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

額田匡則[†] 小西将人[††] 五島正裕[†]
中島康彦[†] 富田眞治[†]

ボリューム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，キューボイド順レイ・キャスティング法では，ベクトル長の短縮と，カラム競合という 2 つの問題がある．従来手法のようにボリュームを 3 次元配列で定義すると，これらの問題は互いに競合し，両方を満足させることは難しい．本稿ではボリューム空間の座標から実効アドレスへのアドレス変換の任意性に注目し，両問題を同時に解決するアドレス変換を提案する．プログラムを実装し評価した結果，提案手法によりベクトル長が改善され，描画性能が 28.2%向上した．

Improvement of a Volume Rendering Algorithm for Maximum Spatial Locality of Reference

Masanori Nukata,[†] Masahito Konishi,[††] Masahiro Goshima,[†]
Yasuhiko Nakashima[†] and Shinji Tomita[†]

A cuboid-order ray-casting algorithm maximizes spatial locality of reference to the volume data and has two problems: shortening of vector length and column conflict in cache memory. It is difficult to solve both of them at the same time. Because these problems conflict each other when the volume data is defined as a normal three-dimensional array. In this paper we propose an address transform method by which these problems are separately solved. The evaluation result shows that rendering performance of our method is improved by 28.2%.

1. はじめに

ボリューム・レンダリング[1)~10)] とは，色と透明度を持つ単位立方格子**ボクセル**（voxel）で構成される**ボリューム**と呼ぶ 3 次元オブジェクトを，2 次元画像に変換する方法の総称である．ボリューム・レンダリングはその計算量のため，CPU/GPU の計算能力への要求もかなり大きいが，近年のデバイス技術の進歩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がより深刻な問題となってきている．それは，従来のボリューム・レンダリング・アルゴリズムには，利用可能な参照の局所性がほとんどないためである．

1.1 レイ・キャスティング法

ボリューム・レンダリングでは，**レイ・キャスティング法**（以下，**RC 法**と略）を基礎とすることが多い．RC 法とは，視点からピクセルにキャストされた**視線**上の**サンプリング点**（以下，SP と略）にあるボクセルの値をサンプリングしてそのピクセルの値を求める手法である．特に，スクリーン上のピクセルを 1 つずつ逐次的に描画する**ピクセル順**（pixel-order）RC 法が一般的である．

RC 法では視点位置によって SP の座標が，そしてその結果，メモリへのアクセス・パタンが動的に決まる．このことが，参照の局所性抽出を困難にする．

1.2 RC 法とキャッシュ・メモリ

RC 法には利用可能な参照の局所性がないため，キャッシュは有効に働かない．各ボクセルは原理的にたかだか 1 回しかアクセスされないため，時間局所性は元々ほとんどない．そのうえ，アクセス・ストライドが一般にキャッシュ・ライン・サイズを超えるため，空間局所性も失われてしまう．

最近の高速 DRAM は，キャッシュとの間のライン転送に特化することで，増大する CPU の要求スループットに応えてきた．その一方で，このような DRAM のランダム・アクセス性能はそれほど改善されていな

† 京都大学
Kyoto University
†† 大阪工業大学
Osaka Institute of Technology

い．そのため，キャッシュが有効に機能しないような状況におけるシステムの性能低下はますます顕著になる．

1.3 キューボイド順 RC 法

我々は，タイリング[11] と同様の考え方によって，ボリュームへのアクセス・パタンを制御する**キューボイド順**（cuboid-order）RC 法を提案した[12),13)]．

キューボイド順 RC 法は，ボリュームを**キューボイド**（cuboid：直方体）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手法である．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 SP をすべて一気に処理すると，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・ヒット率は最大化される．

1.4 キューボイド順 RC 法の改良

ただし，文献 12) で提案したキューボイド順 RC 法では，キャッシュ・ヒット率は最大化されるものの，まだプロセッサの最大性能を引き出すには至っていない．それは，(1) ベクトル長の短縮 と，(2) カラム競合の 2 つの問題による．本稿では，これら 2 つの問題点の解決を試みる．

これらの問題は，個別には，以下のようにすれば解決できる：(1) ベクトル長は，キューボイドの形状を立方体に近づけることで長くすることができる．(2) カラム競合は，整合配列を用いるなどすればよい．

しかし，これら 2 つの問題と解決法は互いに競合しており，両方を同時に満足させることは難しい．それは，従来方式では，ボリュームを単純な 3 次元配列で定義しているためである．

さて，3 次元配列で表されるボリューム空間内の座標 (x, y, z) にあるボクセル vlm$[x][y][z]$ をサンプリングするとしよう．実行時には，式 &vlm$[x][y][z]$ に従い，x，y，z から実効アドレスが計算され，メモリにアクセスすることになる．このことは，ボリューム空間内の座標——**ボクセル・アドレス**(x, y, z) から，**実効アドレス**&vlm$[x][y][z]$ へのアドレス変換と考えることができる．

このような観点からすると，前述した (1) ベクトル長は，ボクセル・アドレス空間の；(2) カラム競合は，実効アドレス空間の問題であるとすることができる．

3 次元配列を用いる従来の方式[2),10)] では，おおよそ x，y，z を連結することによって，比較的容易に実効アドレス &vlm$[x][y][z]$ を得ることができる．しかしその一方で，(1) ボクセル・アドレス空間におけるベクトル長の問題と，(2) ボクセル・アドレス空間におけるカラム競合の問題が，互いに競合することになるのである．

そこで本稿では，これら 2 つの問題を個別に解決できるアドレス変換を提案する．この変換では，上記の 2 つ問題を各空間において個別に解決することが可能になる．そのうえ，最近のプロセッサが持つ命令を用いれば，比較的容易に変換を行うことができる．以下では，まず 2 章でピクセル順 RC 法について述べた後，3 章でキューボイド順 RC 法について説明する．4 章でキューボイド順 RC 法の 2 つの問題の競合について説明し，5 章でその解決法を述べる．最後に 6 章では，5 章で述べた手法を，Itanium 2 サーバ上に実装し評価した結果を示す．

2. ピクセル順レイ・キャスティング法

本章ではピクセル順 RC 法について説明する．

図 1 に，ピクセル順 RC 法の C++コードを示す．

以下に，RC 法における重要な概念と，コードで用いられている型，変数を説明する：

ピクセル値 構造体 Pixel では，RGB の 3 色を float

```
struct Pixel { float r, g, b; };          // ピクセル値
struct Voxel { float ar, ag, ab, t; };    // ボクセル値
struct Vector { float x, y, z; };         // ベクトル

unsigned char vlm[N][N][N];               // ボリューム
Voxel map[UCHAR_MAX+1];                   // カラーマップ
Pixel pxl[N][N];                          // スクリーン

for (int u = 0; v < N; ++v)
  for (int u = 0; u < N; ++u) {
    float dx, dy, dz;                     // 視線ベクトル
    float x, y, z;                        // サンプリング点
    float r, g, b;                        // ピクセル値
    // (1) 初期化
    init(u, v, &dx, &dy, &dz, &x, &y, &z);
    r = g = b = 0.0F;
    // (5) 終了判定 (1FLOP)
    while (IS_IN_VOLUME(x, y, z)) {
    // (2) サンプリング (3FLOP)
    Voxel vxl = map[vlm[(int)x][(int)y][(int)z]];
    // (3) ピクセル値の更新 (10FLOP)
    float t = vxl.t;                      // 透明度
    r = t * r + vxl.ar;
    g = t * g + vxl.ag;
    b = t * b + vxl.ab;
    // (4) SP の更新 (3FLOP)
    x -= dx;  y -= dy;  z -= dz;
  }
  // (6) ピクセル値の書き込み
  pxl[u][v].r = r;  pxl[u][v].g = g;  pxl[u][v].b = b;
}
```

図 1 ピクセル順レイ・キャスティング法のコード
Fig. 1 Code for pixel-order ray-casting algorithm.

