A Parallelized Scanline Algorithm for 3D Rendering

Colin Bulthaup
6.846 Final Project
Professor Anant Agarwal

1. Abstract

For my final project in 6.846, (Parallel Processing,) I chose to implement a 3-dimensional renderer on the Alewife machine. I first created a sequential version of the renderer which ran under the X Windows environment, and I then parallelized my code to run on the Alewife. This code implemented a very fast scanline algorithm which quickly and efficiently partitioned the problem into smaller tasks, thus making it easily parallelizable.

2. Introduction

2.1 Motivation

3D Rendering has become one of the most important uses of computers. Among the many uses for this technology are: movies like Toy Story and Jurassic Park, games like Quake and Tomb Raider, military and commercial simulators, data representation, engineering and architectural modelling, virtual communities, etc. The applications are almost limitless, since everything that can be done in the real world may some day be emulated on a computer.

The current bottleneck on the development of 3D intensive applications is not a lack of ideas, but rather a lack of power. Current processors are insufficient for the demands of 3D rendering. A single high quality rendering of a complicated scene can take on the order of days to complete. This is obviously less than ideal. Luckily though, the rendering problem has a great deal of potential for parallelization. The rendering of a single pixel of a scene is independent of any other pixel and thus a scene can be chopped up and divvied out to multiple processors. The programmers at Pixar, (the makers of Toy Story,) made use of this fact by dividing up their scenes between hundreds of different Sparc workstations, thus achieving near linear speedup.

There is a problem though with rendering across a cluster of workstations. Namely, the communication latency between different processors is so large that the problem size that each processor deals with must be made extremely large in order to balance the communication vs. computation time. Due to this fact, clusters of workstations are ideal for rendering extremely large and complicated scenes that take a very long time to complete. A different solution must be found though if we are interested in doing real-time rendering of smaller scenes, (on the order of 30 frames per second, as compared to 1 frame every day.)

An SMP with short latency between processors is an ideal machine for building such a real-time renderer.

2.2 Design Choices

There are many different methods and styles of rendering. At one end of the spectrum are simple polygonal renderers with very simple illumination models. On the other end are complicated raytracing renderers with radiosity and very complicated illumination models.

In addition, there are also many different algorithms that can be used to render a scene. Everything from the simplistic painters algorithm to a complicated BSP tree oriented algorithm.

For this implementation, I chose to use a relatively simplistic illumination model for the sake of speed that can be improved upon in later versions. I also chose to use a scanline algorithm that performs well and lends itself to parallelization.

3. Basics of the Renderer

3.1 Scene Description

To make sure that the renderer would run as quickly as possible I chose to limit myself to only non-intersecting triangles. This really isn’t that much of a problem since all intersecting or non-intersecting polygons can ultimately be divided up into non-intersecting triangles.

The scene therefore consisted of an array of faces. Each face contained three vectors describing the points of a triangle, an additional vector describing the normal of the face, and a floating point value for the intensity of light reflected off of the face. This intensity value was calculated using the following lighting model.

3.2 Lighting

The lighting model that I am using is quite simple, yet accurate enough for my purposes. The intensity of each face is the sum of the intensity of the ambient light plus the intensity of the diffused light.

$$If = Ia + Id$$

The intensity of the diffused light is in turn the intensity of the light source times the cosine of the reflection angle. Or, in terms of the vectors involved:

$$I_d = I_p k_d (\bar{N} \cdot \bar{L})$$

Making the final equation:

$$I_f = I_a + I_p k_d (\bar{N} \cdot \bar{L})$$

3.3 Transformations

In order to transform points in 3-space, it is first necessary to convert each vector to homogeneous coordinates. In order to do this, we simply add a 4th term to each vector. This 4th term will represent the scaling factor of the vector, and in order to convert from homogeneous coordinates back into 3D coordinates we will need to divide all the other terms of the vector by the 4th term. Thus the vector (1, 2, 3) will become instead (1, 2, 3, 1).

