Efficient Hardware Voxelization

G. Passalis, I.A. Kakadiaris
Visual Computing Lab
University of Houston
{gpassali,ioannisk}@mail.uh.edu

T. Theoharis
Department of Informatics
University of Athens
theotheo@di.uoa.gr

Abstract

This paper presentes a novel algorithm for the voxelization of surface models of arbitrary topology. Our algorithm uses the depth and stencil buffers, available in most commercial graphics hardware, to achieve high performance. It is suitable for both polygonal meshes and parametric surfaces. Experiments highlight the advantages and limitations of our approach.

1. Introduction

Surface and volumetric models constitute the two main classes of objects in computer graphics. Volume models are used mainly where volumetric data are available (e.g., medical imaging, CSG, seismic data). However, surface models are far more efficient to display and store. Often it is useful to convert surface data to voxels so that methods that work with regularly sampled data can be applied. This process of *voxelization* produces a set of values on a regular three dimensional grid.

In recent years the development of widely available graphics hardware has been staggering; in fact the rate of its development has far outstripped that of microprocessors. Although graphics hardware is designed for surface rendering, it is often possible to utilize it in different tasks. In this paper we show how it can be used for voxelization. Future graphics hardware will probably be able to accelerate even the rendering of voxel models (by combining 3D textures and pixel shaders).

1.1. Related Work

The majority of previous approaches in voxelization do not exploit graphics hardware to achieve efficiency. For example, Cohen et al [3] proposed such a method that can be applied to irregular polygon meshes and He et al [5] used voxels for object simplification purposes. In contrast, Fang and Chen [4, 2] proposed a voxelization method that uses graphics hardware. They slice the surface object into slices and for each slice they render the object using the slicing plane as a clipping plane. Finally they reconstruct the 3D data by combining the data from the slices. Even though this operation is done using graphics hardware, the computational time is proportional to the number of slices which is large for complex objects.

One component of graphics cards that has been shown to have applicability in a wide range of tasks [10] is the depth buffer (or z-buffer), which was originally designed for hidden surface elimination in polygonal model rendering. The long list of tasks include volume estimation [8], symmetry detection and surface registration [1, 7]. Prakash et al [9] presented a method that uses a pair of z-buffers in order to voxelize a *convex* object. The limitation is that the object must represent unstructured grid cells for the voxelization to work properly. The use of just two z-buffers results in voxelizing only the part of the surface that is visible to them.

Karabassi et al [6] proposed the use of six z-buffers. This algorithm is the most efficient to date; it is simple to implement and only requires to render the object six times. It can be applied to objects of arbitrary topology and complexity. The main disadvantage is that it can miss concavities in non-convex objects thus producing erroneous results. This limits the usefulness of the algorithm as it can be applied only to a restricted subset of closed objects. Furthermore, it does not indicate whether an object has been correctly voxelized.

1.2. Overview

In this paper we improve on the algorithm proposed by Karabassi et al [6] by generalizing it so that it can be applied to a wider range of objects without sacrificing performance. The goal is to make the algorithm general enough to handle most common objects, thus limiting the need for complex and slow voxelization algorithms for special case objects. Our algorithm actually detects (rare) cases where the task of voxelization may have been incorrectly performed.

The rest of the paper is structured as follows: Section 2 reviews Karabassi's original algorithm, Section 3 describes our algorithm, and Section 4 summarizes the results.

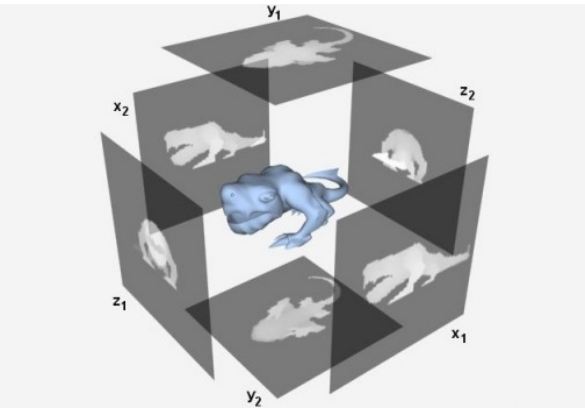


Figure 1. Six z-buffers for a 3D object.

