空間分割モデルの高速な幾何変換

1E－2－2

西尾孝治　小堀研一　久津輪敏郎
大阪工業大学

1　はじめに

現在，2次元画像処理分野においてさまざまなアプリケーションで幾何変換が広く利用されている．

一方，計算機の処理能力や，記憶容量の問題から3次元画像であるボクセル[1]では幾何変換があまり利用されていなかったが，計算機の記憶容量の増加や計算機の処理能力の向上に伴い，ボクセルを用いたアプリケーションに対する要求が増えつつある．たとえば，2値ボクセルを用いた形状モデリングなどがあげられる．しかし，2次元画像のピクセルに用いられていた幾何変換をボクセルにそのまま適用すると，解像度の3乗に比例して計算コストが増加するという問題がある．また，ボクセルを保持するための記憶容量も，解像度の3乗に比例して増加する．

そこで，本研究ではハードウェア面からの改善を待つのではなく，ソフトウェア面からこれらの問題の解決法を提案する．

まず，3次元画像を扱う際，ボクセルで画像を表現するのではなく，空間分割圧縮の一種であるオクトツリー[2-4]を用いる．これにより，記憶容量を解像度の2乗に比例する程度に抑えることができる．

つぎに，オクトツリーを構成するオクタントの数は一般にオクトツリーで表現される形状の表面積に比例するので，幾何変換にかかる計算コストも，解像度の2乗に比例する程度に抑えることができる．

しかし，一般にオクトツリーの操作は8分木の再構成を伴うため，ボクセルの操作に比べて計算コストが大きくなる問題がある．これまでにも，オクトツリーの幾何変換手法[5]が提案されているが，オクタントを表す立方体に幾何変換を行いながら，オクタント同士の干渉判定をすることで，オクトツリーの再構成をおこなうため，ボクセルに比べて処理負荷が大きいという問題があった．

そこで，本稿では，オクトツリーの高速な幾何変換手法を提案する．

A Fast Affine Transformation for Spatial Partitioning Model.
Koji NISHIO, Ken-ichi KOBORI, and Toshiro KUTSUWA
Osaka Institute of Technology

2　ボクセルの幾何変換

幾何変換のマトリックスをMとし，幾何変換を行う前の各ボクセルの中心座標の位置ベクトルをp，pを変換して得られる位置ベクトルをp'とすると，p'は式（2－1）で表される．

$$p' = pM \qquad (2-1)$$

また，変換対象となるボクセルを集合V，変換後のボクセルを集合V'，位置ベクトルpで表されるV中のボクセルの濃度値を$c(p)$，位置ベクトルp'で表されるV'中のボクセルの濃度値を$c'(p')$で表すとすると，幾何変換後のボクセルの濃度値は式（2－2）で表される．

$$c'(p') = c'(pM) = c(p) \qquad p \in V \qquad (2-2)$$

しかし，式（2－2）を用いて幾何変換を行うと，ボクセルで表現された形状に対しては，正しい結果が得られない．たとえば，図1に示すように，ボクセルを2倍に拡大する場合，形状の連続性が保証されず，離散的な画像になってしまう．

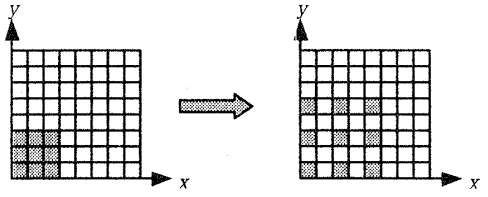

図1　ボクセルの幾何変換

この問題を回避するため，一般にはつぎのような方法が用いられる．

まず，目的とする幾何変換のマトリックスMの逆マトリックスM^{-1}を求める．ここで，pはM^{-1}を用い

て式（2－3）で表される．

$$p = p'M^{-1} \qquad (2-3)$$

したがって，変換後のボクセルをV'とすると幾何変換後のボクセルは式（2－4）で表される．

$$c'(p') = c(M^{-1}p') \quad p' \in V' \qquad (2-4)$$

図2（a)に示すように変換前のボクセルをVとすると，同図(c)に示す変換後のボクセルV'を同図(b)のように逆変換したV'とVの照合を行うことで，形状の連続性を保った変換結果を得ることができる．

