→ EBOOK

ユニバーサルレンダーパイプライン (URP)
クックブック:

シェーダーと視覚効果 (VFX) のレシピ

Unity®

Contents

はじめに

ポストプロセスエフェクトをひとさじ、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、そして発泡水を少々...。さあ、ユニバーサルレンダーパイプライン（URP）を使用して、高品質のライティングと視覚効果（VFX）を "調理" し、ゲームに盛り込みましょう。

この URP クックブックでは、人気の効果を実現するための多数のレシピを 12 章にわたって紹介しています。また、これらのレシピに基づくサンプルシーンは、筆頭著者である Nik Lever 氏が管理する 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、Unity でプロジェクトを開発した経験があり、URP 機能の使用方法と HLSL ベースのシェーダーの作成に関する知識を持つ、中級レベルの Unity ユーザーを対象としています。

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。

— Shader Graph 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加える。

— 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加える。

— 反射と屈折を生成する、など。

e-book および各レシピのダウンロード可能なサンプルシーンは、Unity 6 に対応するように更新され、以下の新要素がこのガイドに加わりました。

— プロシージャルノイズとコンピュートシェーダーに関する 2 つの新レシピ

— トゥーンシェーダーの完全な改訂版

— ライトプローブに代わる、迅速かつ柔軟なアダプティブプローブボリューム (APV) の実装方法に関するセクション

— 新しい Render Graph API を使用して Renderer Feature を作成するステップ

このクックブックと併せて、e-book『*上級 Unity クリエイター向けのユニバーサルレンダーパイプライン (URP) 入門*』を参照してください。Unity の YouTube チャンネルには、URP チュートリアルの再生リストもあり、ゲームのライティングや効果の作成に役立つ一般的なガイドと専門的なヒントが紹介されています。

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。

この e-book のレシピの多くは、HLSL (High-Level Shader Language) を使用しています。この言語に不慣れな場合は、以下のリソースを参考にしてください。

— Unity マニュアルにある カスタム シェーダー の 例

— Ronja 氏の HLSL チュートリアル

— Udemy: Learn Unity Shaders from Scratch

— The Book of Shaders

この画像は『*PRINCIPLES*』からのもので、経験豊富な開発者の手にかかれば URP でここまで実現できるということを示すサンプルです。『*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. の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、現在 App Store または Google Play で入手できます。また、こちら でスタジオへ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-book 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 コンテンツを制作しており、2006 年から Unity を使用しています。30 年以上にわたり少数精鋭の開発会社 Catalyst Pictures を率いていることに加え、急速に進化するゲーム業界において開発者の知識を広げることを目的に、2018 年からコースを提供しています。

Unity の貢献者

Steven Cannavan は Unity のシニアソフトウェア開発コンサルタントであり、グラフィックスとレンダリング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。

MingWai Chan は Unity のグラフィックスエンジニアリングチームのシニアテクニカルアーティストです。Unity で 8 年間勤務しており、2012 年から Unity エディターを使用しています。

Oliver Schnabel は Unity のグラフィックスチームのシニアテクニカルプロダクトマネージャーです。顧客からのフィードバックを取り入れながらグローバルスタジオと協力して、高性能で統一されたスケーラブルなレンダリングスタックの開発に尽力しています。コンピューターグラフィックスとリアルタイム開発における豊富な経験を持ちます。

Jonas Mortensen は Unity のグラフィックスチームのシニアテクニカルアーティストです。

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

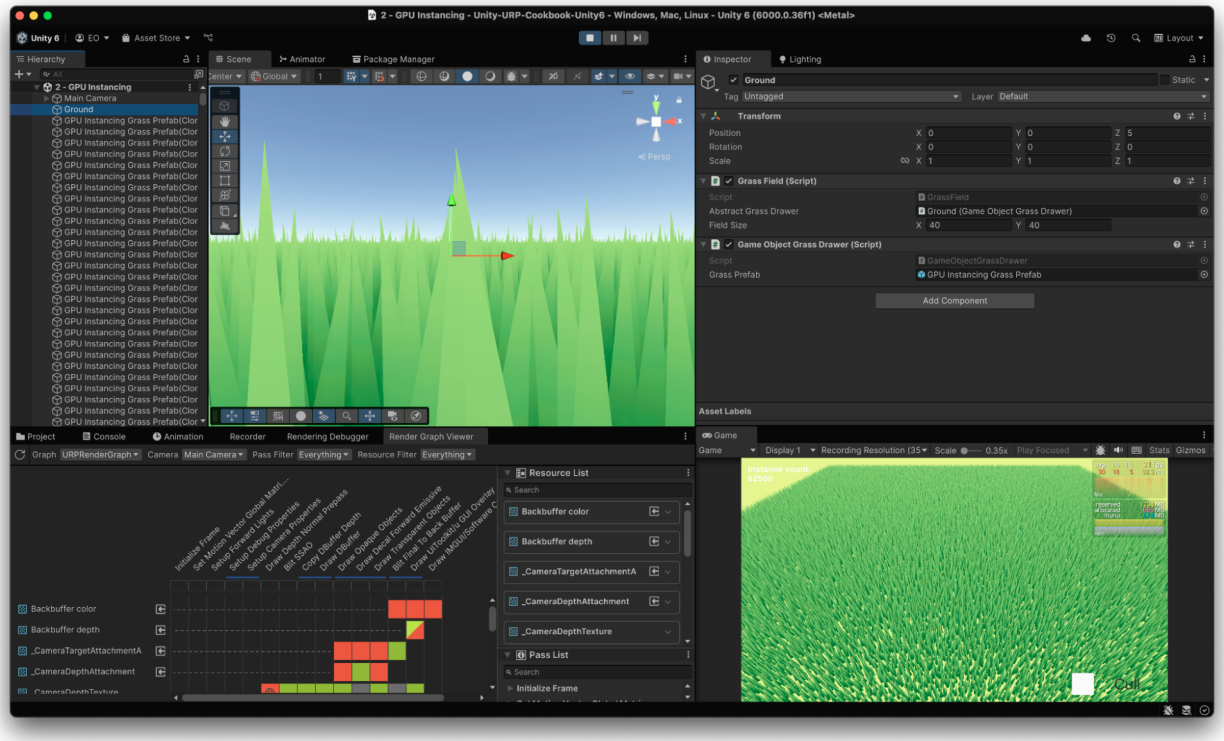
Mathieu Muller は Unity のグラフィックス担当リードプロダクトマネージャーです。グラフィックス製品管理チームを率い、グラフィックスのロードマップと製品ビジョンを監督しています。

Damian Nachman は Unity のグラフィックスチームのシニアテクニカルプロダクトマネージャーで、低レベルグラフィックスの開発と最適化を専門としています。リアルタイムグラフィックスエンジンの開発や、複数業界にわたるベンチマーキングの経験が 10 年あります。

このガイドの始め方

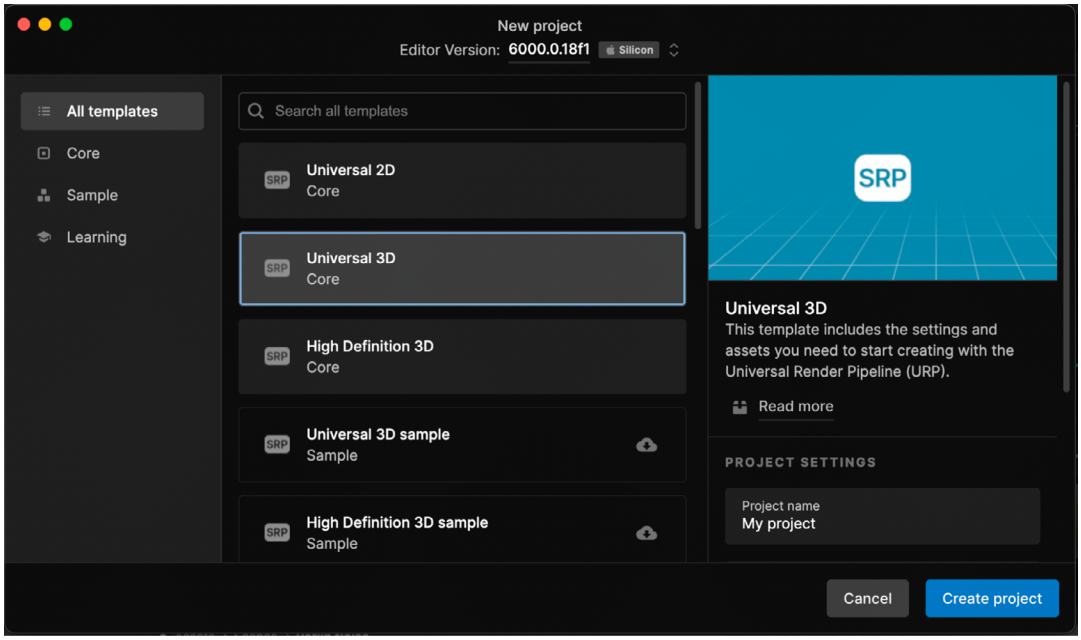
各レシピの手順に沿って進行すれば、新規に URP プロジェクトを立ち上げた際にレシピ通りのライティングエフェクトや視覚エフェクトを再現することができるようになっています。また、各レシピのダウンロード可能なサンプルシーンが提供されている、このガイドに即した GitHub ページ にもアクセスできます。

レシピはすべて、Unity 6 で動作するように大幅に更新されています。



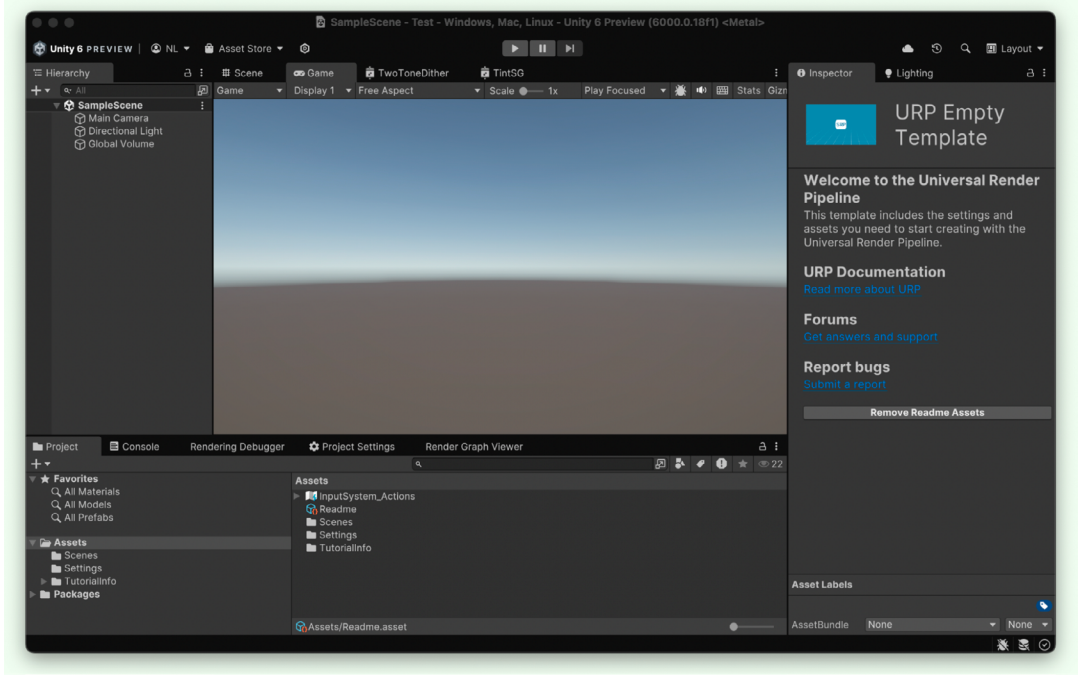
新規 URP プロジェクトを開始する

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 6000.01 以降であることを確認します。プロジェクトの名前と保存場所を指定し、**3D (URP)** テンプレートを選択して **Create** をクリックします。



Universal 3D テンプレートを使用して新規プロジェクトを作成する際、初回にテンプレートのダウンロードが必要になることがあります。

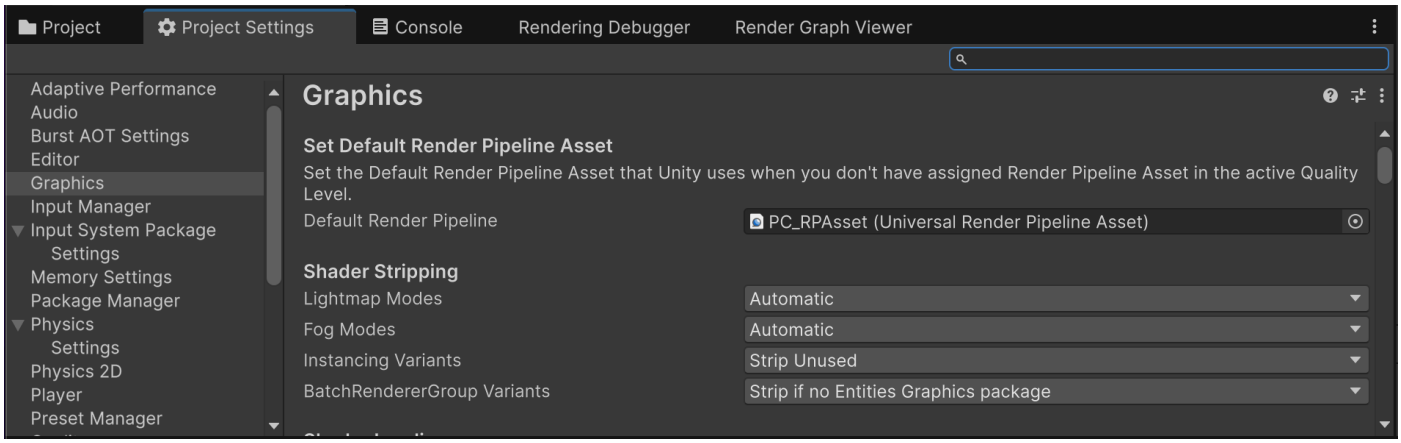
注: テンプレートを使用すると、プロジェクトはライティングを正しく計算するために必要なリニア色空間を使用するように設定されます。



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 の順に移動し、**Graphics** パネルを開くと、**Default Render Pipeline Asset** が表示されます。この URP アセット は、プロジェクトのグローバルなレンダリング設定と品質設定を制御し、レンダーパイプラインのインスタンスを作成します。一方、レンダリングパイプラインのインスタンスには、中間リソースとレンダーパイプラインの実装が含まれています。

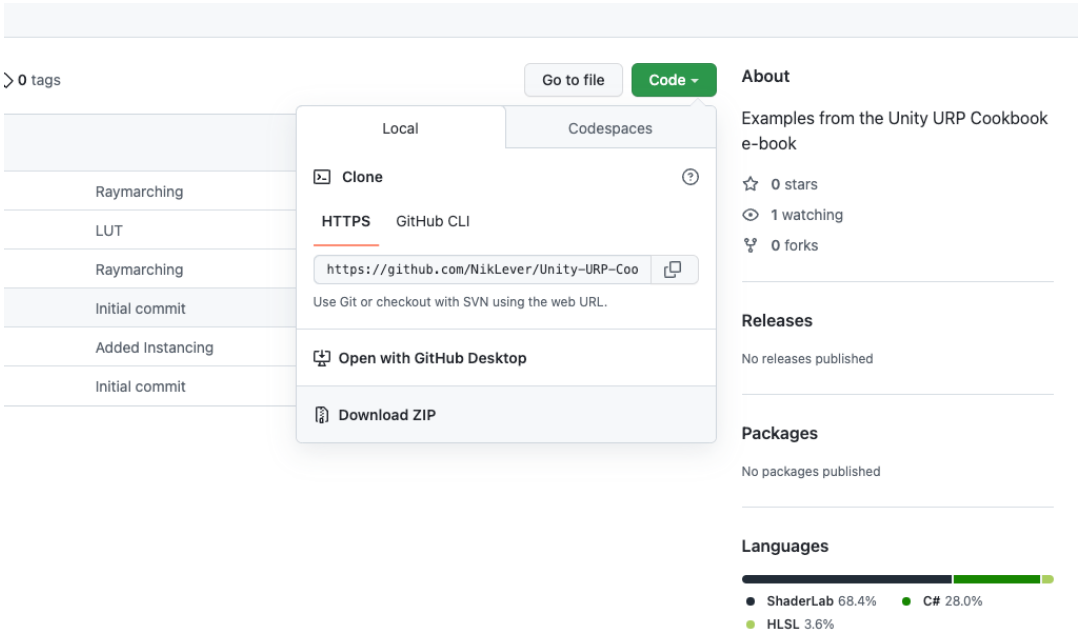
デフォルトで選択されている URP アセットは **PC_RPAsset** ですが、リソースがより限られているデバイスに適したアセットとして **Mobile_RPAsset** に切り替えることも可能です。



Project Settings の Graphics パネル

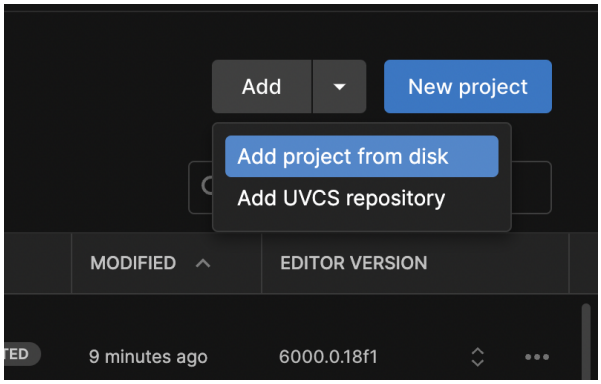
e-book のサンプルシーンをインポートする

こちら からリポジトリのクローンを作成するか、コードを zip ファイルでダウンロードして解凍できます。



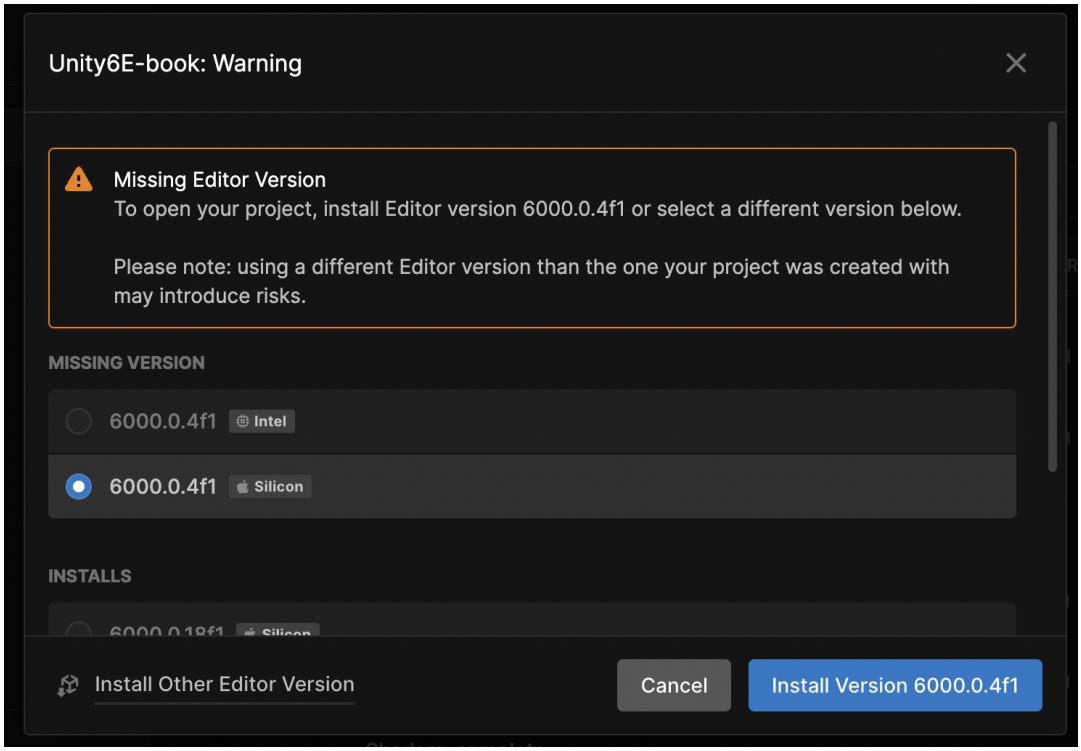
この GitHub リポジトリでは、緑色の Code ボタンをクリックするとプロジェクトをダウンロードできます。

プロジェクトを解凍およびダウンロードしたら、Unity Hub から **Open > Add project from disk** の順に選択してインポートします。



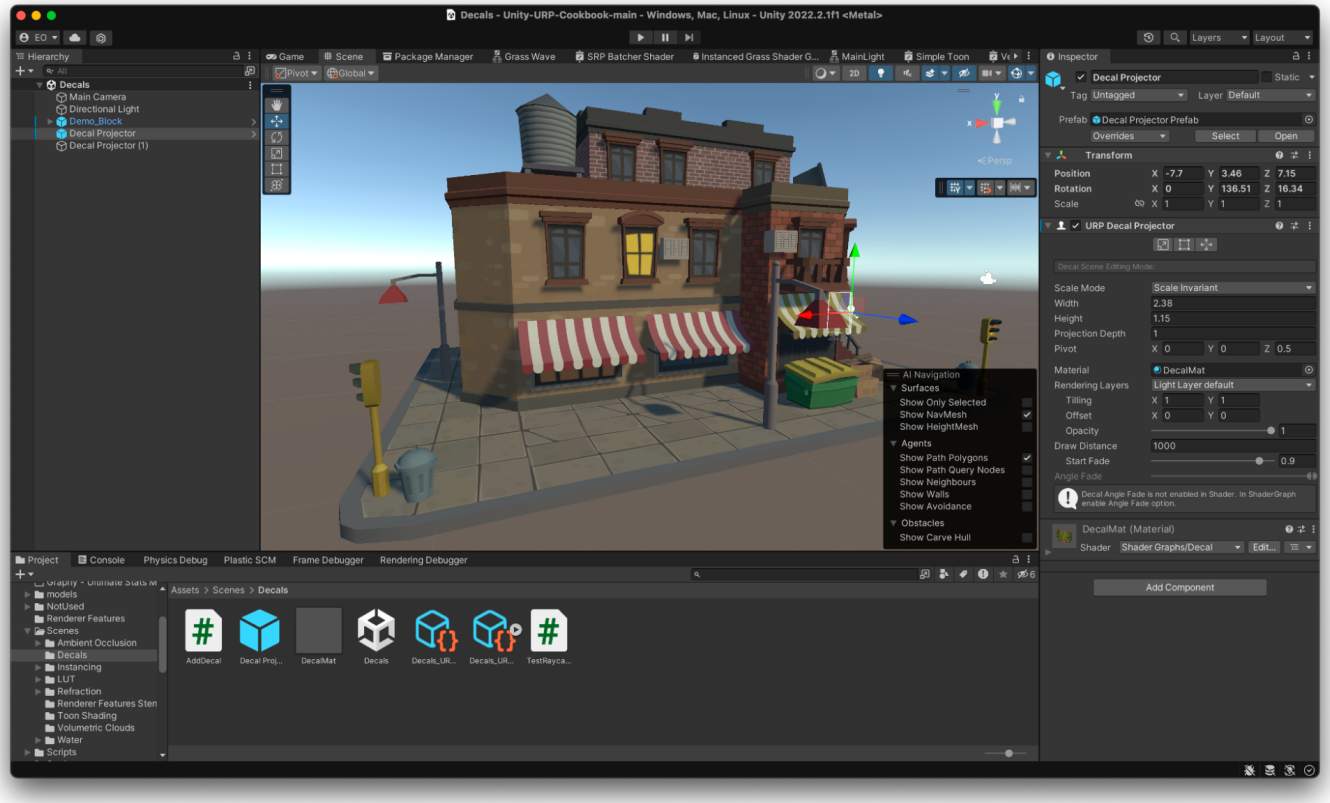
Unity Hub からサンプルプロジェクトをインポートします。

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。



使用しているチュートリアルプロジェクト、または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します。これは Unity Hub から簡単に行えます。

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。

ステンシル

Made with Unity ゲーム『*TUNIC* 』(制作: Andrew Shouldice 氏、TUNIC Team、22nd Century Toys LLC、Isometricorp Games Ltd.、発行: Finji) では、舞台道具によってメインキャラクターが隠れたときには、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。こちらのビデオ チュートリアル でも説明されています。

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 Asset と URP Asset) があります。Universal Renderer Asset からは、レンダーパイプラインの以下の任意の段階に Renderer Feature を挿入できます。

- 影のレンダリング中
- 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

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。

作業を進めるには、エディターで **Scenes > Renderer Features Stencils > SmallRoom - Stencil** の順に選択してサンプルシーンを開きます。

ステンシルの動作:拡大鏡をデスクの上へ移動させると、ドロワーの中身が透けて見えます。

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の中身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。最初のステップでは、レンズに使用されているマテリアル (この例では **MaskMat** という名前) を、**Custom/StencilMask** というシェーダーで変更します。

```
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
```

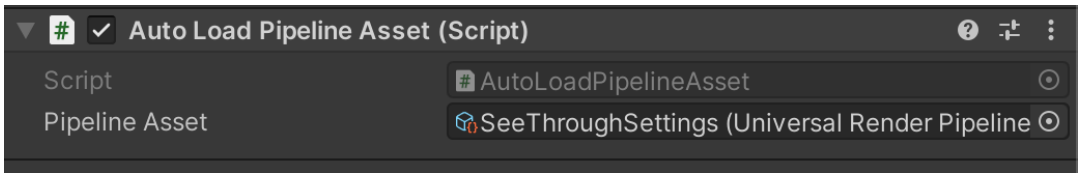
```
            {
                return (half4)0;
            }

                    ENDHLSL
             }
        }
}
```

Custom/StencilMask では、コマンド `ZWrite Off` が設定されていることに注意してください。ほとんどの場合、オブジェクトに `ZWrite Off` を設定すると、そのオブジェクトは表示されなくなります。これは、オブジェクトが深度バッファに深度値を書き込まなくなり、背後のオブジェクトを遮らなくなるためです。オブジェクトは引き続き同じ順序でレンダリングされますが、描画されたピクセルコンテンツは背後のオブジェクトによってオーバーライドされます。つまり、`ZWrite Off` を設定してもレンダリング順序は変わりません (ただし、レンダーキューのインデックスを設定すると変わります)。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。考慮する必要があるのはステンシルバッファへの書き込みのみで、シェーダーのカラーバッファへの出力については不要です。そのため、`ColorMask 0` を指定してカラーバッファへの書き込みを無効にできます。これはやや最適化されたアプローチで、特にシーンがレンズマスクより先にレンダリングされる ディファード レンダリング パス で使用する場合に有効です。

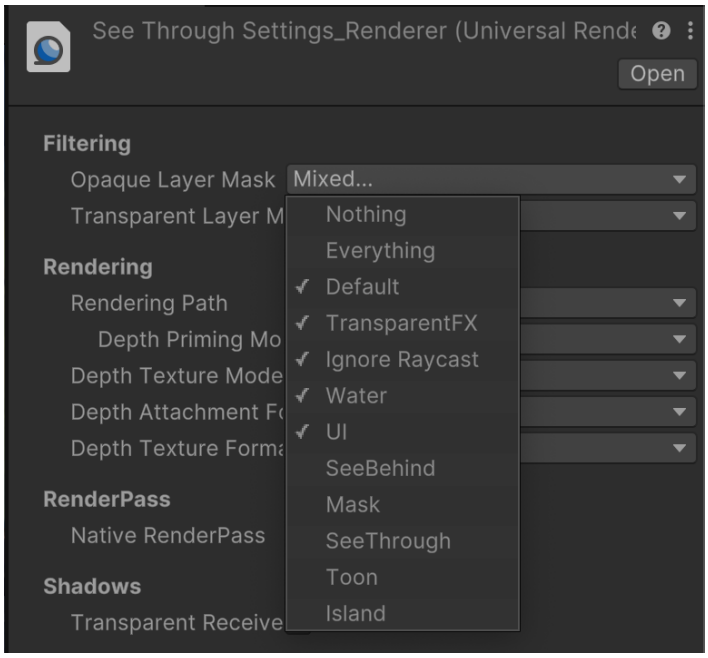
この例では、**Mask** と **SeeThrough** という 2 つのカスタムレイヤーが使用されています。レンズは Mask レイヤーにあり、デスク (ただしその子は除く) は SeeThrough レイヤーにあります。

このシーンでは、**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。これは、シーンファイル、マテリアル、シェーダーと同じフォルダー内 (**Scenes > Renderer Feature Stencils**) にあります。メインカメラにアタッチされたスクリプト AutoLoadPipelineAsset により、このオブジェクトが **Project Settings > Graphics** でスクリプタブルレンダーパイプラインアセットとして設定されるようになります。ここで、このアセットの設定を確認しましょう。



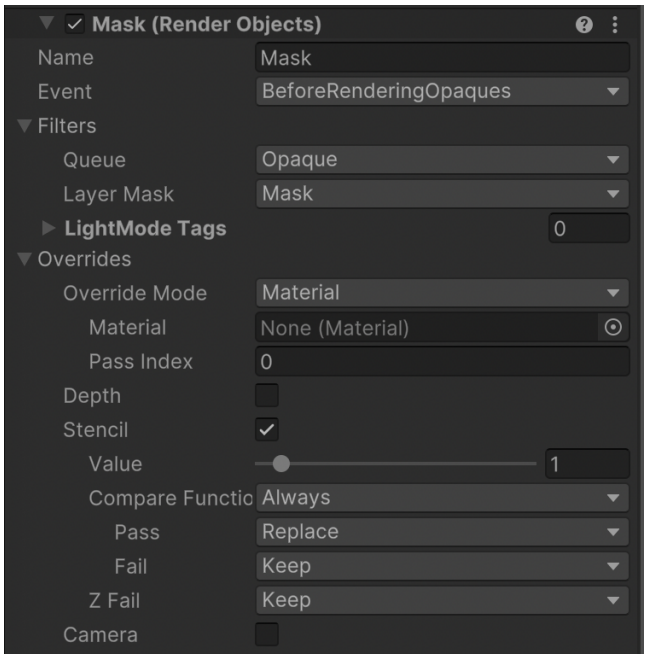
Main Camera > Auto Load Pipeline Asset script に設定されたパイプラインアセット

Scenes > Renderer Feature Stencils から **SeeThrough Settings_Renderer** を選択します。デフォルトからの変更がある最初の設定は **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 を変更します。

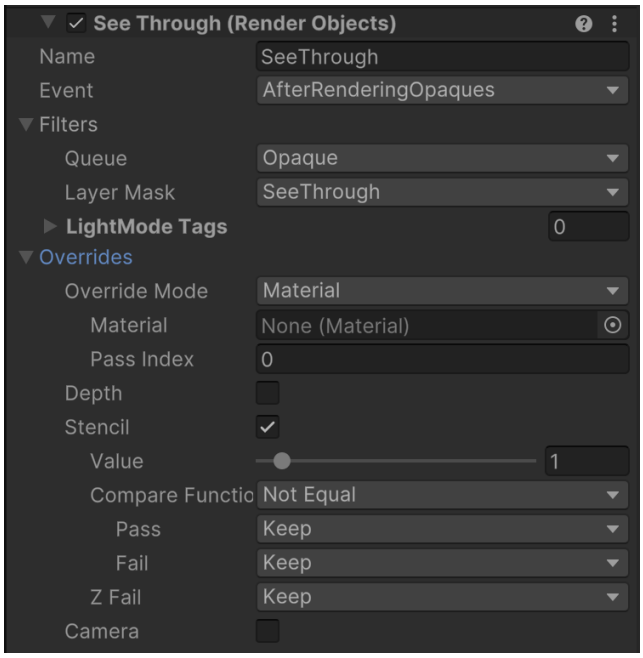
Inspector の Renderer Feature のリストには、**Mask** と **SeeThrough** という名前の 2 つの Render Objects 機能があります。SeeThrough オプションを無効にすると、デスクが消えてしまいます。この状況が発生するのはなぜかと言うと、Opaque Layer Mask が使用されているのはあくまでフィルターで除外されたレイヤーの一部であって、デフォルトレンダリングの一部ではないからです (Render Objects 機能を元にレンダリングされているだけです)。



Mask (Render Objects) の設定

上の画像は、Mask がイベント **BeforeRenderingOpaques** を使用する設定になっており、Mask レイヤー上にレンダリングされたピクセルに対してのみ適用されるようフィルター処理されています。Overrides パネルでは、**Stencil** オプションが有効になっており、バッファには値 1 が保存されます。この書き込みが確実に行われるように、**Compare Function** は **Always**、**Pass** は **Replace** に設定され、既存の値を常に置き換えるようにしています。**Fail** および **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、ビジョン内でレンズが表示されているすべてのピクセルで、ステンシルバッファの値が 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、ステンシルバッファが設定された後に適用されます。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 は **Not Equal**で、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、このパスはレイヤー See Through のピクセルのうち、ステンシルバッファに値 1 が含まれないピクセルのみをレンダリングするようになります。つまり、レンズがある部分のみでデフォルトのレンダリングが残ります。Compare Function を **Equal** に変更してみましょう。そうすると、逆の結果になり、デスクがレンズ内にのみ表示されるようになります。

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

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。

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。

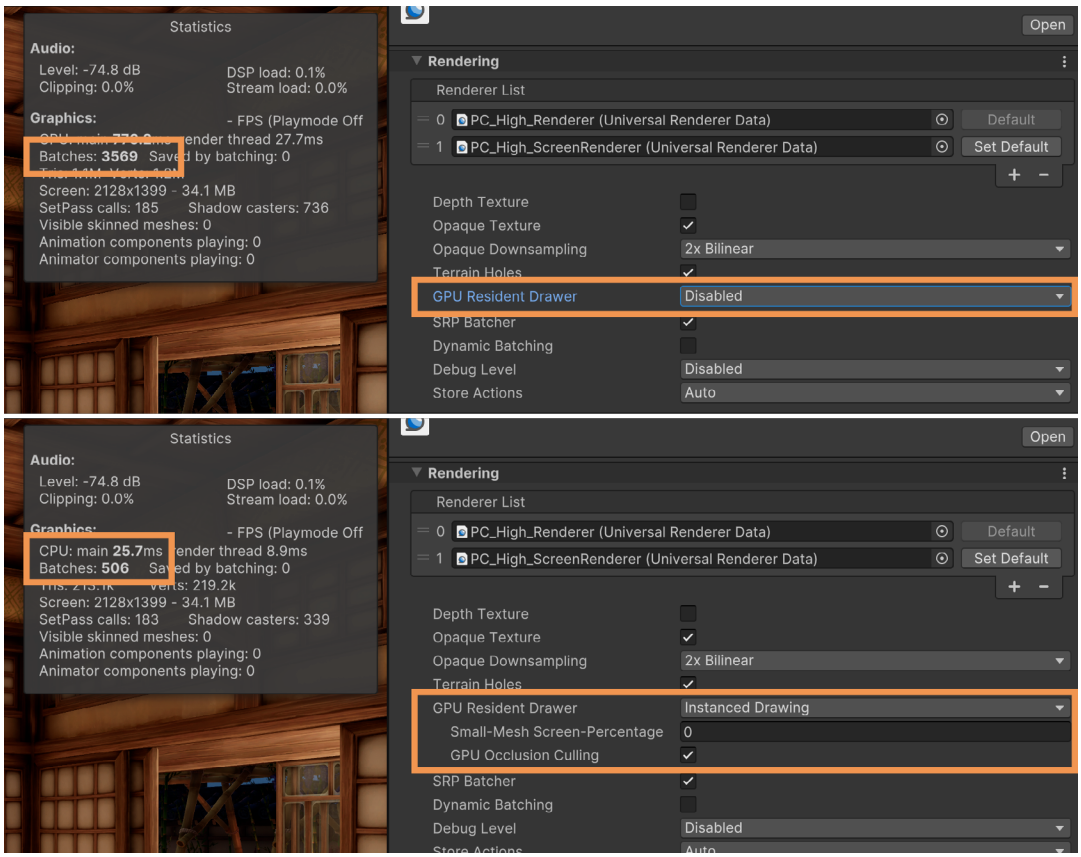
GPU Resident Drawer と GPU オクルージョンカリング

インスタンシングのレシピに進む前に、Unity 6 の新しい汎用ソリューション "GPU Resident Drawer" を見てみましょう。これは、URP アセットのレンダリング セクションで入手できます。

GPU Resident Drawer は、CPU 処理時間を最適化するために設計された GPU 駆動のレンダリングシステムです。これにより、ゲームオブジェクトが

BatchRenderGroup API を活用でき、高速なバッチ処理と CPU パフォーマンスの向上が期待できます。

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



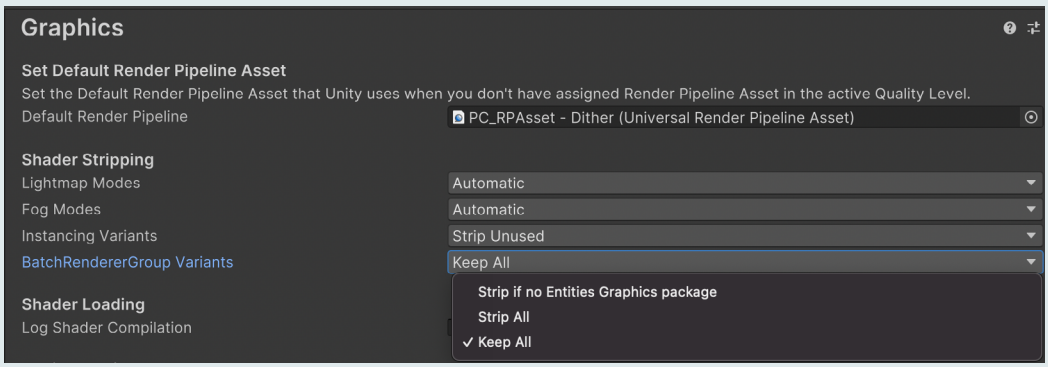
Unity 6 の URP アセットで利用できる GPU Resident Drawer と GPU オクルージョンカリングのオプション

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

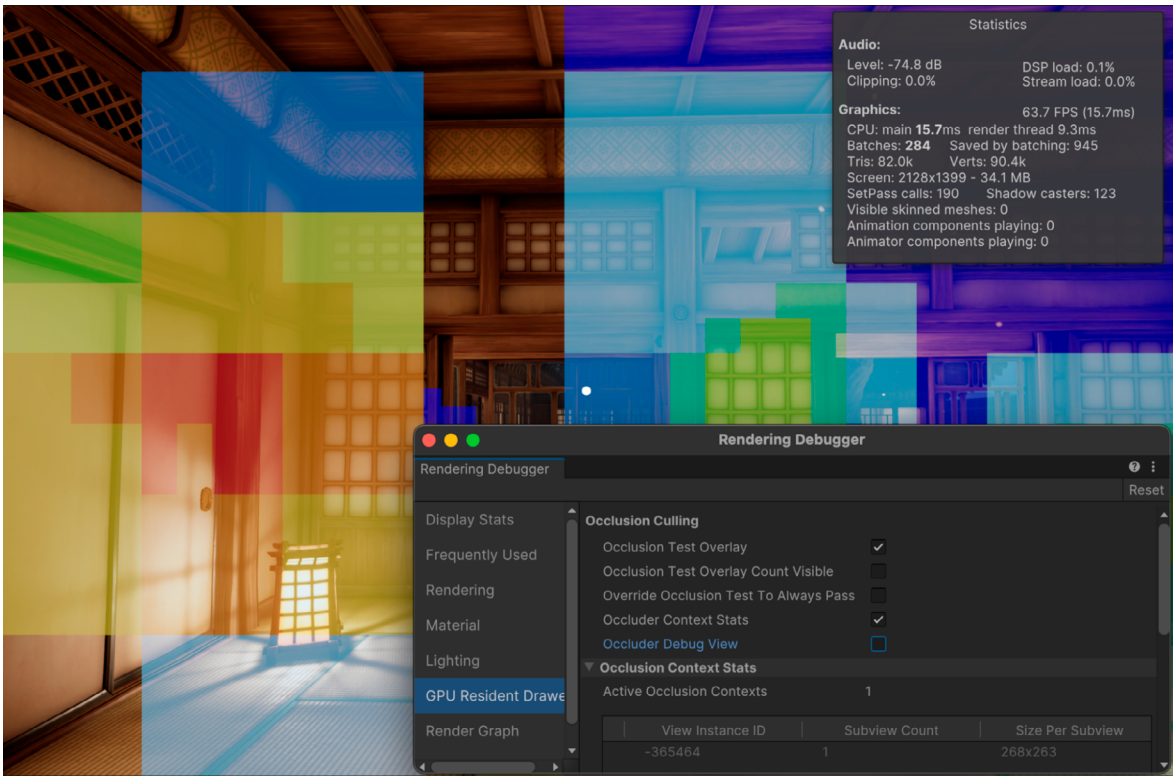
どの程度の改善が見られるかは、シーンの規模やインスタンシングの量によって異なります。レンダリングするインスタンス化可能なオブジェクトの数が多いほど、メリットが大きくなります。

GPU Resident Drawer は MeshRenderer を対象としており、スキンメッシュレンダラー、VFX Graph、パーティクルシステム、または同様のエフェクトレンダラーには対応していません。この機能を利用するために、既存のコンテンツを変更する必要はありません。ただし、カスタムシェーダーを使用している場合は、DOTS インスタンシングと互換性があることを確認する必要があります。例として、この 簡略化されたバージョン を参照してください。

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



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



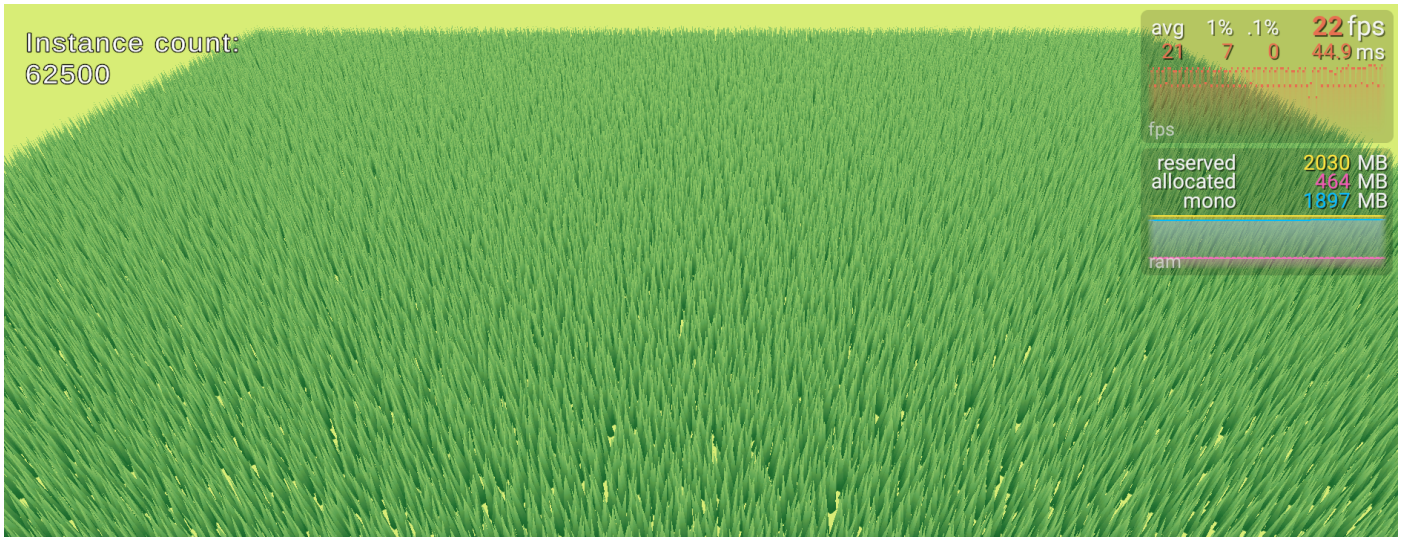
レンダリングデバッガーを使ったオクルージョンテストの確認

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

インスタンシング

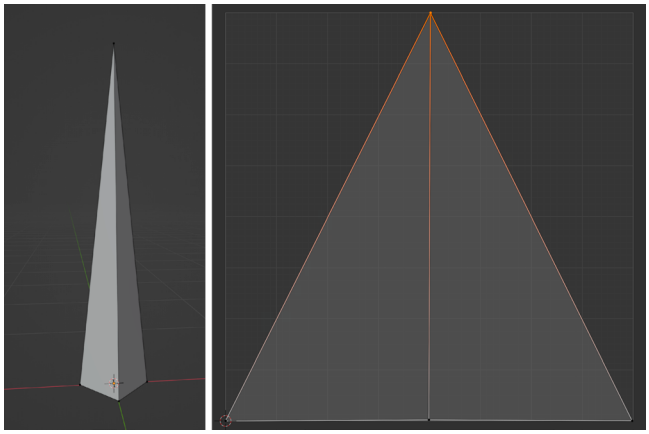
一面に広がる草原を例に、インスタンシング の概念を説明します。これはフォトリアリスティックとはかけ離れていますが、関連する手法の形で描写するには十分なものです。このサンプルはフォルダー（**Scenes > Instancing**）にあります。

注：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

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**、先端の V 値が **1** になるように 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。

