Incremental Triangle Voxelization

Frank Dachille IX and Arie Kaufman
Center for Visual Computing (CVC) and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present a method to incrementally voxelize triangles into a volumetric dataset with pre-filtering, generating an accurate multivalued voxelization. Multivalued voxelization allows direct volume rendering of voxelized geometry as well as volumes with intermixed geometry, accurate multiresolution representations, and efficient antialiasing. Prior voxelization methods either computed only a binary voxelization or inefficiently computed a multivalued voxelization. Our method develops incremental equations to quickly decide which filter function to compute for each voxel value. The method requires eight additions per voxel of the triangle bounding box. Being simple and efficient, the method is suitable for implementation in a hardware volume rendering system.

Key words: Voxelization, volume filtering, hardware, incremental algorithm, cut planes

1 Introduction

Our interest in volume graphics [11] and voxelization is motivated by the recent proliferation of volume rendering algorithms, hardware (e.g., *VolumePro* by Mitsubishi Electric), and the increasing use of discrete volumetric representation in various important application areas. These include medical imaging (e.g., CT and MRI), scientific visualization, simulation (e.g., flight and mission simulation), computer-aided design, animation, and virtual reality. Volume graphics can be used in place of traditional geometric applications as well as those applications that intermix geometric objects with 3D sampled or computed datasets.

Traditional computational bounds to the use of volume graphics (i.e., memory storage, bandwidth, and processing) continue to be shattered, allowing mainstream use of volume graphics. Leading the way is a recently available PC-based hardware accelerator board for volume rendering, *VolumePro* [14], manufactured by Mitsubishi and based on the Cube-4 architecture developed at SUNY Stony Brook [15]. With the advent of widespread volume graphics, new applications and modalities will be forthcoming. In this paper, we seek to spur further development of volume graphics by providing efficient, simple methods to accurately voxelize geometric models and to implement cut planes efficiently.

The advantages of volume graphics are many-fold, the primary being that an object interior can be modeled and visualized and amorphous phenomena can be handled naturally. In addition, the uniformity of representation allows object independent processing based on sound theoretical techniques. In this way, various scanned physical phenomena and objects, synthesized data, and sampled geometric objects can be processed, combined, and rendered together into effective visualizations. Volumetric representations have the advantage of pre-filtering, so that subsequent rendering can proceed efficiently without aliasing. Volume graphics is also relatively insensitive to object and scene complexity; detailed polygon meshes or complex objects can be directly represented using a finite volume, which can often be more compact. Furthermore, volumetric multiresolution pyramids allow antialiased rendering at various image sizes and are simple to generate, topology independent, and efficient for simplification [5]. Applications of pre-filtering are explored in Section 3.

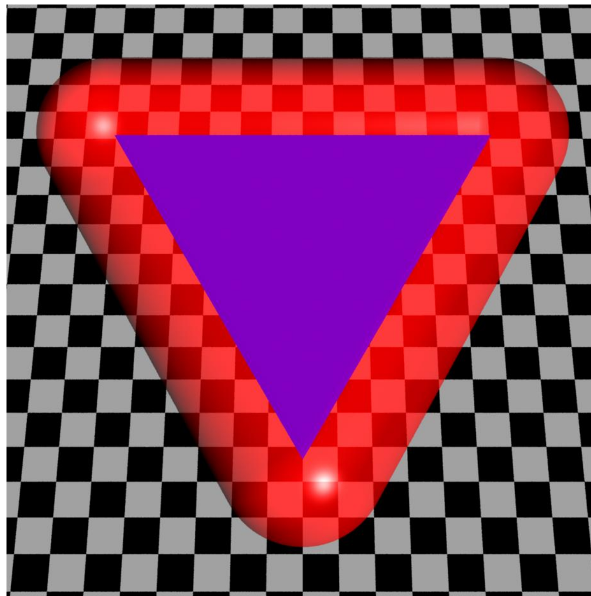

Figure 1: *The 3D region of influence around a triangle.*

Volume visualization is often enhanced by the combination of information from multiple data sources, both discrete and continuous. For example, medical visualizations can benefit from combining volumetric medical data with polygonal objects (e.g., prostheses, radiation therapy beams, and virtual scalpels placed within a sampled MRI or CT dataset). Geophysical visualizations can benefit from rendering analytical objects within a volume (e.g., oil drilling paths, pipeline placement, and man-made structures superimposed within a geophysical dataset).

To create these visualizations, it is necessary to combine volume rendering with traditional surface rendering. One approach is to render the surfaces into a z-buffer, then combine the z-buffer image with volume slices during texture map based volume rendering. However, this approach severely limits the quality and flexibility of the rendering and does not permit translucent surfaces without complicated sorting. Our preferred approach is to convert continuous surfaces into a discrete volumetric representation, called a voxelization [19]. The surfaces can be directly voxelized into the original volume and rendered with the usual means. If the surfaces are dynamic relative to the volume, they can be voxelized into a separate volume and combined only during rendering by inter-

leaving volume samples in the direction of each image pixel (e.g., during ray casting).

Such applications and the new hardware driving them spur the development of efficient and accurate voxelization techniques. Cohen-Or and Kaufman [2] derived the theoretical properties of voxelizations in a raster grid of binary-valued voxels. Objects represented by a 3D grid of discrete values have topological properties analogous to their continuous counterparts. For example, a *6-connected* discrete 3D line is a set of voxels which are adjacent to another through at least one of the 6 voxel faces. A *6-tunnel-free* discrete 3D surface is a set of voxels which do not allow any 6-connected line to pierce it (i.e., the intersection of the two sets is not null).

