インテル® インテグレーテッド・グラフィックス上でリアルタイム・アップスケーリングを実現するチェッカーボード・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「Checkerboard Rendering for Real-Time Upscaling on Intel Integrated Graphics」の日本語参考訳です。

はじめに

このホワイトペーパーでは、インテル® インテグレーテッド・グラフィックス上でリアルタイム・アップスケーリングを実現するチェッカーボード・レンダリング (CBR) を紹介します。最初に、Sony* PS4 Pro、Frostbite* エンジン (英語) などの一般に利用可能なリファレンスを含む、ダイナミック・リアルタイム・アップスケーリング手法に関するいくつかの研究について説明します。次に、1/4 解像度の 2x MSAA (2 倍のマルチサンプリング・アンチエイリアシング) レンダーターゲットを使用する理由を説明し、テンポラルな再構築手法を紹介します。最後に、単純なフォワード・レンダリングと遅延レンダリングの両方で CBR をサポートするためレンダラーを変更する方法を詳しく述べます。ここで使用したテストシーンでは、結果の視覚効果品質への影響を最小に抑えつつ、CBR では 1.14 倍から 1.2 倍、ピクセル・シェーディングがコストの大半を占めるフレームでは 1.4 倍のスピードアップを達成しました。比較的容易に実装でき、さまざまな解像度にわたってスケーリングし、グラフィックス・プロセシング・ユニット (GPU) メーカー間で互換性のある CBR は、GPU を最大限に活用したいと考える開発者にとって魅力的なソリューションです。この記事では、リファレンス実装のソースコードを公開しています。

サンプルコード

ここで説明する手法は、シェーダーモデル 5 を使用して DirectX* 12 MiniEngine の派生バージョンに実装されています。派生バージョンは GitHub* GameTechDev (英語) から入手でき、サンプルのビルドと実行手順が含まれています。

概要

チェッカーボード・レンダリング (CBR) は、シェーディングを大幅に軽量化し、視覚効果品質への影響を最小に抑えて、高解像度のピクセルを生成する手法です。CBR は、アンチエイリアシングに対する最新の後処理アプローチと完全に互換性があり、フォワード・レンダリング・パイプラインと遅延レンダリング・パイプラインのどちらにも実装できます。

このホワイトペーパーでは、この分野における研究と目的について述べ、実装とソリューションに固有のいくつかの問題を説明します。そして、実装と結果の詳細を提供し、パフォーマンスと品質の両方について考察します。最後に、今後の動向と次のステップを紹介します。サンプルコードは、ここで紹介する手法を実証し、フォワード・レンダリングと遅延レンダリング・パイプラインの両方のベースとなる実装を提供します。ここでは、統合グラフィックス処理ユニット (GPU) に注目しますが、フォワード・レンダリングと遅延レンダリングの実装はどちらも AMD* 製と NVIDIA* 製のディスクリート GPU 上で検証済みです。

目的

最近、Sony* PS4 Pro でアップスケーリングのためチェッカーボード・レンダリングが採用されたことから、インテル® インテグレーテッド・グラフィックスでもこの手法を調査する価値があると考えました。特に、高い解像度向けに設計されているため、統合 GPU 上でそのまま実行できないコンテンツのレンダリングに関する問題の解決に役立てたいと思いました。チェッカーボード・レンダリングを使用することで、動的な解像度のレンダリング (DRR: Dynamic Resolution Rendering) のみを使用するよりも高品質なソリューションを提供できるという仮説を立てました。

もう 1 つの目的は、さまざまな解像度で統合 GPU とディスクリート GPU の両方に利点をもたらす、簡単な非侵入型のハードウェアに依存しない実装を作成することでした。サンプルコードは、開発者の品質対パフォーマンスの要求を満たすカスタム・ソリューションを提供する一方で、可能な限り「ドロップイン」で使用できるようにしています。ゲーム開発者は、チェッカーボード・レンダリングにより高品質なビジュアルを提供しつつ、GPU レンダリング・パフォーマンスが異なるプラットフォームをサポートすることで、プレーヤーを増やすことができます。この記事の執筆時点では、ソースコードを含むチェッカーボード・レンダリング・サンプルは、ほかに公開されていません。

先行研究

グラフィックス・アーキテクトは、フレームに表示する処理量をプレーヤーに気付かれずに減らすことでフレームレートを向上する方法を常に考えています。多くの手法は、ジオメトリーの詳細レベルを下げることに注目しています。しかし、ここではそのような手法を適用した後にシェーディング量を減らすため、同じような課題に直面した他の開発者が注目するチェッカーボード・レンダリング (CBR) と呼ばれる手法を採用しています。ここでの目的は CBR における先行研究とはやや異なります。Sony* PS4 Pro は、中程度からハイエンドのアセットとレンダリング・アーキテクチャーを対象とし、1080p (1920 x 1080) から 4K (3840 x 2160) ディスプレイへアップスケーリングすることを目的としていました。ここでは、1080p 向けのターゲットを 540p (960 x 540) でレンダリングし、CBR 手法を使用して 1080p にスケールアップします。CBR アルゴリズムについて詳しく説明する前に、CBR と組み合わせ可能ないくつかの DRR に関する先行研究について述べます。実際、少なくとも 2 つの CBR ソリューションは、CBR に加えて DRR を使用しています。

CBR アルゴリズムに関するホワイトペーパーと公開されているソースコードはそれほどありません。先行研究の調査では、研究論文や公開されているサンプルコードよりも、GDC (英語) や SIGGRAPH (英語) などの技術会議のプレゼンテーションを参照する必要がありました。

動的な解像度のレンダリング

Doug Binks は、「Dynamic Resolution Rendering (動的な解像度のレンダリング)」(英語) の記事でアップスケーリングの問題に対する 1 つのソリューションを提示しました。DRR 手法は、GPU ワークロードに応じて動的に解像度を調整します。これはうまく動作しますが、記事で指摘されているように、レンダーターゲットの解像度でしかレンダリングできません。このシナリオでは、アップサンプリングのアーティファクトがすべて適用されます。例えば、レンダーターゲットの解像度では可視性と色情報しか利用できず、アップスケーリングの品質が制限されます。ジッタリングしたサンプリングを使用するテンポラル・アンチエイリアシングなどの手法は、パフォーマンスと引き換えに品質を高めます。このホワイトペーパーでは、これに関連するトレードオフについても述べます。

Sony* PS4 Pro と CBR

CBR は最初に、Sony* PS4 Pro 上で実行されるコンテンツをアップスケールして 4K (3840 x 2160) 解像度 (英語) で表示する方法を模索する際に注目を集めました。多くのエンジンではこのピクセル・スループットを達成できず、新しいハードウェアのレンダリング性能で最高の画質を実現する取り組みが行われました。Mark Cerny 氏は、Richard Leadbetter 氏の記事で取り上げられている 13 のゲーム中 9 のゲームが CBR を使用していると指摘しています。「デイズゴーン」、「コール オブ デューティ インフィニット・ウォーフェア」、「ライズ オブ ザ トゥームレイダー」、「ホライゾン ゼロ ドーン」などのタイトルは最大 2160p にレンダリング可能で、「ウォッチドッグス 2」、「Killing Floor 2」、「inFAMOUS」、「Mass Effect: Andromeda」はすべて CBR を使用しており 1080p です。Sony* はサンプルコードを提供して、ゲーム開発者が CBR を簡単に使用できるようにしました。CBR を使用しない「シャドウ・オブ・モルドール」などの一部のタイトルは、上記のような動的解像度を使用していました。「デウスエクス マンカインド・ディバイデッド」は CBR を使用しているだけでなく、シーンの複雑さに応じて 1800p と 2160p のフレームバッファーを切り替えていました。ここでの実装が大きく異なる点は、現在のインテル® 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、デプスバッファーと同じ解像度でオブジェクト ID をサポートしていないことです。

「レインボーシックス シージ」の CBR

2016 年に、Ubisoft* の Jalal El Mansouri 氏は「レインボーシックス シージ」で使用されている CBR 手法について説明 (英語) しました。当初の目標は、720p (1280 x 720) のコンソールで 60fps を達成し、4K の PC で適切なフレームレートを達成することであり、Ubisoft* はその手法を模索していました。CBR は、Ubisoft* がテンポラル・インターレース・レンダリングで直面していたテンポラル・エイリアシングの問題を回避する方法の 1 つとして調査されました。チェッカーボードの実装では、MSAA サンプルポイントごとに色と z 値を持つ 2x MSAA を 1/4 のサイズのレンダーターゲットでレンダリングしました。また、リゾルブフェーズで発生するサンプリング・アーティファクトの一部を削除する再サンプリング (ティージングと呼ばれる) とともに、テンポラル・アンチエイリアシングをリゾルブシェーダーに統合しました。興味深いことに、パトカーのライトの点滅など、テンポラルの違いが大きいインエンジン・レンダリング手法は少なくとも 2 フレームを必要とします。Ubisoft* の手法は、我々の実装に最も似ています。

