修士論文

階層格子ボリュームデータの可視化手法

指導教官　　富田 眞治 教授

京都大学大学院工学研究科
修士課程情報工学専攻

藤原 雅宏

平成 10 年 2 月 13 日

階層格子ボリュームデータの可視化手法

藤原 雅宏

内容梗概

科学技術計算結果や医療計測機器からの出力であるボリュームデータの可視化を行う手法としてボリュームレンダリング法が考えられている。ボリュームレンダリング法は計算量が大きく、高速な画像生成にはハードウェアに頼らざる負えないのが現状である。そのため、ボリュームレンダリングを対象とした専用計算機が幾つか開発されているが、それらは均一な立方体の集合で表されたボリュームデータの可視化を目的としている。

しかしながら、科学技術計算に着目すると、空間中の重要な箇所とそうでない箇所で用いる計算格子の密度を適応的に変更することで、より広い範囲をより高い精度で計算する手法が頻繁に採られている。一方、医療計測機器の分野では、近年の CT や MRI を代表とする医療計測機器の進歩により、高解像度の 3 次元データを取得することが可能となった。解像度が増すにつれデータサイズが増大し、その効率的な取り扱いが困難となる。

このようななか、ボリュームデータの格納形態としての階層格子が注目されている。 octree に代表される階層格子は、一様な間隔を持つ空間分解能が異なる複数の格子系により構成されるもので、科学技術計算結果や記憶媒体上の保存形式として優れた特性を持つことが知られている。

この構造を効率的に可視化することは、今後のボリュームデータの肥大化を考慮すると重要なことである。従って、本稿では階層格子上でサンプリングされたボリュームデータを対象とした可視化手法について述べる。

本研究の目的は、ボリュームデータをボリュームスライスと呼ばれる 2 次元平面に分割して管理するボリュームスライス分割型の計算機において、レイキャスティング法を用いた高品位な可視化画像を生成することである。

分割されたボリュームスライスを表現するためのデータ構造として、従来用いられていた 2 次元配列構造に加えランレングス構造、 **quad-tree** 構造の適用を検討した。 ランレングス構造および quad-tree 構造による表現では、ボリュームスライスを格納するために必要なメモリサイズの縮小が期待できる。しかし、これらの構造からはその格納方式に適応した順でのみ高速にデータの

読み出しが可能であるため、レイキャスティング法のような任意の視線に必要なデータをこれらの構造から高速に取得することは困難である。

従って、本研究ではデータの読み出し方式にハードウェアによる実装が可能な規模のアルゴリズムを使用したうえで、視線のボリュームスライスへの入射順序を変換することにより、ボリュームデータへの高速なアクセスを可能とする手法を示した。

この手法の評価のために、3つの構造に対しボリュームデータを格納するために必要な記憶容量と可視化に要する処理時間という観点から分析を行い、平均的な脳画像 (256^3 ボクセル、輝度 256 階調) を用いて評価を行った。

この結果、ボリュームスライスの格納に、2次元配列構造では 65536byte、ランレングス構造では 40450byte、quad-tree 構造では 43228byte 要することが明らかになった。また、ボリュームスライスの劣化を許容することにより、より大きなボリュームスライスを格納することが可能であるが、ランレングス構造の場合は特有の符号化方向に延びるノイズが目立ち、実用面から考慮すれば劣化の許容による大幅なデータサイズの削減は困難であることがわかった。

一方、ボリュームデータへのアクセス回数はボリュームレンダリングの処理速度に影響を与える重要な要素である。2次元配列構造で格納されたボリュームデータの場合、ボリュームデータが格納されているアドレスは計算のみで決定され、データの読み出しに必要なメモリへのアクセス回数は1度だけである。これに対し、ランレングス構造と quad-tree 構造では、視点とボリュームスライスの関係によって異なり、本稿で示すアクセス法を用いた場合、ランレングス構造ではボリュームスライスに入射する視線の間隔に比例し、quad-tree 構造では対数比例することがわかった。

A Hierarchical Grid Volume Data Visualization Method

Masahiro FUJIHARA

Abstract

Volume rendering is an effective method to visualize the result of the scientific computation and output of the medical scanner. It requires highly computation time, so it's difficult to generate images fast without addtional hardware components.

In the scientific computation it's a common way to change the size of the computational grid in order to calculate wider region at high precision. On the other hand, recent advance of the medical equipments such as the CT-scan and the MRI make it possible to provide high resolution three dimensional information. As the resolution increase , it becomes difficult to treat the huge volume data. And it prevents the high-speed visualization.

In these backgrounds, the hierarchical grids for the sampling grid of the volume data receives much attention. The hierarchical grid such as the octree structure is consist of grids which have different resolutions, and it has good characteristics for the storage structure of the result of computer and volume data.

Taking acount of increasing volume size, it's important to treat hierachical grid efficiently. Thus this paper describes the visualization method of the hierarchical volume data.

In this research our goal is to provide a method of hierarchical volume rendering on the volume slice type machine which manages the volume data as a set of two dimensional planes.

I described three data structures to express volume slices. They are run-length and quad-tree structure in addition to traditional two dimensional array.

The size of the volume data can be expected to decrease with adapting Run-length and Quad-tree structure. It's easy to read out voxel data from these structures in their specific order. But random access caused by ray-casting method make it difficult.

Thus this paper described fast volume data access method based on the ray penetration ordering algorithm with simple data fetch algorithm for hardware.

I analyzed these structures from the view of the data size and processing time, and evaluated it by the use of the actual human brain data, it had 256^3 voxels and 256 levels.

The result of the evaluation shows memory size to hold two-dimensional array, run-length structure and quad-tree structure required 65563 bytes, 40450bytes and 67056bytes , respectively.

And it's able to preserve larger volume slice with permitting distortion.But in the case of run-length structure their characteristic noise, parallel with the scan-line, makes images dirty. Therefore it's impossible to decrease much information in the practical use.

Above all the number of memory acceess to get the voxel data will make great effects on the processing time, so it's a notable factor also. In the case of two-dimensional array the address of the stored data can be decided with calculation. Thus the number of memory access to read out the voxel data is only once. In run-length structure it's proportion to the interval of ray penetration. And in quad-tree structure it's proportion to binary logarithm of it.

階層格子ボリュームデータの可視化手法

目次

1 はじめに

ボリュームレンダリング法は、3 次元データの内部構造の可視化が可能であるため、医療画像の生成や科学技術計算結果の解析のための有力な手法として注目されている。

しかしながら、ボリュームデータを格納するためには莫大な記憶容量が必要となる。また、可視化処理に要する計算量は大きく、ボリュームデータの高速な可視化は困難である。そのため、従来から *ReVolver*[1, 2] や CUBE[3, 4] など、ボリュームレンダリングに特化した、さまざまな専用計算機の開発が行われてきた。これらの専用計算機では、立方体格子構造を持つボリュームデータを対象とした高速な可視化処理を実現している。

一方で科学技術計算では、広い領域に対して小さな計算量で高い精度の計算を行うために、解領域の重要度に応じて適応的に計算格子の幅を変える手法がとられる [5]。このように、計算格子が一様でないボリュームデータを、従来の手法で可視化する場合、立方体格子上の点にデータをとり直すリサンプリングによる実現が考えられるが、それではボリュームデータが激増してしまい現実的ではない。

また、近年の CT や MRI などに代表される 3 次元医療計測機器の進歩により、高解像度ボリュームデータを取得することが可能になり [6]、可視化すべきボリュームデータの大きさは増大する傾向にあり、それに伴い肥大化したデータの取り扱いが困難になりつつある。

このような現状において、ボリューム空間を従来の立方体格子ではなく、格子の密度を適応的に変えることが可能である階層格子構造を用いて表現することは有効な手法であると言える。

本稿では、ボリューム空間をボリュームスライスと呼ばれる 2 次元平面の集合として扱うボリュームスライス分割方式のボリュームレンダリング専用並列計算機における、階層格子構造のボリュームレンダリング法による可視化手法を提案する。

以下、第 2 章では、ボリュームレンダリングの概要とボリュームデータを階層的に扱う手法について述べ、第 3 章で、提案する階層格子の可視化手法について述べ、第 4 章において、 第 3 章で示した手法について評価を行い、第 5 章でまとめを述べる。

2　ボリュームレンダリング法

本章では、本研究の背景であるボリュームレンダリング法の原理および、階層格子構造のボリュームデータについて述べる。

2.1　ボリュームレンダリング法の原理

ボリュームレンダリング法は、半透明物質における光の透過モデルに基づいて考案された可視化手法である。ボリュームレンダリング法において、物質は色と透明度が割り当てられたボクセルと呼ばれる3次元構造体に置き換えられ、ボクセルの集合がスクリーン上のピクセルに与える影響を計算することにより可視化が行われる。

本稿では、レイキャスティング法を用いてボリュームレンダリングを行う。レイキャスティング法は、図 2.1のように視点からスクリーン上の各ピクセルに向けて発した視線がつらぬくボクセルの色と透明度を式 (2.1) を用いて順次計算して行く方法である。

立方体格子構造を持つボリューム空間をつらぬく視線上のボクセルが持つ色と透明度を視点に近い順にそれぞれ $c_0, c_1, \ldots, c_n$ および $\alpha_0, \alpha_1, \ldots, \alpha_n$ とするとき、ピクセル値 C は式 (2.1) により求めることができる [7]。

$$C = \sum_{i=0}^{n-1} c_i \cdot (1-\alpha_i) \prod_{j=0}^{i-1} \alpha_j \tag{2.1}$$

また漸化式を用いて、i 番目のボクセルをつらぬいた時の色と透明度は次のように表すことができる。

$$\begin{cases} A_i = A_{i-1} \cdot \alpha_i \\ C_i = C_{i-1} + c_i \cdot (1-\alpha_i) \cdot A_{i-1} \end{cases} \tag{2.2}$$

ただし、初期値は $A_{-1} = 1$, $C_{-1} = 0$ で、 $C = C_{n-1}$ である。

レイキャスティング法は、高品位な可視化画像を生成することができる (第 A.1.2節参照)。しかしながら、メモリ上に格納されているボクセルデータがサンプリング[1)]される順序は、視線の方向に依存する。従って、任意の視線に対

[1)] メモリ上のボクセルデータを読み出す処理をサンプリングと呼ぶ。

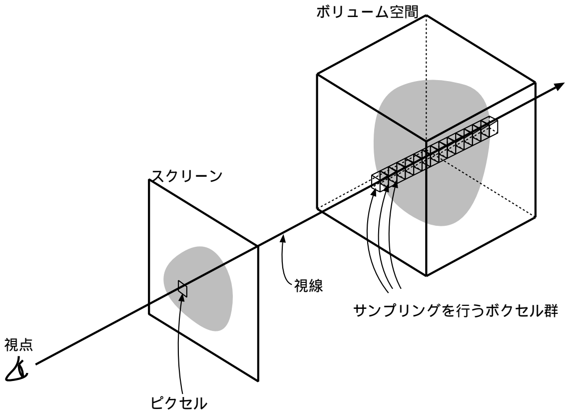


図 2.1: レイキャスティング法

してはボクセルのサンプリング順は、ボリュームデータの格納順とは一致しない。このことにより次の問題が生じる。

- アドレス計算の必要性

  視線がつらぬくボクセルデータが存在するアドレスを算出するための処理が必要となる。

- 連続読み出しが不可能

  一般に、記憶装置上のデータは連続的に読み出すことにより、高速に読み出すことが可能であるが、レイキャスティング法ではこの連続性が確保できない。

これらの問題は、レイキャスティング法によるボリュームレンダリングの高速化を妨げる要因となる。

2.2 サンプリング格子

本稿では、階層格子構造を持つボリュームデータの可視化を行う。階層格子構造は、空間分解能が異なる複数の立方体格子系により構成された格子であり、代表的なものに octree 構造がある。階層格子構造は、次のような優れた特徴を有する。

- 汎用性

  問題の性質への依存が小さく、科学技術計算の結果や 3 次元医療データの管理など比較的広い分野への応用が可能な構造を持つ。

- データ保存形式との親和性

一般にボリュームデータの記憶媒体からの読み出しには、多くの時間を必要とする。ボリュームデータを領域の複雑さに応じて階層的に保存することにより段階的に可視化することができ、システムの利用者は短時間にボリュームデータの概要を観察することができる [5]。

2.3　ボリュームデータとその構造

ボリュームデータを表現するためのデータ構造は、その可視化アルゴリズムや問題に強く依存し、ボリュームデータのサイズや可視化に必要な時間に大きな影響を与えることから、様々なデータ構造が提案されている。本節では、その中でも頻繁に利用される 3 次元配列構造、ランレングス構造、 **octree** 構造について述べる。

2.3.1　3 次元配列構造

ボリューム空間を立方体格子構造をとると考え、格子点のとる値を 3 次元配列の該当する要素に格納する方式である。単純なデータ構造であるためソフトウェアだけでなく、ハードウェアによるボリュームレンダリングにも広く用いられており、この構造を持つ CUBE4 では大きさが 512^3 ボクセルのボリュームデータを平行投影法を用いて 30frames/sec で可視化できるとされている [4]。

しかしながら、ボリュームデータを格納するために必要なデータサイズは、ボリューム空間の大きさに比例する。従って、階層格子構造を持つボリュームデータを 3 次元配列構造に格納する場合には、ボリュームデータの内容に関わりなく、最密格子サイズ [1)]に応じたメモリサイズが必要になる。

2.3.2　ランレングス構造

ボリュームデータをランレングス法 [2)]により符号化することにより、データサイズの削減と可視化処理の高速化を図る手法が提案されている。

代表的なものに、ボリューム空間を同一のボクセルデータを持つボクセルで組織化し、データサイズの節約と計算時間の短縮を狙った **STICKS**[9] がある。

STICKS では、 x, y, z の各座標軸方向に同一のボクセル値を持つボクセルを 1 つにまとめることにより、 n^3 個の格子点を $O(kn^2)$ で保持することができ

---

1) 階層格子を構成する格子系の中で、空間分解能が最も高い格子の大きさを表す。

2) 連続して同一のデータが出現するとき、そのデータの値と連続するデータ数により情報を表現する方法である。データが連続する個数をラン長と呼び、ラン長とその内容からなる 2 項組をランと呼ぶ。

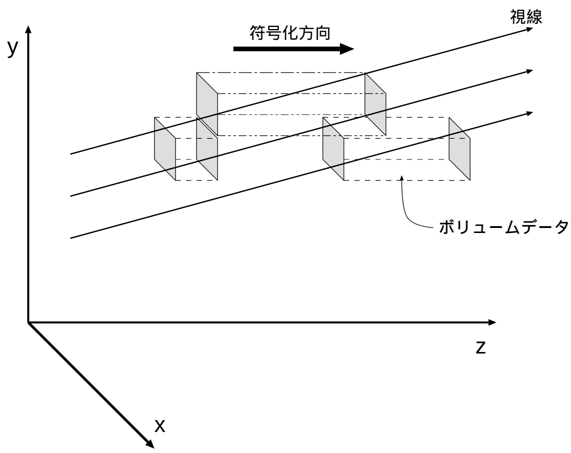


図 2.2: ボリュームデータの STICKS 表現

る。ここで k は平均ラン数をあらわし、最悪の場合には k は n と一致し $O(n^3)$ となる。

STICKS で表現されたボリュームデータの例を図 2.2に示す。この例において、ボリュームデータは Z 軸方向に符号化される。視線がつらぬくボリュームデータの取得に必要なメモリへのアクセス回数や必要な計算回数は、3 次元配列を用いたものよりも少なくてすむことから高速化が期待できる。

