Vol.2018-CG-170 No.4
2018/6/21

GPU グラフィックスパイプラインでの粗片描画

今給黎 隆[†1,a)]

概要：粗片描画 (Coarse pixel shading) は，画素の大きさを部分的に変更することで，陰影計算の負荷が高いシーンでの計算負荷を低減する手法である．解像度が部分的に異なる画像を生成する手法としては，複数の解像度の画像を生成し，最終的に合成する手法が考えられるが，同じ画素を複数回計算する必要があり，重複している分の計算の無駄が生じる．粗片描画は，高速化手法の 1 つであるが，現在の GPU アーキテクチャは出力する画素の大きさを部分的に変更する標準的な機能はなく，リアルタイムアプリケーションに組み込まれることはなかった．GPU を用いて高速に処理する手法も提案されているが，計算パイプラインを使用しており，パイプラインの同期や長時間の GPU スレッドのストールが生じていた．本提案手法では，画素ごとの処理であるフラグメントシェーダで処理するフラグメントをフラグメントシェーダに投入する前に選別することで，無駄な陰影計算を生じないグラフィックスパイプラインで完結した粗片描画を実現する．

Coarse Pixel Shading implementation using graphic pipeline

Takashi Imagire[†1,a)]

1. はじめに

粗片描画 (Coarse pixel shading) は，照明計算するフラグメントの大きさを画素の大きさから変更することで，陰影計算の負荷が高いシーンでの計算負荷を低減する手法である．解像度が部分的に異なる画像を生成する手法としては，複数の解像度の画像を生成し，最終的に合成する手法が考えられるが，同じ画素を複数回計算する必要があり，特に照明計算の負荷が高いシェーディングにおいて重複している箇所の計算が無駄になる．粗片描画処理は，高速化手法の 1 つであるが，現在の GPU アーキテクチャでは，計算するフラグメントの大きさを部分的に変更する標準的な機能はなく，リアルタイムアプリケーションの製品において，組み込まれてこなかった．GPU を用いて高速に粗片描画する手法も提案されているが，計算パイプラインを使用しており，パイプラインの同期や長時間の GPU スレッドのストールが生じ，実践的な速度は得られていない．本提案手法では，画素ごとの処理であるフラグメントシェーダで処理するフラグメントをフラグメントシェーダで実行する以前に選別することで，無駄な陰影計算を生じないグラフィックスパイプラインで完結した粗片描画処理を実現する．

[†1] 現在，東京工芸大学
Presently with Tokyo Polytechnic University
[a)] t.imagire@game.t-kougei.ac.jp

2. 関連研究

GPU はシェーダ群と固定機能で構成されている．固定機能については決められた機能について有効か無効か切り替えるもしくは機能の順番を変更することのみが可能で，功利的に実現できる処理に制約を与える．実現できないことの一つが一つの画像における部分的な解像度の変更である．VR でのレンダリングでは，周辺視野の部分の詳細度は必要ないため，計算を速くする手法が提案されている [1]．また，視野周辺の低解像度化は VR 酔いを起きにくくする [2] という正の効果も存在している．しかしながら，現在の GPU では画素密度が一定の対象しかレンダリングできないため，VR 酔いを起こさないための画像の生成において，複数の解像度の画像をレンダリングし，それらを組み合わせる等の必要があり，レンダリングの仕組みが複雑になると同時に同じフラグメントを何度も計算する無駄

も生じる．

Vaidyanathan ら [3] は，照明計算を行うフラグメントに関して，フラグメントと画素を一対一に対応させるのではなく，複数の画素をまとめて，2 のべき乗の大きなサイズの正方形のフラグメントとして計算することで，レンダリングを高速化する粗片描画 (Coarse Pixel Shading) を提案した．Sathe と Janczak [4] は，粗片描画を遅延レンダリングに適応し，現在の商用ゲームで標準的に用いられているゲームエンジンで実装できる手法として Deffered Coarse Pixel Shading (遅延 CPS) を提案した．しかしながら，Sathe と Janczak による遅延 CPS は，GPU の描画パイプラインの中で閉じておらず，汎用的な GPU 計算を行う計算シェーダで CPS を実装しており，負荷軽減のためのレンダリング手法としての目的には適していない．例えば，遅延 CPS では，32×32 等の数の画素をまとめて 1 つの GPU スレッドで処理を行うが，ある GPU スレッドで 32×32 の画素を一つのフラグメントとして処理を行い，別の GPU スレッドでは全てバラバラのフラグメントとして処理を行うと，1024 倍の処理時間の差が発生し，他の GPU スレッドでは処理を終えたとしても，同じ Wavefront(Warp) の中で長い処理を行う GPU スレッドの終了を待たなければならないという無駄が生じる．また，計算シェーダは，ラスタライズを経由するレンダリングパイプラインと別のコマンドバッファで管理される為，互いのパイプラインの同期処理を避けることができない．

