修士論文

汎用GPUを用いた
ボリュームレンダリングの高速化に関する
研究

指導教員　　富田 眞治 教授

京都大学大学院情報学研究科
修士課程通信情報システム専攻

篠本 雄基

平成18年2月3日

汎用 GPU を用いたボリュームレンダリングの高速化に関する研究

篠本 雄基

内容梗概

汎用 GPU を用いたボリュームレンダリングにおける代表的なアルゴリズムであるテクスチャベース法では，ボリューム参照のアクセスパタンは視点の位置により動的に決まる．アクセスパタンによっては，利用可能な参照の局所性が全くなくなり，最悪の場合，テクスチャキャッシュのヒット率が著しく低下する．

視点位置と独立にボリューム参照の空間局所性を最大化するキューボイド順レイキャスティング法が提案されている．ボリュームをキューボイドと呼ぶサブボリュームに分割し，キューボイド単位のレンダリングを行う．キューボイドをキャッシュにフェッチし，それがリプレースされるまでにキューボイド内のサンプリング点を全て抽出し処理すると，キャッシュヒット率を最大化できる．

本稿では，キューボイド順レイキャスティング法の考え方を GPU に適用したキューボイド順テクスチャベース法を提案する．GPU は，CPU と異なり，容易にアクセスパタンを変更することができない．提案手法は，テクスチャマッピングを行うスライスを分割し，キューボイド順に配置することで，アクセスパタンを制御するものである．スライスの数は，キューボイドの分割数に比例して増加する．

Radeon X800 Pro を用いて評価を行ったところ，キューボイドのサイズをテクスチャキャッシュより小さくすると，性能が低下した．そのため，テクスチャキャッシュを最大限に利用するには至っていない．キューボイドのサイズを適切に設定すると，最悪の性能が 3.2 倍に向上した．

キューボイド順テクスチャベース法の性能低下要因は，キューボイド切り替えとスライスの頂点処理のコストである．ボリュームのアドレス変換を行うことでキューボイド切り替えを不要にし，ボリュームレンダリングにおける α ブレンディング処理の一部をフラグメントプロセッサで行うことでスライス数を削減することを提案した．

Studies on the Speedup of Volume Rendering with Commodity GPUs

Yuki SHINOMOTO

Abstract

A slice-order texture-based algorithm for volume rendering with commmodity GPUs loses spacial locality of reference and suffers from low cache hit ratio at some viewpoints. This is because the access pattern of volume references depends on the position of the viewpoint.

A cuboid-order ray-casting algorithm which maximizes spatial locality of reference has been proposed. The cuboid-order algorithm divides the volume into sub volumes named cuboid, and controls the access pattern by rendering each cuboid. Maximization is achieved by detecting and sampling points in a cuboid fetched into the cache memory, before the cache lines composing the cuboid are replaced.

In this paper we propose a cuboid-order texture-based algorithm based on the cuboid-order ray-casting algorithm. CPUs cannot control the access pattern as easily as CPUs. Our algorithm controls the aceess pattern by dividing the slice into smaller slices and arranges them in the cuboid-order when rendering each cuboid. The number of slices increases in proportion to the number of cuboids.

We evaluate our algorithm with Radeon X800 Pro. The result shows that performance of the cuboid-order algorithm is lower than that of the slice-order when the size of cuboids are smaller than that of texture-cache and 3.2 times higer at some size of cuboids.

The performance of the cuboid-order algorithm suffers from the cost of processing vertices of slices and changing cuboids which are processed. In this paper we proposed the address transformation of the volume and blending with the fragment prosessor of the GPU. The former reduces the cost of chaing cuboids and the latter reduces the number of vertices.

汎用 GPU を用いたボリュームレンダリングの高速化に関する研究

目次

第1章　はじめに

ボリュームレンダリングとは，**ボリューム**と呼ばれる 3 次元の半透明なオブジェクトを直接 2 次元平面に投影する手法の総称である．ボリュームは，多くの場合，**ボクセル**と呼ばれる単位立方格子からなる．ボリュームレンダリングではボリュームを直接投影するため，ボクセルの集合を単一のポリゴンに一旦変換し，投影するポリゴンレンダリングよりも，高精度な画像を得ることができる．医療などの高精度な画像を必要とする分野において，ボリュームレンダリングは広く利用されている．

ボリュームレンダリングは，高精度な画像を得られる反面，その計算量/メモリ量が膨大である [1]．ボリュームレンダリングは，ボリューム・サイズに比例した計算量/メモリ量を必要とする．医療などの分野では，$4K^3$ ボクセルからなるボリュームをレンダリングするケースもあり [2] [3]，その場合の計算量は 6.5TFLOPS，メモリ量は 64GB にもなってしまう．

ボリュームレンダリングと GPU　ボリュームレンダリングの計算のほとんどは，カラー値・透明値に対して同時に演算が行われる．よって，ベクトル処理向けのアプリケーションと言えよう．

汎用 GPU は，ベクトルプロセッサの一種であり，カラー値・透明値に対する演算を同時に行える．また，パイプラインを複数本持つため，カラー値・透明度の組を同時に複数処理できる．さらに，GPU のグラフィクスパイプラインはボリュームレンダリングに適した構造となっている．そのため，GPU を用いたボリュームレンダリングが行われている．

GPU の性能向上とデータ供給能力　ボリュームレンダリングに要する計算量は大きいが，近年の GPU の性能向上によってこの要求は次第に満たされつつある．その一方で，ボリュームを格納するメモリのデータ供給能力の不足による性能低下が，より深刻な問題となってきている．これは，ボリュームレンダリングは，視点によってボリュームへのアクセスパタンが動的に変化し，利用可能な参照の局所性がほとんどないためである．

人間が動画像を滑らかに認識するには，視点の移動中にレンダリング速度が大きく変化しないことが重要である．そのため，アクセスパタンの変化に起因する性能低下を防ぐために，データ供給能力の改善が求められる．

CPU におけるボリュームレンダリング　CPU におけるボリュームレンダリングにおいても，データ供給能力の不足による性能低下は深刻な問題であった．この問題に対して，**キューボイド順レイキャスティング法**が提案されている [4, 5]．キューボイド順レイキャスティング法は，ボリュームへのアクセスパタンを制御して，参照の空間局所性を最大化することで，キャッシュを有効に活用する．キューボイド順レイキャスティング法によって，ランダムアクセス性能が低い DRAM をキャッシュによって補償する CPU においても，データ供給能力の低下に起因する性能低下をほとんど無視できるようになっている．

GPU のメモリアクセスの改善　そこで我々は，GPU を用いたボリュームレンダリングに対して，キューボイド順レイキャスティング法の考え方を適用した**キューボイド順テクスチャベース**法を提案する．ただし，GPU は，CPU と異なり，ストリーム型プロセッサである．そのため，ボリュームへのアクセスパタンを容易には変更できない．GPU でキューボイド順レイキャスティング法を行うには，何らかの工夫が必要となる．

以下本稿では，第 2 章で背景となる GPU を用いたボリュームレンダリングと，キューボイド順レイキャスティング法法について述べる．次に，第 3 章で提案するキューボイド順テクスチャベース法について述べる．第 4 章でキューボイド順テクスチャベースベース法を評価し，第 5 章で評価結果に対する考察を行うとともに，今後の展望について述べる．

第2章　背景

本稿で対象とするボリュームレンダリングと GPU の概要について述べる.

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リューム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.

ボリュームレンダリングは膨大な計算時間と記憶容量が必要とされ, 大型計算機や専用ハードウェアを用いる場合を除いて, リアルタイムに可視化することはこれまで困難であった. しかし，近年の CPU および GPU の性能向上に伴い，安価な PC を用いてもボリュームレンダリングに必要な計算能力を得ることができるようになりつつある.

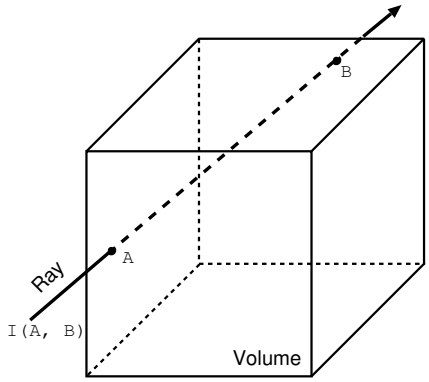

図 1: ボリュームと視線

ボリュームは，3 次元の正方格子，または，非構造格子上に定義される．本研究で扱う正方格子ボリュームでは，格子上の単位立方体をボクセルと呼び，各ボクセルが色と透明度を持つ.

ボリュームレンダリングは，オブジェクトであるボリューム自身が発光して

いると考えると，その原理を理解しやすい．一般的なコンピュータ・グラフィクス，例えばポリゴンのサーフィスレンダリングやレイトレーシングなどでは，オブジェクト以外にも 1 つ以上の光源が必要となる．一方，ボリュームレンダリングではボリューム外部に光源があるのではなく，ボリューム自身が光源となる．ボリューム内の任意の点は点光源として発光し，その光はピクセルに向かって半透明なボリューム内を進むうちに減衰していく．

図 1 ボリュームを貫通する 1 本の視線を示す．視線がボリュームに入る点を A，ボリュームから視線が出る点を B とする．このときのピクセル値 $I(A,B)$ を，

$$I(A,B) = \int_A^B g(s) e^{-\int_A^s \tau(x)dx} ds \tag{1}$$

と定義する．ここで，s, x は視線上の位置を表す変数であり，g はボリュームの光の強度，τ は減衰係数を表す．

計算機上でボリュームレンダリングを実現するには，離散化する必要がある．ボリュームについては，ボクセルと呼ばれる単位立方格子で構成されることになる．あるボクセル内の光の強度 g は均質化されており，ボクセル内の点 p における $g(p)$ を求めると，p に関わらず同じ値が得られる．つまり，図 1 のボリューム内で g は連続的であったが，図 2 に示すようなボクセルで構成されたボリュームにおいては，g は離散的である．

また，ピクセル値計算も離散化される．式 1 の積分を離散化すると，視線上で等間隔にボリュームをサンプリングし，それの総和を求めることになる．離散化された式は，

$$I(A,B) = \sum_{i=A}^{B} g(s_i) e^{-\sum_{j=A}^{s_j} \tau(x_j)} \tag{2}$$

となる．なお，実装に際してはより簡単な近似式を用いることが一般的である．これについては次小節で説明する．

2.1.1 レイキャスティング法

離散化されたボリュームレンダリングでは，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．

ボリュームレンダリングのアルゴリズムとして，レイキャスティング法がよく用いられる．レイキャスティング法では, 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 2). 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(赤，緑，青) 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

$$C = \sum_{i=0}^{n} \alpha(v_i) c(v_i) \prod_{j=0}^{i-1} (1 - \alpha(v_j)) \tag{3}$$

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.

$$C_{k-1} = \alpha(v_{k-1}) c(v_{k-1}) + (1 - \alpha(v_{k-1})) C_k \tag{4}$$

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.

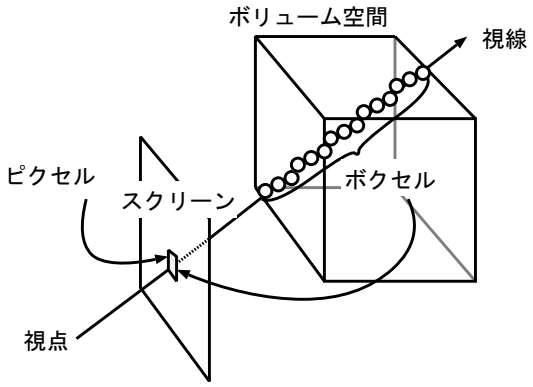

図 2: レイキャスティング法

2.2 汎用 GPU

GPU とは，ディスプレイ画面への画像表示機能を持った拡張カードに搭載されるグラフィクス処理用プロセッサである．多くの GPU は，ディスプレイ画面への画像表示機能に加えて，3 次元グラフィクス処理を高速に行うハードウェア

を搭載している．

GPU のうち，パーソナルコンピュータに搭載される比較的安価な物を汎用 GPU と呼ぶ．汎用 GPU の例として，NVIDIA 社の GeForce，ATI 社の Radeon，Matrox 社の Parhelia などが挙げられる．

2.2.1 GPU の構成

多くの GPU は，以下の構成を取る．

頂点プロセッサ (Vertex Processor) 頂点プロセッサは，ポリゴン幾何モデルを構成する頂点を処理する．頂点データには座標だけでなく，法線ベクトル，テクスチャ座標，RGBα カラー，材質特性も含まれている．

幾何モデルと光源を 3 次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する．これらの処理を，ジオメオリ変換および光源処理と呼ぶ．近年，この計算を，ポリゴン幾何モデルの頂点情報を幾何モデルそのものを変更することなく，プログラマブルに行うことが可能になっている．

トライアングルセットアップエンジン (Triangle Setup Engine) 頂点プロセッサがジオメトリ変換を行った頂点から，3 頂点のグループごとに三角形を求める．

