Parallel Image Generation with Anti-Aliasing and Texturing

TR87-005

1987

Gregory D. Abram

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

Parallel Image Generation With Anti-Aliasing and Texturing

by

Gregory D. Abram

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill, 198[illegible]

Approved by:

Adviser: Professor Henry Fuchs

J. Turner Whitted

Reader: Professor J. Turner Whitted

[illegible]

Reader: Professor John Poulton

Abstract

This dissertation explores the use of an analytic visibility algorithm for the high-speed generation of anti-aliased images with textures. When visibility is known analytically before any sampling takes place, low-pass filtering for anti-aliasing can be done to arbitrary accuracy at a cost proportionate to the output resolution. Furthermore, since the filtering and sampling processes can be expressed as a set of integrations over the image plane, the filtering process can be decomposed into a set of sums of integrations over each visible surface in the image plane, allowing the rendering of each visible surface to be done in parallel using an image buffer to accumulate the results. Thus, analytic visibility can serve as the basis for high-speed, high-quality image synthesis.

In this dissertation, algorithms for computing analytic visibility and for producing filtered renderings from the resulting visible surfaces are presented. In order to provide real-time performance, these algorithms have been designed for parallel execution on simple concurrent structures. Architectures based on these algorithms are presented and simulated to produce expected execution times on a set of test images.

Table of Contents

Introduction: Computer-Generated Imagery

As the power of computers to process data has risen over the past few decades, the use of visual media to convey information between computers and their users has risen apace. Today, the use of computer graphics to aid humans in such diverse fields as engineering design, molecular chemistry and pilot training is widespread and growing. Even so, achieving the full utility of computer-generated imagery remains an elusive goal; for many potential uses of computer graphics, the current state-of-the-art is inadequate for reasons of cost, speed or image quality. Typically, some concessions must be made: either a simplified representation of data may be used, such as a line-drawing of an engineering part, or image generation speed may be sacrificed, leaving a car body designer waiting minutes or hours for a realistic rendering of his work. The cost of a system featuring both speed and quality is currently prohibitive.

In this dissertation I will consider a specific branch of computer image generation: the fast and realistic rendering of objects and scenes such as those we see around us in the real world. Figures 1 through 6 are examples of these types of images: three scenes from a model of the new UNC Computer Science building, two images of aircraft flying over terrain and a scene containing a complex mechanical assembly (a ship). These six images will serve as test images for evaluating the algorithms and architectures reported in this dissertation.

Three problems make the real-time generation of such images a difficult task. First, real world scenery typically contains a great deal of complexity; an image generation system designed to produce such scenery must provide techniques for modeling and rendering images containing high degrees of detail. Further, digitally rendered images are prone to artifacts of computer generation; these effects must be minimized in a high-quality rendering system. Finally, the production of such scenes requires very large amounts of computation; the real-time generation of such images generally requires the use of specially-designed hardware that provide the necessary performance by distributing the computation over concurrently operating components.

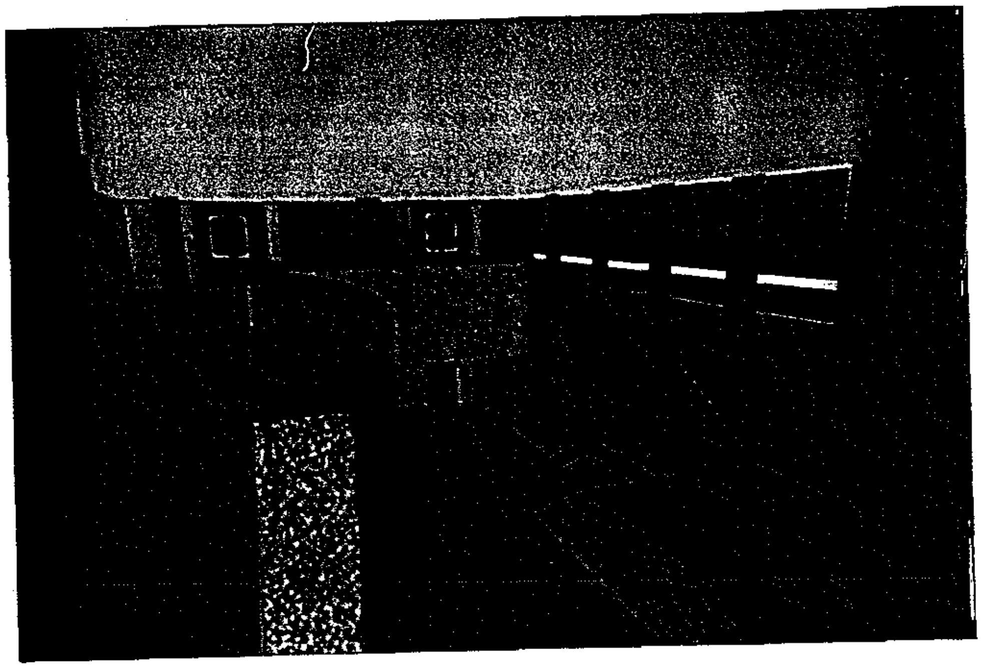


Figure 1. Building Scene 1

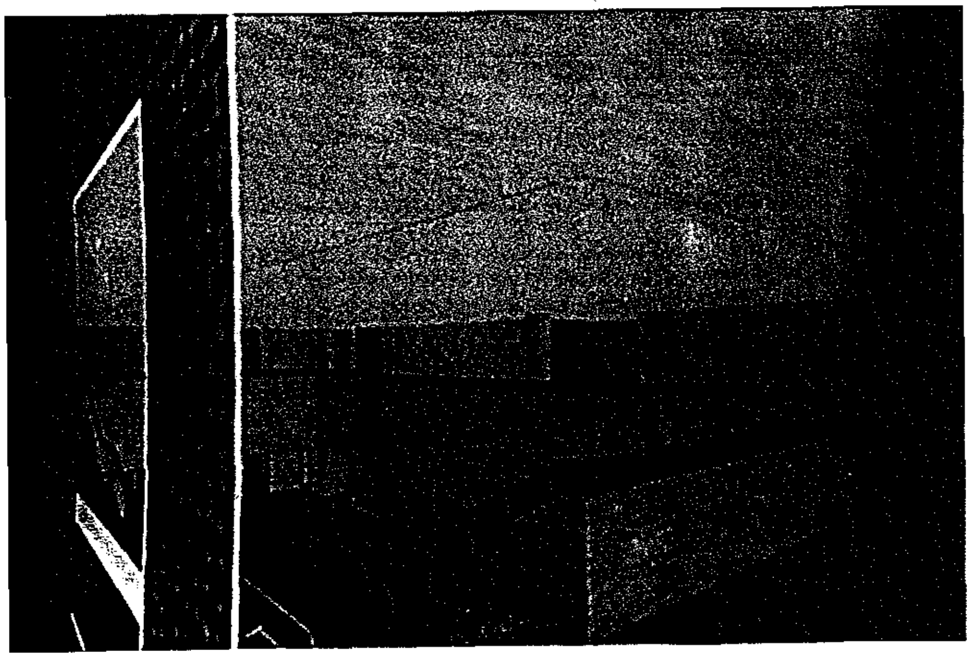


Figure 2. Building Scene 2

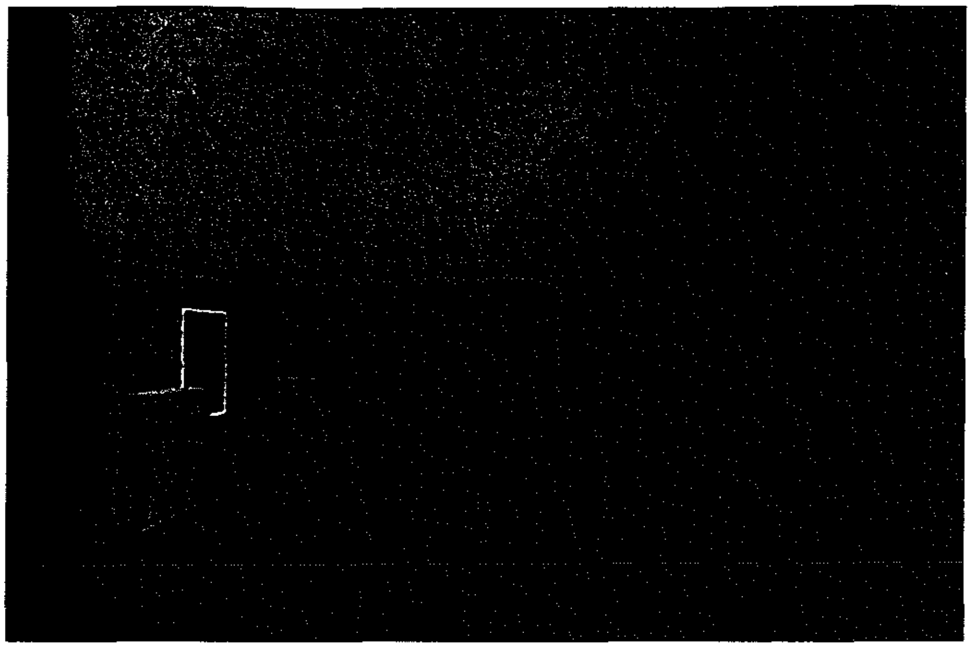


Figure 3. Building Scene 3

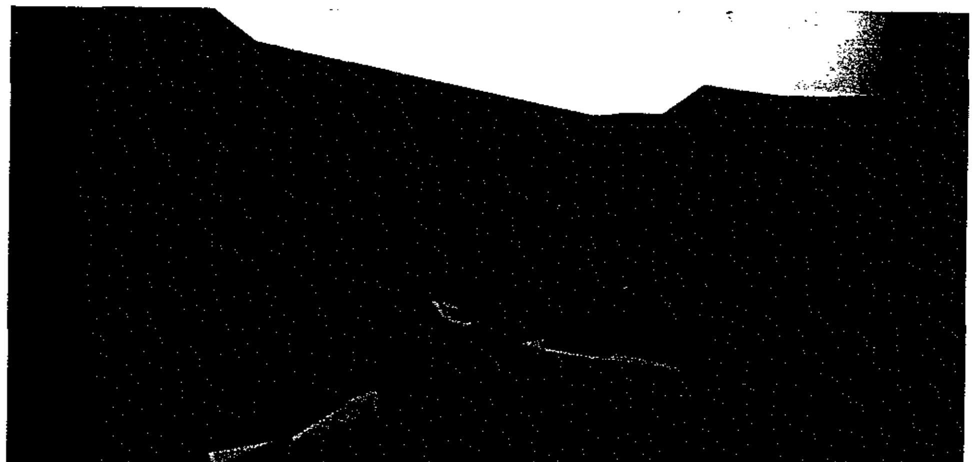


Figure 4. Aircraft Scene 1

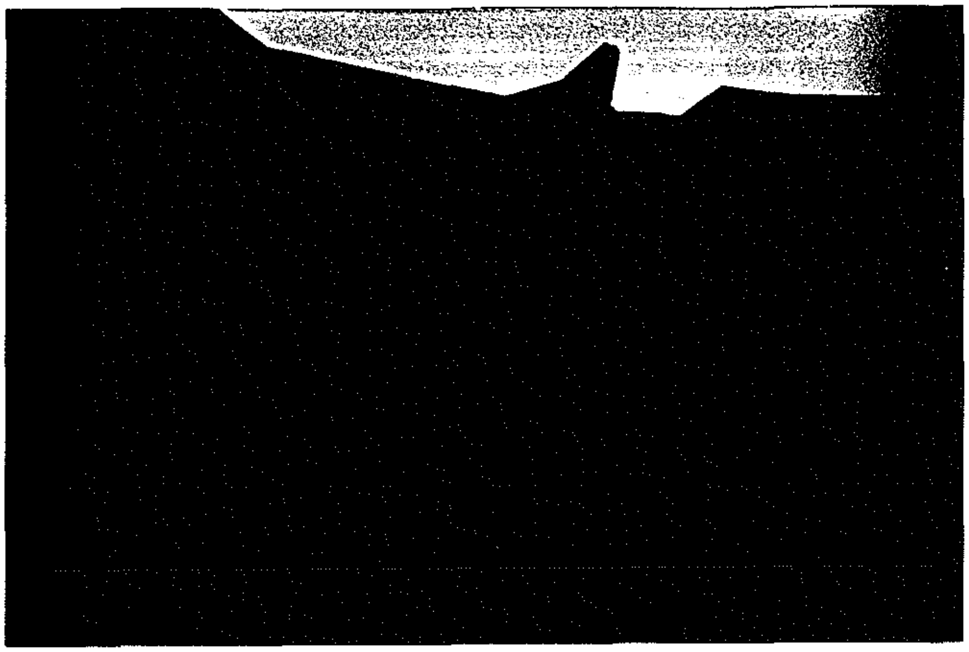


Figure 5. Aircraft Scene 2

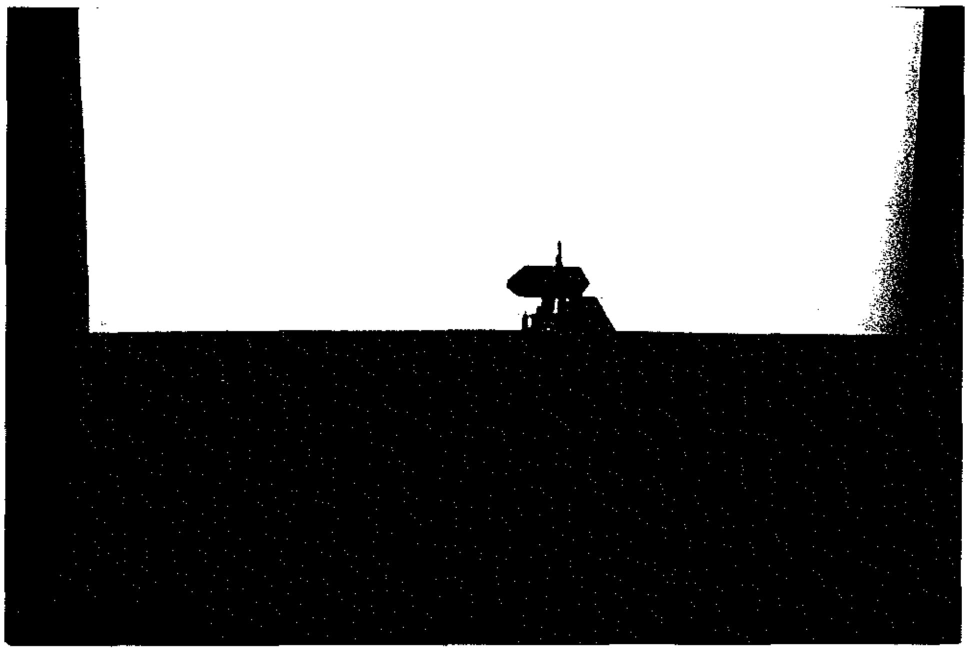


Figure 6. Ship Scene

This dissertation describes a method of rendering which provides a high degree of image quality and realism while decomposing the rendering computation in a manner amenable to concurrent execution on a simple, highly parallel image generator. This approach is based on the use of an *analytic visibility algorithm*, which determines a set of analytically-defined visible surfaces in object-space directly from the set of input surfaces without resorting to a sampling step. This achieves two important results: 1) the different visible surfaces are independent of one another and can be distributed to separate processors for rendering, and 2) the filtering necessary to remove aliasing artifacts can operate on the analytic definitions of the surface, allowing anti-aliasing to be done to arbitrary accuracy independent of the sampling resolution.

The next section of this chapter will outline the steps which all image generation systems must perform in order to produce realistic images of three-dimensional scenes. The following section will discuss the cause of the artifacts visible in many computer-generated images, the aliasing resulting from the discrete nature of most image generation algorithms. The final section of the chapter outlines the solution proposed in this dissertation.

1.1. Image Generation Overview

Because of the high computational expense of image synthesis, most graphics systems tend to sacrifice generality for efficiency in a narrow family of applications. Systems designed for flight simulation can render scenes containing thousands of polygons in real-time, but cannot use the expensive high-quality lighting models that are needed for highly realistic appearance [Sch,81]. Very high-quality image generation systems can produce images of nearly photographic realism, but require many hours to do so [ICG,86, Whi,80]. Some applications (such as molecular graphics) can forego solid-appearing shaded surfaces in favor of line drawings; in such cases, calligraphic displays can provide real-time interaction at relatively low cost.

Even in the face of the diversity of computer image generation systems, we find that most or all image generators must go through a set of generic steps in order to produce views of three-dimensional scenes [NeS,9]. Conceptually, the image generation process can be divided into three distinct components: the representation of scene data to the image generation system, the specification of the viewing parameters defining a specific view of the scene, and the set of procedures that produce an image of the scene data given the set of viewing parameters. In this section these generic steps of image synthesis are detailed.

1.1.1. Modelling and Representation of Scene Data

In human terms, perhaps the most difficult aspect of computer graphics is the *modelling* process, in which the set of objects to be rendered is defined. This process intertwines the needs of the human user, in providing him with the shapes required for the objects of interest and with tools with which he can conveniently and effectively manipulate the objects, with the needs of the image generator: a precise definition of the objects in a form which can be efficiently transformed into actual images.

The representation of scene data is thus a critical consideration in the design of a graphics system. Certainly the most common surface representation (and the one used in this work) uses planar polygons, defined by ordered lists of vertices, to define surfaces. Polygons can be used to model any real surface to any given tolerance; furthermore, "smooth-shading" techiques can be used to give polygonal approximations the appearance of being smoothly curved surfaces [Gou,71, Pho,75].

Other surface representations are also common. Bicubic surfaces (such as Bezier, B-spline and Beta-spline surfaces), provide a class of shapes that can be easily and intuitively manipulated and which are useful in the definition of sculptured objects such as ship hulls, aircraft fuselages and automobile bodies [Bar,81, Coo,67, For,72, Rie,75]. More complex structures may be conveniently defined using Constructive Solid Geometry (CSG) techniques, in which shapes are formed as boolean combinations of simple geometric primitives such as blocks, cones, spheres and torii [ReH,77b]. Current work by Kedem [KeE,84] is investigating a specialized machine architecture for processing CSG-defined objects, producing not only image data but also determining geometrical information such as center of gravity and volume.

1.1.2. Coordinate Transformations and Perspective Projection

Most modelling techniques produce geometrical data (vertex coordinates, control points, centerpoints etc.) that are defined in an arbitrary "world" coordinate system (WCS) independently of any specific view of the model. To specify a scene of the model, a viewpoint and viewing direction must be chosen. Given this information, the model must be geometrically transformed from the world coordinate system to an "eye" coordinate system (ECS), such as seen in figure 7. This coordinate transform is efficiently performed by multiplying world-coordinate information by a 4x4 homogeneous transformation matrix which defines the coordinate transformation from the WCS to the ECS and which is determined by the specified position and direction of view.

Once the model has been transformed to the ECS it must be projected onto the image plane. We can imagine the image plane to be a window through which the observer, sitting at the eyepoint, sees the image; each point on an object surface appears on the image plane where the ray extending from the viewpoint to the point on the object surface intersects the image plane (as seen in figure 8). This *perspective projection,* involving the division of vertex coordinates by the depth of the vertex, provides the foreshortening effects that one expects in realistic scenes[1].

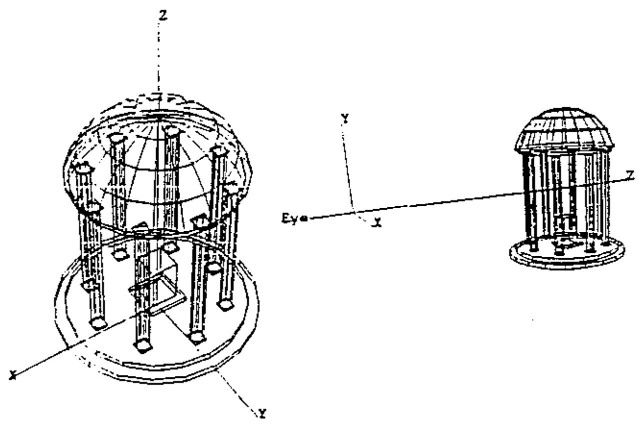


Figure 7:. World coordinate system, eye coordinate system

[1] Note that ray-tracing algorithms produce this effect by actively casting rays from the viewpoint through each sample point in the image plane and out into the scene.

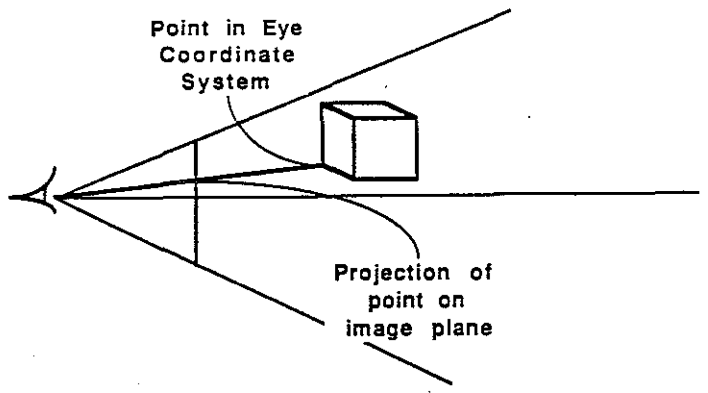

Point in Eye Coordinate System
Projection of point on image plane

Figure 8: Perspective projection

1.1.3. Clipping

In most image generation algorithms, only objects lying inside the frustum defined by the viewpoint and the edges of the image plane and capped by arbitrary hither and yon planes perpendicular to the axis of view can be seen. In order to minimize the work necessary in subsequent image-generation steps, the model is clipped against the walls of this frustum, eliminating portions of the model lying outside the frustum [SuH,74]. Clipping is not done, however, when techniques are used that allow objects outside the frustum to affect the image; this can happen when objects lying inside the viewing frustum are reflective: objects lying outside the frustum might be visible as reflections on the surfaces of objects lying inside the frustum.

1.1.4. Visible surface calculation

From any given point of view, only certain surfaces, and portions of others, will be visible; the remainder are occluded by nearer, intervening surfaces. A correct rendering of a three-dimensional scene requires that the visible surfaces be determined. Visible surface algorithms are probably the most thoroughly investigated area of computer graphics research.

Certain properties of images can be used to speed this calculation. Sutherland, Sproull and Schumacker [SSS,74] define *coherence* as "the extent to which the environment or the picture is locally constant". In general, three forms of coherence can be exploited to simplify the visible surface calculation. Since scene data consists of surfaces, visibility is coherent over the contiguous areas of the screen that correspond to the visible surfaces; if these areas can be identified, no further visibility calculation is necessary within the area. Further, the factors that determine visibility tend to be coherent; thus, by carefully structuring the data, these factors can be maintained incrementally, making the recalculation of visibility at transitions simple. Finally, many sequences of images are temporally coherent; successive frames will change only slightly. Virtually all rendering algorithms exploit some aspects of coherence to speed the rendering process; the taxonomy of visible surface algorithms given in [SSS,74] is based on the manner in which they exploit coherence.

In this research we consider the use of *analytic* visibility algorithms in the real-time rendering of high-quality images. These algorithms, falling within the family of object-space algorithms in the Sutherland, Sproull and Schumacker taxonomy but developed since the time of that paper, differ from other algorithms in that they operate directly on a set of surfaces which *might* be visible in the scene to produce a similarly defined set of surfaces which *are* visible, with all occluded surfaces and all occluded portions of partially-visible surfaces removed. As we will see below, analytic techniques have significant advantages for generating images with minimal artifacts of computer generation.

Chapter two of this dissertation will cover the current state of the art in visible-surface algorithms. This chapter, based around the Sutherland, Sproull and Schumacker taxonomy [SSS,74], is intended to update the state-of-the-art as reported in that paper.

1.1.5. Lighting and texturing

The appearance of an object depends both on its surface characteristics (such as its color, shinyness and texture) and on the the relationship between the surface orientation and the direction to the light sources and viewer. These parameters are combined by a lighting model to produce a shade. Gouraud generated the first smoothly shaded images of polygonally-defined surfaces by combining the normal vectors of each polygon that shares a vertex to approximate a surface-normal vector for the vector, computing an actual color for the vertex based on this surface normal, and then interpolating the colors assigned to the vertices of each surface polygon across the face of the polygon. Later work by Bui Tuong Phong extended smooth-shading to include highlights by interpolating the vertex normals, rather than the colors, and recomputing the lighting model at each sample location. Further work by Whitted, Blinn, Cook and Torrance and many others have produced improved lighting models to provide a wide range of very realistic surface appearances.

A powerful technique to give simple geometrical models the appearance of highly complex, real-world surfaces is to apply a texture, representing the colors of a complex surface, to a simple surface, representing the position and orientation of the surface in the model [BlN,76, Bli,78, Cat,74]. For example, a brick wall can be represented either by many little red polygons separated by many little white polygons, or by a single picture of a brick wall accompanied by a polygon representing the position of the wall in the scene. Using this method, many of the complex calculations necessary to render the wall (such as geometrical transformation, clipping and determining visibility) can be applied to the simple polygon, rather than to each of the many brick polygons. Appying this technique requires that two surface parameters (generally referred to as U and V) be interpolated across the simple surface at rendering time and, each time the lighting model is applied, used to sample a texture to determine a unique color for that spot on the polygon.

1.1.6. Real-time Image Generation

Real-time image update requires the generation of anywhere from five to sixty frames each second (depending on the specific algorithm). If output images are to be generated at a resolution of 512 lines by 512 columns and at 30 frames per second, over 7.5 million individual pixel shades must be calculated each second! Since each pixel calculation requires a reasonably large amount of computation, real-time generation of realistic imagery requires the use of highly parallel special-purpose hardware to provide the necessary performance.

Current real-time systems have been built using high-speed small- and medium-scale integrated circuits. Since this technology provides only a few logic gates per chip, the design of these systems has required that the the actual number of gates in the system be minimized to keep the parts count down. These systems therefore have consisted of random logic implementing the functions necessary for image synthesis in the fewest possible gates. Even so, they require a great deal of hardware making real-time shaded image synthesis a very expensive tool practical in only a very few applications, primerally flight simulation.

In recent years, however, the advent of very large-scale integrated (VLSI) circuits has radically changed the design considerations of high-performance systems. VLSI technology makes it possible to place many thousands of individual processing elements on a single integrated circuit, albeit at a high design cost. This radically changes the strategies underlying system design: rather than re-implement the random logic of the earlier generation in VLSI, new approaches to image synthesis must be developed which can be decomposed onto a set of identical components operating concurrently to provide the necessary performance.

Real-time image synthesis remains a difficult and challenging problem. Current commercially available systems, capable of remarkable performance though at a high price, establish the usefulness of these systems, and current research projects indicates the potential for further dramatic improvements in cost and performance. Chapter three of this thesis will describe the current state-of-the-art in detail.

1.2. Sampling and Aliasing

The discrete, digital nature of computer image generation techniques is often apparent in the resulting images. In his 1976 dissertation [Cro,76], Crow observed that many of the artifacts visible in computer-generated imagery stem from an inadequate sampling rate used in their generation. Perhaps the most familiar aliasing effect is the staircasing (called the "jaggies") visible along the edges of surfaces in very simple computer-generated images. Aliasing may also cause small features to drop out of images and moire effects in highly detailed surfaces.

The most common remedy for aliasing effects is to compute the image at high resolution. Figure 9 shows a simple polygon rendered at 16x16, 32x32 and 64x64 resolution; clearly, as the resolution rises the jaggies diminish. Unfortunately, though, they never will disappear altogether, and since the cost of image generation rises as the square of the resolution, this is a computationally expensive approach. Furthermore, there is a finite (and relatively low) resolution at which images can be conveniently displayed (low-cost systems are generally limited to about 512x512 resolution while more costly systems allow 1024x1024 resolution). Thus simply raising the resolution of the generated images is not an adequate solution to the aliasing problem.

Crow went on to apply techniques of signal processing to understand these artifacts and to offer solutions. The remainder of this section will review his results.

1.2.1. Image functions and raster representations

Any image can be considered to be a two dimensional function *I(x,y)=color* producing the color visible at any point (x,y) in the image plane. This function is defined by the physical properties of the objects in the scene, the light falling on the scene, and the specified viewing parameters. Computer-generated images are approximations of these underlying, idealized images. Artifacts of computer generation correspond to situations in which the image generation techniques used fail to adequately approximate the underlying image.

Many of the most common artifacts in computer-generated imagery stem from the fact that digital scene generators cannot represent the image function directly; instead, rectangular arrays

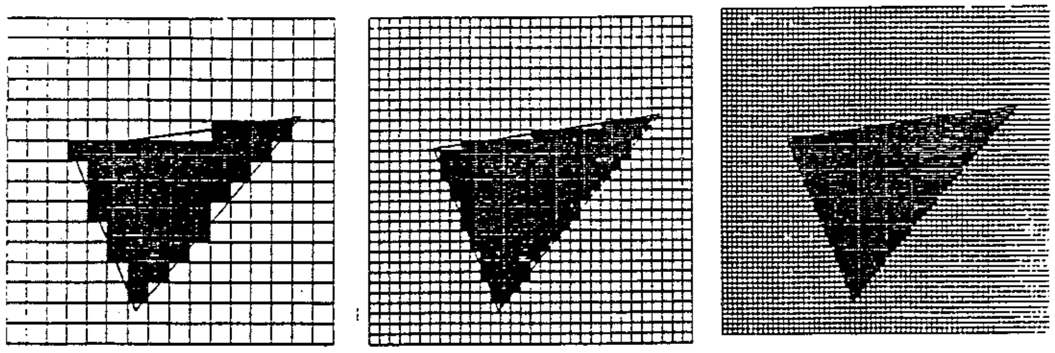


Figure 9: Raster renderings of a polygon

(rasters) of discrete color samples (pixels) are used to store *tabulations* of the image function. As a part of the raster scan-out process, samples are read from the raster and interpolated in an attempt to reconstitute the original image function.

Many image generation algorithms are closely tied to this raster representation of the image function. The simplest algorithms determine a color for each pixel by finding the color visible behind the exact center of the pixel; the image function is said to be *point-sampled* at the center of each pixel. Such algorithms are limited by the resolution of the image raster and are therefore extremely prone to aliasing effects.

1.2.2. Sampling and aliasing

The point-sampling technique produces a result equivalent to the multiplication of the image function $I(x,y)$ by an infinite impulse array that produces samples separated by τ in either direction:

$$III_\tau(x,y) = \begin{cases} 1, x \text{ and } y \text{ are multiples of } \tau \\ 0, \text{otherwise} \end{cases}$$

Figure 10 illustrates this sampling multiplication in the one-dimensional case. Ideally, the resulting tabulation can be used to reconstruct the original image function by interpolating the samples.

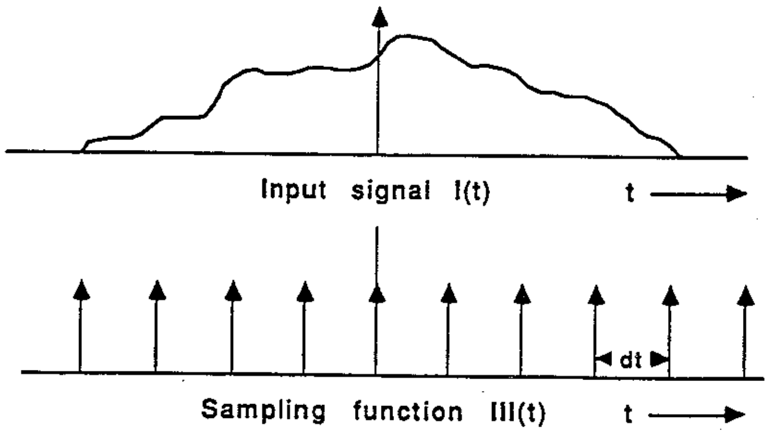


Figure 10: Sampling in one dimension

In the frequency domain, the multiplication of the image function with the sampling function corresponds to the convolution of the Fourier transform of the image function $I'(u,v)$ with the Fourier transformation of the sampling function $III_\tau'(u,v)$, an infinite impulse array similar to III, but now with spacing $1/\tau$, the *sampling frequency* (figure 11). In effect, this convolution produces the sum of an infinite number of copies of the Fourier transform of the original image function, with each copy centered at a sample point of III_τ'. If the replicated copies of I' do not overlap (that is, the original image function contains no frequencies above $1/2\tau$), the original image function can be recovered by removing all frequencies above $1/2\tau$, thereby isolating the copy centered at the origin and leaving the correct Fourier transform of the original function. Ideally, this is done by multiplying by a pulse function of radius $1/2\tau$ (the so-called *Nyquist frequency*), corresponding to the use of the sinc function ($\frac{sin(x)}{x}$) to interpolate the samples in the spatial domain.

Suppose now that the original image function contains frequencies above $1/2\tau$. If so, the replicated Fourier transforms of the image functions overlap; the high frequency components of neighboring replications corrupts the low frequencies of the central copy; they are said to "alias" as low frequency components. Now multiplication by the pulse function cannot correctly isolate a single copy of the Fourier transform of the image function and, in the spatial domain, interpolation

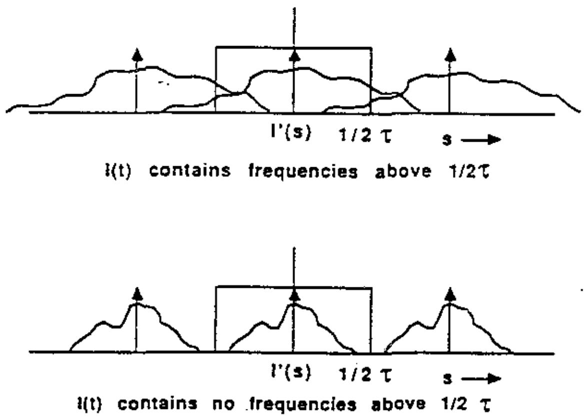

I'(s)
1/2 τ
s
I(t) contains frequencies above 1/2τ
I'(s)
1/2 τ
s
I(t) contains no frequencies above 1/2 τ

Figure 11: Sampling in Fourier space

of the samples produces an incorrect result.

1.2.3. Filtering and alias prevention

Unfortunately, images generally contain edges and thus image functions generally contain very high frequencies. This leaves us with two choices: either we increase the sampling frequency, thereby increasing the Nyquist frequency and reducing the aliasing energy to a tolerable level, or, given a predetermined sampling rate, we must remove unrepresentable frequencies before sampling to leave the best image possible at that resolution.

Aliasing frequencies can be removed from an image function by the application of a low-pass filter, in effect blurring the image to remove sharp edges. This is performed by convolving the image function with a low-pass filter kernel:

$$I_{filter}(x,y) = \iint_{-\infty}^{\infty} I_{real}(u,v)\ F_{low}(x-u,y-v)\ dudv$$

The result is then sampled to produce a tabulation of the filtered image function:

$$I_{sample}(x,y) = III(x,y) I_{filter}(x,y) = III(x,y) \iint_{\infty}^{\infty} I_{real}(u,v)\ F_{low}(x-u,y-v)\ dudv$$

Since the sampling operation drives the result to zero at all but the sample points, there is no need to compute the filtered image function between the sample points. Thus the tabulated, filtered image function can be computed by evaluating the integral of the underlying image function with a low-pass filter kernel centered at each sample point.

1.3. A Proposed Solution: Statement and Thesis Summary

High quality, alias-free rendering of images with textures can be rapidly computed by the use of an analytic visible-surface algorithm that divides the process into many independent parts amenable to parallel execution. The goal of this thesis is to devise a new approach to real-time high quality image synthesis that replaces the large quantities of special-purpose custom circuitry currently required with a simply connected parallel system consisting of many components of only a few different types.

The use of an analytic visible surface algorithm supports both high-quality image generation and simple, highly parallel rendering. If visibility is computed analytically before any sampling takes place, the integrals required for low-pass filtering can be decomposed into the sum of the integrals over each visible surface [AWW,85]:

$$I_{filter}(i,j) = \iint_{\infty}^{\infty} I_{real}(u,v)\; F_{low}(x-u,y-v)\; dudv = \sum_{\text{visible surfaces}} \iint_{\infty}^{\infty} I_{real}(u,v)\; F_{low}(x-u,y-v)\; dudv$$

Thus the contribution of each visible surface to the final image can be computed independently of any other and simply accumulated to form the final image. Further, since visibility is known to the precision of the computer rather than the resolution of the display device, these integrals can be computed numerically to any required accuracy.