本稿では，テクスチャのミップマップの構造と同じくミップマップ構造を持つ早期カリングの機能を使って，計算シェーダで用いられているコンピュートパイプラインを用いない，レンダリングパイプラインのみで CPS を実現する手法を提案する．

3. 提案法の概要

本手法は，遅延シェーディング [5] でのレンダリングを拡張する．図 1 が，提案手法の全体的な流れである．通常の遅延シェーディングでは，G バッファを生成したのちに，それら G バッファの情報を用いて照明計算を行うが，本手法では，G バッファ生成後に，詳細度に応じた階層型のステンシルバッファ (ステンシルピラミッド) を構築し，そのステンシルピラミッドを用いて早期ステンシルテストを行うことで，ミップマップ構造のレンダーターゲットに対して必要なフラグメントだけのレンダリングを実現する．最終的に，ミップマップテクスチャとしてレンダリングされた結果を合成することで，1 枚の画像を生成する．

3.1 G バッファ生成

最初に，G バッファへ照明計算のための情報を出力する．通常，G バッファには，アルベド，鏡面反射強度，ラフネス，法線マップ等が記録されるが，提案法では，G バッファに粗片描画の詳細度 (LOD バッファ) を追加する．LOD バッファには，各画素を計算するフラグメントのレベル l を記録する．レベル l のフラグメントは，一辺が 2^l の画素数の正方形とする．例えば，画面の中心が最も細微で，画面の端に行くほど粗い詳細度となる LOD バッファは，図 2 となる．本稿の検証では，照明モデルとしてフォンの鏡面反射モデルを用いるが，LOD バッファを追加することにより，他の照明モデルにも本手法は適応可能である．

3.2 ステンシルピラミッド構築

次に，早期カリングを行うために階層型のステンシルバッファを構築する．詳細度の最大数の数だけステンシルバッファを用意する．それぞれのステンシフバッファの大きさは，詳細度の 2 のべき乗で小さくする．それぞれのステンシルバッファには，LOD バッファを複製する．今回は，OpenGL の GL_ARB_shader_stencil_export 拡張を用い，ステンシルバッファに LOD バッファを直接書き込んだ．

提案法の実装として，ステンシルバッファではなく，深度バッファを使う手法も考えられるが，浮動小数点数の一致の比較や深度バッファの処理内での浮動小数点数と固定小数の変換が必要となるため，一致するフラグメントを抽出することは難しいため，本手法ではステンシルバッファを早期カリングに用いた．

3.3 照明計算

その後，G バッファを使った照明計算を行う．ステンシルバッファと同様に詳細度の最大レベルまでの 2 のべき乗の大きさを持つミップマップ化されたレンダーターゲットを用意し，各ミップマップレベルに関して早期ステンシルテストを行いながら照明計算を行う．ステンシルテストでは，描画しようとするレンダーターゲットの詳細度とステンシルバッファの値を比較し，値が等しいフラグメントのみ描画を行う．

3.4 ミップレベル合成

最後に，描画結果のミップマップテクスチャを各レベルごとに集めて最終的な画像を生成する．各画素に関して，LOD バッファの値のミップマップレベルにおけるテクスチャを読み込むことと，通常の CPS の画像が生成される (図 3)．

この通常の CPS の結果は，ブロック型のアーティファクトが目立つため，補間によってアーティファクトを軽減する．合成対する先の画素を $\vec{P}$ とする．$\vec{P}$ におけるフラグメントの詳細度を l とする．この時，フラグメントの大きさは，$S(\vec{P}) = 2^l$ となる．フラグメントの中心を $\vec{O}$ として，中心 $\vec{O}(\vec{P})$ から画素 $\vec{P}$ への規格化されたベクトル $\vec{q}$ を

