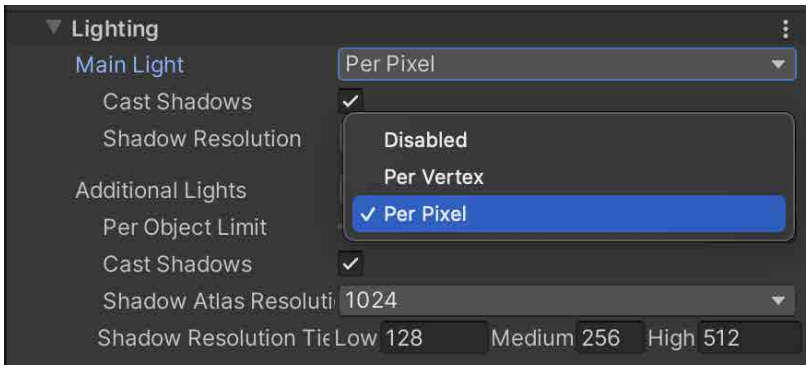
カスケードカウントを 1 に設定した場合（左の画像）と、4 に設定した場合（右の画像）のシャドウマップ

カスケードの各セクションの開始範囲と終了範囲は、ドラッグ可能なポインターを使用するか、関連するフィールドでユニット数を設定することで調整できます（以下の画像を参照）。「Max Distance」は常にシーンに適した値に調整し、スライダーの位置は慎重に選択するようにしてください。作業単位として「Metric」を使用する場合、最後のカスケードは必ず最後のシャドウキャスターの距離（最大）に設定してください。



シャドウカスケードの範囲の調整

追加ライトのシャドウ

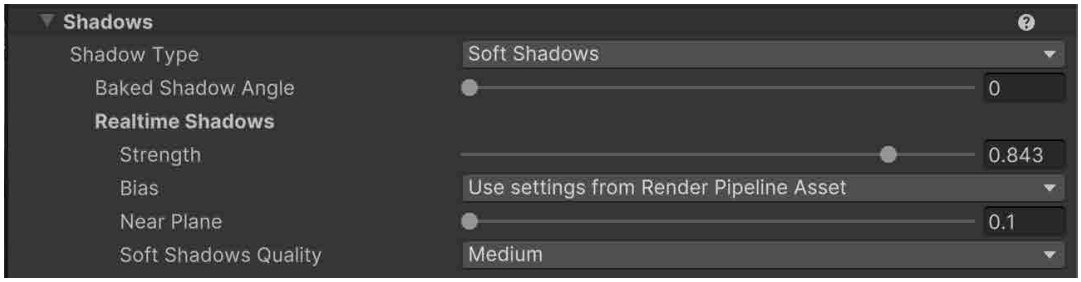


URP アセットの追加ライトで使用できる設定

メインライトのシャドウをソートしたら、次は**追加ライトモード**に移りましょう。追加ライトによる影の投影を有効にするには、URP アセットの追加ライトモードを「**Per Pixel**」に設定します。モードは、「Disabled」、「Per Vertex」、または「Per Pixel」（上の画像を参照）に設定できますが、影に対応しているのは「Per Pixel」だけです。

注：URP は追加のディレクショナルライトの影をサポートしていません。メインライトは常に最も明るいディレクショナルライトであることを忘れないでください。影を持つ追加ライトには、ポイントライトまたはスポットライトを使用してください。

「**Cast Shadows**」ボックスをオンにします。次に、「**Shadow Atlas**」の解像度を選択します。これは、影を落とすすべてのライトのマップを結合するために使用されるマップです。ポイントライトは光を全方向に投影するため、6 つのシャドウマップを使用してキューブマップを作成します。そのため、ポイントライトはパフォーマンス面で最も要件の厳しいライトとなっています。追加ライトのシャドウマップの個々の解像度は、3 つのシャドウ解像度の階層と、「Hierarchy」ウィンドウでライトを選択した際に Light Inspector で設定した解像度の組み合わせで決まります。



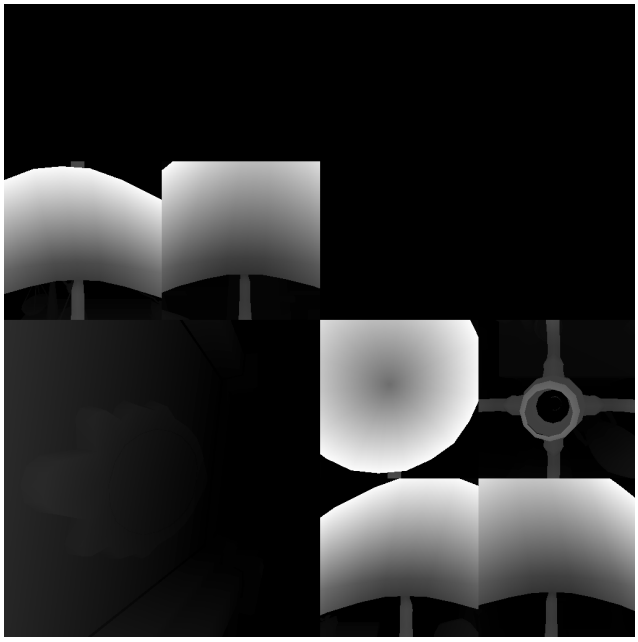
Light Inspector の「Shadows」グループ

Shadow Type を **Soft Shadows** に設定すると、**Baked Shadow Angle** スライダーが有効になります。このプロパティは、影のエッジに人工的な柔らかさを加え、より自然な外観にします。Unity 6 では、**Soft Shadows Quality** を Low、Medium、High で切り替える新しいオプションが利用可能です。

幽霊屋敷の部屋では、鏡の上にスポットライトがあり、机の上にポイントライトがあります。また、7 つのマップもあります。これらの 7 つのマップを 1024px の正方形のマップに適合させるには、各マップのサイズを 256px 以下にする必要があります。このサイズを超えると、シャドウマップの解像度がアトラスに合わせて縮小され、コンソールに警告メッセージが表示されます。

マップの数	アトラスのタイリング	アトラスのサイズ（シャドウ階層のサイズを乗算する数）
1	1×1	1
2–4	2×2	2
5–16	4×4	4

追加ライトのシャドウマップの数とマップごとに選択された階層サイズに基づくシャドウアトラスサイズの設定



追加ライト用のシャドウアトラス

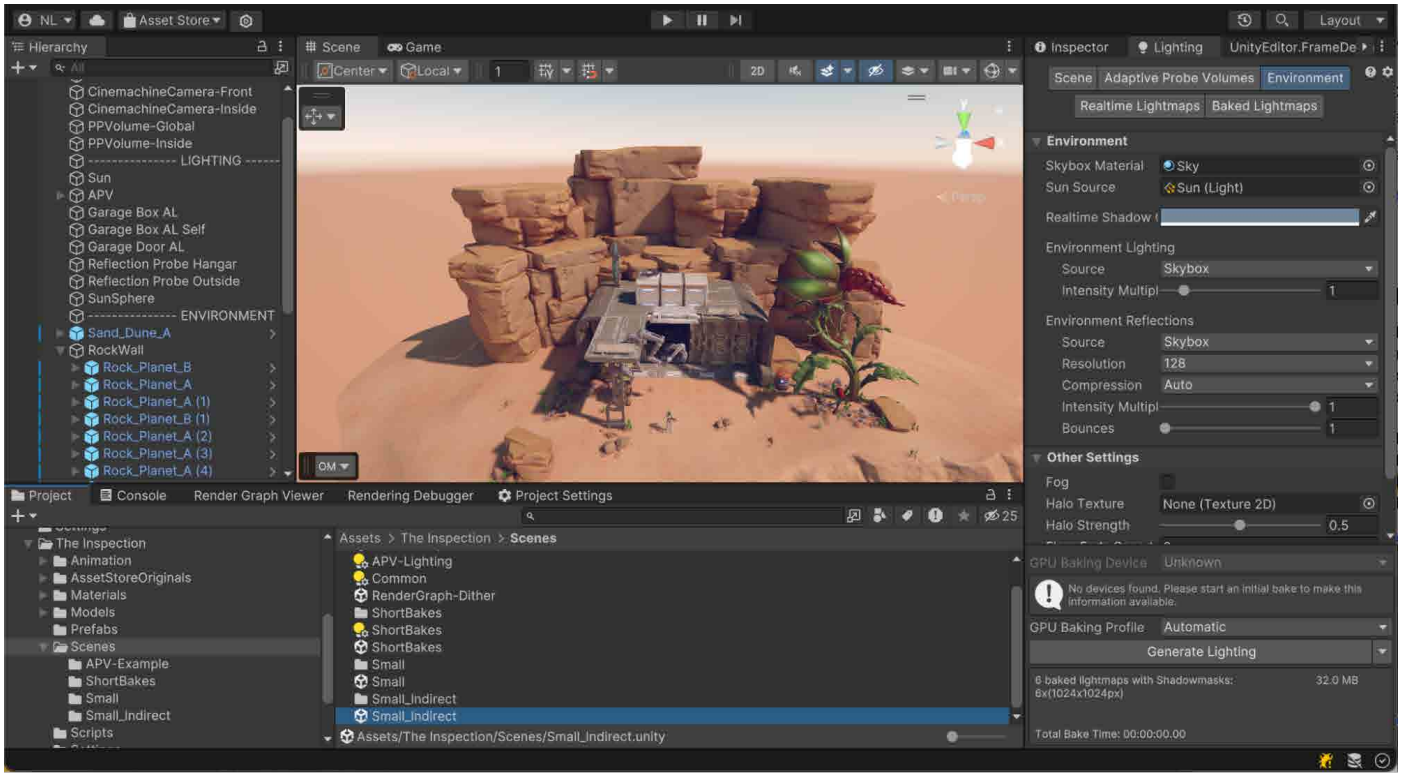
上の画像は、解像度 が「Medium」に設定され、階層値が 256px に設定されたポイントライトで使用される 6 つのマップを示したものです。スポットライトの解像度は「High」に設定されていて、階層値は 512px です。

これは、メインのディレクショナルライト、机の上のポイントライト、鏡の上のスポットライトでライティングされた、幽霊屋敷の低ポリゴンバージョンです。すべてのライトがリアルタイムで、影を投影しています。

ライトモード

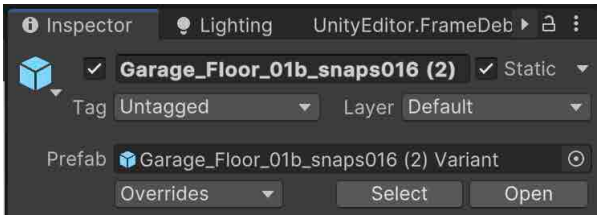
環境は、主に静的なジオメトリを持つため、ライトが静的であれば、何度もライトと影を計算する必要はありません。デザイン時に一度だけ計算し、そのデータをジオメトリのレンダリング時に利用できます。この手法をライトマッピングまたはベイキングと呼びます。

Unity の FPS サンプルプロジェクトを使って手順を説明しましょう。次のスクリーンショットは、このプロジェクトからのものです。プロジェクトは、こちらからダウンロードできます。**「Inspection」>「Scenes」>「Small Indirect」**のシーンでは、URP でリアルタイムとベイクしたライティングを使用する方法が示されています。

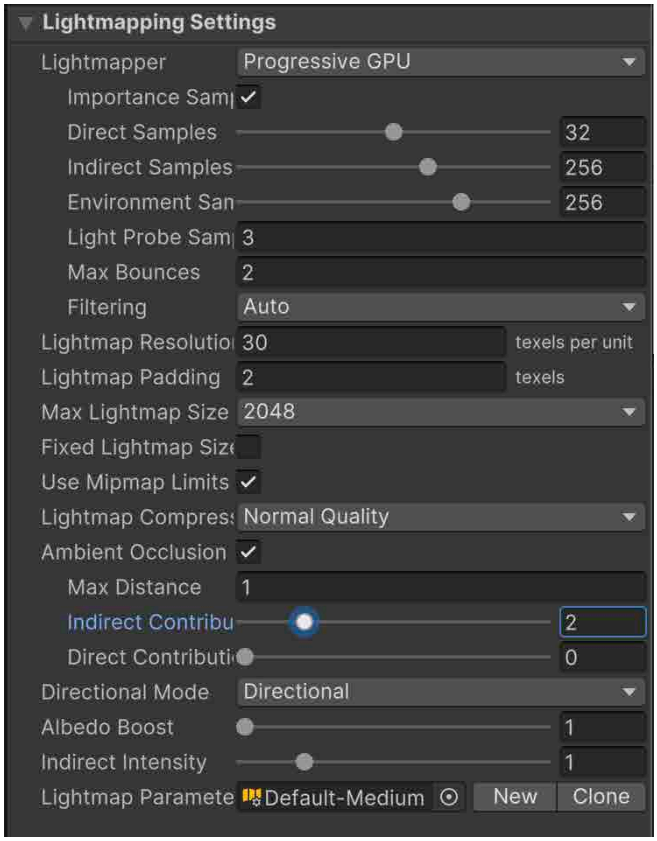


「Inspection」>「Scenes」>「Small Indirect」のシーンは、GitHub の Unity 6 URP e ブックリポジトリから取得できます。

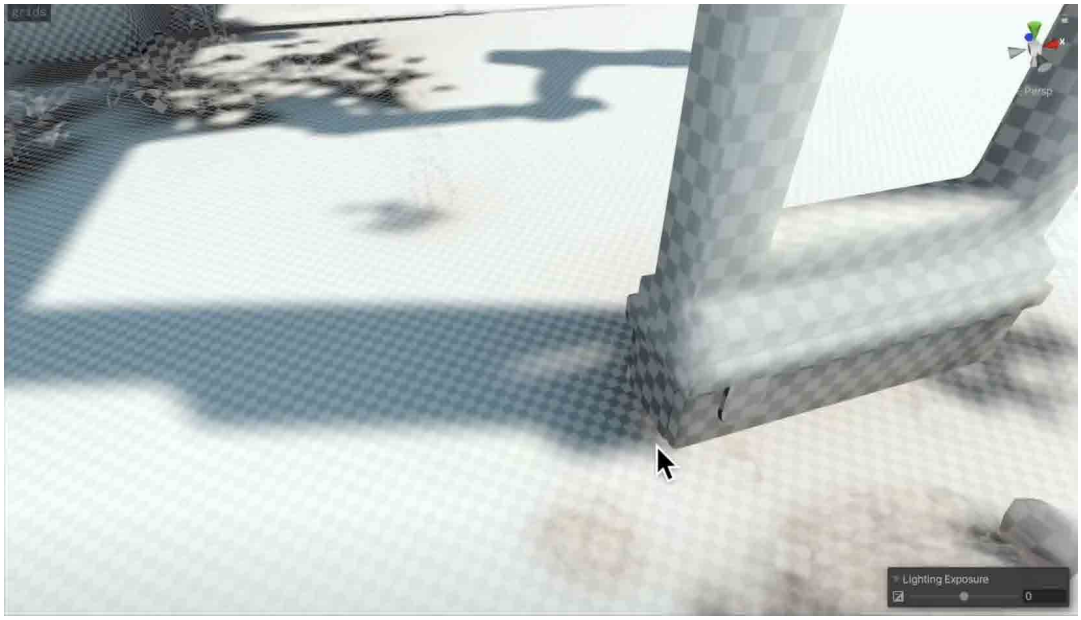
1. FPS サンプルプロジェクトのシーンは、主に静的ジオメトリで構成されています。このジオメトリをライトマッピングに含めるには、Inspector の右側にある「**Static**」ボックスをクリックします。



2. 「**Window**」>「**Rendering**」>「**Lighting**」>「**Scene**」からライトマッピング設定を選択します。ライトマップの解像度を低く維持したまま、設定を調整します。目的の設定が完了したら、最終的なライトマップの生成時に値を大きくします。GPU が対応している場合は、ライトマップ生成を高速化するために「**Progressive GPU**」を選択します。

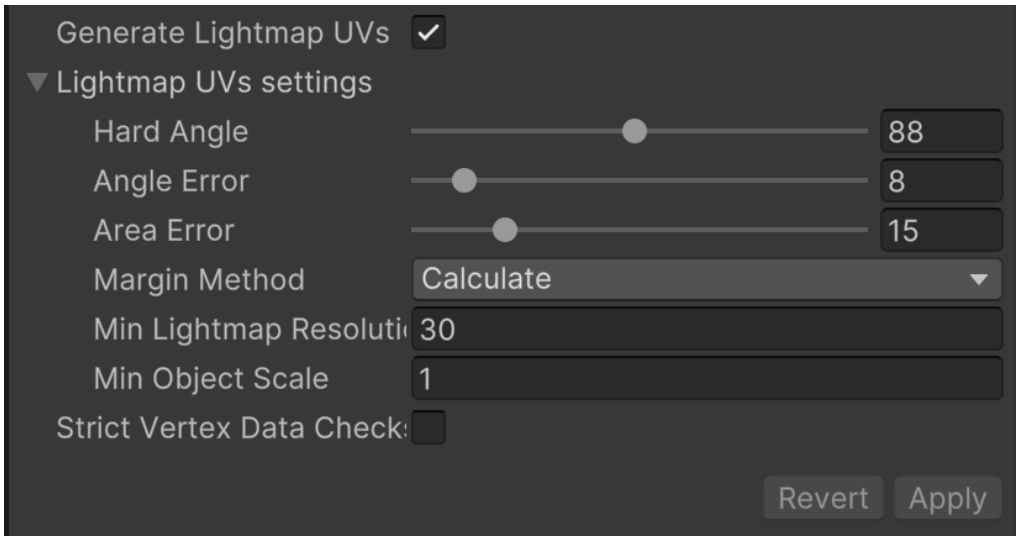


3. フィルタリングは、ノイズを最小限に抑えるためにマップをぼかします。これにより、オブジェクト同士が接する部分の影にズレが生じることがあります。このアーティファクトを最小限に抑えるには、**A-Trous** フィルタリングを使用してください。ライトマッピングで利用できる設定の詳細は、プログレッシブ ライトマッピング のドキュメントを参照してください。

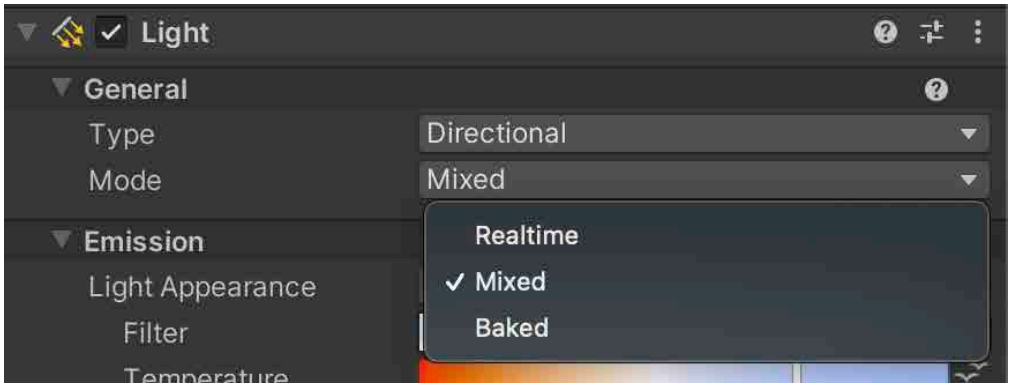


フィルタリングがオブジェクト間の影に与える影響

4. すべての静的ジオメトリで UV 値が重複していないこと、またはインポート時にライティング用 **UV** が生成されることを確認します。



5. 「**Light Mode**」を「**Baked**」または「**Mixed**」に設定します。**Hierarchy** ウィンドウでライトを選択し、**Inspector** を使用します。混合ライトは、静的なオブジェクトだけでなく、動的なオブジェクトにも照明を当てます。



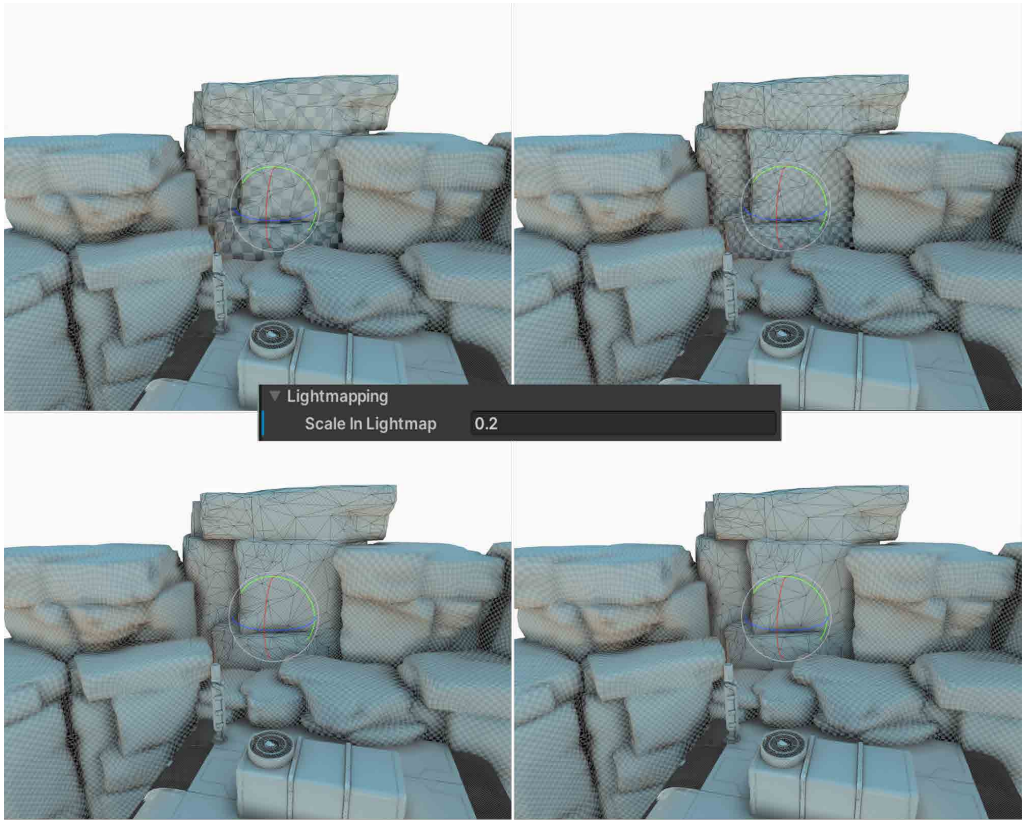
6. 混合ライトを使用する場合は、「**Window**」>「**Rendering**」>「**Lighting**」>「**Scene**」から、「**Light Mode**」を「**Baked Indirect**」、「**Subtractive**」、または「**Shadowmask**」に設定します。

   a. **Baked Indirect**：間接光の影響のみがライトマップとライトプローブにベイクされます（反射のみ）。直接光と影はリアルタイムで計算されます。この設定は処理負荷が高く、モバイルプラットフォームには理想的ではありません。その代わり、静的および動的ジオメトリの両方に正確な影と直接光が適用されます。

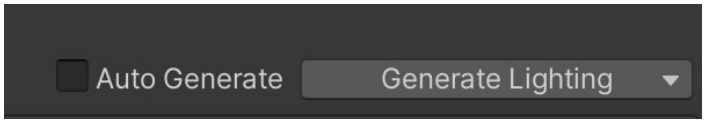
b. **Subtractive**：「Mixed」に設定されたディレクショナルライトからの直接光を静的ジオメトリにベイクし、動的ジオメトリが作る影からライティングを減算します。この結果、ライトプローブやアダプティブプローブボリューム（APV）を使用しない限り、静的ジオメトリが動的オブジェクトに影を投影できなくなるため、不快な視覚的な切れ目が発生する可能性があります。URP では、ディレクショナルライトからの光の影響の推定値が計算され、ベイクしたグローバルイルミネーションからその値が差し引かれます。この推定値は、Lighting ウィンドウの「Environment」セクションにある「Real-time Shadow Color」設定によって固定されるため、減算された色がその色よりも暗くなることはありません。次に、減算された値と元のベイクされた色のうち、最低値の色が適用されます。このモードはローエンドのハードウェアに最適ですが、ベイクドシャドウとリアルタイムシャドウをランタイム時に正しく組み合わせることができないため、アーティファクトが発生する可能性があります。このトレードオフでは、品質よりもパフォーマンスが優先されます。

c. **Shadowmask**：「Baked Indirect」モードと似ていますが、「Shadowmask」では、動的シャドウとベイクされたシャドウの両方を結合し、遠方の影をレンダリングします。これは、追加のシャドウマスクテクスチャを使用し、ライトプローブに追加情報を保存することで実現されます。これによって最高レベルの影の品質が得られますが、メモリとパフォーマンスのコストも最も高くなります。近距離の描画では「Baked Indirect」と視覚的には同じですが、遠くを見たときに違いがわかるので、オープンワールドのシーンに適しています。負荷処理の理由から、ミッドエンドやハイエンドのハードウェアでのみ推奨されます。

7. 「**Asset**」>「**Inspector**」>「**Mesh Renderer**」>「**Lightmapping**」>「**Scale In Lightmap**」から「**Lightmap Scale**」を調整して、遠くのオブジェクトがライトマップ上で占めるスペースを減らします。以下の画像は、背景の岩のライトマップのテクセルサイズを示したものです。設定には 0.05 から 0.5 までの開きがあります。



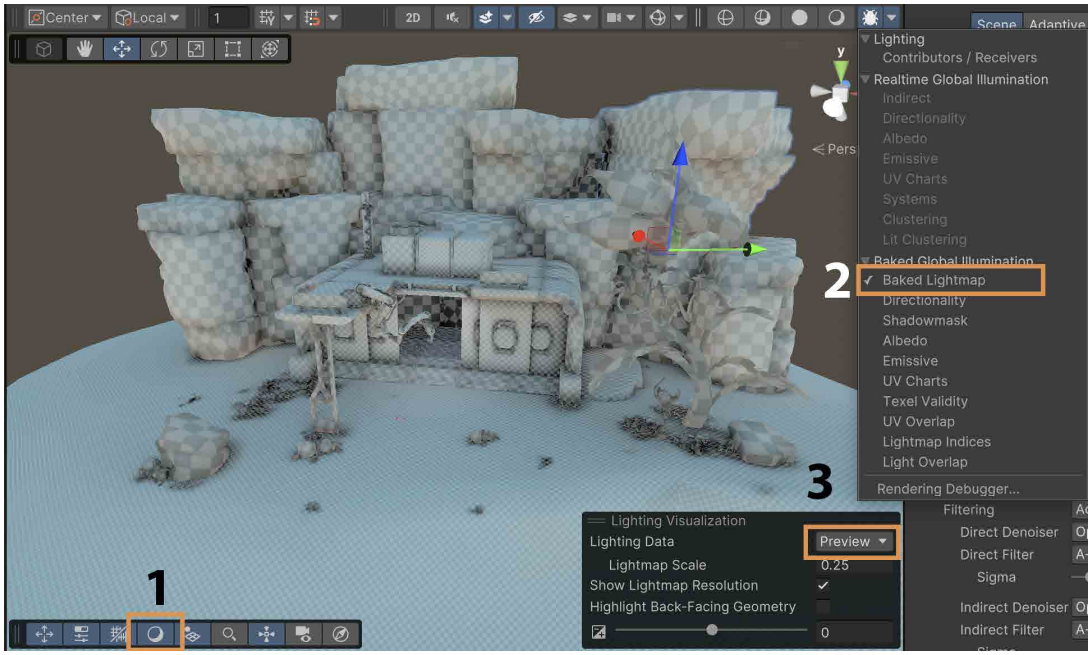
8. 「**Generate Lighting**」をクリックしてベイクします。ベイクの処理時間は、静的オブジェクトの数、ライトの設定（「Mixed」モードまたは「Baked」モード）、およびライトマッピングの設定（特に「Max Lightmap Size」と「Lightmap Resolution」）によって変わります。ベイク時間はベイクに使用されるレイの数に比例するため、Sample Count（Direct Samples、Indirect Samples、Environment Samples）もベイク時間に直接影響します。



「**Window**」>「**Rendering**」>「**Lighting**」>「**Generate Lighting**」からライトマップをベイクします。

9. インタラクティブベイク機能は、Draw Modes がアクティブ（1）* で、ベイクされたライトマップが選択 (2) されている場合に利用できる Unity 6 の新機能です。新しいパネルがプレビューオプション（3）とともに表示されます。このモードでは、変更は生成されたデータに影響を与えません。この機能により、テクニカルアーティストはプロパティを微調整し、長時間の計算に時間を要した以前のベイクを破棄することなく、変更がレンダリングにどのように影響するかを確認できます。

   * 番号は以下の画像を参照してください。



インタラクティブプレビューモードの有効化

Unity では、**Baking Profile** も提供されるようになりました。これは、GPU バックエンドをオンデマンドモードで使用しているときに、**Lighting** ウィンドウで確認でき、パフォーマンスと GPU メモリ使用量のバランス調整をユーザーに提供します。

GPU Baking Profile

注：

2019 版のリリース以降、Unity は明示的にベイクされていないシーンでベイクされた環境照明を自動的に生成するシステムを提供してきました。このシステムは SkyManager と呼ばれていました。しかし、SkyManager は自動動作が分かりづらく、特定の状況でしか機能しないため、ユーザーの混乱を招いていました。さらに、このシステムはエディターとビルドされたプレイヤーで動作に違いが生じ、環境ライティングが予期せず欠落してしまうこともありました。

Unity 6 では、SkyManager がエディターの新しいデフォルトのライティングデータアセットに置き換えられ、新規作成シーンに割り当てられます。このアセットには、環境ライティングのデフォルト設定に一致する環境ライティングが含まれています。Skybox モードを使用してこれらの設定を変更した場合、Lighting ウィンドウの「**Generate Lighting**」ボタンを使用して手動でライティングをリベイクする必要があります。

その他のリソース：

- ライトマッピングのドキュメント
- ライティング 設定 アセットのドキュメント
- ライティングエクスプローラーのドキュメント
- よくあるライトマッピングの問題 5 つとその解決に役立つヒント

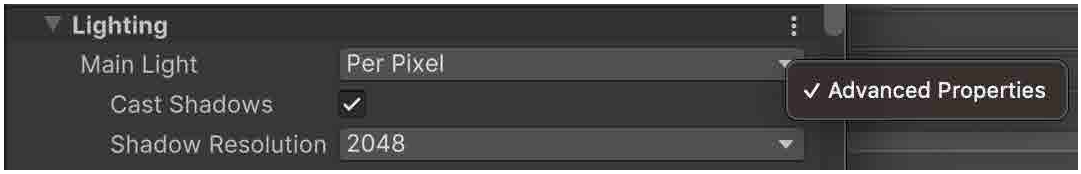
レンダリングレイヤー

レンダリング レイヤー機能を使用することで、特定のライトが特定のゲームオブジェクトにのみ影響するよう設定して、シーン内でそのオブジェクトを強調し、注目させることができます。下の画像では、重要な収集アイテムである注射器が、シーンの影の部分に表示されています。レンダリングレイヤーを使用することで、注射器が見やすくなり、プレイヤーの見落としを防げます。

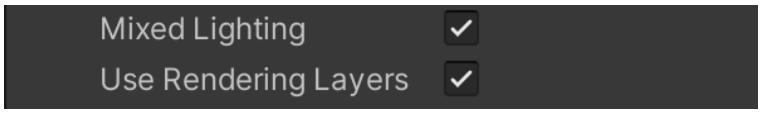
レンダリングレイヤーを使用したオブジェクトのハイライト

レンダリングレイヤーの設定手順は以下の通りです。

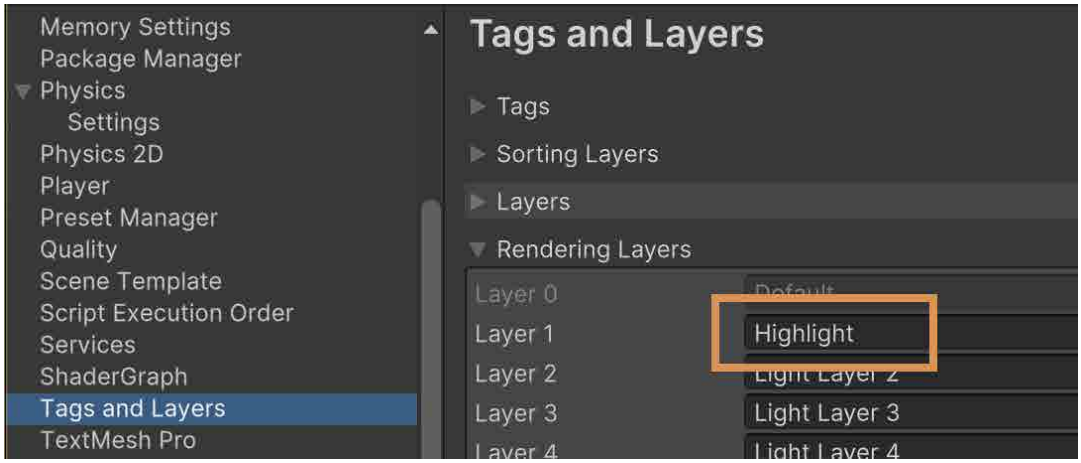
1. **URP アセット**を選択します。「Lighting」セクションで、縦の省略記号（:）のアイコンをクリックし、「**Advanced Properties**」を選択します。



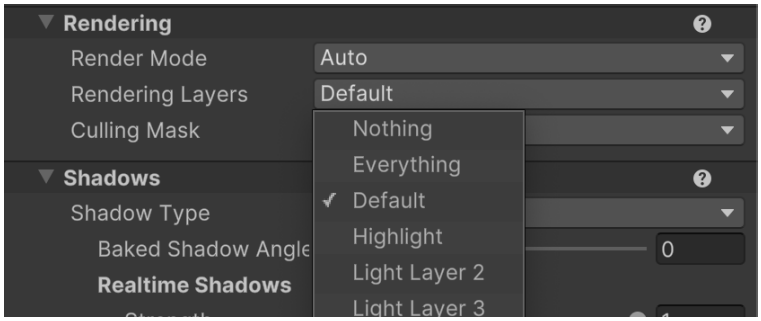
2. 「Lighting」セクションの下に、新しい設定「**Use Rendering Layers**」が表示されます。



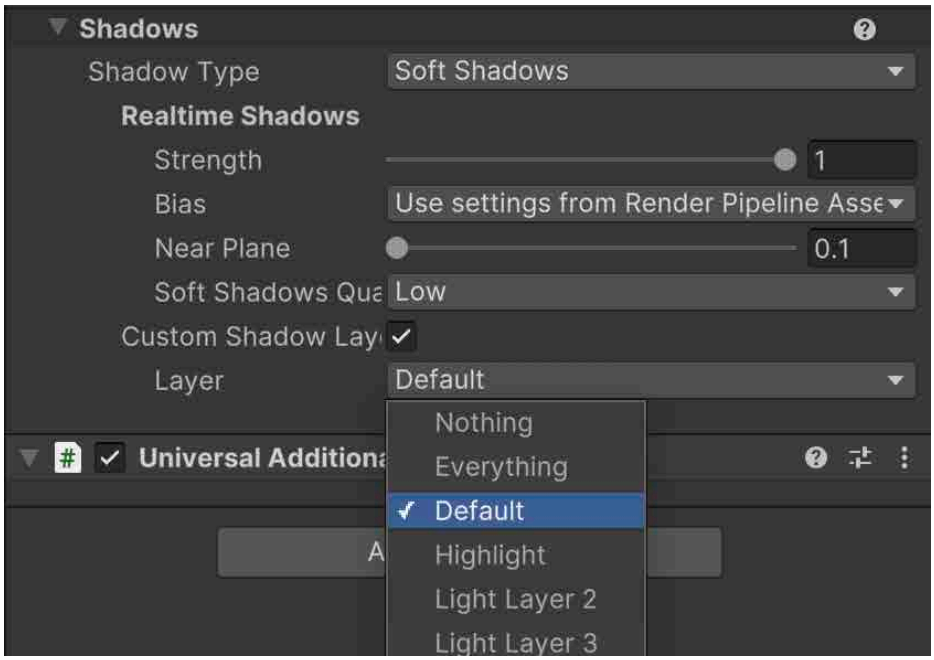
3. **「Project Settings」>「Tags and Layers」>「Rendering Layers」**からレンダリングレイヤーの名前を変更します。



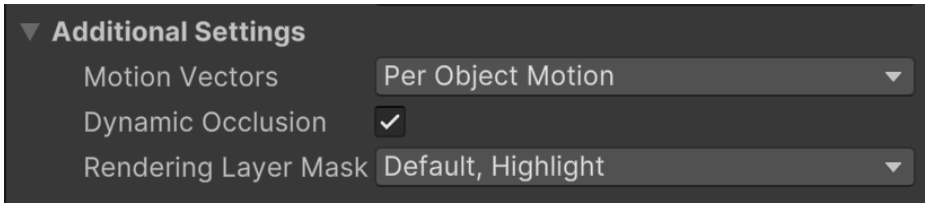
4. **「Light Inspector」>「Rendering」**セクションには、「**Rendering Layers**」のドロップダウンがあります。ライトは複数のレイヤーに影響を与えることができます。



5. レンダリングレイヤーを有効にしたら、新しいライトを作成し、カスタムシャドウレイヤーを設定してください。新しいライトは、シーンの**メインライト**や自身の視錐台から影を落とすことができます。



6. 最後に、**Hierarchy** ウィンドウでこれを適用するオブジェクトを選択し、「**Rendering Layer Mask**」を設定します。



これはコードで動的に設定することもできます。

```
Renderer renderer = GetComponent<Renderer>();
int layerID = 1;
int mask = 1 << layerID;
renderer.renderingLayerMask = (uint)mask;
```

ライトプローブ

前述のライトモードのセクションで説明したように、混合ライティングモードを使用してベイクされたオブジェクトと動的オブジェクトを組み合わせることができます。混合ライティングモードを使用する際は、シーンにプローブを追加することをお勧めします。Unity 6 では、ライトプローブと新しいアダプティブプローブボリューム（APV）の 2 つのオプションがあります。2 つのオプションはどちらも、動的オブジェクトがシーン内を移動し、グローバルイルミネーションの影響を受けることを可能にするものです。

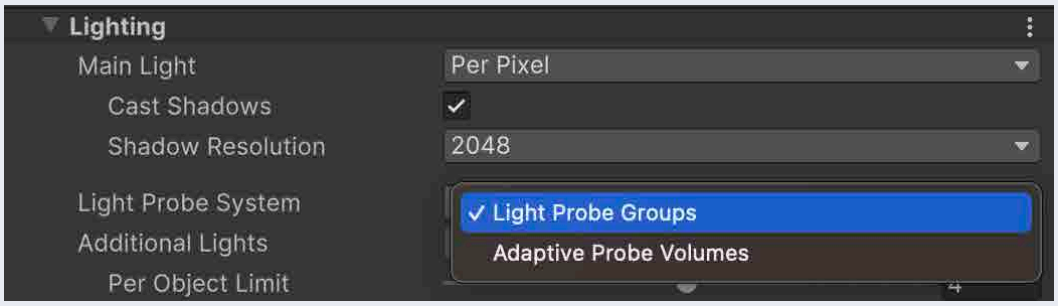
プローブは、シーン内の単なるポイントです。デザイン時に、このポイントでのグローバルイルミネーションが計算されます。実行時にフレームをレンダリングする際、ライティング計算を含む URP シェーダーは、グローバルイルミネーションの値として最も近いプローブをブレンドして使用します。

注：グローバルイルミネーション（GI）は、光が表面に当たって他の表面へ反射し、間接光を生み出す様子を再現するシステムで、直接光源からの光だけに限定されません。

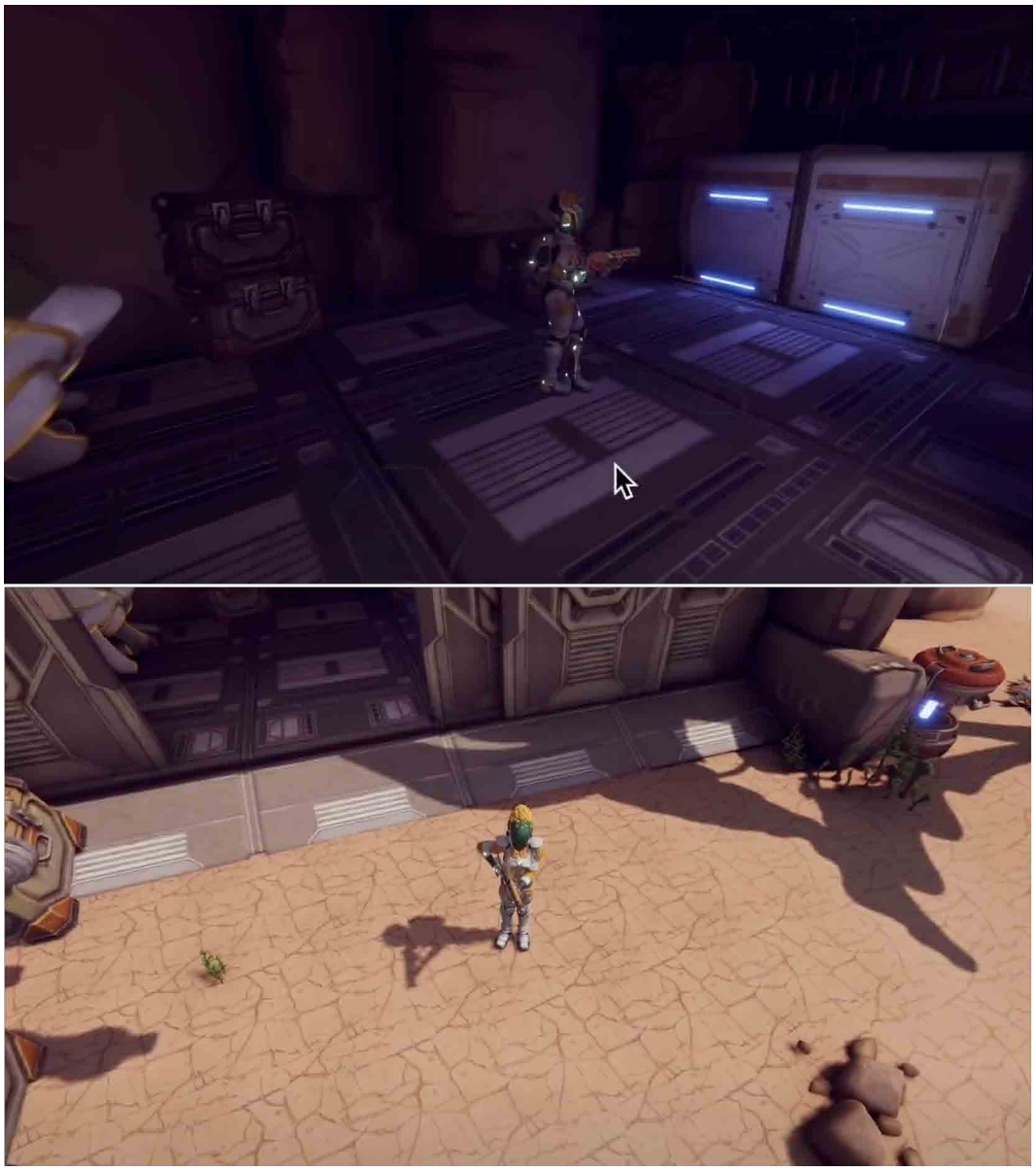
ライトプローブ

ライトプローブは、**「Window」>「Rendering」>「Lighting」**パネルから「**Generate Lighting**」をクリックしてライティングをベイクしたときに、環境内の特定の位置にライトデータを保存します。これにより、環境内を移動する動的オブジェクトのイルミネーションに、ベイクされたオブジェクトで使用されているライティングレベルが反映されるようになります。暗い場所ではオブジェクトが暗くなり、明るい場所では明るくなります。サンプリングはオブジェクトごとに行われるため、大きなオブジェクトが暗い領域から明るい領域にまたがる場合、ライティングに異常が生じる可能性があります。これによりシーン内で問題が発生する場合は、ピクセルごとにサンプリングされる APV の使用を検討してください（このガイドの後半にある APV のセクションを参照）。

注：ライトプローブを使用する際は、アクティブな URP アセットの**「Lighting」>「Light Probe System」**が「**Light Probe Group**」に設定されていることを確認する必要があります。

