Parallel Machines for Computer Graphics

Thesis by

Michael K. Ullner

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1983

(Submitted January 7, 1983)

PARALLEL MACHINES FOR COMPUTER GRAPHICS

Acknowledgments

My stay at Caltech was guided by a succession of advisors: Fred Thompson started me going, back in the old days; Ivan Sutherland pulled when I got stuck; Jim Kajiya pushed, and he carried a shovel for when things got messy. I appreciate their care and patience.

I would also like to thank Zipporah Sabsay for carefully scouring this monstrous document in search of errors in grammar and lapses in style. She did so not once, but twice!

I am especially grateful to Calvin Jackson, who was always willing to share his TEXnical expertise and to offer advice and encouragement. Cal's enthusiasm for the craft of bookmaking is infectious, and much of his taste has filtered into the appearance of this document.

Abstract

Computer graphics provides some ideal applications for the kind of highly parallel implementations made possible by advances in integrated circuit technology. Specifically, hidden line and hidden surface algorithms, while easily defined and simple in concept, entail a substantial amount of computation. This requirement fits the characteristics of integrated circuit technology, where modular designs involving regular communication between many concurrent operations are rewarded with high performance at an acceptable cost.

Ray tracing is a very flexible technique that can be used to produce some of the most realistic of all computer generated images by simulating the interactions of light rays with surfaces in a modeled scene. Because light rays are mutually independent, many may be processed simultaneously, and the potential for concurrency is great. One architecture for expediting a ray tracing algorithm consists of a conventional computer equipped with a special purpose peripheral device for locating the intersections of rays and surfaces. This intersection computation is the most time consuming aspect of a ray tracing algorithm. Although the attached processor configuration can produce images more quickly than an unaided computer, its performance is limited. Alternatively, a pipeline of surface processors can replace the peripheral device. Each processor computes the intersections of its stored surface with rays that flow through the pipe. Such a machine machine can be quite fast, and its performance can be increased by lengthening the pipeline, but the component processors are not very effectively utilized. A third approach combines the advantages of the prior two machines by using an array of processors, each simulating a distinct subvolume of the modeled world by treating light rays traveling through space as messages flowing between processors. Local communication is sufficient because light rays travel continuously through space.

In real time computer graphics, successive images must be produced in times that are imperceptible to a viewer. Although the ray tracing machines fall short of this performance, it is possible to compromise image quality in order to produce a highly parallel machine capable of real time operation. The processors in such a machine are organized to form a binary tree. Leaf processors scan-convert surfaces, producing a sequence of segments, where a segment is the portion of a surface that appears on a single scan line of the display. Processors towards the root of the tree accept two such segment sequences and produce a third in which all segment overlap has been resolved. The final image is available at the root of the tree. The communication bottleneck that would otherwise occur at the root can be eliminated by breaking out

parallel roots, and the resulting tree may be extended to scenes of almost arbitrary complexity merely by increasing the supply of available processors.

Massive parallelism can also be applied to the problem of removing hidden edges from line drawings. A suitable architecture takes the form of a pipeline in which each processor is dedicated to the handling of a single polygon edge. These processors successively clip line segments passing through the pipeline to eliminate portions hidden behind surfaces. Each edge processor can be constructed out of little more than three serial multipliers.

The machines described here are varied in organization, and each functions differently, but their treatment of sorting is one ingredient common to all. Sorting is a key component of hidden surface algorithms running on conventional computers, but its extensive communication requirements make it costly for use in a highly integrated design. Consequently, the highly parallel machines described here operate largely without sorting. Instead, they maintain information in sorted order or make use of already sorted information to limit communication requirements.

Table of Contents

1

Introduction

Computer graphics is the art and science of making pictures by means of computation. Beginning with little more than an idea for a scene to be depicted, the artist/programmer must first transcribe that idea into a geometric model. Next, some form of computing engine processes the model into the electrical signals that control the electron beam in a cathode ray tube. The modulated beam of electrons strikes the inside surface of the tube, exciting a phosphor coating to produce the pattern of light that we perceive as the final image.

Since its inception, progress in computer graphics has been tied to the availability of suitable computing and display hardware. Some of the earlier displays could show only patterns of dots, and the images that could be produced were correspondingly limited. Subsequent displays could draw lines in addition to dots, thereby extending the range of possible images. The frame buffer, a device for storing a digitized, continuous tone image and displaying it on a television monitor, represents a further advance in display hardware. Early frame buffers used magnetic disks for the storage medium, but as memory technology improved, frame buffers began to use large, random access memories. The new devices were not only cheaper and more readily available, but also their very existence suggested new applications and algorithms for computer graphics.

The progress of computing technology, as well as display technology, has influenced the progress of computer graphics. The algorithms used to make pictures tend to be computationally intensive. In fact, it has been estimated that even a Cray-1, which

is generally acknowledged to be among the fastest computers in the world, can produce only about four minutes of high quality motion pictures per month [WHIT82]. This figure takes into account the draft work that must precede the finished product. Most computer graphics, however, is done on minicomputers rather than supercomputers, forcing compromises in image realism. Another branch of computer graphics makes use of special purpose processors in order to produce successive images in times that are imperceptible to the viewer. Because it is so difficult to generate images at this rate, the pictures produced by these machines tend to be less complex than the pictures that can be made without timing restrictions. Such machines are typically used in visual environment simulators, and because the moving images are under direct control of the viewer, many of the image quality compromises that were made in exchange for performance can be overlooked.

One of the more recent technological developments that can be applied to computer graphics is the practice of fabricating large scale integrated circuits. Using this technology, it is possible to fit an entire electronic subsystem onto a single, fingernail-sized chip of silicon. The cost per computing device is correspondingly diminished, making it feasible to apply more computing power to a task with the hope of improving performance. Like all technologies, however, integrated circuits have their own peculiar set of properties that must be considered if the final product is to be a practical one. For integrated circuit technology, it is cheap to replicate components, but communicating between the various parts of a system is very expensive. These characteristics suggest that highly parallel algorithms are suitable, so long as the communication between the various operations taking place concurrently is not excessive.

The following pages explore some computer graphics algorithms that can utilize the technology for fabricating very large scale integrated circuits. The algorithms are designed to make effective use of concurrency while keeping communications requirements down to an acceptable level. The primary emphasis will be on algorithms for detecting and removing those parts of a simulated scene that should be hidden from the viewer. One type of algorithm will concentrate on rapidly producing images of very high quality, while another will accept poorer image quality in order to achieve high performance. Before proposing the algorithms and their associated machine architectures, however, a brief overview of the technology characteristics and the problems involved at the graphics end may be helpful.

1.1 Integrated Circuit Technology

Using large scale integrated circuit technology, it is possible to fabricate a single chip of silicon containing an amount of logic that formerly required an entire printed circuit board. It would be a mistake, however, solely to regard integrated circuits as miniaturized printed circuit boards, because they have a different set of strengths and weaknesses. The major differences have to do with communications. Within

an integrated circuit, the cost of transmitting a signal across the chip is substantially greater than the cost of communicating it locally. Not only does a longer transmission path consume more area, it is also slower because it has a larger capacitance and stores a correspondingly greater charge that must be overcome in order to accomplish the transmission. Compared with these communication paths, the actual active computing elements consume relatively little power, time, or area. This situation led Sutherland and Mead to observe that integrated circuits make "switching elements essentially free ... leaving wires as the only expensive component" [SUTH77].

Just as the communications within a single chip limit performance, the amount of communication between chips is restricted. A single printed circuit board might have hundreds of connections to its environment. In contrast, an unusually large and complex integrated circuit might have eighty pins connecting it with the outside, while typical chips have fewer than fifty. These pins supply power to the chip as well as serving as portals for data transfer. In addition to being limited by the number of pins on a chip, communication between two chips is generally slower than communication between points within a single chip.

Testing is another area in which integrated circuitry requires some extra attention. On a printed circuit board, most of the signals are accessible to an oscilloscope probe. This is not so with an integrated circuit, where the only access to the internals of a chip is through the limited number of signal pins. The problem of testing is still largely unresolved, but one technique that has been used successfully involves chaining a portion of a chip's internal state into a long serial register, which may then be shifted out of the chip through a small number of pins for examination and possible modification [EICH77].

In many ways, the process of fabricating integrated circuits is similar to the techniques of book printing. In both cases, the setup costs are high, but the incremental cost of producing each unit is comparatively low. After printing the pages of the book or fabricating the chip, a culling phase is required to eliminate defective units before packaging the final product. Because highly replicated chips are individually less expensive, an engineer contemplating the use of custom integrated circuits is encouraged to come up with designs that can make use of many copies of a single chip type.

The characteristics of integrated circuit technology tend to push one in the direction of highly parallel designs with regular, if somewhat constricted, communications paths. The goal is to achieve high performance by applying a large number of concurrently operating processing elements to a particular problem. Since it is easy and inexpensive to duplicate chips, it is also desirable to apply a myriad of identical processors. The fact that all processors are the same tends to encourage the use of regular interconnection structures. Notice that this approach is almost completely opposite from the one that is useful in printed circuit technology, where overall performance is achieved by relying on the individual component speeds rather than on their combined speeds.

Many types of processor interconnection structures have been proposed or actually implemented [ANDE75]. The most common of these is the conventional pipeline

consisting of a linear array of processors, each connected to its two neighbors. Work flows into one end of the pipe and passes from processor to processor, rather like the material in an assembly line. Each processor performs a single stage of the computation before it passes a partial result to the next processor. The final result appears at the end of the pipeline. More elaborate pipelines have also been proposed [KUNG80]. These organizations have more than one input and output stream, and the internal communication schemes are more complicated, but they still pass partial results in one direction from processor to processor in a very regular and rigid manner.

Other types of interconnection strategies suggest communication schemes that are different from the approach used in linear pipelines. In tree structures and arrays, for example, each processor is connected to a few nearby processors. In trees, each component processor has connections for a superior processor and for two or more subordinate processors. Component processors in a rectangular array are each connected to four others, although different kinds of arrays may have richer interconnection patterns. For these and other types of structures, asynchronous message passing is often a more useful communication strategy than the rigid, unidirectional flow found in conventional pipelines. Using this technique, the communication occurs only when and where there is information to be transmitted. The disadvantage, of course, is that message passing requires a great deal more logic devoted to its control.

1.2 Computer Graphics

As mentioned earlier, the first step in making a picture is to transcribe an idea into a geometric model suitable for manipulation by computer. This usually means that the surfaces in the scene must be split up into a collection of planar polygons defined in three dimensions, although some techniques for dealing directly with curved surfaces have been developed. One way to prepare the scene model is to use a large digitizing tablet to derive three-dimensional coordinates by tracing blueprints of the objects. After the geometric properties of the surfaces have been described, it is still necessary to provide color and texture information for them. Also, the light sources that illuminate the scene must be identified and described. The result is a complete, three-dimensional model of the scene to be depicted.

Because television and movie screens are flat, the scene model must be projected onto two dimensions before it can be displayed. The projection operation simulates the action of a camera, given its position and the direction in which its lens is pointed. After projection, surfaces that would obscure one another in a natural scene overlap each other in the synthetic one. These obscured surfaces must be detected and deleted if the final image is to appear at all realistic. The process for doing so is called hidden surface elimination, and the corresponding techniques are called visible surface algorithms.

A great variety of visible surface algorithms have been developed over the years,

but they generally share a common feature. The algorithms all perform some form of sorting operation to reduce the amount of computation necessary for locating overlapping surfaces [SUTH74b]. The sorting is used, in some sense, to bring surfaces that appear near to each other on the screen into similar proximity within the machine implementing the algorithm. The goal is to reduce the number of surfaces that must be compared, since this comparison is usually very time consuming. The visible surface algorithm is often considered to be the heart of a computer graphics system. Various algorithms will be considered in greater depth in the chapters that follow.

After the visible surface algorithm has terminated, it is still necessary to determine the colors of the remaining surfaces. Once again, researchers have devised many techniques for modeling the interactions between light and surfaces [BLIN77]. The apparent color of a point on a surface is some function of the color and other properties of the surface, the colors of the light sources, and the relative positions and orientations of the simulated camera, surface, and light sources.

The completed image can be displayed on some type of frame buffer, as mentioned earlier. This device consists of a fairly large memory, where each word in the memory stores the color of a single position on the screen. These individually accessible screen positions are called picture elements, or pixels. The fact that a screen can display only a finite number of pixels poses one final obstacle in the image generation process: aliasing [CROW77]. The number of pixels on the screen is limited, and the values of individual pixels are determined by discretely sampling the continuous simulated scene. If the spatial frequency of the scene is too high relative to the sampling resolution, aliasing will result. It generally shows up as jagged edges, or as small features that disappear and reappear in successive frames of a movie. Although its causes are well understood, there are no really good methods for eliminating aliasing. One commonly used technique can, however, reduce the visible effects. This method involves computing the image at a resolution greater than the screen resolution and then filtering the higher resolution image to obtain the lower resolution one.

There are two reasons that computer graphics provides an ideal application area for parallel implementations. First of all, the algorithms are sufficiently involved and time consuming that a parallel solution can provide genuine performance benefits. Second, the problems can often be cleaved into a collection of smaller problems with manageable intercommunication requirements. These two properties mesh well with the characteristics of integrated circuit technology.

1.3 Overview

Computer graphics systems may be classified into one of four broad categories on the basis of their performance: real time, movie time, still time, and too slow. Of course, the classification of a particular implementation depends not only on the algorithm being used, but also on the hardware support that is available and on the

complexity of the image being produced. Real time systems are the fastest, generating successive images quickly enough for a viewer to interact with them. That is, a viewer can make small changes in the scene and immediately see the visible effects of the changes. Real time performance is generally considered to mean that the system must be able to produce thirty completely new images every second, although for some applications this requirement may be relaxed a bit. Movie time performance is achieved if the system is fast enough to compute the successive frames of a motion picture in a reasonable amount of time, but not fast enough to produce them in real time. Presumably, the goal is to provide better image quality, or lower cost, in a movie time system than in a real time one. The next category is still time performance, where the time required to compute a single image is so long that computing an entire movie would be infeasible. Graphics programs running on minicomputers often fall into this category. Finally, implementations where the mean time between failures of the hardware exceeds the time required to make a picture are simply too slow.

Ray tracing is a very flexible approach to making pictures. With it, one can generate very realistic images that show reflection, refraction, and shadows. The difficulty is that although it is simple in concept, ray tracing requires a great deal of computation. Its performance ranges from still time to too slow, although some very short and limited movies have been made. Fortunately, as discussed in Chapter 2, special purpose hardware can boost the performance of a ray tracing algorithm solidly into the movie time category, making ray tracing a practical alternative to approaches that may be faster, but are more complicated and less functional.

The timing constraints imposed on a real time system are severe, since there is quite a bit to be done in the thirtieth of a second that is available for each frame. Existing real time systems have achieved high performance largely by relying on the speed of their individual components. When using integrated circuit technology, however, the speed of a single chip is not always very great, and performance must be achieved by amassing many chips. The challenge of integrated circuits, therefore, is to devise highly parallel algorithms that have manageable communications requirements, as well. Chapter 3 discusses some approaches to this problem.

2

Ray Tracing Machines

Ray tracing is a technique that is capable of producing some of the most realistic of all computer generated images. It is primarily a method for performing hidden surface elimination, but the primitive operation of tracing a ray can also be used to model shadows, reflective surfaces, and transparent surfaces with refraction. In fact, essentially all of the effects available in shaded computer graphics can be achieved by a ray tracing algorithm. Beyond pure image generation, ray tracing can be used, for example, to estimate the volumes of modeled objects, as well as to simplify the construction of these models.

As might be expected, the drawbacks of ray tracing are almost as pronounced as its benefits. First of all, it is among the slowest of the algorithms for hidden surface elimination that run on commercially available computers. It is so tedious that it is never considered for most applications. Another difficulty is that a ray tracing algorithm produces an image by sampling the object space, rather than by any kind of direct or exact computation. This makes it inherently susceptible to the various aliasing problems that can be caused by sampling too infrequently.

Ray tracing is, however, interesting in the context of parallel computation. It suggests a natural way of separating the problem of hidden surface elimination into a vast number of simpler computations that are largely independent of one another. Therefore, it seems especially suitable for implementation on a machine consisting of a number of relatively unsophisticated processors operating in parallel.

This chapter proposes three different organizations for machines designed to

implement a ray tracing algorithm. Each of the organizations was designed to exploit a different aspect of the concurrency that is potentially available in the algorithm. In the first approach, the primitive ray tracing computation itself provides the concurrency. In the second approach, all of the polygons in the scene are processed at the same time. Finally, in the third approach, polygons are separated into disjoint regions of volume, and these regions are processed simultaneously. Before delving into the details of these three machines, however, it may be helpful to understand the general behavior of a ray tracing algorithm.

2.1 The Ray Tracing Algorithm

Ray tracing algorithms were pioneered by Appel back in the late 1960's. The Mathematical Applications Group, Inc. (MAGI) has been using ray tracing for quite some time in their production of television commercials and video logos, among other things. Other than this, however, the techniques lay dormant for over a decade until fairly recently when they were revived and extended by Whitted. Ray tracing seems to be getting more attention lately as people come to appreciate its simplicity and generality. The remainder of this section describes ray tracing algorithms, and it may be skipped by readers who are already familiar with them.

To understand how a ray tracing algorithm works, imagine a scene consisting of a number of objects illuminated by a single light source, as sketched in Figure 2–1. Light rays emitted from the light source may strike the surfaces of objects in the scene, and secondary light rays will be emitted from the points of intersection in ways that depend upon the surface characteristics of the objects. For example, the color of a secondary ray is partially determined by the color of the surface. In addition, the distribution of secondary rays depends on how glossy the surface is. Secondary light rays may strike other objects in the scene to create many levels of interaction.

Eventually, some of the light rays will reach the eye of a nearby observer, who will perceive them as an image. Suppose that there is a transparent sheet of glass in a wall between the observer and the scene, so that light rays must pass through the glass before the observer can see them. One ideal in computer graphics is to be able to replace this sheet of glass with the screen of a television monitor in such a way that the observer cannot distinguish between the genuine image and a synthetic one. Of course, this ideal cannot be realized in practice, partly because of the two-dimensional nature and limited resolution of the display screen. Assume, therefore, that the observer sits very still, wears a patch over one eye, and is accustomed to viewing the world through a window screen.

The direct approach to determining the synthetic image is to simulate the behavior of light rays using models of the scene, the light source, and the surface interactions. Rays can be simulated by tracing them from the light source to determine whether they will strike an object. If this happens, the surface interactions must be

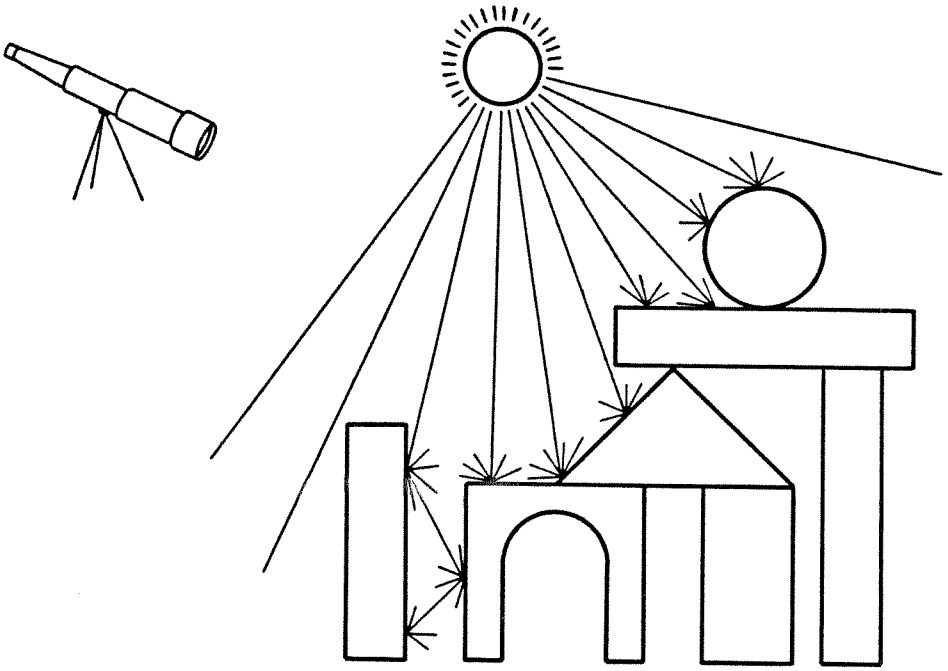


Figure 2–1. Light rays illuminating a scene.

simulated to create new secondary rays, which should then be traced in a similar manner. If, by chance, a ray passes through the simulated sheet of glass on a path leading to the observer, it is captured and displayed on the television screen. Of course, since the number of rays leaving the light source is essentially infinite, as is the number of rays emitted from each surface intersection, the rays must be simulated by using some sort of sampling technique. Unfortunately, only a very small proportion of the rays leaving the light source will ever contribute to the final image, so that any effort spent simulating the others will have been wasted.

Notice that the only rays that are really of any interest are those that arrive at the viewer's eye by passing through the simulated sheet of glass. Furthermore, since the final image will be displayed on a television monitor as a raster of pixels, rays passing through the pixels of this raster are more pertinent than those passing between them. It seems prudent, therefore, to restrict the algorithm to this subset of the rays. A way to do this is to trace rays backwards from the viewing position, through the pixels on the sheet of glass, and out into the scene, as shown in Figure 2–2. These backwards light rays may be thought of as vision rays.

Tracing vision rays reduces the number of first-level rays that must be simulated, but the number of secondary rays remains large. Just as a light ray striking a surface generates secondary rays in all directions, one particular secondary ray could have been generated by a ray coming from any direction. This means that in order to simulate vision rays properly, it would be necessary to trace an infinite number of

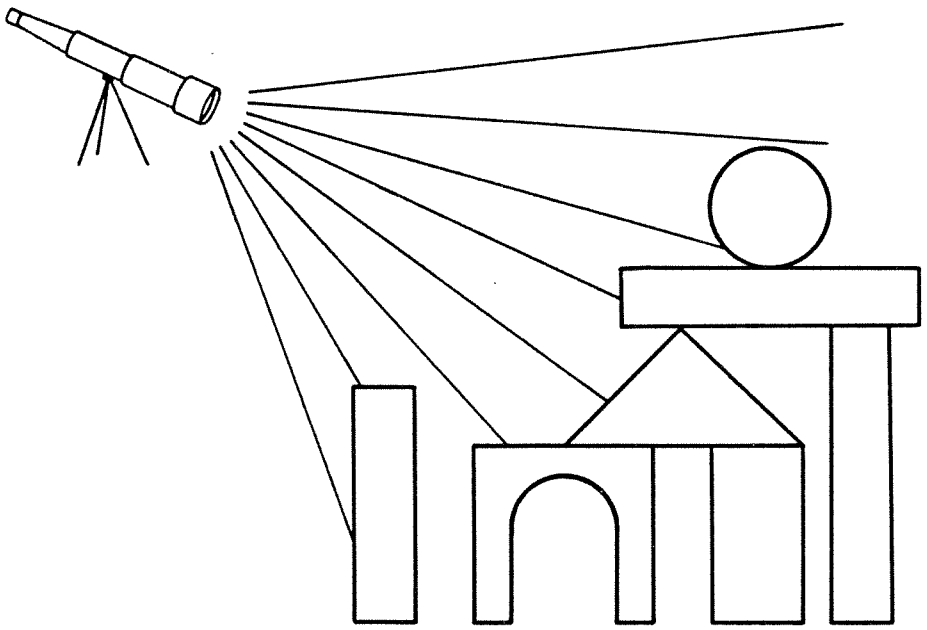


Figure 2–2. Tracing vision rays to sample a scene.

secondary vision rays emitted in all directions from each intersection point. Of course, not all directions are equally likely, and this fact can be used to reduce the number of rays under consideration. These types of approximations are successful to varying degrees.

Consider, for the moment, the simplest possible version of a ray tracing algorithm. For each pixel in the raster, the algorithm traces the corresponding vision ray to the first object that it encounters in the scene. The color of the surface at the point of intersection completely determines the color of the pixel in the final display; that is, secondary rays are just ignored. The resulting image will not show any hidden surfaces, but no shadows or other interesting lighting effects will be apparent either. It will be cartoon-like, resembling an image produced by one of the early algorithms for hidden surface elimination, like Warnock's algorithm [WARN69].

It is not difficult to see why ray tracing algorithms are generally considered to be slow. They make absolutely no use of any kind of coherence. That is, the process of tracing one particular ray does not provide any information that may be used to simplify tracing the next. It is exactly this mutual independence of rays, however, that makes ray tracing compatible with a parallel implementation. Since rays do not affect one another, it is possible to trace several rays concurrently. This aspect of ray tracing will be explored more fully in later sections.

The minimal ray tracing algorithm given above may be extended in a rather natural way to synthesize shadows. As before, the algorithm begins by tracing a vision

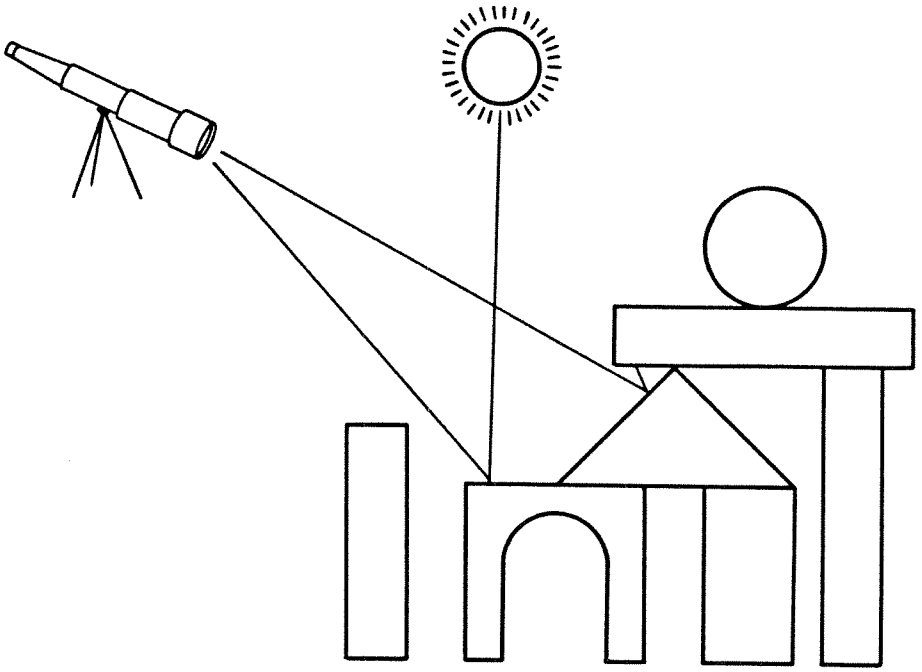


Figure 2–3. Tracing rays to a light source to determine whether intersection points are in shadow.

ray to determine which object is visible through a particular pixel. But this time, the algorithm traces a new ray emitted in the direction of the light source from the point of intersection on the visible surface. See Figure 2–3. If this new ray extends all the way to the light source without striking any other surface, then the intersection point is directly illuminated by the light source; otherwise, the point is in shadow. In a scene illuminated by more than one light source, separate rays must be traced in the direction of each light source. Appel used an algorithm much like this one in the late 1960's to produce pictures on, of all things, a digital pen plotter [APPE68]. This was in the days before there were devices capable of displaying true shaded images.

Reflection is another phenomenon that a ray tracing algorithm can model. As in the case of shadows, the extension to handle reflection is a rather natural one. The algorithm traces a vision ray to determine which point in the scene is visible from a particular pixel. Then, in addition to shadow rays aimed at the light sources, it generates a new ray in the direction of reflection, as in Figure 2–4. This reflection ray may, in turn, strike another surface and generate still more rays. The surface properties and shadowing information at all of the intersection points combine to determine the color of the original pixel.

Refraction can be produced in much the same way as reflection. Instead of generating a new ray in the direction of reflection, however, the algorithm must generate it in the direction of refraction, according to Snell's Law and the index of refraction of the material. Refraction rays are illustrated in Figure 2–5. Actually,

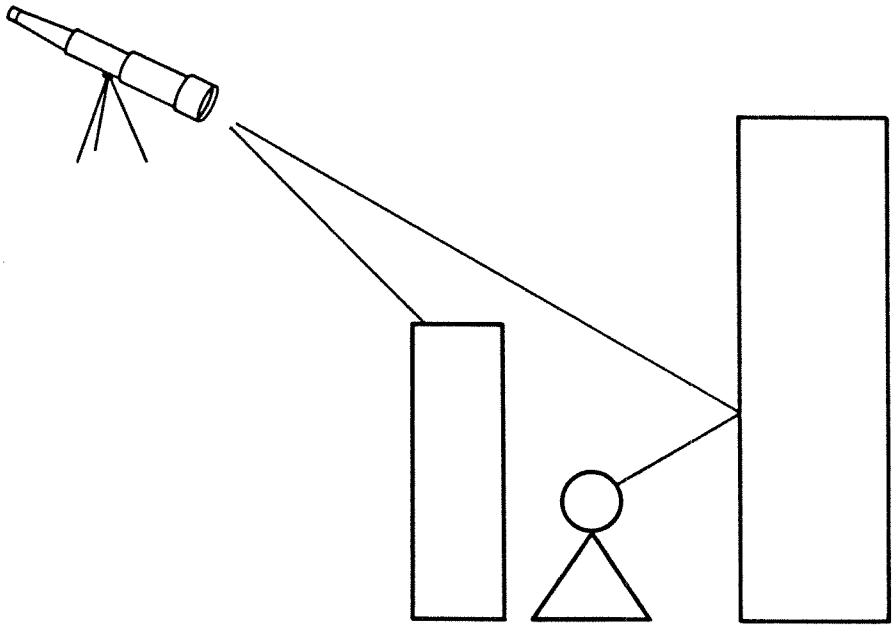


Figure 2–4. Tracing rays from an intersection point in the direction of reflection.

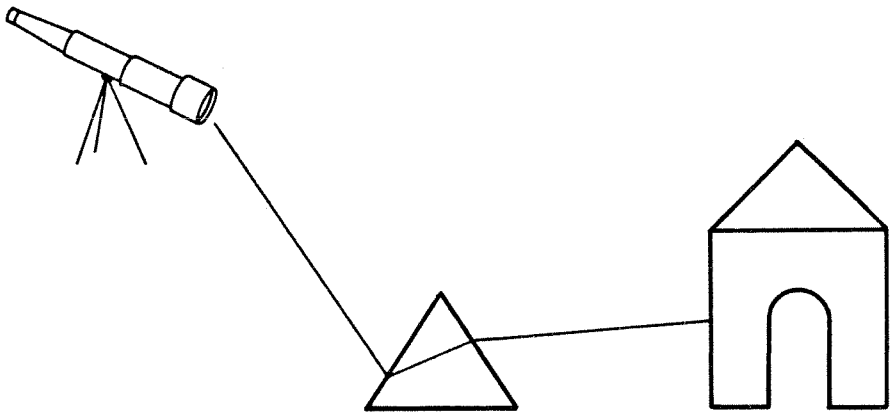


Figure 2–5. Tracing rays from an intersection point in the direction of refraction.

it is perfectly permissible to generate both a reflection ray and a refraction ray. For instance, both kinds of rays would be needed to make a picture of a magnifying glass with reflections visible on its surface.

Sampling too infrequently can be a problem with all ray traced images, but the techniques used for modeling reflection and refraction can further aggravate the situation. For example, rays reflected from a convex surface will be more "spread out" than the incident rays, so that the reflected image is not sampled as frequently as the rest of the image. Whitted has used ray tracing techniques like those described above to make images showing reflection and refraction [WHIT80]. He also suggested a fairly simple way of dealing with the problem of aliasing.

Whitted's technique for anti-aliasing begins by examining the colors of adjacent pixels in the displayed image. If the colors are sufficiently different, the algorithm assumes that the pixels lie on opposite sides of an edge. In this case, it traces another ray between the pixels so as to determine the color at that intermediate point. This process is applied recursively either until the colors are sufficiently similar or until some fixed depth limit is reached. Finally, the algorithm averages, or otherwise filters, the colors at the intermediate points to determine the colors of the original pixels. This procedure may still, however, overlook small objects that fall between adjacent pixels in the raster. Whitted solved the problem by forcing a subdivision in this case, even when the nearby pixels are colored similarly. A pleasant aspect of this anti-aliasing technique is that the effort tends to be concentrated along edges, where it is most needed. Moreover, it meshes well with the rest of the ray tracing algorithm by making use of the same basic computation.

Ray tracing can also be used with the various surface mapping techniques that have been developed. These work by setting up a mapping between a unit square and a surface in the scene, as shown in Figure 2–6, so that values of an arbitrary function can be associated with points on the surface. Values mapped in this way can then take part in the computation of the intensities to be displayed in the final image. Catmull first used the technique to map photographs and images of textures onto surfaces [CATM74]. In his pictures, the function values derived by sampling stored images were used to determine the colors at points on surfaces. Objects in the resulting images appeared to have photographs glued to their surfaces. Blinn was able to produce convincing pictures of wrinkled and bumpy surfaces by mapping perturbations of surface normal vectors [BLIN78].

Still other types of mappings are available in a ray tracing algorithm because of its greater generality. For example, mapping a surface's index of refraction might be an effective way to model a simple lens. Transparency mapping is another technique that can be used when intricate two-dimensional objects are to be displayed. Conventionally, this kind of object would be modeled by a myriad of polygons. Using transparency mapping, however, the usual surface model gives the general spatial location of the object, while the transparency map determines where on this surface the object actually exists. Notice that the number of surfaces has been substantially

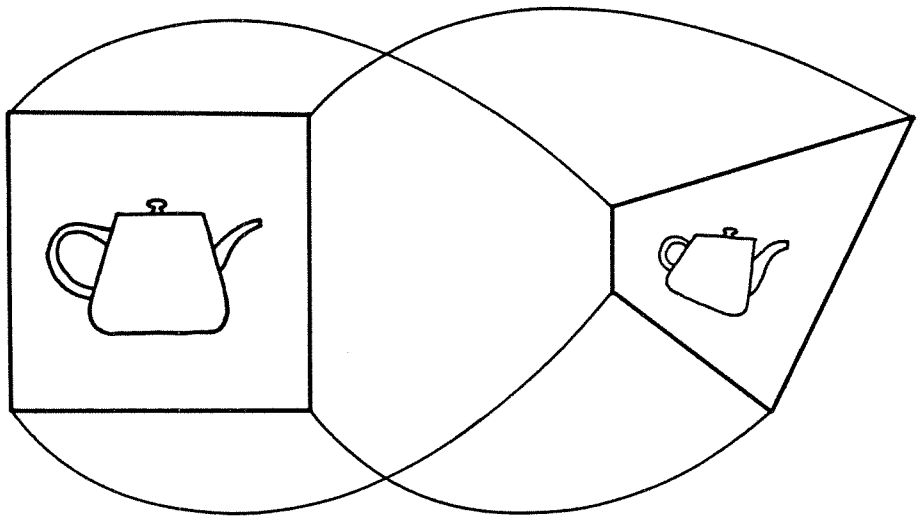


Figure 2–6. Mapping a unit square in the uv-plane to a surface in the scene.

reduced, thereby easing the burden on the hidden surface processor as well as the modeler.

Although ray tracing is probably the most flexible hidden surface and shading algorithm available, there are still many effects that are difficult or impossible for it to achieve. One source of difficulty stems from the fact that ray tracing does not make use of phase information. Thus, it cannot simulate diffraction, but this is probably not a very serious shortcoming for most applications. A more important problem occurs in situations that involve diffuse illumination, reflection, or transmission. In a diffusely illuminated scene, the light does not come from a point source, and therefore shadows do not have sharp edges. Diffuse reflection occurs, for instance, when an object takes on some of the color of another object nearby. Illumination that is reflected by a mirror is a related problem. A view through a translucent screen is an example of diffuse transmission. In each of these cases, the assumptions used to reduce the number of secondary rays are no longer valid. Of course, it is still possible to approximate these effects by tracing a random sample of rays, but it would be very costly to do so.

It was mentioned earlier that the use of a ray tracing algorithm for rendering an image of a scene could, in some cases, ease the task of producing the scene model. The basic idea here is that surfaces may be thought of as enclosing either positive or negative volume and, furthermore, that these volumes can be combined algebraically. For example, to model a hole in an object, a negative object correspond-

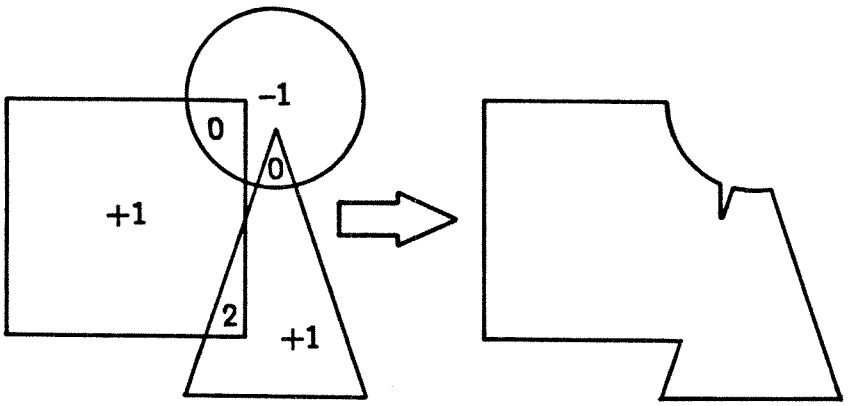


Figure 2–7. Using negative volumes to model a holes in an object. The square and triangle have positive volume, while the circle has negative volume. The numbers correspond to the net volume of each region.

ing to the hole can be superimposed on the positive object, partitioning space into regions of positive, negative, and zero net volume. Such a case is diagrammed in Figure 2–7. The surface of the complete object is the boundary between the positive and zero volumes. To make an image of this object using a ray tracing algorithm, it is necessary to find the intersections of a ray with each of the modeled surfaces. From these intersections, along with the surface descriptions, the algorithm can determine when the ray is passing through the positive volume, negative volume, or zero volume. Finally, the visible point is where the ray first passes into the positive volume. The Mathematical Applications Group, Inc. (MAGI) has been using techniques like this for quite some time to produce television commercials and the like [GOLD71]. More recently, the techniques have been applied to the design of automotive parts [ROTH80].

In the automotive application, ray tracing can produce not only a realistic image of the part, but also an estimate of the volume of an object, so that its weight can be computed. The volume of an object can be determined to any desired precision by passing a raster of parallel rays through it. For each ray, the algorithm accumulates the length of the ray passing through the object's positive volume. When the accumulations from all of the rays are added together and the sum is multiplied by the cross-sectional area of the original raster, the result is an estimate of the object's volume. The precision of this estimate can easily be increased by tracing more rays.

2.2 Computations for Ray Tracing

This section describes some of the computations needed to implement a ray tracing algorithm as outlined in the previous section. The discussion will be limited to planar surfaces, primarily because this simplification makes hardware implementations more feasible. Furthermore, only convex quadrilaterals will be considered. These may, of course, be pieced together in order to form arbitrarily complicated polygons or other surfaces. For the rest of this discussion, the term "polygon" should be understood to mean "convex quadrilateral."

The main computation involved in ray tracing determines whether a given ray intersects a given polygon and, if it does, determines where on the ray and where on the polygon the intersection occurs. The position on the ray is described by the distance between the origin of the ray and the point of intersection. The ray tracing algorithm will use this distance to isolate the first polygon struck by the ray. The position on the polygon is specified by a point within a two-dimensional unit square. If the polygon is considered to be a mapping from the square onto a surface in three dimensions, then the point on the polygon where the ray and polygon intersect corresponds to a point in this unit square. The polygon position will be used in the shading computations and, optionally, for the various effects like texture mapping. Using quadrilaterals makes it possible to have a fairly direct mapping from the unit square to the polygon.

A ray can be specified by two points: $\mathbf{r}_0$ is the origin of the ray, and $\mathbf{r}_1$ is another point on the ray. The ray itself is represented parametrically, so that an arbitrary point along the ray is given by

$$\mathbf{r}(t) = (1-t)\mathbf{r}_0 + t\,\mathbf{r}_1 = (\mathbf{r}_1 - \mathbf{r}_0)t + \mathbf{r}_0.$$

Notice that if the origin of the ray corresponds to the viewing position, negative values of the parameter t represent points behind the viewer. Also, since the parameter value is a measure of the distance between the origin of the ray and a point along its length, it may be used to determine which of two points on the ray is visible. The unit of measure is the distance between $\mathbf{r}_0$ and $\mathbf{r}_1$.

The representation of polygons is somewhat more involved. At some point in the modeling process, polygons are represented as a sequence of vertices, but this format is not especially convenient for all of the computations needed in a ray tracing algorithm. Indeed, the polygon will appear in several guises at various stages in the algorithm.

Recall that the primary computation finds the intersection of a ray and a polygon. This task will be broken down into two steps. The first step finds the intersection of the ray with the plane containing the polygon. The second step determines whether the intersection point actually lies within the polygon and, if so, computes the point on the unit square corresponding to the intersection point in the polygon.

Polygons may be thought of as having a parametric representation. $\mathbf{p}(u, v)$ is a point within the polygon, where u and v each range between zero and one. The vertices of the polygon correspond to the vertices of the unit square in the uv-plane:

$$\mathbf{p}_{01} = \mathbf{p}(0,1), \qquad \mathbf{p}_{11} = \mathbf{p}(1,1),$$
$$\mathbf{p}_{00} = \mathbf{p}(0,0), \qquad \mathbf{p}_{10} = \mathbf{p}(1,0).$$

An arbitrary point within the polygon is given by a bilinear interpolation of the vertices according to the values of u and v. Thus,

$$\begin{aligned}\mathbf{p}(u,v) &= (1-u)(1-v)\mathbf{p}_{00} + (1-u)(v)\mathbf{p}_{01} + (u)(1-v)\mathbf{p}_{10} + (u)(v)\mathbf{p}_{11} \\ &= (\mathbf{p}_{00} - \mathbf{p}_{01} + \mathbf{p}_{11} - \mathbf{p}_{10})uv + (\mathbf{p}_{10} - \mathbf{p}_{00})u + (\mathbf{p}_{01} - \mathbf{p}_{00})v + \mathbf{p}_{00} \\ &= \mathbf{p}_a uv + \mathbf{p}_b u + \mathbf{p}_c v + \mathbf{p}_d,\end{aligned}$$

where

$$\begin{aligned}\mathbf{p}_a &= \mathbf{p}_{00} - \mathbf{p}_{01} + \mathbf{p}_{11} - \mathbf{p}_{10} \\ \mathbf{p}_b &= \mathbf{p}_{10} - \mathbf{p}_{00} \\ \mathbf{p}_c &= \mathbf{p}_{01} - \mathbf{p}_{00} \\ \mathbf{p}_d &= \mathbf{p}_{00}.\end{aligned}$$

The first stage of the ray tracing algorithm, which finds the intersection of the ray with the plane of the polygon, is most easily done if the plane surface is represented algebraically:

$$s_p(\mathbf{q}) = \mathbf{n}_p \cdot \mathbf{q} + d_p.$$

$s_p(\mathbf{q})$ is the distance from the plane of the polygon to an arbitrary point $\mathbf{q}$; $\mathbf{n}_p$ is the normal vector of the plane; and d_p is the distance from the plane to the origin. Thus, the intersection of the plane and the ray can be determined from the solution of

$$s_p(\mathbf{r}(t)) = 0.$$

Substituting and solving for t, we find that

$$t = \frac{\mathbf{n}_p \cdot \mathbf{r}_0 + d_p}{\mathbf{n}_p \cdot (\mathbf{r}_0 - \mathbf{r}_1)}.$$

If $\mathbf{n}_p \cdot (\mathbf{r}_0 - \mathbf{r}_1) = 0$, then the ray is parallel to the plane and therefore does not intersect it. Otherwise, t exists and can be used to find the point of intersection $\mathbf{p}_r$.

Having found the position along the ray, the next stage of the ray tracing computation determines the position on the polygon of the point where the ray and polygon intersect. That is, it finds u and v such that $\mathbf{p}(u,v) = \mathbf{p}_r$. It turns out that u and v can be determined independently of one another. To see how this may be done, first consider the family of lines derived by mapping constant u lines from the uv-plane to the plane of the polygon. Corresponding to each member of this family, there is a plane that both contains the line through the polygon and is perpendicular to the polygon itself. The equation describing this family of planes is

$$s_u(\mathbf{q}) = \mathbf{n}_u(u) \cdot \mathbf{q} + d_u(u).$$

Again, $\mathbf{n}_u(u)$ is the vector normal to a plane, and $d_u(u)$ is the perpendicular distance from the plane to the origin. The value of u at the intersection point $\mathbf{p}_r$ may be determined by solving $s_u(\mathbf{p}_r) = 0$. It remains to find expressions for $\mathbf{n}_u(u)$ and $d_u(u)$.

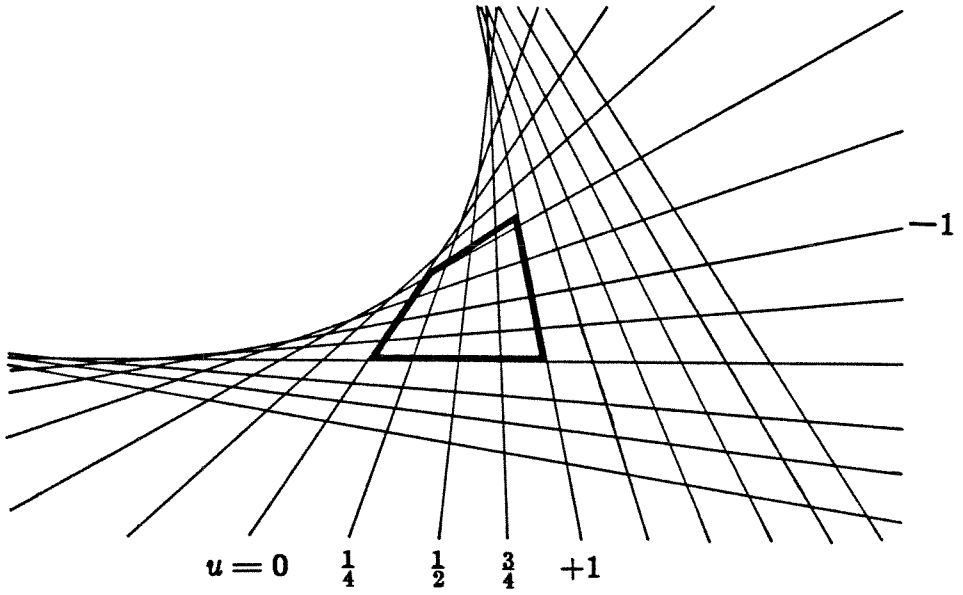


Figure 2–8. Some lines of constant u shown in the plane of a polygon that has no parallel edges. Some points on the plane do not lie along any of the lines; others are on two lines. For example, the $u = \frac{1}{2}$ and $u = -1$ lines intersect inside the polygon.

Before deriving these equations, it may be helpful to examine more closely the situation leading to them. First of all, consider the case, shown in Figure 2–8, where no two edges of the polygon are parallel. In this configuration, some points in the plane of the polygon lie along more than one of the constant u lines. The interior of the polygon is part of the area that is covered more than once. Also, there are other portions of the plane that do not lie along any of the constant u lines. This situation suggests that the expression for the constant u planes will be quadratic in u, so that solving this expression for u at the point of intersection $\mathbf{p}_r$ will yield two solutions. The solutions will be real if $\mathbf{p}_r$ lies in a portion of the plane that is covered by the lines of constant u; otherwise, they will be imaginary. Only one of the real roots, however, will lie between zero and one, representing an intersection on the interior of the polygon.

A distinctly different expression for the constant u planes arises if the $v = 0$ and $v = 1$ edges of the polygon are parallel, as in Figure 2–9. In this case, the lines of constant u cover most of the polygon plane exactly once, but there is a single point through which every line passes. This means that for most values of $\mathbf{p}_r$, it is possible to compute a unique value for u, but if $\mathbf{p}_r$ happens to be the point common to all of the constant u lines, no solution for u will exist. This behavior suggests that the expression for u as a function of $\mathbf{p}_r$ will take the form of a ratio.

It may now be appropriate to mention the reason for excluding concave polygons

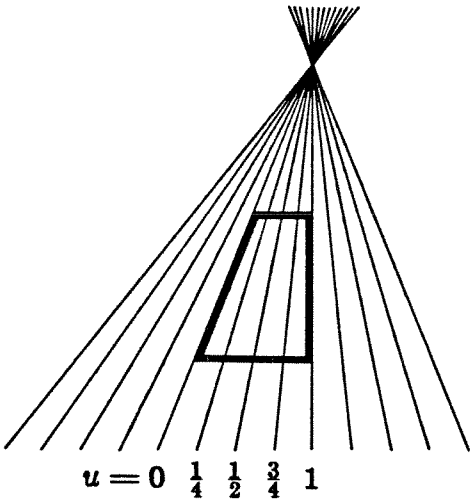


Figure 2–9. Some lines of constant u shown in the plane of a polygon whose $v = 0$ and $v = 1$ edges are parallel. Every point on the plane is on exactly one constant u line except for the single point through which all such lines pass.

from consideration. The problem is that the interpolation scheme used to find points on the interior of a polygon, given its vertices and a uv-coordinate pair, simply doesn't work if the polygon is concave. The difficulty is illustrated in Figure 2–10. First, some of the points on the interior of the polygon can be obtained from either of two valid uv-coordinate pairs. This fact alone would make it impossible to implement texture mapping or any of the related techniques. It also means that the computations for u and v would be mutually dependent. Second, the fatal flaw is that there are valid combinations of u and v that actually give rise to points outside of the polygon. Thus, concave polygons clearly fail to function within the framework that has been set up here.

The normal vector $\mathbf{n}_u(u)$ in the equation of the constant u plane may be derived from the parametric representation for the family of constant u lines in the polygon:

$$\mathbf{p}_u(v) = \mathbf{p}(u, v).$$

Note that this line is parametric in v for any fixed value of u. A vector in the same direction as this line of constant u can be formed by taking the difference of two points along the line, say $\mathbf{p}_u(0)$ and $\mathbf{p}_u(1)$. A vector that is normal to the constant u plane must be perpendicular both to this direction vector and to the normal vector of the polygon plane. A suitable vector may be conveniently constructed with a cross product:

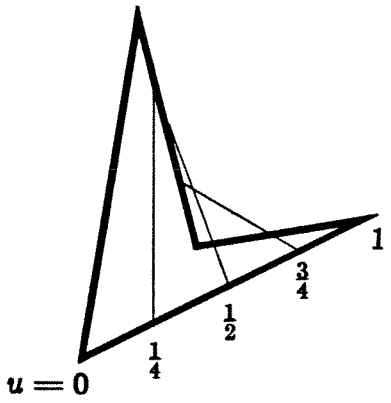


Figure 2–10. A concave polygon shown with some lines of constant u. Notice that each of these lines corresponds to a value of u between zero and one, and also that the lines are restricted to the segment where $0 \leq v \leq 1$. Because there are valid combinations of u and v that specify points outside the polygon, concave polygons must be disallowed.

$$\begin{aligned}
\mathbf{n}_u(u) &= \big(\mathbf{p}_u(1) - \mathbf{p}_u(0)\big) \times \mathbf{n}_p \\
&= \big((\mathbf{p}_a u + \mathbf{p}_b u + \mathbf{p}_c + \mathbf{p}_d) - (\mathbf{p}_b u + \mathbf{p}_d)\big) \times \mathbf{n}_p \\
&= (\mathbf{p}_a \times \mathbf{n}_p)u + (\mathbf{p}_c \times \mathbf{n}_p) \\
&= \mathbf{n}_a u + \mathbf{n}_c.
\end{aligned}$$

Of course, the magnitude of this vector need not be one, so it is not necessarily a unit normal vector, but at least its direction is correct.

An expression for $d_u(u)$ can be formed by substituting into the plane equation a point known to be on the constant u plane, setting the result to zero, and then solving for $d_u(u)$. Using the point $\mathbf{p}_u(0)$ and solving $s_u\big(\mathbf{p}_u(0)\big) = 0$ yields

$$\begin{aligned}
d_u(u) &= -\mathbf{n}_u(u) \cdot \mathbf{p}_u(0) \\
&= -(\mathbf{n}_a u + \mathbf{n}_c) \cdot (\mathbf{p}_b u + \mathbf{p}_d) \\
&= -(\mathbf{n}_a \cdot \mathbf{p}_b)u^2 - (\mathbf{n}_a \cdot \mathbf{p}_d + \mathbf{n}_c \cdot \mathbf{p}_b)u - (\mathbf{n}_c \cdot \mathbf{p}_d) \\
&= d_{u2}u^2 + d_{u1}u + d_{u0}.
\end{aligned}$$

With this expression, along with the one for $\mathbf{n}_u(u)$, the equation of the plane of constant u becomes

$$s_u(\mathbf{q}) = d_{u2}u^2 + (\mathbf{n}_a \cdot \mathbf{q} + d_{u1})u + (\mathbf{n}_c \cdot \mathbf{q} + d_{u0}).$$

The value of u at the point of intersection $\mathbf{p}_r$ is the solution of $s_u(\mathbf{p}_r) = 0$:

$$d_{u2}u^2 + (\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1})u + (\mathbf{n}_c \cdot \mathbf{p}_r + d_{u0}) = 0.$$

In the case where the $v = 0$ and $v = 1$ edges of the polygon are not parallel, d_{u2} will not be zero, and this expression may be solved by substituting into the quadratic formula:

$$u = \frac{-(\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1}) \pm \sqrt{(\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1})^2 - 4(d_{u2})(\mathbf{n}_c \cdot \mathbf{p}_r + d_{u0})}}{2(d_{u2})},$$

or

$$u = -\left[\left(\frac{\mathbf{n}_a}{2d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u1}}{2d_{u2}}\right)\right] \pm \sqrt{\left[\left(\frac{\mathbf{n}_a}{2d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u1}}{2d_{u2}}\right)\right]^2 - \left[\left(\frac{\mathbf{n}_c}{d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u0}}{d_{u2}}\right)\right]}.$$

This expression, of course, yields two solutions for u, but the only useful value is real and lies between zero and one. If both solutions are outside of this range, then either the ray and the polygon plane do not intersect, or the point of intersection is not inside the polygon. Otherwise, the value where $0 \leq u \leq 1$ locates the intersection point, and the other one is spurious.

If the two planes of constant v are parallel, then d_{u2} will be zero, and the equation $s_u(\mathbf{p}_r) = 0$ becomes

$$(\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1})u + (\mathbf{n}_c \cdot \mathbf{p}_r + d_{u0}) = 0,$$

which has the solution

$$u = -\frac{\mathbf{n}_c \cdot \mathbf{p}_r + d_{u0}}{\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1}}.$$

Again, the value of u must exist and lie between zero and one for $\mathbf{p}_r$ to be inside of the polygon.

The technique just described is equally suitable for finding the value of v at the point where the ray strikes the polygon. As in the prior case, the family of planes corresponding to constant values of v is represented algebraically:

$$s_v(\mathbf{q}) = \mathbf{n}_v(v) \cdot \mathbf{q} + d_v(v).$$

Points along the constant v line are $\mathbf{p}_v(u)$, so that the normal vector is

$$\begin{aligned}\mathbf{n}_v(v) &= \big(\mathbf{p}_v(1) - \mathbf{p}_v(0)\big) \times \mathbf{n}_p \\ &= (\mathbf{p}_a \times \mathbf{n}_p)v + (\mathbf{p}_b \times \mathbf{n}_p) \\ &= \mathbf{n}_a v + \mathbf{n}_b.\end{aligned}$$

Substituting this expression into $s_v(\mathbf{q})$ and solving $s_v\big(\mathbf{p}_v(0)\big) = 0$ for $d_v(v)$ gives

$$\begin{aligned}d_v(v) &= -(\mathbf{n}_a \cdot \mathbf{p}_c)v^2 - (\mathbf{n}_a \cdot \mathbf{p}_d + \mathbf{n}_b \cdot \mathbf{p}_c)v - (\mathbf{n}_b \cdot \mathbf{p}_d) \\ &= d_{v2}v^2 + d_{v1}v + d_{v0},\end{aligned}$$

and so the complete plane equation is

$$s_v(\mathbf{q}) = d_{v2}v^2 + (\mathbf{n}_a \cdot \mathbf{q} + d_{v1})v + (\mathbf{n}_b \cdot \mathbf{q} + d_{v0}).$$

In the non-parallel case, where $d_{v2} \neq 0$, solving $s_v(\mathbf{p}_r) = 0$ produces two values for v:

$$v = -\left[\left(\frac{\mathbf{n}_a}{2d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v1}}{2d_{v2}}\right)\right] \pm \sqrt{\left[\left(\frac{\mathbf{n}_a}{2d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v1}}{2d_{v2}}\right)\right]^2 - \left[\left(\frac{\mathbf{n}_b}{d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v0}}{d_{v2}}\right)\right]}.$$

As before, the only one of these values that is meaningful is real and lies in the range $0 \leq v \leq 1$. If both values are out of range, then the ray misses the interior of the polygon. In the parallel case, where $d_{v2} = 0$, v is given by

$$v = -\frac{\mathbf{n}_b \cdot \mathbf{q} + d_{v0}}{\mathbf{n}_a \cdot \mathbf{q} + d_{v1}}.$$

Again, this solution is valid only if it exists and if $0 \leq v \leq 1$.

After the algorithm has examined enough polygons to locate the first point at which the ray strikes a surface, it must apply a lighting model to determine the apparent color of that point. This color will contribute to the color of a pixel in the final image. One such lighting model was devised by Bui-Tuong Phong [PHON75]. It works well in a variety of situations, yet it doesn't require extensive computation. Basically, Phong's model separates the light illuminating a point into ambient, diffuse, and specular components. An additional component for transmitted light may be used to model transparent or translucent surfaces. The overall intensity is the sum of these four components:

$$I = RI_a + \sum RI_p \cos i + \sum W(i)I_p(\cos s)^n + TI_t,$$

where

I = Final intensity value.
R = Proportion of light reflected by the surface.
I_a = Intensity of the ambient illumination.
I_p = Intensity of the point light source.
i = Angle between the light source and the surface normal vector.
$W(i)$ = Specular reflection coefficient.
n = Shininess exponent.
s = Angle between the reflected light ray and the viewing ray.
T = Proportion of light transmitted by the surface.
I_t = Intensity of light passing through the surface.

Notice that if there were more than one light source, the diffuse and specular components of each would simply be added together. Also, if the point on the surface is

shadowed from any of the light sources, the corresponding specular contributions are omitted from the computation. Finally, although the above description was couched in terms of a monochrome image, full color is equally possible by using three equations for the three components of the color vector.