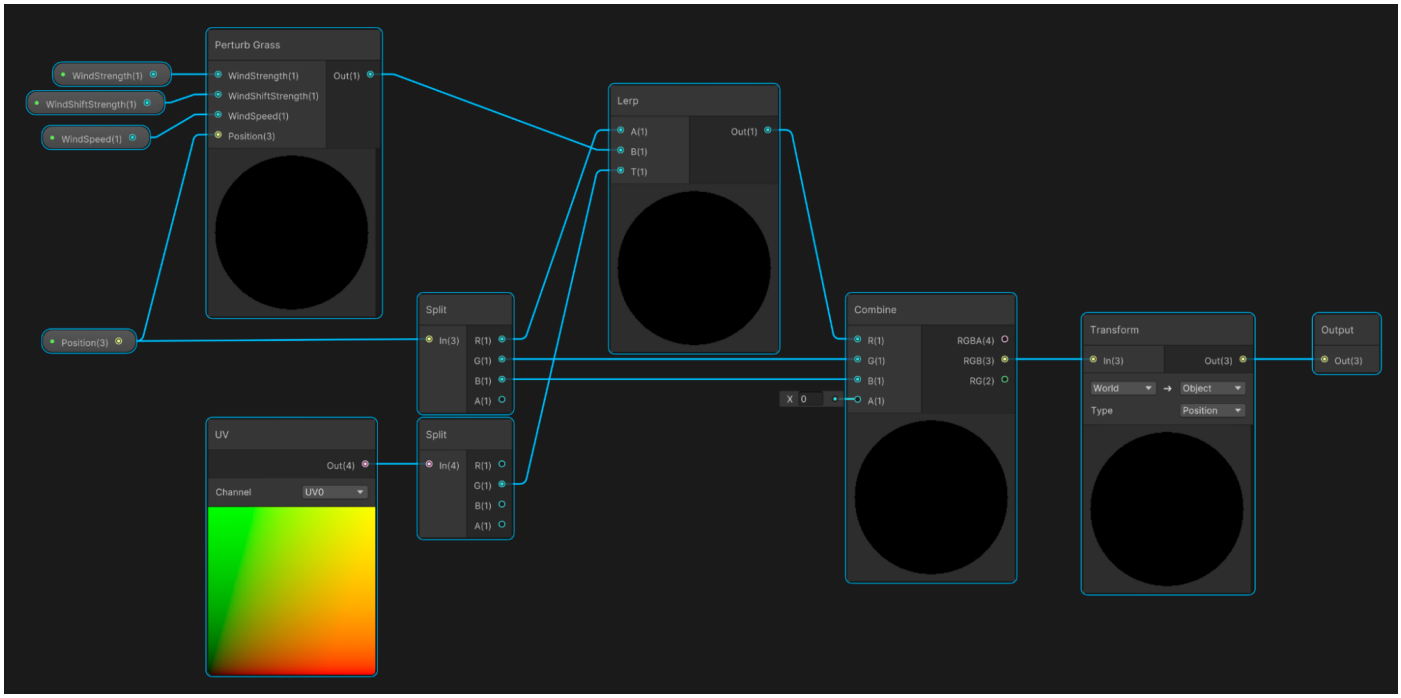


草の葉モデルと UV

SRP Batcher

フォルダー (**Scenes > Instancing > Common > Grass Wave)** にある Shader Graph のサブグラフを見てみましょう。このサブグラフの目的は、WindSpeed、WindShiftStrength、WindStrength に基づいてオブジェクトの頂点 X の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きになるように、**Perturb Grass** というサブグラフで Noise ノードが使用されています。頂点 Y および Z の位置は出力にそのまま渡されますが、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 サンプルでは 250 に設定されています。
- `cellSize` は (0.16, 0.16) です。
- 2 つのループが反復して、`_grassEntities` 2D 配列の各要素を順に設定します。
- それぞれの葉の根元の位置は、`_startPosition` に現在のセルを加算した値で、そこにランダム係数が使用されます。
- `_abstractGrassDrawer` は、草を配置するコードの 2 つのバージョンに使用される基本クラスです。
  - 初期バージョンでは、GPU インスタンシングを使用せず、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できるかを確認します。
  - まず、このシーンでは、草の葉モデルのプレハブを `grassEntities` 2D 配列の各位置に配置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);
		}
	}
}
```

このコードでは、Instantiate を使用して `grassEntities` 配列を走査し、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。29 ページ の草原の画像からわかるように、62,500 枚の葉があるこのシーンを以下のような仕様の 2022 年製 MacBook Air で実行した場合、フレームレートは 29 fps という低速になります。

— ビルトイン Retina ディスプレイ

— プロセッサー:Apple M2 2022

— メモリ:8 GB

シーンはどのようにして最適化できるでしょうか？

注：正方形以外の地形の場合、それぞれの葉の位置をリスト内に保存する描画ツールを作成できます。例えば、ゲーム開発者の Bronson Zgeb 氏による こちらの 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ビルド方法が説明されています。

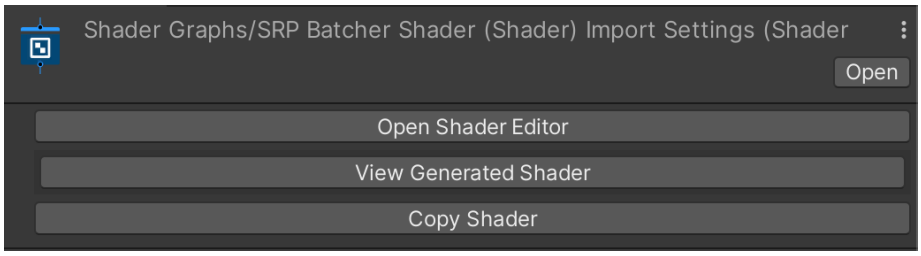
GPU インスタンシング

最適化手法の 1 つとしては、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処理するようにレンダラーに指示します。これにより、ドローコールの回数を削減できます。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 Shader

Graph を選択し、Inspector で **View Generated Shader** をクリックします。



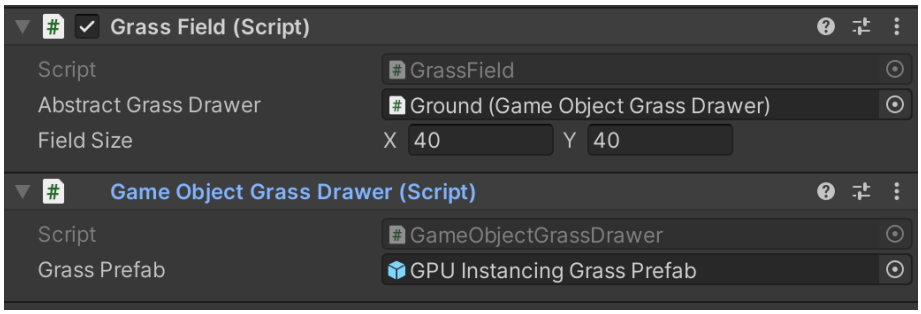
Shader Graph から HLSL シェーダーを生成する

HLSL シェーダーが作成され、Temp フォルダーに配置されて、選択したテキストエディタ—または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：

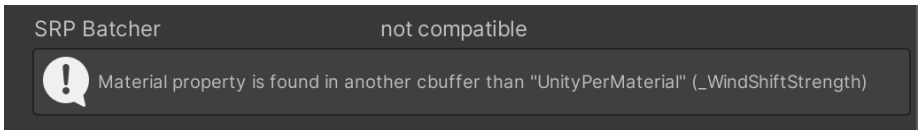
```
// グラフのプロパティ
//CBUFFER_START(UnityPerMaterial)
            float4 _MainTexture_TexelSize;
            half _WindShiftStrength;
            half _WindSpeed;
            half _WindStrength;
//CBUFFER_END
```

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 シーンと同じバージョンの Abstract Grass Drawer が使用されていることに注意してください。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を使用するマテリアルを持つ、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 との互換性がありません。

Inspector で GPU インスタンシングシェーダーを確認すると、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：

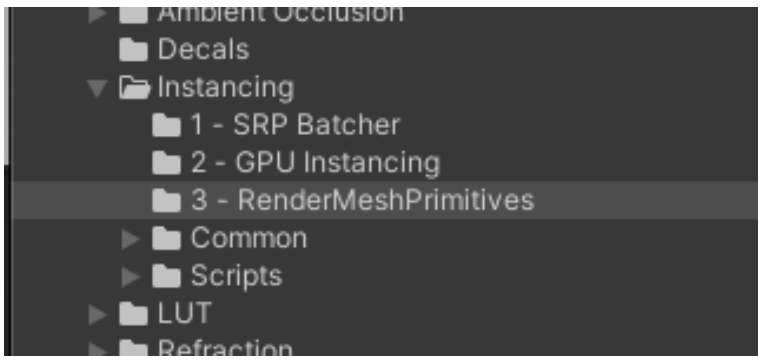
1. 生成されたシェーダーを表示するか、再生成します。
2. シェーダーの名前を編集します。
3. CBUFFER マクロをコメントアウトします。
4. アセットに保存します。

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 の Graphics API には、ゲームオブジェクトを使用せずにメッシュを直接レンダリングするメソッドがいくつかあります。ここで使用されているメソッドは RenderMeshPrimitives です。これは Unity LTS 2021 で導入された機能です。それ以前は DrawMeshInstancedProcedural を使用する必要がありましたが、現在ではこれは非推奨となっています。

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。その動作は、シーン (**Scenes > Instancing > 3 - RenderMeshPrimitives > 3 - RenderMeshPrimitives**) で確認できます。



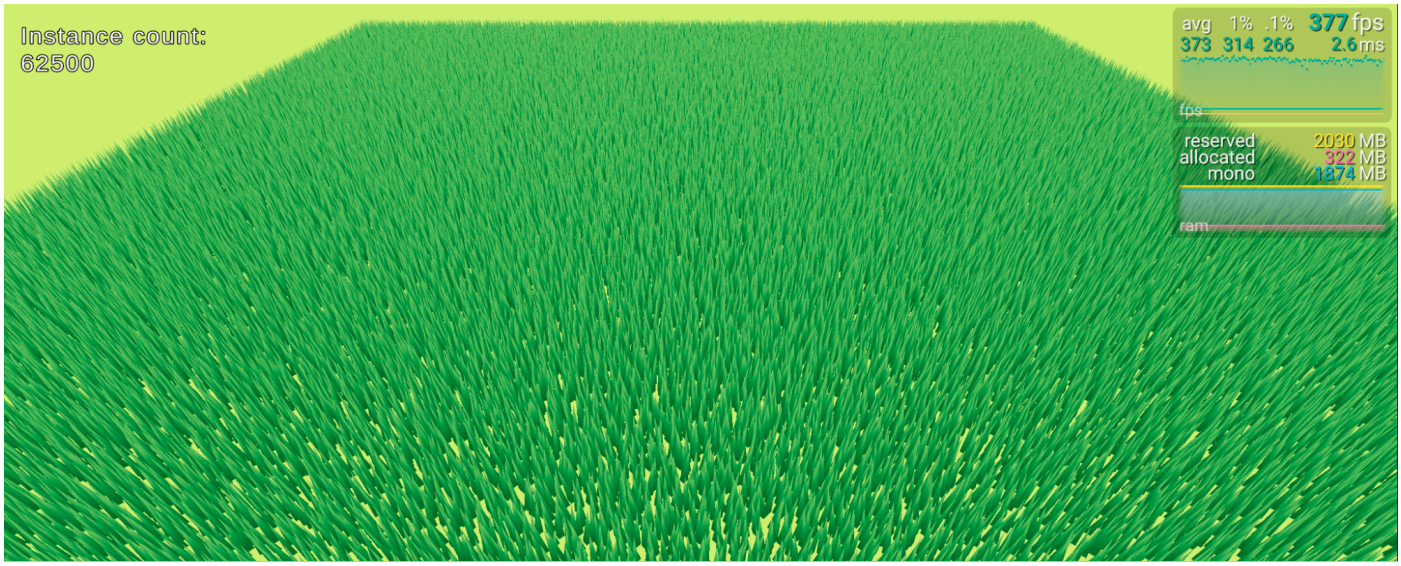
Project ウィンドウのインスタンシングシーン

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 377 fps と注目に値します。SRP Batcher と GPU インスタンシングで作成されたシーンは、それぞれ約 20 fps と 50 fps で動作していました。

この違いの要因は、草原を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 プロパティとして作成す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。

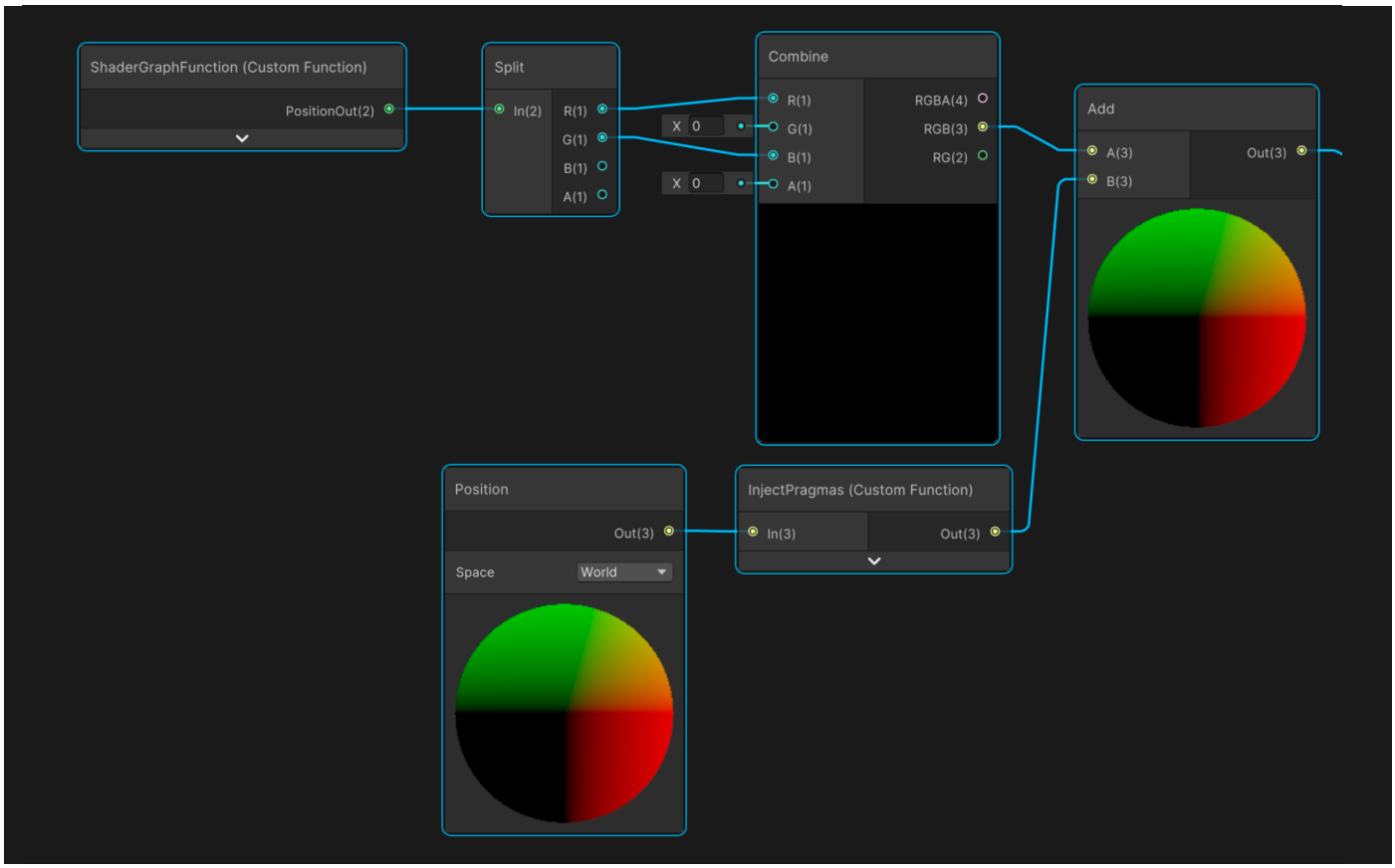
```
_positionsCount = _positions.Count;
_positionBuffer?.Release();
if (_positionsCount == 0) return;
_positionBuffer = new ComputeBuffer(_positionsCount, 8);
_positionBuffer.SetData(_positions);
_instanceMaterial.SetBuffer(Shader.PropertyToID("PositionsBuffer"), _positionBuffer);
```

_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 に簡単にデータを格納できます。これで、SetBuffer メソッドを使用して、このバッファをマテリアルにコピーできるようになりました。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。



ComputeBuffer 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 の頂点位置の **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。ただし、この手法を使用する際には毎回、ある重要なコードブロックを加える必要があります。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;
```

このシェーダーが位置の値を生成するために使用す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。その点を除けば、この Custom Function ノードは、入力の **In** を出力の **Out** に渡すだけの構成です。

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` メソッドを使用して設定され、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。

この時点のグラフでは、個々の葉の位置は `float2` であり、1 つ目の Float が X 値、2 つ目は Z 値です。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たら、WindSpeed、WindStrength、WindShiftStrength、MainTexture を入力として持つマテリアルで使用します。これらの入力は 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ているものと同じです。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、サブメッシュのインデックス、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には、ComputeShader を使用して処理するのが最適です。コンピュートシェーダーによるパフォーマンスの向上については、後のレシピで説明します。

その他のリソース

— このレシピで使用されている アセット

— DrawMeshInstancedIndirect を使用した サンプル プロジェクト

— GPU インスタンシングの ドキュメント

— CatLikeCoding による GPU インスタンシングに関する 記事

— 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

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』は、セルシェーディング手法により、ゲームを漫画のように見せる独特なアートスタイルを実現しています。こちらのクリエイター インタビュー も確認してください。

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 3 つの異なる見た目:標準的なシェーディング (左)、シンプルなトゥーンシェーディング (中央)、トゥーンシェーディング (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。

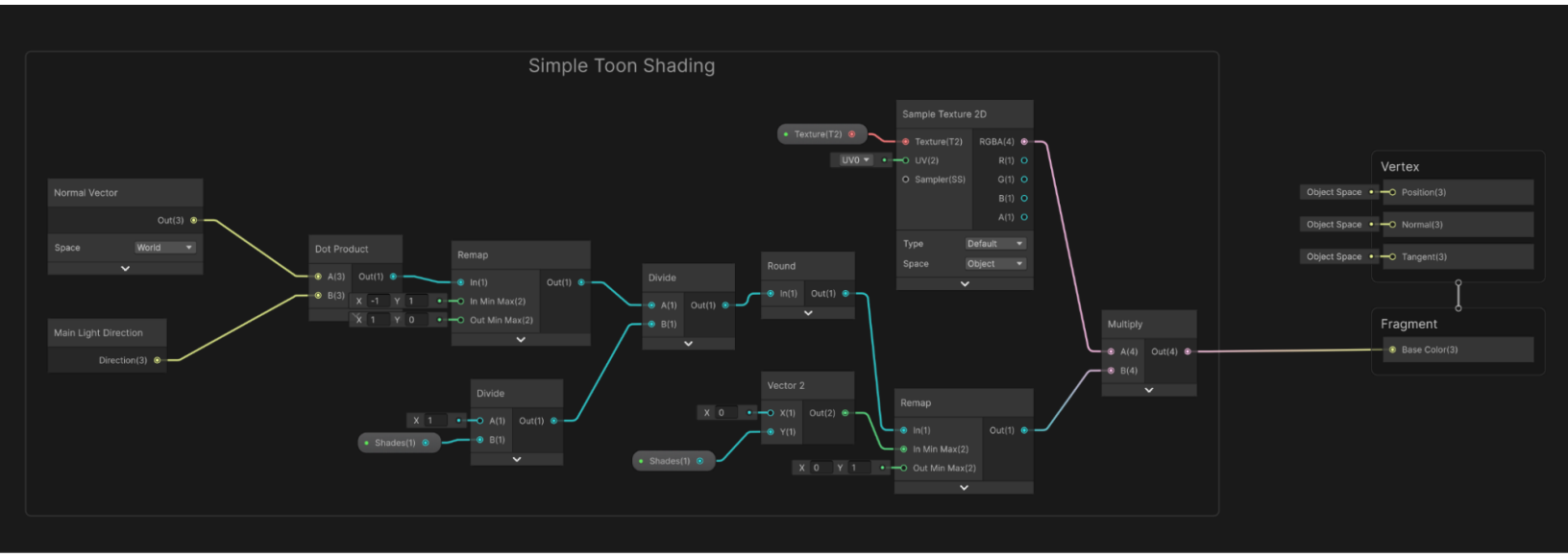
傾斜付きのシンプルなトゥーンシェーダーを使用したシーン

シンプルなトゥーンシェーディング

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。この課題は 2 つの異なるシェーダーに分割され、最初のシェーダーがモデル内の各ピクセルのメインとなる色の選択を担います。

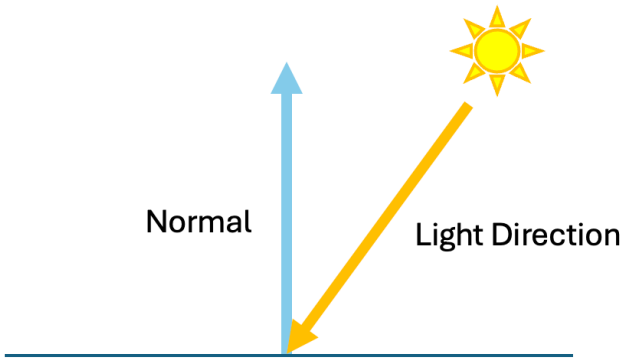
シェーディング

考えられる最もシンプルなライティングモデルは、ランバート ライティング です。このライティングモデルは、ライト方向と、その地点におけるワールド空間法線とのドット積を単純に計算します。下の画像は、このシェーダーのシェーダーグラフを示しています。



シンプルなトゥーンシェーディング用の Shader Graph

まず左側には、Normal Vector ノード と Main Light Direction ノード があります。これら 2 つのベクトルは Dot Product ノードに入力されます。2 つのベクトルが同じ方向の場合は出力が 1、逆方向の場合は出力が -1 になります。ライティングでは、2 つのベクトルが逆方向の場合に最大値になるようにします。



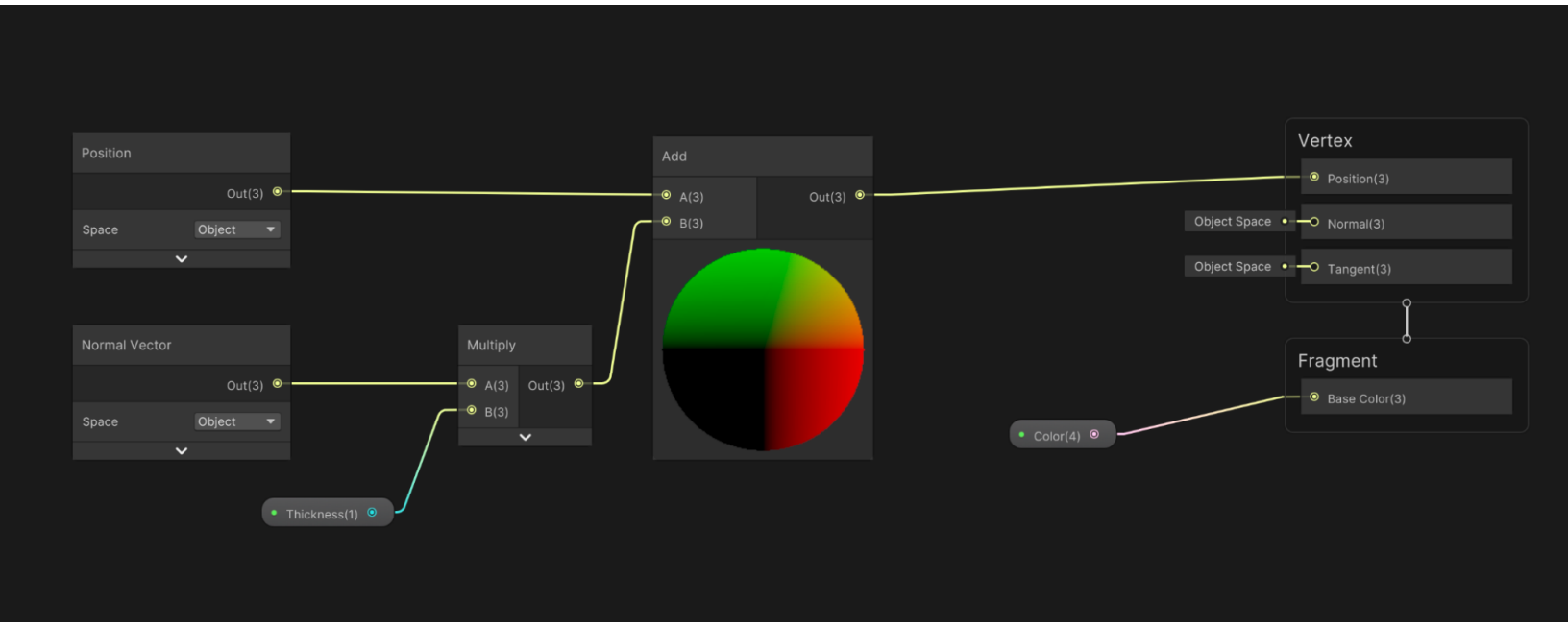
ランバートライティングモデル

ここでは、Remap ノードが使用されています。この便利なノードは、入力値をある範囲から、別の範囲に基づく新しい値に変換できます。**In Min Max** は **-1, 1**、**Out Min Max** は **1, 0** に設定されています。これにより、In の値が -1 の場合は Out が 1、In の値が 1 の場合は Out が 0 になります。

グラフには 2 つのプロパティがあります (`Texture` と `Shades`)。`Shades` は、0 から 1 の範囲で許容される傾斜ステップ数を定義する整数値です。連続的な補間を行うのではなく、値を段階的に実行する仕組みが必要です。まず、Divide ノードを使用して、1 を `Shades` プロパティで除算します。`Shades` が 4 の場合、この時点で Divide ノードの出力は 0.25 になります。次に、2 つ目の Divide ノードを使用し、Remap ノードの出力を `Shades` の Divide ノードの出力で除算します。これにより、値の範囲が [0, 1] から [0, Shades] に変更されます。Round ノードを使用して値を整数値に変換します。この時点で指定できる値は 0、1、...、Shades のみです。

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、GitHub サンプルの **Scenes > Toon Shading > VertexOutline** に含まれています。そのグラフを下に示します。



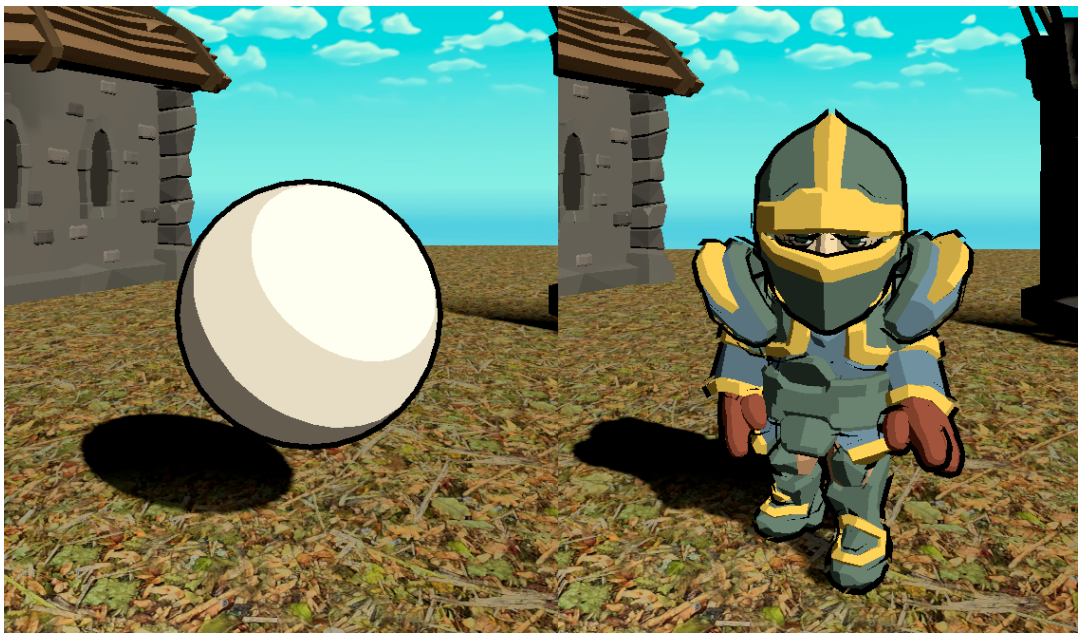
背面の頂点をシフトさせる手法を使用しているアウトラインシェーダー

Space が **Object** に設定された Normal Vector ノードは、Multiply ノードに入力されます。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。これが **Vertex Position** プロパティへの入力になります。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適用

Scenes > Toon Shading > Simple Toon Shading 内のシーンでは、2 つ目のマテリアルが使用されているのが確認できます。同じフォルダー内のマテリアル VertexOutline を Inspector で表示し、Thickness を 0.02 に設定します。

この手法は、単純な凸形状には効果的ですが、形状が複雑になると、下の画像に示すように、均一な太さの線が作成されません。では、別のアプローチを見てみましょう。



拡張背面モデルを使用したアウトライン: シンプルなスフィア (左)、複雑なモデル (右)

トゥーンシェーディング

別の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 > Toon Shading** に移動します。先ほどと同様に、シェーディングタスクは 2 つの異なるシェーダー (シェーディングとアウトライン) に分割されています。最初のステップは、モデル内の各ピクセルのメインとなる色の選択を設定することです。

シェーディング

メインライトの Shader Graph ノードは、**Main Light Direction** ノードのみです。

注: URP におけるメインライトとは、最も強度の高いディレクションライトのことです。

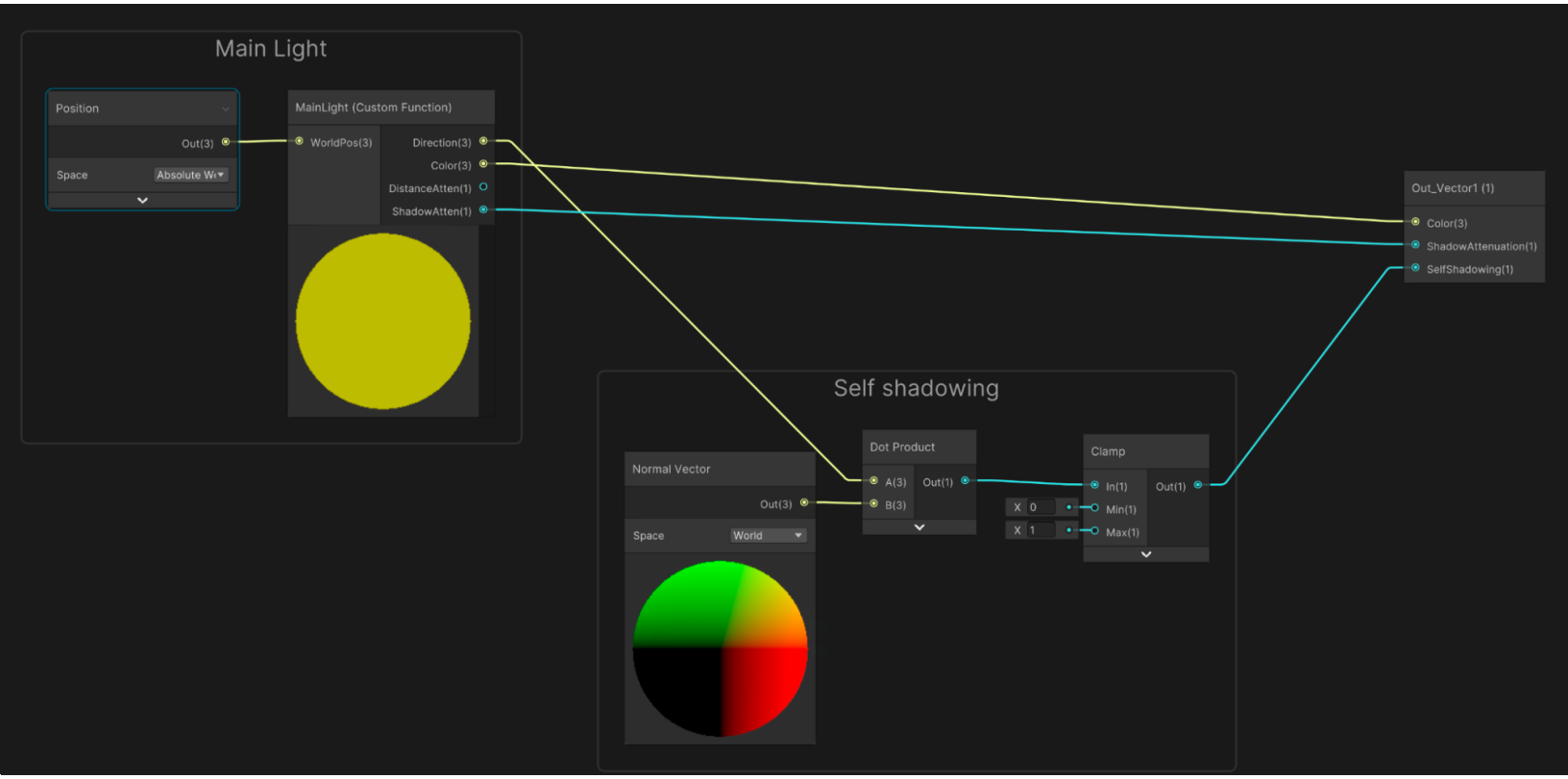
Main Light Direction ノードは、シンプルなトゥーンシェーディングのレシピで使用されていました。そのシンプルなバージョンを改良するために、ライトの色と影を組み込むことができます。まず、ファイル (**Shaders > HLSL > Custom Lighting.hlsl**) にあるカスタム関数を使用して、メインライトにアクセスします。

```
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(WorldPos);

    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 = GetMainLightShadowSamplingData();
        float shadowStrength = GetMainLightShadowStrength();
        ShadowAtten = SampleShadowmap(shadowCoord,
        TEXTURE2D_ARGS(_MainLightShadowmapTexture,
        sampler_MainLightShadowmapTexture),
        shadowSamplingData, shadowStrength, false);
    #endif
#endif
}
```

Shader Graph アセット 作成時の動作を定義するために、`#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推奨します。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 に含まれており、Shader Graph のカスタム関数に使用できます。関数 `GetMainLight` を float4 で使用すると、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。

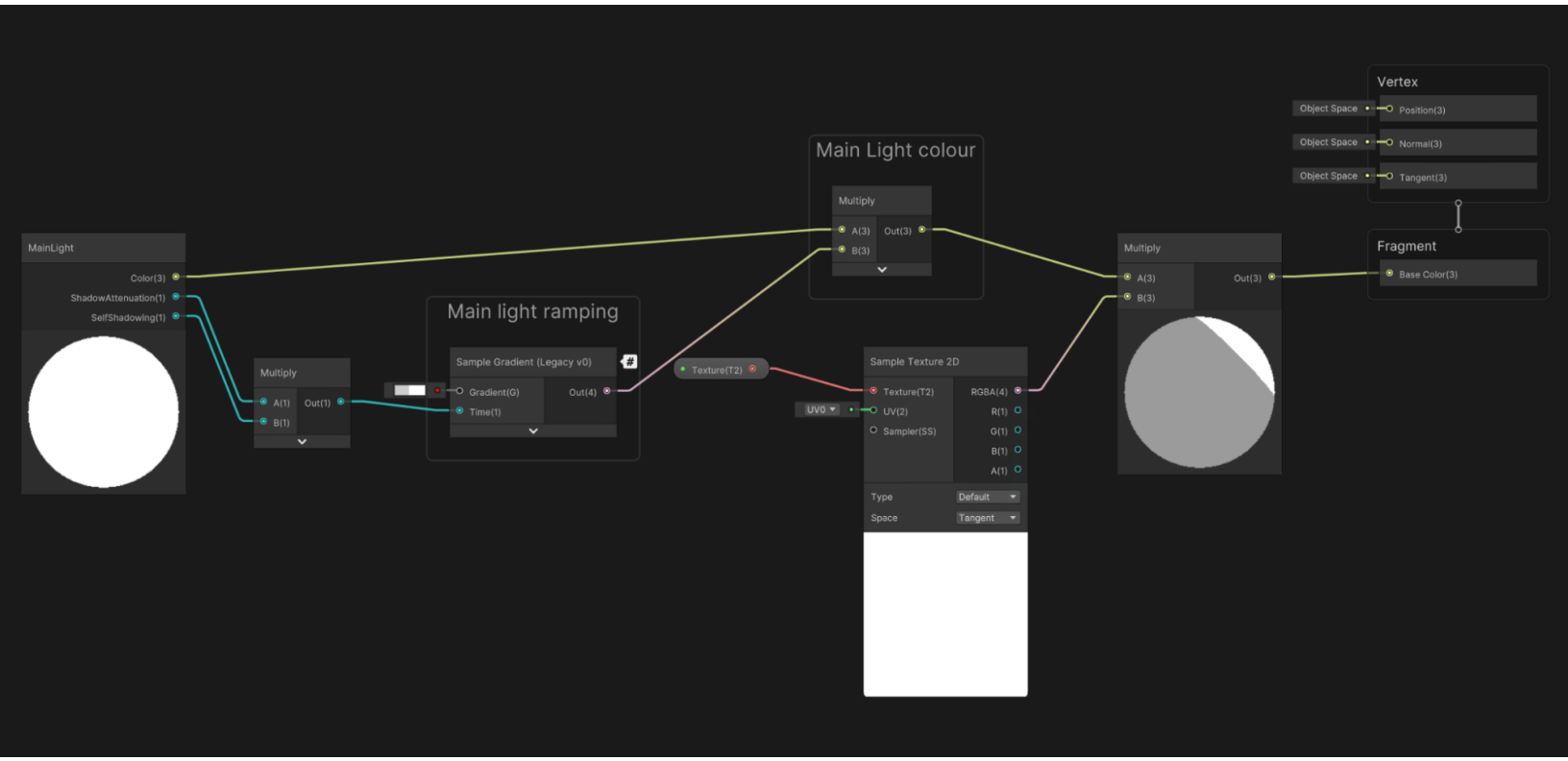
このコードは、フォルダー (**Shaders > Subgraphs**) にある Main Light サブグラフで使用されます (下の画像を参照)。これをよく見ていきましょう。



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。セルフシャドウを可能にするには、ライト方向とワールド法線とのドット積を取得し、それを 0 から 1 の範囲に固定します。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。**Scenes > Toon Shading > Toon Shading** の順に選択して、グラフを確認します (下にも画像で示されています)。



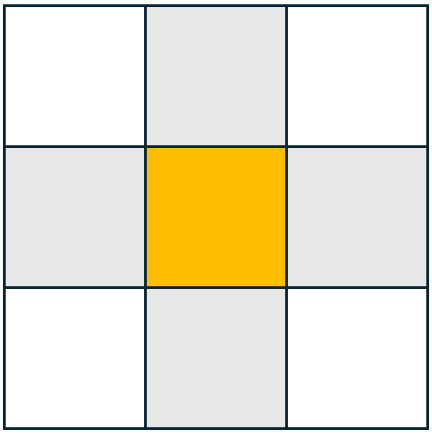
シンプルなトゥーングラフ

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。ここでのポイントは、この出力を、傾斜付きのグラデーションを扱う Sample Gradient ノードに渡すことです。これにより、ライトの強度が滑らかに変化するのではなく、グラデーションに基づいて段階的に切り替わるようになります。

滑らかに変化する入力を受け取り、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、ライト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。

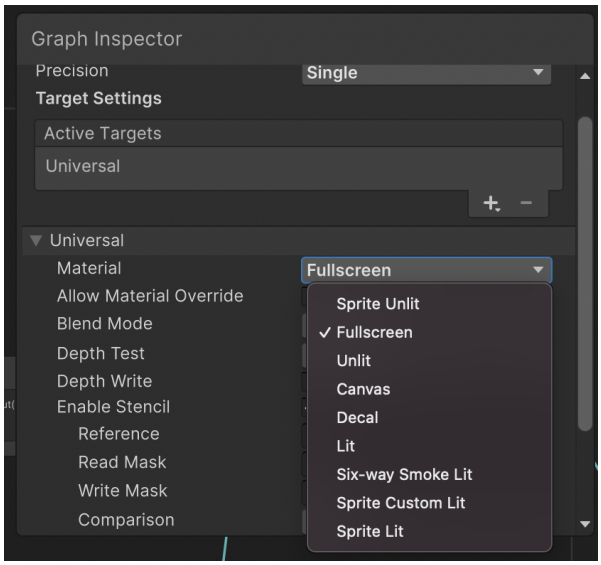
アウトライン

アウトライン作成のより洗練された手法としては、Sobel フィルター によるエッジ検出を使用することです。これにはポストプロセス手法を使用できます。右の画像にあるオレンジの正方形が、現在分析中のピクセルであると想像してください。このオレンジのピクセルとグレーの正方形で、バッファがどれほど異なるかを確認するとします。チェック対象のバッファは、Color G-buffer と Normal G-buffer です。変化量がプロパティの強度を満たす場合、オレンジのピクセルがエッジ境界上にあるとみなし、その色を **Outline Color** プロパティの色に更新できます。



Sobel フィルタリング

これをハンドルするには、**Create > Shader Graph > URP > Full Screen Shader Graph** を使用して Shader Graph を作成します。または、Shader Graph ウィンドウの右上にあるアイコン ("i" が丸で囲まれたアイコン) をクリックして Graph Inspector にアクセスし、すでに存在する Shader Graph の動作を変更することも可能です。

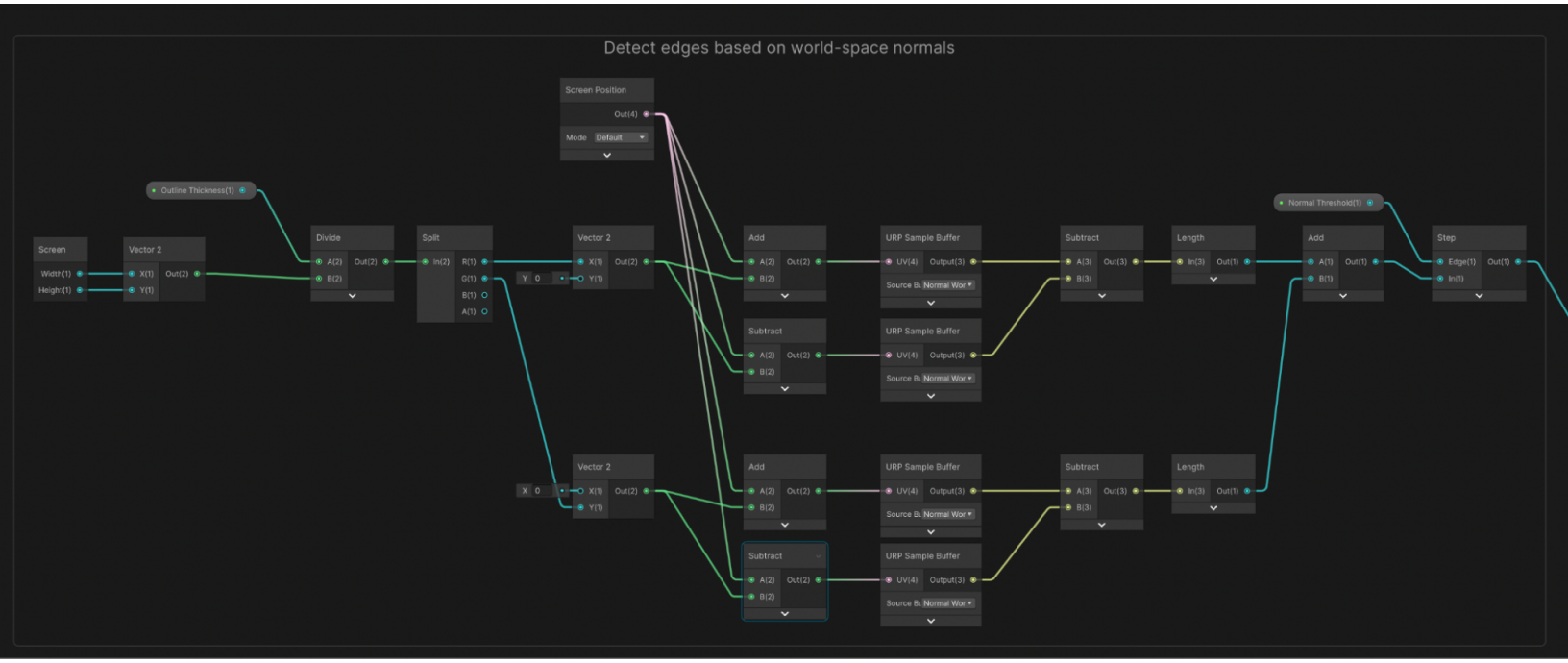


Shader Graph Inspector の Material プロパティのドロップダウン

Scenes > Toon Shading > Outline で Shader Graph を確認します。まずは、グラフの左にある **Detect edges based on the world space normals** (ワールド空間法線に基づき端を検出する) セクションを見ていきましょう。

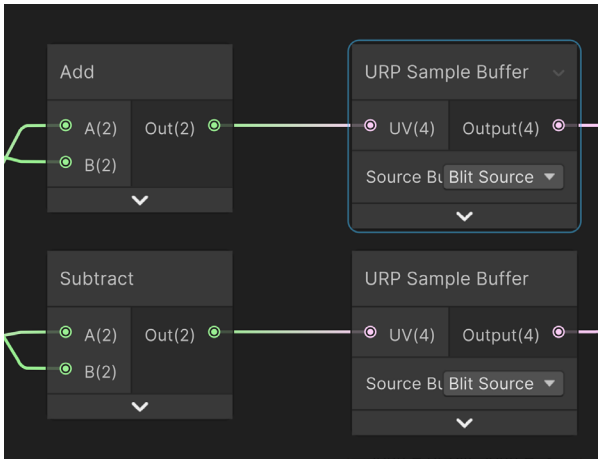
1. 最初のノードは Screen ノードで、レンダリングされる画像サイズをピクセル単位で指定します。これは Vector2 に換算できます。
2. 次に、Divide ノードを使用して、**Outline Thickness** プロパティをこのベクトルで除算します。これで、オフセット値として使用できる値が得られました。
3. オフセット値の x および y 値にアクセスするには、Split ノードを使用します。Split ノードは、r = x および g = y の RGBA 出力を提供します。
4. 2 つの Vector2 を作成します。一方は x 方向、もう一方は y 方向のオフセットを指します。x オフセットには y = 0、y オフセットには x = 0 を設定します。これで、Add ノードと Subtract ノードを使用して、現在の Screen Position に対してオフセットを加算および減算できます。
5. この時点で、スクリーンスペース上の 4 つの位置が得られています。左下が [0,0]、右上が [1,1] です。4 つの URP Sample Buffer ノードを使用して、この位置でのワールド法線を取得します。

6. 右側の法線から左側の法線を減算して、この新しいベクトルの Length を取得します。さらに、上側の法線から下側の法線を減算して、こちらもベクトルの Length を取得します。
7. これで、現在のスクリーンピクセルの左右の差、および上下の差のスカラー値を取得できました。
8. Add ノードを使用してこれらの値を合計し、分析対象の 4 つのピクセルの合計差をスカラー値として算出します。分析を完了するには、Step ノードを使用します。値が **Normal Threshold** プロパティを超えている場合は 1 が返されます。



ワールド空間法線に基づいてエッジを検出します。

9. カラーバッファについても、ほぼ同じことを行いますが、Normal World Space ではなく Blit Source にアクセスし、Normal Threshold ではなく Color Threshold を使用します。

