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

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

従来主に用いられてきたピクセル順のレイ・キャスティング法では，視点の位置によってキャッシュ・ヒット率が著しく低下する．本稿で提案する手法では，ボリューム空間を複数の直方体，キューボイドに分割し，キューボイド順に処理を行うことによって，ボリュームへのアクセス・パタンを制御し，視点の位置によらず，キャッシュ・ヒット率を最大化することができる．プログラムを実装し，評価した．従来方式では，最悪の場合，最良の場合の 6 倍もの時間がかかる．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ることが分かった．

A Volume Rendering Algorithm for Maximum Spatial Locality of Reference

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

The pixel-order ray-casting algorithm suffers from low cache hit ratio depending on the position of the viewpoint. In this paper we propose a cuboid-order algorithm, which divides the volume space into cuboids and controls the access pattern to the volume data by rendering each cuboid. The algorithm achieves the maximum cache hit ratio independent of the viewpoint position. The evaluation result shows that the worst case of the pixel-order algorithm takes six times as long as the best case, while our algorithm 1.15 times as the best case of the pixel-order algorithm independent of the viewpoint position.

1. はじめに

ボリューム・レンダリングとは，ボリュームと呼ぶ，半透明な 3 次元のオブジェクトを，ポリゴンに変換したりしないで，直接 2 次元画像に変換する方法の総称である．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．本稿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 (voxel) 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，その計算量のため，CPU，あるいは，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．

レイ・キャスティング法のアクセス・パタン

正方格子のボリューム・レンダリングでは，ピクセル順 (pixel-order) のレイ・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 法では，視点からスクリーン上のあるピクセルにキャストされた視線 (レイ) 上のサンプリング点にあるボクセルの値を順にサンプリングしてそのピクセルの値を求める．ピクセル順 RC 法では，各ピクセルの値を 1 つずつ順に求め，1 枚の画像を得る．

この時，プログラムの最内側ループでは，1 本の視線上のボクセルを順にアクセスすることになる．そのためボリュームへのアクセスは一般に，ほぼ等間隔なアクセスになるうえ，そのストライドは視点の位置によって動的に決まる．

ストライドの変化に対して，その最大の供給能力を安定して提供できるメモリを構築する手法は現在のところ知られていない．そのため従来では，メモリ・バンクへの実際のアクセスを連続化する手法の開発に主眼が置かれていた．例えば，Shear-Warp 法[1] では，連続アクセスになるようにボリュームを並べ替える．専用計算機 *ReVolver* では，ボリューム・メモリを三重化し，視点の位置によってアクセスするメモリを切り替えている[2]．また，専用アクセラレータ・ボード VolumePro[3]

† 京都大学 Kyoto University

や，金らの専用計算機[4] などでは，投影方法や視野角を制限することによって，連続アクセスにならない状況を避けている．

レイ・キャスティング法とキャッシュ・メモリ

一方，PC やそのグラフィクス・カードなど，最近の汎用プラットフォームでは，デバイス技術の進歩によって，ボリューム・レンダリングに必要となる高い計算能力が安価に入手できるようになってきている．最新の CPU や GPU は，256^3 ボクセルからなるボリュームをリアルタイムに描画する演算能力を有している．

これらの CPU や GPU では，メモリ・アクセスの高速化技術として，キャッシュが不可欠なものとなっている．そのため，キャッシュを有効に利用する高速化手法が強く望まれる．

しかし前述したのと同様の理由により，キャッシュも RC 法に対しては有効に働かない．RC 法には，利用可能な参照の局所性がほとんどないためである．ボリューム・レンダリングでは，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 RC 法では，アクセス・ストライドが一般にキャッシュ・ブロック・サイズを越えるため，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ブロック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．5 章で述べる評価では，キャッシュが有効に機能しない場合，有効に機能する場合の 6 倍もの時間がかかることが示される．

