Submitted to Solid Modeling ‘ 95, November 1994

Rapid Previewing via Volume-based Solid Modeling

Naeem Shareef and Roni Yagel
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277

ABSTRACT

Quick previewing of 3D models is necessary for efficient product design and rapid prototyping. An inherent weakness of most solid modeling systems is that as model complexity increases quick 3D viewing suffers, especially on low cost workstations. We explore an alternative approach to surface representation in which object space, called volume, is subdivided into a 3D grid of cubic cells, each containing information on the object(s) which occupy it. A data structure is introduced that consists of a 2D array of pointers each holding a linked list of adjacent non empty cells. To benefit from data coherency along one dimension, we have developed new modeling and rendering algorithms that are beam-oriented, incremental, and integer-based. To illustrate the usefulness of our approach, we use our approach in Constructive Solid Geometry (CSG) modeling. We describe our prototype system and show, by comparing it to existing systems, that our data structure and its associated algorithms while being of finite resolution provide for suitable and more efficient model visualization.

1. Introduction

Computer modeling of object models is important for better and faster design. Three important aspects of the design process are: quick visualization (previewing), operations such as subtraction or gluing which simulate object construction processes, and quick object property determination, for example volume and surface area. These help in speeding up the design process, especially rapid prototyping.

The main drawback of many modeling systems is the use of surface primitives. A common format is *Boundary Representation* (B-rep), used by systems like ProEngineer and Catia. B-rep approximates an object with surface primitives such as polygons or parametric patches. Object detail is a trade-off between using simple polygon primitives versus higher order meshes, which are harder to manipulate in terms of computation cost. As model complexity increases, the number of surface primitives may be large enough that special rendering hardware is required for fast design. But even seemingly simple operations will experience this pitfall. For example, a polygon-based representation of a sphere, a common CSG primitive, may require a sizable number of polygons to get a good approximation. As another example, a box primitive with only six polygons may experience a polygon explosion if a dent, created by subtraction of a sphere, is introduced.

Volume Graphics is a new and rapidly growing subfield of Computer Graphics which is based on spatial partitioning [8]. Object space, called a *volume* is defined as a set of disjoint ordered cubic regions, called *voxels*, centered at grid points in Z^3. Each voxel represents a region with distance $\pm$ 0.5 units from the grid point in each axis direction. Alternatively, voxels might be defined as the region between grid points, i.e. grid points are vertices. Each voxel contains information on the object that reside within it. Converting geometric objects to their voxel representation (3D scan-conversion) is called *voxelization*. Many volume-based algorithms for volume visualization and manipulation have recently appeared in the literature [7]. Most require simple calculations because voxels are simple primitives. For example, a common operation of a cutting tool that drills holes in an object simply requires identifying and removing those voxels that intersect with the tool. A surface-based model will require expensive polygon operations and determination of new meshes to approximate the cutting path. Another advantage of volumes is that they decuple model creation (voxelization) from rendering thus avoiding the need for model re-evaluation for each rendering. The discrete nature of volumes, while posing quality and accuracy restrictions on the representation, frees the renderer from its sensitivity to model complexity. Once a model has been voxelized, rendering time depends on volume resolution on not on the number or type of objects.

Efficiency of volume-based algorithms depends on the data structure used to represent a volume. The two most important aspects in choosing a structure are its size and accessibility. *Full Grid* (FG) is a structure used by most volume-based algorithms developed so far. It is a static 3D array where each array element (i,j,k) corresponds to a voxel at (x,y,z). The main advantage is that voxels are randomly accessed in constant time. The regular ordering of voxels leads to simple incremental and integer-based algorithms. The main disadvantage, though, is the large storage requirement – the space complexity is $O(n^3)$, where n is the resolution of the volume. For example, a volume of resolution 512^3, where each voxel holds one byte, requires 128 megabytes of storage which is not feasible on most systems. If model resolution is increased, volume size increases polynomially. Algorithms will waste time traversing and processing empty space since FG contains all voxels in object space including those that store no useful information.

Voxel List (VL) is a volume structure on the other end of the spectrum. The structure is a linear linked list of non-empty voxels in an assigned traversal of the volume. Each list element contains a pointer to the next voxel in the list. A considerable amount of compression is achieved for many data sets, but compression degrades as the amount of data increases due to pointer overhead. Voxel access is a linear search exhibiting time complexity $O(n^3)$, where n is the resolution of the volume.

The *Array of Runs* (AR) structure is a compromise between these two extremes. A static 2D array, called a *pointer array*, with size of a volume face (n^2) represents a plane positioned on top of, and parallel to, a volume face. Each array element, called *run*, contains a linked list of non-empty voxels along a ray projected orthogonally from the array into the volume. Voxel access requires first accessing the 2D array, in constant time, and then searching the linear list. We denote the average list length by k where $k \le n$. AR provides data compression along an axis that is better than FG in the average case ($O(kn^2)$ instead of $O(n^3)$) and improved voxel accessibility over VL ($O(k)$ instead of $O(n^3)$). We later present algorithms that make use of the 1D coherency along the run.

Another data structure for volume representation is octree which subdivides the volume into octants [14]. Uniform octants are represented by leaf nodes while non-homogeneous regions are further subdivided and represented by a tree node with eight sub trees. Although modeling and CSG have been successfully implemented with octrees [11] (and variations of octrees [13]) we chose to explore AR since the algorithms for octree are more complex, require tree traversal with backtracking for model generation and rendering. In addition, spatial coherency in 3D is much harder to be identified and exploited. The AR is based on 1D coherency which is much more readily available for algorithm development.

AR-like structures have been presented in previous work. Hook [6] used a pointer array on polygonal data to simulate subtraction operations in a real-time NC milling system. The pointer array is actually the screen operating as an extended Z-buffer and so operations are view dependent. Each pixel holds a list of enter and exit point pairs, called *dexels*, along a ray orthogonally projected from the screen. A dexel is named as such because it represents depth at a pixel location. A cutting tool is converted into dexel format and subtraction implemented by removing those scene dexels which the tool overlaps. Udupa and Odhner [16] use an AR scheme in a representation called *Semiboundary* (SB). Runs contain only surface voxels, i.e. face-adjacent to at least one empty voxel. Each run is a linked list of triplets consisting of distance from the pointer array, a 6-bit code indicating neighboring non-empty voxels, and a unit normal vector. Cutting planes are used to quickly remove object data. Montani and Scopigno [12] introduced a structure called *STICKS* where the pointer array is called the *Sticks Holder Array* (SHA). Each voxel is assigned color. Each run is a list of adjacent voxels with iso-color, called a *Stick*. A storage scheme for personal computers is presented in which the SHA is subdivided into blocks and accessed with virtual memory paging. Blocks are transferred to main memory in a LRU swapping scheme.