The two cosine terms of the illumination equation may be evaluated by taking dot products of appropriate vectors. The value of $\cos i$ may be computed from a unit vector in the direction of the light source $\mathbf{l}$ and the unit surface normal vector $\mathbf{n}$:

$$\cos i = \mathbf{n} \cdot \mathbf{l}.$$

Determining the value of $\cos s$ requires a vector in the direction of reflected light in addition to the normalized vector to the viewing position:

$$\cos s = (-\mathbf{l} + 2\mathbf{n}\cos i) \cdot \left(\frac{\mathbf{r}_0 - \mathbf{r}_1}{\|\mathbf{r}_0 - \mathbf{r}_1\|} \right).$$

The value of $W(i)$ can most easily be found by look up into a table indexed by $\cos i$. One more thing to notice is that R and T, the reflection and transmission parameters of the surface, need not be constant over the entire surface. It is sometimes useful to define them as functions of u and v so as to map a texture onto the surface. The values contained in these maps can be computed either directly as mathematical functions of u and v, or by using u and v as array indices. The index of refraction can be mapped in a similar manner.

A problem with the shading scheme presented above is that complicated surfaces made up of many polygons will appear faceted. Gouraud recognized this and suggested an approximation that smoothly shades such surfaces [GOUR71]. Subsequently, Phong devised an improved method that synthesizes a surface normal vector based on considerations more global than the properties of a single polygon. With the improved method, four normal vectors, $\mathbf{n}_{00}$, $\mathbf{n}_{01}$, $\mathbf{n}_{11}$, and $\mathbf{n}_{10}$, are associated with the four corners of a polygon. These vectors are chosen so as to somehow represent the overall surface in which the polygon is embedded. Normal vectors at points within the polygon are computed by interpolating the four corner vectors:

$$\begin{aligned}
\mathbf{n}_r(u,v) &= (1-u)(1-v)\mathbf{n}_{00} + (1-u)(v)\mathbf{n}_{01} + (u)(v)\mathbf{n}_{11} + (u)(1-v)\mathbf{n}_{10} \\
&= (\mathbf{n}_{00} - \mathbf{n}_{01} + \mathbf{n}_{11} - \mathbf{n}_{10})uv + (\mathbf{n}_{10} - \mathbf{n}_{00})u + (\mathbf{n}_{01} - \mathbf{n}_{00})v + \mathbf{n}_{00} \\
&= \mathbf{n}_a uv + \mathbf{n}_b u + \mathbf{n}_c v + \mathbf{n}_d.
\end{aligned}$$

At this stage, the interpolated vector could be perturbed by some function of u and v in order to perform bump mapping. Finally, although the direction of this interpolated vector is correct, it must be normalized before it can be used in the shading computations as described above.

2.3 A Ray Tracing Peripheral

As mentioned earlier, a ray tracing algorithm is not a particularly speedy way to make a picture. In fact, Whitted and Rubin have quoted CPU times in excess of an hour just to generate a single frame [WHIT80, RUBI80]. Whitted also reported that between 75% and 95% of this time was spent finding the intersections of rays and surfaces. These findings suggest that some hardware support for the intersection computations could dramatically improve the running time of a ray tracing algorithm.

The ray tracing processor proposed in this section acts as one of the peripheral devices of a medium-sized host computer. Its primary task is to find intersections between the rays and polygons supplied by the host machine. The ray tracing peripheral has its own copy of the scene model, not only to reduce the load on the host's memory, but also to permit the model to be organized in a way that is suitable for the intersection computations; the host, however, retains control of the ray tracing algorithm. After the scene model has been loaded into the peripheral processor, communications between the host and the peripheral consist of rays to be traced and the results of those requests. This division of labor is convenient for two reasons. First, since the peripheral processor is not burdened with the control of the ray tracing algorithm, its data paths may be more easily optimized for computing intersections. Second, the algorithm can be very flexible because it is implemented as a program in the host machine, yet its performance is acceptable because the expensive computations are implemented in hardware.

2.3.1 Host-Peripheral Interaction

To understand how the host computer and ray tracing peripheral cooperate to make pictures, it is helpful to examine the individual steps required in the process of image generation. First of all, the host must prepare the scene model in a suitable form and transfer it to the peripheral device. Ray tracing is an object space algorithm, which means that it operates on a model expressed in a full, three-dimensional coordinate system. In particular, the model representation is independent of viewpoint, so the host can generate several views of the same scene without reloading the model. When the model does have to be modified, only those surfaces that actually change need be sent to the peripheral. This property is useful for making movies depicting motion through a relatively fixed scene.

After loading the surface model, the host computer can begin to generate the image. For each pixel on the screen, it selects the corresponding vision ray emanating from the viewing position. It then sends a specification of this ray to the ray tracing peripheral. At some later time, the peripheral responds by reporting whether this ray intersects any polygon in the scene model. If an intersection has occurred, the peripheral also reports the identity of the intersecting polygon that is closest to the

viewing position. Notice that this polygon is the one that would be visible along the original ray. Additionally, the peripheral computes the three-dimensional spatial coordinates and two-dimensional *uv*-coordinates of the intersection point, and the distance from the viewing position to the point of intersection.

After having retrieved the result of the ray tracing operation, the host machine computes the color of the pixel. This computation may in turn require more rays to be traced. For example, if the original ray struck a reflective or refractive surface, the host uses the polygon identity, the point of intersection, and the normal vector at the intersection point to generate a new ray. It then passes this new ray to the ray tracing peripheral for processing. If shadowing is being modeled, the point of intersection and the direction to the light source combine to form still another new ray. In this case, however, the ray is marked as a shadow ray before it is transferred to the peripheral, since the exact identity of the polygon occluding the light source is unimportant and the peripheral can save some time by not attempting to find the closest such polygon. The host computer determines pixel colors on the basis of the results of the ray tracing operations together with additional information about the scene model, possibly including surface coloration and texture information that was never passed to the peripheral device. Once computed, the pixel color may be written to a file on disk, or stored directly in a frame buffer for immediate display.

One way to view the interaction between the host machine and the ray tracing peripheral is to regard the peripheral as a hardware subroutine of a program running in the host. Thus, whenever the ray tracing algorithm would otherwise invoke a subroutine to do an intersection computation, it now invokes a computation in another processor. This view of the situation is, however, somewhat inadequate, because it suggests that the host machine must wait while the peripheral completes its task, thereby missing an opportunity for exploiting parallel operation. A better way to utilize the hardware is to maintain queues between the host and the peripheral. The first queue contains rays to be traced. These are supplied by the host and removed by the peripheral as it becomes idle. The second queue buffers results transmitted in the opposite direction. In order to make use of these queues, the host machine would be multiplexed between two tasks. The first task generates vision rays that correspond to pixels in the final image, and it runs whenever the host would otherwise be idle. The second task processes the results of the ray tracing requests, possibly generating new requests itself. It is this second task that computes actual pixel colors.

The interesting feature of the two-task ray tracing algorithm just described is that it is completely asynchronous. There is no way to predict the exact order in which pixel colors will be computed. This works because the rays are mutually independent; therefore, the order in which they are processed does not affect the outcome of the computation. In this case, only two processors, the host machine and the ray tracing peripheral, are working at the same time, but the fact that rays can be traced independently and in any order offers a foothold for even greater concurrency. Machines capitalizing on this extra concurrency are the subjects of later sections. For

now, let us consider only the parallelism that can be introduced within the ray tracing peripheral itself.

2.3.2 Operation of the Peripheral

The main task of the ray tracing peripheral, as mentioned earlier, is to find intersections of rays and polygons. It must also determine which of the polygons intersecting a given ray is closest to the viewing position. There are many ways of organizing the peripheral to solve this problem, and the approach taken here is one of the most straightforward. Upon receiving a ray from the host, the peripheral attempts to intersect the ray with every polygon in the scene. After doing so, it examines all of the intersections that it found and selects the one closest to the source of the ray. This means, of course, that every ray needed to make the picture must potentially be intersected with every surface in the scene model. Thus, the intersection processor must be very fast indeed if it is to perform this rather substantial task at an acceptable rate.

Fortunately, the operation of intersecting rays and polygons is sufficiently simple and regular that it may be pipelined in order to raise the throughput. The pipe can be thought of as having three stages, each of which may be pipelined internally; refer to Figure 2–11. The first stage fetches successive polygons from a scene model memory and passes their representations to the second stage, which performs the actual intersection computation. The third stage examines each new intersection and discards all but the one that is closest to the origin of the ray. After every polygon in the scene has passed through the pipeline, the final stage of the pipeline contains the result that should be returned to the host machine.

Clearly, most of the work required of the ray tracing peripheral is concentrated in the intersection stage of the pipeline. This stage may conveniently be pipelined internally so as to increase its performance. The three stages of the expanded intersection pipeline correspond to the three steps of the intersection computation that was described in the previous section and is summarized in Figure 2–12. The resulting pipeline is diagrammed in Figure 2–13. In the first stage, the processor determines the value of the ray parameter t at the point where the ray and polygon plane intersect. In the second stage, it computes the three-dimensional coordinates of the intersection point $\mathbf{p}_r$. Finally, in the third stage, it finds the parameter values u and v, which describe the position of the intersection point with respect to the polygon.

Within each stage of this expanded pipeline, still more concurrency can be exploited. Consider first the expression for t, which consists of the ratio of two inner products. The numerator and denominator of this ratio may be computed at the same time. Further concurrency is available within each inner product operation, since the three required multiplications may occur simultaneously, and the additions may be pipelined. Thus, the computation of t can be expanded into a four-stage pipeline:

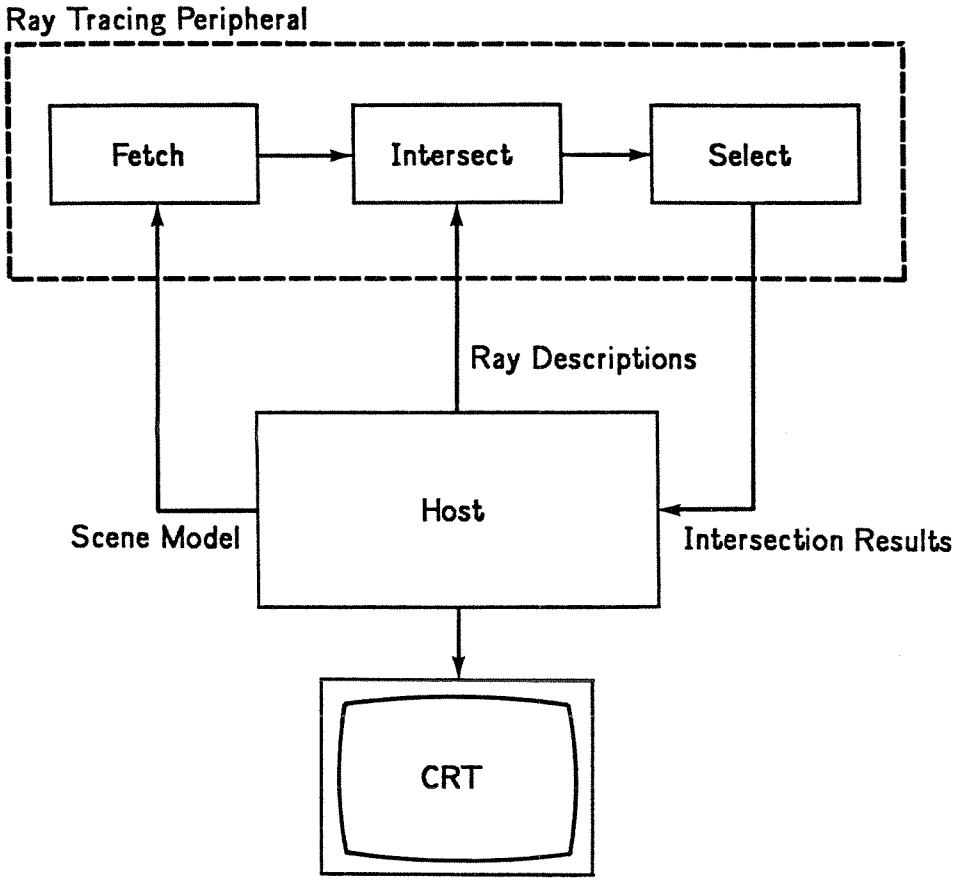


Figure 2–11. The three major pipeline stages in the ray tracing peripheral.

$$
\begin{aligned}
\text{Stage 1:}\quad & q_1 \leftarrow n_{px} r_{0x}, \quad q_2 \leftarrow n_{py} r_{0y}, \quad q_3 \leftarrow n_{pz} r_{0z}, \\
& q_4 \leftarrow (r_{0x} - r_{1x}) n_{px}, \quad q_5 \leftarrow (r_{0y} - r_{1y}) n_{py}, \quad q_6 \leftarrow (r_{0z} - r_{1z}) n_{pz}. \\
\text{Stage 2:}\quad & q_7 \leftarrow q_1 + q_2, \quad q_8 \leftarrow q_3 + d_p, \\
& q_9 \leftarrow q_4 + q_5. \\
\text{Stage 3:}\quad & q_{10} \leftarrow q_7 + q_8, \\
& q_{11} \leftarrow q_9 + q_6. \\
\text{Stage 4:}\quad & t \leftarrow q_{10}/q_{11}.
\end{aligned}
$$

The required devices are shown in Figure 2–14. Next, the value of t is used to find the point $\mathbf{p}_r$. Note that each of its three components is independent of the others, so that all may be computed at the same time. Furthermore, pipelining may be employed within each of the individual component computations, as diagrammed in Figure 2–15. Finally, the expressions for u and v also involve dot products, and Figure 2–16 shows that concurrency is possible here as well.

$$t = \frac{\mathbf{n}_p \cdot \mathbf{r}_0 + d_p}{\mathbf{n}_p \cdot (\mathbf{r}_0 - \mathbf{r}_1)}$$

$$\mathbf{p}_r = (\mathbf{r}_1 - \mathbf{r}_0)t + \mathbf{r}_0$$

$$u = \begin{cases} -\left[\left(\frac{\mathbf{n}_a}{2d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u1}}{2d_{u2}}\right)\right] \\ \quad \pm \sqrt{\left[\left(\frac{\mathbf{n}_a}{2d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u1}}{2d_{u2}}\right)\right]^2 - \left[\left(\frac{\mathbf{n}_c}{d_{u2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{u0}}{d_{u2}}\right)\right]} & \text{if } d_{u2} \neq 0 \\ -\dfrac{\mathbf{n}_c \cdot \mathbf{p}_r + d_{u0}}{\mathbf{n}_a \cdot \mathbf{p}_r + d_{u1}} & \text{if } d_{u2} = 0 \end{cases}$$

$$v = \begin{cases} -\left[\left(\frac{\mathbf{n}_a}{2d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v1}}{2d_{v2}}\right)\right] \\ \quad \pm \sqrt{\left[\left(\frac{\mathbf{n}_a}{2d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v1}}{2d_{v2}}\right)\right]^2 - \left[\left(\frac{\mathbf{n}_b}{d_{v2}}\right) \cdot \mathbf{p}_r + \left(\frac{d_{v0}}{d_{v2}}\right)\right]} & \text{if } d_{v2} \neq 0 \\ -\dfrac{\mathbf{n}_b \cdot \mathbf{p}_r + d_{v0}}{\mathbf{n}_a \cdot \mathbf{p}_r + d_{v1}} & \text{if } d_{v2} = 0 \end{cases}$$

Figure 2–12. Summary of the intersection computation.

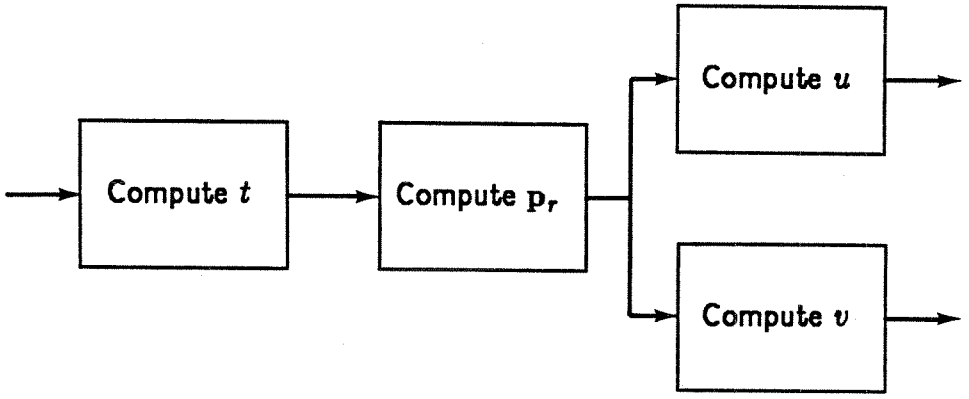


Figure 2–13. A three stage pipeline to perform the intersection computation.

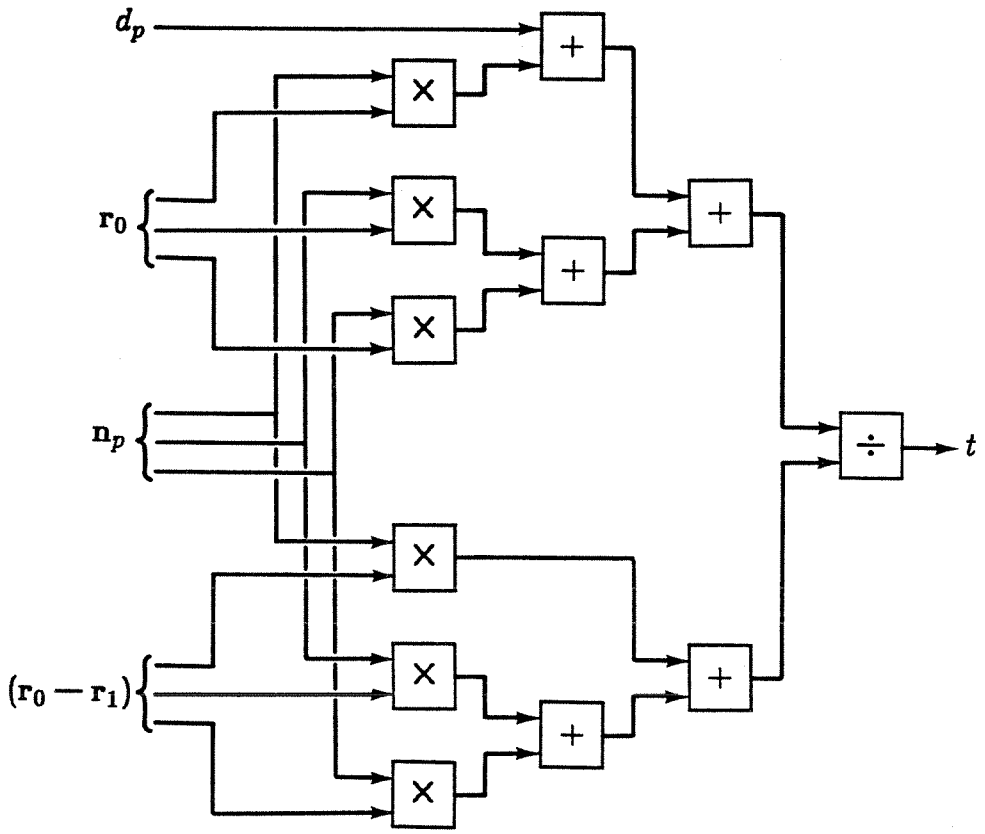


Figure 2–14. A four stage expanded pipeline to compute t. Note that the delays required in an actual implementation are not shown.

A potential difficulty with the pipelined intersection computation arises in connection with the division operation, since the result may be undefined for some values of its inputs. One way of handling this situation might be to empty the pipe when an undefined value arises, but even if this could be done, it would complicate the pipeline. A better solution is to associate a validity bit with each intermediate result flowing through the pipe. By convention, operations in the pipeline will always produce a result, but they will mark that result to indicate whether it is somehow erroneous and should therefore be ignored. Although later stages in the pipeline will accept these nonsense values as if they were meaningful, the fact that their own results must also be invalid will be reflected in the validity bit of the output. Thus, even though some stages of the pipeline may not always compute useful values, the flow of information

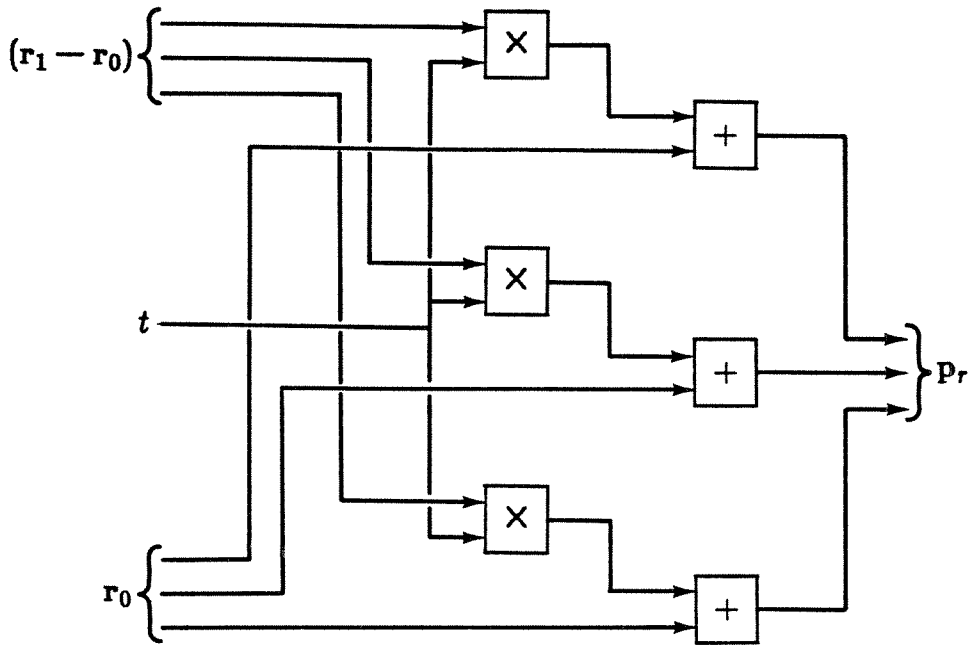


Figure 2–15. A two stage pipeline to compute the value of p_r given t. The three component computations proceed in parallel.

through the pipe is not disrupted. Finally, the last stage of the pipeline, which isolates the intersection closest to the origin of the ray, must take into account the settings of validity bits as it makes its determination.

The validity bit was introduced to correct a problem with the pipelined organization of the intersection processor, but it turns out to be useful for other aspects of the design as well. To begin with, the host machine can supply initial values for validity bits to distinguish the parallel and non-parallel cases in the computation of u and v. If these bits are injected at the proper point within the pipeline, they can serve to extinguish the inapplicable branch of the computation. Next, recall that u and v must each lie in the range from zero to one if the corresponding intersection point is to be within the polygon. If the validity bit is cleared when these boundary conditions are violated, the final stage of the pipeline will not bother to consider the corresponding intersection point when searching for the one closest to the origin of the ray.

Now that the intersection computation has been presented, it is easy to see how the first stage of the pipeline operates. This is the stage that supplies polygon parameters to the intersection computation. From the expressions given above, it is apparent that polygons are described by five vector and five scalar values:

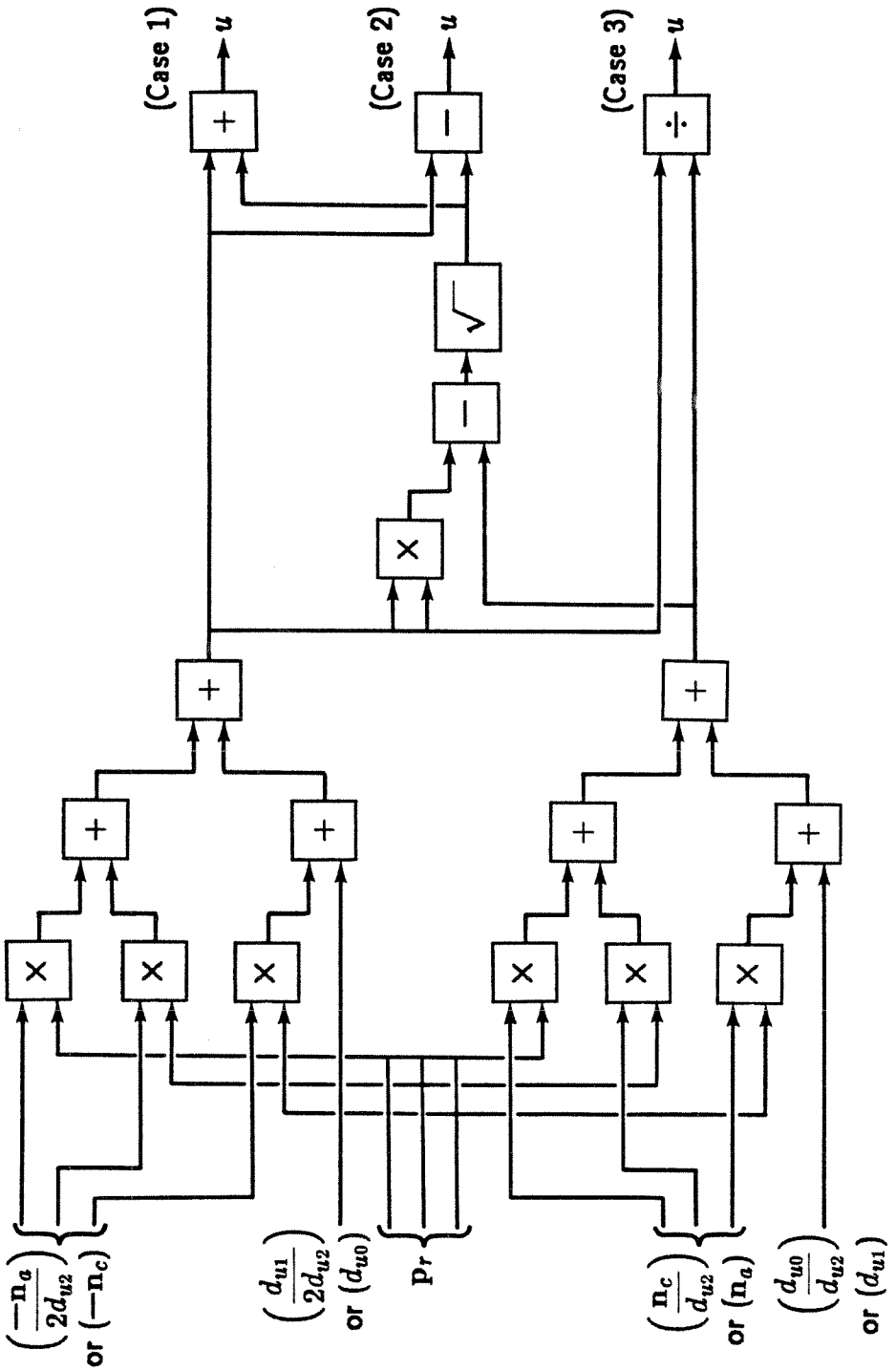


Figure 2–16. A seven stage pipeline to compute the value of u given p_r, shown without delays. The three values generated correspond to the solution to the parallel case and the two solutions to the non-parallel case. The computation of v is, of course, completely analogous.

$$\mathbf{n}_p, \quad d_p,$$

$$\mathbf{n}_a \text{ or } \frac{\mathbf{n}_a}{2d_{u2}}, \quad d_{u1} \text{ or } \frac{d_{u1}}{2d_{u2}}, \quad \mathbf{n}_c \text{ or } \frac{\mathbf{n}_c}{d_{u2}}, \quad d_{u0} \text{ or } \frac{d_{u0}}{d_{u2}},$$

$$\mathbf{n}_a \text{ or } \frac{\mathbf{n}_a}{2d_{v2}}, \quad d_{v1} \text{ or } \frac{d_{v1}}{2d_{v2}}, \quad \mathbf{n}_b \text{ or } \frac{\mathbf{n}_b}{d_{v2}}, \quad d_{v0} \text{ or } \frac{d_{v0}}{d_{v2}}.$$

Each of these twenty polygon parameters is stored in one of twenty independent memories so that all may be accessed simultaneously. Consecutive memory locations hold the parameter values for successive polygons, and the various parameters of a particular polygon are stored at the same address within each of their corresponding memories. Were it not for this parallel access, the values could not be retrieved quickly enough to keep the later pipeline stages full. Notice also, that since the host machine must supply these parameters in order to define the scene model, some sort of dual-porting arrangement must be used to load these memories. It does not have to be extraordinarily fast or complex, because the host does not need to load the scene model at the same time that rays are being traced.

When fully pipelined as outlined above, the intersection computation consists of thirteen stages. This extensive pipelining, combined with the parallelism that is inherent within each stage, makes it possible to perform fifty-four arithmetic operations at once. Also, the structure of the pipeline itself is very modular. The individual devices that perform addition, multiplication, division, and square root operations are connected in a quite straightforward manner to implement the intersection computation.

After the ray tracing peripheral has found the first polygon struck by a ray, it is up to the host processor to apply a lighting model at the point of intersection. Although the host could certainly perform this computation without assistance, the addition of a little extra hardware in the peripheral will accelerate it. Because lighting computations occur less frequently than intersection computations, the corresponding hardware need not be as fast or as extensively pipelined as the intersection processor. Instead of allocating hardware wherever it is possible to take advantage of concurrency in a computation, as was done in the intersection part of the peripheral, the data paths in the extension will be flexible enough to be reused for several of the computations required in the lighting model. To see how this can be done, it is helpful to briefly review the form of these computations.

The first result needed in the lighting computation is the surface normal vector at the point of intersection. Recall that the normal is interpolated from four precomputed vectors:

$$\mathbf{n}_r = \mathbf{n}_a uv + \mathbf{n}_b u + \mathbf{n}_c v + \mathbf{n}_d.$$

The three components of this vector can each be computed separately. For example, the x component is given by

$$n_{rx} = n_{ax} uv + n_{bx} u + n_{cx} v + n_{dx},$$

where n_{ax} is the x component of $\mathbf{n}_a$, and likewise for n_{bx}, n_{cx}, and n_{dx}. Notice that this expression is the inner product of the vectors $[n_{ax}\ n_{bx}\ n_{cx}\ n_{dx}]$ and $[uv\ u\ v\ 1]$, and it therefore takes three inner product operations to find the three vector components. After $\mathbf{n}_r$ has been computed, it must still be normalized before it can be used in the subsequent expressions. Normalizing a vector is accomplished, of course, by dividing each of its components by its magnitude, and the square of a vector's magnitude is the inner product of that vector with itself. The two cosine terms in the lighting model may also be expressed as inner products:

$$\cos i = \mathbf{n}_r \cdot \mathbf{l}$$
$$\cos s = (-\mathbf{l} + 2\mathbf{n}\cos i) \cdot \left(\frac{\mathbf{r}_0 - \mathbf{r}_1}{\|\mathbf{r}_0 - \mathbf{r}_1\|} \right).$$

Note that in the latter case, still another inner product operation is needed for normalizing a vector. Thus, it seems that inner products are central to the computation of the lighting model, and they will therefore be central to the organization of the lighting portion of the ray tracing peripheral as well.

The lighting processor consists mainly of a pipelined inner product device surrounded by enough interface logic to perform the lighting computations under the direction of the host computer. The three multipliers in the inner product device may also be used to scale a vector. Similarly, the three adders can be used to form the sum of two vectors. A device for determining the reciprocal of a number's square root is included for use in vector normalization. Four separate memories hold the components of the $\mathbf{n}_a$, $\mathbf{n}_b$, $\mathbf{n}_c$, and $\mathbf{n}_d$ vectors in order that these values need not be communicated between the host machine and the peripheral for every ray that is traced. Instead, the host machine transmits them along with the polygon descriptions as part of the scene model. These memories can also hold intermediate vector results that are found in the course of the lighting model computation.

The settings of the multiplexors and the clocking of the latches in the lighting portion of the ray tracing peripheral are rather tightly controlled from the host machine. Including a microcode engine in the peripheral to relieve the host of this responsibility would not be too difficult, but doing so would limit the flexibility of the device. Besides controlling data movement in the processor, the host may examine and modify values in its memory as required to implement a lighting model. This capability is necessary, for example, to transfer the results of a mapping operation to the peripheral where they can take part in the lighting computation.

2.3.3 Implementation of Arithmetic

Deciding how numbers should be represented is one of the more difficult aspects of designing a graphics processor like the one proposed here. Floating point numbers are ideal from the standpoint of the final user. For a given word size, floating point rep-

Multiplication	1
Addition	1 or 3
Long Division	4 or 6
Short Division	2
Square Root	2

Figure 2–17. Number of pipeline stages required for arithmetic operations. Note that addition may be implemented in one of two ways, and long division uses addition. The result of a short divide has fewer bits of precision than the result of a long divide.

resentations have a far greater dynamic range than fixed point representations, freeing the user from having to pay so much attention to scaling. On the other hand, designers of processors often consider a fixed point representation to be easier to implement, although compromises are possible. For example, the Evans & Sutherland PS-300 uses a block floating point representation for vectors, where the individual components of a vector have separate mantissas but share a common exponent [EVAN81]. Purely fixed point representations are not without problems, however. In order to insure an adequate range of values, the word size must often be fairly large. This increases the widths of the data paths, slows down the arithmetic operations, and makes them more expensive to implement.

The ray tracing peripheral will use a floating point representation for all numbers. This decision was motivated by two considerations. First, floating point makes the peripheral much easier to use, simplifying the programs in the host computer as well as easing the modeler's task. Second, a device made by TRW reduces the cost and complexity of implementing floating point operations. The device is a single chip parallel multiplier capable of producing a 48-bit product from two 24-bit operands in a maximum of 285 nanoseconds [TRW78]. Because of this device, it is convenient to use a floating point representation with a 24-bit fractional part. The exponent will be represented in eight bits.

Using the TRW multiplier, a floating point multiplication takes about a third of a microsecond, but the other floating point operations cannot be completed so quickly. Each of them may, however, be pipelined to operate at the same rate. Since computations in the ray tracing peripheral are pipelined, it is quite natural to pipeline the individual arithmetic operations internally as well. Figure 2–17 lists the number of pipeline stages required for each operation, and a more complete description of floating point implementation may be found in Appendix A. Using this fully pipelined arithmetic, the complete peripheral can produce three results every microsecond.

2.3.4 Analysis

The performance of the ray tracing peripheral described in this section is fairly respectable. Because of the massive pipelining, once the pipe is full it can compute one ray/polygon intersection every one third of a microsecond (μsec). Depending upon the technique used for implementing addition, the pipeline has either 17 or 33 stages, which implies a latency time of 5.6 or 11μsec. By way of comparison, a DECSYSTEM-2060 can perform a similar computation in about 100 to 200μsec. Since the computation involves 23 multiplications, 26 additions, 3 divisions, and 2 square roots, the use of concurrency and massive pipelining enables the ray tracing peripheral to execute 162 million floating point operations per second (MFLOPS) at its peak rate. The Cray-1 can sustain rates of 138MFLOPS, with short bursts of up to 250MFLOPS under certain circumstances [RUSS78]. These sorts of comparisons cannot be taken too seriously, however, because the ray tracing peripheral is so optimized to its single task that it can do nothing else. On the other hand, its cost and complexity are probably comparable to those of a large minicomputer, so it would seem to be a practical device for at least some applications.

The time required to generate a picture using the ray tracing peripheral is a more appropriate consideration than its brute arithmetic performance. Suppose that the scene model consists of a thousand polygons. At three polygons per microsecond, it takes one third of a millisecond to intersect a ray with each of these surfaces. In an image with 512×512 pixels of resolution, it takes about a minute and a half to trace one ray per pixel. Of course, this time does not include any special lighting effects. About twice as many rays must be traced to model shadows, for example, thus doubling the picture generation time. The anti-aliasing procedure increases the time by another factor of two or three. In the end, it takes nearly ten minutes to finish the picture of a thousand polygon scene. Note that the time is linearly dependent on the number of surfaces in the scene, and consequently, increasing the number of polygons by some factor would boost the execution time by the same amount.

An unappealing aspect of the ray tracing peripheral is that it must intersect every ray with every polygon in the scene. This seems rather wasteful when one observes that surfaces occurring in one part of the image have little effect on the appearance of other parts. The problem can be eased, however, with the addition of a little extra hardware. The basic idea is to superimpose a three-dimensional grid on the volume that contains the scene model, thereby partitioning the modeling space into a number of rectangular parallelepipeds. Each of these smaller volumes is associated with a list of surfaces that intersect the volume. Using this representation, the ray tracing peripheral can confine its attention to only those surfaces that occur in subvolumes through which the ray actually passes. Furthermore, since only the first intersection is needed, if the subvolumes containing the ray are processed in order along the length of the ray, it may be possible to bypass some of them entirely.

The improved algorithm can be implemented with a small processor that fits between the host computer and a slightly modified ray tracing peripheral. Upon receiving a ray from the host machine, this new processor passes the ray on to the intersection processor, and it determines which scene subvolume contains the origin of the ray. Next, the auxiliary processor supplies the ray tracing peripheral with a sequence of polygon descriptor addresses corresponding to surfaces that intersect the subvolume. These addresses identify polygons by giving the location of their parameters in the memory of the ray tracing peripheral. The ray being processed either will or will not intersect one of the polygons in the current subvolume. If it does, and if the point of intersection lies within the subvolume, then the proper intersection point has been found. Otherwise, if there is no intersection point within the first subvolume, the auxiliary processor must generate the addresses of polygons that intersect the next subvolume along the length of the ray. This procedure continues either until an intersection point is found or until the ray travels out of the image space.

The auxiliary processor could be simulated in software running on the host computer, but this would be slow, and furthermore, the task of the auxiliary processor is simple enough that a hardware implementation is feasible. To find the initial subvolume, it examines the coordinates of the origin of the ray; succeeding subvolumes can be determined by an incremental computation. Both of these computations are described more fully in Appendix B. The auxiliary processor has a local memory, which is partitioned by subvolume, for storing polygon addresses. After identifying the subvolume to examine, it locates the appropriate partition and sequentially retrieves the addresses of polygons stored within the ray tracing peripheral. Thus, its local memory provides a level of indirection for translating subvolume indices into polygon addresses. It would be possible to eliminate this indirection by allocating a different region of memory in the intersection processor for each subvolume, but doing so would substantially increase memory requirements because an individual polygon might intersect several subvolumes and would have to be stored redundantly. However, since the additional level of indirection may be regarded as yet another pipeline stage, its presence does not hinder the overall throughput of the ray tracing peripheral.

Subdivision techniques similar to the one proposed here have been reported elsewhere. Perhaps the most similar one was developed by Franklin as part of a linear time hidden surface algorithm [FRAN80]. Franklin's algorithm worked in the image space, and a two-dimensional grid was superimposed on the screen. Briefly, the polygons were sorted into the bins indicated by the grid, and the bins were processed independently. Rubin and Whitted suggested another technique in the context of ray tracing [RUBI80]. They associated an arbitrarily oriented rectangular parallelepiped with each object to be displayed. The parallelepiped serves as a bounding volume for the object, and the technique is applied recursively for sub-objects within the volume. Another method, described by Reddy and Rubin, divides the modeling space into the eight subvolumes formed by the three planes that pass through the center of the modeling space and are orthogonal to the three coordinate axes [REDD78]. Each of

the resulting subdivisions may be further divided in a similar manner until the number of surfaces contained within each subvolume is reduced to some manageable value.

It would not be particularly difficult to implement either of these other ray tracing partitioning schemes with the ray tracing peripheral. Each involves checking a ray to determine whether it passes through a bounding volume before actually examining the contents of the volume. This check can be implemented with the intersection processor because a ray that passes through a bounding parallelepiped must intersect at least one of the parallelepiped's six faces. The check thus requires six surface intersection computations, and would take only two microseconds when pipelined. The auxiliary processor for implementing the grid subdivision algorithm as described earlier requires only minor modifications in order to handle these other forms of subdivision. Indeed, a simple change of microcode would almost certainly suffice.

The performance of the three-dimensional grid subdivision technique suggested here is rather difficult to analyze because it depends so heavily on the scene being rendered; however, some general observations are possible. Note that when the grid resolution is made arbitrarily fine, resulting in infinitesimal subvolumes, most subvolumes will either be empty or will contain a single surface. Subvolumes enclosing surface intersections will contain more than one surface, but this case can be ignored for the present. As a ray is traced, it will pass through a sequence of empty subvolumes before arriving at a subvolume that contains a single surface. Thus, if enough partitions are used, the subdivision technique will reduce to one the number of surfaces that must considered for intersection with a ray. This property is similar to that exhibited by three-dimensional frame buffers, which break the modeling space into small, cubic volume elements, or "voxels" [ATHE81]. Although it executes fairly rapidly, this latter approach has the disadvantage that inaccuracies are introduced because the voxels are merely small, not truly infinitesimal.

There are two factors that limit how finely it is practical to partition the modeling space using the grid subdivision technique. The first and more obvious is the storage required for representing each subvolume. Smaller subvolumes are more numerous, and every subvolume, whether empty or not, uses some minimal amount of memory. Additionally, individual polygons intersect more small subvolumes than they do large subvolumes. Each appearance of a polygon in a subvolume consumes another increment of memory. The second limiting factor is the time required to trace a ray through an empty subvolume. If the grid that subdivides the modeling space is too small, the ray tracing peripheral will have to skip many empty subvolumes before coming to one that encloses some surfaces. There is a danger that the peripheral will spend an excessive proportion of its time examining empty subvolumes, while spending very little time actually performing intersection computations. The grid size must therefore be chosen to balance these two tasks.

It is possible to use some rather crude approximations in order to characterize the behavior of the ray tracing peripheral with various grid sizes. Define n_s to be the number of regions into which each coordinate axis of the modeling space is

partitioned. There are thus n_s^3 individual subvolumes according to this definition. If the average distance that a ray travels before striking a surface is d_r, then the number of subvolumes encountered in that distance can be estimated to be

$$n_d = (n_s - 1)d_r + 1.$$

That is, the number of subvolumes that must be inspected in the process of tracing a ray grows linearly with the number of subvolumes along each coordinate axis. The value of d_r is, of course, heavily dependent upon the scene, but it does not depend on n_s because the same rays must be traced no matter how extensively the modeling space has been subdivided. Next, to determine how many of the examined subvolumes actually contain polygons, suppose that there are n_o occupied subvolumes distributed uniformly throughout the modeling space. For each ray, the number of examined subvolumes that actually contain polygons is

$$n_c = (n_d - 1)\frac{n_o}{n_s^3} + 1.$$

Note that the final subvolume along a ray always holds polygons, because if it did not, the ray would have continued. If the total number of polygons in the scene is n_p, then it is necessary to perform n_i intersection computations for each ray, where

$$n_i = \left(\frac{n_p}{n_o}\right)n_c.$$

The value of n_o is still unknown.

To derive an expression for n_o, the number of subvolumes occupied by polygons, consider the case where the scene consists of just one polygon. If the area of this polygon is a_p, then

$$n_o = \left[(n_s - 1)\sqrt{a_p} + 1\right]^2.$$

Thus, the number of occupied subvolumes grows linearly with the size of the polygon, but quadratically with the number of subdivisions. One way to extend this expression to a scene containing many polygons might be to treat a_p as an average polygon size and to scale the result by the total number of polygons:

$$n_o = n_p\left[(n_s - 1)\sqrt{a_p} + 1\right]^2.$$

This expression, however, produces a highly inflated value because it assumes that every polygon will be alone in its own subvolume. In actual scenes, polygons are usually clumped together as components of larger surfaces. Another approach might therefore be to combine all of the polygons into a single, giant polygon with area $n_p a_p$:

$$n_o = \left[(n_s - 1)\sqrt{n_p a_p} + 1\right]^2.$$

Unfortunately, this estimate is also excessive. One problem is that it tends to place too much emphasis on small polygons that merely add detail to larger surfaces. This kind of detail, which is fine compared to the size of the subdivision, should not contribute

to the total area of the scene. Using a_t as an estimate of the overall surface area, overlooking surface detailing, yields the following estimate for the number of occupied subvolumes:

$$n_o = \left[(n_s - 1)\sqrt{a_t} + 1\right]^2.$$

Notice that even this expression is not quite correct, because in reality the relevant surface area will depend on the number of subdivisions. As the grid becomes finer, the level of detail to be included in the area estimate becomes greater. Because it is so dependent upon the characteristics of the scene, however, this additional refinement will be omitted.

The time required to trace a single ray through the scene is

$$t_r = n_d t_d + n_i t_i + t_l,$$

where t_d is the time needed to move from one subvolume to the next, t_i is the time it takes to intersect a ray with a polygon, and t_l is the pipeline latency. In terms of the number of subdivisions n_s, this expression becomes

$$t_r = \left[(n_s - 1)d_r + 1\right]t_d + \left((1 - n_s^{-1})d_r n_s^{-2} + \left[(n_s - 1)\sqrt{a_t} + 1\right]^{-2}\right)n_p t_i + t_l.$$

Notice that the overall intersection time decreases quadratically with the number of subvolumes, but the time spent stepping between subvolumes increases linearly. There will thus be some degree of subdivision that produces the minimal computation time. The ray tracing peripheral takes a third of a microsecond to intersect a ray with a polygon, and it has a pipeline latency of 5.6μsec. Assume further that rays travel through about half of the modeling space on the average and that it takes one microsecond to traverse a subvolume. Figure 2–18 shows, for various degrees of subdivision, the time required to trace a ray through a scene that contains a thousand polygons and has enough surface area to stretch through the modeling space twice. The minimal time of 14.3μsec occurs at eleven subdivisions per axis, and it is nearly twenty-four times faster than the third of a millisecond that it takes to process a ray without using the subdivision technique. Thus, the picture that would have taken ten minutes without subdivision now requires less than thirty seconds.

The expression for estimating the ray tracing time may be used as a guide when deciding how extensively to subdivide a particular scene. Notice that the curve of Figure 2–18 is much steeper to the left of the minimum time than it is to the right. It is therefore less costly to overestimate the number of subdivisions than it is to underestimate them. Notice also that as the number of subvolumes increases, the total ray tracing time is dominated by the overhead of moving from one subvolume to the next, and the intersection times become negligible. This observation suggests that the total polygon area a_t should be tuned for larger subvolumes. That is, it should not reflect surface detailing that would become important only with very fine subdivision.

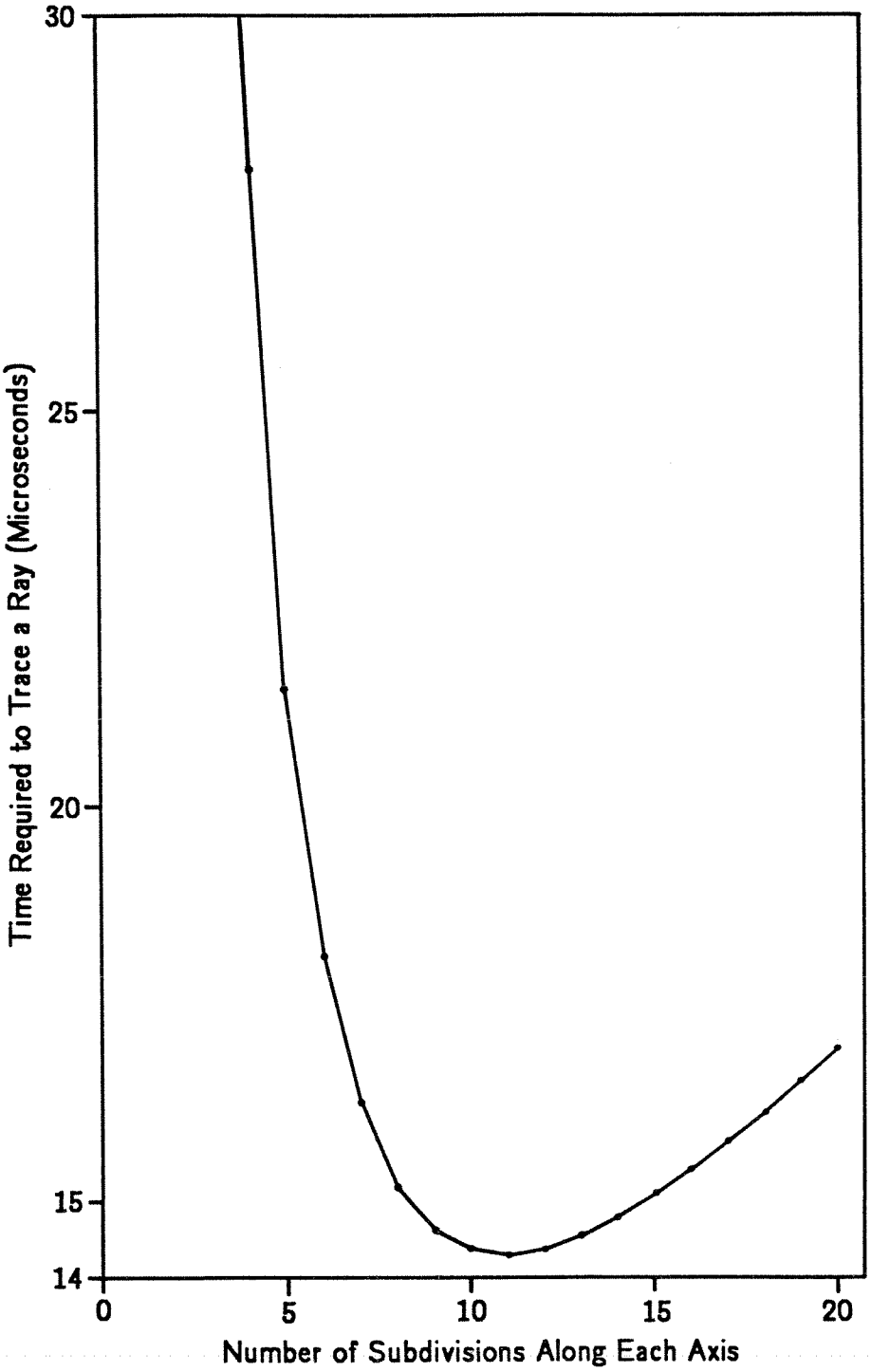


Figure 2–18. Estimated ray tracing time for various degrees of subdivision. The hypothetical scene has a thousand polygons, with total area $a_t = 2$ and average ray travel distance $d_r = \frac{1}{2}$.

Simulation results seem to confirm at least the general character of the performance estimations given above. Figure 2–19 shows times that were computed in the process of forming an actual image. The scene consisted of eighty-three randomly oriented cubes distributed uniformly throughout the modeling space. Figure 2–20 shows the average number of subvolumes visited by a ray for various scene complexities. Notice that as more polygons are added to a scene, the average distance traveled by each ray may actually decrease. Figure 2–21 illustrates how the number of intersection computations per ray falls off as subdivisions become finer and finer. Finally, Figure 2–22 shows the minimal ray tracing times for various scene complexities.

(Text resumes on page 46.)

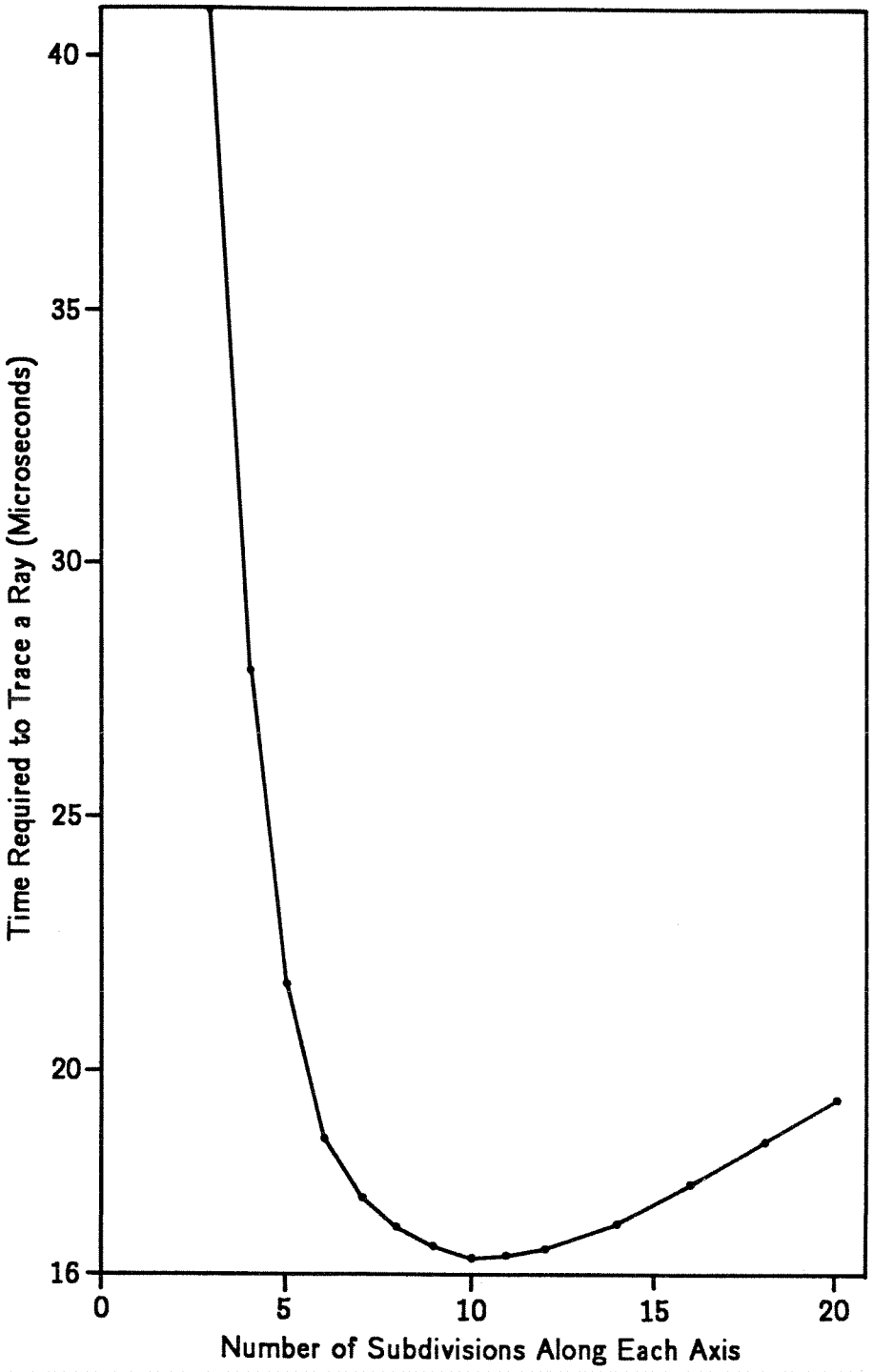


Figure 2–19. Simulated time to trace a ray for various degrees of subdivision. The scene consisted of eighty-three uniformly distributed and randomly oriented cubes.

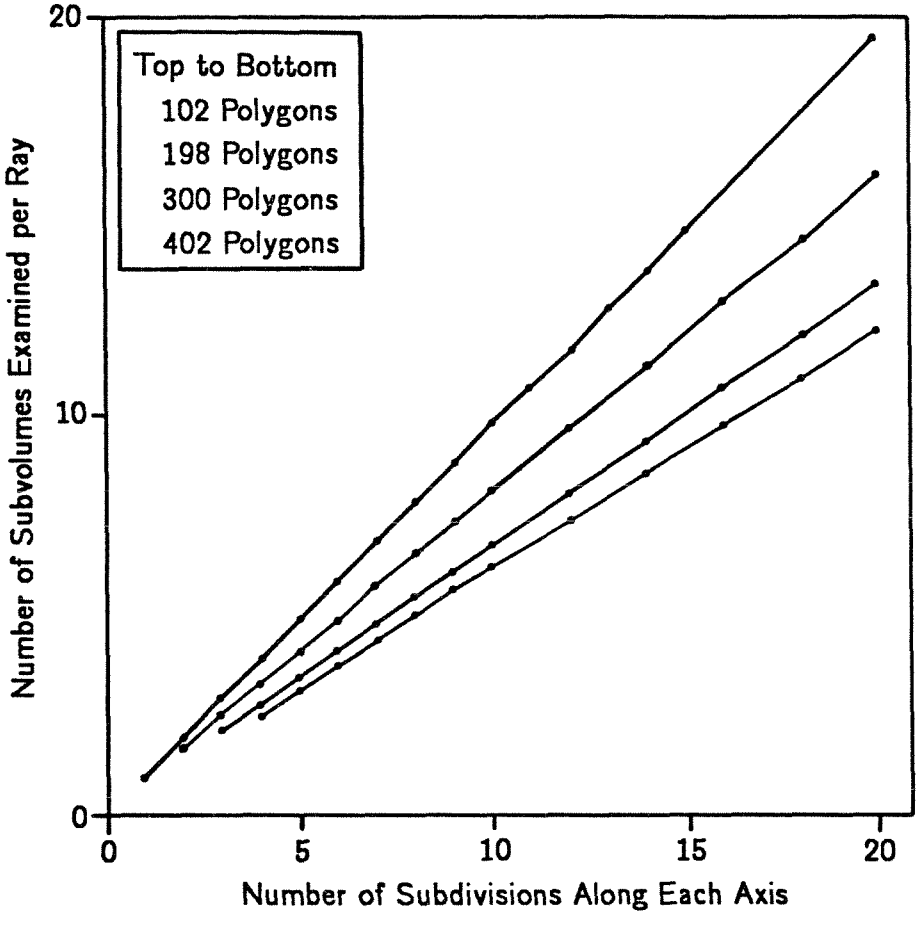


Figure 2–20. Simulated number of subvolumes visited while tracing a ray, plotted against degree of subdivision for various scene complexities.

