

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マッピング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ピング機能と 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るため、multi-texture 機能を使ってサンプリング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, オクトツリー

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 this method 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 drawback of sliced-based rendering is that insufficient interpolation leads to severe visual artifacts, which can be successfully eliminated by increasing sampling rate using multi-texture function. Another problem is the limitation of data size because of the limitation of the amount of video memory that is much smaller than main 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

ポリウムレンダリングは3次元のスカラ場を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、データの一部を表示する。代表的なものに Marching Cubes 法 [] 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 [], サンプル点を描画面に投影(forward projection)する Cell Projection [], Shear-Warp Factorization [], Splatting [] 等がある。通常ポリウ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。

ポリウ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、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ため、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次元線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 [] や、octree [] や k-d 木 [] 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 [] や専用の特殊ハードウェア [] 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 []。ポリウムレンダリングに関しても、ポリウ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 [] 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、またグラフィッ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ポリウ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、グラフィッ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

サンプリング点を増加させ、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ポリウ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ポリウ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グラフィックスハードウェア上で実装し、十分大きいサイズのポリウムデータを、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章でポリウ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

ポリウムレンダリングは、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ポリウ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 、ポリウム内の位置を $x(\lambda)$ とすると、描画面上の画素値 I は

$$I = \int_0^D \tilde{c}(x(\lambda)) \exp\left(-\int_0^\lambda \tau(x(\lambda')) d\lambda'\right) d\lambda \quad (1)$$

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、ポリウムデータとして与えられるスカラ値 $s(x)$ に表示色を割り当てる。通常 τ は $s(x)$ 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。

まず最初にスカラ値 $s(x)$ から色(RGB値)への変換関数を通すことで、各点のスカラ値による色の寄与を決定する(classification)。これはポリウ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。入力スカラ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。ポリウムに対してshadingを行う具体的な方法は5章で説明する。

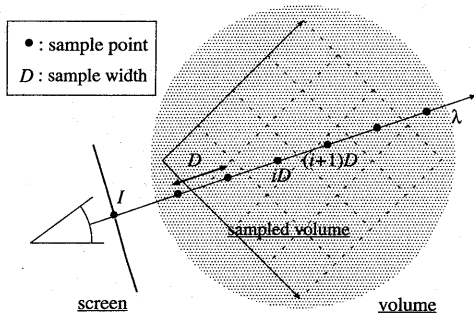


Figure 1: Volume sampling during direct volume rendering

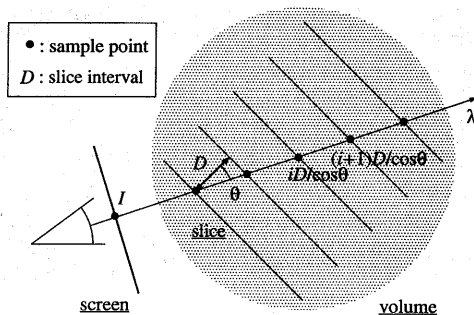


Figure 2: Volume sampling during slice-based direct volume rendering

2.2 Discretization

描画計算を行うために式(1)の光の減衰率の部分に離散化すると

$$\begin{aligned} \exp\left(-\int_0^\lambda \tau(x(\lambda'))d\lambda'\right) &\approx \prod_{j=0}^{\lambda/D} \exp(-\tau(x(iD))D) \quad (2) \\ &\approx \prod_{j=0}^{\lambda/D} (1 - \alpha_j) \quad (3) \end{aligned}$$

と書ける。ただし D はサンプリング幅、 i はサンプリング点のインデックスであり、

$$\alpha_i = 1 - \exp(-\tau(x(iD))D) \quad (4)$$

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、 $\bar{c}_i = \alpha_i c_i$ と表せる。 c_i はその点が完全に不透明なときに与えられる RGB 値である。不透明度との積で表される RGB 値 $\alpha_i c_i$ 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 []。離散化された i 番目の微小領域領域からの色の寄与を

$$C_i \approx \alpha_i c_i (x(iD))D \quad (5)$$

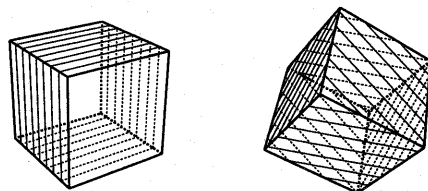
と書くと、式(4)を使って式(1)を離散化すると

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

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、

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

となる。最終的に求めたい値は $I = C'_0$ である。



object-aligned slices

image-aligned slices

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

2.3 Slice-based Volume Rendering

式(7)は、ポリウ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ポリウ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、ポリウ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ポリウ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ポリウームレンダリングを実現できる。

