PC グラフィクスハードウェアを利用した 高精度・高速ボリュームレンダリング手法

山崎 俊太郎 東京大学
加瀬 究 理化学研究所
池内 克史 東京大学

概要

本論文は，PC グラフィクスカードの高速なテクスチャーマップ機能を利用してボリュームレンダリングをインタラクティブな速度で行う手法を提案する．ボリュームレンダリングの計算コストは高いが，ボリュームをスライスの積み重ねで表現し，ハードウェアの加速を利用したテクスチャマップ機能と α ブレンディング機能を使うことで，高速描画が可能である．ただしこの方法ではデータ補間の不足により深刻な画質の劣化が起こるため，multi-texture 機能を使ってサンプル点を増やすことにより画質を向上する．またグラフィクスハードウェアのメモリ容量の制限から扱えるデータサイズが小さくなるので，2^{3N} ブロック化を使った単一階層の適応的サンプリングを行い，画質の劣化を最小限に抑えてインタラクティブな速度で描画を行う．

キーワード: ボリュームレンダリング，インタラクティブ，マルチテクスチャ，2^{3N} ブロック，八分木

High-Quality Interactive Volume Rendering on Standard PC Graphics Hardware

Shuntaro Yamazaki University of Tokyo
Kiwamu Kase RIKEN
Katsushi Ikeuchi University of Tokyo

Abstract

We propose an interactive volume rendering method using flexible texture mapping capability of standard PC graphics hardware. Although volume rendering is computationally expensive, interactive visualization can be achieved by slice-based method in which volume data set is represented as a stack of polygonal slices and rendered by texture mapping and α-blending function accelerated by graphics hardware. One of the drawbacks in sliced-based method is that insufficient interpolation leads to severe visual artifacts, which can be successfully eliminated by inserting intermediate slices which are effectively generated using multi-texture function. Another problem is the limitation of data size due to the limitation of the amount of video memory. We introduce adaptive 2^{3N}-blocking to enable both drastic decimation and interactive processing of volume data, and show high-quality interactive volume rendering can be performed on standard PC graphics hardware.

Keywords: volume rendering, interactive, multi-texture, 2^{3n}-block, octree

1 Introduction

ボリュームレンダリングは 3 次元のスカラー場を 2 次元の画面に表示するための可視化手法であり，その方法は大きく 2 種類に分類できる．一つは**間接法** (indirect volume rendering) と呼ばれる方法で，前処理によって表示する情報を抽出し，データの一部分を表示する．代表的なものに Marching Cubes 法 [6] を利用した等値面表示がある．もう一つは，全てのサンプル点の寄与を計算して全体を表示する**直接法** (direct volume rendering) と呼ばれる方法で，描画面から視線方向に沿って輝度を積分 (backward projection) する Ray Casting [15]，サンプル点を描画面に投影 (forward projection) する Cell Projection [13]，Shear-Warp Factorization [3]，Splatting[18] 等がある．通常ボリュームレンダリングという場合には直接法を指し，本論文ではこちらを扱う．

ボリュームレンダリングを用いると高画質でデータの全体の様子を把握することが容易な画像を得ることができるため，工学，医学，娯楽などの分野で幅広く利用されている．しかし計算に必要な時間や記憶容量が大きいことから，速度が問題にならない場合や，大規模な計算機環境や特殊ハードウェアを使える場合に利用が限られている．

計算コストが高い理由としては主に 1) 視線に沿った輝度の累積計算の際に必要なデータのメモリ局所性が低く，メモリアクセスの効率が悪い点，2) 入力データの再サンプリングの際に必要な 3 次線形補間が複雑である点，3) 多くの場合，入力データ量が膨大である点，が挙げられる．速度を向上させるために，必要のない累積計算を途中で打ち切る early ray termination 最適化 [4] や，octree[5] や k-d 木 [14] を使ってデータの局所的性質を利用する方法等，アルゴリズムに対する改良がされている．しかしこれらの最適化を行っても依然として計算量は大きいため，インタラクティブな速度で描画するために，並列計算機 [16] や専用の特殊ハードウェア [9] を利用する研究もされている．

一方で PC グラフィックスカードの描画速度と機能が向上したため，これを利用してさまざまな可視化手法を通常の PC で実現することが盛んに研究されている [10]．ボリュームレンダリングに関しても，ボリュームを座標軸に垂直なスライスの重ねあわせで表現し，テクスチャマップされたポリゴンの α ブレンディングを使って描画する方法 [1] が提案されている．

スライスを使う方法は，描画にハードウェアの加速が利用できるため高速であるという利点がある一方，スライスと垂直な方向へのデータの補間・サンプリングが行われないために画質が悪く，またグラフィクスハードウェアの記憶容量の制限から，扱えるデータサイズが小さいという欠点がある．