In this paper we describe a structure we call *Segment Wall* (SW) which we contend is a further improvement over previous structures in terms of data compression and broad applicability. In contrast to [6] it is view independent and provide for sampled volume as well as synthetic geometry. Unlike [16] it provides the means to represent the interior of solids as well as their surface. It extends on [12] to include not only iso-value but multi-valued surfaces. In addition, we provide beam-oriented voxelization algorithm that are best suited to the SW.

In order to illustrate its usefulness, we use SW to implement *Constructive Solid Geometry (CSG)*, a powerful modeling technique due to its large object definition domain [3]. Models are defined by primitives, usually geometric objects such as boxes, spheres, and cylinders, that are combined using regularized boolean set operations such as union, intersection, and subtraction. Primitives are defined parametrically, for example a sphere is defined by its center location and radius. A model is compactly represented by a boolean expression, called a *CSG expression*, and usually converted into a more usable tree format called a *CSG tree*. Leaf nodes correspond to primitives and internal nodes to boolean operations. Model construction is a bottom-up tree traversal process ending at the root.

The method used by most systems to visualize CSG models is called *Ray Casting*. It defers model evaluation to rendering time [4]. A ray is traced from the eye, through each pixel on the image, into the scene. Each such ray is intersected with all primitives at the leaf nodes. A list of enter and exit points, called *spans*, are extracted and sorted in depth order from the screen. To compute the spans of the parent node boolean operations are applied to the span lists of child nodes. The recursively computed span list at the root node is used to determine the color at the pixel. Although some acceleration techniques exist (e.g., bounding objects, space subdivision), in principle this approach is slow because each ray must be intersected with many primitives. In addition, any change in viewing or rendering parameters requires model evaluation from scratch.

A second approach, used here, decouples model construction and viewing. A volume-based approach converts the model into a volume format and then applies volume viewing and manipulation algorithms. Primitives are converted to volume format in a process called *voxelization*. Boolean operations which are performed between volumes are reduced to simple voxel-voxel operations. The resulting volume is used for repeated rendering for any change in viewing or shading parameters without requiring re-evaluation of the CSG tree. We introduce algorithms for voxelization, boolean set operations, and viewing using the SW as our data structure for volume representation.

Section 2 of this paper describes the SW. Then, Section 3 introduces efficient voxelization algorithms for geometric objects. Section 4 details an algorithm to implement boolean set operations. Section 5 describes a fast backward projection rendering algorithm that exploits coherency along segments. Section 6 presents timing results of our system on typical models and compares those to the results form other systems. Finally, Section 7 concludes and suggest some directions for future work.

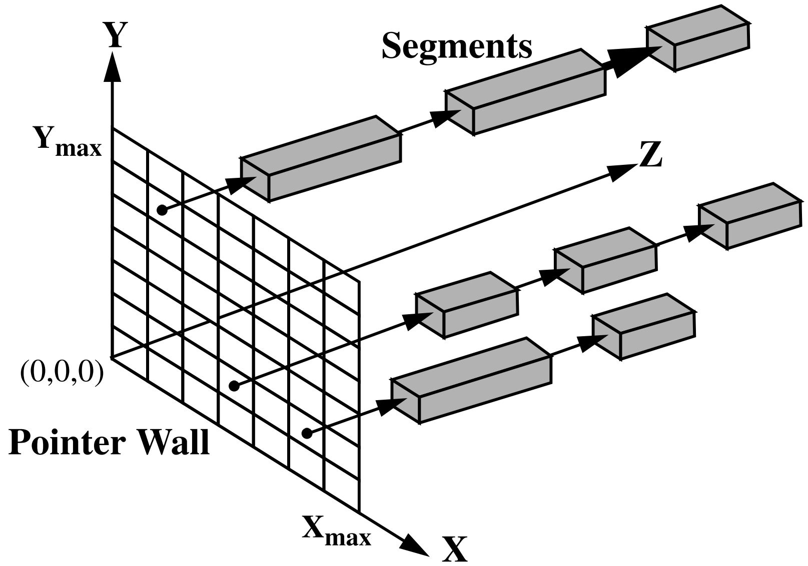


Figure 1. Illustration of the Segment Wall structure. A voxel at (*x*, *y*, *z*) is accessed first through the Pointer Wall, i.e., SW[x][y] then searched along a segment list with z.

2. The Segment Wall Data Structure

A *Segment Wall* (SW) consists of a 2D array of pointers called the *pointer wall*, representing a plane residing in object space (see Figure 1). The pointer wall is optimally placed parallel to one of the major axes. Each entry in the wall contains a list of all non-empty voxels along a ray projected perpendicular to the wall into the volume. Each ray is realized as a linked list of data items, each called a *segment*, which is a maximal set of adjacent non-empty voxels along the ray. Notice that voxels in a segment do not necessarily have the same value. Placement of the pointer wall ensures that runs of voxels are adjacent and along the same projected coordinate so we can best make use of 1D coherency. Each segment is a triple *(Bounds, Vals, Next)*. *Bounds* is a pair of integers $[\,(low, high) \mid low \le high]$, where *low* is the voxel distance of the closest voxel to the wall and *high* is the distance of the farthest voxel. The maximality property implies that (*low − 1*) and (*high + 1*) denote positions of empty voxels, while *(low = high)* corresponds to a segment with a single voxel. Segments do not overlap, that is, the *high* of a segment is always smaller than *low* of the next segment. *Vals* contains object properties of all voxels in a segment and is in fact a linked list of all voxels in the segment. *Next* is a pointer to the next segment along the ray. Each segment list is maintained in increasing sorted order by *low.*

The scheme used in this paper places the pointer wall parallel with the XY plane at Z = 0 (we assume left-handed space) with one corner at the origin. For a volume with dimensions $X_{max} \times Y_{max} \times Z_{max}$, the pointer wall has dimensions $X_{max} \times Y_{max}$ and all segments must satisfy $high < Z_{max}$. The following procedure is used to access a voxel value at *(x,y,z)*:

> **Access** pointer wall at *SW[**x**][**y**]*.
> **Search** segment list for the first segment *S* which satisfies {***z*** ≥ *S.low*}
>   **If** ***z*** > *S.high* **then**
>   **return** empty.
>   **else**   **Search** *S.Vals* for the voxel at ***z*** (with value *V)*
>   **return** *V*