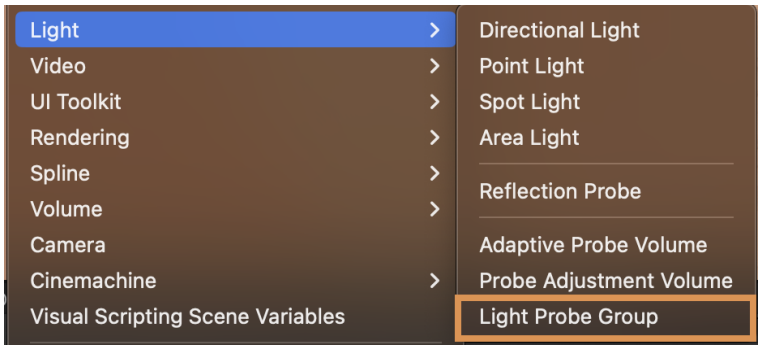


下の FPS サンプル「The Inspection」の画像では、ロボットが格納庫の中と外にいる様子を示しています。



格納庫の中と外で、ライトプローブによってライティングレベルに影響を受けているロボット

ライトプローブを作成するには、**Hierarchy** ウィンドウを右クリックし、**「GameObject」>「Light」>「Light Probe Group」** を選択します。



「Light Probe Group」の新しいゲームオブジェクトの作成

初期状態では、プローブのキューブは合計 8 個あります。ライトプローブの位置を表示して編集し、ライトプローブを追加するには、「**Hierarchy**」ウィンドウで「**Light Probe Group**」を選択し、シーンビューで**「Tools」>「Edit Light Probes」**をクリックします。必ずライトプローブのギズモをアクティブにしてください。

Inspector でのライトプローブの追加または削除および位置の変更

シーンビューが編集モードになり、ライトプローブだけを選択できるようになります。移動ツールを使ってライトプローブを移動させます。

ライトプローブの移動

ライトプローブは、まず動的オブジェクトが移動する可能性のある領域に配置し、その後、ライティングレベルに大きな変化が生じる領域に配置するようにしてください。オブジェクトのライティングレベルを計算するとき、エンジンは最も近いライトプローブのピラミッドを見つけ、それらを使ってライティングレベルの補間値を決定します。

選択したクレートに最も近いライトプローブ

プローブの配置には時間がかかりますが、こちらのようなコードベースのアプローチを使用すると、大きなシーンでの配置作業や、迅速に APV に切り替える場合などに、編集のスピードを上げられます。

クリエイターは、プロジェクト用のモジュール式コンテンツをシーンで構築することがよくあります。これらのシーンは、実行時に「ハブ」シーンで再配置されます。しかし、ライトプローブを含むモジュール式コンテンツを構築する際、クリエイターは、プローブの位置が読み取り専用であるため、これらのプローブをシーンと一緒に再配置することができませんでした。この問題は、Unity 6 の新しい API により解決されました。この API を使用すると、ランタイム時にライトプローブの位置を再配置することができます。

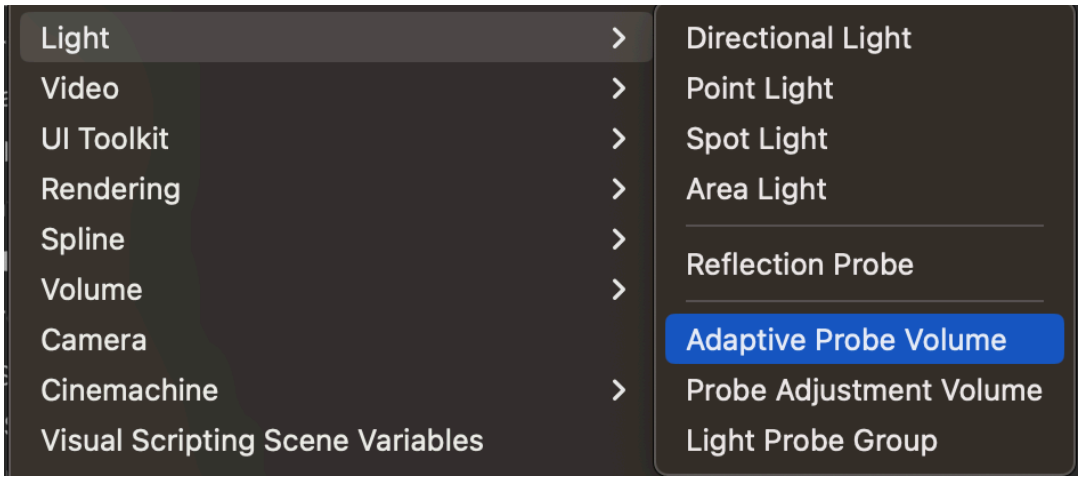
メッシュレンダラーとライトプローブの動作、および設定の調整方法について詳しくは、このドキュメントを参照してください。

アダプティブプローブボリューム

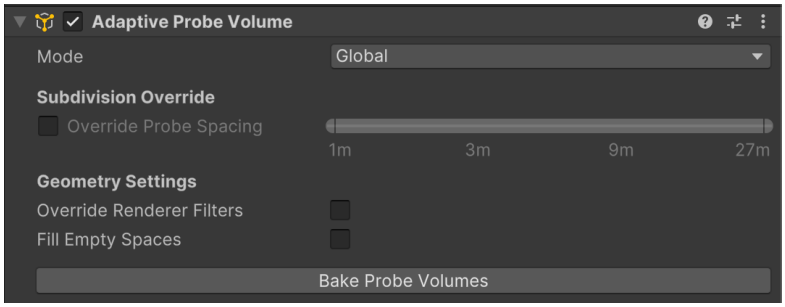
シーンのライティングプローブを慎重に配置したにもかかわらず、シーンのレイアウトが変更されてしまった場合、アダプティブプローブボリューム（APV）の利点をすぐに実感できるでしょう。多くのシーンで、すべてのプローブを数秒で配置できるようになりました。もう一度 FPS サンプル「The Inspection」を使って実用的な例を見てみましょう。この例は、**「The Inspection」>「Scenes」>「APV-Example」で見つけることができます。**

1. まず、アクティブな URP アセットで「**Light Probe System**」オプションが「**Adaptive Probe Volumes**」に設定されていることを確認します。

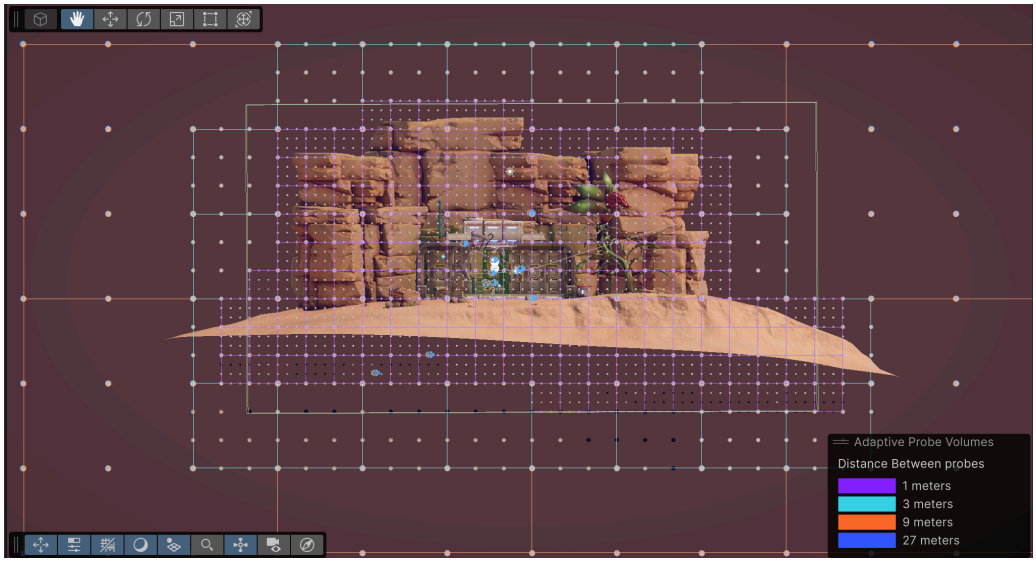
2. Hierarchy ウィンドウで右クリックして、**「GameObject」>「Light」>「Adaptive Probe Volume」（APV）**を選択します。



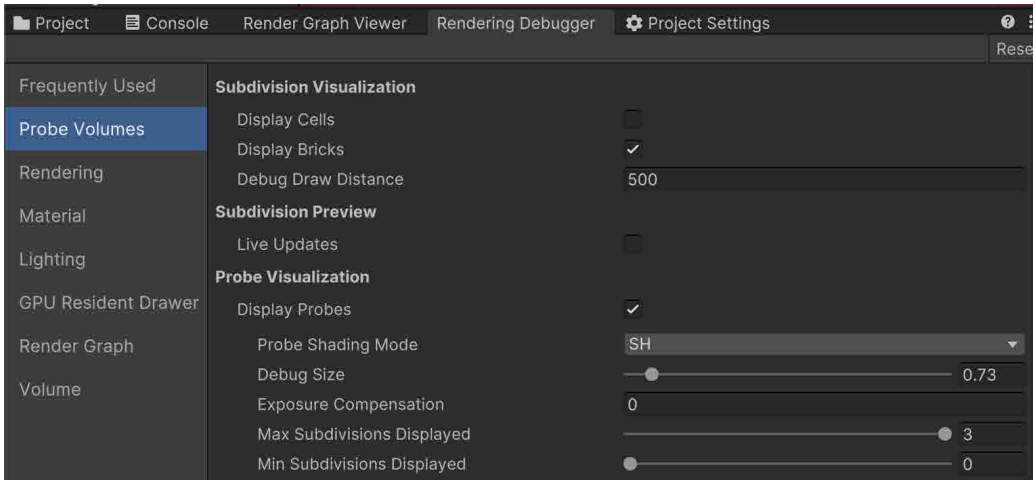
3. モードを「**Global**」に設定し、デフォルト設定（1、3、9、27 メートルのサブディビジョン）を使用します。



4. 「**Bake Probe Volumes**」をクリックしてボリュームをベイクします。現在のシーンがスキャンされ、シーン内のジオメトリに基づいてプローブが配置されます。プローブは、ジオメトリが最も多い場所に最も密集します。



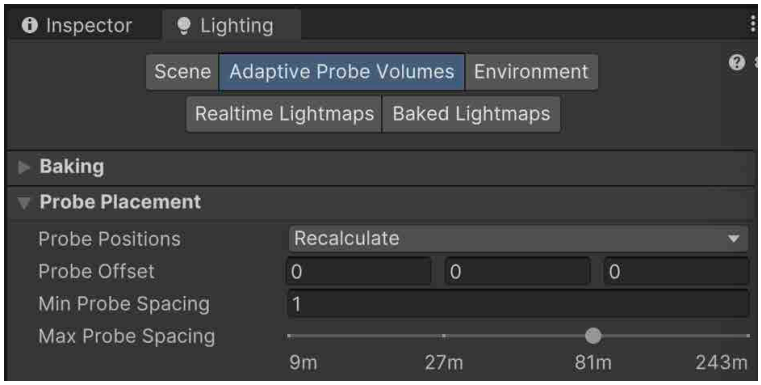
5. ベイクの結果を表示するには、「**Analysis**」>「**Rendering Debugger**」を開きます。「**Probe Volumes**」を選択し、「**Display Probes**」を選択します。異なる解像度を表示するには、「**Display Bricks**」を選択します。



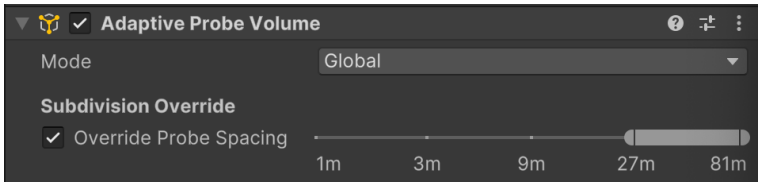
多くのシーンではこれで作業が完了し、休憩できるでしょう。しかし、APV を使うとさらに高い忠実度が得られます。異なるサブディビジョン を持つ複数のボリュームを追加して、プローブの配置と密度を正確にコントロールできます。

URP 3D サンプルのオアシス環境を例に考えてみましょう。シーンのほとんどのアクションがテントの周りで起こっていると想定し、プローブのほとんどをテントの周りに配置したいとします。これを実現するには、次のようにします。

1. 「**Rendering**」>「**Lighting**」>「**Adaptive Probe Volumes**」を開き、「**Max Probe Spacing**」を 81m に変更します。



2. **Global**」に設定した「**Adaptive Probe Volume**」を追加し、「**Override Probe Spacing**」を 27m ～ 81m に設定します。



3. 「**Local**」に設定したアダプティブプローブボリュームを追加し、「**Override Probe Spacing**」を 1m ～ 9m に設定します。ボリュームをテントよりも少し大きく設定します。

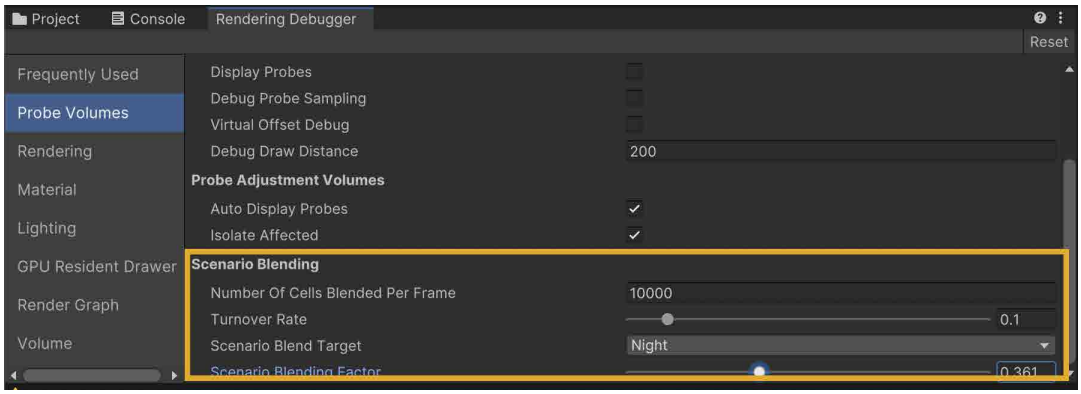
4. プローブボリュームをベイクします。

下の画像からわかるように、ほとんどのプローブはテントの周囲にあります。

プローブの配置

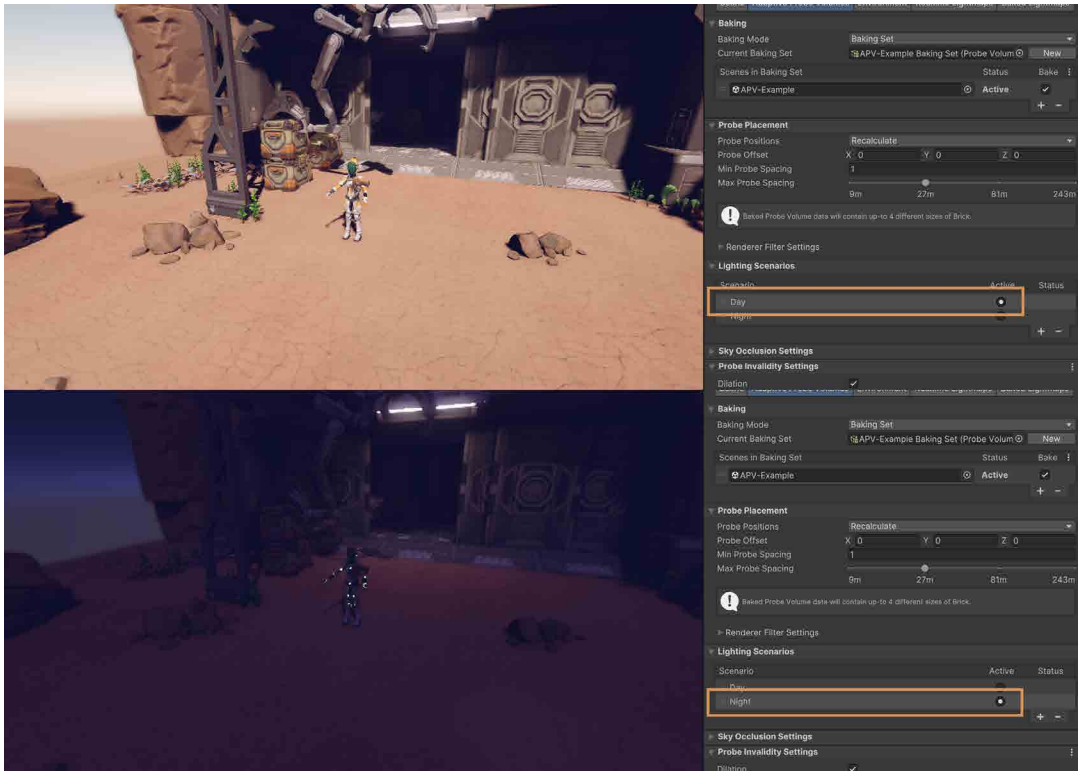
ライティングシナリオアセット

アダプティブプローブボリュームのもう一つの機能は、間接光データの切り替えです。**ライティングシナリオ**アセットには、シーンまたは**ベイク セット**のベイクされたライティングデータが含まれています。異なるライティング設定を異なるライティングシナリオにベイクし、ランタイムまたはデザイン時にレンダリングデバッガーを使用して、URP が使用するものを変更することができます。



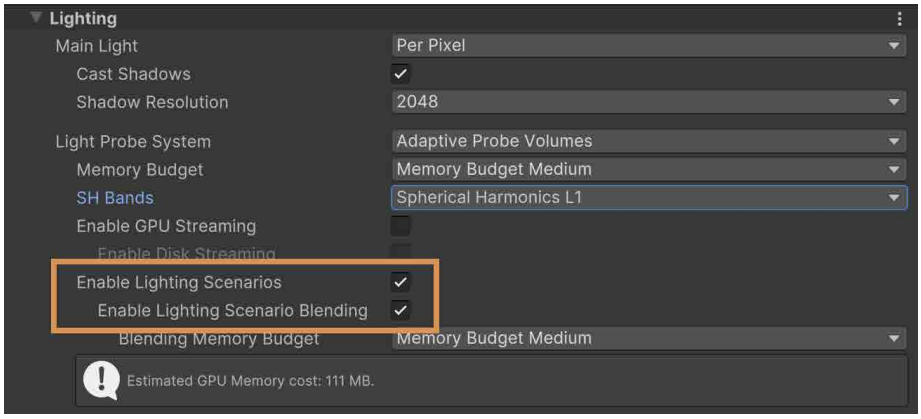
レンダリングデバッガーを使用したシナリオのブレンド

例えば、昼用と夜用の 2 つのライティングシナリオを作成できます。ランタイム時に、2 つのシナリオを切り替えたりブレンドしたりすることができます。



昼夜のライティングシナリオ

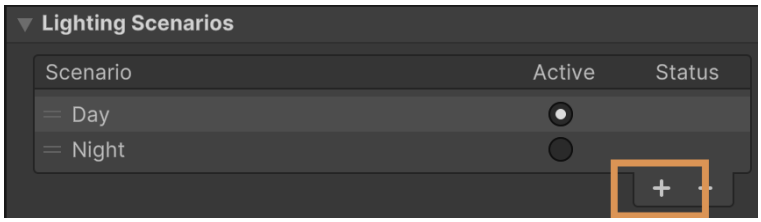
1. ライティングシナリオアセットを使用するには、アクティブな URP アセットに移動し、**「Lighting」 > 「Light Probe Lighting」 > 「Lighting Scenarios」**を有効にします。



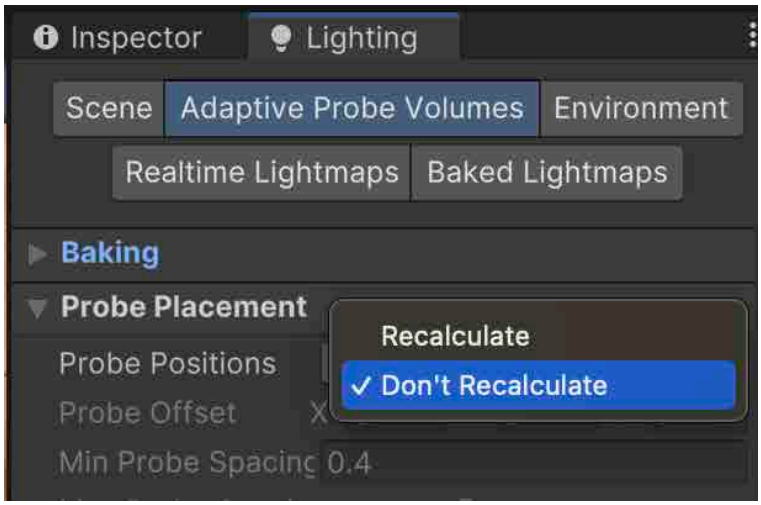
2. ベイク結果を保存できる新しいライティングシナリオアセットを作成するには、以下の手順に従います。

    a. **Lighting** ウィンドウの「**Adaptive Probe Volumes**」パネルを開きます。

    b. 「**Lighting Scenarios**」セクションで、「**Add**」(+) ボタンを選択して、ライティングシナリオアセットを追加します。



3. **Lighting** ウィンドウで、「**Adaptive Probe Volume**」タブの下にある「**Probe Positions**」が「**Don't Recalculate**」に設定されていることを確認します。これにより、Unity はプローブの位置を変更せずにライティングのみをリベイクします。プローブの位置が変更されると、以前にベイクされたシナリオが無効になる可能性があるためです。

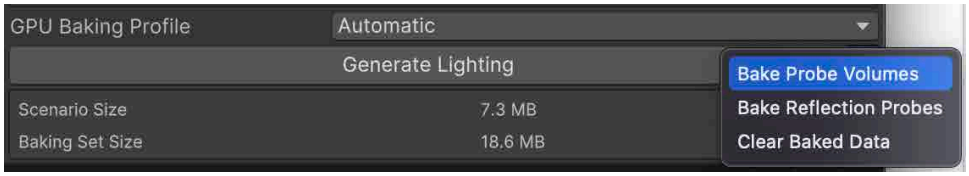


4. ライティングシナリオにベイクするには、以下の手順に従います。

    a. 「Lighting Scenarios」セクションで、ライティングシナリオを選択して有効化します。

    b. 「**Generate Lighting**」を選択します。URP は、アクティブなライティングシナリオにベイク結果を保存します。

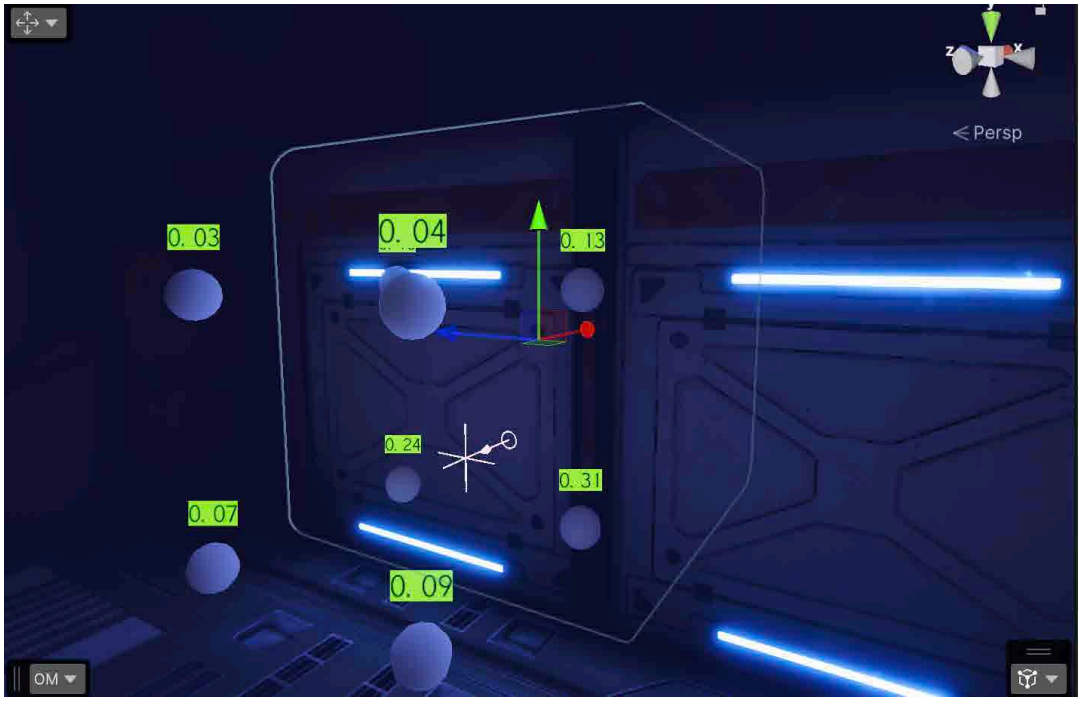
    c. ライトマップを使用していない場合は、「**Generate Lighting**」の隣にあるドロップダウンボタンを使用して、プローブのみに焦点を当てます。



ProbeReferenceVolume API を使用して、ランタイム時に URP が使用するライティングシナリオを設定できます。

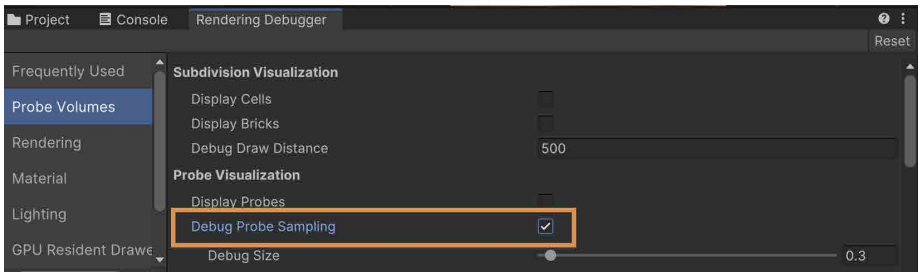
注：
実行時にアクティブなライティングシナリオを変更すると、URP はライトプローブ内の間接光データのみを変更します。ジオメトリの移動、ライトの修正、または直接光の変更には、スクリプトを使用する必要がある場合があります。

アダプティブプローブボリュームの問題の修正



プローブサンプリングのデバッグ

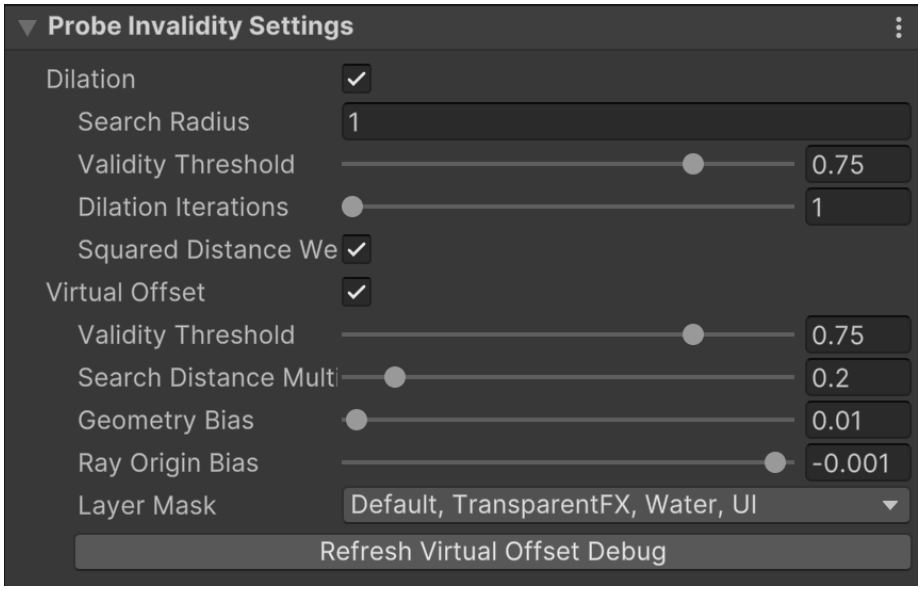
APV のアーティファクトなどの問題を修正するには、**「Window」>「Analysis」>「Rendering Debugger」>「Probe Volumes」>「Debug Probe Sampling」**を使用し、プローブが特定のピクセルでどのようにサンプリングされているかを確認してください。



ピクセルごとのプローブサンプリングの視覚化

ライトプローブはグリッドに追加されるため、配置によっては、明るいはずの場所が暗くなったり、その逆になったりと、レンダリングエラーが発生することがあります。エディターには、テクニカルアーティストがこれらの問題を迅速に修正するためのツールがいくつか用意されています。

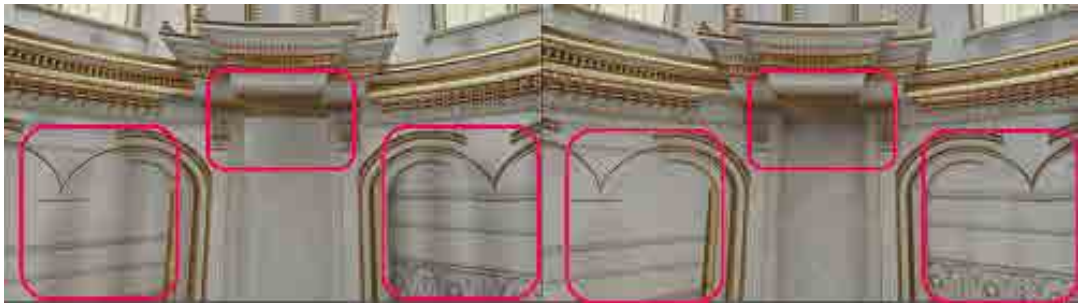
ジオメトリ内部のライトプローブは「無効なプローブ」と呼ばれます。URP は、周囲のライトデータを取得するためにサンプリングレイを発射しますが、そのレイがジオメトリ内部の光の当たらない裏面に当たると、そのプローブを無効としてマークします。APV システムには、これらの問題を修正するためのツールがいくつか用意されています。



「Adaptive Probe Volumes」パネルで利用できるプローブ無効化設定

Virtual Offset は、無効なライトプローブのキャプチャポイントをコライダーの外側に移動させることで、有効なプローブにしようとします。**Dilation** は Virtual Offset の適用後も無効なままのライトプローブを検出し、近くの有効なプローブからデータを補完します。

レンダリングデバッガーを使用して、どのライトプローブが無効か確認できます。

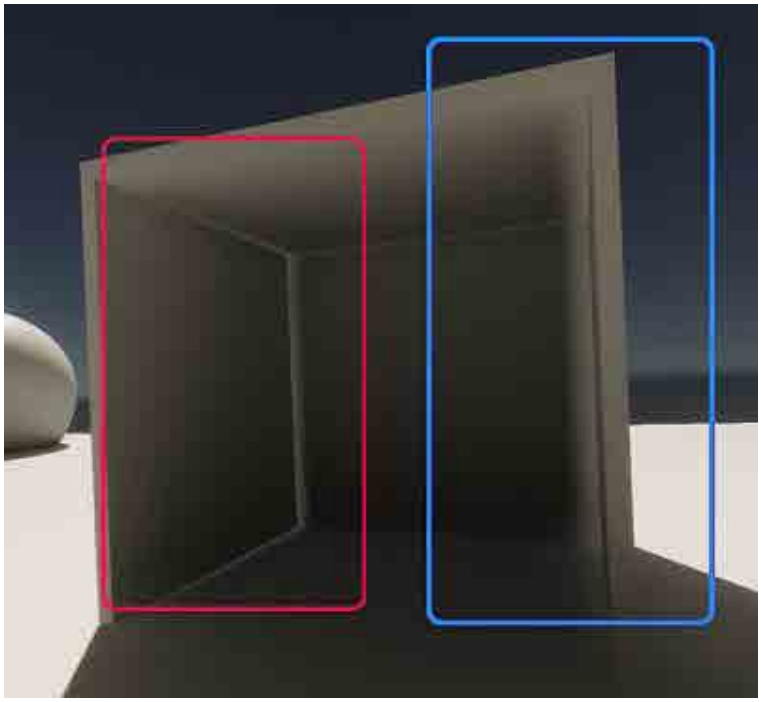


上の画像の左側のシーンでは、Virtual Offset が無効になっており、暗い帯が見えます。右側のシーンでは、Virtual Offset が有効になっています。

同様に、左側のシーンでは Dilation が無効になっており、一部の領域が暗すぎます。右側のシーンでは、Dilation が有効になっています。

ライトリーク

ライトリークとは、壁や天井のコーナーなどで発生する、明るすぎたり暗すぎたりする領域のことです。



ライトリーク

ライトリークは、通常ジオメトリが自身には見えていないライトプローブからの光を受け取ったときに発生します。例えば、ライトプローブが壁の反対側にある場合などです。APV はライトプローブを規則的なグリッドに配置するため、プローブが壁に沿わなかったり、異なるライティングエリアの境界に配置されないことがあります。

ライトリークを修正するには、以下の方法を試してください。

— より厚い壁を作成する。

— シーンに Adaptive Probe Volumes Options オーバーライドを追加する：

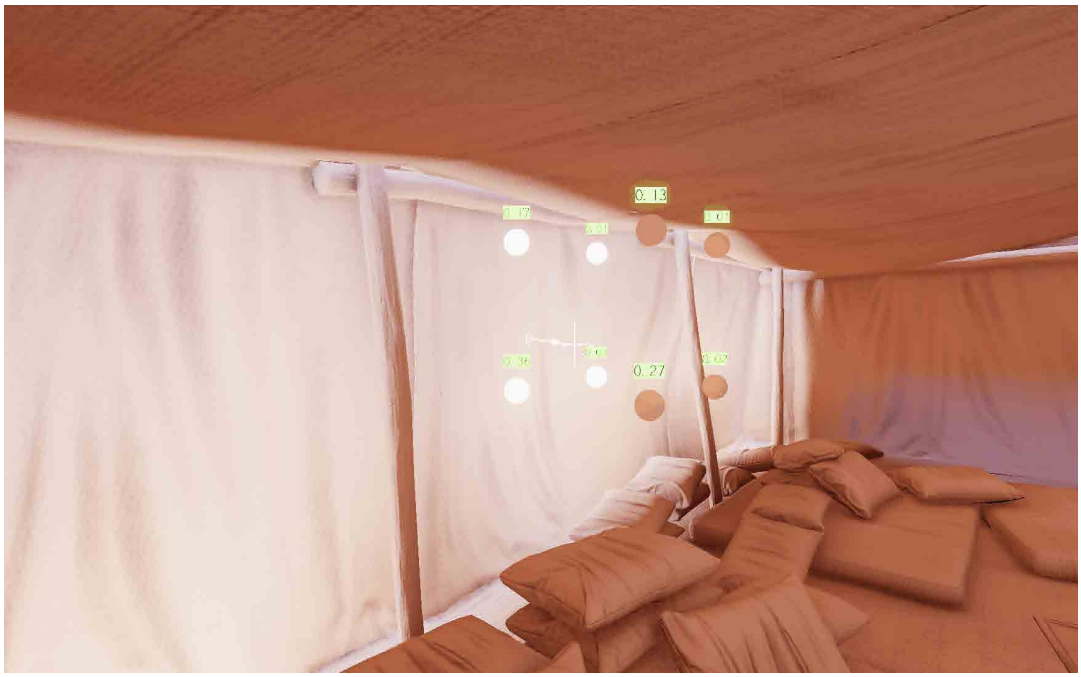
- Volume を追加し、それに **Adaptive Probe Volumes Options** オーバーライドを追加します。これにより、ゲームオブジェクトがライトプローブをサンプリングする位置を調整できます。
- レンダリング レイヤーを 有効にする：
    - Lighting ウィンドウの「Adaptive Probe Volumes」パネルで「**Rendering Layer Masks**」を設定し、APV が各ライトプローブにレンダリングレイヤーマスクを割り当てられるようにします。
- Baking Set プロパティを調整する：
    - ボリュームの追加が効果的でない場合は、Lighting ウィンドウの「Adaptive Probe Volumes」パネルを使用して、Virtual Offset や Dilation の設定を調整します。
- Probe Adjustment Volume コンポーネントを使用する：
    - このコンポーネントを使用して、小さなエリアでライトプローブを無効にします。これによりベイク時に Dilation がトリガーされ、実行時の **Leak Reduction Mode** の結果が改善されます。

レンダリングレイヤー

URP 3D サンプルのオアシス環境をライトプローブ / ライトマップから APV のみに切り替えると、ライトリークの問題が発生します。これは、以下の画像の明るい屋根や壁で確認できます。

URP 3D サンプルのオアシス環境のテント内で光が漏れている様子

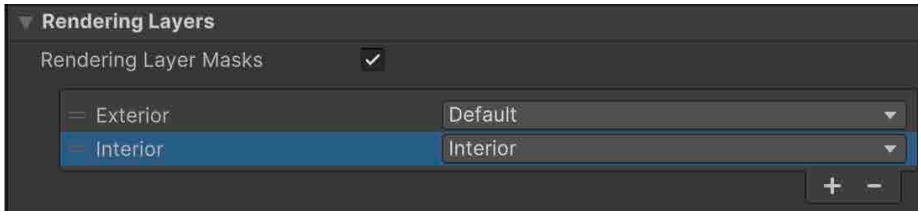
これは、テントの内側と外側のプローブ間でピクセルがブレンドされているためです。**「Window」> 「Analysis」> 「Rendering Debugger」> 「Probe Volumes」> 「Debug Probe Sampling」**を使用すると、ピクセルの値を補間する際にどのプローブが使用されているかを確認できます。



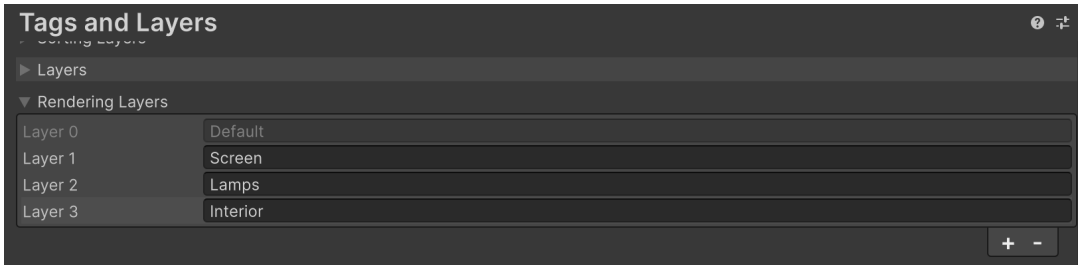
ピクセルの補間プローブの表示

この問題を解決する一つの方法は、**ボリューム**を使用し、**Adaptive Probe Volume Options** オーバーライドを介して、ランタイムで APV のサンプリング方法を変更することです。「**Normal Bias**」や「**View Bias**」の設定を調整して、サンプリング位置を調整してください。Normal Bias は法線に沿って（壁から離れる方向に）プッシュし、View Bias はカメラと同じ側に位置を保ちながら、カメラの方向にプッシュします。ボリューム内のこれらのプロパティを変更すると、ライティング結果と **Debug Probe Sampling View** の両方でリアルタイムに更新が確認でき、サンプリング位置とウェイトがそれに応じて更新されます。しかし、より良い解決策はレンダリングレイヤーを使用することです。

APV はレンダリングレイヤーをサポートしており、最大 4 つの異なるマスクを作成して、特定のオブジェクトに対してサンプリングをそれらのマスクに制限できます。これにより、内部のオブジェクトが外部のプローブをサンプリングするのを防ぐことができ、またその逆も防止できます。**「Window」> 「Rendering」> 「Lighting」> 「Adaptive Probe Volumes」> 「Rendering Layers」**でレンダリングレイヤーを有効にして追加できます。



また、**「Project Settings」>「Tags and Layers」>「Rendering Layers」**からレイヤーを追加する必要があります。

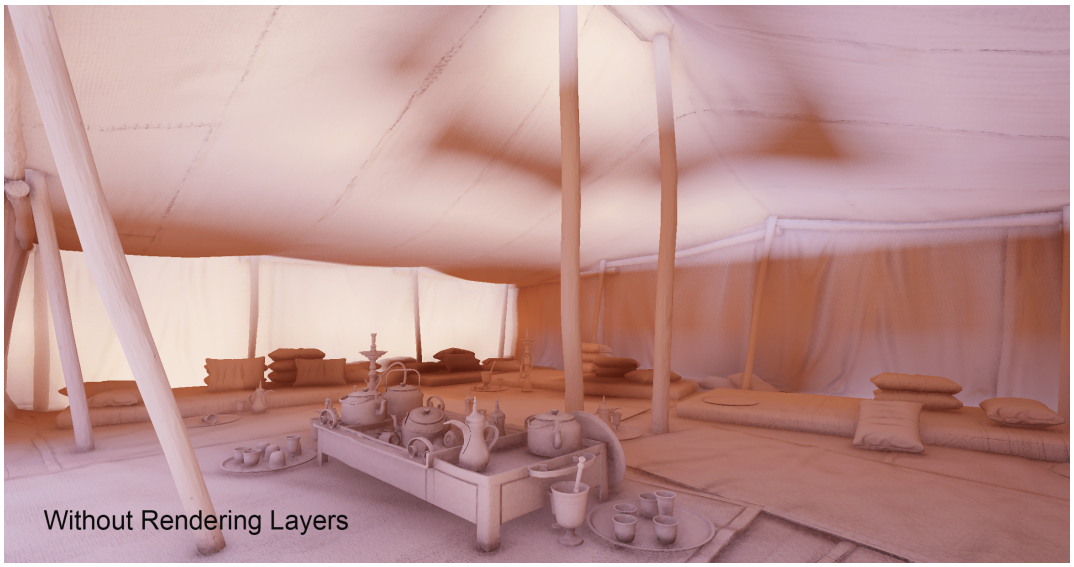


これを実装するには、メッシュ自体を編集して、作成したいエリアごとに分割されていることを確認します。例えば、このプロジェクトでは、メッシュを編集して、内部と外部を複数のメッシュに分割しています。メッシュを分割したら、適切なレンダリングレイヤーを割り当て、**「Adaptive Probe Volumes」タブ**で APV で使用するものを指定します。

テント内のすべてのオブジェクトにレイヤーを割り当てる必要はなく、壁や壁の近くのオブジェクトなど、リークが発生しやすいオブジェクトにのみ割り当てます。

ライティングを生成する際、システムはベイク時に近隣のオブジェクトに基づいて自動的にプローブにレイヤーを割り当てます。これにより、プローブごとに手動でレイヤーを割り当てる必要がなくなります。この自動プローブ割り当てを容易にするために、大きなオブジェクトにレイヤーを割り当てます。オアシス環境のテントの例では、テントの壁と天井に内側のレイヤーを割り当てることで、ベイク時に内側のプローブのほとんどがそれらに当たり、自動的に内側マスクに割り当てられるようにしています。プローブは、最も頻繁に接触するレイヤーに割り当てられます。

これが完了したら、**「Generate Lighting」**をクリックし、内部と外部のマスクを分けたことで、テントのリークが解消されていることを確認します。



レンダリングレイヤーの有無によるライトリークの差

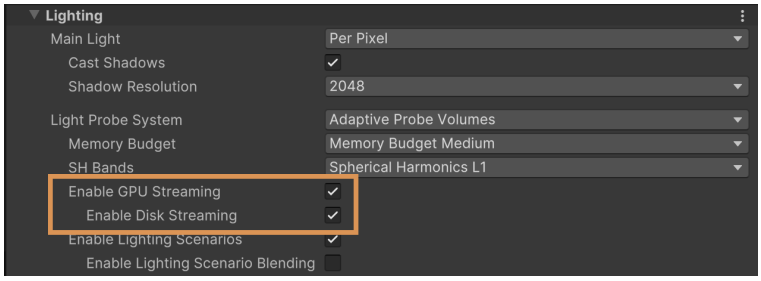
APV に関する問題の修正について、詳しくはこちらをご確認ください。

APV のストリーミング

APV ストリーミングを使用すると、大規模なワールドで APV ベースのライティングを利用できます。APV ストリーミングは、CPU または GPU のメモリ容量を超える APV データをベイクし、実行時に必要に応じてロードします。実行時にカメラが移動すると、URP はカメラの視錐台内のセルからのみ APV データをロードします。

URP の品質レベルに合わせて、ストリーミングを有効または無効にできます。ストリーミングを有効にするには、以下の手順に従います。

1. メインメニューから**「Edit」>「Project Settings」>「Quality」**を選択します。
2. 品質レベルを選択します。
3. 「Render Pipeline Asset」をダブルクリックして、Inspector で開きます。
4. 「Lighting」タブを展開します。
5. ここで、以下の 2 種類のストリーミングを有効にできます。
    a. 「Disk Streaming」を有効にして、ディスクから CPU メモリにストリーミング。
    b. 「GPU Streaming」を有効にして、CPU メモリから GPU メモリにストリーミング。先に Disk Streaming を有効にする必要があります。