After homogenizing the vectors, translation, rotation, scaling, shearing, etc. can all be accomplished with simple matrix multiplication. The matrices for translating, rotating around the z axis, and scaling are as follows:

$$T(dx,dy,dz) = \begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$Rz(\Theta) = \begin{vmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$S(sx,sy,sz) = \begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

It is obviously possible to multiply these matrices by any other matrix to produce more complicated transformations.

3.4 Translating from Object to Image Space

In order to convert a set of 3D vectors into the 2 dimensional viewing screen we apply a Perspective Matrix to each point in the scene. This Perspective Matrix is influenced by several different factors: the position of the viewer, the size and location of the viewing window, the vector representing up, etc.

If we multiply this Perspective Matrix by all of the other Transformation Matrices that we are applying to our scene then we will have a single matrix which does everything. I will call this the Final Matrix.

The Pipeline for converting 3D faces in Object space into 2D faces in Image space is therefore as follows:

- Convert all vectors to homogeneous coordinates.
- Multiply each vector by the Transformation Matrix.
- Calculate the new normal for each face.
- Calculate the intensity of each face using the illumination model.
- Multiply each vector by the Perspective Matrix.
- Convert back to 3D coordinates.

4. The Sequential Scanline Algorithm

4.1 Sort in Y

The first step of the scanline algorithm is to take all of the 2D image faces and sort them using a linear time bucket sort. The maximum y value of a face and the minimum y value of that face are both entered into the sorted array in order to signify the beginning and ending edge of each face. This sort was implemented using an array of linked-lists.

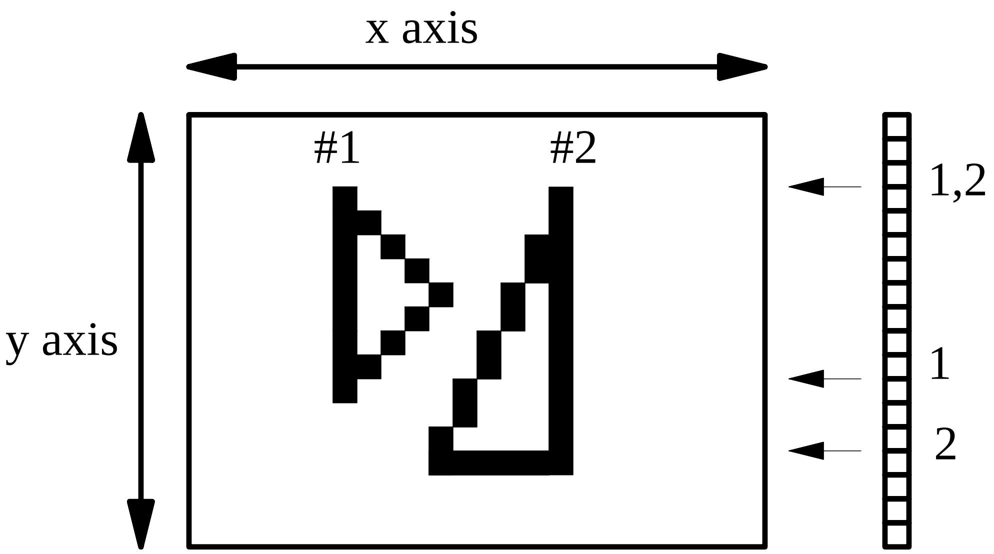


Diagram of the Y-sort

4.2 Sort in X

The next step is to travel down the Y sorted list and add and remove faces from a linked list of current faces. The first time a face appears it should be added to the list, and the second time it appears it should be removed from the list.

For each scanline, the current list of faces should be sorted in the x direction using a similar bucket sort. The minimum x value of a face at that scanline should be sorted along with the maximum x value at that scanline.