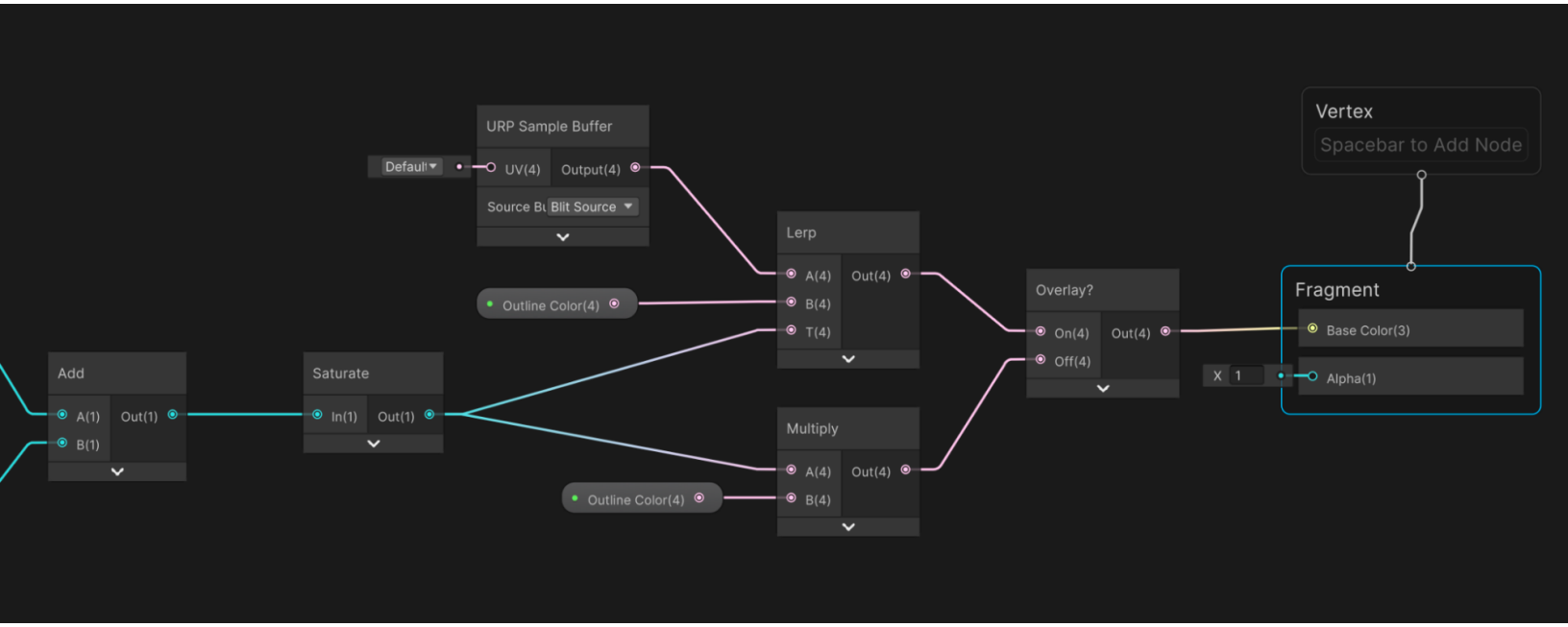


色エッジ検出の場合、サンプルの Source Buffer は Blit Source に設定されています。

10. ステップ 9 の後は、0 または 1 の値を持つ 2 つの出力があるはずです。グラフの残りの部分を下に示します。左から出ている線は、法線エッジ検出の出力 (上) と色エッジ検出の出力 (下) です。Add ノードを使用してこれらの値を加算すると、値が 0、1、または 2 になるはずです。

11. Saturate ノードを使用して、この値を 0 から 1 に固定します。

12. 次に、Lerp ノードを使用して、URP Sample Buffer ノードから得られた既存ピクセル色と、Outline Color プロパティの間を補間します。Overlay を使用することも可能です。このオプションが **On** に設定されている場合、出力は既存の Blit Source と Outline Color プロパティの合成になります。Overlay が **Off** に設定されている場合は、Outline Color または黒がそのまま渡されます。最終的な出力は、Fragment の Base Color 入力につながります。



アウトラインシェーダーの最後のノード

このレシピでは、トゥーンシェーディングの主な手法を紹介します。

その他のリソース

— Unity Open Project の GitHub リポジトリ (この章のコードの大部分はこの Open Project からのものです)

— YouTube 上の Unity Open Project

— Ned Makes Games による YouTube チュートリアル (トゥーンシェーディングに関する短いシリーズが含まれています)

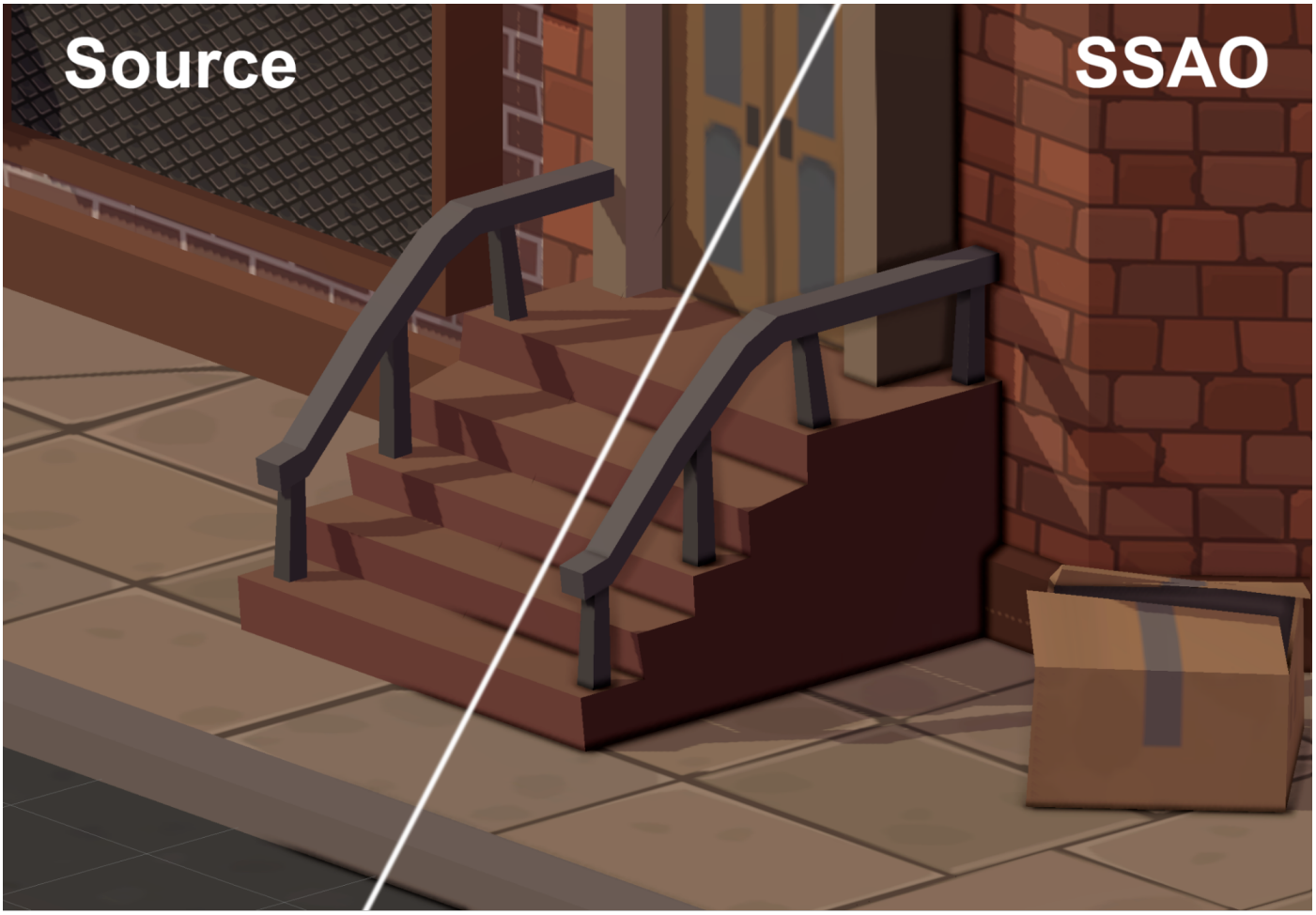
— AE Tuts による Unity の SobelFilter.shader の使い方に関する YouTube チュートリアル (Shader Graph のチュートリアルに特化したチャンネルです)

— Alexander Ameye 氏による Sobel フィルターを使用した辺の検出の解説

— Daniel Ilett 氏によるセルシェーディングシリーズおよび全画面アウトライン シェーダーのチュートリアル

アンビエント オクルージョン

Original Fire Games によるレーシングゲーム『*Circuit Superstars*』は、Made with Unity のゲームです。スクリーンスペースアンビエントオクルージョン (SSAO) などの URP 機能を使用して、ゲーム環境において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。



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、相互に接近した溝、穴、サーフェスを暗くするポストプロセス手法です。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。上の画像では、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。

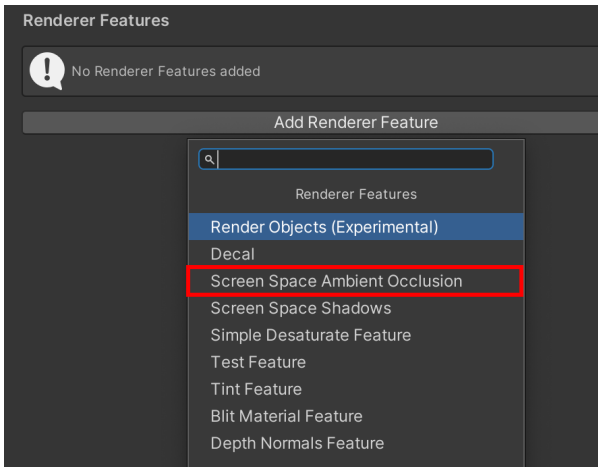
URP には、リアルタイムの スクリーン スペース アンビエント オクルージョン（SSAO）エフェクトが Renderer Feature として実装されています。使用するパスコードは こちら で確認できます。

注：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。

このシーンでは、**Ambient_Occlusion_URP_Settings** という名前の URP アセットが使用されています。これは、**Scene > Main Camera** にアタッチされた `AutoLoadPipelineAsset` スクリプトを介して、シーン読み込み時に自動でロードされます。この URP アセットは、**Ambient_Occlusion_URP_Settings_Renderer** を使用しています。

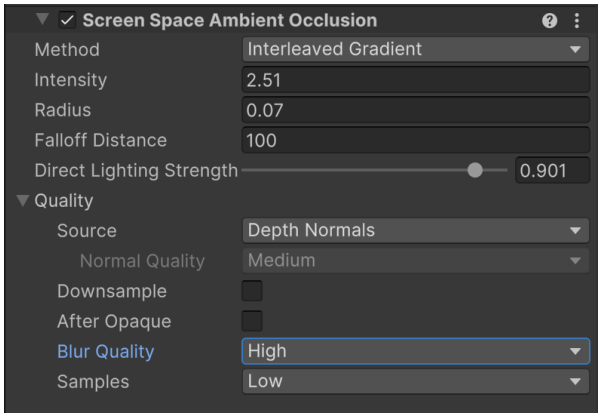
シーンに SSAO を加えるには、Inspector で Universal Renderer Data アセットを開き、**Add Renderer Feature** をクリックします。ドロップダウンメニューから **Screen Space Ambient Occlusion** を選択します。



SSAO Renderer Feature の適用

SSAO プロパティ

SSAO Renderer Feature を加えると、Inspector を介して結果を制御できるようになります。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

- **Method**:Interleaved Gradient と Blue Noise のどちらかを選択します。
- **Intensity**:これは、暗さの強度を制御します。
- **Radius**:現在のピクセルの周囲にあ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。
- **Falloff Distance**:アンビエントオクルージョンが適用されなくなるシーン距離を制御します。

- **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。これは、直接光が当たる部分におけるアンビエントオクルージョンの強度に影響します。
- **Quality > 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**:代わりに SSAO は、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。
- **Quality > Source > Normal Quality**: **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

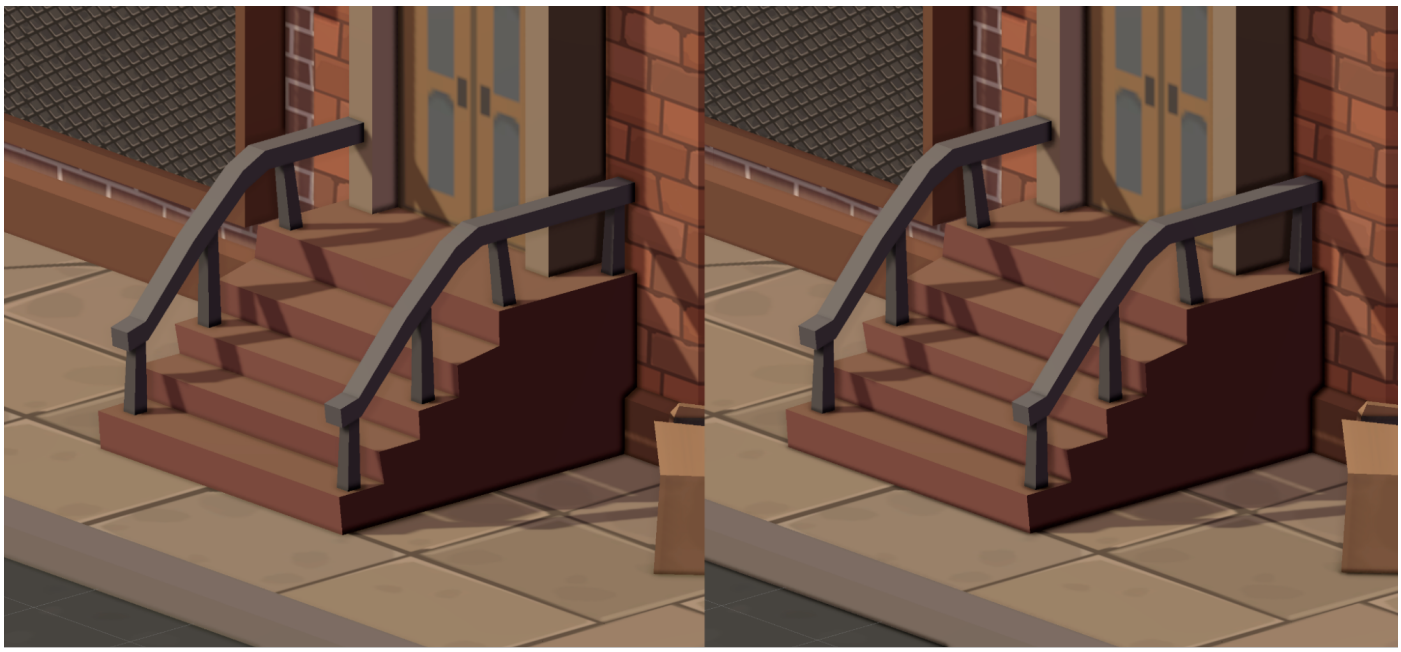
  パフォーマンスへの影響は中程度と見なされます。
- **Quality > Downsample**:これを選択すると、X と Y 両方の方向で処理の解像度が半分になります。これにより、処理するピクセル数が事実上 75% 減るため、GPU 負荷も大幅に軽減されますが、エフェクトのディテールは少なくなります。
- **Quality > After Opaque**:このオプションは、最終的なレンダリングの外観に作用しますが、パフォーマンスに影響を及ぼします。
  - **無効の場合:**SSAO には Depth または Depth Normals のプリパスがあります (下の Source 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。
  - **有効の場合**:After Opaque が選択されている場合、SSAO には Depth Normals が必要です。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。

注:Render Opaque パスで Depth + Normals をレンダリングできるようにしておくのも便利です。このオプションを有効にすると、プリパスを完全にスキップしてパフォーマンスを節約できます。

— **Quality > Blur Quality**:High、Medium、Low のいずれかに設定できます。リソースが限られているデバイスでは、満足できる結果が得られる最も低い設定を使用してください。

— **Quality > Samples**: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、パフォーマンスは低下します。



Direct Lighting Strength の 2 つのバリエーション:0.2 (左) と 0.9 (右)。右の画像では、各ステップの間に濃い線が入っていることに注目してください。

SSAO は、URP の柔軟性を表す好例です。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

— UGuruz による YouTube チュートリアル

— 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)

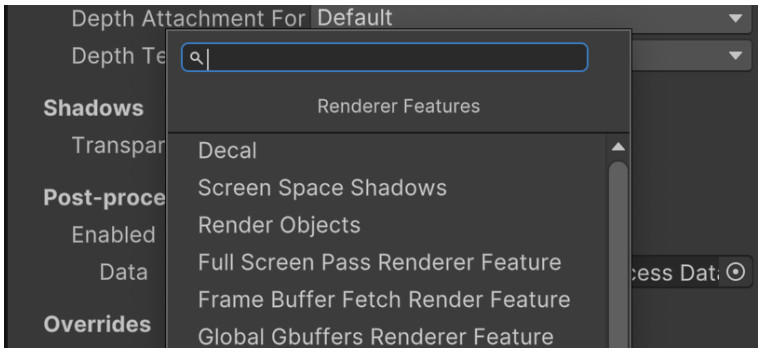
デカール

デカールの多くのユースケースの 1 つは、3D サーフェス上にブロブシャドウを投影することです。これは、Splashteam による Made with Unity のゲーム『*Tinykin*』のキャラクター、Milo に使用されています。

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。このレシピのシーンファイルとアセットは、フォルダー (**Scenes > Decals**) にあります。

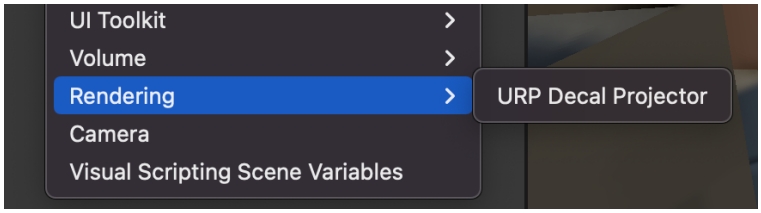
無機質なシーンに加えられたデカール

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 が加えられていることがわかります。通常どおり、メインカメラにアタッチされた `AutoLoadPipelineAsset.cs` スクリプトによって、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。カスタムシーンにデカールを加えるには、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。



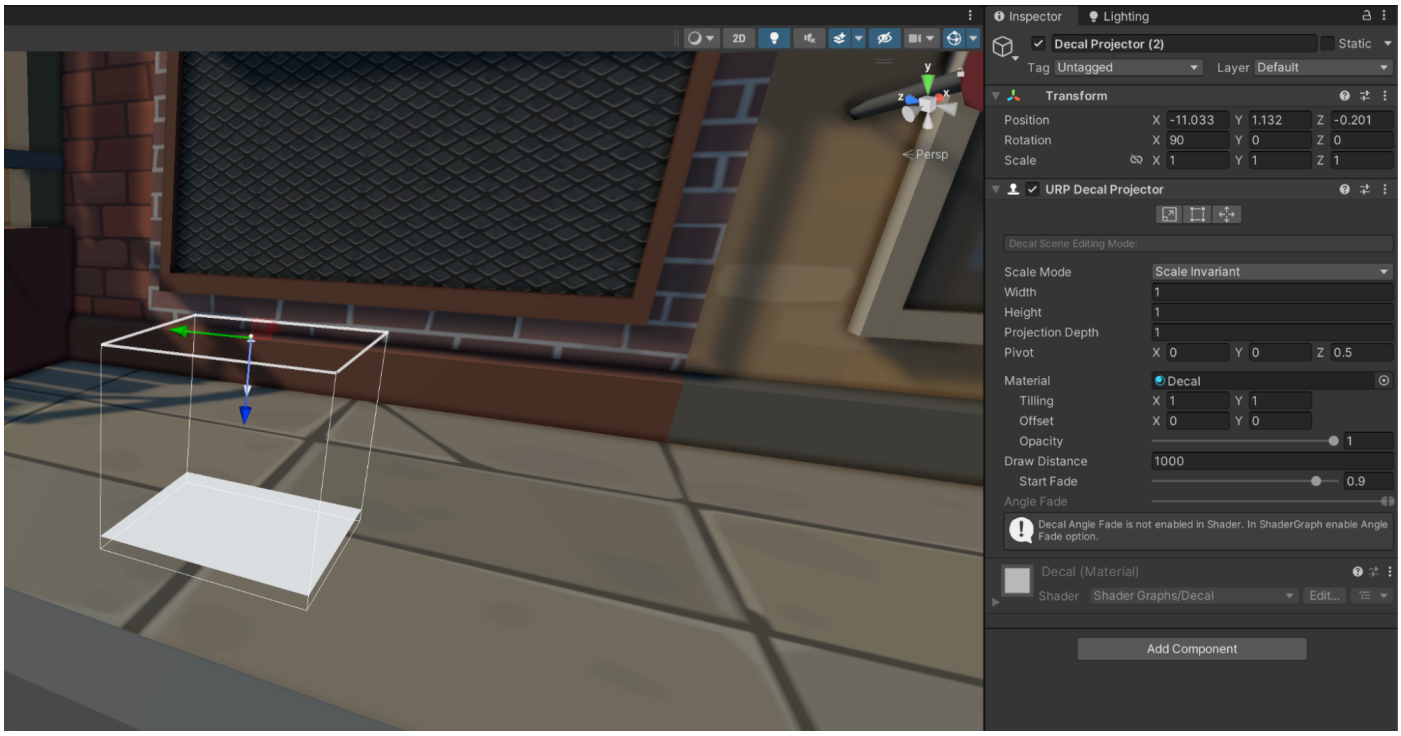
Add Renderer ドロップダウンの Decal オプション

エディターで作業している際にシーンにデカールを加えるには、Hierarchy ウィンドウを右クリックし、**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

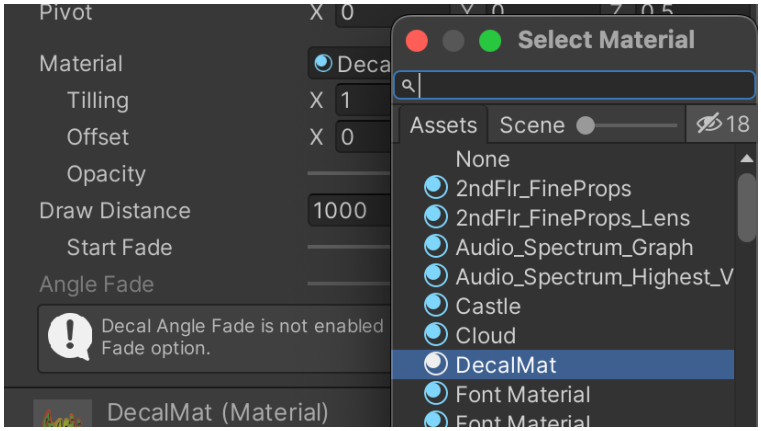
— **Scale Mode:**デフォルトでは、URP Decal Projection コンポーネントの **Scale Mode** は **Scale Invariant** に設定されています。これは、デカールのサイズが **Width** プロパティと **Height** プロパティによってのみ決定されることを意味します。**Inherit from Hierarchy** に切り替えると、ゲームオブジェクトの Transform スケール と Width および Height プロパティが組み合わさってサイズが決定されます。

- **Width と Height:**デカールのサイズを制御するプロパティです。
- **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、プロジェクターバウンディングボックスの中心位置のオフセットを設定します。
- **Material**:投影するマテリアルを設定します。このマテリアルは Shader Graph/Decal シェーダーを使用している必要があります (詳細は後ほど説明します)。
- **Tiling と Offset:**UV 軸に沿ったデカールマテリアルのタイリング値とオフセット値です。
- **Opacity**:不透明度の値を指定します。値が 0 の場合、デカールは完全に透明になります。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:**カメラからデカールまでの距離です。この距離を超えると、プロジェクターはデカールを投影せず、URP はデカールをレンダリングしなくなります。
- **Start Fade**:プロジェクター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 の値で、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% の距離に達した時点でデカールのフェードアウトを開始し、Draw Distance に達すると完全に消えます。
- **Angle Fade**:デカールの後方方向と、デカールを受け取るサーフェスの頂点法線との間の角度に基づいて、デカールのフェードアウト範囲を設定します。

マテリアルを作成する

Decal Projector を使用するには、Shader Graph/Decal シェーダーを使うマテリアルが必要です。この例では、Scene フォルダーにある DecalMat というマテリアルを使用します。ベースマップが割り当てられていますが、法線マップは使用されていません。これは、デカールのサーフェスをゴツゴツした外観にする場合に役立ちます。

Inspector で、この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

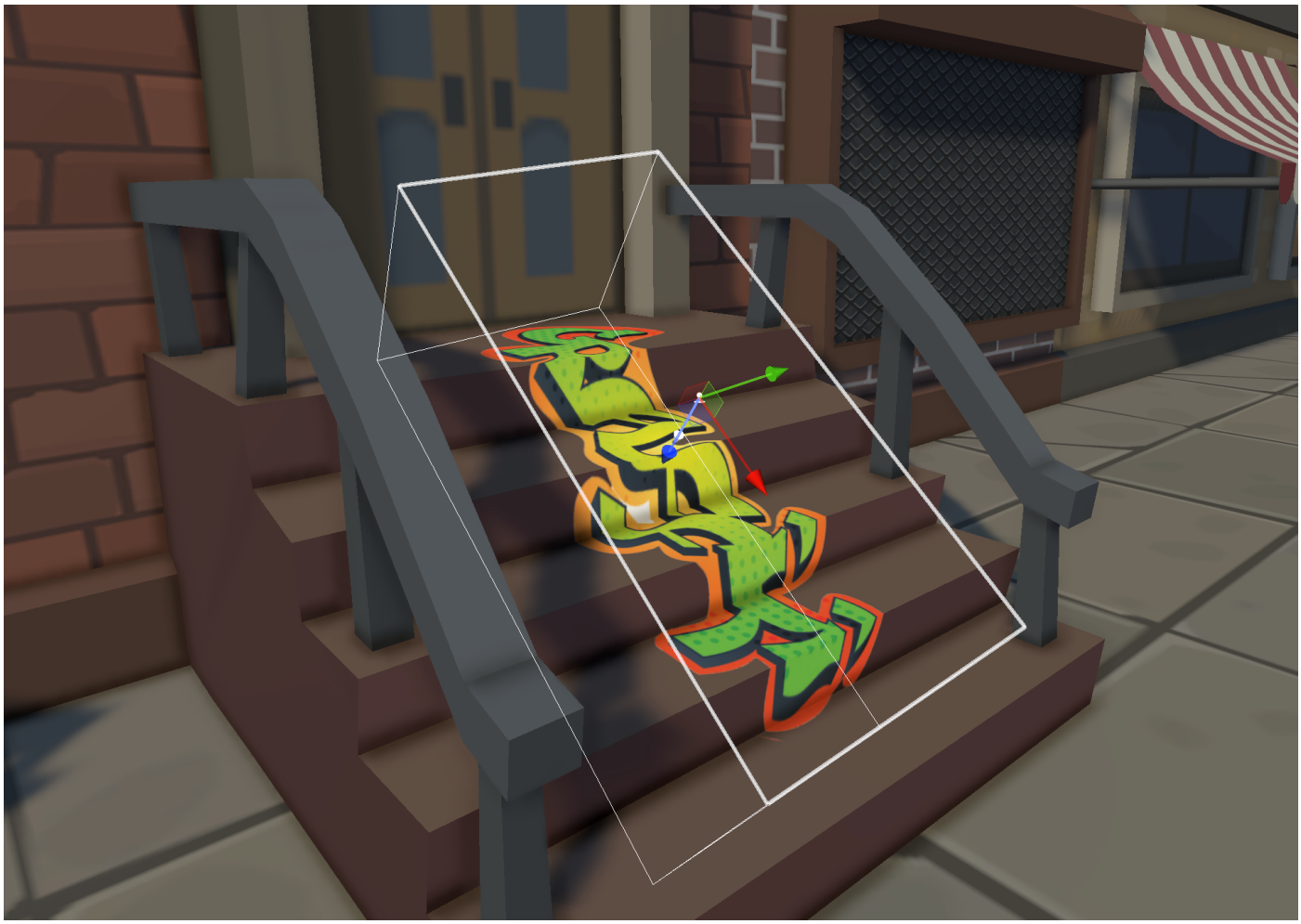
コードでデカールを加える

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、高さなどのプロパティを設定できますが、これらはランタイム中にコードで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、デカールが加わる" という動作の完全なコードは、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(decalProjectorPrefab);

    // 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(decalProjectorComponent.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をクリックした後で RaycastHit が発生する場合です。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 の ドキュメント

— Llam Academy による YouTube チュートリアル

水

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。この画像は、Flow Studio による Made with Unity のサバイバルゲーム『*Len's Island*』からのものです。

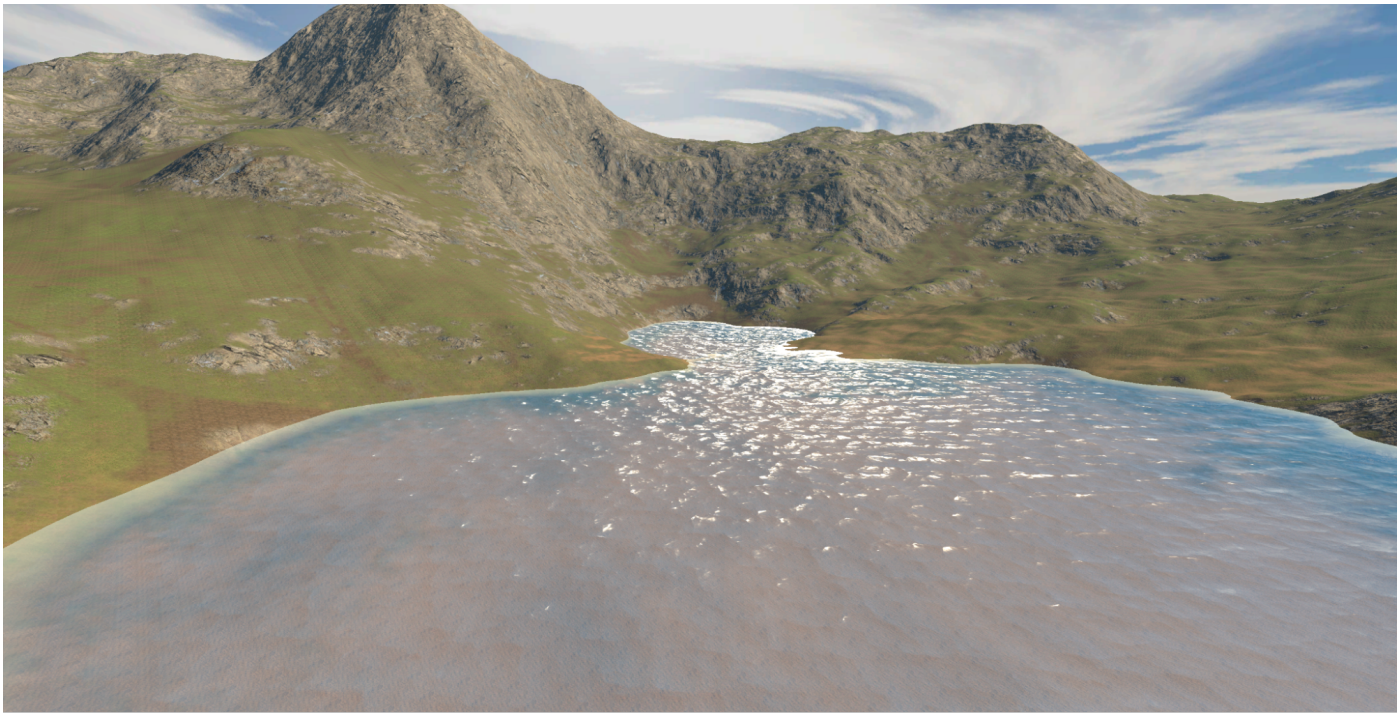
このレシピは、ただただ、水のシェーダーを作成するためのものになります。これはシェーダーグラフ内で作成されるため、ステップはアーティストやデザイナーにとって利用しやすくなります。

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

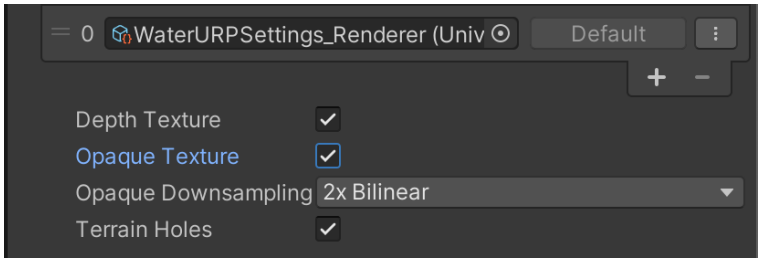
— タイル状の法線マップを動かして、さざ波をサーフェスに加える

— 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



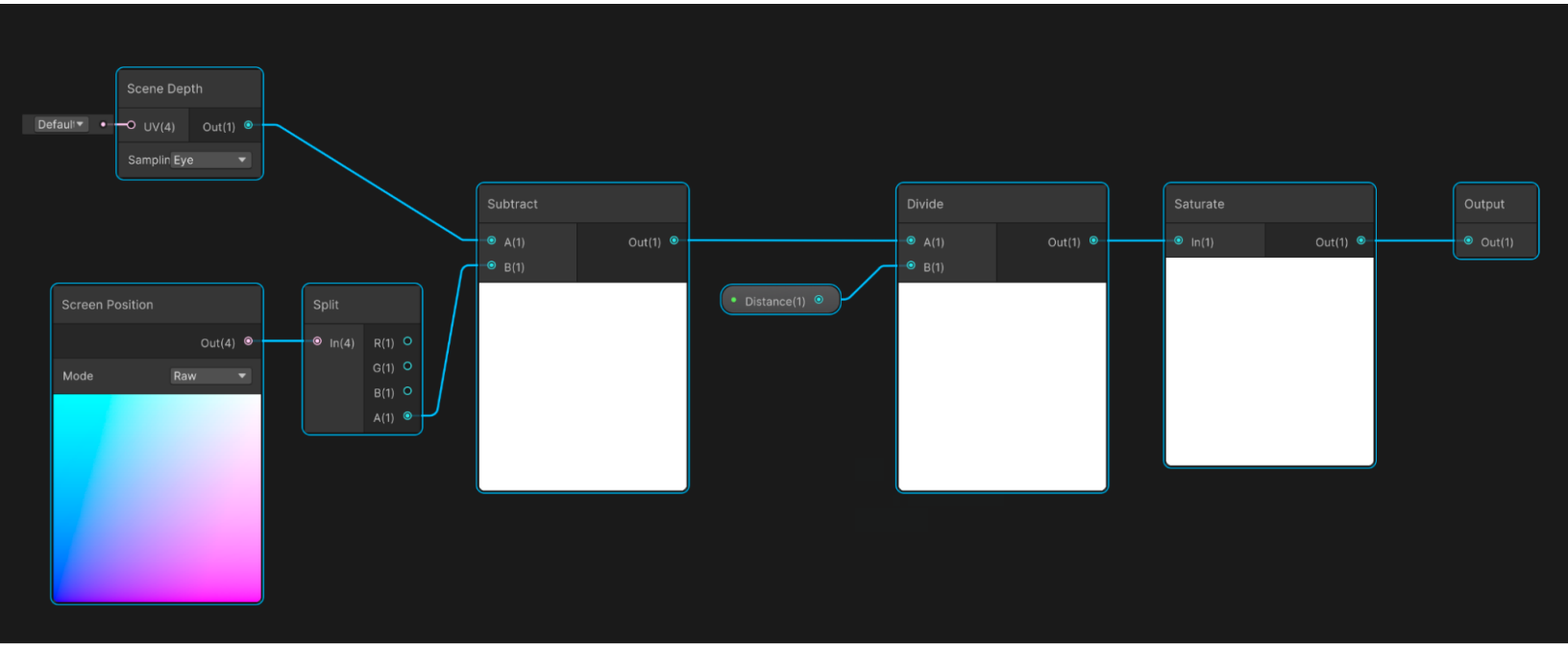
シンプルな水のシェーダーの動きを示す 動画 からの静止画像

完成形を確認するには、フォルダー (**Scenes > Water**) 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ましょう。Water シーンでは、**WaterURPSettings アセット** が使用され、**Depth Texture** と **Opaque Texture** のオプションが有効化されています。Opaque Texture が必要になるのは、このレシピには含まれていない屈折などの効果を加え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



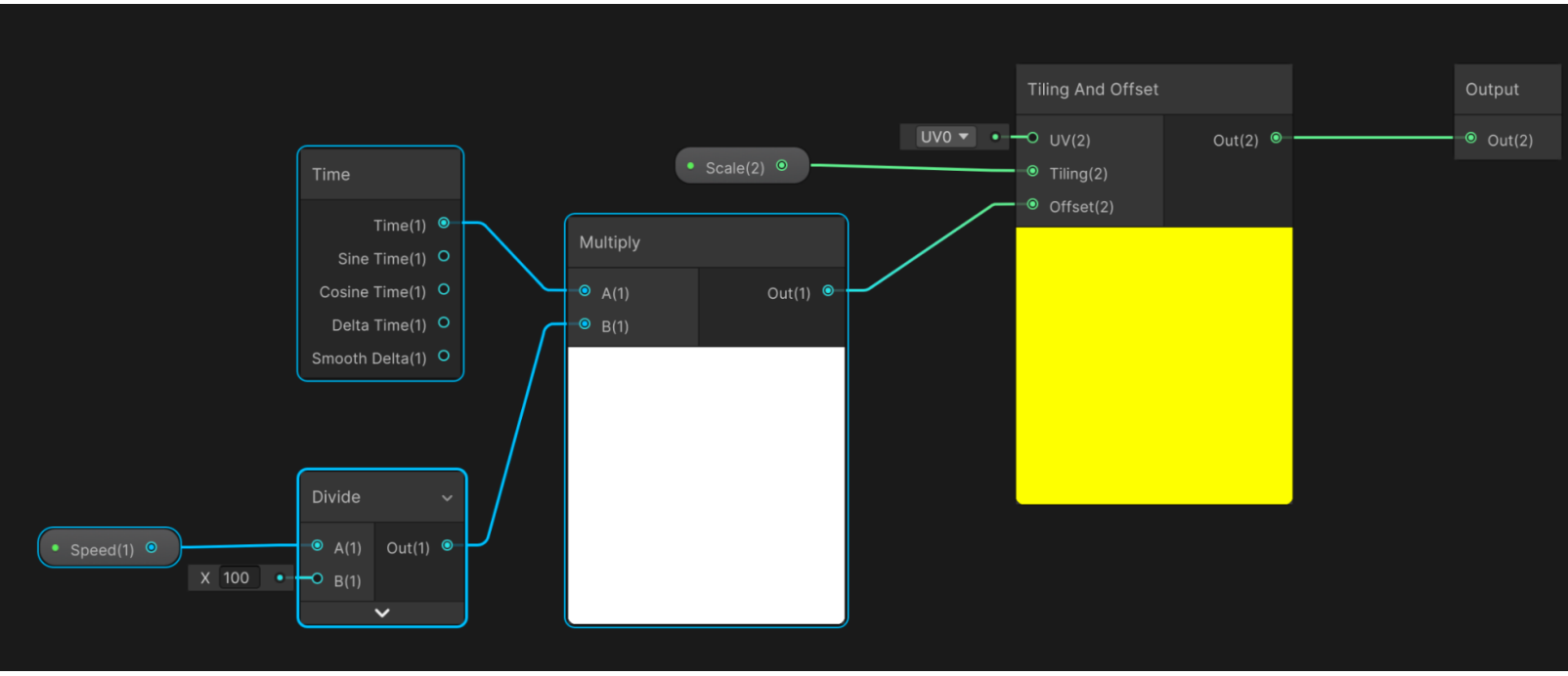
DepthFade サブグラフ

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 モードに設定された Scene Depth ノードにより、現在のピクセルにおける視点から不透明なジオメトリ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 ノードにより、現在の水ピクセルのレンダリング位置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、視点から現在の水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 ノードで、浅い部分と深い部分の間の端をどこにするかを制御します。このサブグラフの出力は 0 から 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 までの間に制限する、特殊な Clamp ノードとして機能します。

TextureMovement サブグラフ

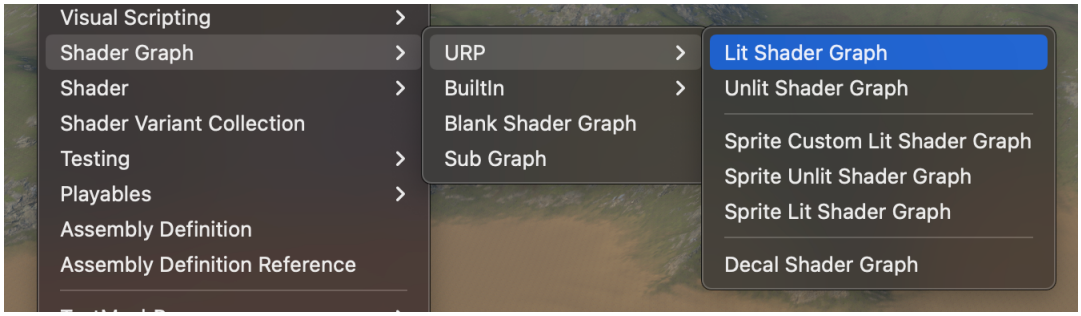


TextureMovement サブグラフ

水のシェーダーには、動きのあるテクスチャがいくつか含まれ、それらは 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 ノードに対する入力の 1 つ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 ノードの Offset への入力として使用されます。**Scale** プロパティは Tiling への入力となります。このシンプルなサブグラフは、時間の経過とともに、Speed と Scale が入力された Sample Texture 2D ノードで使用される UV を更新します。

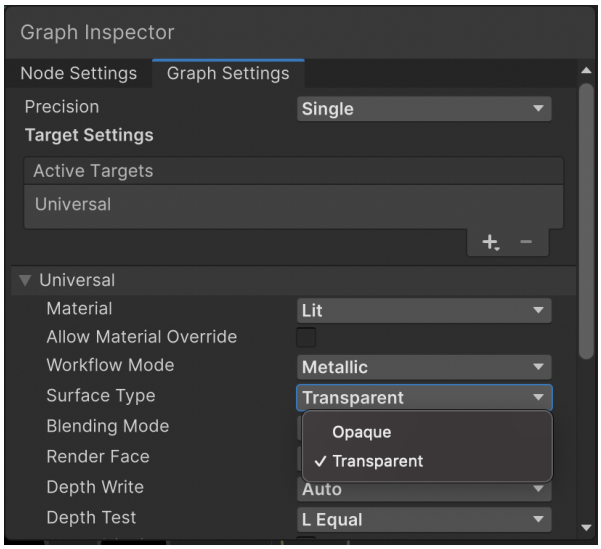
水のシェーダー

ここから、Lit シェーダーグラフ (**URP > Lit Shader Graph**) を使用して水のシェーダーを作成します。



Create > Shader Graph > URP > Lit Shader Graph

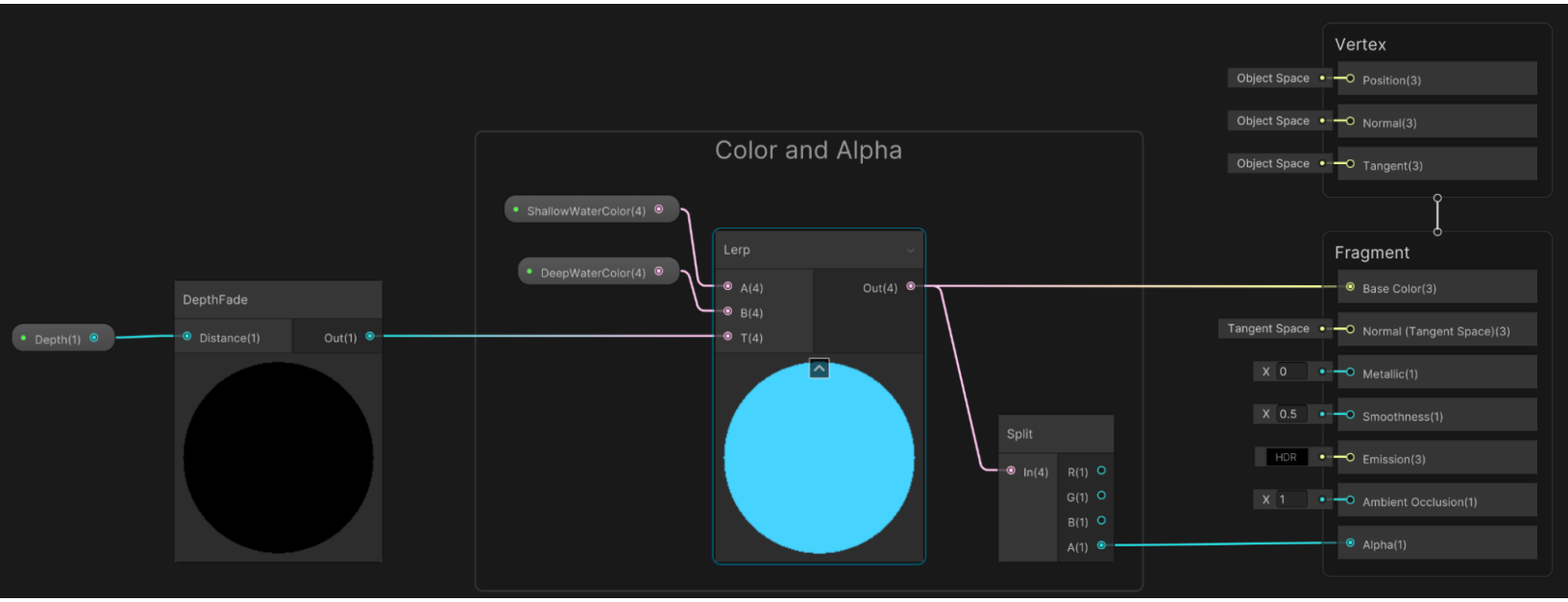
次に、Graph Inspector を使用して **Surface Type** を設定します。