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、ポリウ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 []。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。ポリウ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、ポリウームの断面画像を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性を調べ、視線とスライスの法線のなす角が常に一

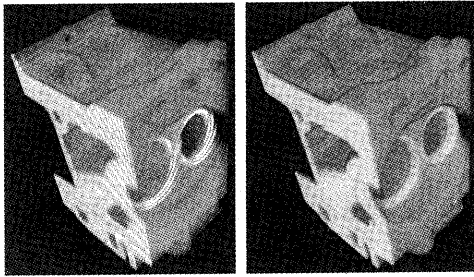


Figure 4: Visual artifacts are caused by the lack of trilinear interpolation (left) and can be eliminated by inserting intermediate slices (right).

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次にテクスチャを切り替える必要がある。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 []。

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

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

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

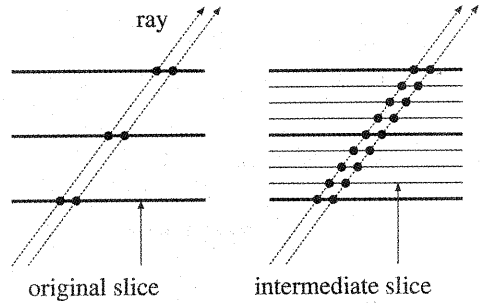


Figure 5: Inserting intermediate slices

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} -blocking

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、データアクセスに必要な時間が増すため、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 [] 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効率的な排除 (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。

まずボリューム全体を固定サイズ 2^{3N} のブロックに分割する。次に、元のデータとの誤差を評価しながら解像度 N を減少させていき、ブロック毎に最適な解像度を選択する (adaptively sampled 2^{3N} -blocking)。元の解像度のボリューム V 内でブロック B を占める部分ボリューム V^B と、 2^{3n} -blocking して得られるボリューム V_n^B の誤差は、3次線形補間を使ってサンプリングされた値の最小自乗誤差で定義する。

B 内の座標 (i, j, k) における V^B の値を $v(i, j, k)$ とすると、 V_n^B の点の値 $v_n(i, j, k)$ は、3次線形補間係数 $\tilde{x} = x - \lfloor x \rfloor$ を用いて

$$\begin{aligned} v_n(i, j, k) &= (1 - \tilde{i})(1 - \tilde{j})(1 - \tilde{k})v_{n-1}(\lfloor i \rfloor, \lfloor j \rfloor, \lfloor k \rfloor) + \\ &\tilde{i}(1 - \tilde{j})(1 - \tilde{k})v_{n-1}(\lfloor i \rfloor + 1, \lfloor j \rfloor, \lfloor k \rfloor) + \\ &(1 - \tilde{i})\tilde{j}(1 - \tilde{k})v_{n-1}(\lfloor i \rfloor, \lfloor j \rfloor + 1, \lfloor k \rfloor) + \\ &\tilde{i}\tilde{j}(1 - \tilde{k})v_{n-1}(\lfloor i \rfloor + 1, \lfloor j \rfloor + 1, \lfloor k \rfloor) + \\ &(1 - \tilde{i})(1 - \tilde{j})\tilde{k}v_{n-1}(\lfloor i \rfloor, \lfloor j \rfloor, \lfloor k \rfloor + 1) + \\ &\tilde{i}(1 - \tilde{j})\tilde{k}v_{n-1}(\lfloor i \rfloor + 1, \lfloor j \rfloor, \lfloor k \rfloor + 1) + \\ &(1 - \tilde{i})\tilde{j}\tilde{k}v_{n-1}(\lfloor i \rfloor, \lfloor j \rfloor + 1, \lfloor k \rfloor + 1) + \\ &\tilde{i}\tilde{j}\tilde{k}v_{n-1}(\lfloor i \rfloor + 1, \lfloor j \rfloor + 1, \lfloor k \rfloor + 1) \end{aligned} \quad (9)$$