本論文では，スライスの重ね合わせに基づくボリュームレンダリングを元に，画質と扱えるデータサイズの 2 点に関して改良する．1 点目に関しては，グラフィクスハードウェアの multi-texture 機能を利用し，描画に必要なメモリ容量を増やすことなくスライスと垂直な方向にサンプル点を増加させ，結果の画質を向上させる．2 点目に関しては，高速度描画に適したボリュームの適応的サンプリングを行って入力データのサイズを縮小し，ハードウェアの容量を上回るボリュームも高速に描画することを可能にする．また以上の手法を一般の PC グラフィクスハードウェア上で実装し，十分大きいサイズのボリュームデータを，十分な画質で，インタラクティブな速度で描画できることを示す．

以下 2章でボリュームレンダリングの方法と問題点について説明し，3章で画質を改善する手法を述べ，4章でデータサイズの問題を改善する方法を提案する．5章で具体的な実装について述べ，6章で結果を，7章で結論を述べる．

2 Direct Volume Rendering

2.1 Rendering Equation

ボリュームレンダリングは，描画面から視線方向に輝度を累積する方法 (backward projection) とサンプル点を描画面に投影する方法 (forward projection) に分けられるが，どちらの場合も描画の方程式は共通で，描画面の各画素から視線方向に伸びる光線に沿ってボリュームの輝度値を積分する．

描画面からの光線に沿った距離を λ，ボリューム内の位置を $\boldsymbol{x}(\lambda)$ とすると，描画面上の画素値 I は

$$I = \int_0^D \tilde{c}\big(\boldsymbol{x}(\lambda)\big) exp\Big(- \int_0^\lambda \tau\big(\boldsymbol{x}(\lambda')\big) d\lambda' \Big) d\lambda \qquad (1)$$

で求められる．ここで $\tilde{c}$ は各点で放出される色 (RGB 値) を表す関数，τ は各点における光線の減衰率を表す関数である．これらを用いて，ボリュームデータとして与えられるスカラー値 $s(\boldsymbol{x})$ に表示色を割り当てる．通常 τ は $s(\boldsymbol{x})$ の関数として定義されるが，*tildec* の計算は次の 2 つの段階を経て行われる．

まず最初にスカラー値 $s(\boldsymbol{x})$ から色 (RGB 値) への変換関数を通すことで，各点のスカラー値による色の寄与を決定する (classification)．これはボリュームの各点から放出される光の効果に相当する．この変換関数は通常ユーザーによって定義され，レンダリングの途中で変更されることがあるが，視点には依存せず一定である．次に各点と光源の位置関係から与えられる色情報を付加する (shading)．これには光の拡散反射や鏡面反射の成分が含まれる．classification の効果と shading の効果は個別に計算して，加算により合成することが可能である．入力のスカラー値の分布をそのまま可視化する場合には，classification だけを使って描画することもできるが，物体の幾何形状を表示する際には，3 次元の空間を正確に認識するために shading が必要である．ボリュームに対して shading を行う具体的な方法は5 章で説明する．

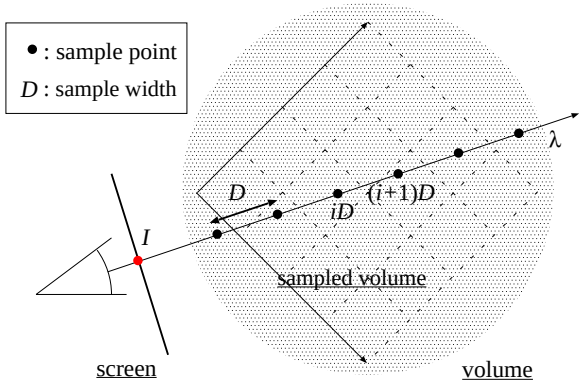


Figure 1: Volume sampling during direct volume rendering

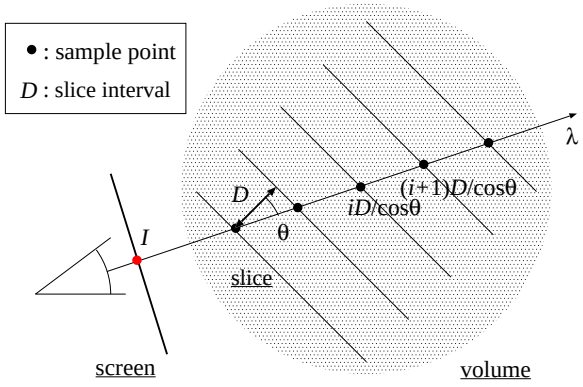


Figure 2: Volume sampling during slice-based direct volume rendering

2.2 Discretization

描画計算を行うために式 (1) の光の減衰率の部分を離散化すると

$$exp\Big(-\int_0^{\lambda}\tau\big(\boldsymbol{x}(\lambda')\big)d\lambda'\Big) \approx \prod_{j=0}^{\lambda/D} exp\Big(-\tau\big(\boldsymbol{x}(iD)\big)D\Big) \quad (2)$$