1.3.1. Algorithms

If such a system is to operate in real-time, both the analytic visibility computation and the rendering of the visible surfaces must be performed in real-time. Central to this thesis are two algorithms: first, a highly parallel analytic visible surface algorithm which distributes the visibility calculation itself among many processors operating in parallel, and second, a fragment rendering algorithm which efficiently computes the contribution of a visible surface to the final output image.

A Parallel Analytic Visible Surface Algorithm

The analytic visible surface algorithm, similar to that of Sutherland, is based on a recursive decomposition of the viewing frustum into sub-volumes until a single surface is visible in each. Any plane defined by the viewpoint and an edge of any surface in the world model may be used to divide the set of surfaces into two, one on each side of the plane. In this manner, the original viewing frustum is recursively divided into smaller and smaller irregular (though convex) volumes until a single surface is found that hides all of the remaining surfaces in each volume. Since the dividing planes pass through the viewpoint, no surface on one side of the plane may hide any surface on the other side of the plane; this process may therefore be applied independently (and in parallel) to each of the two subsets.

A Fast Filtering Rendering Algorithm

As noted above, the final image can be decomposed into the contribution of each visible surface within the scene. Computing the correct contribution of a visible surface, however, requires that a low-pass filter be applied to the surface. This involves the integration of the product of a filter kernel centered at each pixel affected by the surface with the surface itself. This integration may be computed to any degree of accuracy by determining the intersection of the surface with each sample square (eg. pixel region) that it overlaps and using a simple table lookup process to determine the contribution of the sample-square intersection with the neighboring pixels.

1.3.2. Architectures

This thesis will explore the use of these algorithms by developing and evaluating a system architecture based on them. This will consist of three parts: a front end, responsible for database manipulations and initial data preparation; a geometry processor, which performs the analytic visible surface algorithm, and finally a rendering processor, which renders the final image from the set of visible surfaces. The latter two components will consist of many individual processors which share the computational burden performed by the component.

Visibility Algorithms

One of the most extensively studied problem in computer graphics research is that of determining visibility among the surfaces of a tree-dimensional scene. "A Characterization of Ten Hidden-Surface Algorithms", by Sutherland, Sproull and Schumacker [SSS,74], offers the most comprehensive review of visible surface algorithms. However, considerable developments have taken place in the field since the publication of that paper. The considerations that affected the choice of visible-surface algorithms have changed dramatically. This chapter will review the current state-of-the-art in visible-surface algorithms, organized roughly around the taxonomy offered in the Sutherland, Sproull and Schumacker paper but with particular regard to more recent developments.

2.1. Visibility and Coherence

In the Sutherland, Sproull and Schumacker paper, coherence is defined as "the extent to which the environment or the picture is locally constant". In general, three aspects of coherence can be used to simplify the visible surface calculation. First, since scene data consists of surfaces, visibility is constant over contiguous areas of the screen that correspond to the visible portions of surfaces; if these areas can be identified, no further visibility calculation is necessary within the area. Second, the factors that determine visibility tend to be locally constant; thus, by carefully structuring the data, these factors can be maintained incrementally, making the recalculation of visibility at transitions simple. Finally, sequences of images tend to be temporally coherent: information can be retained from one frame to make the calculation of the next easier. Virtually all rendering algorithms exploit some aspect of coherence to speed the rendering process; the Sutherland, Sproull and Schumacker taxonomy of visible surface algorithms is based on the manner in which they exploit coherence.

2.2. A Review of Visible Surface Algorithms

Since 1974, a great deal of research has been done in the area of visible-surface algorithms. This section reviews the state-of-the-art in visible surface algorithms using the Sutherland, Sproull and Schumacker taxonomy as a rough basis.

2.2.1. Edge-Intersection Algorithms

Edge coherence algorithms are based on the observation that the visibility of an edge changes only at easily detectable boundaries: points at which the edge crosses the silhouette edge of a nearer surface. Early edge coherence algorithms [App,67, GaM,69, Lou,70, SSS,74]) were devised as solutions to the hidden-line problem in line-drawn graphics. More recent work [SeG,81, SeW,85] has extended these algorithms to compute analytic visibility by reconstructing polygonal fragments from visible edge fragments.

The edge-coherence visible surface algorithms of [SeG,81, SeW,85] begin by sorting all the edges in the viewing volume by their topmost Y vertex. The algorithms then make a pass through the data from top to bottom, maintaining an active edge list sorted in X that is updated whenever either a vertex is found or two active edges cross. At each such event changes in visibility are easy to determine, since all the necessary information required to determine visibility is localized at the event: for example, if two edges meet, the information necessary to determine the visibility of the segments below the intersection is localized in the two intersecting edges; no other edges affect the calculation.

Visible polygons can be easily reconstructed from this network of visible edges. Each visible edge may be the lefthand edge of a polygon, the righthand edge of a polygon or both. If two left edges are found adjacent in the active edge list, the second must be the edge of a nearer polygon and a right edge, identical to the nearer polygon's left edge but representing the right edge of the farther, clipped polygon, is added to the active edge list; a similar edge is added when two right edges are adjacent in the active edge list. When these new edges are added, the active edge list contains the actual contours of visible polygon fragments. By tracking these contours during the downward pass the analytic definition of visible polygon fragments is produced.

2.2.2. List Priority Algorithms

List priority algorithms use an object-space analysis to determine the priority between surfaces in the scene. This priority is then used to establish precedence between overlapping surfaces in image space. Sutherland, Sproull and Schumacker describe two such algorithms, one by Schumacker et. al. [SBG,69] and the other, by Newell, Newell and Sancha [NNS,72]. Since that time, aspects of both algorithms have been combined in the Binary Space Partition Tree algorithm, first described by Fuchs, Kedem and Naylor [FKN,79] and later developed by Fuchs, Abram and Grant [FAG,83] at the University of North Carolina at Chapel Hill.

2.2.2.1. Schumacker's algorithm

Schumacker's algorithm was developed for use in flight simulators, which characteristically generate many images (frames) of a single, generally static model. Because the model does not change, Schumacker was able to divide the work of determining a priority ordering into two phases: a preprocessing phase, in which inter-object relationships are analyzed without regard to the viewpoint, and a run-time phase, in which the output of the preprocessing is combined with a specified viewpoint to produce a correct priority ordering. The complete algorithm actually consists of two largely independent visibility algorithms: one to determine priority between "clusters" of data and one to determine priority among the surfaces of a single cluster. In this manner, neither algorithm must cope with the entire body of data; the first step acts only on clusters, while the second is applied independently to each cluster.

The inter-cluster priority algorithm introduced the notion of the dividing plane. If two objects are separated by a plane, their relative priority can be determined by simply finding which side of the plane the viewpoint lies on since nothing on the far side of the plane can hide anything on the near side. Based on this notion, a tree can be constructed with the equations of dividing planes at the internal nodes and priority lists at the leaves. Determining a priority ordering is then done by passing down the tree from the root, choosing to go either left or right based on the relation of the viewpoint to the node's plane, which can be determined by examining the sign of the result of evaluating the planar equation Ax + By + Cz + D at the viewpoint.

Priority within a cluster is determined using a priority-graph algorithm. Each polygon of the cluster is tested against every other to determine whether there is any point from which the front face of the first polygon hides the front face of the other. A graph is built based on this test. If the graph contains a cycle, the algorithm fails, and the cluster must be decomposed into smaller clusters. If not, relative priority values can be assigned by traversing the graph. These priorities can then be used at image-generation time to produce a priority ordering of the front-facing polygons.

Between these two algorithms, the Schumacker approach is able to determine a correct priority ordering of all the polygons in the scene. This list is then input to an hardware-implemented scan-line algorithm [SSS,74]. An active polygon list is maintained in a pass down the screen. For each scan-line, the starting and stopping points of each active polygon along the scan-line are incrementally determined and placed in priority order in a list of paired hardware counters. For each pixel, these are decremented, and any with one value less than or equal to zero and the other greater than zero correspond to a polygon visible at that pixel, and the visible surface is simply the first such polygon in the list.

2.2.2.2. The Newell, Newell and Sancha Algorithm

If the polygons in a scene can be completely priority-ordered, they can be added in that order to an image buffer, with higher priority polygons overwriting the effects of earlier, lower priority polygons; a so-called "painter's algorithm". Computing a priority ordering is difficult, however, since the relative priority of polygons doesn't depend on any single quantity. Although distance to the viewpoint can sometimes resolve priority between two polygons (if, for example, they do not overlap in depth), it often cannot; figure 1a and 1b show two polygons with identical depth relationships but with opposite priority orderings. In some cases, polygons may not be orderable at all; figure 1c shows a case of "cyclic overlap", three simple convex polygons that cannot be correctly ordered.

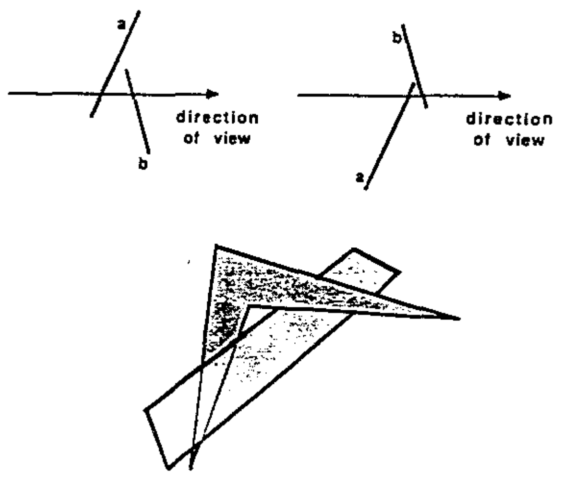


Figure 1. Difficult Cases for Priority Analysis

In 1972, Newell, Newell and Sancha devised an efficient priority sorting algorithm to drive a painter's algorithm. They defined several separate tests which can resolve priority if a correct ordering exists and identifies cases in which no ordering is possible. These tests are applied in increasing order of complexity and expense, thereby eliminating as many polygons as possible from the expensive tests required to distinguish complex interactions between polygons.

The algorithm begins by sorting the scene polygons according to their farthest vertex. This list is then traversed from the last, farthest polygon (the 'candidate' polygon) to the first, nearest polygon, at each step checking that candidate polygon doesn't hide any polygon that precedes it in the list. To do so, the algorithm first applies a depth-overlap test to identify polygons that might be hidden by the candidate polygon. Each such polygon is then subject to a sequence of increasingly expensive tests to determine whether they are hidden by the candidate polygon. If no lower-priority polygon is found, the candidate polygon is produced for the painting process and the algorithm repeats on the remaining list. Otherwise, the lower priority polygon is placed at the end of the list, and the algorithm repeats on the adjusted list. If unorderable polygons are encountered, they are divided until an ordering is possible.

2.2.2.3. The Binary Space Partitioning Tree Algorithm

The Binary Space Partitioning algorithm is in several ways quite similar to the separating-plane algorithm used by Schumacker to prioritize clusters. Like the Schumacker algorithm, the BSP-tree algorithm uses a pre-processing step to extract viewpoint independent information about the scene; furthermore, this step uses a tree of separating planes similar to that of Schumacker. The idea of separating planes is extended to include an intermediate level of priority: no polygon lying in the separating plane itself can be hidden by polygons lying on the far side of the plane from the viewpoint; similarly, no polygon that lies on the near side can be hidden by any polygon in the splitting plane. This idea is the basis of a recurrence rule: if the polygons on the far side of the plane are produced in increasing priority order, followed by the polygons lying in the splitting plane, followed by the polygons on the near side, the entire list will have been correctly prioritized.

The BSP-tree structure consists of a binary tree of polygons with the property that each polygon in the left subtree of a node lies entirely on one side of the plane of the node polygon, while every polygon in the right subtree lies on the other. This data structure is built recursively: given a list of polygons, a root polygon is chosen and the remainder of the list is separated into two sublists along the plane of the root polygon; if any polygon straddles the plane, it is divided into two fragments which are each passed to the appropriate sublists. The process then recurs on each sublist, and the resulting subtrees are attached to the original root node.

An intelligent in-order traversal of this tree structure is then used at run-time to generate the priority ordering. When a node is encountered, the viewpoint is tested against the plane of the node polygon. First, the subtree containing polygons on the far side of the splitting plane is recurred on, producing a correct priority ordering of the polygons found on that side. The polygons lying in the plane are then output, followed by a recurrence on the subtree corresponding to the near side of the plane. At the conclusion of this recurrence, the polygons in this entire tree will have been produced in priority order.

The algorithm used to choose a polygon from a list to serve as a node polygon is crucial to the performance of the BSP-tree algorithm. Since any polygon that straddles a separating plane must be divided along the plane, a poor choice of splitting plane can conceivably double the number of polygons in the polygon list at each recurrence of the treemaking algorithm. Although we at UNC-Chapel Hill have investigated several intelligent algorithms for this choice, we have found remarkably good results by simply testing a small number of polygons from each list and using the best of these. Table 1 shows statistics found by applying the BSP-tree treemaker algorithm to several models. Five candidate polygons were tested from each list, and the one which split the fewest others was selected to serve as the separating plane for the list.

database	input polygon count	output polygon count
OldWell	356	382
Building Model	6,224	9,108
Ship Model	2,538	14,255

Table 1. Polygon Proliferation in BSP-tree Algorithm

Although the BSP-tree algorithm and Schumacker's splitting-plane tree algorithm bear a great deal of similarity, they differ in several significant ways. A major difference is the manner in which separating planes are chosen. The Schumacker splitting planes are chosen to separate clusters, and are defined as a part of the modelling process. The BSP-tree, designed to operate completely automatically on arbitrary sets of polygons, derives separating planes from the planes of model polygons, and accepts the expense of clipping polygons when no perfect separating plane is be found.

A related difference lies in the structure of the tree itself. The Schumacker algorithm uses separating planes to divide space into volumes in which priority among the clusters is constant; a downward path through the tree leads to the leaf node corresponding to the volume containing the current viewpoint, where a priority ordering for that volume is found in the form of a list. This solution works well when there are relatively few such volumes and where the number of objects in each priority list is limited, as is the case when there are relatively few clusters in the data. If, however, this algorithm was to be used to prioritize a large set of polygons, there would be many such volumes, each requiring a priority list containing a reference to every polygon potentially visible in that volume; potentially requiring a huge amount of memory. Instead, the BSP-tree algorithm places polygons at the tree nodes where their planes are used, and traverses the entire tree; polygons are prioritized by the order in which they are encountered during the traversal. Thus each polygon requires only a single reference (actually, one reference for each polygonal fragment produced during the treemaking process) at the cost of traversing the entire tree, rather than simply tracing a downward path.

2.2.3. Depth-Priority Algorithms

As noted earlier, a depth test alone cannot correctly prioritize a set of surfaces in the general case since surfaces span a range of depth. However, if the depth test is taken over a sufficiently small region, this ambiguity can be avoided. At the time of the Sutherland, Sproull and Schumacker paper, two strategies for using depth priority were known: scan-line techniques that apply the depth test at infinitesimal sample points, and Warnock's screen-subdivision technique that divides the image plane until a depth test in each division is unambiguous. Since that time, each of these approaches has been extended to produce continuous visibility.

2.2.3.1. Area-Subdivision Algorithms

Area-sampling algorithms take advantage of the fact that visibility is often coherent over large areas of the screen to lessen the costs of the visibility calculation. The first such algorithm, invented by Warnock, tests the visibility in rectangular "windows" in the image plane, originally the entire screen area. Using a simple test, the algorithm determines whether a single polygon is visible in the window. If so, the visible polygon is produced, the rest discarded and algorithm has correctly determined visibility over the entire window. Otherwise, the window is divided into four subwindows along axis-oriented lines, the list of polygons that intersected the original window are divided into their intersections with the subwindows, and the process is recursively applied to each. The recursion also stops when the window shrinks to the area of a sample-square, in which case the nearest polygon intersecting the window is produced.

An unfortunate aspect of the Warnock algorithm is that, because very few polygons will be axis-oriented, the algorithm frequently recurs to sample-square sized windows along edges. Thus, visibility data in the areas where it is most needed is lacking; the algorithm area-samples over the large, flat areas but in effect point samples at edges. In 1975, Ivan Sutherland addressed this problem by patenting an algorithm that recursively subdivides volumes, initially the entire viewing volume, along planes defined by the viewpoint and some edge in the volume. Whenever a polygon is found that spans an entire volume, it is used as a rear clipping plane since it hides polygons (and portions of polygon) that lie behind it. Eventually, only a single polygon will remain inside each volume, and the recursion terminates.

Both these algorithms share the property that even a completely visible polygon may be fragmented during the visible surface calculation. This problem is avoided (to a large extent) by the 1977 algorithm of Weiler and Atherton [WeA,77]. Beginning with a rough depth sort, the Weiler-Atherton algorithm clips the entire polygon list on the silhouette of the first, producing two lists: one of polygons lying inside the silhouette and one of polygons lying outside it. The inside list is checked to verify that the clip polygon obscures all the polygons lying inside its silhouette; if this test fails, the polygons which are hidden are removed from the list, the clip polygon placed at the end and the algorithm is recursively applied to the result. Finally, the algorithm recurs on the outside list.

2.2.3.2. Scan-line Algorithms

Whereas area-sampling depth-priority algorithms determine visibility by finding areas over which a single surface is entirely closer than any other surface that intersects the area, scan-line depth-priority algorithms determine visibility over spans of each scan-line by comparing depth at the points where visibility might change.

All scan-line algorithms follow a certain form. The set of polygons is first sorted according to their topmost Y coordinate, and as the image is formed scan-line by scan-line a list of currently active polygons is maintained; each time a new scan-line is begun, the polygons that were completed on the previous scan-line are culled and the polygons that begin on the new scan-line are added. In this manner, only the polygons that actually intersect a scan-line are considered during the generation of that scan-line.

For each scan-line, the intersection of every currently-active polygon with the scan-line is computed and sorted in X from left to right. These segments are then analyzed to identify spans along the scan-line in which a single segment is visible. At this point the several scan-line algorithms diverge. One approach, typified by the Romney and Bouknight algorithms, define spans as the gaps between places where the visibility might change: the points at which polygon segments begin and end along the scan-line. If interpenetrating surfaces are disallowed, visibility can be correctly determined across any such span by simply testing the depth of each polygonal segment at the start of the span. However, many such edge crossings will not change the visibility; thus the same polygonal segment may be visible over adjacent spans. The Watkins algorithm finds the righthand end of the span by first fixing the left hand end at a point at which visibility changes, then marching right, adding newly active segments and deleting inactive segments, until an event is found that terminates the span: either the arrival of a nearer segment or the departure of the currently visible segment.

All three of these early scan-line algorithms point-sample in the vertical direction. Scan-lines are considered to be one dimensional and polygon segments are line segments bounded by the intersection of the polygon edges with the scan-line. Given this model, only a single polygon can be visible at any point along the scan-line, and samples are determined by the segment visible at each pixel center. In 1978, Catmull extended the scan-line approach to provide area-sampling.

Instead of considering scan-lines to be one-dimensional, he considered them to cover rectangles on the screen; intersecting a polygon with a scan-line thus produces a polygonal segment, rather than a line. Like the earlier algorithms, these are sorted in X, and an active-segment list, sorted in depth, is maintained as the algorithm passes along the scan-line. For each pixel, this list is examined: if the topmost polygon entirely covers the square corresponding to the pixel, it alone affects the pixel. If the topmost segment only partially covers the pixel, and the next segment completely covers the pixel, the area of the frontmost is computed and used to weight the contributions of the first and second segments. Finally, in complex cases where both the first two segments fail to completely cover the sample square, the intersection of each segment with the sample square are passed to a "pixel integrator", which performs a two dimensional edge-clipping algorithm to determine the contribution of each partially visible segment to the sample.

Previous Work in Real-Time Image Generation

Since no existing general-purpose computer can provide the computation speeds needed for real-time realistic image synthesis, systems designers have relied on the use of highly concurrent hardware to provide the necessary performance. In the past, this has required vast amounts of specially designed circuitry and, as such, these systems have been extremely expensive. With the advent of very large-scale integrated circuitry, however, it is possible to distribute the image generation problem across large assemblages of identical components and thereby lower the cost of such systems [FuP,81].

This chapter will review the state of the art in real-time, high-quality image synthesis. This will be done in two sections: the first covering commercially available products, and the second covering several current research projects in the area.

3.1. Current Commercial Systems

The high cost of current real-time, high-quality image generators has limited their use to a very few applications, primarily the generation of out-the-window views for flight simulators. This application has very demanding performance requirements; the system must provide a degree of realism sufficient to provide effective training including the simulation of atmospheric effects such as haze, it must do so without distracting the pilot with artifacts of computer generation, and it must provide the extreme responsiveness of a jet fighter in flight. In turn, flight simulation offers some simplifying characteristics. Exact realism is not required; stylized imagery that provides the necessary queues to the pilot suffices [Sch,83]. The geometry of the world model can be limited to a relatively few structures on a ground plane and, since exact realism is not necessary, the model may be hand-tooled to simplify the image-generation calculations.

Unfortunately, many of the most interesting details of flight simulator image-generation systems remain proprietary to their manufacturers. The remainder of this section will cover first characteristics shared by most of these systems, followed by more in-depth discussions of the common scan-line oriented systems and then of the current state-of-the-art Evans and Sutherland CT-5 system, one of a very few such systems that does not follow a scan-line approach.

3.1.1. Flight Simulator Overview

The task of a flight simulation image generation system is simply to provide sufficiently realistic out-the-window views for pilot training. As such, two major tasks are required: first, the maintenance of a data base that adequately represents the terrain over which the simulator is to "fly", and second, the generation of the views themselves. Based on this division of tasks, most flight simulation image generators consist of two major components: a database manager, generally implemented on a general-purpose computer, and an image generation system, consisting of large amounts of special-purpose hardware. This image generation component is itself often divided into two major sub-components: a geometrical processor, responsible for most of the object-level geometrical manipulations, and a renderer, responsible for the conversion of geometrical data to pixel values.

3.1.2. Modelling for Flight Simulation

World models used in flight simulation applications must be carefully tailored both to the needs of pilot training and to the realities of real-time image generation. To provide the necessary realism for flight training, the system must support models that contain the cues that the pilot uses to judge his position and motion. Yet, to conform to the needs of real-time image generation, these models must be limited to a relatively small number of geometrical primitives; ASUPT, a system delivered to the airforce in 1974, was capable of generating images containing 2500 edges, while the more current CT-5 system can handle 2500 polygons at a 60hz. update rate [Sch,83].

Fortunately, many tricks can be used to simplify the problem. Textures can be used to give relatively simple geometrical models the appearance of more complex objects; for example, a runway can be modelled simply by a single polygon accompanied by a texture that adds realistic skid marks to the runway without adding geometrical complexity. Furthermore, objects in the model may be defined at varying levels of detail; in this manner the system avoids getting bogged down in the details of objects that are invisible due to their distance from the pilot.

A very important simplification used in flight simulation systems stems from the fact that, to a very large extent, the model remains static through the course of a simulated flight; airport buildings seldom move with respect to each other. Given this property, information about visibility can be determined before actual image generation commences, thereby simplifying and speeding the visibility calculation. Two such algorithms, one developed by Schumacker [SBG,69] and one by Fuchs, Kedem and Naylor [FKN,79] are described in the previous section on image generation algorithms.

3.1.3. A Flight Simulator: the Evans and Sutherland CT-5

Unlike most systems, the Evans and Sutherland CT-5 system has abandoned the scan-line sequential order of earlier simulation systems for a *feature-sequential* ordering. For this reason a double-buffered approach is used: while a frame is being formed in one buffer, video generation takes place from the other. When a frame is complete, the system swaps the two buffers.

Algorithm

Input to the CT-5 video-generation system consists of a set of objects (generally modelled with polygons) ordered in decreasing priority; thus far objects are encountered later in the processing sequence than nearer objects that may occlude them. As each object is processed in turn, a high-resolution bitmask corresponding to the area of the image plane lying within its boundary is generated. This mask is then compared to a stored mask that represents the portion of the image plane that has been covered by previous, and hence nearer, objects. Two results are produced: the object visibility bitmask, representing the portions of the current object that are not obscured, and a new coverage bitmask, which replaces the old in storage.

The object visibility mask, along with shading information, is then passed on to a spatial filtering processor. For each output pixel, the integral of the product of a filter kernel with the visible portion of the object is computed, using the visibility mask to define the domain of integration, and the result is summed into the image buffer. The algorithm is therefore similar to that made possible by the use of an analytic visibility algorithm: the filtering integral is decomposed into the sum of integrals over each visible surface intersecting the filter kernel; the difference is in the representation of the visible fragment.

Image-Plane Parallelism: Spans

The CT-5 divides the image plane into a set of rectangular contiguous regions called 'spans', each corresponding to a set of pixels of the final image. The span defines the area over which the system operates in parallel. For each object in the scene, each span intersecting the object is considered in turn. A span processor computes the coverage bitmask within the span and compares it to the coverage mask for the corresponding span in the image plane; the results of the masking are passed on to the spatial filter processor in span-sized chunks. The filter processor then produces the contributions to pixels within the span in parallel. The image memory itself is structured to give the spatial filter access to each pixel in the current span in parallel so that pixel contributions to each pixel within the span can be summed in parallel.

3.2. Current Research: Parallel Architectures for Image Generation

The advent of VLSI technology has dramatically increased our capability to provide parallel computation. Today we can envision systems containing hundreds, thousands or even millions of separate processing elements, all cooperating to perform a single task. Yet taking advantage of this technology remains a challenge. Graphics, with its huge computational expense and its inherent regularity, seems a natural candidate for VLSI implementation. In this section we will review several of the promising research projects in parallel image generation.

Prospects for concurrency abound in image generation. In general, these seem to fall into two classes: those that divide the computation by dividing the image plane among parallel processors, and those that distribute different parts of the model to separate processors.

3.2.1. Image-plane Parallelism

A fundamental limitation with many frame-buffer systems is the bandwidth into the image memory. Regardless of how many processors are generating pixel values, if these pixels cannot be stored into the frame buffer, the processors will be forced to wait and the potential gains are voided. One solution to this problem is to separate the image memory into separate components that can be cycled independently, then grant each image-generation processor sole access to a component.

If standard off-the-shelf memory chips are used the number of components into which the frame-buffer memory can be divided is limited [Whi,84]. For example, if 16Kx1 bit chips are used, a 512x512x24 frame buffer requires 384 memory chips. In order to grant parallel access to the bits within a pixel, these chips must be organized as 16 memory components consisting of 24 memory chips each, limiting the number of parallel image generation processors to 16. The following table shows the number of separate components that a 512x512x24 frame buffer can divided into for several common chip configurations.

chip organization	number of chips required	number of independent components
16Kx1	384	16
64Kx1	96	4
64Kx4	96	16
256Kx1	24	1
256Kx4	24	4

In this section, several research projects in image-plane parallelism are discussed. The first three of these assume the use of standard memory components and, as such, are rather limited in the number of parallel processors that can be used. The last project, Fuchs' Pixel-planes, uses custom VLSI integrated circuitry to couple a minimal processor with each pixel in the system.

3.2.1.1. Standard Memory Chip Systems

If we cannot generate a complete, full resolution image in real time, perhaps we can generate many low resolution images in parallel which together form the output image at full resolution. This notion led Fuchs [FuJ,79] to propose an interlaced image plane subdivision algorithm. If the system contains IxJ processors, each processor is assigned every Ith pixel on every Jth scan-line; if the target resolution is MxN, each processor effectively controls a (M/I)x(N/J) raster. To generate an image, each polygon is broadcast to the entire set of processors. Each processor then scan-converts the polygons into the pixels under its control. Since polygons cover contiguous areas of the screen, the work is divided roughly equally among the separate processors.

Parke [Par,80] noted that this system required every processor to perform the virtually identical startup calculation necessary to scan-convert the polygon. If each polygon could be handled by a single processor, with the many parallel processors operating concurrently on different polygons, this overhead is reduced. He proposed a system in which the image plane is divided into contiguous areas, with a separate processor assigned to each. Above this processor grid is a splitter tree which distributes each polygon to the processors whose screen-area intersects the polygon. In this manner, other processors can concern themselves with other polygons; the total number of pixels written remains the same, but only a fraction of the per-polygon setups need be done. Unfortunately, this contiguous area subdivision strategy is very sensitive to the distribution of polygons across the image plane. If a disproportionate share of the polygons lie in a single area, the time required to generate the complete image will be constrained by the time required to process the most complex area; other processors will remain idle while this area is completed.

Clark and Hannah [ClH,80] returned to the interlaced approach, but used a hierarchy of control processors to distribute the scan-conversion overhead. At the topmost level, a parent processor sorts the edges of each polygon (which are required to be convex) from left to right across the screen, and computes forward difference equations for each. These are then passed down to a set of I column processors. Each column processor, in control of every Ith column, then determines the extent of the polygon down the first column under its control that intersects the polygon. This information is then passed down to a set of J row processors, each of which controls every Jth pixel along each of the columns under control of the column processor; the row processors then write the appropriate pixels. While this is happening, the column processors are determining the extent of the polygon down the next column under their control; as soon as the row processors are done with the previous column, this information is passed to them and they continue on the next column. While the current polygon is handled by the column and row processors, the parent processor is preparing the next polygon to be passed to the column processors when they complete the previous polygon.

3.2.1.2. Custom Memory Chips: The Pixel-planes Architecture

Certainly the ultimate image-plane subdivision scheme is to assign a separate processor to each pixel in the system. However, due to the very large number of pixels in a graphics system, each such processor must be kept extremely small and accordingly simple. The Pixel-planes project achieves a practical balance between processor area and function by utilizing a unique approach to image generation that reformulates many of the time-consuming pixel operations as the solution to planar equations.

The Pixel-planes approach addresses the three steps that constitute the bulk of time in most image generation systems: the determination of the set of pixels that lie inside a convex polygonal boundary (the scan-conversion calculation), the subset that are not occluded by nearer surfaces (the visible-surface calculation) and the correct color for each pixel in the visible subset (the shading calculation). Each of these steps can be formulated as the results of evaluating one or more planar equations for each pixel in the image. An edge of a polygon can be expressed by the equation $Ax + By + C$ = signed distance of the point (x,y) to the line; pixels lying inside a convex polygon are exactly those which lie on the same side of each edge (eg. those that evaluate positive for every edge of the polygon). The depth of a (planar) polygon can be formulated as $Ax + By + C = \text{depth}$; given the depth of the current polygon at each pixel, a Z buffer algorithm can be used to determine visibility. Finally, planar (Gouraud) shading can be similarly represented: Ax + By + C = intensity. Thus, all three of these crucial steps can be reduced to little more than the evaluation of planar equations at the (x,y) location of each pixel in the scene.

The key to the Pixel-planes architecture is the use of a bit-serial multiplier tree structure which provides each pixel processor with the result of evaluating a planar equation at the pixel's (x,y) location. This tree inputs two coefficients B and C and produces, at the y-th leaf node, the value (By + C). By inputting this result into the constant input of a second tree at each leaf node and a coefficient A at the multiplier input, the values Ax + (By + C) appear at the x-th leaf of the y-th tree, and is received by the pixel processor associated with the (x,y) pixel. Given this information, the pixel processor is only required to do very simple single bit operations such as maintain an 'inside' flag as edges are processed, compare the received depth information against the locally stored old depth, and (if the pixel is inside the polygonal boundary and nearer than the previous closest) store the received intensities into its color buffer.

Each of the 128 pixels in the current version of the Pixel-planes (version 4.2) chip contains 72 bits of general-purpose memory (usable for both depth and intensity buffers). Of the total area of the chips, about 70% is devoted to memory, 12% goes to the per-pixel processors and 18% is required for the bit-serial multiplier trees [Eyk,86]. In order to simplify the actual implementation of the system, about half the area devoted to the multiplier is is redundant.

3.2.2. Object-Space Parallelism

A second approach to parallelism in image generation is to associate a separate processor with each individual object in the scene which handles the computations necessary for that one object. Although this scheme simplifies the processing of any one primitive, since it is done independently of the rest, the complete process still must resolve interdependencies between the different objects (primarily visibility); thus the manner in which the object processor's results are combined becomes the central issue. In this section, two such architectures are discussed. The first, due to Cohen and Demetrescu [CoD,80] and later extended by Weinberg [Wei,81], is designed for polygon-based models. The second, proposed by Kedem [KeE,84], processes data modelled using Constructive Solid Geometry (CSG) techniques [ReH,77a].

3.2.2.1. The Polygon-Pipeline Architecture

The Cohen and Demetrescu system consists of a pipeline of polygon processors each containing the definition of a single polygon (required to be trapezoidal). This pipeline passes tokens corresponding to pixels through a chain of polygon processors in raster (eg. row-major) order at video rates. When a token is received by a polygon processor, the processor computes the depth of its own polygon at the pixel center and compares the results with the depth contained in the pixel token. If the processor's polygon is nearer to the viewpoint than the token's contents, the processor replaces the token's contents with its own depth and shade information. When the token has passed through the entire pipeline, its contents reflects the nearest, hence visible polygon. Since the pipeline operates at video rates, the output of the pipeline may be sent directly to the video generation system.

In order to function correctly, the polygon processors must be be able to determine both whether a given pixel's (x,y) location lies inside its polygon and then the correct polygon depth and shade information at that location. Since the pipeline operates in raster order, these functions are greatly simplified by using incremental methods; by offloading the calculation of delta values to a preprocessor, the polygon processors need do little more than integer adds and compares. The polygon processors can therefore be quite small; they require little more than an adder, a few registers and a small control PLA.

Generating real-time video requires that the trapezoids descriptions be updated from frame to frame. This involves much more complex operations than outlined above for the polygon processors, including geometrical transformation and lighting calculations. Since these are applied at the polygon level, rather than the pixel level, there are relatively few such calculations necessary; they therefore are done in a preprocessing step. Since the pipeline is idle during the video blanking periods, it serves as the conduit to pass this information to the individual polygon processors during vertical blanking time.

Weinberg extended this architecture to provide for anti-aliasing by passing chains of tokens corresponding to fragments of trapezoids that partially cover a pixel, rather than a single token corresponding to the nearest trapezoid that intersects the center of the pixel. Partial coverage is determined by tracking the edges of the trapezoid across the scan-line; if one or more of the trapezoid edges passes through the pixel, the pixel is known to be partially covered. The system then inserts the fragment into the token list for the pixel in depth-order. If the fragment covers the entire pixel, the remainder of the token list is discarded. At the output of the pipeline, a set of filter processors uses this information to determine a pixel shade that approximates the average shade over the visible fragments.

3.2.2.2. The Kedem CSG Machine

Kedem's ray-casting machine is a special-purpose architecture designed to process Constructive Solid Geometry models, providing not only information necessary to render images of the model, but also other important information such as volume and center of gravity. Constructive Solid Geometry is a technique for modelling which uses boolean union, intersection and difference operators to combine geometric primitives to form solid objects. A CSG definition takes the form of a tree, with leaf nodes corresponding to geometric primitives and internal nodes corresponding to operators that combine the objects defined by the two subtrees.

The Kedem CSG machine instantiates this tree directly in hardware. At leaf nodes are Primitive Classifier (PC) processors that determine the intersection of the primitive object with lines (for example, parallel rays extending from pixel centers into the scene for an orthographic projection of the model). As these intersections are determined, they are passed up to the internal nodes of the tree (Combine Classifier processors, or CCs) which combine them according to the operation specified at that node, again passing the results up. When the results emerge from the root of the tree, they correspond to segments of the original ray that lie inside the solid model. These segments can then be used for many applications; the near end of the nearest segment corresponds to the visible surface and can be used to generate an image of the model.

Perhaps the most interesting problem in this research project is the manner in which an arbitrary CSG tree is instantiated in on a fixed hardware structure. Since the hardware must be suitable for any CSG-modelled object up to some limit in complexity (say a fixed limit on the number of geometric primitives used as building blocks), a general tree structure which makes no assumptions about the form of the tree must be used. Yet the sparseness typical of CSG models makes the use of a complete binary tree very inefficient. Instead, a reconfigurable tree structure is called for.