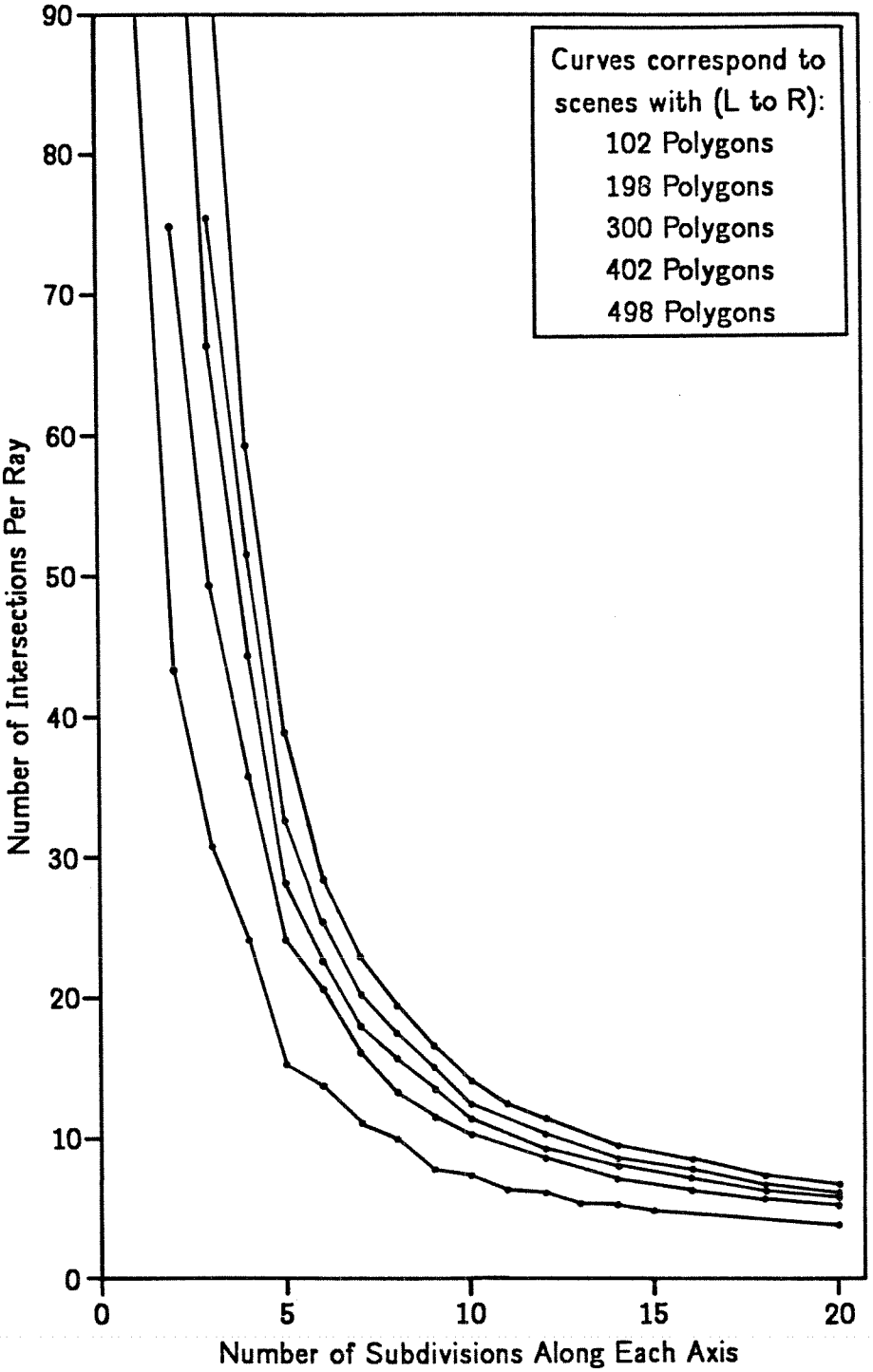


Figure 2–21. Simulated number of intersection computations per ray, plotted against degree of subdivision for various scene complexities.

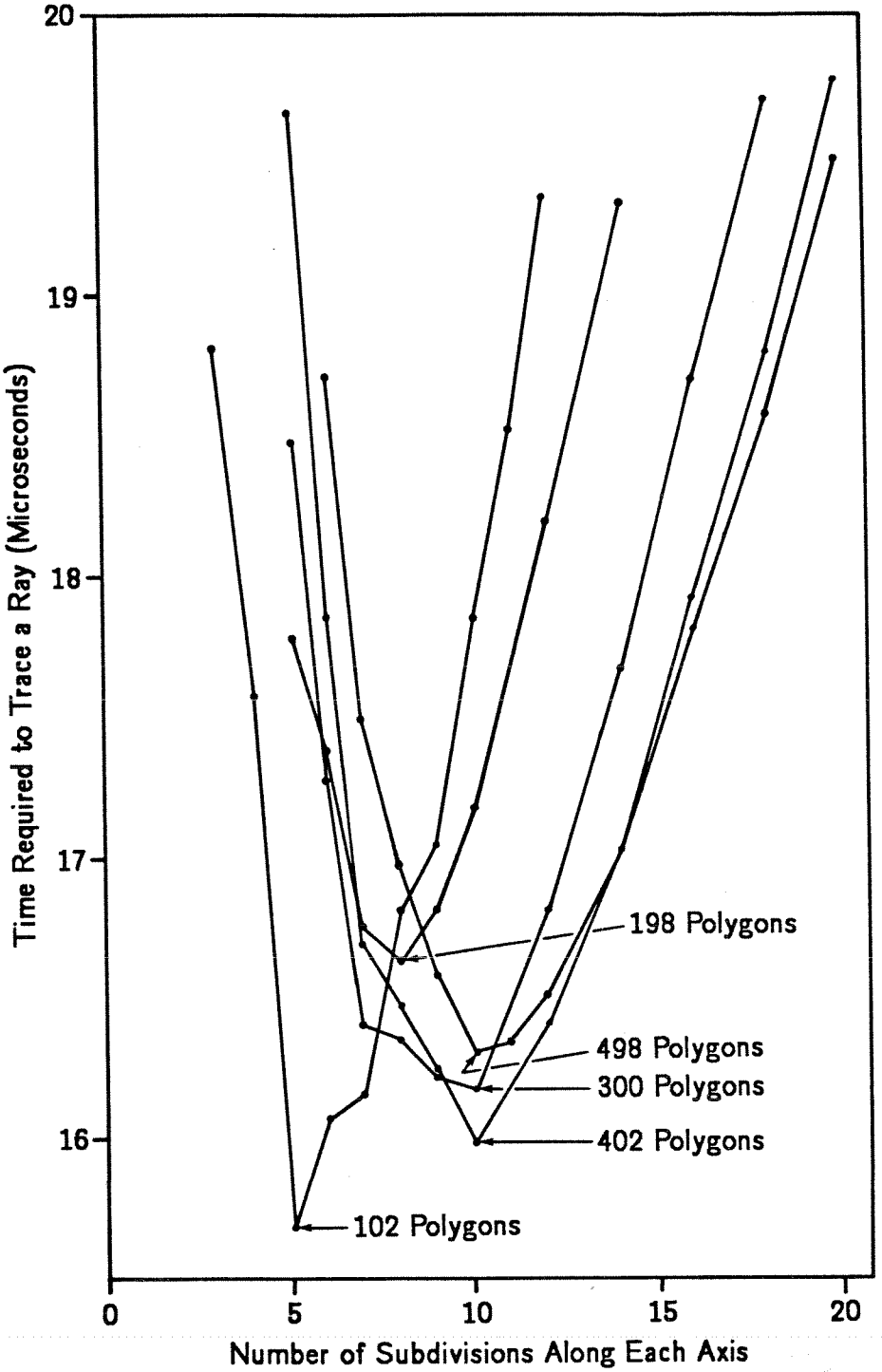


Figure 2–22. Simulated ray tracing time, plotted against degree of subdivision for various scene complexities. The scale has been expanded to show more detail near the minima.

2.4 A Ray Tracing Pipeline

The ray tracing peripheral described in the previous section is useful in many situations, but it does have limitations. In particular, its performance is essentially fixed, since the peripheral cannot easily be enhanced in order to accommodate a more complicated scene. Of course, it would be possible to build several such peripherals and then somehow to multiplex between them, but this solution is rather expensive if the duplicate peripherals are built using the same technology as the original. Suppose, however, that the processors for intersecting rays and polygons are implemented as custom large scale integrated circuits. When considered separately, these processors are quite a bit less powerful than the ray tracing peripheral described earlier, but they are so inexpensive that many more can be used. The hope is that if enough of the small processors can be brought to bear on the problem, their combined performance will exceed that of a ray tracing peripheral made up of standard parts in the conventional manner. This section explores one possibility for using such a collection of small processors to make a ray tracing peripheral capable of relatively high performance.

Not too surprisingly, the shift in economies introduced by the availability of custom integrated circuits causes a corresponding shift in viable machine organizations. Recall that in the ray tracing peripheral of the previous section, all of the polygons in the scene shared the use of a single intersection processor. Now, with comparatively inexpensive processors, each polygon will have its own dedicated intersection processor. Before, the representation of a ray was fixed in the processor, and the polygons were streamed through the pipeline. Now, the individual processors are organized as a pipeline, and it is the rays that are streamed past the relatively fixed polygons.

2.4.1 Operation of the Pipeline

Each of the ray tracing processors that make up the pipeline performs the same basic computation as the complete ray tracing peripheral of the previous section. Every processor maintains the representation of a single polygon, and it passes descriptions of rays through its input and output ports. When it receives a ray description, the processor determines whether that ray intersects its stored polygon, and if so, it locates the intersection. It must compute not only the three-dimensional coordinates of the intersection point, but also the distance along the ray and the uv-coordinates of the point on the polygon. Thus, each processor must potentially compute six numbers.

The processors in the complete ray tracing pipeline are strung together as shown in Figure 2–23. Each processor has, at least conceptually, a single input and a single output, and the output of one processor is connected to the input of the next. The resulting string of processors has one unused input port and one unused output port, which are connected to the host computer to form its interface to the ray tracing pipeline.

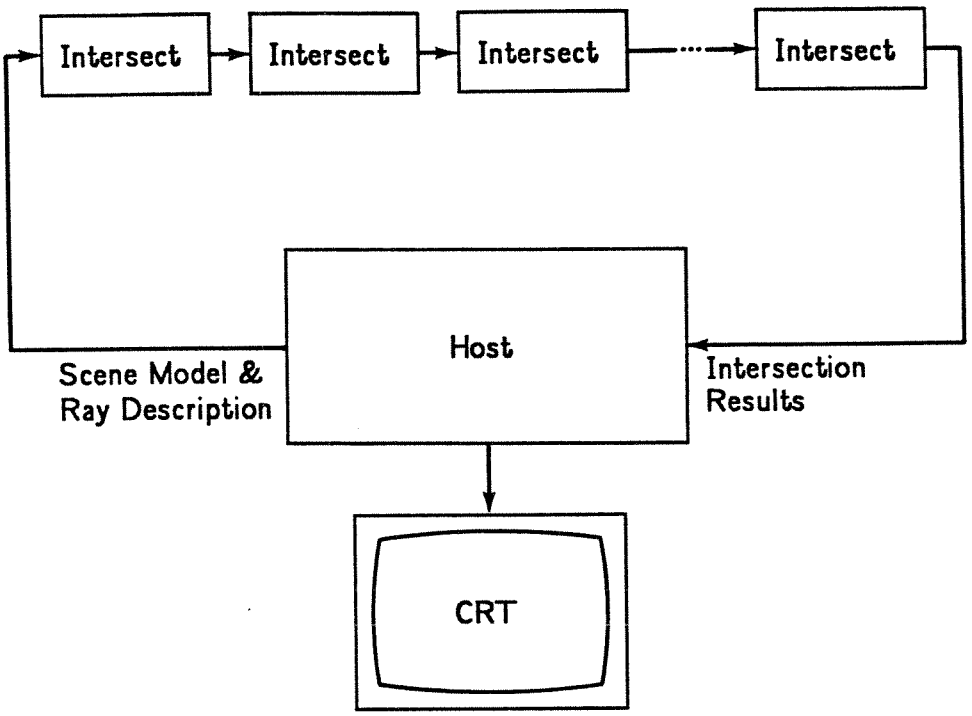


Figure 2–23. Processor organization in the ray tracing pipeline.

The inputs and outputs of polygon processors are capable of passing information about rays. The rays are described by their point of origin, some other point along the ray, and the six numbers mentioned earlier that describe the intersection of the ray with a polygon. Finally, one additional number, which will be discussed more fully in a moment, identifies the polygon giving rise to the point of intersection. Since each point is represented by three numbers, a complete ray description consists of the thirteen numbers summarized in Figure 2–24.

The host computer causes a ray to be traced by passing a ray description to the first processor in the pipeline. The two points contained in this description specify the ray, and except for the distance from the origin of the ray to the point of intersection, the values of other fields in the description are unimportant. The distance field should be initialized to infinity, which indicates that no intersection point has yet been found. When it receives the ray description, the polygon processor at the beginning of the pipeline, like any other processor in the pipe, proceeds to intersect the ray with its

$\mathbf{r}_0$	Origin of the ray.
$\mathbf{r}_1$	Point along the ray.
$\mathbf{p}_r$	Intersection of ray and polygon.
t	Distance along ray of intersection point.
(u, v)	Polygon coordinates of intersection point.
i	Identity of intersecting polygon.

Figure 2–24. Fields of a ray description.

own stored polygon. If the ray and polygon intersect, and if the distance from the origin of the ray to the point of intersection is smaller than the distance carried in the ray description, then the new intersection takes priority over any that may have been found previously. In this event, the processor replaces the intersection fields of the ray description with the results that it has just computed, and it also substitutes its own identity into the polygon identity field to show which polygon produced the intersection point. If the polygon and ray do not intersect, or if the intersection point is farther from the origin of the ray than another one found earlier, then the polygon processor does not disturb the ray description. After doing all of this, the processor passes the possibly updated ray description through its output port for intersection with the next polygon. When the ray description has passed through the entire pipeline, it will have been updated to reference the first polygon struck by the ray.

From the point of view of the host computer, the ray tracing pipeline described above is not very different from the ray tracing peripheral described in the previous section. In either case, the host sends rays to be traced and retrieves the results of those requests. The primary difference is that the pipeline can accept many ray tracing requests before it produces the first result. At any instant, a large number of these requests may be spread out along the pipeline in various stages of completion. Thus, in order to take full advantage of the pipeline's performance, it is imperative that the program running in the host computer be able to pursue many successive ray tracing computations at once, suspending those that must await results. Some techniques for doing so were outlined earlier. Finally, notice that the ray tracing pipeline has no provision for assisting with the lighting model computations. The host can either perform these computations on its own, or it can make use of a lighting model processor like the one described previously.

One aspect of the pipeline that has not yet been treated is the mechanism by which the polygon descriptions are loaded into the processors. These descriptions contain five vector and five scalar values, as they did in the ray tracing peripheral of the previous section. Here, however, descriptions carry an additional value that serves to identify the polygon. The host computer can load a polygon description into the ray tracing pipeline by presenting it to the first processor in the pipe. Upon detection of

such a polygon, this first processor passes its current polygon to the second processor before accepting the new polygon. Every processor in the pipeline behaves in a similar manner, and the net effect is to shift each polygon description by one position in the pipeline. The description formerly held in the final processor is shifted back to the host processor. Notice that if there are fewer polygons than processors, not all processors will be used, and any idle processors must be loaded with dummy polygons that are very far away from the scene model. The host computer must then ignore any intersection results that are marked with the identity of one of these dummy polygons.

2.4.2 Implementation of Arithmetic

Just as in the earlier ray tracing peripheral, using a floating point representation for numbers in the ray tracing pipeline is desirable because it makes the machine so much easier to use. Unfortunately, though, nothing equivalent to a TRW multiplier is available when designing custom integrated circuits. Since one of the properties of the intersection processor must be a relatively low cost, it would not be feasible to devote the substantial chip area that would be required to implement parallel multiplication circuitry. Smaller, shift-and-add multipliers can compute the same result with a much lower investment in silicon. Although they are slow when considered individually, their combined performance can approach that of a parallel multiplier. Also, it turns out that the other floating point operations can be implemented in ways that are comparable in area and speed to the shift-and-add multiplication. Each arithmetic operation can be estimated to take about five microseconds, as detailed in Appendix A.

2.4.3 Communications Requirements

Recall that the ray tracing pipeline operates by passing descriptions of rays from processor to processor. This communication must, of course, take place over the physical wires that connect the processors. One obvious way of transmitting the information is to use a separate wire for each bit of the ray description, but with thirteen words of 32-bits each, this solution is clearly infeasible. Another approach might be to pass a word at a time. Even this improvement is not sufficient, though, because it requires separate 32-bit data paths into and out of every processor. It would be possible to arrange for the two data paths to share the same set of wires, either by connecting all of the processors to a common bus or by using external multiplexors, but neither of these fixes is really satisfactory. The first is undesirable because it routes all messages through a central bottleneck, thereby hampering the performance of the entire pipe. The second fix merely moves the problem instead of eliminating it.

A workable solution to the communications problem is to transmit the ray description data bit-serially, using one wire for each incoming value and one for each out-

going value. Since rays are described by thirteen numbers, this technique requires only twenty-six wires, plus handshake signals. It has other pleasant properties as well. Notice that the cost of transporting a value from the pin of the chip to the site of its eventual use is small because only one wire is needed. When the value arrives at its destination, a shift register, which may be part of the arithmetic device, can be employed to deserialize it. Furthermore, since signal pins are dedicated to specific values, no multiplexor circuitry is needed for sorting out the incoming values or for formatting the outgoing ones. Finally, the use of serial transmission for communicating between chips suggests that it may also be convenient to perform on-chip communication serially, thereby limiting the internal wiring requirements.

The performance of the serial communication mechanism depends, of course, on the time it takes for an integrated circuit to transmit a bit. This time can range from perhaps 20 nanoseconds up to 200 nanoseconds or more, depending upon fabrication technologies, the distance over which the communication must take place, the amount of power the chip can dissipate, and other similar issues. Also, handshake protocols can boost the number of bits that must be transmitted. The fairly conservative estimate of about 150 nanoseconds per bit will serve as a basis for the following discussions. This figure means that transmitting a ray description takes about five microseconds, which is comparable to the floating point operation time.

The ratio of communication time to arithmetic computation time is important because it influences the way in which individual processors should be designed. For example, if the communication time were much greater than the computation time, the extra speed would be largely wasted because no processor can accept or produce results faster than it can compute them. This situation suggests that a communication scheme using fewer wires, or perhaps a different protocol, might be better suited to the available technology. On the other hand, suppose that the communication path was very sluggish compared to the time needed to produce a floating point result. In this case, there is no point in heavily pipelining individual processors, since they would be unable to obtain raw inputs or dispose of processed outputs through the slower communication paths. Instead, it would make sense to use fewer arithmetic devices and to share them between the various parts of the intersection computation. The smaller, streamlined processors would be cheaper, and although they would also be slower, this difference could not be detected through the communication paths. As integrated circuit fabrication technology advances, the speed of operations occurring within a single chip can be expected to increase more rapidly than the speed of transmitting values between chips, and this observation suggests that the ray tracing pipeline will probably become cheaper in the future, but it is unlikely to become very much faster.

With today's fabrication technology, it is probably not feasible to place an entire intersection processor on a single integrated circuit, and therefore the computation must be separated into pieces that do fit on one chip. The partitions must be chosen with care to insure that the communication between them does not become excessive.

There are several places at which the breaks might naturally occur. For example, the computation of t and $\mathbf{p}_r$ could be placed in a single chip, while u and v might each be computed separately. Presumably, the communication between portions of a single processor would occur at the same rate as communication between successive processors. Subcomputations can be pipelined as they were in the earlier ray tracing peripheral, so that the overall performance of the pipeline should not suffer.

2.4.4 Analysis

The performance of the ray tracing pipeline can be estimated rather easily, since it is essentially determined by the time it takes to pass ray descriptions from one processor to the next. Communication takes about 5μsec, so it follows that the overall throughput rate is 5μsec per ray. In other words, the pipeline can complete a ray tracing computation every 5μsec, so long as the pipe is kept full. In addition, because distinct processors are devoted to every polygon in the scene, the throughput rate is independent of the number of polygons. Of course, a ray description must pass through every one of these processors to get through the entire pipe, and the latency time is correspondingly large. If the machine consisted of a thousand processors, for example, the latency time would be 5 milliseconds (ms), or a two-hundredth of a second. In comparison, when operating on the same thousand polygon scene, the ray tracing peripheral of the previous section exhibits both a latency time and throughput rate of one ray every 333μsec. With the ray tracing pipeline, the host computer could potentially trace enough rays to make a 512×512 pixel image in roughly 1.3 seconds. Doubling this figure for shadowing, and doubling or tripling it again to allow for anti-aliasing, suggests that it would take about eight seconds to complete an image of a scene consisting of a thousand polygons.

The ray tracing pipeline does, however, have some substantial difficulties not mentioned in the foregoing analysis. First of all, the overall system is not very well balanced, because the pipeline is substantially faster than the host computer. The ray tracing pipeline can accept a request and produce a result every 5μsec. Thus, in order to fully utilize the pipeline, the host must be able not only to generate ray descriptions at this rate, but also to deal with the responses in the same 5μsec. It is unreasonable to demand such a level of performance from a medium-scale computer like the one postulated for the host. Even in simple situations where no complicated lighting effects are desired, the host would almost certainly be unable to keep up. The overall system would therefore proceed at essentially the rate that an unassisted host would run if the intersection computation took no time. Unfortunately, most of the processors in the pipeline would be idle at this speed. One way to resolve the problem might be to build a special purpose processor to replace the host. This approach is also rather unsatisfactory, because it sacrifices not only the flexibility inherent in the general purpose host, but also the simplicity and uniformity of a machine built from

a single, highly replicated component.

Another problem with the ray tracing pipeline is that most of the intersections being computed at any one time will turn out to have no bearing on the appearance of the final image. Thus, even among those processors that are working, very little useful computation is taking place. In the earlier ray tracing peripheral, the technique of superimposing a three-dimensional grid on the modeling area and then restricting computation to certain subvolumes produced a significant improvement in the efficiency of the machine. No similar kind of subdivision is possible with the pipeline, because it would require a separate pipe for each subvolume in the grid. Any sort of dynamic assignment scheme rapidly becomes unmanageable.

Yet another difficulty is that the ray tracing pipeline is not really so extensible as it might seem to be. In theory, it would be possible to expand its capabilities by adding more intersection processors. Although it might be possible to build a machine composed of ten-thousand of these processors, a machine with a million processors is almost certainly infeasible. Compare this with the ray tracing peripheral of the previous section, which could be expanded to handle a million polygon scene with the addition of ordinary memory.

It would seem, then, that the ray tracing pipeline has at least some of the problems often associated with so-called innovative large scale integrated circuit designs. While it can perform its function very rapidly, this speed is largely wasted in an environment that cannot make use of it. Furthermore, much of the pipeline's potential speed is squandered on unproductive computations. One sometimes hears that integrated circuit technology is advancing at such a rapid pace that computing power will be cheap enough to waste, but it seems more desirable to devise a different machine organization that is capable of more fully utilizing its resources.

2.5 A Ray Tracing Array

Each of the two ray tracing architectures that were presented earlier in this chapter has its own distinct advantages and disadvantages. The ray tracing peripheral made from standard commercial components was able to use its resources efficiently, but its performance could not easily be improved. The ray tracing pipeline, on the other hand, could be expanded almost indefinitely by adding new processors to the pipe. It achieved this extensibility at the expense of squandering its component processors, however, and no reasonable host processor could fully utilize its potential, resulting in still further waste.

The ray tracing array represents an attempt to design a system that combines the useful features of the two other machines while minimizing their defects. Unlike the others, the ray tracing array incorporates some general purpose computing capability in its structure. Its special purpose ray tracing hardware and general purpose computing hardware are packaged together, so that these two features will remain balanced

when the machine is expanded incrementally. Like the standard component peripheral, the organization of the ray tracing array helps to reduce the amount of unproductive computation in order to utilize the available resources more fully.

Another difference is the way in which the array exploits the concurrency inherent in a ray tracing algorithm. Recall that the original ray tracing peripheral made use of the parallelism that was present within the intersection computation itself. The ray tracing pipeline was able, in addition, to perform many intersection computations at once. Moreover, the communication in the pipeline occurred in a very uniform and regular manner. The ray tracing array is also capable of computing many intersections at the same time, but the communications between various parts of the machine appear almost chaotic, and the flow of information from one part of the machine to another follows no easily discernible pattern. This behavior exploits the fact that there is a great deal of flexibility in the ordering of ray tracing computations. In particular, unless one ray was generated directly from another, the two rays may be computed completely independently. Even if the two rays are associated with the same pixel, their respective contributions may be determined in either order because they will simply be added together in the final image.

Two basic notions are required for understanding the structure and operation of the ray tracing array. First, as in the case of the original ray tracing peripheral, a three-dimensional grid is superimposed on the modeling space to section off the volume into a collection of subvolumes. In the original ray tracing peripheral, the single intersection processor was multiplexed between these subvolumes. In the ray tracing array, on the other hand, every subvolume in the modeling space has, at least in concept, a dedicated processor. Each of these processors is responsible for maintaining the surface models in its own subvolume, as well as for computing the intersections of these surfaces with rays passing through the subvolume. The second major notion of the ray tracing array is that rays crossing boundaries between subvolumes can be modeled as messages passed between the corresponding processors. In a sense, the processor array can be thought of as the medium through which the ray messages travel. It is the job of the processors to handle the messages in a way that maintains the analogy between light and the modeled rays. Notice that the structure of the physical communication paths is compatible with this goal because light rays always travel through contiguous regions of space and never jump across a region without passing through it.

As just described, the ray tracing array would be moderately difficult to build with current packaging technology. The problem is that the machine consists of a three-dimensional lattice of processors, each of which must communicate with its six neighboring processors. For a machine of any substantial size, either the spatial arrangement of processors or the wiring between them becomes cumbersome. To avoid this situation, the ray tracing array is actually a two-dimensional grid of processors, as illustrated in Figure 2–25, and individual processors communicate with only four neighboring processors. The third dimension of the partitioning grid must be simulated

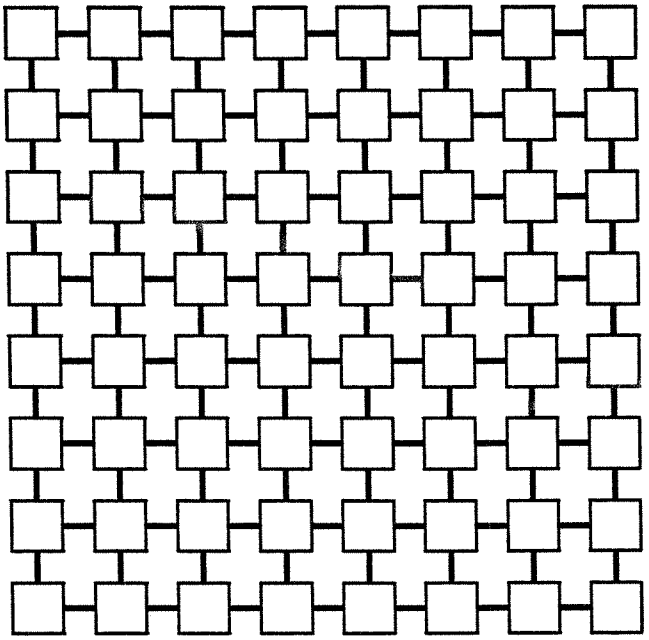


Figure 2–25. Organization of processors in a sixty-four processor ray tracing array. Links between boxes represent bidirectional communication paths between processors.

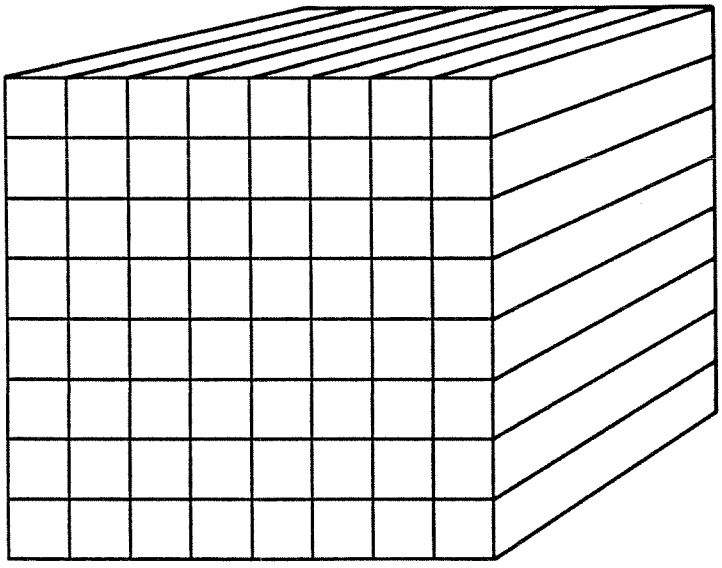


Figure 2–26. Organization of subvolumes in a sixty-four processor ray tracing array.

within each of the processors in the array. The correspondence between processors and subvolumes is shown in Figure 2–26.

The two-dimensional physical organization of the ray tracing array makes it convenient to distribute a frame buffer among the processors. The frame buffer is a region of memory set aside in each processor, and it is used to accumulate the image produced by the ray tracing algorithm. Each processor maintains storage for a block of pixels, where the location of the block in the final image corresponds to the processor's location in the array. For example, the processor in the upper left corner of the array holds pixels for the upper left portion of the image. Additional hardware allows pixel values to be shifted out of the frame buffer in scan line order.

As in the previous designs for ray tracing machines, the ray tracing array operates as one of a host computer's peripheral devices. The array is, however, a much more autonomous device than the previous ones. The host still maintains the scene model, but it no longer takes an active part in the ray tracing computation itself. After delivering the scene and supplying the viewing position parameters, the host is free to perform other tasks until the ray tracing array has completed the image. The function of the host in this setup has been reduced to that of an interface between the user, the graphics processor, and the host's disk drives and other peripherals. Thus, the host machine need not be especially powerful, and it might even be a non-dedicated machine like a timeshared computer.

2.5.1 Operation of the Array

Probably the best way to understand how the ray tracing array works is to examine each of the steps necessary for generating an image. After first initializing the array, the host computer must transfer the scene model to it by successively broadcasting polygon descriptions to every processor in the array. When a processor detects a polygon description being broadcast, it determines whether that polygon intersects the region of space for which it is responsible. The processor then either stores the polygon or discards it, depending on the result of this test. Once the surfaces in the scene model have been transferred, the host computer must broadcast the other attributes of the scene, including the positions and intensities of every light source illuminating it.

In the other ray tracing machines, polygon descriptions were limited to purely geometric information. Now, however, since the ray tracing array must be able to compute the entire picture without host intervention, information needed to apply a shading model must be included as well. In addition to the polygon properties, for example, if the scene requires any kind of mapping to be performed, the maps must also be supplied. Storage for these maps is distributed throughout the array just as the frame buffer image was. That is, every processor in the array stores the portion of the map corresponding to its own position in the array.

k	Message type (e.g., vision, shadow, reflection).
(r, c)	Row and column of the pixel for this ray.
$\mathbf{r}_0$	Origin of this ray.
$\mathbf{r}_d$	Direction of this ray.
$\mathbf{c}$	Color contribution of this ray.

Figure 2–27. Fields of a ray message.

When it has finished loading the scene model, the host computer initiates the image computation by supplying the viewing parameters. Processors within the array begin to generate rays in the form of ray messages which are passed from processor to processor as necessary for implementing the ray tracing algorithm. These ray messages may be thought of as records having the three vector and three scalar fields listed in Figure 2–27. The first field is a scalar that identifies the type of the ray message, and the value of this field determines how the message will be treated as it is passed from one processor to the next. The two other scalar fields hold the row and column of the pixel for which the ray is being traced. Two of the vector fields specify the geometry of the ray: the first is the origin of the ray, and the second is its direction. Finally, the value of the third vector is the maximum contribution that the ray can make to the color of the corresponding pixel. The exact use of this field will be described later.

Processors create initial ray messages for pixels that lie within their own portion of the frame buffer. For example, the processor in the upper left corner of the array will create a ray message to determine the color of the the upper left pixel in the image. The type field of this message will identify the ray as a new ray, the pixel fields will contain the coordinates of this upper left pixel, and the origin of the ray will be the eye position. The processor will compute the direction vector to make the ray pass through the upper left raster point on the viewing plane. Finally, the color field will be set to its maximum value, representing intense pure white. Notice that since only one new ray is generated for each pixel, the color contribution field must be maximal, to signify that the entire color of the pixel will be determined from this ray, and possibly from other rays that it might imply.

Once a processor has created a new ray, it should not immediately begin to search for intersections of the ray with its own polygons, because the ray might not even pass through its portion of the modeling space. Instead, it must somehow route the ray message to the processor responsible for simulating the subvolume that the ray would first strike after leaving the eye. The identity of the subvolume, and therefore of the processor, is easy to determine by examining the origin and direction of the ray. Therefore, if the ray starts in some foreign subvolume, the processor passes the ray message over one of its communication paths in the direction of the desired location.

The receiving processor performs a similar test when it receives a new ray, and it forwards the message once again if necessary. When the ray message finally arrives at its proper starting point, the type field labeling it as a new ray may be changed to the value representing a vision ray.

Processors in the array handle vision rays, just as the first ray tracing peripheral did, by attempting to find the ray/polygon intersection point that lies closest to the origin of the ray. The hardware implementing the intersection computation will be described later, but for now it is enough to realize that the processor either will find an intersection point within its subvolume, or it will not. If no intersection occurs, then nothing is visible in the processor's region of the modeling space, and the ray must be passed to another processor. Since neighboring processors are responsible for adjoining regions of space, the processor responsible for the next subvolume along the length of the ray is adjacent to the processor currently handling the ray. It is not difficult to identify the correct processor and to transmit the ray message through the appropriate communication path. Any attempt to transmit a ray message off the edge of the array, through a communication path that is not connected, results in the destruction of that ray. This situation will occur if the ray doesn't intersect any surface in the scene, and such a ray will not contribute to the color of any pixel. Thus, if a vision ray falls off the array's edge, the corresponding pixel will retain its initial color, black.

A processor's actions when it locates a valid intersection between a vision ray and a polygon can be rather complicated. First, it must compute the effect of ambient illumination on the apparent color of the intersection point. Recall that this component depends only on the color of the surface at the point of intersection and on the color of the ambient light source. Once computed, however, the resulting color must be tempered by the maximum color given in the ray message. That is, the components of the ambient illumination must be scaled by the corresponding components of the maximum illumination. After scaling, the vector representing ambient illumination can be placed into the color field of a ray message that has a type field identifying it as a result message. The other fields of this message are all copied directly from the original ray message. Processors in the array respond to result messages by passing them back to the processor that maintains the portion of the frame buffer containing the pixel mentioned in the message. When a result message eventually arrives at its destination, the color contribution carried in the message will be added to the color already present in the frame buffer.

If shadowing is being modeled, the processor has still more work to do after finding the intersection point and computing the ambient illumination. It must determine the contribution of each light source to the apparent color of the intersection point, assuming for the moment that the intersection is not shadowed. These computations require the colors of the light sources and of the surface, in addition to vectors specifying the position of the light sources and the orientation of the surface. Just as for ambient illumination, the contributions resulting from direct illumination must

be scaled by the maximum color that was given in the ray message. Because direct illumination from light sources contributes to the color of the final pixel only if the intersection point is not in shadow, it is necessary to trace more rays to discover any shadowing surfaces. Therefore, the processor must create a ray message for each light source, and it uses the type field to mark them as shadow rays. The pixel fields are copied directly from the original ray message, but the others are all different. Shadow rays originate at the intersection point, and they are directed toward the respective light sources. The color fields hold the direct illumination components that were just computed.

Ray messages marked as shadow rays are a little easier to handle than vision rays. As they do for vision rays, the processors must intersect shadow rays with surfaces in the scene. If a shadow ray strikes a surface, however, it may be discarded entirely because its contribution to the illumination of the pixel has been blocked by the surface. Alternatively, if the surface is transparent and non-refracting, the color of the shadow ray message can be attenuated before the message is processed further. If a shadow ray passes completely out of the scene model without striking a solid surface, then its origin is directly illuminated, and the light source does indeed contribute to the color of the pixel. The processor holding the ray message at the edge of the array can send the contribution to the proper pixel in the frame buffer by changing the type field of the message to mark it as a result.

Simulating reflection and refraction also requires tracing additional rays from the original intersection point. As before, the processor computes the maximum components of the reflected and refracted illumination on the basis of surface properties, and it scales these values by the maximum color from the original ray message. The resulting reflective and refractive illumination components supply color fields for two more ray messages. These rays originate at the surface intersection point, and they are aimed in the direction of reflection or refraction. The pixel fields are, of course, identical to those found in the original ray message. To be consistent, the type fields of these two new messages should probably identify them as reflection or refraction rays, but it turns out that since they will be handled in exactly the same way as vision rays, they might as well be marked as such. In particular, rays traced to model reflection and refraction might themselves result in shadow rays, or even in more reflection or refraction rays. The original pixel identity is propagated throughout, in order that all eventual contributions to the illumination of that pixel will be properly routed.

There is a potential difficulty that might be encountered when simulating unlimited reflection and refraction. It is possible for rays to become trapped so that, for example, an unbounded number of rays might be traced back and forth between two parallel mirrors. This problem may easily be avoided by requiring all reflective and refractive surfaces to attenuate rays that strike them. In addition, if the color of a newly formed ray message is below some intensity threshold, its contribution to the final pixel color would be negligible, and the message should not be processed.

The ray tracing array can perform texture mapping and similar effects by slightly

modifying the methods described above. Recall that the maps themselves are distributed throughout the array of processors, much as was done with the frame buffer. For example, the upper left portion of the map is stored in the upper left processor in the array, and so on. When a processor determines from its stored scene model and the results of an intersection computation that it needs a value from a map, it dispatches a message identifying the map and the uv-coordinate to be used as an index. Sometime later, the processor responsible for the desired section of the map will send a reply message giving the value out of the map. When the original processor receives this reply, it can proceed exactly as if the extra level of message passing had not been necessary. In particular, it might generate messages for results, shadow rays, reflection rays, or refraction rays. For simple texture mapping, the map value will influence the color fields of these generated rays. For bump mapping, it will have an effect on the illumination computations, as well as on the direction of any reflection or refraction rays that are sent out. If the processor is doing transparency mapping, the value from the map will determine the relative proportions of the maximum color that are placed in the result message and refraction message. Finally, in the case of refraction mapping, the map value will be used in determining the direction of the refraction ray.

Although it was described above as sequential in nature, the operation of the ray tracing array actually admits quite a bit of concurrency. This is due largely to the fact that all of the processors in the array can potentially be active at the same time. Notice that for the most part, processors do not need results computed by other processors in order to proceed. There are two exceptions to this: mapping requires another processor to look up a value, and ordinary ray messages passed from one processor to the next cannot, of course, be processed before they are received. In any case, a processor that finds itself idle need only dip into its stockpile of untraced ray messages to remedy the situation. The order in which rays are traced has no effect on the appearance of the final image, so that it doesn't really matter if the processing of a particular ray is interrupted to await, for example, the response to a map lookup request.

The overview of ray tracing array operation is now complete. The machine proceeds as described above until it runs out of rays to be traced. When this happens, the image is complete, and the picture may be shifted out of the frame buffer. If the next picture is to be a different view of the same scene model, the host computer may immediately send the new viewpoint in order to initiate computation. Otherwise, if the scene model requires modification, the host must provide the updates before transmitting the next viewpoint.

2.5.2 Processor Organization

The individual processors that constitute the ray tracing array may be thought of as miniaturized versions of the ray tracing peripheral and host computer combination described previously. That is, the processors consist of a general purpose computer in

tandem with some special purpose hardware to assist with the ray tracing and lighting model computations. Additional special purpose hardware is dedicated to the task of communicating with a processor's four neighboring processors.

The general purpose computing component in one of the ray tracing array processors may be implemented with a commercial microprocessor. The Motorola MC68000 is probably the most suitable of those that are available [MOTO82]. Its performance approaches that of a medium-sized computer, and its instruction set is rich enough to be usable. Present versions of the MC68000 use a sixteen-bit data bus, and since memory is currently packaged as 64K bit chips, this fact suggests that a natural memory complement is 128K bytes for each processor in the array.

The special purpose intersection component of the processor can be implemented as a set of custom integrated circuits. Its organization is somewhere between those of the ray tracing peripheral and the ray tracing pipeline of the previous sections. On the one hand, like the original ray tracing peripheral, a single intersection processor will be shared between the many polygons stored within one element of the ray tracing array. On the other hand, the processor will use the same shift-and-add style of arithmetic that was employed in the ray tracing pipeline.

The organization of the memory for containing the polygon parameters is also a compromise. Recall that in both the original ray tracing peripheral and the ray tracing pipeline, each of the polygon parameters was kept in a separately accessible memory. The ray tracing pipeline differs in that all of the polygon parameters will be stored in the main memory of the individual processors. The memory is thus shared between the host microprocessor and its intersection processor. Providing a separate memory for every polygon parameter would make the processors too complicated for replication throughout the array.

The choice to store all of the polygon parameters in a single memory affects the organization and performance of the intersection processor. The most important consideration is the fact that the overall throughput of the intersection processor is limited by the rate at which it can fetch polygon parameters from memory. Moreover, since the memory is shared with the microprocessor, the intersection processor cannot monopolize even the bandwidth that is available. These observations suggest that it would benefit the performance of the array element if the intersection processor could be designed to require fewer memory fetches. Fortunately, this is indeed the case. Recall that the purpose of the intersection computation is to find three values. The distance along the ray of the point where the ray and polygon intersect is t, and the position on the polygon of the point of intersection is given by the coordinates u and v. If any one of these three values does not exist or is out of range, the other two values are meaningless and need not be computed; therefore, the polygon parameters required for these unnecessary computations need not be fetched from memory.

One way to satisfy the memory referencing constraints on the intersection processor is to design it as a pipeline, as shown in Figure 2-28. The first stage of the pipe determines the memory address of the polygon parameters, possibly following some

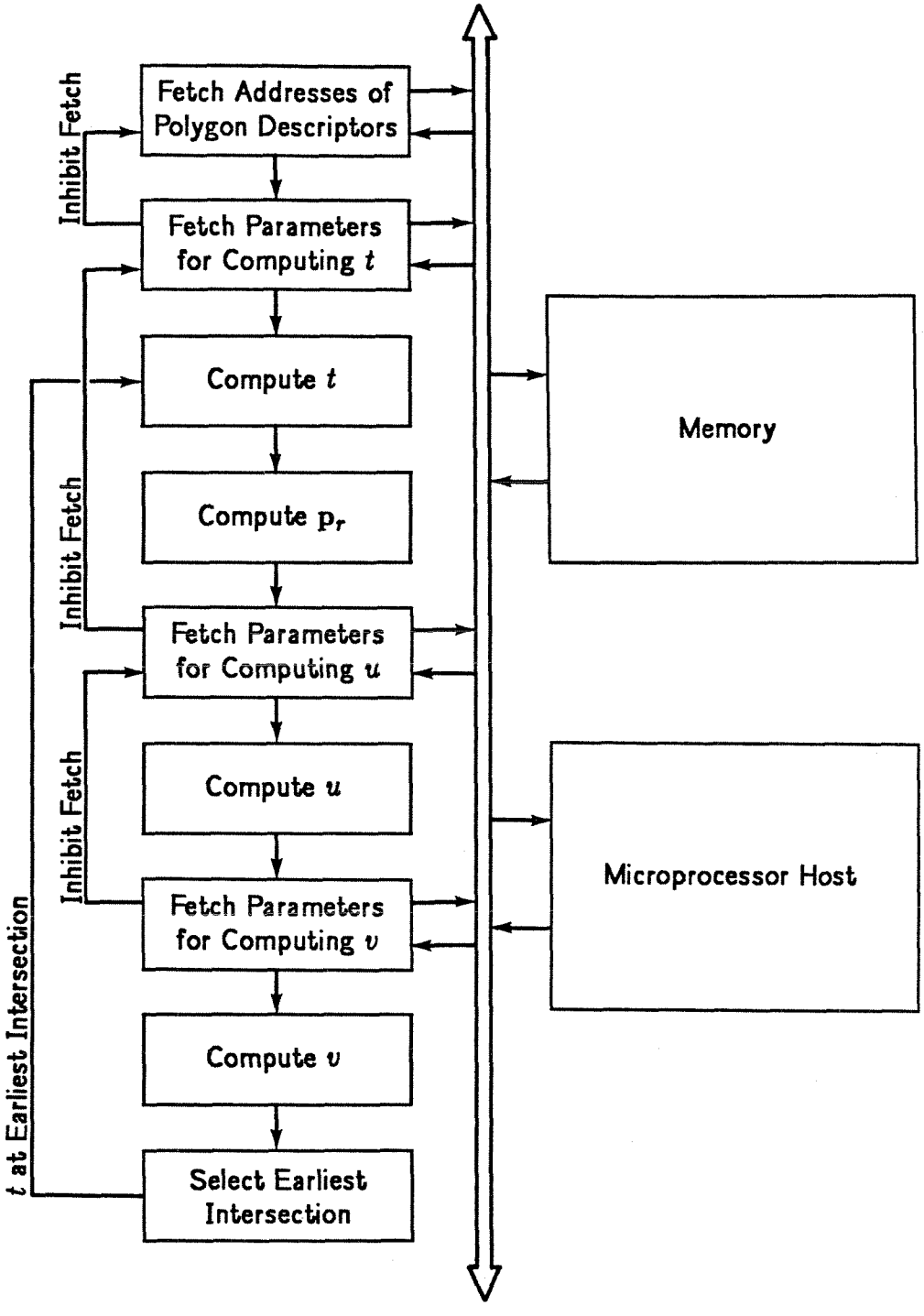


Figure 2–28. Pipelined organization of a processor in the ray tracing array.

pointers in order to accomplish this task. The next pipeline stage fetches the polygon parameters needed to compute the value of t, and the third stage actually computes this value. If t does not exist, or if it is negative, this third stage has no output; otherwise, the value of t is passed to the fourth stage, which computes the actual intersection point $\mathbf{p}_r$. The fifth and sixth stages fetch the polygon parameters associated with u and compute its value. Again, if the value of u is not within the required range from zero to one, the fifth stage has no output. The next two stages behave in a similar manner to compute the value of v. Finally, the ninth stage of the pipeline accumulates the result by maintaining the intersection point that is closest to the origin of the ray. Whenever it updates this intersection point, it also passes the new value of t back to the third pipeline stage. The third stage, which computes t, can use this value to tighten the bounds on the meaningful range of t, thereby further reducing unnecessary computations in later stages of the pipeline. One last note is that the four memory referencing pipeline stages are arranged so that the stages occurring later in the pipeline have priority over those appearing earlier. The object of this ordering is to prevent the log jams that would occur if the later stages could not obtain the required values from memory.

From the standpoint of the microprocessor, the organization of the intersection processor is irrelevant. When it has a ray to be traced, it simply passes the ray origin and direction vector to the intersection processor along with the address of the polygon data structure in memory. After the intersection computation has been completed, the intersection processor will interrupt the host microprocessor, which can then retrieve the results and proceed with the lighting model computation. In order to do this, it may call upon a special purpose lighting processor, as in the case of the original ray tracing peripheral. Now, however, the lighting processor will be implemented as a set of custom integrated circuits so that it may be more economically replicated for each element of the ray tracing array.

In addition to the computing hardware, each array element contains facilities for communicating with its four neighboring processors. These links operate by copying a ray message from the memory of one array element into the memory of an adjoining one. Every array element has four input ports and four output ports, corresponding to the four directions of message travel. All routing is done by software in the microprocessor, so that the port hardware need only deliver the messages. Both input ports and output ports accept a memory address that is interpreted either as the location of a message to be sent or as the location of a buffer that has been set aside to receive a message. After a transfer has taken place, the ports on either side of the communication path signal completion, either by interrupting their respective microprocessor hosts or by setting some flag bit.

One problem that has not yet been mentioned is the memory requirement of the software running in the microprocessor. Because the program is stored redundantly within every processor in the array, there may be a substantial waste of memory. This property might be viewed as the price to be paid for providing an independent path

to memory in each processor and thereby increasing the total memory bandwidth. One way to ease the storage requirements somewhat would be to swap the code used in the various stages of the ray tracing algorithm. The first stage occurs when the polygon model is loaded into the array. Each processor is expected to test the polygons to determine whether they intersect the volume assigned to it. Once this step is complete, the program is no longer required. The next step sorts the polygons in a way that simulates the third dimension of the grid subdivision algorithm. Again, the code for performing this task need not be resident in the microprocessor for any prior or subsequent stage of the algorithm. The same is true for the final phase in which the actual image is generated. Thus, the program in the microprocessor can be split into three smaller parts able to share the same memory.

2.5.3 Analysis

The method by which subvolumes were assigned to processors is straightforward, but there are some problems with it. Recall that each processor was responsible for a cross section of the modeling space, as illustrated in Figure 2–26. The problem with this organization is that for some choices of viewing position, not all processors will be equally busy. For example, if the scene is viewed from the front, vision rays will be distributed uniformly to processors throughout the array, as shown in Figure 2–29. On the other hand, if the eye is off to the side, those processors along the corresponding edge of the array will be forced to handle all of the vision rays, while the other processors remain idle. This situation is illustrated in Figure 2–30.

If the ray tracing array used a three-dimensional array of processors rather than a two-dimensional one, the assignment of subvolumes to processors would not cause this particular problem. Unfortunately, even aside from the interconnection issues, a three-dimensional array would be a very inefficient organization. The number of processors in the array increases as the cube of the number of subdivisions, but, as previous discussions have revealed, the number of non-empty subvolumes grows only quadratically. Thus, most of the subvolumes in a large three-dimensional array will contain no surfaces, and the corresponding processors would be wasted.

A better solution can be obtained by modifying the correspondence between processors and subvolumes. The range of possible modifications is limited, however, because subvolumes that adjoin one another in the modeling space must always be handled by processors that are connected to each other. This restriction is necessary because rays that travel continuously through the scene are modeled as messages passed continuously through the array. As illustrated in Figure 2–31, one way to satisfy the restriction starts out by assigning the subvolumes at the front of the modeling space to the respective processors in the array. Assignment of the remaining layers is similar, except that the correspondence between processors and subvolumes is shifted by one at each successive layer. In order for this reorganization to work,

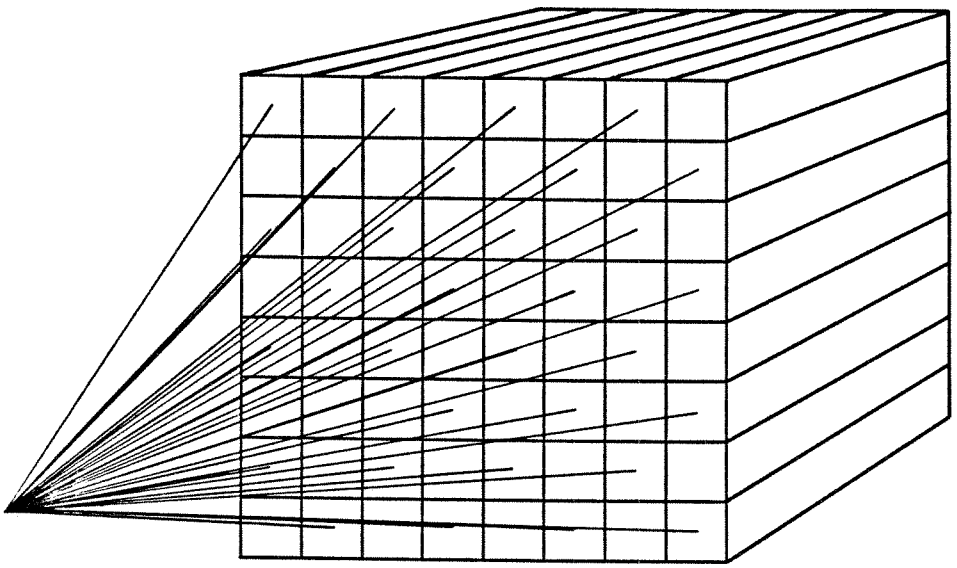


Figure 2–29. Distribution of rays to processors when the eye is in front of the modeling space.

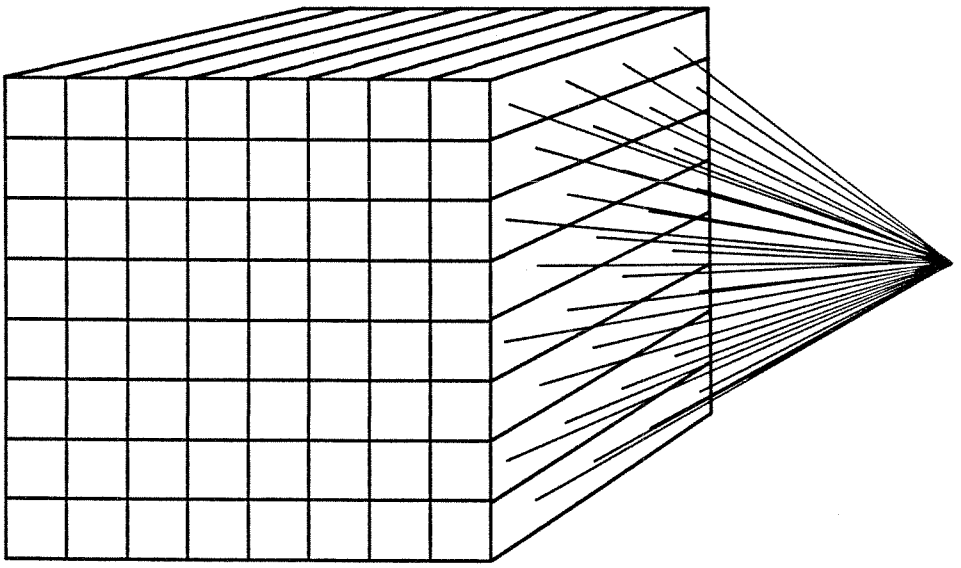


Figure 2–30. Uneven distribution of rays to processors when rays are to the right of the modeling space.

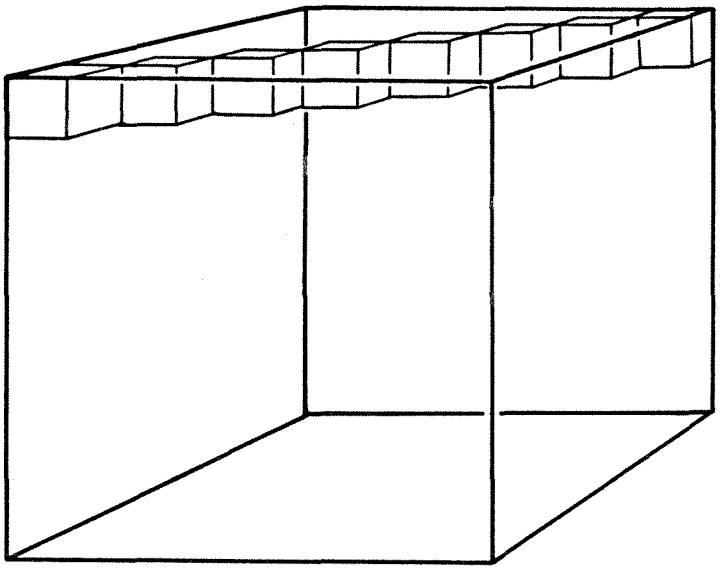


Figure 2–31. Processor assignment skewed horizontally in depth.

processors along the left edge of the array must be connected to processors along the right edge by means of the communication paths that are not otherwise used. If the viewing position is off to the right of the modeling space, processors in the array will be uniformly flooded with vision rays, but if the eye is above the modeling space, all of the vision rays will have to pass through the top layer of processors.

To be able to handle an arbitrary viewing position, the assignment of subvolumes to processors will have to be shifted down, as well as to the left, for successive layers of subvolumes. Such an organization is shown in Figure 2–32. The processors along the top edge of the array must be connected to those along the bottom edge in order to satisfy the continuity restriction. With these additional connections, the processors in the array form a torus. Note that although the subvolume assignment can shift one position either horizontally or vertically at each layer, it cannot shift both horizontally and vertically in the same layer because such an assignment would place adjacent subvolumes in unconnected processors. If there are n_s subdivisions and n_s^2 processors in the array, it takes n_s horizontal shifts and n_s vertical shifts to insure that processors are evenly represented on each of the six faces of the modeling space.

A modification of the skewed processor assignment may be used if there are not enough processors to sufficiently subdivide the modeling space. For example, suppose that the most effective way to subdivide a scene required sixteen partitions along each coordinate axis, but that the available physical array consisted of only sixty-four processors, or eight along each axis. Such a case may be handled by repeating the

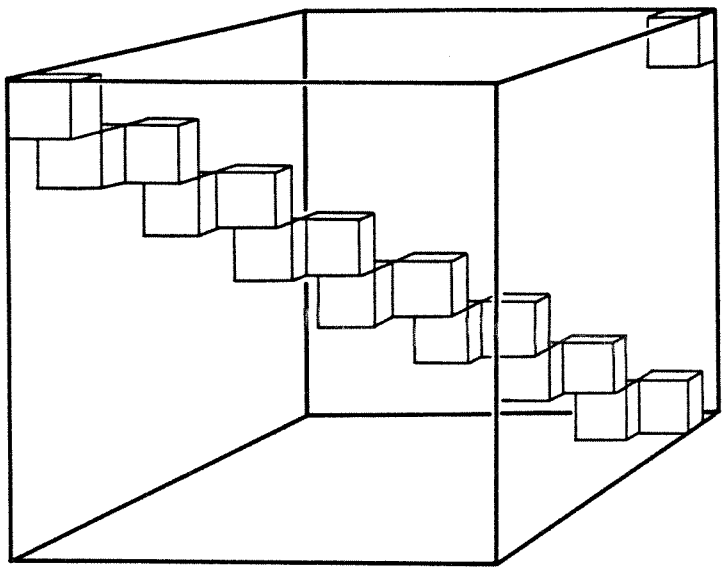


Figure 2–32. Processor assignment skewed both horizontally and vertically in depth.

pattern of skewing so that each processor will be assigned to more than one subvolume along the depth of the modeling space.

Assuming that the pattern of subvolume to processor assignment and the sheer magnitude of the computation are sufficient to insure that every processor in the array is kept busy, it is a simple matter to estimate the performance of the machine. The key issue is the speed with which the array can access the scene model stored in memory. Every processor in the array has its own dedicated memory, accessible through a 16-bit wide data path, in which all of the polygon parameters reside. Recall that the original ray tracing peripheral, on the other hand, stored each of the polygon parameters in a separate memory. Since there are twenty such parameters, each 32-bits wide, the original peripheral was capable if fetching 640 bits at once. To equal this performance, a ray tracing array would need forty processors. An array with a hundred processors would be 250% as fast as the original ray tracing peripheral.

This simple analysis ignores the fact that there are other demands placed on the component processors beyond mere parameter fetching, but it also discounts the savings that are accrued because not all of the parameters need to be fetched. Performance could be increased still further by providing a wider path to memory. The 16-bit data path in the array as described was a limit imposed by currently available microprocessor technology, but nothing in the nature of the problem prevents this from being increased by more than an order of magnitude. Also, since the component processors consist mostly of memory, one might expect the cost of the processors to

drop dramatically with the rapid advancement of memory technology. Thus, larger array sizes, in addition to faster individual processors, should become economical. If built today, a ray tracing array might be just a few times faster than the original ray tracing peripheral. With improvements in the processors made possible by progress in integrated circuit fabrication technology, however, the disparity should widen.

2.6 Extensions

There are doubtless many ways to enhance the capabilities of the ray tracing machines presented in this chapter. One possible improvement would permit the machines to handle more general surfaces. Curved surfaces and non-planar polygons are useful in a variety of applications for which planar, convex quadrilaterals are too restrictive. Another potential area for improvement concerns the overall capacity of the ray tracing machines. In each of the architectures presented, internal memory capacities limited the number of surfaces that could appear in a scene. This situation is not satisfactory because it is the very complex scenes that most urgently require the performance of a ray tracing machine. Thus, the ray tracing machines need a few modifications to help make them more suitable for practical applications.