DECIMA* エンジンの CBR

SIGGRAPH 2017 において、Guerilla* Games の Giliam de Carpentier 氏と小島プロダクションの小島秀夫氏によって DECIMA* エンジン (英語) で使用されている CBR の詳細が発表されました。レンダリングとほとんどの後処理は、独自の実装のチェッカーボード解像度で行われています。2x MSAA レンダーターゲットのサンプルポイントは、高速近似アンチエイリアシング (FXAA: Fast Approximate Anti-Aliasing) 後処理が期待する実際のピクセル位置と一致しないため、画像を 45° 回転してサンプルポイントを通常のグリッドにアライメントしています。これにより、通常のグリッドと後処理のアンチエイリアシングが期待するラスタライズ位置が一致します。Sony* PS4 Pro では、2ms 以下で非常に高品質な結果が得られました。

Frostbite* エンジンの CBR

Frostbite* は、「バトルフィールド 1」と「Mass Effect: Andromeda」で CBR を使用しています。Graham Wihlidal 氏は、GDC 2017 でその手法を Frostbite* エンジン (英語) にも適用したことを発表しました。プレゼンテーションでは、レンダリング・パイプラインの大部分の詳細と CBR をエンジンに統合する際に直面した課題について説明されています。実装には、テンポラル・アンチエイリアシング、DRR、高品質アンチエイリアシング (EQAA: Enhanced Quality Anti-Aliasing) が組み込まれています。

手法の概要

我々の実装の技術的詳細と 1/4 解像度の 2x MSAA レンダーターゲットの使用法を説明する前に、従来のアップスケーリングについて理解しておくことが重要です。最初に、標準のフル解像度のレンダリングでピクセルカバレッジとピクセル色の関係を確認します。次に、半解像度のレンダーターゲットから構築して最終結果に近づけて、複数の半解像度のレンダーターゲットを使用してテンポラル再構築を行います。半解像度のレンダーターゲットの問題を説明することで、1/4 解像度の 2x MSAA レンダーターゲットを採用した理由を示します。

フル解像度のレンダリング

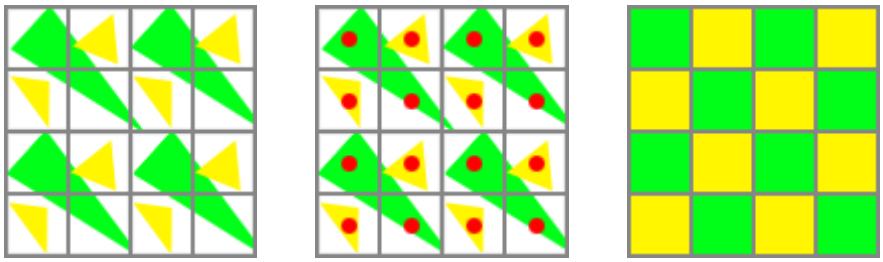


図 1. 三角形の範囲に応じてピクセルがシェーディングされます。各グリッドセルは、シェーディングされるレンダー・ターゲット・ピクセルを表します。左: レンダーターゲットにラスタライズされる三角形。
中央: カバレッジテスト。赤色の点は各ピクセルのサンプルカバレッジ位置を表します。
右: ピクセル色は、カバレッジテストの結果に応じてシェーディングされます。

図 1 では、三角形とピクセルのカバレッジテストの結果に応じてピクセルの色が決まります。左の図はラスタライズされる三角形を示し、中央の図は各ピクセルのサンプルカバレッジ位置を示します。右の図にはカバレッジテストの結果が含まれます。ラスタライズされた三角形がピクセルの中央をカバーしている場合、そのピクセルは三角形の色でシェーディングされます。

アップスケーリングを使用する半解像度レンダリング

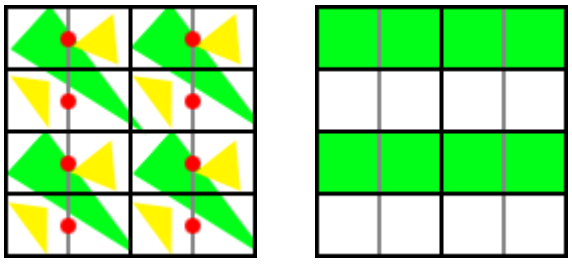


図 2. サンプルカバレッジ位置によってはシェーディング情報が失われます。灰色の線はフル解像度のレンダーターゲットのピクセルを表し、黒色の線は半解像度のレンダーターゲットのピクセルを表します。
左: レンダーターゲットにラスタライズされる三角形。赤い点は、各半解像度ピクセルのサンプルカバレッジ位置を示します。緑色の三角形のみがカバレッジテストにパスしていることが分ります。
右: カバレッジテストの結果に応じてピクセルがシェーディングされます。黄色の三角形はカバレッジテストにパスしないため、アップスケーリング中にレンダーターゲットが正確に再構築されません。

アップスケーリング手法は、解像度を下げてシェーディングを軽減することに注目しているため、シェーディングされるピクセルが少なくなります。図 2 に示すように、三角形がピクセルのカバレッジテストにパスしないと、シェーディング情報は失われます。左の図では、黒色の線は半解像度のレンダーターゲットのピクセルを示します。黄色の三角形はピクセルの中央をカバーしていないため、カバレッジテストにパスしません。つまり、右の図に示すように、フル解像度のレンダーターゲットを正確に再構築することはできません。

テンポラルな再構築を使用する半解像度レンダリング

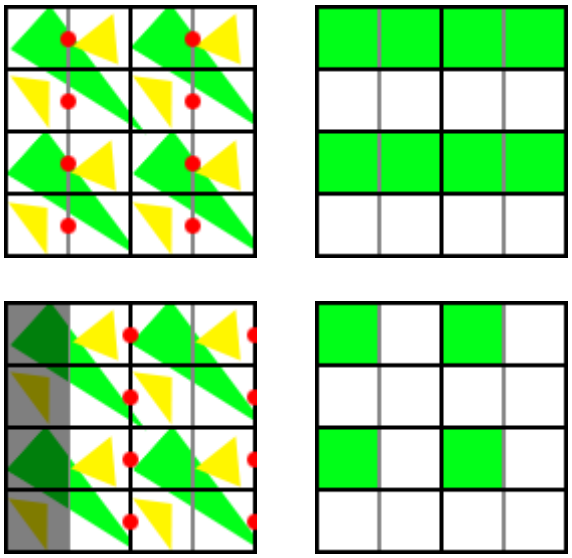


図 3. 2 つのフレームに基づくテンポラルな再構築。サンプルカバレッジ位置によってシェーディング情報が失われます。灰色の線はフル解像度のレンダーターゲットのピクセルを表し、黒色の線は半解像度のレンダーターゲットのピクセルを表します。
左から右へ: レンダーターゲットにラスタライズされる三角形。赤い点は、半解像度レンダーターゲットのサンプルカバレッジ位置を表します。
フレーム N-1: ピクセル色は、カバレッジテストの結果に応じてシェーディングされます。緑色の三角形のみがカバレッジテストにパスしていることが分ります。
フレーム N: ビューポートは右方向へジッタリングされます。サンプルカバレッジ位置が原因で、すべての三角形がカバレッジテストにパスしません。
再構築されたフレーム: アップスケーリング中にレンダーターゲットを正確に再構築することができません。

解像度を抑えたままより多くのピクセルをシェーディングするため、以前のフレームのシェーディング・データを利用するテンポラルな手法が導入されました。これらの手法では、各フレームはレンダーターゲットを変更し、ビューポートをジッタリングします。そして、レンダーターゲット N-1 と N (以前と現在のレンダーターゲット) からフル解像度のレンダーターゲットを再構築します。図 3 の最初の 2 つの図はフレーム N-1 のシェーディングです。3 つ目の図は、ビューポートを右へジッタリングします (フレーム N)。ピクセルカバレッジ位置は、黄色と緑色の両方の三角形を失っていることが分ります。右の図は、前述の非テンポラルな手法と同様に、フル解像度のレンダーターゲットが正確に再構築されないことを示しています。

2x MSAA よるサンプルカバレッジ位置の改善

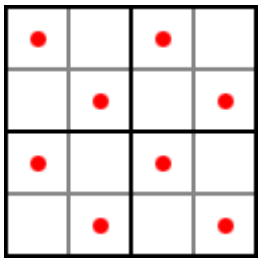


図 4. 2x MSAA シェーディング情報。灰色の線はフル解像度のレンダーターゲットのピクセルを表し、黒色の線は 1/4 解像度のレンダーターゲットのピクセルを表します。赤い点は、2x MSAA のサンプルカバレッジ位置を表します。

