開発者向け UNITY

E ブック

ユニバーサルレンダーパイプライン（URP）クックブック：

シェーダーと視覚エフェクトのレシピ

UNITY 2022 LTS EDITION

Unity®

目次

はじめに

ポストプロセスエフェクトを少々、デカールをカップ 1 杯、カラーグレーディングをひとつまみ、そして発泡水を適量…。そのようにして、ユニバーサルレンダーパイプライン（URP）を使い、ゲームの高品質のライティングと視覚エフェクトを手早く料理するお時間です。

このクックブックでは、12 のレシピから選択して、URP を使用して一般的なエフェクトを作成できます。また、これらのレシピに基づくサンプルシーンを、ガイドの筆頭著者である Nik Lever 氏が調整しているこちらの GitHub リポジトリからダウンロードできます。

このガイドは、上級 Unity ユーザーを対象としています。Unity でのプロジェクト開発、URP、および HLSL を使用したシェーダーの記述について基本的な知識があることを前提としています。

以下については、もう調理のために必要な材料はすべて揃っています。

— ステンシルを使用して X 線のようなイメージエフェクトを作成する

— シェーダーグラフを使用してトゥーンおよびアウトラインシェーダーをビルドする

— ポストプロセスを使用してアンビエントオクルージョンエフェクトを追加する

— Photoshop と LUT 画像を使用してシーンにカラーグレーディングを追加する

— 反射と屈折を生成する、など。

当クックブックは「*Introduction to the Universal Render Pipeline for advanced Unity creators（上級 Unity クリエイター向け URP 入門）*」のガイドと並行して読んでいくと、さらに有用です。Unity の YouTube チャンネルでは、ゲームのライティングとエフェクトを作成するための一般的なガイドと、より具体的な部分に向けたヒントの両方を提供する「URP チュートリアル」シリーズもあります。

ゲーム制作において、目を見張るような効果を創るということをお楽しみいただければ幸いです。

本 e ブックにおける、この画像と表紙は、熟練した開発者の手によれば URP を使用してここまで実現できるという一例である『*PRINCIPLES*』からのものです。『*PRINCIPLES*』とは、『*白猫プロジェクト*』や『*クイズ RPG 魔法使いと黒猫のウィズ*』のシリーズを開発した COLOPL Inc における技術ブランドである COLOPL Creators によるアドベンチャーゲームです。URP も使われている、Unity の最新機能を駆使した息をのむグラフィックスと没入型の 3D サウンドの地下世界を体験してみましょう。『*PRINCIPLES*』は、App Store と Google Play で入手できます。こちらでスタジオのインタビューを視聴することもできます。

著者と貢献者

この e ブックの筆頭著者である Nik Lever 氏は、90 年代半ばからリアルタイム 3D コンテンツの作成を行っており、2006 年から Unity を使用しています。30 年以上にわたって小規模な開発会社（Catalyst Pictures）を率いている同氏は、急速に進化するゲーム業界で知識を広げようとする開発者を支援するべく、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、グラフィックス製品管理チームのシニアマネージャーです。Ali はゲーム業界で 18 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。

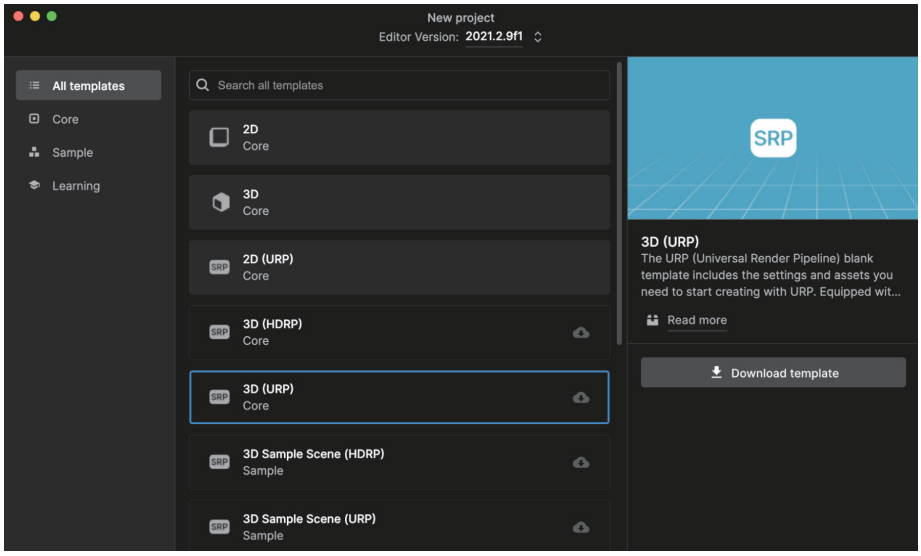
Steven Cannavan は、Accelerate Solutions Games チームのシニア開発コンサルタントで、スクリプタブルレンダーパイプラインを専門としています。Steven はゲーム開発業界で 15 年以上の経験を持っています。

ガイドを開いて はじめよう

各レシピの手順に沿って進行すれば、新規に URP プロジェクトを立ち上げた際にレシピ通りのライティングエフェクトや視覚エフェクトを再現することができるようになっています。また、このガイドに即した Github ページにアクセスできます。各レシピにおいてダウンロード可能なサンプルとして提供されています。

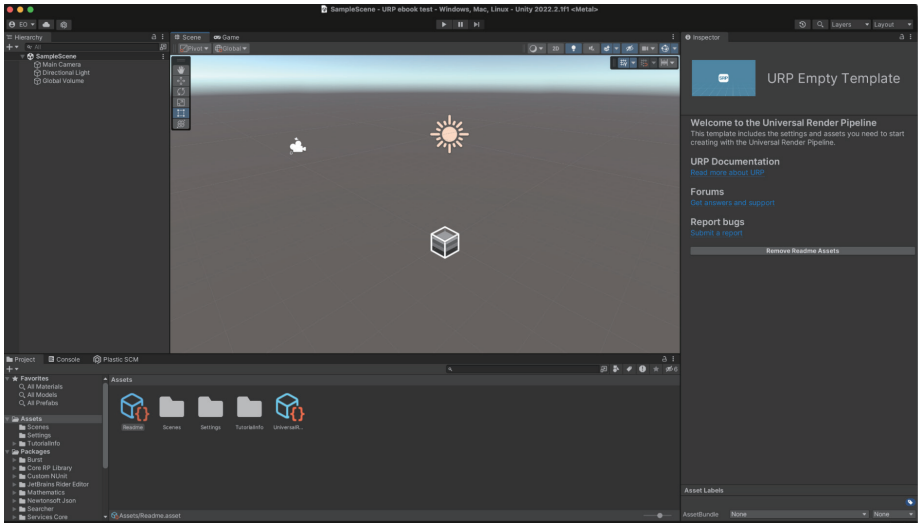
新規 URP プロジェクトの開始

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

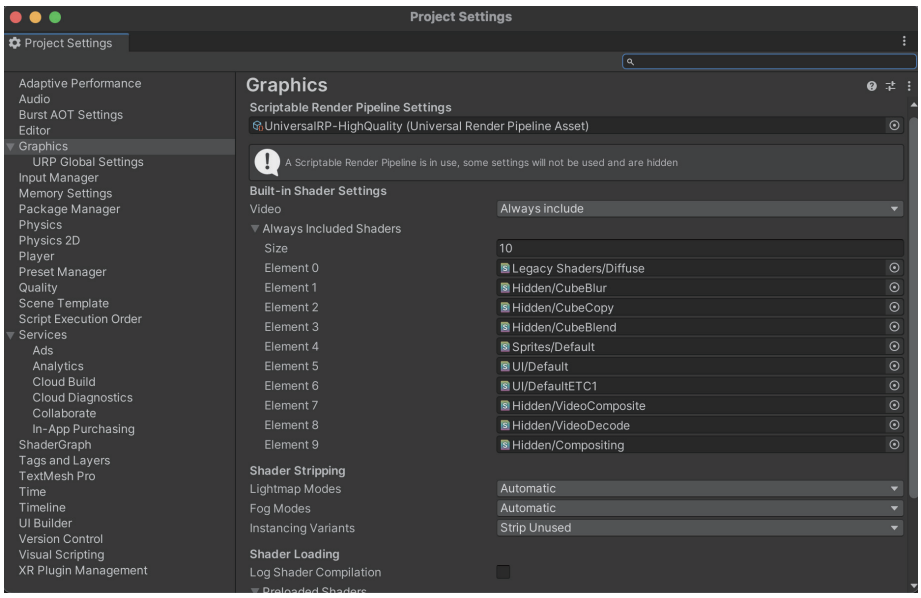
ノート：テンプレートを使用すると、プロジェクトはライティングを正しく計算するために必要なリニア色空間を使用するように設定されます。



このテンプレートは空ですが、URP とそのアセットが事前設定済みでインストールされています。

「Edit」>「Project Settings」に移動し、**「Graphics」**パネルを開きます。**「Scriptable Render Pipeline Settings」**の URP アセットが、選択された SRP として表示されます。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。

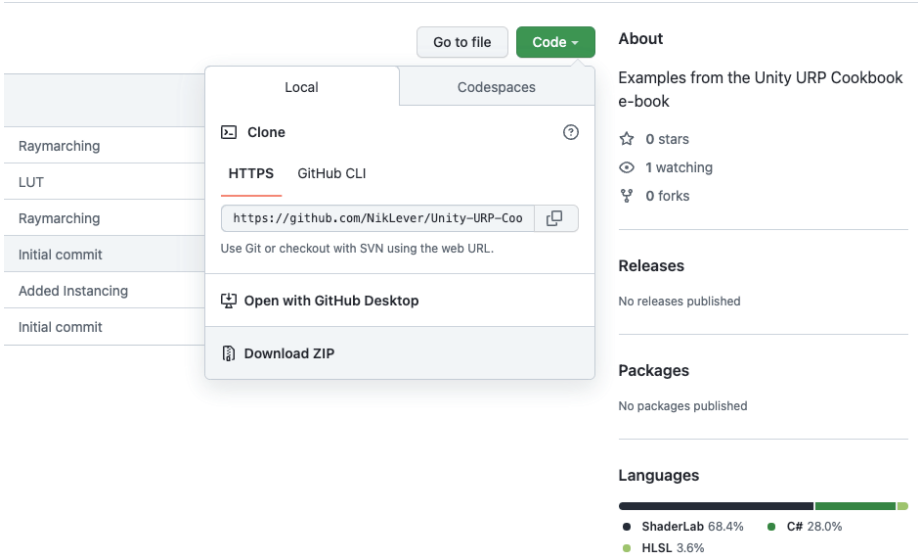
「UniversalRP-HighFidelity」がデフォルト設定の URP アセットとなっていますが、**「UniversalRP-Balanced」**または**「UniversalRP-Performant」**に切り替えることもできます。



「Project Settings」の「Graphics」パネル

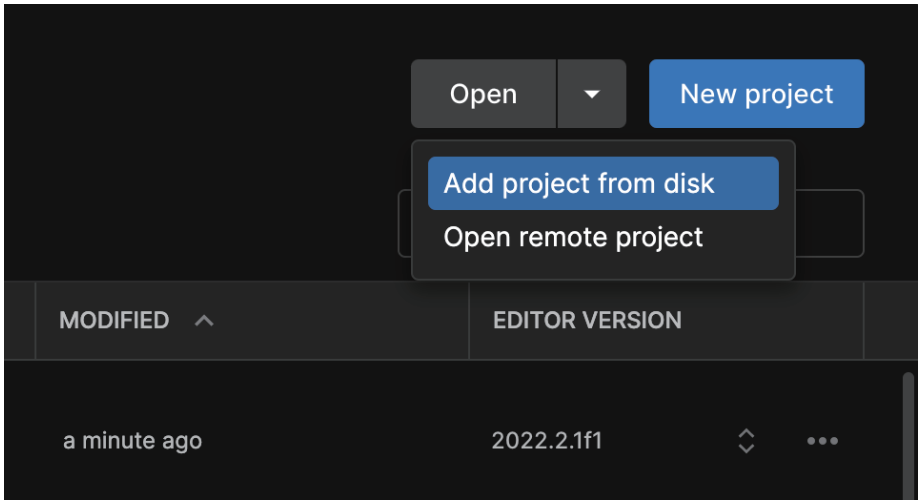
e ブックのサンプルシーンのインポート

こちらからリポジトリのクローンを作成するか、もしくは zip ファイルでコードをダウンロードして解凍もできます。



GitHub リポジトリ。緑の「Code」ボタンをクリックしてプロジェクトをダウンロード可能です。

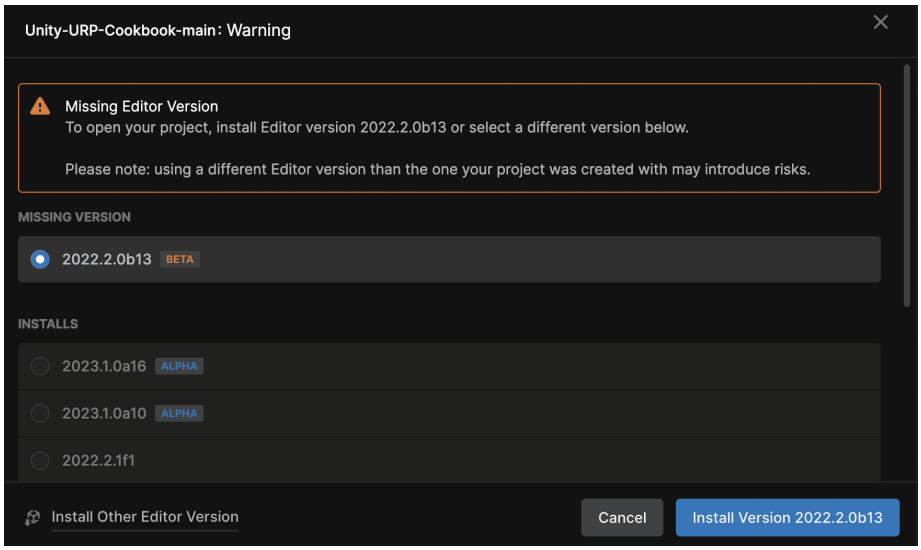
プロジェクトが解凍され、ダウンロードの後、**「Open」>「Add project from disk」**により Unity Hub からインポートします。



Unity Hub からサンプルプロジェクトをインポートする方法

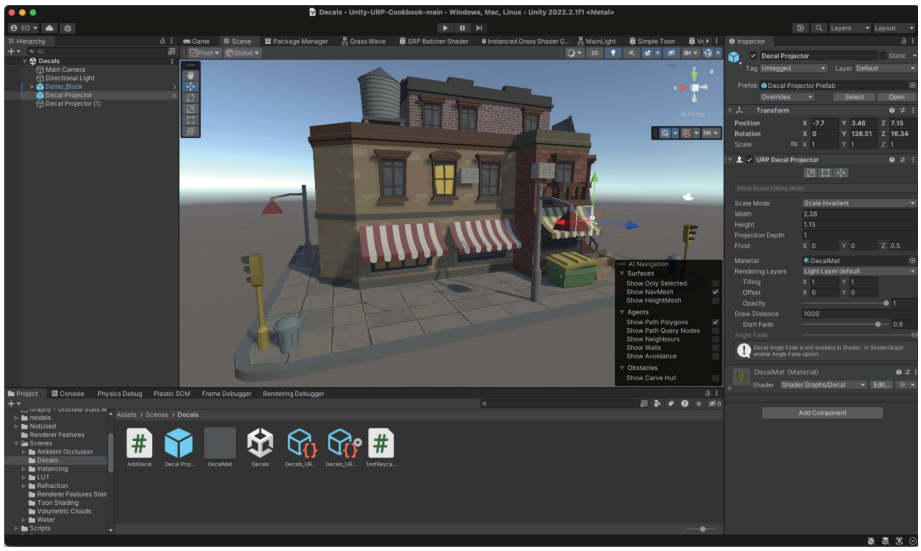
サンプルプロジェクトに使用したのと同じバージョンのエディターで作業していることが重要です。エディターのバージョンが一致していない場合、Hub はエディターバージョンが対応していないことに関する警告メッセージを表示します。

対応していないバージョンについては、画像のとおりで、画面右下部にある青色のボタンからインストールすることができます。



当ガイドレシピ中の沿って行っているチュートリアルプロジェクト、またもしくは対象のダウンロードしているチュートリアルプロジェクトと一致する Unity エディター のバージョンをインストールするのが良いでしょう。幸い、これは Unity HUB を介することで簡単に行うことができます。

適切なエディターバージョンがインストールされたら、通常どおりにプロジェクトを開くことができます。



各レシピは、フォルダー内に、このブックで参照されている手順およびファイルと共に格納されています。

URP には、最終的なレンダリングを制御する 2 つのアセット（Universal Renderer Data アセットと URP アセット）があります。前者から、Renderer Feature を追加して以下のようなレンダーパイプラインの任意のステージに挿入できます。

- 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

Renderer Feature は、ライティングとエフェクトで実験する多くの機会を提供します。このセクションでは、必要最小限のコードのみを使用するステンシルに注目します。

作業を進めるために、エディターで**「Scenes」>「Renderer Features」>「SmallRoom - Stencil」**でサンプルシーンを開きます。

Made with Unity ゲームの『*TUNIC*』（制作：22nd Century Toys LLC、TUNIC Team、Andrew Shouldice 氏および Isometricorp Games Ltd.、発行：Finji）では、舞台道具がメインキャラクターを視界から遮っているときにはキャラクターのシルエットが描画されています。このエフェクトは、URP の Renderer Feature を使用して実現できます。このビデオチュートリアルでも説明されています。ステンシルの動作：拡大鏡をデスクの上へ移動させると、引き出しの中身が透けて見えるようになっている。

ステンシルの動作：拡大鏡をデスクの上へ移動させると、引き出しの中身が透けて見えるようになっている。

上の画像が示すように、この例の目的は、拡大鏡のレンズを変換して、X 線画像のようにデスクを透視できるようにすることです。今回の取り組みにおいては、レイヤーマスク、シェーダー、Renderer Feature の組み合わせを使用します。最初の一歩としては、**Custom/StencilMask** というシェーダーを使用して、レンズで使用されるマテリアル（この場合は、**MaskMat** というマテリアル）を変更することからになります。

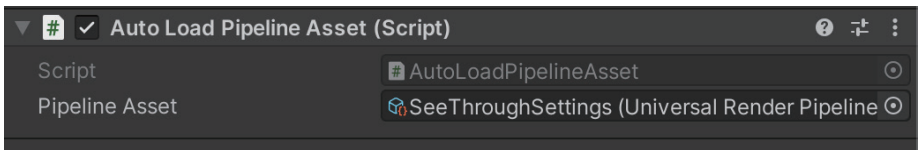
```
Shader "Custom/StencilMask"
{
       Properties{}

       SubShader{

              Tags {
                     "RenderType" = "Opaque"
              }

              Pass {
                     ZWrite Off

                     HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "Packages/com.unity.render-pipelines.
universal/ShaderLibrary/Core.hlsl"

            struct Attributes
            {
                float4 positionOS   : POSITION;
            };

            struct Varyings
            {
                float4 positionHCS  : SV_POSITION;
            };

            Varyings vert(Attributes IN)
            {
                Varyings OUT;

                OUT.positionHCS = TransformObjectToHClip(IN.
positionOS.xyz);

                return OUT;
            }

            half4 frag() : SV_Target
            {
                return (half4)0;
            }

                     ENDHLSL
              }
       }
}
```

Custom/StencilMask では、コマンド `ZWrite Off` があることに注意するようにしてください。大半の場合、オブジェクトに対して `ZWrite Off` を設定していると、オブジェクトが消失してしまいます。オブジェクトのレンダリング順序が変更されて、シーンより前にレンダリングがされるからです。レンダーキューのインデックスをジオメトリより大きい値に変更すると、また表示されるようになります。この例では、ジオメトリ値の 2000 のままになっています。

レンズで実行するアクションは、ステンシルバッファへの値の書き込みだけです。カラーバッファにはステンシルで書き込むものの、シェーダーの出力には書き込まないことを考慮する必要があるため、カラーの書き込みは無効にすることができます（`ColorMask 0`）。シーンはレンズマスクより前にレンダリングされるので、これはやや最適化に向けたアプローチです。ディファードレンダリングパスと連携させる場合には特に最適化されます。

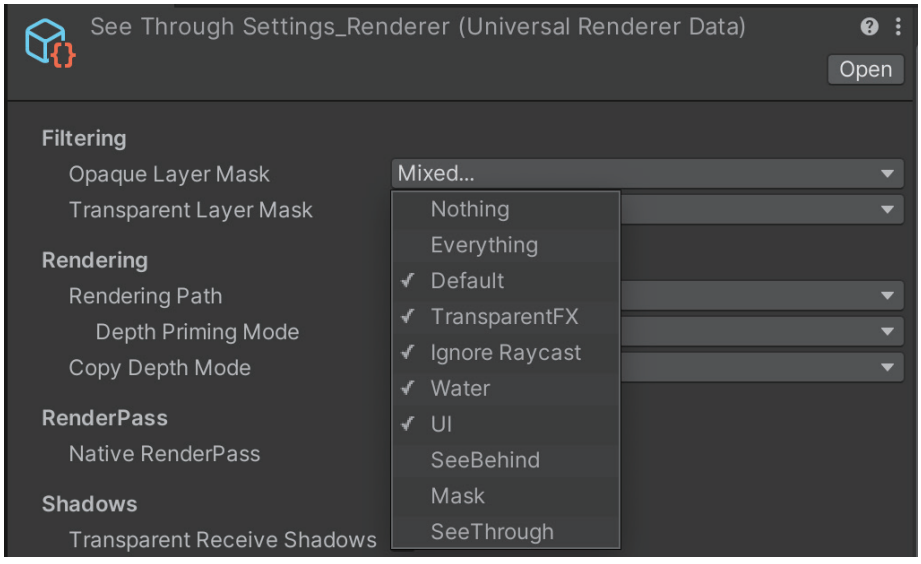
この例では、**Mask** と **SeeThrough** の 2 つのカスタムレイヤーを使用します。レンズは Mask レイヤーにあり、デスク（ただしその子は除く）は SeeThrough レイヤーにあります。

このシーンでは、シーンファイル、マテリアル、およびシェーダーと同じフォルダー（**「Scenes」>「Renderer Feature Stencils」**）にある **See Through Settings_Renderer** という名前の Renderer Data オブジェクトを使用しています。メインカメラにアタッチされたスクリプト（**「Auto Load Pipeline Asset」**）により、**「Project Settings」>「Graphics」**でこれが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ことが確認できます。ここで、このアセットの設定を確認しましょう。



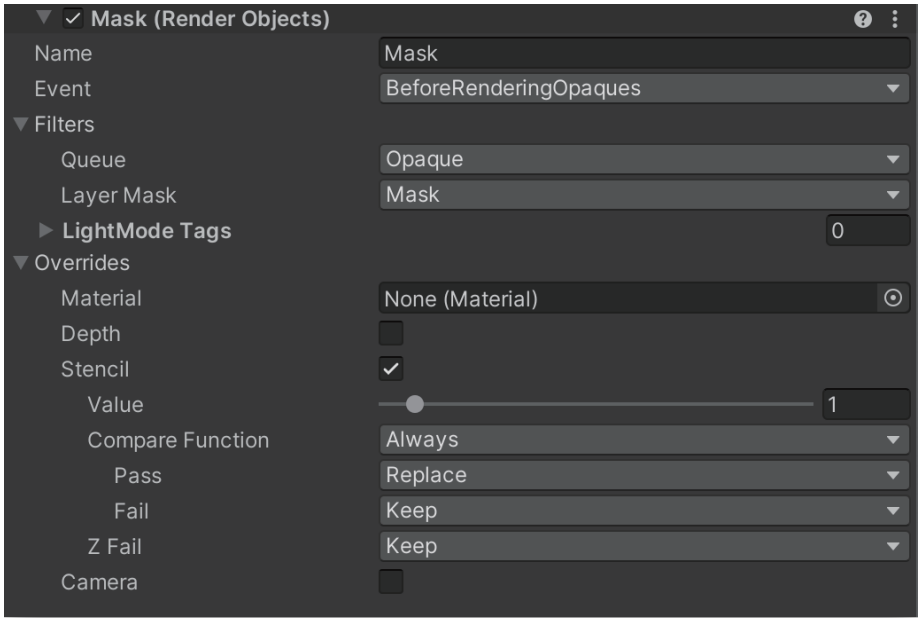
「Main Camera」>「Auto Load Pipeline Asset script」に設定されたパイプラインアセット

「Scenes」>「Renderer Feature Stencils」から**「SeeThrough Settings_Renderer」**を選択します。デフォルトから変更された最初の設定は**「Opaque Layer Mask」**です。これは Mask と SeeThrough を除外することに注意してください。



「See Through Settings_Renderer」で「Opaque Layer Mask」の変更

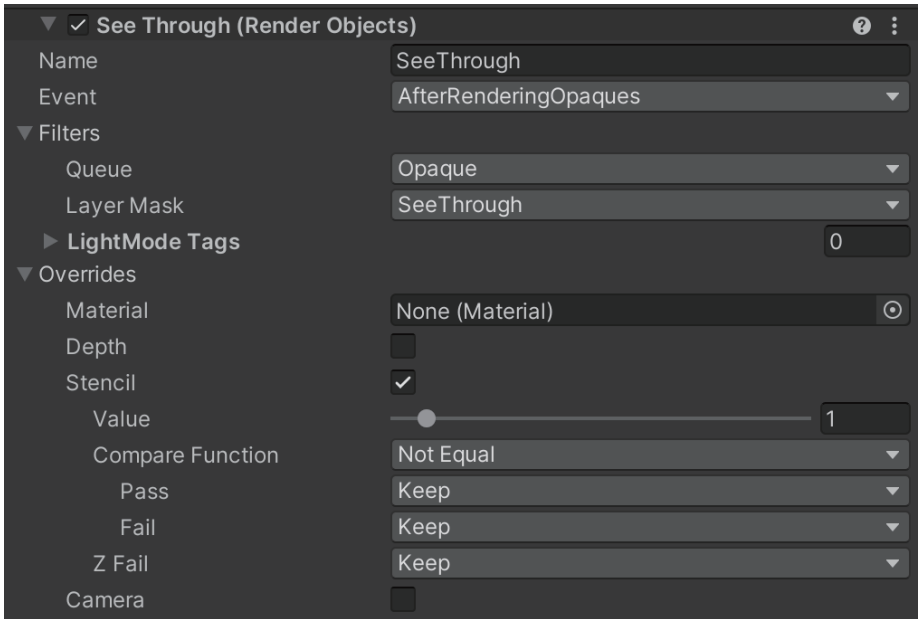
Inspector の Renderer Feature のリストに、「**Mask**」と「**SeeThrough**」という名前の 2 つの Render Objects 機能があります。「SeeThrough」オプションを無効にすると、デスクが消えてしまいます。この状況が発生するのはなぜかと言うと、Opaque Layer Mask が使用されているのはあくまでフィルターで除外されたレイヤーの一部であって、デフォルトレンダリングの一部ではないからです（Render Objects 機能を元にレンダリングされているだけです）。



Mask（Render Objects）の設定

上の画像は、Mask がイベント **BeforeRenderingOpaques** を使用する設定で、Mask レイヤーにレンダリングされたピクセルに対してのみ動作するようにフィルター処理するようにした状態になっています。「Overrides」パネルの「**Stencil**」オプションが有効です。バッファーに保存される値は 1 です。この書き込みが確実に行われるようにするために、「**Compare Function**」は「**Always**」に設定され、「**Pass**」は「**Replace**」に設定されているため、既存の値は常に置換されるようになっています。「**Fail**」と「**Z Fail**」は「**Keep**」に設定されています。

URP は、Mask レイヤーのレンダリングを試みます。オーバーライドマテリアルは設定されていないため、この Mask レイヤーのオブジェクトによって定義されたマテリアルを使用するようになります。MaskMat マテリアルと StencilMask シェーダーを持っているだけのレンズです。「Compare Function」を「**Always**」に、「Pass」を「**Replace**」に設定することで、レンズがビジョン内にある場所はどこでもステンシルバッファになり、各ピクセルの値が 1 に設定されるようになります。



See Through（Render Objects）の設定

上掲の画像中の、2 つ目となる Render Objects Renderer Feature を見てみましょう。これはイベント「**AfterRenderingOpaques**」を使用するように設定されており、つまりはステンシルバッファが設定された後に適用されることになります。その「**Layer Mask**」は「**SeeThrough**」に設定され、「**Value**」は「**1**」に設定されています。値 1 が検出された場合、ピクセルはレンダリングされません。

「Compare Function」設定は「**Not Equal**」に設定され、「Pass」、「Fail」、「Z Fail」はすべて「**Keep**」に設定されています。この Render Objects パスはステンシルバッファから読み取られるだけで、そこには書き込まれません。そのため、このパスは、ステンシルバッファに値 1 が含まれないレイヤー See Through の任意のピクセルをレンダリングするようになり、レンズがある場所のみデフォルトのレンダリングのままにします。Compare Function を「**Equal**」に変更して結果をあべこべにして、デスクがレンズにのみ表示されるようにしてみましょう。

「Compare Function」を「Equal」に変更した場合の効果

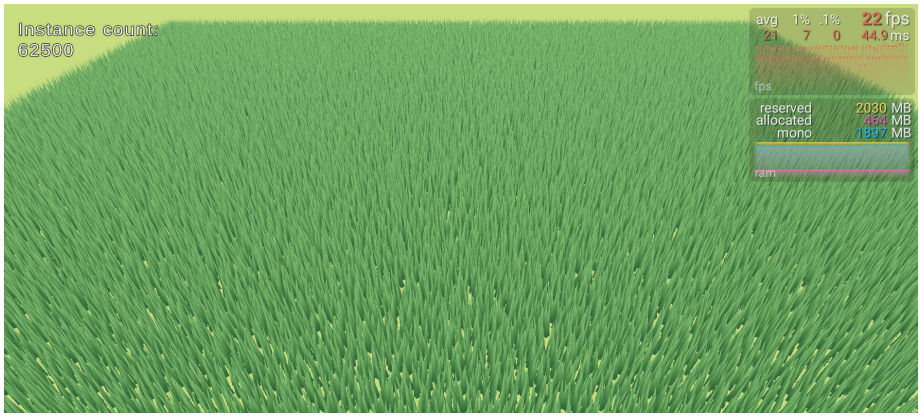
Renderer Feature は、ドラマチックなカスタムエフェクトを実現するための優れた方法なのです。

インスタンシング

CPU と GPU の間でのデータのやりとりは、レンダーパイプラインの多大なボトルネックとなっています。同じジオメトリとマテリアルを使って何度もレンダリングしなければならないモデルがある場合には、Unity はそれを行うための優れたツールを提供しています。この章では、それらについて説明します。

一面に広がる草原を使用して、インスタンシングの概念を描写していきます。これはフォトリアリスティックとはかけ離れていますが、関連する手法の形で描写するには十分なものです。**Scenes > Instancing** フォルダーに例があります。

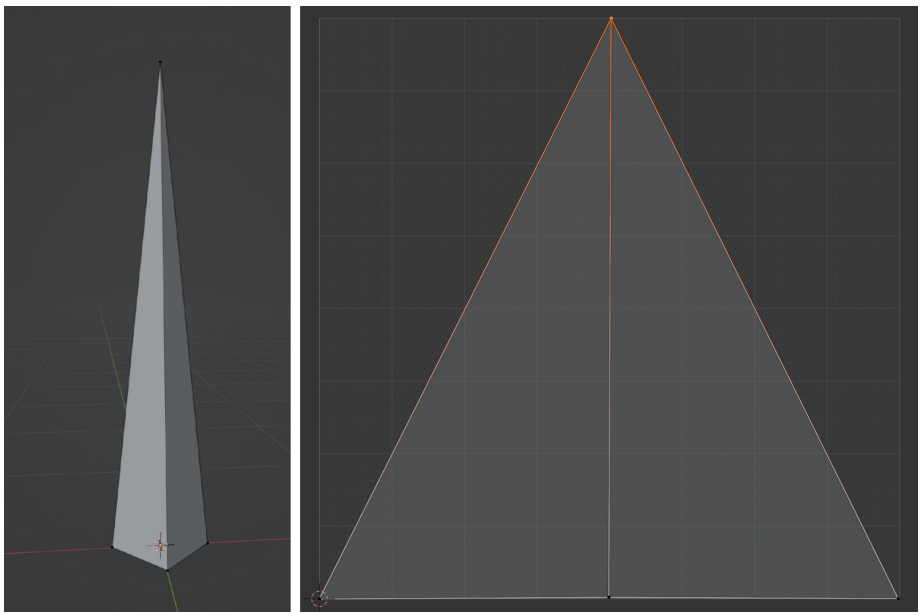
> **ノート：**アセットに関する記事『Making Grass in Unity with GPU Instancing』の著者に感謝します。



