特別研究報告書

汎用グラフィクスハードウェアによる
並列ボリュームレンダリングシステム

指導教官　　富田 眞治 教授

京都大学工学部情報学科

中田 智史

平成16年2月2日

汎用グラフィクスハードウェアによる
並列ボリュームレンダリングシステム

中田 智史

内容梗概

近年の計算機性能の急速な向上に伴い，大規模かつ高精度な数値シミュレーションを行い，その結果を可視化し，提示することが求められている．そしてその際には，ボリュームレンダリングによって可視化が行われるようになってきている．ボリュームレンダリングは対象を３次元的に中身の詰まったボリュームとして表現し，その内部構造や動的特性を表示するための技術である．ボリュームレンダリングによって可視化を様々な視点から対話的に行うことによって，ボリューム空間中の複雑な３次元構造を容易に理解することができる．そのため、対象へのインタラクティブな操作に対応してシミュレーションを行なうとともに，その結果を実時間で可視化することが科学や医療などの様々な分野で求められている．

しかし，ボリュームレンダリングは表示させる画像のピクセル数に応じた積分計算が必要となる．さらに，ボリュームデータは空間の最小単位であるボクセルの集合で表されているため，そのデータ量は膨大になる。そのために１秒間に 30 フレームの秒画を行う実時間可視化は一般に困難であり，並列計算機や専用のハードウェアを用いる必要があった．一方で，近年の PC グラフィクスカードでは飛躍的な描画速度と機能の向上により非常に高速に３次元の描画が行うことができるようになった．そのため，グラフィクスカードのハードウェアでサポートされたテクスチャマッピング機能とアルファブレンディング機能を用いて高速にボリュームレンダリングを行うことも可能となって来ている．しかし，医療データのような大規模なデータを扱う場合には膨大な演算量やグラフィクスカードの記憶容量，メモリ帯域幅の制限から，実時間で処理することはできない．

本稿では大規模なデータに対しても，特別なハードウェアなしに可視化を実時間で可能とすることを目指し，グラフィクスカードを用いて並列ボリュームレンダリングを行うシステムの実装について報告する．グラフィクスカードによるボリュームレンダリングでは主記憶からグラフィクスカードへのデータ転送の時間が最も問題となる．そこで，ボリュームデータを分割しグラフィクス

カードを備えたPCクラスタの各ノードに割り当て，それぞれでレンダリングした結果を1つにまとめる．このとき，クラスタの各ノードで生成されるレンダリング結果を1つのマシンに集めるための通信時間が問題となる．そこで，通信時間を短縮するために，ランレングスエンコーディングによってデータを圧縮して通信を行う．さらに，より高速に，より大規模なデータの描画を行うためにOctree構造によるボリュームデータ圧縮を実装し，並列化およびこれらの高速化手法の評価を行った．

その結果，グラフィクスカードのメモリサイズをこえるような大規模なデータに対して，並列化により，単一で処理するよりも2倍〜30倍の描画速度が得られた．ランレングスエンコーディングによる通信データの圧縮はネットワークの通信速度に依存する結果となった．100Mbpsではその効果が確認できたが，1Gbpsのネットワークを用いると，ネットワークの通信速度が上がったため通信時間に対する圧縮と解凍の処理時間の比が大きくなり，高速化には至らなかった．Octree構造によるボリュームデータの圧縮により，データサイズと不要標本点を削減でき，描画速度が向上し，512^3のデータを並列化のみの場合と比べて約5倍の速度で描画が可能となった．

Parallel Volume Rendering Using Standard Graphics Hardware

Satoshi NAKATA

Abstract

Today's rapid improvement of the computer performance brings a high expectation of a large scale numerical simulation and visualization of the resulte. And volume rendering is coming to be used for visualization. In volume rendering, the object regarded as a three dimentional volume which is filled. And volume rendering is a technology to display the internal structure and dynamic characteristic of an object. We can understand complicated three dimentionbal structure in volume space easily by visualizing by volume rendering technique interactively from various viewpoints. Simulation and real time visualization corresponding to intractive operation are required in the various fields such as science and medical treatment.

However, volume rendering needs integral computation in proportion to pixels of the image. Volume data consists of a set of voxels that are the minimum elements of space. And the size of volume data is generally enormous. Therefore it is difficult to render 30 images per second with volume rendering. So, it used to realize real time volume rendering only with a parallel computer or a special hardwear. On the other hand, today's great progress of PC graphics card makes it possible high speed rendering with PC graphics card is possible. And then, volume rendering with PC graphics card comes to be possible with texture mapping and alpha-blending function accelerated by graphics hardware. But real time volume rendering of large scale data st even with such a graphics card is still not possible for lacking of memory capacity , memory bandwidth and performance of graphics card.

In this paper, we describe the parallel volume rendering system using graphics cards to realize real time volume rendering of large scale data set without a special hardware for volume rendering. It takes much time to load volume data in graphics card from main memory. This is the biggest problem of volume rendering with graphics card . So, we divide volume data into 8 subvolumes and assigne one of them to each node of PC cluster equipped with graphics card.

Then, each node renders subvolume. After that, image that is created at each node as a result of volume rendering of subvolume is send to one machine. At this machine, an image is composed of these eight images.

Further, we propose two techniques to performe larger scale and higher speed rendering. One is image compression by run length encoding. The other is compression of the volume data based on octree data structure. By run length encoding, this technique intends to curtail communication time that is also a serious problem of parallel rendering between each node of cluster and a master machine. This thesis reports the implemetation and the evaluation of these techniques.

Consequently, it come to be possible to get from 2 to 30 times speed up, when we treat the larger data than the size of graphics memory. The result of compression by run length encoding depended on the netwoek. It is effective on 100Mbps network. But on 1Gbps network, time that it takes to encode and decode is more than the reduction of communication time by compressing. So we can't get speed up. And much higher speed rendering come to be possible by reduction of volume data and number of a sampling points that are not necessary based on octree data structure. We can obtain about 5 times speed up at 512^3 data set with image and voxel compression, comparing with the speed of parallel volume rendering without these compression.

汎用グラフィクスハードウェアによる
並列ボリュームレンダリングシステム

目次

第1章　はじめに

近年の計算機性能の急速な向上に伴い，大規模かつ高精度な数値シミュレーションへの期待が高まっている．そして，そのシミュレーションの結果として出てきた数値データを可視化し提示することが求められている．また，その可視化においてはボリュームレンダリングが用いられるようになってきている．ボリュームレンダリングは対象を3次元的に中身の詰まったボリュームとして表現し，その複雑な内部構造や動的特性を表示するための技術である．それにより対象へのインタラクティブな操作に対応してシミュレーションを行なうとともに，その結果を実時間で可視化することが科学や医療など様々な分野で求められている．

しかし，ボリュームレンダリングは表示させる画像のピクセル数に応じた積分計算が必要となる．さらに，3次元空間を格子で分割し，各格子にその地点のデータを持たせたボリュームは，空間全体をデータとして保持するためデータ量が膨大になる．そのため，1秒間に30フレームの秒画を行う実時間可視化は一般に困難である．しかし，近年のPCグラフィクスカードの飛躍的な描画速度と機能の向上により非常に高速に3次元の描画が行えるようになった．そして，グラフィクスカードのハードウェアによる加速を利用して，高速にボリュームレンダリングを実行することも可能となってきている．しかし，512^3や，それ以上のような大規模なデータを扱う場合には膨大な演算量やグラフィクスカードの記憶容量，メモリ帯域幅の制限から，単一のグラフィクスカードではまったく実時間では処理することはできない．

そこで，PCクラスタを用いて，大規模なデータに対しても並列化することによりインタラクティブな可視化を可能とすることを目指す．並列ボリュームレンダリングの環境として我々は次の3つの側面を考えている．処理に関して柔軟性を持つソフトウェアによる実現，超高速を目指す専用ハードウェアによる実現，比較的安価に高速な処理を行うことができるグラフィクスカードによる実現である．本稿ではグラフィクスカードを用いた実現に関して述べる．以下，2章で汎用グラフィクスハードウェアによるボリュームレンダリングについて説明し，3章でその並列化について述べる．そして，4章で評価を，5章でまとめを述べる．

第2章　汎用グラフィクスハードウェアによるボリュームレンダリング

まず，ボリュームレンダリングと近年のグラフィクスカードについて記す．その後でグラフィクスカードを用いてボリュームレンダリングを行う手法を述べる．

2.1　ボリュームレンダリング

ボリュームデータは空間の最小単位であるボクセル (voxel) の集合で表されている．[2] ボクセルは３次元座標上の点からサンプリングされたもので，もとの対象物について，その地点での１つもしくは複数の観測値や計算値（例えば，傾向強度，密度，電荷，温度など）を持たせることにより，空間全体をデータとして保持している．そして，格子点以外の地点のデータは，その地点近傍の格子点から線形補間を用いて求める．これにより，空間内のすべての地点におけるデータを得ることができる．