Surface Type の設定

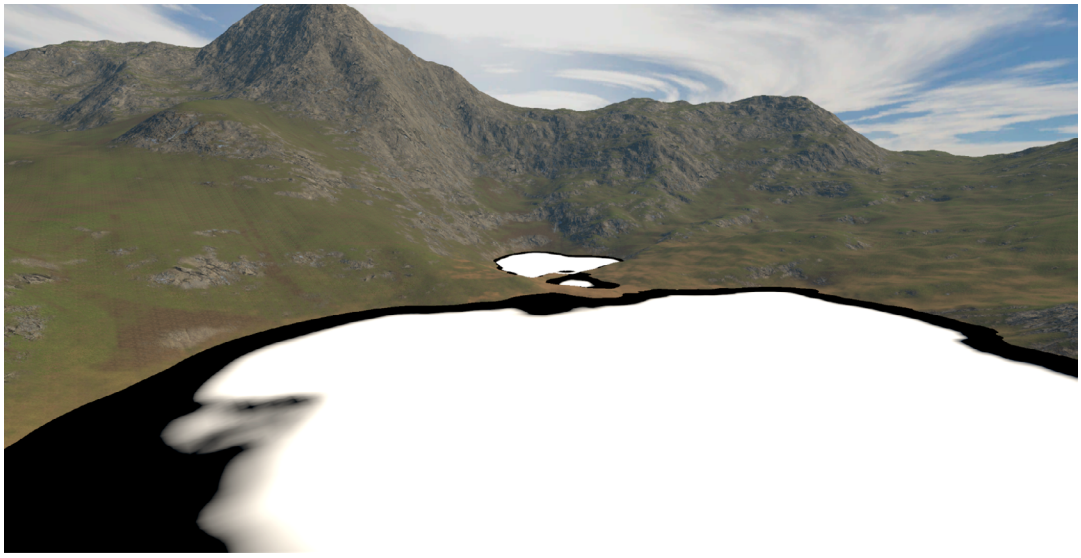
ここで、グラフを編集します。まずは色から始めます。

色



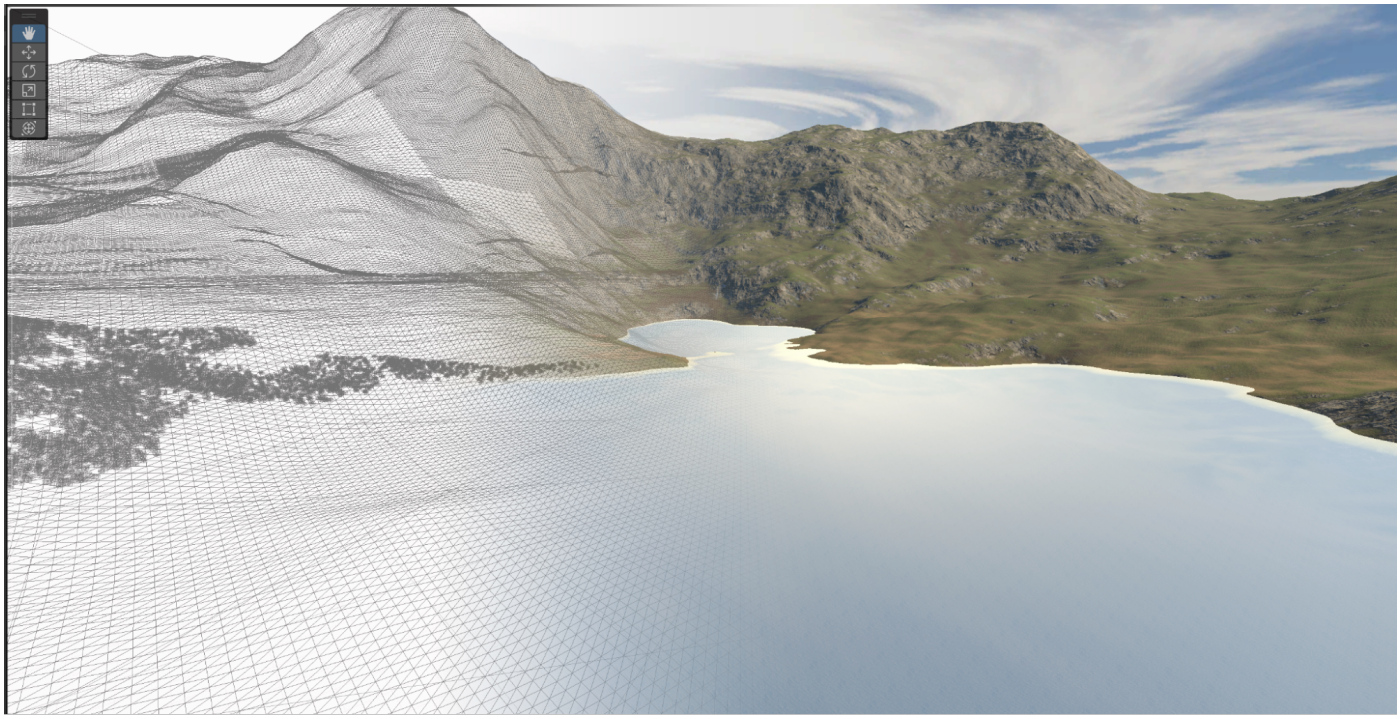
色とアルファ

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、Float 型の **Depth** プロパティを使用して制御を行います。出力が Fragment シェーダーの Base Color 入力に直接つながっている場合、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 の出力を直接 **Fragment > Base Color** に接続

DepthFade を Base Color の入力に直接つなげずに、Lerp ノードにつなげます。**ShallowWaterColor** は黒の代わりに入力 A を、**DeepWaterColor** は白の代わりに入力 B を指定します。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** につなげます。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。

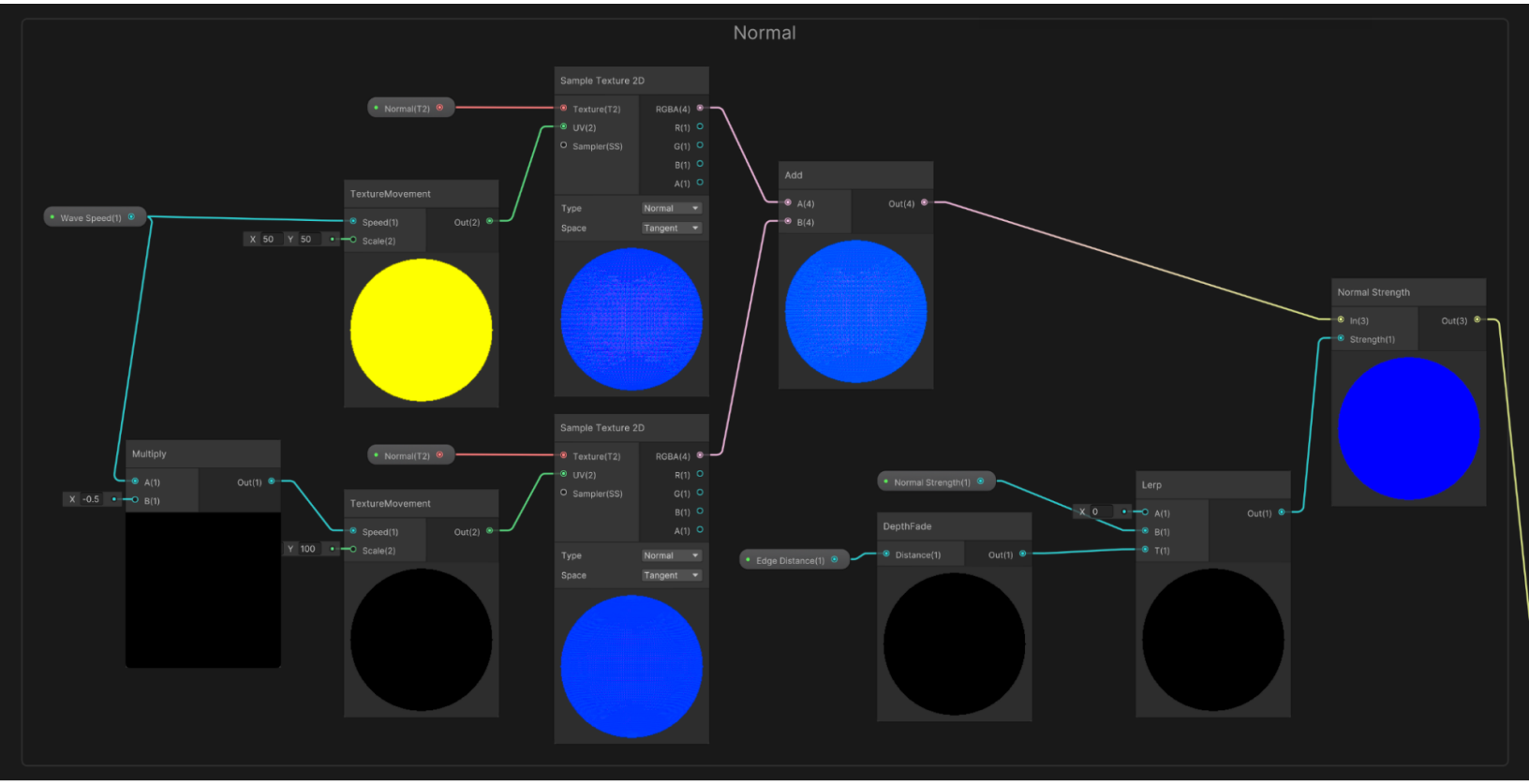


密度が高いメッシュおよび色が設定された水

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

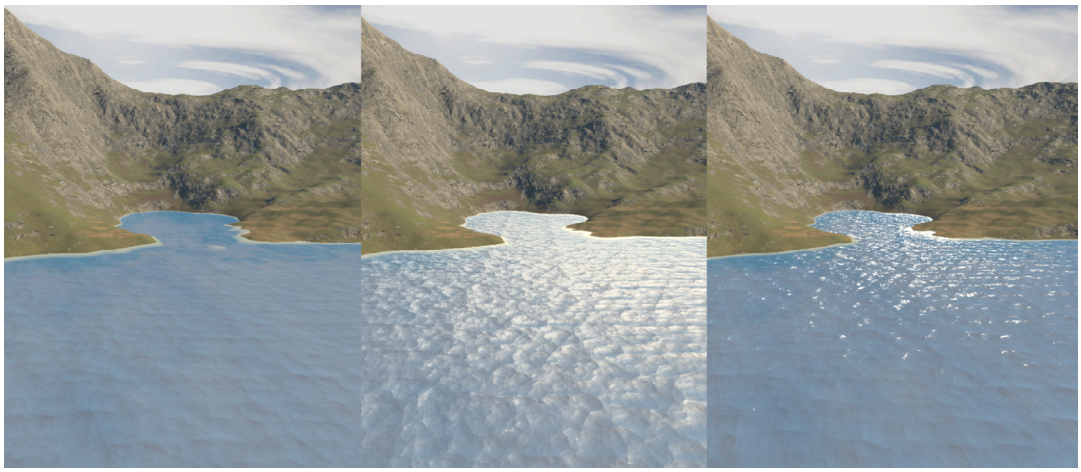


Fragment > Normal の制御

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** です。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 に固定されています。また、2 つ目の TextureMovement ノードでは、Multiply ノードによって **Speed** が Wave Speed プロパティの半分になるよう事前処理されます。

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 つの法線を加算することで、2 つの動くテクスチャを組み合わせた効果が得られます。このシェーダーには **Normal Strength** という Float プロパティがあり、これは 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。そこで、波の広がりを制御する **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。

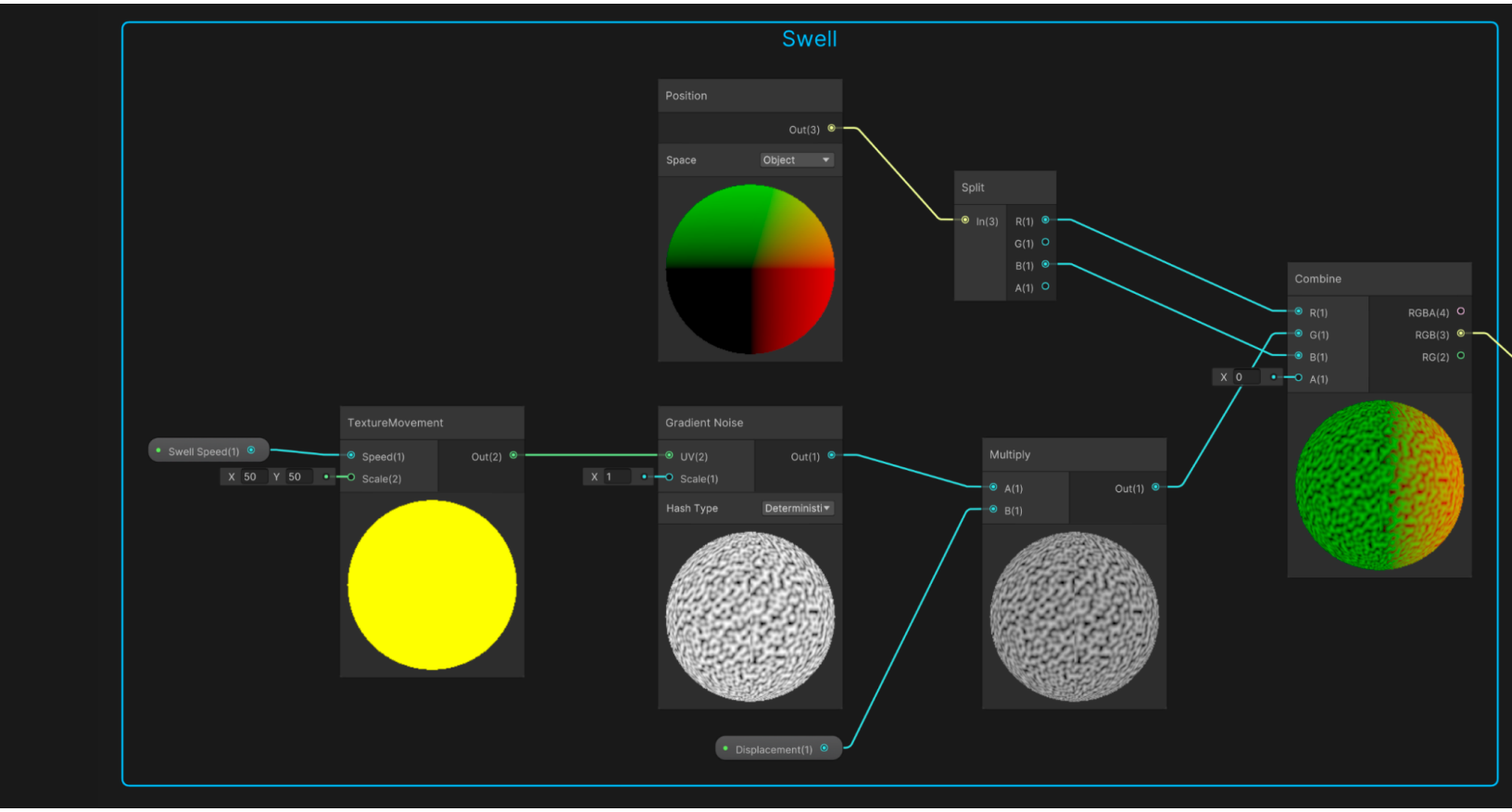
これで、制御可能なさざ波ができました。リフレクションに関するプロパティは、シンプルな Float プロパティを使用して Fragment の **Smoothness** を制御することで調整できます。以下の画像は、Smoothness 値を変えた場合のエフェクトを示しています。



さざ波に適用する滑らかさのレベルの変化 (左から順に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。

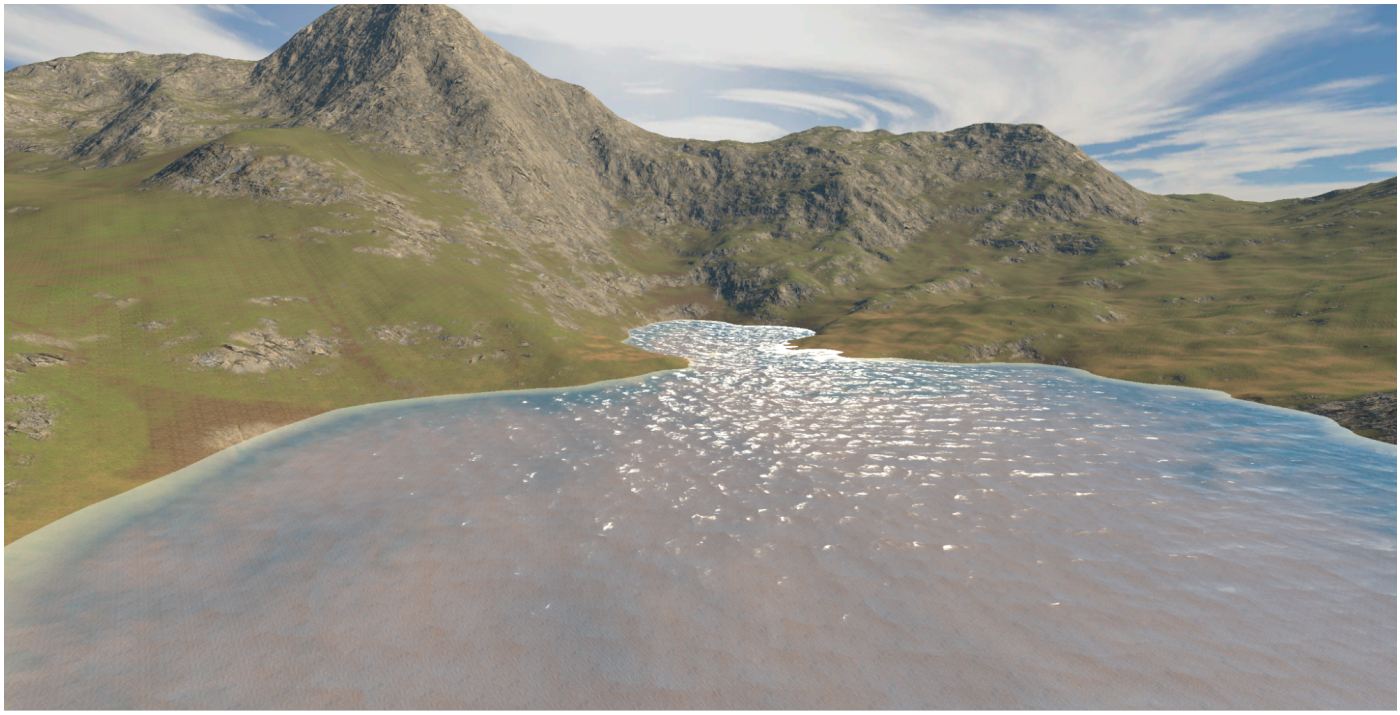
うねり



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 Float プロパティ **Swell Speed** を使用して設定され、Scale は 50,50 に固定されています。この出力は、Scale が 1 に固定された Gradient Noise ノードへの UV 入力として機能します。シェーダーの Float プロパティ **Displacement** を使用して、この値を Multiply ノードで制御します。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

このレシピはシンプルな水のシェーダーの基本を作成するものです。コースティクスリフレクション、屈折、泡を使用すると、効果をさらに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。

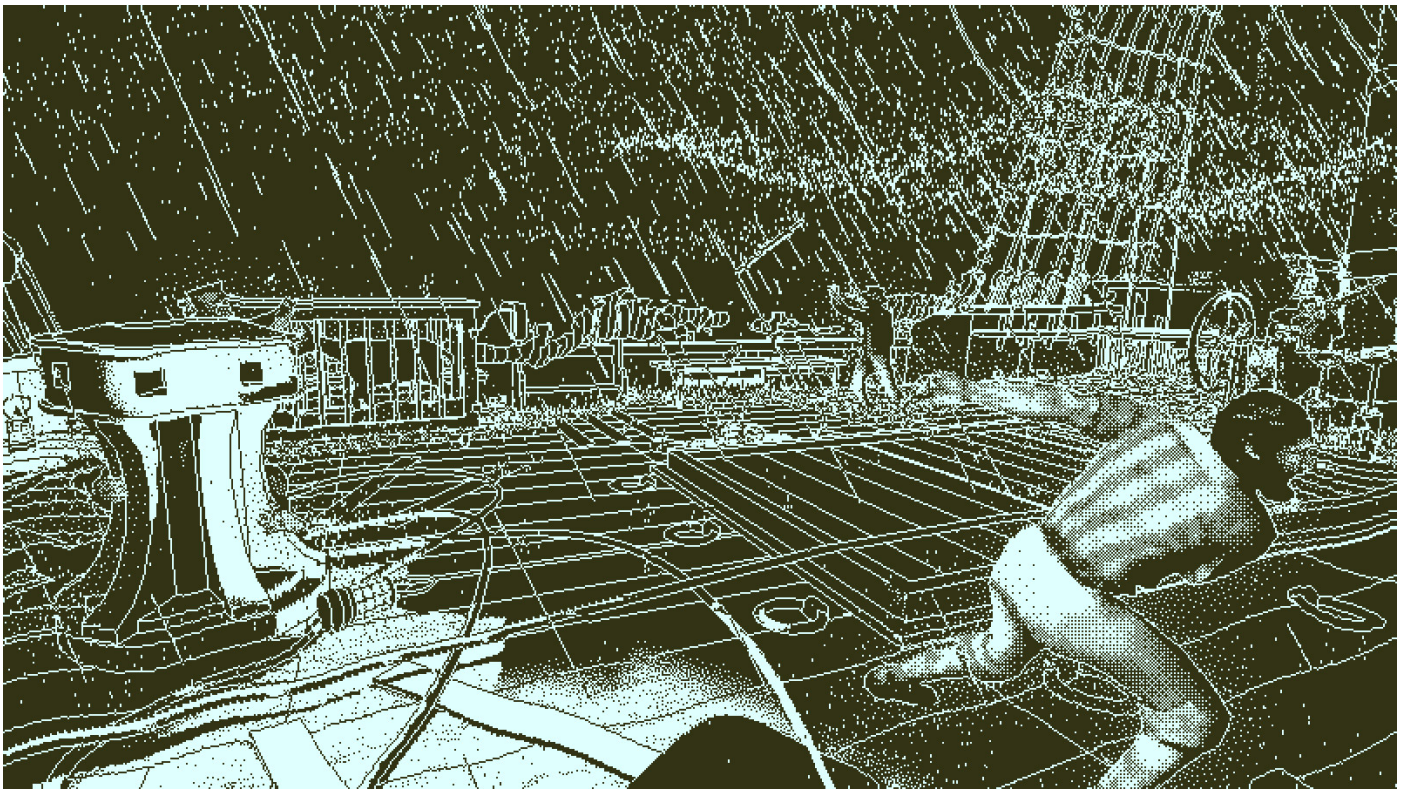
詳細情報

— Unity による YouTube チュートリアル

— Alan Zucconi 氏による コースティクス リフレクション のチュートリアル

— Binary Lunar 氏による 水の スタイル化 のチュートリアル

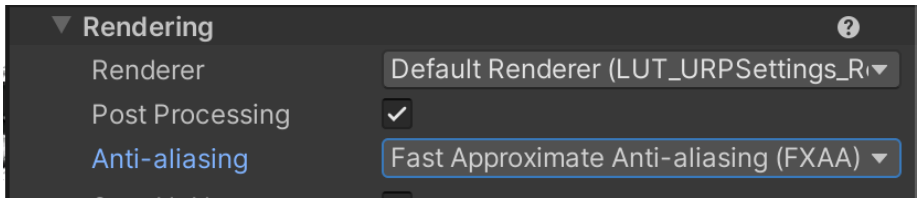
カラーグレーディング用のLUT



Lucas Pope 氏による Made with Unity のミステリーアドベンチャー FPS ゲーム『*Return of the Obra Dinn*』は、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、唯一無二のルックアンドフィールを実現しています。これは、このレシピに従うことで再現できます。

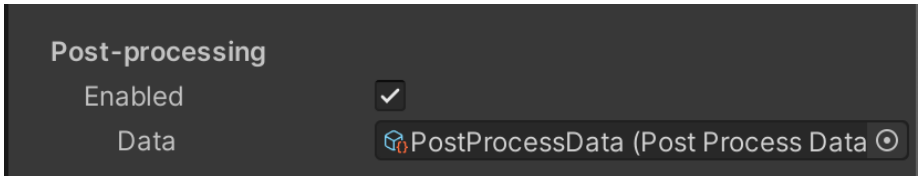
Color Lookup を使用してグレーディングエフェクトを生成する

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。新しく作成された 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** パネルから有効にします。



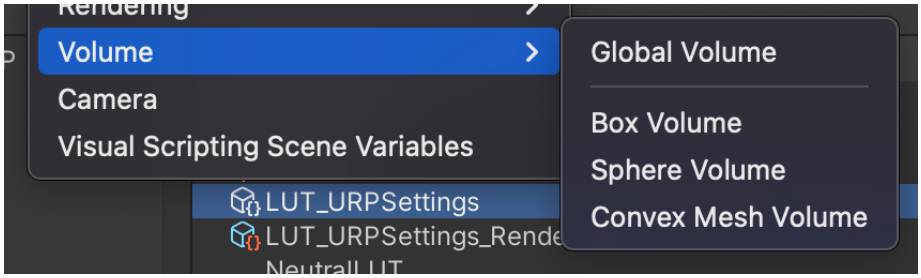
Camera > Rendering で Post Processing を選択

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。



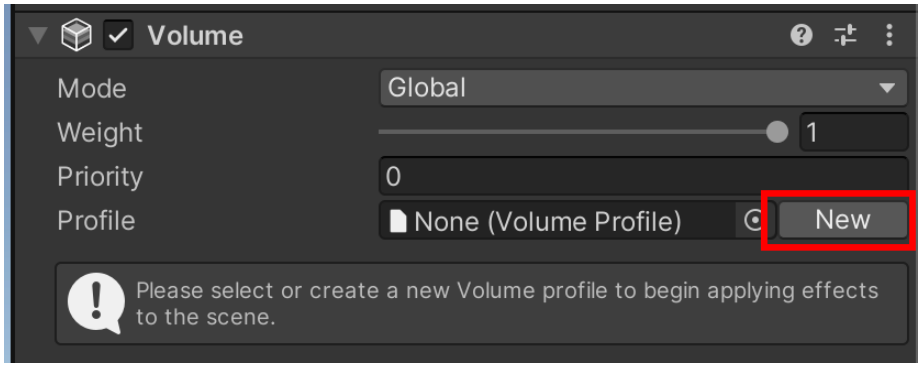
Universal Renderer のデータアセットで Post-processing を選択

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。



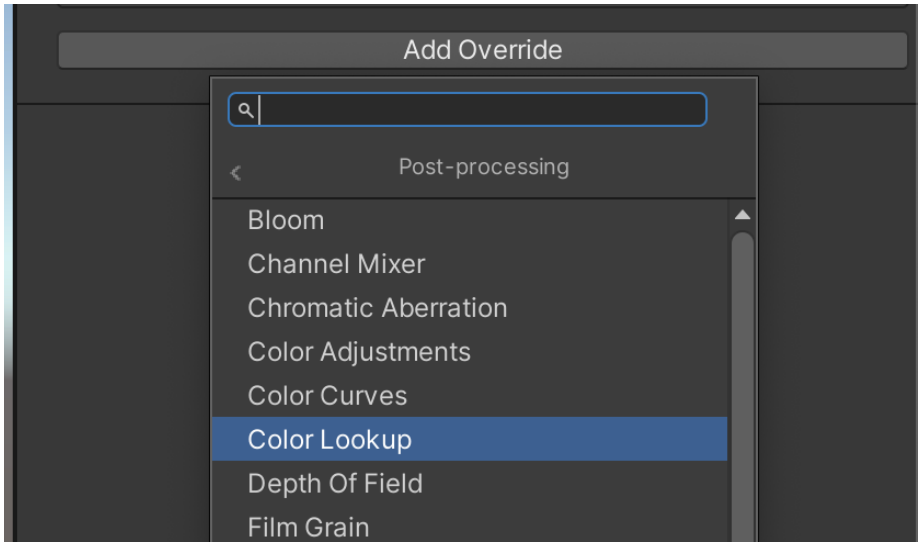
Global Volume の作成

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。

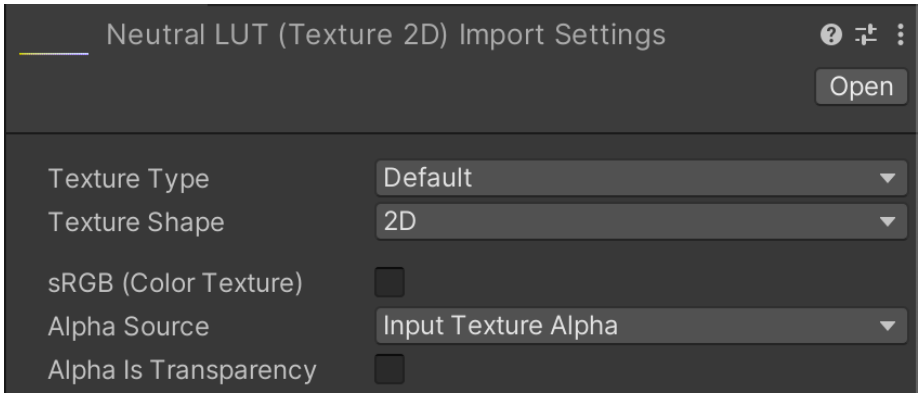


Color Lookup ポストプロセスフィルターの追加

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、こちらの リンク からダウンロードできます。

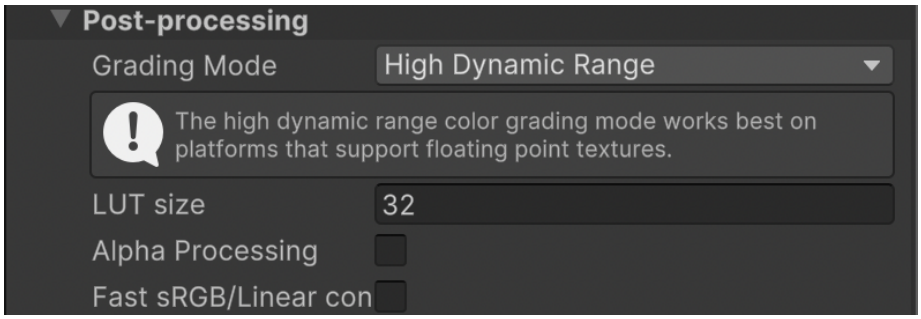
NeutralLUT.png

LUT 画像の **sRGB (Color Texture)** は無効にする必要があります。これは、画像を選択し Inspector で表示して行うことができます。



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

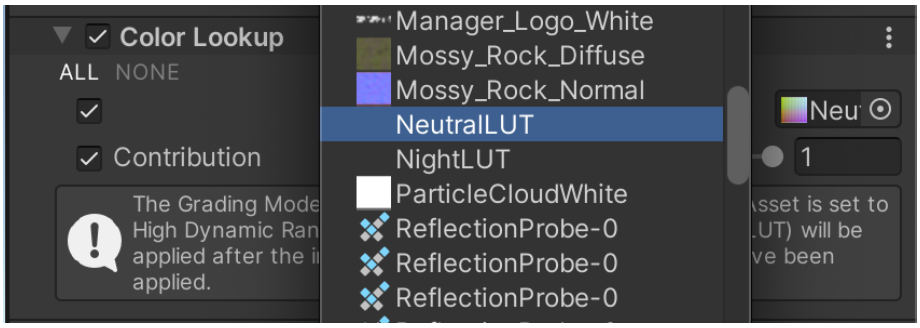
前の NeutralLUT 画像のブロックを数えてください。32 個あることがわかります。または、16 個のブロックを使用できます。ブロック数として 32 個または 16 個どちらを選択するとしても、URP アセットの設定が選択と一致するようにします。32 を選択した場合は、Post-processing パネルの **LUT size** が **32** に設定されていることを確認します。**Grading Mode** オプションを使用して自由に試してみてください。



LUT size の設定

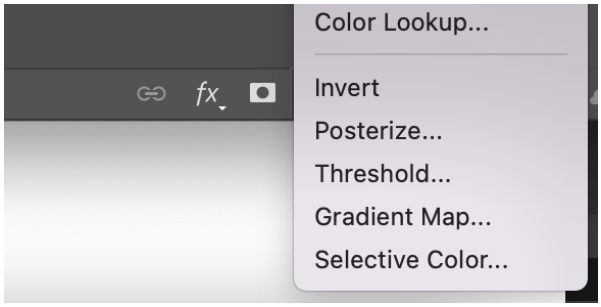
Color Lookup 設定パネルで **NeutralLUT.png** をルックアップテクスチャとして割り当てても、レンダリングされた画像に変化はありません。このフィルターはテクスチャを使用して新しい色を設定します。コードが、現在のピクセルカラーを取得し、それを使用して LUT 画像上のテクセルを探します。ニュートラル LUT 画像では、テクセルカラーは現在のピクセルカラーと同じになります。

本当の効果が現れるのは、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) で加工したときです (このセクションの最後の "その他のリソース" には、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)。



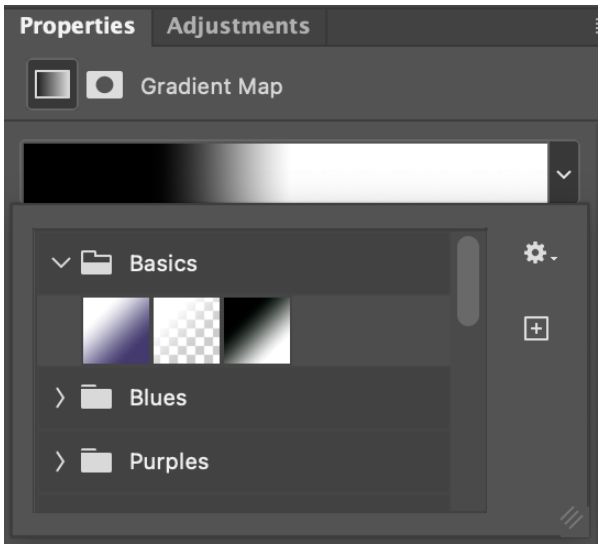
ルックアップテクスチャの割り当て

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。このボタンを選択して、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。



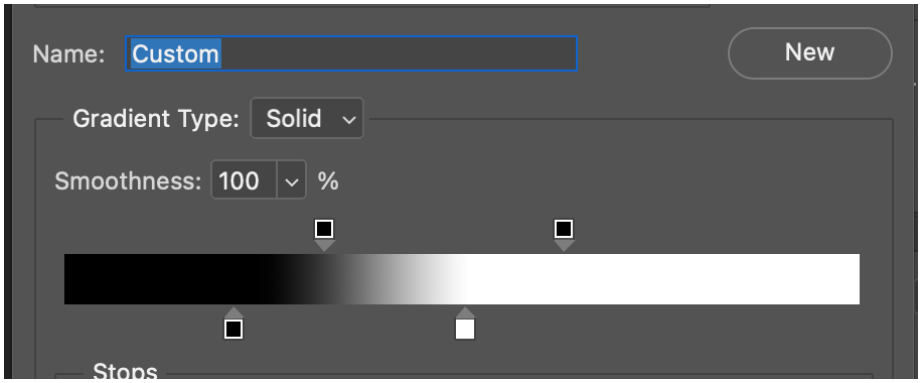
色調整レイヤーの作成

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。

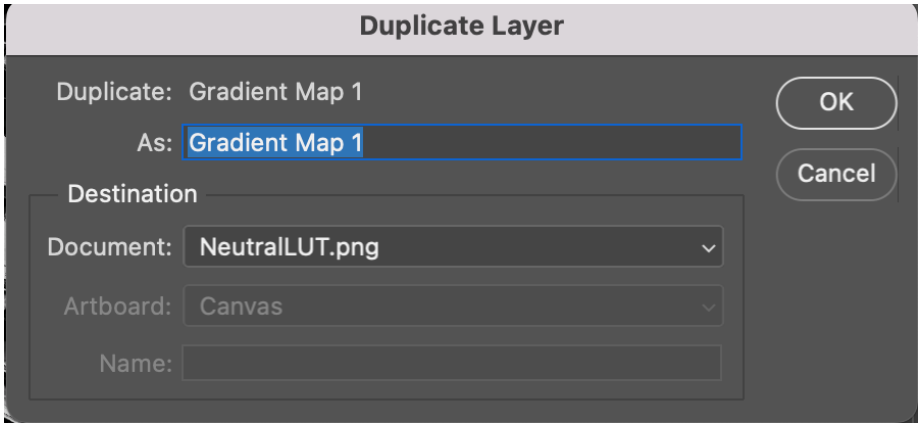


コントラストを強くするためにストップを変更

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り、調整レイヤーを右クリックして **Duplicate Layer** を選択します。新しいパネルで、**Destination > Document** に NeutralLUT.png を指定します。

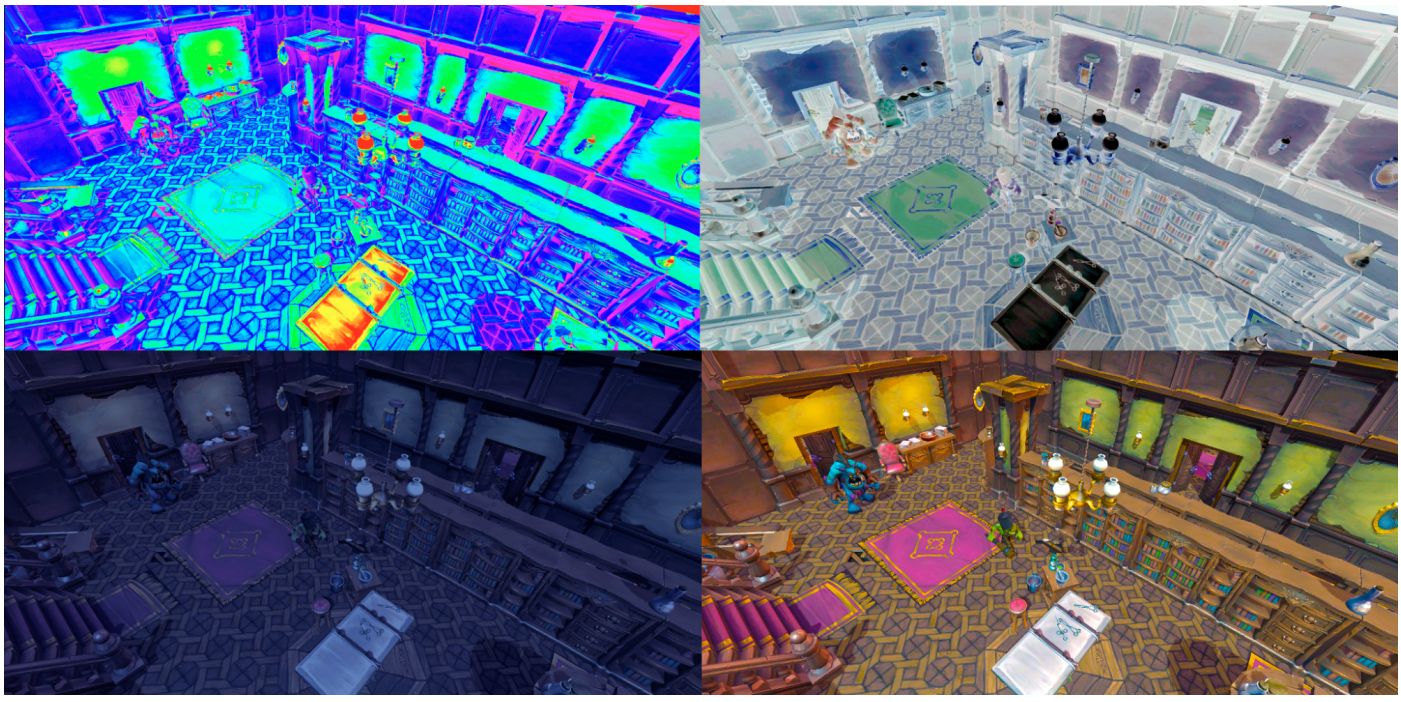


調整レイヤーの複製

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 パネルで必ず 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

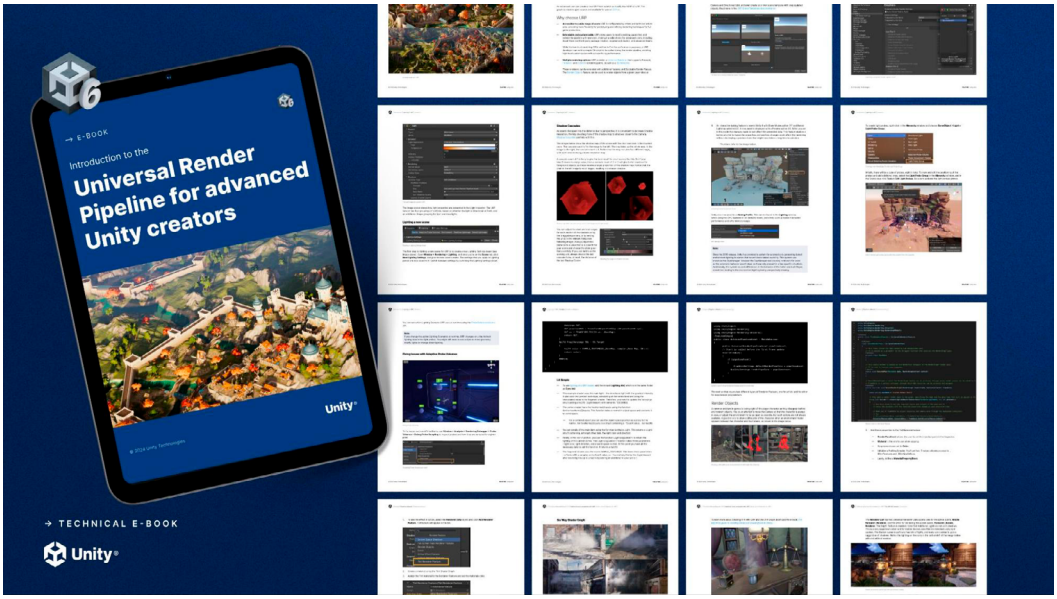
— PHLEARN による YouTube チュートリアル

— GDQuest による YouTube チュートリアル

アダプティブプローブボリューム

Clockstone が Unity と URP を使用して制作したゲーム『*LEGO® Bricktales* 』では、プレイヤーはレゴの世界に没入できます。ブロックのリアルな質感を生み出す上で、優れたライティングが大きな役割を果たしています。

アダプティブプローブボリューム (APV) は、混合モードライティング用の最新の Unity ソリューションであり、ライトプローブよりも設定とメンテナンスが簡単です。URP で利用可能なライティング手法に不慣れの場合は、Unity の e-book『上級 Unity クリエイター 向けの ユニバーサル レンダー パイプライン （ URP ） 入門』を参照してください。



ダウンロード

ライトの混合モード設定を使用すると、ベイクされたオブジェクトと動的オブジェクトを組み合わせることができます。混合モードを使用する際は、シーンにもプローブを加えることが推奨されます。Unity 6 にはライトプローブと APV の 2 つのオプションがあります。2 つのオプションはどちらも、動的オブジェクトがシーン内を移動し、グローバルイルミネーションの影響を受けることを可能にするものです。しかし、APV は従来のライトプローブよりも優れた利点と設定効率をもたらします。

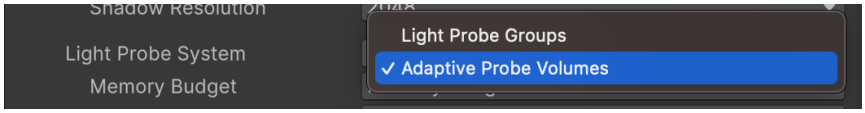
プローブは、シーン内の単なるポイントです。デザイン時に、このポイントでのグローバルイルミネーションが計算されます。実行時にフレームをレンダリングする際、ライティング計算を含む URP シェーダーは、グローバルイルミネーションの値として最も近いプローブをブレンドして使用します。

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

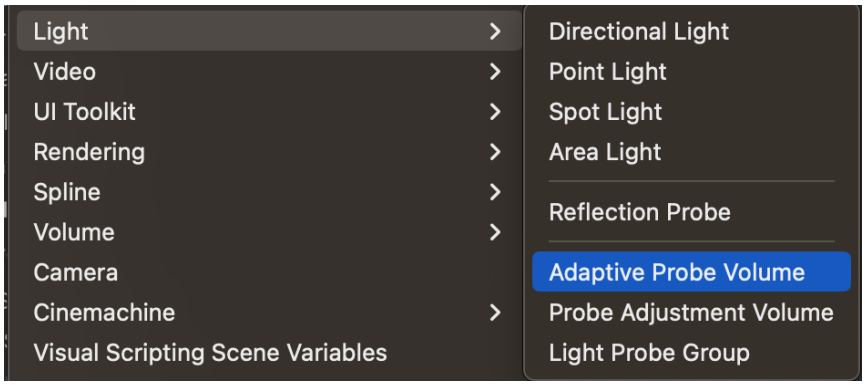
シーンで APV を使用する

シーンのライトプローブを慎重に配置したものの、シーンのレイアウトが変更された経験のあるテクニカルアーティストであれば、APV の利点はすぐに理解できるでしょう。なぜなら、多くのシーンでは APV を使用することで、すべてのプローブを数秒で配置できるためです。もう一度 FPS サンプル"The Inspection" を使用して実用的な例を見てみましょう。

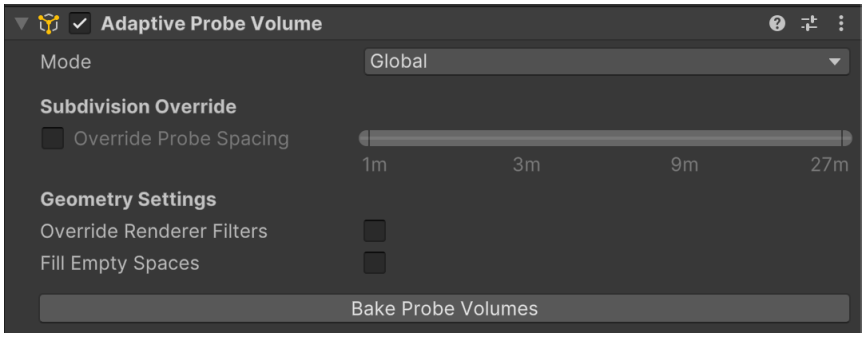
1. まず、アクティブな URP アセットで **Light Probe System** オプションが **Adaptive Probe Volumes** に設定されていることを確認します。



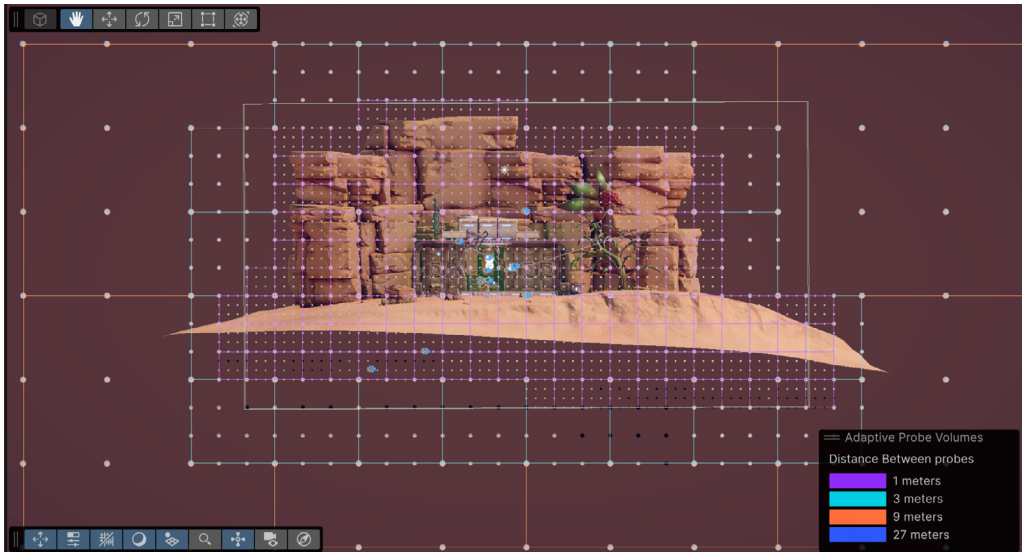
2. Hierarchy ウィンドウで右クリックして、**GameObject > Light > Adaptive Probe Volume**（APV）を選択します。



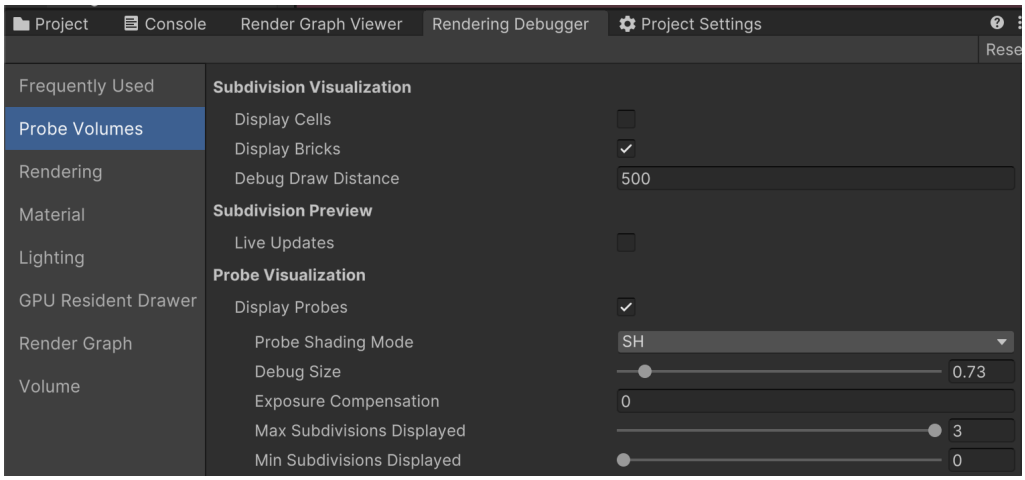
3. モードを **Global** に設定し、デフォルト設定 (1、3、9、27 メートルのサブディビジョン) を使用します。