2.3.3 octree 構造

octree 構造はボリューム空間の 8 等分割を行い、各部分空間を代表するボクセル値を節点に保持することにより構成される。さらなる分割が必要な部分空間に対しては 8 等分割を繰り返し、分割が生じた部分空間を表す節点の子節点に割り当てることにより、ボリューム空間を表現することができる。

この octree 構造を利用して、ボリューム空間中のボクセルデータがない透明な領域の情報を保持することにより計算時間の短縮を狙ったデータ構造とそのデータ構造を利用したレイキャスティング法 **VSPOT** が提案されている [10]。

この手法では、隣接する 8 つのボクセルを octree の葉節点に格納する。1 つの節点の中のボクセルが全て透明ボクセルであれば、その節点の親節点にデータが無いことを表す 0 を格納し、1 つでも透明でないボクセルが存在すれば、1 を格納する。この操作を繰り返すことにより、透明ボクセルの存在を早期に知ることができる階層化されたボリュームデータを得ることができる。

図 2.3に、このようにしてできた octree 上を通過する視線のピクセル値を計算する過程を、2 次元平面を用いて表した例を示す。

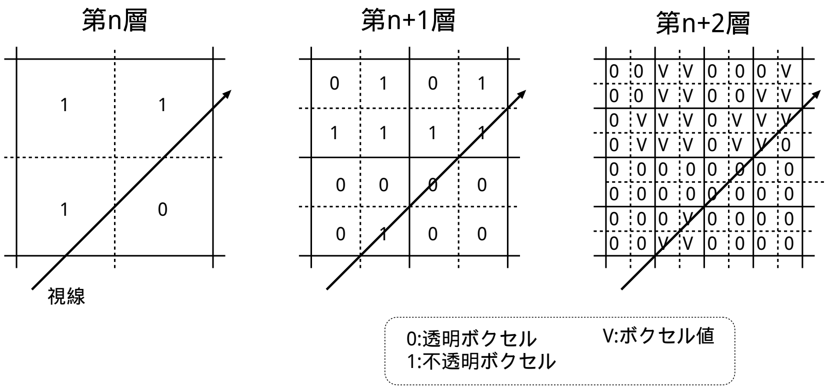


図 2.3: octree 構造を用いた透明ボクセルの表現

VSPOT では、可視化処理の対象を第 n 層の格子として、視線が通過するボクセルデータを読み込む。そのボクセルが示す値が 0 の場合は、その位置に不透明なボクセルは存在しないので計算の必要は無いと判断し、視線がつらぬく次のボクセルデータに処理を移す。ボクセルが示す値が 1 の場合は、可視化処理の対象を第 $n+1$ 層の格子として該当箇所を同じ方法で調べる。

この操作を繰り返すことにより、不透明なボクセルデータが存在する領域のみに対して演算を行うことが可能となる。

2.4 ボリュームスライスのデータ構造

ボリュームデータを特定の座標軸に垂直な平面で分割した平面の集合として扱うことで平面間での処理の並列性を抽出し、高速化を図る手法が提案されている [1, 16]。この平面をボリュームスライスと呼ぶ。

本節では、ボリュームスライスを表現するためのデータ構造として 2 次元配列構造、ランレングス構造、 quad-tree 構造について述べる。

2.4.1 2 次元配列構造

2 次元配列構造は、ボリュームスライスを表現する最も基本的な構造である。ボリュームスライスを 2 次元配列構造で表現する場合、図 2.4のように、最密格子サイズでボリュームスライスを分割しメモリ上の連続したアドレス空間に格納する。

一般に、配列の大きさが決まればボクセルデータの格納されているメモリ上のアドレスを計算により容易に求めることができるため、ボクセルデータへのランダムアクセスが可能な構造である。

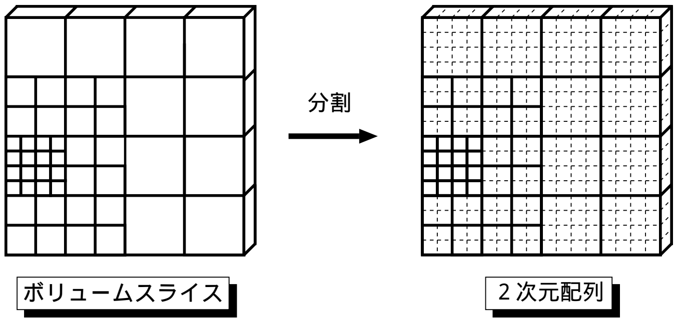


図 2.4: 2 次元配列構造への分割

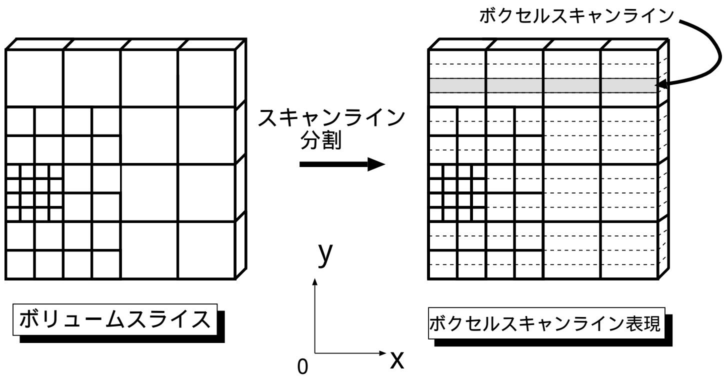


図 2.5: ボクセルスキャンラインへの分割

2.4.2 ランレングス構造

ランレングス構造は、ボクセルデータと同一ボクセルデータが連続して現れる長さであるラン長からなるランの集合である [9]。本稿では図 2.5のようにボリュームスライスを最密格子サイズで x 軸方向に分割する。この分割されたデータをボクセルスキャンライン [16] と呼び、全ボクセルスキャンラインをランレングス構造で表現することでボリュームスライスを構成する (図 2.6)。

この構造において、ボクセル値へのアクセスはボクセルスキャンライン方向への順次アクセスが適しており、ボクセルに対するランダムアクセスが行われた場合は、ボクセルの x 座標値に比例した数のランを読み飛ばす必要が生じるため、ボクセル読み込みのレイテンシが大きくなる。

2.4.3 quad-tree 構造

quad-tree 構造は図 2.7のようにボリュームスライスを 4 等分割し、ボクセルデータを木の節点の各要素に割り当て、さらに分割が必要な領域に関しては 4 等分割を繰り返すことにより平面を構成する [8]。本稿では、 quad-tree 構造に

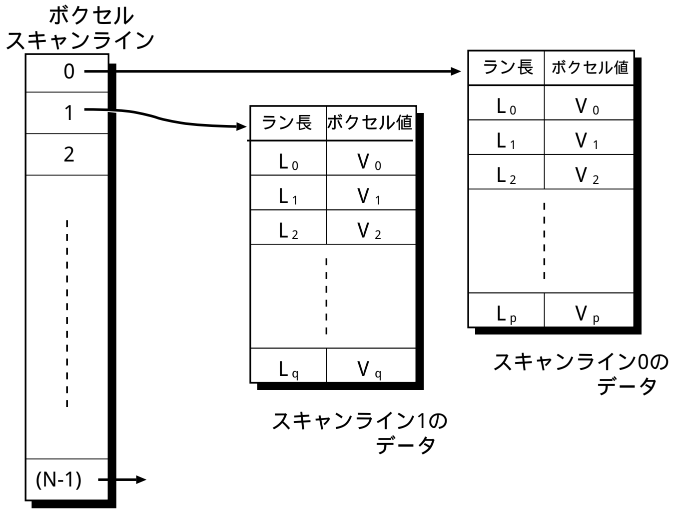


図 2.6: ランレングス構造によるボリュームスライスの表現

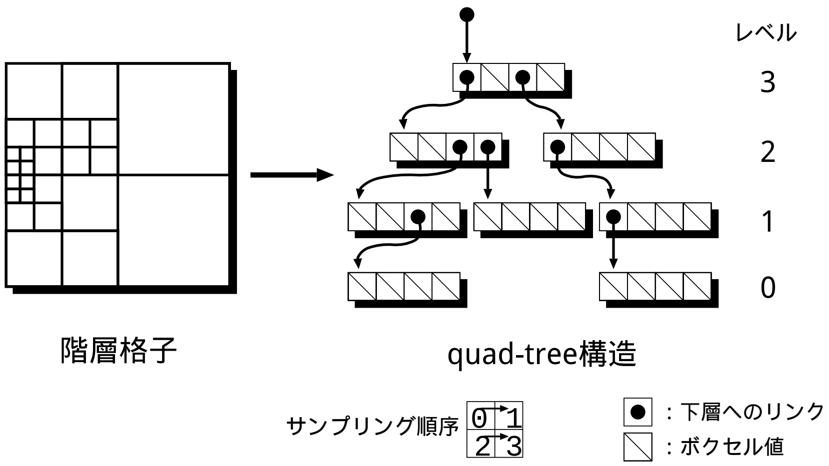


図 2.7: ボリュームスライスの quad-tree 表現

おいてスライスの分割回数をレベルと呼び、最上位層から順にレベル $1, 2, \ldots, N$ と定義する (図 2.7)。また各節点は 4 つの要素を持ち、下位レベルの節点へのリンクもしくはボクセルデータを格納する。

quad-tree 構造により表現されたボリュームスライスへのアクセスは、親子，兄弟関係にある節点が表すボクセルへのアクセスは低レイテンシで行えるが、ランダムアクセスには不向きであり、最悪の場合、木の高さである N 回のメモリアクセスを要する。

2.5 ボリュームレンダリング専用並列計算機 *ReVolver*/C40

ReVolver/C40 は、レイキャスティング法を用いて立方体格子構造を持つボリュームデータの可視化を行うボリュームレンダリング専用並列計算機である。

ボリュームデータをボリュームスライスの集合として管理するボリュームスライス分割方式の並列計算機であり、図 2.8のように各ボリュームスライスにプロセッサを割り当てる。

2.5.1 *ReVolver*/C40 の特徴

ReVolver/C40 が持つ特徴をあげる。

- 透視投影法による画像生成
  ボリュームデータを透視投影法で可視化することにより、物体の遠近感を正しく表現することができる (第 A.1.1節参照)。
  透視投影法では任意方向の視線が発生する。そのためメモリアクセスが不規則になり、ボリュームデータの効率的な読み出しが困難となる。
- 高品質な画像生成
  レイキャスティング法による画像生成を行うため、高品質な画像を得ることができる (第 A.1.2節参照)。また、シェーディング処理を行うことにより、光源を考慮した現実感のある画像を生成できる。
- 高速描画
  可視化対象を理解、解析する上で物体の回転や拡大処理を実時間で行うことは重要である。特に、科学技術計算では可視化結果を参考に計算に用いるパラメータを調整することが多く、計算結果の高速描画により作業時間の短縮を図ることができる。*ReVolver*/C40 では、512^3 ボクセルのボリュームデータを透視投影法を用いて 94.4ms/frame で可視化することができる [11]。

2.5.2 *ReVolver*/C40 アーキテクチャ

ReVolver/C40 のアーキテクチャの特徴は次のとおりである。

1. サンプリング方法の単純化
   従来のサンプリング方法は、視線の方向に対して等間隔にボクセルをサンプリングするものであった。しかし、図 2.8 (a) のようなバンク構成でこのサンプリング法を用いるとコンフリクトを生じてしまう。そこで、*ReVolver*/C40 では主軸等間隔サンプリング法を用いている [1]。この手法においては

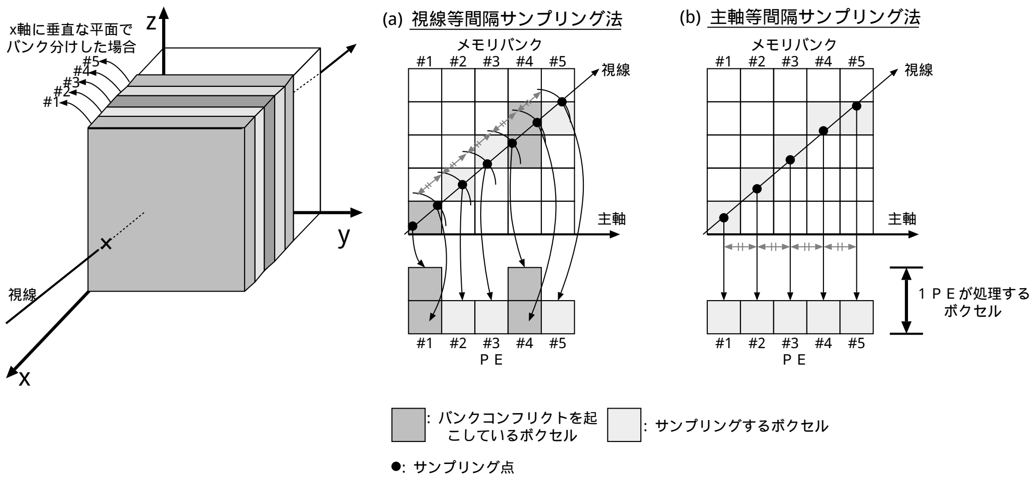


図 2.8: ボクセルのサンプリング方法

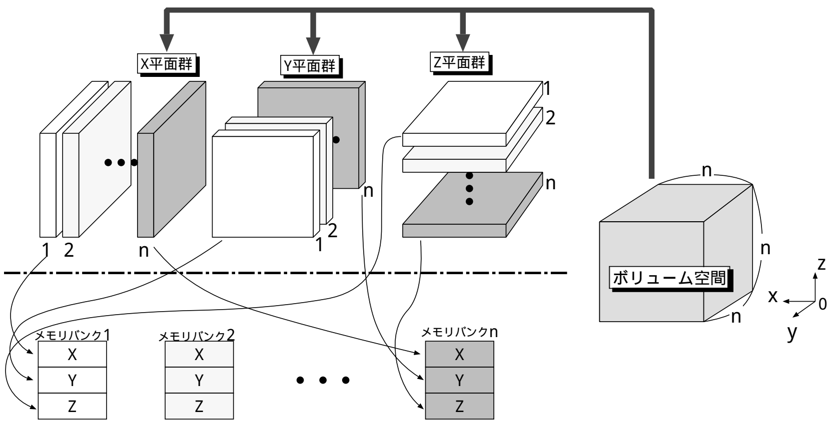


図 2.9: バンクコンフリクトフリーなメモリ構成

図 2.8 (b) のように主軸 [1]の方向に等間隔にサンプリングを行う。

2. バンクコンフリクトフリーなメモリ構成

主軸に垂直な平面でボリューム空間を分割したボリュームスライスを異なるメモリバンクに格納することにより、その主軸に対してバンクコンフリクトフリーにアクセスすることができる。