The capacity problem occurs in conventional computers as well as in the special purpose ray tracing machines: programs or data may be too large for the main memory of the computer. If this happens, the usual technique, called virtual memory, is to store most of the information on disk, while keeping in main memory only the portion currently being used [DENN70]. When this technique works well, the program rarely references data on disk, and it therefore runs almost at the speed of main memory. Programs with this behavior are said to exhibit locality of reference because their attention is largely confined to a small region of their environment. On the other hand, programs that do not exhibit locality tend to reference widely spaced portions of their environment, and they run at the substantially slower speed of the disk. It turns out that ray tracing machines can use similar techniques in order to boost the complexity of the scenes that they can handle.

Ray tracing is well suited to disk swapping techniques for two reasons. First, since rays may be processed in any order, the processing of any particular ray may be interrupted and suspended indefinitely. This kind of suspension may be necessary if, for example, the ray requires a portion of the scene model that is currently stored on disk. Second, rays travel smoothly and continuously through space rather than jumping around at random. Their references to the scene model are thus fairly localized. Furthermore, successive rays do not differ by very much, resulting in even greater locality. It is interesting to note that these properties are exactly the ones that made ray tracing so susceptible to a highly parallel implementation.

One way of using a disk to hold part of the scene model works in much the same way that virtual memory does on a conventional computer. Recall that the grid

subdivision technique separates the modeling space into a collection of subvolumes. The contents of a single one of these subvolumes makes a convenient unit for swapping to the disk. Since a ray passes through only a few subvolumes, only a small portion of the scene need be kept in the ray tracing machine, and the remainder can be stored on disk. Furthermore, successive vision rays pass through pretty much the same sequence of subvolumes, so that the set of subvolumes kept in the processor should not require frequent changes. Shadow and reflection rays may at first seem to complicate this behavior, but notice that successive shadow and reflection rays are similar to each other because they are generated from similar intersection points. Thus, although the processor must be able to store one set of subvolumes for vision rays, one for shadow rays, and one for reflection rays, each of these sets will be fairly stable.

Virtual memory systems on conventional computers use a technique called demand paging for determining when a section of data should be transferred from disk to main memory. The basic idea is that the program will proceed normally until it attempts to reference a piece of its environment that is currently being stored on disk. When this happens, the program will be suspended until the offending information has been retrieved, whereupon the program will be resumed as if nothing had happened. The ray tracing machines can use a similar technique. If a ray attempts to enter a subvolume whose contents are not currently in the scene memory, the ray tracing process can be suspended while the required subvolume is fetched from disk. It may even be that the waiting time can be put to good use by tracing some other ray.

A demand paging scheme like the one just outlined would certainly work in the context of ray tracing machines, but it seems as though there might be a better way. Notice that a demand paging algorithm waits until a problem arises before taking any action. In a conventional computer, this approach is sensible because the algorithm has no way of predicting the future memory requirements of a program. In the ray tracing machines, on the other hand, vision rays are usually traced in a fixed order, and it should therefore be possible to identify in advance the subvolumes that will be needed. The swapping algorithm could maintain a supply of likely candidates in an attempt to reduce the number of accesses to unavailable subvolumes. Of course, the regular demand paging algorithm would have to be available for those cases in which the fancier methods failed.

Subvolume swapping should work well for the original ray tracing peripheral because the machine processes rays in a very orderly and regular manner. In contrast, the operation of the ray tracing array is much more chaotic. At any particular instant, the machine might contain many unrelated rays in various stages of completion, and there is little chance that these rays will be confined to some small collection of subvolumes. Therefore, if the machine swapped subvolumes, it would probably spend much of its time waiting for the disk. Fortunately, there is another approach that is better suited to the properties of the ray tracing array.

In a way, the alternate swapping scheme is a dual of the technique described above: when a ray enters a subvolume whose contents are currently on disk, instead

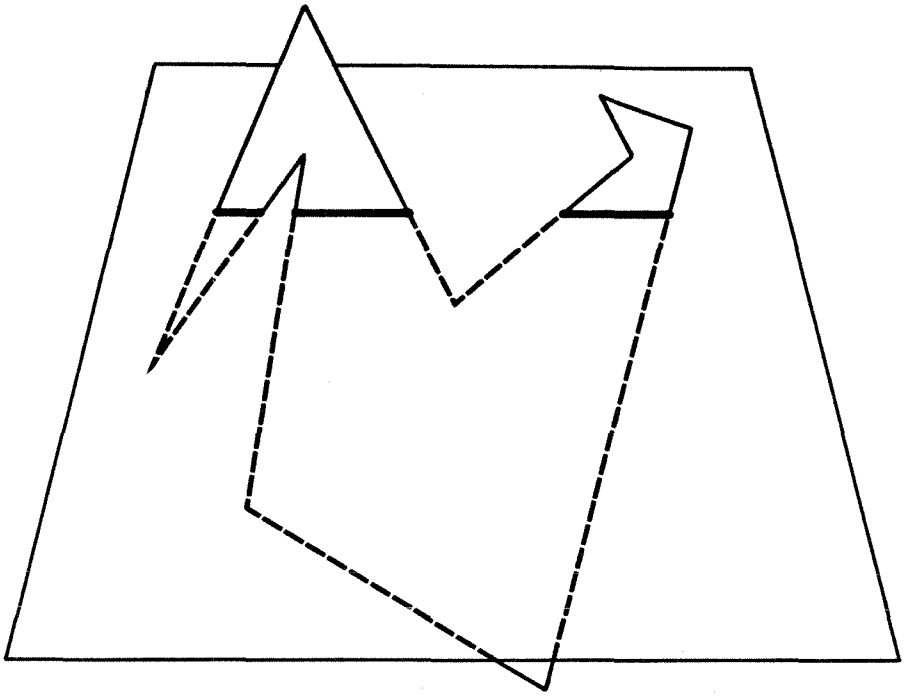
of fetching the subvolume from disk, the new approach stores the ray on the disk. To see how this scheme would work, imagine that the modeling space is divided into a few large subvolumes. These subvolumes are as large as possible, but small enough that the machine can hold every surface in the enclosed portion of the scene model. In effect, the machine will be considering a portion of the modeling space as if it held the complete scene. Without swapping, when a ray message reached the edge of the processor array, it could be discarded. Since the machine now contains only a part of the entire scene, the message that was formerly thrown away must be captured for storage on disk. When there are no active messages for the current portion of the modeling space, the polygons in the next portion can be loaded into the array, and the suspended ray messages can be reintroduced.

The ray swapping scheme makes use of the fact that the sequential transfer rate of a disk drive is reasonably high. Thus, there should be no problem in rapidly streaming ray messages onto and off of the disk. If the contents of a subvolume is stored in contiguous disk locations, the time required to load that subvolume may also be reduced. Notice that it may be necessary to reload the scene model a few times in the process of rendering a frame, because rays can be reflected back and forth across subvolume boundaries. By choosing overlapping subvolumes, it may be possible to minimize this difficulty, however.

The ray tracing machines described in this chapter have all operated on scene models consisting of nothing but convex, planar quadrilaterals. This restriction served to simplify discussions, but a practical ray tracing machine would probably have to deal with other kinds of surfaces. Curved surfaces would be nice, and more complex polygons would be essential. Except for the details of the intersection computation, the ray tracing machines are equally well suited to handling these other types of surfaces. None of the subdivision techniques or message passing protocols rely on the fact that only simple surfaces were being modeled. Indeed, the only change necessary to handle more complicated surfaces would be to replace the intersection processor.

Probably the most useful upgrade would be to allow the machines to handle more generalized polygons. These surfaces would not be restricted to four vertices, and they would not have to be convex, or even planar, for that matter. The generalized polygons would be specified by a sequence of vertex points with their corresponding vertex normal vectors. Texture mapping and similar operations could be performed if a uv-coordinate pair were also included with each vertex. To be in a convenient form for the intersection computation, the ray would be represented as the intersection of two planes. The intersection computation itself proceeds in two stages. First, as shown in Figure 2–33*a*, using one of the ray planes, the intersection processor must determine which of the polygon edges cross the plane, and it must determine where the crossings occur. Note that these crossings are the endpoints of line segments that lie both in the polygon and in the plane. Second, the processor must attempt to intersect these line segments with the other ray plane, as illustrated in Figure 2–33*b*. If such an intersection occurs, it is the point common to the polygon and the two ray

Step 1:



Step 2:

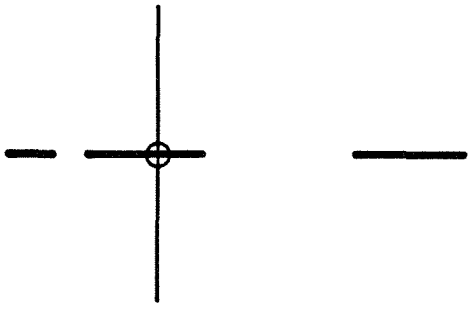


Figure 2–33. Intersecting a ray with a more general polygon.

planes. Thus, it is the ray/polygon intersection that was being sought.

The algorithm for intersecting rays with general polygons decomposes conveniently into a pipeline. The first stage of the pipeline accepts polygon vertices and calculates the distance from the vertex to the first ray plane. This computation requires nothing more than an inner product, which may be implemented as described earlier. The second stage of the pipeline examines these distances to detect edges that cross the ray plane. When it finds such a crossing, it computes the point of intersection, using a division and a few multiplications. Note that it must compute the interpolated normal vectors and uv-coordinates as well as the three-dimensional coordinates of the intersection point. The third pipeline stage calculates the distance between these intermediate intersections and the second ray plane. Again, this computation takes the form of an inner product. The fourth pipeline stage counts the number of intermediate intersection points on either side of the second ray plane, and it accumulates the two that are closest to the plane. If, after all of the vertices have been processed, there were an odd number of intersections on each side, then one of the segments must intersect the second plane. The fifth pipeline stage accepts the two closest intermediate intersections and computes the final intersection. It may also determine the distance from the origin of the ray to the intersection point by taking an inner product.

For some applications, it is preferable to use a genuinely curved surface rather than an approximation pieced together out of polygons. Bicubic surface patches are a very flexible surface modeling tool, but unfortunately, the computation required to produce an image of one of these surfaces is rather substantial and not very regular, making a hardware implementation more difficult. Kajiya has proposed a pipelinable technique for using a ray tracing algorithm to render these patches, but it can require as many as about 6000 floating point operations for every ray/patch intersection [KAJI82]. Given the current state of technology, it is not reasonable to postulate the massive quantity of hardware required to mount a frontal assault on the problem, and the issue of rapidly displaying patches remains open.

A simpler kind of curved surface that is often useful is called a quadric surface. It is specified as the solution of

$$\begin{aligned} 0 = Ax^2 + Bxy + Cxz + Dx \\ + Ey^2 + Fyz + Gy \\ + Hz^2 + Iz \\ + J. \end{aligned}$$

Suppose that rays are represented parametrically:

$$\begin{aligned} x(t) &= (x_1 - x_0)t + x_0, \\ y(t) &= (y_1 - y_0)t + y_0, \\ z(t) &= (z_1 - z_0)t + z_0. \end{aligned}$$

Substituting the parameterized point along the ray into the equation of the quadric surface yields an expression that is quadratic in t and which may be solved by

straightforward application of the quadratic formula. This computation may easily be pipelined. Notice that it is possible to combine quadric surfaces by keeping track of all intersections with a particular ray. For example, one surface may be denoted as enclosing negative volume, so that it would create a hole if it ever intersected a surface enclosing a positive volume. This extra functionality would, of course, require some additional processing at the end of the pipe, but it fits cleanly into the overall scheme of ray tracing.

3

Real Time Machines

For many applications of computer graphics, the quality of the final image is less important than the speed at which successive images can be produced. In flight simulation, video games, and other types of visual environment simulation, new images must be generated at the rate of about thirty times a second in order to maintain the desired illusion of reality. As the viewer becomes absorbed in the simulation, the fact that the synthetic world is not perfect in every detail may go unnoticed. Thus, although the ray tracing machines of the previous chapter are capable of generating very realistic images, they are unsuitable for many applications because they cannot operate in real time. This chapter examines another class of machines designed for high performance at the expense of some image quality.

Some time ago, researchers began to realize that the problem of hidden surface elimination could be eased somewhat by taking into account the characteristics of the display on which the final image would be viewed. In particular, an electron beam scans the face of a television monitor in a fixed order and with limited resolution, suggesting that the scene model should be transformed into the screen coordinate system, or image space, before the visible surface algorithm proceeds. The computations in this discretely sampled coordinate system are simpler than the ones that would be required in the continuous world coordinate system in which the scenes are modeled.

Some of the commercially available systems for visual simulation simplify the computations by performing them in image space. An early example is a machine based on Watkins' algorithm, which isolates visible surfaces by examining the scene one

scan line at a time [WATK70]. Only those surfaces that may appear on the current scan line need be considered. Also, the computation to find the visible surfaces proceeds incrementally from left to right across the scan line. The hardware implementation of Watkins' algorithm consists of a few very fast and fairly complex pipeline stages, each of which makes only limited use of concurrency. Instead, the machine relies on the speed of its individual components to achieve its high performance. The current generation of flight simulator, the Evans & Sutherland CT–5, uses a different visible surface algorithm, but the basic hardware configuration is similar [ROBI81]. It, too, is organized as a pipeline consisting of a few massive stages. The success of these machines illustrates the suitability of this organization to the technology in which they are implemented.

As the technology for fabricating large scale integrated circuits advances, a revised set of economies is emerging. Although the speed of individual components may not be very great, the cost of each component is dropping dramatically, so that it is becoming feasible to use many more components in a particular system. Furthermore, integrated circuit technology may be likened to printing in that the cost of making the second copy of something is less than the cost of making the first. These considerations suggest that machines composed of a myriad of identical devices operating in parallel would be suitable for implementation with custom integrated circuit technology. This chapter examines some machines for real time visible surface detection and display that have these properties.

3.1 Model Preparation

All image space hidden surface elimination algorithms must, of course, begin by converting the scene model from the object space, or world coordinate system, to the image space, or screen coordinate system. The exact procedure for doing so is described in a variety of textbooks on computer graphics [FOLE82, NEUM79]. The required steps will be outlined here in order to introduce the discussions that follow. Readers familiar with these notions may wish to skip this section.

First of all, it is assumed that the scene model is composed of planar, convex quadrilaterals embedded in three dimensions. The polygons are specified by the points at their four vertices, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$. Normal vectors, denoted by $\mathbf{n}_1$, $\mathbf{n}_2$, $\mathbf{n}_3$, and $\mathbf{n}_4$, are associated with each of these vertices. As in the previous chapter, the normal vectors will be used only for the shading computation and do not necessarily bear any relation to a vector perpendicular to the plane of the polygon. Finally, each polygon is associated with a color vector $\mathbf{c}$, also for use in the shading computation.

Polygons are represented in homogeneous coordinates. That is, a vertex (x, y, z) is represented as a row vector $[wx\ \ wy\ \ wz\ \ w]$, where w is an arbitrary non-zero scale factor. Conversely, $[x\ \ y\ \ z\ \ w]$ is a homogeneous representation of the point $(x/w, y/w, z/w)$. It is usually convenient to choose a value of 1 for w; hence the

homogeneous representation of (x, y, z) would be $[x\ y\ z\ 1]$. Normal vectors are treated a little differently because they do not undergo translation. A vector (x, y, z) has the homogeneous representation $[x\ y\ z\ 0]$. The apparent complication of using homogeneous coordinates is justified because they allow all of the necessary geometric transformations to be represented as 4×4 matrices. Of course, the polygon color $\mathbf{c}$ is never transformed, and it is therefore exempt from this treatment.

The first transformation that must be applied to the polygons of the scene model is the viewing transformation, which maps points in the world coordinate system into points in the eye coordinate system. The world coordinate system is right-handed, and its origin and scaling are chosen in a way that is convenient for the modeling task. In contrast, the eye coordinate system is somewhat standardized. It is a left-handed system where the viewing screen is parallel to the xy-plane and centered about the z-axis at $z = 1$. The eye is at the origin, directed toward the positive end of the z-axis. Visible points are enclosed within a pyramid whose apex is at the eye point and whose faces pass through the four edges of the screen. Two additional planes parallel to the xy-plane are usually chosen to bound the values of z that are visible, thereby forming a truncated pyramid.

The transformation from the world to the eye coordinate system may be accomplished by multiplying the homogeneous representations of the polygon vertices by the appropriate transformation matrix:

$$\mathbf{v}_e = \mathbf{v}_w V.$$

In this expression, $\mathbf{v}_e$ and $\mathbf{v}_w$ are vertices in the eye and world coordinate systems, and V is the viewing transformation. Generally, the viewing transformation is equivalent to a sequence of translation, rotation, and scaling operations. The polygon normal vectors do not need to be transformed so long as the locations of the light sources are also expressed in world coordinates.

The next step is to apply a perspective transformation that maps polygons from the eye coordinate system into the screen coordinate system. This transformation moves the eye position along the z-axis to negative infinity and adjusts the scene so that rays from the eye to points in the scene pierce the screen exactly where they did before the transformation. Thus, in the screen coordinate system, rays from the eye to the scene are parallel to the z-axis. This means that two points aligned on the same vision ray will have the same x- and y-coordinates, corresponding to the location on the screen to which both points project. The visible point in such a case will be the one with the smaller z-coordinate.

Application of the perspective transformation is also accomplished with a matrix multiplication:

$$\mathbf{v}_s = \mathbf{v}_e P,$$

where P is the perspective transformation, $\mathbf{v}_e$ is a vertex in the eye coordinate system, and $\mathbf{v}_s$ is its screen coordinate representation. The perspective transformation is chosen so that the truncated pyramid enclosing the visible region of space in

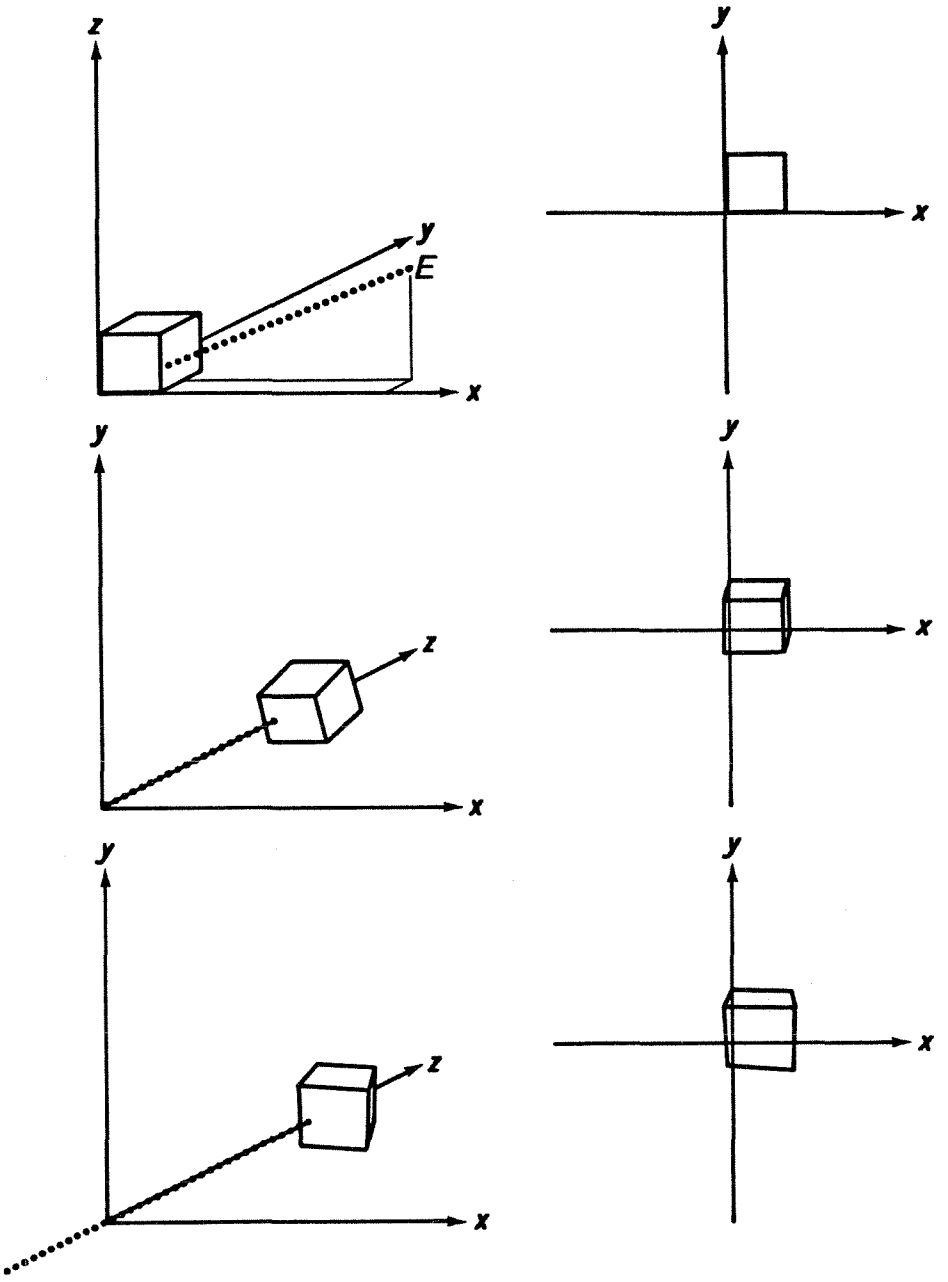


Figure 3–1. Steps in the transformation from World to Screen coordinates. Top: The scene model is given in the right-handed World coordinate system with the eye at point E, directed as shown. Center: After the viewing transformation, the scene model is represented in the left-handed Eye coordinate system with the eye directed from the origin along the z-axis. Bottom: The perspective transformation and perspective division move the position of the eye along the z-axis to negative infinity to achieve the transformation to Screen coordinates. Orthographic projections onto the xy-axis are shown to the right.

the eye coordinate system maps onto a standard region in the screen coordinate system. Visible values of x and y in the screen coordinate system lie in the range from -1 to $+1$, and visible values of z are between 0 and 1. Furthermore, the perspective transformation is chosen to map straight lines in the eye coordinate system to straight lines in screen coordinates. The perspective transformation does not, however, preserve the value of w, the scaling factor in the homogeneous representation of points. This means that in order to derive the three-dimensional coordinates in screen space, the scale factor must be divided out of the homogeneous representation. This operation is known as perspective division.

After the perspective transformation has been applied to the polygons, they must be clipped to eliminate those portions that lie outside the visible volume and, therefore, should not appear on the screen. Clipping must be performed before the perspective division because the divide operation destroys a sign bit that distinguishes points in front of the eye from those behind it. Recall that the bounds for x_s in the screen coordinate system are $-1 \leq x_s \leq +1$. In the homogeneous representation, before the perspective division has been done, these bounds are $-w \leq x \leq +w$. Similarly, the bounds for y and z are $-w \leq y \leq +w$ and $0 \leq z \leq w$, for positive values of w. These six inequalities define the six clipping planes. Suppose that the line between the points $[x_1\ y_1\ z_1\ w_1]$ and $[x_2\ y_2\ z_2\ w_2]$ crosses the $x \leq w$ clipping plane. In other words, suppose that $x_1 \leq w_1$ and $x_2 > w_2$. If the line between these two points is parameterized by t, it intersects the boundary where

$$(x_2 - x_1)t + x_1 = (w_2 - w_1)t + w_1,$$

or

$$t = \frac{x_1 - w_1}{(w_2 - w_1) - (x_2 - x_1)}.$$

The value of the parameter t can be used to find the coordinates of the point at which the line leaves the visible volume. In addition to finding this new polygon vertex, it is necessary to interpolate the normal vectors associated with the two ends of the line, giving a new normal that will be associated with the new vertex. Finally, after a polygon has been clipped, perspective division may be applied to its vertices, and the result may be passed to the visible surface algorithm.

The Phong shading algorithm described in the previous chapter gives good results, but it is difficult to compute in real time. Recall that it was necessary to interpolate and normalize the four vertex normal vectors for each visible point on the polygon before the shading model could be applied to compute the color of the interior points. An earlier method, developed by Gouraud, is substantially simpler, although it can sometimes produce undesirable effects in the final image [GOUR71]. Basically, Gouraud's technique involves computing the colors at the four polygon vertices and interpolating these colors, rather than the normal vectors, at points on the interior of the polygon. The resulting picture is smoothly shaded, but because the color interpolations have no real physical basis, moving objects may not look quite correct. Moreover, in some

cases, Mach bands may be apparent. Nevertheless, the method is so simple, and the constraints in a real time system are so rigorous, that Gouraud shading is a viable choice.

3.2 Previous Parallel Algorithms

Although commercial systems for real time display of moving shaded images tend to be implemented as pipelines of a few very fast stages, other implementations that are more parallel have also been reported. Interestingly enough, the first real time visual environment simulator made use of quite a bit of concurrency. The machine was built by General Electric in the late 1960's [SCHU69]. By relying on hardware duplicated for each polygon, it was able to simultaneously scan convert a large number of polygons. A network of combinatorial logic served to select the visible polygon for display at each pixel.

Somewhat more recently, Fuchs has proposed a machine consisting of several microprocessors operating in parallel [FUCH77, FUCH79]. In this scheme, responsibility for pixels appearing on the display is divided between the individual processors so that each processor need compute the appearance of only those pixels assigned to it. As illustrated in Figure 3–2, pixels are assigned to processors in regular, interleaved patterns that depend on the number of processors in the machine. For example, if there were only two processors, pixels might be assigned in a checkerboard pattern. In systems composed of more than two processors, each processor would be assigned a rectangular array of pixels, and the final image would be formed by interleaving the grids of pixels from the individual processors. Of course, the grid resolution would depend on the number of available processors, because the resolutions of the interleaved grids must combine to make the desired final image resolution.

Fuchs' machine finds visible surfaces by using a depth buffer, or Z buffer, algorithm running simultaneously on all of the processors [CATM74]. The arrangement of processors is shown in Figure 3–3. The algorithm begins when a transformed and clipped polygon is broadcast to the processors from some external host machine. Each processor then samples the polygon to determine the depth at every pixel within the polygon. The depth is, of course, the z value of the polygon at the pixel and is related to the distance from the viewer to the surface. The new depth values are compared with the depths of the closest surface so far encountered at corresponding pixels. The polygon is visible at pixels where the new depth is less than the old, stored depth. For these visible pixels, the processors compute a new color value and replace the old depth and color with the new ones. Finally, when a processor has finished with the current polygon, it announces completion. When all processors have finished, the host can broadcast the next polygon.

It is easy to see how the processors in Fuchs' machine achieve concurrent operation. Since all of the processors are working on the same polygon, and since the resolu-

(a)

A	B	A	B	A	B	A	B	A	B
C	D	C	D	C	D	C	D	C	D
A	B	A	B	A	B	A	B	A	B
C	D	C	D	C	D	C	D	C	D
A	B	A	B	A	B	A	B	A	B

(b)

A	B	C	D	A	B	C	D	A	B
E	F	G	H	E	F	G	H	E	F
A	B	C	D	A	B	C	D	A	B
E	F	G	H	E	F	G	H	E	F
A	B	C	D	A	B	C	D	A	B

(c)

A	B	C	D	A	B	C	D	A	B
E	F	G	H	E	F	G	H	E	F
C	D	A	B	C	D	A	B	C	D
G	H	E	F	G	H	E	F	G	H
A	B	C	D	A	B	C	D	A	B

Figure 3–2. Correspondence between processors and pixels in Fuchs' multiple microprocessor architecture. Processors are identified with letters. (a) A four processor machine. (b) An eight processor machine. (c) Another arrangement of an eight processor machine.

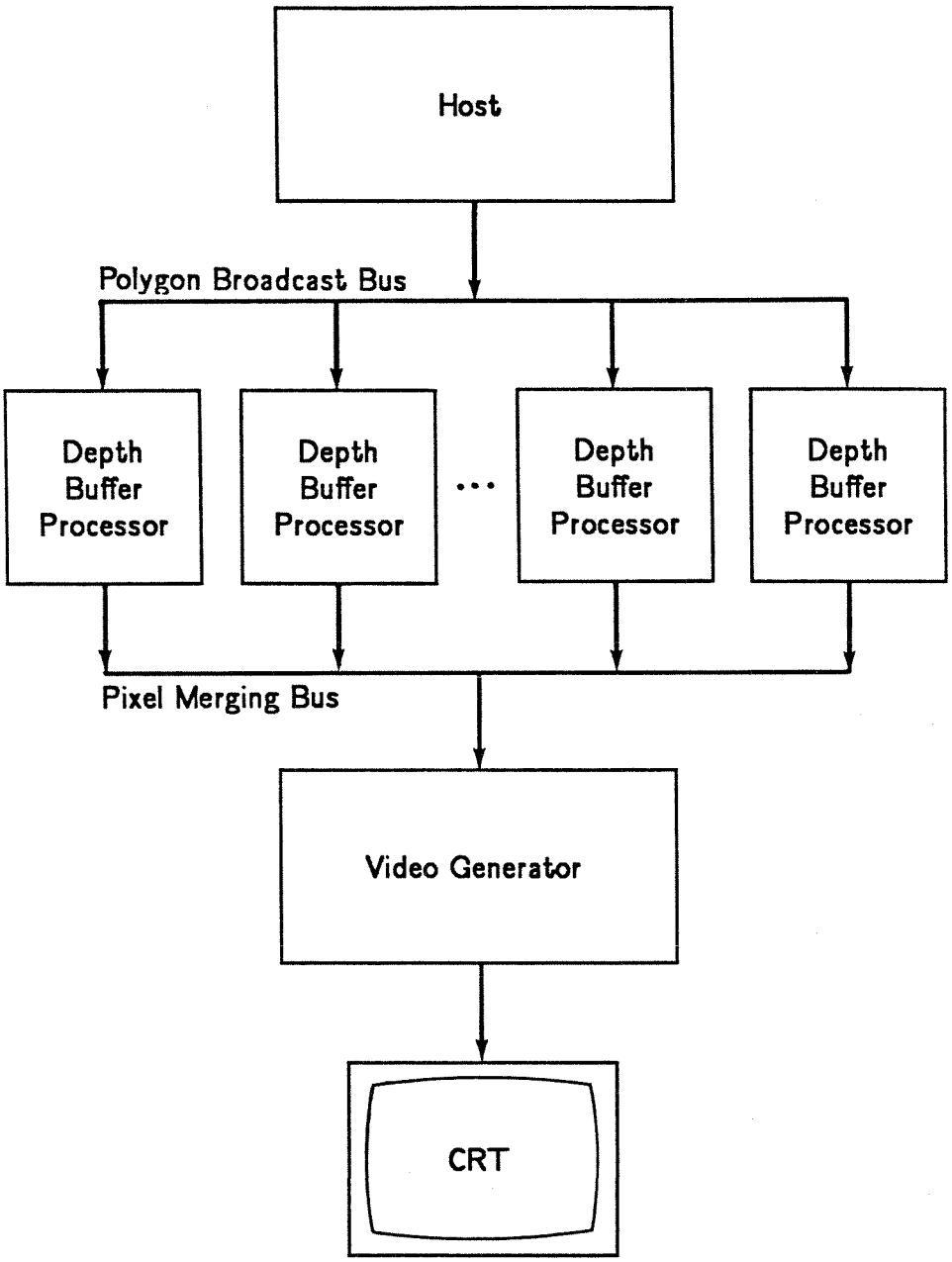


Figure 3–3. Processor interconnection plan in Fuchs' machine. The host computer broadcasts polygon descriptions to each of the smaller processing elements, where they are scan converted and stored in the depth buffer as appropriate. After all of the polygons have been processed, the final image is shifted out for display.

tion of pixels assigned to each processor is identical, every processor in the machine will have to consider approximately the same number of pixels. Unfortunately, the number of pixels that are both inside the polygon and not obscured by some other polygon may vary from processor to processor, so that the number of lighting computations performed by each processor may not be identical. The workload should, however, be similar enough that all of the processors will finish at about the same time, and therefore individual processors will probably not be idle much of the time. Because the scan conversion and lighting computation tasks performed by the processors are the most time consuming aspects of a depth buffer algorithm, the speedup should be nearly proportional to the number of processors that are available.

Parke has suggested a machine whose organization is much like that of the Fuchs machine [PARK80]. The primary difference is in the way pixels are assigned to processors. In Parke's machine, processors are responsible for contiguous blocks of pixels, as illustrated in Figure 3–4. If there were four processors, for example, each might be assigned one of the four screen quadrants. Instead of broadcasting every polygon to all of the processors, polygons are broken up according to screen position, and the resulting fragments are sent to the responsible processor. The polygons are broken by a binary tree of splitting processors, each of which performs a clipping operation; see Figure 3–5. Polygons enter at the root of the tree, which separates them into those portions appearing on, say, the left half of the screen and those appearing on the right. The two processors at the next level of the tree accept the corresponding halves and break them into the top and bottom parts. The splitting operation continues until the polygon fragments reach the leaves of the tree where they are processed by a depth buffer algorithm. Concurrency is possible because the machine can simultaneously maintain many polygons in various stages of completion. The overall processor utilization depends on the extent to which polygons are evenly distributed on the screen. Parke also proposed a hybrid scheme that uses the splitting tree down to a certain level and then broadcasts the polygon fragments to a collection of processors organized as in Fuchs' machine.

Another Fuchs invention is a system called Pixel-planes, which devotes a simple processor to each pixel of the final image [FUCH81]. This machine is also based on a depth buffer algorithm. Conceptually, each processor is capable of simultaneously evaluating a line equation with the coordinates of every pixel on the screen, besides being able to perform a depth comparison and store the values that describe the pixel. Given three coefficients, A, B, and C, each processor will substitute the xy-coordinate of the corresponding pixel into the equation $F(x, y) = Ax + By + C$. The result of this computation is interpreted at various stages in the algorithm as the distance from the pixel to the edge of a polygon, as the distance from the viewer to the polygon, and as the color of the polygon. The hardware for computing $F(x, y)$ is shared by the pixel processors in a clever way that makes the implementation more practical than it might otherwise seem.

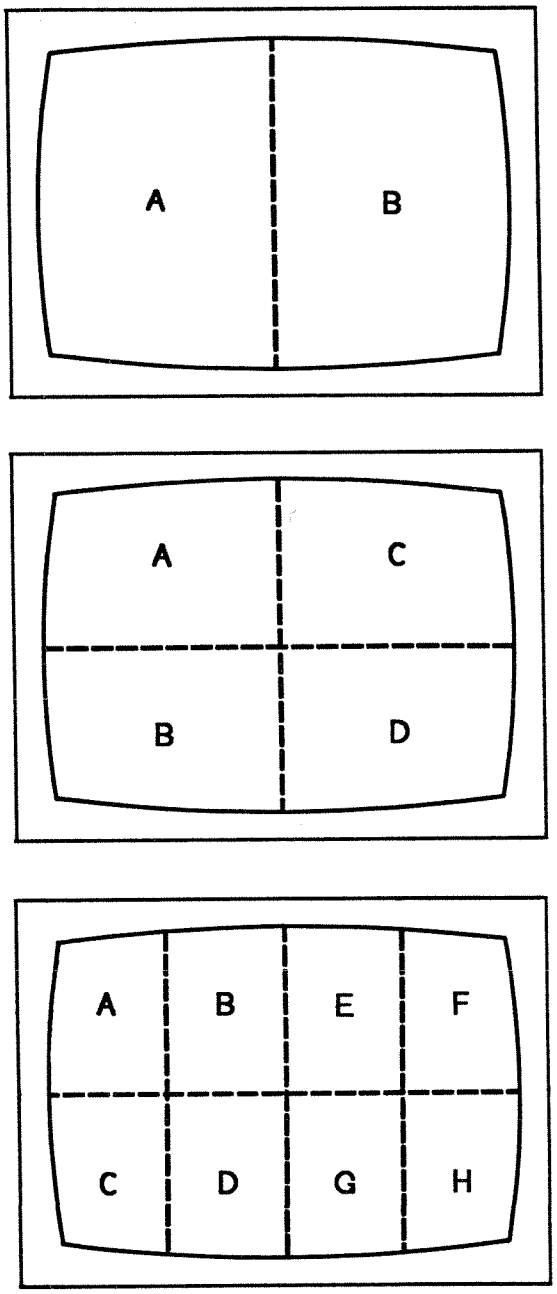


Figure 3–4. Assignment of processors to regions of the screen in Parke's machine. Top to Bottom: assignment for a machine with two processors, four processors, and eight processors.

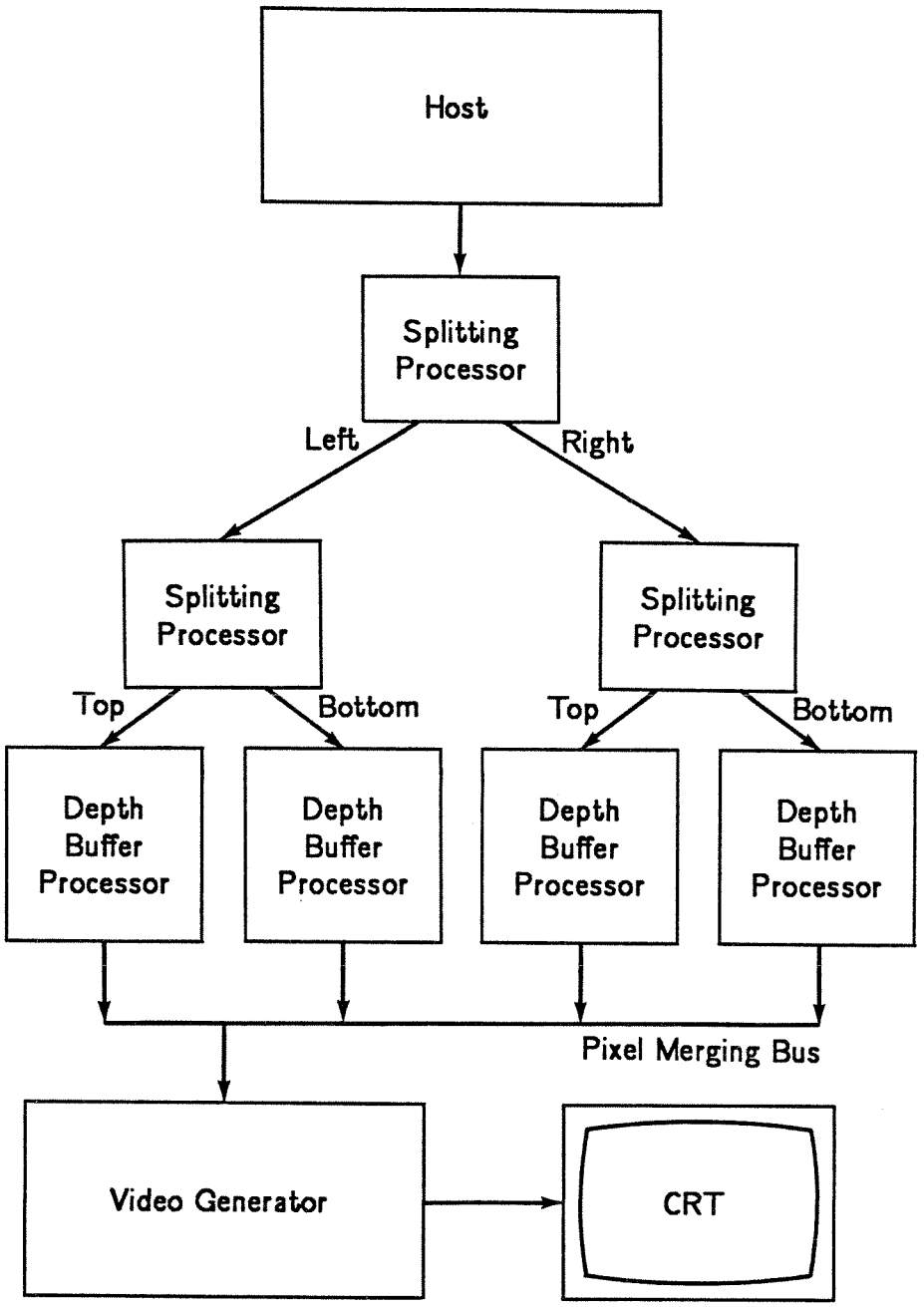


Figure 3–5. In Parke's machine, a tree of splitting processors separate polygons according to the region of the screen in which they appear. A machine with four depth buffer processors is shown here.

Once a polygon has been converted to a suitable form, three steps are required to process it with Pixel-planes. The first step serves to identify the pixels on the interior of the polygon. To do this, a host processor supplies coefficients of the line equations that describe the two-dimensional projection of the polygon edges on the screen. As each set of coefficients is processed, the value of the line equation at each pixel will be either positive or negative, depending upon whether the vertex lies to one side or the other of the line. Any pixels that lie on, say, the positive sides of all of the edges are considered to be on the inside of the polygon. After interior pixels have been identified, it is necessary to find the depth, or the distance from the viewer to each point within the polygon. Once again, using three coefficients supplied by a host machine, each processor corresponding to a point inside the polygon computes this distance for its own pixel. To solve the hidden surface problem, the processors compare this newly computed depth with the stored depth of the closest polygon so far encountered. In pixels where the new depth is closer than the old depth, the new depth replaces the old. The state of the other processors remains unchanged. Finally, the third step of the Pixel-planes algorithm computes the color of those pixels that were visible. Again, the colors are specified with the three coefficients of $F(x, y)$.

Fuchs estimates the performance of Pixel-planes at up to about 30,000 polygons per second. It can thus display a scene consisting of perhaps a thousand polygons in real time. A 16×32 array of processors is expected to fit on a single chip, which means that 512 chips would hold enough processors for a 512×512 image. This does not, of course, include circuitry for preparing the polygons in a suitable form, or for interfacing the Pixel-planes processors to a television monitor.

Demetrescu has proposed a highly parallel machine that uses a kind of dynamic depth buffer algorithm. In this machine, each polygon in the scene is assigned its own processor in a pipeline of processors, shown in Figure 3–6. The individual processors are capable of scan converting a polygon and performing some simple depth comparisons. To produce a picture using this arrangement, descriptions of pixels from the background of the scene are introduced at the beginning of the pipeline. These pixel descriptions carry depth and color information, and when a processor receives one, it compares the incoming depth with the depth of the corresponding pixel from its own polygon. It then passes the description of the visible pixel to the next processor in the pipeline. Pixel descriptions appearing at the end of the pipeline correspond to the unobscured surfaces in the scene. Note that the processing is done in scan line order, so that the output of the pipeline could conceivably be routed almost directly to a raster scan display device.

Depth buffer algorithms have two inherent limitations. Since polygons are processed sequentially, the number of polygons that can be handled in real time is determined by the time required to process a single polygon. The difficulty is somewhat mitigated by the fact that the algorithm can handle an unlimited number of polygons if given enough time. A more severe problem, however, is the aliasing caused by sampling at screen resolution without somehow first filtering the polygons. The usual way of

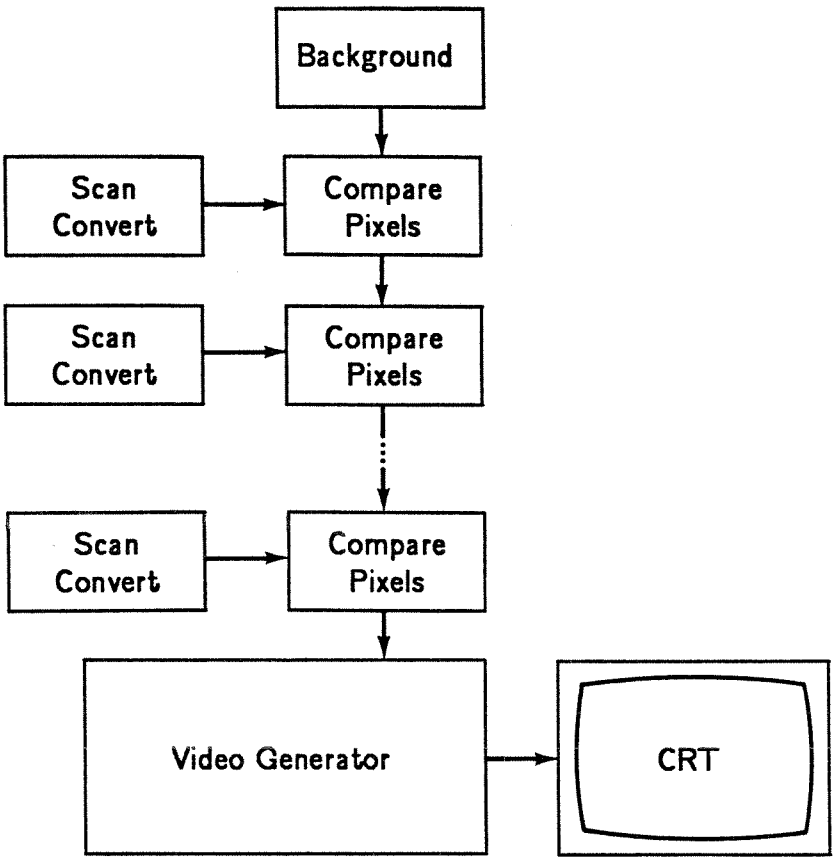


Figure 3–6. Demetrescu's machine is a pipeline of processors that can each scan convert a polygon and compare pixels from two polygons. Background pixels enter the pipe to be replaced at some stage by pixels that are closer to the viewer. Pixels visible in the final image emerge at the end.

reducing this problem is to sample the scene at a higher resolution and then to filter the high resolution image down to screen resolution for display. Unfortunately, increasing the resolution of the computation causes a dramatic increase in the processing and storage requirements since, of course, the number of pixels in an image increases as the square of the linear resolution.

3.3 A Scan Line Tree

Most of the visible surface algorithms that have been devised rely on sorting in one form or another. Software implementations use sorting to focus attention on one part of the screen at a time in order to limit the number of objects that must be considered together. Hardware implementations can use sorting to move objects so that those appearing close to each other on the screen will be stored in nearby parts

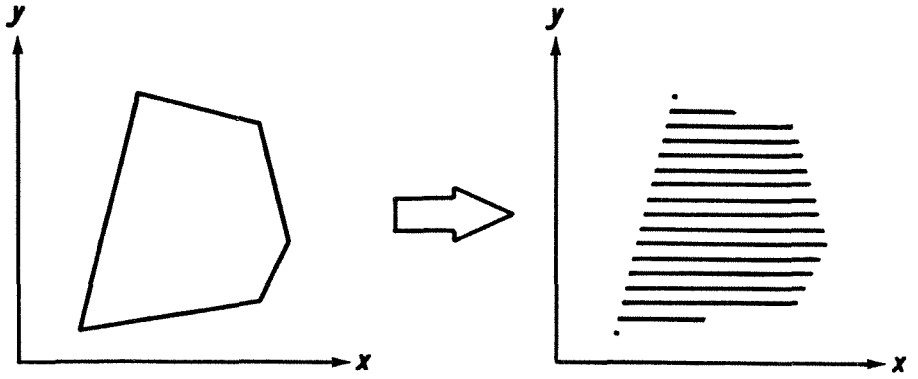


Figure 3–7. Scan conversion is the process of breaking a transformed and clipped polygon into the series of strips that appear on each line of a raster scan display.

of the machine where it is convenient to compare them. One difficulty with hardware sorting schemes is that it is sometimes necessary to move data over great distances to achieve the desired ordering. This data motion is either very slow or very expensive in terms of communications costs. Another approach is to start out with sorted streams of data and then to merge nearby streams. The scan line tree represents an attempt to make use of this alternative.

Suppose that the scene to be rendered consists of just a single polygon. The first step required to make a picture of this scene is, of course, to transform the polygon to the screen coordinate system and to clip it to the visible volume as outlined earlier. Next, if there is anything left after clipping, the transformed polygon must be scan converted. Scan conversion is the process by which the polygon is broken up into a collection of horizontal bands corresponding to the scan lines of the raster display. See Figure 3–7. Each of these bands, or segments, is represented by its two endpoints in screen coordinates and by two color vectors that are interpolated from the vertex colors. After scan conversion, each segment must be broken up into the individual pixels that are visible on the scan line. The values of pixels within the segment are interpolated between the color vectors at its two endpoints. Notice that segments are generated in a left-to-right, top-to-bottom order so that the pixel conversion takes place in the same order in which the electron beam sweeps the screen in the television monitor to produce the image.

Next, suppose that the scene consists of two polygons. Again, the first step in the

production of an image is to transform, clip, and scan convert the polygons. These operations can take place independently for each of the two polygons. Because the polygons might overlap or even intersect one another, however, they must undergo further processing before the final pixel conversion can take place. It is necessary somehow to merge the two streams of segments, as shown in Figure 3–8, in order to eliminate any segments that overlap. The two streams are ordered by screen position, making merging a relatively simple operation that compares one segment at a time from each of the two streams. If either of the segments ends before the other one starts, it may be appended directly to the output stream to be replaced with the next segment from the corresponding input stream. On the other hand, if the segments do overlap, it may be necessary to fracture one of them and pass only part of it to the output stream. The final result is a stream of non-overlapping segments, ordered from left-to-right and top-to-bottom, that are ready to be broken up into pixels for display.

The method just described for handling a two polygon scene suggests a way to produce an image of a scene containing an arbitrary number of polygons. Notice that the output of the merge operation is an ordered sequence of non-overlapping segments, which is exactly the property that the merge operation requires of its inputs. Thus, a way to make a more complicated picture is to arrange a hierarchy of merge operations. At the bottom of the hierarchy, merge operations would accept scan converted polygons as their inputs. In the rest of the hierarchy, the outputs of the merge operations would be routed to the inputs of the next higher level until, at the top of the hierarchy, the final output is produced. Like results at every other level in the hierarchy, the final result is an ordered sequence of non-overlapping segments ready to be split into individual pixels for display.

It is easy to imagine a binary tree of processors that can implement the hierarchy just described. Such a tree is illustrated in Figure 3–9. The leaves of the tree are processors that can accept a polygon, transform it to screen coordinates, and clip it to the viewing volume. These processors are all connected to a single bus through which the polygon models may be loaded. Also, the bus provides a mechanism by which the activities of the individual processors may be coordinated. After the clipping phase, each processor containing a polygon not outside the viewing volume passes the clipped polygon to a corresponding scan conversion processor. Each of these scan converters transforms a polygon into the ordered sequence of segments required by the merging processors that form the rest of the tree. The two inputs of each segment merging processor are connected either to scan converters or to other merging processors, depending on their positions in the tree. Finally, the segment merging processor at the root of the tree is connected to a fourth kind of special purpose device that splits segments into pixels and displays the image in a television monitor.

The scan line tree is similar in some ways to the pipeline suggested by Demetrescu and described earlier. If the tree is maximally unbalanced so that each node has a left descendant but none has a right one, the structure degenerates into a linear pipeline. Furthermore, if segments are restricted to be a single pixel in width, the behavior

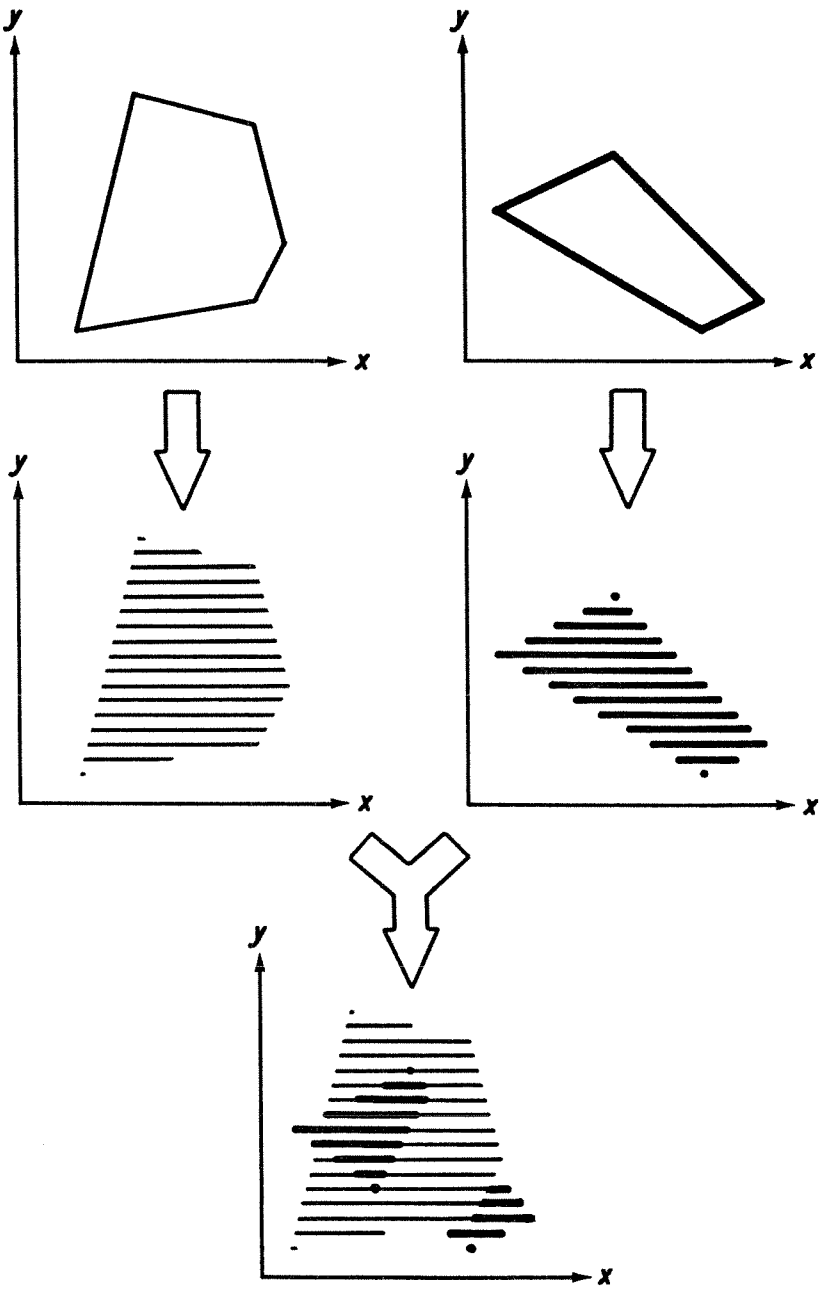


Figure 3–8. In a scene consisting of two polygons, the results of the two independent scan conversions must be merged to eliminate those portions of each polygon that are hidden by the other.

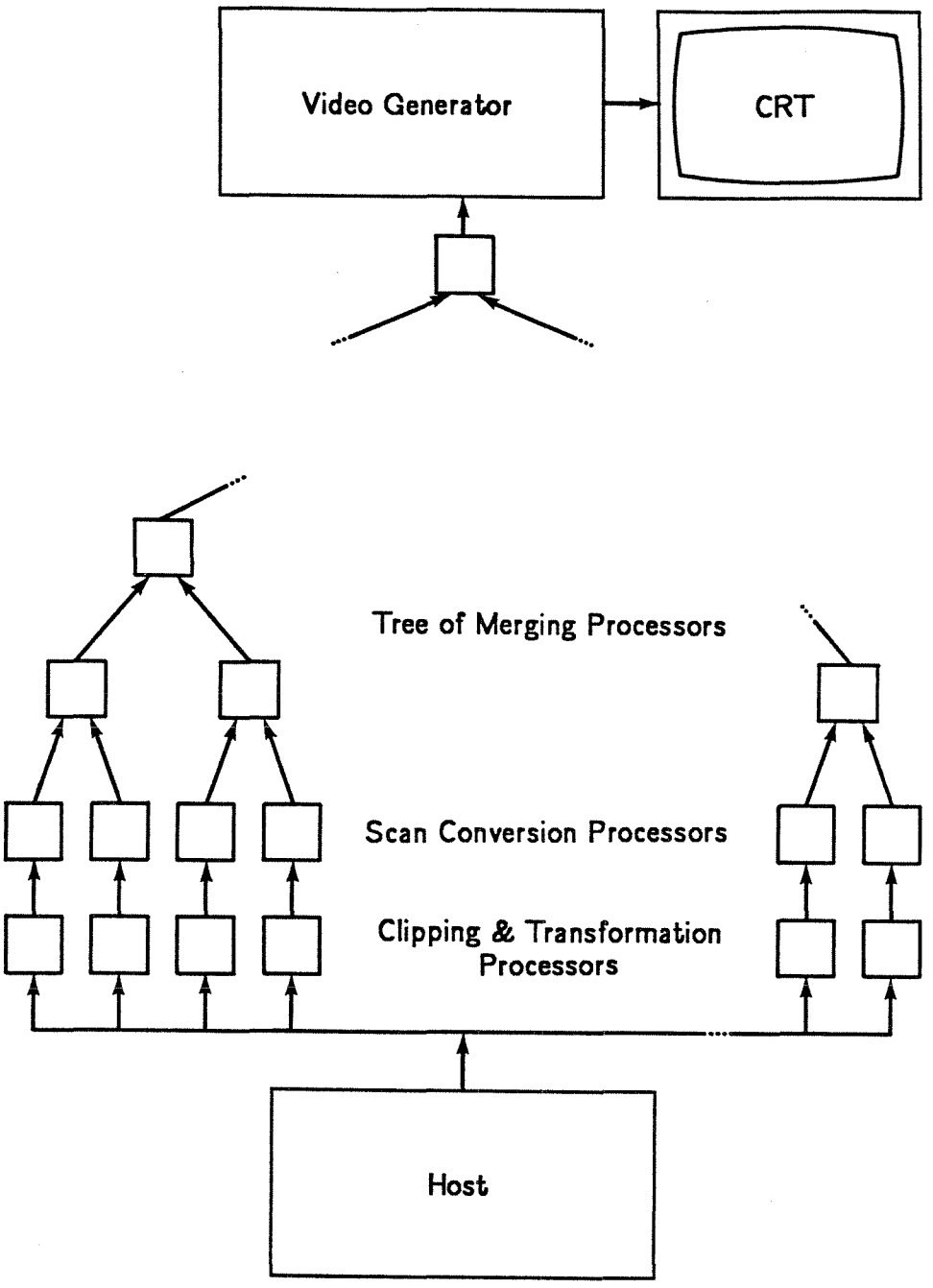


Figure 3–9. Organization of a scan line tree. Clipping and transformation processors at the leaves of the tree receive polygons from a host machine. When they have been transformed into screen coordinates and clipped to the visible region, the polygons are sent to the scan conversion processors. The segments generated here travel through the tree of merging processors, which remove any hidden segments. The remaining segments travel to the root, where they are separated into pixels for display.

of the two machines becomes essentially identical. What, then, are the advantages of introducing additional communication paths and additional complexity in the merging operation? First of all, the extra communication paths inherent in the tree structure are not wasted. They are used to increase the overall speed with which the machine can communicate with its environment. The exact mechanism involved will be described in a later section. Second, by considering segments that consist of many pixels, the total number of communications can be reduced because each communication conveys more information. Third, the number of pixels in an image increases as the square of the linear resolution, while the number of segments increases linearly. Thus, to double the final resolution while maintaining the same level of performance, the speed of the pixel processors must be increased by a factor of four, whereas the segment processors need be only twice as fast. Actually, because of the tree organization, it turns out that the extra performance can be obtained by increasing the number of processors, rather than by increasing their individual speeds. Once again, the exact technique for doing so will be presented in a later section.

