インテルの x86 プラットフォーム向け Unity* 最適化ガイド: パート 4

この記事は、インテル® デベロッパー・ゾーンに公開されている「Unity* Optimization Guide for Intel x86 Platforms: Part 4」の日本語参考訳です。

目次

チュートリアルのパート 3 に戻る:
インテルの x86 プラットフォーム向け Unity* 最適化ガイド: パート 3

エディターの最適化

圧縮されたテクスチャー

必要以上に高解像度のテクスチャーは、モバイルゲームや性能の低いハードウェアではボトルネックになります。シーンで圧縮されたテクスチャーを使用していること、および [Generate Mip Maps (ミップマップを生成する)] チェックボックスをオンにしてミップマップを有効にしていることを常に確認します。ミップマップは前述の LOD システムに似ていますが、テクスチャーの解像度を制御します。描画しているオブジェクトがカメラから遠く離れている場合、オブジェクトは数ピクセルで表現できるため、1024 x 1024 テクスチャーを使用して詳細に表現する必要はありません。

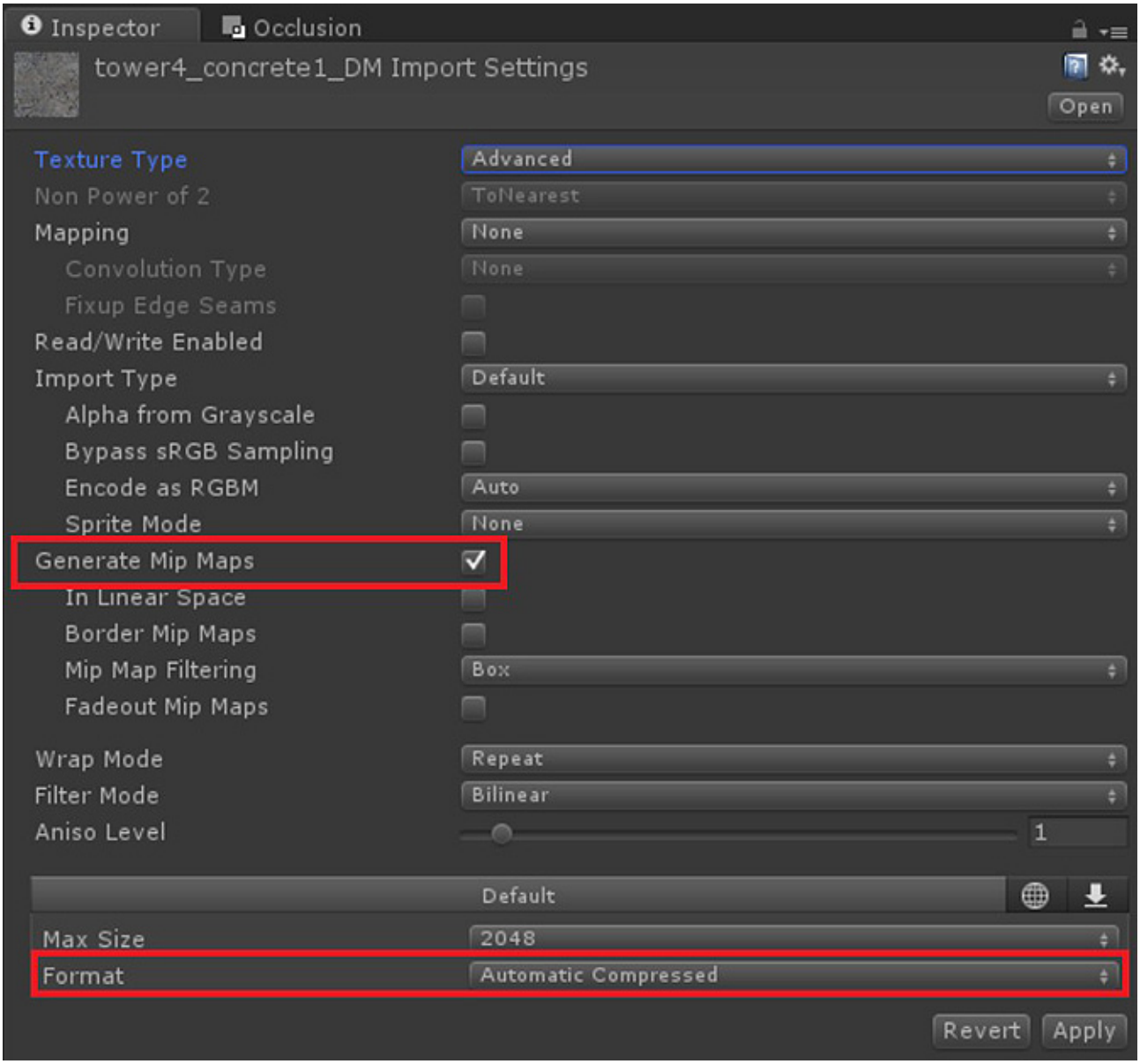

図 44. 選択したテクスチャーの [Inspector (インスペクター)] タブで圧縮されたテクスチャーとミップマップの生成を選択

インテル® GPA でフレームキャプチャーを行い、調査する描画呼び出しの **[Texture (テクスチャー)]** タブで、テクスチャーが圧縮されミップマップが生成されていることを確認します。ミップマップを生成すると、追加のデータが生成されるため、場合によってはパフォーマンスが低下します。アプリケーションで、このオプションを常に確認してください。