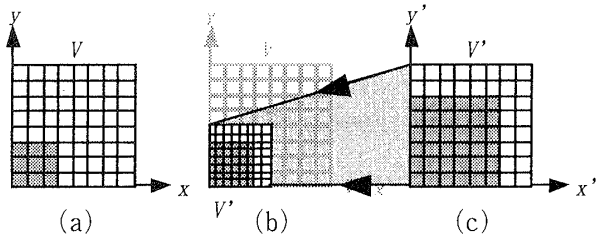

図 2　逆変換を用いた幾何変換

3　オクトツリーの幾何変換

3.1　計算コスト

空間分割モデルに対して幾何変換を行う場合，その処理時間は一般に空間を構成するセルの数に比例する．

ボクセルの幾何変換を行う場合，1つのボクセルを変換するのに要する時間をC_v，ボクセルの数をN_vとすると，すべてのボクセルを変換するのに要する処理時間T_vは式(3-1)で表される．

$$T_v = C_v \times N_v \qquad (3-1)$$

一方，オクトツリーの幾何変換は，1つのオクタントを変換するのに要する時間をC_{oct}，オクタントの数をN_{oct}とすると，オクトツリーを構成するすべてのオクタントを変換するのに要する処理時間T_{oct}は式(3-2)で表される．

$$T_{oct} = C_{oct} \times N_{oct} \qquad (3-2)$$

ここで，空間の一辺の解像度を2^nとすると，ボクセル数N_vは式(3-3)で表される．

$$N_v = O(2^{3n}) \qquad (3-3)$$

また，オクタント数は一般に形状の表面積に比例するので，オクタント数N_{oct}は式(3-4)で表される．

$$N_{oct} = O(2^{2n}) \qquad (3-4)$$

したがって，ボクセルの変換時間T_v，およびオクトツリーの変換時間T_{oct}は，それぞれ式(3－5)，および式(3－6)で表される．

$$T_v = O(2^{3n}) \qquad (3-5)$$

$$T_{oct} = O(2^{2n}) \qquad (3-6)$$

以上のことから，幾何変換の対象となる空間の解像度を2倍にすると，ボクセルの変換時間は2^3倍に，オクトツリーの変換時間は2^2倍になり，オクトツリーはボクセルに比べて解像度に対する計算コストの増加率が小さくなる．

一般に，ボクセル1つの変換時間C_vとオクタントの変換時間C_{oct}の間には式（3－7)の関係が成り立つので，変換対象となる画像の解像度が低い場合は，ボクセルの幾何変換のほうが高速になる．

$$C_v < C_{oct} \qquad (3-7)$$

そこで，本稿ではオクタントの高速な探索法であるノードアドレス法[6]を用いてオクタントの変換時間C_{oct}を短縮するとともに，オクトツリー全体の変換を高速化する手法を提案する．

3.2　ノードアドレス法

本手法では，オクトツリーのオクタントの探索法として，ノードアドレス法を応用して高速に8分木の探索を行う．これにより，オクトツリーの幾何変換の高速化が期待できる．

ノードアドレス法とは，オクタントの大きさと位置を3次元の2進実数を用いて表現する手法である．ここでは，説明の都合上2次元のクォードツリーを用いて説明する．

ノードアドレス法では，クォドラントを表す正方形の左下の座標を2進数実数で表現する．この2進

数の小数点以下の桁数がクォドラントの木に対する深さを表す．図3(a)のクォドラントaをノードアドレス法を使って表現すると，$(x, y)=(0.10_2, 0.01_2)$となる．