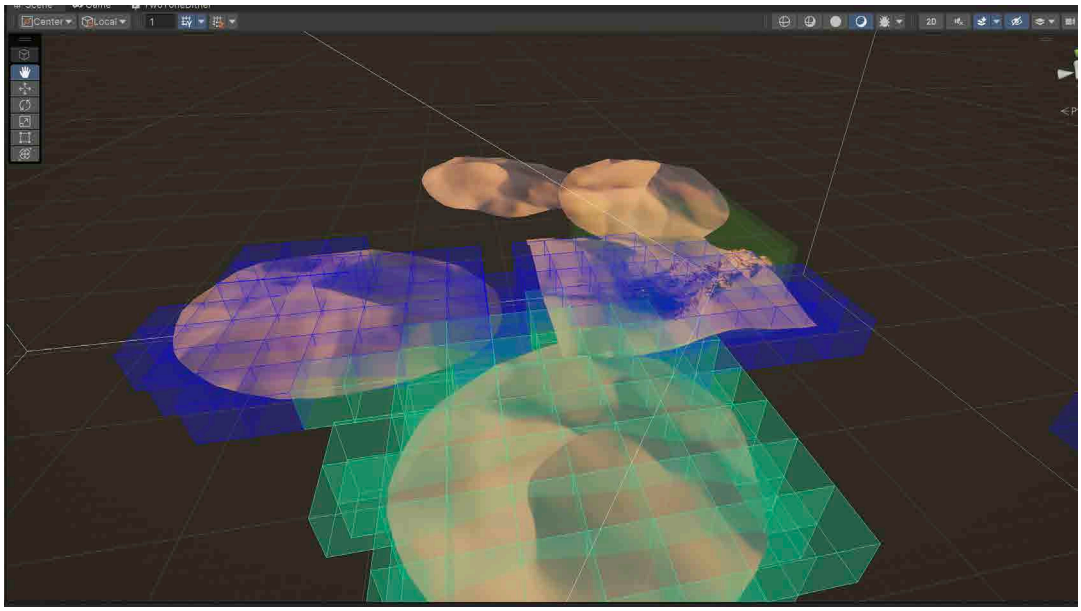


ストリーミング設定は同じウィンドウ内から可能です。詳細については、URP アセットを参照してください。

ストリーミングのデバッグ

URP がロードして使用する最小のセクションはセルで、これは APV 内の最大のブリックと同じサイズです。ライトプローブの密度を調整することで、APV 内のセルのサイズに影響を与えることができます。

APV 内のセルを表示したり、ストリーミングをデバッグするには、レンダリングデバッガーを使用してください。



APV ストリーミング

空のオクルージョン

空のオクルージョンとは、ゲームオブジェクトが空から色をサンプリングする際に、光がそのオブジェクトに届かない場合、Unity がその色を減衰させる処理のことです。Unity の空のオクルージョンでは、実行時に更新されるアンビエントプローブからの空の色が使用されます。これにより、空の色の変化に合わせて動的にゲームオブジェクトをライティングできます。例えば、空の色を明るい色から暗い色に変化させて、昼夜のサイクルをシミュレートできます。

注：
空のオクルージョンを有効にすると、APV のベイクに時間がかかったり、実行時に Unity がより多くのメモリを使用する可能性があります。

空のオクルージョンを有効にすると、Unity は APV 内の各プローブに追加の静的な空のオクルージョン値をベイクします。空のオクルージョン値は、静的なゲームオブジェクトからの反射光も含め、空からプローブが受け取る間接光の量です。

空のオクルージョンを使用する主な利点は、実行時に空のライティングを変更できることです。

これを説明するために、左側の一連の画像を見てみましょう。

a. 上の画像は、実行時に変化するため空のライティングをベイクできない場合に発生する問題を示しています。この画像では、アンビエントプローブのみを使用し、ベイクを行っていないため、仕上がりが良くありません。

b. 2 番目から 5 番目の画像では、アンビエントプローブを空のオクルージョンと併用しています。この画像に対して、空のオクルージョンを無効にし、通常の APV ベイクでライティングすることもできますが、その場合、ライティングは実行時に変化しません。

シーンに空のオクルージョンを適用した結果の一例この画像は、7stars による Unity Asset Store パッケージ「Azure[Sky] Dynamic Skybox」から取得したもの。

空のオクルージョンを有効にするには、以下の手順に従います。

1. 「**Progressive GPU Lightmapper**」を有効にします。Progressive CPU を使用している場合、Unity は空のオクルージョンをサポートしません。**「Window」>「Rendering」>「Lighting」**に移動します。
2. 「Scene」パネルに移動します。
3. 「Lightmapper」を「Progressive GPU」に設定します。

4. 「Adaptive Probe Volumes」パネルを開きます。
5. 「Sky Occlusion」を有効にします。

ライティングデータを更新するには、空のオクルージョンを有効化または無効化した後に APV をベイクする必要があります。空のオクルージョンをベイクすると、シーンのライティングにアンビエントプローブの更新が反映されます。URP では、スカイボックスモードではなく、カラーモードまたはグラデーションモードを使用している場合のみ、アンビエントプローブがリアルタイムで更新されます。したがって、アニメーション化された空のビジュアルに一致するように、色を手動でアニメーション化する必要があるかもしれません。

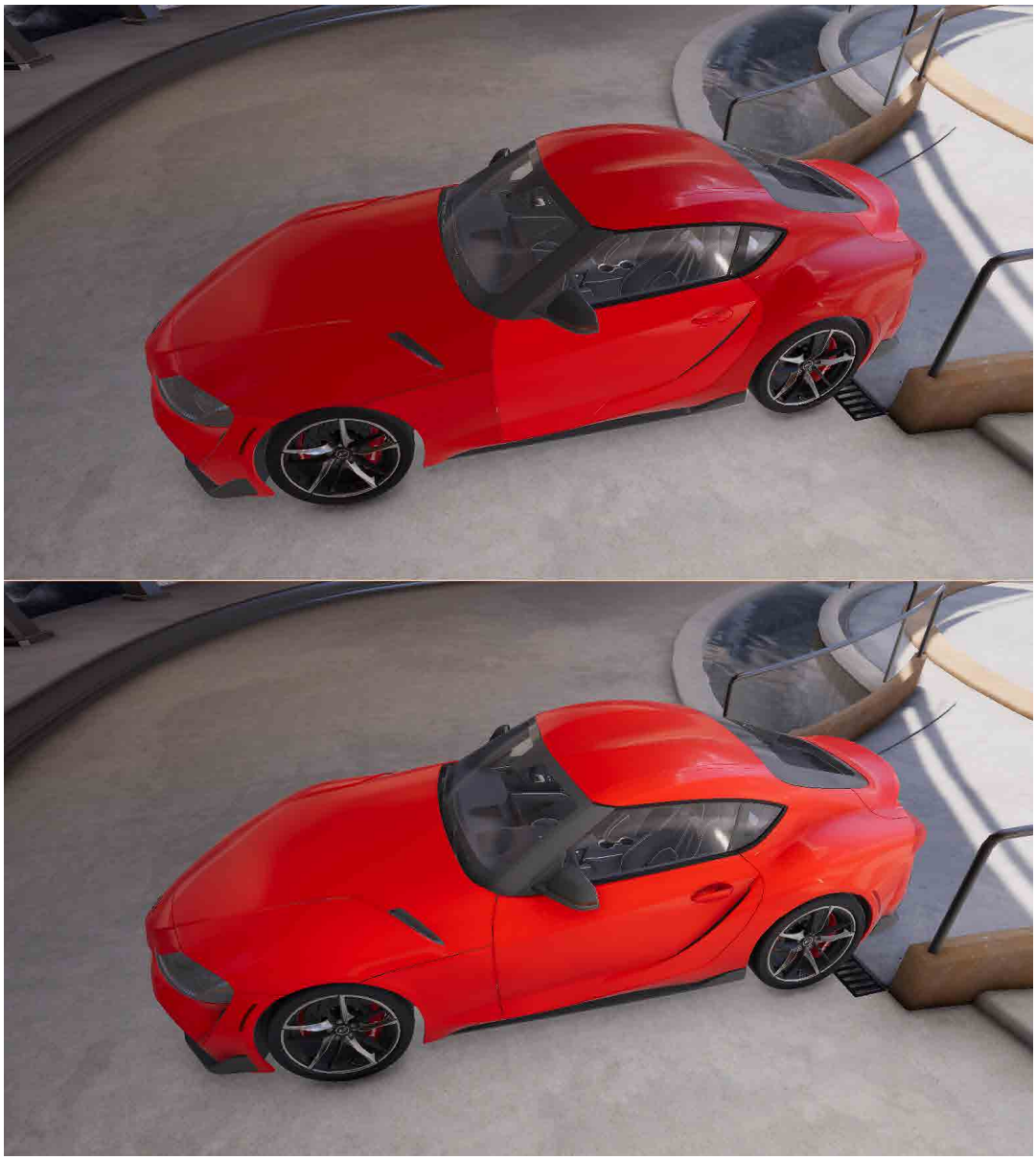
注：
URP では、プローブの頂点ごとの高品質サンプリングがサポートされるようになりました。これは、特にローエンドデバイスでパフォーマンスを向上させるのに役立ちます。サンプリングモードを設定するには、「Lighting」セクションの **URP アセット**を使用します。オプションを表示するには、**Advanced Properties** を有効にする必要があります。「Lighting」パネルの右上にある省略記号を押すと有効化できます。Advanced Properties がアクティブになると、「**SH Evaluation Mode**」のドロップダウンが表示されます。

詳細情報

— アダプティブプローブボリュームのドキュメント

— GDC 2023 のセッション：Adaptive Probe Volumes を用いた効率的でインパクトのあるライティング

ライトプローブと APV の比較



上の画像ではライトプローブグループ、下の画像では APV が使用されている。画像は ArchVizPro による Unity Asset Store パッケージ「ArchVizPRO Photostudio URP」のもの。

下の画像（上に掲載）は、APV を使用した場合の暗から明へのスムーズなトランジションを示しています。上の画像では、ライトプローブグループの影響で、オブジェクトごとに単一の補間プローブが使用されるため、車のドアに明るい光が当たっています。これは、ドアが他の部分とは別のゲームオブジェクトであり、異なるプローブを使用しているためにレンダリングエラーが発生しているからです。

以下の表は、ライトプローブと APV の機能を比較したものです。

ライトプローブグループ	アダプティブプローブボリューム
ジオメトリが変更された場合、プローブの配置や移動に時間がかかる	配置が迅速で、ジオメトリが変更されても更新が容易
ライティングオブジェクトに対して単一の補間プローブが使用される： — オブジェクトが暗い場所から明るい場所にかけてスムーズに変化できず、不自然に浮いて見える。 — 大きなオブジェクトで問題が発生する可能性がある。	各ピクセルが個別にライティングされる： — スムーズなトランジションが実現。 — APV のグリッド構造により任意の場所を簡単にサンプリングできるため、APV を使うことで優れたボリュメトリック効果が得られる。
静的オブジェクトは通常ライトマップを使用する。プローブを使用するのは動的なオブジェクトのみ。	ライトマップやライトマップ UV は不要。 — シーン内のすべてのオブジェクトに対して単一のライティングソリューションを使用。 — メモリ使用量を抑えつつ、広大な世界にライティングを適用。
ランタイム時にプローブを自由に配置および移動可能	プローブはグリッド構造で配置され、ランタイム時に移動不可。
スイッチ GI には対応していない。	ライティングシナリオアセットでは、異なるライティング（例：昼から夜への切り替え、ライトのオン / オフなど）を切り替えることが可能。

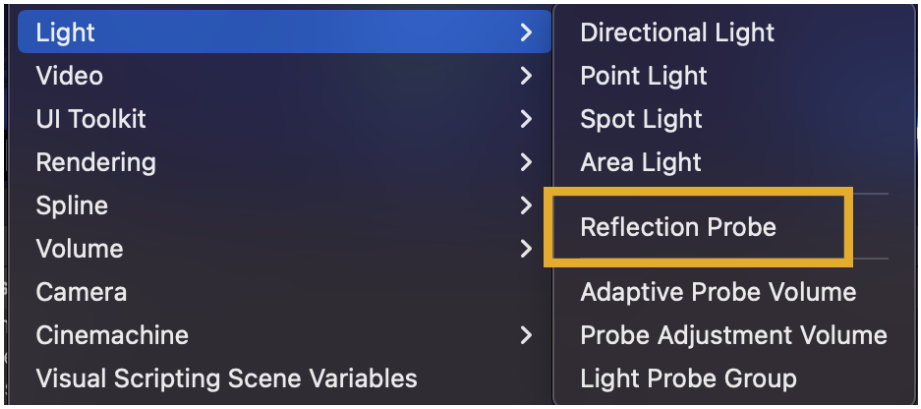
リフレクションプローブ

Maya や Blender などのレイトレーシングツールでは、反射面の各フレームピクセルの値を正確に計算するのに時間がかかることがあります。このプロセスはリアルタイムレンダラーでは時間がかかりすぎるため、ショートカットがよく使用されます。

リアルタイムレンダラーでの反射には、環境マップ（事前レンダリングされたキューブマップ）が使用されます。Unity では、SkyManager を使用してデフォルトマップを提供しています。単一のマップをシーン内のすべての場所からの反射のソースとして使用すると、不自然な反射が発生する可能性があります。このセクションで示したロボットの例で考えてみましょう。このキャラクターの金属部分に常に空を反射させた場合、空が見えない格納庫の中では、とても奇妙な見た目になります。そのような場合に、リフレクションプローブが役立ちます。

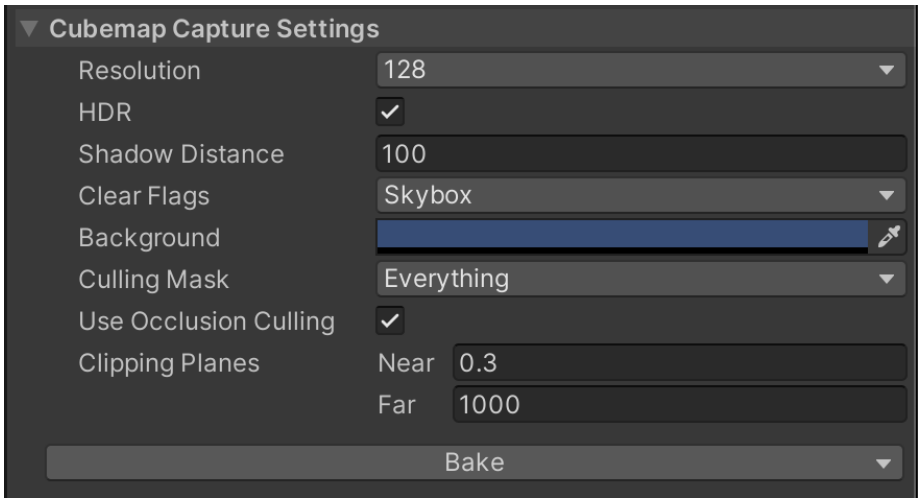
リフレクション プローブ コンポーネントは、シーン内のキー位置に配置される、事前レンダリング済みのキューブマップです。リフレクションプローブは 1 つのシーンで複数使用できます。動的オブジェクトがシーン内を移動する際に、最も近いリフレクションプローブを選択し、それを反射の基準として使用することができます。また、プローブ間をブレンドするようにシーンを設定することもできます。

リフレクションプローブを追加するには、**Hierarchy** ウィンドウを右クリックし、**「Light」>「Reflection Probe」**を選択します。



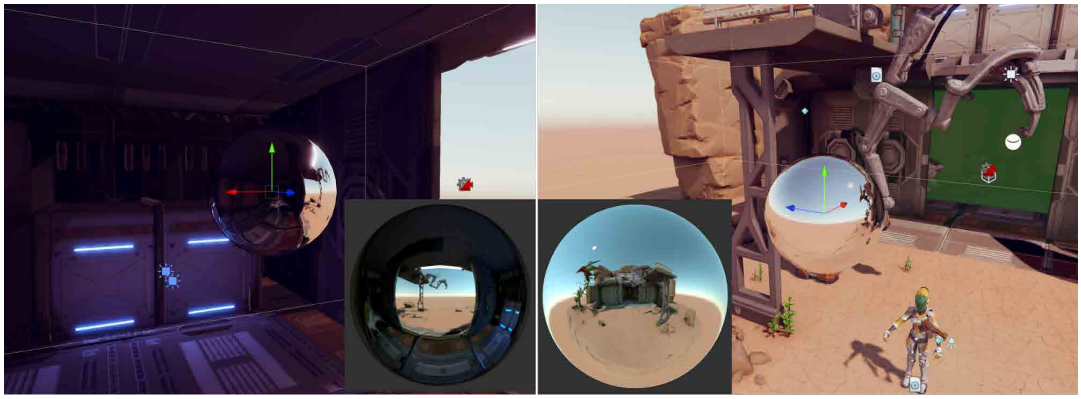
Reflection Probe コンポーネント

次に、プローブの位置を決め、設定を調整します。プローブが正しく配置され、設定の調整が済んだら、**「Bake」**をクリックしてキューブマップを生成します。



Reflection Probe コンポーネントの設定

以下の画像は、FPS サンプル「The Inspection」で使用されている 2 つのリフレクションプローブを示したものです（1 つは格納庫の中、もう 1 つは外）。



各リフレクションプローブでは、その周囲の画像をキューブマップテクスチャに取り込みます。

リフレクションプローブのブレンディング

ブレンディングはリフレクションプローブの優れた機能で、「**Renderer Asset Settings**」パネルから有効化できます。フォワード + パスを選択すると、レンダラーアセットの設定に関係なく、ブレンディングが常にオンになります。

ブレンディングでは、反射オブジェクトが 1 つのゾーンから別のゾーンへと移動していくときに、一方のプローブのキューブマップを徐々にフェードアウトしながら、もう一方のプローブへとフェードインしていくことができます。この段階的な移行により、2 つのリフレクションプローブの境界を通過するオブジェクトが、突然まったく異なる反射を持たないようにします。

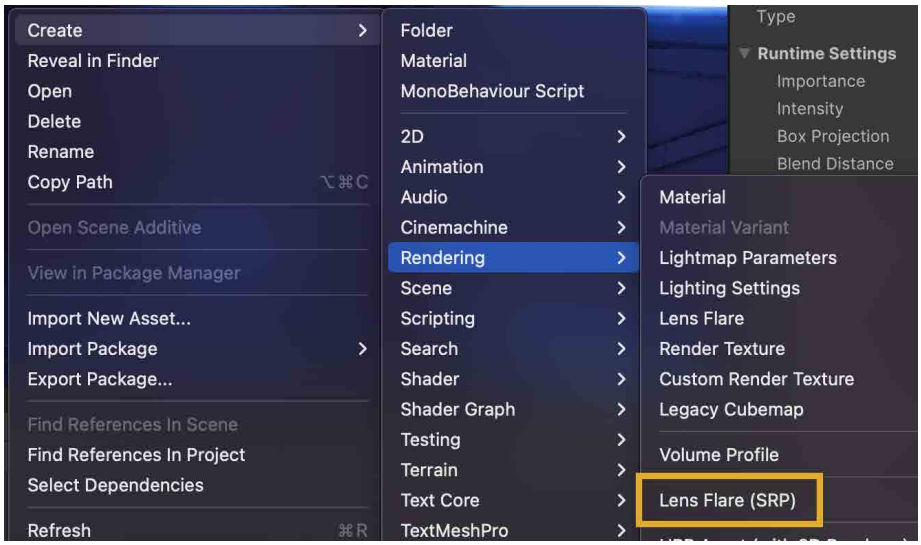
Box Projection

通常、リフレクションキューブマップは所定のオブジェクトから無限の距離にあると想定されます。オブジェクトが回転すると、キューブマップの別の角度が表示されますが、反射されている周囲環境に対してオブジェクトが近づいたり遠のいたりすることはできません。これは屋外のシーンでは良好に機能してくれるものの、屋内のシーンではその限界が出てしまいます。部屋の内壁は明らかに無限の距離にはないので、オブジェクトが近づくほど壁の反射が大きくならないと不自然です。

「Box Projection」オプションを使用すると、プローブから有限距離の反射キューブマップを作成できます。これにより、キューブマップの壁からの距離に応じて、オブジェクトにさまざまなサイズの反射を映すことができます。周囲のキューブマップのサイズは、プローブの影響ゾーンによって決まります（「Box Size」プロパティに基づきます）。例えば、部屋の内部を反映するプローブを使う場合は、部屋の寸法に合わせてサイズを設定する必要があります。

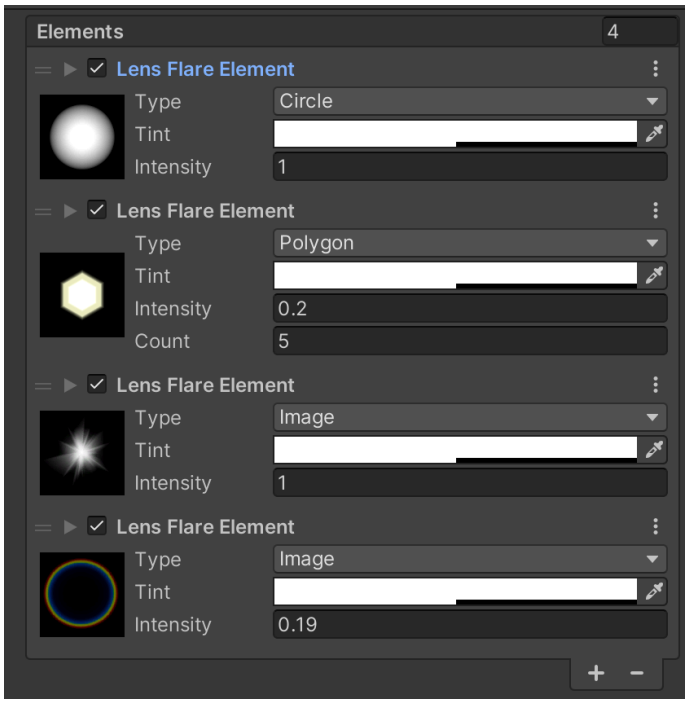
レンズフレア

ビルトインレンダーパイプラインを使用していた場合、URP ではレンズ フレアの作成ワークフローが更新されています。設定の最初のステップは、レンズフレア（SRP）データアセットを作成することです。**Project** ウィンドウ内の適切な Assets フォルダーで右クリックし、**「Create」>「Rendering」>「Lens Flare (SRP)」**を選択します。



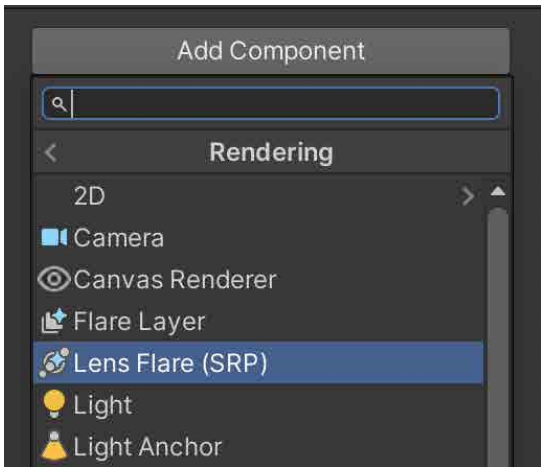
レンズフレア（SRP）データアセットの作成

このアセットを使用して、「**Type**」を Circle、Polygon、または Image アセットに設定し、「Tint」と「Intensity」の設定を調整することで、フレアの形状を設定します。



レンズフレア要素の追加と設定

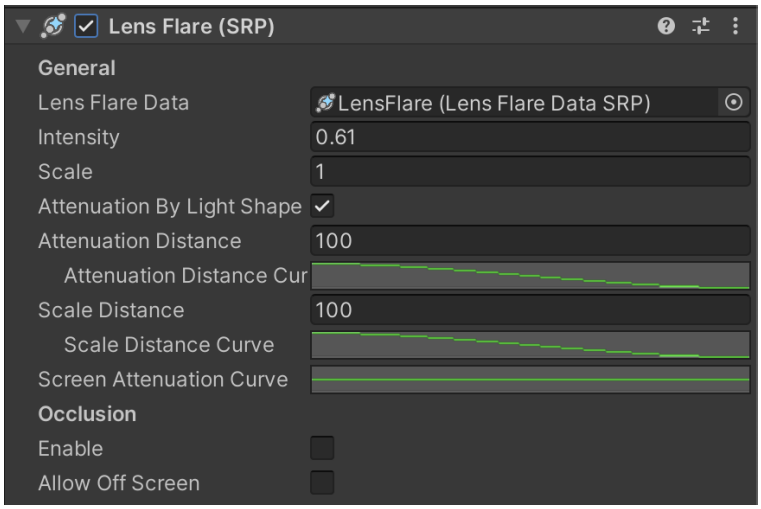
レンズフレアをレンダリングするには、フレアの原因となる光源を選択してから、**「Add Component」> 「Rendering」> 「Lens Flare (SRP)」**を選択します。



レンズフレアのレンダリングの設定

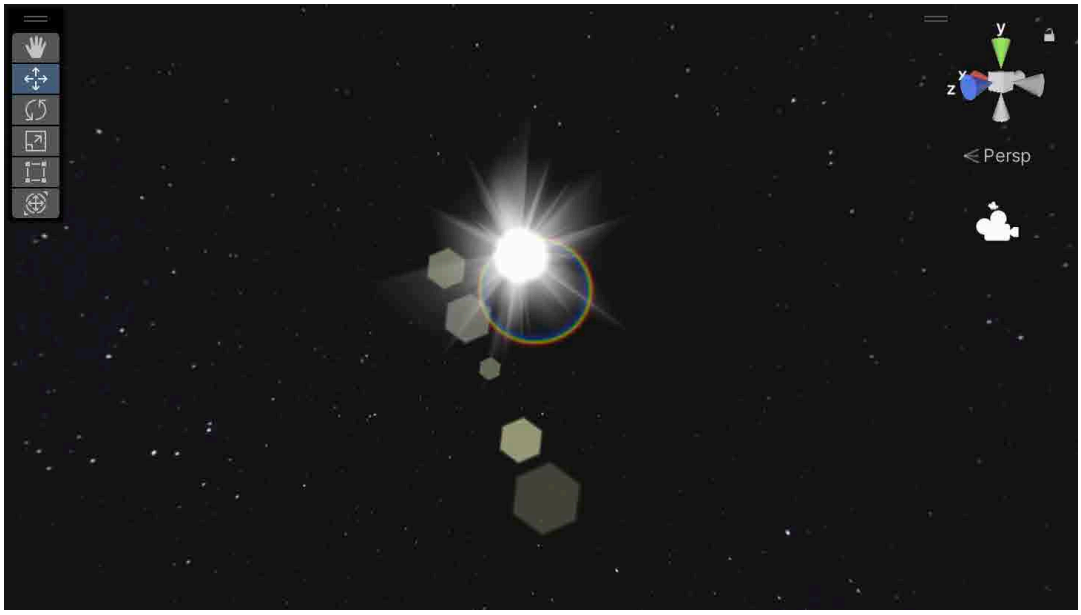
レンズフレアデータアセットを選択します。

このコンポーネントの「**Settings**」パネルで、作成した**レンズフレアデータアセット**を**レンズフレアデータプロパティ**に割り当てます。



レンズフレア（SRP）コンポーネントの設定

レンズフレアを使用すると、追加や調整のワークフローが非常に柔軟であることがわかります。



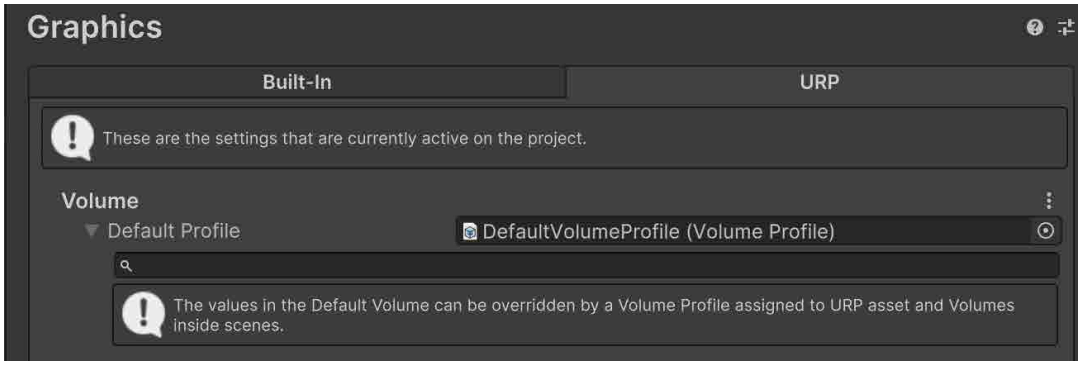
レンズフレアの例

スクリーンスペースレンズフレア

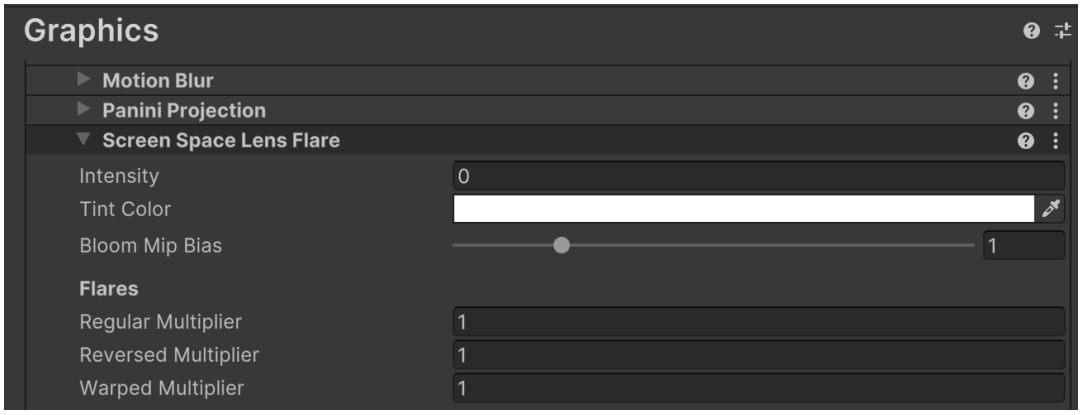
複数のライトに対してレンズフレアを設定するには時間がかかります。Unity 6 では、新しいポストプロセス技術として、Screen Space Lens Flare オーバーライド（SSLF）が導入されました。この技術では、明るいスペキュラーハイライトや発光メッシュなど、明るい表面からフレアを生成できます。これに対して、レンズフレア (SRP) エフェクトは、ライトからのみフレアを生成します。スクリーンスペースレンズフレアは、ポストプロセス技術を使用します。

SSLF を使用するには、以下の手順に従います。

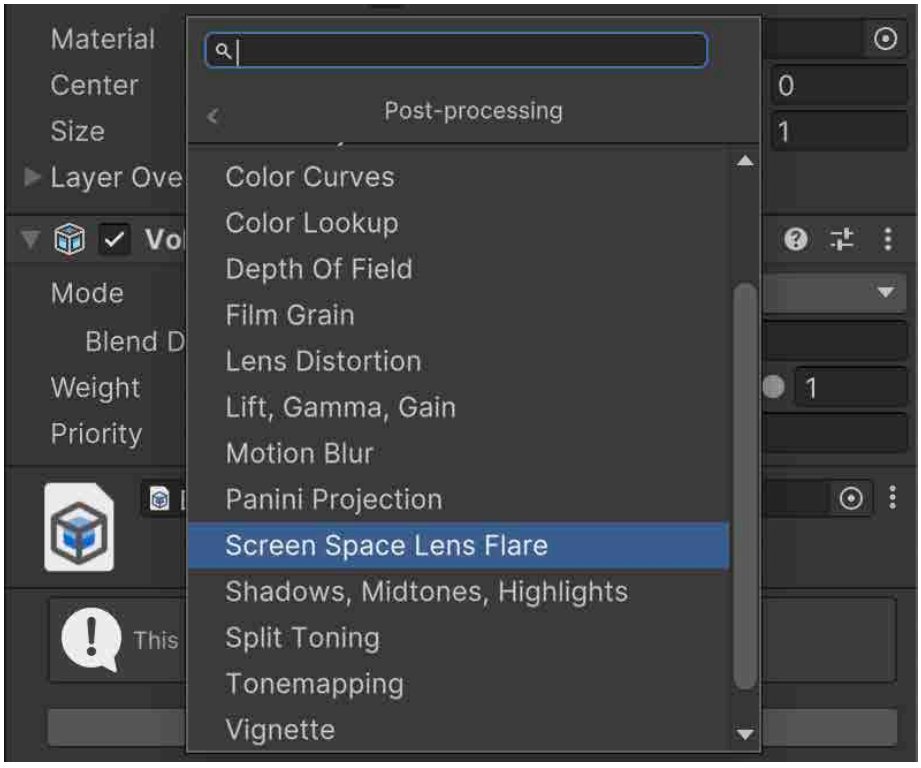
1. SSLF はポストプロセスフィルターであるため、適用するにはボリュームコンポーネントを使用します。シーンにボリュームコンポーネントを追加するか、Unity 6 の新しいデフォルトボリュームを使用します。デフォルトボリュームの設定は、**「Edit」>「Project Settings」>「Graphics」**で調整します。



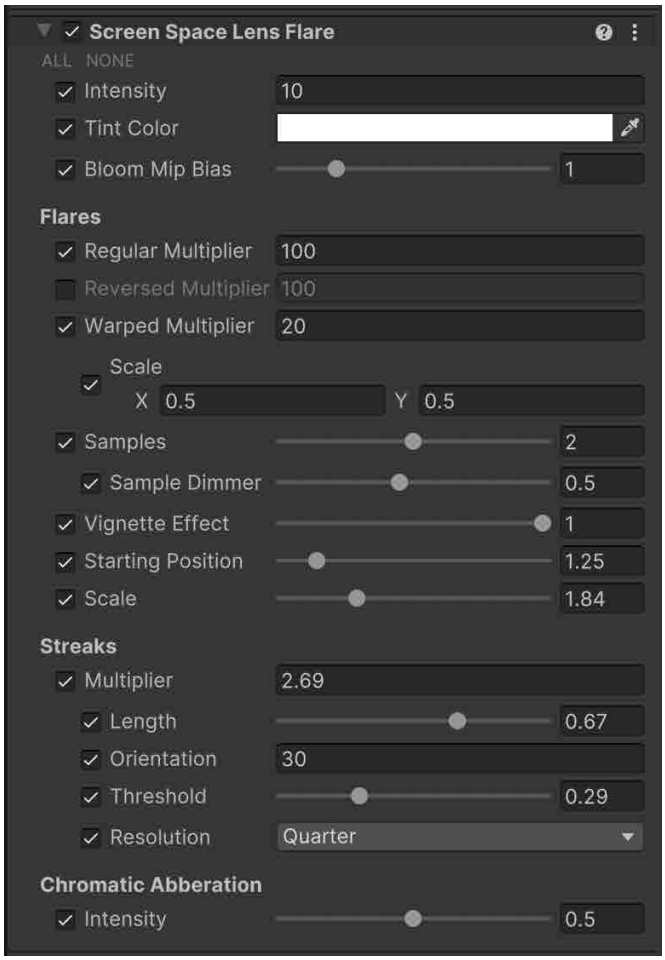
2. デフォルトボリュームを使用する場合は、設定パネルで Screen Space Lens Flare のオーバーライドを探します。



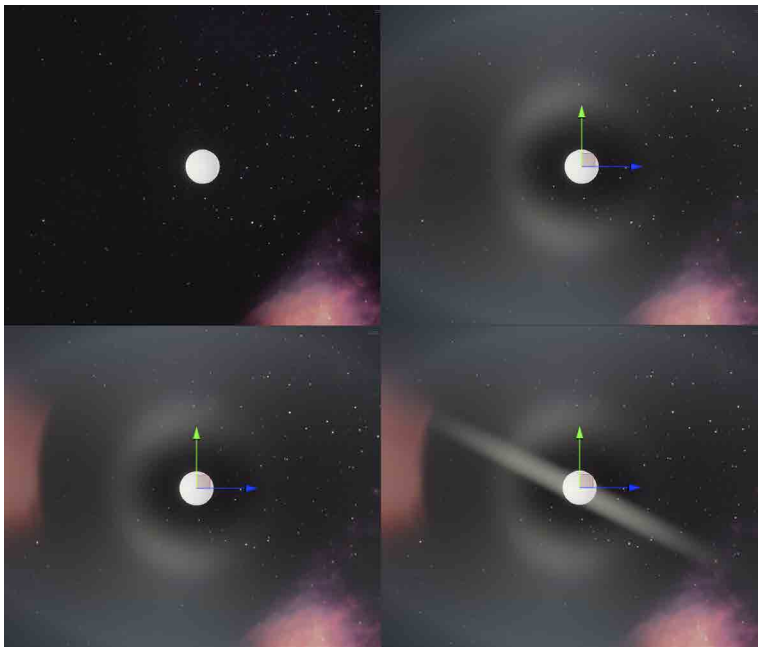
3. シーンのボリュームを使用する場合は、オーバーライドを追加できるようにボリュームプロファイルを作成します。その後、**「Post-processing」>「Screen Space Lens Flare」**からオーバーライドを追加します。



4. これで、設定をいろいろと試して、シーンで希望するスタイルを実現できます。強度を 0 以上に設定する必要があることに注意してください。



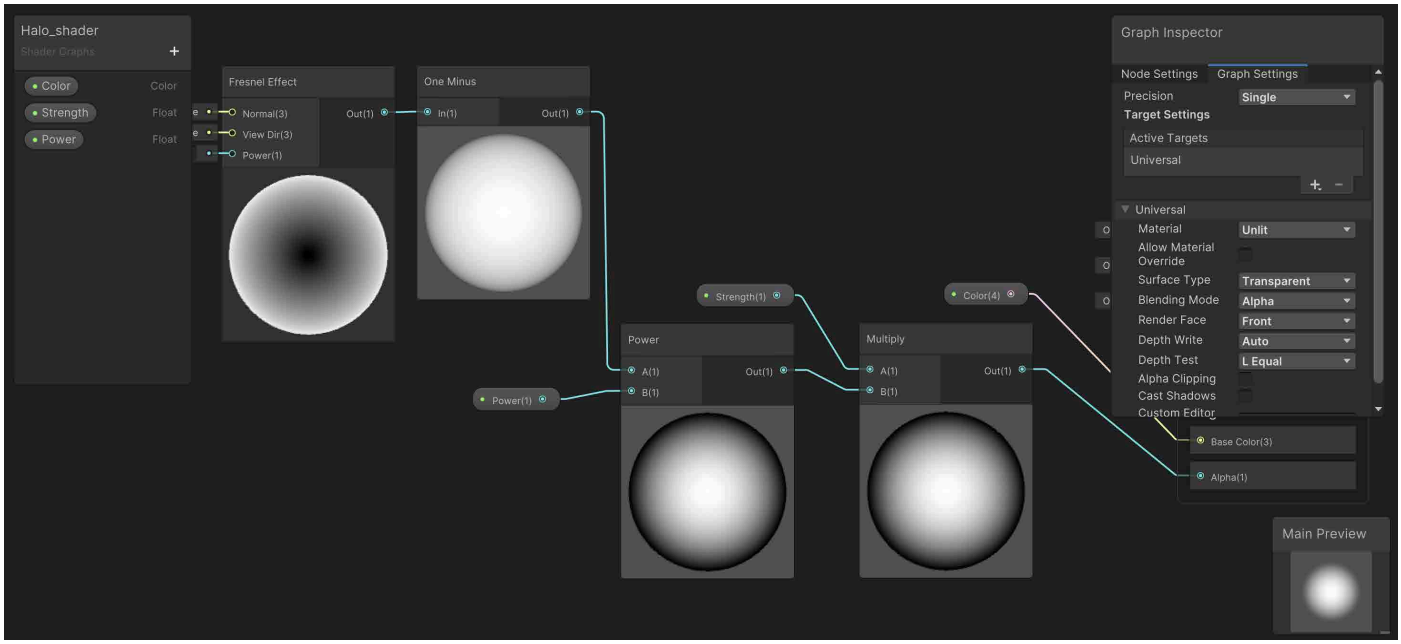
さらに制御を行うには、SSLF をレンズフレア（SRP）コンポーネントと組み合わせることができます。



左上から時計回り：SSLF なし。フレア。フレア、歪んだフレア、光の筋。フレア、歪んだフレア

ライトハロー

URP では、ライトに対して「**Draw Halo**」プロパティは使用できませんが、ビルボードで簡単に模倣できます。もう 1 つのオプションは、スフィアのアルファ透明度を設定することです。下の 1 番目の画像はそのようなシェーダーの Shader Graph を示しており、2 番目の画像はその結果を示しています。Shader Graph を使用してこのシェーダーを作成する方法の詳細については、追加ツールの章を参照してください。

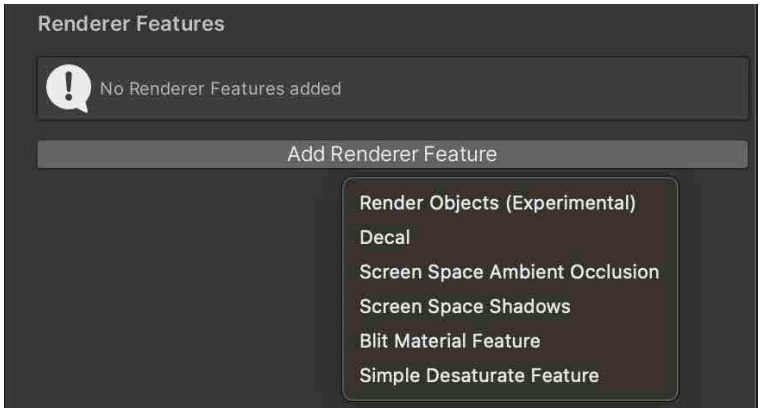


Shader Graph を使用したフレネル透明度

前述の Shader Graph シェーダーを適用したマテリアルを持つスフィアを使用したハロー効果

Screen Space Ambient Occlusion

アンビエントライトは、デフォルトではジオメトリを考慮しないため、アンビエントライトが強すぎると現実味を欠くレンダリングになることがあります。現実世界では、2 つのオブジェクト間の隙間は、狭い方が広い場合よりも暗くなる可能性が高いです。アンビエントオクルージョンは、Unity プロジェクトでこの問題に対処するのに役立ちます。URP で使用するには、URP アセットが使用しているレンダラーを選択します。「**Add Renderer Feature**」に移動し、「**Screen Space Ambient Occlusion**」(SSAO) を選択します。



Add Renderer Feature から SSAO を選択

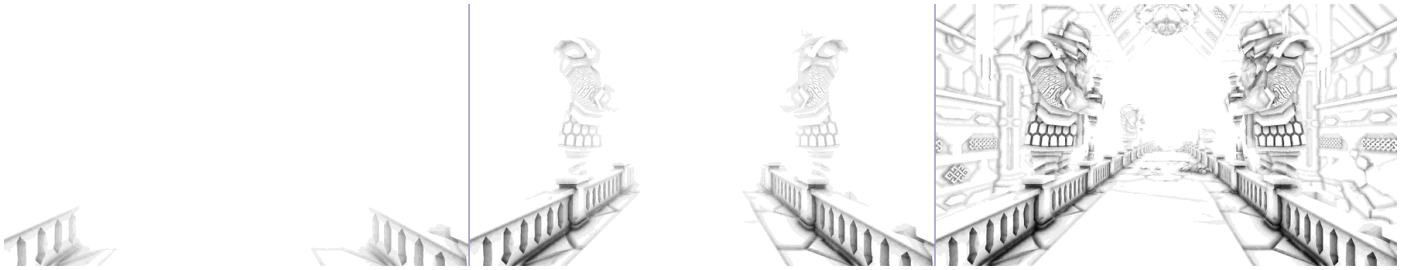
デフォルトの SSAO 設定を使用するか、必要に応じて調整してください。

SSAO 設定

SSAO のプロパティを見てみましょう。

- **Method**：このプロパティは、SSAO エフェクトが使用するノイズのタイプを定義します。
- **Intensity**：このプロパティは、暗くする効果の強度を定義します。
- **Radius**：SSAO エフェクトは、Unity がアンビエントオクルージョン値を計算する際、現在のピクセルからこの半径内にある法線テクスチャのサンプルを取得します。Radius 値を低く設定すると、SSAO Renderer Feature がソースピクセルに近いピクセルをサンプリングするため、パフォーマンスが向上します。

