Unity®

UNITY 2022 LTS EDITION → E ブック

基礎編

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）

目次

入門

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に、ビルトインレンダーパイプラインからユニバーサルレンダーパイプライン（URP）にプロジェクトを移行するための支援を提供することを目的としています。取り上げるトピックには以下が含まれます。

— 新規プロジェクト用に URP を設定する方法、または既存のビルトインレンダーパイプラインベースプロジェクトを URP に変換する方法

— ビルトインレンダーパイプラインベースのライト、シェーダー、および特殊効果を URP 用に更新する方法

— 2 つのレンダーパイプライン間でのコールバックの違い、URP でのパフォーマンス最適化などについて理解する方法

Unity がサポートするプラットフォームの増加に伴い、ビルトインレンダーパイプラインの制限も明白になってきました。ビルトインレンダーパイプラインアーキテクチャは、プラットフォームや API が増えるたびに、変更や保守が複雑になります。

2018 年、Unity は 2 つの新しいスクリプタブルレンダーパイプライン（SRP）、HD レンダーパイプライン（HDRP）と URP をリリースしました。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 3D サンプルは Unity Hub で入手可能

SRP アーキテクチャの目的は、高い柔軟性とカスタマイズ性、現在サポートされているまたは将来的にサポートされるプラットフォーム全体でのパフォーマンス向上、創造性を解き放つための迅速なイテレーションの提供です。

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。

数年にわたる集中開発の結果、URP テクノロジーは今や確固たるものとなり、本番環境で使用可能なものとなりました。このガイドは、そのメリットを活用し、ゲーム開発の成功を手助けするためのものです。

著者と貢献者

この e ブックの著者である Nik Lever 氏は、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 年からは 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で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

Felipe Lira は、グラフィックスおよび URP 担当のシニアマネージャーです。ゲーム業界のソフトウェアエンジニアとして 13 年以上の経験を持つ Felipe は、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、Unity ランタイムおよびエディターのグラフィックス製品管理リードを務めています。Ali はゲーム業界で 20 年間働いた経験を持っており、Halfbrick Studios による『Fruit Ninja』や『Jetpack Joyride』などのヒットタイトルに貢献しています。

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 15 年以上の経験を持っています。

Unity URP エンジニアリングおよびサンプルプロジェクトチームからも、重要な貢献がありました。

URP で作成されたシーン

レンダリングの進化：ビルトインレンダーパイプラインから SRP へ

Unity の最大の長所の 1 つは、広範なプラットフォームで使用できることです。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ターンキーソリューションとして開発されました。これは、グラフィックス機能の組み合わせをサポートし、フォワードパイプラインとディファードパイプラインで使用する際に便利です。

しかし、Unity がより多くのプラットフォームのサポートを追加し続ける中で、私たちはビルトインレンダーパイプラインに関連した以下の欠点を認識し始めました。

— コードの大部分が C++ で書かれており、開発者が変更することができない、ブラックボックスシステムである。

— 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動的なステートインジェクションの変更を行えるようになります。

— ユーザーインジェクション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、以下の機能によっ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。

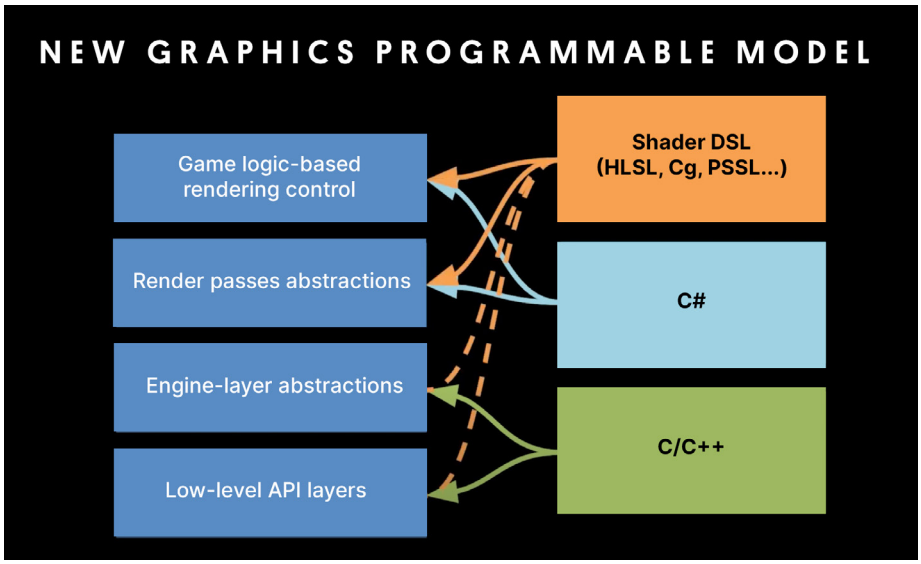
— ハイエンドデバイスからローエンドデバイスまで、最大数のハードウェアプラットフォームに対応するインテリジェントで信頼性の高いスケーリング。

— 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# を使用すれば、変更のたびに新しい実行ファイルをコンパイルする必要がありません。

— 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優れたパフォーマンスを発揮する、シャープなグラフィックスを作成できる柔軟性。

以下の画像は、SRP の仕組みを示しています。アーティストにとって使いレンダーパスやすい Shader Graph などのツールで作成可能な HLSL シェーダーを使用するだけでなく、C# を使用してレンダーパスとレンダリングコントロールを制御、カスタマイズします。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あり、GitHub で入手可能です。

URP を選択する理由

— **幅広いユーザーがアクセス可能：**URP は、アーティストとテクニカルアーティストの両者が設定可能で、プロトタイピングの柔軟性を高めるとともに、完全なゲーム制作のためのレンダリング技術を洗練させるものです。

— **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、パイプラインを新しい機能で拡張することを可能にし、アセットストアやサードパーティパッケージクリエイターや経験豊富なスタジオ、上級者チームなどの上級ユーザーにとっての確かな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で呼び出されるシンプルな C# スクリプトを書く

ことができ、パフォーマンスを犠牲にすることなく高レベルでのカスタマイズを可能にします。

- **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+、ディファードレンダリングパスをサポートするユニバーサルレンダラーと、2D レンダラーを提供します。

  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様々なイベントで、所定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。また、これらのオブジェクトのレンダリング時に、マテリアルやその他のレンダリング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。

- **パフォーマンスの向上：**URP は、多くの場合、同等の品質設定に対して、ビルトインレンダーパイプラインと同等以上のパフォーマンスを提供します。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より効率的に評価します。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単一のレンダーパスにまとめることができます。

  - メッシュ描画時の CPU および GPU 効率が改善されました。これは SRP Batcher によるもので、ドローコールが減り、深度の処理方法が改善されるためです。

  - URP は、モバイルデバイス上でタイルメモリをより効率的に使用し、消費電力の削減、バッテリー寿命の延長、ひいては長時間のプレイセッションの可能性につながります。

  - URP には、ビルトインレンダー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。

- **最新ツールとの互換性：**URP は、Shader Graph、VFX Graph、レンダリングデバッガーなど、アーティストにとって使いやすい最新のツールをサポートしています。

現在は URP または HDRP で構築されているプロジェクトが大半を占めていますが、ビルトインレンダーパイプラインは、Unity 2022 LTS と Unity 6 でも利用可能なオプションとなっています。

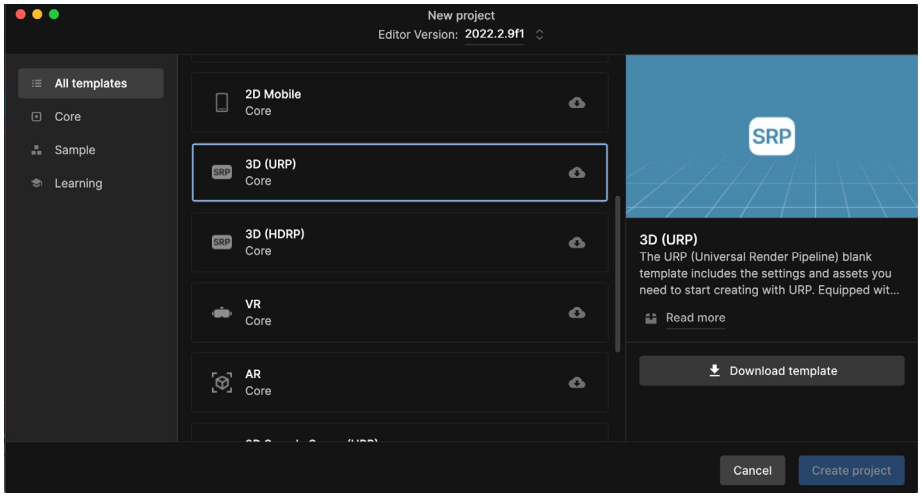
ビルトインレンダーパイプラインと URP の機能の包括的な比較については、このリンクを参照してください。

変換プロセス

このセクションでは、URP で新しいプロジェクトを開始したり、既存のプロジェクトを URP に変換する手順を説明します。

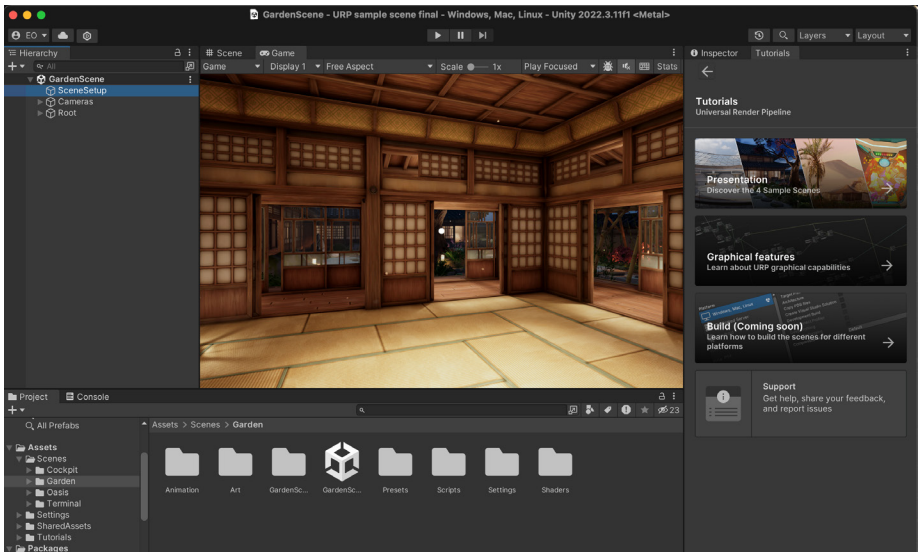
新しい URP プロジェクトを開く方法

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 2022.2 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、「**3D (URP)**」テンプレートまたは「**3D サンプルシーン (URP)**」を選択して、「**Create**」をクリックします。



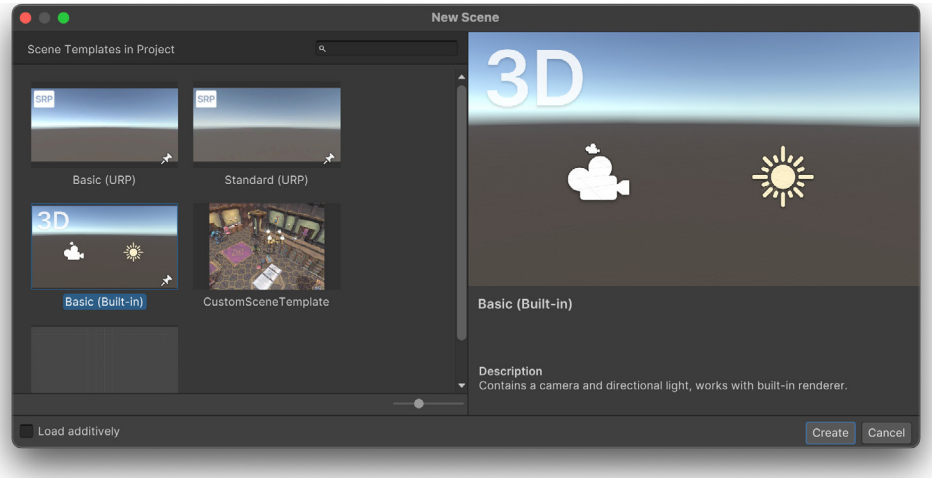
URP テンプレートを使って新規プロジェクトを作成します。最初にテンプレートをダウンロードする必要がある場合があります

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。



Unity Hub で入手可能な、URP 3D サンプルに含まれる 4 つの環境のうちの 1 つ

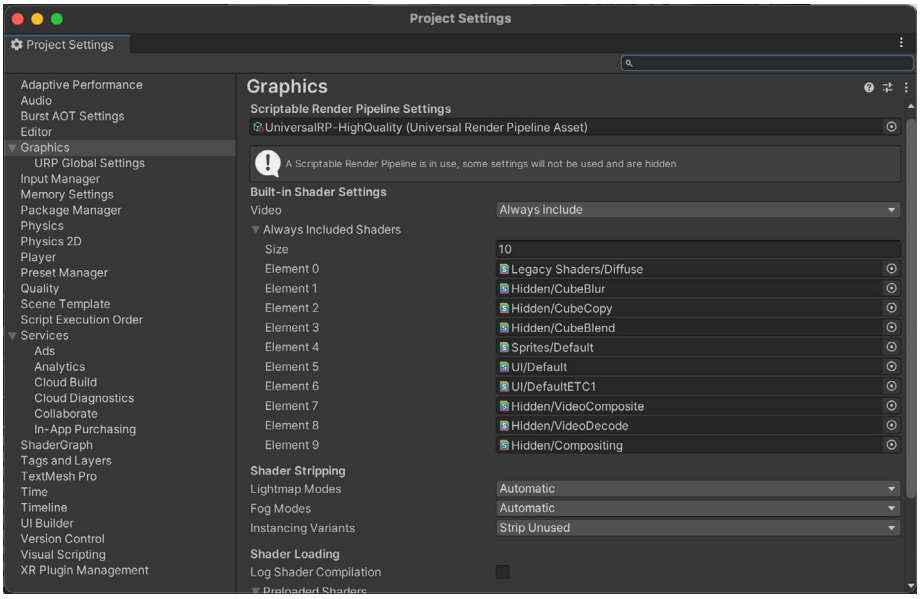
カメラやディレクショナルライトなどの基本ゲームオブジェクトを使用して、「**File**」>「**New Scene**」で新しいシーンを作成したり、あらかじめオブジェクトが配置された独自のシーンテンプレートを作成したりすることもできます。詳しくは URP Scene Templates ドキュメントをご覧ください。



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。URP をエディター内で使用するには、「**Scriptable Render Pipeline Settings**」から「URP Asset」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。

「**UniversalRP-HighFidelity**」がデフォルト設定の URP アセットとなっていますが、「**UniversalRP-Balanced**」または「**UniversalRP-Performant**」に切り替えることもできます。



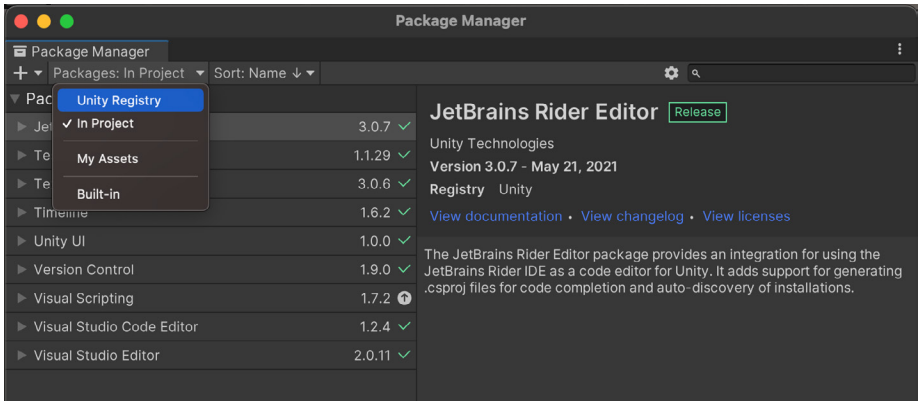
「Project Settings」の「Graphics」パネル

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

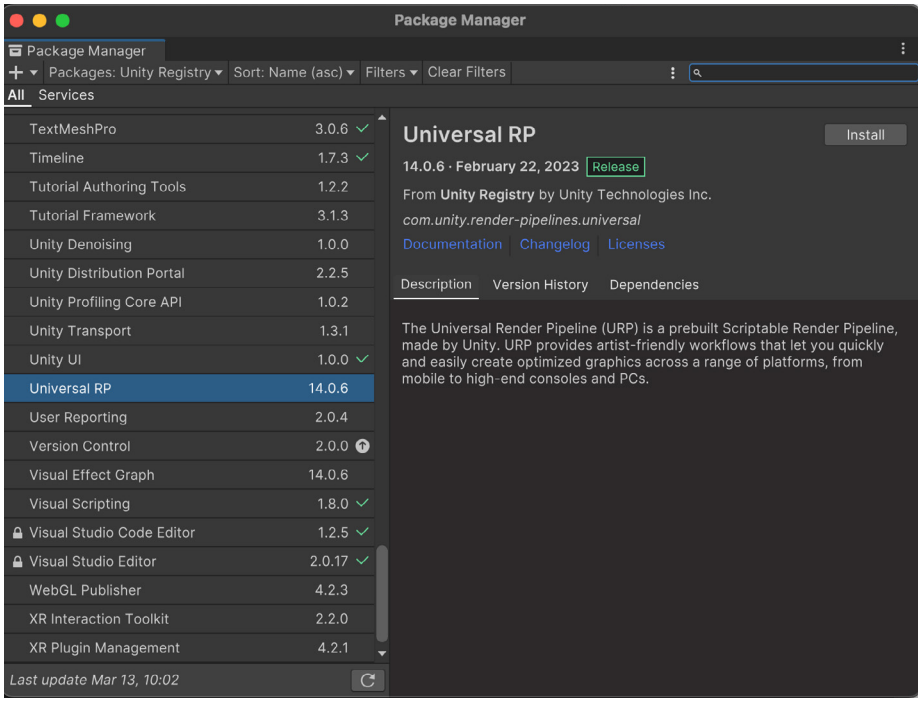
重要: このセクションの手順を実行する前に、ソースコントロールを使用してプロジェクトのバックアップを取るようにしましょう。このプロセスではアセットの変換を行いますが、Unity ではこの変更に対する取り消しオプションは提供されていません。ソースコントロールを使用している場合、必要に応じてアセットの変更前のバージョンに戻すことができます。

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、**URP パッケージ**は Unity 2022.2 や 2022 LTS には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。



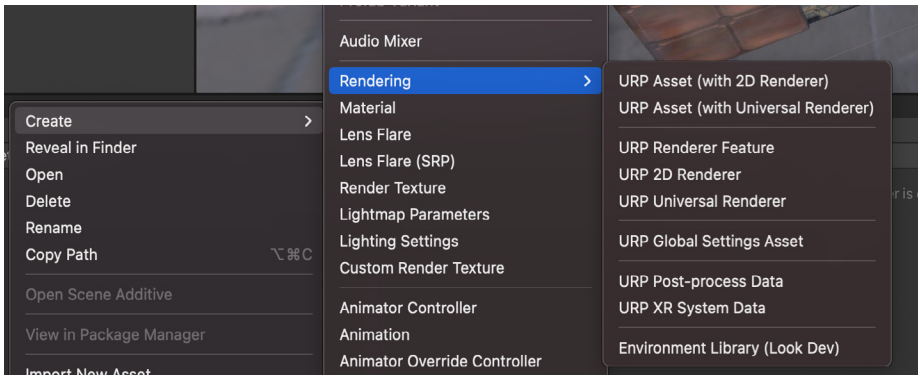
Package Manager 内に Unity Registry パッケージが表示されている

「Window」>「Package Manager」に移動し、**「Packages」**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。まだ 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

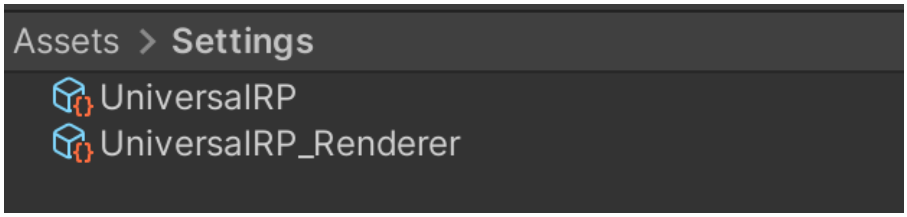
URP アセットを作成するには、「**Project**」ウィンドウを右クリックし、「**Create**」>「**Rendering**」>「**URP Asset (with Universal Renderer)**」を選択します。アセットに名前を付けます。



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すると、URP アセットや URP パッケージがプロジェクト内で使用可能な状態になります。

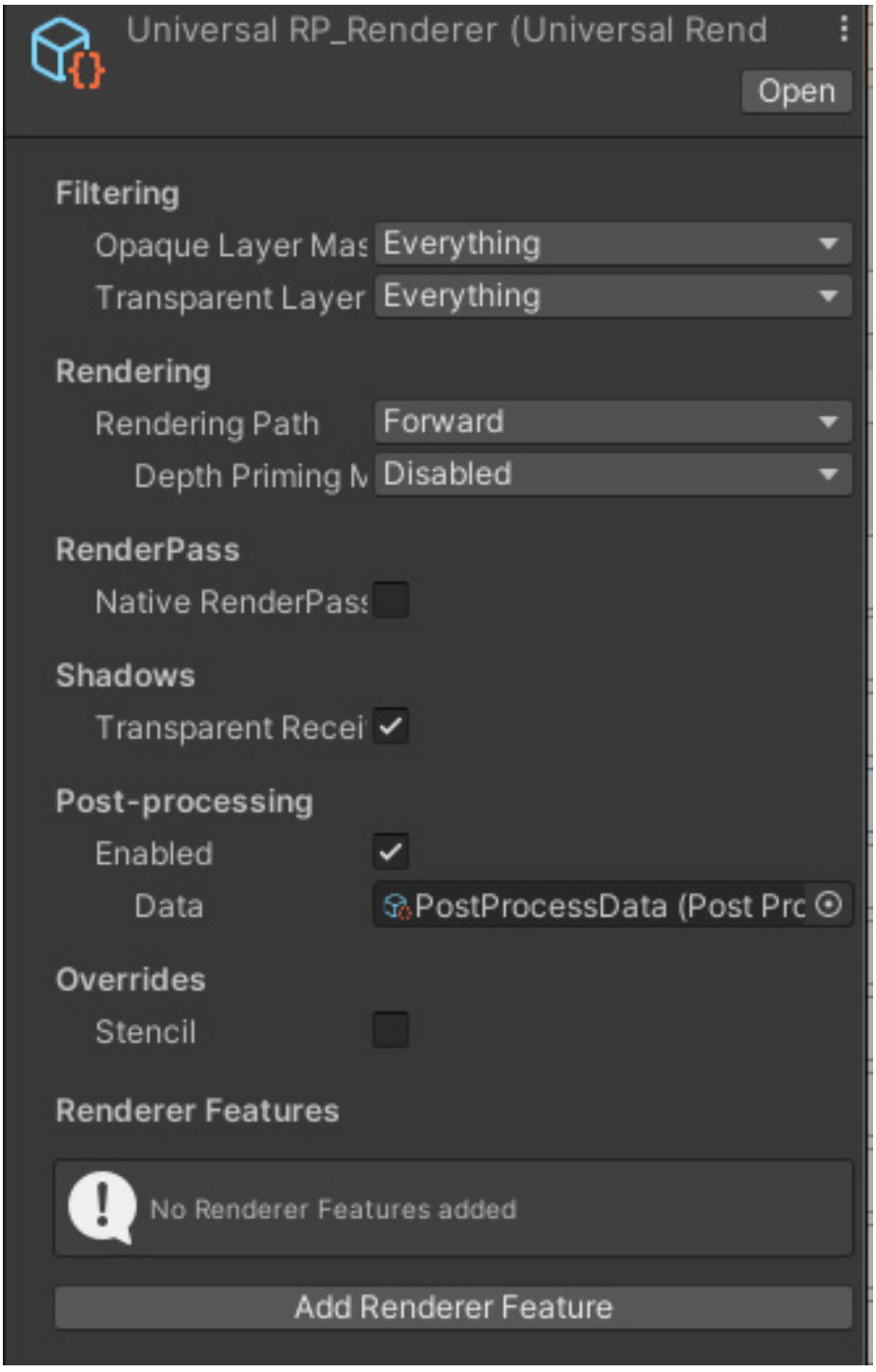
URP は、単一の URP アセットを作成するのではなく、アセット拡張子を持つ 2 つのファイルを使用します。



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、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る**レンダラーデータアセット**です。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。

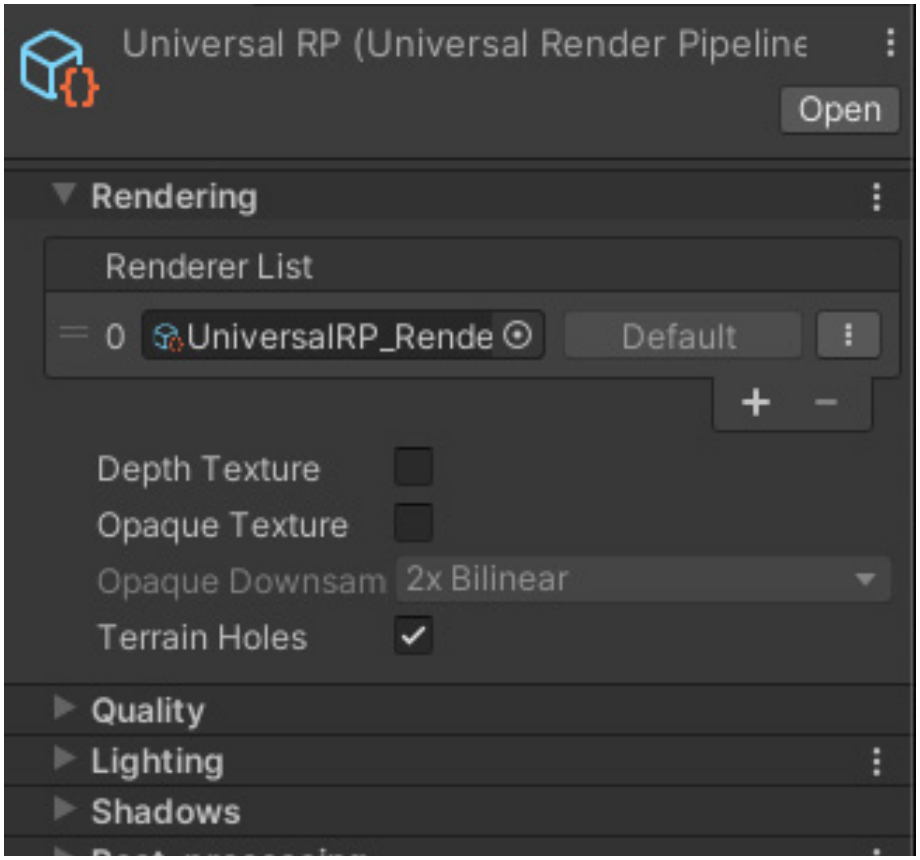
さらに、UniversalRP_Renderer は、URP のハイレベルなレンダーロジックとパスを制御します。フォワードパスとディファードパス、そして 2D ライト、2D シャドウ、ライトブレンドスタイルなどの機能を使用可能にする 2D レンダラーをサポートしています。URP を拡張して、独自のレンダラーを作成することもできます。



UniversalRP_Renderer データアセットの Inspector

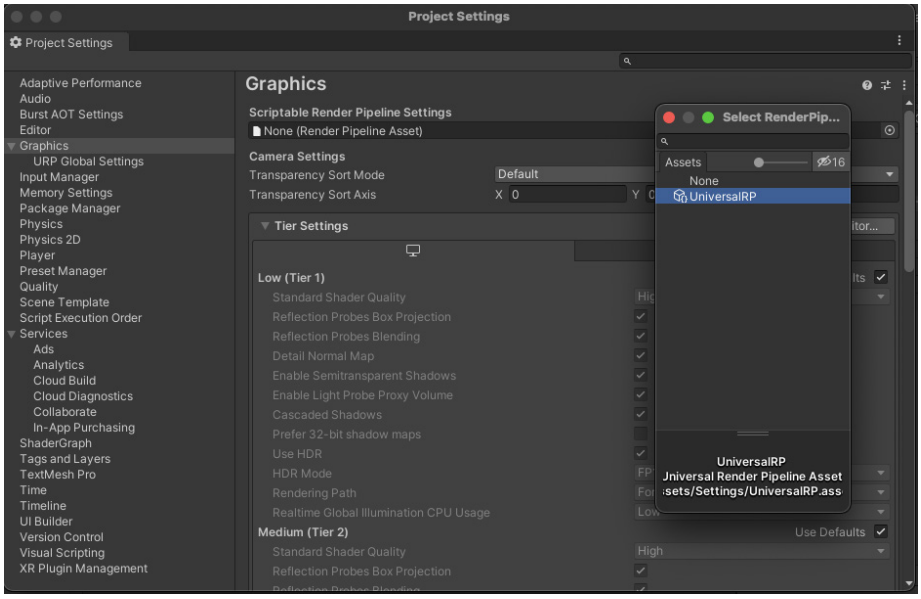
もう 1 つの URP アセットは、品質、ライティング、影、およびポストプロセスの設定を制御する役割を果たします。このセクションの後の方でプロセスを説明していますが、異なる URP アセットを使用して品質設定を制御できます。この設定アセットは、レンダラーリストを介してレンダラーデータアセットにリンクされています。新しい URP アセットを作成すると、設定アセット内に、1 つのアイテム（同時に作成されるレンダラーデータアセットで、デフォルトとして設定される）を含むレンダラーリストが作成されます。このリストには、代わりのレンダラーデータアセットを追加することができます。

デフォルトのレンダラーは、Scene 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ぶことで、デフォルトレンダラーをオーバーライドできます。これは必要に応じてスクリプトを介して実行できます。



Inspector 内の URP アセット

これらのステップに従って URP アセットを作成しても、Scene ビューまたは Game ビューで開いているシーンは、ビルトインレンダーパイプラインを使用します。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。



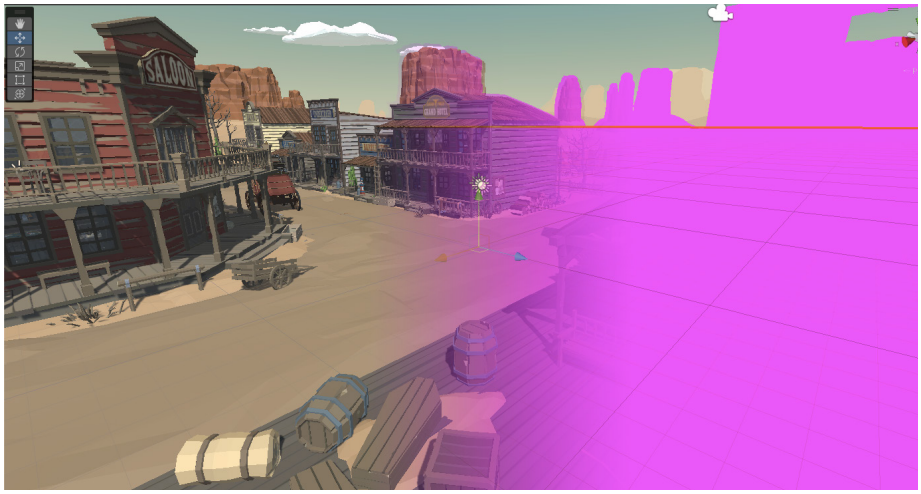
スクリプタブルレンダーパイプラインアセットを選択する

切り替えについての警告メッセージが表示されますが、「**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、作成した美しいシーンがすべ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によって使用されるシェーダーが、URP でサポートされていないことによるものです。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ー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 コンバーター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、このコンバーターは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 コンバーターを使用しても変換されません。シェーダーと新しいツールのセクションでは、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、使用すればほとんどの現実世界のマテリアルを表現できます。メタリックおよびスペキュラーの両方のワークフローで、すべてのスタンダードシェーダー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/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 パッケージととも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。

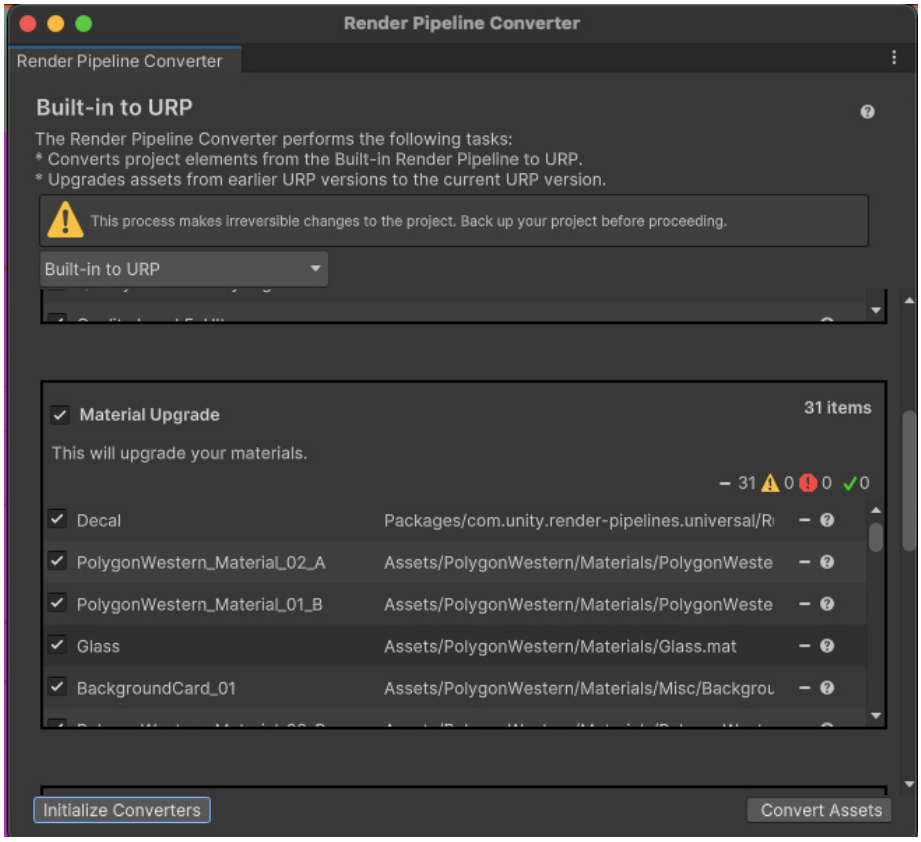
— **Universal Render Pipeline/Particles/Unlit:** この Unlit シェーダーは、GPU 負荷が少ないシェーダーです。

Simple Lit は多くのレガシー/モバイルシェーダーを置き換えますが、性能は同じではありません。レガシー/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ない一方、

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 ドキュメントのこの表で、各 URP シェーダーがどのビルトインレンダーパイプラインシェーダーに対応するかを確認できます。

上記のコンバーターを 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た場合でも、プロジェクトはスキャンされ、変換の必要があるアセットが各コンバーター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、コンバーターの初期化後、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ー

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、品質のデフォルトオプションがいくつかあります。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、Scene ビューと Game 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できます。

レンダーパイプラインコンバーターを使用しているときに「**Rendering Settings**」のオプションを選択して、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対応するような URP アセットのセットが作成されます。下の 1 番目の表は、ビルトインレンダーパイプラインと URP の低い設定との対応を、2 番目の表は高い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」パネルから選択します。URP アセット設定は、URP アセットを選択すると、Inspector でアクセス可能です。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から URP へ：低い設定

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）*	なし
Anti-aliasing	Disabled	なし	Disabled
Render Scale	なし	なし	1
Real-time Reflection Probes	No	No	
Resolution Scaling Fixed DPI Factor	1	1	なし
VSync Count	Don’t sync	Don’t sync	
Depth Texture	なし	なし	No
Opaque Texture	なし	なし	No
Opaque Downsampling	なし	なし	なし
Terrain Holes	なし	なし	Yes
HDR	なし	なし	Yes
Textures			
Texture Quality	Half res	Half res	なし
Anisotropic Textures	Disabled	Disabled	なし
Texture Streaming	No	No	なし
Particles			
Soft Particles	No	なし	なし
Particle Raycast Budget	16	16	なし
Terrain			
Billboards Face Camera Position	No	No	なし

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
Shadows			
Shadowmask Mode	Shadowmask	Shadowmask	なし
Shadows	Disabled	なし	なし
Shadow Resolution	Low resolution	なし	なし
Shadow Projection	Stable fit	なし	なし
Shadow Distance	20	なし	なし
Shadow Near Plane Offset	3	なし	なし
Shadow Cascades	No Cascades	なし	なし
Cascade splits	なし	なし	なし
Working unit	なし	なし	なし
Depth Bias	なし	なし	なし
Normal Bias	なし	なし	なし
Soft Shadows	なし	なし	なし
Async Asset Upload			
Time Slice	2	2	なし
Buffer Size	16	16	なし
Persistent Buffer	Yes	Yes	なし
Level of Detail			
LOD Bias	0.4	0.4	なし
Maximum LOD level	0	0	なし
Meshes			
Skin Weights	4 bones	4 bones	なし
Lighting			
Main Light:	なし	なし	Per pixel
• Cast Shadows	なし	なし	No
• Shadow Resolution	なし	なし	なし
Additional Lights:	なし	なし	Disabled
• Per Object Limit	なし	なし	なし
• Cast Shadows	なし	なし	なし
• Shadow Atlas Resolution	なし	なし	なし
• Shadow Resolution tiers	なし	なし	なし
• Cookie Atlas Resolution	なし	なし	なし
• Cookie Atlas Format	なし	なし	なし
Reflection Probes:	なし	なし	なし
• Probe Blending	なし	なし	No
• Box Projection	なし	なし	No
Post-processing			
Grading Mode	なし	なし	Low Dynamic Range
LUT size	なし	なし	16
Fast sRGB/Linear conversion	なし	なし	No

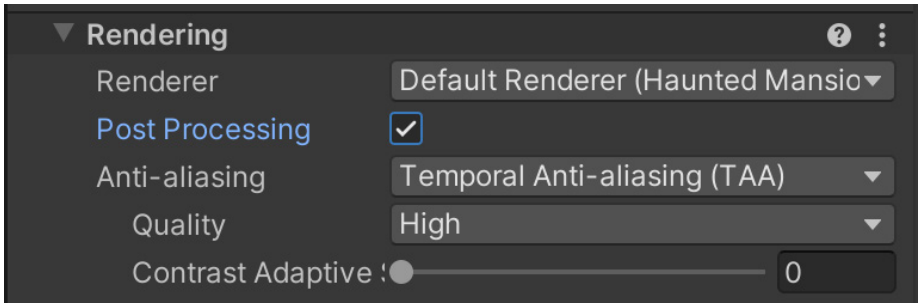
* URPでは、Pixel Light Count は**「Additional Lights」>「(Per pixel)」>「Per Object Limit」**を使用してハンドルされます。