$$\approx \prod_{j=0}^{\lambda/D}(1-\alpha_j) \quad (3)$$

と書ける．ただし D はサンプリング幅，i はサンプル点の番号で，

$$\alpha_i = 1 - exp\Big(-\tau\big(\boldsymbol{x}(iD)\big)D\Big) \quad (4)$$

と置いた．α_i は不透明度 (α 値) と呼ばれる値である．

式 (4) で定義された α_i を使って，$\tilde{c}_i = \alpha_i c_i$ と表せる．ここで c_i は透明度を考慮しない RGB 値であり、$\alpha_i c_i$ は opacity-weighted color または associated color と呼ばれ，不透明度を考慮した時の色である [7]．離散化された i 番目の微小領域領域からの色の寄与を

$$C_i \approx \alpha_i c_i \quad (5)$$

と書くと，式 (4) を使って式 (1) を離散化した結果は

$$I \approx \sum_{i=0}^{n}\alpha_i C_i \prod_{j=0}^{i-1}(1-\alpha_j) \quad (6)$$

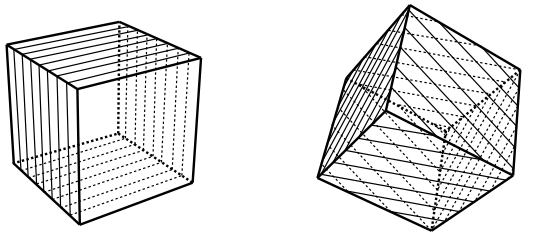
となる．光線の無限遠方から i までの累積輝度 C'_i を使ってこの式を変形して，

$$C'_i = \alpha_i C_i + (1-\alpha_i)C'_{i+1} \quad (7)$$

を得る。最終的に求めたい値は $I = C'_0$ である．

2.3 Slice-based Volume Rendering

式 (7) は，ボリュームのサンプリングを視線方向に伸びる光線に沿って一定間隔で行い，描画面から遠いほうから順に RGB 値を繰り返し α ブレンディングすることでボリュームレンダリングの計算が実現できることを示している．そこで，ボリューム全体を視線方向になるべく垂直な複数のスライスの積み重ねで表現し，テクスチャマップされたポリゴンの α ブレンディング計算でボリュームレンダリングを行うことを考える (Figure 2)．このように表現することでグラフィクスハードウェアのポリゴン表示機能を利用して高速なボリュームレンダリングを実現できる．



object-aligned slices　　image-aligned slices

Figure 3: Slices perpendicular to volume axis (left) and eye direction (right)

ハードウェアがソリッドテクスチャ(3 次元テクスチャ) をサポートしている場合には，視線に垂直な面 (image-aligned slice: Figure 3右) を生成し，ボリュームをソリッドテクスチャとして持つ方法が可能である [2]．視線と面が常に垂直であるためには，視点変更の度にスライスの位置とテクスチャ座標を再計算する必要がある．

ソリッドテクスチャが利用できない場合には 2 次元のテクスチャマップを利用する．ボリュームを座標軸に沿って切断したスライス (object-aligned slices: Figure 3左) の集合を生成して，ボリュームの断面画像を 2 次元テクスチャとしてマップする．表示の際には視線方向と座標軸の関係を調べ，視線とスライスの法線のなす角が常に一定以下になるようにスライスする軸の方向を選択する．スライス軸の変更の再にテクスチャを切り替える必要があり，インタラクティブ性を保証するためには x, y, z 軸方向のスライスを 3 組用意しておく必要がある．この場

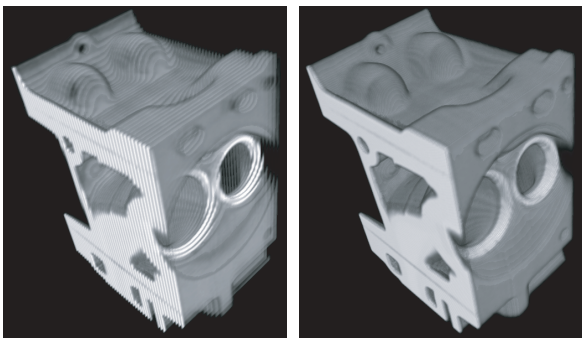


Figure 4: Visual artifacts are caused by the lack of trilinear interpolation (left) and can be eliminated by inserting intermediate slices (right).