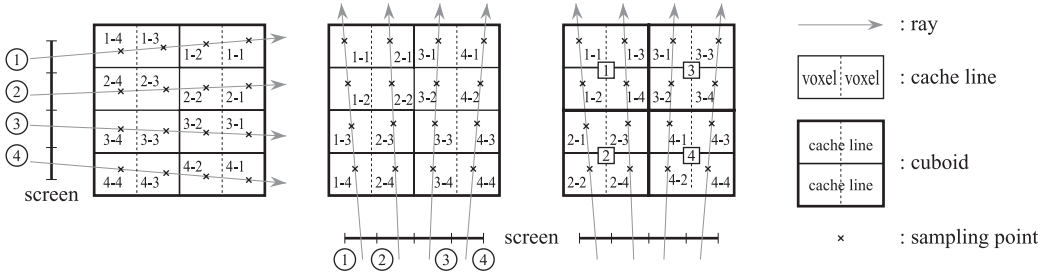

図 2 視点の位置と SP の処理順序
Fig. 2 Viewpoint positions and calculation orders of cuboids.

で表している．

ボクセル値 構造体 Voxel は，透明度 t と，RGB の 3 色と不透明度 a = 1 − t の積を持つ．

ボリューム，ボクセル データ量を抑制するため，ボリュームは 256 色の疑似フルカラーで表す．vlm から読み出した 1 (B) のインデクスでカラーマップ map を牽いて，実際のボクセル値を得る．

スクリーン，ピクセル pxl は，スクリーンの各ピクセルの値を表す．最内側ループでは，一時変数 (r, g, b) を用いて計算している．

視線，視線ベクトル 視線は，視点からピクセルへ向かう半直線である．視線ベクトル $\boldsymbol{R} = (\mathrm{dx}, \mathrm{dy}, \mathrm{dz})$ は，サンプリング周期を長さとする視線方向のベクトルである．

サンプリング点（SP） 視線上，ボリュームの一番奥の点を $\boldsymbol{P_0}$ とすると，SP (x, y, z) は，$\boldsymbol{P_0} - n \cdot \boldsymbol{R}$ $(n = 0, 1, 2, \cdots)$ で与えられる．

プログラムの動作

ピクセル順 RC 法のコード全体は図 1 に示すように，3 重のループからなる．外側の u，v の 2 重ループが計算すべきピクセル pxl[u][v] を定め，最内側の while ループがその値を計算している．

最内側ループでは，以下のように処理が進む：

(1) **初期化** 視線ベクトルを計算する．SP は $\boldsymbol{P_0}$，(r, g, b) は 0 に初期化する．

(2) **サンプリング** SP が指す vlm を読み出し，map を牽いて，ボクセル値 vxl を得る．

(3) **ピクセル値の更新** (r, g, b) を，サンプリングされたボクセル値によって累積的に更新する．

(4) **SP の更新** SP の座標から視線ベクトルを引くことによって，次の SP を得る．

(5) **終了判定** 新しい SP がボリュームを外れていたら最内側ループを終了する．

(6) **ピクセル値の書き込み** 得られたピクセル値をスクリーンのピクセル pxl[u][v] に書き込む．

各処理の計算量は図に示すとおりで，1 イタレーションあたりの計算量は 13FLOP である．最内側ループ終了時点での (r, g, b) が，ピクセル値 pxl[u][v] を与える．

ピクセル値はサンプリングと積和演算を交互に繰り返すことで求まる．SP に注目すると，視線上を $\boldsymbol{P_0}$ からスクリーン側に向かって 1 つずつ順に移動している．

2.1 キャッシュとの親和性

前述のように，ボリュームに対する参照には時間局所性はほとんどない．したがって，キャッシュとの親和性を考えるにあたっては，空間局所性が重要である．

ピクセル順 RC 法の空間局所性は，以下に示すように，視点の位置に強く依存する．

図 2 に，2 次元のピクセル順 RC 法における視点の位置と SP の処理順序を示す．図中，丸数字が視線の，SP の近傍にある数字が SP の処理順序をそれぞれ示す．キャッシュ・ラインの矩形が示すように，同図では横方向がアドレスが連続する方向となっている．

同図左では，視点が横方向にあるため，フェッチされたライン内のボクセルは順にサンプリングされ，キャッシュ・ヒット率は最大化されている．

一方同図中央では，視点が縦方向にあるため，ヒット率が低下する．視線①の SP 1-1 のためにフェッチされた左上のキャッシュ・ラインは，視線②に対して SP 2-1 を処理するときには，容量性のミスを起こすことがある．その場合，フェッチした 1 ラインのうちの 1(B) しか利用できないことになる．

この場合，たとえばライン長を 128(B) とすると，主記憶からのラインの転送量は 128 倍となる．主記憶に対する要求バンド幅は $1/13 \times 128 = 9.8$(B/FLOP) と，通常の数値処理と同等の値となり，メモリ・バウンドな処理に変わる．現存する PC や WS の主記憶は，このような高いバンド幅を提供していない．

2.2 ま と め

ピクセル順 RC 法では視点位置によりアクセス・パ

タンが動的に決まり，したがってキャッシュの利用効率も視点位置により決まる．アクセス・パタンがアドレス順になる最良の場合にはキャッシュ・ヒット率が最大化され計算バウンドな処理となるが，一方，最悪の場合はキャッシュの利用効率が著しく低下しメモリ・レイテンシが支配的なメモリ・バウンドな処理となってしまう．

6 章の評価で用いる環境でプログラムを実装し評価した結果，最良の場合と最悪の場合の描画速度の差は 4.4 倍であった．この環境ではメモリ・レイテンシが 41 サイクルと非常に高速なため最悪の場合の性能低下は抑えられているが，汎用 PC のようにレイテンシの大きな環境では性能は大きく低下してしまう．

この問題に対し，我々は視点位置と独立にキャッシュ・ヒット率を最大化するキューボイド順 RC 法を提案した．キューボイド順 RC 法では，視点位置にかかわらずキャッシュ・ミスは発生しない，したがって，ピクセル順のようなメモリ・レイテンシによる性能低下はない．次章では，キューボイド順 RC 法について説明する．