現在の汎用グラフィクス・カードでは，たしかに，この問題はそれほど顕著ではない．しかしそれは，GPU のクロック速度は CPU の 1/5 ~ 1/6 程度であるのに，ボリュームを格納する VRAM に CPU の主記憶と同等のテクノロジを用いているためである．現在のところ，別段対策を講じなくても，VRAM は GPU の処理速度に見合うスループットを辛うじて提供している．しかし近い将来，CPU の場合と同様に，GPU の処理能力の向上に VRAM の供給能力が追い付かなくなることは確実である．

提 案 手 法

本稿では，タイリング[5] と同様の考え方によって，ボリュームへのアクセス・パタンを制御する手法を提案する．本手法では，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補償する今日の汎用プラットフォームであっても，メモリ・システムに起因する速度低下がほとんど無視できるようになる．

以下では，まず 2 章で RC 法について述べた後，3 章と 4 章で提案手法について説明する．5 章では，提案手法を Itanium プロセッサに実装し評価した結果について述べる．

そのため本稿では，Itanium のような汎用 CPU への実装を例に，アルゴリズムの説明を行っている．しかし，参照の空間局所性を最大化するという提案手法の骨子は実行するプラットフォームにに依存しないものであり，汎用グラフィクス・カードの GPU や，専用計算機への実装もまた可能であることに注意されたい．

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 法について説明し，そのメモリ・アクセスの性質について明らかにする．

2.1 ピクセル順レイ・キャスティング法

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，ボクセル値　構造体 Pixel，Voxel では，RGB の 3 色と透明度を float で表している．

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 1B 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．

コード全体は 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　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．