3.3.1 Transformation and Clipping Processors

If the scan line tree is to operate in real time, it must be able to transform and clip all of the polygons in the scene at the rate of thirty times per second. However, since a transformation and clipping processor has been devoted to each polygon, the polygons may be dealt with independently and in parallel, giving each processor a full thirtieth of a second to complete its single polygon. This is a fairly generous amount of time in which to perform the required task, and it therefore seems as though fully pipelined processors like the ones used in the ray tracing machines are unnecessary.

One alternative to the fully pipelined approach is to employ a more programmed architecture. In this case, the various arithmetic operations that implement transformation and clipping would share a few arithmetic devices under program control. The difficulty with this scheme is that the program is duplicated within each processor. Although the algorithm that must be performed is reasonably simple, it is nevertheless complicated enough that its representation would form a substantial portion of each processor.

A third approach to the design of the transformation and clipping processors is to concentrate all of the program logic at the end of the bus connecting the processors, duplicating only the control and computation logic within each processor. Thus, the collection of processors might be thought of as a single instruction stream, multiple data stream (SIMD) computer [FLYN66]. Of course, some care must be exercised when designing the instruction set to insure that the desired application program may be executed by all of the processors in lockstep.

The polygon transformation task presents no difficulty for a SIMD architecture. All polygons have four vertices at this stage, and each vertex may be transformed

m_{11}:= **literal**; m_{21}:= **literal**; m_{31}:= **literal**; m_{41}:= **literal**;
m_{12}:= **literal**; m_{22}:= **literal**; m_{32}:= **literal**; m_{42}:= **literal**;
m_{13}:= **literal**; m_{23}:= **literal**; m_{33}:= **literal**; m_{43}:= **literal**;
m_{14}:= **literal**; m_{24}:= **literal**; m_{34}:= **literal**; m_{44}:= **literal**;
$x := x_1 m_{11} + y_1 m_{21} + z_1 m_{31} + m_{41}$;
$y := x_1 m_{12} + y_1 m_{22} + z_1 m_{32} + m_{42}$;
$z := x_1 m_{13} + y_1 m_{23} + z_1 m_{33} + m_{43}$;
$w := x_1 m_{14} + y_1 m_{24} + z_1 m_{34} + m_{44}$;

Figure 3–10. Program to load a transformation matrix and transform the first vertex.

by means of a vector-by-matrix multiplication, implemented as a fixed sequence of operations, without loops or conditionals, as illustrated in Figure 3–10. When transforming four vertices, the operations can be arranged so that each element of the transformation matrix need be broadcast only once, even though it is used four times. This alteration does not affect the straight-line property of the program, but it does suggest that the speed at which literals are transmitted does not have to be so great as the speed of transmitting individual instructions.

In addition to transforming the polygon vertices to the screen coordinate system, the transformation and clipping processors must compute the color at each polygon vertex. Again, the program for implementing this computation may be written as straight line code suitable for a SIMD architecture. A simplified lighting model will be used to reduce the number of operations required in the calculation of the four corner colors. The intensity of one component of a vertex color vector will be given by

$$I = (I_a + I_p \cos i)R,$$

where

I = Final intensity value.
I_a = Intensity of the ambient illumination.
I_p = Intensity of the point source illumination.
i = Angle between the incident ray and the surface normal vector.
R = Reflectance of the polygon.

As usual, the cosine is computed as a dot product, and the three components of the color vector are treated identically. A program for computing the color of the first vertex is shown in Figure 3–11.

After the processors have transformed the polygons to screen coordinates and determined the vertex colors, they must clip the new polygons to the viewing volume. Unfortunately, the clipping process is not quite as simple as the transformation process was. The difficulty centers around the fact that the clipped polygons do not necessarily have exactly four vertices, as they do before clipping. Moreover, different polygons in the scene may have different numbers of vertices. Those polygons that lie outside

a_r:= **literal**; a_g:= **literal**; a_b:= **literal**;
p_r:= **literal**; p_g:= **literal**; p_b:= **literal**;
p_x:= **literal**; p_y:= **literal**; p_z:= **literal**;
$d := \max(0, n_{1x}p_x + n_{1y}p_y + n_{1z}p_z)$;
$r_1 := (a_r + p_r d)c_r$;
$g_1 := (a_g + p_g d)c_g$;
$b_1 := (a_b + p_b d)c_b$;

Figure 3–11. Program to compute the color at the first vertex given the color of the ambient illumination, $[a_r\ a_g\ a_b]$, the color of the point light source, $[p_r\ p_g\ p_b]$, and the direction of the point light source, $[p_x\ p_y\ p_z]$. The colors at the other three vertices are computed in a similar manner.

the visible region will end up having no vertices at all. The implication of all this is that the transformation and clipping processors cannot all execute exactly the same sequence of instructions to manipulate their corresponding polygons. There must be some mechanism for routing different instruction streams to different processors in the machine.

Fortunately, there is a fairly simple technique which can be used to handle the clipping situation. Processors must be capable of ignoring selected sections of the instruction stream. To implement this behavior, there will be an instruction that will test some state of the processor and, if the proper conditions are met, will cause the processor to ignore any further instructions. A signal on a reset line connected to all of the processors will restore normal operation. Thus, processors may be shut down selectively, but must be reactivated together. To see how this mechanism will work, suppose that there is some action to be performed with each vertex of each polygon in the scene. This task could be programmed as a loop that incremented a vertex index with each iteration. At the beginning of the loop, the vertex index must be compared with the number of vertices in each polygon. Processors whose vertex index exceeds the number of vertices will be instructed to shut down. The remaining processors will receive and execute a stream of instructions to perform the desired operation on the vertex. Finally, at the end of the loop, the vertex indices must be incremented before beginning the next iteration. After the last iteration, the dormant processors must be reactivated by means of the reset signal. Notice that the stream of instructions forming the body of the loop must be broadcast enough times to insure that all of the processors have finished with all of the vertices and are therefore in the dormant state. In the clipping algorithm, it turns out that it is always possible to determine the maximum number of times that each loop may iterate.

The clipping method to be used here is based on Sutherland and Hodgman's Reentrant Polygon Clipping Algorithm [SUTH74a]. This technique results from the observation that the process of clipping a polygon to a six-faced bounding volume may be separated into a sequence of six simpler operations in which the polygon is clipped to a single bounding plane. In the original description of the algorithm, these six stages were pipelined either in software or in hardware to reduce the intermediate storage

requirements, increase performance, or both. In the transformation and clipping processor, however, where computing power is far more costly than memory, the six stages of the clipping algorithm will be performed in a strictly sequential manner. That is, a polygon will be clipped against the first boundary plane, forming a new polygon to be clipped against the second boundary plane, and so on, until all six boundary planes have been checked.

As noted earlier, it is always possible to determine in advance how many iterations will be needed to implement a particular loop in the clipping algorithm. For example, when clipping the original convex quadrilateral against the first boundary plane, there are, of course, four vertices to be examined. The clipping process may introduce an extra vertex, so that when clipping against the second boundary plane, the loop might execute at most five times. In fact, each stage of the clipping algorithm may potentially add a vertex to the polygon, which means that after the final stage, the polygon might have as many as ten vertices.

A program for clipping against the first boundary plane, $x \leq w$, is shown in Figure 3–12. The program maintains two lists of vertices. The input list contains transformed vertices from the original quadrilateral, and the output list, which is initially empty, will hold the vertices of the partially clipped polygon. This output list constitutes the input to the next stage of the algorithm. In the program shown, the body of the loop individually examines successive vertices around the perimeter of the polygon. If the previous vertex was invisible, but the current one lies within the clipping volume, then the polygon edge between them passes through the clipping plane. In this case, the intersection point is a vertex of the clipped polygon, and it must therefore be computed and appended to the output vertex list. Notice that it is also necessary to interpolate the vertex color vectors, as well as the vertices themselves. Next, if the current vertex is visible, it too must be copied to the output list. Finally, the index of the input polygon vertex must be incremented in preparation for the next iteration of the loop.

One rather annoying property of the clipping program as just presented is that it must test the same loop termination condition more than once in the course of a single iteration. This is because the mechanism for deactivating processors cannot associate a particular reset command with the corresponding command that originally turned the processor off. That is, there is no way to undo a selected shutdown command. Of course, it would be possible to add such a feature, but doing so is not really critical for the proposed application, and it therefore seems better to retain the original, less elaborate mechanism.

A more severe problem occurs in connection with the intersection computation. Notice that most of the instructions given for the clipping algorithm are involved in performing this operation. Note also that since the instructions occur within the body of the loop, they must be broadcast to the processors during every iteration of the loop. Thus, for the first stage of the clipping algorithm, the intersection instructions must be broadcast four times. However, a convex polygon clipped against a plane

n:= 0; **comment**: Number of output vertices. ;
i := 1; **comment**: Index of the current input vertex. ;
j:= 4; **comment**: Index of the previous input vertex. ;
repeat 4 times
  if $i > 4$ **then shutdown**;
  if $x_u[j] \leq w_u[j]$ **then shutdown**;
  if $x_u[i] > w_u[i]$ **then shutdown**;
  comment: Previous vertex was invisible and current vertex is visible,
    so the edge crosses the boundary plane. ;
  t:= $\dfrac{w_u[j] - x_u[j]}{(x_u[i] - x_u[j]) - (w_u[i] - w_u[j])}$;
  n:= $n + 1$;
  $x_c[n]$:= $(1-t)x_u[j] + tx_u[i]$; $r_c[n]$:= $(1-t)r_u[j] + tr_u[i]$;
  $y_c[n]$:= $(1-t)y_u[j] + ty_u[i]$; $g_c[n]$:= $(1-t)g_u[j] + tg_u[i]$;
  $y_c[n]$:= $(1-t)z_u[j] + tz_u[i]$; $b_c[n]$:= $(1-t)b_u[j] + tb_u[i]$;
  $w_c[n]$:= $(1-t)w_u[j] + tw_u[i]$;
  reset;
  if $i > n$ **then shutdown**;
  if $x_u[i] > w_u[i]$ **then shutdown**;
  comment: Current vertex is visible. ;
  n:= $n + 1$;
  $x_c[n]$:= $x_u[i]$; $y_c[n]$:= $y_u[i]$; $z_c[n]$:= $z_u[i]$; $w_c[n]$:= $w_u[i]$;
  $r_c[n]$:= $r_u[i]$; $g_c[n]$:= $g_u[i]$; $b_c[n]$:= $b_u[i]$;
  reset;
  j:= i; i:= $i + 1$;

Figure 3–12. Program to clip the polygon against the $x \leq w$ plane. The subscripts u and c refer to unclipped and clipped vertices.

can intersect that plane in at most two points. As written, the clipping algorithm must transmit the intersection code two extra times to compensate for the fact that it does not determine in advance which segments actually cross the clipping boundary. Later stages in the clipping algorithm transmit this code even more wastefully. A more conservative approach merely makes a note of the necessary intersection computations as it examines the polygon vertices, but it does not actually perform the computations at that time. The code for making these notes is substantially smaller than the intersection code, so that the loop iteration should execute quite a bit more quickly. Finally, after all of the iterations have been completed, just two transmissions of the intersection code will complete the clipping stage.

A problem related to the wasteful transmission of intersection code is the fact

n_c:= 0; **comment**: Number of vertices in the clipped polygon.
i:= 0; **comment**: Index of the current vertex.
j:= n_u; **comment**: Index of the previous vertex.
n_s:= 0; **comment**: Number of boundary intersections.
repeat 5 **times**
  if $i > n_u$ **then shutdown**;
  if $x[i_u[j]] \geq -w[i_u[j]]$ **then shutdown**;
  if $x[i_u[i]] < -w[i_u[i]]$ **then shutdown**;
  comment: The edge from i to j passes into the visible region.
    Record the presence of an intersection. ;
  n_c:= $n_c + 1$; n_v:= $n_v + 1$; $i_c[n_c]$:= n_v;
  n_s:= $n_s + 1$;
  $s_i[n_s]$:= $i_u[i]$; s_j:= $i_u[j]$; s_n:= n_v;
  reset;
  if $i > n_u$ **then shutdown**;
  if $x[i_u[i]] < -w[i_u[i]]$ **then shutdown**;
  comment: The current vertex is visible. ;
  n_c:= $n_c + 1$; $i_c[n_c]$:= $i_u[i]$;
  reset;
  j:= i; i:= $i + 1$;
comment: Now perform the actual intersections. ;
repeat 2 **times**
  if $n_s = 0$ **then shutdown**;
  i:= $s_i[n_s]$; j:= $s_j[n_s]$; n:= $s_n[n_s]$;
  t:= $\dfrac{-(w[j] + x[j])}{(x[i] - x[j]) - (w[j] - w[i])}$;
  $x[n]$:= $(1-t)x[j] + tx[i]$; $n_x[n]$:= $(1-t)n_x[j] + tn_x[i]$;
  $y[n]$:= $(1-t)y[j] + ty[i]$; $n_y[n]$:= $(1-t)n_y[j] + tn_y[i]$;
  $z[n]$:= $(1-t)z[j] + tz[i]$; $n_z[n]$:= $(1-t)n_z[j] + tn_z[i]$;
  n_s:= $n_s - 1$;
  reset;

Figure 3–13. Improved program to clip the polygon against the $x \geq -w$ boundary. The three s arrays identify polygon edges that will have to be clipped against the boundary.

i:= 1;
repeat 10 **times**
  if $i > n_c$ **then shutdown**;
  $x[i_c[i]] := \frac{x[i_c[i]]}{w[i_c[i]]}$; $y[i_c[i]] := \frac{y[i_c[i]]}{w[i_c[i]]}$; $z[i_c[i]] := \frac{z[i_c[i]]}{w[i_c[i]]}$;
  i:= i + 1;
  reset;

Figure 3–14. Program to perform the perspective division.

that the vertices are copied from the input list to the output list. Since vertices are represented by the four components of the vertex point and the three components of the vertex color, it is necessary to move a total of seven values. The code for copying vertices must be transmitted even when the values are never actually moved. One way of reducing this overhead is to impose an extra level of indirection in the identification of vertices. That is, the input and output vertex lists will now contain indices of points, rather than the points themselves, so that a vertex may be moved from the input list to the output list simply by copying its index. A program incorporating these revisions is shown in Figure 3–13, using the second, $x \geq -w$ clipping boundary as an example. The other stages of the clipping algorithm will remain analogous.

Once the clipping algorithm has been executed for all six bounding planes, the perspective division may be applied to the vertices of the clipped polygon. Recall that this operation is performed by dividing the x, y, and z components of the vertex points by the associated w component. The code is shown in Figure 3–14. The perspective division has the effect of scaling the points so that their x and y components range from -1 to $+1$ and their z components lie between zero and one. Thus, after suitably denormalizing the floating point representations of the vertices, it is convenient for representing points in subsequent computations to treat the remaining fractional parts as fixed point numbers.

Strictly speaking, once it has performed the perspective division, the task of the transformation and clipping processor is complete; however, the design of the scan conversion processor can be substantially simplified if the polygon that it receives has been preprocessed into a suitable form. Briefly, the preparation involves sorting the vertices by screen position and then finding some slopes so that the computations in the scan conversion processor may be performed incrementally. Vertex sorting is simplified by the fact that the polygons are convex. First, it is necessary to find the topmost vertex that appears on the screen. This is defined to be the first vertex encountered in the raster scan. Next, the program must substitute some coordinates into a line equation to determine whether the vertices are arranged in a clockwise or counter-clockwise order when viewed on the screen. This fact distinguishes the left edge leaving the topmost vertex from the right edge. Finally, the vertices on the left

and right edges of the polygon are traced separately from the top of the polygon to the bottom. As each vertex is traced, the slope of the edge connecting it with the vertex above it is computed, and the results are passed along to the scan conversion processor. Notice that the top and bottom vertices must be processed twice, because each is considered to be part of both the left and right sides of the polygon. Figure 3–15 shows a program that preprocesses a polygon for the scan conversion processor.

To get a better idea of how complicated the transformation and clipping processors must be, it is useful to scrutinize the code that they must execute. The memory requirements are the simplest to analyze. The processor must, of course, be able to store the untransformed representation of the polygon. This is represented as four vertex points, four normal vectors, and a color vector for a total of nine vectors having three components each. Each component requires a floating point number, so that the initial polygon uses up twenty-seven words of storage. Next, after transformation, the polygon vertices are represented in homogeneous coordinates with four floating point values per vertex. Adding the three components of the color vector brings the total after transformation to seven words of storage for each vertex. There are initially four vertices after transformation, and each of the six stages of the clipping process might potentially create two more, so the storage required for the transformed points is sixteen vertices, or 112 words. The miscellaneous storage used in the program takes, perhaps, a couple dozen words. Thus, it seems as though 256 words of memory is a comfortably adequate amount. Each word holds a 32-bit floating point number, so that the total amount of memory would be 8K bits. It is not unreasonable to fabricate this much memory on a single integrated circuit.

Examination of the programs that must run on the transformation and clipping processor reveals that they are composed mainly of floating point operations, with the addition of a few comparison operations. Notably absent are any kind of control operations like subroutine calls or jumps, since the program control has been factored out to the end of the processor bus. It is also apparent that array indexing is a common way of accessing memory. It seems appropriate, therefore, to postulate a processor with a floating point arithmetic unit, a floating point accumulator, and several index registers capable of operations like incrementing and decrementing. Instructions would contain single eight-bit addresses that could be modified by one of the index registers. The complexity of such a processor appears to compare favorably with that of some of the larger existing microprocessors, and hence it seems reasonable to expect that the transformation and clipping processor could be implemented as a single integrated circuit. It does not require a much further leap of imagination or technology to expect that the processor could fit on the same chip as its memory.

The overall performance of the transformation and clipping processor may be estimated by counting the number of instructions that it executes. The instruction time is dominated, in turn, by the floating point operation time. Recall that this latter quantity was estimated in the previous chapter at about five microseconds for a basic, shift-and-add style of floating point operation. It seems reasonable, then, to assume

comment: Find the leftmost top vertex. ;
t:= 1; i:= 2;
repeat 9 **times**
  if $i > n_c$ **or** $y[i_c[i]] \neq y[i_c[t]]$ **or** $x[i_c[i]] \geq x[i_c[t]]$ **then shutdown**;
  t:= i;
  reset;
  if $i > n_c$ **or** $y[i_c[i]] \geq y[i_c[t]]$ **then shutdown**;
  t:= i;
  reset;
  i:= $i + 1$;
comment: Determine the ordering of the points. ;
t_-:= $(t-1)$ **mod** n_c; t_+:= $(t+1)$ **mod** n_c;
a:= $y[i_c[t_-]] - y[i_c[t]]$;
b:= $x[i_c[t]] - x[i_c[t_-]]$;
c:= $x[i_c[t_-]]y[i_c[t]] - x[i_c[t]]y[i_c[t_-]]$;
d:= $\mathrm{sgn}(a\,x[i_c[t_+]] + b\,y[i_c[t_+]] + c)$;
if $d \geq 0$ **then shutdown**;
s:= t_-; t_-:= t_+; t_+:= s;
reset;
j:= t_-; i:= t_+; c:= 1;
repeat 12 **times**
  if $c \neq 3$ **or** $y[i] \leq y[j]$ **then shutdown**;
  c:= 4;
  reset;
  if $c \neq 2$ **or** $y[i] < y[j]$ **then shutdown**;
  c:= 3; i:= t; j:= t_-; d:= $-d$;
  reset;
  if $c = 4$ **then shutdown**;
  q:= $1/\,\mathrm{quant}(y[j] - y[i])$;
  send $\mathrm{quant}(y[i_c[i]])$;
  send $\mathrm{quant}(x[i_c[i]])$; **send** $(\mathrm{quant}(x[i_c[j]]) - \mathrm{quant}(x[i_c[i]]))q$;
  send $\mathrm{quant}(z[i_c[i]])$; **send** $(\mathrm{quant}(z[i_c[j]]) - \mathrm{quant}(z[i_c[i]]))q$;
  send $\mathrm{quant}(r[i_c[i]])$; **send** $(\mathrm{quant}(r[i_c[j]]) - \mathrm{quant}(r[i_c[i]]))q$;
  send $\mathrm{quant}(g[i_c[i]])$; **send** $(\mathrm{quant}(g[i_c[j]]) - \mathrm{quant}(g[i_c[i]]))q$;
  send $\mathrm{quant}(b[i_c[i]])$; **send** $(\mathrm{quant}(b[i_c[j]]) - \mathrm{quant}(b[i_c[i]]))q$;
  reset;
  if $c \neq 1$ **then shutdown**;
  c:= 2;
  reset;
  j:= i; i:= $i + d$ **mod** n_c;

Figure 3–15. Program to pre-process polygons for the scan conversion processor.

Vertex Transformation	272
Color Computation	153
Clipping	1904
Perspective Division	152
Scan Conversion Preprocessing	1171
Total	3652

Figure 3–16. Number of instructions consumed by each part of the program running in the clipping and transformation processors.

an instruction time of some six or seven microseconds. At thirty frames per second, there is time to execute about 5000 instructions during each frame time. The program to transform, clip, and preprocess a polygon takes only about 3652 instructions; the breakdown according to task is shown in Figure 3–16. These counts were derived by assuming a simple machine architecture, hand compiling the programs, and then counting the resulting instructions.

Notice that the basic program run by the clipping and transformation processors uses only 3652 of the approximately 5000 instructions that can be executed every thirtieth of a second. The nearly 1400 remaining instructions may be put to good use. For example, if the scene consists of a several independently moving objects, the extra time may be used to implement this motion. By successively disabling the appropriate processors, it is possible to load different transformation matrices for selected polygons in the scene. Presumably, these polygons are the ones that should move in relation to the others. After every transformation matrix has been loaded, the code for applying the transformations and then clipping the polygons may be broadcast to all of the processors for simultaneous execution.

3.3.2 Scan Conversion Processors

The purpose of a scan conversion processor is to break the polygon received from the transformation and clipping processor into a sequence of segments. Each of these segments is that portion of a polygon appearing on a single scan line. Segments are represented by eleven fixed point numbers: a y value identifies the scan line; a pair of x and z coordinates represent the two endpoints of the segment, giving horizontal position and depth; and two vectors specify the colors at each end of the segment. As the segments are generated from the polygon description, they are passed on to the merging processors at the leaves of the tree for comparison with segments from other polygons.

Because the transformation and clipping processor has prepared the polygon description in a suitable form, the design of the scan conversion processor is com-

paratively uncluttered. The processor consists of two essentially identical halves that calculate successive values for the left and right endpoints of the segment. A device shared between the two halves maintains the current y value, or scan line identity, and each of the two endpoint trackers contains registers to hold the five other values that describe segment endpoints. Corresponding adders apply the increments that update the values for successive scan lines. Comparators determine when an edge description has been exhausted and a new one must be obtained. Finally, there is a memory to hold the left and right edge lists that describe the polygon. Entries from these lists will be loaded into the various registers as required to scan convert the polygon. Each entry consists of eleven numbers, and there may be at most ten such entries. Since each number is represented by twelve bits, the overall memory requirement is thus less than 2K bits, including space for other miscellaneous information.

The operation of the scan conversion processor is fairly straightforward. At the beginning of every frame, it copies the coordinates of the topmost polygon vertex from the first entry in each list into the registers for both the left and right edges of the segment. It then loads the increment registers from the second entries in these edge lists. Recall that one of the components in an edge list entry is the y coordinate of the bottom of the edge, where the next edge begins. In order to correctly handle horizontal edges, the scan conversion processor must at this point compare these bottom y values with the y values of the current scan line. If the bottom of either edge has been reached, new values for x, z, and the color vector components must be loaded from the current entry in the corresponding edge list, and new increments must be obtained from the next list entry. If it was indeed necessary to update the edge on either side of the polygon, the edge termination test must be repeated. Once the processor has established that both edges of the polygon are correctly represented, it generates a segment from the current x and z values for each end of the polygon and the corresponding color vectors. Having done so, it must repeat the edge termination test before generating the next segment. The processor continues until it has emptied the two edge lists at the bottom of the polygon.

The relevant performance figure for the scan conversion processors is the time required to communicate a segment to the merging processor. The computation performed within the scan conversion processor is so minimal that the communication time is dominant. Since each segment is represented by its y-coordinate and the x, z, and color vector components at each of its endpoints, a segment description requires eleven twelve-bit values. It is clearly infeasible to send these 132 bits in parallel, but the choice between sending the words in sequence or dedicating a bit to each word and sending them serially is less obvious. The bit serial approach will be taken here on the assumption that the required encoding and decoding hardware at each end of the communication will be simpler. At about 150 nanoseconds per bit, the complete transfer should take a little under two microseconds.

It may at first seem a bit wasteful to design memory into the scan conversion processor when the clipping and transformation processor has memory that already

holds the same information. Nevertheless, there are two reasons for proceeding as described here. The first is that the extra memory allows the transformation and clipping operation to be pipelined with scan conversion, giving each processor a full frame time to complete its assigned task. If the memory were shared, the two processors would have to operate in sequence. The second and more important reason for duplicating the memory is that this organization permits the same transformed and clipped polygons to be scan converted more than once if, for some reason, new versions could not be readied in the time that is available. If the scene were being drastically modified, for example, there might not be enough time to both load and transform the altered polygons in a single frame time. Also, if the scene contained too many moving objects, it might not be possible to apply all of the motion transformations quickly enough. Since there is an option for displaying successive frames more than once, these problems will not cause the screen to flicker, although they may result in rather jerky motion.

3.3.3 Merging Processors

The merging processors are somewhat simpler than one might expect, although not quite so simple as one might desire. Recall that the merging processors accept two sequences of segments and produce a third from them. In each of the two input sequences, segments are expected to be non-overlapping and to occur in scan line order. This means that within each sequence, one segment must end before the beginning of the next, and that segment beginnings must be arranged in the order that they would be scanned by the raster beam. Using segments from these two input streams, the merging processor forms a single output segment stream that satisfies the same ordering and non-overlapping constraints as its inputs. Moreover, the output sequence of segments satisfies the additional constraint that it contain all of the visible segments and portions of segments from both of the two input streams. In order to satisfy this extra requirement, the merging processor may have to break segments into several pieces. Notice that the output of the scan conversion processor meets the specifications for inputs to merging processors, because the segments are generated in scan line order and because segments from a single polygon cannot overlap. Thus, identical copies of merging processors can be used at the leaves of the tree, where they accept inputs from scan conversion processors, and at levels closer to the root, where they accept inputs from other merging processors.

The basic strategy of a merging processor is to consider only one segment at a time from each of the two input streams. The merging processor operates by comparing one segment from the left input stream with one from the right to produce a sequence of one or more output segments made up of the visible portions of the inputs. As input segments are consumed, they are replaced by the next segment in the sequence from the corresponding subtree. When one of the segment sequences is exhausted, the

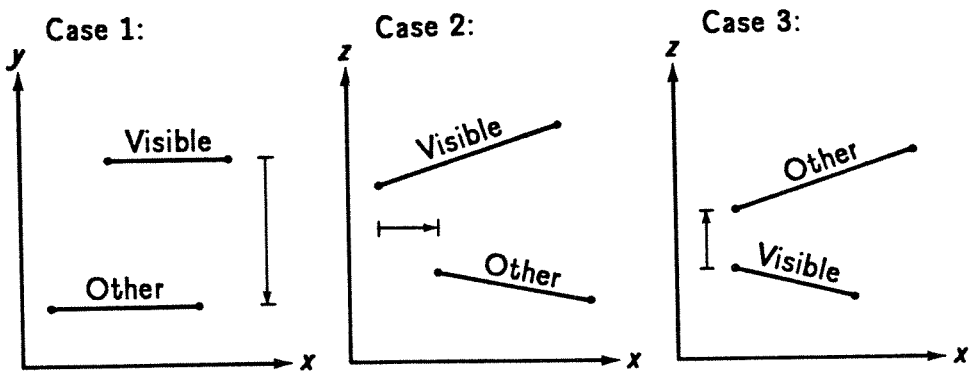


Figure 3–17. Step 1 of the Merging Algorithm: Classify the segments. The left endpoint of the visible segment is either (1) on an earlier scan line than the left endpoint of the other segment; (2) on the same scan line, but to the left of the other segment; or (3) on the same scan line as the other segment and aligned with it, but in front of it. Note the difference in coordinate axes.

output stream may be copied directly from the other segment sequence, since nothing remains to hide its segments. One way of implementing this termination condition is to place at the end of each sequence a dummy segment that does not appear on the screen.

Once it has loaded a segment from each subtree, the merging processor must determine which of these segments is first visible. That is, it must determine which of the segments will be the first to be displayed in the raster scan. To make this determination, the processor compares the left endpoints of each segment, as shown in Figure 3–17. If one endpoint appears on an earlier scan line than the other, then of course, it will be the first to be scanned. If the two segments appear on the same scan line, the leftmost one is first. If the beginnings of the two segments coincide, then the one closer to the viewer, having the smaller value of z, must be chosen. Finally, if the two segments start at exactly the same point, an arbitrary choice may be made. At first, it may seem that the task of determining the first visible segment is rather complicated, but the choice requires only a single integer comparison. The two numbers for comparison are constructed from the two left endpoints by concatenating coordinate values. For each endpoint, the negative y coordinate supplies the high order bits, the x coordinate goes in the middle, and the z coordinate makes up the low order part of the word. The segment with the smaller of these combined values is the first to be encountered in the raster scan. It will be called the visible segment in the discussions to follow, and the other segment will be called the other segment.

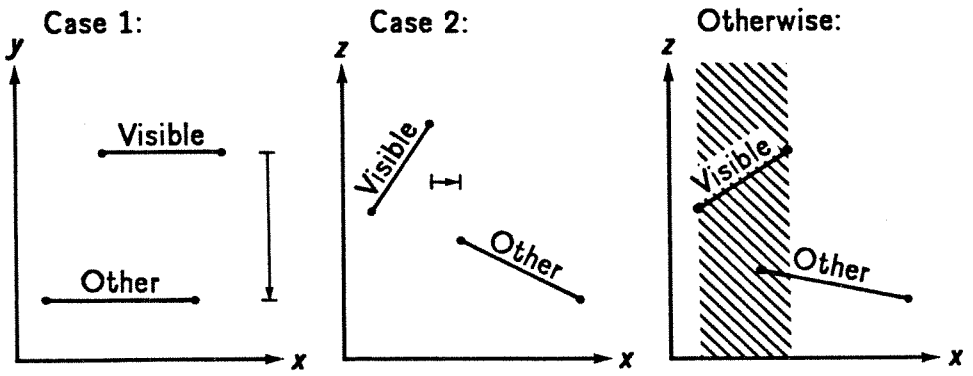


Figure 3–18. Step 2 of the Merging Algorithm: Check for non-overlapping segments. If the visible segment is either (1) completely above the other segment or (2) completely to its left, then the segments do not overlap. Otherwise, the segments overlap and the left end of the other segment lies within the shaded region.

After identifying the visible and other segments by examining their left endpoints, the next two points of concern to the merging processor are the next two points in the raster scan at which the visible segment may become invisible: the right endpoint of the visible segment and the left endpoint of the other segment. If the visible segment ends before the other segment begins, then it is not obscured at all and can be placed directly into the output stream. The test for determining this condition, shown in Figure 3–18, is the same kind of integer comparison that was initially used to identify the visible segment. If the visible segment is indeed copied intact to the output stream, it must be replaced with the next segment in the input stream from which it was originally obtained. This action reinitializes the algorithm, and another test must be performed to identify the visible segment, as described above.

If the visible segment does not end before the other segment begins, then the segments overlap, and the visible segment might pass behind the other segment somewhere along its length. To check for this condition, the merging processor must determine whether the left endpoint of the other segment is in front of, or behind, the line containing the visible segment; see Figure 3–19. Since both segments are on the same scan line, the requisite computations may be carried out in the plane of constant y. When the left endpoint of the other segment is substituted into the equation for the line containing the visible segment, the sign of the result indicates on which side of the line the point lies. If the left endpoint is behind the visible segment, the other segment does not immediately obscure the visible segment. Conversely, if the point is

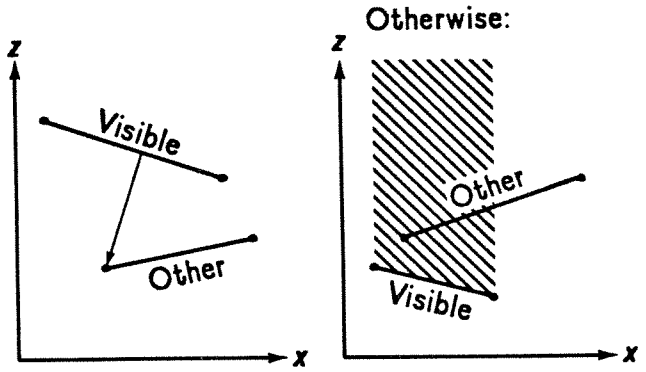


Figure 3–19. Step 3 of the Merging Algorithm: Check for disappearing visible segment. If the left end of the other segment is in front of the visible segment, then the visible segment disappears behind the other segment. Otherwise, the left endpoint lies within the shaded region, below.

in front, the visible segment is actually visible only until the beginning of the other segment. Should this be the case, the merging processor must break the visible segment into the two portions on either side of the other segment's left endpoint. It must then generate the left portion as an output segment, while retaining the right portion. Notice that the visible and other designations are incorrect at this point because of the change in visibility. Therefore, the merging processor must swap them, after which the algorithm can proceed as though it had just classified the two segments.

Even if the left endpoint of the other segment was obscured by the visible segment, it is still possible that the other segment passes through the visible segment at some point along its length. To check for this intersection condition, the merging processor compares the right endpoint of the other segment with two lines, as illustrated in Figure 3–20. The first line contains the visible segment, and the second passes through the right endpoint of the visible segment and the left endpoint of the other segment. If the other segment's right endpoint lies in front of both of these lines, then the two segments do indeed intersect, and the merging processor must compute the point of intersection. It must then divide both segments into the portions on either side of the intersection point. The left part of the visible segment is sent to the output stream, and the right part becomes the new visible segment. The other segment is replaced by its right portion after the left has been discarded. Once again, this manipulation leaves the two segments with incorrect designations, and the merging processor must swap them before proceeding with the two newly classified segments.

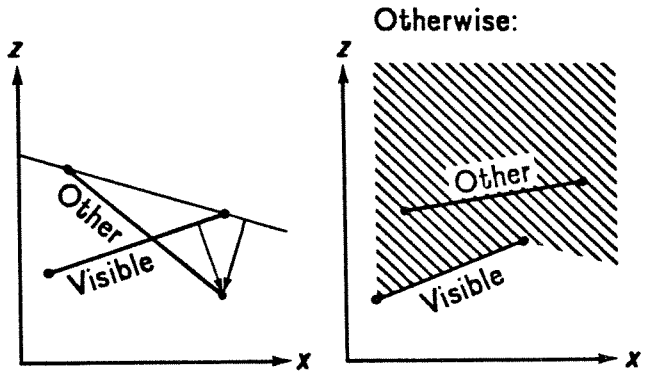


Figure 3–20. Step 4 of the Merging Algorithm: Check for intersecting segments. If the right end of the other segment is in front of both the line containing the visible segment and the line through the left end of the other segment and the right end of the visible segment, then the two segments intersect. Otherwise, the two segments appear as shown in the diagram to the right.

If the two current segments do not intersect, it is still possible that the other segment might extend out from behind the visible segment. To detect this condition, the merging processor compares the right endpoints of the two segments. The test is illustrated in Figure 3–21. If the right endpoint of the visible segment occurs before the right endpoint of the other segment, then the visible segment ends first. The merging processor must clip the other segment to the portion following the visible segment, copy the visible segment to the output stream, and fetch a replacement visible segment from whichever input stream it originally came. Finally, if the test failed, the other segment ends first and is therefore completely hidden by the visible segment. The merging processor discards it and fetches a replacement from the proper input stream. In either of the two alternatives, the algorithm resumes at the classification step.

To summarize, the merging processors repetitively execute a six step algorithm. The first step classifies the two segments obtained from the two input streams as either the visible segment or the other segment. The second step determines whether the two segments overlap at all. If not, the visible segment may be output intact. The third step in the algorithm checks for the case where the visible segment disappears behind the edge of the other segment. If this happens, the unobscured portion of the visible segment is output. Step four tests for segment intersection, and again, if an intersection occurs, only the unobscured portion of the visible segment is output. The fifth step in the merging algorithm tests for the case where the other segment extends from behind the visible segment. If it does, the other segment is clipped

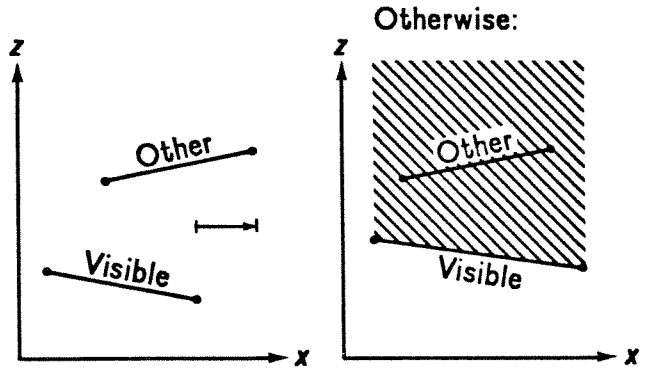


Figure 3–21. Step 5 of the Merging Algorithm: Check for protruding other segment. If the right end of the other segment is to the right of the right end of the visible segment, then the other segment extends out from behind the visible segment. Otherwise, the other segment is completely hidden behind the visible one.

to the unobscured portion, and the visible segment is copied to the output stream without modification. Finally, if the algorithm reaches step six, the other segment is completely hidden behind the visible segment and should be replaced with the next segment.

It turns out that the merging processor can utilize quite a bit of the concurrency available in the algorithm that has just been presented. The basic idea is to evaluate all of the tests and to compute many of the other results simultaneously, and afterward to use a sort of priority encoder for selecting the proper result. Notice that not all of the computations will produce meaningful values, but the final selection will ignore nonsense results. Although this may at first seem like a rather simplistic approach, it is feasible for two reasons. First, all of the arithmetic is performed with fairly low precision integers, so that the required devices are relatively inexpensive ones. Second, it takes only a few arithmetic operations to perform each test, making it acceptable to dedicate hardware for each of them.

The first test performed by the merging processor determines which of the two current segments is the visible one and which is the other segment. Recall that this test compares two numbers formed by concatenating coordinate values from the two current segments. While the test could conceivably be performed in parallel with the others, it is fast enough that doing so would result in only minimal performance gains. Moreover, since all subsequent tests depend upon knowing which segment is the visible one, without this information the tests would have to be twice as complicated in order

to cover both possibilities. The initial test for classifying the two current segments will therefore be performed in advance of the other tests.

The remaining tests fall into two categories. The first type, used to detect non-overlapping segments and a completely hidden other segment, are integer comparisons analogous to the test for initially classifying the segments. That is, coordinate values from each segment are concatenated to form two numbers, which are then compared. The tests are

$$\begin{aligned} \text{conc}(-y_{VR}, x_{VR}) &< \text{conc}(-y_{OL}, x_{OL}) \\ x_{OR} &< x_{VR}. \end{aligned}$$

The subscripts V and O refer to the visible and other segments, while L and R designate the left and right segment endpoints. These two tests, of course, require only very minimal hardware support.

The second type of test determines the side of a line on which a point occurs. In order to perform this test, the merging processor must substitute the point into the equation of the line, after first computing the coefficients of the equation. The sign of the result indicates where the point lies. The three tests of this nature are

$$\begin{aligned} 0 < \;& a_V x_{OL} + b_V z_{OL} + c_V \\ 0 < \;& a_V x_{OR} + b_V z_{OR} + c_V \\ 0 < \;& a_{OV} x_{OR} + b_{OV} z_{OR} + c_{OV}, \end{aligned}$$

where

$$\begin{aligned} a_V &= z_{VR} - z_{VL} & a_{OV} &= z_{VR} - z_{OL} \\ b_V &= x_{VL} - x_{VR} & b_{OV} &= x_{OL} - x_{VR} \\ c_V &= x_{VR} z_{VL} - x_{VL} z_{VR} & c_{OV} &= x_{VR} z_{OL} - x_{OL} z_{VR}. \end{aligned}$$

These tests involve ten multiplications and twelve additions.

The actual results computed by the merging processor take the form of broken segments. These occur either when the visible segment passes behind the other segment, when the other segment passes out from behind the visible segment, or when the two segments intersect. If segments are considered to be parametrically defined line segments, the place at which a segment should be broken is represented by the parameter values at the right end of the left portion and at the left end of the right portion. In the cases where segments are split because they overlap, the parameter values are

$$t_{V-} = \frac{x_{OL} - \epsilon - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{V+} = \frac{x_{OL} - x_{VL}}{x_{VR} - x_{VL}}$$

$$t_{O+} = \frac{x_{VR} + \epsilon - x_{OL}}{x_{OR} - x_{OL}}.$$

In the expressions above, ϵ is a small number based on the horizontal resolution of the computations. The other segment requires only a single parameter because when it emerges from the visible segment, the left portion is always hidden and is therefore always discarded. The computation may be implemented with three dividers and a handful of adders.

In comparison to the situation where segments overlap, the calculation required

when the segments intersect might seem to be a bit more complicated. The expression for the value of x at the point of intersection is

$$x = \frac{b_V c_O - b_O c_V}{a_V b_O - a_O b_V},$$

where a_V, b_V, and c_V are defined as above, and

$$\begin{aligned} a_O &= z_{OR} - z_{OL} \\ b_O &= x_{OR} - x_{OL} \\ c_O &= x_{OR} z_{OL} - x_{OL} z_{OR}. \end{aligned}$$

Notice, however, that most of the values used here are line equation coefficients that have already been computed, as described earlier. Computation of the additional coefficients requires only two more multipliers and a few more adders. Furthermore, once the coefficients are available, the other multiplications and the division required to find the value of x can be implemented by reusing some of the devices that are already in place. This resource sharing is possible because after computing the line equation coefficients, the devices would otherwise be idle. The details of this approach will be covered shortly.

After the x value at the intersection point has been found, the actual parameter values that will be used to break the segments must be computed. The parameters are given by

$$t_{V-} = \frac{x - \epsilon - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{V+} = \frac{x - x_{VL}}{x_{VR} - x_{VL}}$$
$$t_{O+} = \frac{x + \epsilon - x_{OL}}{x_{OR} - x_{OL}}.$$

Again, the hardware for evaluating these expressions is already in place. The same devices that were used to compute the parameter values where segments overlap may now be used to find the parameter values at the intersection point, because they will have completed their original task by the time the intersection has been found. Thus, the intersection computation may be accomplished with only a fairly modest increase in hardware.

The only task remaining for the merging processor is to break the segments as indicated by the tests. Notice that except in the case of intersecting segments, by the time the line segment parameters defining the break are ready, the test results will also be complete. At this point, performance may be enhanced by providing a parallel set of segment splitting hardware for each of the various ways in which a segment may be split. In most cases, there will be three pieces of segment that are of interest: the left part of the visible segment is generally copied into the output stream, while the right parts of the visible and other segments are retained. Because it is always hidden, the left portion of the other segment is always thrown away, and there is no need to compute it. When the parameter values are available, it is not difficult to break a segment. The new end of the other segment, for example, is given by

$$
\begin{aligned}
x_{O+} &= x_{VR} + \epsilon \\
z_{O+} &= (z_{OR} - z_{OL})t_{O+} + z_{OL} \\
r_{O+} &= (r_{OR} - r_{OL})t_{O+} + r_{OL} \\
g_{O+} &= (g_{OR} - g_{OL})t_{O+} + g_{OL} \\
b_{O+} &= (b_{OR} - b_{OL})t_{O+} + b_{OL}.
\end{aligned}
$$

Similar expressions give the new ends of the visible segment parts. To compute all three new segments requires twelve multiplications and twenty-four additions. Here again, it is possible to reuse existing devices to implement these computations.

The action of the merging processor may be partitioned into five stages, each of which uses results computed in prior stages. The computations needed at each stage are listed in Figure 3–22. Successive stages reuse the same set of multipliers, dividers, and other arithmetic devices. In the first stage, the processor computes all of the test results necessary for distinguishing the various possible segment configurations. It also computes some line equation coefficients that will be useful if the segments intersect. Finally, it determines the parameter values that will be used to break the segments in case they overlap. These computations require twelve multiplications and three divisions. At the end of the first stage, the merging processor will have identified the segment configuration. If the visible segment should be copied whole to the output stream, the processor may do so at this point. The second, third, and fourth stages are executed only if has been determined that the two current segments intersect. The second and third stages compute the value of x at the point of intersection, and the fourth stage determines the parameter values required to break the segments around this point. These intersection stages reuse four of the multipliers and all three of the dividers that were required in the first stage. The fifth stage, which actually breaks the segments, is executed only if the segments overlap or intersect. It takes twelve multiplications to break the segments, and these too are implemented with the same devices used in the first stage.

If the staged operation of the merging processor were implemented with conventional parallel adders, multipliers, and dividers, the circuitry necessary for transporting values to the proper devices would be a switching nightmare. For example, the multipliers must be connected in several configurations, requiring the use of a parallel multiplexor for each multiplier. The wiring associated with the multiplexors can easily occupy an immense region on an integrated circuit, even without considering the size of the multiplexors themselves. On the other hand, this is exactly the sort of application at which bit serial communication and bit serial arithmetic excel. When additions and multiplications are combined, the speed of a serial implementation is comparable with that of a parallel one. Moreover, the area consumed by serial adders and multiplexors is substantially smaller than that required by their fully parallel counterparts. Finally, the basic wiring cost for transporting a value from one part of a circuit to another is reduced because a bit serial communication scheme requires only a single wire.

Stage 1. Identify the segment configuration.

$$\begin{aligned}
a_V &= z_{VR} - z_{VL} & b_V &= x_{VL} - x_{VR} & c_V &= x_{VR}z_{VL} - x_{VL}z_{VR} \\
a_{OV} &= z_{VR} - z_{OL} & b_{OV} &= x_{OL} - x_{VR} & c_{OV} &= x_{VR}z_{OL} - x_{OL}z_{VR} \\
a_O &= z_{OR} - z_{OL} & b_O &= x_{OR} - x_{OL} & c_O &= x_{OR}z_{OL} - x_{OL}z_{OR}
\end{aligned}$$

$$\begin{aligned}
\text{test}_1 &= \text{conc}(-y_{OL}, x_{OL}) - \text{conc}(-y_{VR}, x_{VR}) \\
\text{test}_2 &= a_V x_{OL} + b_V z_{OL} + c_V \\
\text{test}_3 &= a_V x_{OR} + b_V z_{OR} + c_V \\
\text{test}_4 &= a_{OV} x_{OR} + b_{OV} z_{OR} + c_{OV} \\
\text{test}_5 &= x_{VR} - x_{OR}
\end{aligned}$$

$$t_{V-} = \frac{x_{OL} - \epsilon - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{V+} = \frac{x_{OL} - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{O+} = \frac{x_{VR} + \epsilon - x_{OL}}{x_{OR} - x_{OL}}$$

Stage 2. Compute values for the intersection computation.

$$\begin{aligned}
i_1 &= b_V c_O & i_2 &= b_O c_V \\
i_3 &= a_V b_O & i_4 &= a_O b_V
\end{aligned}$$

Stage 3. Find the segment intersection point.

$$x = \frac{i_1 - i_2}{i_3 - i_4}$$

Stage 4. Find the parameter values where the segments intersect.

$$t_{V-} = \frac{x - \epsilon - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{V+} = \frac{x - x_{VL}}{x_{VR} - x_{VL}} \qquad t_{O+} = \frac{x + \epsilon - x_{OL}}{x_{OR} - x_{OL}}$$

Stage 5. Break the segments.

$$\begin{aligned}
x_{V-} &= x_{OL} - \epsilon & x_{V+} &= x_{OL} \\
z_{V-} &= (z_{VR} - z_{VL})t_{V-} + z_{VL} & z_{V+} &= (z_{VR} - z_{VL})t_{V+} + z_{VL} \\
r_{V-} &= (r_{VR} - r_{VL})t_{V-} + r_{VL} & r_{V+} &= (r_{VR} - r_{VL})t_{V+} + r_{VL} \\
g_{V-} &= (g_{VR} - g_{VL})t_{V-} + g_{VL} & g_{V+} &= (g_{VR} - g_{VL})t_{V+} + g_{VL} \\
b_{V-} &= (b_{VR} - b_{VL})t_{V-} + b_{VL} & b_{V+} &= (b_{VR} - b_{VL})t_{V+} + b_{VL}
\end{aligned}$$

$$\begin{aligned}
x_{O+} &= x_{VR} + \epsilon \\
z_{O+} &= (z_{OR} - z_{OL})t_{O+} + z_{OL} \\
r_{O+} &= (r_{OR} - r_{OL})t_{O+} + r_{OL} \\
g_{O+} &= (g_{OR} - g_{OL})t_{O+} + g_{OL} \\
b_{O+} &= (b_{OR} - b_{OL})t_{O+} + b_{OL}
\end{aligned}$$

Figure 3–22. Values computed in the five stages of merging processor operation.

At this point, a few words about bit serial devices may be in order. Generally, in serial systems, the successive bits of a number are transmitted in sequence along a single wire, with the least significant bit transmitted first and the most significant bit last. The successive bit values are usually spaced according to some system clock. Thus, a serial multiplexor needs to switch only a single bit, regardless of how many bits are used to represent a number. A serial adder accepts two such bit streams and produces a third. At each bit time, the adder computes the sum of the two input bits and a stored carry bit. The low order bit of this two bit sum is sent to the output stream. The high order bit is stored in a register, and it will be used as the carry input during the next stage of the addition. For this reason, serial adders are sometimes called carry-save adders.

Serial multipliers are very similar to the shift-and-add multipliers used elsewhere. They can be thought of as consisting of three registers: one for the multiplicand, one for the multiplier, and one for the sum. These registers are, of course, loaded and unloaded serially. As in the parallel case, the least significant bit of the multiplier determines whether the multiplicand should be added to the sum. In the serial device, however, the carry bits do not propagate across the sum, but rather are stored with each bit of the result in the sum register. When the sum is shifted right to produce the successive bits of the product, the carry bits remain in place, and the sum eventually propagates to the correct carry bit. A serial multiplication like this can actually be faster than a parallel shift-and-add multiplication because no carry propagation chain is involved. In contrast, serial division is a bit more difficult because, among other reasons, the natural order for producing a quotient has the most significant bit first. For the current application, therefore, the division devices will be implemented as parallel, shift-and-subtract dividers with serial-to-parallel and parallel-to-serial converters on either end.

The complete merging processor consists of twelve serial multipliers and three dividers, together with a host of serial adders, multiplexors, and registers, all controlled by a relatively simple finite state machine. The state machine accepts test results and configures the multiplexors to insure that the proper values are computed during the five stages of the merging algorithm. Notice that although the multipliers and dividers are shared by successive stages of the merging algorithm, adders can be allocated rather liberally. This apparent waste is feasible because serial adders occupy so much less area than devices like multipliers and dividers, and because reusing an adder would require a multiplexor occupying nearly as much space as another adder.

Lyon has implemented some serial devices as integrated circuits [LYON80], and their properties can be used to estimate the area and speed of a merging processor. His 16-bit serial multiplier occupies an area 1800λ wide by about 300λ high, where lambda (λ) is a unit defined by Mead and Conway as half the minimum line width [MEAD80]. A 12-bit multiplier would probably be about 1400λ in width. Thus, assuming that dividers are about the same size as multipliers, fifteen of these devices stacked one atop another would be about 4500λ high. Tripling the width of a single

multiplier to account for miscellaneous registers, adders, and control logic gives a rough estimate of the processor's overall horizontal dimension. At 4500λ by 4200λ, the merging processor would be a large chip, but still one that is within the range of current technology. Furthermore, the cost of a merging processor should drop rapidly with the progress of fabrication technology.

The amount of time required to process a pair of segments depends, of course, on their configuration: non-overlapping segments use only the first stage of the merging algorithm, whereas intersecting segments need all five. In actual scenes, intersecting surfaces tend to occur rather infrequently, and the merging processor will rarely have to perform more than two stages of the algorithm. The time required for each stage is dominated by the speed of the multiplication and division operations. Lyon reported that his circuits run at 12MHz, but the dividers used in merging processors are limited by their internal carry propagation chain and by the fact that their operands must be shifted in before they can be used. With this as a guide, one might estimate that an actual merging processor would require about 3 microseconds (μsec) for every stage of the merging algorithm. Thus, non-overlapping segments could be processed in 3μsec, overlapping segments would take 6μsec, and intersecting segments would require 15μsec. The smaller figures are comparable to the time it takes to communicate segments between processors and are therefore satisfactory. On the other hand, the intersection time appears to be rather large; but recall that intersecting segments are relatively rare, so that the average processing time will be closer to the smaller figures. Short queues on the inputs and outputs of merging processors can help to smooth over these irregularities. These time estimates are intended primarily to serve as a basis for the ensuing discussions. Their absolute accuracy is of secondary importance because it turns out that the performance of the overall scan line tree need not be limited by the speed of individual merging processors.

3.3.4 The Pixel Conversion Processor

The pixel conversion processor accepts segments from the root merging processor of the scan line tree and breaks them into their constituent pixels. Unlike the other processors, there is only one pixel conversion processor in a scan line tree. The other processors in the tree can perform relatively slowly as individuals because there are so many of them. The pixel conversion processor, on the other hand, must be fast enough to generate pixels at the rate required to supply the electron beam sweeping the television monitor. For this reason, the the pixel conversion processor will use standard high-speed components in order to achieve the desired level of performance.

Recall that the segments received from the root of the scan line tree are described by eleven numbers: a y value identifies the scan line containing the segment, x and z values locate each end of the segment within the scan line, and a pair of vectors specify the colors at the segment ends. The z values are not useful for pixel conversion,

but the x values, besides describing where the segment starts and stops, are needed to interpolate pixel colors between the segment's two endpoints. The interpolation is done incrementally so that the tight loop can be performed using only high-speed additions. Each color component has its own increment, given by

$$r_i = \frac{r_R - r_L}{x_R - x_L} \qquad g_i = \frac{g_R - g_L}{x_R - x_L} \qquad b_i = \frac{b_R - b_L}{x_R - x_L}.$$

The color interpolation loop is

$r := r_L$; $g := g_L$; $b := b_L$;
for x_L **to** x_R **do**
  $r := r + r_i$;
  $g := g + g_i$;
  $b := b + b_i$;

At the end of each iteration, the three color components are sent to the digital-to-analog converters that actually drive the electron guns sweeping the screen.

The hardware for computing the color increments must be fast enough to keep up with the generation of pixels. The first step, finding the reciprocal of $x_R - x_L$, can be done with a high speed table lookup. Since the values of x are represented with a precision of twelve bits, the size of the table is perfectly reasonable. The second step is to multiply the reciprocal by each of the values $r_R - r_L$, $g_R - g_L$, and $b_R - b_L$ to find the three color increments. TRW multipliers are well suited to this application. After the multiplication step, the products can be passed to the incremental color computations.

Aside from the reciprocal lookup table and the three multipliers, the pixel conversion processor consists primarily of registers and adders. Three registers hold the three color components of the current pixel, and three more hold the increment values. Two additional registers hold the current x value and the final x value, while a third contains the current value of y. It might be a good idea to duplicate the color and x registers in order to pipeline the transition between successive segments. Finally, multiplexors and control logic serve to select either the current set of segment color registers or a set of background color registers if no segment is present for the current values of x and y. Thus, the pixel conversion processor consists of perhaps a board's worth of high speed logic.

3.3.5 Analysis

The scan line tree has a major performance flaw related to its structure. Notice that the root of the tree, which is a merging processor, can produce a segment about once every 6μsec. However, with 512 scan lines refreshed at 30 times per second, only about 65μsec are available for each scan line. This means that scan lines can contain

an average of only about ten or eleven segments. In some applications, like video games, this level of performance may be adequate, but many other potential uses of the machine require far more detail in the final displayed image.

The performance shortcoming of the scan line tree is an example of a problem that plagues many highly concurrent systems. The difficulty is that although a machine may have a great deal of computing power locked within its component chips, the time required for transmitting signals between chips, or even to transmit them across a single chip, limits the speed at which the machine can produce its results. The communication problem seems to occur often enough in the design of highly parallel machines that it might well be called the non-von Neumann bottleneck.

Tree structured machines like the one presented here seem especially susceptible to the communications malady. This is because a tree is essentially a hierarchy of bottlenecks. In a binary tree, each node in the tree has two connections below, but only a single connection above. If it passed values directly from its inputs to its output, a node could accept values from each input at only half the rate that it could produce values at its output. In the scan line tree, this means that merging processors, which produce a result every 6μsec or so, can accept a value from each input only once every 12μsec on the average. This analysis assumes, of course, that the number of segments created by splitting is balanced by the number of segments destroyed because they are completely hidden. The communications bandwidth is thus progressively constricted as the computation proceeds from the leaves of the tree to its root. In order to avoid this inherent bottleneck, it is necessary somehow to provide more bandwidth at the root.

One straightforward way to double the bandwidth out of a tree structured machine is to double the hardware by using two machines. In the problem of hidden surface elimination, one of the machines might be computing the left half of the picture while the other machine computed the right. This approach certainly works, although it seems rather extravagant. Notice that each scan conversion processor at a leaf of the tree produces a segment every 6μsec, and since the leaf processors generate only one segment per scan line, there is a substantial performance reserve at that level. Duplicating the tree near its leaves therefore seems rather pointless.