ラスタライザ (rasterizer) トラインアングルセットアップによって作成された三角形を，画素の集まりに分解する．画素に分解する操作をラスタライズと呼び，分解された画素をフラグメントと呼ぶ．フラグメントは，単一のピクセルに対応し，フラグメントの処理結果がピクセルとなる．ラスタライザは固定機能のユニットであり，ラスタライズのアルゴリズムは GPU 毎に異なっている．

フラグメントプロセッサ (Fragment Processor) フラグメントに対して処理を行い，画素値を計算する．後述するテクスチャマッピングは，ここで行われる．かつては固定機能のユニットであったが，近年，フラグメントプロセッサの処理をプログラムすることが可能になっている．

テクスチャユニット (texture unit) テクスチャ(Texture) とは，物体の表面の質感を表現するためにポリゴンに貼り付ける画像である。テクスチャの 1 要素をテクセルと呼ぶ．

テクスチャユニットは，テクスチャ操作を行う固定機能のユニットである．フラグメントプロセッサの要求に応じて，ビデオメモリからテクスチャの

読み出しを行う．テクスチャが拡大あるいは縮小される場合，テクセル値の補間を行う．

テクスチャキャッシュ(Texture Cache)　テクスチャユニットは，テクスチャを高速に読み出すために，読み取り専用のキャッシュであるテクスチャキャッシュを持つ．テクスチャキャッシュの多くは，2次元テクスチャへのアクセスに最適化されている．また，一部のGPU[6]は，1次テクスチャキャッシュに加えて，より大きな容量を持つ2次テクスチャキャッシュを持つ．2次テクスチャキャッシュは，複数のテクスチャユニットで共有される．テクスチャキャッシュの容量は，CPUのそれと比較して小さい．

また，CPUのキャッシュと異なり，容量やラインサイズといった仕様が公開されていない．

ピクセルユニット (Pixel Unit)　ラスタユニット (Raster Unit) とも呼ばれる．フラグメントプロセッサによって処理されたフラグメントの後処理を行う．後述するαブレンディングは，ピクセルユニットで行われる．

フレームバッファ(Frame Buffer)　ディスプレイの内容を保持する．全ての処理が終了したピクセルはフレームバッファに書き込まれる．フレームバッファ用の領域は，ビデオメモリ内に確保される．

RAMDAC (Random Access Memory Digital-to-Analog Converter)　フレームバッファのデジタルデータを，アナログ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．

ビデオメモリ (Video Memory)　レンダリングパイプラインが処理する頂点やテクスチャデータを格納する場所である．また，フレームバッファとしても使用される．

ビデオメモリは，GDDRと呼ばれるメモリが用いられている．GDDRは，メインメモリで用いられるDRAMと同様のテクノロジを用いている．ビデオメモリはGPUにシステムバス等を介することなく直接接続されるため，メインメモリと比較すると，高いクロック周波数で動作し，広いメモリ帯域を持つ．市販のGPUに搭載されるビデオメモリの容量は，最大でも512MBであり，メインメモリの容量と比較すると少ない．

2.2.2　グラフィクスパイプライン

頂点プロセッサ，フラグメントプロセッサ，その他の非プログラマブルなユニットおよびアプリケーションは，すべてデータフローによってリンクされる．

GPU は，レンダリング処理をいくつかのステージに分割し，パイプライン処理を行う．図 3 に GPU のグラフィクスパイプラインを示す．

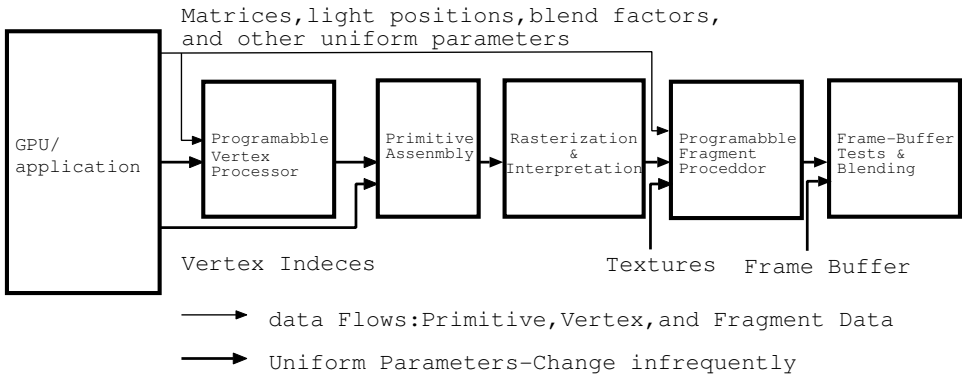

図 3: グラフィクスパイプライン

CPU と GPU は，サーバ/クライアントの関係となっている．クライアントである CPU は，サーバである GPU にレンダリング命令を送信する．GPU は，レンダリング命令を解釈し，レンダリング処理を行う．

パイプラインは，CPU から送信された入力で開始され，フレームバッファに描かれるピクセルで終了する．CPU からの送信データは，コマンド，テクスチャ，頂点データである．テクスチャはビデオメモリに保存される．CPU から受け取ったコマンドから，命令のストリームが生成され，頂点データは頂点プロセッサに渡される．頂点プロセッサは，前述のように頂点データのジオメトリ変換を行う．次にトライアングルセットアップエンジンが 3 頂点のグループごとに三角形を求める．この三角形からラスタライザがフラグメントのストリームを生成する．フラグメントプロセッサはフラグメントを処理単位とし，テクスチャマッピングや RGBα 値の計算を行う．ピクセルユニットがフレームバッファに存在するピクセルの値との α ブレンディングを行い，最後にフレームバッファにピクセル値が書き込まれ，ディスプレイに出力される．

2.2.3 GPU のプログラミングモデル

GPU のプログラミングには，一般にグラフィクス API が用いられる．グラフィクス API (Graphics API) とは，CPU から GPU にアクセスするために呼び出す関数群である．プログラマはグラフィクス API を用いることで，容易に GPU の機能を利用することができる．グラフィクス API の例として，Windows

環境で用いられる DirectX[7]，OS に依存しない OpenGL[8] などが挙げられる．

グラフィクス API により，レンダリングするスクリーンのサイズ，視点の位置，平行投影・透視投影のどちらを用いるか，といった初期化に関する指定を GPU に与える．続いて図形のレンダリング命令を送信する．OpenGL における四角形のレンダリング命令を以下に示す．

```
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_QUADS);
glVertex3f(-1.0, -1.0, 0.0);
glVertex3f( 1.0, -1.0, 0.0);
glVertex3f( 1.0,  1.0, 0.0);
glVertex3f( -1.0, 1.0, 0.0);
glEnd();
```

このように，グラフィクス API を用いて GPU に四角形をレンダリングさせる場合は，四角形をレンダリングする命令とあわせて，四角形の頂点座標，頂点が持つ色情報などを送信する．上記の例では，全ての頂点色に白が設定され，四角形の内部は白で塗りつぶされる．頂点座標の指定方法は，図形のレンダリング時に直接指定する方法と，頂点を列挙した配列である頂点配列を予め用意し，頂点配列のインデックスを指定する方法の 2 つがある．頂点配列は，メインメモリまたはビデオメモリに保存される．また，レンダリング命令そのものをビデオメモリに保存することで CPU-GPU 間のデータ転送量を削減するディスプレイ・リストと呼ばれる機能も GPU には備わっている．

2.2.4 上位レベルシェーダ言語

頂点プロセッサおよびフラグメントプロセッサのプログラミングも，グラフィクス API を通して行う．頂点プロセッサで実行されるプログラムを頂点プログラムと呼び，フラグメントプロセッサで実行されるプログラムをフラグメントプログラムと呼ぶ．

頂点プログラムおよびフラグメントプログラムは，頂点毎，フラグメント毎に繰り返し呼び出される．そのため，高速化のためにビデオメモリに保存されることが多い．

両プログラムはアセンブリ言語の形式をとっており，従来プログラミングが複雑であった．

近年，Cg (C for Graphics)，HLSL (High Level Shader Language)，GLSL

(OpenGL Shading Language) などの GPU プログラミング用の高級言語が NVIDIA 社，microsoft 社，OpenGL ARB によって開発されている．これらの言語は，CPU における高級言語がアセンブリコードにコンパイルされるのと同様に，専用のコンパイラを用いて前述のアセンブリ言語にコンパイルされる．コンパイルされるシェーダ言語の種類は，オプションで指定できる．そのため，使用するグラフィックスカードの機能やグラフィクス API に合わせたシェーダ言語を選択できる．

2.2.5 GPU が行う処理の特徴

GPU の頂点プロセッサおよびフラグメントプロセッサが行う処理は，以下のような特徴を持つ．

- 各パイプラインは，命令のオペランドと結果の格納先が，GPU 内のレジスタである．
- ピクセルの色情報である RGBα は，1 ベクトルとして 1 命令でまとめて演算できる．
- RGBα の値を入れ替えて，新しいベクトルを作ることができる．例えば，RRRR や BGG といったベクトルを作ることができる．なおこの操作は計算コストを生じさせない．
- パイプラインにより，複数の頂点およびフラグメントが同時に処理される．
- 全ての頂点およびフラグメントに対して，同一の計算を施す．
- 異なる頂点およびフラグメントの演算は，相互に干渉できない．

これらの特徴は，レジスタ-レジスタ型ベクトル計算機のベクトルプロセッサと類似しており，GPU はベクトルプロセッサの一種であるといえる．GPU がベクトルプロセッサと異なる点として，以下があげられる．

- 頂点プロセッサ，フラグメントプロセッサという 2 つのベクトルプロセッサを持つ．
- あるパイプラインの計算結果を，他のパイプラインの入力にする技法をチェイニングと呼ぶ．GPU は自動的に頂点プロセッサの計算結果が，フラグメントプロセッサの入力にチェイニングされるが，チェイニングするパイプラインを任意に選択することはできない．
- 演算結果はフレームバッファに画像として出力される．
- 出力画像から読み出せる情報は，ピクセルの RGBα の値のみであり，プログラムの計算結果の出力に制限がある．

2.2.6 GPU の機能拡張

GPU は，グラフィクス API の機能拡張に対応する形で，機能を向上させてきた．

DirectX を例に取ると，DirectX 7 以前の GPU は，固定された機能しか持っておらず，主要な CG 処理は,Hardware T&L (Hardware Transformation and Lighting) とよばれる座標変換と，ライティング処理であった．

DirectX 8 対応の GPU において，より複雑な CG 処理を行うために，プログラマブルパイプラインの概念が導入された．プログラマブルパイプラインは，プログラムすることでパイプラインの内容を変更することができる．

現在，パイプライン中でプログラム可能なユニットは，頂点プロセッサとフラグメントプロセッサの 2 つである．頂点プロセッサでは，各頂点の処理を変更できる頂点プログラム (vertex program) を利用でき，フラグメントプロセッサでは，各フラグメントの処理を変更できるフラグメントプログラム (fragment program) を利用できる．

DirectX 9 対応の GPU は，プログラマブルパイプラインの機能が強化され，より汎用的な計算を行えるようになった．また，頂点プロセッサにおいて動的分岐をサポートするようになった．ただし，フラグメントプロセッサは，動的分岐をサポートしておらず，静的分岐のみを使用できた．静的分岐とは，両方の分岐結果を実行し，結果の片方をマスクすることで分岐を実現することを意味する．

DirectX 9c 対応の GPU は，フラグメントプロセッサにおいても動的分岐を行えるようになった．

計算精度の面でも，GPU の性能は向上している．かつての GPU は，固定小数点の精度でしか演算を行えなかった．今日の GPU は，頂点プロセッサおよびフラグメントプロセッサにおいて，単精度浮動小数点の精度で演算が行えるようになっている．

このように，GPU のプログラミングに関する制約は徐々に取り払われつつある．

そのため，近年，GPU を汎用のベクトルプロセッサとして捉え，グラフィクス処理以外の計算用途に用いる GPGPU (General-Purpose computation on GPUs) と呼ばれる研究が一部で行われている [9]．

2.3 Texture Based Volume Rendering

GPU によるボリュームレンダリングでは，テクスチャマッピングを利用したアルゴリズムが用いられる [10, 11]．本稿では，このアルゴリズムをテクスチャベース法と呼ぶ．テクスチャベースと，そのアクセスパタンについて説明する．

2.3.1 テクスチャベース法

テクスチャベース法は，以下の順序で処理が行われる．

1. ボリュームをテクスチャに変換し，ビデオメモリに保持する．
2. 視線に対して垂直な四面体ポリゴンを用意する．
3. 視点の変更とクリッピングを行う．
4. 四面体ポリゴンにテクスチャの断面をマッピングし，ボクセル値を計算する．
5. マッピングが行われたスライスを, 視点から遠い順に順次 α ブレンディングする

ボリュームは，CPU のメインメモリから GPU のビデオメモリにグラフィクス API を介して転送され，テクスチャとして保持されている．以後，ボリュームを格納したテクスチャをボリュームテクスチャと呼ぶ．視線に対して垂直な四面体ポリゴンをスライスと呼ぶ．スライスにボリュームテクスチャの断面をマッピングすることで，ボリュームを視線に対して垂直な断面の集まりとして再構成する．ボリュームテクスチャの断面が貼り付けられたスライスを, 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 (図 4)．この手法を用いることで, GPU の機能を利用した高速処理が可能である [12]．以下，個々の処理について説明する．

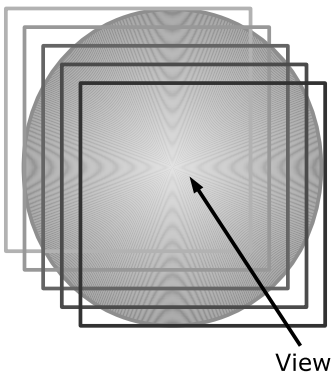

図 4: テクスチャベース法

ボリュームのテクスチャへの変換　テクスチャは，カラーデータ，輝度データといった，データの多次元配列である．テクスチャ配列内の個々のデータをテクセルと呼ぶ.

