社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－ARC－164（25）
2005／8／4

並列ボリュームレンダリングにおける投機的描画に関する考察

篠本雄基[†]　三輪忍[†]　嶋田創[†]
森眞一郎[†]　中島康彦[†]　富田眞治[†]

本稿では，汎用 GPU を用いた並列ボリュームレンダリングにおける問題点を指摘する．GPU によるレンダリングと CPU による画像合成は互いに独立した処理であるため，パイプライン処理が可能であるが，観察視点の指定間隔がレンダリング時間よりも長い場合，パイプラインに空きが生じる．このパイプラインの空きを利用した投機的描画について考察する．

Consideration for Speculative Rendering in PVR

Yuki Shinomoto ,[†] Shinobu Miwa ,[†] Hajime Shimada,[†]
Shin-ichiro Mori ,[†] Yasuhiko Nakashima[†] and Shinji Tomita [†]

In this paper, we point out a problem in a parallel volume rendering with commodity GPUs.Since rendering with GPU and image composition with CPU are processed independently, we can overlap these jobs.However, we cannot overlap them when a viewpoint is set intermittently.To utilize both GPU and CPU, we consider a spaculative rendering.

1. はじめに

近年の計算機処理能力の向上による大規模シミュレーションシステムの実用化に伴ない, より大規模な 3 次元データの解析を支援する可視化システムの実用化が求められている．このような大規模な 3 次元データの解析を支援する可視化方法の一つとしてボリュームレンダリングが挙げられる．ボリュームレンダリングは, 複雑な 3 次元構造の理解が容易であり, 工学, 医学などの分野で幅広く利用されている．

近年，ボリュームレンダリングにおいて専用ハードウェアや汎用 GPU(Graphics Processing Unit) の採用によるレンダリング時間の大幅な短縮が報告されている．しかし，大規模データの実時間可視化を実現するにあたり，演算性能ならびにメモリ容量の不足を補うための並列化が必要である．このような並列可視化環境においては，各ノードでのレンダリング処理と各ノードで生成された中間画像の合成処理の両方の高速化が必須である．また，レンダリング処理と中間画像の合成処理は独立した処理であり，両処理をパイプライン化することで，さらなる高速化を図ることができる．

今，ボリュームデータを観察する際に，ユーザがマウス等を連続的に動かして視点をインタラクティブに変化させる場合を考える．

位置情報がマウス等から与えられる間隔がレンダリング時間より長い場合には，パイプラインに空きが生じる．そこで我々は，このパイプラインの空き時間を利用して，視点の移動先を予測してレンダリングを行う投機的描画を提案する．投機的描画の予測が的中した場合は，パイプライン空きが生じることがなくなり，スムーズな描画が実現できると考えられる．

本稿では，汎用 GPU を用いた並列ボリュームレンダリングにおけるパイプライン処理と投機的描画について述べる．以下，2 章で研究の背景となる汎用 GPU を用いたボリュームレンダリング手法について説明する．3 章では提案する投機的描画について述べ，4 章でまとめを述べる．

2. 汎用 GPU を用いた並列 VR

我々が対象としている汎用 GPU を用いた並列ボリュームレンダリングについて説明する．

2.1 Volume Rendering

ボリュームレンダリングとは，3 次元のスカラー場をボクセルの集合として表現し, 2 次元平面へ投影することにより, 複雑な内部構造や動的特性を可視化する手法である．

ボリュームレンダリングでは，対象空間内のボクセルすべての寄与を計算して 2 次元平面へ投影する．このため表示像が正確であり, はっきりとした境界を持た

† 京都大学
Kyoto University

ない雲や炎といった自然現象やエネルギー場の可視化に適用できるという特徴をもつ. このようにボリュームレンダリングを用いると, 複雑な 3 次元構造の理解が容易となるため, 工学, 医学などの分野で幅広く利用されている.

