A PC-BASED PHOTOGRAPHIC-QUALITY IMAGE GENERATOR FOR FLIGHT SIMULATION

Izidor C. Gertner
George Wolberg
Department of Computer Science
City College of New York New York, NY 10031

George A. Geri
George R. Kelly
University of Dayton Research Institute
Higley,Arizona 85236-2020

Byron J. Pierce
Melvin Thomas
Elizabeth L. Martin
Air Force
Armstrong Laboratory
Mesa, Arizona 85206

ABSTRACT

Conventional image generation techniques rely largely on polygon rendering techniques. We describe here a system that uses off-the-shelf hardware to realize high-end image generation. We have developed a prototype image generator based on two Intel i860 processors and a host 486-PC. This hardware performs perspective transformations, clipping, and texture mapping. Parametric surfaces are generated by fitting either a bilinear or bicubic polynomial to standard Defense Mapping Agency (DMA) terrain height data. Real-time texture mapping algorithms are then used to place realistic textures, obtained from real-world photographs, onto the terrain height map. In our implementation, a multiresolution image pyramid is used to generate properly filtered images on demand at the resolution required by the viewing geometry. A wide range of terrain data approximations is used depending on altitude. Coarse (fine) approximations are implemented for high (low) altitude flight. A multiresolution terrain pyramid is used to achieve this approximation. This pyramidal approach is embedded into our real-time texture mapping system with the use of an incremental scanline algorithm. The current prototype can generate a 256 x 256 x 8-bit image at 15 frames/second using only two i860 processors, and the algorithms scale sub-linearly with the number of processors.

ABOUT THE AUTHORS

Izidor Gertner earned his Ph.D in Electrical Engineering from the Technion-Israel Institue of Technology. He has performed extensive research on fast algorithms and parallel computation. **George Wolberg** received the Ph.D degree in Computer Science from Columbia University in 1990, and was awarded an NSF Presidential Young Investigator Award in 1991. His research interests include image processing, computer graphics, and computer vision. **George A. Geri** received a Ph.D in Physics from Rensselaer Polytechnic Institute. His current research interests include human vision and the perceptual evaluation of efficient imagery. **George R. Kelly** graduated from the University of Saskatchewan with a B.S. degree in Engineering Physics. He has eight years of experience in the development of high resolution helmet mounted displays for flight simulator training devices. **Byron J. Pierce** received a Ph.D in Psychology from Arizona State University. He is currently on exchange to the Defense and Civil Institute of Environmental Medicine in North York, Ontario and is conducting behavioral research on spatial perception and 3-D graphical displays. **Melvin Thomas** received an M.S. degree in Physics from Northern Arizona University. His current interest is the development of low-cost displays for flight simulation. **Elizabeth L. Martin** received her Ph.D from University of Arizona. Her research interests include all aspects of aviation training with emphasis on visual simulation.

A PC-BASED PHOTOGRAPHIC-QUALITY IMAGE GENERATOR FOR FLIGHT SIMULATION

Izidor C. Gertner
George Wolberg
Department of Computer Science
City College of New York /
CUNY
New York, NY 10031

George A. Geri
George R. Kelly
University of Dayton
Research Institute
Higley, Arizona 85236-2020

Byron J. Pierce
Melvin Thomas
Elizabeth L. Martin
Air Force
Armstrong Laboratory
Mesa, Arizona 85206

1. INTRODUCTION

Real-time flight simulation has traditionally required expensive graphics workstations. The goal of this work is to exploit recent advances in processor and memory speeds to design a high-end PC-based multiprocessor image generator for flight simulation. By matching computer architecture with the algorithms best suited for this application, a total system can be designed at far less cost than those commercially available. Furthermore, the use of off-the-shelf components will permit the system to benefit from the economies of scale for processor and memory technologies. The principal application targeted here is low-altitude flight over a largeterrain. Several key tasks must be addressed for this application: texture mapping, multi-resolution terrain and image data, high-quality antialiasing, high-speed geometry pipeline for processing large data sets, and load balancing. This paper describes a low-cost parallel processing solution to these tasks. In particular, we present a parallel polygon rendering algorithm for general-purpose MIMD(Multiple Instruction Multiple Data) message passing architectures. The hardware configuration consists of a host 80486 processor and several slave i860 processors. The current implementation consists of only two i860 processors but the design is scalable to more processors.