しかし、主軸は 1 つには定まらないので、図 2.9のようにボリュームデータを 3 重化して、主軸が x, y, z 軸のそれぞれの場合に対してバンクを設ける。

この構成により、任意の視線に対してその主軸に対応したメモリバンクか

[1] 視線ベクトルの成分の中で絶対値が最大であるものを選び、その座標軸を主軸と定義する。

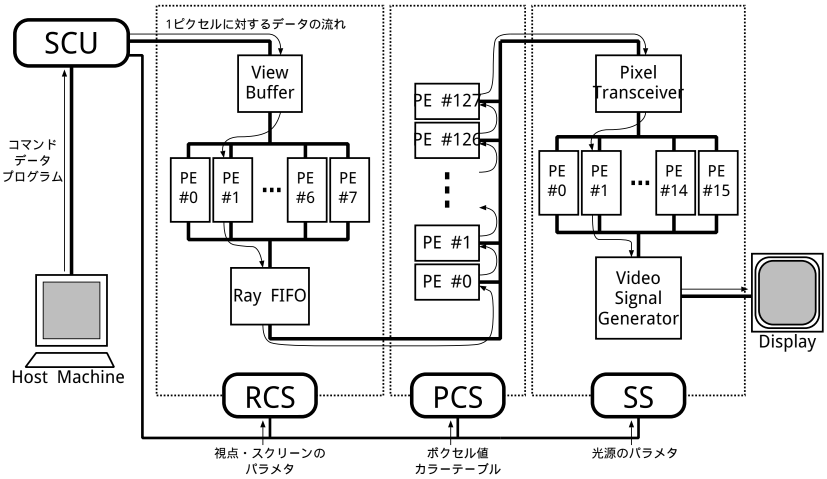


図 2.10: $ReVolver$/C40 の全体構成

らボクセルデータを読み出すことができる。

2.5.3 $ReVolver$/C40 の全体構成

図 2.10 に示すように、$ReVolver$/C40 は、レイキャスティングステージ、ピクセル値計算ステージ、シェーディングステージの 3 つのステージと、全体の制御を行うシステムコントロールユニットから構成される。各ステージが、可視化処理をパイプラインで行うことにより、高速化を図っている。

- システムコントロールユニット (System Control Unit : SCU)
  $ReVolver$/C40 全体の制御や各ステージの状態を監視するユニットである [12]。ホストマシンからコマンド、プログラム、ボリュームデータを受け取り、適切なステージに転送するのが主な役割である。
- レイキャスティングステージ (Ray Casting Stage : RCS)
  スクリーン上のすべてのピクセルについて、視点からピクセルへ向う視線ベクトルを生成し、ピクセル値計算ステージへ転送する。スクリーンを均等に分割し、各プロセッサエレメント (PE) がそれぞれ分割された領域上のピクセルに対する視線生成処理を行うことで処理の並列化を行う [13]。
- ピクセル値計算ステージ (Pixel Calculation Stage : PCS)
  PCS の各 PE は、図 2.10のようにパイプライン状に接続されており、分割されたスライスが割り当てられる。
  RCS で生成された視線に対し、視線上のボクセルをサンプリングしてボクセル値を読み出し、その値から属性テーブルを参照して色と透明度に変換し、式 2.2を用いてピクセル値計算を行う [14]。

- シェーディングステージ (Shading Stage : SS)
  PCS で計算されたピクセル値に対して、 depth gradient shading 法を用いてシェーディングを行う [15]。スクリーン分割を行うことで、シェーディング処理を 16 台の PE で並列実行させる。

3　階層格子ボリュームデータの可視化

本章では、ボリュームスライス分割方式の並列計算機において、階層格子構造を持つボリュームデータをレイキャスティング法により可視化する手法について提案する。

従来のように、階層格子構造で表現されたボリュームスライスを 2 次元配列構造で格納する場合、格納するために必要となるメモリサイズが巨大になり、大きな階層格子構造を扱うことは困難である。

したがって、本稿では 2 次元配列構造の他に、ランレングス構造、 **quad-tree** 構造の 3 種のデータ構造によりボリュームスライスを表現することにより、低容量で格納する手法を提案する。

3.1　ボリュームスライスの生成

ボリュームスライス分割方式をとる計算機においては、ボリュームデータからボリュームスライスを抽出する必要がある。本節では、 octree で表現されれた一辺の大きさが 2^n のボリューム空間を z 軸に垂直な方向に分割した場合に、 $slice$ 番目のボリュームスライスを得るために必要な手続きを記述する。

なお、階層格子についてはアプリケーションによりデータ構造が大きく異なるのでここでは述べない。

3.1.1　2 次元配列構造の生成

Octree 構造を持つボリュームデータから 2 次元配列構造で表現された $slice$ 番目のボリュームスライスを取得する手順を次に示す。ただし、 WriteMemory(adr, $value$) はメモリ上のアドレス adr に値 $value$ を格納する関数であり、 Get3DAddress(x, y, z) は、 octree 上の 3 次元座標値をアドレスに変換する関数であり、図 3.1(a) に示すサンプリング順序でデータが格納されている場合は、 $2^2 \times z + 2^1 \times y + 2^0 \times x$ となる。

Algorithm 1 Octree 構造から 2 次元配列構造の生成

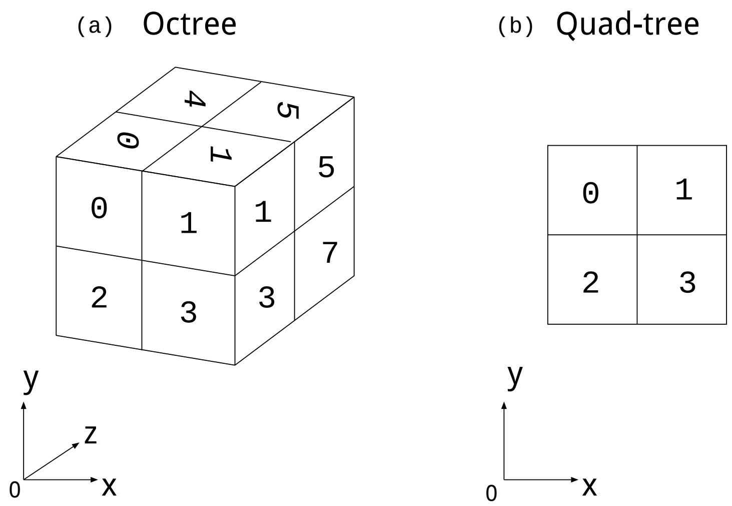


図 3.1: ボクセルデータ格納順序

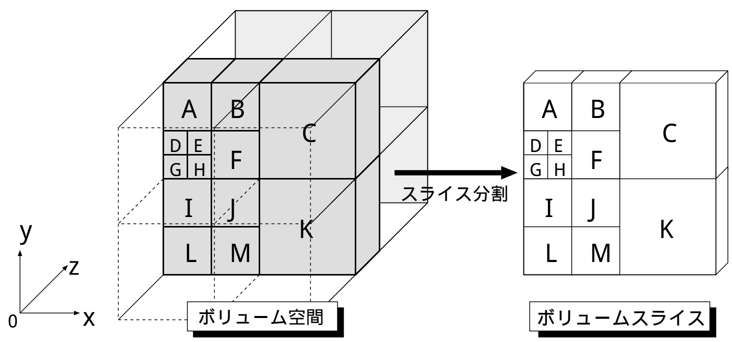


図 3.2: Octree からの quad-tree の抽出

function Generate2DArrayfromOctree()
begin
  for $y_p = 1$ to スクリーンの高さ
    for $x_p = 1$ to スクリーンの幅 **begin**
      ptr = 探索対象となる Octree の根節点へのポインタ;
      $i = n - 1$;
      while $i \geq 0$ **do begin**
        ptr = ReadMemory(ptr + Get3DAddress($slice_i$, y_i, x_i);
        if ptr がボクセルデータ **then break**;
        if ptr が子節点へのリンク **then** $i = i - 1$;
      end
      WriteMemory($Get2DAddress(2^n \times y + x) \times SizeofVoxel$, ptr);
    end
end

3.1.2 quad-tree 構造の生成

octree 構造の根節点へのポインタ ptr を入力としてとり、 quad-tree の生成を行い、生成された quad-tree の根節点を返す関数は次のように定義できる。

quad-tree 構造は、 octree 構造をもつボリュームデータの z 軸方向の探索範囲を当該スライスを示す $slice/2^{level} \bmod 2$ に限定した上で、 octree を再帰的に探索することによって取得することができる。

Algorithm 2 octree からの quad-tree の生成

```
function GenerateQuadTreefromOctree(ptr, x, y, level)
{ ptr : 探索を開始する節点を指すポインタ }
{ x, y : 探索を行う領域を指す。 x, y ∈ {0, 1} }
{ level : 探索を行う quad-tree のレベル }
begin
    ptr = ReadMemory(Get3DAddress(x, y, slice/2^level mod 2));
    if ptr がボクセル値 then
        return  ptr;
    return MakeQuadTree(
        GenerateQuadTreefromOctree(ptr, 0, 0, level − 1),
        GenerateQuadTreefromOctree(ptr, 1, 0, level − 1),
        GenerateQuadTreefromOctree(ptr, 0, 1, level − 1),
        GenerateQuadTreefromOctree(ptr, 1, 1, level − 1)
    );
end
```

ここで、 MakeQuadTree(a, b, c, d) は quad-tree の節点を生成し、節点の 4 つの要素にそれぞれ左から a, b, c, d を格納し、生成した節点へのポインタを返す関数である。

この手順を適用することにより、図3.2のようにoctree 構造のボリュームデータから任意のボリュームスライスを quad-tree 構造として抽出することができる。

3.1.3 ランレングス構造の生成

octree 構造のボリュームデータをアルゴリズム 2を用いて quad-tree 構造のボリュームスライスに変換する。このボリュームスライスをスキャンラインごとに再帰的に走査することによってランレングス構造を取得することができる。

ここでは、 quad-tree を x 軸方向に走査する場合を考え、レベル l の節点で

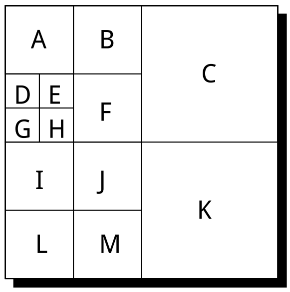


図 3.3: quad-tree の例

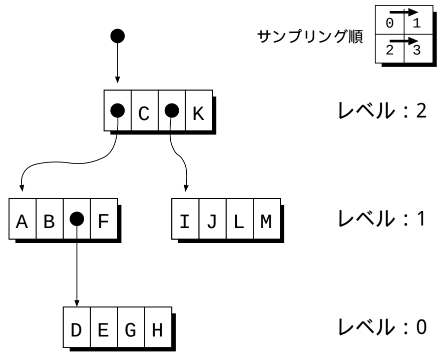


図 3.4: quad-tree 構造

得たボクセル値は、 x 軸方向に 2^l の長さを持つボクセルとして出力する。

以上まとめると、次のアルゴリズムにより quad-tree 構造からランレングス構造を抽出することが可能である。

Algorithm 3 quad-tree からのランレングスデータ抽出

function TraverseQuadTree(ptr, x, y, $scanline$, $level$)
{ ptr : 探索を開始する節点を指すポインタ }
{ x, y : 探索を行う領域を指す。 x, $y \in \{0, 1\}$ }
{ $scanline$: ランレングス情報を取得するスキャンライン番号 }
{ $level$: 探索を行う quad-tree のレベル }
begin
  ptr = ReadMemory($x + y \times 2$);
  if ptr がボクセル値 **then**
    return $< ptr, level >$;
  FIFO $\leftarrow$ TraverseQuadTree(ptr, 0, $scanline/2^{level}$ mod 2, $scanline$, $level-1$);
  FIFO $\leftarrow$ TraverseQuadTree(ptr, 1, $scanline/2^{level}$ mod 2, $scanline$, $level-1$);
end

この実行結果、 FIFO [1]には指定されたスキャンラインのランレングス情報が格納される。図 3.3のボリュームスライスを上記のアルゴリズムにおいて、スキャンラインごとにランレングスデータを抽出した結果を表 3.1に示す。

1) First-in First-Out メモリ構造

表 3.1: FIFO の内容

スキャンライン	項目	FIFO の内容				スキャンライン	項目	FIFO の内容			
0	レベル	1	1	2	-	4	レベル	1	1	2	-
	ボクセル値	A	B	C	-		ボクセル値	I	J	K	-
1	レベル	1	1	2	-	5	レベル	1	1	2	-
	ボクセル値	A	B	C	-		ボクセル値	I	J	K	-
2	レベル	0	0	1	2	6	レベル	1	1	2	-
	ボクセル値	D	E	F	C		ボクセル値	L	M	K	-
3	レベル	0	0	1	2	7	レベル	1	1	2	-
	ボクセル値	G	H	F	C		ボクセル値	L	M	K	-

3.2 視線入射順制御

レイキャスティング法でボリュームレンダリングを行う場合、スクリーン上の各ピクセルに対して、算出された視線ベクトルがつらぬくボクセルデータを各ボリュームスライスから読み出す処理 (サンプリングと呼ぶ) が必要である。しかしながら、レイキャスティング法により生じた視線ベクトルは視点、ピクセルおよびボリューム空間の位置関係により変化するため図 3.5のようにボリュームスライス上の各データ構造に対するランダムアクセスが生じる。

2 次元配列構造では、ランダムアクセスに対しても高速なボクセルデータへのアクセスが可能であるが、ランレングス構造および quad-tree 構造においては、第 2.4節で述べたように各データ構造に適応した順序でのアクセスに関してのみ低レイテンシでのデータ取得が可能である。そのため、ボリュームスライスに対するランダムアクセスが必要なレイキャスティング法では処理速度の低下が避けられない。

本章では、視線のボリュームスライスへの入射順序を制御することにより、ボリュームスライス上をつらぬく視線が描く軌跡を常にボクセルスキャンラインに平行かつ一定方向にすることで、データのアクセス順序を制限する手法を提案する。

3.2.1 平行投影法

ここでは、ボリュームデータは図 3.6のように x, y, z 空間の各軸に垂直に位置し、 z 軸に垂直な平面によってボリュームスライスに分割されているものとする。また、座標原点からスクリーンの左上へのベクトルを$\mathbf{S}_b = (b_x, b_y, b_z)$、

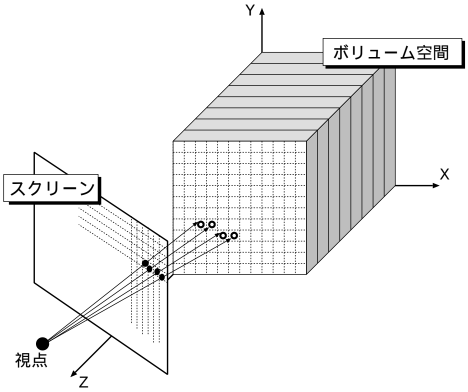


図 3.5: ボクセルデータへのランダムアクセス

スクリーンの水平方向の単位ベクトルを$\mathbf{S}_h = (h_x, h_y, h_z)$、スクリーンの垂直方向の単位ベクトルを$\mathbf{S}_v = (v_x, v_y, v_z)$ と定義する。