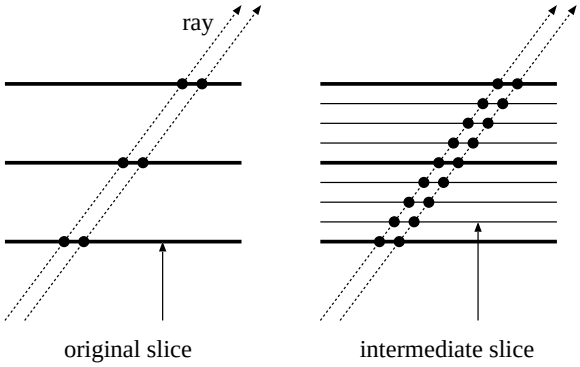


Figure 5: Inserting intermediate slices

合，ハードウェアの制限から各テクスチャ画像はメモリ中で連続領域を占めていなければならないので，テクスチャは軸方向ごとに合計 3 組の複製を持つ必要がある．

多くのハードウェアでソリッドテクスチャよりも 2 次元のテクスチャのほうが高速のため，object-aligned なスライスを切り替えて表示する方法が描画速度は速い．

3 Improvement of visual quality

通常のボリュームレンダリングでは入力のスカラー値を 3 次線形補間しながらサンプリングするのに対し，スライスを使ったレンダリングではスライス上で 2 次線形補間しながらサンプリングする．スライス方向には任意の解像度でサンプリングが可能であるのに対し，スライスの法線方向には一定間隔でしかサンプリングできないため，描画する解像度や視点を変更すると，補間精度の不一致により画質上の不具合が現れることがある．Figure 4 左に見られる縞状の模様は，スライスを使ったレンダリングの際に，スライスと垂直な方向への補間精度が不足することによって生じるモアレである．

3 次補間を近似してサンプリング幅を合わせるの方法のひとつに，元のスライス間に上下のスライスを線形補間して得られる**中間スライス**を挿入する方法がある (Figure 5)．モアレを防ぐためには，中間スライスの発生間隔を，描画の際のスライス上でのサンプリング間隔と同程度にする必要がある．中間スライスのテクスチャはメモリ上に保持するのではなく，multi-texture 機構を使って描画時に生成できる [11]．

隣接するスライス S_i と S_{i+1} の間の，$i+\alpha$ の位置に発生させる中間スライス $S_{i+\alpha}$ は

$$S_{i+\alpha} = (1-\alpha)S_i + \alpha S_{i+1} \tag{8}$$

と定義できる．座標の補間により中間スライスの位置を決定し，新規に発生したポリゴンにグラフィクスハードウェアの multi-texture 機能で生成したテクスチャをマップする．

モアレを防ぐためには，ポリゴンの描画の際のポリゴン上でのサンプリング幅とスライス幅が同程度であればよい．そこで描画の際に画面上で 1 つのスライスが閉める領域を調べ，そのサイズに応じて中間スライスの発生枚数を変更することでモアレを解消することができる (Figure 4右)．中間スライスの発生枚数を変更すると式 (4) の D が変わるため，各点における α 値を更新する必要がある．

4 Adaptively sampled 2^{3N}-block

テクスチャマップと α ブレンディングを使うと，グラフィクスハードウェアの加速を利用して高速にボリュームレンダリングできるが，テクスチャ保持するビデオメモリ (VRAM) のサイズには制限があるため，メモリ容量を上回る巨大なデータを描画する際には速度が劇的に落ちる．一般に VRAM は主記憶と比べて 10 倍から 100 倍程度小さく，サイズの制限は大きな問題になる．

データの局所性を仮定できるボリュームデータに対しては，octree 表現を使うことでメモリの利用効率を上げることが期待できる．しかし octree の階層構造が多段になり複雑化すると，データの局所的な連続性が落ちることからデータアクセスに必要な時間が増す．したがって処理速度が必要な場合には octree を使うことは必ずしも効果的ではない．多重解像度表現のメモリ効率の優位性を確保しながら同時に処理速度を落とさない表現方法として，8 分木 (octree) の一般化である 2^{3N} 分木 (2^{3N}-tree) を使う方法 [8] がある．この方法は，処理の対象となるシーン全体が巨大で，物体への距離に応じた詳細度制御 (levels of detail) や視界の外側にあるデータの切り捨て (view frustum culling) が必要がある場合に効果的である．

一方，通常のボリュームレンダリングでは，多くの場合データの解像度は画面の解像度より十分小さいため，階層構造のレベルを上下する必要はない．またテクスチャの読み込みには比較的時間がかかるため，インタラクティブ性を確保するためには，常に全てのデータを最高解像度で保持しておくほうが望ましい．そこで，本研究ではデータを階層表現せずに固定解像度のブロックで表現することでデータ構造を簡略化してインタラクティブ性を確保すると同時に，ブロック内の局所性を利用してデータ量を削減することでメモリ利用の効率化を行う．

まず最初にボリューム全体 V を固定サイズ 2^{3N} のブロック V^B に分割する (**2^{3N}-blocking**)．このとき各ブロックには 2^{3N} サイズの部分ボリュームが割り当てられる．次に各ブロック毎に解像度を N から順に 1 つ減少させて，その際に生じる元のデータとの誤差を計算する．全てのブロックで誤差計算を行い，最も誤差が小さいブロックの解像度を下げる操作を繰り返すことで，全体としてデータのサイズを減少させながら，各ブロックに対する最適な解像度を決定する．