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．

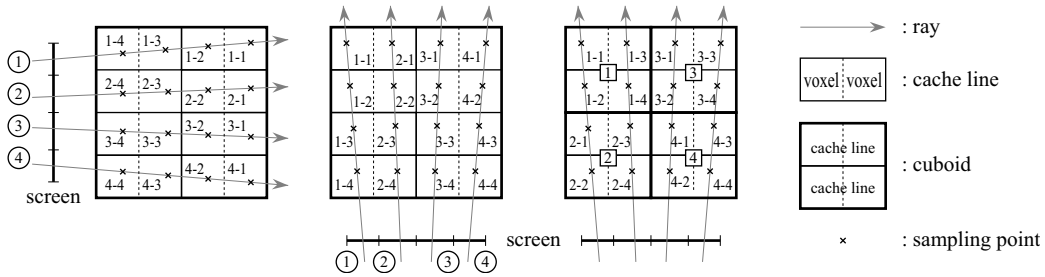

図 2　視点の位置と SP の処理順序

```
struct Pixel { float r, g, b; };          // ピクセル値
struct Voxel { float r, g, b, a; };       // 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 v = 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
        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
        float a = 1.0F − t;               // 不透明度
        r = t ∗ r + a ∗ vxl.r;
        g = t ∗ g + a ∗ vxl.g;
        b = t ∗ b + a ∗ vxl.b;

        // (4) SP の更新 (3FLOP)
        x −= dx;  y −= dy;  z −= dz;
    }
    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

2.2　レイ・キャスティング法のメモリ・アクセス

RC 法は，3 重のループからなる比較的単純なプログラムであるが，メモリへのアクセス・パタンは，通常の数値処理などと比べるとかなり複雑である．

メモリに対する要求バンド幅

図 1 のコードから，1 回のサンプリング，すなわち，最内側ループ 1 イタレーションあたりの演算回数は，浮動小数点数から整数への変換を含めて，16FLOP になることが分かる．1 ボクセルを 1B とすると，メモリに対する要求バンド幅，すなわち，1FLOP あたりのデータ転送量は 1/16B/FLOP となる．

一方，例えば内積計算などでは，要求バンド幅は 2word/2FLOP = 4 ~ 8B/FLOP である．したがって，単純に量だけを比較すれば，RC 法は，通常の数値処理に比べて，メモリに対する要求が極めて低い，計算バウンドな処理であると言える．

キャッシュとの親和性

N^3 ボクセルのボリュームを N^2 ピクセルのスクリーンに投影する場合，1 つのボクセルは平均 1 回サンプリングされるだけである．すなわち，ボリュームに対する参照には，本来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 を処理する時には，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 ラインの内の 1B しか利用できないことになる．

この場合，例えばライン・サイズを 128B 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 $1/16 \times 128 = 8$(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．

3.　キューボイド順レイ・キャスティング法

通常の数値処理の中には，タイリング[5] 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．

本稿で提案する手法は，タイリングと同様の考え方によって，RC 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 RC 法では，視点の移動に伴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，通常の数値処理などに対するように，コードを静的に変換することはできない．動的に決定されるアクセス・パタンをどのように制御するかが，提案手法のポイントとなる．

3.1 キューボイド順

RC 法のアクセス・パタンを制御する目的は，通常の数値処理とは若干異なる．前述したように，ボリュームへの参照は本質的に時間局所性を持たない．したがって，視点の位置に関わらず，空間局所性を最大化することが目的となる．これは，あるキャッシュ・ラインをフェッチしたときに，そのライン内のすべての SP を処理しつくすことによって達成される．ボリュームへの参照は本質的に時間局所性を持たないから，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．RC 法は本来計算バウンドな処理であるため，それだけで十分な性能が期待できる．

ただし，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，ラインを数十個程度束ねたキューボイド (cuboid: 直方体) を単位とする．キャッシュ・ラインは，そのサイズを L(B) とすると，ボリューム内では $1 \times 1 \times L$ の細長い拍子木状の空間を占める．その理由は後述するが，キューボイドはこの細い拍子木を横に束ねて，$C \times C \times L$ の太めの拍子木状とする．5 章の評価では，$8 \times 8 \times 128 = 4$K(B) としている．

RC 法におけるボリュームへのアクセス・パタンは，次の条件を満たす SP の処理順序として与えられる：

条件 **1**　1 つの視線上の SP は，順に処理する．

ピクセル順 RC 法では，各ピクセルに対する視線上の SP をスクリーン奥から順に連続して処理することで，この条件を満たしている．提案手法では，更に，以下の条件を加える：

条件 **2**　キューボイド内の SP は，連続して処理する．