Kaufman [8, 9, 10, 12] presented efficient methods to generate binary voxelizations of many geometric primitives. Huang et al. [6] detailed the accuracy (i.e., separability and minimality) properties of binary voxelizations of planar objects. However, a direct visualization of binary-valued voxels typically appears to be a set of cuboid bricks with hard, jagged edges. To avoid this image aliasing we use pre-filtering, in which scalar-valued voxels are used to represent the percentage of spatial occupancy of a voxel [19], an extension of the two-dimensional line anti-aliasing method of Gupta and Sproull [4]. The scalar-valued voxels determine a fuzzy set such that the boundary between inclusion and exclusion is smooth. Direct visualization from such a fuzzy set avoids image aliasing. Recent work on voxelization has focused on generating a distance volume for subsequent use in manipulation [1] or rendering [3].

Šrámek and Kaufman [17] showed that the optimal sampling filter for central difference gradient estimation in areas of low curvature is a one-dimensional oriented box filter perpendicular to the surface. Since most volume rendering implementations utilize the central difference gradient estimation filter and trilinear sample interpolation, the oriented box filter is well suited for voxelization. Furthermore, this filter is an easily computed linear function of the distance from the triangle. Their voxelization method was accurate, but did not address efficient methods for triangle primitive voxelization.

This paper proposes an efficient, incremental algorithm for multivalued triangle voxelization suitable for both software and hardware implementations. The term multivalued refers to scalar-valued voxels, as opposed to binary-valued voxels. Voxelization is conceptually similar to 2D rasterization, which is conventionally performed in hardware for sake of speed. 3D voxelization is more computationally intensive than 2D rasterization by one dimension, so it is important to consider a hardware solution. Our algorithm could be built into volume rendering hardware to voxelize polygons at interactive rates. The hardware could then provide combined visualization of continuous polygons and/or discrete volumetric data by combining the two volume datasets during rendering.

Conventional graphics hardware only rasterizes points, lines, and triangles; higher order primitives are expressed as combinations of these basic primitives. Similarly, we choose to voxelize only triangles since all other primitives can be expressed in terms of triangles. Polygon meshes, spline surfaces, spheres, cylinders, and others can be subdivided into triangles for voxelization. Points and lines are special cases of triangles so they can also be voxelized by this algorithm. To voxelize solid objects, we can first voxelize the boundary as a set of triangles, then fill the interior using a volumetric filling procedure.

Figure 1 shows the region which is affected by the multivalued voxelization of a triangle. All voxels within the translucent surface, which is at a constant distance from the triangle, must be updated during voxelization. Jones [7] presented a method for identifying this region around a triangle, voxelizing it, and repeating for an entire triangle mesh. His method located the minimum distance to the triangle by direct calculation for each voxel. Our method produces similar voxelizations using a more efficient incremental method.

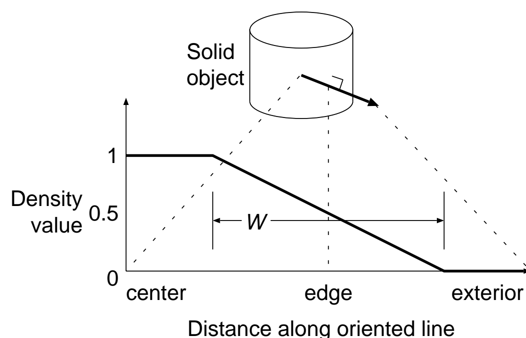

Figure 2: *Density profile of the oriented box filter along a line from the center of solid primitive outward, perpendicular to the surface. The width of the filter is W.*

Triangle rasterization methods have yielded some important algorithms which are extensible to triangle voxelization. Pixel Planes and Pixel Flow hardware [13] scan converts triangles with SIMD processors by computing three plane equations per pixel to determine whether or not it is inside the triangle. These equations, called *edge functions*, are linear expressions that maintain the distance from an edge by efficient incremental arithmetic. Shilling [16] used edge functions for antialiasing primitive edges. Our work extends this notion into three dimensions and applies antialiasing during the scan conversion of volumetric triangles.

2 Algorithm

The general idea of the algorithm is to voxelize a triangle by scanning a bounding box of the triangle in raster order. For each voxel in the bounding box, a filter equation is evaluated and the result is stored in memory. The value of the equation is a linear function of the distance from the triangle. The result is stored using a fuzzy algebraic union operator — the *max* operator. Thus, the complexity of the algorithm is $O(nk^3)$ where k is the average size in volume units of n triangles. The complexity has a lower bound of $\Omega(nk^2)$, since the triangles may be oriented perpendicular to a major axis and the thickness is constant.

2.1 Inclusion testing

The inclusion of a voxel in the fuzzy set varies between zero and one inclusive, determined by the value of the oriented box filter. The surface of the primitive is assumed to lie on the 0.5 density isosurface. Therefore, when voxelizing a solid primitive as in Figure 2, the density profile varies from a value of one inside the primitive to zero outside, and varies smoothly through the edge. For a surface primitive such as the triangle in Figure 3, the density is one on the surface and drops off linearly to zero at distance W from the surface. For the remainder of this paper, we only treat the voxelization of surfaces, not solids.