The *Vals* structure contains object properties of all voxels within a segment. For the purposes of this paper, we define *Vals* as a linked list of object properties called an *Attribute List*. Each data item is called an *attribute* which is a maximal set of adjacent voxels with the same object properties. A segment is made up of a sorted list, by distance from the pointer wall, of contiguous attribute runs each consisting of a pair *(Pos, Prop)*, where *Pos* is the position of the first voxel in the run closest to the *low* of the segment, and *Prop* is a pointer to object properties. *Pos* must satisfy $low \le Pos \le high$. If two adjacent attributes have values (*Pos*, *Prop*) and (*Pos'*, *Prop'*) such that *Pos* < *Pos'*, then it is assumed that voxels in the range [*Pos... Pos'*] have object property *Prop*. If (*Pos*, *Prop*) is the last attribute, then voxels in [*Pos... high*] have property *Prop*. An attribute list, therefore, is a list with cardinality of at least one with form:

$$\langle (Pos_1, Prop_1), \ldots, (Pos_n, Prop_n) \mid low = Pos_1 < Pos_2 < \ldots < Pos_n < high \rangle$$

Figure 2 illustrates an attribute list of a segment with three objects. Other variations for *Vals* can be constructed to handle partial occupancy (i.e., objects that occupy only a portion of the voxel's space) and multiple objects within a voxel. However, in the work presented here we concentrate on uniform (binary) occupancy and single object per voxel.

The main advantage of using a wall of pointers structure is that spatial coherence along an axis reduces the number of voxels that need to be visited in rendering algorithms. The reduced representation also makes applications amenable to systems with limited memory capacity.

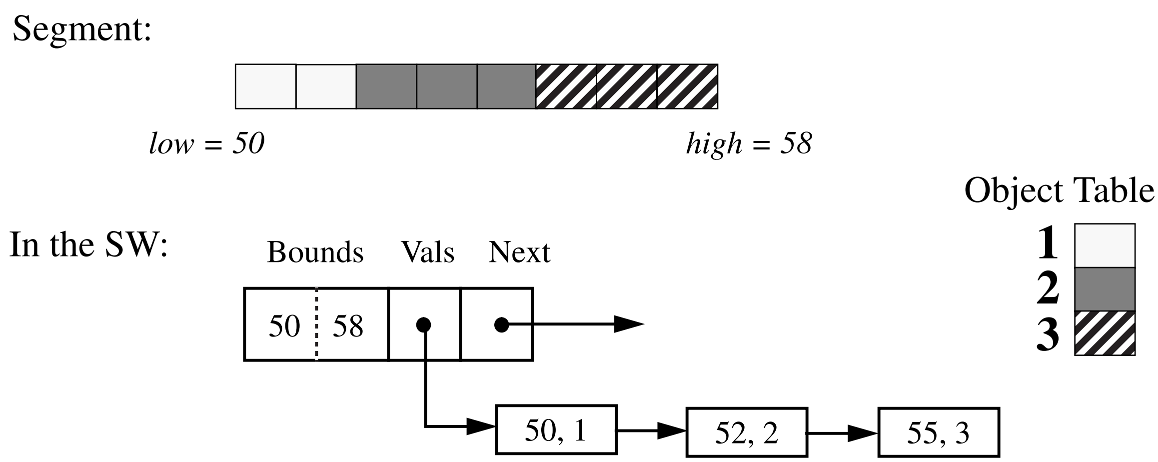


Figure 2. **Top: A typical segment with z bounds *low* = 50 and *high* = 58. Bottom: its internal representation with three color ID's shown in the corresponding *attribute list*.**

3. Beam Oriented Voxelization

Voxelization, as defined in the context of this paper, is a procedure to discretize (scan-convert) a continuous 3D geometric object into volume space. Essentially it is a point classification process where all voxels that reside on the surface or inside a continuous object are identified. 2D scan conversion algorithms for 1D objects such as lines, curves, and circles are well known [3]. Recently, a number of 3D voxelization algorithms for 3D primitives such as lines, curves, circles, cones, and tori have been developed [1]. Some methods extend 2D scan conversion algorithms to 3D, which is not straightforward for some object. All these 3D approaches assume a full grid volume. The SW structure has limited access via the pointer wall, so new voxelization algorithms are required. Algorithms either enumerate voxels using point classification in an exhaustive fashion or use recursive subdivision to decompose the model [14]. In this paper we restrict ourselves to solid objects with an inside and outside.

The structure of the SW implies the need for new algorithms for voxelization we call Beam Oriented Voxelization (BOV). BOV algorithms try to identify those segments which make up an object. Since access to volume data requires linked list traversal, SW access must be kept to a minimum. Optimally, each of those segments in a particular segment list should be determined and inserted in one step rather then inserting each voxel individually. One straightforward approach for BOV is to extend the method used in Ray Tracing. Rays are cast from the screen and intersected with the object. Enter and exit points are collected into spans. The pointer wall corresponds to the screen and spans to segments. As mentioned before, ray-object intersections are expensive. Simple extrusion algorithms where the extrusion path is a line perpendicular with a volume face are easy with SW. The master template can be constructed with a 2D bit editor and then scan-converted. If the extrusion path is perpendicular to the pointer wall then segments of the length of the extrusion path are inserted for each point in the master. Paths parallel with the pointer wall insert the scan-converted master as segments along the path.

Voxelize_Sphere(x_c, y_c, z_c, R, color)

  for y:= 0 **to** R **do**

    $r = \sqrt{R^2 - y^2}$;

    Draw_y_circle (x_c, y_c + y, z_c, r, color);

    Draw_y_circle (x_c, y_c -y, z_c, r, color);

end ***Voxelize_Sphere*****;**

Draw_y_circle (x_c, y_c, z_c, R, color)

  for x:= 0 to R do

    $r = \sqrt{R^2 - x^2}$;

    InsertRun (x_c + x, y_c, z_c, r, color);

    InsertRun (x_c, y_c -y, z_c, r, color);

end ***Draw_y_circle;***

Figure 3. Pseudo code for beam-oriented voxelization of a sphere. The basic operation the algorithm performs is to insert a run of voxels.