提案手法は，ピクセル順に対して，キューボイド順 **RC** 法と呼ぶことができる．キューボイド順 RC 法では，まずキューボイドの処理順序を定め，各キューボイドに対してその内部のすべての SP を処理することになる．例えば，図 2 右の例では，① ~ ④ の順序でキューボイドを選択し，その中の SP を連続して処理している．キューボイドがキャッシュ・サイズより小さく，ライン競合が起こらなければ，未参照のボクセルを含むラインがリプレースされることはなく，視点が縦方向の場合であっても，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 RC 法の実装にあたっては，以下の点に注意する必要がある：

(1) ピクセル値計算の中断　あるキューボイドから別のキューボイドに処理を移す時には，ピクセル値の計算を一旦中断する必要がある．図 2 右の例では，ピクセル① の計算を SP 1-1，1-2 の処理を行ったところで中断し，ピクセル② に対して SP 1-3 の処理を開始している．ピクセル① の計算は，SP 2-1 で再開される．このようなピクセル値計算の中断と再開は，低コストで実現可能か．

(2) キューボイドの処理順序　条件 1 を満たすキューボイドの処理順序を効率よく発見できるか．

(3) キューボイド内の **SP** の処理　条件 1 を満たした上で，キューボイド内のすべての SP を効率よく処理することができるか．

(4) キューボイドの形状　アクセス・パタンを制御するためのオーバヘッドはどの程度か．オーバヘッドを最小化するには，キューボイドの形状をどのように定めたらよいか．

以下本章では，3.2 節と 3.3 節において，上記 (1)，(2) ついて説明する．性能上最も重要である (3) については，章を改めて，4 章で詳しく説明する．5 章では，提案手法の評価について述べるが，(4) については，そこでまとめる．

3.2 ピクセル値計算の中断

結論から言えば，ピクセル値の計算は任意の時点で中断が可能である．図 1 のコードの最内側ループの終了条件を書き換えて，SP がキューボイドを外れた時点で最内側ループを終了するようにすると，その時点でのピクセル値計算の途中結果 (r, g, b) は，pxl[u][v] に保存される．このピクセルの値の計算を再開する時には，pxl[u][v] の値を (r, g, b) に読み込めばよい．

一方，SP を表す (x, y, z) は，一時変数として用意されているため，このままでは中断することができない．pxl と同様，Vector sp[N][N] として，各ピクセルごとの配列を静的変数として用意し，中断時に現在の値を保存する必要がある．

また視線ベクトル (dx, dy, dz) は，効率のため，sp と同様に Vector rv[N][N] と，各ピクセルごとの配列を静的変数として用意するとよい．予めすべての視線ベクトルを求めておくことで，ピクセル値計算の再開の度にその視線ベクトルを再計算することを避ける．

sp[N][N] と rv[N][N] は，併わせて $24N^2$B となる．これは，ボリューム vlm[N][N][N] の $24/N$ にあたり，$N = 128$ で 19%，$N = 256$ で 9%になる．

このデータ量は，N が大きければ問題にならないが，無視できるほどでもない．ただし，これらの配列への参照に対しては，キューボイドの処理順序を工夫することによって，時間局所性を抽出することができる．次節では，キューボイド間の処理順序について述べる．

3.3 キューボイドの処理順序

すべての SP をキューボイド順に処理するためには，前節で述べたようにピクセル値計算の中断が可能であることに加えて，前述した条件 1 を満たす必要がある．

すなわち，すべての視線に対して，その視線が通過するキューボイドは，スクリーン奥にあるものから順に処理しなければならない．

このことは，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．図 3 の関数 loop_x は，キューボイドの x 軸方向の番号 x の順序を決定する．この関数中の文字 x を y，z に書き換えた関数 loop_y，loop_z を次々呼び出すことによって，キューボイドの処理順序が決定される．

図 **4** に示す 2 次元のボリュームを例に，図 3 のコードの動きを説明しよう．コード中の 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 4 では 2 である．同様に cy は 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 0，1，3，2 と決まる．それぞれに対して，loop_y によって y 軸方向の処理順序が 0，1，2，3 と決まる．結局図 4 のキューボイドは，1-1，1-2，... ，4-4 の順に処理される．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．

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．すなわち，図 3 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 4 のキューボイドは，1-1，2-1，3-1，4-1，1-2，... ，4-4 の順に処理され，やはり正しく動作する．

しかし，図 4 の場合では，先に示したとおりの xy 型の方が性能がよい．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，ピクセル値ごとの配列 pxl[N][N]，sp[N][N]，rv[N][N] に対する参照の時間局所性が高まるからである．

軸間の順序を選ぶには，視点からボリュームの中心に向かうベクトルの x，y，z 各要素の絶対値を比較し，小さい要素から順に外側から内側へと並べればよい．図 **5** に，スクリーンに対する 3 次元ボリュームの投影の例を示す．同図中，視点は，$x = 2$，$y = 1$ のキューボイドの内部にある点+に投影されている．同図の場合，yxz 型が選択される．最内側ループでは，z 軸方向，すなわち，画面の奥から手前方向に処理が進む．その外側，x，および，y 軸の方向では，1-1，1-2，... ，4-4 の順に処理が進む．