SRP Batcher 互換マテリアルを使用してレンダリングされた草原

HoYoverse による人気の Made with Unity ゲーム、『*原神*』は、植物が青々と茂る広大なオープンワールドが特徴です。モバイルデバイスから最新のコンソールまで、あらゆる主要なプラットフォームのすべてで稼働します。このセクションでは、優れたアプローチで同様の草エフェクトを再現するやり方のヒントを提供しています。

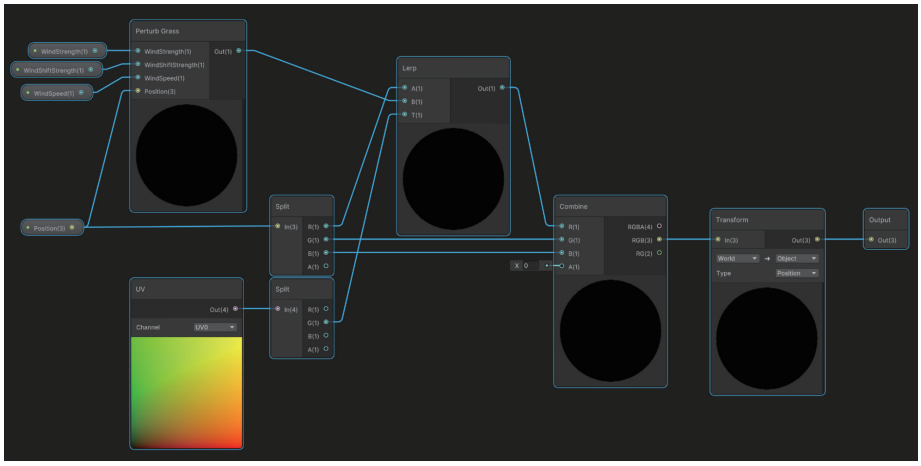
まず、シンプルに保つために、1 枚の草の葉と 2 つの三角形が必要です。各草の葉の根元の **V** 値が **0** で、先端の V 値を **1** にして、UV を設定します。こうやって使うことで、先端の頂点をオフセットして風をシミュレートできます。



草の葉モデルと UV

フォルダー **Scenes > Instancing > Common > Grass Wave** にあるシェーダーグラフのサブグラフを見てみましょう。この目的は、WindSpeed、WindShiftStrength、WindStrength に基づいてオブジェクトの頂点の X 値を無秩序にすることです。すべての草の葉がわずかに異なる動作をするように、サブグラフ Perturb Grass で **Noise** ノードが使用されています。頂点 Y および Z の位置は出力に直接渡されますが、X 値のオフセットは **Lerp** ノードを使用して処理されます。

補間を制御する T 入力は、UV の V 値から取得されます。草の葉の根元では、これは 0 であり、線形補間の結果が、モデル化された位置である線形補間の入力 A になることを意味します。葉の先端の V は 1 であり、線形補間の結果は、処理されたオフセットである入力 B となります。



Grass Wave サブグラフ

それぞれの葉を変形するメソッドができました。次はこれを完全なシェーダーに変えます。草の、それぞれの葉をマテリアルシェーダーとして使えるようになります。

フォルダー **Scenes > Instancing > 1 - SRP Batcher > SRP Batcher Shader** を見てみましょう。これはシンプルなシェーダーで、Grass Wave サブグラフが Vertex > Position を制御し、Sample Texture 2D がフラグメントシェーダーの通常色入力として機能しています。

ここで、以下のコードを使用して、草原を追加しましょう。

```
_startPosition = -_fieldSize / 2.0f;
_cellSize = new Vector2(_fieldSize.x / GrassDensity,
_fieldSize.y / GrassDensity);

var grassEntities = new Vector2[GrassDensity, GrassDensity];
var halfCellSize = _cellSize / 2.0f;

for (var i = 0; i < grassEntities.GetLength(0); i++) {
for (var j = 0; j < grassEntities.GetLength(1); j++) {
```

```
            grassEntities[i, j] =
                  new Vector2(_cellSize.x * i + _
startPosition.x,
                              _cellSize.y * j + _
startPosition.y) +
new Vector2( Random.Range(-halfCellSize.x, halfCellSize.x),
                               Random.Range(-halfCellSize.y,
halfCellSize.y));
      }
}
_abstractGrassDrawer.Init(grassEntities, _fieldSize);
```

このコード例をさらによく見てみると、以下のことがわかります：

- `_fieldSize` は (40, 40) です
- `_startPosition` は (-20, -20) です
- `GrassDensity` は Github サンプルで 250 に設定されています
- `cellSize` は (0.16, 0.16) です
- 2 つのループが `_grassEntities` 2D 配列の各要素の設定を繰り返しています
- それぞれの葉の根元の位置は、`_startPosition` に現在のセルを加算してから、ランダム係数が導入されています
- `_abstractGrassDrawer` は、草を追加するコードの使用の 2 つのバージョンの基本クラスです
  - 初期バージョンでは、GPU インスタンシングを無視し、シーン **Scenes > Instancing > 1 - SRP Batcher > 1 - SRP** を開いて実行することで SRP Batcher が問題をどの程度うまく処理するかを確認します
  - まず、`grassEntities` 2D 配列の各位置で、草の葉モデルプレハブをシーンに追加する必要があります。コードはファイル **Scenes > Instancing > Scripts > GameObjectGrassDrawer.cs** にあります

```
public override void Init(Vector2[,] grassEntities, Vector2
fieldSize) {
_grassEntities = new GameObject[grassEntities.GetLength(0),
     grassEntities.GetLength(1)];
      for (var i = 0; i < grassEntities.GetLength(0); i++) {
            for (var j = 0; j < grassEntities.GetLength(1);
j++) {
                  _grassEntities[i, j] =
Instantiate(_grassPrefab,
                                        new Vector3(
    grassEntities[i, j].x,
    0.0f,
```

```
    grassEntities[i, j].y),
     Quaternion.identity);
             }
       }
}
```

ここで、インスタンス化を使用して grassEntities 配列を反復処理し、割り当てられたプレハブから新しいゲームオブジェクトを作成します。これは動作しますが、シーンのフレームレートに著しい影響を及ぼします。ページ 15 の草原の画像から、以下の仕様の 2020 iMac で実行した場合、62,500 枚の葉に対してフレームレートが 22 fps という低速であることがわかります。

— Retina 5K、27 インチ、2020

— プロセッサー：3.8 GHz 8-Core Intel Core i7

— メモリ：32 GB 2667 MHz DDR4

— 起動ディスク：Macintosh HD

— グラフィックス：AMD Radeon Pro 5500 XT 8 GB

シーンはどのようにして最適化できるでしょうか？

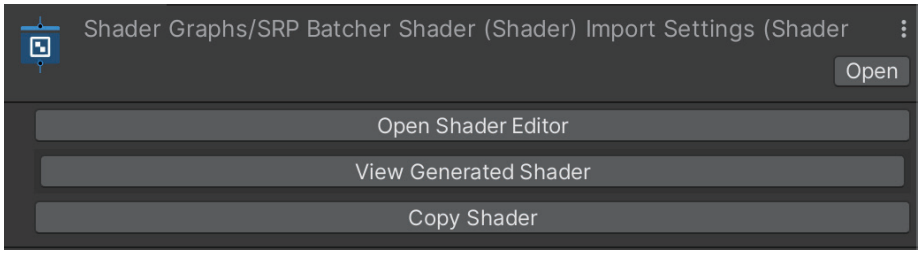
> **ノート：**正方形以外の Terrain（地形）の場合、リスト内のそれぞれの葉の位置を保存する描画ツールを作成できます。例えば、このブログ記事では、シーンをクリックするたびにオブジェクトを配置する操作を効率化するツールの構築方法を説明しています。

GPU インスタンシング

1 つの最適化手法としては、GPU インスタンシングを有効にすることです。この手法の例については、GitHub サンプルの **Scenes > Instancing > 2 - GPU Instancing > 2 - GPU Instancing** を参照してください。

「**Enable GPU Instancing**」というマテリアル設定は、同じマテリアルを使用するモデルをバッチ化し、ドローコールの回数を削減するようレンダラーへ指示するものです。設定は「Advanced Options」パネルで利用できます。

SRP Batcher と GPU インスタンシングは、お互いがそれぞれ排他的になっています。URP を使用している場合、マテリアルに SRP Batcher との互換性があると、「Enable GPU Instancing 」が選択されていても SRP Batcher が使用されます。シェーダーグラフで作成されたシェーダーは、デフォルトで SRP Batcher と互換性があります。SRP Batcher の互換性を無効にするには、HLSL シェーダーを作成するシェーダーグラフを選択し、Inspector で「**View Generated Shader**」をクリックします。



シェーダーグラフからの HLSL シェーダーの生成

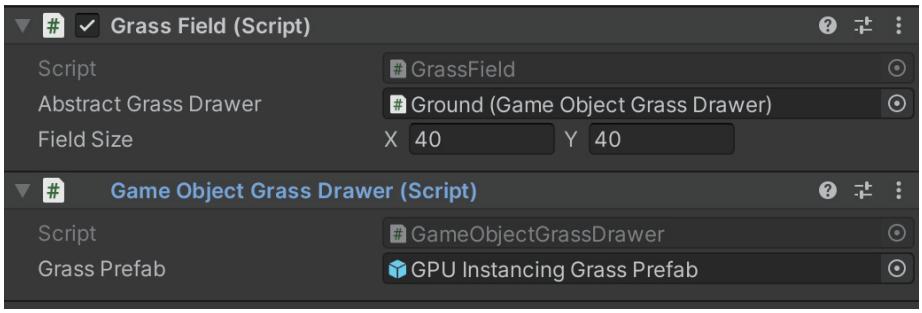
シェーダーが作成され、Temp フォルダーに配置されて、Visual Studio または選択したコードエディターで開かれます。シェーダーの名前を以下の名前に変更します：

シェーダー "Custom/GPU Instancing Shader"

CBUFFER を検索し、CBUFFER マクロをコメントにします：

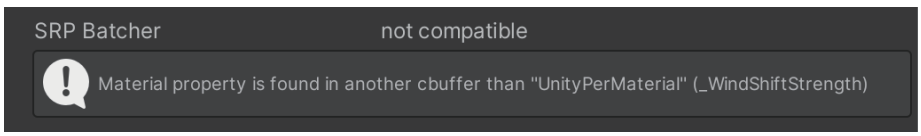
```
// グラフのプロパティ
//CBUFFER_START(UnityPerMaterial)
            float4 _MainTexture_TexelSize;
            half _WindShiftStrength;
            half _WindSpeed;
            half _WindStrength;
//CBUFFER_END
```

シェーダーをアセットに保存します。



GPU インスタンシングのシーンにおいて Ground ゲームオブジェクトへ割り当てられたスクリプト

GPU インスタンシングのシーンでは、SRP Batcher シーンと **Abstract Grass Drawer** は同じバージョンを使用することになるのには注意するようにしてください。唯一の違いは、GPU インスタンシングの **GameObjectGrassDrawer** バージョンが、GPU インスタンシングシェーダーが使用されたマテリアルを持っている、別のプレハブに割り当てられていることです。



GPU インスタンシングシェーダーは SRP Batcher との互換性がない

Inspector で GPU インスタンシングシェーダーを確認すると、それが SRP Batcher と互換性のないことがわかります。

コードの生成のために使用したグラフをどういった形であれ変更した場合は、以下のようにカスタマイズを行う手順を繰り返すようにしなければなりません：

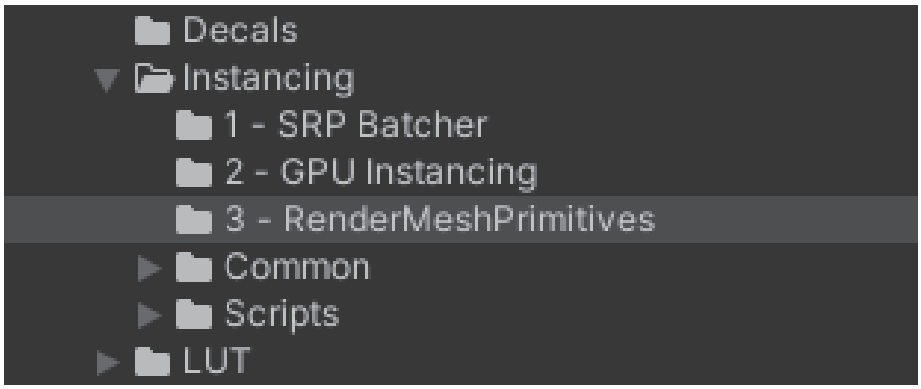
1. 生成されたシェーダーを表示するか、再生成します
2. シェーダーの名前を編集します
3. CBUFFER マクロをコメントにします
4. アセットに保存します

しかし、この全作業の後、テストを行ったとしても結果は SRP Batcher のみがほんの少しだけ向上したことが示されるのみとなります（おそらく CPU 依存のため）。もっとよい方法が必要です。

RenderMeshPrimitives

Unity Graphics API には、ゲームオブジェクトで必要となるものを別のアプローチを使うことで、メッシュを直接レンダリングする方法は数多くあります。ここで使用したメソッドは、Unity LTS 2021 で導入された機能である RenderMeshPrimitives です。これより前は、今では古いものとしてマークされている DrawMeshInstancedProcedural を使用する必要がありました。

RenderMeshPrimitives では、ComputeBuffer を使用して、個々のメッシュ位置を情報元としたマテリアルを扱う必要があります。シーン **Scenes > Instancing > 3 - RenderMeshPrimitives > 3 - RenderMeshPrimitives** を表示することで、その動作を確認できます。



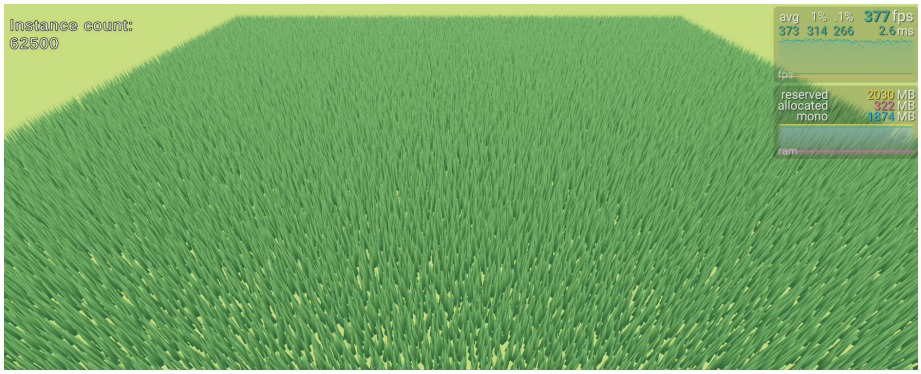
Project ウィンドウのインスタンシングシーン

以下の草原の画像からわかるように、フレームレートの向上は、まさに注目に値する 377 fps となっています。SRP Batcher と GPU インスタンシングで作成されたシーンは、約 20 fps で実行されていました。

このケースにおいての違いは、草原が単一のドローコールを使用してレンダリングしていることです。

Event #4: Draw Mesh (instanced)	(1 draw calls, 62500 instances)
Shader	Shader Graphs/Instanced Grass Shader Graph, SubShader #0
Pass	Universal Forward
Keywords	PROCEDURAL_INSTANCING_ON
Blend	One Zero
ZClip	True
ZTest	LessEqual
ZWrite	On
Cull	Back
Conservative	False

草原のフレームデバッガー統計



RenderMeshPrimitives を使用してレンダリングされた草原

これは、それぞれの葉の位置を **Material** プロパティで作ることで実現できます。葉をレンダリングするためのデータは、その並列化を使用して最適なスピードで草原全体をレンダリングする GPU に存在します。

位置を生成するコードを改めて見ていきましょう。これはファイル **Scenes > Instancing > Scripts > InstancedGrassDrawer.cs** 内の `UpdatePositions` メソッドにあります。

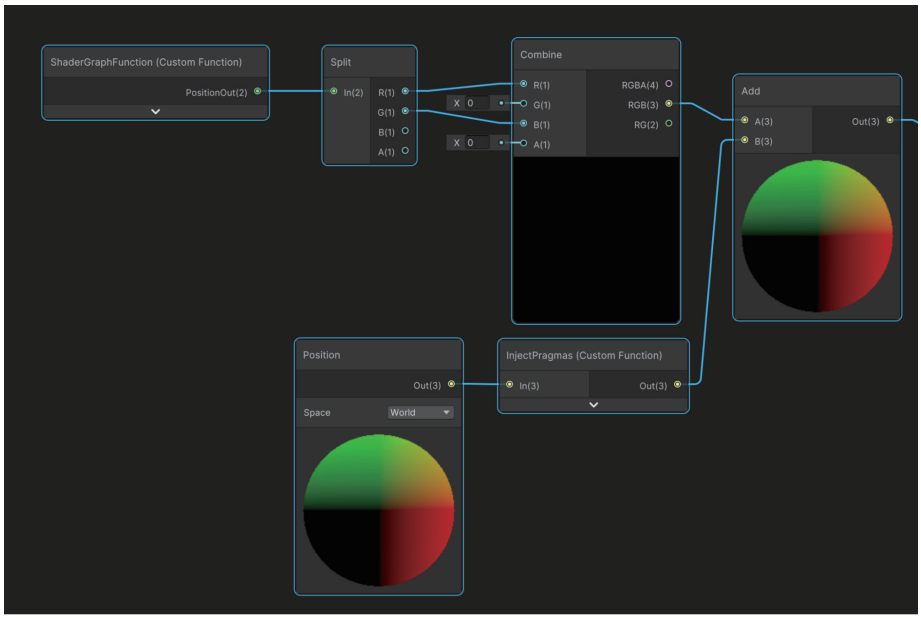
```
_positionsCount = _positions.Count;
_positionBuffer?.Release();
if (_positionsCount == 0) return;
_positionBuffer = new ComputeBuffer(_positionsCount, 8);
_positionBuffer.SetData(_positions);
_instanceMaterial.SetBuffer(Shader.PropertyToID("PositionsBuff
er"), _positionBuffer);
```

`_positions` は草の位置の Vector2 リストを保持します。`_positionsBuffer` が存在する場合は、それをリリースします。変数の後ろにある「?」についてよく知らない方のために説明すると、これは null チェックであり、以下の省略形を意味しています：

```
if (positionsBuffer != null) _positionsBuffer.Release()
```

カウントパラメーターと各項目のバイトサイズを取得する ComputeBuffer を作成します。Vector2 には 2 つの Float が含まれます。単一の Float は 32 ビット（4 バイト）で、2 つの Float では 8 バイトになります。SetData を使用して _positions リストを渡すことで ComputeBuffer を設定するのは簡単です。ここで、SetBuffer メソッドを使用して、これをマテリアルにコピーできます。マテリアル内のこのバッファには、positionsBuffer という名前を使用してアクセスします。

Scenes > Instancing > 3 - RenderMeshPrimitives > Instanced Grass Shader にあるグラフを見てみましょう。



Compute Buffer からの頂点位置の取得

画像中の下の部分から見ていくと、Grass Mesh 頂点位置の **Space** パラメーターが「**World**」に設定されていることがわかります。しかし、この手法を使用するたびに追加する必要のある重要なコードブロックがあります：RenderMeshPrimitive を使用してレンダリングされるメッシュには #pragma が必要です。これは、カスタム関数を使用することで実行できます。ファイルから関数を調達する代わりに、次の文字列を追加します。

```
#pragma instancing_options procedural:ConfigureProcedural
Out = In;
```

位置の値を生成するために現在このシェーダーで使用されているコードメソッドは、ConfigureProcedural という名前の関数から取得されます。それ以外にも、この Custom Function ノードはその入力である In をその出力である Out に単純に渡してくれます。

手間のかかる作業は ShaderGraphFunction というカスタム関数内で行われます。この関数は、シーンファイルと同じフォルダー内のファイル InstancedPosition にあります。

```
#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
StructuredBuffer<float2> PositionsBuffer;
#endif

float2 position;

void ConfigureProcedural () {
	#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
	position = PositionsBuffer[unity_InstanceID];
	#endif
}

void ShaderGraphFunction_float (out float2 PositionOut) {
	PositionOut = position;
}
```

位置は ConfigureProcedural メソッドを使って設定され、スクリプトに Float バージョンと Half バージョンがある ShaderGraphFunction を使用して、出力へと渡されます。

グラフのこのポイントで、個々の葉の位置は float2 であり、最初の Float は X 値、2 つ目は Z です。**Split** ノードを使用して、これを個々の Float へと変換し、**Combine** ノードを使用して 2 つ目の Float を 3 つ目に移動します。Split および Combine ノードは、XYZW ではなく個々の Float RGBA を呼び出しますが、G を B に移動することで、実質的に Y を Z に移動します。葉と頂点位置が設定され、これらを組み合わせて実際のワールドの頂点の位置を取得できます。

このシェーダーの準備ができると、入力 WindSpeed、WindStrength、WindShiftStrength、そして MainTexture を持つマテリアルと合わせて使用できます。これは SRP Batcher および GPU インスタンシングのバージョンで使用されるのと同じものです。違いは、各頂点の位置の算出方法だけです。草の葉をレンダリングする方法については、もう一度スクリプト InstancedGrassDrawer.cs を参照してください。スクリプト内の変数は、GrassField.cs スクリプトの Awake メソッドによって呼び出される Init メソッドで初期化されます。

```
public override void Init(Vector2[,] grassEntities, Vector2
fieldSize) {

_grassEntities = grassEntities;
	_grassBounds = new Bounds(transform.position,
			new Vector3(fieldSize.x, 0.0f,
fieldSize.y));
	_positions = new List<Vector2>();
	_renderParams = new RenderParams(_instanceMaterial);
	_renderParams.worldBounds = _grassBounds;
	_renderParams.shadowCastingMode = ShadowCastingMode.Off;
}
```

Graphics.RenderMeshPrimitives を使用するには、RenderParams インスタンスが必要です。これは、割り当てられたマテリアルである _instanceMaterial から作成されます。他にも 2 つのプロパティが追加で割り当てられます。

実際のレンダリングは Update コールバックを使用して行われます：

```
private void Update() {
       if (_positionsCount == 0) return;
       Graphics.RenderMeshPrimitives(_renderParams, _instanceMesh, 0,
_positionsCount);
}
```

RenderMeshPrimitives は 4 つのパラメーター（RenderParams インスタンス、レンダリングするメッシュ、サブメッシュインデックス、レンダリングするコピーの数を識別するカウント値）を受け取ります。シェーダーを使用する場合、各コピーは一意の unity_InstanceID を持ちます。その値は 0 から count -1 です。

ComputeBuffer を使用したレンダリングは高速で、設定がかなりシンプルです。_positionBuffer を操作することで、草を刈り取ったり、吹き飛ばしたりできます。CPU と GPU 間のデータの受け渡しを回避するために、これは ComputeShader で最適に処理されます。

その他のリソース

— このレシピで扱ったアセット

— DrawMeshInstancedIndirect を使用したプロジェクトの例

— GPU インスタンシングのドキュメント

— GPU インスタンシングに関する記事（CatLikeCoding ）

— ComputeBuffers を使用したインスタンシング

トゥーンシェーディングとアウトラインシェーディング

このレシピは、トゥーンシェーダーとアウトラインシェーダーを作成する一般的な方法に基づいています。

1 つのシーンでの 3 つの異なる外観（スタンダードシェーディング（左）、ポストプロセス（中央）、マテリアルごとのシェーディング（右））

一緒に使用されることの多いトゥーンシェーダーとアウトラインシェーダーは、まったく異なる 2 つの課題を提示してきます。トゥーンシェーダーは、URP 互換の Lit シェーダーを使用して作成される色を受け取り、連続するグラデーションを許可する代わりに出力に傾斜を付けるので、カスタムライティングモデルが必要です。この例では、シェーダーグラフを使用して作成されます。ただし、シェーダーグラフではカスタムライティングがサポートされないので、Main lights と Additional lights へ直接アクセスするために使用できるノードはありません。代わりに、カスタムノードを利用してこれらにアクセスできます。

Roll7 による Made with Unity の三人称視点アクションシューティングゲーム、『*ローラードローム（Rollerdrome）*』には、ゲームを漫画のように見せる独特なアートディレクションがあります。これはセルシェーディング手法で実現されています。こちらのクリエイターインタビューをぜひご覧ください。

シェーダーがどのように見えるかを確認するには、**Scenes > Toon Shading > Simple Toon Shading** に移動します。

シンプルな傾斜の付いたトゥーンシェーダーのあるシーン

トゥーンシェーダー

この例では、シンプルに保つためにメインライトとテクスチャ付きメッシュのみサポートします。アウトライン、追加のライト、グローバルイルミネーション、またはライトマッピングは含まれません。これらの機能については、この章で後ほど説明します。

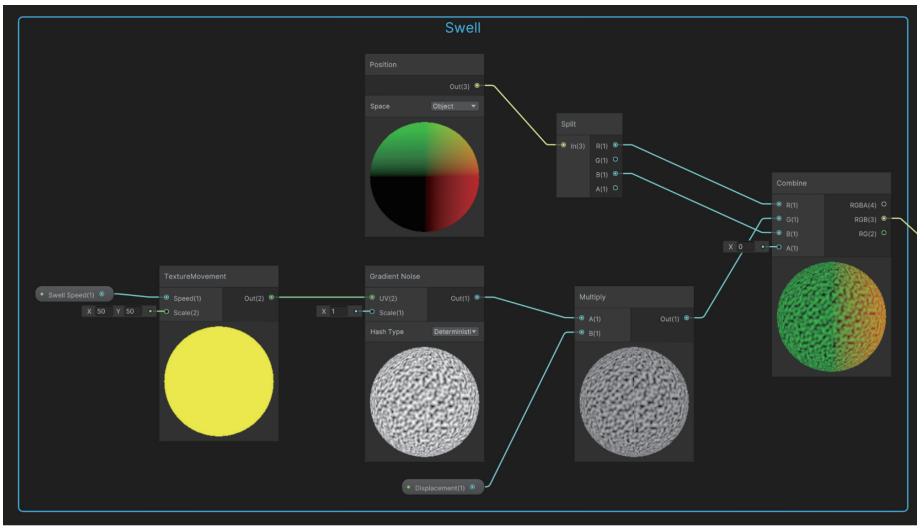
まず、ファイル **HLSL > Custom Lighting.hlsl** にあるカスタム関数を使用してメインライトにアクセスしましょう。

```
void MainLight_float(float3 WorldPos, out float3 Direction, out
float3 Color, out float DistanceAtten, out float ShadowAtten)
{
#ifdef SHADERGRAPH_PREVIEW
    Direction = float3(0.5, 0.5, 0);
    Color = 1;
    DistanceAtten = 1;
    ShadowAtten = 1;
#else
       float4 shadowCoord = TransformWorldToShadowCoord(WorldP
os);

    Light mainLight = GetMainLight(shadowCoord);
    Direction = mainLight.direction;
    Color = mainLight.color;
    DistanceAtten = mainLight.distanceAttenuation;

       #if !defined(_MAIN_LIGHT_SHADOWS) || defined(_RECEIVE_
SHADOWS_OFF)
              ShadowAtten = 1.0h;
       #else
         ShadowSamplingData shadowSamplingData = GetMainLightSha
dowSamplingData();
         float shadowStrength = GetMainLightShadowStrength();
         ShadowAtten = SampleShadowmap(shadowCoord, TEXTURE2D_
ARGS(_MainLightShadowmapTexture,
         sampler_MainLightShadowmapTexture),
         shadowSamplingData, shadowStrength, false);
    #endif
#endif
}
```

シェーダーグラフアセットを作成しながら動作を定義する `#ifdef SHADERGRAPH_PREVIEW` プリプロセッサーディレクティブの内部にコードブロックを追加することをおすすめします。これにより、グラフプレビューウィンドウでデフォルトにする値を指定します。

WorldPos は関数 `TransformWorldToShadowCoord` を使用してシャドウ座標に変換されます。このコードで使用される関数はユニバーサルレンダーパイプライン（URP）パッケージからのものであり、シェーダーグラフのカスタム関数に使用できます。関数 `GetMainLight` が float4 で使用される場合、返されるライトの **ShadowAttenuation** プロパティが設定されます。これは、このカスタム関数を使用するグラフで必要になります。

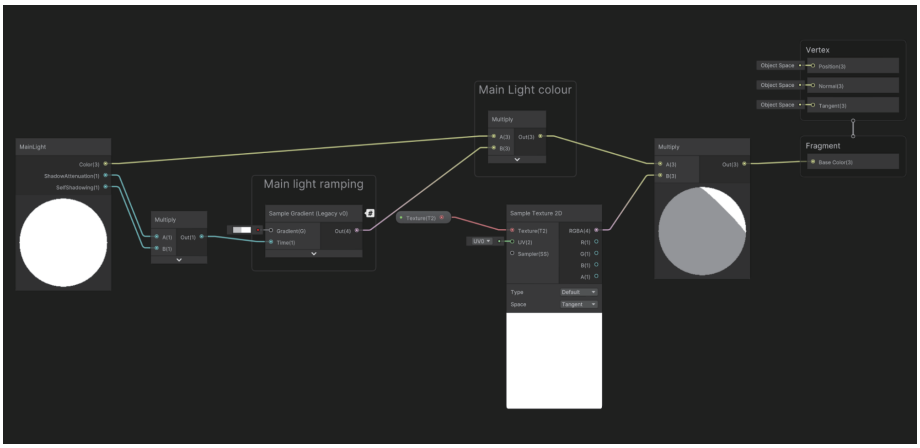
このコードは、フォルダー **Shaders > Subgraphs** にあるメインライトサブグラフ（下の画像を参照）で使用されます。これをよく見ていきましょう。



メインライトサブグラフ

Custom Function ノードは、Absolute World に設定された **Position** ノードを唯一の入力として受け取ります。関数は、Direction、Color、DistanceAtten（未使用のまま）、および ShadowAtten を返します。セルフシャドウイングを可能にするには、ライト方向とワールド法線のドット積を取得し、これを 0 から 1 の間に固定します。負の値は望ましくありません。

メインライトにアクセスする方法ができたので、それを使用してシンプルなトゥーンシェーダーを作成できます。**Shaders > Simple Toon** を参照して、グラフ（そして下の画像）を確認しましょう。



シンプルなトゥーングラフ

最初のノードはメインライトサブグラフです。ShadowAttenuation 出力と SelfShadowing 出力を乗算します。秘訣は、この出力を傾斜付きのグラデーションと連携する **Sample Gradient** ノードに渡して、光量が滑らかにならず、グラデーションに基づいて段階的に跳ね上がるようにすることです。

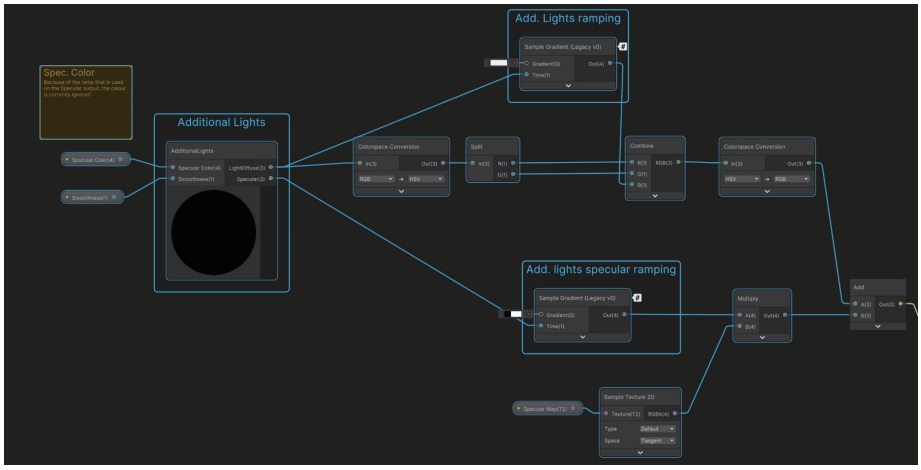
滑らかに変化する入力を受け取ってそれをグラデーションで処理することは、シェーディングの多くの課題に役立つ手法です。グラフの残りの部分では、光の色を傾斜レベルと組み合わせてから、これをサンプリングされたテクスチャと組み合わせて、通常色に使用する色を生成します。