3. キューボイド順レイ・キャスティング法

密行列に関する数値処理の中には，タイリング[11]などの技法によって，参照の局所性——主に時間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 PC や WS 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．

本章では，キューボイド順 RC 法について説明する．キューボイド順 RC 法は，タイリングと同様の考え方によって，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，通常の数値処理と異なり，RC 法はボリューム参照の時間局所性を持たないので，空間局所性を最大化することが目的となる．

3.1 キューボイド順 RC 法の概要

ボリューム参照の空間局所性の最大化は，あるキャッシュ・ラインをフェッチしたときに，そのラインをサンプリングするすべての SP を処理しつくすことによって達成される．前述のように，ボリューム参照は時間局所性を持たないので，それだけでキャッシュ・ヒット率の上限が達成されることになる．

アクセス・パタンを制御する際には，オーバヘッドの低減のため，キャッシュ・ラインそのものではなく，数十個のラインから構成されるサブ・ボリューム，キューボイドを単位とする．キャッシュ・ラインは，そのサイズを 128(B) とすると，ボリューム空間では 1×1×128 の細長い拍子木状の空間を占める．キューボイドはこの細い拍子木から成る直方体である．キューボイドの形状には任意性があるが，これについての議論は 4 章で行う．

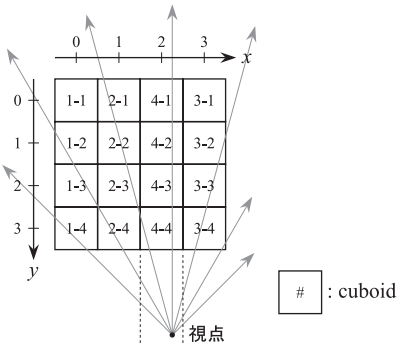

図 3 キューボイドの処理順序
Fig. 3 Calculation order of cuboids.

キューボイドのサイズは，1 次キャッシュのサイズを考慮して定める．キューボイド全体をキャッシュに保持するために，キューボイドのサイズを 1 次キャッシュのサイズの 1/4〜1/2 とする．たとえば，6 章のプログラムでは，1 次キャッシュ16(KB) の半分の 8(KB) をキューボイドに割り当てている．このとき，キューボイドを構成するラインの数はキューボイドの容量をライン長で割った 8(KB)/128(B) = 64 である．したがって，キューボイドは，64 本のラインを束ねた 8 × 8 × 128 のような形状をした直方体となる．

キューボイド順 RC 法は，ボリュームをキューボイドに分割してこれを処理単位とし，キューボイド順にレンダリングを行う．キューボイドのレンダリングとは，(1) キューボイドを通る視線をすべて抽出し，(2) 各視線のピクセル値を計算する ことである．視線抽出の方法については次節で説明する．ピクセル値計算については，以下で述べる点に注意する必要がある．

前章で述べたように，ピクセル値は視線上を $\boldsymbol{P_0}$ からスクリーンに向かって順にサンプリングすることで求められる．したがって，キューボイドの処理順序は全ピクセルについてサンプリング順序を正しく保つように決定しなければならないが，このような順序は必ず存在する．**図 3** に 2 次元の例を示す．処理順序を 1-1，1-2，···，4-4 とすると，すべ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よう．

また，あるキューボイドの処理が終了し別のキューボイドに処理を移すとき，ピクセル値計算を中断しなければならない．前述のように，ピクセル値はサンプリングと積和演算の繰返しで計算されるので，任意の

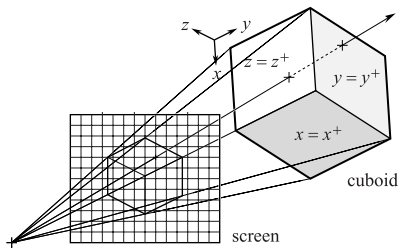

図 4 キューボイドの投影
Fig. 4 Projection of a cuboid.

点で中断可能である．計算の途中結果と，最後の SP の座標を記憶しておくと，計算を再開することができる．

キューボイドの処理順序を上記のように決めると，全視線について SP は $\boldsymbol{P_0}$ から順に処理されるため，途中結果はつねに 1 ピクセルにつき 1 つである．こうすると，ピクセル値の途中結果の保存にはスクリーンが利用できる．なぜなら，ピクセル順 RC 法においてスクリーンが利用されるのはピクセル値が求まった後で，それまでは自由に使用してよいからである．ただし，SP の座標については保存のための領域が別途必要となる．そのサイズは，途中結果と同じく全ピクセルについて保存する必要があるため，スクリーンと等しくなる．

3.2 キューボイドの処理

キューボイドの処理とは，当該キューボイドに含まれる SP をすべて列挙し処理することである．前述のように，SP の座標は視点位置によって動的に決まるので，静的なコード変換のような単純な方法ではアクセス・パタンを制御することはできない．そこで，以下に示す方法でキューボイド内の SP を処理する．

(1) **キューボイドの投影** キューボイドをスクリーンに投影する．**図 4** に示すように，一般的には 6 角形のキューボイドのシルエットが得られる．

(2) **ピクセルの選択** 値計算の対象となるピクセルは，キューボイド内に SP がある，すなわち，その視線がキューボイド内を通るものである．シルエット内部の任意のピクセルは，視線がキューボイド内を通る．逆に，シルエット外部のピクセルについては，キューボイド内を通ることはない．

(3) **スキャン変換** シルエットをスキャン変換することで，(2) で選択したピクセルを 1 つずつ列挙する．これは，6 つの四辺形ポリゴンで構成された直方体をレンダリングするのとまったく同じ方法で可能である．

(4) **ピクセル値計算** ピクセル値計算は図 1 で示したコードの最内側ループとほぼ同じである．ただし，ループの前後にピクセル値の途中結果のロード/ストアが必要となる．ループは SP がキューボイドから外れたら終了し，その SP の座標も保存しておく．こうすると，当該ピクセルのキューボイド内にあるすべての SP を処理することになる．

ここで，(4) ピクセル値計算について，サンプリングの始点と終点はともにキューボイドの内側にあることに留意していただきたい．最内側ループのベクトル長は視線上の SP の個数，すなわち，キューボイドの境界によりクリップされる視線の長さである．ピクセル順 RC 法におけるベクトル長が N で決まるのに対し，キューボイド順のそれはキューボイドの形状で決まる．

3.3 キューボイド順 RC 法の性能

キューボイド順 RC 法では，キャッシュ・ヒット率が最大化されるため，メモリに対する要求は 1/13(B/FLOP) ときわめて小さい．したがって，メモリの供給能力不足に起因する性能低下は起こらない．結局，本手法の性能低下要因は，(1) キューボイドの投影処理，(2) スキャン変換，(3) 最内側ループのベクトル長の短縮 の 3 点である．