Section 2 describes the rendering process and includes a discussion of Gouraud shading and its use in a fast incremental implementation of texture mapping. A description of the algorithm and hardware configuration is given in Section 3, after a brief discussion of parallel rendering. Additional details concerning clipping and filtering are given in Section 4.

2. RENDERING

The process of generating images from abstract data models is known as *rendering*. Many different rendering techniques exist for visualizing a 3D scene. They vary from polygon rendering to ray tracing and radiosity techniques. The latter two methods offer more realism at the expense of system performance. Most commercial graphics workstations support polygon rendering only when special-purpose hardware is available. In this paper, we describe rendering, on off the shelf hardware.

Rendering Pipeline

There is a well-established pipeline for rendering 3D scenes (Foley , 1990). The elements of this pipeline include: modeling transformation, trivial accept/reject classification, illumination,viewing transformation, clipping, rasterization, and display. The first stage is responsible for traversing the 3D scene database and transforming the objects from the modeling coordinate system to the world coordinate system. This assembles all objects, each possibly modeled in different local coordinate systems, into one common world coordinate system. In order to avoid needless processing later in the pipeline, polygons that fall outside the view volume are culled. In the illumination step, contributions from each light source are evaluated for each polygon and color intensities are computed at each vertex. The viewing transformation step projects the 3D object coordinates into 2D screen coordinates. Culling is an optimization step that discards back-facing polygons that face away from the viewer. Clipping discards those parts of the projected polygons that lie outside the display screen. Rasterization converts transformed polygons into pixel color values.

It consists of three steps: scan conversion, visible-surface determination, and shading. Scan conversion determines which pixels lie in the projected polygons. Visible surface determination generally makes use of a z-buffer to store the depth at each pixel. The depth value at each pixel is used to determine whether the pixel's color information should be stored in the image (frame) buffer for subsequent display. If the new point lies closer than the pixel already stored in that position, the shading calculation is performed to determine the color. Three popular shading models include: flat, Gouraud, and Phong. Flat shading uses a *uniform* color to fill the polygon. Gouraud shading interpolates the color values stored at the vertices. Phong shading interpolates the vertex normals and recomputes the Phong illumination model at each pixel. Gouraud shading is most popular because it offers a good balance between realism and speed.

Gouraud Shading

ouraud shading is a popular intensity interpolation algorithm used to shade polygonal surfaces in computer graphics (Gouraud, 1971). It serves to enhance realism in rendered scenes that approximate curved surfaces with planar polygons. In addition to serving as a shading algorithm, we use a variant of this approach to interpolate texture coordinates. We begin with a review of Gouraud shading in this section, followed by a description of its use in texture mapping in the next section.

Gouraud shading interpolates the intensities all along a polygon, given only the true values at the vertices. It does so while operating in scanline order. This means that the output screen is rendered in a raster fashion, (e.g., scanning the polygon from top-to-bottom, with each scan moving left-to-right). This spatial coherence lends itself to a fast incremental method for computing the interior intensity values. The basic approach is illustrated in Fig. 1.

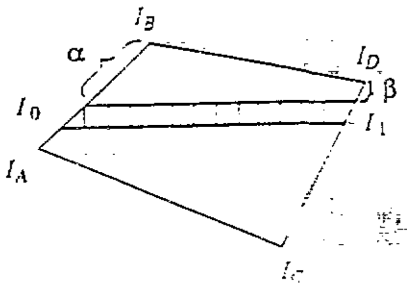

Figure 1: Incremental scanline interpolation.