このシンプルなグラフの制限事項は、グローバルイルミネーションと追加のライトを考慮しないことです。グラフ **Shaders > Toon Shader** にはこれらの機能がすべてあり、アウトラインエフェクトを追加します。アウトラインの追加方法を説明する前に、追加のライトにアクセスする方法を見てみましょう。

この場合も、カスタム関数が必要です。これはメインライト **HLSL > CustomLighting.hlsl** 用のものと同じ HLSL ファイル内にあります。

```
void AdditionalLights_float(float3 SpecColor, float Smoothness,
float3 WorldPosition, float3 WorldNormal, float3 WorldView, out
float3 Diffuse, out float3 Specular)
{
    float3 diffuseColor = 0;
    float3 specularColor = 0;

#ifndef SHADERGRAPH_PREVIEW
    Smoothness = exp2(10 * Smoothness + 1);
    WorldNormal = normalize(WorldNormal);
    WorldView = SafeNormalize(WorldView);
    int pixelLightCount = GetAdditionalLightsCount();
    for (int i = 0; i < pixelLightCount; ++i)
    {
        Light light = GetAdditionalLight(i, WorldPosition);
        half3 attenuatedLightColor = light.color * (light.
distanceAttenuation * light.shadowAttenuation);
        diffuseColor += LightingLambert(attenuatedLightColor,
light.direction, WorldNormal);
        specularColor += LightingSpecular(attenuatedLightColor,
light.direction, WorldNormal, WorldView, float4(SpecColor, 0),
Smoothness);
    }
#endif

    Diffuse = diffuseColor;
    Specular = specularColor;
}
```

コードには、いくつかの入力（スペキュラー色、Smoothness Float、WorldPosition、WorldNormal、および WorldView 方向）が必要です。これはディフューズカラーとスペキュラカラーの組み合わせを出力します。各追加ライトに対して繰り返し、`LightingLambert` 関数を使用して diffuseColor を累積することで、これを行います。これは、ライトの方向と WorldNormal だけを使用した最もシンプルなライティングモデルです。specularColor は `LightingSpecular` 関数を使用して累積されます。



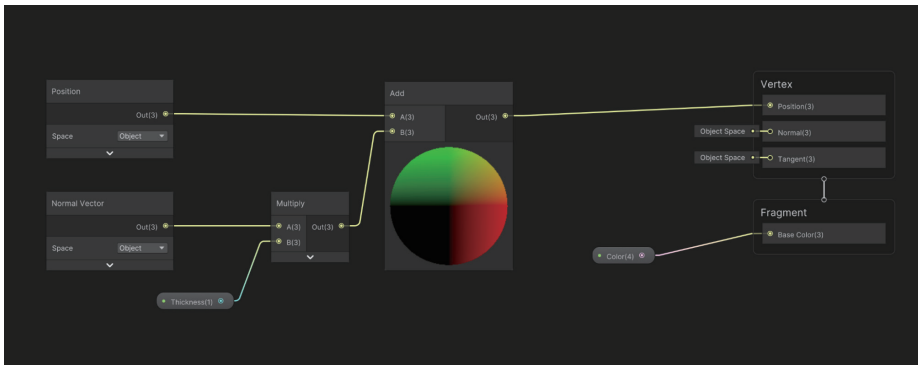
トゥーンシェーダーグラフに追加のライトが入っている状態

上の画像はこのコードがどう扱われているかを表しています。ディフューズレベルは Ramp を使用して調整します。これには他にも有用なテクニックが示されています。色空間を RGB から HSV に変換して、V や B コンポーネントをスケールすることは、ライトレベルの調整に似ており、そうした後に RGB に戻しています。

完成しているトゥーングラフは丹念に学び取る価値があります。これは独自のカスタムシェーダーで扱うことになる有効な手法がたくさん示されているからです。

アウトライン

アウトラインを加える最もシンプルな手法は、裏面を向けているポリゴンのみをレンダリングし、頂点の法線に沿って、その頂点をわずかに移動する頂点シェーダーを使用する 2 つ目のパスを追加することです。シェーダーは、**Shaders > VertexOutline** を介して Github サンプルに含まれています。そのグラフをここに示します。



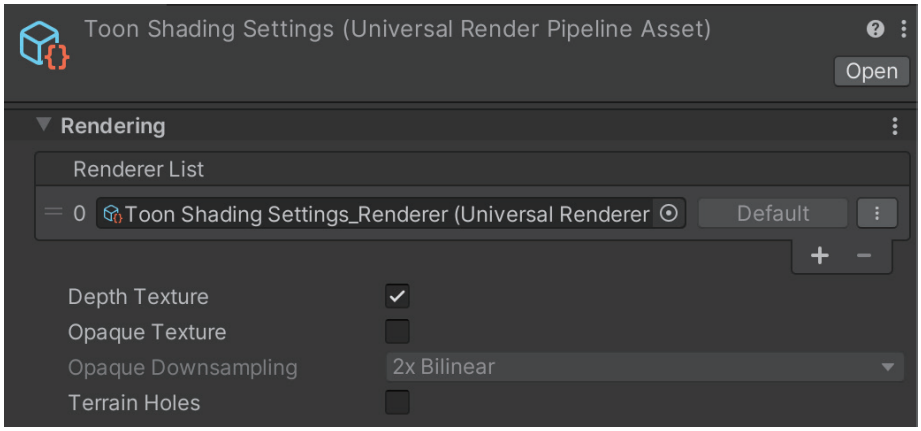
アウトラインシェーダー。裏面が見えているところの頂点を、微動させるという手法を使っている状態です。

Space が Object に設定された **Normal Vector** ノードは、**Multiply** ノードにフィードされます。これは、マテリアルの **Thickness** 値で乗算されます。ここからの出力は Object Position に追加され、頂点位置がオブジェクトのモデル化位置からわずかに移動されます。これは **Vertex Position** プロパティへの入力です。シェーダープロパティ「**Universal**」>「**Render Face**」は、「**Graph Inspector**」>「**Graph Settings**」内のパネルを使用して「**Back**」に設定されます。シェーダーグラフではシングルパスのみ許可されるため、これをレンダリングに追加するには、ゲームオブジェクトインスペクターを使用して 2 つ目のマテリアルを追加する必要があります。

2 つ目のマテリアルの追加

Scenes > Toon Shading > Simple Toon Shading 内のシーンは、使用される 2 つ目のマテリアルを示しています。同じフォルダー内のマテリアル **VertexOutline** を Inspector で表示し、「Thickness」を 0.01 に設定します。

より洗練された手法としては、エッジ検出を使用します。以下に 2 つのバリエーションとして示されていますが、深度テクスチャか法線テクスチャ（もしくはその両方）をたくさんの場所からサンプリングして、エッジとして示されるはっきりした差異を見つけ出すものです。深度テクスチャは、URP アセットの Inspector を使用して提供されます。ここのボックスにチェックマークを入れるということは、赤チャンネルに格納されている深度情報を使って _CameraDepthTexture というテクスチャが作成される、ということを意味しています。



_CameraDepthTexture の作成

エッジ検出のためにトゥーンシェーダーグラフで使用される HLSL スクリプトは、法線もスキャンします。法線の情報はどう取得するのでしょうか？それには、Renderer Feature を使用する必要があります。**Renderer Features > DepthNormalsFeature.cs** を参照し、これが **Toon Shading Settings_Renderer** の Renderer Feature に設定されていることに注目してください。この Renderer Feature は、法線の情報を _CameraDepthNormalsTexture という名前のテクスチャに保存します。

DepthNormalsFeature Renderer Feature によって作成された _CameraDepthNormalsTexture

これら 2 つのテクスチャを入手したところで、アウトラインを生成するには、現在の UV 位置の左、右、上、下の画像をスキャンする必要があります。ファイル **HLSL > Outline.hlsl** を参照すると、トゥーンシェーダーグラフで使用されるコードを確認できます。

```
void OutlineObject_float(float2 UV, float OutlineThickness,
float DepthSensitivity, float NormalsSensitivity, out float
Out)
{
    float halfScaleFloor = floor(OutlineThickness * 0.5);
    float halfScaleCeil = ceil(OutlineThickness * 0.5);

    float2 uvSamples[4];
    float depthSamples[4];
    float3 normalSamples[4];

    uvSamples[0] = UV - float2(_CameraDepthTexture_TexelSize.x,
_CameraDepthTexture_TexelSize.y) * halfScaleFloor;
    uvSamples[1] = UV + float2(_CameraDepthTexture_TexelSize.x,
_CameraDepthTexture_TexelSize.y) * halfScaleCeil;
    uvSamples[2] = UV + float2(_CameraDepthTexture_TexelSize.x
* halfScaleCeil, -_CameraDepthTexture_TexelSize.y *
halfScaleFloor);
    uvSamples[3] = UV + float2(-_CameraDepthTexture_
TexelSize.x * halfScaleFloor, _CameraDepthTexture_TexelSize.y *
halfScaleCeil);

    for(int i = 0; i < 4 ; i++)
    {
        depthSamples[i] = SAMPLE_TEXTURE2D(_CameraDepthTexture,
sampler_CameraDepthTexture, uvSamples[i]).r;
        normalSamples[i] = DecodeNormal(SAMPLE_TEXTURE2D(_
CameraDepthNormalsTexture, sampler_CameraDepthNormalsTexture,
uvSamples[i]));
    }

    // 深度
    float depthFiniteDifference0 = depthSamples[1] -
depthSamples[0];
    float depthFiniteDifference1 = depthSamples[3] -
depthSamples[2];
    float edgeDepth = sqrt(pow(depthFiniteDifference0, 2) +
pow(depthFiniteDifference1, 2)) * 100;
    float depthThreshold = (1/DepthSensitivity) *
depthSamples[0];
    edgeDepth = edgeDepth > depthThreshold ?1 : 0;

    // 法線
    float3 normalFiniteDifference0 = normalSamples[1] -
normalSamples[0];
    float3 normalFiniteDifference1 = normalSamples[3] -
normalSamples[2];
    float edgeNormal = sqrt(dot(normalFiniteDifference0,
normalFiniteDifference0) + dot(normalFiniteDifference1,
normalFiniteDifference1));
```

```
    edgeNormal = edgeNormal > (1/NormalsSensitivity) ?1 : 0;

    float edge = max(edgeDepth, edgeNormal);
    Out = edge;
}
```

現在の UV 位置に float2 値を加算または減算することで、uvSamples 配列が作成される方法に注目してください。これは関数への 1 つの入力です。入力 UV からのオフセットのサイズは **OutlineThickness** プロパティに基づきます。

テクスチャ _CameraDepthTexture および _CameraDepthNormalsTexture をそれぞれサンプリングすることで、depthSamples と normalSamples の配列を作成します。次に、深度と法線の両方について、生成された各サンプル配列の 0 番目の項目を最初の項目から減算すること（`depthSamples[1] - depthSamples[0]`）で difference0 を取得し、2 つ目の項目を 3 つ目の項目から減算することで difference1 を取得します。これらは、左下から右上まで、および右下から左上までの差です。

単純な Float である depthSamples の場合、差を 2 乗してそれらを加算し、平方根を取得してから、これを DepthSensitivity 入力の逆数に基づいて計算された depthThreshold に対してテストします。float3 値である法線の場合、それぞれの差を 2 乗する代わりに、各差とそれ自体のドット積を取得します。ベクトルとそれ自体のドット積は、その大きさの平方根です。これこそまさに必要としているものです。

最後に、関数の出力は edgeDepth または edgeNormal の最大値です。この関数は、このピクセルのアウトラインを表示する場合は 0 を返し、そうでない場合は 1 を返します。グラフに追加し、計算された色を乗算するために使用できます。関数が 0 を返す場合、計算された色は黒に戻ります。

この問題に対する別のアプローチは、ポストプロセスエフェクトを追加することです。**Scenes > Toon Shading > SobelFilter Shading** 内のシーンを見てみましょう。ポストプロセスエフェクトはレンダリングされた出力に対して機能します。BlitMaterialFeature という名前の Renderer Feature が使用されます（Renderer Features フォルダーにあります）。このスクリプトは、Blit 関数を使用してレンダリングされた画像を処理します。オプションで、これはピクセルをコピーするときに使用するマテリアルを取得できるので、各ピクセルをソースから宛先に単純にコピーする代わりに、各ピクセルを処理し、ピクセルの最終的な色を調整できます。

この例で使用されるマテリアルは **SobelFilter** で、シェーダー **Shaders > SobelFilter.shader** を使用します。主な動作は、Sobel フィルターを使用して行われます。

```
float sobel (float2 uv)
{
float2 delta = float2(_Delta, _Delta);

float hr = 0;
float vt = 0;
```

```
hr += SampleDepth(uv + float2(-1.0, -1.0) * delta) *  1.0;
hr += SampleDepth(uv + float2( 1.0, -1.0) * delta) * -1.0;
hr += SampleDepth(uv + float2(-1.0,  0.0) * delta) *  2.0;
hr += SampleDepth(uv + float2( 1.0,  0.0) * delta) * -2.0;
hr += SampleDepth(uv + float2(-1.0,  1.0) * delta) *  1.0;
hr += SampleDepth(uv + float2( 1.0,  1.0) * delta) * -1.0;

vt += SampleDepth(uv + float2(-1.0, -1.0) * delta) *  1.0;
vt += SampleDepth(uv + float2( 0.0, -1.0) * delta) *  2.0;
vt += SampleDepth(uv + float2( 1.0, -1.0) * delta) *  1.0;
vt += SampleDepth(uv + float2(-1.0,  1.0) * delta) * -1.0;
vt += SampleDepth(uv + float2( 0.0,  1.0) * delta) * -2.0;
vt += SampleDepth(uv + float2( 1.0,  1.0) * delta) * -1.0;

return sqrt(hr * hr + vt * vt);
}
```

Sobel フィルターは 3×3 の行列で、レンダリングされた画像の各ピクセルを分析するために機能します。デジタル画像の研究者である Irwin Sobel 氏の名前に由来しています。水平と垂直両方で画像をスキャンし、値を累積して両方のパスの 2 乗和の平方根を返します。各フラグメント / ピクセルの最終的な色は、次のコードを使用して生成されます。

```
half4 frag(Varyings input) : SV_Target
{
float s = pow(1 - saturate(sobel(input.uv)), 50);
half4 col = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.
uv);
return col * s
}
```

Sobel の出力は、HLSL 関数 `saturate` を使用して 0 と 1 の間に固定されます。これを 1 から減算して値を反転し、これを 50 乗します。値を反転するのは、Sobel 関数がエッジでより大きい値を返すからです。このケースでは、エッジでは 0 が必要で、エッジから離れた場所では 1 が必要です。次に、レンダリングされた画像 _MainTex をサンプリングして、この 2 つの値の乗算 col * s を返します。

その他のリソース

— Unity Open Project Github（この章のコードの大部分は Open Project からのものです）

— YouTube 上の Unity Open Project

— YouTube チュートリアル（Ned Makes Games）

— Unity の SobelFilter.shader の使用に関する YouTube チュートリアル（AE Tuts）

— 「Edge detection using a Sobel filter」（Alexander Ameye）

アンビエントオクルージョン

アンビエントオクルージョン

アンビエントオクルージョンは、Unity 2020.2 および URP 10.0 以降のバージョンで使用できるポストプロセス手法です。エフェクトは、近接して連続している溝や、穴、交点、そしてサーフェスを暗くしてくれます。現実世界では、そういった領域はアンビエントライトを遮断、もしくは遮蔽されがちで、そのために暗く見えています。前の画像では、左側がアンビエントオクルージョンなしでレンダリングされ、右側はアンビエントオクルージョンを使用してレンダリングされています。階段の周りのエッジがどのように暗くなっているかに注目してください。

Original Fire Games によるレーシングゲーム、『*Circuit Superstars*』は、Unity で制作されたゲームであり、SSAO など、新しく導入された URP 機能などが数点使われており、ゲーム環境においての車やモデルをしっかり接地させ、ビジュアルに奥行きも加えています。

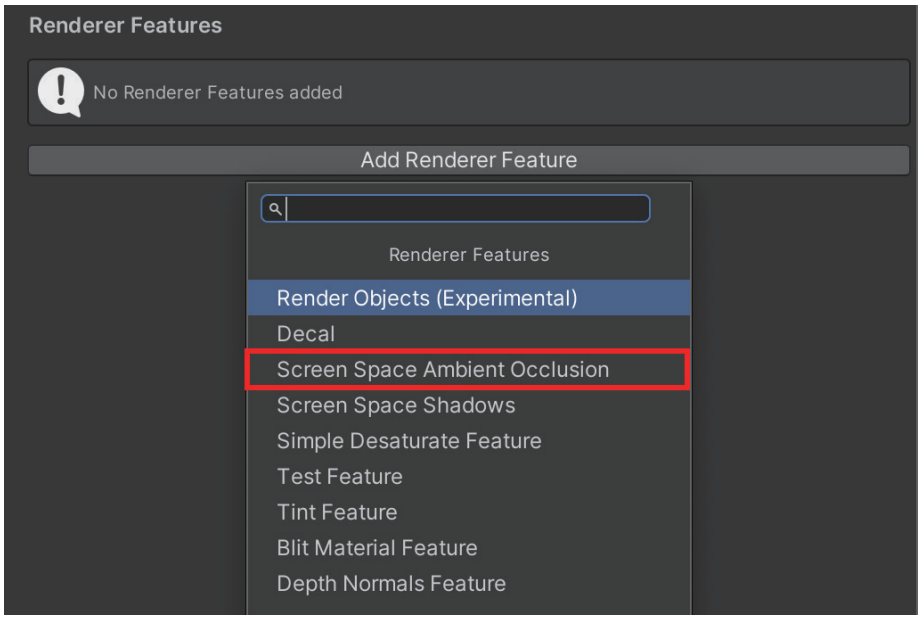
URP には、リアルタイムのスクリーンスペースアンビエントオクルージョン（SSAO）エフェクトが Renderer Feature として実装されています。使用するパスコードをこちらで表示できます。

> **ノート：**SSAO エフェクトは Renderer Feature であり、URP のポストプロセスエフェクトからは独立して機能します。ボリュームに依存したり、ボリュームと相互作用したりすることはありません。

実際の動作を見るには、**「Scenes」>「Ambient Occlusion」>「Ambient Occlusion」**を開きます。このシーンは、Unity Asset Store で無料のアセットとして入手できる低ポリゴンの都市環境です。

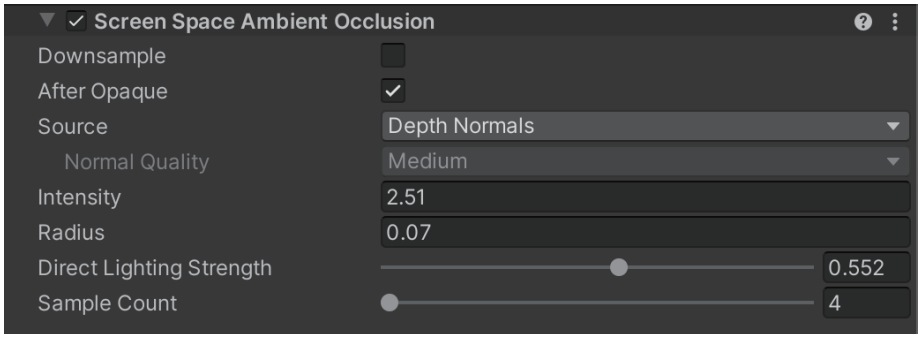
シーンでは、**Ambient_Occlusion_URP_Settings** という名前の URP アセットを使用します。これは、**Scene > Main Camera** に添付された `AutoLoadPipelineAsset` スクリプトを介してシーンを開くと自動的にロードされます。URP アセットは、**Ambient_Occlusion_URP_Settings_Renderer** を使用します。

シーンに SSAO を追加するには、Inspector で Universal Renderer Data アセットを表示し、「**Add Renderer Feature**」をクリックします。オプションのドロップダウンメニューで、「**Screen Space Ambient Occlusion**」を選択します。



!SSAO Renderer Feature の追加

SSAO Renderer Feature を追加すると、Inspector を介して結果を制御できます。利用できるプロパティを見てみましょう。



SSAO のオプション

Downsample

これを選択すると、X と Y 両方の方向で処理の解像度が半分になります。これにより、処理するピクセル数が事実上 75% 減るため、GPU 負荷も大幅に軽減されますが、エフェクトのディテールは少なくなります。

After Opaque

このオプションは、最終的なレンダリングの外観に作用しますが、パフォーマンスに影響を及ぼします。

— **無効な場合**：SSAO には「Depth」または「Depth Normals」プリパスがあります（下の「Source」オプションを参照）。SSAO は、それらの後に計算され、ライティング計算の実行時に DrawOpaques パスに適用されます。見栄えのよいアンビエントオクルージョンが提供され、ユーザーは SSAO の Direct Lighting Strength 値を制御できますが、パフォーマンスにマイナスの影響を及ぼします。

— **有効な場合**：「After Opaque」が選択されている場合、SSAO は「Depth Normals」を必要とします。「Depth」が選択されている場合、Depth プリパス（作成されている場合）または不透明度のレンダリング後に実行された CopyDepth パスから深度を取得します。SSAO は、ライティング計算の一部になるのではなく、DrawOpaques パスの後のすべてのものの上に追加されます。ここでのメリットは、プリパスをスキップできるため、パフォーマンスに役立つ可能性があることです。

> **ノート**：Render Opaque パスで Depth + Normals をレンダリングすることもできるため、そのオプションを有効にしてプリパスを完全にスキップし、パフォーマンスを助けることができます。

Source

このオプションでは、法線ベクトル値のソースを選択します。SSAO Renderer Feature は、サーフェス上の各点がアンビエントライトにどの程度露出しているかを計算するために法線ベクトルを使用します。

Source に使用可能な選択肢：

— **Depth Normals**：SSAO は DepthNormals パスで生成された法線テクスチャを使用します。このオプションにより、Unity はより正確な法線テクスチャを使用できます。

— **Depth**：SSAO は、代わりに深度テクスチャを使用して法線ベクトルを再構築します。このオプションは、カスタムシェーダーで DepthNormals パスブロックの使用を避けたい場合にのみ使用してください。このオプションを選択すると、**Normal Quality** プロパティが有効になります。

この 2 つのオプション間を切り替えると、パフォーマンスが変動する可能性があります。これは、ターゲットプラットフォームおよびアプリケーションによって異なります。多くのアプリケーションでは、パフォーマンスの違いはわずかです。ほとんどの場合、Depth Normals の方が優れた外観を生成します。

Normal Quality

これは、「**Source**」プロパティが「**Depth**」に設定されている場合にのみアクティブになります。

このプロパティのオプション（「Low」、「Medium」、「High」）は、Unity が深度テクスチャから法線ベクトルを再構築するときに取得する深度テクスチャのサンプル数を決定します。各品質レベルのサンプル数は次のとおりです。

— Low：1

— Medium：5

— High：9

パフォーマンスへの影響は中程度と見なされます。

Intensity

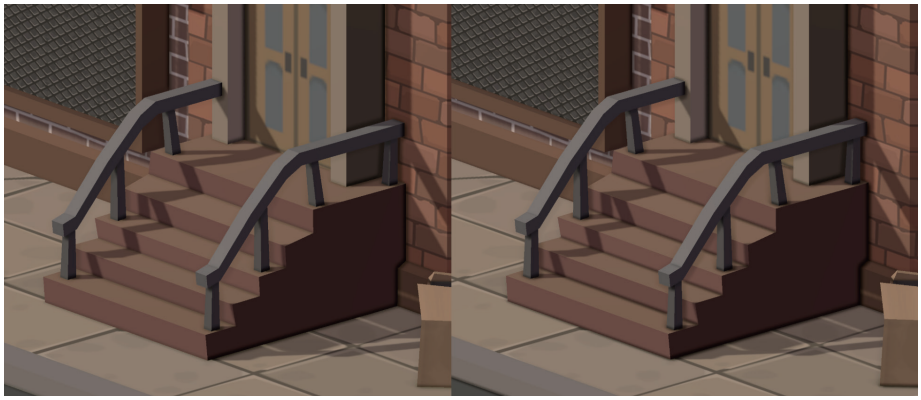
これは、暗さの強度を制御します。

Radius

このプロパティは、現在のピクセルの周りで取得する法線テクスチャのサンプル数を制御します。値が大きいほどパフォーマンスに大きな影響が及ぶため、できるだけ小さく保ってください。「Radius」値は、カメラから、ターゲットピクセルにレンダリングされるオブジェクトまでの距離に基づいてスケールされます。

Direct Lighting Strength

このプロパティは、ライティング計算の実行されたときの処理にまつわるものであるため、「**After Opaque**」オプションが無効にしてあることを根拠とします。直接光が当たるアンビエントオクルージョンの強度に変化を及ぼします。



Direct Lighting Strength の 2 つのバリエーション：0.2（左）と 0.9（右）

Sample Count

SSAO Renderer Feature は、ピクセルごとに、指定された半径内でこの数のサンプルを取得し、アンビエントオクルージョン値を計算します。この値を大きくすると効果がより滑らかで精密になりますが、パフォーマンスが低下します。「Sample Count」値を 2 倍にすると、GPU の計算負荷が 2 倍になります。

SSAO は、URP の自由度を表すもう 1 つの好例です。Renderer Features を使って解決することができる問題の数を縛るもの、それは開発者の発想力次第でしょう。

その他のリソース

— YouTube チュートリアル（UGuruz）

— アンビエントオクルージョンのドキュメント

— レシピで使用されているアセット（Marcelo Barrio 氏に感謝します）

デカール

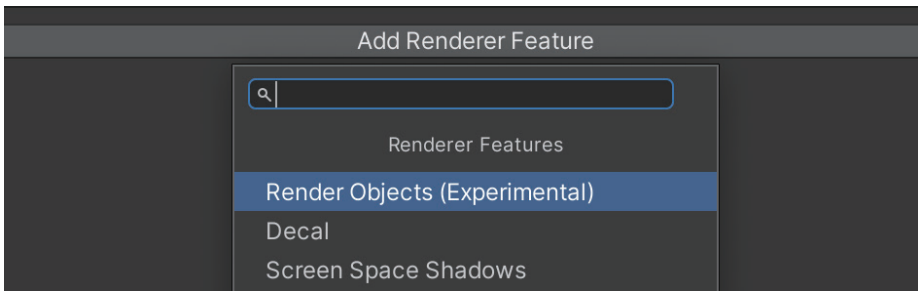
デカールは、サブサーフェスにオーバーレイを追加してくれる良案です。多くの場合、デカールは、シーン中にプレイヤーの操作と連動して弾痕やタイヤの溝など、そういった外観をゲーム環境に加えるために使われます。以下の画像の階段からわかるように、デカールはメッシュを包み込みます。フォルダー **Scenes > Decals** に、このレシピのシーンファイルとアセットがあります。

無機質なシーンに加えられたデカール

デカールは Renderer Feature を使ってシーンへレンダリングします。レシピフォルダーにある **Decals_URP_Settings_Renderer** を見てみると、Decals Renderer Feature が追加されていることがわかります。とはいえ普段と同じくして、メインカメラにアタッチされた `AutoLoadPipelineAsset.cs` スクリプトにより、シーンのロード時に適切なパイプラインアセットが確実に使用されるようになっています。

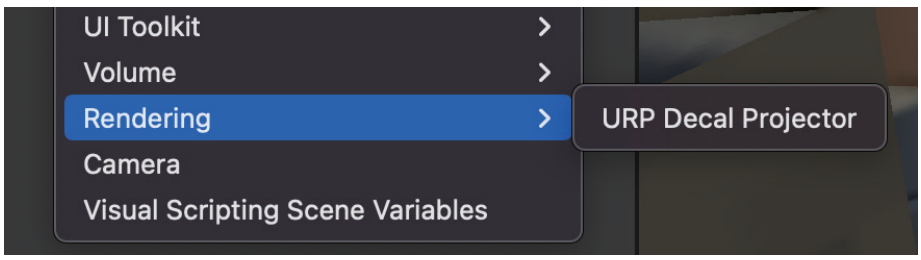
デカールの多くのユースケースの 1 つは、プロブシャドウの 3D サーフェスへの投影です。これは、Splashteam が Unity を使用して制作したゲーム『*Tinykin*』のキャラクター Milo で使用されています。

デカールをカスタムシーンに追加するには、プレイヤーがレンダリングに現在使用している Universal Renderer データアセットを選択し、Inspector で「Add Renderer Feature」ドロップダウンから「**Decal**」を選択します。



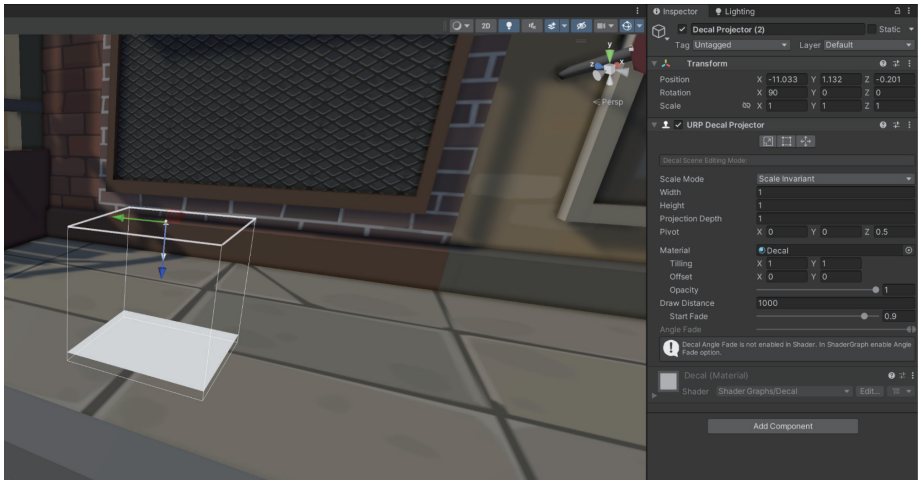
「Add Renderer」ドロップダウンの「Decal」オプション

エディターで作業しているときにデカールをシーンに追加するには、Hierarchy ウィンドウを右クリックし、**「Create」>「Rendering」>「URP Decal Projector」**を選択します。



URP Decal Projector の作成

通常行われている作業と変わらず、エディターで URP Decal Projector の位置と向きを設定します。Decal Projector は、平行投影を使用するので、サーフェス上のデカールキャストのサイズはサーフェスからプロジェクターまでの距離の影響を受けません。はじめに、新しい Decal Projector が白いブロックとして表示されます。軸の矢印に加えて、投影の方向を示す白い矢印が表示されます。



新しい Decal Projector

URP Decal Projection のプロパティ

— **Scale Mode**：デフォルトでは、URP Decal Projection コンポーネントの「**Scale Mode**」は「Scale Invariant」に設定されています。これは、デカールのサイズが「**Width**」プロパティと「**Height**」プロパティによってのみ決定されることを意味します。「**Inherit from Hierarchy**」に切り替えると、ゲームオブジェクトの Transform スケールが「Width」および「Height」プロパティと結合されます。

— **Width と Height**：これらのプロパティは、デカールのサイズを制御します。

— **Projection Depth**：プロジェクターバウンディングボックスの深度を設定します。プロジェクターはローカル Z 軸に沿ってデカールを投影します。

— **Pivot**：ルートゲームオブジェクトの原点を基準とする Projector バウンディングボックスの中心のオフセット位置を設定します。

— **Material**：投影するマテリアルを設定します。マテリアルは、Shader Graph/Decal を使用します（これについてはすぐに詳しく説明します）。

— **Tiling と Offset**：UV 軸に沿ったデカールマテリアルのタイリングおよびオフセット値です。

— **Opacity**：不透明度値を指定します。値が 0 の場合、デカールは完全に透明になり、値が 1 の場合、デカールは「Material」で定義されたとおりに不透明になります。

— **Draw Distance**：この Projector がデカールの投影を停止し、URP がデカールをレンダリングしなくなる、カメラからデカールまでの距離です。