ビルトインレンダーパイプラインから URP へ：高い設定

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）	なし
Anti-aliasing	Disabled	なし	2x
Render Scale	なし	なし	1
Real-time Reflection Probes	Yes	Yes	なし
Resolution Scaling Fixed DPI Factor	1	1	なし
VSync Count	Every V Blank	Every V Blank	なし
Depth Texture	なし	なし	No
Opaque Texture	なし	なし	No
Opaque Downsampling	なし	なし	なし
Terrain Holes	なし	なし	Yes
HDR	なし	なし	Yes
Textures			
Texture Quality	Full res	Full res	なし
Anisotropic Textures	Disabled	Disabled	なし
Texture Streaming	No	No	なし
Particles			
Soft Particles	No	なし	なし
Particle Raycast Budget	256	256	なし
Terrain			
Billboards Face Camera Position	Yes	Yes	なし
Shadows			
Shadowmask Mode	Distance Shadowmask	Distance Shadowmask	なし
シャドウ	Hard and Soft Shadows	なし	なし
Shadow Resolution	Medium resolution	なし	2048
Shadow Projection	Stable fit	なし	なし
Shadow Distance	40	なし	50
Shadow Near Plane Offset	3	なし	なし
Shadow Cascades	2 Cascades	なし	2
Cascade splits	33/67	なし	12.5/33.8/3.8
Working unit	Percent	Percent	Metric

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
Depth Bias	なし	なし	1
Normal Bias	なし	なし	1
Soft Shadows	なし	なし	Yes
Async Asset Upload			
Time Slice	2	2	なし
Buffer Size	16	16	なし
Persistent Buffer	Yes	Yes	なし
Level of Detail			
LOD Bias	1	1	なし
Maximum LOD level	0	0	なし
Meshes			
Skin Weights	Unlimited	Unlimited	なし
Lighting			
Main Light:	なし	なし	Per pixel
• Cast Shadows	なし	なし	Yes
• Shadow Resolution	なし	なし	
Additional Lights:	なし	なし	Per pixel
• Per Object Limit	なし	なし	4
• Cast Shadows	なし	なし	Yes
• Shadow Atlas Resolution	なし	なし	2048
• Shadow Resolution tiers	なし	なし	512/1024/2048
• Cookie Atlas Resolution	なし	なし	2048
• Cookie Atlas Format	なし	なし	Color high
Reflection Probes:	なし	なし	
• Probe Blending	なし	なし	Yes
• Box Projection	なし	なし	No
Post-processing			
Grading Mode	なし	なし	Low Dynamic Range
LUT size	なし	なし	32
Fast sRGB/Linear conversion	なし	なし	No

Unity 2022 LTS の URP で利用可能なオプションは、**「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)」を選択することです。



TAA が選択されている

品質設定の使用方法

品質設定は、以前は「Project Settings」ダイアログボックスの「Quality」パネルから行えました。URP を使用する場合、設定は「Quality」パネルと各 URP アセットに分かれています。次の表は、各設定がどこにあるかを示しています。

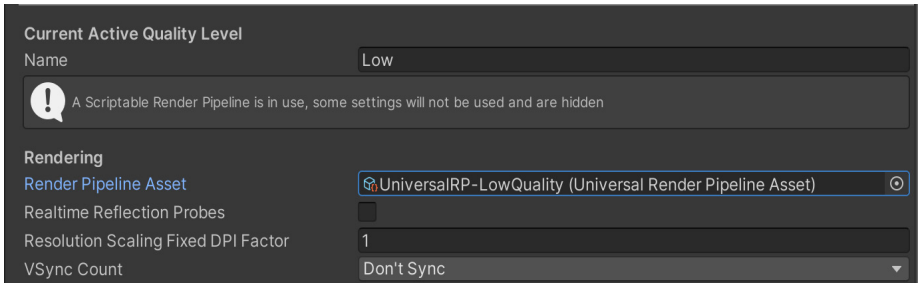
URP 使用時の品質設定

設定	「Quality」パネル	URP アセット
Rendering		
Anti-aliasing		√
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		√
設定	**「Quality」パネル**	**URP アセット**
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 Resolution tiers		√
• Cookie Atlas Resolution		√
• Cookie Atlas Format		√
Reflection Probes:	√	
• Probe Blending		√
• Box Projection		√

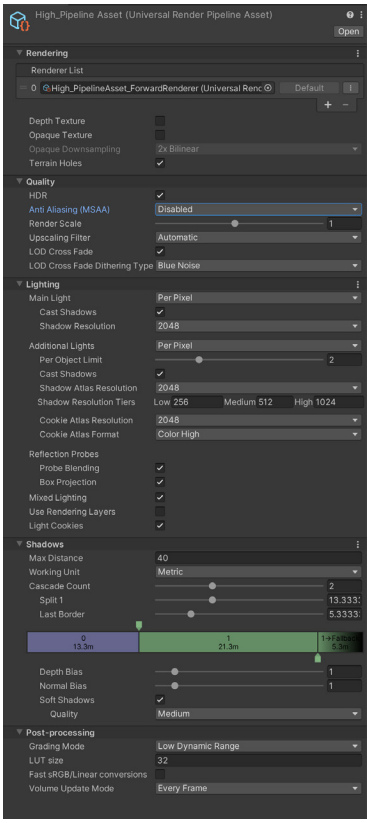
Post-processing		
Grading Mode		√
LUT size		√
Fast sRGB/Linear conversion		√

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。品質レベルを設定しない場合、レンダーパイプラインアセットは「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ものをデフォルトで使用することにご注意ください。このため、URP アセットの品質設定を調整する際に混乱を招くことがあります。例えば、URP アセットで設定されている品質レベルを、Scene ビューと Game ビューで現在使用されているものと同じだと思い込んでしまうかもしれません。



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



注意：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

この画像は、Inspector 内の URP アセットと、その利用可能なすべての設定を示しています。URP ドキュメントで、各設定の詳細を確認できます。

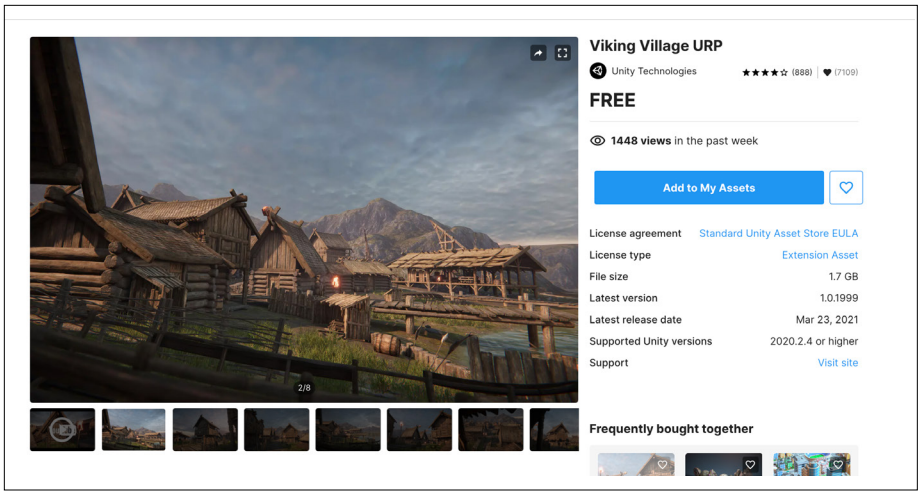
URP アセットの「Quality」パネルでは、忠実度を上げるため、HDR フォーマットを 64 ビットに設定することができます。ただし、パフォーマンスが低下し、追加のメモリも必要になるため、ローエンドのハードウェアではこの設定を避けてください。

「Quality」パネルのもう 1 つの特徴は、「**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 は、異なる LOD ジオメトリのスムーズな遷移を可能にし、スワップ中に発生する急激なスナップやポッピングを回避します。

サンプルプロジェクトをビルトインレンダーパイプラインから URP に変換

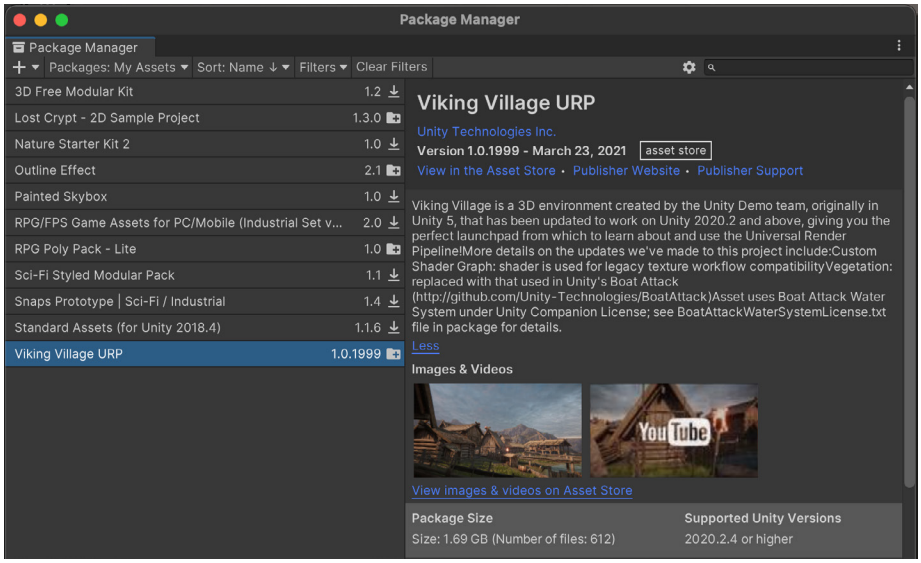
Unity のデモプロジェクト「Viking Village URP」には、ライトプローブ、リフレクションプローブ、カスタム ScriptableRenderPass を使用する水の特殊効果、Shader Graph 経由で変換されたシェーダー、および URP ポストプロセスなどの URP の機能が含まれています。このプロジェクトは、Unity Asset Store で無料で入手できます。

エディターで「Viking Village URP」を開き、このセクションの手順に従ってください。「**Add to My Assets**」をクリックし、エディターで利用可能なパッケージリストにこのデモを追加して開始します。



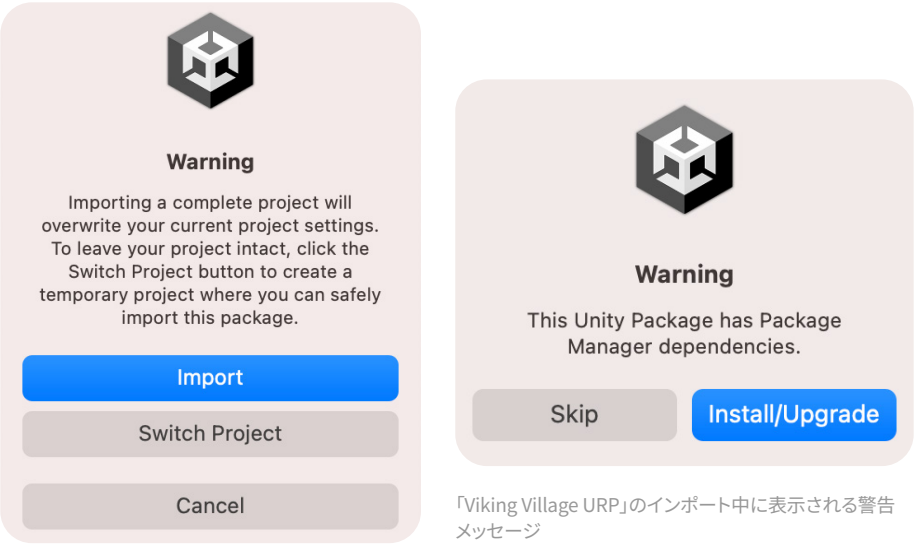
Unity Asset Store の「Viking Village URP」

次に Unity Hub から新しい 3D プロジェクトを作成します（URP テンプレートを使用する必要はありません）。「**Window**」>「**Package Manager**」に移動し、「**Packages**」ドロップダウンから「**My Assets**」>「**Viking Village URP**」を選択し、「**Import**」をクリックします。



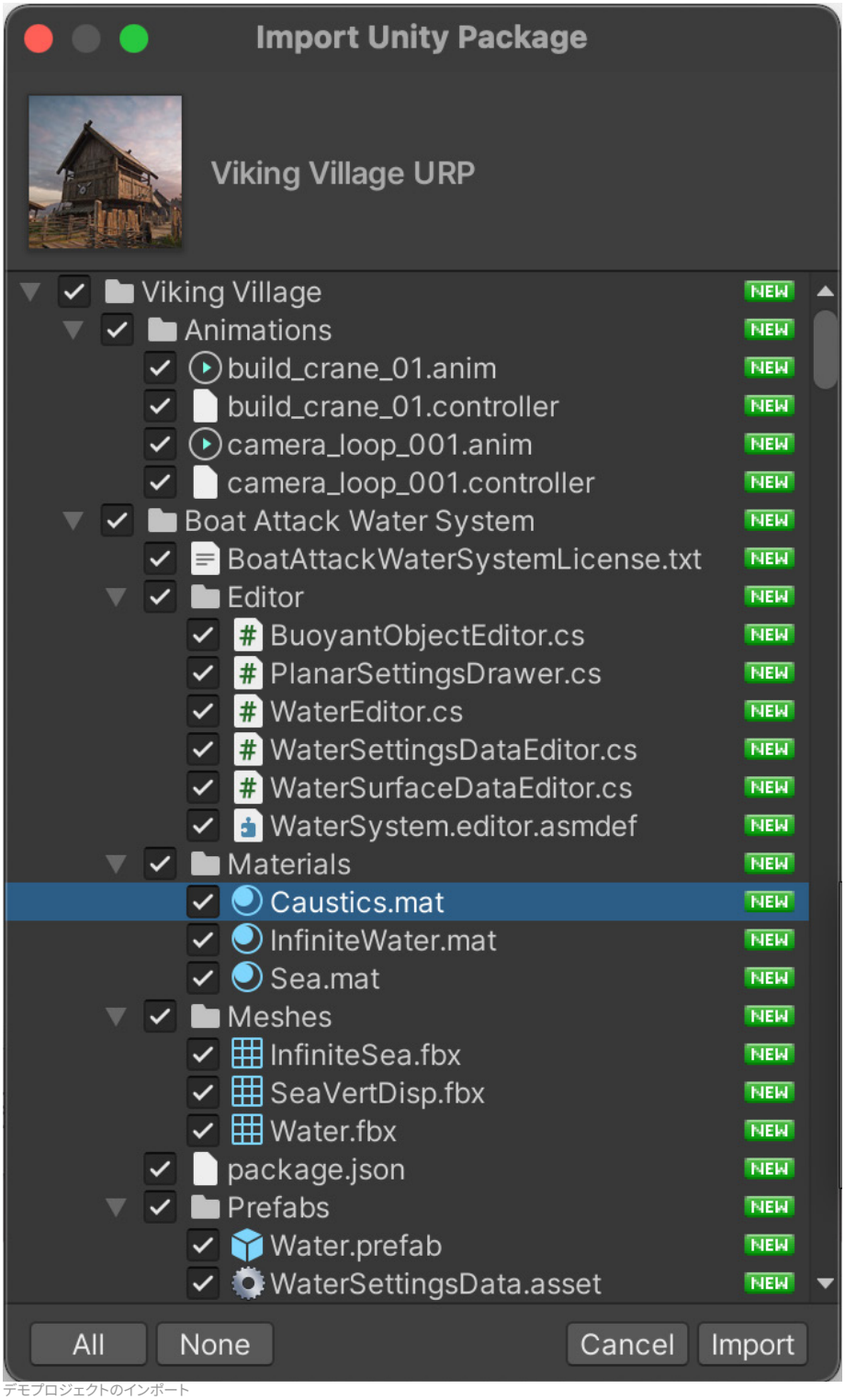
Package Manager 上に表示されている「Viking Village URP」

いくつかの警告メッセージが表示されます（以下参照）。最初の警告は、完全なプロジェクトをインポートすると、現在のプロジェクト設定が影響を受けることを警告するものです。ここでは空のプロジェクトを作成したため、続行して問題ありません。2 つ目の警告は、特定パッケージのインストールやアップグレードに関するものです。デフォルトの青いボタンをクリックしてください。URP のデフォルトはリニアカラースペース、ビルトインレンダーパイプラインのデフォルトはガンマカラースペースであるため、誤ったライティング設定を避けるために、この操作は必須です。



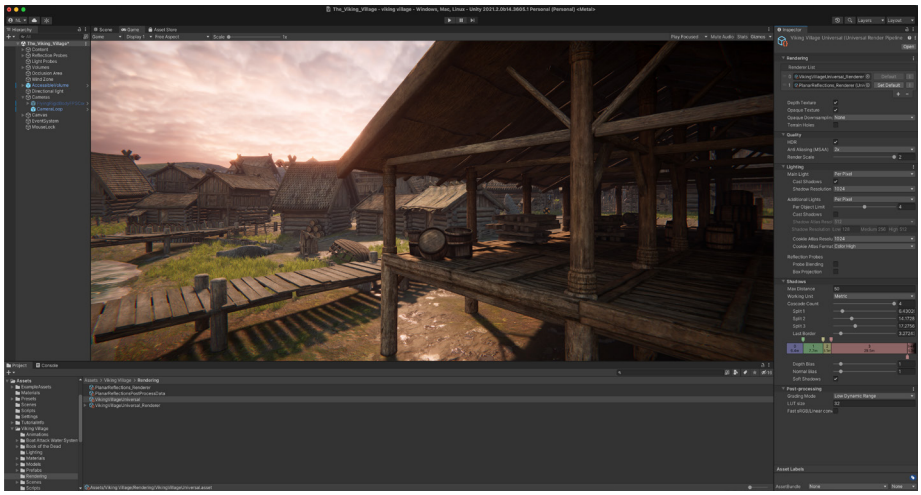
「Viking Village URP」のインポート中に表示される警告メッセージ

ダウンロードが完了すると、以下のパネルが開きます。すべて選択したままにして、「**Import**」をクリックしてください。



デモプロジェクトのインポート

すべてのアセットのインポートが完了するまで待ち、**「Viking Village」>「Scenes」>「The_Viking_Village」**に格納されているデモに移動します。**「Window」>「Package Manager」**をクリックし、ドロップダウンで**「Unity Registry」**を選択した後、**「Universal RP」**を選択します。URP パッケージを 14.x にアップデートします。



Game ビューで見る「Viking Village」

スクリプタブルレンダーパイプラインアセットを通じて「Graphics」パネルで設定された URP アセットは、**「Viking Village」>「Rendering」>「VikingVillageUniversal」**で名前を指定します。これはハイエンドハードウェア向けに設定されているため、古いハードウェアでは低いフレームレートで再生される可能性があります。

以下のステップに従って、様々な品質レベルを試してください。

1. 「Window」>「Rendering」>「Render Pipeline Converter」でアセットのセットを生成します。
2. 「**Built-in Render Pipeline to URP**」のオプションを選択し、「**Rendering Settings**」を選択します。
3. 「**Initialize Converters**」をクリックします。
4. パネル内に、多くの設定オプションが表示されます。「**Convert Assets**」をクリックして、URP アセットを作成します。
5. URP アセットは、**「Project Settings」>「Quality」**パネルを通じて、利用可能な品質レベルに割り当てられます。
6. 最高品質のアセットが、「Graphics」パネルで「VikingVillageUniversal」に置き換わります。**「Viking Village」>「Rendering」>「VikingVillageUniversal_Renderer」**は、Renderer Feature と水のエフェクトを使用しています。
7. これらを復元するには、上記のレンダラーを**レンダラーリスト**に追加し、品質レベルで使用される各 URP アセットのデフォルトに設定します。これにより、「Quality」パネルで品質レベルを素早く切り替えることができます。

Scene ビューで見る「Viking Village」

URP でのライティング

このセクションでは、URP のライティングの挙動と、2 つのレンダーパイプラインにおけるワークフローの違いについて説明します。

Unity のライティングに触れるのが初めての場合、これらのリソースを使用してください。

- ライティングに関するドキュメント
- ライティングとレンダリングの概要
- The art of lighting game environments
- Real-time lighting in Unity
- Harnessing light with the URP and the GPU Lightmapper

プロジェクトをビルトインレンダーパイプラインから URP に変換すると、ライティングの違いに気づくかもしれません。これは、ビルトインレンダーパイプラインがデフォルトでガンマライティングモデルを使用する一方、URP がリニアモデルを使用するためです。そのため、強度の値が 1.0 ではないライトは調整の必要があります。

また、エディター内の設定コントロールの場所や、ハードウェアのスペックが大きく異なる場合の対処方法にも違いがあります。このセクションの残りの部分では、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いくつかのテクニックについて説明します。

前回と同様、ここに挙げた 3 つの場所でプロパティを設定します。A と B は両方のレンダーパイプラインにおいて基本的に同じですが、C は URP のみに適用されます。

A. **「Window」>「Rendering」>「Lighting」：**このパネルでは、ライトマッピングと環境設定を設定し、リアルタイムのライトマップとベイクしたライトマップを表示できます。ビルトインレンダーパイプラインと URP に変換しても変わりません。

B. **Light Inspector：**ビルトインレンダーパイプラインと URP Inspector には大きな違いがあります。詳細については、Light Inspector のセクションをご確認ください。

C. **URP Asset Inspector：**これは、主に影の設定を行う場所です。URP でのライティングは、このパネルで選択した設定によって大きく影響を受けます。

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、これは 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あります。例えば、「Clear Coat」オプションを使って自動車にメタリックな光沢を持たせる場合には、これを選択します。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、様々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のデフォルトの選択肢です。ベイク、混合、およびリアルタイムのライティングをサポートしており、フォワードまたはディファードのレンダリングで機能します。 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、優れた外観をもたらすことが可能です。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をサポートする必要のないオブジェクトにパフォーマンスの向上をもたらします（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）。

ビルトインレンダーパイプラインと URP のライティングフォールオフと減衰の比較

ビルトインレンダーパイプラインと URP のもう 1 つの違いは、スポットライトとポイントライトに適用されるライトのフォールオフ/減衰の計算方法です。

URP は、このページで説明されているように、物理ベースの InverseSquared フォールオフを使用します。一方で、同じページで説明されていますが、ビルトインレンダーパイプラインは、物理ベースではない古いフォールオフを使用しています。ライトの半径はフォールオフに影響します。その結果、半径の大きなライトが生成され、必要以上に多くのオブジェクトに接触することで、カリングのパフォーマンスに影響を与える可能性があります。

Lit か Simple Lit か?

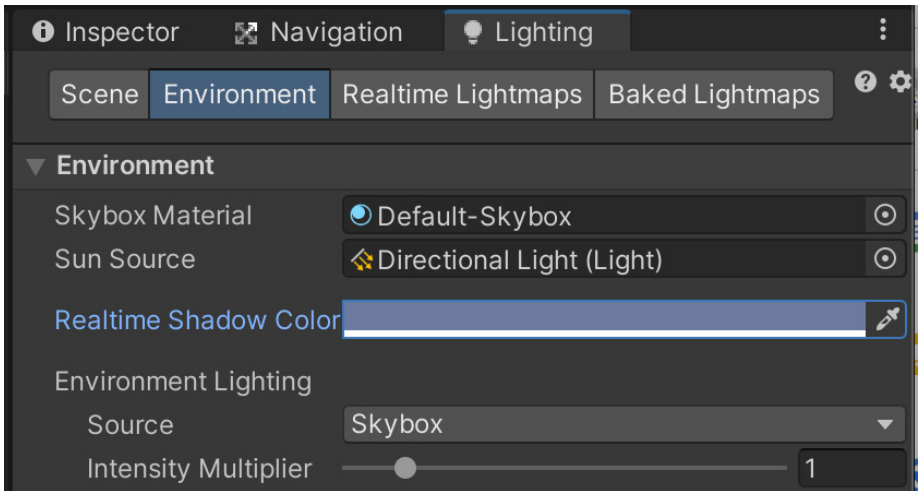
Lit シェーダーと Simple Lit シェーダーのどちらを選ぶかは、主にアートの観点から決定されます。Lit シェーダーを使用すると、アーティストがリアルなレンダリングを行うことがより容易になりますが、より様式化されたレンダリングが求められる場合は、Simple Lit の方が好ましい結果が得られます。

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、ライトはメインライトと追加ライトに分けられます。メインライトは影響が最も大きいディレクショナルライトです。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後で学びます。これは Object Per Light 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。ただし、カメラごとに使用できる動的ライトの数は、次の表で示されているように、ハードウェアによっても制限されます。

URP フォワードレンダラー使用時のカメラライトの制限数

ライトのタイプ	**カテゴリ**	**レンダリング可能なライトの最大数（モバイル以外）**	**レンダリング可能なライトの最大数（モバイル）**	**レンダリング可能なライトの最大数（OpenGLES 2.0）**	**影のサポート**
Directional	Main	1	1	1	True
Spot	Additional	256*	32*	16*	True
Point	Additional	256*	32*	16*	True
Directional	Additional	256*	32*	16*	False

* すべての追加ライトのバジェットは同じです。

シーンをカリングする際は、そのフレーム内でサポートされている動的ライトを最大数まで選択します。一方、オブジェクトのレンダリング時は、これらのライトのうち最も重要なものだけを選択して、各オブジェクトを動的に照らします。

動的ライトの数が少ないプロジェクトでは問題ないかもしれませんが、ライトを増やすと、異なるライトが動的にカリングされるため、ライトポッピングが発生する可能性があります。もちろん、シーン内の動的ライトの数が増えると、パフォーマンスコストが生じます。各動的ライトは、カメラに対してカリングされ、優先順位でソートされる必要があります。また、オブジェクトごとに各ライトをレンダリングするコストもかかります。普段のとおり、忠実度とパフォーマンスのバランスを保つようにしてください。

リアルタイムとミックスモードライト

リアルタイムライトとミックスモードライトは、まずカメラ錐台に対してカリングされます。オクルージョンカリングが有効化されている場合、シーン内の他のオブジェクトによって見えないライトもカリングされます。

カリングプロセス後に残った可視ライトのリストは、次にカメラからの各ライトの距離によりソートされます。可視ライトがある場合、上の表の制限が適用されます。例えば、シーン内に 1,000 個のライトが含まれていても、カメラの視界内にあるのが 200 個だけであれば、それらはすべて非モバイルプラットフォームの制限に当てはまります。

ここで、可視ライトのリストはオブジェクトごとにカリングされます。ライトは、オブジェクトのピボットにおける強度順にソートされます。これにより、明るいライトが優先されます。オブジェクトが、オブジェクトに対して許可される最大数以上のライトの影響を受けている場合、超過分のライトは無視されます。

ライトの制限に達している場合は、次のオプションを検討してください。

— ライトの位置と強度が静的な場合は、ライトをベイクし、ライトプローブを使用して動的ジオメトリのレンダリングに追加します。詳細については、ライトプローブのセクションを参照してください。

— ライトレイヤーを使用して、どのジオメトリがどのライトの影響を受けるかを制限します。

— ライトの範囲を制限します。ディレクショナルライトはグローバルであるため、このオプションは適用されません。

— エミッシブマテリアルを使ってライティングを模倣します。

ここで説明するライトの制限は、URP 使用時のデフォルトレンダラー、フォワードレンダラーで適用されるものです。

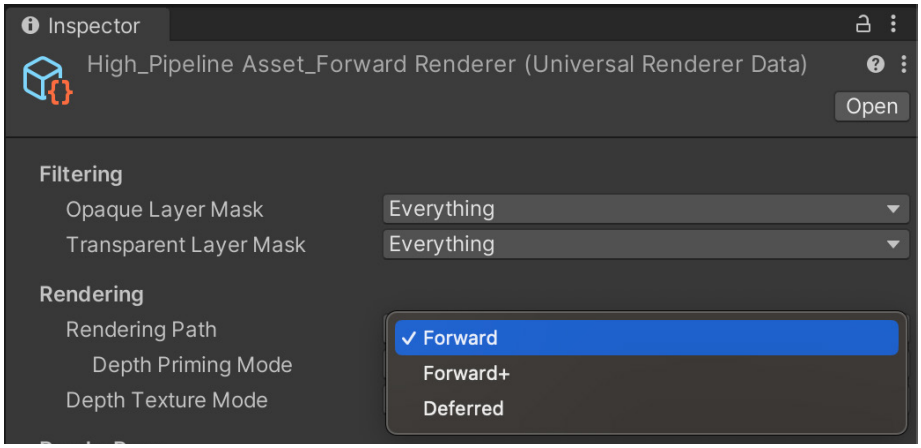
フォワードレンダラーは、シングルパスアプローチを使用して、1 回のドローコールでオブジェクトのライティングを計算します。これはパフォーマンスの高いオプションですが、GPU の制限により、ピクセルの色を設定する際、オブジェクトが考慮できるライトの数が制限されます。ハイエンドのハードウェアをターゲットにしている場合は、URP のディファードレンダリングパスを使用することで、これらの制限を回避できます。

レンダリングパスの比較

Unity 2022 LTS では、レンダリングオプションとして、フォワード、フォワード+、ディファードの 3 つが提供されています。

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、カメラ単位の制限が適用	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 バッファで法線の量子化（精度は落ちるが性能は向上） • 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が及ぶ可能性あり） 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	Yes	Yes	No
頂点ライティング	Yes	No	No
カメラスタッキング	Yes	Yes	制限付きでサポート：Unity は、ディファードパスを使用して、ベースカメラのみをレンダリングし、フォワードレンダリングパスを使用して、すべてのオーバーレイカメラをレンダリングします。

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。

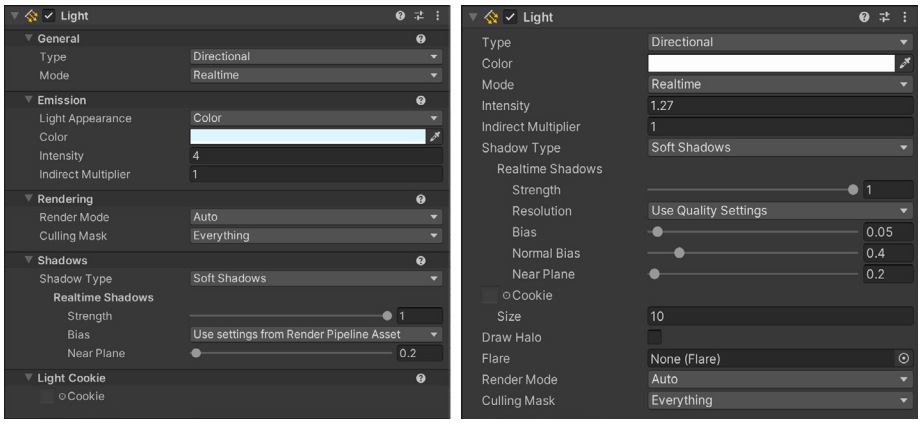
— **「Additional Lights」>「Per Object Limit」**：Unity はこのプロパティを無視します。

— **「Reflection Probes」>「Probe Blending」**：リフレクションプローブブレンディングは常にオンになっています。

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。

ビルトインレンダーパイプラインの場合と同様に、URP はディレクショナル、スポット、ポイント、エリアライトをサポートしていますが、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（左）と、ビルトインレンダーパイプラインの「Light Inspector」パネル（右）の比較。

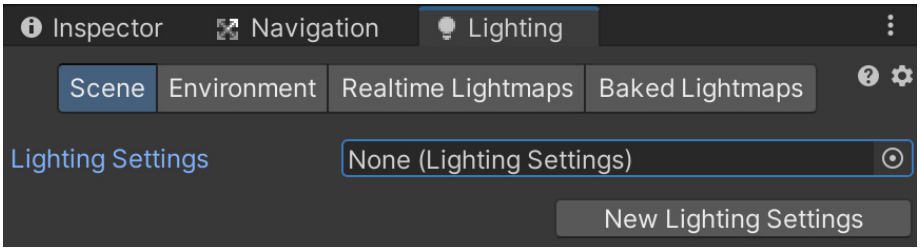
上の画像は、Light Inspector の 2 つのバージョンでライトプロパティがどのように表示されるかを示しています。URP バージョンには、ライトがディレクショナルかポイントかによって分けられる、以下の 5 つの制御グループと、スポットとエリアライト用の追加の形状グループがあります。

次の表は、URP とビルトインレンダーパイプラインの Light Inspector の違いをリスト化したものです。

URP の Light Inspector のプロパティ	**説明**	**ビルトインレンダーパイプラインの Light Inspector のプロパティ**
Light Appearance	「Color」または「Filter and Temperature」を選択します。「Color」では、放出される光の色を設定します。「Filter and Temperature」では、色（フィルター）と温度の両方を使用して、寒色と暖色の間でライティングを切り替えます。	なし

Bias	Bias はシャドウアクネを制御します。デフォルトでは、URP アセットを使用します。または、この Inspector を使用して、カスタム値を設定できます。	Bias/Normal Bias
Light Cookie	テクスチャがライトクッキーを使用するように設定されていて、ライトの種類がディレクショナルである場合、新しいパネルではクッキーの x と y のサイズ、およびそのオフセットを制御できます。ポイントライトのクッキーはキューブマップである必要があります。URP は色付きクッキーをサポートしていますが、ビルトインレンダーパイプラインはグレースケールのクッキーのみサポートしています。	Cookie
Shape：Spot	スポットライトの場合に、内側と外側の円錐の角度を制御できるようになりました。	Spot Angle、Range
Shape：Area	これは、エリアライトの形状を制御するために使用されます。	Shape、Width、Height、Radius
なし	これは、ビルボードや、ライトの中心にあるスフィアのアルファ値を制御する Fresnel シェーダーを使って簡単に再現できます。詳細については、ハローライトのセクションを参照してください。	Draw Halo
なし	URP でのレンズフレアの実装方法については、レンズフレアのセクションを確認してください。	Flare

新規シーンのライティング

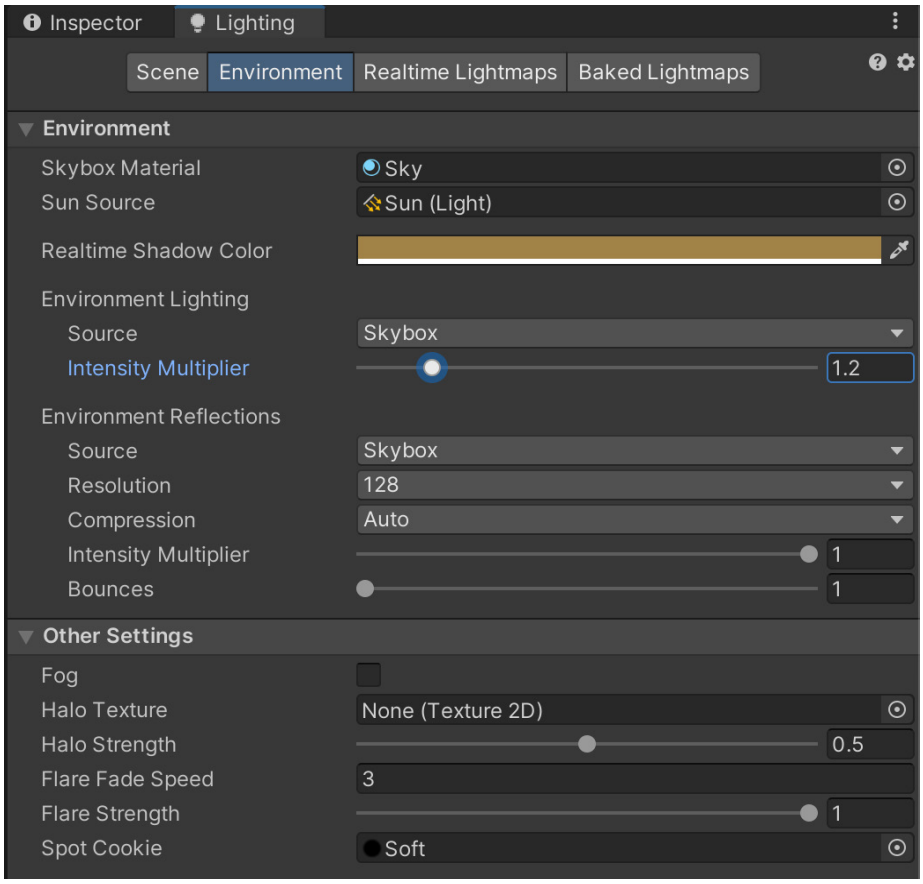


ライティング設定アセットの作成

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（上の画像参照）。**「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」パネルで適用した設定は、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と、設定が切り替わります。

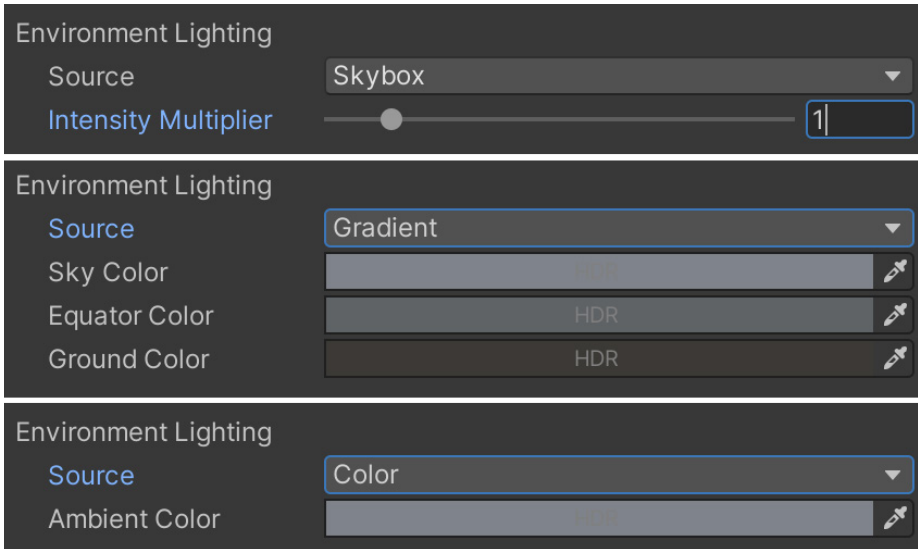
アンビエントまたは環境ライティング

ビルトインレンダーパイプラインと URP でのアンビエント/環境ライティングの定義方法に変更はありません。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。