GPU は，ボリュームをテクスチャとして保持する．GPU が 3 次元テクスチャをサポートしている場合は，ボリュームを 3 次元テクスチャとして扱う方法が可能である (図 5). ボリュームは，CPU のメインメモリから GPU のビデオメモリにグラフィクス API を介して転送され，3 次元テクスチャとして保持される.

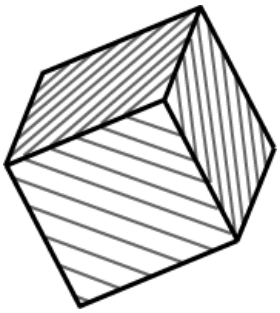

図 5: 3 次元テクスチャ

GPU が 3 次元テクスチャをサポートしておらず，2 次元テクスチャしか使用できない場合は，ボリュームをスライスの重ね合わせで表現し，複数枚の 2D テクスチャとして扱う．今日の GPU のほとんどは 3 次元テクスチャをサポートしているため，本稿では 3 次元テクスチャを用いる場合についてのみ述べる.

スライスの設定　スライスが配置される 3 次元空間の座標系を，ワールド座標系と呼ぶ．画像が投影されるスクリーンの座標系を，ウインドウ座標系と呼ぶ. まず，スクリーンの中心と，ワールド座標系の原点を一致させる.

次に，スライスの大きさとピクセルの対応関係をわかり易くするために，長さが 1×1 の正方形スライスがスクリーン上の 1 ピクセルを占めるように視体積を設定する.

次にスライスの頂点を設定する．視点が Z 軸上にあり，視線方向がワールド座標系の原点とすると，全てのスライスの中心が Z 軸上にあるようにスライスを配置する．各スライスは，Z 軸上に等間隔で配置される．スライスの間隔が，レイキャスティング法におけるボリュームのサンプリング間隔に相当する.

テクスチャベース法では，正しいレンダリング結果を得るためには，スライスは視線に対して常に垂直である必要がある．これは，スライス自体を回転さ

せると，スライスと視線の成す角度が小さくなるにつれて投影面積が小さくなり，サンプリングされるテクセルの数が減少するためである．スライスが視線に対して平行になると，投影されたスライスは1本の線になる．この場合，スクリーンの各ピクセルにつき1回しかボリュームテクスチャのサンプリングが行われず，ボリュームレンダリングの意味を成さなくなる．

クリッピング　全てのテクセルをサンプリングするために必要なスライスの枚数は，視点によって動的に変化する．確実に全てのボリュームをサンプリングするために，ボリュームの対角線の最大値をスライスの枚数とする．スクリーンの大きさは，ボリュームの投影面積の最大値以上とする．

GPUの処理はストリーム型であるため，スライス内に存在するフラグメントは全て処理される．このままレンダリングを行うと，実際にはテクスチャマッピングを行う必要がない部分も処理が行われ，大幅にレンダリング速度が低下する．

視点変更に応じて，CPUで頂点を動的に計算することもできるが，CPU側の計算が完了しないとGPUへ頂点情報が送信されないため，GPUで計算することが望ましい．

GPUには，レンダリング領域を制限するクリッピングという機能が備わっている．クリッピングは，クリップ平面を指定することで実行される．クリップ平面を6面設定すると，クリップ平面によって囲まれる空間のみがレンダリング対象となる．クリップ平面によって囲まれる空間の外に位置するポリゴンは，レンダリングが省略される．クリップ平面と交差するスライスのみがレンダリング対象となり，クリップ平面とスライスの交点が，スライスの新たな頂点となる．対応するテクスチャ座標は，GPUにより再計算される．

このクリップ平面を視点変更に応じて回転させることで，レンダリング領域を常にボリュームテクスチャの体積に等しくすることができる．(図6).

観察視点の変更　観察視点の変更は，ボリュームを回転することによってなされる．前述のように，スライスは視線に対して常に垂直である必要がある．そのため，観察する視点を変更する際は，スライス座標系は回転しない，スライスの頂点に対応するテクスチャ座標と，クリップ平面をそれぞれ回転させることにより，観察視点を変更した画像が得られる．

ボリュームをスクリーンの中央に配置する場合を考える．この場合，観察視点の変更は，スクリーンの中央にあたるワールド座標系の原点を中心に，テク

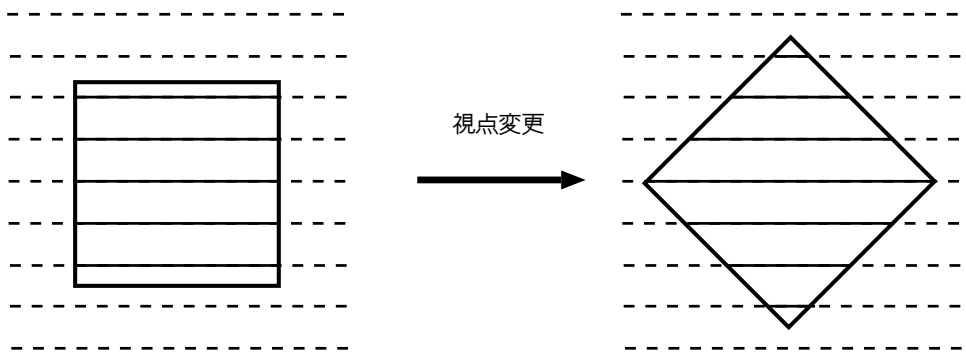

図 6: テクスチャ座標とクリップ平面の回転

スチャ座標およびクリップ平面を回転させることによってなされる．クリップ平面の座標はワールド座標系にあるため，クリップ平面の回転時における中心はワールド座標系の原点である．一方，テクスチャ座標の範囲は [0,0, 1.0] であるため，ワールド座標系の原点に対応するテクスチャ座標は (0.5, 0.5, 0.5) である．そのため，このまま回転させるとクリップ平面の回転とのずれが生じる．クリップ平面の回転をテクスチャ座標の回転を一致させるため，まず座標系を平行移動し，ワールド座標の原点とテクスチャ座標系の原点を一致させる．その後テクスチャ座標系を回転し，先ほど平行移動した方向と逆の方向に座標系を平行移動し，クリップ平面の中心に戻る．

テクスチャマッピング　ポリゴンの表面にテクスチャを貼り付ける操作を，テクスチャマッピングと呼ぶ．

テクスチャマッピングを行う際は，ポリゴンの各頂点に，読み出されるテクセルのテクスチャ内での位置を表すテクスチャ座標を対応させる．頂点座標とテクスチャ座標の関係は，ワールド座標系とテクスチャ座標系を一致させる作業にあたる．

図 7 では，100×100 ピクセルのポリゴンに同サイズのテクスチャをマッピングしている．ポリゴンとテクスチャのサイズが異なる場合は，テクスチャは拡大あるいは縮小されてマッピングされる．

テクスチャ座標は通常，テクスチャのサイズに関わらず，[0, 1] の範囲に正規化して指定する点に注意が必要である．この範囲を超える座標を与えた時にどのように処理するかは，パラメータとして GPU に与える．各フラグメントのテクスチャ座標は，ラスタライズの過程で線形補間される．

テクスチャベース法においては，スライスの Z 座標の値に合わせてボリュー

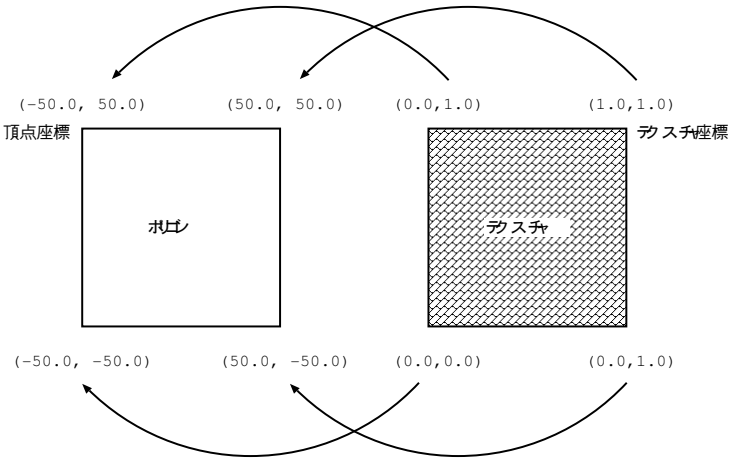

図 7: 頂点座標とテクスチャ座標の対応

ムテクスチャの Z 座標の値を設定する．スライスの位置に合わせて，マッピングされるボリュームテクスチャの断面が変化することになる．

ボクセル値の計算　ボリューム値はスカラー値であるため，そのままでは画像として観察しにくい．ボリュームレンダリングを行う際は通常，ボリューム値を伝達関数を用いて RGBα のチャネルを持つボクセル値に変換する．

フラグメントプロセッサが固定機能しか持っていない GPU を用いたボリュームレンダリングでは，CPU であらかじめボリューム値に伝達関数をかけてボクセル値に変換する．ボクセル値に変換された 3 次元配列を，GPU は 3 次元テクスチャとして保持する．この時，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．そのため，ビデオメモリの搭載量が限られている GPU では，これまで大規模なボリュームを可視化することができなかった．

しかし，この欠点は解消されつつある．GPU の機能拡張により，伝達関数によりあらかじめボクセル値を求めるのではなく，GPU 内部でボクセル値を計算することが可能になったためである．

GPU 内部でボクセル値を計算する手法として，Dependent Texture (依存テクスチャ) を利用する方式 [13] と，フラグメントプログラムを利用する方式の 2 つがある．

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．まず，ボリューム値そのものを保持する 3 次元テクスチャを用意する．加えて，ボリューム値と，RGBα 要素を持つボクセル値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 次元テクスチャをサンプリングしたボリューム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ボクセル値を得ることができる (図 8)．

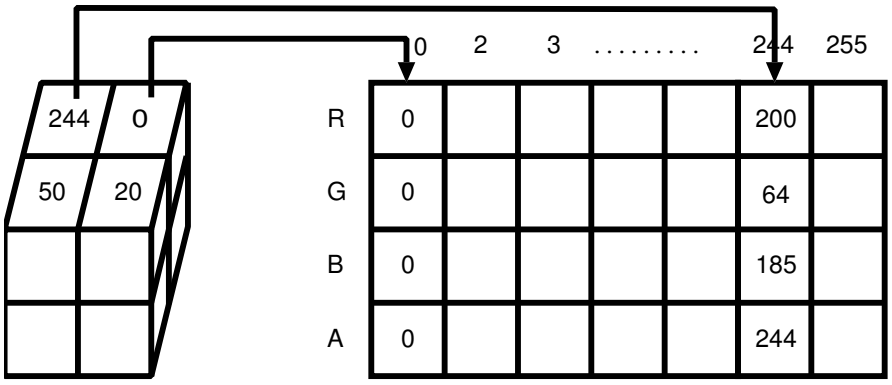

図 8: 依存テクスチャ

ボクセル値を持つ 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．欠点として，2 種類のテクスチャを交互にサンプリングするため，テクスチャキャッシュのリプレースが起こる可能性がある点が挙げられる．

フラグメントプログラムを用いる手法では，3 次元テクスチャからサンプリングしたボクセル値に対して，フラグメントプログラム内で伝達関数に相当する計算を行うことでボクセル値を得る．伝達関数の変更は，使用するフラグメントプログラムを変更することによって行われる．

依存テクスチャを用いる場合と比較すると，フラグメントプログラムの命令数が増加するという欠点がある．利点として，1 種類のテクスチャしか用いないため，テクスチャキャッシュの利用効率が上がる点が挙げられる．

グラフィクスパイプラインとの対応　テクスチャベース法は，GPU のグラフィクスパイプラインに適したアルゴリズムである．

まず，頂点プロセッサで，クリップ平面の計算・スライスの頂点座標のクリッピング・テクスチャ座標の回転が計算される．次に，スライスがラスタライズ

され，フラグメントに分解される．続いて，フラグメントプロセッサではテクスチャマッピングが行われる．ピクセルユニットで α ブレンディングが行われ，最終的なレンダリング結果がフレームバッファに書き込まれ，ディスプレイに出力される．これら一連の処理がパイプライン処理される．