A better approach is to concentrate the duplication at the site of the bottleneck by grafting an extra root onto the tree. This surgery can be accomplished by first removing the root processor and exposing the two subtrees. Splitting processors, to be described shortly, are then attached to each of the exposed stumps. Finally, two new merging processors are connected to the splitting processors. The operation is depicted in Figure 3–23. A splitting processor has a left output and a right output, each of which produce segments for the corresponding half of the screen. The two left outputs form the inputs of the left merging processor, and the right merging processor is connected in a similar manner. Notice that the two input streams of the substitute root are balanced by two output streams. Thus, if everything in the new root can run at full speed, each of the two subtrees can supply a segment every 6μsec. More

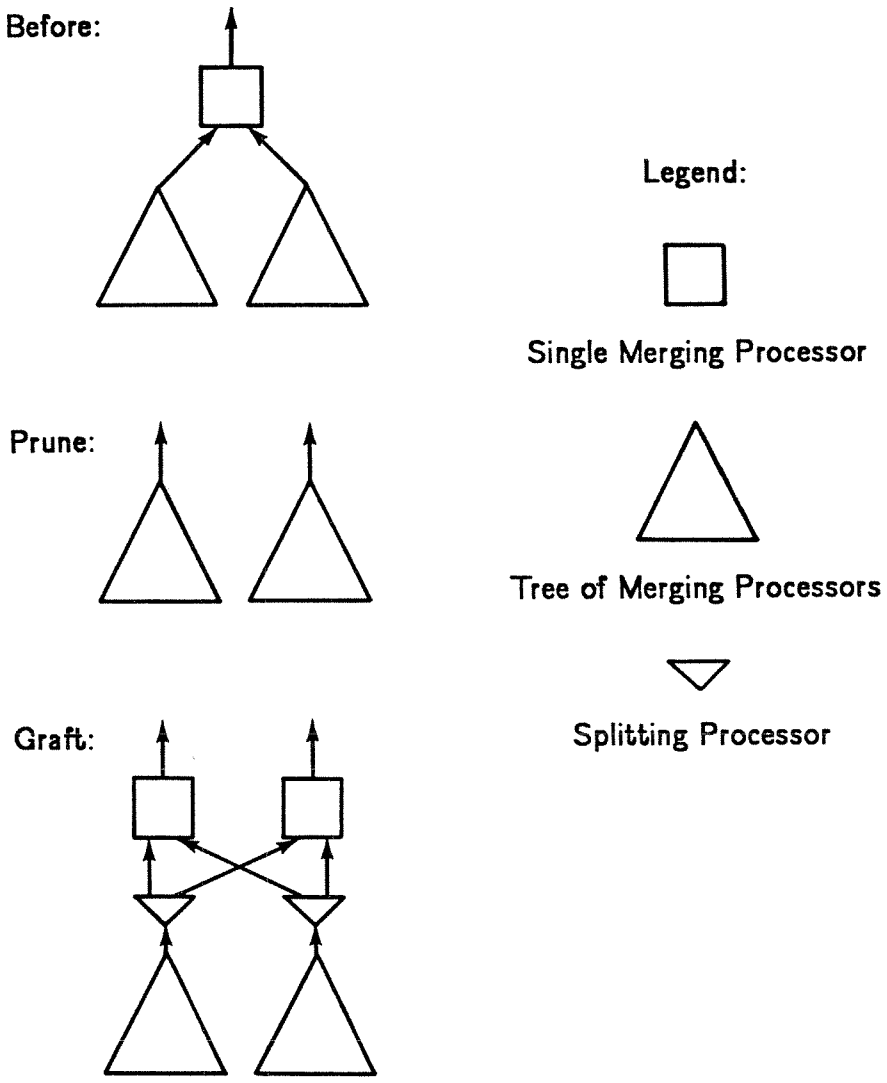


Figure 3–28. Replacing the root of a scan line tree can double its output bandwidth.

importantly, the two output paths can produce two segments every 6μsec, doubling the output bandwidth of the tree with only a modest expenditure in hardware.

The job of the merging processor is to separate the incoming segments into those that appear on the left half of the scan line from those that appear on the right. When it receives a segment from its input, it copies that segment into the proper one of two queues connected to its two outputs. Once every scan line, it may be necessary to break a segment that straddles the boundary between the two halves of the screen. Notice that the queues on the output path are required, because without them the splitting processor could run only at the speed of a single output, rather than at the combined speed of both outputs.

Using splitting processors to increase the capacity of the scan line tree makes

some assumptions about the way in which polygons are distributed throughout the leaves of the tree. Suppose, for example, that every polygon visible in the final image was stored in a leaf of the left subtree. In this case, all of the segments that form those polygons must pass through the root of the left subtree, and inclusion of the splitting processors has merely moved the bottleneck. For the splitting processors to be effective, the visible polygons must be evenly distributed between the subtrees. The easiest way to accomplish this distribution is to assign polygons to processors in some random fashion, so that on the average, subtrees will be evenly loaded with visible polygons. A more predictable alternative uses a subdivision scheme similar to the one described in connection with the ray tracing machines. This approach partitions the modeling space into a collection of subvolumes, and it insures that the polygons enclosed by a particular subvolume are evenly spread among the available subtrees. Such an arrangement should insure that the subtrees are evenly loaded, regardless of viewpoint, and the risk of bottleneck is therefore reduced.

Although doubling the output bandwidth of the scan line tree is certainly a step in the right direction, it still permits a maximum scene complexity of only about twenty segments per scan line. This may not be enough for some applications, especially considering the fact that the screen resolution is 512 pixels per scan line. Fortunately, the trick of splitting the tree at its root can be applied repetitively. For example, by removing two levels of nodes from the root of the tree, the outputs of four merging processors will be exposed. Suppose that two of these outputs are passed through a pair of splitting processors, as described above, and that the other two outputs are passed through another pair. The result would be two outputs for the left half of the picture and two for the right half. If the two left outputs were passed through yet another set of splitting and merging processors, the result would be outputs for the first and second quarters of the image. Similarly, the two right half outputs would be split into the third and fourth quarters. Thus, by replacing two levels of the tree, the output bandwidth maybe increased by a factor of four to about 40 segments per scan line. The bandwidth may be further increased by breaking out as many levels of the tree as are necessary. Figure 3–24 shows the top of a tree with sixteen outputs for a total of 160 segments per scan line. If n levels are replaced, increasing performance by a factor of 2^n, then $n2^n$ splitting processors and $(n-1)2^n+1$ extra merging processors will be required. For $n = 6$, giving about 640 segments per scan line, which is more than can be displayed, this works out to 384 splitting processors and 321 extra merging processors. In a machine with thousands of such processors, this represents a fairly modest increase.

When splitting is extensive, an opportunity for even greater integration presents itself. In a scan line tree with just a single root, all of the segments visible in the final image will be rendered by a single pixel conversion processor. On the other hand, when a tree is split into many roots, it is possible to consider providing a distinct pixel conversion processor for each root. Each of these processors will be required to produce pixels for a vertical strip of the final image. The width of the strip depends, of

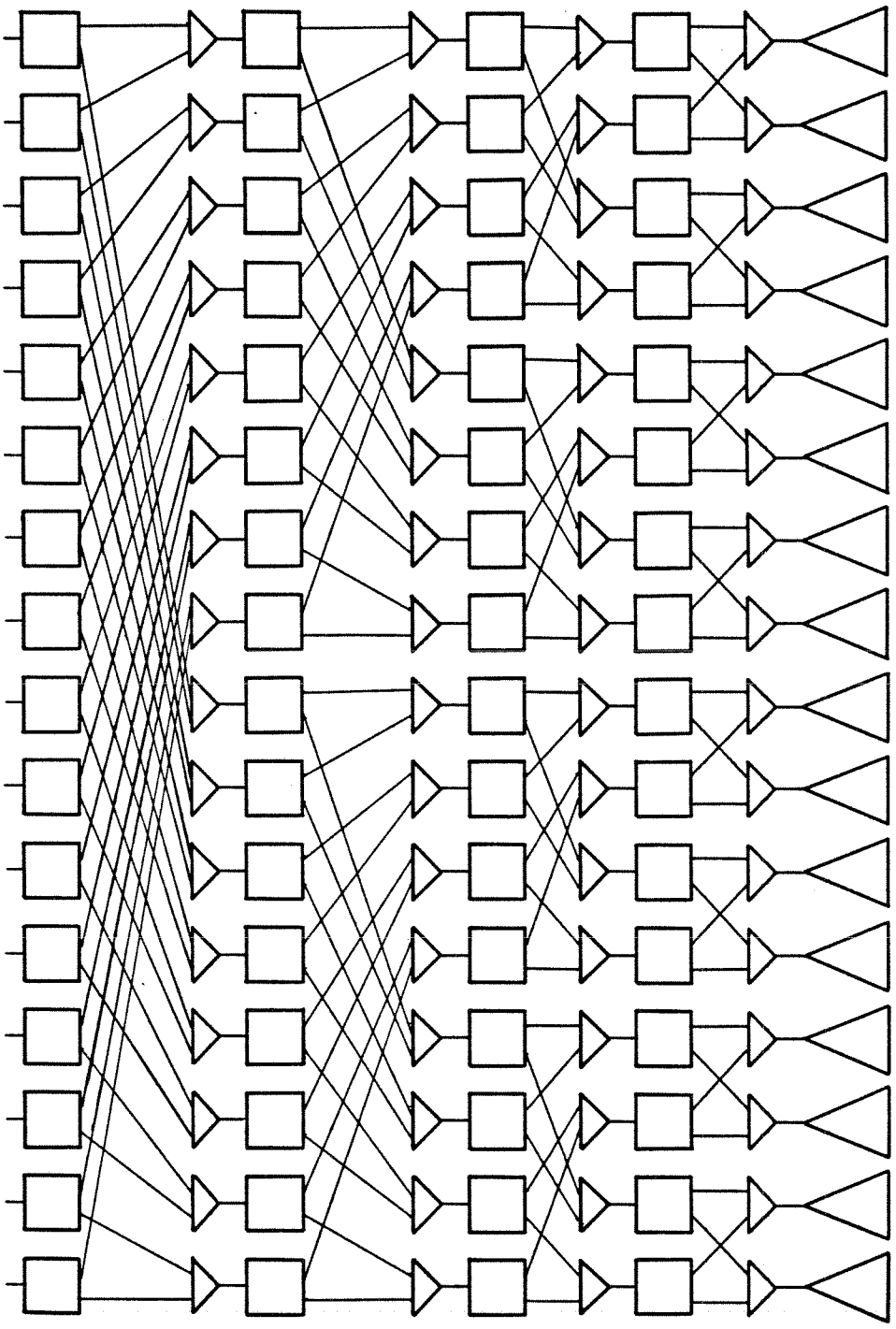


Figure 3–24. A scan line tree where the top four levels have been split to increase the bandwidth by a factor of sixteen.

repeat forever
  comment: Get a segment description from the input stream. ;
  retrieve$(y, x_L, z_L, r_L, g_L, b_L, x_R, z_R, r_R, g_R, b_R)$;
  if $x_R \leq \frac{1}{2}$ **then**
    comment: The segment lies on the left half of the screen. ;
    enqueueLeft$(y, 2x_L, z_L, r_L, g_L, b_L, 2x_R, z_R, r_R, g_R, b_R)$
  else if $x_L > \frac{1}{2}$ **then**
    comment: The segment lies entirely on the right half of the screen. ;
    enqueueRight$(y, 2(x_L - \frac{1}{2}), z_L, r_L, g_L, b_L, 2(x_R - \frac{1}{2}), z_R, r_R, g_R, b_R)$
  else
    comment: The segment crosses from left to right. ;
    $t_- := \dfrac{\frac{1}{2} - x_L}{x_R - x_L}$; $\quad t_+ := \dfrac{\frac{1}{2} + \epsilon - x_L}{x_R - x_L}$;
    $z_- := (z_R - z_L)t_- + z_L$; $\quad z_+ := (z_R - z_L)t_+ + z_L$;
    $r_- := (r_R - r_L)t_- + r_L$; $\quad r_+ := (r_R - r_L)t_+ + r_L$;
    $g_- := (g_R - g_L)t_- + g_L$; $\quad g_+ := (g_R - g_L)t_+ + g_L$;
    $b_- := (b_R - b_L)t_- + b_L$; $\quad b_+ := (b_R - b_L)t_+ + b_L$;
    enqueueLeft$(y, 2x_L, z_L, r_L, g_L, b_L, 1, z_-, r_-, g_-, b_-)$;
    enqueueRight$(y, 2\epsilon, z_+, r_+, g_+, b_+, 2(x_R - \frac{1}{2}), z_R, r_R, g_R, b_R)$;

Figure 3–25. Program describing the behavior of a splitting processor.

course, on the extent to which the root was split. As splitting increases, the number of pixels produced by a single pixel conversion processor during each scan line decreases, while the number of segments that the processors must accept is fixed by the rate at which they can appear at the split roots. It should therefore be quite feasible to design a single chip version of the pixel conversion processor that can perform at the required rate. Of course, some faster hardware must still be included to compose the resulting bands of pixels into the continuous stream needed for display on a television monitor.

The structure of a splitting processor is not very complicated, consisting as it does mainly of memory for implementing its queues. The one test that it must perform identifies segments that should be broken by comparing the x-coordinates of their endpoints with the splitting coordinate. The single computation, which performs the actual break, can be implemented with a divider and a few multipliers, as described for the merging processor. It might seem as though a processor's splitting coordinate would depend on its position in the tree, but this is not the case. A processor can always split segments about the center of the screen, provided that it also transforms the output segments so that they are represented in full screen coordinates. This transformation may be accomplished with single bit shifts.

The size of the queues in splitting processors is another design parameter that must be settled. Basically, the queue must be large enough to hold about one scan line's worth of segments as received from the merging processor at its input. To see

this, notice that the queue must clearly be able to hold all of the segments destined for the left output, because until all such segments are accepted from the input, none of the segments for the right output will be accessible. Similarly, by symmetry, the queue must be able to hold all of the segments in the right portion of the image, because the left segments for the next scan line cannot be accepted until the right segments on the current scan line have passed through the input. Thus, the splitting processor must be able to buffer the segments for one full scan line. Since merging processors can produce a segment once every 6μsec or so, and since the scan line time is 65μsec, a scan line might contain about eleven segments. Of course, a single scan line in the image could conceivably be more complex than this, but if it were, it could not be displayed in real time. The situation can be remedied by splitting the tree at a deeper level to increase the overall bandwidth and ease the burden on individual nodes. Another possibility is to buffer more than a single scan line and to hope that the scene complexity averaged over several lines would be more acceptable. The pixel conversion processor would be required to smooth out irregularities in complexity at some later stage in the picture generation process. This approach doesn't work, however, because of the scan line coherence exhibited by most pictures. That is, consecutive scan lines don't differ by very much, and there is a good chance that if one particular scan line is too complicated, the next one will also be too complicated.

A remarkable property of a splitting processor is that the amount of memory required for buffering segments does not depend upon the position of the processor within the tree. It is also independent of the number of other splitting processors that are in the same tree. Instead, the extra buffering requirement of a deeply split tree is met by an increased supply of splitting processors. The storage requirement depends only on the ratio of the scan line time to the segment communication time. This fact means that a single splitting processor design could be used in a variety of tree configurations. Notice also that since eleven 12-bit integers represent a segment, it takes only about 1500 bits to represent the eleven segments that can occur on a scan line. Considering the simplicity of the remainder of the splitting processor, it should not be at all difficult to fabricate the complete processor, including buffer memory, on a single integrated circuit.

Another difficulty with the scan line tree as presented here is the fact that not all of the processors in the tree may be fully utilized. Recall that after the transformation and clipping processors at the leaves of the tree have finished with their respective polygons, each processor may or may not produce a polygon as a result. When a polygon is outside of the viewing volume, it will be clipped away to nothing. Having received no input, the scan conversion processors for such invisible polygons must sit idle, and the merging processor in at least one level of the tree will not have any segments to merge.

A similar problem occurs in connection with back facing polygons, which form part of some closed object in the scene and face away from the viewer so that they cannot be seen in the final image. Back facing polygons are processed in the same

manner as other polygons, consuming processing resources even though they are invisible. The transformation and clipping processors could, of course, perform a back facing polygon cull to eliminate such polygons, but doing so would simply idle some scan conversion processors; it would not result in more efficient use of resources.

The annoying property of the scan line tree is, then, that its constituent processors may not always be fully utilized. Thus, a tree of a given size has the processing power to handle more polygons than it will usually receive at its leaves. The reason for this situation is that polygons are assigned to leaves before it has been established that the polygons even lie within the visible volume or face the viewer. The other side of the argument is that since the culling is not done until after polygons have been assigned to leaves of the tree, the set of polygons held in the leaves is constant from frame to frame. Thus, some waste of processing power seems to be the price for eliminating the communication that would be required for assigning polygons to leaves during every frame time. Also, notice that if polygons are culled in advance, some processors may also be idled because the number of polygons in the scene may be reduced to fewer than the number of leaves in the tree.

One possible trade-off that can increase processor utilization in exchange for communication cost is to replace all of the transformation and clipping processors with a single, more conventional transformation and clipping pipeline. Such a device might contain dozens of TRW multipliers with associated logic, and because of its pipelined nature, it would be able to transform and clip a stream of polygons in rapid succession. Invisible polygons would be clipped away, and back facing polygons could be detected and discarded, leaving only those polygons that are at least potentially visible.

Distributing the remaining polygons to the leaves of the tree requires some care to insure that the communication is fast enough. One technique is to shift the polygon descriptions serially from one scan conversion processor to the next. Several of these shift chains would be multiplexed from the output of the transformation and clipping processor in order to improve the overall bandwidth. This is the Gatling gun approach. Once the polygons have been transported to the scan conversion processors, the machine can proceed as usual, except that now more of the individual processors can be utilized.

For some ranges of scene size, the conventional approach to transformation and clipping may be more economical than devoting a processor to each polygon. Clark's Geometry Engine, for example, uses a mere twelve chips to form a pipeline that can transform and clip polygons [CLAR80, CLAR82]. Its performance has been estimated at about 900 polygons every thirtieth of a second. If polygon routing can indeed be performed quickly enough, the Geometry Engine seems suitable for machines processing a moderate number of polygons. The Evans & Sutherland PS300 is a vector drawing device that uses hardware based upon a collection of TRW multipliers to accomplish high-speed transformation and clipping, and the implementation fits on a small number of quite large printed circuit boards. This hardware can handle

between 5,000 and 10,000 polygons in real time, depending upon vector lengths and the amount of clipping that is required [EVAN81]. A similar device would be useful in larger scan line trees.

Another trick can be used when transformations are performed outside the scan line tree. A bit of sorting can increase the number of polygons that may be handled by a tree of a given size. Notice that the scan conversion processors are occupied only for those scan lines overlapped by their corresponding polygons. After scan converting these polygons, the processors are idle for the rest of the frame. Instead, the processors could scan convert a sequence of polygons, so long as each polygon in the sequence was completely above the next. This ordering restriction must be imposed to insure that the output of the scan conversion processor conforms to its ordering constraint. The hardware for routing the polygons might use some kind of a bucket sort to accomplish this ordering.

The phenomenon of aliasing is still another problem that must be considered in connection with the scan line tree. Since the machine operates by sampling the scene at scan line resolution, it is inherently susceptible to the staircasing and disappearing polygon effects that are the primary symptoms of aliasing. Probably the simplest way of dealing with the problem is to compute the image at a higher resolution than the television monitor is capable of displaying, and then to filter the higher resolution image down to the screen resolution. The filtering operation can be performed by taking the weighted average of nearby pixels in the high resolution image to form a single pixel in the low resolution image. This technique is not perfect, of course, because the high resolution image was itself computed by sampling the true image. The effects of aliasing will therefore be visible to a lesser extent in the high resolution image as well, and no amount of filtering can eliminate them. The higher resolution sampling and filtering technique does, however, reduce the visible effects of aliasing to a more acceptable level.

The same basic hardware in the scan line tree can be used to compute a higher resolution image, although more hardware will be required. Increasing the resolution in the x direction, parallel to the scan lines, does not present much of a problem. It may be done by increasing the resolution of the computations within the merging processor. In fact, the twelve bit resolution already used may be sufficient. Increasing the vertical or y resolution, on the other hand, is somewhat more difficult because more scan lines must be computed in the same frame time. This means that the amount of time available to compute each scan line will be reduced. To offset the reduction, it is necessary to increase the depth to which the root of the tree is split. Notice, however, that this change affects only the root of the tree, where the computations bottleneck. The performance at the leaves is still not taxed by the additional modification.

Perhaps the most appealing aspect of the scan line tree is its flexibility. An inventory of just four standard processor types is sufficient to configure machines having a wide range of capacities and performance. Identical clipping and transformation, scan conversion, merging, and splitting processors can be used in differing proportions

to implement all of these machines. For example, the performance requirements of a video game are relatively modest, while cost is an important consideration. In such an application, a tree capable of handling only a hundred or so polygons at low resolution might be sufficient. At this performance level, transformation and clipping could be handled in the conventional manner with Clark's Geometry Engine. Including the two hundred or so chips in the scan line tree, the entire machine might be expected to fit on a couple of boards.

At the other extreme, flight simulators must be able to handle scenes consisting of thousands of polygons. A scan line tree capable of handling 2,500 polygons at high resolution requires fewer than 10,000 chips if all processors are packaged individually. This compares favorably with the Evans & Sutherland CT–5, a state of the art commercial flight simulator. The CT–5 is composed of about 85,000 chips and can handle a scene of similar complexity [HALV82]. Thus, if built today, the scan line tree seems to be a viable alternative to the conventional approach. Moreover, the performance of the scan line tree can be extended by increasing the number of processors, or the number of processors can be reduced in order to lower costs in exchange for performance.

As integrated circuit technology advances, the packaging of processors in the scan line tree can be improved so as to reduce the overall chip count even further. For example, one chip type might contain a scan conversion processor together with a merging processor, since a complete system has the same number of both processor types. Similarly, it might be feasible to package splitting processors with merging processors.

3.3.6 Extensions

The structure and operation of the scan line tree is flexible enough to permit consideration of enhancements to improve its performance or image quality. Recall that the scan line tree uses Gouraud shading, which computes the color of a polygon at its vertices and interpolates these to find the colors of interior points. This algorithm was chosen for its simplicity, but anomalies in the resulting image can occur because the technique has no real geometric basis. A better approach is to use Phong shading, which interpolates the vertex normal vectors rather than their colors. This method requires applying the lighting model at every pixel of the final image. The scan line tree could easily interpolate the normal vectors at the required speed. In fact, if the three components of the color vector that it now processes were simply regarded as the three components of a normal vector, it could perform the required computations with no hardware modifications at all. Scan converting the final segments in real time using Phong shading is, however, another matter. The pixel conversion processor would need extensive modification; but if the necessary changes could be made without excessively complicating the pixel conversion processor, it seems feasible to consider

making Phong shaded images in real time.

Another possible extension to the scan line tree involves the use of texture mapping. This technique maps a two-dimensional function onto the surface of a polygon and uses the function values to determine the color of points within the polygon, to perturb the surface normal vectors before performing a lighting computation, or to otherwise change the appearance of a polygon in a possibly irregular manner. In order to accomplish texture mapping, the scan line tree must maintain uv-coordinate pairs in the same way that it now maintains color vectors. Each vertex of the polygon is initially assigned a corner of the unit square on the uv-plane. As the polygon is broken up in the process of clipping, scan conversion, and segment merging, these coordinates must be interpolated. This additional feature would, of course, require a modest amount of extra hardware within the scan conversion, merging, and splitting processors. When segments are finally presented to the pixel conversion processor, each endpoint will have an associated uv-coordinate. Once again, it is up to the pixel conversion processor to handle them in real time, probably by performing some sort of a table lookup for each pixel on the screen.

As described so far, the number of polygons that the scan line tree can handle is limited by the number of processors that are available. With the addition of a moderate amount of hardware, the system can sacrifice real time performance in favor of rapidly produced images of more complicated scenes. It does this by reusing the existing processors several times in order to simulate a larger tree of processors. The method begins by partitioning the polygons of the scene model into groups. The number of polygons in each group must not exceed the number of processors at the leaves of the tree, but there are no other restrictions on the way polygons are assigned to groups. Once the first group has been loaded into the tree, the polygons are processed normally. Instead of pixel converting the resulting segments, however, the machine must copy them into some external storage device, which can be a semiconductor memory, a bubble memory, or even a disk, depending on storage and performance requirements. After the first group of polygons has been processed, the second may be loaded into the leaves of the tree. This second group is also handled in the usual way, except that the result must be merged with the segments from the first group. Notice that because the stored segments were copied directly from the output of a merging processor, they are in scan line order and do not overlap. They may therefore be used directly as an input of an additional merging processor. The other input of this extra processor comes directly from the root of the scan line tree. If there are more than two groups of polygons, the output of the new merging processor should be saved for comparison with the next group of polygons. This procedure should be repeated until all of the polygons in the scene have been considered. When the scene is complete, the stored segments can be routed to the pixel conversion processor for display. Depending upon the speed of the memory and the complexity of the lighting model being used, the television monitor may be refreshed either by repetitively pixel converting the stored segments or by copying the final image into a conventional frame buffer.

3.4 Hidden Line Elimination

The problem of eliminating hidden surfaces from a shaded image in real time has been solved in many ways, but the problem of quickly eliminating hidden edges from a line drawing has proven resistant to solution. No really workable methods have been proposed. The difficulties are twofold. In the first place, the calculations for the hidden line problem must all be exact to the precision of the numeric representation being used, whereas computations in a hidden surface algorithm need be performed only to the resolution of the display. The limited resolution, in turn, permits the use of simplifying approximations. The second major difficulty in a hidden line algorithm is the fact that there is no natural one-dimensional ordering that can be imposed upon the two-dimensional objects. In a hidden surface algorithm, pixels on the screen can be ordered according to the pattern of the raster scan, permitting the use of sorting techniques to improve the performance or to simplify the implementation. No such ordering exists in the unquantized hidden line problem, making the use of sorting more difficult. Sutherland has devised an algorithm that sorts polygon edges on the basis of their relation to other edges in the scene, but its recursive nature limits its suitability for a hardware implementation [SUTH75]. As integrated circuit technology advances, it is becoming practical to consider alternatives that rely on simple, brute force computations. The application of massive parallelism can serve as a substitute for sorting, making hidden line elimination machines almost feasible.

One straightforward approach to designing a real time hidden line elimination machine is to devote a processor to each polygon and to connect them as a linear pipeline as in Figure 3–26. After the transformed and clipped polygons have been loaded into the processors, they must be passed through the pipeline once again. During this second pass, each processor accepts a polygon from its input and compares that new polygon with the one it has stored. If the new polygon is at all hidden by the stored polygon, the processor modifies it so that only the visible portions remain. It then passes these visible parts through its output to the next processor in the pipeline, which treats them in a similar manner. Polygons, or polygon fragments, that survive the complete trip through the pipeline are visible and may be displayed.

One of the difficulties with the pipelined approach is that the individual processors would have to be rather sophisticated in order to be capable of comparing arbitrary polygons. If one were willing to disassemble polygons at the beginning of the pipeline and reassemble them at the end, the polygons could be broken into simpler geometric forms like trapezoids. The processors in this modified machine would need to deal only with trapezoids, and not with general polygons. This approach would increase the number of processors required, but each processor would be less complicated.

Another possibility is to maintain the assignment of one polygon per processor, but to pass polygon edges, rather than complete polygons, through the pipeline. This revised machine is diagrammed in Figure 3–27. The processors in such a machine would forward only those parts of edges that are not hidden by their polygon. Com-

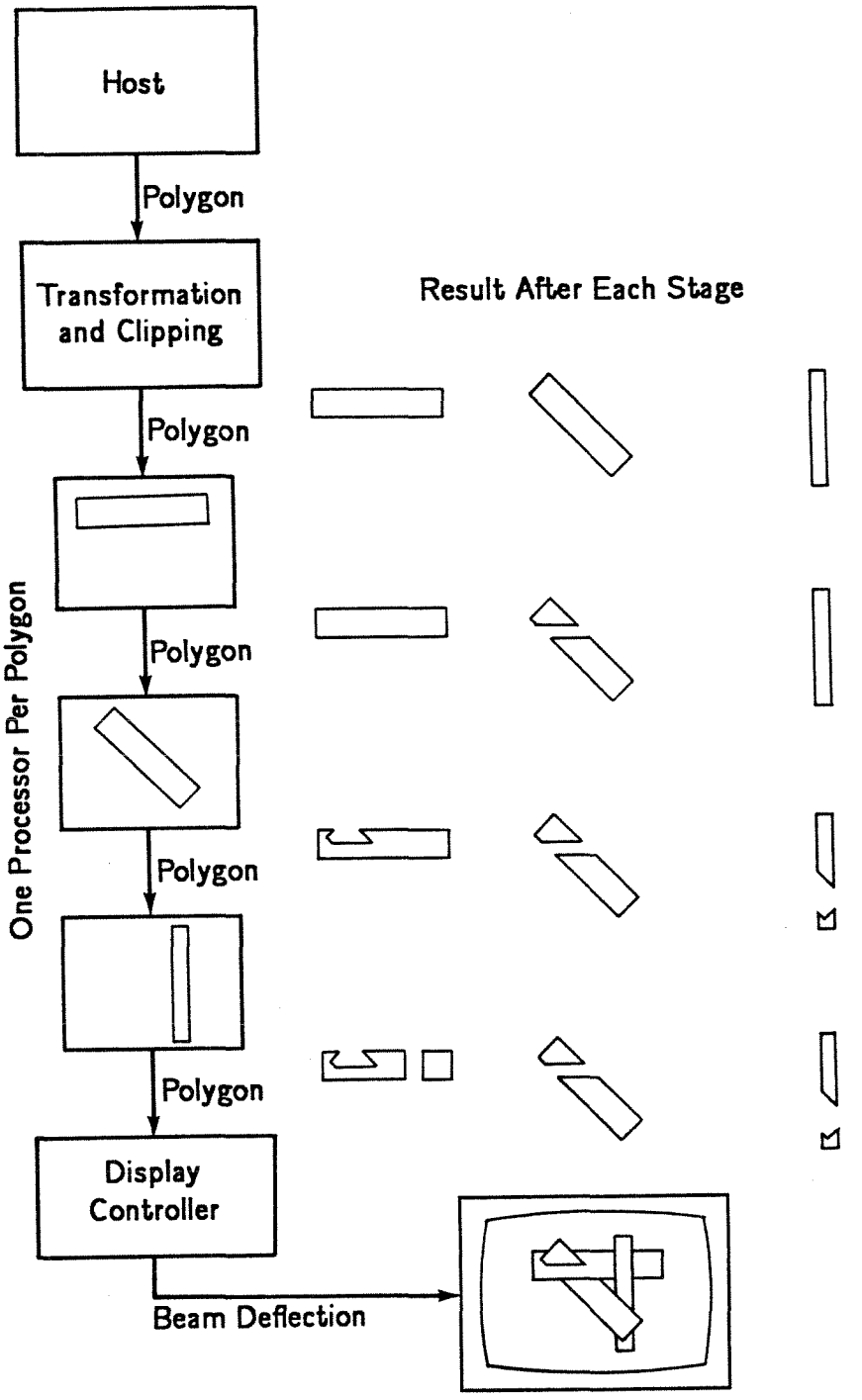


Figure 3–26. A pipeline for hidden surface elimination where processors compare polygons with polygons. The stages of the computation are shown to the right.

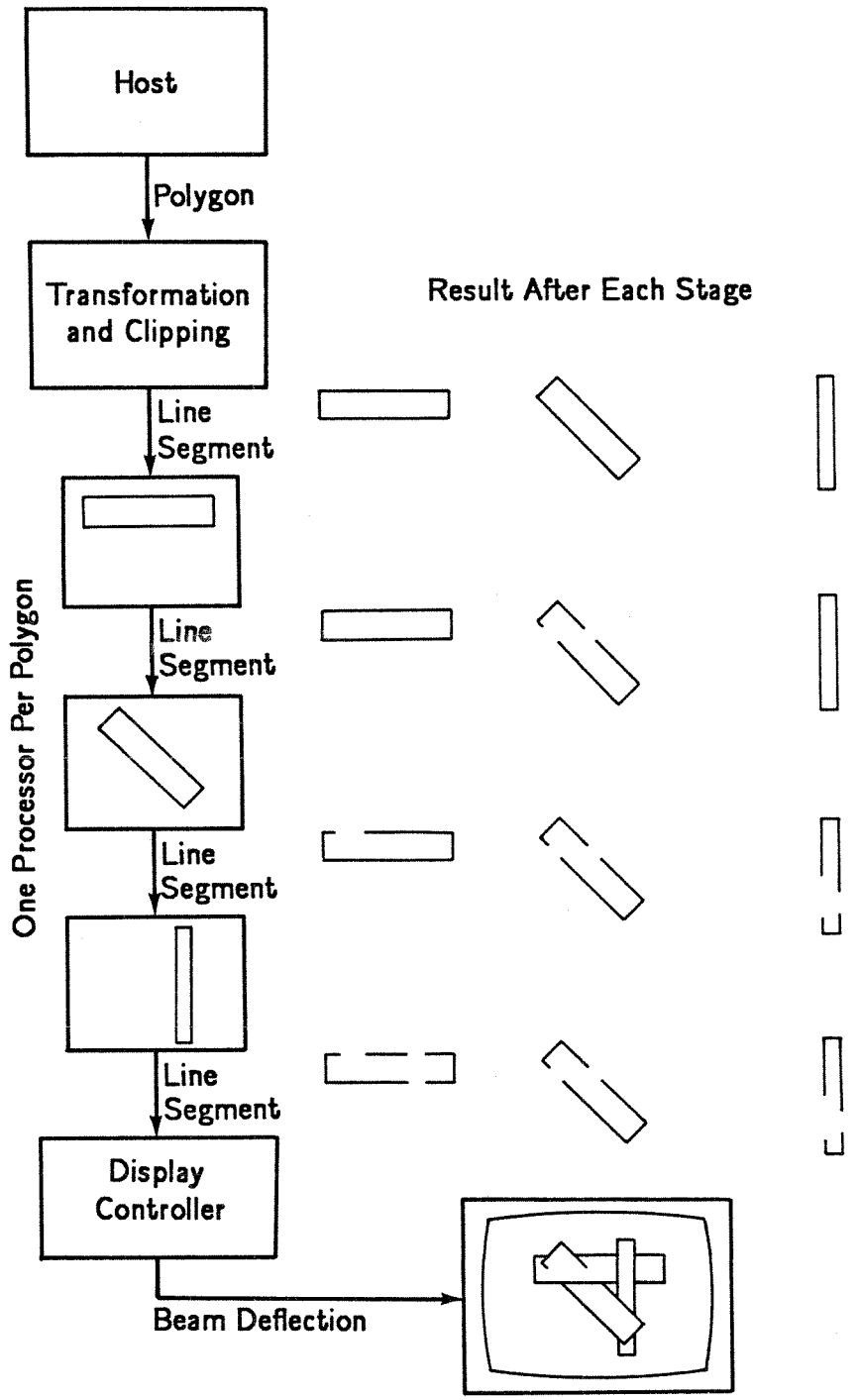


Figure 3–27. A pipeline for hidden surface elimination where the processors are assigned to polygons and edges flow through the pipe. The results after each stage of the computation are shown to the right.

paring a line with a polygon is a substantially simpler task than the comparison of two arbitrary polygons. The price for simplifying this comparison is that implied edges, which occur where polygons intersect, cannot be shown because the planar properties of polygons are, of course, lost when they are treated as separate, unassociated edges. On the other hand, it is quite natural to mix wire frame drawings with solid surfaces using this technique. Moreover, for some applications it should be possible to precompute implied edges and to include them as extra lines in the scene model.

Yet a further simplification can be made by considering infinite extensions of the polygons, since it is easier to deal with complete lines and planes than it is to process line segments and polygons. To see how this might work, refer to Figure 3–28 and notice that the volume of space hidden behind a convex polygon in three dimensions is bounded by planes. One of the boundaries is the plane containing the polygon itself, and there is another boundary for each of the polygon's edges. The edge boundary planes each contain one edge of the polygon, and they extend off to infinity in a direction parallel to the lines of sight. In the screen coordinate system, these edge boundaries are perpendicular to the screen, which is contained in the xy-plane. The region of space hidden by a convex polygon is the semi-infinite convex volume enclosed by these boundary planes.

The task of determining the visibility of a point with respect to these boundary planes is like the clipping problem turned inside out. Recall that in the usual pipelined clipping algorithm, each clipping plane divides space into a region that is definitely invisible and a region in which visibility is indeterminate. If a point is not ruled invisible by any of the clipping plane tests, it is assumed to be visible. The tests are reversed in the anti-clipping algorithm needed for hidden line elimination. A boundary test can determine that a point is visible, but no single test can definitely assert that a point is invisible. Only after being compared with all of the boundaries can a point be considered invisible by default.

To determine which parts of a line segment are visible, it is necessary to determine which part passes through the hidden region of space enclosed by a polygon's implied boundary planes. It is easiest to do this by determining the visibility of the complete line and then later restricting this result to the original line segment. Line segments will be represented as a parametric interpolation of their two endpoints. A complete line is thus represented by recognizing values of the parameter that lie outside the range from zero to one, and the intersection of a line with one of the boundary planes is represented by the value of its line parameter at the point of intersection. In addition, it is necessary to know which end of the line lies on the visible side of the boundary plane. The ends of the line can be identified as corresponding either to very positive or to very negative values of the line parameter, and so it is possible to associate with each intersection a one bit flag that signals whether the positive end of the line is on the visible side of the plane.

The result of testing a line against each of a polygon's boundary planes is a set of parameter values with the corresponding flags that signal which end of the line is

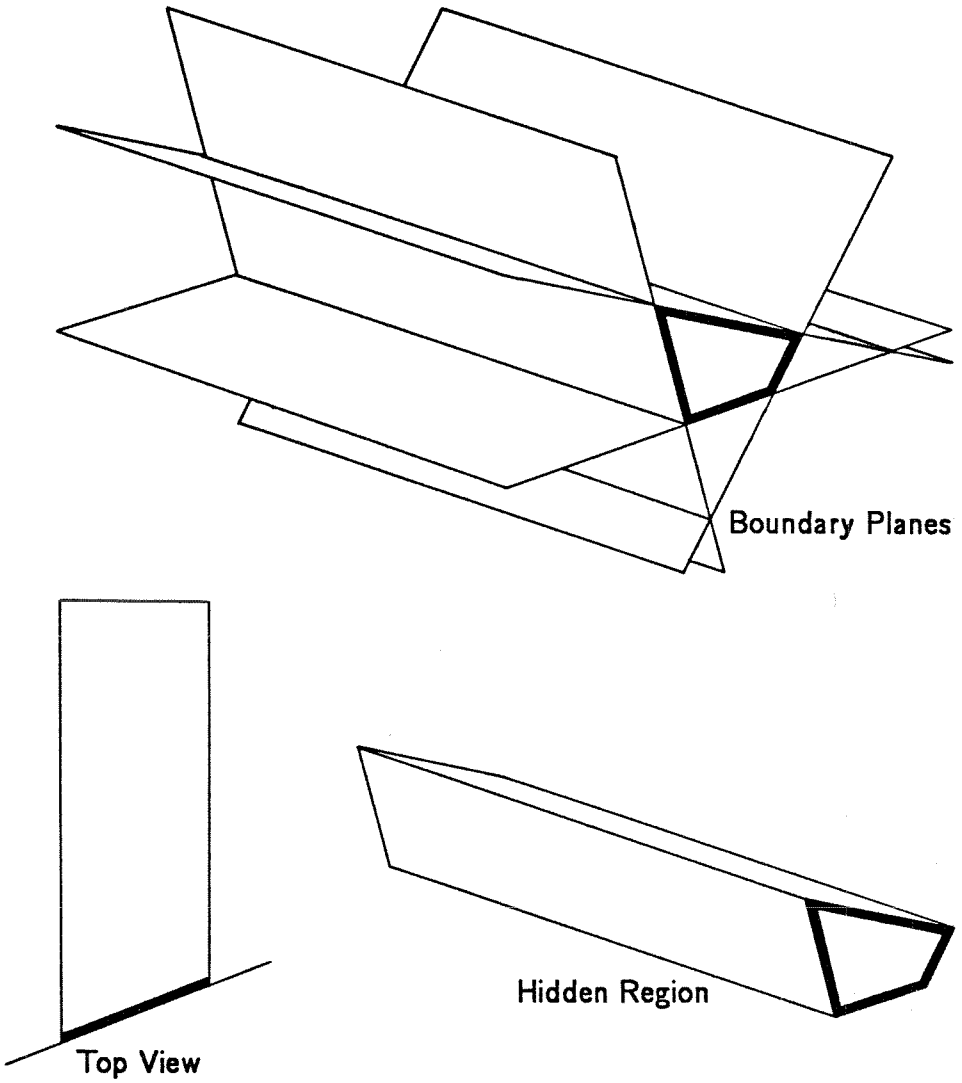


Figure 3–28. The region of space hidden by a polygon is bounded by planes. There is one plane for each edge of the polygon, plus one plane for the polygon itself.

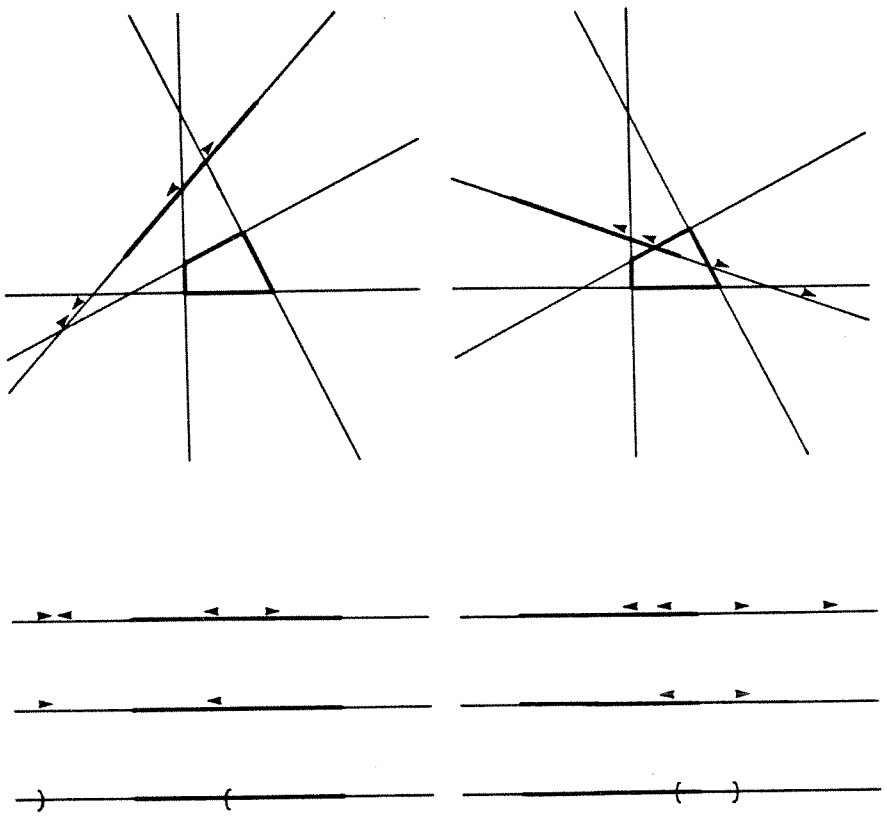


Figure 3–29. Each intersection with a boundary plane asserts the visibility of one end of the line, shown here as arrows directed toward the visible end. After eliminating redundant information, at most two intersections remain for each line.

definitely visible, as shown in Figure 3–29. There is clearly quite a bit of redundant information here. For example, suppose that two different intersection points each have flags specifying that the positive end of the line is on the visible side of a boundary plane. In this case, the positive end of the line is represented as being visible in two different ways: each visible region extends from one of the intersection points to positive infinity, and the region extending from the point with the smaller parameter value includes the region extending from the other point. Because this other point lies within a region already represented as being visible, it contributes no information and may be eliminated. After repeatedly applying this technique, at most two intersection points will remain. One of them describes the earliest point, the point having the smallest parameter value, at which the positive end of the line is known to be visible. The other describes the latest point where the negative end is definitely visible. If the two visible ranges overlap, then the complete line is visible; otherwise, the portion of the line not covered by either visible region is hidden behind the polygon. Of course,

those parts of the line where parameter values fall below zero or rise above one are always invisible, so that some further simplification might be possible.

The complete algorithm for eliminating hidden lines compares each line segment in the scene with every polygon. The polygon comparison, in turn, intersects the line containing the segment with every bounding plane implied by the polygon. The result of this comparison is a collection of intersection points together with their corresponding visibility flags. After eliminating redundancies, at most two intersections remain. One of them describes the extent to which the positive end of the line is visible, and the other does the same for the negative end of the line. Before the line is passed to the next polygon, these two intersections are converted into a form that identifies the single hidden region of the line rather than the two visible regions. As lines are passed from polygon to polygon, these invisible parts are combined even further, as illustrated in Figure 3–30. For example, if two polygons obscure overlapping portions of the line, then the representations of the two portions may be consolidated. Finally, after comparing the line with every polygon in the scene, any parts that were not found to be invisible may be displayed. Notice that if this algorithm were implemented in software, it would be uncomfortably slow because every line segment must be compared against every polygon. The algorithm is simple enough to implement in hardware, however, where massive concurrency can help to compensate for the unsophisticated comparison technique.

A machine architecture to implement this algorithm takes the form of a pipeline much like the one shown earlier in Figure 3–27. Although from the earlier figure it might appear that each polygon will be handled by a single processor, in actuality, responsibility for each polygon is distributed among a pipelined succession of processors in a manner not unlike the standard clipping pipeline. Figure 3–31 illustrates the arrangement. Polygons in the scene are assigned one processor for each of their edge boundary planes, plus an additional processor for the plane of the polygon itself. Descriptions of the planes are shifted into the processors to prepare them for the coming hidden line processing. Once the planes have been loaded, the line segments for display are fed through the pipeline. Each processor must intersect incoming line segments with its own stored boundary plane to determine which end of the line is on the visible side of the boundary. Additionally, the pipeline of processors may be used to eliminate redundant intersections, as discussed above. At the end of the sequence of plane processors assigned to a polygon, there is a cleanup processor which examines the remaining intersection points to determine whether any or all of the segment has been hidden. The segment description, if any, that is passed on to the processors representing the next polygon will reflect these changes. Any segments that survive unscathed their journey through the entire pipeline are not hidden and may be displayed.

Communication between successive processors in the pipeline takes the form of asynchronous message passing. Three kinds of messages are used. The first type, plane messages, carry descriptions of the bounding planes that define the scene. Planes are

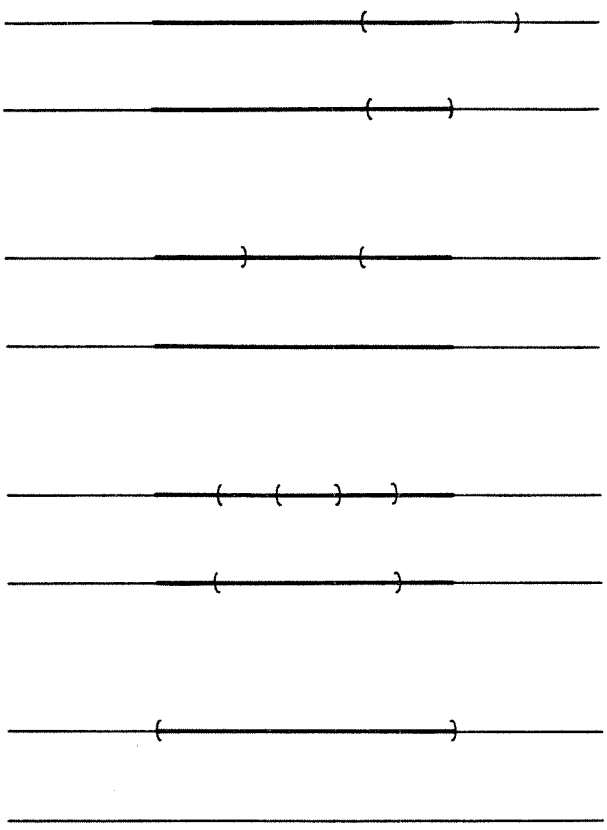


Figure 3–30. Cleanup processors will merge hidden portions of the line that overlap, and they will eliminate completely hidden lines.

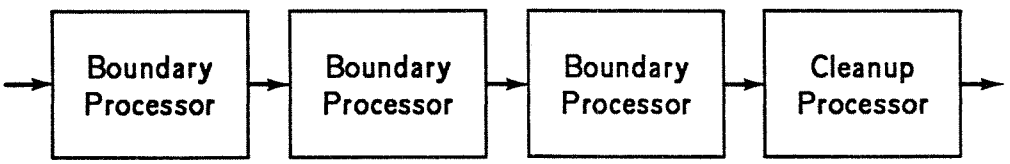


Figure 3–31. Sequence of boundary plane processors and cleanup processors assigned to a single polygon.

represented by the four coefficients of a plane equation, plus some control bits that will be described later. The second type of message describes a line segment as two endpoints. The third message type describes the intersection of a line with a plane, using the parameter value of the intersection point. There are also some control bits specifying, for example, which end of the line is known to be visible. Intersection messages do not contain any information that identifies the line segment to which they apply. Instead, they are assumed to correspond to the most recent line segment message to have passed through the pipe, and so in effect, they follow the line segment description through the pipeline.

The plane messages are used to load the descriptions of the scene's bounding planes into the pipeline. The data path for transmitting plane messages forms what is essentially a shift register running through all of the processors. The communications occur along a set of wires different from those used to carry the other two types of messages, and buffer memory is provided within each processor so that the operations of processing the current scene and loading the next may be overlapped. When it is time to switch between the two, a command from the end of the pipeline causes the contents of the buffer memory to be copied into the processor's active memory. Once this has been done, the planes defining the next scene may be loaded.

When a plane processor receives a segment message, it must first forward the message to the next processor in the pipeline and then compute the intersection of the line with its stored plane. Recall that the processor's boundary plane is represented by the four coefficients of a plane equation:

$$ax + by + cz + d = 0.$$

Also, the line segment carried in the message is represented by its two endpoints, (x_1, y_1, z_1) and (x_2, y_2, z_2). If the line is parameterized by t, an arbitrary point along the line is given by

$$\big((x_2 - x_1)t + x_1, (y_2 - y_1)t + y_1, (z_2 - z_1)t + z_1\big).$$

Substituting this point into the line equation gives

$$\big((ax_2 + by_2 + cz_2) - (ax_1 + by_1 + cz_1)\big)t + (ax_1 + by_1 + cz_1) + d = 0,$$

or

$$(k_2 - k_1)t + k_1 + d = 0,$$

where

$$k_1 = ax_1 + by_1 + cz_1 \quad \text{and} \quad k_2 = ax_2 + by_2 + cz_2.$$

Solving for the parameter t yields

$$t = \frac{k_1 + d}{k_1 - k_2}.$$

Thus, it takes a couple of dot products and a division to find the value of the parameter at the point of intersection. It turns out, though, that the actual value of t is really

needed only to find the three-dimensional point of intersection, and this point, in turn, is not really needed until the segment must be displayed. Therefore, the plane processors can save a division, with its associated time and hardware, simply by representing the parameter t as a rational number. A single division device at the end of the pipeline can replace a divider within each processor. With this modification, the intersection position must be carried as two numbers, t_n and t_d, where $t_n = k_1 + d$ and $t_d = k_1 - k_2$ so that $t = t_n/t_d$. An additional advantage of using a rational representation is that the case where the line and plane are parallel, and therefore where no intersection point exists, will not cause any difficulty. Should this happen, the value of t_d will simply turn out to be zero. This behavior is similar to what would occur if homogeneous coordinates were being used. Also, by not performing the division too early, loss of numeric precision may be reduced.

After finding where the line and plane intersect, the plane processor must still determine which end of the line lies on the visible side of the boundary plane. To see how this may be easily done, reexamine the expression for the distance from the plane to a point on the line:

$$\text{Distance} = (k_2 - k_1)t + (k_1 + d) = -t_d t + t_n.$$

As t increases to positive infinity, the distance will have the same sign as $-t_d$. Thus, because the positive side of the plane is the visible one, if $-t_d > 0$, the positive end of the line is known to be visible. If the line and plane are parallel so that $-t_d = 0$, the distance has the same sign as t_n, which is the distance from the plane to the first endpoint of the line segment. Therefore, the positive end of the line is known to be visible if $-t_d > 0$, or if $-t_d = 0$ and $t_n \geq 0$. Similarly, the negative end of the line is known to be visible if $-t_d < 0$, or if $-t_d = 0$ and $t_n \geq 0$. A one bit visibility flag v will be set to reflect these conditions:

$$v := (-t_d < 0).$$

The intersection is ignored completely if $-t_d = 0$ and $t_n < 0$, because this is the case where the entire line is behind the boundary plane. Barring this case, if the positive end of the line is visible, v will be cleared to zero; otherwise it will be set to one. Once it has computed the intersection and visibility, the plane processor retains them until another segment message is available at its input. Before accepting the new segment, it packages the intersection values and visibility flag as an intersection message and passes them on to the next processor. The reason for not immediately forwarding the intersection will become apparent shortly.

When a plane processor receives an intersection message, it compares the description of the intersection with the values that it had previously computed and stored. The purpose of this action is to reduce the number of intersection messages flowing through the pipeline by eliminating the redundant ones. Suppose that the values received in the intersection message are t'_n, t'_d, and v', while the stored values are t_n, t_d, and v. If $v' \neq v$, the two intersections describe the visibility at different ends of the

line, and the pair cannot be simplified. By convention, intersection messages referring to the positive end of the line precede those for the negative end through the pipeline, so the plane processor must forward the message having a visibility flag of zero and retain the other message.

If the intersection message received by a plane processor describes the same end of the line as the processor's stored message, in other words if $v' = v$, then one of the descriptions is redundant and may be eliminated. In order to identify the redundant description, it is, of course, necessary to compare the parameter values at the point of intersection. The values are represented as rational numbers, so in order to compare them, the processor must perform a cross multiplication while paying careful attention to sign. Notice that the multipliers used earlier to find the intersection are now idle and may be reused for the cross multiplication. The comparison allows the plane processor to identify the redundant intersection, which it then discards. The remaining intersection is retained for comparison with the next incoming intersection message. These operations may be summarized as follows.

$$c{:=}\ (t_n t'_d < t'_n t_d)\ \textbf{xor}\ (t_d < 0)\ \textbf{xor}\ (t'_d < 0);$$
$$\textbf{if}\ (c = v)\ \textbf{then}\ v{:=}\ v';\quad t_n{:=}\ t'_n;\quad t_d{:=}\ t'_d$$

Thus, when the two intersections describe the same end of the line, no output message is generated, and when the next segment message arrives, any remaining intersection description is flushed through the output before the segment is accepted.

A different type of processor, called a cleanup processor, is situated in the pipeline after the sequence of processors assigned to the boundary planes of a single polygon. The purpose of the cleanup processor is to use the intersections found by the plane processors to identify portions of the line segment that are not visible. If it turns out that the entire line segment is hidden, the cleanup processor will repress the segment message to insure that it gets no further through the pipe. If the hidden part of the line does not have parameter values between zero and one, the intersection messages that carry this information are superfluous because they do not apply to the original line segment, and the cleanup processor therefore eliminates them. Finally, if part of the original line segment is obscured, the cleanup processor records this condition by following the segment message with a pair of specially marked intersection messages for the two ends of the hidden part. These new messages are identical to the original intersection messages, except that another control bit h is set to mark them as the ends of a hidden region. Although not mentioned earlier, when a plane processor receives an intersection message with the h bit on, it must immediately forward the message without modification. The final task of the cleanup processor is to merge overlapping hidden parts of the segment so that the number of messages flowing through the pipeline does not become excessive.

The actual operation of the cleanup processor is not unreasonably complicated. When it receives a segment message, it finishes with the previous segment before storing the new message and reinitializing itself. If it receives an intersection message

with $h = 0$, signalling a freshly generated intersection, it retains the intersection and waits for a second one to appear. When another intersection message arrives to describe the visibility of the other end of the line, the cleanup processor must compare them using the same cross multiplication technique found in the plane processors. If the visible regions overlap, the line does not pass behind the polygon, and the intersections are of no consequence. On the other hand, if the visible regions are disjoint, the portion of the line that lies between them is hidden, and the processor must determine whether this hidden region is part of the actual line segment. This will be the case only if one of the ends lies between zero and one. Once again, it is not necessary to perform a division in order to make this determination, rather it is sufficient to compare the numerator and denominator, t_n and t_d, at each intersection. If the hidden region does indeed encompass part of the line segment, the hidden bits h in the two intersection messages must be set.

The next phase of cleanup processor operation detects and merges overlapping hidden designations so as to reduce the message flow through the pipeline. In order to accomplish this merging, the processor attempts to keep the sequence of hidden parts sorted according to their position along the line. When in sorted order, hidden parts that should be merged will be adjacent to each other and may therefore be compared more easily. Unfortunately, the sequence of hidden parts may be arbitrarily long, and to sort the entire sequence at once would require an arbitrary amount of memory for storing the individual items. Notice, however, that sorting is not crucial to the operation of the pipeline; it merely serves to reduce the message traffic. This observation suggests that it is sufficient for a single processor to sort the messages only partially, on the assumption that some processor downstream will be able to improve the ordering even further and that the messages will eventually be sorted perfectly.

The collection of cleanup processors performs a kind of bubble sort. Each processor has enough memory for two hidden parts, which are made up of four intersection messages. If they are in the wrong order, it will swap them before transmitting one of them to its output. Here again, the cross multiplication technique described earlier is used for the comparison. This organization means, of course, that completely invisible segments may not be culled until after they have passed through several cleanup processors, since segment messages must precede their associated intersection messages. For example, if there are three hidden parts for a single segment, when the third part arrives, the first one must be passed on to make room for the third. Before the first hidden part may be transmitted, though, the segment message must be released so that it can precede the subsequent intersection messages. Once sent, the segment message cannot be retrieved, and it may turn out that a completely hidden segment is flowing through the pipe. Fortunately, it doesn't really matter, because even though the segment has slipped through this stage, some later cleanup processor will catch it. Finally, it is necessary to have a sequence of cleanup processors at the end of the pipeline in order to filter out exactly this type of debris.

As just described, the pipeline for eliminating hidden lines consists of a sequence of

plane processors followed by a cleanup processor for each polygon. This organization is inconvenient at best, because it allocates a fixed number of processors to each polygon. If a polygon has fewer boundaries than the number of plane processors, then some of the processors will be wasted. On the other hand, if a polygon were too complex, it would have to be broken down into a number of simpler polygons. Also, with this organization, the cleanup processors must occur more frequently than one might otherwise prefer. A solution to the problem is to move part of the task of the cleanup processor into the plane processors. With the new modification, the plane processors will be responsible for setting the hidden bit h in intersection messages leaving the processor containing the last bounding plane of a particular polygon. In addition, the processor must generate a parity bit p that signals whether an intersection message is the first or second to leave the processor. An extra bit f in the plane description serves to distinguish the final boundary plane for a particular polygon. The necessary modifications to the polygon processor are thus very minor ones, but they permit cleanup processors to be placed arbitrarily within the pipeline. In particular, a cleanup processor may now occur in the midst of a sequence of boundary planes defining a single polygon.

Because it retains most of its former duties, the modifications required of the cleanup processor are also relatively minor. The primary difference is that it must now merely forward intersection messages that do not have h, the hidden bit, set. Also, the processor can no longer assume that hidden parts of the segment are well formed, since the plane processors do not attempt to verify this condition. The cleanup processor must therefore examine incoming hidden parts to determine whether they do indeed describe hidden parts of the line, or whether in fact they declare that the entire line is visible with respect to a single polygon. That is, the cleanup processor must insure that the visible regions associated with each intersection message do not overlap. A related problem is that the intersection messages with the hidden bit h set may now arrive singly, rather than in pairs. The processor can detect this situation by examining the parity bit p and can supply the necessary mate for these unmatched intersection messages. Beyond the two changes just mentioned, however, the cleanup processors behave as described above.

The actions performed by the processors in the pipeline are fairly simple ones. Figure 3–32 shows what happens when a plane processor receives a segment message, and Figure 3–33 does the same for an intersection message. Figure 3–34 shows a program that describes the behavior of a cleanup processor. The important thing to note here is that really very little is happening. Rather, it is the fact that the processors are amassed that leads to the performance of the overall machine.