理想的なシナリオでは、低解像度でレンダリングしビューポートをジッタリングすることで、フル解像度と同じサンプルカバレッジ位置に配置できます。幸いにも、2x MSAA の標準のサンプルカバレッジ位置は、これらの場所にあります。ピクセルの中央ではなく、各 1/4 解像度ピクセルの第 2 象限と第 4 象限にサンプルカバレッジ位置が配置されます。

2x MSAA とテンポラル再構築を使用するサンプルカバレッジ位置

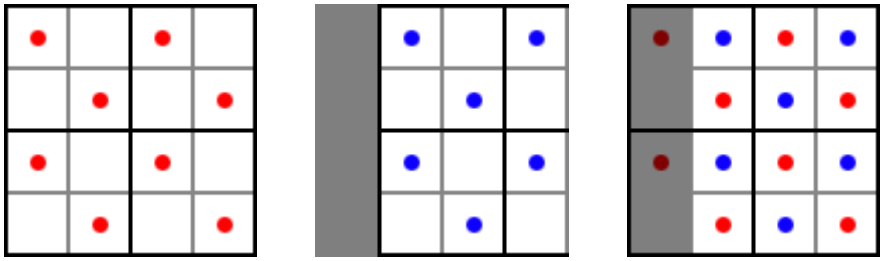


図 5. 2x MSAA のシェーディング情報。灰色の線はフル解像度のレンダーターゲットのピクセルを表し、黒色の線は 1/4 解像度のレンダーターゲットのピクセルを表します。
左: 赤い点は、フレーム N-1 時点の 2x MSAA のサンプルカバレッジ位置を表します。
中央: 青い点は、フレーム N 時点の 2x MSAA のサンプルカバレッジ位置 (1 ピクセル右へジッタリング済み) を示します。
右: フレーム N-1 と N を一時的に組み合わせることで、MSAA を使用しないフル解像度のレンダリングとほぼ同じカバレッジが得られます。ピクセルの最初の列は、ビューポートのジッタリングのために再構築されません。

2x MSAA サンプルカバレッジ位置を一時的にジッタリングして、フル解像度のレンダリングの (最初と最後のピクセル列を除く) すべてのカバレッジ位置とオーバーラップさせることができます。図 5 に示すように、内部の場所 (黒色の枠線内の灰色の線で区切られた領域) は、2x MSAA サーフェスのサンプル位置です。左の図は、2x MSAA サーフェスの 2 つのサンプル位置を示しています。次の図は、1/4 サイズの画像をハーフピクセル右へシフトしたものであり、最終的なレンダーターゲットではフルピクセルに相当します。右の図は、同じ 2x MSAA サンプルパターンを使用して、2 つの半解像度のフレームの結果を組み合わせることで、完全なフルサイズのレンダーターゲットのサンプルポイントを生成できることを示しています。この手法で作成されるチェッカーボード・パターンでは、フル解像度のレンダーターゲットがほぼ完全に再構築されます。

2x MSAA とテンポラル再構築を使用する正確なフル解像度の再構築

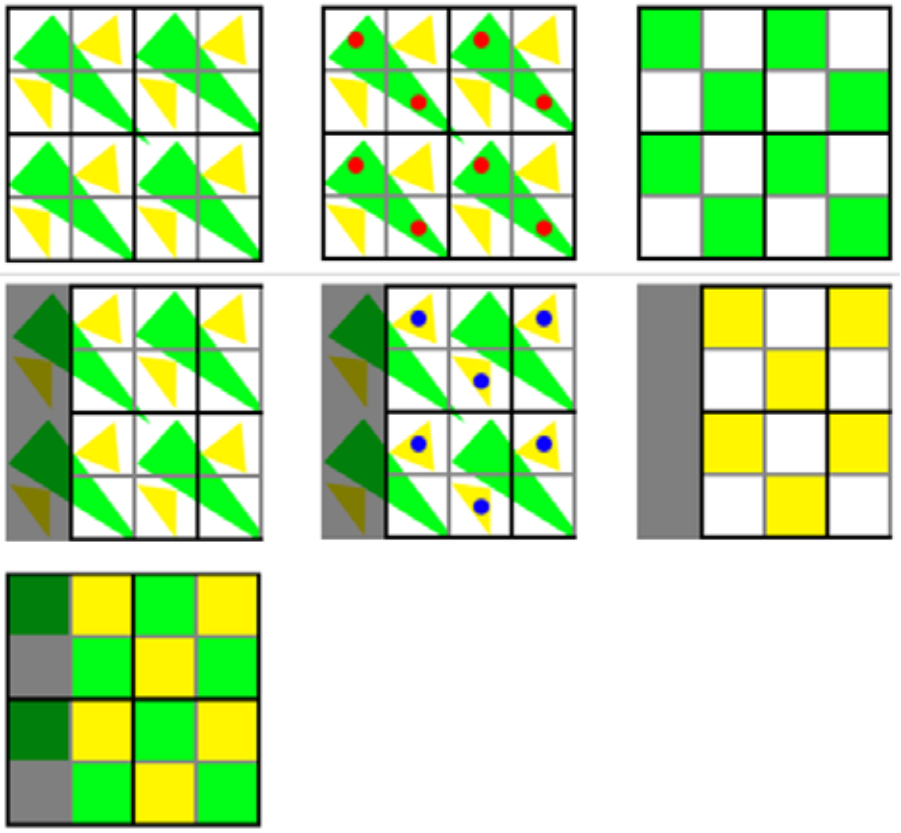


図 6. 2x MSAA とテンポラル再構築を使用する正確なフル解像度の再構築。灰色の線はフル解像度のレンダーターゲットのピクセルを表し、黒色の線は 1/4 解像度のレンダーターゲットのピクセルを表します。
上段左: レンダーターゲットにラスタライズされる三角形。上段中央: フレーム N-1。
赤い点は 2x MSAA サンプルカバレッジ位置を示します。
上段右: 緑色の三角形はカバレッジテストにパスし、フレーム N-1 に応じてシェーディングされます。
中段左: ビューポイントはフル解像度の 1 ピクセル分右へジッタリングされます。
中段中央: フレーム N。青い点は 2x MSAA サンプルカバレッジ位置を示します。
中段右: 黄色の三角形はカバレッジテストにパスし、フレーム N に応じてシェーディングされます。
下段左: フレーム N-1 と N を組み合わせることで、フル解像度のレンダーターゲットがほぼ完全に再構築されます。ピクセルの最初の列は、ビューポートのジッタリングのため再構築されません。

図 1 のオリジナルのフル解像度のレンダリングを思い出してみてください。2x MSAA テンポラル再構築の概念を利用することで、図 6 に示すようにフレーム N-1 と N からフル解像度のレンダーターゲットを再構築できます。上段では、2x MSAA サンプリング位置によって緑色の三角形が正確にシェーディングされます。中段では、ビューポイントがフル解像度の 1 ピクセル分右へジッタリングされ、ジッタリングされたビューポイントと 2x MSAA サンプリング位置によって黄色の三角形が正確にシェーディングされます。最後の図 (下段) は、再構築されたレンダーターゲットです。フル解像度のレンダーターゲットとほぼ同じであることが分ります。

CBR をサポートするためのレンダラーの変更

レンダーターゲットを変更する目的が判明したところで、このセクションでは 2 つの単純な標準パイプライン (フォワードと遅延) の変更について説明し、両方のケースで CBR を使用する方法を示します。フォワードレンダラーのほうが簡単ですが、業界では遅延レンダラーのほうが一般的に使用されています。ここでは、先にフォワードレンダラーについて説明し、次に遅延レンダラーについて説明します。