ここで、実スクリーンに対し次の写像 f で定義される仮想スクリーンを想定する。仮想スクリーン上の座標 (p, q) に対して q 番目のスキャンラインから出される視線は任意のボリュームスライス上で x 軸に平行な軌跡を持つ。

$$f : \begin{pmatrix} s_x \\ s_y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ k_z & 1 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix} \tag{3.1}$$

ここで k_z は、

$$k_z = - \begin{vmatrix} h_y & h_z \\ n_y & n_z \end{vmatrix} \bigg/ \begin{vmatrix} v_y & v_z \\ n_y & n_z \end{vmatrix} \tag{3.2}$$

である。

従って、仮想スクリーン上のスキャンラインに平行に視線を生成することにより、任意のボリュームスライスに対して常にボクセルスキャンライン方向に視線を入射することができる。また、次のスキャン方向の判別式 D の符号により、常に一定方向に視線を入射することができる（第 3.4.1 節参照）。

$$D = \frac{1}{n_z} \left(\begin{vmatrix} h_x & h_z \\ n_x & n_z \end{vmatrix} + k_z \begin{vmatrix} v_x & v_z \\ n_x & n_z \end{vmatrix} \right)$$

通常の視線の入射順と、入射順序が制御された結果、生成される視線の様子を図 3.7に示す。ここでスキャンラインの傾きは $-k$ となり、方向は D の符号により決定する。

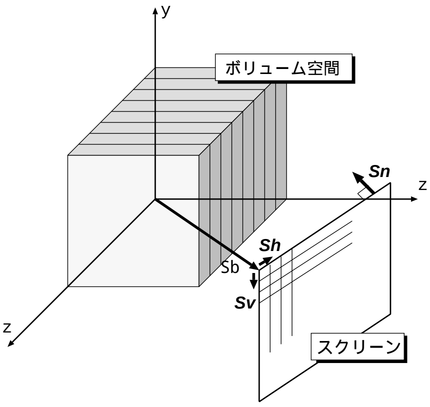


図 3.6: 平行投影法によるレイキャスティング

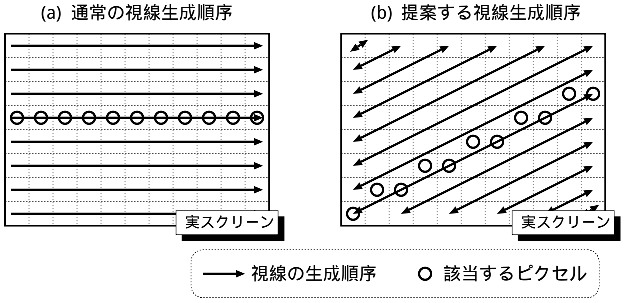


図 3.7: 視線入射順序制御

3.2.2 透視投影法

ボリュームデータは、平行投影法の場合と同様に各座標軸に垂直に置かれており、z 軸に垂直な平面によってボリュームスライスに分割されているとする（図 3.8）。視点の位置ベクトルを $\mathbf{E}$、視点からスクリーンの左上へのベクトルを $\mathbf{S}_b = (b_x,\ b_y,\ b_z)$ とし、平行投影法と同様に式（3.1）の写像 f で仮想スクリーンを定義する。ただし k_z は、

$$k_z \quad = \quad - \begin{vmatrix} h_y & h_z \\ b_y & b_z \end{vmatrix} \bigg/ \begin{vmatrix} v_y & v_z \\ b_y & b_z \end{vmatrix} \tag{3.3}$$

とする。

透視投影法においてスクリーンを構成するベクトルである $\mathbf{S}_h$ と $\mathbf{S}_v$ を次のように制限することにより、仮想スクリーン上のスキャンラインに沿って視線を生成すれば、任意のボリュームスライスに対して常にボクセルスキャンライン方向に視線を入射することができる。

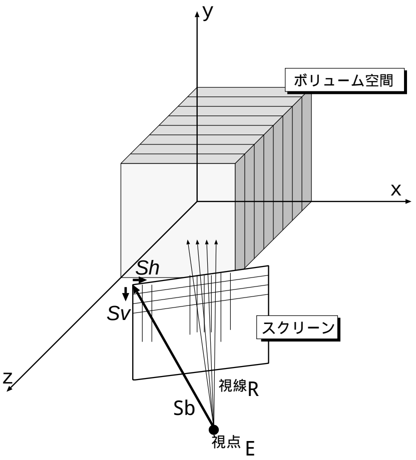


図 3.8: 透視投影法によるレイキャスティング

$$\begin{vmatrix} h_y & h_z \\ v_y & v_z \end{vmatrix} = 0 \tag{3.4}$$

また、次のスキャン方向の判別式 D の符号により、仮想スクリーン上でのスキャン方向を決めることにより、ボリュームスライスに上で常に一定の方向に視線を入射することができる（第 3.4.2 節参照）。

$$D = \frac{(n-e_z)}{w^2}\left(\begin{vmatrix} h_x & h_z \\ b_x & b_z \end{vmatrix} + k_z \begin{vmatrix} v_x & v_z \\ b_x & b_z \end{vmatrix}\right)$$

ただし $w = b_z + p(h_z + k_z v_z) + q v_z$ である。

3.3 ボクセルデータへのアクセス手法

視線入射順制御を行うことにより、ボリュームスライスに対するボクセルデータのサンプリング方向を x 軸正方向に限定することができる。本章では、この連続性を利用しプロセッサとボリュームメモリとの間に、ボクセルデータアクセス機構を付加することにより、構造化されたボリュームスライスから、ボクセルデータを低レイテンシでサンプリングする手法を提案する。

3.3.1 2 次元配列構造

2 次元配列構造では、ボリュームスライス上の座標とメモリ上のボリュームデータとの対応が単純であり、計算によりアドレスを求めることができる。ボリュームスライスの大きさを 2^n とし、視線がボリュームスライスを通過する座

標を (x, y) [1]とするとき、ボクセルデータを取得するには次式が示すアドレスにあるメモリ上のデータを読むだけでよい。ただし、$SizeofVoxel$ は 1 つのボクセルデータを格納するために必要なデータサイズである。

$$(y \times 2^n + x) \times SizeofVoxel \tag{3.5}$$

この手順は次のように書き表せる。ただし、$\mathrm{ReadMemory}(adr)$ はメモリ上のアドレス adr に格納されている値を返す関数である。

Algorithm 4 2 次元配列からのボクセルデータ取得法

function $\mathrm{GetVoxelValuefrom2DArray}(x, y)$
{ x, y : サンプリングを行うボクセルの座標 }
begin
　　return $\mathrm{ReadMemory}((y \times 2^n + x) \times SizeofVoxel\ )$;
end

3.3.2　ランレングス構造

ボリュームスライスの大きさを 2^n とし、サンプリングする座標を $(x,\ y)$ とするとき、各スキャンラインのランレングス構造の先頭アドレスを取得し、先頭のランからラン長 L_i を順次加算することで、$X_{i-1} \leq x < X_i \quad (0 < i < n)$ を満たすラン i を求める。ただし、$X_{-1} = 0$ と定義する。このとき、V_i が求めるべきボクセルデータである。

この手順をまとめると次のようになる。ここで $\mathrm{GetRunLength}(a, b)$ はボクセルスキャンライン a の b 番目のランのラン長を取得する関数である。また、$\mathrm{GetVoxelValue}(a,\ b)$ はボクセルスキャンライン a の b 番目のランのボクセルデータを取得する関数である。

Algorithm 5 ランレングス符合からのボクセルデータ取得法 (1)

function $\mathrm{GetVoxelValuefromRunlengthData1}(x, y)$
{ x, y : サンプリングを行うボクセルの座標 }
begin
　　$X_{max} = \mathrm{GetRunLength}(y, 0)$;
　　$i = 0$;

[1] ここでは、座標の小数部分は切り捨てる。

while $x < X_{max}$ **do**
  $i = i + 1;$
  $X_{max} = X_{max} +$ GetRunLength(y, i);
end
return GetVoxelValue(y, i);
end

この手順でボクセルデータを得る場合、サンプリングを行う x 座標が大きくなればなるほど、多くのメモリアクセスを必要とし実用的ではない。

しかし、第 3.2節の視線入射順の制御により、ボリュームスライスへのアクセスは x 軸方向に対して常に増加する方向に限定することが可能である。従って、直前のサンプリング時に求めたランの絶対座標を記憶しておき、新たにサンプリングするデータの絶対座標と比較することにより、ボクセルデータ読み出しに要するメモリアクセス回数の削減が期待できる。

直前にサンプリングされたボクセルデータを持つランの位置を示す X_{pos} と、そのランが占める領域を表す x 座標の上限値 X_{max} をそれぞれ記憶しておき、新たにサンプリングするボクセルの座標を (X, Y) とするとき、次のようにボクセルデータの取得を行う。

- $X \leq X_{max}$ のとき
  X_{pos} のランが持つボクセルデータが求めるボクセルデータである。
- $X > X_{max}$ のとき
  X_{pos} を次のランの位置に移動後、新たなランをメモリから読み出し、X_{max} に新たに読み出したラン長を加え、$X \leq X_{max}$ となるまで上記の処理を繰り返し行う。

このアルゴリズムは次のように実現される。

Algorithm 6 ランレングス符合からのボクセルデータ取得法 (2)

function GetVoxelValuefromRunlength2(x, y)
{ x, y : サンプリングを行うボクセルの座標 }
begin
  while $x < X_{max}$ **do**
    $X_{max} = X_{max} +$ GetRunLength(y, X_{pos});
    $X_{pos} = X_{pos} + 1;$
  end

return GetVoxelValue(y, $X_{pos} - 1$);
end

このアルゴリズムをハードウェアで実装する場合、 X_{pos} , X_{max} の管理領域が 2^n 個必要となることが問題となるが、視線の入射順序を制御することにより、１つに抑えることが可能である。

3.3.3 quad-tree 構造

ボリュームスライスの一辺の大きさを 2^n とする。サンプリングする座標を (x, y) とするとき、ボクセルデータは次の手順で得ることができる [8]。なお、ここで K_i は K の i bit 目の値を意味する。

Algorithm 7 quad-tree からのボクセルデータ取得法

function GetVoxelValuefromQuadTree1(x, y)
{ x, y : サンプリングを行うボクセルの座標 }
begin
    ptr = 探索対象となる quad-tree の根節点へのポインタ;
    $i = n - 1$;
    while $i \geq 0$ **do**
        ptr = ReadMemory($ptr + 2 \times y_i + x_i$);
        if ptr がボクセルデータ **then return** ptr;
        if ptr が子節点へのリンク **then** $i = i - 1$;
    end
end

この手順では、常に quad-tree の根節点からボクセルデータの探索を行なっている。従って、１つのボクセルデータをサンプリングするたびに、そのボクセルデータが属する節点の高さに比例する回数のメモリアクセスが必要となり、ボクセルデータ取得処理の速度向上を妨げている。

しかしながら、前述のとおりボリュームデータに対するアクセスの空間的な連続性を考慮し、直前のサンプリングの際に各レベルで訪れた節点のポインタから、新たにサンプルする座標のアドレスを求めることによりメモリアクセスの回数を減らすことができると思われる。よって、本稿では次の手法によるボクセルデータ取得を行う。

ここで、直前にサンプリングしたボクセルのレベルを L、座標を (x, y)、新たにサンプリングするボクセルの座標を (X, Y)、ボリュームスライスの大きさ

を 2^n とすると、 X と x について、(n-1) bit から L bit 比較を行ない、ともに k bit$(0 \leq k \leq L)$ 連続して一致した場合、次のことが考えられる。

- $(k = L)$ の場合
  前回サンプリングした点と、新たにサンプリングする点は、同じボクセルデータを持つので、新たにメモリからボクセルデータを読み出す必要はない。
- $(k < L)$ の場合
  前回の探索時に保存していたレベル $k+1$ の節点へのポインタから quad-tree を探索することで、新しいサンプリングポイントのボクセルデータを得ることができる。

この手順は次のように書き表せる。

Algorithm 8 quad-tree からのボクセルデータ取得法 (2)

function GetVoxelValuefromQuadTree2(L, x, y, X, Y)
{ L : 直前にサンプリングしたボクセルのレベル }
{ x, y : 直前にサンプリングしたボクセルの座標 }
{ X, Y : 新たにサンプリングするボクセルの座標 }
{ ※ここで K_i は K の i bit 目の値を意味する。 }
begin
　$k = 0$;
　while $k < L$ **and** $x_{n-k-1} = X_{n-k-1}$ **do**
　　$k = k + 1$;
　{ 同一ボクセルに対するアクセスなら、前回と同じボクセルデータを返す。 }
　if $k = L$ **then return** $PreviousPointer[k+1]$;
　$ptr = PreviousPointer[k+1]$;
　while true do begin
　　$ptr =$ ReadMemory$(ptr + 2 \times y_{n-k-1} + x_{n-k-1})$;
　　$k = k + 1$;
　　$PreviousPointer[k+1] = ptr$;
　　if ptr がボクセルデータ **then return** ptr;
　end
end

quad-tree からのボクセルデータ取得例として、直前のサンプリング座標 E を $x = 1(001_2)$, $y = 2(010_2)$、新たなサンプリング座標 F を $X = 1(010_2)$, $Y =$

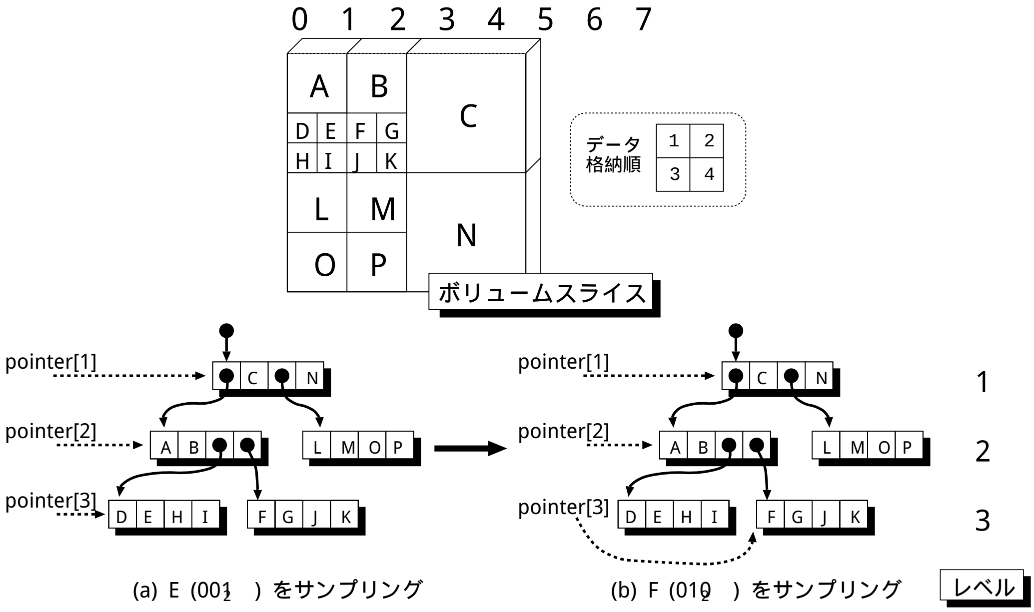


図 3.9: quad-tree 構造からのサンプリング

$2(010_2)$ の場合を考える (図 3.9)。直前のサンプリングにおいて、各レベルのポインタは図 3.9(a) のように設定される。ここでアルゴリズム 8を適用すると以下のように新たなボクセルデータを得ることができる。