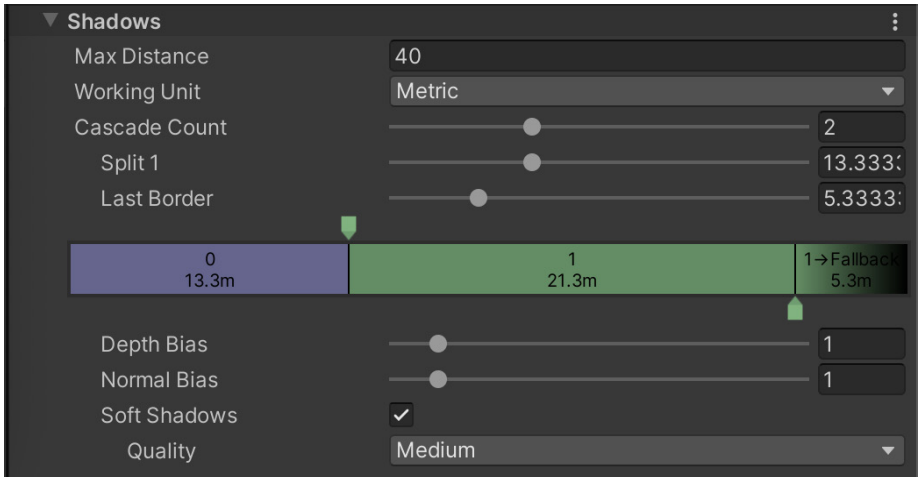


環境ライティングオプション

シャドウ

ビルトインレンダーパイプラインから URP に移行する際の最大の変更は、影の設定方法にあります。

影の設定は、**「Project Settings」>「Quality」**からアクセスできなくなりました。
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、影の忠実度を定義するために使用できるレンダーパイプラインアセットを介してシーンを表示する方法について説明します。



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。最後の設定は、シャドウマップの解像度です。

Unity で影を操作したことがある方なら、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が、より多くの処理能力とメモリが必要となります。影の処理が増える要因としては、以下のものがあります。

— 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。このメインライトを対象とした数値は、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。

— 画面で可視化されているシャドウレシーバー（すべて含める必要があります）

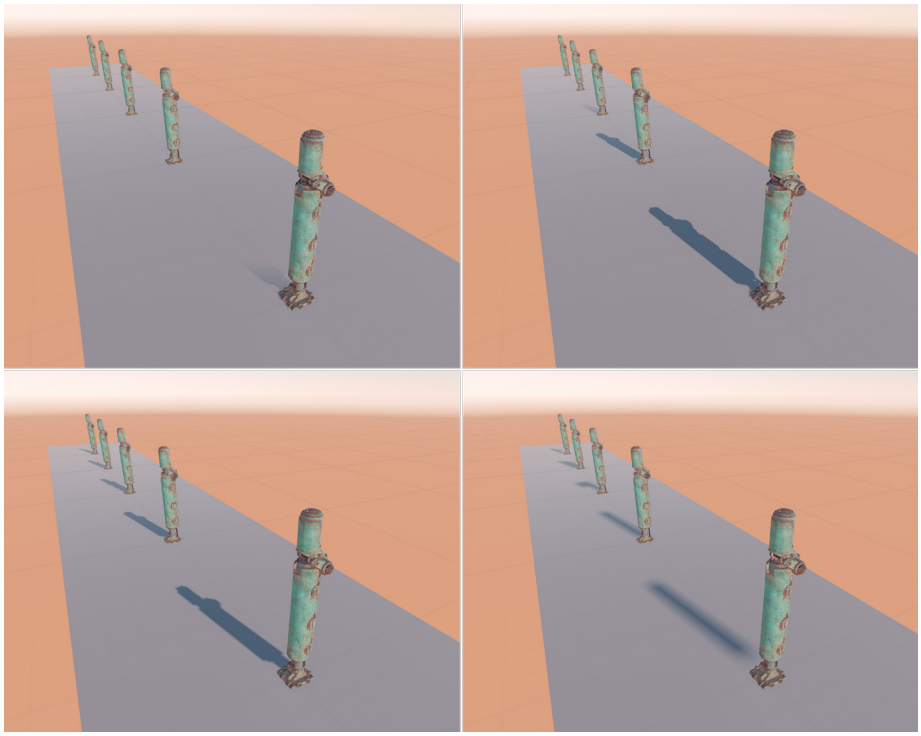
— シャドウカスケードの分割

— シャドウフィルタリング（ソフトシャドウ）

最高の解像度が必ずしも理想的とは限りません。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。以下の幽霊屋敷の部屋の画像では、前景の椅子が机の引き出しに影を落としていますが、これは、解像度が 1024 より大きいと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：シャドウの最大距離



メインライトシャドウの様々な最大距離：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。これはシーン単位で設定されます。上の画像では、ポールの間隔が 10 単位です。Max Distance は、10 ～ 400 単位で設定できます。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 単位の距離で影が途切れています。60 単位（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はるかに大きい場合、シャドウマップの範囲が広くなりすぎる結果になります。つまり、ショット内領域の解像度が必要なレベルよりもはるかに低くなるということです。

「Max Distance」プロパティは、ユーザーが目視できるものと、シーンで使用されている単位で直に関連付けられている必要があります。許容できる影になる範囲で最小の距離を目標にして設定してください（下記の注記を参照）。プレイヤーがカメラから 60 単位離れた動的オブジェクトの影しか見えないようにする場合、「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の影が表示されることになりますが、動的シャドウだけがシャドウディスタンスの範囲内で描画されます。

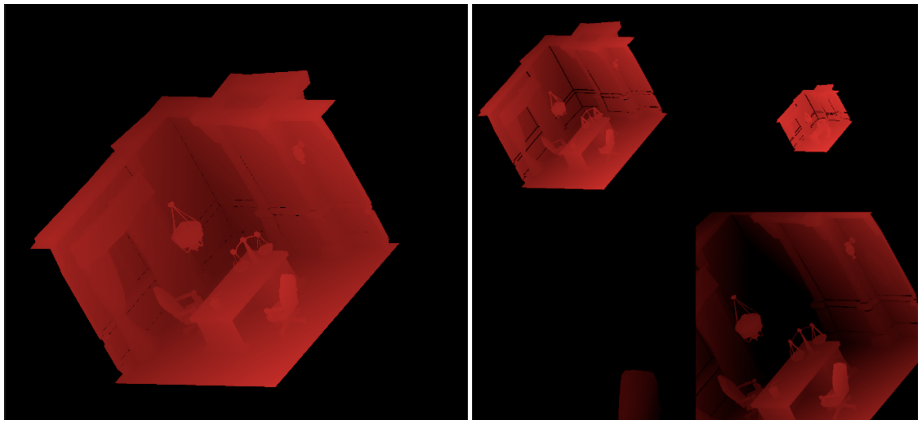
> **注：**URP では、「Shadow Projection」として「**Stable Fit**」のみがサポートされています。これは、「Max Distance」の設定をユーザーに委ねるものです。ビルトインレンダーパイプラインでは、「Shadow Projection」プロパティとして「Stable Fit」と「Close Fit」の両方がサポートされています。後者のモードの場合、左下と右下の画像は同じ品質になります。「Close Fit」では、シャドウディスタンス平面が最後のキャスターに合わせて縮小されるためです。欠点は、「Close Fit」の場合、影の錐台が「動的に」変えられるため、影に揺らぎ効果がもたらされる場合があるということです。

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ので、影の解像度を下げて、シャドウマップのより多くの部分を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。

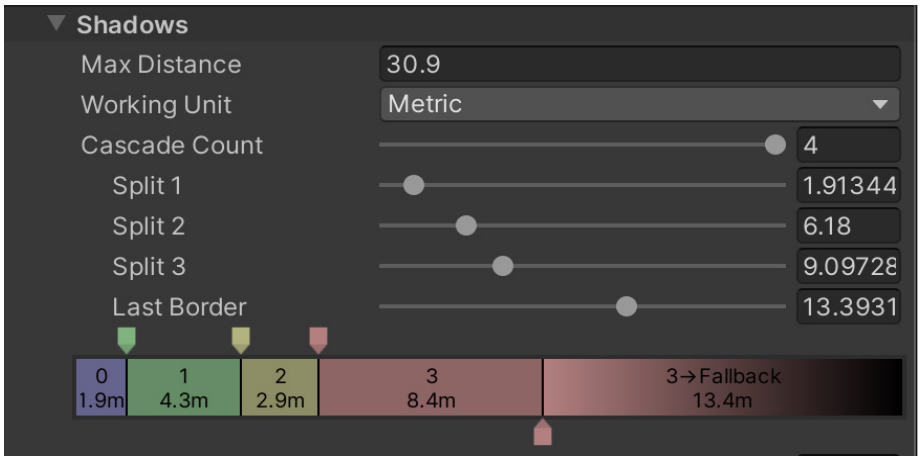
下の画像は、幽霊屋敷の部屋にある椅子と机が映ったシーンのシャドウマップです。左側の画像では、カスケードカウントは 1 です。マップがエリア全体を専有しています。左の画像では、カスケードカウントは 4 です。マップには 4 つの異なるマップが含まれており、各エリアにより解像度の低いマップが割り当てられることに注意してください。

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、最大距離が大きい場合は、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の影をより良好に表示することができます。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。



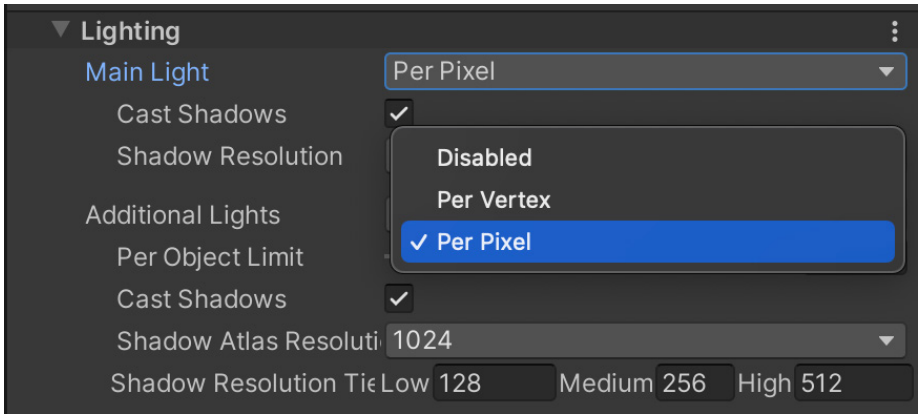
カスケードカウントを 1 に設定した場合（左の画像）と、4 に設定した場合（右の画像）のシャドウマップ

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単位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。「Working Unit」として「Metric」を使用する場合は、常に、最後のカスケードが最後のシャドウキャスターの距離（最大）になるように選択してください。



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

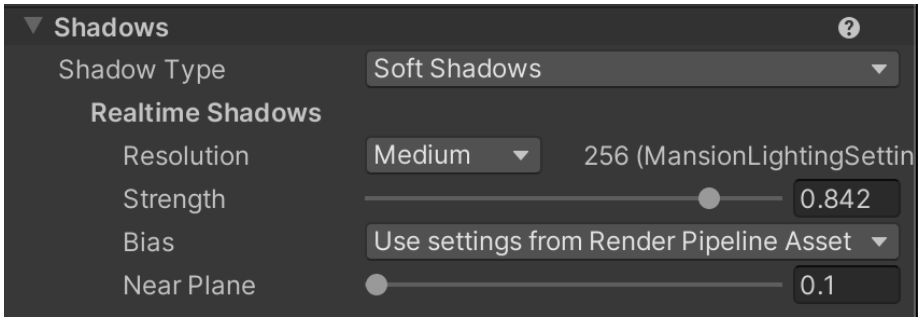


URP アセットの追加ライトで使用できる設定

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。メインライトが常に最も明るいディレクショナルライトであることを覚えておいてください。影を伴う追加ライトには、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。

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、影に対して機能するのは「Per Pixel」だけです。

「**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。ポイントライトでは 6 つのシャドウマップが投影され、キューブマップが作成されます。これは、光がすべての方向に投影されるためです。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を設定する際には、3 つの影の解像度の階層での組み合わせに加えて、「Hierarchy」ウィンドウでライトを選択する際に Light Inspector から選択した解像度が使用されます。

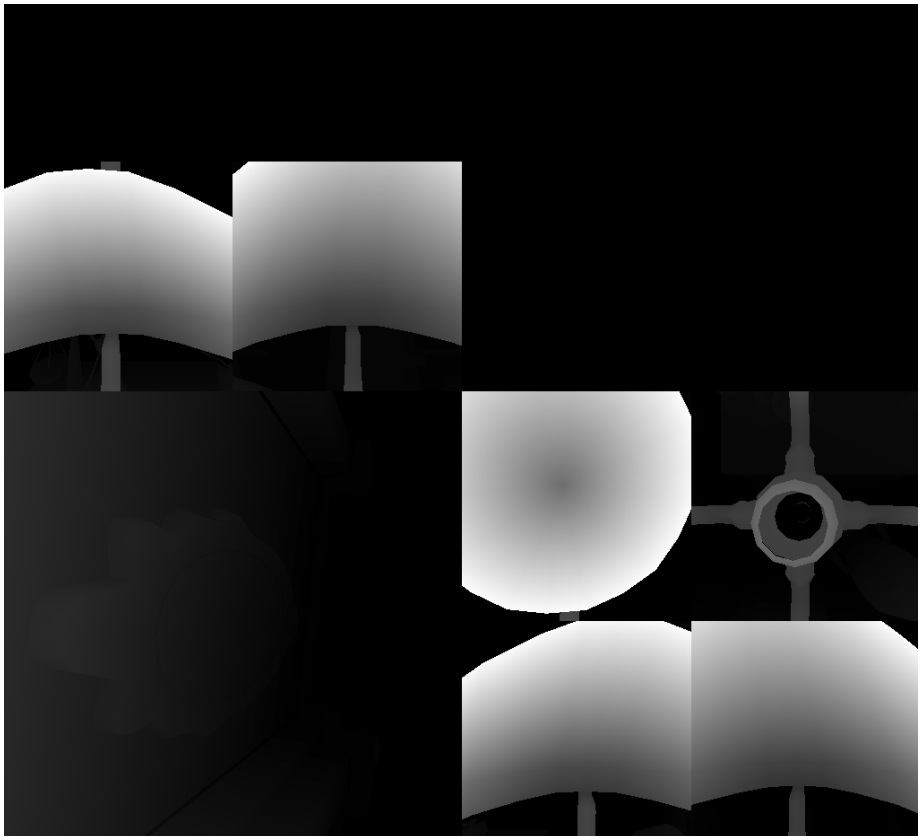


Light Inspector の Shadows グループ

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	1x1	1
2 〜 4	2x2	2
5 〜 16	4x4	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。

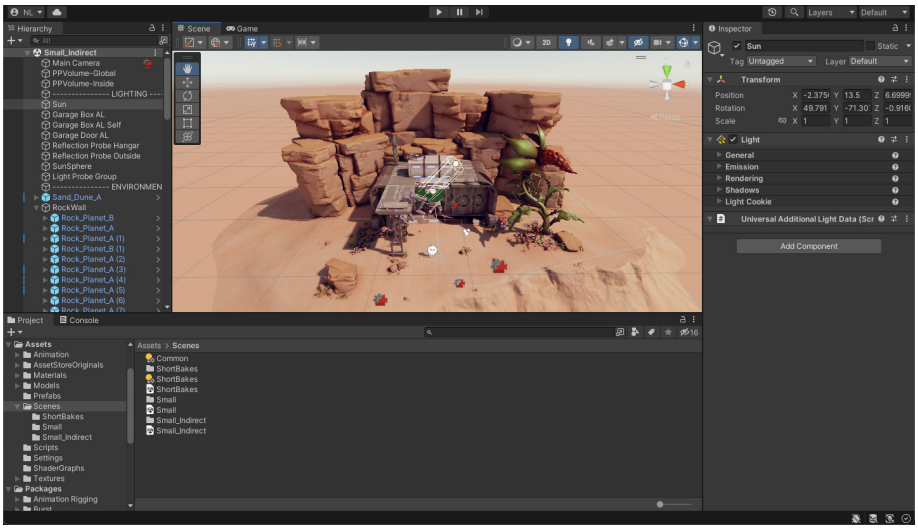
ライトモード

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。デザイン時に一度計算し、そのデータをジオメトリのレンダリング時に使用できます。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。

ビルトインレンダーパイプラインと URP の間で、ライトマッピングのワークフローに違いはありません。Unity の FPS Sample プロジェクトを使って手順を説明しましょう。

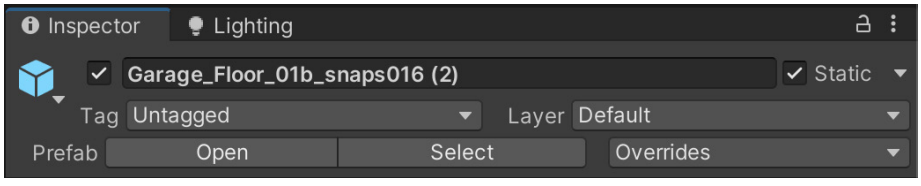
> **注:** 低頻度とは、ライトマップが画面の更新よりもはるかに低いレートで更新されることを指します。スペキュラーローブはリアルタイムライトによってのみ計算可能です。ライトプローブを使用して動的オブジェクトにグローバルイルミネーション（GI）を適用できますが、この方法でも低周波のディフューズライトしかキャプチャできません。ビルトインレンダーパイプラインはライトプローブプロクシボリューム（LPPV）をサポートしており、ライトマップが動的オブジェクトに提供するのと同じレベルの品質をライトプローブに提供します。しかし、LPPV はスケールできない比較的遅いシステムであるため、URP ではサポートされていません。代わりに、URP ではアダプティブプローブボリュームがサポートされる予定です。これはライトマップを置き換え、静的オブジェクトと動的オブジェクトの両方で機能する可能性があります。

以下は、Unity プロジェクトの FPS サンプル「The Inspection」のスクリーンショットです。このプロジェクトはこちらからダウンロード可能です。このシーンは、URP でリアルタイムとベイクのライティングを使用する方法を示したものです。

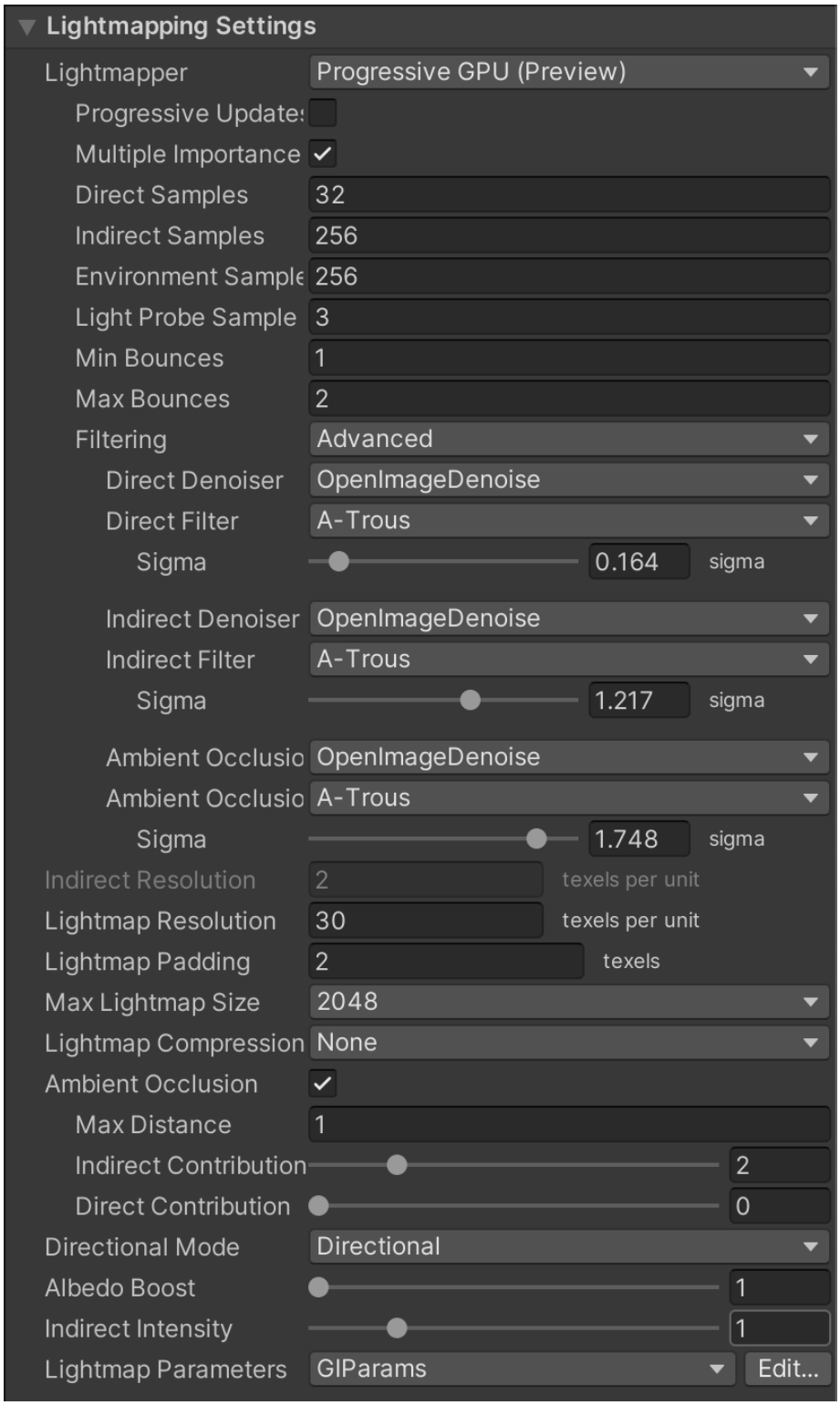


Unity 作 FPS サンプル「The Inspection」のシーン

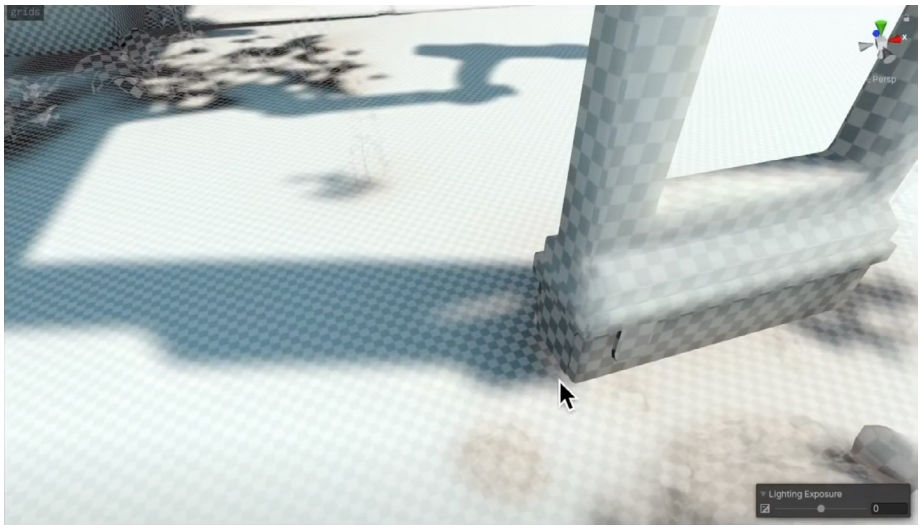
1. FPS サンプルプロジェクトのシーンには、その大部分に静的ジオメトリが含まれています。このジオメトリをライトマッピングに含めるには、Inspector の右側にある「**Static**」ボックスをクリックします。



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。「**Progressive GPU (Preview)**」を選択すると、ライトマップの生成を高速化できます（GPU がそれをサポートしている場合）。

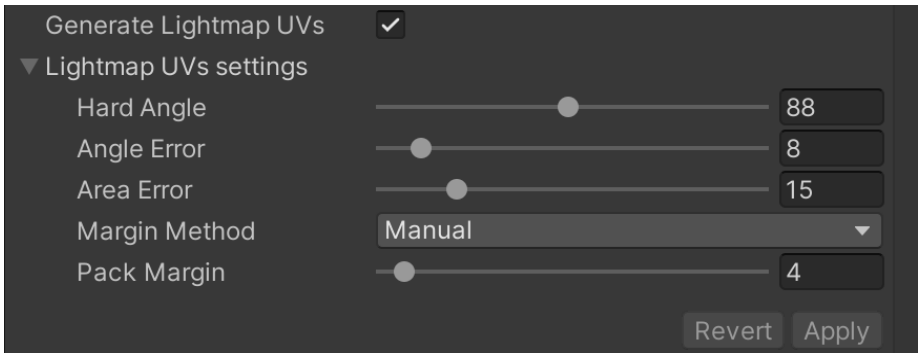


3. フィルタリングでは、ノイズを最低限に抑えるためにマップがぼかされます。これにより、あるオブジェクトが別のオブジェクトと接する場所で影にズレが生じることがあります。このアーティファクトを最小限に抑えるには、**A-Trous** フィルタリングを使用します。ライトマッピングに使用できる設定の詳細については、プログレッシブライトマッピングのドキュメントを参照してください。

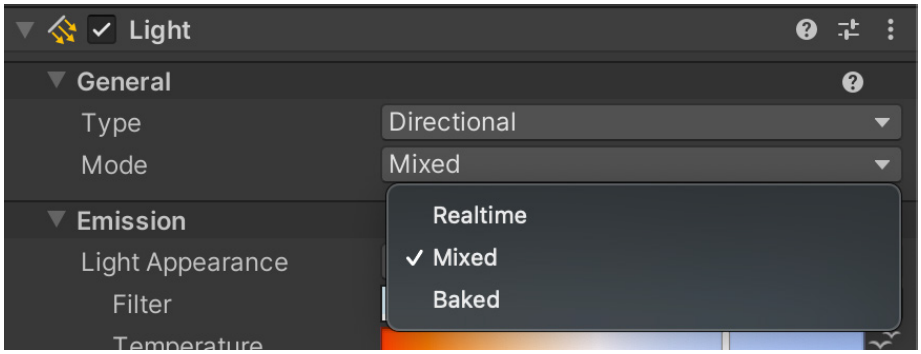


フィルタリングがオブジェクト間の影に与える影響

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 **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、動的オブジェクトと静的オブジェクトの両方を照らします。

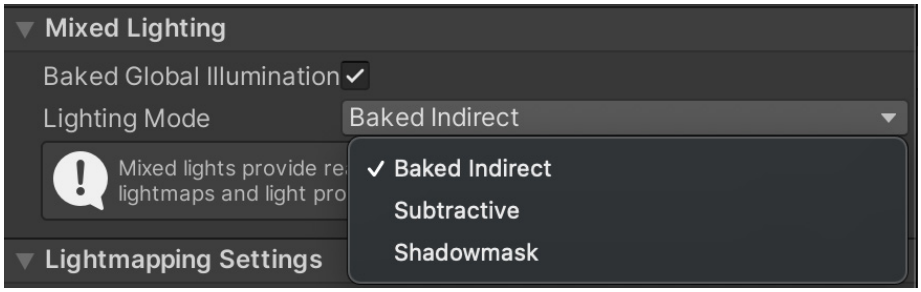


6. 混合ライトを使用する場合は、**「Window」>「Rendering」>「Lighting」>「Scene」**から、**「Light Mode」**を**「Baked Indirect」**、**「Subtractive」**、または**「Shadowmask」**に設定します。

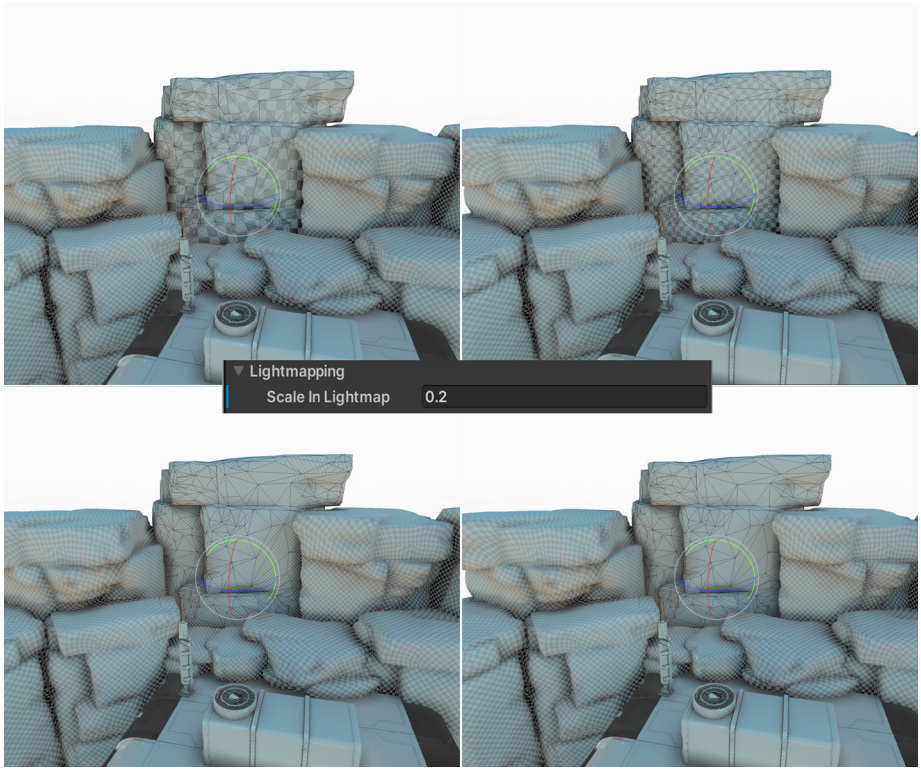
    a. Baked Indirect：間接光の影響のみがライトマップとライトプローブにベイクされます（ライトの反射のみ）。直接光と影はリアルタイムになります。これは高コストなオプションなので、モバイルプラットフォームに対しては理想的ではありません。しかし、静的ジオメトリと動的ジオメトリの両方で、正確な影と直接光を表現できます。

    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、動的ジオメトリによって投影された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。そして、減算された値の最小の色と、元のベイクされた色を選択します。これはローエンドハードウェアに対する最適なオプションです。

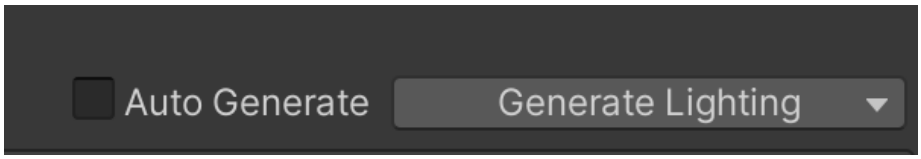
    c. Shadowmask：「Baked Indirect」モードと似ていますが、「Shadowmask」では、動的シャドウとベイクされたシャドウの両方が結合され、影が遠くにレンダリングされます。これは、追加のシャドウマスクテクスチャを使用し、ライトプローブに追加情報を保存することで実現されます。このオプションでは、最も忠実度の高い影が提供されますが、メモリ使用とパフォーマンスの点で最も高コストなオプションでもあります。近距離のショットについては、「Baked Indirect」と見た目が同じです。遠くを見たときに違いがわかるので、オープンワールドのシーンに適しています。処理コスト上の理由から、ミッドエンドやハイエンドのハードウェアでのみ使用することをお勧めします。



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」）によって変わってきます。



ライトレイヤーを使用したオブジェクトのハイライト

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

— ライトエクスプローラーのドキュメント

— ライトマッピングに関する一般的な 5 つの問題とその解決のためのヒント

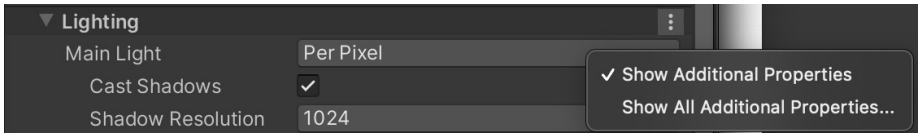
レンダリングレイヤー

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により、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。

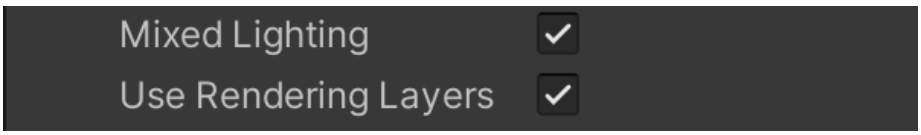
レンダリングレイヤーを使用したオブジェクトのハイライト

レンダリングレイヤーの設定手順は以下の通りです。

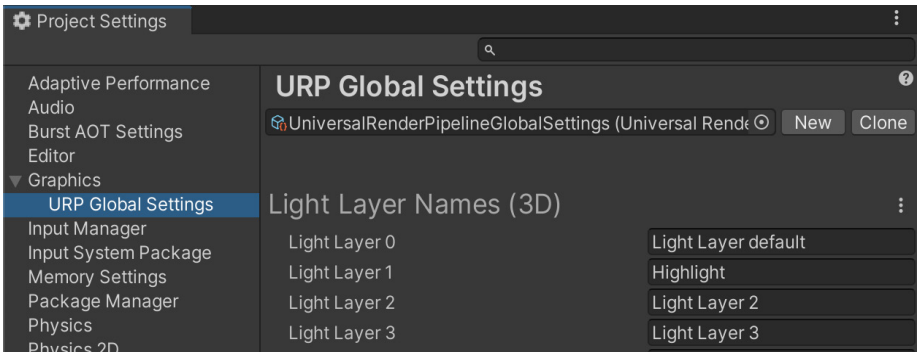
1. 「**URP Asset**」を選択します。「Lighting」セクションで、縦の省略記号 (⋮) のアイコンをクリックし、「**Show Additional Properties**」を選択します。



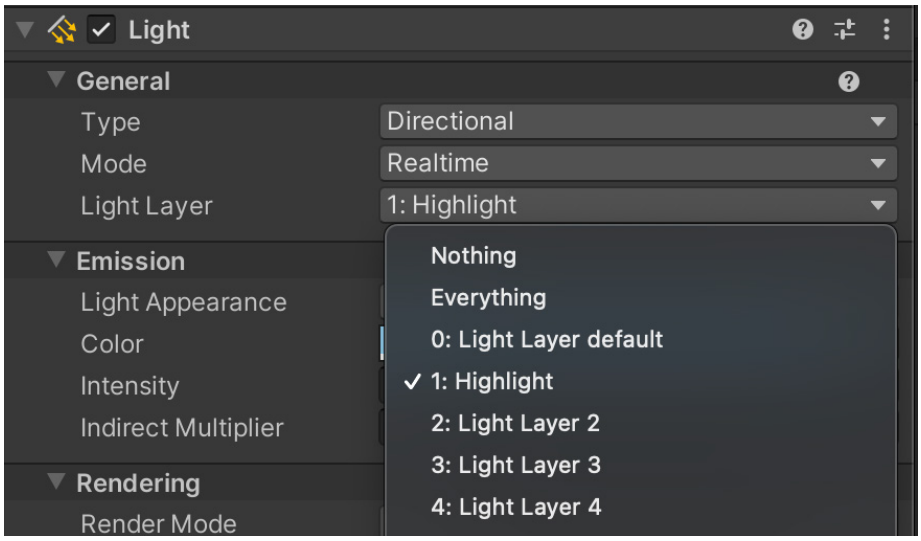
2. 「**Use Rendering Layers**」という新しい設定が、「Lighting」セクションの下に表示されます。



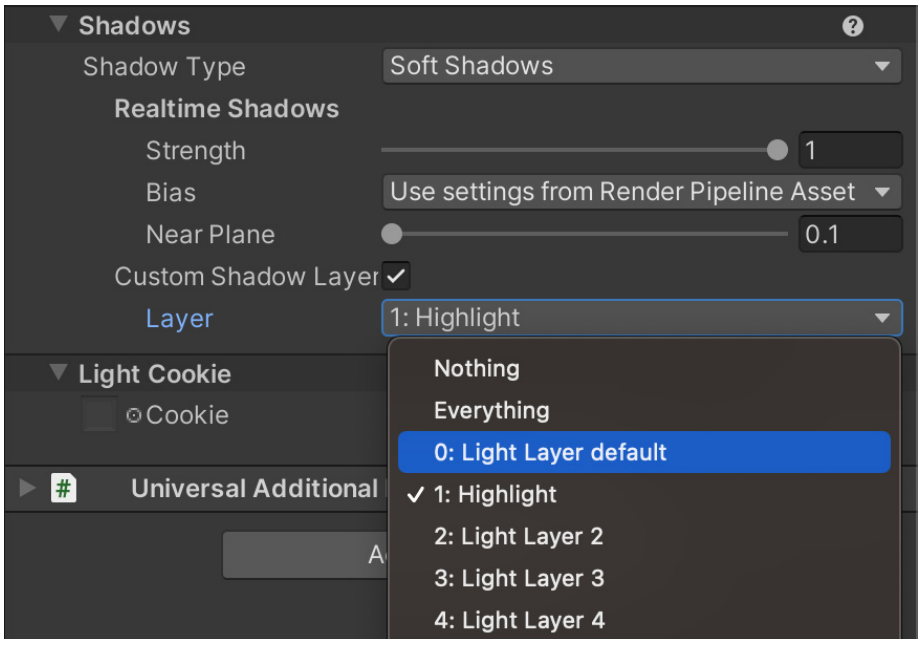
3. 「**Project Settings**」>「**Graphics**」>「**URP Global Settings**」で、レンダリングレイヤーの名前を変更します。



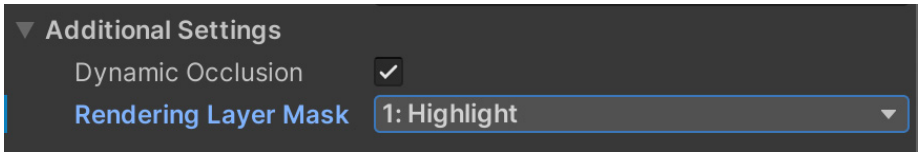
4. これでレンダリングレイヤーが有効になり、**Light Inspector** に**レンダリングレイヤー**のドロップダウンが追加されます。ライトは複数のレイヤーに影響を与えることができます。



5. レンダリングレイヤーを有効にした場合、カスタムシャドウレイヤーを設定する必要があります。新しいライトは、シーンの**メインライト**または自身の錐台から影を投影できます。



6. 最後に「**Hierarchy**」ウィンドウで、適用するオブジェクトを選択して**レンダリングレイヤーマスク**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。