と書ける。ただし $v_0 = v$ である。式 (refequation:trilinear) を用いると、この解像度における誤差は

$$R_n^B = \sum_{(i,j,k) \in B} |v(i, j, k) - v_n(i, j, k)|^2 \quad (10)$$

と表せる。式 (10) が最も小さくなるブロック B の解像度 n を減少させていき、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ボリュームを縮小する。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。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

```

 $S_i \leftarrow$  (インタラクティブに処理できるサイズ)
(ボリューム  $V$  を  $2^{3N}$  のブロック  $B$  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) >  $S_i$  do
  for all  $B \in V$  do
     $V_i^B =$  (現在のボリューム)
     $\Delta R^B = R_{i-1}^B - R_i^B$ 
  end for
   $V_i^B \leftarrow$  ( $\Delta R^B$  が最小の  $B$  のボリューム)
   $V_i^B \leftarrow V_{i-1}^B$ 
end while

```

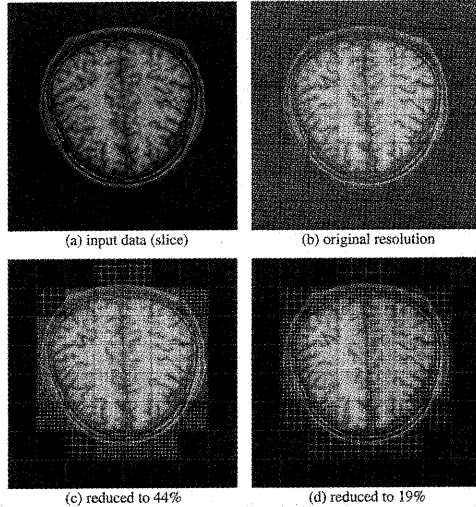


Figure 6: The size of input volume (a)(b) can be reduced to 44% (c) and 14% (d) by using 2^{3N} -block indicated by red squares.

$R_n^B = 0$ 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。

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。これを実現する方法として nVidia 社製グラフィックスカードの拡張機能である Register Combiner を使う方法があるが、本研究では OpenGL グラフィクスライブラリ [] の COMBINE テクスチャ環境を使って処理を行う。

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 []。各

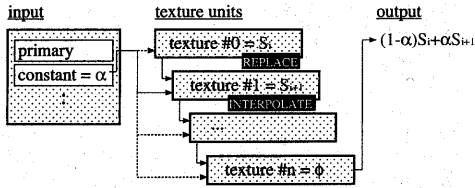


Figure 7: Texture unit configuration for intermediate slices generation

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。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 チャンネル共にテクスチャ値を α ブレンドする。INTERPOLATE 関数は3つの引数を取り、それぞれ $Arg0, Arg1, Arg2$ としたとき

$$Arg0 \cdot Arg2 + Arg1 \cdot (1 - Arg2) \quad (11)$$

という式でテクスチャを混合するので、 $Arg0 = \text{TEXTURE} (= S_i)$, $Arg1 = \text{PREVIOUS} (= S_{i+1})$, $Arg2 = \text{CONSTANT} (= \alpha)$ とすると式(11)より

$$\alpha S_i + (1 - \alpha) S_{i+1} \quad (12)$$

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ので、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、光源を考慮しないポリウ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 11)。

ここでは、光の拡散反射を Gourand shading を使って表示する方法を説明する。ポリウ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 l 、ポリウムの

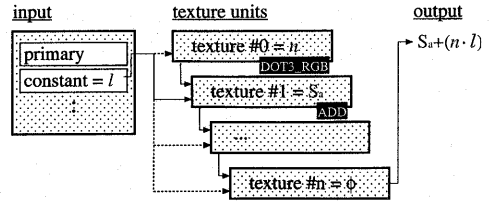


Figure 8: Texture unit configuration for shaded direct volume rendering

輝度勾配を n 、ポリウ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

$$I = I_e + I_d(n \cdot l) \quad (13)$$

を使って計算する。

あらかじめスカラー場の勾配を計算するフィルターを使って、勾配をボクセルのRGB値とするポリウムを用意しておくこと、効率よく拡散反射成分を計算できる []。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チャンネルを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チャンネルに返す関数である。この関数は2つの引数を取り、それぞれ $Arg0, Arg1$ としたとき、

$$4 \times \left(\begin{aligned} &(Arg0_R - 0.5) \times (Arg1_R - 0.5) + \\ &(Arg0_G - 0.5) \times (Arg1_G - 0.5) + \\ &(Arg0_B - 0.5) \times (Arg1_B - 0.5) \end{aligned} \right) \quad (14)$$

