グラフィクスとCAD　42－9
(1989. 12. 15)

空間分割による高速レイトレーシング

広田源太郎

日本アイ・ビー・エム株式会社
東京基礎研究所

オクツリー空間におけるのレイトレーシングアルゴリズムを提案する.また,このアルゴリズムが有効であることを均等分割空間におけるレイトレーシングとの理論的,実験的な比較によって示す.

Fast Ray Tracing by Space Subdivision

Gentaro HIROTA

Tokyo Research Laboratory, IBM Japan, Ltd.

An algorithm for tracing rays in the space that is subdivied by octree is proposed. Efficiency of this algorithm is shown by theoretical and experimental comparison with ray tracing in a uniformly subdivided space.

レイトレーシングの高速化

Whittedが，レイトレーシングの処理時間はその75%が光線とパッチの交差判定に費やされていると指摘して以来 [Whitted80], その高速化に大きな努力が費やされてきた．個々の交差判定の高速化も重要であるが，多数のパッチ（表示すべき面）からなるシーンを処理するためには，各光線と交差し得るパッチの候補を絞り，交差判定の回数を減らす必要がある,そのためにおもに3つの手法が研究されてきた．

1. hierarchical bounding volume
2. space subdivision
3. item buffer

いずれの手法もプリプロセスとして，パッチをなんらかのキーを用いて分類し,ある光線が与えられたとき，その光線と同じキーを持つパッチのみを候補としてピックアップし交差判定を行なう．

1.の手法はシーン内のパッチをいくつかのパッチ群に分け,それぞれに対しちょうど包含するような物体(bounding volume)を用意するものである [Whitted80] [Rubin80].キーはbounding volumeであり，光線と交差するbounding volumeに含まれるパッチのみが候補になる．一般的で強力な方法であるが,プリプロセスが複雑で自動化しにくいところに問題がある．

2.は本論文で用いている手法で，シーンの定義されている空間を小空間(ボクセル),に分割し，パッチをそれが含まれるボクセルをキーにして分類するものである.空間の分割はxy,yz,zxの各平面に平行に行なわれる [松本83] [Glassner84] [Fujimoto86] [Jansen86].分割のアルゴリズムは決定的で，自動的なプリプロセスが可能である.交差判定は光線が通過するボクセルに含まれるパッチに対してのみ行なわれる．

3.では光線の方向がキーになる [Haines86].いくつかの代表的位置（視点やパッチ上の座標）を始点とする光線を想定し,その始点を中心にした立方体を定義し，各方向に位置するパッチを立方体の該当する位置に記録しておく．立方体上の記録は各方向にどのパッチが存在するかを示したマップになる．これはラジオシティでフォームファクタの計算に使われるヘミキューブと非常に近いものである．この方法の最大の問題は，代表的な光線の始点としてどこを選ぶかである．空間分割と併用し，ボクセルを始点とする方法も考えられている [Arvo87].

空間分割

一般に空間分割が細かければ細かいほどボクセルに含まれるパッチが少なくなり，交差判定が減少して処理時間も短くなる．しかし分割を細かくすることは同時に処理時間を増やす作用も持つ．それをまとめると次のようになる．

1. プリプロセスタイム

   分割が細かくなるとパッチを含むボクセルが増加し，それだけポインタの代入が増えてプリプロセスも長くなる．ただ，パッチがボクセルより小さければ，そのパッチを含むボクセルの数はあまり増えないので影響は少ない．

2. ボクセル探訪時間(traverse time)

   分割が細かいということは光線が通過するボクセルが多いことを意味し，探訪時間は長くなる．

3. メモリアクセスタイム

   分割が細かくなればプリプロセスによって膨大なデータが作られることになる．大きなデータにアクセスすると，キャッシュのミスヒットやページフォールトが頻繁になり，1.と2.を含む全体の時間に影響する．

実際に処理速度を計測してみると1.は全体の処理時間の中で1%程度であり，あまり問題にならない．問題は2.と3.である．以下具体的なアルゴリズムに対してボクセル探訪時間(T_{trv}, time for traverse),交差判定時間(T_{eq}, time for solving equation),所要メモリ(s, space)の 3つの観点から理論的分析を行なってみよう．[Cleary88].

