大規模ボクセルデータの高速生成アルゴリズム

Fast Generation Algorithm for Large Scale Voxel Data

○ 俵 丈展, 理研, 〒 351-0198 埼玉県和光市広沢 2 ー 1 , E-mail : takehirotwr@riken.jp
小野 謙二, 理研／北大, 〒 351-0198 埼玉県和光市広沢 2 ー 1 , E-mail : keno@riken.jp
Takehiro Tawara, RIKEN, 2-1, HIROSAWA, WAKO-SHI, SAITAMA, 351-0198, JAPAN
Kenji Ono, RIKEN, 2-1, HIROSAWA, WAKO-SHI, SAITAMA, 351-0198, JAPAN

Fluid analysis is a necessary component for developing manufacturing products. It is widly used in many areas: performance evaluation, shortening the development period and cost reduction. Voxel based fluid simulation is often used to simulate actual industrial products because of the simplicity and robustness. The method tends to be a large scale analytical model because a higher resolution of a voxel grid is required to increase the analytical precision. Our goal is to generate large scale voxels for large scale polygon data on a commonly used PC at short times. Our algorithm can handle incomplete polygon data as an input, and generates an octree as well as a regular grid. We also take into account to compute volume and area ratio of fluids per voxel in sub-voxel precision.

1. 序文

流体解析は工業製品の設計開発には不可欠であり，性能開発から開発期間の短縮やコスト削減まで様々な領域で役立っている．設計利用の観点からは，ターンアラウンドの短い FOA (First Order Analysis) 解析から詳細な物理モデルを利用した解析まで，目的に応じて様々なアプローチが存在する．

設計における流体解析のプロセスでは，実際の工業製品の複雑な形状を扱うこともあり，格子生成が最も労力を要する処理である．これは対象とする形状の複雑さが主要因であるが，実際の設計で用いるデータは不備なデータであることにも原因の一端がある．この入力となるジオメトリデータの不完全性や欠落が，格子生成の困難さを助長している点も見逃せない．データの不備を修正し，完全に水密な表面データを作成するのは，相当な労力を必要とし，CFD プロセスのボトルネックのひとつとなっている．

流体解析の FOA の代表的な方法として，ボクセルベースの方法がある．ボクセルベースの手法は，基本的に直交格子を用いて流体を計算する．この方法は形状の近似度により図 1 のように幾つかのレベルに分類できる．各近似レベルは，それぞれモデリングと計算のコスト・予測精度に特徴がある．これらのうち，バイナリボクセルを用いる手法は，物体形状を表すポリゴンとの接触判定を行えばよいだけであるので，不完全なデータでも格子生成に失敗することがなく，水密データを必要としない．このため，先の問題点を回避できる利点がある．

ボクセル法は，同じ大きさのセルで計算領域を全て覆うと計算セル数が多くなり，大規模な解析モデルになる傾向である．したがって，大規模かつ高速なボクセルモデルを生成するアルゴリズムの開発が必要となる．従来より，ポリゴンからボクセルを生成するアルゴリズムが提案されてきたが，大規模なモデルを短時間で生成できるものは少ない．また，不完全な入力ジオメトリに対して，合理的な近似ボクセルモデルを生成する必要がある．さらに，図 1 のバイナリボクセル，体積率・面積率の属性を算出する方法や Octree 格子の生成にも対応できる格子生成手法が望まれる．本論文では，上記のような特徴をもつボクセル生成のアルゴリズムとプログラムの開発について報告する．

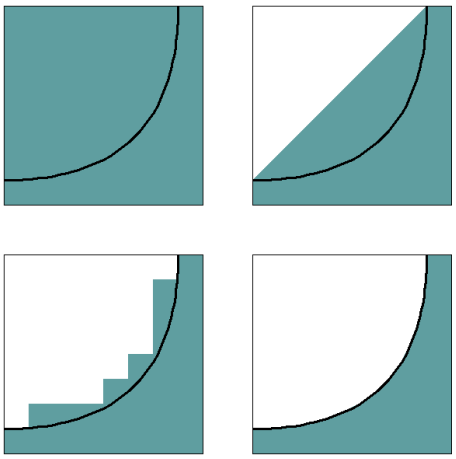

Fig. 1: Different kinds of a shape approximation in a voxel. Upper left) binary voxel, Upper right) a plane approximation, Lower left) sub voxel approximation, Lower right) implicit function.