```
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();
}
```

図 **3**　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

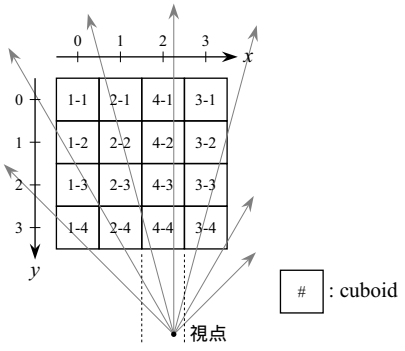

図 **4**　図 **3** のコードによるキューボイドの処理順序

4. キューボイドの処理

図 **6** に，キューボイドに対する処理の概略を示す．キューボイドを構成する 6 つの面は，スクリーン側に見えている可視面と，可視面によって隠される隠面に分けられる．キューボイドを通過する視線は，可視面のうちの 1 つ，および，隠面のうちの 1 つと，それぞれ交わる．処理すべき SP は，これらの 2 つの交点の間にある．

ボリュームに対するキャッシュ・ヒット率は，キューボイド順に処理を進めることによって最大化することができる．したがって，キューボイド内の SP の処理順序は，ヒット率とは無関係に定めてよい．後述する理由により，キューボイド内の SP はピクセル順に処理する．

キューボイド内の SP をピクセル順に処理するため，図 1 に示したピクセル順 RC 法のコードを一部流用す

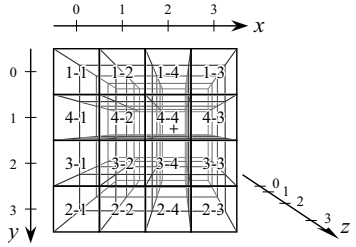

図 **5**　視点の位置とキューボイドの処理順序

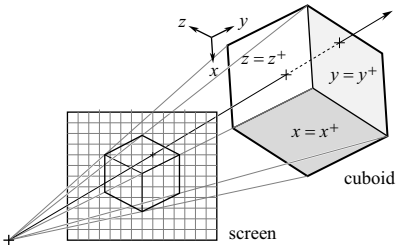

図 **6**　キューボイドの投影

ることができる．図 1 のコードでは，外側の u，v の 2 重のループにおいて計算すべきピクセルを選択し，最内側のループがそのピクセル値を計算していた．キューボイドに対しても，この基本的な骨格は変わらない．以下，4.1 節と 4.2 節において，キューボイドに対する外側と内側の処理について詳しく述べる．また，4.3 節でキューボイドの形状，サイズと配置について議論する．

4.1 ピクセルの選択

処理すべきピクセルは，それへの視線がキューボイドを通過するようなピクセルすべてである．このようなピクセルは，ピクセル順 RC 法の場合と異なり，単純なループによって得ることはできない．処理すべきピクセルは，キューボイドをスクリーンに投影することによって得られる．キューボイドのスクリーンに対する射影の内部のピクセルに対する視線は，当然のことながら，このキューボイドを通過する（図 6）．

4.1.1 可視面と隠面

キューボイドを投影する際には，ピクセルを 2 重に列挙することを避けるため，まず可視面と隠面を区別する必要がある．提案手法では，投影の対象がキューボイド——直方体であることが分かっているので，Z 値の比較などの複雑な計算は必要なく，以下のように，ほとんどオーバヘッドなしに区別することができる．

キューボイドを構成する 6 面は，方程式 $x = x^-, x = x^+, y = y^-, y = y^+, z = z^-, z = z^+$ によって与えられる．ただし，$x^- < x^+, y^- < y^+, z^- < z^+$ とする．

可視面と隠面は，図 3 に示したコードによってキューボイドを選択していく過程で自動的に判別される．図 3 のコードは，x と cx の大小関係により，大きく 3 つの領域に分かれている．x に関する可視面と隠面は，キューボイドがどの領域に属するかよって決まる．領域 x < cx では $x = x^+$，領域 x > cx では $x = x^-$ がそれぞれ可視面であり，領域 x = cx では両方ともが隠面となる．y，z についても同様に求められる．図 5 では，視点は $(2, 1, > 3)$ にある．例えば，キューボイド $(0, 0, 3)$ は，x，y，z の各要素が視点のそれらよりそれぞれ小さいから，$x = x^+$，$y = y^+$，$z = z^+$ の 3 面が可視面であることが分かる．

このように，キューボイドを選択する過程で判定できるので，あるキューボイドの処理を開始する時には，既にその可視面と隠面が分かっていることになる．判定のための特別な処理は必要ない．

4.1.2 キューボイドの投影

ピクセルの列挙は，前項で得られた可視面を投影することによって行う．この処理は，投影する面を長方形のポリゴンとするポリゴン・レンダリングと等価である[6]．すなわち，面の頂点を座標変換して得られるエッジをスキャン変換することによって，求めるべき四辺形内部のピクセルをすべて列挙することができる．

