A Flexible 3D Slicer for Voxelization Using Graphics Hardware

Hsien-Hsi Hsieh, Yueh-Yi Lai, Wen-Kai Tai* and Sheng-Yi Chang
Department of Computer Science and Information Engineering
National Dong Hwa University

Abstract

In this paper we present a simple but general 3D slicer for voxelizing polygonal model. Instead of voxelizing a model by projecting and rasterizing triangles with clipping planes, the distance field is used for more accurate and stable voxelization. Distance transform is used with triangles on voxels of each slice. A voxel is marked with opacity only when the shortest distance between it and triangles is judged as intersection. With advanced programmable graphics hardware assistance, surface and solid voxelization are feasible and more efficient than on a CPU.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations;

Keywords: voxelization, GPU, distance transform.

1 Introduction

Object representation is a broad topic in research. In computer graphics, polygons play a dominant role in 3D graphics because they approximate arbitrary surfaces by meshes. In games and animations, surface representation is the main technique used in rendering and visualization. However, volumetric representation, an alternative method to traditional geometric representation, is well known since the 1980s. It provides a simple and uniform description to measure and model a volumetric objects and establish the research field of volumetric graphics. Voxelization is a process of constructing a volumetric representation of an object. Voxelizing a polygonal object is not only a shift of the representation, it gives an opportunity to manipulate mesh object with volumetric operations such as morphological operation and solid modeling. Many applications, for example, CSG modeling, virtual medicine, haptic rendering, visualization of geometric model, collision detection, 3D spatial analysis, and model fixing, work on volumetric representation or use it as the inter-medium.

In this paper, we calculate the accurate distance field on GPU to compute coverage of each voxel in voxelization for polygonal models. Our method works for arbitrary triangulated models without any preprocessing for models, except organizing meshes slab by slab in order to prune the unnecessary computation while voxelizing complex models. By using the power of GPU, Hausdorff distance is guaranteed between each sampled voxel and the polygonal model. Surface voxelization with distance field on a GPU works well and runs faster than on a PC with a CPU. Our method is a reliable and accurate solution for the polygonal model under a given distribution of sampling voxels and a specific opacity criterion. Besides, error tolerance in voxelization is easy to manipulate by adjusting the threshold of opacity criterion for voxelization, which also dominates the smoothness of features and the thickness of surface voxelization.

*e-mail:wktai@mail.ndhu.edu.tw

The rest of paper is organized as follows. Some related works are surveyed in the section 2. In section 3, we present the computation of Hausdorff distance and our framework. The experimental results are illustrated in section 4. Finally, we conclude the proposed approach and point out some future works.

2 Related Works

Volume construction approaches are often referred to as scan-conversion or voxelization methods. Researchers mainly focused on modeling aspects such as robustness and accuracy. Wang and Kaufman [Wang and Kaufman 1993] used a method that samples and filters the voxels in 3D space to produce alias-free 3D volume models. They used filters to produce final density from the support of the region that polygons lie inside. Schroeder and Lorensen [Schroeder et al. 1994] created a volumetric model by finding closest polygon from distance map and classify the opacity of voxels. Huang et al. [Huang et al. 1998] described separability and minimality as two desirable features of a discrete surface representation and extended 2D scan line algorithm to perform voxelization. Dachille and Kaufman [Dachille IX and Kaufman 2000] presented an incremental method for voxelizing a triangle with pre-filtering to generate multivalued voxelization. Widjaya et al. [Widjaya et al. 2003] presented the voxelization in common sampling lattices as general 2D lattices including hexagonal lattices and 3D body-center cubic lattices. Ju [Ju 2004] constructed an octree grid for recording intersected edges with the model and expanded nodes to scan-convert a polygon on an octree, and then generate signs from the boundary edges and faces of the octree.

In recent years, attention on performance of voxelization raises. More and more studies try to explore the benefits of graphics hardware for more efficient rendering. Chen and Feng [Chen and Fang 1999] presented a slicing-based voxelization algorithm to generate slices of the underlying model in the frame buffer by setting appropriate clipping planes and extracting each slice of the model, which is extended and published in later [Chen and Fang 2000]. Karabassi and Theoharis [Karabassi and Theoharis 1999] projected an object to six faces of its bounding box through standard graphics system for the outermost parts and read back the information from depth buffer. However it works well only on convex objects. Dong et al. [Dong et al. 2004] proposed a real-time voxelization method using GPU acceleration to rasterize and texelize an object into three directional textures and then synthesize textures back to the final volume.