Many modeling implementations contain, among others, box, sphere, and cylinder primitives. These objects can be voxelized easily with incremental integer-based algorithms. One approach is to project the object onto the pointer wall, scan convert the silhouette, and insert segments. A box, placed with faces parallel with volume faces, projects to a rectangle. A segment of appropriate depth is inserted at each intersection point of the rectangle and pointer wall. The sphere projects a circle. Figure 3 outlines a sphere voxelization algorithm that is based on computing the extent of the segment that intersects the beam at any XY coordinate. We use a more efficient 2D incremental circle algorithm [3] to extract scanlines from the circle. For each scanline extracted the circle algorithm is applied again to generate scanlines for a circle of radius the length of the scanline and inserted as segments in a similar way to what is shown in Figure 3. Due to symmetry only one quadrant of the circle needs to be generated. This algorithm is quick since we are only applying an integer-based and incremental algorithm twice. The incremental circle algorithm can also be used to voxelize cylinders oriented parallel or perpendicular with the pointer wall. If perpendicular with the pointer wall then a circle is scan converted and segments size height of the cylinder are intersected at each circle point. If parallel with the pointer wall then the scan converted circle is inserted as segments along the length of the cylinder.

4. Boolean Set Operations with the SW

In this section we describe how to combine two SW structures with a boolean set operation. The method described here can be applied for any boolean set operation. Since volumes contain the insides of objects in an ordered regular grid, boolean operations are performed at the voxel level.

Boolean operators on two SW's involve super-imposing one SW over the other and combining respective segment lists. If one or both segment lists is/are null then the resulting segment list will be easy to construct. For example, the union of an empty segment list and a non-empty one is simply the non-empty one. Two non-empty lists are processed by combining segments pair-wise. Segment combinations can be categorized into six cases as shown in Figure 4. Cases 1 and 4 show two disjoint segments. Cases 2 and 5 show partially overlapping segments, and cases 3 and 6 show one segment completely overlapping the other. As seen in the Figure 4, the bounds (i.e., *low* and *high* values), of the resultant segments will be some combination of the bounds from the operand segments. So, boolean operations simply involve recognizing which case holds and then copying the bound values for the resultant segment(s).

The algorithm traverses both segment lists simultaneously with two pointers starting with the segment closest to the pointer wall. The pair of segments currently pointed to, is identified as one of the six cases and then processed. The resultant segment(s) are appended to the end of a running partial list which is the resultant segment after the operations is done. Only one pointer moves at a time in a "catch up" style order where the pointer lagging behind is moved until it catches up and surpasses the other pointer. Then the roles are switched and the other pointer plays catch up. This is a desirable method because it is not known where the next segment will be located along the Z axis. The lists only have to be traversed once since overlapping segments are determined in the

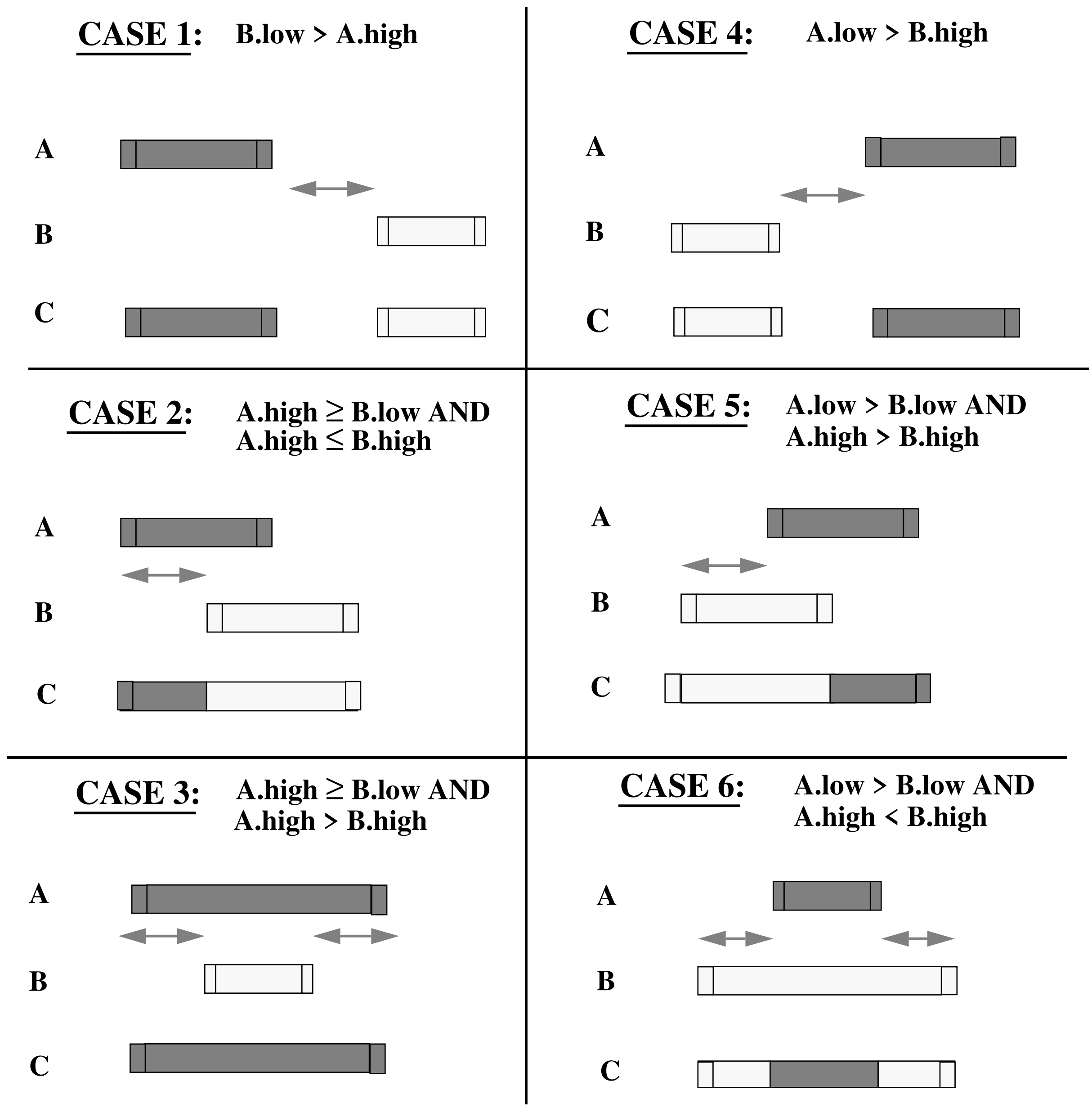
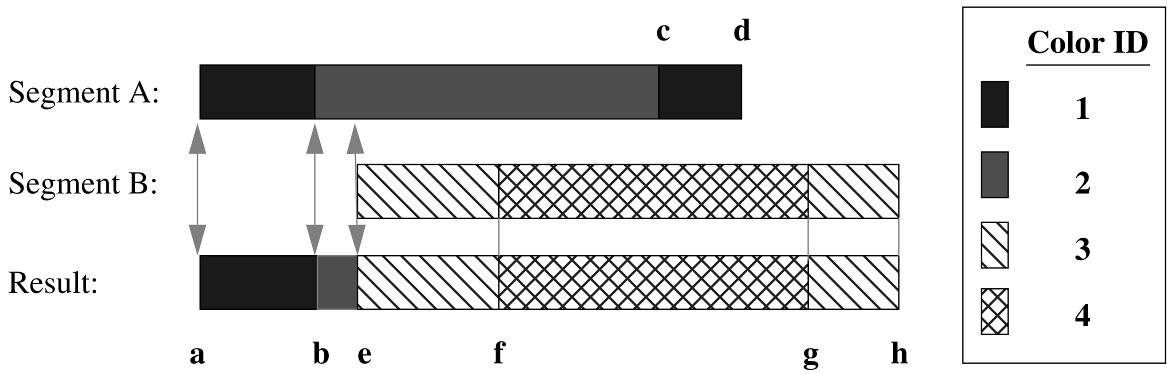