このうち，(1) 投影処理は 8 頂点に対するアフィン変換であり，その計算量はキューボイド全体のそれに比べて十分小さいので無視してよい．また (2) スキャン変換について，その計算量は，キューボイド全体のそれのたかだか 0.5%にすぎず[12]，同様に無視してよい．

これらに対して，(3) ベクトル長の短縮による性能低下は比較的深刻である．前述のように，キューボイド順 RC 法のベクトル長はキューボイド内の視線長であるので，ピクセル値計算はベクトル長の小さい反復を複数回繰り返すことになる．このとき，ピクセルあたりの計算量は不変だが，ループ実行の序盤および終盤の並列度の低い部分の割合が相対的に増え，その結果性能が低下してしまう．

図 5 にベクトル長と性能の関係を示す．6 章に示す環境でプログラムを実装し，ベクトル長を変化させてそれぞれの場合の描画性能を計測した．グラフの横軸はベクトル長，縦軸は**実効 MFLOPS 値**である．実効 FLOPS 値とは，ピクセル値計算の計算量 $13N^3$ (FLOPS) を描画時間で割った値である．同図から，ベクトル長が小さくなるほど性能が大きく低下することが分かる．

まとめると，キューボイド順 RC 法の性能はベクトル長により決まる．ベクトル長が性能に与える影響は

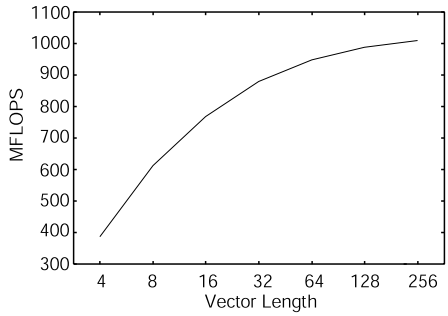

図 5 ベクトル長と性能
Fig. 5 MFLOPS value to vector length.

大きく，したがって，キューボイドの形状をいかにするかは重要な問題である．

4. キューボイド順 RC 法の問題点

本章では，キューボイド順 RC 法における 2 つの問題，(1) ベクトル長の短縮 と，(2) カラム競合対策 について述べ，そして従来手法において，これらの解決法が互いに競合することを説明する．

4.1 ベクトル長の短縮

4.1.1 視点位置とベクトル長

キューボイドを斜めから見ると，ベクトル長は各ピクセルごとに異なる．以下，これらの平均を**平均ベクトル長**と呼ぶ．平均ベクトル長は，キューボイドの体積を，スクリーン上に投影されたキューボイドの面積で割ると求められる．

キューボイドの形状を $C_x \times C_y \times C_z$ とする．また，視点が無限遠点にあるとする．このとき視点位置は，視線と x，y，z 軸のなす角 θ_x，θ_y，θ_z で表すことができる．平均ベクトル長は，これらのパラメータを用いて，$f(C_x, C_y, C_z, \theta_x, \theta_y, \theta_z) = C_x C_y C_z / (C_x C_y \cos\theta_z + C_y C_z \cos\theta_x + C_z C_x \cos\theta_y)$ と表すことができる．ただし，$\theta_z = \sqrt{1 - \cos^2\theta_x - \cos^2\theta_y}$ である．

平均ベクトル長 f について，キューボイドの形状を固定して θ_x，θ_y で微分し，最大値，および，最小値を求める．前述のように，キューボイド順 RC 法の性能はベクトル長で決まるので，平均ベクトル長 f が最大値をとる視点位置の場合を**最良**の場合，逆に最小値をとる場合を**最悪**の場合と呼ぶことにする．なお，キューボイドの形状ごとにそれぞれ最良の場合，最悪の場合が存在し，視点位置も互いに異なる点に注意していただきたい．

図 6 は，スクリーンに投影されたキューボイドの像である．キューボイドの形状を $8 \times 8 \times 128$ とすると，

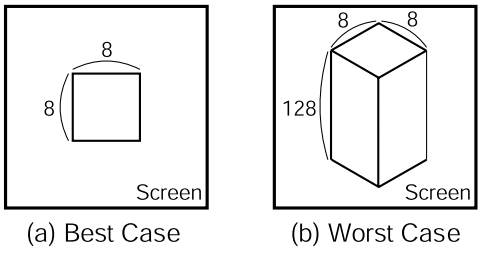

図 6 最良の場合と最悪の場合
Fig. 6 Best case and worst case.

最良の場合におけるキューボイドの像は同図 (a) のように 8×8 の正方形となり，このときの平均ベクトル長は $f = 128$ である．また最悪の場合では，同図 (b) のようにキューボイドの対角線がスクリーンと平行になり，$f = 5.66$ である．

4.1.2 最適なキューボイド形状

CPU/GPU の演算能力の向上により，現在ではボリューム・レンダリングのリアルタイム描画が実現している[14]．このような用途では，最悪の場合の性能を向上させることが重要である．このとき，最良の場合の性能を悪化させることになろうとそれは許容される．

さて，前述のようにキューボイド順 RC 法にはキューボイドの形状に任意性があり，キューボイドのとりうるすべての形状について平均ベクトル長の最小値が存在する．これらの最小値が最大となる形状のとき，性能の下限が最も高い，つまり，性能が高いといえる．このような理想的な形状とは，立方体である．直観的にも，立方体ならばどの方向から見ても投影面積がほぼ等しくなる，すなわち，視点位置の変化による性能の低下が小さくなることが分かる．

ただし実際には，キューボイドの体積によっては，必ずしも立方体にできるわけではない．キューボイドの辺の長さは整数値，特に 2 のべき乗である方が都合がよい．よって，この制約を満たし，かつ立方体に最も近い直方体が最適な形状となる．

以下に最適な形状の求め方を説明する．キューボイドの体積を C とする．

(1) $c^3 < C$ かつ $c = 2^n$ を満たす最大の c を求める．
(2) c の大きさにより，以下のように形状を決める．
  (a) $c^3 > C/2$ ならば，キューボイドの形状を $c \times c \times c$ とする．
  (b) $c^3 > C/4$ ならば，$c \times c \times 2c$ とする．
  (c) $c^3 < C/4$ ならば，$c \times 2c \times 2c$ とする．

例をあげると，6 章の評価に用いた環境では，$C = 8 \times 8 \times 128 = 8\text{K}$ なので，最適な形状は $16 \times 16 \times 32$

である．以上のようにキューボイドの形状を定めると，最悪の場合の性能を最大化することができる．

4.2 カラム競合対策

前述のように，キューボイド順 RC 法ではキューボイドの処理中，キューボイド全体をキャッシュに保持する必要がある．もしキューボイドを構成するラインが特定のカラムに偏っていると，競合が発生しラインはリプレースされてしまう．したがってカラム競合を防ぐためには，キューボイドを構成するラインを全カラムに均等に割り当てるのが理想的である．

整 合 配 列

図 1 に示すように，ボリュームを単純な 3 次元配列 $\mathrm{vlm}[N][N][N]$ で定義すると，隣接するライン間のストライドは 2 のべき乗となる．したがって，キューボイドを構成するラインは特定のカラムに偏って割り当てられることになる．