- **Falloff Distance**：カメラから指定距離以上離れたオブジェクトには、SSAO を適用しません。低い値は、遠くにあるオブジェクトが多く含まれるシーンのパフォーマンスを向上させます。
- **Direct Lighting Strength**：このプロパティは、直接光に露出された領域で効果がどの程度見えるかを定義します。
- **Quality**：これらの品質設定の詳細については、ドキュメントを参照してください。
  - Source
  - Downsample
  - After Opaque
  - Blur Quality
  - Samples



アンビエントオクルージョンテクスチャのみを適用したシーンで、異なるフォールオフ距離を使用した結果を示している

SSAO は、狭い隙間にシェーディングを追加します。左の 3 つの画像を見てみましょう。

上の画像では、SSAO は適用されていません。真ん中の画像は、計算された SSAO、下の画像は SSAO の適用結果が表示されています。グラインダーおよび秤と机の接触部分のエッジが、より強くなっていることに注目してください。

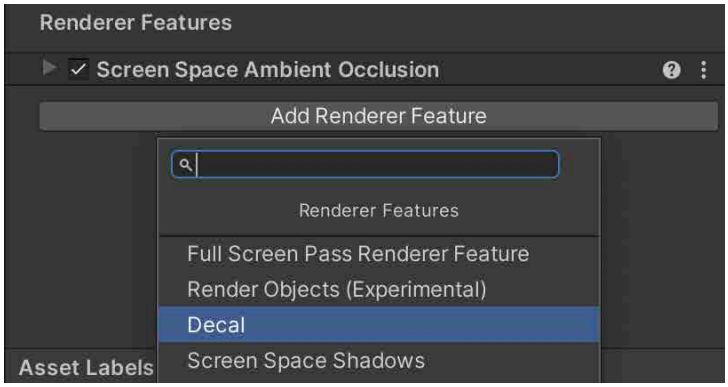
SSAO はポストプロセスの技法のひとつで、このガイドの後半で詳細に説明します。

3 つのバージョンの幽霊部屋：上は SSAO なし、中央は SSAO 適用、下は SSAO を適用してレンダリング

デカール

Decal Projector

Decal Projector コンポーネントは、メッシュにディテールを加える素晴らしい方法です。銃弾の穴、足跡、看板、亀裂などの要素に使用できます。Decal Projector は投影フレームワークを使用しているため、平らでない表面や曲面に適応します。URP で Decal Projector を使用するには、レンダラーデータアセットを見つけて、**Decal Renderer Feature** を追加する必要があります。

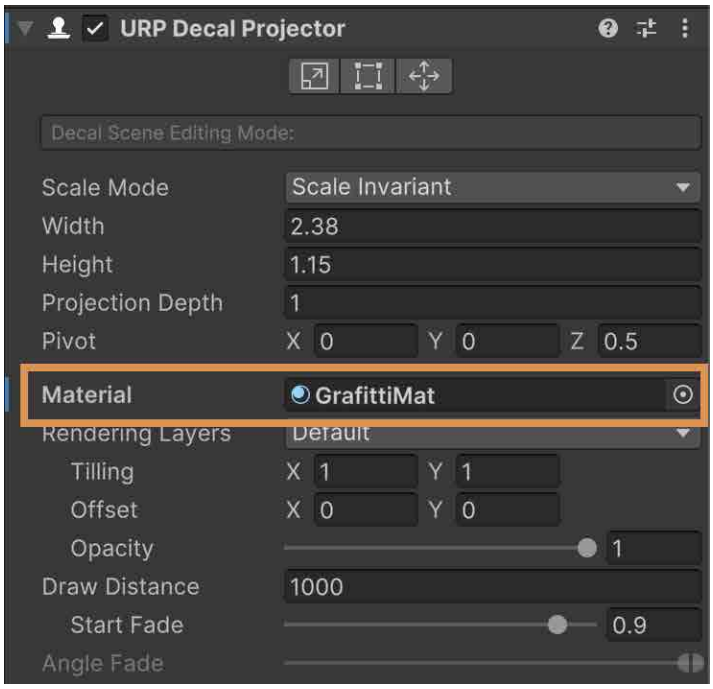


Decal Renderer Feature の追加

ほとんどの場合、デフォルト設定のままで問題ありません。

これでシーンでデカールを使用する準備ができました。Hierarchy ビューで右クリックし、**「Rendering」> 「URP Decal Projector」を選択してデカールを作成します。**デフォルトでは、プロジェクターはサーフェスに白い正方形を投影するマテリアルデカールを使用します。通常のツールを使ってプロジェクターを正しい位置と向きに配置します。Inspector で「**Width**」、「**Height**」、「**Projection Depth**」を調整します。

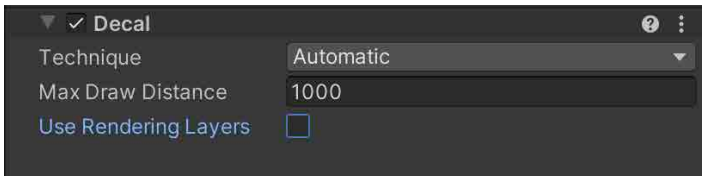
デカールをカスタマイズするには、**「Shader Graph」>「Decal」**シェーダーを使ってマテリアルを作成します。次に、URP Decal Projector に割り当てます。



Decal Projector コンポーネントの設定

Decal Projector の Inspector には、3 つの**編集モード**ボタン、Scale、Crop、Pivot/UV があります。詳細はこちらから確認できます。

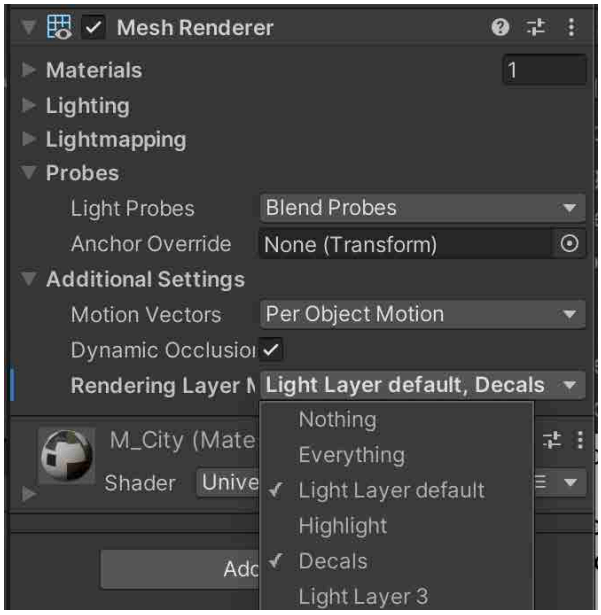
デフォルトでは、プロジェクターはその錐台内のすべてのサーフェスに影響を与えます。Decal Renderer Feature には「**Use Rendering Layers**」という設定が含まれています。これを有効にすると、特定のメッシュをターゲットにしやすくなります。



Decal Renderer Feature の設定

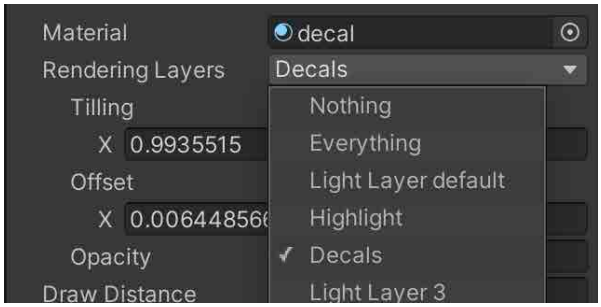
このレンダリングオプションの設定と使用方法については、レンダリングレイヤーのセクションを再度確認してください。デカールの設定手順は次の通りです。

1. **「Edit」>「Project Settings …」>「Tags and Layers」>「Rendering Layers」**を使用して、レンダリングレイヤーに名前を付けます。
2. プロジェクターを適用したいメッシュを選択します。Inspector で**「Mesh Renderer」>「Additional Settings」>「Rendering Layer Mask」**を見つけて、名前を付けたレンダリングレイヤーをマスクに追加します。



メッシュレンダラーのレンダリングレイヤーマスクにレンダリングレイヤーを追加

3. URP Decal Projector を選択し、Inspector で「Rendering Layers」プロパティに、名前を付けたレンダリングレイヤーを選択します。



下の画像は、デカールが適用されたシーンとされていないシーン、そしてレンダリングレイヤーを使って壁の投影を制限したシーンを示しています。

左から右：デカールが適用されていない、すべてのオブジェクトがデカールの影響を受けている、レンダリングレイヤーを使用して壁にのみデカールが適用されている

シェーダー

このセクションは、既存のカスタムシェーダーを URP 向けに変換したい、または Shader Graph を使用せずにコードでカスタムシェーダー作成したいユーザーのためのものです。基本的なシェーダーと高度なシェーダーの両方をビルトインレンダーパイプラインから URP に移植するために必要な情報を提供しています。含まれる表には、利用可能な HLSL シェーダー関数やマクロなどの有用なサンプルが載っています。各項目に、他の多くの便利な関数を含んだ関連するインクルードファイルへのリンクが提供されています。

シェーダーのコーディング経験がある人にとっては、コンパクトで効率的なシェーダーを記述するために HLSL で提供されている機能を知ることができます。このセクションの情報を読み、URP へのシェーダーの移植がそれほど大変なものではないと感じてもらえれば幸いです。

もう 1 つの方法は、Shader Graph を使用してカスタムシェーダーのバージョンを作成することです。Shader Graph の紹介は、追加ツールのセクションに含まれています。

URP シェーダーとビルトインレンダーパイプライン シェーダーの比較

以下のコードスニペットから分かるように、URP シェーダーは ShaderLab 構造を使用しています。そのため、Property、SubShader、Tags、Pass などはすべて、シェーダーのコーディング経験者には馴染みがあるはずです。

```
SubShader {
  Tags {“RenderPipeline” = “UniversalPipeline” }
   Pass {
      HLSLPROGRAM
      ...
      ENDHLSL
   }
}
```

SubShader ブロックの基本構造

まず注目すべきは、URP シェーダーが SubShader タグで “RenderPipeline” = “UniversalPipeline” というキーと値のペアを使用している点です。

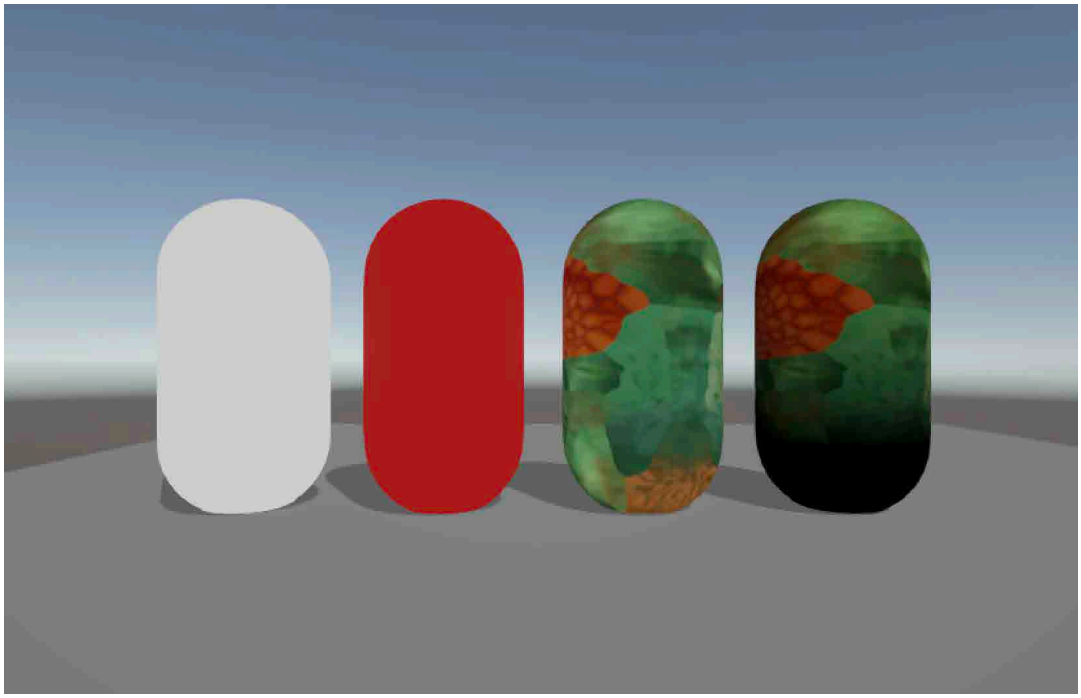
RenderPipeline という名前の SubShader タグは、この SubShader で使用するレンダーパイプラインを Unity に伝えます。UniversalPipeline という値は、Unity が URP でこの SubShader を使用するよう指定しています。

レンダーパスコードを見ると、HLSLPROGRAM / ENDHLSL マクロの間にシェーダーコードが記述されています。URP の場合、パス内のシェーダーコードは HLSL で書かれています。

Unity は GPU でサポートされている最初の SubShader ブロックを使用します。最初の SubShader ブロックに “RenderPipeline” = “UniversalPipeline” タグがない場合、URP では実行されません。代わりに、次の SubShader が存在する場合は、Unity はそれを実行しようとします。どの SubShader もサポートされていない場合、お馴染みのマゼンタのエラーシェーダーがレンダリングされます。

カスタムシェーダー

「Create」>「Shader」>「Unlit Shader」でカスタムシェーダの作成を開始すると、ビルトインレンダーパイプライン用のテンプレートが生成されます。これは SRP バッチャーと互換性のないコードを使用しています。なぜこうなるのでしょうか？ほとんどの開発者にとって、URP 用のカスタムシェーダーを作成するには Shader Graph を使用する方が便利です。しかし、長年 Unity で作成してきたシェーダーを持つ開発者も多く、それらを URP で最適に使用する方法を知りたいと思っているでしょう。このセクションでは、5 つの例を用いて、その方法を示します。エディターでシーンを表示したい場合は、前述のサンプルリポジトリの**「Shaders」>「Shaders」**から行うことができます。詳しくはドキュメントをご覧ください。



URP のシェーダー（左から順に）：ハードコードされた白の Unlit、プロパティを使って色を設定した Unlit 、テクスチャを使った Unlit、メインライト（フロア）を使ったシンプルな Lambert 照明。シャドウのアクセス方法や使用方法については、下部の「シャドウ」セクションを参照してください。

Unlit

もっともシンプルなシェーダーである、基本的な Unlit の例を見てみましょう。

- このシェーダを使用するすべての可視ピクセルは、同じ色に設定されます。
- 色がハードコードされているため、**Properties** は空です。
- 「**Tags**」の「**RenderType**」は「**Opaque**」（不透明）に設定されており、「**RenderPipeline**」は「**UniversalPipeline**」に設定されています。使用する #include は **Core.hlsl** で、便利な関数やマクロが多数含まれています。
    - ビルトインシェーダーに慣れている方であれば、これが UnityCG.cginc と似た役割を果たすことがわかるでしょう。`TransformObjectToHClip` 関数は、Core.hlsl インクルード内に含まれています。この関数は、オブジェクト空間を等質空間に変換することです。
- 頂点シェーダーの vert は、フラグメントシェーダーの frag に渡される **Varyings** 構造体の positionHCS 値を設定します。
- HLSL では、シェーダーステージ間で渡されるすべての値にセマンティクスを使用します。セマンティクスとは、シェーダーの入力または出力に付けられる文字列で、パラメーターの使用目的に関する情報を伝えるものです。
    - 例えば、**Attributes** の `positionOS` のセマンティクスは `POSITION` です。コンパイル時、頂点シェーダーの POSITION は、頂点のオブジェクト空間での位置を指します。フラグメント

シェーダーは単純に白を返し、上の画像の左側の白いカプセルが表示されます。

```
Shader “CustomURP/Unlit”
{
  Properties
  { }
  SubShader
  {
      Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
      Pass
      {
          HLSLPROGRAM

          #pragma vertex vert
          #pragma fragment frag
          #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
          struct Attributes
          {
              float4 positionOS   : POSITION;
          };
          struct Varyings
          {
              float4 positionHCS  : SV_POSITION;
          };
          Varyings vert(Attributes IN)
          {
              Varyings OUT;
              OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);
              return OUT;
          }
          half4 frag() : SV_Target
          {
              half4 customColor = half4(1, 1, 1, 1);
              return customColor;
          }

          ENDHLSL
      }
  }
}
```

Unlit カラー

— 2 つ目のシェーダーの例である Unlit カラーでは、色を設定するためのプロパティを含める必要があります。`[MainColor]` という属性を使いましょう。MainColor の使用は任意ですが、Unity にプロパティの使い方を知らせるのに役立ちます。プロパティの名前は `_BaseColor` で、Inspector 上の名前は **Base Color**、型は `Color` です。

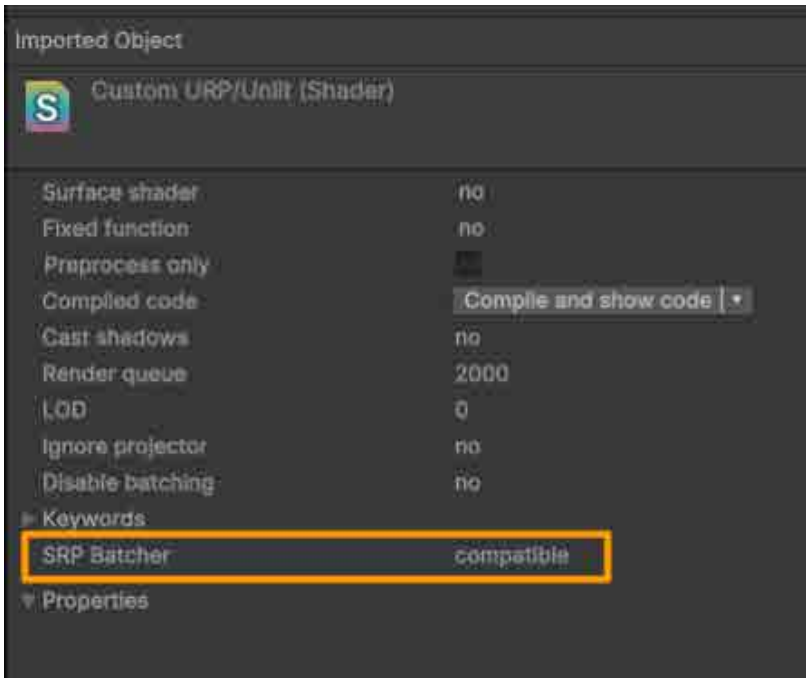
— URP のシェーダー変数は、SRP Batcher と互換性がある必要があります。この例では、`CBUFFER_START(UnityPerMaterial)` と `CBUFFER_END` の 2 つのマクロの間（`CBUFFER` ブロック呼ばれる）で宣言することで互換性を実現しています。

— こうすることで、フラグメントシェーダーが `_BaseColor` の値を返すようになります。

```
Shader “CustomURP/UnlitColor”
{
   Properties
   {
       [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
   }
   SubShader
   {
       Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
       Pass
       {
           HLSLPROGRAM
           #pragma vertex vert
           #pragma fragment frag
           #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
           struct Attributes
           {
               float4 positionOS   : POSITION;
           };
           struct Varyings
           {
               float4 positionHCS  : SV_POSITION;
           };
           CBUFFER_START(UnityPerMaterial)
               half4 _BaseColor;
           CBUFFER_END
           Varyings vert(Attributes IN)
           {
               Varyings OUT;
               OUT.positionHCS = TransformObjectToHClip (IN.positionOS.xyz);
               return OUT;
           }
```

```
            half4 frag() : SV_Target
            {
                return _BaseColor;
            }
            ENDHLSL
        }
    }
}
```

シェーダーが SRP Batcher と互換性があるか確認するには、シェーダーを選択して Inspector を確認します。



シェーダーの Inspector に、シェーダーが SRP Batcher と互換性があるかどうかが表示されます。

テクスチャ付き Unlit

画像の 3 つ目の例では、テクスチャが使われています。

- シェーダーには `_BaseMap`（Inspector 上では **Base Map**）と呼ばれるプロパティがあり、これは `[MainTexture]` 属性と 2D 型を使用します。
- テクスチャを使用するには、頂点からフラグメントシェーダーに渡す補間された UV 値が必要です。`Attributes` および `Varyings` の両方の構造体に、`TEXCOORD0` セマンティクスを持つ `float2, uv` を追加します。
- `CBUFFER` ブロックの直前に、2 つのマクロを追加します。`_BaseMap` プロパティを取る `TEXTURE2D` と、`SAMPLER(sampler_BaseMap)` です。

- タイリングとオフセットを機能させるには、2D プロパティの名前に _ST というサフィックスを付けた float4 の定義が必要です。これは、頂点シェーダーが使う TRANSFORM_TEX というマクロを使用する際は必ず必要です。このマクロは頂点の UV 値（この例では IN.uv）と、TEXTURE2D マクロを使って宣言された 2D プロパティを受け取ります。これにより、タイリングとオフセットがサポートされるようになります。
- フラグメントシェーダーは、SAMPLE_TEXTURE2D マクロを使用します。これは、Texture2D、サンプラー、UV 値の 3 つのパラメーターを受け取り、色の値を返します。この値が、frag メソッドの返り値になります。

```
Shader “CustomURP/UnlitTexture”
{
   Properties
   {
       [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
       [MainTexture] _BaseMap(“Base Map”, 2D) = “white” {}
   }
   SubShader
   {
       Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
       Pass
       {
           HLSLPROGRAM
           #pragma vertex vert
           #pragma fragment frag
           #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
           struct Attributes
           {
               float4 positionOS   : POSITION;
               float2 uv           : TEXCOORD0;
           };
           struct Varyings
           {
               float4 positionHCS  : SV_POSITION;
               float2 uv           : TEXCOORD0;
           };
           TEXTURE2D(_BaseMap);
           SAMPLER(sampler_BaseMap);
           CBUFFER_START(UnityPerMaterial)
               half4 _BaseColor;
               float4 _BaseMap_ST;
           CBUFFER_END
           Varyings vert(Attributes IN)
           {
```

```
                Varyings OUT;
                OUT.positionHCS = TransformObjectToHClip (IN.positionOS.xyz);
                OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
                return OUT;
            }
            half4 frag(Varyings IN) : SV_Target
            {
                half4 color = SAMPLE_TEXTURE2D(_BaseMap, sampler_Base Map, IN.uv);
                return color;
            }
            ENDHLSL
        }
    }
}
```

Lit Simple

- URP シェーダーでライティングを使用するには、**Core.hlsl** と同じフォルダーにある **Lighting.hlsl** をインクルードします。
- このシェーダーの例では、最大強度のディレクショナルライトであるメインライトを使用しています。また、Lambert モデルを使用し、頂点レベルで計算してフラグメントシェーダーで補間した値を使用しています。そのため、`Varyings` 構造体にセマンティクスが `TEXCOORD2` の `half3 lightAmount` を追加記述する必要があります。
- 頂点シェーダーでは、`GetVertexNormalInputs` 関数を用いて `VertexNormalInputs` を取得しています。この関数は、オブジェクト空間の法線をワールド空間に変換します。
    - 中心に配置されたオブジェクトの場合、オブジェクト空間の位置を法線の代わりに使用できます。`VertexNormalInputs` は、`float4` 値の `normalWS` を含む構造体です。
- メインライトの詳細は、`GetMainLight` 関数を使用して取得します。これは、ライトの色や方向などを含む `Light` 構造体を返します。
- 最後に、vert 関数で現在の頂点でのライティング情報を返すために `LightingLambert` 関数を使用します。`LightingLambert` 関数は、ライトの色、ライトの方向、そしてワールド空間の法線の 3 つのパラメーターを受け取ります。この時点で、関数を呼び出すのに必要なデータはすべて揃っています。この関数の返り値は `half3` です。
- フラグメントシェーダーは、`SAMPLE_TEXTURE2D` マクロを使用します。これは、Texture2D、サンプラー、UV 値の 3 つのパラメーターを受け取ります。これを、`w` 値を 1 に設定して `half4` に拡張した `lightAmount` で乗算します。

```
Shader “CustomURP/LitSimple”
{
   Properties
   {
       [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
       [MainTexture] _BaseMap(“Base Map”, 2D) = “white” {}
   }
   SubShader
   {
       Tags { “RenderType” = “Opaque” “RenderPipeline” = “UniversalPipeline” }
       Pass
       {
           HLSLPROGRAM
           #pragma vertex vert
           #pragma fragment frag
           #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl”
           #include “Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl”
           struct Attributes
           {
               float4 positionOS   : POSITION;
               float2 uv           : TEXCOORD0;
           };
           struct Varyings
           {
               float4 positionHCS  : SV_POSITION;
               float2 uv           : TEXCOORD0;
               half3 lightAmount   : TEXCOORD2;
           };
           TEXTURE2D(_BaseMap);
           SAMPLER(sampler_BaseMap);
           CBUFFER_START(UnityPerMaterial)
               half4 _BaseColor;
               float4 _BaseMap_ST;
           CBUFFER_END
           Varyings vert(Attributes IN)
           {
               Varyings OUT;
               OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);
               OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
               VertexNormalInputs positions =
               GetVertexNormalInputs(IN.positionOS);
               Light light = GetMainLight();
```

```
                OUT.lightAmount = LightingLambert(light.color, light.direction, positions.normalWS.xyz);
                return OUT;
            }
            half4 frag(Varyings IN) : SV_Target
            {
                half4 color = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv) * half4(IN.lightAmount, 1);
                return color;
            }
            ENDHLSL
        }
    }
}
```

影

- 影の強さを制御するには、_ShadowStrength（Inspector では **Shadow Strength**）プロパティを追加します。これは、初期値が 0.5 の Float 値です。この値を CBUFFER ブロックに追加します。
    - 影を使用するには、pragma, multi_compile に、次の記述を追加する必要があります：_MAIN_LIGHT_SHADOWS
    - MAIN_LIGHT_SHADOWS_CASCADE
    - _MAIN_LIGHT_SHADOWS_SCREEN
- Varyings 構造体には、TEXCOORD3 セマンティクスを使用する float4 型の shadowCoords があります。
- vert 関数では、GetVertexPositionInputs 関数を使用してオブジェクト空間をワールド空間に変換します。これで、GetShadowCoord 関数を使用して、頂点の位置をシャドウマップ上の位置に変換することができます。これを Varyings 構造体の変数 shadowCoords に保存します。
- frag 関数では、補間された shadowCoord を MainLightRealtimeShadow 関数に渡して、shadowAmount を取得します。strength は、1 から _ShadowStrength を引いた値に設定されています。color の値は、_BaseColor を strength と shadowAmount の最大値で調整したものです。

```
Shader “CustomURP/SimpleShadows”
{
   Properties
   {
       [MainColor] _BaseColor(“Base Color”, Color) = (1, 1, 1, 1)
       _ShadowStrength(“Shadow Strength”, Float) = 0.5
   }
   SubShader
   {
       Tags { “RenderType” = “AlphaTest” “RenderPipeline” =  “UniversalPipeline” }
       Pass
       {
           HLSLPROGRAM
           #pragma vertex vert
           #pragma fragment frag
           #pragma multi_compile _ _MAIN_LIGHT_SHADOWS _MAIN_LIGHT_
           SHADOWS_CASCADE _MAIN_LIGHT_SHADOWS_SCREEN
           #include “Packages/com.unity.render-pipelines.universal/
           ShaderLibrary/Lighting.hlsl”
           struct Attributes
           {
               float4 positionOS  : POSITION;
           };
           struct Varyings
           {
               float4 positionCS  : SV_POSITION;
               float4 shadowCoords : TEXCOORD3;
           };
           CBUFFER_START(UnityPerMaterial)
               half4 _BaseColor;
               float _ShadowStrength;
           CBUFFER_END
           Varyings vert(Attributes IN)
           {
               Varyings OUT;
               OUT.positionCS = TransformObjectToHClip(IN.positionOS.xyz);
               VertexPositionInputs positions =
               GetVertexPositionInputs(IN.positionOS.xyz);
               // 頂点の位置をシャドウマップ上の位置に変換
               float4 shadowCoordinates = GetShadowCoord(positions);
               OUT.shadowCoords = shadowCoordinates;
               return OUT;
           }
           half4 frag(Varyings IN) : SV_Target
           {
```

```
                // 影の座標に基づいてシャドウマップから値を取得
                half shadowAmount = MainLightRealtimeShadow(IN.shadowCoords);
                half strength = 1.0 - _ShadowStrength;
                half4 color = _BaseColor * max(strength, shadowAmount);

                return color;
            }

            ENDHLSL
        }
    }
}
```

カスタム シェーダーの場合、URP へのアップグレード時に、準備が必要になります。以下の関数表をお役立てください。

- カスタム URP シェーダー の Transform ポジション
- カスタム URP シェーダーでカメラを使用する
- カスタム URP シェーダーでライティングを使用する
- カスタム URP シェーダーで影を使用する

この動画チュートリアルでは、Unity プロジェクトで、カスタム Unlit ビルトインシェーダーを URP に変換する方法を学ぶことができます。

チュートリアルを見る

注：URP シェーダーを書こうと考えているユーザーにとって、Cyanilux によるこのチュートリアルは素晴らしいリソースです。

パイプラインコールバック

SRP の素晴らしい特徴の 1 つは、C# スクリプトを使用してレンダリングプロセスのあらゆる段階でコードを追加できることです。例えば、以下の段階でスクリプトを挿入することができます。

- シャドウのレンダリング中
- プリパスのレンダリング中
- G-buffer のレンダリング中
- ディファードライトのレンダリング中
- 不透明度のレンダリング中
- スカイボックスのレンダリング中
- 透明度のレンダリング中
- ポストプロセスのレンダリング中

ユニバーサルレンダラーデータアセットの Inspector にある **Add Renderer Feature** オプションから、レンダリングプロセスにスクリプトを挿入できます。すでに説明したように、URP を使用する場合、ユニバーサルレンダラーデータオブジェクトと URP アセットが存在します。URP アセットには、少なくとも 1 つのユニバーサルレンダラーデータオブジェクトが割り当てられたレンダラーリストがあります。これは、**「Project Settings」>「Graphics」>「Scriptable Render Pipeline Settings」**で割り当てられたアセットです。

異なるシーンに対して複数の設定アセットを試す場合、以下のスクリプトをメインカメラにアタッチしておくと便利です。Inspector で**パイプラインアセット**を設定します。これにより、新しいシーンがロードされたときにアセットが切り替わります。

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
[ExecuteAlways]
public class AutoLoadPipelineAsset : MonoBehaviour
{
       public UniversalRenderPipelineAsset pipelineAsset;
       // 最初のフレームが更新される前に Start が呼び出される
       void OnEnable()
       {
           if (pipelineAsset)
           {
                GraphicsSettings.defaultRenderPipeline = pipelineAsset;
             QualitySettings.renderPipeline = pipelineAsset;
           }
       }
}
```

シーンのロード時にユニバーサルレンダーパイプライン（URP）アセットを切り替えるスクリプト

次のセクションでは、アーティストと経験豊富なプログラマー向けの、2 つの異なるタイプの Renderer Feature について説明します。

Render Objects

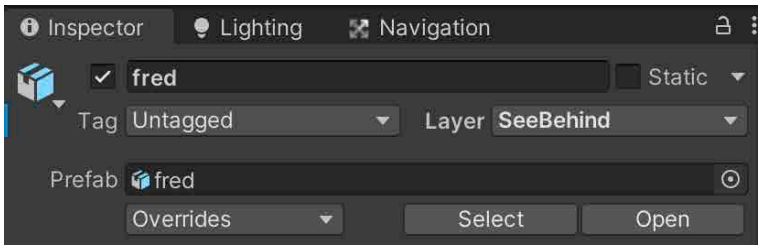
ゲームの一般的な問題として、プレイヤーキャラクターが環境オブジェクトの後ろに隠れて見えなくなることが挙げられます。キャラクターが常に視界に入るようにカメラを動かしたり、環境ができるだけオープンになるように調整することで対応できるかもしれません。しかし、常にこのオプションが利用できるわけではありません。この状況で使える有用なテクニックは、下の画像のように、キャラクターとカメラの間に環境モデルが現れた時、キャラクターのシルエットを表示することです。

環境モデルによってキャラクターが隠れている時にシルエットを表示する

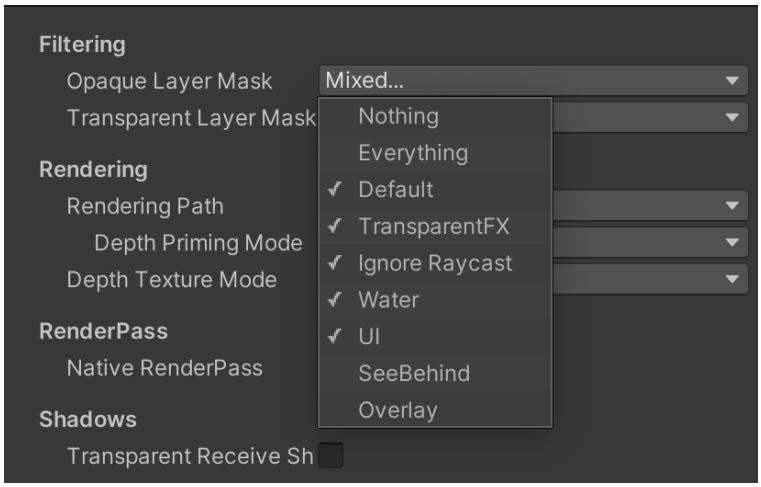
このシルエットの作成方法を説明します。

1. まず、キャラクターが隠れたときに使用するマテリアルが必要です。マテリアルを作成し、シェーダーを「**Universal Render Pipeline**」>「**Lit or Unlit**」に設定します（前の画像は Lit オプションを示しています）。「**Surface Inputs**」>「**Base Map**」の色を設定します。この例では、マテリアルを「Character」と呼びます。

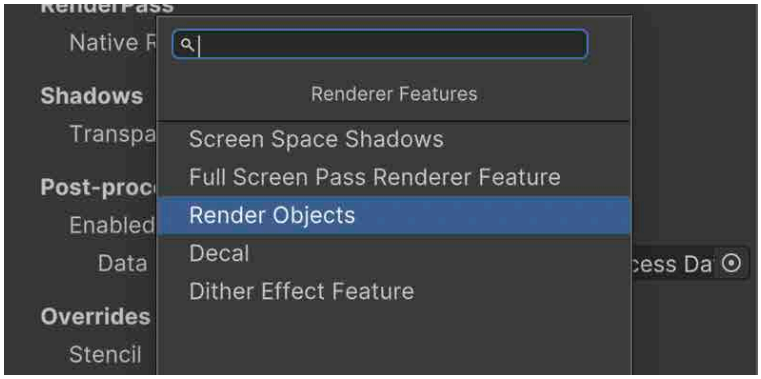
2. キャラクターを必要以上にレンダリングしないように、特別なレイヤーに配置しましょう。キャラクターを選択して、**SeeBehind** レイヤーをレイヤーリストに追加し、キャラクターに対して選択します。



3. URP アセットが使用する**レンダラーデータ**オブジェクトを選択します。「**Opaque Layer Mask**」に移動し、SeeBehind レイヤーを除外します。これにより、キャラクターが見えなくなります。

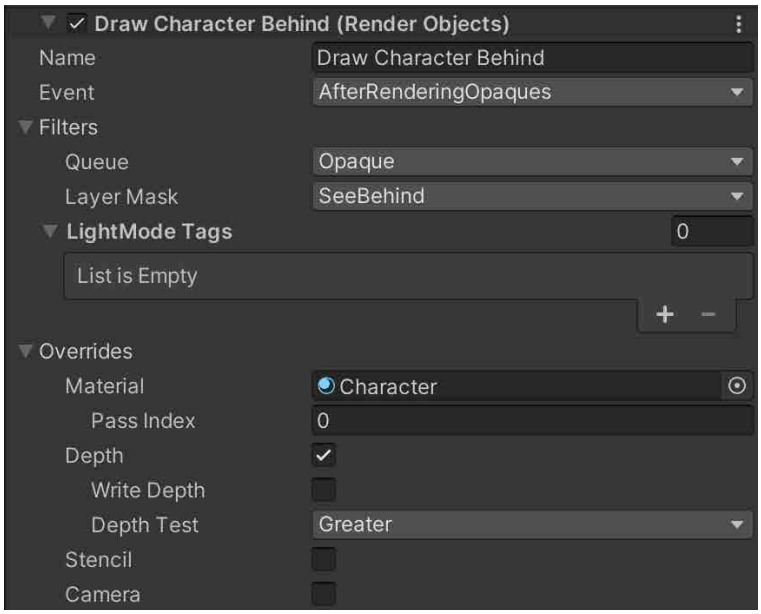


4. 「**Add Renderer Feature**」をクリックし、「**Render Objects**」を選択します。

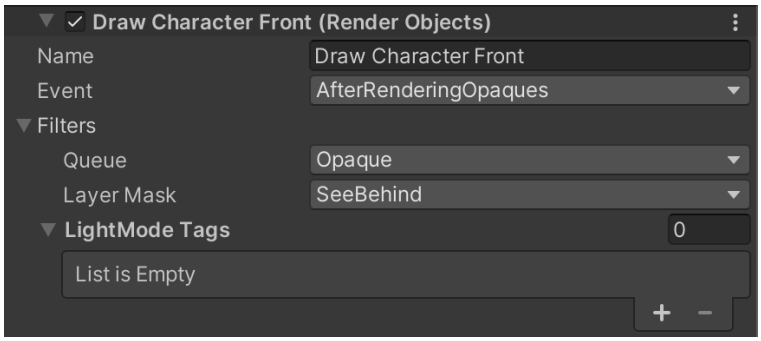


5. このレンダーオブジェクトの**パス**の設定を入力します。名前を付けて、レンダリングがトリガーされるタイミングを選択します。この例では、AfterRenderingOpaques と命名しています。

   Layer Mask をキャラクターに対して選択した **SeeBehind** レイヤーに設定します。**Overrides** を展開し、**Override Mode** を **Material** に設定します。ステップ 1 で作成したマテリアルを選択します。レンダリング時に、深度バッファに書き込むことなく深度を使用したい場合は、**Depth** を使用することが推奨されます。**Depth Test** を **Greater** に設定し、パスが深度バッファに格納されている距離よりも遠いピクセルのみをレンダリングするようにします。



6. この段階では、キャラクターが他のオブジェクトの後ろに位置するときのみ、そのキャラクターのシルエットが見えます。キャラクターが完全に視界内に入っている時は、全く表示されません。この問題を修正するため、もう 1 つ **Render Objects** 機能を追加します。今回は「Overrides」パネルを更新する必要はありません。このパスは、他のオブジェクトで隠れていないときにキャラクターを描画します。



シルエット手法は、レンダーパイプラインでの注入を容易にするために SRP ワークフローを活用するメリットの一例です。

レンダーグラフシステム

レンダー グラフ システムにより、カスタム SRP をメンテナンス可能なモジュール化された方法で作成できます。RenderGraph API を使用すれば、カスタム SRP のレンダーパスを高レベルで表現するレンダーグラフを作成できます。これは、レンダーパイプラインがレンダーパス間でリソースをどのように使用するかを明示するものです。グラフィカルなシステムではありません。

このようにレンダーパスを記述することには、2 つの利点があります。1 つは、レンダーパイプラインの設定が簡素化されること、もう 1 つは、レンダーグラフシステムがレンダーパイプラインの一部を効率的に管理できるようになり、実行時のパフォーマンスが向上する可能性があることです。

レンダーグラフシステムを使用するには、通常のカスタム SRP とは異なる形でコードを書く必要があります。

主な原則

RenderGraph API を使ってレンダーパスを書く前に、以下の基本的な原則を理解しておきましょう。

- リソースを直接扱わず、レンダーグラフシステム固有のハンドルを使用します。すべての **RenderGraph** API は、これらのハンドルを使用してリソースを操作します。レンダーグラフが管理するリソースの種類は、**RTHandles**、**ComputeBuffers**、そして **RendererLists** です。
- 実際のリソースリファレンスには、レンダーパスの実行コード内でのみアクセスできます。
- このフレームワークでは、レンダーパスを明示的に宣言する必要があります。各レンダーパスで、どのリソースから読み込み、どのリソースに書き込むかを明示する必要があります。
- データは各レンダーグラフの実行間で保持されません。つまり、レンダーグラフの実行内で作成したリソースは、次の実行には引き継がれません。
- フレーム間など、リソースの永続性が必要な場合は、通常のリソースと同じようにレンダーグラフの外で作成し、インポートすることで対応可能です。依存関係の追跡という点では、レンダーグラフの他のリソースと同じように機能しますが、ライフタイムの管理はグラフ側では行われません。
- レンダーグラフは、主にテクスチャリソースに RTHandles を使用します。これは、シェーダーコードの書き方や設定方法にさまざまな影響を与えます。

リソース管理

レンダーグラフシステムは、フレーム全体の高レベル表現をもとに、各リソースの生存期間を計算します。つまり、RenderGraph API を介してリソースを作成しても、レンダーグラフシステムがその時点でリソースを作成するわけではありません。代わりに、API はリソースを表すハンドルを返し、すべての RenderGraph API でこのハンドルを使用します。レンダーグラフがリソースを作成するのは、リソースを書き込む必要がある最初のパスの直前のみです。ここでの「作成」は、必ずしもレンダーグラフシステムがリソースを割り当てることを意味するわけではありません。レンダーパス中にリソースを使用できるように、そのリソースに必要なメモリを提供することを意味しています。同様に、リソースを読み込む必要がある最後のパスが終了した後に、そのリソースメモリを解放します。これによって、レンダーグラフシステムは、パスで宣言した内容に基づいて、最も効率的な方法でメモリを再利用することができます。レンダーグラフシステムが特定のリソースを必要とするパスを実行しない場合、システムはリソースにメモリを割り当てません。

レンダーグラフ実行の概要

レンダーグラフの実行は 3 段階のプロセスで、レンダーグラフシステムが毎フレーム最初から実行します。これは、グラフがフレームごとにユーザーのアクションなどによって動的に変化する可能性があるためです。

- **設定：**最初のステップは、すべてのレンダーパスを設定することです。ここで、実行するすべてのレンダーパスと、各レンダーパスが使用するリソースの宣言を行います。
- **コンパイル：**2 つ目のステップは、グラフをコンパイルすることです。このステップで、レンダーグラフシステムは、他のレンダーパスが出力を使用しないレンダーパスをカリングします。これにより、グラフをセットアップする際に固有のロジックを減らせるため、多少整理されていないセットアップでも問題なく対応できます。このステップでは、リソースの生存期間の計算も行われます。これにより、レンダーグラフシステムは、リソースを効率的に作成およびリリースし、非同期コンピュートパイプライン上でパスを実行する際に適切な同期ポイントを計算することができます。
- **実行：**最後に、グラフが実行されます。レンダーグラフシステムは、カリングされなかったすべてのレンダーパスを宣言順に実行します。各レンダーパスの実行前に、レンダーグラフシステムは必要なリソースを作成し、後続のレンダーパスで使われない場合はそのレンダーパスの実行後にリリースします。

具体的な例を見てみましょう。この例の最終的なコードはこちらから取得できます。

レンダーグラフシステムを使用して、レンダーパイプラインに注入される Renderer Feature の作成方法について学びましょう。このチュートリアルでは、最新の手法を用いて帯域幅を減らしながら、モバイルプラットフォームに適した高パフォーマンスなポストプロセスパスを作成します。

チュートリアルを見る

Renderer Feature

Renderer Feature は、URP の任意の段階で使用でき、最終的なレンダリング結果に影響を与えます。例として、マテリアルを使って画像の各ピクセルを処理し、色調効果を作り出すポストプロセス技術を実装してみましょう。

1. まず、「Project Assets」フォルダーから適切なフォルダーを探します。右クリックして、**「Create」 > 「Rendering」 > 「URP Renderer Feature」** を選択します。「**TintFeature**」と命名します。

2. デフォルトの「**TintFeature**」ファイルをダブルクリックします。これは、Renderer Feature のボイラープレートを含む C# スクリプトです。

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
using UnityEngine.Rendering.RenderGraphModule;

0 references
public class TintRendererFeature : ScriptableRendererFeature
{
    2 references
    class CustomRenderPass : ScriptableRenderPass
    {
        // This class stores the data needed by the RenderGraph pass.
        // It is passed as a parameter to the delegate function that executes the RenderGraph pass.
        3 references
        private class PassData
        {
        }

        // This static method is passed as the RenderFunc delegate to the RenderGraph render pass.
        // It is used to execute draw commands.
        1 reference
        static void ExecutePass(PassData data, RasterGraphContext context)
        {
        }

        // RecordRenderGraph is where the RenderGraph handle can be accessed, through which render passes can be added to th
        // FrameData is a context container through which URP resources can be accessed and managed.
        0 references
        public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
        {
            const string passName = "Custom Render Pass";

            // This adds a raster render pass to the graph, specifying the name and the data type that will be passed to the
            using (var builder = renderGraph.AddRasterRenderPass<PassData>(passName, out var passData))
            {
                // Use this scope to set the required inputs and outputs of the pass and to
                // setup the passData with the required properties needed at pass execution time.

                // Make use of frameData to access resources and camera data through the dedicated containers.
                // Eg:
                // UniversalCameraData cameraData = frameData.Get<UniversalCameraData>();
                UniversalResourceData resourceData = frameData.Get<UniversalResourceData>();
```