Figure 4. Six cases for segment overlap of two segment lists using the Union operator.

catch up scheme. The algorithm takes $O(kn^2)$, where k denotes the average number of segments contained in each segment list in the SW, and n is the resolution of the pointer wall. This is an improvement over voxel-voxel full grid implementations which take $O(n^3)$ time. Running the algorithm on SW will be faster than the STICKS representation [12] because SW will always have equal or fewer number of segments.

To handle multivalued segments, attribute lists are combined according to segment bounds. Since we assume that a voxel may contain only one object, we use the convention that one object will "dominate" over the other in boolean operation. Mixtures of objects are handled with more complex *Val* structures. The attributes of only those voxels that will end up in the resultant segment will be copied. Two pointers are used to traverse both attribute lists simultaneously, as was the case with segment traversal. First, the *low* and *high* bounds for the resultant segment(s) are determined as described above. Attribute positions are compared and the boolean operation to determine resultant attribute list values. Once the region between one *Pos* value in an attribute and the next in the list is processed the attribute pointer is moved ahead. Therefore, a resultant attribute list is constructed incrementally from the *low* to the *high* ends of operand segments, determining which attribute regions will be copied into the resultant list. Attribute lists are traversed only once with no backtracking. Figure 5 illustrates an example union operation for two overlapping segments.



	Segment A	Segment B	Result
Attribute Lists	[(a,1), (b,2), (c,1)]	[(e,3), (f,4),(g,3)]	[(a,1), (b,2), (e,3), (f,4), (g,3)]

Figure 5. Case 2 of overlapping segments of two objects, A and B, and determination of resultant attribute list. In this example object B dominates over A, which means that attribute values of A will be copied into the resultant attribute list.

5. Viewing Algorithm

Many rendering algorithms have recently been developed for volumes. A straightforward approach is to convert to a surface-based representation and then render using existing surface algorithms. Two methods which construct a polygonal mesh from volume traversal are surface tracking [5] and marching cubes [9]. Since polygons are inserted at the voxel level, which may introduce more surface primitives than if surfaces were used to start with, the overhead of conversion can be large. Decimation algorithms [15] can reduce the number of surface primitives, but may loose details in small object features and add to conversion overhead.

Rendering algorithms that directly operate on voxels can be classified as either forward projection, backward projection, or hybrid, depending on how the volume is traversed and sampled. A backward projection approach, e.g. ray casting, casts rays from the screen and sample the volume along the ray. Montani [12] presented a ray casting algorithm for their STICKS volume structure. This algorithm may directly be employed with the SW structure with minor variations to handle attribute list access.

A forward projection approach traverses and projects voxels onto the screen. The main issue is how hidden object (voxel) removal is handled since voxels can be traversed in any order. One popular algorithms is a Z-buffer for volumes, which is borrowed from surface graphics. Here, voxels are traversed in an arbitrary order then transformed by rotation angles α, β, γ and projected onto the screen. The algorithm for opaque volumes compares projected voxels with the Z-buffer and writes a voxel if its depth is smaller than an already residing voxel. Semi-transparent volumes require some structure to keep track of all projected voxels for compositing, which increases Z-buffer operations and updating time.

Frieder et al [2] utilized the regular order of voxels to develop a backward projection algorithm called Back-to-Front (BTF). Voxels are traversed in slice-row-column order such that farther voxels from the screen are projected before closer ones. Closer voxels simply overwrite farther voxels previously projected onto the screen, thereby avoiding an extra structure and computation overhead for hidden objects. Frieder et al showed that voxel traversal order only depends on direction of traversal along each axis, determined by viewpoint location, and that order of axis traversal can be chosen arbitrary. For example, a viewpoint placed in positive XY space and negative Z requires that the X and Y axes be traversed in a positive direction from the origin, and the Z axis to be traversed from the positive end to the origin. Assignment of slices, rows, and columns to axes is arbitrary if the direction of traversal is preserved. Frieder et al define slices perpendicular with the Z-axis, rows along the Y-axis, and columns along the X-axis and affixes the screen at a negative Z location. Each traversed voxel is transformed using a 4×4 transformation matrix on user-defined rotation angles α, β, and γ and orthogonally projected onto the screen. We denote this matrix by *T*:

$$T = \begin{bmatrix} t_{00} & t_{10} & t_{20} & t_{30} \\ t_{01} & t_{11} & t_{21} & t_{31} \\ t_{02} & t_{12} & t_{22} & t_{32} \\ t_{03} & t_{13} & t_{23} & t_{33} \end{bmatrix}$$

Machiraju and Yagel [10] introduced an optimization to this algorithm by observing that the projection values for face-adjacent neighbors of a transformed and projected voxel is a simple increment added to its computed projection value. This means that after one voxel, called the *seed*, is transformed with the 4×4 transformation matrix T, projected values for the rest of the voxels in the volume are computed by simply adding a step value to the projection value of its neighbor, which was also computed incrementally from the value in previous step. This algorithm reduces costly matrix operations to only three additions. This is based on the fact that traversal is already incremental as part of the algorithm. After the seed voxel is transformed three increments are computed for neighbors in each axial direction and projection values are incrementally computed in the slice-row-column traversal order. That is, if the transformed value for (x,y,z) is $(\bar{x},\bar{y},\bar{z})$ then: $(\bar{x}, \bar{y}, \overline{z+1}) = (\bar{x}, \bar{y}, \bar{z}) + [t_{20}, t_{21}, t_{22}]$.

