修士論文

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・アルゴリズムとその評価

指導教官　　富田 眞治

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

額田 匡則

平成 16 年 2 月 6 日

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・アルゴリズムとその評価

額田 匡則

内容梗概

ボリューム・レンダリングで従来用いられてきたピクセル順レイ・キャスティング・アルゴリズムでは，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り，場合によっては利用可能な参照の局所性が全くなくなる．本来，ボリューム・レンダリングはメモリへの要求バンド幅の極めて小さい計算バウンドな処理であるが，最悪の場合，キャッシュ・ヒット率が著しく低下し，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．

本稿では，視点位置とは独立に，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・キャスティング法を提案する．提案手法では，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・ヒット率を最大化することができる．提案手法により，ピクセル順で問題となっていたメモリのデータ供給能力不足による性能低下は無視できるようになる．

ただし，これだけではプロセッサの最大性能を引き出すことはできない．提案手法には，ベクトル長の短縮とカラム・コンフリクトという 2 つの問題があるからである．これらの問題と解決法は互いに競合し，両方を同時に満足させることは難しい．それは，ボリュームを単純な 3 次元配列で定義しているからである．

実際には，ボリューム空間の座標からアドレスへの変換には任意性があり，2 つの問題は分離可能である．われわれは，ベクトル長の問題はボリューム空間で；カラム・コンフリクトの問題はアドレス空間で個別に解決するアドレス変換を提案する．これにより，キューボイド順レイ・キャスティング法の性能は最大化される．

Itanium2 サーバ上のプログラムを実装し評価した結果，描画性能は提案手法により 2.2 倍向上することが分かった．

A Volume Rendering Algorithm for Maximum Spatial Locality of Reference and its Evaluation

Masanori Nukata

Abstract

A pixel-order ray-casting algorithm for volume rendering suffers from low cache hit raito. Because the access pattern of volume references depends on the position of the viewpoint. The volume rendering quantitatively requires very small bandwidth and is computation bound. But it is actually memory bound because of low cache hit raito and memory latency.

In this paper we propose a cuboid-order ray-casting algorithm which maximizes spatial locality of reference. 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 processing all sampling points in a cuboid fetched into the cache memory, before the cache lines conposing the cuboid are replaced.

But the cuboid-order algorithm does not bring maximum performance of the processor. It has two problems: shortening vector length of the inner loop and column conflict. It is difficult to solve both of them at the same time, because solutions of these problems conflict each other under the condition that the volume data is defined as a normal three-dimensional array.

In fact, address-transformation which makes an effective address from a coordinate in the volume space has facultativity, and above-mentioned problems can be solved separately. In this paper we propose the address-transformation which enables to solve both problems, and makes performance of the cuboid-order algorithm maximum.

We evaluate our algorithm with an Itanium2 server. The result shows that performance of the cuboid-order algorithm is 2.2 times higher than that of the pixel-order.

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・アルゴリズムとその評価

目次

第 1 章　はじめに　1

第 2 章　ボリューム・レンダリング　5

2.1　ボリューム・レンダリング 5
2.2　レイ・キャスティング法 6
2.3　第 2 章のまとめ 8

第 3 章　ピクセル順レイ・キャスティング法　9

3.1　ピクセル順レイ・キャスティング法 9
3.2　最内側ループ内演算回数の削減 12
3.3　レイ・キャスティング法のメモリ・アクセス 13
3.4　第 3 章のまとめ 16

第 4 章　キューボイド順レイ・キャスティング法　17

4.1　キューボイド順 17
4.2　ピクセル値計算の中断 19
4.3　キューボイドの処理順序 20
4.3.1　処理順序の決定法 20
4.3.2　処理順序決定法の実装 23
4.4　第 4 章のまとめ 23

第 5 章　キューボイドの処理　26

5.1　ピクセルの選択 27
5.1.1　可視面と隠面 27
5.1.2　キューボイドの投影 28
5.1.3　スキャン変換の実装 28
5.2　ピクセル値の計算 29
5.2.1　ピクセル値計算の中断と再開 29
5.2.2　始点と終点 29
5.2.3　ピクセル順との比較 30

5.2.4 最内側ループの実装 30
5.3 第 5 章のまとめ 30

第 6 章 ボリューム空間座標から実効アドレスへの変換 32
6.1 ベクトル長の短縮 32
6.2 カラム・コンフリクト 34
6.3 両問題の分離 35
6.4 アドレス変換 35
6.4.1 キューボイド番号とキューボイド内オフセット 35
6.4.2 アドレス変換 36
6.4.3 アドレス変換の計算コスト 38

第 7 章 性能評価 40
7.1 キューボイド順レイ・キャスティング法のオーバーヘッド 40
7.2 実機による評価 44

第 8 章 おわりに 47

謝辞 48

参考文献 49

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，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・レンダリングにより生成された画像は，オブジェクト内部が透過的に反映されるのが特徴である．そのため，ボリューム・レンダリングは医療画像やエンターテインメントなど幅広く応用されている．

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・キャスティング法を基礎とすることが多い．レイ・キャスティング法では，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．ピクセル順レイ・キャスティング法では，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] 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，キャッシュもレイ・キャスティング法に対しては有効に働かない．レイ・キャスティング法には，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．そのうえレイ・キャスティング法では，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．第 7 章の評価では，キャッシュが有効に機能しない場合における描画性能は，有効に機能する場合の 23%にまで低下することが示される．

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 程度であるのに対し，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・パタンを制御するキューボイド順レイ・キャスティング・アルゴリズム [6] [7] を提案する．

提案手法では，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．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全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．

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．

アドレス変換

ただし，キューボイド順レイ・キャスティング法ではキャッシュ・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・コンフリクト の 2 つの問題による．

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・コンフリクトは，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 $\mathrm{vlm}[x][y][z]$ をサンプリングするとしよう．実行時には，式 $\&\mathrm{vlm}[x][y][z]$ にしたがい，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$\&\mathrm{vlm}[x][y][z]$ 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 1. ベクトル長は，ボクセル・アドレス空間の；2. カラム・コンフリクトは，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 $\&\mathrm{vlm}[x][y][z]$ 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・コンフリクトの問題が，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．その上，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，まず第 2 章でボリューム・レンダリングの原理について説明する．そして第 3 章でレイ・キャスティング法について述べた後，第 4 章と第 5 章で提案手法のアルゴリズムを説明する．そして，第 6 章ではボクセル・アドレスから 実効アドレスへのアドレス変換について述べる．第 7 章では，提案手法のオーバーヘッドを机上で求め，Itanium2 サーバ上でプログラムを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 のような汎用プロセッサ上の実装を例に，その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．

第2章　ボリューム・レンダリング

本章ではボリューム・レンダリングの概要と，ボリューム・レンダリングで主に用いられる画像生成の手法の 1 つであるレイ・キャスティング法について説明する．

2.1　ボリューム・レンダリング

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．ボリューム・レンダリングを除く一般的なコンピュータ・グラフィクス——ポリゴンのシェーディングやレイ・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．

以下でこのことを数学的に述べる．

図 1 に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 $I(\mathrm{A}, \mathrm{B})$

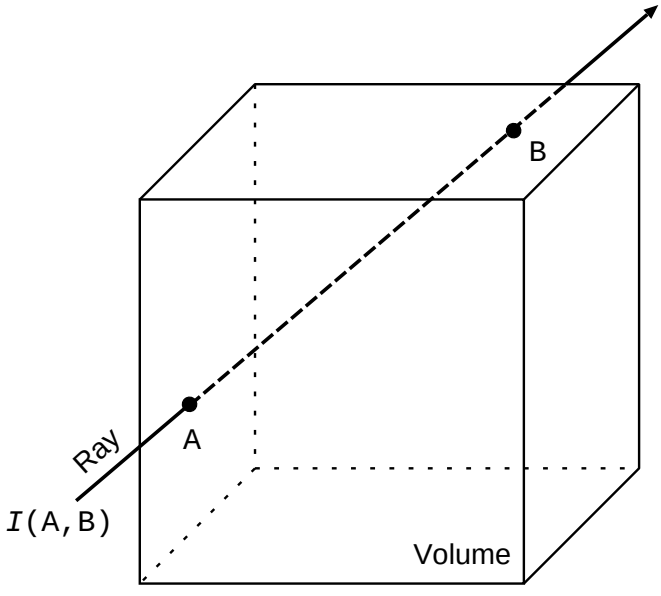


図 1: ボリュームと視線

を，

$$I(\mathrm{A},\mathrm{B}) = \int_{\mathrm{A}}^{\mathrm{B}} g(s)\, e^{-\int_{\mathrm{A}}^{s} \tau(x)\mathrm{d}x}\mathrm{d}s \tag{1}$$

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法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，

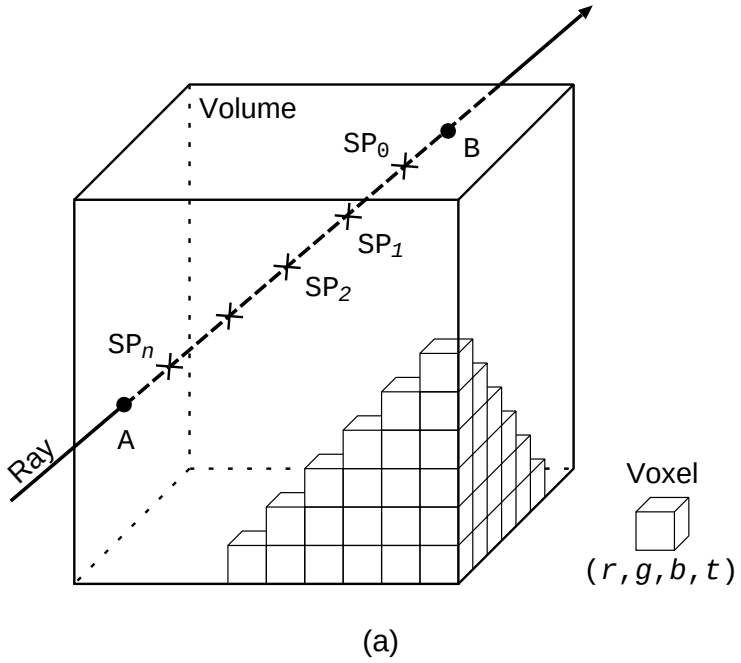
$$I(\mathrm{A},\mathrm{B}) = \sum_{i=\mathrm{A}}^{\mathrm{B}} g(s_i)\, e^{-\sum_{j=\mathrm{A}}^{s_i} \tau(x_j)} \tag{2}$$