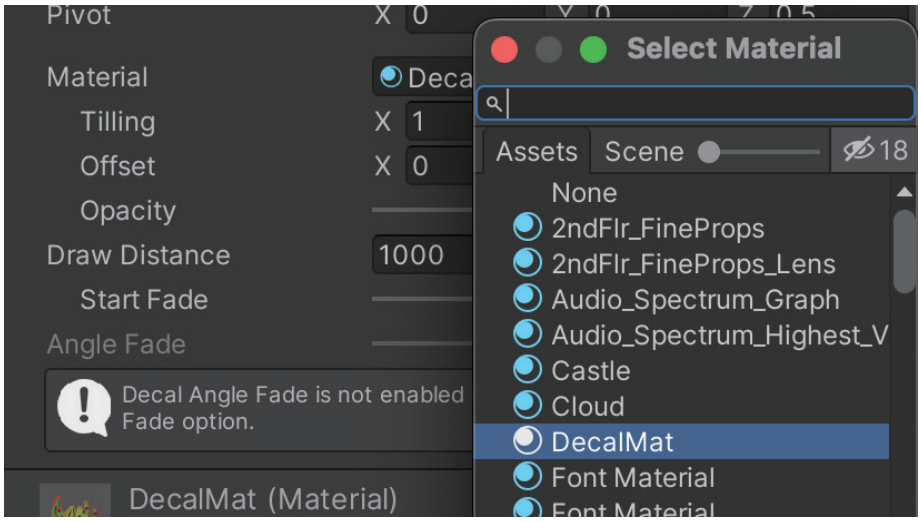
— **Start Fade**：Projector がデカールのフェードアウトを開始する、カメラからの距離を（スライダーで）設定します。0 から 1 の値は、「Draw Distance」に対する割合を表します。値が 0.9 の場合、Unity は「Draw Distance」の 90% の距離でデカールのフェードアウトを開始し、「Draw Distance」でフェードアウトを終了します。

— **Angle Fade**：スライダーを使用して、デカールの逆方向と受け取るサーフェスの頂点法線との間の角度に基づく、デカールのフェードアウト範囲を設定します。

マテリアルの作成

Decal Projector は、シェーダー Shader Graph/Decal を使用するマテリアルを使用する必要があります。この例では、Scene フォルダーにあるマテリアル DecalMat を使用します。ベースマップが割り当てられていますが、法線マップはありません。これは、デカールのサーフェスをゴツゴツした外観にしたい場合に役立ちます。

Inspector で、マテリアルがプロジェクターに割り当てられています。



URP Decal Projector マテリアルの割り当て

コードでのデカールの追加

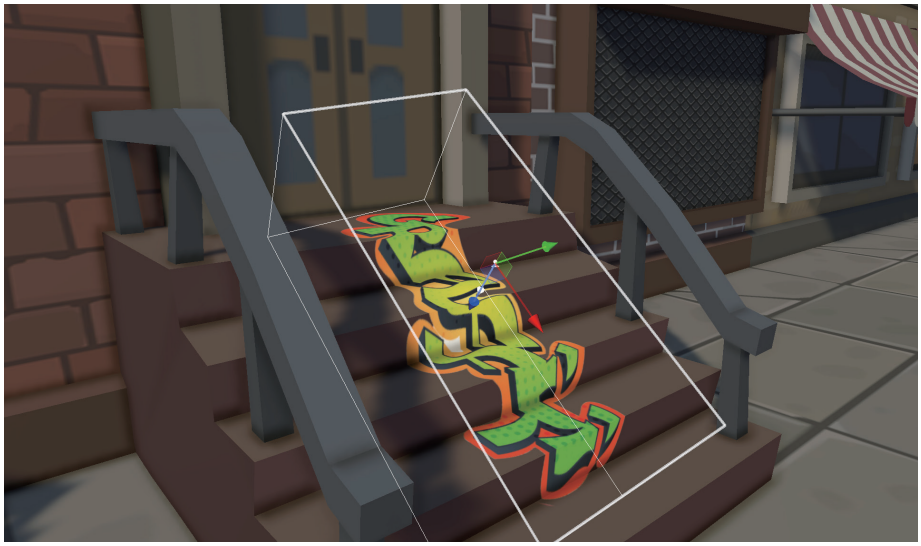
エディターでの開発中に URP Decal Projector をシーンに追加できますが、実行時にユーザーの操作の結果として追加する方が一般的です。プレハブを作成してマテリアル、幅、高さを設定できますが、コードで実行時にこれを簡単に更新できます。このコード例では、インスタンス化、配置、向きにのみ焦点を当てます。「コライダーでマウスをクリックした結果デカールが追加される」という動作が完成しているコードは、レシピフォルダーの AddDecal.cs スクリプトにあります。

```
void AddDecalProjector(Vector3 pos, Vector3 normal)
{
    GameObject decalProjectorObject = Instantiate(decalProjecto
rPrefab);

    // DecalProjector の新しいマテリアルインスタンスを作成する
    //（個々のデカールでマテリアルを制御したい場合）
    //DecalProjector decalProjectorComponent =
decalProjectorObject.GetComponent<DecalProjector>();
```

```
    //decalProjectorComponent.material = new Material(decalProj
ectorComponent.material);

    // サーフェスから離す
    pos += normal * 0.5f;

    Quaternion up = Quaternion.AngleAxis(Random.Range(0, 360),
Vector3.left);
    Quaternion rot = Quaternion.LookRotation(-normal,
up.eulerAngles);

    decalProjectorObject.transform.SetPositionAndRotation(pos,
rot);
}
```

この関数が呼び出されるのは、コライダー上でのマウスダウンイベントの後に RaycastHit があるときです。**pos** は hit.point で、**normal** は hit.normal です。decalProjectorObject という名前のプレハブがインスタンス化されます。位置を取得するために、Projection Depth を超えないように pos Vector3 をサーフェスから離す必要があります。これを行うには、法線に沿ってポイントを動かします。デカールを方向付けるには、ランダム化したアップベクトルを最初に作成します。デカールをサーフェスに合わせるために必要な回転をさせるために、法線の周囲を任意の角度で回すには、パラメーターとして、逆向きの法線とランダム化したアップベクトルを使用しましょう。

デカールにはゲームで幅広い用途があります。URP Decal Projector はツールボックスの中でも役に立つツールです。



シーンビューのデカール

その他のリソース

— Decal Renderer ドキュメント

— Llam Academy の YouTube チュートリアル

水

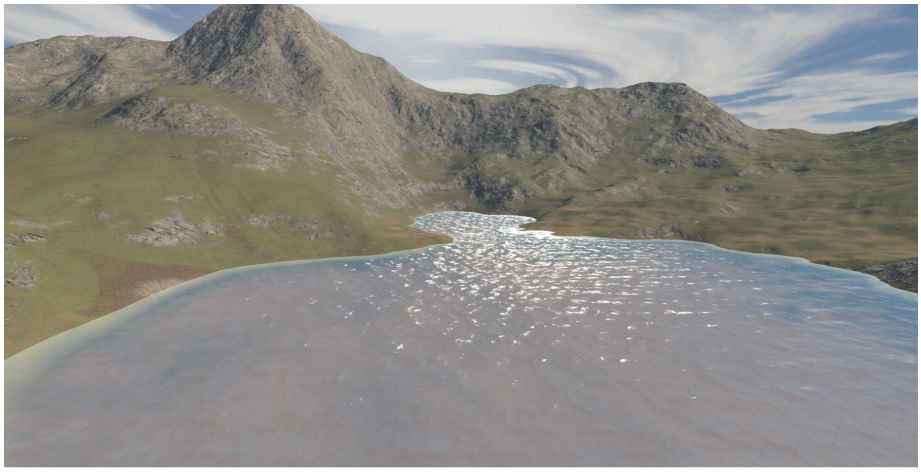
このレシピは、ただただ、水のシェーダーを作成するためのものになります。これはシェーダーグラフ内で作成されるため、ステップはアーティストやデザイナーにとって利用しやすくなります。

このシェーダーは以下、3 つの段階を経て構築されます。

— 水の色を作成する

— タイル状の法線マップを動かして、さざ波をサーフェスに加える

— 動くディスプレイスメントを頂点位置に追加して、うねり（swell）のエフェクトを作成する

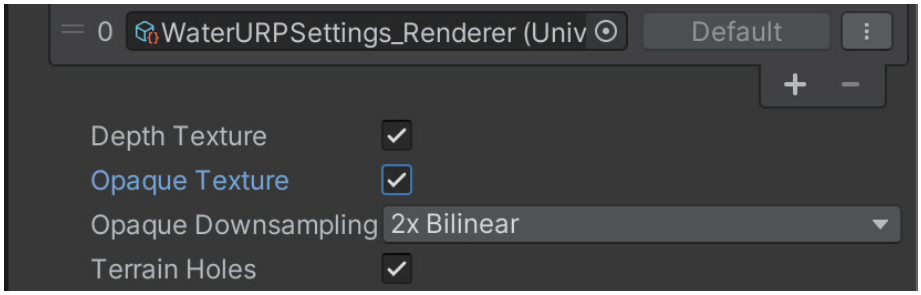


シンプルな水のシェーダーの動きを見せる動画からの静止画。

完成形を確認するには、**Scenes > Water** フォルダーの Water シーンを開きます。最終的なシェーダーでは、DepthFade と TextureMovement の 2 つのサブグラフが使用されます。水のシェーダーについて確認する前に、これらについて説明しておき

水と水生植物というものは、ビデオゲーム内で広々とした美しい環境を作るための重要な 2 つのビジュアル要素です。サバイバルゲーム『*Len’ s Island*』のこの画像は、Unity を使用して Flow Studio によって制作されました。

ましょう。Water シーンでは、「**Depth Texture**」と「**Opaque Texture**」オプションが有効になった **WaterURPSettings アセット**を使用します。「Opaque Texture」が必要になるのは、このレシピに含まれない屈折などのエフェクトを追加する場合のみであることに注意してください。



WaterURPSettings アセットで「Depth Texture」と「Opaque Texture」が選択されている

DepthFade サブグラフ

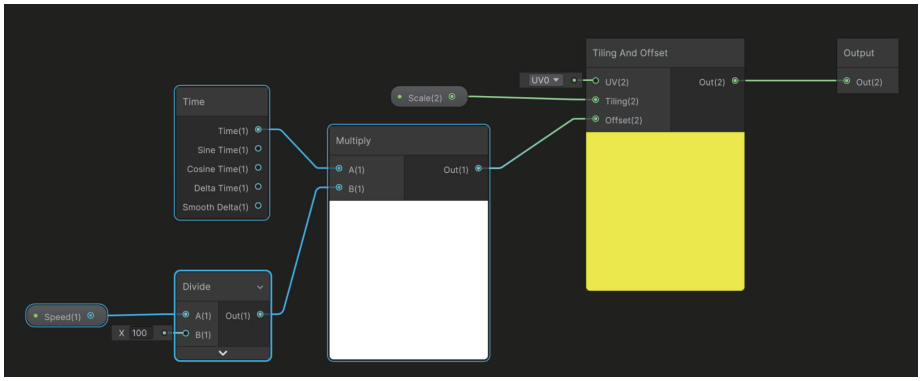
DepthFade サブグラフ

水深が浅い部分と深い部分には、それぞれ別の色が必要です。水の最終的な色は、「**Depth**」プロパティに基づいて、これら 2 つの色が混ざったものになります。「Depth」は、水のサーフェスとその下のジオメトリの距離です。水のシェーダーは透明に設定されるため、不透明なジオメトリはあらかじめレンダリングされます。また、「Depth Texture」が「URP Settings Asset」で選択されているため、現在の深度を読み取ることができます。

「Sampling」が「Eye mode」に設定された「**Scene Depth**」ノードによって、視点から不透明なジオメトリの現在のピクセルまでの距離が提供されます。出力値のモードとして「Raw」が選択された「**Screen Position**」ノードには、水の現在のピクセルのレンダリングに関する情報が保持されます。「**Split**」ノードが使用されるのは、視点から水の現在のピクセルまでの距離が保持される W コンポーネントを必要とするためです。

水までの距離を、既存の不透明なジオメトリまでの距離から差し引くと、真下へのレイではなく視線からのレイではありますが、水の深度を導き出すことができます。次に、「**Divide**」ノードによって、浅い部分と深い部分の間のエッジがどこに表示されるかが制御されます。このサブグラフの出力は 0 と 1 の間になる必要があるため、「**Saturate**」ノードを使用します。これは、出力を常に 0 から 1 までの間に制限して、特別な「**Clamp**」ノードとして作動します。

TextureMovement サブグラフ

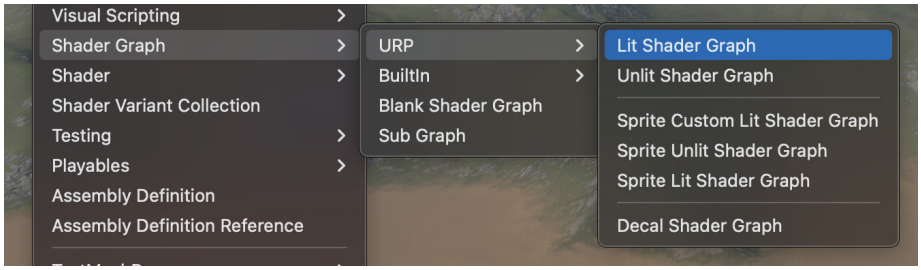


TextureMovement サブグラフ

水のシェーダーには動くテクスチャがいくつか含まれ、TextureMovement サブグラフによって処理されます。このサブグラフでは、「**Time**」ノードが「**Multiply**」ノードに対する 1 つの入力として使用されます。入力の「Speed」は 100 で除算され、「Multiply」ノードの 2 つ目の入力になります。「Multiply」ノードの出力は、「**Tiling and Offset**」ノードに対する「Offset」入力として使用されます。「**Scale**」プロパティによって「Tiling」入力が形成されます。時間が経過すると、この単純なサブグラフによって UV が更新され、「Speed」と「Scale」の入力が与えられた「**Sample Texture 2D**」ノードによって使用されます。

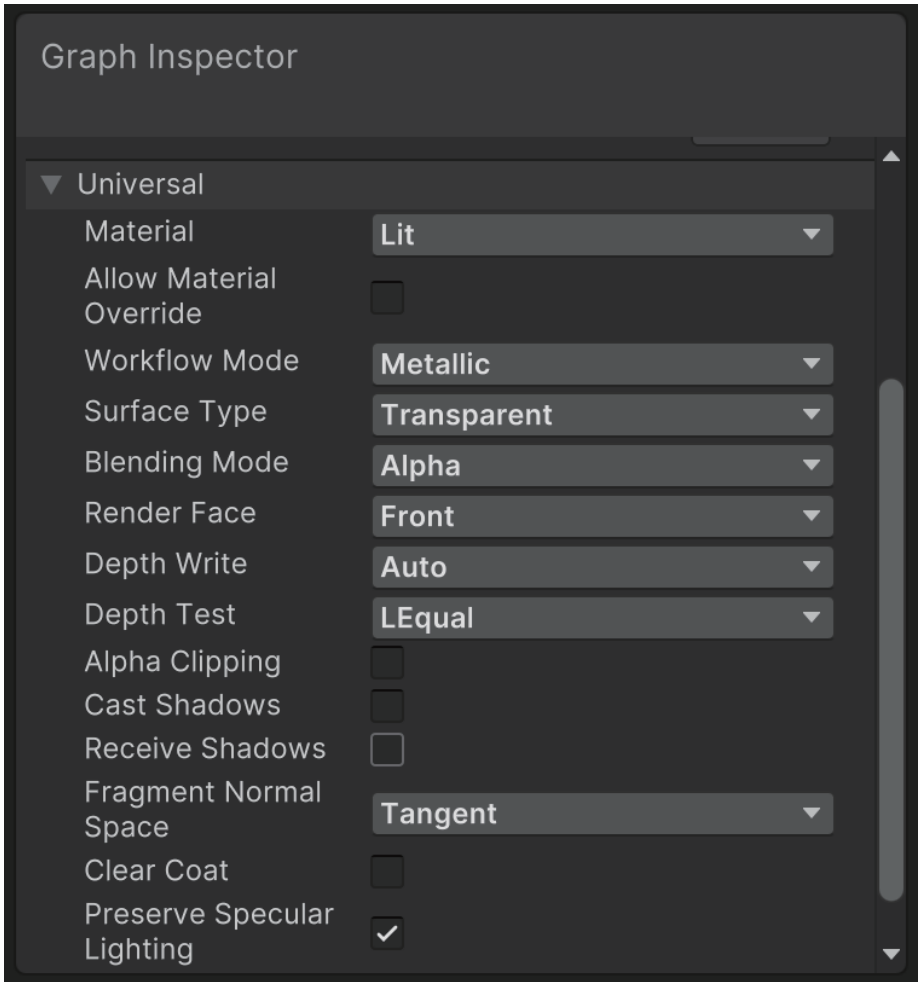
水のシェーダー

いよいよ、Lit シェーダーグラフ（**「URP」>「Lit Shader Graph」**）に基づいて水のシェーダーを作成します。



「Create」>「Shader Graph」>「URP」>「Lit Shader Graph」

次に、「Graph Inspector」を使用して「**Surface Type**」を設定します。



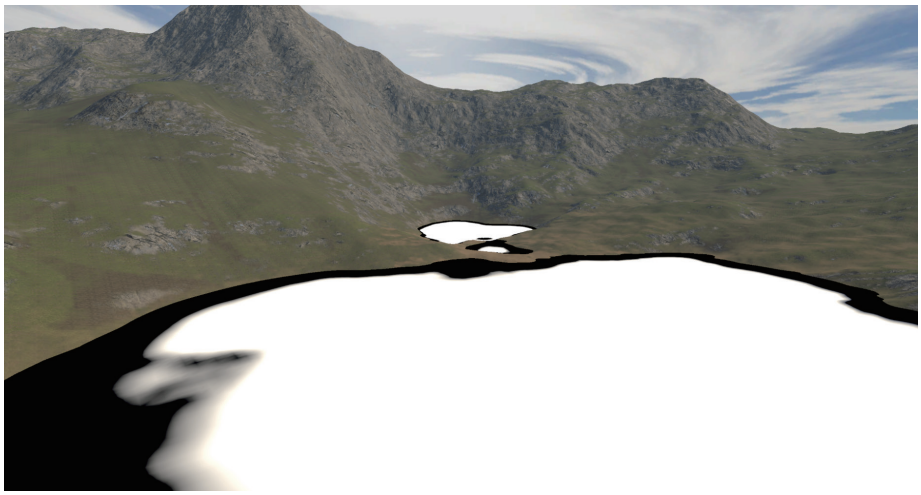
「Surface Type」の設定

ここで、グラフを編集します。まず「Color」です。

色

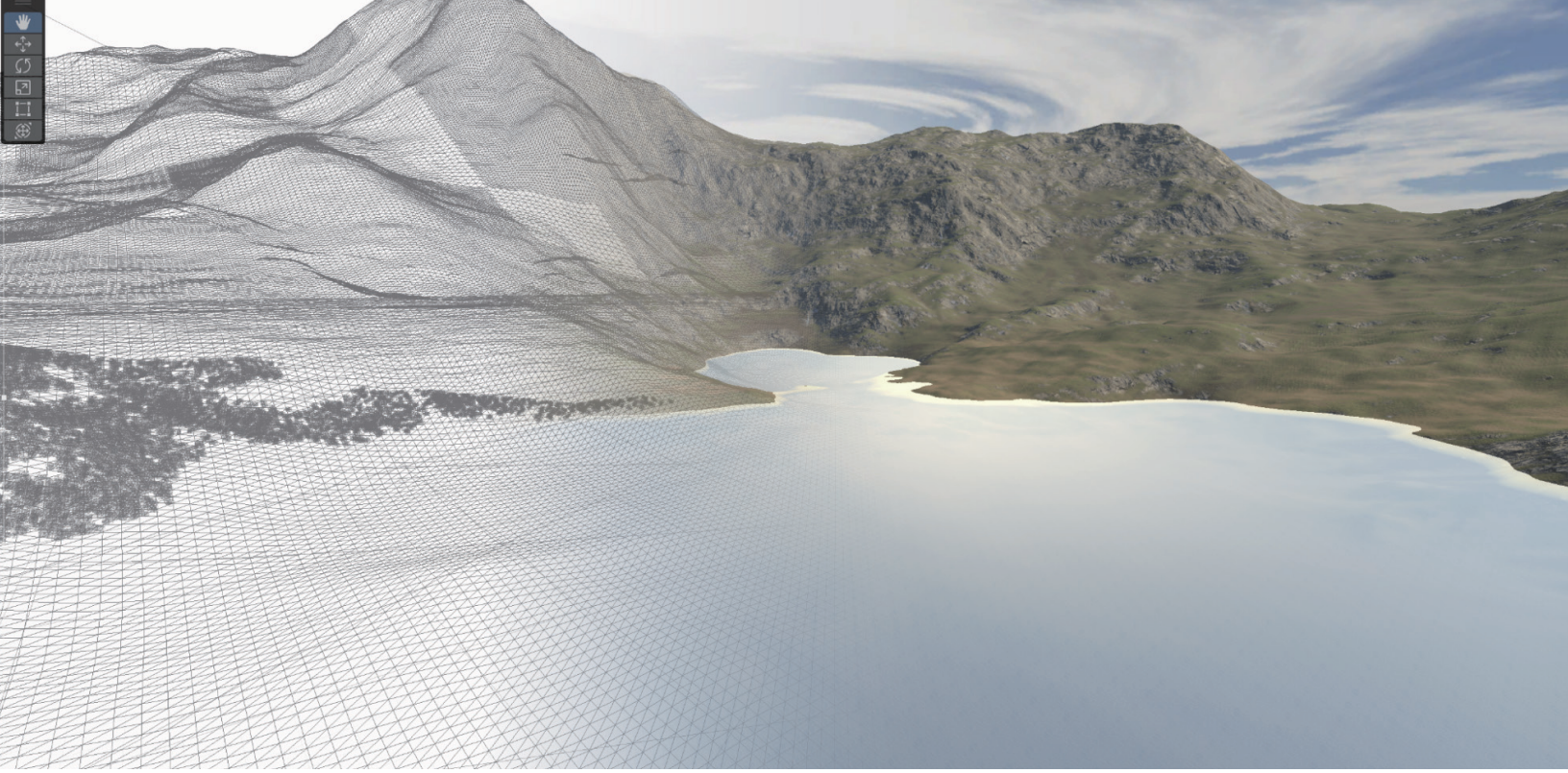
色とアルファ

色を処理するには、DepthFade サブグラフを追加します。このサブグラフでは、制御のために、フロート表示の「**Depth**」プロパティを使用します。この出力は「Fragment」シェーダーの「Base Color」入力に直接つながっており、結果として以下に示す画像が生成されます。浅いところの水は黒、深いところの水は白になります。「Depth」の値を大きくすると、黒い部分がさらに広がります。黒は 0、白は 1 で示されます。



「DepthFade」の出力を**「Fragment」>「Base Color」**に直接つなげる

「DepthFade」を「Base Color」の入力に直接リンクせずに、「Lerp」ノードにつなげます。「**ShallowWaterColor**」は黒を置き換える入力 A、「**DeepWaterColor**」は白を置き換える入力 B です。これらの色のアルファを設定するときは、浅い部分の水の透明度を高くします。「Lerp」の出力を**「Fragment」>「Base Color」**につなげます。アルファについては、「Split」ノードを使用し、A 出力を**「Fragment」>「Alpha」**にリンクします。こうすることで、結果として以下の画像のシーンが生成されます。



密度が高いメッシュおよび色が設定された水

実際の水は平面ですが、頂点ディスプレイスメントを設定するため、メッシュには画像のように多くの頂点があります。

単純で平らなサーフェスから始めて、さらに処理が必要です。すなわち法線マップです。

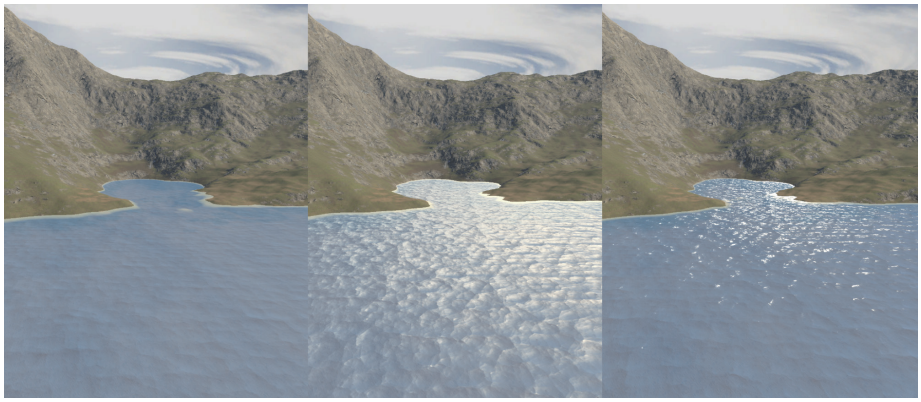
法線マップ

「Fragment」>「Normal」の制御

法線マップによって、動くさざ波がサーフェスに加えられます。最初の入力プロパティは「**Wave Speed**」で、これは TextureMovement サブグラフの「Speed」入力として使用されます。「**Scale**」は「50,50」にハードに設定されています。また、TextureMovement の 2 つ目のノードを介して、「**Speed**」が「Multiply」ノードによって事前処理され、「Wave Speed」プロパティ値の半分が差し引かれます。

法線を計算する次のステップでは、TextureMovement サブグラフの 2 つのノードによって処理される UV を使用して、法線テクスチャを 2 回サンプリングします。2 つの法線を一緒に追加して、2 つの動くテクスチャのエフェクトを組み合わせます。このシェーダーには「**Normal Strength**」フロートプロパティがあり、これを「Normal Strength」ノードの「Strength」入力として使用できます。ただし、端に近いところでは、さざ波の動きを止める必要があります。このために、広がりを制御する「**Edge Distance**」シェーダープロパティを含む DepthFade サブグラフのノードを使用します。これは、「Lerp」ノードの T 入力として使用され、0 から「Normal Strength」までがブレンドされます。グラフのこのステージの出力は、「Fragment」>「Normal」につながります。

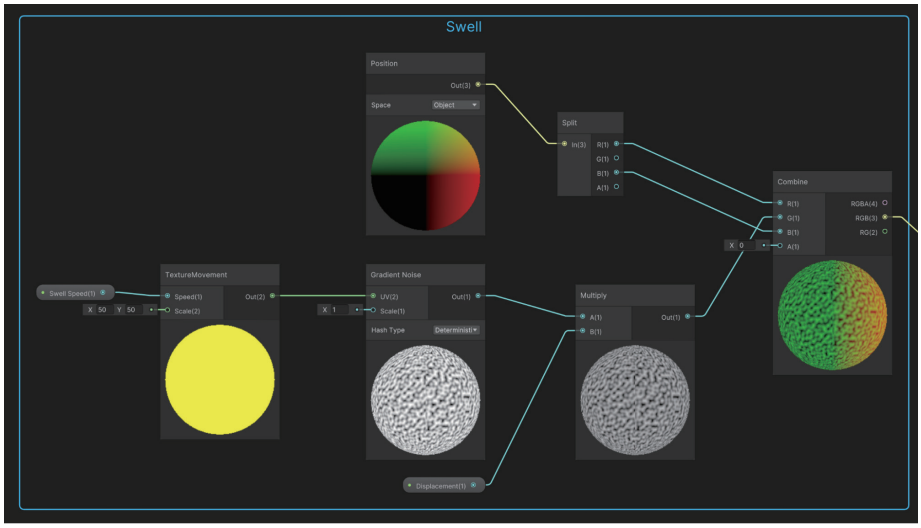
これで、制御して扱うことができる、さざ波ができました。反射に関するプロパティは、単純なフロートプロパティを使用して、「Fragment」の「**Smoothness**」を制御することで調整できます。以下の画像は、「Smoothness」値を変えた場合のエフェクトを示しています。



さざ波に適用する滑らかさのレベルの変化（左から右に 0、0.5、1）

次のステップでは、頂点ディスプレイスメントを使用して水に動きを加えます。

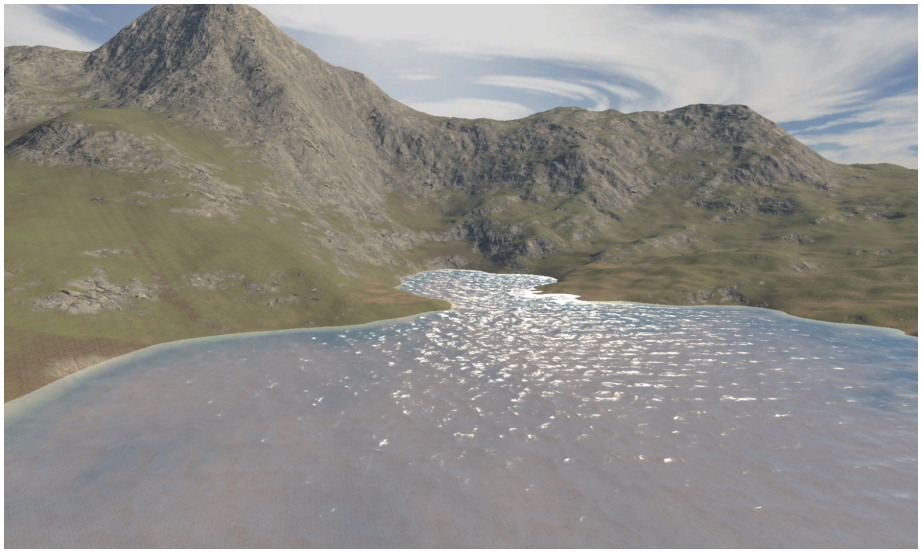
うねり（Swell）



「Gradient Noise」を使用したうねりの制御

このステップでは、TextureMovement サブグラフのノードを再び使用します。「Speed」は、シェーダーのフロートプロパティ「**Swell Speed**」を使用して設定され、「Scale」は「50,50」にハードに設定されます。これが、「Scale」が「1」にハードに設定された「Gradient Noise」ノードに対する「UV」入力として使用されます。「Multiply」ノードを使用し、シェーダーのフロートプロパティ「**Displacement**」を使用してこの値を制御します。これらのノードの目的は、オブジェクト空間の頂点の Y 値を設定することです。「Position」ノードの「**Space**」パラメーターが「Object」に設定されていることに注意してください。これが「Split」ノードにリンクし、さらに「Combine」ノードにリンクします。「Combine」は「Split」ノードから R 値（「Position X」）と B 値（「Position Z」）を直接受けとります。Y の G 値は「Gradient Noise」パスに由来します。「RGB(3)」出力は「Vertex」>「Position」にリンクします。

再生モードでシーンを表示すると、うねりが、水全体、特に水際の部分で動くことがわかります。



完成形

このレシピは単純な水のシェーダーの基本を作成するものです。「Caustic Reflections」、「Refraction」、および「Foam」を使用するとさらに効果を高めることができます。詳しいガイダンスについては以下のリンクを参照してください。

詳細情報

— Unity YouTube チュートリアル

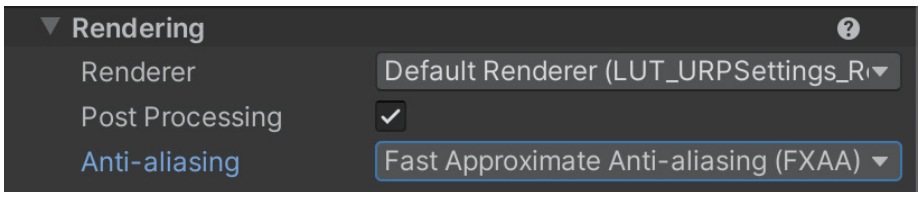
— コースティクスのリフレクションについてのチュートリアル（Alan Zucconi）

