A Scan-Line Algorithm for Volume Rendering of Multiple Curvilinear Grids

UCSC-CRL-95-57

Jane Wilhelms Paul Tarantino Allen Van Gelder
University of California, Santa Cruz

November 20, 1995

Abstract

This report presents a volume rendering technique that is based on a scan-line algorithm with depth sorting. The algorithm can handle any connected grid, including multiple non-convex intersecting curvilinear grids, and is designed to be run in parallel on a MIMD architecture. Variable resolution is achieved by changing the height and width of the output screen. Scan conversion and compositing are done in software, which eliminates the need for special graphics hardware, as well as any artifacts associated with graphics hardware. A description of the data structures and pseudo-code is given along with a solution for rendering accurate depths in perspective viewing. The algorithm is compared, with respect to time and accuracy, against cell projection.

1 Introduction

Direct volume rendering produces a 3D visualization of 3D scalar data. Producing high-resolution images can be very expensive, especially if the data is in a curvilinear grid [Wil93]. Multiple grids that intersect make accurate rendering even more difficult for cell projection methods and create cell subdivision problems for algorithms that use cells as fundamental volume units [MHC90, Cha93, GP93, Luc92].

Our method decomposes the cells of each grid into six faces. Each face is represented by two triangles. The triangles are treated as individual polygons and passed to the renderer, which depth sorts the polygons for each pixel and integrates color from one polygon to the next. Accurate images are produced without subdividing any polygons.

1.1 Background

There are several algorithms for direct volume rendering of curvilinear grids. Most of these algorithms use either ray casting [Gar90, Use91, RW92] or cell projection [ST90, Wil92b, VGW93] techniques. These direct volume rendering algorithms address issues such as speed, picture accuracy, and flexibility. In addition to these issues, our algorithm addresses scalability, intersecting grids, and portability.

Ray tracing algorithms are based on the concept of sending rays from the eye, through each screen pixel, and into the volume. The color and opacity for each pixel is composited from the scalar values sampled from the cells intersected by the ray. A substantial amount of time is spent calculating intersections of the ray with entry and exit faces of cells. Early implementations of ray tracing curvilinear grids were done by Wilhelms [WCA$^+$90] and Garrity [Gar90]. Wilhelms and Van Gelder discuss a hiercharical ray-casting approach [VGKW95] which achieves acceleration by taking large steps over regions where data need not be processed at a fine resolution. Their approach, which uses multi-dimensional trees [WVG94], gives the user control over an error tolerance which directly effects the acceleration.

Cell projection methods are designed to take advantage of the speed of graphics hardware to render polygons. Volume elements are projected onto the screen in front-to-back or back-to-front order. Projection methods are generally very fast since color and opacity composition can be done by the hardware, however, if interpolation between sample points and integration in depth are not done accurately, visual artifacts may occur [WVG91]. Projection methods are very efficient for rendering rectilinear grids, since they can take advantage of cell coherence [WVG91]. Irregular volumes can be more expensive for projection methods since the cells have different shapes and the visibility ordering is not as trivial [VGW93].

Parallelization of both projection and ray casting are discussed by Challinger [Cha93]. She discusses several important parallelization issues including task generation (the decomposition of a large job into smaller tasks), synchronization, and memory management. She also describes a hybrid approach using a scan-line algorithm to sort cells and X-buckets to sort edges.

Giertsen and Peterson [GP93] used a cell-based scan-line algorithm to parallelize volume rendering over a network. They break the screen up into several small sections and distribute the sections to available processors. The processed sections are then received and displayed on the screen.

1.2 Scan-line Algorithms

Both the ray-casting and projection algorithms for volume rendering share a common factor in that they both can be improved by taking advantage of spatial coherence. By doing this, the two algorithms lose their distinct characteristics and merge to become a scan-line algorithm. Scan-line hybrid algorithms have been implemented by several people [MHC90, Cha93, GP93, Luc92]. They all take advantage of spatial coherence by transforming the volume into screen space and rendering the cells, or faces of cells in front to back order for each scan-line. Coherence is achieved by processing the scan-lines in sequential order.

Giertsen [GP93] intersects each volume element with the scan plane to get a polygon which is filled. He also divides the final image into rectangular sections for parallelization. Some scan-line methods [MHC90, GP93] sort volume elements into scan-lines, while others [Cha93, Luc92] break the faces of each cell into individual polygons which are then sorted.

This paper describes a method for rendering possibly intersecting polygons that have been extracted from multiple, intersecting curvilinear grids. A parallel implementation is presented and different approaches to maximizing scalability are discussed.

This algorithm is similar to that of Lucas [Luc92], with the exceptions that the facial polygons are decomposed into triangles, the visibility sorting is done during the creation and maintenance of X-buckets to allow for highly intersecting multiple grids, and we are using the bounding box as a virtual polygon, instead of clipping in the data structure.

2 Algorithm and Data Structures

The algorithm described here is based on Watkin's scan-line algorithm [Wat70], with several modifications, which will be described later in this section. Triangles are assumed to be the only type of polygons that are being rendered. This makes it easier to identify active edges and also keeps data consistency when interpolating between edges at different rotations. It is also easy to break up the quadrilateral faces of a curvilinear grid into a set of triangles. The details in this paper assume triangles, however, the algorithm could be adapted to use any polygon. This increases the number of polygons by a factor of 2. Other methods [Wil92b, Wil92a] have been reported that decompose each hexahedral cell into five tetrahedra, resulting in an increase in the number of polygons by a factor of $3\frac{1}{3}$.