ブロック番号を B として，このブロックにおける解像度 n のボリュームを V_n^B と書くと，元のボリュームは V_N^B と表される．解像度を変化させた時，V_N^B と V_n^B の誤差は，それぞれのボリュームを V_N^B のサンプル点の位置で 3 次線形補間を使ってサンプリングした値の自乗誤差で定義する．即ち，ブロック B 内の座標 (i,j,k) における V_n^B の値を $v_n^B(i,j,k)$ とすると，ブロック B の解像度 n における元のボリュームとの誤差 R_n^B は

$$R_n^B = \sum_{(i,j,k)\in V_N^B} \left| v_N^B(i,j,k) - v_n^B(i,j,k) \right|^2 \qquad (9)$$

と表せる．全てのブロックに対して式 (9) を計算し，この値が最も小さくなるブロックの解像度 n を減少させ，インタラクティブな速度で処理を行うのに必要なサイズまで全体のボリュームサイズを縮小する．Figure 6に実際のデータに対するブロック化の過程とデータサイズの変化の様子を示す．以上のボリュームサイズ縮小のアルゴリズムをまとめると以下の通り．

ADAPTIVELY SAMPLED 2^{3N}-BLOCKING

$S_t \leftarrow$(インタラクティブに処理できるサイズ)
{ ボリューム V を 2^{3N} ブロック B の集合に分割 }
for all $B \in V$ **do**
  {B のボリュームを V_N^B で初期化 }
end for
while (ボリューム全体のサイズ) $> S_t$ **do**
  for all $B \in V$ **do**
    $i \leftarrow$(B ボリュームの解像度)
    $\Delta R^B = R_{i-1}^B - R_i^B$
  end for
  {ΔR^B が最小の B のボリューム V_i^B を選択 }
  $V_i^B \leftarrow V_{i-1}^B$
end while

$R_n^B = 0$ の場合は画質を全く変えずに解像度を減らすことができる．そうでない場合は，最も画質の損失が少ないと期待されるブロックから順に，必要メモリ容量がハードウェアがサポートしている容量に収まるようになるまで，解像度を落としていく．

image-aligned なスライスを使ったボリュームレンダリングでは，2^{3N}-blocking を行った後，各ブロックに 1 つのソリッドテクスチャ，1 組のスライス集合を与えて，ブロックごとに描画する．object-aligned なスライスを使ったレンダリング時には，同様のブロック化のほかに，2 次元の各スライス内で 2^{2N}-blocking をすることができる．これによって全体として同じ誤差でもボリュームのサイズをより小さくすることが可能である．ただし，この場合には，誤差があまり大きくなると，隣接スライス間での解像度の変動が大きくなり，結果としてモアレが強く現れることがある．

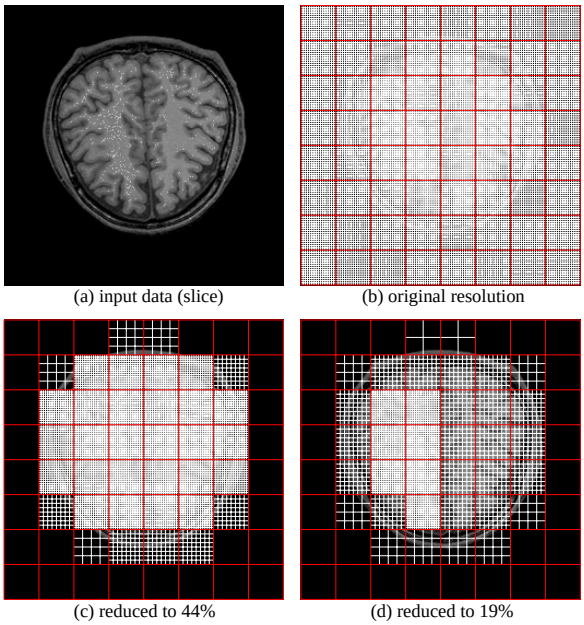


Figure 6: The size of input volume (a)(b) can be reduced to 44% (c) and 14% (d) by using 2^{3N}-block indicated by red squares.