Renderer Feature のデフォルトコード

3. TintRendererFeature に、以下のプロパティを追加します。

   - **RenderPassEvent** は、ユーザーが Inspector で注入ポイントを設定できるようにします。
   - **Material** は、コピー時に使用します。
   - Requirements は **Color** に設定されています。
   - ProfilingSampler を初期化します。`_BlitTexture` と `_BlitScaleBias` へのアクセスするための 2 つの ID が取得されます。
   - 最後に、**MaterialPropertyBlock** を定義します。

```
public RenderPassEvent injectionPoint = RenderPassEvent.AfterRendering;
public Material passMaterial;
public ScriptableRenderPassInput requirements =
                              ScriptableRenderPassInput.Color;
private ProfilingSampler m_Sampler;
private static readonly int m_BlitScaleBiasID =
                              Shader.PropertyToID("_BlitScaleBias");
private static MaterialPropertyBlock s_SharedPropertyBlock = null;
```

4. **CustomRenderPass** を「**TintPass**」という名前に変更し、**TintPass** クラスに以下のプロパティを追加します。Material には、レンダリングされた画像の現時点の状態に適用するシェーダーが含まれています。PassData は、パスを宣言するときに使用するデータを定義します。ここで、レンダリングコードからアクセスできるデータを設定します。

```
private Material m_Material;
private string m_PassName;
private ProfilingSampler m_Sampler;
private class PassData
{
    internal Material material;
    internal TextureHandle source;
}
```

4.TintPass にコンストラクターを追加してマテリアルを初期化し、レンダーパイプラインにおけるこのパスの位置を設定します。

```
public TintPass(Material mat, string name)
       {
           m_PassName = name;
           m_Material = mat;
           m_Sampler ??= new ProfilingSampler(GetType().Name + "_" + name);
        }
```

5. ExecutePass、OnCameraSetup、Execute、OnCameraCleanup の関数を削除します。

6. RecordRenderGraph 関数に、以下のコードを追加します。

   — resourceData インスタンスを取得します。これは、ポストプロセス後にアクティブなカラーテクスチャからテクスチャ記述子を取得するために使用されます。

   — 名前を変更し、新たなテクスチャをリクエストします。Render Graph は、必要に応じてそれを割り当てます。

   — 最初の Blit は、現在のカラーバッファを中間テクスチャにコピーし、次のパスの入力として使用できるようにします。AddRasterRenderPass を使ってパスが作成されます。最初のパスでは、passData インスタンスのソースを resourceData インスタンスの activeColorTexture に設定します。

   — UseTexture メソッドを使用してビルダー入力テクスチャを設定し、SetRenderAttachment を使用して出力を設定します。

   — 最後に、ビルダーが使うレンダー関数を設定します。以下のステップで作成される ExecuteCopyColorPass という関数を使用します。

   — 2 つ目の Blit はコピー時にマテリアルを使用します。最初の Blit と似ていますが、passData インスタンスにマテリアルを割り当て、SetRenderFunc に ExecuteMainPass 関数を指定しています。

```
public override void RecordRenderGraph(RenderGraph renderGraph, ContextContainer frameData)
{
  UniversalResourceData resourceData = frameData.Get<UniversalResourceData>();
  var colCopyDesc =
          renderGraph.GetTextureDesc(resourceData.afterPostProcessColor);
  colCopyDesc.name = "_TempColorCopy";
  TextureHandle copiedColorTexture = renderGraph.CreateTexture(colCopyDesc);
  using (var builder = renderGraph.AddRasterRenderPass<PassData>(m_PassName +
                "_CopyPass", out var passData, m_Sampler))
  {
      passData.source = resourceData.activeColorTexture;
      builder.UseTexture(resourceData.activeColorTexture, AccessFlags.Read);
      builder.SetRenderAttachment(copiedColorTexture, 0, AccessFlags.Write);
      builder.SetRenderFunc(
         (PassData data, RasterGraphContext rgContext) =>
         {
             ExecuteCopyColorPass(rgContext.cmd, data.source);
         });
  }
  using (var builder = renderGraph.AddRasterRenderPass<PassData>(m_PassName +
                "_FullScreenPass", out var passData, m_Sampler))
  {
      passData.source = resourceData.activeColorTexture;
```

```
        passData.material = m_Material;
        builder.UseTexture(copiedColorTexture, AccessFlags.Read);
        builder.SetRenderAttachment(resourceData.activeColorTexture, 0,
                  AccessFlags.Write);
        builder.SetRenderFunc(
           (PassData data, RasterGraphContext rgContext) =>
           {
               ExecuteMainPass(rgContext.cmd, data.material, data.source);
           });
    }
}
```

7. ExecuteCopyColorPass を提供する必要があります。ここでは単に、Blitter メソッドの BlitTexture を使用します。この関数にはさまざまなパラメーターが用意されています。この例では、こちらのバージョンを使用しています。

```
public static void BlitTexture(CommandBuffer cmd,
RTHandle source, Vector4 scaleBias, float mipLevel, bool
bilinear)
```

この関数は、RecordRenderGraph 関数の前に置く必要があります。

```
private static void ExecuteCopyColorPass(RasterCommandBuffer cmd, RTHandle sourceTexture)
        {
            Blitter.BlitTexture(cmd, sourceTexture, new Vector4(1, 1, 0, 0), 0.0f, false);
        }
```

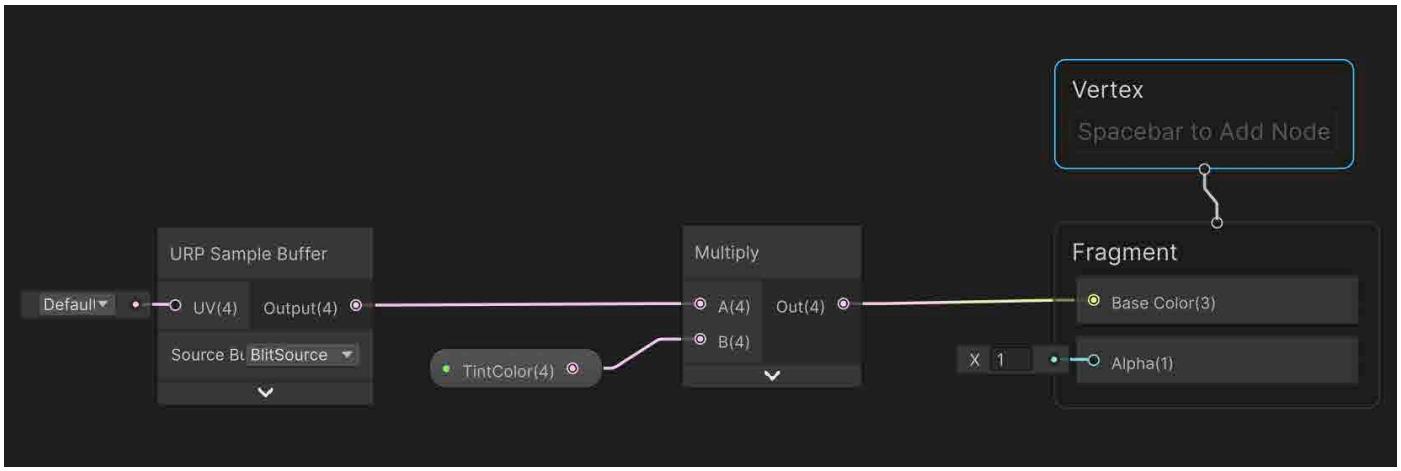
8. ここで、ExecuteMainPass 関数を定義します。コア Blit.hlsl に依存するシェーダーを持つユーザーマテリアルを動作させるには、uniform の _BlitScaleBias を設定する必要があります。

```
private static void ExecuteMainPass(RasterCommandBuffer cmd, Material material,
     RTHandle copiedColor)
{
    s_SharedPropertyBlock.Clear();
     s_SharedPropertyBlock.SetVector(m_BlitScaleBiasID, new Vector4(1, 1, 0, 0));
     cmd.DrawProcedural(Matrix4x4.identity, material, 0, MeshTopology.Triangles,
                        3, 1, s_SharedPropertyBlock);
}
```

9. `CustomRenderPass` の名前を TintPass に変更し、m_ScriptablePass を `m_pass` に変更します。
10. `Create` メソッドの既存のコードを削除します。カスタムコンストラクターを使用して新しい TintPass を作成します。renderPassEvent を定義し、入力を設定します。

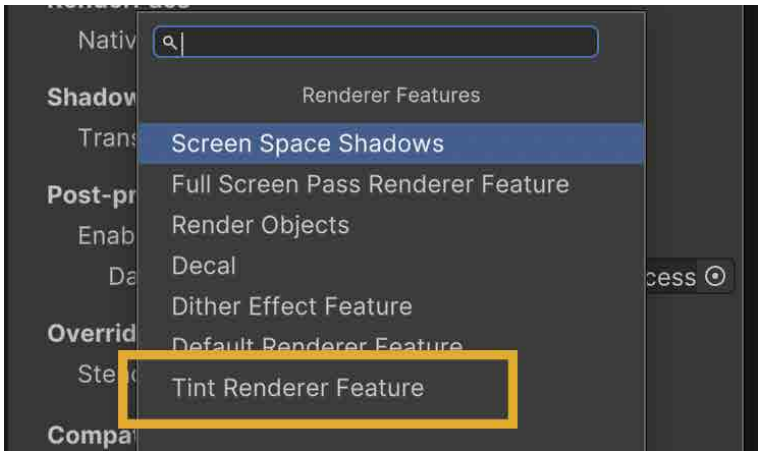
```
public override void Create()
    {
        m_pass = new TintPass(passMaterial, name);
        m_pass.renderPassEvent = injectionPoint;
        m_pass.ConfigureInput(requirements);
    }
```

次のステップで、Shader Graph シェーダーで動作するように設計された Renderer Feature の例が完成します。これは **TintColor** プロパティ、BlitSource を使用する **URP Sample Buffer** ノード、既存の BlitSource を TintColor で変調する **Multiply** ノードからなる単純な例です。以下のステップを見てみましょう。



Tint Shader Graph

1. 実際の動作を確認するには、**レンダラーデータ**オブジェクトを選択し、「**Add Renderer Feature**」をクリックします。TintFeature がリストに表示されます。



2. Tint Shader Graph を使用してマテリアルを作成します。
3. Tint マテリアルを Renderer Feature に割り当て、マテリアルの色を設定します。

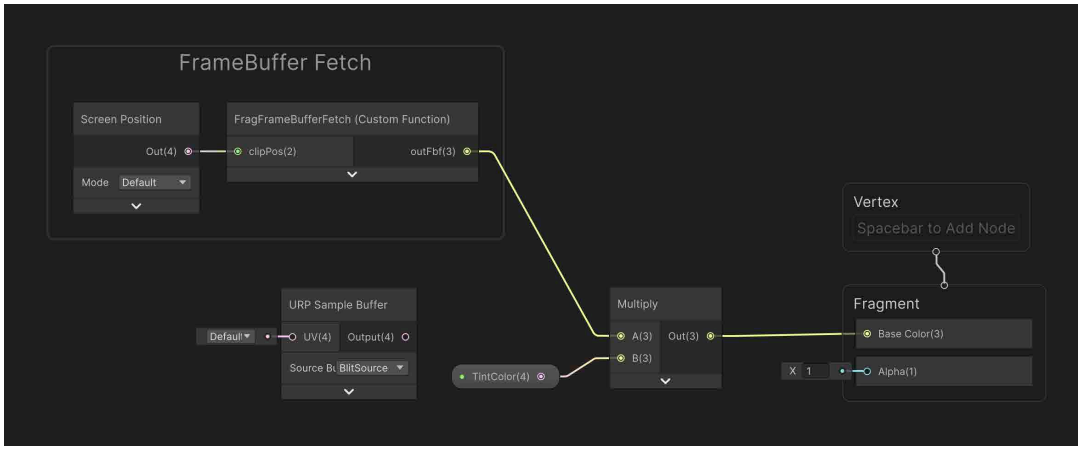
TintFeature の効果：加工されていない部分（左）と着色された部分（右）

Unity 6 の Render Graph Viewer ウィンドウは、**「Window」>「Analysis」>「Render Graph Viewer」**から開くことができます。このビューアーを使えば、パイプライン内で何が起きているかを確認できます。

Render Graph Viewer ウィンドウ

Draw Objects Pass、**TintRendererFeature_CopyPass**、**TintRendererFeature_FullScreen Pass** がすべて別のパスであることに注目してください。これらは、1 つのパスにまとめたほうが良いかもしれません。しかし、ここではテクスチャの使用方法との兼ね合いで、個別のパスが必要なのです。パスの名前をクリックし、パスリストを展開すると、パスの詳細を確認できます。「Draw Objects Pass」をクリックすると、「Failed to merge」というメッセージが表示されます。これは、次のパスがこのパスで出力されたデータを通常のテクスチャとして読み込むために発生します。TintRendererFeature_CopyPass でも同じ問題が発生しています。これらを修正しましょう。

新しいマテリアルが必要になります。このプロジェクトの Resources フォルダーには、**FrameBufferFetch** というマテリアルが含まれています。これは、2 つのパスを持つ同名のシェーダー（FrameBufferFetch）を使用します。注目すべきなのは、アクティブなフレームバッファをサンプリングする 2 番目のパスです。このアプローチを使用するには、Tint Shader Graph を調整する必要があります。URP Sample Buffer ノードの代わりに、カスタム関数ノードを使用します。これは、**FrameBufferFetch.hlsl** ファイルの HLSL を使用します。基本的には、`LOAD_FRAMEBUFFER_X_INPUT` マクロを使用するだけです。



FrameBufferFetch カスタムノードを使用した Tint Shader Graph

TintFeature に戻りましょう。TintPass で、新しいプライベート静的プロパティを追加する必要があります。

```
private static Material s_FrameBufferFetchMaterial;
```

これは、カスタムコンストラクター内で Load メソッドを使用して割り当てられます。

```
s_FrameBufferFetchMaterial ??= UnityEngine.Resources.
Load(“FrameBufferFetch”) as Material;
```

ExecuteCopyColorPass では Blit を削除し、DrawProcedural を代わりに使用する必要があります。単位行列を再び使用し、2 つ目のパスでは s_FrameBufferFetchMaterial マテリアルを使用します。最初のパスはインデックス 0 で、2 番目のパスはインデックス 1 です。トポロジーは再び三角形を使用するため、indexCount を 3、instanceCount を 1 に設定します。

```
cmd.DrawProcedural(Matrix4x4.identity,  s_FrameBufferFetchMaterial,  1,
MeshTopology.Triangles, 3, 1, null);
```

次に、RecordRenderGraph メソッドを使用します。最初のパスでは、UseTexture を SetInputAttachment に置き換えます。このメソッドは FrameBufferFetch を使って前のパスにアクセスします。

```
builder.SetInputAttachment(resourceData.activeColorTexture, 0 )
```

2 つ目のパスでも同じことを行います。

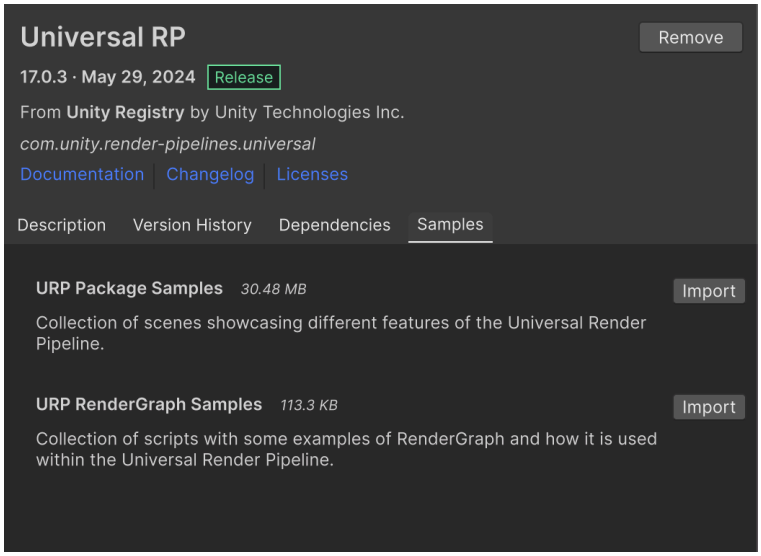
```
builder.SetInputAttachment( copiedColorTexture, 0 )
```

Render Graph Viewer を確認し、必要に応じて更新してください。Draw Objects パスと 2 つの TintRendererFeature パス、合計 3 つのパスが 1 つのパスにまとめられ、パフォーマンスが向上していることがわかります。ビューアーウィンドウの **F** は、入力が FrameBufferFetch 経由でアクセスされることを意味しています。

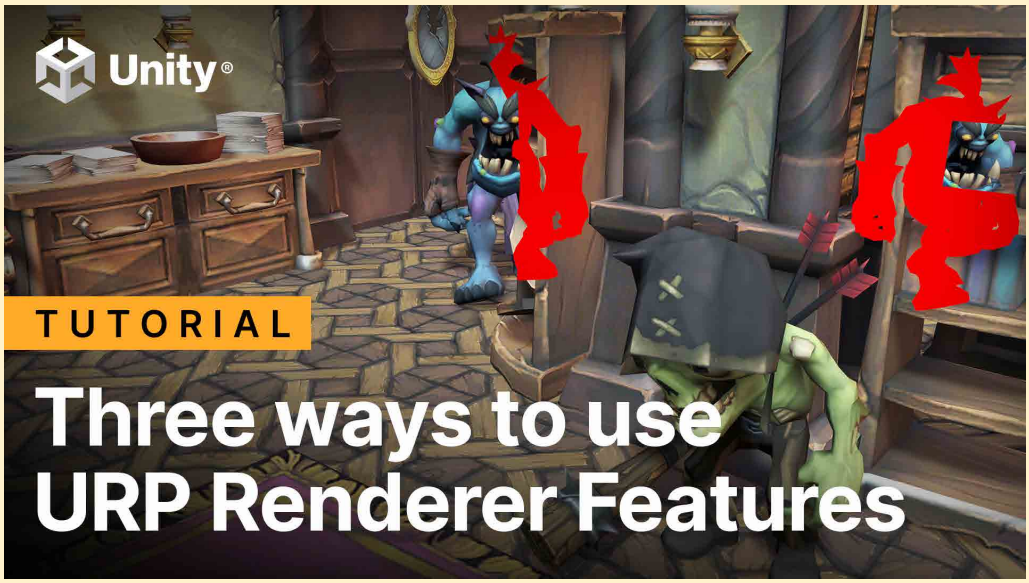
3 つのパスがマージされた

FramebufferFetch のサポートは、Vulkan、Metal、DirectX 12 をターゲットとするモバイルプラットフォームで利用できます。他のプラットフォームでは、エンジンはフォールバックとしてテクスチャサンプリングを使用します。FrameBufferFetch を使用することで、帯域幅の使用量を減らし、帯域幅が制限されている場合にパフォーマンスを向上させることが可能になります。また、一般的に、バッテリーの使用量も抑えられます。

レンダーグラフシステムの他の使用例を確認したい方は、Package Manager からパッケージサンプルをダウンロードできます。「Universal RP」を検索し、「Samples」タブをクリックしてください。そこから、URP RenderGraph サンプルをインポートすることが可能です。



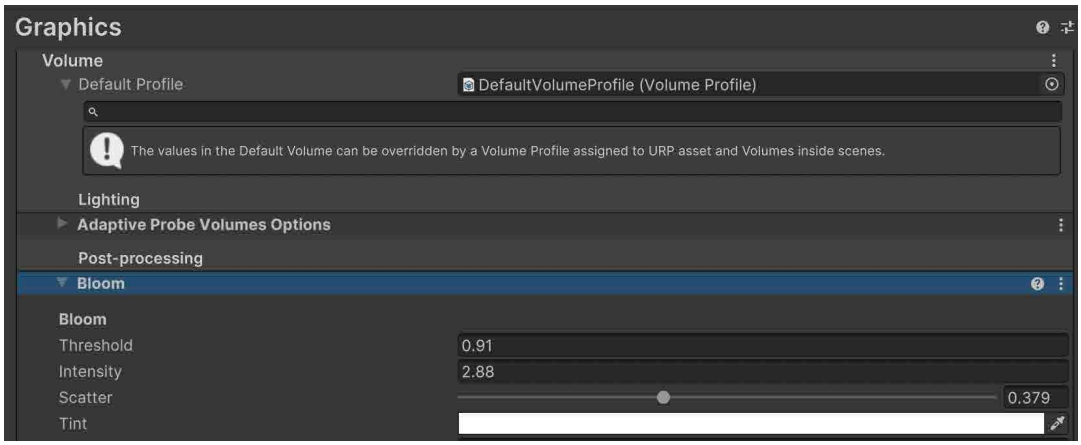
Universal RP パッケージと Package Manager を介した URP RenderGraph サンプルのインポート



この動画チュートリアルでは、Renderer Feature を使った 3 つの実践的な演習をご紹介します。カスタムポストプロセスエフェクトの作成方法、ステンシルエフェクトの作成方法、そしてキャラクターを環境でオクルードする方法です。

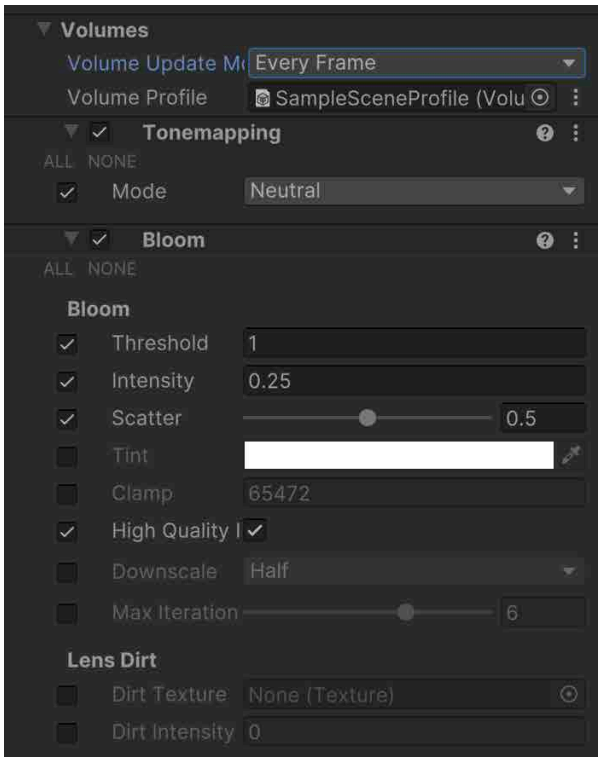
Post-processing

URP は、ポストプロセス効果を追加する際、ボリュームフレームワークを使用します。Unity 6 には、**デフォルト ボリューム**が含まれています。これは、**「Project Settings 」> 「Graphics」> 「Volume」**からアクセス可能です。ここで適用された設定はプロジェクト全体に影響しますが、シーンに追加されるボリュームによってオーバーライドできます。



デフォルトボリュームオプション

Unity 6 のもう 1 つの新機能は、アクティブな URP アセット用のボリュームで、これもシーンに追加されたボリュームでオーバーライドできます。



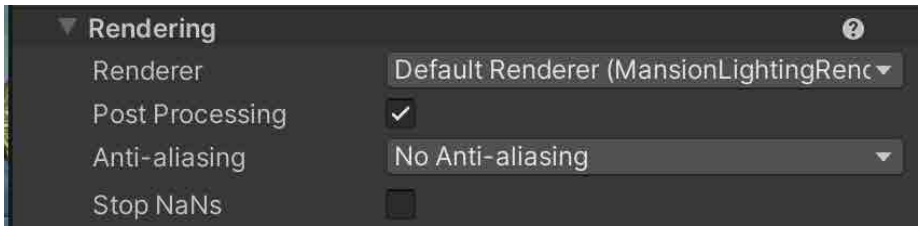
「URP Asset」>「Volumes」

シーンにボリュームを追加する際、どのポストプロセスエフェクトを適用するか選択できます。ボリュームは、グローバルとローカルのどちらにも設定できます。グローバルの場合、ボリュームは シーン内のすべての場所でカメラに影響を与えます。Mode がローカルに設定されている場合、ボリュームはコライダーの境界内にあるカメラに影響します。

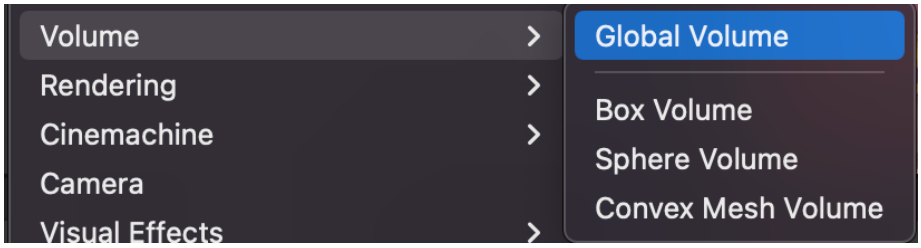
ポストプロセスエフェクトの適用：左上の画像には効果は適用されておらず、右上の画像には Bloom、左下の画像には Vignette が適用され、右下の画像には Color Adjustment が追加されています。

URP ポストプロセスフレームワークの使用

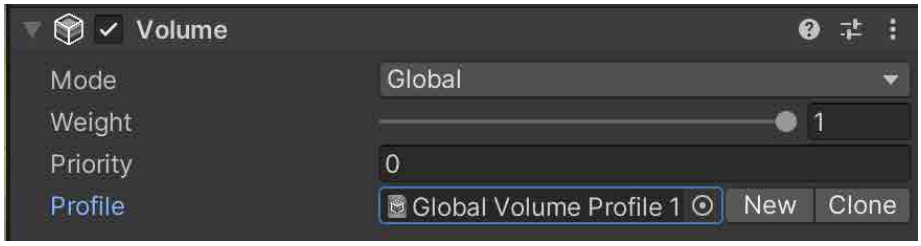
1. 最初のステップは、メインカメラのポストプロセスを有効化することです。「**Hierarchy**」ウィンドウで「**Main Camera**」を選択し、「**Inspector**」に移動した後、「**Rendering**」パネルを展開します。**Post Processing** オプションにチェックを入れます。



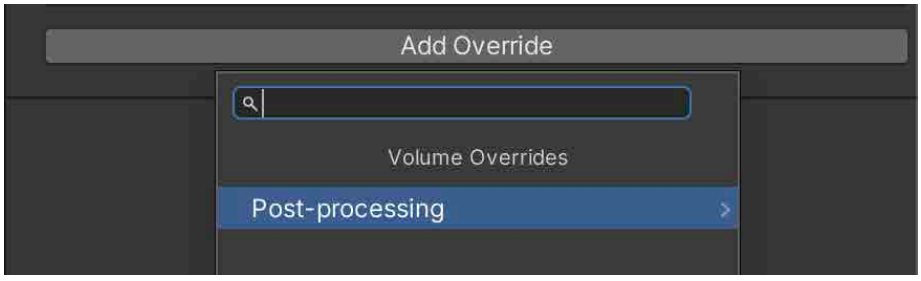
2. **Hierarchy** ウィンドウを右クリックし、**「Create」>「Volume」>「Global Volume」**を選択して、グローバルボリュームを作成します。

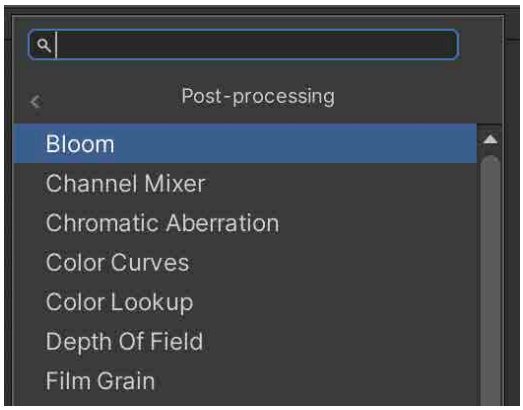


3. Hierarchy ウィンドウのグローバルボリュームが選択された状態で、Inspector 内の「**Volume**」パネルから、「**New**」をクリックして新しい**プロファイル**を作成します。



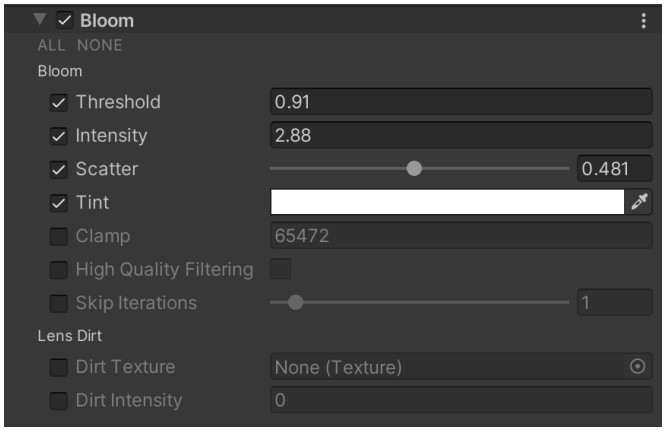
4. ポストプロセスエフェクトを追加し始めます。下の方にある利用可能な効果をリスト化した表を確認してください。「**Add Override**」をクリックし、「**Post-processing**」を選択します。この例では、Bloom エフェクトが選択されています。



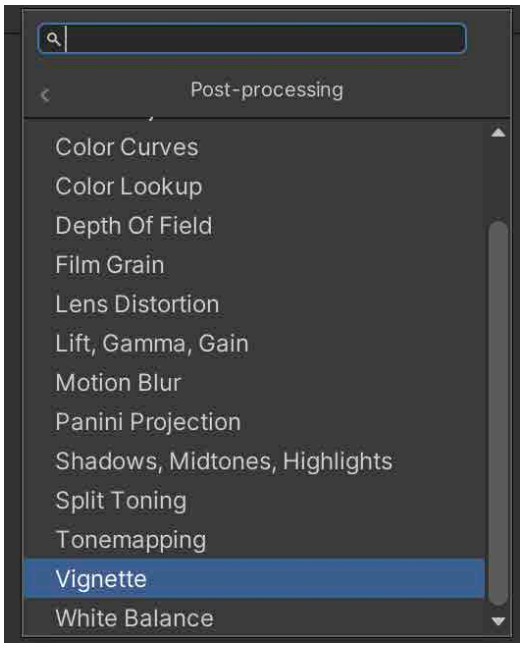


Bloom エフェクトを選択

5. それぞれの効果に専用の設定パネルがあります。この画像は、Bloom 用の設定を示しています。



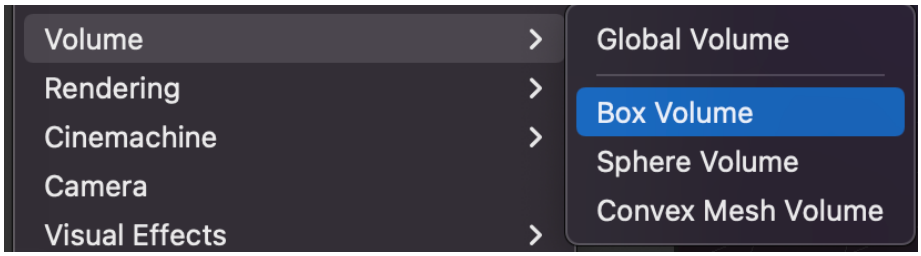
6. 複数の効果（この例では Vignette など）を簡単に追加し、各「Settings」パネルから設定を行うことができます。



ローカルボリュームコンポーネントの追加

ボリュームフレームワークを使用すると、カメラが移動するにつれて異なるポストプロセスプロファイルがトリガーされるようにシーンを設定できます。これは、ローカルボリュームコンポーネントを追加することで実現できます。この設定手順を見ていきましょう。

1. **Hierarchy** ウィンドウ内で右クリックして、**「Create」>「Volume」>「Box Volume」**を選択します。または、形状が目的に適している場合は、「**Sphere Volume**」を、ボリューム領域を定義するコライダーコンポーネントの形状をより厳密に制御したい場合は、「**Convex Mesh Volume**」を選択します。

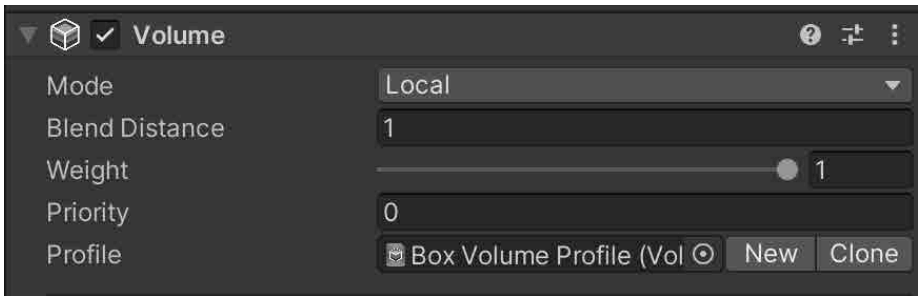


2. **Inspector** の「**Volume**」パネルから、このボリュームデータを保存する新しい**プロファイル**を作成します。このパネルでは、以下を設定することもできます。

   a. **Blend Distance：**これは、URP がブレンドを開始するボリュームのコライダーからの最大距離と、このプロファイルがフェードインするコライダー内の範囲です。コライダーの端では、ポストプロセスエフェクトはフェードアウトし、コライダーの端からの Blend Distance は完全にフェードインします。

   b. **Weight：**Weight は、ポストプロセスエフェクトの最大強度を定義します。Weight が 1 に設定されると、効果の強度は最大になります。値が 0 の場合、効果は全く適用されず、0.5 の場合、強度は最大で 50% になります。

   c. **Priority：**URP では、複数のボリュームがシーンに等しく影響を与える場合、この値を使ってどのボリュームを使用するかを決定します。値が大きいほど、優先度が高くなります。グローバルとローカルをマージする場合は、グローバルをデフォルトの 0 に保ち、ローカルボリュームを 1 以上に設定します



ローカルボリュームの設定

3. 下の画像のように、**Box Collider** コンポーネントを使用して**ボリューム**の位置を設定し、その寸法を制御します。

アタッチされた Box Collider コンポーネントを使用して、ボックスボリュームの位置とサイズを設定

ポストプロセスはプロセッサーに大きな負担をかける可能性があるため、ローエンドのハードウェアやモバイルデバイスへの影響を慎重に検討してください。どうしてもプロジェクトで使用したい場合は、対象のハードウェアでテストを行ってください。フィルターの中には、他のものよりもプロセッサー負荷が低いものがあります。このドキュメントでは、モバイルフレンドリーな効果の概要を説明しています。

以下は URP で利用可能なポストプロセスエフェクトです。

効果（エフェクト）	説明
ブルーム	定義された明度を超えるピクセルの周囲に輝きを追加します。
チャンネル ミキサー	全体的な混合に対する各入力カラーチャンネルの影響度を変更します。
色 収差	画像の暗い部分と明るい部分を分離する境界に沿って色の漏れを作成します。

色 調整	最終的にレンダリングされる画像の全体的なトーン、明度、コントラストを調整できます。
色 カーブ	色相、彩度、明るさの特定の範囲を調整する高度な方法です。
カラールックアップ	ルックアップテクスチャを使用して、各ピクセルの色を新しい値にマッピングします。
被写界深度	カメラレンズの被写界深度のシミュレーションを行います。
フィルム グレイン	写真フィルムのランダムな光学テクスチャをシミュレートします。
レンズ ディストーション	レンダリングされた最終的な画像を歪めることで、現実のカメラレンズの形をシミュレートします。
リフト ガンマ ゲイン	異なるトラックボールを使用して、画像内の異なる範囲に影響を与えます。トラックボール下のスライダーを調整することで、その範囲の色の明度を調整します。
モーション ブラー	現実世界のカメラで、カメラの露出時間よりも速く移動する物体を撮影したときに生じる画像のぼやけをシミュレートします。
パニーニ 投影	有効視野（FOV）が非常に広いシーンでの透視図をレンダリングするのに役立ちます。
スクリーン スペース レンズ フレア	単一のライトだけでなく、シーン全体に適用されるレンズフレアを追加します。
シャドウ 中間調 ハイライト	レンダリングのシャドウ、中間調、ハイライトを個別に制御します。
スプリット トーン処理	シーンのシャドウとハイライトに異なる色調を追加できます。
トーンマッピング	画像の HDR 値を新しい値の範囲に再マップします。
ビネット	画像の中央に比べて端を暗くします。
ホワイト バランス	不自然な色かぶりを取り除くことで、現実世界で白く見えるものが最終的な画像でも白くレンダリングされるようにします。

モーションブラー

上の画像はモーションブラーがオフ、下の画像はモーションブラーがオンになっています。

モーションブラーのポストプロセス効果は、現実世界のカメラで、カメラの露出時間よりも速く移動する物体を撮影したときに生じる画像のぼやけをシミュレートします。これは通常、急速に動くオブジェクトや長い露出時間によって引き起こされます。

モーションブラーの使用

URP を使用するすべてのポストプロセスエフェクトと同様に、モーションブラーはボリュームシステムを使用します。そのため、モーションブラーのプロパティを有効にして変更するには、シーン内のボリュームにモーションブラーのオーバーライドを追加する必要があります。

プロパティ	説明
Mode	モーションブラー技法を選択します。 オプション： — **Camera Only**：カメラのモーションだけでオブジェクトをぼかします。この手法はモーションベクトルを使用しないため、「Camera and Objects」よりもパフォーマンスが良くなります。 — **Camera and Objects**：カメラとゲームオブジェクトの両方のモーションを使用します。ゲームオブジェクトのモーションベクトルは、カメラのモーションベクトルを上書きします。
Quality	エフェクトの品質を設定します。低いプリセットはパフォーマンスを向上させますが、ビジュアルの品質は低下します。
Intensity	モーションブラーフィルターの強さを 0 から 1 の間の値で設定します。値を大きくするとブラー効果が強くなりますが、Clamp パラメーターによってはパフォーマンスが低下することがあります。
Clamp	カメラの回転で生じる速度の最大長を設定します。これにより、高速時のボケが制限され、過剰なパフォーマンスコストを避けることが可能になります。この値は、画面のフル解像度に対する割合として測定されます。値の範囲は、0 から 0.2 までです。デフォルトの値は 0.05 です。

パフォーマンスの問題のトラブルシューティング

モーションブラーのパフォーマンスへの影響を抑えるには、次の方法があります。

— **Quality を低く設定する**：品質を低くするとパフォーマンスは改善しますが、ビジュアルアーティファクトが増える可能性があります。

— **Mode プロパティを「Camera and Objects」から「Camera Only」に変更する：**「Camera Only」の手法の方が、「Camera and Objects」よりもパフォーマンスが良くなります。

— **Clamp を低く設定する:**こうすることで、Unity が考慮する最大速度を減らします。値を低くすると、パフォーマンスが改善します。

コードによるポストプロセスの制御

C# スクリプトを使用して、ポストプロセスのプロファイルを動的に調整することもできます。次のコード例は、Bloom エフェクトの強度を調整する方法を示しています。Vignette が適用されている場合は、コードでビネットの色を制御できます。例えば、プレイヤーキャラクターがダメージを受けた時、一時的に赤く色付けできます。

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
public class PPController : MonoBehaviour
{
       // 最初のフレームが更新される前に Start が呼び出される
       void Start()
       {
       Volume volume = GetComponent<Volume>();
       Bloom bloom;
       if (volume.profile.TryGet<Bloom>(out bloom))
       {
              bloom.intensity.value = 0;
       }
   }
}
```

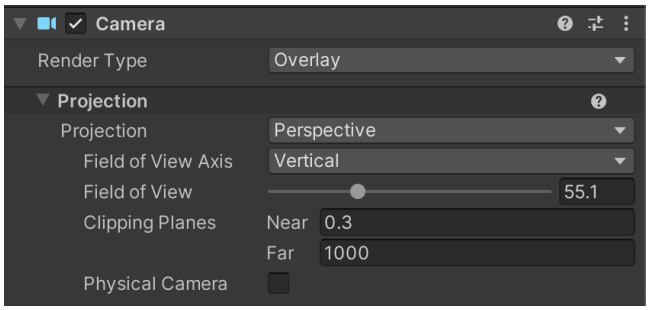
Camera Stacking

ゲームでよくある要件の 1 つは、異なるカメラから見たジオメトリを 1 つのレンダリングで組み合わせる機能です。以下の画像では、前景の棚がゲーム内のインベントリとして機能しています。収集したアイテムは棚に追加され、プレイヤーはそれらを重要なポイントで選択できます。有効視野（FOV）が異なるだけでなく、ライティングやポストプロセスも異なることに注目してください。これは URP の Camera Stacking 機能を使って設定されています。

Camera Stacking を使用した例

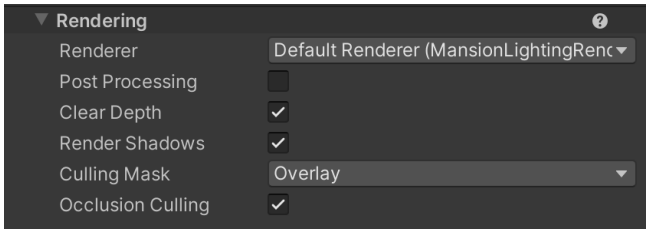
この機能の設定方法を見てみましょう。

1. **Hierarchy** ビューを右クリックして「**Camera**」を選択し、カメラを作成します。オーディオリスナーコンポーネントを削除します。
2. **「Inspector」>「Camera Settings」**パネルを使用して、このカメラを **Render Type Overlay** に設定します。



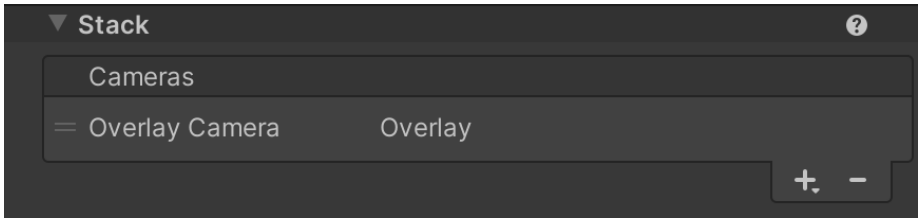
3. カメラとカメラがレンダリングするゲームオブジェクトに対して、新しい**レイヤー**を作成します。

4. Inspector を使用して、カメラの**「Rendering」>「Culling Mask」**を更新します。



5. シーンの適当な場所にカメラを移動し、**ゲームオブジェクト**を **Layer Overlay** に配置して追加します。

6. **メインカメラ**の**「Rendering」>「Culling Mask」**を更新して、オーバーレイをレンダリングしないようにします。



7. 「**Stack**」パネルで「**+**」ボタンを使用して**オーバーレイカメラ**を追加します。

コードでスタックを制御

ポストプロセスと同様、コードからスタックを制御し、ランタイム時に動的にカメラを追加または削除することができます。以下のコード例をご覧ください。

