GPU-accelerated Multi-valued Solid Voxelization by Slice Functions in Real Time

Duoduo Liao*
George Washington University

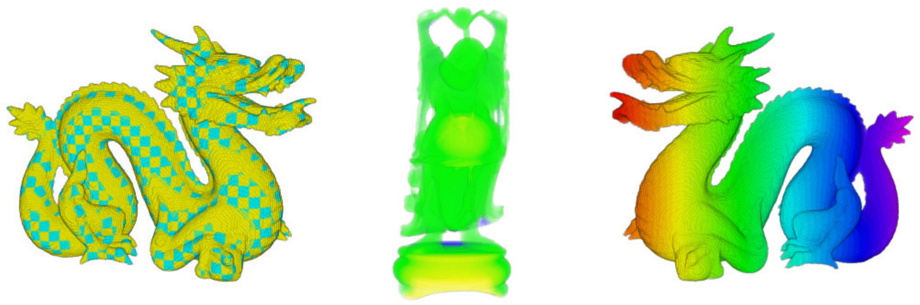

Figure 1: Solid voxelization results (256^3) with different slice functions. (a) is a volumetric dragon model in FUNC_COLORFUL_CHECKER, (b) is a volumetric happy Buddha model with a slice function of semi-translucent jade, (c) is a volumetric dragon model in FUNC_COLOR_RAMP.

Abstract

This paper presents a GPU-accelerated slice-independent solid voxelization approach that utilizes a dynamic slice function mechanism and masking techniques to significantly improve solid voxelization speed in real time as well as create various multi-valued solid volumetric models with different slice functions. In particular, by dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with any kind of interior materials, such as rainbow, marble, wood, translucent jade, etc. In this paper, the design of the dynamic slice function, the principle and algorithm of solid slice creation, the algorithm of real-time solid voxelization, and GPU-based acceleration techniques will be discussed in detail. The algorithms introduced in this paper are easy to implement and convenient to integrate into many applications, such as volume modeling, collision detection, medical simulation, volume animation, volume deformation, 3D printing, and computer art. The experimental results and data analysis for the complex objects demonstrate the effectiveness, flexibility, and diversity of this approach.

CR Categories: I.3 [Computer Graphics]: Solid Representations, Frame Buffer Operations, Volume Modeling, Solid Voxelization, Masking, Procedural Texturing, Hardware Acceleration

Keywords: volume modeling, solid voxelization, masking, solid slice function, GPU acceleration

*e-mail: dliao@gwu.edu, duoduo@acm.org

1 Introduction

Recently volumetric data have been used in a wide range of applications including medical imaging, scientific visualization, CAD/CAM, virtual reality, and entertainments. There is a tremendous demand for a wide variety of volumetric models and the approaches to generate them. While there has been significant progress in the area of volume rendering for over the past decades, creating and acquiring high-fidelity volumetric models remains a challenging and expensive process. Especially, how to fast acquire internal features of 3D objects in real time is still one of great challenges. Some biomedical scanning devices, such as CT/MRI, can acquire some internal features. However, these methods can be slow, expensive, destructive, impractical or, used for certain materials or objects. Voxelization is another way to acquire the internal volumetric data of geometry objects. It is an indispensable stage in volume graphics [Kaufman 1991].

The earliest voxelization idea, 3D scan-conversion [Kaufman and Shimony 1986] for voxel-based volume, was inspired by the extension of 2D scan-conversion for a pixel-based image. In early years, the voxelization research started with geometric objects, such as lines, curves, and surfaces, as studied in [Cohen and Kaufman 1995][Danielsson 1970][Huang et al. 1998][Kaufman 1987][Kaufman 1988][Mokrzycki 1988][Sramek and Kaufman 1999]. The extensive surface voxelization studies have continued in recent years [Chen and Fang 2000][Dachille and Kaufman 2000][Eisemann and D`ecoret 2006][Wang and Kaufman 1993]. But this kind of voxelization does not provide internal features. The voxelization of solid geometric objects, called *solid voxelization*, is much more difficult and time-consuming because this process requires an inside test for each voxel in the volume space. Generally speaking, less research on solid voxelization has been carried out. With the development of graphics hardware in recent years, some solid voxelization methods based on pure software, hardware acceleration, or software and hardware mixing has been studied. Unfortunately, limitations in algorithms, such as low speed, missing voxels, slice dependency, binary-only voxels, incontinuous voxelization, non-transparency, etc. still exist.