均等分割

最も簡単な空間分割は，xyz方向をそれぞれ均等に分割する均等分割（uniform space subdivision)である．データは3次元配列で表現され，ボクセルはすべて同じ大きさの直方体（立方体）になる．処理時間と所要メモリは次のようになる．

$$T_{trv} = C_1 nl \qquad (1)$$

$$T_{eq} = C_2 \frac{l}{n^2} \qquad (2)$$

$$s = C_3 n^3 \qquad (3)$$

ここでnは空間分割の分解能で，空間が$n \times n \times n$に分割されていることを意味する．簡単のため空間全体を単位立方体とすると，ボクセルの1辺は$\frac{1}{n}$になる．lは光線の平均の長さであり，シーン内のパッチの分布（単位体積あたりのパッチの面積）に強く依存する．一般的にいってlは空間全体で分布が密なら小さく，疎であれば大きくなる．$C_1 \sim C_3$は定数である．

式(1)はT_{trv}が通過するボクセルの数に比例することを示している．T_{eq}は光線があるパッチにヒットするまでに遭遇するパッチの数に比例する．確率的に単位体積あたりのパッチの数は一定なので，遭遇するパッチの数は光線が通過するボクセルの体積の総和に比例する．これを底面$\frac{1}{n^2}$高さlの角柱で近似し式(2)が得られる．

所要メモリは3次元配列なのでn^3に比例する．

実際の計算機上ではアクセスタイムの問題から消費メモリサイズは極力減らすべきである．特に実メモリに納まらないようなサイズになると，スラッシングによって著しく速度が低下してしまう．したがってアルゴリズムを比較するためには消費メモリサイズsを統一しなければならない．そこで処理時間をsで表わことにする．

$$T_{trv} = C_4 l s^{\frac{1}{3}} \qquad (4)$$

$$T_{eq} = C_5 \frac{l}{s^{\frac{2}{3}}} \qquad (5)$$

オクツリーによる空間分割

建築，インテリア，インダストリアルデザイン，自然物などのアプリケーションを考えたときパッチの分布というのは限られた体積内に集中している．ボリュームレンダリングの場合も，人間にはっきり知覚させるために，境界面を抽出して強調することが多い．このようにレンダリングのアプリケーションにおいては，表示すべきパッチは局所的に存在する．したがって均等分割を用いた場合，ほとんどのボクセルは空になってしまう．このようなデータを保持するためのメモリとその空間を探訪する時間の無駄は非常に大きい．

パッチの存在するところにボクセルを集中させるためには適応的に空間の分割を行なわなければならない．ここでは自動的なプリプロセスを容易にするため，最も単純な適応分割法であるオクツリーを採用した．では具体的にどう分割すれば良いだろうか．第1の方法は，物体の形状を表現するために使われてきたもので，面の存在する空間をすべてある精度まで分割するというものである．つまり面を含むボクセルはすべて同じ大きさになる．しかし従来レイトレーシングを行なう場合には，形状の表現精度よりボクセルに含まれるパッチの数に注意が払われてきたため，ボクセル内のパッチの数が一定以内に納まれば分割を終了するという第2の方法が一般的であった．ところがこの分割法は，パッチを含んでかつ，かなりサイズの大きなボクセルを生成する．大きなボクセルほど通過する光線が多いので交差判定の数が減らない可能性がある.またボクセルの座標そのものを交点計算に用いる場合は，その座標にある程度の精度が要求される．そこで2 つの方法の中間的な方法が必要になると思われる．　1つの有望な方法は一旦プリプロセスを終えてからレイトレーシングを開始し，多くの処理時間が費やされているボクセルを拾い出して，さらに細かく分割するというものである．

ここではすべて第1の方法を用いている．パッチを含むボクセルを$\frac{1}{n}$まで分割するとして，処理時間と所要メモリを推定してみよう．

パッチの分布が偏っているとみなせばツリーの探訪はほぼ最短時間で終わる．したがって

$$T_{trv} = C_6 \log nl \qquad (6)$$

となる．パッチを含むボクセルはすべて1 辺$\frac{1}{n}$なのでT_{eq}は式(2)と等しい．

メモリはほとんどオクツリーの末端のボクセル（リーフ）に費やされる.その数を推定してみよう．第1の方法でオクツリーを生成しているので最小のボクセルがリーフになる．パッチを含むボクセルの体積の総和 V_pはシーン内のパッチの総面積をAとすれば次式で近似できる．