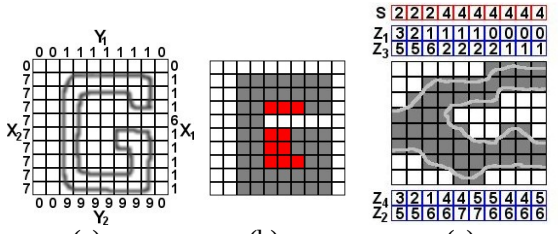
2. Voxelization using six single layer z-buffers

The algorithm by Karabassi et al [6] voxelizes objects by extracting their z-buffers using OpenGL [11]. OpenGL can render polygonal meshes and analytical surfaces. Since analytical surfaces are converted internally by OpenGL to a triangular mesh and rendered using the normal graphics pipeline, they also produce a z-buffer making the voxelization algorithm invariant to the initial representation of the object. Therefore the voxelization process can be separated from the computation of the z-buffer.

In total six z-buffers are rendered for each object, two per axis, as shown in Fig. 1. The camera is placed on each of the six faces of a computed bounding box of the object and the z-buffers are taken using parallel projection. For each pair of opposite directions (e.g., +X, -X), the values of the two associated z-buffers provide a regular sampling of the object's surface from the 'front' and 'back' views. A voxel is inside the object only if it is 'inside' the z-buffer values for all three pairs. The pseudo-code follows:

- Compute the 6 z-buffers ($X_1, X_2, Y_1, Y_2, Z_1, Z_2$)
- For each $voxel(i, j, k)$ do
  1. Check if ($i \geq X_1(k, j)$ and $i \leq X_2(k, j)$)
  2. Check if ($j \geq Y_1(i, k)$ and $j \leq Y_2(i, k)$)
  3. Check if ($k \geq Z_1(i, j)$ and $k \leq Z_2(i, j)$)
  4. If all checks are true set $voxel(i, j, k) = 1$ else set $voxel(i, j, k) = 0$

This algorithm assumes that the object surface is closed: a ray in any direction from every point inside the object must intersect the object's surface an odd number of times. Karabassi's algorithm can handle convex and a subset of concave objects. The limitation lies in 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 Fig. 2 (a),(b) the letter 'G' was voxelized incorrectly because the concavity is not visible from any direction. Furthermore it is not easy to tell whether a concave object has been correctly voxelized.



(a) (b) (c)

Figure 2. (a) Letter 'G' and its z-buffers (X_1, X_2, Y_1, Y_2). (b) Erroneous voxels marked in red. (c) Double z-buffers and stencil buffer.

3. The Proposed Voxelization Algorithm

Our goal is a voxelization algorithm that alleviates the above problems while using only standard graphics hardware, so as to maintain performance and applicability. The fundamental flaw in Karabassi's algorithm is that it can not correctly classify a voxel if any of the three lines that pass from this voxel and are parallel to each of the three axes (X, Y, Z) intersect the object surface more than two times.

We propose two improvements to Karabassi's algorithm that significantly increase the range of objects that are correctly handled. The first is an obvious generalization: the use of an arbitrary configuration of more than six z-buffers. The second is the combined use the z-buffer and the stencil buffer which can handle up to one concavity per direction. Furthermore, our approach detects object cases that may have been voxelized incorrectly.

3.1. Multiple z-buffers

Some concavities are not visible from any of the six main directions but may be clearly visible from some others (Fig. 5 (a)). We generally need only one direction with clear line of sight to determine if a voxel is inside the object. Changing the configuration of the six z-buffers and replacing them with an arbitrary number can improve the voxelization result.

To generalize the orientation of the z-buffers we will use the equation of the plane that defines a z-buffer. Such a plane is defined by three points $\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$. The plane equation is $\overrightarrow{a} * x + \overrightarrow{b} * y + \overrightarrow{c} * z + \overrightarrow{d} = 0 \Longleftrightarrow \overrightarrow{d} = -(\overrightarrow{a} * a_x + \overrightarrow{b} * a_y + \overrightarrow{c} * a_z)$ and the parametric plane equation is $P(u, v) = \overrightarrow{a} + (\overrightarrow{b} - \overrightarrow{a}) * u + (\overrightarrow{c} - \overrightarrow{a}) * v$. The normal to the plane is computed as $\overrightarrow{n} = \frac{(\overrightarrow{a} - \overrightarrow{b}) \times (\overrightarrow{b} - \overrightarrow{c})}{|(\overrightarrow{a} - \overrightarrow{b}) \times (\overrightarrow{b} - \overrightarrow{c})|}$.

