『Imperator: Rome』の最適化されたグラデーション境界レンダリング

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「Optimized Gradient Border Rendering in Imperator: Rome」の日本語参考訳です。

はじめに

『Imperator: Rome』は Paradox Development Studio* によって開発された最新の戦略ウォーゲームです。インテルは Paradox Development Studio* と協力して、インテル® Iris® Pro グラフィックス 580 上でこのゲームのパフォーマンスの最適化に取り組みました。通常とはやや異なるパフォーマンス解析アプローチが必要であったため、興味深いプロジェクトでした。このプロジェクトにおける課題は、パフォーマンスが低いことではなく、グラデーション境界のレンダリング・システムが原因でときどき表示の途切れが観察されたため、我々はパフォーマンスとは何かを再定義することになりました。インテルと Paradox の開発者は、数種類のインテル® インテグレーテッド・グラフィックス・ハードウェアにおいて、スムーズなフレームレートでのゲームプレイを実現しました。

この記事では、インテルと Paradox の開発者による取り組みを紹介します。最初に、従来のパフォーマンス解析を行い、そこからどのような結論が得られるか見てみます。次に、表示の途切れの原因をどのように発見したのかを示し、パフォーマンスを再定義します。そして、技術的な背景について説明してから、問題のシステムが何をするもので、どのように動作するのかを、詳しく見ていきます。最後に、適用した最適化とそれによる結果を説明します。

初期のパフォーマンス評価

開発者担当エンジニアがゲーム・プロジェクトで最初に行うことは、パフォーマンスを制限している要因を特定することです。通常、**GPU 依存**または **CPU 依存**という用語が使用されますが、これは、GPU または CPU がパフォーマンスを制限している最大の要因であることを意味します。この 2 つのハードウェアは、多くの場合、パフォーマンスの問題の原因として最初に疑われます。

パフォーマンスを制限する要因を特定する方法は簡単です。一般に、実行時間の少なくとも 95% で GPU がビジー状態であれば GPU 依存で、そうでなければ、おそらく CPU 依存です。しかし、パフォーマンスにとってこれがすべてではない場合はどうでしょうか? 平均 FPS (1 秒あたりのフレーム処理数) を上回るだけでは「良いパフォーマンス」と言えない場合はどうしたら良いのでしょうか?

パフォーマンスを制限する要因の特定を支援するツールは多数ありますが、個人的には GPUView が気に入っています。GPUView は、Windows* アセスメント & デプロイメント キット (Windows* ADK) に含まれる無料のプロファイラーです。CPU と GPU で何が起こっているのかを素早く把握できるため、ゲームのパフォーマンス解析には最適です。GPU スレッドと CPU スレッドでどれだけの処理が必要かも分かります。

『Imperator: Rome』(Pompey パッチ) を Pause (一時停止) モードにしてすべての CPU シミュレーションを停止して、専用のアセットを持つゲーム内の特別なステージの 1 つであるエジプトにカメラをパンした状態で、GPUView トレースを収集した結果が図 1 です。

図 1. Pause モードのゲームの GPUView トレース。3 つのレンダリングされたフレーム、GPU 上の処理、およびアクティブなスレッドが示されている。

Flip Queue から分かるように、画面に表示された 3 つのフレームのトレースを示しています。Flip Queue は、垂直同期 (VSync) とアプリケーションが画面に表示するデータの関係を示します。見やすいように、VSync イベントの表示を青色の垂直線に切り替えることもできます。Flip Queue のバーは、単色と網掛けの 2 つのセクションに分割されています。単色セクションは、表示用のコンテンツの生成 (通常、これは Desktop Windows Manager (DWM) によって処理される) に費やされた時間を示し、網掛けセクションは、このデータが VSync イベントを待機しているアイドル時間を示します。

ここで示す期間では、Hardware Queue セクションの右下に表示されているように、GPU は 99.66% ビジー状態にあります。前述の定義によると、このゲームは GPU 依存となります。つまり、ゲームのワークロード実行は GPU の効率に完全に依存しており、CPU ワークロードを最適化してもパフォーマンスは向上しませんが、GPU コンテンツを最適化するとパフォーマンスは向上します。

次に、実行中の CPU スレッドを示す緑色のセクションを見てみましょう。最も多くの処理を行っているスレッドは約 15% ビジー状態で、ワーカースレッドは約 3% ビジー状態です。このことから、ゲームは GPU 依存であり、CPU 処理はパフォーマンスにほとんど影響していないことが明らかです。

今度は、CPU シミュレーションをバックグラウンドで実行した状態でトレースしてみましょう (図 2)。ローマとしてプレイしていると、ゲームの序盤でサムナイトと戦闘を開始し、「開戦を宣言」するとすぐに臣下も参戦しました。主力のローマ軍 18,000 人とサムナイト軍 8,000 人の間で陸戦が行われており、同時に要塞の包囲戦も行われています。この時点でゲームは Play (プレイ) モードです。

GPU は 99% ビジー状態であり、シミュレーション実行中も、ゲームは GPU 依存であることを示しています。このタイムフレームで最も多くの処理を行っているスレッドは約 32% ビジー状態です (Pause モードでは

15% ビジー状態でした)。シミュレーションは、CPU に大きな影響を与えたようです。しかし、ゲームはまだ GPU 依存です。

さまざまな情報が得られたところで、初期のパフォーマンスを評価してみましょう。インテル® Iris® Pro グラフィックス 580 上で『Imperator: Rome』(Pompey パッチ) は、Pause モードか Play モードかに関係なく、明らかに GPU 依存であることが分かりました。GPU が主要なボトルネックであり、理論的にはパフォーマンス最適化の最良のターゲットです。

しかし、実際のパフォーマンスはどうでしょうか? テストでは約 30 FPS の平均スコアを達成しました。それでは何が問題でしょうか? 30FPS を達成したのであれば、ゲームのパフォーマンスは良く、問題なくプレイできるのではないでしょうか?