$$V_p = \frac{A}{n} \qquad (7)$$

これをリーフ1つの体積

$$v = \frac{1}{n^3} \qquad (8)$$

で割ってリーフの数が得られる．したがって消費メモリは

$$s = C_7 n^2 \qquad (9)$$

となる．均等分割と同じくsを用いて書き直すと，

$$T_{trv} = C_8 \log s + C_9 \log l \qquad (10)$$

$$T_{eq} = C_{10} \frac{l}{s} \qquad (11)$$

となりオーダーで比較してオクツリーを用いた方が有利と言える．特に T_{eq}の差は明白である．

オクツリーの探訪

T_{trv}に関しては，それが非常に大きいとする研究もあり，詳しい議論が必要である．

オクツリーの探訪はおもに2つの部分に分けられる．1つは光線とボクセルの交差計算という数値的な部分．2つめは交差計算によって得られた位置から，交差するボクセルをツリーから捜し出すリスト検索の部分である．

オクツリー探訪の方式はまた，直列式とトップダウン式に分けられる．光線が通過するボクセルのうち，できるだけ小さいものを順番に追っていくのが直列式，通過するボクセルを常にツリーのトップレベルから下位へみていくのがトップダウン式である．リスト検索という点で2つの方式を比較してみよう.直列式の場合,隣のボクセルを見つけるためにツリーを登ったり降りたりするのでそのオーバーヘッドが大きい．トップダウン式の場合ツリーを降りる方向の検索しかない反面，ツリーの各レベルで検索の途中結果を保存しておかなければならず，探訪がすぐに終了してしまえば，この保存しておいた結果は無駄になってしまう．したがって各光線が通過するボクセルが少なければ直列式，多ければトップダウン式が有利になるが，レイトレーシングがどちらの場合に該当するかは判断が難しい．

交差計算という点ではどうだろうか．この計算は1種のクリッピングであり，浮動小数点の割り算のような高価な演算が使用されてきた．空間分割によって，光線とパッチの交差判定計算を少なくしつつも，ツリー探訪にはやむを得ず高価な演算を使っていることになる．

本論文で提案する方法は整数の加算とシフトのみによってツリーの探訪するものであり交差計算の面で計算コストを軽減するものである．

アルゴリズムの概説

オクツリーは空間をxyz各軸に垂直な平面で2等分している．光線も空間の分割に習って2等分していくことにしよう．

まずz方向の分割である．いま，空間が $0 \leq z \leq 1$で定義されているとする．このとき光線と2つの平面$z = 0, z = 1$との交点が整数の限られたレンジで表現できれば,その後の2等分は加算とシフトのみで実現できる．これは光線の方向に依存するので,個々の光線に対しxyz軸のうち適当なものを選びz軸と読み換えればよい．つぎにy方向の分割を考えよう．z方向の分割によって，z座標は必要なくなってしまったので，分割はxyの2次元空間で行なう.xy軸のうち適当な方をy軸と読み換えれば，直線$y = 0, y = 1$と光線の交点はやはり整数の範囲に納まるので計算は簡単である．最後にx方向の分割は簡単な1次元の2等分になる．

データ構造

レイトレーシングに最適なオクツリーのデータ構造について考えてみよう．まず消費メモリについて考えてみるよう．オクツリーの場合内部ノード（子を持つノード）はリーフの数の1/7程度である．内部ノードに使われるメモリを節約してもあまり意味がない.また,リーフはパッチのリストなので，そのボクセルが含むパッチの数だけポインタが必要であり,それ以下にデータを減らすことは難しい．したがって,リニアツリーのようなデータ圧縮法 [Samet88].は効果がない．

そこでオクツリーは,各ノードが子のノードを指す8個のポインタを持つ単純な方法を使用した．もちろんこのほうがアクセスが速い.また各ノードにディレクトリを用意し8個のポインタのうちNULLでないものがどこにあるかすぐに判別できるようにした．現在はまだ実現していないが，ディレクトリによって8個のポインタのうちNULLでないもののみがアドレッシングされているので,NULLポインタを除去することによりデータ圧縮も可能である．

探訪アルゴリズム

定義

表示しようとしているパッチは単位立方体の内部に存在しているとする.

$$(0 \le x \le 1, 0 \le y \le 1, 0 \le z \le 1) \quad (12)$$

追跡すべき光線を以下の式で表わす.

$$\vec{r} = \vec{o} + t\vec{d} \quad (13)$$

ここで

- $\vec{r} = (r_x, r_y, r_z)$ 光線上の点.
- $\vec{o} = (o_x, o_y, o_z)$ 光線の始点.
- t パラメータ. $t > 0$.
- $\vec{d} = (d_x, d_y, d_z)$ 光線の方向余弦.