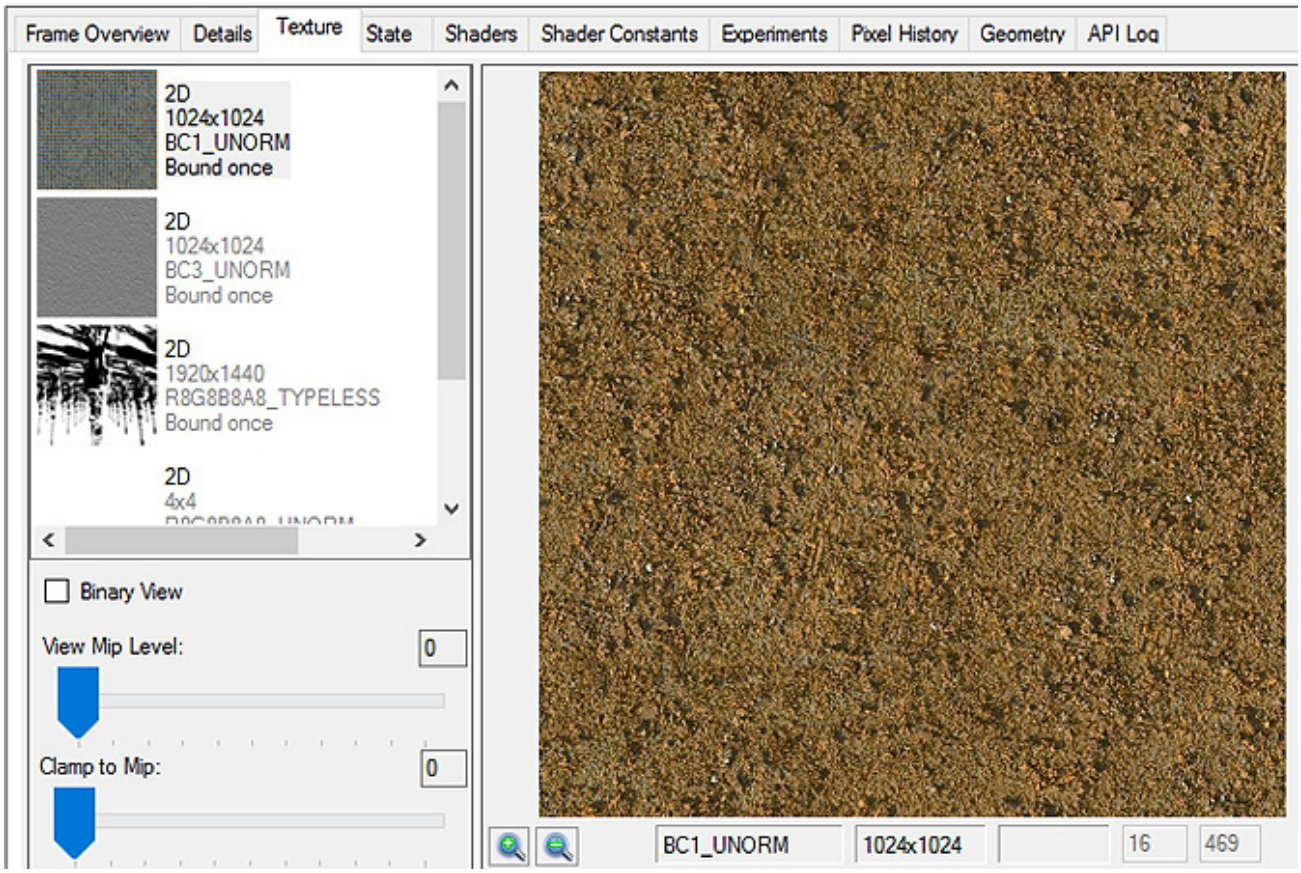

図 45. インテル® GPA Frame Analyzer でテクスチャーの 4 番目の MIP レベルとフォーマットを表示

モバイル・ストックシェーダー

モバイル・アプリケーションでストックシェーダーを使用する場合、Unity* により提供されるモバイルバージョンに切り替えて、精度が低い浮動小数点値とモバイル固有の最適化が使用されていることを確認します。マテリアルを選択してドロップダウン・メニューの [Mobile (モバイル)] セクションを検索し、アプリケーションで試します (図 46)。