図 2. Play モードのゲームの GPUView トレース。ゲームの序盤に、
ローマ軍とサムナイト軍の間で陸戦と要塞の包囲戦中に収集。

パフォーマンスの不具合の発見

レポートはパフォーマンスが良好であることを示していましたが、問題がありました。Pause モードでは問題なくプレイでき、Play ボタンを押した後も問題ありませんでしたが、カメラをパンし始めると、数フレームごとにパフォーマンス上の不具合が生じ、ゲームがフレームをスキップしているように見えました。そして、再度ゲームを一時停止すると、問題は発生しなくなりました。問題を視覚化するため、無料のパフォーマンス測定ツールである PresentMon を使用して収集したデータが図 3 のグラフです。ここで、Y 軸は Present の処理時間 (ミリ秒単位) を示し、X 軸はフレーム数を示します。

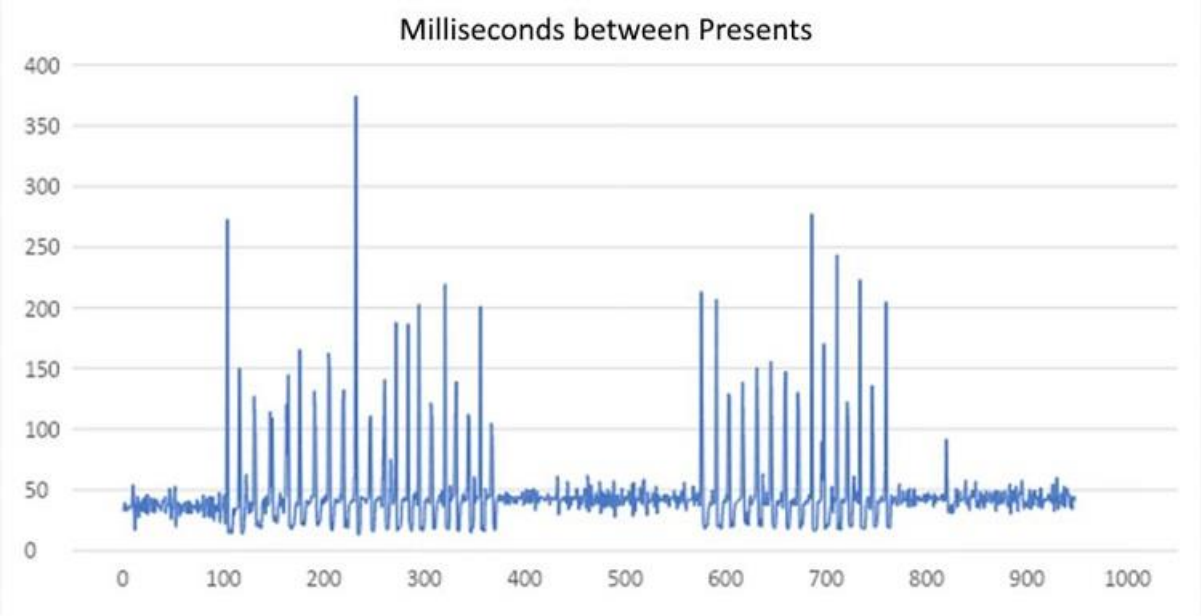

図 3. Present の処理時間 (ミリ秒単位) を示すグラフ。パフォーマンスが大きく変動しているセクションがあり、Play モードに入ると始まり、Pause モードに入ると終わる。

このグラフから、パフォーマンスは白か黒かのように単純には評価できないのではないか、と考えるようになりました。ゲームは平均 30FPS を達成していましたが、パフォーマンスに問題があり、完全に楽しむことができませんでした。さらに、初期のパフォーマンス評価では、完全に GPU 依存でしたが、CPU シミュレーションにより、少なくとも周期的にそうとは言えなくなりました。

CPU シミュレーション中に問題が発生したため、CPU ワークロードとの関連を最初に疑いました。計算負荷の非常に高い処理があってゲームが周期的に CPU 依存になっている、もしくは、GPU と同期が必要な CPU 処理があり実行をブロックしていると考えました。調査のため、CPU プロファイラー・ツールであるインテル® VTune™ プロファイラーを使用してトレースデータを収集しました。

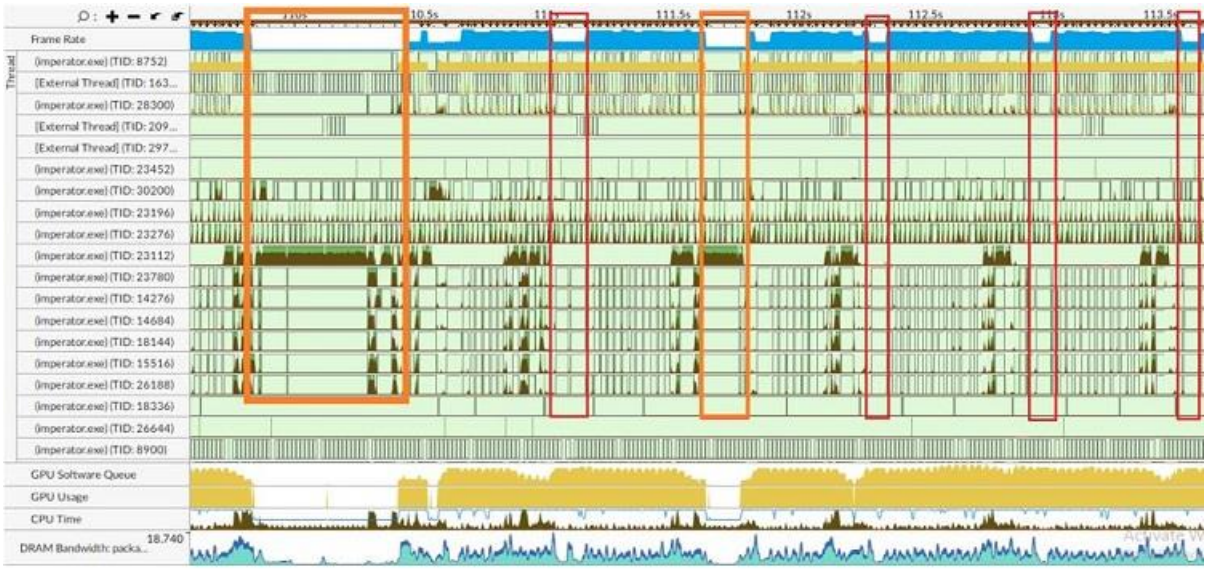

図 4. インテル® VTune™ プロファイラーでゲームの Play モードに入るところをトレース。オレンジと赤のボックスは、フレームレートが低下しているセクションを示している。

私の推測はすべてはずれました。図 4 の最初の行の青色のグラフはフレームレートを示しています。オレンジと赤のボックスは、フレームレートが低下している期間を示しています。オレンジのボックスでは、パフォーマンスの低下は CPU アクティビティーと厳密に相関していることが分かりますが、赤のボックスではそうではありません。また、オレンジのボックスは Play モードに入った直後の 2 回のみ見られます。残りのトレースには、赤のボックスと同じ要素が含まれています。