Kedem has devised a structure on which any binary tree of less than N leaf nodes can be instantiated. This consists of a grid of log(N) rows of N CC processors each, with PC processors attached along the lower edge. The upper left CC processor corresponds to the root node of the tree and passes final results to the outside world. The parent link of each CC processor can be connected to a child link of either the CC to its left or the CC above.

To place an arbitrary binary tree of less than N leaf nodes on this structure, it is first made right-heavy (eg. each right subtree contains at least as many leaves as the corresponding left subtree). The tree is then traversed in pre-order. Left links of each internal tree node are connected down the grid, and right links are connected across to the right, to the first unoccupied column of the grid. When a leaf node is encountered, it is linked through any remaining rows of CCs to the PC at the bottom of its column.

The Development of a Parallel Polygon Clipper Algorithm

The goal of this dissertation research is to develop algorithms and architectures for the real-time synthesis of images with high-quality anti-aliasing and texture mapping. In order to achieve these performance goals a specialized visibility algorithm has been developed. In the first section of this chapter, the considerations and background that led to the development of this algorithm are detailed. The second section details the algorithm itself, along with results establishing the effectiveness of the algorithm. The next chapter will present the architecture of a system component designed to apply this algorithm to complex models in real time.

4.1. Desiderata for a Visible Surface Algorithm

We have stated several major goals in this dissertation: that the system must produce imagery with correct anti-aliasing, it must support the use of many textures to lend realism to the resulting images, and it must operate in real time. From these goals we can identify two major desiderata for the visibility algorithm.

Correct Anti-Aliasing As we saw in chapter 1, anti-aliasing requires that the image function be low-pass filtered to remove aliasing frequencies before raster sampling. To do so, the underlying image function must be convolved with a low pass filter kernel. To compute this convolution, we need a representation of the underlying image function. Virtually all current image-generation algorithms use a high resolution tabulation to represent the image function for convolution. To avoid the pre-sampling required to generate this tabulation, we would like to use an analytic representation of the underlying image function which can be filtered to any desired level of accuracy. This requires that a visibility algorithm be used which determines visibility over continuous areas of the image plane.

Real-time Performance The visibility algorithm must not be a bottleneck that prevents the system from operating in real time.

The first of these two considerations leads to the choice of an analytic visibility algorithm to serve as the basis for this system. We must now consider how to meet the second: we need an area-sampling visibility algorithm that can operate in real time on complex images. In order to provide the necessary performance, it will be necessary to distribute the algorithm over a set of cooperating parallel processors.

4.2. Desiderata for Parallel System Structure

If such a parallel implementation of a visibility algorithm system is to be both practical and efficient, the algorithm must be decomposable over a set of parallel processors in an efficient, practical manner. This system should meet several important goals.

Replicated Components Rather than build many different hardware components, each performing a different task, the system should consist of many copies of a single component design, thereby minimizing both design time and system cost.

Local Communication Structure In cooperating to perform the overall task, the communications flow between the processors must be local, avoiding the need for an expensive and complex general communications structure.

Efficient Utilization The many cooperating processors should share the computational load evenly.

Extensibility The system should be easily extensible for applications of varying complexity.

4.3. Previous Algorithms

Of the several types of visible-surface algorithms presented in chapter two, three provide area sampling: the visible edge-based algorithms of Sechrest and Greenberg [SeG,81], and Sequin and

Wensley [SeW,85], the area-decomposition algorithms of Sutherland [Sut,75], Warnock [War,70] and Weiler and Atherton [WeA,77], and the scan-line method of Catmull [Cat,78].

In considering how these algorithms might be implemented on a highly parallel architecture, it seems unlikely that either visible edge or scan-line based algorithms might be implemented efficiently on parallel hardware. At the center of each of these algorithms is a complex data structure formed in an inherently sequential top to bottom, left to right pass through the image. When these algorithms must make a decision based on some event in the image (such as encountering the topmost vertex of polygon in its downward pass) the correct response depends not only on the local event, but also upon the state of a data structure formed earlier. Furthermore, the effect of the response may not be local; it may require the modification of this global data structure. It is hard to imagine that the management of this data structure might be distributed over concurrent processors.

The remaining algorithms of Sutherland and of Weiler/Atherton, however, are based on a "divide and conquer" approach that can provide the basis for an effective parallel implementation.

4.3.1. Screen-Area Decomposition and Parallel Execution

Perhaps the greatest contribution of Warnock's work was the realization that, if the visibility problem across the entire image was too complex to calculate directly, the problem may be simplified by considering smaller and smaller subareas of the image. Although his algorithm was limited by the technology of the time, calling for simple rectilinear area subdivision (since multiplies could be replaced by a logarithmic midpoint-subdivision algorithm requiring no multiplications) thereby producing poor results on angled edges, his "divide and conquer" provided the basis for important later work by Sutherland and Weiler / Atherton.

"Divide and conquer" algorithms are very well tailored to parallel execution. By dividing the problem into independent subproblems of like kind, the subproblems can be distributed to subprocessors identical to the parent. Furthermore, such a recursive decomposition of the problem contains an inherent hierarchy that can provide a basis for a simple local communications structure. These considerations have led to the narrowing of consideration to algorithms such as Sutherland's and that of Weiler/Atherton which are based on a recursive geometrical decomposition of the visibility problem.

4.3.2. Sutherland's Algorithm

The Sutherland algorithm is based on the recursive decomposition of the viewing frustum based on planes defined by the viewpoint and polygon edges or, at appropriate times, by the planes of polygons that intersect all sides of such a volume and therefore "cap" the volume. Each "side" plane divides a volume into two sub-volumes with the visibility in each completely independent of visibility in the other. Much like Warnock's algorithm, this approach recursively simplifies the visibility problem by considering smaller and smaller independent subproblems until a point is reached where the subproblems can be solved directly.

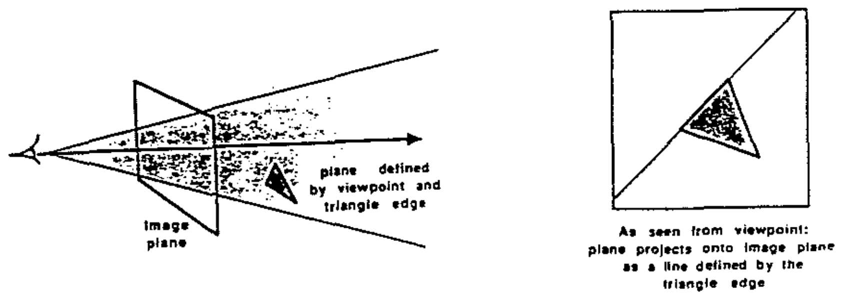

plane defined by viewpoint and triangle edge
image plane
As seen from viewpoint: plane projects onto image plane as a line defined by the triangle edge

Figure 1: Volume division by a separating plane

The initial input to Sutherland's algorithm is a single list of polygons lying inside an open-ended viewing volume, bounded by the planes defined by the viewpoint and the edges of the screen. From this list, a polygon edge is selected to define a separating plane to be used to divide the volume. Note that the plane derived from this edge projects in the image plane as the line containing the edge of the projected polygon; in this manner, the algorithm may be thought to divide the image plane along the lines defined by projected edges.

Sutherland was granted a patent in 1975 for a parallel machine based on this algorithm. Figure 2 shows a rough diagram of that machine.

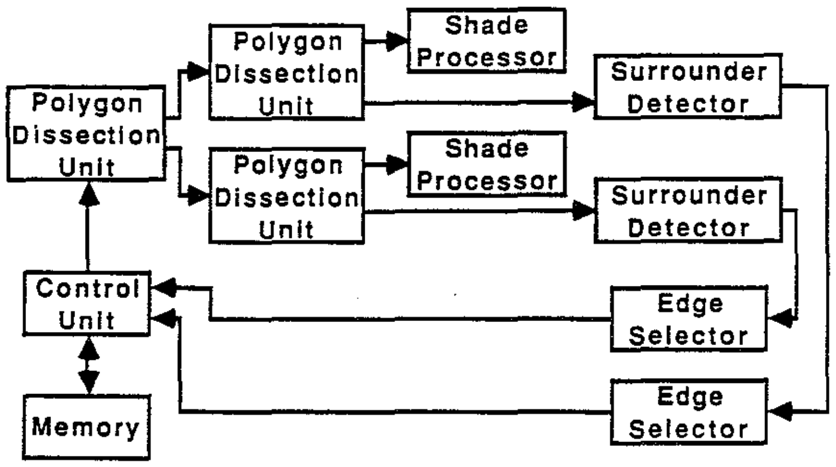


Figure 2: Sutherland's architecture

As the list of polygons passes through the splitter, polygons are divided along the splitting plane and the fragments passed to the two "surrounder detectors" that look for polygons that cap the frustum. When such a polygon is found, the plane of that polygon is used by the associated second stage splitter to remove polygons lying behind this plane from the list. If a subsequent surrounder is found in front of the earlier surrounder, it replaces it in the second stage clipper. The result of this process is two sublists corresponding to the two new subvolumes, and these lists are placed back into memory[1]. When the final polygon of a list is passed, another list is selected and the operation continues.

This algorithm shrinks the volumes under consideration in two ways: first, by splitting the volume by planes passing through the viewpoint and dividing the volumes, and second, by finding surrounder polygons to use as "caps", thereby narrowing the volumes in depth. An unfortunate aspect of this second form is that any fragments lying behind the first surrounder in a list will be

[1] Note that since the polygons input to the algorithm are not assumed to be sorted, polygons lying behind the surrounder but which are previous to it in the list will not be discarded. However, as the algorithm is reapplied to the resulting list the algorithm will eventually remove them.

passed by the algorithm; only farther fragments that trail the surrounder in the list will be eliminated by it. Therefore, even if the surrounder completely hides every other fragment in the list, this will not be discovered until the list has made several further passes through the splitting process, needlessly fragmenting the surrounder.

A very important consideration is the manner in which an edge is selected from the polygons lying inside a volume to define the plane to use to separate the volume. In accordance with the overall strategy of shrinking the volumes to simplify the visibility calculation inside each, the best plane to choose would be one which divides the volume evenly, thereby achieving logarithmic behaviour. The algorithm approximates this choice by using the edge which has the greatest extent perpendicular to the plane which divided the parent volume and produced the current volume. This information is a natural byproduct of the earlier split; in deciding which side of the plane each vertex lies on, the perpendicular distance of the vertex to the separating plane must be calculated. By retaining this information, the edge to use on the subsequent split is determined during the previous split.

This selection criteria hinders the prospects for concurrency in this algorithm. The edge defining the plane to be used to split a volume is not identified until the split which produces the volume is complete. Thus, as the list of polygons residing within a volume is split, the two generated sublists must be saved in memory until the former split is complete.

A further problem encountered in the Sutherland algorithm is the possible proliferation of polygonal fragments. Figure 3 shows the image-plane projection of such an event, a scene containing three polygons *X*, *Y* and *Z*.

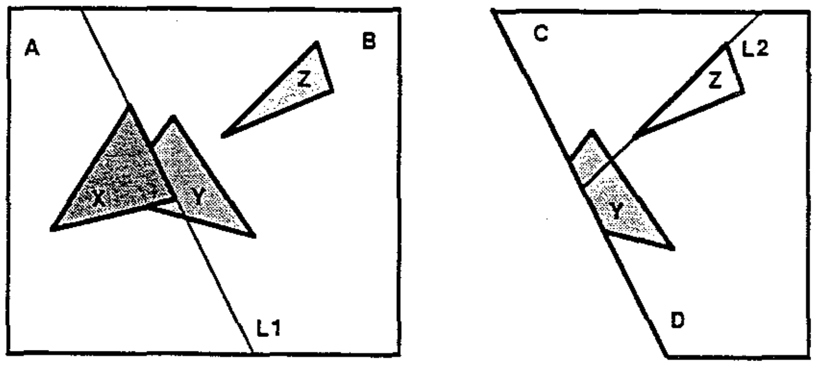


Figure 3: Fragment proliferation in Sutherland's algorithm

Initially, line L_1 is used to divide the original volume into two subvolumes A and B; line L_2 then divides volume B into two subvolumes, labelled C and D. Notice that polygon Y is first divided into two fragments by line L_1, and then one of these fragments is further divided by line L_2. Given an input of three polygons, the Sutherland algorithm produces five polygonal fragments, X, three fragments of Y and Z. No currently published results indicate how the algorithm might perform on reasonably complex data.

4.3.3. The Weiler/Atherton Algorithm

The Weiler/Atherton algorithm again is based on decomposing the screen area into areas in which the visibility calculation can be easily solved. Instead of recursively dividing areas (or volumes) by lines (or planes) like Warnock and Sutherland, the Weiler/Atherton algorithm uses an initial rough depth sort as a basis to "guess" an area of the screen where visibility is trivial, the area covered by the frontmost polygon in the front-to-back sorted list. The screen area is then divided into two areas: the area covered by this first "clip" polygon, and the remainder of the image plane. A simple test is made to verify that depth sort correctly prioritized the polygons in the area of the clip polygon; if not, the polygonal fragments lying inside the clip polygon silhouette and which lie in front of the clip polygon are resorted and the algorithm recurs on the result.

Otherwise, the clip polygon is produced as a completely visible surface. In either event, the algorithm then recurs on the list of polygonal fragments lying outside the clip polygon silhouette.

The Weiler/Atherton algorithm avoids the fragment proliferation problem by using a concave polygon clipper. In the earlier example shown in figure 3, the initial clip would remove the hidden parts of polygon *Y*, producing exactly two polygons, *X* and the convex polygon *Y–X*. While the algorithm may fragment polygons unnecessarily when the initial polygon sort is wrong (figure 4 shows the output of the algorithm if the initial sort places Y before X), in general, only a single fragment will be produced for each partially or wholly visible polygon.

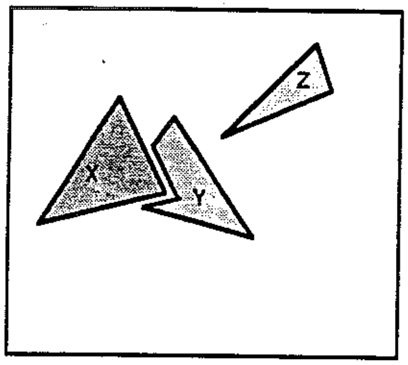


Figure 4: Output of the Weiler/Atherton algorithm on the example of Fig. 3

While the avoidance of polygon proliferation is potentially a substantial advantage, the cost is high. Clipping a convex polygon against the silhouette of another may produce many more edges than were present in the original; worse yet is the fact that the result may in fact have holes (as in figure 5).

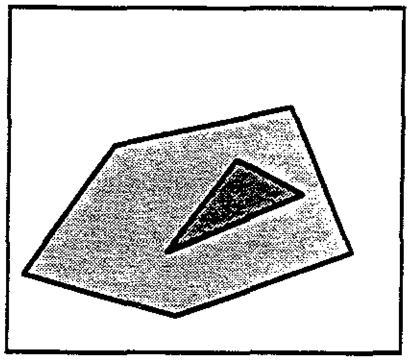


Figure 5: A clip producing a hole

Although the algorithm avoids the necessity of clipping against a convex silhouette with holes by retaining the original silhouette of each polygon, the cost of each successive clip rises quickly. Furthermore, the concave polygon clipper is exceedingly prone to arithmetic error. These factors combine to make the concave polygon clipper very complex.

4.4. A New Algorithm: Culling the Best From Both

By combining characteristics of each of these algorithms we can derive an algorithm which may out-perform either. We will add Weiler/Atherton's use of a polygonal sort to Sutherland's simple convex clip to produce a new algorithm which has produces very good results in simulation.

4.4.1. Overview The general strategy of this algorithm is similar to Weiler/Atherton: an initial rough depth sort is used to choose a polygon which is likely to be completely visible, then the remainder of the polygons will be clipped against its silhouette. Rather than use a concave polygon clipper to do so, however, a simple convex polygon clipper such as that used by the Sutherland algorithm will be used. Each edge of the clip polygon is used to divide a list of polygons into two sublists: a list of polygonal fragments lying on the same side as the clip polygon (the "inside" list) and a list lying on the opposite side (the "outside" list). The actual clip therefore consists of several sequential clips, one for each edge of the clip polygon, with the inside list from one step passed as input to the next. Thus, in the case illustrated in figure 6, the result of clipping against the silhouette of the clip polygon will be four lists of fragments residing in the four "outside" areas and the final "inside" list of fragments lying inside the silhouette of the clip polygon. Again, like Weiler/Atherton, this inside list is examined by a final clip against the plane of the clip polygon to verify the correctness of the depth sort in the area of the clip polygon's silhouette.

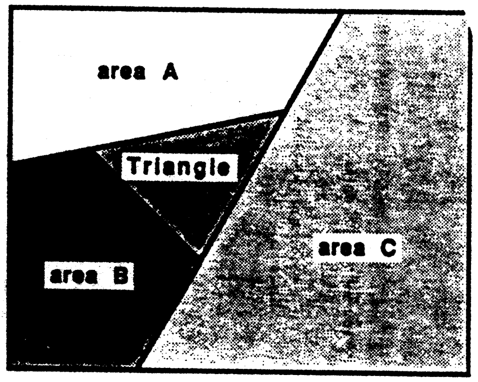


Figure 6. A triangle and the subareas produced by the clip

4.4.2. The Effects of Sorting

Like the Weiler/Atherton algorithm, this algorithm uses a sorting step to maintain polygon lists in a rough priority order. Given this ordering, the algorithm is able to choose a polygon from the list which is likely to be completely visible. The remaining polygons in the list can then be clipped against this silhouette, and a relatively simple check be made to verify the visibility of the clip polygon within its own silhouette. This approach achieves two very important results.

First, this algorithm is likely to produce fewer fragments, and require fewer polygon clips, than the Sutherland algorithm. In order to remove portions of polygons lying behind the visible polygon in an area, polygons intersecting the visible polygon must eventually be clipped against edges of the visible polygon. The Sutherland algorithm, choosing a clip edge from the list based on the longest extent perpendicular to the previous clip, may in fact unnecessarily divide the visible polygon. If some other polygon intersects both an edge of the visible polygon and the edge selected to divide the list, each of the resulting two fragments must eventually be divided by the edge of the visible polygon. Table 1 shows the number of clips required and the number of fragments produced by the two algorithms for several test images.

database	Sutherland Algorithm			Approximate Sort Algorithm			Exact-Sort (BSP) Algorithm			
	in	after clip	out	in	after clip	out	in	after BSP	after clip	out
OldWell	404	404	3764	404	404	917	404	431	431	739
Building Lobby	6255	2565	26924	6255	2565	4128	6255	7159	3816	2721
Ship (Side view)	2531	2531	33418	2531	2531	7764	2531	14219	14129	14197
Ship (Quarter view)	2531	2531	16624	2531	2531	4372	2531	14219	14219	5995

Table 1: Fragment counts for various scenes

The second major advantage makes this algorithm more amenable to a systolic multiprocessor implementation that avoids costly passes through a memory. Whereas the Sutherland algorithm must complete the clipping of a list of polygons against a plane to determine the edge to clip the resulting sublists, this algorithm makes clip edges immediately available from the first polygon produced in either sublist. The sublist can therefore be immediately passed to a subsequent clipper; the edge required by that next clipper is extracted from the first polygon it receives. In this way, no temporary storage is required to save a sublist until the correct clip edge is determined.

4.4.3. Absolute vs. Approximate Sorting

As described above, this algorithm uses an approximate visibility sort as a basis to "guess" a completely visible polygon in each list. We can use a bucket sort based on the depth of the centroid of each polygon to make this sort quick and relatively accurate. Even so, the possibility of errors makes this an expensive approach both in hardware and in execution time. If we are not sure of the sort, the final inside list must be checked to determine whether every polygon in the list lies hidden behind the first polygon. To do so, we must retain the polygon in the pipeline processor that checks this condition until the marker designating the end of the list is encountered; only then is the accuracy of the sort determined. The polygon is then either output as visible or appended to the list of fragments lying in front of it.

Alternatively, if we are sure that the original polygon list was in absolute priority order, the first polygon in the final inside list is guaranteed to lie in front of the remainder of the list. We can therefore dispense with the final clip of the remainder of the list against the plane of the clip polygon, and even more importantly, we needn't provide the pipeline processors with sufficient memory to retain clip polygons until their visibility is verified.

In general, absolute priority sorts are expensive and difficult to come by. The expense of a Newell, Newell and Sancha preprocessing step to order the polygons would be very likely to become the system bottleneck. In many applications, however, we can take advantage of the fact that the model stays fixed while the viewpoint moves. If so, the BSP-tree algorithm can be used offline to prepare a data structure that can be used at runtime to provide an exact priority sort very cheaply.

4.4.4. Adding Transparency Effects

This algorithm can be easily extended to include cheap, effective transparency effects. The light that strikes the eye from a point on a transparent surface can be modelled as a sum of contributions of surfaces visible through the transparent surface, weighted by the transmittance t of the surface, and the transparent surface itself, weighted by $1-t$. As we saw earlier, the correct value for a pixel (x,y) after sampling is:

$$I_{filter}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I_{real}(x,y)F_{low}(x-u,y-v)\,du\,dv$$

which can be decomposed over the silhouettes of visible, opaque polygon fragments:

$$I_{filter}(x,y) = \sum_{\substack{\text{visible}\\\text{polygons}}} \iint_{\text{silhouette}} I_{real}(x,y)\, F_{low}(x-u,y-v)\, dudv$$

If we assume that the image function over the silhouette of a visible *transparent* surface is a linear combination of the contribution of the transparent surface and the surfaces visible through the transparent surface, we get the following expression for I_{real} over the silhouette of the transparent surface:

$$I_{real} = t^* I_{\substack{\text{transparent}\\\text{surface}}} + (1-t)^* I_{\substack{\text{transmitted}\\\text{surfaces}}}$$

If there are transparent surfaces visible behind the foremost transparent surface, this equation can be recursively applied to find the correct contribution of the scene visible through the transparent surface.

Notice that the image function within the silhouette of the visible transparent surface is simply the sum of the contributions of appropriately attenuated surfaces: the transparent surface itself and the scene surfaces visible through the transparent surface. Thus, if we can correctly attenuate the intensity of the light being emitted by each surface, we can express our filtering integral:

$$I_{filter}(x,y) = \sum_{\substack{\text{visible}\\\text{polygon}\\\text{silhouettes}}} \iint_{\text{silhouette}} \sum_{\substack{\text{attenuated visible}\\\text{fragments inside}\\\text{silhouette}}} I(x,y) F_{low}(x-u,y-v) dudv$$

We can then move the sum outside the integral:

$$I_{filter}(x,y) = \sum_{\substack{\text{visible}\\\text{polygon}\\\text{silhouettes}}} \sum_{\substack{\text{attenuated visible}\\\text{fragments inside}\\\text{silhouette}}} \iint_{\text{silhouette}} I(x,y)\, F_{low}(x-u,y-v) dudv$$

Thus, if we can correctly attenuate each polygon fragment visible either directly or through one or more transparent surfaces, these fragments can be passed on to the rendering process.

Unfortunately, the surfaces visible through a transparent surface are generally very difficult to find due to the refraction of light passing through transparent surfaces. If we make the simplifying assumption that the transparent surface does not bend light rays passing through it, the surfaces visible through the transparent surfaces are simply those that would be visible inside the silhouette of the transparent surface if the transparent surface were not there.

Given this assumption, it is easy to find the correctly attenuated set of polygonal fragments lying inside the transparent polygon silhouette: when this algorithm has found a visible surface, it has also found a list of fragments lying inside the silhouette of the visible surface which, normally, is discarded as hidden behind an opaque surface. If instead we attenuate the intensities of the fragments in this list by t and recursively apply the visible surface algorithm to it, we determine the set of correctly attenuated fragments visible through the original transparent surface.

4.4.5. Coordinate Systems for Clipping

The Sutherland and Weiler-Atherton algorithms differ in the space in which they operate: Sutherland operates in the three-dimensional eye coordinate space resulting from the application of the viewing transformations on the original coordinate system, while Weiler-Atherton operates in the screen space resulting from the application of the perspective transformation to the eye coordinate system. Conceptually, the new algorithm can operate in either: the *eye space* can be divided along the *planes* defined by the viewpoint and polygon edges, or the *screen space* can be divided along the *lines* defined by projected polygon edges. Choosing a space to operate in depends on the correctness of the solution and the relative costs.

Initially, planar polygonal surfaces in our input model consist of sets of vertices, potentially with interpolant values (such as texture indices) attached. If the interpolants are linear across the polygonal surfaces, such vertices can be considered planar coordinates points in n-space (x,y,z,i_k). If we define the perspective transformation on the vertices to be:

$$P(x,y,z,i_k) = (x',y',z',i_k') = (\frac{x}{z},\frac{y}{z},\frac{1}{z},\frac{i_k}{z})$$

we find that the transform of a planar surface remains planar and, furthermore, that the transformation is reversible. Therefore, the interpolations necessary for the polygon-clipping operation can be done in perspective-transformed screen space.

The geometrical effect of the perspective operation is to transform viewing rays, extending from the viewpoint out into the scene, to lines parallel to the Z axis. Thus we find that the clipping planes used in a three dimensional clip, defined by the viewpoint and an edge in the scene, are transformed to planes defined by the perspective transform of the edge and the Z axis and is therefore parallel to the Z axis. Given this, the most common operation of the algorithm, the determination of the distance of a point to a clipping plane, can be done without regard to Z, requiring one fewer multiply and add.

We therefore have two choices: we can perform the perspective operation on all the vertices in the model and perform the visibility algorithm on the result, or perform the visibility algorithm on the original eye-space coordinates and perform the perspective transformation on the vertices of visible polygon fragments. In the first case, we save on the expense of the distance calculation; in the second, because there may be many fewer vertices in the set of visible polygonal fragments than in the original model. In the test results presented herein, the non-perspective eye-space model has been used.

5

A Parallel Implementation of the Parallel Polygon Clipper

In the preceding chapter, a visibility algorithm tailored both to high-quality image synthesis and to a parallel implementation was developed. In this chapter an architecture based on this algorithm will be presented.

The first section of this chapter will present the geometrical structure of the algorithm and how it can be used as the basis for a simple and efficient parallel architecture. In the next section, this architecture will be detailed. Finally, the last section will report simulation results on this architecture and compare them to a loose performance bound on more general machines that exploit the same form of parallelism in the parallel polygon clipper algorithm.

5.1. The Geometrical Structure

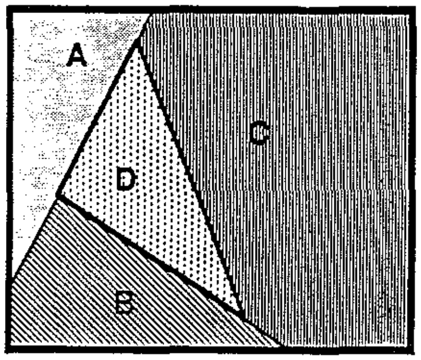
The efficient implementation of an algorithm on highly concurrent hardware requires that the algorithm contain an inherent geometrical structure that can be used as a basis for simple, local communications between cooperating processors. The "divide and conquer" strategy used in developing this algorithm results in such a geometrical structure; the algorithm function takes on the form of a tree. Although this structure proves to be a poor candidate for direct implementation in hardware for reasons given below, it serves as the indirect basis for a simple, efficient and inexpensive architecture.

5.1.1. A Tree Structure

The algorithm presented in the last chapter is based on the division of a list of polygons lying in an area of the screen into completely independent sublists based on a division of the original area into subareas defined by the first polygon in the list. Thus, if the first polygon is a triangle (as in figure 1.a), the initial input list is divided into three sublists consisting of polygonal fragments residing in the three external areas labelled A, B, and C and a set of fragments residing inside the clip polygon silhouette D. If the initial input list was completely sorted, the clip polygon hides the fragments lying in area D and the fragments in this area can be discarded. Otherwise, this list is clipped against the plane of the clip polygon and a fourth list of fragments consisting of fragments in area D that lie in front of the clip polygon, and trailed by the clip polygon itself, is produced.

This division operation can be represented schematically as a node with a single input for the a polygon list and one output for the fragments residing in each subarea defined by the clip polygon. This node is diagrammed in figure 1.b. If the initial polygon list consists of the polygons 1, 2 and 3 in figure 2, polygon 1 is used as the clip polygon and the division operator produces four sublists: empty lists for areas A and B, a list containing the portion of polygon 2 that intersects area C (labelled 2.a) and all of polygon 3, and for area D, a list consisting of the clip polygon itself, since fragment 2.b lay behind the plane of the clip polygon.

The resulting polygon lists are then processed similarly. Since the first two lists are empty, no further recursion is necessary; there is nothing visible in the corresponding areas. Since the fourth list consists of only a single polygon, it is completely visible and is produced for rendering. However, visibility within the third area remains unresolved; the algorithm therefore recurs on the list of polygons lying in this region. Again, the first polygon in the list (2.a) defines four areas E, F, G, and H; the division operator produces an empty list for area E, a list consisting of fragment



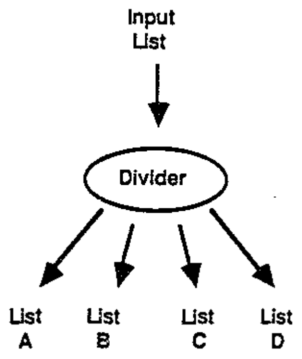


Figure 1a: Area decomposition resulting from triangle
Figure 1b: A tree node corresponding to the triangle decomposition

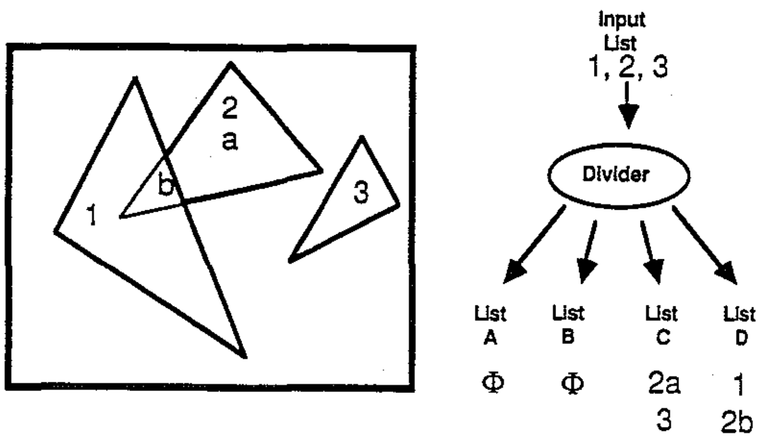


Figure 2a: Three polygons example
Figure 2b: List node for this example

3.a for area F, a list consisting of fragment 3.b for area G, and the clip polygon 2.a for area H (again, a fragment of polygon 3 is clipped away by the plane of the clip polygon). Since no list contains more than one fragment, recursion ceases and a set of visible fragments is determined: polygons 1, 2.a, 3.a and 3.b.

Notice that the input to the second division operation is an output of the first division operation. By connecting these arcs in a single diagram, we produce the tree structure represented in figure 3. Every application of this algorithm to a set of polygons results in a tree structure such as this; the actual form of the tree is determined by the input data.

A Binary Tree Formulation

A simpler structure results when we transform this tree to a binary form. The area-division operation can be decomposed into a sequential set of edge clips; in figure 1, the first such clip separates the initial area S along the first edge of the clip polygon into an "outside" area, corresponding to area A, and an "inside" area, S-A. This "inside" list is then passed to a second clip, which divides the area S-A along the second edge of the clip polygon into the "outside" area B and the "inside" area (S-A)-B. Again, the inside list is passed to a third clip, which divides (S-

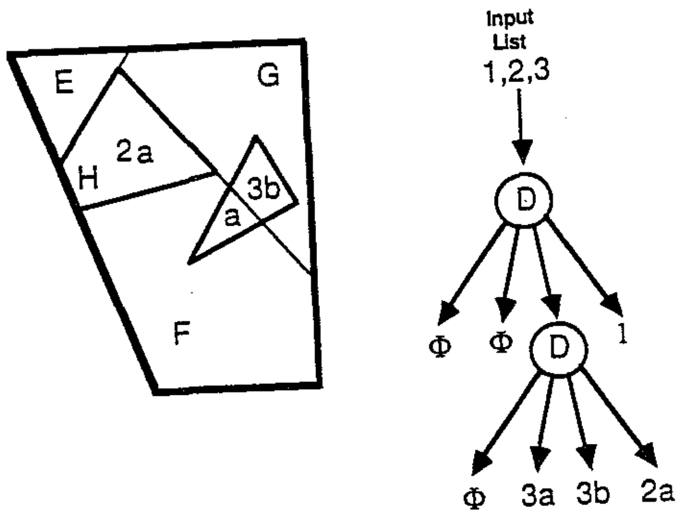


Figure 3: The tree resulting from previous example

A)-B into the "outside" area C and the "inside" ((S-A)-B)-C), now matching the silhouette area D. This final "inside" list is then clipped against the plane of the clip polygon. In this manner, the M-way tree structure of figure 3 is transformed into the binary structure shown in figure 4.

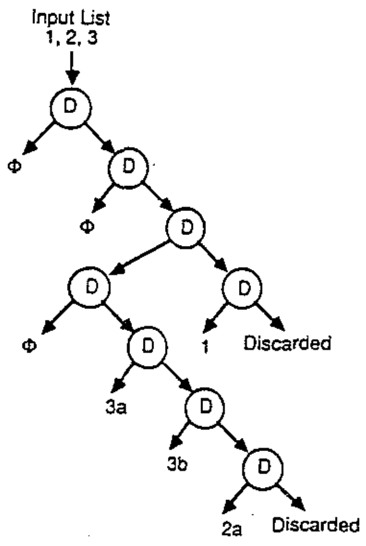


Figure 4: Binary version of tree in Figure 3

5.1.2. A Tree-Structured Processor

The previous section showed that the algorithm contains a very simple geometrical structure: a binary tree. Based on this structure, we might be tempted to build a tree-structured machine, consisting of clippers connected by data paths carrying lists, on which the algorithm would be directly instantiated. Unfortunately, such a structure seems very impractical for several reasons.

Tree Size

As we saw in earlier examples, applying this algorithm to a set of polygons results in a tree structure. However, the particular tree structure produced by the algorithm is highly data-dependent; different sets of input polygons produce different tree structures. If we were to build a tree machine on which the algorithm could execute directly, the hardware would either have to be large and complete enough to handle any set of polygons, or some alternate connection scheme would be necessary to handle overflow cases.

It is obviously impractical to build a tree machine big enough for any real application. In each of the six test images the binary tree structure reached a depth of at least 48. A complete tree machine big enough to cope with these relatively simple images would require at least $2^{49}-1$ processors! Since current (and foreseeable) technology cannot supply us with a billion processors, and since our desired applications are substantially more complicated than the Old Well, it is clear that a sufficiently large machine cannot be built. If a smaller tree structure is to be used, some facility for re-using processors where needed would be necessary. Since lists might be produced at any leaf node of such a limited tree, this would require a communications structure capable of connecting any leaf node (1/2 the number of nodes in the tree) with available overflow processors; furthermore, some temporary storage would be required in the event that no overflow processors were available. This additional communications circuitry defeats the goal of simplicity in the interconnection structure.

Processor Utilization

The implied tree structures of applications of this algorithm are also typified by being quite sparse. Since clips frequently produce empty lists, the entire subtree of processors positioned to receive the empty list will be idle (and wasted). Figure 5 shows the data flow through the first five levels of the tree structure produced by the application of the polygon clipper algorithm to one of the test images (Building Scene 1), showing only the datapaths that actually carry non-empty lists: already, many parts of a complete binary tree structure would never be utilized. Figure 6 shows the actual number of nodes that are active at each level of the tree structure for the six test images: in the early stages, the decomposition of the image plane causes the number of lists to grow; later, as visibility is determined in more and more areas of the image, the number diminishes.