The main contribution of this paper is to propose a real-time solid voxelization approach using dynamic solid slice functions and masking techniques based on GPU acceleration to convert solid geometric objects into any size of diverse multi-valued volumetric models at the same time. This approach significantly improves the voxelization speed for solid geometric objects in real time. Furthermore, the algorithm is slice-independent voxelization processing, which is very suitable for parallelization. Additionally, Stencil buffer is a core technique used for the important masking techniques in this solid voxelization algorithm. Although stencil buffer technique has been used widely in many applications, such as shadow volumes, surface-based CSG rendering, it was first used for solid voxelization in [Liao 2001][Liao 2002]. This paper extends and enhances this technique to combine stencil buffer with GPU techniques to first propose the dynamic solid slice functions to generate diverse solid volumes.

The rest of the paper is organized as follows. Section 2 provides a brief survey of related work. Section 3 describes the definition and design of dynamic solid slice functions. Section 4 discusses the important principles and algorithms of the solid slice creation and real-time solid voxelization with arbitrary orientation, respectively. Section 5 provides some detailed experimental results. At last, the conclusions and future work are addressed.

2 Related Work

Although less research has been done on solid voxelization than surface voxelization in volumetric modeling and representation in a volumetric domain, the relevant prior work has been certainly studied for over the past years, but they are still limited.

In early year, a general purpose algorithm based on point classification [Lee and Requicha 1982] is too slow. Traditional pure software based 3D scan-conversion approaches for voxelization were studied thoroughly [Kaufman and Shimony 1986][Kaufman 1988]. Later, some algorithms based on filtering and distance volume use different filtering or distance values to fill all interior voxels or to thicken the boundaries of the polygons to avoid aliasing [Breen et al. 1998][Sramek and Kaufman 1999][Wang and Kaufman 1993]. Additionally, a new, morphological criterion was presented for determining whether a geometric solid is suitable for voxelization [Baerentzen 2000].

In recent years, more and more studies explore the benefits of graphics hardware for more efficient rendering. Some fast hardware-based techniques have been employed for solid voxelization. A slice-based algorithm [Chen and Fang 2000] for solid voxelization is presented through setting one slice-thick projection as a clipping plane to generate a voxel-based closed surface boundary and then applying a logical XOR operation between the current slice and the previous slice that has already been classified against the solid object. However, only a binary volume is generated due to XOR operations. The problem of missing voxels leading to prolonged lines exists.

Later, a stencil buffer based solid voxelization algorithm was first proposed in [Liao and Fang 2002]. The stencil buffer is utilized as a mask to generate interior voxels while solid geometric objects are clipped by a clipping plane without setting a thick projection for each slice. However, this algorithm needs to draw the object twice to count the number of front and back faces. The total voxelization speed is influenced by 2D texture mapping due to the large amount of memory swapping between GPU and CPU. Also, only one fixed color for an object volume is generated.

One multiple Z-buffer based solid voxelization algorithm [Karabassi et al. 1999] was proposed to use six z-buffers to project an object to six faces of its bounding box for the outermost parts and read back the information from depth buffer to synthesize the volume. However, it only works well on convex objects. Also, it take much time to generate the inside voxels due to using pure software methods to check each voxel. Later, this algorithm was improved to handle non-convex objects and use double layer buffering techniques to determine the number of actual intersections per voxel [Passalis et al. 2004]. However, this algorithm becomes slow as the number of z-buffers increases.

Recently, the max-norm distance computation algorithm [Varadhan et al. 2003] is used for the design of a reliable voxel-intersection test to determine whether the surface of a primitive intersects a voxel for solid voxelization and generate adaptive distance fields with the guarantee of Hausdorff distance. However, the overall algorithm runs slowly even with rasterization hardware for local refinement. Another distance-field voxelization approach by GPU acceleration [Hsieh et al. 2005] was presented but tested only for surface voxelization. The solid voxelization has not been fully tested due to the bottleneck of pixel shader performance.

Some GPU-based voxel coding approach [Dong et al. 2004] and [Eisemann and DŽcoret 2006] were proposed. The former uses GPU acceleration to rasterize and texelize an object into three directional textures and then synthesize these textures back to the final volume. This algorithm is designed for surface voxelization but could be extended to solid voxelization, which uses 3D scan-filling operation along three axis directions then check for common voxels. However, prolonged lines appear where some voxels are missing at the rasterization stage. The 3D scan-filling operation is time-consuming. The latter algorithm employs similar techniques. However, only a subset of voxels, instead of the entire interior voxels, is generated. Likwise, some voxels are missing as aligned with the view direction and some are not voxelized continuously due to a large slope in z direction.