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．これについては次節で説明する．

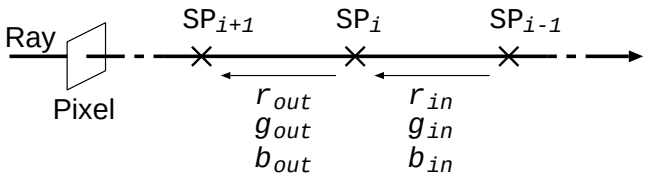
2.2 レイ・キャスティング法

レイ・キャスティング法は，視点から各ピクセルに向かって視線を飛ばし，その視線に沿って等間隔にボクセル値——色 (r, g, b) と透明度 t——のサンプリングを行い，それを用いてピクセル値を計算する．図 2(a) に，ボクセルで構成されたボリュームと，それを貫通する視線，そして，視線上のサンプリング点を示す．視点からスクリーン上のピクセルに向かって飛ばされた視線は，図 1 と同様に点 A でボリュームに入り，そのまま直進して点 B でボリュームから出る．線分 AB 上には等間隔にサンプリング点があり，図に示すように各サンプリング点を $\mathrm{SP}_0 \cdots \mathrm{SP}_n$ とする．これは，式 2 における s_i に対応する．ピクセル値は以下の式で求められる．

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \sum_{i=0}^{n} (1 - t_i) \begin{pmatrix} r_i \\ g_i \\ b_i \end{pmatrix} \prod_{j=i+1}^{n} t_j \tag{3}$$



(a)



(b)

図 2: 離散ボリュームと視線上でのサンプリング

上式からは直感的には分かり難いが，実際の計算は SP_0 から SP_n に向かって順に[1)]，

1. ボクセル値のサンプリング
2. ピクセル値計算
3. 隣の SP へ移動

の 3 つのステップを逐次的に繰り返すことになる．図 2(b) に示すように SP_{i-1} の計算結果を (r_{in}, g_{in}, b_{in}) とすると，SP_i における計算結果 $(r_{out}, g_{out}, b_{out})$ は以下の

[1)] このような計算方法を Back-to-Front という逆に SP_n から SP_0 に向かって計算する方法を Front-to-Back という．Front-to-Back は透明度の累積を計算する必要があるため Back-to-Front より計算量が大きいが，累積透明度が閾値以下になった時点でピクセル値計算を打ち切る Early Ray Termination が可能である．本稿の議論は Back-to-Front を前提としているが，提案手法の Front-to-Back への応用は容易である．

ようになる．

$$\begin{pmatrix} r_{out} \\ g_{out} \\ b_{out} \end{pmatrix} = t_i \begin{pmatrix} r_{in} \\ g_{in} \\ b_{in} \end{pmatrix} + (1 - t_i) \begin{pmatrix} r_i \\ g_i \\ b_i \end{pmatrix} \tag{4}$$

2.3 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画像生成手法であるレイ・キャスティング法について述べた．レイ・キャスティング法では，ボリュームのサンプリングは視線に沿ってなされるので，サンプリング点の座標は視点位置と視線方向に大きく依存する．サンプリングとは，メモリに格納されたボリュームに対してアクセスすることに他ならないが，そのアクセス・パタンもまた視点位置と視線方向に依存する．次章では，ボリューム・レンダリングにおいて従来一般的に用いられてきたピクセル順レイ・キャスティング法について，そのアルゴリズムを説明し，主記憶へのアクセス・パタンの性質について明らかにする．

第3章　ピクセル順レイ・キャスティング法

従来一般的に用いられてきたレイ・キャスティング法を，提案手法と区別するためにピクセル順レイ・キャスティング法と呼ぶことにする．ピクセル順とは，以下に説明するようにスクリーン上のピクセルを 1 つずつ逐次的にレンダリングすることを表している．

本章では，ピクセル順レイ・キャスティング法のメモリ・アクセスの性質について明らかにする．そのためにソフトウェアによる実装を例に，そのコードを提示しその説明を行う．

3.1　ピクセル順レイ・キャスティング法

図 3 は，ピクセル順レイ・キャスティング法の C++コードである．また，図 4 はコードで用いられている変数と，プログラムの動作を図示したものである．

本節では，まず，レイ・キャスティング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．そしてその後で，プログラムの動作について説明する．

コード内の変数

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 1(B) のインデクスでカラーマップ `map` を引いて，実際のボクセル値を得る．図 4 では，左側の大きな立方体がボリューム `vlm` を，ボリューム内の小さな立方体がボクセルを，そして，右側のテーブルがカラーマップ `map` をそれぞれ表している．

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．図 4 では，2 次元配列 `pxl` をボリュームの手前にある正方形で表している．また最内側ループでは，ピクセルの各色を，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．また，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．同図では視線に平行な太い矢印で表している．

サンプリング点　視線上，ボリュームの一番奥の点を $\boldsymbol{P_0}$ とすると，サンプリン

```
struct Pixel { float r, g, b; };        // ピクセル値
struct Voxel { float r, g, b, 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
      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. ピクセル値の計算 (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. サンプリング点の移動 (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 3: ピクセル順レイ・キャスティング法のコード

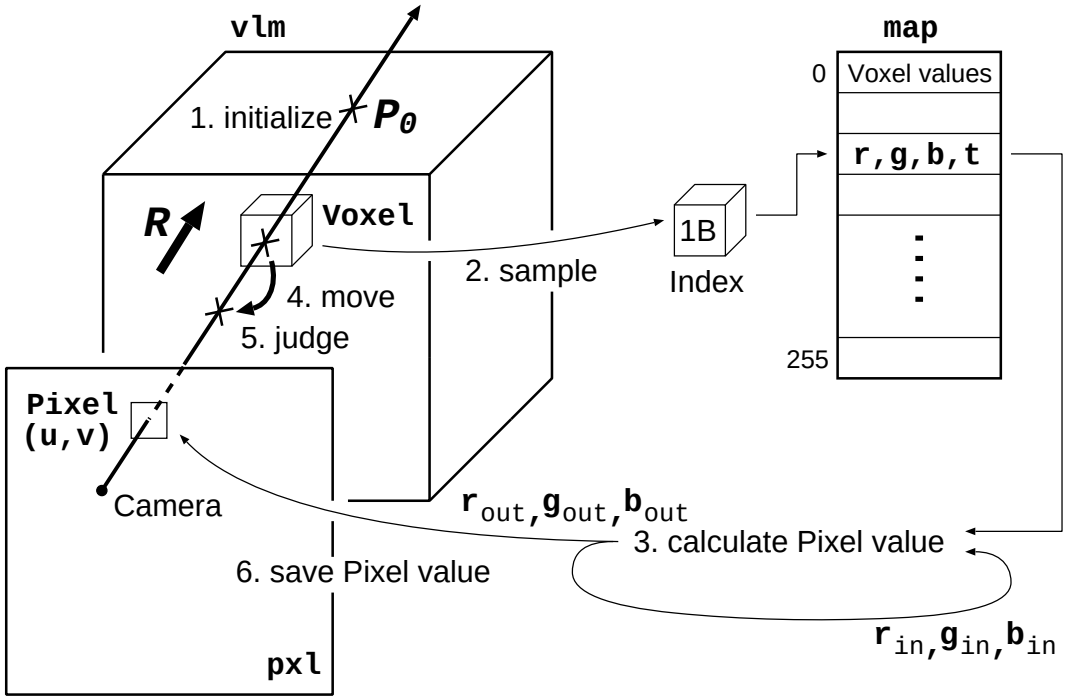


図 4: ピクセル順レイ・キャスティング法の変数と最内側ループの処理

グ点 (x, y, z) は，$\boldsymbol{P_0} - n \cdot \boldsymbol{R}$ $(n = 0, 1, 2, \cdots)$ で与えられる．同図では × 印でサンプリング点を表している．

プログラムの動作

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．

図 4 中の数字はコード中のそれに対応しており，制御の流れを示す．そして同図中の矢印はデータの流れを表している．

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．サンプリング点は $\boldsymbol{P_0}$，(r, g, b) は 0 に初期化する．
2. サンプリング　サンプリング点の座標が float 型の変数 (x, y, z) で与えられる．これらを整数に変換した ((int) x，(int) y，(int) z)) は，サンプリング点を含むボクセルの座標である．これをもとにアドレス &vlm[(int)x][(int)y][(int)z] を計算し，メモリから読み出す．得られた 1(B) のインデクスでテーブル map を引き実際のボクセル値 (r，g，b，t) を得る．

3. ピクセル値の計算　2. で得られたボクセル値を用いてピクセル値を計算する．コードの計算式右辺にある (r, g, b) は，初期値，あるいは，前イタレーションの計算結果であり，図 2 における (r_{in}，g_{in}，b_{in}) である．
4. サンプリング点の移動　あるサンプリング点についてピクセル値計算が終了すると，次のサンプリング点に処理を移す．ボリューム空間内では，同じ視線上の隣接するサンプリング点に移動することになる．このときの変位は $-\boldsymbol{R}$ であり，サンプリング点の座標から視線ベクトルを引くと次のサンプリング点が求まる．
5. 終了判定　4. で得た新しいサンプリング点の座標をボリュームの境界と比較し，ボリュームから外れているかどうか調べる．サンプリング点がボリューム内にあればピクセル値計算を続け，そうでなければ終了する．
6. ピクセル値の書き込み　ピクセル値計算終了後，変数 (r, g, b) には最後のサンプリング点における計算結果 (r_{out}，g_{out}，b_{out}) が格納されている．これが求めるピクセル値であり，スクリーン上のピクセル `pxl[u][v]` に書き込む．

3.2　最内側ループ内演算回数の削減

前節で示したコードは教科書的な例である．無論そのまま実装に用いても全く正しく動作するが，少しの工夫で最内側ループの浮動小数点数演算数を削減することができる．

図 5 に，改良したコードを示す．カラーマップ `map` は本来，色と透明度を保持するためのものであった．ここに透明度と，そして，RGB 各色と不透明度の積をあらかじめ計算して格納しておく．同図のコードでは，上側にある 1 重ループでその処理を行っている．こうしておくと，図 3 のループ内にある，不透明度の計算と，色と不透明度の乗算を行う必要はなくなり `map` からロードした値をそのまま利用することができる．

図 3 のコードから，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，16(FLOP) (Floating Operations) になることが分かる．一方，図 5 ではピクセル値の更新が 6(FLOP) で済み，合計で 12(FLOP) である．全体の計算量は，サンプリングが N^3 回行われるとすると，$16 \times N^3$(FLOP) が $12 \times N^3 + 4 \times$ (UCHAR_MAX + 1)(FLOP) になる．

以下，本稿ではここで述べた改良されたコードを用いることにする．