The algorithm begins with the conversion of the vertices from world space to screen space, preserving depth values. This is done by passing each vertex through the geometry and the projection matrices and storing the locations back in memory.

2.1 Y-buckets

The volume is decomposed into polygons by representing each face of each cell with two triangles. Each triangle is given a sequential identifier (Id). The next step is to process all of the polygons into the Y-buckets. A polygon is considered active for a scan-line if it contributes to the image for that scan-line. There is one Y-bucket for each scan line and it contains the all of the triangles that become active (start contributing to the image at that scan line) on that scan line. The Y-buckets are implemented as one array that holds all of the Id's in scan-line order. Each Y-bucket can be accessed by a starting and ending subscript in to the Y-bucket array. The algorithm can save time by eliminating any triangles that may not contribute to the final image because of the following reasons:

- It is entirely off of the visible screen space.
- It is entirely out of the region defined to be drawn. (i.e. a clipped region)
- It does not cross a scan line. (The ceilings of the Y components of all three vertices are equal.)

- It does not cross a pixel in the X direction. (The ceilings of the X components of all three vertices are equal.)

Once the Y-buckets have been built, each scan line is processed and drawn into the frame buffer. An active list containing the polygon Id's of those polygons that contribute to the scan-line (i.e. in the Y direction) is maintained. Before processing the next scan-line, the previous scan line must be updated by adding in new triangles from the current Y-bucket and removing any triangles that are no longer active. Table 1 defines the data structure used for the active list in Y direction. The active list in Y direction

Long	Long	Long	Long	Long	Long	Long
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value

Table 1: Data Structure for Active Polygon List in Y direction. (Array)

is implemented as an array. As each new triangle is added into the active list, its orientation to the scan line must be determined by examining the X and Y values of the vertices. Figure 1 shows the two basic

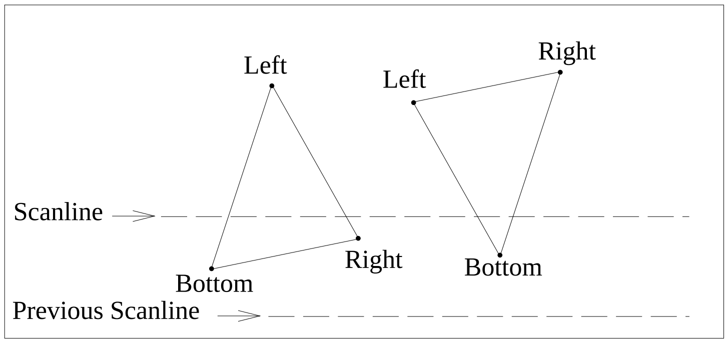


Figure 1: Figure of the two basic orientations of a triangle. 1.) Two vertices below scan-line and 2.) Two vertices above the scan-line.

orientations that a triangle can have with respect to the scan-line - two vertices below the scan-line or two vertices above the scan-line. There are actually twelve orientation cases since we do not know which of the three vertices are the bottom, left, and right. The bottom vertex is always below the scan-line. The left vertex is always above the scan-line. The right vertex is chosen such that by traversing the vertices of the triangle in counter-clockwise order, the vertices are visited in the order (right, left, bottom). In order to determine the bottom and right vertices (or left and right vertices for the second case), the slopes of the two edges that intersect the scan-line must be tested. In the case where two vertices are above the scan-line, an intermediate Y value is determined and saved that will signify a change in one of the two edges that are used. If the triangle falls into the case where two vertices are below the scan-line, then the intermediate Y value is set to the same value as the Y-bucket. After the active list is updated, the active triangles, which are not sorted in any particular order, must be processed into X-buckets.

2.2 X-buckets

There is an X-bucket for each pixel across the scan line which contains the triangles that become active starting on that pixel. A polygon is considered active for a pixel if the pixel, which is represented by integers, lies inside or exactly on the edge of the polygon. The X-buckets, defined in Table 2, must be built and maintained in visibility-order for correct color and opacity calculations, therefore, it has been implemented as a linked list.

Long	Float	Float	Float	Float	Float	Float	Float	Float	Int	Pointer
Polygon ID	Current Data Value	Current Depth Value	Left X Value	Right X Value	Left Z Value	Right Z Value	Left Data Value	Right Data Value	Max X Value	Next

Table 2: Data Structure for Active Polygon List in X direction. (Linked list)