この時，どの頂点，どのエッジが，どの面間で共有されているか分かっているため，共通部分の計算を省略することができる．

このように列挙されたピクセルのそれぞれに対して，最内側ループでその値を計算する．

4.2 ピクセル値の計算

最内側ループでは，その外側で選択された視線に対して，その視線と可視面/隠面との交点間にある SP を，スクリーン奥から順に処理することになる．最内側ループは，(1) 3.2 節で述べたピクセル値計算の中断と再開に対応すること，(2) 始点と終点が異なること の 2 点を除き，図 1 のコードをほぼそのまま流用することができる．

4.2.1 ピクセル値計算の中断と再開

ピクセル値計算を中断/再開するためには，3.2 節で述べたように，配列 sp[u][v]，rv[u][v] を用意する．最内側ループでは，その開始前と終了後に，pxl[u][v]，sp[u][v]，rv[u][v] への読み出しと書き戻しをそれぞれ 1 回行う．最内側ループ内部は，図 1 のままでよい．キューボイド内の処理をピクセル順とするのは，この，中断と再開のためのロード/ストアの回数を最小化できるためである．

4.2.2 始点と終点

ループの始点/終点は，視線と隠面/可視面との交点によって決まる．交点を求めるには，一般には，視線がキューボイドのどの面と交わるかを判定した上で，その平面と視線の方程式を解く必要がある．しかし提案手法では，以下で述べるように，始点/終点のいずれに対してもそのような処理は必要ない．

始　　点

pxl[u][v] に対する最内側ループが終了するとき，次の SP が sp[u][v] に保存される．したがって，その手前にあるキューボイドにおいて pxl[u][v] の計算を再開する時には，単にこの sp[u][v] を読み出すだけで，最初の SP の座標を得ることができる．

終　　点

最内側ループは，次の SP が処理中のキューボイドから外れていたら終了する．提案手法では，SP (x, y, z) のうちのいずれか 1 つとループ不変数との比較によって，この判定を行うことができる．

現在処理中の視線は，前項で述べたように，ある可視面を通過する視線を列挙する過程で選択されたものである．したがって，その視線が交わる可視面は既に分かっている．

図 6 のように，それが $z = z^+$ 面であったとしよう．この式と視線の方程式を解くまでもなく，z > z^+ ならば SP はキューボイドを外れたことが分かる．

ピクセル順でも，図 1 のコードで IS_IN_VOLUME() と示れているように，SP がボリュームから外れたことを検出する必要がある．その処理量は，提案手法と変わらない．

4.2.3 ピクセル順との比較

以上述べてきたように，(1) ピクセル値計算の中断/

再開，(2) 始点/終点 のそれぞれに関して，最内側ループのコード自体は図 1 に示したピクセル順のものをほとんど変更なしに流用できることが分かる．

ただし，キューボイドはボリューム全体に比べて小さいため，提案手法では，最内側ループのベクトル長が短くなることが避けられない．

4.3 キューボイドの形状，サイズと配置

結局，ピクセル順に対する提案手法の性能低下要因は，前々節で述べた (1) キューボイドの投影処理 と，前節で述べた (2) 最内側ループのベクトル長の短縮 の 2 点である．本節では，このことを踏まえて，キューボイドの形状について議論し，合わせて，キューボイドのサイズと配置について述べる．

キューボイドの形状

キューボイドの投影におけるスキャン変換の処理量は，スキャンラインの数，すなわち，キューボイドの射影のスクリーン縦方向の長さに比例する．したがって，投影の処理量を削減し，また，最内側ループのベクトル長を長くするために，キューボイドは，投影面積が小さく，視線方向に長い形状が望ましい．

そのため，視点の位置によってキューボイドの形状を変化させることも考えられる．そうしない場合，視点の移動に対して安定した性能を得るため，立方体に近い形状が望ましい．

多階層キャッシュとキューボイドのサイズ

キューボイドのサイズは，最小のキャッシュ・サイズ，すなわち，1 次キャッシュ・サイズの 1/2 ~ 1/4 程度とする．また，キューボイドの，アドレスが連続する方向——z 軸方向の辺の長さは，最大のライン・サイズ，すなわち，最高次キャッシュのライン・サイズの倍数とする．これらによって，各キャッシュ階層において，容量の不足によって必要なラインがリプレースされないことが保証される．