スクリーンのピクセルごとにボリュームデータ内へ光線を飛ばし，各ボクセルに割り当てられた不透明度，シェーディング値，そして入射光の強度を元とした，視線方向へのボクセル値の積分をボリューム空間を抜けるまで行う．それにより，各ピクセルの値を求め，レンダリング画像の作成を行う．

また， ボリュームレンダリングの積分計算の順番により，次の２つの方法 [2] がある．それを図１に示す．

- front to back
  スクリーンの各ピクセルから光線を飛ばして，スクリーンに近いほうからその光線上の各標本点の累積値を計算する．この場合，カラーと透明度の累積を求める計算式が必要となるが，透明度の累積を用いて Early Ray Termination[3] による高速化が可能となる．

$$C = C_d + A_s(1 - A_d)C_s \tag{1}$$

$$A = A_d + (1 - A_d)A_s \tag{2}$$

- back to front
  スクリーンの各ピクセルから光線を飛ばして，その光線上の標本点の値をスクリーンに遠いほうから足し合わせる．
  遠い標本点から順に足していくので，空間内のすべての標本点の累積計算

をしなければならないが，透明度に関する累積計算をする必要がなくなる．

$$C = A_s C_s + (1 - A_s) C_d \tag{3}$$

式 (1)(2)(3) において，C はカラー値，$C = (R, G, B)$，A は不透明度 (アルファ) としている．添え字 s は現在の標本点の値を示し，添え字 d は光線が現在の標本点にあたるまでの累積を示す．

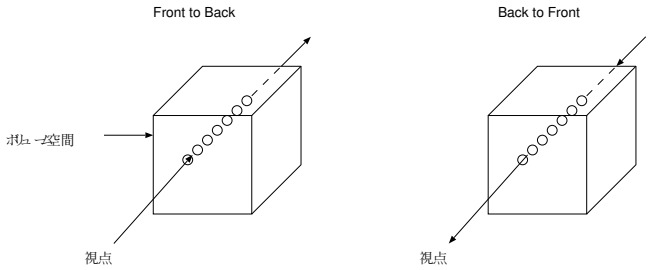


図 1: ボリュームレンダリング

2.2 グラフィクスカード

グラフィクスカードで行う 3 次元描画は次の 4 つの処理の繰り返しである (図 2)．

1. 点，線，ポリゴンといったプリミティブの幾何変換や光の加減の計算などのジオメトリ演算を行う
2. これらのプリミティブから画像を構成する 2 次元のフラグメントの集合へと変換 (ラスタライズ) される
3. テクスチャ等のフラグメント演算を行う
4. フレームバッファへ書き込む

フラグメント　プリミティブのラスタライズによって生成され，各フラグメントは単一のピクセルに対応し，カラー値，デプス値，さらにテクスチャを有効化している場合はテクスチャ座標の値を含むものである．

ピクセル　画像を構成する最小要素である．

テクセル　テクスチャデータを構成する最小要素である．

2.2.1 グラフィクスカードの構成

一般的なグラフィクスカードは図 3 のような構成である．

Vertex Engine(Vertex Shader)　形や色などを定義したモデルと光源を 3 次

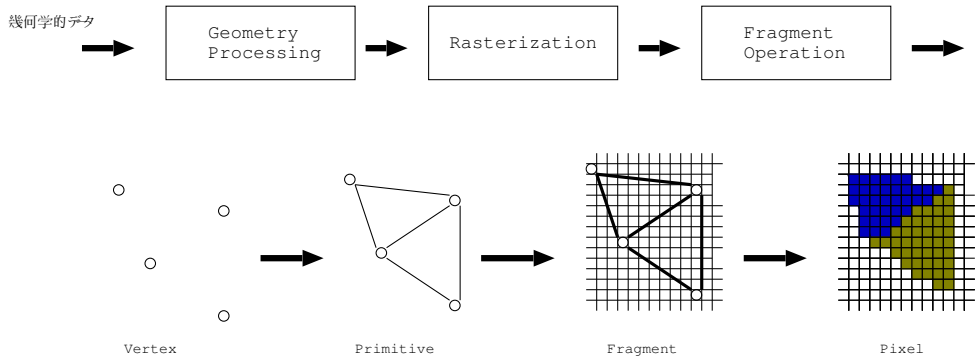


図 2: グラフィクスパイプライン

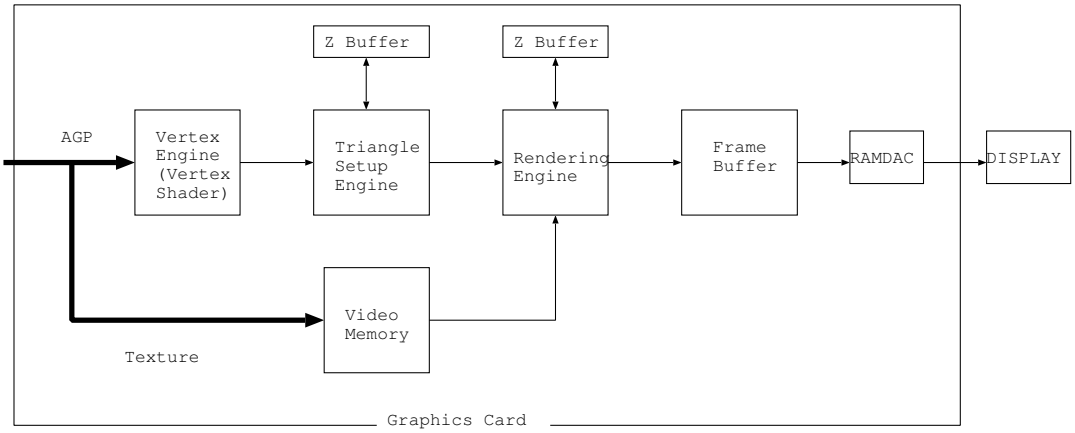


図 3: グラフィクスカード

元のステージ上に配置し，視点から見たそれらの形状や位置関係，光源が及ぼす明暗や陰影などの効果などを計算し，立体的な画像を描画していく．そのポリゴンの位置や物体に対する光の加減を計算するのが Vertex Engine である．

また，最近のグラフィクスカードはプログラマブル・バーテックス・シェーダーを備えている．これは 3 次元グラフィックスにおけるモデリング・データの頂点情報を，描画時にプログラムで制御する機能である．モデリング・データを変えることなく，最終的な描画データに移動や変形，ライティングの変更などを加えて，多彩な表現を可能にする．

これを用いることで，モーフィング，マトリックス・スキニング（関節のような頂点間の不連続部分を滑らかに処理する），ライティング・モデルなどを，プログラマ自身のアルゴリズムで記述できる．例えば，水面を制御することで波紋を描いたり，顔の皺を制御することで笑った表情を作っ

たりすることが可能である．CPU ではなく，グラフィックス・チップ内でこれらの処理を行うことにより，システムに負荷をかけずに表現力を大幅に向上させられる．

Triangle Setup Engine　Vertex Engine でのジオメトリ演算やライティングにより作成されたポリゴンからビットマップデータを作成する．

Rendering Engine　ラスタライズされたポリゴンや背景などに対して，テクスチャリングを行う．

また，最近ではピクセルシェーディング [4] が可能となっている．ピクセルシェーディングはモデルの表面を処理するシェーダーで，色や明暗，質感の表現はもちろん，複数のテクスチャを組み合わせて，ピクセル単位のライティング（バンプ・マッピング）や環境マッピングなどが実行できる．これにより，金属の質感などをプログラムによって描画することが可能である．特に有効なのは，リアルタイムの描画処理で，様々なエフェクトをかけたり，髪の毛や波紋の動きをリアルに表現したりといったことが，シェーダー側のプログラム処理で実現できる．

Z Buffer　各ピクセルの視点からの距離を保存する．Z バッファの値が大きいピクセルは，小さな値のピクセルに上書きされる．

Frame Buffer　上記のような処理の終了した結果を記憶しておく．

RAMDAC　フレームバッファのデジタルデータを，ディスプレイに出力できるようにアナログデータに変換する．クロックが高速になればなるほど帯域幅が上がり，より高解像度の表示が可能になる．

Video Memory　レンダリングパイプラインが処理する前のピクセルデータの一時的な展開場所として，さらには前述の Z バッファやフレームバッファとして使用される．また，テクスチャデータを格納しておく場所でもある．