CBR をサポートするためのフォワード・レンダリングの変更

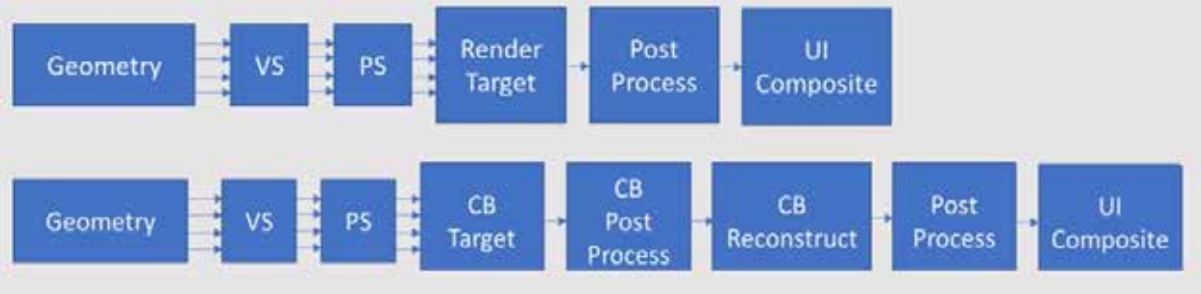


図 7. 上: フォワード・シェーディング・パイプラインの概要。下: CBR を実装したフォワード・シェーディング・パイプライン。

フォワード・レンダリング・パイプラインの変更は簡単です。ここでは、完全なサンプルコードを提供します。2 つの 1/4 解像度のレンダーターゲットを交互に処理するように従来のフォワード・パイプラインを変更します。チェッカーボード結果と CBR フェーズの最終ステージには、最終的なフルサイズのレンダーターゲットが再構築されるオプションの後処理があります。図 7 は、これらの変更の概略図です。結果は、以下の図 8 で確認できます。

図 8: 2x MSAA バッファーのフレーム N-1 とフレーム N。シェーディングがチェッカーボード形式で行われています。左から右へ: フレーム N-1、ビューポートがジッタリングされた フレーム N、再構築されたレンダーターゲット

フォワード・レンダリング・アルゴリズム

変更後のレンダリング・パイプラインのステップは以下のとおりです。

1. 1/4 解像度 (幅 1/2、高さ 1/2) の 2 つのカラーバッファーとデプスバッファー (フレーム N-1 と N 用) を作成します。これらのバッファーは、ピクセルごとではなくサンプルごとのシェーディングで 2x MSAA を使用するように設定します。
    1. テクスチャーの解像度を上げるため、MIP LOD Bias をテクスチャーに適用します。(MIP は「小さな空間内の多数のもの」を意味するラテン語の Multum in Parvo を表し、LOD は詳細レベルを意味する Level Of Detail を表します。)Direct3D* 12 では、3D シーンのパス中に -0.5f の D3D12_SAMPLER_DESC MipLODBias を使用します。
2. フレーム N-1 をレンダリングします。
3. 1/4 解像度でビューポートをフル解像度の 1 ピクセル分右へジッタリングしてフレーム N をレンダリングします。
4. 両方のフレームを使用してフル解像度のレンダーターゲットを再構築するチェッカーボード再構築シェーダーを実行します。

CBR をサポートするための遅延レンダリング・パイプラインの変更

遅延パイプラインの CBR はフォワード・レンダリングの CBR よりも複雑です。フォワード・パイプラインでは、不透明オブジェクトと透明オブジェクトはどちらもレンダリング時にソートおよびシェーディングされます。前述のとおり、これらのオブジェクトをチェッカーボード・バッファーへシェーディングするにはレンダーターゲットを変更するだけで済みます。ここでは、遅延レンダリング・パイプラインの概要を示してから、CBR をサポートするための変更について説明します。

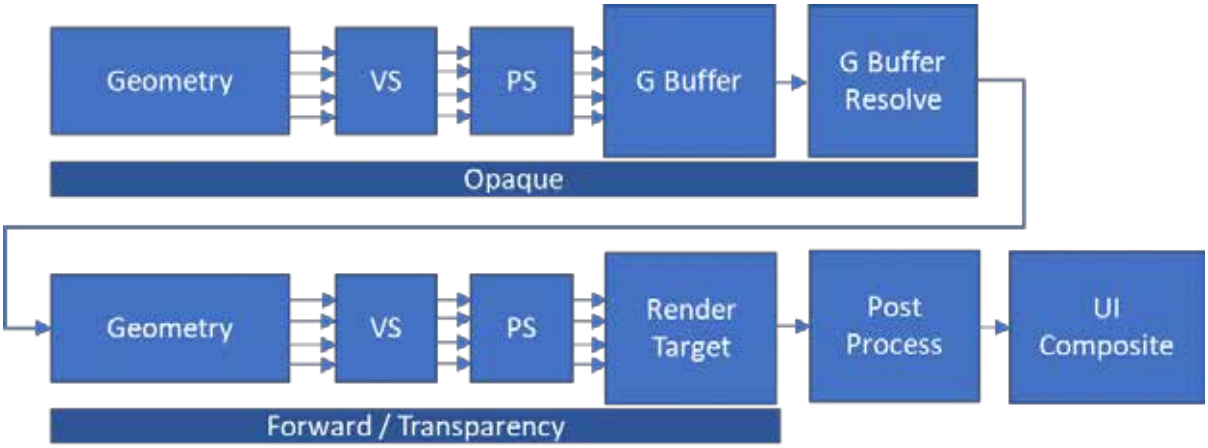


図 9. 遅延シェーディング・パイプラインの概略図。

遅延パイプラインでは、ピクセル色は次の 3 つのステップで決定されます。

1. マテリアル情報は G-Buffer と呼ばれる別のレンダーターゲットにレンダリングされます。この例では、G-Buffer には albedo、normal、specular の 3 つのレンダーターゲットがあります。
2. 次に、G-Buffer をリゾルブステップで使用します。G-Buffer のリゾルブは、レンダーターゲット (albedo、normal、specular) を使用してピクセルのシェーディングの色を決定し、結果をレンダーターゲットに書き込みます。
3. 最後に、フォワード・レンダリングに適したオブジェクト (例えば、透明オブジェクト) がレンダーターゲットに直接シェーディングされます。

CBR をサポートするための変更

ここで紹介するチェッカーボード遅延パイプラインでは、最初にフェーズ 1 を変更します。図 10 に示すように、2x MSAA を有効にしてフル解像度の 1/4 の解像度で G-Buffer のレンダーターゲットがチェッカーボード・バッファーとして作成されます。

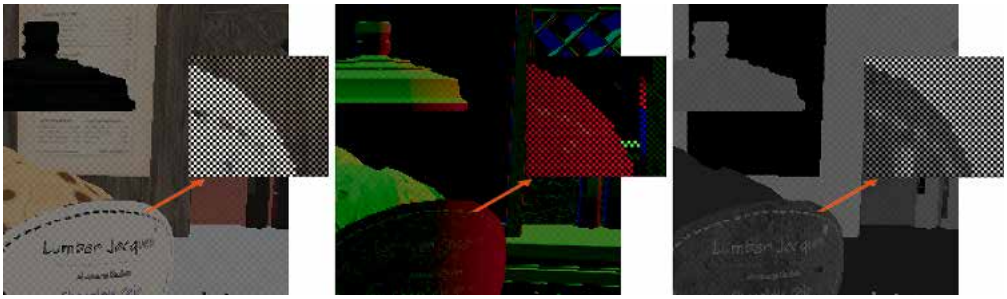


図 10. 各 G-Buffer レンダーターゲットは 2x MSAA チェッカーボード・バッファーとして書き込まれます。

フェーズ 1 では、チェッカーボード・バッファーがレンダリングされます。フェーズ 2 では、G-Buffer リゾルブ シェーダーは 2x MSAA を認識し、ピクセルごとに 2 つのシェードを格納可能なレンダーターゲットに出力する必要があります。ここで紹介する手法は、チェッカーボードと高さは同じで幅が 2 倍の非 MSAA レンダーターゲットを作成します。これにより、一意のテクセルで各サンプル位置をシェーディングして格納できます。このテクスチャーをシェード・リゾルブ・ターゲット (SRT: Shade Resolve Target) と呼びます。図 11 に例を示します。

図 11. SRT は 2x MSAA ターゲットの 2 倍の幅があり、各 2x MSAA サンプルのシェーディングされた結果を保持します。

フォワード・シェーディング・ステップは解決済みの G-Buffer ターゲットに直接ブレンドできるため、フェーズ 3 も変更する必要があります。しかし、次の 2 つの問題が妨げとなります。

1. SRT は、2x MSAA ターゲットの 2 倍の幅があります。
2. CBR を使用する透明オブジェクトをシェーディングしたいと考えていますが、SRT は MSAA ターゲットではありません。

この問題を解決するため、図 12 に示すように、フォワード・オブジェクトを別の 2x MSAA バッファー (チェッカーボード・フォワード・バッファー (CFB)) にレンダリングします。