```
for(int i = 0; i <= UCHAR_MAX; ++i) {
   Voxel vxl = map[i];
   float a = 1.0F - vxl.t;            // 不透明度
   vxl.r *= a;
   vxl.g *= a;
   vxl.b *= a;
   map[i] = vxl;
}
for (int u = 0; v < N; ++v)
   for (int u = 0; u < N; ++u) {
      ........
      while (IS_IN_VOLUME(x, y, z)) {
      Voxel vxl = map[vlm[(int)x][(int)y][(int)z]];
      // ピクセル値の計算 (6FLOP)
      float t = vxl.t;                // 透明度
     r = t * r + vxl.r;
      g = t * g + vxl.g;
      b = t * b + vxl.b;
      x -= dx;  y -= dy;  z -= dz;
   }
   pxl[u][v].r = r;  pxl[u][v].g = g;  pxl[u][v].b = b;
}
```

図 5: ループ内演算回数の削減

3.3 レイ・キャスティング法のメモリ・アクセス

レイ・キャスティング法は，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　前述のように，最内側ループ 1 イタレーション，すなわちサンプリング 1 回あたりの演算回数は，12(FLOP) である．1 ボクセル

を 1(B) とすると，メモリに対する要求バンド幅，すなわち，1(FLOP) あたりのデータ転送量は $1/12 \approx 0.083$(B/FLOP) と極めて小さくなる．

一方，一般的なアプリケーション——例えば内積計算などでは，要求バンド幅は 2(word)/2(FLOP) $\approx 4 \sim 8$(B/FLOP) である．したがって，単純に量だけを比較すれば，レイ・キャスティング法は，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 (computation-bound) 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．

このような状況では，キャッシュ・ヒット率ではなく，ライン利用率を考えると都合がよい．ライン利用率とは，あるラインがフェッチされてからリプレースされるまでに利用されたデータの量を，ライン・サイズで割ったものと定義する．

ピクセル順レイ・キャスティング法の空間局所性は，以下に示すように，視点の位置に強く依存する．

図 6 に，2 次元のピクセル順レイ・キャスティング法における視点の位置とサンプリング点の処理順序を示す．図中，丸数字が視線の，サンプリング点の近傍にある数字がサンプリング点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．

同図上では，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アドレス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．つまり，ライン利用率は 1 である．

一方同図左下では，視点が縦方向にあるため，ヒット率が低下する．視線① のサンプリング点 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 に対してサンプリング点 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 1(B) しか利用できないことになる．このとき，ライン・サイズを L(B) とすると，ライン利用率は $1/L$ となる．

ここでライン・サイズ L が十分小さければ問題はないが，実際には L の値は大きく，ライン利用率は著しく低下する．なぜなら，前述したように，最近の DRAM はキャッシュとの間のライン転送に特化しており，そのためライン・サイ

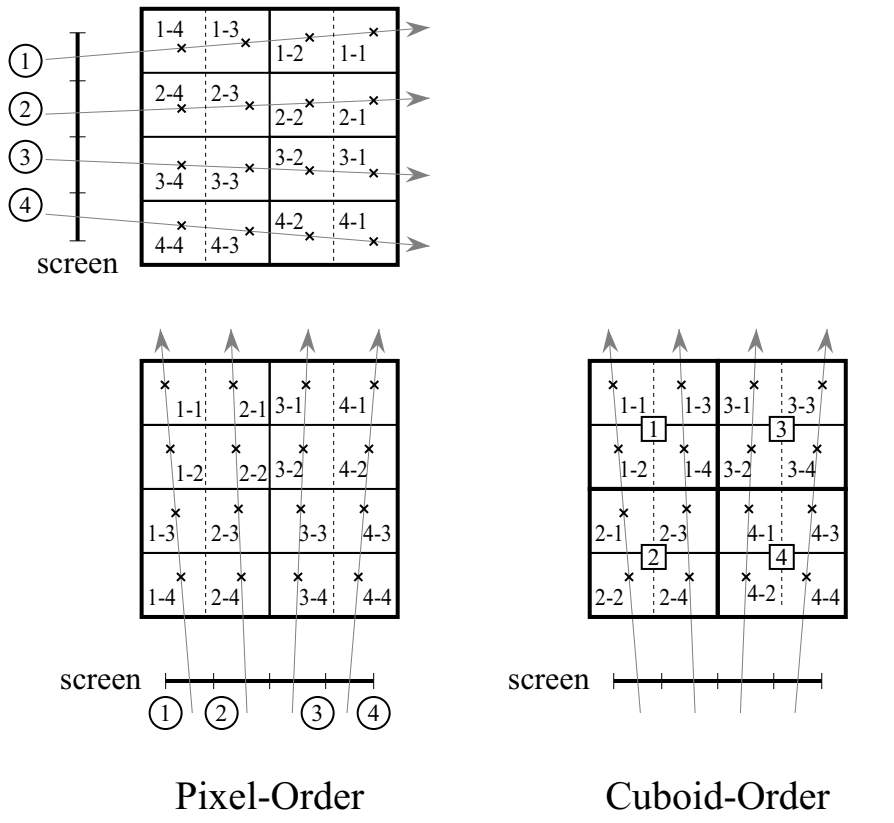


図 6: 視点の位置とサンプリング点の処理順序

ズを大きくとっているからである．たとえば，第 7 章の評価に用いた Itanium2 プロセッサのライン・サイズは，128(B) と大きく，ライン利用率は 1/128 となる．

$L = 128$ の場合，ライン利用率の比，すなわち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 $1/12 \times L = 128/12 \approx 10.7$(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．特にこの場合，メモリ・アクセスの度にキャッシュ・ミスが発生するので，処理時間の大部分をメモリ・レイテンシが占めることになる．

3.4 第3章のまとめ

本章ではピクセル順レイ・キャスティング法のアルゴリズムを説明し，ボリュームへのアクセス・パタンとキャッシュ・メモリとの親和性について考察した．

ピクセル順レイ・キャスティング法では視点位置によりアクセス・パタンが動的に決まり，従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ライン利用率が最大化され計算バウンドな処理となるが，一方，最悪の場合ではライン利用率は $1/L$（L はライン・サイズ）にまで低下し，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．

第 7 章の評価では，最悪の場合における描画性能は，最良の場合のそれの 23%になることが示される．評価に用いた Itanium2 サーバでは，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ライン利用率を最大化するキューボイド順レイ・キャスティング法を提案する．キューボイド順レイ・キャスティング法では，視点位置に関わらずキャッシュ・ミスは発生しない，したがって，ピクセル順レイ・キャスティング法のようなメモリ・レイテンシによる性能低下はほとんどなくなる．次章以下では，キューボイド順レイ・キャスティング法について説明する．

第4章　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，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．

4.1　キューボイド順

レイ・キャスティング法のアクセス・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，そのラインがリプレースされるまでにライン内のすべてのサンプリング点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，それだけでライン利用率の上限が達成されることになる．レイ・キャスティング法は本来，メモリへの要求が極めて小さい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，オーバーヘッドの低減のため，1 つのキャッシュ・ライン単体ではなく，複数のラインからなるキューボイド (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 128(B) とすると，ボリューム空間内では $1 \times 1 \times 128$ 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で構成された直方体とする．

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，第 7 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・サイズで割った 8K(B)/128(B) = 64 である．つまり，キューボイドは 64 本のラインから成る直方体であり，その形状には任意性がある．この任意性に関する議論は第 6 章で行うので，ここではとりあえず，キューボイドは $8 \times 8 \times 128$ のような形状をしているものとしておく．

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・パタンは，次の条件を満たすサンプリング点の処理順序として与えられる：

条件 **1**　1 つの視線上のサンプリング点は，順に処理する．

ピクセル順レイ・キャスティング法では，各ピクセルに対する視線上のサンプリング点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．図 3 の最内側ループがそれを示している．

提案手法では，更に，以下の条件を加える：

条件 **2**　キューボイド内のサンプリング点は，連続して処理する．

提案手法は，ピクセル順に対して，キューボイド順レイ・キャスティング法と呼ぶことができる．キューボイド順レイ・キャスティング法では，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サンプリング点を処理することになる．例えば，図 6 右の例では，□1 ~ □4 の順序でキューボイドを選択し，その中のサンプリング点を連続して処理している．

さて，キューボイド順レイ・キャスティング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　キューボイドには複数の視線が通っている．したがってキューボイド内のサンプリング点のみを処理するには，あるピクセル値の計算を計算を途中で一旦中断し，そして他のピクセル値計算に処理を移す必要がある．図 6 右の例では，ピクセル○1 の計算をサンプリング点 1-1，1-2 の処理を行ったところで中断し，ピクセル○2 に対してサンプリング点 1-3 の処理を開始している．ピクセル○1 の計算は，処理がキューボイド□2 に移った後にサンプリング点 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　任意の視線で条件 1 を満たすには，キューボイドについてもまたスクリーン奥から順に処理するようにしなければならない．そのようなキューボイドの処理順序を効率よく発見できるか．
3. キューボイド内のサンプリング点の処理　条件 1 を満たした上で，キューボ

イド内のすべてのサンプリング点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ーヘッドはどの程度か．オーバー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，4.2 節と 4.3 節において，上記の 1.，および，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，第 5 章で詳しく説明する．また，4. については第 6 章で議論する．

4.2　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 5 のコードの最内側ループの終了条件を書き換えて，サンプリング点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．ピクセル値計算が条件 1 を満たしているならば，途中結果はかならず 1 つである．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，サンプリング点を表す (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，合わせて $24N^2$(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．