1. 直前のサンプリング座標と新たなサンプリング座標を比較する。
   この結果、 bit1 で $x_1 \neq X_1$ [1]となり相違が生じる。
2. 直前のサンプリングで得たレベル 2 のポインタ $ptr[2]$ (図 3.9(a)) を取得する。
   $ptr = PreviousPointer[2]$
3. 以下同様に、ボクセルデータが取得できるまで繰り返す。
   $ptr = PreviousPointer[3] = ReadMemory(ptr + 2 \times y_1 + x_1);$
4. $ptr = ReadMemory(ptr + 2 \times y_0 + x_0);$
5. 終了
   この時点で、新たなサンプリングの結果、取得したボクセルデータは ptr となる。また、各レベルのポインタは図 3.9(b) のように付け替えられている。

[1] K_i は K の i bit 目の値を意味する。

3.4 視線入射順制御の証明

3.4.1 平行投影法

スクリーン上の座標が $(x,\ y)$ の点を示すベクトル**R**は次式で表される。

$$\mathbf{R} = \mathbf{S}_b + x\mathbf{S}_h + y\mathbf{S}_v$$

この点**R**をボリューム空間中の平面 $z = n$ へ平行投影した点を示すベクトル**R'**は次式で表される。

$$\mathbf{R'} = \mathbf{R} + s(x,\ y,\ n)\mathbf{S}_n$$

ここで $s(\alpha,\ \beta,\ \gamma)$ は、

$$s(\alpha,\ \beta,\ \gamma) = \frac{\gamma - (b_z + \alpha h_z + \beta v_z)}{n_z}$$

従って、仮想スクリーン上の点 $(p,\ q)$ は写像 f より次の座標を持つ。

$$\mathbf{R'} \quad = \quad \mathbf{S}_b + p\mathbf{S}_h + (q + pk_z)\mathbf{S}_v + s(p,\ q + pk_z,\ n)\mathbf{S}_n$$

R'を p で微分すると以下のようになる。

$$\frac{d\boldsymbol{R'}}{dp} \quad = \quad \mathbf{S}_h + k_z\mathbf{S}_v - \frac{h_z + k_z v_z}{n_z}\mathbf{S}_n$$

ここで q を定数としたとき、ベクトル $d\boldsymbol{R'}/dp$ の y 成分 $d\boldsymbol{R}_y'/dp$ が 0 なら、この写像はボクセル平面 $z = n$ 上で x 軸に平行な軌跡を描く。

$$\begin{aligned}
\frac{d\boldsymbol{R}_y'}{dp} &= h_y + k_z v_y - \frac{n_y}{n_z}(h_z + k_z v_z) \\
&= \frac{1}{n_z}\left(\begin{vmatrix} h_y & h_z \\ n_y & n_z \end{vmatrix} + k_z \begin{vmatrix} v_y & v_z \\ n_y & n_z \end{vmatrix} \right) \\
&= \frac{1}{n_z}\left(\begin{vmatrix} h_y & h_z \\ n_y & n_z \end{vmatrix} - \begin{vmatrix} h_y & h_z \\ n_y & n_z \end{vmatrix} \right) \\
&= 0
\end{aligned}$$

よって、仮想スクリーン上の任意のスキャンライン q 上の点は、任意のボリュームスライス n 上に x 軸に平行な軌跡を描くことが証明された。

一方、サンプリング方向は $d\boldsymbol{R}_x$'$/dp$ の符号により決まる。

$$\begin{aligned}\frac{d\boldsymbol{R}_x\text{'}}{dp} &= h_x + k_z v_x - \frac{n_x}{n_z}(h_z + k_z v_z) \\ &= \frac{1}{n_z}\left(\begin{vmatrix} h_x & h_z \\ n_x & n_z \end{vmatrix} + k_z \begin{vmatrix} v_x & v_z \\ n_x & n_z \end{vmatrix} \right)\end{aligned}$$

3.4.2 透視投影法

スクリーン上の座標が $(x,\, y)$ の点を示すベクトル$\mathbf{R}$は次式で表される。

$$\mathbf{R} = \mathbf{E} + \mathbf{S}_b + x\mathbf{S}_h + y\mathbf{S}_v$$

この点$\mathbf{R}$を、ボリューム空間中のスライス平面 $z = n$ へ透視投影した点を示すベクトル$\mathbf{R}$'は次式で表される。

$$\mathbf{R}\text{'} = \mathbf{E} + (\mathbf{S}_b + x\mathbf{S}_h + y\mathbf{S}_v)s(x,\, y,\, n)$$

ここで $s(\alpha,\, \beta,\, \gamma)$ は、

$$s(\alpha,\, \beta,\, \gamma) = \frac{\gamma - e_z}{b_z + \alpha h_z + \beta v_z}$$

従って、仮想スクリーン上の点 $(p,\, q)$ は写像 f より次の座標を持つ。

$$\mathbf{R}\text{'} = \mathbf{E} + s(p,\, q + pk_z,\, n)\{\mathbf{S}_b + p\mathbf{S}_h + (q + pk_z)\mathbf{S}_v\}$$

$\mathbf{R}$'を p で微分すると以下のようになる。

$$\frac{d\boldsymbol{R}\text{'}}{dp} = \frac{(n - e_z)}{w^2}\left\{ q(v_z\mathbf{S}_h - h_z\mathbf{S}_v) + k_z(b_z\mathbf{S}_v - v_z\mathbf{S}_b) + (b_z\mathbf{S}_h - h_z\mathbf{S}_b) \right\}$$

ただし、 $w = b_z + p(h_z + k_z v_z) + qv_z$ である。

ここで q を定数としたとき、ベクトル $d\boldsymbol{R}$'$/dp$ の y 成分 $d\boldsymbol{R}_y$'$/dp$ が 0 ならこの写像はボクセル平面 $z = n$ 上で x 軸に平行な軌跡を描く。

$$\frac{d\boldsymbol{R}_y\text{'}}{dp} = \frac{(n - e_z)}{w^2}\left(q\begin{vmatrix} h_y & h_z \\ v_y & v_z \end{vmatrix} + k_z\begin{vmatrix} v_y & v_z \\ b_y & b_z \end{vmatrix} + \begin{vmatrix} h_y & h_z \\ b_y & b_z \end{vmatrix} \right)$$

ここで k_z に式 3.3を代入し、式 3.4で示される制約を考慮すると、

$$
\begin{aligned}
\frac{d\boldsymbol{R}_y\text{'}}{dp} &= \frac{(n-e_z)}{w^2}\left(\begin{vmatrix} h_y & h_z \\ b_y & b_z \end{vmatrix} - \begin{vmatrix} h_y & h_z \\ b_y & b_z \end{vmatrix}\right) \\
&= 0
\end{aligned}
$$

この写像はボクセル平面 $z=n$ 上で x 軸に平行な直線を描く。従って、仮想スクリーンの任意のスキャンライン q 上の点は、任意のボリュームスライス n 上に x 軸に平行な軌跡を描くことが証明された。

また、サンプリング方向は $d\boldsymbol{R}_x\text{'}/dp$ の符号により決まる。

$$
\begin{aligned}
\frac{d\boldsymbol{R}_x\text{'}}{dp} &= \frac{(n-e_z)}{w^2}\left(q\begin{vmatrix} h_x & h_z \\ v_x & v_z \end{vmatrix} + k_z\begin{vmatrix} v_x & v_z \\ b_x & b_z \end{vmatrix} + \begin{vmatrix} h_x & h_z \\ b_x & b_z \end{vmatrix}\right) \\
&= \frac{(n-e_z)}{w^2}\left(k_z\begin{vmatrix} v_x & v_z \\ b_x & b_z \end{vmatrix} + \begin{vmatrix} h_x & h_z \\ b_x & b_z \end{vmatrix}\right)
\end{aligned}
$$

4　ボリュームスライス構造の評価

本章では、第 2.5.3章で述べたボリュームスライスを表現する 2 次元配列構造、ランレングス構造、 **quad-tree** 構造の 3 つのデータ構造について、ボリュームデータを格納するために必要な記憶容量、可視化に要する処理時間という観点から、その理論的な限界を示し、実際の医療画像を用いた評価を行うことによりその有効性を確かめる。

4.1　データ格納容量

一辺が 2^n のボリューム空間の可視化を行ない、主軸方向にボリューム空間をスライスに分割した場合の 1 スライスの格納に要する容量を比較する。 1 ボクセルを格納するために必要な容量を $SizeofVoxel$、ポインタを格納するためのサイズを $SizeofPointer$、ラン長を格納するためのサイズを $SizeofRunLength$ とする。

4.1.1　2 次元配列構造

2 次元配列では、 1 スライス分のデータを格納するために必要なデータサイズは次式のとおりである。

$$2^{2n} \times SizeofVoxel \tag{4.1}$$

4.1.2 ランレングス構造

- データサイズ

  ランレングス構造では、各スキャンラインのランレングス情報を保持する領域と、その領域へのポインタを格納する配列が、ボリュームスライスを表現するために必要な容量を決める (図 2.6)。

  配列を格納するために必要な容量は、ボクセルスキャンライン数が 2^n なので $2^n \times SizeofPointer$ となる。一方、ランレングス情報を格納するために必要な容量は、平均ラン長 L を用いて $2^n/L \times (SizeofVoxel + SizeofRunLength)$ となる。

  従って、次の記憶容量が必要となる。

$$2^n \times \{SizeofPointer + 2^n/L \times (SizeofVoxel + SizeofRunLength)\} \tag{4.2}$$

- 圧縮率

  2 次元配列構造とランレングス構造に必要なメモリ容量を比較すると次のようになる。式 (4.1), (4.2) より、ランレングス構造を用いて 2 次元配列構造を圧縮して格納した場合の圧縮率[1]は、次式で示される。

$$2^{-n} \cdot \frac{SizeofPointer}{SizeofVoxel} + \frac{1}{L} \cdot (1 + \frac{SizeofRunLength}{SizeofVoxel}) \tag{4.3}$$

  ここで式 (4.3) が 1 以下のとき、ランレングス構造で表現されたボリュームスライスは 2 次元配列構造のものより小さい容量で格納することができる。このような L を求めると

$$L > \frac{SizeofRunLength + SizeofVoxel}{SizeofVoxel - 2^{-n} \cdot SizeofPointer}$$

  n が大きいとき

$$\begin{aligned} L &> \frac{SizeofRunLength + SizeofVoxel}{SizeofVoxel} \\ &= 1 + \frac{SizeofRunLength}{SizeofVoxel} \end{aligned}$$

[1] 圧縮率 $= \frac{\text{圧縮後のデータサイズ}}{\text{圧縮前のデータサイズ}}$

ここで $SizeofRunLength$ と $SizeofVoxel$ を $1byte$ と定義すると、 n が十分大きいとき、平均ラン長が 2 以上のときにランレングス構造によるボリュームスライス圧縮は有効な圧縮法となることが示された。これは医療画像のような空間周波数が低い箇所が多いデータでは有効であることを示している。

- 最悪の場合のメモリ容量
  ランレングス構造によりボリュームスライスを格納する場合、最悪の場合は平均ラン長が 1 のときである。このとき、必要となるデータサイズは次のとおりである。

$$2^n \cdot SizeofPointer + 2^{2n}(SizeofVoxel + SizeofRunLength)$$

これは、 2 次元配列構造の $2^{-n} \cdot SizeofPointer + \dfrac{SizeofRunLength}{SizeofVoxel}$ 倍でありn が十分大きいとき、 2 次元配列構造の $\dfrac{SizeofRunLength}{SizeofVoxel}$ 倍のメモリ容量が必要となる。

4.1.3 quad-tree 構造

- データサイズ
  quad-tree 構造では、ボリュームスライスを表現するために必要な節点数が格納するためのデータサイズを決める。ここで、レベル L $(1 \leq L < N)$ のボクセルがボリュームスライスの R_L $\quad (\sum_{i=1}^{n} R_i = 1)$ を占めるとき、レベル L のボクセル数 V_L は、

$$V_L = 2^{2L} \times R_L \qquad (1 \leq L < n) \tag{4.4}$$

となる。従って、レベル L のスライスを格納するために必要な節点数 N_L は、下位レベルへのリンクを考慮すると次式で表される。

$$N_L \;=\; \sum_{i=0}^{n-L} \frac{V_{n-i}}{4^{n-L-i+1}}$$

V_i を代入して

$$N_L \;=\; \sum_{i=0}^{n-L} 4^{L-1} \times R_{n-i}$$

よって、合計節点数 S は、

$$\begin{aligned} S &= \sum_{j=1}^{n} N_j \\ &= \sum_{j=1}^{n}\sum_{i=0}^{n-j} 4^{j-1} \times R_{n-i} \end{aligned}$$

となる。

1 節点には 4 つのポインタを格納する必要があるため、 quad-tree を格納するために必要なデータサイズは、次のとおりである。

$$SizeofPointer \times \sum_{j=0}^{n-1}\sum_{i=0}^{j} 4^{j} \times R_i \tag{4.5}$$

- 圧縮率

 2 次元配列構造および quad-tree 構造で表現されたボリュームスライスを格納するために必要なメモリ容量を比較すると次のようになる。式 (4.1), (4.5) より、 quad-tree 構造による圧縮率 r は、レベル L のボクセルが占める面積比 R_L $(\sum_{i=1}^{n} R_i = 1)$ を用いて次式で表される。

$$r = \frac{SizeofPointer}{SizeofVoxel} \times \sum_{j=1}^{n} 4^{j-n} \sum_{i=0}^{n-j} R_{n-i}$$

ここで $\sum_{i=1}^{j} R_n \leq 1$ なので

$$\begin{aligned} r &\leq \frac{SizeofPointer}{SizeofVoxel} \times \sum_{j=1}^{n} 4^{j-n} \\ &= \frac{4}{3} \times \frac{SizeofPointer}{SizeofVoxel} \times (1 - 4^{-n-1}) \end{aligned}$$

ここで n が十分大きいとき

$$= \frac{4}{3} \times \frac{SizeofPointer}{SizeofVoxel}$$

これは即ち、情報量を保存しながら quad-tree により構造化した場合には、その圧縮率が $\frac{4 \times SizeofPointer}{3 \times SizeofVoxel}$ を越えることはないことを示す。