ただし原点と座標軸は以下の関係を満たすように選ばれる.

$$0 \le d_x \le d_y \le d_z \quad (14)$$

初期化

$\vec{r}$上の点の存在しうる範囲をrange-X (R_x), range-Y (R_y), range-Z (R_z) (Fig.1) , range-R (R_r),の4つに分けて考える.

まずR_zを初期化する.光線$\vec{r}$を延長した直線Lを考え,平面$z = 0$との交点をP_{z0}, 平面$z = 1$との交点をP_{z1}とする.

$$\begin{aligned} P_{z0} &= (P_{z0x}, P_{z0y}, 0) \\ P_{z1} &= (P_{z1x}, P_{z1y}, 1) \end{aligned} \quad (15)$$

このときR_zの初期値は

$$\begin{aligned} &\{(x, y, z) \mid \\ &P_{z0x} \le x \le P_{z1x}, \\ &P_{z0y} \le y \le P_{z1y}, 0 \le z \le 1\} \end{aligned} \quad (16)$$

となる.

(14)の条件から整数の精度でも交点の計算がオーバーフローすることはない.次にR_yの初期値は,直線Lをxy平面に投影した直線L_1と直線$y = 0, y = 1$との交点P_{y0}, P_{y1}を

$$\begin{aligned} P_{y0} &= (P_{y0x}, 0) \\ P_{y1} &= (P_{y1x}, 1) \end{aligned} \quad (17)$$

としたとき

$$\begin{aligned} &\{(x, y, z) \mid \\ &P_{y0x} \le x \le P_{y1x}, 0 \le y \le 1\} \end{aligned} \quad (18)$$

となる.

やはり条件(14)から1つの例外を除いてオーバーフローの心配がない.例外は$d_x = d_y = 0$の場合, つまり光線がz軸に平行なときでxy平面では点(o_x, o_y)になる。ここではこれをy軸に平行な直線とみなし,

$$\{(x, y, z) \mid x = o_x, 0 \le y \le 1\} \quad (19)$$

とする.

R_xの初期値は, 平面$x = 0$と$x = 1$の間のすべて,つまり

$$\{(x, y, z) \mid 0 \le x \le 1\} \quad (20)$$

である.

最後にR_rは$\vec{o}$から$\vec{d}$方向の空間すべての範囲をR_oとしたとき,

$$\begin{aligned} &\{(x, y, z) \mid \\ &o_x \le x, o_y \le y, o_z \le z\} \end{aligned} \quad (21)$$

これと他のすべての範囲の交わりになる。

$$R_r = R_o \cap R_x \cap R_y \cap R_z \quad (22)$$

再帰的分割

オクツリーのノードは空間をxyzの各方向に2等分している. ツリーの探訪も同様に, R_z, R_y, R_xをz,y,xの各方向に分割することによって行なう.

以下アルゴリズムの記述 (Fig.2) に沿って説明する. まずR_zを分割しR_{zl}とR_{zr}とする (Fig.3a).ここで原点に近い方をl(left)側,遠い方をr(right)側とよぶ.この分割に必要な演算は, 整数の加算とシフトのみであり, 計算コストが小さい. ここでディレクトリをチェックし, ノードのl側の4つのポインタの中にNULLでないものが存在するかどうか調べる. 存

在すればR_rとR_{zl}の交わりを求め，空でなければR_yの分割へ進む．R_{zr}についても同様の処理を行なう．このように，常にl側からr側に探訪を進めることによって，光線の進行方向に沿った処理が行なわれる．