3 Solid Slice Function

3.1 The Definition of Solid Slice Function

The realtime solid voxelization proposed in this paper is the slice-based voxelization. That is, such a volumetric object voxelized by a solid geometric object is composed of a set of solid slices. A *solid slice* is an image or texture with interior pixel information besides the boundary pixel information. Thus, the *i*th solid slice $\boldsymbol{S_i}$ can be described as $S_i(x, y) = \{f(x, y, i) \in R^3\}$, where $x = 1,..., Nx$, $y = 1,..., Ny$, $i = 1,..., Nz$, $f(x, y, i)$ is a solid slice function. Such a volumetric object $\boldsymbol{V}$ can be described as $V(z) = \{S_i, i = 1,..., Nz\}$.

In theory, a solid slice function can be any function, which is static or dynamic, continuous or incontinuous in 2D or 3D space or even higher dimensional space. For example, such a function could be a color, intensity, index, fixed texture, noise, procedural texture [Ebert et al. 2003], filter, blending function, transformation, deformation, logic, etc. In addition, for special purpose or application needs, these functions can be defined by users themselves. Specifically, the solid slice function f_s can be described as

$$f_s(x,y,z) = \begin{cases} fixed-value \\ f(x,y) \\ f(x,y,z) \\ f_{customized}(x,y,z) \end{cases}$$

The design of a solid slice function constitutes an important process of real-time solid voxelization. One or more parameters can be defined in each function, and may be modified dynamically by the users during interaction since the entire solid voxelization can be done in real time. That is, each solid slice defined in this paper can be not only filled by constant values but also described by a certain function. Furthermore, each solid slice can be applied by different functions to generate special effects for the volumetric object.

3.2 Several Typical Solid Slice Functions

Several typical functions used for this paper are described in Table 1. FUNC_NOISE_WOOD, FUNC_NOISE_MARBLE, and FUNC_NOISE_ROCK are solid slice functions defined by Perlin noise [Perlin 2002] in 3D continuous space. FUNC_MY_TYPE* are other customized solid slice functions such as translucent jade. These functions can be designed to meet users' special requirements according to their applications. In addition, some example effects of these solid slice functions are shown in Figure 1, 6, and 7. There are detailed explanations in Sec. 5.

Table 1: Math description of some typical solid slice functions

Function Type	Math Function Description	Comment
FUNC_SINGLE_COLOR	$f_s(x,y,z) = color$	*color*: a constant
FUNC_COLOR_RAMP	$f_s(x,y,z) = Colorramp\ (color[2], z)$	*color[2]*: two constants
FUNC_INTERPOLATED_COLORS	$f_s(x,y,z) = Interpolat\ e(color[4], x, y)$	*color[4]*: four constants
FUNC_COLORFUL_CHECKER	$f_s(x,y,z) = Chec\ \mathrm{ker}(color[2], z)$	*color[2]*: two constants
FUNC_NOISE_WOOD	$f_s(x,y,z) = Perlin\_noise(x,y,z)$	Perlin noise in 3D space
FUNC_NOISE_MARBLE	$f_s(x,y,z) = Perlin\_noise(x,y,z)$	Perlin noise in 3D space
FUNC_NOISE_ROCK	$f_s(x,y,z) = Perlin\_noise(x,y,z)$	Perlin noise in 3D space
FUNC_MY_TYPE*	$f_s(x,y,z) = f_{\mathrm{Customized}}(x,y,z)$	Other customized functions

4 Solid Voxelization

To achieve real-time solid voxelization, many OpenGL buffers techniques are used for solid slice creation. One of the core ideas of this approach is to use depth buffer and stencil buffer to create a mask by surface parity check and then to apply diverse slice functions to generate corresponding solid slices. Furthermore, using this idea, the multi-valued volume can be efficiently created only once at the same time of the voxelization. This is much different from prior inefficient multi-valued volume computation methods, in which a binary volume is usually first generated and then the value is applied to each pixel.

4.1 Surface Parity Check and Mask Creation

The surface of an object can be categorized as front or back facing, in relation to the position of the viewer. *Surface parity* refers to whether a surface in the depth buffer is inside or outside of a given volume. The parity of each depth buffer element can be determined by toggling a parity flag (i.e. front or back facing) at that pixel in front of the depth buffer. Regions of odd parity, where the flag is set to one, correspond to depth buffer elements volumetrically inside an object. The parity flag information is recorded correspondingly in the stencil buffer as a *mask*.