— 水のスタイル化のチュートリアル（Binary Lunar）

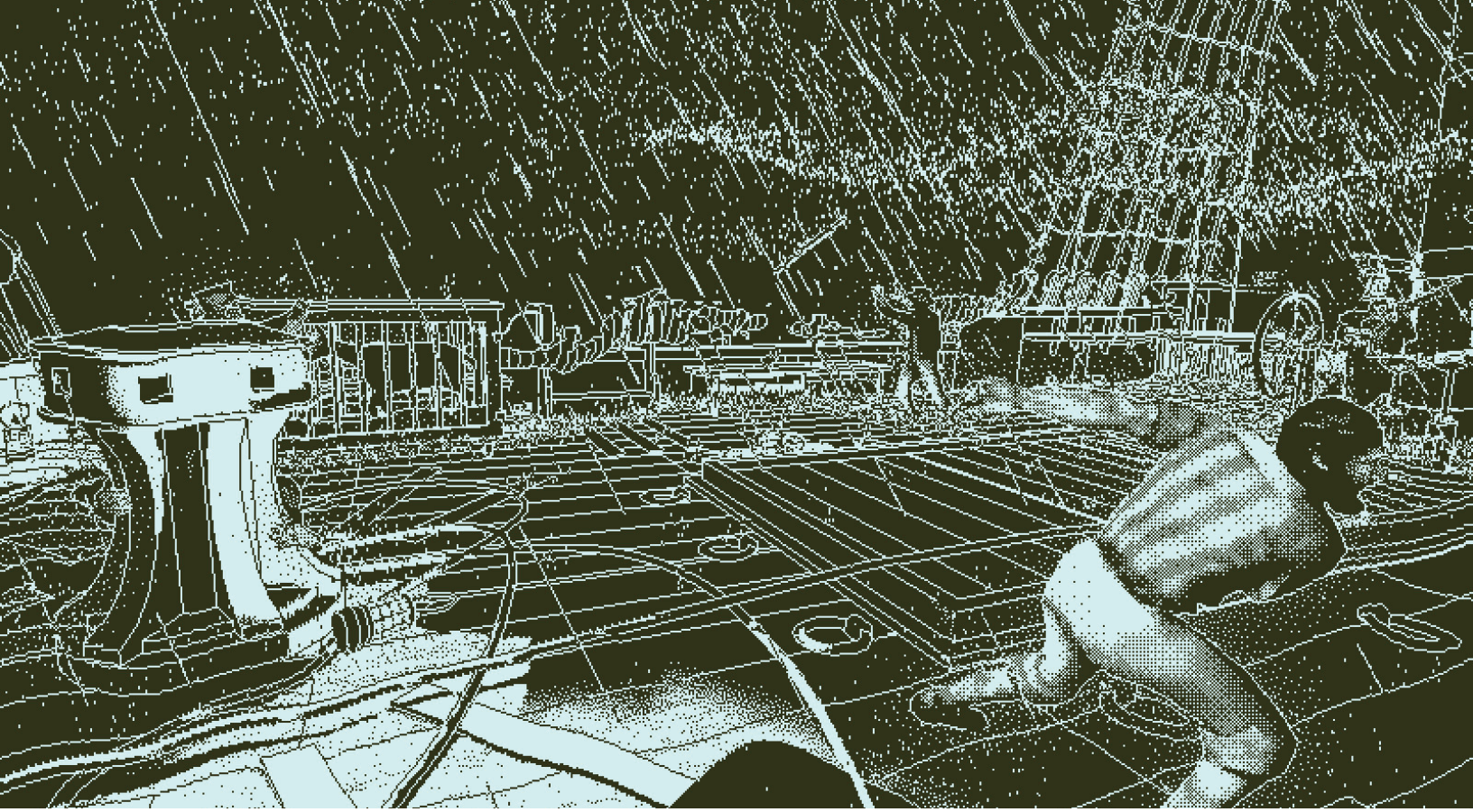
カラーグレーディング用の LUT

カラールックアップを使用したグレーディングエフェクトの生成

URP で提供されているポストプロセスフィルターをまだ使用したことがないのであれば、このセクションが役に立ちます。このレシピで使用するのは 1 つのフィルターですが、必要なステップはすべてのフィルターに当てはまります。新しい URP シーンでは、ポストプロセスがデフォルトで無効になっているため、**「「Camera」 > 「Rendering」」**パネルを使用して有効にしてください。

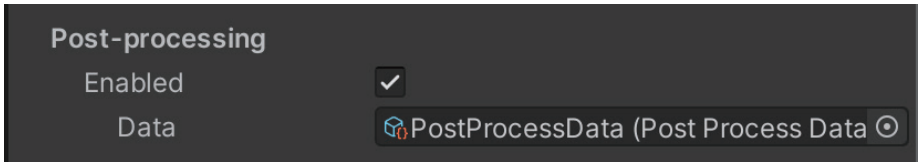


「Camera」 > 「Rendering」 で 「Post Processing」 を選択



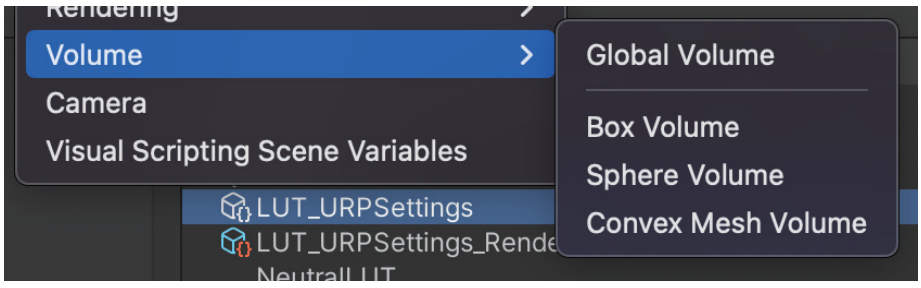
ミステリーアドベンチャー FPS ゲーム『*Return of the Obra Dinn*』は Unity を使用して Lucas Pope 氏により制作されましたが、粗削りなアートスタイルと独特のカラーパレットによって独特のルックアンドフィールを実現しています。これは、このレシピに従って実現することができます。

また、Universal Renderer のデータアセットのポストプロセスも有効にする必要があります。



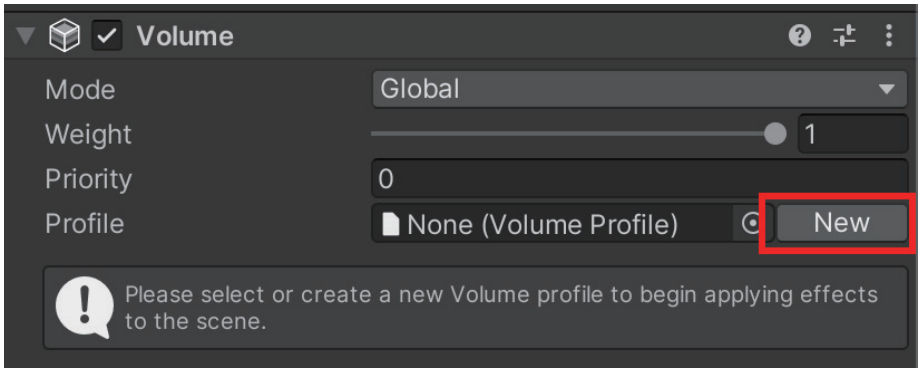
Universal Renderer のデータアセットで「Post-processing」を選択

カメラが配置された場所にフィルターを適用するには、「Global Volume」を追加します。Hierarchy ウィンドウを右クリックし、**「Volume」>「Global Volume」**を選択します。



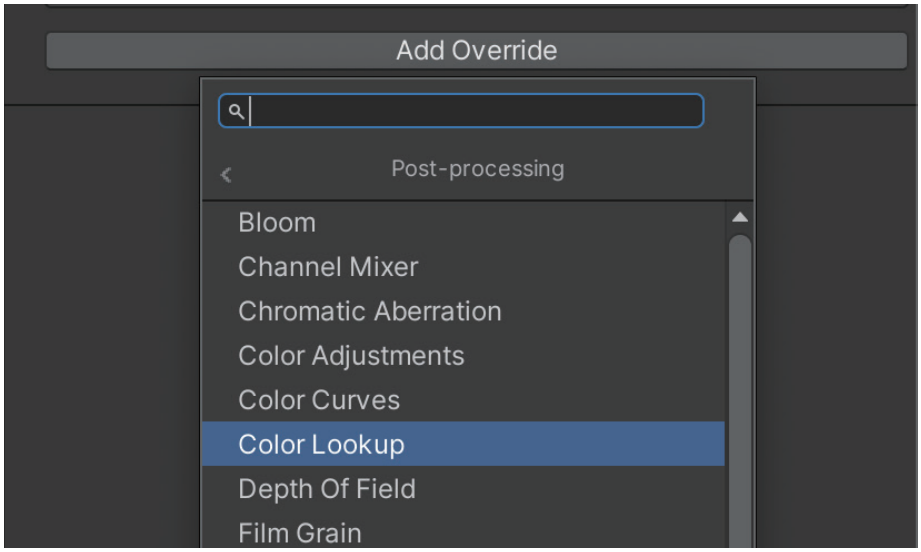
「Global Volume」の作成

新しいゲームオブジェクトを選択し、「**New**」をクリックして新しいプロファイルを作成します。



新しいプロファイルの作成

ここでオーバーライドを追加できます。「**Add Override**」ボタンを押し、「post-processing」を選択してから、「**Color Lookup**」を選択します。

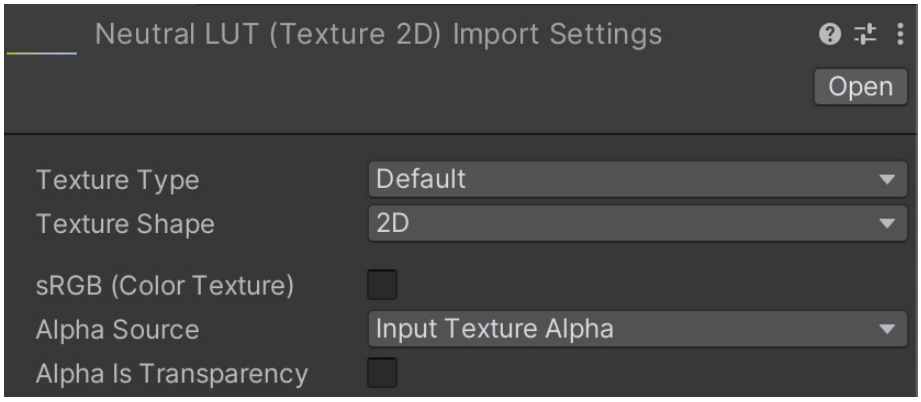


「Color Lookup」ポストプロセスフィルターの追加

「**All**」ボタンをクリックします。ここで、LUT（ルックアップテーブル）画像テクスチャが必要です。これは、デフォルトのレンダリングの色を変えるためにフィルターによって使用される細長い画像です。画像ファイルは、**「Scenes」>「LUT」>「NeutralLUT.png」**にあります。または、このリンクを使用してダウンロードしてください。

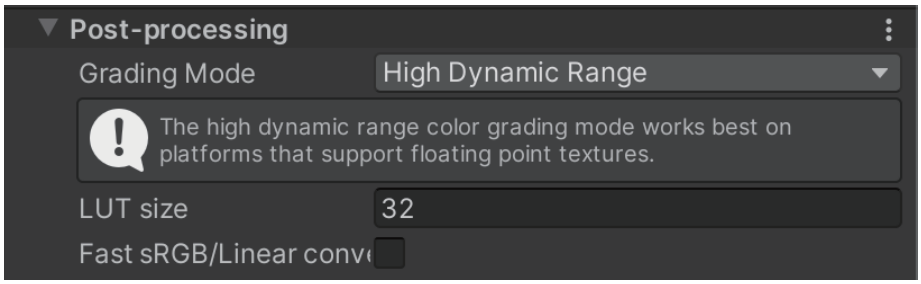
NeutralLUT.png

LUT 画像の「**sRGB (Color Texture)**」は無効にする必要があります（画像を選択し Inspector で表示して行います）。



すべての LUT テクスチャの「sRGB (color Texture)」の無効化

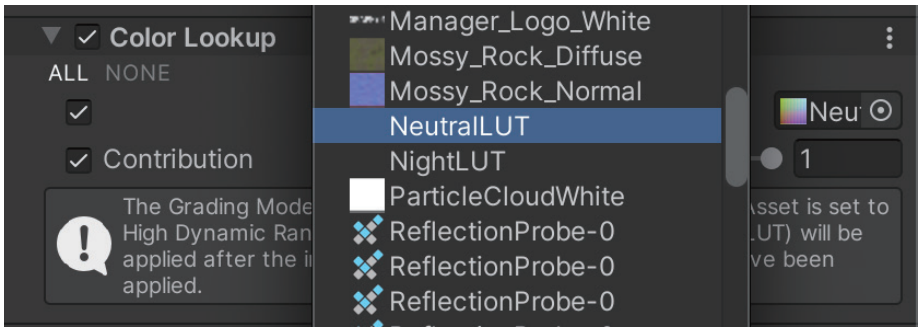
前の NeutralLUT 画像のブロックを数えてください。32 個あることがわかります。または、16 個のブロックを使用できます。ブロック数として 32 個または 16 個どちらを選択するとしても、URP アセットの設定が選択と一致するようにします。32 を選択した場合は、「Post-processing」パネルの「**LUT size**」が「**32**」に設定されていることを確認します。「**Grading Mode**」オプションを使用して自由に試してみてください。



「LUT size」の設定

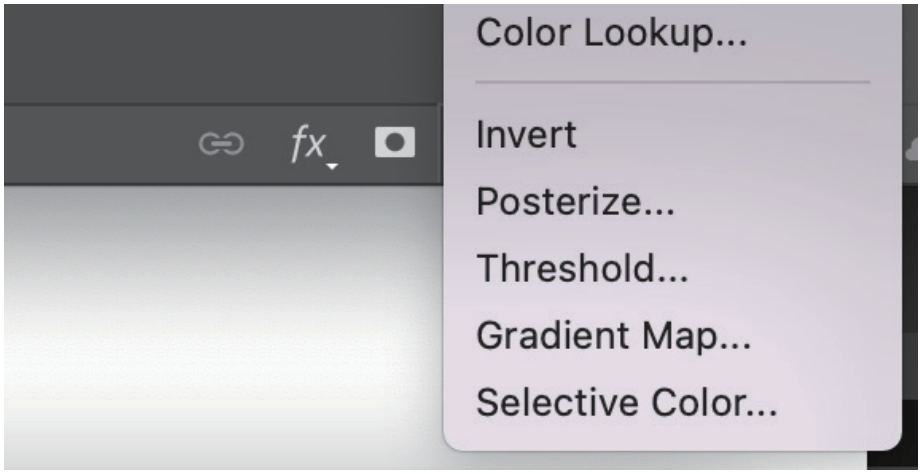
「**Color Lookup**」設定パネルを使用して「**NeutralLUT.png**」をルックアップテクスチャとして割り当てても、レンダリングされた画像に変化はありません。このフィルターはテクスチャを使用して新しい色を設定します。コードが、現在のピクセルカラーを取得し、それを使用して LUT 画像上のテクセルを探します。ニュートラル LUT 画像では、テクセルカラーは現在のピクセルカラーと同じになります。

不思議なことが起きるのは、ルックアップテクスチャとして使用する画像をペイントプログラム（Photoshop や Krita など）を使用して処理するときです（このセクションの最後の「その他のリソース」に、カラーグレーディングのための Krita の使用方法について説明する YouTube 動画へのリンクがあります）。



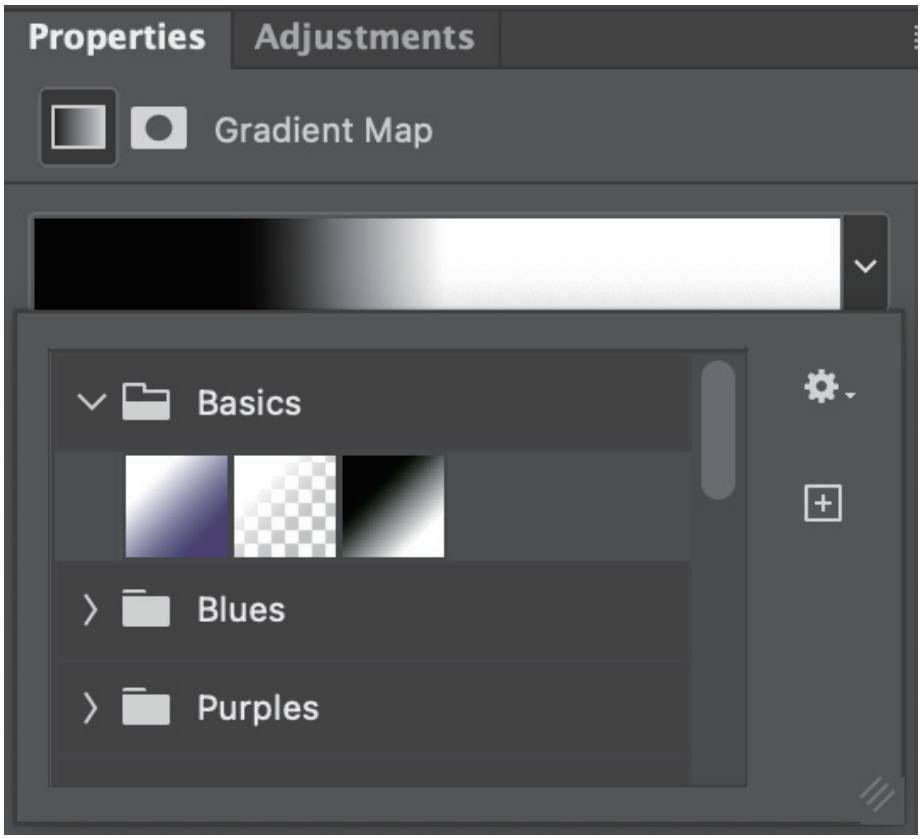
ルックアップテクスチャの割り当て

シーンの画面キャプチャを撮って、Photoshop で開きます。「**Layers**」パネルの一番下に、白地に黒色の円形ボタンがあります。これを選択して、パネル内で「**Gradient Map**」を探します。新しい色調整レイヤーが追加されます。



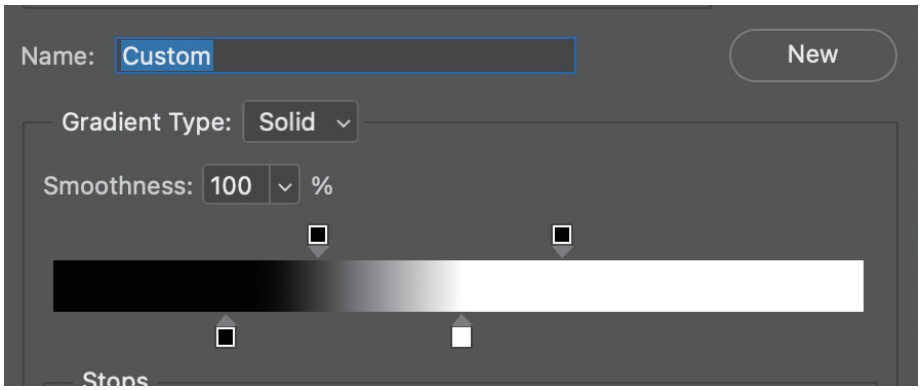
色調整レイヤーの作成

ハイコントラストの白黒画像を生成する色調整レイヤーを作成するには、「Gradient Map」ドロップダウンをクリックし、「Basics」の白黒を選択します。



白黒グラデーションの選択

コントラストを強くするには、グラデーションをクリックして新しいウィンドウを開きます。ストップを使用してコントラストを調整します。

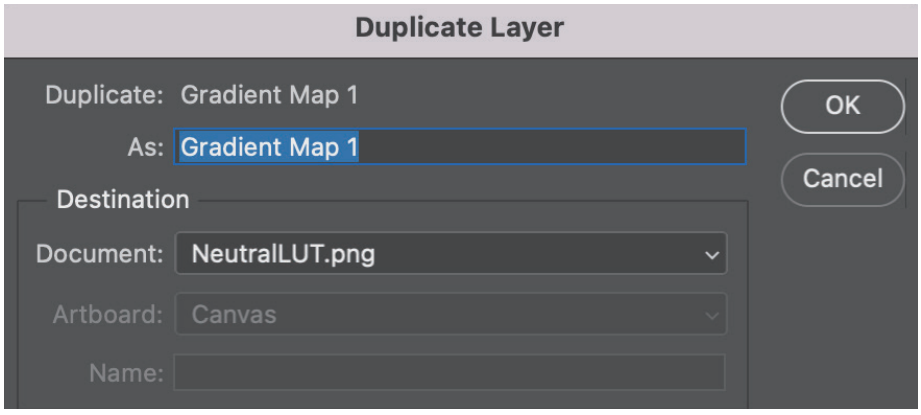


コントラストを強くするためにストップを変更

これで画面キャプチャは白黒になるはずです。

「Gradient Map」のエフェクト

お好みのグレーディングを選択したら、そのレイヤーを NeutralLUT.png ファイルに適用する必要があります。ファイルを Photoshop で開きます。画面キャプチャに戻って、調整レイヤーを右クリックし、「**Duplicate Layer**」を選択します。新しいパネルで、**「Destination」>「Document」**として NeutralLUT.png を選択します。

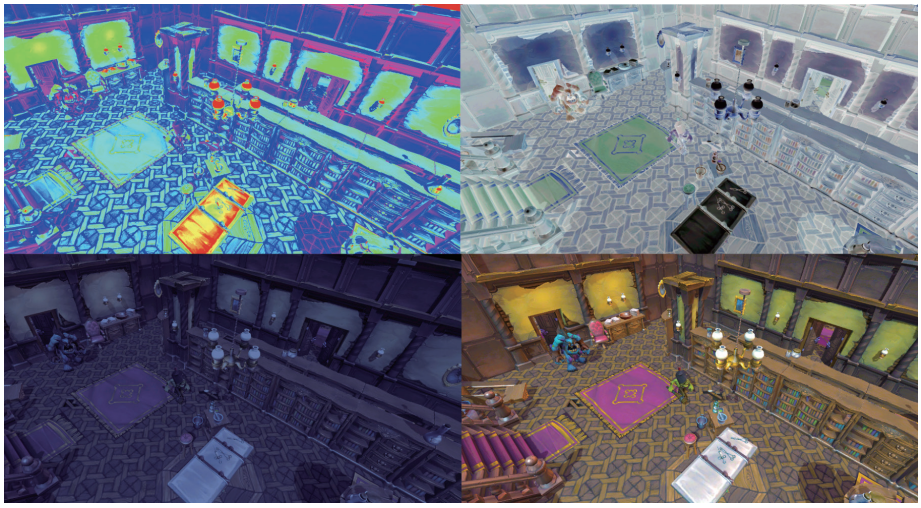


調整レイヤーの複製

現在テクスチャは以下のように表示されています。

B&WLUT.png

これを保存して、プロジェクトの Assets フォルダーにドラッグします。「Inspector」パネルを使用して必ず「sRGB (Color Texture)」を無効にしてください。最後のステップで、新しい LUT テクスチャを「Color Lookup」フィルターのルックアップテクスチャとして割り当てます。



さまざまな LUT テクスチャの使用

LUT テクスチャの使用は、劇的なカラーグラデーションを生み出す効果的な方法です。このアプローチは多くのゲームに役立ちます。

その他のリソース

— URP でのポストプロセスに関するドキュメント

— YouTube チュートリアル（PHLEARN）

— YouTube チュートリアル（GDQuest）

ライティング

URP でのライティングは、ビルトインレンダーパイプラインを使用する場合と似ています。一番大きな違いは設定を行う場所にあります。

このセクションでは、GPU プログレッシブライトマッパー、ライトプローブ、リフレクションプローブを使用して、リアルタイムライティングとシャドウに関連するレシピ（ベイクしたライティングや混合ライティングも含まれる）を扱います。フルコースディナーを用意できるほど十分なレシピを選ぶことができます。

開始する前に、シェーダーと色空間についての注意事項をいくつか説明しておきます。

シェーダー

URP でライティングを使用するとき、シェーダーの選択肢が示されます。通常、物理ベースレンダリング（PBR）モデルを使用する Lit と、Blinn-Phong モデルを使用する Simple Lit を混在させることはありません。以下の表で、URP のシェーダーについて説明します。

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、アーティストにとってリアルなレンダリングが容易になりますが、より定型化されたレンダリングが求められる場合は、Simple Lit のほうが好ましい結果が得られます。

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。たとえば、「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。

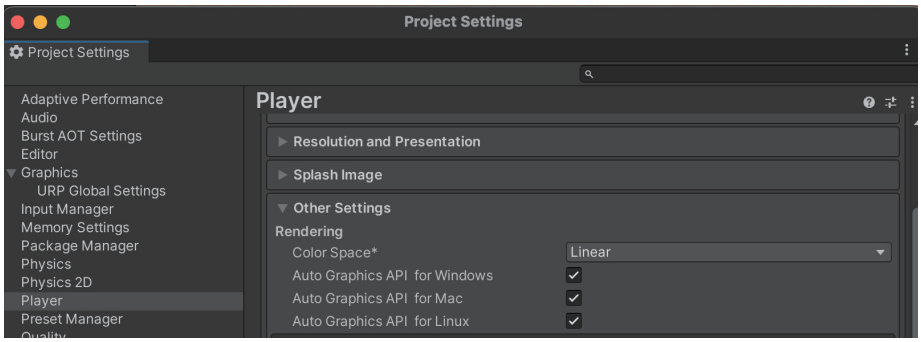
Unity と URP を使用して制作されたゲーム『*LEGO® Bricktales*』（Clockstone）では、プレイヤーがレゴの世界に没入できます。優れたライティングが、ブロックのリアルな質感を生み出すことに大きく貢献しています。

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フォトリアリスティックな品質でレンダリングできます。光量や反射は実物のように見え、眩しい日差しから暗澹とした洞窟まで、さまざまなライティング条件に対して反応します。 これは、ライティングを使用するほとんどのマテリアル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、フォトリアリスティックな結果としてはやや控えめな印象になります。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）。

異なるシェーダーを使っ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。

色空間

URP プロジェクトのライティングを設定するとき、リニア色空間とガンマ色空間のどちらを使用するかを選択できます。前者がデフォルトであり、強く推奨されています。これは、**「Edit」>「Project Settings…」>「Player」>「Color Space」** を使用して設定されます。URP 3D テンプレートを使用して作成されたプロジェクトはデフォルトで「Linear」に設定されています。

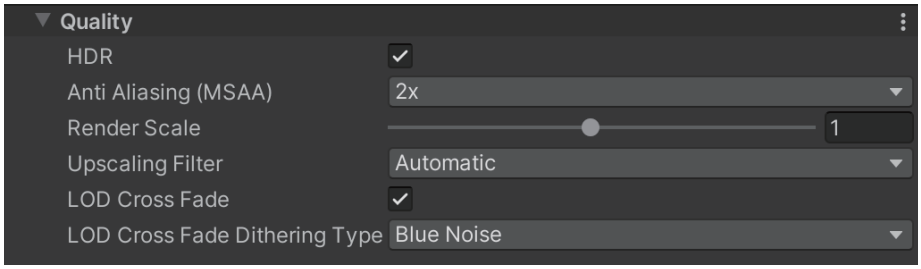


色空間の設定

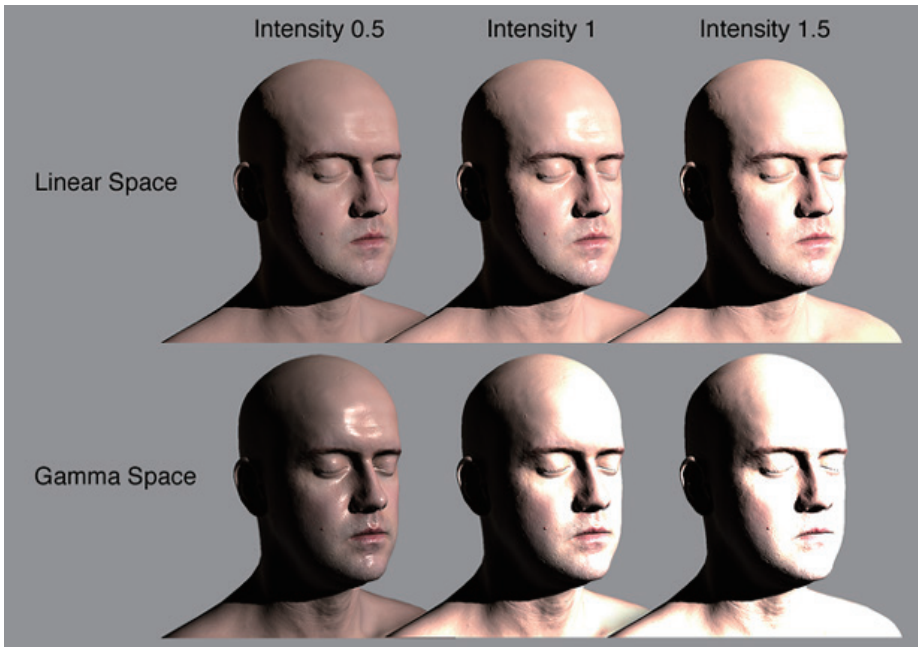
2 つの色空間の違いは、人間の目が光に反応する方法と関連しています。人間の目は特に明るい光を感知し、光の強度に直線的に反応するわけではありません。ガンマ色空間では、人間の目が識別しやすく表現できるようにリニア値を変更しますが、これは数学的には正確ではありません。このため、ハードウェア性能が向上し、PBR ライティングモデルが好まれるようになると、リニア色空間が使用されるようになりました。

ガンマを使用する必要があるのは、ターゲットハードウェアが古くて sRGB フォーマットがサポートされない場合などですが、ほとんどのケースでは完全にサポートされています。アートの観点からガンマに切り替えることもあります。プロジェクトが HDR ではなく LDR 向けに設定されており、Simple Lit ライティングモデルまたは Unlit を使用する場合などです。そのようなケースでは、ガンマ色空間への切り替えは、目指すアートスタイルを実現するためにメリットがあります。

LDR または HDR は、URP 設定アセットの「Quality」セクションを使用して設定されます。



HDR を使用するように URP を設定



リニア空間とガンマ色空間での異なる強度のライティングの比較

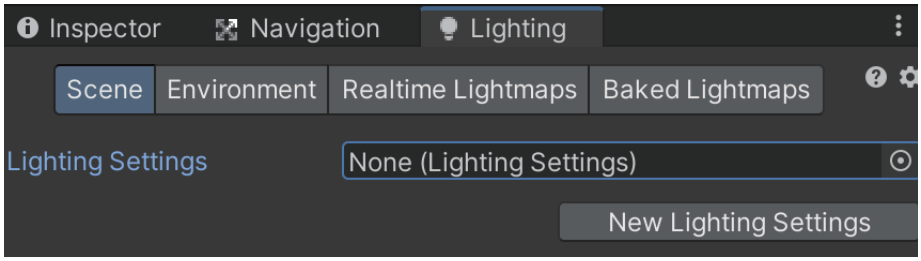
ガンマおよびリニアの色空間やガンマおよびリニアのテクスチャを使用するワークフローの詳しい説明については、こちらのドキュメントを参照してください。

ジオラマシーン

リアルタイムグローバルイルミネーションおよび混合ライティング

このレシピのほとんどで使用されるスクリーンショットは、Unity プロジェクト「*FPS Sample: The Inspection*」のものです（こちらからダウンロードできます）。このサンプルは Unity 2020 LTS 用に作成されましたが、ライティングの原則は Unity 2022 LTS に適用可能です。

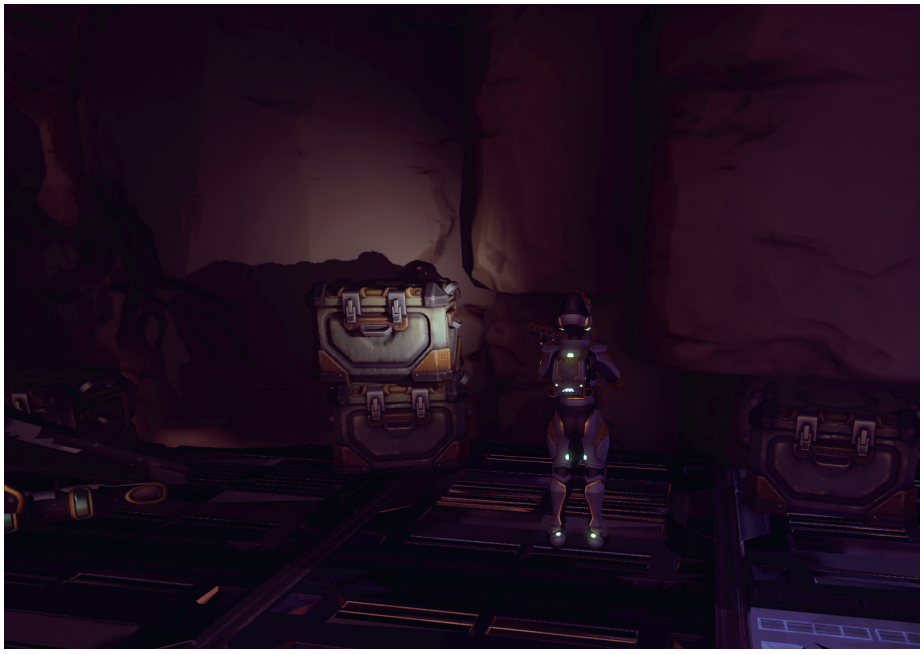
「Scenes」>「Small_Indirect」シーンを開くと、洞窟、メカニックアーム、ロボットが配置されたジオラマが表示されます。