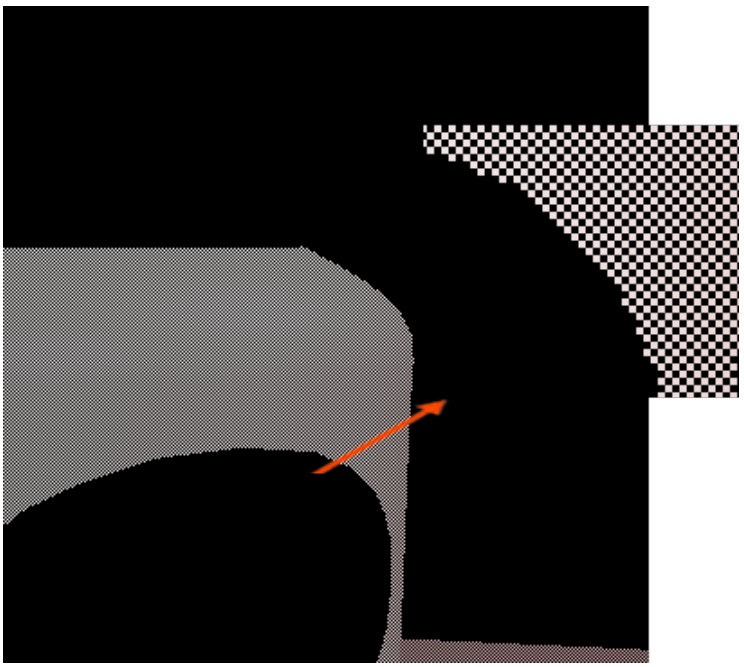


図 12. CFB はすべてのフォワード・レンダリング済みオブジェクトとその合計アルファ値を格納します。

チェッカーボード再構築の最終ステップでは、フレーム N-1 とフレーム N の SRT と CFB バッファーを読み取り、それらのシェーディングを使用してフル解像度のレンダーターゲットを再構築します (図 13)。

図 13. 遅延シェーディング実装のチェッカーボードによって再構築されたレンダーターゲット

最後に、パイプラインの変更の概略を図 14 に示します。不透明オブジェクトと透明オブジェクトのレンダリング・フェーズとチェッカーボード再構築が追加されています。

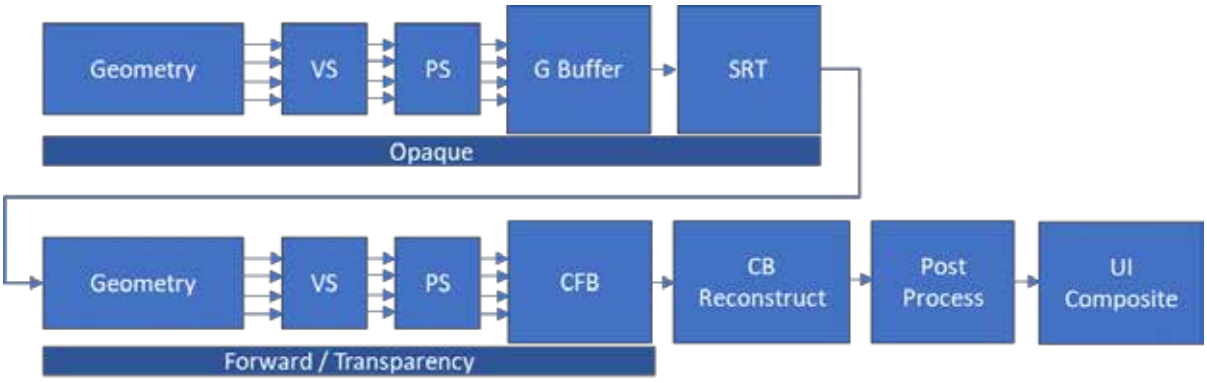


図 14. 変更後の CBR 対応の遅延レンダリング・パイプライン

モーションのシェーディング

実際の環境では、フル解像度のレンダーターゲットの再構築はさらに困難です。ゲームは静的ではなく、各フレームには多くのモーションが含まれます。チェッカーボード再構築ステップは、シェーディングされた要素のピクセル位置が変化することを考慮する必要があります。また、必要なシェーディング情報が以前のフレームで隠れていたため、不足している可能性があります。

モーションベクトルの使用をチェッカーボード再構築パスに追加することが、この問題を解決する最初のステップです。モーションベクトルは、再構築ステップ中にフレーム間の移動の追跡とフレーム N-1 のピクセル・ルックアップの調整に使用されます。図 15 に例を示します。このサンプルコードでは、モーションベクトルはデプスバッファーから取得します。ただし、これはスタティック・オブジェクトの場合のみです。実際のシナリオでは、ダイナミック・オブジェクトのモーションベクトルを提供する必要があります。ピクセルごとのモーションベクトルを緑色のチャネルにマップした画像を図 16 に示します。

ダイナミック・オブジェクトのモーションベクトルを適用するには注意が必要です。フレーム N-1 の各ピクセルに必要なモーションベクトルは、現在のフレーム N ではレンダリングされていません。このモーションベクトルを取得する方法はいくつかあります。シナリオに最適なソリューションの選択はレンダラーに任せます。

- 不足しているモーションベクトルは、フレーム N にある周辺の現在のモーションベクトルに基づいて推定できます。ほとんどのケースはこれで対応できますが、遅い動きのアルファテスト・オブジェクト (わずかに揺れる植物など) ではアーティファクトが発生する可能性があります。
- モーション・ベクトル・パスはビューポートをフレーム N-1 のジッタリングに設定して、現在のモーションベクトルをレンダリングします。このソリューションは、フレーム N-1 のモーション・ベクトル・バッファーを最新のデータで更新します。さらに、エンジンにオブジェクト ID の概念がある場合、ダイナミック・オブジェクトのみをサブミットする必要があります。チェッカーボード再構築は、オブジェクト ID が一致する場合はモーションベクトルをサンプリングし、そうでない場合はデプスバッファーから取得できます。

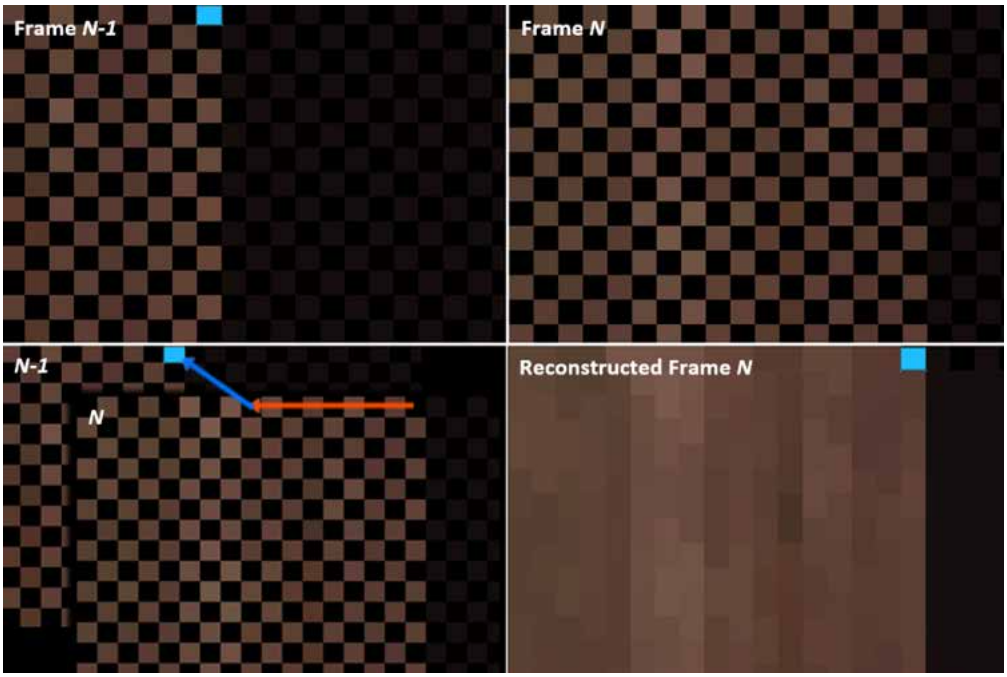


図 15. 再構築ステップ中のモーションベクトルの使用法。水色の四角は単一のピクセルを分かりやすくハイライト表示したものです。
左上: フレーム N-1。
右上: フレーム N。カメラが数ピクセル左へ移動しています。
左下: フレーム N がフレーム N-1 にオーバーレイされています。再構築ステップは水色のピクセル位置の画像を処理しています。赤色の矢印は、フレーム N-1 とフレーム N 間のモーションを示しています。これは (青色の矢印で示す) フレーム N-1 のテクセルのオフセットとして使用されます。
右下: フレーム N-1 とフレーム N を基に再構築されたレンダーターゲット。

図 16. デプスバッファーから取得したピクセルごとのモーションベクトル。カメラは上方向 (正の Y) に移動し、ピクセルごとの Y モーションがレンダーターゲットの緑色のチャネルに反映されます。このシナリオでは、前景オブジェクトのほうがフレーム間のモーションベクトルが大きくなります。

シェーディング情報の不足

モーションベクトルによって再構築シェーダーが以前のフレームに存在しないシェーディング情報をポイントした場合、より複雑な問題が発生します。図 17 の例について考えてみます。