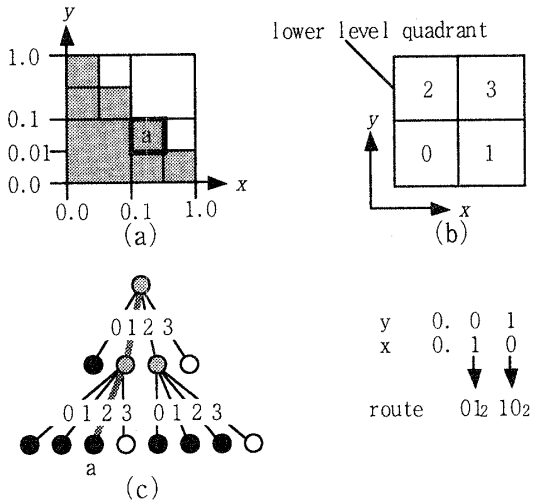

図 3　ノードアドレス法

次に，この表現を使ってルートクォドラントからクォドラントaへクォードツリーを探索する方法について述べる：

一般に，2進数実数vの小数第d位の値を$bit(v, d)$とすると，n次元の座標$v(v_1, v_2, \cdots, v_n)$を含むノードへの経路において，深さdのノードから深さ$(d+1)$のノードへの枝の番号b_dは式(3－8)で与えられる．

$$b_d = \sum_{i=1}^{n} \left\{ 2^{i-1} bit(v_i, d+1) \right\} \qquad (3-8)$$

クォードツリーでは各親クォドラントは4つの子クォドラントを持っている．これらの子クォドラントに0から3の番号をつける．この位置関係を同図(b)に示す．この番号は2桁の2進数で表現することができる．ここで，クォドラントaのノードアドレス表現のy成分，及びx成分の小数点以下の最上位を取り出すと0と1が得られる．これを2進数に対応させて01_2=1を得る．同様に最上位の次の位から10_2=2を得る．同図(c)からもわかるように，1，2はルートクォドラントからクォドラントaへクォードツリーをたどる場合の経路を表している．このように，ノードアドレス法を用いると，注目するクォドラントへのツリー上での経路を簡単に求めることができる．

オクトツリーの場合は，z座標が加わるが，同様にしてオクトツリー上での経路を求めることができる．

3.3　オクトツリーの幾何変換

ボクセルの幾何変換では変換後の出力画像のボクセルに対して逆変換を行うことで，形状の連続性を保った幾何変換を行っている．オクトツリーの幾何変換でも同様の手法を用いて，変換後に得られる形状の連続性を保証することを考える．

ボクセルの場合，出力画像もボクセルで表現されているため，これらのボクセルに対して逆変換を行うことは容易である．しかし，オクトツリーの場合は，幾何変換前の出力画像は空間全体を表現するルートオクタントがホワイトオクタントであるため，ボクセルのように空間が細かいセルの集合で表現されていない．このため，逆変換による形状の照合を行うことができず，幾何変換を行うことができない．そこで，本手法では以下のようにしてオクトツリーの幾何変換を実現した．

最初に，本手法によるオクトツリーに対する幾何変換の概要を図4に示す．ここでは説明の都合上2次元で示す．

図4（a）は変換対象となるオクトツリーを表す．本手法は，次の4つのステップから構成される．

（Step 1）シード探索処理

変換後に得られる形状の表面部分に存在すべきオクタントを1つ探索し，出力画像のオクトツリーに生成する．このオクタントをシードオクタントと呼ぶ．（図4(b)）

（Step 2）表面生成処理

シードオクタントをもとに，変換後の形状表面を26連結で探索し，形状表面のみを出力画像のオクトツリーに出力する．（図4(c)）

（Step 3）ペイント処理

出力画像のオクトツリーで形状内部のオクタントをブラックオクタントに変える．（図4(d)）

（Step 4）正規化処理

生成された出力画像のオクトツリーには，不必要に細分割されている部分があるので，これらの部分をより大きなオクタントにまとめることで，オクトツリー表現の冗長な部分をなくす正規化処理を行う．（図4 (e)）