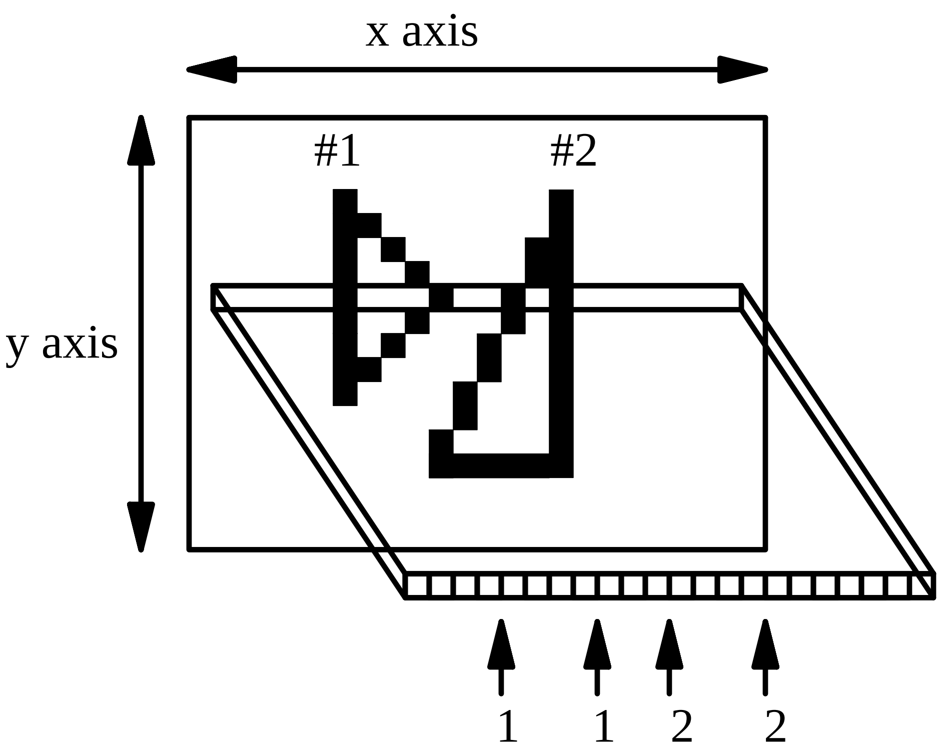


Diagram of the X-sort

4.3 Sort in Z

The final step then is to travel down the x-sorted list of faces and add and remove them from a second list of current faces. The one difference between this list and previous lists is that the insertion into and removal from the linked list will be controlled by the current z of each face. Thus as we travel down the x-sorted list we will have a z-sorted stack of current faces for each pixel. It will then be a simple matter to color each pixel based on the intensity of the face at the top of the z-sorted stack.

For a simple solid shading model, the entirety of each face will be colored the same intensity. For a more complicated shading model like Gourad or Phong we would interpolate the color value for a given pixel based on the intensities at the three corners.