3 Hausdorff Distance Computation and Voxelization

In this section, we first discuss the computation of Hausdorff distance between a given triangle and a point. Then we explain how we use GPU to compute the distance field of triangles and modify the rendering pipeline.

3.1 Distance Field Computation

For a given 3D point $P(x,y,z)$ and a triangle $T(V_0,V_1,V_2)$, Hausdorff distance is the shortest distance between the point P and any point v on the triangle. A point on triangle T can be parametrically defined by two linearly independent vectors with two weights (s,t) by

$$T(s,t) = B + s\vec{e_0} + t\vec{e_1}, \tag{1}$$

where $(s,t) \in D = \{(s,t) : s \in [0,1], t \in [0,1], s+t \leq 1\}$, and $B = V_0, \vec{e_0} = V_1 - V_0$ and $\vec{e_1} = V_2 - V_0$.

For any point on triangle T, the distance from T to P is

$$\|T(s,t) - P\|, \tag{2}$$

or we can use the squared-distance function instead

$$Q(s,t) = \|T(s,t) - P\|^2, \tag{3}$$

where a point $p' = (\bar{s},\bar{t})$ exists which makes $Q(\bar{s},\bar{t})$ minimum.

Therefore, the computation of distance can be reduced into a minimization problem. For an efficient computation, we can expand $Q(s,t)$ as

$$Q(s,t) = as^2 + 2bst + ct^2 + 2ds + 2et + f, \tag{4}$$

where

$$\begin{array}{lllllll} a & = & \vec{e_0}\cdot\vec{e_0} & & b & = & \vec{e_0}\cdot\vec{e_1} \\ c & = & \vec{e_1}\cdot\vec{e_1} & & d & = & \vec{e_0}\cdot(B-P) \\ e & = & \vec{e_1}\cdot(B-P) & & f & = & (B-P)\cdot(B-P) \end{array} \tag{5}$$

From analyzing the gradient of $Q(s,t)$, the minimum $\bar{s}$ and $\bar{t}$ happens only when ∇Q is zero, where

$$\bar{s} = \frac{be-cd}{ac-b^2} \qquad \bar{t} = \frac{bd-ae}{ac-b^2} \tag{6}$$

If $(\bar{s},\bar{t}) \in D$, the minimum distance is the distance between p' and P; otherwise, according to the sign of $\bar{s}$ and $\bar{t}$, there are six possible regions that the shortest distance point p'' may lie on triangle T, as shown in Figure 1. Efficient solutions are well addressed on the book [Schneider and Eberly 2003] by CPU computation with simple calculation and logic classification.

However, in GPU, there is no efficient dynamic flow control to determine the shortest point on a triangle. Therefore, instead of directly computing the point of shortest distance on a triangle, we compute the distance from the 3D point to four possible points which may be inside the triangle or on the three boundaries and then the minimum scalar is the shortest distance. These four points are

$$\begin{array}{lllllll} (s_0,t_0) & = & (\frac{be-cd}{ac-b^2}, \frac{bd-ae}{ac-b^2}) & & (s_1,t_1) & = & (0,-\frac{e}{c}) \\ (s_3,t_3) & = & (\frac{c+e-b-d}{a-2b+c}, \frac{a+d-b-e}{a-2b+c}) & & (s_2,t_2) & = & (-\frac{d}{a},0), \end{array} \tag{7}$$

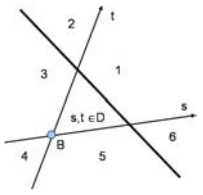

Figure 1: Six regions in s,t coordinate. Space is partitioned by range of parameters s and t for efficient shortest position classification and calculation.

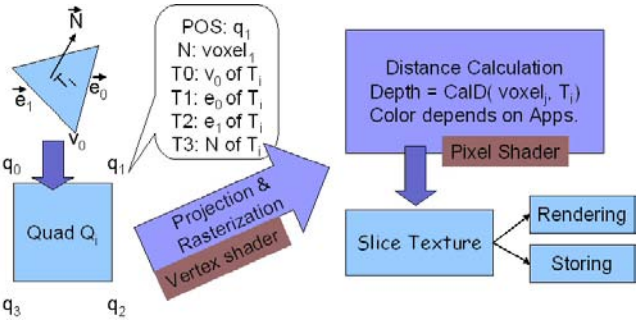

Figure 2: Rendering pipeline for generating a distance field of a triangle. A quad is rendered instead of a triangle. Five channels of each vertex of a quad (position, normal, and 4 texture coordinates) is filled with position of quad, position of voxel, and information of a triangle: v_0, $\vec{e_0}$, $\vec{e_1}$, and normal $\vec{N}$ respectively.

where position (s_0,t_0) assumes point p' is inside the triangle, positions (s_1,t_1), (s_2,t_2) and (s_3,t_3) assume point p'' is on boundaries of $s=0$, $t=0$, and $s+t=1$. All calculated points are truncated in the range of [0, 1] so that three end vertices of the triangle T are also in consideration and it guarantees these points are on the triangle for distance computation. Therefore, the minimum distance is the shortest distance from the point P to the triangle T.