この結果を Paradox の開発者に見せたところ、オレンジのボックスではシミュレーションの初期化が行われていることが分かりました。最もビジーなスレッドは、ゲームプレイのコマンドを実行する Session Thread (セッションスレッド) です。オレンジのボックスは、新しいゲームの最初の Pause 解除を示し、最初の更新中に Session Thread が初期セットアップを行っていたためフレームレートが低下していました。ゲームの状態は更新時にロックされ、ロック中 GUI は情報を照会できないため、Session Thread はグラフィック・ユーザー・インターフェイス (GUI) を処理する Render (レンダー) と Main Thread (メインスレッド) を停止します。

興味深いことに、この短期的な CPU 依存は、永続的なパフォーマンスの問題の原因ではありませんでした。もし、これが原因であれば、赤のボックスで GPU がストール中に単一のスレッドが連続して大きなチャンクを処理しているはずです。問題の原因として残る可能性は 1 つ、GPU です。内部ツールを使用して、ゲームプレイの DirectX* API 呼び出しデータを収集しました。この手法は GPU 依存のワークロードで使用され、API 呼び出しとすべての関連データを収集します。ワークロードは、CPU に速度制限がなく、パフォーマンスに全く影響しないかのように測定されます。ストリームを再生してデータを収集し、結果を解析してまとめたのが図 5 です。

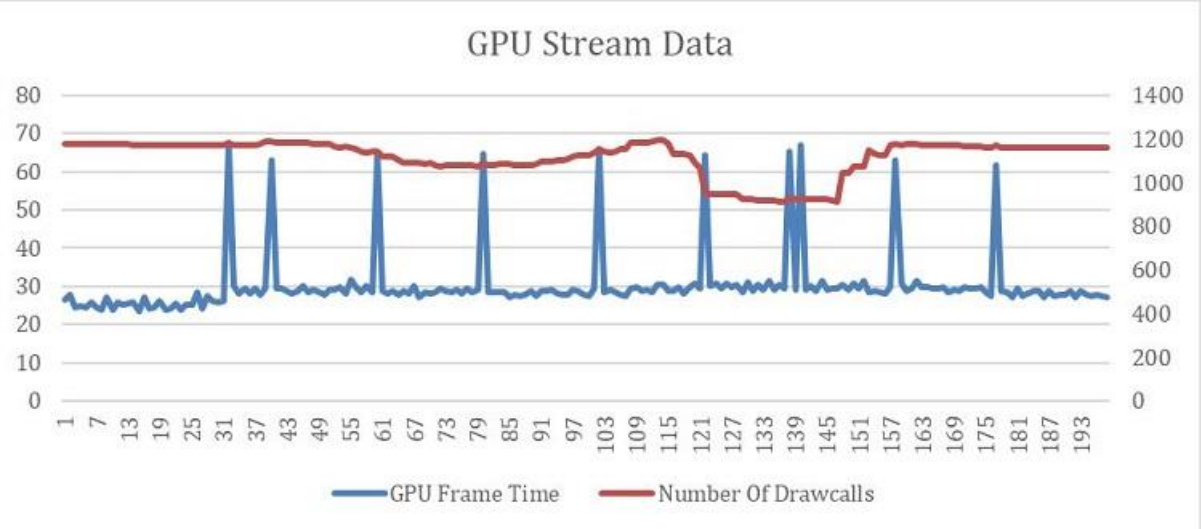

図 5. GPU ストリームのフレーム時間と描画呼び出しの数。

問題の原因が判明しました。数フレームごとに、GPU は約 2 倍の処理を行っていました。これは、例えば、カリングされていないオブジェクトからの追加の描画呼び出しを処理しなければならない場合に発生することがあります。確認したところ、描画呼び出しの数はパフォーマンスと関連していませんでした。

数フレームごとに GPU で追加の処理を行っており、追加の描画呼び出しはほとんどありませんでした。フレームの中にいくつかの追加の描画呼び出しがあったか、一部の描画呼び出しに通常よりも長い時間がかかっていただけかもしれません。インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA) を使用して、ストリームからそれらのフレームの 1 つをキャプチャーすることができました (図 6)。

ようやく、問題の原因が分かりました。フレームに 7 つの追加の描画呼び出しがあり、その実行に約 40ms かかっていました。これが、フレームが途切れる原因だったのです。このフレームのレンダリングには、通常のフ

レームの 2 倍の時間がかかっていました。このことを Paradox に知らせたところ、Border Rendering System (境界レンダリング・システム) が GPU で非常に長い時間を費やしていることが分かりました。Paradox 開発者の Daniel Eriksson 氏は、このシステムについて詳しく説明してくれました。

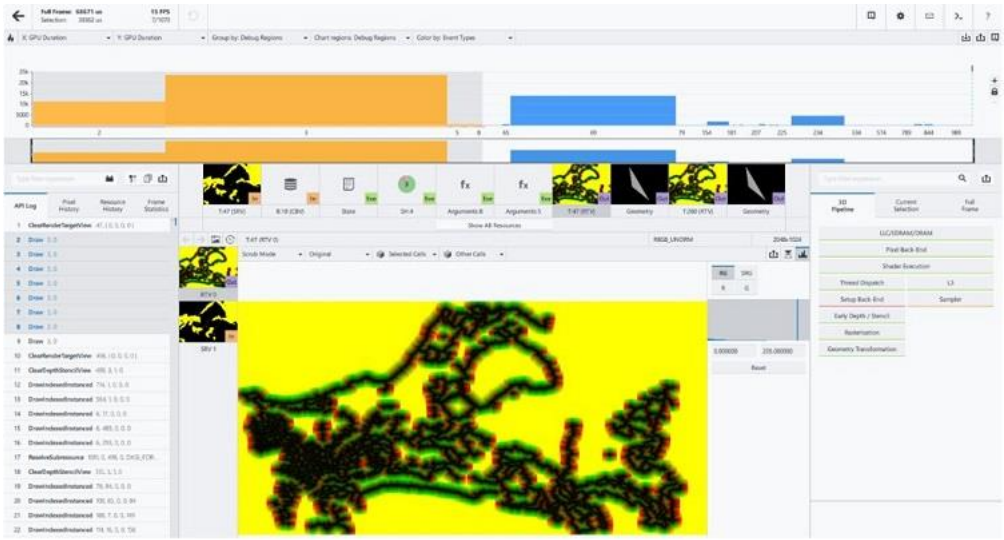

図 6. インテル® GPA ツールでキャプチャーした処理時間の長いフレーム。オレンジのボックスは、高速フレームには存在しない追加の描画呼び出しを示しており、これらの実行には統合 GPU 上で約 40ms かかる。