- フレーム N-1 では、ワインボトルの後ろにカウンターの椅子が隠れています。
- フレーム N では、カメラが左へ移動し、カウンターの椅子が表示されています。

図 17. 左: フレーム N-1。ワインボトルの後ろにカウンターの椅子が隠れています。
右: フレーム N。カメラが左へ移動し、カウンターの椅子が表示されます。

カウンターの椅子のシェーディングのリゾルブでは、再構築シェーダーは本質的に「フレーム N-1 のカウンターの椅子のシェーディングは X ピクセル右へ移動したもの」として右方向へのモーションベクトルを読み取ります。しかし実際には、カウンターの椅子はワインボトルの後ろに隠れており、再構築ステップ中にそれらのピクセルをブレンドすると、図 18 のようなアーティファクトが発生します。これを検出し、チェッカーボード再構築の計算を調整する必要があります。

図 18. 赤枠はカウンターの椅子のリゾルブで表示されるアーティファクトをハイライト表示しています。この問題は、フレーム N-1 で必要なシェーディング情報がワインボトルによって隠されているために発生します。

ここでは、不足しているシェーディング情報を検出して解決する 2 つの簡単なソリューションを示します。1 つ目のアプローチは、以下のコードブロック 1 に示すように、フレーム N-1 と N のリニアデプス値を比較します。デプス値の差分が最小しきい値を超える場合、図 19 のようにシェーディング情報が隠されていると仮定します。このホワイトペーパーでは、これをシェーディング・オクルージョン確認 (CSO: Check Shading Occlusion) ステップと呼びます。

2 つ目のアプローチはより基本的であり、1/4 解像度のテクセルを超える移動では常にシェーディング情報が隠されていると仮定します。ここではこれをシェーディング・オクルージョン仮定 (ASO: Assume Shading Occluded) ステップと呼びます。2 つ目のアプローチは最初のアプローチよりも精度は下がりますが、サンプル数が少なくて済みます。移動しないでわずかに変化したシェーディングと 1/4 解像度のテクセルを超えて移動したシェーディングの視覚的な結果は同一です。図 20 は、CSO と ASO の相違点の例です。CSO ではシェーディング情報が取得され、ASO では正しくないピクセル色が推定されています。オブジェクト ID バッファーや色の比較などを使用するより複雑なソリューションを実装することも可能です。必要に応じて、パイプラインに合わせてより複雑なソリューションを採用できます。

```
// If there is pixel motion between frames
if ( qtr_res_pixel_delta.x || qtr_res_pixel_delta.y )
{
   float4 current_depth;

   // Fetch the interpolated depth at this location in Frame N
   current_depth.x = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Left ], cardinal_quadrants[ 1 ] );
   current_depth.y = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Right ], cardinal_quadrants[ 1 ] );
   current_depth.z = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Down ], cardinal_quadrants[ 0 ] );
   current_depth.w = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Up ], cardinal_quadrants[ 0 ] );

   float current_depth_avg = (projectedDepthToLinear( current_depth.x ) +
                              projectedDepthToLinear( current_depth.y ) +
                              projectedDepthToLinear( current_depth.z ) +
                              projectedDepthToLinear( current_depth.w )) * .25f;

   // reach across the frame N-1 and grab the depth of the pixel we want
   // then compare it to Frame N&amp;apos;s depth at this pixel to see if it&amp;apos;s within
range
   float prev_depth = readDepthFromQuadrant( prev_qtr_res_pixel, quadrant_needed );
   prev_depth = projectedDepthToLinear( prev_depth );

   // if the discrepancy is too large assume the pixel we need to
   // fetch from the previous buffer is missing
   float diff = prev_depth - current_depth_avg;
   missing_shading = abs(diff) >= tolerance;
}
```

コードブロック 1. CSO **ステップは、フレーム** N-1 **のシェーディング情報が不足しているかどうか判断します。フレーム** N **の該当ピクセルの推定デプス値とフレーム** N-1 **の該当ピクセルの実際のデプス値を比較します。差異が経験的な許容値よりも大きい場合、シェーディング情報が隠されていると仮定できます。**

図 19. **左**: **フレーム** N-1 **とフレーム** N **の間にカメラがスクリーン幅の** 1/4 **を移動しています。黄色でハイライトされた領域は** CSO **によってシェーディング情報が不足していると判断されたピクセル。**
右: **隠されたシェーディング情報は現在のフレームの周辺のピクセルをブレンドして再構築されます。**

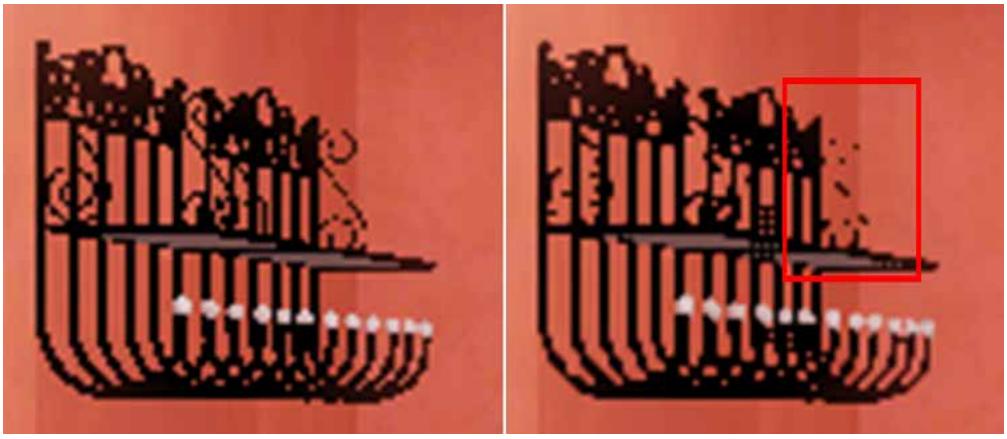


図 20. CSO と ASO を使用した場合のアーティファクト。デモ用にアンチエイリアシングなしで 500 倍に拡大した画像。
左: CSO はオクルージョンを仮定せず、モーションベクトルを使用してコートハンガーのシェーディングを取得します。
右: ASO はオクルージョンを仮定して、コートハンガーのシェーディングを推定します。壁のシェーディングが間違って使用されており、コートハンガーのシェーディングの一部が失われています (赤枠で示した部分)。

チェッカーボード再構築シェーダーの疑似コード

ここで使用したチェッカーボード再構築シェーダーの疑似コードは次のとおりです (コードブロック 2 は HLSL (High-Level Shading Language) の要約です)。

ピクセルごとに次の処理を行います。

1. ピクセルがフレーム N (最新のピクセル) でレンダリングされているか確認します。
    1. レンダリングされている場合は、フレーム N からサンプリングして終了します。
    2. そうでない場合は、フレーム N-1 のピクセルが必要になるためアルゴリズムを続行します。
2. モーションを確認します。
    1. モーションがない場合は、フレーム N-1 からサンプリングして終了します。
    2. そうでない場合は、フレーム N-1 のシェーディングのサブピクセル位置を特定します。
3. モーションベクトルを適用してフレーム N-1 のサブピクセル位置を特定します。
    1. カメラの移動によりフレーム N のサブピクセル位置が N-1 とオーバーラップする場合 (つまり、ジッタリング効果が取り消される場合)、シェーディング情報は利用できません。基本方向へのフレーム N のピクセルを使用してブレンドを実行して終了します。
    2. そうでない場合は、アルゴリズムを続行します。
4. CSO を使用する場合は次の処理を行います。
    1. (モーションベクトルをオフセットとして) フレーム N-1 のデプスをサンプリングします。
    2. フレーム N の周辺のデプスの平均を計算します (基本方位)。
    3. 2 つのデプスを比較します。
    4. デプス値の差分が最小しきい値を超える場合、必要なシェーディングは隠されていると仮定します。
5. シェーディングが隠されている場合、または ASO を使用する場合は、基本方向へのフレーム N のピクセルを使用してブレンドを実行します。
6. そうでない場合は、(モーションベクトルをオフセットとして) フレーム N-1 のサブピクセルをサンプリングして終了します。

```
// if the pixel we are writing to is in a MSAA
// quadrant which matches our latest CB frame
// then read it directly and we&apos;re done
if ( frame_quadrants[ 0 ] == quadrant || frame_quadrants[ 1 ] == quadrant )
   return readFromQuadrant( qtr_res_pixel, quadrant );
else
{
   // We need to read from Frame N-1

   ...

   // Get the screen space position this pixel was rendered in Frame N-1
   uint2 prev_pixel_pos = ...

   // Which MSAA quadrant was this pixel in when it was shaded in Frame N-1
   uint quadrant_needed = ...

   ...

   // if it falls on this frame (Frame N&apos;s) quadrant
   // then the shading information is missing
   // so extrapolate the color from the texels around us
   if ( frame_quadrants[ 0 ] == quadrant_needed ||
        frame_quadrants[ 1 ] == quadrant_needed )
      missing_shading = true;
   else if ( qtr_res_pixel_delta.x || qtr_res_pixel_delta.y )
   {
      // Otherwise we might have the shading information,
      // Now we check to see if it&apos;s occluded

      // If the user doesn&apos;t want to check
      // for occlusion we just assume it&apos;s occluded
      // and this pixel will be an extrapolation of Frame N&apos;s pixels around it
      // This generally saves on perf and isn&apos;t noticeable
      // because the shading will be in motion
      if ( false == check_shading_occlusion )
         missing_shading = true;
      else
      {
         ...

         // if the discrepancy is too large assume the pixel we need to
         // fetch from frame N-1 is missing
         float diff = prev_depth - current_depth_avg;
         missing_shading = abs(diff) >= tolerance;
       }
     }

     // If we&apos;ve determined the pixel (i.e. shading information) is missing,
     // then extrapolate the missing color by blending the
     // current frame&apos;s up, down, left, right pixels
     if ( missing_shading == true )
        return colorFromCardinalOffsets( qtr_res_pixel,
                                         cardinal_offsets,
                                         cardinal_quadrants );
     else
        return readFromQuadrant( prev_qtr_res_pixel, quadrant_needed );
}
```