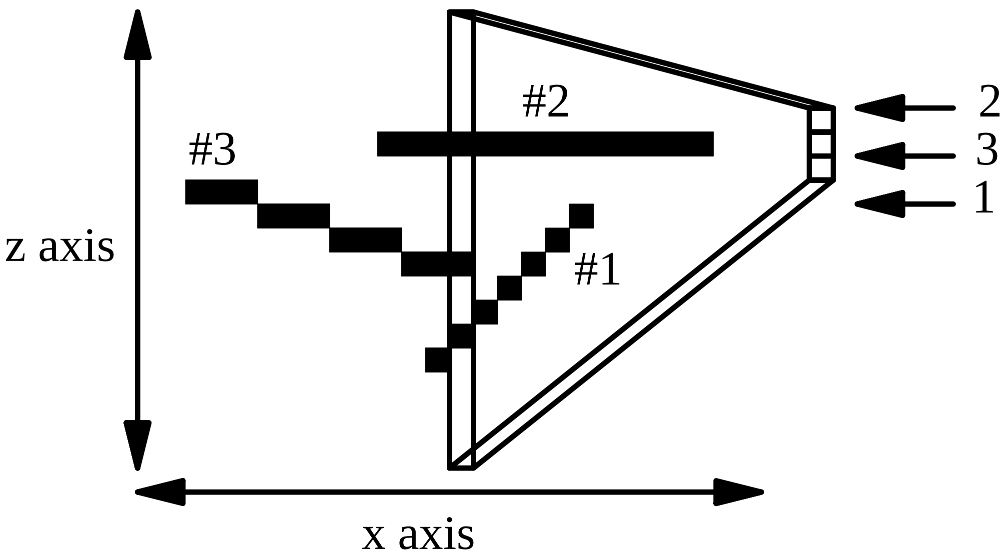


Diagram of Z-sort

These sorts can be represented by the following snippet of pseudocode.

```
for (each face) {
     Y-sort
}
for (each scanline) {
     Add Y-sort[scanline] to current y-list of faces
     for (each face in current y-list) {
          X-sort
}
     for (each pixel in scanline) {
          Add X-sort[pixel] to current x-list of faces
     }
}
```

4.5 Optimization Issues

There are several other very important issues that need to be looked at when writing a fast real-time renderer. First of all, the face structures that are being passed around should be pared down as small as possible with only the minimum required information included. This reduces the time required for memory accesses and also reduces the communication time in a parallel implementation.

Secondly, the inner loop of the scanline algorithm has to be made as fast as possible, (even at the expense of longer preparation time.) This inner loop could conceivably be called, in the worst case, NPOLYS*WIDTH*HEIGHT times, and thus any speedups made to this portion of the code will have significant effects. One way of speeding up the inner loop is to avoid all function calls. The expense of pushing state onto the stack and branching to a different procedure is very large within the inner loop and must be avoided. Another obvious optimization is to reduce the number of multiplies and divides within the inner loop, (hopefully to zero.) In addition, depending on the processor that the renderer is running on, it might be desirable to remove all floating point operations and deal solely with fixed points integers.

There are several other small optimizations that can be made, such as using lookup tables for trigonometric functions, but these first two optimizations are the most important. In addition, other large scale optimizations like using BSP trees to partition the faces are also possible but they are beyond the scope of this paper.

5. The Parallel Scanline Algorithm

5.1 Target Machine

The parallelized version of the renderer was designed for the Alewife Parallel Processing machine and was coded in the Alewife Parallel C. The renderer was tested using the NWO Alewife Simulator and was scaled up to 8 processors.

5.2 Fine Grain Parallelization

There are a number of fine grain parallelizations that could have been made to the sequential renderer code. Most of these would have been modifications to the vector and matrix multiplication code, but since all of the vectors and matrices in the renderer code are of order 4x4 or less this optimization did not seem very significant.

5.3 Coarse Grain Parallelization

The much more significant optimization involved coarse grain parallelization of the algorithm. By dividing up the faces between processors and by allocating the scanline processing among the processors it was possible to get very significant speedups.

There are two main arrays of faces that are important to the renderer. One of these arrays is the array of real_polygons which represent all of the faces in 3D space, and the other is the array of image_polygons which represent all of the faces in 2D image space. These two arrays were striped across all of the processors with each processor getting MAX_POLYS/NPROCS different faces. These arrays were filled with test polygons by a do_in_parallel call to init(), and in the official version of the renderer they would have been loaded with the complete scene description.

In order to render a scene, the parallel version of the program would go through several steps:

- Update the clock variable to indicate the passage of time in the animation
- Start the update_everything() procedure on all of the processors.
- Within the update_everything() procedure each processor would first update the transformation matrix based on the new clock value, second update all of the normals of each face owned by the processor based on the new transformation matrix, third update all of the light intensity values for each face based upon the new normals, and fourth sort the polygons owned by the processor into the y-sorted array.

- Start the render_image() procedure on all of the processors.