Figure 2 illustrates the principle of surface parity check on each solid slice. The parity flag is initially set to be zero. Given a slice plane, the parity of each depth buffer element is toggled at that pixel in front of the depth buffer. Along the Line Of Sight (LOS) from the viewpoint, the flag is toggled when the LOS intersects with front or back facets of the solid object which are in front of the slice plane. If the final parity flag is set to one, it shows that pixel is inside the solid object. Otherwise, the pixel is outside the solid object. In this approach, stencil buffer is used as a mask to store the parity check, which will be used to determine the interior information of the clipped solid object. After the mask created in the stencil buffer, the solid slice can be obtained through the mask in combination with the solid slice function.

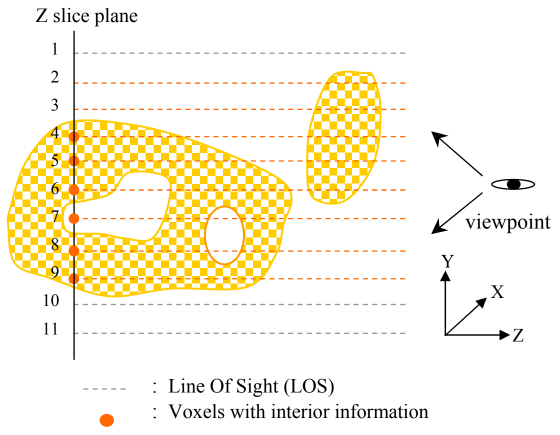

Figure 2: Surface parity check and mask creating

In general, all the processes of the surface parity check and mask creating can be implemented by GPU hardware. This important feature makes full use of GPU hardware acceleration. Consequently, the solid slice can be obtained at interactive or even real-time speed, which depends on the complexity of input solid slice functions.

4.2 The Principle of Solid Slice Creation

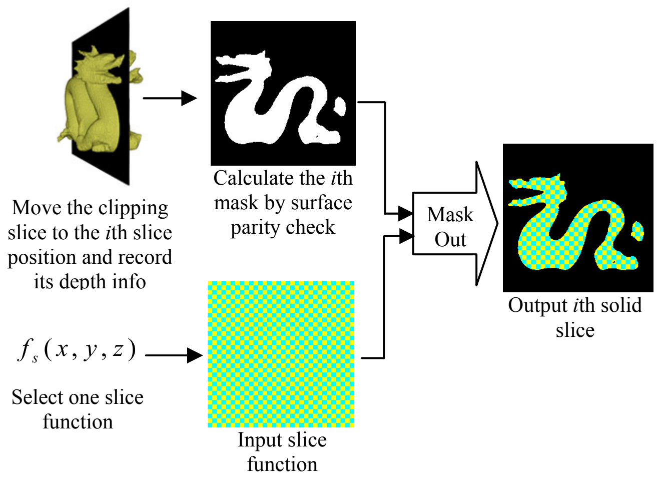
Move the clipping slice to the ith slice position and record its depth info
Calculate the ith mask by surface parity check
Mask Out
Output ith solid slice
$f_s(x,y,z)$
Select one slice function
Input slice function

Figure 3: The principle of one solid slice creation

Figure 3 illustrates the principle of one solid slice creation based on masking and solid slice function techniques. First, one slice plane is moved to its corresponding position in 3D space. A depth buffer with the same size of this slice, normally *Nx* by *Ny*, is used for recording its depth information. Second, a stencil buffer with the same size is used to generate a mask of the solid object clipped by the slice plane at that slice position. Third, in combination with one selected slice function, the interior information of the clipped solid object on the corresponding slice plane is generated dynamically. Thus, one solid slice is created with interior information specified by the slice function.

4.3 The Algorithm of Solid Slice Creation

The algorithm of solid slice creation is summarized in Figure 4. Conceptually, one depth buffer, one stencil buffer, and one color buffer are utilized to generate one solid slice at one time. Additionally, the generation of the final solid slice actually in the frame buffer avoids any copying after rendering. The entire algorithm can be fully implemented on the standard graphics system. If multiple or larger buffers are used, more solid slices can be created at the same time. Thus, the total creation speed will increase correspondingly.

```
Algorithm 1: Solid Slice Creation
CreateSolidSlice()
Input:
L ← a display list of a closed surface-based object
i ← the number of the ith slice which need to be voxelized into a solid slice
SolidSliceFunc() ← a specified solid slice function
Output:
S← the solid slice
Begin
1 Clear the color, stencil, and depth buffers
2 Initialize depth buffer of ith slice with the same size as the
   projection window
3 Use stencil buffer to toggle parity flag for front or back
surfaces by calling the display list L in front of ith slice
4 Apply the solid slice function SolidSliceFunc() to ith slice
into color buffer
5 Use stencil mask for parity check to mask out the solid slice
   5.1 If stencil buffer value is 1 then
           Corresponding portion of ith slice is updated by slice
         function in color buffer
   5.2 Else cleared to background color
       End of if
6 Get voxelized solid slice S from the frame buffer
7 Output the entire voxelized solid slice S
End
```