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



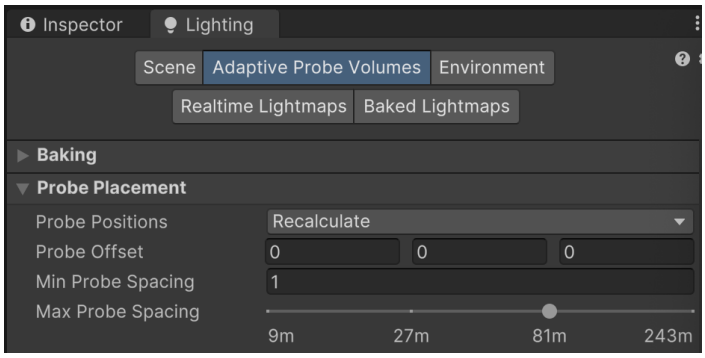
5. ベイクの結果を表示するには、**Analysis > Rendering Debugger** を開きます。**Probe Volumes** を選択し、**Display Probes** を選択します。異なる解像度を表示するには、**Display Bricks** を選択します。



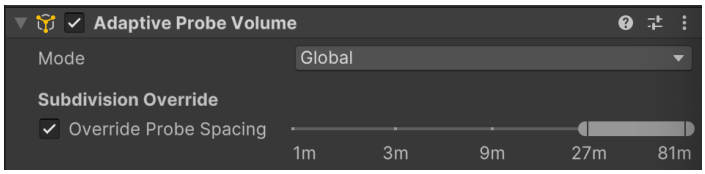
多くのシーンではこれで作業が完了し、休憩できるでしょう。しかし、APV を使うとさらに高い忠実度が得られます。異なるサブディビジョン を持つ複数のボリュームを追加して、プローブの配置と密度を正確にコントロールできます。

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

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



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



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

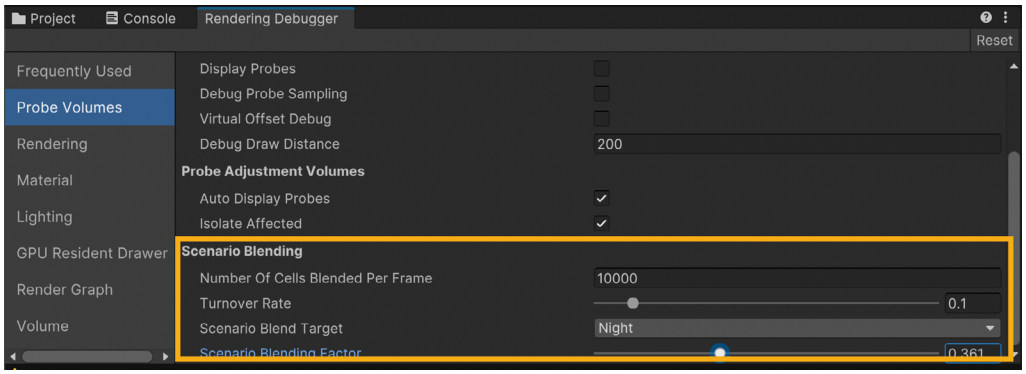
4. プローブボリュームをベイクします。

下の画像からわかるように、ほとんどのプローブはテントの周囲にあります。

プローブの配置

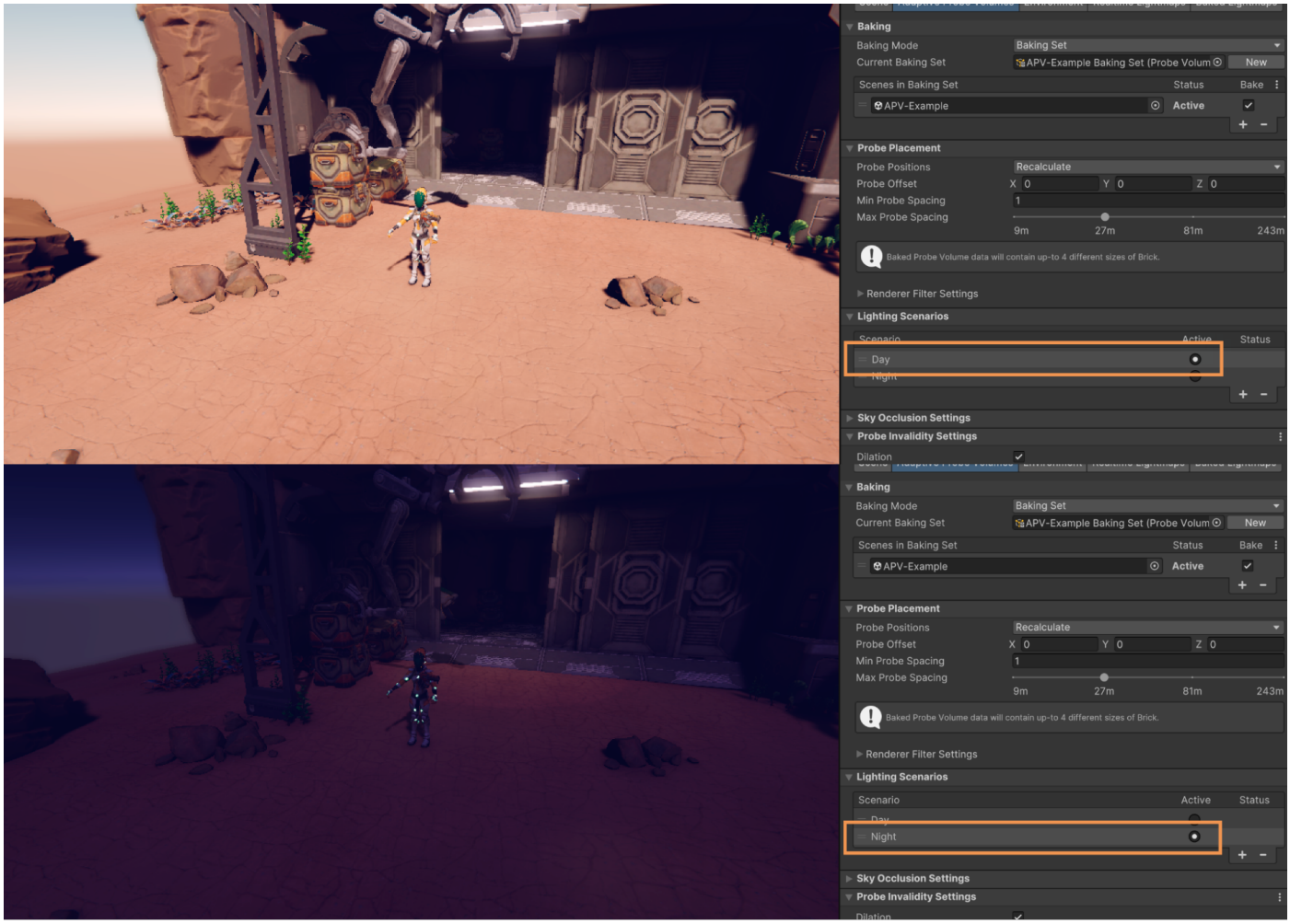
ライティングシナリオアセット

APV のもう 1 つの機能は、間接光データの切り替えです。**ライティングシナリオ**アセットには、シーンまたは**ベイク セット**のベイクされたライティングデータが含まれています。異なるライティング設定をそれぞれ別のライティングシナリオにベイクし、ランタイムまたはデザイン時にレンダリングデバッガーを使用して、URP が使用するシナリオを変更できます。



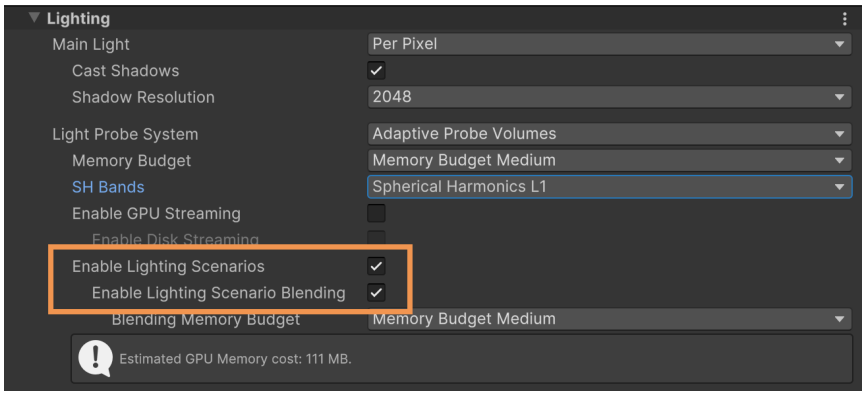
レンダリングデバッガーを使用したシナリオのブレンド

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



昼夜のライティングシナリオ

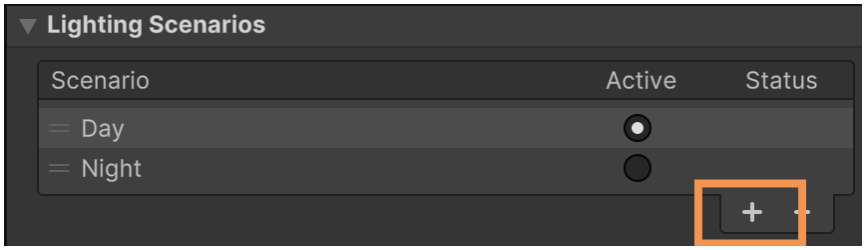
1. ライティングシナリオアセットを使用するには、アクティブな URP アセットに移動し、**Lighting > Light Probe Lighting > Lighting Scenarios** を有効にします。



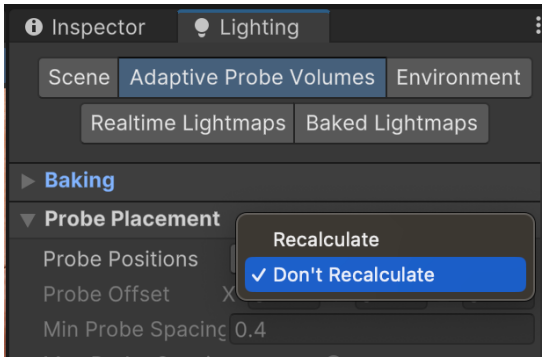
2. ベイク結果を保存できる新しいライティングシナリオアセットを作成するには、以下の手順に従います。

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

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



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

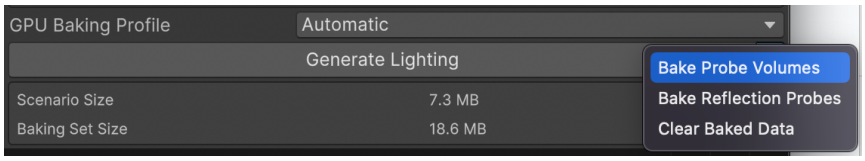


4. ライティングシナリオにベイクするには、以下の手順に従います。

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

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

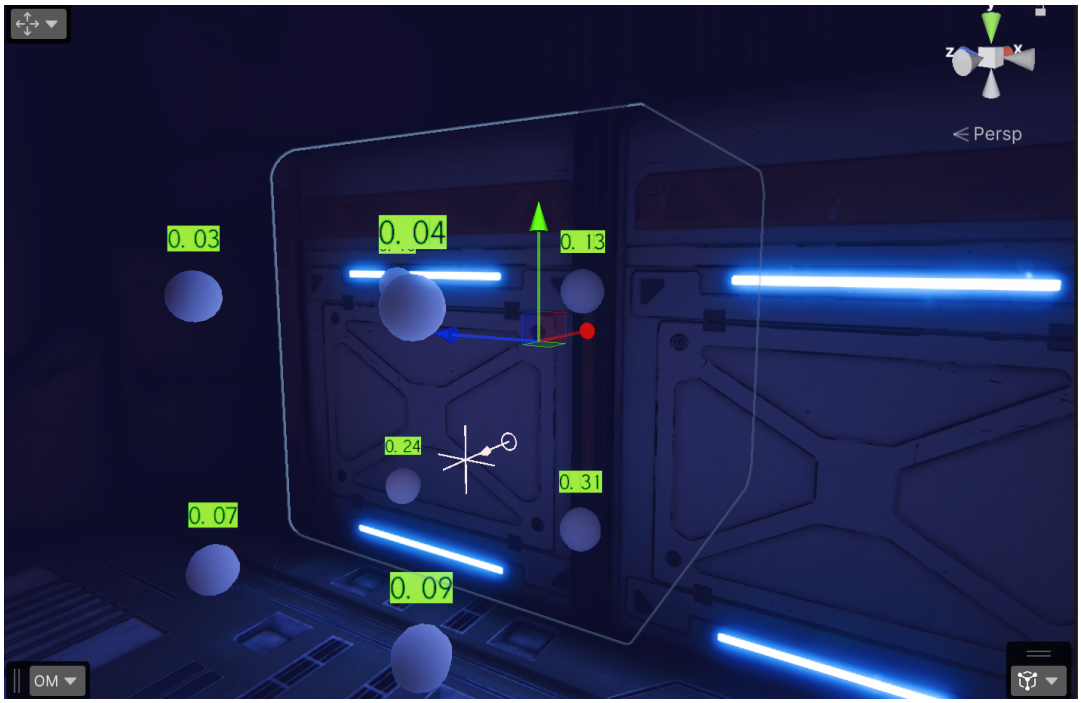
   c. ライトマップを使用していない場合は、**Generate Lighting** の横にあるドロップダウンボタンを使用して、プローブのみに焦点を当てます。



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

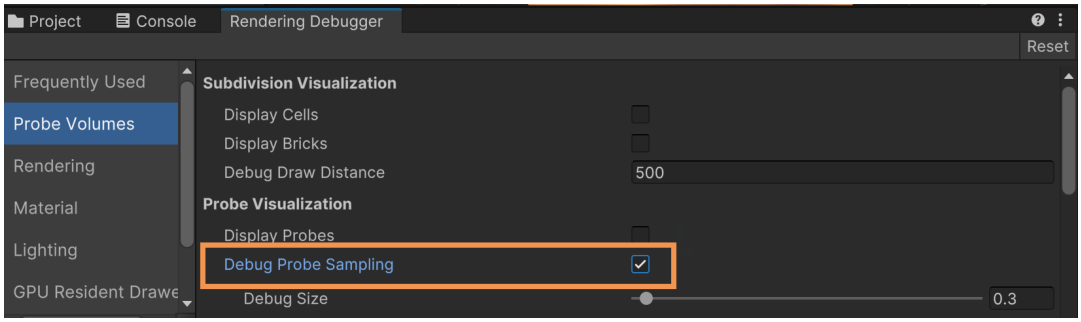
注：
実行時にアクティブなライティングシナリオを変更すると、URP はライトプローブ内の間接光データのみを変更します。ジオメトリの移動、ライトの修正、または直接光の変更には、スクリプトを使用する必要がある場合があります。

APV の問題を修正する



プローブサンプリングのデバッグ

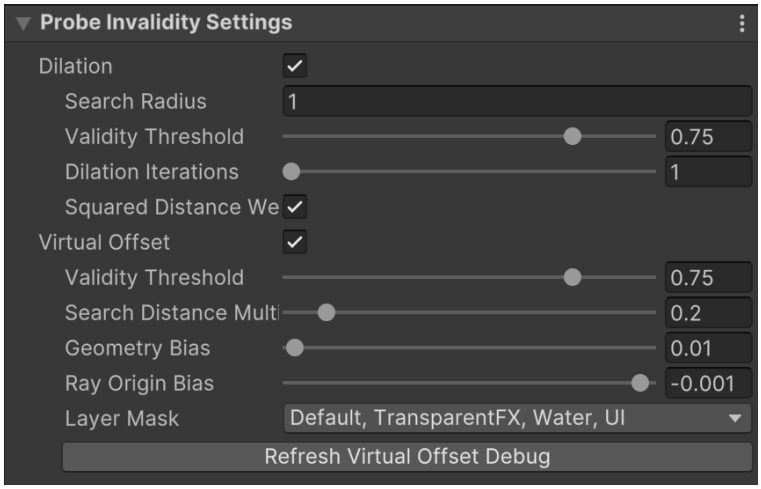
APV のアーティファクトなどの問題を修正するには、**Window > Analysis > Rendering Debugger > Probe Volumes > Debug Probe Sampling** を使用し、プローブが特定のピクセルでどのようにサンプリングされているかを確認してください。



ピクセルごとのプローブサンプリングの視覚化

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

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



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

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

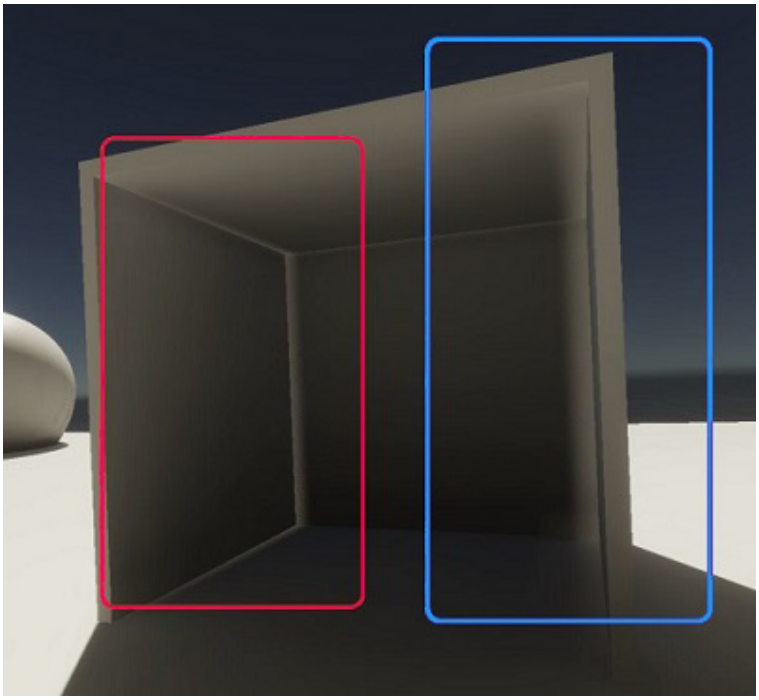
無効なライトプローブの確認には、レンダリングデバッガーを使用できます。

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

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

ライトリーク

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



ライトリーク

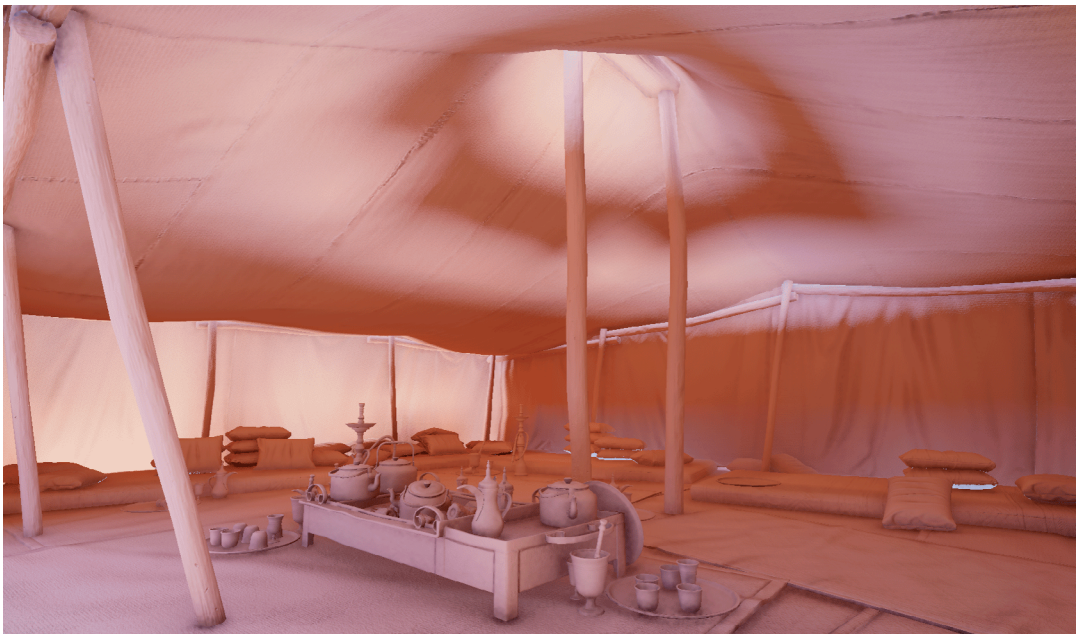
ライトリークは、通常ジオメトリが自身には見えていないライトプローブからのライトを受け取ったときに発生します。例えば、ライトプローブが壁の反対側にある場合などです。APV はライトプローブを規則的なグリッドに配置するため、プローブが壁に沿わなかったり、異なるライティングエリアの境界に配置されないことがあります。

ライトリークを修正するには、以下の方法を試してください。

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

レンダリングレイヤー

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



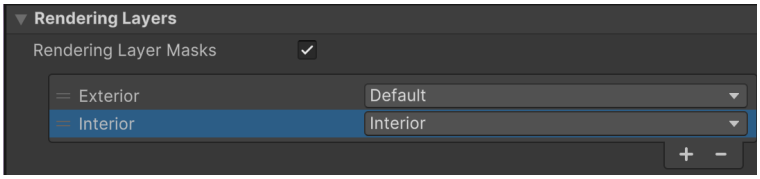
URP 3D サンプルのオアシス環境のテント内で光が漏れている様子

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

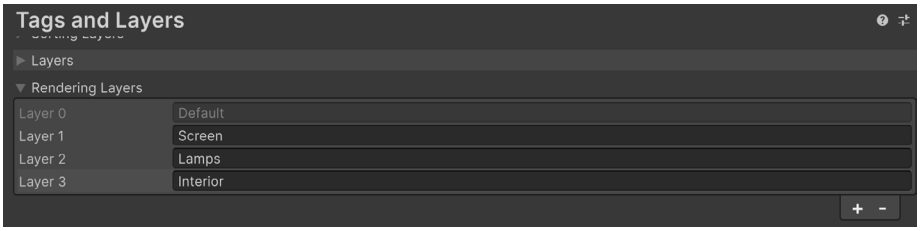
ピクセルの補間プローブの表示

この問題を修正する 1 つの方法は、ボリュームを使用して、**Adaptive Probe Volume Options** オーバーライドを介して、ランタイム時に APV のサンプリング方法を変更することです。**NormalBias** と **ViewBias** の設定は、サンプリング位置の調整に役立ちます。Normal Bias は法線に沿って (壁から離れる方向に) プッシュし、View Bias はカメラと同じ側に位置を保ちながら、カメラの方向にプッシュします。ボリューム内のこれらのプロパティを変更すると、ライティング結果と **Debug Probe Sampling View** の両方でリアルタイムに更新が確認でき、サンプリング位置とウェイトがそれに応じて更新されます。しかし、より良い解決策はレンダリングレイヤーを使用することです。

APV はレンダリングレイヤーをサポートしています。そのため、異なるマスクを 4 つまで作成でき、特定のオブジェクトに対して、そのマスクに限定したサンプリングを行うことができます。これにより、内部のオブジェクトが外部のプローブをサンプリングするのを防ぐことができ、またその逆も防止できます。**Window > Rendering > Lighting > Adaptive Probe Volumes > Rendering Layers** でレンダリングレイヤーを有効にして追加できます。



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

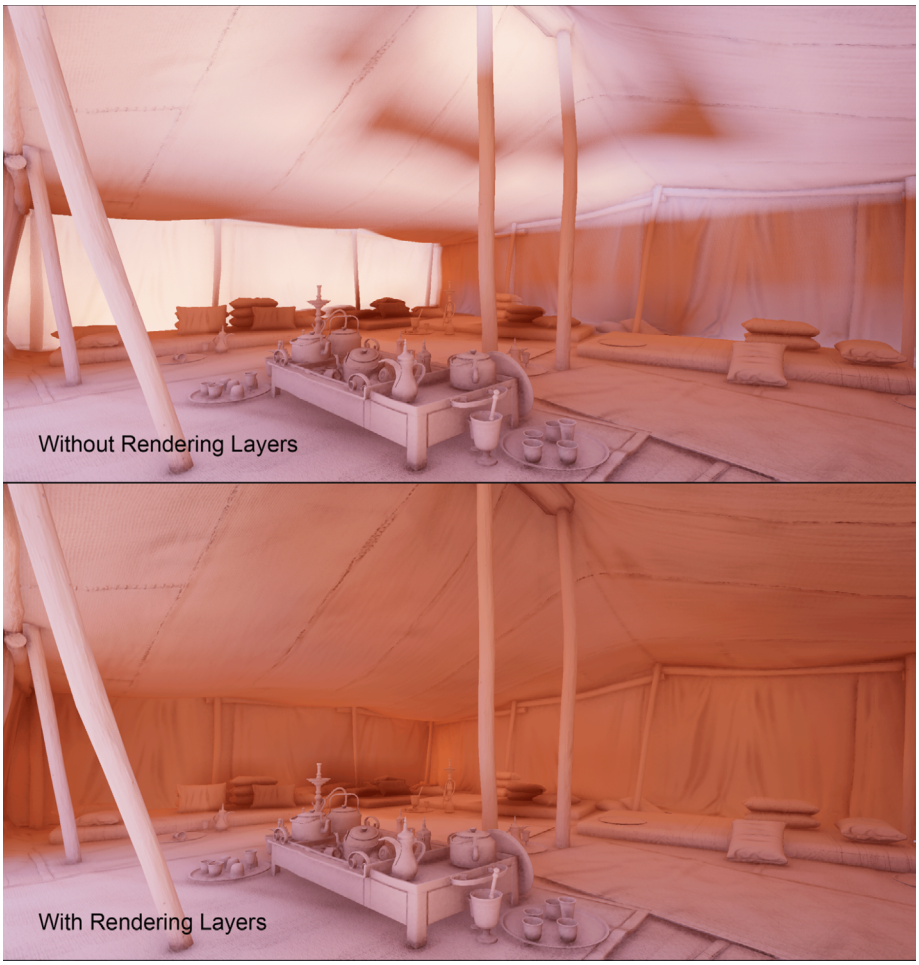


これを実装するには、メッシュ自体を編集して、作成したいエリアごとに分割されていることを確認します。例えばこのプロジェクトでは、メッシュの編集によって内部と外部を複数のメッシュに分割しています。メッシュを分割したら、適切なレンダリングレイヤーを割り当て、**Adaptive Probe Volumes タブ**で APV で使用するものを指定します。

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

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

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



レンダリングレイヤーの有無によるライトリークの差

APV に関する問題の修正について、詳しくは こちら を参照してください。

APV のストリーミング

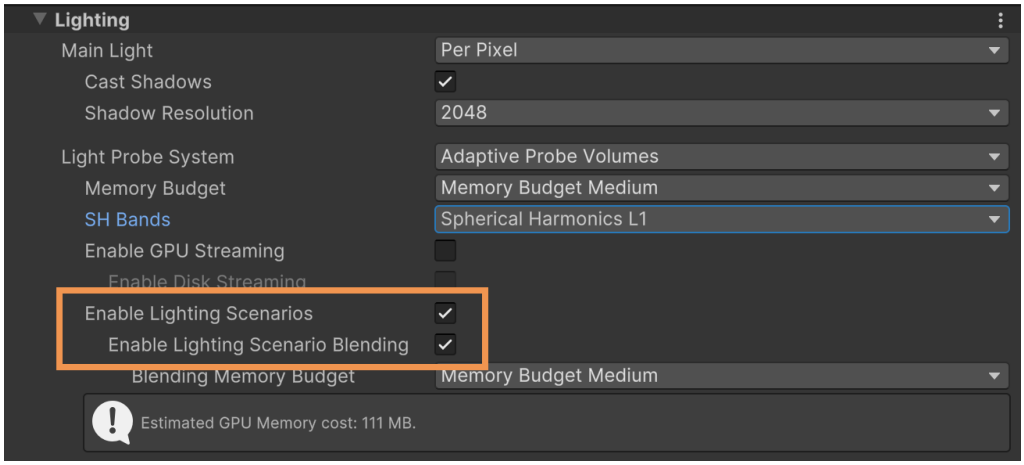
APV ストリーミングを使用すると、大規模なワールドで APV ベースのライティングを利用できます。APV ストリーミングは、CPU または GPU のメモリ容量を超える APV データをベイクし、実行時に必要に応じてロードします。実行時にカメラが移動すると、URP はカメラの視錐台内のセルからのみ APV データをロードします。

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

1. メインメニューから **Edit > Project Settings > Quality** を選択します。
2. 品質レベルを選択します。
3. Render Pipeline Asset をダブルクリックして、Inspector で開きます。
4. Lighting タブを展開します。
5. ここで、以下の 2 種類のストリーミングを有効にできます。

    a. Disk Streaming を有効にして、ディスクから CPU メモリにストリーミング。

    b. GPU Streaming を有効にして、CPU メモリから GPU メモリにストリーミング。先に Disk Streaming を有効にする必要があります。

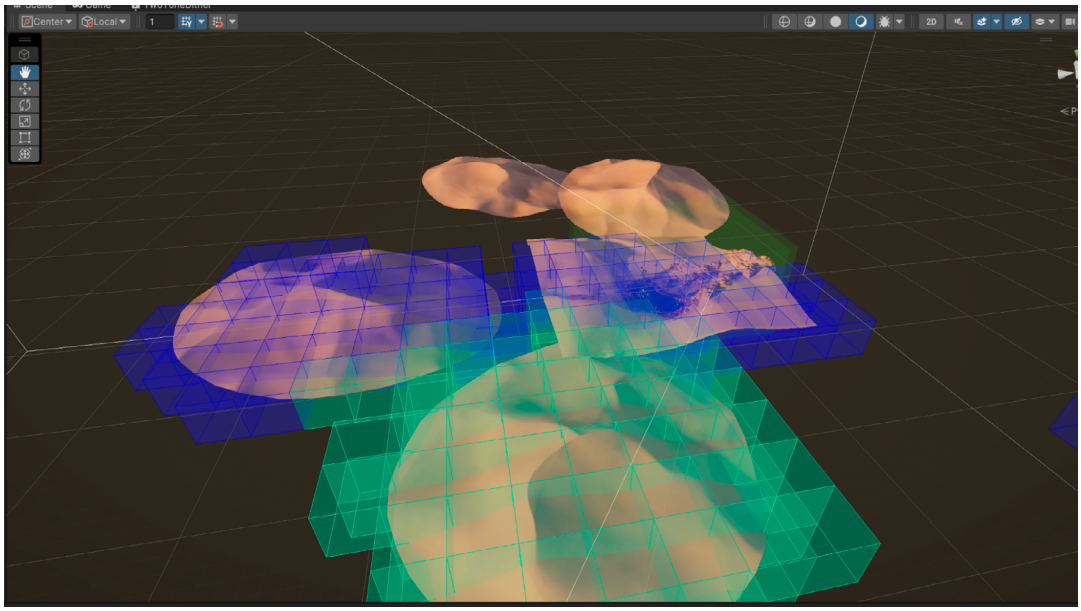


ストリーミング設定は同じウィンドウ内から可能です。詳細については、URP アセットを参照してください。

ストリーミングのデバッグ

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

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



APV ストリーミング

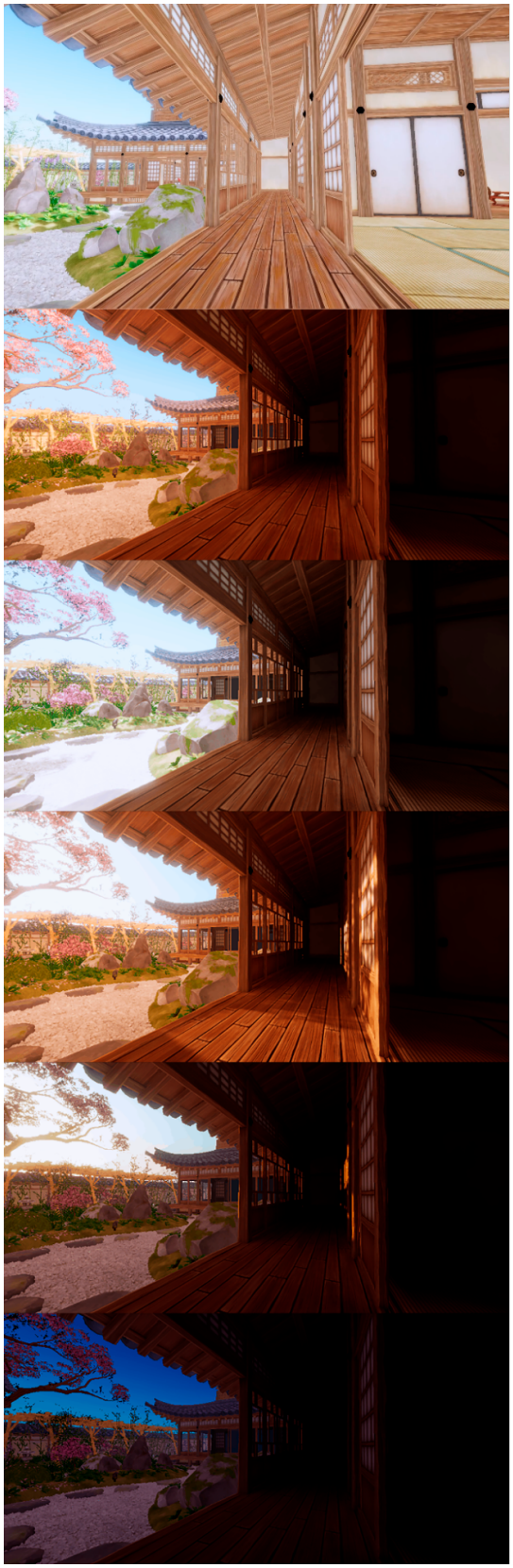
空のオクルージョン

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

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

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

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



これについては、左側の一連の画像を使用して説明します。

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

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

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

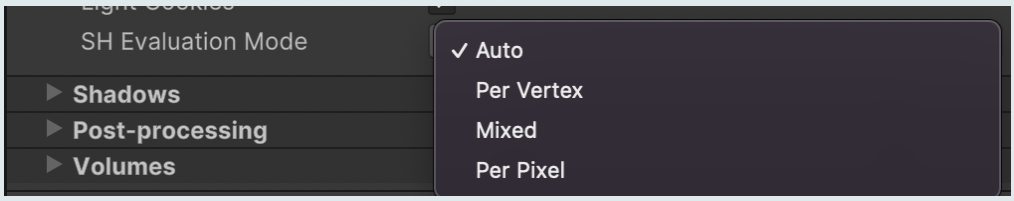
空のオクルージョンを有効にするには、以下の手順に従います。

1. GPU ライトベイカー (旧称: プログレッシブ GPU ライトマッパー) を有効にします。Progressive CPU を使用している場合、Unity は空のオクルージョンをサポートしません。**Window > Rendering > Lighting** に移動します。
2. Scene パネルに移動します。
3. Lightmapper を Progressive GPU に設定します。

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

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

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

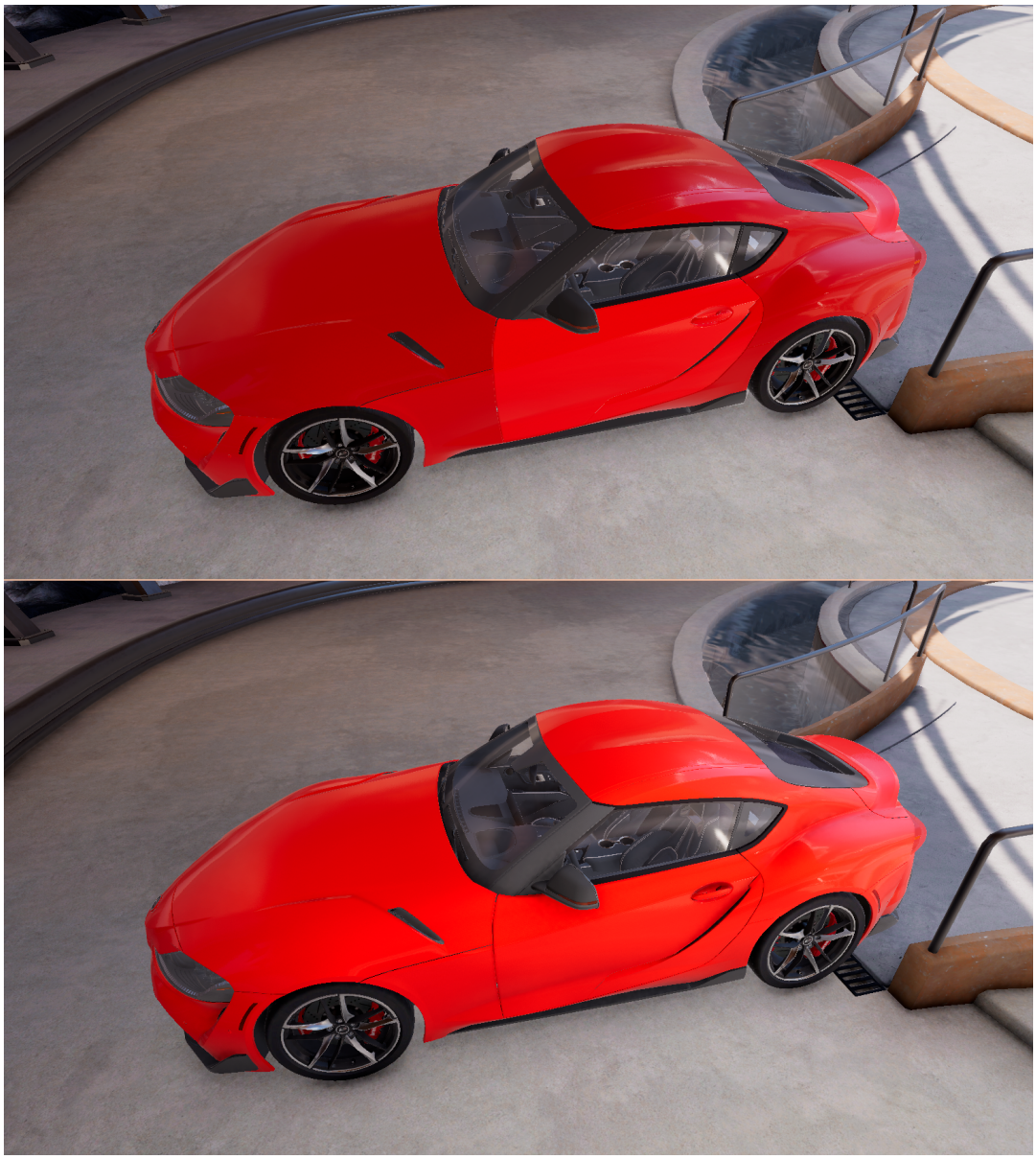


詳細情報

— アダプティブプローブボリュームの ドキュメント

— GDC 2023 のセッション: アダプティブ プローブ ボリューム を 用いた 効果的で インパクトのある ライティング

ライトプローブと APVの比較



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

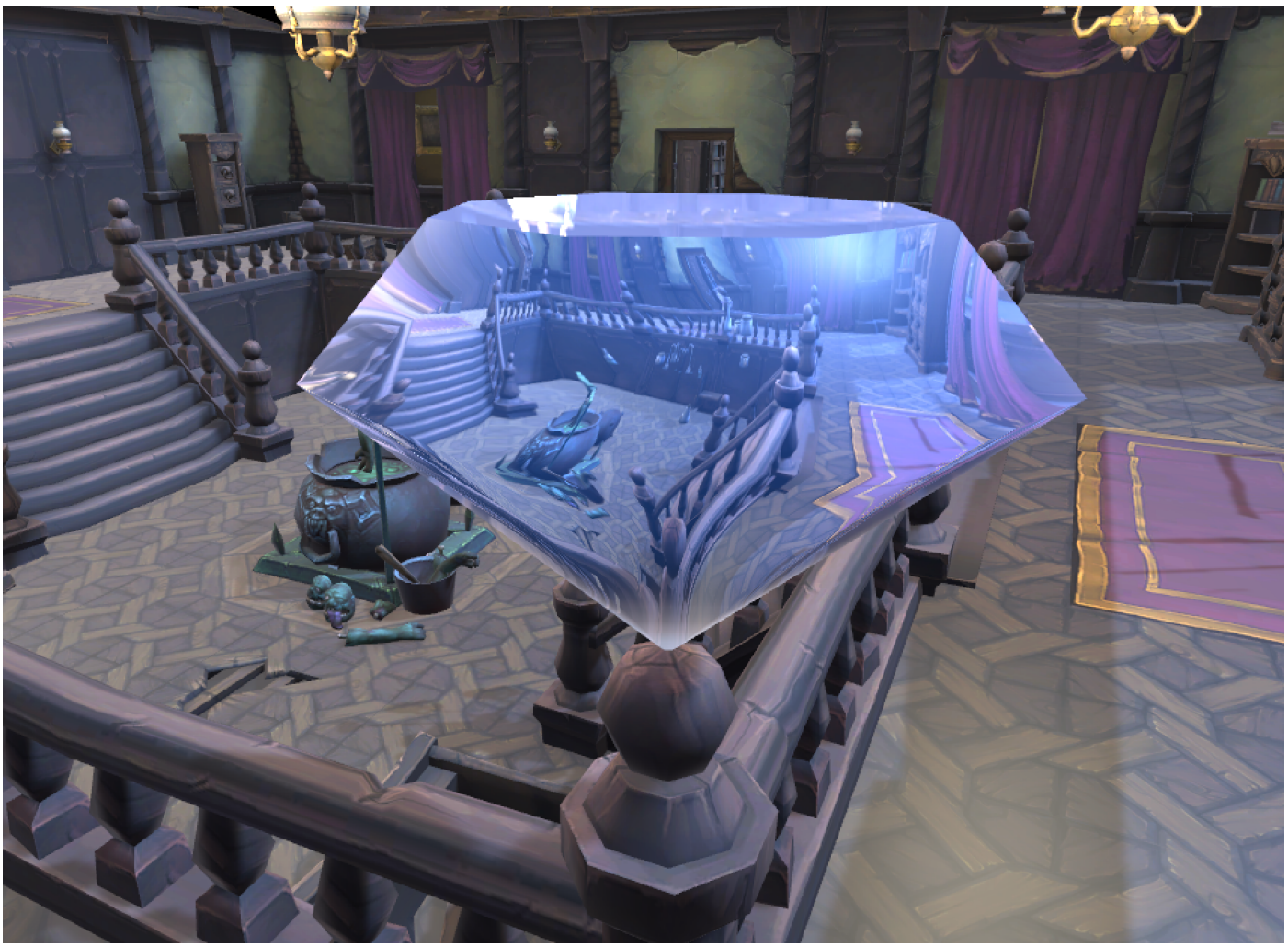
下の画像では、APV の使用によって暗い部分から明るい部分への遷移が滑らかになることが示されています。上の画像では、ライトプローブグループの影響で、オブジェクトごとに単一の補間プローブが使用されるため、車のドアに明るい光が当たっています。これは、ドアが他の部分とは別のゲームオブジェクトであり、異なるプローブを使用しているためにレンダリングエラーが発生しているからです。

以下の表は、ライトプローブと APV の機能を比較したものです。

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

スクリーンスペース屈折

開発者の Claudio 氏および Antonio 氏によるインディーゲーム『*Arctico*』では、プレイヤーはベースキャンプを設営し、氷河の地形を探検しなければなりません。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、この効果はスクリーンスペースリフレクションによって実現されます。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ます。一方、スクリーンスペース屈折は、透明度や、光が媒体を通過する際の屈折のシミュレーションに使用されます。



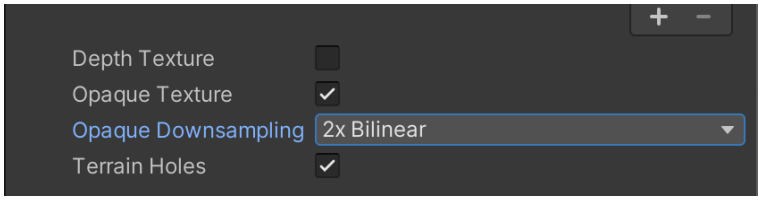
スクリーンスペース屈折の例

スクリーンスペース屈折は、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使用し、レンダリング対象の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。

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。上の画像で示されている追加の着色は、計算されたピクセルカラーと **Color** プロパティを線形補間することで実現されています。

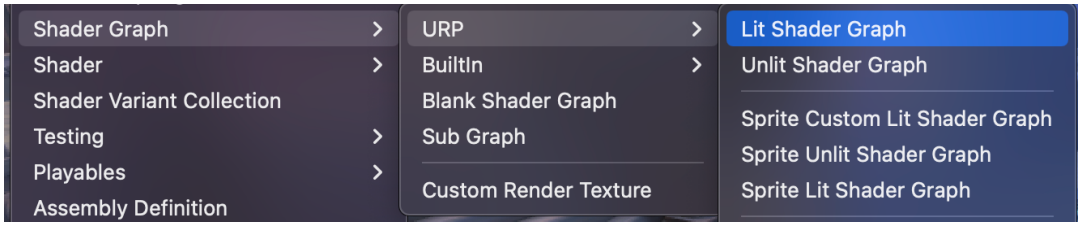
この効果の動作を表示するには、**Scenes > Refraction > Refraction** の順に移動します。

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** にあります。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** の順に選択します。



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するマテリアルを作成するには、Shader Graph アセットを選び、**Create > Material** の順に選択します。作成したマテリアルを、屈折さ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。**