```
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 7: 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

4.3 キューボイドの処理順序

4.3.1 処理順序の決定法

すべてのサンプリング点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．

処理順序

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 7 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 8 に示す 2 次元のボリュームを例に，図 7 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 8 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決まる．loop_x では，ボリュームを x < cx，x > cx，x = 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 x < 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，$\cdots$，cx – 1 の順に決めると，視点遠方の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

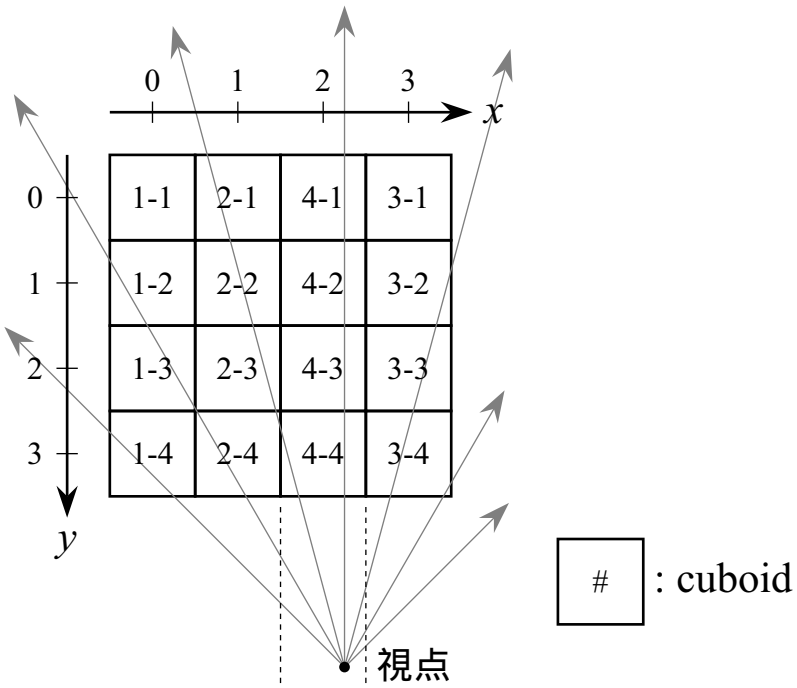


図 8: 図 7 のコードによるキューボイドの処理順序

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．

loop_x は，ボリュームをスライスに切り，スライス間の処理順序を決める関数と見なすことができる．同図では，x = 0，x = 1，x = 2，x = 3 の 4 つのスライスがある．内側の関数 loop_y は，各スライスに対して y 軸方向の処理順序を決める．図の例では，コードに従い 0，1，2，3 と決められる．

結局図 8 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．図 7 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 8 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 8 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

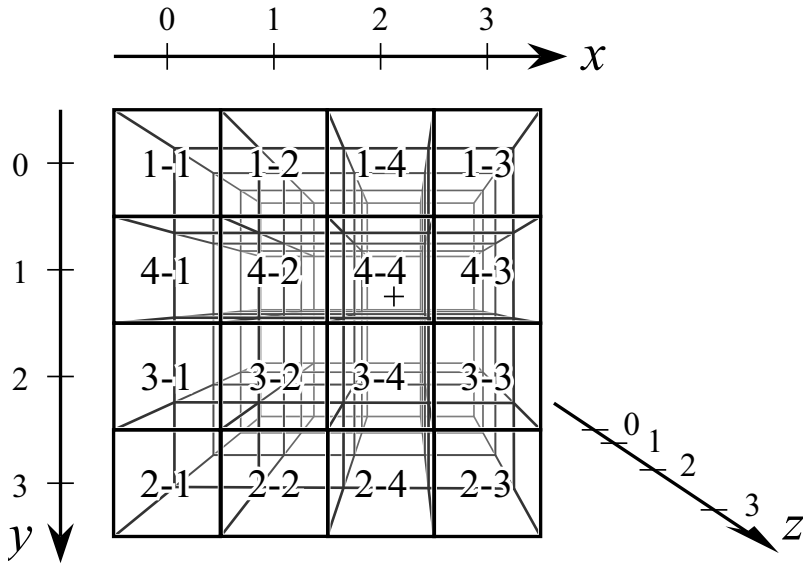


図 9: 視点の位置とキューボイドの処理順序

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．

軸間の順序は，

1. 視線の選択し，
2. 視線ベクトルの成分の絶対値を比較しソートする

ことで決定する．

順序の決定には視線ベクトルを用いる．透視投影の場合ピクセル毎に視線ベクトルは異なるので，代表となる視線を 1 つ選びそれを用いる．代表にはスクリーンの中心を通る視線を選択するとよい．視野角がよほど広角でない限り，これで十分である．

軸間の順序は，視線ベクトルの 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 9 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．

4.3.2 処理順序決定法の実装

前節で述べたように x，y，z 軸間の順序には任意性がある．従って，図 7 のようにコード上で軸間の順序を静的に指定するならば $3! = 6$ 通りの場合分けが必要となる．コードが煩雑になるのを防ぐためにも場合分けに頼らない汎用性のあるコードを書く方が良いだろう．以下に実装の一例を示す．

図 10 のコードは x，y，z 軸を，軸間の順序が静的に定まった仮想的な軸，p，q，r 軸に置き換えることで場合分けの発生を防いでいる．ここでは，外側から順に p，q，r としている．

コード中の配列 pqr2xyz および xyz2pqr は，それぞれ *pqr-xyz*，*xyz-pqr* 変換に用いるものである．例えば xyz2pqr[X] の値は，p，q，r 軸のうち x 軸に対応するものを指している．*pqr-xyz* 変換および逆変換は以下の手順で行う．

1. *xyz-pqr* 変換　関数 loop_p を呼ぶ前に，*xyz-pqr* 変換およびその逆変換を求める．例では関数 transform でそれを行っている．軸間の順序はスクリーン中心を通る視線の方向から求めるので，引数として視線ベクトル v を渡している．
2. ソート　v[pqr2xyz[P]]，v[pqr2xyz[Q]]，v[pqr2xyz[R]] の 3 つは，それぞれ p，q，r 軸に対応する視線ベクトルの成分である．これらが p，q，r 軸の順で降順になるように pqr2xyz の値をソートする：各軸の x，y，z 軸との対応を変更する．この結果，*pqr-xyz* 変換が得られる．
3. 逆変換の初期化　2. で得られた *pqr-xyz* 変換を用いて *xyz-pqr* 変換を求める．これは簡単である．
4. 逆変換　キューボイドの処理を行う関数 render_cuboid のはじめに，*pqr-xyz* 変換を行い処理すべきキューボイドの番号を得る．

以上のようにすると，コードはやや複雑になるものの，場合分けをなくすことができる．

4.4 第 4 章のまとめ

本章ではキューボイド順レイ・キャスティング法の概要を述べた．キューボイド順レイ・キャスティング法は，ボリュームを複数のキャッシュ・ラインからなる直方体——キューボイドに分割しキューボイド順に処理することでメモリへのアクセス・パタンをコントロールする手法である．キューボイドの処理とは，キューボイド全体がキャッシュ・メモリへフェッチされ，キューボイドを構

成するラインがリプレースされるまでにキューボイド内の全てのサンプリング点を一気に処理することである．こうすることで，ライン利用率の最大化は達成される．

キューボイド順レイ・キャスティング法ではキューボイド単位のレンダリングを行うため，ピクセルの値計算を途中で中断，および，再開する必要があるが，これは可能である．ただし，各視線についてどのサンプリング点まで処理したかを記憶する必要があるため，N^2 の領域が必要となる．

また，キューボイドの処理順序は，任意の視線についてサンプリング点をスクリーン奥から順に処理するように決めなければならない．そのような処理順序は任意の視点位置について必ず存在し，単純なコードで決定される．

さて，前述のようにレイ・キャスティング法ではサンプリング点の座標が視点位置により動的に決まる．よってキューボイド順レイ・キャスティング法では，いかに効率よくキューボイド内のサンプリング点を抽出するかという点が重要である．次章では提案手法の核となるキューボイドの処理，キューボイド内のサンプリング点の抽出方法について説明する．

```
enum XYZ { X = 0, Y = 1, Z = 2 };
enum PQR { P = 0, Q = 1, R = 2 };
unsigned char pqr2xyz[3] = { X, Y, Z };   // pqr-xyz変換
unsigned char xyz2pqr[3] = { P, Q, R };   // xyz-pqr変換

void transform(float* v) {
   sort(v);                               // 2. ソート
   init_xyz2pqr();                        // 3. 逆変換の初期化
}

int p, q, r;

void render_cuboid(void) {
   int x, y, z;
   int pqr[3] =  p, q, r ;                // 4. 逆変換
   x = pqr[xyz2pqr[X]];
   y = pqr[xyz2pqr[Y]];
   z = pqr[xyz2pqr[Z]];
      ........
}

void loop_r(void) {
   for (r = 0; r < cr; ++r)
      render_cuboid();
      ........
}

void order(void) {
   float v[3];                            // 視点-スクリーン中心ベクトル
   init(v);
   transform(v);                          // 1. xyz-pqr 変換
   loop_p();
}
```

図 10: 場合分けしないで処理順序を決めるコード

第5章　キューボイドの処理

図 11 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．四辺形ポリゴンで構成された直方体をポリゴン・レンダリングする際に区別される可視面，隠面と全く同じである．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．処理すべきサンプリング点は，これらの 2 つの交点の間にある．

