General voxelization algorithm with scalable GPU implementation

GEORGE PASSALIS[1]
GEORGE TODERICI[2]
THEOHARIS THEOHARIS[1]
IOANNIS A. KAKADIARIS[2]
[1]Department of Informatics and Telecommunications, University of Athens
[2]Computational Biomedicine Lab, University of Houston

March 1, 2006

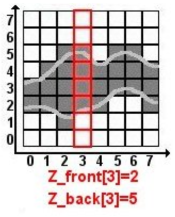
Abstract

An efficient algorithm for the voxelization of solid objects represented by boundary surfaces of arbitrary topology is presented. The algorithm is based on the use of depth buffers, and is suitable for polygonal meshes and parametric surfaces. The presented method unifies and extends our previous approaches under a single general algorithm; three practical variants are then given that implement it on different generations of graphics hardware in order to achieve high performance. The implementations for older hardware impose certain limitations on the depth complexity of the object.

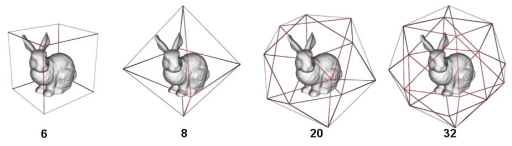
1 Introduction

Surfaces and voxels constitute the two main classes of object representation in computer graphics and visualization. Volume representations are used mainly where 3D regularly sampled data are available (e.g., medical imaging, constructive solid geometry, seismic data). Additionally, they can be used as an alternative representation for geometric objects that are defined using any surface representation (e.g., polygonal, parametric). This requires a process that will allow us to convert surface data to voxels. The process of *voxelization* produces a set of values on a regular three dimensional grid and allows methods that work with regularly sampled data to be applied to objects defined using surface representations. Therefore voxelization allows us to model and manipulate geometric objects in a way that in certain tasks (e.g, boolean operations) is more efficient than with their original surface representation.

The goal of the voxelization process is to produce a volumetric grid that approximates as closely as possible the original object. Depending on whether the voxels store values of 0 and 1 only or real values in $[0,1]$, voxelization algorithms are divided into



(a)



(b)

Figure 1: (a) 2D Case: An object defined by two curves. (b) Multiple direction z-buffer configurations.

binary and non-binary. An important issue in voxel models is adjacency; depending on the type of neighbours that a voxel is allowed to have (by face, edge or vertex) we can have 6-, 18- or 26-adjacent voxel models. Since voxelization is an integral part of voxel graphics [8], a number of voxelization approaches have been proposed. These approaches [4, 3, 2, 13], allow any geometric object to be voxelized with varying properties (e.g., binary/non-binary, variable surface thickness), but do not exploit graphics hardware to achieve efficiency.

In recent years the development of widely available graphics hardware has far outstripped that of general purpose microprocessors. Although graphics hardware is designed for surface rendering, it is often possible to utilize it in different tasks. In this paper we show that the process of voxelization is one such task and by combining and extending our previous work we present a general binary voxelization algorithm that exploits widely available graphics hardware.

Our algorithm is based on combining the depth buffers (or z-buffers) of an object. The z-buffer has been shown to have applicability in a wide range of tasks [14]. In contrast to slicing methods ([6, 1]) the number of the required z-buffers is not correlated with the voxel resolution. We subsequently present three different implementations of the general algorithm that target different generations of graphics hardware. The descriptive power of the implementations scale with the capabilities of the hardware; the last implementation can handle any object of arbitrary complexity.

2 General Voxelization Method

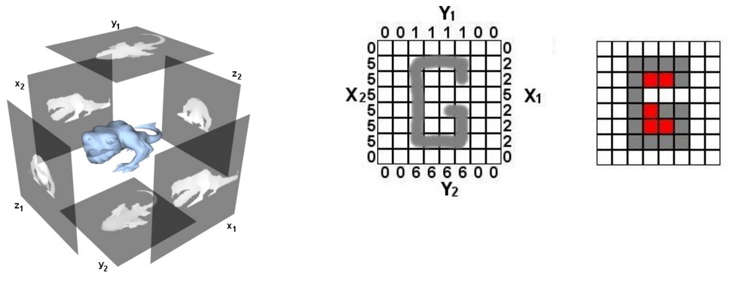
A 3D point $\bar{v}$ is outside a closed surface object, if any ray from $\bar{v}$ to infinity intersects the object an even number of times. If we render an object using an orthographic projection, we can define rays parallel to the direction of projection starting at each image pixel. We can use the z-buffer as a 2D array that stores the values of the nearest object intersection per pixel (i.e. per ray). If, for a certain direction of projection, we have enough z-buffers to store all surfaces of the object (i.e. a number equal to the object's depth complexity) then we can use the above test to determine which voxels are inside the surface object and which are not.