しかし, 膨大な計算時間と記憶容量が必要とされ, 大型計算機や特殊ハードウェアを用いる場合に利用が限られており, リアルタイムに可視化することは一般に困難である.

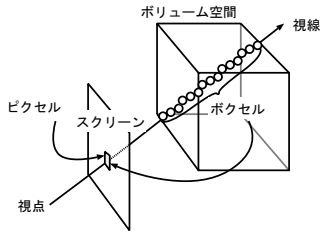

図 1 ボリュームレンダリング

ボリュームレンダリングは描画面の各画素から視線方向に沿ってボクセル値の持つ色情報を積分していく. これを離散化すると, スクリーン上の各ピクセルごとに発生する視線に沿って, 視線と交差するボクセル値のサンプリングを視線上のボクセルがなくなるまで繰り返し, ピクセル値を求めることになる (図 1). この方法は視点から近い順にサンプリングする方法 (front to back) と, 視点から遠い順にサンプリングする方法 (back to front) に分けられる. back to front の場合, ボクセルの値を視点に近い順から, v_0, v_1, $\cdots$, v_n とし, RGB の各色情報 c_k と不透明度 α_k がボクセル値 v_k の関数 (伝達関数) で表されるとすると, ピクセル値 C は

$$C = \sum_{i=0}^{n} \alpha(v_i) c(v_i) \prod_{j=0}^{i-1} (1 - \alpha(v_j)) \qquad (1)$$

と表される. このピクセル値計算式は累積値 C_k を用いて次式のような漸化式に変形される.

$$C_{k-1} = \alpha(v_{k-1}) c(v_{k-1}) + (1 - \alpha(v_{k-1})) C_k \qquad (2)$$

ここで $C = C_0$ である. 式 (2) はボリュームのサンプリングを視線方向に従って一定のサンプリングで行い, 描画面に遠い方から順に RGB 値を α ブレンディングすることでボリュームレンダリングができることを示している. α ブレンディングとは二枚の画像の持つ RGB 値を, α 値の示す比率で線形内挿して, 合成画像の RGB 値を算出する方法である.

2.2 Texture Based Volume Rendering

汎用 GPU によるボリュームレンダリングでは, テクスチャベースのアルゴリズムが用いられる. ボリュームはある軸に対して垂直なスライスの重ね合わせで表現される. そのスライスをテクスチャとしてポリゴンにマッピングし, それらを視点から遠い順に順次 α ブレンディングすることでボリュームレンダリングを行う. この手法を用いることで, 汎用 GPU の機能を利用した高速処理が可能である.

GPU が 3 次元テクスチャをサポートしている場合には, 視線に対して垂直な面を用意し, ボリュームデータを 3 次元テクスチャとして扱う方法が可能である (図 2).

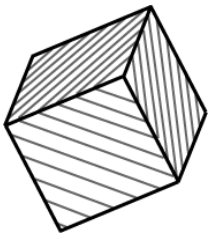

図 2 3 次元テクスチャ

テクスチャのマッピングは, 伝達関数を用いてボクセル値を変換した RGBA の色情報を持つ 3 次元テクスチャを 1 回サンプリングすることで行う. 3 次元テクスチャのサイズは, 元のボリュームデータの 4 倍となるため, ビデオメモリの消費量が大きくなるという欠点がある.

この欠点を解消するために, 近年の汎用 GPU がサポートしている Dependent Texture(依存テクスチャ) を利用する方式がある. 依存テクスチャとは, あるテクスチャをサンプリングした値をテクスチャ座標として用い, 別のテクスチャをサンプリングする機能である.

ボクセル値そのものを保持する 3 次元テクスチャと, ボクセル値と RGBA の関係を表す伝達関数表を保持する 1 次元テクスチャを用意する. この 1 次元テクスチャを, 伝達関数のルックアップテーブルとして用いる. 3 次元テクスチャからサンプリングしたボクセル値をテクスチャ座標とし, ルックアップテーブルの 1 次元テクスチャをサンプリングすることで RGBA の色情報を得ることができる (図 3).