Figure 4: Algorithm of solid slice creation

4.4 The Algorithm of Real-time Solid Voxelization

The solid voxelization algorithm proceeds by moving a slice plane, which is parallel to the projection plane, with a constant step size in volume space. For each new slice, if it is inside the bounding box of the solid geometric object, a new solid slice is created by calling *CreateSolidSlice()*. Then it is directly stored into a given 3D texture volume without taking time to do swapping between GPU and main memory. In addition, This algorithm can do voxelization in an arbitrary orientation.

The algorithm of real-time solid voxelization for arbitrary orientation is summarized in Figure 5.

```
Algorithm 2: Realtime Solid Voxelization
SolidVoxelization()
Input:
O ← a closed surface-based object
Nx, Ny, Nz ← size of the output volume
Bmin, Bmax ← minimum and maximum 3D vectors (x, y, z)
of the bounding box
SolidSliceFunc() ← a specified solid slice function
Output:
V← the voxelized solid volume
Begin
1 Set the viewport (0, 0, Nx, Ny)
2 Set orthogonal projection
3 Set background color to be (0, 0, 0, 0)
4 Do transformation for the object, O, to make its orientation
   specified for voxelization consistent with Z axis.  Generate
   the display list L for the geometric object O drawing.
5 Initialize or preprocess the slice function SolidSliceFunc()
6 For each slice i = 0 to Nz – 1 do
   6.1 If slice i insides the bounding box (Bmin, Bmax) then
         6.1.1 ith solidSlice ← CreateSolidSlice(L, i,
SolidSliceFunc())
         6.1.2 Insert ith solidSlice into the 3D texture volume V
         End of if
   End of for
7 Output the entire volume V
 End
```

Figure 5: Algorithm of realtime solid voxelization

5 Experimental Results

Some experimental image results of the algorithm of real-time solid voxelization described in this paper are presented in Figures 1, 6, and 7. The time spent to create models at different resolutions and sizes are listed in Table 2 and Table 3.

All tests were run on a Dell PC with Pentium (R) D a single processor 3.0GHZ and 2GB RAM equipped with an Nvidia GeFore 7800 GTX (8-bit stencil buffer, 24-bit depth buffer, 256MB texture memory, and 3D texture mapping supported). The program is written in Microsoft VC++, OpenGL 2.0.3, and Cg. The operating system is Winsows XP. All volumetric data models were rendered using 3D texture mapping.

5.1 The Image Results of Solid Voxelization

In Figure 1, for the dragon model, (a) and (c) are the solid voxelization results by using the solid slice function FUNC_COLORFUL_CHECKER and FUNC_COLOR_RAMP defined in Table 1. For the happy Buddha model, (b) is the solid voxelization result by using the slice function of semi-translucent jade.

In Figure 6, (a) shows the dragon model is voxelized using the slice function FUNC_COLOR_RAMP. However, the voxelization orientation and color settings are different from Figure 1 (c). The orientation is from the dragon head to its tail while Figure 1 (c) is from the left side to the right side. Figure 6 (b) uses FUNC_INTERPOLATED_COLOR. (c) and (d) are the cutting effects after voxelization using the slice function FUNC_NOISE_MARBLE and FUNC_NOISE_WOOD, respectively. All interior materials can be seen in the image results. They keep continuous in 3D space by using Perlin 3D noise during solid voxelization.

Likewise, in Figure 7, (a) uses FUNC_COLOR_RAMP. (b) is the solid voxelization result of one customized solid slice function to simulate the effect of semi-translucent jade. (c) is the back effect of (b). (d) uses the slice function FUNC_NOISE_ROCK. (e) is the cutting effect of (d). (f) uses the slice function FUNC_NOISE_MARBLE and (g) is the cutting effect of (f).