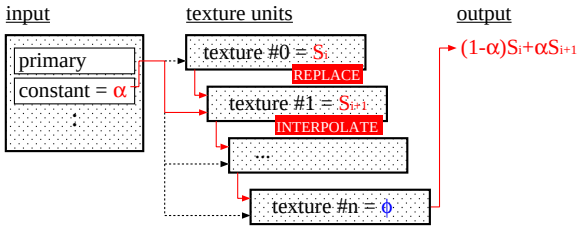


Figure 7: Texture unit configuration for intermediate slices generation

5 Implementation

5.1 Intermediate slices

中間スライスを生成するためには，混合比を指定しながら multi-texture 処理を行う必要がある．本研究では OpenGL グラフィクスライブラリ [12] の `COMBINE` テクスチャ環境使ってこれを実現する．

multi-texture が利用できる環境では 2 つ以上のテクスチャユニットが Figure 7のように接続している [12]．各ユニットは別々のテクスチャ，テクスチャ座標，テクスチャ環境を持ち，それぞれ `TEXTURE` (テクスチャ値) の他

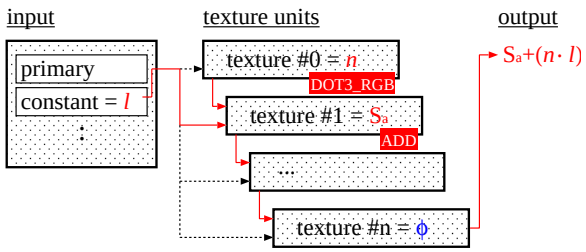


Figure 8: Texture unit configuration for shaded direct volume rendering

に，PRIMARY_COLOR (diffuse 値)，PREVIOUS (直前ユニットの出力)，CONSTANT (ユーザー定義の定数) を入力として利用できる．

中間スライスのテクスチャの生成ははじめの 2 つのユニットで行う (Figure 7)．まず定数として中間スライスの位置 α を保存し，ユニット 1 で S_i の，ユニット 2 で S_{i+1} のテクスチャ値を取得する．ユニット 1 は REPLACE テクスチャ環境を使い，テクスチャ値をそのまま取り出す．ユニット 2 では COMBINE テクスチャ環境の INTERPOLATE 関数を使い，RGB および α チャネル共にテクスチャ値を α ブレンドする．INTERPOLATE 関数は 3 つの引数をとり，それぞれ $Arg0, Arg1, Arg2$ としたとき

$$Arg0 \cdot Arg2 + Arg1 \cdot (1 - Arg2) \tag{10}$$

という式でテクスチャを混合するので，$Arg0$ = TEXTURE (= S_i)，$Arg1$ = PREVIOUS (= S_{i+1})，$Arg2$ = CONSTANT (= α) とすると式 (10) より

$$\alpha S_i + (1 - \alpha) S_{i+1} \tag{11}$$

となり，これは式 (8) の中間スライスの式と一致する．生成された中間スライスのテクスチャは，常のテクスチャと同様に扱うことができる．

テクスチャと同様にポリゴンの座標を α で補間して得られる中間スライスのポリゴンに対して，この補間されたテクスチャをマップすると，中間スライスが得られる．中間スライスを他のスライスと同様に α ブレンディングすることで，処理に必要なメモリ容量を増やすことなく，スライス枚数を増やすことができる．

5.2 Volume Shading

3 次元形状の認識を容易にするために，光源を考慮した shading を行うことは有効である．特にデータが均質で不透明度が高い場合，光源を考慮しないボリュームレンダリングでは元の形状の輪郭線以外を識別することは困難である (Figure 9)．

ここでは，光源の影響として拡散反射のみを考え，Gourand shading を使って表示する方法を説明する．2.1章で述べたように，ボリュームは各点において光を放出すると考えられるので，描画結果はこの放射光と拡散反射光の合成になる．光源方向を $\boldsymbol{l}$，ボリュームの輝度勾配

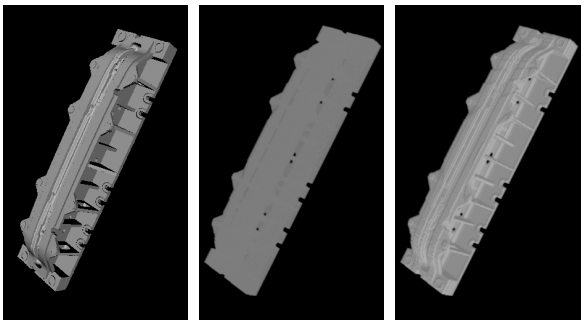


Figure 9: Result of polygonal surface rendering (left), direct volume rendering (center) and shaded direct volume rendering (right).

を $\boldsymbol{n}$，ボリュームの各点における光の放出を I_e，光源の光の反射を I_d とすると各点での shading は

$$I = I_e + I_d(\boldsymbol{n} \cdot \boldsymbol{l}) \tag{12}$$

を使って計算する．

あらかじめ，適当なフィルタ処理を行ってスカラー場の勾配ベクトルを計算しておき，その値である 3 次元ベクトルをボクセルの RGB 値として保持するボリュームを用意しておくことで，効率よく拡散反射成分を計算することができる [17]．ここでは COMBINE テクスチャ環境の DOT3_RGB 関数を使うことで光源のパラメータをインタラクティブに変更する方法を説明する．