2. 従来研究

3D オブジェクトの表現方法には，境界表現とボリューム表現に大別される．前者はポリゴンや曲面による陽的な表現で，後者は陰関数や他のボリュームデータによる陰的な表現である．本稿では，現在の CAE の主流である陽的な境界表現の中でも，もっとも簡単な形式の一つである STL フォーマットを入力として，形状のボクセル表現を行う方法を考える．この幾何形状のボクセル化をボクセライゼーションと呼ぶが，このプロセスでは元の形状から合理的な近似ボクセルを生成することが重要である．三次元オブジェクトのボクセル化は，2 次元オブジェクトにおけるピクセル化アルゴリズムの拡張[7]が提案され，整数演算による高速化アルゴリズムが開発された[6]．これらのスキャンライン変換をベースにした方法は，元のポリゴンにギャップが存在する場合には適切なボクセルを生成できない．

Huang et al.[4]は，ポリゴンや平面メッシュに対してト

ポロジー的にできるだけ正確なボクセルを生成する方法を提案した．これは離散的なボクセル空間において，対象となる有限厚さを考慮した面（ソリッド）と近隣ボクセルとの接続性を考慮し，良いボクセル近似を行っている．

Rueda et al.[9]は，テッセレーションやソーティングを要しない，ハードウェアに実装できる簡便でロバストなボクセル化法を提案，OpenGL ライブラリを用いて実装し，性能を評価している．評価に用いた節点数が 10 万程度と小規模であること，評価機が Pentium3 であることから，我々が扱う数百万節点を現実的に扱えるかどうかは明らかでない．また，本手法は等間隔のボクセル化に限られている．

ボクセライゼーションの高速化に関しては，近年の GPU の性能を援用してソリッドボクセライゼーションを行う方法が報告されている．Fan et al.[2]は，フレームバッファのブレンディング関数を利用して 3D オブジェクトのスライスを生成し，これらのスライスイメージを 3D テクスチャマッピングの機能を用いてボクセルモデルを作成する方法を提案した．これはハードウェアを利用した高速な方法であるが，スキャンラインを利用したラスタライズと同じ問題点がのこる．原田ら[10]はデプスピーリングとショートレイキャスティングを併用し，数百万頂点からなるポリゴンを数十ミリ秒でボクセル化することを実現している．しかしながら，この手法はレイキャスティングを基本としているため，ボクセル解像度よりも細かいポリゴンはボクセル形状に反映されない問題点がある．

我々は，数百万節点程度の大規模なポリゴンデータを対象に大規模なボクセルを PC 上で短時間で生成することを目的としている．また，等間隔ボクセルだけでなく，Octree にも対応したアルゴリズム，および欠損のあるデータに対するロバスト性，サブボクセル以下の情報の反映，セルの体積率や面積率などの情報の取得も考慮したアルゴリズムを開発した．

3. アルゴリズム概要

まず初めに，我々のアルゴリズムの概要を示す．入力は三角形メッシュとし，直交等間隔格子または Octree データを出力する．入力の三角形メッシュは数十万から数百万三角形を想定し，最小格子解像度は 512^3 を想定する．また，交差判定の高速化のため，全てのポリゴンは事前に三角形の集合に変換される．アルゴリズムは以下に示す 4 つのステップからなる．

(1) 境界ボクセルの判定

(2) 境界ボクセルの塗りつぶし

(3) 連続領域の塗りつぶし

(4) 体積率，面積率の計算

以下の節でそれぞれのステップについて詳しく解説する．

4. 境界ボクセルの判定

三角形メッシュからボクセルデータを生成する際，全ての三角形と全てのボクセルとの交差判定を行うことは現実的ではない．一つの三角形は基本的に局所的な極めて小さな領域に存在しているからである．このような空間的に局所的なデータを効率的に表現する方法として，空間を局所的に再分割していく Kd-tree と Octree が代表的である．以下の節でそれぞれの木構造を使って三角形メッシュと交差しているボクセルを効率的に判定する方法を解説する．

4.1 Kd-tree の生成

Kd-tree とは 2 分木を K-次元のデータに拡張した木構造のことである．2 分木が一次元の値の大小で木を生成していくのに対し，Kd-tree は各ノードで，ある一つの次元を選択し，その次元の値の大小を比較して 2 分木を生成していく．我々は各三角形を要素とし，2 つに分割される領域のどちらに含まれるかを判定し三次元データの Kd-tree を生成する．

各ノードでどの次元を比較対象にするかについては，$X \to Y \to Z \to X$ と順番に変えていく方法と，領域で最大幅を持つ軸を選ぶ方法が考えられる．我々はデータの領域が立方体以外の場合に対しても効率的な後者の方法を使用した．