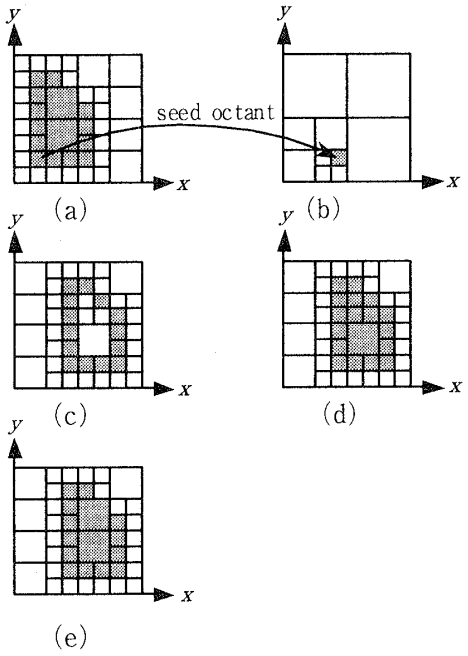

図 4　オクトツリーの幾何変換

次に，これらの処理の詳細を述べる．

3.4　シード探索処理

入力画像のオクトツリーの中で，形状表面に存在するオクタントを探索し，このオクタントに幾何変換を行う．これにより，出力画像のオクトツリーの形状表面に相当するオクタントが得られる．

しかし，幾何変換が拡大変換などの場合では，必ずしも変換後の形状表面のシードオクタントが得られるとは限らない．

そこで，幾何変換で得られたシードオクタントを逆幾何変換することで，入力画像のオクトツリーの形状表面に位置するかどうかを検証する．

このオクタントが入力画像中で形状表面に存在する場合は，出力画像中のこのオクタントをシードオクタントとして採用する．

これに対し，逆変換したオクタントが入力画像中で形状表面に位置しない場合は，出力画像中のオクタントの周囲26近傍に仮想のボクセルを生成し，これらを逆変換することで，形状表面に相当するボクセルを探索する．この処理を形状表面に位置するボクセルが1つ見つかるまで繰り返し，探索された形状表面に位置するボクセルをシードオクタントとする．

3.5　表面生成処理

まず，出力画像中のシードオクタントの周囲26近傍に仮想のボクセルを生成し，これらの中で形状表面に位置するものを出力画像のオクトツリーのブラックオクタントとする．

つぎに，出力画像のオクトツリー中の各ブラックオクタントに上記の処理を再帰的に行うことで，形状表面を構成する．

ここで，処理中のブラックオクタント o_n はボクセル群 S_n で表される周囲26近傍に対して再帰処理を呼び出すが，領域 S_{n-1} で表される o_{n-1} の周囲26近傍のボクセルはすでに再帰呼び出しが行われているので，これらのボクセルに対する呼び出しは冗長な処理となる．そこで，o_{n-1} から o_n へのベクトル v_n について，次の3通りに分類して，処理の冗長さを防ぐ．

（1）v_n が式(3-9)で表される場合，図5 (a)に示すように式(3-10)を満たす方向ベクトル v_{n+1} に沿った方向のボクセル群 S_n を呼び出す．

$$v_n = \begin{cases} (i,0,0) \\ (0,j,0) \\ (0,0,k) \end{cases} \qquad (3-9)$$
$$i,j,k \in (-1,1)$$

$$|v_n + v_{n+1}| > \sqrt{3} \qquad (3-10)$$
$$v_{n+1} = (x,y,z)$$
$$x,y,z \in (-1,0,1)$$

（2）v_n が式(3-11)で表される場合，同図(b)に示すように式(3-10)を満たす方向ベクトル v_{n+1} に沿った方向のボクセル群 S_n を呼び出す．

$$v_n = \begin{cases} (i,j,0) \\ (0,j,k) \\ (i,0,k) \end{cases} \qquad (3-11)$$
$$i,j,k \in (-1,1)$$

（3）v_nが式（3－12）で表される場合，同図(c)に示すように式(3-10)を満たす方向ベクトル v_{n+1} に沿った方向のボクセル群 S_n を呼び出す．