For each scanline, the intensities at endpoints x_0 and and x_1 are computed. This is achieved through linear interpolation between the intensities of the appropriate polygon vertices. This yields I_0 and I_1 in Fig. 1, where

$$I_0 = \alpha I_A + (1-\alpha)I_B, \qquad 0 \leq \alpha \leq 1$$

$$I_1 = \beta I_C + (1-\beta)I_D, \qquad 0 \leq \beta \leq 1$$

Then, beginning with I_0, the intensity values along successive scanline positions are computed incrementally. In this manner, I_{x+1} can be determined directly from I_x, where the subscripts refer to positions along the scanline. We thus have

$$I_{x+1} = I_x + dI$$

where

$$dI = \frac{(I_1 - I_0)}{(x_1 - x_0)}.$$

Note that the scanline order allows us to exploit incremental computations. As a result, we are spared from having to evaluate two multiplications and two additions per pixel. Additional savings are possible by computing I_0 and I_1 incrementally as well. This requires a different set of constant increments to be added along the edges.

Incremental Texture Mapping

Although Gouraud shading has traditionally been used to interpolate intensity values, we now use it to interpolate texture coordinates. The computed *(u,v)* coordinates are used to index into the input texture. This permits us to obtain a color value that is then applied to the output pixel. The following segment of C code is offered as an example of how to process a single scanline.

```
dx = 1.0 / (x1 - x0);   /* normalization factor */
du = (u1 - u0) * dx;    /* constant increment for u */
dv = (v1 - v0) * dx;    /* constant increment for v */
dz = (z1 - z0) * dx;    /* constant increment for z */
```

```
for(x = x0; x < x1; x++) {
/* visit all scanline pixels */
        if(z < zbuf[x]) {
/* is new point closer? */
                zbuf[x]  = z;
/* update z-buffer */
                scr[x] = tex(u,v);
/* write texture value to screen */
        }
        u += du;                /* increment u */
        v += dv;                /* increment v */
        z += dz;                /* increment z */
}
```

The procedure given above assumes that the scanline begins at *(x0,y,z0)* and ends at *(x1,y,z1)*. These two endpoints correspond to points *(u0,v0)* and *(u1,v1)*, respectively, in the input texture. For every unit step in *x*, coordinates *u* and *v* are incremented by a constant amount, e.g., *du* and *dv*, respectively. This equates to an affine mapping between a horizontal scanline in screen space and an arbitrary line in texture space with slope *dv / du* (see Fig. 2).

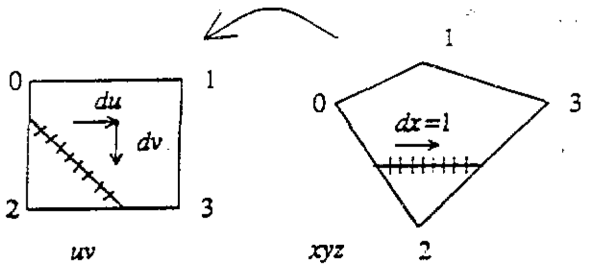

Figure 2: Incremental interpolation of texture coordinates.

Since the rendered surface may contain occluding polygons, the *z* -coordinates of visible pixels are stored in *zbuf*, the *z* -buffer for the current scanline. When a pixel is visited, its *z*-buffer entry is compared against the depth of the incoming pixel. If the incoming pixel is found to be closer, then we proceed with the computations involved in determining the output value and update the *z*-buffer with the depth of the closer point. Otherwise, the incoming point is occluded and no further action is taken on that pixel.

The function *tex(u,v)* in the above code samples the texture at point *(u,v)*. It returns an intensity value that is stored in *scr* , the screen buffer for the current scanline. For color images, RGB values would be returned by *tex* and written into three separate color channels. In the examples that follow, we let *tex* implement point sampling, e.g., no filtering. Although this introduces well-known artifacts, our goal here is to examine the geometrical properties of this simple approach. We will therefore tolerate artifacts, such as jagged edges, in the interest of simplicity. The filtering necessary for high-quality image generation is described in Section 4.

Figure 3 shows a checkerboard image mapped onto a quadrilateral using the approach described above.