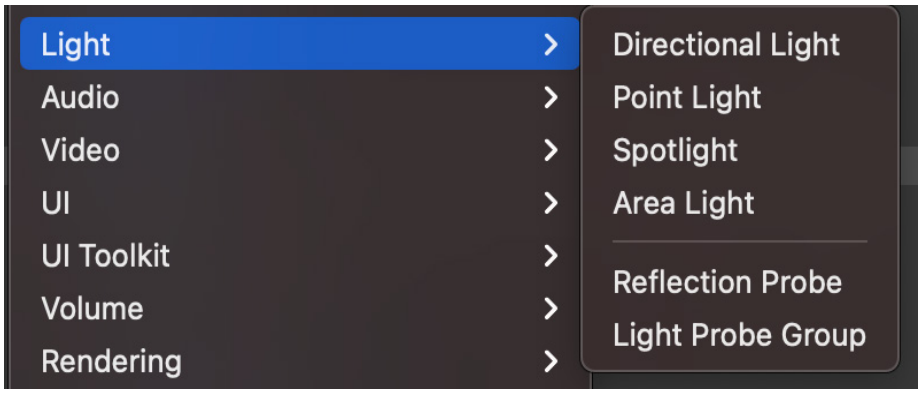
```
Renderer renderer = GetComponent<Renderer>();
int layerID = 1;
int mask = 1 << layerID;
renderer.renderingLayerMask = (uint)mask;
```

ライトプローブ

前のセクションで見たように、混合ライティングモードを使用することで、ベイクされたオブジェクトと動的オブジェクトを「Light Mode」セクションで組み合わせることができます。このモードを使用する場合は、シーンにライトプローブを追加することをお勧めします。ライトプローブは、開発者が**「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。下の画像（FPS サンプル「The Inspection」）では、格納庫の中と外でのロボットのキャラクターの見え方が確認できます。

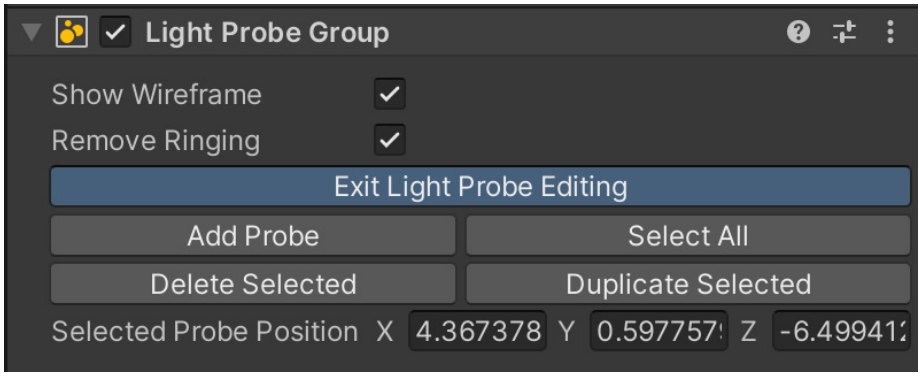
格納庫の中と外で、ライトプローブによってライティングレベルに影響を受けているロボット

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、「**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、ライト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、追加の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、Inspector で「**Light Probe Group**」>「**Edit Light Probes**」をクリックします。



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

Scene 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

ライトプローブの配置には時間がかかることもありますが、この例のようなコードベースのアプローチによって、編集を高速化することもできます（特に大規模なシーンの場合）。

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。

リフレクションプローブ

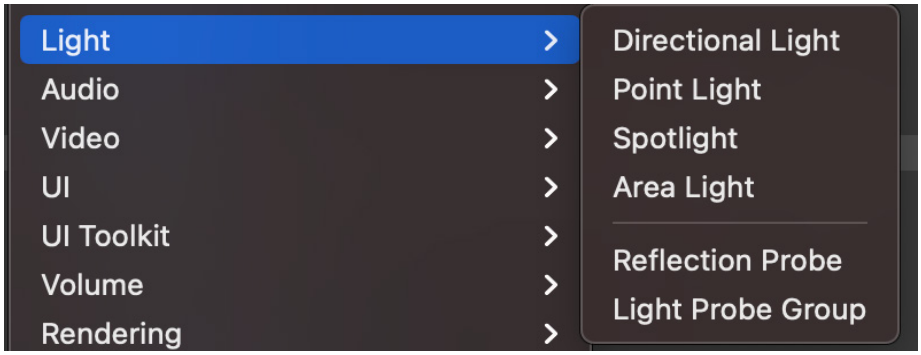
Maya や Blender などのレイトレーシングツールでは、反射面の各フレームピクセルの値を正確に計算するのに時間がかかることがあります。このプロセスはリアルタイムレンダラーでは時間がかかりすぎるため、ショートカットがよく使用されます。

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

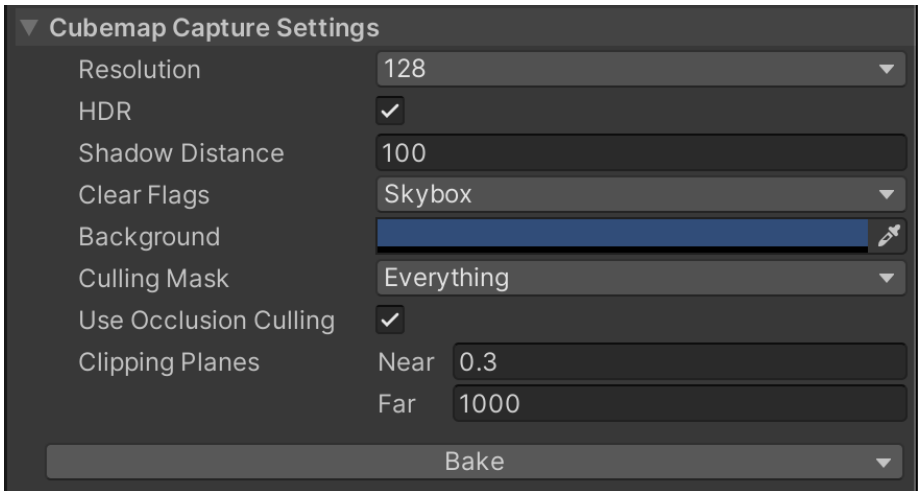
基準として使用することができます。また、プローブ間をブレンドするようにシーンを設定することもできます。

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



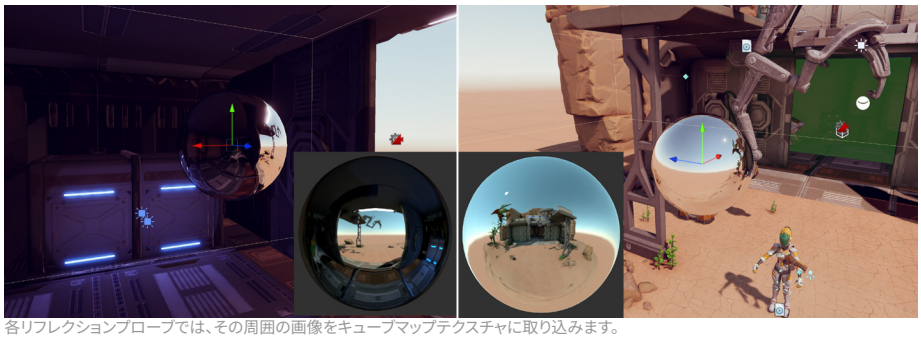
リフレクションプローブの作成

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。



リフレクションプローブの設定

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す。ブレンディングは「**Renderer Asset Settings**」パネルで有効化できます。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。この段階的な遷移によって、オブジェクトがゾーン境界を横断するときに、別のオブジェクトが反射の中に突然入ってくる状況を回避することができます。

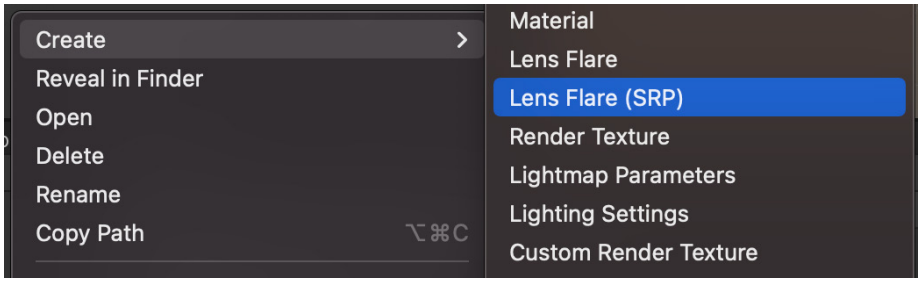
ボックス投影

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、オブジェクトに様々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。

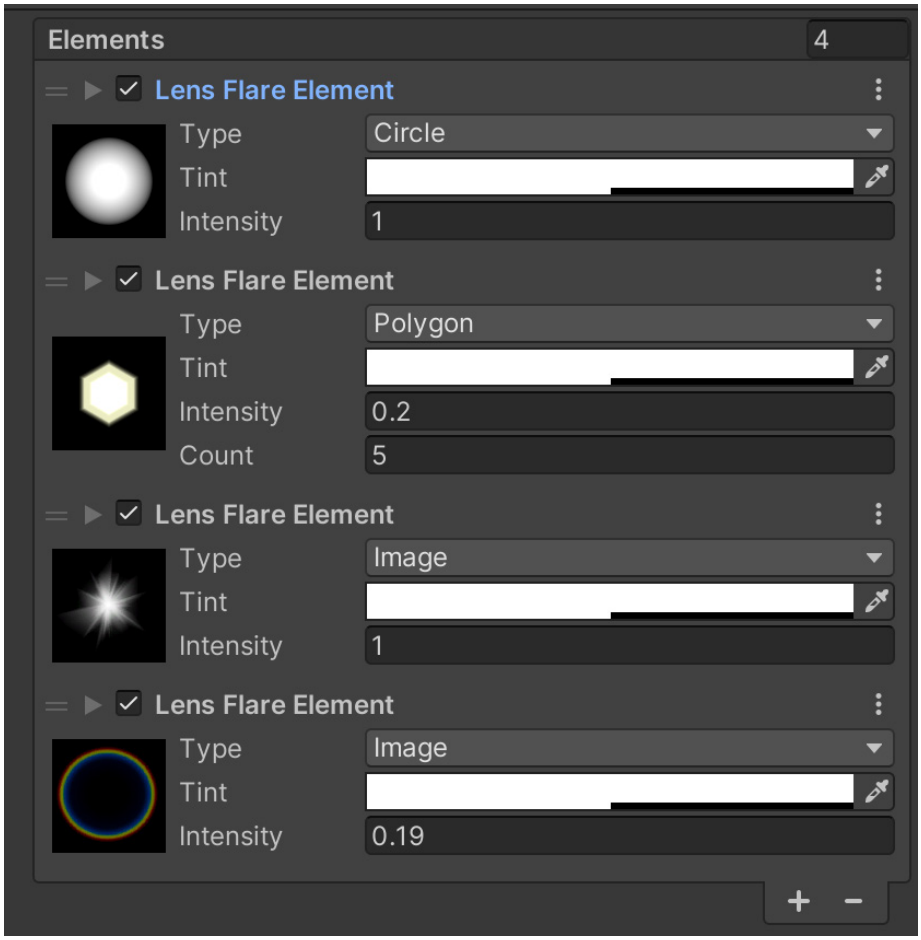
Lens Flare

URP でのレンズフレアの作成ワークフローが更新されました。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」>「Lens Flare (SRP)」**を選択します。



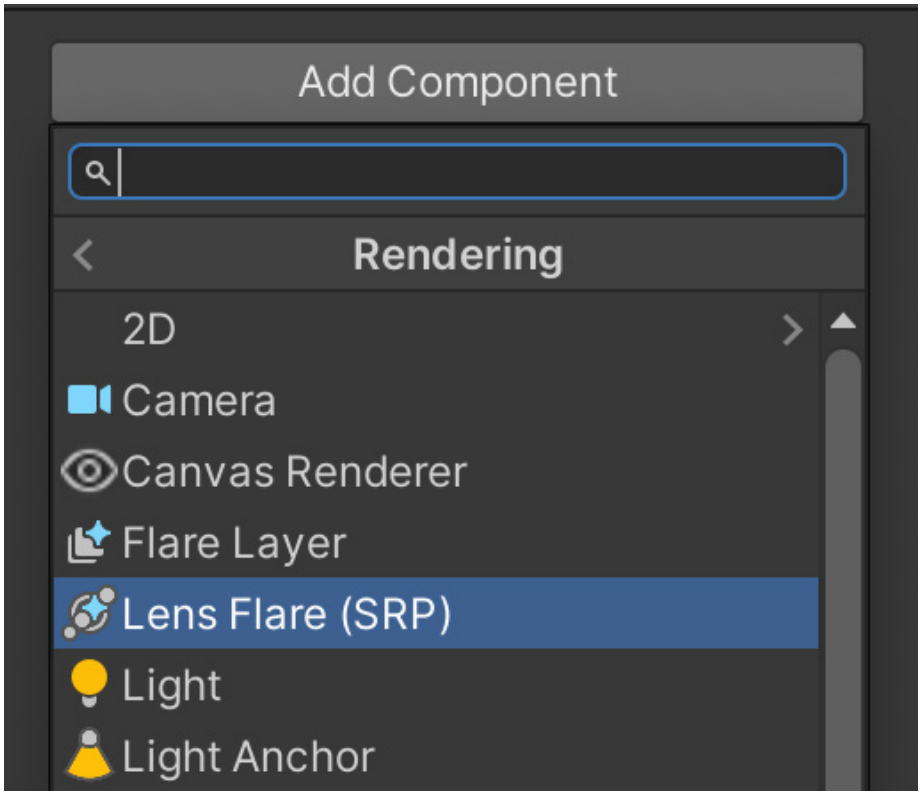
レンズフレア（SRP）データアセットの作成

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」を調整することで、フレアの形状を設定します。



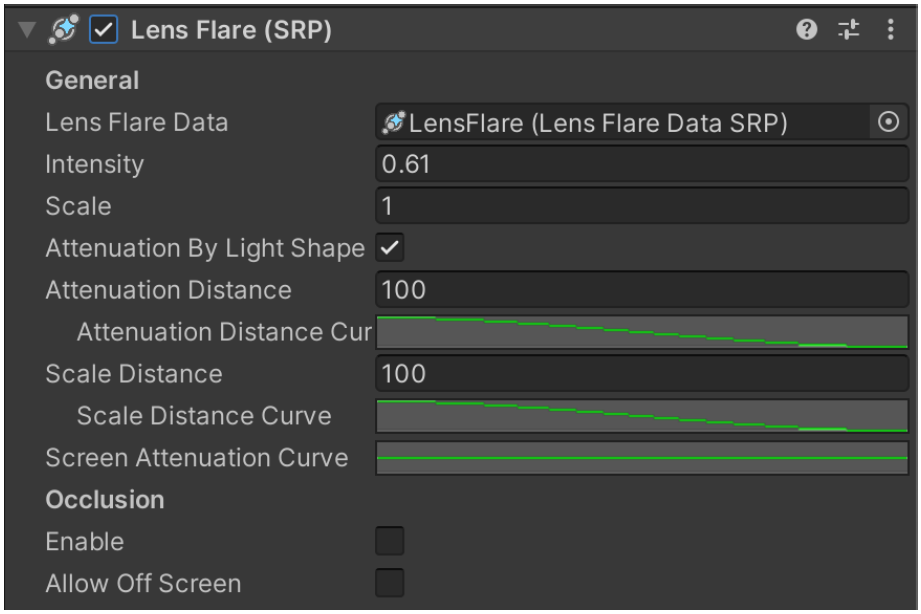
レンズフレア要素の追加と設定

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。



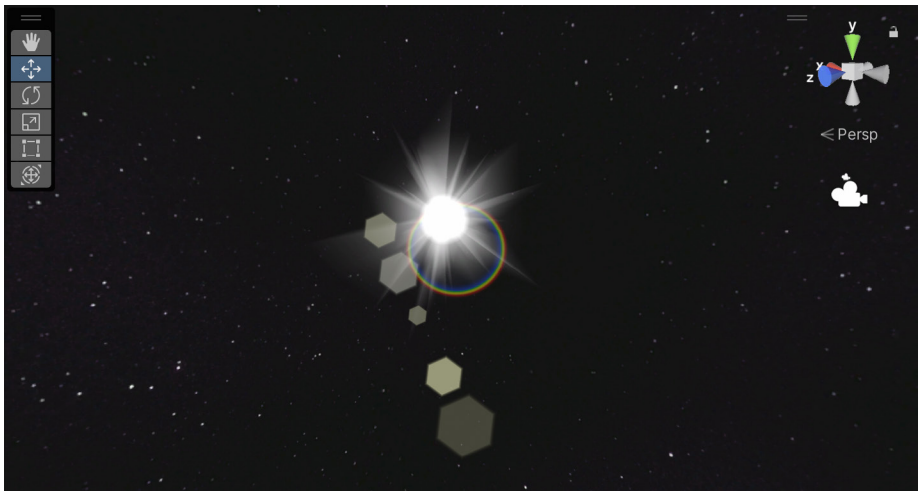
レンズフレアのレンダリングの設定

このコンポーネントの**「Settings」**パネル（以下の画像参照）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

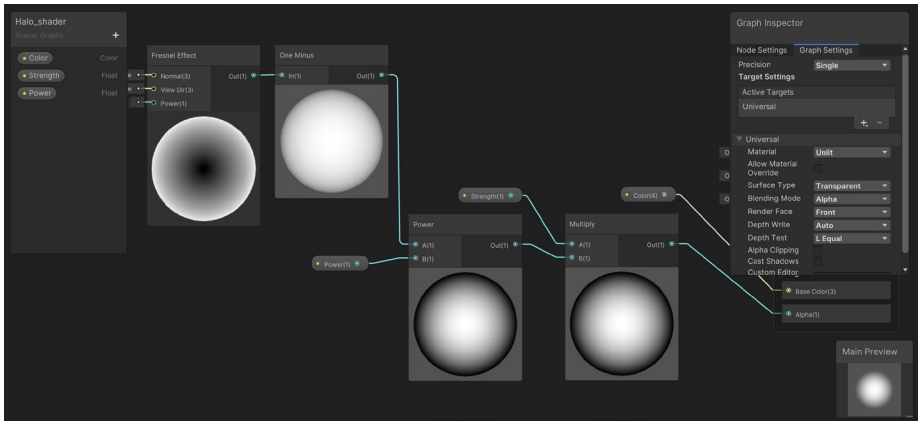
システムは非常に柔軟です。



レンズフレアの例

ライトハロー

URP では、ライトに対して「Draw Halo」のオプションを使用できませんが、ビルボードで簡単に模倣することができます。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 番目の画像は上記の方法を利用したシェーダーの 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。

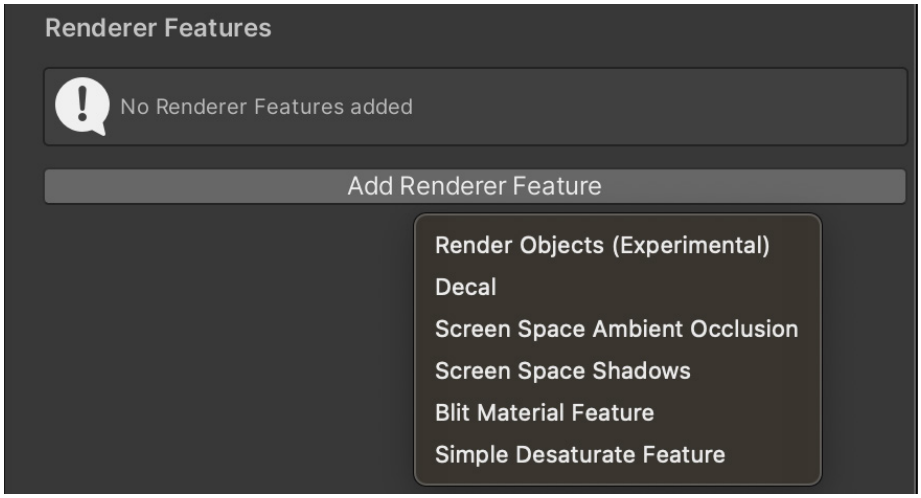


Shader Graph を使用したフレネル透明度

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ライトハロー

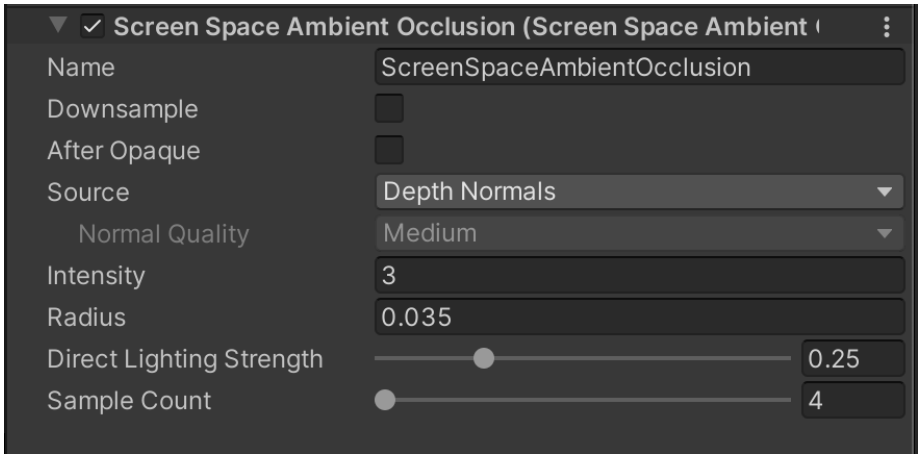
スクリーンスペースアンビエントオクルージョン

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

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

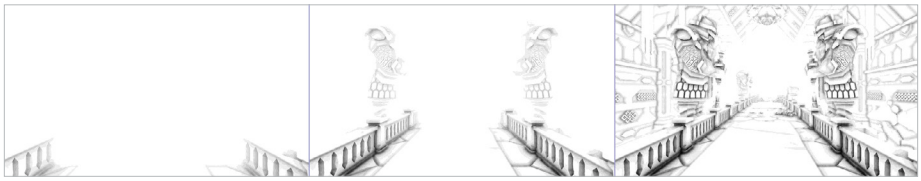
— **Method：**このプロパティは、SSAO エフェクトが使用するノイズのタイプを定義します。

— **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。

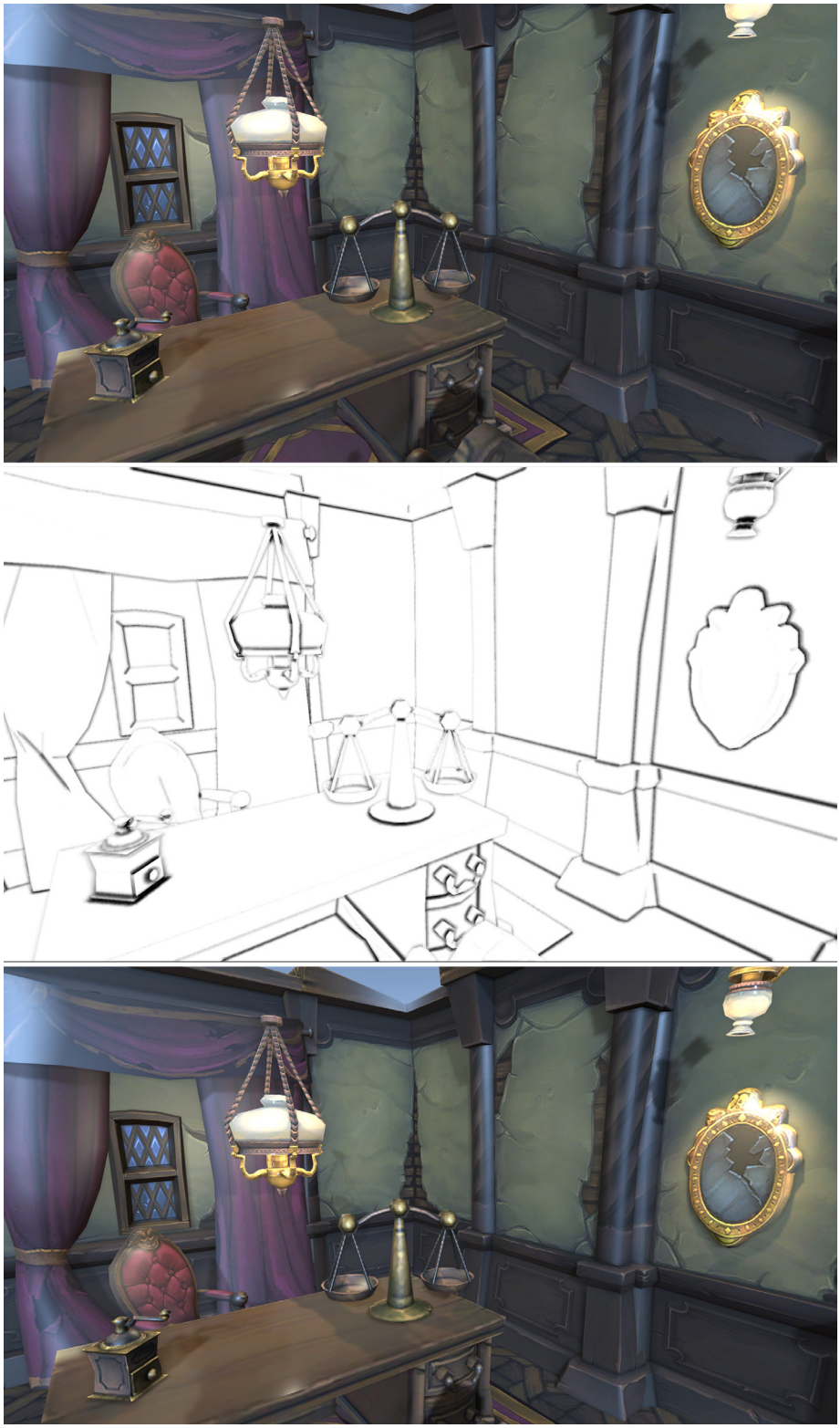
— **Direct Lighting Strength：**このプロパティは、直接光に露出された領域で効果がどの程度見えるかを定義します。



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。次の 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。



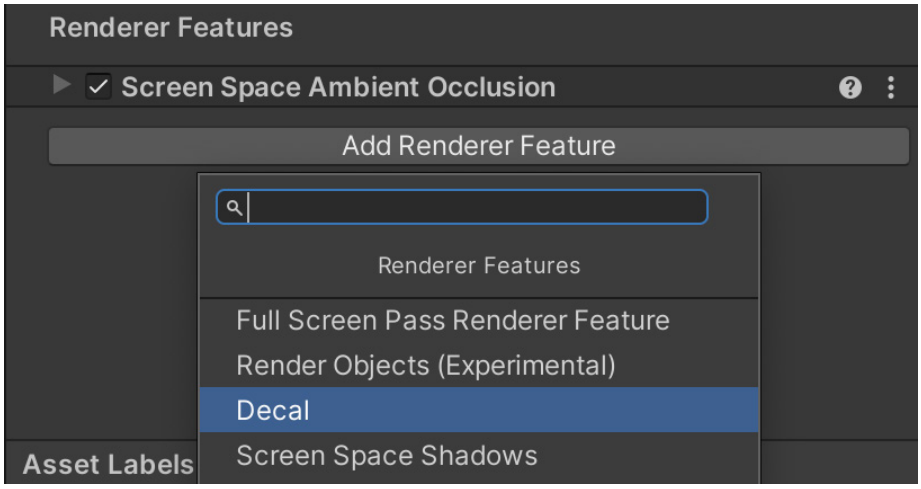
幽霊屋敷の部屋のスクリーンショット（上が SSAO なし、真ん中が SSAO を適用したもの、下が SSAO を適用してレンダリングしたもの）

SSAO はポストプロセスのテクニックのひとつで、このガイドの後半で詳細に説明します。

デカール

Decal Projector は、メッシュにディテールを追加するための優れた方法です。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。

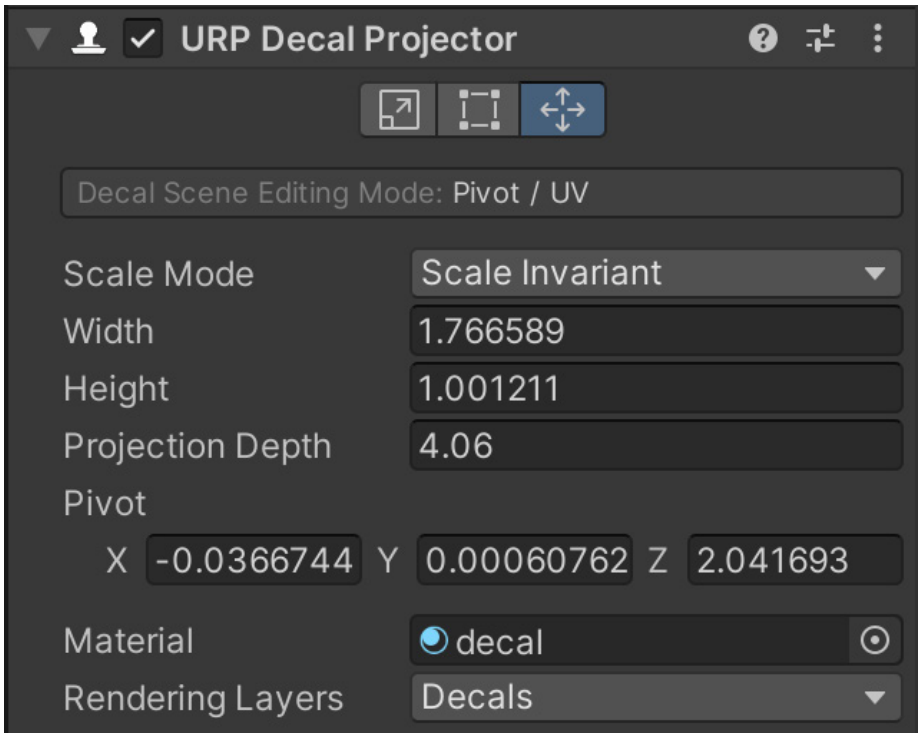
Decal Renderer Feature の追加



ほとんどの目的においては、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。

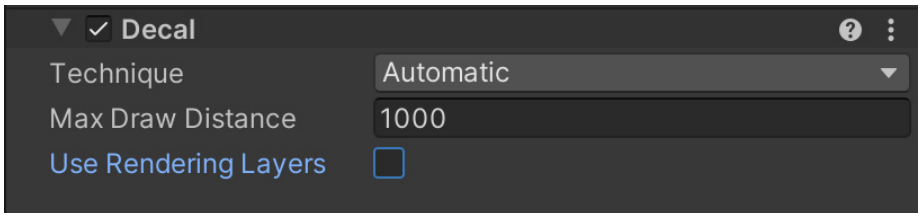
デカールをカスタマイズするには、**「Shader Graph」>「Decal」**シェーダーを使ってマテリアルを作成します。このシェーダーには 3 つの入力、ベースマップ、法線マップ、法線ブレンドがあります。マテリアルの準備ができたら、Decal Projector に割り当てます。



Decal Projector 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。詳細はこちらを参照してください。

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。

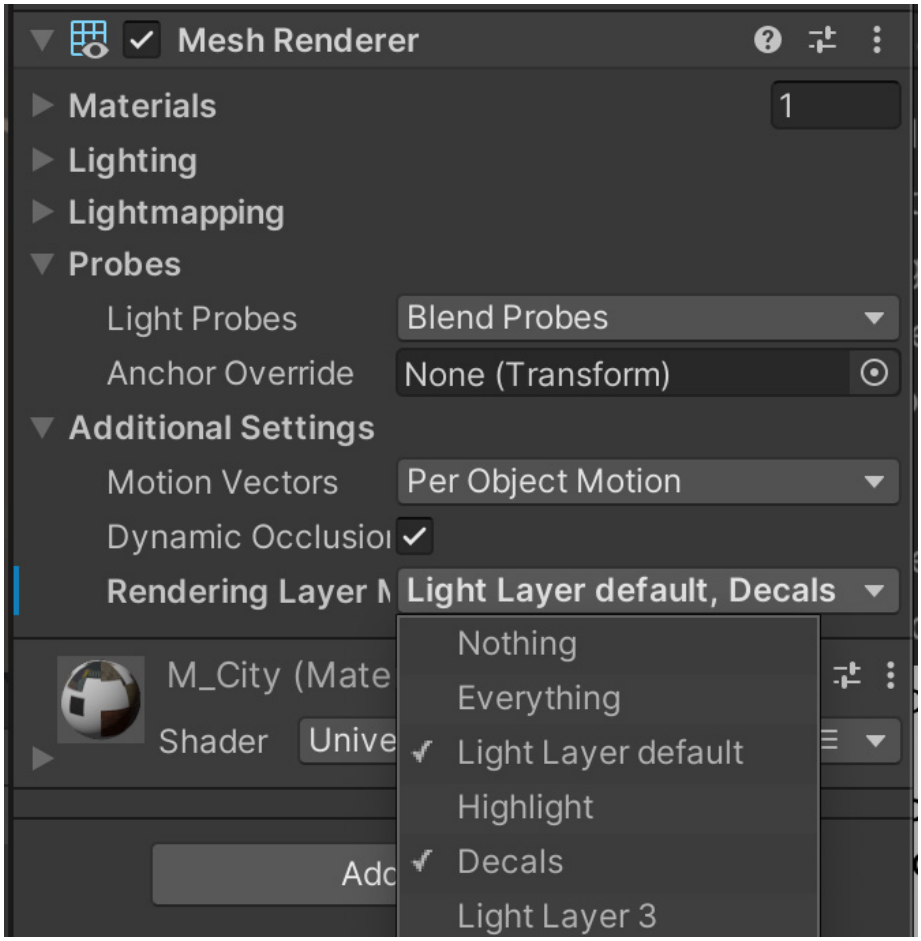


Decal Renderer Feature の設定

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。

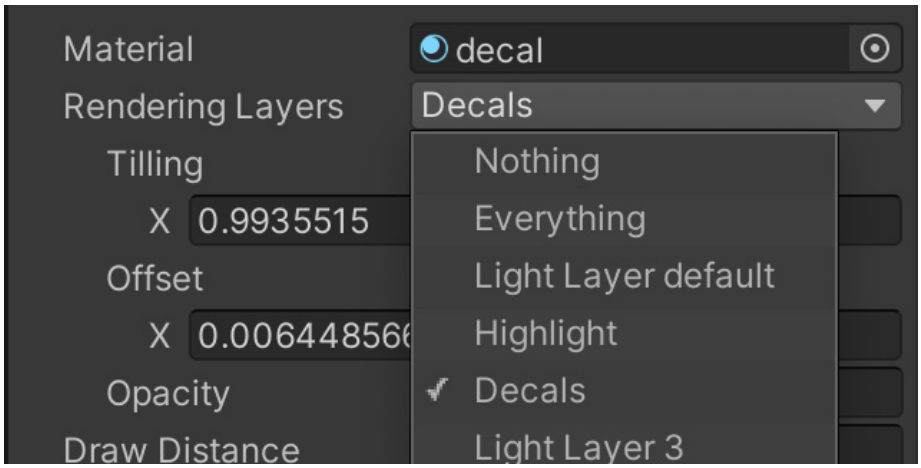
1. 「**Edit**」>「**Project Settings …**」>「**Graphics**」>「**URP Global Settings**」を使用して、レンダリングレイヤーに名前を付けます。

2. プロジェクターを受信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。

左から右に向かって、以下のオプションが表示されます。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

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を比較して、まず注目すべき点は、SubShader タグでキーバリューペア `"RenderPipeline" = "UniversalPipeline"` が使用されていることです。

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。これは、以前の CG（C for Graphics）シェーダープログラミング言語が、HLSL（High Level Shading Language）に置き換えられたことを示していますが、シェーダーの構文と機能はほぼ同じです。Unity が HLSL に移行してからしばらく経つため、すでにご存知かと思いますが、現在では CGPROGRAM / ENDCG マクロの使用は推奨されていません。これらのマクロを使用するということは、UnityCG.cginc を使用するということです。このような形で SRP とビルトインレンダーパイプラインシェーダーライブラリを混在させると、いくつかの問題が発生します。

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。ShaderLab の大部分は、ビルトインレンダーパイプラインから変更されていませんが、ビルトインレンダーパイプライン用に書かれたシェーダーは URP によって自動的に無効化されます。
その理由は、内部ライティングプロセスの変更にあります。ビルトインレンダーパイプラインがオブジェクトに到達するすべてのライトに対して別々のシェーダーパスを実行する（マルチパス）のに対し、URP フォワードレンダラーはライトのループ内のすべてのライティングを 1 つのパスで評価します。この変更により、ライトデータを格納するための異なるデータ構造と、新しい規則を持つ新しいシェーディングライブラリが導入されることになります。

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 ブロックが "RenderPipeline" = "UniversalPipeline" タグを持っていない場合、URP で実行されません。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。すべての SubShader が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をレンダリングします。

SubShader には、複数の Pass ブロックを含められますが、それぞれ特定の LightMode でタグ付けする必要があります。URP はシングルパスフォワードレンダラーを使用するため、GPU にサポートされている「UniversalForward」パスのみが、フォワードレンダリングでオブジェクトのレンダリングに使用されます。

すでに見たように、**「Window」>「Rendering」>「Render Pipeline Converter」**を使用すると、すべてのマテリアルに対して、ビルトインレンダーパイプラインシェーダーから URP シェーダーへの変換が自動で実行されます。では、カスタムシェーダーの場合はどうなるのでしょうか？

カスタムシェーダー

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。以下のリストは、アップグレードプロセスの一環として、古いシェーダーに対して実行する必要があるアクションです。

当社は、カスタム unlit ビルトインシェーダーを URP に変換する方法について、ステップバイステップの動画チュートリアルを作成し、一緒に確認できる Unity プロジェクトも用意しました。

内容

.cginc のインクルードファイルを、以下に示す HLSL 形式の同等のファイルに置き換えます。

ビルトインレンダーパイプライン	HLSL
UnityCG.cginc	Github リンク
AutoLight.cginc	Github リンク Github リンク