2.2.2　Radeon9700Pro

本研究では ATi 社の Radeon9700Pro[5] を使用した．Radeon9700Pro では 4 つプログラマブル・バーテックス・シェーダーを持ち，レンダリングエンジンは 8 パイプ化された浮動小数点ピクセル演算ユニット (FPPU)(図 4) を内蔵し，複数の処理を並列実行することにより 3 次元グラフィックスの処理を大幅に向上させる．この FPPU で様々なフラグメント演算を行い，その結果をフレームバッファへと書き込む．

Radeon9700 では 1 パスで 16 テクスチャの描画が可能になっている．三角形

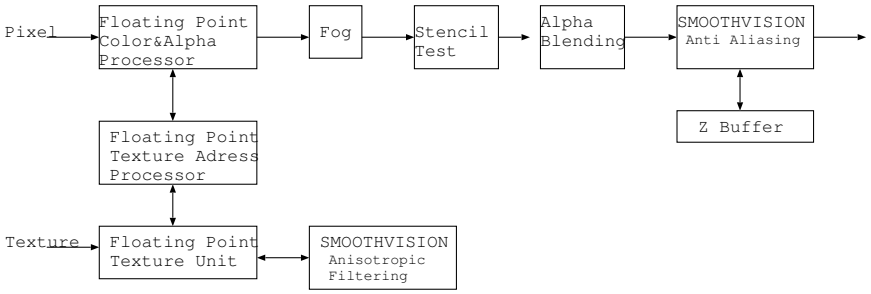


図 4: FPPU

描画速度は 3 億トライアングル/Sec 以上で，1 秒間に描画できるピクセル数は 2.4G ピクセル/Sec である．

ピクセルシェーダエンジンやバーテックスシェーダエンジンの並列処理化により，ビデオメモリやメインメモリへデータを読み書きするデータ量が大幅に増えるため，それらの帯域幅も大きくなっている．ビデオメモリのコントローラは 256bit 幅になっており，メモリのクロックは 620MHz なので，20GB/Sec のビデオメモリの帯域が確保される．メインメモリへのアクセスは AGP を利用して行なわれるが，AGP は AGP 8X(533MHz,32bit,2.1GB/Sec) をサポートする．これらのメモリへのアクセス帯域幅の広帯域化，高効率化により，より高解像度で大きなサイズのテクスチャを利用することが可能になっている．

また，Z バッファの処理を効率化する HyperZ III が用いられている．HyperZ III では Z バッファをブロックに分け，必要のない部分のピクセルをピクセルシェーダへ入れる前に捨ててしまう Hierarchical Z と Early Z，Z バッファの高速なクリアや圧縮などを行う Fast Z バッファと Z バッファ圧縮の機能により，メモリ帯域をより効率よく利用することが可能になっている．

2.2.3 その他のグラフィクスカード

本研究で使用した Radeon9700Pro(2) 及びその他の主なグラフィクスカード (Radeon8500(1),Radeon9800Pro(3), nVidea 社の GeForce4Ti4600(4),GeForceFX 5900Ultra(5)) の性能を表 1 にまとめる．

また，表に挙げたグラフィクスカードのメモリは Radeon9800 及び GeForceFX 5900 で DDR2 に対応し，その他は DDR SDRAM である．

表 1: グラフィクスカードの主なスペック

カード	(1)	(2)	(3)	(4)	(5)
コアクロック (MHz)	275	325	380	325	450
メモリクロック (MHz)	550	620	680	650	850
メモリ (MB)	64	128	256	128	256
メモリインターフェース (bit)	128	256	256	128	256
メモリバンド幅 (GB/sec)	9.4	19.8	21.8	10.4	27.2
レンダリングパイプライン数	4	8	8	4	8
バスタイプ (AGP)	4X	8X	8X	4X	8X

2.3 グラフィクスカードによるボリュームレンダリング

グラフィクスカードによるボリュームレンダリング [7] ではボリュームをある軸に垂直なスライスの重ね合わせで表現する．ボリュームデータは伝達関数によりボクセル値からカラー要素を持つテクスチャデータへと変換される．

そのテクスチャデータを用いて，テクスチャマッピングとアルファブレンディング機能によりボリュームレンダリングが実現される．そのため，テクスチャベースのボリュームレンダリングを行う場合，その処理のほとんどはラスタライズステージで行われる．その反面ジオメトリ演算は非常に単純である．

2.3.1 アルファブレンディング

アルファブレンディング [6] とは前面の画像要素の色と後面の画像要素の色とを，アルファ値の示す比率で線形内挿して，合成画像の色を算出するものである．アルファ値とは各ピクセルの色を決定する RGB(赤，緑，青) 以外のもう一つの要素で，素材の不透明度を決定するものであり，通常 0.0 から 1.0 の間の値をとる．1.0 は素材が不透明で後ろにあるものは隠されるということを示し，0.0 は素材が完全に透明で見えないことを表す．

アルファブレンディングでは前面のカラー値－ソース (source) －が，現在保存されている後面のカラー値－ターゲット (destination) －と組み合わせて処理される．

ソースとターゲットの混合係数をそれぞれ $(A_s, 1 - A_s)$ とし，ソースとターゲットの RGBA 値を $(R_s, G_s, B_s, A_s)(R_d, G_d, B_d, A_d)$ で表示すると，最終的な

RGBA 値は次のように求められる．

$$(R_sA_s + R_d(1-A_s), G_sA_s + G_d(1-A_s), \\ B_sA_s + B_d(1-A_s), A_sA_s + A_d(1-A_s)) \tag{4}$$

2.3.2 テクスチャマッピング

テクスチャは物体の表面の質感を表現するために貼り付けるデータであり，テクスチャを物体を表す図形オブジェクトに貼り付けることをテクスチャマッピング [6] という．テクスチャマッピングは，非常に複雑な画像を大掛かりな図形モデルを構築することなく生成できる．ポリゴンの各頂点がテクスチャ画像上のどの点に対応しているのか，というのをあらかじめ関数として持っており，ポリゴンを描画するときに，それをもとにポリゴンに対応するピクセルの１点１点がテクスチャ上のどの位置（色）を参照しているのかを求めてその色で点を打つことでテクスチャの描画を実現する．

テクスチャ座標はポリゴンの各頂点がテクスチャ画像上のどの位置に対応するのかを表わすものである．テクスチャ座標を図 5 に示す．通常は 0.0〜1.0 の値を取り，テクスチャ画像の左下が，$(s=0, t=0, r=0)$ で，右上が（1,1,1) となる．

$$\boldsymbol{s} = \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

テクスチャ座標 (同次座標系)　オブジェクト座標 (同次座標系）

$\boldsymbol{M}$(4×4):変換行列 (平行移動, 回転等）

$\boldsymbol{T}$(4×4):オブジェクト座標とテクスチャ座標の対応を表す関数

とすると

$$\boldsymbol{s} = \boldsymbol{MTx} \tag{5}$$

によりオブジェクト座標とテクスチャ座標の対応を定め，それにより３次元オブジェクト空間内のある座標でのカラーをテクスチャにより決定する．

4 つのテクスチャ座標 (s,t,r,q) がテクスチャ行列で乗算される場合，その結果のベクトルは同次テクスチャ座標として解釈される．この q 座標は複数の射影や，透視変換が必要な場合に使用する．また，1 次元テクスチャの場合は s のみを，2 次元テクスチャでは s,t のみを，3 次元テクスチャでは s,t,r を用いる．

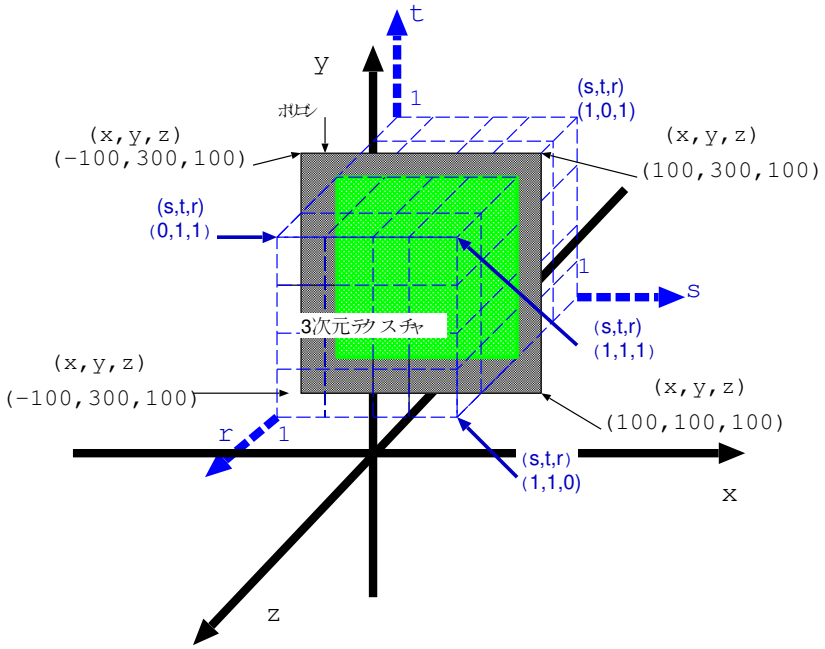


図 5: オブジェクト座標とテクスチャ座標

2.3.3 テクスチャベースのボリュームレンダリング

グラフィクスハードウェアでサポートされたテクスチャマッピングとアルファブレンディングを用いてボリュームレンダリングを行う．ボリュームデータのある軸に垂直な断面をテクスチャデータとしてポリゴンにマッピングし，それらを不透明度を用いて合成する．それにより，式 (3) の計算を全てのピクセルに対して実行することができる (図 6)．