技術背景

先に進む前に、後述のセクションで使用する技術用語とアルゴリズムを紹介したいと思います。

ディスタンス・フィールド

Distance Field (ディスタンス・フィールド) は、複雑な形状を表現するツールとしてテクスチャーを使用する興味深い手法です。このようなテクスチャーは、色の代わりに、各ピクセルが最も近いジオメトリーまでの距離を保持するグリッドとして考えることができます。国境をジオメトリーとして想像してみてください。各ピクセルのジオメトリーまでの距離が分かれば、境界のグラデーションをきれいに表現することができます。これはまさに、Paradox が『Imperator: Rome』で行っていたことです。

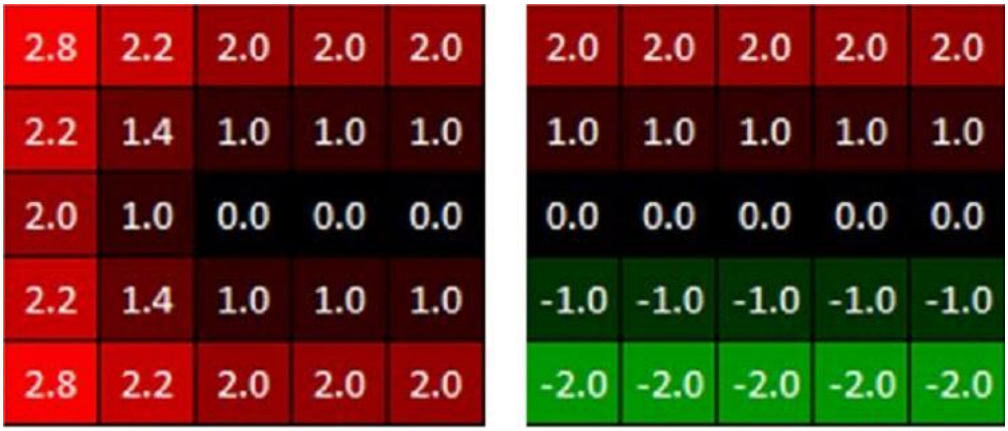

図 7. ディスタンス・フィールド・テクスチャーの例 (「マップ」と呼ばれることもある)。左は符号なしディスタンス・フィールドで、右は符号付きディスタンス・フィールド。ここでは説明目的で値 0.0 (形状を表す) を使用しているが、実際のテクスチャーではここまで明示する必要はない。

図 7 は、2 つの単純な 5x5 ディスタンス・フィールド・テクスチャーです。左は符号なしディスタンス・フィールドで、右は符号付きディスタンス・フィールドです (符号は、ピクセルがジオメトリーの内側にあるか、外側にあるかを示します)。ここでは、表現する形状 (境界) に「内側」がなく、負の値になることはないため、符号なしディスタンス・フィールドのみを使用します。これはまた、エッジに近い部分の精度が低くなることを意味します。Paradox では、マスクのパラメーターを調整する際に、エッジ周辺のアーティファクトを避けるため、アーティストに注意を促しています。

マップを通常のテクスチャーとして保持する場合と比べて、ディスタンス・フィールドにはいくつかの利点があります。まず、マップは非常に大きいため、3 つの値 (色ごとに 1 つ) ではなく 1 つの値を保持することで、サンプリング・コストとテクスチャー・サイズを大幅に軽減できます。また、ディスタンス・フィールドは、ラインの近くで通常のテクスチャーよりも高品質を維持できます。この 5x5 テクスチャーを使用して 1080p レンダリング・ターゲットにレンダリングすると、細く鮮明なラインを得ることができるのに対し、従来のテクスチャーでは、拡大され、ぼやけてしまいます。さらに、このゲームではモッドを許可しているため、プレーヤーとクリエーターがマップを容易に変更できなければなりません。

ジャンプ・フラッディング・アルゴリズム

2 つ目の重要な用語は、ジャンプ・フラッディング・アルゴリズム (JFA: Jump Flooding Algorithm) です。『Imperator: Rome』では、JFA を使用してディスタンス・フィールドのテクスチャーを構築していますが、JFA は、ボロノイ図の構築など、ほかの用途にも使用できます。このアルゴリズムへの入力は、「データがない」ことを意味する任意の値を含むテクスチャーで、これは JFA にシードを提供します。『Imperator: Rome』では、このシードはゲームの世界の境界を示します。アルゴリズムは、テクスチャー全体を数回トラバースして、各ピクセルのディスタンス・フィールドを計算します。指定されたピクセルから「ステップ」距離離れたところにある 8 つの隣接するピクセル (基本方向 4、順方向 4) をサンプリングして計算します。「ステップ」はアルゴリズムの反復ごとに変わります。8 つのピクセルすべての情報を使用して、最も近い境界までの一時的な最接近距離を決定します。

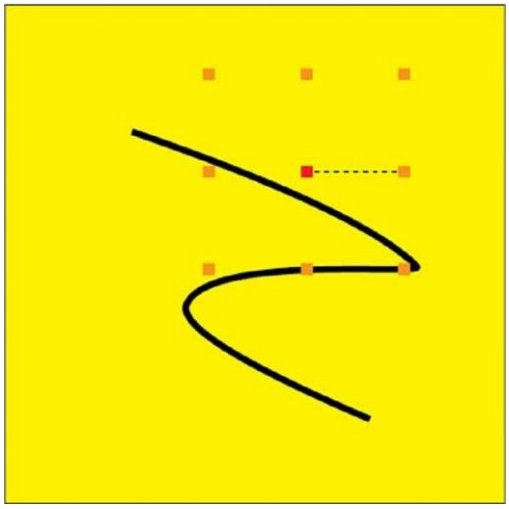

図 8. ジャンプ・フラッディング・アルゴリズム。単一のピクセルの 1 反復の距離を計算。

図 8 は JFA を視覚化したものです。中央の赤い点は、距離を計算するピクセルです。周辺のオレンジの点は、サンプリングする 8 つのピクセルです。黒い線はシード (ジオメトリーまたはエッジ) です。これは、『Imperator: Rome』の実際の領土の境界を表しています。点線は、反復ごとに小さな値になるステップを表します。この特定のピクセルでは、より近いジオメトリーがあるため、結果は正しくありません。しかし、これは、ステップ値が小さくなる後続の反復によって調整されます。