The optimum value for filter width W is determined to be $2\sqrt{3}$ voxel units [17]. Rendering from a multi-valued voxelized model is most often performed by ray tracing an implicit function $f() = 0.5$. This places an isosurface at the density value of 0.5, halfway between the minimum and maximum

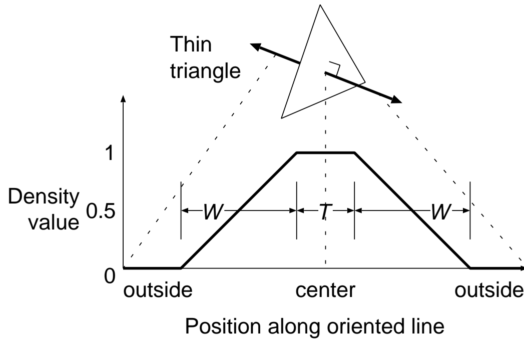

Figure 3: *Density profile of the oriented box filter along a line perpendicular to the triangle surface primitive.*

values. To resolve surface orientation, some for of shading (e.g., Phong) is applied based on an estimated gradient (normal) at the surface intersection. The oriented box pre-filter is designed to be combined with the central difference gradient estimation (i.e., $G_x = f(x+1, y, z) - f(x-1, y, z)$, etc.). Because the overall width of the central difference filter is at most $2\sqrt{3}$ units, a correct gradient is always found on the 0.5 density isosurface. Normally, the surface thickness T is zero unless thick surfaces are desired (see Figure 3). If more accurate shading is desired, an analytical normal could be computed using the original surface model, and stored at each grid point at a storage premium.

Based on a 0.5 density isosurface, the apparent thickness of a surface voxelization is $T + W$. By thresholding at 0.5 density, a 6-tunnel-free set of voxels is generated when $W \geq 1$ [6]. This property is useful for volumetric filling, (e.g., in order to generate solid objects).

All voxels with non-zero values for a triangle are within a bounding box $S = W + T/2$ voxel units larger in all directions than a tight bounding box. Therefore, the first step of the algorithm determines a tight bound for the triangle, then inflates it in all directions by S voxel units and rounds outward to the nearest voxels.

Figures 4 and 5 show the seven regions surrounding a triangle which must be treated separately. Each candidate voxel must be tested for inclusion within the seven regions, then filtered with a different equation for each region. In the interior region of the triangle (R1), the value of the oriented box filter is simply proportional to the distance from the plane of the triangle. In regions along the edges of the triangle (R2, R3, and R4), the value of the filter is proportional to the distance from the edge of the triangle. In regions at the corners of the triangle (R5, R6, and R7), the value of the filter is proportional to the distance from the corner of the triangle.

The regions are distinguished by their distance from seven planes. The first plane a is coplanar with the triangle and its normal vector **a** points outward from the page in Figure 5. The next three planes b, c, and d have normal vectors **b**, **c**, and **d** and pass through the corner vertices C_1, C_2, and C_3, respectively. The final three planes e, f, and g are perpendicular to the triangle and parallel to the edges; their normal vectors (**e**, **f**, and **g**) lie in the plane of the triangle and point inward so that a positive distance from all three planes defines region R1. All the plane coefficients are normalized so that the length of the normal is one — except for normal vectors **b**, **c**, and **d** which are normalized so that their length is equal to the inverse of their respective edge lengths. In that way, the computed distance from the plane varies from zero to one along the valid length of the edge. Table 1 summarizes the requisite conditions for inclusion in each region.

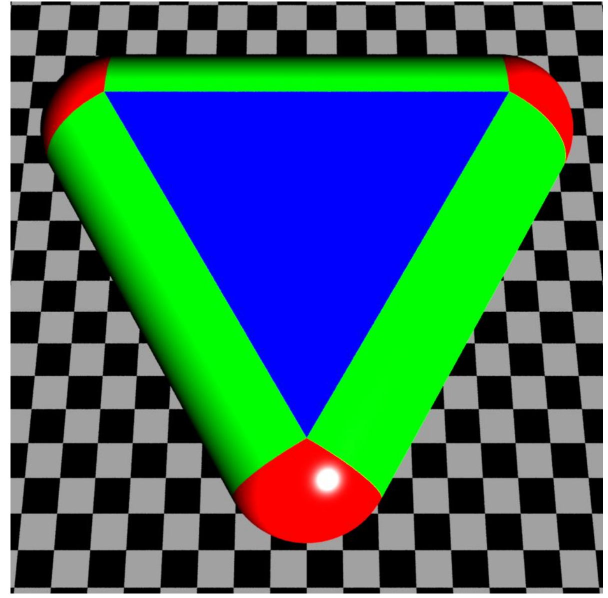

Figure 4: *Illustration of the seven voxelization regions around a triangle. Each affected voxel is either closer to the triangle face, an edge, or a corner.*

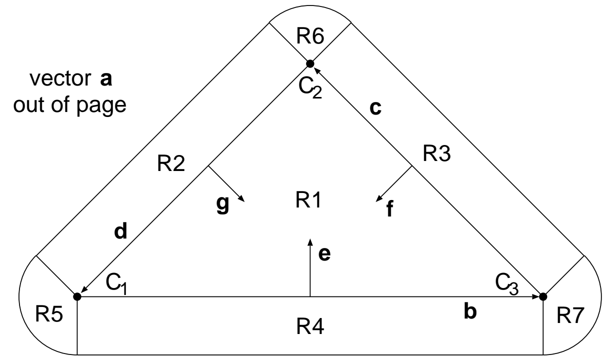

Figure 5: *2D illustration of the seven voxelization regions (R1-R7). The regions are delineated by seven planes a-g, whose normal vectors are shown.*