RGBA の色情報を持つ 3 次元テクスチャを用いる場合と比較すると, ビデオメモリの消費量およびメモリ帯域の消費量を節約できる利点がある. また, ルックアップテーブルのみを変更することで, 少ないオーバヘッドで動的に伝達関数を変更することができる.

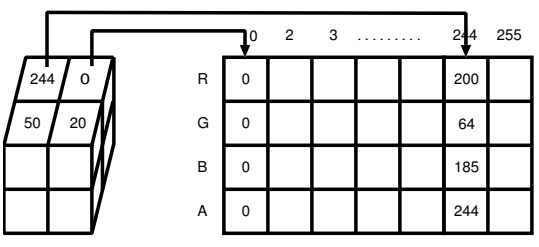

図 3 依存テクスチャ

ボクセル値を 8bit の RGBA データとして描画した場合, 現在の汎用 GPU は，$256 \times 256 \times 256$ のサイズのボリュームデータをリアルタイムに描画することができる．ここでのリアルタイムとは秒間 30 回以上，すなわち $0.03sec$ 以下の時間で描画できることを意味する．

しかし, さらに大きいサイズのボリュームデータを描画する場合, 演算速度, メモリバンド幅の制限がボトルネックとなり, 描画速度が低下する．最新の GPU である GeForce7800GTX の場合で，Core Clock: 420MHz, Pipeline Unit: 24, Memory Pipeline: 600MHz GDDR3, Memory: 256bit 256MB という仕様である．

ビデオメモリが 256MB の場合，格納できるボリュームデータは，ボクセル値を 8bit の RGBA データとして描画した場合 $256 \times 256 \times 512$ のサイズが限界である．Dependent Texture を用いた場合は，$512 \times 512 \times 512$ のボリュームデータを格納することができる．

近年の汎用 GPU には大容量のビデオメモリが搭載される傾向にあり，512MB のビデオメモリを搭載したモデルも市販されている．

しかし，GPU の処理能力 (主にメモリバンド幅) が追い付いておらず，$512 \times 512 \times 512$ サイズのボリュームデータをリアルタイムに描画することは，最新の GPU をもってしても不可能である．このように, 現状の汎用 GPU の性能は大規模なボリュームデータの描画を行うにはまだまだ性能が不足しており, 複数の汎用 GPU を用いた並列化を行う必要がある．

2.3 並 列 化

汎用 GPU のテクスチャマップ機能と α ブレンディングを用いることにより, ボリュームレンダリングの高速処理が可能になる．しかし, テクスチャを保持するビデオメモリの容量には制限があるため, ビデオメモリの容量を上回るサイズのデータを描画することはできない．描画するデータが汎用 CPU のビデオメモリの容量を超えない大きさのテクスチャに分割して描画させることにより, そのままでは描画できない大きなサイズのデータを描画することが可能となる．しかし, 保持しているテクスチャデータのサイズがビデオメモリの容量を超える場合, テクスチャが描画に使われる度にメインメモリから転送しなくてはならない．このため, テクスチャデータの転送時間がボトルネックとなり描画速度が急激に低下する．そこで複数の汎用 GPU を用いて並列化を行う．これによりデータサイズの大きなボリュームデータを扱うことができる．

まず, $N_x \times N_y \times N_z$ のボリュームデータを x, y, z 方向に d 等分に分割し, P 台のノードにそれぞれを割り当てる．各サブボリュームをそれぞれの GPU に与えてボリュームレンダリングを行い, 中間画像を生成する．このようにして生成された d^3 枚の中間画像を, 視点からの距離の遠いもの，または距離の近いものから α ブレンドして一つの画像にまとめることにより, 最終結果を得る[2)]．

図 4 にシステム構成の例を示す．図の構成では，Master ノードからの視線情報を基に，4 台の Slave ノードで構成する PC クラスタで中間画像を生成し，その結果を Master ノードに集めた後に画像合成を行う．あるいは，Slave ノード間で通信し，中間画像を並列合成後，その結果を Master ノードが受信する．