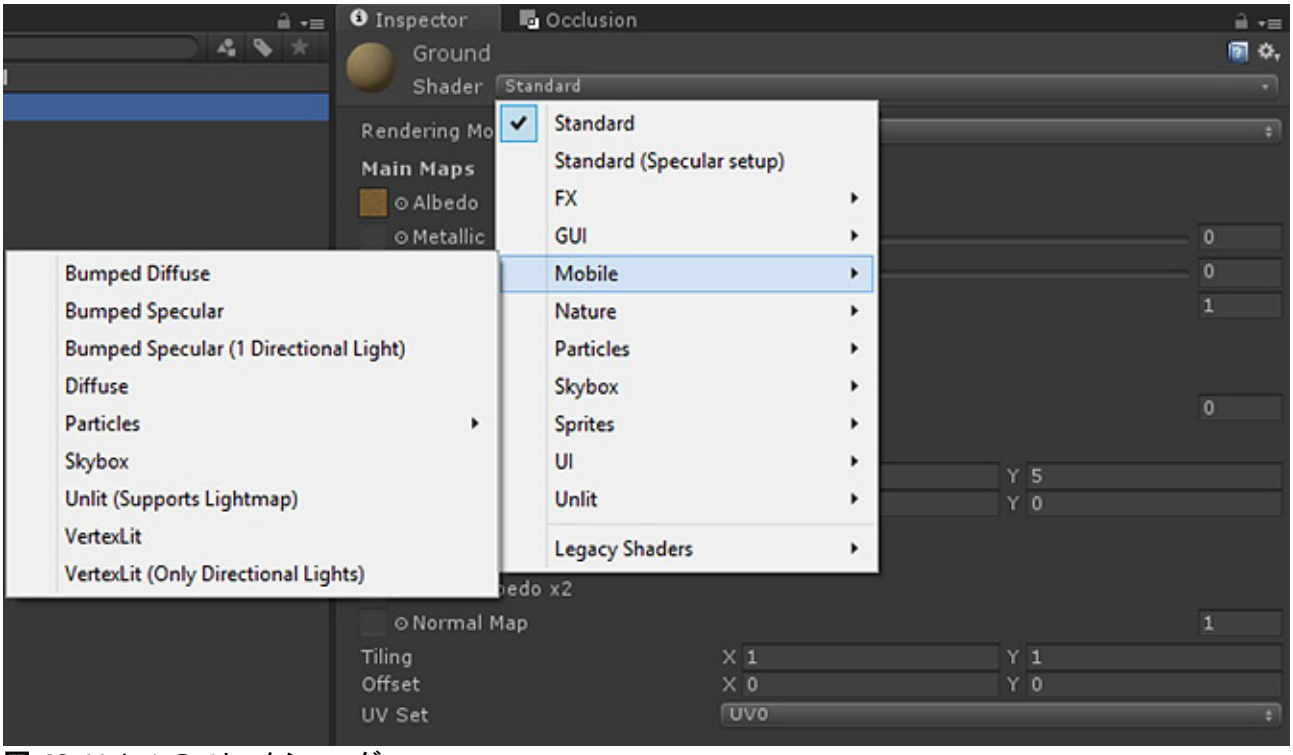

図 46. Unity* のストックシェーダー

スタティック・バッチ

Unity* 5.3 では、複数のスレッドで 1 つのバッファーにジオメトリーを作成し、バッファーとステートをバインドした後、バッファーの各範囲に対して複数の描画呼び出しを行うように、バッチシステムが変更されました。以前の Unity* では、すべてのジオメトリー・データを合成した後に単一の描画呼び出しを行っていました。新しいバージョンのバッチでは、描画呼び出しの数は減っていませんが、時間を短縮するため、描画に必要なステート変更の数が減っています。[File (ファイル)] > [Build Settings (ビルド設定)] > [Player Settings (プレーヤー設定)] を選択した後、[Static Batching (スタティック・バッチ)] オプションをオンにして、スタティック・バッチを有効にします。Unity* は、これらの設定をデフォルトで有効にします。

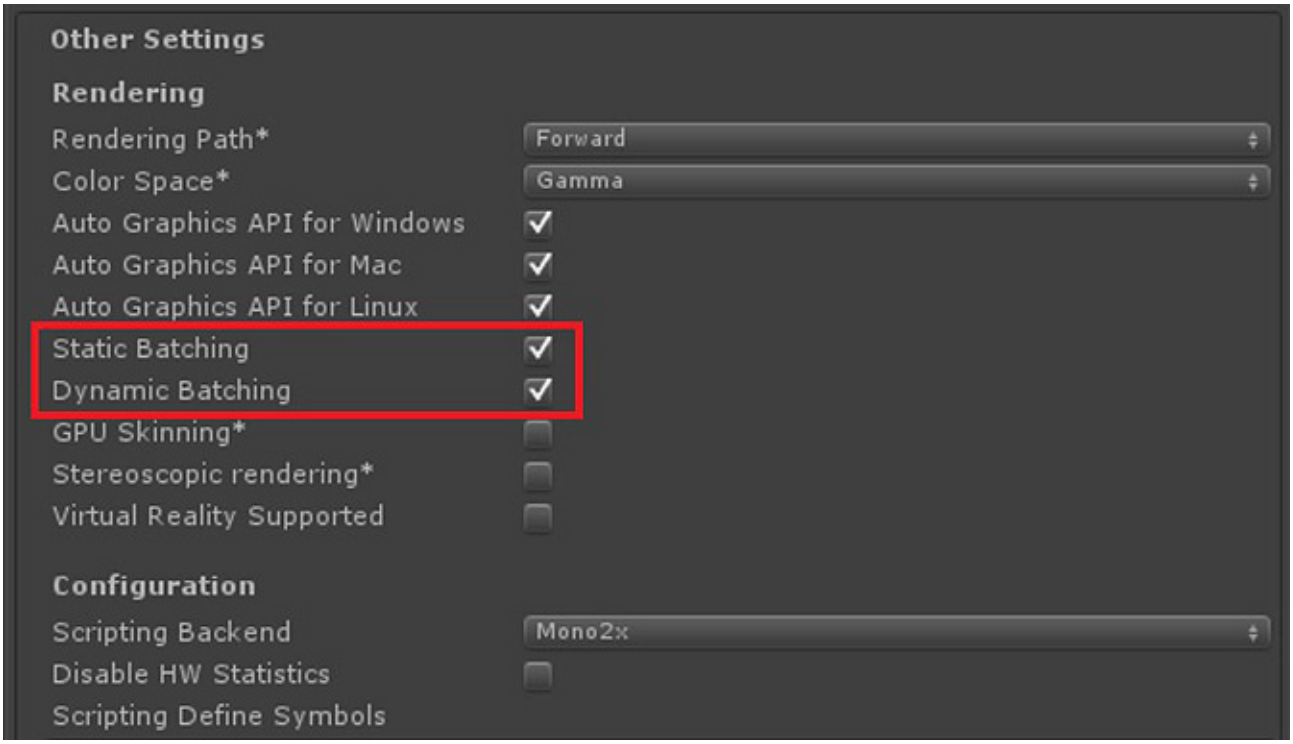

図 47. [Other Settings (その他の設定)] の [Rendering (レンダリング)] の [Static Batching (スタティック・バッチ)] チェックボックスと [Dynamic Batching (ダイナミック・バッチ)] チェックボックス

静止しているゲーム・オブジェクトはすべて、[Static Batching (スタティック・バッチ)] チェックボックスをオンにします。そうすることで、マテリアルを共有する複数の類似オブジェクトでスタティック・バッチが使用されます。

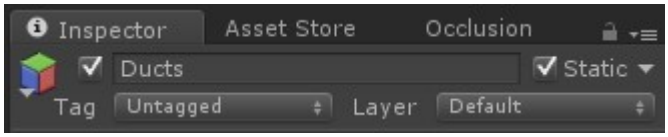

図 48. オブジェクト・ベースのバッチのチェックマーク

- マテリアルの共有を維持するには、Renderer.material の代わりに Renderer.sharedMaterial を使用します。
- 可能であれば、同じマテリアルで使用する複数のテクスチャー (Diffuse、Bumped Specular、その他) を合成してテクスチャー・アトラス化し、バッチ・オブジェクトの数を増やします。

ほとんどの場合、スタティック・バッチはアプリケーションに多くの利点をもたらしますが、使用しないほうが良い状況もあります。メモリー使用量を抑える必要がある場合、スタティック・バッチは使用すべきではありません。ジオメトリー・データを共有するオブジェクトであっても、各インスタンスの頂点/インデックス・データのコピーを描画呼び出しの頂点/インデックス・バッファーにパックする必要があります。一例として、Unity* では、スタティック・バッチが深い森の風景で不適切なことを示しています。この状況では、同じマテリアルを含む各ツリーが呼び出しを行う前にこれらのバッファーにパックされ、パフォーマンス問題を引き起こすことがあります。