2.2 Distance from a Plane

For any planar surface, the distance of any point from the surface can be computed using the plane equation coefficients.

$$Dist = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}}$$

which simplifies to $Dist = Ax + By + Cz + D$ when the coefficients are pre-normalized. This computation can be made incremental so that when stepping along any vector, the distance only changes by a constant. For example, if the distance from a plane is $Dist$ at position $[x, y, z]$, then stepping one unit distance in the X direction changes the distance to

$$\begin{aligned} Dist' &= A(x+1) + By + Cz + D \\ &= Ax + By + Cz + D + A \\ &= Dist + A. \end{aligned}$$

Table 1: *The necessary (but not always sufficient) conditions for inclusion in a region based on the distances along seven plane normal vectors.*

	Plane Normal Vector						
Region	**a**	**b**	**c**	**d**	**e**	**f**	**g**
R1	(-S,S)				[0,∞)	[0,∞)	[0,∞)
R2	(-S,S)			[0,1]			(-∞,0)
R3	(-S,S)		[0,1]			(-∞,0)	
R4	(-S,S)	[0,1]			(-∞,0)		
R5	(-S,S)	(-∞,0)		(1,∞)			
R6	(-S,S)		(1,∞)	(-∞,0)			
R7	(-S,S)	(1,∞)	(-∞,0)				

Define Plane(A, B, C, D);
Find triangle bounding box(bb);
$Dist = A \times bb.min.x + B \times bb.min.y + C \times bb.min.z + D$;
$xStep = A$;
$yStep = B - A \times bb.width$;
$zStep = C - B \times bb.height - A \times bb.width$;
For $z = bb.min.z$ to $bb.max.z$ with unit steps
    For $y = bb.min.y$ to $bb.max.y$
        For $x = bb.min.x$ to $bb.max.x$
            store $f(Dist)$ in $[x, y, z]$
            $Dist = Dist + xStep$;
        end For
        $Dist = Dist + yStep$;
    end For
    $Dist = Dist + zStep$;
end For

Algorithm 1: *Incremental algorithm for computing the distance from a plane.*

In general, stepping along any vector $\mathbf{r} = [r_x, r_y, r_z]$, the distance from the plane changes by

$$Dist' = Dist + \mathbf{r} \odot [A, B, C]$$

where $\odot$ indicates the dot product. While scanning the bounding box of the triangle, the distance from the plane of the triangle can be computed incrementally with just a single addition per voxel (see Algorithm 1). This incremental algorithm is a 3D extension of the edge function used by Schilling [16].

The Y-step is more complicated than the X-step because it not only steps one unit in the Y direction, but it also steps back multiple units in the X direction, exactly like a typewriter glides back to the left margin of the paper *and* advances the line with one push of the return key. Similarly, the Z-step combines stepping back in both the X and Y directions and stepping forward one unit in the Z direction. This simple preprocessing step ensures efficient stepping throughout the entire volume. If numerical approximation issues arise, then it is possible to store the distance value at the start of each inner loop and restore it at the end, eliminating numerical creep due to roundoff in the inner loops.

For multivalued voxelization, seven plane distances are required, so seven additions are required per voxel to compute the plane distances. Other computations per voxel include incrementing the loop index, comparisons to determine the appropriate region, and, if necessary, computations to determine the density.

2.3 Distance from a Triangle

In region R1, the density value of a voxel is computed with the box filter oriented perpendicular to plane a. Given a distance $DistA$ from plane a, the density value V is computed using:

$$V = 1 - \frac{|DistA| - T/2}{W}.$$

In region R2, the density is computed using the distance from planes a and g:

$$V = 1 - \frac{\sqrt{DistA^2 + DistB^2} - T/2}{W}.$$

Similarly, region R3 uses planes a and f, and region R4 uses planes a and e. Region R5 uses the Pythagorean distance from the corner point C_1:

$$V = 1 - \frac{\sqrt{(C_1^x - x)^2 + (C_1^y - y)^2 + (C_1^z - z)^2} - T/2}{W}.$$

Likewise, regions R6 and R7 use corner points C_2 and C_3, respectively.

2.4 Shared Edges

At the shared edge of adjacent triangles, we want to avoid cracks. Fortunately, the oriented box filter guarantees accurate filtering of the edges for any polyhedra, provided we correctly compute the union of the voxelized surfaces. Multivalued voxelization is based on fuzzy algebra, in which true/false Boolean decisions are abandoned in favor of scalar values indicating a continuously variable percentage of truth, or in our case, occupancy. The union operator can be defined [19] over multivalued density values $V(x)$ with $V_{A \bigcup B} \equiv max(V_A(x), V_B(x))$. Other Boolean operators are available; however, the *max* operator preserves the correct oriented box filter value at shared edges. At the edge of a triangle, the oriented box filter generates a cylinder on the 0.5 density isosurface (see Figure 4). If an edge is shared between two triangles, then the two coincident edge cylinders are superimposed yielding a smooth transition between them. Unfortunately, the *max* operator can introduce discontinuities at polygon intersections (e.g., a triangle piercing another).