グラデーション境界

『Imperator: Rome』では、国境は地形に重ねて表示されます。黒いスプラインは境界の形状を示し、境界の各辺はマップモードに応じて、国やその他のデータに関連した色のグラデーションになります。この記事では、色のグラデーションについてのみ述べます。図 9 は、ゲームで表示される境界の例です。色のグラデーションは、Color Map (カラーマップ) で色を検索して、ディスタンス・フィールド・テクスチャーからサンプリングする 2 つの独立したステップで達成されます。

図 9. プライマリー・カラーとセカンダリー・カラーが適用されたゲームプレイのスクリーンショット。ネペテの境界は赤をプライマリー・カラーとしており、ここがローマの境界の終わりであることを示している。紫はピルギのプライマリー・カラーで、領土の所有権を示しており、セカンダリー・カラーの赤は、ここが現在ローマの支配下にあることを示している。

カラー・ルックアップ

シェーダーがアクセスできる色は 3 種類あります。

- プライマリー: 最も頻繁に使用される色です。マップモードでプライマリー・カラー値として使用され、例えば、デフォルトのマップモードで自国を示すのに使用されます (図 9 のネペテやカルシオリの境界の赤)。
- セカンダリー: この色は斜線と組み合わされます。マップモードでセカンダリー・カラー値として使用され、例えば、デフォルトのマップモードで侵略国を示す斜線で使用されます (図 9 のピルギの赤い斜線)。
- ハイライト: 州や地域をハイライト表示するのに使用されます (めったに使用されず、図 9 でも表示されていません)。

これらの情報はすべてカラーマップと呼ばれるテクスチャーに保持されます。カラーマップは、州の所有権が変更されたり、ほかの国が領土を占領し始めたり、プレーヤーがマップモードを変更した場合など、ゲームの状態に関連する要素が変更されるたびに再計算されます (図 10)。カラーマップは、図 11 の右の図に示すようなレイアウトで色を保持します。赤い四角はプライマリー・カラー、緑の四角はセカンダリー・カラー、そして青い四角はハイライトカラーが格納されている場所を示します。

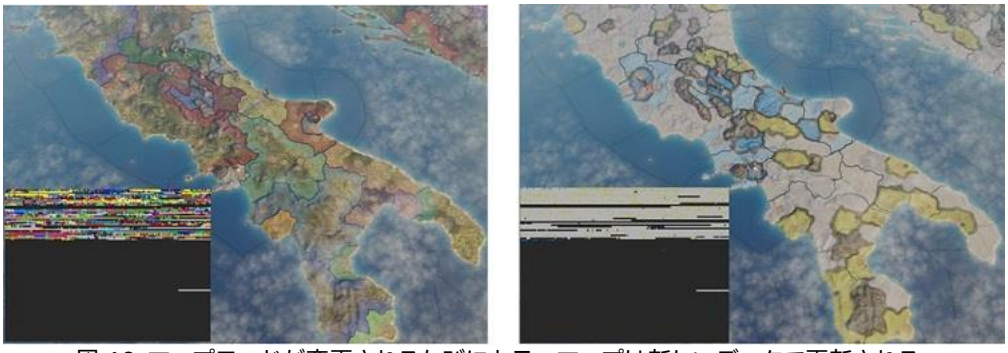

図 10. マップモードが変更されるたびにカラーマップは新しいデータで更新される。

マップ上の任意の点で使用する色を 1 レベルの間接指定で取得できます。最初に、使用するオブジェクト (地形や山など) ごとに、ピクセルシェーダーでワールドスペースからの点座標を 0 ~ 1 の範囲に正規化します。そして、それを基に大きなルックアップ・テクスチャー (図 11 の左の図に示す 8192x4096、**R8G8_UNORM**) をポイント・サンプリングして、はるかに小さなカラーマップ (図 11 の右の図に示す 256x256、**R8G8B8A8_UNORM**) からプライマリー・カラー値の取得に使用する UV 座標を得ます。セカンダリー・カラーとハイライトカラーは、これらの座標をオフセットするだけで取得できます。

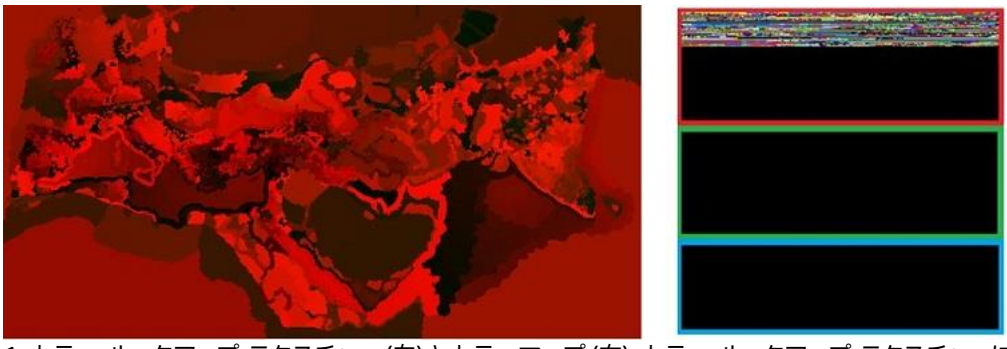

図 11. カラー・ルックアップ・テクスチャー (左) とカラーマップ (右)。カラー・ルックアップ・テクスチャーには、カラーマップでプライマリー (赤い四角)、セカンダリー (緑の四角)、またはハイライト (青い四角) カラーを取得するのに使用する UV 座標が含まれている。

この間接指定には理由があります。ルックアップ・テクスチャーは、ゲームの開始時に一度だけ計算する必要があり、UV 座標には 2 つのチャネルのみが必要です。このテクスチャーはその後変更されません。地域のレイアウトはスクリプトやモッドにより変更される可能性があるため、このテクスチャーは事前定義済みのアセットとして読み込むのではなく、計算する必要があります。色の更新が必要な場合は、3 つの非圧縮 8192x4096 の 4 チャネル・テクスチャーの代わりに、この 1 つのカラーマップを更新するだけで済みます。