次に，分割する位置をどのように決めるかという問題がある．もっとも単純な方法は選択された軸の幅の中心で分割する方法である．最良の方法は 2 つの領域で扱うデータ量が半分ずつになるようにメディアンを探す方法である．我々は，実装の簡単な前者の方法を使用した．領域内でデータが一様に分布している場合はメディアンを探す方法と同等の効率が得られる．

さらに，我々は，分割位置を直交等間隔格子に一致させるようにした．これにより，Kd-tree と直交等間隔格子との間で相互利用が可能となった．

以下に Kd-tree 生成の疑似コードを示す．このコードの中でもっとも時間のかかる部分は領域と三角形の交差判定である．我々は Separating Axis Theorem[3]に基づく，Möller[1]によって提案された高速な交差判定手法を用いた．

```
BuildKdTree(ids)
  bbox = GetBoundingBox()
  axis = GetMaxAxis(bbox)
  depth = 0
  Subdivide(root, bbox, ids, axis, depth)

Subdivide(node, bbox, ids, axis, depth)
  // code for a leaf node
  if (Terminate(node))
    node->setLeaf(ids)
    return;
  // code for an interior node
  split = AlignOnTheGrid(bbox, axis, pitch)
  children = new VxKdTreeNode[2]
  node->setInterior(children)
  lbbox = GetLeftBBox(bbox, axis, split)
  rbbox = GetRightBBox(bbox, axis, split)
  sz = ids.size()
  for (i=0; i<sz; i++)
    triangle = GetTriangle(ids[i])
    lids = IntersectBoxTriangle(lbbox, triangle)
    rids = IntersectBoxTriangle(rbbox, triangle)
  axis = GetMaxAxis(lbbox)
  Subdivide(node->left(), lbbox, lids, axis, depth+1)
  axis = GetMaxAxis(rbbox)
  Subdivide(node->right(), rbbox, rids, axis, depth+1)
```

さらに，Kd-tree のノードのデータは 8 バイトにまとめることが可能である．ノードのデータサイズが小さく

なれば探索に要する時間とメモリーの消費量を減らすことができる．ノードのデータはノードの種類によって保存される値が異なる．ノードの種類には木構造の末端の Leaf ノードとそれ以外の木構造の途中に位置する Interior ノードがある．Interior ノードでは最初の 4 バイトに分割位置 (30bit) と分割軸 (2bit) を保存し，後の 4 バイトに子ノードへのポインタを保存する．分割軸の値としては 0，1，2 がそれぞれ X，Y，Z 軸に対応する．Leaf ノードでは最初の 4 バイトに要素の数 (30bit) と Leaf のマーク (2bit) を保存し，後の 4 バイトに要素の配列へのポインタを保存する．Leaf のマークは Interior ノードと区別するために 2bit 必要で，数値の 3 が保存される．

```
VxKdTreeNode {
  union {
      uint m_split;   // used for an Interior node
      uint m_nelm;    // used for a Leaf node
      uint m_flags;   // used for both nodes
  };
  union {
      // used for an Interior node
      VxKdTreeNode* m_children;
      // used for a Leaf node
      VxKdTreeElement* m_elements;
  };
};
```

4.2 Octree の生成

Octree(八分木) は三次元領域を各軸で 2 分割し，8 つの領域に分けていく木構造である．Kd-tree に比べて，木の深さが浅くなる．そのかわり，常に 8 つに分割するためメモリーを余分に消費する．

また，我々は，流体解析で精度をあげるのに便利なように，隣接するノードの深さの差が 1 以下になるよう， 深さレベルの平滑化を行った．平滑化の手法は次の通りである．まず，木構造のルートノードからたどって，8 つの兄弟ノード間の隣接と従兄弟ノード間の隣接をすべて調べ，レベル差が 1 より大きい場合はレベルの浅い方のノードをマークする．次にマークされた全てのノードを 1 回分割する．分割の必要がなくなるまでこの操作を繰り返す．

以下に Octree 生成の疑似コードを示す．

```
BuildOctree(ids)
  bbox = GetBoundingBox()
  depth = 0
  Subdivide(root, bbox, ids, depth)

Subdivide(node, bbox, ids, depth)
  // code for a leaf node
  if (Terminate(node))
    node->setLeaf(ids)
    return;
  // code for an interior node
  children = new VxOctreeNode[8]
  node->setInterior(children)
  for (j=0; j<8; j++)
    cbbox[j] = GetChildBBox(bbox, j)
  sz = ids.size()
  for (i=0; i<sz; i++)
    triangle = GetTriangle(ids[i])
    for (j=0; j<8; j++)
      cids[j] = IntersectBoxTriangle(cbbox[j], triangle)
  for (j=0; j<8; j++)
    Subdivide(node->child(j), cbbox[j], clids[j], depth+1)
```