The *max* operator permits us to voxelize triangles in any order without consequence. The implication of using *max* in our algorithm is that we must read the current voxel value from memory, then possibly modify it and write it back to memory. Thus, a maximum of two memory cycles are required per voxel, although this is true for any algorithm that voxelizes in a separate pass for each primitive.

3 Pre-filtering

Voxelization is a pre-filtering operation; the representation is filtered during generation so that aliasing is avoided during rendering. Pre-filtering is a powerful tool that allows complex calculations to take place off-line so that subsequent rendering from multiple viewpoints is optimized. Here we present two techniques that can be performed with our voxelization method.

Pre-filtering can be used to generate a series of volumes [5] of different resolutions (see Figure 6). This technique is useful for rendering images of different sizes; the size of the volume is chosen to correspond to the size of the final image. In this way, aliasing is avoided at all image resolutions and no unnecessary work is performed in rendering parts of the scene not visible at the image scale. Furthermore, low resolution volumes generated by our method provide accurate topology of the model.

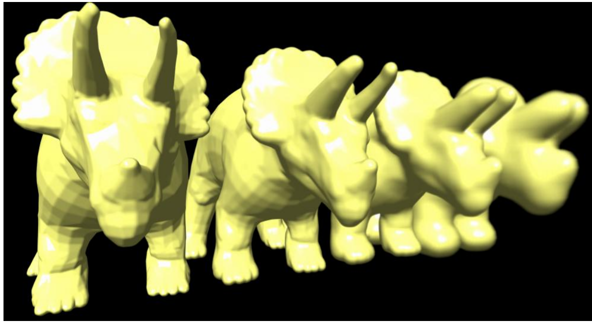

Figure 6: *Multiresolution triceratops voxelizations: maximum dimensions of 512, 256, 128, and 64 voxels.*

Figure 7: *Pre-filtered helicopter blade with motion blur efficiently rendered in a single pass at the same speed as without motion blur.*

Pre-filtering can additionally be used to model motion blur. As an object sweeps past a camera, it sweeps out a complex volume during the time the shutter is open, causing motion blur. To accurately render this, conventional rendering techniques actually render multiple images and blend them into a single image. This is accurate, but very slow. With pre-filtering, we can perform the sweeping operation once, during voxelization, so that motion blur can be rendered in the same time as regular volume rendering. This only works for certain cases where the motion is constant, (e.g., the same direction and/or rotation). A good example of this is a helicopter blade which spins at a constant speed during flight. We voxelized the blade spinning at the rate of 5Hz for an animation frame rate of 30Hz. That means that the blade sweeps out an arc of $\frac{5}{30}(2\pi)$ radians each frame. We voxelize by integrating the voxel density over the time of the frame. Because the inner portion of the blade sweeps out a smaller volume, the average density is much higher than the outer portion, where each voxel is occupied only a small portion of the time. The volume rendering transfer function is set so that the lower density values are less opaque and higher density values are more opaque. This correctly gives the visual impression of higher opacity near the center and lower opacity near the edge. The resulting image is shown in Figure 7.

4 Implementation

4.1 Software

The algorithm and volume rendering routines are implemented in object oriented C++. However, the inner loop of the algorithm avoids using C++ classes for a significant performance increase.

The efficiency can be further increased by limiting the amount of unnecessary computation. Specifically, the bounding box often contains a greater percentage of voxels unaffected by the triangle than affected by it. Efforts to obtain a tighter bounding box generally increase the complexity of the algorithm. Therefore, such optimizations only increase efficiency when the size of the triangles is large (e.g., edges longer than 100 voxels). The bounding box can be made tighter by recursively subdividing the triangle when edge lengths exceed some constant.

A software implementation allows optimizations that are not possible in hardware. In software, usually a single comparison per voxel is all that is necessary since most of the voxels are unaffected by the voxelization of a single triangle. For a voxel to be considered, the distance from plane a must be less than or equal to S units. Therefore, a simple rejection test is used to eliminate most voxels from consideration. By eliminating most computation early, time spent traversing empty space is minimized and most of the time is spent computing the filter function.

The memory access patterns can be optimized for optimum cache coherence by reordering the computation. In our case, the triangles can be divided into volumetric groups suitable for the cache of the target platform. For example, with a 512KB cache and 1-byte voxels, the triangles could be voxelized into 64^3 sub-blocks of the volume, one sub-block at a time.

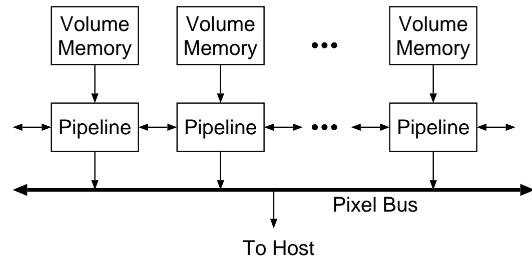

Figure 8: *The parallel, distributed architectural organization of the Cube-4 volume rendering accelerator.*

4.2 Hardware

With the advent of volume rendering hardware such as *VolumePro* [14], real-time volume visualization will soon be available for practical use. To visualize intermixed polygons and volumes, the polygons can be voxelized into the target volume and rendered in a single pass. If the polygons move with respect to the volume, then voxelization should occur in a copy of the original volume, so as not to corrupt the data. The multivalued voxelized polygon voxels can be tagged to distinguish them from volume data. In this way, polygons can be colored and shaded separately from other data.