リモートホストを用意することによる利便性を考慮しなければ，Master ノードを用意せず，全てのノードが描画と画像合成を行うことで，最終画像の転送を省略することができる．

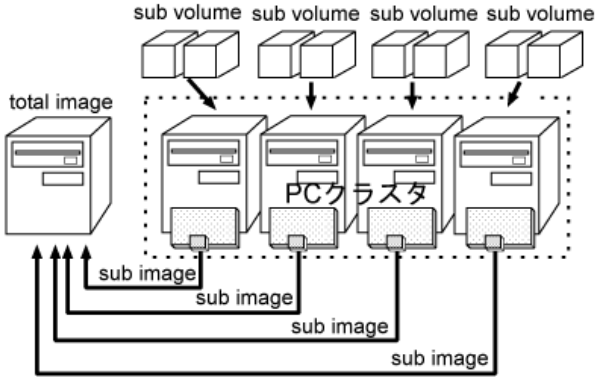

図 4 並列化

PC クラスタ上でのソフトウェアベースの画像合成アルゴリズムとして，Binary-Swap-Composition(以下 BSC)[1)] や SLIC[3)] がある．

BSC は分割統治型の並列画像合成アルゴリズムで, 通信量と演算量を抑え，高い並列性を抽出可能なアルゴリズムである．

N 台のプロセッサで，最終合成を行う場合を考える．BSC は，交換 (swap)，合成 (composite)，収集 (gather) という三つのフェイズから構成される (図 5)．

swap

(1) 各プロセッサは，合成すべきスクリーンを二分割する

(2) 自身と同じ領域を持つプロセッサと，互いが異なる領域を担当するようように，分割したスクリーンを交換する

この交換を，$\log_2 N$ 段繰り返す．

composite

$\log_2 N$ 段の交換後には，各プロセッサは自身が担当するスクリーン領域の，すべてのプロセッサによるサブスクリーンを持つことになる．すなわち，最終合成を行うことが可能な条件を満たしている．したがって，各プロセッサが自身の担当領域を最終合成することにより，並列最終合成を行

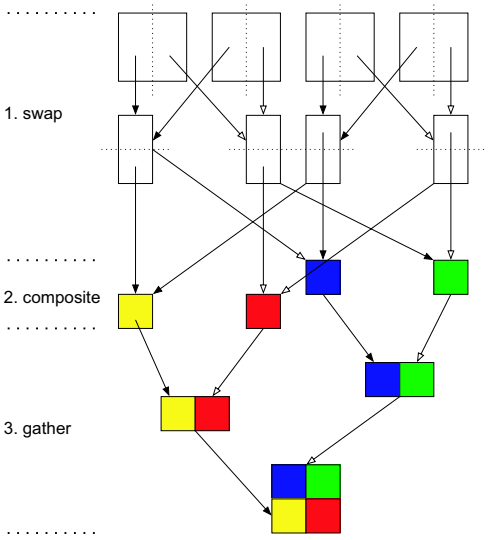

図 5 Binary Swap Composition Tree

うことができる．

gather

並列最終合成が終わった後では，単純に領域をつなぎ合わせるだけで最終画像を得ることができる．このつなぎ合わせは，N 個の最終合成画像を通常の木による合成を行うことで，1 つのノードに集めることで完了する．

SLIC は，各ノードが生成した中間画像同士の重複関係を視線方向に基づいて解析し，合成の必要がない背景領域や他の中間画像と重なりのない領域を並列画像合成の対象から省くことで合成処理の演算量を削減するアルゴリズムである．

SLIC は，各ノードが生成した中間画像同士の重複関係を視線方向に基づいて解析し，合成の必要がない背景領域や他の中間画像と重なりのない領域を並列画像合成の対象から省くことで，合成処理の演算量を削減する．