視線に垂直な平面（ポリゴン）を複数用意し，それらにボリュームデータを厚みを持ったソリッドテクスチャ(3 次元テクスチャ）として与える．3 次元のテクスチャ座標とテクスチャ変換行列によってテクスチャが割り当てられている空間内の任意の平面に対するテクスチャ(物体の任意の断面) を構成することができる．この座標を用いて視線に対して垂直に並んだ各ポリゴン上にマッピングされるテクスチャを求める．そして，視点に遠いポリゴンから 2.3.1 で述べ

たアルファブレンディングにおいて．各要素の混合係数を $(A_s, 1 - A_s)$ とし，式 (3) の計算を行う．

視点変更の度に各ポリゴンへのテクスチャが再計算される．即ち，視点変更において実際に変更されるのはポリゴン上の図柄であり，ポリゴンの位置は常に一定である．

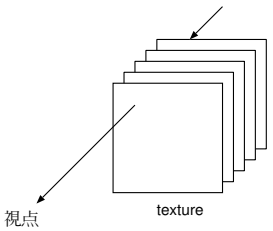


図 6: テクスチャベース

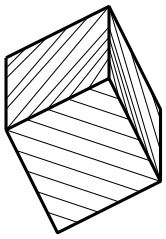


図 7: ３次元テクスチャ

第３章　並列化

本章では第 2 章で述べたテクスチャベースボリュームレンダリングを並列化する手法を示し，さらに 2 つの高速化手法について述べる．

3.1　並列ボリュームレンダリングの基本アルゴリズム

3.1.1　並列化

ボリュームレンダリングは光線に沿ったボリューム空間内の標本点の透明度の累積をスクリーン上のピクセル毎に行うものである．したがって，その処理はピクセル毎に独立であり，その計算は空間内を通過する光の積分計算である．よって，空間をいくつかに分割し，それぞれの区間に対する演算結果を足し合わせることで最終結果を得ることができる．そこで，ボリュームデータを x, y, z 全ての軸方向に 2 等分し，図 8 のように 8 つのデータ (サブボリューム) に分割する．その分割したデータをそれぞれクラスタの各ノードに分散してテクスチャベースでボリュームレンダリングし，各サブボリュームごとのレンダリング結果 (中間画像) を生成する（図 9）．

そして，フレームバッファ上でレンダリング結果が存在する部分だけを読み出し，それを一つのマスタに集める．データは最初に分配されるだけで回転等のパラメータの変更による再描画において，各ノードは現在保持しているサブ

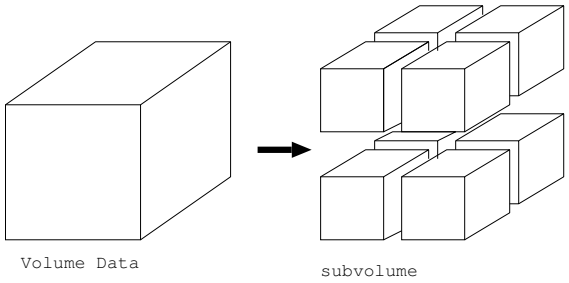


図 8: データ分割

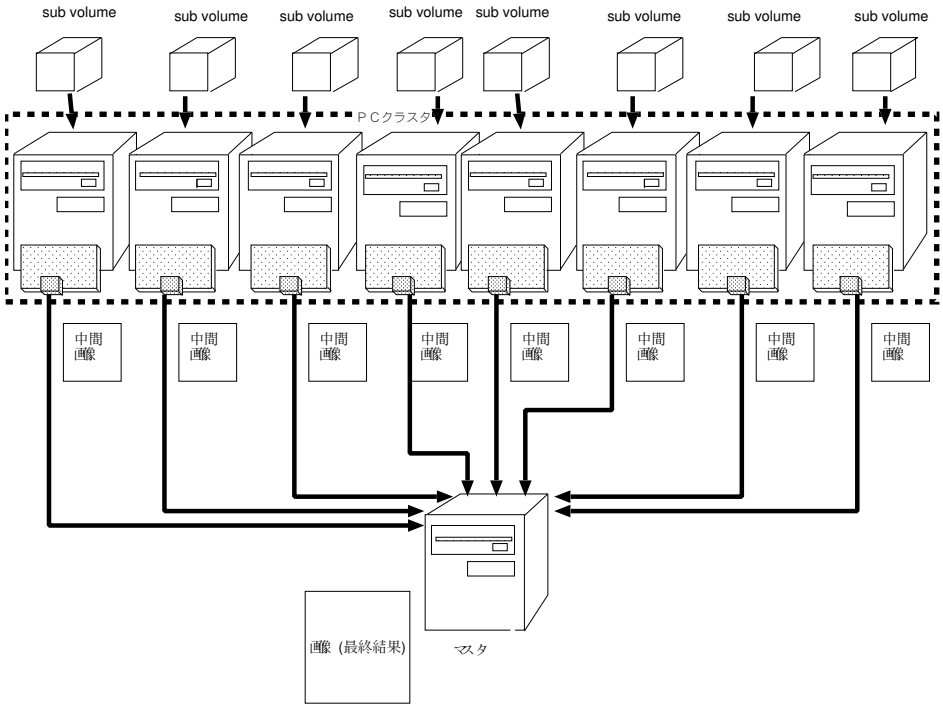


図 9: 並列ボリュームレンダリングシステム

ボリュームに関してのみレンダリングを行う．各サブボリュームからそれに対応するオブジェクト座標での代表点をとり，サブボリュームのスクリーンからの距離を得る．その距離を用いて各ノードで生成されたレンダリング結果の画像を視点に遠いものから順にスクリーン上の適切な位置に式 (3) によりアルファ値で合成することにより最終結果を得る．

3.1.2 不透明度の累積

分割された各サブボリュームに関してテクスチャベースでボリュームレンダリングを行う．しかし，図 8 のようにデータを分割して並列化を行う場合，2.3.3 で述べた単体でボリュームレンダリングを行う場合とは異なり，

カラー値 (RGB) に関する混合係数

$$(A_s, 1 - A_s) \longrightarrow C = A_s C_s + (1 - A_s) C_d \tag{6}$$

不透明度アルファに関する混合係数

$$(1, 1 - A_s) \longrightarrow A = 1 \times A_s + (1 - A_s) A_d \tag{7}$$

とカラー C と不透明度 A に関して異なる処理が必要となる．

並列にテクスチャベースでボリュームレンダリングを行う場合，奥のサブボリュームから生成した画像と，手前のサブボリュームから生成した画像とを重ね合わせなければならない．そのため，一つのサブボリューム全体の不透明度を適切に計算しておく必要があるからである．また，各ノードで処理しているボリュームの背後にもデータが存在する可能性があるので，初期化は必ず $(R, G, B, A) = (0, 0, 0, 0)$ としなければならない．

通常 Back to Front のボリュームレンダリングではターゲットのアルファ値を使用することはないが，図 8 のようにデータを分割する場合，各サブボリュームからのレンダリング結果の合成の際にアルファ値を使用する．混合係数を $(A_s, 1 - A_s)$ としたときのアルファ値の計算は不等明度の累積を正しく計算するものではないので，式 (7) の計算が必要となる．

ここでは OpenGL の拡張機能 (glBlendFuncSeparateEXT) を用いて，アルファブレンディングの際に RGB とアルファとに異なる係数を指定し，これを実現した．

この拡張機能が利用できず，混合処理の際に RGB とアルファに異なる混合係数を指定できないグラフィクスカードの場合は，各テクセルの RGB にあらかじめアルファ値を掛けておく方法が考えられる．これは，テクスチャデータが透明度を考慮したカラー値 (intensity)[2] を持つ，というモデルであると考えることになる．テクスチャデータの各 R,G,B にあらかじめアルファ値を掛けて (AR,AG,AB,A) としておき，混合係数を $(A_s, 1 - A_s)$ ではなく，$(1, 1 - A_s)$ として計算することにより式 (6)(7) と同様の結果を得ることができる．