ダイナミック・バッチ

ダイナミック・バッチは、スタティック・バッチと同じ概念ですが、代わりにダイナミック・オブジェクト (移動するオブジェクト) の描画呼び出しをバッチします。

- ダイナミック・バッチは、頂点の属性が合計 900 未満のオブジェクトにのみ適用されます (値は変更されることがあります)。

- リアルタイム・シャドウを受け取っているオブジェクトはバッチされません。

HDR - ハイダイナミック・レンジ

遅延レンダリングを使用するブルームやフレアのような HDR 効果を含むシーンがある場合、[Camera (カメラ)] 設定の [HDR] オプションをオンにすると描画呼び出しが大幅に減ります。各カメラに個別の [HDR] チェックボックスがあることに注意してください。DirectX® 10 以降で遅延レンダリングを使用する場合は、HDR を使用すると良いでしょう。HDR は、フラグメント・シェーダーと最も密接に関連しています。HDR 効果を使用する場合は、次のガイドラインに従います。

- 遅延レンダリングを使用します。
- HDR 効果を使用するすべてのカメラの [HDR] チェックボックスをオンにします (図 49)。

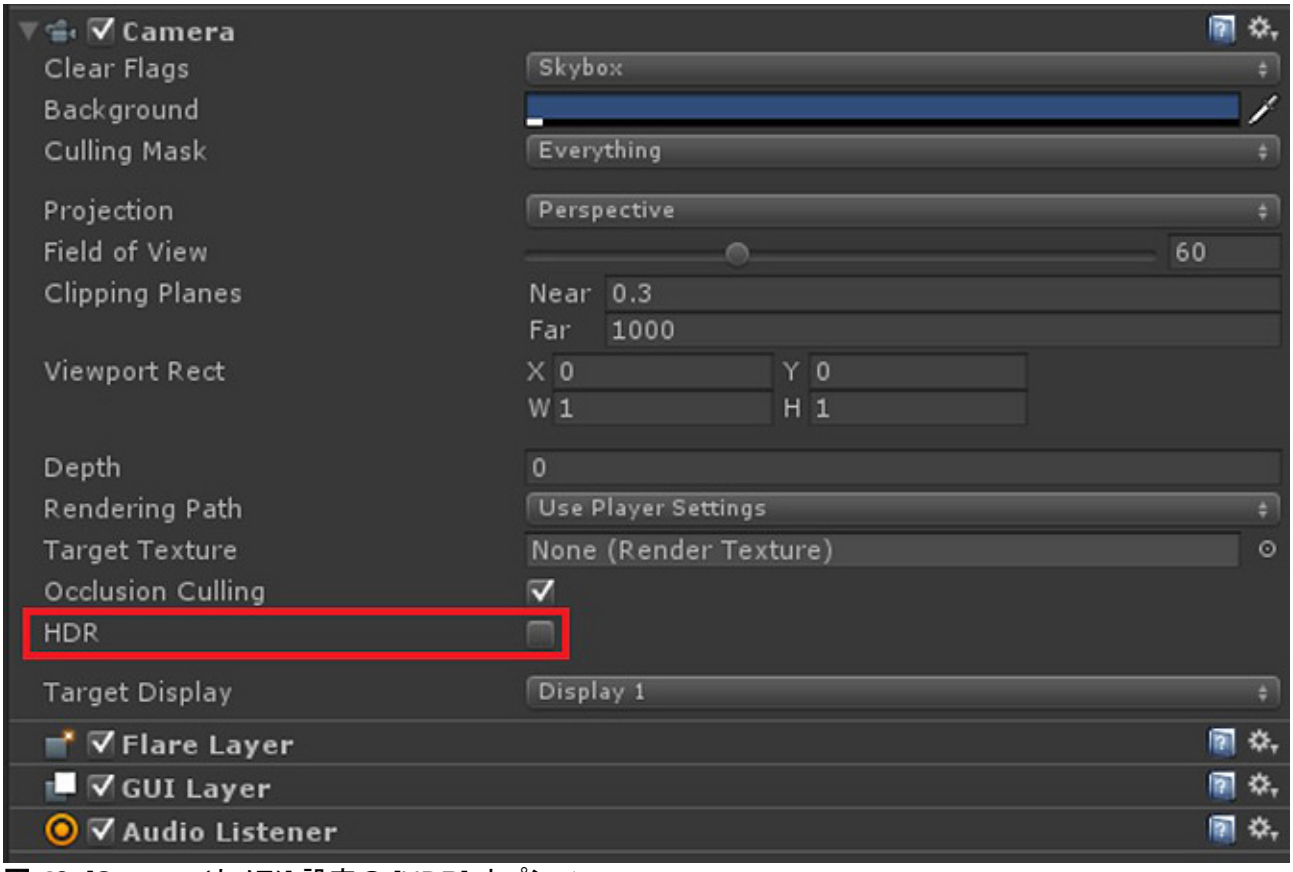

図 49. [Camera (カメラ)] 設定の [HDR] オプション

最適なレンダリング・パスの選択

アプリケーションに最適なレンダリング・パスは、その目的により異なります。以下に、Unity* が提供しているレンダリング・パスの概要を、長所と短所を含めて説明します。以下の情報から、プロジェクトに最適なパスは分かりますが、残りの最適化と同様に、各オプションをテストしてみてください。ゲームに最適なレンダリング・パスを確認するには、ボタンでレンダリング・パスの切り替えを行えるコードを記述して、Unity* Profiler およびインテル® GPA で各パスの効果をリアルタイムに観察すると良いでしょう。