$$v_n = (i,j,k) \qquad (3-12)$$
$$i,j,k \in (-1,1)$$

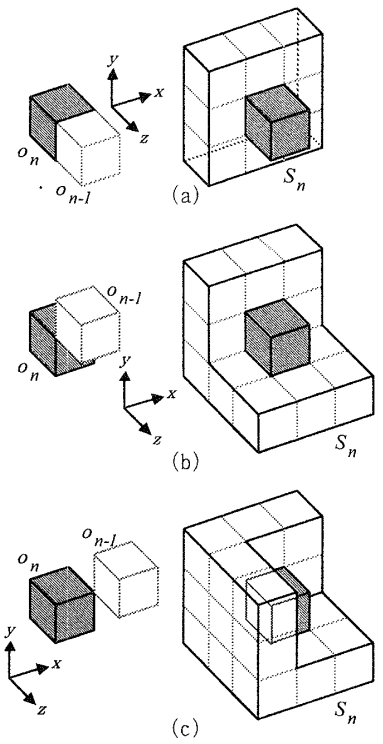

図 5　近傍の探索

このように再帰呼び出しの方向を限定することで，表面生成処理の計算コストを最高で 9/26 に，最悪でも 19/26 に削減することができる．

一方，2項演算の積，および和を一回行うのにかかる計算コストをそれぞれ C_{mult}，C_{add} とすると，3次元座標の幾何変換の計算コスト C_{trans} は式（3－13）で表される．

$$C_{trans} = 12C_{mult} + 9C_{add} \qquad (3-13)$$

これに対し，本手法では幾何変換によって再帰呼び出しされるオクタントの座標を求めるのではなく，26 近傍を表す基本ベクトルに幾何変換を行ったものを処理中のオクタントの座標に加えることで求める．このため，本手法による幾何変換の計算コスト C_{trans} は式（3－14）のようになる．

$$C_{trans} = 3C_{add} \qquad (3-14)$$

以上のことから，本手法では形状表面に位置するオクタントの幾何変換にかかる計算コストを削減することができる．

3.6　ペイント処理

表面生成処理では形状表面に位置するオクタントをブラックオクタントとして生成した．この状態では形状の内部，および外部はともにホワイトオクタントのままである．そこで，形状内部に位置するホワイトオクタントをブラックオクタントに変える必要がある．

この処理は，出力画像のホワイトオクタントを逆幾何変換して入力画像のブラックオクタントの内部に位置するものを，ブラックオクタントに変えることによって行う．

3.7　正規化処理

ここまでの処理で生成された出力画像のオクトツリーは形状表面の部分は図 6 (a)に示すように，必ず最小のオクタントで表現されている．しかし，同図(b)に示すようにより大きなオクタントで表現できる場合は，より大きなオクタントにまとめる．この操作をマージと呼ぶ．

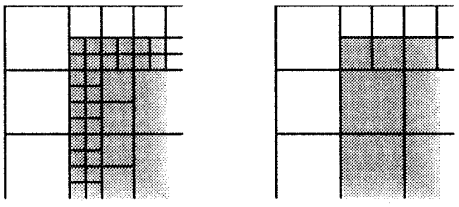

図 6　オクトツリーの正規化

具体的には，出力画像のオクトツリーのターミナルオクタントからルートオクタントへ向かって順にマージを行う．マージは，処理中のオクタントの 8 つの子オクタントがすべてホワイトオクタントであ

る場合はホワイトオクタントに，8 つの子オクタントがすべてブラックオクタントである場合はブラックオクタントに変えることで行う．

この処理により，冗長な表現を持たないオクトツリーを生成することができる．

4　実験

本手法の有効性を検証するために従来法との比較実験を行った．従来法として，ボクセルに対する幾何変換を取り上げた．空間の一辺の解像度を 2^{depth} として，depth を 5 から 9 まで変化させた場合の幾何変換にかかる処理時間を計測した．実験には図 7 (a)から(f)に示す形状をオクトツリー[7]，およびボクセルで表現したものを用いた．

なお，実験には，PC/AT 互換機 (Pentium Ⅱ 300MHz, Windows　NT) を用いた．

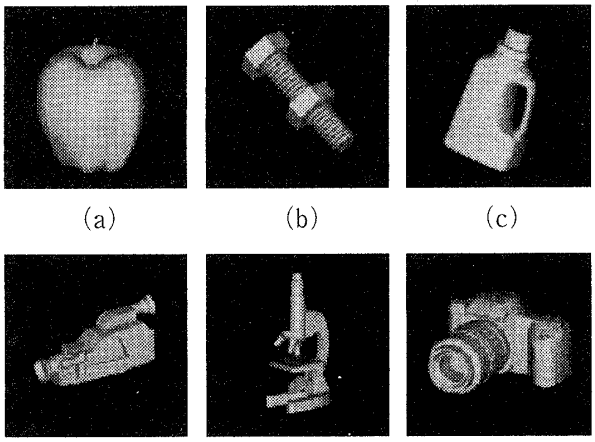

図 7　実験形状

（1）平行移動

図 7 (a)から(f)に示す形状を x，y，z 軸の各方向に 1 ボクセルだけ平行移動する幾何変換を行った．このときの解像度に対する 6 つの形状の平均処理時間を図 8 に示す．