しかし，ピクセルの色を決定する際に補間を行っていると，厳密にはこれら 2 つの方法は等しい結果をもたらすものではないが，アルファをあらかじめ掛けておくことにより元の画像に十分近い結果を得ることが可能である．

3.2　中間画像圧縮による高速化法

並列化をすると，各ノードでのサブボリュームのレンダリング結果を 1 つのマスタに集める際の通信時間が問題となる．そこで，各ノードで生成された中間画像を転送する際に，生成された中間画像内の透明部分をランレングスエンコーディングにより圧縮することで通信データ量を減らし，高速化を図る．

ランレングスエンコーディングは繰り返しの情報を圧縮することによって，データのサイズを減少する方式である．重複する色の連続している部分を破棄して、データを圧縮する．これを画像中の透明部分に関してのみ行う．透明 $(A = 0)$ のピクセルは RGB の値に関わらず可視化結果に影響はないので，RGB の値は必要はない．そこで，送信しようとする画像の中に含まれる連続する $A = 0$ となるピクセルの個数を RGB の領域を用いて表し，データ量を減らす（図 10）．

この圧縮操作をクラスタの各ノードが生成した中間画像に対して行い，圧縮されたデータをマスタへと送る．マスタは各ノードから受け取ったデータを解凍し，合成する．

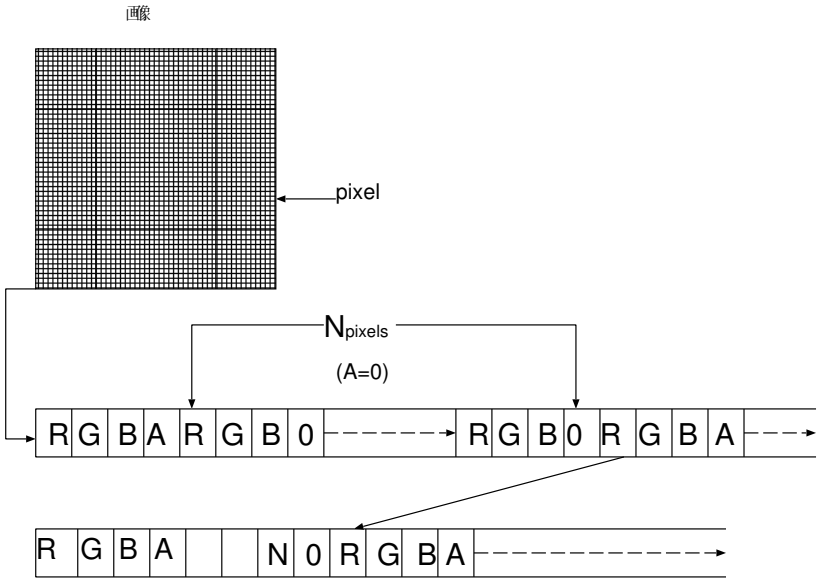


図 10: 画像圧縮

3.3　ボリューム圧縮による高速化法

巨大なボリュームデータを効率的に扱うために，サンプリング格子の密度を適応的に変えることにより，ボリューム空間をその重要度に応じた解像度で表

現することができる階層格子構造 [8] は有力なボリュームデータの表現手法とされている．

3.3.1 階層格子ボリュームデータ

これまでは均等にサンプリングされたボリュームデータのみを対象としてきた．これはグラフィクスカードで 3 次元テクスチャとして扱うことのできるボリュームデータは構造格子に分割されたものに限定されるからである．しかし，ボリュームデータでは空間内に非均質に物体 (状態) が分布している場合もある．そこで，空間内のデータの分布に応じて格子の密度を変えることによりデータサイズの減少を図る．一つのボリュームデータを分割しそれぞれ異なるデータと考えることによりグラフィクスカードで解像度が異なるボリュームデータを扱う．

階層格子ボリュームデータは解像度が異なる複数の立方体格子により構成される．代表的なものに Octree 構造があり，これにより空間中のデータの分布に応じて格子の密度を変えることができる．

3.3.2 Octree 構造

解像度の異なる複数の立方体格子系を持つボリュームデータを Octree 構造により構成する．Octree 構造はボリュームデータを x, y, z 各軸方向に 2 等分し，8 分割されたデータを木の節点の各要素に割り当て，この分割された各データに対して同様の 8 等分を繰り返すことにより立体を構成する構造である．ブロック内のすべてのボクセル値が等しくなるまで分割されるとそこで分割を止める．そして，その等しい値をもつボクセルのみからなる 1 つのブロックをその値を持つ 1 つのボクセルで置き換えることにより，データ量を削減することができる (図 11(b))．

さらに，そのボクセル値に対応するカラーのアルファ値が 0 ならば，完全に透明なのでそのデータは可視化結果には影響をもたらさない．そのため，1 ボクセルに置き換えるのではなく，完全に削除することができる (図 11(c))．

また，大きなデータを小規模のデータの集合として扱うことで，データの局所性を利用しやすくなるという利点もある．

3.3.3 Octree 構造データを用いたテクスチャベースドボリュームレンダリング

ボリュームデータを 8 分割し，その分割したデータを木の節点に割り当てる．さらに分割できる場合はこの作業を繰り返し，ボリュームデータを Octree 構造で表現する．本稿では Octree 構造において分割回数をレベルとよび，最上位層

から順にレベル $1, 2, \cdots, n$ と定義する (図 12)．Octree の各節点は 8 つの要素を持ち，下位レベルの節点へのリンクもしくはボリュームデータへのポインタを格納する．このようにデータを分割し，それらを異なるテクスチャとして扱い，異なるポリゴンにマッピングする．

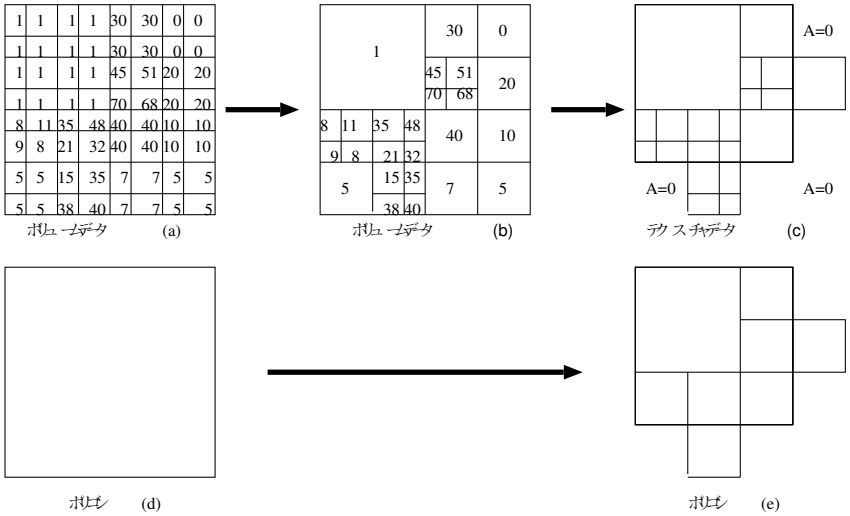


図 11: Octree 構造ボリュームデータ分割

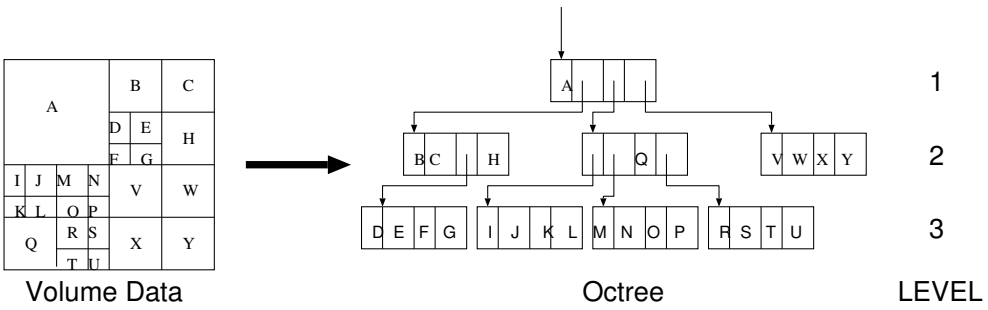


図 12: ボリュームデータの Octree 表現

このようにして分割されたデータを視点から遠いブロックから順にテクスチャベースボリュームレンダリングを行なう．その際，前後関係を本質的に保持する必要があるものだけに半順序関係を与えることでレンダリングするべきブロックの順を決めることができる．