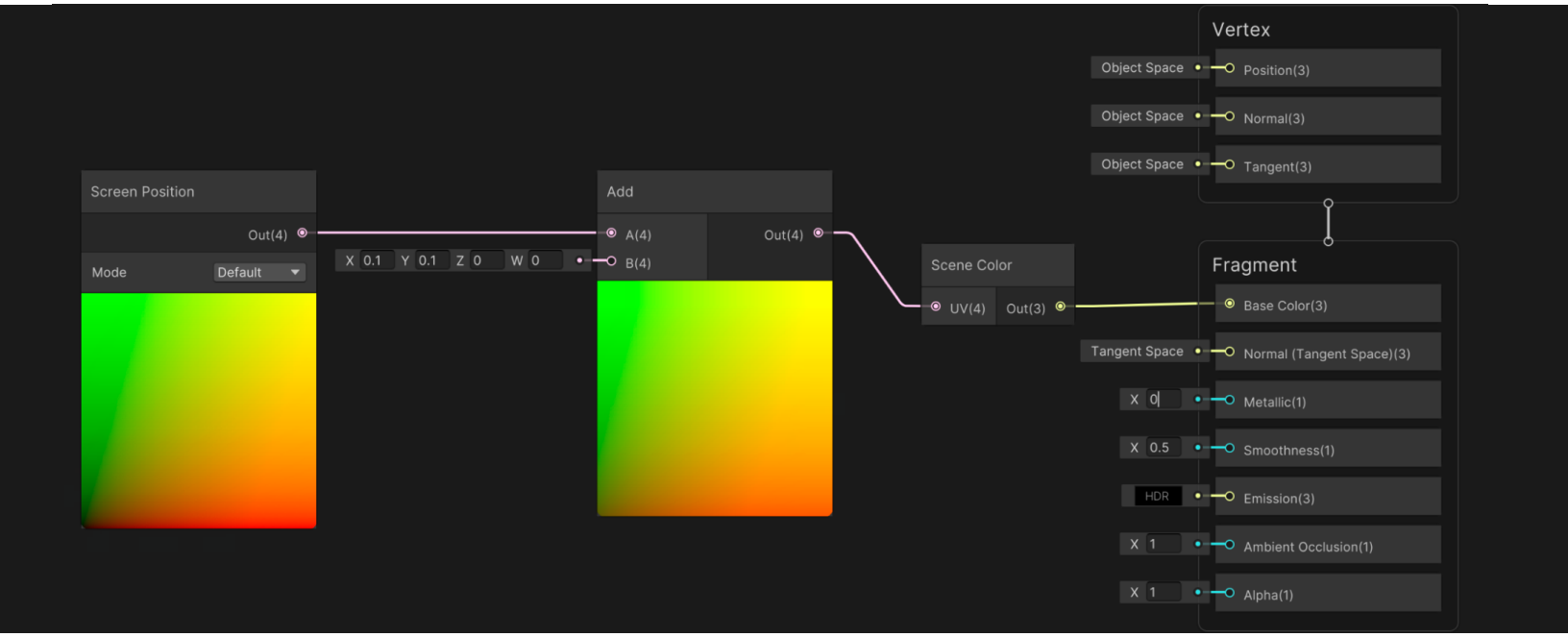
Scene Color ノードの使用

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** の順に移動し、**Transparent** を設定します。

Scene Color ノードは、デフォルトでは正規化されたスクリーン座標を UV に使用し、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、滑らかさの影響を受けたライティングが適用され、下の画像が生成されます。

Scene Color の使用結果

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。

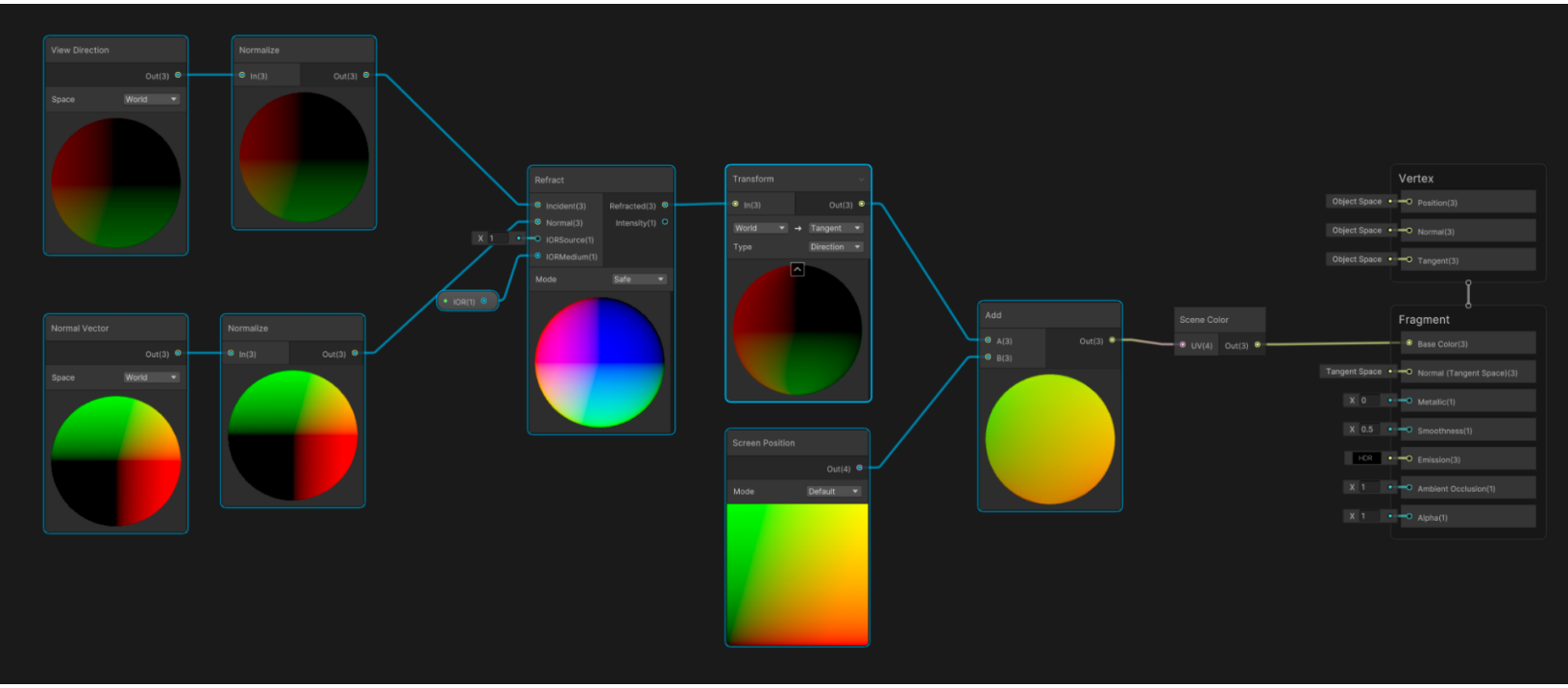
Opaque Texture オフセット

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。

スケーリング用に、新しい Float プロパティの **IOR** (屈折率の略) を加えることができます。これをスライダーとして設定し、最小値を 1、最大値を 6 にします。ビュー方向については、View Direction ノードを加え、Normalize ノードに接続し、単位の長さになるようにします。

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。

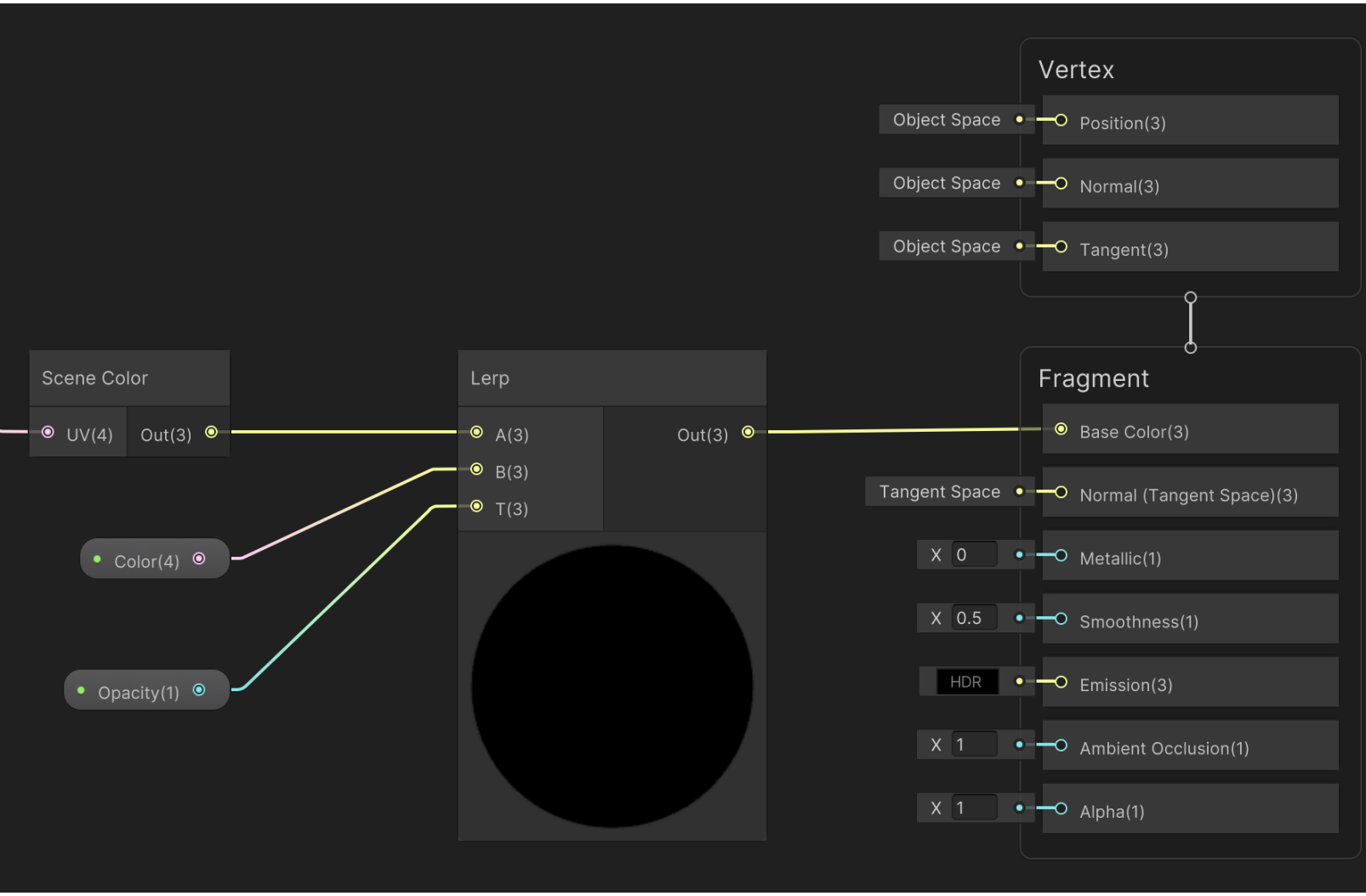


基本的な屈折グラフ

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 プロパティを加えることで、結果に着色できるようになります。Lerp ノードを加え、スライダーモード (範囲 0 から 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。



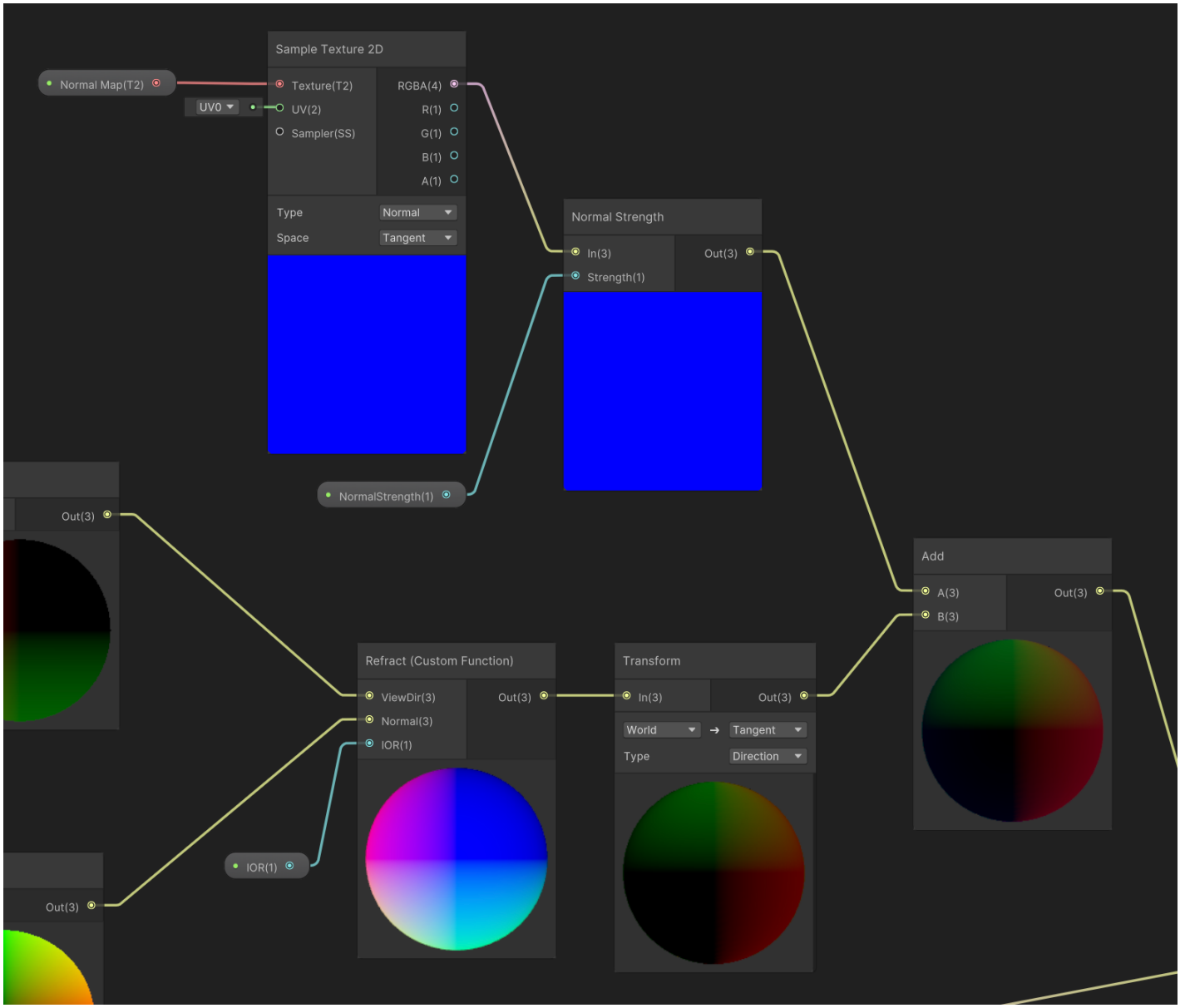
グラフに着色ステージを追加

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

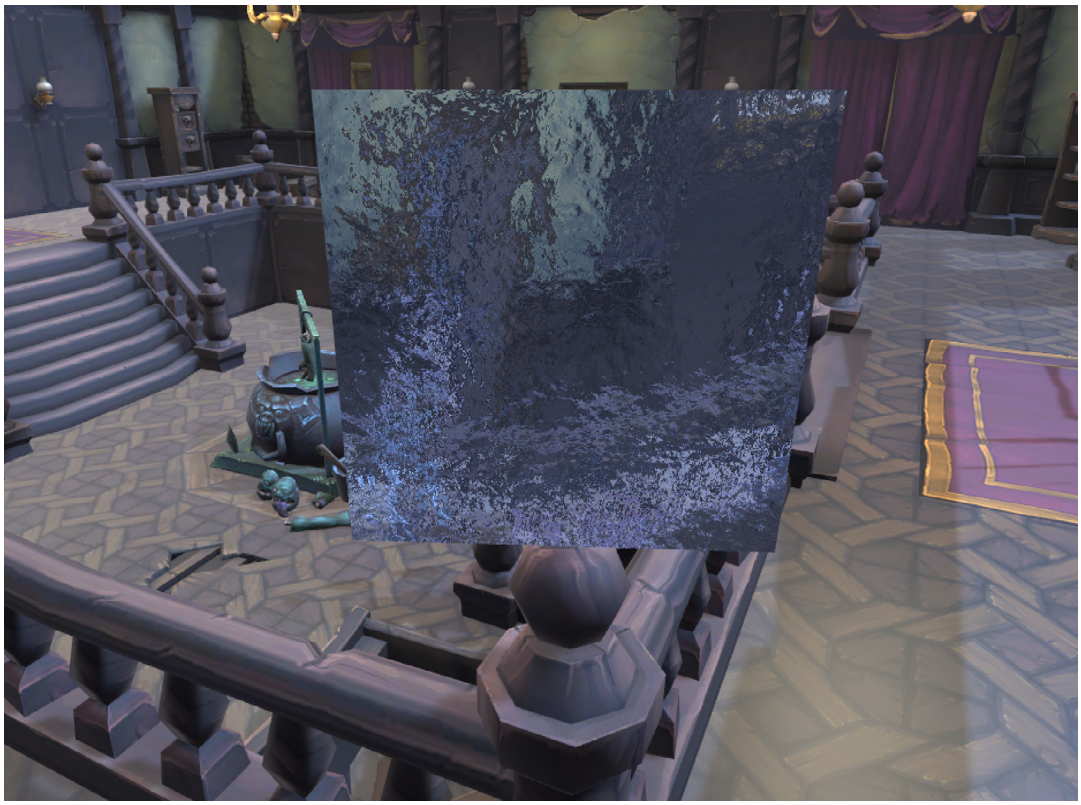
法線は屈折に影響します。つまり、1 つの平面は Opaque Texture のオフセットバージョンを取得するだけです。

ここで法線マップを追加します。まず、シェーダーに **Texture 2D** プロパティを加え、**Normal Map** と名付けます。さらに、**Normal Strength** という名前の Float プロパティをスライダー (範囲 0 から 1) として加えます。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** プロパティに設定します。**Normal Strength** ノードを作成し、入力には Sample Texture 2D ノードの RGBA(4) 出力を設定します。**Strength** 入力には **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、下の画像に示されている効果が生成されます。この例では、ひし形ではなく 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 から 6 ではなく -0.15 から 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シンプルなので、ファイルではなく文字列として使用できます。

```
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 氏による スクリーン スペース 屈折 の解説

— Steven Cannavan 氏による ScreenSpace 平面 リフレクション の GitHub リポジトリ

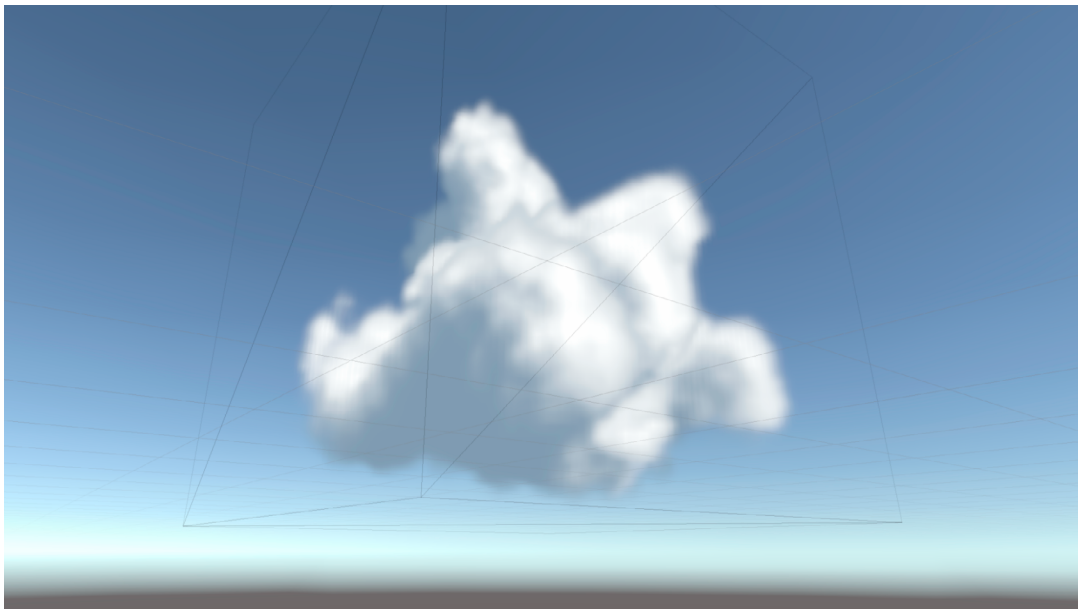
— Kyle W. Powers 氏による リフレクション プローブ と スクリーン スペース リフレクション の解説

— AE Tuts による Shader Graph を使用した屈折 のチュートリアル

— Binary Lunar 氏による Unity でのクリスタルの Shader Graph

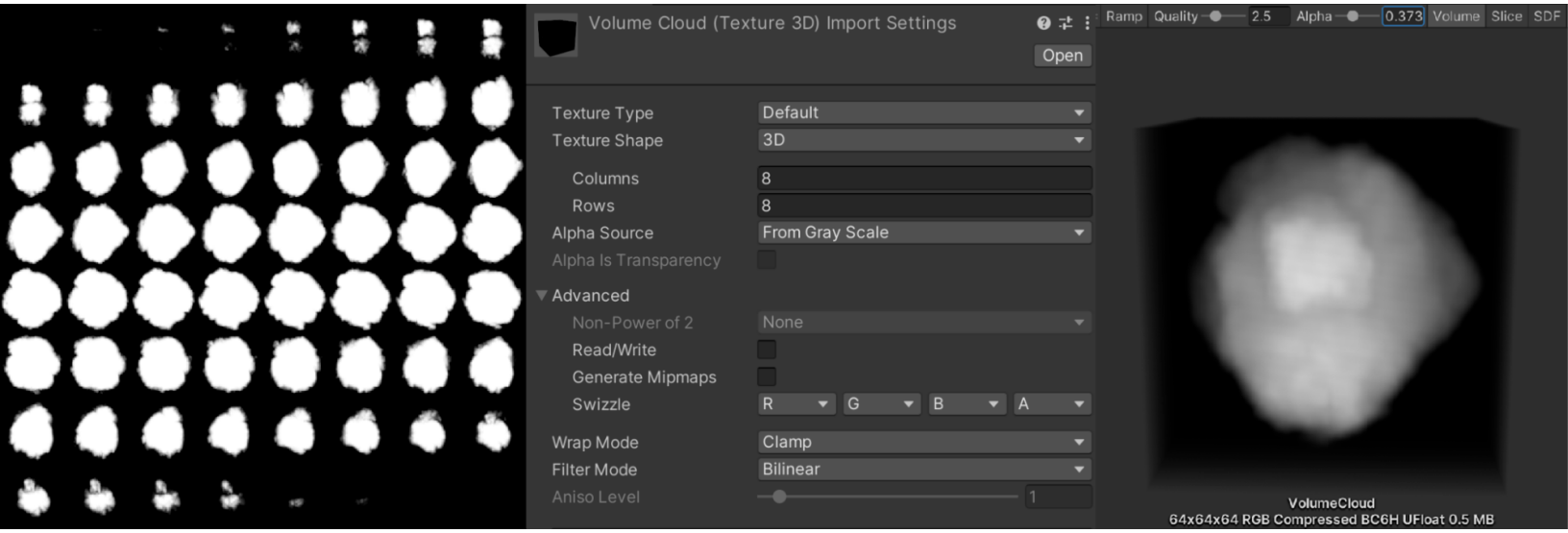
ボリュメトリック

Zoink! によるゲームの『*Lost in Random*』では、プレイヤーを幻想的な王国に入り込む独特のアートスタイルを実現していますが、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。こちらの記事 で説明されているように、ボリュメトリックフォグが URP によって再現されています。



ボリュ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。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。



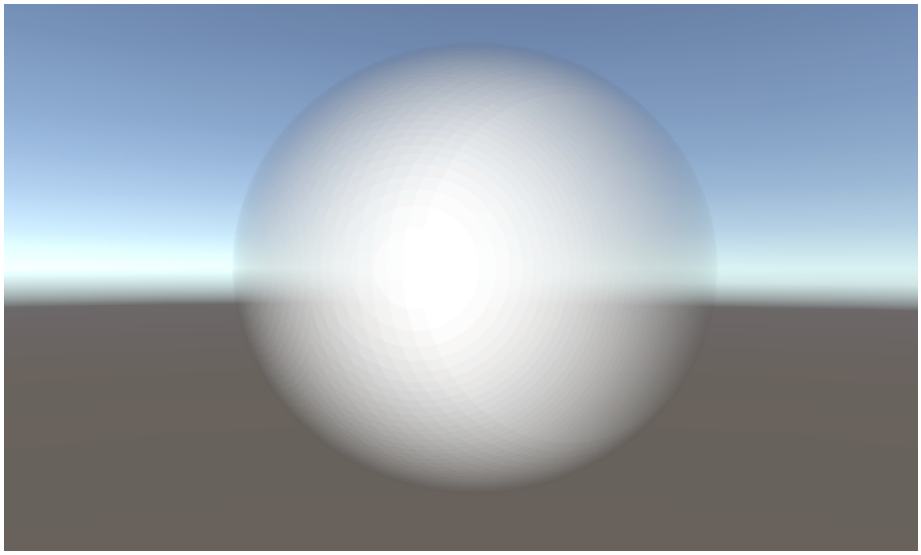
左から順に、3D テクスチャ、そのインポート設定、Inspector でのプレビュー

3D の雲テクスチャを作成する方法について説明する YouTube のショート動画

左に示す YouTube のショート動画 では、Blender を使用して 3D の雲テクスチャを作成する方法を解説しています。

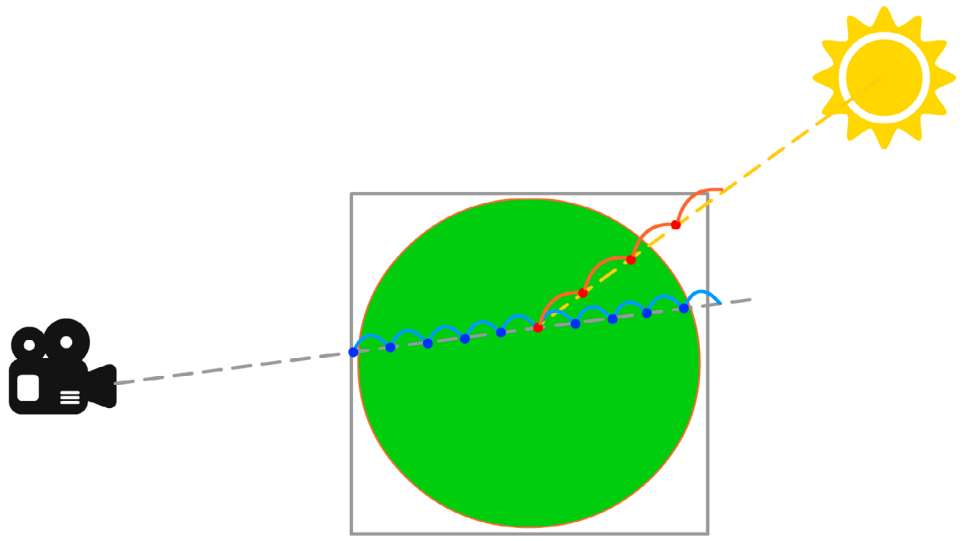
これまでのレシピと同じく、このシェーダーもシェーダーグラフで作成します。完成したものを確認するには、**Scenes > Volumetric Clouds** の順に移動して、**VolumetricClouds** シーンを開きます。このシーンには、カメラ、ディレクショナルライト、キューブが含まれることに注意してください。キューブでは、マテリアル RaymarchMat が使用されます。

このレシピを開始するには、RaymarchMat マテリアルに、Nik Lever 氏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、辺の部分が透明になります。



シェーダーグラフ Raymarchv1SG の使用

キューブをレンダリングしているはずなのに、スフィアが表示されているのはなぜでしょうか。その答えはレイ マーチング です。Wikipedia ページ では、レイマーチング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一種であり、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。

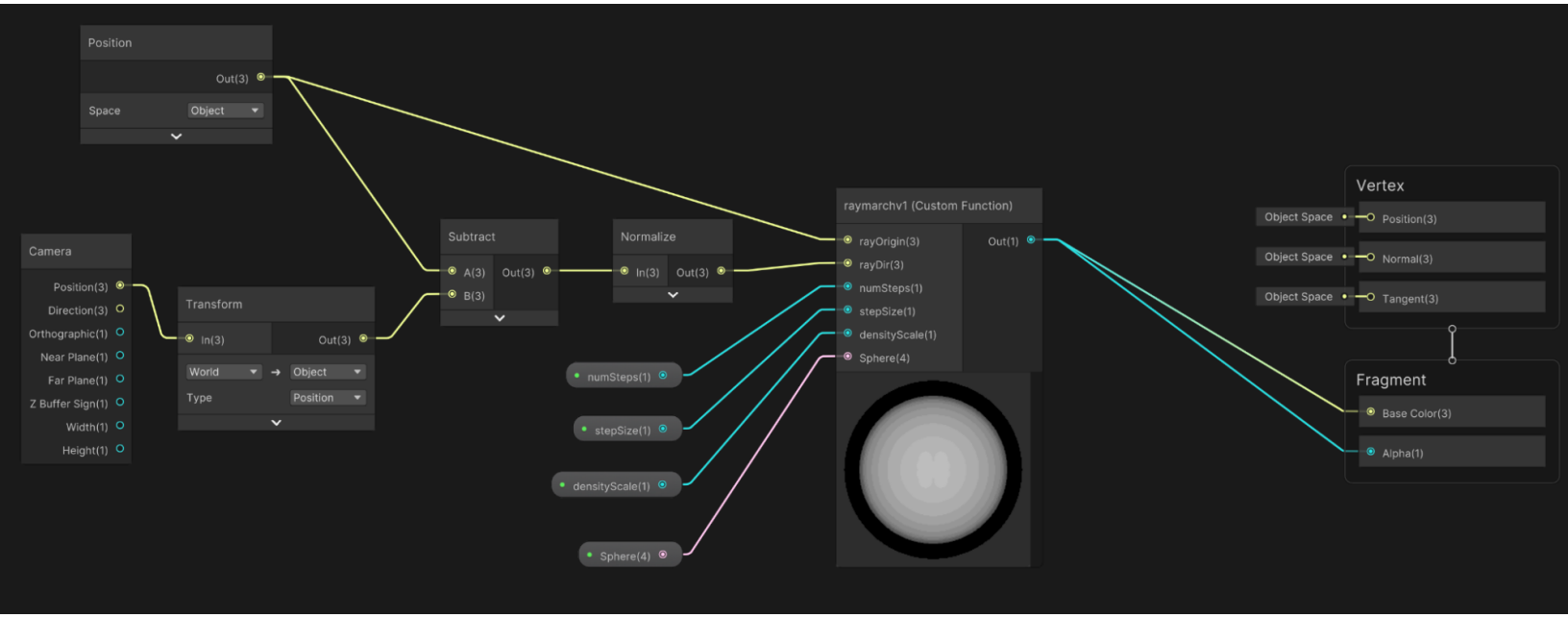
このグラフでは、**Scripts > HLSL > Raymarch.hlsl** のファイルをベースとした Custom Function ノードが使用されます。このバージョン 1 では、関数 raymarchv1 を使用します。変数 density は 0 に初期化されます。次に、numSteps の回数だけ for ループを実行します。rayOrigin は、rayDirection で定義された方向に、stepSize に応じて動きます。

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;
}
```

計算はオブジェクト空間で行います。rayOrigin は Position ノードを使用して取得します。rayDirection を取得するには、Camera ノードの Position 出力を Transform ノードに接続し、Transform ノードの入力を World、出力を Object に設定する必要があります。

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。そのため、Position を入力 A、Camera Position を入力 B に指定した Subtract ノードを使用して、レイ方向 (rayDirection) 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。バージョン 2 では 3D テクスチャを導入します。まず、RaymarchMat で **Shader Graphs > Raymarch2SG** を使用するように設定します。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 プロパティから直接取得される UnityTexture3D volumeTex。

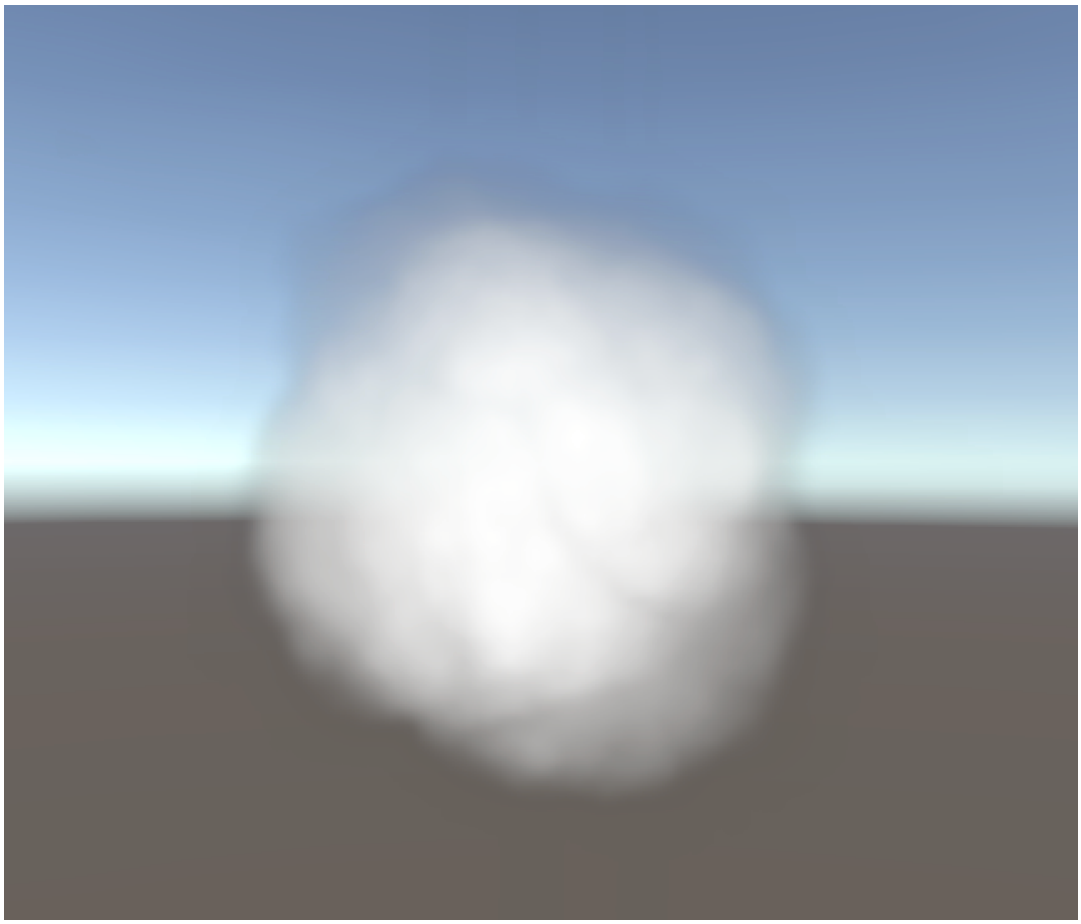
— マクロ SAMPLE_TEXTURE3D。SamplerState インスタンスが必要な 3D テクスチャを処理するときに必要です。

    — 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 にオフセットを加えた float3 のサンプル位置を使用して、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。

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。ここでは、raymarch という関数を使用します。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**、**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。先ほど示した "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 (青い点) ごとに、メインライトに向けたレイ (図の黄色の点線) が取得されます。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ライトは少なくなります。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。

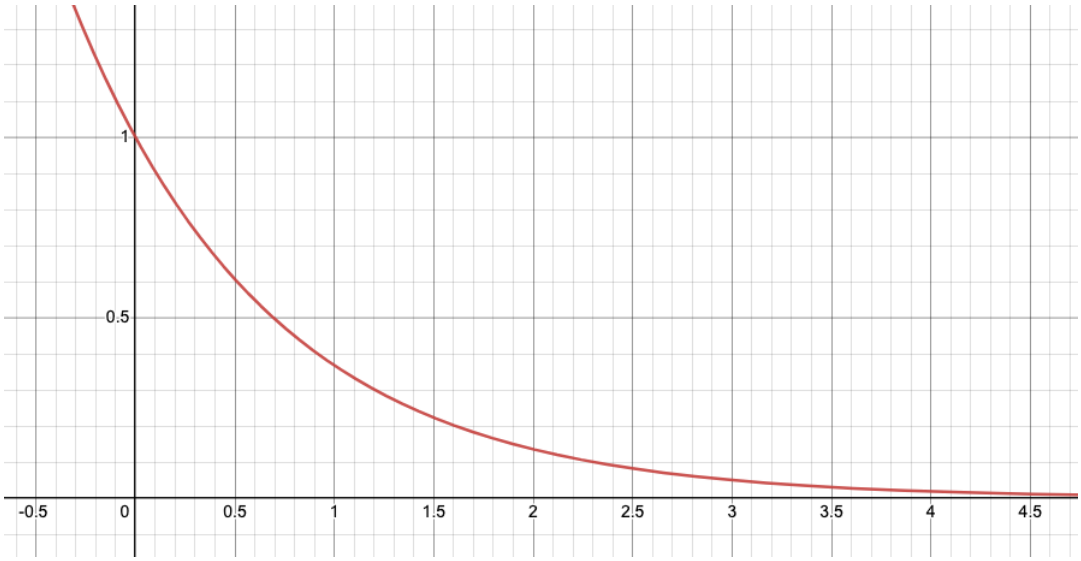
```
void raymarch_float( float3 rayOrigin, float3 rayDirection, float numSteps,
                     float stepSize, float densityScale, UnityTexture3D volumeTex,
                     UnitySamplerState volumeSampler, float3 offset,
                     float numLightSteps, float lightStepSize, float3 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);
}
```

ライトのループはシンプルで、**numLightSteps** 変数で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);

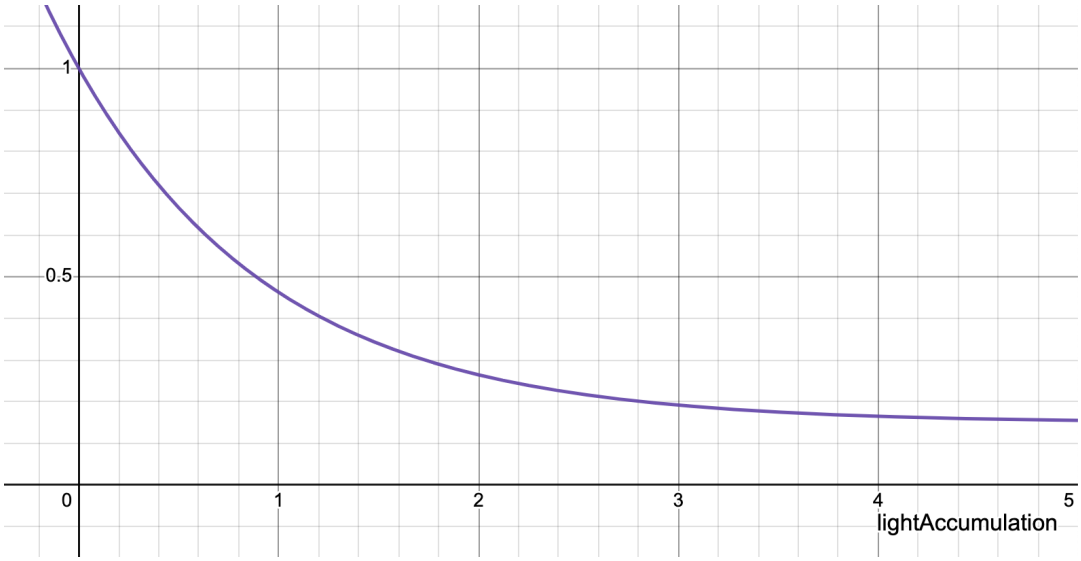
まず、lightTransmission は $e^{-lightAccumulation}$ として設定されます。定数 e はオイラー数で、約 2.718 です。以下のグラフにこの関数の結果を示します。水平軸は lightAccumulation の値、垂直軸は exp(-lightAccumulation) です。累積ライト密度 lightAccumulation が 0 のとき、exp(-lightAccumulation) は 1 です。lightAccumulation が増加すると、exp(-lightAccumulation) は急激に減少し、lightAccumulation が 5 以上になると 0 に近づきます。