Z-buffers are actually computed in pairs (by reversing the front-back face definition). One element of the pair describes the 'front' of the object while the other describes the 'back'. This is demonstrated in Fig. 1 (a) for a simple 2D case. Notice that for column 3 the front z-buffer has value 2 and the back z-buffer has value 5. Hence, in column 3, only the voxels of rows 2, 3, 4 and 5 belong to the object. Objects with higher depth complexity require more pairs of z-buffers in order to be voxelized correctly. These can be acquired using an iterative process: We first obtain the front and back buffers with the minimum depth, and then we repeat this process by rendering only triangles that have greater depth than the z-buffers of the previous step. Thus we get depth-sorted z-buffer pairs, with each pair describing a different layer of the object. The number of iterations is determined by the depth complexity of the object, which is the maximum number of ray-object intersections that can occur along this direction.

To avoid precision problems caused by triangles parallel to the direction of projection, we repeat the iterative process from different orientations. Typically we select three orthocanonical directions but in some implementations a greater number can be utilized (Fig. 1 (b)). For each direction, the voxel needs to be inside *at least one* z-buffer pair. In order to check if the voxel's center is inside or outside a z-buffer pair (Z_1^d, Z_2^d) taken from an arbitrary direction d, we need to project the point ontto the image space of the z-buffers. This is achieved using the same modelview ($\mathbf{M}^d$), projection ($\mathbf{P}^d$) and viewport ($\mathbf{V}^d$) matrices that were used for the creation of the z-buffer pair. The x and y components of the projected point are used as u and v in image space, while the z component is compared with the values stored in the z-buffers, after it's normalized inside $[0,1]$. The pseudocode for this is the following (ND is the number of directions, ZP is the number of z-buffer pairs per direction):

```
1. For each direction $d = 1..ND$ compute all z-buffer pairs $\{Z_1^d ... Z_{2ZP}^d\}$
   (the pairs are depth-sorted)
2. For each voxel $V_{(i,j,k)}$ (with center at $\bar{c}_{(i,j,k)}$) do
   • Set $DirectionCheck^{1..ND} = false$
   • For each direction $d = 1..ND$ do
     – Project voxel center: $\bar{v}' = \mathbf{V}^d \cdot \mathbf{P}^d \cdot \mathbf{M}^d \cdot \bar{c}_{(i,j,k)}$
     – For each z-buffer pair $n = 1..ZP$ do
       * If $(\bar{v}'_z \geq Z_{2n}^d(\bar{v}'_x, \bar{v}'_y)$ and $\bar{v}'_z \leq Z_{2n-1}^d(\bar{v}'_x, \bar{v}'_y))$ set
         $DirectionCheck^d = true$
   • If all $ND$ checks are true set $V_{(i,j,k)} = 1$ else set $V_{(i,j,k)} = 0$
```

The inherent advantage of this method is that the number of iterations is correlated with the depth complexity of the object and not with the voxel resolution as in slicing methods, resulting in higher efficiency. It can handle objects of arbitrary depth complexity with the only limitation that these objects have a strictly closed surface. Moreover, it allows a scalable hardware implementation as described in the following



(a) (b) (c)

Figure 2: (a) ’Monster’ object with six corresponding z-buffes. (b) The letter ’G’ with four 1D z-buffers (X_1, X_2, Y_1, Y_2). (c) Erroneous voxelization of ’G’ (marked with red).

section. By changing the *if* statement in the inner loop, the algorithm can produce a 26-adjacent description of the object’s surface.

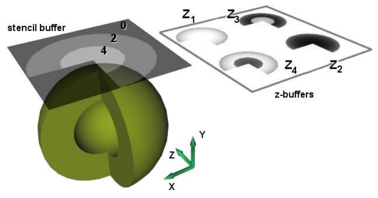
3 GPU implementations of the voxelization method

The hardware accelerated implementations use the OpenGL API [15]. OpenGL can render polygonal meshes and analytical surfaces (like NURBS) and since analytical surfaces are converted internally to a triangular mesh, they also produce a z-buffer, making the voxelization algorithm invariant to the initial representation of the object.