頂点ライト・レンダリング

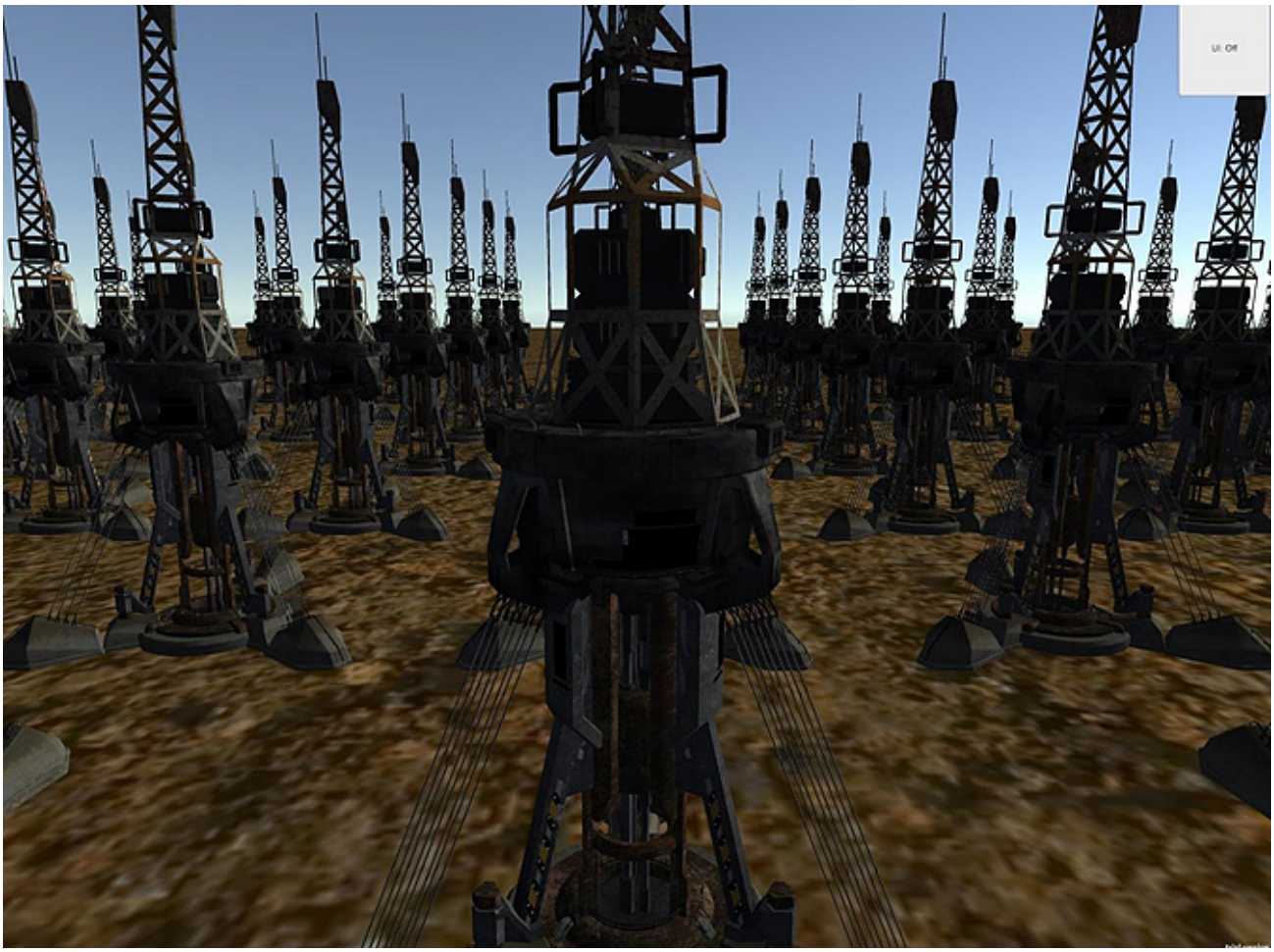

図 50. 頂点ライトパスでレンダリングされたシーンのレンダーターゲット

- 長所
  1. ピクセル単位ではなく頂点単位でライティング計算を行います。
     - 4K モニターの画面全体をカバーする 1024 頂点のモデルの場合、ライティング計算の回数は次のようになります。
       - 頂点単位のライティング計算を行う場合は 1024 回
       - ピクセル単位のライティング計算を行う場合は 8294400 回
  2. モバイル・パフォーマンスを大幅に向上します。
  3. 簡単に解析できます。ベースパスですべて行われます。
- 短所
  1. リアルタイム・シャドウおよびその他のピクセル単位の効果はサポートしていません。
  2. ライティングの画質は低くなります (図 50)。

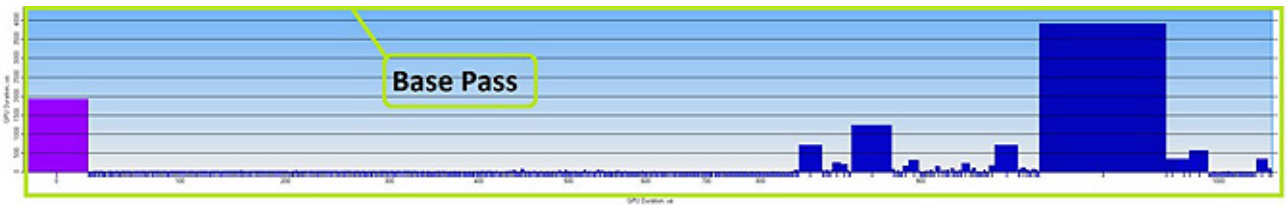

図 51. 頂点ライト・レンダリング・パスの内訳 (すべてベースパス)

フォワード・レンダリング

図 52. フォワードパスでレンダリングされたシーンのレンダーターゲット

- 長所
  - ピクセル単位、頂点単位、球面調和手法を組み合わせてライティングを行います。
  - リアルタイム・シャドウおよびその他のピクセル単位の効果をサポートしています。
  - 遅延パスの G バッファーの作成に必要なメモリーコストが発生しません。
- 短所
  - 注意しないと、同じピクセルに対する描画呼び出しが行われます。
- パスの内訳
  - ベースパス
    - 最初に、最も明るいディレクショナル・ライトが予約されます。
    - 次に、重要としてマークされたほかの (最大 3 つの) ピクセル単位のライトが描画されます。重要としてマークされたライトがない場合、シーンから次に明るい 3 つのライトが選択されます。重要としてマークされたライトの数が [Project (プロジェクト)] > [Quality (画質)] の [Per-Pixel Light Count (ピクセル単位のライト数)] で設定した値よりも多い場合、追加の処理が行われます。
    - 次に、最大 4 つのライトが頂点単位でレンダリングされます。
    - 最後に、残りのライトが球面調和計算を使用して描画されます (これらの値は常に計算されるため、ほとんど GPU を利用しません)。
  - ピクセル単位のライティング・パス
    - ベースパスの後に残っているピクセル単位のライトに対して行われる追加のパス。
  - 半透明オブジェクト・パス