ディスタンス・フィールド・テクスチャー

2 番目に必要な情報は、どれぐらい色をブレンドするかです。それには、ディスタンス・フィールド・テクスチャーが必要です。

ディスタンス・フィールドは非常にシンプルです。カラー・ルックアップ・テクスチャーと同様に、正規化されたワールドスペース座標を使用してサンプリングします。ピクセルシェーダーですべての地形の座標が正規化され、滑らかなグラデーションになるように、ディスタンス・フィールド・テクスチャーが数回サンプリングされます。ディスタンス・フィールド・テクスチャーの各ピクセルには、「エッジ」までのおおよその距離が含まれています。この値を、いくつかの定数とアーティストが調整可能なパラメーターと組み合わせて、地形にブレンドする色の量を計算します。

図 12. 最終的なディスタンス・フィールド・テクスチャー。各ピクセルは最も近い境界までの距離を保持している。

境界の描画

ディスタンス・フィールドからサンプリングした距離にアーティストから提供されたパラメーターを適用すると、Mask Value (マスク値) が得られます。このマスク値を使用して、カラーマップのステップで取得した色を地形にブレンドします。図 13 は、すべてのコンポーネントを 1 つの図にまとめたものです。

線形フィルターでサンプリングした場合、低解像度の画像は高解像度の画像とほぼ同じ精度の結果が得られるため、ディスタンス・フィールドは効果的です。グラデーション境界システムは、エッジに近い部分の精度に問題がある符号なしディスタンス・フィールドを使用します。符号付きディスタンス・フィールドにはこの問題がありません。符号付きディスタンス・フィールドについては、Valve 社の Chris Green 氏による「ベクター・テクスチャーと特殊効果のアルファテスト済み倍率の改善」を参照してください。

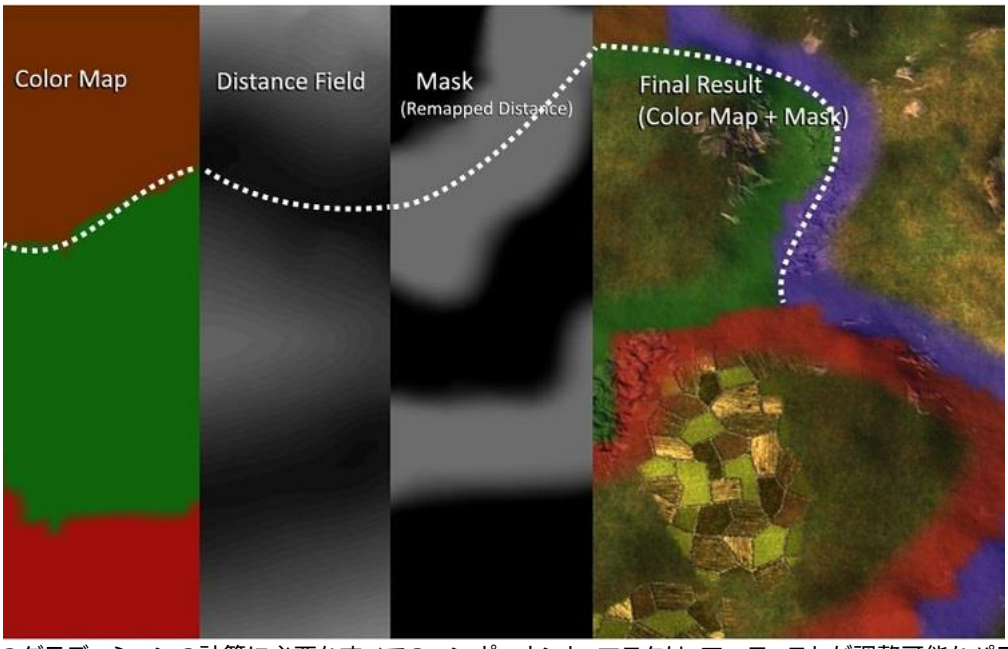

図 13. 地形のグラデーションの計算に必要なすべてのコンポーネント。マスクは、アーティストが調整可能なパラメーターをディスタンス・フィールド値に適用した結果。このケースでは、カラーマップは、図 11 の両方のテクスチャーをサンプリングした結果の組み合わせ。白い点線は境界のおおよその位置のマークアップ。

ディスタンス・フィールドの計算

残る問題は、ディスタンス・フィールドをどのように作成するかです。ディスタンス・フィールド・テクスチャー内の各ピクセルが、最も近いエッジまでの正確な距離を保持するようにしたいところですが、実際には、正確な結果を得ることはできません。ただし、それに近い結果をできるだけ素早く得ることはできます。

ジャンプ・フラッディング・アルゴリズムを使用して GPU でディスタンス・フィールドを計算します。それには、まずエッジを見つける必要があります。エッジが見つかったら、JFA を実行して (O(log(n)) 反復で隣接するピクセルにその情報を知らせます。ここで、n は最も離れているピクセルまでの距離です。

GPU で JFA を実行するには、通常、少なくとも 2 つのピクセルバッファーが必要です。これらのバッファーは、ピンポン形式で読み書きされます。1 つのテクスチャーから読み取り、別のテクスチャーへ書き込んで、2 つのテクスチャーを交換して、次の反復では前の反復の結果を読み取り、次の反復の入力へ書き込みます。このグラデーション境界システムでは、この 2 つのテクスチャーは **R8G8_UNORM** 形式を使用します。JFA 中に 2 つのチャネルを使用して、両軸の最も近いエッジまでの距離を別々に格納することで、ユークリッド距離行列を使用して高精度を達成できます。**R8_UNORM** 形式のテクスチャーを使用して、最終的な正規化された距離を格納します (図 12)。

詳細を説明する前にこれまでの内容をまとめると、ディスタンス・フィールドの計算は 4 つのステップを実行します。

1. 沿岸境界のプリパス。海沿いの地域で沿岸境界の作成を停止するマスクを作成します。
2. 初期化ステップ。(JFA のシードに使用されるダウンサンプリングされたテクスチャー内の) 各ピクセルが、サンプリングされたすべてのピクセルが境界を作成する同じ領域に属する場合は (255, 255) になり、サンプリングされたピクセルが異なる領域に属する場合は (distance-x, distance-y) になるようにします。このステップでは、カラー・ルックアップとカラーマップのサンプリングを何度も実行します。
3. 5 反復での実際の JFA アルゴリズムの実行。
4. JFA 出力からのディスタンス・フィールド・テクスチャーの作成。