We extend the optimized BTF algorithm just described for the SW structure. To avoid expensive pointer wall accesses, slices should be defined such that segment lists coincide with rows or columns. Our implementation defines slices perpendicular to the X-axis, rows along the Y-axis, and columns (where segments lists reside) along the Z-axis. Since a slice is perpendicular with the pointer wall, it coincides with a column on the pointer wall. Placement of the first slice and incremental slice ordering traversal depends on the direction of traversal along the X-axis which in turn depends on the viewpoint. The seed is chosen within this first slice on the pointer wall column at the top or bottom, depending on traversal direction along the Y-axis. If the Z-axis needs to be traversed in the negative direction then the seed voxel is still chosen on the pointer wall, but is not projected onto the screen. We recursively traverse a segment from the back if Z direction traversal order is negative. Computed projected values are saved from the last slice, row, or column for the next incremental calculation using pre-computed step values along each axis. Column traversal corresponds to segment list traversal.

Although segments are disconnected runs along the Z axis, we can avoid computing the transformation from scratch for each segment with only one multiplication. The passage of incremental projection values for two neighboring segments in a traversal are computed with an addition of the projection value of the end voxel of the last segment projected and the distance between the two segments. For example, if the coordinate of the *high* of the first segment is at (x,y,z) and the *low* of the next segment is at $(x,y,z+k)$ then:

$$(\bar{x}, \bar{y}, \overline{z+k}) = (\bar{x}, \bar{y}, \bar{z}) + k\,[t_{20}, t_{21}, t_{22}] .$$

The SW structure lends itself to further optimizations. First, we notice that many objects with sizable solid regions will have segments where only end voxels will be visible. This is the case where a segment is entirely surrounded by face adjacent segments that span the entire length of the hidden segment. In a pre-processing step to viewing, each segment is labeled as either *Hidden* or *Non-Hidden*, where at least one non-end voxel has at least one empty face adjacent neighbor. During segment projection, only end voxels of hidden segments are processed. Incremental projection values are computed for end voxels with one multiplication, as was computed between segments. The extra storage for the labels is only a one bit flag assigned to each segment in the SW. This is a small space cost to an already compressed structure as compared with the probable large time savings in rendering.

The BTF algorithm described for SW structure avoids traversal of chunks of empty space by ignoring runs with null segment lists. We also observe that there is a high probability that object data will be localized to some area. A simple extension to our rendering algorithm is to define a *Bounding Box*. Its extent is indicated by two diagonally opposite vertex points $((x_1, y_1), (x_2, y_2))$ on the segment wall. The BTF slice traversal will start from x_1 or x_2 and rows between y_1 and y_2 will be visited. If object data is sparsely distributed with respect to the pointer wall then the bounding box scheme becomes less amenable. Another approach is to indicate exactly which rows and columns of the pointer wall contain non-null segment lists. Here we define a pointer wall structure for the pointer wall. A 1D array with size X_{max} is aligned with the columns of the pointer wall. Each entry, an X location, is a linked list of runs, which are pairs of integer values that denote intervals for consecutive non-null segment lists. The array is defined along the X axis for convenience because of how slices are defined. Columns of empty segment lists are reduced to one comparison. Rows in each slice are found quickly along the ordered runs. The linked list of runs should be doubly linked because row traversal can be in both directions. The extra structure required with this scheme is still small compared with the space savings of the SW. It will also do a better job of identifying non-null segment lists than the bounding box.

An inherent problem with the BTF algorithm itself is the degradation of image quality when the volume is rotated at various angles by the introduction of “holes”. Since volumes are defined in discrete Z^3 space, a subset of continuous R^3 space, contiguous voxels before rotation may be some distance apart after rotation and projection to the screen. The SW wall structure can alleviate this problem. Since segments are contiguous voxels in object space, they correspond to a connected line in screen space. The two end voxels of a segment define endpoints of a line on the screen, so a 2D incremental line algorithm may be employed to discretely connect the points. Multi-valued segments can be partitioned in object space and then partitioned in screen space via the line algorithm. This method does not handle holes between segments. Splatting [18] is an approach which increases the area of influence of a voxel on the screen by coloring the pixel it hits and some local region or pixels around it. It is a more time consuming process because may types of filters may be used, but will improve the image quality.

Our system currently implements the incremental seed BTF algorithm using orthogonal projection. Segments are labeled as hidden or non-hidden with a flag in pre-processing step to viewing. If a voxel has a pointer to the object it belongs to then normal calculation and shading is straightforward. As mentioned before, this information may be lost, for example subtraction with CSG, where object information is lost. In this case local voxel information is used to compute normals. We have implemented depth shading and phong shading with the following normal estimation schemes: compute the normal from (1) the geometric definition of object (stored in a separate table), (2) by computing the grey-level gradients in object space, (3) by computing depth gradients in image space by using a depth buffer, and (4) by employing context sensitive shading where a depth buffer is segmented into regions and normals are computed with information available within a context [17].

6. Results

Our prototype CSG system, we call *VolCSG*, currently implements box, sphere, and cylinder primitives and union, intersection, and difference boolean set operations. We compare timings of our system and a CSG model rendering called *RayShade*, which uses the common ray tracing approach used by many systems. RayShade was instructed to perform simple raycasting with no transparency or reflections. Both systems use only phong shading without modelling global illumination. Normals were computed from the continuous definition of the objects and not extracted from the volume. The screen size and viewpoints were the same in both runs. The image size is 577^2 pixels while the volume size is 385^3 voxels. Images of these datasets can be seen in Figure 6 (MACHPART), Figure 7 (SPHEREFLAKE-40), Figure 8 (PIPES rendered by VolCSG) and Figure 9 (PIPES rendered by RayShade). Both VolCSG and RayShade were written in ANSI C and the results we report here are on a Hewlett Packard workstation model 735.

Table 1 presents the results of the VolCSG system. We observe that the SW structure is less sensitive to scene and object complexity. The slowest rendering time is only twice as slow as the fastest. In addition, although voxelizaion time for increasingly complex models grows linearly (see the various sphereflake models), the average rendering time remains almost constant. In comparison, Table 2 shows the results from RayShade. The slowest rendering time (49 seconds) is almost eight (7.7) times slower that the fastest rendering time (6.4 seconds). In addition, object complexity makes a big difference in rendering time.

DATA SET	**Voxelization**	**Rendering (total)**	**Voxelization + Rendering (total)**	**Rendering (average)**
MACHPART	19.7	85.2	104.9	5.5
OPENBOWL	73.9	152.4	226.3	9.52
PIPES	17.9	121.4	139.3	7.6
SPHEREFLAKE-1	0.45	73.0	73.45	9.1
SPHEREFLAKE-10	5.85	89.1	95.0	11.1
SPHEREFLAKE-20	9.71	90.6	100.3	11.3
SPHEREFLAKE-30	13.0	90.7	103.7	11.3
SPHEREFLAKE-40	16.0	90.7	106.7	11.3

TABLE 1. Eight CSG models run on VOLCSG. The first three are typical models. 16 views in a 360 degree path around the scene were taken for the first three models and 8 views were taken for the SPHEREFLAKE models. Column 1 shows voxelization times. Column 2 shows total rendering times for all views. Column 3 is the sum of 1 and 2. Column 4 is the average rendering time per view.