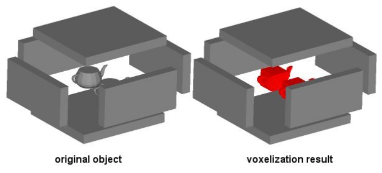
3.1 Implementation using six single layer z-buffers

The algorithm presented by Karabassi et al [7] is the simplest instantatiation of the general algorithm; it only requires hardware capable of computing the z-buffer (e.g., Nvidia TNT series). In total, six z-buffers are rendered for each object, two per axis, as shown in Fig. 2 (a). The camera is placed on each of the six faces of a conceptual bounding box of the object and the z-buffers are computed using orthographic projection. The simplified pseudocode for this implementation is derived from the generalized pseudocode if we set $ND = 3$ and $ZP = 1$:

```
1. Compute the 6 z-buffers ($Z_1^x, Z_2^x, Z_1^y, Z_2^y, Z_1^z, Z_2^z$)
2. For each voxel $V_{(i,j,k)}$ do
    • Check if ($i \geq Z_1^x(k, j)$ and $i \leq Z_2^x(k, j)$)
    • Check if ($j \geq Z_1^y(i, k)$ and $j \leq Z_2^y(i, k)$)
    • Check if ($k \geq Z_1^z(i, j)$ and $k \leq Z_2^z(i, j)$)
    • If all checks are true set $V_{(i,j,k)} = 1$ else set $V_{(i,j,k)} = 0$
```



(a)



(b)

Figure 3: (a) Object with stencil and depth buffers in the Y direction. (b) Object voxelization with undetermined areas marked in red.

Karabassi's algorithm can handle both convex and a limited set of concave objects. The limitation arises from the fact that certain surface details (like concavities) may not be visible from any of the six directions; it is not possible to voxelize such parts of the object correctly. As seen in the simple 2D case of Fig. 2 (b) and (c) the letter 'G' was wrongly voxelized because the concavities was not visible from any direction. Furthermore there is no way to tell whether a object has been correctly voxelized.

Even though this implementation suffers from the aforementioned limitation, it is more descriptive than similar methods such as the one proposed by Prakash et al [12]. Prakash uses a pair of z-buffers in order to voxelize a *convex* object, but the object must represent unstructured grid cells for the voxelization to work properly.

3.2 Implementation using multiple double layer buffers

We recently presented an algorithm [11] that takes advantage of additional GPU capabilities, like the stencil buffer, to alleviate the problems associated with the previous implementation while maintaining wide applicability. We utilize the multiple buffer configuration described in section 2. Since we need only one direction with clear line of sight to determine if a voxel is inside the object, the set of concave objects that can be handled increases with the number of buffers. There is a trade-off between accuracy and performance since the number of z-buffers increases the computational cost.

Additionally, we use double layer buffering which can be implemented in graphics hardware by exploiting the method used for culling back faces. By defining either the clockwise or counter-clockwise triangles as front facing, it is possible to acquire a total of four z-buffers from each direction instead of two. This is equivalent to two iterations of the general algorithm and is shown in Fig. 3 (a) where the inner sphere boundary is visible in the back facing z-buffers $Z3$ and $Z4$.

Double layer buffering requires a way of determining the number of actual surface intersections per ray/voxel; if it is 2 we use just the outer z-buffers, if it is 4 we use the double layer. We use the stencil buffer to count the number of times the z-buffer is assigned a new value at each pixel. Note that for closed objects, stencil values should be even. The stencil buffer has an additional use: it marks voxels as unclassifiable if they have stencil values greater than 4 in all directions. This is depicted in Fig. 3 (b) where the object was voxelized using six double layered z-buffers.

