Fast Large Scale Voxelization using a Pedigree

Takehiro Tawara[1]
Kenji Ono[2]

Functionality Simulation and Information Team,
RIKEN,
2-1, Hirosawa, Wako, Saitama, 351-0198, JAPAN.
[1]takehirotwr@riken.jp
[2]keno@riken.jp

Abstract

Fluid analysis is a necessary component for developing manufacturing products. It is widely 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 in terms of grid generation. The method tends to demand a large scale analytical model because a higher resolution of a voxel grid is required to increase the numerical precision. Our goal is to generate large scale voxels from large scale polygon data on a commonly used PC or workstation 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 ratios of fluids per voxel in sub-voxel precision.

Keywords: Voxelization, Octree, Pedigree.

Introduction

Fluid analysis is a necessary component for developing manufacturing products. It is widely used in many areas: performance evaluation, shortening a development period and cost reduction. In a process of fluid analysis of product design, generating grids is the most time consuming part because of handling a complex geometry of actual products. To make things worse, input geometry data is often incomplete: small gaps between vertices or edges, reversed normals, T-junctions and others. Fixing such incomplete data requires a lot of time and labor, and it becomes the bottle neck in the whole process of fluid analysis.

Voxel based fluid simulation is often used to simulate actual industrial products [12, 8, 1] because of the simplicity and robustness in terms of grid generation. However, the analytical model tends to be a large scale because a higher resolution of voxel grids is required to increase the numerical precision.

Therefore, a fast algorithm to generate a large scale voxel model is required. So far, there are not so many algorithms to generate such large scale voxels in a short amount of time. And it is also required to get reasonable approximation for an incomplete input geometry. Our goal is to generate large scale voxels for large scale polygon data on a commonly used PC or workstation 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 ratios of fluid per voxel in sub-voxel precision. In this paper, we propose such an algorithm and a software developed by the authors.

Previous Work

Representation of 3D objects is roughly separated into boundary and volume representations. The former is explicit representation by polygons or curved surfaces, and the later is implicit one by implicit surfaces or other volume data. In this paper, we consider voxel representation of a 3D shape defined by triangle mesh, which is the current mainstream representation of a surface in Computer-Aided Engineering (CAE). This Conversion from geometry data to voxel representation is called voxelization. In this process, it is important to generate rational approximation of original shapes. To voxelize 3D objects, rasterization algorithm in 2D was extended to 3D by Kaufman et al. [10] and a fast algorithm using integer arithmetic was also developed [9]. However, these scanline conversion methods couldn't generate proper voxels when original polygons had gaps.

Huang et al. [7] proposed an algorithm to generate possibly topologically accurate voxels for polygons and planar meshes. This algorithm achieved good approximation taking into account surfaces with finite thickness and connectivity to neighboring voxels in discrete voxel space.

Rueda et al. [13] proposed a simple and robust algorithm with no tessellation and no sorting, and they implemented and evaluated on hardware using OpenGL library. It is not sure to be able to handle million nodes, we need to handle, in practice, because evaluation has been done for 100K nodes using Pentium3 PC. In addition, this method limited to generate regular grids.

For acceleration of voxelization, several algorithms using GPU are reported. Fan et al. [3] generated slices of 3D objects using a blending function of a frame buffer and created voxel model from these slices using a 3D texture. Although this algorithm was fast, the same problems remained as using scanline conversion methods. Harada et al. [6] achieved voxelization for a few million vertices in tens of milli seconds using depth peeling and short ray casting. However, because this method was based on ray casting, smaller polygons comparing to voxel resolution were ignored.

Our goal is to generate very large scale voxels for a few million triangles in short time on a commonly used PC or workstation.

Algorithm Overview

Here, we briefly overview our algorithm. Triangle mesh is given as an input and a regular grid or an octree is generated as an output. We assume that the scale of an input is about a few million triangles and an output resolution is greater than or equal to 2048^3. For the fast intersection tests, all polygons are converted to a set of triangles in advance. Our algorithm consists of the following four steps.

1. Voxelizing Surfaces
2. Smoothing Octree Levels
3. Filling Regions in Octree
4. Computing Volume and Area Ratios

In the following sections, we discuss our algorithm in details.