- 半透明オブジェクトに対して行われる追加のパス。

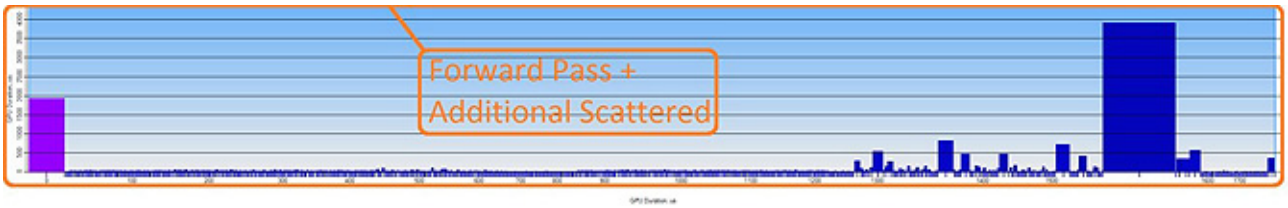

図 53. フォワードパスでレンダリングされたシーンのレンダーターゲット

遅延シェーディング

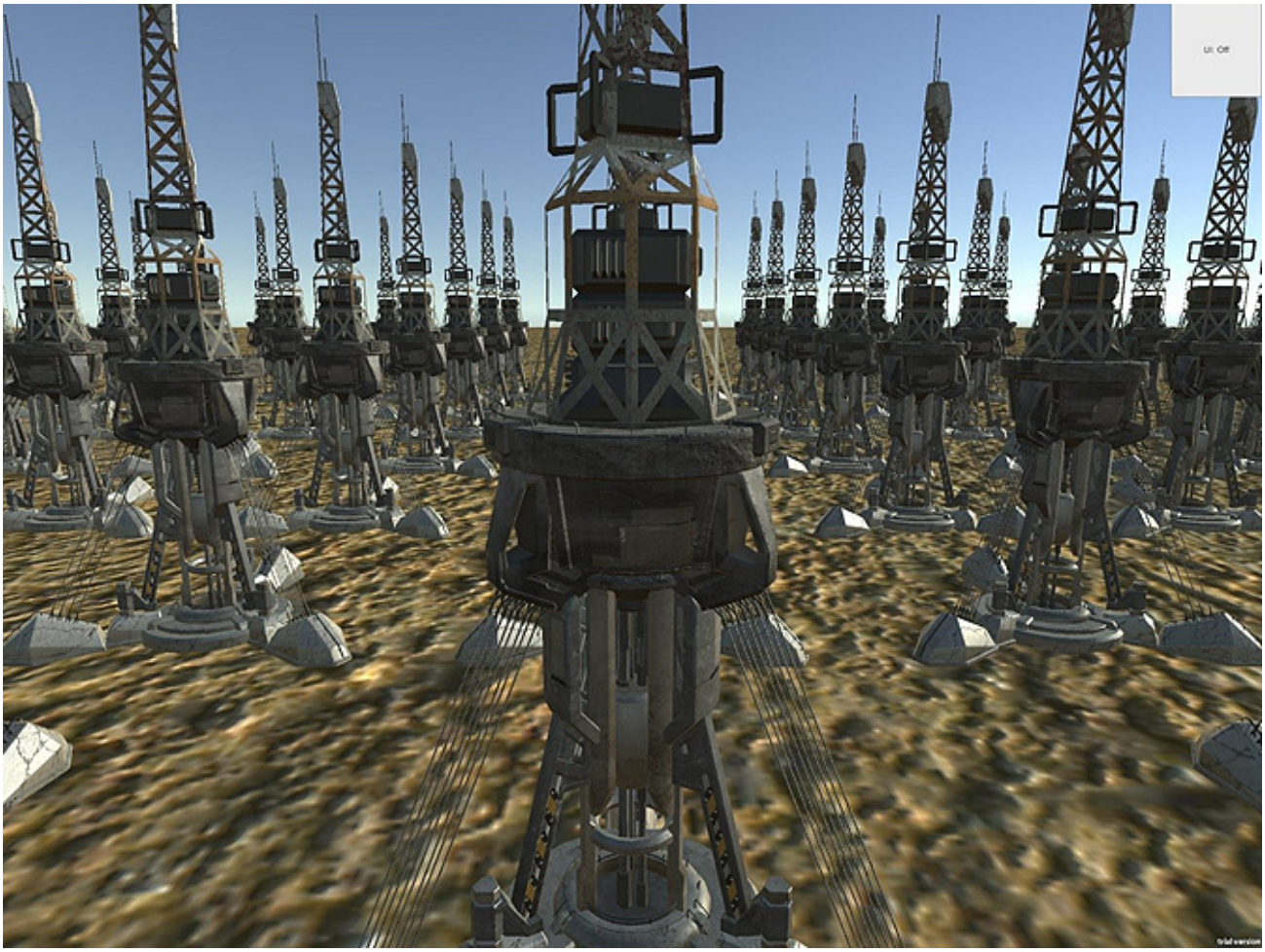

図 54. 遅延パスでレンダリングされたシーンのレンダーターゲット

- 長所
  1. ライティングのパフォーマンスはシーンの複雑さとは関係ありません。
  2. メモリー使用量の多いライティング計算 (FLOPS) はメモリー依存になる可能性が高くなります。
  3. リアルタイム・シャドウおよびその他のピクセル単位の効果をサポートしています。
- 短所
  1. 半透明レンダリングは直接サポートしていません。これらのオブジェクトは追加のフォワードパスで描画されます。
  2. G バッファーを作成するためメモリー使用量が多くなります。
  3. アンチエイリアスはサポートしていません。
  4. メッシュレンダラーの [Receive Shadows (影を受け取る)] フラグはサポートしていません。