A further utilization problem lies in the repeated dedication of the same amount of hardware for nodes at each level of the area decomposition. The root node will divide the initial list into roughly half; each child processor therefore has only about half as much work to do. This problem extends down the tree: in general, processors at level i will receive a list about $1/2^i$ as long as the

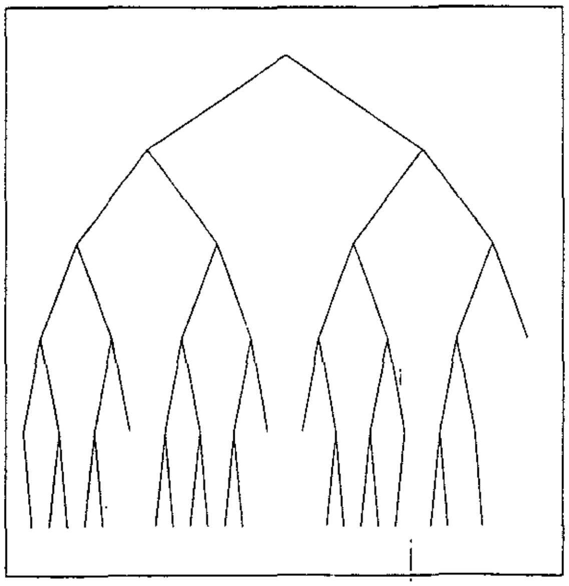


Figure 5: 5 levels of the tree structure in the Building Scene 1

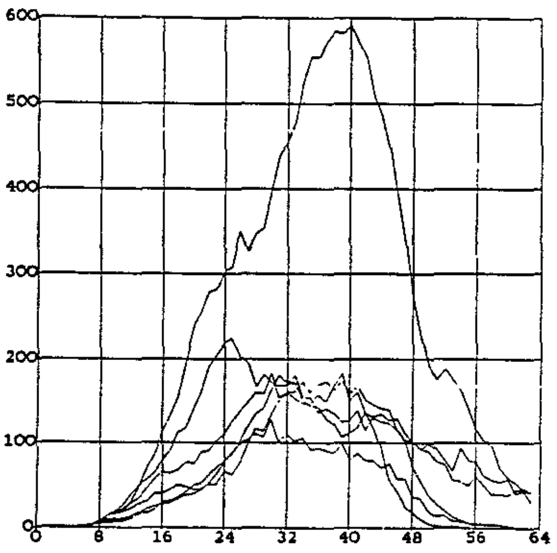


Figure 6: Number of active nodes at each level of the tree structure for the six test images

initial input list. However, this observation leads to another: at level i, we have 2^i processors; therefore, up to a certain point (which is data dependent), there is roughly the same amount of work to be done at each level. Eventually, as the visible surfaces are found, the fragments lying behind them are elimitated, and the workload begins to diminish. Figure 7 shows the actual number of polygon fragments processed at each level of the tree.

5.2. Implementing a Tree on a Linear Pipeline

In the previous section, we observed that as we descend through the levels of the tree hierarchy, the number of processing nodes rose as the amount of work required of each fell. This leads us to consider the replacement of a *level* of the tree with a single processor which performs the function of all the tree nodes at the matching level. Thus the tree of processors is replaced by a pipeline of processors, with each pipeline stage corresponding to a level of the tree (as in figure 8).

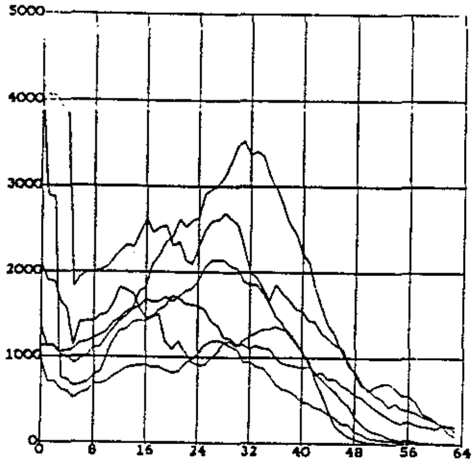


Figure 7: Polygon fragments processed at each level of the tree

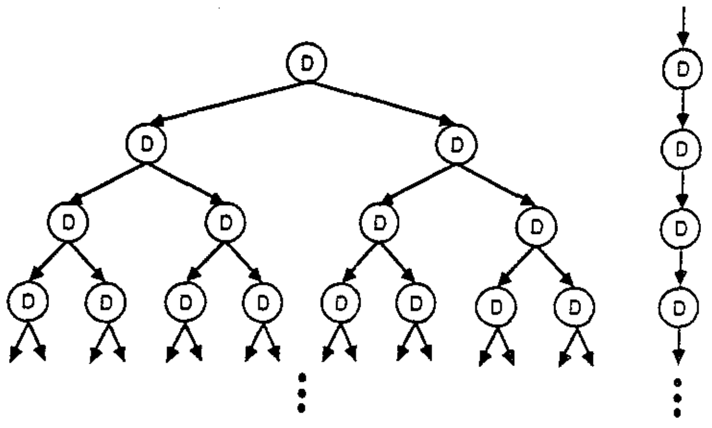


Figure 8: Juxtaposing the tree- and pipeline machines

Operation Identification In a tree-structured machine, the operation performed by a node is implicitly determined by the data received by the node; only a single list is seen by the node, and the first polygon in the list defines the operation for the node (such as the dividing plane on which to divide the remainder). In a pipeline machine, each stage of the pipeline receives every list that passes through the corresponding level of the tree. Furthermore, polygons from these lists arrive intermixed; since the application of a division operation to a list can produce fragments lying on either or both sides of the dividing plane, the output of the node consists of members of both the inside and outside sublists merged in a random pattern.

In order to emulate the operation of each active treenode at the corresponding depth, each pipeline node must store the current state of every tree node that it is emulating. This state information, herein called the *environment* of the tree node, consists of little more than the definition of the dividing plane to be applied at the tree node and a few bits of other state information. As each polygonal fragment arrives at the pipeline node, the node must determine the operation to be applied to that fragment; in effect, it must determine which tree node *would* have received the fragment, access the environment of that node, and apply the correct operation. To do so, each fragment in the system must be tagged to designate its destination tree node. I have used an algorithm which numbers the tree nodes in a top-to-bottom, left-to-right fashion and attaches to each polygon a tag containing the index of the tree node that should recieve it. In this fashion the index of the left successor of node i is $2i$, and the right successor is $2i+1$.

Hashing the Environment Memory As we saw earlier, the tree structure necessary to handle real models may be very deep. For this reason we cannot attach enough memory to each pipeline processor to allow unique storage for the environment of each tree node at the matching level; at level 28, this would require memory for 2^{28}=32 million environments! Clearly, we cannot attach sufficient memory to provide unique storage for each possibly active processor. Fortunately, although there may be literally billions of nodes at a given level of the tree, only a very few will be active. We can therefore provide storage for a vastly smaller number of active processors to be allocated as needed. This can be done by using a hash function, applied to the list tag, to find the appropriate environment.

Overflowing the Environment Memory For any fixed amount of environment memory, there is a certain probability that a case will eventually be found in which the number of tree nodes active at a level will exceed the number for which storage is available in the corresponding pipeline node. This condition can be handled very gracefully if a tagging scheme (such as the one above) is used in which list tags are unique not only at a level, but throughout the tree. If so, if a pipeline processor encounters a list for which it has no available memory it can simply pass the list down to its successor unchanged; the successor can either handle the list or pass it on further.

Load Balancing The splitting operation of a pipeline processor may produce zero, one or two fragments. This fact would cause delays if the pipeline processors were directly connected; successors would have to wait until their predecessors produce fragments for them to operate on, and if a pipeline processor operation produces two fragments, it would have to wait for the first to be processed by its successor before it could hand on its second.

These delays can be avoided by placing a queue between the stages of the pipeline as shown in figure 9: With such a queue, operations producing zero fragments will roughly offset operations producing two fragments and delays are avoided. Only in the eventuality that a queue fills up will delays be incurred, and then most frequently, only by a single stage. If a pipeline processor detects that its output queue has filled it will suspend operations; its predecessor can continue until its own

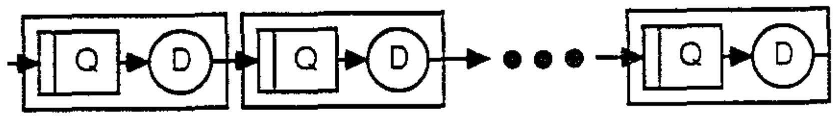


Figure 9: Pipeline processors separated by queues

output queue fills. Only then is the delay propagated farther backward through the pipeline. The effect of the size of this queue is an important parameter of the machine design.

Looping the Pipeline Simulation results have shown that the tree structure encountered in real-world data may be quite deep, ranging from the high twenties from views of the Old Well model to the high forties for views of the UNC Computer Science building. Since the bulk of the work is actually done in the upper levels of the tree, processors actually allocated to these levels will be poorly utilized; furthermore, there seems to be no limit on the number of levels that may actually be encountered. For this reason, it is necessary to re-use the pipeline processors in overflow conditions. This does, however, prove to be easily done by simply connecting the pipeline as a loop, with the outputs of the final stage appended to the input queue of the first, as shown in figure 10.

5.3. Simulation

In chapter 4 a visibility algorithm tailored to parallel execution was developed; in the previous sections of this chapter, a general architecture based on this algorithm was described. In this section, the performance of this architecture is considered. In order to do so, a simple implementation of the architecture has been specified, and a software simulator built. In this section, this simulator and the results of the simulation are described.

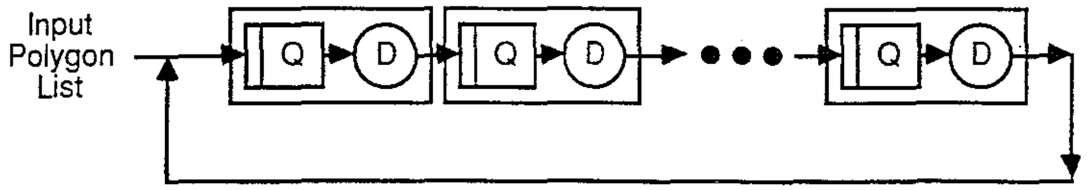


Figure 10: Figure showing looped pipeline

5.3.1. Simulator Model

The description of the simulator model will consist of three parts: a description of the specifics of the architecture of a pipeline node, the specifications of the data formats passing through the system, and a description of the algorithms executing in the pipeline nodes.

5.3.1.1. Node Architecture

The pipeline machine consists of an arbitrary number of pipeline processors connected in a loop. Each processor consists of 5 major components: a controller, a floating point data path with registers, microcode control store, a local memory, and an input queue organized as in figure 11. A single system bus is used to connect the memory elements, the floating point datapath and the outputs. All memories and data paths are assumed to be 32 bits wide.

The input to the pipeline node consists of a 32 bit wide port connecting the previous pipeline stage to the node's input queue. Associated with this port is a single control signal allowing the node to inform the previous stage that the input queue is full. The node has two 32 bit wide output ports: one producing visible fragments for rendering and one passing data to the subsequent pipeline node's input queue. Associated with this port is a control signal allowing the node

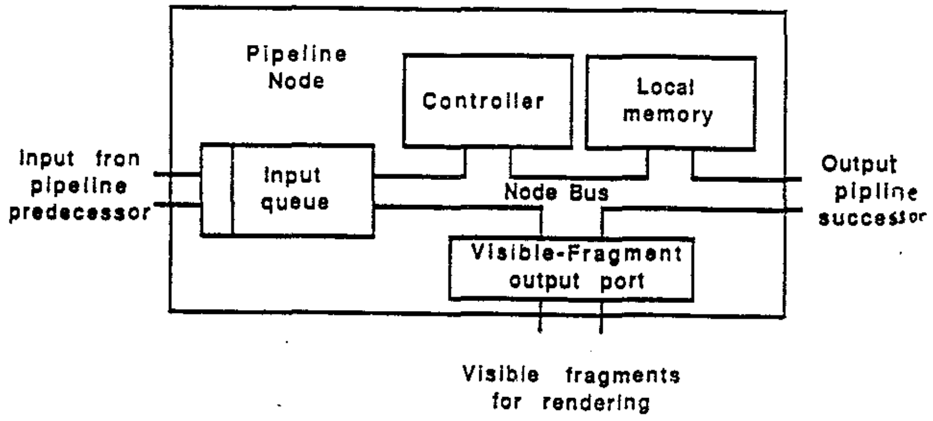

Pipeline Node
Controller
Local memory
Input fron pipeline predecessor
Input queue
Node Bus
Output pipline successor
Visible-Fragment output port
Visible fragments for rendering

Figure 11a: Pipeline node architecture

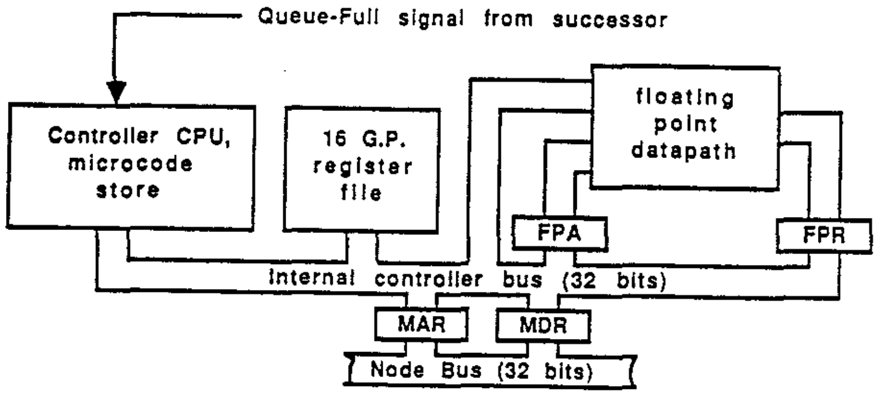

Queue-Full signal from successor
Controller CPU, microcode store
16 G.P. register file
floating point datapath
FPA
FPR
internal controller bus (32 bits)
MAR
MDR
Node Bus (32 bits)

Figure 11b: Pipeline Controller Architecture

controller to sense the state of the subsequent node's input queue.

The node architecture is roughly based on the ADAGE RDS-3000 BPS-32 microcodable processor with the usual integer multiplier data path replaced by a floating point data path. We therefore consider the controller to be based on the AMD-2901 bit-slice family. Under this assumption, we can write programs for the node in the gia2 programming language, written at UNC for the BPS-32, in a manner allowing us to determine an exact count of the machine-level instructions being executed while actually executing the program on a VAX computer.

5.3.1.2. Data Formats

Data flowing through the pipeline are of two formats: polygon blocks representing potentially visible surfaces, and end blocks, marking the end of lists to facilitate the recovery of resourses allocated to the lists. The first word of each of these formats is referred to as the *information word,* and contains a type field identifying the associated block as a polygon fragment that is the first in a list, any other polygon fragment, dividing plane blocks and end blocks.

5.3.1.2.1. Polygonal Fragments

Polygon blocks define fragments of scene surfaces which are members of still-active area lists. At the beginning of this block is a header block which contains information that is global to the polygon: the list identifier, texture identifiers, the surface color, and the number of vertices. Following the header is a list of vertex blocks, each containing both the geometric coordinates of the vertex, values such as the normal vector components, texture indices assigned to the vertex and, if the edge emanating from the vertex has not yet served as a dividing plane, the equation of the dividing plane defined by that edge.

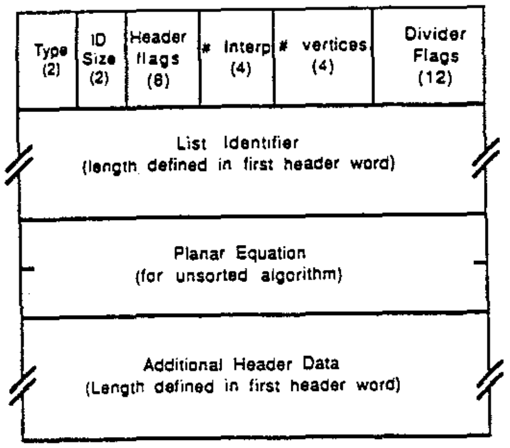
The header block consists of an information word containing four fields describing the remainder of the block, followed by the polygon's list identifier, and finally, some further number of words of header information. The information word contains five fields: a two-bit type field, identifying this to be one of the two polygon fragment types, a two-bit field containing the count of words in the list identifier, an 8-bit field containing flags defining the remaining contents of the header block, a 4-bit field containing the vertex count, a 4-bit field containing the number of interpolants associated with each vertex, and a 12-bit field containing a flag for each vertex that indicates the presence of a dividing plane in the associated vertex block. The header and vertex blocks are diagrammed in figure 12.

5.3.1.2.2. End Blocks

The final data format is the end block, marking the end of a list. The end block consists of a word containing a type field and the number of words in the list identifier, followed by the list identifier. The end block is diagrammed in figure 13.b.

5.3.1.3. Internal Data Structures

Each node in the pipeline must maintain the environment of many logical tree processors. In order to do so, two internal data structures are maintained: an *environment table* , containing the state of each list passing through the node, and a *polygon table*, used to store the first polygon in


Type (2)
ID Size (2)
Header flags (8)
# Interp (4)
# vertices (4)
Divider Flags (12)
List Identifier (length defined in first header word)
Planar Equation (for unsorted algorithm)
Additional Header Data (Length defined in first header word)

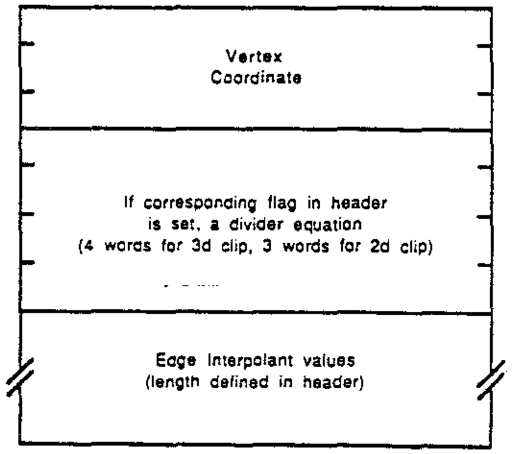

Vertex Coordinate
If corresponding flag in header is set, a divider equation (4 words for 3d clip, 3 words for 2d clip)
Edge Interpolant values (length defined in header)

Figure 12a: Header block diagram
Figure 12b: Vertex block diagram

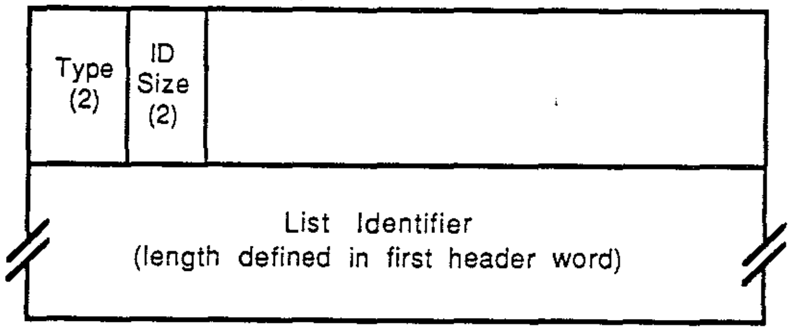


Figure 13: End Block diagram

an unsorted list until its visibility is verified.

The Environment Table

The environment table contains the current state of each polygon list that is currently active in a node. This information consists of eight words: a flag word, a two-word field for the list identifier, a four word field for the list dividing plane and, if the input polygon list is not assumed to be sorted, a pointer to the first polygon that was encountered in the list.

Because it is impractical to provide enough memory for the storage of the environment of all the lists that *might* be encountered by a node, a hashing scheme is used to map the many list identifiers that may be encountered in a node into a much smaller list of environment blocks. In the simulator, the bottom bits of the low-order word of the list identifier are used as a starting address and a sequential search is made from that point. If no environment block is free when the first polygon in a list is encountered, the polygon is simply passed on to the node's pipeline successor. When any other input is encountered and a corresponding environment node cannot be found, the list is assumed to lie in a later pipeline node and, again, the block is passed on.

The Polygon Table

If the input list is not assumed to be sorted, it is necessary to verify that the first polygon in a list is indeed completely visible. Thus the remaining polygons in the list must be clipped against the plane of the first. If the end of the list is encountered and no fragments lying in front of the first polygon were encountered, then the first polygon is completely visible and can be output. Otherwise, the first polygon must be tacked onto the end of the list of fragments that lay in front of it.

In either event, the polygon must be saved until the end of the list it began is encountered. Since storage is required only when the list is being clipped against the plane of the first polygon, rather than against planes defined by an edge of the first polygon and the viewpoint, it is very wasteful of memory to allocate polygon storage along with each environment block in the node memory. A separate list is therefore kept. When a polygon storage block is required, a linked list of available polygon blocks is consulted. If one is available, it is removed from the list and a field in the environment block for the list is set to point to it. If none is available, the first polygon is passed on to be handled in a subsequent pipeline processor.

5.3.1.4. Node Algorithms

In order to enhance their understandability, this section will present pseudo-code versions of the algorithms actually executing in the pipeline nodes.

Node Loop Each pipeline node operates in a simple loop: it reads the first word of a block from its input queue and examines it to determine the type of the block. It then passes control to one of four routines matching each type; these routines perform the function appropriate for the received block and produce results, if any, to the input queue of the subsequent pipeline node and to the visible fragment output.

```
NodeLoop:
    while (true) {
        PopInputQueue(1, &InfoWord);
        switch(InfoWord.type) {
            case FIRSTPOLY:           FirstPoly();      break;
            case OTHERPOLY:           OtherPoly();      break;
            case DIVIDERBLOCK:        DividerBlock();   break;
            case ENDBLOCK:            EndBlock();       break;
        }
    }
```

Figure 14: Pipeline Node Loop Structure

In this section, the algorithms actually executing in a pipeline node are presented. The three subsections of this section detail each of the subroutines: the processing of the first polygon block in a list, the processing of other polygon blocks, and the processing of end blocks. A simple pseudo-code version of each routine will be given. Appendix (whatever) contains the actual code used in the simulator.

Processing First Polygons

The handling of the first polygon in a list depends on whether the initial polygon list was completely sorted, as by a BSP-tree preprocessor, or partially sorted as by a bucket sort. In the first case, the polygon is known to be visible and can be output. In the second case, the polygon must be saved until the visibility of the polygon has been verified by clipping the final inside list against its silhouette. First the simpler, completely-sorted version will be presented.

```
FirstPoly:
    /*
     * If there is one or zero dividing planes, then we will output
     * the polygon to the visible port. Otherwise, the polygon and
     * attendent dividing planes, are passed on to the pipeline
     * successor.
     */
    if (InfoWord.NumDividers == 1)
        OutputAddress = VISPORT;
    else
        OutputAddress = SUCCESSOR;

    /*
     * Read the List Identifier from the input queue
     */
    ListID = GetListID();

    /*
     * We will need an environment block.
     */

    Environment = NewEnvironment(ListID);
    if (Environment == NULL) {

        No environment was available. Copy the block to the
        successor port.

        return;

    }

    /*
     * The first fragment in either list will be the first in that list
     */
    Environment->FirstOuter = TRUE;
    Environment->FirstInner = TRUE;

    /*
     * If another dividing plane awaits in the block, the inside list must
     * be clipped against another edge by the successor. Otherwise,
     * the input list can be discarded.
     */
    if (InfoWord.NumDividers > 1)
        Environment->SendInner = TRUE;
```

```
else
    Environment->SendInner = FALSE;

Copy header info to OutputAddress

/*
 * Now we go into a vertex loop.  Read the vertices and write
 * them to the output address.  When dividing planes are found,
 * scarf the first one for the current environment and pass the
 * remainder on to the subsequent pipeline node as part of the
 * dividing plane block.
 */
first = 1;
for (each vertex block) {

    Copy vertex to OutputAddress

    if (vertex has a dividing plane)
          if (first) {
              first = 0;
              Store dividing plane in environment block
          } else
              Copy dividing plane to OutputAddress
}

return;
```

Figure 15b: Algorithm for the first polygon in a list, fully sorted.

The version of this algorithm for roughly sorted input lists differs in that the polygon block is passed along the pipe with each node extracting a dividing plane until no further dividing planes are found. When a node receives a first polygon that contains no further dividing planes, it uses the planar equation of the polygon itself (contained in the header) as a dividing plane, and stores the polygon itself in local memory. An additional flag in the environment (FragmentsInFront) is used to recall whether any fragments have been encountered which lay in front of the clip polygon.

```
FirstPoly:
    /*
     * Read the List Identifier from the input queue
     */
    ListID = GetListID();
```

```
/*
 * If there is an edge dividing plane, we will be passing the first
 * polygon on, and we need only an environment block.  Otherwise, we
 * will need a polygon block also.
 */
Environment = NewEnvironment(ListID);
if (InfoWord.NumDividers > 0)
    OutputAddress = GetPolygonBlock();

if (Environment == NULL || OutputAddress == NULL) {

    No storage was available.  Copy the block to the
    successor port.

    return;

}

/*
 * The first fragment in either list will be the first in that list
 */
Environment->FirstOuter = TRUE;
Environment->FirstInner = TRUE;

/*
 * If we are clipping against the plane of the polygon, we
 * do not send on the inner list, since it corresponds to the
 * fragments lying behind the clip polygon.  If we are clipping
 * against the plane of a polygon, set CapClip.  Copy
 * the polygon to the polygon block.  Otherwise, we DO send
 * on the inner list, and we copy the polygon on to the
 * successor.
 */
if (InfoWord.NumDividers > 0) {
    Environment->SendInner = TRUE;
    Environment->CapClip = FALSE;
    Copy polygon planar equation to successor
} else {
    Environment->SendInner = FALSE;
    Environment->CapClip = TRUE;
    Copy polygon planar equation to polygon block
}

/*
 * Now we go into a vertex loop.  Read the vertices and write
 * them to the output address.  When dividing planes are found,
 * scarf the first one for the current environment and pass the
 * remainder first one for the current environment and pass the
 * remainder on to the subsequent pipeline node as part of the
 * dividing plane block.
 */
first = 1;
for (each vertex block) {
        Copy vertex to OutputAddress

        if (vertex has a dividing plane)
            if (first) {
                first = 0;
                Store dividing plane in environment block
            } else
                Copy dividing plane to OutputAddress
    }

    return;
```

Figure 16: Algorithm for the first polygon in a list, roughly sorted.

Processing Remaining Polygons

Each polygon following the first in a list must be clipped against the dividing plane defined by the first polygon and stored in the node's environment memory. The algorithm maintains two vertex lists for vertices falling on either of the two sides of the dividing plane plane. As the vertices are read one by one from the input queue, the signed distance of the point to the dividing plane is evaluated and stored, along with the point, in a local vertex list. If all the vertices lay on the same side of the dividing plane, the polygon is tagged appropriately and transferred to the successor. Otherwise, the polygon must be clipped.

The clipping algorithm is a Sutherland/Hodgman clip [SuH,74] modified to support the association of dividing planes with polygon edges. Two vertex lists are allocated for the vertices lying on the either side of the dividing plane. A pass is made through the vertex list, first assigning the vertex itself (and the dividing plane stored with the vertex, if any) to the appropriate list, and then considering the edge emanating from the vertex. If the edge crosses the dividing plane, the intersection point is found and assigned to both lists. Note that the resulting edge on the opposite side of the dividing plane from the original point *continues* the edge that began at that vertex; thus, if the original vertex had a dividing plane it is also assigned to the intersection vertex in the opposite list. The intersection point in the original list begins a new edge that lies in the plane of the dividing plane. If the polygon is being clipped against a dividing plane that is defined by the viewpoint and a polygon edge, this dividing plane already has separated the viewing frustum and need not do so again; in this case, the intersection point does not receive an edge. If, however, the polygon is being clipped against the plane of a polygon, it will be necessary to divide the frustum by the dividing plane defined by the derived edge and the viewpoint; thus a new dividing plane must be computed from the derived edge and associated with the intersection vertex.

A pseudo-code implementation of this algorithm is given in figure 17.

```
OtherPolygon:
    /*
     * Read the List Identifier from the input queue
     */
    ListID = GetListID();

    /*
     * Look for the environment associated with the list
     * containing the polygon.
     */
    Environment = EnvironmentSearch(ListID);
    if (Environment == NULL) {

        /*
         * If not found, this list was passed on through this node
         */

        Copy the input to the output

        return;
    }

    Move the dividing plane equation from the environment block
    to registers for fast evaluation of ax+by+cz+d

    Read the polygon header into local memory.  When the
    clipping operation completes we may have to send the
    header information along with both fragments

    /*
     * Now we start the clipping algorithm.  First the vertex list is
     * read into local memory.  As each vertex is encountered the inner
     * product of the vertex with the dividing plane plane is evaluated and
     * the side determination: + side, - side, or clipped, is made.
     */
```

```
LastSide = 0;
Clip = FALSE;

for (Number of Vertices) {

    /*
     * Evaluate A*x
     */
    MPX = input word
    MPY = A
    MULTIPLY
    store input word locally

    /*
     * Evaluate A*x + B*y
     */
    MPX = input word
    MPY = B
    MULTIPLYACCUMULATE
    store input word locally

    /*
     * Evaluate A*x + B*y + C*z
     */
    MPX = input word
    MPY = C
    MULTIPLYACCUMULATE
    store input word locally

    /*
     * Add in D
     */
    MPX = D
    MPY = MPResult
    ADD
    store MPResult locally

    if (MPResult < 0)
          if (LastSide > 0) Clip = TRUE;
          else LastSide = -1;
    else
          if (LastSide > 0) Clip = TRUE;
          else LastSide = 1;

    Copy remainder of vertex interpolants to
    local memory

    If vertex carries a dividing plane, copy it to
    local memory
}

if (LastSide == 0) {
    /*
```

```
         * Then polygon lay in divider plane.  Throw it away.
         */
        return;
    }

    if (Clip == FALSE) {

        /*
         * Then the polygon lay on one side or the other.  Send
         * it on.
         */

        if (LastSide < 0) {
              /*
               * Then output INSIDE if necessary (not necessary
               * if this is a clip against a polygon plane or, in
               * the sorted algorithm, this is the final INSIDE list).
               */
              if (Environment->SendInner == TRUE) {
                 Environment->InnerFirst = FALSE;
                 write InfoWord to successor
                 write INNER(ListID) to successor
                 write polygon header to successor
                 write vertex list to successor
              }
        } else {
              /*
               * Always send OUTER list.  If Environment->OuterFirst is
               * set, the result will be the first polygon in that list
               * and must be flagged as such.
               */
              if (Environment->OuterFirst == TRUE) {
                 Environment->OuterFirst = FALSE;
                 modify InfoWord to indicate FirstPoly
              }
              write InfoWord to successor
              write OUTER(ListID) to successor
              write polygon header to successor
              write vertex list to successor
        }

        return

    }

    /*
     * Otherwise, a clip is necessary.
     */
    for (each vertex) {

        if (vertex.distance is > 0) {

              Add vertex to Pos list
```

```
          if (vertex has a divider)
             Add divider to Pos list

          if (next vertex inner product is < 0) {

             Compute intersection point and add to Pos and Neg lists

             if (current vertex has a divider)
                   Add divider to Neg list

             /*
              * If we are clipping against the plane of a polygon,
              * this is the derived edge starting point.
              */
             if (Environment->CapClip == TRUE)
                   set StartPoint to point to Pos list vertex
          }

      } else if (vertex.distance < 0) {

          Add vertex to Neg list
          if (vertex has a divider)
             Add divider to Neg list

          if (next vertex inner product is < 0) {

             Compute intersection point and add to Pos and Neg lists

             if (current vertex has a divider)
                   Add divider to Pos list
             /*
              * If we are clipping against the plane of a polygon,
              * this is the derived edge ending point.
              */
             if (Environment->CapClip == TRUE)
                   set EndPoint to point to Neg list vertex
          }

      } else {  /* Vertex lies in divider plane */

          Copy vertex to both Pos and Neg lists

          /*
           * If the vertex has a divider, add it to the list for
           * the polygon edge departing the plane of the divider.
           * Thus if the edge leaves to the POSITIVE side, the divider
           * is added to the Pos list; if it leaves to the NEGATIVE
           * side, then the NEG list. If this is a polygon-plane clip,
           * then the vertex in the OTHER list will be a starting/ending
           * point for the derived edge.
           */

          if (vertex has a divider) {
                if (nextvertex.distance > 0) {

                    Copy divider to Pos list

                    if (Environment->CapClip == TRUE)
                        Set StartPoint to point to Neg list vertex

                } else if (nextvertex.distance < 0) {

                    Copy divider to Neg list

                    if (Environment->CapClip == TRUE)
                        Set StartPoint to point to Pos list vertex

                } else {
                    edge emanating from the vertex lies in the
                    divider plane.   Ignore it.
                }
            }

        }
}
```

If we clipped against a polygon plane, we now compute the dividing plane defined by the derived edge from the vertex pointed to by StartPoint and ending at the vertex pointed to by endpoint, and the viewpoint. Assign the resulting dividing plane equation to the vertex pointed to by StartPoint and, with signs inverted, to the vertex pointed to by EndPoint.

```
/*
 * If the appropriate flag is set, asend the Inner fragment
 */
if (Environment->SendInner == TRUE) {
    Environment->InnerFirst = FALSE;
    write InfoWord to successor
    write INNER(ListID) to successor
    write polygon header to successor
    write Neg list to successor
}

/*
 * Always send the Outer fragment
 */
if (Environment->OuterFirst == TRUE) {
    Environment->OuterFirst = FALSE;
    modify InfoWord to indicate FirstPoly
}
write InfoWord to successor
write Outer(ListID) to successor
write polygon header to successor
write Pos list to successor
```

```
        return;
    }
```

Figure 17: Algorithm for other-than-first polygon blocks

Processing End Blocks

Finally, an end block terminate a list. First, the environment table is searched to find the corresponding environment. If either or both of the inside and outside lists were non-empty, the lists are terminated by appropriate end blocks. Finally, the environment table entry is freed. This algorithm is detailed in figure 18.

```
EndBlock:
    /*
     * Read the List Identifier into local memory
     */
    ListID = GetListID();

    /*
     * Look for the environment associated with the list
     * containing the polygon.
     */
    Environment = EnvironmentSearch(ListID);
    if (Environment == NULL) {

        /*
         * If not found, this list was passed on through this node
         */

        Copy block on through to successor

        return;
    }

    /*
     * Otherwise, we found the environment. If the environment
     * flag indicates that a INSIDE polygon fragment has been
     * sent, then we need to terminate that list.
     */
    if (Environment->FirstInner == FALSE) {
        Send an endblock with INNER(ListID);
    }
```

```
/*
 * Now for the OUTER list. If this is a CAPCLIP, then if any
 * fragments were sent to the OUTER list, we have to append the
 * clip polygon to the OUTER list and terminate it with an end
 * block. Otherwise, if these were no OUTER fragments, we have
 * found a visible polygon and can output it.
 */
if (Environment->CapClip == TRUE) {

    if (Environment->FirstOuter == TRUE) {

        Send the clip polygon pointed to by the environment block
        to the successor tagged appropriately.

        Send an end block to that list also.

    } else {

        /*
         * OUTER list empty means the clip polygon was visible.
         */

        Output the clip polygon to the visible port
    }

    Free up the polygon block.

} else {

    /*
     * Then this is a simple outer list. If any fragments went that
     * way we must send on an end block.
     */
    if (Environment->FirstOuter == TRUE) {

        Send an end block to for the outer list.

    }
}

free up the environment

return;
```

Figure 18: Algorithm for end blocks

5.3.1.5. Simulation Management

Managing the simulation of a concurrent system on a general-purpose computer requires the use of a manager, responsible for sharing the resources of the comptuer among the ostensibly-concurrent components, and for aquiring statistics as the simulation proceeds.

In this simulation, each concurrent pipeline node is represented internally as a data structure retaining the state of the node. This state consists of the current contents of the node's local memory, its internal registers, and its input queue. We represent this as a C data structure:

```
typedef int WORD;

struct {
   /*
   16 General purpose registers
   */
   WORD R0, R1, R2, R3, R4, R5, R6, R7, R8,
       R9, R10, R11, R12, R13, R14, R15;
   /*
   Memory address, data registers
   */
   WORD MAR, MDR;
   /*
   Floating point datapath registers: an input and
   a result. The second input is coincident with the MDR.
   */
   WORD FPA, FPR;
   /*
   Pointers to the input and output queues
   */
   QUEUE *InQ, *OutQ;
   /*
   Local memory (addressed as an array)
   */
   WORD *Memory;
} NODE;
```

Figure 19: Node data structure

Timing Statistics

Execution statistics are collected during the simulation runs as the simulation proceeds. The simulation monitor executes a simple loop: a simulated node is selected for execution, that node then executes one pass through *NodeLoop*, reading a block from its input queue and executing the appropriate node algorithm to completion, and then returns control to the monitor for the selection of the next node to be fired. Since the time used by the simulated node differs depending on the type and size of the input block it processes, a separate time counter is maintained for each simulated node, representing the current time of that node.