R_yの分割は2次元的に行なわれ（Fig.3b），引続くR_xの分割は1次元的に行なわれる（Fig.3c）．R_zの分割同様，常にディレクトリのチェックと交わりのチェックが行なわれ，不用な処理は省かれる．R_xの分割後，R_zの分割が再帰的に呼び出され，ツリーはトップダウンに探訪される．探訪中リーフに出会えばリーフのポイントするパッチと光線の交差判定を行なう．

```
OctreeTraverse(RAY)
  RAY : the ray
{
  O = root of octree
  Ro = bounding volume of RAY
  Rx = initial x-bounding volume
  Ry = initial y-bounding volume
  Rz = initial z-bounding volume
  Rr = intersection of Ro & Rx & Ry
& Rz
  IF (Rr is not empty)
    ZDIVID(O,Rr,Rx,Ry,Rz)
}
ZDIVID (O,Rr,Rx,Ry,Rz) {
  IF (O is leaf of tree)
    solve equation
    IF (ray intersect the primitive)
      calcurate intensity
    ELSE
      RETURN
  ELSE
    divid Rz into Rzl,Rzr
    IF (z left part of O is exist)
      Rrl = intersection of Rr & Rzl
      IF (Rrl is not empty)
        YDIVID(O,Rrl,Rx,Ry,Rzl)
    IF (z right part of O is exist)
      Rrr = intersection of Rr & Rzr
      IF (Rrr is not empty)
        YDIVID(O,Rrr,Rx,Ry,Rzr)
}
YDIVID (O,Rr,Rx,Ry,Rz) {
  divid Ry into Ryl,Ryr
  IF (y left part of O is exist)
    Rrl = intersection of Rr & Ryl
    IF (Rrl is not empty)
      XDIVID(O,Rrl,Rx,Ryl,Rz)
  IF (y right part of O is exist)
    Rrr = intersection of Rr & Ryr;
    IF (Rrr is not empty)
      XDIVID(O,Rrr,Rx,Ryr,Rz)
}
XDIVID (O,Rr,Rx,Ry,Rz) {
  divid Rx into Rxl,Rxr
  IF (x left part of O is exist)
    Rrl = intersection of Rr & Rxl
    IF (Rrl is not empty)
      ZDIVID(child of O,Rrl,Rxl,Ry,R
z)
  IF (x right part of O is exist)
    Rrr = intersection of Rr & Rxr
    IF (Rrr is not empty)
      ZDIVID(child of O,Rrr,Rxr,Ry,R
z)
}
```

Fig.2 algorithm

実験

上記のアルゴリズムと,均等分割空間をDDAで探訪するアルゴリズムをインプリメントし,実際にレイトレーシングを行なって計算時間を比較した．プログラムはすべてCで記述し,実行はIBM3081上で行なった.オクツリーの探査アルゴリズムは再帰的であるが,プログラミング上は再帰をすべて展開してループのみを用いた.

サンプルとして,フラクタルで生成した15,000ポリゴンの山の幾何データを用いた.これは限られた体積にパッチが集中している典型的な例である.またシーン内に設定された3つの点光源に対し影の生成を行い,光線の方向が分散するようにした．Fig.4がその結果である．Fig.4aはツリーまたは3Dアレイとパッチリストに消費されたメモリに対し,光線1本当たりの処理時間をプロットしたものである.またFig.4bは,同じく消費メモリに対して最小ボクセルの表面積（ワールド座標系での面積)と光線1本当たりの交差判定回数である.

この例ではオクツリーが圧倒的に有利である.また最小ボクセルがある程度小さくなると交差判定回数が集束しそれ以上効率が上がらないことが分かる.これは最小ボクセルがパッチに対し十分小さくなり式(11)がもはや成り立たなくなっていることを意味する.さらに小さなパッチを含むシーンに対しては集束位置が消費メモリの高位へ移動するはずである.

おわりに

さきに述べたとおり,現在用いているオクツリーによる空間の分割法は最適ではない.またNULLポインタの除去, 同一パッチに対する交差判定の除去など改良すべき点は多い.今後これらの改良を加えると同時に,より厳密な数学的解析と多様なシーンに対する計算時間の測定を行なう必要があろう.

文献

[Whitted 80] Whitted, Turner: An Improved Illumination Model for Shaded Display. Communications of the ACM, Vol. 23, No. 6, pp.343-349. (June 1980).

[Rubin 80] Rubin, Steve, and Turner Whitted: A Three-Dimensional Representation for Fast Rendering of Complex Scenes. Computer Graphics, Vol. 14, No. 3, pp. 110-116. (July 1980).

[松本 83] 松本均, 村上公一: Octree データ構造を用いた Ray-Tracing 法. 情報処理学会第27回全国大会講演論文集, pp. 1535-1537. (October 1983).

[Glassner 84] Glassner, Andrew S.: Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics and Applications, Vol. 4, No. 10, pp. 15-22. (October 1984).

[Fujimoto 86] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata: ARTS: Accelerated Ray-Tracing System. IEEE Computer Graphics and Applications, Vol. 6, No. 4, pp. 16-26. (April 1986).