コードブロック 2. **上記の疑似コードルーチンの** HLSL **コードの要約。クワドラントは** 1/4 **解像度のピクセル内の** MSAA **サンプル位置を参照します。**

視覚的な結果

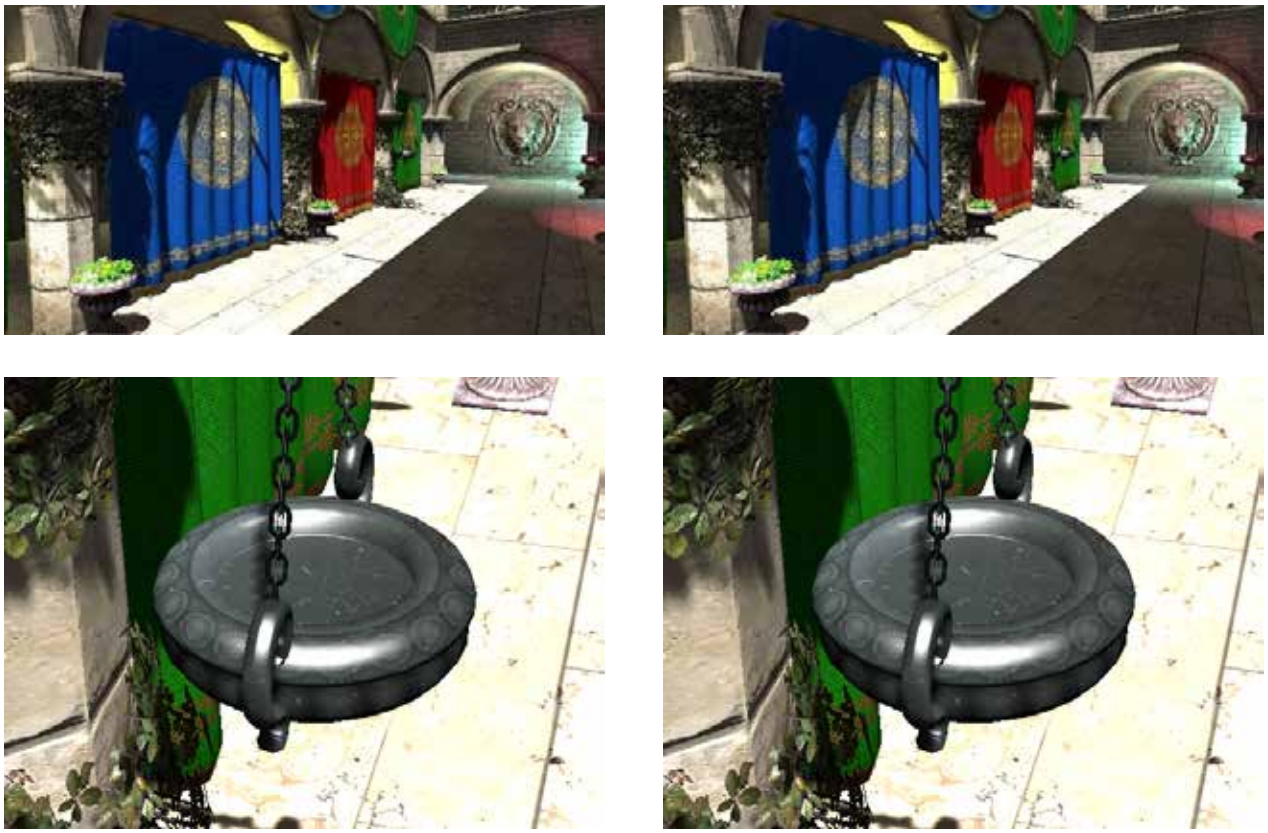


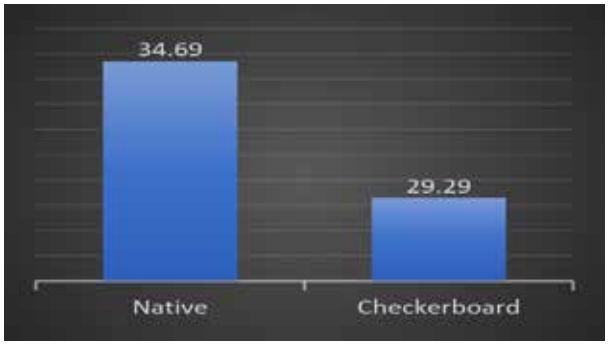
図 21. **左上**: アトリウムスポンザ宮殿モデルの廊下のフル解像度のレンダリング。
右上: チェッカーボード再構築を使用した廊下の 1/4 解像度のレンダリング。
左下: 鉢のフル解像度のレンダリング。
右下: チェッカーボード再構築を使用した鉢の 1/4 解像度のレンダリング。

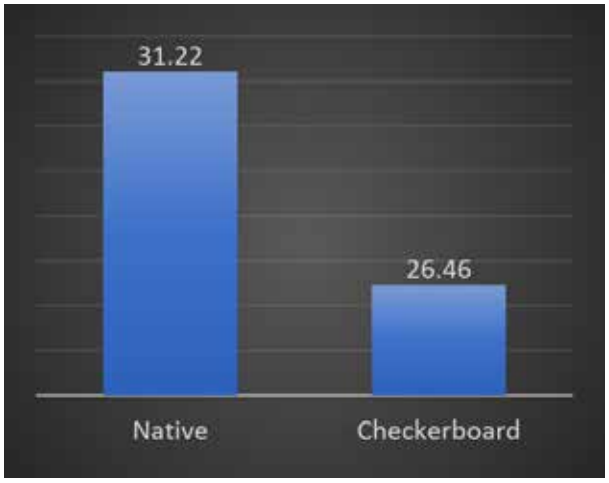
図 21 に示すように、静的なシーンや動きが複数のピクセルを横切るシーンでは、主観的にフル解像度と同等の結果が得られます。静的なシーンでは、レンダーターゲットの再構築中にテンポラルなシェーディング情報が不足することはありません。動きが複数のピクセルを横切るシーンでは、劇的なモーションによりフレーム間の一貫性が失われ、目は再構築中の推定エラーを認識しません。しかし、わずかな動きでは、従来のエイリアシングで見られるエッジクロールと同様に、色のコントラストが高いプリミティブ・エッジで視覚的なアーティファクトが発生します (図 22)。CMAA (英語)、FXAA (英語)、TAA (英語) を含む一般的なアンチエイリアシング・ソリューションは、これらのアーティファクトを排除または大幅に軽減します。

図 22. わずかなカメラの動き。紫色のピクセルは、フレーム N-1 でシェーディング情報が隠されており、推定が必要になります。これにより、追加のエッジクロールが発生する可能性があります。

パフォーマンス結果

すべての画像を 1080p でレンダリングして CBR とフル解像度のパフォーマンスを比較しました。図 23 に示すように、CBR は 1/4 解像度のテクセルに満たないモーションを含むシーンのフレーム時間を約 5ms 短縮しました。より大きなモーションのシーンでは、アトリウムスポンザ宮殿モデルのシーンをカメラでフライスルーして ASO と CSO 再構築ソリューションの両方をテストしました。ASO はフル解像度と比較してパフォーマンスが平均 15% 向上し (図 24)、CSO は平均 12% 向上しました (図 25)。CSO のパフォーマンス・ゲインがやや低い原因は、シェーディングの軽減によって利点が得られない (つまり、ピクセル・シェーディング・ステージでボトルネックが発生しない) フレームのオクルージョンの確認にコストがかかるためです。