その他の便利な HLSL のインクルードファイル

空間変換に関連した関数は、Core.hlsl の使用時にデフォルトで追加されるこのインクルードファイルで定義されています。

HLSL 空間変換関数

URP ヘルパー関数	説明
`float4x4 GetObjectToWorldMatrix()`	オブジェクトからワールド空間への変換を行う UNITY_MATRIX_M 行列を返す ビルトインレンダーパイプラインの `unity_ObjectToWorld` に相当する。
`float4x4 GetWorldToObjectMatrix()`	ワールドからオブジェクト空間への変換を行う UNITY_MATRIX_I_M 行列を返す 返される行列は、UNITY_MATRIX_M の逆行列。 ビルトインレンダーパイプラインの `unity_WorldToObject` に相当する。
`float4x4 GetWorldToHClipMatrix()`	ワールドからクリップスペースへの変換を行う UNITY_MATRIX_VP 行列を返す
`float4x4 GetViewToHClipMatrix()`	ビューからクリップスペースへの変換を行う UNITY_MATRIX_P 行列を返す
`float3 TransformObjectToWorld(float3 positionOS)`	オブジェクト空間での位置を受け取り、ワールド空間での位置を返す
`float3 TransformObjectToWorldDir(float3 dirOS, bool doNormalize = true)`	オブジェクト空間での向きを受け取り、ワールド空間での向きを返す
`float3 TransformWorldToObject(float3 positionWS)`	ワールド空間での位置を受け取り、オブジェクト空間での位置を返す

`float3 TransformWorldToView(float3 positionWS)`	ワールド空間での位置を受け取り、ビュー空間での位置を返す
`real3x3 CreateTangentToWorld(real3 normal, real3 tangent, real flipSign)`	法線と接線を受け取り、接空間からワールド空間への変換を行う行列を生成する
`real3 TransformTangentToWorld(real3 normalTS, real3x3 tangentToWorld, bool doNormalize = false)`	接空間の法線を受け取り、ワールド空間の法線を返す
`real3 TransformWorldToTangent(real3 normalWS, real3x3 tangentToWorld, bool doNormalize = true)`	ワールド空間の法線を受け取り、接空間の法線を返す

変数名の末尾に使用される空間タイプの表記：

- WS:World Space
- TS:接空間
- VS:ビュー空間
- OS:オブジェクト空間

フォグや UV を含むその他のシェーダー関数は、Core.hlsl の使用時にデフォルトで追加されるこのインクルードファイルで定義されています。次の表はその例をいくつか示しています。

URP ヘルパー関数	**説明**
`VertexPositionInputs GetVertexPositionInputs(float3 positionOS)`	オブジェクト空間の位置を受け取り、ワールド、ビュー、クリップ空間の位置を持つ構造体を返す この関数は頂点シェーダーでのみ使用可能。
`VertexNormalInputs GetVertexNormalInputs(float3 normalOS)`	オブジェクト空間の法線を受け取り、ワールド空間の法線、接線、従接線ベクトルの構造体を返す これらのベクトルは、CreateTangentToWorld で、接空間からワールド空間への変換を行う行列を生成するのに使用可能。input. tangentWS、input. bitangentWS、input. normalWS を返す。
`float3 GetCameraPositionWS()`	ワールド空間のカメラ位置を返す ビルトインレンダーパイプラインの _WorldSpaceCameraPos 変数に似ている
`float3 GetViewForwardDir()`	ワールド空間における現在のビューの前方（中心）方向を返す
`float3 GetWorldSpaceViewDir(float3 positionWS)`	ワールド空間ビューの方向（視線方向）を計算する

シェーダーヘルパー関数は、シェーダーコーディングにおいて非常に重要です。時間を節約できるだけでなく、よく使用される計算を高度に最適化した実装です。このインクルードファイルには、以下に関連する多くのヘルパー関数が含まれています。

- プラットフォーム固有の関数
- 一般的な数学関数
- テクスチャユーティリティ
- テクスチャフォーマットサンプリング
- 深度のエンコーディング/デコーディング
- 空間変換
- 地形/ブラシのハイトマップのエンコード/デコードとその他のユーティリティ

その一部を以下の表に列挙します。real 型はファイル内で設定されます。様々なフラグによって、half または float になる可能性があります。

ヘルパー関数	**ヘルパー関数**
`real DegToRad(real deg)`	`real RadToDeg(real rad)`
`bool IsPower2(uint x)`	`real FastACosPos(real inX)`
`real FastASin(real x)`	`real FastATan(real x)`
`uint FastLog2(uint x)`	`real3 Orthonormalize(real3 tangent, real3 normal)`
`real Pow4(real x)`	`float4x4 Inverse(float4x4 m)`
`float ComputeTextureLOD(float2 uv, float bias = 0.0)`	`float Linear01Depth(float depth, float4 zBufferParam)`

プリプロセッサーマクロ

プリプロセッサーマクロは、一般的に使用される便利な手法です。ビルトインレンダーパイプラインシェーダーを新しい URP シェーダーに移植する際、ビルトインレンダーパイプラインのマクロを URP に対応するもので置き換える必要があります。

この表は、その例をいくつか示しています。

ビルトインレンダーパイプライン	**URP**
`UNITY_PROJ_COORD(a)`	Replace with `a.xy/a.w`
`UNITY_INITIALIZE_OUTPUT(type, name)`	`ZERO_INITIALIZE(type, name)`

シャドウマッピング*	
UNITY_DECLARE_SHADOWMAP(tex)	TEXTURE2D_SHADOW_PARAM(textureName, samplerName)**
UNITY_SAMPLE_SHADOW(tex, uv)	SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord3)
UNITY_SAMPLE_SHADOW_PROJ(tex, uv)	SAMPLE_TEXTURE2D_SHADOW(textureName, samplerName, coord4.xyz/coord4.w)
テクスチャ/サンプラー宣言* **	
UNITY_DECLARE_TEX2D(name)	TEXTURE2D(textureName); SAMPLER(samplerName);
UNITY_DECLARE_TEX2D_NOSAMPLER(name)	TEXTURE2D(textureName);
UNITY_SAMPLE_TEX2D_SAMPLER(name,samplername,uv)	SAMPLE_TEXTURE2D(textureName, samplerName, coord2)

表の備考

* シャドウマッピングマクロを使用するにはこのシャドウインクルードファイルが必要です。

** _PARAM は、テクスチャとサンプラー引数を受け取る関数を宣言するのに使用できるマクロです。詳細については、このドキュメントを確認してください。

*** ビルトインレンダーパイプラインのテクスチャ/サンプラー宣言については、このドキュメントを参照してください。

LightMode タグ

LightMode タグは、ライティングパイプラインでの Pass の役割を定義します。ビルトインレンダーパイプラインにおいては、ライティングとの相互作用が必要なシェーダーの大半は、必要な詳細がすべて処理されたサーフェスシェーダーとして記述されます。ただし、ビルトインレンダーパイプラインのカスタムシェーダーは、LightMode タグを使用して、ライティングパイプラインでの Pass の処理方法を指定する必要があります。

以下は、ビルトインレンダーパイプラインで使用される LightMode タグと URP で期待されるタグの対応表です。ビルトインレンダーパイプラインの古いタグ、PrepassBase、PrepassFinal、Vertex、VertexLMRGBM、および VertexLM は、URP ではサポートされていません。また、ビルトインレンダーパイプラインでは対応するものがない URP のタグもあります。

URP のシェーダーを記述する際は、提供されているシェーダーや、それらで使用されているパスを確認するようにしましょう。次のコードサンプルは、Lit シェーダーのコードの一部です。コードのフルバージョンはこちらでご覧ください。

ビルトインレンダーパイプライン（こちらを参照）	**説明**	**URP**（こちらを参照）
いつも	常にレンダリングし、ライティングは適用しない。	-
`ForwardBase`	フォワードレンダリングで使用される。アンビエント、メインディレクショナルライト、頂点/SH ライト、ライトマップが適用される。	`UniversalForward`
`ForwardAdd`	フォワードレンダリングで使用され、ピクセル単位の加算ライトが適用される。ライトごとにパスが記述される。	`UniversalForward`
ディファード	ディファードシェーディングで使用される。G バッファをレンダリング。	`UniversalGBuffer`
`ShadowCaster`	オブジェクトの深度をシャドウマップまたは深度テクスチャにレンダリング。	`ShadowCaster`
`MotionVectors`	オブジェクトごとのモーションベクトルを計算するために使用される。	`MotionVectors`
	URP は、フォワードレンダリングパスでこのタグ値を使用する。パスはオブジェクトのジオメトリをレンダリングし、すべての光の影響を評価する。	`UniversalForwardOnly`
-	URP は、このタグ値を 2D レンダラーで使用する。パスは、オブジェクトをレンダリングし、2D ライトの影響を評価する。	`Universal2D`
-	パスがカメラの視点からの深度情報のみを深度テクスチャにレンダリングします。	`DepthOnly`
-	このパスは、Unity エディターでライトマップをベイクする時にのみ実行される。Unity は、プレイヤーの構築時、このパスをシェーダーから除去する。	`Meta`
-	このタグ値を使用して、オブジェクトのレンダリング時に追加のパスを描画する。フォワードおよびディファードレンダリングパスの両方で使用可能。 URP は、パスに `LightMode` タグがない場合、このタグ値をデフォルト値として使用する。	`SRPDefaultUnlit`

`UniversalForward` と `ShadowCaster` パスは、多くのプラグマと 2 つのインクルードファイルを含んでいます。インクルードファイル内のコードを確認することで、ニーズに合ったカスタムバージョンを作成するのに役立ちます。

```
// フォワードパス。すべてのライトを 1 つのパスでシェーディング。GI + 放出 + フォグ
Pass
{
    // Lightmode は、
    // UniversalRenderPipeline.cs.SRPDefaultUnlit で設定された
ShaderPassName に一致し、
    // LightMode タグがないパスも、ユニバーサルレンダーパイプライン(URP)によりレンダリング
される
    Name "ForwardLit"
    Tags{"LightMode" = "UniversalForward"}

    Blend[_SrcBlend][_DstBlend], [_SrcBlendAlpha][_DstBlendAlpha]
    ZWrite[_ZWrite]
    Cull[_Cull]
    AlphaToMask[_AlphaToMask]

    HLSLPROGRAM
    #pragma exclude_renderers gles gles3 glcore
    #pragma target 4.5
    …

    #pragma vertex LitPassVertex
    #pragma fragment LitPassFragment

    #include "Packages/com.unity.render-pipelines.universal/Shaders/
LitInput.hlsl"
    #include "Packages/com.unity.render-pipelines.universal/Shaders/
LitForwardPass.hlsl"
    ENDHLSL
}

合格
{
    Name "ShadowCaster"
    Tags{"LightMode" = "ShadowCaster"}

    ZWrite On
    ZTest LEqual
    ColorMask 0
    Cull[_Cull]

    HLSLPROGRAM
    #pragma exclude_renderers gles gles3 glcore
     …

    #pragma vertex ShadowPassVertex
    #pragma fragment ShadowPassFragment

    #include "Packages/com.unity.render-pipelines.universal/Shaders/
LitInput.hlsl"
    #include "Packages/com.unity.render-pipelines.universal/Shaders/
ShadowCasterPass.hlsl"
    ENDHLSL
}
```

注意:URP シェーダーの作成を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で、Inspector の **Add Renderer Feature** オプションを使用して、レンダリングプロセスにスクリプトを挿入できます。すでに取り上げたように、URP の使用時は、ユニバーサルレンダラーデータオブジェクトと URP アセットがあります。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。

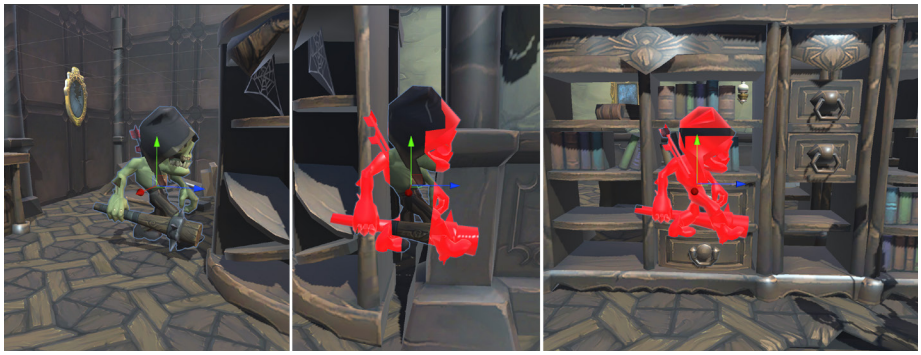
```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

[ExecuteAlways]
public class AutoLoadPipelineAsset :MonoBehaviour
{
        public UniversalRenderPipelineAsset pipelineAsset;

        // 最初のフレームが更新される前に Start を呼び出します
        void OnEnable()
        {
        if (pipelineAsset)
        {
                GraphicsSettings.renderPipelineAsset = 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）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に、キャラクターのシルエットを表示することです。

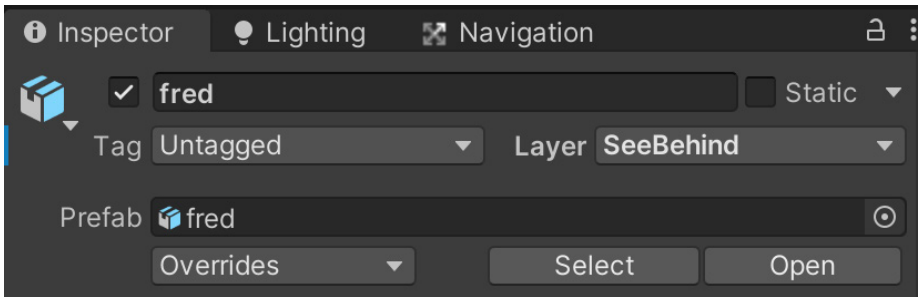


環境モデルによってキャラクターが隠れている時にシルエットを表示する

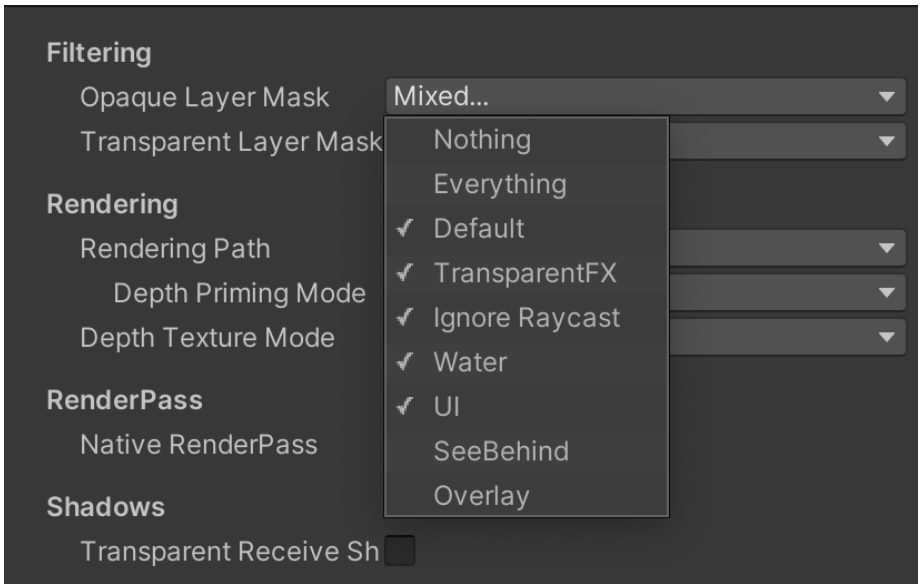
このシルエットの作成方法を説明します。

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。

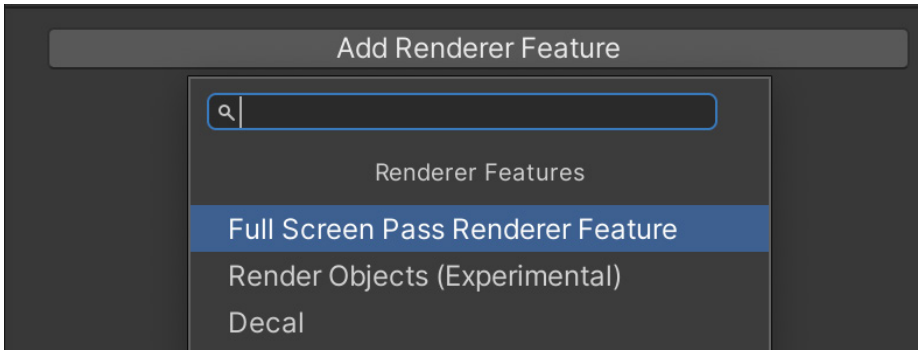
2. キャラクターを必要以上にレンダリングしないように、特別なレイヤーに配置しましょう。キャラクターを選択して、**SeeBehind** レイヤーをレイヤーリストに追加し、キャラクターに対して選択します。



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。

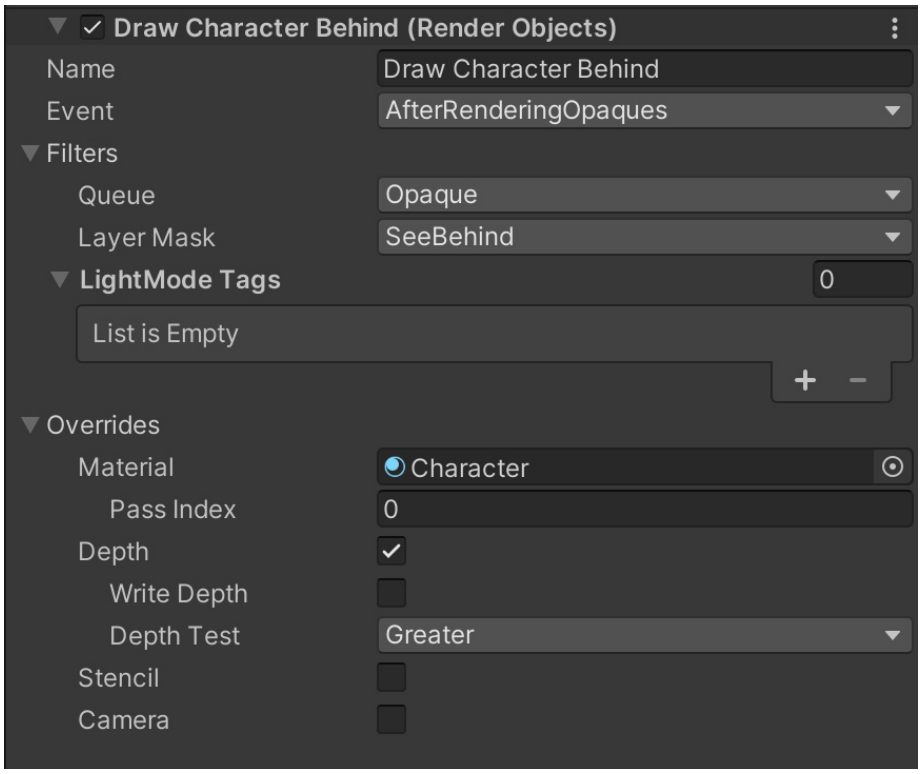


4. 「**Add Renderer Feature**」をクリックし、「**Render Objects (Experimental)**」を選択します。

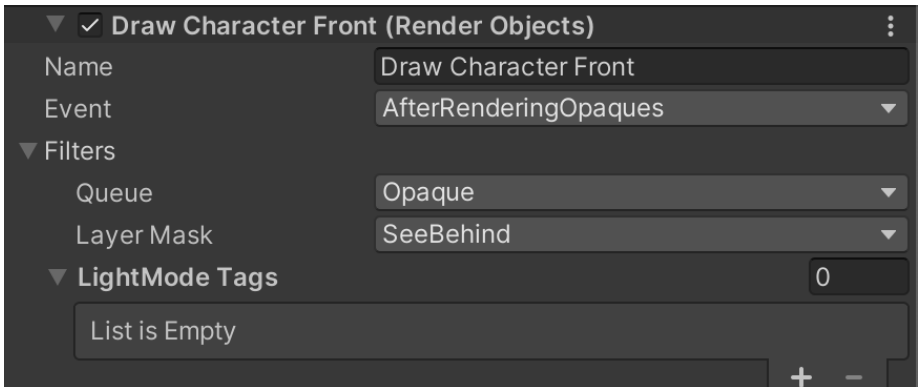


5. このレンダーオブジェクトの**パス**の設定を入力します。名前を付けて、レンダリングがトリガーされるタイミングを選択します。この例では、AfterRenderingOpaques と命名しています。

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んで更新することなく、深度を使用したいと思うでしょう。**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、Render Objects 機能をもう 1 つ追加します。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。



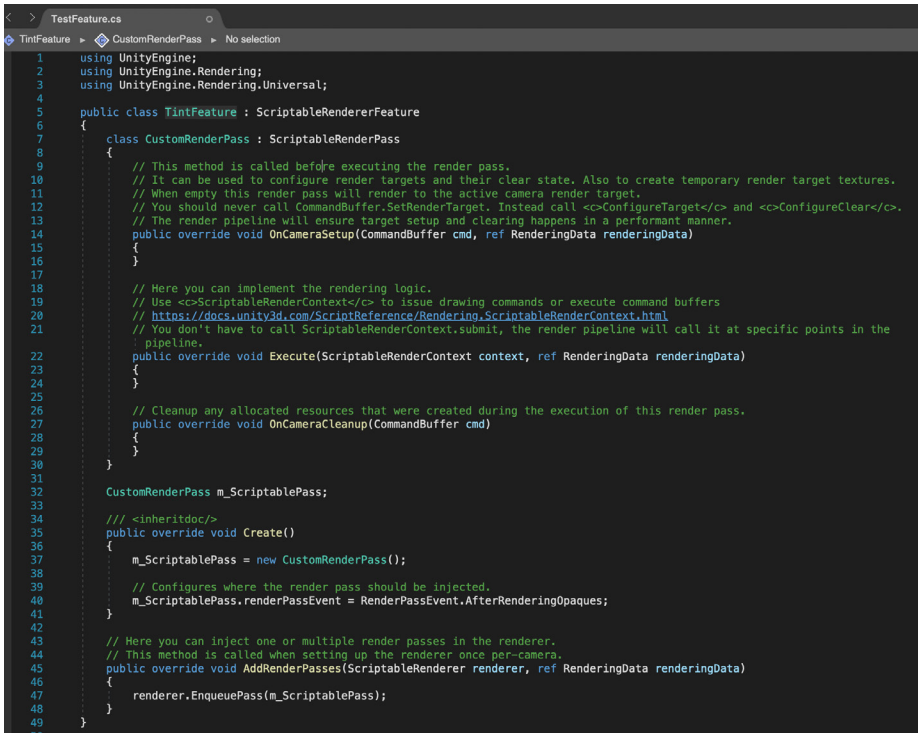
シルエットを使ったテクニックは、コーディングに依存するビルトインレンダーパイプラインのワークフローでは難しい効果を、URP のワークフローを使って追加する良い例です。

Renderer Feature

Renderer Feature は、URP のどの段階でも使用でき、最終的なレンダリング結果に影響を与えます。ポストプロセスエフェクトを追加する簡単な例を見てみましょう。ビルトインレンダーパイプラインを使用するプロジェクトでは、OnRenderImage コールバックを使用して Graphics.Blit を追加する必要があります。この例では、画像の各ピクセルを処理するためにマテリアルを使用するバージョンの関数を使用しています。

1. まず、プロジェクトの 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** という名前を付けます。

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。



Renderer Feature のデフォルトコード

3. **CustomRenderPass** を **TintPass**という名前に変更し、CustomRenderPass クラスにこれらのプロパティを追加します。マテリアルには、レンダリングされた画像の現在の状態に適用するシェーダーが含まれます。

```
Material material;
RTHandle cameraColorTarget;
Color color;
```

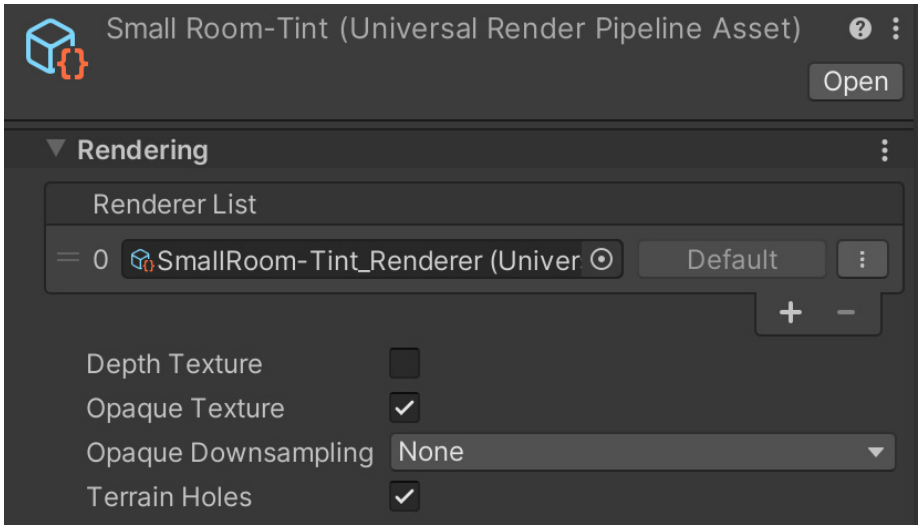
4. TintPass にコンストラクターを追加してマテリアルを初期化し、レンダーパイプラインにおけるこのパスの位置を設定します。

```
public TintPass(Material mat)
{
material = mat;
renderPassEvent = RenderPassEvent.BeforeRenderingPostProcessing;
}
```

5. TintPass クラスの cameraColorTarget と color プロパティを初期化する **SetTarget** メソッドを追加します。

```
public void SetTarget(RTHandle colorHandle, Color col)
{
    cameraColorTarget = colorHandle;
    color = col;
}
```

6. 新しいシェーダーを作成して、**TintBlit** と命名し、以下のコードをコピーします。**RenderPipeline** タグに注目してください。**ZWrite** と **Cull** は、両方ともオフになっています。**Core.hlsl** は **com.unity.render-pipelines.universal** から、**Blit.hlsl** は **com.unity.render-pipelines.core** からインポートされます。URP Asset Inspector で「Opaque Texture」を選択すると、パイプラインはレンダーテクスチャ、_CameraOpaqueTexture を作成します。



URP アセットに対して「Opaque Texture」を選択

シェーダーはこれをサンプリングし、_Color 値を使用して調整します。

```
Shader "Custom/TintBlit"
{
    SubShader
    {
        Tags { "RenderType"="Opaque" "RenderPipeline" =
"UniversalPipeline"}
        LOD 100
        ZWrite Off Cull Off
        Pass
        {
            Name "TintBlitPass"

            HLSLPROGRAM
            #include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Core.hlsl"
            // Blit.hlsl ファイルは頂点シェーダー(Vert)、
            // 入力構造体(Attributes)、出力構造体(Varyings)を提供する
            #include "Packages/com.unity.render-pipelines.core/Runtime/
Utilities/Blit.hlsl"

            #pragma vertex Vert
            #pragma fragment frag

            TEXTURE2D(_CameraOpaqueTexture);
            SAMPLER(sampler_CameraOpaqueTexture);

            float4 _Color;

            half4 frag (Varyings input) :SV_Target
            {
                float4 color = SAMPLE_TEXTURE2D(_CameraOpaqueTexture,
sampler_CameraOpaqueTexture, input.texcoord);
                return color * _Color;
            }
            ENDHLSL
        }
    }
}
```

7. CustomRenderPass m_ScriptablePass を以下のプロパティに置き換えます。
   Shader と Color はレンダラーデータアセットの Inspector で設定できます。

```
public Shader shader;
public Color color;

Material material;

TintPass renderPass = null;
```

8. 次のコードを TintFeature の **Create** メソッドに追加してください。この関数は TintFeature の作成時に呼び出されます。提供されたシェーダーのマテリアル、カスタムコンストラクターを使用した TintPass クラスの新しいインスタンス、および TintBlit シェーダーから作成された新しいマテリアルを初期化するために使用されます。

```
material = CoreUtils.CreateEngineMaterial(shader);
renderPass = new TintPass(material);
```

9. レンダーパスを準備するため、**SetupRenderPasses** オーバーライドを追加する必要があります。Game ビューでのみ色付けを設定したいので、コードは if 文でラップされています。ScriptableRenderPassInput.Color 引数を渡して **ConfigureInput** を呼び出すと、不透明なテクスチャがレンダーパスで利用できるようになります。最後に、先ほど作成した SetTarget メソッドを呼び出して、renderPass に必要な cameraColorTarget と color を持たせます。

```
public override void SetupRenderPasses(ScriptableRenderer renderer,
                                in RenderingData renderingData)
    {
       if (renderingData.cameraData.cameraType == CameraType.Game)
       {
            renderPass.ConfigureInput(ScriptableRenderPassInput.Color);
            renderPass.SetTarget(renderer.cameraColorTargetHandle,
                                 color);
        }
    }
```

10. TintPass のインスタンスを作成して初期化したので、レンダーキューに追加します。次のコードスニペットを **AddRenderPasses** メソッドに追加し、ここでも、if 文の中にコードを記述して、現在のカメラタイプが Game になっているかを判定します。

11. マテリアルが作成されるので、Dispose オーバーライドを追加して破棄します。

```
if (renderingData.cameraData.cameraType == CameraType.Game)
            renderer.EnqueuePass(renderPass);
```

```
    protected override void Dispose(bool disposing)
    {
        CoreUtils.Destroy(material);
    }
```

12. TintPass に戻りましょう。cameraColorTarget を設定する必要があります。次のコードスニペットを **OnCameraSetup** に追加します。

```
  ConfigureTarget(cameraColorTarget);
```

13. これですべての初期化が完了したので、現在のレンダーテクスチャをコピーし、結果を処理するためにマテリアルを使用する実際の作業を行うことができます。以下のコードを **Execute** メソッドに追加します。

```
var cameraData = renderingData.cameraData;
if (cameraData.camera.cameraType != CameraType.Game)
                return;

if (material == null)
                return;

CommandBuffer cmd = CommandBufferPool.Get();

material.SetColor("_Color", color);
Blit(cmd, cameraColorTarget, cameraColorTarget, material, 0);

context.ExecuteCommandBuffer(cmd);
cmd.Clear();

CommandBufferPool.Release(cmd);
```

14. 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。

15. **TintFeature** コードのフルバージョンと、その下に最終的な結果を載せます。

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

public class TintFeature :ScriptableRendererFeature
{
    class TintPass :ScriptableRenderPass
    {
        Material material;
        RTHandle cameraColorTarget;
        Color color;

        public TintPass(Material mat)
        {
            material = mat;
            renderPassEvent = RenderPassEvent.
BeforeRenderingPostProcessing;
        }

        public void SetTarget(RTHandle colorHandle, Color col)
```

```
        {
            cameraColorTarget = colorHandle;
            color = col;
        }

        public override void OnCameraSetup(CommandBuffer cmd, ref
RenderingData renderingData)
        {
            ConfigureTarget(cameraColorTarget);
        }

        public override void Execute(ScriptableRenderContext context, ref
RenderingData renderingData)
        {
            var cameraData = renderingData.cameraData;
            if (cameraData.camera.cameraType != CameraType.Game)
                return;

            if (material == null)
                return;

            CommandBuffer cmd = CommandBufferPool.Get();

            material.SetColor("_Color", color);
            Blit(cmd, cameraColorTarget, cameraColorTarget, material, 0);

            context.ExecuteCommandBuffer(cmd);
            cmd.Clear();

            CommandBufferPool.Release(cmd);
        }
    }

    public Shader shader;
    public Color color;

    Material material;

    TintPass renderPass = null;

    public override void Create()
    {
        material = CoreUtils.CreateEngineMaterial(shader);
        renderPass = new TintPass(material);
    }

    public override void SetupRenderPasses(ScriptableRenderer renderer,
                                           in RenderingData renderingData)
    {
        if (renderingData.cameraData.cameraType == CameraType.Game)
        {
            // ScriptableRenderPassInput.Color 引数を渡して ConfigureInput
を呼び出す
            // これでレンダーパスで不透明なテクスチャが利用可能になる
            renderPass.ConfigureInput(ScriptableRenderPassInput.Color);
            renderPass.SetTarget(renderer.cameraColorTargetHandle,
color);
        }
    }

    public override void AddRenderPasses(ScriptableRenderer renderer,
```

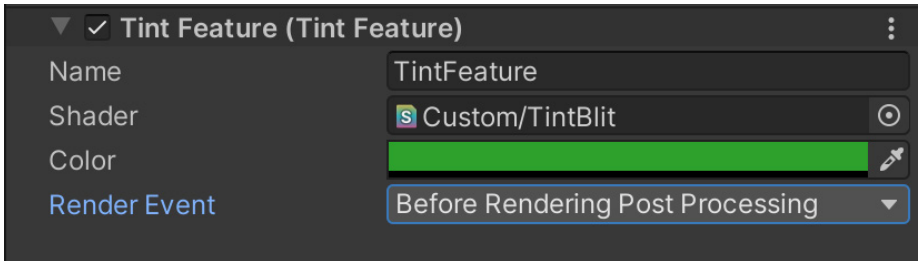
```
                                    ref RenderingData renderingData)
    {
        if (renderingData.cameraData.cameraType == CameraType.Game)
            renderer.EnqueuePass(renderPass);
    }

    protected override void Dispose(bool disposing)
    {
        CoreUtils.Destroy(material);
    }
}

;
```

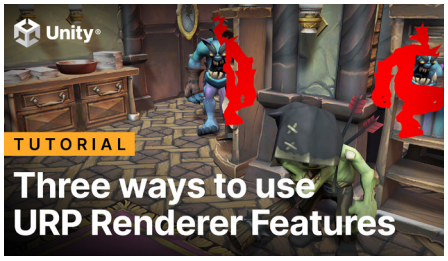
TintFeature の効果：加工されていない部分（左）と着色された部分（右）

ユーザーにレンダーパイプラインのどの段階でこれを使用するか選ばせる場合は、以下に示す通り、追加のプロパティを使用できます。

```
public RenderPassEvent renderEvent;
…
// メソッドを作成
renderPass = new TintPass(material, renderEvent);
…
//TintPass コンストラクター
public TintPass(Material mat, RenderPassEvent renderEvent)
{
   material = mat;
   renderPassEvent = renderEvent;
}
```

レンダラーデータアセットを使用して、Inspector でプロパティを割り当てます。





この動画チュートリアルでは、Renderer Feature を使用した 3 つの実践的な演習を紹介します。具体的には、カスタムポストプロセスエフェクト、ステンシルエフェクト、環境によってオクルードされたキャラクターの作成方法を扱います。

Renderer Feature のベストプラクティスについては、Ned Makes Games によるカスタム Renderer Feature の制御方法に関する動画チュートリアルなど、コミュニティ主導の例をさらにご覧いただけます。

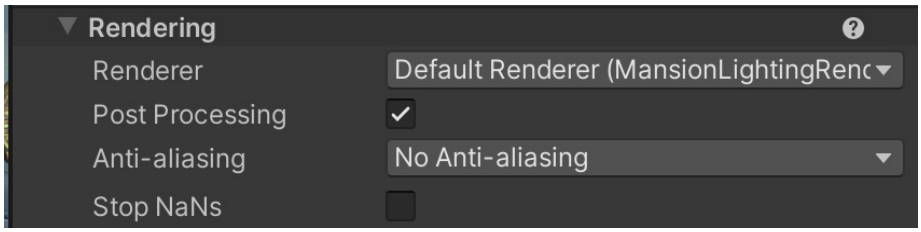
ポストプロセッシング

Built-in Post-Processing Stack v2 パッケージは URP と互換性がありません。
URP はポストプロセスエフェクトのための追加パッケージを必要としません。代わりに、ボリュームフレームワークを使用します。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ボリュームに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場所を問わず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。

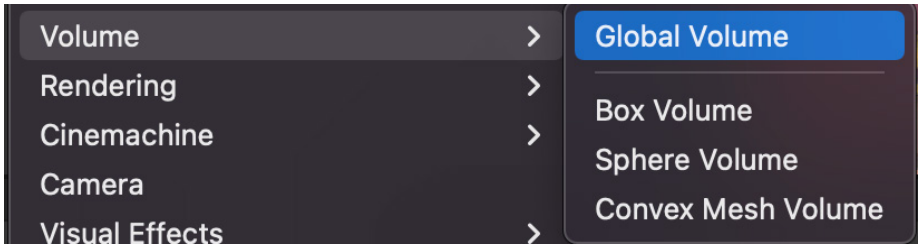
ポストプロセスエフェクトの適用：左上の画像には効果は適用されておらず、右上の画像には Bloom、左下の画像には Vignette が適用され、右下の画像には Color Adjustment が追加されています。

URP ポストプロセスフレームワークの使用

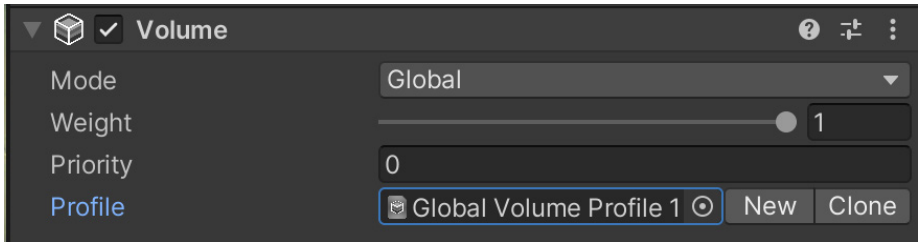
1. 最初のステップは、メインカメラのポストプロセスを有効化することです。「**Hierarchy**」ウィンドウで「**Main Camera**」を選択し、「**Inspector**」に移動した後、「**Rendering**」パネルを展開します。**Post Processing** オプションにチェックを入れます。



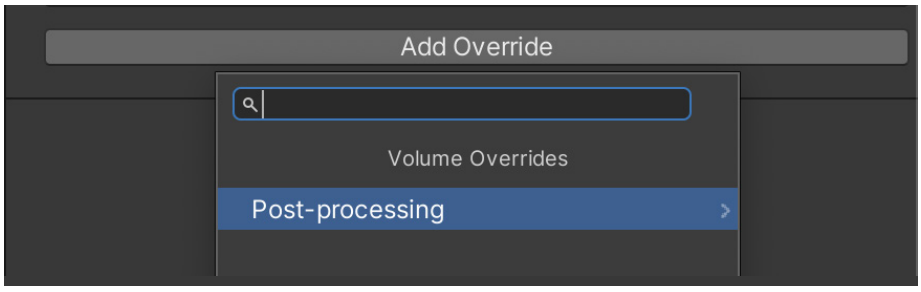
2. 「**Hierarchy**」ウィンドウを右クリックし、「**Create**」>「**Volume**」>「**Global Volume**」を選択して、グローバルボリュームを作成します。