The algorithm is efficiently implemented in the distributed pipelines of the Cube-4 volume rendering system. This algorithm adds just a small amount of hardware to the existing pipelines and performs accurate multivalued voxelization at interactive rates (see Section 5). Multiple Cube-4 pipelines work in parallel to retrieve voxels from distributed memories and perform ray casting in real time (see Figure 8). The volume is raycast beam-by-beam, one slice at a time into a buffer which eventually becomes the *Base Plane* (see Figure 9). After raycasting is complete, the *Base Plane* is 2D warped to the *Image Plane*. A primary advantage of the Cube-4 volume rendering algorithm is that the volume data is accessed coherently in a deterministic order. This feature allows orderly scanning of a bounding box similar to the software implementation with deterministic memory access.

The overall voxelization pipeline is shown in Figure 10.

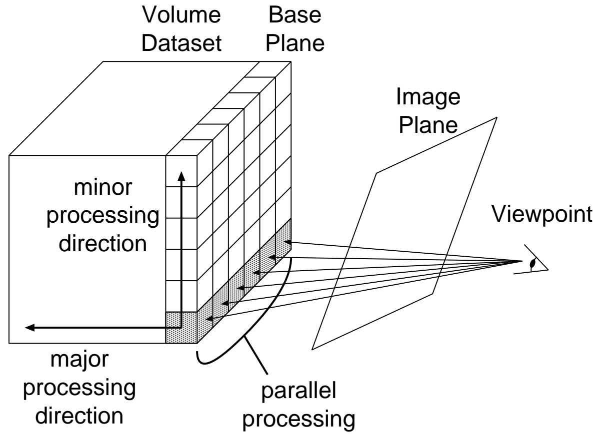

Figure 9: *Processing geometry of Cube-4. Processing occurs in slice order, beam by beam, then the* Base Plane *is 2D warped to the* Image Plane.

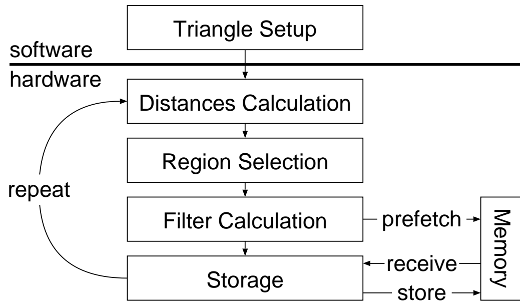

Figure 10: *An overview of the hardware voxelization pipeline.*

If on-the-fly voxelization is important, then there would be separate pipelines for volume rendering and voxelization. If voxelization could occur in a separate pass, then these two pipelines would be rolled into one, with the voxelization pipeline re-using most of the hardware from the volume rendering pipeline. The setup for each triangle occurs on the host system, much like setup is performed on the host for 2D rasterization. Per pipeline, this algorithm requires the use of 28 registers, 7 comparators, 12 adders, 3 multipliers, and one lookup table (LUT).

In the first hardware stage of the pipeline, the distances from the seven planes are computed. Seven simple distance units are allocated with four registers for each of the seven planes. One register holds the current distance from the plane and the other three hold the increments for the X, Y, and Z-steps. During each clock cycle of voxelization, the pipeline steps either in the X, Y, or Z direction, so the current distance is updated according to the direction of movement.

Since the hardware for looping through the volume is already built into the volume rendering pipeline, it is re-used here to scan the bounding box of the triangle. It is important to note that the hardware is a systolic array built with p parallel pipelines. The pipelines always operate on contiguous voxels in a beam, so it is necessary that the bounding box edges be a multiple of p. This potentially leads to inefficient load balancing, but p is typically small (i.e., four or eight).

After the seven plane distances are calculated, the values flow down the pipeline where tests are done in the next pipeline stage to determine in which region the current voxel resides. Only seven comparators are needed to decide the outcome of the truth table (see Table 1), due to the mutual exclusion of some cases. For instance, in Figure 5 if you are on the negative (lower) side of plane b, then it is not necessary to test the distances from plane f or g depending on the value of the distance from plane e.

The next pipeline stage, which computes the filter function, is only activated if the current voxel is within S voxel units of the triangle. Otherwise, the current voxel is unaffected by the triangle and different regions require different calculations, ranging from a simple linear expression to a complex Pythagorean distance evaluation. Since hardware must be able to handle all cases equally well, it must be able to perform a square root approximation by means of a limited resolution LUT. Luckily, the range of inputs and outputs is small, so the size of the required LUT is tiny by most standards. Furthermore, the Cube-4 hardware has several LUTs available for volume rendering which can be re-used for voxelization. Instead of providing three separate units to compute the expression: $V = 1 - (\sqrt{Dist} - T/2)/W$, it is more efficient to roll all the calculations into one LUT. In this case, the input is $Dist^2$, defined over [0,12], and the output is the density value V in the range [0,1].

Due to the mutual exclusion of the seven regions, it is sufficient to provide hardware for only the most complex filter calculation. The most complex calculation is the corner distance computation of regions R5, R6, and R7 which requires 5 adders and 3 multipliers in addition to the already mentioned square root LUT. The line distance computations in regions R2, R3, and R4 are simpler, requiring only 1 adder, 2 multipliers, and the square root LUT. Region R1 requires a single multiply to obtain the distance squared, which is the required input to the LUT.