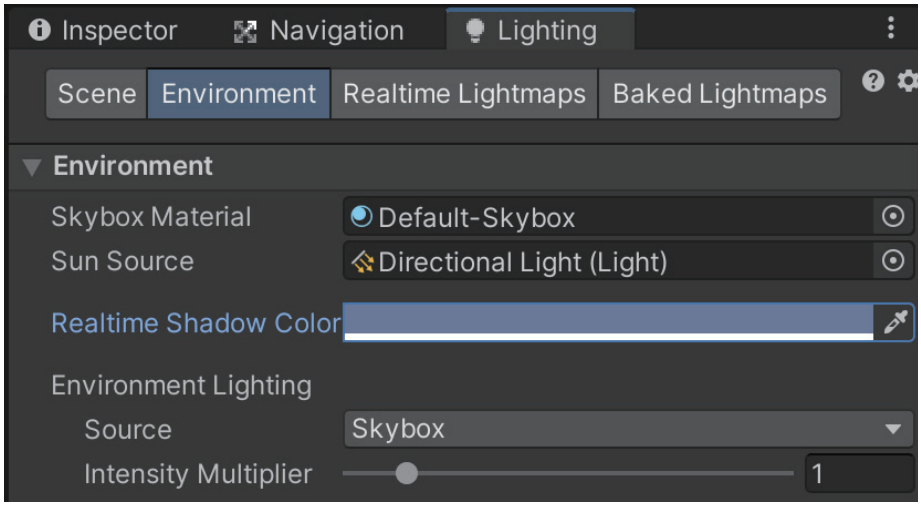
ライティング設定アセットの作成

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。**「Window」>「Rendering」>「Lighting」**を開き、「**Scene**」タブをクリックし、「**New Lighting Settings**」をクリックして、新しいアセットに名前を付けます。これで、「Lighting」パネルで適用した設定が保存されます。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。

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。



洞窟の壁を照らす武器のライト



「Sun Source」の設定

メインライト（ジオラマシーンの「Sun」）は、ライトモード「Mixed」として設定されています。つまり、リアルタイムライティングとベイクしたライティングの両方に対応します。シーンに動的オブジェクトがある場合は、少なくとも 1 つのリアルタイムライトが必要です。これが動的オブジェクトを照らし、オブジェクトが動くとその影が更新されます。ロボットの武器もライトを備えており、「Realtime」として設定されます。これは洞窟の中では一番目立ちます。シーンを再生し、WASD キーを使用してロボットを動かすと、洞窟内の小道具にどのように影ができるかを確認できます。

URP では、ライティング設定は 3 箇所で調整します。

- **「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。
- **「Inspector」ビューの Light コンポーネント：**ゲームオブジェクトにアタッチされた Light コンポーネントはライトとして作動します。詳細については以下の表を参照してください。
- **URP アセットインスペクター：**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 Inspector で選択した設定によって大きく影響を受けます。

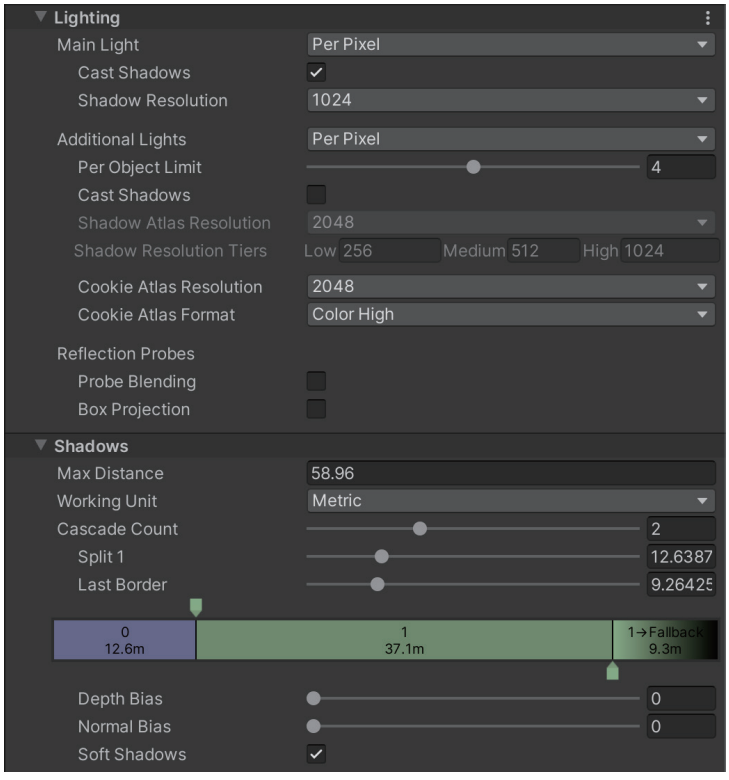
次の表は、URP とビルトインレンダーパイプラインのライトインスペクターの違いを示したものです。

URP のライトインスペクターのプロパティ	**説明**
General	
Type	Spot、Directional、Point、または Area
Mode	Baked、Mixed、または Realtime
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ます。
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。

Emission	
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。
Color	放出される光の色をカラーピッカーで設定します。
Intensity	ライトの明度を設定します。ディレクショナルライトの初期値は 0.5 です。ポイント、スポット、エリア（矩形または円盤状）の各ライトの初期値は 1 です。
Indirect Multiplier	間接光の輝度を変化させるために使用します。「Indirect Multiplier」を 1 より小さい値に設定すると、反射光は跳ね返るごとに暗くなります。1 より大きい値にすると、光は跳ね返るごとに明るくなります。これは例えば、ディテールが見えるように、影の中にある暗い面（洞窟の中など）を明るくする必要がある場合に便利です。
Range	ライトがレンダリングに影響する、ゲームオブジェクト位置からの距離を制御します。
Rendering	
Render Mode	Auto – ライトとカメラの近接性によってランタイムに決定、Important – 常にピクセル単位品質、Not important – 常に早く処理される、頂点単位品質
Culling Mask	どのレイヤーがライトの影響を受けるかを制御するために使用されます。
Shadows	
Shadow type	「No shadows」、「Soft shadows」、または「Hard shadows」。
Baked Shadow Radius	「Type」が「Point」または「Spot」に設定され、「Shadow Type」が「Soft Shadows」に設定されている場合、このプロパティによって影のエッジが人工的にぼかされ、より自然な見た目になります。
Light Cookie	
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 では色付きクッキーがサポートされます。

シャドウ

影の設定は、URP の使用時に、Renderer Data オブジェクトと URP アセットを使用して設定します。これらのアセットを使用して、影の忠実度を定義できます。



URP アセット

『*Syberia: The World Before*』（Microids）は、Unity を使用して制作されたもう 1 つのゲームの例です。これは専用機および PC 向けのストーリー性が高いゲームで、光と影が効果的に使われることで美しい建築物や街路が再現されています。

メインライト：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。シャドウマップの解像度設定は最後にするようにしましょう。

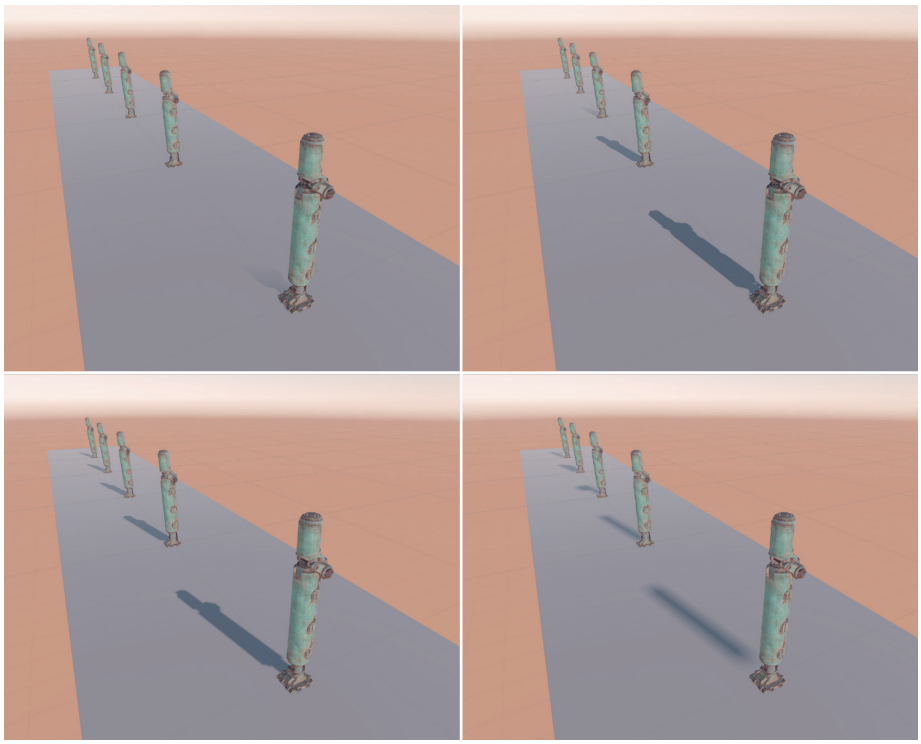
Unity で影を操作したことがある方なら、リアルタイムシャドウでは、ライトの視点から見たオブジェクトの深度を含む、シャドウマップをレンダリングする必要があることをご存知だと思います。このシャドウマップの解像度が高いほど、より現実感のある見た目になります。ただし、処理能力とメモリの両方で拡張が必要となります。影の処理が増える要因としては、以下のものがあります。

1. シャドウマップでレンダリングされるシャドウキャスターの数。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。
2. 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）
3. シャドウカスケードの分割
4. シャドウフィルタリング（ソフトシャドウ）

最高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、左上の画像が 256、左下の画像が 1024、右上の画像が 2048、右下の画像が 4096 に設定されています。中央の画像は 1024。

メインライト：シャドウマックスディスタンス



メインライトのシャドウの最大距離の変化：左上の画像 – 10、右上の画像 – 30、左下の画像 – 60、右下の画像 – 400

メインライトシャドウのもう 1 つの重要な設定は、「**Max Distance**」です。これはシーンユニットで設定されます。上の画像では、ポールの間隔が 10 ユニットです。Max Distance は、10 ユニットのものから 400 ユニットのものまであります。左上の画像では、1 つ目のポールだけが影を落としていて、カメラの位置から 10 ユニットの距離で影が途切れています。60 ユニット（左下の画像）では、すべての影が表示されていて、影の忠実度が適切です。Max Distance が見えているアセットよりも度を超えて大きい場合、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ユニットで直に関連付けられている必要があります。十分と思われる影となる限りで最小距離を目標にして設定してください（下記の注記を参照）。カメラから 60 ユニット離れた動的オブジェクトの影までしかプレイヤーに表示されない場合は、最大距離を 60 に設定します。混合ライトのライティングモードが「Shadowmask」に設定されている場合、シャドウディスタンスを超えてい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。

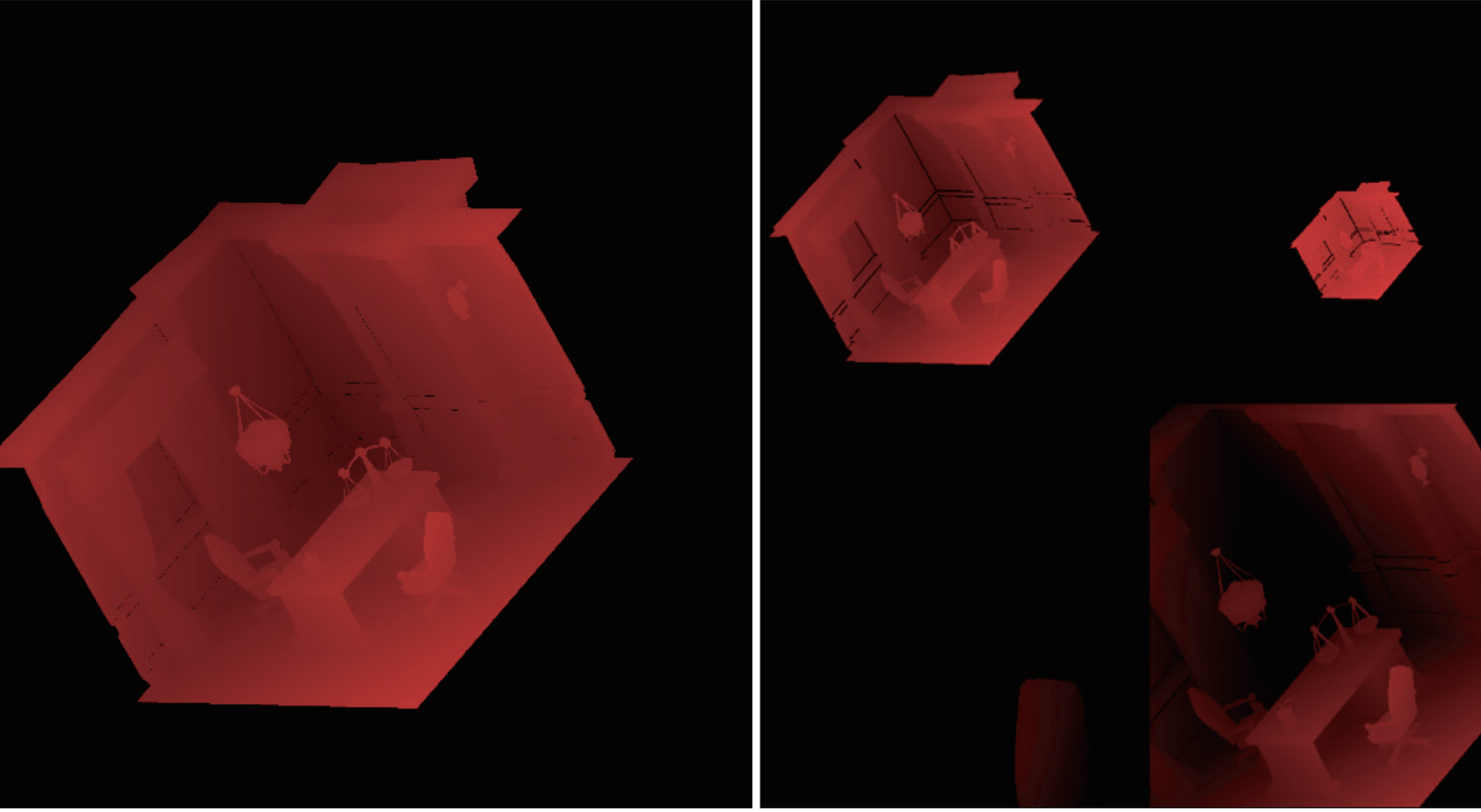
> **ノート：**URP では、「Shadow Projection」として「**Stable Fit**」のみがサポートされています。これは、「Max Distance」の設定をユーザーに委ねるものです。ビルトインレンダーパイプラインでは、「Shadow Projection」プロパティとして「Stable Fit」と「Close Fit」の両方がサポートされています。後者のモードの場合、左下と右下の画像は同じ品質になります。「Close Fit」では、シャドウディスタンス平面が最後のキャスターに合わせて縮小されるためです。欠点は、「Close Fit」の場合、影の錐台が「動的に」変えられ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

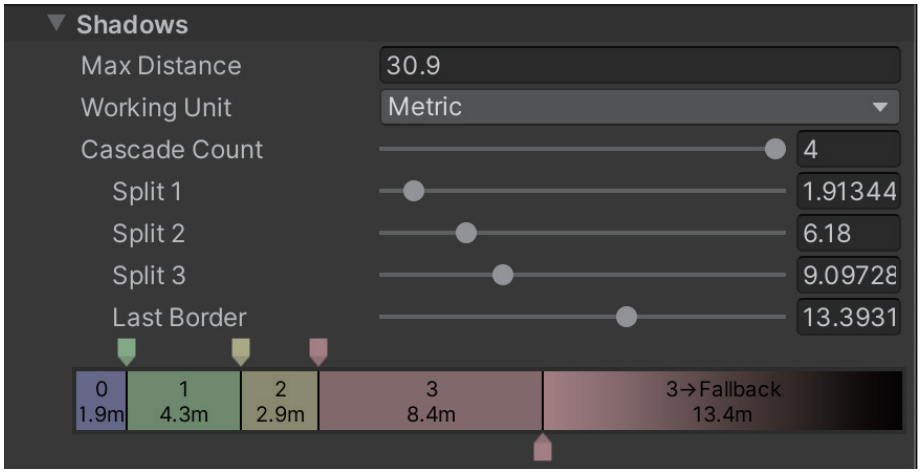
下の画像は、最初のレシピで目にした、椅子と机が置かれた幽霊屋敷の部屋のシーンのシャドウマップです（**「Scenes」>「Renderer Features Stencil」>「SmallRoom-Stencil」**）。左側の画像では、カスケードカウントは 1 です。マップがエリア全体を専有しています。右の画像では、カスケードカウントは 4 です。このマップには 4 つの異なるマップが含まれていて、各エリアには、より低解像度のマップが割り当てられいます。

このような小規模なシーンでは、多くの場合、カスケードカウントが 1 でも最適な結果が得られます。ただし、Max Distance が大きい場合は、カスケードカウントを 2 や 3 にした方が、割り当てられるシャドウマップの割合が大きくなるので、前景のオブジェクト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。



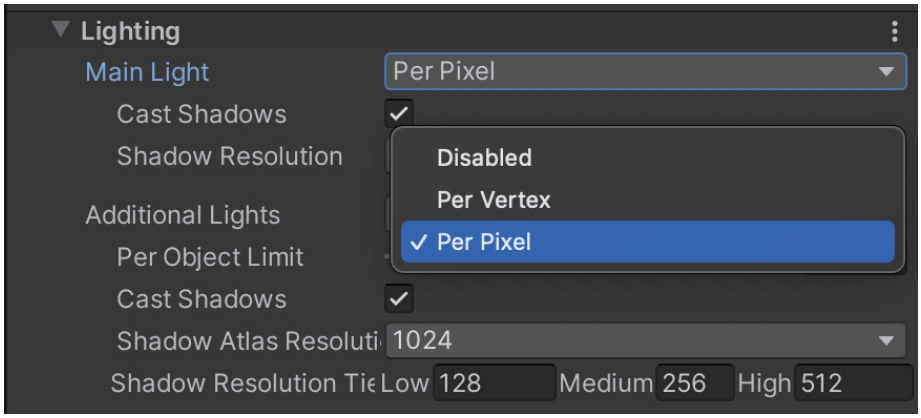
カスケードカウントを 1 に設定した場合（左の画像）と、4 に設定した場合（右の画像）のシャドウマップ

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ユニット数を設定することで調整できます（以下の画像を参照）。「Max Distance」は常にシーンに適した値に調整し、スライダーの位置は慎重に選択するようにしてください。「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

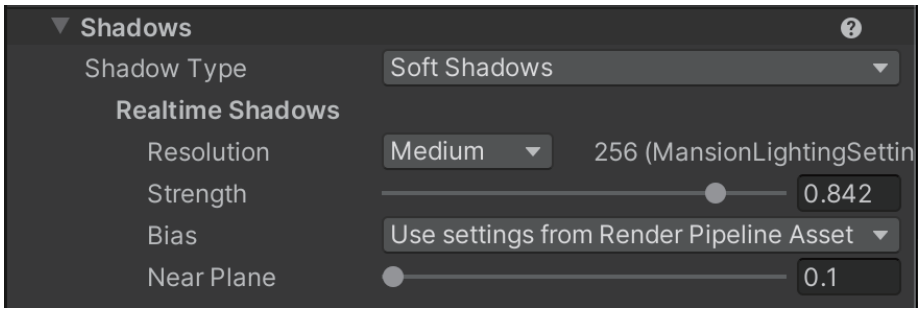
追加ライトのシャドウ



URP アセットの追加ライトで使用できる設定

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ライトインスペクターから選択した解像度が使用されます。

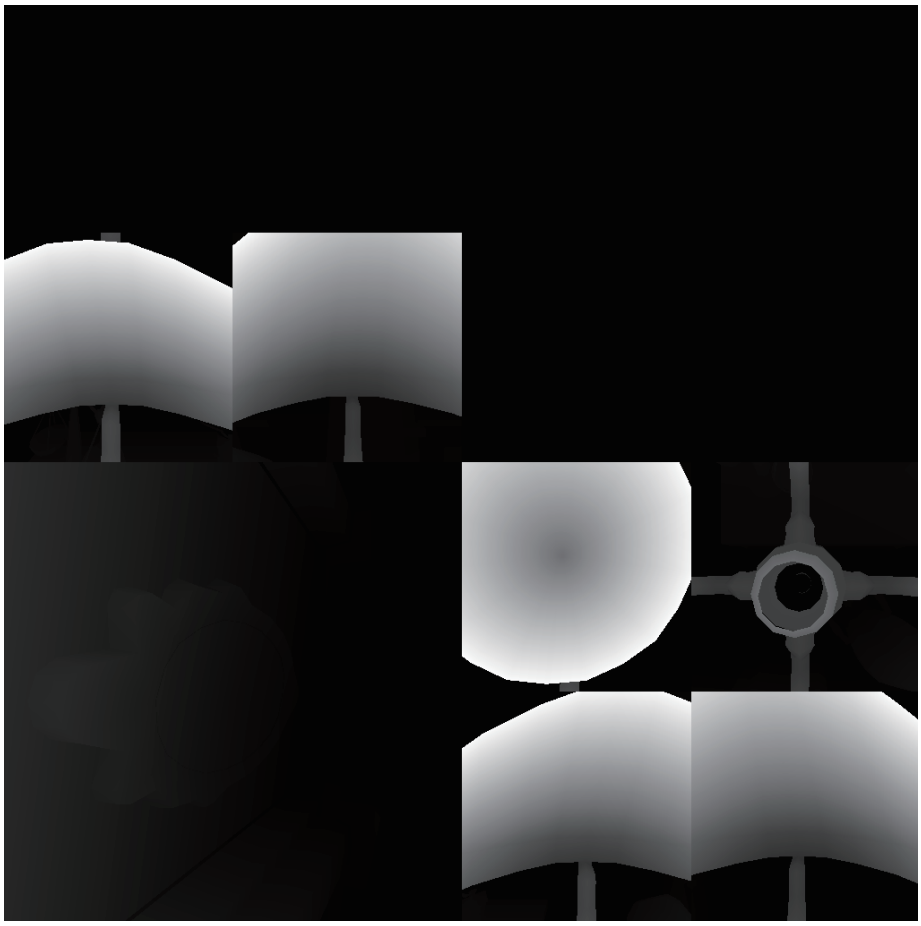


ライトインスペクターの「Shadows」グループ

この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（これは、e ブック『*Introduction to the Universal Render Pipeline for advanced Unity creators（上級 Unity クリエイター向け URP 入門）*』で説明しているプロジェクトの 1 つです）。また、7 つのマップもあります。これらの 7 つのマップを 1024px の正方形のマップに適合させるには、各マップのサイズを 256px 以下にする必要があります。このサイズを超えると、シャドウマップの解像度がアトラスに合わせて縮小され、コンソールに警告メッセージが表示されます。

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

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



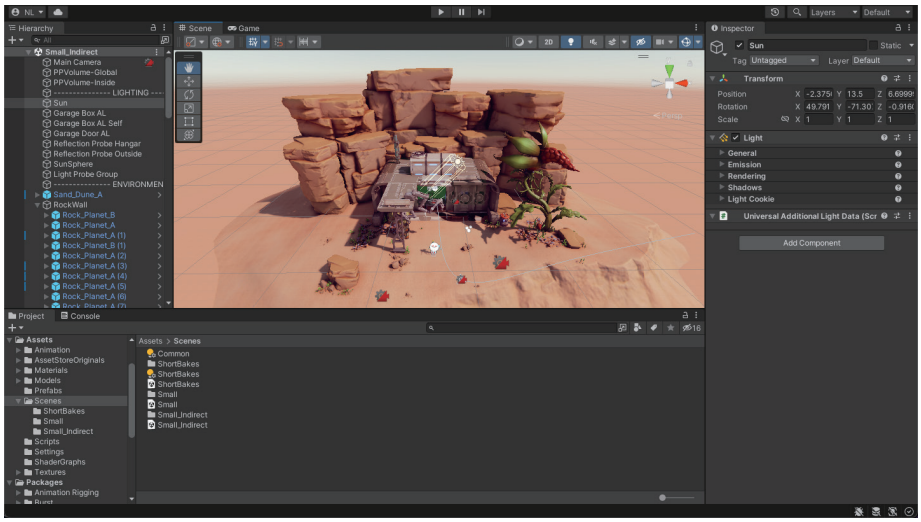
追加ライト用のシャドウアトラス

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

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

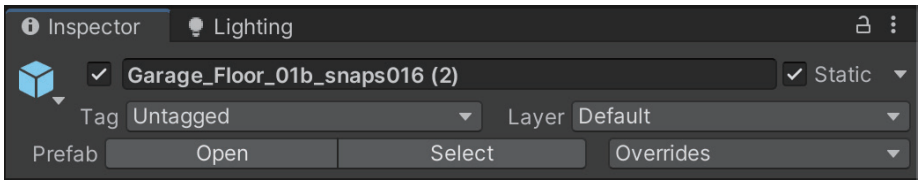
ベイクしたライティング

Unity の FPS Sample プロジェクトを使って手順を説明しましょう。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。



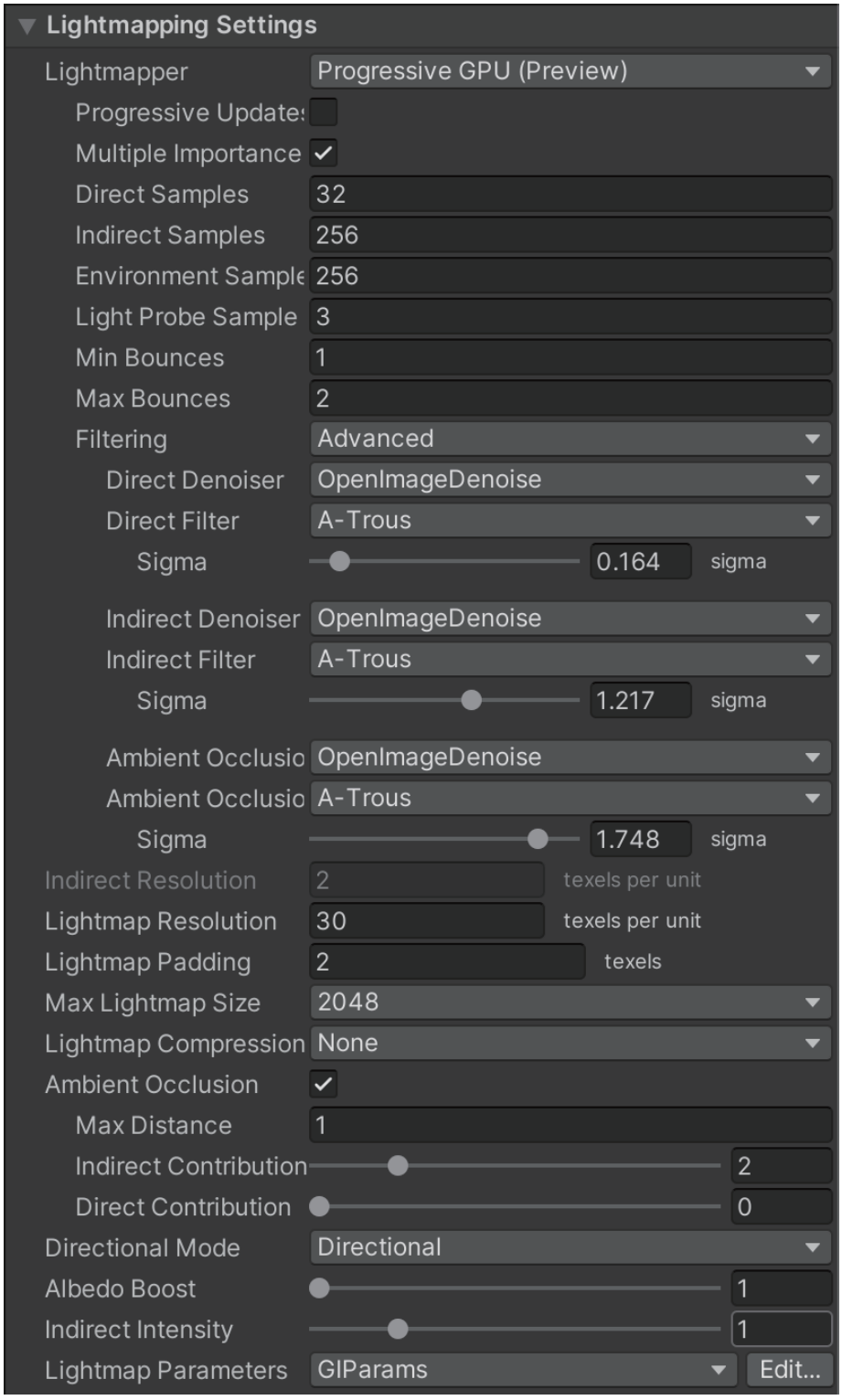
FPS Sample：『*The Inspection*』（Unity）のシーン

FPS Sample プロジェクトのシーンでは、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。



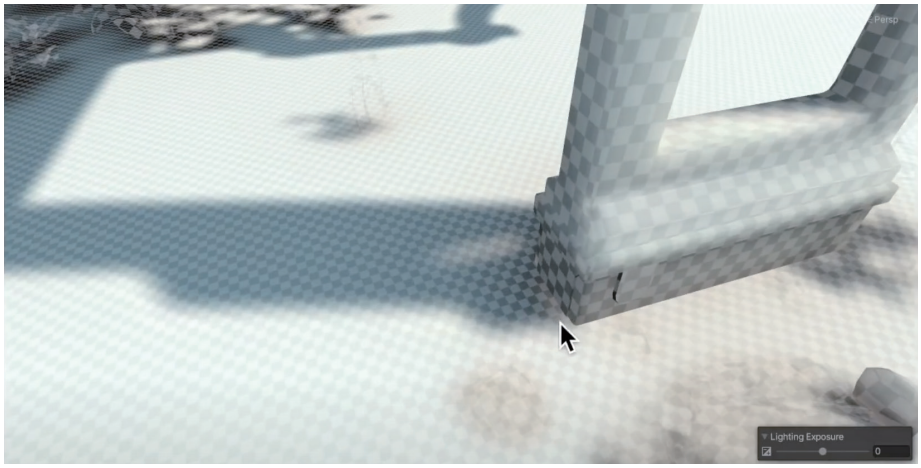
ライトマッピングにジオメトリを含める

「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）。



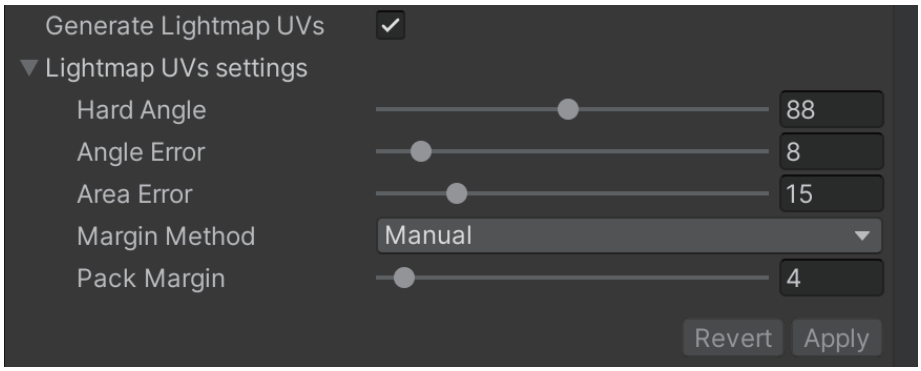
ライトマッピングの設定

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。



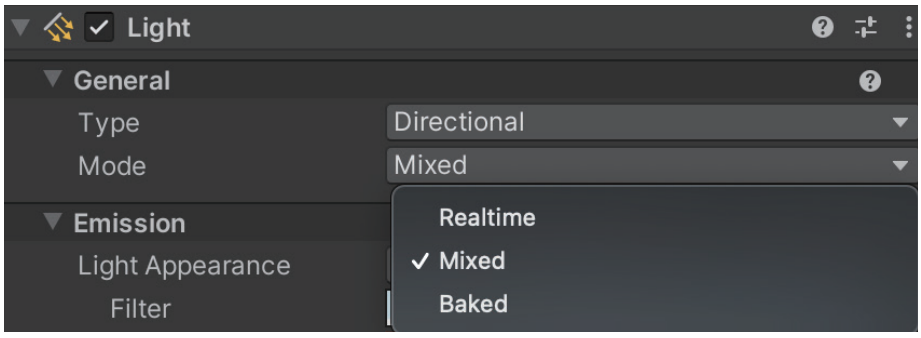
フィルタリングがオブジェクト間の影に与える影響

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。



Generate Lightmap UVs

「**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。



「Light Mode」を「Baked」または「Mixed」に設定

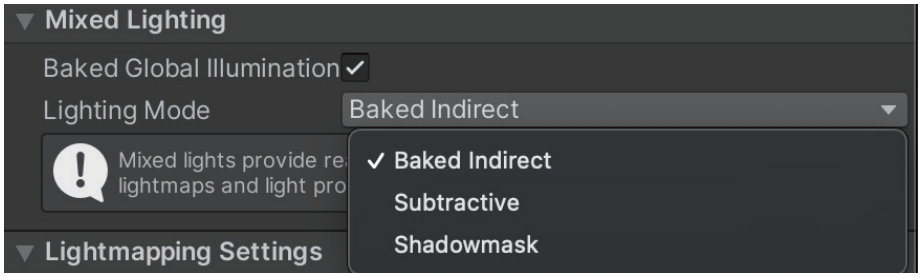
混合ライトを使用する場合は、**「Window」 > 「Rendering」 > 「Lighting」 > 「Scene」**から、**「Light Mode」**を**「Baked Indirect」**、**「Subtractive」**、または**「Shadowmask」**に設定します。