- Within the render_image() procedure each processor would begin spinning on a scanline variable which represented the currently available scanline. As soon as it got the lock on the scanline variable, a processor would update the current_y faces which represent the faces visible on the current scanline. It would then make local copies of the current_y faces and increment and unlock the scanline variable. From there it would do an x-sort of the faces and begin travelling down the scanline in a manner identical to the sequential program. As soon as it finished with the current scanline it would go back to spinning on the scanline variable and wait to repeat the process.

- Finally, if a process gets a lock on the scanline variable and finds that the scanline is greater than the height then it will unlock the scanline variable and return.

- As soon as all processors return from the render_image() procedure then we know that the image is done being drawn.

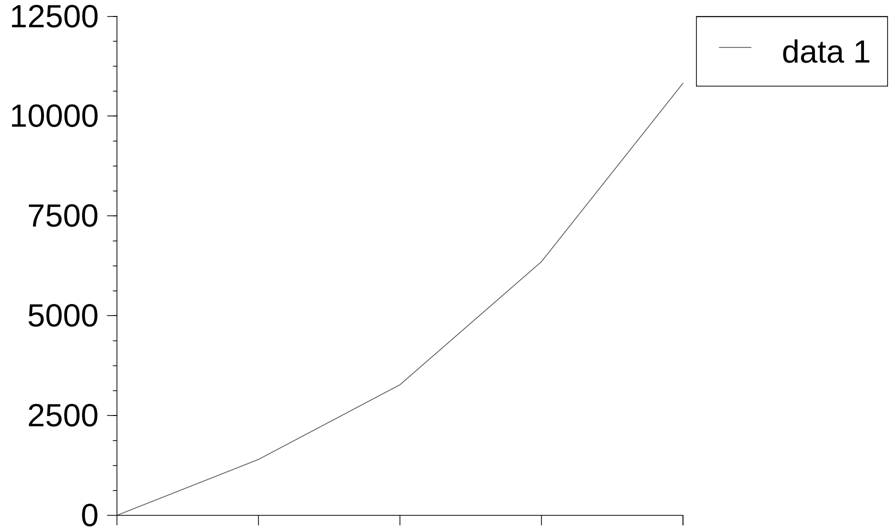
5.4 Expected Results

I expected to find that the parallelized version of the scanline algorithm would perform significantly better with larger image sizes and would perform worse as the density of faces to image size increased. This is due to the setup cost associated with beginning calculation of a scanline. The communication required to update and copy the shared faces is significant and would become a more serious problem as the number of faces increased. I also expected to find an ideal number of processors for a given configuration of number of faces and image size. More or less than this ideal number would cause worse performance. Therefore, the problem of rendering an image does not scale ideally, but for a given scene with given specifications it should be possible to find the ideal number of processors.