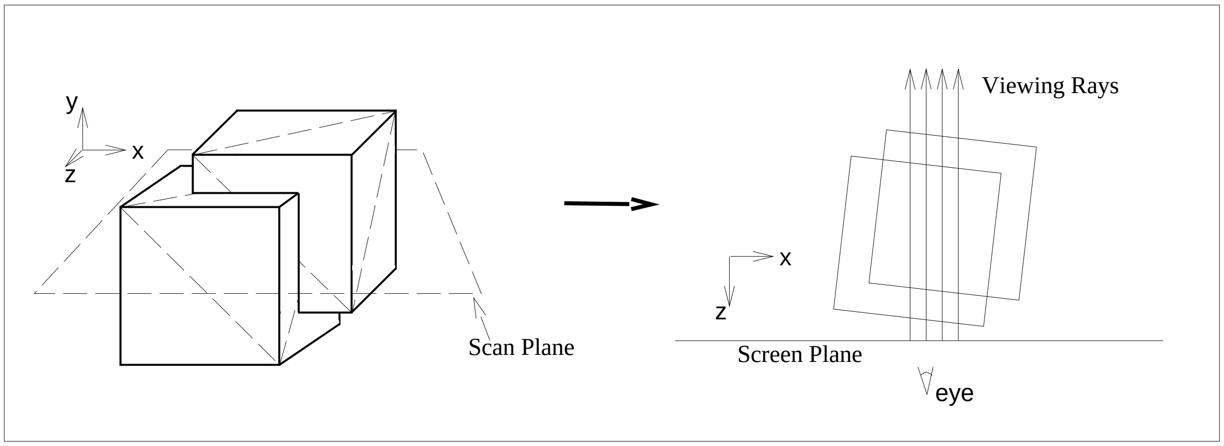


Figure 2: Figure of intersecting cell faces and their scan plane representation.

Figure 2 shows how intersecting cell faces are processed for a given scan-line. As the scan line is processed, an active list is maintained and updated for each pixel. The active list contains the data value and the depth for each triangle that contributes to that pixel and is sorted by depth. It can then be traversed to accumulate the color and opacity for that pixel. When calculating the X-bucket for a triangle, the X values for the pixel where the triangle becomes active and where it becomes inactive on the current scan-line must be determined by linearly interpolating the two active edges of the triangle. The code listed in Figure 3 determines which two of the three edges to use for a given scan-line.

2.3 Interpolation

During the course of updating the active lists, both in the Y and X direction, values for X, Z, and data are calculated through linear interpolation. The main requirement of the interpolation method is that the interpolated values for both end points must be exactly equal to the endpoints and never over shoot the end points by any amount. If the interpolated value of X did overshoot the endpoint by a small amount, and that amount resulted in X being greater that the next higher integer, the polygon would incorrectly contribute to an extra pixel in the final image. The equation below insures that triangles that share a vertex and do

```
if(scanline is below Y-intermediate)
   {
   use bottom->left and bottom->right
   }
else
   {
   if(left's Y value is greater than right's Y value)
      {
      use bottom->left and right->left
      }
   else
      {
      use left->right and bottom->right
      }
   }
```

Figure 3: Determining which edges to use for given scan-line.

not overlap in world space will not overlap at any point in this algorithm. The equation that we used for linear interpolation between vertices is

$$x = \frac{x_1}{(y_1 - y_0)}(y - y_0) + \frac{x_0}{(y_1 - y_0)}(y_1 - y)$$

This particular formula calculates x given the scan-line y and the two points of a line (x_0, y_0) and (x_1, y_1) that intersects the scan-line. This formula will give exact values at both end points, without over shooting and eliminates overlap between adjacent polygons. The order of calculation should never change throughout the algorithm to insure that two polygons which share an edge always get the same value of x along that edge for any given scan-line.

2.4 Pseudo code for three major components

```
DoYScan(screenHeight, screenWidth)
    {
    ConvertVerticesToScreenSpace();
    BuildYBuckets();
    for(scanline goes from 1 to screenHeight)
        {
        ActiveYList = UpdateActiveYList(ActiveYList, scanline);
        X_Buckets = BuildXBuckets(ActiveYList, scanline);
        DoXScan(X_Buckets, screenWidth);
        }
    }
```

Figure 4: Outer loop of Y scan algorithm.

Figure 4 is a simple pseudo code representation of the outer loop of the Y scan algorithm. Each scan line calls the *DoXScan* procedure which could be thought of as the inner loop. The *DoXScan* procedure, represented as pseudo code in Figure 5, processes the scan line from left to right. It calls the *DrawThisPixel* routine in Figure 6 which traverses the ActiveXList to determine the color and opacity for that pixel. Those polygons

```
DoXScan(X_Buckets, screenWidth)
    {
    for(X goes from 1 to screenWidth)
        {
        ActiveXList = UpdateActiveXList(ActiveXList, X, X_Buckets);
        DrawThisPixel(ActiveXList);
        }
    }
```

Figure 5: Inner loop of Y scan algorithm.

```
DrawPixel(ActiveXList)
    {
    while(not at the end of ActiveXList)
        {
        if(ActiveXList.Polygon is on a grid surface)
           {
           toggle inside/outside for that grid;
           }
        if(ActiveXList.Polygon is a clipping polygon)
           {
           toggle in/out for clipping region;
           }
        if((we are inside clipping region) and (inside a grid))
           {
           IntegrateColor(Depth to next polygon, data of this and next polygon);
           AddColorToThisPixel;
           }
        ActiveXList = ActiveXList->next;
        }
    }
```