The algorithm for hidden line elimination used in the pipeline has two unfortunate consequences from the standpoint of scene modeling. In the first place, as mentioned earlier, the machine cannot generate the implied edges that occur when two polygons intersect. This feature was sacrificed in exchange for the simplicity and regularity that make a hardware solution feasible. Perhaps there is some other algorithm that can

to receive segment$(x_1, y_1, z_1, x_2, y_2, z_2)$:
  if u **then transmit** intersection(t_n, t_d, v, f, p);
  transmit segment$(x_1, y_1, z_1, x_2, y_2, z_2)$;
  $k_1 := ax_1 + by_1 + cz_1$;
  $k_2 := ax_2 + by_2 + cz_2$;
  $t_n := k_1 + d$;
  $t_d := k_1 - k_2$;
  $u := (t_d \neq 0$ **or** $t_n > 0)$;
  $v := (t_d > 0)$;
  $p := 0$;

Figure 3–32. Action of a plane processor when it receives a segment message. The used flag u tells whether the rest of the values refer to a valid intersection point.

to receive intersection(t'_n, t'_d, v', h', p') :
  if h' **then**
    comment: Pass through a hidden segment message. ;
    transmit intersection(t'_n, t'_d, v', h', p')
  else if not u **then**
    comment: Not already in use. Capture the intersection. ;
    $t_n := t'_n$; $t_d := t'_d$; $u := 1$; $v := v'$
  else if (not v**) and (not** v'**) and** $\left(\frac{t'_n}{t'_d} < \frac{t_n}{t_d}\right)$ **then**
    comment: The new intersection supersedes the one here already. ;
    $t_n := t'_n$; $t_d := t'_d$
  else if (not v**) and** v' **then**
    comment: Forward the positive intersection and save the negative one. ;
    transmit intersection(t_n, t_d, v, h, p);
    $t_n := t'_n$; $t_d := t'_d$; $v := v'$; $p := 1$
  else if v **and (not** v'**) then**
    comment: Forward the positive intersection. ;
    transmit intersection(t'_n, t'_d, v', h', p);
    $p := 1$
  else if v **and** v' **and** $\left(\frac{t'_n}{t'_d} > \frac{t_n}{t_d}\right)$ **then**
    comment: The new intersection supersedes the one already here. ;
    $t_n := t'_n$; $t_d := t'_d$
  else comment: The new intersection is redundant, ignore it. ;

Figure 3–33. Action of a plane processor when it receives an intersection message.

```
to receive segment(x'_1, y'_1, z'_1, x'_2, y'_2, z'_2) :
  comment: Dispose of prior segment before accepting this one. ;
  flush;  sent:= 0;  suppress:= 0;  p:= 1;
  x_1:= x'_1;  y_1:= y'_1;  z_1:= z'_1;  x_2:= x'_2;  y_2:= y'_2;  z_2:= z'_2;

to receive intersection(t'_n, t'_d, v', h', p') :
  if h' then comment: Insure that nothing is missing. ;
    if p' = p then insert(-∞, 1, 1, 1, 1);
    if p' = 0 and v' = 1 then insert(+∞, 1, 0, 1, 0);  p':= 1;
    insert(t'_n, t'_d, v', h', p')
  else comment: This intersection is not ready for cleanup. ;
    flush;
    if not suppress then transmit intersection(t'_n, t'_d, v', h', p');

to insert(t'_n, t'_d, v', h', p') :
  if n = 4 then makeSpace;
  n:= n + 1;  p:= p';
  int n:= t'_n, t'_d, v', h', p';
  if n = 4 then sortMerge;
to sortMerge :
  if int1 < int3 then swap int1 with int3;
  if int2 < int3 then comment: Merging is possible. ;
    if int4 < int2 then int2:= int4;
    n:= n - 2;

to makeSpace :
  if int1 > 1 then int1:= 1;
  if int2 < 0 then int2:= 0;
  if int1 > int2 then
    if int1 = 1 and int2 = 0 and not sent then suppress:= 1;
    if not sent and not suppress then
      transmit segment(x_1, y_1, z_1, x_2, y_2, z_2);  sent:= 1;
    if not suppress then comment: At least part of the segment is visible.;
      transmit intersection(int1);  transmit intersection(int2);
  int1:= int3;  int2:= int4;  n:= n - 2;

to flush :
  while n > 1 do makeSpace;
  if n = 1 then insert(-∞, 1, 1, 1, 1);  makeSpace;
```

Figure 3–34. Program describing the behavior of a cleanup processor.

achieve a similar goal without having to make this compromise. Another unfortunate property of the algorithm is that the approach of using boundary planes precludes the use of concave polygons in the scene. Probably the least offensive way around this limitation is to break all concave polygons into collections of convex ones. The new polygons may then be surrounded with boundary planes and treated in the usual manner. Notice, however, that if the line segments sent through the pipeline are taken from the original concave polygon, the final scene will not show any artifacts of the splitting operation.

It is interesting to estimate the amount of hardware needed to implement each processor in the pipeline. First of all, consider the requirements for communications between successive chips. Segment messages consist of six numbers that specify the endpoints of a line segment. If 16-bit numbers were used, the total size of the message would be 96 bits. Similarly, an intersection message includes two numbers and three flags for a total of 35 bits. Polygon messages contain four numbers and a flag bit, or 65 bits all together. Recall that polygon messages travel in parallel with the other two kinds of messages, enabling the machine to load the next scene while processing the current one. Once again, it appears that some form of bit serial communication strategy is indicated to insure that the number of wires connected to each processor will remain reasonable. A straightforward way of allocating signal wires is to dedicate one for every value that must be transmitted. Suppose, then, that each communication path consists of six wires shared between the segment and intersection messages, plus five wires for the polygon messages. In addition, to allow for handshake protocols, there will be one request line for each of the three message types and one acknowledge line for each of the two message paths. Doubling the total because each processor has both an input and an output shows that each processor requires 32 wires for communication, not including power and miscellaneous control signals like reset. Since each type of message includes a 16-bit number, at 500 nanoseconds per bit, it would take about 8μsec to transmit a complete message.

Because communications are performed serially, it makes sense to use serial implementations of arithmetic within the processors themselves. Consider first the plane processor, where the complexity of the computations are almost entirely dominated by the multiplications. Serial multipliers are clearly appropriate, but notice also that serial adders and comparators are equally feasible because they can easily keep up with the communication between processors. The only real decision concerns the number of multipliers that should be devoted to each processor. The choice must be made by comparing the multiplication time with the time required to pass a message from one processor to the next. The plane processor needs to perform six multiplications, and if multiplication were twice as fast as communication, for example, there would really be no point in using a full six multipliers, because they would be idle for half of the communication time. Three multipliers could be multiplexed to compute the same values in a similar amount of time. It turns out that this performance estimate is a reasonable one, so suppose that the plane processor does indeed contain three multi-

pliers. Using the size of Lyon's serial multipliers as a guide once again, and doubling the result to account for adders, registers, and control logic, leads to an estimate of about 900λ by 3600λ for the size of a single plane processor. It should be possible to fit several of these polygon processors on a single integrated circuit, even with present fabrication techniques.

The size of individual cleanup processors is less critical than the size of plane processors because they occur less frequently in the pipeline. Here, two multipliers are needed to form the cross product that compares intersection positions. Aside from these multipliers, the primary requirements of the cleanup processors are: first, enough memory to hold the segment and intersections being cleaned; second, signal routing circuitry to move values around; and third, control logic to orchestrate the whole affair. The memory requirements are not outlandish—fewer than three hundred bits—and memory can be packed fairly densely, so that this requirement should not present a problem. Since values are represented bit-serially, the associated multiplexors also consume relatively little area. Finally, the behavior of the processor is not very complex, and a small finite state machine should be able to handle the overall control. Therefore, it seems that a cleanup processor would easily fit on a single integrated circuit. A more attractive packaging scheme, however, places a single cleanup processor on the same chip as several plane processors. The chip would be pipelined internally, and its external connections would be no different from those of a package containing only a single processor. Using this approach, the complete pipeline could be formed from a single chip type. Technology is improving rapidly enough that this packaging scheme should be possible very soon, if it is not already.

The primary performance limitation of the hidden line elimination pipeline is the delay necessary for a segment description to pass all the way through the pipeline. Suppose that the pipeline consists of 10,000 plane processors. If ten processors could be fabricated on a single chip, this pipeline would require only a thousand identical and regularly connected packages. If each polygon in the scene had four sides, 10,000 plane processors would be enough to handle 2,000 polygons. If the message passing time is the performance bottleneck, as indeed it seems to be, then it will take about 80 milliseconds, or about a twelfth of a second, for a segment to pass through the pipeline. This figure merely hovers on the border of real time, but even so, it is respectable. Conversely, the machine should be able to process between 800 and 900 polygons in a thirtieth of a second. Granted, there are times when a message will be delayed, as, for example, when new intersection messages are generated, but at other times, messages will be eliminated either by the plane processors or by the cleanup processors, causing gaps of idle processors. The delays should tend to close these gaps, thereby maintaining processor utilization.

4

Observations and Conclusions

Some time ago, Sutherland, Sproull, and Schumacker scrutinized ten hidden surface algorithms and found that sorting was an important characteristic common to all of them [SUTH74b]. They noted further that each of the algorithms made use of some form of coherence—the property that nearby parts of an image are not very different—to reduce the computation necessary for sorting. It is interesting to observe the interplay of concurrency, sorting, and coherence in the parallel machines that were presented in the prior two chapters.

Of all the machines that were discussed, the ray tracing peripheral constructed of commercial components made the least use of concurrency. Because it exploited only the concurrency available within the intersection computation, its operation was much like that of a program running on a fast, but strictly sequential, conventional computer. The version of the machine described initially made no use of sorting to limit its computations, and consequently it bogged down for large scenes. The machine performed much more efficiently when the modeling space was partitioned into a three-dimensional grid of subvolumes. Although the ray tracing peripheral did not itself perform the sorting required to implement this subdivision, it was able to capitalize on the availability of sorted information. Possibly because it did no sorting, the machine made no explicit attempt to take advantage of any coherence that might be present in the scene. In the discussion of subvolume swapping, however, it became apparent that coherence does play a role in reducing the number of disk accesses required to render a scene, because the rays traced for nearby pixels in the image tend

to pass through pretty much the same sequence of subvolumes.

The organization of the ray tracing pipeline enables it to perform potentially more simultaneous computations than any of the other architectures suggested here. At any instant, every one of its component processors could conceivably be in the midst of an intersection computation. Moreover, because each processor is connected to only two others, and because messages flow unidirectionally through the pipe, the communication needed during the operation of the pipeline is quite moderate. Unfortunately, it is also true that the machine performs many more intersection computations than are actually necessary to render the scene. It cannot make use of a sorted scene model, as can the original ray tracing peripheral, to reduce the number of surfaces that it must consider. In order to reduce communication requirements and simplify the task of individual processors, the design of the ray tracing pipeline has completely forsaken any advantage that it might derive from the use of sorting or image coherence. Not surprisingly, the resulting machine has enough drawbacks to make it an unlikely candidate for actual implementation.

The ray tracing array was an attempt to combine the efficiencies of the original peripheral with the performance of the pipeline. Like the pipeline, the ray tracing array contains many intersection processors, but it differs in the sense that the capability for general purpose computation is duplicated as well. Furthermore, the processor interconnection structure is richer in the array than in the pipeline. Like the original ray tracing peripheral, the array is able to make effective use of a sorted scene model in order to limit the number of intersection computations required to form an image. In addition, the array relies upon the presence of sorted data in another, perhaps subtler, way. Because adjoining subvolumes in the scene are sorted into connected processors in the array, individual rays passing through the scene may be modeled as messages passed from processor to processor. Thus, using sorted image data helps to reduce the need for global communication.

The scan line tree, although quite different from the ray tracing machines in terms of structure and operation, exhibits some intriguing similarities when viewed from afar. Its operation is based on the use of sorted data, but for the most part it does not perform any sorting. Rather, the segments that comprise individual polygons are generated in sorted order at the leaves of the tree, and the rest of the tree functions in a way that maintains this ordering. The more obvious benefit obtained from the use of sorted information is a reduction in the number of comparisons that must be performed. Sorting brings nearby parts of the image into corresponding proximity in the machine, where they may easily be compared. Widely dispersed portions of the image need never be explicitly compared. In addition, because the scene remains sorted at every stage of the tree's operation, nearby parts of the image are always in nearby processors. Thus, the use of sorting also helps to limit the task of communication in the machine. As in the ray tracing machines, coherence does not play a major role in the operation of the scan line tree. Although adjacent pixels on a single scan line may be considered as a group,

there is no attempt to exploit the similarity of successive scan lines.

Sorting is by nature a task requiring intensive communication. In any real machine, distinct objects to be sorted must be represented in distinct regions of space. The job of sorting is to move objects from where they are to where they should be, and it must potentially be possible to move any object to an arbitrary location. Accomplishing such an extensive movement of data in a physical machine requires many wires, quite a bit of time, or both. For example, the splitting processors near the root of the scan line tree, which actually perform a sort, require the richest interconnection scheme in the whole machine.

It is noteworthy that the machines presented here largely avoid having to sort any scene information. Therefore, it should not be too surprising to discover that they make little use of image coherence, since sequential algorithms use coherence primarily to ease the sorting task. Instead, the machines rely on having presorted information to ease communication requirements.

In contrast, the earlier machines described by Fuchs and Parke actually do incorporate sorting as part of their operation. Recall that each uses a depth buffer algorithm to identify the surface visible at each pixel in the image. For example, in the Pixel-planes machine, which is the most parallel of the ones discussed, individual pixel processors examine successive polygons and perform a depth sort by accumulating the depth and color of the surface closest to the viewer. Note that each of the depth buffer machines has a single place through which the entire scene must pass. The resulting bottleneck, which is the performance limiting factor, appears to be related to the sorting. None of these machines attempt to make use of image coherence in order to limit the work of sorting.

It may be possible to define a parallel machine that actually does sort the scene model and yet requires only moderate internal communication. Such a machine would almost certainly need to exploit image coherence in order to simplify the sorting operation. For example, Watkins found that the set of polygons appearing on a scan line doesn't change very much from one scan line to the next, and further that the polygons appear in pretty much the same order. Moreover, if the order does change between scan lines, merely swapping a few pairs of adjacent polygons will almost always correct the situation. To Watkins, this observation suggested that a bubble sort was suitable for ordering the segments found on a scan line, but in the context of parallel machines it might mean that a linear array of segment processors would be useful. Each processor need only look to its neighbor when comparing or swapping segments. A similar form of coherence might be used when vertically sorting polygons for display in successive frames. Of course, the problem of sorting new polygons into the active scan line still remains, but it may turn out that some form of tree structure can be applied.

It appears that sorting, which was found to be the central task of programs for hidden surface elimination, is also important in the parallel machines that perform a similar task. In sequential algorithms, sorting was used to reduce the number

of comparisons necessary for the production of an image. In parallel algorithms, operating on a sorted scene can in addition reduce the amount of communication that takes place between processes. Similarly, it may be possible to exploit image coherence in a parallel algorithm. In a sequential algorithm, using coherence could help to reduce the number of comparisons needed to accomplish a sort. In a parallel machine, on the other hand, coherence might be used to limit the amount of communication required for sorting.

Appendix A

Implementing Arithmetic

Earlier discussions have used the performance of the primitive floating point arithmetic operations as the basis for estimating the performance of various machines. Two kinds of implementations, one using commercially available parts and the other requiring custom integrated circuits, were postulated. This Appendix derives the performance estimates that were used earlier.

A.1 Using Commercial Components

The basic floating point operation is, of course, multiplication. TRW suggests a circuit for implementing a floating point multiply using their 285 nanosecond (ns) 24-bit multiplier and only a handful of additional packages [TRW78]. A multiplier chip processes the mantissa, and the other chips add the exponents. The TRW multiplier even has an on-board shifter for normalizing the product, and it informs the support circuitry when normalization is necessary so that the exponent can be corrected. The multiplier itself is the slowest component of the circuit. Allowing 50ns for the latches needed to pipeline the operation, the floating point multiplication can be done in 335ns. To simplify the following discussions, this figure will be rounded to a third of a microsecond. TRW also suggests a scheme for doubling the speed of the multiplication by using two multipliers. It turns out that this trick requires no extra chips aside from the extra multiplier, and the multiplication rate could easily be boosted to six

per microsecond in this way. In theory, higher rates can be achieved by using still more multipliers, but in practice attempting to do so creates a number of difficult problems. Therefore, in the discussions to follow, the figure of three floating point multiplications per microsecond will serve as a conservative measure of performance.

Floating point addition is something of a nuisance. The difficulty is that before two numbers can be added, one of them must be denormalized in order to align the binary points. This denormalization operation is a barrel shift by an arbitrary number of bit positions, which is difficult to implement both quickly and inexpensively, especially for larger word sizes. The TRW multiplier can serve as a barrel shifter if one of its input operands is set to a power of two. This solution requires just one chip, and it is, of course, comparable in speed to the multiply operation. Another possibility is to use a commercial shifter. One such device, made by AMD, could be used to make a 24-bit, 60ns shifter using about eighteen chips [AMD77]. A third alternative is to fabricate a custom designed integrated circuit.

Suppose that the two numbers to be added are given by

$$n_1 = 2^{c_1} m_1 \text{ and } n_2 = 2^{c_2} m_2,$$

where c_1 and c_2 are the exponent parts or characteristics, and m_1 and m_2 are the fractional parts or mantissas. If $c_1 = c_2$, then n_1 and n_2 may be added by directly forming the sum of m_1 and m_2. On the other hand, if $c_1 \neq c_2$, the denormalization process adjusts the representations of the numbers so that the exponents do match and the addition may be performed. Multiplying the numbers by a judiciously chosen representation of the value one denormalizes them:

$$\begin{aligned} n_1 &= (2^{c_1} m_1)\left(\frac{2^{\max(c_2-c_1,0)}}{2^{\max(c_2-c_1,0)}}\right) = \left(2^{c_1+\max(c_2-c_1,0)}\right)\frac{m_1}{2^{\max(c_2-c_1,0)}} \\ &= \left(2^{\max(c_2,c_1)}\right)\frac{m_1}{2^{\max(c_2-c_1,0)}} \\ n_2 &= (2^{c_2} m_2)\left(\frac{2^{\max(c_1-c_2,0)}}{2^{\max(c_1-c_2,0)}}\right) = \left(2^{c_2+\max(c_1-c_2,0)}\right)\frac{m_2}{2^{\max(c_1-c_2,0)}} \\ &= \left(2^{\max(c_1,c_2)}\right)\frac{m_2}{2^{\max(c_1-c_2,0)}}. \end{aligned}$$

Thus, to denormalize n_1, the mantissa m_1 must be shifted right by $\max(c_2 - c_1, 0)$ bit positions. If shifts are being implemented as multiplications, they can be formed by taking the high-order 24-bits of the 48-bit product of m_1 and $2^{24-\max(c_2-c_1,0)}$. Likewise, n_2 can be denormalized by multiplying m_2 by $2^{24-\max(c_1-c_2,0)}$ and taking the upper half of the result. In either case, the exponent is $\max(c_1, c_2)$.

Once denormalization is complete, the sum of the fractions can be computed in a straightforward manner. In general, the sum will not be normalized, and this adjustment must therefore be made before the addition operation can be regarded as complete. The required shift operation can once again be performed by another multiplier with its associated support circuitry, or by another barrel shifter.

As the above discussion has indicated, implementing floating point addition is quite a bit more complicated than implementing multiplication. It is desirable, however, for the addition operation to proceed at the same rate as multiplication, because unless it does so, the floating point multipliers cannot be fully utilized. If additions were slower than multiplications, the entire pipeline would have to be slowed to the speed of the addition. There is no problem if barrel shifters are used for normalization and denormalization, because they are fast in comparison to the multiplication time. On the other hand, if multipliers were used for shifters, additions would be substantially slower than multiplications. Notice, however, that in a pipelined system it is not the addition time, but rather the addition rate that is important, and this rate can be improved by pipelining the addition operation itself. A three-stage pipe is sufficient, wherein the first stage performs the denormalization, the second stage computes the sum, and the third stage renormalizes it. When organized in this way, the addition operation can proceed at the same rate as multiplication, namely at three operations per microsecond, no matter which implementation strategy is chosen. In fact, it will be convenient to design all of the operations in the pipeline to operate at this rate.

As in the case of addition, implementing a high-speed floating point division operation is not exactly a straightforward task. Other systems have successfully used Newton's method [CRAY80], and a similar approach will be taken here. The basic idea is to compute the reciprocal of the denominator using Newton's method and then to multiply this reciprocal by the numerator to complete the division operation. Recall that Newton's method is a technique for iteratively finding the roots of the equation $f(x) = 0$ [DAHL74]. Given a guess x_n, Newton's method computes a better guess x_{n+1} according to the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

To divide a by b, it is necessary first to compute the reciprocal of b, which is the root of the function $f(x) = 1/x - b$. The corresponding Newton iteration formula is

$$x_{n+1} = (2 - bx_n)x_n.$$

Newton's method converges quadratically, doubling the number of bits of precision with each iteration, so that the number of iterations can be reduced if the initial guess is very good. The easiest way of arriving at the first guess is by table lookup. If the table is large enough to produce a result that is accurate to twelve bits, a single iteration of Newton's method will improve this guess to the full twenty-four bits.

In order to perform a floating point division at the desired rate of three operations per microsecond, it must be pipelined. The pipe is longer than it was for addition, however. The first stage of the pipe uses a table lookup to compute the initial guess for the Newton iteration. Suppose that the value of b has the floating point representation $2^{e_b} m_b$. The high order twelve bits of m_b are used to find a reciprocal $2^{e_r} m_r$, and the approximate reciprocal of b is

$$x_0 = 2^{(c_r - c_b)} m_r.$$

Notice that since b is normalized, the high-order bit of m_b is always set. Therefore, it is the next twelve most significant bits that are actually used in the table lookup.

The second stage of the division pipeline begins to compute the result of the Newton iteration. Since this value is the reciprocal of b, it must be multiplied by a to finish the divide operation. The final quotient is

$$(2 - bx_0)x_0 a.$$

The second stage of the pipe simultaneously computes bx_0 and $x_0 a$. The next stage performs the addition, producing $2 - bx_0$. Finally, the remaining stage does the last multiplication to find the quotient.

Using Newton's method to implement a division operation is accurate, if also rather involved. The accuracy is necessary when computing the value of t at the point where the ray and surface intersect, but other division operations in the intersection computation can be less precise. Recall that the values of u and v are used first to determine whether a ray strikes the interior of a polygon, then to interpolate surface normal vectors, and finally to act as indices into texture maps. If these values were known only to twelve bits, resulting in an error of one part in 2^{12}, the error would not be visible on a display with only 512×512 or $2^9 \times 2^9$ resolvable pixels. In the parallel case of the intersection computation, where one or both of the values of u and v are given by quotients, a less accurate division operation would suffice. For these values, the first stage of the division pipeline can produce a reasonable guess at the reciprocal of the denominator by direct table lookup, requiring just a single pipeline stage to do so. A second stage multiplies by the numerator to complete the division in two stages with a result that is accurate enough for computing u and v.

The floating point square root operation, used to compute the values of u and v, could also be implemented using Newton's method, but again, the full accuracy is not necessary. Instead, a table lookup technique can perform a floating point square root operation with a result accurate to twelve bits. Suppose that the number is represented by $2^c m$. Its square root is

$$(2^c m)^{\frac{1}{2}} = (2^c)^{\frac{1}{2}} (m)^{\frac{1}{2}} = (2^{c_c} m_c)(2^{c_m} m_m).$$

Finding the square root can thus be reduced to a pair of table lookups followed by a multiplication. The first lookup uses the exponent as an index, while the second uses the twelve high-order bits of the mantissa. The results of these lookups must then be multiplied to obtain the square root. Notice that this operation must be implemented as two pipeline stages in order to maintain the desired throughput rate. The first stage performs the table lookup, and the second performs the multiplication.

A.2 Using Custom Components

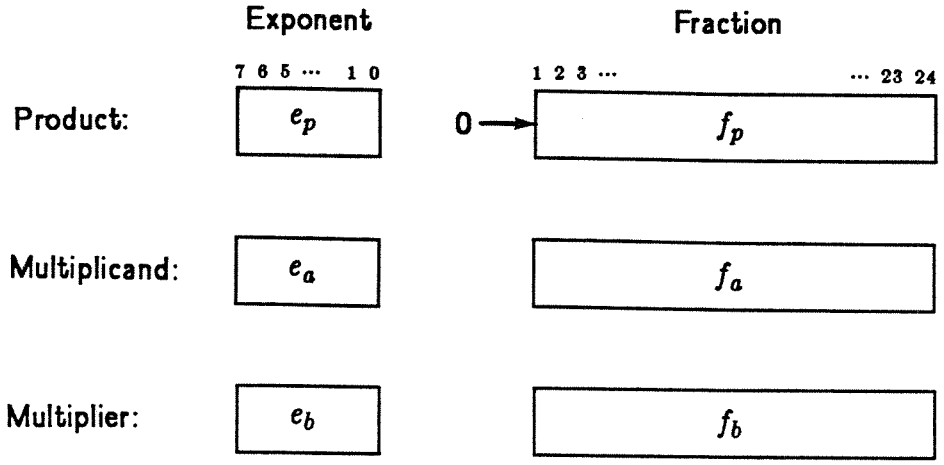
The options available when designing a custom integrated circuit to perform arithmetic are different from those available when designing with standard parts. In particular, it is usually impractical to consider using a full parallel implementation of multiplication and division because of the vast silicon area that would be required. This section describes one alternative that uses more economical shift-and-add techniques.

The floating point multiplication operation can be separated into two fairly distinct parts. First, the exponent of the product is formed by adding the exponents of the multiplicand and the multiplier, and second, the fractional part of the product is the high-order half of the 48-bit product of the two 24-bit input fractions. These two results may be computed separately, although it may be necessary to apply a one-bit normalization shift when finally combining them.

Implementing the fractional multiplication requires a 25-bit register to hold the product and two 24-bit registers for the multiplicand and multiplier, as shown in Figure A–1. The product register is initialized to zero. If the low-order bit of the multiplier is a one, then the multiplicand register is added to the product register. Next, the product and multiplier registers are each shifted to the right by one bit position. These steps are repeated twenty-three more times, for a total of once for each bit of the multiplier fraction. When completed, the product register will contain the high-order half of the fractional product, as required. The low-order half was progressively shifted out and discarded. An additional shift may be necessary to normalize the product. The time required to compute the product is thus determined by the time needed to perform twenty-four additions and twenty-five shift operations, but note that it may be convenient to combine the shifts and the additions into a single operation.

The floating point addition operation is conceptually the same as the version made with commercial components. It is summarized in Figure A–2. The primary difference is that instead of performing the initial denormalization and final renormalization in a single cycle, these operations are accomplished with a sequence of single bit shifts. The denormalization operation shifts the fractional part of one addend while simultaneously incrementing its exponent. Since the fraction consists of twenty-four bits, denormalization requires at most twenty-four shift operations. A single addition forms the sum, and the final renormalization may require up to twenty-four additional shifts. Thus, forty-eight shifts and one addition are required to complete a floating point addition.

A floating point division operation is made up of repeated shifts and subtractions. The exponent of the result is the difference of the dividend exponent and divisor exponent. Three more registers, shown in Figure A–3, are used to form the quotient of the fractional parts. Two of these registers initially contain the dividend and divisor, and the quotient will be formed in the third. If the dividend is at least as large as the

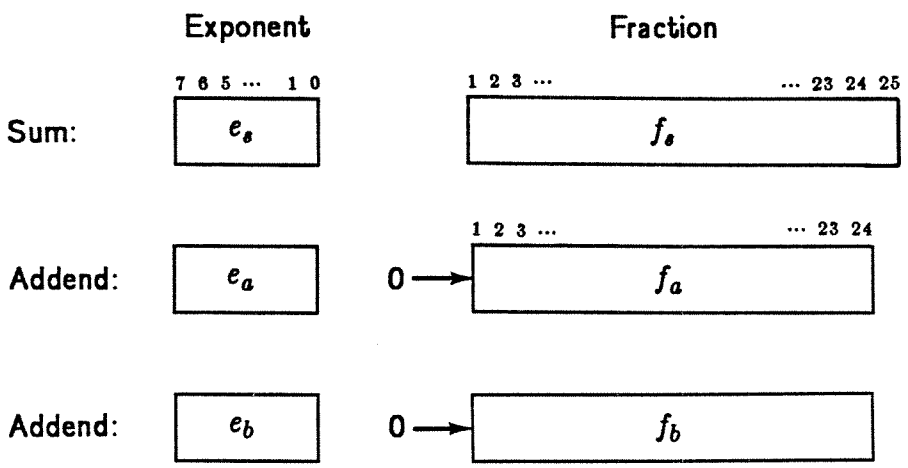


$e_p := e_a + e_b; \quad f_p := 0;$
repeat 24 times
  if $f_b(24) = 1$ **then** $f_p := f_p + f_a;$
  shift f_p **right** 1;
  shift f_b **right** 1;
if $f_p(1) = 0$ **then**
  shift f_p **left** 1;
  $e_p := e_p - 1;$

Figure A–1. Registers and operations used in floating point multiplication.

divisor, the difference between them replaces the dividend, and the low-order bit of the quotient is set. Otherwise, this bit is cleared, and the dividend is left unchanged. Next, the quotient and dividend are each shifted to the left by one bit position. After these two steps have been repeated twenty-four times, the quotient is complete except for a possible normalization step. The division operation uses twenty-four additions and up to twenty-five shifts, but again, as in the case of multiplication, it is convenient to combine the shifts and additions into a single operation.

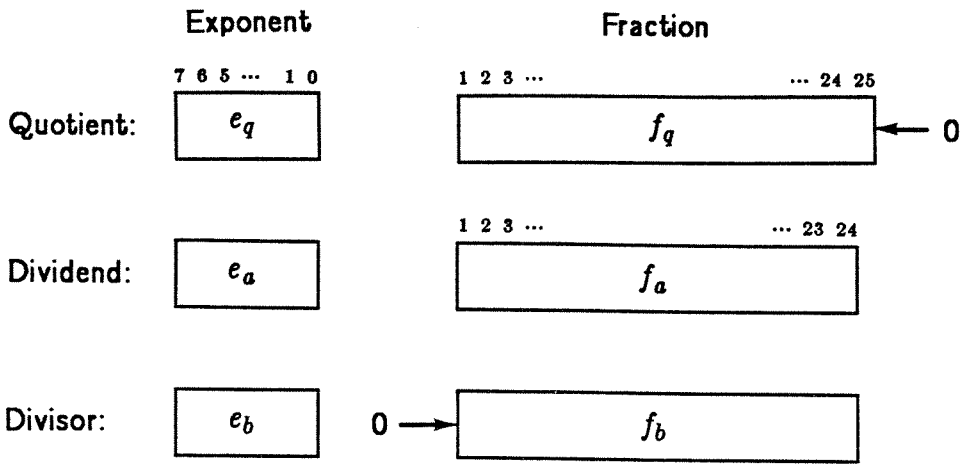
The square root operation may at first seem to be a little tough, but the method taught in high schools for extracting roots using only pencil and paper adapts rather nicely to a hardware implementation [FLOR63]. Refer to Figure A–4. Before processing the fraction, it is necessary to find the square root of the exponent part. If the



if $e_a > e_b$ **then**
  repeat $\max(e_a - e_b, 24)$ **times shift** f_b **right**;
  $e_s := e_a + 1$
ef $e_a < e_b$ **then**
  repeat $\max(e_b - e_a, 24)$ **times shift** f_a **right**;
  $e_s := e_b + 1$;
$f_s := (f_a/2) + (f_b/2)$;
$i := 0$;
while $f_s\langle 1\rangle = 0$ **and** $i < 24$ **do**
  shift f_s **left**;
  $e_s := e_s - 1$;
if $f_s\langle 1\rangle = 0$ **then** $e_s :=$ **smallest**;

Figure A–2. Registers and operations used in floating point addition and subtraction.

exponent is even, the exponent of the result is just half the exponent of the original number, and this value may be computed with a shift operation. On the other hand, if the original exponent is odd, it can be made even by adding one, but the fractional part must be halved in order to compensate. Extracting the root of the fractional part uses two registers. The result register holds twenty-six bits, although the low order two bits are permanently fixed at the value 01. The operand register is forty-nine bits wide. Its high order twenty-four bits are initially zero and will eventually contain the remainder of the root extraction. The next twenty-four bits are initialized to the input fraction, and the final bit is available in case the fraction had to be shifted to correct for an odd exponent. The root extraction procedure consists of a pair of steps repeated twenty-four times. First, the result register is compared with the high

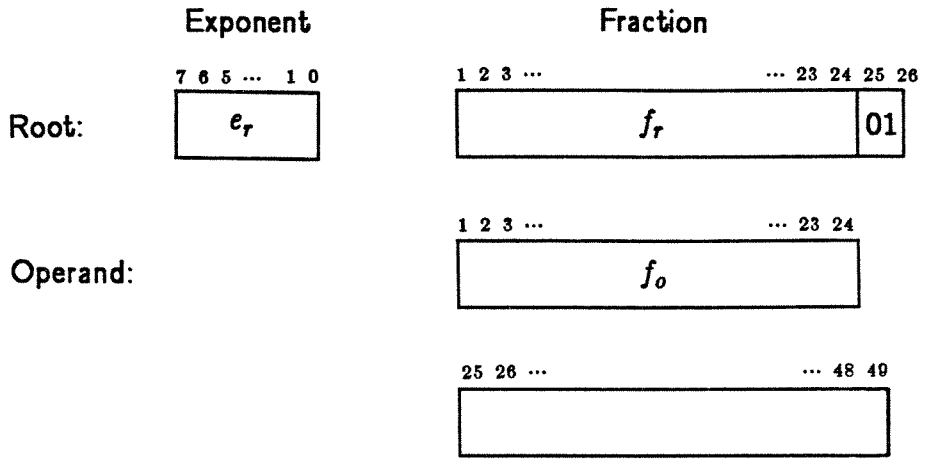


$e_q := e_a - e_b + 1;$
repeat 24 times
  if $f_a \geq f_b$ **then** $f_q\langle 25\rangle := 1;$ $f_a := f_a - f_b$
    else $f_q\langle 25\rangle := 0;$
  shift f_q **left;**
  shift f_b **right;**
if $f_q\langle 1\rangle = 0$ **then**
  shift f_b **left;**
  $e_q := e_q - 1;$

Figure A–8. Registers and operations used in floating point division.

order part of the operand register. If the operand register is larger, its contents are replaced with the difference, and a one is shifted into the twenty-fourth bit position of the result register. Otherwise, zero is shifted into the result. Second, the operand register is shifted to the left by two bit positions. Like division, then, the square root operation uses twenty-four additions and twenty-five shifts. Again, most of the shifts can be combined with the additions.

The performance of floating point arithmetic may be estimated by counting the primitive operations used in the implementation. To summarize the foregoing discussion, floating point multiplication, division, and square root operations each need twenty-four integer additions and twenty-five shifts. Floating point addition uses forty-eight shifts, but only a single integer addition. Recall also that the additions




```
if e_r⟨0⟩ = 1 then
  e_r:= e_r + 1;
  shift f right;
repeat 24 times
  if f_0⟨1 : 26⟩ ≥ f_q⟨1 : 26⟩ then
    f_0⟨1 : 26⟩:= f_0⟨1 : 26⟩ − f_r⟨1 : 26⟩;
    shift f_0 left twice;
    shift f_r⟨1 : 24⟩ left;
    f_r⟨24⟩:= 1
   else
    shift f_0 left twice;
    shift f_r⟨1 : 24⟩ left;
    f_r⟨24⟩:= 0;
```

Figure A–4. Registers and operations used to form the square root of a floating point number.

and shifts can to a large extent be combined. Therefore, if shifts can be performed at twice the rate of additions, then every floating point operation can execute in a similar amount of time. It turns out that such an assumption is justified, because the delay introduced by the carry propagation chain of an adder is its major performance limitation, and shift registers have no similar bottleneck. The performance of floating point operations is thus largely determined by the speed of integer additions, which may be conservatively estimated at between 100ns and 200ns. Including a small allowance for pipelining delays, it should take no more than about five microseconds to form the result of a floating point operation.

...continued next page...

Appendix B

Moving Between Subvolumes

Some of the ray tracing machines described earlier operate by partitioning the modeling space into a grid of subvolumes. This is done in order to reduce the number of polygons that must be processed to find the earliest intersection of a ray with a surface in the scene. It was also mentioned in passing that there is a relatively simple way to determine the sequence of subvolumes through which a ray travels. One such method will be described here, along with a technique for determining the first subvolume visited by a ray.

Consider first the equation describing the grid planes that are perpendicular to the x-axis. Suppose that the coordinates in the modeling space range from $x_{\min}$ to $x_{\max}$ and that the space is to be split into n_x portions along the x-axis. In this case, the equations of the planes partitioning the x-axis are given by

$$x = \left(\frac{x_{\max} - x_{\min}}{n_x}\right) i_x + x_{\min},$$

where i_x is an integer from zero to n_x that identifies the plane. Next, recall that the parametric representation of a ray is

$$\mathbf{r}(t) = (\mathbf{r}_1 - \mathbf{r}_0)t + \mathbf{r}_0.$$

Substituting this into the plane equation and solving for t gives

$$t = \left(\frac{x_{max} - x_{min}}{(r_{1x} - r_{0x})n_x}\right) i_x + \left(\frac{x_{min} - r_{0x}}{r_{1x} - r_{0x}}\right) = a_x i_x + b_x,$$

where r_{1x} is the x component of $\mathbf{r}_1$; likewise for r_{0x} and $\mathbf{r}_0$. a_x and b_x are simply abbreviations that will be convenient in later discussions. Similar expressions can be found in terms of the planes that split the y- and z-axes:

$$t = \left(\frac{y_{max} - y_{min}}{(r_{1y} - r_{0y})n_y}\right) i_y + \left(\frac{y_{min} - r_{0y}}{r_{1y} - r_{0y}}\right) = a_y i_y + b_y$$

$$t = \left(\frac{z_{max} - z_{min}}{(r_{1z} - r_{0z})n_z}\right) i_z + \left(\frac{z_{min} - r_{0z}}{r_{1z} - r_{0z}}\right) = a_z i_z + b_z.$$

To find which subvolume of the modeling space should be examined initially, it is first necessary to find the position along the ray at which it entered the modeling space. Using the x-component as an example, this can be done by computing the smallest value of t at which the x-component of the ray lies between x_{min} and x_{max}, the valid range of x. Similar values must be computed for the y- and z-components. The maximum of these three values of t is the position on the ray where it potentially passes into the modeling volume. If the maximum is negative, it must be replaced by zero because negative values of t correspond to points behind the origin of the ray. To find the index of the initial subvolume, it is necessary to substitute the maximum value of t into the three equations for t_x, t_y, and t_z. Solving for i_x, i_y, and i_z gives the initial subvolume indices. If these indices do not lie within the range from one to n_x, n_y, or n_z, then the ray misses the subvolume entirely. A program implementing this algorithm for computing the initial subvolume index is shown in Figure B–1.

Finding the successive subvolumes through which a ray passes is an incremental task. At each stage, only one of the three subvolume indices needs to be changed, because the next subvolume must be adjacent to the current one. This also means that the index that is modified must differ from its current value by exactly one. Consider, for the moment, the index in the x direction, i_x. Since rays are traced outward from their origin, values of t in the successively examined subvolumes must increase. From the expression for t in terms of i_x, it is apparent that if a_x is positive, incrementing i_x will increase the value of t. If a_x is negative, i_x must be decremented in order to increase the value of t. Corresponding statements can be made about i_y and i_z. Thus, there is a preferred next direction for each of the three components of the index. To determine which of the three directions is the correct one, it is necessary to examine the three values of t found with the incremented or decremented versions of i_x, i_y, and i_z. The direction producing the smallest value of t is the one that should be taken. For example, suppose that when the updated value of i_y was substituted into the equation for t, the resulting value was smaller than that produced by either i_x or i_z. In this case, the index of the next subvolume would have the same i_x and i_z components as the current one, but the i_y component would be incremented or decremented according to the sign of a_y.

$t := 0;$

$a_x := \dfrac{x_{max} - x_{min}}{(r_{1x} - r_{0x})n_x}; \quad b_x := \dfrac{x_{min} - r_{0x}}{r_{1x} - r_{0x}}; \quad t := \max(t, \min(b_x, a_x n_x + b_x));$

$a_y := \dfrac{y_{max} - y_{min}}{(r_{1y} - r_{0y})n_y}; \quad b_y := \dfrac{y_{min} - r_{0y}}{r_{1y} - r_{0y}}; \quad t := \max(t, \min(b_y, a_y n_y + b_y));$

$a_z := \dfrac{z_{max} - z_{min}}{(r_{1z} - r_{0z})n_z}; \quad b_z := \dfrac{z_{min} - r_{0z}}{r_{1z} - r_{0z}}; \quad t := \max(t, \min(b_z, a_z n_z + b_z));$

$i_x := \left\lceil \dfrac{t - b_x}{a_x} \right\rceil; \quad t_x := a_x i_x + b_x + |a_x|;$

$i_y := \left\lceil \dfrac{t - b_y}{a_y} \right\rceil; \quad t_y := a_y i_y + b_y + |a_y|;$

$i_z := \left\lceil \dfrac{t - b_z}{a_z} \right\rceil; \quad t_z := a_z i_z + b_z + |a_z|;$

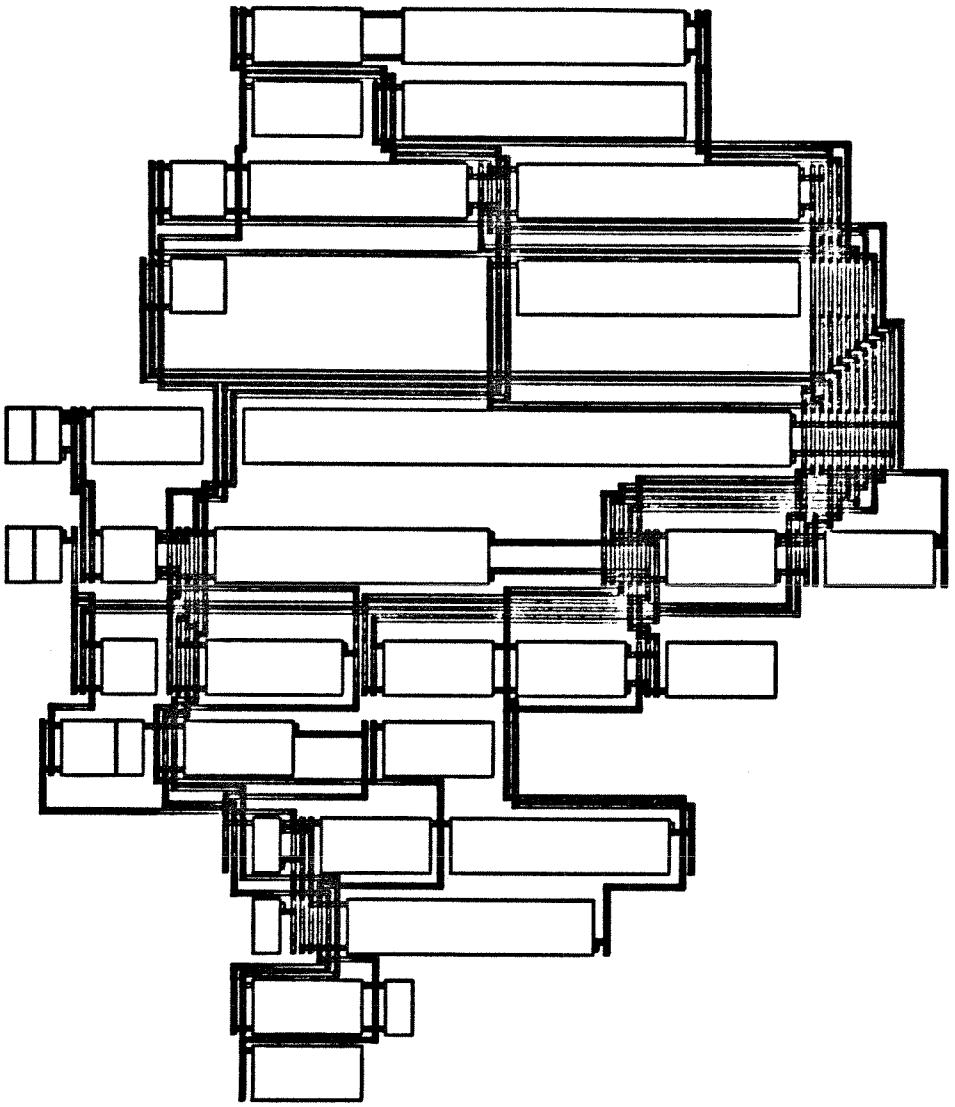
if $i_x < 1$ **or** $n_x < i_x$ **or**
$i_y < 1$ **or** $n_y < i_y$ **or**
$i_z < 1$ **or** $n_z < i_z$ **then comment**: The ray misses the scene. ;

Figure B–1. Program to compute the initial subvolume index.

if $t_x < \min(t_y, t_z)$ **then begin** $i_x := i_x + \text{sgn}(a_x); \quad t_x := t_x + |a_x|$ **end**
ef $t_y < \min(t_x, t_z)$ **then begin** $i_y := i_y + \text{sgn}(a_y); \quad t_y := t_y + |a_y|$ **end**
ef $t_z < \min(t_x, t_y)$ **then begin** $i_z := i_z + \text{sgn}(a_z); \quad t_z := t_z + |a_z|$ **end**;

Figure B–2. Program to increment the subvolume index.

The computations outlined above can easily be performed incrementally. To do this, it is necessary to maintain three values of t, one for each component of the subvolume index. These values represent the distance along the ray of the next subvolume boundary crossing in each of the three directions. Choosing a direction is as simple as determining the smallest of these three values. Suppose that the value of t corresponding to the z direction was smaller than either of the other two. When it is time to select the next subvolume, the new index would be computed by either incrementing or decrementing the current value of i_z, depending of course on the sign of a_z. The value of t maintained for the z direction would also have to be updated by adding $|a_z|$. A program for incrementing the subvolume index is given in Figure B–2.



Appendix C

Programming in Silicon

As the investigations described in this dissertation progressed, it became apparent that the existing tools for integrated circuit design were not very suitable for the kinds of graphics processors that were envisioned at the time. It seemed appropriate, then, to develop a family of tools more optimized to that application. The goal was to allow integrated circuit design to be treated as more of a programming task than a layout generation task. The designer would proceed by coding his application in a rather ordinary, though greatly simplified, programming language. The resulting source code could be at least partially debugged by using a functional simulator. After debugging, another tool would translate the program into a layout suitable for fabrication as an integrated circuit.

While the tools were being developed, it seemed that the best strategy would be to implement the graphics processors using bit serial arithmetic. This technique offers single wire communication and reduced circuit area when compared to a conventional parallel implementation of arithmetic operations. Consequently, most of the emphasis has been on the tools for bit serial designs. As work proceeded on the graphics machines, however, the functional benefits of floating point arithmetic began to outweigh the implementation benefits of serial arithmetic, and the serial design tool became somewhat irrelevant. On the other hand, some aspects of the tool might be considered interesting enough in their own right to warrant mention. Therefore, the use and implementation of the design aids that were developed will be reported here, even though most of the work was done in 1981.

The tools to be described are not applicable to the design of an arbitrary chip. In particular, they are not meant to handle cases where analog effects are needed, where there are stringent timing requirements, or where layout tricks are to be used. On the other hand, they do seem to be useful for a variety of applications in digital signal processing and computer graphics.

The tools are aimed mainly toward individuals who have a special purpose application in mind, and who want a few working chips that don't really push the limits of technology. These individuals might not be willing to make a substantial investment in time or effort in order to obtain these chips. Large semiconductor houses, on the other hand, operate under a different set of constraints. They typically compete in high volume markets and can afford the greater design time that it takes to get a more efficient chip. They might therefore be expected to be less interested in ways of minimizing human interaction in the design process at the expense of performance in the final result.

The term "Silicon Compiler" is an unfortunate one, having been coined more for its appeal as a buzz word than for any technical merit. The trouble is that while Fortran compilers compile Fortran, Pascal compilers compile Pascal, and Simula compilers compile Simula, silicon compilers do not compile silicon. At best, they compile *into* silicon, but in many cases their operation more nearly resembles that of an assembler. In any event, the phrase "Silicon Compiler" has caught on and appears to be with us forever. Also, it seems inevitable that the term will be applied to some of the tools described here. Therefore, succumbing to the unavoidable, the programming language discussed below has been named Silicon, making the term "Silicon compiler" appropriate in at least some sense.

C.1 Silicon Programming

Superficially at least, Silicon resembles many other programming languages, most notably Pascal [WIRT71]. A major difference is the lack of complicated data structures and variable typing. Anyone with any exposure to programming should have no trouble understanding the sample programs used in this document, and descriptions of the more common aspects of the language have therefore been minimized.

A feature of Silicon not shared by many other programming languages is its communications ports. A program can use them to communicate with its operating environment, or several portions of a single program can communicate among themselves. These uses correspond to the pads and on-chip communication of an integrated circuit. Basically, ports consist of some sort of channel that has two ends. Integers go into one end of the channel and come out the other end at some later time. Operations on communications ports occur in a strictly sequential manner. That is, one end of a port may be multiplexed, but it may not be connected to more than one thing at once. This model is well matched to the physical reality of a wire.

The availability of concurrency in Silicon is another feature distinguishing it from most other programming languages. The basic idea is that every operation in the program may be performed as soon as its operands are ready, so that an arbitrary number of operations might proceed in parallel. The compiler attempts to implement the program in a way that makes operations occur as soon as possible, although it may not be able to exploit all of the concurrency that is potentially available in a given program. This simple notion of concurrency, together with the communications ports mentioned above, makes it possible to program fairly complicated combinations of cooperating processes, even though Silicon has no syntactic features devoted to the description of concurrency. This will be illustrated later by means of example.

The techniques for dealing with concurrency in Silicon are similar to those described by Kahn and MacQueen [KAHN77]. The difference is that parallel operations in Silicon are not specified explicitly. This means that concurrency may be available where the programmer might not have anticipated it. Given a greater amount of potential concurrency, it is conceivable that the compiler would be able to do a better job of selecting which operations should actually be done in parallel.

Apart from communications ports, the only datatype in Silicon is integer. Integer values are represented in the generated layout as binary numbers consisting of an arbitrary but fixed number of bits. That is, the size of every number must be known to the compiler, but numbers having different sizes may coexist within the same program. In general, it is not necessary for the programmer to specify the size of each number, since the compiler can usually determine it from context. The compiler will, of course, generate an error message if it is unable to make this determination.

Variables in Silicon are somewhat different from those in a language like Pascal, even though they can be used in much the same way. The difference is that in Silicon, variables are names for values. In contrast, Pascal variables are names for storage locations, and the storage locations in turn *contain* values. In Silicon, storage allocation is completely independent of variable naming. From the programmer's point of view, this means that in certain cases, temporary variables may be used without incurring any extra cost in storage. Furthermore, a variable used in one section of the program does not necessarily have any relationship to a variable with the same name used in another part of the program. They are related only if each variable names the same value, as would be the case if there were no intervening assignment statements.

As mentioned earlier, the details of Silicon can easily be understood by drawing on prior experience with other programming languages, like Pascal. On the other hand, the broader notions of how one programs chips in Silicon are best illustrated by means of example. The following few sections present a variety of examples, each chosen to highlight some particular facet of Silicon.

C.2 Example: Inner Product

This first example is a simple Silicon program that computes a running inner product of two sequences of numbers. That is, it accepts two streams of numbers, multiplies the corresponding numbers from each stream, and maintains the sum of all such products obtained so far. Every time it gets a new pair of input numbers, it places the current value of the accumulation into the output stream. Thus, for every pair of numbers appearing at the inputs, one number will be generated at the output. Here is the program:

```
1    a:= asyncSerIn(16,0);
2    b:= asyncSerIn(16,0);
3    c:= asyncSerOut(16,0);
4    sum:= 0;
5    DO {
6      sum:= sum + a.get*b.get;
7      c.put(sum)
8    }
```

The first three lines of this program create communications ports for receiving inputs and transmitting outputs. The arguments specify that the size of the words passing through the ports will be sixteen bits, and that there will be zero words of buffering inside the port itself. All three ports are asynchronous serial ports, so some handshaking will be involved when values are passed through the ports. The details of this handshaking are taken care of by the compiler and need not be of any concern to the programmer. Notice that the port values are assigned to variables in the same way that an integer value is assigned to a variable, as in Line 4.

The code for the actual inner product appears in the final four lines of the program. The DO-statement causes the statements enclosed within braces to be executed repeatedly. In Line 6, the next value of the sum is computed with two values obtained from the input ports. This new sum is sent to the output port in the following line. Notice that quite a bit of synchronization may be going on behind the scenes in these two statements. Since the two input ports are asynchronous, one of them may produce a value much sooner than the other, and the compiler will have to insert some buffering. Similarly, it may take some time for the environment to accept the sum. This kind of detail is handled completely by the compiler.

In the program above, input values were permitted to arrive at arbitrary times. This, in turn, required the compiler to generate some extra circuitry to perform the synchronization. It is sometimes possible to write programs that accept their inputs at a fixed rate and produce their outputs at a fixed rate. Such programs are much simpler in terms of the generated code. A synchronous version of the inner product program is as follows.

```
1    a:= syncSerIn(16,1);
2    b:= syncSerIn(16,1);
3    c:= syncSerOut(16);
```

```
4    sum:= 0;
5    DO {
6      sum:= sum + a.get*b.get;
7      c.put(sum)
8    }
```

Notice that the only difference between this program and the asynchronous version is that the asynchronous ports have been replaced with synchronous ones. Again, the first parameter is the word size, but here, the second parameter specifies the number of bit times that will elapse between successive words passing through the port. The output port does not have a second parameter because the compiler computes the separation value. Many signal processing programs may be written synchronously in this manner.

A third version of the program computes the inner product of groups of four pairs of input numbers. That is, the program will send the sum to the output only after it has accepted four pairs of numbers. After that, the sum will be reset to zero in preparation for the next four pairs. The code for this program is again rather straightforward:

```
1    a:= asyncSerIn(16,0);
2    b:= asyncSerIn(16,0);
3    c:= asyncSerOut(16,0);
4    DO {
5      sum:= 0;
6      n:= 4#3;
7      WHILE n ~= 0 DO {
8        n:= n-1;
9        sum:= sum + a.get*b.get
10     };
11     c.put(sum)
12   }
```

One point that may need clarification here is the assignment statement in Line 6. The number sign (#) specifies that the value four should be represented with three bits. Why was it necessary to specify the size of this particular constant but none of the others? The compiler must be able to determine the size of every value so that it can generate the correct layout to represent that value. Often, the size of a number can be determined from the context of its use. For example, the zero in Line 5 is known to be a sixteen-bit value because the variable **sum** is used in a sixteen-bit expression in Line 9. Similarly, since the size of four was specified in Line 6, the sizes of zero and one in Lines 7 and 8 can be determined as well. Generally speaking, the compiler is able in most cases to determine the sizes of constants. Where it cannot, it will issue an error message.

It turns out that a small amount of concurrency is available in this program. The computation of the sum and the updating of the loop counter in Lines 8 and 9 do not depend on each other and may therefore be executed in parallel. The compiler is able to detect and, in some cases, capitalize on situations like this by generating parallel hardware.

The third version of the inner product program, above, is an example of a program that could not be made synchronous merely by changing the port definitions. This is because the compiler cannot determine the number of times that the WHILE-loop will execute, and so it cannot determine the separation between successive values passing through the output port. In a more general case, the body of a WHILE-loop might not even be repeated the same number of times whenever the loop is executed. In the current case, however, it is possible to rewrite the program to make it synchronous, as demonstrated in this fourth version:

```
1    a:= syncSerIn(16,1);
2    b:= syncSerIn(16,1);
3    c:= syncSerOut(16);
4    DO c.put(a.get*b.get + a.get*b.get +
5             a.get*b.get + a.get*b.get)
```

A final version of the inner product program, shown below, computes a running inner product, but maintains the sum with extended precision. The input and output numbers will be treated as sixteen-bit unsigned binary fractions with the binary points at the left. Internally, the sum will be represented as a thirty-two bit unsigned binary fraction. In Line 6, the input values are extended, without affecting the positions of the binary points, by concatenating sixteen bits of zero onto their high-order ends. This operation is basically a ploy to permit retrieval of the entire, thirty-two bit product when the two values are multiplied, since Silicon provides only the low-order half of a product. After the multiplication, accumulation proceeds as before, except that now the sum is a thirty-two bit fraction. Finally, in Line 7, the most significant half of the sum is extracted using bit subscription and is sent to the output port.

```
1    a:= asyncSerIn(16,0);
2    b:= asyncSerIn(16,0);
3    c:= asyncSerOut(16,0);
4    sum:= 0;
5    DO {
6      sum:= sum + (0#16'a.get)*(0#16'b.get);
7      c.put(sum[31:16])
8    }
```

C.3 Example: PDP-8

The next example, a PDP-8, is substantially larger than the previous one. The part of the program that emulates the instruction set of the PDP-8 was adapted almost directly from the manual [DEC72]. The bus structure of the PDP-8, however, uses many more signals than can easily be brought out of a single integrated circuit. Therefore, this version of the PDP-8 has a rather drastically modified I/O and memory bus. The design of this replacement bus is not especially clever, but it should serve to illustrate the essential features of Silicon.

The PDP-8 has three main registers that are of concern to the programmer. They are initialized by the following statements.

```
1    (*  Register definitions.  *)
2    pc:= 0#12;    -- Program counter.
3    ac:= 0#12;    -- Accumulator.
4    l:=  0#1;     -- Link register.
```

The program counter and accumulator are each twelve bits wide. The link register, really an extension of the accumulator, is one bit wide.

This example illustrates the two comment conventions available in Silicon. A region of text may be commented out by enclosing it between matched pairs of parentheses and asterisks, as was done in Line 1. Alternatively, the compiler will ignore any text following and including a double dash on any single line, as shown on Lines 2 through 4.

The modified bus structure for the PDP-8 consists of a data bus, an address bus, and some miscellaneous control lines. It can be declared by the following Silicon statements.

```
5    (*  Communications port definitions.  *)
6    addressBus:= syncParOut(12);
7    dataBus:= syncParIO(12);
8    memFetch:= syncParOut(1);
9    memStore:= syncParOut(1);
10   startIO:= syncParOut(1);
11   stopIO:= syncParOut(1);
12   skipIO:= syncParIn(1);
13   loadIO:= syncParIn(1);
```

Notice that these statements declare parallel communications ports rather than the serial ones used in the previous example. The single parameter gives the width of the port. This is both the size of the words that will pass through the port and the number of pads that will be used to implement it. Input ports respond to the **get** attribute by sampling the current state of the pads and returning that value. Output ports drive the associated pads with the value received in the most recent **put** attribute invocation. The bidirectional communications ports, which are created by **syncParIO**, respond to both the **put** and **get** attributes. They behave either as an input port or an output port, depending on which of these attributes was most recently executed.