— **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。ただし、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。

— **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。この場合、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。

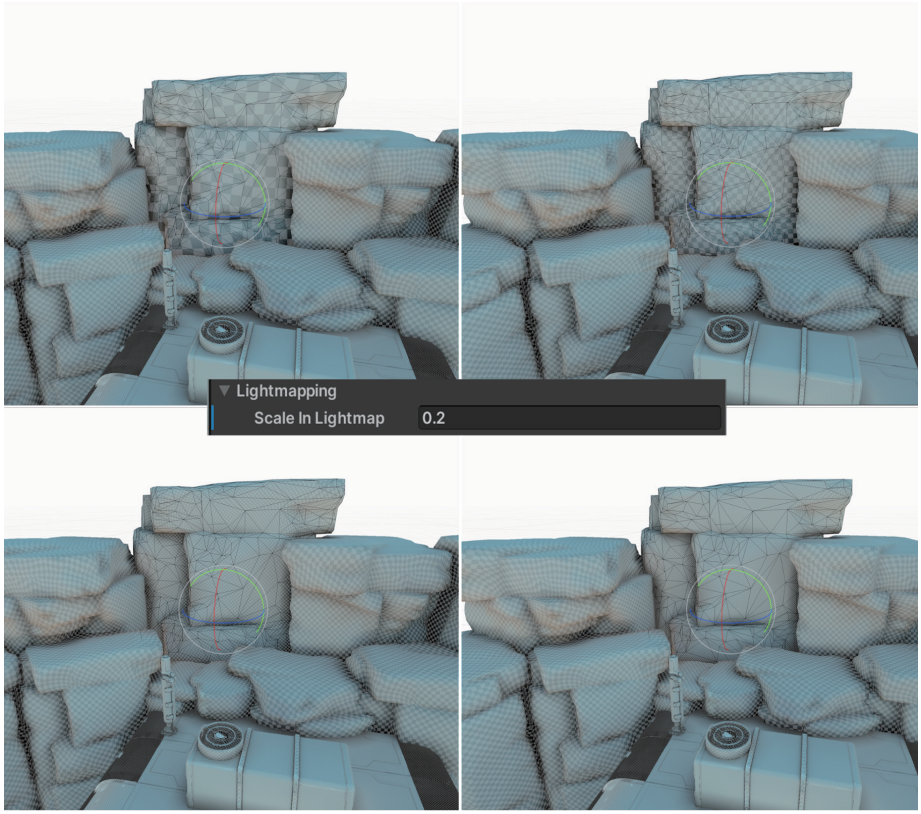
  URP では、ディレクショナルライトからの光の影響の推定値が計算され、ベイクしたグローバルイルミネーションからその値が差し引かれます。この推定値は、「Lighting」ウィンドウの「Environment」セクションにある「Real-time Shadow Color」設定によって固定されるため、減算された色がその色よりも暗くなることはありません。減算された値の最小の色と、元のベイクした色を選択します。これはローエンドハードウェアに対する最適なオプションです。

— **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。



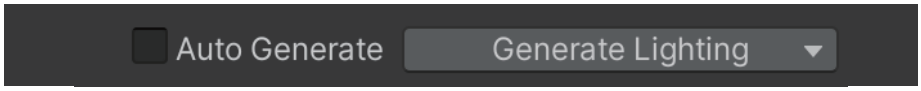
「Baked Indirect」グローバルイルミネーション

「Asset」 > 「Inspector」 > 「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少なくします。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。



スケール設定のテクセルサイズ：テクセルサイズは、左上の画像が 0.5、右上の画像が 0.2、左下の画像が 0.1、右下の画像が 0.05 に設定されています。

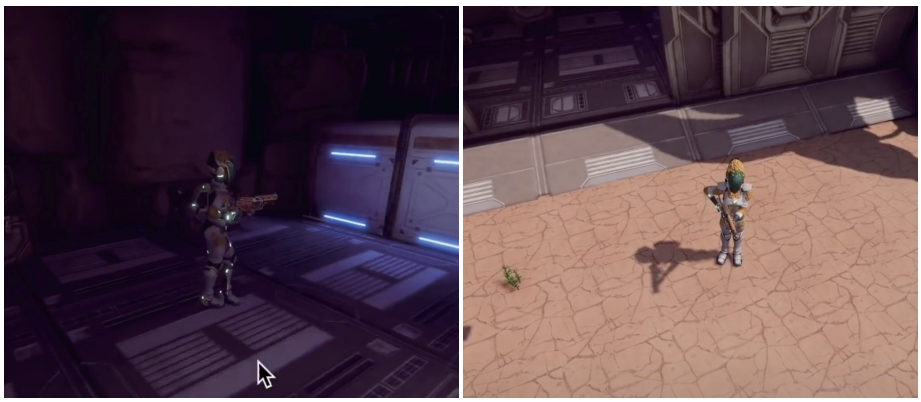
「**Generate Lighting**」をクリックしてベイクします。ベイクの処理時間は、静的オブジェクトの数、メッシュの複雑度、ライトの設定（「Mixed」モードまたは「Baked」モード）、およびライトマッピングの設定（特に「Max Lightmap Size」と「Lightmap Resolution」）によって変わってきます。



「Generate Lighting」でベイクとができます。

ライトプローブ

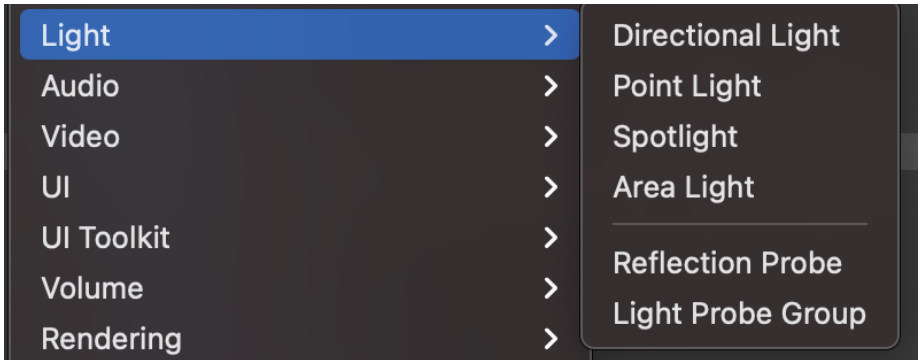
「Mixed」モードライティングを使用する場合は、シーンにライトプローブを追加することをお勧めします。ライトプローブは、開発者が**「Window」>「Rendering」>「Lighting」**パネルから「**Generate Lighting**」をクリックしてライティングをベイクしたときに、環境内の特定の位置にライトデータを保存します。これにより、環境内を移動する動的オブジェクトのイルミネーションに、ベイクされたオブジェクトで使用されているライティングレベルが反映されるようになります。暗い場所ではオブジェクトが暗くなり、明るい場所では明るくなります。下の画像（「*FPS Sample: 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

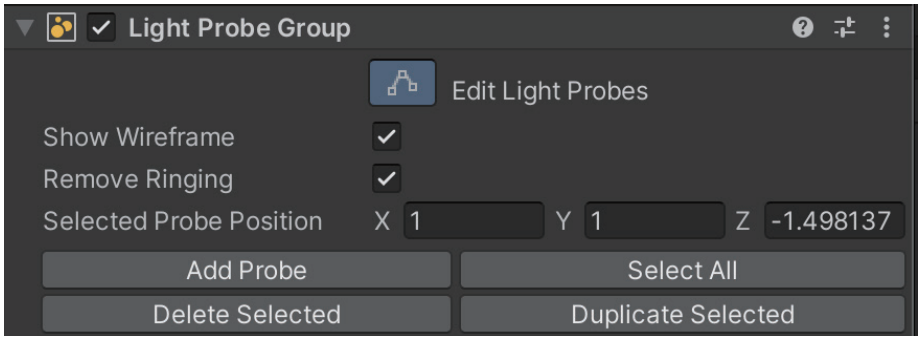
ライトプローブを作成するには、**Hierarchy** ウィンドウを右クリックし、**「Light」>「Light Probe Group」**を選択します。

『*Slime Rancher 2*』（Monomi Park）は、カスタマイズしたレンダーパイプラインを使用して Unity で制作された、カラフルで速いペースの一人称視点アドベンチャーゲームです。アクションゲームで美しいライティングを実現するために、動的オブジェクトのためのライトプローブや静的オブジェクトのためのベイクしたライトなど、さまざまなライティングテクニックを組み合わせるこ



「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。

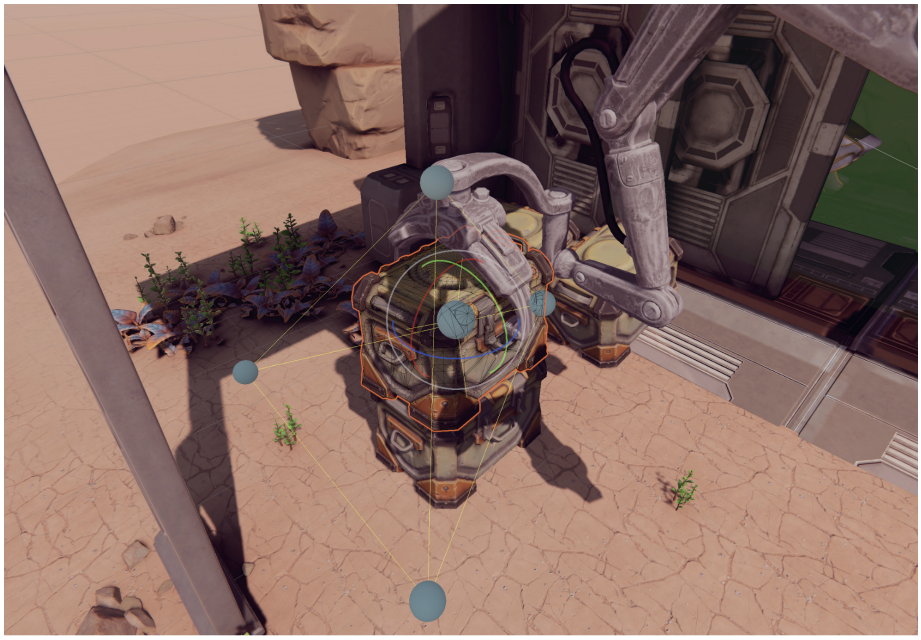


Inspector でのライトプローブの追加または削除および位置の変更

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

ライトプローブの移動

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



選択したクレートに最も近いライトプローブ

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、こちら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、こちらのドキュメントを参照してください。

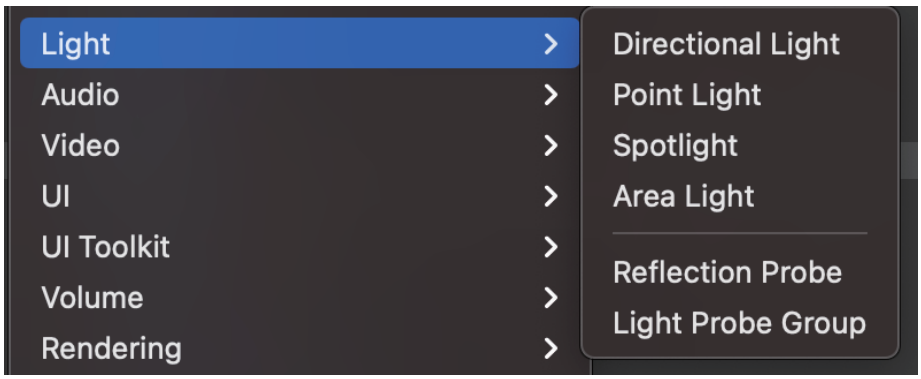
リフレクションプローブ

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。

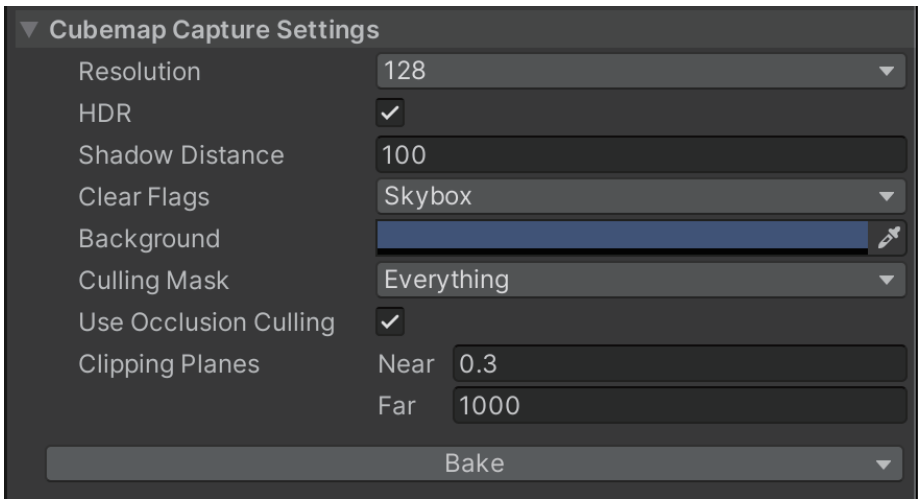
リフレクションプローブは、シーン内のキー位置に配置される、事前レンダリング済みのキューブマップです。リフレクションプローブは 1 つのシーンで複数使用できます。動的オブジェクトがシーン内を移動する際に、最も近いリフレクションプローブを選択し、それを反射の基準として使用することができます。また、プローブ間をブレンドするようにシーンを設定することもできます。

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。



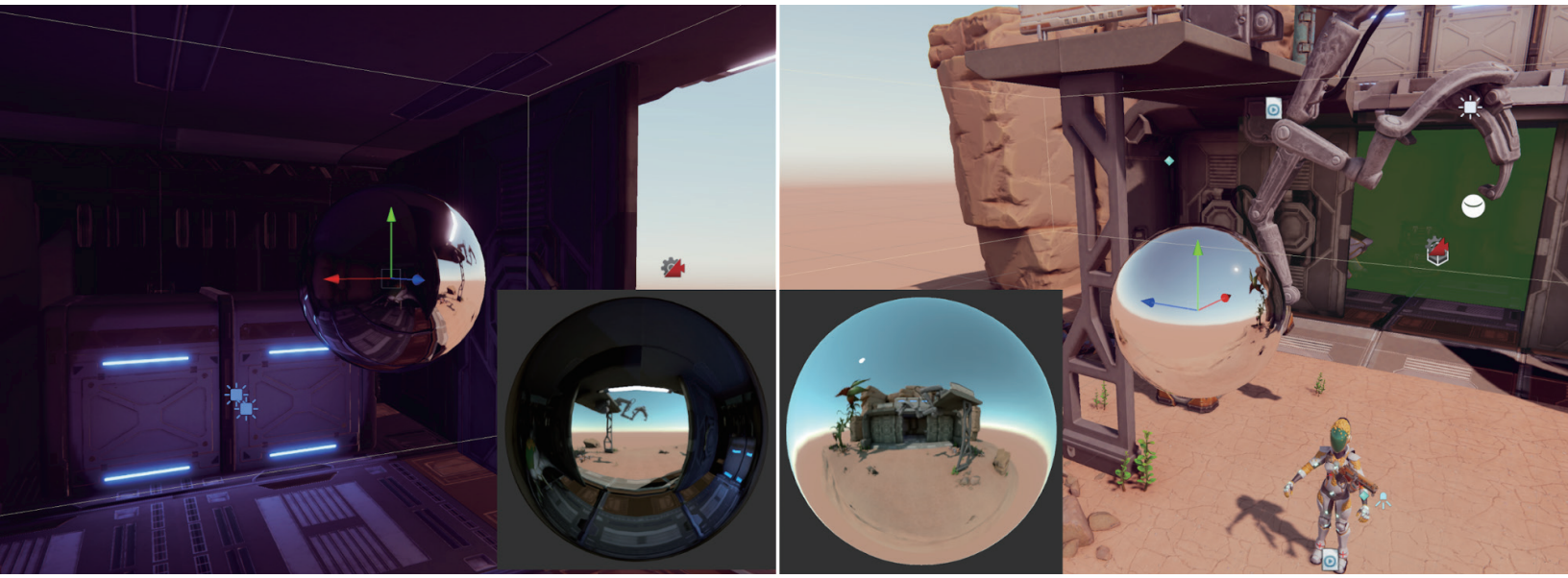
リフレクションプローブの作成

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 Sample: 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**」パネルで有効にできます。

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、オブジェクトにさまざまなサイズの反射を映すことができます。周囲のキューブマップのサイズは、プローブの影響ゾーンによって決まります（「**Box Size**」プロパティに基づきます）。例えば、部屋の内部を反映するプローブを使う場合は、部屋の寸法に合わせてサイズを設定する必要があります。

その他のリソース

- ライティングおよびライトマッピングのドキュメント
- Unity Learn の「ライティングとレンダリングの概要」
- CGCookie の「The art of lighting game environments（ゲーム環境でのライティング技術）」
- Brackeys の「Real-time lighting in Unity（Unity でのリアルタイムライティング）」
- Unite Now の「Harnessing light with the URP and the GPU Lightmapper（URP と GPU ライトマッパーによるライトの活用）」
- Unity Learn の「ライトマップの設定」
- 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

SCREEN SPACE REFRACTION（スクリーンスペースリフラクション）

スクリーンスペースリフラクションは、レンダーパイプラインによって作成された現在の不透明テクスチャをソーステクスチャとして、レンダリングされるモデルにピクセルをマップします。不透明テクスチャに含まれないモデルを表示することはできません。この方法では、画像のサンプリングに使用される UV を変形させます。

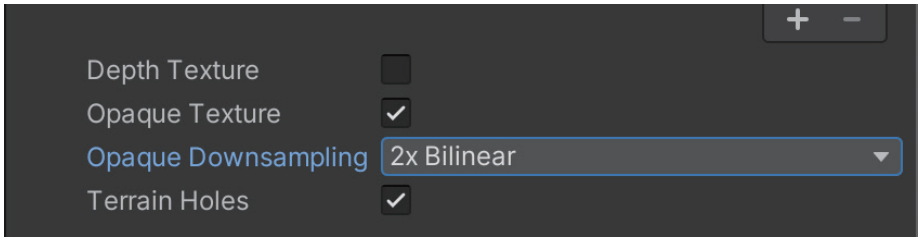
スクリーンスペースリフラクションの例

インディーゲーム『*Arctico*』（開発者 Claudio および Antonio）では、ベースキャンプを設営して氷河地形を探検しなければなりません。このゲームでは豊富な水のサーフェスに映り込みがありますが、これはスクリーンスペースリフレクションによって、実現できるエフェクトです。スクリーンスペースリフレクションは反射面のシミュレーションに用いられる一方、スクリーンスペースリフラクションは透明度や、媒質を通過する光の屈折のシミュレーションにも使用されます。

このレシピでは、法線マップを使用した、リフラクション（屈折）エフェクトの作成とリフラクションエフェクトの着色の方法について説明します。前の画像に表示されている追加の着色は、「**Color**」プロパティで、計算済みのピクセルカラーを線形補完して行います。

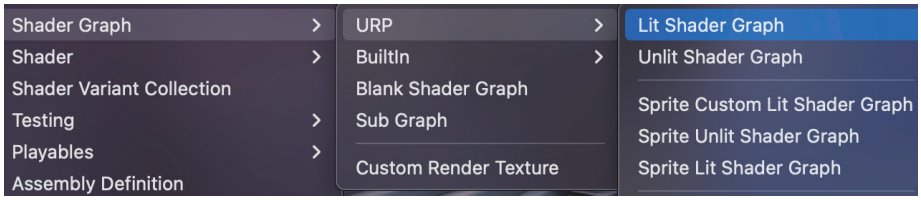
エフェクトの動作を表示するには、**「Scenes」>「Refraction」>「Refraction」**を確認してください。

この方法では、不透明テクスチャをシェーダーで使用可能にする必要があります。現在割り当てられている URP 設定アセットは、**「Edit」>「Project Settings…」>「Graphics」>「Scriptable Render Pipeline Settings」**で見つかります。Inspector で、「**Opaque Texture**」が有効になっていることを確認します。「Opaque Downsampling」も有効にすると、パフォーマンスが少し向上します。また、こうすると屈折性オブジェクトを通して見えるものが少しぼやけ、見た目がさらによくなることがあります。



「Opaque Texture」および「Opaque Downsampling」の設定

シェーダーを作成する最初のステップは、新しいシェーダーグラフアセットの作成です。Project ウィンドウを右クリックし、**「Create」>「Shader Graph」>「URP」>「Lit Shader Graph」**を選択します。



新しい Lit シェーダーグラフの作成

このシェーダーを使用してマテリアルを作成するには、シェーダーグラフアセットを選択して、**「Create」>「Material」**を選択します。このマテリアルを、屈折性を持たせるオブジェクトに適用します。

ここで、シェーダーグラフアセットをダブルクリックして開きます。**「Scene Color」**ノードを作成し、これを**「Fragment」>「Base Color」**に接続します。

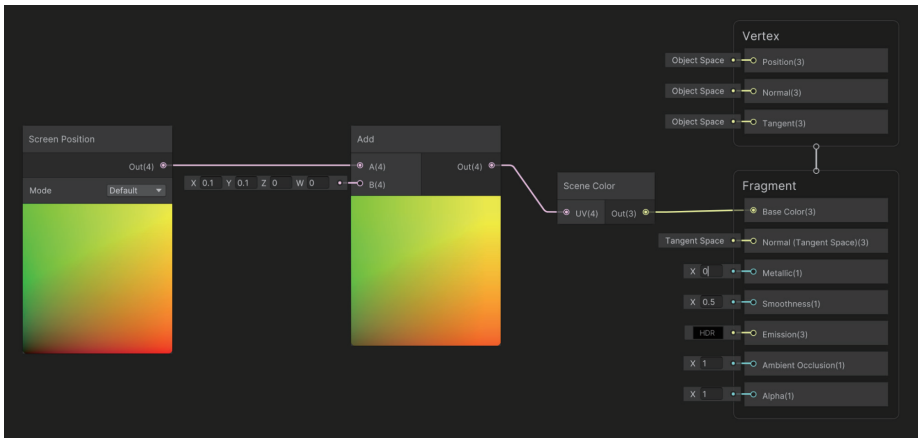
「Scene Color」ノードの使用

「Scene Color」は、透明なマテリアルのみを処理します。すでにレンダリングされた不透明オブジェクトを利用するためです。レンダーパイプラインでは、透明オブジェクトは不透明オブジェクトの後でレンダリングされます。**「Graph Inspector」>「Graph Settings」>「Surface Type」**を「**Transparent**」に設定します。

「Scene Color」ノードは、デフォルトでは UV 用に正規化されたスクリーン座標を使用し、滑らかさの影響を受けるライティングで、不透明テクスチャを各ピクセルにマップします。この結果として以下の画像が生成されます。

「Scene Color」を使用した結果

目的は「Scene Color」ノードによって使用される UV の操作であるため、デフォルトの UV 動作をオーバーライドする必要があります。「**Screen Position**」ノードと「**Add**」ノードを作成します。「Screen Position」ノードの出力を「Add」ノードの入力 A にドラッグし、B 入力を [0.1, 0.1, 0, 0] として設定します。



UV を制御するノードの追加

ここでは、「Opaque Texture」オフセットを確認します。

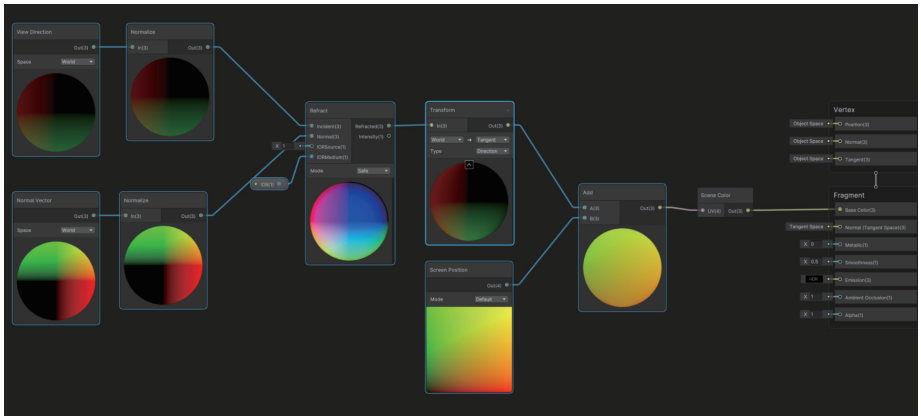
「Opaque Texture」オフセット

レンダリングされるピクセルごとに、カメラのビュー方向、現在の画面位置でのオブジェクトの法線、およびスケーリング値によってオフセットを制御する必要があります。シェーダーグラフには、これら 3 つの入力に基づいて屈折を計算するノードが含まれます。実際には 2 つのスケーリング値がありますが、使用するのは 1 つのみです。

新しいフロートプロパティ「**IOR**」（スケーリングの屈折率（Index of Refraction）の短縮表記）を追加できます。これを最小 1、最大 6 のスライダーとして設定します。ビュー方向については、「**View Direction**」ノードを追加して、「**Normalize**」ノードにリンクし、単位が長さであることを保証します。

「Normal」ノードセットを「World Space」に追加し、再び「Normalize」ノードに追加します。「Refract」ノードを作成し、正規化された「View Direction」を「Incident」の入力にリンクします。正規化された「Normal」を「Refract」ノードの「Normal」入力にリンクし、「IOR」プロパティを「IORMedium」入力にリンクします。

この時点で、「Refracted」出力がワールド空間にありますが、スクリーンスペース UV をオフセットするには、これを接空間に配置する必要があります。入力を「World」、出力を「Tangent」として設定する「Transform」ノードを追加します。タイプについて「Direction」を選択します。これを、「Screen Position」が B 入力である「Add」ノードの A 入力として使用します。グラフは以下の表示のようになります。

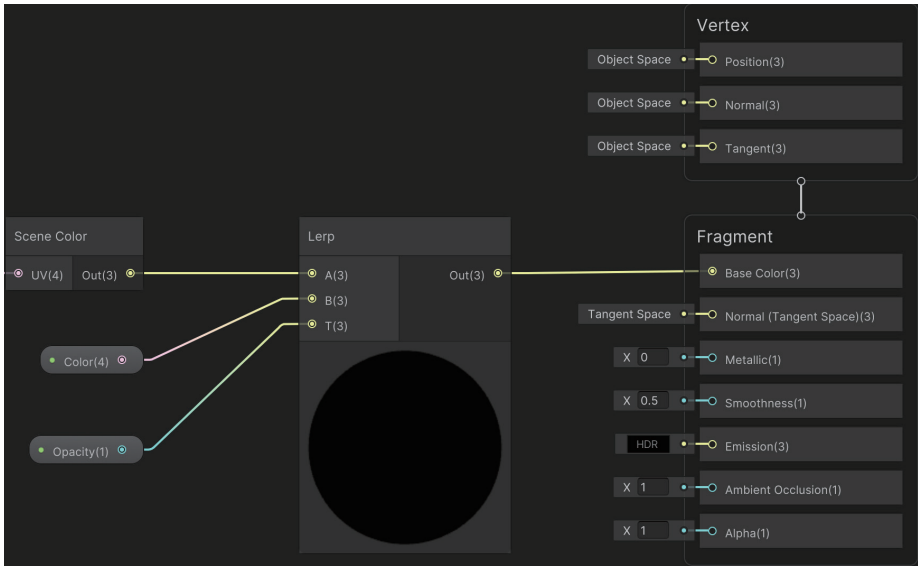


基本的な屈折グラフ

IOR が 5.44 の場合、以下の画像の視覚エフェクトが生成されます。

基本のスクリーンスペースリフラクション

「**Color**」プロパティを追加して結果に着色できます。「Lerp」ノードを追加し、スライダーモード（範囲 0 ～ 1）に設定された「**Opacity**」プロパティを T 入力として使用します。「Scene Color」の出力は入力 A、「Color」の出力は入力 B として設定されます。



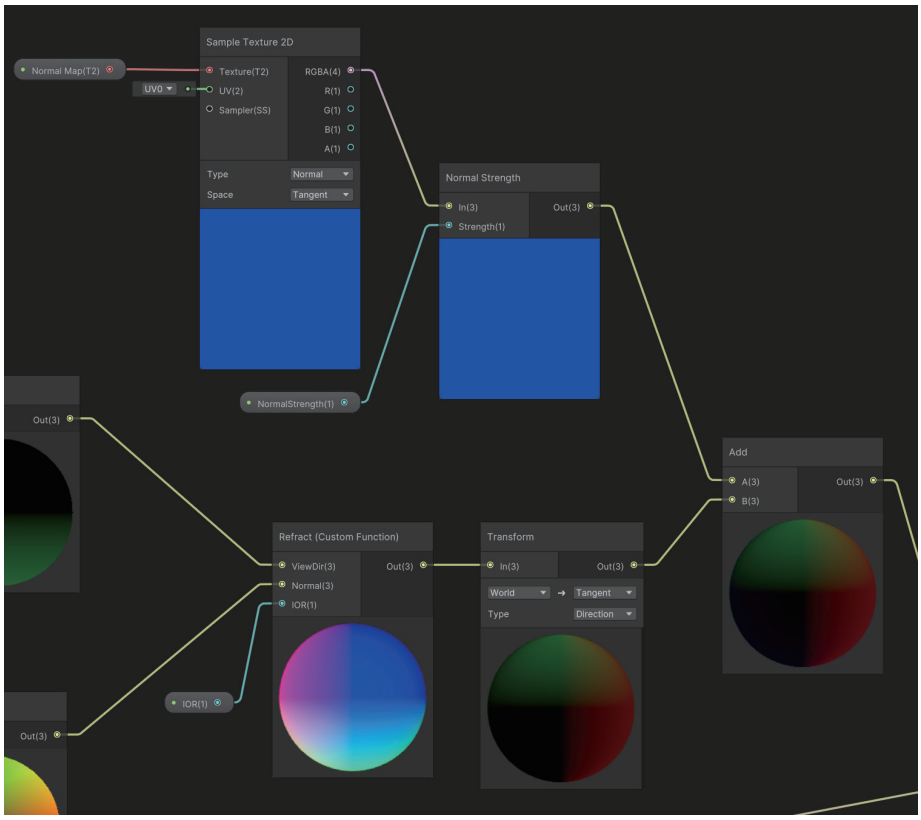
グラフに着色ステージを追加

これで出力を着色できるようになりました。

着色されたバージョン

法線は屈折に影響します。つまり、1 つの平面は「Opaque Texture」のオフセットバージョンを取得するだけです。

ここで法線マップを追加します。最初に、「**Texture 2D**」プロパティを、「**Normal Map**」と名付けたシェーダーに追加し、「**Normal Strength**」という名前のフロートプロパティをスライダー（範囲 0 ～ 1）として追加します。「Sample Texture 2D」ノードを作成し、「Type」を「Normal」、「Space」を「Tangent」に設定します。「Texture」入力を「Normal Map」プロパティに対して設定します。「**Normal Strength**」ノードを作成し、入力として「Sample Texture 2D」ノードの「RGBA(4)」出力を設定します。「Strength」入力として「Normal Strength」プロパティを設定します。入力 A が「Normal Strength」ノードの出力、入力 B が「Transform」ノードの「World」から「Tangent」ノードからの出力として、「Add」ノードを作成します。これらのステップに従うと、以下のグラフが完成します。