Octree ノードも Kd-tree と同様，8 バイトにデータをまとめることができる．Octree ノードの場合は分割軸の代わりに Leaf ノードのマーク (1bit) と平滑化のフラグ (1bit) を保存する．

```
VxOctreeNode {
  union {
      uint m_nelm;    // used for a Leaf node
      uint m_flags;   // used for both nodes
  };
  union {
      // used for an Interior node
      VxOctreeNode* m_children;
      // used for a Leaf node
      VxOctreeElement* m_elements;
  };
};
```

4.3 境界ボクセルの塗りつぶし

前節の Kd-tree もしくは Octree を参照して，三角形と交差するボクセルの媒質 ID を直交等間隔格子に埋めていく．媒質 ID は三角形の材質 ID とする．一つのボクセル内に異なった材質の複数の三角形を含むときは，最大の材質 ID をそのボクセルの媒質 ID とする．再帰的実装の場合スタックオバーフローが起こる場合があるため，我々は配列型のスタックによるループによる実装を行った．各格子には処理済みのマーク (1bit)，境界面のマーク (1bit)，媒質 ID(6bit) を保存する．媒質 ID は 6bit で 64 種類まで保存可能である．

5. 連続領域の塗りつぶし

直交等間隔格子上で媒質 ID の塗りつぶし操作を行う．塗りつぶしは，ある格子の隣接 6 面に対して未処理のボクセルを調べて連続する領域を同一の媒質 ID で埋めていく．この時，スタック (Stack) による Last In, First Out (LIFO) では未処理のボクセルが次々に積まれてメモリーを大量に消費するため，キュー (Queue) による First In, First Out (FIFO) の方法でループによる実装を行った．一つの連続する領域をすべて塗りつぶしたら，媒質 ID を一つ増加させ，新しい媒質 ID で次の連続領域を埋めていく．全てのボクセルが埋まるまでこの操作を繰り返す．

6. 体積率，面積率の計算

体積率，面積率の計算は境界ボクセルについてのみ行えばよい．我々は，誤差の判定が容易で頑健なスーパーサンプリングによる手法を用いた．スーパーサンプリング手法とは最小格子解像度のボクセルをさらに再分割することである．我々の Kd-tree，Octree はオリジナルの三角形の情報を保持しているため，境界ボクセルの再分

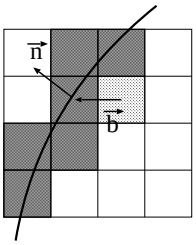

Fig. 2: Detection of an inside voxel. Darkly shaded voxels show boundary voxels and a lightly shaded voxel is an inside voxel.

割が可能である．一つの境界ボクセルの体積率，面積率の計算の手順を以下に示す．

(1) 境界ボクセルの判定

(2) 境界ボクセルの塗りつぶし

(3) 固体領域の塗りつぶし

(4) 流体領域の数え上げ

境界ボクセルの判定と塗りつぶしはそれぞれ第 4. 節と第 4.3 節の方法を最小格子のサブボクセルに対して行う．

固体領域の塗りつぶしは，第 5. 節の方法に少し手を加えた方法を使う．まず，図 2 で薄い影付きのボクセルで示されるような，隣が境界ボクセルであるボクセルを見つける．そして，隣接ボクセルへのベクトル $\vec{b}$ と隣接ボクセルの平均法線 $\vec{n}$(隣接ボクセル内に含まれる三角形の法線の平均) との内積が正のとき，そのボクセルは固体領域であると判定する．固体領域であると判定された場合，その地点から第 5. 節と同様の方法で塗りつぶしを行う．

最後に，体積率，面積率は流体領域のボクセルの数を体積，面積の総ボクセル数で割ったものとする．

7. 結果

計算のための PC として OS は 32bit Linux (Fedora Core 5)，CPU は Core 2 Duo E6600 (デュアルコア 2.4GHz) を使用した．現在，我々のソフトウェアはマルチスレッドに対応していないため，CPU は 1 コアのみを使用している．入力データのエンジンのモデルは約 224 万三角形で，自動車の外装のモデルは約 80 万三角形である．最小格子解像度を 512^3 とした場合の直交等間隔格子と Octree 格子のバイナリーボクセルの生成を行う．図 3 に Octree 生成時の画像を示す．また，表 1 に Kd-tree と Octree を使った場合のそれぞれの計算時間を示す．Octree を使った場合は隣接ボクセルの平滑化処理も含む．連続領域の塗りつぶしにかかる時間は最小格子解像度のみに依存するため，計算時間はほぼ同一になっていることがわかる．木構造の生成にかかる時間は Octree

	building	smoothing	filling	total
Kd-tree				
Engine	35.8	0	19.4	55.2
Automobile	11.4	0	19.9	31.3
Octree				
Engine	22.9	69.3	19.0	111.2
Automobile	6.9	13.7	20.3	40.9

Tab. 1: Timings for building trees. All units are in seconds.