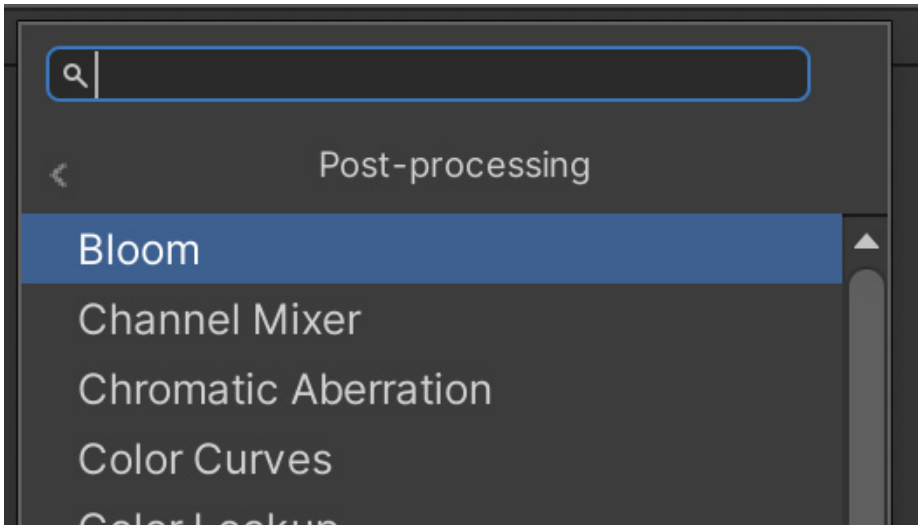


3. 「Hierarchy」ウィンドウのグローバルボリュームが選択された状態で、Inspector 内の「**Volume**」パネルから、「**New**」をクリックして新しい**プロファイル**を作成します。



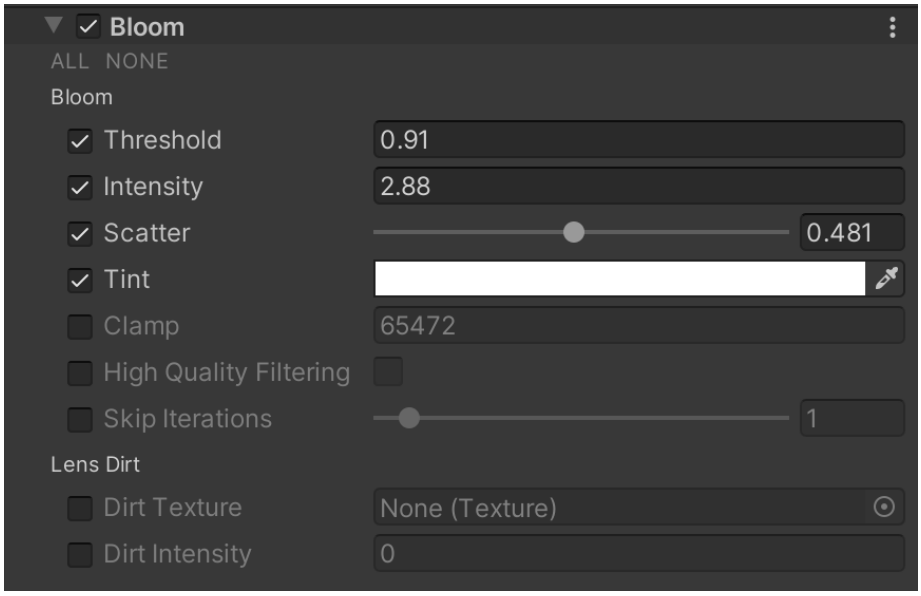
4. ポストプロセスエフェクトを追加し始めます。下の方にある利用可能な効果をリスト化した表を確認してください。「**Add Override**」をクリックし、「**Post-processing**」を選択します。この例では、「Bloom」エフェクトが選択されています。



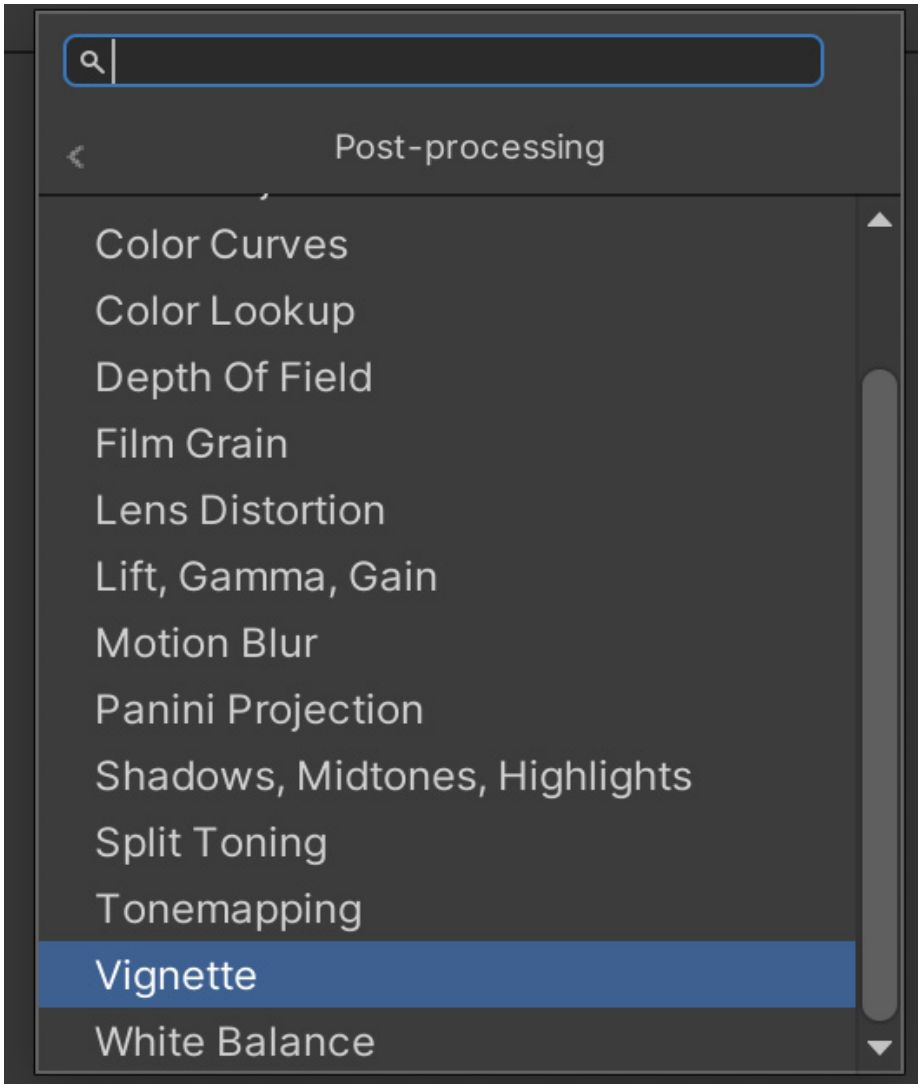


「Bloom」エフェクトを選択

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。



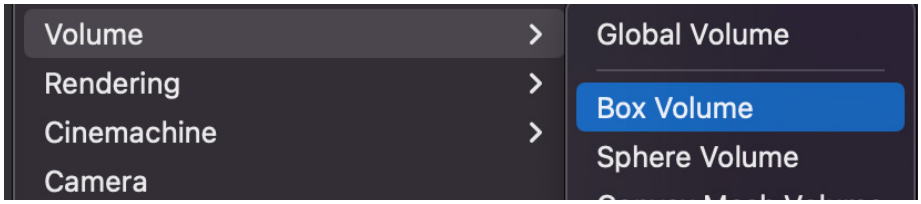
6. 複数の効果（この例では Vignette など）を簡単に追加し、それぞれの設定パネルを使って設定することができます。



ローカルボリュームの追加

ボリュームフレームワークを使用すると、シーンを設定し、カメラのシーン移動に合わせて、異なるポストプロセスプロファイルをトリガーするように設定できます。これは、ローカルボリューム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の形状をより厳密に制御する場合は「**Convex Mesh Volume**」を選択します。

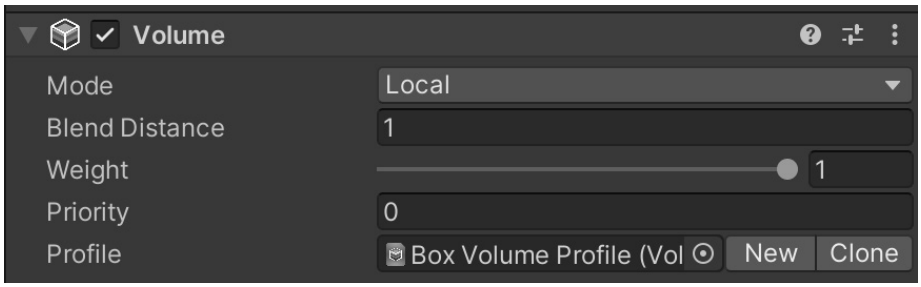


2. **Inspector** の「**Volume**」パネルから、このボリュームデータを保存する新しい**プロファイル**を作成します。このパネルでは、以下を設定することもできます。

   a. **Blend Distance**：これは、URP がブレンディングを開始するボリュームのコライダーから最も遠い距離と、このプロファイルがフェードインするコライダー寸法での距離です。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、シーンへの影響の大きさが等しい複数のボリュームがある場合に、どのボリュームを使用するか決めるために、この値が使用されます。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、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。

これらは 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
Bloom	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。
Channel Mixer	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。
Chromatic Aberration	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。
Color Adjustments	Color Adjustments（色調整）エフェクトを使用すると、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。
Color Curves	グレーディングカーブは色相、彩度、明るさの特定の範囲を調整する高度な方法です。
Color Lookup	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。
Depth of Field	この効果は、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。
Film Grain	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。
Lens Distortion	レンダリングされた最終的な絵を歪めることで、現実のカメラレンズの形をシミュレートします。
Lift Gamma Gain	各種のトラックボールを使用して、画像内の様々な範囲に影響を与えることができます。トラックボールの下にあるスライダーを調整して、その範囲の色の明るさをオフセットします。
Motion Blur	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画像に生じるぼやけをシミュレートします。
Panini Projection	この効果は、有効視野（FOV）が非常に広いシーンで透視図をレンダリングするのに役立ちます。
Shadows Midtones Highlights	レンダリング対象のシャドウ、中間調、ハイライトそれぞれを制御できます。
Split Toning	これを使用して、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。
Tonemapping	画像の HDR 値を新しい範囲の値に再マップするプロセスです。
Vignette	この効果は、画像の中央に比べて端が暗くなる効果を含みます。
White Balance	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。