5.2 The Performance of Solid Voxelization

In Table 2 (a) and Table 3, different resolutions of the dragon and Buddha models were voxelized using FUNC_SINGLE_COLOR slice function into solid volumetric models by the size of 64^3, 128^3, 256^3, and 512^3, respectively. All the solid voxelization performances are less than 0.5ms, even for 1M polygons of Buddha model by the voxelization size 512^3. In Table 2 (b), different resolutions of the dragon models were voxelized using other color-based slice functions, FUNC_COLOR_RAMP and FUNC_INTERPOLATED_COLORS into solid volumetric models by the size of 128^3 and 256^3, respectively. All the solid voxelization performances are less than 0.5ms.

Table 2: Time spent of solid voxelization of the dragon model

Model	# of Vertices	# of Polygons	Time Spent			
			64^3	128^3	256^3	512^3
			1MB	8MB	64MB	512MB
			T_1	T_2	T_3	T_4
Dragon_res4	5,205	11,102	0.05ms	0.07ms	0.12ms	0.22ms
Dragon_res3	22,998	47,794	0.04ms	0.06ms	0.11ms	0.19ms
Dragon_res2	100,250	202,520	0.04ms	0.06ms	0.10ms	0.20ms
Dragon	437,645	871,414	0.05ms	0.07ms	0.12ms	0.33ms

(a) Solid voxelization using `FUNC_SINGLE_COLOR` slice function

Model	# of Vertices	# of Polygons	Time Spent			
			FUNC_COLOR_RAMP		FUNC_INTERPOLATED_COLORS	
			128^3	256^3	128^3	256^3
			8MB	64MB	8MB	64MB
Dragon_res3	22,998	47,794	0.07ms	0.12ms	0.09ms	0.13ms
Dragon	437,645	871,414	0.07ms	0.12ms	0.15ms	0.15ms

(b) Solid voxeliaztion using other color-based slice functions

Table 3: Time spent of solid voxelization of the buddha model

Model	# of Vertices	# of Polygons	Time Spent			
			64^3	128^3	256^3	512^3
			1MB	8MB	64MB	512MB
			T_1	T_2	T_3	T_4
Buddha_res4	7,108	15,536	0.04ms	0.06ms	0.10ms	0.17ms
Buddha_res3	32,328	47,794	0.04ms	0.06ms	0.10ms	0.18ms
Buddha_res2	144,647	293,232	0.04ms	0.06ms	0.10ms	0.18ms
Buddha	543,652	1,087,716	0.06ms	0.09ms	0.13ms	0.32ms

6 Conclusions and Future Work

In this paper, this new real-time solid voxelization approach markedly improves the voxelization speed for solid geometric objects in real time. In particular, by dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with many different kinds of interior materials or textures. This approach has been demonstrated to be effective, robust, flexible, and diverse for both convex and concave complex models. It is easy to implement and integrate this approach into various interactive applications, such as volume modeling, volumetric collision detection, medical simulation, volume animation, 3D printing, and computer art. However, there are still some limitations in the current approach. The computation of complex solid slice functions, such as Perlin noise, takes much more time than that of color-based functions. Like many other voxelization algorithms, anti-aliasing problem exists. The surface boundary of the volumetric object is not generated smoothly during the voxelization in both 2D and 3D spaces. Due to slice-based voxelization, the running time is influenced by the number of slices.

There are several improvements and directions for the future work. More efficient methods based on software and hardware acceleration may be explored to improve the computation for complex solid slice functions and quality (i.e. smoothness and accuracy) of the volumetric data during voxelization. Another interesting area of future research is to extend this approach to volume animation and volume deformation to explore the solid voxelization of time-varying input geometric model data in a progressive manner. Moreover, with increasing demands of real-time large volumetric data processing, many parallel and large-scale data volumetric rendering approaches have been studied intensively. However, very few parallel voxelization algorithms are investigated so far. Since the voxelization approach proposed in this paper is based on slice-independent processing, it is very suitable for parallelization. In the feature, parallel voxelization algorithms will be employed to further improve the performance of real-time voxelization for large-data processing.

References

BAERENTZEN , J. A., SRAMEK, M., AND CHRISTENSEN, N. J. 2000. A Morphological Approach to Voxelization of Solids. In *Proceedings Of 8th Central Europe on Computer Graphics, Visualization and Digital Interactive Media*, Pilsen, Czech republic, 44-51.

BREEN, D. E., MAUCH, S., AND WHITAKER, R. T. 1998. 3D Scan Conversion of CSG Models into Distance Volumes. In *Proceedings of IEEE/ACM Symposium on Volume Visualization*, 7-14.

CHEN, H. and FANG, S. 2000. Hardware Accelerated Voxelization. *Computer and Graphics* 24, 3, 433-442.