これに対し，従来は整合配列を用いることで対処していた．配列 vlm の z 軸方向のサイズを素数にするとストライドが素数となるため，隣接するラインは異なるカラムに割り当てられる．

この方法では確かにカラム競合は起こらないが，新たに別の問題が生じる．z 軸方向のサイズを N より大きくするため，N^2 の領域が必要となるが，これはまったく利用されない無駄な領域である．特にライン長が 128 と大きい場合，この問題は深刻である．$N = 1024$ の場合，$N^3 = 1024^3 = 1(\mathrm{GB})$ のボリュームに対し，配列のサイズは 1.4(GB) にもなってしまう．

4.3 両課題の競合

キューボイド順 RC 法における 2 つの課題は，個別には以上のようにすれば解決できる．しかし，両方を同時に解決しようとすると互いに競合してしまい，両方を満足させることはできない．

3.1 節で述べたように，カラム競合を避けるためにはキューボイドはキャッシュ・ラインを組み合わせたブロックでなければならない．たとえば，6 章の環境では，形状 $1 \times 1 \times 128$ のキャッシュ・ラインを束ねた，$8 \times 8 \times 128$ のブロックをキューボイドとしている．これを構成するラインのカラムは，整合配列を用いることで均等に分散させることができる．

ここで，z 軸方向の大きさ 128 はライン長 128 により決まる点に注意する．4.1.2 項で述べたように，理想的なキューボイドの形状は $16 \times 16 \times 32$ であるが，z 軸方向については 128 より小さくすることはできない．

結局，ボリュームを整合配列で定義する従来手法では，z 軸方向の形状の自由度はライン長により制限され，任意の形状をとることはできない．次章では，カラム競合を避け，かつ，キューボイドの形状を任意に決める方法について述べる．

5. 提 案 手 法

本章では，前章で述べた互いに競合する 2 つの問題を同時に解決する方法について説明する．

5.1 両問題の分離

整数 x，y，z によって，ボリューム空間内のボクセルを指定するとする．実行時には，式 $\&\mathrm{vlm}[x][y][z]$ に従い，x，y，z から，実効アドレスが生成され，メモリにアクセスすることになる．このことは，ボリューム空間内のボクセルの座標——**ボクセル・アドレス**(x, y, z) から，実効アドレス $\&\mathrm{vlm}[x][y][z]$ へのアドレス変換と考えることができる．このような観点からすると，前述したベクトル長は，ボクセル・アドレス空間の；キャッシュに対する親和性は，実効アドレス空間の問題であるとすることができる．

4.3 節において，(1) ベクトル長の問題 と，(2) キャッシュとの親和性の問題 が互いに競合することを述べた．これはボリュームを単純な 3 次元配列で定義していることが原因である．従来の方式におけるアドレス変換の式では，図 9 上に示されているように，おおよそ x，y，z を連結することによって実効アドレスを得ているが，実際には，この変換方法には任意性がある．以下で提案するアドレス変換により，ボクセル・アドレス空間におけるベクトル長の問題と；実効アドレス空間におけるキャッシュに対する親和性の問題とを切り放して，別々に解決することが可能になる．

5.2 アドレス変換

本節では，前述の両問題を解決するアドレス変換について述べる．まず，アドレス変換の説明に必要となる概念としてキューボイド番号とキューボイド内オフセットについて述べた後，アドレス変換の説明を行う．

5.2.1 キューボイド番号とキューボイド内オフセット

たとえば仮想アドレス・システムでは，アドレスはページ番号とページ内オフセットに分けて考える．それと同様に，ボリューム空間におけるボクセル・アドレスは，**キューボイド番号**と，**キューボイド内オフセット**に分けることができる．ただし，仮想アドレスなどでは，もともと 1 次元であるアドレスをページ番号とページ内オフセットに『2 次元化』するのに対して；ボクセル・アドレスは，もともと 3 次元であるものをさらに『2 次元化』することになる．

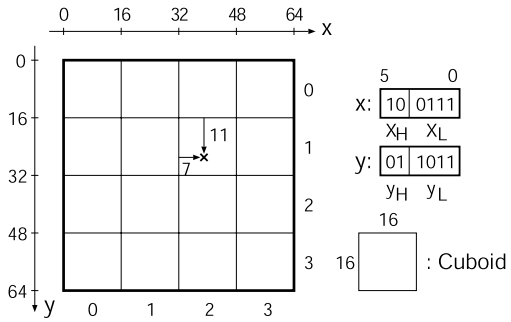

図 7 キューボイド番号とキューボイド内オフセット
Fig. 7 Cuboid number and offset.

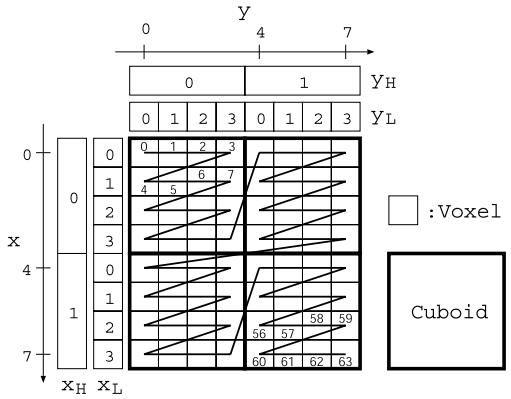

図 8 実効アドレス空間へのマッピング
Fig. 8 Mapping to effective address space.

図 7 に示す 2 次元のボリューム空間を例に，キューボイド番号とキューボイド内オフセットについて説明する．同図では，ボリュームのサイズ N は，$N = 64$ となっているので，ボクセル・アドレス (x, y) の x と y は，それぞれ $\log_2 N = \log_2 64 = 6$ ビットになる．また，キューボイドの x 軸，y 軸方向の長さ C_x，C_y がそれぞれ $C_x = C_y = 16$ となっているので，下位 $\log_2 C_x = \log_2 C_y = \log_2 16 = 4$ ビットがキューボイド内オフセット，残りの上位 2 ビットがキューボイド番号となる．ボクセル・アドレス (x, y) を，ビットの連結 "|" を用いて，$(x, y) = (x_H \,|\, x_L, y_H \,|\, y_L)$ と表すと，キューボイド番号は (x_H, y_H)，そのキューボイド内のオフセットは (x_L, y_L) となる．たとえば，図中のボクセル・アドレス $(39, 27)$ は，$(39, 27) = (2 \times 16 + 7, 1 \times 16 + 11)$ であるから，$(x_H \,|\, x_L, y_H \,|\, y_L) = (2 \,|\, 7, 1 \,|\, 11)$ と表せる．

5.2.2 実効アドレス空間へのマッピング

図 8 は 8×8 の 2 次元配列の順序付けを示している．8×8 の 2 次元配列について，$(x, y) = (0, 0)$ から $(7, 7)$ までの 64 個すべての要素を，0 から 63 まで 1 次元に順序付けしている．この順序付けは，全要素の順序を重複も欠落もなく一意に定める全単射の写像であるので，アドレス変換として利用することができる．