3.3 Implementation using programable GPUs

Modern GPUs (e.g., Nvidia GeForce 6 series) are fully programmable on a per fragment or per vertex basis. We can utilize this flexible feature (using the CG language [10]) to fully implement the method presented in Section 2. As explained there, we simultaneously need two z-buffers, used in an iterative manner to process an arbitrary object. Since no currently available GPU offers two z-buffers, we make use of the exposed programmability to simulate the second z-buffer by storing the depth information from the previous iteration on a texture. The depth texture acts as a second z-buffer test that takes place before the actual z-buffer test. If the current depth is lower (that is, we are rendering a point that is closer than the value stored in the texture), then the associated pixel fails the depth test. The iteration of this process results in depth-sorted z-buffer pairs. The pseudocode of the generalized algorithm changes only in step 1; step 2 remains unmodified. Step 1 is implemented as follows (with $ND = 3$):

```
For each direction $d = 1..ND$ do
    • Compute maximum stencil value and set $ZP = \frac{maxstencil}{2}$
    • Compute $Z_1^d$ and $Z_2^d$ with default OpenGL rendering
    • Set the $FrontTex = Z_2^d$ and $BackTex = Z_1^d$
    • For each $n = 2..ZP$ do
        1. Enable fragment program and pass as parameters the textures
           $FrontTex$, $BackTex$
        2. Render the front-facing faces and store the z-buffer in $Z_{2n}^d$
        3. Render the back-facing faces and store the z-buffer in $Z_{2n-1}^d$
        4. Set $FrontTex = max(Z_{2n}^d, FrontTex)$ and $BackTex =
           max(Z_{2n-1}^d, BackTex)$
```

Note that for accuracy reasons it is preferable to use floating point textures to store the z-buffers from previous iterations. In recent GPU's, 32bit float textures can be used to store the depth component which minimizes precision related artifacts. Fan, Li et al [9] presented a GPU-based voxelization algorithm that utilizes the depth peeling technique (originally proposed by Everitt [5] for order-independent rendering of transparent objects). Compared to our implementation it shares the same principles but the advanced GPU capabilities are utilized in a different manner.

4 Results

We performed extensive tests for all three implementations of our method using a variety of objects. Note that the stencil buffer can be used before the actual voxelization

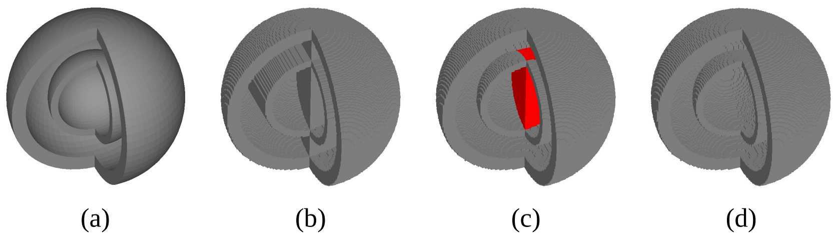


Figure 4: Voxelization of two homocentric hollow spheres: (a) Original object (b) First implementation (certain areas were voxelized erroneously); (c) Second implementation (using the stencil we mark possible areas of error in red); (d) Third implementation.

to determine which implementation should be used. In Fig. 4 we show a synthetic object that demonstrates the limitations of the first two implementations. Fig. 5 shows the effect of the choice of voxel resolution. Our algorithm works with any resolution so long as it is bellow the maximum screen resolution supported by the GPU. Modern GPUs support up to 2048x2048, allowing for very detailed voxelizations. Generally, the optimum resolution is determined by the nature of the application.

Fig. 6 examines the correlation between computational time and voxel resolution. We used two objects with complexity 70K and 15K triangles and tested the three implementations with voxel resolutions ranging from 64x64x64 to 512x512x512. It can be seen that performance is largely independent of object complexity, an inherited characteristic of the z-buffer. With respect to voxel resolution n, the method's first part (z-buffer acquisition) has $O(n^2)$ complexity, while the second part (voxel grid computation) has $O(n^3)$.

Note that of these two parts, only the first can be performed on the GPU. A performance comparison for this part between GPU and CPU is depicted in Fig. 7 (a). The GPU is an order of magnitude faster than the CPU for the same task, even though the GPU times include the transfer of the z-buffers to main memory. To perform an ”end-to-end” comparison between the GPU and CPU, the cost of the second part (computed using the CPU) was added, and the total times are reported in Fig. 7 (b). Again, the use of the GPU offers a significant performance improvement. All experiments were performed on a 3Ghz Pentium 4 with 1GB RAM and a GeForce 6600GT.

A limitation of the presented method is that the surface of the object must be strictly closed with no irregular faces (e.g., crossing, duplicate). Synthetic objects usually don't suffer from such problems but not all real life objects are ideally triangulated. An indication that such problems exist is the appearance of odd values in the stencil buffer. We also noticed that in objects with large numbers of triangles perpendicular to the z-buffers' plane certain artifacts may appear, especially at lower voxel resolutions. This is attributed to the non-linear accuracy of the z-buffer's internal format.