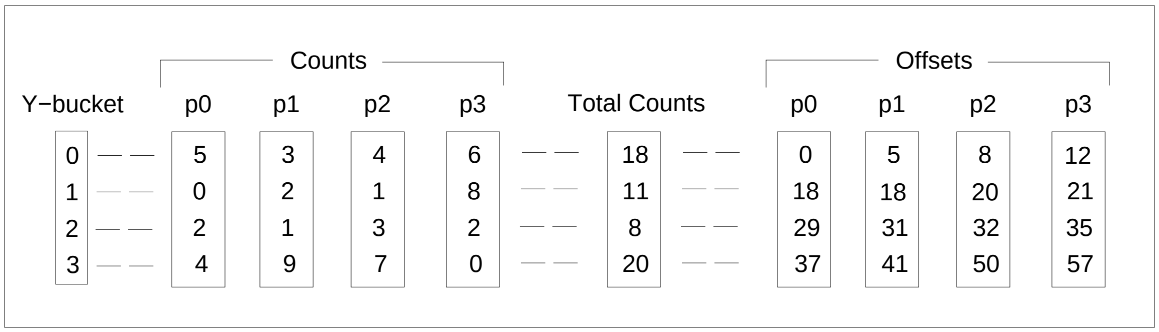
Figure 6: Draw a pixel routine.

that are on a surface of a grid (i.e. the three vertices have one grid index that is either 0 or max) will toggle a flag that keeps track of whether we are inside a grid or not. Similarly, those polygons that are part of the bounding box, or clipping region, will toggle a flag that controls whether we accumulate color or not.

3 Parallel Implementation

Watkin's scan-line algorithm, although it appears sequential, can be implemented to take advantage of a MIMD architecture. By breaking up the various sequential components of the scan-line algorithm into equal loads for each process, the algorithm can be almost entirely parallelizable. The only parts that are not easily parallelizable are the sections where the work is actually being broken up.

This algorithm has three primary components that are implemented in parallel. The first, is the transformation of vertices from world space to pixel space. The work for this part is easily sectioned for the number of processes that will be used. The second component is the generation of the Y-buckets. This can also be broken up, however it requires using a two-pass approach. During the first pass, each process is given an equal number of polygons to process. Each polygon either belongs in a Y-bucket, or is dropped because it does not cover a pixel in the drawing region. The Y-bucket value is kept for each polygon in a separate array of shorts called the *bucket_clip_info* array. In addition, if the polygon is discarded for any reason, an invalidation flag is set for that polygon in the *bucket_clip_info* array. This array is used by the second pass after the Y-bucket array has been sized properly. Also, a counter, which will be used later as a subscript, for that Y-bucket is incremented. After the first pass is completed, the subscripts that were generated by each process are combined to determine the total size of the Y-bucket array, and to determine the index where each Y-bucket will start. Since we know the Y-bucket counts for each process, each process is given the starting index into the Y-bucket array. During the second pass each process handles the same polygons, minus those that were invalidated. The single Y-bucket array is filled by all of the processes, by using the *bucket_clip_info* array that was previously produced.



	Counts					Offsets			
Y−bucket	p0	p1	p2	p3	Total Counts	p0	p1	p2	p3
0	5	3	4	6	18	0	5	8	12
1	0	2	1	8	11	18	18	20	21
2	2	1	3	2	8	29	31	32	35
3	4	9	7	0	20	37	41	50	57

Figure 7: Counters and offsets used by 4 processors in Y-bucket creation.

The third part involves the processing of each scan line in order from bottom to top. There are several ways of implementing this part in parallel. The first implementation of this part consisted of a critical section of code where a processor updates the current active list for the scan line, takes a copy of it, and then exits the critical section. After exiting the critical section, it builds the X-buckets and processes the scan line. The next available processor waits to enter the critical section and get to the next available scan-line. This implementation is not 100% scalable since it contains a critical section that will act as a bottle neck as more processors are added, however, it was fairly easy to implement, and is the method that we used. One way to eliminate waiting for the next scan-line to become available is to assign a region of sequential scan-lines to each processor. This will work if the volume is evenly distributed across the screen in the vertical direction. If it is not, then loading problems will arise resulting in one processor doing more work than the others.

One alternative to the first approach is to have the processor update the current scan line in small segments (100 triangles each) and after each segment is updated, signal to the next processor that the segment is ready. This way, all the processors can be updating their own scan-line and will only have to wait for the next segment. If all of the processors have approximately the same throughput, each processor will only have to wait for the first segment to be completed by the previous processor. Although this approach is not 100% scalable, it will reduce the time spent waiting for the previous scan-line to be completely updated. The communications between processes is more complicated in this approach and could cause delays if there is no effecient way to signal to other processes.

4 Measurements

This section describes our results of testing the algorithm on a large dataset (see Table 3). We discuss the size of the dataset, and its impact on speed and memory requirements, as well as the effects of using different screen sizes and image scaling factors. Results are presented in graphical form with an emphasis on showing the gains in speed for one to four processors.

The dataset was rendered with a spatial rotation of $(-90°X,\ -10°Y,\ 0°Z)$, and scale factors that ranged from 1.0 to 5.16. The speedup data for four processors in particular is not exact since the machine, an SGI Reality Engine II had four processors, which means that some time was spent by one or more processors on background processes. This introduced some variation in performance, but trends in speed can still be observed for four processors.

Grid 1	Grid 2	Grid 3	Grid 4	Grid 5	Grid 6	Grid 7	Grid 8	Grid 9
88x39x60	92x57x28	98x77x48	24x33x23	33x33x22	25x40x21	75x37x33	14x25x30	53x23x50
205920	146832	362208	18216	23958	21000	91575	10500	60950

Table 3: Grid dimensions and data points for NASA space shuttle.