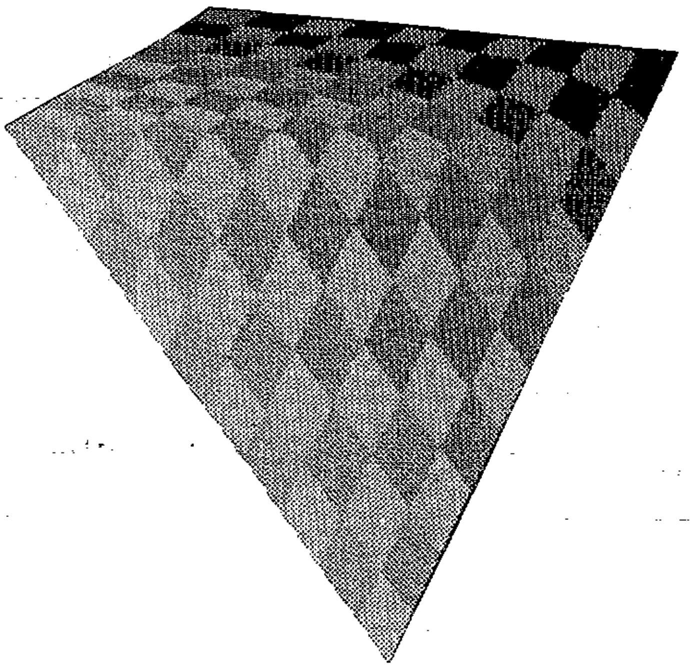

Figure 3: Naive approach applied to Checkerboard.

There are several problems that are readily noticeable. First, the textured polygon shows undesirable discontinuities along horizontal lines passing through the vertices. This is due to a sudden change in *du* and *dv* as we move past a vertex. It is an artifact of the linear interpolation of *u* and *v*. Second, the image does not exhibit the foreshortening that we would expect to see from perspective. This is due to the fact that this approach is consistent with bilinear transformation. As a result, it can be shown to be exact for affine mappings but it is inadequate to handle perspective mappings.

It is important to note that Gouraud shading has been used for years without major noticeable artifacts because shading is a slowly-varying function. However, applications such as texture mapping bring out the flaws of this approach more readily with the use of highly-varying texture patterns.

Incremental Perspective Transformations

A theoretically correct solution results by more closely examining the requirements of a perspective mapping. Since a perspective transformation is a ratio of two linear interpolants, it becomes possible to achieve theoretically correct results by introducing the divisor, i.e., homogeneous coordinate *w*. We *thus* interpolate *w* alongside *u* and *v*, and then perform two divisions per pixel. The following code contains the necessary adjustments to make the scanline approach work for perspective mappings.

```
dx = 1.0 / (x1 - x0);        /* normalization factor */
du = (u1 - u0) * dx;     /* constant increment for u */
dv = (v1 - v0) * dx;    /* constant increment for v */
dz = (z1 - z0) * dx;    /* constant increment for z */
dw = (w1 - w0) * dx;  /* constant increment for w */
for(x = x0; x < x1; x++) { /* visit all scanline
pixels */
        if(z < zbuf[x]) {   /* is new point closer? */
                zbuf[x] = z; /* update z-buffer */
                scr[x] = tex(u/w,v/w);     /* write
texture value to screen */
        }
        u += du;                       /* increment u */
        v += dv;                       /* increment v */
        z += dz;                       /* increment z */
        w += dw;                       /* increment w */
}
```

Figure 4 shows the result of this method after it was applied to the Checkerboard texture. Notice the proper foreshortening and the continuity near the vertices. See (Wolberg, 1990) for a more complete discussion on the topic of real-time texture mapping and fast texture coordinate evaluation using quadratic and cubic interpolation.

3. PARALLEL RENDERING

Coupled with texture mapping, polygon rendering provides realism and visual cues for flight simulation. The chief shortcoming of polygon rendering is that it may crudely approximate the (possibly) smooth surface shape. This problem, however, can be dealt by tesselating the surface into finer (smaller) polygons. This increases the number of polygons that must be rendered per frame, thereby increasing the computational load of the image generator.