即ち，1 回分割した 8 つのサブボリュームに関して視点からの距離によってソーティングをし，その結果を用いて，Octree の各節点で 8 要素のソーティン

グ結果の順にリストをたどる．これによりレンダリングするべき順番が得られるので，ソーティングはデータの分割回数によらず，常に 8 要素に関してのみ行う．1 回分割したサブボリュームの位置関係は，1 つのサブボリュームをさらに分割した 8 つのブロックに関しても同様である．また，そのブロックに関してさらに分割を繰り返しても，やはり同様のことが言える (図 13)．そのため，最初の 8 つのサブボリュームのソーティング結果を繰り返し用いることができる．

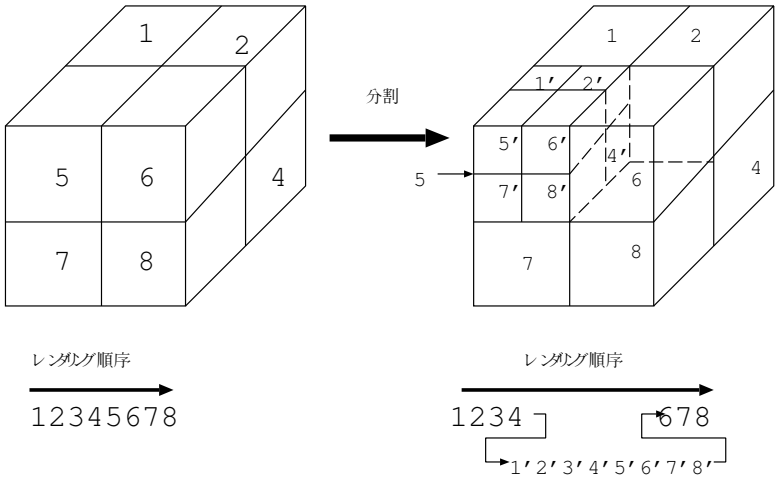


図 13: Octree 構造データのレンダリング順序

処理するデータのレベルに応じて，そのデータをマッピングするポリゴンの大きさと枚数を決定する．処理するデータのレベルを $level$ とすると，元のポリゴンを x, y 軸方向に $1/2^{level}$ の大きさにし，枚数を $1/2^{level}$ にして描画する．それらのポリゴンが 1 要素のテクスチャに対応しているときは，テクスチャマッピングは行わず，その色を持つポリゴンを描画する．但し，このように $1/2^{2level}$ の大きさのポリゴンを 2^{level} 枚描くことになるので，全体の面積は変わらないが，ポリゴン数は増える．そのため，ジオメトリ演算が増加する．また，分割したすべてのテクスチャに対してオブジェクト座標との対応関数を設定するなど，テクスチャの切替えのオーバーヘッドも増えることにもなる．

アルファ値が 0 となり削除されたボリュームデータに対応するポリゴンは描画しない (図 11(e))．これにより，レンダリング結果に影響を及ぼさない標本点に関する演算を省くことが可能である．

クラスタの各ノードで割り当てられたサブボリュームについて，CPU で Octree

を構成する．分割したデータを異なるテクスチャとしてグラフィクスカードに格納しておく．そして，CPU で先に述べたようにそのリストを順にたどり，データへのポインタを持つ節点に到達すると，そのデータに関してボリュームレンダリングをグラフィクスカードで行う．

第 4 章　評価

4.1　環境

OpenGL グラフィクスライブラリを用いて PC クラスタ上で，汎用グラフィクスカードによるボリュームレンダリング，及びその並列化，データ圧縮を次のような環境で行った．

表 2: 実装環境

	CPU	主記憶	Graphics Card	OS
マスタ	Pentium4 1.6GHz	512MB	Radeon9700Pro	Linux 2.4.19
PC クラスタ	Pentium3 1GHz	512MB	Radeon9700Pro	Linux 2.4.18-3

PC クラスタは 4CPU である．Radeon9700Pro のビデオメモリは 128MB DDR SDRAM であるが，その中には 2.2.1 で述べた Z バッファ，フレームバッファ等に使用する領域も含まれるので，テクスチャデータを格納するために割り当てられるメモリの大きさは 128MB より小さくなる．また，4.7 の実験を除きネットワークは 100 Mbps の Ethernet である．通信の API としては UDP ソケットを用いた．

並列化はこの PC クラスタの各ノードにボリュームデータを 8 分割したサブボリュームを 2 つずつ割り当て，ボリュームレンダリングを行う．その結果の画像をマスタに集め，合成する．

スクリーンのサイズは 512×512 である．スクリーンサイズは視点変更の際に，ボリュームデータの投影がスクリーンからはみ出ることがないように，十分な大きさに設定した．正面から見たとき，サブボリュームをマッピングするポリゴンがスクリーンへ投影される領域は 170×170(ピクセル) である．そのため，各ノードがマスタに送信する画像はそれぞれ $170 \times 170 \times 4$(RGBA 各要素 8bit)$\times 2$(枚)$\approx 226KB$ となっている．

ボリュームデータはすべてRGBA32bitなので，表のサイズの4倍の大きさとなる．

この実験では次のようなボリュームデータを使用した（図14）．

- skull 頭蓋骨のボリュームデータでサイズは128×128×64である (図14-(a))．
- bonsai-1 盆栽のボリュームデータでサイズは 256 × 256 × 128 である (図14-(b))．
- bonsai-2 盆栽のボリュームデータでサイズは 256 × 256 × 256 である (図14-(c))．
- bonsai-3 盆栽のボリュームデータでサイズは 256 × 256 × 512 である (図14-(b))．これはbonsai-1を奥行き方向に4本並べたものである．
- tree クリスマスツリーのボリュームデータでサイズは256×512×512である (図14-(d))．
- chest 人間の胸部のボリュームデータでサイズは512×512×512である (図14-(f)))．

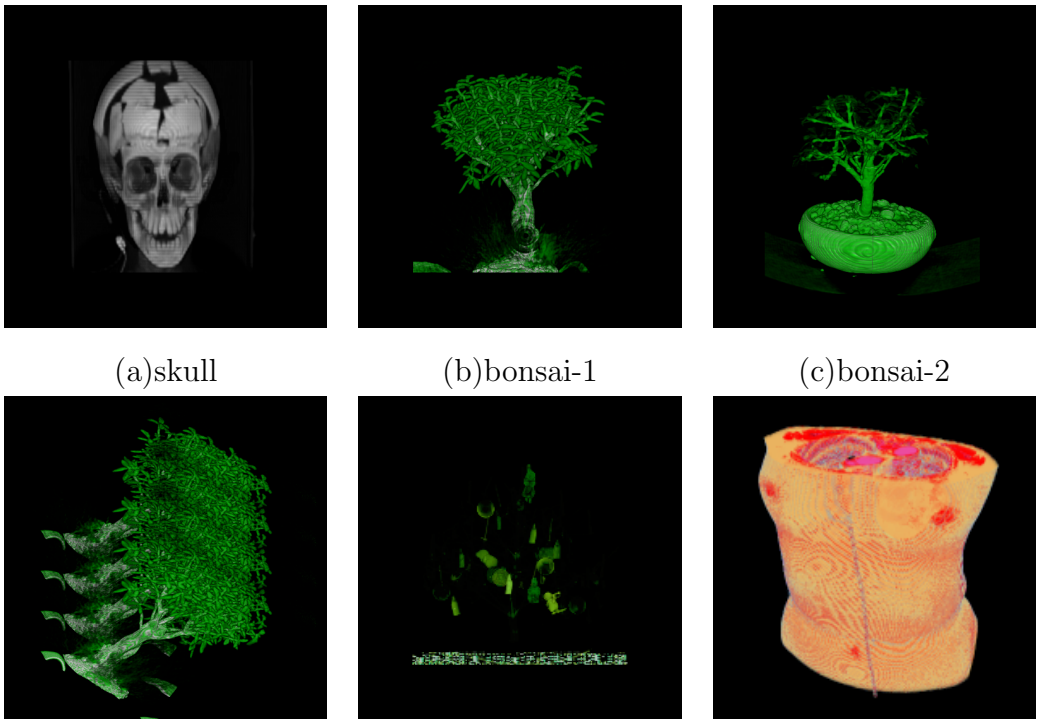
盆栽とクリスマスツリーに関しては，bonsai-1のカラーテーブルを使用した．chestは$R = 255sin(voxel \times \pi), G = 255sin(2 \times voxel \times \pi), B = 255cos(voxel \times \pi), A = 255(1 - exp(-voxel \times \pi))$とした．以降のすべての結果において描画速度は視点を固定し一定方向からのレンダリングを複数回繰り返し，それに要した時間の平均とし，その単位はfps(frame per second)である．