Scale -	1.00	1.20	1.44	1.73	2.07	2.49	2.99	3.58	4.30	5.16
Active Polygons (Million)	0.865	0.995	1.153	1.346	1.495	1.660	1.836	2.022	2.192	2.360
Single processor	54.39	66.87	83.52	98.88	112.55	125.34	138.02	149.15	161.75	173.80
Two processors	30.66	37.46	46.34	56.10	63.30	71.32	77.67	84.33	90.17	97.57
Three processors	21.30	26.19	32.58	38.82	44.10	49.36	54.24	58.26	62.70	68.15
Four processors	16.85	20.16	25.06	30.10	34.14	37.89	41.62	44.89	49.15	53.03

Table 4: Elapsed time comparisons (in seconds) on SGI Reality Engine II with four 150 MHz processors, 256MB memory. (NASA space shuttle data set)

4.1 Dataset

The dataset that we chose for our experiment was the NASA space shuttle [JS90], which is made up of nine intersecting curvilinear grids and 941,159 total data points. The data includes the position of every point in each grid along with values for its density, energy and momentum vector. We chose the space shuttle because it has multiple grids, and represents a large-sized amount of data.

4.2 Timing analysis

Transforming the point locations from world space to screen space was measured separately from the other sections of the algorithm. The average total CPU time to transform the shuttle points to screen space was 1.25 seconds and did not increase more than a total of 0.02 seconds for four processors. Figure 9 shows the

Figure 8: Shuttle images with transfer function. (scale = 1.0, 1.73, 2.99, 5.16)

speedups for transforming the points to screen space for multiple processors. The results show that this part of the algorithm is scalable with a greater than 3.5 speedup for four processors.

Speedup factors for creating the Y-buckets are shown on Figure 9. CPU times for creating Y-buckets varied from 16.8 seconds to 18.9 seconds. The time needed to create Y-buckets increases slightly (less than 10% change from scale = 1.0 to scale = 5.16) when the number of active polygons increases. An active polygon is one that is not clipped from the drawing area due to size or position on the screen. This increase in time is due to the fact that the Y-bucket array grows in size and active polygon id's must be stored in the array.

Speedup factors for generating the scan-lines from the Y-buckets are shown on Figure 9. These speed

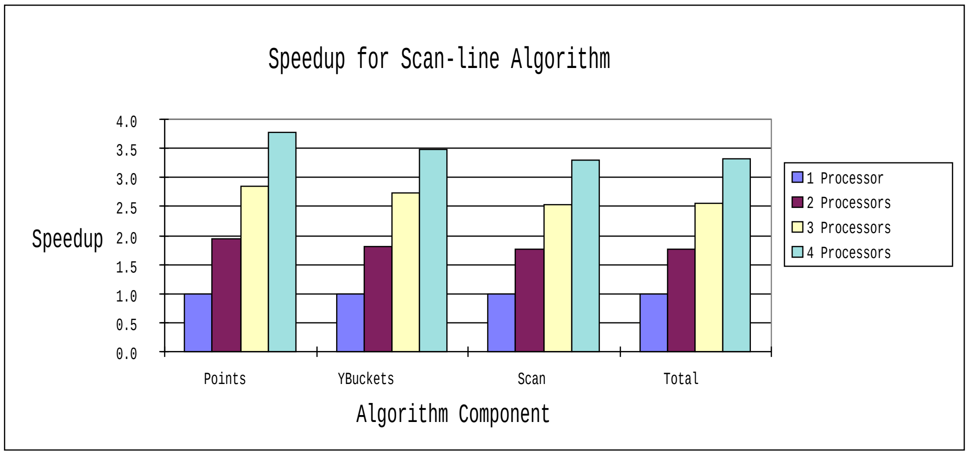


Figure 9: Speedup of algorithm for shuttle (scale = 3.58).

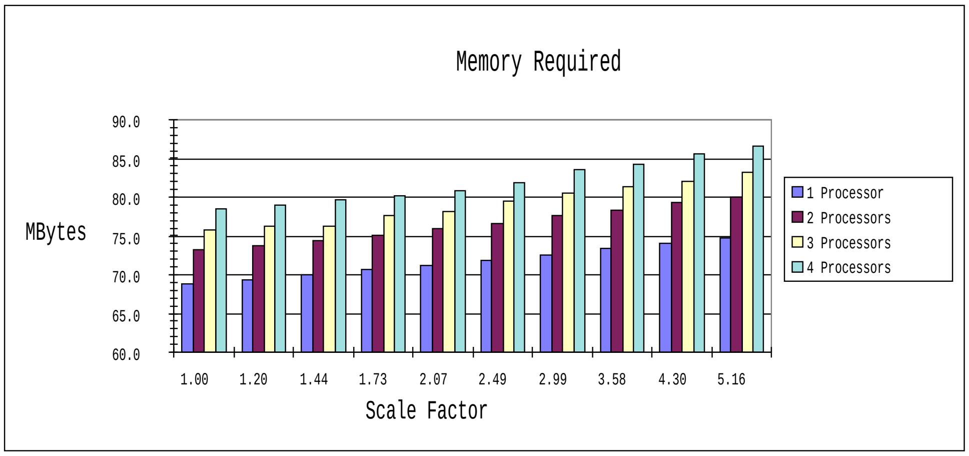


Figure 10: Amount of memory used by the program.

ratios are not as high as those for transforming points to screen space or building Y-buckets because of the critical code mentioned in section 3, however, the increase in speed is still significant for four processors.