```
using UnityEngine;
using UnityEngine.Rendering.Universal;
public class StackController : MonoBehaviour
{
      public Camera overlayCamera;
      // 最初のフレームが更新される前に Start が呼び出される
      void Start()
      {
      Camera camera = GetComponent<Camera>();
      var cameraData = camera.GetUniversalAdditionalCamera
      Data();
      cameraData.cameraStack.Remove(overlayCamera);
      }
}
```

ポストプロセスと Camera Stacking は、どちらも URP を使用して簡単に設定でき、ゲームに豊かで雰囲気のある効果を作成するための強力なツールです。

SubmitRenderRequest API

時には、ゲームをユーザーの画面とは別の場所にレンダリングしたい場合があります。**SubmitRenderRequest** API は、そのような目的のために設計されています。可能なユースケースを見てみましょう。

画面キャプチャのコーディング

以下のスクリプトは、ユーザーが画面上の GUI を押すと、ゲームを画面外の**レンダーテクスチャ**にレンダリングします。スクリプトはメインカメラにアタッチする必要があります。**RenderTexture** は **Start** コールバックで作成します。1920 × 1080 ピクセルでビット深度は 24 です。ユーザーが「Render Request」ボタンを押すと、RenderRequest メソッドが呼び出されます。

RenderRequest メソッド内で、カメラコンポーネントが参照されています。RenderPipeline.StandardRequest インスタンスを作成し、現在のパイプラインが RenderRequest フレームワークをサポートしているかどうかをチェックします。サポートしている場合は、Start コールバックで初期化した RenderTexture をこのリクエストオブジェクトの宛先として設定し、RenderPipeline.SubmitRenderRequest を使用してレンダーを初期化します。このメソッドはカメラのインスタンスとリクエストオブジェクトを受け取ります。

この時点で、Texture2D は現在のシーンのレンダリングを含んでいます。これをファイルに保存するには、まず RenderTexture を Texture2D インスタンスに変換する必要があります。**ToTexture2D** メソッドは、その一例を示しています。Texture2D を取得したら、Texture2D インスタンスの **EncodeToPNG** メソッドを使用して、バイト配列を取得できます。次に、System.IO.File のメソッド、**WriteAllBytes** を使用してバイト配列をファイルに保存します。

スクリプトを直接使用する場合、画面キャプチャはゲームの **Assets** フォルダー内の **RenderOutput** という新しく作成されたフォルダーに保存されます。ファイル名は R_ から始まり、その後に 0 から 100,000 の間のランダムに生成された整数が続きます。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Rendering;
[RequireComponent(typeof(Camera))]
public class StandardRenderRequest : MonoBehaviour
{
   [SerializeField]
   RenderTexture texture2D;
   private void Start()
   {
       texture2D = new RenderTexture(1920, 1080, 24);
   }
   // ユーザーが GUI ボタンをクリックすると、
   // 特定のフレームをレンダリングするため、様々な出力テクスチャを含んだレンダーリクエストが送信される
   private void OnGUI()
   {
       GUILayout.BeginVertical();
       if (GUILayout.Button("Render Request"))
       {
           RenderRequest();
       }
       GUILayout.EndVertical();
   }
   void RenderRequest()
   {
       Camera cam = GetComponent<Camera>();
       RenderPipeline.StandardRequest request = new RenderPipeline.StandardRequest();
       if (RenderPipeline.SupportsRenderRequest(cam, request))
       {
           // 2D テクスチャ
           request.destination = texture2D;
           RenderPipeline.SubmitRenderRequest(cam, request);
           SaveTexture(ToTexture2D(texture2D));
       }
   }
   void SaveTexture(Texture2D texture)
   {
```

```
        byte[] bytes = texture.EncodeToPNG();
        var dirPath = Application.dataPath + "/RenderOutput";
        if (!System.IO.Directory.Exists(dirPath))
        {
            System.IO.Directory.CreateDirectory(dirPath);
        }
        System.IO.File.WriteAllBytes(dirPath + "/R_" + Random.Range(0,
        100000) + ".png", bytes);
        Debug.Log(bytes.Length / 1024 + "Kb was saved as: " + dirPath);
        #if UNITY_EDITOR
                UnityEditor.AssetDatabase.Refresh();
        #endif
    }
    Texture2D ToTexture2D(RenderTexture rTex)
    {
        Texture2D tex = new Texture2D(rTex.width, rTex.height,
        TextureFormat.RGB24, false);
        RenderTexture.active = rTex;
        tex.ReadPixels(new Rect(0, 0, rTex.width, rTex.height), 0, 0);
        tex.Apply();
        Destroy(tex);//prevents memory leak
        return tex;
    }
}
```

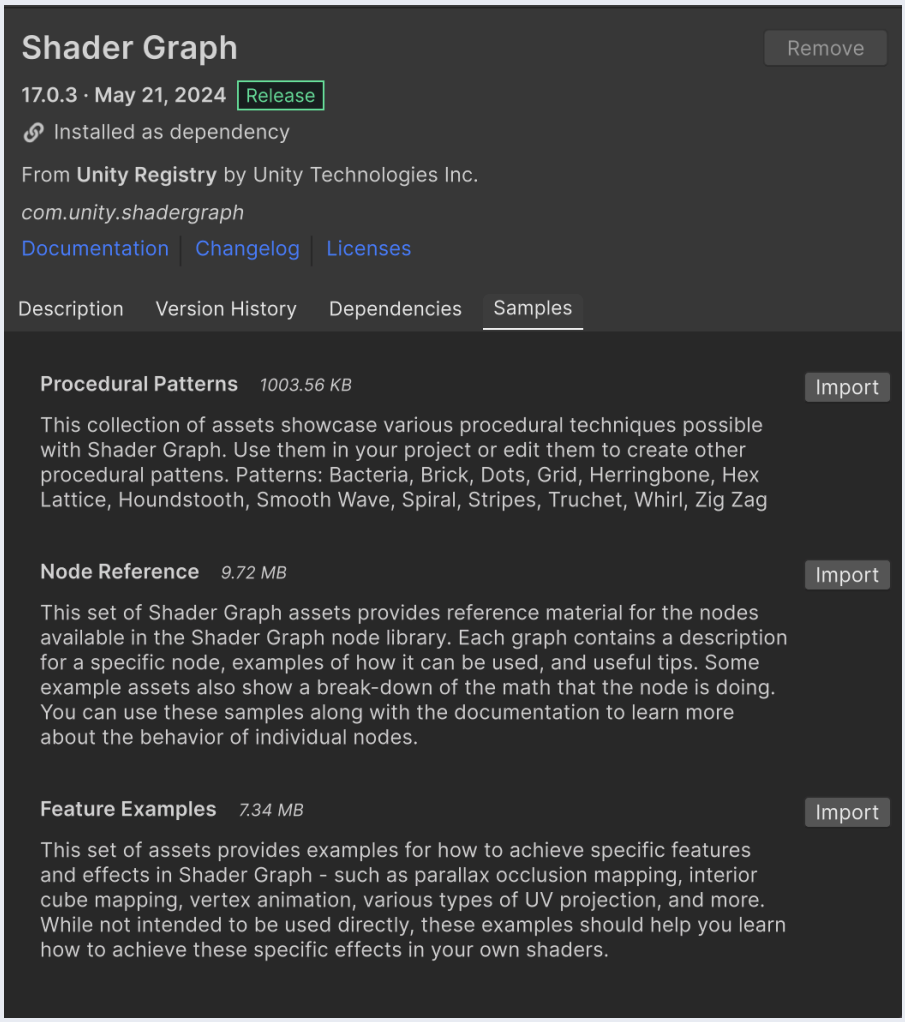
URP と互換性のある追加のツール

URP を使用するもう 1 つの利点は、複雑な作成作業をテクニカルアーティストが利用できるようにする Unity のオーサリングツールとの互換性です。この章では、Shader Graph と VFX Graph について紹介します。

Shader Graph

Shader Graph は、アーティストのワークフローにカスタムシェーダーを導入します。Shader Graph ツールは、URP テンプレートを使用してプロジェクトを開始するか、URP パッケージをインポートする際に含まれます。

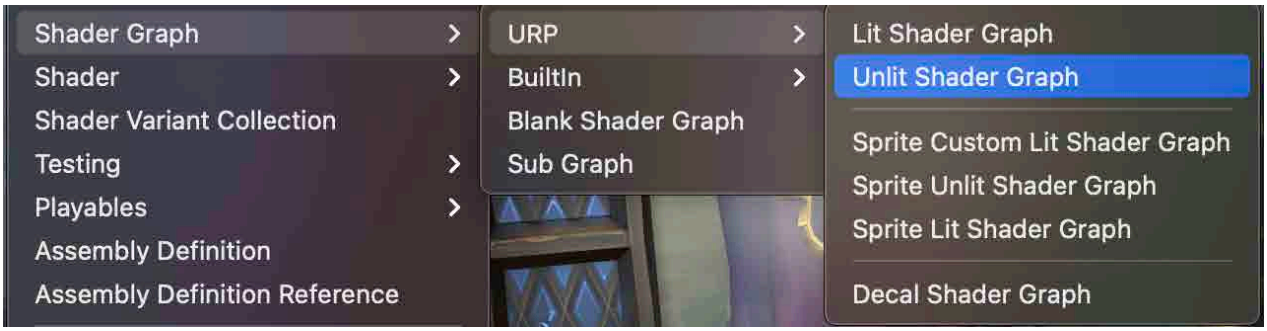
注：



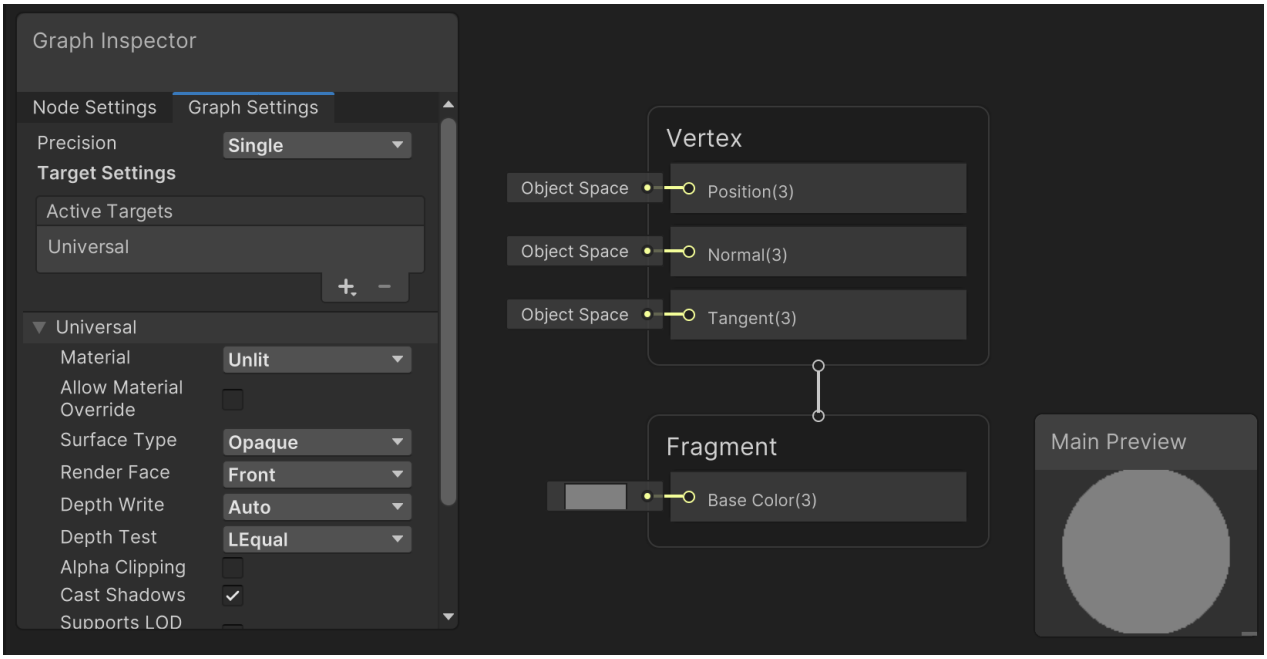
Shader Graph ノードのサンプルライブラリは、Package Manager ウィンドウの Shader Graph パッケージにあります。Unity 6 の新しい実用可能な Shader Graph シェーダーについては、こちらのブログ記事をご覧ください。

Shader Graph については別途詳しいガイドが必要ですが、ライティングの章で取り上げたライトハローシェーダーの作成を通して、重要な基本ステップを確認しましょう。

1. 「**Project**」ウィンドウで右クリックし、適切なフォルダーを見つけて、**「Create」>「Shader Graph」>「URP」>「Unlit Shader Graph」**を選択します。 今回は Unlit を選択してください。新しいアセットを FresnelAlpha と名付けます。

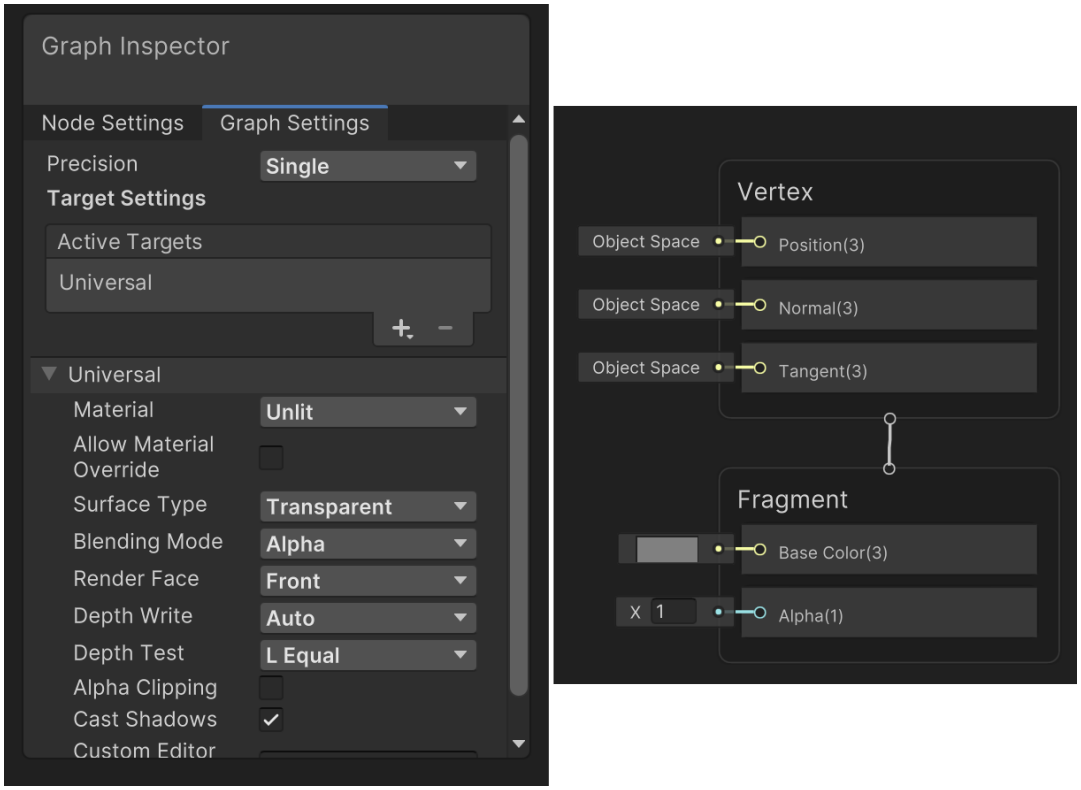


2. 新しい **Shader Graph アセット**をダブルクリックし、Shader Graph エディターを起動します。

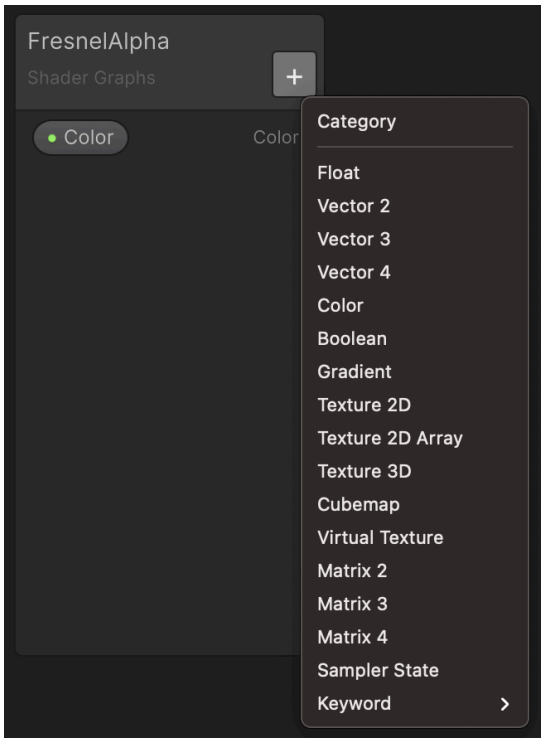


シェーダーを触った経験があれば、Vertex ノードと Fragment ノードについては知っているでしょう。デフォルトでは、このシェーダーは、マテリアルを使用するモデルが Vertex ノードを使用してカメラビューに正しく配置され、Fragment ノードを使用して各ピクセルがグレーカラーに設定されるようにします。

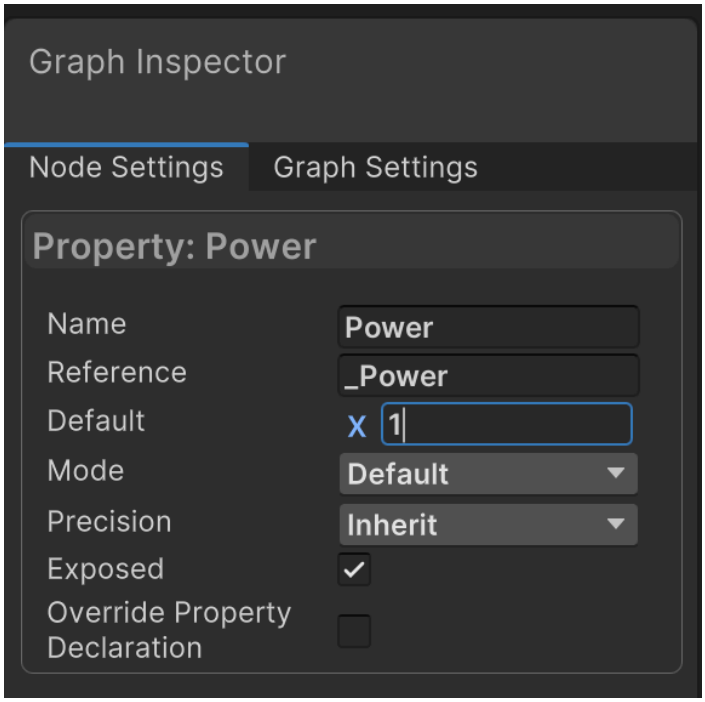
3. このシェーダーはオブジェクトのアルファ透明度を設定します。したがって、Transparent キューに適用する必要があります。**「Graph Inspector」>「Graph Settings」>「Surface Type」**を **Transparent** に変更します。Fragment ノードに Base Color だけでなく Alpha 入力が追加されていることがわかります。



4. シェーダーにプロパティを追加します。例えば、Color を Color として追加し、Power と Strength を Float 値として追加します。



5. **「Graph Inspector」>「Node Settings」>「Default」**を使用して、デフォルト値を設定します。Color を white に、Power を 4 に、Strength を 1 に設定します。

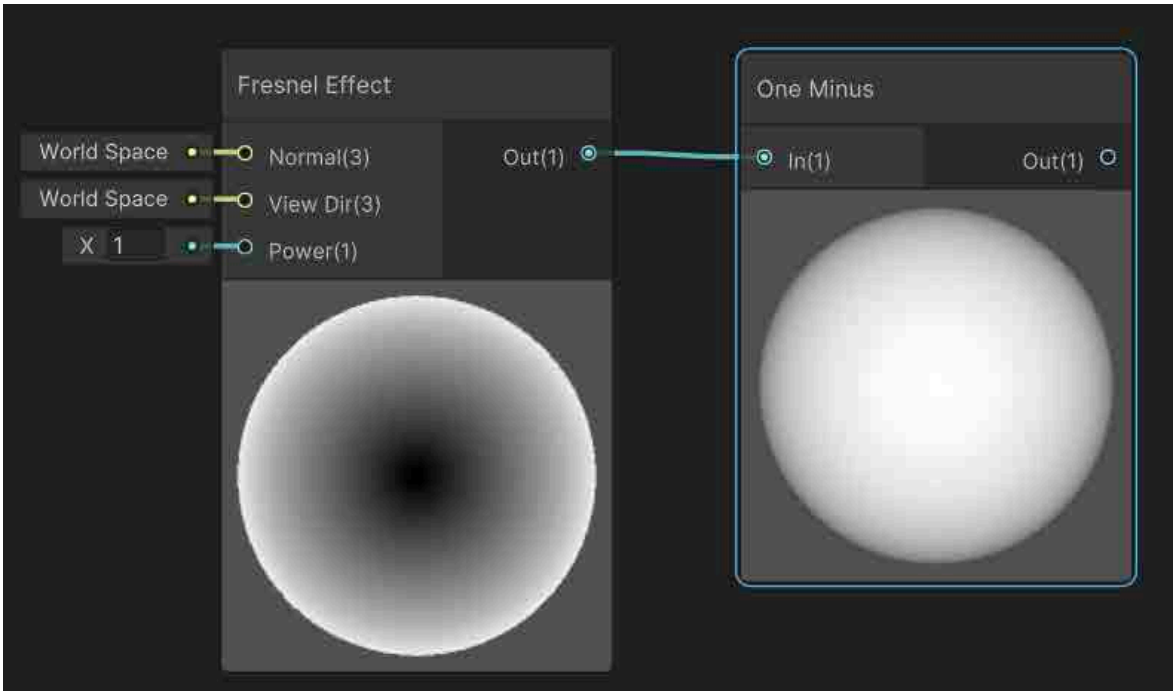


6. Shader Graph は、ノードを結合することで機能します。各ノードは、1 つ以上の入力と、1 つの出力を持ちます。ノードを追加するには、上部の「**Search**」パネルで右クリックし、「**Create Node**」を選択して「**Fre**」と入力します。そうすると、**Fresnel Effect** ノードが表示されます。

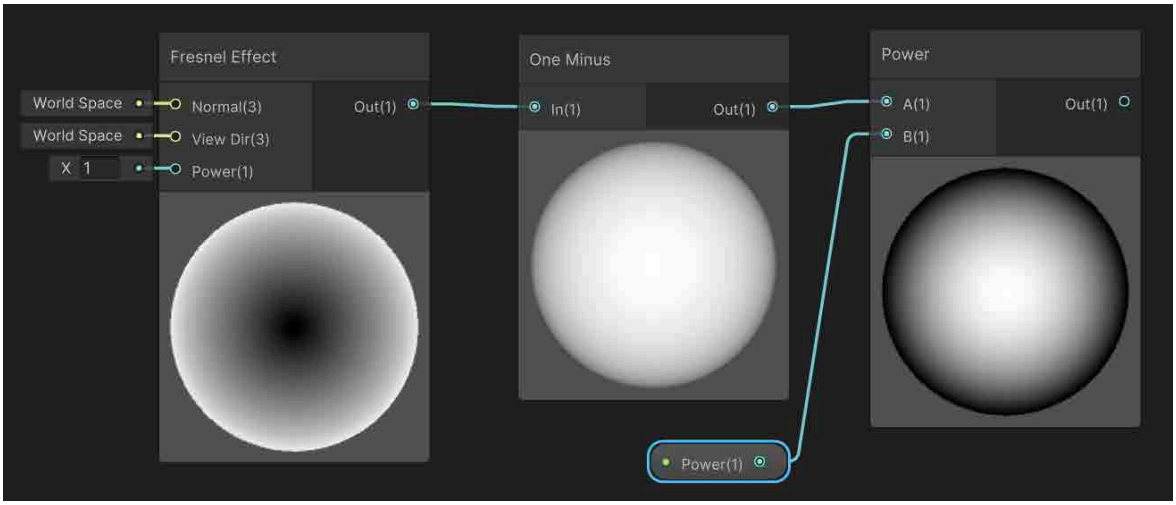
7. ノードには、効果のプレビューが表示されます。Fresnel エフェクトは、端に向かって明るくなることが分かります。値は、ビュー方向と法線方向の差であり、スフィアの場合、端でその差が最大となっています。

   アルファ値は、端で最小となります。One Minus ノードを使用して、結果を反転できます。これを行うには「**Create Node**」をクリックして、**One** と入力します。**One Minus** ノードを選択します。次に、Fresnel Effect ノードの Out(1) から One Minus ノードの In(1) へドラッグします。この 1 は、値の型が単一の Float であることを表しています。これが 3 だった場合、3 つの成分を持つベクトル

となります。ノードは以下のように結合します。

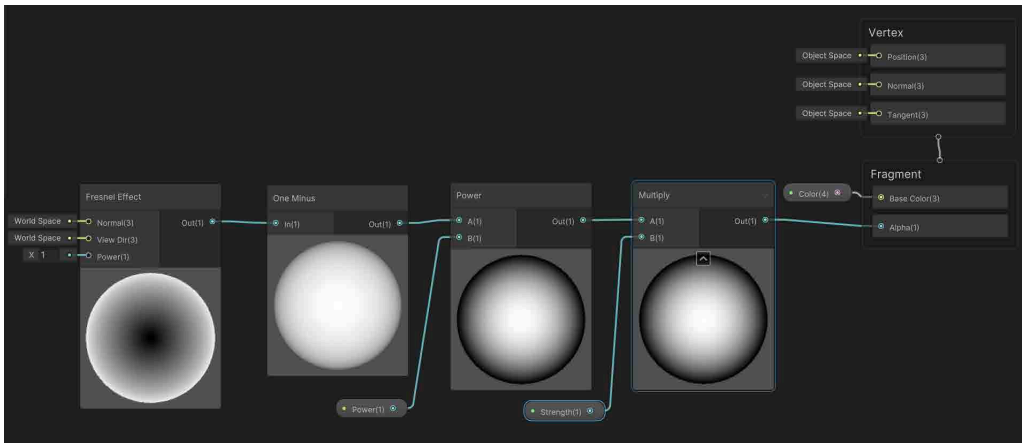


8. グラデーションのサイズと全体の透明度を制御する方法を見ていきましょう。グラデーションのサイズを変更するには、**Power** ノードを使用します。Power ノードを作成し、One Minus Out(1) を Power A(1) に接続します。Power プロパティをグラフにドラッグし、Power B(1) に接続します。この操作を完了すると、グラフは次のようになるはずです。

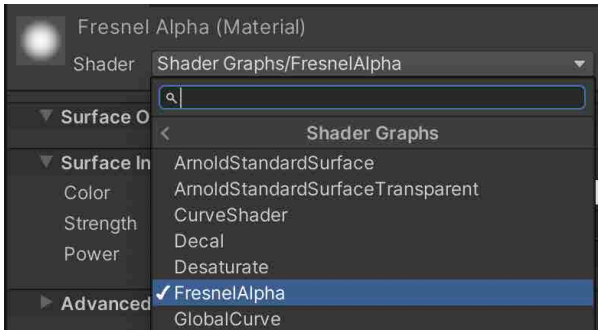


9. **Multiply** ノードを使用して、全体の透明度を制御します。Multiply ノードを作成した後、Power Out(1) を Multiply A(1) に接続します。Strength プロパティをグラフにドラッグし、Multiply B(1) に接続します。次に、Multiply Out(1) と Fragment Alpha(1) を結合し、Color(4) プロパティをグラフにドラッグして、Fragment Base Color(3) に接続します。

ここで、Color プロパティ は 4 つの成分を持つベクトルで構成されているのに対し、Base Color は 3 つの成分を持つベクトルで構成されていることが分かります。Shader Graph は、Color の最初の 3 つの成分を Base Color のベクターにマップします。



10. アセットを保存して、新しいマテリアルを作成します。シェーダーを、**Shader Graphs/FresnelAlpha** に格納されているこの新しいマテリアルに割り当てます。

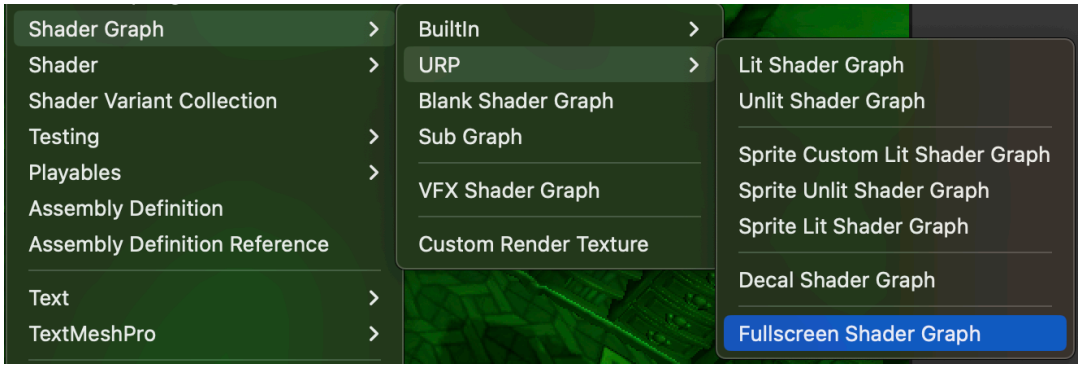


11. これで、マテリアルをオブジェクトに適用し、端の可視性を制御できるようになりました。

シェーダーがスフィア状のポイントライトに適用され、その周りにハローエフェクトを作り出している。

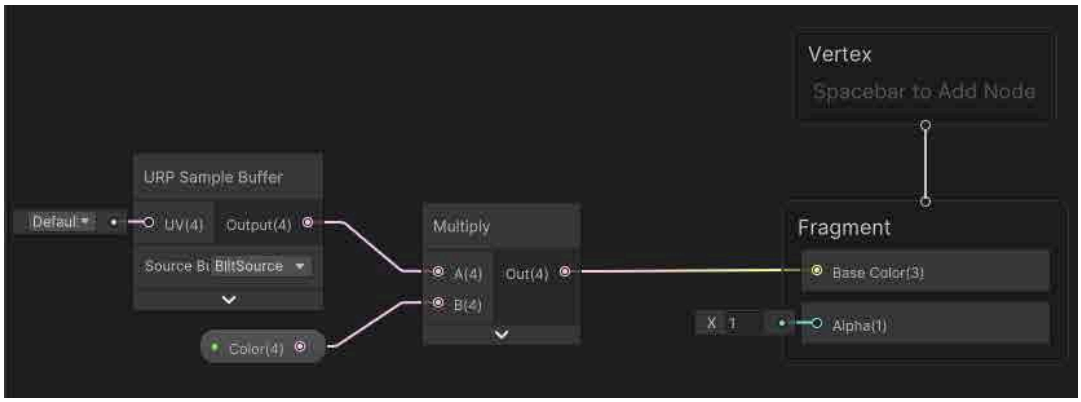
全画面 Shader Graph

全画面 Shader Graph では、カスタムポストプロセスパスを作成できます。Project ペインで右クリックし、**「Create」>「Shader Graph」>「URP」>「Fullscreen Shader Graph」**を選択します。



全画面 Shader Graph の作成

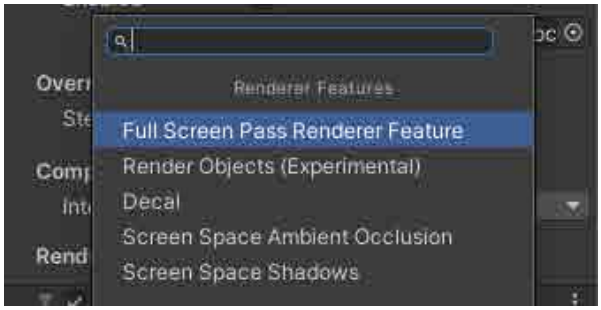
BlitSource オプションを使用する URP Sample Buffer ノードを使用して、フラグメントシェーダーのピクセルの色にアクセスすることができます。以下のグラフは、シンプルな色付けの例です。また、URP Sample Buffer は、エッジ検出やモーショントレイルに役立つワールド法線とモーションベクトルを利用可能にします。



簡単な色付けの例

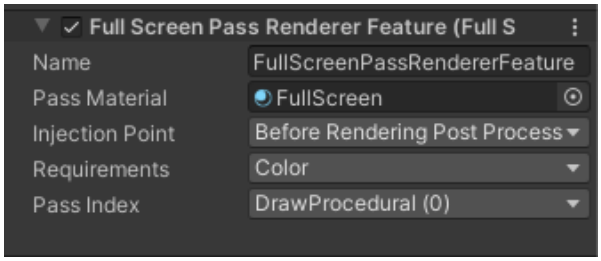
この例を使用するには、このシェーダーを使用するマテリアルを使用して、現在のカメラのレンダーテクスチャの結果を Blit する方法が必要です。

アクティブなレンダラーデータアセットを選択した状態で、Inspector を使用して Renderer Feature を追加します。「**Full Screen Pass Renderer Feature**」を選択します。



Full Screen Pass Renderer Feature を追加

あとは、この Renderer Feature の設定を更新するだけです。作成した全画面 Shader Graph を使用するマテリアルを設定し、レンダーパイプライン内の位置を選択します。



Renderer Feature の設定

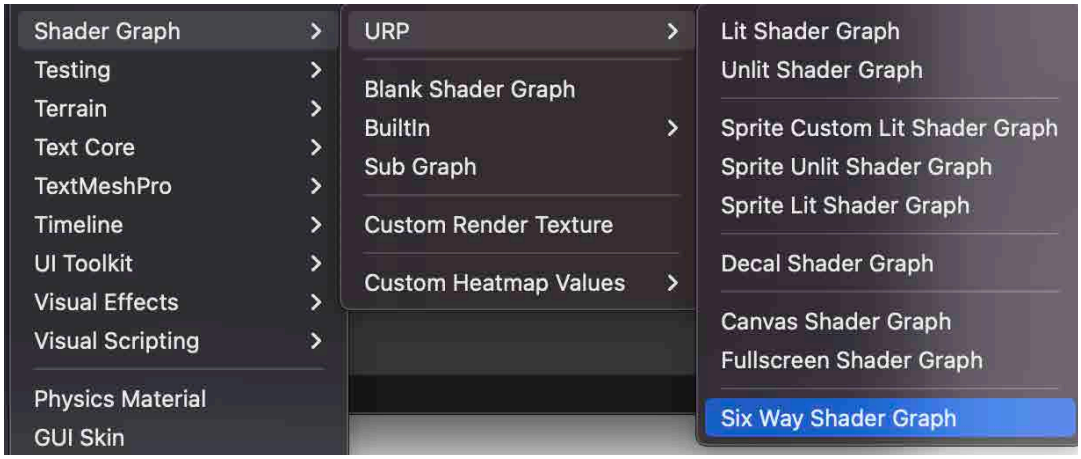
以下の画像では、左側に色付け効果が適用されています。全画面 Shader Graph は、カスタムポストプロセスエフェクトを作成するための便利な方法です。

色付け効果

Six Way Shader Graph

以下は、同じスモーク効果を異なるライティング条件で撮影した 4 つのバージョンです。左上から順に：ディレクショナルライトとアンビエント。右上：プローブボリュームとディレクショナルライト。左下：アンビエントとプローブ。右下：スポットライトとプローブ。

Six Way Shader Graph は、Houdini、Blender、Embergen などの DCC ツールでベイク可能な 6 方向のライトマップを使用して、スモークエフェクトがよりリアルになるよう動的にリライトできる Unity 6 の機能です。



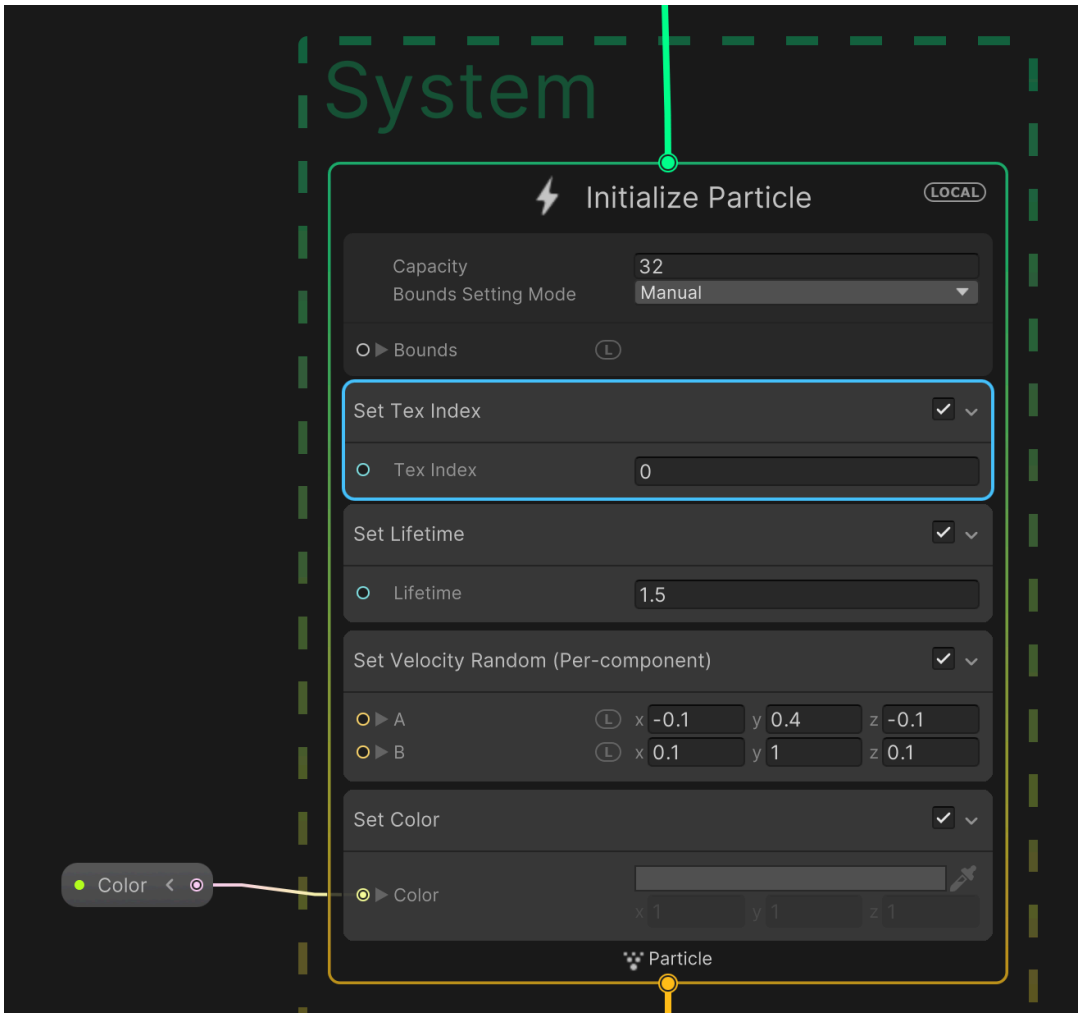
Six Way Shader Graph の作成

6 方向のライティングについて詳しく知るには、「VFX Graph で 6 方向のライティングを使って作るリアルなスモークライティング」のブログ記事をご覧ください。また、このブログ記事では、サンプルプロジェクトといくつかの上級者向けの提案とともに、Shader Graph のプロセスを説明しています。

VFX Graph

Unity では、キャラクターが指先から火の玉を放ったり、ワームホールを通過したりなど、あらゆる視覚効果を実現できます。ゲームの視覚効果（VFX）は、雰囲気を高め、ストーリーを伝え、プレイヤーを真に魅了するディテールを追加します。

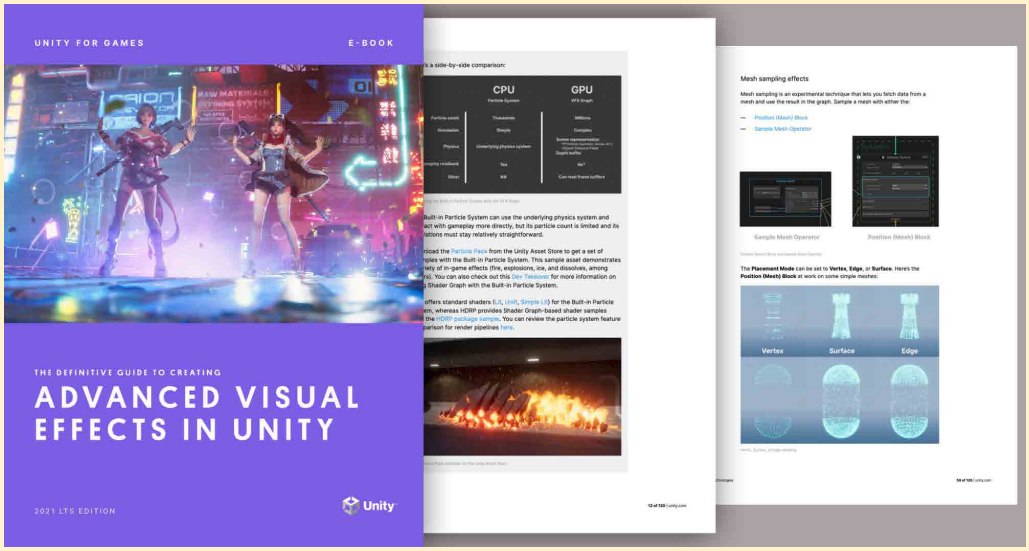
Unity は、VFX Graph などのツールを通してリアルタイムグラフィックスの限界を押し広げています。このノードベースのエディターを使用すれば、テクニカルアーティストや VFX アーティストは、シンプルな一般的なパーティクルの動作から、パーティクル、ライン、リボン、トレイル、メッシュなどを含む複雑なシミュレーションまで、ダイナミックなビジュアルエフェクトをデザインできます。



VFX Graph の編集

URP と VFX Graph を使用した VFX 制作について詳しく知りたい方は、e ブック「*Unity での上級者向け VFX 制作の完全ガイド*」をご覧ください。

VFX Graph を用いて作られたスモーク効果

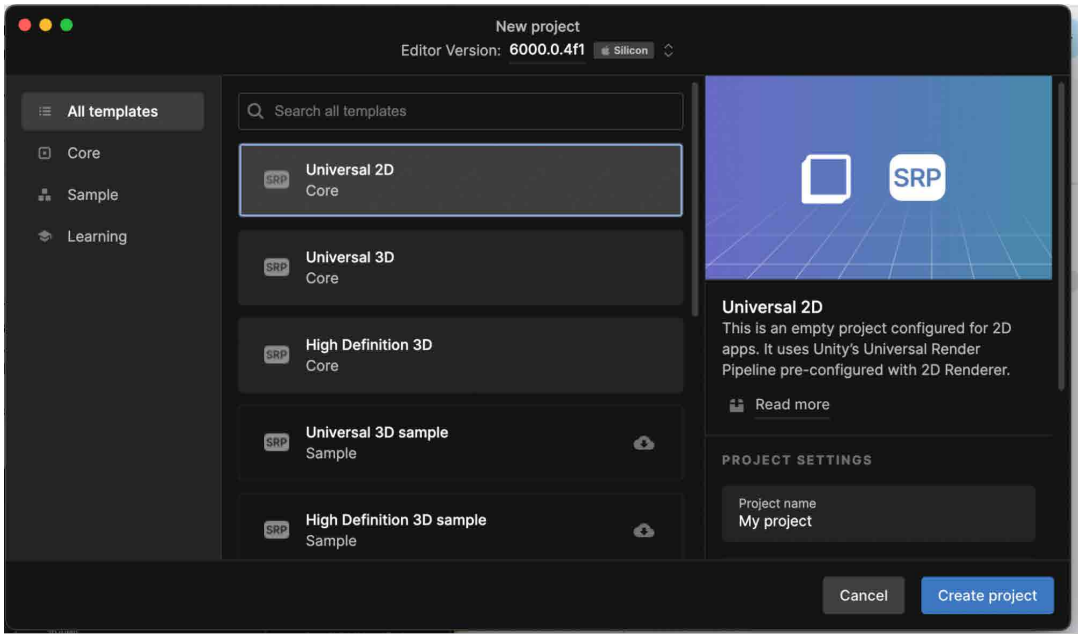


Unity eBook をダウンロード

2D レンダラーと 2D ライト

今日の 2D ゲームがどれほど革新的になれるかに限界はありません。ハードウェア、グラフィックス、そしてゲーム開発ソフトウェアの進化により、リアルタイムライト、高解像度のテクスチャ、ほぼ無制限のスプライトを持つ 2D ゲームを作成することが可能になりました。

2D ゲームを開発している方には、専用の URP 2D レンダラーが用意されています。一番簡単に開始する方法は、Unity Hub の 2D URP テンプレートを使用することです。このテンプレートは、「Project Settings」>「Graphics」>「Scriptable Render Pipeline Settings」を介して、URP 2D レンダラーがプロジェクトに割り当てられるようにします。2D URP テンプレートと共に、検証済みの事前コンパイルされた 2D パッケージがすべてインストールされ、デフォルト設定は 2D プロジェクト向けに最適化されています。これにより、すべてのパッケージを手動でインストールするよりもプロジェクトのロードが速くなります。