3.2 Geometry Rendering for Voxelization

Voxelization by projection and rasterization faces the difficulty of non-uniform sampling of polygons because polygons with arbitrary orientations are not parallel to projection plane for maximum projection area. Even classifying polygons and projecting them to individual best-fit plane, there still have no guarantee on valid rasterization. However, distance field is omni-directional, i.e., insensitive to the projection plane, and has no assumption on input geometry and therefore no extra preprocessing is required.

Our approach is a slice-based approach for distance field generation and voxelization. Figure 2 shows the rendering process for generating the distance field for a triangle. For each triangle $T_i = \{T_i(s,t) | v_0 + se_0 + te_1, s \geq 0, t \geq 0, s+t \leq 1\}$, a full-filled quad $Q_i = \{q_{i_0}, q_{i_1}, q_{i_2}, q_{i_3}\}$ is rendered and rasterized to generate the distance field from voxels on a slice to the triangle. Triangle data and voxel positions are associated with the rendering quads. Voxel positions are stored in the channel of vertex normal, the triangle data (base vertex B, vectors $\vec{e_0}$ and $t\vec{e_1}$, and the normal $\vec{N}$) are separately stored in channels of texture coordinates and transmitted to GPU. Voxel positions are linearly interpolated in rasterization of rendering pipeline and pairs of triangle data and voxel positions are sent to pixel processors for Hausdorff distance computation. After distance computation, the shortest distance between a triangle and a

voxel is stored in the pixel depth and the pixel color is assigned for identification depending on applications. For example, binary surface voxelization uses color information to identify whether a voxel intersects a geometry such as 0 for empty and 1 for opacity; distance visualization uses color information to display the distance from geometries, etc.

The distance field of polygonal objects is constructed incrementally by rendering quads of triangles. Each pixel of depth buffer keeps the shortest distance from its voxel to triangles rendered. Depth buffer updates when a triangle is rendered. Unless distance is recalculated on different slice or rendered objects deform, quads which have been rendered have no need to be re-rendered again even new geometry are added in the scene. Depth buffer of the viewport is initialized as infinitude.

The rendering pseudo code is abstracted as follows:

```
for each triangle t on slice i {
  Create a quad Q for the triangle t
  for k = 0 to 3 {
    // assign a full-filled quad
    // q is end vertices of quad
    Q.q[k].position = ScreenBoundary.q[k];
    // assign voxel position, and triangle data
    Q.q[k].normal = Slice[i].q[k];
    Q.q[k].tex0 = t.B;
    Q.q[k].tex1 = t.e0;
    Q.q[k].tex2 = t.e1;
    }
  RenderQuad(Q);
}
```

3.3 Surface Voxelization

We use local distance field to reduce work load of GPU because the distance field far away from a triangle is meaningless for surface voxelization. For each triangle, we extend its parallel projected bounding rectangle by a small scalar for effective rasterization especially for triangles perpendicular to the projection plane. Due to coherence and precision in interpolating voxel positions, triangles are rendered with extended bounding rectangles. While a pixel is rasterized by a quad, Hausdorff distance is calculated according to the interpolated voxels, i.e., centers of voxels, and the triangle data. Only if the distance is less than the given threshold, e.g., distance from a uniform voxel center to its corner, the pixel is marked as opacity. Using local distance field could guarantee a small region of Hausdorff distance but greatly improve the performance of surface voxelization.

For more efficient voxelization process on GPU, triangles can be culled early by space partitioning. We construct an active triangle list for each slice. Currently we define slabs along Z-axis. According to partitioning planes, triangles are filtered and rearranged slab by slab. Many triangles can be pruned while rendering a slice. It is significantly helpful while voxelizing very complex models. Because distance field is insensitive to projecting directions of triangles, selection of partitioning plane has no influence on effectiveness of voxelization.