The total elapsed times to render each picture are presented in Table 4. Overall speedup factors are in Figure 9. It is clear that although 100% scalability is not achieved, there is a significant speedup (approx 3.25) for four processors.

4.3 Memory

Memory use is an important factor in measuring the efficiency of an algorithm. Figure 10 shows the memory (in Megabytes) used by the program for a range of scale factors and multiple processors. This information was recorded by keeping statistical information associated with all calls to *malloc()* and *free()*. Most of the memory was used for storage of the following major items:

- Space shuttle grid - 33.9 Mbytes.
- Transformed points - 11.3 Mbytes.

- Frame buffer (R,G,B,Alpha,Composite) - 5.0 Mbytes
- *bucket_clip_info* array - 11.0 Mbytes

The rest of the memory is used by the program to keep track of the polygons that it is rendering. It can be seen from Figure 10 that memory usage increases with the number of processors. This is due to the fact that each processor is rendering a scan line and needs the memory to hold the data structures for that scan line, such as the active polygon list and the X-buckets. Also, as the number of active polygons increase, the memory needed by active lists and buckets increases as well.

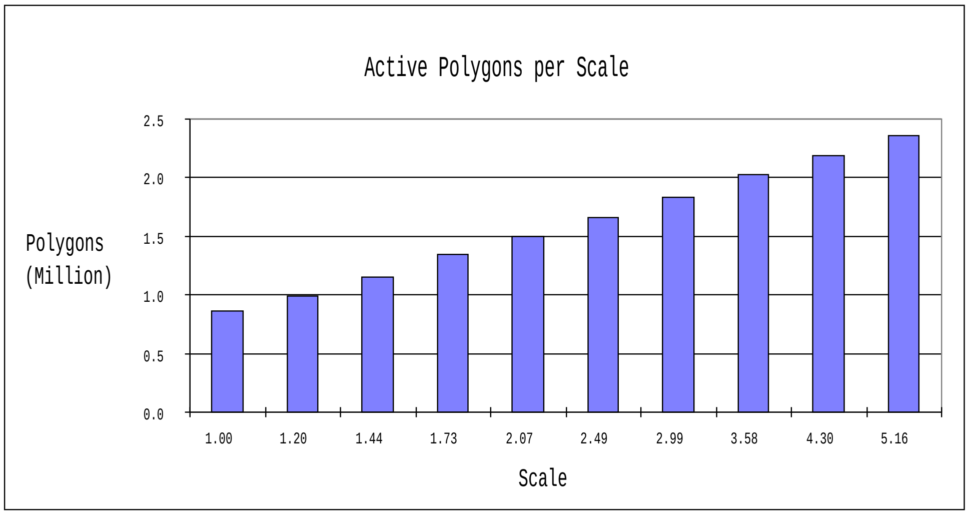


Figure 11: Number of active polygons for different scale factors.

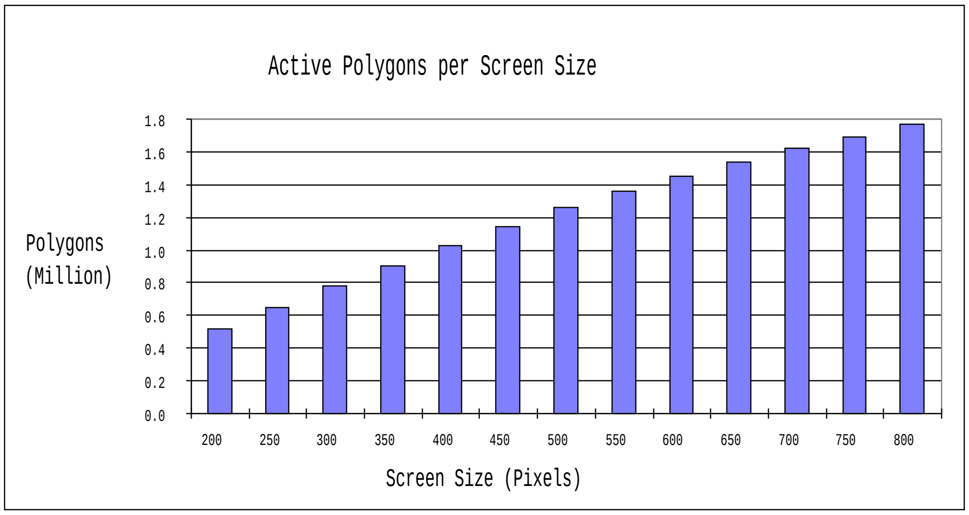


Figure 12: Number of active polygons for different screen sizes.

4.4 Other factors

It is clear from Figures 11 and 12 that as the window size is increased or as the scale is increased, the number of active polygons is increased as well. As the scale is increased, there is an upper limit to the number of active polygons which is less than the total number of polygons in the data set because at high scale factors, more of the volume is projected off of the screen and the polygons that project off of the screen are discarded by the algorithm during Y-bucket creation. Increasing the window size will allow very small polygons, that may have been discarded because they did not cross a pixel, to become active since they are now projected onto a larger portion of the screen.

4.5 Comparison with cell projection

A blunt fin dataset [HB85], which is smaller (40x32x32 = 40,960 points) and only contained a single grid, was used to compare rendering times of the scan-line algroithm with times from a cell projection algorithm called QP [VGW93]. Both algorithms were run on an SGI Reality Engine II with four 150MHz processors, however, only one processor was used for this test.