COHEN, D. and KAUFMAN, A. 1995. Fundamentals of Surface Voxelization. *CVGIP: Graphics Models and Image Processing*, 56 (6), 453-461.

DACHILLE, F. and KAUFMAN, A. 2000. Incremental Triangle Voxelization. In *Proceedings Of Graphics Interface 2000*, 205-212.

DANIELSSON, P.E. 1970. Incremental Curve Generation. *IEEE Transactions on Computers*. C-19, 80-87.

DONG Z, CHEN W., BAO, H., ZHANG, H. and PENG, Q. 2004. Real-time Voxelization for Complex Polygonal Models. In *Proceedings of Pacific Graphics 2004*, Seoul, Korea. 73-78.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., and WORLEY, S. 2003. *Texturing & Modeling: A Procedural Approach*, Third Edition (eries in Computer Graphics). The Morgan Kaufmann.

EISEMANN, E. and D`ECORET, X. 2006. Fast Scene Voxelization and Applications. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*. Redwood City, California.

FERNANDO, R. 2004. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. The Addison-Wesley.

HAUMONT, D. and WARZEE, N. 2002. Complete polygonal scene voxelization. *ACM Journal of Graphics Tools*, 7(3):27–41.

HUANG, J., YAGEL, R., FILIPPOV, V., and KURZION, Y. 1998. An Accurate Method for Voxelizing Polygon Meshes. In *Porc. IEEE/ACM Symposium on Volume Visualization*, 119-126.

HSIEH, H., LAI, Y., TAI, W., AND CHANG, S. 2005. A flexible 3D slicer for voxelization using graphics hardware. In *Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia GRAPHITE '05*. Dunedin, New Zealand. 285-288.

KARABASSI, E.A., PAPAIOANNOU, G., and THEOHARIS, T. 1999. A Fast Depth Buffer Based Voxelization Algorithm, *Journal of Graphics Tools*, ACM, 4(4), 5-10.

KAUFMAN, A. and SHIMONY, E. 1986. 3D Scan-conversion Algorithms for Voxel-based Graphics. In *Proceedings of 1986 Workshop on Interactive 3D Graphics*, 45-75.

KAUFMAN, A. 1987. Efficient Algorithms for 3D Scan-conversion of Parametric Curves, Surfaces, and Volumes. In *SIGGRAPH'87*, volume 21, 171-179.

KAUFMAN, A. 1988. Efficient Algorithms for Scan-converting 3D Polygons. *Computers and Graphics*, 12(2):213-219.

KAUFMAN, A. 1991. Volume Visualization. *IEEE Computer Society Press*.

LEE, Y. T. and REQUICHA, A. A. G. 1982. Algorithms for Computing the Volume and Other Integral Properties of Solids. *Communications of the ACM,* 25(9):635-650.

LIAO, D. 2001. *Volume Fusion*. M.S. Thesis, Purdue University.

LIAO, D. and FANG S. 2002. Fast volumetric CSG Modeling Using Standard Graphics System. In *Proceedings of ACM Symposium on Solid Modeling and Application*, 204-211.

MOKRZYCKI, W. 1988. Algorithms of Discretization of Algebraic Spatial Curves on Homogeneous Cubical Grids. *Computers & Graphics*, 12(3/4) 477-487.

PASSALIS, G., KAKADIARIS, I.A., and THEOHARIS, T. 2004. Efficient hardware voxelization. In *Proceedings of Computer Graphics International 2004 (CGI'04)*. Volume , Issue , 16-19, 374-377.

PERLIN, K. 2002. Improving noise. In *Proceedings of the 29th ACM Annual Conference on Computer Graphics and interactive Techniques (SIGGRAPH '02)*. San Antonio, Texas, 681-682.

PHARR, M., FERNANDO, R. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. The Addison-Wesley.

SRAMEK, M. and KAUFMAN, A. 1999. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics* 3(5), 251-266.

VARADHAN, G., KRISHNAN, S., KIM, Y. J., Diggavi, S., and Manocha, D. 2003. Efficient max-norm distance computation and reliable voxelization. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing,* 116–126. Eurographics Association.

WANG, S. and KAUFMAN, A. 2003. Volume Sampled Voxelization of Geometric Primitives. In *Proceedings Of IEEE Visualization'93*, 78-84.