4 Experimental Results

We implement our fragment program using HLSL on a Pentium 4 3.0 MHz PC with 1G RAM and a nVidia Geforce FX5800 graphics card running on Windows XP with DirectX 9.0c. We use Vertex Shader 1.1 and Pixel Shader 2.0 to implement fragment program in scattering pairs of voxel positions and triangle data and in distance calculation and visualization. Table 1 shows the performance

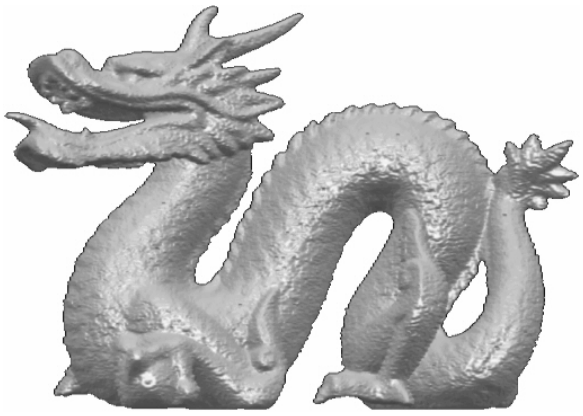

Figure 4: Rendering from the results of voxelization (512^3): dragon of 870K faces in 512^3 voxels.

of surface voxelization on different models and in different voxel resolutions. Figure 3 and Figure 4 demonstrate quality of voxelization results. In the experiment, opacity threshold is set to the distance from voxel center to its corner. That means if the shortest distance between a voxel and a triangle is less than the threshold, the voxel will be marked as opacity. Note that voxels are normalized to cubes in rendering so the scale of output may differ from the original polygonal model.

In Table 1, we list average time on surface voxelization per slice, per voxel, and per triangle. In the same resolution, voxelization time is proportional to the complexity of polygonal model. For each voxel, process time is always less than 0.1 ms. Even when voxel resolution increase, GPU still could handle voxelization for complex object in stable throughput which may be increased much more for CPU.

Due to speed up by using local distance field and culling for unrelated geometry, voxelization by distance field can be displayed slice by slice interactively under 128^3 voxel resolution. When voxel resolution is low, voxelization time highly depends on complexity of model. However, when the voxel resolution increases higher, even for a low complexity model, it still need more time to voxelize a model than in lower voxel resolution. On average, resolution of 256^3 could provide a benefit of reliable voxelization both in quality and time cost.

Rendering cost is stable for a triangle even when resolution of volume increases while it is linear on a CPU. Voxelization with proposed method is still slower than methods using traditional projection and rasterization by graphic hardware. However, our method is stable, correct and flexible because the opacity of each voxel is determined by thresholding the distance field of objects.

5 Conclusion

In this paper, we propose a GPU-based 3D slicer approach for voxelizing polygonal models. We calculate minimum distance between pairs of sampled voxels and triangles of arbitrary models with guarantee of Hausdorff distance. With programmable hardware vertex/pixel processors, efficient surface voxelization, solid voxelization, and visualization of the distance field all are feasible on the proposed 3D slicer.

However, in current implementation, performance of pixel shader is the bottleneck in overall processing speed. Area of rasterization also has a significant influence on the loading of pixel shader.

Model	Faces	Res.	Time(s)	$\frac{Time}{Slices}$	$\frac{Time}{Voxels}$	$\frac{Time}{Tri.}$	Res.	Time(s)	$\frac{Time}{Slices}$	$\frac{Time}{Voxels}$	$\frac{Time}{Tri.}$
Beethoven	5027	128	13.94	0.11	6.65	2.77	256	85.86	0.34	5.12	17.08
Teapot	6320	128	14.31	0.11	6.82	2.26	256	87.62	0.34	5.22	13.86
Cup	7494	128	15.24	0.12	7.27	2.03	256	93.65	0.37	5.58	12.50
Bunny	10000	128	16.24	0.13	7.75	1.62	256	93.98	0.37	5.60	9.40
Bunny	69451	128	43.21	0.34	20.60	0.62	256	231.85	0.91	13.82	3.34
Dragon	871414	128	84.21	0.66	40.15	0.10	256	325.87	1.27	19.42	0.37
Buddha	1087716	128	170.44	1.33	81.27	0.16	256	347.65	1.36	20.72	0.32
Dragon	871414	512	1748.11	3.41	13.02	2.01	* The time unit in $Time/Voxels$ is μs				
Buddha	1087716	512	1825.47	3.57	13.60	1.68	* The time unit in $Time/Tri.$ is ms				

Table 1: Surface voxelization on different models and in different voxel resolutions.