The results presented in Table 1 include voxelization times. The advantage of the SW approach is manifested when the cost of voxelization can be amortized over a large number of images produced from the same dataset. As the number of images increases, the preprocessing time invested in voxelization becomes negligible.

EXAMPLE	**Rendering (total)**	**Rendering (average)**	**Speedup: RayShade/Segment Wall**
MACHPART	372.2	23.3	**4.2**
OPENBOWL	627.5	39.2	**4.1**
PIPES	324.8	20.3	**2.7**
SPHEREFLAKE-1	51.2	6.4	**0.7**
SPHEREFLAKE-10	115.6	14.5	**1.3**
SPHEREFLAKE-20	203.7	25.5	**2.6**
SPHEREFLAKE-30	279.0	34.9	**3.1**
SPHEREFLAKE-40	392.0	49.0	**4.3**

TABLE 2. Eight CSG models run on RayShade. Models are identical to the ones used for VOLCSG. Column 1 shows total rendering times for all views. Column 2 shows average rendering times per view.

Examining the right most column in Table 2 we observe that as the model becomes more and more complex the advantage of VolCSG over RayShade becomes obvious. Even for simple models like the one we used and small number of images (8-16) VolCSG performs four times faster than RayShade.

Finally, examining the images produces by RayShade (Figure 9) and those produced by VolCSG (Figure 8) we conclude that the quality of images is certainly useful and could be used at least as a mechanism for rapid previewing. We work on improving image quality to the extent that VolCSG will be able to produce images of similar quality to RayShade while providing order of magnitude speedup for relatively complex models.

7. Conclusion

A new volume data structure is presented in this paper. It has advantages over other similar structures in terms of extra compression and flexibility to handle any voxel type. We present the structure in the context of solid modeling. We have developed algorithms for voxelizing objects called Beam Oriented Voxelization that are incremental and integer-based. Effectiveness of this approach was illustrated using CSG. In a two-phased approach in which model construction and viewing are decoupled we have developed efficient algorithms to implement boolean set operations and quick viewing of the structure. A prototype CSG modeler system has been developed with box, sphere, and cylinder primitives and union, intersection, and difference boolean operations. We compared rendering times with a typical CSG viewing approach in a system called RayShade. We have shown promise for this approach in terms of quick viewing and fewer space requirements than a full grid representation.

We are looking into BOV for other types of objects, such as cones, tori, and polyhedra. We plan to implement forward viewing algorithms (ray casting) that will be greatly optimized to exploit the special coherency inherent to the SW. As we mentioned in the paper, the data structure for *Vals* can model other types of voxels such as partial occupancy and multiple objects per voxel. We are looking at extending our modeling and rendering algorithm to use this capability. We expect also that these extensions will provide the means to generate higher quality images by providing intermediate density values for partially occupied voxel.

Unfortunately, the discrete nature of the volumetric approach poses limits to image quality and the accuracy of the model for the purpose of model measurements. however, the main advantage of this approach is that quick feedback is provided to the model designer especially if rapid prototyping is used. Since volumes are a more “natural” approach in representing objects, computing object properties such as volume and surface involve simply counting up voxels in the volume. Object properties, such as solid texturing, can be directly introduced into our scheme. We see our method being used as a first level screening tool that can provide the designer a rapid feedback.

Whenever the designer is happy with the model he can proceed to further improve the model while employing more accurate but time consuming rendering and measurement tools.

Acknowledgments

We would like to thank Raghu Machiraju for the splatting code. This project was supported by the National Science Foundation under grant CCR-9211288.

Color Figures

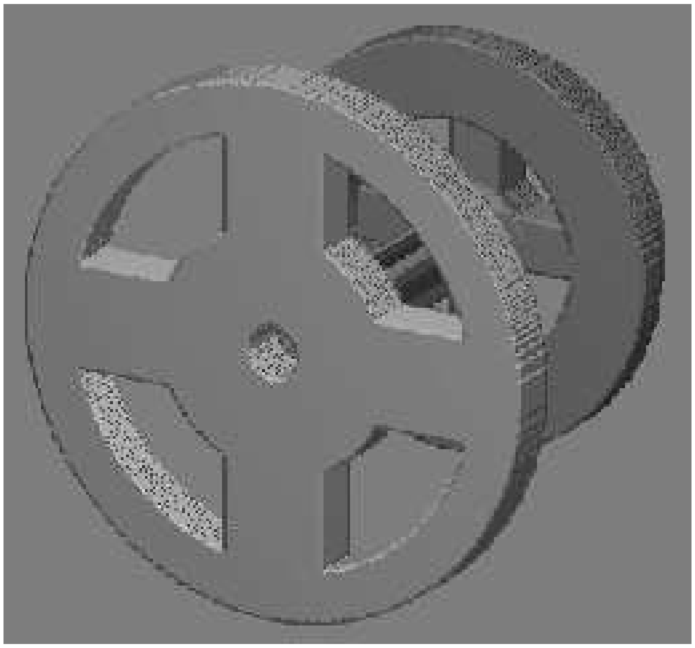


Figure 6. The MACHPART dataset

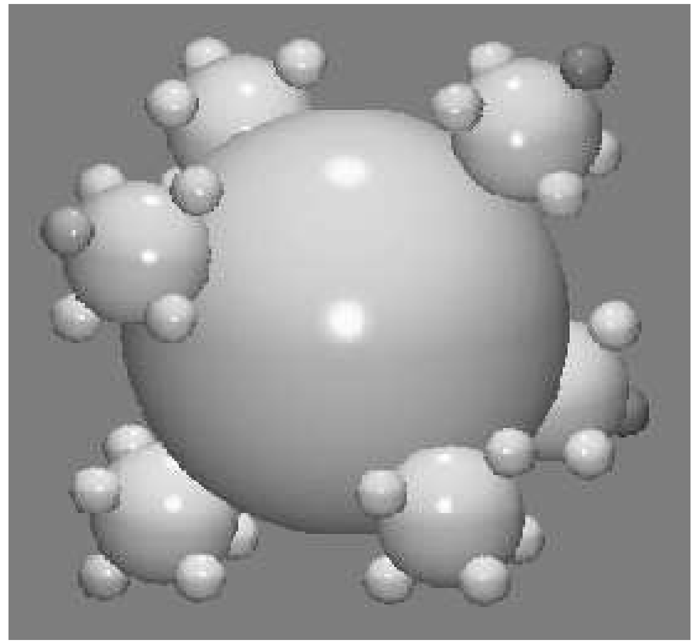


Figure 7. The SPHEREFLAKE-40 dataset.