2.3.2 テクスチャベース法のアクセスパタン

テクスチャベース法のボリュームへのアクセスパタンは，レイキャスティング法と同じく，視点位置と視線方向に大きく依存する．レイキャスティング法は，ピクセル順に処理を行う．ピクセル順とは，スクリーン上のピクセルを 1 つずつ逐次的に処理することを表している．一方で，テクスチャベース法はスライス順に処理を行う．スライス順とは，スライス内のフラグメントを全て処理してから，次のスライスを処理することを表している．レイキャスティング法では視線方向にボリュームがサンプリングされるのに対して，テクスチャベース法では視線方向に垂直な方向にボリュームテクスチャがサンプリングされる．スライス内でのサンプリングパタンは，ラスタイライズのパタンに一致する．2 つの手法のアクセスパタンは直交する．

テクスチャベース法もレイキャスティング法と同様に，アクセスパタンによっては，テクスチャキャッシュの利用効率が低下する．第 4 章で述べるように，最悪の場合でレンダリング速度が約 1/8 まで低下する．テクスチャベース法でキャッシュヒット率が最良となるのは，スライスとボリュームテクスチャのビデオメモリ内でのアドレスの並びが平行になる場合である．キャッシュヒット率が最悪となる場合は，スライスに対してアドレスの並びが垂直になる場合である．テクスチャベース法における最良の視点が，レイキャスティング法における最悪の視点に対応することになる．

GPU におけるメモリアクセスの重要性　CPU における RC 法は，本来は通常の数値処理と比較してメモリに対する要求の低い，計算バウンドな処理である [4]．一方で，GPU を用いたテクスチャベース法では，テクスチャマッピングと α ブレンディングが，パイプライン処理される．そのため，テクスチャマッピングに要する時間がレンダリングのスループットに影響する割合が CPU と比較して大きい．

GPU はテクスチャキャッシュを持つことで，テクスチャへの高速なアクセスを実現している．このような状況下では，テクスチャキャッシュに存在しないテクセルをビデオメモリから読み出すことは，できる限り避けるべきである．

テクスチャキャッシュの多くは，2次元テクスチャのアクセスに最適化されており，3次元テクスチャのランダムアクセスは考慮されていない．これは，ゲームやCGではサーフィスレンダリングが主であり，2次元テクスチャの使用頻度が高いためである．アクセスパタンが視点により動的に変化するテクスチャベース法に対しては，テクスチャキャッシュも有効に働かない．メモリアクセスの最適化が，GPUにおいても求められる．

2.4 キューボイド順レイキャスティング法

CPUを対象として，空間の参照局所性を最大化するボリュームレンダリング・アルゴリズムであるキューボイド順レイキャスティング法が提案されている．

通常の数値処理の中には，タイリング[14, 15]などの技法によって，参照の局所性を高められるものがある．参照の局所性を高められれば，現存するパーソナルコンピュータでも，キャッシュによって高いバンド幅を提供することができる．

キューボイド順レイキャスティング法は，タイリングと同様の考え方によって，レイキャスティング法におけるボリュームへのアクセスパタンを制御するものである．ただしレイキャスティング法では，視点の移動にともなって制御の対象となるアクセスパタンそのものが変わるため，通常の数値処理に対するように，コードを静的に変換することはできない．

ボリュームを複数の直方体，キューボイドと呼ぶサブボリュームに分割し，各キューボイドを順に処理することで画像を得る(図9)．キューボイドのサイズをキャッシュのそれより小さくして，1つのキューボイドの全体がキャッシュに乗るようにする．そして1つのキューボイドに内在するサンプリング点のみを一気に処理すると，キャッシュラインのリプレースは発生せず，その結果キャッシュヒット率を最大化することができる．

キューボイド順レイキャスティング法では，視点の位置に関わらず参照の空間局所性を最大化することができるため，ランダムアクセス性能が低いDRAMをキャッシュによって保証する今日の汎用プラットフォームであっても，メモリアクセスに起因する速度低下がほとんど無視できるようになる．

従来方式では，最悪の場合，最良の場合の約6倍もの描画時間がかかっていたものが，提案方式では，視点の位置によらず，従来方式の最良の場合の1.15倍の時間で描画できたと報告されている[5]．

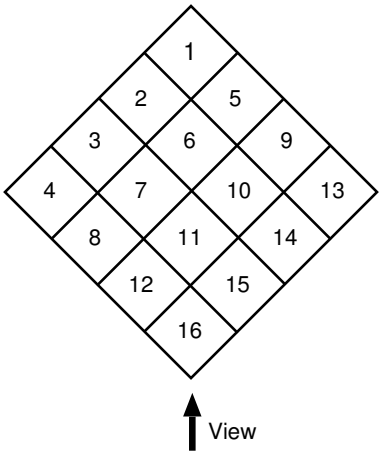

図 9: キューボイド順

CPU と GPU の処理速度　文献 [4] では，Itanium2 サーバを用いて，128^3 のサイズのボリュームを最良の場合に 9FPS の速度でレンダリングしている．一方，後ほど第 4 章で述べるように，Radeon X800 Pro により 512^3 のサイズのボリュームレンダリングした場合，提案手法を用いなくても最悪の場合に 1.8FPS の速度でレンダリングできている．レンダリング速度より，Radeon X800 Pro は Itanium2 の 12.8 倍高速である．ボリュームレンダリングという用途では，GPU は CPU に対して優位にあるといえる．

2.5　第 2 章のまとめ

本章では，ボリュームレンダリングの概要と，一般的に用いられるアルゴリズムであるレイキャスティング法について述べた．

本稿で対象とするのは，汎用 GPU を用いたボリュームレンダリングである．GPU の概要と，GPU で一般的に用いられるアルゴリズムであるテクスチャベース法について述べた．

レイキャスティング法・テクスチャベース法の双方ともに，ボリュームへのアクセスパタンは視点位置と視線方向に大きく依存する．GPU においては，視点によってテクスチャキャッシュの利用効率が低下し，第 4 章で述べるように，最悪の場合でレンダリング速度が約 1/8 まで低下する．

CPU を対象として，参照の局所性を最大化するボリュームレンダリング・アルゴリズムであるキューボイド順レイキャスティング法が提案されている．キューボイド順レイキャスティング法はボリュームへのアクセスパタンを制御するも

のである．

次章では，キューボイド順レイキャスティング法の考え方に基づいたボリュームレンダリング・アルゴリズムであるキューボイド順テクスチャベース法を提案する．キューボイド順テクスチャベース法を用いることで，テクスチャキャッシュの利用効率を最大化することができる．

第3章　キューボイド順テクスチャベース法

本章では，キューボイド順レイキャスティング法の考え方をGPUに適用したボリュームレンダリング・アルゴリズムであるキューボイド順テクスチャベース法を提案する．

キューボイド順テクスチャベース法は，キューボイド順レイキャスティング法の考え方を用い，参照の局所性を最大化することを目的とする．参照の局所性を最大化することで，テクスチャキャッシュを最大限に利用することができる．

ボリュームテクスチャは，キューボイド (Cuboid:直方体) と呼ぶサブテクスチャに分割される．ここで，キューボイドは，テクスチャキャッシュのサイズよりも小さい．

処理はキューボイド順で行う．キューボイド順とは，キューボイド内のテクセルを全てサンプリングするような順序である．キューボイド順でレンダリングすると，キューボイドはテクスチャキャッシュのサイズよりも小さいため，どの方向からレンダリングしてもテクスチャキャッシュがリプレースされることがなくなる．その結果，GPUの処理能力を最大限に発揮することができる．

キューボイドおよびキューボイド順の定義は，キューボイド順レイキャスティング法のそれと同様である．キューボイド順テクスチャベース法とキューボイド順レイキャスティング法の主な相異点は，キューボイドの処理部分とアクセスパタンの制御の実装方法である．また，キューボイドをテクスチャとして扱うことに起因する，さまざまな制限が存在する．

キューボイド順テクスチャベース法を実装するにあたっては，以下に挙げる問題がある．

テクスチャキャッシュ　GPUが持つテクスチャキャッシュは，容量やラインサイズの仕様が公開されていない．そのため，テクスチャキャッシュに乗るキューボイドのサイズを前もって求めることができない．

また，テクスチャキャッシュのサイズは，CPUのキャッシュと比較して小さいため，CPUと比較して，キューボイドのサイズをより小さくしなくてはならない．キューボイド分割に伴うオーバヘッドがある場合，キューボイドのサイズが小さくなることでその影響がより大きくなる可能性がある．

アクセスパタンの制御　キューボイド順にレンダリングを行う目的は，テクスチャのアクセスパタンを制御することである．テクスチャベースでは，ボリュー

ムを覆うサイズのスライスを用意し，それにテクスチャマッピングを行う．GPU はストリーム型の処理を行う．ラスタライズの結果，ポリゴンの内側にあるフラグメントは全て処理される．フラグメントの処理を途中で中断して，次のスライスに移るといったことはできない．

GPU は，ループを簡単に変更できる CPU と異なり，アクセスパタンを容易には変更できない．そのため，アクセスパタンを制御するために特別な方策が必要となる．

前者の問題については，第 4 章における予備評価で，GPU が持つテクスチャキャッシュのサイズの推定を行う．後者の問題については，キューボイドの分割数に合わせてスライスを分割し，キューボイド順に配置することでアクセスパタンを制御することで対処する．

以降，本章では，GPU におけるキューボイドの分割方法，アクセスパタンの制御方法，キューボイド内の処理方法について説明する．

3.1 キューボイドの分割

GPU は，ボリュームテクスチャを GPU 内部で分割して，新たなサブテクスチャを作る機能を備えていない．そのため，CPU が予めボリュームをサブボリュームに分割した上で，個々のサブボリュームをテクスチャとする．本稿では，このサブテクスチャをキューボイドと呼ぶ．

3.1.1 キューボイドの形状

GPU によっては，サイズが 2 の巾乗でないテクスチャを扱うことができない．また，サイズが 2 の巾乗でないテクスチャを扱える GPU であっても，2 の巾乗であるテクスチャを用いた方が高速に処理を行える．そのため，キューボイドのサイズは 2 の巾乗となるようにする．本稿では同様の理由により，ボリュームのサイズも 2 の巾乗とする．

キューボイドの形状は，立方体となるように分割する．立方体にすることで，視点変更によるキューボイドの投影面積の変化が少なくなるとともに，後述するスライス枚数の変動が少なくなる．

3.1.2 テクスチャの切り替え

フラグメントプロセッサは，アクセスするテクスチャをプログラム実行時に動的に指定することができない．そのため，使用するテクスチャは，CPU によって指定される．各キューボイドテクスチャには，個別の ID を与える．テクス

チャマッピングを行う際は，CPU が，テクスチャの ID を GPU に送信することで，使用するテクスチャを切り替えることができる．ID は，各軸方向のキューボイド番号を連結したものとする．

3.2 キューボイド順

前述のように，キューボイド順テクスチャベース法のキューボイド順の定義は，キューボイド順レイキャスティング法と同様である．処理順の決定は，GPU ではなく，CPU で行う．これは，第 2 章で述べたように，テクスチャの切り替えは CPU から行わなくてはならないためである．また，キューボイドのレンダリング命令は CPU からグラフィクス API を介して行い，GPU 側でキューボイドの配置される位置を決定できないからでもある．

各キューボイドは，スクリーン奥にあるものから順に処理しなくてはならない．順序の決定は，距離を計算するなどの複雑な計算は必要なく，x, y, z の各軸ごとのループによって実現できる．スクリーン奥にあるキューボイドから順に選択するコードを以下に示す．

```
int x, y, z;
void loop_x(void){
   for(x = 0; x < cx; ++x)
      loop_y();
   for(x = X_MAX; x > cx; --x)
      loop_y();
   x = cx;
      loop_y();
}
```

ここで，X_MAX はボリュームの x 軸方向の分割数から 1 を引いた数である．図 10 に示す 2 次元のボリュームを例に，上記のコードの動きを説明する．コード中 cx は視点の x 座標を含むキューボイドの x 軸方向の番号で，図 10 では 2 である．同様に cy は $\mathrm{cy} > 3$ である．

最外側のループ loop_x によって x 軸方向の処理順序が決まる．loop_x では，ボリュームを $\mathrm{x} < \mathrm{cx}$, $\mathrm{x} > \mathrm{cx}$, $\mathrm{x} = \mathrm{cx}$ の 3 つの領域に分割し，それぞれで処理順序を決める．まず，コードでは領域 $\mathrm{x} < \mathrm{cx}$ にあるキューボイドの処理順序を決めている．0 から cx - 1 まで 0, 1, 2, …, cx - 1 の順に決めると，視点遠方の

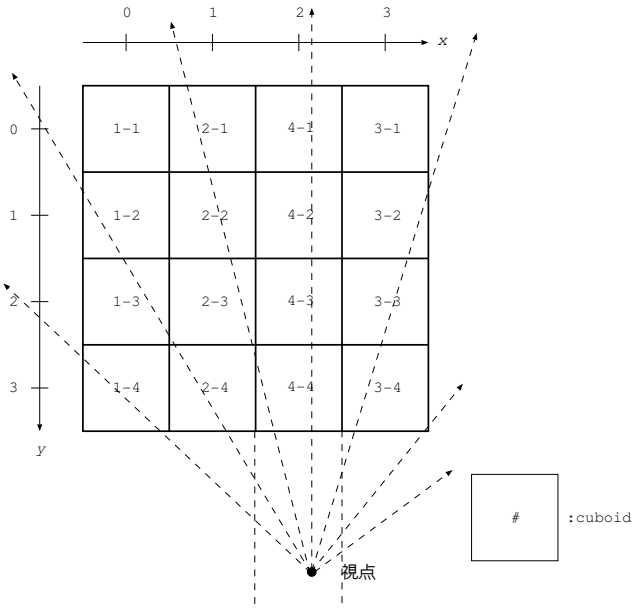

図 10: 処理順序の決定

キューボイドから処理されることになる．次の領域 x > cx では逆に，デクリメンタルに X_MAX から cx + 1 の順に決める．そして最後に，x = cx を追加する．こうすると，視点遠方のキューボイドから順に処理することができる．同図の例では，処理順序は 0, 1, 3, 2 と決まる．