（2）回転

（1）と同様に，原点を中心として x 軸回りに 5 度，y 軸回りに 10 度，z 軸回りに 15 度回転する幾何変換を行った．このときの解像度に対する 6 つの形状の平均処理時間を図 9 に示す．

（3）縮小

（1）と同様に，原点を中心として x，y，z の各軸方向に 0.5 倍する幾何変換を行った．このときの解像度に対する 6 つの形状の平均処理時間を図 10 に示す．

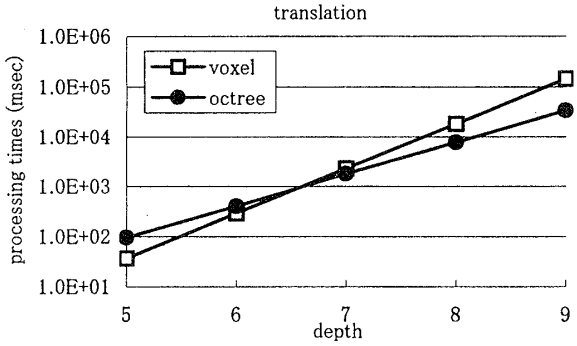

図 8　平行移動

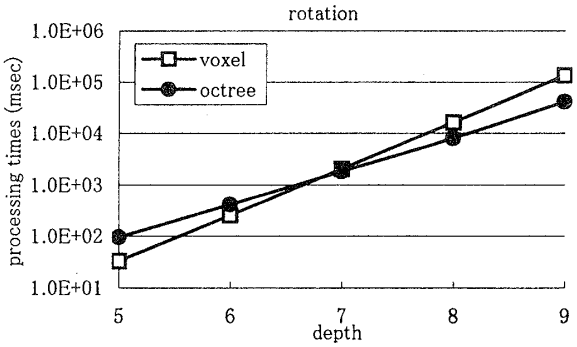

図 9　回転

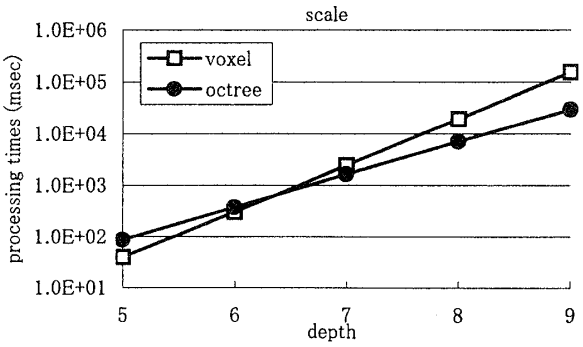

図 10　縮小

また，図7(a)から(f)に示す形状を空間の一辺の解像度を 2^{depth} としてボクセル，およびオクトツリーで表現した場合のボクセル数とオクタント数を図11に示す．

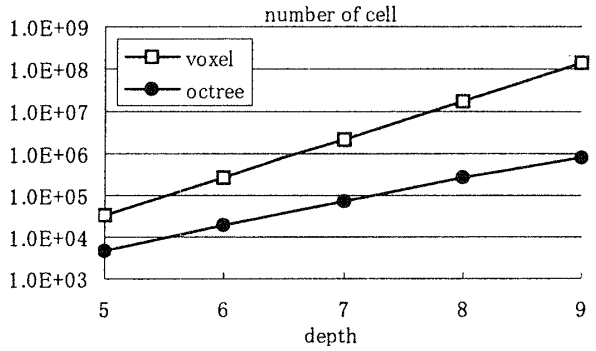

図11　ボクセル数とオクタント数の比較

また，ボクセルの濃度値を1bit,オクタントの濃度値を1bit で保持し，子オクタントを8つまとめてメモリに配置するものとして，その先頭ポインタを32bit で保持するとすると，オクタント1つあたり33bit の記憶容量が必要となる．以上のことからボクセル，およびオクトツリーの保持に必要な記憶容量を求めたものを図12に示す．