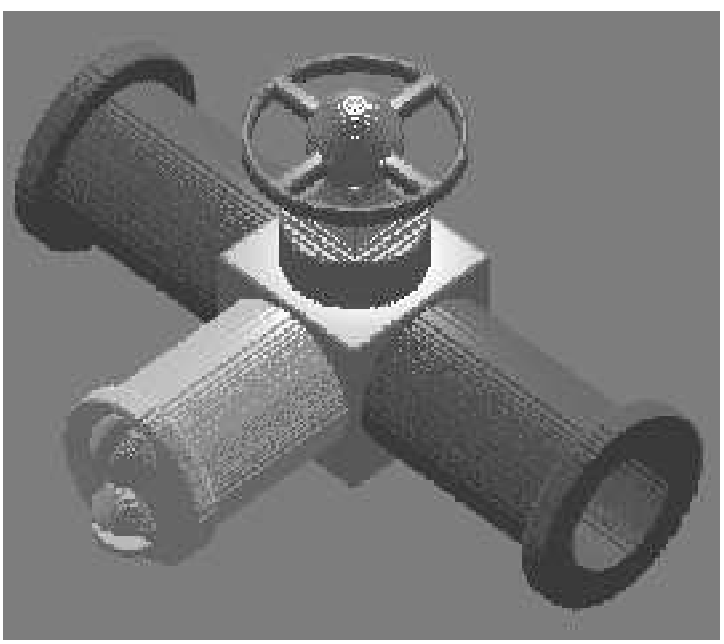


Figure 7. The PIPES dataset rendered by VolCSG.

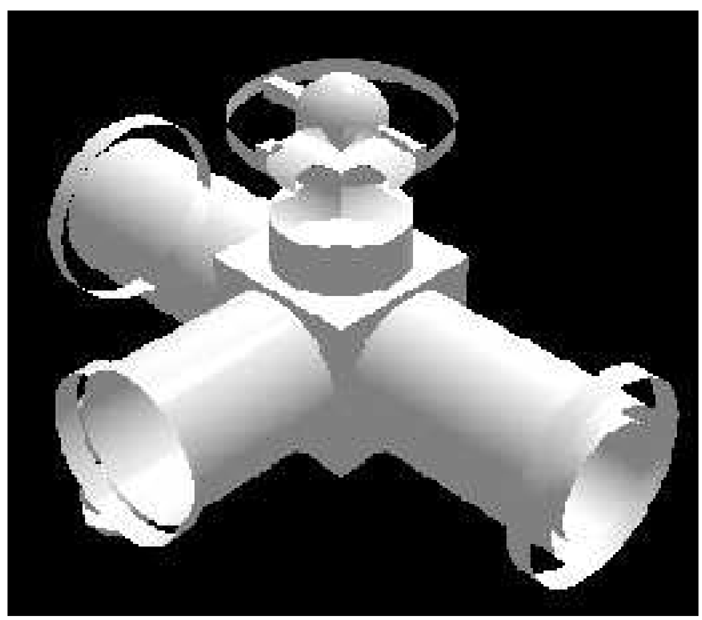


Figure 8. PIPES rendered by RayShade.

8. References

[1] Cohen D. and A. Kaufman, “Scan-Conversion Algorithms for Linear and Quadratic Objects”, in *Volume Visualization,* IEEE Computer Society Press, 1990, A. Kaufman (ed.) pp. 280-301.

[2] Frieder G., D. Gordon and R.A. Reynolds, “Back-to-Front Display of Voxel-Based Objects”, *IEEE Computer Graphics and Applications*, 5, 1, January 1985, pp. 52-60.

[3] Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

[4] Goldstein R.A. and R. Nagel, “3-D Visual Simulation”, *Simulation*, 16, 1, January 1971, pp. 25-31.

[5] Gordon, D. and J.K. Udupa, “Fast Surface Tracking in Three-Dimensional Binary Images”, *Computer Vision, Graphics and Image Processing*, 45, 989, pp. 96-214.

[6] van Hook, T., “Real-Time Shaded NC Milling Display”, *SIGGRAPH 86*, Vol. 20, No. 4, Aug. 1986, pp. 15-20.

[7] A. Kaufman (ed.), *Volume Visualization*, IEEE Computer Society Press, 1990.

[8] Kaufman A., D. Cohen, and R. Yagel, “Volumetric Graphics”, IEEE Computer, Vol. 26, No. 7, July 1993, pp. 51-64.

[9] Lorensen W.E. and H.E. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, *Computer Graphics*, 21, 4, July 1987, pp. 163-169.

[10] Machiraju, R., Yagel, R. “Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors”, *Proceedings of SUPERCOMPUTING’93*, Portland, Oregon, November 1993, pp. 699-708.

[11] Meagher, D., “Geometric Modeling Using Octree Encoding”, *Computer Graphics and Image Processing*, 1982, pp. 129-147.

[12] Montani, C., and R. Scopigno, “Rendering Volumetric Data Using The STICKS Representation Scheme”, *Computer Graphics*, Vol. 24, No. 5, Nov. 1990, pp. 87-93.

[13] Naylor, B.F., “Interactive Solid Geometry Via Partitioning Trees”, *Graphics Interface ‘92*, pp. 11-18.

[14] Samet, H., and R. Webber, “Hierarchical Data Structures And Algorithms For Computer Graphics”, *IEEE Computer Graphics & Applications*, July 1988, pp. 59-75.

[15] Schroeder W., J. Zarge, and W. Lorensen, “Decimation of Triangle Meshes”, *Computer Graphics*, 26, 3, July 1992.

[16] Udupa, J.K., and D. Odhner, “Fast Visualization, Manipulation, and Analysis of Binary Volumetric Objects”, *IEEE Computer Graphics & Applications*, Nov. 1991, pp. 53-62.

[17] Yagel R., D. Cohen, and A. Kaufman, “Normal Estimation in 3D Discrete Space”, *The Visual Computer*, Vol. 8, No. 5-6, June 1992, pp. 278-291.

[18] Westover, L.,“Footprint Evaluation for Volume Rendering,”, Computer Graphics, Vol. 24, No. 4, August 1990, pp. 367-376.