データをマッピングするポリゴンの枚数は4.2を除きボリュームデータの各軸方向のサイズの最大値としている．

4.2 2次元テクスチャと3次元テクスチャの比較

テクスチャベースボリュームレンダリングは2次元テクスチャを用いて実現することも可能である．しかし，2次元テクスチャでは，あらかじめ元のボリュームデータのx, y, zそれぞれの軸に垂直なスライスを3組用意 (三重化）しておかなければならいので，データ量が3倍になってしまうという欠点がある．

３次元テクスチャを用いると2次元テクスチャの時のような三重化によるデータ量の増加の問題は解決される．しかし，多くのグラフィクスカードでは３次元テクスチャよりも2次元テクスチャのほうが高速に処理できる．Radeon9700Proではskullデータで176.06fps(2次元テクスチャ)と112.490fps(３次元テクスチャ)，bonsai-1データで80.98fpsと29.46fpsであった．但し，ポリゴン数はskullで64



(a)skull (b)bonsai-1 (c)bonsai-2

(d)bonsai-3 (e)tree (f)chest

図 14: ボリュームデータ

枚，bonsai-1 で 128 枚である．

しかし，データが大きくなり，グラフィクスカードのメモリの大きさをこえると，描画毎に主記憶からグラフィクスカードへのデータ転送が必要になる．そのデータ転送には非常に時間がかかる．そのため我々が想定するような大規模なデータを扱う場合，インタラクティブな描画を目指す上ではメモリの大きさによる制約のほうが問題となってくる．従って，本稿ではメモリ効率の良い 3 次元テクスチャにより実装した．

4.3 並列化の結果

グラフィクスカードによるボリュームレンダリングを並列化して行った結果を，1 台で単独にボリュームレンダリングした場合，図 8 のようにデータを分割してそれらをすべて 1 台で順にレンダリングを行った場合 (逐次) と比較して表 3 に示す．

表中の × は実行できないことを示す．これはテクスチャデータがグラフィクスカードのメモリの大きさを越えるためである．1 つのボリュームデータとし

表 3: 描画速度

	volume data	skull	bonsai-1	bonsai-2	bonsai-3	tree	chest
描画	1 台	54.26	14.19	9.20	×	×	×
速度	逐次 (1 台)	60.00	17.31	9.49	1.94	0.76	0.027
(fps)	並列 (4 台)	9.75	8.58	7.76	3.94	3.86	0.99

て扱うと，1 台のマシンでは $256^2 \times 512$ 以上の大きさのデータに対しては実行できなかった．データを分割することにより，サブボリュームがグラフィクスカードのメモリに格納できる大きさになったため，描画のたびにそれらのうちいくつかを入れ換えることによって 1 台でも $256^2 \times 512$ 以上の大きさのデータを扱うことはできた．しかし，そのような場合，極端に描画速度が落ちる．これは描画の度に主記憶からグラフィクスカードへのデータ入れ換えが必要となるためだと考えられる．主記憶からグラフィクスカードへのデータ転送の時間が大きく，表 3 の逐次の tree や chest のように非常に低速な描画しかできない．

並列化することにより，データがグラフィクスカードのメモリをこえる大きさになる bonsai-3,tree,chest ではその効果が見られるようになった．特に chest では約 30 倍の向上があった．

4.4 中間画像圧縮の結果

並列化を行った場合の 1 回の描画にかかる時間の内訳を表 4 に示す．

表 4: 描画時間の内訳

ボリュームデータ	bonsai-1	chest
全体	約 117msec(8.58fps)	約 1000msec(0.99fps)
レンダリング (各ノードでの画像生成)	約 25msec	約 910msec
通信 (画像データの転送)	約 86msec	約 86msec
合成	約 5msec	約 5msec

スクリーンの大きさは同じなのでレンダリング時間以外はボリュームデータによらない．

通信に時間がかかり，ボリュームデータが小さい場合には大きく影響する．そのため，並列化における中間画像の通信を 3.2 で述べたように画像圧縮して行った．その結果を表 5 に示す．圧縮率はもとの画像データサイズと圧縮操作後のデータサイズの差の比率である．また，表中の圧縮率の値は各ノードでの圧縮率の平均をとっている．

表 5: 中間画像圧縮による描画速度

	volume data	skull	bonsai-1	bonsai-2	bonsai3	tree	chest
描画速	非圧縮	9.75	8.58	7.76	3.94	3.86	0.99
度 (fps)	圧縮	11.85	9.05	9.18	3.93	4.22	0.93
圧縮率 (%)		31.73	18.35	37.08	16.63	65.85	0.72

skull や bonsai-2，そして特に高い圧縮率が得られる tree では，描画速度の向上が見られる．また，bonsai-1 でもわずかに描画速度は向上した．これらの中でサイズの小さいデータほど描画時間に占める通信時間の割合が大きいため，描画速度の上昇した割合は大きくなっている．

また，bonsai-3 ではほとんど変化が無く，ほとんど圧縮できない chest では逆に遅くなった．

4.5 Octree データ分割の結果

4.5.1 データ圧縮の結果

並列化したうえで Octree 構造ボリュームデータ分割を行った．データ毎に様々な分割レベルに関して描画速度を比較した結果を表 6，表 7 に示す．

表 6: Octree 分割による描画速度

	volume data	skull	bonsai-1	bonsai-2	bonsai-3	tree	chest
描	レベル 1	9.75	8.58	8.06	3.94	3.86	0.99
画	レベル 2	10.25	8.94	8.45	3.97	4.02	2.06
速	レベル 3	10.54	9.58	10.00	5.42	6.89	3.13
度	レベル 4	9.02	7.79	9.99	5.71	9.71	5.37
(fps)	レベル 5	3.94	1.82	6.98	0.066	9.46	2.92

表 7: Octree 分割による圧縮率

	volume data	skull	bonsai-1	bonsai-2	bonsai-3	tree	chest
圧縮率 (%)	レベル 1	0	0	0	0	0	0
	レベル 2	12.50	0	25.00	0	28.13	0
	レベル 3	32.80	0.57	53.91	0	55.67	24.22
	レベル 4	38.82	9.40	69.09	1.861	79.20	35.01
	レベル 5	63.40	34.01	79.13	16.22	85.30	42.29

分割数が少ないとデータ量の削減の効果がほとんど得られない．しかし，分割数を大きくするとデータ量を減らすことができる半面，処理するポリゴン数や描画時のオーバーヘッドが増加するため遅くなる．その最適値はデータと処理系に依存する．表 6 より $256^2 \times 512$(bonsai-3) より小さなデータにおいてはレベル 3 の場合が最も効率が良くなっている．それ以上のサイズのデータにおいてはレベル 4 がピークとなった．

4.5.2 データ参照の局所性

データを分割し，それをデータ圧縮，標本点の削減を行わずに 1 台で順にレンダリングした結果を表 8 に示す．

表 8: Octree 分割による描画速度

	volume data	bonsai-1	bonsai-2
描画速度 (fps)	レベル 0	14.91	9.20
	レベル 1	12.20	7.64
	レベル 2	12.41	7.73
	レベル 3	19.27	11.71
	レベル 4	5.60	5.57

いずれのデータもレベル 1 からレベル 3 までは分割するごとに描画速度が上昇し，レベル 4 で下がった．レベル 3 では分割せずに描画した場合 (レベル 0) を越える速度に達した．

また，データを分割して圧縮し，透明なポリゴンは描画しない場合 (1)，データ圧縮し，透明でもそれをテクスチャとして与え描画した結果 (2)，ブロック内

のデータがすべて等しい値なら，その色を持つポリゴンを描いた結果 (3) を比較した (表 9 および表 10).