をRGB値としてそれぞれ返す。返り値は $[0, 1]$ で切り捨てられたため、直角より大きい角をなすベクトルの積は全て0になる。この関数を使うことで拡散反射の計算に必要な内積計算を行う。光源の色 I_d は内積 $(n \cdot l)$ 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。

これを実現するためには、Figure 8にあるようにテクスチャユニット上で、定数に光源方向 l 、1つ目のテクスチャユニットで輝度勾配 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 $I_d(n \cdot l)$ を計算し、2つ目のユニットで ADD 関数を使って $I_e + I_d(n \cdot l)$ 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、主記憶が 1GB、グラフィクスハードウェアは nVidia 社

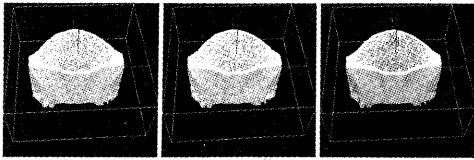


Figure 9: Visual result of 2^{3N} -blocking. Left is the rendered image of the original volume, which can be reduced to 44% of original size without any data loss (center), and drastically decimated to 14% (right).

製 GeForce3 AGP4x 接続, ビデオメモリが 64MB DDR, OS が Microsoft Windows 2000 である。

画質に対する改良として, 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。入力サイズは $256 \times 256 \times 128$ 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)。

サイズの問題に対する改良として, 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。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 [], 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 9 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。

ブロックサイズに関して, $256 \times 256 \times 256$ 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 10)。 $N = 4$ 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ボリュームの単純化を行い, レンダリング速度の変化を調べた結果を Table 2 に示す。単純化の効果はデータの性質に依存するが, 多くのデータに対して一定の速度向上が得られている事がわかる。

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

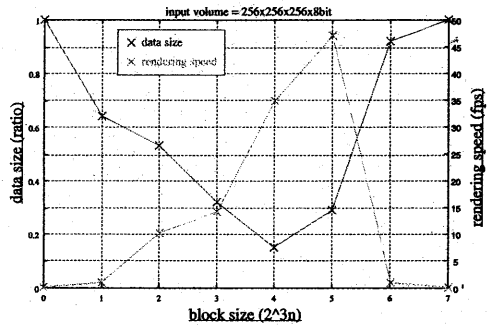


Figure 10: Relation between block size, volume size and rendering speed

Table 1: Change of total data size (Mbyte) by volume decimation by applying 2^{3N} - and 2^{2N} -blocking using blocks with sides 32 voxel length.

Input size	original	2^{3N} -blk.	2^{2N} -blk.
$128 \times 128 \times 128$	2.0	1.24	0.64
$256 \times 256 \times 128$	8.0	6.1	3.2
$256 \times 256 \times 256$	16.0	12.3	4.3
$256 \times 256 \times 256$	16.0	12.0	11.6
$512 \times 512 \times 512$	128.0	28.3	13.5

Table 2: Change of rendering speed (FPS)

Input size	100%	47%	14%
$128 \times 128 \times 128$	47.5	47.5	47.5
$256 \times 256 \times 128$	17.3	47.5	47.5
$256 \times 256 \times 256$	4.5	24.7	47.5
$256 \times 256 \times 256$	4.5	13.2	15.2
$512 \times 512 \times 512$	1.2	6.5	24.7

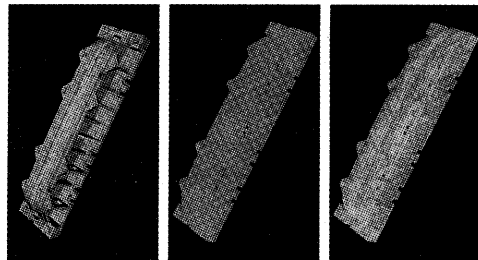


Figure 11: Result of polygonal surface rendering (left), direct volume rendering (center) and shaded direct volume rendering (right).

ムレンダリングをインタラクティブな速度で実現でき、また通常ではメモリに収まらないサイズのデータに対しても十分な速度で描画することができた。

ポリウ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ポリウ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、従来速度の点からポリウムを扱うことができなかつた分野への応用が期待できる。

ポリウムデータはサイズがサンプリング幅の3乗で増加するために、データの大きさが常に問題となる。本研究では描画は常に一定の解像度で行ったが、さらに大きいサイズのデータを扱うためにはインタラクティブに操作している間と静止しているときに解像度を変更させるなど、より柔軟な多重解像度表現を実現が必要であり、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.