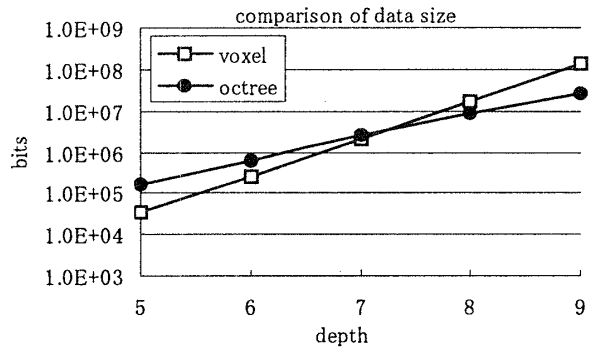

図12　データ量の比較

5　考察

まず，図8，図9，および図10からボクセルの幾何変換では処理時間は depth を1レベル上げるごとに約8倍に，オクトツリーの場合は約4倍になっていることがわかる．これは，幾何変換にかかる計算コストは変換を行うボクセル，およびオクタント数に比例するためと考えられる．

つぎに，変換対象となる画像の解像度が低い部分では，オクトツリーに比べボクセルの処理時間が短くなっている．これは，解像度が低い場合はボクセルに比べ，オクタント1つあたりの処理負荷が大きいためである．

しかし，扱う画像の解像度が高くなると，同解像度でボクセル数とオクタント数の差が大きくなるために，ボクセルにくらべてオクトツリーの処理時間が短くなる．

また，ボクセル，オクトツリーともに平行移動，および回転の各変換では同解像度でほぼ同じ処理時間であった．これに対して縮小変換では他の変換に比べてオクトツリーを用いた場合，処理時間が短くなった．これは，変換後の形状の表面積が減少したために表面生成処理の対象となるオクタントが減少したことと，形状の体積が減少したためにペイント処理の対象となるオクタントが減少したためと考えられる．

最後に，図11から解像度を2倍にすると，ボクセル数は約8倍，オクタント数は4倍になることがわかる．さらに，図12から解像度が低い場合はボクセルに比べオクトツリーのデータ量が多くなっているが，解像度が高くなるとオクトツリーのデータ量は少なくなる．なお，実用上のアプリケーションを考えると解像度に対応する depth は少なくとも7以上が必要であると考えられるため，本手法によるオクトツリーの幾何変換は有効であると考えられる．

6　まとめ

従来，3次元画像として一般に用いられていたボクセルに比較して，記憶容量と計算コストを削減したオクトツリーによる3次元画像の幾何変換を提案した．また，実験により，ボクセルにくらべて，幾何変換の対象となる画像の解像度の増加に対して，処理時間の増加を低く抑えることができることを確認した．

今後の課題として，本手法を応用して，オクトツリーを用いたCADシステムを構築することが考えられる．

【文献】

1) A.Kaufman and R.Bakalash:"Memory and Processing Architecture for 3D Voxel-Based Imagery", IEEE Computer Graphics & Applications, Vol.8, No.6, pp.10-23 (1988).

2) F.Foley, A.VanDam, S.Feiner and J.Hughes:"Computer Graphics Principles and Practice Second Edition", Addison Wesley Publishing Company, pp.533-562 (1990).

3) C.L.Jakins and S.L.Tanimoto:"Oct - Trees and Their Use in Representing Three-Dimensional Objects", Computer Graphics and Image Processing, Vol.14, No.3, pp.249-270 (1980)

4) D.Meagher:"Geometric Modeling Using Octree Encoding", Computer Graphics and Image Processing, Vol.19, No.2, pp.129-147 (1982).

5) J.Weng and N.Ahuja:"Octree of Objects in Arbitrary Motion": Representation and Efficiency",Computer Vision, Graphics, and Image Processing, Vol.39, pp.167-185 (1987).

6) 國井, 藤代:" ソフトウェア工学ハンドブック", オーム社, pp.465-468 (1986).

7) 石黒, 小堀, 久津輪:"境界表現から Octree 表現への一変換手法",システム制御学会論文誌, Vol.8, No.3, pp.97-105 (1995).