DOT3_RGB 関数はテクスチャ値の RGB チャネルを 3 次元ベクトルとみなし，その内積結果をテクスチャ値の RGB チャネルに返す関数である．この関数は 2 つの引数をとり，それぞれ $Arg0, Arg1$ としたとき，

$$\begin{aligned} 4\times \Big(& (Arg0_R - 0.5) \times (Arg1_R - 0.5) + \\ & (Arg0_G - 0.5) \times (Arg1_G - 0.5) + \\ & (Arg0_B - 0.5) \times (Arg1_B - 0.5) \Big) \end{aligned} \tag{13}$$

を RGB 値としてそれぞれ返す．返り値は $[0, 1]$ で切り捨てられる．この関数を使うことで拡散反射の計算に必要な内積 $\boldsymbol{n} \cdot \boldsymbol{l}$ を計算する．光源の色 I_d は $\boldsymbol{n} \cdot \boldsymbol{l}$ の結果に MODULATE 関数を使って I_d を掛けることで実現できるが，ここでは簡単のために $I_d = (1, 1, 1)$ (白色光) の拡散反射のみを考える．最後に ADD 関数を使って放射光の反射成分と混合する．

具体的には，Figure 8にあるようにテクスチャユニットにおいて，定数に光源方向 $\boldsymbol{l}$ を与え，1 つ目のテクスチャユニットで輝度勾配 $\boldsymbol{n}$，2 つ目のユニットで放射光 I_e を取得する．そして 1 つ目のユニットで DOT3_RGB 関数を使って $I_d(\boldsymbol{n} \cdot \boldsymbol{l})$ を計算し，2 つ目のユニットで ADD 関数を使って $I_a + I_d(\boldsymbol{n} \cdot \boldsymbol{l})$ を計算する．

6 Experiments

OpenGL グラフィクスライブラリを利用して PC 上で実験を行った．使用した環境は CPU が Pentium4 1.7GHz,

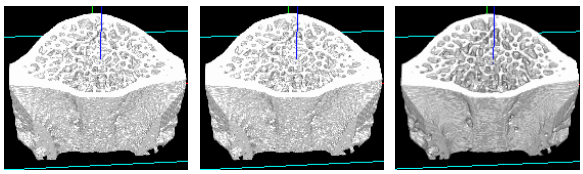


Figure 10: Visual result of 2^{3N}-blocking. Left is the rendered image of the origianal volume, which can be reduced to 44% of original size without any data loss (center), and drastically decimated to 14% (right).

主記憶が 1GB，グラフィクスハードウェアは GeForce3 AGP4x 64MB DDR，OS が Microsoft Windows 2000 である．

3章で述べた，中間スライスを使った 3 次線形補間の近似を行った結果を Figure 4に示す．入力はサイズは 256×256×128 で描画面サイズは約 2 倍である．元の結果では特に輝度変化の激しいところでモアレが観察できる (左図) が，各スライス間に中間スライスを 1 枚発生させてスライス枚数を 2 倍に増やした結果，画質が改善された (右図)．

4章で述べた，ブロック単位での適応的な解像度変化を行った場合のサイズの変化を Table 1に示す．誤差の許容値は，元のボリュームと空ボリューム (全ボクセルが 0) との差の 5%とした．2^{3N} ブロックを使った場合に 25%から 75%程度の，2^{2N} ブロックを使った場合にはさらにデータを縮小できていることが分かる．

一般に，断層撮影などから得られるボリュームデータは全体の 70%から 95%が透明領域であり [4]，この部分のデータ量をブロック化により効果的に削減できる．一般にボリュームの縮小化を進めるほど画質は劣化するが，2^{3N}-blocking を使うとほとんどが質の劣化なく，データ量を大幅に削減できる．Figure 10に実際のデータに対して blocking を行いデータを削減した時の画質の変化を示す．

ブロックサイズは，小さくなると描画時のオーバーヘッドが増加し，大きくなるとデータの局所性を利用しにくくなるため，最適値は処理系とデータに依存する．実験では，256×256×256 サイズの 8bit ボリュームデータでさまざまなサイズに関して描画速度を比較した結果，$N = 5$，すなわち 1 辺の長さが 32 の場合に最も高速となった (Figure 11)．$N = 4$ の時データサイズが最も小さくなるが，描画速度が高速にならないのは，ブロック数が増えることによる描画のオーバーヘッドが増加するためだと思われる．

最後に，さまざまなデータに対して 2^{3N} ブロック化を行い，レンダリング速度の変化を調べた結果を Table 2に示す．様々なデータに対して，解像度変化無し，誤差の許容値 5%，許容値 10%の単純化を行った結果，単純化の効果はデータの性質に依存するが，一定の速度向上が得られており，10%の許容値を設定することで巨大なボリュームをインタラクティブな速度で描画できる事がわかる．

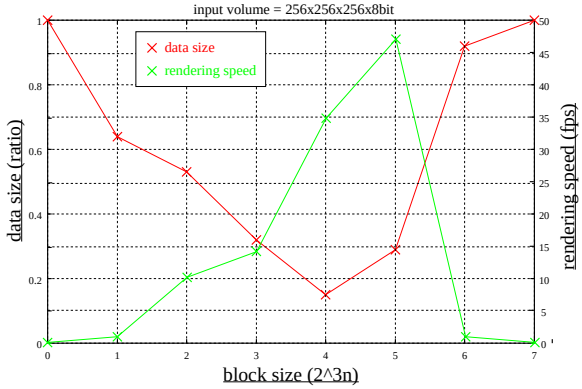


Figure 11: Relation between block size, volume size and rendering speed with the tolerance 5%.