同図は，提案するアドレス変換によるボクセル・アドレスと実効アドレスのマッピングをも表している．2 次元配列がボリュームに，順序が実効アドレスにそれぞれ対応している．また，x，y 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割して表している．

提案するアドレス変換では，マッピングを以下のように行う．

キューボイドのサイズを $C_x \times C_y$ とする．

(1) ボクセル $(0 \,|\, 0, 0 \,|\, 0)$ を始点とする．
(2) y 軸方向に移動する．
(3) キューボイドの y 軸方向の境界 $(x, y) = (x_H \,|\, x_L, y_H \,|\, C_y - 1)$ に達すると y 軸方向に折り返し，$(x_H \,|\, x_L + 1, y_H \,|\, 0)$ に移動する．
(4) キューボイドの頂点，$(x_H \,|\, C_x - 1, y_H \,|\, C_y - 1)$ に達すると x 軸方向に折り返し，$(x_H \,|\, 0, y_H + 1 \,|\, 0)$ に移動する．
(5) ボリューム空間の端，$(x_H \,|\, C_x - 1, N - 1)$ に達すると y 軸方向に折り返し，$(x_H + 1 \,|\, 0, 0 \,|\, 0)$ に移動する．

図 8 を使い説明する．同図では，キューボイドのサイズを $C_x = C_y = 4$ としている．

$(0 \,|\, 0, 0 \,|\, 0)$ から出発し，(2) に従い y 軸方向に移動する．ボクセル $(x, y) = (0 \,|\, 0, 0 \,|\, 3)$ まで移動すると，次は (3) に従い $(0 \,|\, 1, 0 \,|\, 0)$ に移動する．

以下同様にし，ボクセル $(0 \,|\, 3, 0 \,|\, 3)$ まで移動すると，キューボイド $(x_H, y_H) = (0, 0)$ を構成する $4 \times 4 = 16$ ボクセルをすべて通ったことになる．これは，キューボイドが実効アドレス空間上の連続領域にマッピングされることを意味している．

$(0 \,|\, 3, 0 \,|\, 3)$ の次は，(4) に従い $(0 \,|\, 0, 1 \,|\, 0)$ に移動する．同様にキューボイド $(0, 1)$ の 16 ボクセルをジグザグに通り，$(x, y) = (0 \,|\, 3, 1 \,|\, 3)$ に移動する．(5) に従い $(1 \,|\, 0, 0 \,|\, 0)$ に移動し，同様にキューボイド $(1, 0)$，そして $(1, 1)$ を覆う．

$(x, y) = (1 \,|\, 3, 1 \,|\, 3)$ まで移動すると，キューボイド $(0, 0)$，$(0, 1)$，$(1, 0)$，$(1, 1)$ の順にすべてのボクセルを通ったことになる，つまり，ボリューム全体の実効アドレス空間へのマッピングが定まったことになる．実効アドレス空間では，各キューボイドは連続領域にマッピングされることになる．

前述したように，キューボイド順 RC 法では，1 つ

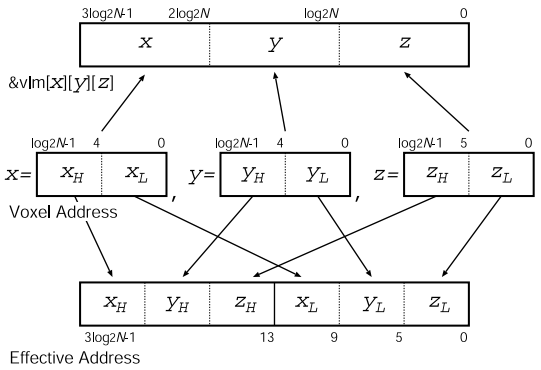

図 9 ボクセル・アドレスから実効アドレスへのアドレス変換
Fig. 9 Transformation from voxel address to effective address.

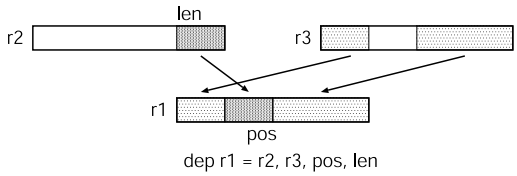

図 10 IA-64 の dep 命令
Fig. 10 IA-64 dep instruction.

のキューボイド内の SP を一気に処理する．そのため，1 つのキューボイドを処理している間は，実効アドレス空間上では，この連続する領域のどこかをアクセスすることになる．キューボイド順 RC 法のアドレスのアクセス順序は視点の位置に依存するため，連続領域内のキャッシュ・ラインがどのような順序でキャッシュされるかは分からない．しかし，この連続領域内のラインは，1 つのキューボイドの処理中には，たかだか 1 回キャッシュされることになる．

このことによって，従来手法の問題点は以下のように解決される：

(1) **ベクトル長** 提案手法では，キューボイドはその形状にかかわらず実効アドレス空間上の連続領域，すなわち，連続するキャッシュ・ラインにマッピングされる．したがって，キューボイドの形状 $C_x \times C_y \times C_z$ は，ライン長とは独立に，ベクトル長が長くなるように自由に選んでよい．

(2) **カラム競合** キューボイドのサイズは，定義からキャッシュ容量より小さい．通常のキャッシュ・システムでは，アドレスが連続するキャッシュ容量未満の領域をアクセスして，カラム競合が起ることはない．

なお，3 次元の場合についても同様の方法でマッピングする．

5.2.3 実効アドレスの計算

図 9 にアドレス変換を示す．従来の単純な 3 次元配列を用いた場合には，ボクセル・アドレス (x, y, z) に対して，実効アドレスは $x \,|\, y \,|\, z$ となる．同図下に示す提案するアドレス変換では，実効アドレスは，前述のキューボイド番号を上位に，キューボイド内オフセットを下位に集めた形となっている．すなわち，ボクセル・アドレス $(x, y, z) = (x_H \,|\, x_L, y_H \,|\, y_L, z_H \,|\, z_L)$ に対して，実効アドレスは $x_H \,|\, y_H \,|\, z_H \,|\, x_L \,|\, y_L \,|\, z_L$ とする．図 7 に示したボクセル・アドレス $(39, 27) = (2 \,|\, 7, 1 \,|\, 11)$ の例では，実効アドレスは $2 \,|\, 1 \,|\, 7 \,|\, 11$ となる．

5.3 実効アドレスの計算コスト

提案手法のアドレス変換では，ボクセル・アドレス (x, y, z) の x，y，z をそれぞれ分割した後，連結する必要があり，従来手法に比してその計算コストの増大が懸念される．しかし実際には，そのコストの増加は許容可能である．