ボリュームに対するライン利用率は，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．キューボイドはその大きさをキャッシュ・サイズより小さくとってあるので，キューボイド内部のみを処理するならばラインのリプレースは発生せずキャッシュに保持され続ける．したがって，キューボイド内のサンプリング点の処理順序は，自由に定めてよい．どのような順序であろうと，初回のミスを除き必ずキャッシュ・ヒットする．後述する理由により，キューボイド内のサンプリング点はピクセル順に処理する．

キューボイド内のサンプリング点をピクセル順に処理するため，図 5 に示したピクセル順レイ・キャスティング法のコードを一部流用することができる．図 5 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

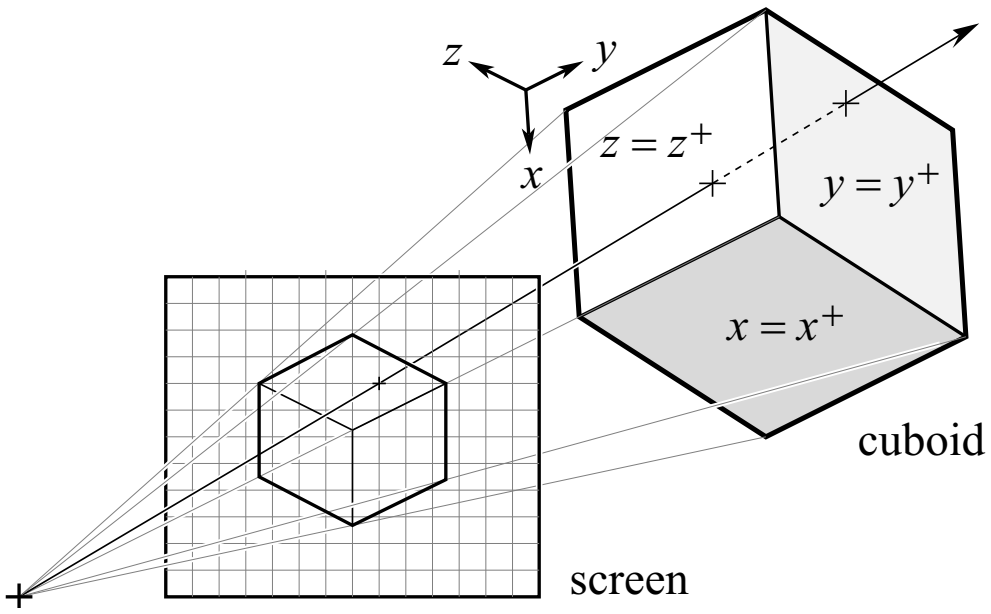


図 11: キューボイドの投影

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，5.1 節と 5.2 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．

5.1 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，ピクセル順レイ・キャスティング法の場合と異なり，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 11)．

5.1.1 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，ほとんどオーバーヘッドなしに区別することができる．

キューボイドは，6 つの平面から構成される．各平面は yz 平面，zx 平面，xy 平面のいずれかと平行である．たとえば，方程式 yz 平面に平行なものは 2 つあり，それぞれ方程式 $x = x^-, x = x^+$ によって与えられる．ただし，$x^- < x^+$ とする．他の平面についても同様に，$y = y^-, y = y^+, z = z^-, z = z^+$ ($x^- < x^+, y^- < y^+, z^- < z^+$) とする．

可視面と隠面は，図 7 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 7 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．yz 平面に平行な 2 つの面 $x = x^+$ と $x = x^-$ が可視面，および，隠面のどちらになるかは，キューボイドがどの領域に属するかよって決まる．

図 12 は，上側に視点位置とキューボイドを俯瞰で，下側にその場合のキューボイドの視点からの見え方を示したものである．可視面を実線，隠面を破線で描いている．左から順に，キューボイドが領域 $x < cx$，$x = cx$，$x > cx$ にある場合を表している．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

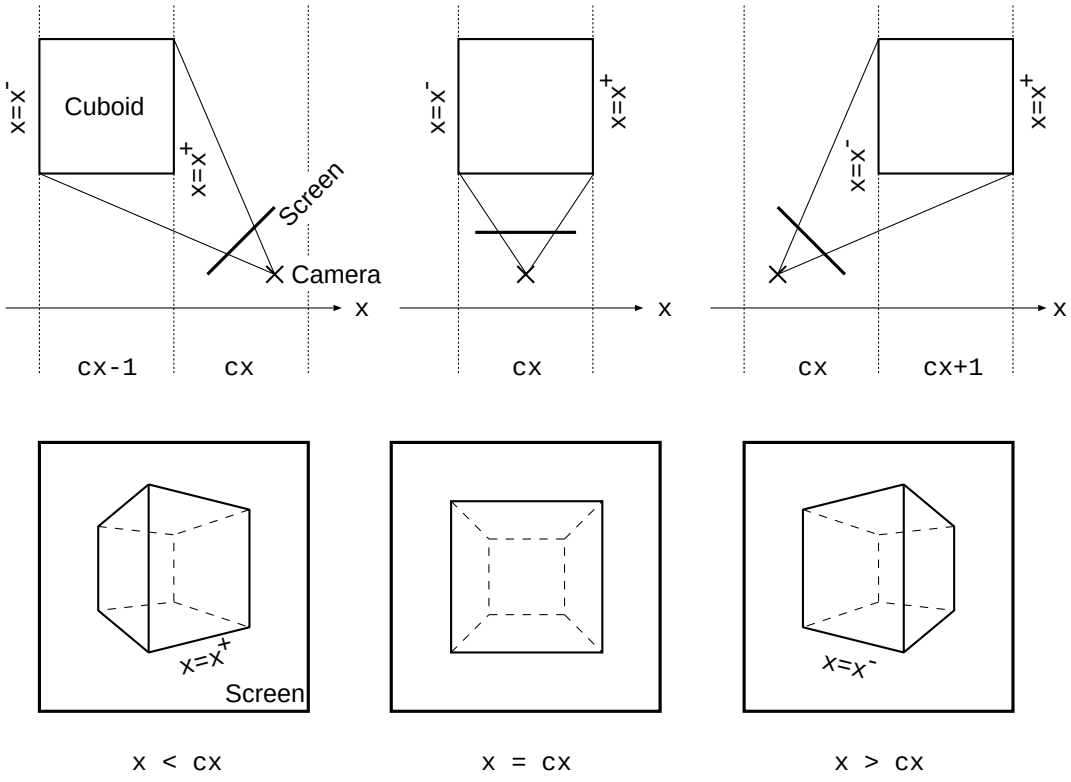


図 12: 可視面と隠面

に求められる．

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．

5.1.2 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 [8]．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 (ラスタライズ) 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．

5.1.3 スキャン変換の実装

上記のとおり，スキャン変換それ自体には特別な方法は必要ない．しかし，次節で説明するが最内側ループの終了判定は可視面ごとに異なるため，スキャン

している面がどの可視面であるかという情報をもっていなければならない．

この情報は，頂点と面の関係を表すテーブルを予め作成しておき，それを引くことで得られる．キューボイドの頂点と面の関係——頂点の属する面，隣接する頂点——は既知である．これにどの面が可視面かという情報を加えると，必要な情報が得られる．

5.2 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，および，隠面との交点間にあるサンプリング点を，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. 4.2 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 5 のコードをほぼそのまま流用することができる．

5.2.1 ピクセル値計算の中断と再開

ピクセル値計算を中断し，そして再開するためには，4.2 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 5 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．

5.2.2 始点と終点

ループの始点，および，終点は，視線と隠面，および，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，および，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，次のサンプリング点が 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，最初のサンプリング点の座標を得ることができる．

終点

最内側ループは，次のサンプリング点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，サンプリング点 (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 11 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，$z > z^+$ ならばサンプリング点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 5 のコードで `IS_IN_VOLUME()` と示しているように，サンプリング点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．

5.2.3　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 5 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．

5.2.4　最内側ループの実装

最内側ループの終了判定は，面ごとに異なる．たとえば，上で可視面 $z = z^+$ を通過する視線のピクセル値計算の終了判定について述べたが，このときの終了判定 $z > z^+$ を他の面の終了判定に流用することはできない．キューボイドは 6 面体であるので，終了判定の場合分けは全部で 6 通りとなる．

性能を重要視するなら，終了判定を選択する分岐をループ外に出すとよい．第 7 章の評価に用いたプログラムは，終了判定のみが異なる最内側ループを 6 通り用意している．どのループにジャンプするかを決める分岐命令は，スキャン変換の最外側ループに置き，内側のループにジャンプ先のアドレスを渡すようにする．こうすると性能とコードの単純さを両立できる．

5.3　第 5 章のまとめ

キューボイドの処理は，以下のようなループ構造によって行われる：

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 11 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　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．キューボイドを構成する 6 面を，スクリーン側にある可視面と，可視面により隠される隠面に分類し，可視面をスキャン変換することでピクセルを順に選択していく．

3. ピクセル値の計算　選択されたピクセルについて，以下の始点から終点まで，図 5 に示すのと同じコードによりピクセル値の計算を行う．なお，始点/終点は以下のようにして得る．

   始点　最初の SP は配列 sp[N][N] に保存されており，これを用いる．

   終点　サンプリング点の座標がキューボイドから外れているかどうかを調べることで得られる．

   ループ実行が終了すると，ピクセル値の途中結果とともに，終点の次のサンプリング点の座標を sp[N][N] に保存しておく．

以上で提案手法のアルゴリズムはすべて説明した．結局，キューボイド順レイ・キャスティング法における性能低下の要因は，

1. キューボイドの投影処理
2. スキャン変換処理
3. 最内側ループのベクトル長の短縮

の 3 点である．

詳細は第 7 章で述べるが，1. 投影処理と 2. スキャン変換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．

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，キューボイド順レイ・キャスティング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．ベクトル長は最小で約 6 と非常に小さくなり，このときの性能低下は深刻である．

次章ではこの性能低下を最小にする方法について述べる．

第6章　ボリューム空間座標から実効アドレスへの変換

前章で述べたように，キューボイド順レイ・キャスティング法には 1. ベクトル長の短縮 という問題がある．また，以上の議論ではあまり触れなかったが，2. カラム・コンフリクト という問題がある．