- パスの内訳
  1. G バッファーパス
     - すべての不透明オブジェクトをレンダリングして G バッファーを作成します。レイアウトは次のとおりです。
       - レンダーターゲット 0: ARGB32 – RGB チャネルに拡散色、アルファチャネルにオクルージョン・データ
       - レンダーターゲット 1: ARGB32 – RGB チャネルに反射色、アルファチャネルにラフネス
       - レンダーターゲット 2: ARGB2101010 – RGB にワールドスペースのノーマル、アルファチャネルは未使用
       - レンダーターゲット 3: ARGB32 (非 HDR) または ARGBHalf (HDR) – 発光、ライティング、ライトマップ、反射プローブバッファー
       - デプスバッファーとステンシルバッファー
  2. ライティング・パス
     - G バッファーパスで生成されたテクスチャーを使用してピクセル単位のライティング計算を行います。Unity* は、ジオメトリー・バインドのボリュームをデプステストに渡して、オクルード/部分的にオクルードされたライトを簡単に見つけられるようにします。
     - 拡散ライティング値を含む RGB チャネルとモノクロの反射色を含むアルファチャネルでテクスチャーを作成します。
  3. ライト・アプリケーション・パス
     - 最後のパスは、ライティング・パスにより生成されたテクスチャーを使用して各オブジェクトにライティングを適用し、すべてのオブジェクトを再度描画します。
  4. 半透明オブジェクト・パス
     - 追加のフォワードパスが必要です。

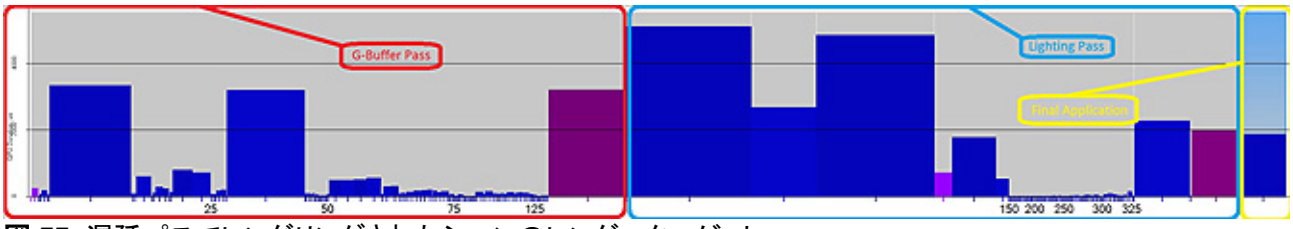

図 55. 遅延パスでレンダリングされたシーンのレンダーターゲット

*利用可能なレンダリング・パスに関する詳細は、http://docs.unity3d.com/jp/current/Manual/Rendering-Tech.html を参照してください。

追加のフォワード描画呼び出しに注意する

前述したように、フォワード・レンダリング・パスは、エディターの画質設定の **[Per-Pixel Light Count (ピクセル単位のライト数)]** で設定した値まで、ジオメトリーに影響する各ライトについて追加の描画呼び出しを行います。これらの呼び出しでは、最初に重要としてマークされたライトが使用されますが、重要としてマークされたライトがない場合、Unity* は次に明るいライトを選択します。場合によっては、この処理で大量のオーバーヘッドが発生することがあります (特にライトのベイクがオプションの場合)。次のインテル® GPA のスクリーンショットは、ベース描画と 3 つの色を示しています。次の状況を回避する必要がある場合、ライトをベイクするか、画質設定のピクセル単位のライト数を小さくすると良いでしょう。