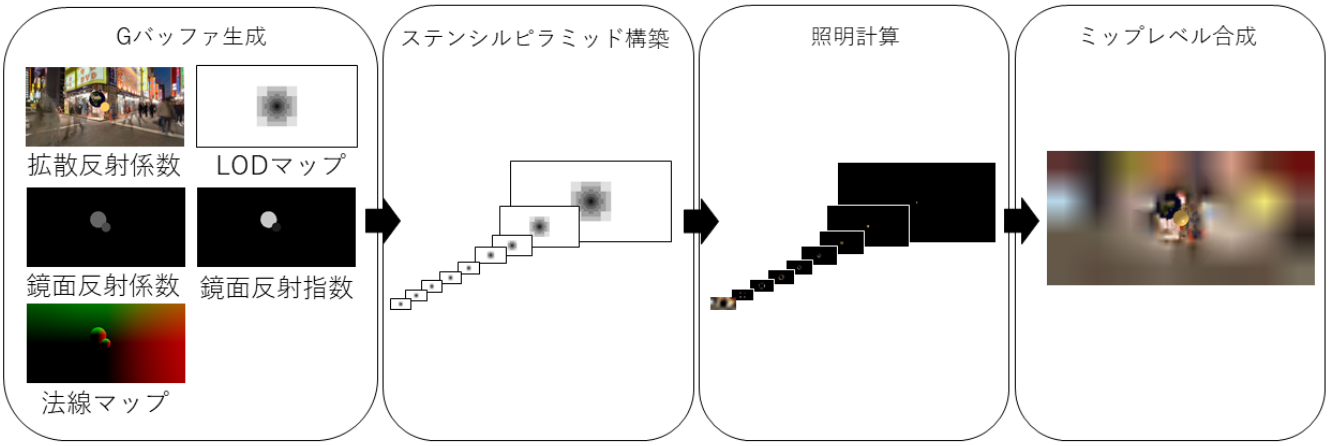

図 1　提案手法のレンダリングプロセス．

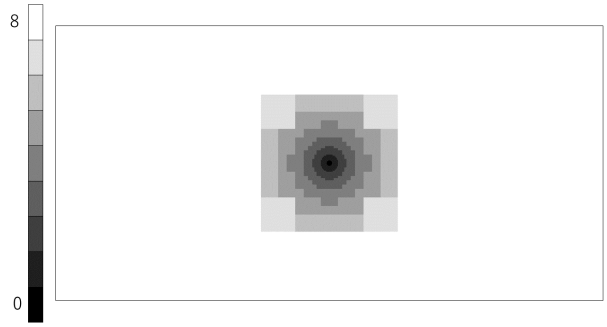

図 2　LOD バッファ．色の濃さは，フラグメントのレベルに対応する．

図 3　最近接近傍法で各フラグメントのレベルのテクスチャをサンプリングして合成をした場合の結果

$$\vec{q} = \frac{\vec{P} - \vec{O}(\vec{P})}{S(\vec{P})}, \tag{1}$$

により定義を行う．画素 $\vec{P}$ の横方向と縦方向に隣接するフラグメント $\vec{P}^x, \vec{P}^y$ は，次の式で求められる．

$$\vec{P}^x = \begin{cases} (O(\vec{P})_x - (\frac{1}{2}S(\vec{P}) + 1), P_y) \ (q_x < 0) \\ (O(\vec{P})_x + (\frac{1}{2}S(\vec{P}) + 1), P_y) \ (0 <= q_x) \end{cases}, \tag{2}$$

$$\vec{P}^y = \begin{cases} (P_x, O(\vec{P})_y - (\frac{1}{2}S(\vec{P}) + 1)) \ (q_y < 0) \\ (P_x, O(\vec{P})_y + (\frac{1}{2}S(\vec{P}) + 1)) \ (0 <= q_y) \end{cases}, \tag{3}$$

ここで，下付きの添え字は，ベクトルの x 方向及び y 方向の成分である．バイリニアフィルタリングと同様に，4 つのプラグメントの線形和により最終的な色を決定する．もう一つのプラグメントは，$\vec{P}^x, \vec{P}^y$ の内，$\vec{P}$ に近いフラグメントに関して直交する向きでの近傍を採用する

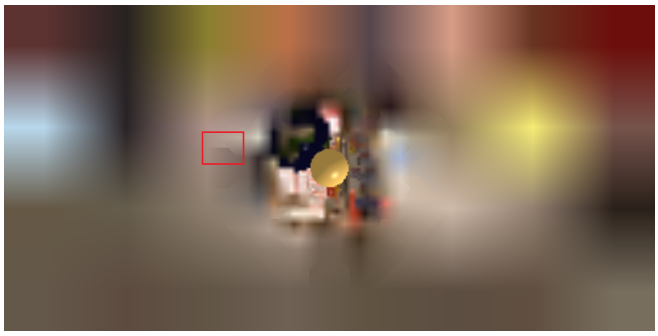

図 4　提案法による結果画像．赤枠内に不連続な色の変化が見られる．

$$\vec{P}^2 = \begin{cases} (P^x_x, O(\vec{P}^x)_y - (\frac{1}{2}S(\vec{P}^x) + 1)) \ (|q_y| < |q_x|) \\ (O(\vec{P}^y)_x - (\frac{1}{2}S(\vec{P}^y) + 1), P^y_y) \ (|q_x| < |q_y|) \end{cases}, \tag{4}$$