Program	Scale = 1.0	Scale = 2.0
QP	17.01 sec	17.01 sec
Scan-line	22.07 sec	49.26 sec

Table 5: Scan-line and Qp times for Blunt Fin data set.

Figure 13 shows images produced by the QP program (on top) and by the scan-line program (on bottom). Table 5 shows the CPU times for each program. It is clear that QP is the faster of the two programs, but at scales of 1.0 or less, the scan-line algorithm is competitive. The increase in time for the scan-line algorithm is offset by its superior image quality. Also, the scan-line algorithm handles intersecting grids where QP can not.

5 Perspective Rendering

During the initial testing of the program, we observed that the depth calculation between polygons was losing accuracy due to the fact that we were using a standard projection matrix which transforms the viewable world space into a unit viewing volume [McL91]. This meant that the Z values of the vertices were being transformed to values that were very close to each other, which introduced substantial floating point errors and resulted in some ordering inconsistencies. The problem was corrected by using a different projection matrix (described below). This matrix preserves the Z values by transforming them directly into pixel space, along with the X and Y values.

The volume being viewed is inside a user definable visible bounding box (Vbb), of diagonal d (see Figure 14). The Vbb is first centered at (0,0,0) in world space, then it is rotated by the user rotations and translated back by e_o. Then the user translates are applied, with the restriction that

$$z_t + \tfrac{1}{2}d + NearClip \leq e_o$$

(our implementation uses $NearClip = .05e_o$), so that the denominator $(e_o - z_t - z_v)$ is positive. Using similar triangles, we get the following relationships

$$e_s = \frac{s \cdot h_s \cdot e_o}{d},$$

Figure 13: Blunt fin images with transfer function. (scale = 1.0, 2.0) QP (top) Scan-line (bottom)

$$\begin{aligned}\frac{x_s}{e_s} &= \frac{x_v + x_t}{e_o - z_t - z_v},\\ \frac{y_s}{e_s} &= \frac{y_v + y_t}{e_o - z_t - z_v},\\ \frac{z_s + z_{so}}{e_s} &= \frac{z_v + z_t}{e_o - z_t - z_v} = 1 - \frac{e_o}{e_o - z_t - z_v}.\end{aligned}$$

Conversion to screen space is

$$x_s = \frac{e_s(x_v + x_t)}{e_o - z_t - z_v}, y_s = \frac{e_s(y_v + y_t)}{e_o - z_t - z_v}, \text{ and}$$

$$z_s = \frac{e_s(z_v + z_t)}{e_o - z_t - z_v} - z_{so} = \frac{e_s \cdot z_v \cdot e_o}{(e_o - z_t)(e_o - z_t - z_v)}.$$

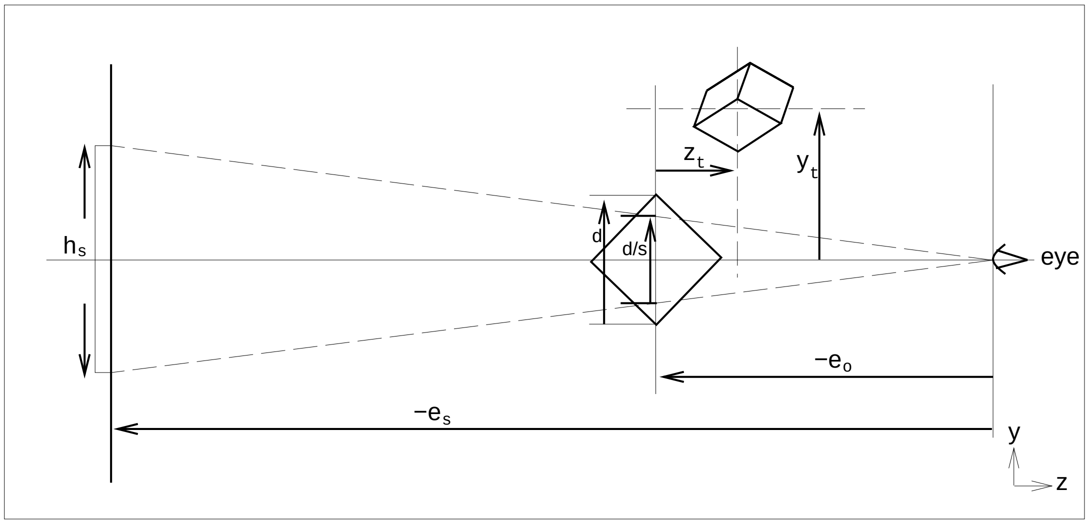


Figure 14: Figure of perspective view - eye is at (0,0,0) world space. Volume is projected onto screen at left.

Symbol	Definition	Units
h_s	screen height	pixels
e_s	distance from the eye to the screen	pixels
e_o	distance from the eye to the volume origin	world units
d	diagonal of the volume's visible bounding box	world units
s	user's scale factor	pure number
z_t	translate in z direction	world units
y_t	translate in y direction	world units
x_t	translate in x direction	world units
z_v	vertex value of z	world units
y_v	vertex value of y	world units
x_v	vertex value of x	world units
z_s	transformed vertex value of z to screen	pixels
y_s	transformed vertex value of y to screen	pixels
x_s	transformed vertex value of x to screen	pixels
z_{so}	offset (defined below)	pixels

Table 6: Symbols used in perspective calculation.