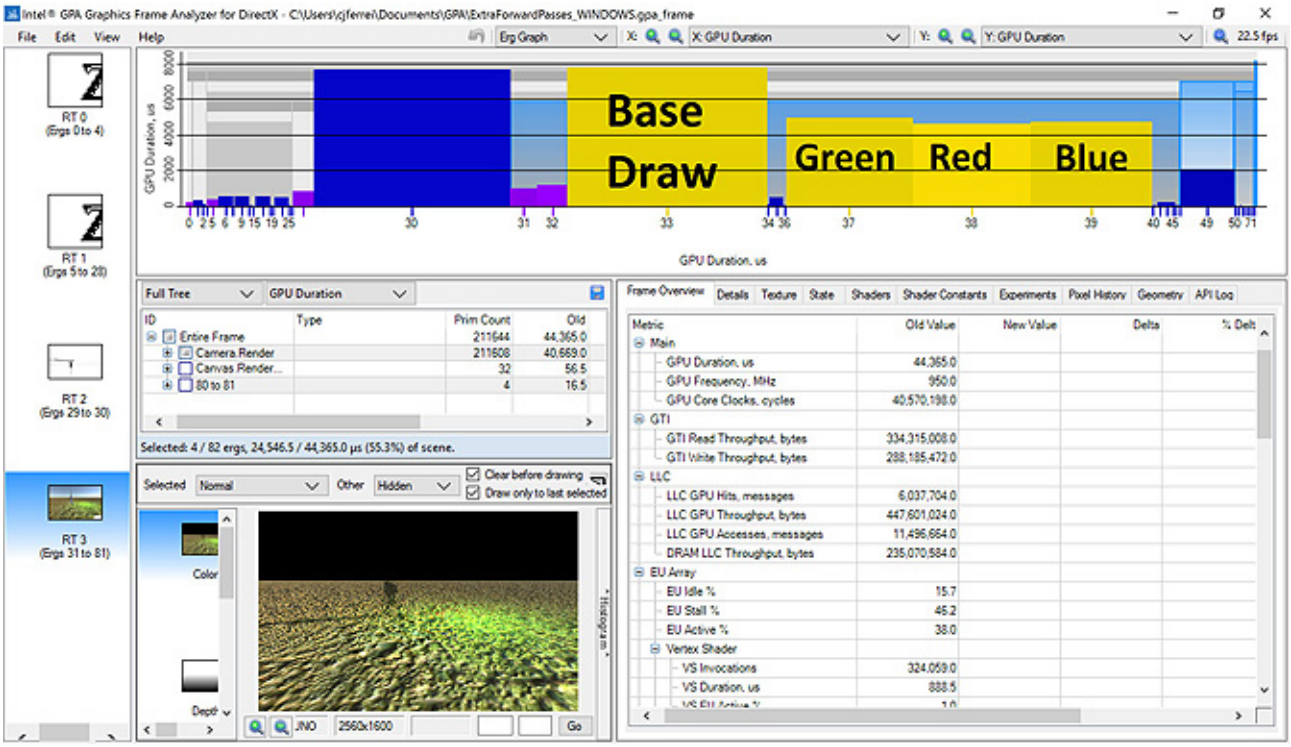

図 56. インテル® GPA のスクリーンショット。この床を描画するために必要な 4 つの描画呼び出しが GPU 時間の 55.3% を占めている。

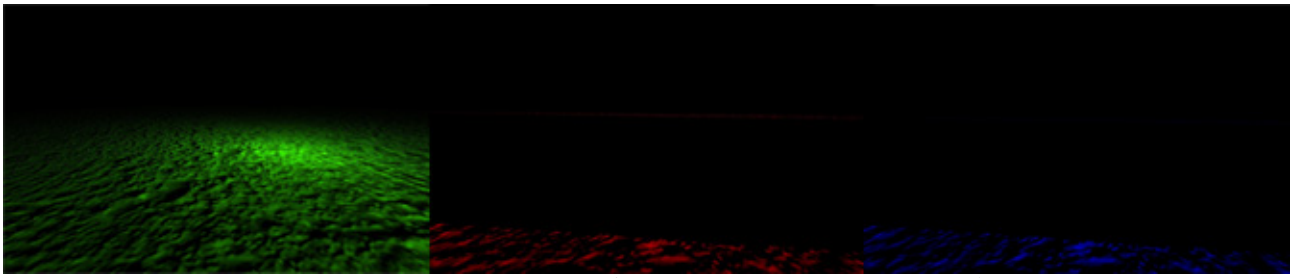

図 57. インテル® GPA で緑、赤、青ライトに必要な追加のフォワードパスのカラーバッファーを表示

Android™: バイナリーの分割

最新の Unity* リリースでは、fat バイナリーを作成するか、ARM* と x86 部分にバイナリーを分割して作成できるようになりました。同じプロセスを使用して x86 または ARM* のいずれかを選択し、さまざまな配布をテストできます。圧縮、コード、その他の詳細を評価すると、ビルドのトラブルシューティングやベンチマークに役立ちます。

fat APK をビルドしてもバイナリーのサイズはそれほど増加しません。x86 または ARM7* を選択してよりサイズの小さなバイナリーをビルドすることもできますが、その場合、2 つの個別のビルドを管理する必要があります。

プレーヤー設定 ([File (ファイル)] > [Build Settings (ビルド設定)] > [Player Settings (プレーヤー設定)]) で、次の操作を行います。

1. [Other Settings (その他の設定)] を開きます。
2. [Configuration (構成)] の [Device Filter (デバイスフィルター)] で、FAT (ARMv7+X86) を選択します。図 58 を参照してください。

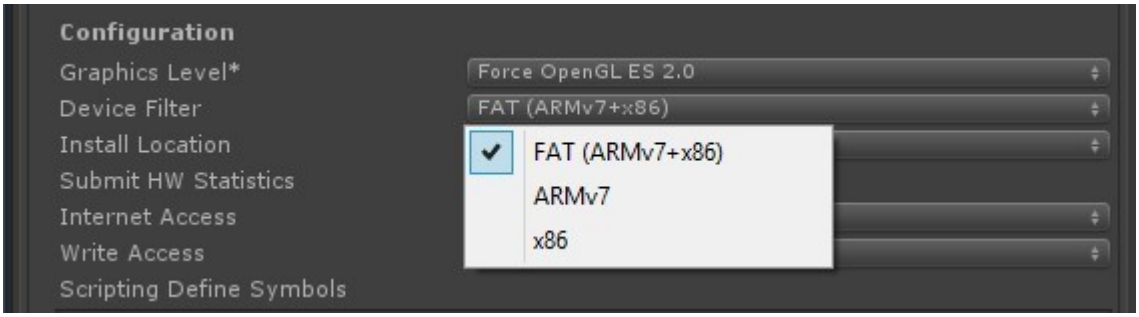

図 58. [Other Settings (その他の設定)] の [Configuration (構成)]。[Device Filter (デバイスフィルター)] に 3 つのオプションが表示されています。

3. [Build Settings (ビルド設定)] ウィンドウで [Build (ビルド)] ボタンを選択して、選択したバイナリーの作成を開始します。

Unity* Android™ ゲームの配布で x86 がサポートされました。

サポートの詳細は、Unity* x86 開発者向けページ (www.intel.com/software/unity (英語)) を参照してください。

まとめ

最適化は、グラフィックスを多用するゲームで高レベルのパフォーマンスを達成するための作業です。上記の手法を組み合わせると、パフォーマンスの向上に役立ちます。これらのツールを使用すると、より細部に渡る調整を行うことができます。

関連情報

ファイルサイズの削減: http://docs.unity3d.com/jp/current/Manual/ReducingFilesize.html

影: http://docs.unity3d.com/jp/current/Manual/Shadows.html

著者紹介

Cristiano Ferreira は、インテル コーポレーション デベロッパー・リレーション部門のソフトウェア・エンジニアで、ゲームおよびグラフィックスを専門としています。インテル® プロセッサー・ベースのハードウェアでゲーム開発者が最高のユーザー体験を提供できるように支援しています。

Steve Hughes は、インテル コーポレーション のシニア・ソフトウェア・エンジニアで、デスクトップおよびタブレットからスマートフォンまで、x86 プラットフォーム上でのゲーム開発のサポートを行っています。インテルに入社する前は、ゲーム開発者として 12 年間、さまざまな会社でゲーム開発のあらゆる面に取り組んでいました。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。