最終的な色 C は，画素 $\vec{P}$ におけるフラグメントの色を $C(\vec{P})$ として，次の式で求められる．

$$\vec{C} = \begin{cases} lerp(lerp(C(\vec{P}), C(\vec{P}^y), |q_y|), \\ \quad lerp(C(\vec{P}^x), C(\vec{P}^2), |q^x_y|), |q_x|) \\ \quad (|q_y| < |q_x|) \\ lerp(lerp(C(\vec{P}), C(\vec{P}^x), |q_x|), \\ \quad lerp(C(\vec{P}^y), C(\vec{P}^2), |q^y_y|), |q_y|) \\ \quad (|q_x| < |q_y|) \end{cases}, \tag{5}$$

ここで，$lerp$ は，線形補間 $lerp(A, B, t) = (1 - t)A + tB$ であり，$\vec{q}^x, \vec{q}^y$ は，画素 $\vec{P}^x, \vec{P}^y$ におけるフラグメントのそれぞれの中心からの正規化された差分である．

4. 結果

本手法の検証には，外付け GPU として Radeon RX560 を搭載したノート PC (Razer Blade Stealth) を使用した．3D API には OpenGL，フレームワークに GLFW 3.2.1 を採用

した．提案手法において，GL_ARB_shader_stencil_export 拡張を採用した．NVIDIA 製の GPU 等，本拡張をサポートされていない環境では，アプリケーションは動作しない．

図 4 が，提案法の結果である．1×1 から 256×256 の 9 段階の詳細度で祖片描画を行った．光源の数を 200 個に設定して照明計算を行った場合，1 画素を 1 フラグメントとする通常の描画では，36.4 FPS で実行され，提案法の結果は 60.1FPS となり，65%の実行速度の向上が得られた．また，テクスチャを最近接近傍法でサンプリングした画像合成の処理では，実行速度は 68.6FPS となった．

5. まとめと今後の課題

提案手法では，各画像の詳細度を階層型のステンシルバッファに格納し，早期ステンシルテストを用いてミップマップテクスチャにレンダリングを行うことで，高速な粗片描画を実現した．本手法は，大きな GPU 上の効率の損失がない初めての祖片描画のアルゴリズムである．粗片描画は，Intel による GL_INTEL_MULTI_RATE_FRAGMENT_ SHADER 拡張が提案されているが，詳細は公開されておらず，手法が確立しているとは言えない．本手法は，遅延レンダリングの手法に対して，LOD バッファの追加とステンシルバッファを構築・利用する処理をさしはさむだけで CPS を実現でき，実用的な手法と言える．

本手法では，最近接近傍のサンプリングではなく，近傍 4 点の色の線形合成を行って最終的な色を導出したが，詳細度が変化する近傍で不連続性が観測される．具体的には，図 4 の赤枠内において不連続な色の変化に基づく横方向の筋が観測される．この不具合は詳細度が下がるほど目につく為，詳細度の上限を抑制するか，サンプリング手法を改善する必要で性がある．

今回の事例としては，２次元の辺面画像の生成にアルゴリズムを適応したが，VR 酔い対策としての CSP の利用 [6] に適用することで，ユーザーの体験を向上する成果を実現したい．

謝辞　本研究は H30 年度東京工芸大学ブランディング研究テーマの助成を受けたものである．

参考文献

[1] Guenter, B. and Finch, M. and Drucker, S. and Tan, D. and Snyder, J.: *Foveated 3D Graphics*, ACM Trans. Graph. 31(6), pp.164:1–164:10 (2012).
[2] Chiba, M. and Nakamura, Y. and Watanabe, D. and Mikami, K.: *Reduction of VR Sickness and Improvement of Immersive Considering Vection by CHIBA Mask*, Digra-J Annual Conference, pp.11–14, Digital Games Research Association JAPAN (2016).
[3] Vaidyanathan, K. and Salvi, M. and Toth, R. and Foley, T. and Akenine-Möller, T. and Nilsson, J. and Munkberg, J. and Hasselgren, J. and Sugihara, M. and Clarberg, P. and Janczak, T. and Lefohn, A.: *Coarse Pixel Shading*, Proceedings of High Performance Graphics 10 pp.9–18 (2014).
[4] Sathe, R. P. and Janczak, T.: *Deferred Coarse Pixel Shading*, GPU Pro 7, CRC Press, pp.145–153 (2016).
[5] Valient, M.: *Deferred rendering in killzone 2*, in Develop Conference, `http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf` (2007).
[6] Tachibana, K. and Imagire, T.: *A Fast Rendering Technique for VR Sickness Prevention Using Coarse Pixel Shading*, in I3D 2018 Posters (2018).