The final stage of the pipeline computes the *max* operation using the current voxel value and the computed density estimate. The *max* operator is simply a comparator attached to a multiplexor such that the greater of the two values is written back to memory. Since most voxels in the bounding box are not close enough to the triangle to be affected by it, memory bandwidth will be saved by only reading the necessary voxels. Further savings can be reached by only writing back to memory those voxels that change the current voxel value. Since there is some latency between asking for and receiving word from memory, the voxel should be fetched as soon as possible in the pipeline and the results queued until the memory is received. The final stage is write-back to memory, which can be buffered without worry of dependencies.

4.3 Cut Planes

During volume rendering, the distance units of the voxelization pipeline are unused. Each of the seven units could be used to implement cut planes that divide the volume into regions, positive and negative. The voxels in the intersection of the positive-valued regions are rendered visible while all other voxels are rendered invisible. Two such planes can be used as the yon and hither clipping planes. The remaining five planes could be arbitrarily oriented by the user for isolating a region of interest. For proper anti-aliasing, the oriented box filter is applied to the opacity using a non-zero width W.

Alternately, the visible region can be defined using only a single thick plane ($T \gg 1$). Again, the final opacity is determined by modulating with the value of the oriented box filter. In this way, it is possible to perform oblique multiplanar reformatting, useful in the medical field. Using this new definition, the yon and hither clipping planes can be implemented using a single thick cutting plane, located halfway between the clipping planes. The remaining 6 plane distance units can be used

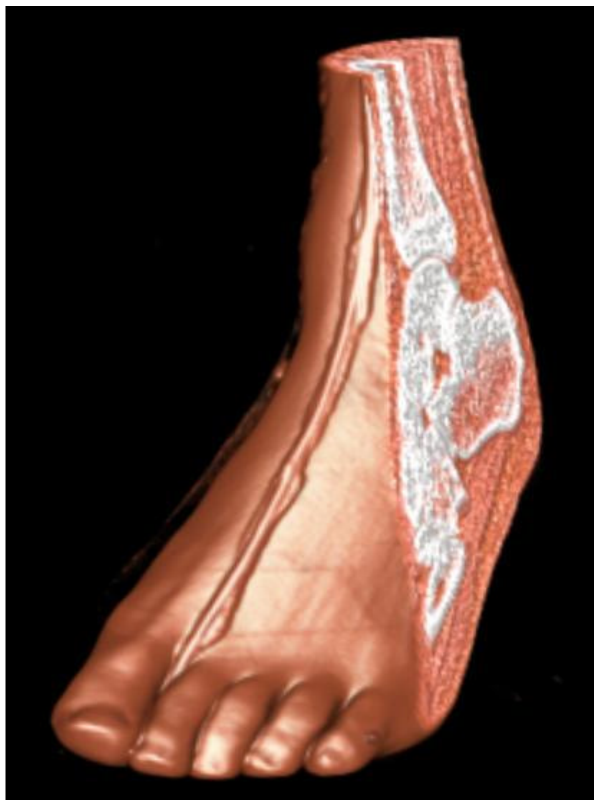

Figure 11: *Cross-section of a CT visible female foot rendered using our incremental cut plane algorithm.*

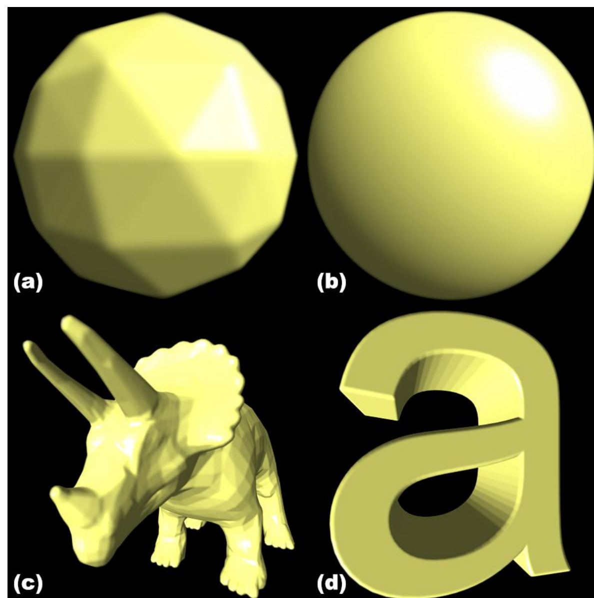

Figure 12: *(a) 128 triangle approximation of a sphere voxelized into a* 256^3 *volume in 1.8 s, (b) 32768 triangle approximation of a sphere voxelized into a* 256^3 *volume in 16.7 s, (c) 5660 triangle model of a triceratops voxelized into a* $175\times229\times512$ *volume in 5.8 s, (d) 1352 triangle model of the letter "a" voxelized into a* $123\times256\times256$ *volume in 5.3 s.*

for arbitrary region cutting. We simulated Cube-4-like object order volume rendering including an incremental cut plane. Figure 11 shows the CT foot of the visible female efficiently cut away to reveal inner structure.