表 4.1: ボリュームスライス格納に要したデータサイズ

ボリュームデータの密度	データ構造	要素数 (個)	要素あたりのデータサイズ ($byte$/ 個)	管理領域サイズ ($byte$)	合計データサイズ ($byte$)
(a) 高密度	2 次元配列構造	65536	1	2	65538
	ランレングス構造	19201	2	514	38916
	quad-tree 構造	8382	8 *	2	43228
(b) 中密度	2 次元配列構造	65536	1	2	65538
	ランレングス構造	11725	2	514	23964
	quad-tree 構造	4956	8 *	2	25392
(c) 低密度	2 次元配列構造	65536	1	2	65538
	ランレングス構造	2937	2	514	6388
	quad-tree 構造	1162	8 *	2	6056

* 最密格子サイズのボクセルデータを格納するためのデータサイズは $4byte$ で算出。

- 最悪の場合のメモリ容量

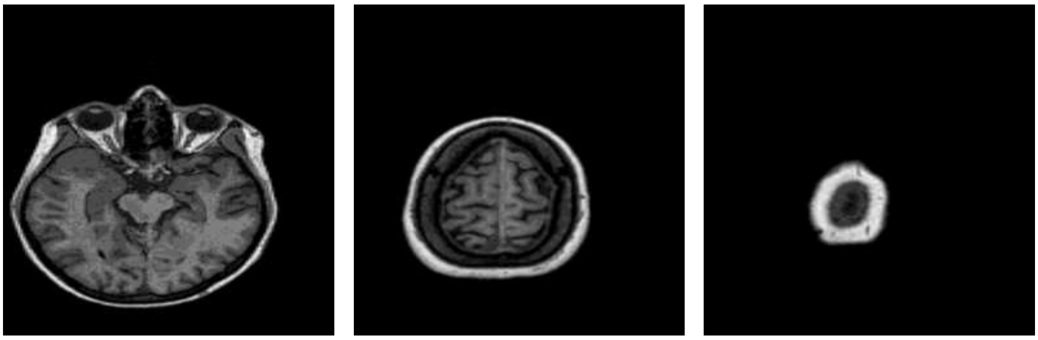
  quad-tree 構造によりボリュームスライスを格納するとき、格納に必要なメモリ容量が最大になるのは、ボリュームスライスが全て最下層のボクセルにより占められるときであり、式 (4.5) において $R_n = 1, R_i = 0(i \in \{1, 2, \ldots, n-1\})$ のときである。従って、 quad-tree を格納するために必要なメモリ容量は、次のとおりである。

$$SizeofPointer \times \sum_{j=1}^{n} 4^{j-1} \times R_n = \frac{1}{3} \cdot (4^n - 1) \cdot SizeofPointer$$

4.1.4 医療画像によるボリュームデータサイズの評価

2 次元配列構造、ランレングス構造、 **quad-tree** 構造の 3 つのデータ構造について、ボリュームスライスを表現するために必要なデータサイズを知るため、人間の頭部のボリュームデータ[1)]を用いた評価を行う。評価には、ボリュームデータに占める物体の割合が異なる 3 つのボリュームスライスを用いた (図 4.1)。

1) 256^3 ボクセル, 1 ボクセルあたり 8bit



(a) 高密度　　(b) 中密度　　(c) 低密度

図 4.1: ボリュームサイズの評価に用いたボリュームスライス

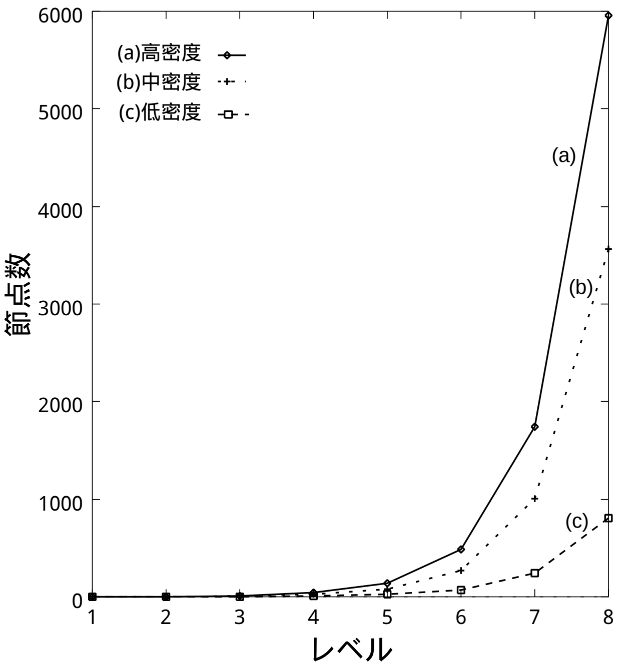


図 4.2: quad-tree レベル別節点数

データ特性　ボリュームデータについて quad-tree を構成し、レベル別の節点数 (図 4.2) と、レベル別の面積比 (図 4.3) を求めた。

レベル別の面積比において、低レベル層 (Level 2) と高レベル層 (Level 8) にピークが見受けられる。高いレベル層のピークは、ボリュームスライスの空間周波数が高い部分を密度の高い格子でサンプリングしていることを示す。低いレベル層のピークは物体周辺の空白部分が少ない節点で効率良く格納されていることを示している。

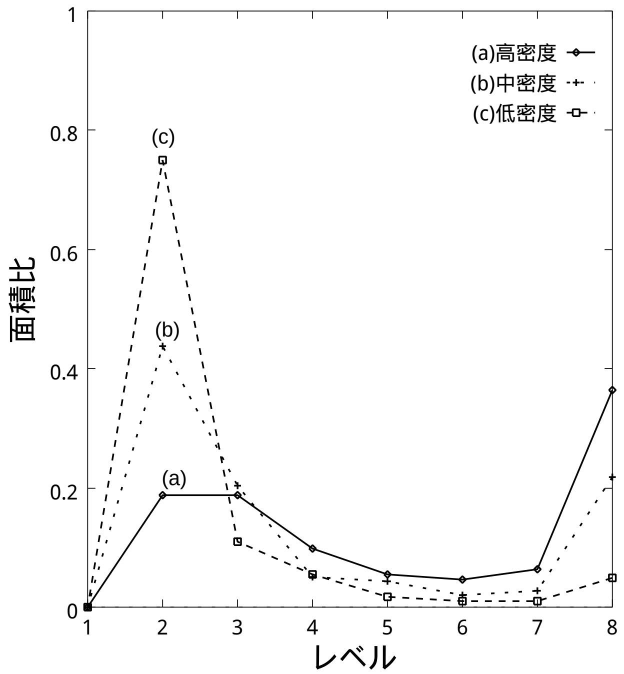


図 4.3: quad-tree レベル別面積比

格納容量　図 4.1に示すボリュームスライスを、各構造で表現するために要したデータサイズを表 4.2 [1)]に示す。

ランレングス構造による圧縮の結果、 2 次元配列構造に対して密度により 9.7% ~ 59.4% のデータサイズで格納された。一方、 quad-tree 構造の要素あたりのデータサイズは 8byte だが図 4.2に示すように quad-tree 構造の節点の大部分は最大レベルのボクセルデータであり、それらはもはや分割する必要がないので、ポインタを格納するための領域は必要なく、要素あたりのサイズを $4byte$ にすることができる。この点を考慮すると quad-tree 構造を格納するために必要なデータサイズは、それぞれ $43228, 25392, 6056byte$ となり 2 次元配列構造に対して 9.2% ~ 66.0% のデータサイズで格納された。

4.1.5　劣化が許される環境におけるボリュームデータ容量比較

3 次元計測機器から出力されるボリュームデータのあるボクセルには、その隣接ボクセルとの強い相関がある。そこで、ランレングス構造または quad-tree 構造で許容された範囲内の自乗誤差を持つ近傍ボクセルを一つにまとめて格納す

[1)] 表中の要素あたりのデータサイズは、ボクセルデータ格納サイズ、ラン長格納サイズ、ポインタ格納サイズをそれぞれ $1, 1, 2\ byte$ として算出した。

表 4.2: ボリュームスライスの SN 比とデータ容量

ボリュームスライスデータ構造	SN 比 (dB)	要素数 (個)	要素あたりのデータサイズ ($byte$/ 個)	管理領域サイズ ($byte$)	合計データサイズ ($byte$)
2 次元配列構造	58dB	65536	1	2	65538
ランレングス構造	58dB	19201	2	514	38916
	46dB	19160	2	514	38834
	34dB	8712	2	514	17938
quad-tree 構造	58dB	8382	8 *	2	43228
	46dB	7240	8 *	2	38400
	34dB	1931	8 *	2	11808

* 最密格子サイズのボクセルデータを格納するためのデータサイズは $4byte$ で算出。

ることにより、より大きいボリュームスライスを格納することが可能である。

また、逆にボリュームスライスを格納するのに十分なメモリを持たない環境においても、ボリュームスライスを劣化させることにより全体像を可視化することができる。

画質とボリュームデータ量の関係を表 4.2に示す。また、各 SN 比における画像の様子を図 4.4, 4.5 に示す。

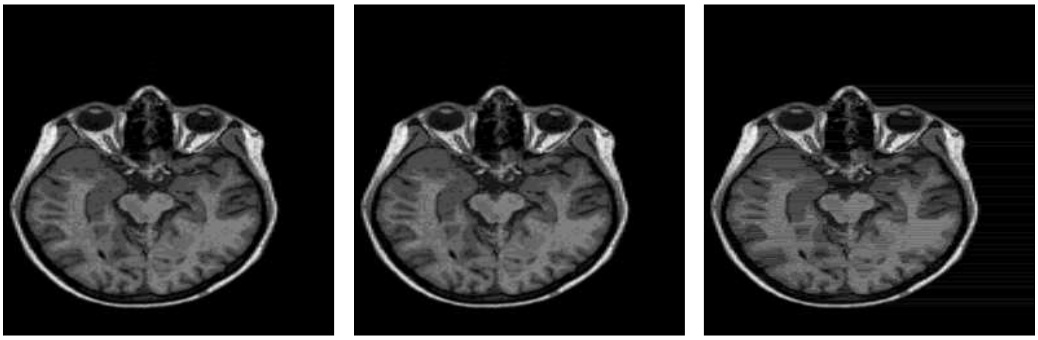
この結果、データサイズを大幅に縮めるためには、人間により視覚できる程のボリュームスライスの劣化を許さなければならないことが明らとなった。しかしながら、大規模なボリュームデータの全体像を把握したい場合などには細部の画像の劣化は許容されるのでこの手法は有用である。

4.2 メモリアクセス回数

ボクセルデータを取得するために必要なメモリアクセスの回数はハードウェアで可視化を行う際に大きな影響を与える。ここでは、ボクセルデータをサンプリングするために必要なメモリへのアクセス回数について述べる。

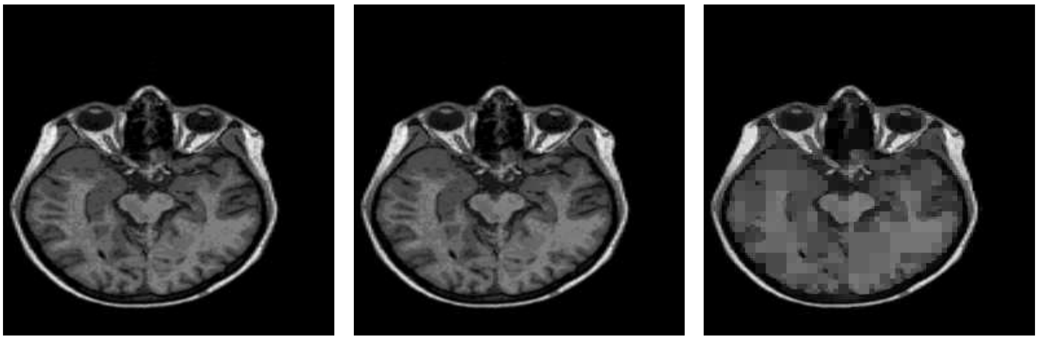
4.2.1 2 次元配列構造

ボクセルデータを取得するためにメモリに対してアクセスしなければならない回数は 1 回だけである。



(a) SN 比: 58dB　(b) SN 比: 46dB　(c) SN 比: 34dB

図 4.4: ランレングス構造で保持されたボリュームスライス



(a) SN 比: 58dB　(b) SN 比 : 46dB　(c) SN 比 : 34dB

図 4.5: quad-tree 構造で保持されたボリュームスライス

4.2.2 ランレングス構造

各スキャンラインで最初にボクセルデータを取得する際に、 x 座標値に比例した回数のメモリアクセスが必要である。

2 度目以降は、 1 度目のサンプリング時に得た情報を基に相対的にボクセルデータが検索される。このとき、ボリュームメモリへのアクセスは、視線の入射間隔を ΔR、最密格子サイズを Δg、 平均ラン長を L とすると 1 度のサンプリングで平均 $\dfrac{\Delta R/\Delta g}{L}$ 回のメモリアクセスが必要となる。

視線の入射間隔は、ボリュームスライスに入射する視線の間隔をボクセル単位で表したものであり、平行投影法の場合はボリュームデータの拡大して観測するとき、透視投影法ではボリュームデータに接近することにより小さくなる (図 4.6)。

- 視線の入射間隔 $\Delta R \leq$ 最密格子サイズ Δg のとき

  視点がボリューム空間に近付いているとき、即ちボリュームを拡大して観

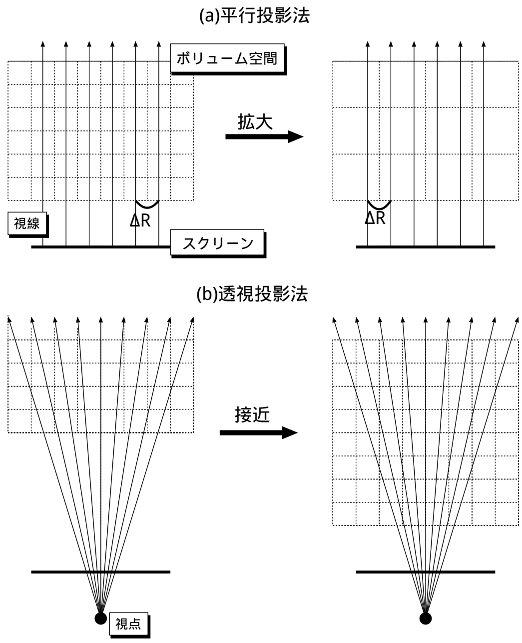


図 4.6: 視線の入射間隔

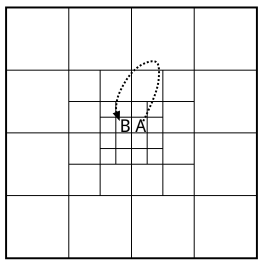


図 4.7: ボクセルデータ取得の最悪な場合

測しているときや、ピクセルの解像度がボクセルの解像度に比べて高いときに生じる。平行投影の場合は、ボクセルのサンプリングは $\Delta g/\Delta R$ 回に１度メモリへのアクセスを必要とする。