合成処理の対象から省かれた領域に対応する中間画像は，必要に応じて最終画像表示ノードへ直接転送を行う．負荷の均等化に際しては，各ノードにおいて中間画像の重複回数と重複状態を基め，その重複状態が同じ範囲 (スパン) を負荷分散の単位として，スパン単位の合成処理を各ノードに静的に割り当てる負荷分散方式を採用している．

文献[3] によると，画像合成にかかる時間は，画像合成アルゴリズムに SLIC を用い，1GHz の CPU を搭載した PC8 台を 100BaseTX Ethernet で接続したシステムにより 512×512 のサイズの画像を合成した場合で約 $0.1sec$ と報告されている．

ここで，合成時間の大半は中間画像の転送時間であるため，近年安価に入手できるようになった 1000BaseT Ethernet を用いれば，合成時間はより短くなると考えられる．

2.4 パイプライン処理

並列ボリュームレンダリング処理のうち，GPU によるボリュームレンダリングと，CPU による中間画像の合成は，それぞれ独立した処理である．そのため，CPU が画像合成を開始すると同時に，GPU が次に観察する視点における描画を開始することで，GPU と CPU のパイプライン処理が可能である．

GPU は CPU が描画命令を発行することで動作する．CPU が GPU に描画命令を発行している間は，CPU は中間画像の合成処理を行うことができない．この問題を解決するために，1 つの CPU に対して，2 個のプロセス，プロセス 1 とプロセス 2 を生成する．プロセス 1 は GPU との通信のみを行い，GPU のビデオメモリから中間画像をメインメモリにコピーする．そしてプロセス 2 は GPU とは無関係に中間画像の合成処理のみを行うことで，GPU と GPU の効率的な動作が可能であると考えられる．プロセス 1 がメインメモリにコピーした中間画像は，プロセス 2 と通信を行うことで受け渡しを行う．

また，GPU のビデオメモリに描画命令を記憶させておく機能であるディスプレイリストを利用すれば，CPU はディスプレイリストを呼び出す 1 命令を送信するだけで GPU は複数の命令を実行し，その間 CPU は GPU と並列に描画命令の発行以外の計算を行えるようになる．中間画像をメインメモリにコピーする際は GPU と CPU の同期を取る必要があり，グラフィクス API が持つ同期命令を利用する．

GPU で描画を行い，中間画像をメインメモリにコピーされるのにかかる時間を T_{rend}，CPU が中間画像の合成を行うのにかかる時間を T_{comp} とする．パイプライン処理を行わない場合，全体の描画時間 T は $T = T_{rend} + T_{comp}$ となる．パイプライン処理を行った場合，$T_{rend} > T_{comp}$ の時 $T = T_{rend}$ となり，$T_{rend} < T_{comp}$ の時 $T = T_{comp}$ となる．CPU と GPU の双方が常に処理を行うためには，$T_{rend} = T_{comp}$ である必要がある．

並列ボリュームレンダリングにおいてパイプライン処理を行うと，パイプライン処理を行わない場合と比較して，同じサイズのボリュームデータをより少ないノード数でリアルタイムに描画できるようになる．

T_{comp} が大きく，パイプライン処理全体のボトルネックとなっている場合，画像合成処理を分割してパイプライン化することが考えられる．

各ノードが画像合成処理の前半を担当し，後半を別途用意した画像合成ノードが担当することで，パイプライン処理時の T_{comp} を削減することができる．

2.5 並列化時の問題点

観察する視点が与えられるまで描画は開始できないため，パイプライン処理を行うためには，現在指定された視点において GPU が描画を終了した時点で，次

に観察する視点が与えられていなければならない．

一定方向に視点を動かしつづけるなど，予め視点の軌跡が指定してある場合や，視点を固定して時間変化するボリュームデータを描画する場合などはこの条件を満たし，パイプライン処理が可能である．

ユーザがマウスを動かすことで，視点が連続して変化する状況を考える．位置情報が与えられる間隔がレンダリング時間より長い場合では．次の視点における描画を現在の視点における画像合成と並列に行うことができないため，パイプラインに空きが生じてしまう．(図 6)．