内側のループ loop_y によって y 軸方向の処理順序が 0, 1, 2, 3 と決まる．cx > X_MAX あるいは cx < 0 の時は，ループ loop_y() は実行されない．結局，図 10 のキューボイドは，1-1, 1-2, …，4-4 の順に処理される．全ての視線に対して，スクリーン奥にあるキューボイドが先に処理されることが確認できる．

3.2.1 軸間の順序

上述の説明では，x, y の各軸の間の順序に任意性がある．上述のコードは xy 型，すなわち，内側ループの処理が y 軸方向に進むように記述されている．一方，yx 型，すなわち，内側ループの処理が x 軸方向に進むコードでは，図 10 のキューボイドは，1-1, 2-1, …, 4-4 の順に処理され，やはり正しく動作する．

しかし，図 10 の場合では，先に示した通りの xy 型の方が性能が良い．できるだけ視線に沿った方向に処理を進めることによって，中間値を保存する配列に対応するフレームバッファ上の各ピクセルに対する参照の時間局所性が高まるためである．

3.3 アクセスパタンの制御

前述のように，GPU は，あるスライスの処理を途中で中断して，次のスライスに移るといったことはできない．スライス内のフラグメントが処理されるパタンは，ラスタライズのパタンに一致する．ラスタライザは固定機能のユニットであり，ラスタライズのパタンをプログラムするすることはできない．

キューボイド順テクスチャベース法では，あるキューボイド内のテクセルを全てサンプリングしてから次のキューボイドの処理に移るように，テクスチャのサンプリングパタンを通常のテクスチャベース法から変更する．

キューボイド順テクスチャベース法では，アクセスパタンの制御は，ラスタライズされるスライスの配置をキューボイド順にすることで行う．スライスをキューボイド単位で配置するために，まずスライスを分割する．分割したスライスの大きさは，スクリーンに投影されたキューボイドを覆う大きさである．キューボイドの大きさが元のボリュームの半分である場合，分割されたスライスの大きさは元のスライスの半分となる．

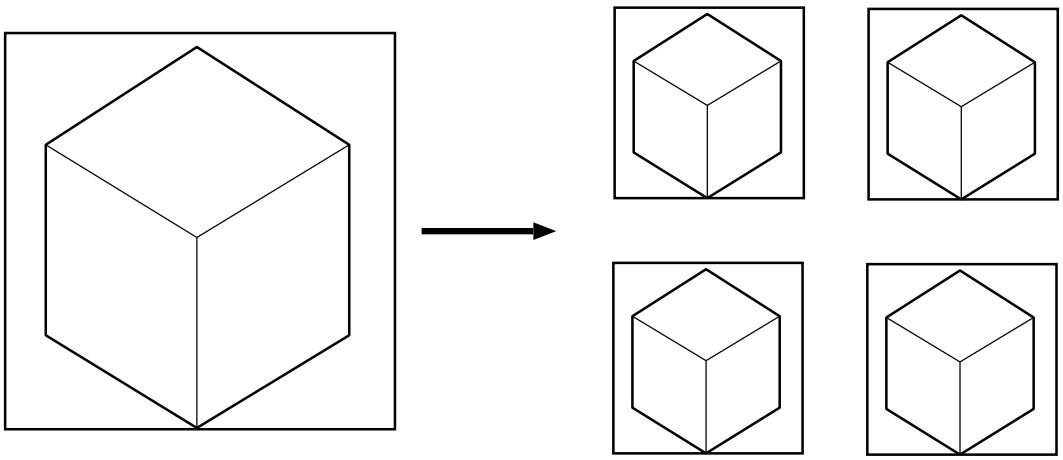

図 11: スライスの分割

3.3.1 スライスの配置

各キューボイドに対応するスライスは，ボリューム内におけるキューボイドの位置に配置する．

各キューボイドを覆うスライスは，始め原点に位置している．各軸方向のキューボイド番号を基に，キューボイドのボリューム内における位置に平行移動する (図 12)．

平行移動する距離と方向は，ボリュームの原点と，キューボイドの中心を結んだベクトルの方向と長さによって決定される．

予め各キューボイドの位置にスライスを配置しないのは，原点においてクリッピングを行うためである．テクスチャベース法では，スライスをキューボイドの投影面積の最大値よりも大きく取った上で，レンダリング領域をクリッピングして余分な領域のレンダリングを省略している．現在の GPU はクリッピング機能に制限があり，一度に設定できるクリップ平面数の最大値は 6〜 8 となっている．もし全てのキューボイドのスライスを同時にクリッピングしようとすると，クリップ平面が不足する．

そのため，まず各キューボイドのスライスを原点でクリッピングしたのち，所定の位置に移動するか，クリップ平面をキューボイド毎に移動させるかのどちらかの方法で対処しなくてはならない．これらの方法は，GPU の処理量としては同等である．本稿では前者の方法を採用する．

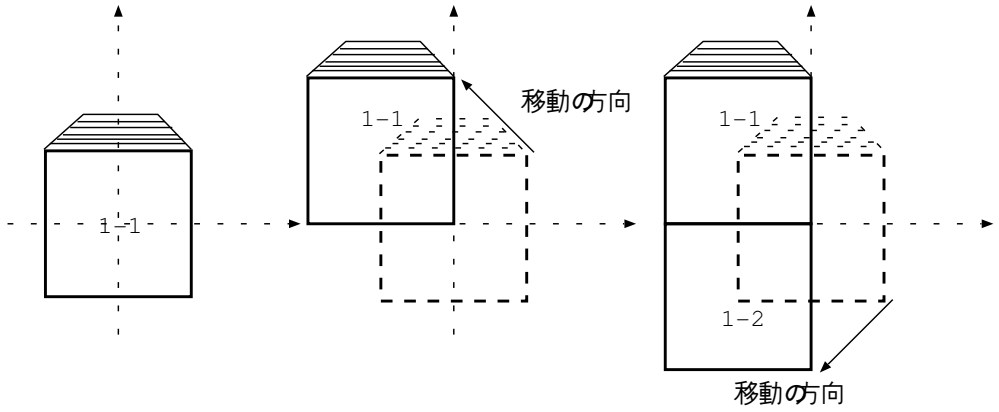

図 12: スライスの移動

3.3.2 視点変更時のスライス移動

テクスチャベース法では，視点の変更時はスライスの頂点座標は変化せず，テクスチャ座標のみが移動した．キューボイド順テクスチャベース法では，視点が変更されると，ボリューム全体の回転にあわせて，キューボイド単位でスライスが移動する．これは，視点が変更されることでボリュームが回転しても，ボリュームの中心は移動しないが，ボリューム内における各キューボイドの中心は移動するためである．

ただし，テクスチャベース法の原理から，スライスは視線に対して常に垂直でなくてはならない．この制限のため，移動後のスライスも，視線に対して垂直となっていなければならない．各キューボイドの中心の移動距離を CPU で求めてから平行移動を行うこともできるが，計算が繁雑となる．頂点プロセッサの回転機能を組み合わせて用いた方がより高速である．

まず，視点の変更に合わせて，スライスの座標を回転させる．この回転は，全てのキューボイドに対して影響する．次に，各キューボイドのボリューム内での位置へ，スライスを平行移動する．すでに座標系が回転しているため，平行移動後のキューボイドの中心は，ボリューム全体を回転させた時の位置となっている．続いて，キューボイドの中心を原点としたクリップ平面とテクスチャ座標の回転を行う．

スライスの座標系が回転したため，スライスは視線に対して斜めになっている．テクスチャベース法の条件を満たすために，スライスが視線に対して垂直になるように，スライスの座標のみを最初の回転とは逆方向に回転させる (図 13)．この操作をキューボイドを覆うスライスの全ての組について行う．

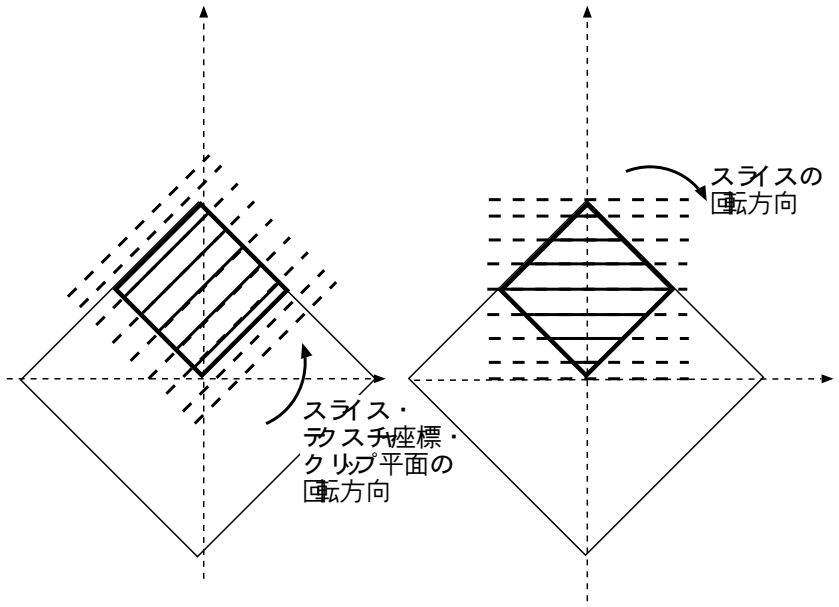

図 13: 視点変更時におけるスライスの回転

3.4　キューボイドの処理

キューボイド順テクスチャベース法では，キューボイドはレイキャスティング法ではなくテクスチャベース法によりレンダリングする．キューボイドの大きさに合わせた枚数のスライスを用意し，視点に対して奥から順次テクスチャマッピングを行う．

この処理をキューボイド順に繰り返すことで，全てのテクセルをサンプリングした上で，アクセスパタンをキューボイド順にすることができる．使用するテクスチャの切り替えは，あるキューボイドのレンダリングが完了した後，次のキューボイドを覆うスライスの処理に移る前に行う．

スライスのサイズは，キューボイドの投影面積よりも大きくとるため，クリッピング処理を行い，レンダリング領域をキューボイドの体積に等しくする．一度に設定できるクリップ平面の数には制限があるため，クリッピング処理を全てのキューボイドについて同時に行うにはできない．各キューボイドのレンダリング結果は順次フレームバッファに書き込まれるため，レンダリングの中間値を保存するための余分な領域は必要ない．

3.5　増加する処理

ここでは，キューボイド分割を行うことにより増加する処理について述べる．

キューボイド分割を行っても，テクスチャのサンプリング回数は全体として変化しないため，フラグメントプロセッサの処理量は変化しない．一方で，スライスの頂点の処理量，すなわち頂点プロセッサの処理量は，ボリュームを n^3 個のキューボイドに分割すると $O(n^2)$ の割合で増加する．

頂点プロセッサは，複数のポリゴンが同一の頂点が共有している場合は，計算結果を再利用する．しかし，キューボイドを覆うスライスの頂点は，クリッピングにより動的に決定されるため，クリッピングが行われる前の各スライスは，頂点が全く共有されない．頂点がポリゴン間で共有されないため，頂点プロセッサの処理量は，頂点数に比例する．

キューボイドの総数が n^3 倍になると，頂点数は n^2 倍に増加する．n^3 倍とならないのは，スライスの奥行き方向の枚数は変化しないためである．分割数が増大した際の，頂点プロセッサにおける処理時間の増大が懸念される．

3.5.1 頂点数の削減

通常のテクスチャベース法はフラグメントプロセッサの負荷が頂点プロセッサの負荷よりも高いため，頂点プロセッサの処理速度はレンダリング速度に影響しない．キューボイド順テクスチャベース法では処理する頂点数が多いため，頂点プロセッサの処理速度の改善も考慮しなくてはならない．

キューボイド分割による頂点数の増加は避けられないが，少しの工夫により，処理する頂点数を削減することができる．

スライスは，キューボイドを覆う大きさであればその形状を問わず，クリッピングされた結果は四角形のスライスを用いた場合と等しくなる．そこで，スライスの形状を四角形からより頂点数の少ない三角形に変更する．直角二等辺三角形を用いるとすると，1 辺の長さは，ボリュームの最も長い対格線の 2 倍の長さであれば，視点の位置によらずボリューム全体を覆うことができる (図 14)．

GPU は三角形をポリゴンの 1 単位として扱うため，四角形は 2 つの三角形としてレンダリングされる．スライスを四角形から三角形に変更することにより，頂点数が半分になる [9]．