0 から 4 の範囲の e^{-x} のグラフ

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

次に、shadow 値が計算されます。ここでは、**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);
```

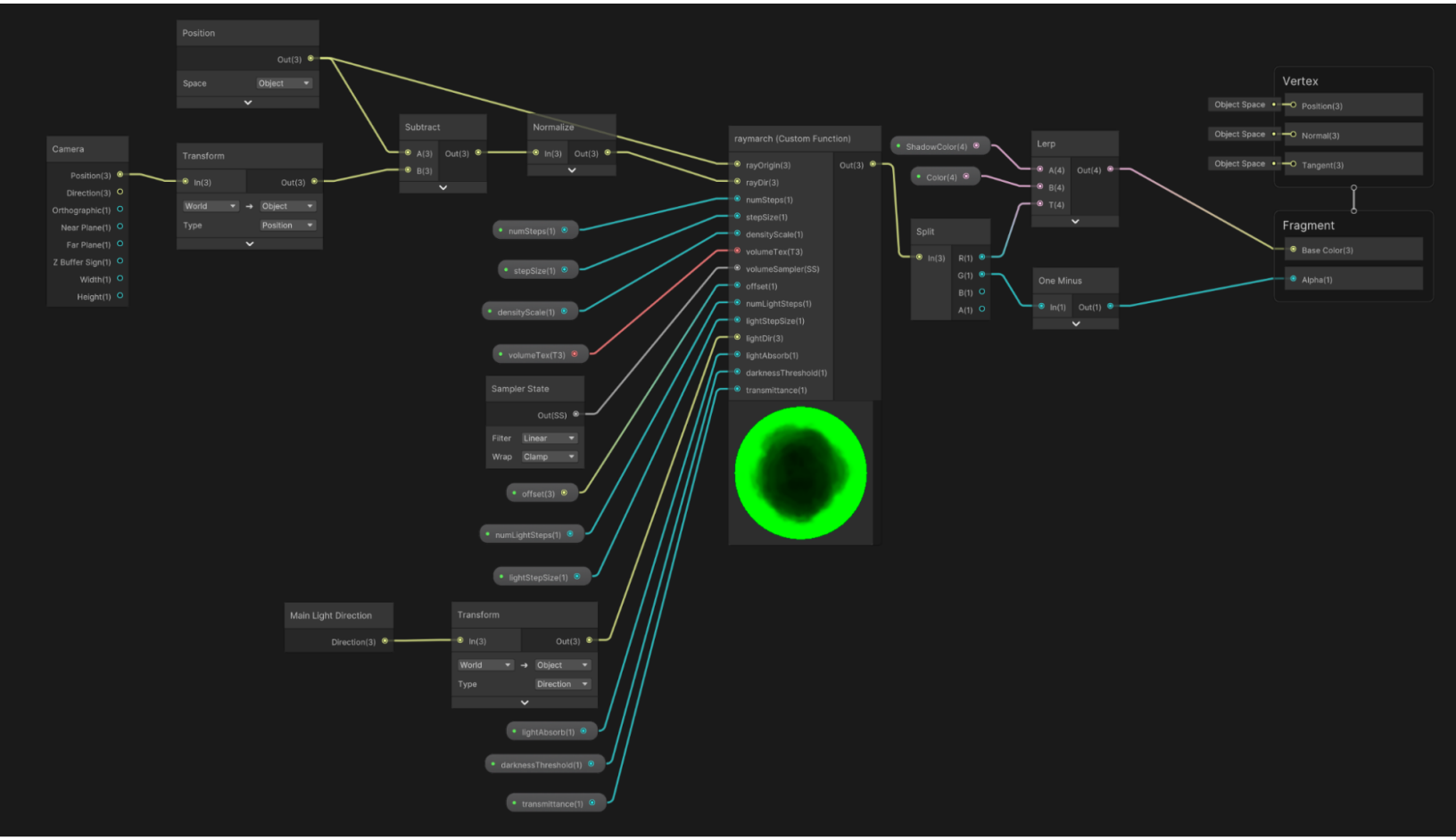
transmittance の初期値はプロパティとして渡されますが、ビュー方向のステップごとに、その値は $e^{-density*lightAbsorb}$ によって乗算されます。プロパティ **lightAbsorb** は、雲の中で拡散によって光がどれだけ失われるかを制御します。

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加わっており、color は B 入力、shadowColor は A 入力として設定されます。

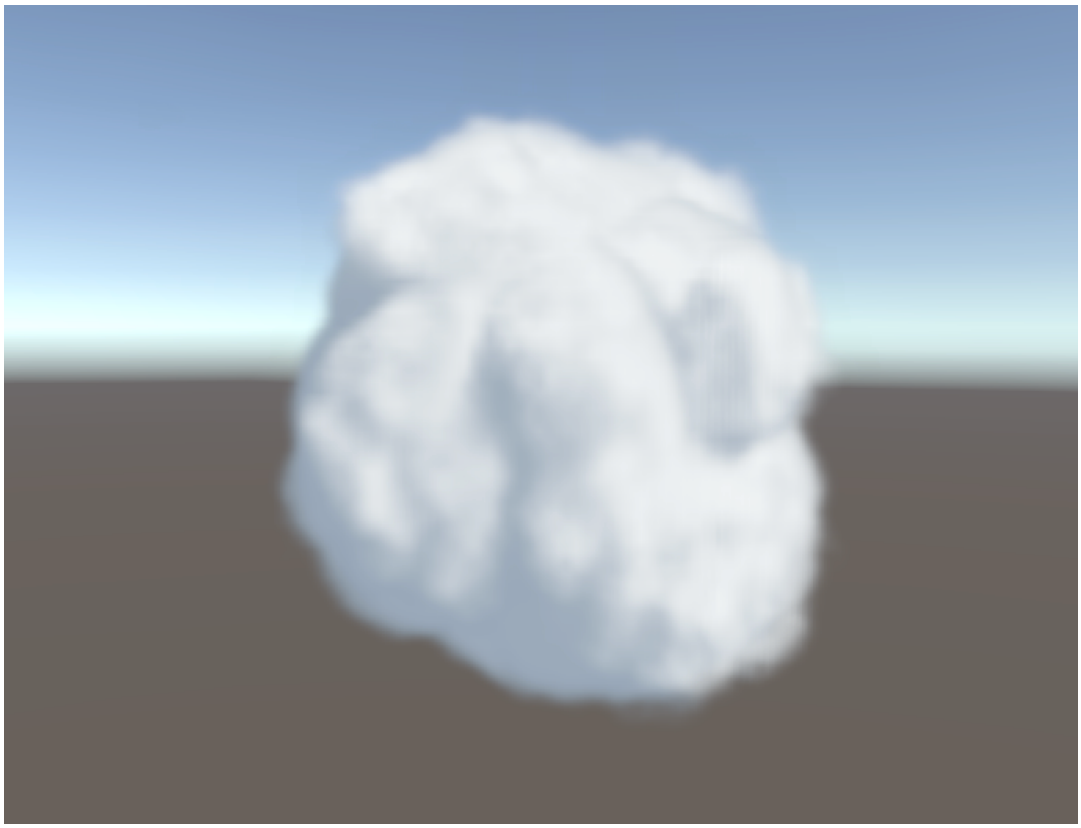
finalLight raymarch ノードの Out.x が 0 の場合、shadowColor が Lerp ノードの出力に渡されます。finalLight が 1 の場合は、color が出力に渡されます。範囲が 0 ～ 1 の場合、shadowColor と color の線形補間が出力です。Lerp ノードの出力を Fragment > Base Color に直接つなげます。

Alpha は、transmission 値が Out.y の raymarch ノードを使用します。この値は、Alpha を 1 にすべき場合は 0、0 にすべき場合は 1 となります。One Minus ノードを使用して Split ノードの B 値を補正し、Fragment > Alpha に接続します。



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。



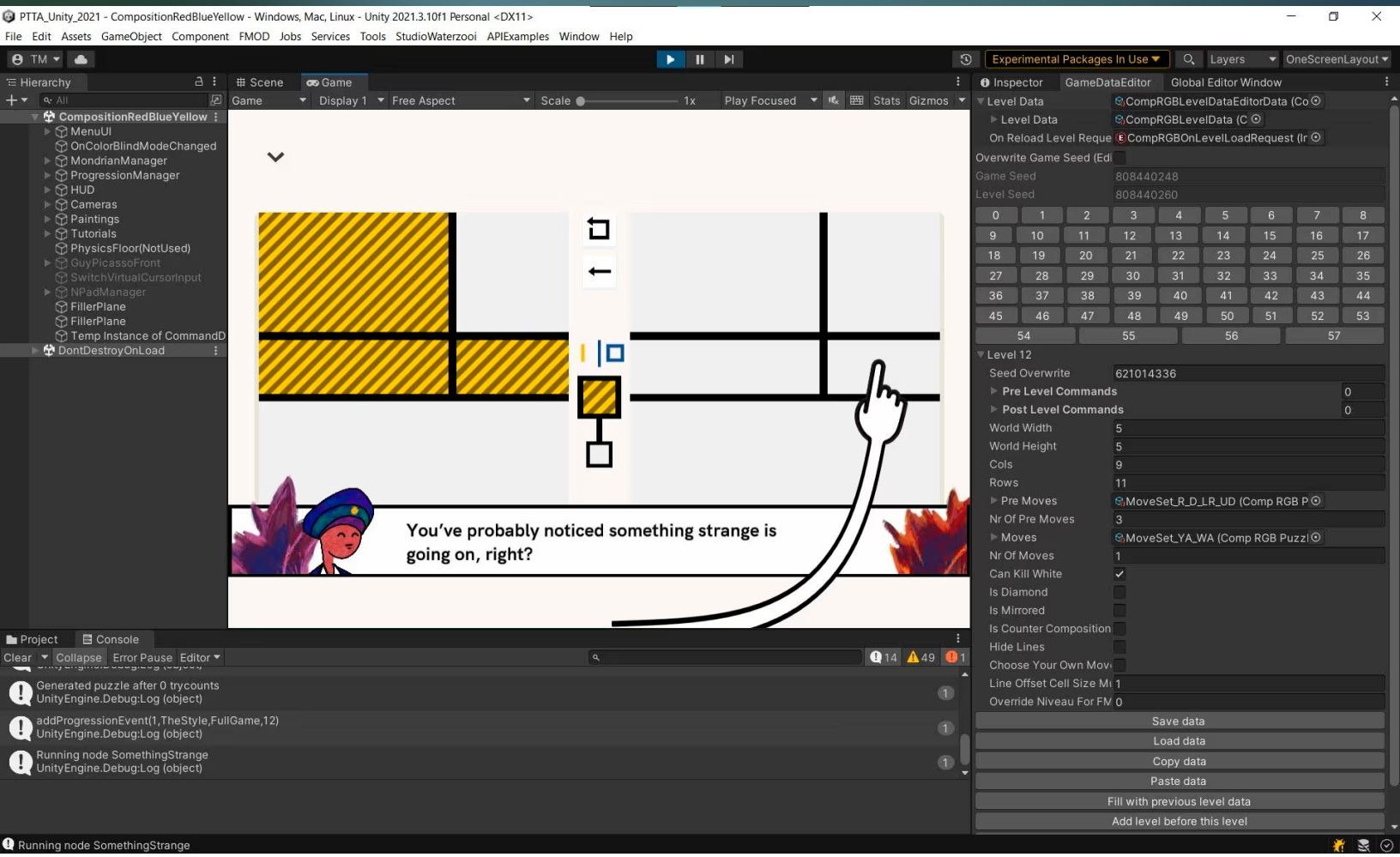
レイマーチングを使用した雲

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 の使用や、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）があります。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 氏による ボリュメトリック レイ マーチング クラウド シェーダー の解説
- Sebastian Lague 氏による動画『Coding adventure:Clouds』
- Camelia Slimani 氏による Houdini を使用した ボリュメトリック クラウドの 作成方法
- OccaSoftware による Altos スカイ システム
- Adrian Polimeni 氏による リアル タイム ボリュメトリック クラウド の GitHub リポジトリ

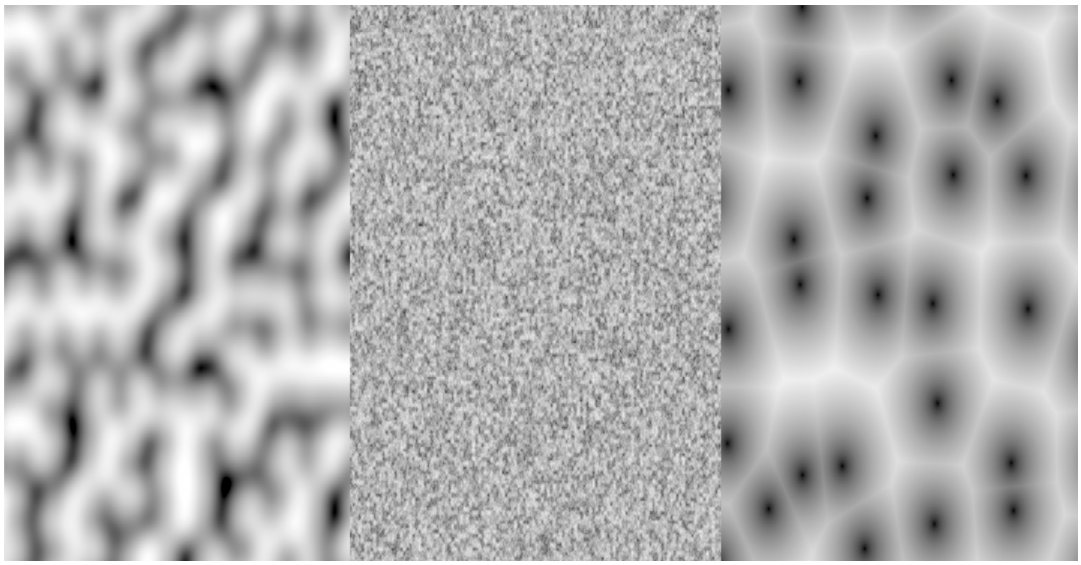
プロシージャルノイズ



Thomas Waterzool 氏によるゲーム『*Please, Touch the Artwork*』では、プロシージャル生成を用いてレベル作成プロセスを高速化しています。これは、ゲームデザイン要素をランダム化する好例です。

プロシージャルノイズはパワフルなテクニックで、テクスチャ、地形、その他の環境要素を、ランダムに見えるようにしつつも細かく制御可能な方法で生成できます。これらの効果をアルゴリズムによって生成することで、開発者が細部まで手作業を行わなくても、広大で多様な世界を作成できます。

Unity では、グラデーションまたは パーリンノイズ、シンプレックス ノイズ、ボロノイまたは ウォーリー など、いくつかのタイプのプロシージャルノイズをサポートしており、それぞれに独自の用途と利点があります。プロシージャルノイズを理解して実装することで、ゲームや相互作用する体験の美的側面とパフォーマンス側面の両方を強化できます。



プロシージャルノイズの例:左からグラデーション、シンプレックス、ボロノイ

プロシージャルノイズのタイプ

Unity で最も一般的なプロシージャルノイズのタイプはパーリンノイズとシンプレックノイズで、どちらも Ken Perlin 氏が開発したものです。

パーリンノイズは、滑らかで自然な変化を持つテクスチャ、地形、アニメーションを作成するための定番アルゴリズムです。急激なエッジを避けた、滑らかで連続的なパターンを生成するため、山岳地帯の地形、雲のテクスチャ、水の波形などに適しています。シンプレックノイズは、特に 3 次元アプリケーション向けのより効率的な代替手段です。パーリンノイズよりも滑らかで計算負荷が低いため、パフォーマンスが重視されるモバイルゲームや VR 環境に役立ちます。

Unity でプロシージャルノイズを実装する

Unity では、プロシージャルノイズはシェーダーを通じて、または C# スクリプトで直接、実装できます。2D テクスチャの場合、`Mathf.PerlinNoise()` 関数は特に使いやすくて便利です。X 座標と Y 座標に基づいてノイズ値を生成し、地形、テクスチャ、パーティクルエフェクトに簡単に適用できます。頻度、振幅、スケールなどのパラメーターを調整することで、開発者はノイズ出力を制御してさまざまな視覚効果を実現できます。

例えば、開発者が地形のハイトマップを作成するためにパーリンノイズを使用するとします。これは、パーリンノイズ値によるグリッドを生成し、それらの値を高さレベルにマップすることで実現できます。各ノイズ値は地形上の点に対応しているため、手動で作成しなくても、丘や谷などの自然な地形を生成可能です。その後、Unity の Terrain Tools では、これらのハイトマップを適用して 3D 地形を作成し、テクスチャを加えたり、他のプロシージャル生成手法を使用したりすることで、さらに調整できます。

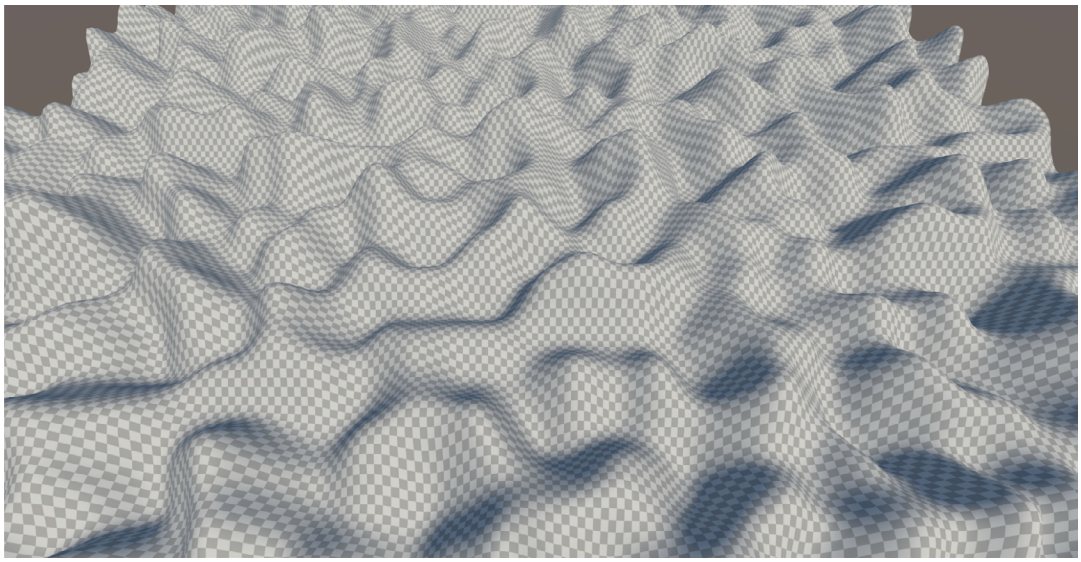
プロシージャルのハイトマップの例

Unity において、C# スクリプトでパーリンノイズを使用してハイトマップを生成する基本的な例を示します。このスクリプトは、Terrain (地形) ゲームオブジェクトにパーリンノイズを適用することで、丘や谷がある地形を作成します。

1.Unity で Terrain (地形) ゲームオブジェクトを作成します。

2.以下のスクリプトを Terrain (地形) にアタッチします。

3.必要に応じてパラメーターを調整し、地形の外観を制御します。



プロシージャルの高さの例

```
using Unity.VisualScripting;
using UnityEngine;

[ExecuteInEditMode]
public class ProceduralHeight :MonoBehaviour
{
    public int depth = 20; // 地形の最大高さ
    public float scale = 20f; // ノイズの "引き伸ばし" 具合を制御
```

```
private int width = 256; // 地形の幅
private int height = 256; // 地形の高さ
private float offsetX = 100f; // X 座標のオフセット（開始時にランダム化）
private float offsetY = 100f; // Y 座標のオフセット（開始時にランダム化）

private int _depth;
private float _scale;

private Terrain terrain;

void Start()
{
    // 毎回異なる地形になるようオフセットをランダム化
    offsetX = Random.Range(0f, 9999f);
    offsetY = Random.Range(0f, 9999f);

    _depth = depth;
    _scale = scale;

    terrain = GetComponent<Terrain>();
    terrain.terrainData = GenerateTerrain(terrain.terrainData);
}

TerrainData GenerateTerrain(TerrainData terrainData)
{
    terrainData.heightmapResolution = width + 1;
    terrainData.size = new Vector3(width, depth, height);
    terrainData.SetHeights(0, 0, GenerateHeights());
    return terrainData;
}

float[,] GenerateHeights()
{
    float[,] heights = new float[width, height];
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            float xCoord = (float)x / width * scale + offsetX;
            float yCoord = (float)y / height * scale + offsetY;
```

```
                heights[x, y] = Mathf.PerlinNoise(xCoord, yCoord);
            }
        }
        return heights;
    }

    void Update(){
        if ( depth != _depth || scale != _scale ){
            terrain.terrainData = GenerateTerrain(terrain.terrainData);
             _depth = depth;
             _scale = scale;
        }
    }
}
```

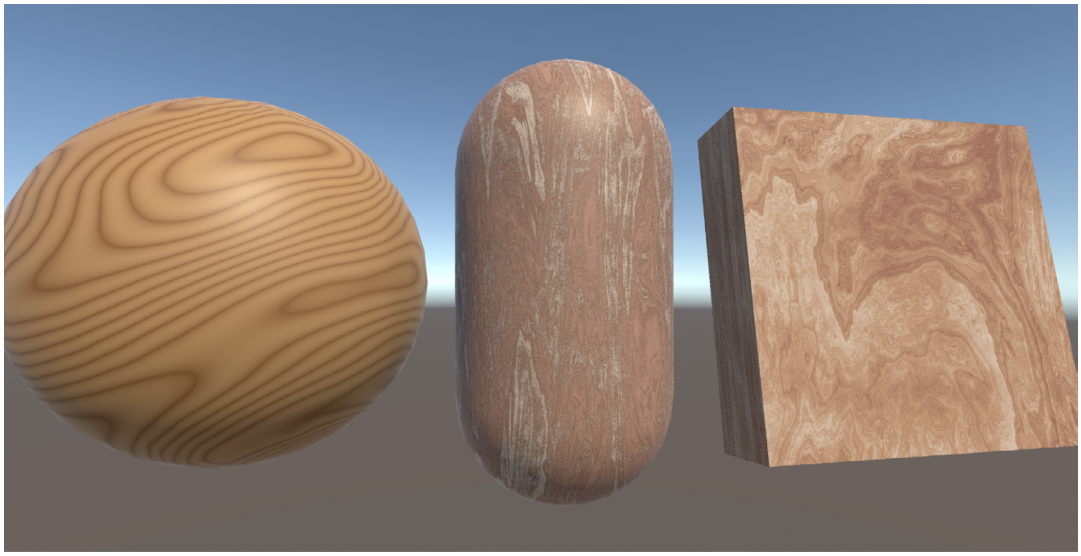
このスクリプトの主な値とプロパティをいくつか見てみましょう。

- **depth**:地形の垂直スケールです。値を大きくすると、より高い丘が作成されます。
- **width と height**:地形の寸法 (横幅と奥行き) です。
- **scale**:ノイズの "ズーム度合い" を制御します。値を大きくすると、より滑らかで広がりのある丘になります。
- **offsetX と offsetY**:スクリプトが実行されるたびに、地形のレイアウトを変化させるためにランダムオフセットが追加されます。

`GenerateHeights()` 関数は、地形上の各点の高さ値を表す 2D 配列を作成します。各値は `Mathf.PerlinNoise` を使用して生成され、xCoord と yCoord に基づく滑らかで自然なバリエーションが得られます (width、height、scale によってスケールされます)。

リソースの中では、この例は **Scenes > Procedural Noise > Terrain** にあります。このシンプルなハイトマップ生成手法は、さらに複雑な処理にも応用できます。例えば、複数レイヤーのパーリンノイズを使用して、フラクタルな地形を作成したり、他のノイズ関数を適用したりできます。

ノイズを使用して木目テクスチャを生成する

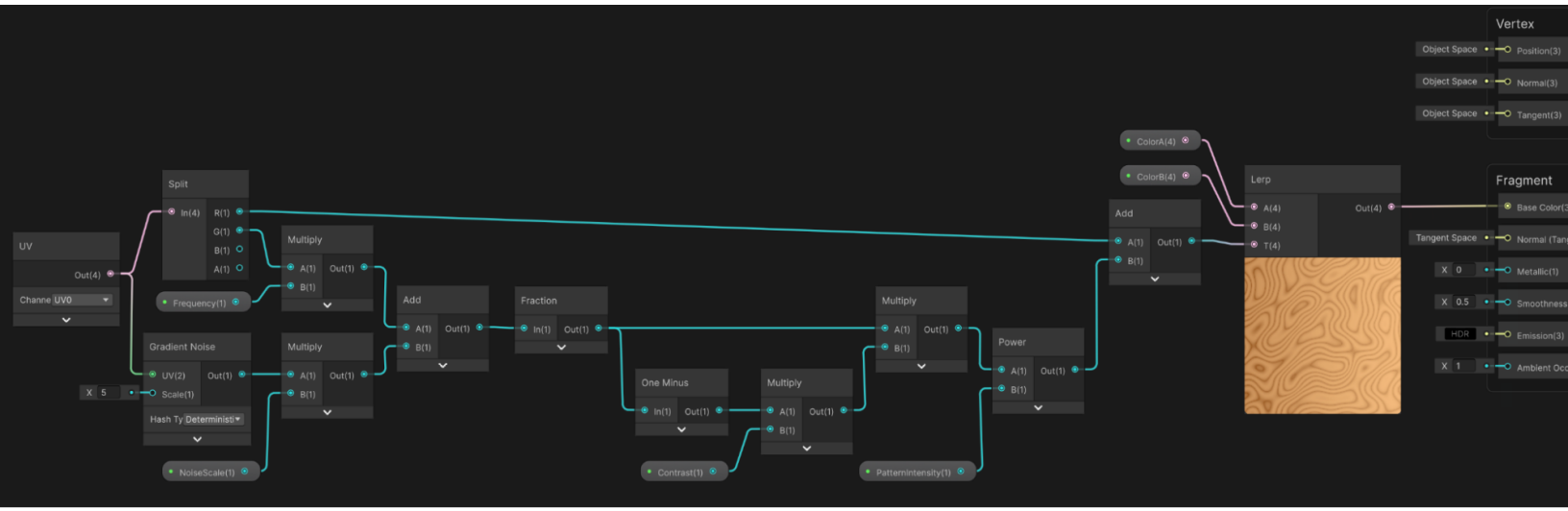


木目テクスチャにプロシージャルノイズを使用した例

プロシージャルノイズのもう 1 つの用途は、上の画像にある木目のテクスチャのように、自然なテクスチャを模したシェーダーを作成することです。この例を確認するには、**Scenes > Procedural Noise > Wood** を参照してください。ここでは、2 種類の木目のシェーダーを紹介します。1 つは高度にスタイライズされたもので、もう 1 つはよりリアルなものです。まずは、スタイライズされたバージョンの作成から開始しましょう。

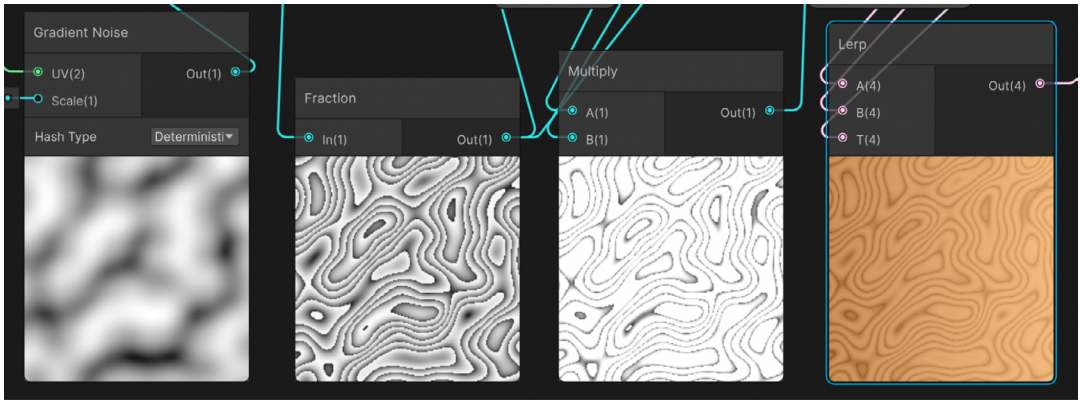
1. **Shader Graph > URP > Lit Shader Graph** から Shader Graph を作成します。
2. 以下のプロパティを加えます。
    a. `ColorA` (色のデフォルト値:7D490B)
    b. `ColorB` (色のデフォルト値:BB905D)
    c. `Frequency` (Float のデフォルト値:2)
    d. `NoiseScale` (Float のデフォルト値:6)
    e. `PatternIntensity` (Float のデフォルト値:0.6)
    f. `Contrast` (Float のデフォルト値:4)
3. **Geometry > UV** ノード (Channel: UV0) を作成します。
4. **Procedural > Noise > GradientNoise** ノードを作成します。ステップ 3 の出力を **UV** 入力に接続し、**Scale** 入力を **5** に設定します。**Hash Type** は **Deterministic** のままにします。
5. **Multiply** ノードを作成し、その入力 **A** にステップ 4 の出力、入力 **B** に `NoiseScale` プロパティを設定します。
6. **Add** ノードを作成し、その入力 **B** にステップ 5 の出力を接続します。
7. ステップ 3 の **UV** ノードに戻ります。**Gradient Noise** ノードの上に **Split** ノードを作成し、その入力に **UV** ノードの出力を接続します。

8. **Multiply** ノードを作成し、その入力 **A** に Split ノードの **G** 出力、入力 **B** に `Frequency` プロパティを設定します。
9. ステップ 8 の出力をステップ 6 の入力 **A** に接続します。
10. **Fraction** ノードを作成し、その入力に **Add** ノードの出力を接続します。
11. **One Minus** ノードを作成し、その入力にステップ 10 の出力を接続します。
12. **Multiply** ノードを作成し、その入力 **A** にステップ 11 の出力、入力 **B** に Contrast プロパティを接続します。
13. 別の **Multiply** ノードを作成し、その入力 **A** にステップ 10 の出力、入力 **B** にステップ 12 の出力を接続します。
14. **Power** ノードを作成し、その入力 **A** にステップ 13 の出力、入力 **B** に `PatternIntensity` プロパティを接続します。
15. **Add** ノードを作成し、その入力 **A** にステップ 7 の **Split** ノードの **R** 出力、入力 **B** にステップ 14 の出力を接続します。
16. **Lerp** ノードを作成し、その入力 **A** に `ColorA` プロパティ、入力 **B** に `ColorB` プロパティ、入力 **T** にステップ 15 の出力を設定します。
17. ステップ 16 の出力を **Fragment の Base Color** に接続します。



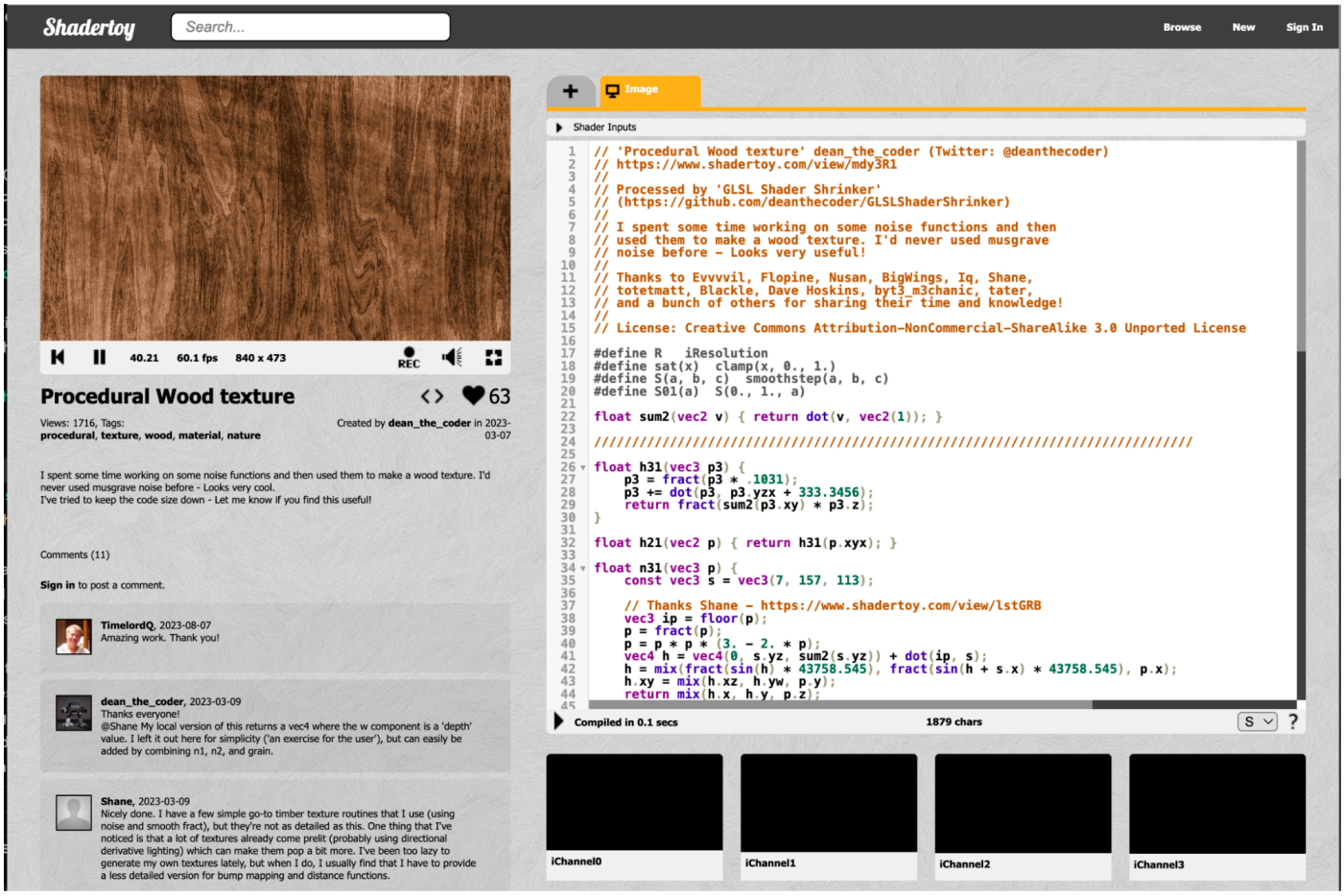
スタイライズされた木目の Shader Graph

UV **u 値** は、最後の Add ノードに直接渡される木目の位置を設定するために使用されます。**v 値** には、一連の複雑な操作が加えられます。まず Gradient Noise に始まり、Multiply ノードを通じて操作され、木目線 (パターン) の頻度とスケールが調整されます。シェーダーのキーとなるのは、Fraction ノードを介して計算値の小数部分を使用することです。プロパティを使用して、木目やパターンのコントラストと強度を調整することもできます。



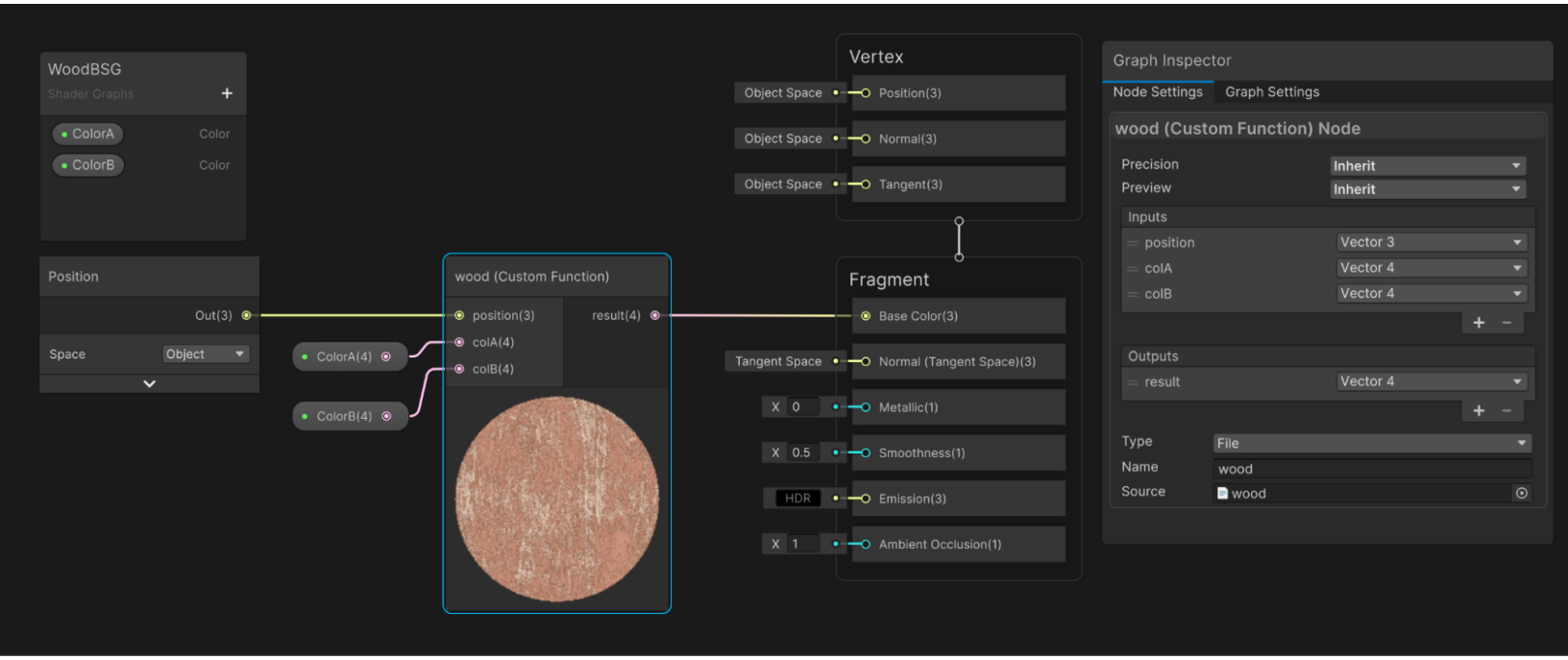
シェーダーの各段階: Gradient Noise (最も左)、小数部分を使用して複数の木目線を生成する効果 (左から 2 番目)、コントラストを強調する効果 (右から 2 番目)、色を加える効果 (最も右)

Custom Function ノードを使用して実現される、よりリアルなバージョンを見てみましょう。これは、ShaderToy の こちらの 例 を改変したものです。ShaderToy はシェーダーコードの優れたソースですが、すべてのコードが HLSL ではなく GLSL 構文で書かれています。そのため、Shader Graph の Custom Function ノードで使用するには変換する必要があります。HLSL コーディングの知識があれば、この変換は比較的簡単にできます (vec → float、fract → frac、mix → lerp など)。



ShaderToy ウェブサイトのシェーダーの 例

このカスタム関数は、下に示すコードで確認できます。デフォルトの色を使用する場合は、**Geometry > Position** ノードを **Object** に設定するだけです。または、2 つの **Color** プロパティを加えて色を定義することも可能です。`ColorA` は木目線の色、`ColorB` は通常色を指定します。



ユーザー定義の色を使用した、カスタム関数による木目の Shader Graph

```
#define sat(x)      clamp(x, 0.0, 1.0)
#define S(a, b, c)  smoothstep(a, b, c)
#define S01(a)      S(0.0, 1.0, a)

float sum2(float2 v) { return dot(v, float2(1.0, 1.0)); }

//////////////////////////////////////////////////////////////////////////////

float h31(float3 p3) {
       p3 = frac(p3 * 0.1031);
       p3 += dot(p3, p3.yzx + 333.3456);
       return frac(sum2(p3.xy) * p3.z);
}

float h21(float2 p) { return h31(p.xyx); }

float n31(float3 p) {
       const float3 s = float3(7, 157, 113);
```

```
	// Shane 氏に感謝します - https://www.shadertoy.com/view/lstGRB
	float3 ip = floor(p);
	p = frac(p);
	p = p * p * (3. - 2. * p);
	float4 h = float4(0, s.yz, sum2(s.yz)) + dot(ip, s);
	h = lerp(frac(sin(h) * 43758.545), frac(sin(h + s.x) * 43758.545), p.x);
	h.xy = lerp(h.xz, h.yw, p.y);
	return lerp(h.x, h.y, p.z);
}

// roughness:(0.0, 1.0]、デフォルト:0.5
// 負の値を含まないノイズ [0.0, 1.0] を返す
float fbm(float3 p, int octaves, float roughness) {
	float sum = 0.,
	      amp = 1.,
	      tot = 0.;
	roughness = sat(roughness);
	for (int i = 0; i < octaves; i++) {
	       sum += amp * n31(p);
	       tot += amp;
	       amp *= roughness;
	       p *= 2.;
	}
	return sum / tot;
}

float3 randomPos(float seed) {
	float4 s = float4(seed, 0, 1, 2);
	return float3(h21(s.xy), h21(s.xz), h21(s.xw)) * 1e2 + 1e2;
}

// 負の値を含まないノイズ [0.0, 1.0] を返す
float fbmDistorted(float3 p) {
	p += (float3(n31(p + randomPos(0.)), n31(p + randomPos(1.)), n31(p + randomPos(2.))) * 2.
- 1.) * 1.12;
	return fbm(p, 8, .5);
}

// float3: ディテール (オクターブ数)、次元 (反転コントラスト)、ラクナリティ
// 負の値を含むノイズを返す
float musgraveFbm(float3 p, float octaves, float dimension, float lacunarity)
{
```

```
       float sum = 0.,
             amp = 1.,
             m = pow(lacunarity, -dimension);
       for (float i = 0.; i < octaves; i++) {
              float n = n31(p) * 2. - 1.;
              sum += n * amp;
              amp *= m;
              p *= lacunarity;
       }

       return sum;
}

// X 軸に沿った波形ノイズ
float3 waveFbmX(float3 p) {
       float n = p.x * 20.;
       n += .4 * fbm(p * 3., 3, 3.);
       return float3(sin(n) * .5 + .5, p.yz);
}

//////////////////////////////////////////////////////////////////////////////
// 数学関数
float remap01(float f, float in1, float in2) { return sat((f - in1) / (in2 - in1)); }

//////////////////////////////////////////////////////////////////////////////
// 木目のマテリアル
float3 matWood(float3 p, float3 colA, float3 colB ) {
       float n1 = fbmDistorted(p * float3(7.8, 1.17, 1.17));
       n1 = lerp(n1, 1.0, 0.2);
       float n2 = lerp(musgraveFbm(float3(n1, n1, n1) * 4.6, 8.0, 0.0, 2.5), n1, 0.85);
       float dirt = 1. - musgraveFbm(waveFbmX(p * float3(.01, .15, .15)), 15., .26, 2.4) * .4;
       float grain = 1. - S(.2, 1., musgraveFbm(p * float3(500, 6, 1), 2., 2., 2.5)) * .2;
       n2 *= dirt * grain;

    // float3 の 3 つの値は木目の RGB 色で、それぞれに合わせて調整する
       return lerp(lerp(colA, colB, remap01(n2, .19, .56)), float3(.52, .32, .19), remap01(n2,
.56, 1.));
}
```

```
//ハードコードされた色
void wood_float( float3 pos, out float4 result){
    float3 colA = float3(.03, .012, .003);
    float3 colB = float3(.25, .11, .04);
    result = float4(pow(matWood(pos, colA, colB), float3(1,1,1) * 0.4545), 0);
}

//ユーザー選択可能な色。colA は木目線の色を表す
void wood_float( float3 pos, float3 colA, float3 colB, out float4 result){
    result = float4(pow(matWood(pos, colA, colB ), float3(1,1,1) * 0.4545), 0);
}
```

このコードは、関数 `wood__float` から始まり、`matWood` を呼び出します。これは、関数 `fbmDistorted`、`musgraveFbm`、`waveFbm` を使用しています。開発者は最良の結果を得るために、多くの "マジックナンバー" を慎重に選び、長時間をかけて調整したと考えられます。**fBm** (フラクショナルブラウン運動) またはマルチフラクタルノイズは、Ken Musgrave 氏が開発したプロシージャルノイズの一種です。そのため、マスグレイブノイズと呼ばれることがよくあります。これは、リアルなテクスチャや地形の生成に使用されます。パーリンノイズなどの従来のノイズ関数とは異なり、マスグレイブノイズは、周波数と振幅が異なる複数のレイヤー (オクターブ) のノイズを組み合わせることで、雲、山、流水などの複雑な自然現象をシミュレートします。

musgraveFbm 関数を詳しく見ていきましょう。まずはパラメーターです。

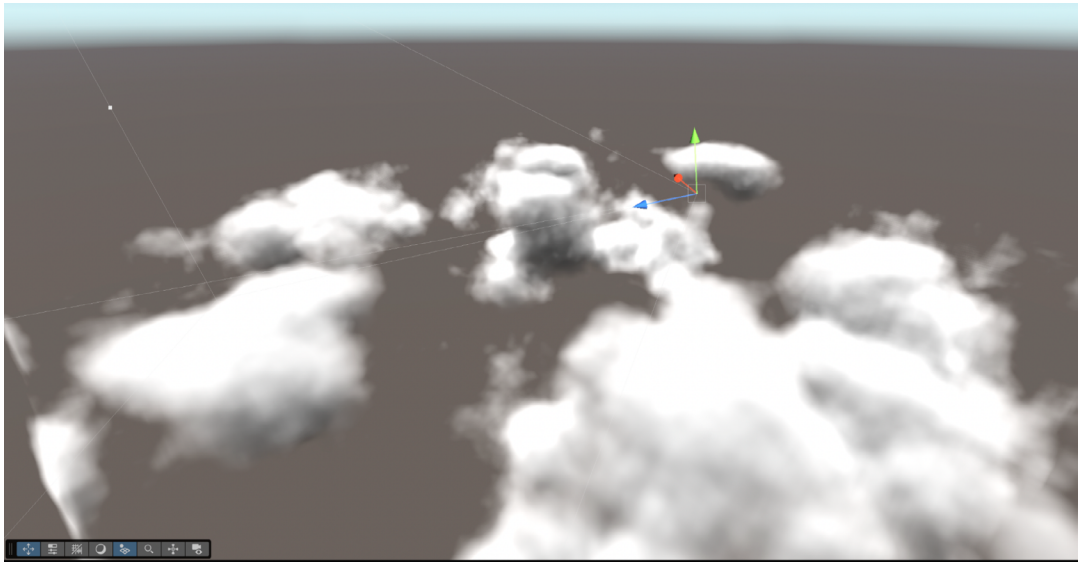
- **P (float3)**:位置の値で、通常はオブジェクト空間の座標です。
- **octaves (float)**:マスグレイブノイズは、いくつかのオクターブのノイズ関数を重ねることで動作します。その結果、レイヤーを重ねるごとにディテールが増し、テクスチャが徐々に複雑に見えてくる合成効果が得られます。
- **dimension (float)**:コントラストを逆に制御します。
- **lacunarity**:オクターブ間の周波数割合を制御します。lacunarity の値を大きくすると、各オクターブのスケールが小さく細密になり、より複雑なパターンになります。

この関数は、以下の 3 つの Float 値を初期化します。

1. 戻り値として使用される `sum` 値
2. 振幅値 `amp`
3. 乗数 `m`

この乗数は、`lacunarity` を負の `dimension` で累乗したものです。その後、各オクターブのループに入ります。関数 n31 は、float3 の入力からパーリンノイズ値を返します。これは 0 から 1 の範囲で出力されるため、-1 から 1 の範囲に再マップし、現在の振幅値 `amp` を乗算した後で、累積 `sum` 値に加えます。次に、乗数定数値 `m` を乗算して `amp` 値を調整します。最後に、`lacunarity` パラメーターを乗算してサンプル位置 `p` を変更します。

マスグレイブノイズは、最小限の手入力でリアルかつ自然なテクスチャを生成できるため、コンピューターグラフィックスの分野では高く評価されています。さまざまなノイズ関数を重ね、変化させることで、現実世界の風景、雲、そして前述の木目の例のような有機的なマテリアルの複雑さを模した、入り組んだフラクタルなパターンを作り出します。



マスグレイブノイズを使用した Raymarch クラウド

プロシージャルノイズの利点

シェーダーでは、プロシージャルノイズを使用して、アニメーション化されたテクスチャや動的マテリアルを作成することもできます。例えば、水のシェーダーにノイズを加えると、リアルなさざ波の効果を生み出すことができます。ノイズを時間の経過とともにアニメーション化することで、水面が動いているように見え、滑らかさとリアルさを表現できます。プロシージャルノイズをカラーグラデーションや透過エフェクトと組み合わせると、雲や揺れる草など、多様で動的なマテリアルを作成できます。

プロシージャルノイズには、主にスケーラビリティビリティと多様性の面でいくつかの利点があります。大規模で多様な環境を手動で作成するには、多くの時間と大量のメモリが必要になります。プロシージャルノイズを使用すると、開発者は多くのストレージスペースを使用することなく、一貫性を保ちながらも毎回異なる複雑な環境を作成できます。さらに、プロシージャルノイズのアルゴリズムは決定論的であるため、同じシード値を使用すれば、必要に応じて同じ "ランダムな" 環境を再作成できます。

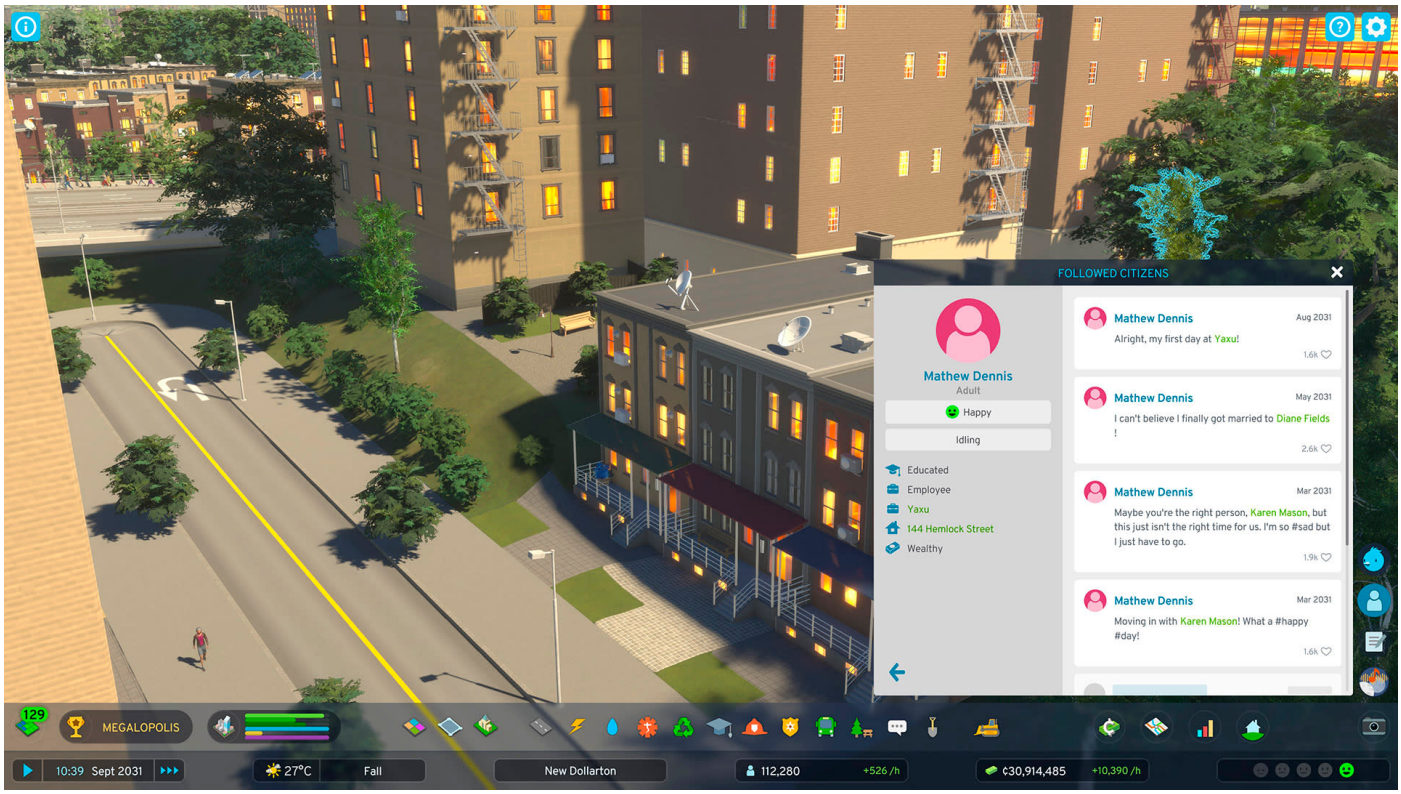
この手法により、無限の環境も実現できます。これは、連続的な地形が必要なゲーム (オープンワールドゲームやサバイバルゲームなど) で便利です。環境を連続的に生成することで、Unity は必要に応じて地形のセクションをロードおよびアンロードでき、メモリ使用量を抑えつつパフォーマンスの向上が可能です。

課題と最適化

プロシージャルノイズは、特に 3D アプリケーションや大規模な環境で使用する場合、計算負荷が高くなることがあります。パフォーマンスを最適化するには、ノイズ生成の解像度を制限したり、以前に計算したノイズ値をキャッシュに保存する設定を採用したりしてください。また、異なるスケールで異なるノイズ値をブレンドすると、複雑さが増すだけでなく、複数のディテールレイヤーを持つことでリアリティも向上します。

総じて、Unity におけるプロシージャルノイズは、プレイヤーのインタラクションに動的に適応する、多様でスケーラブルかつ魅力的な世界の創造に役立つ汎用的なツールです。

コンピュートシェーダー

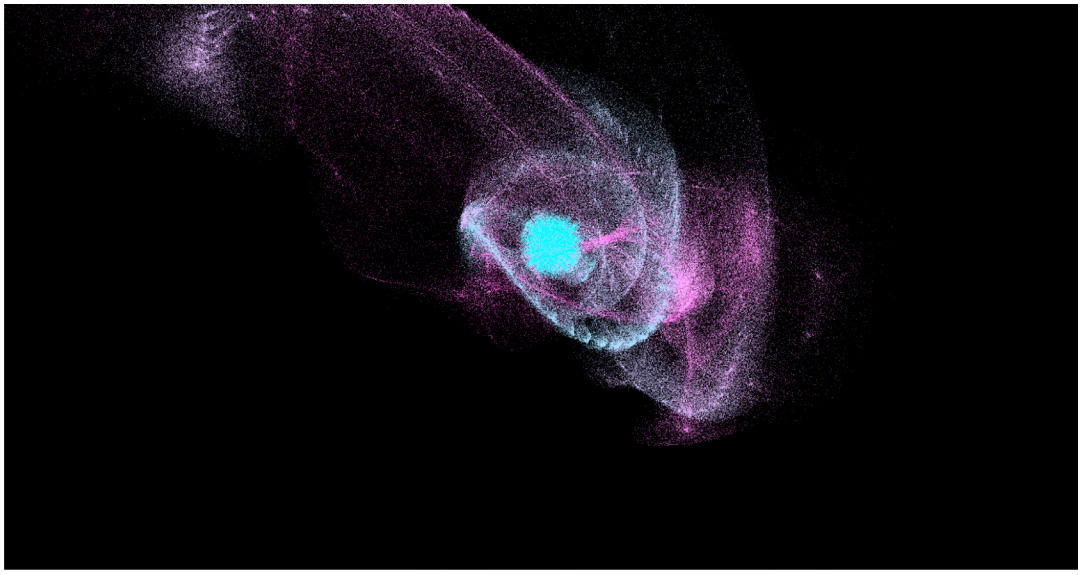


Colossal Order と Paradox Interactive による『Cities:Skylines II』のような複雑なシミュレーションをゲームに取り入れる場合には、コンピュートシェーダーを使用してこの作業の一部を GPU に処理させることを検討できます。

このレシピではコンピュートシェーダーを使用します。コンピュートシェーダーは、同じ計算を複数のエンティティに適用するような、計算負荷の高いタスクに使用できます。例として、パーティクルエフェクトとフロッキング (群れを作る) を見てみましょう。コンピュートシェーダーを初めて使用する場合は、このレシピの最後にあるリソースを確認してください。

Unity には、パーティクルシステムを作成するために Built-In Particle System と VFX Graph という 2 つのシステムが用意されていますが、このレシピでは独自のシステムを作成します。これは、インスタンス化されたメッシュで動作するシェーダーを作成するために必要な手法を理解するのに役立ちます。こうした手法を使用すれば、数万個のメッシュを使用した視覚効果も作成できるようになります。使用する GPU によっては、100 万もの低ポリゴンメッシュを扱うことも可能です。このような手法は、草、髪の毛、水、軍隊、群衆を作成する際に活用できます。

ParticleFun



ParticleFun のレシピの動作

Scenes > Compute Shaders > ParticleFun > ParticleFun を開き、Visual Studio Code で同じフォルダー内の `ParticleFun.cs`、`ParticleFun.compute`、`ParticleFun.shader` を開きます。プログラムを実行すると、パーティクルがマウス位置に向かって移動し、時間の経過とともに色が変化することがわかります。その前にコードを確認しましょう。まずは以下のスクリプトです。