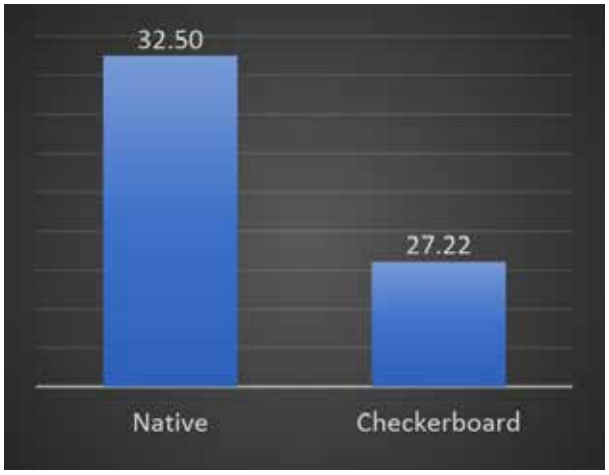


図 23. **後処理** (AA、SSAO **など**) **なしで** 1920 x 1080 **でレンダリングした** GPU **フレーム時間** (**ミリ秒**) **のグラフ。**

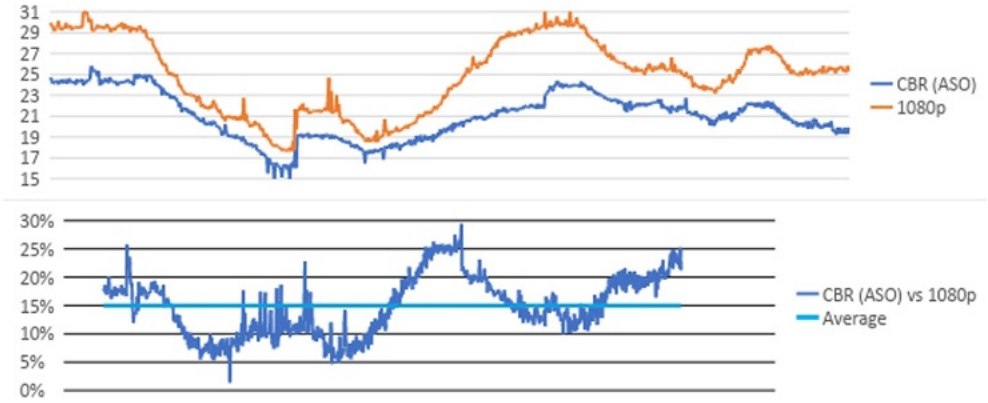


図 24. **ASO を使用した** CBR **と** 1080p **のパフォーマンスの比較。**
上: **ミリ秒単位の** GPU **フレーム時間 (値が小さいほうが良い)。**
下: CBR **と** 1080p **のパフォーマンス向上のパーセンテージ (値が大きいほうが良い)。**

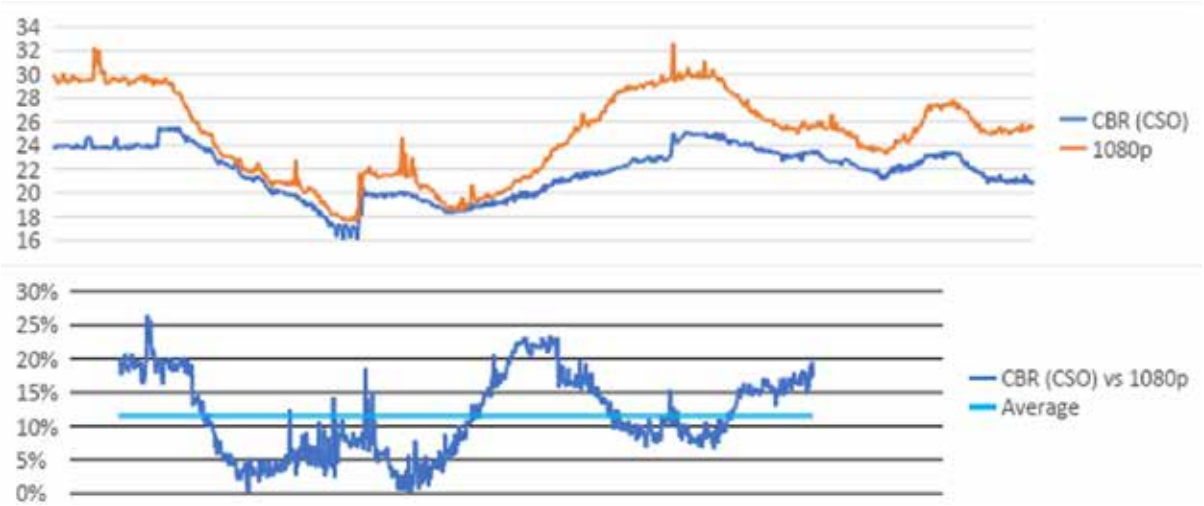


図 25. CSO **を使用した** CBR **と** 1080p **のパフォーマンスの比較。**
上: **ミリ秒単位の** GPU **フレーム時間 (値が小さいほうが良い)。**
下: CBR **と** 1080p **のパフォーマンス向上のパーセンテージ (値が大きいほうが良い)。パフォーマンスの向上がわずかな領域はジオメトリー依存であり、**CSO **のオクルージョン確認コストの増加による影響が明らかです。**

今後の動向

今後の実装でいくつかのアイデアを試してみたいと考えていますが、ここではそのうちの 2 **つを紹介します。**

1 **つ目は、動的な解像度のレンダリングと** CBR **を単一の実装に統合することです。前出の** Frostbite* **エンジンと** Sony* PS4 Pro **ゲーム「デウスエクス マンカインド・ディバイデッド」はこのアプローチを採用しており、これら** 2 **つの手法を組み合わせることは有用であると考えられます。**

2 **つ目は、後処理のアンチエイリアシング・ソリューションをチェッカーボード再構築シェーダー内に統合することです。どちらの手法もカラーとデプス・バッファー・サンプリングの組み合わせを使用しており、理論的には同じシェーダー内に統合することで別々にアンチエイリアシング・パスを実行するコストを軽減できます。**

まとめ

このホワイトペーパーで紹介した手法は、CBR と既存のフォワードまたは遅延シェーディング・パイプラインを統合する簡単なソリューションを提示しました。ピクセル・シェーダー・ステージに大きく依存するワークロードでは、CBR によりフレーム時間がフル解像度と比較して最大 30% 軽減されました。比較的容易に実装でき、複数の解像度にわたってスケーリングし、GPU メーカー間で互換性のある CBR は、GPU を最大限に活用したいと考える開発者にとって魅力的なソリューションです。

謝辞

このホワイトペーパーと付属のサンプルコードについて、技術的作業や内容のレビューに貢献してくれた多くの方々に感謝します。Kai Xiao をはじめとするインテル ART チームは、ディスカッションや初期チェッカーボード・ソースコードの共有に協力してくれました。Stephen Junkins は、初期レビューを提供し作業を支援してくれました。Marissa du Bois は、サンプルコードを公開できるようにさまざまな内部処理を行ってくれました。Lumberyard Bistro モデルを提供してくれた Amazon*、Open Research Content Archive (ORCA) をホスティングしている NVIDIA* Corporation、最新のアトリウムスポンザ宮殿モデル (英語) を提供してくれた Crytek にも感謝しています。

参考文献 (英語)

- Amazon Lumberyard Bistro, Open Research Content Archive (ORCA).
- Doug Binks, Dynamic Resolution Rendering Article, July 13, 2011.
- Giliam de Carpentier and Kohei Ishiyama, Decima: Advances in Lighting and AA. SIGGRAPH 2017 Advances in Real-Time Rendering.
- Crytek Sponza Model, Crytek, cryengine3 downloads.
- Jalal El Mansouri, Rendering Rainbow Six Siege. GDC 2016.
- Brian Karis, High-Quality Temporal Supersampling. SIGGRAPH 2014 Advances in Real-Time Rendering in Games
- Richard Leadbetter: Interview with Mark Cerny, PS4 Pro Architect. Inside PlayStation Pro 4 Pro: How Sony made the first 4K games console. October 20, 2016.
- Timothy Lottes, FXAA. February 2009.
- Filip Strugar, Conservative Morphological Anti-Aliasing (CMAA) – March 2014 Update. March 18, 2014.
- Graham Wihlidal 4K Checkerboard in Battlefield 1 and Mass Effect: Andromeda. GDC 2017.

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。