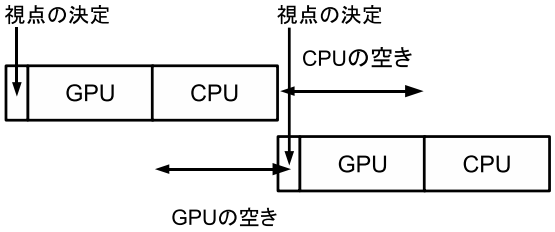

図 6 パイプラインの空き

そこで，次の視点が定まっていない状況で，パイプラインに生じた空き時間を利用して次に指定される視点を予測し，投機的描画を行うことを考える．

3. 投機的描画

投機的描画を行うために，視点情報の送信ならびに予測と，最終画像の受信を行うマスタノードを用意する．または，描画ノードのうち 1 台に，中間画像の合成に加えて視点情報の送信ならびに予測を行わせる．

3.1 基 本 方 針

ユーザがマウス等を用いて観察する視点を指定したら，マスタノードは指定された視点の情報と，予測した次の視点の情報を各ノードに送信する．各ノードの GPU は，指定された視点におけるボリュームレンダリングを行い，CPU は GPU のビデオメモリ内にある中間画像をメインメモリにコピーする．

次に CPU がノード間で通信を行い中間画像の合成を開始すると同時に，GPU は予測した次の視点における描画を開始する．CPU による中間画像の合成が完了し，最終画像が提示された時点で，GPU による予測視点における描画が完了する (図 7)．

続いてユーザが次に観察する視点を指定し，先程と同様に，指定された視点の情報と，予測した次の視点の情報を各ノードに送信する．指定された視点が先程の予測視点と一致した場合は，各ノードの CPU は既に GPU が生成している中間画像を合成し，GPU は次の予測視点におけるボリュームレンダリングを開始する．指定された視点と予測視点が異なっていた場合は，GPU は指定された視点においてボリュームレンダリングをやり直す．

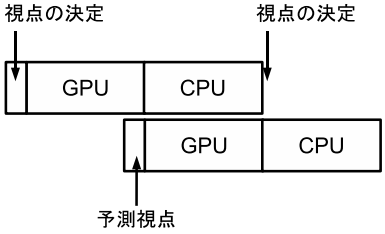

図 7 投機的描画

3.2 詳細度制御と組み合わせる場合

秒間 30 枚のレンダリング速度が得にくい状態において，視点移動中に画像の解像度を下げることで，実時間応答性を保証するシステムが多く存在する．

例えば，スクリーンの面積を $\frac{1}{4}$ にし，最終画像の完成後，画像を 2 倍に拡大する．スクリーンの面積が $\frac{1}{4}$ なることは，視線ベクトルが $\frac{1}{4}$ になることを意味し，T_{rend} は $\frac{1}{4}$ に減少する．画像合成の演算量，通信量は共に $\frac{1}{4}$ になり，T_{comp} は $\frac{1}{4}$ に減少する．すなわち，パイプライン処理を行った場合よりも短い時間で最終画像を提示できる．また，ボリュームレンダリング特有の話として，ボリュームデータのサンプリング間隔を広げると，T_{rend} をさらに減少させることができる．

この詳細度制御を，投機的描画と組み合わせることができれば，予測が外れた場合にも詳細度を落とした画像を提示することで実時間応答性を確保できると考えられる．

低解像度での描画時間を T_L，実時間応答性の確保に求められる描画時間を T_R とする．視点が指定されると低解像度での描画を開始し，CPU による画像合成が始まった時点で，予測視点において本来の解像度での描画を開始する．次の視点が指定され，予測が的中した場合は，そのまま予測視点における描画を続け，外れた場合は低解像度での描画に切替える．