実際に削減される処理量は，ポリゴン間で共有される頂点の処理方法によって異なる．共有される頂点を再計算しない GPU では，三角形にすることにより処理量は 3/4 に削減される．共有される頂点も全て再計算する GPU では，処理量は 1/2 まで削減される．

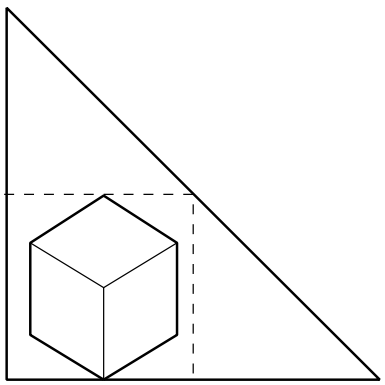

図 14: 三角形スライスとクリッピング

3.6 第3章のまとめ

キューボイド順レイキャスティング法の考え方をGPUに適用したキューボイド順テクスチャベース法を提案した．

キューボイド順テクスチャベース法は，ボリュームをテクスチャキャッシュのサイズよりも小さいキューボイドに分割する．分割したキューボイドに対してキューボイド順に処理することで，テクスチャへのアクセスパタンを制御する手法である．キューボイド順の処理とは，キューボイド全体がテクスチャキャッシュにフェッチされ，リプレースされるまでにキューボイド内の全てのテクセルをサンプリングすることである．こうすることで，テクスチャキャッシュの利用効率は最大化される．

キューボイドおよびキューボイド順の定義は，キューボイド順レイキャスティング法のそれと同様である．キューボイド順テクスチャベース法とキューボイド順レイキャスティング法方の主な相異点は，キューボイドの処理部分とアクセスパタンの制御の実装方法である．また，キューボイドをテクスチャとして扱うことに起因する，さまざまな制限が存在する．

ストリーム型プロセッサであるGPUでは，キューボイド単位でスライスを用意することで、アクセスパタンを制御する．キューボイドの総数がn^3倍となると，スライス数はn^2倍に増加する．このことによる性能低下が懸念される．

第4章　評価

本章では，提案するキューボイド順テクスチャベース法の評価を行う．評価に先立ち，2種類の予備評価を行う．

第2章で述べたように，GPUはテクスチャキャッシュの仕様が公開されていないため，キューボイドのサイズを事前に求めることができない．予備評価1として，サイズの異なるボリュームテクスチャについて，スクリーンの大きさとスライスの枚数を固定し，視点を変更しながらレンダリング速度を評価する．

第3章で述べたように，キューボイド分割を行うことにより，処理する頂点数は増大する．予備評価2として，キューボイドの分割数と頂点プロセッサにおける処理時間の関係を調べる．

キューボイド順テクスチャベース法の評価として，サイズの異なるキューボイドについて，視点を変更しながらレンダリング速度を評価した．

4.1　評価環境

CPUはIntel Pentium 4 2.5GHZを，GPUはATI Radeon X800 Proをそれぞれ用いて評価を行った．

Linux 2.6.8上でC, OpenGL 1.5およびCg (C for graphics)を用いて実装した．Radeon X800 Proの諸元を表1に示す．

表1: Radeon X800 Proの諸元

コアクロック	475 MHz
メモリ帯域	28.8 GB/sec.
フィルレート	5.7 GPixels/sec.
ジオメトリレート	712.5 MTriangles
Memory	256 MB
Memory Interface (bit)	256
Memory Data Rate	900MHz
Pixels per Clock (peak)	12

4.2 予備評価1：テクスチャキャッシュのサイズ

GPU が持つテクスチャキャッシュは，容量やラインサイズの仕様が公開されていない．そのため，最適なキューボイドの形状やサイズを前もって求めることができない．

そこで，まずテクスチャキャッシュの容量を推定する．視点が変化した時に，速度の変化が少なければ，ボリュームテクスチャのサイズがテクスチャキャッシュのサイズよりも小さいと考えられる．

サイズの異なるボリュームテクスチャを，同一サイズのスライスにマッピングし，レンダリング速度の変化を評価した．ボリュームテクスチャをタイル状にスライスにマッピングし，各テクセルが 1 回のフェッチにつき 1 度しかサンプリングされないようにする．

測定条件は，視点に対してボリュームが垂直に位置する場合で，スクリーンサイズが 512×512, スライス数が 512 となるよう設定した．クリッピングを用いて，視点に対してボリュームが斜めに位置する場合もスライスの総面積が変化しないようにした．スライスの形状は三角形とした．ボリュームテクスチャのサイズは， $512^3(128MB) \sim 8^3(512Byte)$ の間で変化させた．

テクスチャのデータ形式は，ALPHA_8 を用いた．これは，各テクセルが 1Byte の α 値のみを持つ形式である．ボクセル値の計算は，ルックアップテーブルを 2 次元テクスチャとして用意することはせず，フラグメントプロセッサ内でテクセルの α 値をそのまま RGBα 値として出力するようにした．これは，ボリュームテクスチャのアクセスパタンの変化とレンダリング速度の関係を見ることが目的であるためである．

テクスチャ座標を X 軸を中心に 0~360° の範囲で回転させた場合の結果を図 15 に，テクスチャ座標を Y 軸を中心に 0~360° の範囲で回転させた場合の結果を図 16 にそれぞれ示す．評価結果の単位は，フレーム毎秒 (FPS) である．

図 15 および図 16 より，ボリュームテクスチャのサイズを小さくするとレンダリング速度の変化が小さくなっている．8^3 まで小さくなると，どの視点でもほぼ同じ速度となっている．このことから，Radeon X800 Pro が持つテクスチャキャッシュのサイズは $4KB \sim 512Byte$ 程度であると推測できる．このサイズを越えるボリュームテクスチャであれば，アクセスパタンによってはテクスチャキャッシュが容量性のミスを起こすと考えられる．

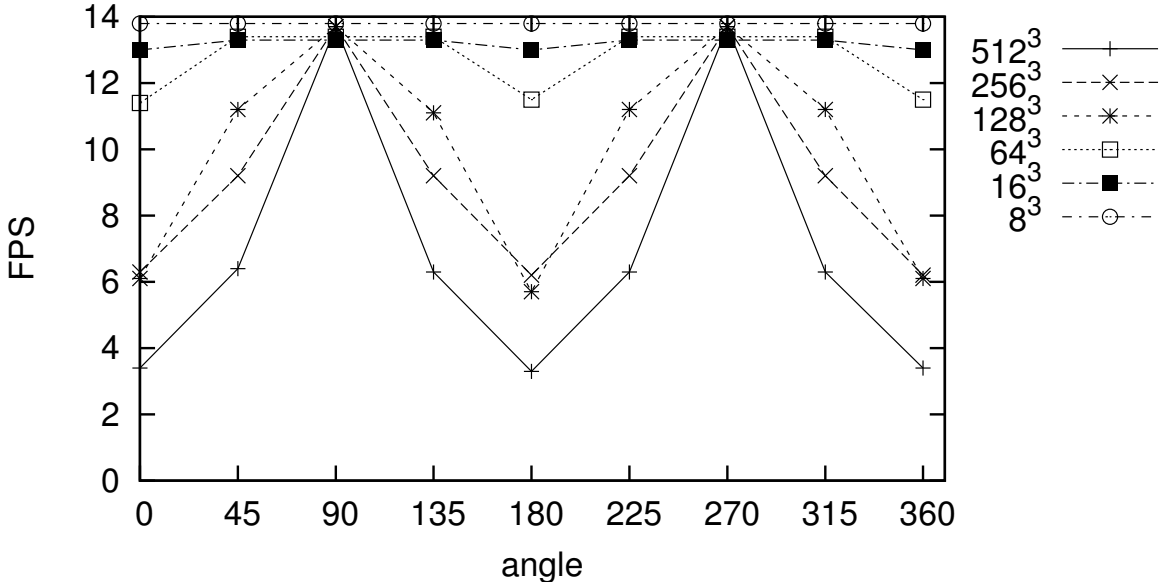

図 15: ボリュームのサイズと速度の関係 (X 軸中心)

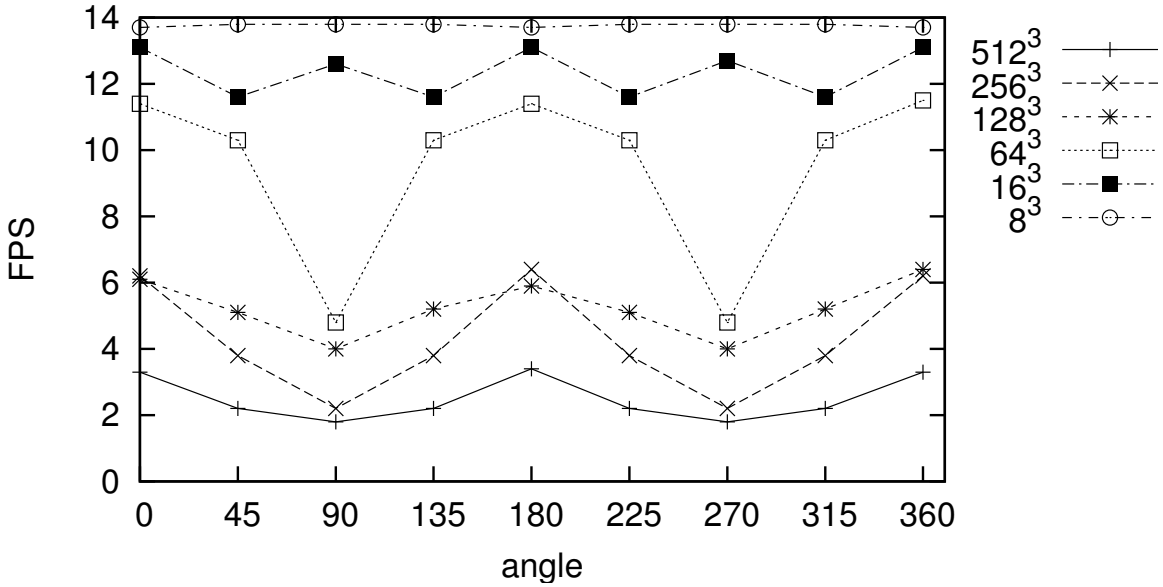

図 16: ボリュームのサイズと速度の関係 (Y 軸中心)

CPU において，キャッシュメモリが容量性のミスを起こし，メインメモリからデータを読み出す場合，DRAM にランダムアクセスを行うことになる．DRAM でランダムアクセスをする場合，メモリアドレスの間隔は，アクセス時間にほとんど影響しない．

GPU においては，テクスチャキャッシュに収まらないサイズであっても，ボリュームテクスチャのサイズを小さくすることによって，レンダリング速度の変化が小さくなっている．ビデオメモリは DRAM と同等のテクノロジを用いているが，テクスチャユニットが持つメモリコントローラと直結され，よりグラフィクス処理に適した構造となっている．ビデオメモリのメモリコントローラは，ランダムアクセス時にメモリアドレスの間隔が狭い場合にアクセス時間が改善されるような構造となっていると推測する．

4.3 予備評価 2：頂点プロセッサの負荷

フラグメントプロセッサの負荷を小さくした上で，キューボイドの分割数と速度の関係を調べた．スクリーンサイズは 1×1 ピクセルとした．この条件下では，フラグメントプロセッサが処理するのは 1 ピクセルのみとなるため，フラグメントプロセッサの負荷は頂点プロセッサの負荷と比べて十分小さくなる．そのため，レンダリング速度は頂点プロセッサの処理時間を表していると見なすことができる．スライスの形状は三角形とした．各キューボイドにおけるスライスの枚数は，キューボイドの面が視線に対して垂直となる角度で，キューボイドの 1 辺の長さとなるよう設定した．使用したボリュームのサイズは，512^3Byte である．評価環境は，予備評価 1 と同様である．

キューボイドのサイズは，$512^3 \sim 16^3$ の間で変化させる．スクリーンサイズ以外は，通常のキューボイド順テクスチャベース法と同様の処理を行う．すなわち，各スライスはキューボイド順にレンダリングし，テクスチャの切り替えも行う．X 軸を中心に 0~180° の範囲で回転させながらレンダリング速度を評価した．評価結果を表 2 に示す．速度の単位はフレーム毎秒 (FPS) である．

第 3 章で述べたように，処理する頂点数は，各軸方向の分割数を n とすると $O(n^2)$ の割合で増加する．表 2 を見ると，概ね $O(1/n^2)$ の割合で FPS が低下している．角度によって FPS が変化するのは，クリッピングによりスライスの枚数が変化するためである．回転角が 45° の場合，スライスの枚数は回転角が 0° の場合の $\sqrt{2}$ 倍となり，FPS は $1/\sqrt{2}$ となる．

表 2: キューボイドの総数と頂点プロセッサの処理速度の関係

	キューボイドの総数					
角度	512^3	256^3	128^3	64^3	32^3	16^3
0	1316.7	386.3	131.5	41.8	10.7	2.0
45	924.4	255.7	85.3	25.6	6.5	1.7
90	1310.3	383.7	130.6	41.6	10.6	2.0
135	922.2	255.5	85.3	25.6	6.5	1.7
180	1304.3	383.6	130.6	41.6	10.6	2.0