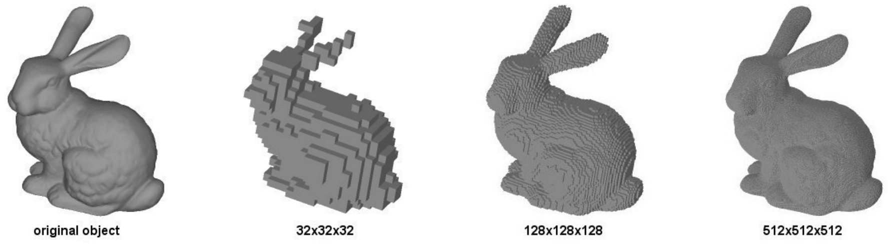
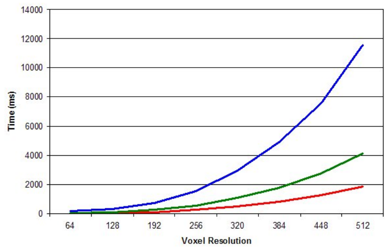
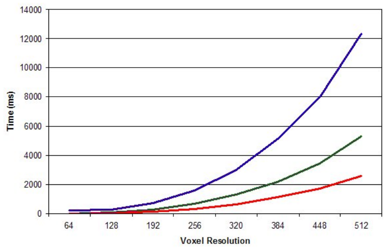


Figure 5: Object ’bunny’ voxelized in different resolutions.

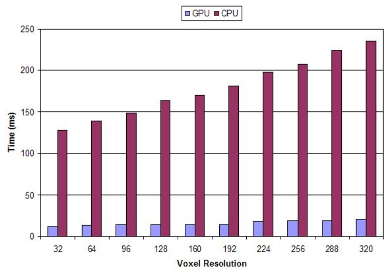


(a)

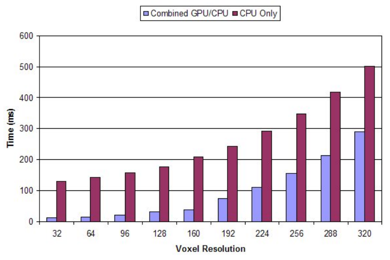


(b)

Figure 6: Performance for different voxel resolutions (red for the first implementation, green for the second, blue for the third) using different objects (a) ’Bunny’ (70k triangles); (b) Double hollow sphere (15K triangles).



(a)



(b)

Figure 7: Performance using the first implementation and the Bunny object: (a) First part (z-buffer acquisition): GPU vs CPU; (b) First plus second part (voxel grid computation): Combined GPU/CPU vs CPU only.

References

[1] H. Chen and S.Fang. Fast voxelization of three-dimensional synthetic objects. *Journal of Graphic Tools*, 3(4):33–45, 1998.

[2] D. Cohen and A.Kaufman. 3d line voxelization and connectivity control. *CGAA*, 17(3):80–87, 1997.

[3] D. Cohen and A. Kaufman. Fundamentals of surface voxelization. *Graphical Models and Image Processing*, 57(6):453–461, November 1995.

[4] D. Cohen, A. Kaufman, and Y. Wang. Generating a smooth voxel-based model from an irregular polygon mesh. *The Visual Computer*, 10(6):295–305, 1994.

[5] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001.

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

[7] E.A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *ACM Journal of Graphics Tools*, 4(4):5–10, 1999.

[8] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[9] W. Li, Z. Fan, X. Wei, and A. Kaufman. *GPU Gems II, Simulation with Complex Boundaries*, chapter 47. Addison Wesley, 2005.

[10] Nvidia. developer.nvidia.com.

[11] G. Passalis, I.A. Kakadiaris, and T. Theoharis. Efficient hardware voxelization. In *Proceedings of Computer Graphics International*, Crete, Greece, June 2004.

[12] C.E. Prakash and S. Manohar. Volume rendering of unstructured grids - a voxelization approach. *Computer Graphics*, 19(5):711–726, 1995.

[13] M. Sramek and A. Kaufman. Antialiased voxelization of geometric objects. *TVCG*, 5(3):251–266, 1999.

[14] T. Theoharis, G. Papaioannou, and E.A. Karabassi. The magic of the z-buffer: A survey. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 379–386, Pilsen, CZ, February 2001.

[15] R.S. Wright and M. Sweet. *OpenGL SuperBible*. Waite Group Press, 1999.