Voxelizing Surfaces

First, input triangle mesh is voxelized and triangle IDs are stored in an octree data structure. An octree is a tree structure in 3D space. Because triangle mesh is a surface representation and triangles non-uniformly exist in 3D space, it is convincing to choose such an adaptive space partitioning data structure for memory efficiency. An octree is generated by a divide and conquer manner. So, 3D space is recursively subdivided until no triangle is included in a subdivided space or a recursive depth is reached to a user defined depth. For each interior node, a pointer to a series of eight children is stored. For each boundary node, a representative group ID of triangles in a node and a sequence of triangle IDs are stored. At the same time of building tree, an octree position, leaf and boundary flags are stored for every node. We discuss about an octree position in the next section.

The most time consuming part in this step is box-triangle intersections. We used the fast intersection algorithm proposed by Möller [2], which is based on Separating Axis Theorem [5].

Pedigree

In this section, we introduce a general representation of positional information in an octree to get better handling nodes in the later operations. An octal number of a child in a octree node is represented as $oct = x + 2y + 4z$, where x, y, z are binary numbers (i.e., 0 or 1). It is rewritten using bit operations $oct = x|(y \ll 1)|(z \ll 2)$. A child position (x, y, z) in a node is restored as $(x, y, z) = (oct\&1, (oct \gg 1)\&1, (oct \gg 2)\&1)$, vice versa. An octree node position is represented as an

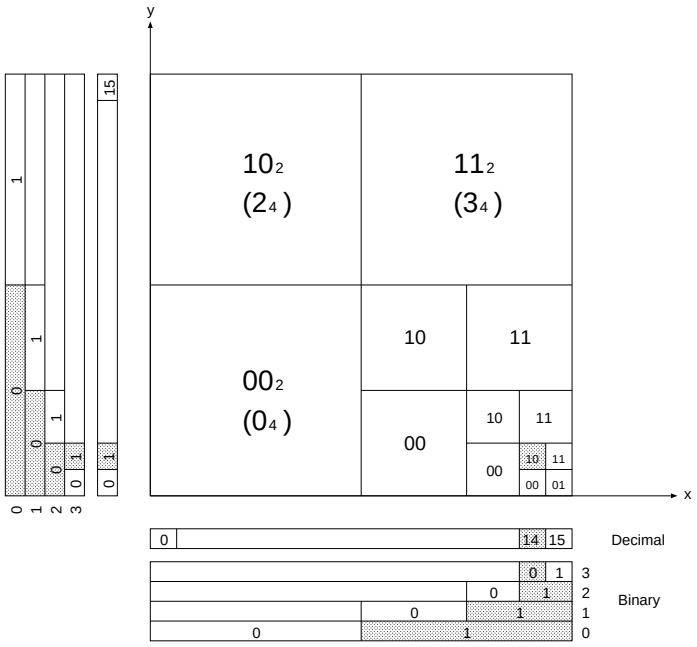

Figure 1: Quad tree index in decimal, binary and quad number. A shaded index is represented as (14, 1) in decimal, (1110, 0001) in binary, and 1112 in quad number.

series of octal numbers, each column represents the child position of each tree depth. We call this octree node position as a pedigree [11]. A pedigree is written as $\sum_{i=0}^{N} oct_i \ll (N-i)$, where N is the tree depth ($0 \leq N \leq 15$). To simplify finding neighbors, we store a pedigree decomposed into x, y, z axises and an octree depth in the form (x, y, z, d). Similar idea was reported as locational code [4].

Figure1 illustrates different representations of a position of a quad tree in the case of 2D space.

Memory Layout

One of key points of our algorithm is the memory layout per octree node. Our strategy is to compactly pack common data for each octree node and separates boundary data from an octree node. Here, we show pseudo code of our data structure.

```
class OctreeNode {
    OctreeIPos m_pos;
    union {
        OctreeNode* m_children;
        OctreeLeafData* m_ldata;
        OctreeBoundaryData* m_bdata;
    };
};

struct OctreeIPos {
    unsigned short x, y, z, w;
};
```

Figure 2 shows a memory layout of OctreeIPos data which is a decomposed pedigree data introduced in the previous section. Each octree node has 8 bytes of OctreeIPos data and one pointer to children for an interior node or boundary data for a boundary node. It costs only 12 bytes for a 32 bit system (16 bytes for a 64 bit system). Octree depth is stored in 4 bits. The maximum depth becomes 15 corresponding to $2^{15} = 32768$ regular grid resolution for each axis. It is enough large for our application. A medium ID is stored in 8 bits corresponding to 256 mediums. We also store one bit flags for filling, subdivision, boundary and leaf.