次の表は、プロセスの各ステップの概要を示します。各画像は、64x64 テクスチャーを表します。初期化フェーズの点 (4 ピクセル) は、JFA のシードを表します。ゲームでは、この点は国境に置き換えられます。

	初期化。JFA のシードに使用される初期データを提供する必要があります。このデータを作成するには、1 パスで前述のカラー・サンプリング手法によりエッジを検出し、係数 4 でルックアップ・テクスチャーのサイズからダウンサンプリングします (3 ピクセル間のデータを表現できるため、低コストのサンプリングとディスタンス・フィールドは、優れた距離の圧縮方法です)。中央の 4 つのピクセルは黒ではありませんが、最も近い別の色のピクセルまでの距離 (x,y) を含みます。黄色のピクセルの値は (255,255) で、これはこのピクセルに最も近いエッジが可能な限り遠くにあることを意味します。
	JFA 1。最初の反復では、各ピクセルは 16 ピクセルのオフセットで隣接するピクセルをサンプリングします。ピクセルが保持するベクトル長が、隣接するピクセルまでの距離 + 隣接するピクセルが保持するベクトル長よりも小さい場合、ピクセルの値は、隣接するピクセルの値 + 使用されたオフセットで更新されます。
	JFA 2。オフセット 8 で前のステップを繰り返します。

·	**JFA 3。**オフセット 4 で前のステップを繰り返します。
·	**JFA 4。**オフセット 2 で前のステップを繰り返します。
·	**JFA 5。**オフセット 1 で前のステップを繰り返します。このステップと前のステップの違いはわずかです。初期化ステップでは、実際には 2x2 の領域が初期化されます。前のステップのテクスチャーをよく見ると、それが分かります。ギャップはありませんが、ピクセルが 1 つおきに「オーバーシュート」しており、あまり正確ではありません。この時点で、各ピクセルには、最も近いエッジまでの距離を表す 2 つの値 (x 軸と y 軸) が含まれており、これらはベクトルとして扱うことができます。
·	最後に、グラデーション・テクスチャーを作成します。各ピクセルの 2 つのベクトル長を計算するだけで作成できます。

もちろん、これだけでは Paradox のアーティストを満足させることはできません。異なる色の州境を検出する上記の手法は、マップのほとんどで問題なく機能します。しかし、『Imperator: Rome』の美しい海岸線でもグラデーションがブレンドされ、見栄えが良くありません。エッジ検出パスで海岸線のエッジを無視しようとしましたが、隣国境界に大きな川がある場合などに問題が発生しました。

図 14. 緑の領域はグラデーションが必要で、赤の領域は不要。

解決策は、プリパスによって、色をスーパーサンプリングして、無視するマップ領域をエッジ検出に知らせるマスクを作成することです。特定の「ワイルドカード」色を検索して、近くにワイルドカードでない色が 1 か 0 個ある、ワイルドカード・ピクセルのエッジを無視するようにマスクを設定します。

以下のアニメーション GIF は、JFA の各ステップでマップ全体の距離情報がどのように生成されるかを示すものです。GPU で実行され、非常に大きなテクスチャーからサンプリングを何度も行う、この大規模で複雑なシミュレーションがパフォーマンスの問題を引き起こしていました。

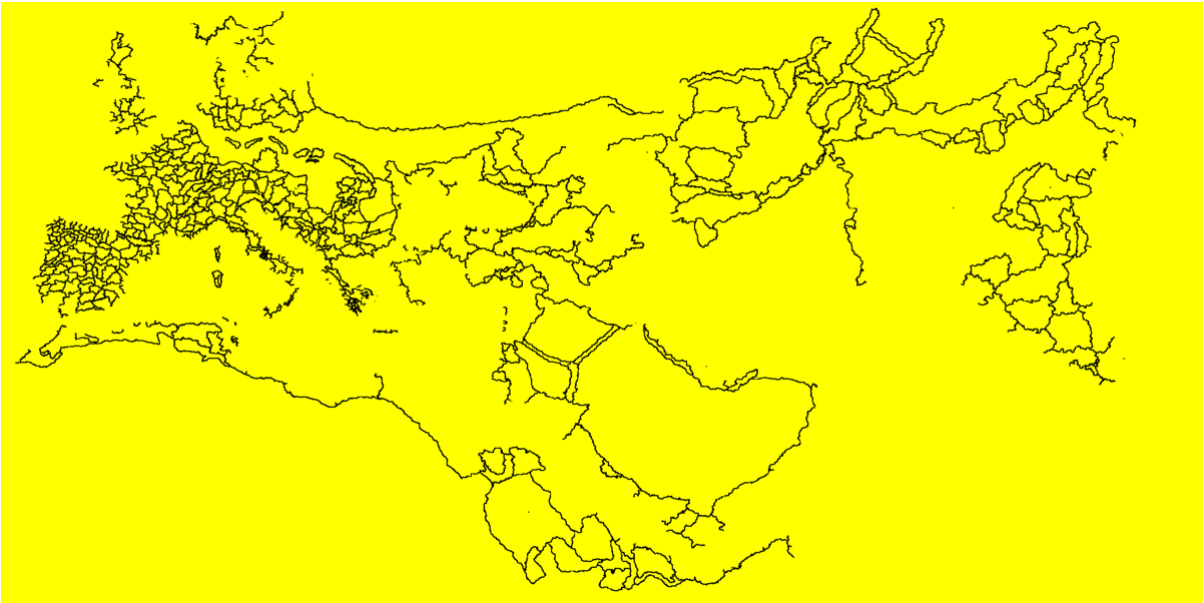

図 15. マップ全体のディスタンス・フィールド計算プロセスのアニメーション。

システムの最適化

ディスクリート GPU では、グラデーション計算時のパフォーマンスの低下は非常にわずかです。グラデーションが再計算されるたびに数ミリ秒のスパイクが見られましたが、これらはまれであり、プレーヤーがボタンを押した直後に発生していました。スパイクがより顕著でゲームプレイの妨げになる統合 GPU では、この機能をプロファイルしたことがありませんでした。これは良い教訓になりました。

この問題をプロファイルして最初に分かったことは、これらの再計算イベントが当初考えていたほどまれではないということでした。グラデーションは 州の 3 つの色の 1 つにしか依存していないにもかかわらず、そのいずれかが更新されるとグラデーションも更新されることが判明しました。また、グラデーションを再計算する前に、実際に変更されているものがあるか確認していませんでした。この 2 つの問題を修正することで、ゲームのスパイクの頻度は日に 1 回から月に 1 回程度に大幅に減りました。