While a simulated node has control and is processing a block of input, the elapsing time is accrued in the node's current time register. Time elapses in a node in three ways: the time required to execute instructions, the time that passes while the node waits for input to arrive at its input queue, and time which passes while the node waits for room to be made in the input queue of the node's successor for a word of output.

The instruction-execution time is accounted for by counting the actual machine-level instructions executed and weighting them according to their type. Macros have been placed in the node algorithm code to count the operations as they are executed; these macros increment both the current time register and separate counters for each type of instructions executed. Four instruction classes are defined: controller operations, consisting of register-to-register operations taking place within the node controller, bus operations, in which data is moved along the node's internal bus, floating point operations excluding division, and division operations. The weights applied to each are a parameter of the simulation.

Time that elapses waiting for input is accounted for by tagging each word placed in an inter-processor queue with the time at which it was placed there. When a node attempts to read a word from its input queue, the current time of the node is compared to the arrival time of the word. If the comparison indicates that the request comes *before* the input word was made available by the node's predecessor, the node must "wait" for the data to become available; this delay is accounted for by updating current time register to the arrival time of the datum.

The time elapsing waiting for room to be made in the successor's input queue is more difficult to account for. The processing of a block by a node may fill the successor's input queue, forcing the node to wait. However, no provisions have been made to transfer control out of the middle of the processing of a block; thus, control cannot be transferred to the successor to make room. Instead, a very conservative assumption has been made, and is described below.

CPU Allocation

Each time the monitor loops, the node with the earliest starting time is selected for execution. The starting time is defined to be the maximum of the current time of the simulated node (indicating the time it last completed processing an input block), the time at which the first word in its input queue was placed there by its predecessor, (indicating the arrival of a block to be processed), and the time at which sufficient room is made in the input queue of the node's successor for any data which the node may need to place there. When no data remains in any of the simulated node's input queues, the simulation completes, and the maximum current time of the set of simulated nodes indicates the total time required by the pipeline processor on the input data.

Accuracy of Timing

The allocation of the CPU for an entire block operation time, selected for simplicity and speed, is admittedly coarse. Although the use of a timing queue to synchronize the functions of different nodes correctly models the waiting that can occur when an empty input queue occurs, it incorrectly models the wait state that *should* occur when the output queue fills during a block operation (eg. while splitting a polygonal fragment in OtherPoly, the output queue may fill up, causing the node to pend until the sucessor node makes room by extracting data from the queue). Since there is no provision for a transfer of control from the middle of the block operation to the sucessor, the queue will not empty.

In order to cope with this problem, the manager must verify that space is available for output *before* the processing of a block can begin. Conceivably, operation could commence without available space, possibly performing the bulk of the operation before needing to do output; during this time the successor might extract enough data so that operation could continue without delay. For this reason, this assumption makes the simulation results conservative.

5.3.2. Variables in the Simulator Structure

The performance of this machine may vary as three design parameters are changed: the length of the pipeline, the size of the interprocessor input queues, and the amount of storage allocated to each node to retain the environments of simulated tree nodes. The simulator incorporates the overflow schemes described earlier: the pipeline is looped; processors pend waiting for room in their output queues, and lists are passed on when no environment memory is available. In order to assess the effects of these parameters on the expected performance of the machine, each scene will be simulated several times varying these parameters.

5.3.3. Simulation Results

The simulator described above has been run on four input databases producing the six images shown in chapter one. In this section these results are described.

Input Databases The four input databases include a model of the new UNC Computer Science building, two aircraft databases showing one and two fighter aircraft flying over a simple fixed-subdivision stochastic terrain, and a model of a Russian battlecruiser (the *Kashin*). Each database consists of triangles and quadrilaterals. The building and aircraft databases contain both surface normal and texturing surface interpolants, while the ship contains only surface normals.

The databases exhibit varying geometrical characteristics. The building model consists of a large, roughly detailed architectural database placed in a cylindrical volume on which the surroundings of the building are to be seen through windows and doors. Only a relatively small portion of this database lies within the frustum of a given internal view of the model; the remainder is removed by the clipping process (or eliminated from the database beforehand; the database is divided into components, such as the different floors, which may be combined as needed for specific views). By changing the viewpoint and viewing direction we can produce scenes of differing complexity; a scene from the lobby balcony viewing across the lobby (as in the building scene 1 scene) will show a relatively small depth complexity, since only a small portion of the internal walls of the building lie between the viewpoint and the surrounding cylinder, while a view looking back from the far side of the lobby (as in the building scene 2 scene) will show a higher depth complexity since the viewing axis passes through the entire length of the building. One room in the building contains furniture models (shown in the building scene 3 scene).

The ship model contains a detailed superstructure but no internal detail at all, again placed inside a surrounding cylinder. The single image produced from this model is seen from a viewpoint over the left quarter of the ship and includes most of the ship. Back-facing polygons have been removed from this scene.

The aircraft scenes are made up of an environment, consisting of a terrain model and a cylindrical surrounding, with one and two fighter aircraft models. In these scenes the ground consists of a stochastically subdivided plane, and therefore is comprised of a set of abutting triangles. The aircraft models themselves consist of 417 polygons each. Thus these scenes are typified by regions of relatively high complexity (eg. the areas of the image plane covered by the aircraft), areas of medium complexity (the stochastic terrain) and low complexity (the cylinder on which the sky is painted).

Database Preparation A preprocessing step is first run on each database to prepare it for the visibility calculation. Each polygon in the database is first transformed to the eye coordinate system. The planar equation of the polygon and the separating planes defined by the polygon edges and the viewpoint are then determined. A Z bucket sort then roughly sorts the polygon set: each polygon is placed in one of 10,000 bins based on the average depth of the vertices of the polygon; polygons behind the viewpoint are placed in the first bucket. These bins are then traversed in a front-to-back order to produce the approximately-sorted initial input list. This list is preceded by an artificially-constructed "clip" polygon defining separating planes which define the viewing frustum; based on this clip polygon, the first stages of the pipeline clipper clip the polygon set to a five-sided viewing frustum.

Sample Image Complexity The following table shows the complexity of the six sample images. For each image, the number of polygons in the input database is first given, followed by the number of polygons remaining after the input data has been clipped to the viewing frustum and the number of input polygons that are visible (at least in part). The final column shows the average number of polygonal fragments hidden by each visible fragment; this number indicates the depth complexity of the image.

image	input polygon count	polygons remaining after clip	partially visible polygons	average hidden list size
building scene 1	4,193	659	335	2.68
building scene 2	4,173	1,831	301	6.04
building scene 3	2,099	1,124	268	4.52
aircraft scene 1	946	515	232	3.26
aircraft scene 2	1,352	921	391	4.27
ship scene	1,302	1,089	763	3.10

Polygon Proliferation in the Test Images The next list indicates the performance of the algorithm on the six test images. For each image, the number of polygons lying within the viewing frustum and the number that are at least partially visible are first repeated, followed by the number of visible fragments produced and the average number of fragments produced for each partially visible input polygon.

image	polygons after clip	partially visible polygons	fragments produced	fragments per visible polygon
building scene 1	659	335	2,388	7.12
building scene 2	1,831	301	1,795	5.96
building scene 3	1,124	268	1,229	4.21
aircraft scene 1	515	232	1,007	4.34
aircraft scene 2	921	391	1,641	4.20
ship scene	1,089	763	5,802	7.60

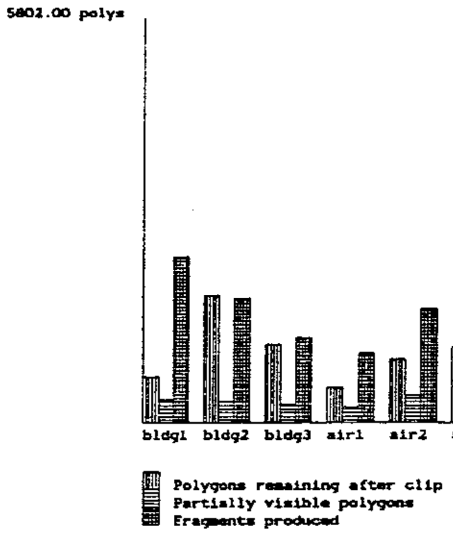


Chart 1: Polygon counts for test images

16-Processor Pipeline Performance The following table shows the expected performance of a 16-node pipeline visibility processor on each of the 6 test images. Two statistics are reported: the total execution time required, and the average idle time of the pipeline processors. These results assume that controller and bus operations each require 50 nanoseconds, floating point additions, subtractions, multiplications and comparisons require 100 nanoseconds, and floating point divisions require 250 nanoseconds. Each node in the pipeline contains sufficient memory to prevent any memory overflow problems (512 environments and 128 polygons in each node). The initial node in the pipeline (which must store the entire original database) has an ample input queue of 256,000 words; subsequent pipeline nodes have input queues of 1,024 words.

image	execution time(sec)	average idle proportion
building scene 1	0.289	44.4%
building scene 2	0.336	29.8%
building scene 3	0.178	33.7%
aircraft scene 1	0.108	31.6%
aircraft scene 2	0.207	30.6%
ship scene	0.434	33.3%

Effects of Longer Pipelines

By lengthening the pipeline, more processors can be brought to bear on the computation. The next table indicates the execution time and idle time proportions for pipelines of 8, 16, 32 and 48 processors on the test images. Again, each pipeline node contains sufficient memory to avoid overflow, and pipeline nodes are separated by queues of 1024 words.

	8		16		32		48	
image	time	idle%	time	idle%	time	idle%	time	idle%
building scene 1	0.447	28.2	0.289	44.4	0.177	55.4	0.152	65.1
building scene 2	0.601	20.0	0.336	29.7	0.210	42.5	0.148	46.1
building scene 3	0.310	23.6	0.179	33.7	0.127	46.7	0.085	48.4
aircraft scene 1	0.184	20.2	0.108	31.6	0.067	44.6	0.067	60.3
aircraft scene 2	0.354	19.0	0.207	30.6	0.143	48.4	0.111	53.9
ship scene	0.733	21.1	0.434	33.3	0.320	54.5	[1]	[1]

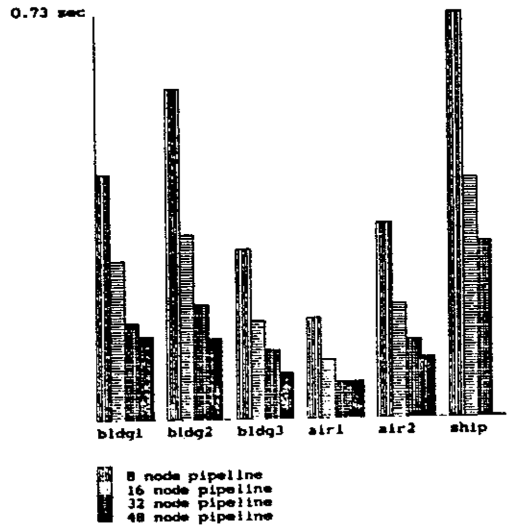


Chart 2: Effects of pipeline length on execution time

The correspondence of pipeline processors to levels of the tree limits the pipeline length to the depth of the tree which, in the test cases, is in the area of 64; any pipeline processors past this point will remain idle. Furthermore, the bulk of the processing is actually done in the first 30 to 40 levels (as seen in figure 20 below, which shows the number of polygons processed at each depth for each of the six test images). Thus we find that, although the architecture shares the computation efficiently over 30 to 40 processors, the addition of further pipeline processors past that point meets with limited success.

Varying Internal Memory Size Ideally, each pipeline node emulates the function of every active tree-processor node at the same depth. If there are more active nodes at some depth than there is memory to store the necessary state information in the corresponding pipeline node, some of the work must be passed on to subsequent pipeline nodes that have available memory.

The default memory size used in the simulator provides 256 environment table slots (at 8 32-bit words each) and 64 polygon nodes (at 64 words each), requiring 6,144 words; miscellaneous

[1] The ship model cannot be run on pipes of length greater than 32 on our VAX 11/780 due to the excessive memory

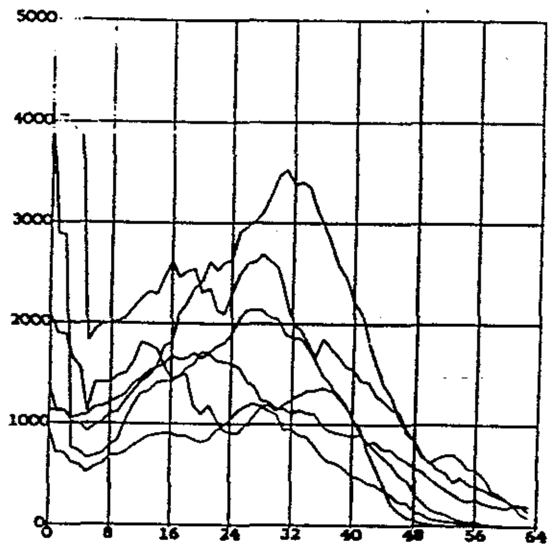


Figure 20: Polygons processed at each depth

storage increases the memory needs to 8,951 words. Of our test scenes, only the ship scene encountered more than a trivial amount of memory overflow at this memory size. In the following table shows the effects of varying the node memory size on the time required to determine visibility on this scene.

storage: #environments/ #polygons	total size of memory in bytes	number of lists passed through	pipeline length		
			8	16	32
256/64	42,364	94,461	6.419 sec.	3.787 sec.	2.272 sec.
512/128	63,292	17,711	2.616 sec.	1.605 sec.	0.982 sec.
1024/128	79,676	17,711	3.898 sec.	2.425 sec.	1.444 sec.
1024/512	204,092	0	0.733 sec.	0.434 sec.	0.320 sec.
2048/512	236,860	0	0.698 sec.	0.412 sec.	0.298 sec.

Estimating the Per-Polygon Time The simulation results provide the information necessary to estimate the average amount of time required by a pipeline node to process a polygon. By summing the total execution time of all the nodes in a pipeline machine and subtracting the total idle time of the nodes in the machine, we determine the amount of time actually spent processing polygons; by dividing the result by the total number of polygons processed by all the nodes, we

requirements.

derive an average per-polygon time for that simulation run. The following table shows the results of this analysis (in microseconds) for each of several pipeline lengths for the the six test images.

	microseconds per polygon pipeline length			
image	8	16	32	48
building scene 1	31.41	31.60	31.86	31.60
building scene 2	31.12	31.49	31.57	31.48
building scene 3	28.47	28.50	28.57	28.03
aircraft scene 1	27.27	27.31	27.32	26.90
aircraft scene 2	29.06	29.09	29.03	28.51
ship scene	28.55	28.54	28.42	[2]

These figures vary for many reasons. The average polygon time rises with the proportion of polygons that must be clipped against a separator plane. The cost of accessing an environment depends on the likelihood of a hash-table collision. Even so, the resulting figures for the average per-polygon time fall very close together, and the average of 29.38 microseconds per polygon will serve for later analyses.

5.3.4. Discussion of Results

The above results indicate the performance of a very simple implementation of the pipeline architecture which is feasible using currently-available hardware. Although these results fail to show real-time performance on these scenes, they do indicate a promising basis for further work. The architecture and implementation outlined above forego many opportunities for further parallelism in exchange for simplicity. In this section two possibilities are considered: tailoring the pipeline node architecture for the process of polygon clipping, and alternatives to the linear pipeline architecture.

[2] Again, a simulation of a pipeline of length greater than 32 cannot be run on the ship scene due to insufficient memory in the simulating computer, a VAX 11/785 with a 30 megabyte process size.

5.3.4.1. Node Level Architecture

At the node level, this simulation has assumed a very simple structure consisting of a general-purpose microcodable processor and a single floating-point datapath. This generality results in a node which spends a great majority of its time doing overhead operations rather than the actual arithmetic necessary for polygon clipping. The following table shows the number of node instructions executed on a 16-node pipeline machine for each of the six test images, broken down into the number of controller operations, bus operations, non-division floating-point operations and division floating-point operations. The final column contains the proportion of instructions that are controller and bus operations, indicating the proportion of the node's execution time that is spent on data movement rather than actual vertex testing and polygon clipping. The high proportion of controller and bus operations implies the possible savings to be had by a more carefully tailored node processor. Chart 3 graphically shows the predominance of data-shuffling, rather than arithmetic, operations.

image	controller operations	bus operations	floating-point operations	division operations	proportion controller and bus operations
building scene 1	30,545,244	16,951,200	1,848,039	18,587	0.962186
building scene 2	42,551,080	26,461,652	2,981,472	26,950	0.958229
building scene 3	21,654,822	13,114,135	1,529,089	13,881	0.957508
aircraft scene 1	13,519,562	8,158,151	899,018	8,068	0.959836
aircraft scene 2	26,818,792	15,565,596	1,714,107	16,437	0.960771
ship scene	54,796,960	29,408,082	3,935,630	56,668	0.954734

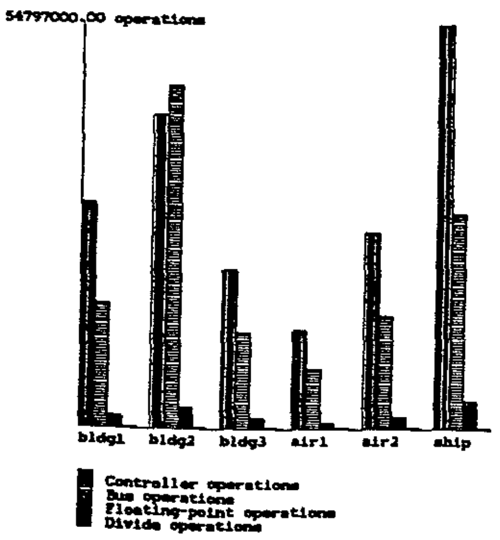


Chart 3: Relative numbers of controller operations, bus transfers, floating-point operations and division operations.

Further gains may be had via the use of multiple floating-point data paths. In this algorithm, the predominant operation is the testing of a geometric point(a vertex) against a separating plane, requiring the evaluation of the planar equation of the separator

$$Ax + By + Cz + D = distance$$

for the vertex in question. The three multiplies in this equation can be executed in parallel, and the additions may be cascaded, reducing the equation from 7 floating-point operation times to three. Finally, we note that the testing of all the vertices of a polygon might be done simultanously.

These factors alone may make a machine based on the linear pipeline architecture perform on realistic scenes in real time. Alternatively, we can reconsider the linear pipeline approach itself.

5.3.4.2. General List-Parallel Systems

The linear pipeline architecture is an example of a *list-parallel* system: it divides the overall visibility computation by distributing polygon fragment lists to be handled concurrently on different processors. In exchange for a very simple communications structure, the pipeline architecture fixes the pattern in which the workload is distributed over the cooperating processors and limits the number of processors that can be used effectively. In this section a general structure that avoids these limitations is proposed, and by discounting the additional cost of the more complex communication structure, a bound will be found on the performance of list-parallel systems. By comparing this bound to the performance of the pipeline organization, the efficiency of the pipeline architecture is shown.

In the pipeline organization, lists are distributed according to the depth of the problem's tree structure at which they occur; lists that occur at depth *i* will be handled by the *i*th processor. This organization limits the number of processors over which the calculation is distributed to the maximum depth of the tree. If the linear pipeline structure is replaced by a communication structure forming a complete graph on the set of processors, allowing the output of any processor to be passed directly to the input of any other, these two limitations are avoided, and a general list-parallel system results: any list may be passed to any processor for handling. Figure 21 illustrates a simplistic model of such a system: a ring of processors (here only 8) with a unique path from the output of each node to the input of every other node. The linear pipeline architecture is a special case of such an architecture: the initial list is input to the first of *n* processors in the system; the output of processor *k* is routed to processor *k*+1, and the output of processor *n* is routed to processor 1. This section explores performance bounds on these general list-parallel systems.

Workload Distribution In list-parallel systems, the workload is distributed in units of lists; when the first polygon in a list is produced by a clipping operation applied to its parent list, a processor must be chosen to handle that list. Ideally, the choice should be made intelligently: the assignment of lists to processors should be done in a manner to minimize the overall execution time. Unfortunately, the information required to do so isn't available: since the overall execution time is that of the longest-running processor, it depends on the number of lists assigned to that processor, the length of those lists, and any idle time that occurs as a result of delays in the arrival of input data. Of these, only the number of lists currently assigned to each processor is known: each

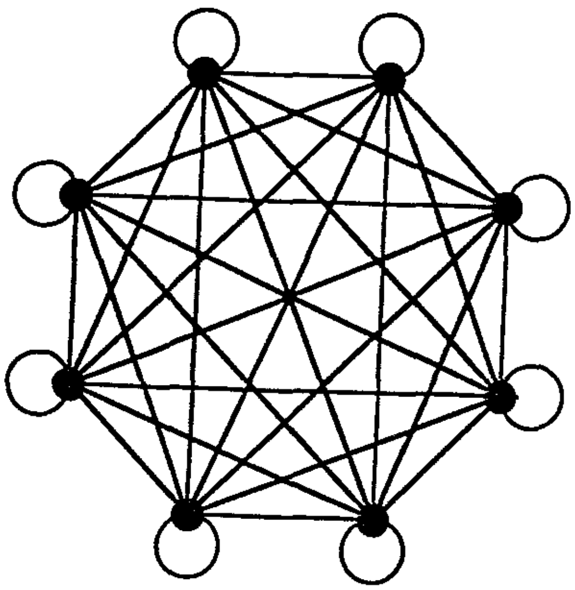


Figure 21: A Complete Communication Structure On 8 Processors

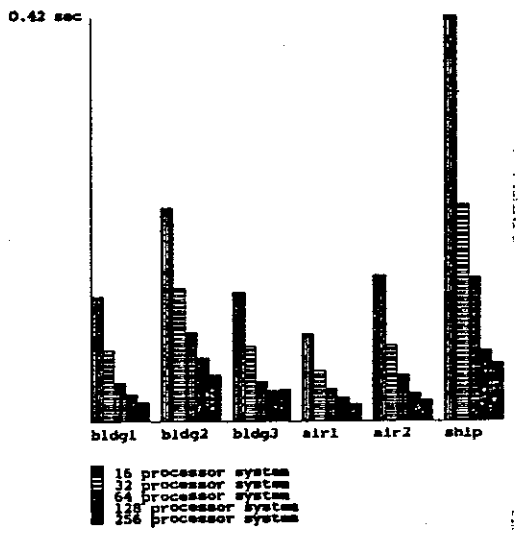
processor can simply count the lists that are currently active. The length of each active list and the intervals at which the elements of the list arrive at the node processing the list cannot be known until the entire list has been created; since this process is occurring concurrently elsewhere in the system (recall that the lists are passed systolically from the node creating the list to the node processing the list) this information is not known until the endblock of the list is encountered, at which time the list will have been completely processed and is no longer a part of the node's workload.

If an exact distribution is impossible, some heuristic must be used to determine the distribution of the lists. The linear pipeline architecture described above divides the lists according to the heuristic that the workload at each level of the tree-structure of the problem is roughly the same: lists are assigned to processors based on which level of the tree-structure at which they occur. Alternatively, we can observe that there will be many more lists than processors (as will generally be the case: of our test cases, the fewest lists (2,923) occur in aircraft scene 1) and assume that by distributing the lists randomly among the processors the total amount of work assigned to each a roughly equal amount of work will be assigned to each.

An Upper Bound On Performance For Randomly-Distributed List-Parallel Systems Using information from earlier simulations, we can estimate an upper bound on the performance of general list-parallel systems based on the given list-distribution algorithm. If communications delays and idle time due to delayed input or insufficient resources are discounted, execution time will be limited only by the maximum number of polygon fragments that must be processed in any one node. If the polygon lists that occur in the application of the polygon clipper algorithm to the test images are distributed among the nodes, the number of polygons received by each node may be counted. By applying the 29.38 microsecond/polygon time estimate to the largest number of polygons processed by any one node, we can determine a lower bound on the execution time of any general list-parallel system using the given distribution algorithm.

The lists that occur in applying the polygon-clipper algorithm to the six test cases have been distributed to a set of processors according to the random distribution algorithm described earlier, with one modification: since the first few lists that occur tend to be large (since they contain the polygon fragments that lie in relatively large areas of the screen), the first *n* lists are assigned sequentially to the *n* available processors[3]. Appendix 2 contains graphs showing the distribution of polygons to processors that result from the above distribution algorithm for general machines of 16, 32, 64, 128 and 256 processors applied to the six test images; the horizontal line indicates the ideal distribution, representing the total number of polygons processed divided by the number of processors used. Since the first five processors clip the entire set of model polygons against the five sides of the viewing frustum, they process lists that are likely to be substantially longer than any others that occur during visibility processing; since this clipping operation might be handled separately from visibility processing, the results for these first five processors may be considered inflated. For this reason, a vertical line is placed to separate the processors that perform only visibility clipping from those that also do frustum clipping. The next table shows the resulting execution times (in seconds, assuming the 29.38 microsecond per-polygon execution time derived earlier) for general list-parallel systems of the same sizes; results assuming that the frustum clip has already been done are given parenthetically. Finally, chart 4 shows execution time plotted as a function of the number of processors for the six test images.

image	total number of lists	number of processors					
		1	16	32	64	128	256
building scene 1	6,047	2.105 (1.750)	0.217 (0.133)	0.168 (0.760)	0.142 (0.041)	0.131 (0.029)	0.127 (0.021)
building scene 2	4,767	3.489 (2.845)	0.315 (0.225)	0.216 (0.141)	0.182 (0.095)	0.147 (0.068)	0.138 (0.050)
building scene 3	3,334	1.840 (1.540)	0.170 (0.137)	0.118 (0.080)	0.082 (0.043)	0.071 (0.033)	0.067 (0.034)
aircraft scene 1	2,923	1.182 (1.061)	0.088 (0.093)	0.065 (0.054)	0.040 (0.035)	0.033 (0.026)	0.031 (0.019)
aircraft scene 2	4,763	2.168 (1.976)	0.170 (0.153)	0.097 (0.080)	0.067 (0.049)	0.054 (0.030)	0.048 (0.023)
ship scene	15,084	5.096 (4.897)	0.374 (0.422)	0.202 (0.226)	0.118 (0.151)	0.088 (0.075)	0.066 (0.062)



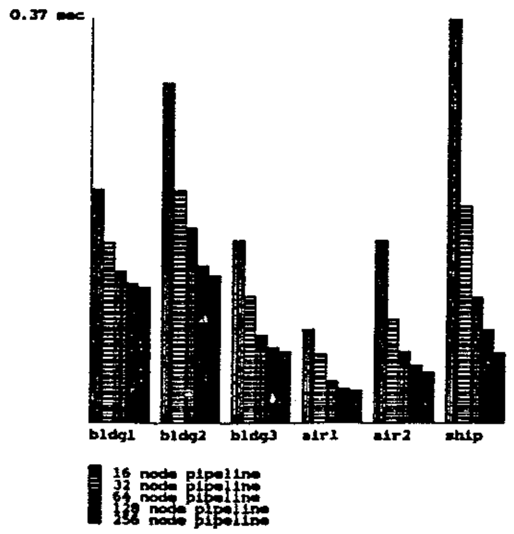


Chart 4: Execution time (in millisec) for general list-parallel systems of increasing numbers of processors for the six test images. The left chart indicates times including the frustum clip.

[3] As in the results reported for the pipeline simulations earlier, these results include clipping the entire initial polygon list to the viewing frustum, and thus are appropriate for a direct comparison with the earlier results. Because execution time on larger general list-parallel systems may depend solely on the length of a single longest list, rather than the average distribution of average-sized lists, these frustum-clip lists may distort the results. Since the frustum clip could be done externally to the visibility clip pipeline, a set of results which assume that the frustum clip has been done prior to the visibility processing has also been produced and is shown in the following table.

From these results, two important results are apparent. First, for a limited number of processors, the linear pipeline architecture proves to be quite efficient. The following table compares the estimated execution time of the linear pipeline architecture including communications delays and idle time, to the general list-parallel execution times for 16 and 32 processors. Chart 5 below compares these results graphically.

image	number of processors			
	16		32	
	pipe	general	pipe	general
building scene 1	0.289	0.217	0.177	0.168
building scene 2	0.336	0.315	0.210	0.216
building scene 3	0.179	0.170	0.127	0.118
aircraft scene 1	0.107	0.088	0.067	0.065
aircraft scene 2	0.208	0.170	0.143	0.097
ship scene	0.434	0.374	0.320	0.202

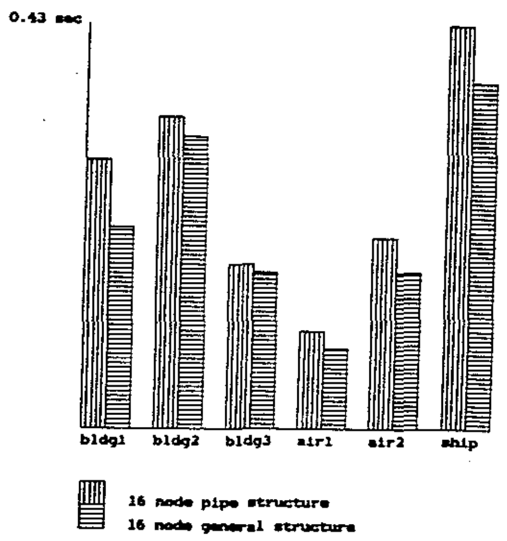


Chart 5: Performance of linear and general list-parallel systems

Finally, these results show that the marginal increase in speed due to the addition of processors past 64 is very small. An absolute limiting factor is the length of the longest list that occurs; regardless of how many processors are included, the processor that handles the longest list seems to remain the system bottleneck.

Visible Polygon Tiling

As we saw in earlier chapters, the use of an analytic visibility algorithm has significant advantages both in the generation of complex, high-quality imagery and in prospects for using parallel computation to produce such images quickly. In chapter 5, an architecture based on one such visibility algorithm was presented.

Once a set of analytically defined visible polygon fragments has been produced, a complete image generation system must still produce pixel values from the visible fragments. As was shown in chapter 1, the underlying image must be low-pass filtered to remove aliasing frequencies before sampling; providing the means to do so without prior super-sampling was one of the major advantages of analytic algorithms. In this chapter an algorithm that is designed to compute the low-pass filtered image directly from the analytic scene definition produced by the visibility algorithm will be detailed.

Again, if the entire image generation system is to operate in real time, the tiling step must itself operate in real time. Rendering a high-quality image is, however, a complex task; evaluating a lighting model generally requires several inner product evaluations; a well-filtered texture map reference may take several table lookups. Rendering a complete image may require hundreds of thousands of such calculations. In order to provide the necessary performance, it will be necessary to look for ways to divide this task among cooperating processors. In the final section of this chapter one possible architectural approach to this problem will be presented.

6.1. Anti-Aliasing Techniques

As we saw earlier, each sample of a filtered image is produced by integrating the product of a filter kernel, centered at the sample point, and the underlying image function:

$$I_{filter}(x,y) = \iint I(u,v)F(x-u,y-v)\,du dv$$

Evaluating these integrals presents several difficult problems. We currently have no method of evaluating the image function continuously, nor a method to form the continuous product of two continuous functions, nor a method to integrate the continuous result. We can, however, *approximate* the underlying image function: by assuming that the image function is constant over a sufficiently small area we can approximate it by evaluating a single sample within the area. This allows us to decompose each filter integral into the sum of integrals over each such area and factor the constant intensity out of the integral:

$$\iint I(u,v)F(x-u,y-v) \approx \sum_{subareas} I_{subarea} \iint_{subarea} F(x-u,y-v)\,du dv$$

This approach is typified by the super-sampling algorithm. The underlying image function is first sampled at a sufficiently high resolution to reduce aliasing effects to a tolerable level. Each super-sample serves as a constant approximation of the underlying image function in a grid square centered at the super-sample point. To evaluate an output pixel value, each sample lying inside the domain of a filter kernel centered at the output sample is weighted by the integral of the filter kernel over the corresponding grid square, and the results for each grid square covered by the filter kernel are summed.

6.1.1. Sources of Aliasing Effects

Aliasing problems in the underlying image function can be ascribed to two sources: changes in shade that occur across the face of a single visible surface and changes in shade that occur at the boundaries between dissimilar surfaces[1].

[1] A third aliasing problem, the temporal effects caused by the discrete sequences of images through time in a moving sequence, is not addressed in this work.

6.1.2. Aliasing in the Shading Function

Within the boundary of a visible surface, changes in image intensity are due to variations of the *shading function* applied to the surface. A shading function is associated with each surface to determine the shade of each point on the surface given a viewpoint and the sources of light falling on the surface. Shading functions combine characteristics of the surface, such as the shininess of the surface, with parameters that are specific to the point on the surface at which the shading function is to be evaluated: the surface normal at the sample point is used to determine the effects of the light sources at the point; interpolated indices can be used to map color textures or bump maps onto the surface.

Evaluating a shading function requires that functions of surface interpolants be evaluated[2]. For example, a color texture can be mapped onto a surface by interpolating texture indices u and v across the surface (figure 1):

$$ColorTexture(u,v)=(Red(u,v),Green(u,v),Blue(u,v))$$

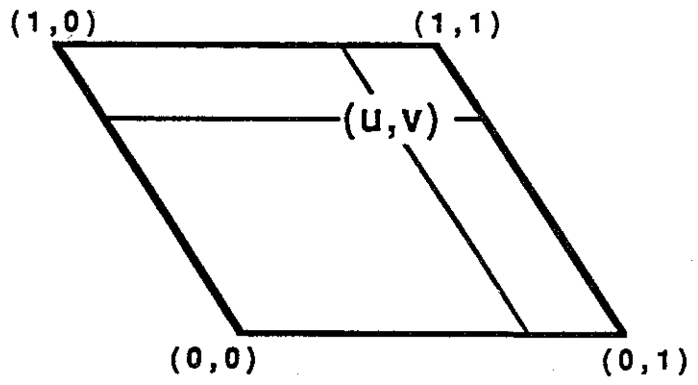


Figure 1: Texture-Mapping Interpolants

[2] In this chapter we will assume that these functions are defined over the range from zero to one in both directions.

A simple and effective lighting model expresses the effects of a light source on a surface as a combination of two functions of the surface normal, where the surface normal may be either the varying normal of a truly curved surface or interpolated to give a flat surface the *appearance* of being curved (figure 2) [NeS,9]:

$$\mathit{Intensity} = \mathit{DiffuseIntensity} + \mathit{SpecularIntensity},$$

where:

$$\mathit{DiffuseIntensity}(\mathit{NormalVector}) = (\mathit{NormalVector} \cdot \mathit{LightVector})$$

$$\mathit{SpecularIntensity}(\mathit{ReflectedVector}, \mathit{LightVector}) = (\mathit{ReflectedVector} \cdot \mathit{LightVector})^{\mathit{power}}$$

If any of these functions of the surface parameters contain frequencies higher than the Nyquist frequency for the final output resolution, they must be filtered before the final sampling is done. Just as super-sampling can reduce aliasing energies resulting from the boundaries between dissimilar surfaces, it can reduce aliasing energies resulting from high frequencies in the shading function, by sampling the image function at a sufficiently high resolution to reduce aliasing

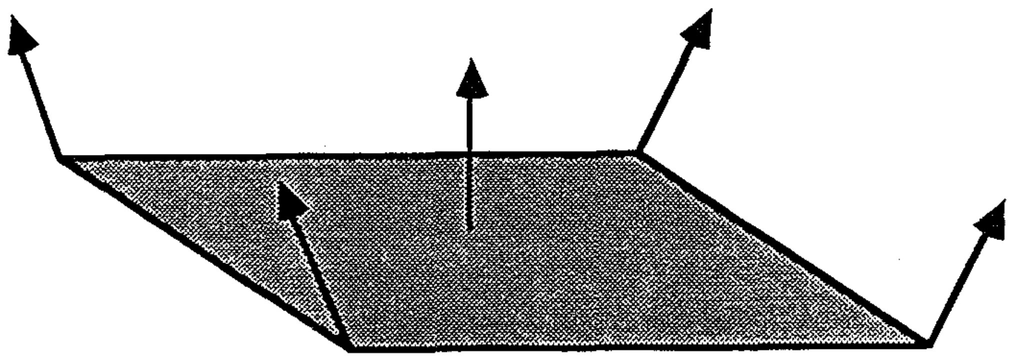


Figure 2: Surface-Normal Interpolation for Simulating Curved Surfaces

energies to a tolerable level, it produces a representation of the image function that can be used as input to a low-pass filtering operation that removes frequencies that are unrepresentable at the desired output frequency before the final sampling operation.