One approach to parallelizing the rendering process is to map the stages of the rendering pipeline

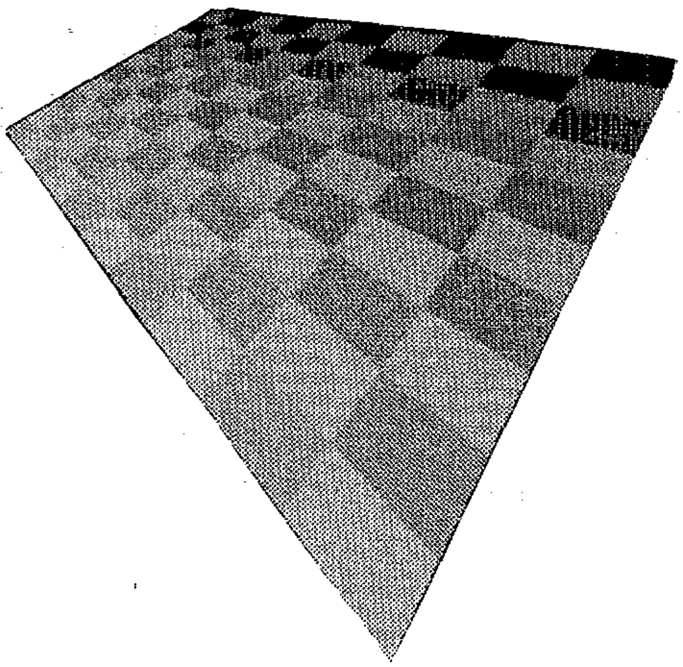
Figure 4: Perspective mapping using scanline algoribthm.

directly into hardware. In a pipeline architecture, though, the system can run only as fast as its slowest stage. Since it is generally difficult to evenly distribute the processing load over all processors, pipeline systems are not viable for rendering. Instead, we turn to a more general parallel algorithm whereby parallelism is obtained by replicating processing elements.

Close inspection of the rendering problem identifies that there are two main tasks to be parallelized: the transformation phase and the rasterization phase. Parallelism in the transformation and rasterization phases are referred to as *object parallelism* and *image parallelism*, respectively. Object parallelism is achieved by processing geometric primitives (triangles) independently of one another. Image parallelism is obtained by computing pixel values independently for individual pixels or groups of pixels.

Several groups of researchers have attacked this problem from both sides. A system with a high degree of object parallelism is described in (Torborg, 1987). The Pixel-Planes system, with a high degree of image parallelism, is described in (Fuchs, 1981). A system exploiting both object and image parallelism is described in (Molnar, 1992).

From an algorithmic standpoint, many approaches are possible. An excellent survey can be found in (Whitman, 1992). The most noteworthy systems are based on general-purpose MIMD architectures. We targeted this class of parallel architecutures because it conforms most closely with the growing trend of high-performance general-purpose processors with large instruction and data caches. In this manner, parallelism is obtained by replicating a single type of processing element. This approach is also consistent with parallelization achieved by mapping the problem onto workstations connected over a local area network.

There has been significant attention drawn to this approach in recent years. In [Barton 89], for instance, some of the issues involved in mapping the rendering pipeline onto message-passing systems are discussed. Other image and object parallelization results are presented in (Roble, 1988) and (Li, 1991). In recent work, load balancing and communication latencies in the message-passing environment are addressed in (Ellsworth, 1993). Finally, (Ortega , 1993) describes a data-parallel renderer suitable for both SIMD and MIMD architectures.

In the work described in this paper, we exploit both object and image parallelism to work on MIMD distributed-memory message-passing systems. Our rendering algorithm will run on systems containing up to p processors, where p is less than the number of scanlines in the frame buffer. Our method is unique in that it multiplexes the transformation and rasterization phases on the same processors. This has several advantages, including reduced memory utilization, overlapped computation and communication, and reduced communication contention. In addition, we ensure that all large data structures are distributed among the processors without wasteful duplication. In our case, this includes the list of polygons and the f rame buffer in which the final image is assembled. We distribute these structures evenly among the processors, allowing the algorithm to scale very complex scenes and high image resolutions by increasing the number of processors. Note that distributing the triangles corresponds to object parallelism, while distributing the image buffer corresponds to image parallelism.