キャッシュとキューボイドの配置

ライン競合によるキューボイド内のラインのリプレースを避けるためには，整合配列を用いればよい．ピクセル順では，整合配列を用いたとしても，競合が起こる視点の位置が必ず存在する．キューボイド順では，1 つのキューボイドを処理している間に，そのキューボイド内のラインが互いに競合を起こさなければよい．1 つのキューボイドに属するラインは静的に決まっているため，そのような整合配列は容易に見つけることができる．

5. 性 能 評 価

本章では，まず 5.1 節で投影処理の処理量を机上で求める．その結果を踏まえて，最後に，5.2 章で，実機による評価結果について述べる．

5.1 投影処理の処理量

ボリューム空間を N^3 ボクセル，スクリーンを N^2 ピクセル，視線 1 本あたりのサンプリング回数を N 回とする．簡単のため，キューボイドのサイズを $C^3 = (N/n)^3$ ボクセルとする．キューボイドの数は n^3 個となる．

座標変換とスキャン変換における，1 キューボイドあたりの計算量は以下のようになる：

座標変換　同次行列との行列—ベクトル積と同次化のため，1 頂点あたり 20FLOP かかる．
1 つのキューボイドは 8 つの頂点を持つが，そのうち 4 つは直前に処理したキューボイドで求めた結果を再利用することができる．したがって，$20 \times 4 = 80$FLOP となる．

スキャン変換　キューボイドの対角線の長さ l は $l = \sqrt{3}C$ であり，これがスクリーン垂直方向に平行投影されるとき，スキャンラインが最も多くなる．
この時可視面は，垂直方向の長さが $l/2$ のエッジ 9 本からなる．エッジとスキャンラインの交点は，1 キューボイドあたり $9l/2$ (個) となる．
頂点のソートのため，最大 27FLOP かかる．
エッジの傾きを求めるため，1 エッジあたり 3FLOP，1 キューボイドあたり $3 \times 9 = 27$FLOP かかる．
エッジとスキャンラインの交点の x 座標を求めるため，1 交点あたり 2FLOP，1 キューボイドあたり $2 \times 9l/2 = 9l = 9\sqrt{3}C$ FLOP かかる．

以上をまとめると，投影の処理量は，全体で $(134 + 9\sqrt{3}C) \times n^3$ FLOP となる．

さて，ピクセル値計算の計算量は 1SP あたり 16FLOP (2 章参照)，全体で $16N^3 = 16n^3C^3$FLOP である．これに対する投影処理量の割合は $(134 + 9\sqrt{3}C) \times n^3 \div 16n^3C^3 = 134/16C^3 + 9\sqrt{3}/16C^2$ となる．これは，キューボイドのサイズを $C^3 = 4$KB，$C = 16$ とした場合，0.5%に過ぎない．

提案手法の，ピクセル順 RC 法に対する主な性能低下要因は，(1) キューボイドの投影処理 と (2) 最内側ループのベクトル長の短縮 であると述べたが，このうち (1) 投影処理のオーバヘッドはほとんど無視できる．

5.2 実機による評価

ピクセル順，および，キューボイド順 RC 法のプログラムを Itanium 2 サーバに実装し評価した．Itanium 2 の諸元を表 **1** に示す．

プログラム

ピクセル順は図 1 のコードを，キューボイド順は，それに 4.2 節で述べた変更を加えたものを用いた．ループ・アンローリング，ソフトウェア・パイプライニングなどの最適化は施していない．GCC 2.96 を用いてコンパイルした．最適化オプションは-O4 である．

キューボイドの形状

Itanium 2 は，最高次 (3 次) キャッシュのライン・サイズが 128B，1 次キャッシュのサイズが 16KB であるので，キューボイドは，$8 \times 8 \times 128 = 8$KB とする．短い辺の方向から見た場合には，最内側ループのベクトル長は 8 程度とかなり短く，オーバヘッドの増大が懸

念される．

視点の位置

視点の位置は，最良の場合と最悪の場合の 2 つを計測した．これらの位置は，図 2 では，それぞれ，左と中に相当する．