本章では，まずこれら 2 つの問題点について説明し，その後これらを解決する方法について述べる．

6.1　ベクトル長の短縮

図 13 にベクトル長と性能の関係を示す．第 7 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．これには，投影処理やスキャン変換などのオーバーヘッドは含めない．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．

視点位置とベクトル長

さて，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毎に異なる．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投影面積で割ると求められる．視点を動かすと平均ベクトル長も変化し，これが最大となるときが最良の場合，逆に最小となるときが最悪の場合

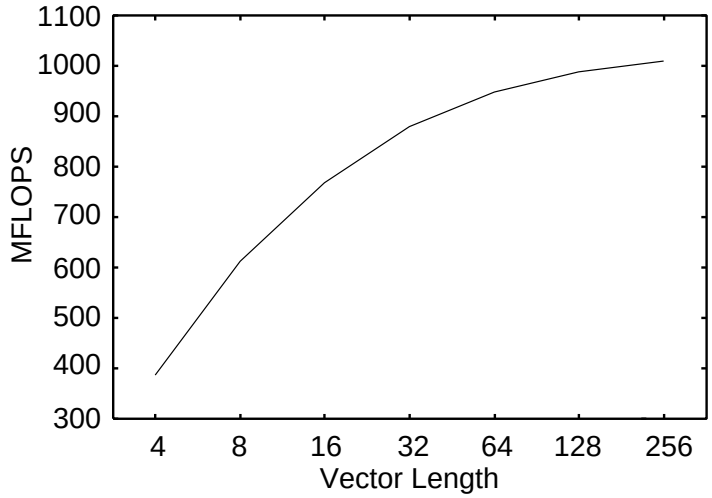


図 13: ベクトル長と性能

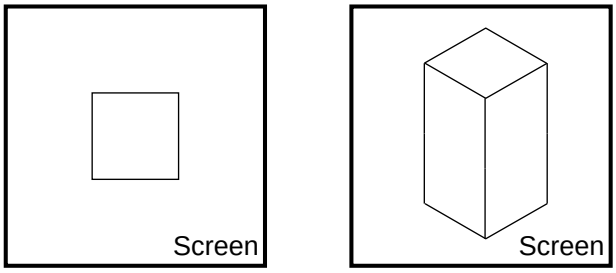


図 14: 最良の場合と最悪の場合

である．

図 14 は，スクリーンに投影されたキューボイドの像である．キューボイドのサイズを $8\times8\times128$ とすると，同図左のように像が 8×8 の正方形のときが，投影面積が最小となる最良の場合である．逆に同図右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る場合が，最悪の場合である．最悪の場合における平均ベクトル長は最小の 5.7 となる．

理想的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．このような用途では，最悪の場合の描画速度を求め性能の下限を示すことが重要である．言い換えると，フレーム・レートは最悪の場合の性能で決定されるので，最悪の場合の性能向上を重要視すべきである．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．

以下，最悪の場合について考える．最悪の場合の平均ベクトル長が最大となるのは，キューボイドが立方体のときである．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からである．キューボイドの形状を $8\times8\times128$ から，同体積の $16\times16\times32$ に変更すると，平均ベクトル長は 5.7 から 13.9 に改善される．

キューボイド順レイ・キャスティング法の原理から，最悪の場合におけるベクトル長をこれ以上大きくすることはできない．従って，キューボイドの形状が立方体のとき，キューボイド順レイ・キャスティング法の性能は最大化される．

6.2 カラム・コンフリクト

カラム・コンフリクトは，以上の議論ではあまり触れなかったが，キューボイド順レイ・キャスティング法を成り立たせるにあたり非常に重要な問題である．以下に，それを説明する．

前述のように，キューボイド順レイ・キャスティング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していなければならない．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，コンフリクトが発生しラインはリプレースされてしまう．特に，キャッシュ・メモリのウェイ数が少ない環境では少しの偏りでカラムがコンフリクトしてしまう．従ってカラム・コンフリクトを防ぐためには，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 3 に示すように，ボリュームを単純な 3 次元配列 `vlm`$[N][N][N]$ で定義すると，ボリューム空間内で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当てられる．

一般に，アクセス・ストライドが等間隔になる問題に対しては，整合配列を用いることで対処する．この問題も例外ではなく，整合配列によって解決できる．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当てられることになる．

この方法では確かにカラム・コンフリクトは起こらないが，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，ボリュームの N^3 に加えて N^2 の領域が必要となるが，これは全く利用されない無駄な領域である．特にライン・サイズが 128 と大きい場合，この問題は深刻である．$N = 1024$ の場合，$N^3 = 1024^3 = 1$(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．

整合配列を用いない方法

整合配列を用いなくとも，簡単な方法でカラム・コンフリクトを防ぐことが可能である．各キューボイドをアドレスが連続する $8 \times 8 \times 128 = 8$(KB) の領域に配置すればよい．キャッシュ・メモリの原理から，アドレスの連続するラインは異なるカラムに割り当てられる．したがって，カラム数を 32 とすると，64 ラ

インからなるキューボイドは各カラムに 2 ラインずつ均等に振り分けられることになる．この方法では，整合配列のような余分な領域は必要ない．

6.3 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 &vlm[x][y][z] にしたがい，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 &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．

従来の方式におけるアドレス変換の式では，図 16 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．そのため，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が，互いに競合することになる．

前述のように，ボクセル・アドレス空間におけるラインの形状は $1 \times 1 \times 128$ である．したがって，キューボイドの形状を立方体にしようとしても，形状の z 軸方向の自由度が小さいため，z 軸方向に細長い $8 \times 8 \times 128$ にせざるを得ない．また，カラム・コンフリクトを避けるために整合配列が必要となるのは前節で述べたとおりである．

実際には，ボクセル・アドレスと実効アドレスの間のアドレス変換には任意性がある．次節で提案するアドレス変換によって，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．

6.4 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．

6.4.1 キューボイド番号とキューボイド内オフセット

例えば仮想アドレス・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

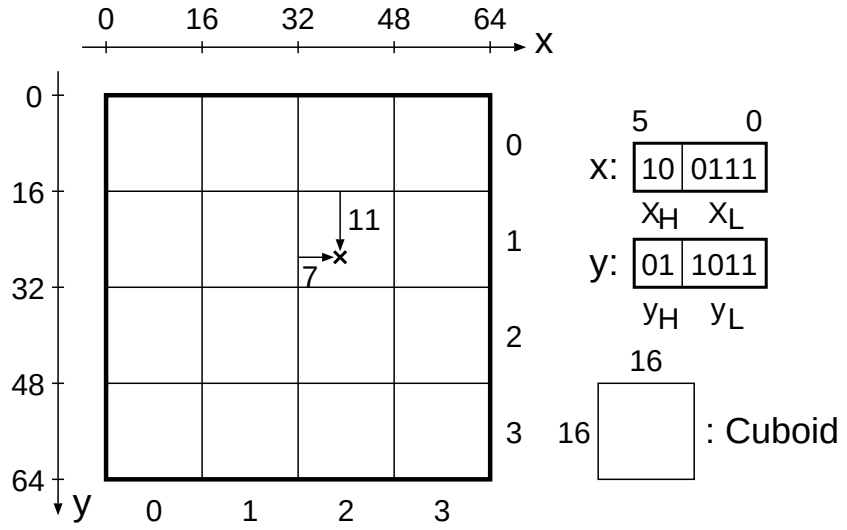


図 15: キューボイド番号とキューボイド内オフセット

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 次元であるものを更に『2 次元化』することになる．

図 15 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．例えば，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．

6.4.2 アドレス変換

提案するボクセル・アドレスから実効アドレスのアドレス変換は，上述したキューボイド番号とキューボイド内オフセットに基づいて行う．図 16 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

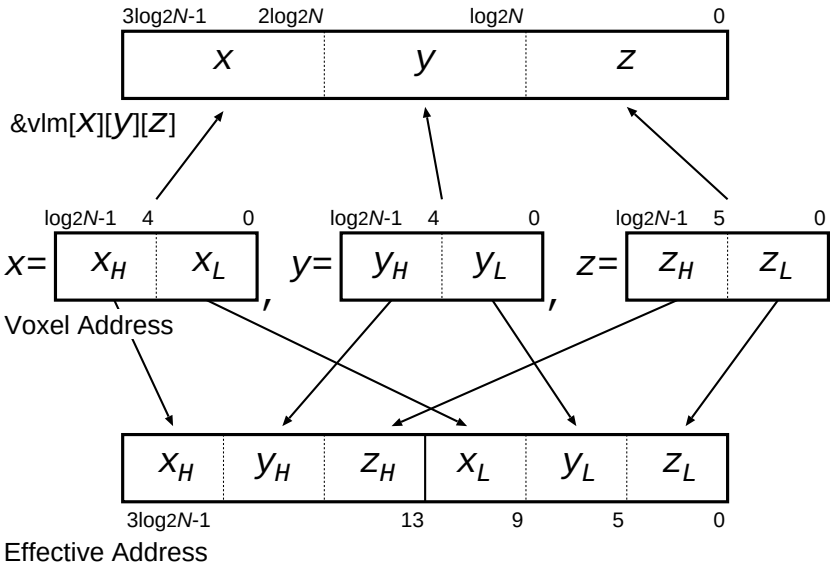


図 16: ボクセル・アドレスから実効アドレスへのアドレス変換

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 15 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．

さて，提案するアドレス変換で得られた実効アドレス $x_H \,|\, y_H \,|\, z_H \,|\, x_L \,|\, y_L \,|\, z_L$ では，キューボイド番号 (x_H, y_H, z_H) が上位に，キューボイド内オフセット (x_L, y_L, z_L) が下位に集められていることが重要である．キューボイド番号が (x_H, y_H, z_H) であるキューボイド内部にある，すべてのボクセルは，キューボイド内オフセットが $(x_L, y_L, z_L) = (0, 0, 0)$ から $(C_x{-}1, C_y{-}1, C_z{-}1)$ で表される．これらのボクセルは，実効アドレス空間上では，連続する領域 $x_H \,|\, y_H \,|\, z_H \,|\, 0 \,|\, 0 \,|\, 0$ から $x_H \,|\, y_H \,|\, z_H \,|\, C_x{-}1 \,|\, C_y{-}1 \,|\, C_z{-}1$ に存在する．前述したとおり，キューボイドのサイズはキャッシュ容量未満にするので，1 つのキューボイド内のすべてのボクセルは，実効アドレス空間上では，連続するキャッシュ容量未満の領域を占めることになる．