キューボイドが小さくなるにしたがって，角度毎の FPS の差が縮まっている．これは，各軸方向の分割数 n が増加すると共にテクスチャの切り替え回数も $O(n^3)$ の割合で増加することに起因する．分割数が増加すると，テクスチャ切り替えのコストが全体の処理量に占める割合が大きくなる．

キューボイドのサイズが 32^3 まで小さくなると，FPS は 10.7 ～ 6.5 まで低下している．視点によっては，頂点プロセッサの処理時間がフラグメントプロセッサの処理時間を上回ることが考えられ，性能低下が懸念される．

4.4 キューボイド順テクスチャベース法の評価

予備評価 1 より，キューボイドのサイズが 8^3 であれば，テクスチャキャッシュに乗ると予想できる．予備評価 2 より，キューボイドのサイズが 32^3 以下になると，頂点プロセッサの処理時間が全体の処理時間に影響を及ぼすことが懸念される．実際にキューボイド順テクスチャベース法によるボリュームレンダリングを行い，いくつかのサイズのキューボイドを用いて，レンダリング速度を評価した．

使用したボリュームテクスチャのサイズは，512^3 である．キューボイドの面が視線に対して垂直となる角度において，スクリーンサイズが 512^2，各キューボイドにおけるスライスの枚数がキューボイドの 1 辺の長さとなるよう設定した．

キューボイドのサイズを変化させながら，レンダリング速度を計測した．キューボイドの総数は，$(512/\text{キューボイドの 1 辺の長さ})^3$ である．テクスチャ座標を X 軸を中心に 0~360° の範囲で回転させた場合の評価を図 17 に，テクスチャ座標を Y 軸を中心に 0~360° の範囲で回転させた場合の評価を図 18 にそれぞれ示

す．なお，キューボイドのサイズを 8^3 とした場合は，途中で GPU がハングアップし，レンダリングできなかった．

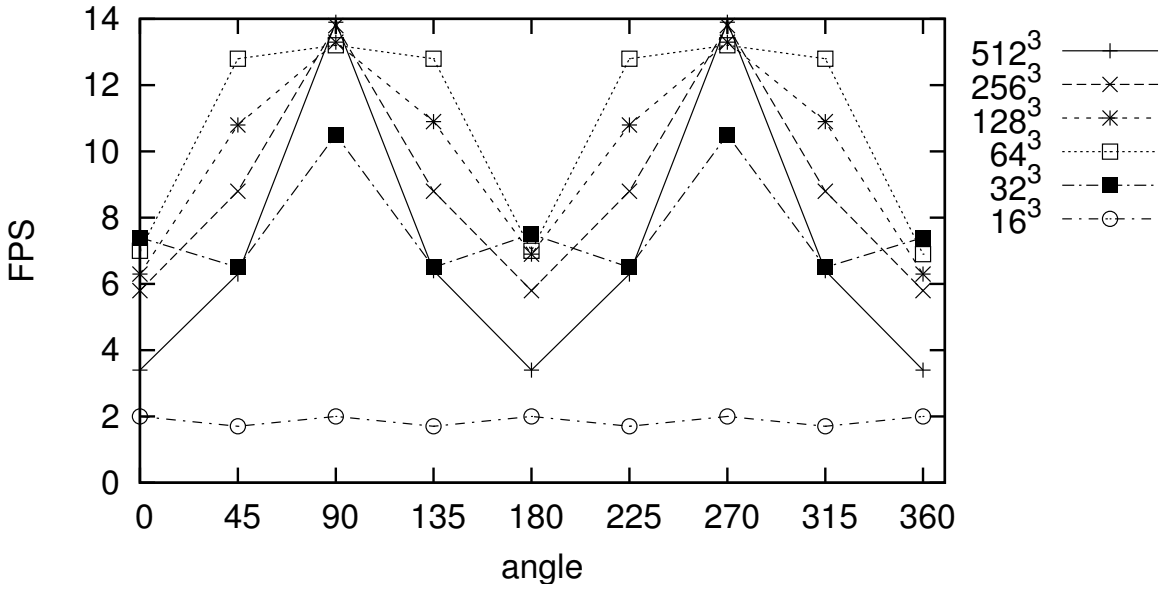

図 17: キューボイドのサイズと速度の関係 (X 軸中心)

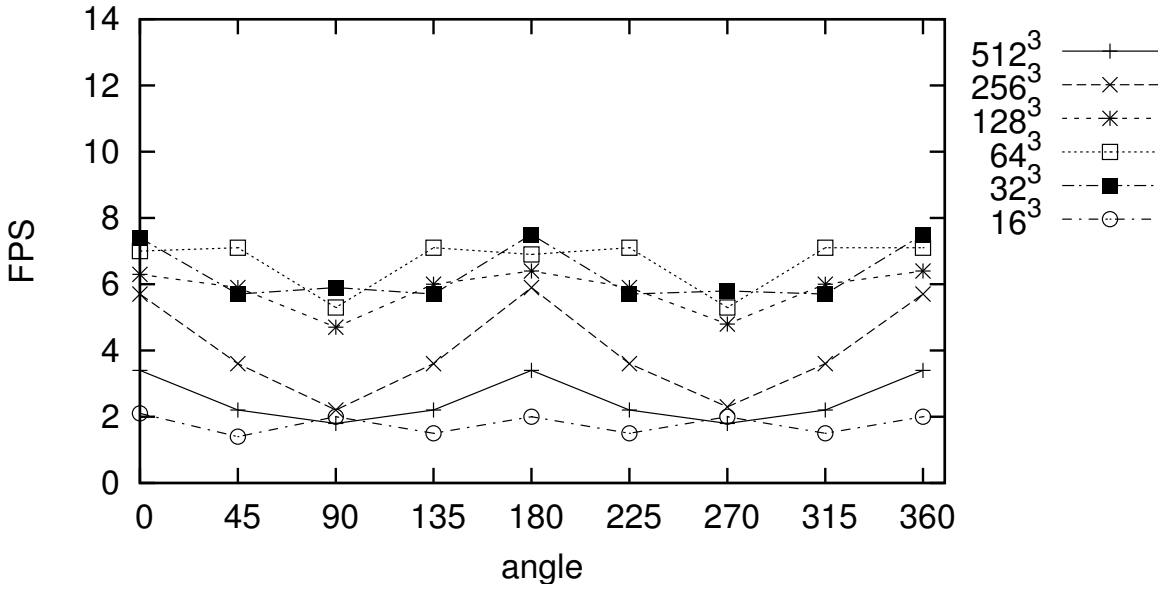

図 18: キューボイドのサイズと速度の関係 (Y 軸中心)

キューボイドのサイズについて，$512^3 \sim 64^3$ の範囲では，最良の性能は 512^3 の 13.9FPS からほとんど変化しない．最悪の性能は 512^3 の 1.8FPS から 64^3 の 5.3FPS へ向上している．32^3 では，最良の性能は 10.5FPS に低下し，最悪の性能は 5.9FPS に向上している．16^3 では，全ての視点においてレンダリング速度が大きく低下している．

予備評価 1 におけるレンダリング速度の推移とは異なっているが，予備評価 1 ではボリュームテクスチャの特定の断面がタイル状に繰り返しマッピングされるため，アクセスパタンがキューボイド順テクスチャベースとは根本的に異なる．

キューボイドのサイズが 8^3 である場合にレンダリングできなかったのは，GPU が保持できるテクスチャ数の上限を超過したためであると思われる．キューボイドの総数は $64^3 = 262,144$ 個である．グラフィクス API には，テクスチャの総数に関する制限はない．GPU の制限により，上限が存在するものと思われる．

キューボイドのサイズが 32^3 以下になると，レンダリング速度は表 2 の処理時間と等しくなっている．よって，32^3 以下のサイズでは頂点プロセッサの処理とテクスチャ切り替えの処理が性能のボトルネックとなっている．

最悪の性能の改善　ボリュームを回転させながら観察する際，滑らかな移動を実現するためには，視点によるレンダリング速度の変動がなるべく小さいことが望まれる．この場合，最悪の性能を向上させることが重要である．そのために最良の速度が多少低下しても，許容される．

図 17, 18 より，キューボイド分割を行わない場合，最悪の場合のレンダリング速度は 1.8FPS である．最悪の性能が最大となるのは，キューボイドのサイズを 32^3 にした場合で，5.7FPS まで改善している．この時，最良の性能は 13.9FPS から 10.5FPS に低下している．最悪の場合の性能向上が目的であり，最良の場合の性能低下は許容される．

キューボイドのサイズが 8^3 程度まで小さければ，テクスチャキャッシュを最大限に利用でき，最悪の場合のレンダリング速度が大きく向上すると考える．しかし，このサイズでは，頂点プロセッサの処理とテクスチャ切り替えの処理がボトルネックとなり，最良・最悪ともにレンダリング速度が大きく低下してしまっている．

第5章　考察

本章では，第 4 章の評価結果を基に，考察を行う．第 4 章で述べたように，キューボイド順テクスチャベース法の速度向上が伸び悩む要因は 2 つある．まず，ボリュームを分割して個別のキューボイドに分割することにより，テクスチャの総数が増え，テクスチャ切り替えのコストが増加する．次に，キューボイド毎に個別にスライスを用意するため，キューボイドの数の増加にあわせて頂点プロセッサにおける処理のコストが増大する．これらのコストがボトルネックとなり，性能向上が伸び悩んだ．

5.1　頂点プロセッサの理論性能と実効性能の比較

頂点プロセッサの性能が最大限に利用されているかどうかを見るために，予備評価 2 の結果と理論性能を比較する．頂点プロセッサが 1 秒あたりに処理した頂点数を，頂点プロセッサの実効性能と呼ぶことにする．実効性能の計算式は，

スライスの頂点数×1 キューボイドあたりのスライス数×キューボイドの総数×FPS

である．単位は Vertices とする．回転角 0° における分割数毎の実効性能を，表 3 に示す．

表 3: 頂点プロセッサの実効性能

分割数	1	2^3	4^3	8^3	16^3	32^3
実効性能	2,022,451	2,373,427	3,231,744	4,109,107	4,207,411	3,145,728

実効性能の最大値は約 4.2 MVertices となっている．これは，表 1 におけるジオメトリレートの理論性能と比較すると，1/100 にも達していない．理論性能は，各頂点に単純な処理しかしない場合の値であり，各頂点に平行移動と回転を施すキューボイド順テクスチャベース法と単純に比較はできないが，頂点の処理に関して改善の余地があると言える．

5.2　キューボイド順テクスチャベース法の改善

本節では，キューボイド順テクスチャベースの性能低下の要因となっている頂点プロセッサの処理とテクスチャ切り替えのコストを削減するための方法に

ついて考察する．

5.2.1 アドレス変換

テクスチャ切り替えのコスト軽減について考察する．ボリュームテクスチャをキューボイドに分割する方法をとる限り，テクスチャ切り替えのコストは必ず発生する．そこで，1 個のボリュームテクスチャ内で，各テクセルがキューボイド順に並ぶようなアドレス変換を行うことを考える．キューボイド内のテクセルが連続して並んでいれば，複数のテクスチャに分割する場合と同様のメモリアクセスを実現できる．

本稿では，GPU が持つ機能を活かしたアドレス変換の手法を提案する．変換後のボリュームテクスチャにアクセスする際に行うアドレス変換は，頂点プロセッサで設定された 3 次元のテクスチャ座標を基に，フラグメントプロセッサが行う．

キューボイドの 2 次元テクスチャへの変換　GPU のテクスチャキャッシュは 2 次元テクスチャのアクセスに最適化されているため，各キューボイドは 2 次元テクスチャに格納すると，効率良くサンプリングが行える．そこで，まずキューボイドを 2 次元テクスチャに変換することを考える．キューボイドのサイズが n^3 である場合，サイズが $n^{3/2} \times n^{3/2}$ である 2 次元テクスチャに変換する．8^3 のキューボイドの場合で，16×16 となる．

まず，GPU において 3 次元アドレス を 2 次元アドレスに変換することを考える．これには，2 つの方法がある [16].