Unity Hub の 2D URP テンプレート

既存のプロジェクトをアップグレードする場合は、プロジェクトの Assets フォルダー配下の最適なフォルダーに移動します。右クリックし、「Create」>「Rendering」>「URP Asset (with 2D Renderer)」の順に選択します。名前を付け、「Project Settings」>「Graphics」>「Scriptable Render Pipeline Settings」を使用して選択します。シーンビューで、編集時に必ず 2D ボタンを選択してください。

2D レンダラーと設定アセットを作成

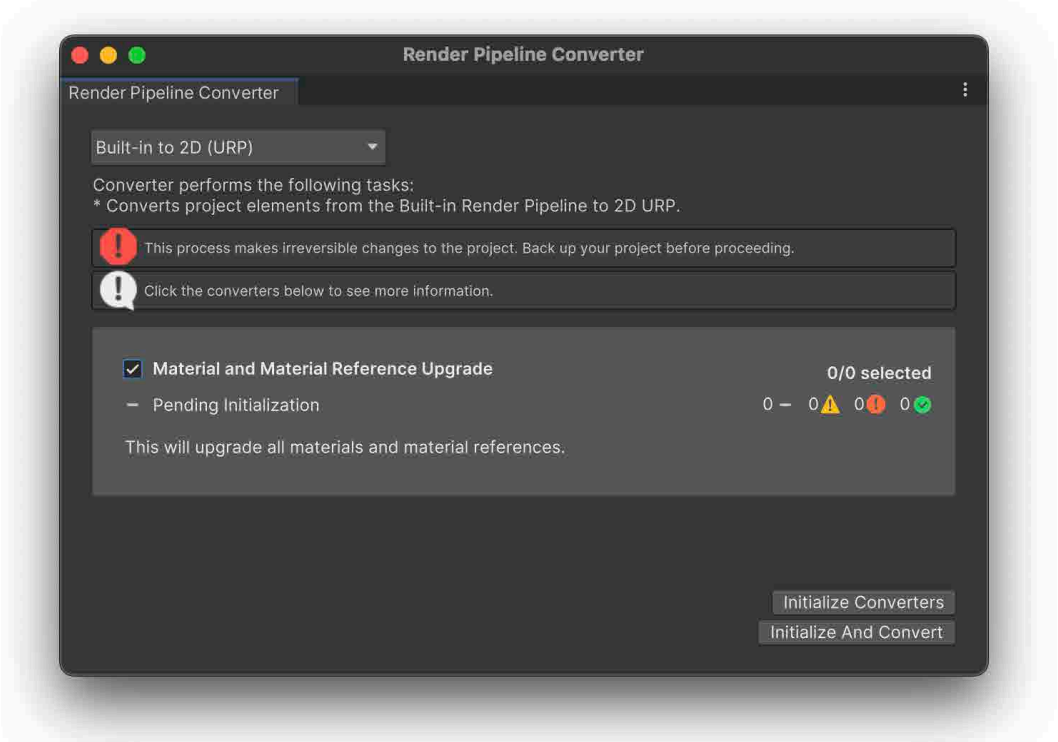
既存のプロジェクトを URP 2D Renderer に切り替える場合、レンダリングエラーを示す「見慣れた」マゼンタ色が表示されることがあります。

既存のプロジェクトを URP 2D レンダラーで更新すると、シーン内にレンダリングエラーが発生する可能性がある

幸い、この問題は**「Window」>「Rendering」>「Render Pipeline Converter」**で解決できます。**「Built-in to 2D (URP)」**を選択し、「Material and Material Reference Upgrade」パネルをクリックします。次に、「Initialize Converters」をクリックした後、「Convert Assets」をクリックしていくつかの項目の選択を解除するか、「Initialize And Convert」でこのプロセスをワンクリックで処理を行います。依然としてマゼンタ色のスプライトが表示される場合は、一部のマテリアルのシェーダーを手動で置き換える必要があるかもしれません。以下の表にリストアップされているシェーダーの 1 つを選択します。

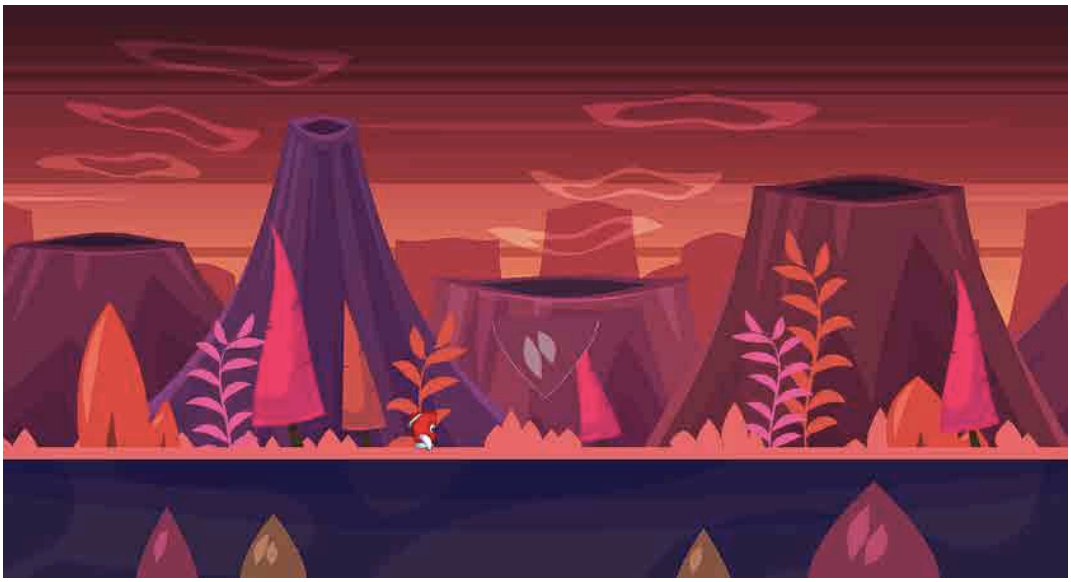
URP で利用可能な 2D シェーダー

シェーダー	説明
Sprite-Lit-Default	レンダリング時に 2D ライトを使用する
Sprite-Mask-Default	ステンシルバッファで動作する
Sprite-Lit-Default	レンダリング時にテクスチャの色のみを使用する



ビルトインレンダーパイプライン 2D プロジェクトを URP 2D に変換する

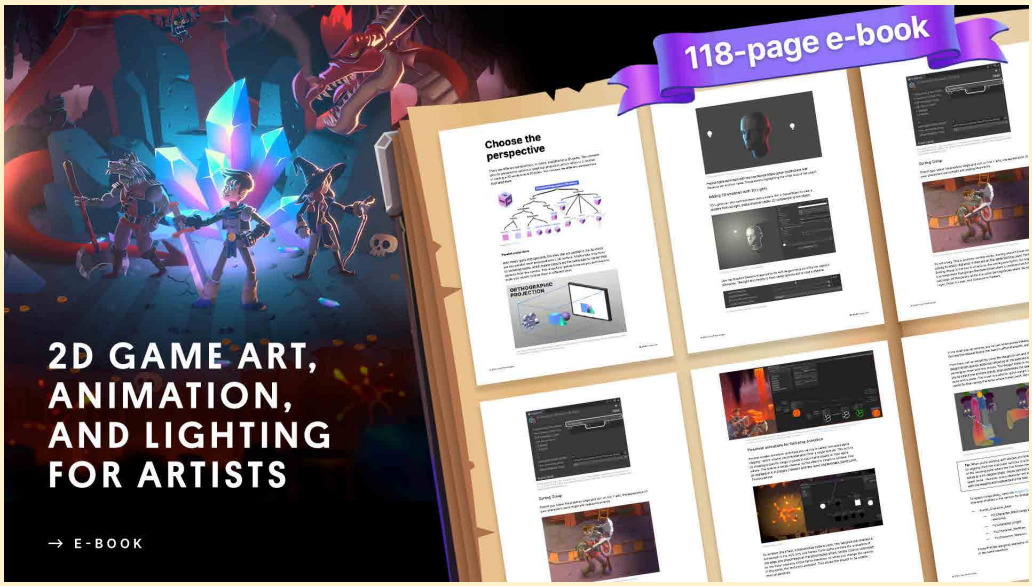
2D ライトは、URP 2D レンダラーで使用できます。これらは、パフォーマンスと柔軟性を強化します。新しいツールを活用することで、より没入感のある体験を作成し、ベイクしたライトを使用して様々なスプライトバリエーションを作成する時間を節約し、新しいゲームプレイの可能性を作り出すことができます。既存のプロジェクトを移行すると、シーン内に URP 2D ライトがない状態になります。スプライトが Sprite-Lit-Default シェーダーを使用している場合、レンダリングにライティングが適用されているのを見て驚くかもしれません。しかし、ライトがない場合、シーンにはデフォルトのグローバルライトが割り当てられ、ライティングのない外観になります。



シーンにライトがない場合、レンダリングはデフォルトで Unlit になります。

Unity の 2D ゲーム開発リソース

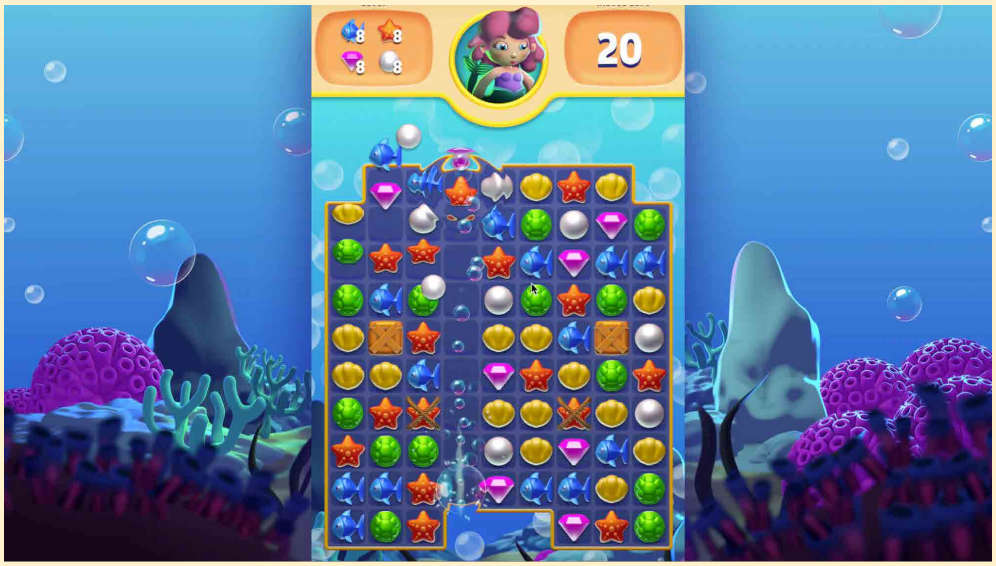
e ブック「*2D game art, animation, and lighting for artists*」は、商用 2D ゲームを制作したい Unity 開発者およびアーティスト向けに作成された、最大かつ最も包括的な開発ガイドです。



2D グラフィックス、ツール、アニメーションに関する Unity e ブックをダウンロード

Unity の 2D サンプルプロジェクト

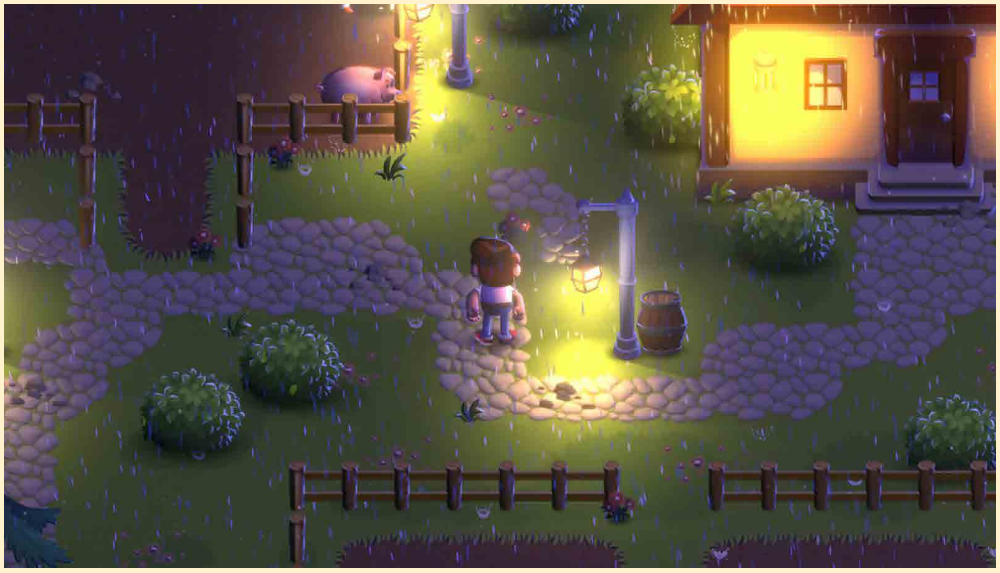
『Gem Hunter Match』



『*Gem Hunter Match*』は、URP で作成された目を引くライティングと視覚効果によって、2D パズル / マッチ 3 ゲームを競合とどう差別化できるかを示しています。2D スプライトに奥行きを持たせるための準備とライティング方法、スプライトを輝かせるためのカスタム Lit シェーダーの適用、グレアや波紋のエフェクト作成など、さまざまな手法を学ぶことができます。

- 『*Gem Hunter Match*』をダウンロード
- 『*Gem Hunter Match*』の紹介を見る

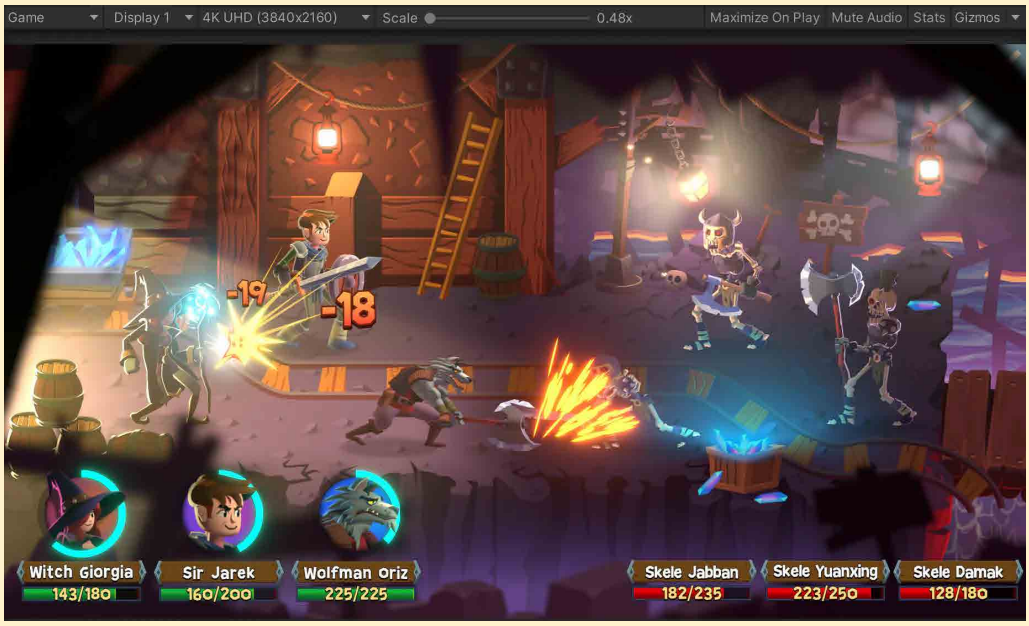
『Happy Harvest』



『*Happy Harvest*』からは、URP で 2D ライト、シャドウ、特殊効果を作成するための最新機能を活用する方法を学ぶことができます。スプライトにシャドウを焼き込まないこと、スプライトをフラットに保つこと、シャドウとボリューム情報を二次的なテクスチャに移動させること、高度なタイルマップ機能など、2D クリエイターなら誰でも使えるベストプラクティスが盛り込まれています。

- 『*Happy Harvest*』をダウンロード
- 『*Happy Harvest*』の紹介を見る

UI Toolkit サンプル –『Dragon Crashers』



この 2D サンプルプロジェクトは、横スクロールの放置系ロールプレイングゲームのバーティカルスライスで、2D ツール一式とアートワークをどのように組み合わせてビジョンを実現できるかを紹介しています。このデモのコンテンツは、全てご自身のクリエイティブなプロジェクトに追加することが可能です。

- *UI Toolkit サンプル –『Dragon Crashers』をダウンロード*
- *UI Toolkit –『Dragon Crashers』の紹介を見る*

Unity 6 の他の URP 機能

このセクションでは、さまざまなデバイスでのパフォーマンスや忠実度を向上させることができる、Unity 6 の URP の主な機能を紹介します。

Spatial-Temporal Post-Processing (STP)

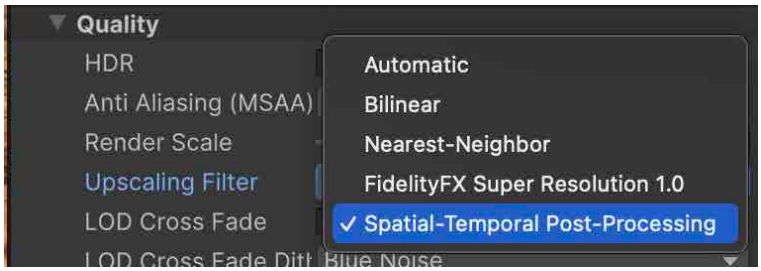
URP 3D サンプルのオアシス環境を 3 つの方法でレンダリングしたもの。左：レンダースケール 1、中央：レンダースケール 0.25、右：レンダースケール 0.25（STP 有効）

競合する超解像技術の数は増え続けていますが、その多くは単一のハードウェアベンダーに特化しています。さらに、これらのソリューションはモバイル向けに設計されていません。

Spatial-Temporal Post-Processing (STP) は、Unity の超解像ソリューションで、モバイルからコンソールまであらゆるデバイスで動作し、オーバーヘッドを抑えつつ URP で高品質な結果を実現します。上の重ね合わせた画像からわかるように、高品質な結果が得られます。画像の左側は、STP なし、レンダースケールは 1 です。中央は、STP なし、レンダースケール 0.25 で、ぼやけて忠実度が低くなっています。右側は、レンダースケールを 0.25 に保ちつつ STP を有効にしており、描画するピクセル数を 1/16 に抑えながら、STP アップスケーリングでシャープに見せています。

STP は、シェーダーモデル 5.0 をサポートするあらゆるデバイスで動作する、時空間アンチエイリアスアップスケーリングソリューションです。モバイルを念頭に置いて設計されており、DLSS2、FSR2、XeSS に匹敵するビジュアル品質を、より低いパフォーマンスコストで提供することを目指しています。

STP を有効化するには、アクティブな URP アセットの**「Quality」>「Upscaling Filter」**から設定します。その後、レンダースケールを設定します。



Spatial-Temporal Post-Processing (STP) の有効化

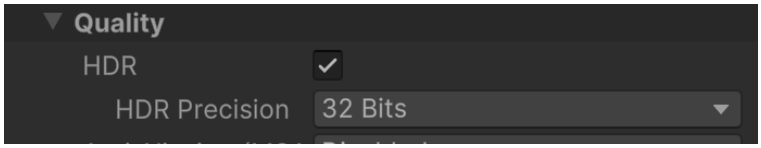
PC およびコンソール用ハイダイナミックレンジディスプレイ出力

「Rendering Debugger」 > 「Lighting」 > 「HDR Debug Mode」 > 「Gamut View」

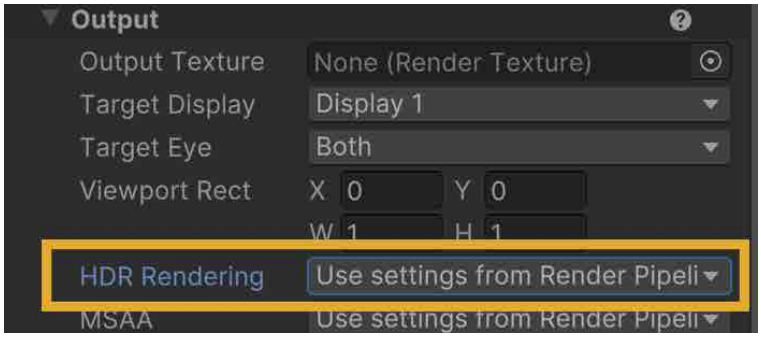
ハイダイナミックレンジ (HDR) ディスプレイは、より自然光に近い輝度差を再現できるディスプレイデバイスです。HDR 出力を行うことで、これらのデバイスで表示されるリニアライティングのレンダリングや HDR 画像のコントラストと品質をより効果的に保持できます。

HDR は、以下の 2 か所で有効化する必要があります。

1. URP アセット



2. Camera Inspector：**「Output」 > 「HDR Rendering」**を、**レンダーパイプラインアセットの設定を使う**ように設定します。



HDR では、このオプションをオンにすることで、HDR ディスプレイ上で URP によるより高品質な画像レンダー出力を利用できます。その結果、これらのデバイスでは、自然なライティング条件をより効果的に再現した色やコントラストのある画像をより忠実に表示できます。

Unity 2022 で利用可能になった既存のデスクトップおよびコンソール向けサポートに加え、Unity 6 では以下のプラットフォームでモバイル向けサポートが導入されました。

- iOS プレイヤー（iOS 16+、iPadOS 16+）
- Vulkan および GLES を使用している Android プレイヤー（Android 9+、デバイスの能力による）

HDR ディスプレイをサポートする一般的なモバイル機器には、iPhone X 以降、Samsung の Galaxy S10 以降、Galaxy Note 10 以降、Galaxy Tab S6 以降などが含まれます。

Pipeline State Object の使用

Unity 6 では、Pipeline State Object（PSO）のトレースおよびプリックの強力なワークフローが導入され、最新のプラットフォームをターゲットにした際に、よりスムーズでスタッターが少ないプレイヤー体験を実現できます。

この API セットは、以前の Unity バージョンで導入された「シェーダーウォームアップ」API から大幅に強化されています。従来のシェーダーウォームアップは古いグラフィックス API（OpenGL、DirectX11 など）には十分ですが、新しい PSO ワークフローにより、開発者は Vulkan、DirectX12、Metal などの最新のグラフィックス API をより効果的に活用できます。

PSO の作成とキャッシング

最新のグラフィックス API をターゲットとする場合、GPU ベンダーのグラフィックスドライバーは、PSO の作成プロセスの一環としてランタイムシェーダのコンパイル（およびその他のレンダリング状態の変換）を実行します。その結果、PSO の作成には時間がかかり、実行時のアプリケーションに顕著なスタッターが発生する可能性があります。このオーバーヘッドは、アプリケーションが大量の PSO を動的にコンパイルする必要がある複雑なプロジェクトでは悪化する可能性があります。

Unity のプロファイラーでは、GraphicsPipelineImpl マーカーを使用して PSO 作成時のスタッターを特定できます。

PSO 作成のプロファイリング

多くの場合、GPU ベンダーのグラフィックスドライバーは、コンパイルされた PSO を自動的にディスクにキャッシュし、その後のアプリケーション実行時の PSO 作成を高速化します。しかし、アプリケーションは、新しく遭遇したシェーダーバリアントやマテリアル用に PSO をコンパイルする必要があるかもしれません。さらに、OS やドライバーのアップデートにより、ドライバーが管理する PSO キャッシュが無効になることも多いです。

PSO の最適な事前準備方法は、アプリケーションやユースケースによって異なる場合があります。例えば、レベル遷移やシーンのロード中に PSO を同期的に事前準備する方法を選択することもできます。これは、アプリケーションの応答性を高めるため、フレームごとに一定数の PSO を作成しながらプログレッシブに（タイムスライスで）実行することができます。

または、アプリケーションのバックグラウンドで PSO を非同期で事前準備することもできます。これによりアプリケーションが停止することはありませんが、ウォームアップ中は CPU パフォーマンスが一時的に低下する可能性があります。

新たな PSO コレクションのトレーシング

まず、レンダリング中にアプリケーションが生成する PSO をトレースする必要があります。

1. C# スクリプトで GraphicsStateCollection を新規作成します。このコレクションは、アプリケーションまたはシーンの PSO に対応します。
2. PSO をコレクションにトレースし始めるには、GraphicsStateCollection.BeginTrace メソッドを呼び出します。アプリケーションによって作成された新しいグラフィックスパイプラインが、コレクションに追加されます。ほとんどの場合、シーンやアプリケーションの起動時にトレースを開始するのが望ましいです。
3. トレースを終了するには、GraphicsStateCollection.EndTrace メソッドを呼び出します。ほとんどの場合、シーンやアプリケーションの終了時にトレースを終了するのが望ましいです。

トレースが完了したら、GraphicsStateCollection.SaveToFile を使用して PSO コレクションをディスクに保存します。

さらに細かく制御したい場合は、記録された PSO とバリアントデータを検査し、必要に応じてコレクションを修正できます。GraphicsStateCollection.GetVariants を使用すると、PSO コレクションに記録されているすべてのシェーダーバリアントを取得できます。その後、GraphicsStateCollection.GetGraphicsStatesForVariant を使用して、各バリアントで使用されるグラフィックスステートを読み取ることができます。最後に、AddGraphicsStateForVariant / RemoveGraphicsStatesForVariant を使用して、各バリアントに関連付けられたグラフィックスステートを変更します。

注：

PSO の GPU 表現はプラットフォームによって異なる場合があります。関連するグラフィックス API をターゲットにしてプレイヤー内でトレースを実行し、ターゲットごとに別々のコレクションを保持することを強く推奨します。

Player Connection を使用してターゲットデバイスをトレースする場合、GraphicsStateCollection.SendToEditor を使用して PSO コレクションをエディターに送信し、ディスクに保存できます。また、GraphicsStateCollection.runtimePlatform で、PSO コレクションをトレースする際の使用プラットフォームをクエリすることもできます。

PSO コレクションの事前準備

トレースが完了したら、Unity に PSO コレクションの事前準備をリクエストできます。これは描画時よりも前に行うのが理想的です。ほとんどの場合、ウォームアップを行う理想的なタイミングは、アプリケーションやシーンのロードシーケンス中です。

PSO の事前準備は、2 つのウォームアップメソッドを介して行えます。

- GraphicsStateCollection.WarmUp は、コレクション内のすべての PSO の作成をスケジュールします。
- GraphicsStateCollection.WarmUpProgressively は、コレクション内の一定数の PSO の作成をスケジュールします。

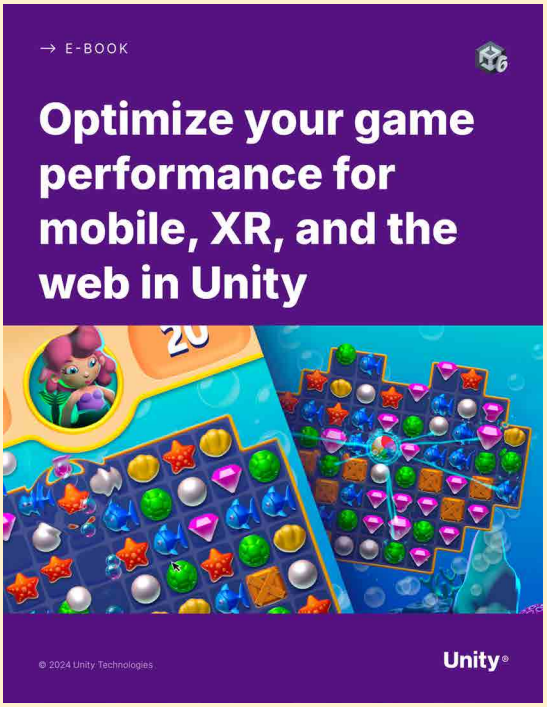
どちらのメソッドもジョブハンドルを返し、それを使って PSO ウォームアップが同期実行か非同期実行かを判定できます。

PSO が作成されると、多くの場合、ドライバーはそれをディスクにキャッシュします。次回以降、PSO が事前準備される際に、キャッシュから直接ロードできるようになります。

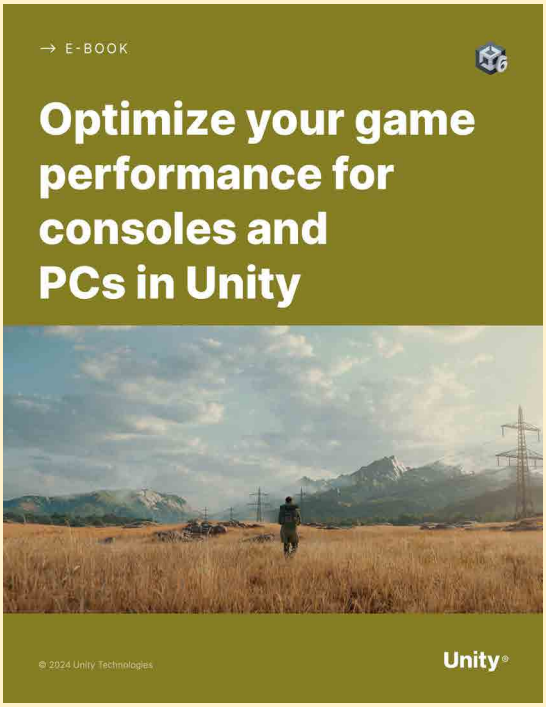
プラットフォームサポート

新しい PSO ワークフローは Unity 6 から利用可能で、Metal、Vulkan、Direct3D 12 をターゲットとするプレイヤーで使用できます。今後のバージョンでは、OpenGLES や Direct3D 11 のような古いグラフィックス API に対しても、暗示的なシェーダーウォームアップのフォールバックという形で互換性を提供する予定です。

パフォーマンス

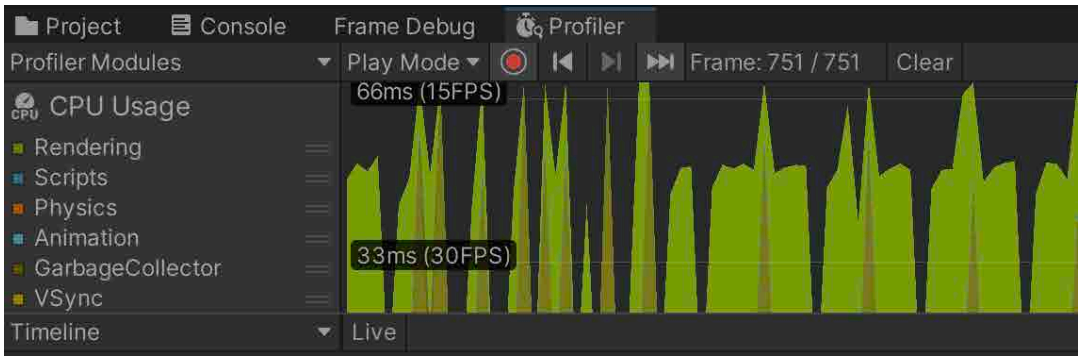


Unity のプロファイリングツール、プログラミングやコードアーキテクチャ、プロジェクト構成やアセットなどを使用するためのヒントを得ることができます。モバイルゲームのパフォーマンスを向上させる方法を学びましょう。
ダウンロード



コンソールや PC プロジェクトの詳細なプロファイリング、プログラミングコードとアーキテクチャ、アセットとグラフィックスの最適化、UI、物理演算、アニメーションの最適化に役立つヒントを手に入れましょう。
ダウンロード

パフォーマンスは作業中のプロジェクトに大きく依存します。常にプロファイリングを行い、開発サイクル全体を通してゲームをテストしてください。**「Window」 > 「Analysis」 > 「Profiler」**でプロファイラーを開き、この章の提案事項に従ってください。



Profiler ウィンドウ

このセクションでは、ゲームのパフォーマンスを向上させる 7 つの方法について説明します。

- ライティングの管理
- ライトプローブ
- リフレクションプローブ
- カメラ設定
- パイプライン設定
- フレームデバッガー
- プロファイラー

これらの最適化については、このチュートリアルでも説明しています。

プロファイリングのヒントについては、以下のビデオチュートリアルをご覧ください。

視聴する

視聴する

視聴する

URP でのライティングとレンダリングの最適化

URP は最適化されたリアルタイムライティングを念頭に構築されています。URP フォワードレンダラーは、オブジェクトあたり最大 8 つのリアルタイムライトと、デスクトップゲーム向けにはカメラあたり最大 256 のリアルタイムライト、モバイルやその他のハンドヘルドハードウェア向けにはカメラあたり 32 のリアルタイムライトをサポートしています。また、URP では、パイプラインアセット内でオブジェクトごとにライトの設定を行うことができ、ライティングをより細かく制御することができます。

ライティングの章で説明したように、ベイクしたライティングは、シーンのパフォーマンスを向上させる最良の方法の 1 つです。リアルタイムライティングはコストが高くなる可能性がありますが、シーン内のライトが静的である場合、ベイクされたライトによってパフォーマンスを回復することができます。ベイクされたライティングテクスチャは、継続的に計算する必要がなく、1 回のドローコールにまとめられます。これは、シーンで複数のライトが使用される場合に特に便利です。ライティングをベイクするもう 1 つの大きな理由は、シーン内で反射光または間接光をレンダリングし、ビジュアル品質を向上させることができることです。

グローバルイルミネーションについても同様にライティングのセクションで取り上げています。このプロセスは、環境内で反射する光線をシミュレートし、反射光で周囲のオブジェクトを照らします。下図は、同じシーンの 3 つのライティング設定を示しています。ベイクされたライトデータなし、ベイクされたライティングあり、そしてポストプロセスが適用されたものです。

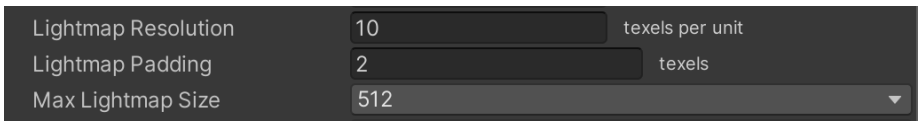
左から順に、ライティングデータなし、ベイクされたライティングあり、ポストプロセス追加

ベイクすると、シーン内の影の部分が反射光を受けて照らされます。効果は控え目かもしれませんが、このテクニックはシーン全体によりリアルに光を拡散させ、全体的な見た目を向上させます。

前の画像を見ると、ベイク時に地面のスペキュラーハイライトが失われるのが分かります。ベイクされたライトは、ディフューズライトのみ含みます。可能な限り、直接光の影響をリアルタイムで計算し、グローバルイルミネーションは画像ベースライティング（IBL）/ シャドウマップ / プローブから取得するようにしてください。

影に対するライトベイクの効果：左がベイク前、右がベイク後

ライトをベイクする時は、**Lightmap Resolution** と **Lightmap Size** の最小値を使用してください。**「Window」>「Rendering」>「Lighting」>「Scene」**に移動します。これはテクスチャのメモリ使用量を減らすのに役立ちます。



Lightmap Resolution と Max Lightmap Size を設定する

ライトプローブ

ライティングのセクションで説明したように、ライトプローブはベイク中にシーン内のライティングデータをサンプリングし、動的オブジェクトが移動したり変化したりする際に、反射光の情報を使用できるようにします。これにより、オブジェクトはベイクされたライティング環境に溶け込み、より自然に感じられるようになります。（Unity エンジンにはライトプローブに代わるものが実装されました。詳しくは APV のセクションをご覧ください。）

ライトプローブは、レンダリングフレームの処理時間を増やすことなく、レンダリングに自然さを加えます。そのため、ローエンドのモバイルデバイスも含む、すべてのハードウェアに最適です。

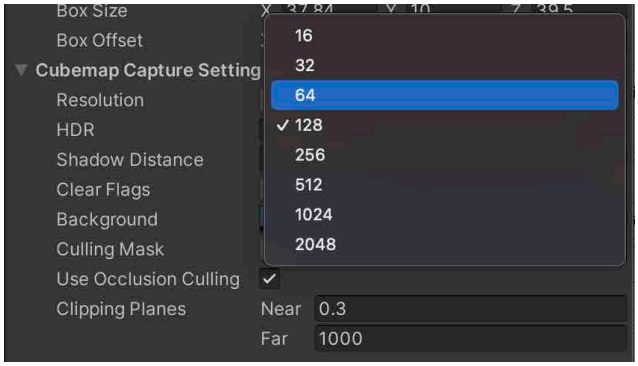
動的オブジェクトのレンダリング時にライトプローブを使用した場合の効果：ライトプローブあり（左）、ライトプローブなし（右）

リフレクションプローブ

リフレクションプローブを使用してシーンを最適化することもできます。リフレクションプローブは、環境の一部を近くのジオメトリに投影して、よりリアルな反射を再現します。デフォルトでは、Unity はリフレクションマップとしてスカイボックスを使用します。しかし、1 つ以上のリフレクションプローブを使用することで、周囲の環境をよりリアルに再現した反射を得られます。

滑らかなサーフェスでリフレクションプローブを使用した場合の効果：リフレクションプローブあり（左）、リフレクションプローブなし（右）

リフレクションプローブをベイクする際に生成されるキューブマップのサイズは、カメラが反射するオブジェクトにどれだけ近づくかによって異なります。必ずニーズに合った最小のマップサイズを使用してシーンを最適化してください。



リフレクションプローブキューブマップのサイズの調整

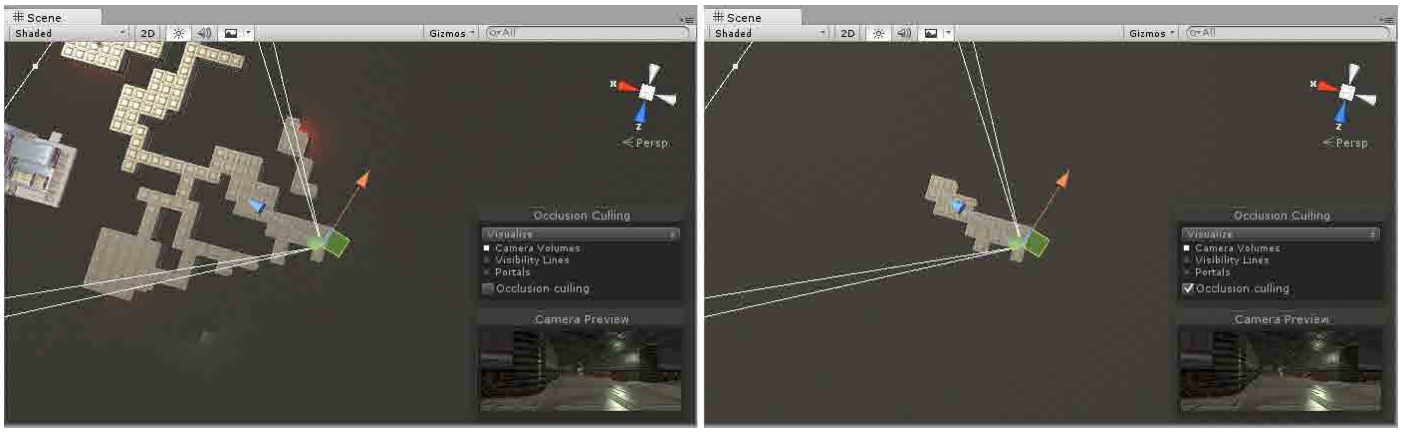
カメラ設定

URP を使用すると、パフォーマンス最適化のために、カメラの不要なレンダラー処理を無効化することができます。これは、プロジェクトでハイエンドとローエンドの両方のデバイスをターゲットにしている場合に便利です。ポストプロセス、シャドウレンダリング、深度テクスチャなど、負荷の高いプロセスを無効化すると、ビジュアルの忠実度は下がりますが、ローエンドデバイスでのパフォーマンスが向上します。

オクルージョンカリング

カメラを最適化するもう 1 つの優れた方法はオクルージョン カリングです。デフォルトでは、Unity のカメラは、壁や他のオブジェクトの背後に隠れている可能性のあるジオメトリを含め、常にカメラの錐台内のすべてを描画します。プレイヤーから見えないジオメトリを描画するのは無駄であり、貴重なミリ秒を消費してしまいます。そこで、オクルージョンカリングの出番です。

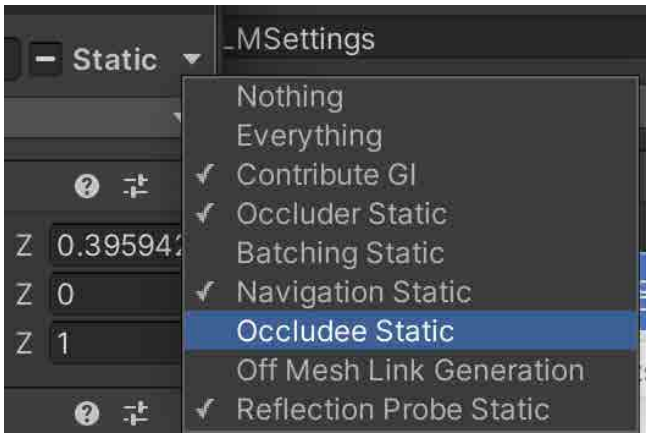
オクルージョンカリングは、カメラとの間に別のアイテムが出現すると、その後ろにある多くのオブジェクトが隠れてしまうようなシーンに最適です。下の画像で示されているように、セル状の通路からなる迷路型ゲームは、オクルージョンカリングを使用するのに理想的な候補です。



フルスタムカリング（左の画像）、オクルージョンカリング（右の画像）

オクルージョンデータをベイクすることで、Unity はシーンの遮られて見えない部分を無視します。フレームごとに描画されるジオメトリを減らすことで、パフォーマンスが大幅に向上します。

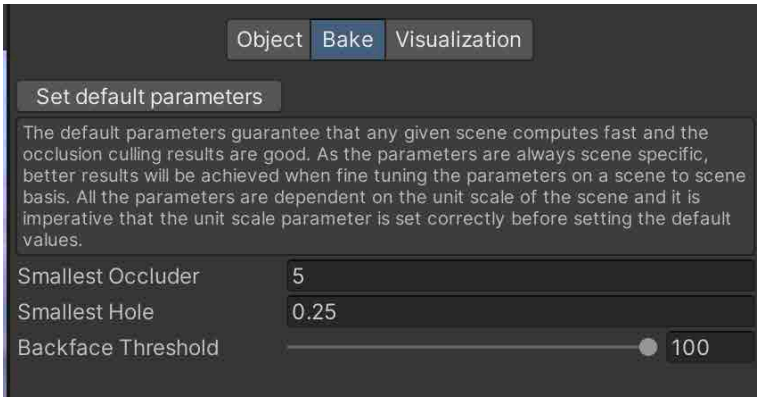
シーンでオクルージョンカリングを有効にするには、任意のジオメトリに **Occluder Static** または **Occludee Static** のマークを付けます。遮蔽物は、被遮蔽物としてマークされたオブジェクトをオクルードできる中型から大型のオブジェクトです。遮蔽物になるオブジェクトは、不透明で、地形またはメッシュレンダラーコンポーネントを持ち、実行時に移動しない必要があります。被遮蔽物はレンダラーコンポーネントを持つオブジェクトであれば何でもよく、同様にランタイムに動かない小さなオブジェクトや透明なオブジェクトも含まれます。



静的プロパティは、通常のドロップダウンで設定します。

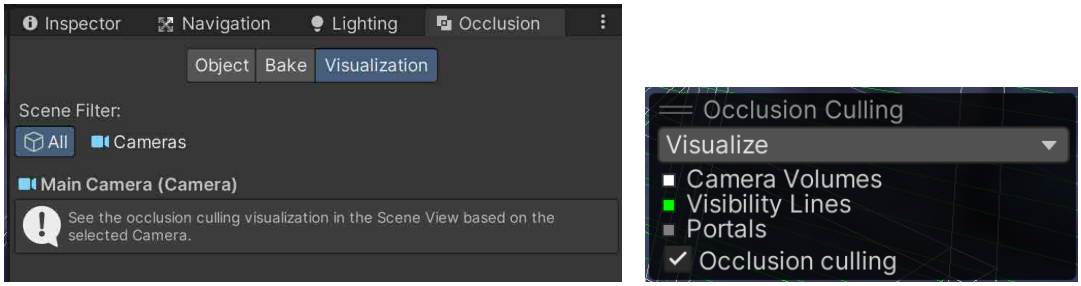
オクルージョンデータに含まれるオブジェクトの設定

「Window」>「Rendering」>「Occlusion Culling」を開き、「**Bake**」タブを選択します。**Inspector** の右下にある「**Bake**」を押します。Unity がオクルージョンデータを生成し、データをプロジェクトのアセットとして保存して、アセットを現在のシーンにリンクします。