しかし、各スパイクは顕著であり、ゲームプレイの妨げになることに変わりありません。最初の 2 つの修正 (関連する色の変更に対してのみトリガーし、グラデーションを再計算する前に実際に変更されたものを確認する) を実装した後、マップの更新が必要な部分を簡単に確認して測定できるようになりました。ほとんどの場合、一度にいくつかの州を更新するだけで済みます。ログを確認すると、いくつかの州だけを更新するか、ほぼマップ全体を更新していることが分かりました。その間のケースもありますが、それらは非常にまれです。

このことを考慮して、最も効率的な最適化は、一度にいくつかの州を更新する一般的なケース向けに最適化することだと判断しました。グラデーション境界の全体的な更新は、ポストエフェクトのように、クリップ領域全体をカバーする 1 つの三角形の頂点シェーダーと実際の処理を行うピクセルシェーダーで処理されます。実装した最適化では、ジオメトリーをマップ全体の 1 つの「フルスクリーン」三角形から、各クワッドを更新が必要なディスタンス・フィールドの 1 領域をカバーするクワッドのリストに変換します。シンプルで、洗練されており、非常に効率的です。この最適化だけで、GPU の処理量だけでなく、このゲームのパフォーマンス低下の最大の原因の 1 つであるサンプリングの数も減らすことができました。このゲームでは、非常に大きなテクスチャーから多数のサンプリングを行っており、非効率的なキャッシュ利用やシステムメモリーからデータをフェッチする必要性が生じていました。

クワッドの生成は簡単です。各州の境界ボックスはすでに分かっているため、クワッドがグラデーションの最大領域カバーするように幅を調整して、そのルーフまたはフロアをクワッドに変換するだけです。つまり、ディスタンス・フィールドの最大距離でクワッドの両側を押し出します。更新が必要な領域が複数ある場合、それらの領域のクワッドはオーバーラップしている可能性があります。つまり、ピクセルが 2 回計算される可能性があります。このオーバーラップの問題を回避するため、レンダリングの前に、バックグラウンド・スレッドでオーバーラップするクワッドをカットしてマージするアルゴリズムを CPU で実行します。このクワッドマージ・タスクは、利点よりもコストが大きくなる転換点があります (転換点はハードウェアにより異なります)。そのため、一般的なケース (1 ~ 10 の州が更新される) 向けに最適化し、クワッドの数が十分に大きい場合は、マップ全体を更新します。

図 16 は、プレーヤーが 6 州を併合した後に生成されたクワッドです。この領域外に変更がないことは分かっているので、マップ全体ではなく、この小さな領域に対してのみ JFA を実行できます。クワッドのエッジに近いピクセルは、領域外のピクセルをサンプリングしても、同じ結果を生成することができます。

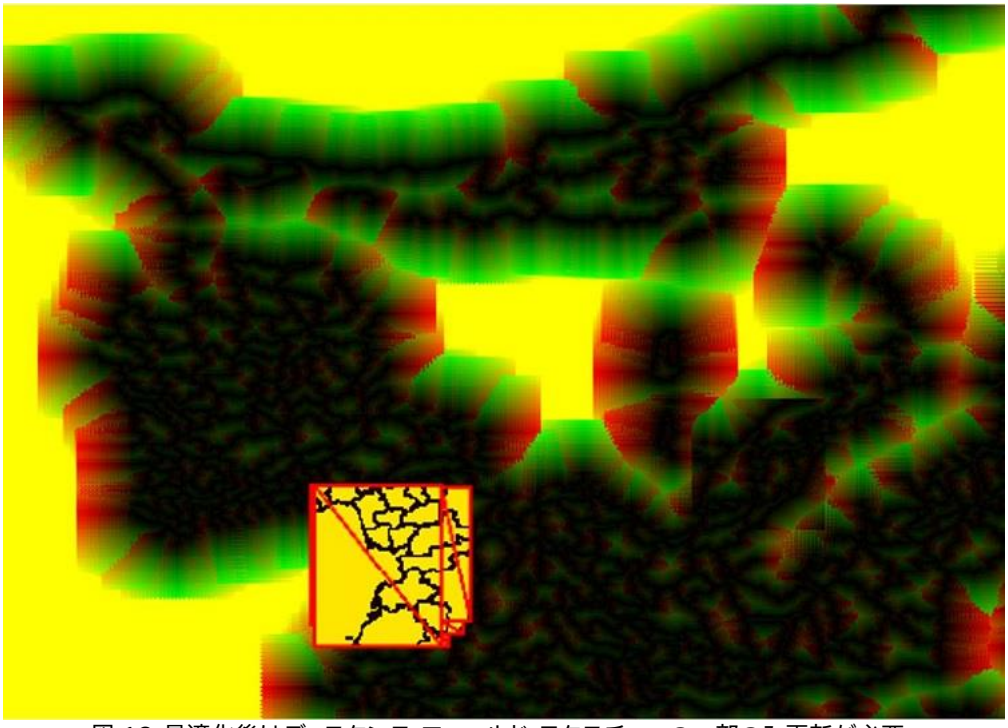

図 16. 最適化後はディスタンス・フィールド・テクスチャーの一部のみ更新が必要。
クワッドは、プレーヤーが 6 州を併合した後に更新が必要な領域を表している。

まとめ

ここで紹介したように、パフォーマンスの問題は、必ずしも CPU 依存か GPU 依存であるとは限りません。むしろプレーヤーの満足度を測る指標であると考えることができます。Paradox 開発者の素晴らしい取り組みにより、フレームの途切れの問題が修正され、より多くのプレーヤーがゲームを楽しめるようになりました。

最後になりましたが、この記事の共著者である Paradox Development Studio* の Daniel Eriksson 氏と、レビューと最終調整に協力してくれた同社の Marcus Beausang 氏、そして、詳細なレビューを提供してくれたインテルの Adam Lake に感謝します。手間はかかりましたが、その甲斐あって、記事は当初よりもずっと良い状態になりました。

ツールと参考資料

Windows* ADK のダウンロードとインストール

PresentMon (パフォーマンス測定ツール) (英語)

インテル® VTune™ プロファイラー (CPU プロファイラー・ツール)

Valve 社の Chris Green 氏によるベクター・テクスチャーと特殊効果のアルファテスト済み倍率の改善 (英語)

製品とパフォーマンス情報

[1] インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。
注意事項の改訂 #20110804