[Jansen 86] Jansen, F. W.: Data Structures for Ray Tracing. in "Data structures for raster graphics," Springer Verlag, pp. 57-73. (1986).

[Haines 86] Haines, Eric A., and Donald P. Greenberg: The Light Buffer: A Shadow-Testing Accelerator. IEEE Computer Graphics and Applications, Vol. 6, No. 9, pp. 6-16. (September 1986).

[Arvo 87] Arvo,James, and David Kirk: Fast Ray Tracing by Ray Classification. Computer Graphics, Vol. 21, No. 4, pp. 55-64. (July 1987).

[Cleary 88] Cleary, John G. and Geoff Wyvill: Analysis fo an algorithm for fast ray tracing using uniform space subdivision. Visual Computer, Vol. 4, No. 2, pp. 65-83. (July 1988).

[Samet 88] Samet, Hanan and Robert E. Webber: Hierarchical Data Structures and Algorithms for Computer Graphics. IEEE Computer Graphics and Applications, Vol. 8, No. 3, pp. 48-68. (May 1988).

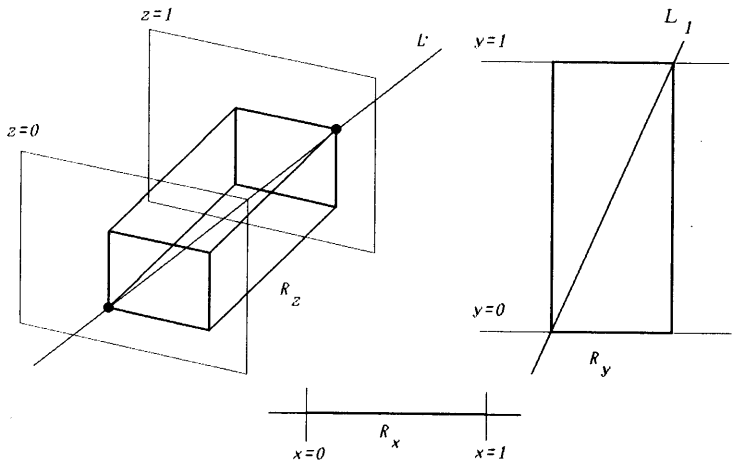

Fig.1 range R_x, R_y, R_z

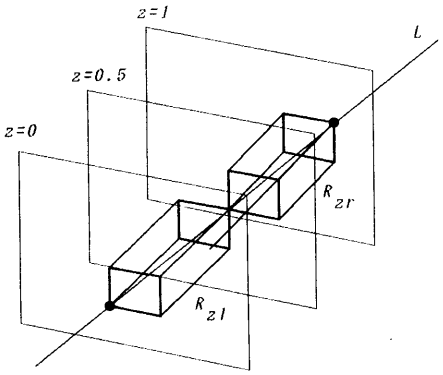

Fig.3a

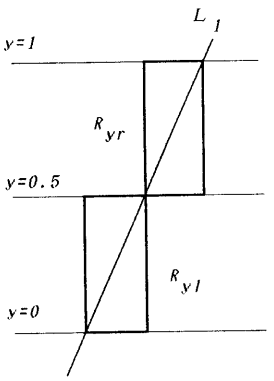

Fig.3b

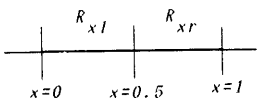

Fig.3c

Fig.3 binary subdivision of range R_x, R_y, R_z

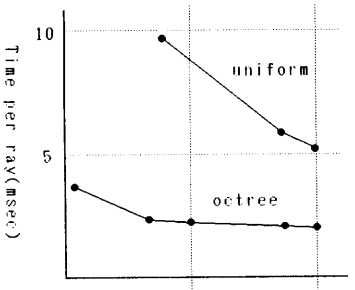

Fig.4a

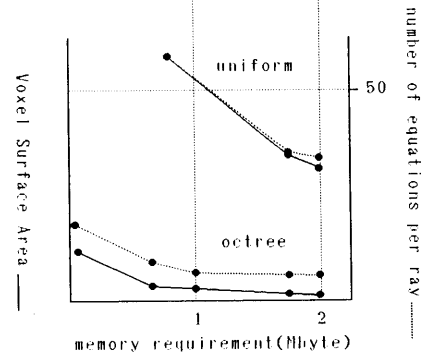

Fig.4b