- 視線の入射間隔 $\Delta R >$ 最密格子サイズ Δg のとき
  ボリューム空間を遠方から眺めているときや、ボクセルの解像度がピクセルに対して高いときに生じる。１度のサンプリングでランを平均 $\Delta R/\Delta g$ 個のボクセルを読み飛ばす必要がある。従って、平均ラン長を L とするとき、 $\dfrac{\Delta R/\Delta g}{L}$ 回のメモリアクセスが必要となる。

4.2.3 quad-tree 構造

本手法でボクセルデータを取得する場合、次のメモリアクセス回数で、ボクセルデータを取得することができる。

$$max(\ (\text{データのレベル} - \textbf{連続した一致 bit 数}), 0)\ \text{回}$$

ここでデータのレベルは、新たにサンプリングするデータが格納されている quad-tree 上でのレベルである。

アクセス回数の削減　ボリュームスライスの大きさが 2^n であるとき、アルゴリズム 7でボクセルデータを取得する場合は、次の回数のメモリアクセスが必要である。

$$(n- \text{データのレベル})\ \text{回}$$

一方、アルゴリズム 8でボクセルデータを取得することにより、次の回数のメモリアクセスによりボクセルデータを取得することができる。

$$(n- \text{データのレベル} - \text{一致 bit 数})\ \text{回}$$

ここでデータのレベルは、新たにサンプリングしようとしているデータが格納されている節点のレベルである。また、一致 bit 数は前回のサンプリング座標と、新たなサンプリング点の座標とで一致している bit 数を最上位 bit から数えあげたものである。アルゴリズム 8において、ボリュームスライスの中央に最密格子サイズが存在するときには、アルゴリズム 7と同じ

(n - データのレベル) 回

のメモリアクセスが必要となる。図 4.7の点 A から点 B に移動する場合がその例である。しかしながら、平均的な医療画像において画面全体を描画する際の平均アクセス数を見ると図 4.8のように対数的な増加をするので大きな問題とはならない。

高速化手法　quad-tree 構造の場合は、次に示す手法をとることにより、さらにボクセルデータを高速に取得することができる。

- 可視化階層の制限

  quad-tree の各節点に、その子節点が持つボクセルデータの代表値を保持することにより、 quad-tree 構造の段階的な可視化が可能となる。従って、quad-tree をたどる木のレベルを制限することで、ボリュームメモリへのアクセス回数を削減することが可能である。しかし、この場合 図 4.7の中心部などメモリアクセスが多発する箇所で像がぼやけてしまうという問題が生じる。

  しかしながら、実際に可視化が行われる環境においては、像の概要を見ながら大まかに注目する位置を決定した後、詳細情報の可視化を行う場合が多い。従って、可視化階層の制限は可視化装置の運用実態から見れば妥当

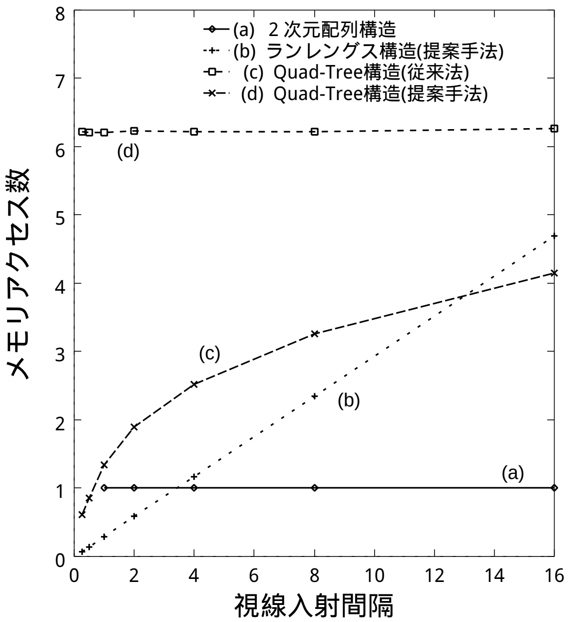


図 4.8: 平均メモリアクセス回数

な制限である。

- スクリーン分割
  スクリーンを幾つかの正方領域に分割し、各分割に対して quad-tree を割り当てることにより木の高さを削減することができる。
- テーブルの利用
  2^i 個の要素を持つテーブルを用意することにより、quad-tree のレベル $(n-i)$ からの探索が可能である。しかしながら、i が大きくなるとこの手法は現実的ではなくなる。

4.2.4 メモリアクセス回数の評価

図 4.1(a) のボリュームスライス上の各ボクセルに対し、x 軸方向に順次データをサンプリングした場合に要するメモリアクセス回数の測定を行った結果を図 4.8に示す。この図は、視線の入射間隔によるサンプリングに必要な平均メモリアクセスの回数の変化を示している。

視線の入射間隔は平行投影法においては、ボクセルの格子間隔 ΔV とピクセルの格子間隔 ΔP により決まり、$\Delta V/\Delta P$ となる。また、透視投影法では視点、スクリーン、ボリューム空間の位置関係により異なる。

2 次元配列構造 (a) では、入射間隔によらず 1 度のメモリアクセスでデータを取得することができる。また、ランレングス構造 (b) では入射間隔に比例す

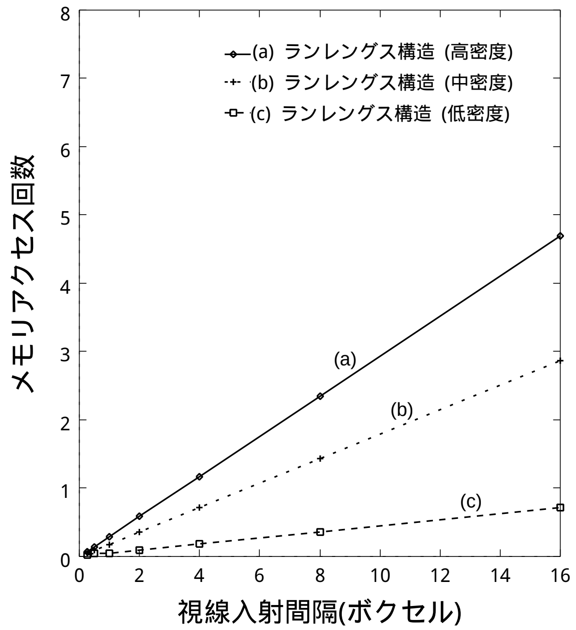


図 4.9: 密度別平均メモリアクセス回数 (ランレングス構造)

る回数でアクセスが終了する様子がわかる。

一方、アルゴリズム 7で quad-tree 構造からボクセルデータをサンプリングする場合 (c)、平均アクセス回数は約 6.2 回である。これは quad-tree の高さに依存している。また、アルゴリズム 8で quad-tree 構造からボクセルデータをサンプリングする場合 (d)、そのアクセス回数は対数関数的に増加している。このことは、アルゴリズム 8が、順次入射してくる視線の空間的な局所性を利用していることを示している。

スクリーンとボリュームデータの解像度のバランスがとれる視線間隔 1 のときの階層格子へのアクセス回数は、ランレングス構造 (図 4.8(b)) においては平均 0.3 回となり、 2 次元配列構造 (図 4.8(a)) よりも高速化された。また、 quad-tree 構造 (図 4.8(d)) の場合は平均 1.3 回となる。これは、一般的な運用状況において従来の 2 次元配列を用いた実時間可視化システムに本稿が提案する手法を用いて階層格子ボリュームデータの可視化を行った場合、十分な可視化速度を得ることができることを示している。

従って、 $\dfrac{\Delta V}{\Delta P \cdot L} > n$ が成り立つときランレングス構造による平均メモリアクセス回数は、 quad-tree 構造によるものより多くなる。即ち、ピクセルに対してボリュームデータの解像度が高いときや、物体の内部構造が複雑で平均ラン長が短くなった場合がそれにあたる。

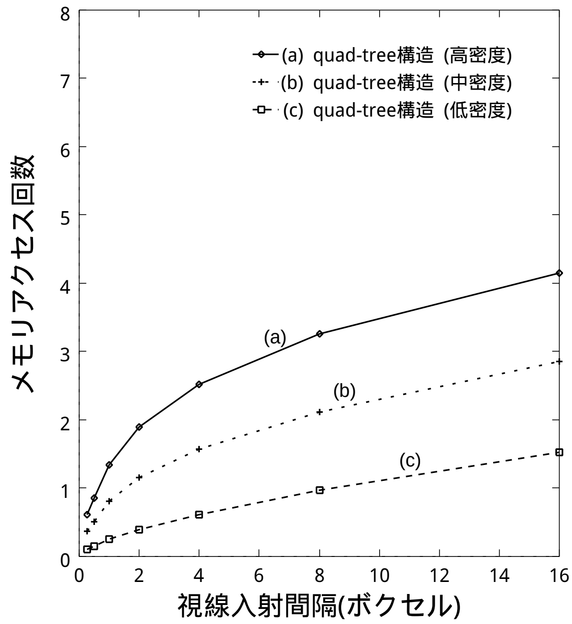


図 4.10: 密度別平均メモリアクセス回数 (quad-tree 構造)

4.2.5 補間によるメモリアクセスの増加

図 4.11のように、ボリュームを構成する単位立方体の頂点にボクセルデータが存在すると考え、単位立方体の内部の値はサンプリング点の座標を囲む 4 頂点のボクセルデータから線形補間により次のように求めることにより、図 4.12に示すボリュームスライスから図 4.13のような精細なボリュームスライスを得ることができる [1]。

ここで、サンプリング点の小数部が $(x_i\ ,\ y_i)$ であるときは、次式により色 c_i と透明度 α_i を求めることができる。

$$
\begin{array}{rcccc}
c_i & = & (1-x_i)\cdot(1-y_i)\cdot c_{(i,0)} & + & x_i\cdot(1-y_i)\cdot c_{(i,1)} \\
 & + & (1-x_i)\cdot y_i\cdot c_{(i,2)} & + & x_i\cdot y_i\cdot c_{(i,3)} \\
\alpha_i & = & (1-x_i)\cdot(1-y_i)\cdot \alpha_{(i,0)} & + & x_i\cdot(1-y_i)\cdot \alpha_{(i,1)} \\
 & + & (1-x_i)\cdot y_i\cdot \alpha_{(i,2)} & + & x_i\cdot y_i\cdot \alpha_{(i,3)}
\end{array}
$$

ボリュームスライスの補間は、ボリュームデータの解像度が低いときや、物体を拡大して観測する際には特に有効である。しかしながら、補間を行う場合のために必要なサンプリング点の数が 4 倍になるので、メモリアクセス回数が可視化処理時間に及ぼす影響が大きくなる。従って本節では、補間による各データ構造のアクセス回数の増加について述べる。

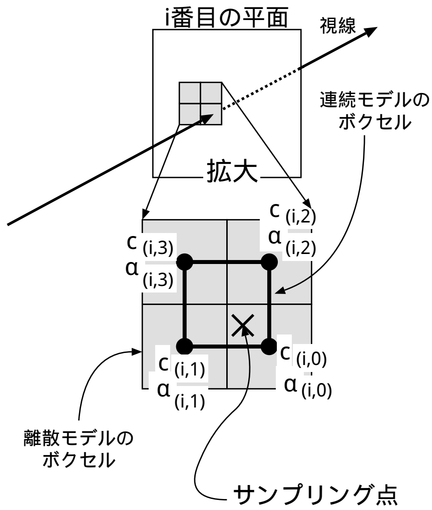


図 4.11: 連続モデルのボクセルの補間

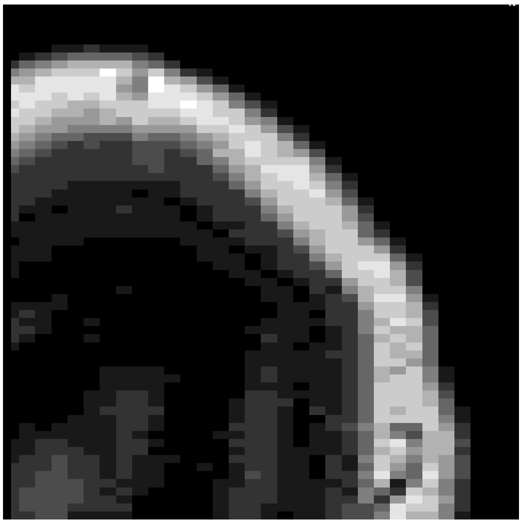


図 4.12: 補間処理を行わないボリュームスライス

図 4.13: 補間処理を施したボリュームスライス

連続モデルにおけるボクセルへのアクセス手法

- 2 次元配列構造

  連続モデルでは、サンプリング点 (x, y) を含む 4 つのボクセルデータ (x, y) , $(x-1, y)$, $(x\ , y-1)$, $(x-1, y-1)$ を読み出す必要がある。これは、ボクセルデータの読み出し処理を 4 度行うことにより実現され、離散モデルに対して約 4 倍の時間を必要とする[1]。

[1] DRAM の Fast Page Mode を用いると 3 倍程度に抑えることができる。

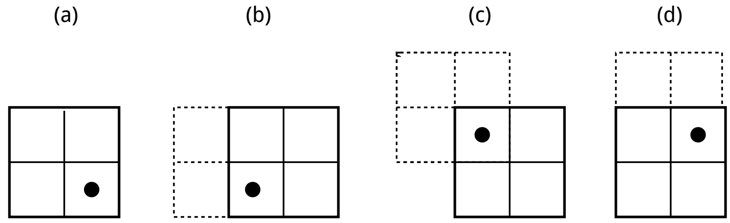


図 4.14: quad-tree 離散モデルのサンプリング点

- ランレングス構造

  連続モデルでは、サンプリング点の１つ上のスキャンラインのボクセルデータが必要となる。このため最悪の場合[1)]、メモリから４回ランレングスデータを読み出す必要がある。

- quad-tree 構造

  quad-tree の場合、アルゴリズム 8により連続モデルに使用するボクセルデータのメモリからの読み出し方法が決定される。

  (a) サンプリング点が一つのボクセルに収まるとき
  新たにメモリからボクセルデータを読み込む必要はない。

  (b) 全てのサンプリング点が同一節点に収まるとき
  ボクセルデータが存在する節点はすでに判明しているので、１度のメモリアクセスでボクセルデータを読み出すことができる。

  (c) 当該節点以外の節点にサンプリング点が存在するとき。
  この場合、次の３つのケースが考えられる。

  1. quad-tree の左下領域に視線が通過したとき
  基本位置の左の節点を読む必要がある。
  2. quad-tree の左上領域に視線が通過したとき
  基本位置の左、上、左上の節点を読む必要がある。
  3. quad-tree の右上領域に視線が通過したとき
  基本位置の上のボクセルを読む必要がある。

  このため最悪の場合[2)]、３節点分のデータを読み出す必要がある。

[1)] 当該スキャンラインと、１つ上のスキャンラインのサンプリング点 $(x,\ y)$, $(x\ ,\ y-1)$ がともにランの最初にあり、かつラン長が最小格子サイズのとき。