表 9: bonsai-1 Octree レベル 4

	描画速度 (fps)	圧縮率 (%)
非圧縮	4.88	0
(1)	5.59	9.42
(2)	4.81	9.42
(3)	5.21	9.42

表 10: bonsai-2 Octree レベル 3

	描画速度 (fps)	圧縮率 (%)
非圧縮	10.73	0
(1)	23.61	53.91
(2)	15.48	53.91
(3)	15.72	53.91

この表の (1) と (2) の結果より，テクスチャデータサイズの圧縮とともに，標本点削減による演算量の減少の効果も大きく影響していることが分かる．また (2) と (3) より，差はわずかだがすべてのボクセル値が等しくなったブロックを 1 要素のテクスチャデータとして処理するよりも，その値を持つポリゴンとして描画したほうが速くなった．これは，テクスチャをマッピングしていないポリゴンでは，テクスチャの座標の計算と，テクスチャデータへのアクセスが不要になったためだと考えられる．

4.6 中間画像およびボリュームデータの圧縮の結果

並列化した上で中間画像の圧縮及びボリュームデータの圧縮を組み合わせた結果を表 11 に示す．Octree のレベルは最も効率のよいレベル (skull,bonsai-1,bonsai-2 でレベル 3，それ以外のデータではレベル 4) とした．

表 11: 描画速度

	volume data	skull	bonsai-1	bonsai-2	bonsai-3	tree	chest
描画速	非圧縮	9.75	8.58	7.76	3.94	3.86	0.99
度 (fps)	圧縮	12.26	9.80	11.82	5.71	14.41	5.01

これらの手法を用いることにより，並列化のみの場合に比べて，すべてのデータにおいて描画速度が向上した．特に tree では約 3.7 倍，chest では約 5 倍の描画速度が得られた．

4.7 1Gbps Ethernet を使用した結果

4.1 で述べた環境からネットワークを 1GbpsEthernet に変更した．その結果，通信時間が 85.96msec(100Mbps) から 20.22msec(1Gbps) になり，すべてのデータにおいて描画速度の向上が見られた．

また，この環境で中間画像圧縮を行った．その結果を表 12 に示す．表中の通信時間は圧縮解凍処理を含んだ時間である．

表 12: 1Gbps での通信時間

volume data	skull	bonsai-1	bonsai-2	bonsai-3	tree	chest
通信時間 (msec)	24.02	25.58	23.96	27.15	19.04	29.95
圧縮率 (%)	31.73	18.35	37.08	16.63	65.85	0.72

tree を除くすべてのデータで通信と圧縮解凍処理をあわせた時間が，圧縮せずに送った場合の通信時間を越えた．tree だけは圧縮することによってわずかに速くなった．

4.8 考察

4.8.1 並列化

グラフィクスカードのメモリサイズ以下の大きさのデータの場合，1 枚のグラフィクスカードでも高速な描画が可能であり，通信等が必要となる並列化のほうがむしろ遅くなる．しかし，並列化を行うことによって演算を各ノードで分担するとともに，扱うことのできるグラフィクスカードのメモリが増えたため，bonsai-3 以上のデータでは大きく描画速度が向上し，その効果が見られる．大規模なデータに関しては並列化は効果があると言える．

4.8.2 中間画像圧縮

skull,bonsai-2,tree では十分その効果は見られた．但し，tree ではレンダリングにかかる時間の割合が大きいため，描画速度としてはあまり向上していない．

圧縮操作はその圧縮率に依存するが，クラスタで行う圧縮に約 1〜3msec，マスタで行う解凍に約 12msec，合計で 13〜15msec の時間が必要となる．そのため，圧縮により通信時間が 15msec 程度短縮されなければ，圧縮の効果はない．通信時間は通信データ量にほぼ比例するので，圧縮率が約 17 %以下になる場合

は圧縮するべきではないと考えられる．実際，圧縮率が 17 %に近い bonsai-1 や bonsai-2 では描画速度にあまり変化はなかった．

今回扱ったような物体のボリュームデータに関しては，17 %程度の透明部分を含む場合は比較的多く見られるので，この手法が有効である可能性は高いと思われる．

4.8.3 Octree データ構造

chest に関しては Octree 構造によるボリュームデータサイズの削減により，データがグラフィクスカードに入る大きさまで圧縮することができ，描画速度が大きく向上した．その他のデータに関しても，データサイズと処理標本点数の削減の効果が現れ，描画速度は上がっている．

chest データのレベル 2 のときはまったく圧縮できていないにもかかわらず，レベル 1 と比べてほぼ 2 倍の描画速度が得られている．レベル 1 では各ノードに割り当てられた 2 つのサブボリューム (64MB) のうち 1 つしかグラフィクスカードのメモリに入れることができない．そのため 1 つのサブボリュームを処理し終えると，もう一つのサブボリュームを必ず主記憶からグラフィクスカードへ転送しなければならない．しかし，データをもう 1 段階細かく分割することにより，64MB のサブボリューム 1 個と，もう一方のサブボリュームを 8 分割したもののうちいくつかを常にグラフィクスカード内に格納しておくことができる．そのため，入れ換えるデータはレベル 1 の場合に比べて小さくなり，データ転送の時間が減り，約 2 倍の描画速度を得ることができた．

また，データ圧縮を行わなくても，データを分割することにより，描画速度が向上した．これは，データが小さく分割されたことにより，連続的にデータの一定の領域にアクセスするようになり，局所性を利用しやすくなったためだと考えられる．

これらの結果より，Octree 分割がどのようなボリュームデータに対しても有効であることが言える．

しかし，Octree 構造のデータの作成には時間がかかる．分割レベルと圧縮率にも依存するが，$256^2 \times 128$ のデータをレベル 3 まで分割するのに 270〜800msec 必要となる．Radeon9700Pro がサポートする AGP8X では 2GB/s の速度があるため，$256^2 \times 128$ のデータは約 16msec でそのままグラフィクスカードへ送られることになる．そのため，頻繁にボリュームデータが変化するような場合には適した手法ではないと言える．

4.8.4 1Gbps Ethernet

ネットワークを 1Gbps の Ethernet を用いることにより，通信時間が大幅に短縮された．

しかし，通信が高速になった分，ほとんどのデータで圧縮による通信時間の短縮よりも圧縮解凍処理の時間のほうが大きくなってしまった．結果より，このネットワーク環境下では約 66 %以上の圧縮率が得られないと，本稿で述べた手法では高速化できない．それに近い圧縮が可能な tree データではわずかに効果はあったが，これ程の高い圧縮率が得られる可能性は低いと思われるので，中間画像の圧縮はこの環境下では効果的な手法ではないと言えるだろう．

第 5 章　まとめ

本稿では汎用グラフィクスハードウェアによる並列ボリュームレンダリングの実装及び評価を行った．その結果，グラフィクスメモリの大きさを越えるようなデータを扱う場合，並列化により大きく描画速度の向上が見られた．また，ランレングスエンコーディングによる通信時間の短縮はネットワークが 100Mbps の環境では効果が見られたが，1Gbps の場合は圧縮解凍処理の時間が負担になり，その効果は得られなかった．さらに，グラフィクスカードを用いたボリュームレンダリングで，もっとも問題となるメモリの制限を解決するために，Octree 構造の分割によるデータ圧縮を行った．それにより，Octree 構造によりデータのサイズを削減するとともに，データの局所性を利用でき，さらに不要な標本点を削除することにもなり，より高速な描画が可能になった．しかし，Octree 構造によるデータ圧縮では各ノードに割り当てられるデータの大きさがそれぞれ異なる場合が多い．データによってはノードごとの負荷に大きな差が見られる可能性があるので，負荷分散も考える必要がある．また，全体に状態が分布する物理シミュレーションのようなデータに対しては，Octree 構造によるデータ分割ではあまり効果が見られないので，そのようなデータに対する圧縮法が必要となるだろう．また，実時間可視化のためには，1Gbps のネットワークを使用しても依然として通信時間は大きな問題となる．そのため，通信と中間画像の合成の並列化などが今後の課題である．

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚なる謝意を表します.

また，本研究に関して適切な御指導を賜った森眞一郎助教授，中島康彦助教授，五島正裕助手，津邑公暁助手に心から感謝致します．さらに，日頃からご助力頂いた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します.

本稿で用いた胸部のボリュームデータは，(株) ケイジーティー宮地英生氏より御提供いただいただきました．また，盆栽 (bonsai-1) のボリュームデータは University of Erlangen-Nuremberg から，盆栽 (bonsai-2) のボリュームデータは volvis から，クリスマスツリーのボリュームデータは The Computer Graphics Group XMas-Tree Project から利用させていただきました.

参考文献

[1] 中嶋正之, 藤代一成, コンピュータビジュアリゼーション, 共立出版, 2000

[2] Barthold Lichtenbelt,Randy Crane,Shaz Naqvi,Introduction To Volume rendering,Prentice Hall PTR,1998

[3] M.Levoy,Efficient ray tracing of volume data,*ACM Trans.on Graphics*,pp245-261,July 1990

[4] Klaus Engel,Thomas Ertl,Interactive High-Quarity Volume Rendering with Flexible Consumer Graphics Hardware,*Eurographics 2002*, 2002

[5] ATI Press Release-2002, http://www.ati.com/companyinfo/press/2002/4512.html, 2002

[6] Mason Woo,Jackie Neider,Tom Davis,OpenGL プログラミングガイド, ピアソンエデュケーション,1997

[7] Shuntaro Yamazaki,Kiwamu Kase,Katsushi Ikeuchi,Hardware Accelerated Volume Rendering using Standard PC Graphics Card,*RIKEN Symposium,Integrated Volume-CAD System Research*,Dec 5 2001

[8] 藤原雅宏, 五島正裕, 森眞一郎, 富田眞治, 階層格子ボリュームデータの実時間可視化, 計算機アーキテクチャハイパフォーマンスコンピューティング,1998