```
public class ParticleFun :MonoBehaviour
{
    private Vector2 cursorPos;

    // 構造体
    struct Particle
    {
        public Vector3 position;
        public Vector3 velocity;
```

```
        public float life;
    }

    const int SIZE_PARTICLE = 7 * sizeof(float);

    public int particleCount = 1000000;
    public Material material;
    public ComputeShader shader;

    int kernelID;
    ComputeBuffer particleBuffer;

    int groupSizeX;
   RenderParams rp;
```

パーティクルは、position (位置)、velocity (速度)、life (寿命) の値を持ちます。個々のパーティクルのデータは 7 つの Float を使用しているため、パーティクルのサイズは Float のサイズの 7 倍になります。次に、ユーザーが Inspector で調整できる多数の public 変数を宣言しています。マテリアルは `ParticleFun.shader` を使用するパーティクルマテリアルで、シェーダーは `ParticleFun.compute` になります。

`Start` メソッドでは、`Init` を呼び出します。`Init` メソッドは各パーティクルを初期化します。位置は **同次クリップスペース** に設定されています。これは各軸における -w から w の間の値です。w は座標の 4 番目の成分です。w = 1 の場合、-1, -1, -1 は錐台付近の左下、1, 1, 1 は錐台の右上です。

```
Vector v = new Vector3();
v.x = Random.value * 2 - 1.0f;
v.y = Random.value * 2 - 1.0f;
v.z = Random.value * 2 - 1.0f;
v.Normalize();
v *= Random.value * 5;

particleArray[i].position.x = v.x;
particleArray[i].position.y = v.y;
particleArray[i].position.z = v.z;

particleArray[i].velocity.x = 0
particleArray[i].velocity.y = 0;
particleArray[i].velocity.z = 0;

particleArray[i].life = Random.value * 5.0f + 1.0f;
```

まず、x、y、z を -1 から 1 の間のランダム値に設定した Vector3 を作成します。次に、このベクトルを正規化します。ベクトルの長さを 1 に設定することを忘れないでください。長さを 5 に拡張します。このベクトルを使用して、パーティクル配列内の各パーティクルの位置を設定します。速度は 0、寿命は 1 から 6 の間のランダム値に設定されます。

```
// コンピュートバッファを作成する
particleBuffer = new ComputeBuffer(particleCount, SIZE_PARTICLE);

particleBuffer.SetData(particleArray);

// カーネルの ID を探す
kernelID = shader.FindKernel("CSParticle");

uint threadsX;
shader.GetKernelThreadGroupSizes(kernelID, out threadsX, out _, out _);
groupSizeX = Mathf.CeilToInt((float)particleCount / (float)threadsX);

// コンピュートバッファをシェーダーとコンピュートシェーダーにバインドする
shader.SetBuffer(kernelID, "particleBuffer", particleBuffer);
material.SetBuffer('"particleBuffer", particleBuffer);

rp = new RenderParams(material);
  rp.worldBounds = new Bounds(Vector3.zero, 10000*Vector3.one);
```

次のステップは、**ComputeBuffer** を作成することです。これには要素数 (count) と各要素のサイズ (stride) という 2 つのパラメーターがあります。次に、SetData メソッドを使用してバッファにデータを格納する必要があります。これにより、RAM から GPU メモリにデータが転送されます。**ComputeShader** からアクセスするには、すべてのデータが GPU メモリ内にある必要があります。ComputeShader 内のコードは、カーネルと呼ばれる特殊な種類の関数を使用して呼び出されます。各カーネルには一意の ID があり、関数名をパラメーターにして `FindKernel` メソッドを呼び出すことで、その ID を取得できます。

各カーネルには、x、y、z の 3 つのスレッドパラメーターがあります。コンピュートシェーダーの真髄は、それらの並行実行にあります。このパーティクルの例では、スレッドグループのサイズは 256、1、1 に設定されています。GPU から最高のパフォーマンスを引き出すには、実際のデバイスアーキテクチャについて知る必要があります。

C# スクリプトから、コンピュートシェーダーメソッド `GetKernelThreadGroupSizes` を使用してスレッドグループサイズにアクセスできます。すべてのパーティクルをカバーするスレッドを確保するには、以下のコードに示すように、カーネルをパーティクル数の回数ディスパッチする必要があります。

```
Mathf.CeilToInt((float)particleCount / (float)threadsX)
```

この例では、すべての処理が x スレッド内で行われます。particleBuffer がマテリアルとコンピュートシェーダーに渡されることに注意してください。

```
shader.SetBuffer(kernelID, "particleBuffer", particleBuffer);
material.SetBuffer("particleBuffer", particleBuffer);
```

これが、この例の基本トリックです。GPU に常駐する共有 ComputeBuffer は、コンピュートシェーダーと頂点フラグメントシェーダーの両方で使用できます。つまり、コンピュートシェーダーでバッファのコンテンツを操作しておき、オブジェクトを頂点フラグメントシェーダーでレンダリングする際に、レンダリングで同じバッファを使用します。

RenderParams インスタンスを初期化する必要があります。ここでは大きな Bounds インスタンスを設定するだけです。これは、Graphics.RenderPrimitives を使用してパーティクルを実際にレンダリングする際に必要になります。

Update メソッドを見てみましょう。ここでは、コンピュートシェーダーの deltaTime と mousePosition を設定します。次に、先ほど確認した kernelID を Dispatch (ディスパッチ) します。Dispatch の際には、x、y、z の各次元に対してワークグループの数を設定します。particleCount の回数だけカーネルを実行するため、x スレッドグループのサイズが 256 の場合は、groupSizeX を particleCount / 256 の浮動小数点の値を切り上げた整数値として事前に計算しておきます。切り上げとは、7 を 2 で除算したときに得られる浮動小数点値 3.5 を、次の整数である 4 にする処理です。x 次元に groupSizeX を使用することで、カーネルはインデックス 0 から particleCount-1 までの各スレッドで実行されます。particleCount が 256 のちょうどの倍数でない場合は、それを超えるインデックスのスレッドも実行されることになります。Dispatch が完了すると、particleBuffer に各パーティクルの新しい位置値が含まれます。

ここからは Graphics インターフェースのメソッドを使用しますが、その前に説明が必要です。このメソッド RenderPrimitives は、以下の 4 つのパラメーターを使用します。

- RenderParams インスタンス (最低限レンダリングする Bounds 領域を定義します)
- メッシュトポロジーのタイプ (この例では点をレンダリングしていますが、三角形の線をレンダリングすることも可能です)
- シングルインスタンスでの頂点数 (点の場合は常に 1 になります)
- インスタンス数 (この例ではパーティクル数です)

```
void Update()
    {

        float[] mousePosition2D = { cursorPos.x, cursorPos.y };

        // コンピュートシェーダーにデータを送信する
        shader.SetFloat("deltaTime", Time.deltaTime);
        shader.SetFloats("mousePosition", mousePosition2D);
```

```
        // パーティクルを更新する
        shader.Dispatch(kernelID, groupSizeX, 1, 1);

        Graphics.RenderPrimitives(rp, MeshTopology.Points, 1, particleCount );
    }
```

実際のレンダリングは、マテリアルにアタッチされているシェーダーによって処理されます。その点について見ていきましょう。

`ParticleFun.shader` ファイルでは、バッファへの参照を加える必要があります。これにはパーティクル構造体の定義が必要です。

```
struct Particle{
float3 position;
       float3 velocity;
       float life;
};
StructuredBuffer<Particle> particleBuffer;
```

シェーダーはこのバッファに書き込みを行わないため、バッファを `RWStructuredBuffer` ではなく `StructuredBuffer` として設定します。書き込みはコンピュートシェーダーが行います。

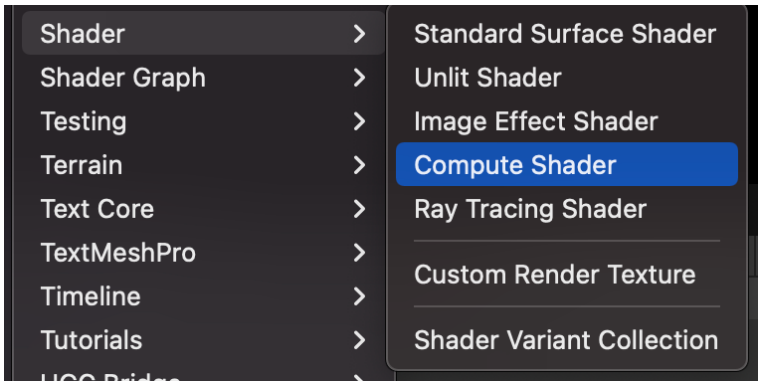
`_PointSize` プロパティがあります。`vert` 関数に渡されるインスタンスである Attributes 構造体には、`instanceID` プロパティがあることに注意してください。ポイントシェーダーの場合、instanceID は 0 からパーティクル数 -1 までの値に設定されます。この値をバッファのインデックスとして使用します。コンピュートシェーダーがこの位置値を更新するため、コンピュートシェーダーで位置を制御し、頂点フラグメントシェーダーをレンダリングに使用することになります。MeshTopology.Points を使用する場合、シェーダーはセマンティック PSIZE の入力パラメーターを点のピクセルサイズに設定する必要があります。ここでは、スクリプトによって渡された変数 `PointSize` を設定しています。

```
Shader "Custom/ParticleFun"
{
    Properties
    {
        _PointSize("Point size", Float) = 5.0
    }

    SubShader
    {
        Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }
```

```
        Pass
        {
            HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            struct Particle{
                float3 position;
                float3 velocity;
                float life;
            };

            StructuredBuffer<Particle> particleBuffer;

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

            CBUFFER_START(UnityPerMaterial)

            float _PointSize;
            CBUFFER_END


            struct Attributes
            {
                float4 positionOS   :POSITION;
                uint instanceID :SV_InstanceID;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };

            struct Varyings
            {
                float4 positionHCS  :SV_POSITION;
                float4 color :COLOR;
                float size:PSIZE;
            };

            Varyings vert(Attributes IN)
            {
                Varyings OUT;
```

```
                Particle particle = particleBuffer[IN.instanceID];

                // 色
                float lerpVal = particle.life * 0.25f;
                OUT.color = half4(1.0f - lerpVal+0.1, lerpVal+0.1, 1.0f, lerpVal);

                // 位置
                OUT.positionHCS = TransformObjectToHClip(particle.position);

                OUT.size = _PointSize;

                return OUT;
            }

            half4 frag(Varyings IN) :SV_Target
            {
                return IN.color;
            }
            ENDHLSL
        }
    }
}
```

それでは、コンピュートシェーダーに移りましょう。コンピュートシェーダーを作成するには、Project ウィンドウで右クリックして **Create > Shaders > Compute Shader** を選択します。



コンピュートシェーダーの作成

パーティクル用のバッファを定義する必要があります。そのためには、スクリプト内の構造体と一致する構造体が必要です。また、このシェーダーはバッファに書き込まれるため、RWStructuredBuffer を宣言する必要があります。

```
// パーティクルのデータ
struct Particle
{
       float3 position;
       float3 velocity;
       float life;
};

// シェーダーと共有するパーティクルのデータ
RWStructuredBuffer<Particle> particleBuffer;
```

CSParticle カーネルでは、以下のようになっています。

```
Particle particle = particleBuffer[id.x];

particle.life -= deltaTime;

float3 delta = float3(mousePosition.xy, 0) - particle.position;
float3 dir = normalize(delta);

particle.velocity += dir;
particle.position += particle.velocity * deltaTime;

particleBuffer[id.x] = particle;

if (particle.life < 0) respawn(id.x);
```

id.x をインデックスとして、バッファからパーティクルを取得します。このカーネルを Dispatch で実行する方法により、前述のように、0 から particleCount-1 までの値が設定されます。次に、パーティクルの寿命を減らします。スクリプトからコンピュートシェーダーに画面更新ごとに渡される、deltaTime プロパティを使用します。これは、最後の更新から経過した秒数を示します。Unity では新しくレンダリングされたフレームを 1 秒間に約 60 回表示することを目指しているため、これは非常に小さな値になります。したがって、deltaTime は約 16 ms、つまり 0.016 秒になります。

パーティクルからマウス位置までのベクトルを作成し、**z** 値を 0 に設定します。これは、パーティクルの z のデフォルト値です。マウス位置の値は、マウスのワールド空間の位置に設定されます。ParticleFun.cs ファイルでは、OnGUI イベントを使用します。

```
void OnGUI()
    {
        Vector3 p = new Vector3();
        Camera c = Camera.main;
        Event e = Event.current;
        Vector2 mousePos = new Vector2();

        // イベントからマウス位置を取得する
        // イベントでの y の位置が反転していることに注意する
        mousePos.x = e.mousePosition.x;
        mousePos.y = c.pixelHeight - e.mousePosition.y;

        p = c.ScreenToWorldPoint(new Vector3(mousePos.x, mousePos.y,
                                 c.nearClipPlane ));

        cursorPos.x = p.x;
        cursorPos.y = p.y;
    }
```

このコードはマウスのクリックを検出し、座標を画面上の位置に変換します。スクリーン座標では bottom を 0 として使用するのに対し、`Event.mousePosition` は top を 0 として使用します。y 値を反転させるには、カメラの `pixelHeight` から `mousePosition.y` を減算します。次に、カメラのメソッド `ScreenToWorldPoint` を使用します (このメソッドには z 値が必要です)。最後に、`nearClipPlane` よって適切な結果が得られます。

マウス位置をワールド座標に変換する方法の詳細については、Game Dev Beginner の こちらの 投稿 を参照してください。

これで、パーティクルを加速させてマウス位置から遠ざけるためのベクトルが得られました。次に、deltaTime によって調整されたパーティクルの速度を使用します。

パーティクルの寿命が 0 未満の場合、パーティクルを再生成し、高速ランダム関数を使用して必要なランダム値を生成します。XorShift 乱数ジェネレーターは、George Marsaglia 氏によるリアルタイムグラフィックスにおける偉大な発見でした。

パーティクルは、マウス位置を中心とし、z = 0 にある半径 0.8 のスフィア内に配置されます。新しいパーティクルの寿命が 4 秒にリセットされ、速度は 0 にリセットされます。

```
void respawn(uint id)
{
       rng_state = id;
       float tmp = (1.0 / 4294967296.0);
       float f0 = float(rand_xorshift()) * tmp - 0.5;
       float f1 = float(rand_xorshift()) * tmp - 0.5;
       float3 normalF3 = normalize(float3(f0, f1, 0.0)) * 0.8f;
       normalF3 *= float(rand_xorshift()) * tmp;
       particleBuffer[id].position = float3(normalF3.x + mousePosition.x, normalF3.y +
mousePosition.y, 0.0);
       // このパーティクルの寿命をリセットする
       particleBuffer[id].life = 4;
       particleBuffer[id].velocity = float3(0,0,0);
}
```

life プロパティを使用して、パーティクルの色を制御してみましょう。頂点フラグメントシェーダーの `vert` 関数に戻り、以下のコードを色の割り当てに加えます。

```
       float life = particleBuffer[instance_id].life;
       float lerpVal = life * 0.25f;
       o.color = fixed4(1.0f - lerpVal+0.1, lerpVal+0.1, 1.0f, lerpVal);
```

これでパーティクルの色が変わります。lerp 値は 1 から 0 の間の値になります。これは respawn 関数によって life が 4 に設定され、その値に 0.25 を乗算するためです。赤チャンネルは 0.1 から始まり、寿命が減少するにつれて 1.1 に増加します。緑チャンネルは 1.1 から始まり、時間の経過とともに 0.1 に減少します。青は常に 1 で、アルファは時間の経過とともに減少し、ピクセルはフェードアウトします。

この例は、同じアセットの複数のインスタンスをレンダリングする際に、コンピュートシェーダーと頂点フラグメントシェーダーを組み合わせることがいかに有用かを示しています。ここでは、単一のピクセル色値という最もシンプルなアセットが使用されます。次の例は、このコンセプトを拡張して、複数のメッシュオブジェクトをレンダリングする方法を紹介します。

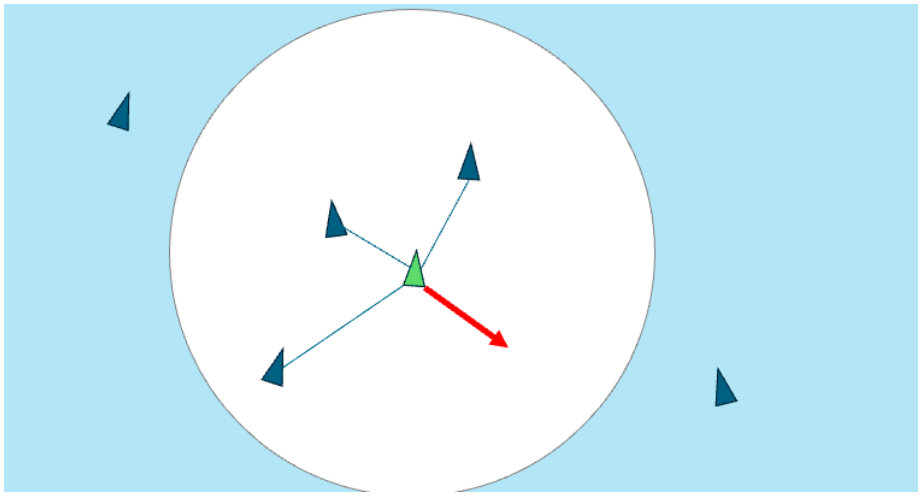
メッシュオブジェクトを加える

インスタンス化されたフロッキングシーン

このメッシュオブジェクト例では、フロッキングを実装しています。フロッキング (群れを作る) というアイデアは、飛んでいる鳥を観察し、群れの動きはシンプルなルールで制御できるという結論から生まれました。1 つ目のルールは、"鳥は近くて視界に入る小さな群れの動作しか認識していない" というものです。これは、『Siggraph 87』で Craig Reynolds 氏が発表した論文で初めて提案されました。同氏は群れのメンバーを Boid (ボイド) と呼びました。これは、コンピューターでシミュレートされた群れの個々のメンバーを指す一般的な用語として定着しています。ボイドは "Bird-oid" (鳥に似た) オブジェクトの略称です。

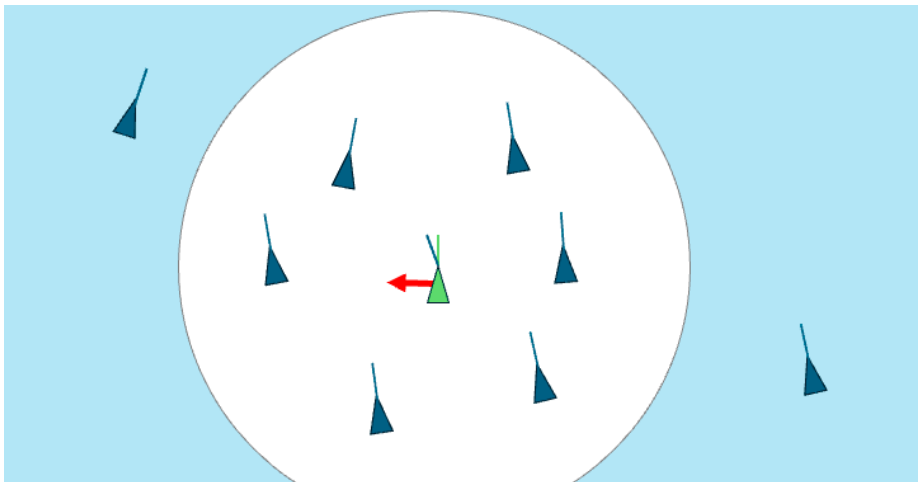
ユーザー定義の一定の半径内にいて、視界に入る群れのメンバーをスキャンしたら、以下の 3 つのルールを使用して各ボイドの位置、向き、速度を調整します。

1. **分離**:他のローカルボイドと密集しないように回避行動を取ります。これを実現するには、現在のボイドから他のローカルボイドへのベクトルとは反対方向のベクトルを求めます。



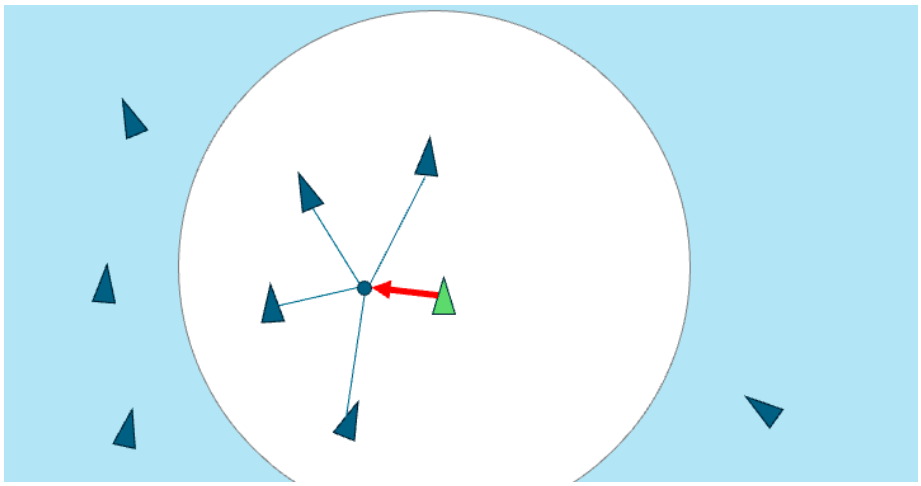
画像: Craig Reynolds 氏『*Boids:Background and Update*』

2. **整列**:ローカルボイドの平均方向に合わせて向きを調整します。現在の方向をローカルボイドの平均方向で線形補間します。



画像: Craig Reynolds 氏『*Boids:Background and Update*』

3. **結合**:ローカルボイドの平均位置に向かって移動します。ローカルボイドの平均位置を求め、そこに向かって移動します。



画像: Craig Reynolds 氏『*Boids:Background and Update*』

分離と結合は相反すると思われるかもしれませんが、分離では現在のボイドから他のボイドへのベクトルを扱うのに対し、結合ではボイドグループの平均位置を計算して考慮していることを覚えておいてください。

具体例を見てみましょう。

Scenes > ComputeShaders > Instanced からシーン **InstancedFlocking** を開き、VS Code で同じフォルダー内にある `InstancedFlocking.cs` と `InstancedFlocking.compute` を開きます。この例では、別の ComputeBuffer を使用しますが、コンピュートシェーダーに焦点を当てます。

ボイドには、位置、方向、ノイズオフセットの各値があります。この構造体にはコンストラクターメソッドが加えられていますが、構造体内のデータは 2 つの Vector3 と 1 つの Float のみです。

```
public struct Boid
    {
        public Vector3 position;
        public Vector3 direction;
        public float noise_offset;

        public Boid(Vector3 pos, Vector3 dir, float offset)
        {
            position.x = pos.x;
            position.y = pos.y;
            position.z = pos.z;
            direction.x = dir.x;
            direction.y = dir.y;
            direction.z = dir.z;
            noise_offset = offset;
        }
    }
```

ユーザーが群れの動作を調整できる public プロパティはいくつかあります。シェーダーをコーディングしていく中で、それぞれを順番に確認していきます。

`InitBoids` は、ボイドの配列を作成および入力します。位置はスフィア内のランダムな値です。

`InitShader` は、ボイドの配列から ComputeBuffer を作成および設定し、コンピュートシェーダーにいくつかのプロパティを設定します。

`Update` メソッドは時間とデルタタイムを設定し、カーネルをディスパッチします。コンピュートシェーダーで各ボイドの新しい位置と方向を計算したら、`RenderMeshIndirect` を使用して実際にボイドをレンダリングします。`RenderMeshIndirect` には、`RenderParams` インスタンス、メッシュ、`GraphicsBuffer` インスタンスが必要です。`RenderParams` インスタンスは `Start` メソッドで初期化されます。

```
renderParams = new RenderParams(boidMaterial);
renderParams.worldBounds = new Bounds(Vector3.zero, Vector3.one * 1000);
```

コンストラクターにマテリアルが渡されることに注意してください。このマテリアルはインスタンシングをサポートしている必要があります。ここで使用する方法は、URP Lit シェーダーを編集することです。そのためには、フォルダー (**Library/PackageCache/com.unity.render-pipelines.universal/Shaders**) から以下のファイルをコピーします。

— Lit.shader

— LitForwardPass.hlsl

— ShadowCasterPass.hlsl (シャドウをサポートする場合)

これらのファイルをまとめてサブフォルダーに格納します。

VS Code で **Scenes > ComputeShaders > Instanced > Shader** を開いてみましょう。PackageCache フォルダーからコピーした 3 つのファイルがあることが注目してください。Lit.shader に対する変更は最小限です。シェーダー名が変更されています。

```
Shader "Custom/Flocking/Instanced"
{
    …
```

また、LitForwardPass.hlsl へのパスが単一のドット "." に変更されています。これは、Lit.shader ファイルと同じフォルダー内のファイルを使用することを意味します。LitForwardPass.hlsl ファイルで、Boid 構造体と boidsBuffer を定義します。

```
struct Boid
{
    float3 position;
    float3 direction;
    float noise_offset;
};

StructuredBuffer<Boid> boidsBuffer;
```

`create_matrix` 関数を加えます。これにより、位置、方向、上向きベクトルから回転と位置のマトリックスが作成されます。

```
float4x4 create_matrix(float3 pos, float3 dir, float3 up) {
    float3 zaxis = normalize(dir);
    float3 xaxis = normalize(cross(up, zaxis));
    float3 yaxis = cross(zaxis, xaxis);
    return float4x4(
        xaxis.x, yaxis.x, zaxis.x, pos.x,
        xaxis.y, yaxis.y, zaxis.y, pos.y,
        xaxis.z, yaxis.z, zaxis.z, pos.z,
        0, 0, 0, 1
    );
}
```

Attributes 構造体に SV_instanceID 要素を加えることが重要です。これにより、頂点シェーダーで instanceID にアクセスできるようになります。

```
struct Attributes
{
    float4 positionOS   :POSITION;
    float3 normalOS     :NORMAL;
    float4 tangentOS    :TANGENT;
    float2 texcoord     :TEXCOORD0;
    float2 staticLightmapUV   :TEXCOORD1;
    float2 dynamicLightmapUV  :TEXCOORD2;
    uint instanceID :SV_InstanceID;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Lit.shader では、関数 LitPassVertex を頂点シェーダーとして定義しています。この関数は、フォワードレンダーパイプライン用の LitForwardPass.hlsl にあります。

これで、入力 Attributes パラメーターに instanceID が加わったので、boidsBuffer から個々のボイドを取得できます。create_matrix 関数とボイドの位置と方向を使用してマトリックスを作成します。このファイルのデフォルトバージョンにある、GetVertexPosition に渡されるパラメーターを編集し、input.positionOS に先ほど作成したマトリックスを乗算します。GetVertexPosition では float3 が想定されているため、mul 関数の外側に .xyz を加えることで float4 の出力を float3 に変換します。NormalInput にも同じことを行います。

```
Boid boid = boidsBuffer[input.instanceID];

float4x4 mat = create_matrix(boid.position, boid.direction, float3(0.0, 1.0, 0.0));

VertexPositionInputs vertexInput = GetVertexPositionInputs(mul(mat, input.positionOS).xyz);

VertexNormalInputs normalInput = GetVertexNormalInputs(mul(mat, input.normalOS), mul(mat,
input.tangentOS));
```

これで、使用するマテリアルができました。これを使用して複数のメッシュオブジェクトを表示する方法を見てみましょう。

スクリプトには、新たにバッファ argsBuffer が加わっています。これはグラフィックスバッファであり、レンダリング時に頂点フラグメントシェーダーがこのバッファを使用します。このバッファを初期化するコードを加える必要があります。

バッファの作成時に、タイプを IndirectArguments、配列内の要素を 1、要素のサイズを IndirectDrawIndexedArgs のサイズとして設定します。そして、IndirectDrawIndexedArgs の単一の配列を作成します。GetIndexCount メソッドを使用して indexCountPerInstance を boidMesh の頂点数に設定し、instanceCount を numOfBoids プロパティに設定します。

SetData メソッドを使用して、argsBuffer にデータをコピーします。これで、データが GPU に常駐します。

```
argsBuffer = new GraphicsBuffer(GraphicsBuffer.Target.IndirectArguments, 1, GraphicsBuffer.
IndirectDrawIndexedArgs.size);
GraphicsBuffer.IndirectDrawIndexedArgs[] data = new GraphicsBuffer.IndirectDrawIndexedArgs[1];
data[0].indexCountPerInstance = boidMesh.GetIndexCount(0);
data[0].instanceCount = (uint)numOfBoids;
argsBuffer.SetData(data);
```

これに続くのが Update メソッドです。コンピュートシェーダーで Time と deltaTime を設定し、Dispatch を実行します。ボイドの位置と方向が計算されたら、RenderMeshIndirect を使用してボイドをレンダリングします。

```
void Update()
    {
        shader.SetFloat("time", Time.time);
        shader.SetFloat("deltaTime", Time.deltaTime);

        shader.Dispatch(this.kernelHandle, groupSizeX, 1, 1);

        Graphics.RenderMeshIndirect( renderParams, boidMesh, argsBuffer );
    }
```

このスクリプトについて知っておくべきことは以上です。次に取り上げるのは、コンピュートシェーダーのフロッキングコードです。

ここでは、先ほど説明した 3 つのシンプルなルールである分離、整列、結合を適用する必要があります。まず、id.x に基づいてバッファからボイドを取得します。次に、分離、整列、結合の初期値を設定します。これらの値を計算する際には、近くのボイドのみを考慮します。当然、現在のボイドもこの半径内にあるため、nearbyCount は 0 ではなく 1 から始まります。ループが現在のボイドを指している場合は、そのボイドを無視できます。

更新を適用するには、i が id.x と異なる必要があります。次に、i 変数が指すボイドを取得します。現在のボイドと一時ボイドまでの距離が、近傍距離である neighbourDistance プロパティよりも近いかどうかを確認するチェックがもう 1 つあります。

```
Boid boid = boidsBuffer[id.x];

float3 separation = 0;
float3 alignment = 0;
float3 cohesion = flockPosition;
```

```
uint nearbyCount = 1; // 自身を加算する（ループ内では無視される）

for (int i = 0; i < boidsCount; i++)
{
  if (i != int(id.x))
  {
    Boid tempBoid = boidsBuffer[i];
    if (distance(boid.position, tempBoid.position) < neighbourDistance){
      // 位置の計算はここに記述する
    }
  }
}
```

最も複雑な計算は分離です。分離については、このセクションで後ほど詳しく説明しますが、まずは整列と結合について見てみましょう。整列では各ボイドの方向の合計、結合では各ボイドの位置の合計を求めます。結合の変数 cohesion の値は flockPosition (群れの位置) を持っています。

```
alignment += tempBoid.direction;
cohesion += tempBoid.position;
nearbyCount++;
```

次に、ループの外で `nearbyCount` 値を使用して、累積値を `nearbyCount` 値で除算することで整列と結合の各値の平均を取得します。結合については、現在のボイドの位置を減算して結果を正規化する必要があります。これで、3 つのプロパティの合計としてターゲット方向が得られます。ただし、新しく計算された方向を適用する際は、既存のボイド方向とのブレンド比率を非常に低く抑えるようにします。lerp を使用すると、計算された方向の 6% のみが、既存の方向の 94% に対してブレンドされます。

これで、ボイド方向 (回転値ではなくフォワードベクトル) が取得できました。これを使用して、スピードと `deltaTime` を乗算することで位置を更新できます。最後のステップは、更新結果をバッファに適用することです。

```
float avg = 1.0 / nearbyCount;
alignment *= avg;
cohesion *= avg;
cohesion = normalize(cohesion - boid.position);

float3 direction = alignment + separation + cohesion;

boid.direction = lerp(direction, normalize(boid.direction), 0.94);
boid.position += boid.direction * boidSpeed * deltaTime;

boidsBuffer[id.x] = boid;
```

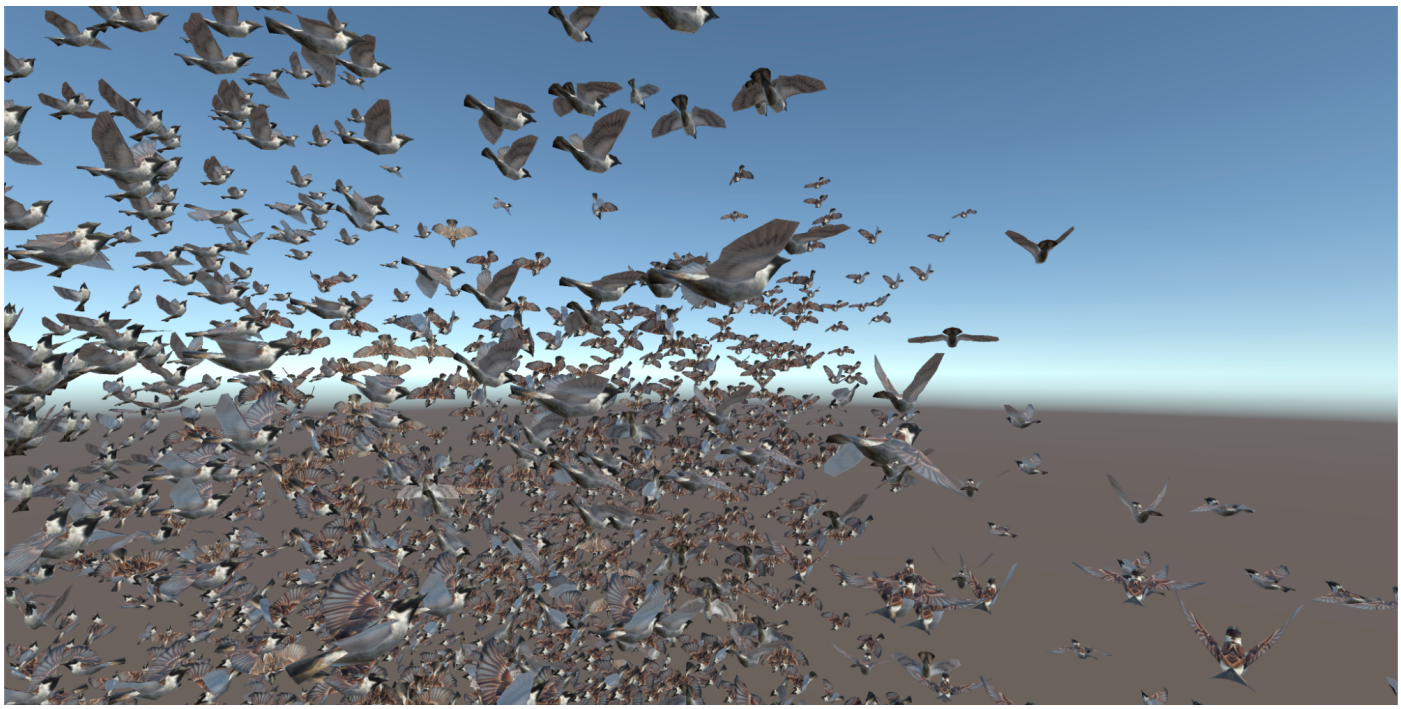
現在のプログラムを実行すると、分離値をまだ適用していないため、すべてのボイドが 1 つにマージされます。これを修正しましょう。

for ループに戻り、まず一時ボイドから現在設定中のボイドへのベクトルと、その長さを取得します。一時ボイドが近傍距離として設定された値よりも遠くにいる場合は、無視します。この距離よりも近くにいる場合は、分離値にそれを考慮する必要があります。オフセットベクトルを使用して、設定中のボイドに近づくにつれて増加する値でスケールします。1/dist を使用しているため、dist が 0 にならないようにする必要があります。つまり、0 による除算は避けてください。

では、1/dist - 1/neighbourDistance とはどのような意味でしょうか。dist が小さいときに何が起こるか考えてみましょう。1/dist は非常に大きくなり、1/neighbourDistance の小さな値は無視できます。このボイドが近傍距離の境界付近にいる場合は、ほぼ 1/neighbourDistance - 1/neighbourDistance の値、つまり 0 になります。そのため、ボイドがターゲットボイドに近い場合は、分離ベクトルが大幅に大きくなり、近傍距離の値に近くなると無視されます。

```
float3 offset = boid.position - tempBoid.position;
float dist = length(offset);
if (dist<neighbourDistance){
       dist = max(dist, 0.000001);
       separation += offset * (1.0/dist - 1.0/neighbourDistance);
}
```

これで、フロッキングの例が完成しました。数秒後にすべてのボイドが集合し、群れの位置を中心に旋回するようになります。



SkinnedFlocking シーン

最後の例では、Skinned Mesh Renderer を使用します。**Scenes > ComputeShaders > Skinned** フォルダー内の SkinnedFlocking シーンを開いて実行します。たくさんの鳥が飛び回っているのが確認できます。VS Code で Scripts フォルダー内の `SkinnedFlocking.cs` を開きます。Shader フォルダー内の SkinnedFlocking.compute と LightForwardPass.hlsl も開きます。

まずはスクリプトから見ていきましょう。基本的には前の例と同じですが、いくつかの変更があります。まず、Boid 構造体に frame プロパティがあります。これは、表示するアニメーションフレームを選択するために使用します。また、ここでは `numOfFrames` プロパティがあります。これはコンピュートシェーダーとカスタム Lit シェーダーに渡されます。スクリプトでは、シェーダーの `EnableKeyword` メソッドと `DisableKeyword` メソッドを使用して、シェーダー内の定義を加えたりまたは削除したりします。この例では 3 つ目のバッファである vertexAnimationBuffer があり、新しいメソッドである `GenerateVertexAnimationBuffer` もあることがわかります。このメソッドについて見ていきましょう。

プログラムを開始する前に、Animator コンポーネントを使用してメッシュを一連のポーズに設定します。次に、そのポーズの頂点位置をバッファに格納します。このバッファから適切なインデックスを選択することで、メッシュのさまざまなポーズを連続して表示できます。最初に必要なのは Animator コンポーネントです。Unity のアニメーションではレイヤーを使用できますが、ここではシンプルにして 1 番目のレイヤーのみを使用します。

```
animator = boidObject.GetComponentInChildren<Animator>();
int iLayer = 0;
```

アニメーターがアニメーションクリップ FlapWings を自動的に再生するように設定されていることがわかります。ポーズを設定するには、ステート情報、ポーズを保存する新しいメッシュ、いくつかの変数が必要です。

```
AnimatorStateInfo aniStateInfo = animator.GetCurrentAnimatorStateInfo(iLayer);

       Mesh bakedMesh = new Mesh();
       float sampleTime = 0;
       float perFrameTime = 0;
```

このスクリプトには、FlapWings アニメーションに設定された `animationClip` という public プロパティがあります。ベイクするフレーム数を決定するには、AnimationClip に含まれているフレームレートと長さのプロパティを使用します。この値は、頂点に使用する `vertexAnimationBuffer` のインデックスを設定するために頻繁に使用されるため、2 の累乗にすると効率的です。`Mathf` オブジェクトには、この値を設定する便利なメソッドがあります。これで、個々のフレームの継続時間を取得できます。

```
numOfFrames = Mathf.ClosestPowerOfTwo((int)(animationClip.frameRate * animationClip.length));
       perFrameTime = animationClip.length / numOfFrames;
```

メッシュの頂点数が必要になります。ここで、頂点データを格納するための Vector4 配列を設定できます。配列の長さは頂点数 × numOfFrames になります。これで頂点データを取得する準備が整いました。これは for ループで実行されます。アニメーションの再生、レイヤーと開始時刻の設定に使用できる aniStateInfo オブジェクトを取得したことを思い出してください。次に、deltaTime を 0 にして update を呼び出します。これにより、メッシュがこのアニメーションで定義された位置に更新されます。そして、メッシュを bakedMesh オブジェクトにベイクし、頂点を反復処理して、その値を頂点配列に格納します。

```
var vertexCount = boidSMR.sharedMesh.vertexCount;

       Vector4[] vertexAnimationData = new Vector4[vertexCount * numOfFrames];
       for (int i = 0; i < numOfFrames; i++)
       {
              animator.Play(aniStateInfo.shortNameHash, iLayer, sampleTime);
              animator.Update(0f);

              boidSMR.BakeMesh(bakedMesh);
              //ここで頂点配列に頂点データを格納する
              for(int j = 0; j < vertexCount; j++)
              {
                     Vector4 vertex = bakedMesh.vertices[j];
                     vertex.w = 1;
                     vertexAnimationData[(j * numOfFrames) +  i] = vertex;
              }

              sampleTime += perFrameTime;
       }
```

この時点で、ComputeBuffer の設定に使用できる頂点の配列が作成されます。このバッファをマテリアルに渡します。コンピュートシェーダーはこのバッファを必要とせず、シェーダーのみを必要とします。このバッファは読み取りのみに使用されます。

```
vertexAnimationBuffer = new ComputeBuffer(vertexCount * numOfFrames, 16);
vertexAnimationBuffer.SetData(vertexAnimationData);
boidMaterial.SetBuffer("vertexAnimation", vertexAnimationBuffer);
```

次に、コンピュートシェーダーがどのような影響を受けているかを見てみましょう。

現在のスピード、deltaTime、boidFrameSpeed プロパティを使用して、Boid 構造体の frame プロパティを調整します。移動速度の速い鳥ほど、羽ばたきも速くなります。frame プロパティが numOfFrames プロパティを超えていないかどうかを確認します。超えていた場合はこの値を減算し、boid.frame が 0 から numOfFrames の範囲になるようにします。

```
boid.frame = boid.frame + velocity * deltaTime * boidFrameSpeed;
if (boid.frame >= numOfFrames) boid.frame -= numOfFrames;
```

次に、シェーダーを確認しましょう。頂点シェーダー関数に渡される Attributes インスタンスには、`vertexID` と `instanceID` が含まれていることが不可欠です。

```
struct Attributes
{
    float4 positionOS   :POSITION;
    float3 normalOS     :NORMAL;
    float4 tangentOS    :TANGENT;
    float2 texcoord     :TEXCOORD0;
    float2 staticLightmapUV   :TEXCOORD1;
    float2 dynamicLightmapUV  :TEXCOORD2;
    uint instanceID :SV_InstanceID;
    uint vertexID :SV_VertexID;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

`LitVertexPass` 関数では、`instanceID` を使用して boidsBuffer からボイドを取得します。FRAME_INTERPOLATION が定義されていない場合、`positionOS.xyz` をインデックス vertexID * numFrames + boid.frame の vertexAnimation バッファに設定します。

FRAME_INTERPOLATION が定義されている場合、現在の boid.frame 値と次のフレームの値を lerp でブレンドした値を、`positionOS.xyz` に設定します。ブレンド値は、boid.frame の小数部分によって決まります。小数部分が 0 の場合、ブレンドはフレーム値になります。小数部分が 0.5 の場合、現在のフレームと次のフレームがそれぞれ 50% ずつ使用された線形補間になります。

```
Boid boid = boidsBuffer[input.instanceID];

    #ifdef FRAME_INTERPOLATION
        uint next = boid.frame + 1;
        if (next >= numOfFrames) next = 0;
        float frameInterpolation = frac(boidsBuffer[input.instanceID].frame);
        input.positionOS.xyz = lerp(vertexAnimation[input.vertexID * numOfFrames + boid.
frame], vertexAnimation[input.vertexID * numOfFrames + next], frameInterpolation);
    #else
        input.positionOS.xyz = vertexAnimation[input.vertexID * numOfFrames + boid.frame];
    #endif
```

アプリを再生すると、鳥が羽ばたくスピードがその速度に応じて変化する様子が確認できます。これを行うには、以下のステップに従います。

1. 鳥が羽ばたくアニメーションを実行するための、さまざまなポーズを格納した頂点バッファを用意します。
2. コンピュートシェーダーで、Boid 構造体の frame プロパティに 0 から numOfFrames の間の値を設定します。
3. このプロパティの更新スピードは、鳥の速度によって決まります。
4. サーフェスシェーダーでは、Boid 構造体の frame プロパティを使用して、現在のフレームと次のフレームを取得します。
5. 頂点シェーダーでは、vertexID と現在および次のフレームの値を組み合わせて、頂点バッファから適切な値を取得します。

このレシピでは、コンピュートシェーダーを使用してレンダリングされる単一の点を配置する方法から、アニメーション化された複数のメッシュをレンダリングする方法へと発展させています。コンピュートシェーダーは、計算負荷の高いプロセスのパフォーマンスを大幅に向上させるため、効果的に使用する方法について学ぶ価値があります。

その他のリソース

コース「Learn to Write Unity Compute Shaders」

まとめ

このガイドの冒頭で述べたように、e-book『*上級　Unity　クリエイター 向けの ユニバーサル レンダー パイプライン（URP）　入門* 』は、経験豊富な Unity 開発者やテクニカルアーティストが URP で利用可能な最新機能を最大限に活用するのに役立つ貴重なガイドです。

Unity の上級者向けの技術系 e-book はすべて、Unity　のベスト プラクティス ハブ から入手できます。e-book は、上級者向けの 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®