Algorithm Description

The algorithm works as follows. Each of p processors is given a list of polygons to render. For optimization purposes, all polygons are restricted to be triangles. This does not pose a restriction because the input scene description of the terrain consists of elevation data on a regularly spaced mesh. Each 2×2 set of nodes on the mesh is treated as two contiguous triangles. The image buffer is divided among the p processors in equal-sized horizontal strips (see Fig. 5). Each stripe contains the same number of scanlines, which results in uniform, predictable memory requirements on each processor. The optimal size of the strips is view- and scene-dependent and remains an area of research. The following strategy is followed by each processor:

1) The lighting, transformation, and clipping steps are performed by each processor on its list of triangles. This results in 2D triangles mapped into screen coordinates on the projection plane.

2) The 2D triangles are split, if necessary, into trapezoids along the local image buffer boundaries (see Fig. 5). Each trapezoid is then sent to the processor responsible for that corresponding image buffer segment.

3) Upon receiving a trapezoid, a processor rasterizes it into its local image buffer using a standard z-buffer algorithm to eliminate hidden surfaces. The real-time texture mapping algorithm described earlier is used to shade the projected triangles.

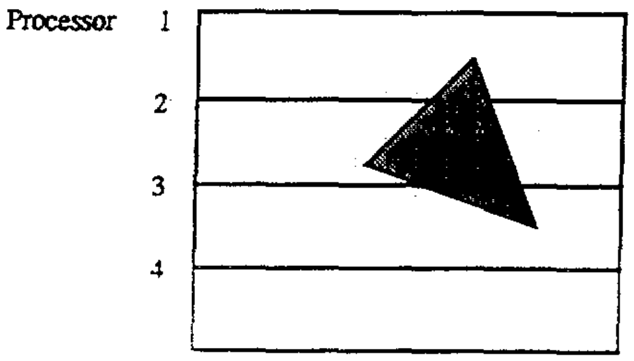

Figure 5: The image buffer is distributed across processors.

Triangles are split at boundaries. Processors start with nearly the same number of triangles, but several factors will tend to unbalance the load. This may be due to the different number of operations required to cull, clip, and subdivide the triangles. Similarly, varying the number and sizes of incoming

trapezoids will cause potentially large variations in the rasterization time.
As a result, it is best to avoid any synchronization points in the process to reduce significant amounts of idle time. Indeed, there is no synchronization point in this approach. Furthermore, to reduce communication overhead, trapezoids destined for the same processor are buffered into larger messages before sending.

Division of Labor

The host 486 processor is responsible for several tasks:

1) Monitor the joystick and update the display information on the host screen. This information includes altitude/attitude, speed, and frames per second.

2) Update the position variables for the eye in accordance with the flight dynamics.

3) Compute the M_{wv} matrix that converts all points from the world coordinate system to the viewing coordinate system. The matrix is computed asynchronously, i.e., only when the slave processors signal that they are ready to begin processing a new frame. This achieves a better sensation of real-time navigation. It is important to note that the perceived speed is equal to the product of frames/second and meters/frame. Since frames/second will likely vary, perceived speed can be made more uniform only if the matrix is not associated with equal intervals in time.

4) Distribute M_{wv} and lists of triangles to the processors.

The slave processors currently consist of two i860s. The program will be migrated to the new Analog Devices 21060 processors that are faster and support larger cache sizes, permitting larger image buffer segments to reside locally.

4. INTERPOLATION AND ANTIALIASING FILTERING

Flying over large terrain raises several interesting problems with respect to clipping and filtering. It becomes unfeasible to visit all of the triangles that comprise the terrain. Even straightforward accept/reject clipping tests becomes an overwhelming task when several million triangles must be considered at real-time rates. Instead, we propose the following solution that exploits the mesh structure of our terrain elevation data.

We project a ray from from the eye through the four corners of the view plane window. These four rays pierce the image base plane. All terrain elevation data contained in the resulting projected area are considered for viewing. This proves to be far less than the entire data set. Special care is taken when looking towards the horizon and some of the four rays may not intersect the image base plane.