従来手法における &vlm[x][y][z] のようなアドレス変換は，アドレス計算のコストを削減するのに都合がよい．一方，提案手法では，アドレス変換は x，y，z をそれぞれ $x_H \,|\, x_L$，$y_H \,|\, y_L$，$z_H \,|\, z_L$ と分割し，その後これら 6 つの値を連結する必要がある．そのため，従来手法で 3 つの値を連結すればよいのに比べて，計算量の増加は避けられない．

しかし，最近のプロセッサが備える暗号処理用の命令を用いれば，そのコストは小さく抑えることができる．たとえば，次章の評価で用いる IA-64 命令セットには，**図 10** に示す dep（Deposit：格納）命令がある．この命令を利用すれば，提案のアドレス変換を 8 命令で計算することができる．なおこの命令では，レジスタ r2 の値が右端になければならないため，dep 命令 6 個ではアドレス変換できないことに注意されたい；x_H，y_H，z_H を r2 にセットするには，x，y，z をそれぞれ右シフトする必要がある．次章の評価では，この命令を用いた結果を示す．

6. 性能評価

6.1 性能評価

Itanium2 サーバ上でプログラムを実装し，評価した．**表 1** にプロセッサの諸元を示す．コンパイラは gcc version 2.96 を用いた．オプションは-O4 である．この環境では，実効アドレスの計算式は，前述の dep 命令を含むコードに変換される．ただし，プログラムの最内側ループはアセンブリ・レベルのハンド・コーディングによりソフトウェア・パイプライニングを施

表 1 Intanium 2 の諸元
Table 1 Specifications of Itanium 2.

動作周波数		1 GHz		
浮動小数点命令同時発行数		2命令		
ピーク演算性能		2GMACS☆		
システム・バス・バンド幅		6.4 GB/s		
キャッシュ		L1D	L2	L3
サイズ		16 KB	256 KB	3 MB
ライン・サイズ		64B	128B	128B
ウェイ数		4	8	12
レイテンシ	INT	1	5	12
load to use (cycles)	FP	NA	6	12

☆ Multiply and ACcumulate per Second

表 2 平均ベクトル長と配列サイズ
Table 2 Average vector length and size of array.

	N	128	256	512	1024
$8\times8\times128$	平均ベクトル長 最良	128	128	128	128
	平均ベクトル長 最悪	5.66	5.66	5.66	5.66
	配列サイズ（MB）	2.00	24.1	160	1409
$16\times16\times32$	平均ベクトル長 最良	32.0	32.0	32.0	32.0
	平均ベクトル長 最悪	11.1	11.1	11.1	11.1
	配列サイズ（MB）	2.00	16.0	128	1024

している．

従来手法と提案手法の評価を行う．従来手法では，ボリュームは整合配列により定義しており，キューボイドの形状は $8\times8\times128$ である．一方，提案手法における形状は，4.1 節で述べた最適な形状 $16\times16\times32$ である．

6.1.1 平均ベクトル長およびボリュームの配列サイズ

表 2 に各手法の最良，および，最悪の場合の平均ベクトル長，および，ボリュームの配列サイズを示す．同表上側の欄が従来手法，下側が提案手法である．また，4.1 節で述べたように，平均ベクトル長が最大になる場合が最良，最小となる場合が最悪の場合である．

平均ベクトル長は，N によらず，キューボイドの形状と視点位置で決まるため，すべての N について値は一定となっている．また，提案手法により最良の場合については短縮しているが，最悪の場合については改善されている．

配列サイズについては，従来手法では $N=128$ の場合を除き，整合配列を用いているために配列サイズが N^3 より大きくなっているが，提案手法ではすべての N について配列サイズは N^3 である．

6.1.2 実効 FLOPS 値

プログラムを実行し，実効 FLOPS 値を計測した．結果を**表 3** に示す．従来手法，提案手法ともにカラム競合は発生せず，性能は平均ベクトル長でのみ決まることが確認できる．提案手法における最良，および最悪の場合のピーク性能（4GFLOPS）に対する割合を求めると，それぞれ 22.0%，17.1% であった．

表 3 実効 MFLOPS 値
Table 3 Effective MFLOPS value.

	N	128	256	512	1024
$8\times8\times128$	最良の場合	1024	991	987	986
	最悪の場合	566	523	522	517
$16\times16\times32$	最良の場合	887	877	876	877
	最悪の場合	683	678	683	685

従来手法に対する，提案手法による性能の変化を求める．前述のように，評価すべきは最悪の場合における性能である．評価結果から最悪の場合の性能向上を求めると，28.2%である．最良の場合については性能が低下しているが，これは許容される．

6.2 ループの最大性能

今回実装したプログラムの最内側ループの最大性能を求める．ループ内の演算数は，整数への変換を含め 13(FLOP) であり，また，投入間隔は 5 サイクルである．したがって，ベクトル長が無限長のときの性能は 13(FLOP) / 5 (サイクル) × 1(GHz) = 2.6(GFLOPS) となる．

提案手法における最良，および最悪の場合の，ループの最大性能（4GFLOPS）に対する割合を求めると，それぞれ 33.8%，26.2% である．

6.3 提案手法の応用

提案手法の応用について考察する．以下に示す理由から，提案手法はボリューム・レンダリングに特化した手法であり，応用範囲は狭いと考える．

提案手法において，問題となるのがアドレス計算のコストである．dep 命令のようなビット操作命令を用いることでコストを小さくできるが，それでも不十分である．

ボリューム・レンダリングにおいては，以下の 2 つの理由によりこのコストを隠蔽することができるため，提案手法を適用することができる．

理由 1 アクセス・パタンが未知である

一般的なループ・プログラムではアクセス・パタンが既知であり，アドレス計算はストライドを加算することで行えることが多い．これが可能なのは，仮想空間（たとえば，ボリューム空間）とアドレス空間の対応が比較的単純であることによる．提案手法においては，これらの間のマッピングが複雑化しており，たとえば，ボリュームに単純に等間隔にサンプリングする場合でも，アクセスごとに SP の座標を用いてアドレスを計算する必要がある．このコストは大きく，かえって性能を低下させかねない．

一方，アクセス・パタンが未知なボリューム・レンダリングでは，図 1 に示したように，まず SP の座標が浮動小数点数として求められるため，それを整数に変換した後にアドレスを計算する必要がある．したがって，従来手法における &vlm[x][y][z] の計算は，ストライドの加算ではなく，x，y，z の 3 つの値を連結する操作が必要となる．つまり，ボリューム・レンダリングでは，従来手法におけるアドレス計算のコストがもともと大きいため，提案手法によるコストの増大は，比較的小さい．

理由 2 計算量が大きい

一般の数値処理において，メモリ・アクセス 1 回あたりの計算は，たかだか積和演算（2 FLOP）程度であるのに対し，ボリューム・レンダリングでは 13 FLOP と非常に大きい．計算量が大きいプログラムでは，つまり，ALU への要求が FPU へのそれより小さいプログラムでは，アドレス計算を隠蔽することができる．逆に，計算量が小さい場合は隠蔽できず，性能の低下につながる可能性がある．