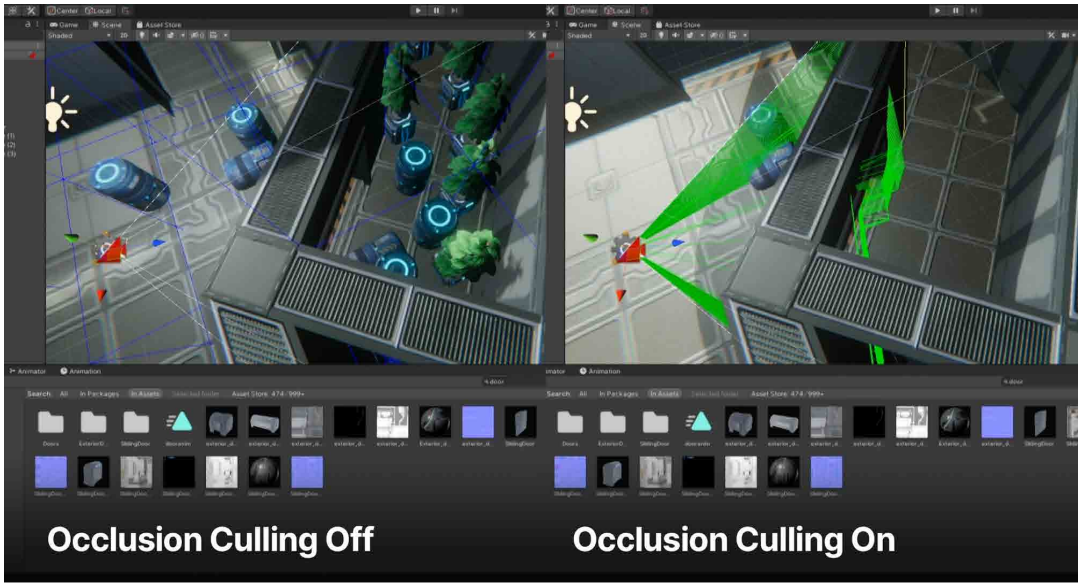
オクルージョンカリングの「Bake」タブ

「**Visualization**」タブを使用して、オクルージョンカリングの動作を確認できます。シーン内の**カメラ**を選択し、**シーン**ビューの**オクルージョンカリング**ポップアップウィンドウを使ってビジュアライゼーションを設定します。ポップアップは小さなカメラビューウィンドウの後ろに隠れている場合があります。その場合は二重線アイコンを右クリックして「**Collapse**」を選択します。ポップアップを移動したら、右クリックして展開しカメラビューを復元します。



Visualization タブとオクルージョンカリングポップアップ

カメラを動かすと、オブジェクトが現れたり消えたりするのがわかるはずです。



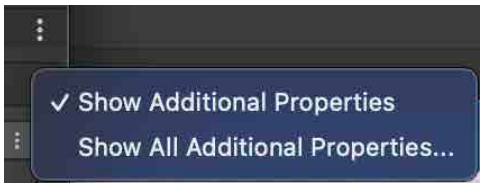
オクルージョンカリングがオフの場合の効果（左の画像）とオンの場合の効果（右の画像）

Unity 6 で GPU Resident Drawer を使用している場合、GPU オクルージョンカリングを使用できます。

パイプライン設定

URP アセットの設定を変更し、異なる品質レベルを使用することの効果については、すでに説明しましたが、ここではプロジェクトに最適な結果を得るために品質レベルを試すための追加のヒントをいくつか紹介します。

- パフォーマンス向上のために影の解像度と距離を減らします。
- 深度テクスチャや不透明テクスチャなど、プロジェクトが必要としない機能を無効化します。
- SRP Batcher を有効にして新しいバッチ処理方法を使用します。SRP Batcher は、同じシェーダーバリアントを使用するメッシュを自動的にバッチ処理し、ドローコールを減らします。シーンに多数の動的オブジェクトがある場合、これはパフォーマンスを上げる有効な方法です。SRP Batcher のチェックボックスが表示されていない場合は、縦に並んだ 3 つのドットアイコン (⋮) をクリックし、「**Show Additional Properties**」を選択します。



URP アセットの Inspector で追加プロパティを有効化する

フレームデバッガー

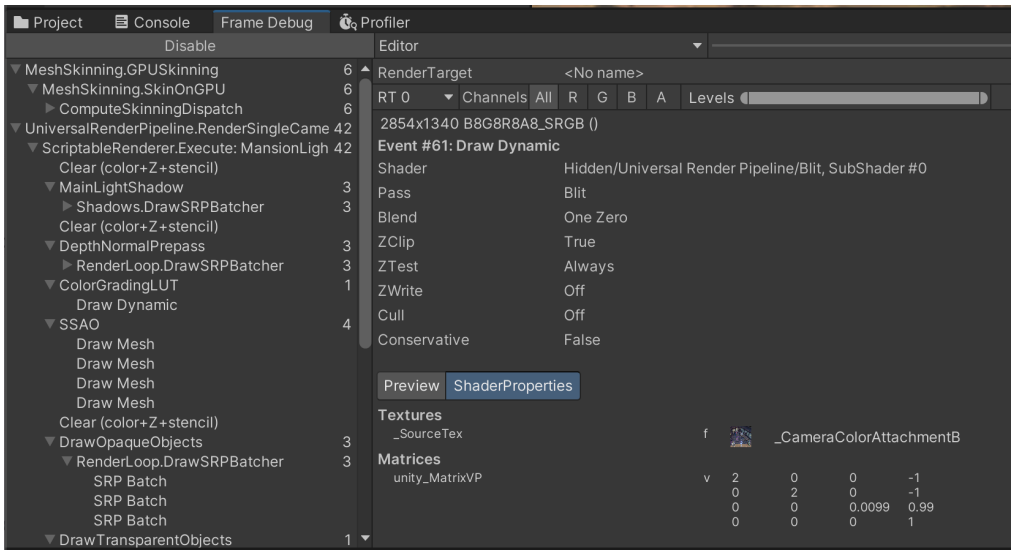
フレームデバッガーを使用すると、レンダリング中の挙動をよりよく理解できます。Frame Debugger ウィンドウで追加情報を表示するには、**URP アセット**を使用して**デバッグレベル**を調整します。SRP Batcher のチェックボックスと同様に、これは **Show Additional Properties** が有効になっている Inspector にのみ表示されます。

デバッグレベルの設定

デバッグレベルを調整すると、パフォーマンスに影響することがあります。フレームデバッガーを使用していないときは、常にオフにしてください。

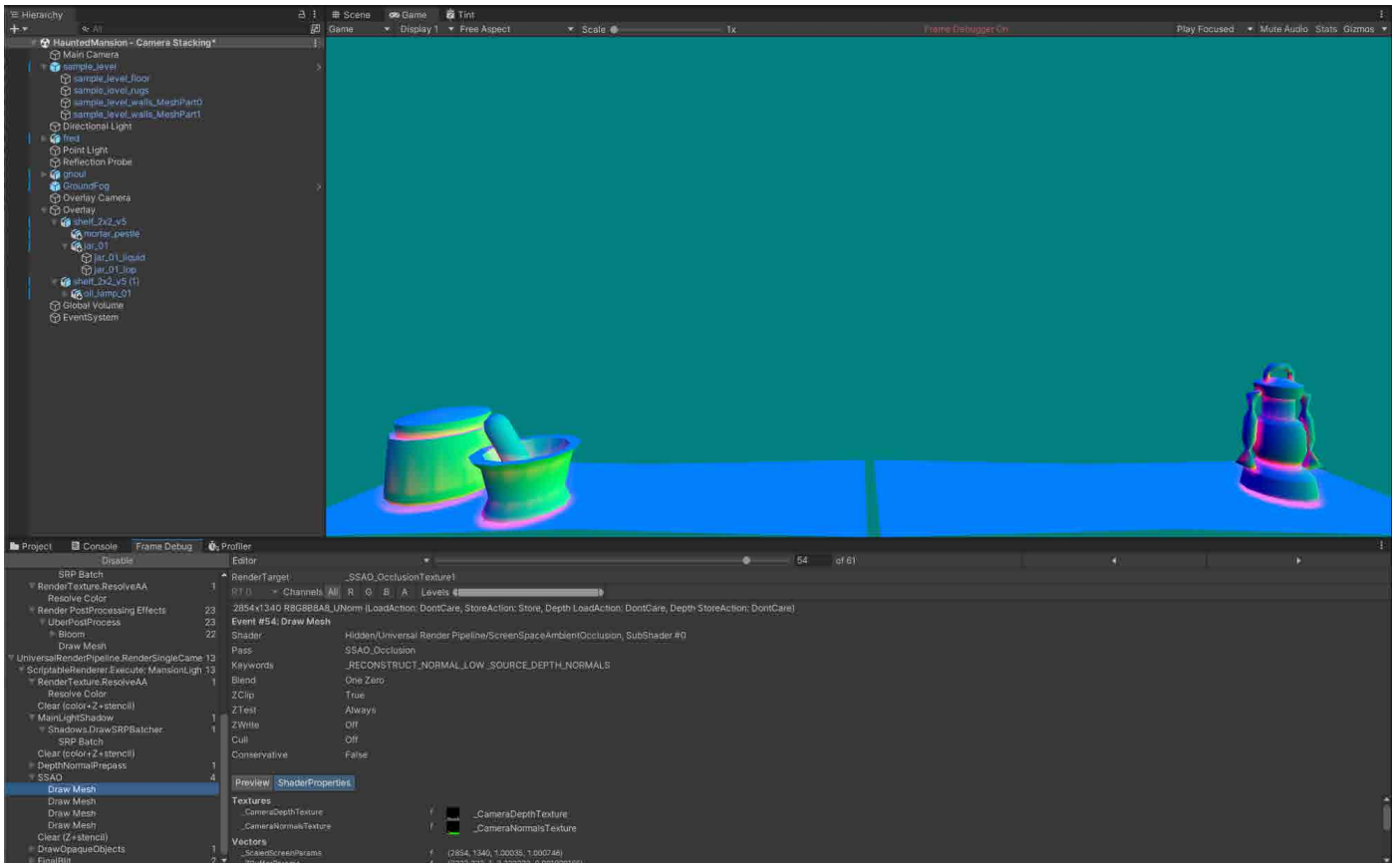
フレームデバッガーは、最終画像をレンダリングするまでに実行されたすべてのドローコールのリストを表示し、特定のフレームのレンダリングに時間がかかっている理由を特定するのに役立ちます。また、シーンのドローコール数が非常に多い理由を特定することもできます。

「Window」>「Analysis」>「Frame Debugger」をクリックしてフレームデバッガーを開きます。ゲームの再生中に「**Enable**」ボタンを選択します。これでゲームが一時停止され、ドローコールを調査できます。



フレームデバッガーの詳細

レンダーパイプライン（左側のペイン）のステージをクリックすると、**ゲーム**ビューにそのステージのプレビューが表示されます。



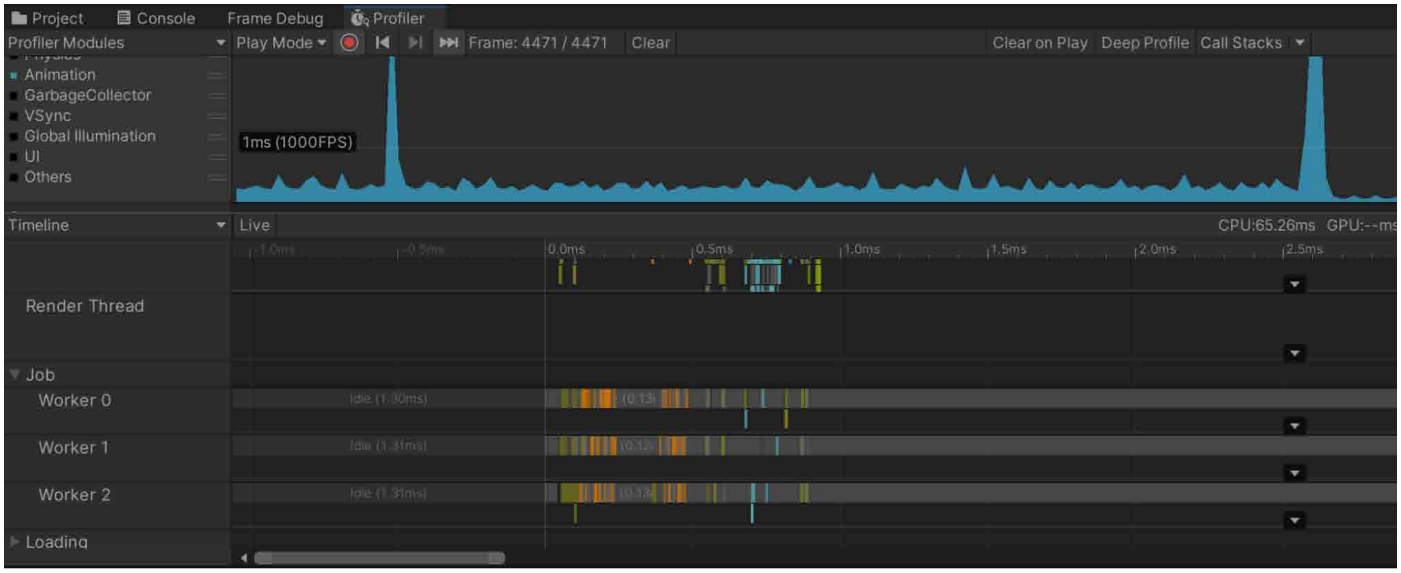
フレームデバッガーは、ゲームビューのレンダリングプロセスの各ステップ（この場合は SSAO 生成ステップ）を表示します。

Unity プロファイラー

フレームデバッガーと同様に、プロファイラーはプロジェクトでフレームサイクルを完了するのにかかる時間を判断するのに最適な方法です。プロファイラーでは、レンダリング、メモリ、スクリプトの概要を確認できます。完了までに時間がかかるスクリプトを特定できるので、コードの潜在的なボトルネックを突き止めるのに役立ちます。

「Window」>「Analysis」>「Profiler」から、プロファイラーを開きます。**再生モード**では、このウィンドウでゲーム全体のパフォーマンスの概要を確認できます。また、ライブビューを一時停止して**階層モード**を使用すると、1 つのフレームを完了するまでにかかった時間の詳細を見ることができます。プロファイラーはフレーム内で Unity が実行した各コールを表示します。

さらに詳細な分析を行うには、低レベルのネイティブプラグイン Profiler API を使用します。この Profiler API を使用してプロファイラーを拡張し、ネイティブプラグインコードのパフォーマンスをプロファイリングしたり、Sony Playstation の Razor、Microsoft（Windows と Xbox）の PIX や、Chrome Tracing、ETW、ITT、VTune、Telemetry などのサードパーティ製のプロファイリングツールに送信するプロファイリングデータを準備することができます。



Profiler ウィンドウで低レベルのネイティブプラグイン Profiler API を使用している

以下は、低レベルのネイティブプラグイン Profiler API の使用例です。

```
#include <IUnityInterface.h>
#include <IUnityProfiler.h>
static IUnityProfiler* s_UnityProfiler = NULL;
static const UnityProfilerMarkerDesc* s_MyPluginMarker = NULL;
static bool s_IsDevelopmentBuild = false;
static void MyPluginWorkMethod()
{
	if (s_IsDevelopmentBuild)
	s_UnityProfiler->BeginSample(s_MyPluginMarker);
	// Unity Profiler で「MyPluginMethod」として表示させたいコード
	// ...
	if (s_IsDevelopmentBuild)
	s_UnityProfiler->EndSample(s_MyPluginMarker);
}
extern “C” void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPluginLoad(IUnityInterfaces*
unityInterfaces)
{
	s_UnityProfiler = unityInterfaces->Get<IUnityProfiler>();
	if (s_UnityProfiler == NULL)
	return;
	s_IsDevelopmentBuild = s_UnityProfiler->IsAvailable() != 0;
```

```
        s_UnityProfiler->CreateMarker(&s_MyPluginMarker,
        “MyPluginMethod”, kUnityProfilerCategoryOther,
        kUnityProfilerMarkerFlagDefault, 0);
}
extern “C” void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API UnityPluginUnload()
{
        s_UnityProfiler  = NULL;
}
```

追加リソース

Unity で高度なプロファイリングスキルを構築したい場合は、まず無料の e ブック「*Ultimate guide to profiling Unity games*」をダウンロードしましょう。このガイドには、Unity でのアプリケーションのプロファイリング、メモリの管理、消費電力の最適化に関する高度なアドバイスや知識がまとめられています。

他の有用なリソースには、Catlike Coding による「Measuring Performance」、The Gamedev Guru による「Unity Draw Call Batching」などがあります。

URP の 3D サンプル

URP の 3D サンプルは Unity Hub で入手可能です。URP で構築された様々な 3D プロジェクトを表現するために、4 つの環境が用意されており、それぞれが独自のアートスタイル、レンダリングパス、シーンの複雑さを備えています。

この URP3D サンプルは、数年間 URP を使用してきた多くの開発者にとって馴染みのある建設現場のシーンを置き換えます。新しいサンプルプロジェクトには、URP の機能を説明するいくつかのミニシーンが含まれています。

それぞれのシーンを見ていきましょう。

Garden

このシーンは、モバイルやコンソールからハイエンドのゲーミングデスクトップまで、URP を使用して複数のプラットフォームに合わせて効率的にコンテンツをスケールする方法を示しています。様式化された PBR レンダリング、カスタマイズ可能な植生、新しいフォワード + レンダラーを使用した、従来のライト数の制限を超える多数のライトのレンダリング機能が特徴です。

Oasis

これは、非常に詳細なテクスチャ、VFX Graph エフェクト、SpeedTree、およびカスタム水ソリューションを使用した写実的なシーンです。コンピュートシェーダーをサポートするデバイスをターゲットにしています。

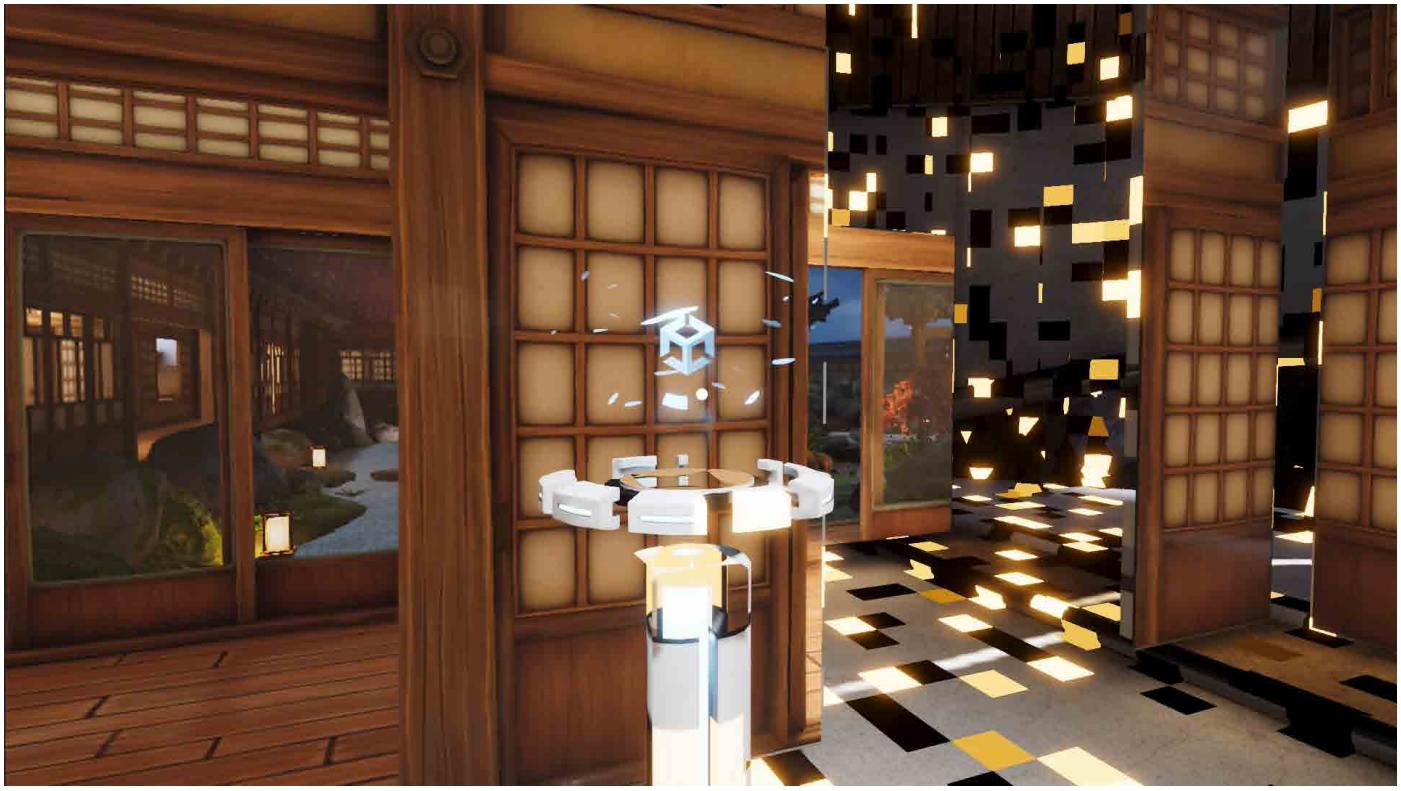
Cockpit

このシーンは、Shader Graph を使用したカスタムライティングコードを使用しています。Meta Quest 2 のようなコードレスの VR デバイス用に設計されています。

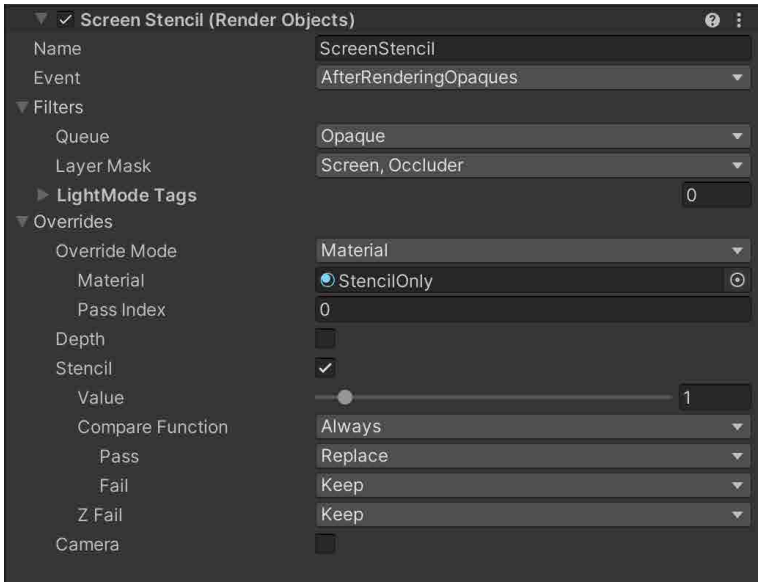
Terminal

このシーンは、他のサンプルシーン間のリンクとして機能し、次のシーンに移動するためのトランジションエフェクトを提供します。

シーン間の移動



サンプルプロジェクトでは、シーン間の移動にトランジションエフェクトを使用しています。トランジションエフェクトは、画面外のレンダーターゲットを使用して、トランジションが完了する前に受信するシーンをレンダリングします。入力シーンは、Shader Graph で作成されたカスタムシェーダーを使用して、出力されるシーンに配置された大型モニターにレンダリングされ、フルスクリーンスワップは、Render Objects Renderer Feature を介したステンシルを使用して処理されます。



Screen Stencil Renderer Feature

この効果を実際に見るには、台座に向かって Unity のロゴが表示されるまで歩き、ロゴが画面の中央に来るようにします。これでトランジションがトリガーされます。

すべてのシーンアセットはロード時にロードされますが、有効になるのは 1 つのシーンのみです。ターミナルシーンから開始した場合に、ランタイムで使用されるカメラは、FPS_Controller ゲームオブジェクトのものと同じです。**MainCamera** はアクティブなシーンをレンダリングし、**ScreenCamera** はモニターに表示されるシーンをレンダリングします。

ターミナルシーンの FPS_Controller

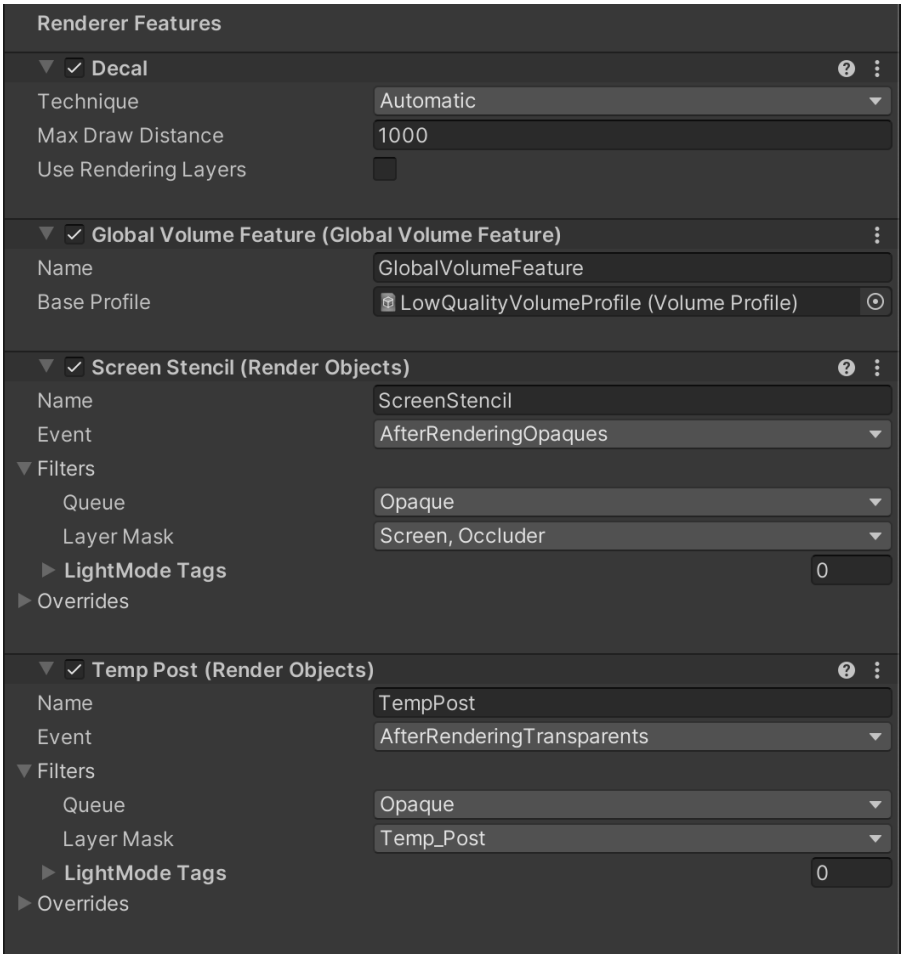
トランジション中、入力シーンカメラはレンダーターゲットにレンダリングされます。URP は 1 つのメインディレクショナルライトしかサポートしないため、問題が発生する可能性があります。**「Scripts > SceneManagement > SceneTransitionManager.cs」**というスクリプトがレンダリング前に実行され、アクティブなシーンのメインライトを有効にし、他のライトを無効にして、この制限を守るように設定されています。

以下のスクリプトをご覧ください。**OnBeginCameraRendering** メソッドでは、まずメインカメラをレンダリングしているかどうかをチェックします。**isMainCamera** が true の場合、**ToggleMainLight** が呼び出され、**currentScene** のメインディレクショナルライトが有効になり、入力シーンである **screenScene** のメインディレクショナルライトが無効になります。ただし、**isMainCamera** が false の場合はその逆になります。

同じスクリプトが、RenderSettings オブジェクトの設定を調整することによって、レンダリングされるシーンに合わせてフォグ、リフレクション、スカイボックスの切り替えを処理します。

```
179         /// <summary>
180         /// This function is called per camera by the render pipeline.
181         /// We use it to set up light and render settings (skybox etc) for the different scenes as they are displayed
182         /// </summary>
183         void OnBeginCameraRendering(ScriptableRenderContext context, Camera camera)
184         {
185             bool isMainCamera = camera.CompareTag("MainCamera");
186 
187             if (!isMainCamera && screenScene == null)
188             {
189                 //If no screen scene is loaded, no setup needs to be done for it
190                 return;
191             }
192 
193             //Toggle main light
194             ToggleMainLight(currentScene, isMainCamera);
195             ToggleMainLight(screenScene, !isMainCamera);
196 
197             //Setup render settings
198             SceneMetaData sceneToRender = isMainCamera ? currentScene : screenScene;
199             RenderSettings.fog = sceneToRender.FogEnabled;
200             RenderSettings.skybox = sceneToRender.skybox;
201             if (sceneToRender.reflection != null)
202             {
203                 RenderSettings.customReflectionTexture = sceneToRender.reflection;
204             }
205 
206             if (!isMainCamera && camera.cameraType == CameraType.Game)
207             {
208                 camera.GetComponent<OffsetCamera>().UpdateWithOffset();
209             }
210         }
211 
212         private void ToggleMainLight(SceneMetaData scene, bool value)
213         {
214             if (scene != null && scene.mainLight != null)
215             {
216                 scene.mainLight.SetActive(value);
217             }
218         }
```

Render Objects Renderer Feature を使用して、入力シーンと出力シーン間の遷移を処理します。ステンシルバッファに値を書き込むことで、次に続くパスでチェックできます。レンダリングされるピクセルが特定のステンシル値を持っている場合、色バッファに既にある値を保持し、そうでない場合は自由に上書きできます。**Renderer Features** は、パスの組み合わせを使用して最終的なレンダーを構築する非常に柔軟な方法です。



モバイルフォワード + レンダラーの Renderer Features

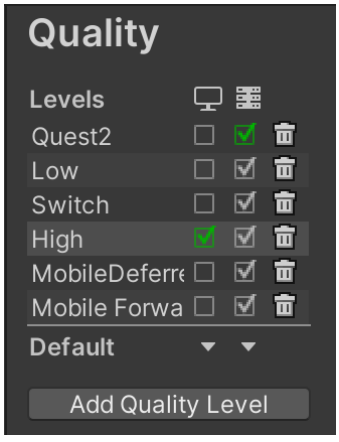
遷移中にカメラの位置を合わせるために、プロジェクトには、各シーンごとにオフセットトランスフォームを保存する **SceneMetaData** スクリプトがあります。また、遷移中の入出力シーンを処理する **SceneTransitionManager** スクリプトもあります。**Update** メソッドは遷移の進行状況を追跡します。**ElapsedTimeInTransition** が **m_TransitionTime** より大きくなると、**TriggerTeleport** が呼び出されることにより、**Teleport** メソッドが呼び出されます。これにより、プレイヤーの位置と向きが変更され、出力シーンから入力シーンへのシームレスな切り替えが行われます。

```
120     void Update()
121     {
122         float t = m_OverrideTransition ? m_ManualTransition : ElapsedTimeInTransition / m_TransitionTime;
123
124         if (InTransition)
125         {
126             ElapsedTimeInTransition += Time.deltaTime;
127
128             if (ElapsedTimeInTransition > m_TransitionTime)
129             {
130                 TriggerTeleport();
131             }
132
133             ElapsedTimeInTransition = Mathf.Min(m_TransitionTime, ElapsedTimeInTransition);
134         }
135         else
136         {
137             ElapsedTimeInTransition -= Time.deltaTime * 3;
138
139             if (ElapsedTimeInTransition < 0 && CoolingOff)
140             {
141                 CoolingOff = false;
142             }
143
144             ElapsedTimeInTransition = Mathf.Max(0, ElapsedTimeInTransition);
145         }
146
147         //Update weights of post processing volumes
148         if (m_Loader != null && !CoolingOff)
149         {
150             float tSquared = t * t;
151             m_Loader.SetVolumeWeights(1 - tSquared);
152         }
153
154         Shader.SetGlobalFloat(m_TransitionAmountShaderProperty, t);
155     }
```

ScreenTransitionManager.cs の Update メソッド

スケーラビリティ

URP は様々なハードウェアをサポートしており、新しいサンプルシーンでは異なるデバイスとの連携方法を示しています。**「Project Settings」>「Quality」**には、異なるオプションがあります。



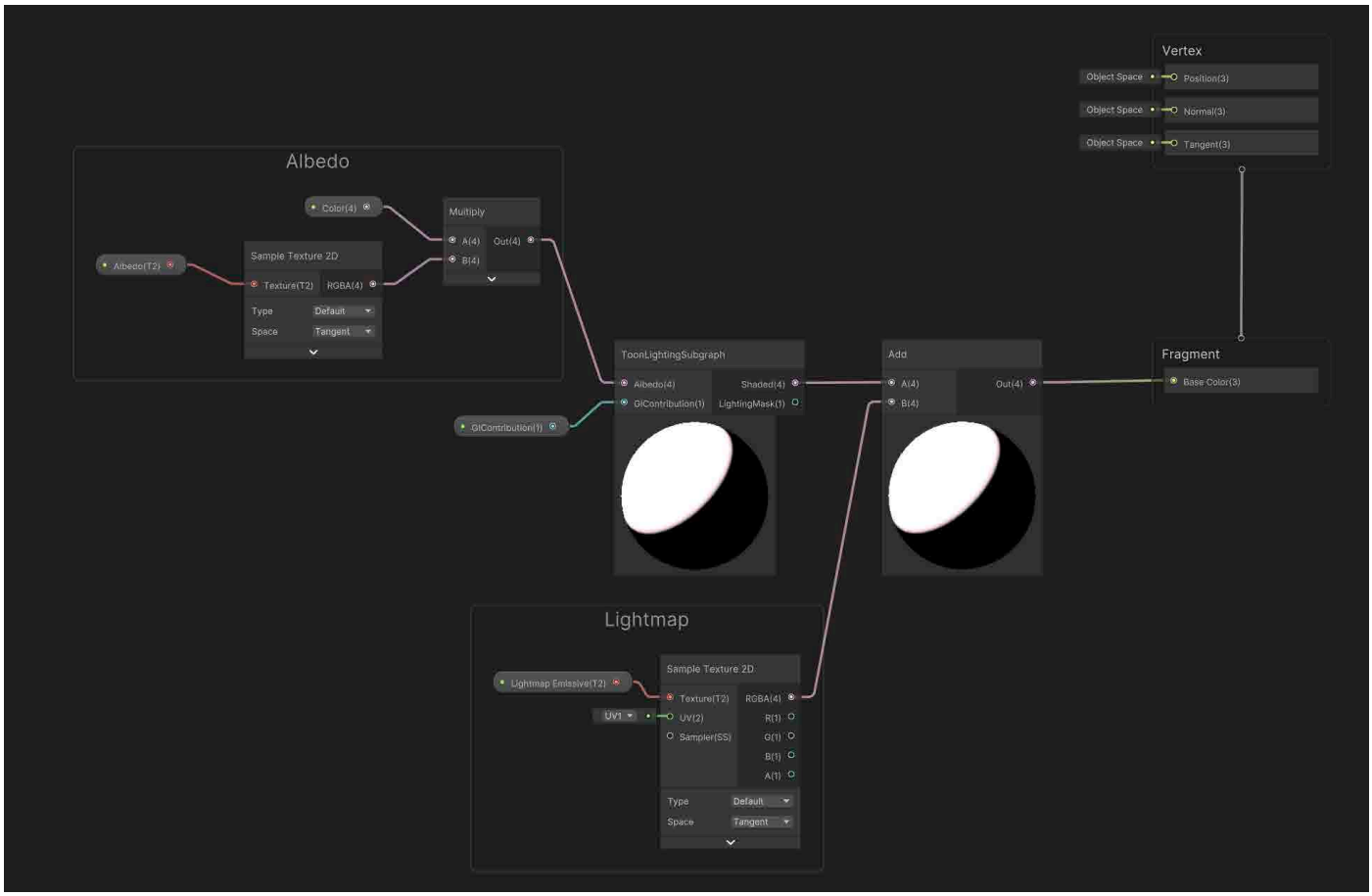
品質レベル

各オプションは異なる**レンダーパイプラインアセット**を使用します。品質のセクションで説明したように、URP はこのパネルとレンダーパイプラインアセットの設定を組み合わせて品質を処理します。

スタンドアロン VR ヘッドセットでは、リアルタイム 3D グラフィックスを表示する際に大きな課題が生じます。高解像度のスクリーンを持っているため、各目は個別に処理されなければならず、それにより各レンダリングされたフレームが 2 倍の作業を必要とするのです。さらに、最低目標 fps が 72 であるため、1 秒あたりに必要なピクセルが多くなります。この課題を回避する 1 つの方法は、様式化されたライティングを使用することです。下のコックピットのシーンでは、トゥーンシェーディングライティングモデルを使用しています。

URP 3D Sample のコックピットの環境

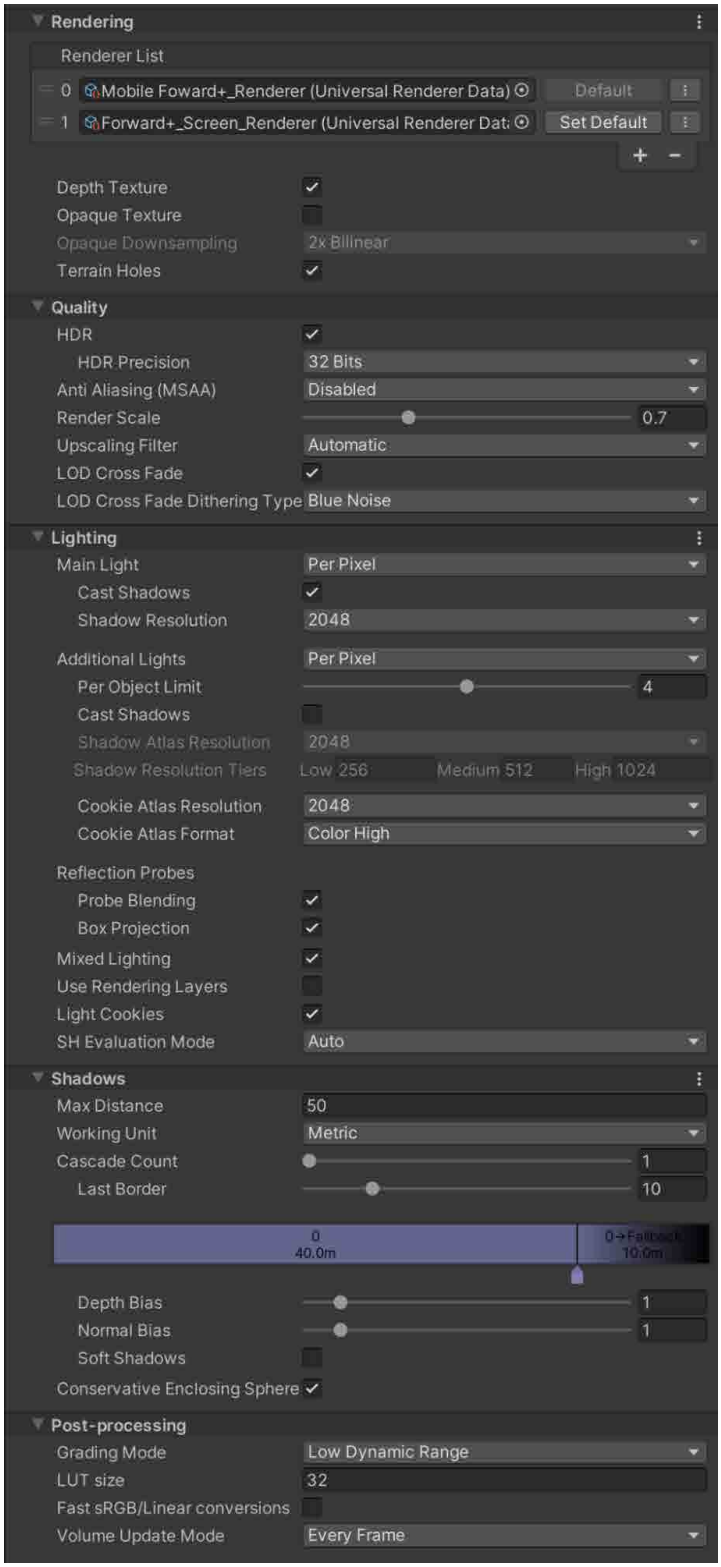
カスタムライティングは Shader Graph を使用しており、コーディングは最小限です。



通常のトゥーンシェーダーと同様に、法線ベクトルとメインライトの方向をドット積で結合し、ライティングレベルを決定します。その後、ランプを使用して、値を滑らかに変化させるのではなく、段階的にライトレベルを設定します。コックピットシーンで使用されているライティングモデルは、ベイクされたグローバルイルミネーションを使用して計算を実行し、微妙なアウトライン効果を追加するためにエッジ検出も行います。カスタムライティングは Shader Graph を使って処理されます。

トゥーンシェーダーの作成に関するチュートリアルについては、「*The Universal Render Pipeline cookbook*」を参照してください。

モバイルデバイスでのサンプルプロジェクトの実行



モバイルフォワード + URP アセット

ゲーム開発者にとって一般的な課題は、モバイルデバイスでゲームをスムーズに動作させることです。新しいサンプルプロジェクトの **Settings** フォルダーには、**モバイルフォワード +** URP アセットが含まれています。URP アセットは、品質設定を調整するための主な方法です。フォワード + は、フレームごとに大幅なカリング処理を行う CPU に依存しているため、ローエンドのモバイルデバイスには必ずしも最適な選択肢ではありません。このようなデバイスに最適なのは、サンプルプロジェクトの URP アセットで使用されているディファードレンダラーです。

左の画像は、モバイルフォワード + アセットの設定を示しています。

レンダラーリストには、2 つのユニバーサルレンダラーデータアセットがあります。1 つはアクティブシーン用の **Mobile Forward+_Renderer**、もう 1 つは画面シーンをレンダリングするための **Forward+_Screen_Renderer** です。深度テクスチャが有効になっています。追加ライトは影を投影しないことに注意してください。これは非常に負荷の高いオプションで、通常モバイルデバイスではライトクッキーを使って模倣可能です。特に庭のシーンにはたくさんのライトがあり、多くのライトが影を示すためにクッキーを使用しています。以下の画像の左下にある岩のライティングを、クッキーの有無で比較してください。

クッキーの有無による庭のシーンのポイントライト

モバイルプラットフォームをターゲットにする場合に特に役立つ 3 つのヒントをお伝えします。

- レンダリングするピクセル数を減らします。最近のモバイルのほとんどは、高い DPI（ドットパーインチ）を備えています。大半のゲームの場合、DPI は 96 あれば十分です。例えば、Screen.DPI が 300 の場合、2400 × 1200 の画面で 96/300 のレンダースケールを使用すると、768 x 384 ピクセルをレンダリングすることになり、ピクセル数がほぼ 10 分の 1 になるため、パフォーマンスが大幅に向上します。URP アセットでレンダースケールを設定するか、ランタイム時に値を調整することができます。
- MobileForward+_Renderer アセットには、Technique オプションが「Automatic」に設定された Decal Renderer Feature があることに注意してください。これは、非表示サーフェス除去を備えた GPU では、スクリーンスペースに切り替わりますこれにより、これらのデバイスでの無駄なリソース消費となる、深度プリパスを回避することでパフォーマンスが向上します。
- フォワード + の CPU オーバーヘッドが高すぎるデバイスでは、ディファードレンダリングを使用します。
- レンダーグラフシステムによる、より積極的なレンダーパスマージを有効にするには、以下を検討してください。
    - プロジェクトで必要ない場合は、URP アセット設定で Depth Texture と Opaque Texture の設定を無効にしてください。
    - Opaque Texture を使用する場合、Opaque Downsampling を「None」に設定してください。不透明なテクスチャをダウンサンプリングすると、URP の中間テクスチャに解像度の変化が生じ、パスをマージできなくなります。

— Depth Texture を使用する場合、Depth Texture Mode を「After Transparents」に設定してください。そうすることで、メインのレンダーパス（Opaque、Sky、Transparents）の間に CopyDepth パスが注入されるのを避け、レンダーグラフがこれらのパスを正常にマージできるようになります。

URP アセット設定やドキュメントと合わせて、これら 4 つのシーンを注意深く研究することで、紹介されているテクニックを自分のプロジェクトで使用する方法を学ぶことができます。

まとめ

URP への切り替えを検討している開発者やアーティストは、フルバージョンの Unity ドキュメント、Unity Learn、Unity Blog、Discussions を必ず確認してください。

Unity の Product Board では、現在開発中の URP 機能の概要に加え、次のリリースについて知ることができます。また、機能リクエストを追加することもできます。

皆様の開発の成功をお祈り申し上げます。


Unity®