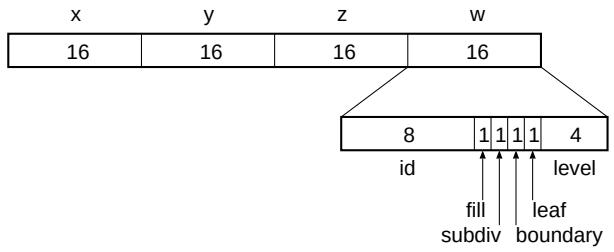

Figure 2: Memory layout of OctreeIPos packed in 64 bits.

Octree Traverse using a Pedigree

Another key point of our algorithm is a fast octree traverse mechanism using a pedigree. An octree data structure can be considered as an octal representation of regular grids. Similarly, a pedigree is an octal representation of a grid position. A set of the three left most bits of grid indices of three axes corresponds to an octal

position of a child in the first level of an octree (See oct_1 in Figure 3). In the same way, a set of the next bits becomes an octal position of a child in the second level of an octree (i.e., oct_2). Using this mechanism we can traverse an octree from the root node without any branching statement. There are two notes. One is that oct_0 is not used because the zero level is defined as the root node in our octree. The root node does not have an octal position representing itself. Another note is the significant bits are depend on the tree depth of a node. So the position of the left most bit differs for each node.

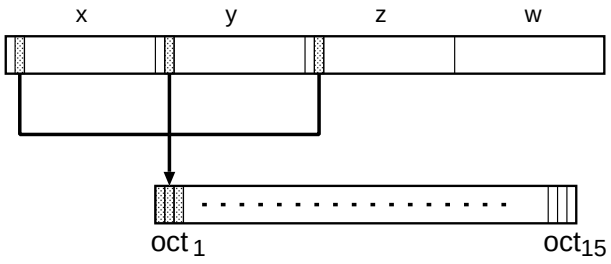

Figure 3: Pedigree composed from grid indices. An octal number oct_1 indicates the position of a child of the root node. Subsequent octal numbers $oct_2, oct_3, \cdots, oct_{15}$ correspond positions of a child for each level.

Finding Neighbors in Octree

Finding neighboring nodes in an octree is a most repeatedly executed function in smoothing and filling steps discussed in the following sections. We decides to have no explicit pointers to neighboring nodes because it easily explodes memory consumption and needs to reorganize connections when an octree structure is changed. Alternatively, we traverse in an octree using a pedigree as shown in the previous section. In fact, this search is further simplified for sibling nodes which are children of the same parent node. Three nodes of six neighbors (positive and negative X, Y and Z directions) are always siblings. Moreover siblings are allocated on consecutive memory space. Therefore, neighboring search for three directions can be done by a simple lookup table and shifting a pointer.

Smoothing Octree Levels

For better computation in fluid simulation, octree levels are smoothed so that the difference of levels of neighboring nodes is less than or equal to one. To make this condition, six neighboring directions are investigated. If we find a two level deeper node, the current node is marked to subdivide. All nodes except the last two

octree levels are processed because nodes in the two levels are never subdivided. It reduces the number of processing nodes to one-64th for fully subdivided octree. This process is repeated until all nodes have no subdivision flag.

Filling Regions in Octree

It is necessary to identify continuous regions for putting initial and boundary conditions in fluid simulation. For this purpose, we cluster regions in an octree. In painting applications, flood fill algorithm is well known. The algorithm first pick a start pixel, then the connected pixels with the same color of the start pixel are filled by a new color. We extend this algorithm to an octree structure.

First of all, boundary nodes are marked for filling regions when surfaces are voxelized. In the next, a non-marked node is chosen and filled from this node. Using a pedigree, our general positioning representation in an octree data structure, a continuous non-marked region is filled by an unique medium ID. We use a queue, which is a First-In-First-Out (FIFO) structure, to manage processing nodes for memory efficiency.