前述したように，キューボイド順レイ・キャスティング法では，1 つのキューボイド内のサンプリング点を一気に処理する．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．キューボイド順レイ・キャスティング法のアドレス

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　提案するアドレス変換では，(C_x, C_y, C_z) で定まるキューボイドの形状に依存しておらず，(C_x, C_y, C_z) は，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・コンフリクト　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．

6.4.3　アドレス変換の計算コスト

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．一般的なループ・プログラムでは，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からである．しかしボリューム・レンダリングでは，このような方法ではボクセルのアドレスを計算できないことに注意されたい．ボリューム・レンダリングでは，図 3 に示したように，まずサンプリング点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，&vlm[x][y][z] の計算は，単にストライドを加算すればよいのではなく，x，y，z の 3 つの値を連結する操作が必要となる．

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，そのコストは十分に小さく抑えることができる．例えば，次章の評価で用いる IA-64 命令セットには，図 17 に示す dep（Deposit：格納）命令 [9] 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

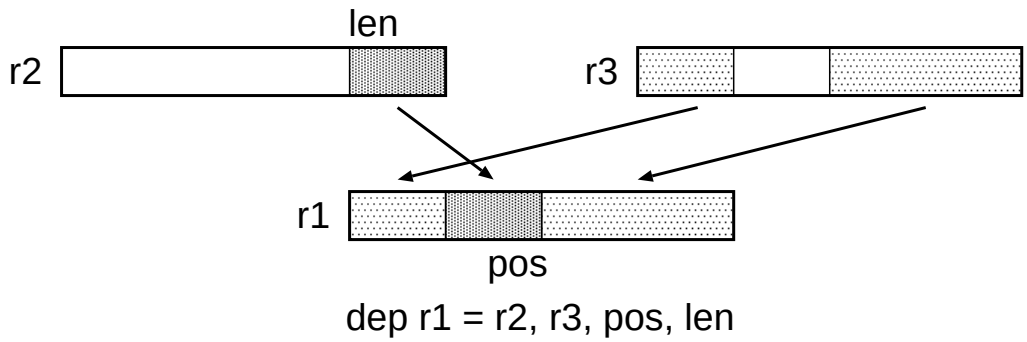


図 17: IA-64 の dep 命令

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．

第7章　性能評価

本章では，キューボイド順レイ・キャスティング法の評価を行う．まず，キューボイド順レイ・キャスティング法のオーバーヘッドを机上で求め，その次にピクセル順，キューボイド順レイ・キャスティング法について Itanium2 サーバ上のプログラムを実装しその描画性能を計測する．

7.1　キューボイド順レイ・キャスティング法のオーバーヘッド

前述のように，キューボイド順レイ・キャスティング法におけるキューボイドの処理手順は，

1. キューボイドの投影
2. スキャン変換
3. ピクセル値計算

である．以下では，それぞれの過程におけるオーバーヘッドの計算量を求める．

キューボイドの投影

キューボイドの投影は，キューボイドの各頂点をアフィン変換することにより行う．頂点 1 つあたりの計算量は，4 行 4 列の変換行列と 4 次元ベクトルの積に 28 (FLOP)，それに加えて，同次座標からスクリーン上の座標への変換に 3 (FLOP) かかり，計算量の合計は 31 (FLOP) である．ボリューム内のキューボイドの個数は，ボリュームの体積 N^3 をキューボイドの体積 $16\times16\times32=8$ (KB) で割った $N^3/8$K である．したがって，投影処理の計算量は 31 (FLOP/頂点) $\times$ 8 (頂点) $\times N^3/8$K である．

全体の計算量に対する投影処理の占める割合を求める．第 4 章で述べたように，サンプリング点あたりの計算量は 13 (FLOP) であった．1 ボクセルあたり 1 度サンプリングされるとすると，全体の計算量は $13N^3$ (FLOP) である．投影処理の計算量をこれで割ると，$(31\times8\times N^3/8\text{K})/(13N^3)\approx0.0023$ である．

投影処理の占める割合は，レンダリング処理全体の 0.23% と十分小さく，無視してよい．

スキャン変換

スキャン変換処理にかかる計算量は視点位置に依存する．スキャン変換では，スクリーン横 (u 軸) 方向に走る各スキャンラインとキューボイド像のエッジの交点の座標を求めるため，その計算量はスクリーン上におけるキューボイド像

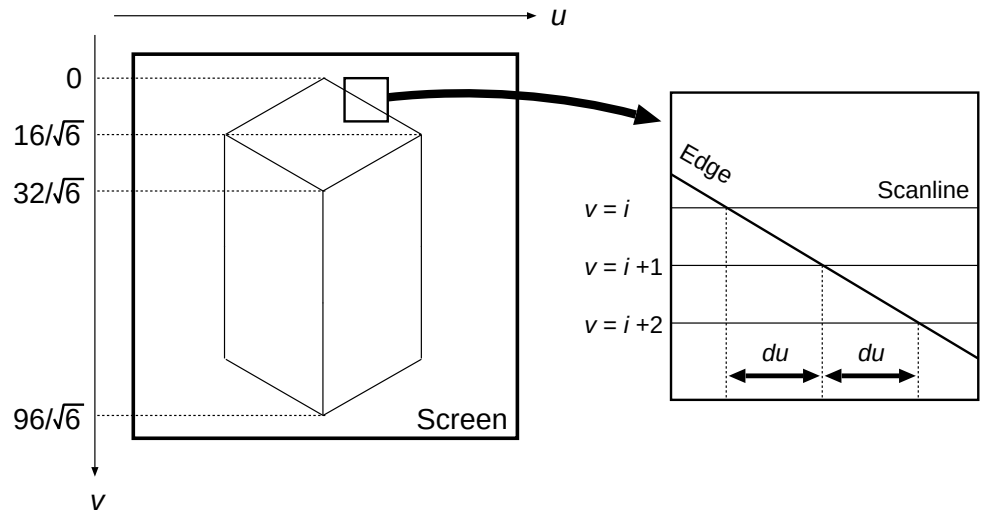


図 18: キューボイドの対角線が v 軸と平行な場合

の縦（v 軸）方向の大きさに比例する．

以下，計算量が最大となる最悪の場合について考える．それ以外の場合は，最悪の場合の計算量で抑えられる．

視点が無限遠点にあるとする．このときの投影方法は平行投影である．最悪の場合は，図 18 左に示すように，キューボイドの対角線がスクリーンの v 軸と平行となるときである．このときの視点位置は，図 14 右に示した，ベクトル長が最小となる場合と同じである．

スキャン変換では，キューボイド像のエッジと，スキャンラインとの交点の座標を計算する．図 18 右に示すように，スキャンラインは u 軸と平行で等間隔に走っているので，座標計算は u 軸方向の変位 du を求め，インクリメンタルに加算することで求められる．

キューボイドあたりの計算量を求める．同図上側から，領域 $0 \le v \le 16/\sqrt{6}$ ではエッジ数は 2，$16/\sqrt{6} \le v \le 32/\sqrt{6}$ では 4，そして，$32/\sqrt{6} \le v \le 96/\sqrt{6}$ では 3 であることが分かる．したがって，計算量は $2 \times 16/\sqrt{6} + 4 \times (32/\sqrt{6} - 16/\sqrt{6}) + 3 \times (96/\sqrt{6} - 32/\sqrt{6}) = 48\sqrt{6}$ (FLOP) である．実行時には得られた u 座標をスキャンラインのループ・カウンタの始点，および，終点として用いるため整数に変換する必要がある．これを考慮に入れると，計算量は 2 倍になる．さらに，9 本のエッジの du を求めるのに 9×3 (FLOP) かかるので，計算量は合計で $27 + 96\sqrt{6}$ (FLOP) である．

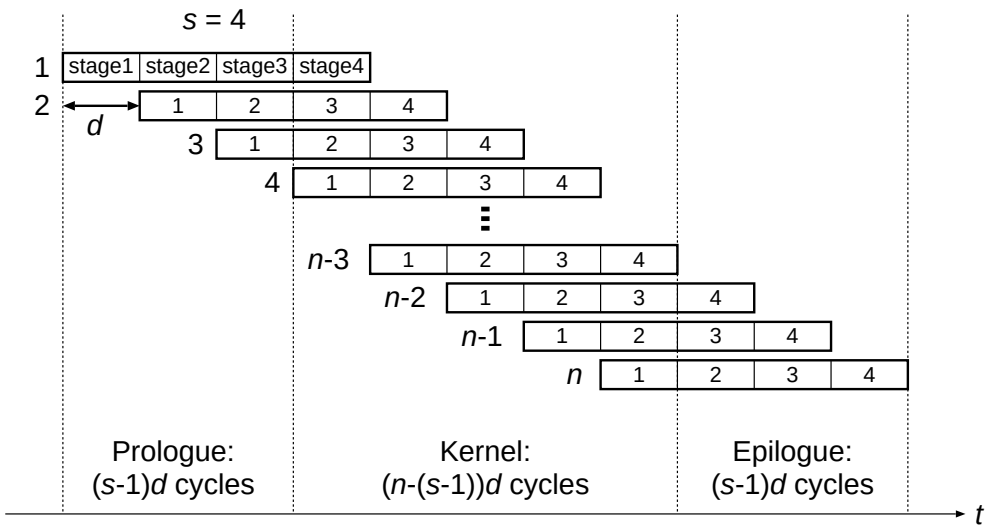


図 19: ソフトウェア・パイプライニング

上記の投影処理と同様に，計算量の割合を求めると，$(27+96\sqrt{6})\times(N^3/8\mathrm{K})/(13N^3) \approx 0.0025$，すなわち，0.25% である．スキャン変換の計算量は十分小さく，したがって無視してよい．