We next need the projection of a given point V (the voxel) onto the plane. This is computed as the intersection of the plane with a line that starts from V and has the direction of the plane's normal. The line equation is given by $L(\omega) = V - \overrightarrow{n} * \omega$. Solving the plane and line equations results to the point of intersection of the line and the plane,

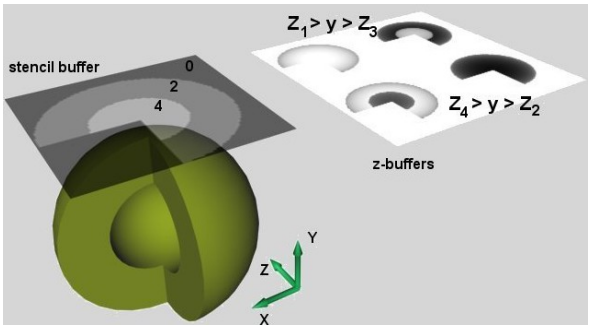


Figure 3. Stencil and z-buffers for Y direction.

and by using it in the parametric plane equation we obtain its u, v parameters. The u, v parameters can then be used to obtain a value from the z-buffer. The pseudo code follows:

- Compute z-buffer pairs $\{Z_{l1}, Z_{l2}\}$ for each direction $l = 0..N-1$
- For each $voxel(i, j, k)$ do
  - For each pair of buffers $\{Z_{l1}, Z_{l2}\}$ do
    1. Compute intersections of line $L_{voxel(i,j,k)}(\omega)$ with Z_{l1} and Z_{l2} ((u_{l1}, v_{l1}) and (u_{l2}, v_{l2}))
    2. Compute distances d_{l1} and d_{l2} from V to the two z-buffer planes
    3. Check if ($d_{l1} \geq Z_{l1}(u_{l1}, v_{l1})$ and $d_{l2} \leq Z_{l2}(u_{l2}, v_{l2})$)
  - If all N checks are true set $voxel(i, j, k) = 1$ else set $voxel(i, j, k) = 0$

There is a trade-off between accuracy and performance. As the number of z-buffers increases the algorithm becomes slower. We do not consider the use of a large number of z-buffers to be an efficient approach. Usually 8 to 16 z-buffers in combination with the double layer approach are enough.

3.2. Double layer buffers

For most concave real life objects there are rays that intersect the object's surface more than twice, in any given direction. Thus some voxels cannot be correctly classified by the above approach. The double layer approach addresses this problem by moving the threshold beyond which the problem appears, from two to four intersections. Even though the number of z-buffers needed to handle the general case is not finite, in practice this approach handles most cases, minimizing the limitation of Karabassi's algorithm.

In Fig. 2 (c) the double layer approach is demonstrated for the two dimensional case. For each direction, we have an inner and an outer z-buffer, hence the name double layer z-buffer. The object boundary forms a concavity that is not visible by the two outer z-buffers Z_1, Z_2, but can be detected using the inner layer z-buffers Z_3, Z_4. Instead of checking if the voxel is between Z_1 and Z_2 we now check between Z_1 and Z_3 and then between Z_4 and Z_2.

Double layer buffering can be implemented in graphics hardware by exploiting the method used for culling back faces. Because we can define either the clockwise or counter-clockwise triangles as front facing, it is possible to render the object twice from each direction (e.g., +Y) obtaining a total of four z-buffers per direction pair (e.g., +Y, -Y). This is depicted in Fig. 3 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 intersections per ray/voxel; if it is 2 we use just the outer z-buffers, if it is 4 we use the double layer; if it is greater than 4 we can mark this voxel as unclassifiable. For this purpose, we can use the stencil buffer which is standard in modern graphics hardware, mainly because of its use in the shadow volume technique. We use the stencil to count the number of times the z-buffer is assigned a new value at each pixel. To that end, we render the object an extra time after disabling the z-test (always true) so that all triangles are rendered. Every time the z-buffer is updated the corresponding pixel value in the stencil buffer is incremented. The stencil values for closed objects should be multiples of two. This is shown for the two dimensional case in Fig. 2 (c) and for the three dimensional in Fig. 3.

Furthermore, the stencil buffer provides some extremely useful information. If a voxel has stencil values greater than four in any direction pair, we can mark it as unclassifiable. We can thus identify exactly all unclassifiable voxels and then decide on the success of the voxelization process using our algorithm, or whether a slow and more accurate method should be applied to the object. This is demonstrated in Fig. 5 (d): we intentionally surrounded the teapots with boxes in all directions so that the algorithm would not be able to voxelize the teapots. The algorithm marked the teapots as unclassifiable (shown in red). Note that even in this case the result is acceptable since the user can decide if the unclassifiable area belongs to the object.

4. Results

We tested our algorithm on a number of different objects and compared it to Karabassi's original algorithm. Note that our algorithm handles all objects handled by that algorithm plus classes of objects that previously could not be voxelized correctly. The cost for this is an extra z-buffer per direction and a stencil buffer per pair of opposite directions. Fig. 5 shows the application of the two algorithms to various objects. Case (a) can be handled either by the multiple z-buffers or the double layer z-buffer approaches. Cases (b), (c) require the use of double layer z-buffers (in (c) the our algorithm found the cavity in the monster's mouth).