Computing Volume and Area Ratios

For accurate computation in fluid simulation, volume of fluid (VOF) and/or area of fluid (AOF) are required. To compute volume and area ratios robustly and accurately, we supersample a boundary voxel. This is possible in our case because we store a sequence of triangle IDs including in a boundary voxel. Alternatively, we can estimate ratios by a plane approximating a surface in a boundary voxel.

Results

We summarize the results of our voxelizer which has three procedures: building an octree, smoothing octree levels, labeling with medium IDs. Our software is executed on a workstation which consists of AMD Opteron 2220SE dual-core 2.8GHz x2, 16GB memory, 64 bit SuSE Linux Enterprise Server 10. Note that current our software does not utilize multi core processors, therefore only one core is used for computation. We tested two models: a Buddha model of 200,000 triangles and an automobile model of 3,815,564 triangles (See Figure 4). Each octree is split up to 12 and 11 levels respectively. It corresponds 4096^3 and 2048^3 regular grids each other. Table 1 shows memory consumption and timings. An automobile model requires more memory and times than a Buddha model because its geometry is more complex. In fact a Buddha model only occupies one-fourth of an entire region (2048x4096x2048 regular grids). As a result, we achieved generating a large scale voxels at very short times on a today's workstation.

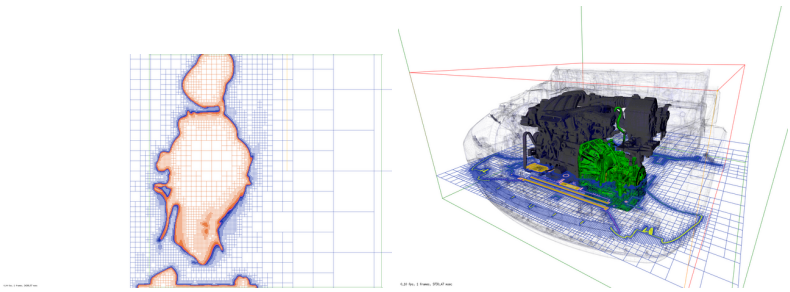

Figure 4: Screenshots of voxelization of Buddha and automobile foreside models.

	leaf	boundary	interior	total	time
Buddha	898.6 MB	1.2 GB	202.3 MB	2.2 GB	134.66 sec
4096^3	58,891,057	33,896,450	13,255,358		
Automobile	1.1 GB	1.9 GB	265.0 MB	3.2 GB	221.73 sec
2048^3	71,259,831	50,293,560	17,364,770		

Table 1: Memory consumption and Timings. The number of nodes is shown in the lower part of each cell.

Conclusions

We proposed a novel voxelization algorithm using a pedigree of an octree data structure. A presented method generates large scale voxels for large scale polygon data on a commonly used PC or workstation 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.

Acknowledgement

This study was partially supported by Industrial Technology Research Grant Program in '04 from New Energy and Industrial Technology Development Organization (NEDO) of Japan. Automobile data was provided by Nissan Motor Co., Ltd.

References

[1] M.J. Aftosmis, M.J. Berger, and J.E. Melton. Robust and efficient cartesian mesh generation for component-based geometry. *AIAA Journal*, 36(6):952–960, 1998.

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

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

[4] S. Frisken and R. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3), 2003.

[5] 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.

[6] T. Harada and S. Koshizuka. Fast Solid Voxelization using Graphics Hardware. In *Japan Computational Engineering, No.20060023*, 2006.

[7] 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.

[8] S.L. Karman Jr. Splitflow: A 3d unstructured cartesian/prismatic grid cfd code for complex geometries. In *AIAA-95-0343*, 1995.

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

[10] 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.

[11] T. Ogawa. An efficient numerical algorithm for the tree-data based flow solver. In *Computational Fluid Dynamics 2000*, pages 337–342, 2000.

[12] Kenji ONO, Hiroyuki AOKI, Sanae SATO, and Hiroyuki SHIOZAWA. Construction of rapid simulation tool for automotive cfd. In *8th International Conference on Numerical Grid Generation in Computational Field Simulation*, pages 81–90, 2002.

[13] 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.