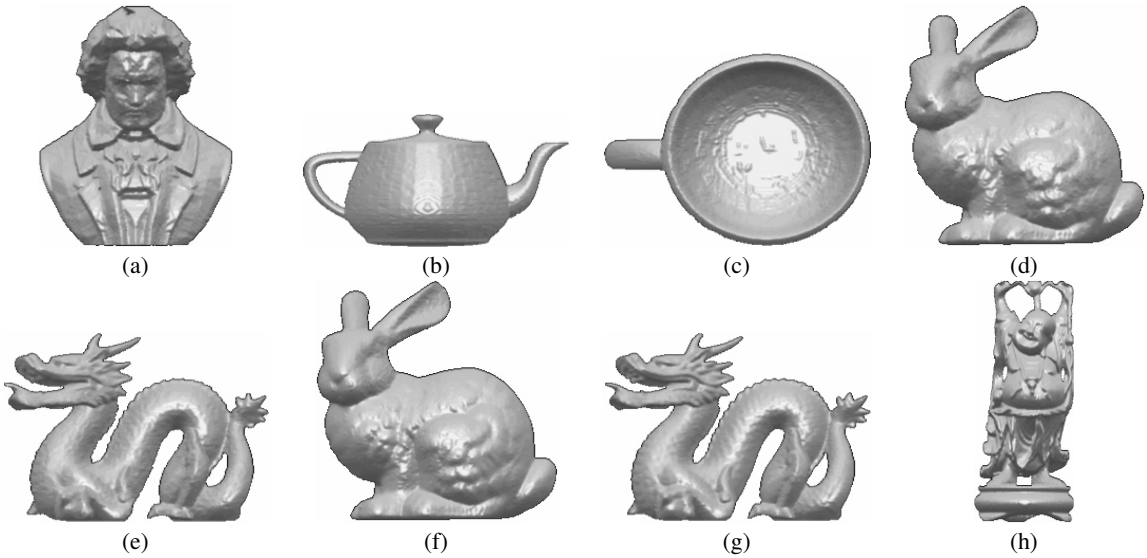

Figure 3: Rendering from the results of voxelization (256^3): (a) Beethoven in 256^3 voxels, (b) Teapot in 256^3 voxels, (c) Cup in 256^3 voxels, (d) Bunny of 10000 faces in 256^3 voxels, (e) dragon of 10000 faces in 256^3 voxels, (f) Bunny of 69451 faces in 256^3 voxels, (g) dragon of 870K faces in 256^3 voxels, and (h) Buddha of 1M faces in 256^3 voxels.

Therefore, in the near feature, searching a better computational methodology for GPU is one direction to improve performance of distance field computation. In addition, a sophisticated culling for error-free distance computation will be a technique in demand. To improve the quality of voxelization, adaptive dense voxelization and a mechanism for quality measurement and guide on GPU is another interesting topic.

References

CHEN, H., AND FANG, S. 1999. Fast voxelization of 3D synthetic objects. *ACM Journal of Graphics Tools 3*, 4, 33–45.

CHEN, H., AND FANG, S. 2000. Hardware accelerated voxelization. *Computers and Graphics 24*, 3, 433–442.

DACHILLE IX, F., AND KAUFMAN, A. E. 2000. Incremental triangle voxelization. In *Graphics Interface*, 205–212.

DONG, Z., CHEN, W., BAO, H., ZHANG, H., AND PENG, Q. 2004. Real-time voxelization for complex polygonal models. In *Proceedings of Pacific Graphics '04*, 43–50.

HUANG, J., YAGEL, R., FILIPPOV, V., AND KURZION, Y. 1998. An accurate method for voxelizing polygon meshes. In *Proceedings of IEEE symposium on Volume visualization*, 119–126.

JU, T. 2004. Robust repair of polygonal models. *ACM Transactions on Graphics 23*, 3, 888–895.

KARABASSI, G. P. E.-A., AND THEOHARIS, T. 1999. A fast depth-buffer-based voxelization algorithm. *ACM Journal of Graphics Tools 4*, 4, 5–10.

SCHNEIDER, P., AND EBERLY, D. H. 2003. *Geometry Tools for Computer Graphics*. Morgan Kaufmann.

SCHROEDER, W. J., LORENSEN, W. E., AND LINTHICUM, S. 1994. Implicit modeling of swept surfaces and volumes. In *Proceedings of IEEE Visualization*, 40–45.

WANG, S. W., AND KAUFMAN, A. E. 1993. Volume sampled voxelization of geometric primitives. In *Proceedings of IEEE Visualization*, 78–84.

WIDJAYA, H., MUELLER, T., AND ENTEZARI., A. 2003. Voxelization in common sampling lattices. In *Proceedings of Pacific Graphics '03*, 497–501.