T_L が T_R よりも大幅に小さくなった場合，本来の解像度での描画に時間をかけることができ，次の視点において予測が的中した際に実時間応答性を損なうことなく詳細な画像を提示できる．

3.3 視点が断続的に動く場合

ユーザが，常に視点を移動し続けるのではなく，視点を移動した後に一定時間視点を固定し，その後また視点を移動する場合を想定する．

視点が固定されている間に，予測視点を複数用意し，それぞれの視点における画像を描画することが考えられる．ある予測視点における描画が完了したら，最終画像をメインメモリに保存しておき，次の予測視点における描画を開始する．最終画像の保存に使用するメインメモリの容量を定めておき，容量が許す限り最終画像を保存しておく．視点が固定されている間，この処理を繰り返す．

視点の移動が再開され，予測視点のうちいずれかが次に指定された視点に的中すれば，即座に最終画像を提示することができる．

予測が的中した時点で，すぐに次の視点を予測してパイプライン処理を開始する．

3.4 予測ミスによる描画時間の増加

予測が外れた場合に，描画をやり直すのに掛かる時間を T_{loss} とする．

描画処理および画像合成処理を中止するオーバヘッドがほとんどないのであれば，2.4 章における $T_{rend} = T_{comp}$，$T_{rend} < T_{comp}$，$T_{rend} > T_{comp}$ いずれの場合も $T_{loss} = 0$ である．次の視点における描画時間の全体 T は，パイプライン処理をせずに描画した場合と同じであり，よって予測が外れたことによるペナルティは生じない．

$T_{rend} > T_{comp}$ で，GPU の描画命令をディスプレイリストに全て保存しているなど，GPU が描画処理を途中で中断できない場合を考える．GPU が予測視点における描画を行っている最中に視点が指定され，予測が間違っていた場合は，$T_{loss} \neq 0$ となり，ペナルティが生じる．

T_{rend} と T_{comp} の大小関係は視点によって変化する．そのため，$T_{rend} = T_{comp}$，$T_{rend} < T_{comp}$，$T_{rend} > T_{comp}$ のそれぞれの場合について投機的描画のモデルを用意し，状況に応じて切替える必要があると考えられる．

また，予測が外れた場合に，予測視点における画像を破棄せずに利用できれば，予測が外れた場合も高速化が図れる．

予測が外れても，予測視点と実際に指定された視点とのずれが一定の範囲内であり，ユーザが実時間応答性を優先するのであれば，予測視点における画像をそのまま提示してもよい場合があると考えられる．

また，予測視点における画像と，1 つ前の視点における画像を用いて補間を行い，実際に指定された視点における画像を擬似的に作成することも考えられる．

4. ま と め

本稿では，パイプライン処理を行った場合における従来の汎用 GPU を用いた並列ボリュームレンダリングの問題点を指摘した．解決策としてパイプラインの空きを利用した投機的描画を提案し，考察を行った．

今後は，提案した投機的描画を PC クラスタ上に実装し，評価を行う予定である．

謝　　辞

日頃より御討論いただく京都大学大学院情報学研究科富田研究室の諸氏に感謝します．

本研究の一部は，日本学術振興会科学研究費補助金基盤研究 S (課題番号 14213201)，ならびに，文部科学省特定領域研究 S (課題番号 13224050) による．

参 考 文 献

1) Ma, K.-L., Painter, J. S., Hansen, C. D. and Krogh, M. F. Parallel Volume Rendering Using Binary-Swap Image Composition, *IEEE Comput. Graph. and Appl.* (July 1994), 59–67.
2) Muraki, S., Ogata, M., Ma, K.-L., Koshizuka, K., Kajihara, K., Liu, X., Nagano, Y. and Shimokawa, K. Next-Generation Visual Supercomputing using PC Clusters with VolumeGraphics Hardware Devices, Proceedings of IEEE/ACM Supercomputer Conference (2001).
3) Stompel, A., Ma, K.-L., Lum, E. B., Ahrens, J. and Patchett, J. SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering, *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Oct. 2003).