The variable z_{so} is an offset to the screen position and is chosen so that the center of the Vbb is mapped to 0 in z_s. (The center of the volume might map anywhere, as it is not necessarily within the Vbb.) z_{so} is represented by

$$z_{so} = \frac{e_s \cdot z_t}{e_o - z_t}.$$

To invert the z_v-to-z_s mapping, just solve for z_v. The following equations solve for the depth between two values $(z_{v2} - z_{v1})$.

$$e_o(z_s + z_{so}) - (z_t + z_v)(z_s + z_{so}) = e_s(z_v + z_t)$$

$$z_v + z_t = \frac{e_o(z_s + z_{so})}{e_s + z_s + z_{so}}$$

$$
\begin{aligned}
e_s + z_{so} &= e_s(1 + \frac{z_t}{e_o - z_t}) = \frac{e_s \cdot e_o}{e_o - z_t} \\
z_{v2} - z_{v1} &= \frac{e_o[(z_{s2} + z_{so})(e_s + z_{s1} + z_{so}) - (z_{s1} + z_{so})(e_s + z_{s2} + z_{so})]}{(e_s + z_{s2} + z_{so})(e_s + z_{s1} + z_{so})} \\
&= \frac{e_o[e_s(z_{s2} - z_{s1})]}{(e_s + z_{so} + z_{s2})(e_s + z_{so} + z_{s1})} \\
&= \frac{e_o \cdot e_s(z_{s2} - z_{s1})}{(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s2})(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s1})}
\end{aligned}
$$

By using this approach, along with the fact that the depth increases as the angle (θ) from the eye to the pixel on the screen increases, the actual depth can be calculated as

$$Depth = \frac{z_{v2} - z_{v1}}{cos(\theta)}$$

6 Conclusions

The renderer described in this paper allows rendering of large multiple intersecting curvilinear grids without the use of expensive graphics hardware. The design allows rendering of any volume data that can be represented by triangles. Factors such as screen size and scale can effect the time needed to render the volume, which provides a natural variable resolution feature. The renderer is parallelizable and measurements show that elapsed time can be greatly reduced without greatly increasing memory requirements. Accurate depth calculation can be achieved by using the projection method introduced in Section 5.

References

[Cha93] Judy Challinger. *Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids.* PhD thesis, University of California, Santa Cruz, December 1993.

[Gar90] Michael P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, December 1990.

[GP93] Christopher Giertsen and Johnny Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

[HB85] Ching-Mao Hung and Pieter G. Buning. Simulation of blunt-fin-induced shock-wave and turbulent boundary-layer interaction. *J. Fluid Mechanics*, 154:163–185, 1985.

[JS90] F.W. Martin Jr. and J.P. Slotnick. Flow computations for the space shuttle in ascent mode using thin-layer navier-stokes equations. *Applied Computational Aerodynamics, Progress in Astronautics and Aeronautics*, 125:863–886, 1990.

[Luc92] Bruce Lucas. A scientific visualization renderer. In *Visualization '92*, pages 227–233. IEEE, October 1992.

[McL91] Patricia McLendon. *Graphics Library Programming Guide.* Silicon Graphics, Inc., Mountain View, CA, 1991.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (ACM Workshop on Volume Visualization)*, 24(5):27–33, December 1990.

[RW92] Shankar Ramamoorthy and Jane Wilhelms. An analysis of approaches to ray-tracing curvilinear grids. Technical Report UCSC-CRL-92-07, UCSC, University of California, CIS Board, Santa Cruz, CA, January 1992.

[ST90] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, December 1990.

[Use91] Sam Uselton. Volume rendering for computational flid dynamics: Initial results. Technical Report RNR-91-026, NAS-NASA Ames Research Center, Moffett Field, CA, 1991.

[VGKW95] Allen Van Gelder, Kwansik Kim, and Jane Wilhelms. Hierarchically accelerated ray casting for volume rendering with controlled error. Technical Report UCSC-CRL-95-31, University of California, Santa Cruz, UCSC CIS Department, Applied Sciences Building, Santa Cruz, CA 95064, 1995.

[VGW93] Allen Van Gelder and Jane Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. Technical Report UCSC-CRL-93-02, University of California, Santa Cruz, UCSC CIS Department, Applied Sciences Building, Santa Cruz, CA 95064, 1993. (extended abstract in *Proc. IEEE Visualization 93*, Oct. 1993).

[Wat70] G.S. Watkins. *A Real Time Visible Surface Algorithm.* PhD thesis, University of Utah, Salt Lake City, June 1970.

[WCA$^+$90] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct volume rendering of curvilinear volumes. *Computer Graphics*, 24(5):41–47, December 1990. Special Issue on San Diego Workshop on Volume Visualization.

[Wil92a] Peter Williams. Interactive splatting of nonrectilinear volumes. In *Visualization '92*, pages 37–44. IEEE, October 1992.

[Wil92b] Peter Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[Wil93] Jane Wilhelms. Pursuing interactive visualization of irregular grids. *The Visual Computer*, 9(8):450–458, 1993.

[WVG91] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):275–284, 1991.

[WVG94] Jane Wilhelms and Allen Van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *ACM Workshop on Volume Visualization 1994*, Washington, D.C., October 1994. See also technical report UCSC-CRL-94-02.