ピクセル順 RC 法の場合，2.2 で述べたように，最良ではアクセスが連続になり，最悪ではフェッチした 1 ライン中の 1B しか利用できない．

キューボイド順の場合，最良では最内側ループのベクトル長が最長 (128) となり，最悪では最短 (8) となる．

測 定 結 果

表 **2** に，描画速度と実効 **FLOPS** 値を示す．実効 FLOPS 値は，ピクセル値計算の計算量 $16N^3$FLOP を，描画時間で割った値である．表中，() 内は，同一の N におけるピクセル順の最良に対する比である．描画速度は，$N = 128$ の最良の場合で，約 9frame/s である．

ピクセル順 RC 法では，ボリューム全体が 3 次キャッシュ(3MB)に乗る $N = 128$ の場合を除くと，最良と最悪で約 6 倍の差が出ている．

提案手法では，ベクトル長が十分に長い最良では，ピクセル順とほぼ同じ速度を示している．それに対して，ベクトル長が短い最悪では，20%程度の速度低下が見られる．

6. お わ り に

従来用いられてきたピクセル順 RC 法では，視点の位置によっては，キャッシュ・ヒット率が著しく低下し描画速度がひどく悪化してしまう．そこで本稿では，ボリュームをキューボイドに分割し，キューボイド順に処理を進めるキューボイド順 RC 法を提案した．提案手法では，視点の位置に関わらず，ボリュームに対するキャッシュ・ヒット率を最大化することができる．

プログラムを実装し，評価した．ピクセル順では，最悪の場合，最良の場合の 6 倍もの時間がかかる．提案手法では，最悪の場合でも，ピクセル順の最良の場合の 1.2 倍程度の時間で描画することができた．

キャッシュ・ヒット率が最大化されるため，低速な主記憶が性能に与える影響は非常に小さい．したがって，

表 **1** Intanium 2 の諸元

動作周波数		1GHz		
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		6.4GB/s		
キャッシュ		L1D	L2	L3
サイズ		16KB	256KB	3MB
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** 実効 MFLOPS 値

	N	128	256	512
ピクセル順	最良	309 (1)	309 (1)	310 (1)
	最悪	147 (0.48)	52.0 (0.17)	50.0 (0.16)
キューボイド順	最良	304 (0.98)	306 (0.99)	308 (0.99)
	最悪	250 (0.81)	246 (0.79)	246 (0.77)

ボリューム・アクセスのための特殊なメモリなどを開発する必要はなく，通常の PC，WS や，汎用のグラフィクス・カードであっても，CPU/GPU の演算性能を十分に発揮できるようになる．

ただし，今回の評価に用いたプログラムでは，最内側ループの最適化をコンパイラに任せているため，理論最大の 1/10 以下の性能しか引き出せていない．今後は，最内側ループを最適化するとともに，マルチメディア命令の利用などを考えていきたい．

謝辞

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 による．

参 考 文 献

1) Lacroute, P. and Levoy, M.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, *SIGGRAPH* (1994).
2) 對馬雄次ほか: ボリュームレンダリング専用並列計算機 —*ReVolver*/C40—, *JSPP* (1995).
3) TeraRecon VolumePro WWW page: http://www.terarecon.co.jp/products/volumepro_prod.html.
4) 金喜都ほか: 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 (1997).
5) Wolfe, M.: More Iteration Space Tiling, *Proc. of Supercomputing '89*, pp. 655–664 (1989).
6) Foley, J. et al.: *Computer Graphics: Principles and Practice*, Ohmsha (2001).