	Nr. of leaf voxels	ratio
Kd-tree		
Engine	3,502,921	2.6 %
Automobile	1,053,715	.8 %
Octree		
Engine	4,290,049	3.2 %
Automobile	1,182,952	.9 %

Tab. 2: Number of leaf nodes and the ratio of leaf nodes in total voxels.

が 40%程速いことがわかる．これは，Octree の方が木構造の深さが浅くなる (Kd-tree で最大 27，Octree で最大 9，$2^{27} = 8^9 = 512^3$) ためと推測される．

表 2 に Kd-tree と Octree の Leaf ノードの数と全ボクセル数の 512^3 に占める割合を示す．Leaf ノードの数は Kd-tree の方が 10–20%少なく，切断の軸が適切に選択されてメモリーの消費量が少ないことがわかる．Kd-tree, Octree いずれの場合も生成に要する時間は少なく，現在一般的に使用されている PC で高速にボクセルの生成が行えることがわかる．

8. 今後の予定

前節で記したように，現在，我々のソフトウェアはマルチスレッドに対応していない．今後，PC の CPU はメニィコア化していくのは確実なので，我々のアルゴリズムもマルチスレッドに対応していく予定である．

現在のアルゴリズムは，三角形の隣接で最小格子解像度より小さな隙間等の小さなデータの不揃いには対応しているが，最小格子解像度より大きな穴等がある場合にはユーザーが期待するような媒質 ID による塗分けは困難である．このような問題を解決するには，Ju[5]，Rother et al.[8]による手法の利用が期待できる．

謝辞

図 3 で示される入力形状データは日産自動車株式会社よりご提供頂きました．

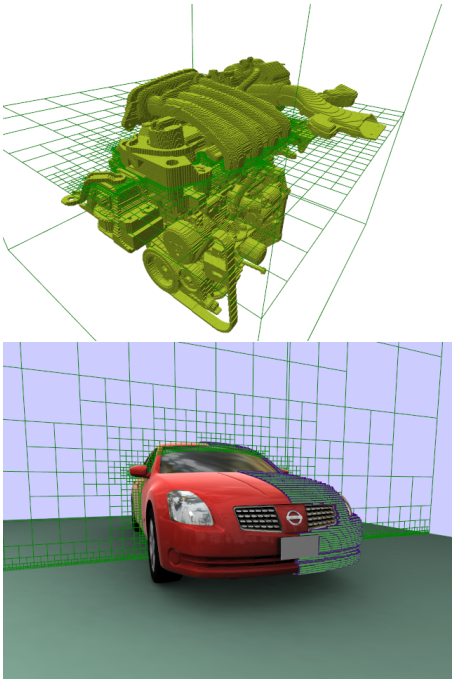

Fig. 3: Examples of Octrees. Upper image) an engine model, Lower image) an automobile model.

参考文献

1. Tomas Akenine-Möller. Fast 3D Triangle-Box Overlap Testing. *Journal of Graphics Tools*, 6(1):29–33, 2001.

2. S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

3. S. Gottschalk, M.C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Computer Graphics (ACM SIGGRAPH '96 Proceedings)*, pages 171–180, August 1996.

4. Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An Accurate Method for Voxelizing Meshes. In *IEEE Symposium on Volume Visualization*, pages 119–126, 1998.

5. Tao Ju. Robust Repair of Polygonal Models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH2004)*, 23(3):888–895, 2004.

6. A. Kaufman. An Algorithm for 3D Scan-Conversion of Polygons. In *Proc. Eurographics '87*, pages 197–208, August 1987.

7. A. Kaufman and E. Shimony. 3D Scan-Conversion Algorithms for Voxel-Based Graphics. In *Proc. ACM Workshop on Interactive 3D Graphics*, pages 45–76, October 1986.

8. Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "GrabCut": Interactive Foreground Extraction using Iterated Graph Cuts. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH2004)*, 23(3):309–314, 2004.

9. Antonio J. Rueda, Rafael J. Segura, Francisco R. Feito, Juan Ruiz de Miras, and Carlos Ogayar. Voxelization of solids using simplicial coverings. In *WSCG*, 2004.

10. 原田隆宏 and 越塚誠一. グラフィクスハードウェアを用いた高速なソリッドボクセライゼーション. In 日本計算工学会論文集, *No.20060023*, 2006.