Table 1: Change of total data size (Mbyte) by applying 2^{3N}- and 2^{2N}-blocking with block size 32^3.

	original	2^{3N}-blk.	2^{2N}-blk.
Data1 ($128\times128\times128$)	2.0	1.24	0.64
Data2 ($256\times256\times128$)	8.0	6.1	3.2
Data3 ($256\times256\times256$)	16.0	12.3	4.3
Data4 ($256\times256\times256$)	16.0	12.0	11.6
Data5 ($512\times512\times512$)	128.0	28.3	13.5

Table 2: Acceleration of rendering speed with change of the tolerance of decimation (fps)

	original	5%	10%
Data1 ($128\times128\times128$)	>47.5	>47.5	>47.5
Data2 ($256\times256\times128$)	17.3	>47.5	>47.5
Data3 ($256\times256\times256$)	4.5	24.7	>47.5
Data4 ($256\times256\times256$)	4.5	13.2	15.2
Data5 ($512\times512\times512$)	<0.1	6.5	24.7

7 Conclusions

PC グラフィクスカードの機能を使ってインタラクティブなボリュームレンダリングを行う手法を提案した．グラフィクスカードの加速を利用するためにスライスを使ったレンダリングを行う際に問題となる現れる視覚的な画質の問題を中間スライスを使って解決し，扱えるサイズの問題をスライスに適した単純化を行うことで改善した．その結果，特殊なハードウェアなしに高画質のボリュームレンダリングをインタラクティブな速度で実現でき，最大 $512 \times 512 \times 512$ のサイズの巨大なデータに対してもほとんど画質を損なうことなく十分な速度で描画することができた．

ボリュームレンダリングは計算量が大きく，インタラクティブに行うためには特殊な計算機や専用のハードウェアが必要であった．本研究の手法を用いると，通常の PC でも大きいサイズのボリュームデータをインタラクティブにレンダリングできることから，従来速度の点からボリュームを扱うことができなかった分野への応用が期待できる．

本研究では，直接法のボリュームレンダリングだけを対象としたが，等値面抽出を同じ枠組みで行う方法，またポリゴンとボリュームを同時に表示する方法，などが今後の課題である．

References

[1] M. Brady, K. Jung, H.T. Nguyen, and T Nguyen. Two-phase perspective ray casting for interactive volume navigation. In *Visualization 97*, 1997.

[2] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *symposium on Volume visualization*, Oct 1994.

[3] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the SIGGRAPH annual conference on Computer graphics*, 1994.

[4] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. on Graphics*, pp. 245–261, July 1990.

[5] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. on Graphics*, Vol. 9, No. 3, July 1990.

[6] W.E. Lorensen and H.E. Cline. Marching cubes: a high resolution 3d surface reconstruction algorithm. In *Proceedings of the SIGGRAPH annual conference on Computer graphics*, 1987.

[7] N. Max. Optical models for direct volume rendering. *IEEE Trans. Visualization and Computer Graphics*, 1995.

[8] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *Proceedings of the SIGGRAPH annual conference on Computer graphics*, 1999.

[9] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Proceedings of the SIGGRAPH annual conference on Computer graphics*, pp. 251–261, 1999.

[10] Hanspeter Pfister, Michael Cox, Peter N. Glaskowsky, Bill Lorensen, and Richard Greco. Why the pc will be the most pervasive visualization platform in 2001. In *Proceedings of the conference on Visualization 99*, 1999.

[11] C. Rezk-Salama, K. Engel, M.Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[12] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.3)*.

[13] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Workshop on Volume Visualization, Computer Graphics*, 1990.

[14] K. R. Subramanian and Donald S. Fussell. Applying space subdivision techniques to volume rendering. In *Visualization 90*, 1990.

[15] H. Tuy and L. Tuy. Direct 2d display of 3d objects. *IEEE Mag. Computer Graphics and Applications*, 1984.

[16] Guy Vezina, Peter A., Fletcher, and Philip K. Robertson. Volume rendering on the maspar mp-1. In *Workshop on Volume Visualization*, Oct 1992.

[17] Rudiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the SIGGRAPH annual conference on Computer graphics*, 1998.

[18] L Westover. Footprint evaluation for volume rendering. *Computer Graphics*, Vol. 24, No. 4, Aug 1990.