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 を呼び出します
        void Start()
        {
        Volume volume = GetComponent<Volume>();
        Bloom bloom;
        if (volume.profile.TryGet<Bloom>(out bloom))
        {
                bloom.intensity.value = 0;
        }
    }
}
```

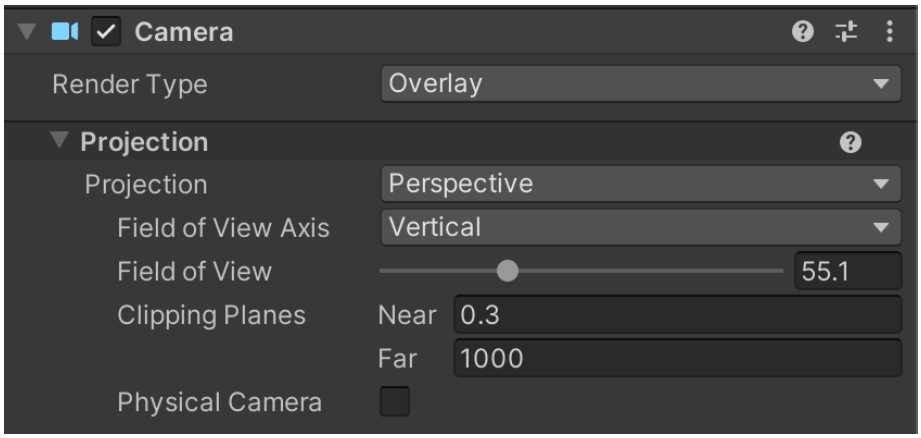
Camera Stacking

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。上の画像では、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。この機能の設定方法を見てみましょう。

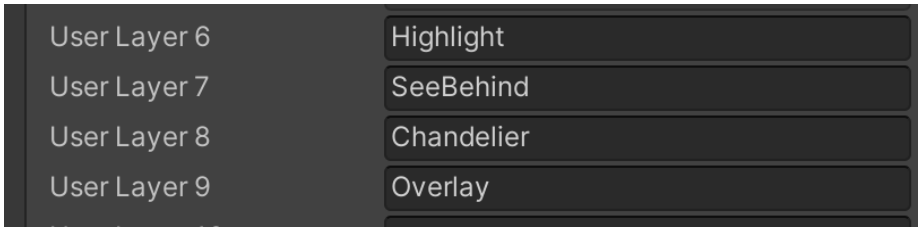
Camera Stacking を使用した例

1. **Hierarchy** ビューを右クリックして**「Create」>「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。

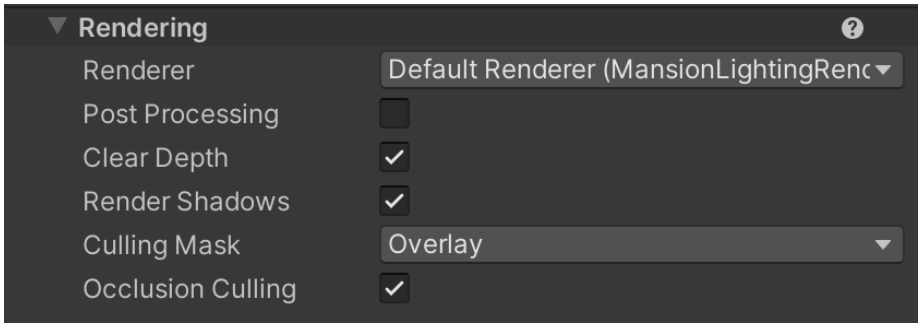
2. **「Inspector」>「Camera Settings」**パネルを使用して、このカメラを **Render Type Overlay** に設定します。



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。



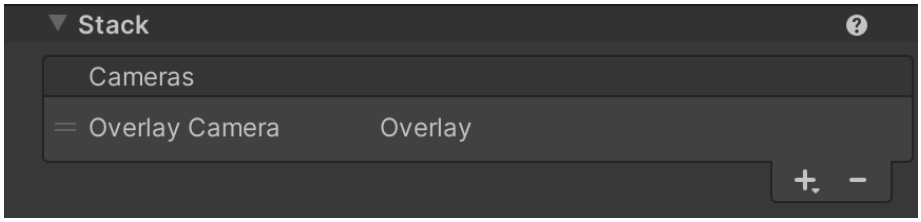
4. Inspector を使用して、カメラの**「Rendering」>「Culling Mask」**を更新します。



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 を呼び出します
        void Start()
        {
        Camera camera = GetComponent<Camera>();
        var cameraData = camera.GetUniversalAdditionalCameraData();
        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、このような目的を念頭に置いて設計されています。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。**Start** コールバックで**レンダーテクスチャ**を作成します。1920 x 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

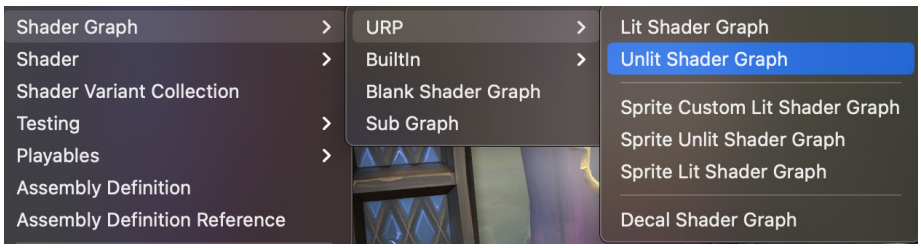
URP を使用するもう 1 つの利点は、複雑な作成作業をテクニカルアーティストが利用できるようにする Unity の最新のオーサリングツールとの互換性です。この章では、Shader Graph を使用してシェーダーを作成する方法と、ビジュアルエフェクトを使用してパーティクルエフェクトを作成する方法について説明します。

Shader Graph

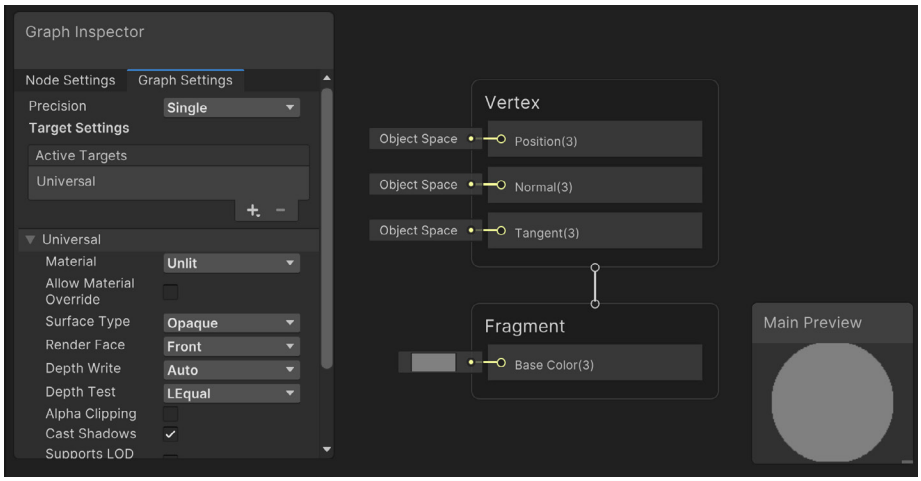
Shader Graph は、アーティストのワークフローにカスタムシェーダーを導入します。Shader Graph ツールは、URP テンプレートを使用してプロジェクトを開始するか、URP パッケージをインポートする際に含まれます。

Shader Graph については、また別の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、基本的で重要なステップを説明します。

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、**「Create」>「Shader Graph」>「URP」>「Unlit Shader Graph」**を選択します。この例では「Unlit」を選択します。新しいアセットを FresnelAlpha と名付けます。

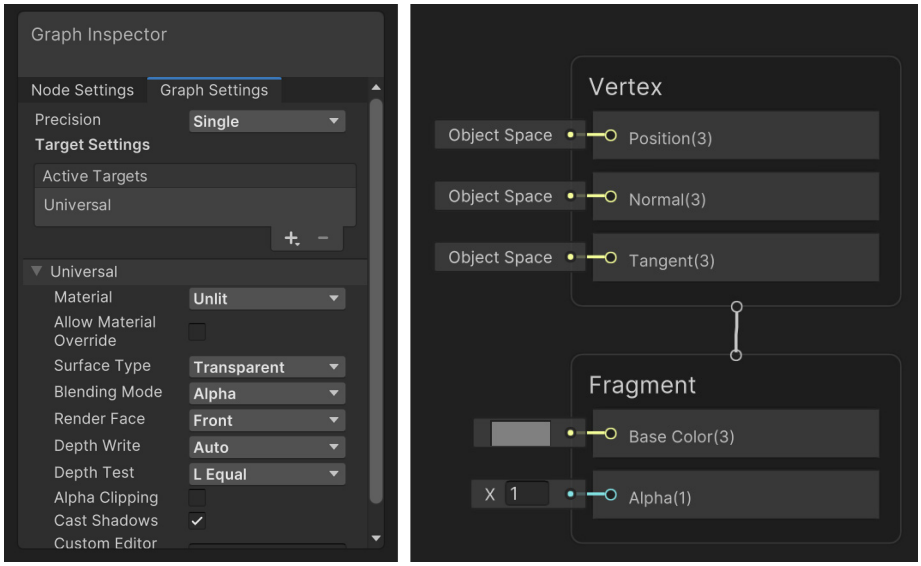


2. 新しい **Shader Graph アセット**をダブルクリックし、Shader Graph エディターを起動します。

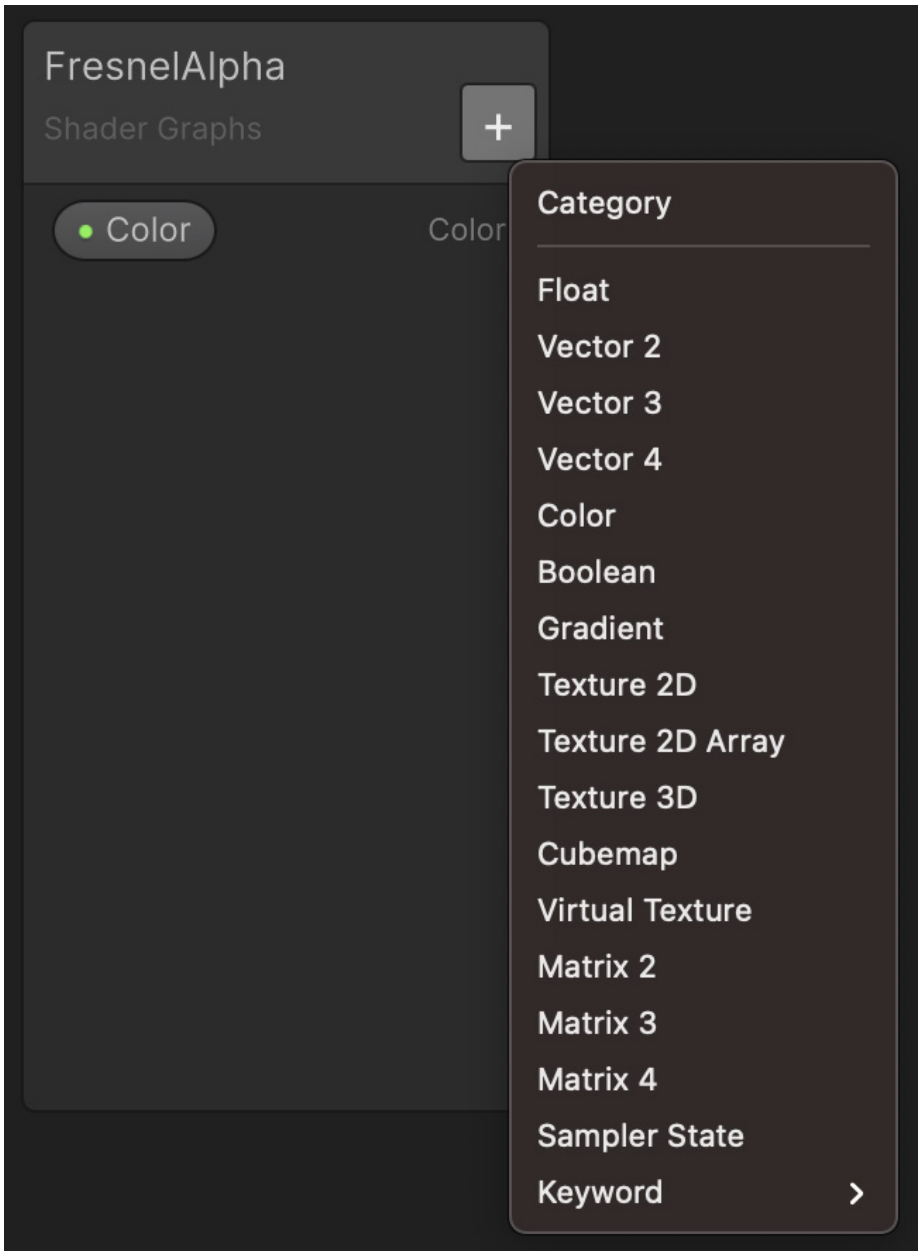


シェーダーを触った経験があれば、Vertex ノードと Fragment ノードについては知っているでしょう。デフォルトでは、このシェーダーは、マテリアルを使用するモデルが Vertex ノードを使用して Camera ビューに正しく配置され、Fragment ノードを使用して各ピクセルがグレーカラーに設定されるようにします。

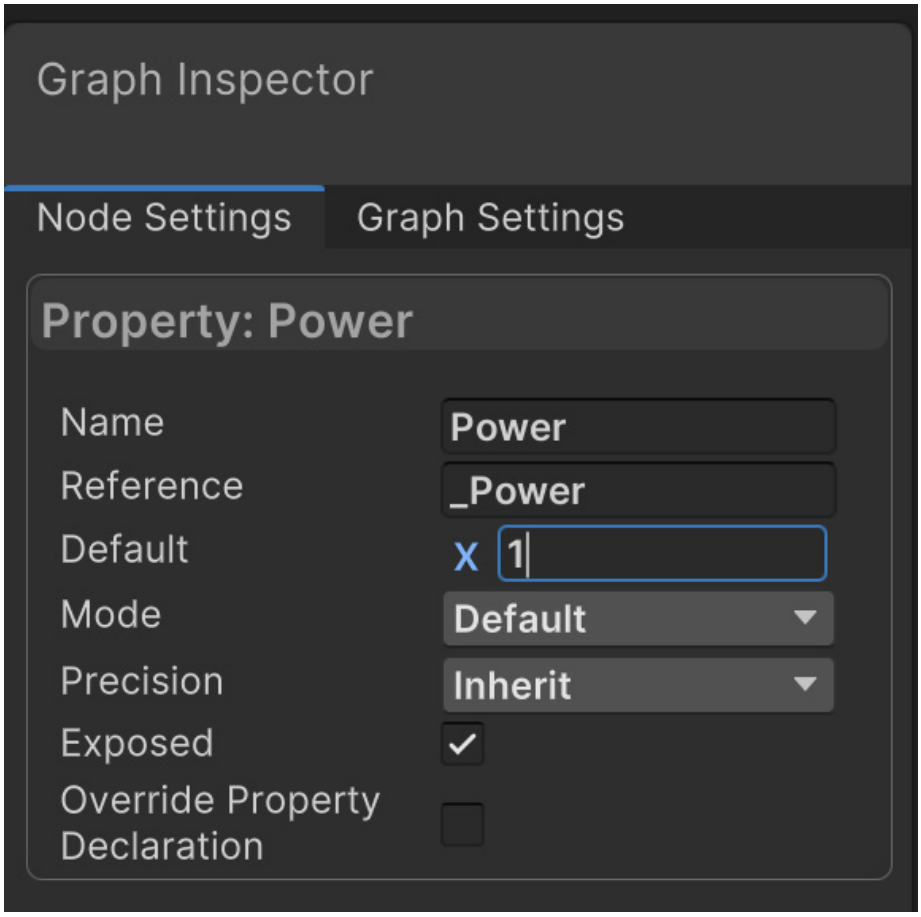
3. このシェーダーはオブジェクトのアルファ透明度を設定します。したがって、Transparent キューに適用する必要があります。**「Graph Inspector」>「Graph Settings」>「Surface Type」**を **Transparent** に変更します。Fragment ノードに Base Color だけでなく Alpha 入力があることがわかります。



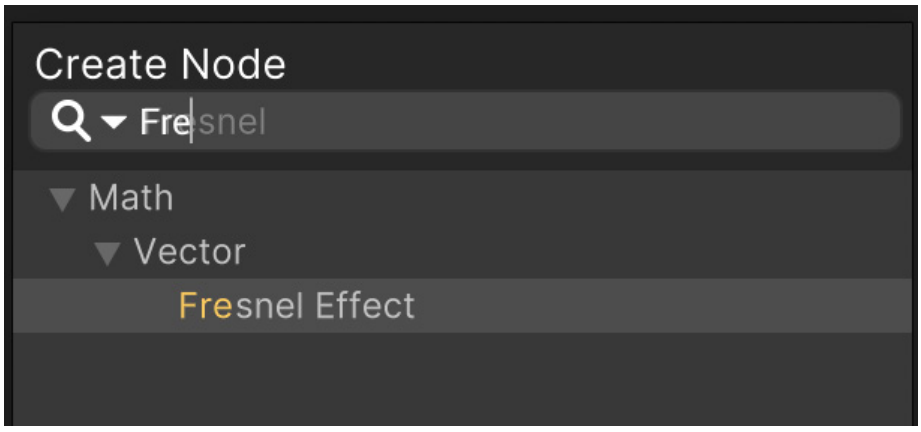
4. シェーダーにプロパティを追加します。例えば、Color を Color として追加し、Power と Strength を Float 値として追加します。



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。



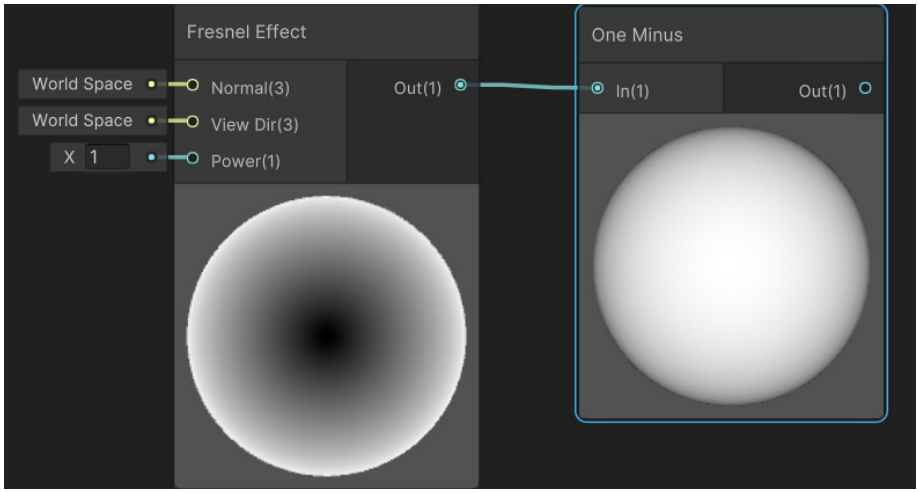
6. Shader Graph は、ノードを結合することで機能します。各ノードは、1 つ以上の入力と出力を持ちます。ノードを追加するには、上部の「**Search**」パネルで右クリックし、「**Create Node**」を選択して「**Fre**」と入力します。**Fresnel Effect** ノードに、その結果が表示されます。



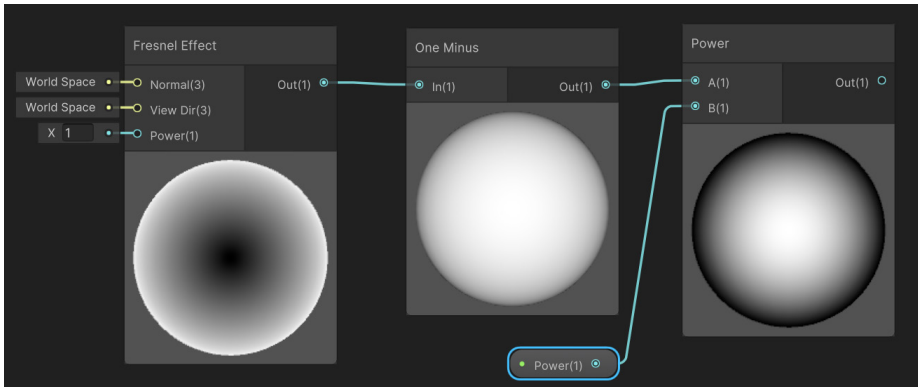
7. ノードに、効果のプレビューが表示されます。Fresnel エフェクトは、端に向かって明るくなることが分かります。値は、ビュー方向と法線方向の差であり、スフィアの場合、端でその差が最大となっています。

   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。

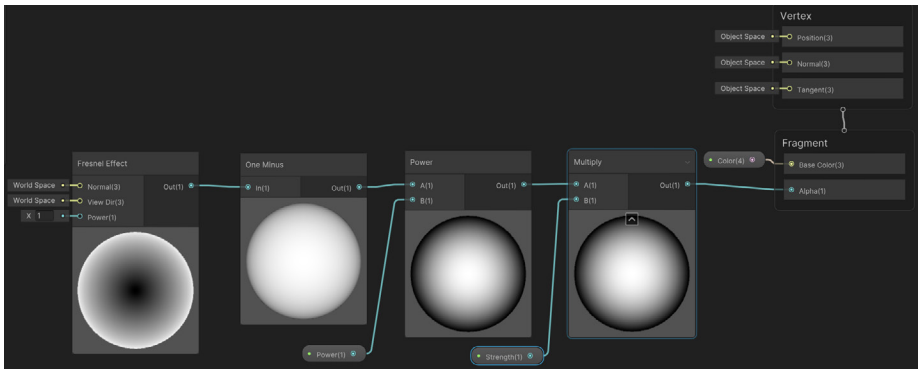


8. グラデーションのサイズと全体の透明度を制御する方法を見ていきましょう。グラデーションのサイズを変更するには、**Power** ノードを使用します。Power ノードを作成し、One Minus Out(1) を Power A(1) に接続します。Power プロパティをグラフにドラッグし、Power B(1) に接続します。この操作を完了すると、グラフは次のようになるはずです。

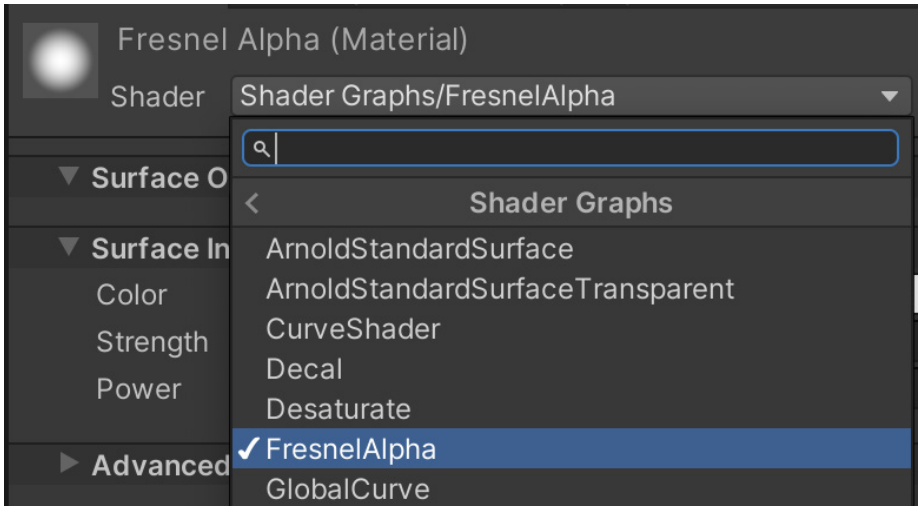


9. **Multiply** ノードを使用して、全体の透明度を制御します。Multiply ノードを作成した後、Power Out(1) を Multiply A(1) に接続します。Strength プロパティをグラフにドラッグし、Multiply B(1) に接続します。次に、Multiply Out(1) と Fragment Alpha(1) を結合し、Color(4) プロパティをグラフにドラッグして、Fragment Base Color(3) に接続します。

   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。

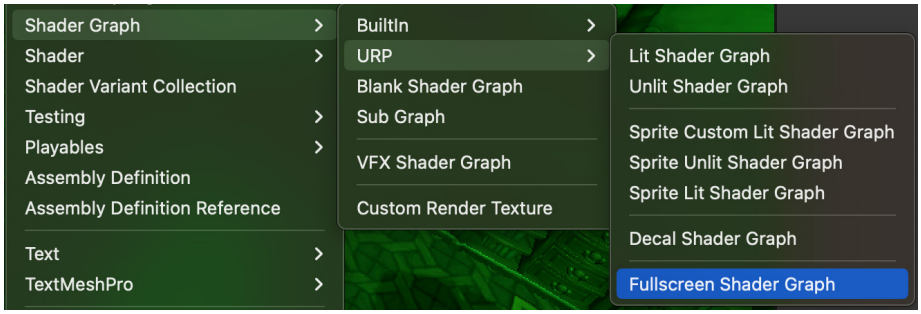


11. これで、マテリアルをオブジェクトに適用し、端の可視性を制御できるようになりました。

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

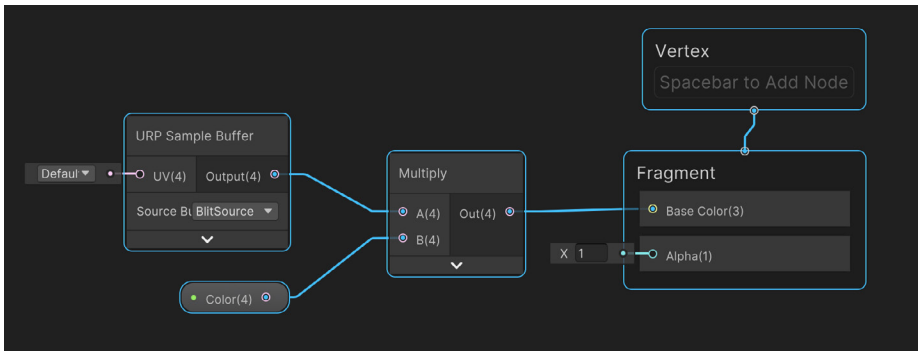
全画面 Shader Graph

全画面 Shader Graph は、Unity URP 2022 LTS で新たに導入されました。これは、カスタムポストプロセスパスの作成を可能にします。「Project」ウィンドウを右クリックし、**「Create」>「Shader Graph」>「URP」>「Fullscreen Shader Graph」**を選択します。



全画面 Shader Graph の作成

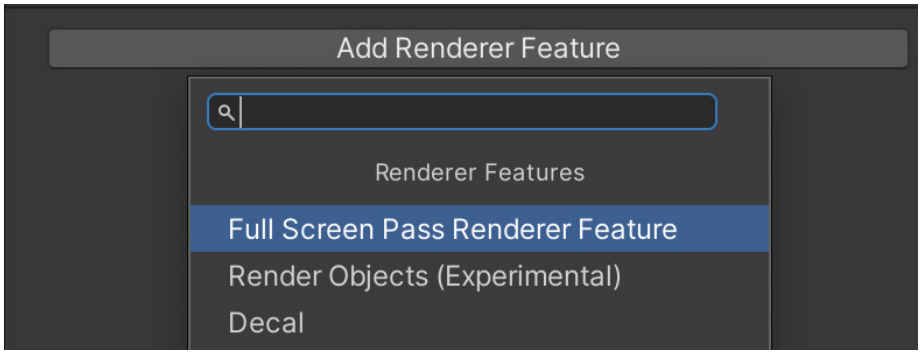
BlitSource オプションを使用する URP Sample Buffer ノードを使用して、フラグメントシェーダーのピクセルの色にアクセスすることができます。以下のグラフは、シンプルな色付けの例です。また、URP Sample Buffer は、エッジ検出やモーショントレイルに役立つワールド法線とモーションベクトルを利用可能にします。



簡単な色付けの例

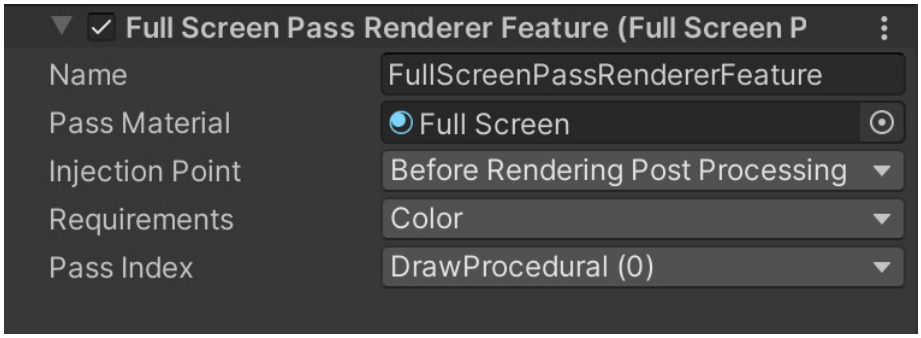
この例を使用するには、このシェーダーを使用するマテリアルを使用して、現在のカメラのレンダーテクスチャの結果を Blit する方法が必要です。

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

関連リンク:

— このブログポストでは、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。

— Unity のウェブサイトで Shader Graph のページをチェックしてください。

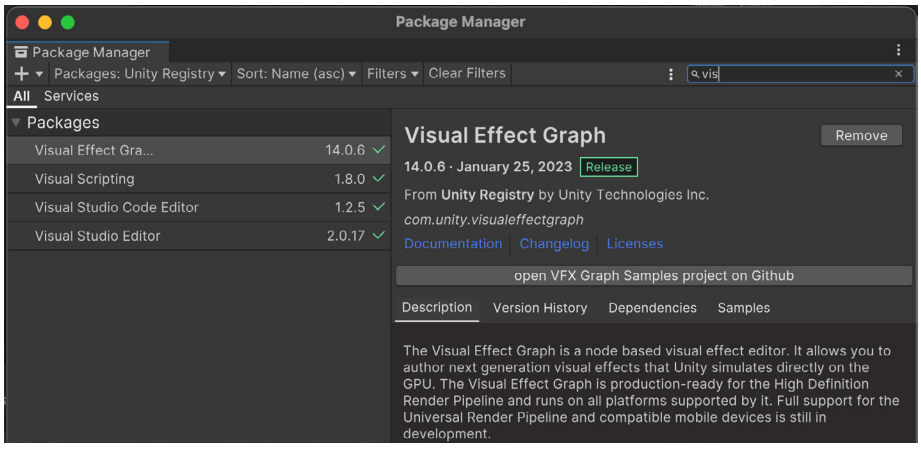
Visual Effect Graph

Visual Effect (VFX) Graph を使用すると、アーティストにとって使いやすいノードベースのグラフで、無数のパーティクルエフェクトを作成できます。VFX Graph を使用して、火、煙、霧、火花、魔法のオーブ、その他多くの効果をプロジェクトに追加できます。

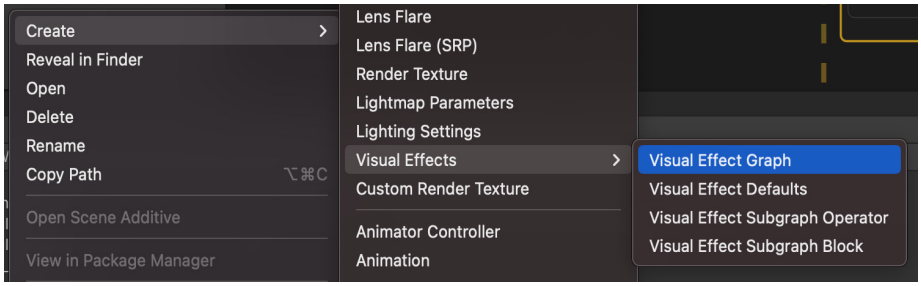
VFX Graph は GPU 上で動作するコンピュートシェーダーを使用して最高のパフォーマンスを実現するため、VFX Graph で作成された効果を使用しているゲームは、コンピュート処理をサポートしているデバイスを対象にする必要があります。コードをテストし、コンピュート非対応デバイス用のフォールバック処理を含め、ローエンドのモバイルデバイスをターゲットとするゲームでは、VFX Graph をあまり使用しないようにしてください。

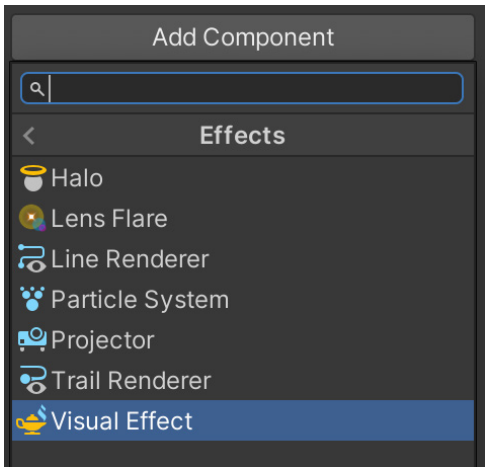
VFX Graph についての理解を深めるため、煙エフェクトを作成するステップを見ていきましょう。

1. VFX Graph は、**Package Manager** からパッケージとしてダウンロード可能です。



2. VFX Graph のインストール後、**「Project window」>「Assets」**フォルダー内で右クリックすると、新しいオプションが表示されるようになります。**「Create」>「Visual Effects」>「Visual Effect Graph」**を選択し、新しいアセットに Smoke という名前を付けましょう。

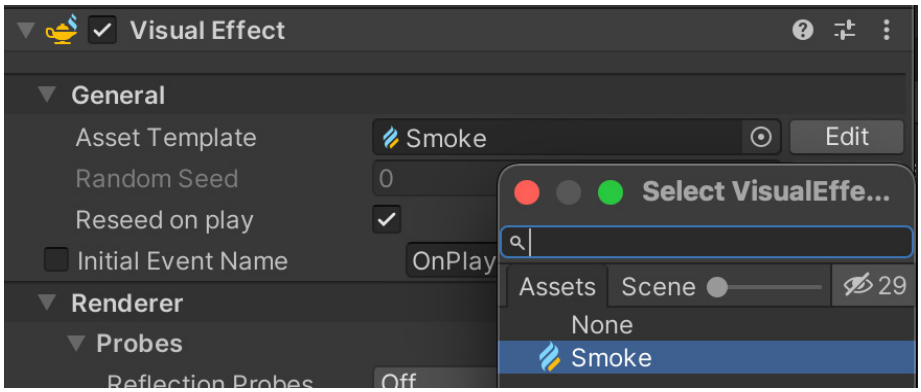




3. 空の**ゲームオブジェクト**を作成し、「Hierarchy」ウィンドウで選択します。**Inspector** で、**「Add Component」>「Effects」>「Visual Effect」**を選択します。

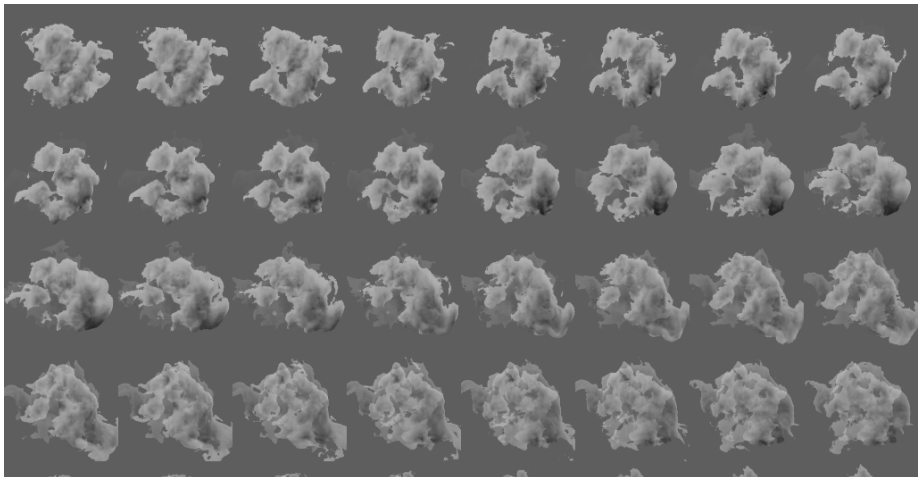
または、Visual Effect Graph Asset をエディター内の **Hierarchy** ビューに追加することもできます。これでコンポーネントがアセットと一緒に追加され、ステップ 3 と 4 を省略できます。

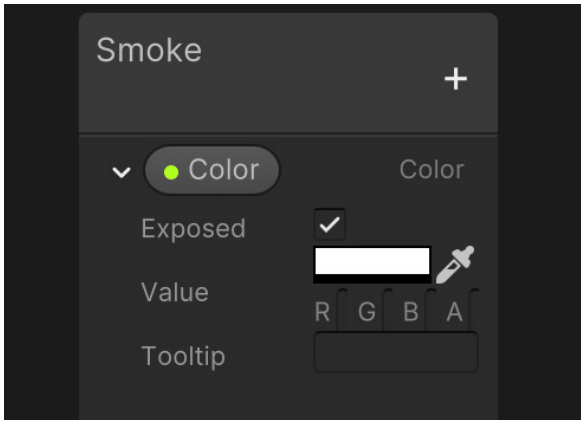
4. 「**Component Settings**」パネルを使用して、**Smoke VFX Graph** を **Asset Template** として選択します。



5. これで、VFX Graph を編集できるようになります。ダブルクリックして、「Visual Effect Graph 」ウィンドウを開きます。そこには、Spawn、Initialize、Update、Output Context ノードが事前ロードされています。

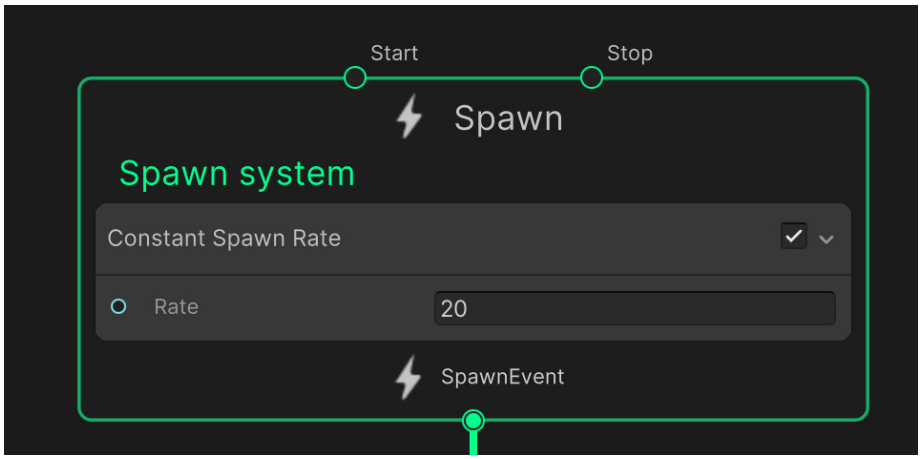
アニメートされたスモークスプライトを含むアトラス形式のテクスチャを使用します。8x8 のグリッドに配置された 64 枚の画像が、個々のパーティクルのソースとして機能します。どのフレームにおいても、1 つのパーティクルはグリッドから 1 枚の画像だけを表示します。各フレームがレンダリングされるたびに、事前に定義された速度で画像を切り替えます。下の画像は、Smoke Sprite のアトラスです。



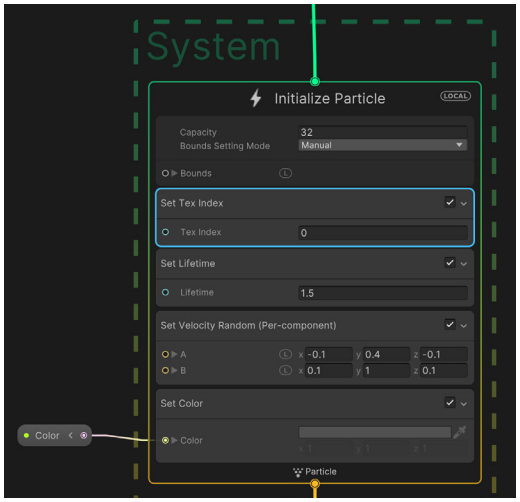


6. 「+」ボタンをクリックして、**Color** プロパティを追加します。これで、ユーザーが Inspector から煙の色を変更できるようになります。

7. Spawn ブロックを見てみましょう。デフォルトの Spawn ブロックには、**Constant Spawn Rate** ノードが配置されています。これを 20 に設定します。

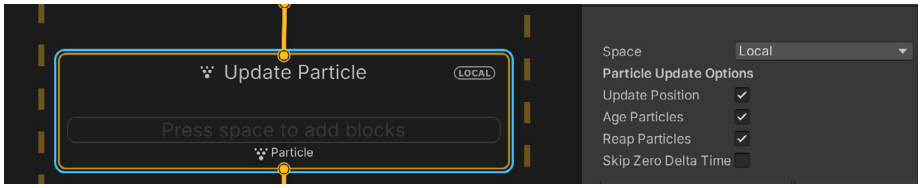


8. 次の Initialize ブロックでは、パーティクルが最初に作成されたときの処理方法を定義します。**Set Lifetime Random** ノードを削除します。次に、**Set Tex Index** を追加して 0 から 63 のランダムな値に設定し、各スモークパーティクルに異なる外観を与えます。これは重要です。なぜなら、パーティクルは先ほどの Smoke Sprite シートから画像を表示するため、最初のインデックスを 0 にしておきたいからです。

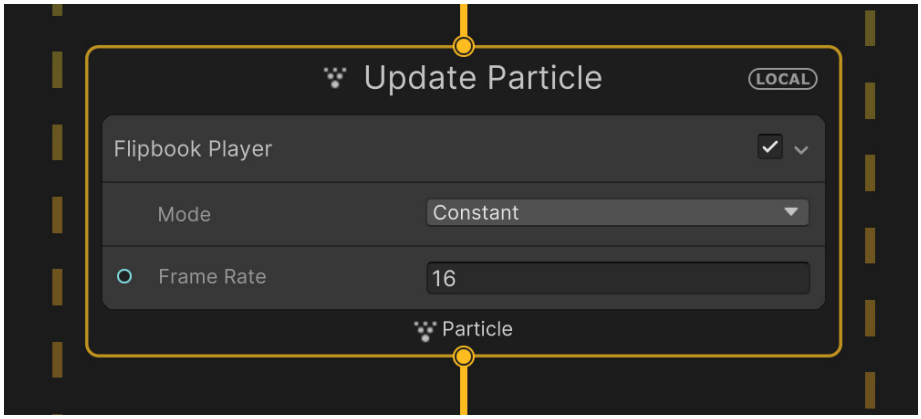


次に、1.5 秒に設定した **Set Lifetime** ノードを追加します。パーティクルの発射速度にバリエーションを加えるには、**Set Velocity Random** ノードを使用します。A を -0.1、0.4、-0.1 に、B を 0.1、1、0.1 に設定します。パーティクルの Color を設定して、スプライトを明るくしたり暗くしたりするには、**Set Color** ノードを追加し、作成した Color プロパティをその入力にドラッグします。

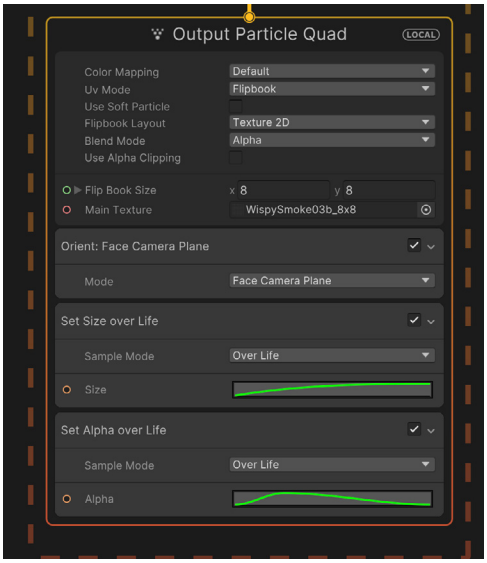
9. 次の Update ブロックでは、各フレームの更新時に何が起こるかを定義します。デフォルトでは、これは空のブロックとして表示されますが、実際にはいくつかの暗示的な非表示ブロックが含まれており、Update の選択時に、Inspector で無効化できます。



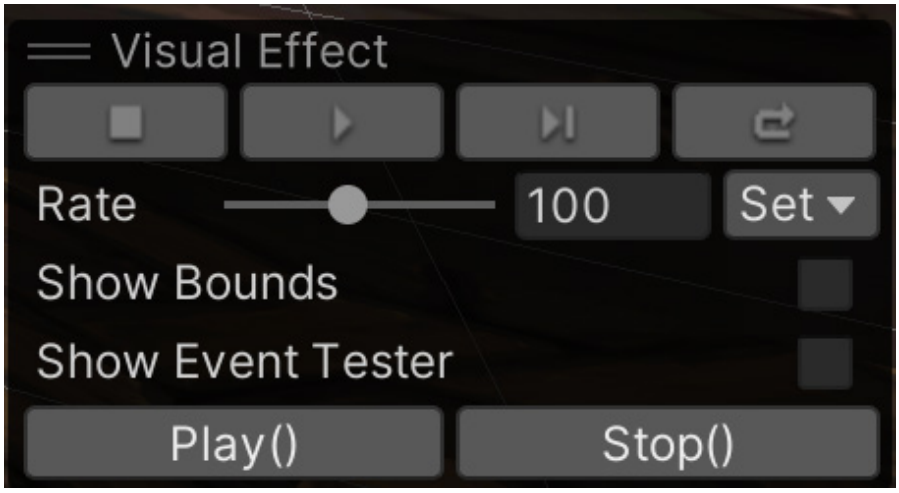
各パーティクルの画像にスプライトシートを使用していることを思い出してください。VFX Graph では、これは Flipbook を使用していることを意味します。**Flipbook Player** ノードを追加し、**Mode** を **Constant** に、**Frame Rate** を 16 に設定します。Flipbook の連続フレームを 1 秒間に 16 フレームずつ切り替えます。



10. 次に、パーティクルの最終的な出力を設定します。**UV Mode** を **Flipbook**（フレーム間の遷移をより滑らかにするには **Flipbook Blend**）に設定し、**Flipbook Layout** を **Texture 2D** に設定します。スプライトシートを使って、**Flipbook Size** を 8x8 に設定し、**Main Texture** をこの**テクスチャ**に設定します。**Set Color Over Life** を**Set Alpha Over Life** に置き換えます。デフォルトのカーブはパーティクルのライフタイムにわたってパーティクルをブレンドします。



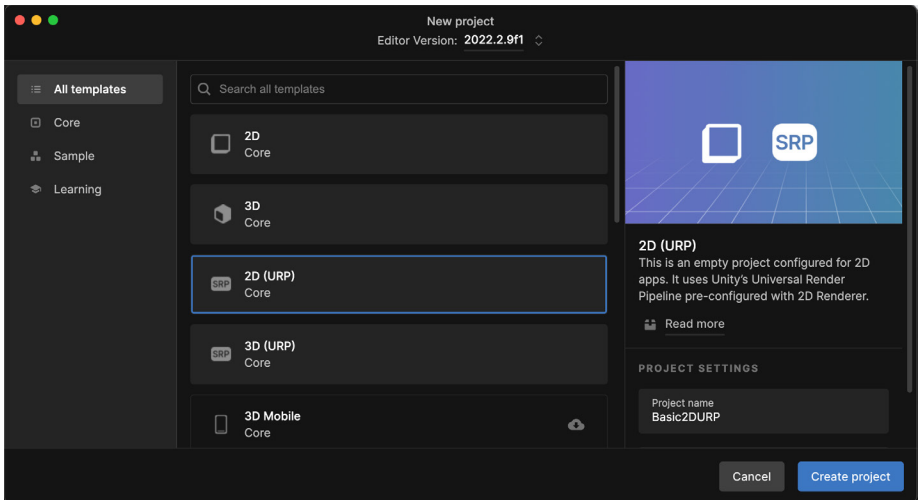
11. この VFX Graph がアタッチされた**ゲームオブジェクト**を選択します。Scene ビューには、ランタイム外で効果のデモを確認するのに使用できるパネルが表示されているはずです。パネルが表示されない場合は、**パーティクルシステム**の表示トグルがオンになっていることを確認してください。



以下は、最終的な煙エフェクトが適用されている画像です。

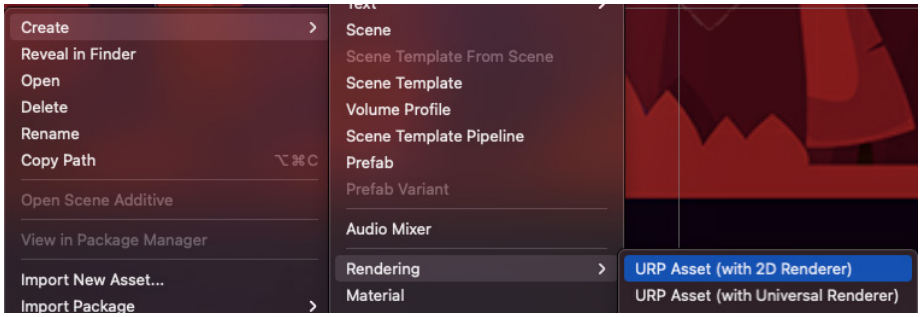
2D レンダラーと 2D ライト

2D ゲームを開発している場合、専用の URP 2D レンダラーがあります。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 テンプレートとともに、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を見つける必要があります。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。Scene ビューで、編集時に必ず**「2D」**ボタンを選択してください。



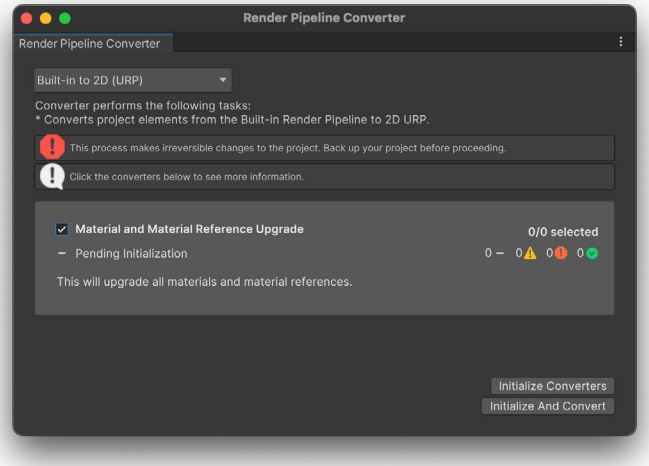
2D レンダラーと設定アセットを作成

既存のプロジェクトをアップグレードする場合、URP 2D レンダラーに切り替えると、お馴染みのマゼンタ色のレンダリングエラーが発生することがあります。

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ことがあります。

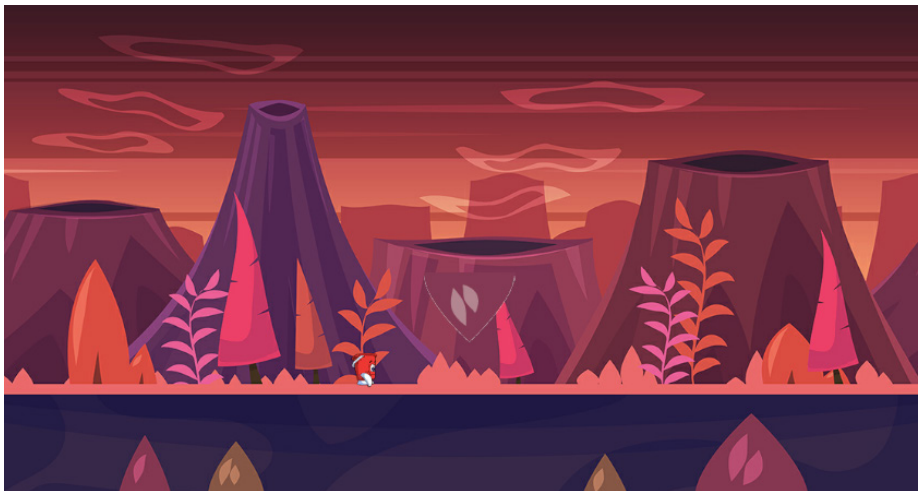
幸い、この問題は**「Window」>「Rendering」>「Render Pipeline Converter」**で解決できます。**「Built-in to 2D (URP)」**を選択し、**Material** および**「Material Reference Upgrade」**パネルをクリックします。次に、**「Initialize Converters」**をクリックした後、**「Convert Assets」**をクリックしていくつかの項目の選択を解除するか、**Initialize And Convert** でこのプロセスをワンクリックで処理を行います。依然としてマゼン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



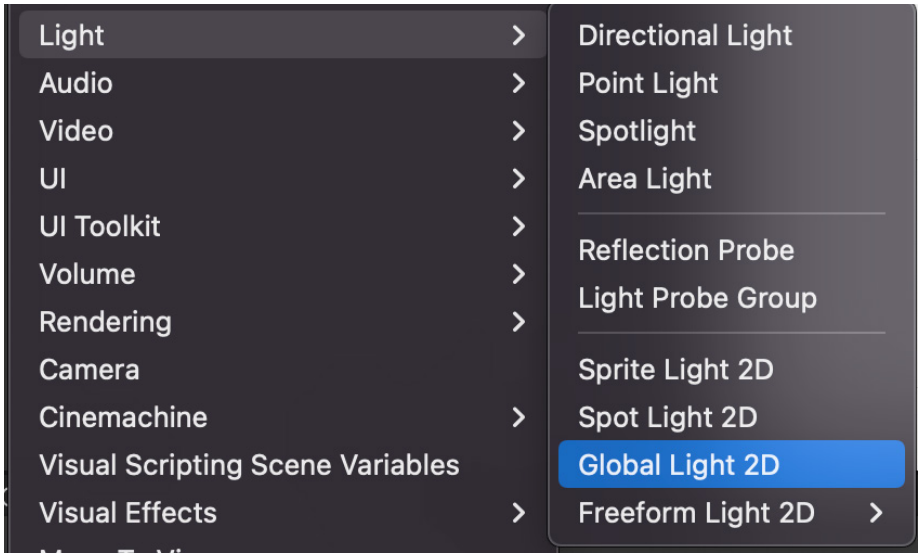
ビルトインレンダーパイプライン 2D プロジェクトを URP 2D に変換する

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。

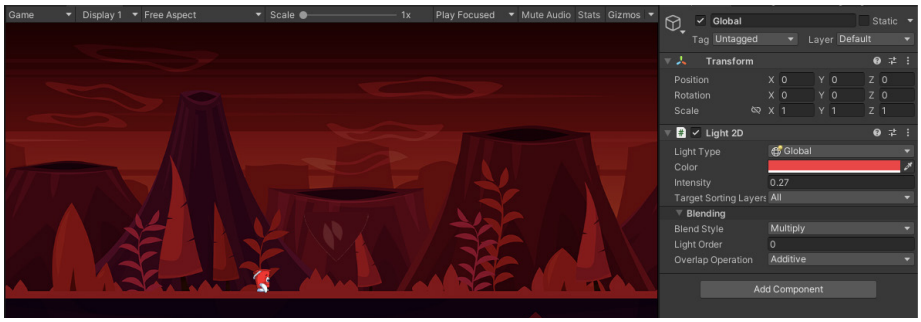


シーンにライトがない場合、レンダリングはデフォルトで Unlit になります。

「**Hierarchy**」ウィンドウを使用してライトを追加します。右クリックして「**Light**」>「**Global Light 2D**」を選択します。



これで、**Settings**、**Color**、**Intensity**、そしてその影響を受ける **Target Sorting Layers** の調整ができるようになりました。



グローバルライト 2D 設定では、キャラクターは Unlit シェーダーを使用します。

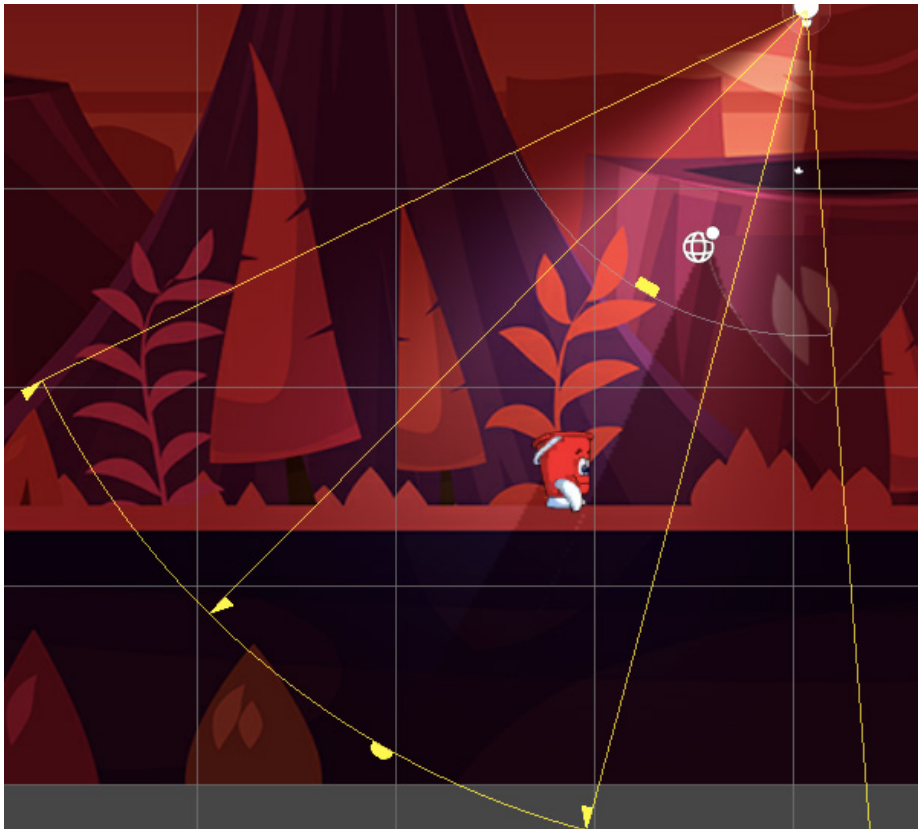
2D URP フレームワークには、4 つのライトタイプが含まれます。

— **Sprite**：スプライトを使用してライティングレベルを制御します。

— **Freeform**：ポリゴン形状のライトを作成します。

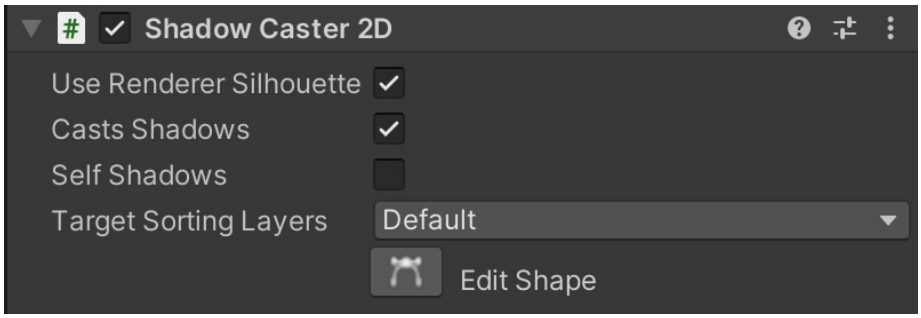
— **Spot**：選択したライトの角度と方向を細かく制御できます。ポイントライトとして使用します。デフォルトでは、内円錐と外円錐の範囲は、360 度にわたります。また、内側と外側の半径を調整し、ライトが影を落とすかどうか、また影の強度を決定できます。

— **Global**：ターゲットのソートレイヤー上のすべてのオブジェクトを照らします。



スポットライト 2D を編集する

スプライトが影を投影する場合、Shadow Caster 2D コンポーネントを追加する必要があります。



Shadow Caster 2D コンポーネントを追加

URP 2D レンダラーは、ローエンドのハードウェアでも十分に動作するファーストクラスの 2D ゲームを作成するために必要なツールをすべて提供します。

Unity 2D デモ「Dragon Crashers」からの画像。Unity の 2D 開発 e ブック、「2D game art, animation, and lighting for artists」は、「Dragon Crashers」のクリエイティブディレクターによって執筆されたものです。

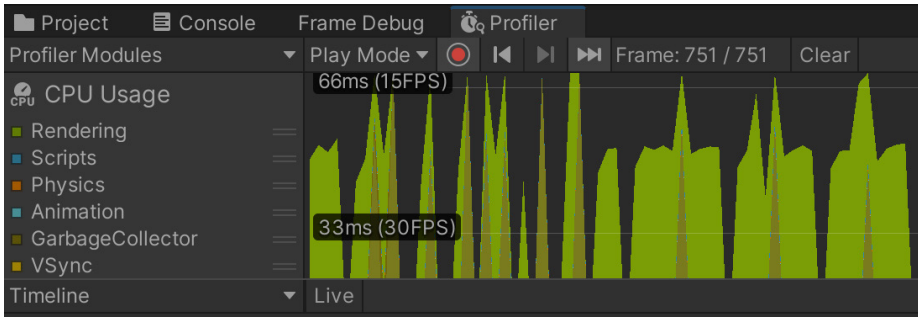
関連リンク:

— Unity の 2D デモ、「Dragon Crashers」は Unity Asset Store で入手可能です。

— 「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上級者向け開発ガイドです。

パフォーマンス

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。

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用のカメラあたり最大 256 のリアルタイムライト、さらにモバイルやその他のハンドヘルドプラットフォーム用のカメラあたり 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と仮定すると、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、視覚的な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。

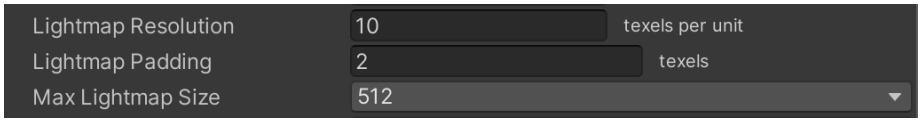
左から順に、ライティングデータなし、ベイクされたライティングあり、ポストプロセス追加

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。

前の画像からは、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 を設置する

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。

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をも含む、すべてのハードウェアに最適です。

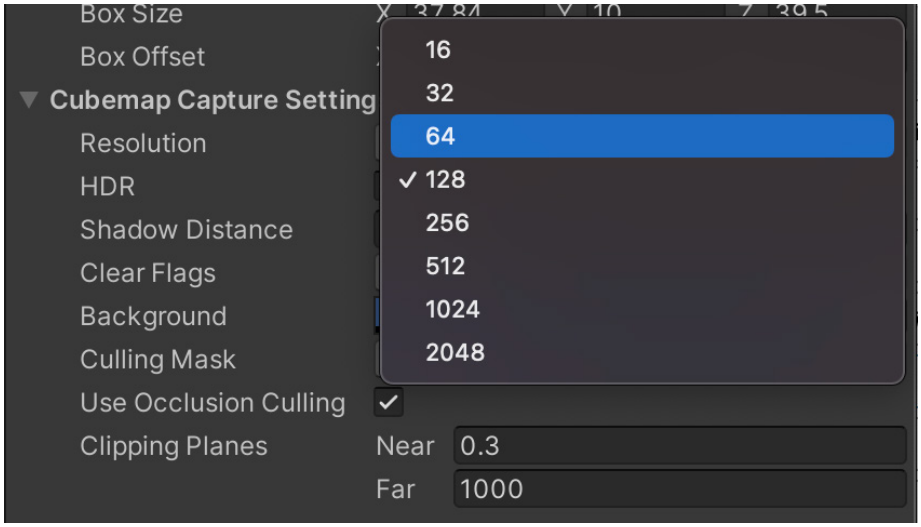
動的オブジェクトのレンダリング時にライトプローブを使用した場合の効果：ライトプローブあり（左）、ライトプローブなし（右）

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、より周囲の環境にマッチする反射を再現できます。

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接近する距離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

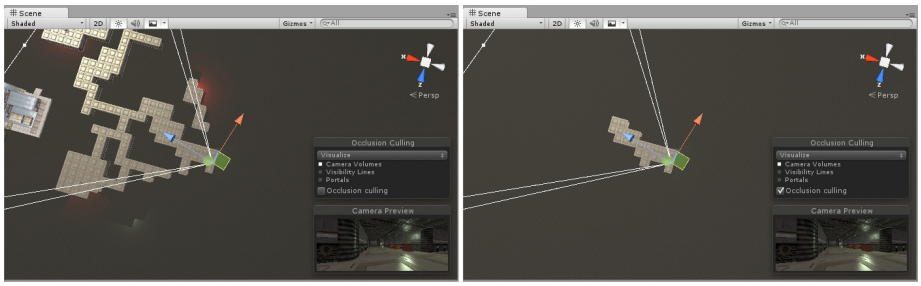
カメラ設定

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後ろ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意味はなく、ミリ秒単位の貴重な時間を使うことになります。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。

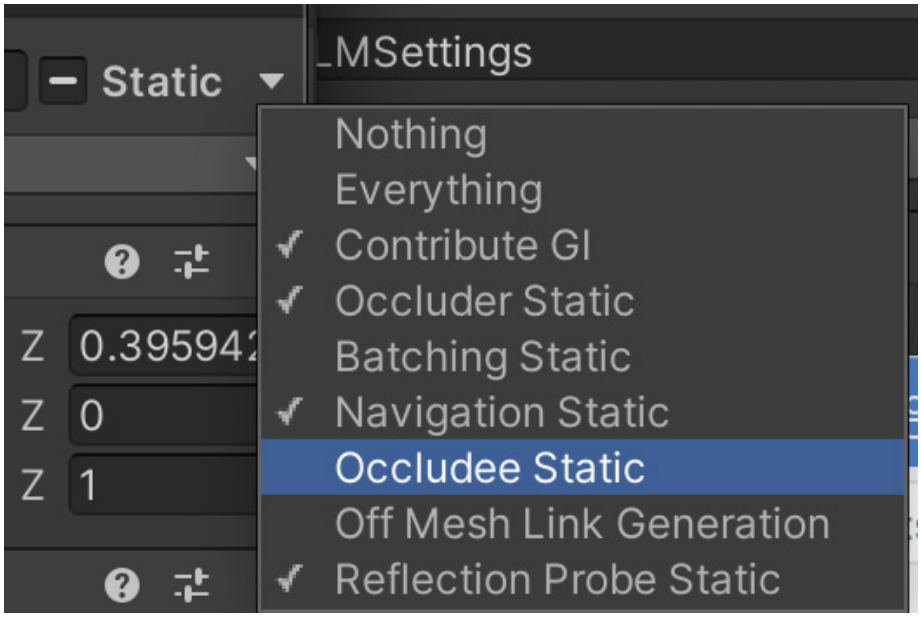


フルスタムカリング（左の画像）、オクルージョンカリング（右の画像）

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。

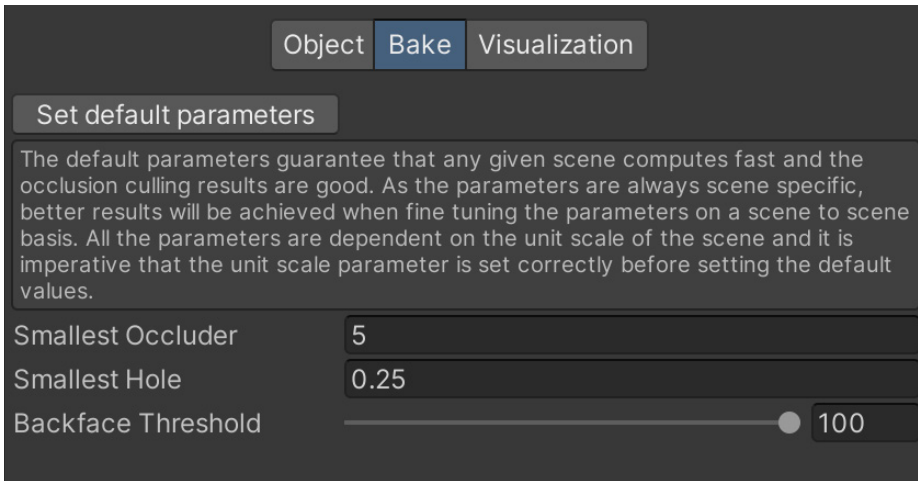
シーンでオクルージョンカリングを有効にするには、任意のジオメトリに **Occluder Static** または **Occludee Static** のマークを付けます。遮蔽物は、被遮蔽物としてマークされたオブジェクトをオクルードできる中型から大型のオブジェクトです。遮蔽物になるオブジェクトは、不透明で、地形またはメッシュレンダラーコンポーネントを持ち、実行時に移動しない必要があります。被遮蔽物はレンダラーコンポーネントを持つオブジェクトであれば何でもよく、同様にランタイムに動かない小さなオブジェクトや透明なオブジェクトも含まれます。

静的プロパティは、通常のドロップダウンで設定します。



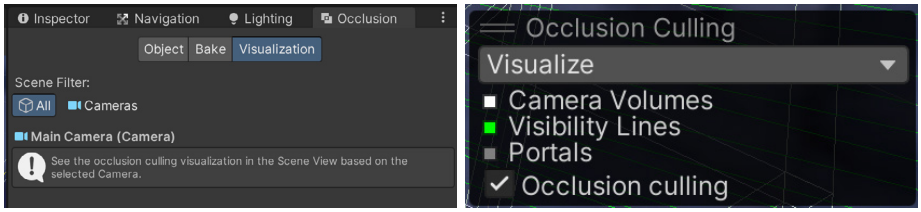
オクルージョンデータに含まれるオブジェクトの設定

「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。



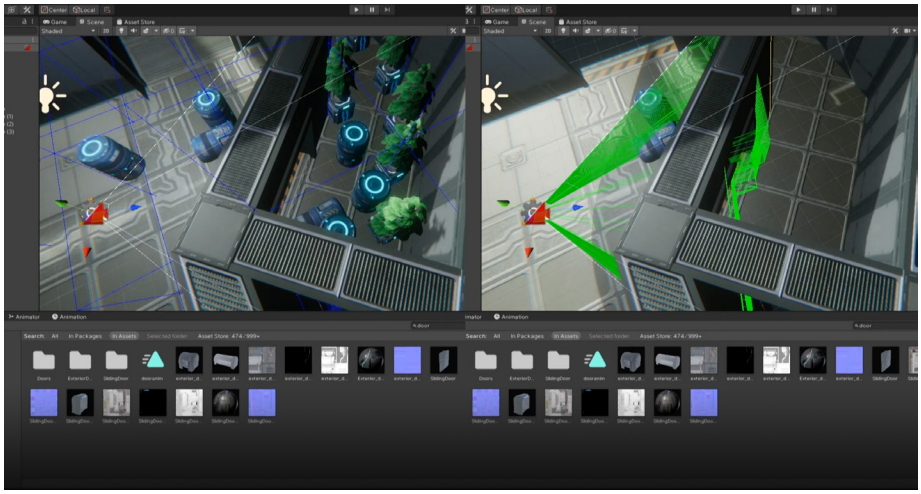
オクルージョンカリングの Bake タブ

「**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「**Camera**」を選択し、**Scene** ビューの「**Occlusion Culling**」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 Camera 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 Camera 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。

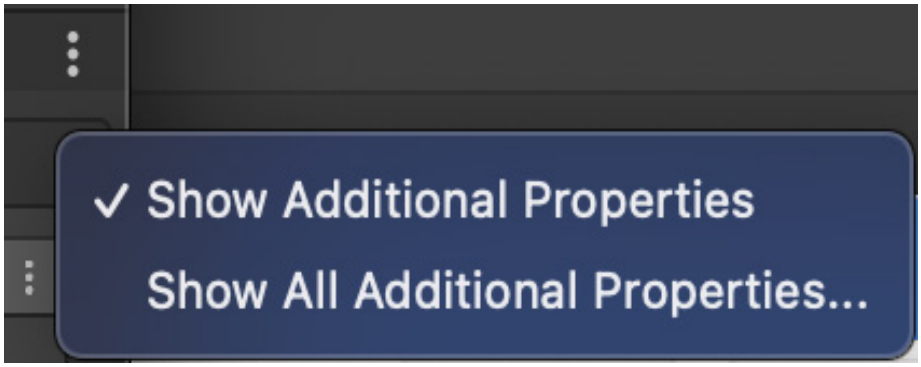


オクルージョンカリングがオフの場合の効果（左の画像）とオンの場合の効果（右の画像）

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、前回説明しましたが、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。

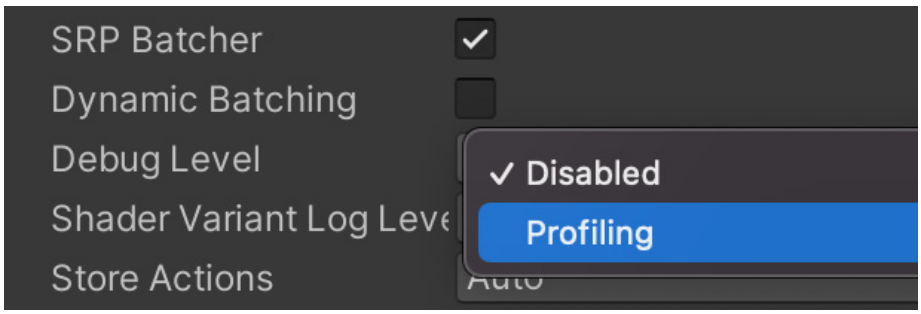
- パフォーマンス向上のために影の解像度と距離を減らします。
- 深度テクスチャや不透明テクスチャなど、プロジェクトが必要としない機能を無効化します。
- 新しいバッチ処理方法を使用するには、SRP Batcher を有効にします。SRP Batcher は、同じシェーダーバリアントを使用するメッシュに自動的にバッチ処理を適用して、ドローコールを減らします。シーンに多数の動的オブジェクトがある場合、これはパフォーマンスを上げる便利な方法になります。SRP Batcher のチェックボックスが表示されていない場合は、3 つの縦のドットアイコン (⋮) をクリックし、「**Show Additional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



URP Asset Inspector の追加プロパティの有効化

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 チェックボックスと同様に、これは **Show Additional Properties** が有効化された Inspector にのみ表示されます。

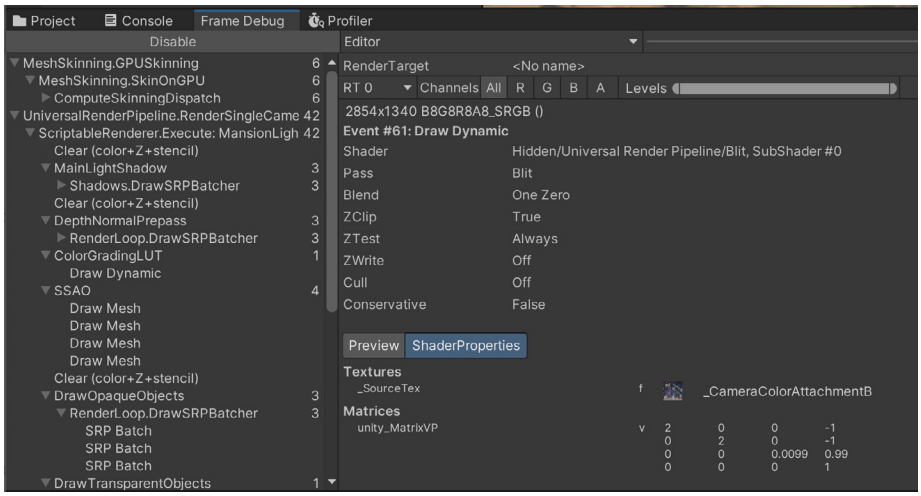


デバッグレベルの設定

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。

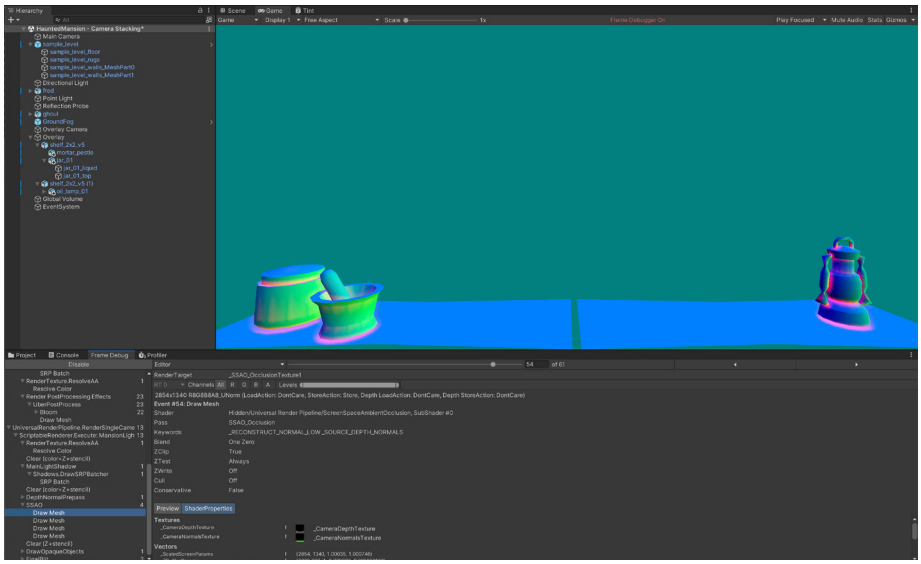
フレームデバッガーは、最終画像をレンダリングするまでに実行されたすべてのドローコールのリストを表示し、特定のフレームのレンダリングに時間がかかっている理由を特定するのに役立ちます。また、シーンのドローコール数が非常に多い理由を特定することもできます。

「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、**Game** ビューにそのステージのプレビューが表示されます。



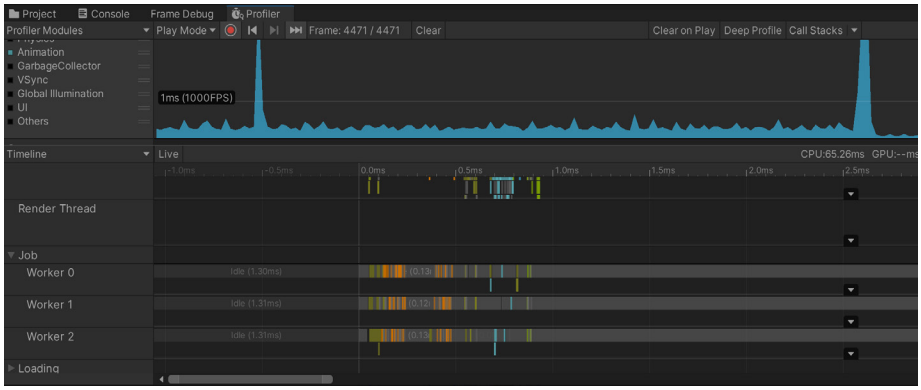
フレームデバッガーは、Game ビューのレンダリングプロセスの各ステップ（この場合は SSAO 生成ステップ）を表示します。

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。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、このウィンドウはゲームの全体的なパフォーマンスの概要を提供します。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。



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