- 3D アドレスを大きな 1D アドレス空間に変換してから，1D アドレスを 2D アドレスに変換する．
- 3D テクスチャの各スライスを結合して 2D テクスチャに変換する．

本稿では，アドレス変換に要する命令数がより少ない前者の方法を用いる．

まず，1 次元アドレスを 2 次元アドレスに変換する Cg コードを示す．

```
float2 addrTranslation_1Dto2D( float address1D, float2 texSize )
{
//事前に求め，定数として与える
 float2 CONV_CONST = float2( 1.0 / texSize.x,
                             1.0 / (texSize.x * texSize.y ));

 float2 normAddr2D = address1D * CONV_CONST;
```

```
 float2 address2D = float2( frac(normAddr2D.x), normAddr2D,y );
 return address2D;
}
```

ここで，frac(normAddr2D.x) は，normAddr2D.x の小数部分を求める関数である．texSize は，アクセスする 2 次元テクスチャのサイズである．address1D は変換を行う 1 次元アドレスである．

次に，3 次元アドレスを 1 次元アドレス空間に変換する Cg コードを示す．

```
float2 addrTranslation_3Dto2D(float3 address3D,
                              float3 sizeTex3D,
                              float2 sizetex2D)
{

//事前に求め，定数として与える
float3 SIZE_CONST = float3(1.0, sizeTex3D.x,
                           sizeTex3D.y * sizeTex3D.x);

float address1D = dot( address3D, SIZE_CONST);
return addrTranslation_1Dto2D( address1D, sizeTex2D);
}
```

dot(address3D, SIZEl_CONST) はベクトル address3D および SIZE_CONST の内積計算を行う関数である．内積を用いることで，3D アドレスから 1D アドレスへの変換が GPU のベクトル演算により効率的に行える．これら 2 つの関数を用いて，3 次元を 2 次元アドレスに変換できる．

この計算により，テクスチャ座標の 3 次元アドレスで，2D テクスチャに変換されたキューボイドにアクセスできる．次に，2 次元テクスチャに変換されたキューボイドを，1 個の 3 次元テクスチャに再結合することを考える．

キューボイドの再結合　2 次元テクスチャに変換されたキューボイドを，各キューボイドの位置関係を保ったまま 3 次元テクスチャに結合する (図 19)．3 次元テクスチャに結合するのは，GPU が扱える 2 次元テクスチャの最大サイズは 4096×4096 であり，大きなボリュームを 2 次元テクスチャには変換できないためである．

再結合されたボリュームテクスチャへのアクセス　文献 [5] では，**キューボイド番号**と**キューボイド内オフセット**を用いて，アドレス変換を行ったボリューム

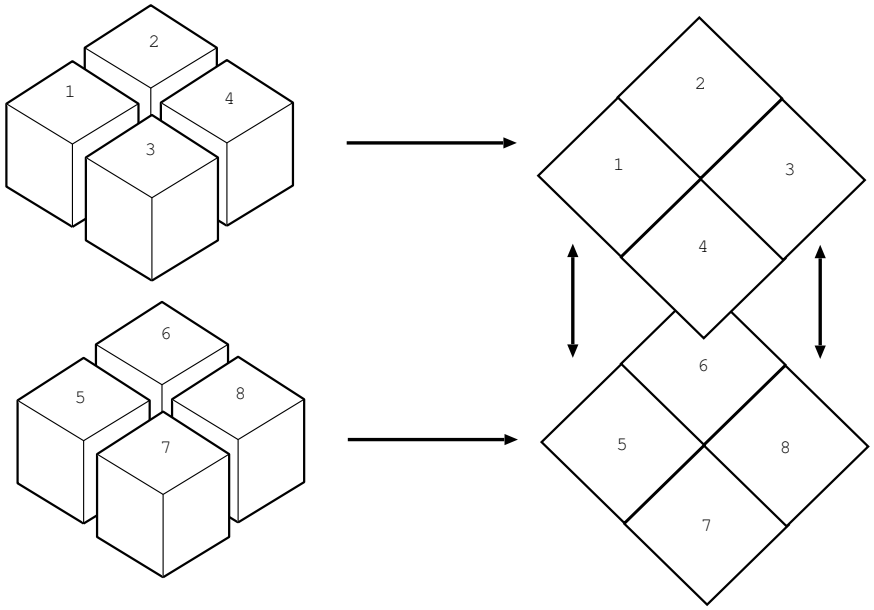

図 19: キューボイドの再結合

にアクセスしている．この考え方を援用し，再結合されたボリュームテクスチャへアクセスする．

キューボイドを連結したボリュームテクスチャにアクセスするには，テクスチャ座標をキューボイド番号とキューボイド内オフセットに分解する．テクスチャ座標を，キューボイドの各辺のサイズを [0:1.0] の範囲に正規化した値で除算する．この計算の商がキューボイド番号であり，余りがキューボイド内オフセットである．

このようなアドレス変換を行うことで，テクスチャの切り替えが不要となる．アドレス変換のコストは生じるものの，変換処理は GPU のベクトル演算を用いて効率良く行える．

5.2.2 スライス数の削減

アドレス変換により，テクスチャを切り替える必要はなくなる．しかし，スライス数の増加は避けられない．スライスの配置によって，テクスチャのアクセスパタンを制御しているためである．

テクスチャベース法は，2 枚のスライスを用いて α ブレンディングを行っている．この処理は，フラグメントプロセッサで 2 個のテクセルをサンプリングして α ブレンディングすることで代用できる．α ブレンディングはピクセルユニットで行われる処理であるが，フラグメントプロセッサにおいても効率よく

計算できる．Cg では，線形内挿を行う lerp() という関数が用意されており，この関数で α ブレンディングを実現できる．フラグメントプロセッサで α ブレンディングを行う回数に比例して，スライスの枚数も削減できる．この処理は最終的に，フラグメントプロセッサ内でレイキャスティング法に相当する計算を行うことになる．

テクスチャベース法では，サンプリング間隔とテクスチャ座標は，スライスの座標と視線に対する奥行き方向の間隔によって指定された．レイキャスティング法では，視線ベクトルとサンプリングポイントの更新もフラグメントプロセッサで行う．ただし，キューボイド間の α ブレンディングは，これまで通りピクセルユニットで行われる．

テクスチャサンプリングの開始点と終了点は，ボリュームと同体積の立方体をスクリーンに投影することによって決定できる．GPU においては，これは，立方体の頂点を回転することに他ならない．立方体の面のうち，視点に面していない裏面の頂点座標がサンプリングの開始点となり，裏面がサンプリングの終了点となる．これらの判定は，GPU の機能により簡単に行える．

なお，サンプリングを終了するためには，フラグメントプロセッサにおいて動的分岐が実行できることが必要である．従来は，フラグメントプロセッサ内で実行できる命令数の制限と，動的分岐を行えないという制限により，GPU でレイキャスティング法は用いることができなかった．近年，GPU の機能拡張により，GPU でレイキャスティング法を用いることが可能になっている [17]．

キューボイドのサイズを 8^3 とすると，テクスチャベース法で必要となるスライスの枚数は，最小の場合で 8 であり，頂点数は 24 である．一方，提案手法で必要となる立方体の頂点数は，キューボイドのサイズによらず 8 であり，処理する頂点数を大幅に削減できる．

立方体の頂点とテクスチャ座標が一致するため，クリッピングが不要となる．各立方体を予めレンダリングされる位置に配置することができ，キューボイド毎の平行移動が不要となる．また GPU は，複数のポリゴンが同一の頂点を使用する場合，計算結果を再利用することで不要な再計算を避ける構造となっている．そのため，隣り合う立方体では，計算結果が再利用される．これらの要因により，RC 法を用いることで，実際には頂点数の削減量以上に処理量を削減できる．

アドレス変換と立方体ポリゴンを用いることで，キューボイド分割時に頂点

プロセッサが性能のボトルネックではなくなる．一方で，フラグメントプロセッサの命令数が増加することにより，一定の性能低下が予想される．フラグメントプロセッサにおける動的分岐を備えたGPUで，評価を行う予定である．

現時点でレンダリング速度が低下しても，データ供給能力が性能向上の足かせにならない限り，提案手法によるレンダリング速度はGPUの性能向上に比例して今後も向上すると考える．

5.3 GPUに必要である機能

GPUを汎用的な可視化の手段と用いるためには，現在のGPUが持つ機能は未だ十分とは言えない．

現在のGPUは，CPUで発行された命令を実行するという構造に起因する制限が存在する．GPUの性能を活かすためには，GPUがCPUからより独立して動作することが必要であると考える．

ただし，GPUとCPUは排他的に利用されるものではない．GPUとCPUはパイプライン処理が可能であるため[13]，例えば，CPUでシミュレーションを実行し，GPUでその結果を可視化することで，CPUとGPUの双方を同時に利用できる．このように，今後はCPUとGPUを協調して動作させることが重要となるであろう．

GPU内部でのテクスチャ切り替え

キューボイド順テクスチャベース法では，GPUの制限から，キューボイドをレンダリングする前にアクセスするテクスチャをCPUがグラフィクスAPIを介して指定する必要がある．そのため，キューボイドの分割数が増加すると，テクスチャ切り替えのコストがレンダリング速度に影響を及ぼしてしまう．アドレス変換を行うことでテクスチャ切り替えが不要となることを5.2.1節で示したが，アクセスするテクスチャをGPU内部で切り替えることができれば，より高速な動作が期待できる．これは，各スライスがアクセスするテクスチャのインデックスをテクスチャユニットが保持し，条件に応じてアクセスするインデックスを変化させることで実現できると考える．

第6章　おわりに

本稿では，キューボイド順レイキャスティング法の考え方をGPUに適用したキューボイド順テクスチャベース法を提案した．

キューボイド順テクスチャベース法を評価した結果，キューボイドのサイズを適切に設定すると，最悪の性能が約3倍に向上した．キューボイド順テクスチャベース法を用いない場合と比較して，最悪の性能は大幅に改善された．しかし，テクスチャキャッシュを最大限に利用するには至っていない．キューボイドのサイズが小さくなると，すなわちキューボイドの分割数が大きくなると，使用するテクスチャの切り替え回数とスライスの頂点処理が増加し，性能向上が伸び悩んだ．

キューボイド順テクスチャベース法の性能向上を阻んでいる要因は，テクスチャ切り替えのコストと，スライスの頂点処理に関するコストである．テクスチャ切り替えのコストについては，キューボイドを個々のテクスチャとせずに，アドレス変換を用いる手法を提案した．キューボイドを1個のボリュームテクスチャに再結合することで，テクスチャの切り替えが不要となる．スライスの頂点処理のコストについては，フラグメントプロセッサにおいてαブレンディングを行うことで削減できることを述べた．

提案した改善手法が適用できるGPUは現在限られているが，GPUの機能拡張にあわせて，今後より多くのGPUに適用できるようになる．そのため，近い将来，GPUを用いたボリュームレンダリングにおけるメモリアクセスの問題は解決すると考える．その時，GPUを用いたボリュームレンダリングは汎用的な可視化手段として広く用いられるようになるであろう．

謝辞

本研究の機会を与えていただいた富田 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ご指導を賜った森 眞一郎 助教授，中島 康彦 助教授，嶋田 創 特任助手，三輪 忍 助手に心から感謝いたします.
さらに，日頃からご討論頂く京都大学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いたします.

参考文献

[1] Lichtenbelt, B., Crane, R. and Naqvi, S.: *Introduction To Volume Rendering*, Hewlett-Packard (1998).

[2] 中村昌典, 他: フラットパネルディテクタ搭載 X 線 TV システム SHIMAVISION ELNOS の開発, 島津製作所 Medical Now, Vol. 44, pp. 14–15 (2000).

[3] 田益久, 他: SHD 画像データベース構築の試み-イメージ解析，検索，高速通信を統合したプロトタイプ製作-, 小型衛星研究会遠隔医療ワーキンググループ (SPWS-TMWG) 電子フォーラム 005 (1999).

[4] 額田匡則, 小西将人, 五島正裕, 中島康彦, 富田眞治: 参照の空間局所性を最大化するボリューム・レンダリング・アルゴリズム, 情報学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 137–146 (2003).

[5] 額田匡則, 小西将人, 五島正裕, 中島康彦, 富田眞治: 参照の空間局所性を最大化するボリューム・レンダリング・アルゴリズムの改良, 情報学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 11(ACS 7), pp. 356–367 (2004).

[6] Montrym, J. and Moreton, H.: The GeForce 6800, *IEEE Micro*, Vol. 25, No. 2, pp. 41–51 (2005).

[7] DirectX: http://www.microsoft.com/windows/directx/.

[8] OpenGL: http://www.opengl.org/.

[9] GPGPU: http://www.gpgpu.org/.

[10] Muraki, S., Lum, E. B., Ma, K.-L., Ogata, M. and Liu, X.: A PC Cluster System for Simultaneous Interactive Volumetric Modeling and Visualization, *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 95–102 (2003).

[11] 山崎俊太郎, 加藤究, 池内克史: 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, 情報処理学会 CVIMl-130-10 (2001).

[12] Rezk-Salama, C., K. Engel, M. B., Greiner, G. and Ertl, T.: Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Texturesand Multi-Stage Rasterization, *Proceedings of Eurograph-*

ics/SIGGRAPH Workshop on Graphics Hardware (2000).

[13] 篠本雄基, 三輪忍, 嶋田創, 森眞一郎, 中島康彦, 富田眞治: 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, 情報処理学会研究報告 2005–ARC–164 (SWoPP 2005), pp. 145–150 (2005).

[14] Wolfe, M.: More Iteration Space Tiling, *Proceedings of Supercomputing (SC'89)*, pp. 655–664 (1989).

[15] Lam, M. S., Tothberg, E. E. and Wolf, M. E.: The Cache Performance and Optimizations of Blocked Algorithms, *In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74 (1991).

[16] Pharr, M. and Fernando, R.(eds.): *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*, Addison-Wesley Pub (2005).

[17] Stegmaier, S., Strengert, M., Klein, T. and Ertl, T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting, *Proceedings of Eurographics/IEEE VGTC Workshop on Volume Graphics 2005*, pp. 187–195 (2005).