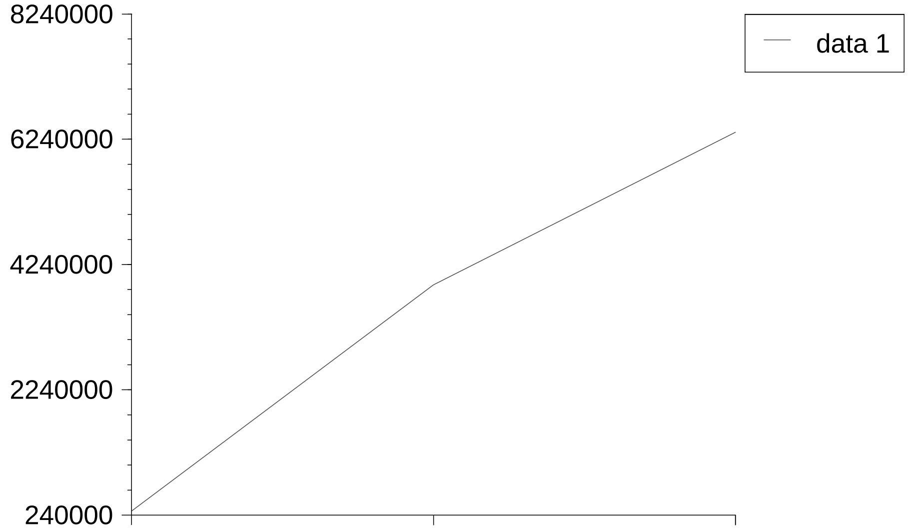
6. Results

6.1 Sequential Analysis

For the sequential version of the program I found there to be a slight non-linear growth in the time taken to render an image as the number of faces increased. I believe that this problem is caused by inefficiencies in the way that I was doing the z-sorting of the faces. After correcting this problem, it should be possible to get the growth in time to be linear. (The diagram that follows is for a 200x200 image with increasing number of faces.)



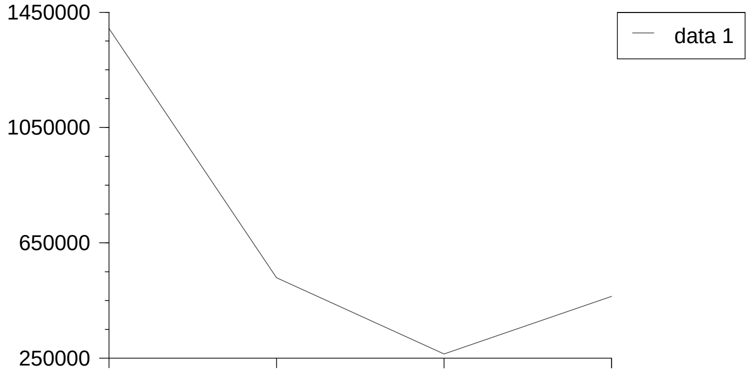
I also observed though that the time it took to complete a render with respect to the image size grew only at log(# faces). This bodes well for the performance of the system when dealing with extremely large images, (like immersive simulators.) (The diagram that follows was plotted on a logarithmic scale and was for a 50 face problem at increasing image sizes.)



6.2 Parallel Analysis

The parallel analysis is in agreement with my expected results. The ratio between number of faces and image size significantly effected the performance of the parallelized version of the renderer. There was indeed an ideal value for the number of processors for a particular configuration of image size and number of faces.

This chart shows the time to render a 100x100 image with 32 faces with increasing numbers of processors.

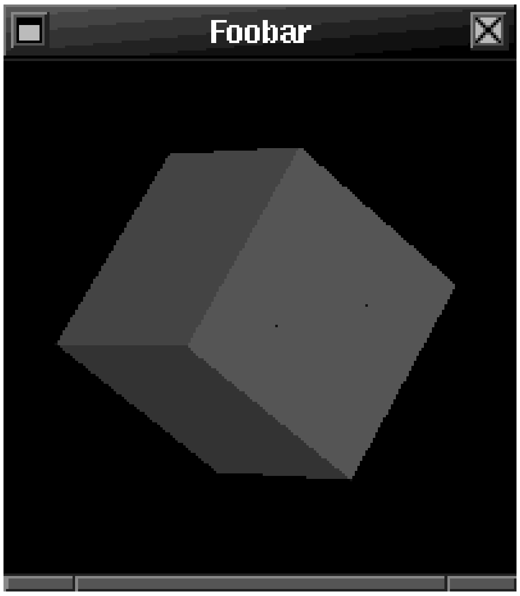


7. Conclusions

There are many different conclusions that can be drawn from this project:

First and foremost is that the scanline algorithm works and it is possible to render scenes in real-time.

Secondly and just as important is that it was possible to parallelize the renderer and get very significant improvements in speed. This bodes well for large scale rendering applications implemented on parallel processing machines.



Third, the implementation of the z-ordering needs some work, but other than that the entire rendering application scales linearly with number of faces and even better with image size.

Fourth, there is an ideal system configuration for each set of scene parameters, (number of faces and image size.) On systems with extra processors, it might be possible to begin rendering multiple scenes at the same time by partitioning the machine. This is something to look into for further research.