```
#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;
}
```

左側に、低レベルのネイティブプラグイン Profiler API の使用例を載せています。

追加リソース

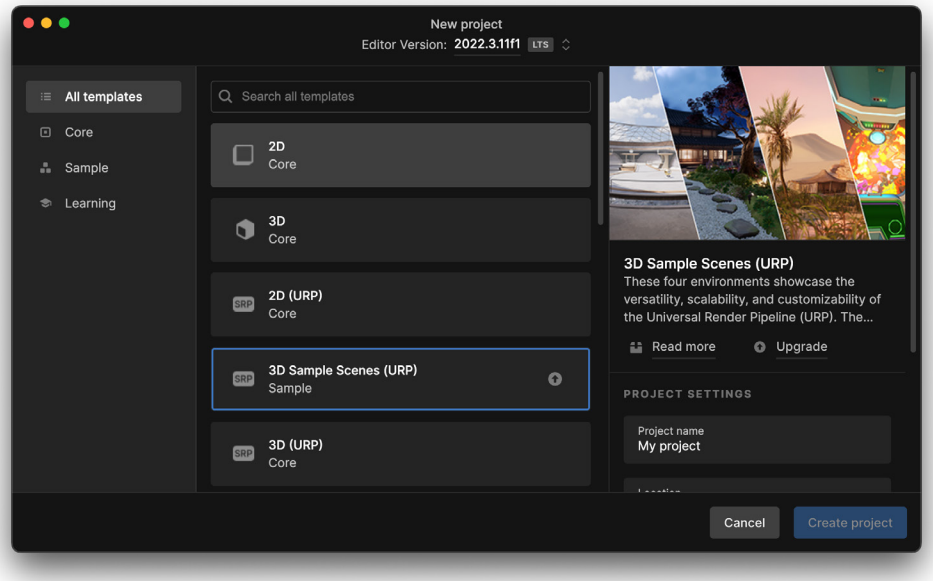
Unity で高度なプロファイリングスキルを構築したい場合は、まず無料の e ブック「Ultimate guide to profiling Unity games」をダウンロードしましょう。このガイドには、Unity でのアプリケーションのプロファイリング、メモリの管理、消費電力の最適化に関する高度なアドバイスや知識が始めから終わりまでまとめられています。

Nik が推奨する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 3D サンプル

新しい URP 3D サンプルは Unity Hub で入手できます。このサンプルプロジェクトは、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。URP 3D サンプルには、Unity 2022 LTS における URP の機能を説明する 4 つの異なる環境が含まれています。

各環境を見ていきましょう。



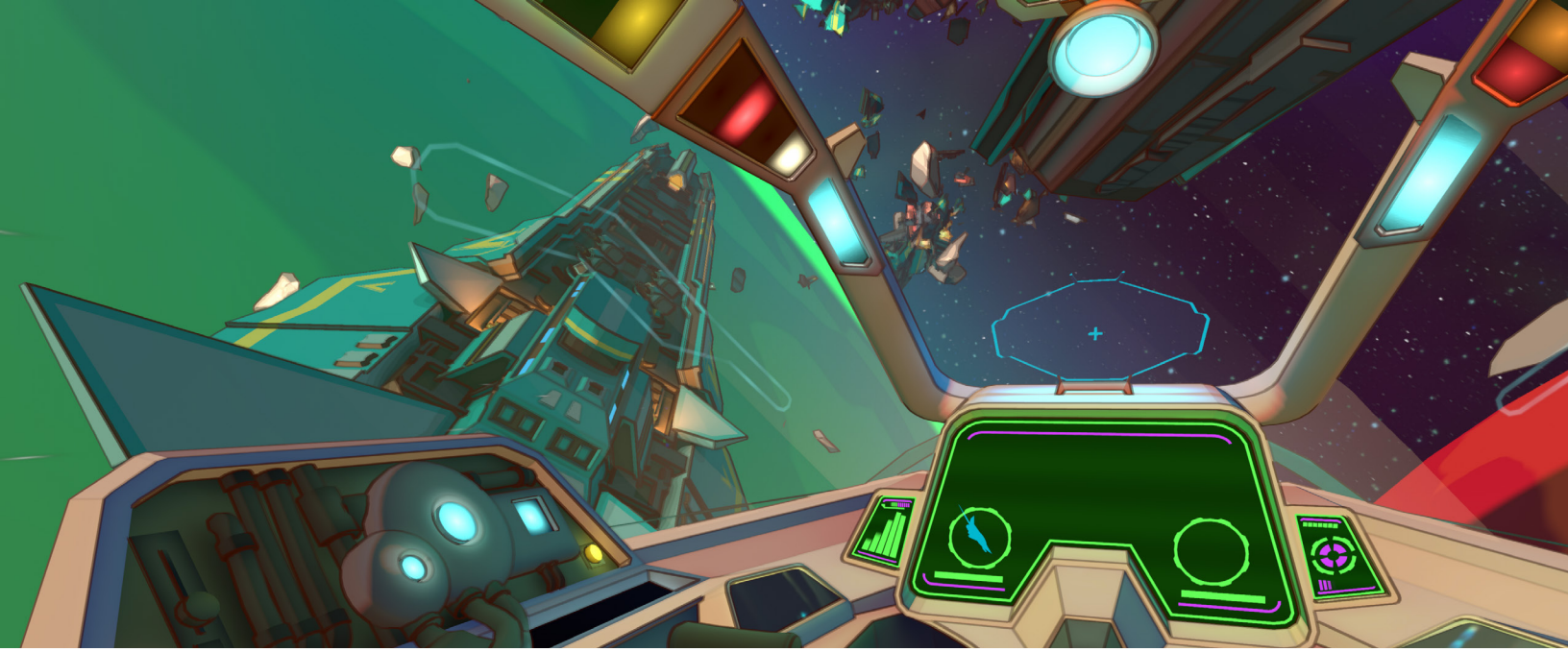
URP 3D サンプルは、Unity 22 LTS で新しいプロジェクトを開始すると Unity HUB で利用できるようになります。サンプルの詳細は、このウェブサイトから確認できます。

庭園

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。

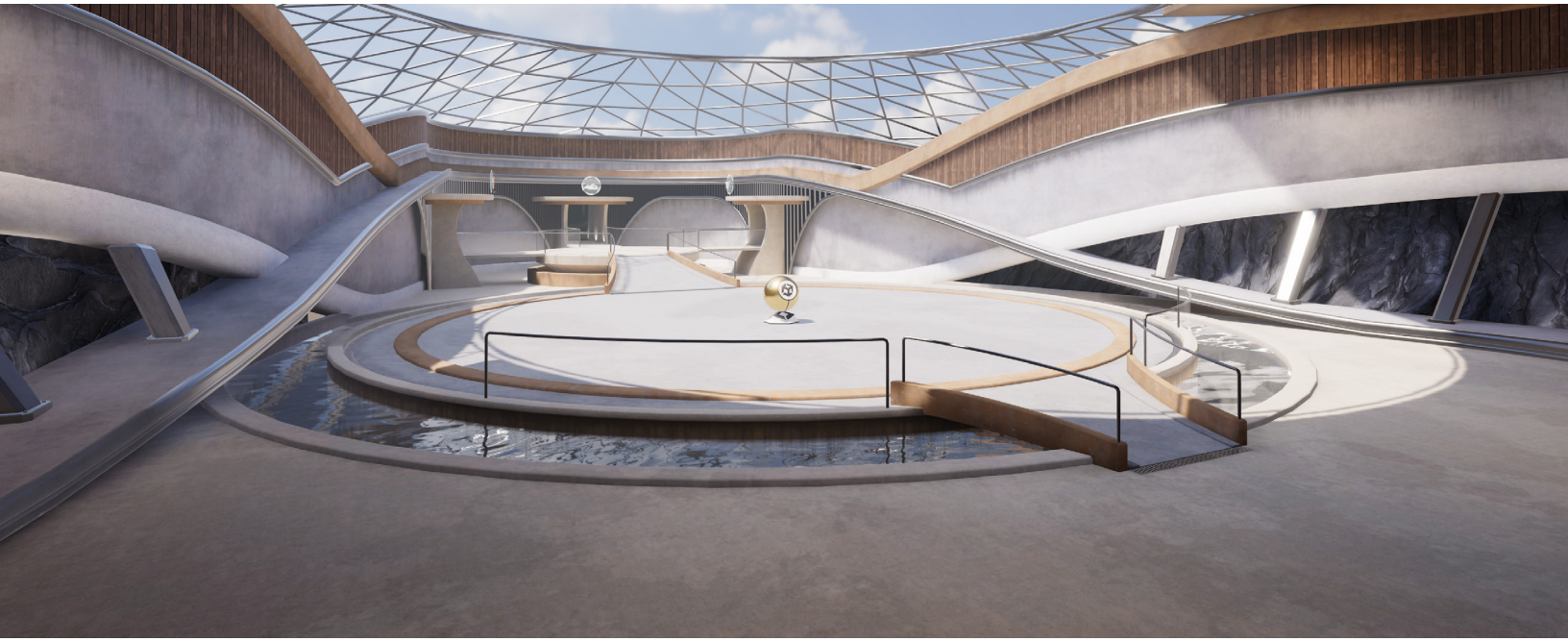
オアシス

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。



コックピット

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。

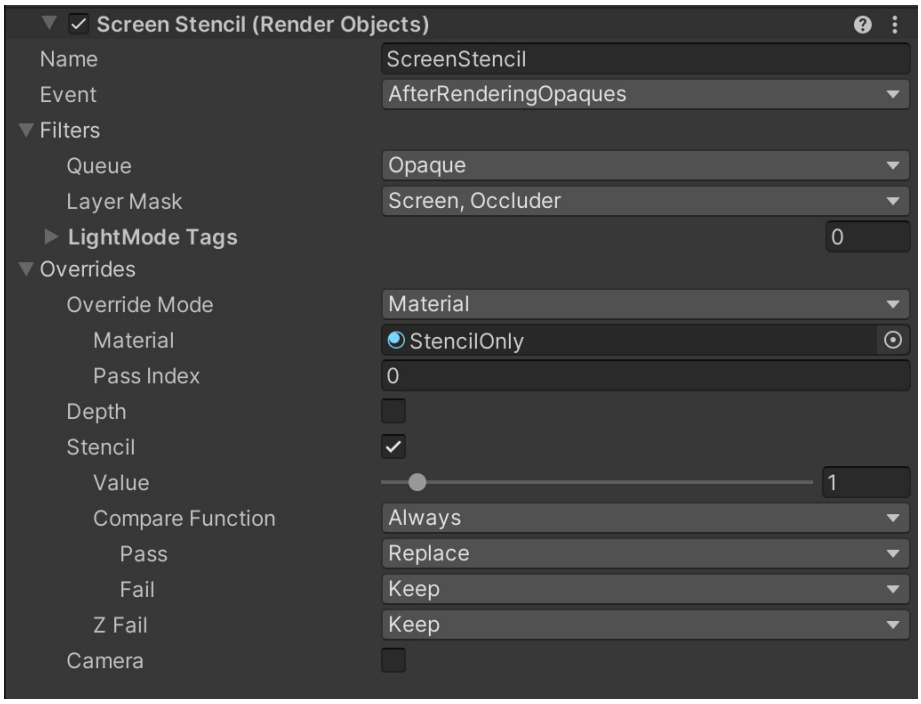


ターミナル

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、あるシーンから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。また、look-dev 用のアセットを追加できる最適な設定も備えています。

環境間の移動

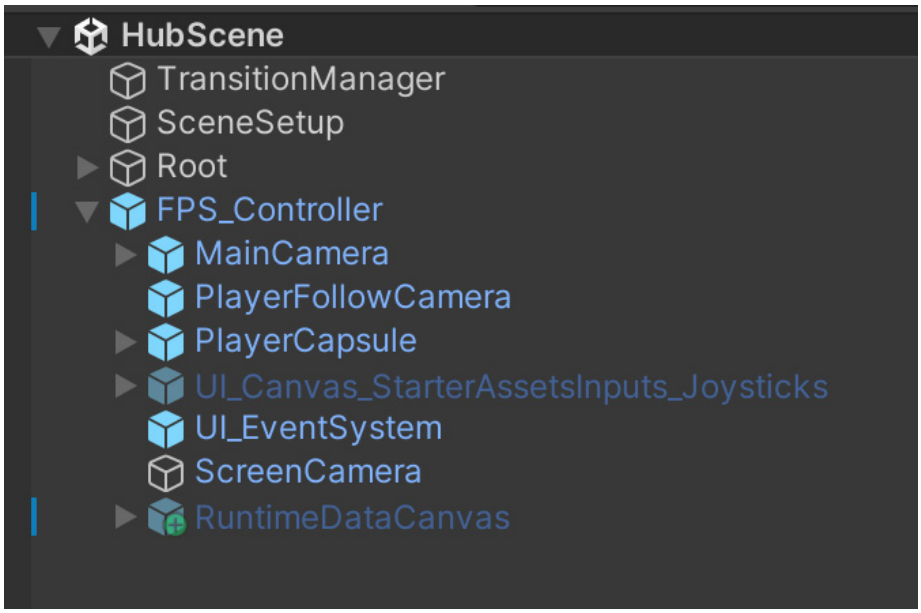
サンプルプロジェクトでは、シーン間の移動にトランジションエフェクトを使用しています。トランジションエフェクトは、画面外のレンダーターゲットを使用して、トランジションが完了する前に受信するシーンをレンダリングします。入力シーンは、Shader Graph で作成されたカスタムシェーダーを使用して、出力されるシーンに配置された大型モニターにレンダリングされ、フルスクリーンスワップは、Render Objects Renderer Feature を介したステンシルを使用して処理されます。



Screen Stencil Renderer Feature

この効果を実際に見るには、Unity のロゴが表示されるまで台座に向かって歩き、ロゴを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。



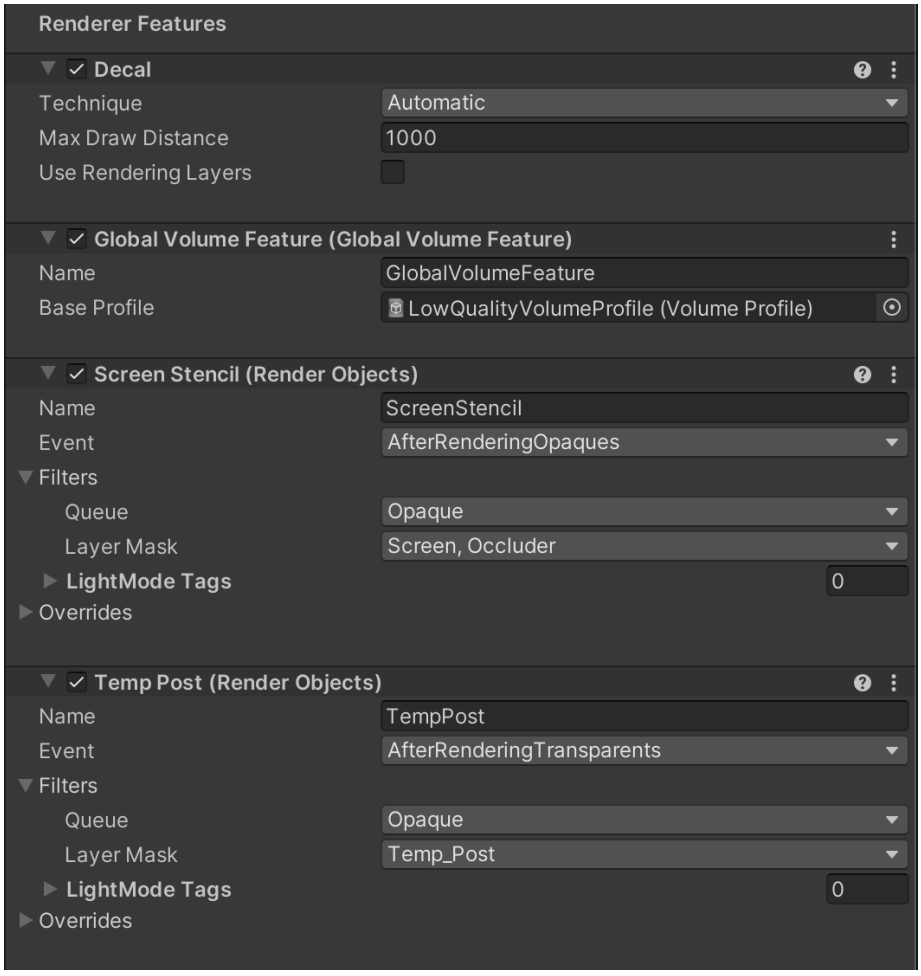
ターミナルシーンの FPS_Controller

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** の呼び出しにより、**currentScene** のメインディレクショナルライトをアクティブにし、入力シーンである **screenScene** のメインディレクショナルライトを無効にします。ただし、**isMainCamera** が false の場合は逆の操作になります。

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。

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、色バッファにすでにある値を保持し、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。

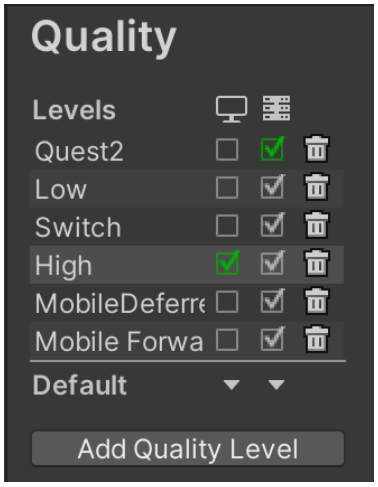


モバイル Foward+ レンダラーの Renderer Features

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、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



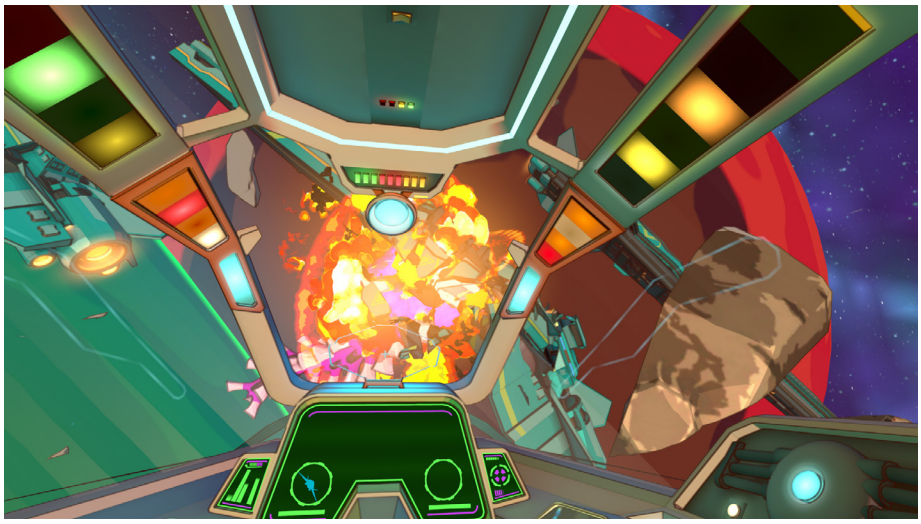
品質レベル

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。

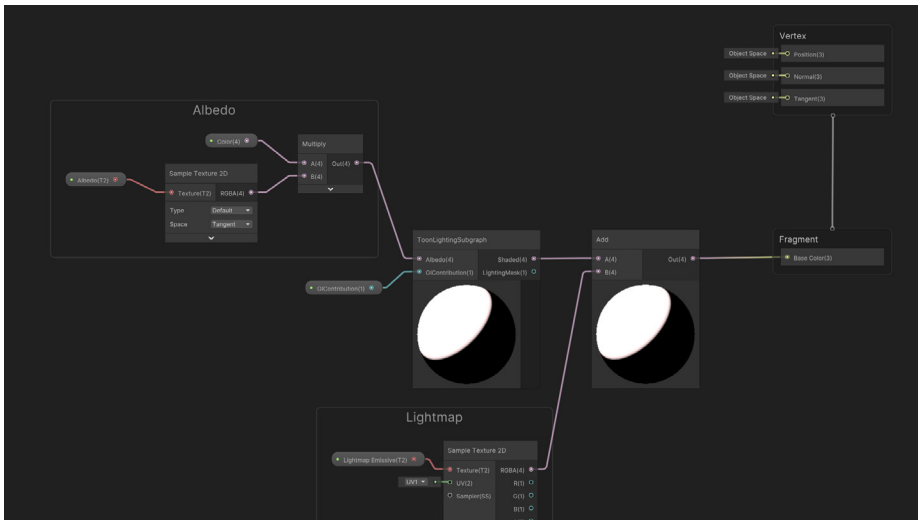
各オプションは異なる**レンダーパイプラインアセット**を使用します。品質のセクションで説明したように、URP はこのパネルとレンダーパイプラインアセットの設定を組み合わせて品質を処理します。

スタンドアロン VR ヘッドセットには、リアルタイム 3D グラフィックスを表示する際に、重要な課題があります。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。



コックピットのサンプルシーン

カスタムライティングは Shader Graph を使用して処理され、コーディングは不要です。



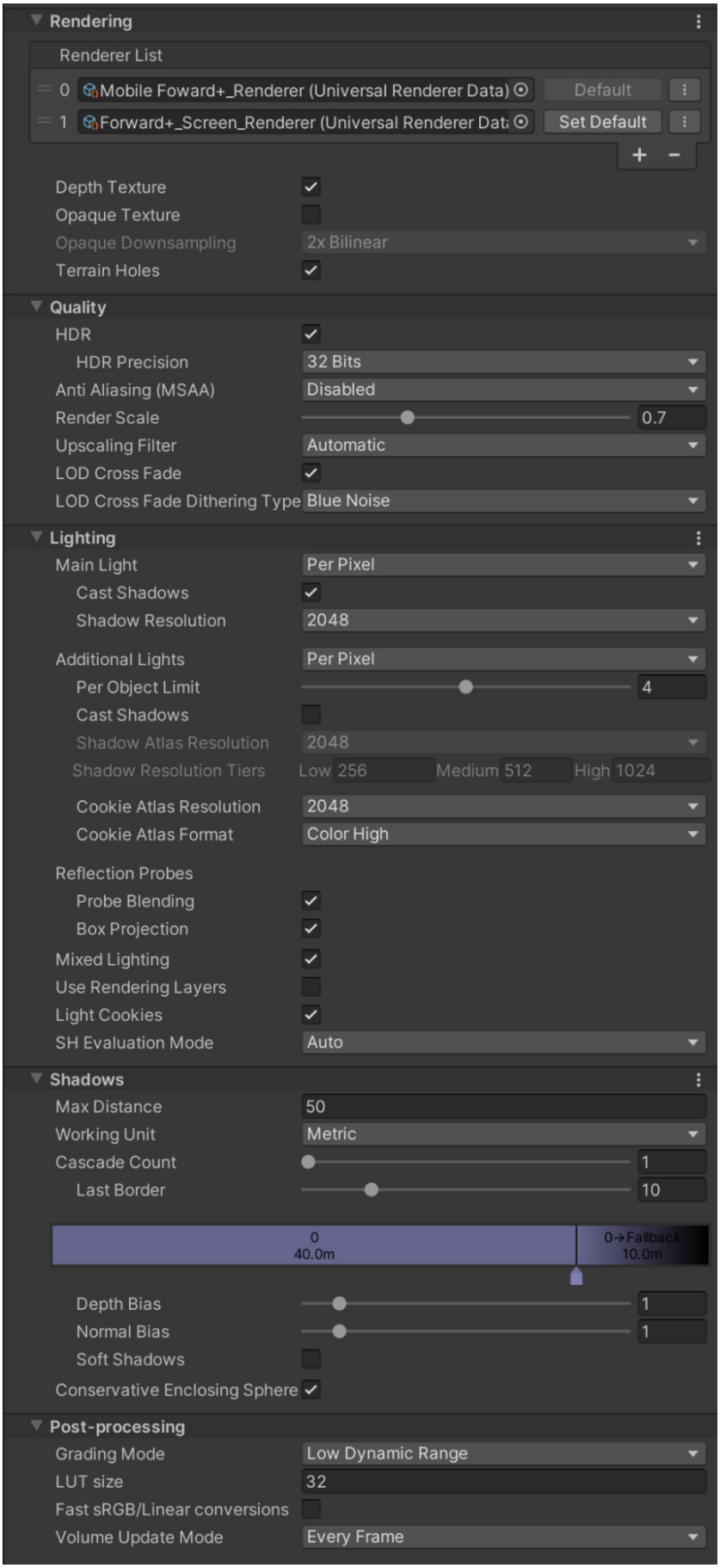
通常のトゥーンシェーダーと同様に、法線ベクトルとメインライトの方向をドット積で結合し、ライティングレベルを決定します。その後、ランプを使用して、値を滑らかに変化させるのではなく、段階的にライトレベルを設定します。コックピットシーンで使用されているライティングモデルは、ベイクされたグローバルイルミネーションを使用して計算を実行し、微妙なアウトライン効果を追加するためにエッジ検出も行います。カスタムライティングは Shader Graph を使って処理されます。

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

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。

次ページのスクリーンショットは、モバイルフォワード+ アセットの設定を示しています。



モバイルフォワード+ URP アセット

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。次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。

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 x 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。

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。

クッキーの有無による庭のシーンのポイントライト

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、URP フォーラムを必ずチェックしてください。

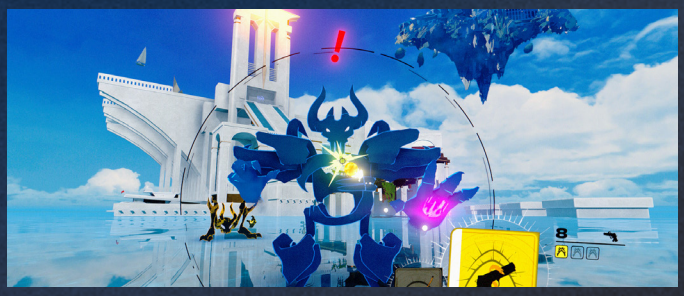
Unity の Product Board では、現在開発中の URP 機能の概要に加え、次のリリースについて知ることができます。機能リクエストを追加することもできます。

この e ブックの締めくくりに、Unity の URP が提供するレンダリングの力と柔軟性を活用して作成された、独創的で素晴らしいゲームをいくつかご紹介します。

皆様の開発の成功をお祈り申し上げます。

『Dave the Diver』(開発元：MINTROCKET)

コンソールおよび PC 向けの『Lost in Random』(開発元：Thunderful Games、販売元：Electronic Arts)



『Neon White』(開発元：Angel Matrix と Ben Esposito、販売元：Annapurna Interactive)

『Pixel Ripped 1978』(開発元：ARVORE Immersive Experiences)

『Death in the Water 2』(開発元：Lighthouse Games Studio)

『Bare Butt Boxing』(開発元：Tuatara)

『Can't Live Without Electricity』(開発元：MELOVITY)

unity.com