法線マップの追加

適切な法線マップを使用すると、以下の画像に表示されるエフェクトが生成されます。このケースでは、ダイヤモンドではなく 1 つのクアッドが使用されています。平面メッシュの屈折は、「Opaque Texture」オフセットとして表示されるだけです。平面メッシュを含む法線マップの使用は、このアーティファクトを隠す便利な方法です。

法線マップの使用

「Refract」ノードを使用する代わりとしては、「Vector3 viewDir」入力と「Vector3 normal」入力、および「IOR」出力を備えた「Custom Function」ノードを追加するということもできます。このオプションを使用する場合は、「IOR」プロパティをスライダー（範囲は 1 ～ 6 ではなく -0.15 ～ 2）として設定します。「Vector3」を出力として設定します。このコードは非常に単純なため、ファイルではなく文字列の使用だけで足ります。

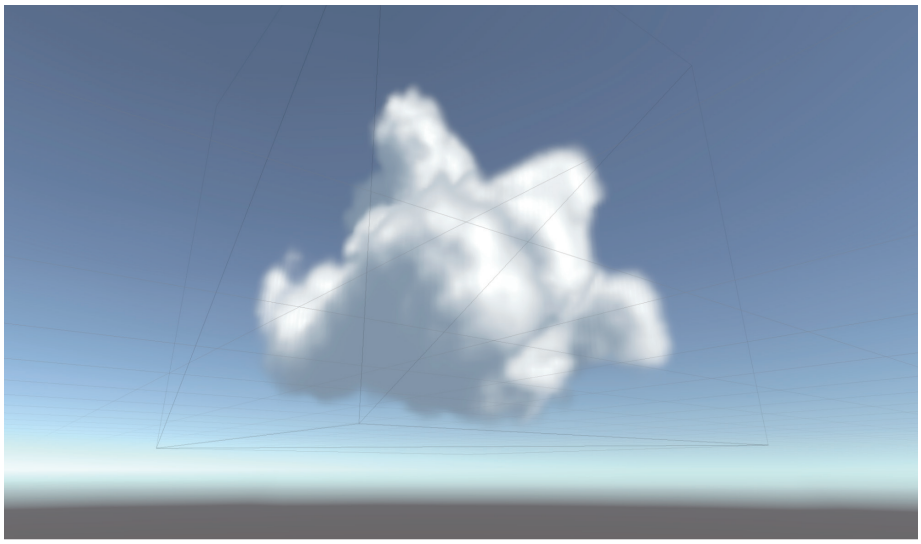
```
Out = refract(viewDir, normal, IOR);
```

これで結果に変化が出ます。試してみてください。

その他のリソース

- David Lettier の「Screen space refraction（スクリーンスペースリフラクション）」
- Steven Cannavan の「ScreenSpace planar reflection（ScreenSpace 平面リフレクション）」GitHub リポジトリ
- Kyle W. Powers の「Reflection probes vs Screen space reflection（リフレクションプローブとスクリーンスペースリフレクション比較）」
- AE Tuts の「Shader Graph refraction（シェーダーグラフの屈折）」チュートリアル
- Binary Lunar の「Crystal Shader Graph in Unity（Unity のクリスタルシェーダーグラフ）

ボリューメトリック

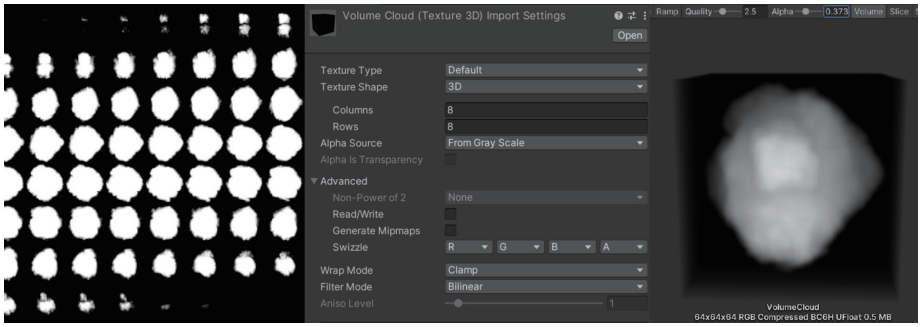


ボリューメトリッククラウド

これは、レイマーチングを使用して 3D テクスチャをレンダリングするレシピです。Unity でサポートされる 3D テクスチャは、1 つのテクスチャ上のグリッドに配置された画像の配列であり、テクスチャアトラスに似ています。違いは各画像のサイズが同じであることです。3D UV 値を使用して、使用する個々の画像の行と列を定義する UV.Z を含む画像のグリッドからテクセルを取得できます。

ゲーム『*Lost in Random*』（Zoink!）では、プレイヤーは独特なアートディレクションによる幻想的な王国に入り込みますが、そこでは素晴らしいライティングが雰囲気を作り出すために大きな役割を果たしています。この記事で説明されているように、ボリューメトリックフォグが URP によって再現されています。

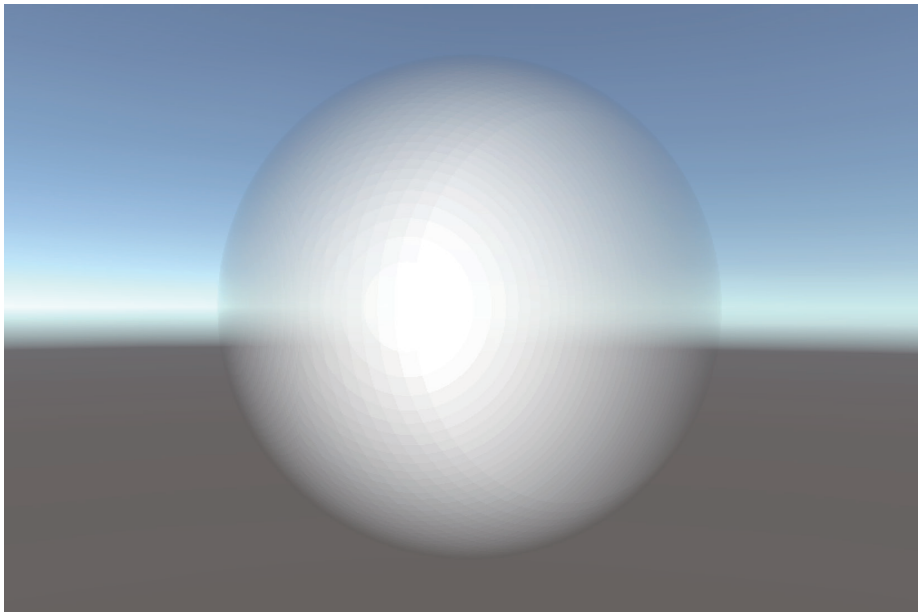
以下の画像に、典型的な 3D テクスチャ、そのインポート設定、Inspector でのプレビューを示します。



3D テクスチャ、そのインポート設定、Inspector でのプレビュー（左から右）

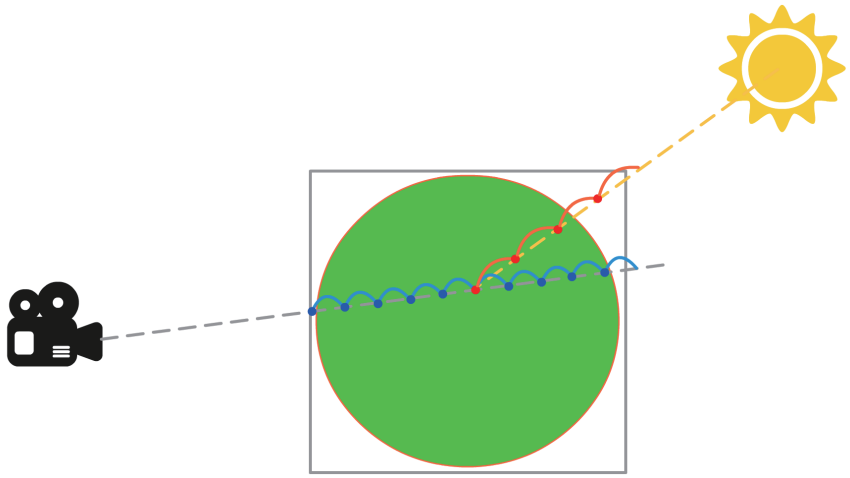
これまでのレシピと同じく、このシェーダーもシェーダーグラフで作成します。完成したものを確認するには、**「Scenes」>「Volumetric Clouds」** に移動して **VolumetricClouds** シーンを開きます。このシーンには、カメラ、ディレクショナルライト、キューブが含まれることに注意してください。キューブでは、マテリアル RaymarchMat が使用されます。

レシピを開始するには、RaymarchMat マテリアルに、Nik Lever によって作成された「**Shader Graphs/Raymarchv1SG**」という名前のシェーダーを加える必要があります。これによってスフィアが表示されます。**densityScale** を調整すると、端が透明になります。



シェーダーグラフ Raymarchv1SG の使用

レンダリングするのはキューブで、スフィアではないはずです。どうなっているのでしょうか。答えはレイマーチングです。レイマーチングは、Wikipedia ページでは次のように説明されています。「3D コンピューターグラフィックスのレンダリング手法の種類であり、レイが反復して走査する際に、各レイが小さなレイセグメントに効率よく分割され、各ステップでいくつかの関数がサンプリングされます。この関数は、ボリュームレイキャスティングのボリューメトリックデータや、サーフェスの交差を迅速に検出するための距離フィールドなどの情報をエンコードできます。」



レイマーチング

このバージョン 1 で、スフィアは Vector4 を使用して定義されます。XYZ によってスフィアの位置がオブジェクトと相対的に定義され、W によって半径が定義されます。ピクセルごとに、カメラから直接届くレイ（上図のグレーの点線）の方向が計算されます。密度値を 0 に設定してから、この線に沿って進むと、スフィア内の青い点ごとに計算が行われ、わずかな値が密度に加算されます。レイがスフィアを通過す

るときに、カメラからレンダリングするピクセルまでの直線上に、スフィアのどれくらいの部分が存在するかを表す値を取得します。この密度値は、シェーダーグラフで「Base Color」として使用されます。ここでは太陽と赤い点は無視してください。これらについては、ライティングを追加する時点（このシェーダーのバージョン 4）で検討することになります。

このグラフでは、**「Scripts」>「HLSL」>「Raymarch.hlsl」**のファイルに基づいて「Custom Function」ノードが使用されます。このバージョン 1 では、関数 raymarchv1 を使用します。変数 density が 0 に初期化されます。次に、numSteps 件数の for ループに入ります。rayOrigin は、rayDirection で定義された方向に、stepSize に応じて動きます。

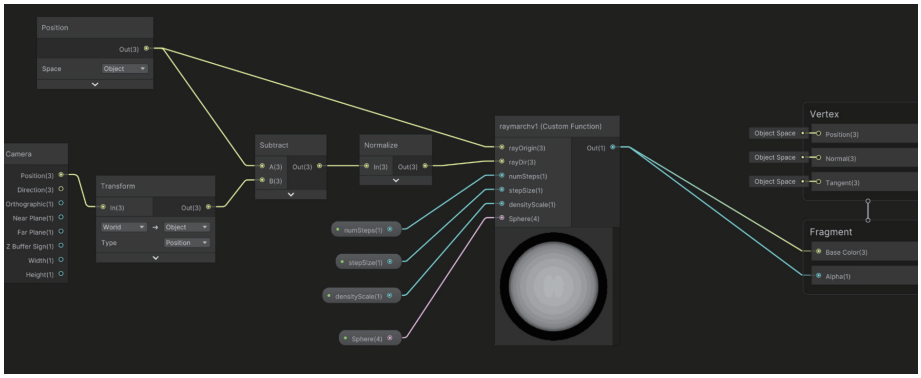
スフィアの原点からどれくらい離れているでしょうか。HLSL 関数の距離を使用して、スフィア原点から rayOrigin の現在の値までのベクトル長を計算できます。これがスフィアの半径（Sphere.w）よりも小さい場合は、density 値に 0.1 を追加します。出力値 result は、累積された density 値と densityScale を乗算したものです。

```
void raymarchv1_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                       float stepSize, float densityScale,
float4 Sphere,
                       out float result )
{
       float density = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              // 密度を計算
              float sphereDist = distance(rayOrigin, Sphere.
xyz);

              if(sphereDist < Sphere.w){
                     density += 0.1;
        }

       }

       result = density * densityScale;
}
```

計算についてはオブジェクト空間で作業します。rayOrigin は「**Position**」ノードを使用して取得します。rayDirection を取得するには、「position」出力が「**Transform**」ノードにリンクされ、入力が「World」、出力が「Object」に設定された「**Camera**」ノードが必要です。

これで、オブジェクト空間でのピクセル位置とカメラ位置を取得しました。これで、「Position」が入力 A、「Camera Position」が入力 B の「**Subtract**」ノードを使用してレイ方向を取得できます。この rayDirection は「**Normalize**」ノードを使用

して正規化されます。「Custom Function」ノードのその他の入力はフロートプロパティです。numSteps はレイごとの青い点の数、stepSize は青い点の間の距離です。densityScale と Vector4 スフィアは前に説明したとおりです。density 出力は「Base Color」と「Alpha」に直接つながっています。このシェーダーは透明かつ unlit と設定されているため、ライティングを計算する必要があることに注意してください。



バージョン 1 のレイマーチシェーダー

レイマーチングが実際に行われるのは、3D テクスチャが形状に追加されるときです。3D テクスチャはバージョン 2 で導入されます。最初に**「Shader Graphs」>「Raymarch2SG」**を使用するように RaymarchMat を設定します。使用されるカスタム関数は raymarchv2 です。

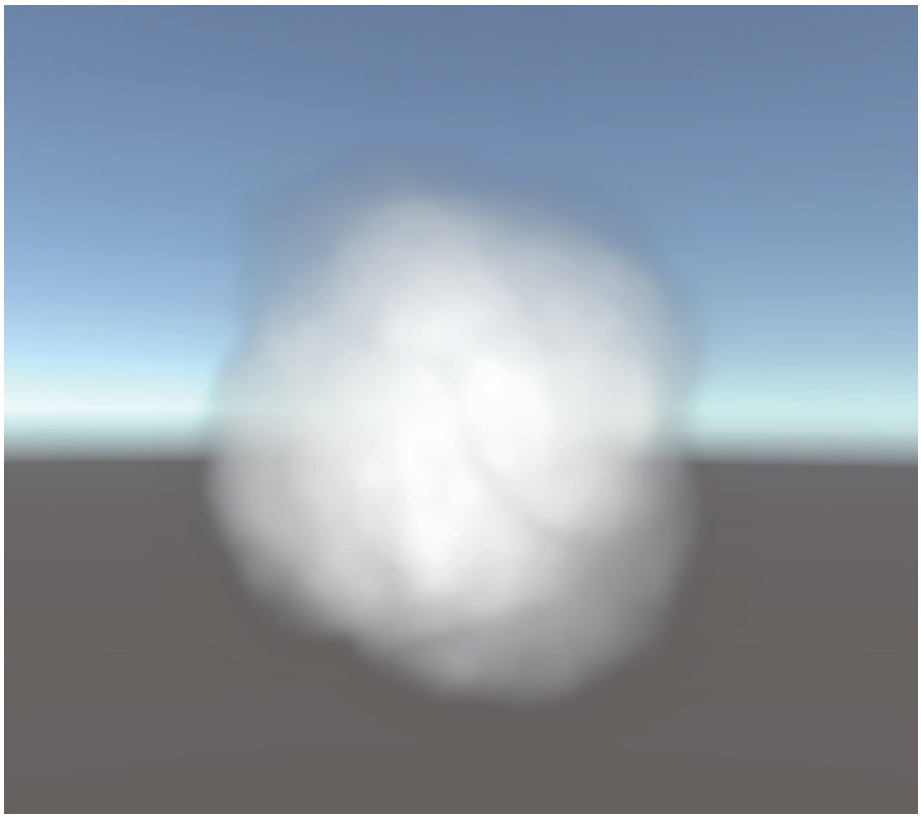
```
void raymarchv2_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                       float stepSize, float densityScale,
UnityTexture3D volumeTex,
                       UnitySamplerState volumeSampler, float3
offset,
                       out float result )
{
       float density = 0;
       float transmission = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              // 密度を計算
              float sampledDensity = SAMPLE_TEXTURE3D(volumeTex,
volumeSampler, rayOrigin + offset).r;
              density += sampledDensity;

       }

       result = density * densityScale;
}
```

3 つの新しい入力があります。

- 「**Material**」プロパティで直接生成される UnityTexture3D volumeTex。
- マクロ SAMPLE_TEXTURE3D。SamplerState インスタンスが必要な 3D テクスチャを処理するときに必要です。
    - SamplerState のノードがあり、それを使用して「Wrapping」オプションを選択できます。「clamp」に設定すると、範囲 0 ～ 1 に含まれない UV 値で、0 未満の値は 0、1 を超える値は 1 に固定されます。
- Offset。これは、キューブ内で 3D テクスチャを回転するために使用する値です。

次に、スフィア内にいるかどうかを調べるのではなく、rayOrigin の float3 サンプル位置にオフセットを加えたものを使用して **sampledDensity** 値を取得します。ここでは 1 つのチャンネル（赤いチャンネル R）のみが必要です。

以下の画像は、バージョン 2 をレンダリングしたものです。雲のように見えてきました。



バージョン 2 のシェーダー

最終バージョンのシェーダーにはライティングが導入されます。マテリアル RaymarchMat に対して Graphs/Raymarchv3SG という名前のシェーダーを使用します。このとき、関数 `raymarch` を使用します。この関数は、**numLightSteps**、**lightStepSize**、**lightDir**、**lightAbsorb**、**transmittance** という 6 個の新しいパラメーターを使用し、float3 ベクトルを返します。

最終値を得るために、3 つの新しい変数（**transmission**、**lightAccumulation** および **finalLight**）を初期化します。コードは、「ライトのループ」のコメントまではバージョン 2 と同じです。前に示した「レイマーチング」の図をもう一度見てください。ビュー方向のレイに沿ったステップ（青いドット）ごとに、メインライトに向けたレイ（図の黄色の点線）を取得します。赤い点は 3D テクスチャのステップごとのサンプリングを表します。雲が多いほど、ビュー方向レイの部分に当たる光は少なくなります。このプロセスによって各ピクセルの明るさが決まります。

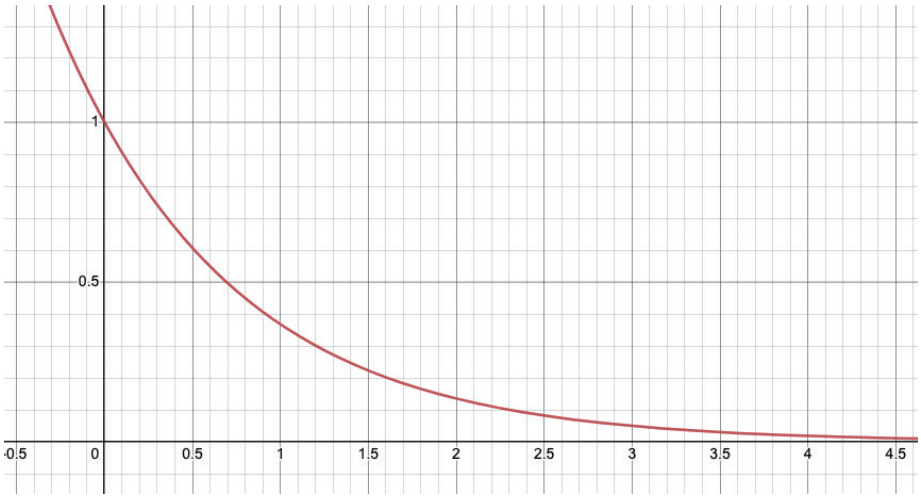
```
void raymarch_float( float3 rayOrigin, float3 rayDirection,
float numSteps,
                     float stepSize, float densityScale,
UnityTexture3D volumeTex,
                     UnitySamplerState volumeSampler, float3
offset,
                     float numLightSteps, float lightStepSize,
float3 lightDir,
                     float lightAbsorb, float darknessThreshold,
float transmittance,
                     out float3 result )
{
       float density = 0;
       float transmission = 0;
       float lightAccumulation = 0;
       float finalLight = 0;

       for(int i =0; i< numSteps; i++){
              rayOrigin += (rayDirection*stepSize);

              float3 samplePos = rayOrigin+offset;
              float sampledDensity =
                  SAMPLE_TEXTURE3D(volumeTex, volumeSampler,
samplePos).r;
              density += sampledDensity*densityScale;
              // ライトのループ
              float3 lightRayOrigin = samplePos;

              for(int j = 0; j < numLightSteps; j++){
                     lightRayOrigin += -lightDir*lightStepSize;
                     float lightDensity =
             SAMPLE_TEXTURE3D(volumeTex, volumeSampler,
lightRayOrigin).r;
                     lightAccumulation += lightDensity;
              }

             float lightTransmission = exp(-lightAccumulation);
              float shadow = darknessThreshold +
                                 lightTransmission * (1.0 -
darknessThreshold);
              finalLight += density*transmittance*shadow;
```

```
            transmittance *= exp(-density*lightAbsorb);

        }

        transmission = exp(-density);

        result = float3(finalLight, transmission, transmittance);
}
```

ライトのループは簡単に理解できます。**numLightSteps** 変数に指定した回数だけ繰り返されます。ネストされたループがあるため、numLightSteps 数をできるだけ低く抑えるようにしてください。lightDir を差し引いて samplePos からメインライトに移動します。次に、lightDensity が lightAccumulation に加算されます。ライトのループの外側でも同じ計算が必要です。

float lightTransmission = exp(-lightAccumulation);

最初に、lightTransmission が $e^{-lightAccumulation}$ と設定されます。定数 e（Euler 数）は約 2.718 です。以下のグラフにこの関数の結果を示します。水平軸は lightAccumulation の値、垂直軸は exp(-lightAccumulation) です。累積された光密度 lightAccumulation が 0 のとき、exp(-lightAccumulation) は 1 です。lightAccumulation が増えると、exp(-lightAccumulation) は急激に下がり、lightAccumulation が 5 以上になると 0 に近くなります。

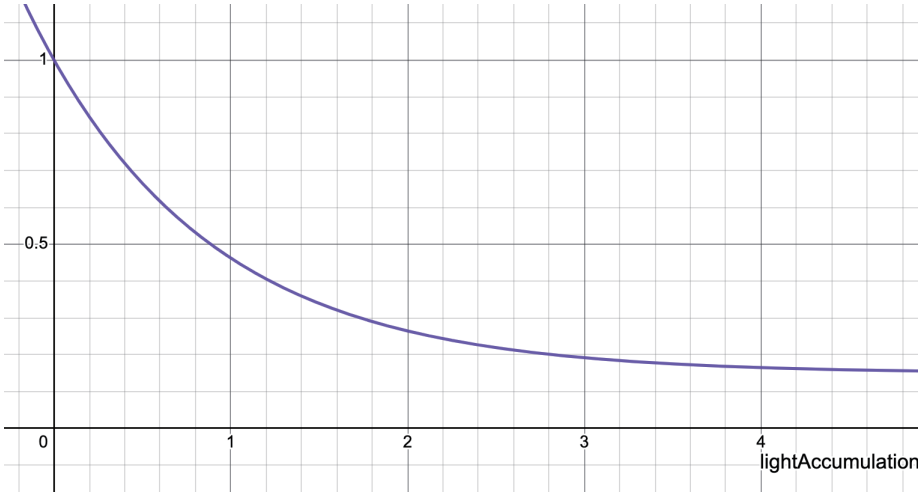


e^x（範囲 0 ～ 4）のグラフ

float shadow = darknessThreshold +
lightTransmission * (1.0 - darknessThreshold);

shadow 値が次に計算されます。**darknessThreshold** という名前のプロパティを使用します。以下のグラフでは、darknessThreshold が 0.15 の場合の shadow

値を垂直軸に示します。lightAccumulation が 0 の場合 shadow は 1 ですが、lightAccumulation が 5 に近づくと、shadow は darknessThreshold 定数値に近づきます。



shadow 値

finalLight += density*transmittance*shadow;

density * transmittance * shadow が finalLight 累積値に加算されます。累積された光密度 lightAccumulation が高いと、shadow は 0 に近づくため、finalLight の累積値は小さくなります。

transmittance *= exp(-density*lightAbsorb);

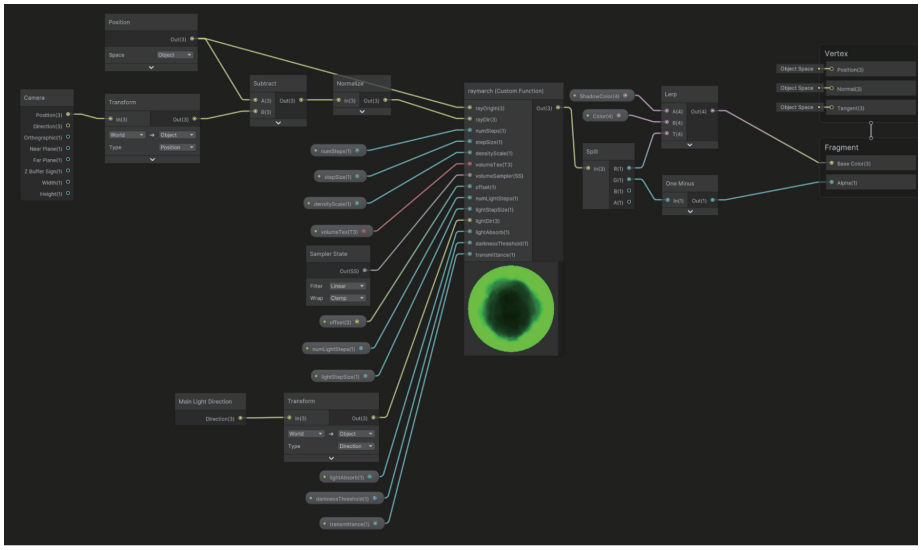
transmittance の初期値は渡されるプロパティです。ただし、ビュー方向ステップごとに、値に $e^{-density*lightAbsorb}$ が乗算されます。プロパティ **lightAbsorb** により、雲の中で光が拡散してどれくらい失われるかが制御されます。

バージョン 3 では、結果は finalLight、transmission、および transmittance を含む float3 です。

バージョン 3 のグラフを以下に示します。「Custom Function」の出力が float3 となったところで、「Split」ノードが追加されます。出力 R は「Lerp」ノードの T 入力につながります。バージョン 3 には、「**color**」や「**shadowColor**」など、いくつか新しいプロパティがあります。前者は B 入力、後者は A 入力です。

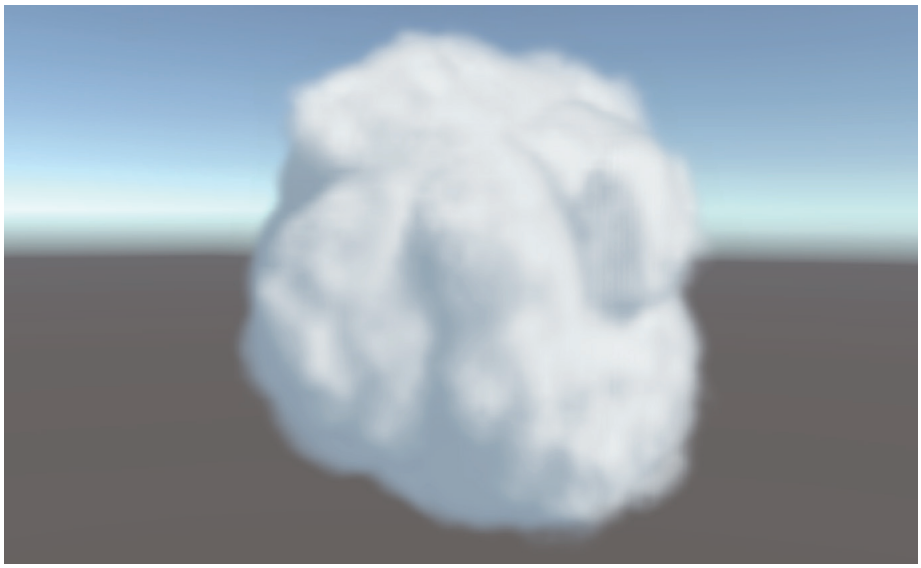
「finalLight raymarch」ノードの Out.x が 0 の場合、shadowColor が「Lerp」ノードの出力に渡されます。finalLight が 1 の場合は、color が出力に渡されます。範囲が 0 ～ 1 の場合、shadowColor と color の線形補間が出力です。「Lerp」ノードの出力を「Fragment」>「Base Color」に直接つなげます。

「Alpha」は、transmission 値が Out.y の「raymarch」ノードを使用します。この値が 0 のとき「Alpha」は 1、1 のとき「Alpha」は 0 になります。「One Minus」ノードは、「Split」ノードの B 値を修正するために使用されます。これを「Fragment」>「Alpha」にリンクします。



最終バージョン

これによって結果は以下のようになります。



レイマーチングを使用した雲

Houdini は 3D テクスチャを作成するときに便利なツールです。3D テクスチャの代替としては、多層化されたパーリンノイズの使用、またはタイル状にできるノイズテクスチャの事前ベイク（Unity 使用）が含まれます。このレシピがレイマーチングの使用を始めるスタート地点となれば幸いです。

その他のリソース

- dmeville の「Volumetric ray marching cloud shader（ボリューメトリックレイマーチングクラウドシェーダー）」
- Sebastian Lague の「Coding adventure: Clouds（コーディングアドベンチャー：雲）」
- Camelia Slimani の「Creating Volumetric Clouds with Houdini（Houdini を使用した Volumetric Clouds の作成）」
- OccaSoftware の「Altos sky system（Altos スカイシステム）」

まとめ

Unity から上級者向けの多くのリソースが提供されています。このガイドの冒頭で説明したように、e ブック『*Introduction to the Universal Render Pipeline for advanced Unity creators（上級 Unity クリエイター向け URP 入門）*』は、経験を積んだ Unity の開発者やテクニカルアーティストが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 に移行する際に役立つ貴重なガイドです。この e ブックで取り上げるトピックは以下のとおりです。

- 新規プロジェクト用に URP を設定するか、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
- 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
- 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

すべての上級者向け技術 e ブックおよび記事は、Unity ベストプラクティスハブにあります。e ブックは、上級者向けベストプラクティスのドキュメントページにもあります。

Unity クリエイターのためのプロフェッショナルトレーニング

Unity プロフェッショナルトレーニングにより、Unity でより高い生産性を発揮し、効率的にコラボレーションを行うためのスキルと知識が得られます。業界を問わず、あらゆるスキルレベルのプロフェッショナル向けに設計された、様々な手法で教えてくれる広範なトレーニングが一覧できます。

全教材が Unity の経験を積んだインストラクショナルデザイナーのパートナーであるエンジニアや製品チーム協力の下で制作されています。これは、常に Unity の最新テクノロジーに関する最新のトレーニングを受けることができることを意味します。

こちらで Unity プロフェッショナルトレーニングが、チームや皆さんへどれほど貢献できるのか、その詳細をご覧ください。

unity.com