Due to perspective foreshortening, we can expect many triangles to project to subpixel regions. This is particularly true for distant triangles. Such many-to-mappings give rise to undersampling, and therefore aliasing. Aliasing is a condition in which artifacts appear in the image due to undersampling the signal. A solution to aliasing is to bandlimit (blur) the image before sampling. In order to avoid having to perform blurring in real time, we preprocess the input image (texture), constructing a multi-resolution image pyramid. Those points of the texture that map into small regions are sampled from the heavily blurred pyramid level. In this manner, preprocessing the image into successively blurred levels of a pyramid permit us to avoid averaging filtering during run-time. A 17-point Hann windowed sinc function is used to build the pyramid (Wolberg, 1990). The major trade-off here is that the area of integration is approximated to be a square.

Image pyramids are of use when there is a many-to-one mapping, i.e., minification. Those points of the texture that map into large regions are magnified and must therefore be interpolated from the highest resolution pyramid level (the original). Bilinear and cubic convolution are two interpolation techniques currently supported by the system. The former (latter) method makes use of the 2×2 (4×4) set of nearest neighbors to compute the value of the pixel at a fractional position.

5. SUMMARY

We describe in this paper a PC-based photographic-quality image generator for flight simulation. In order to effectively texture map full color imagery onto high resolution terrain data at near real-time rates, a multiprocessor system has been designed to achieve both object and image parallelism. The main thrust of the algorithm is to divide the frame buffer into p horizontal strips, each associated with one of *p* processors. A processor rasterizes incoming triangles (or trapezoids) into its local buffer only if that triangle lies in that segment of the image. If the incoming primitive either straddles the local segment or lies entirely outside of it, the processor sends the (clipped) primitive to the appropriate processor. The original list of triangles is distributed to the *p* processors by a host 486 processor. Subsequently, though, the *p* processors pass primitives among themselves, if necessary. Each processor balances its work between clipping primitives and rasterizing them. When no more rasterization needs to be done to complete an image, the local image buffers are collected into one output frame buffer.

6. REFERENCES

Barton, E. (1989). Data Concurrency on the Meiko Computing Surface. *Parallel Processing for omputer Vision and Display.* P.M. Dew, R.A.Earnshaw and T.R. Heywood, eds., Addison-Wesley, pp. 402-407.

Ellsworth, D. (1993). A Multicomputer Polygon Rendering Algorithm for Interactive Applications. *Proceedings of the 1993 Parallel Rendering Symposium,* ACM Press, pp. 43-48.

Foley, J.D., Van Dam, A., Feiner S.K., Hughes, J.F. (1990). *Computer Graphics: Principles and Practice,* 2nd Ed., Addison-Wesley, Reading, MA.

Fuchs, H. and Poulton, J. (1981). Pixel-Planes: A VLSI-oriented design for a Raster Graphics Engine. *VLSI Design* , Vol.2, No. 3,pp. 20-28.

Gouraud, H. (1971). Continuous Shading of Curved Surfaces. *IEEE Trans. On Computers*, Vol. 20,No.6, pp., 623-628.

Li, J. and Miguet, S. (1991). Z-buffer on a Transputer-Based Machine. *Proceedings of the Sixth Distributed Memory Computing Conference.* IEEE Computer Society Press, April , pp. 315-322.

Molnar, S., Eyles, J. and Poulton, J. (1992). PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics*, Vol. 26, No. 2, July, pp. 231-240.

Ortega, F., Hansen,C., and Ahrens, J. (1993). Fast Data Parallel Polygon Rendering. *Proceedings Supercomputing'93.* IEEE Computer Society Press, November, pp. 709-718.

Roble, D. (1988). A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube. *Proceedings of Pixim'88, Paris, October* , pp. 177-192.

Torborg, J.G. (1987). A Parallel Processor Architecture for Graphics Arithmetic Operations. *Computer Graphics.* Vol. 21, No. 4, pp. 197-204.

Whitman, S. (1992). *Multiprocessor Methods for Computer Graphics Rendering.* Jones and Bartlett, Boston.

Wolberg, G. (1990). *Digital Image Warping.* IEEE Computer Society Press.