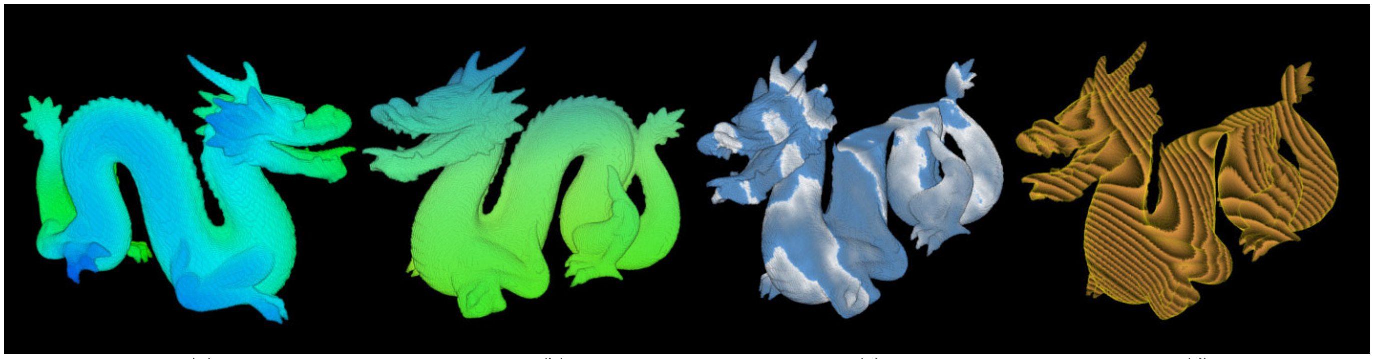

Fig. 6. Solid voxelization results (256^3) of the dragon model with different slice functions. (a) is a volumetric result with `FUNC_COLOR_RAMP`, (b) is a volumetric result with `FUNC_INTERPOLATED_COLOR`, (c) is a cut volumetric result with `FUNC_NOISE_MARBLE`, and (d) is a cut volumetric result with `FUNC_NOISE_WOOD`.

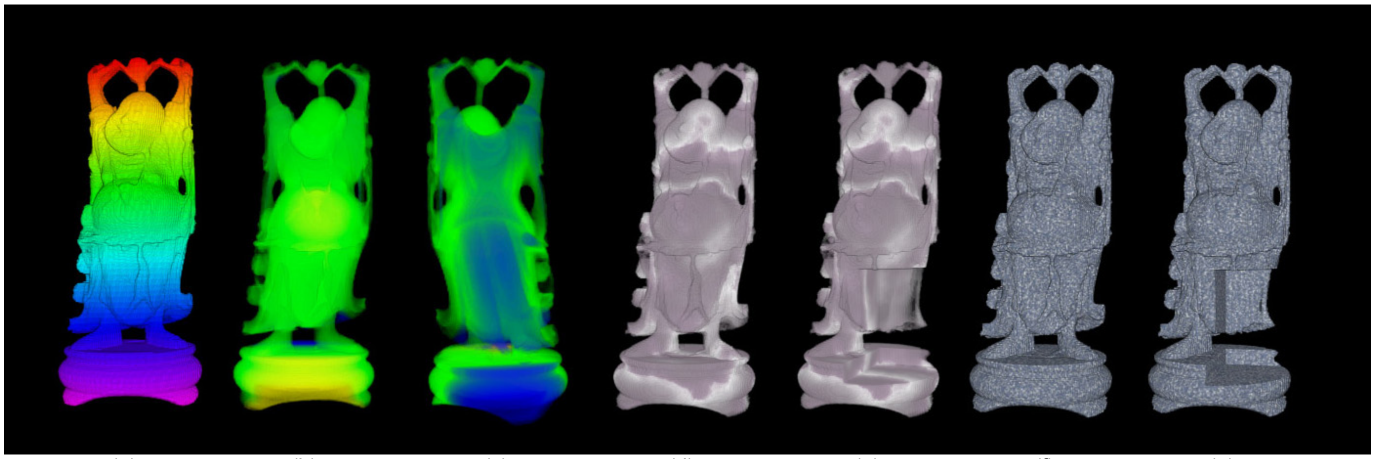

Fig. 7. Solid voxelization results (256^3) of the happy Buddha model with different slice functions. (a) is a volumetric result with `FUNC_COLOR_RAMP`, (b) is a volumetric result with one customized slice function of semi-translucent jade, (c) is a back effect of (b), (d) is a volumetric result with `FUNC_NOISE_MARBLE`, (e) is a cut volumetric result of (d), (f) is a volumetric result with `FUNC_NOISE_ROCK`, and (g) is a cut volumetric result of (f).