Filtered Sampling

Aliasing in the shading function differs from aliasing across surface boundaries in one very important way: the shading function to be applied to a surface is completely known before the image is rendered. The component functions DiffuseIntensity and SpecularIntensity depend solely on the position of the viewpoint and the light sources and can be expressed analytically as a function of these parameters without regard to the interrelation of surfaces; similarly, the texture pattern mapped onto a surface is associated with the surface as a part of the modelling process and can be expressed as a function of the surface interpolants without regard to the position or orientation of the surface in the scene. It is therefore possible to process these functions *before* sampling so that they can be filtered efficiently as a part of the sampling process.

The difficulty in doing so is that, although the functions themselves are known before sampling, the rate at which they will be sampled is not known. For example, the same function may be applied to a surface near the viewpoint, resulting in a high sampling rate, or very far from the viewpoint, when it may only be sampled a few times. Thus the functions must be prepared so that it is easy to produce result filtered for a wide range of cutoff frequencies. Two fast table-driven algorithms are available.

Pyramidal Parametrics One technique [Wil,83], known as MIP-mapping (from the latin *multum in parvo*, "many things in a small place"), uses a pre-processing step to produce a set of tabulated functions, each filtered for a different sampling rate. The first tabulation generally consists of 256x256 values of the function filtered for sampling intervals of 1/256 in the u and v directions. Each successive tabulation is then filtered for half the sampling rate of its predecessor and is stored at half the resolution.

This process produces a three-dimensional tabulated function, with the first two dimensions corresponding to the spatial dimensions of the original function and the third, "up", corresponding to the required degree of filtering. Evaluating a sample is then done by an interpolation from this table (figure 3).

Although this technique can produce very good results, it suffers from the failing that it assumes that the sampling rates are the same in the two spatial dimensions. In real images, this is very likely not the case. Figure 4 shows a simple case in which u has a much higher sampling rate than v. When this occurs, it is necessary to assume the slower sampling rate, overly filtering the function in one direction, but preventing aliasing effects in the other direction.

Sum-Table Filtering An alternative method of table-driven filtering allows for differing sampling rates in the two spatial directions [Cro,84, Gla,86]. For each sample, this algorithm defines the filtering region of the function domain to be the rectangular region centered at the sample point and of dimensions reflecting the sampling rates: if samples in the U and V directions occur with spacing dU and dV, the rectangular region is dU high and dV wide. The algorithm then assumes the use of a simple box filter function over this rectangle, producing the average value of the

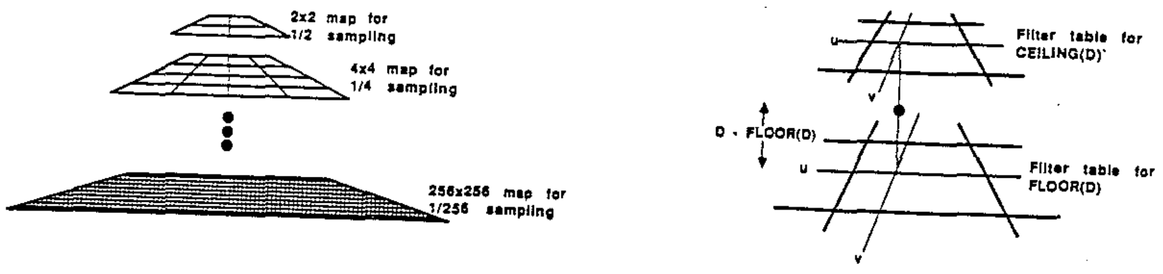

2x2 map for 1/2 sampling
4x4 map for 1/4 sampling
256x256 map for 1/256 sampling
u
v
Filter table for CEILING(D)
D - FLOOR(D)
u
v
Filter table for FLOOR(D)

Figure 3: Interpolation in MIP-Mapping

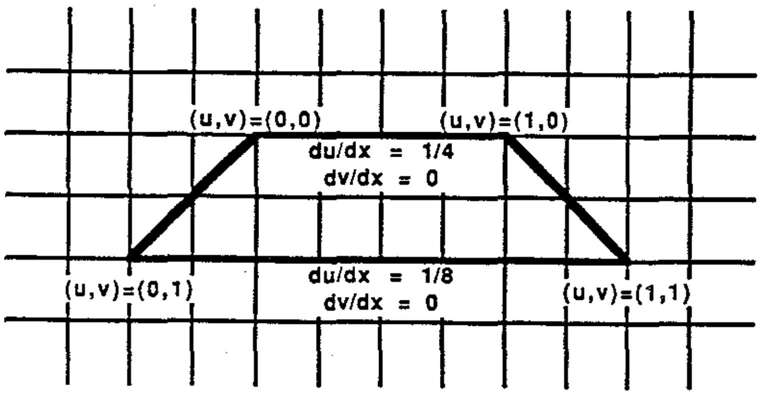


Figure 4: Differing Sampling Rates in U and V

function over this rectangle.

This result can be produced very quickly by the use of an integration table (due to its discrete nature, known as a "sum-table"):

$$Table(x,y) = \int_0^x\int_0^y F(u,v)\,du\,dv$$

Given this table, it is easy to determine the integral of the function any rectangular area of the domain (Figure 5):

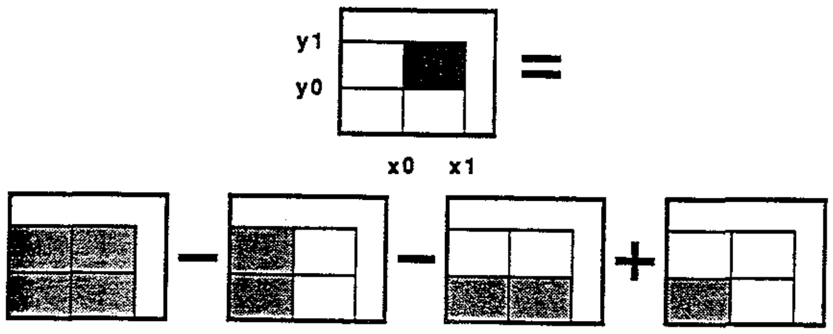


Figure 5: Integration over a rectangular region of the domain

$$\int_{x0}^{x1}\int_{y0}^{y1} F(u,v)\,du\,dv =$$

$$\int_{0}^{x1}\int_{0}^{y1} F(u,v)\,du\,dv - \int_{0}^{x0}\int_{0}^{y1} F(u,v)\,du\,dv - \int_{0}^{x1}\int_{0}^{y0} F(u,v)\,du\,dv + \int_{0}^{x0}\int_{0}^{y0} F(u,v)\,du\,dv$$

By dividing the resulting integral by the area of its domain, the result is produced.

This algorithm, like the previous one, has problems. The first is the obviously questionable assumption of a box filter, far from optimal for sharp frequency cutoff characteristics. Also, if a single sample is taken for an axis-aligned sample square in the image plane (as is generally the case), we should integrate over the projection of this square in the domain of the function. Unfortunately, the effects of coordinate transformations map this square into a non-axis aligned region that is not even guaranteed to be a parallelogram. Even so, this technique can produce very good results.

6.1.3. Aliasing at Surface Boundaries

Another source of aliasing frequencies in the underlying image function are the discontinuities caused by the boundaries between differing surfaces. When the domain of the filter kernel centered at an output sample overlaps the boundary between surfaces, the correct result represents not only the shading functions for the different surfaces, but also the region of the filter kernel domain that is covered by each surface:

$$Sample(x,y) = \iint Image(u,v) Filter(x-u,y-v)dudv =$$

$$\sum_{\substack{\text{visible}\\\text{fragments}\\\text{intersecting}\\\text{filter domain}}} \iint_{\text{fragment area}} Shade_{fragment}(u,v) Filter(x-u,y-v)dudv$$

In order to evaluate the filtering integrals correctly it is necessary to identify the different surfaces lying under the filter kernel. Visibility must therefore be determined to a higher frequency than the output resolution; otherwise, the edge will appear jagged.

The simple super-sampling anti-aliasing algorithm also provides this function: by evaluating many point-samples within the filter kernel domain, visibility is determined at the resolution necessary to alleviate aliasing effects. Unfortunately, however, to do so the underlying image function must be sampled at a much higher frequency than necessary. As we saw above, we can efficiently evaluate a filtered version of the shading function for a surface at the output resolution without unnecessary loss of detail; if we must super-sample the image function to determine coverage within the filter domain, this advantage is lost. Instead, we might separate evaluations of the shading function from visibility tests to make these tests as inexpensive as possible.

Adaptive Super-Sampling for Visibility Carpenter, in [CPC,84] describes the A-buffer, a method for separating the filtering of the shading function for surfaces from the filtering of visibility for coverage. In this algorithm, a Z-sorted list of polygon fragments that overlap each output sample square is accumulated as the potentially-visible surfaces in the scene are processed. These fragments take one of two forms: if the fragment completely covers the sample square, the list entry simply contains the color of the fragment. If, however, the fragment only *partially* covers the sample square, the list entry contains a 4x8 subpixel bitmask, representing the portion of the sample square that is covered in addition to the results of a single evaluation of a shading function filtered for the output resolution.

When all fragments that overlap a sample square have been entered, the fragments are merged in a front-to-back pass through the fragment list by accumulating a mask showing the current coverage of the sample square. At any point in this pass, the contribution of the current fragment to the output sample is found by weighting the fragment shade by the proportion of the bitmask it covers which has not already been covered by nearer surfaces. In this manner, visibility and shade are evaluated the resolution at which they are needed.

6.2. Analytic Visibility, Sampling and Anti-Aliasing

When visibility is known analytically *before* sampling, it is not necessary to super-sample to determine visibility within each filter kernel; instead, the exact area of intersection of the visible surface with the filter kernel is known analytically. Determining the contribution of the visible surface to an output sample therefore requires only the following steps:

Step 1.

The division of the visible surface into sample squares over which a constant approximation is adequate;

Step 2.

For each such sample square:

Step a.

The evaluation of a single sample inside the sample square to serve as the approximation;

Step b.

The integration of the filter kernel over the sample square;

Step c.

The weighting of the sample approximation by the integral of the filter kernel;

Step 3.

Summing the results for each sample squares to provide the final contribution.

Previous Filter Integration Techniques

There are several known algorithms for evaluating the integral of a filter kernel over an irregular boundary. In this section, three will be discussed: the most common approach of super-sampling for digital integration, the area-sampling algorithm of Catmull [Cat,78] and the algorithm of Feibush, Levoy and Cook [FLC,80], originally developed for use with the Weiler/Atherton analytic visiblity algorithm.

Super-Sampling and Digital Integration The super-sampling algorithm as presented earlier can easily be extended to operate on the visible surfaces produced by an analytic visibility algorithm. First, as above, the visible surface is subdivided into sample squares. Within each sample square, integration over the irregular intersection of the visible polygon with the sample square is performed by super-sampling; a bit mask of the sample area is generated containing ones for super-samples that are covered by the visible polygon and zeroes otherwise. Integration is then done using a tabulation of the filter kernel at the super-sampling resolution; the approximate integral is the sum of table entries for which the corresponding bit mask is a one.

$$OutputSample(x,y) = Color(x,y) * \sum_{\substack{-KernelRadius<i<KernelRadius \\ -KernelRadius<j<KernelRadius}} W(i,j)*Mask(X+i,Y+j)$$

where:

$$Mask(I,J)=\begin{cases} 1, & \textit{super-sample (I,J) lies in the visible surface boundary,} \\ 0, & \textit{otherwise} \end{cases}$$

Catmull's Area-Sampling Algorithm In 1978, Catmull published a scan-line polygon tiling algorithm which combined the visibility calculation with an area-sampling anti-aliasing technique. This algorithm considered the image plane to be divided into a rectangular non-overlapping grid, each centered at an output pixel. The algorithm is designed to determine the proportion of each square that is covered by each surface that overlaps it. The shades of these surfaces are then weighted according to this proportion, amounting to the application of a box filter.

The algorithm first uses a standard scan-line framework to maintain a list of "active" polygons that intersect each scan-line as it is encountered in a top-to-bottom pass through the image. To begin processing an individual scan-line, each polygon in this active list is clipped against the rectangular area of the image plane corresponding to the current scan-line. The fragment lying outside the rectangle is replaced into the active list. The fragment lying inside the rectangle is further divided into three sections (see figure 6): the irregular piece corresponding to sample-squares intersected by the left edge of the polygon, the center piece consisting of sample squares totally covered by the fragment, and a right irregular piece intersecting the right edge of the polygon. Each of these fragments is then placed into a X-bucket matching its leftmost sample square.

This X-bucket structure is used to maintain an active-span list, kept sorted by Z, during a left-to-right pass across the scan-line. Processing for a given pixel begins by merging the fragments encountered at the pixel location into the X-active list. This list is then examined to determine coverage for the pixel. If the first element in the list completely covers the sample-square, the shade of that surface is stored directly into the frame buffer. If, however, the first

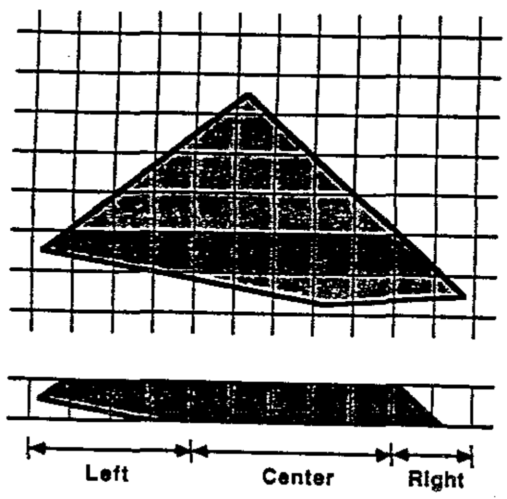


Figure 6: Span decomposition in Catmull's algorithm

element *does not* cover the sample square entirely, a more complex calculation is necessary. If the *second* element completely covers the sample square, the calculation is simple: the area of the first surface is determined and used to weight the two visible surfaces at that sample square.

If, however, the second surface does not completely cover the sample square, a more complex "pixel integrator" must be used to evaluate the partial coverage of several surfaces visible within the sample square. The pixel integrator algorithm is an analytic visibility algorithm much like the one described in earlier chapters, except applied completely within a single sample square. This produces a set of simple convex fragments that are visible in the sample square and hence contribute to the shade of the corresponding pixel. The pixel is then assigned the sum of each visible fragment's shade weighted by its area.

The Feibush/Levoy/Cook Algorithm This algorithm was developed to analytically filter the visible polygons resulting from the Weiler/Atherton analytic visibility algorithm. For a given filter kernel, a bounding box encompassing the filter kernel region is determined (perhaps spanning several pixels in radius). Then, for every pixel in a visible polygon, the bounding box is centered at the pixel and the edges of the polygon are tested against it. For each pixel where the filter region bounding box lies entirely within the polygonal boundary, the pixel receives no contribution from other visible surfaces and the pixel can be assigned the color of the current surface directly (figure 7.a). Other pixels, where the filter kernel overlaps the edge of the polygon (figure 7.b), will receive the shade of the current surface weighted by the proportion of the volume of the filter kernel lying within the boundary of the surface.

To determine this proportion for a given pixel, the visible surface is first clipped against the edges of the filter kernel bounding box centered at the pixel. Each edge of the resulting polygon (which, since visibility was originally computed via the Weiler/Atherton algorithm, may be concave and contain holes) is oriented so that, if the edges of a contour are traversed in a clockwise order, the area to the right of the edge corresponds to the interior of the polygon (figure 8.a). For each such edge, a triangle is formed using the two endpoints of the edge and the pixel center (figure

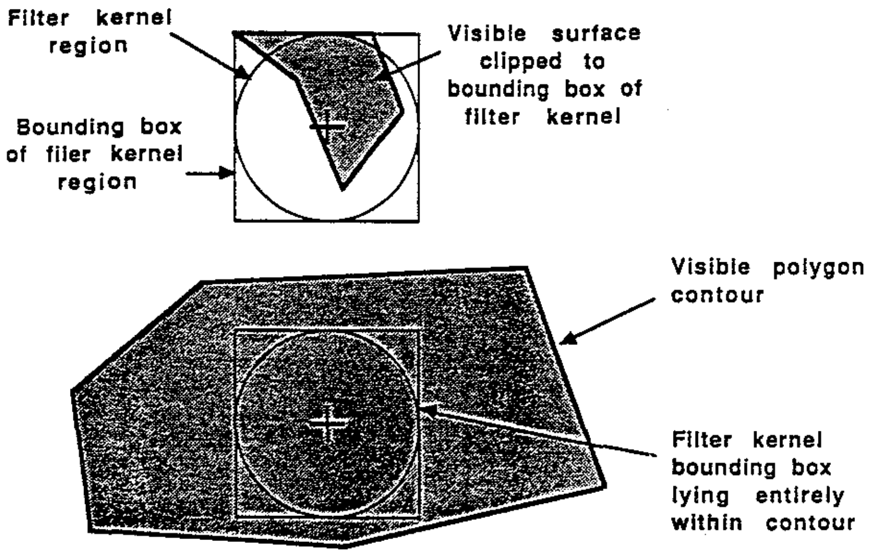

Filter kernel region
Bounding box of filer kernel region
Visible surface clipped to bounding box of filter kernel
Visible polygon contour
Filter kernel bounding box lying entirely within contour

Figure 7: Filter/polygon overlap in Feibush/Levoy/Cook
a. Complete coverage
b. Partial coverage

8.b). From each such triangle two right triangles are defined by the pixel center, one endpoint and the intersection of an line perpendicular to the original polygon edge and passing through the pixel center with the original edge (figure 8.c). The volume of the filter kernel over these triangles can be found by using the base and height of the triangles as indices in a table-lookup process. If the perpendicular intersection point lies outside the original edge along the line defined by the edge (such as in figure 8.d), the volume over the original triangle is the difference between the volumes over each right triangle; otherwise, it is the sum.

The actual volume of the filter kernel over the irregular shape of the clipped polygon fragment can then be found from these triangular volumes. Using the orientation of the edges to

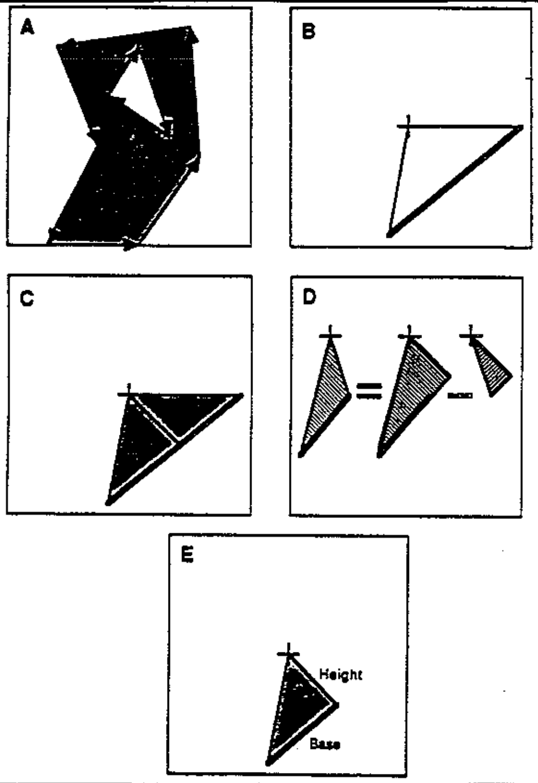


Figure 8: Triangle decomposition in Feibush/Levoy/Cook
a. Contour edge orientation
b. Triangle formed by an edge and the pixel center
c. Right-triangle decomposition: intersection point in interior
d. Right-triangle decomposition: intersection point in exterior
e. Parameterized right triangle

determine whether the edge defining the triangles is a right or left polygon boundary, the triangle volumes are added to a sum if the cross product between the edges connecting the pixel center to the edge endpoints is positive, and subtracting them from the sum when the result is negative (see figure 9).

6.3. A New Analytic Integration Algorithm

In two of these previous algorithms, the integral of a filter kernel over an irregular area is evaluated by considering the irregular area to be the sum and difference of regular regions over which the integral is known. In the super-sampling case, the regular regions are the areas of the super-sampling grid squares and the integral over each is found in the weight table. The method of Cook, Levoy and Feibush reduces the area to the sums and differences of right triangles and a table indexed by the length of the two right sides of a triangle produces the integral over the triangle.

A new algorithm, developed in collaboration with Lee Westover and Turner Whitted [AWW,85] provides an efficient method of evaluating these integrals. Like both the above algorithms, the new algorithm decomposes the irregular shapes into regular shapes for fast table-driven integration. Like the Cook, Levoy and Feibush algorithm, this algorithm (in the vast

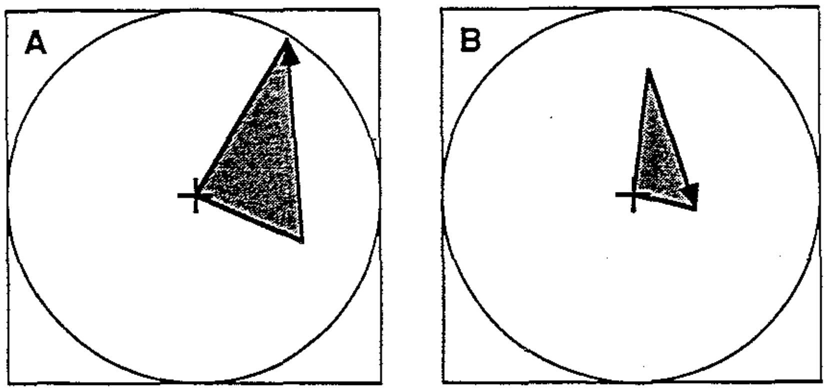


Figure 9: Addition and subtraction based on cross product
a. Area to be added to sum
b. Area to be subtracted from sum

majority of cases) operates at the output resolution; for a given output resolution, accuracy depends only on table size and not on execution time. In a very few cases situations are encountered that can not be handled by the algorithm; a subpixel mask (similar to that described above under supersampling) is used to handle these cases.

6.3.1. Sample-Square Decomposition

The Abram, Westover and Whitted algorithm uses a sample-square decomposition to regularize the shape. Given an output pixel grid, the image area is divided into abutting sample squares centered at each output pixel. Much like the Catmull algorithm, the algorithm uses an outer loop passing down the surface in the Y direction, clipping the surface to each row of sample squares that it intersects (see figure 10). An inner loop then tracks the edges of the clipped portion across the row, in effect (though not in practice) clipping the polygon to each sample square.

Polygon/Sample Square Intersections

Excepting the case where actual polygon vertices are found in the interior of a sample square, the intersection of a polygon with a sample square consists of a (frequently empty) set of "interior"

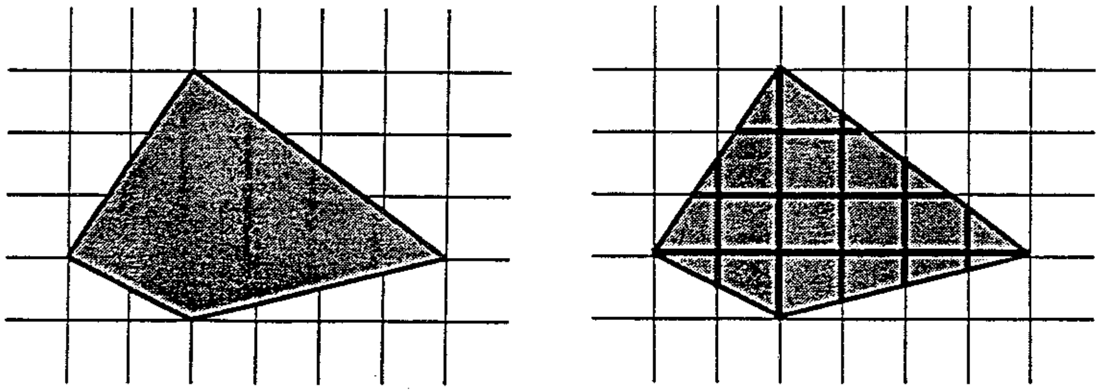


Figure 10: Sample-square decomposition of polygon

edges, passing from edge to edge through the interior of the sample square, and some subset of the edges of the sample square itself (see Figure 11). Each such interior edge is oriented and defines a region of the sample square lying outside the polygon. If we begin with the integral of a filter kernel over the entire area of the sample square and successively subtract the integral over the area excluded by each interior edge, the result will be the integral over the actual polygon/sample square intersection. Note that this can be done simultaneously for each filter kernel that overlaps the sample square.

Edge/Sample Square Interactions

It remains to determine the integral of a filter kernel over the areas excluded by the interior edges. After rotational symmetries have been removed, such edges can intersect a sample square in only three ways (see figure 12): by excluding a triangular region, a trapezoidal region or a pentagonal region. Noting that the pentagonal region can be handled by subtracting the complementary triangular region from the whole area, we are left with only two. Each of these can then be parameterized by the edge lengths shown in figure 12, and a table lookup can produce the desired integrals.

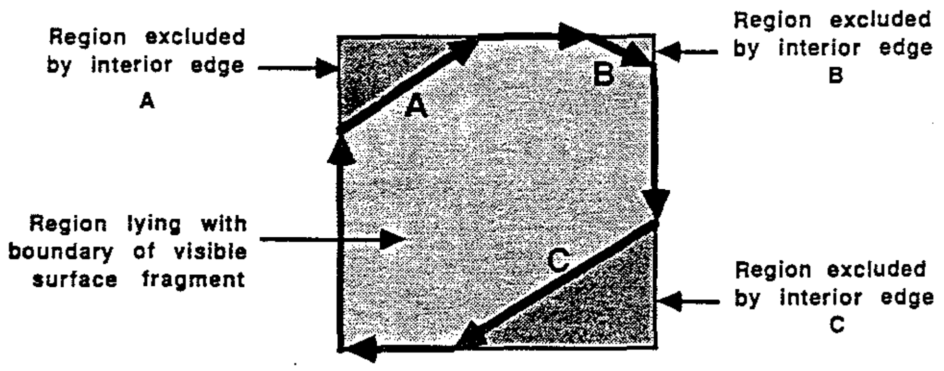

Region excluded by interior edge A
Region excluded by interior edge B
A
B
Region lying with boundary of visible surface fragment
C
Region excluded by interior edge C

Figure 11: Intersection of polygon with sample square

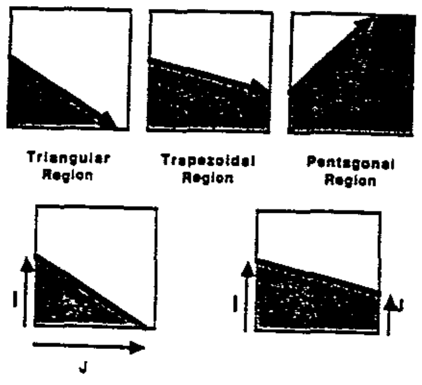


Figure 12: Interior edge types and parameterization

Punt Cases

The above algorithm fails in the relatively rare case that a vertex lies in the interior of the sample square (figure 13.a). In these cases, the algorithm falls back on a table-driven super-sampling approach. Edges incident on interior vertices are extended to the boundary of the sample square. Each resulting edge, together with any other interior edges, are parameterized as before. In this case, however, each table lookup produces a bit mask containing ones for super samples on the *included* side of the edge and zeroes otherwise (figure 13.b). By AND'ing these bit masks together a final bit mask containing ones for super samples lying within the fragment boundary and zeroes otherwise is produced (figure 13.c). Finally, the actual integral over the fragment is produced by summing the integrals over each super-sample square covered by the fragment.

Fragment Type Breakdown

This algorithm has been implemented and has been used to generate a short videotape ("A Trot Through The UNC Computer Science Building") consisting of over 1500 images containing over 1,000,000 polygons. These polygons resulted in approximately 250,000,000 sample-square fragments. The following table shows the breakdown of fragment types:

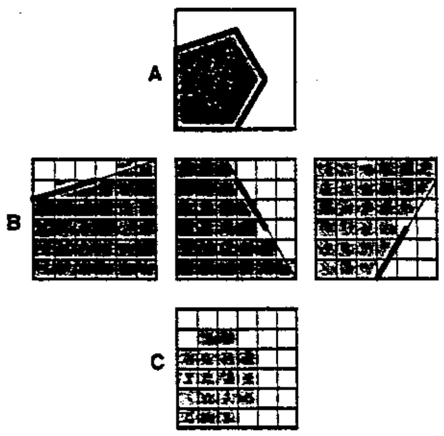


Figure 13: Handling punt cases
a. A fragment with interior vertices is a punt case
b. Edge extension and resulting bit masks
c. AND'ed bit masks give coverage mask

Type	Fraction of total cases
Total Coverage	85.6%
1 Interior Edge	13.0%
>1 Interior Edge	0.6%
Interior Vertex (Punt Case)	0.8%

6.3.2. Overlapping Filter Kernels

It is widely believed that improvements in image quality can be had by using filters that span several sample-squares. Figure 14 below illustrates the areas covered by circular filter kernels centered at output samples that have 1.5 output-sample radius: a fragment in the center sample-square contributes to the shade of each output sample in a 3x3 neighborhood of the sample. The Abram, Westover and Whitted integration algorithm can be extended to provide for such cases. The fragment is parameterized exactly as above, but now the integration tables return nine values: the integrals of every filter kernel overlapping the sample square over the fragment. These values are then used to prepare contributions to the 3x3 output samples centered at the fragment sample-square.

Now, however, the rotation of the fragment becomes significant. In the case of figure 15, 90 degree rotations of the fragment do not change the effect of the fragment on the center sample (assuming a rotationally symmetric filter kernel), but it do alter the effect of the fragment on the samples above and below. Thus, when the fragment is parameterized the implied rotation must be stored and used later to rotate the resulting integral table accordingly. (Alternatively, this table rotation can be avoided by using 8, rather than two, tables: we store triangle and trapezoid tables for each of the four rotations).

6.3.3. The Sample-Square Decomposition Algorithm

The actual "dicing" phase of this algorithm, in which the analytic polygon definition is clipped against sample-squares to form regularized fragments, is actually very simple and fast. Since edges of a sample-square fragment that lie along an edge of the sample-square edge do not affect the table-driven integration process, these edges need not be explicitly calculated; it is only necessary to identify the sample squares that intersect the polygon and, for each, determine the

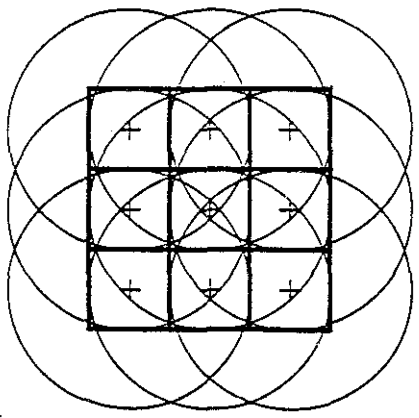


Figure 14: Nine filter kernels overlapping fragment

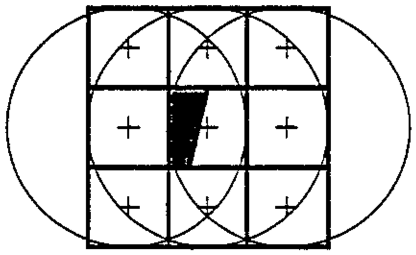


Figure 15: Case requiring table rotation

intersection of the actual polygon contour with each. In this section an algorithm for doing so is discussed. Appendix 2 contains a C-language implementation of this algorithm.

The algorithm begins with an edge-setup step which forms two lists of edges corresponding to the left and right contours of the (convex) polygon. An outer loop then passes down the image plane determining the fragments of the left and right contours that lie within each row of sample squares that intersects the polygon. These fragments, along with the fragments of the top and bottom edges of the sample row that lie between the left and right contours, are formed into two new contour lists: one representing the upper contour of the intersection of the polygon with the sample row, and the other, the bottom. Given these two contour lists, an inner loop then passes across the row, determining the intersection of the row-fragment contour with each sample square across the row.

The Edge-Setup Phase

In order to simplify the intersection of the edges of the polygon with the edges of rows and columns of sample squares, forward differences representing both the step in X per integral step in Y (dx/dy, to be used in determining successive intersections with the horizontal edges of the sample row) and the step in Y per integral step in X (dy/dx, to be used in intersecting the edges with successive vertical boundaries between sample squares) are calculated. If any values are to be interpolated across the face of the polygon, similar forward differences are computed for each.

The Y-Loop

The Y-loop must determine the segments of the original polygon contour that lie within the top and bottom boundaries of each row of sample squares that the contour intersects. The left and right contours are tracked using data structures which maintain the current state of the contour as the algorithm passes from row to row down the image plane. These data structures, which containing such information as the Y value at which the edge terminates, the current values of X and other edge interpolants and with their forward differences, are initialized to the initial state of the topmost edge in either contour list.

Beginning with the topmost row that intersects the polygonal contour and passing down row to row, the algorithm first checks to see if the current left edge terminates inside the current row. If so, the remaining fragment of that edge is added to the row-contour list. The left data structure is then updated to the initial state of the next left edge. This process continues as long as edges terminating within the current row are encountered. Finally, (if the bottommost vertex lies below the current row) a new edge, beginning with the current state of the left edge and terminating with the intersection of that edge with the lower boundary of the row, is produced. If no contour vertex has been encountered in the current row (as is most often the case), this intersection is found trivially by adding the Y forward differences of the edge on to its current state; otherwise, a slightly more difficult calculation is required to calculate the intersection point.

The X-Loop

Much like in the Catmull algorithm, the sample-rectangle fragment is considered to consist of three sections: a left section, in which the original polygon contour (and hence either or both of the top and bottom contours) actually passes through the interior of the sample-rectangle; a center section, in which the top and bottom contours lie along the top and bottom edges of the sample-rectangle, and a right section similar to the left. We note that, since sample squares in the central section are completely covered by the fragment, we need not track the fragment contour; thus processing for completely covered sample squares (which comprise 85.6% of the total number of fragments) requires only the interpolation of surface characteristics; thus, in over 85% of the cases, the inner loop of sample-square decomposition requires no more work than that of simplest point-sampling polygon scan-conversion.

Processing Edge Sections

Sample-squares lying in the left and right sections of the fragment consist of three types: first, those that contain at least one vertex of the original polygon contour which falls in the *interior* of the sample square; those that contain vertices of the rectangle contour that lie on the sample-square edge, and those which do not contain a contour vertex but which contain at least one interior edge. Processing these sections consists of tracing along the top and bottom contours from vertex to vertex, interpolating the Y position of the contour at each sample-square boundary from contour vertex to contour vertex, again a simple linear interpolation that can be handled using forward differences, but now using the dy/dx difference. At each intersection point the Y value is easily converted to a table parameter.

Whenever a vertex lying on the top or bottom edge of the sample-rectangle is encountered, several operations may be required. First, the vertex may complete an interior edge, a case easily detected by examining the dy/dx forward difference of the edge entering the vertex: if it is non-zero, the previous edge was an interior edge. If so, the X value of the vertex is parameterized. Next, if the vertex is not the rightmost (thus terminating) vertex of the sample-rectangle fragment, the edge-interpolation data structure must be updated to reflect the new edge. The vertex may begin an interior edge (if the departing dy/dx is non-zero); if so, it is parameterized. If so, and if the other end of the edge does not lie in the current sample-square (either in the interior or on the edge) the intersection of the edge with the right-hand sample-square boundary must be computed.

The most difficult case arises when a vertex lying in the interior of a sample-square is encountered. These are the 'punt' cases and comprise 0.8% of the total number of fragments. As described earlier, each edge incident on an interior vertex must be extended to the edges of the sample square. In most cases, one end of the edge will already lie on the sample-square edge; only when the sample-square contains two successive interior vertices or when the sample square is the leftmost sample-square of the fragment will the edge have to be extended both ways.

6.3.4. A Software Implementation

This algorithm has been implemented on a VAX 11/780 in the C programming language. This section describes this implementation along with timing results.