ピクセル値計算

ピクセル値計算の計算量については，ピクセル順，キューボイド順ともに等しく $13N^3$ (FLOP) である．しかし，第 5 章，および，第 6 章で述べたように，キューボイド順レイ・キャスティング法ではベクトル長が短縮し性能が低下する．ベクトル長と描画性能の関係は図 13 で示したとおりである．

ピクセル順のベクトル長がボリュームのサイズ $N = 128, 256, 512, 1024, \cdots$ であるのに対し，キューボイド順のそれはキューボイドのサイズで決まり，最悪で 13.9 まで低下する．これによる性能低下は上記の 2 点に対し比較的深刻である．以下にその性能低下の原因を説明し，キューボイド順レイ・キャスティング法の性能はベクトル長により決まることを示す．

次節の評価に用いたプログラムは，最内側ループをソフトウェア・パイプライニングにより最適化している．一般的に，ソフトウェア・パイプライニングにより最適化されたループは，図 19 に示すように，連続するイタレーションの相異なるステージをオーバーラップして実行される．

まず，最適化されたループについて，実行時のサイクル数を求める．反復回数を n，ステージ数を s，そして，投入間隔を d とすると，求めるサイクル数は，

1. プロローグ部：$(s-1)\,d$ サイクル
2. カーネル部　：$(n-(s-1))\,d$ サイクル
3. エピローグ部：$(s-1)\,d$ サイクル

の和，$(n+s-1)\,d$ サイクルである．ただし，これにはキャッシュ・ミスによるストールは含まれない．

ベクトル長の短縮により，ピクセル値計算における並列度の低いプロローグ部，および，エピローグ部が占める割合が相対的に大きくなり，性能が低下する．以下，具体例を用いてそれを示そう．長さ $N=1024$ の視線について，

1. ピクセル順レイ・キャスティング法において，ピクセル値計算をベクトル長 $n=1024$ のループで実行する場合

と，

2. キューボイド順レイ・キャスティング法において，キューボイド毎に分割して行う場合，すなわち，ベクトル長 $n=32$ のループを $1024/n=32$ 回繰り返すことで求める場合

についてそのサイクル数を比較する．ここで，ベクトル長 $n=32$ は，キューボイドの形状が $16\times16\times32$ のときの最良の場合における平均ベクトル長である．上式にベクトル長を代入すると，前者については $(1024+s-1)\,d$ サイクル，後者は $32\times(32+s-1)\,d$ サイクルである．

パラメータ s，d については，実際のコードからその値を得る．次節の評価に用いたプログラムでは，リスト・スケジューリングにより最適化した結果，ピクセル順，キューボイド順ともにステージ数 $s=9$，投入間隔 $d=5$ であった．これらを代入すると，1. ピクセル順のサイクル数 5160，2. キューボイド順のサイクル数 6400 が得られ，ベクトル長の短縮によりサイクル数が増大することが確認できる．

実際には，これにキャッシュ・ミス発生時のストールが加わるため，実行時のサイクル数はこれより大きくなる．しかし，ライン利用率が最大化されたキューボイド順レイ・キャスティング法では，ボリューム参照時のキャッシュ・ミスはラインあたり 1 回のみ，すなわち，定数回であるので，実行時のサイクル数はベクトル長で決まるといえる．

さて，以上では VLIW プロセッサ上での実行を例に挙げ説明したが，命令レベルでの並列化を動的に行うスーパースケーラ・プロセッサについても同様に，ベクトル長の短縮による性能低下が起こることに注意されたい．

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[1)]		
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

[1)] Multiply and ACcumulate per Second

7.2 実機による評価

ピクセル順，および，キューボイド順レイ・キャスティング法のプログラムを Itanium2 サーバ上で実装し評価した結果を示す．Itanium2 プロセッサの諸元を表 1 に示す．

プログラム

ピクセル順レイ・キャスティング法は図 5 のコードを，キューボイド順レイ・キャスティング法は，それに第 4 章，および，第 5 章で述べた変更を加えたものを用いた．コンパイラは GCC 2.96 で，最適化オプションは-O4 である．ただし，最内側ループについては，前節で述べたようにソフトウェア・パイプライニングにより最適化している．また，ストリーミング SIMD 拡張命令は使用していない．

評価結果

まず，ピクセル順レイ・キャスティング法における最良の場合と最悪の場合について性能を計測する．最良の場合の視点位置は，図 6 左上に示したようにアクセス・パタンがアドレス順になるような位置，一方，最悪の場合は同図左下のように視線がライン方向と垂直になるような位置である．

評価結果を表 2 上側に示す．表内の数値は，実効 MFLOPS 値——描画の計算量 $13N^3$(FLOP) を描画時間で割った値である．また，() は同一の N における，

表 2: 実効 MFLOPS 値

N		128	256	512	1024
PO	最良の場合	1268 (1.0)	1348 (1.0)	1356 (1.0)	1360 (1.0)
	最悪の場合	592 (0.47)	418 (0.31)	326 (0.24)	307 (0.23)
CO	最良の場合	887 (0.70)	877 (0.65)	876 (0.65)	877 (0.64)
	PO の最悪の場合	767 (0.60)	761 (0.56)	763 (0.56)	765 (0.56)
	最悪の場合	683 (0.54)	678 (0.50)	683 (0.50)	685 (0.50)

(MFLOPS)

ピクセル順の最良の場合に対する比である．なお，表中ではピクセル順を PO，キューボイド順を CO と略記している．

次に，キューボイド順レイ・キャスティング法について評価する．キューボイド順の最良の場合は，平均ベクトル長が最大になるときである．このときの視点位置は，ピクセル順における最良の場合と同じである．一方，平均ベクトル長が最小になる最悪の場合については，ピクセル順のそれとは視点位置が異なる．したがって，比較のためキューボイド順レイ・キャスティング法の性能は，1. 最良の場合，2. ピクセル順における最悪の場合，3. キューボイド順の最悪の場合の 3 つの場合について性能を測定する．評価結果は同表下側に示している．

考察

ピクセル順の最良の場合では，N が大きくなるほど，すなわち，ベクトル長が大きくなるほど性能は向上するが，最悪の場合では逆に N が大きくなるほどキャッシュの利用効率が悪化し性能が低下する．前述のように，評価環境のメモリ・レイテンシは 41 サイクルと非常に高速である．そのため，メモリ・レイテンシが支配的となる最悪の場合における性能もそれほど低下していないようにみえる．しかし，汎用 PC のようなレイテンシの大きな環境でボリューム・レンダリングを行おうとすると，最悪の場合の性能はこれよりさらに低下することになる．

一方，前節で述べたようにキューボイド順レイ・キャスティング法の性能はベクトル長で決まる．それぞれの場合について，その性能は N とは独立であることが確認できよう．最良の場合については，ピクセル順のベクトル長が N であるのに対し，ピクセル順では 32 である．したがって，性能は低下する．逆に

ピクセル順における最悪の場合について，ベクトル長は短縮するものの，ライン利用率が最大化されるため性能は向上する．

さて，前述したように，ボリューム・レンダリングにおいて評価すべきは最悪の場合の性能である．レイ・キャスティング法の性能は，ライン利用率が最大の 1 で，かつ，ベクトル長の大きい場合，すなわち，ピクセル順における最良の場合に最大となる．これを 1 とすると，最悪の場合の性能は，最小で 0.23 である．これに対し，キューボイド順の最悪の場合では 0.50 と，ピクセル順より 2.2 倍向上している．

繰り返しになるが，ピクセル順の性能はメモリ・レイテンシで，キューボイド順の性能はプロセッサの演算速度で決まる．したがって，汎用 PC などレイテンシの大きな環境では，性能向上率はさらに高くなる点に注意していただきたい．

第8章　おわりに

従来用いられてきたピクセル順レイ・キャスティング法では，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と呼ぶサブ・ボリュームに分割し，キューボイド順に処理を進めるキューボイド順レイ・キャスティング法を提案した．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．

ただし，提案手法によりメモリの供給能力の不足に起因する性能低下はほぼなくなったが，これだけではプロセッサの最大性能を引き出すには至らない．キューボイド順レイ・キャスティング法には，1. ベクトル長の短縮 と，2. カラム・コンフリクト という 2 つの問題があるからである．従来のようにボリュームを単純な 3 次元配列として定義していると，これらの問題は互いに競合し両方を満足することは難しい．

本稿では，1. ベクトル長の短縮 はボリューム空間，2. カラム・コンフリクトはアドレス空間の問題であり，分離可能である点に注目し，これらの問題を同時に解決することができるアドレス変換を提案した．これにより，ベクトル長は最大化され，カラム・コンフリクトの問題も解決することができる．

Itanium2 サーバ上でプログラムを実装し評価した．今日ではボリューム・レンダリングのリアルタイム描画がすでに実現しており，したがって，描画時間が最大となる最悪の場合の性能で評価する．結果，提案手法により，描画性能は 2.2 倍向上することが分かった．

提案手法により，ボリューム・レンダリングにおいて従来問題となっていたメモリの性能不足による性能低下は解決された．本来，ボリューム・レンダリングにおけるメモリの要求バンド幅は 0.083(B/FLOP) と極めて小さく，システムの主記憶に特別に高速な DRAM を用意する必要はない．提案手法により，汎用 PC を用いて安価にボリューム・レンダリングを実現することが可能となった．

謝辞

本研究の機会を与えて下さり，適切な御指導を賜わりました富田眞治教授に深甚な謝意を表します．

また，貴重な御助言をいただいた森眞一郎助教授，中島康彦助教授，五島正裕助手，小西将人氏に深く感謝致します．

さらに，日頃暖かく御鞭撻下さった京都大学情報学研究科富田研究室の諸兄に感謝致します．

ありがとうございました．

参考文献

[1] Lacroute, P. and Levoy, M.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, *SIGGRAPH'94*, pp. 451–458 (1994).

[2] 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).

[3] TeraRecon VolumePro WWW page: http://www.terarecon.co.jp/products/volumepro_prod.html.

[4] 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).

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

[6] 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).

[7] 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).

[8] Foley, J. et al.: *Computer Graphics: Principles and Practice*, Ohmsha (2001).

[9] Intel: *IA-64 Intel(R) Architecture Software Developer's Manual* (2002).