まとめると，提案手法はアクセス・パタンが未知であり，計算量が大きいアプリケーションについてのみ適用可能であり，その応用範囲は狭いといえる．

7. おわりに

本稿では，キューボイド順 RC 法におけるボリューム空間のアドレス——ボクセル・アドレスから実効アドレスへのアドレス変換を提案した．これにより，従来のアドレス変換 &vlm[x][y][z] において互いに競合していた問題，(1) ベクトル長の短縮 と，(2) カラム競合対策 の 2 つを同時に解決することが可能となった：ベクトル長は，キューボイド形状を立方体にすることで最大化され；メモリ・アクセスは実効アドレスの連続する領域内に対してのみなされるようになり，カラム競合が起こることはなくなった．

Itanium2 サーバ上でプログラムを実装し評価した結果，提案手法により描画性能は 28.2%向上することが分かった．これは，キューボイド形状を立方体にすることで，最内側ループのベクトル長が改善されたことによる．また，従来はカラム競合を避けるため整合配列を利用していたが，配列の定義に必要な N^2 の無駄な領域が削減された．

従来の実効アドレスの生成方法が，ボクセル・アドレス x，y，z の連結であるのに対し，提案手法では，ビット列のシャッフルを行うため計算量は大きくなってしまう．しかし，最近の汎用プロセッサが備える暗号処理用のビット列操作命令を用いれば，そのコストは十分に小さく抑えることができる．将来，このような暗号処理用の命令の普及により，汎用 PC 上におけるボリューム・レンダリングは高速化されるだろう．

本研究の一部は，文部省科学研究費補助金，基盤研究(B)(2)#13480083，日本学術振興会科学研究費補助金基盤研究 S（課題番号 16100001），21 世紀 COE プログラム（課題番号 14213201），ならびに，文部科学省特定領域研究 S（課題番号 13224050）による．

参 考 文 献

1) Foley, J.D., van Dam, A., Feiner, S.K. and Hughes, J.F.: *Computer Graphics: Principles and Practice*, Ohmsha (2001).
2) Lichtenbelt, B., Crane, R. and Naqvi, S.: *Introduction to Volume Rendering*, Hewlett Packard (1998).
3) 金 喜都，對馬雄次，中山明則，森眞一郎，富田眞治：視覚制限ピクセル並列処理によるボリューム・レンダリング向きの超高速専用計算機のアーキテクチャ，情報処理学会論文誌，Vol.38, No.9, pp.1668–1680 (1997).
4) 對馬雄次，中山明則，荻野友隆，金 喜都，森眞一郎，中島 浩，富田眞治：ボリュームレンダリング専用並列計算機—*ReVolver*/C40，並列処理シンポジウム JSPP'95, pp.11–18 (1995).
5) Lacroute, P. and Levoy, M.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, *SIGGRAPH'94*, pp.451–458 (1994).
6) Avila, R., He, T., Hong, L., Kaufman, A., Pfister, H., Silva, C., Sobierajski, L. and Wang, S.: A Diversified Volume Visualization System, *IEEE Visualization 1994* (1994).
7) Bakalash, R., Kaufman, A., Pacheco, R. and Pfister, H.: An Extended Volume Visualization System for Arbitary Parallel Projection, *7th Workshop on Graphics Hardware*, Vol.EG92 HW (1992).
8) Cabral, B., Cam, N. and Foran, J.: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware, *IEEE/ACM Symposium on Volume Rendering*, pp.91–98 (1994).
9) Drebin, R.A.: Volumetric Rendering of Computed Tomography Data: Principles and Techniques, *IEEE Computer Graphics and Applications*, No.10(2), pp.24–32 (1990).
10) Elvins, T.T.: A Survey of Algorithms for Volume Visualization, *Computer Graphics*, No.26(3), pp.194–201 (1992).
11) Wolfe, M.: More Iteration Space Tiling, *Proc. Supercomputing* (*SC'89*), pp.655–664 (1989).

12) 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，情報処理学会論文誌：コンピューティングシステム，Vol.44, No.SIG 11(ACS 3), pp.137–146 (2003).
13) 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，先進的計算基盤システムシンポジウム SACSIS 2003, pp.333–340 (2003).
14) de Boer, M., Hesser, J., Gropt, A., Gunther, T., Poliwoda, C., Reinhart, C. and Manner, R.: Evaluation of a Real-Time Direct Volume Rendering System, *11th Workshop on Graphics Hardware*, Vol.EG96 HW, pp.109–119 (1996).

(平成 16 年 1 月 31 日受付)
(平成 16 年 5 月 21 日採録)

額田　匡則（学生会員）

1978 年生．1994 年岡山大学付属中学校卒業．1997 年岡山県立岡山大安寺高校卒業．2001 年京都大学工学部情報学科卒業，同年より同大学大学院情報学研究科修士課程．ボリューム・レンダリングの研究に従事．2003 年先進的計算基盤システムシンポジウム優秀学生論文賞受賞．2004 年京都大学大学院情報学研究科修士課程修了，同年より同大学院情報学研究科博士後期課程．

小西　将人（学生会員）

1976 年生．1992 年綾部市立八田中学校卒業．1995 年京都府立綾部高校卒業．1999 年京都大学工学部情報学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了，同年より大阪工業大学情報学科講師．並列計算機アーキテクチャの研究に従事．

五島　正裕（正会員）

1968 年生．1992 年京都大学工学部情報工学科卒業．1994 年同大学大学院工学研究科情報工学専攻修士課程修了．同年より日本学術振興会特別研究員．1996 年京都大学大学院工学研究科情報工学専攻博士後期課程退学，同年より同大学工学部助手．1998 年同大学大学院情報学研究科助手．高性能計算機システムの研究に従事．2001 年情報処理学会山下記念研究賞，2002 年同学会論文賞受賞．IEEE 会員．

中島　康彦（正会員）

1963 年生．1986 年京都大学工学部情報工学科卒業．1988 年同大学大学院修士課程修了．同年富士通（株）入社．スーパコンピュータ VPP シリーズの VLIW 型 CPU，M アーキテクチャ・命令エミュレーション，高速 CMOS 回路等に関する研究開発に従事．工学博士．1999 年京都大学総合情報メディアセンター助手．同年同大学大学院経済学研究科助教授．計算機アーキテクチャに興味を持つ．2002 年情報処理学会論文賞受賞．IEEECS, ACM 各会員．

富田　眞治（正会員）

1945 年生．1973 年京都大学大学院博士課程修了，工学博士．同年京都大学工学部情報工学教室助手．1978 年同助教授．1986 年九州大学大学院総合理工学研究科教授．1981 年京都大学工学部情報工学科教授．1998 年同大学大学院情報学研究科教授．計算機アーキテクチャ，並列計算機システムに興味を持つ．情報処理学会論文賞受賞（1987 年，1992 年，2002 年）．電子情報通信学会，情報処理学会フェロー．著書『並列計算機構成論』,『並列処理マシン』,『コンピュータアーキテクチャI』等．電子情報通信学会，IEEE，ACM 各会員．