Input This program is designed to accept the output of the pipeline polygon clipper simulator described in the last chapter, consisting of a list of polygon blocks. Each polygon block begins with an information word, defining the contents of the block, some number of header words containing information which is global to the polygon, followed by a list of vertex blocks which contain the coordinates of the vertices (in the eye coordinate system) and the value of each surface interpolant at the vertex.

Output The output of this algorithm consists of a set of fragment blocks defining each fragment for subsequent shading and weighting. Each block consists of three sections: a header, consisting of two 16 bit words containing the (x,y) address of the sample-square containing the fragment; a (frequently empty) set of interior edge descriptors, each containing two two-bit fields denoting which sample-square edges the interior edge is intersects, two five-bit intersection parameter fields and a one-bit flag denoting whether either end of the interior edge was incident on an interior vertex; and finally, for each surface interpolant, the maximum and minimum values for the interpolant over the area of the fragment (in 32-bit floating point).

Timing Results The time required by this implementation of the algorithm can be expressed as the sum of the time spent in each of the three phases: the setup of the polygon edges (including the perspective transformation of each vertex), the setup of the horizontal spans, and the production of sample-square fragments (plus a negligible amount of per-polygon overhead). The cost of each of these phases is separated into algorithm overhead plus the marginal cost of each interpolant; for example, the time required to prepare a horizontal span includes the time necessary to set up the top and bottom contours plus the time required to compute the *di/dx* forward difference for each interpolant. Thus we arrive at the following model for the time used by this implementation:

$$Time_{total} = (Setup\ Time) + (Span\ Time) + (Fragment\ Time)$$

where:

$$Setup\ time = T_{edge} \times N_{edges} + T_{edge\ interpolant} \times (N_{edges} \times N_{interpolants})$$

$$Span\ time = T_{span} \times N_{spans} + T_{span\ interpolant} \times (N_{spans} \times N_{interpolants})$$

$$Fragment\ time = T_{frag} \times N_{frags} + T_{frag\ interpolant} \times (N_{frags} \times N_{interpolants})$$

Estimates of the timing parameters of this model have been made by timing 24 runs of the VAX implementation. The analytic visible-surface algorithm of chapter four was used to generate sets of visible polygon fragments for four scenes: three interior views of the UNC Computer Science Building database and one exterior view of the ship database (figures 1,2,3 and 6 of chapter 1). Each was processed six times varying the resolution and number of parameters interpolated across the surface. In order to improve the accuracy of these estimates (the UNIX *getrusage*() timing utility is accurate only to the nearest 1/60 second), each run actually processes the input set of polygons ten times[3]. The results are given in the following table:

[3] These times do not include any I/O time; the entire input was read into a memory-resident data buffer before the clock started and similarly, results are saved in a memory-resident output buffer until the clock is stopped. Any system time incurred as part of memory paging is excluded by the timing facility.

Run	$N_{interpolants}$	N_{edges}	N_{spans}	N_{frags}	Time(sec)
0	1	83,150	371,520	279,730	863.27
1	1	166,020	371,550	402,440	1094.17
2	1	165,820	371,550	543,860	1179.79
3	1	72,690	94,030	375,350	476.69
4	1	61,270	70,950	442,540	437.71
5	1	61,420	47,470	794,310	552.52
6	1	143,830	94,030	663,600	774.73
7	1	121,630	70,940	801,350	716.46
8	1	122,030	47,480	1,502,640	971.18
9	1	143,830	94,030	1,159,270	985.86
10	1	121,630	70,940	1,453,640	1045.00
11	1	122,030	47,480	2,865,420	1626.68
12	4	83,150	371,520	279,730	1222.97
13	4	166,020	371,550	402,440	1583.39
14	4	165,820	371,550	543,860	1698.12
15	4	72,690	94,030	375,350	714.13
16	4	61,270	70,950	442,540	674.82
17	4	61,420	47,470	794,310	855.72
18	4	143,830	94,030	663,600	1134.83
19	4	121,630	70,940	801,350	1107.49
20	4	122,030	47,480	1,502,640	1587.25
21	4	143,830	94,030	1,159,270	1503.27
22	4	121,630	70,940	1,453,640	1578.93
23	4	61,015	23,740	1,432,710	1270.52

Setting up 24 equations in the six unknown timing parameters, we can use linear least-squares to produce the approximate values shown in the following table.

T_{edge}	1325.06 usec
$T_{edge\ interpolant}$	173.67 usec
T_{span}	1864.34 usec
$T_{span\ interpolant}$	365.22 usec
$T_{fragment}$	354.89 usec
$T_{fragment\ interpolant}$	89.8 usec

Testing the resulting model against the original, experimentally-derived times, we get the following results:

run	model results	measured time	relative error
0	866.588074	863.270020	0.003844
1	1105.964600	1094.169922	0.010780
2	1168.406006	1179.789917	-0.009649
3	469.905151	476.690002	-0.014233
4	439.731140	437.709991	0.004618
5	561.302063	552.519958	0.015895
6	756.696899	774.729980	-0.023277
7	733.850098	716.460022	0.024272
8	1011.434509	971.179993	0.041449
9	977.113647	985.860046	-0.008872
10	1023.913330	1045.000000	-0.020179
11	1617.441528	1626.679932	-0.005679
12	1226.610840	1222.970093	0.002977
13	1589.858032	1583.390015	0.004085
14	1690.176880	1698.119995	-0.004678
15	699.653564	714.130005	-0.020271
16	663.042358	674.820007	-0.017453
17	867.306641	855.719971	0.013540
18	1142.042358	1134.829956	0.006355
19	1119.949707	1107.489990	0.011250
20	1574.667725	1587.250000	-0.007927
21	1495.985840	1503.270020	-0.004846
22	1585.730957	1578.929932	0.004307
23	1273.894653	1270.520020	0.002656

6.3.5. Implementation on a Custom Processor

These times can be greatly reduced in an implementation on a dedicated custom processor. Currently it is possible to build simple microcodable floating point processors with 50 ηsecond cycle time and large register files. In contrast, the following table shows approximate times (in microseconds) for the various floating-point operations on the VAX 11/780 on which the above statistics were acquired. These figures were produced by executing each operation one million times and subtracting the loop overhead, and are given in microseconds.

operation	microseconds
addition	3.46
subtraction	3.81
multiplication	3.63
division	7.22

Based on these speeds, a speedup of perhaps 20 to 50 times seems possible for a straightforward re-implementation of the algorithm on a dedicated microcodable processor.

Custom Hardware Currently, raster graphics systems capable of producing pixels at speeds on the order of 50 ηseconds per pixel [SwT,86] are becoming available. These systems use a scan-line oriented polygon filling algorithm: a Y-loop passes down the screen interpolating edge information from scan-line to scan-line; for each scan-line, an inner loop then iterates across the scan-line from one edge to the other, using linear interpolation to produce both depth values, for Z-buffer visibility, and red, green, and blue intensities for Gouraud shading.

These systems achieve their speed by using custom VLSI or gate-array technology to instantiate some or all of the polygon rendering process directly in hardware. Separate, and thus concurrent, hardware performs the Y-loop calculations, the scan-line setup, and the actual generation of pixels across the scan-line. By using parallel datapaths for each interpolant, each pixel update requires little more than a single addition time.

A similar approach should work for the Abram, Westover and Whitted algorithm. Now, however, we need *two* sets of parallel interpolant datapaths: one for the top contour of the sample-rectangle and one for the lower. Additionally, two further scan-line interpolants must be considered: the Y-offset of the top and bottom contours. The interpolation of these provides the set of interior edges for each pixel across the sample-rectangle. While such an implementation will likely incur more overhead than the simple point-sampling algorithm (for example, to produce the maximum and minimum of each surface interpolant within the sample squares for later texture mapping), the inner loop time should be reduced to a very few microseconds or less.

6.4. A Parallel Visible-Surface Renderer

As we have shown in section 6.3, a single processor executing the above rendering algorithm cannot achieve real-time performance. However, the analytic visibility algorithm developed in earlier chapters provides for independence between fragments; multiple parallel processors can thus be used to render different visible fragments simultaneously, using an accumulating image buffer to sum the contributions of the parallel renderers and form the final image. In this section a parallel architecture based on this property will be developed.

6.4.1. Parallel Rendering

A primary difficulty encountered in the design of parallel rendering systems that are based on frame-buffer image memory is the contention among the pixel producers for access to the memory. Each 512x512 frame contains 256K pixels; if the system is to operate at 30 frames per second, 7.5 million pixels must be determined and stored: the visible-surface (e.g. Z buffer) or anti-aliasing algorithm (e.g. super-sampling) may require many more. Using a 3x3 sample filter kernel, the algorithm described above generates at least 9 pixel contributions per output sample, requiring 70 to 100 million store operations per second!

How can a memory be structured to store so many pixels, particularly when the source of the pixels is a perhaps large set of pixel generators vying for their share of memory bandwidth? Fuchs, in [Fuc,77] and later in [FuJ,79] proposed that the pixels of the frame buffer memory be interlaced among several separate memory units (figure 16). Several processors, each closely coupled to one or more of the memory units, then share the process of rendering polygons. A broadcast controller transmits the definition of each polygon to every processor; each processor then scan-converts the polygon into the pixels under its control. In effect, each generates the image at a lower resolution.

Parke [Par,80] considered the division of the image plane into contiguous partitions, again each closely coupled to a polygon processor. A splitting tree was used to divide each input polygon among the memory partitions affected by it, and the associated processor renders the polygon into the pixels under its control. In this manner, the per-polygon and per-scanline overhead is incurred by only a fraction of the polygon processors which are instead operating on fragments lying in their own partition of the image plane.

Neither of these approaches seems appropriate for the present system. If parallel processors closely coupled to an interlaced memory are used, the dicing algorithm executing in each cannot

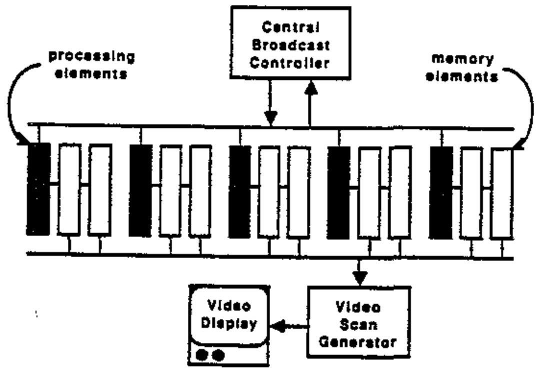


Figure 16. Fuch's Distributed Renderer

take advantage of the adjacency of successive sample rectangles during the vertical pass or of successive sample squares during the horizontal pass of the algorithm. The Parke algorithm depends on the uniform distribution of image complexity across the image plane to ensure that work load is evenly distributed among the scan-conversion processors, a very questionable assumption.

6.4.2. A Loosely-Coupled Architecture

The algorithms for visibility and rendering presented earlier make it possible to decouple the process of generating pixel contributions from analytic visible polygon fragments from the process of accumulating the contributions to form the image. Based on this division, we can envision a system in which a set of tiling processors, generating pixel contributions, are separated from a partitioned intelligent memory that performs the accumulation of pixel contributions locally by a communications network that transports pixel contributions from the tiler to the appropriate memory (figure 17).

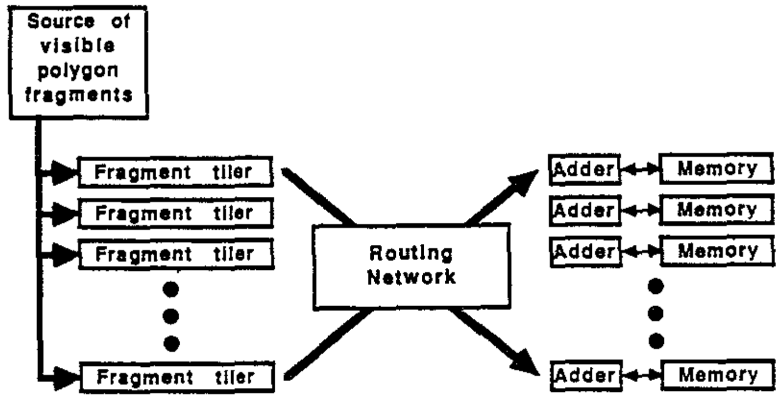


Figure 17. Architecture of a Parallel Renderer

Tilers

The generation of pixel contributions from analytic polygon definitions can again be broken down into three subcomputations: a "dicing" step which computes the intersection of the polygon with a given sample square, a shading step, in which the various surface characteristics and lighting parameters are used to determine the shade for the sample-square fragment, and finally a weighting step in which the shade is weighted according to the integrals determined from the table-lookup process. Note that these three steps can be easily pipelined as in figure 18.

Memory Components

In this architecture, the overall image memory will be partitioned into submemories, each associated with a very simple processor capable of accumumulating pixel contributions. In order to achieve balance (as in the Fuchs architecture above) the memories will be interlaced.

The Communications Network

The final, possibly most important component to consider is the communications network that carries pixel contributions from the tilers to the memories. It is easy to see that a single arbitrated bus cannot provide the 10 nanosecond cycle time necessary to carry 100 million pixel contributions

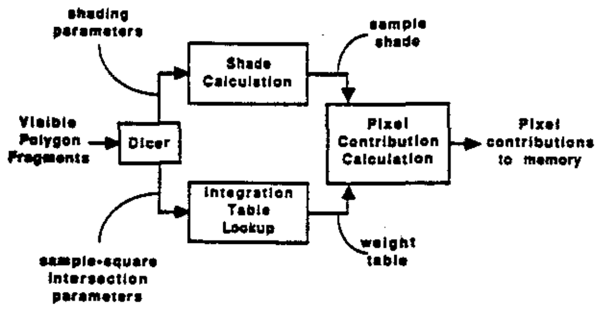

shading parameters
Visible Polygon Fragments
Dicer
Shade Calculation
Integration Table Lookup
sample-square intersection parameters
sample shade
Pixel Contribution Calculation
weight table
Pixel contributions to memory

Figure 18. Pipeline Stages of a Tiling Processor

per second from the generating tilers to the memories; instead, a parallel network carrying many contributions concurrently must be used.

One factor which greatly simplifies this problem is that data flows in only one direction through the communications component: data passes from the tilers to the memories, and never in the reverse direction. Since the tilers never need to wait for a response to memory requests, the time it takes any single message to pass from a tiler to the addressed memory can be traded off to minimize the total communications time. We therefore consider a multiple-stage switching network approach, in which each message must pass through several discrete stages between tiler and memory, but which can carry many messages from stage to stage concurrently.

One network topology that provides the necessary communication paths is the Omega (or rectangular SW Banyan) network [GoL,73, GGK,83, Law,75]. This network connects n inputs to n outputs via $log(n)$ stages, each consisting of a perfect shuffle interconnection followed by $\frac{n}{2}$ 2x2 switches (as in figure 19a). Each input message is routed to one of the two output ports based on the value of a single bit of the destination address.

Since a switch can receive a message at both inputs simutaneously but can only transfer one at a time, one message must pend at the input port of the switch. In order to avoid backing up the

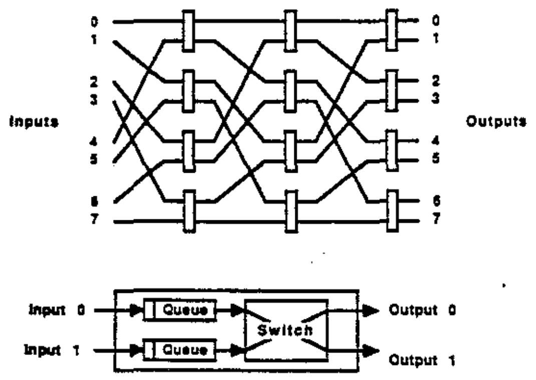


Figure 19. The 8 in, 8 out Omega Network
a. Topology
b. Architecture of a 2x2 Switch with Queues

network, a queue will be added to the input ports of the network (figure 19b). The switch is thus insulated from overload conditions; the network only backs up when a switch queue fills, and then one switch at a time backward through the network.

6.4.2.1. System Size

The architecture as introduced above contains several variables: the number and speed of tiling processors, the number and speed of memory units, the speed of the network switches, and the size of the load-balancing queues. These variables can be traded off in many ways; for example, we might use a few fast and expensive tilers or many slower cheaper processors; we also might trade off queue size in the switches for transmission speed.

A simulator of this architecture has been built to estimate performance of the architecture for a set of system parameters. Given the number and speed of various components and an input set of visible polygons, the simulator reports the time taken to generate the image and the proportional utilization of the hardware. Using this simulator, we can balance the system components to provide any desired level of performance. In order to do so, we need some estimates for the range in which these values can vary.

Tilers

As noted above, a 512x512 rendering system running at 30 frames per second requires the generation of about 100 million pixel contributions per second. Given a set of nine weights and a shade, the third stage must multiply each weight by the shade and produce the result to the routing network. Because the first two sections operate on the single sample-square fragment while the third must produce nine separate pixel contributions, we can assume that the time required for the component to handle a single sample-square fragment will be dominated by the time required to produce contributions to the nine output samples affected by the sample square fragment.

The fastest architecture for this process would use parallel hardware to perform the weighting multiplies for the red, green and blue channels concurrently. Since current widely-available hardware can perform such 8x8 multiplies in 70 to 100 nanoseconds, a lower bound on the time required to produce a weighted sample might be as little as 100 nanoseconds; using a single multiplier, the time might rise to about 300 nanoseconds. (Since nine such contributions are produced for every sample-square fragment, this will require that the dicing algorithm described above operate in from 1 to 3 microseconds per sample-square). Based on these numbers and on the estimated 100 million contributions per second required for real-time performance, we find that 16 fast or 32 slow processors running without significant delay would provide the necessary performance.

Memories

Again, the image memory for a real-time system must absorb as many as 90 to 100 million contributions per second, or about one every 10 nanoseconds. Using currently available technology (eg. 70 nanosecond RAMs and a 20 nanosecond adder) the read/add/store cycle can be done in about 160 nanoseconds; thus we need 16 memory partitions to provide the necessary performance.

The Communications Network

The primary factors determining the speed of the switching network are the size of the messages routed through the network, the width of the inter-processor data paths and whether they communicate within or between integrated circuits. These factors are interrelated: for example, the use of narrow data paths may make it possible to place the entire switching network on a single IC and thereby achieve a higher bandwidth between switches, but will require multiple cycles of the data path to transfer the message.

The size of the messages are determined by the data they must carry: a pixel address and a contribution to be added to the addressed pixel. The address, partitioned into a destination memory address and a pixel address within the addressed memory, requires 18 bits for a 512x512 image; 20 bits are required for a 1024x1024 image. The pixel contribution is a 24-bit word consisting of 8-bit contributions to the red, green and blue pixel values. Thus the message requires 44 bits.

6.4.3. Simulation Results

The rendering architecture has been run on the visible polygon fragments produced by the execution of the visiblity-processor simulator of chapter 5 on the six test images[4]. Each of these runs models a system of 32 tiling processors, producing pixel-contribution packets every 300 ηseconds, and 16 intelligent memory components, each capable of receiving and accumulating contribution packets in 150 ηseconds. The next table indicates the amount of work required to generate each test image measured in the number of contribution packets generated by the tilers and accumulated by the memories, followed by the time required to generate each, assuming no communications overhead and an even distribution work among the tilers.

[4] In several of the test images, a single visible polygon fragment comprised a large area of the screen; thus, whichever tiling processor was assigned that polygon became the system bottleneck. To avoid this situation, polygons are quartered until their bounding box covers no more that 4096 pixels. This process could take place in the processor that distributes visible polygon fragments to the fragment tilers.

image	number of packets	ideal image generation time(sec)
building scene 1	2,826,648	0.0265
building scene 2	2,736,216	0.0257
building scene 3	2,647,386	0.0248
aircraft scene 1	2,679,111	0.0251
aircraft scene 2	2,750,886	0.0258
ship scene	2,811,951	0.0264

The following tables indicate the performance of the simulated system for several sets of network parameters. The first table reports the system performance when network switches can transfer a packet from their input queue to the destination input queue in 150 ηseconds, and when each input queue contains space for 16 packets. Three values are reported: the estimated time required to generate the image under those assumtions, the average proportion of the image generation time that the tilers and memories were functional (e.g. not blocked by a clogged network), and the ratio of the estimated time to the minimum time reported in the last table.

image	estimated execution time (sec.)	average utilization proportion	performance ratio
building scene 1	0.067330	0.393582	2.540768
building scene 2	0.063118	0.406414	2.460543
building scene 3	0.063023	0.393812	2.539284
aircraft scene 1	0.063331	0.396594	2.521469
aircraft scene 2	0.065141	0.395907	2.525848
ship scene	0.069798	0.377689	2.647682

150 ηsecond switches, 16-packet queues

Under these assumptions of network performance, this architecture can produce only about 15 frames per second, utilizing only about 40% of the available performance of the tilers and memory units. These figures indicate that the communications network is a significant system bottleneck. In order to improve these statistics, we first try enlarging the input queues between the stages of the network, hoping that longer queues will balance the load better. The next table indicates the estimated performance when the switch queues can store 64 packets, rather than the 16 in the above simulation.

image	estimated execution time	average utilization	performance proportion
building scene 1	0.062096	0.426755	2.343265
building scene 2	0.057455	0.446473	2.239776
building scene 3	0.057134	0.434406	2.301996
aircraft scene 1	0.056432	0.445080	2.246787
aircraft scene 2	0.058985	0.437226	2.287147
ship scene	0.062319	0.423020	2.363956

150 ηsecond switches, 64-packet queues

The addition of more memory for input queues does not substantially affect the speed of the system. Instead, we consider the effects of doubling the speed of the switches. The next table indicates the estimated performance when the switches can transfer packets in 75 ηseconds.

image	estimated execution time	average utilization	performance proportion
building scene 1	0.035209	0.752652	1.328635
building scene 2	0.032181	0.797110	1.254533
building scene 3	0.033723	0.735980	1.358732
aircraft scene 1	0.033634	0.746760	1.339119
aircraft scene 2	0.035157	0.733548	1.363238
ship scene	0.036939	0.713664	1.401219

75 ηsecond switches, 16-packet queues

With the network operating at 75 ηseconds, the system achieves real-time, 30 frames per second performance on the test images. For completeness, the last table shows the estimated performance for a system with 75 ηsecond switches and 64-packet queues. Finally, chart 1 compares these results graphically.

image	estimated execution time	average utilization	performance proportion
building scene 1	0.035209	0.752652	1.328635
building scene 2	0.032181	0.797110	1.254533
building scene 3	0.033723	0.735980	1.358732
aircraft scene 1	0.030591	0.821048	1.217956
aircraft scene 2	0.032002	0.805876	1.240886
ship scene	0.033979	0.775836	1.288933

75 ηsecond switches, 64-packet queues

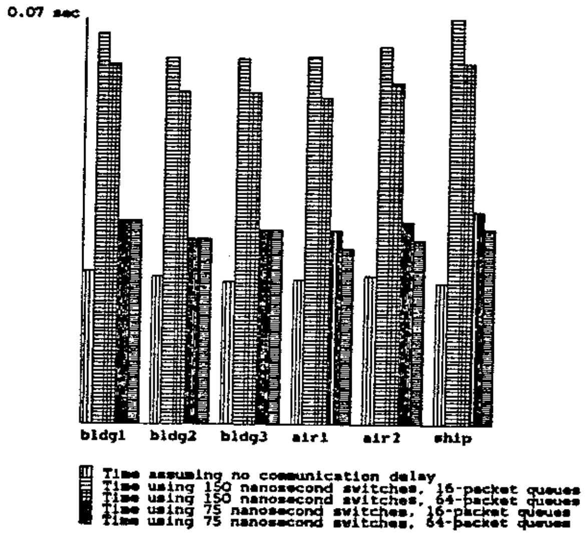


Chart 1: Comparing image generation times for various networks

Again, we find that the extention of the input queues does not offer a significant improvement in performance.

An Improvement From the above simulations, it is clear that the network traffic presents a difficult problem: handling the necessary data in real time in this architecture requires the use of very fast switches. In order to achieve such switch speeds, it seems necessary to provide broad parallel datapaths between switching elements, minimizing the number of switches that can be placed on a single chip and raising the cost of inter-chip connections.

Alternatively, the network traffic can be cut to a small fraction of that of the current architecture by adding a pixel cache [SwT,86] to the tiling processors. By allocating enough memory to buffer the pixels covered by a fragment, the contributions to the pixels covered by the fragment can be accumulated locally to the tiling processor; thus, contributions of a polygon to a given pixel are only transferred through the network once, rather than the nine times (representing the effects of the intersection of the polygon with each of the 3x3 sample squares in the neighborhood of the pixel) required by the above architecture. Since the visible-polygon rendering algorithm operates in a scan-line fashion, each tiler requires space for only three scan lines of pixels (3 scan lines of 512 pixels at 3 bytes per pixel, or 4,608 bytes of memory) of local buffer space.

7

Summary, Conclusions and Future Work

This research has shown that analytic computation of visibility can have significant advantages in both the speed and quality of image generation. When visibility is known analytically the low-pass filtering necessary for anti-aliasing can be done to arbitrary accuracy, and the time-consuming rendering task can be distributed over many cooperating processors for real-time performance. Algorithms for computing analytic visibility and for rendering filtered images from the resulting visible fragments have been presented, and architectures for their parallel execution have been designed and simulated. The results show that analytic visibility can serve as a basis for the real-time rendering of realistic scenery.

Many avenues remain for continuing this research. These include extensions of the polygon-clipper visibility algorithm, refinements to the visibility and rendering architectures, and the development of a complete image generation system based on the visibility and rendering components developed in this dissertation. In this final chapter, a few of the more interesting possibilities are discussed.

7.1. Extensions to the Polygon Clipper Algorithm

While the polygon clipper algorithm as described in chapter 4 serves as an adequate basis for the real-time generation of anti-aliased imagery, it can be improved both in terms of the quality of the images it can support and the efficiency with which it operates. In this section some possible improvements in these areas are presented.

Shadow Effects It should be possible to add shadow effects to the images that can be created by the polygon clipper algorithm. Crow, in [Crow, 1977], shows how shadows can be computed by the use of shadow volumes, the regions of space that lie in the shadow of each surface in the scene. Each shadow volume is defined by a set of polygons in the plane defined by the light source and the edges of the surface casting the shadow, with one edge coincident with the surface edge and two edges defined by the rays from the light source through each endpoint of the surface edge (see figure 1). A visible surface is in shadow if it lies behind an odd number of shadow polygons.

Suppose that the shadow polygons of a scene are added to the end of the initial polygon list. When all the regions exterior to the silhouette of a polygon fragment have been removed, the last inside list consists of all the fragments (both real surfaces and shadow polygons) lying inside the volume defined by the first polygon in the list and the viewpoint. This list is then clipped against the plane of the first polygon. If no real polygons are found to lie in front, the first polygon is visible and the shadow polygons lying in front define the parts of that polygon which lie in shadow. Otherwise, the clip polygon is placed at the end of the list of real polygons (preceding the list of shadow polygons) and the process continues.

Using Surface Coherence The cost of the polygon clipper algorithm rises with the number of times that a list must be divided along a splitting plane. The algorithm of chapter 4 requires that every edge in a scene defines a splitting plane. In many cases, however, continuous surfaces are

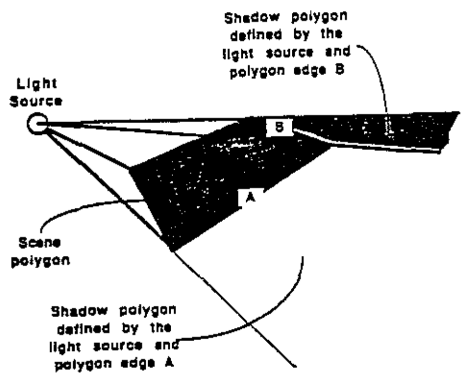

Shadow polygon defined by the light source and polygon edge B
Light Source
B
A
Scene polygon
Shadow polygon defined by the light source and polygon edge A

Figure 1. Shadow Polygons

defined by a mesh of polygons. When such surfaces are convex, the different polygons in the mesh do not overlap; thus it is unnecessary to split the image plane along edges interior to convex mesh surfaces. Instead, a correct result can be produced by considering only the edges of the *silhouette* of the convex mesh surface. If such surfaces are identified, the amount of work required by the polygon clipper should drop significantly.

7.2. Further Research in Parallel Analytic Visibility

The architectures for the visibility and rendering processors described in chapters 5 and 6 are very simplified, useful for simulation and proving the feasibility of the approach, but not as the basis for a real machine design. Significant improvements to each were offered in the last sections of each those chapters, and should be investigated. A further possibility worth investigating is the replacement of the pipeline visibility architecture with a different approach.

Using Intra-List Parallelism The visibility architecture of chapter 5 utilizes *inter-list* parallelism: since polygon lists consist of polygons lying in non-overlapping regions of the screen, they can be handled independently and hence, concurrently. It also might be possible to exploit the independence of operations *within* a list. Each polygon in a list is clipped against a single edge; conceivably, these clips can be carried out simultaneously: the polygons of a list could be distributed among many parallel processors, the splitting plane broadcast to all, and the list divided in a single split time (or a few, if there are fewer splitters than polygons in the list).

This approach might make it possible to use many more processors to execute the algorithm. In chapter 6 we saw that the performance of the pipeline architecture was limited by the fact that the longest list must be handled by a single processor; thus, the length of the longest list defined the minimum time required by any number of processors running in a strictly inter-list parallel manner. In a intra-list parallel system using a large number of splitter processors, the longest list can be handled in a very few splitting times. Several shorter lists might be placed different parts of the splitter array and operated on simultaneously.

Although this approach presents many difficulties, it offers substantial advantages in terms of the number of processors that might be brought to bear on the problem.

7.3. A Complete Image Generation System

The visibility and rendering processors form only the rendering component of a complete image generation system: given a prepared set of polygons, they will render a high quality image in real time. Preparing that set of polygons requires two further components: a database manager, responsible for selecting the polygons to be displayed and a geometry processor, which will prepare the polygon list for the rendering components. The development of a complete system providing the necessary performance on the preparatory stages of image generation remains open. In this section, one possible architecture of a complete system is roughly sketched (figure 2).

The four stages of this system are separated by double buffers to form a pipeline so that, at a single instant, the four stages are operating on the data of four sequential frames. In this manner, frames are produced at the rate of the slowest of the four components, rather than the total pipeline

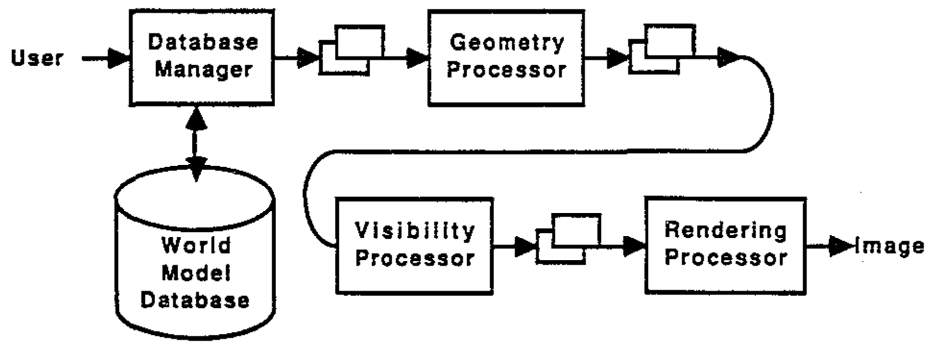


Figure 2. An Overall System Architecture

time (although it still requires an entire pipeline time for the effects of a viewer input to be seen on the screen).

The Database Manager Given a specification of a scene (such as a point of view and the orientation of the viewer), the database manager must supply a set of polygons which might be in the scene. Since the performance of the image generation system depends largely on the size of its input list, the database manager must attempt to minimize the number of polygons in that list. Many standard techniques may be used. Simple methods exist for structuring a database to make it easy to determine parts of the world model which do not intersect the viewing frustum and can be eliminated. Objects may be defined at several levels of detail appropriate for viewing at different ranges from the viewer.

As the database manager selects polygons for display, they are placed in an output buffer. When complete, this buffer and the specification of the view are passed to the next stage, the geometry processor.

The Geometry Processor Following the database manager in the pipeline is a geometry processor. The geometry processor must take the display list produced by the database manager and the viewing information specified by the user and produce a set of polygons prepared for input to the visibility process. This involves three tasks.

First, the polygon data received from the database processor must be transformed to the eye coordinate system. This transformation, appearing in virtually all graphics systems, involves the maintenance of a matrix stack, processing the matrix commands described above, and the transformation of coordinate points by the topmost matrix on the stack.

Given the transformed vertices of a polygon in the eye coordinate system, the splitting planes for that polygon must be determined. The geometry processor calculates both the plane of the polygon and the dividing planes defined by the edges of the polygons and the viewpoint.

Finally, the polygons must be sorted into a rough priority order. In the system used for the simulation, a simple bucket sort based on the centroid of the polygons proved to be an effective method. This bucket sort can be simply implemented by adding a small amount of intelligence to the buffers that accept the output of the geometry processor and pass it on to the subsequent visibility processor.

The Visibility Processor The visibility processor is an extension of the design of chapter 5 (see figure 4). Initially, the pipeline input comes from the output buffer of the geometry processor; when this source is exhausted, the queue receiving the output of the final stage of the pipeline is substituted. The visible-polygon output port of each pipeline processor is connected to a bus carrying the visible fragments to a buffer to await rendering. Assuming an average of 50 32-bit words per visible polygon fragment, a 100 ηsecond 32-bit bus requires 5 μseconds per visible polygon fragment, or 6000 visible polygon fragments at 30 frames per second.

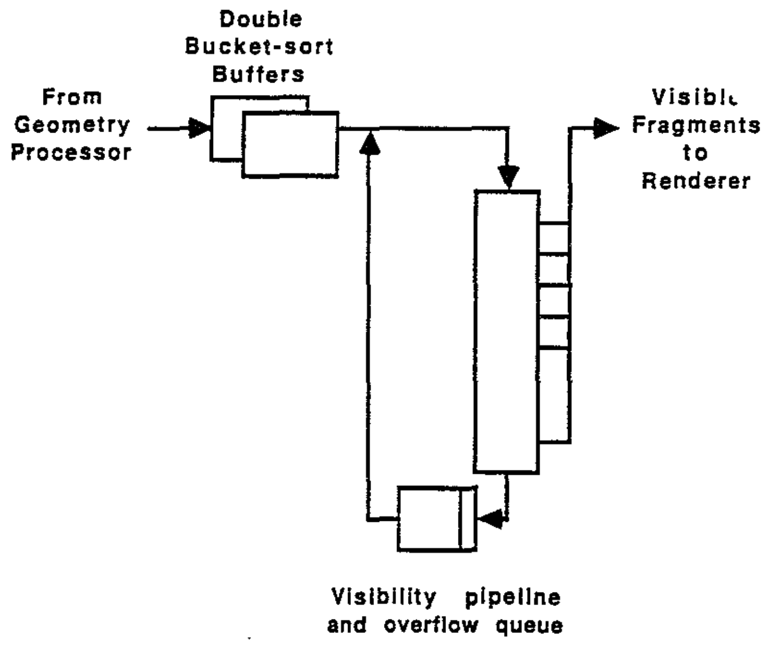

Double Bucket-sort Buffers
From Geometry Processor
Visible Fragments to Renderer
Visibility pipeline and overflow queue

Figure 4. The Visibility Component Architecture

The Rendering Processor The rendering processor (figure 5) consists of a controller which extracts polygon fragments from the visibility processor output buffer and visible polygon fragments and distributes them to the renderers. A single bus may be used to transfer the visible polygon fragments to the tiling processors; as above, a single 100 ηsecond bus suffices to transfer 6000 visible polygon fragments to the tilers at 30 frames per second. This controller may subdivide disproportionately large visible polygon fragments for balance among the tilers.

Appendix 1: Bibliography

1. G. Abram, L. Westover and T. Whitted (1985), "Efficient Alias-free Rendering Using Bit-masks and Look-up Tables", *ACM Computer Graphics*, Vol. 19 No. 3, 53-59.

2. G. Abram and H. Fuchs (1985) "VLSI Architectures for Computer Graphics" in *Microarchitecture of VLSI Computers*, P. Antognetti, F. Anceau and J. Vuillemin, eds., Martinus Nijhoff Publishers, Dordrecht, 172-188.