[2)] ケース (3) で隣接ボクセルがすべて最小格子サイズのとき

5 おわりに

本稿では、階層格子ボリュームデータの可視化手法について述べた。

階層格子ボリュームデータをボリュームスライスに分割し、各 PE に割り当てることにより、フレーム間で並列処理を行うことができる。ボリュームスライスの格納法として 2 次元配列構造、ランレングス構造、 quad-tree 構造の 3 つの構造についてボリュームデータを格納するために必要な記憶容量、可視化に要する処理時間という観点から評価を行った。

格納に要するメモリサイズは、ボリュームデータが持つ統計的な偏りと解像度によって変化する。 256^3 ボクセルの大きさの脳画像のボリュームスライスを格納するために、 2 次元配列構造では 65538byte、ランレングス構造では 38916byte、 quad-tree 構造では 43228byte が必要であった。

また quad-tree 構造では、その圧縮率が理論的に $\frac{4}{3} \cdot \frac{SizeofPointer}{SizeofVoxel}$ を越えないことを示した。ここで、 $SizeofPointer$ はポインタを格納するためのメモリサイズであり、 $SizeofVoxel$ はボクセル値を格納するためのメモリサイズを表し、現状ではそれぞれ 4byte と 1byte が一般的である。従って、そのような環境では本稿に述べているメモリ管理方式を採った quad-tree 構成では圧縮率は最悪でも 4 倍を越えないことが明らかとなった。一方、ランレングス構造では最悪な場合 $\frac{SizeofRunLength}{SizeofVoxel}$ となることが示された。

またボリュームデータを格納するメモリが十分にない環境において、ボリュームスライスの劣化を許容することにより、ランレングス構造、 quad-tree 構造では、より以上の大きさのボリュームスライスを保持することができるが、ランレングス構造の場合は特有の符号化方向に延びるノイズが目立ち、実用面を考慮すれば大幅な削減は困難であることがわかった。

一方、メモリアクセスにかかる時間は可視化に要する時間に大きな影響を与える。ボクセルスキャンライン上に順序付けられて入射する視線に対して、ランレングス構造からのボクセル読み出し時間は、ボリュームデータ上の視点の入射間隔に比例する。また、 quad-tree 構造へのアクセス回数はサンプリングしようとしているボクセルの格子の幅と、前回のサンプリングの位置とにより変化する。

従って、ランレングス構造でボリュームスライスを保持した場合、その可視化速度は視線間隔の影響を受けやすい。そのため、ボリュームデータの狭い範

囲を重点的に観測する場合には高速な可視化が行える反面、大域的な観測を行う場合には大幅な速度低下を伴う。一方、 quad-tree 構造は視線間隔による影響を受けにくく、広い範囲の観測において実用的な可視化性能を得ることができた。

謝辞

本研究の機会を与えて下さった、富田眞治教授に深く感謝の意を表します。数々の御指導、御助言を頂いた森眞一郎助教授に心から感謝いたします。

また日頃御討論いただく、吉谷直樹氏、重田大助氏をはじめ、計算機工学講座の諸氏に感謝いたします。

参考文献

[1] 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* のアーキテクチャ, 情報処理学会論文誌, Vol. 36, pp. 1709–1718 (1995).

[2] 明石英也: ボリューム・レンダリング向き並列計算機のアーキテクチャ, 修士論文, 京都大学 (1992).

[3] Pfister, H., Kaufman, A. and Chiueh, T.: Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization, In *1994 Workshop on Volume Visualization*, pp. 75–83 (1994).

[4] Pfister, H., Kaufman, A. and Wessels, F.: Towards a Scalable Architecture for Real-Time Volume Rendering, *In Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pp. 123–130 (1995).

[5] Nielson, G. et al.: *Scientific Visualization*, IEEE Computer Society Press (1997).

[6] BOER, M. et al.: Evaluation of a Real-time Direct Volume Rendering System, *Computer & Graphics*, Vol. 21, No. 2, pp. 189–198 (1997).

[7] Levoy, M.: Display of Surfaces from Volume Data, *IEEE Computer Graphics & Applications*, Vol. 8, No. 5, pp. 29–37 (1988).

[8] Samet, H. and Webber, R.: Hierarchical Data Structures and Algorithms for Computer Graphics, *IEEE Computer Graphics & Applications*, pp. 48–68 (1988).

[9] Montani, C. and Scopigno, R.: Rendering Volumetric Data using the

STICKS Representation Scheme, *SIGGRAPH '90*, Vol. 24, No. 5, pp. 87–93 (1990).

[10] 田山典男, 清水則明, 漆間文俊: 可変歩幅 DDA による 3 次元メモリ空間での高速なボクセル追跡, 情報処理, Vol. 32, pp. 581–589 (1991).

[11] 吉谷直樹ほか: *ReVolver*/C40 の設計及び実装, 情報処理学会研究報告, 97–ARC–125, Vol. 36, pp. 25–30 (1997).

[12] 重田大助: *ReVolver*/C40 のシステムコントロールユニット, 京都大学特別研究報告書 (1998).

[13] 木中一博: *ReVolver*/C40 のレイキャスティングステージの実装と評価, 京都大学特別研究報告書 (1997).

[14] 中山明則: ボリュームレンダリング専用並列計算機 *ReVolver*/C40 のピクセル値計算ステージの設計, 修士論文, 京都大学 (1996).

[15] 吉谷直樹: *ReVolver*/C40 のシェーディングステージの設計と実装, 京都大学特別研究報告書 (1996).

[16] Lacroute, P. and Levoy, M.: Fast volume rendering using a shear-warp factorization of the viewing transformation, *SIGGRAPH '94*, ACM, pp. 451–458 (1987).

[17] Reynolds, R. et al.: A dynamic screen technique for shaded graphics display of slice-represented objects, *Computer Vision, Graphics, and Image Processing*, Vol. 38, No. 3, pp. 275–298 (1987).

[18] Lacroute, P.: Fast volume rendering using a shear-warp factorization of the viewing transformation, Technical report, CSL-TR-95-678 , Stanford University (1995).

[19] Westover, L.: Footprint Evaluation for Volume Rendering, *SIGGRAPH '90*, Vol. 24, No. 4, pp. 367–376 (1990).

[20] 西松研, 瀬戸麻由子, 河野洋一, 福盛秀雄, 村岡洋一: 非構造格子を対象としたボリュームレンダリング 法の並列化, 情報処理学会研究報告, 97–HPC–65, pp. 45–50 (1997).

付録

A.1　ボリュームレンダリング緒手法

A.1.1　投影方式

3 次元のボリュームデータを 2 次元平面上で表現する際の投影方式には、平行投影法と透視投影法の 2 つがある (図 A.1)。

- 平行投影法
  物体中の各点をスクリーンに対して垂直に移動することにより、画像を生成する方式である。可視化された画像は、現実の物体の長さを正しく表しておりスクリーン上での物体の計測が可能である。
- 透視投影法
  物体を遠近法を用いてスクリーン上に可視化を行う。この方式により、人間の視覚と合致した自然な画像を生成することができる。

これらの方式はともに重要であるが、多くのボリュームレンダリング専用計算機やソフトウェアは処理の高速化を図るため、平行投影法による可視化を行っている [3, 4, 6]。

しかしながら、可視化対象となるボリュームデータが大きくなればなるほど、平行投影法と透視投影法により生成された画像の差は広がり、可視化結果に影響を与えることになる。

従って、透視投影法による可視化ができることが望ましい。

A.1.2　投影方向

ボリュームレンダリングにおいて、可視化は 2 つの方式に大別することができる。ピクセルをボクセルに投影する後方投影型と、ボクセルをピクセルに投影する前方投影型である。

ここでは、それぞれの代表的な例であるレイキャスティング法と、スプラッティング法について述べる。

レイキャスティング法　レイキャスティング法は、第 2.1節で述べたように、視点からスクリーンの各ピクセルに向けて発した視線が通るボクセルの色と透明度を式 (2.1) を用いて順次計算していく方法である。

一般的に次のアルゴリズムで表すことができ、スクリーン上の各画素を走査する y_p, x_p ループの中に、ボリュームデータを走査するための z_p ループが含

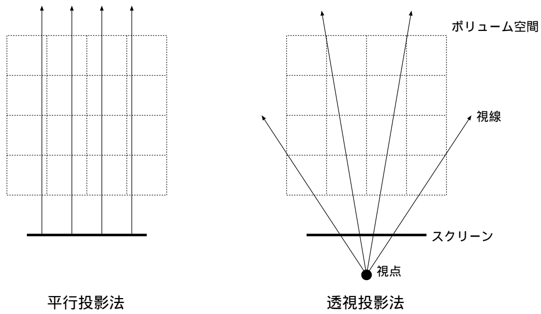


図 A.1: 平行投影法と透視投影法

まれている [18]。ここで、 ResamplingVoxel(x, y, z) は、スクリーン上のピクセル (x, y) をつらぬく視線が z 番目に交差するボクセルと、補間のために必要なボクセルの集合を返す関数である。また、 AddContriubution$(v_x, v_y, v_z, x_p, y_p)$ は、座標 (v_x, v_y, v_z) に位置するボクセルの影響を、スクリーン上の座標 (x_p, y_p) へ与える関数である。

Algorithm 9 レイキャスティング法

for $y_p = 1$ to スクリーンの高さ
    for $x_p = 1$ to スクリーンの幅
        for $z_p = 1$ to 視線上のボクセル数
            foreach v in ResamplingVoxel(x_p, y_p, z_p)
                AddContriubution$(v_x, v_y, v_z, x_p, y_p)$;

スプラッティング法　ボリューム空間を通過する視線に対する演算を行う後方投影法に対して、前方投影法は各ボクセルをピクセルに投影することにより、ボリュームデータを可視化する方法である。

代表的なものにスプラッティング法があり、ボクセルが持つ色と透明度をボクセルが影響を与える各ピクセルに対して、順次投影する手法である。

スプラッティング法は、次のアルゴリズムで表すことができ、ボリューム空間を走査する 3 重ループ z_v, y_v, x_v の中に、そのボリュームデータが影響を与える複数のピクセルを走査する 2 重ループ x_p, y_p が含まれる。

ここで、 DistributePixel(x, y, z) は、ボクセル (x, y, z) が影響を与えるスクリーン上のピクセルの集合を返す関数である。

Algorithm 10 スプラッティング法

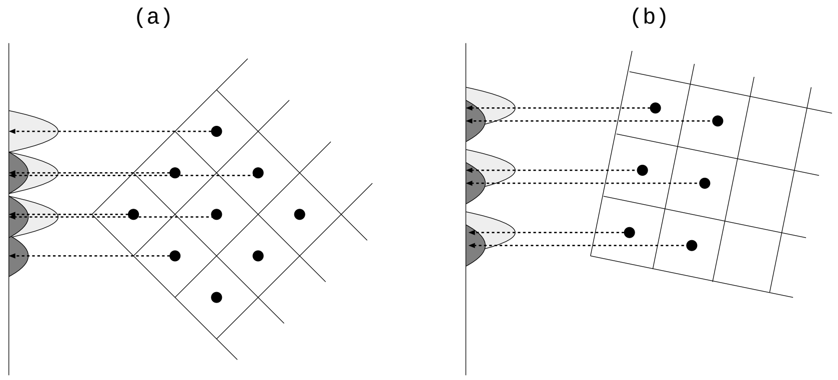


図 A.2: 前方投影における重み付け問題

```
for z_v = 1 to VolumeDepth
    for y_v = 1 to VolumeHeight
        for x_v = 1 to VolumeWidth
            foreach p in DistributePixel(x_v,y_v,z_v)
                AddContriubution(v_x, v_y, v_z, x_p, y_p);
```

　この手法では、ボリュームデータへのアクセスがデータの格納順となりボリュームデータに対して高速にアクセスすることが可能となるが、１つのボクセルは、投影されたスクリーン上の複数のピクセルに対して影響を与える。この影響を正しく与えるために、図 A.2(a) のようにガウス分布などの統計情報に従った重み付けが必要となるが、この重みはボクセルとスクリーンの位置に依存する。

　重みが小さな場合は、図 A.2(b) のようにスクリーン上に投影されたボクセルの間に隙間が空いてしまう。また、逆に大きい場合は、ボクセル同士が重なり合うように見え、実際よりも明るく見えてしまう。

　従って、高品質な画像を生成するためには、適当な重みを求めることが重要となるが、この重みを高速に求めることは困難である。特に透視投影の場合は、ボリューム毎に異なる重みを算出する必要がある。また、理論的には前方投影法も後方投影法と同じ画質の映像を生成することが可能であるが、そのためにかかる時間は非常に大きい [19]。

Shear-warp factorization 法　この手法は、近年注目を集めているオブジェクト順レンダリングアルゴリズムの 1 つである [16, 18]。この手法では、図 A.3のように 3 つの手順を経てボリュームデータの可視化を行う。

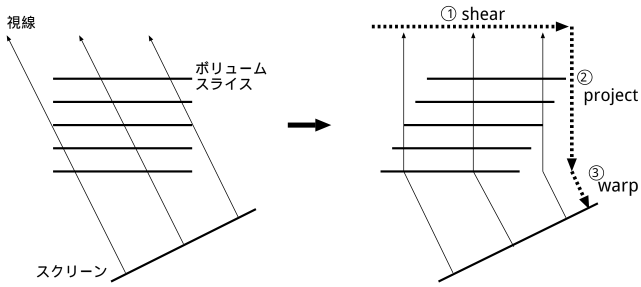


図 A.3: Shear-warp factorization

1. Shear 過程
   視線がボリュームスライスに垂直になるように、各スライス内でボリュームデータのリサンプリングを行う。
2. Project 過程
   前方投影を行う。スライスはリサンプリングされているので、この処理は単にスライスを、式 (2.2) により合成するのみである。
3. Warp 過程
   Project 過程で得られた画像は、Shear により歪められた画像であるので、2 次元の幾何変換により、歪みのない画像を生成する。

Shear-warp factorization 法では、ボリュームスライスとスキャンラインの対応を単純なものにすることで、次の利点を有する。

- 重み計算の省略
  視線とボクセルスキャンラインの位置関係が常に一定であるので、前方投影法で問題となる、ボクセル値分配のための重みを計算する手間を省いている。
- 並列性の向上
  スキャンライン毎に並列化が可能となる。
- アドレス計算の省略
  リサンプリングの結果、前方投影時にはアドレス計算が不要となる。また、ボリュームデータに対してその格納順にアクセスすることが可能となる。
- 無意味な演算の削減
  無色・透明なボクセルに対する演算など、最終生成画像に影響を与えない無駄な演算を容易に省くことができる。

Shear-warp factorization 法により、前方投影法での重大な問題であるボクセルがピクセルへ与える影響の重み付けにかかる計算時間を短縮することができる。しかしながら、リサンプリングに要する時間は無視できない。また、Warp 過程で行う画像の回転処理により、画像の劣化が生じる。

一方、レイキャスティング法は重み付けに関する問題を考えることなく、高品質な画像を得ることができるが、ボリュームデータの格納順序とアクセス方法が合致せず、描画速度に影響を与える。