Fig. 4 (a) examines the correlation between computational time and voxel/object resolutions. The algorithm handles object complexity very well since the computation

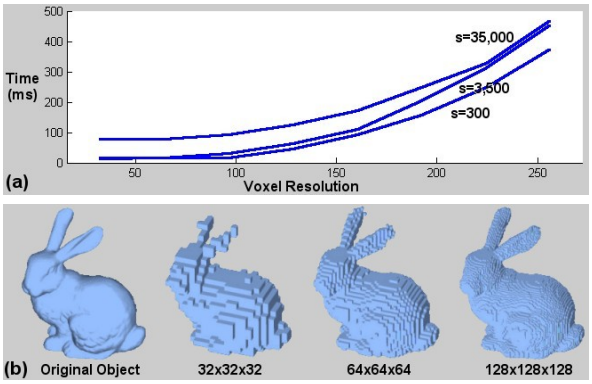


Figure 4. (a) Elapsed time for different voxel and object resolutions (s is object resolution). (b) Voxelization in different resolutions.

time increases sublinearly with object resolution. Computation time increases quadratically w.r.t. voxel resolution instead of cubically, as one might except. The reason behind this is that the bottleneck of the algorithm is the transfer of the buffers from video to main memory. Therefore, performance is mostly affected by the size of these buffers which leads to quadratic behavior. Fig. 4 (b) shows the effect of the choice of voxel resolution. Our algorithm applies to any resolution as long as it is bellow the maximum screen resolution supported by the graphics card.

Our algorithm assumes that the object is strictly closed and has no irregular faces (e.g., crossing, duplicate). Although objects that are triangulated correctly don't suffer from these problems, not all real life objects are ideally triangulated. Odd values in the stencil buffer is an indication that such problems exist. We also observed that triangles perpendicular to the z-buffer planes may produce certain artifacts, especially in lower voxel resolutions. This is attributed to the non-linear accuracy of the z-buffer.

5. Conclusion

A voxelization algorithm that is both efficient and robust was presented. Graphics hardware is exploited to attain high performance while retaining generality. Our method scales very well with increasing voxel and object resolutions, making it suitable for most voxelization applications. Additionally, it self detects the rare cases where it fails. Future work will be focused on how to handle the general case efficiently. The direction that will be followed is to exploit the programmable pipeline offered by the latest graphics hardware.

References

[1] R. Benjemaa and F. Schmitt. Fast global registration of 3D sampled surfaces using a multi-z-buffer technique. *Image and Vision Computing*, 17:113–123, 1999.

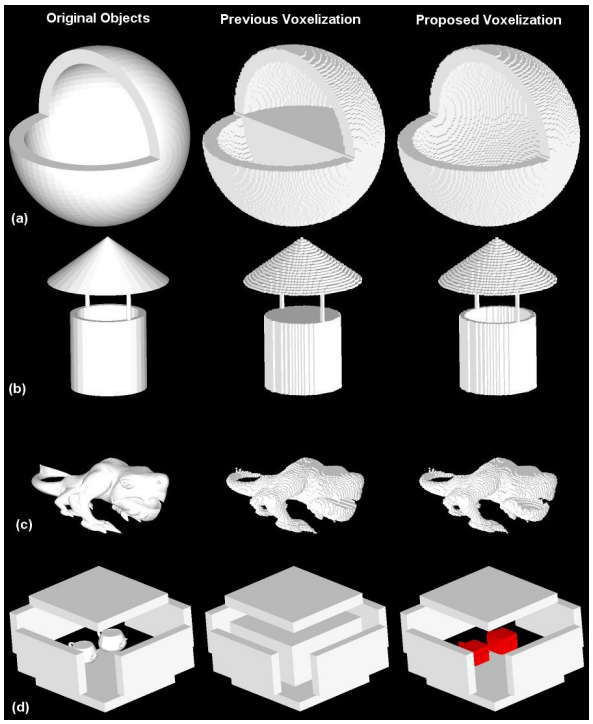


Figure 5. Voxelization results.

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

[3] 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.

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

[5] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. In *Proceedings of the IEEE Visualization*, pages 296 – 303, Atlanta, GA, 1995.

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

[7] G. Papaioannou, E. Karabassi, and T. Theoharis. Reconstruction of 3D objects through matching of their parts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(1):114–124, January 2002.

[8] G. Passalis, I. Kakadiaris, T. Theoharis, and M. Miller. Non-invasive automatic breast volume estimation for post-mastectomy breast reconstructive surgery. In *IEEE Engineering in Medicine and Biology Society*, Cancun, Mexico, 2003.

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

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

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