5 Results and Discussion

The algorithm has been implemented on a 195 MHz MIPS R10000 and tested with various datasets (see Table 5). This algorithm has also been implemented and tested as part of a full-featured voxelization system [18]. The incremental voxelization method produces accurate, multivalued voxelization of triangles. The primary result is that voxelization proceeds at a high rate — often on the order of thousands of triangles per second. The rate varies with the average triangle size and their orientation. Sphere-3, shown in Figure 12a, is an approximation of a sphere with 128 triangles using three levels of recursive subdivision from an octahedron. Sphere-7 (see Figure 12b) is a better approximation of a sphere generated by four levels of recursive subdivision of Sphere-3.

The use of edge functions to compute distances has proven to be fast and easy to implement, even in hardware. The last column of Table 5 estimates the time required to voxelize the object in hardware, assuming a hardware fill rate of 2.5 Megavoxels/sec. This estimate is based on current *VolumePro* hardware which is capable of rendering a 256^3 volume at 30Hz (recall that voxelization requires up to twice the memory bandwidth of rendering).

Most of the voxelizations were at a medium resolution (i.e., at least 256 voxels), but a few were at a low resolution for comparison with prior work. Jones [7] voxelized objects into a 60^3 volume, taking on the order of 100 triangles per second. By comparison, our method in software voxelizes the same number of triangles into a similarly sized volume approximately one order of magnitude faster with a similar machine. Compared to Wang and Kaufman's method [19], ours provides at least one order of magnitude speedup for triangle primitives, but actually performs slower voxelizing solid primitives such as spheres and cones, which are approximated by a large set of triangles instead of a single implicit function. Such primitives are better voxelized using direct methods appropriate for each primitive, as in [18].

Other models voxelized were a triceratops model and the letter "a" (see Figures 12c and 12d). At a high resolution, the voxelized triceratops model appears as if it were an antialiased polygonal rendering. Pre-filtering is one of the advantages of voxelization followed by volume rendering compared to polygonal rendering, as shown in Section 3.

Since this is a rasterization method, the quality of the voxelization is directly related to the quality of the input triangle mesh and the desired resolution. Model meshes need not be manifold or otherwise structured, unless volumetric filling is employed. Degenerate triangles form either a line or a point which are handled as a special case using only three or one of our defined regions, respectively.

6 Acknowledgments

This work has been supported by NSF grant MIP9527694 and ONR grant N000149710402. The authors wish to thank Justine Dachille, Milos Šrámek, Kathleen McConnell, and the anonymous reviewers for their help and comments.

References

[1] D. E. Breen, S. Mauch, and R. T. Whitaker. 3D scan conversion of CSG models in distance volumes. In *1998 Volume Visualization Symposium*, pages 7–14. IEEE, Oct. 1998.

Table 2: *Results of voxelization of various geometric objects in software and hardware (estimated).*

Model	Number of Triangles (triangles)	Maximum Dimension (voxels)	Average Tri Size (voxels^2)	Software Time (s)	Software Throughput (tris/sec)	Software Fill Rate (Mvox/sec)	Hardware Time (ms)
Triceratops	5660	512	127.2	27.3	204	5.6	629
Sphere-3	128	256	1028	1.9	67	8.0	61
Sphere-7	32768	256	5.7	22.8	1437	2.8	255
Text	1352	256	164.5	5.8	231	3.4	74
Triceratops	5660	105	4.6	5.0	1100	2.5	50
Sphere-7	32768	60	0.2	14.6	2243	2.5	148

[2] D. Cohen-Or and A. Kaufman. Fundamentals of surface voxelization. *Graphical Models and Image Processing: GMIP*, 57(6):453–461, Nov. 1995.

[3] S. F. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *1998 Volume Visualization Symposium*, pages 23–30. IEEE, Oct. 1998.

[4] S. Gupta and R. F. Sproull. Filtering edges for gray-scale displays. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, volume 15(3), pages 1–5, Aug. 1981.

[5] T. He, L. Hong, A. Varshney, and S. W. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

[6] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *1998 Volume Visualization Symposium*, pages 119–126. IEEE, Oct. 1998.

[7] M. W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, Dec. 1996.

[8] A. Kaufman. An algorithm for 3D scan-conversion of polygons. In *Eurographics '87*, pages 197–208, Aug. 1987.

[9] A. Kaufman. Efficient algorithms for 3D scan-conversion of parametric curves, surfaces, and volumes. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21(4), pages 171–179, July 1987.

[10] A. Kaufman. *Volume Visualization*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

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

[12] A. Kaufman and E. Shimony. 3D scan-conversion algorithms for voxel-based graphics. In *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pages 45–75, Oct. 1986.

[13] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 231–240, July 1992.

[14] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. *Proceedings of SIGGRAPH 1999*, pages 251–260, Aug. 1999.

[15] H. Pfister and A. Kaufman. Cube-4 - A scalable architecture for real-time volume rendering. In *1996 Volume Visualization Symposium*, pages 47–54, Oct. 1996.

[16] A. Schilling. A new simple and efficient anti-aliasing with subpixel masks. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 133–141, July 1991.

[17] M. Šrámek and A. Kaufman. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):251–267, July 1999.

[18] M. Šrámek and A. Kaufman. vxt: A C++ class library for object voxelization. In *International Workshop on Volume Graphics*, pages 295–306, Mar. 1999.

[19] S. W. Wang and A. Kaufman. Volume-sampled 3D modeling. *IEEE Computer Graphics and Applications*, 14(5):26–32, Sept. 1994.