3. J. Amanatides (1984), "Ray Tracing with Cones", *ACM Computer Graphics*, Vol. 18 No. 3, 129-135.

4. G.A. Anderson and E.D. Jensen (1975), "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", *Computer Surveys*, Vol. 7, No. 4, 197-213.

5. A. Appel (1967), "The Notion of Quantitative Visibility and the machine Rendering of Solids", *Proceedings of teh ACM National Conference*, 387-393.

6. J. Backus (1978), "Can Programming Be Liberated from the von Neumann Style?", *Communications of the ACM*, Vol. 21, No. 8, 613-641.

7. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick and R.A. Stokas (1968), "The Illiac-IV Computer", *IEEE Transactions on Computers*, Vol. C17 No. 8, 746-757.

8. B.A. Barsky (1981), "The Beta-Spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures", *Ph.D. Dissertation*, Department of Computer Science, the University of Utah.

9. A. Bechtolsheim and F. Baskett (1980), "High-Performance Raster Graphics for Microcomputer Systems", *ACM Computer Graphics*, Vol. 13, No. 2, 43-47.

10. J. Blinn and M. Newell (1976), "Texture and Reflections in Computer Generated Images", *Communications of the ACM*, Vol. 19, No. 10, 542-547.

11. J. Blinn (1977), "Models of Light Reflection for Computer Synthesized Images", *ACM Computer Graphics*, Vol. 11, No. 2, 192-199.

12. J. Blinn (1978), "Simulation of Wrinkled Surfaces", *ACM Computer Graphics*, Vol. 12, No. 3, 286-292.

13. S.A. Browning (1980), "A Tree Machine", *Lambda*, 2nd. quarter, 31-36.

14. K.R. Castleman (1979), *Digital Image Processing*, Prentice-Hall, Englewood Cliffs, N.J..

15. E. Catmull (1974), "A Subdivision Algorithm for Computer Display of Curved Surfaces", *Ph.D. Dissertation*, Department of Computer Science, the University of Utah.

16. E. Catmull (1978), "A Hidden Surface Algorithm with Anti-Aliasing", *ACM Computer Graphics*, Vol. 12, No. 3, 6-11.

17. E. Catmull and A.R. Smith (1980), "3-D Transformations of Images in Scanline Order", *ACM Computer Graphics*, Vol. 14 No. 3, 279-285.

18. E. Catmull (1984), "An Analytic Visible Surface Algorithm for Independent Pixel Processing", *ACM Computer Graphics*, Vol. 18 No. 3, 109-11.

19. P. Chu and B. New (1984), "Microprogrammable Chips Blend Top Performance With 32-Bit Structures", *Electronic Design*, November 15, 229-262.

20. J. Clark (1980), "Structuring a VLSI System Architecture", *Lambda*, 2nd. Quarter, 25-30.

21. J. Clark and M. Hannah (1980), "Distributed Processing in a High-Performance Smart Image Memory", *Lambda*, Vol.I, No. 3, 4th. Quarter, 40-45.

22. J. Clark (1982), "The Geometry Engine: A VLSI Geometry System for Graphics", *ACM Computer Graphics*, Vol. 15, No. 3, 127-133.

23. D. Cohen and S. Demetrescu (1980), Presentation at ACM Siggraph'80 Panel on Trends on High Performance Graphic Systems,

24. M. Cohen and D. Greenberg (1985), "A Radiosity Solution for Complex Environments", *ACM Computer Graphics*, Vol. 19, No. 4, 31-40.

25. M. Cohen, D. Greenberg, D. Immel and P. Brock (1986), "An Efficient Radiosity Approach for Realistic Image Synthesis", *IEEE Computer Graphics and Applications*, Vol. 6, No. 6,

26. R.L. Cook, T. Porter and L. Carpenter (1984), "Distributed Ray Tracing", *ACM Computer Graphics*, Vol. 18 No. 3, 137-145.

27. S.A. Coons (1967), "Surfaces for Computer-Aided Design of Space Forms", *Technical Report*, MAC-TR-41, M.I.T. .

28. F. Crow (1976), "The Aliasing Problem in Computer Synthesized Shaded Images", *Technical Report*, UTEC-CSc-76-015, Univ. of Utah.

29. F. Crow (1977), "The Aliasing Problem in Computer-Generated Shaded Images", *Communications of the ACM*, Vol. 20, No. 11, 799-805.

30. F. Crow (1981), "A Comparison of Antialiasing Techniques", *IEEE Computer Graphics and Applications*, Vol. 1, No. 4, 40-49.

31. F. Crow and M. Howard (1981), "A Frame Buffer System With Enhanced Functionality", *ACM Computer Graphics*, Vol 15, No. 3, 63-69.

32. F. Crow (1984), "Summed-Area Tables for Texture Mapping", *ACM Computer Graphics*, Vol. 18 No. 3, 207-212.

33. J. Eyles (1986), Personal Communication,

34. E. Feibush, M. Levoy and R. Cook (1980), "Synthetic Texturing Using Digital Filters", *ACM Computer Graphics*, Vol. 14, No. 3, 294-301.

35. A.R. Forrest (1972), "On Coons and Other Methods for the Representation of Curved Surfaces", *Computer Graphics and Image Processing*, Vol. 1, 341-359.

36. H. Fuchs (1977), "Distributing a Visible Surface Algorithm Over Multiple Processors", *Proc. ACM National Conference*, 449-451.

37. H. Fuchs and B.W. Johnson (1979) "An Expandable Multiprocessor Architecture for Video Graphics" in *Proc. Sixth Annual Symposium on Computer Architecture*, 58-67.

38. H. Fuchs, Z. Kedem and B. Naylor (1979), "Predetermining Visibility Priority in 3-D Scenes", *ACM Computer Graphics*, Vol. 13, No. 2, 175-182.

39. H. Fuchs and Z. Kedem (1980), "On Visible Surface Generation by a Priori Tree Structures", *ACM Computer Graphics*, Vol. 14, No. 3, 124-133.

40. H. Fuchs and J. Poulton (1981), "PIXEL-PLANES: A VLSI-Oriented Design for a Raster Graphics Engine", *VLSI Design*, Vol.II, No. 3, 3rd. Quarter, 20-32.

41. H. Fuchs, J. Poulton, A. Paeth and A. Bell (1982) "Developing Pixel Planes, A Smart Memory-Based Raster Graphics System" in *Proc. MIT Conference On Advanced Research in VLSI*, 137-146.

42. H. Fuchs, G. Abram and E. Grant (1983), "Near Real-Time Shaded Display of Rigid Objects", *ACM Computer Graphics*, Vol. 17, No. 3, 65-72.

43. A. Fujimoto, C. Perrott and K. Iwata (1984), "A 3-D Graphics Display System with Depth Buffer and Pipeline Processor", *IEEE Computer Graphics and Applications*, Vol. 4, No. 6, 11-23.

44. R. Galimberti and U. Montanari (1969), "An Algorithm for Hidden-Line Elimination", *Communications of the ACM*, Vol. 12, No. 4, 206.

45. A. Glassner (1986), "Adaptive Precision in Texture Mapping", *ACM Computer Graphics*, Vol. 20, No. 3, (to appear).

46. L.R. Goke and G.J. Lipovsky (1973), "Banyan Networks for Partitioning Multiple Processor Systems", *Proceedings of the 1st Annual Symposium on Computer Architecture*, 21-28.

47. C. Goral, K. Torrance, D. Greenberg and B. Battaile (1984), "Modeling the Interaction of Light Between Diffuse Surfaces", *ACM Computer Graphics*, Vol. 18, No. 4, 213-222.

48. A. Gottlieb and J.T. Schwartz (1982), "Networks and Algorithms for Very-Large-Scale Parallel Computation", *Computer*, Vol. 15, 27-36.

49. A. Gottlieb, R. Grisman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir (1983), "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", *IEEE Transactions on Computers*, Vol. C32 No. 2, 175-189.

50. H. Gouraud (1971), "Computer Display of Curved Surfaces", *IEEE Transactions on Computers*, Vol. C-20 No. 6, 623.

51. S. Gupta and R. Sproull (1981), "A VLSI Architecture for Updating Raster-Scan Displays", *ACM Computer Graphics*, Vol. 15, No. 3, 71-78.

52. R.A. Hall and D. Greenberg (1983), "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics and Applications*, Vol. 3, No. 8, 10-20.

53. S. Haruyama and B. Barsky (1984), "Using Stochastic Modelling for Texture Generation", *IEEE Computer Graphics and Applications*, Vol. 4, No. 3, 7-21.

54. C.A.R. Hoare (1978), "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, 666-677.

55. T. Ikedo (1984), "High-Speed Techniques for a 3-D Color Graphics Terminal", *IEEE Computer Graphics and Applications*, Vol. 4, No. 5, 46-58.

56. D. Immel, M. Cohen and D. Greenberg (1986), "A Radiosity Method for Non-Diffuse Environments", *ACM Computer Graphics*, Vol. 20, No. 4, 133-142.

57. A.K. Jones, R.J. Chansler, I. Durham, P. Feiler and K. Schwans (1977), "The Implementation of the Cm* Multi-Microprocessor", *AFIPS (Proceedings of the National Computer Conference)*, Vol. 46, 657-663.

58. M. Kaplan and D. Greenberg (1979), "Parallel Processing Techniques for Hidden Surface Removal", *ACM Computer Graphics*, Vol. 13, No. 2, 300-307.

59. A. Kay (1977), "Microelectronics and the Personal Computer", *Scientific American*, Vol. 237 No. 3, 230-244.

60. D. Kay (1979), "Transparency, Refraction and Ray-Tracing for Computer Synthesized Images", Master's Thesis, Cornell University, Ithaca N.Y., .

61. D. Kay and D. Greenberg (1979), "Transparency for Computer Synthesized Images", *ACM Computer Graphics*, Vol. 13, No. 2, 158-164.

62. G. Kedem and J. Ellis (1984), "Computer Structures for Curve-Solid Classification in Geometric Modelling", Technical Report TR137, Department of

Computer Science, University of Rochester, .

63. D.J. Kuck (1968), "Illiac-IV Software and Applications Programming", *IEEE Transactions on Computers*, Vol. C17 No. 8, 757-770.

64. H.T. Kung (1982), Parallel Processing Systems,

65. J.M. Lane, L.C. Carpenter, J.T. Whitted and J.F. Blinn (1980), "Scan Line Mathods for Displaying Parametrically Defined Surfaces", *Communicationh of the ACM*, Vol. 23 No. 1, 23-34.

66. D. Lawrie (1975), "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers*, Vol. C-24, 115-1155.

67. P.P. Loutrel (1970), "A Solution to the Hidden-Line problem for Computer-Drawn Polyhedra", *IEEE Transactions on Computers*, Vol. C-19, No. 3, 205.

68. Mezei, Puzin and Conroy (1974) "Simulation of Patterns of Nature by Computer Graphics" in *Information Processing '74*, North-Holland, London, 861-865.

69. W. Myers (1984), "Staking Out the Graphics Display Pipeline", *IEEE Computer Graphics and Applications*, Vol. 4, No. 7, 60-65.

70. M.E. Newell, R.G. Newell and T.L. Sancha (1972), "A New Approach to the Shaded Picture Problem", *Proc. ACM National Conference*, 443.

71. W.M. Newman and R.F. Sproull (1979), *Principles of Interactive Computer Graphics (2nd. ed.)*, McGraw-Hill, Englewood Cliffs, N.J., New York.

72. A. Norton, A. Rockwood and P. Skolmoski (1982), "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space", *ACM Computer Graphics*, Vol. 16, No. 3, 1-9.

73. H. Nyquist (1924), "Certain Factors Affecting Telegraph Speeds", *Bell Systems Technical Journal*, Vol. 3, 324-346.

74. H. Nyquist (1928), "Certain Topics in Telegraph Transmission Theory", *AIEE Transactions*, 617.

75. A.V. Oppenheim and R.W. Shafer (1975), *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., New York, Englewood, N.J..

76. F.I. Parke (1980), "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems", *ACM Computer Graphics*, Vol. 14, No. 3, 48-56.

77. B.T. Phong (1975), "Illumination for Computer Generated Pictures", *Communications of the ACM*, Vol. 18, No. 6, 311-317.

78. R. Pike (1984), Presentation at University of North Carolina at Chapel Hill,

79. R. Pinkham, M. Novak and K. Guttag (1983), "Video RAM Excels At Fast Graphics", *Electronic Design*, July 21, 161-172.

80. A.A.G. Requicha and H.B.Voelcker (1977), "Geometric Modelling of Mechanical Parts and Processes", *IEEE Computer*, Vol. 10, No. 12, 48-52.

81. A.A.G. Requicha and H.B.Voelcker (1977), "Constructive Solid Geometry", *Technical Memo*, #25, Production Automation Project, Univ. of Rochester.

82. R.F. Riesenfeld (1975), "Aspects of Modelling in Computer-Aided Geometric Design", *AFIPS (Proceedings of the National Computer Conference)*, Vol. 44, 597-602.

83. S. Rubin and T. Whitted (1980), "A 3-Dimensional Representation for Fast Rendering of Complex Surfaces", *ACM Computer Graphics*, Vol. 14, No. 3, 110-116.

84. J.A. Rudolph, L.C. Fulmer and W.C. Meilander (Feb. 15, 1971), "The Coming of Age of the Associative Processor", *Electronics*, Vol. 44 No. 4, 91-96.

85. J.A. Rudolph (1972), "A Production Implementation of an Associative Array Processor: STARAN", *AFIPS, Proceedings of the Fall Joint Computer Conference*, Vol. 41, 229-241.

86. B. Schachter and N. Ahuja (1979), "Random Pattern Generation Processes", *Computer Graphics and Image Processing*, Vol. 10, 95-114.

87. B. Schachter (1980), "Long Crested Wave Models", *Computer Graphics and Image Processing*, Vol. 12, 187-201.

88. B. Schachter (1981), "Computer Image Generation for Flight Simulation", *IEEE Computer Graphics and Applications*, Vol. 1, No. 4., 29-64.

89. B. Schachter (1983), *Computer Image Generation*, Wiley-Interscience, Englewood Cliffs, N.J., New York, Englewood, N.J., New York.

90. R.A. Schumacker, B. Brand, M. Gilliland and W. Sharp (1969), "Study for Applying Computer-Generated Images to Visual Simulation", *AFHRL-TR-69-14*, U.S. Air Force Human Resources Laboratory.

91. R.A. Schumacker (1980), "A New Visual System Architecture", *IITEC Conference Proceedings*, 2nd, 1-8.

92. J.T. Schwartz (1980), "Ultracomputers", *ACM Transactions on Parallel Languages and Systems*, 484-521.

93. D. Schweitzer (1983), "Artificial Texture: An Aid to Surface Visualization", *ACM Computer Graphics*, Vol. 17, No. 3, 23-30.

94. S. Sechrest and D. Greenberg (1981), "A Visible Polygon Reconstruction Algorithm", *ACM Computer Graphics*, Vol. 15, No. 3, 17-27.

95. C. Seitz (Jan. 1982) "Ensemble Architectures for VLSI - A Survey and Taxonomy" in *Proceedings of the Conference on Advanced Research in VLSI*, P. Penfield, ed. 130-135.

96. C.H. Sequin and P.R. Wensley (May, 1985), "Visible Feature Return at Object Resolution", *IEEE Computer Graphics and Applications*, Vol. 5, No 5, 37-50.

97. C. Shannon (1949), "Communication in the Presence of Noise", *Proc. Institute of Radio Engineers*, Vol. 37, No. 1, 10-21.

98. D.L. Slotnick, W.C. Borck and R.C. McReynolds (1962), "The SOLOMON Computer", *Proceedings of the Fall Joint Computer Conference*, Vol. 22, 97-107.

99. D.L. Slotnick (1971), "The Fastest Computer", *Scientific American*, Vol. 224, No. 2, 76-87.

100. L. Snyder (1981) "Overview of the CHiP Computer" in *VLSI 81: Very Large Scale Integration*, J.P. Cray, ed. Academic Press, 237-246.

101. R. Sproull, I. Sutherland, A. Thompson, S. Gupta and C. Minter (1983), "The 8 x 8 Display", *ACM Transactions on Graphics*, Vol. 2, No. 1, 32-56.

102. I. Sutherland (1965), "The Ultimate Display", *Proceedings of the IFIP Congress*, Vol. 2, 506-508.

103. I. Sutherland and G. Hodgman (1974), "Computer Graphics Clipping System For Polygons", *U.S. Patent 3,816,726*,

104. I. Sutherland, R. Sproull and R.A. Schumacker (1974), "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, Vol. 6, No. 1, 7-45.

105. I. Sutherland (1975), "System Of Polygon Sorting By Dissection", *U.S. Patent 3,889,107*,

106. R.J. Swan, A. Bechtolsheim, K. Lai and J.K. Ousterhout (1977), "The Implementation of the Cm* Multi-Microprocessor", *AFIPS (Proceedings of the National Computer Conference)*, Vol. 46, 645-655.

107. R.J. Swan, S.H. Fuller and D.P. Siewiorek (1977), "Cm*: A Modular Multi-Processor", *AFIPS (Proceedings of the National Computer Conference)*, Vol. 46, 637-644.

108. R. Swanson and L. Thayer (1986), "A Fast Shaded-Polygon Renderer", *ACM Computer Graphics*, Vol. 20, No. 4, 95-102.

109. C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull and D.R. Boggs (1982) "ALTO: A Personal Computer" in *Computer Structures: Principles and Examples*, Siewiorek, D.P., C. Gordon Bell, and A. Newell, ed. McGraw-Hill, 549-572.

110. L.G. Valiant (1981), "Universality Constraints in VLSI Circuits", *IEEE Transactions on Computers*, Vol. c-30, No. 2, 135-140.

111. J. Warnock (1970), "A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures", *Technical Report*, UTEC-CSc-70-101, Univ. Utah Computer Science Department .

112. K. Weiler and P. Atherton (1977), "Hidden Surface Removal Using Polygon Area Sorting", *ACM Computer Graphics*, Vol. 11, No. 2, 214-222.

113. R. Weinberg (1981), "Parallel Processing Image Synthesis and Anti-Aliasing", *ACM Computer Graphics*, Vol. 15, No. 3, 55-62.

114. D. Whelan (1982), "A Rectangular Area Filling Display System Architecture", *ACM Computer Graphics*, Vol. 16, No. 3, 147-153.

115. T. Whitted (1980), "An Improved Illumination Model for Shaded Display ", *Communications of the ACM*, Vol. 20, No. 6, 343-349.

116. T. Whitted (1981), "Hardware Enhanced 3-D Raster Display System", *Proc. Seventh Man-Computer Communications Conference*, 349-356.

117. M.C. Whitton (1984), "Memory Design for Raster Graphics Displays", *IEEE Computer Graphics and Applications*, Vol. 4, No. 3, 48-66.

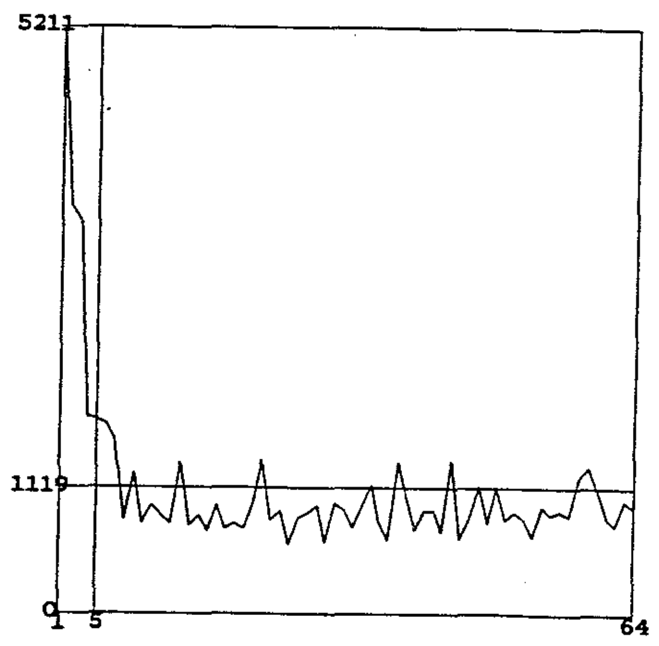
118. L. Williams (1983), "Pyramidal Parametrics", *ACM Computer Graphics*, Vol. 17, No. 3, 1-11.

119. R. Williamson and P. Rickert (1983), "Dedicated Processor Shrinks Graphics Systems to Three Chips", *Electronic Design*, August 4, 143-148.

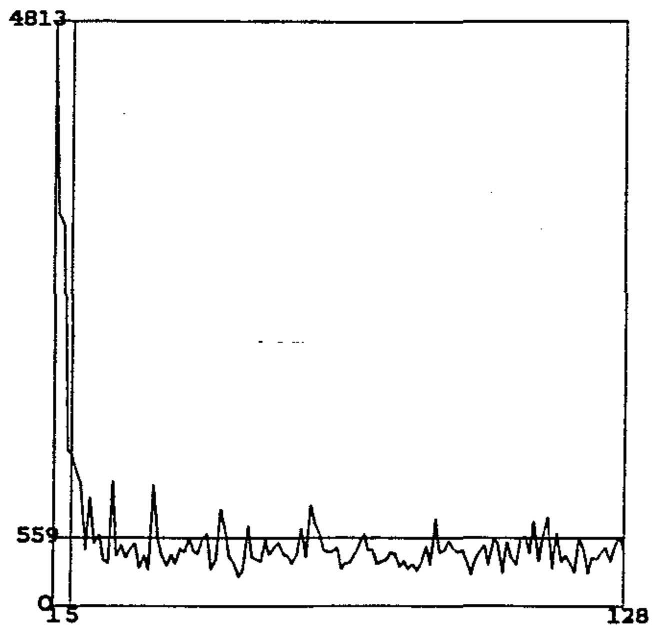
120. C. Wu and T. Feng (1978), "Routing Techniques For A Class Of Multistage Interconnection Networks", *Proceedings of the 1978 International Conference on Parallel Processing*, 197-205.

Appendix 2: Random List Distribution in Test Cases

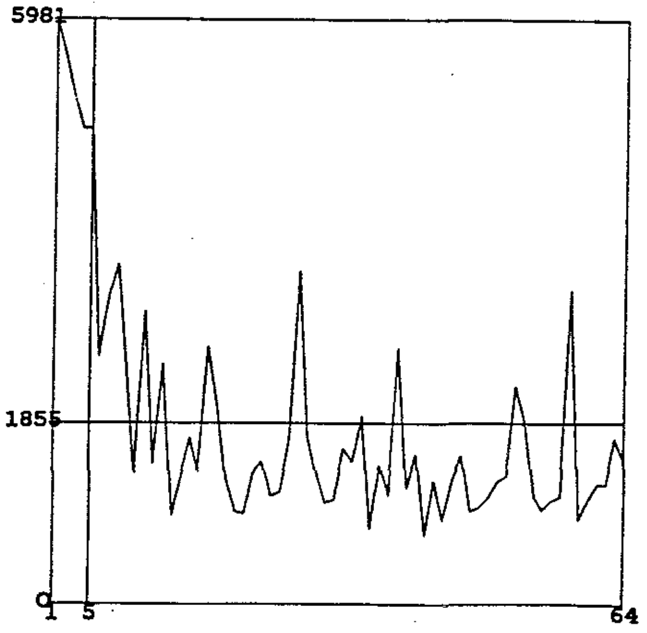
The following graphs illustrate the number of polygons assigned to each processor when lists are randomly distributed among the available processors. The vertical line divides the initial five processors that perform the viewing frustum clip from those that perform only visibility clips. The horizontal line indicates the ideal distribution of polygons: each of *n* processors receives 1/*n* of the total number of polygons.



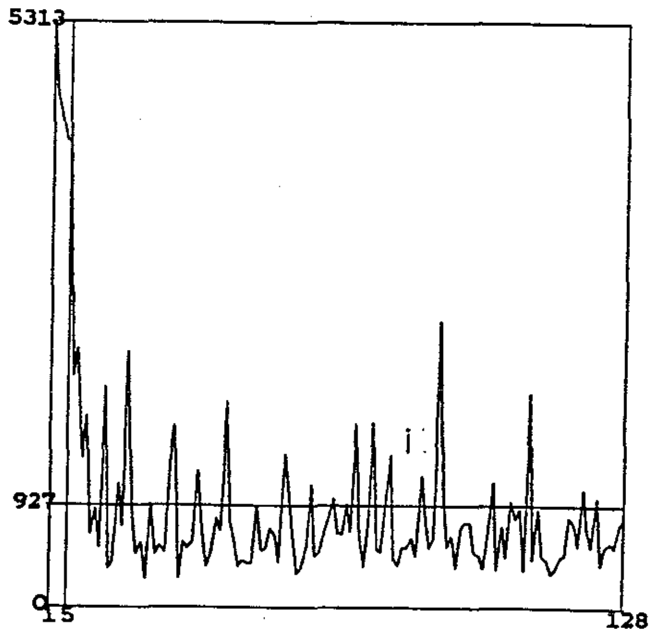
Building Scene 1: 64 Processors



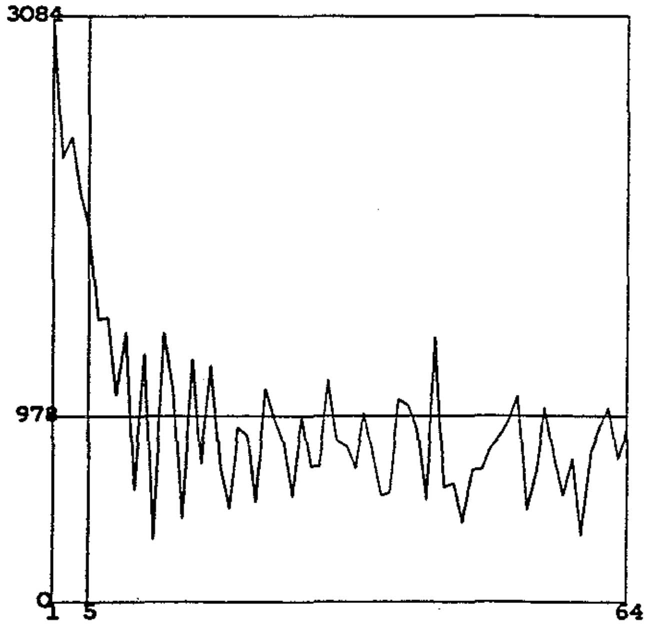
Building Scene 1: 128 Processors



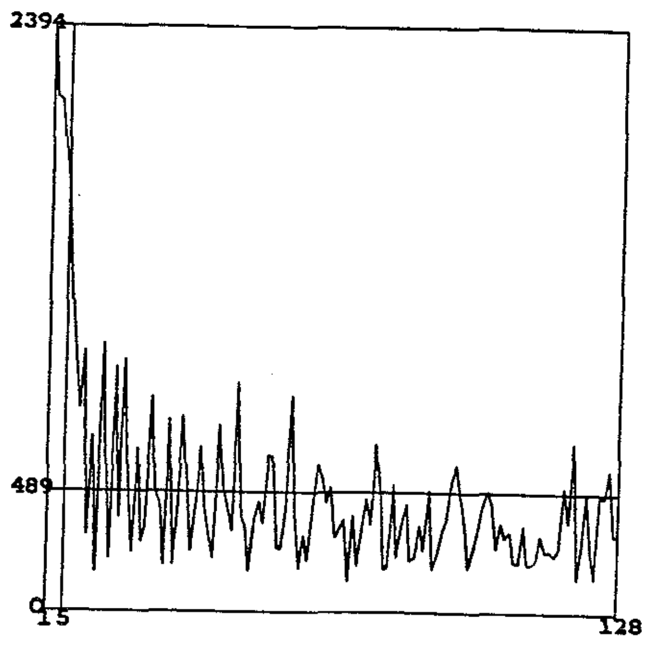
Building Scene 2: 64 Processors



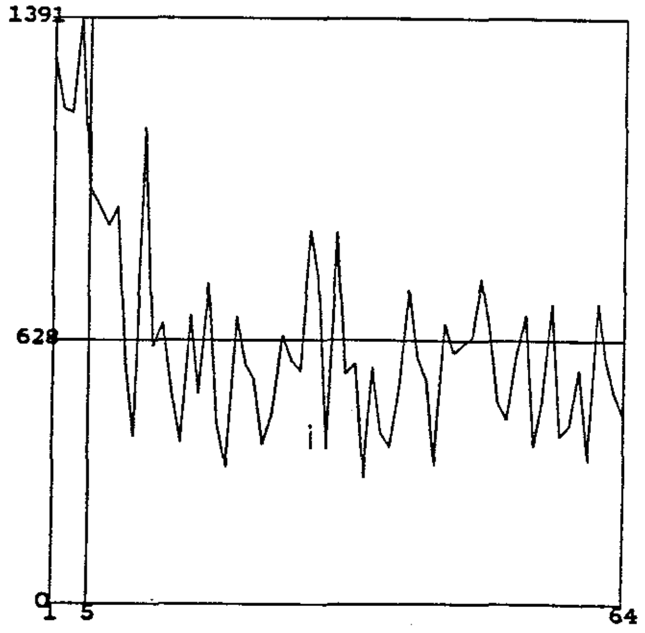
Building Scene 2: 128 Processors



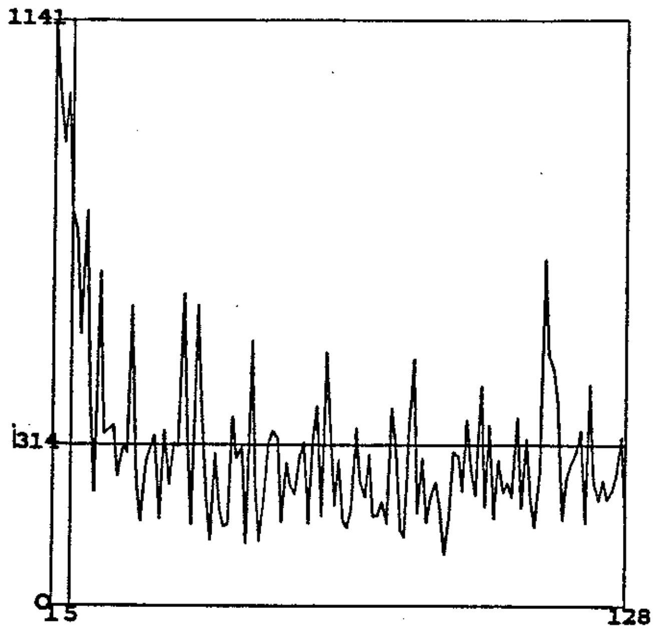
Building Scene 3: 64 Processors



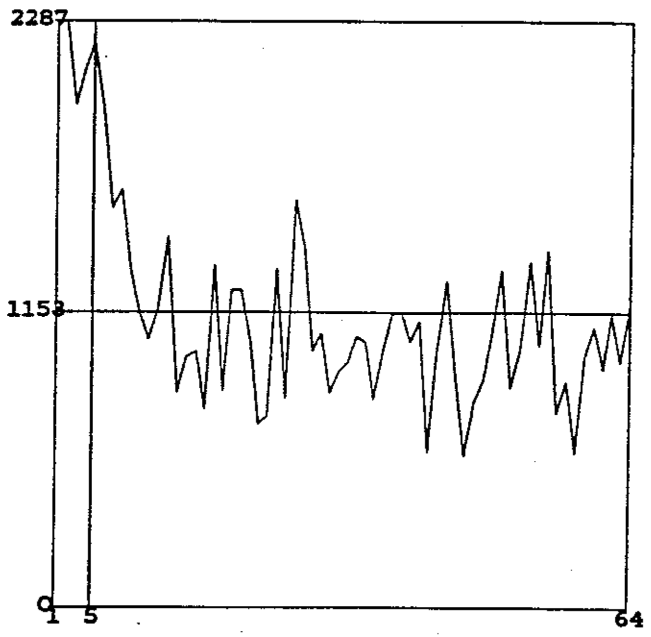
Building Scene 3: 128 Processors



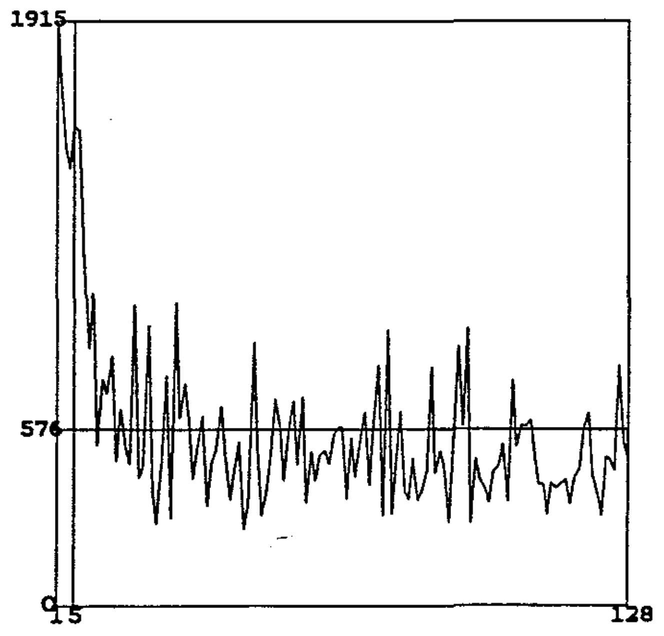
Aircraft Scene 1: 64 Processors



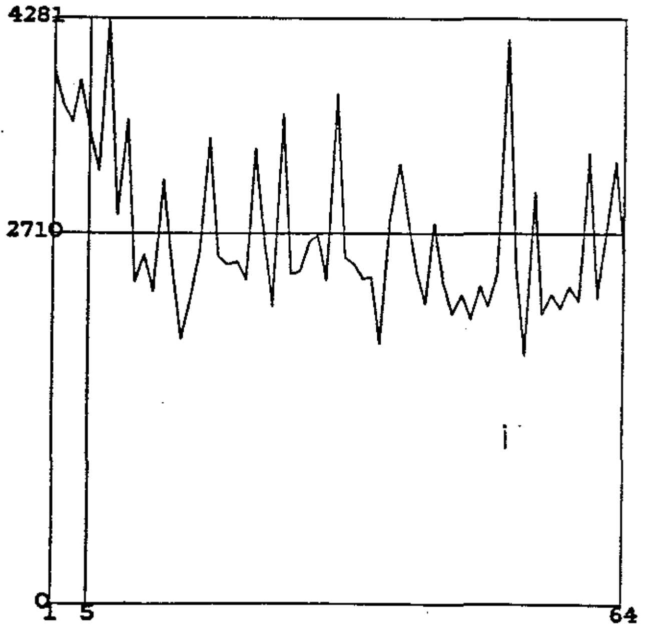
Aircraft Scene 1: 128 Processors



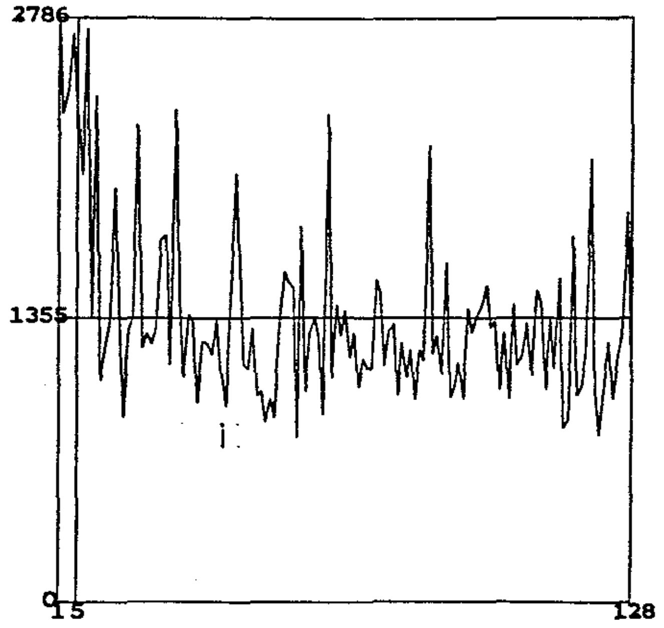
Aircraft Scene 2: 64 Processors



Aircraft Scene 2: 128 Processors



Ship Scene: 64 Processors



Ship Scene: 128 Processors