The operation of storing a word into the memory of the PDP-8 is one that is apt to be useful in many parts of the program. Silicon provides two mechanisms by which such commonly used pieces of code may be given a name and invoked as often as necessary. The first mechanism is a simple macro facility for replicating code at various points within the program. The other mechanism uses a similar syntax but behaves more like a subroutine. The difference is that when a macro is referenced, the compiler is free to allocate new devices in order to implement the code in the macro body. On the other hand, each invocation of a subroutine must reuse the same set

of devices. Thus, subroutines economize on device count by reducing the possibilities for concurrency.

The memory store operation may be written as a macro:

```
14    (*  Store a word of data at a
15        specified address.  *)
16    MACRO store(address,data
17                |memStore,addressBus,dataBus,d)
18      BEGIN
19        addressBus.put(address);
20        dataBus.put(data);
21        memStore.wait(addressBus).wait(dataBus)
22                .put(1).delay(d).put(0).delay(d);
23        dataBus.wait(memStore).get
24      END;
```

This definition has two input parameters, **address** and **data**, which convey the address of the word in memory that is to be modified and the new value for that word. Notice that these parameter names are written before the vertical bar and are enclosed within the parentheses following the macro name. The definition also references four global variables: **memStore**, **addressBus**, **dataBus**, and **d**. The last of these, **d**, is expected to have as its value an integer specifying timing delays. The others are some of the communications ports that were described earlier. Global input parameters are listed after the vertical bar in the declaration.

The body of the definition shows how the address and data values are placed onto their respective busses; this happens in Lines 19 and 20. Notice that these two lines are independent of each other, so that the operations may occur in either order, or even at the same time. In the next line, the **memStore** control port waits for both of these operations to complete before transmitting a write pulse. The **wait** attribute of parallel communications ports takes as its parameter another communications port. When this attribute is invoked, both ports complete any pending operations and wait for the other to become idle. When both are idle, each is free to proceed. Thus, the **wait** attribute may be used to synchronize ports that are otherwise independent. The delay attribute of parallel ports, used in Line 22, causes the port to wait for the specified number of clock cycles. The actual wait time must be a compile-time constant. After pulsing the memory store line, the program stops driving the data bus by attempting to read from it, ignoring the value so obtained. Notice that a little bit of concurrency is available when this definition of **store** is invoked. The caller does not have to wait while the store operation finishes because the definition does not return any value that the caller needs.

The memory fetch operation can be packaged like the store operation:

```
25    (*  Fetch a word of data from a
26        specified address.  *)
27    MACRO fetch(address|memFetch,addressBus,dataBus,d)=>
28              (data)
29      BEGIN
30        addressBus.put(address);
```

```
31        memFetch.wait(addressBus).put(1).delay(d);
32        data:= dataBus.wait(memFetch).get;
33        memFetch.wait(dataBus).put(0).delay(d)
34      END;
```

This definition has a single input parameter that supplies the address of the memory location. As in the case of the store operation, the definition of **fetch** references four global variables that specify the communications ports and the delay time. These could, of course, have been passed as input parameters, but then there would be five parameters to be included every time a memory fetch was needed. The single output parameter, **data**, serves to return the memory value to the caller of the definition.

The body of the **fetch** definition is also much like that of **store**. The statement in Line 30 drives the address onto the address bus. On Line 31, the memory fetch signal is held high for a while in order to allow sufficient time for the memory to respond. Notice, however, that this will not happen until after the **addressBus** port has begun to supply the address. Next, after the memory fetch line has been asserted for a while, a value is sampled from the data lines. Finally, in Line 33, the memory fetch signal is returned to its quiescent state. By delaying the **memFetch** output port, the program insures that the signal remains unasserted for some minimal length of time. Even though this delay is taking place, the result of invoking the **fetch** definition is available as soon as the value is received from the data bus, because after that point, the result no longer depends on the memory fetch port.

Another useful definition is one that computes an effective address from an instruction and the value of the program counter. In the PDP-8, nine bits are available to specify a twelve-bit memory address. Two of these are flag bits, so there are really only seven bits in the instruction word that end up as part of the final twelve-bit effective address. Depending on one of the flag bits, the five high-order address bits are either zero or come from the five high-order bits of the program counter. The other flag bit specifies whether indirect addressing is to be performed. Finally, if indirect addressing is used with one of a few special memory locations, the content of that location is automatically incremented. The code is somewhat simpler than this explanation might suggest:

```
35    (*  Compute the effective address referenced in
36        the current instruction.  *)
37    MACRO effectiveAddress
38          (|instruction,pc,
39            memFetch,memStore,addressBus,dataBus,d)
40        =>(address)
41      BEGIN
42        i:= instruction[11-3];  -- Indirect bit.
43        p:= instruction[11-4];  -- Page bit.
44        a:= instruction[11-5:11-10];  -- Low-order addr.
45        address:= (IF p THEN 0 ELSE pc[11-0:11-4])'a;
46        IF i THEN  -- Indirect addressing.
47          { a, address:= address, fetch(address);
48            IF a[11-0:11-8] = 1 THEN store(a,address+1)
49          }
```

```
50      END;
```

This definition has no input parameters, but it does reference seven global variables. The first two of these, **instruction** and **pc**, are expected to name the current instruction, supplying the low-order bits of the address, and the program counter, possibly supplying the high-order bits. The remaining five global variables are those required by the memory fetch and store definitions, which may be needed to compute the effective address. The single output parameter, **address**, is used to return the result.

Notice that the definition of **effectiveAddress** contains three local variables: **i**, **p**, and **a**. Any variable used within the body of a definition, but not declared in the header, is assumed to be local to that definition. Of course, as for any other variable in Silicon, local variables are declared simply by their use.

The bit subscripts used in Lines 42 through 45 and on Line 48 are rather more complicated than they might be because the PDP-8 numbers bits within a word differently than does Silicon. In the PDP-8, bit zero is the high-order bit, while in Silicon it is the low-order bit. Thus, the expression **11-3** in Line 42 converts bit three in PDP-8 conventions to bit eight in Silicon conventions. Expressions may be used as the subscript values even though the compiler requires constant subscripts because constant expressions will be evaluated at compile-time.

Examples of definition invocation appear in Lines 47 and 48 in the definition of **effectiveAddress**. They look just like an ordinary procedure call that might be found in many other languages. The result is returned in place of the call, also in the usual manner. Line 47 also contains a multiple-valued assignment statement. In this case, the expressions on the right side produce two values that are assigned to the corresponding variables on the left side.

With all of the pieces in hand, the structure of the main PDP-8 emulation loop is relatively straightforward. The program repeats a fetch-decode-execute cycle indefinitely. A **CASE**-statement describes instruction decoding, and the individual cases contain the code for executing the corresponding instruction. Since a PDP-8 has only eight instructions, this code is not very difficult to produce. Actually, the **CASE**-statement was never implemented in the most recent implementation of Silicon, but a conditional statement could serve equally well. Here, then, is the main body of the code, with additional remarks to follow:

```
51      (*  Delay time.  *)
52      d:= 5;
53
54      (*  This is the main instruction
55          fetch-decode-execute loop.  *)
56      DO
57        { instruction:= fetch(pc);
58          pc:= pc + 1;
59          CASE instruction[11-0:11-2] OF
60
61          0:  -- AND Y (Logical AND)
```

```
62          ac:= ac AND fetch(effectiveAddress)
63
64        1:  -- TAD Y (Two's complement ADD)
65          ac, l:= ac + fetch(effectiveAddress),
66                  NOT (l EQV addSubCarryOut)
67
68        2:  -- ISZ Y (Increment and skip if zero)
69          { address:= effectiveAddress;
70            temp:= fetch(address) + 1
71            store(address,temp);
72            IF temp = 0 THEN pc:= pc + 1
73          }
74
75        3:  -- DCA Y (Deposit and clear accumulator)
76          { store(effectiveAddress,ac);
77            ac:= 0
78          }
79
80        4:  -- JMS Y (Jump to subroutine)
81          { address:= effectiveAddress;
82            store(address,pc);
83            pc:= address + 1
84          }
85
86        5:  -- JMP Y (Unconditional jump)
87          pc:= effectiveAddress
88
89        6:  -- IOT (Input/Output transfer)
90          { addressBus.put(instruction);
91            dataBus.put(ac);
92            startIO.wait(addressBus).wait(dataBus)
93                   .put(1).delay(d).put(0).delay(d);
94            dataBus.wait(startIO).get;
95            stopIO.wait(dataBus).put(1).delay(d);
96            IF skipIO.wait(stopIO).get THEN pc:= pc + 1;
97            IF loadIO.wait(stopIO).get THEN
98              ac:= dataBus.get;
99            stopIO.wait(skipIO).wait(loadIO)
100                 .wait(dataBus).put(0).delay(d)
101         }
102
103       7:  -- Operate instructions.
104           IF NOT instruction[11-3] THEN  -- Group 1.
105             {   -- Clear accumulator.
106               cla:= instruction[11-4];
107                 -- Clear link.
108               cll:= instruction[11-5];
109                 -- Complement accumulator.
110               cma:= instruction[11-6];
111                 -- Complement link.
112               cml:= instruction[11-7];
113                 -- Rotate accumulator and link right.
114               ror:= instruction[11-8];
115                 -- Rotate accumulator and link left.
116               rol:= instruction[11-9];
117                 -- Rotate two places.
118               two:= instruction[11-10];
119                 -- All three rotate bits.
```

```
120             rot:= instruction[11-8:11-10];
121               -- Increment accumulator.
122             iac:= instruction[11-11];
123             (* Sequence 1 *)
124             IF cla THEN ac:= 0;
125             IF cll THEN l:= 0;
126             (* Sequence 2 *)
127             IF cma THEN ac:= NOT ac;
128             IF cml THEN l:= NOT l;
129             (* Sequence 3 *)
130             IF iac THEN ac:= ac + 1;
131             (* Sequence 4 *)
132             CASE rot OF
133             2: l,ac:= ac[11], ac[10:0]'l
134             3: l,ac:= ac[10], ac[9:0]'l'ac[11]
135             4: l,ac:= ac[ 0], l'ac[11:1]
136             5: l,ac:= ac[ 1], ac[0]'l'ac[11:2]
137           }
138         EF NOT instruction[11-11] THEN  -- Group 2.
139           {   -- Clear accumulator.
140             cla:= instruction[11-4];
141               -- Skip on minus accumulator.
142             sma:= instruction[11-5];
143               -- Skip on zero accumulator.
144             sza:= instruction[11-6];
145               -- Skip on non-zero link.
146             snl:= instruction[11-7];
147               -- Reverse the sense of the skip.
148             rev:= instruction[11-8];
149               -- OR accumulator with switch register.
150             osr:= instruction[11-9];
151               -- Halt.
152             hlt:= instruction[11-10];
153             skip:= (sma AND ac[11-11])
154                 OR (sza AND ac ~= 0) OR (snl AND 1);
155             (* Sequence 1 *)
156             IF NOT (skip EQV rev) THEN pc:= pc + 1;
157             (* Sequence 2 *)
158             IF cla THEN ac:= 0;
159             (* Sequence 3 *)
160             IF osr THEN { (* No Switch Register! *) };
161             IF hlt THEN BREAK
162           }
163          ELSE  -- Extended instruction.
164           { (* No Extended Arithmetic Element (EAE). *)
165           }
166     }
```

The first six PDP-8 instructions, with opcodes 0 through 5, are the memory referencing instructions. These are all fairly straightforward, although there are a few interesting points. First, notice the variable **addSubCarryOut** in the code for the TAD instruction on Line 66. This variable, which was never explicitly assigned a value, is automatically set to the value of the carry out of the most recent addition or subtraction operation. In the present program, it is used to toggle the link bit or not, depending on the result of the prior addition. Although potentially available

after every addition or subtraction, **addSubCarryOut** costs nothing in terms of the final layout unless it is actually used. If ignored, it will simply atrophy.

Next, consider the ISZ instruction on Lines 68 through 73. The code for this instruction uses two temporary variables: **temp** and **address**. The compiler can determine that these are temporaries because they are referenced nowhere else. The variable **address** is used again in the JMS instruction, but there is an intervening assignment statement, and the values named by the two occurrences of the variable are therefore independent. The compiler will not generate storage for temporary variables unless it must do so in order to satisfy timing or synchronization constraints.

Lines 114 through 118 contain three statements that extract values from various bit fields of the instruction and assign them to variables. Notice, however, that these variables are not referenced anywhere else in the program. The three statements may have been written to improve the clarity of the program, or they may simply be a leftover from a previous version of the program. In any case, not only will the compiler refrain from generating storage for these variables, it will not even generate circuitry to compute their values. In general, the compiler will not generate code that cannot affect the behavior of the communications ports. This property is also useful, for example, if the program includes code to be used with a functional simulator when debugging. Arbitrarily complex debugging code may be left in even the final version of a program, from which it will automatically be removed by the compiler.

The single I/O instruction on the PDP-8, called IOT, is rather similar to a memory write followed by a memory read. It supplies the contents of the accumulator to the data bus, using the instruction itself as the address. The **startIO** port takes the place of the **memStore** port used for accessing memory. Peripherals that have access to the bus are expected to interpret the low-order nine bits of the instruction in some reasonable manner. After the write phase of the I/O operation is complete, the program begins the read phase. Again, this is similar to a memory fetch except that the **stopIO** port is used instead of **memFetch**. The input port **skipIO** provides a way for the peripheral device to specify a skip operation. This is typically used for testing the status of a device. The **loadIO** input port signals whether this is an input operation. The peripheral asserts this signal if it has placed on the data bus a value that should be loaded into the accumulator.

The operate instruction, with opcode 7, is probably the most complicated and is certainly the strangest instruction in the PDP-8. The low-order bits of the instruction each specify some operation that may take place. The bits may be set in various combinations, causing the corresponding combinations of operations to occur. Actually, the situation is somewhat more complicated even than this. There are some flag bits in the instruction that select between alternate interpretations of the remaining bits. These alternatives are known as the Group 1 and Group 2 operate microinstructions and the extended instructions.

For the purpose of this discussion, it is enough to consider only the Group 1 operate microinstructions. The bits used to encode the instruction are broken up into

four functional groups for clearing, complementing, incrementing, and shifting the accumulator and link register. The four groups are supposed to operate in sequence, so that, for example, by specifying the bits for clearing and complementing the accumulator, one could be certain of setting all of its bits. Within one of the groups, operations could proceed in parallel. Thus, the accumulator and the link could be cleared at the same time.

These constraints on the execution order of microinstructions are implicit in the Silicon code. For example, the compiler can determine that the complement operation in Line 127 cannot be performed until the clearing operation in Line 124 is complete. On the other hand, neither of these statements interacts with Line 125, which clears the link register, so the clearing may occur in parallel with either of the others. The remaining cases are similar. While on the subject of concurrency, notice that all of the bit field extractions on Lines 106 through 122 might potentially be performed at the same time if the compiler finds this to be an advantageous implementation.

The version of the PDP-8 presented here is by no means an ideal one. It was intended to serve mainly as an illustration of some of the features of Silicon. It is especially weak in its off-chip dealings. The memory and I/O structure is not especially flexible, and in particular, the PDP-8 emulator chip imposes what may turn out to be unreasonable timing constraints on its memory and peripheral devices. There is no provision for a delayed response to any processor request. Finally, interrupt handling and direct memory access (DMA) have been entirely omitted. None of the modifications required to correct these defects is impossible, but neither would they increase the illustrative value of the example.

C.4 Example: Convolution

The next example, taken from the field of digital signal processing, is a simple one-dimensional convolution that might be used to construct a finite impulse response (FIR) filter. The program contains four values that make up the convolution kernel, and it has storage for four samples of the input signal. Figure C–1 shows a block diagram. At each step of the computation, the kernel values are multiplied by the corresponding signal values. The four products are then added together, and the result is the next in the sequence of values that make up the convolution. Finally, a new sample value is shifted in, replacing the oldest one which is shifted out, and the process is repeated. Here is the program that implements the convolution:

```
1    sIn:= asyncSerIn(16,0);     -- Input signal samples.
2    cOut:= asyncSerOut(16,0);   -- Convolution output.
3    (*  Storage for the input signal...  *)
4    s1, s2, s3, s4:= 0, 0, 0, 0;
5    (*  Kernel values to be supplied...  *)
6    k1, k2, k3, k4:= ...something...;
7    DO
```

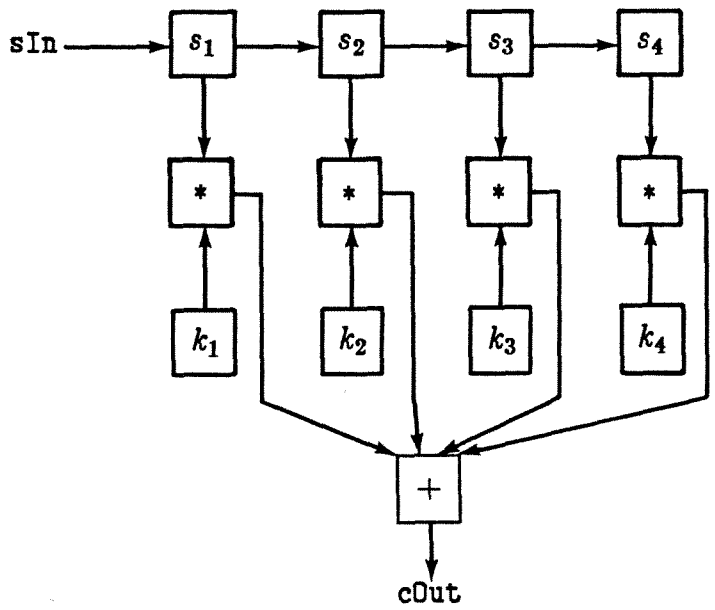


Figure C–1. Block diagram of a machine capable of convolving a four element kernel with a sequence of signal samples. Boxes labeled s_i represent storage for samples, while those labeled k_i represent kernel values.

```
 8      { (*  Generate a value of the convolution.  *)
 9        cOut.put(s1*k1 + s2*k2 + s3*k3 + s4*k4);
10        (*  Shift in a new value of the input signal.  *)
11        s1, s2, s3, s4:= sIn.get, s1, s2, s3
12      }
```

Although this program is relatively straightforward, it has some problems. First of all, the program works only for a convolution kernel with four coefficients. Although it is easy to increase this limit by changing the program, the required modifications would involve more than the simple alteration of a few constants. A powerful macro preprocessor could help in this particular situation, but the basic problem is that Silicon lacks any sort of higher-level data structures like arrays.

Another problem with the convolution program given above is that the kernel is fixed. That is, in order to change the characteristics of the filter, it would be necessary to fabricate a new chip. The following program eliminates that problem by allowing the kernel coefficients to be shifted in one at a time.

```
1    sIn:= asyncSerIn(16,0);     -- Input signal samples.
2    kIn:= asyncSerIn(16,0);     -- Kernel coefficients.
3    cOut:= asyncSerOut(16,0);   -- Convolution output.
4    (*  Storage for the input signal...  *)
5    s1, s2, s3, s4:= 0, 0, 0, 0;
6    (*  Storage for the kernel coefficients...  *)
7    k1, k2, k3, k4:= 0, 0, 0, 0;
8    DO
```

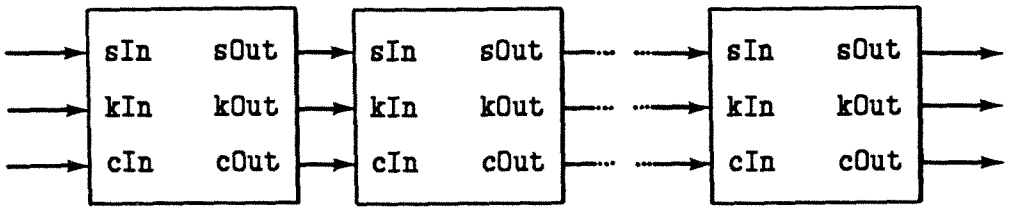


Figure C–2. A pipeline of convolution chips. **sIn**, **kIn**, and **cIn** represent the signal, kernel, and carry inputs, while **sOut**, **kOut**, and **cOut** represent the corresponding outputs.

```
 9      IF kIn.ready THEN  -- Shift in a kernel value.
10        k1, k2, k3, k4:= kIn.get, k1, k2, k3
11      EF sIn.ready THEN  -- Shift in a new sample.
12        { s1, s2, s3, s4:= sIn.get, s1, s2, s3;
13          cOut.put(s1*k1 + s2*k2 + s3*k3 + s4*k4)
14        }
```

Notice that the main loop has been transformed into something more nearly resembling an idle loop. The construct **kIn.ready** returns the value **True** if a value may be obtained from the input port allocated for kernel coefficients. It returns **False** if there are no values waiting. In either case, it immediately returns so that the program may proceed. Thus, the idle loop continually checks for either a sample value or a new kernel coefficient. In the first case, it uses the sample value to compute the next element in the convolution. When a new kernel value becomes available, it is shifted into the kernel.

A third version of the convolution program allows several chips to be cascaded. This feature would useful if the convolution kernel contained more coefficients than could be fabricated on a single integrated circuit. The new version of the program is basically the same as the prior one, but sample values and kernel values must be shifted out so that they are available to the next chip in the sequence. In addition, the partial sums of the convolution must be passed along so that they may be reflected in the final result. The diagram in Figure C–2 shows how several convolution chips might be cascaded. The final result is available on the **cOut** port of the last chip. Notice also that the **cIn** port of the initial chip must be supplied with a constant stream of zeroes, or whatever constant offset is desired. Here is the program:

```
 1    sIn, sOut:= asyncSerIn(16,0), asyncSerOut(16,0);
 2    kIn, kOut:= asyncSerIn(16,0), asyncSerOut(16,0);
 3    cIn, cOut:= asyncSerIn(16,0), asyncSerOut(16,0);
 4    s1, s2, s3, s4:= 0, 0, 0, 0;
 5    k1, k2, k3, k4:= 0, 0, 0, 0;
 6    DO
 7      IF kIn.ready THEN
 8        { kOut.put(k4);
 9          k1, k2, k3, k4:= kIn.get, k1, k2, k3
10        }
```

```
11      EF sIn.ready THEN
12        { sOut.put(s4);
13          s1, s2, s3, s4:= sIn.get, s1, s2, s3;
14          cOut.put(cIn.get + s1*k1 + s2*k2
15                           + s3*k3 + s4*k4)
16        }
```

C.5 Example: Self-Sorting Memory

A self-sorting memory is a device that stores numbers presented to it [ARMS77]. When these numbers are retrieved, they appear in order of decreasing numerical value. The version of the self-sorting memory presented here is a particularly simple one with no added frills. As shown in the diagram of Figure C–3, each cell is capable of storing a single value, and every cell has a bidirectional link to each of its two neighbors.

Numbers to be sorted are presented to the topmost sorting cell, which then compares the new value to its previously stored value. It passes the smaller of these two values to its lower neighbor. Because all cells behave in the same way, larger numbers end up near the top of the memory, while smaller ones sink to the bottom. If the memory becomes full, the smallest values will fall from the bottom of the chain and will be lost.

The environment of the memory retrieves values from the top cell. As each cell gives up its value, it obtains a new value from its lower neighbor. Notice that the bottom of the memory must be supplied with an arbitrary number of very small values. This particular version of the self-sorting memory does not detect overflow or underflow, although modifications could be made to handle these conditions. The code for a single memory cell is as follows.

```
1   MACRO sortCell(topIn,topOut,botIn,botOut)
2     BEGIN
3       myValue:= topIn.get;
4       DO
5         IF topIn.ready THEN {
6           new:= topIn.get;
7           IF new > myValue THEN myValue,new:= new,myValue;
8           botOut.put(new)
9         }
10        EF topOut.waiting THEN {
11          topOut.put(myValue);
12          myValue:= botIn.get
13        }
14    END
```

The **waiting** attribute in Line 10 returns **True** if the process on the other end of the communications port is expecting to receive a value through that port.

The definition of **sortCell** expects to send values to and receive values from neighboring cells over communications ports. Thus, the definition might be invoked

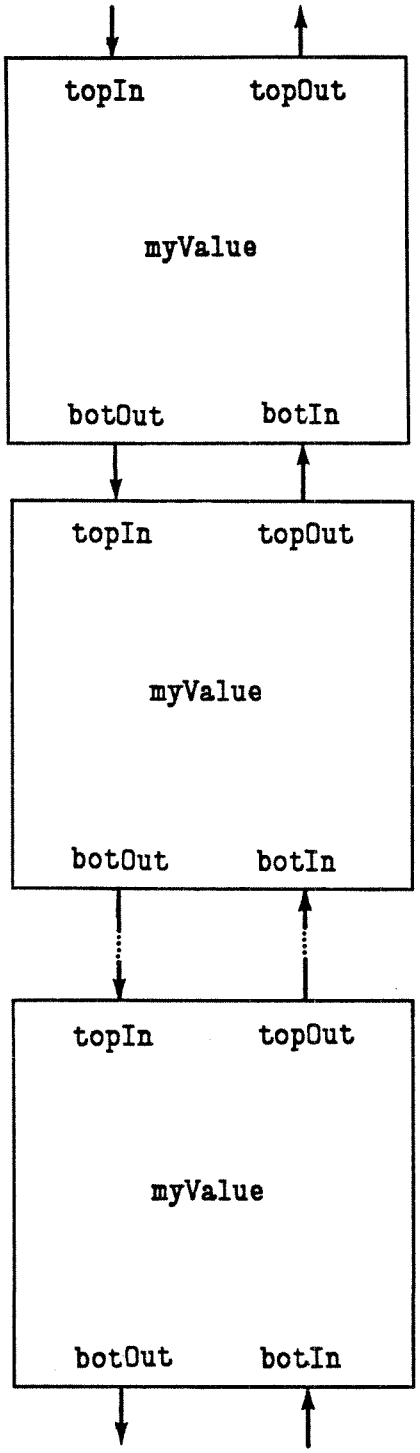


Figure C–3. A set of chips implementing a self-sorting memory. **topIn** is a port for accepting values to be sorted, **botOut** passes overflow values to the rest of the memory, **topOut** produces sorted values, and **botIn** retrieves sorted values from the rest of the memory.

by the following code.

```
sortCell(asyncSerIn(16,0),asyncSerOut(16,0),
         asyncSerIn(16,0),asyncSerOut(16,0))
```

This statement would create a chip containing just one sorting cell. Since the cells are probably fairly small, however, it would be nice to fit several of them on a single chip.

Internal communications ports may be used to allow independent parts of a program to communicate within a single chip. These ports are like other communications ports, except that both ends are available to the program, and no pads are generated in the layout. Internal ports are created with a statement like this:

```
i,o:= asyncInt(16,0)
```

This statement creates an internal port capable of passing sixteen it values and having no internal buffering. Notice that two port values are returned. One of them is for doing input, while the other does output.

The following program uses internal ports to describe a chip that consists of two sorting cells.

```
1   i1, o1:= asyncInt(16,0);
2   i2, o2:= asyncInt(16,0);
3   sortCell(asyncSerIn(16,0),asyncSerOut(16,0),i2,o1);
4   sortCell(i1,o2,asyncSerIn(16,0),asyncSerOut(16,0));
```

The two cells are connected by two internal ports, one for communicating in either direction. Since the attributes of each end of an internal port are the same as those for the corresponding external port, the definition of a sorting cell works equally well for either kind of port.

Now the program above is adequate for a memory consisting of just two cells, but would become a bit tedious if extended for a hundred-cell memory. The following definition might be used to describe a larger memory. It accepts the number of cells to create, along with an input port and an output port for accessing the top cell.

```
1   MACRO sorter(n,topIn,topOut)
2     IF n > 1 THEN {
3       newTopIn,botOut:= asyncInt(16,0);
4       botIn,newTopOut:= asyncInt(16,0);
5       sortCell(topIn,topOut,botIn,botOut);
6       sorter(n-1,newTopIn,newTopOut)
7     }
8     ELSE sortCell(topIn,topOut,
9          asyncSerIn(16,0),asyncSerOut(16,0))
```

This is a recursive definition that creates one sorting cell per invocation. It creates internal ports to connect at the bottoms of all but the final sorting cell, which is connected directly to an external port. With this definition, a complete sorting chip

consisting of twenty-five cells could be described by the following single statement:

```
sorter(25,asyncSerIn(16,0),asyncSerOut(16,0))
```

The recursive definition given above worked only because the depth of recursion was controlled by a constant expression. Clearly, the compiler must be able to determine how many sorting cells to create, since they cannot be created dynamically in the final chip. The constant 25 is explicitly mentioned in the definition invocation, permitting the compiler to determine the value of n within the definition. Armed with this, it can evaluate the condition and choose the appropriate clause of the IF-statement. Thus, the compiler will not generate code for conditionals if it can test the condition at compile time. This makes for a kind of conditional compilation.

Notice that the trick for iterating sorting cells may be applied to the convolution example presented earlier. Recall that the problem was to write a program that could produce a convolution chip for a kernel of arbitrary size. As the program was written, the only way to do this was to manually change the number of variables present in the program. It is now apparent, however, that the mechanism shown there for connecting several chips could serve equally well to connect parts of a single chip if internal ports were used. Such a scheme would certainly work, but it would not be very efficient.

It should come as no surprise that using internal ports will cost more in terms of area in the final layout than not using them. It seems, therefore, that they would have to offer some potential benefit before one would consider using them. In the case of the self-sorting memory, internal ports were used to advantage because they allowed cells to operate in parallel. The top cell might receive a new value before another value had trickled all the way to the bottom. This would not be possible if the program had been written sequentially. For the convolution example, however, no real concurrency could be gained by adding internal ports. Even as written, the necessary multiplications may be done in parallel, and several of the additions can proceed concurrently. Also, even if the convolution were broken up using internal ports, the partial sum would have to propagate all the way through the device before the sample values could be shifted.

The conclusion to be drawn from all of this is that the programmer must be aware of what he is doing. Even though the compiler will, at least in theory, produce a correct implementation for any valid program, no optimizing compiler can repair a poorly designed algorithm.

C.6 Example: Two-Dimensional Graphics

The final example of Silicon shows how simple pieces can be fit together to form complete chips. Some basic tools for two-dimensional graphics, including coordinate transformation, clipping, and line rasterization, will be used for illustration. The fundamental idea is shown in the diagram of Figure C–4. The program will process

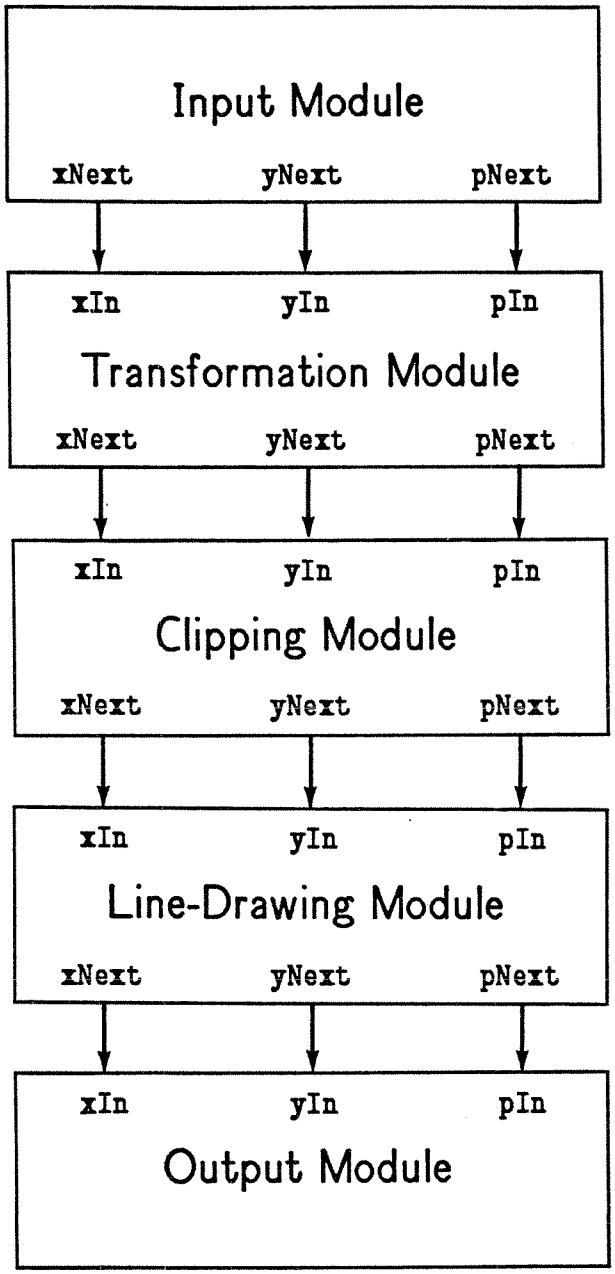


Figure C–4. Basic two-dimensional graphics pipeline consisting of transformation, clipping, and line-drawing modules. Ports whose names start with **x** and **y** pass coordinates from one module to the next, while those starting with **p** pass parameters that define the behavior of the modules.

line segments represented as two successive coordinates passed through the **x** and **y** ports. The other port, labeled **p** in the diagram, will be used for passing parameters to the clipper and transformer. These constants will be shifted through each of the pieces, just as values were shifted between chips in the convolution example.

Recall that a point (x, y) can be transformed in homogeneous coordinates as follows.

$$[\,x \quad y \quad 1\,]\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix} = [\,(ax+by+c) \quad (dx+ey+f) \quad 1\,]$$

This formulation allows for rotation, translation, and scaling. Also, recall that the endpoints of line segments transform independently, so that the transformation program need not consider points in pairs. With this in mind, here is the code for performing the transformation:

```
1    MACRO transform(xIn,yIn,pIn)=>(xNext,yNext,pNext)
2      BEGIN
3        xNext, xOut:= asyncInt(16,0);
4        yNext, yOut:= asyncInt(16,0);
5        pNext, pOut:= asyncInt(16,0);
6        a, b, c, d, e, f:= 1, 0, 0, 0, 1, 0;
7        DO
8          IF pIn.ready THEN
9            { pOut.put(f);
10             a, b, c, d, e, f:= pIn.get, a, b, c, d, e
11           }
12         EF xIn.ready AND yIn.ready THEN
13           { x, y:= xIn.get, yIn.get;
14             xOut.put(a*x + b*y + c);
15             yOut.put(d*x + e*y + f)
16           }
17     END;
```

The construct **EF** in Line 12 is an abbreviated notation for the sequence **ELSE IF**.

The declaration of **transform** illustrates the conventions that will be followed by the graphics modules. They will each have three input parameters and three output, or result, parameters. The input parameters are input ports that supply points and constants to the module. The result parameters are also input ports. Other modules can use them to retrieve points and constants that have passed through the module.

The next module, which performs clipping, is divided into four submodules. Each clips the line segment against one of the edges of the clipping area. By the time a line segment has passed through all four modules it will have been completely clipped. The edge clippers use a little trick so that they can all be described by the same code. As line segments are sent from one module to the next, they will be given a quarter turn to the left. This allows each module to behave as though it were clipping against the left edge. After passing through all four modules, a line segment will have rotated back to its original position. Here is the code for the edge clipper:

```
18   MACRO clipEdge(xIn,yIn,pIn)=>(xNext,yNext,pNext)
```

```
19      BEGIN
20        xNext, xOut:= asyncInt(16,0);
21        yNext, yOut:= asyncInt(16,0);
22        pNext, pOut:= asyncInt(16,0);
23        xEdge:= 0;
24        DO
25          IF pIn.ready THEN
26            { pOut.put(xEdge); xEdge:= pIn.get }
27          EF xIn.ready AND yIn.ready THEN
28            { x1, y1:= xIn.get, yIn.get;
29              x2, y2:= xIn.get, yIn.get;
30              v1:= (x1 >= xLeft);
31              v2:= (x2 >= xLeft);
32              IF v1 AND NOT v2 THEN
33                { x1, x2:= x2, x1;
34                  y1, y2:= y2, y1;
35                  v1, v2:= v2, v1
36                };
37              IF v2 AND NOT v1 THEN
38                x1, y1:= xLeft,
39                  yIntercept(x1-xLeft, y1, x2-xLeft, y2);
40              IF v1 OR v2 THEN
41                { xOut.put(-y1); yOut.put(x1);
42                  xOut.put(-y2); yOut.put(x2)
43                }
44            }
45    END;
46
47    MACRO yIntercept(x1, y1, x2, y2)=>(yInt)
48      REPEAT
49        xInt:= (1#1'x1 + 0#1'x2)[15:0];
50        yInt:= (1#1'y1 + 0#1'y2)[15:0];
51        IF xInt < 0 THEN x1, y1:= xInt, yInt;
52        IF xInt > 0 THEN x2, y2:= xInt, yInt
53      UNTIL xInt = 0;
```

Notice that the code for computing the actual edge intersection has been broken out into a separate definition. It uses the midpoint subdivision algorithm in order to avoid doing an explicit division operation [SPRO68].

Four edge clippers are joined in the following definition:

```
54    MACRO clipper(xIn,yIn,pIn)=>(xNext,yNext,pNext)
55      xNext, yNext, pNext:=
56        clipEdge(
57          clipEdge(
58            clipEdge(
59              clipEdge(xIn,yIn,pIn))));
```

This functional style of notation is useful for joining the various elements of a pipeline.

The line drawing module uses Bresenham's Algorithm [BRES65]. Pairs of input coordinates are interpreted as the endpoints of line segments, and the output coordinates are the points that lie on the line segment. Here is the code:

```
60    MACRO abs(n)=>(n) IF n < 0 THEN n:= -n;
61
62    MACRO twice(n)=>(n) n:= n+n;
```

```
63
64    MACRO lineDraw(xIn,yIn,pIn)=>(xNext,yNext,pNext)
65      BEGIN
66        xNext, xOut:= asyncInt(16,0);
67        yNext, yOut:= asyncInt(16,0);
68        pNext, pOut:= asyncInt(16,0);
69        DO pOut.put(pIn.get);
70        DO
71          { x1, y1:= xIn.get, yIn.get;
72            x2, y2:= xIn.get, yIn.get;
73
74            dx:= x2 - x1;  absx:= abs(dx)
75            dy:= y2 - y1;  absy:= abs(dy)
76
77            CASE (dx < 0)'(dy < 0)'(absx-absy < 0) OF
78            0:  -- Octant 1
79              { da:= absx; db:= absy;
80                m1x:= +1;  m1y:=  0;
81                m2x:= +1;  m2y:= +1
82              }
83            1:  -- Octant 2
84              { da:= absy; db:= absx;
85                m1x:=  0;  m1y:= +1;
86                m2x:= +1;  m2y:= +1
87              }
88            2:  -- Octant 8
89              { da:= absx; db:= absy;
90                m1x:= +1;  m1y:=  0;
91                m2x:= +1;  m2y:= -1
92              }
93            3:  -- Octant 7
94              { da:= absy; db:= absx;
95                m1x:=  0;  m1y:= -1;
96                m2x:= +1;  m2y:= -1
97              }
98            4:  -- Octant 4
99              { da:= absx; db:= absy;
100               m1x:= -1;  m1y:=  0;
101               m2x:= -1;  m2y:= +1
102             }
103           5:  -- Octant 3
104             { da:= absy; db:= absx;
105               m1x:=  0;  m1y:= +1;
106               m2x:= -1;  m2y:= +1
107             }
108           6:  -- Octant 5
109             { da:= absx; db:= absy;
110               m1x:= -1;  m1y:=  0;
111               m2x:= -1;  m2y:= -1
112             }
113           7:  -- Octant 6
114             { da:= absy; db:= absx;
115               m1x:=  0;  m1y:= -1;
116               m2x:= -1;  m2y:= -1
117             }
118
119           d:= twice(db) - da;
120           x:= x1;  y:= y1;
```

```
121             DO
122               { xOut.put(x); yOut.put(y);
123                 IF (x = x2) AND (y = y2) THEN BREAK;
124                 IF d < 0 THEN
125                   { x,y:= x + m1x, y + my1;
126                     d:= d + twice(db)
127                   }
128                 ELSE
129                   { x,y:= x + m2x, y + m2y;
130                     d:= d + twice(db) - twice(da)
131                   }
132               }
133           }
134       END;
```

Two more modules are needed to complete the program. These communicate off-chip to get input line segments and send output points. The code for the two modules is pretty simple.

```
135     MACRO input=>(xNext,yNext,pNext)
136       BEGIN
137         xNext:= asyncSerIn(16,0);
138         yNext:= asyncSerIn(16,0);
139         pNext:= asyncSerIn(16,0)
140       END;
141
142     MACRO output(xIn,yIn,pIn)
143       BEGIN
144         xOut:= asyncSerOut(16,0);
145         yOut:= asyncSerOut(16,0);
146         pOut:= asyncSerOut(16,0);
147         DO xOut.put(xIn.get);
148         DO xOut.put(xIn.get);
149         DO xOut.put(xIn.get)
150       END;
```

The input definition simply supplies the three external input ports. The output definition creates three output ports and then continually copies values to them.

With all of the tools in place, the program for the complete chip is quite trivial:

```
151     output(lineDraw(clip(transform(input))))
```

C.7 Extensions

In its current form, Silicon lacks some features that would be useful for programming certain applications and essential for others. These features were omitted from the first version of Silicon because of a desire to restrict the scope of the original implementation. For the most part, however, they fit well into the existing framework of the language. This section lists and briefly describes some of the possible extensions.

One area in which there is substantial room for improvement concerns the interactions between a Silicon program and its operating environment. In particular, it

would be useful to develop a wider range of communications ports. Each different kind of port might have differing electrical properties, or it might be functionally well suited to a particular range of applications. As ports become more specific, the Silicon code for dealing with them can be simplified somewhat. Since more of the control operations would be handled directly by the port, it might be reasonable to expect that a better overall chip implementation would result.

An example of a more specialized communications port is a signalling port, which can be used to implement handshaking protocols. Signalling ports would be implemented with a single pad that could either be sampled or driven, depending upon whether the port was for input or output. These ports would be used for sending or receiving transitions. That is, an output signalling port would changes its state on command, while an input signalling port would wait until it had detected a state change. For example, the following program fragment uses two signalling ports and a parallel input port to read a number. A two-phase signalling convention is used for the synchronization.

```
1    signalIn.get;
2    n:= inputPort.wait(signalIn).get;
3    signalOut.wait(inputPort).put;
```

The code in Line 1 waits for a signal that the value is ready. The next two lines sample the value and signal the fact that the value has been accepted. Notice that this program could have been written with simple parallel ports, but it would have been substantially more complicated. The `wait` operations, for example, would have had to be performed by a loop that continually sampled an input value. In addition to being more awkward to code, this scheme would have resulted in a far less efficient implementation than could be obtained with signalling ports.

Another useful extension to Silicon would be the addition of explicit memories. Recall that the Silicon compiler generates storage wherever it is required and that this operation is practically invisible to the programmer. Sometimes, however, it is more natural for the programmer to deal with explicitly defined memories. For example, a register file and a table of constants can conveniently be thought of as a random access memory (RAM) and a read-only memory (ROM). Memories would be defined in the same way as communications ports. Thus, a memory consisting of eight words of sixteen bits apiece might be defined as follows.

```
reg:= RAM(8,16);
```

It is easy to imagine the definition of a dual-port memory, which might look something like this:

```
port1, port2:= dualPortRAM(8,16);
```

After defining the memory, values could be stored in and retrieved from it like this:

```
1    reg.store(address,newValue);
2    oldValue:= reg.fetch(address);
```

Of course, read-only memories would have only a `fetch` attribute. The constant values would have to be supplied during compilation.

Memories like those described above are rather similar in their behavior to communications ports. Notice that a particular memory can perform only one operation at a time, although two different memories may be busy simultaneously. Also, the sequence of memory operations must be preserved. For example, read and write operations on a memory must be performed in the order specified by the source code if the program is to work correctly. Entities like communications ports and memories are known as resources in Silicon. Later sections will show how similar their implementations can be.

Silicon would benefit greatly from the ability to include user-supplied operation and resource definitions. Sometimes a specific computation that is difficult to perform in Silicon is comparatively simple to implement with a custom-designed layout. It would be convenient for the programmer to reference these lower level modules from a Silicon program. Such a scheme allows a designer to derive some of the benefit of a custom design, while at the same time having the compiler do much of the detail work. These are the same reasons offered for the use of assembly language subroutines with higher level languages. The syntax for invoking user-supplied operations and resources would be the same as that for invoking definitions, and it would be necessary to design the compiler in a way that made it easy for the designer to define his additions. This in itself is a fairly major undertaking.

Finally, there appears to be some need in Silicon for higher level data structures, like arrays. Notice that arrays are quite different from RAMs. For the latter, it is the programmer who manages the memory. In the former case, the compiler must not only manage, but also allocate, the storage used by the array elements. Another problem is that it is not even obvious how arrays should behave. Questions like these are perhaps best left unanswered until more experience with Silicon has been accumulated.

C.8 Implementation Overview

It is fairly obvious that programs written in Silicon are technology independent. That is, there is nothing in the language that would, for example, favor CMOS over nMOS. On the other hand, it may not be so clear that programs are also representation independent. Although numbers are represented as bit strings of known length, nothing specifies the representation of those bits, or of the devices that manipulate them. For example, a sixteen-bit number might be communicated on sixteen separate wires or on a single wire that contains successive bits of the number at sixteen successive clock times. Between these extremes of fully parallel and fully bit-serial, byte-serial and nibble-serial representations are possible.

It might be argued that the use of, let us say, a serial communications port would restrict a program to a serial implementation, but this is not so, because a port only

determines the interaction of a chip with its environment; it does not constrain the internal workings of the chip. Notice, however, that although transparent to the Silicon programmer, the implementation of a communications port will have to be tailored to the main representation being used. For example, a serial port used on a parallel chip must include some mechanism for converting between the two formats.

In addition to being independent of the spatial aspect of numeric representation, Silicon does not depend upon any particular timing strategy. Although clocked, synchronous chips will be of the most interest for this discussion, other protocols are possible. For example, the interior of a chip might follow the conventions for self-timed systems [SEIT80]. Again, such chips could range anywhere from completely serial to fully parallel.

The reason for Silicon's representation independence is that operations in the language specify a function to be performed rather than a device for performing that function. For example, a plus sign specifies an addition operation; it does not declare that an adder should be generated. On the contrary, several addition operations may be implemented with the same adder, or several adders may be required to implement an addition specified by a single plus sign. Furthermore, addition operations and subtraction operations might even be implemented with the same device, if this were convenient in the representation being used. All of these considerations are below the level of detail that is visible in the Silicon source code.

Although Silicon programs are representation independent in the purest sense, there are some difficulties in the engineering sense. The problem is that a chip implemented with a fully parallel representation, for example, might not have an equally efficient implementation if a bit-serial representation were used. The reverse might also be true. This should come as no surprise, just as it is no great shock that not every program is a good program. Programmers have long known that a more efficient program can result if the characteristics of the programming language implementation are taken into account. What this means for Silicon is that it may not be desirable or even feasible to implement a particular program in every possible representation.

The representation independence of Silicon programs does have some advantages, however. First, a functional simulator that works at the source code level will work for any representation. Second, it turns out that about half of the code for the compiler itself does not depend upon the target representation; this should help to encourage the production of new versions of the compiler. Third, notice that there is nothing forcing every part of the chip to use the same representation; different parts of the same chip may be implemented differently.

The remainder of this section is devoted to an overview of the structure of the Silicon compiler. There are four relatively distinct phases of the compilation process, each of which is briefly described below. Later sections will treat each aspect of the compiler in greater detail.

The first phase of the compiler performs a lexical and syntactic analysis of the

incoming Silicon source code. The program is tokenized and parsed by a simple set of recursive descent procedures that transform it into a reverse polish string. Definitions are also extracted, and one of the first phase results is a symbol table that associates definition names with the reverse polish representation of the definition body. The actions of the first phase are all pretty straightforward and will not be described further.

The second phase performs a dataflow analysis of the program using the polish strings generated in the first phase. The dataflow analysis is used in later phases to determine which operations specified in the source code can be executed at the same time. Phase two is also where constant expressions are evaluated, definition invocations are processed, and constant sizes are determined. The output of the second phase is a directed graph representation of the program, called a dataflow diagram. It will be described more completely later on, but for now it suffices to realize that the dataflow diagram is a convenient representation of the program for use in subsequent phases of the compiler.

The third phase of the Silicon compiler applies function preserving transformations to the dataflow diagram in an attempt to somehow improve it. Notice that unlike the operations of the previous phases, this optimization process depends very heavily on the data representation being used. For example, in a parallel, bus structured representation, implementing two addition operations with a single adder device might require the simple alteration of a PLA. To do this with serial devices, however, would require extra wires for routing the operands, in addition to switching and logic circuitry for choosing between the two sets of operands. Thus, transformations that are optimizations for one representation may have just the opposite effect for other representations. In fact, this observation leads to one of the main reasons that it is possible to compile a single Silicon program into multiple representations: The dataflow diagram is a sufficiently neutral form that it may be stretched in any one of a number of directions.

The fourth and final phase of the compiler resembles a conventional "chip assembler." It accepts the list of devices, along with the interconnection and timing information that was produced by the third phase, and generates the final layout. This is just a matter of placing standard cells and tailoring them to suit their surroundings. It seems pretty clear that this phase is not only representation dependent, but also technology dependent.

C.9 Dataflow Analysis

Dataflow analysis is really the heart of the Silicon compiler. It produces a dataflow diagram, which is a convenient form for later optimization. The dataflow diagram is also a halfway point in the generation of a layout. Serial versions of the compiler use it as a starting point for the final geometry, while parallel versions use it to locate

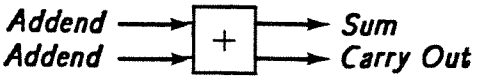


Figure C–5. A dataflow diagram node that will be created for the addition operation.

opportunities for parallelism in the program implementation. This section describes the process of dataflow analysis in reasonably great detail, but it stops short of delving into the programming details.

The process of dataflow analysis is basically a symbolic execution of the Silicon program. The reverse polish strings produced by the parsing phase are interpreted sequentially, but instead of actually performing the computations specified in the polish string, the interpreter builds a data structure that represents the computations. This data structure, called a dataflow diagram, is a directed, possibly cyclic graph where the nodes represent operations and the arcs linking the nodes represent values.

The way in which dataflow analysis is performed is similar to the implementation of type checking in conventional programming languages. In fact, the compiler performs a sort of elementary type checking as it generates the dataflow diagram. The compiler insures, for example, that the operands of arithmetic operators are indeed numeric, that communications ports are used properly, and that the correct numbers of parameters are supplied for each definition invocation. It can also detect such errors as a reference to an uninitialized variable.

Addition is an example of one of Silicon's simple operations. Its representation in a dataflow diagram is shown in Figure C–5. As illustrated, addition is drawn as a node having two input arcs for the two addends and two output arcs for the sum and carry results. For this example, neither the input arcs nor the output arcs are connected to anything, but in the actual dataflow diagram the input arcs of every node will be connected to the output arcs of some other nodes. Not all output arcs need be connected, of course, because it is not strictly necessary to use the result of every operation. Also, a single output arc might be connected to more than one input arc, corresponding to the case where a single result is used several times. For example, the addition of three numbers might be represented as in the fragment of a dataflow diagram that is given in Figure C–6.

Notice how similar dataflow diagrams are to the notion of a dataflow machine. It is often useful, in fact, to think of the dataflow diagrams as a kind of idealized dataflow machine. Values may be thought of as flowing along the arcs in little capsules. When an operation receives all of its inputs, it produces a result capsule. If an output is

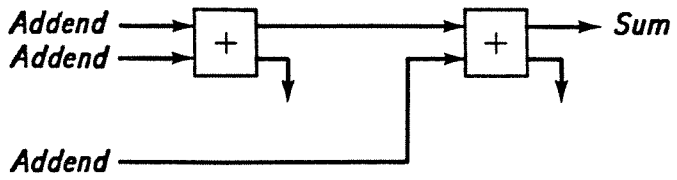


Figure C–6. A fragment of a dataflow diagram that will be generated to represent the addition of three numbers.

connected to more than one input, the capsule effortlessly replicates itself as often as necessary. Notice that this ideal dataflow model may or may not have anything to do with the implementation of the final chip. Whether it does will depend to a large extent upon the actual representation being used.

The program for generating the dataflow diagram makes use of two main data structures. First, a symbol table associates values with variable names. The values are arcs in the dataflow diagram under construction. The second data structure consists of a simple stack used to store intermediate values during expression evaluation.

A simple example will help to illustrate how the stack and the symbol table can be used to construct a dataflow diagram. Consider the following Silicon expression.

```
a*a + b*b
```

Recall that the parsing phase of the compiler will translate this expression into a reverse polish string:

```
a a * b b * +
```

Suppose that the values of the two variables `a` and `b` are known and therefore appear in the symbol table. When the interpreter encounters the variable names in the polish string it will thus be able to push their values onto the stack. After execution of the first two tokens, the stack will look like it does in Figure C–7*b*. When it executes the * in the polish string, the usual thing for a polish interpreter to do would be to actually execute the operation. Here, however, the interpreter will pop two values from the stack and use them as input arcs to a newly constructed node in the dataflow diagram. It will then push the output arc of the new node onto the stack, which would look like Figure C–7*c*. The remaining stages of execution are shown in Figure C–7*d* through *g*. Notice that the final result on the top of the stack is an arc leading from the desired sum. The second output from the addition operation, the carry out, is not pushed onto the stack; instead, its value is inserted in the symbol table with the variable name `addSubCarryOut`.

It is easy to see how an assignment statement works in Silicon. Suppose that the following statement is being executed.

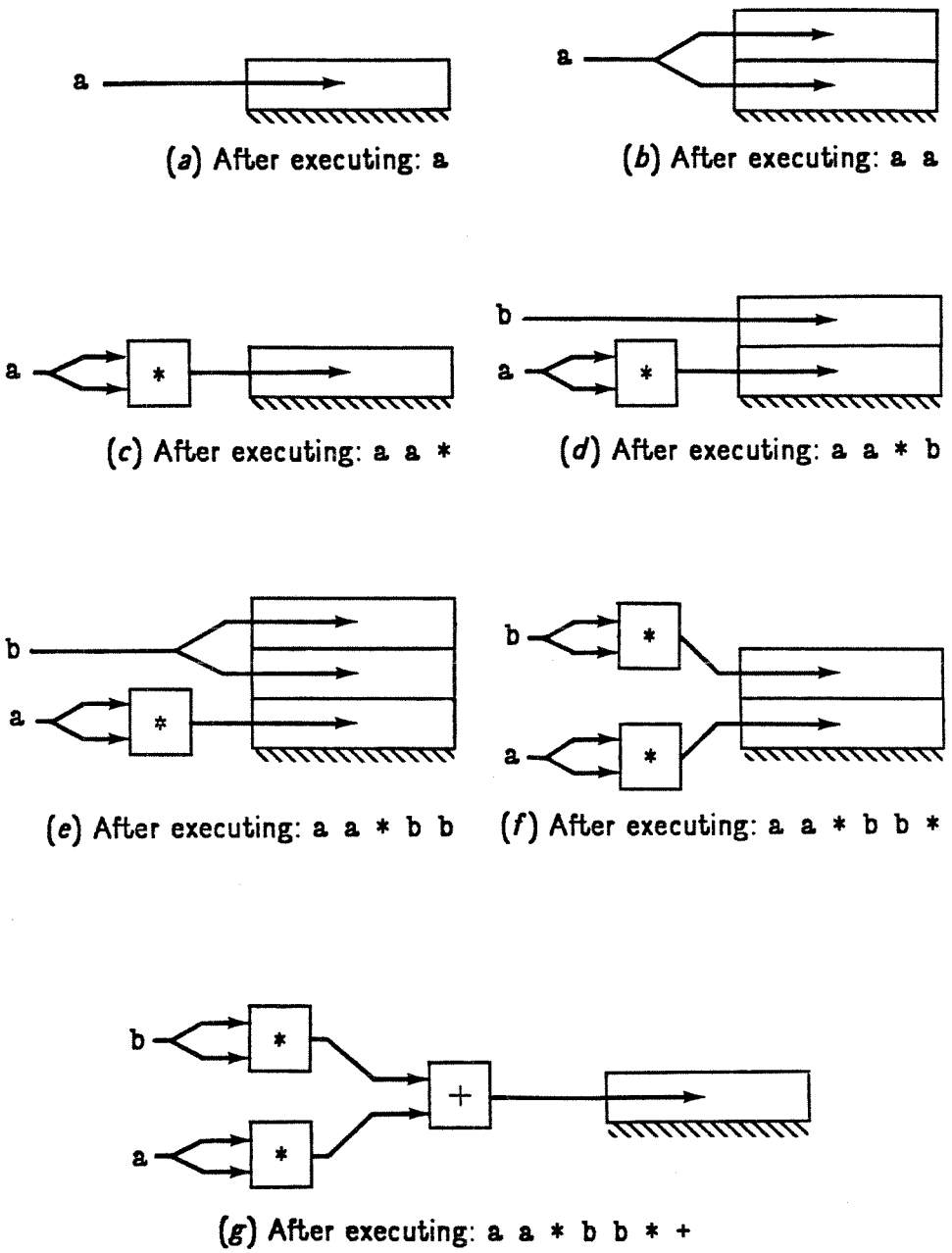


Figure C–7. State of the interpretation stack after each stage in the analysis of the polish expression a a * b b * +.

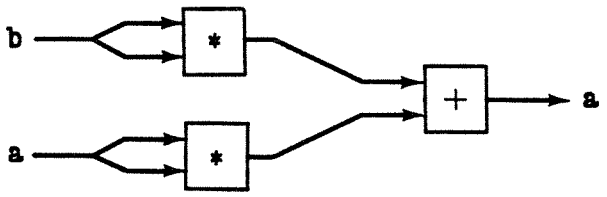


Figure C–8. Dataflow diagram resulting from the assignment statement a:= a*a + b*b.

```
a:= a*a + b*b;
```

The expression is evaluated as described above, leaving its value on the top of the stack. To implement the assignment statement, the interpreter simply pops this value off the stack and uses it to modify the symbol table entry for the variable **a**. The resulting fragment of the dataflow diagram is shown in Figure C–8. Notice that the variable **a** names different values at different stages of the dataflow analysis, and there is thus no single arc in the dataflow diagram that corresponds to a particular variable. In fact, after the dataflow analysis, the compiler has no further need for the variable names other than for use in error messages.

Macro expansion is another exercise in stack and symbol table manipulation. When the actual parameters of the macro are computed, they are left on the top of the stack. Next, if the macro references any global variables, these too are pushed onto the stack. At this point, both the symbol table and the current code stream are saved away on different stacks, to be replaced by a new symbol table and the code for the body of the macro. The formal parameter names and the global input variable names are then bound to the values on the top of the stack. After the code for the body of the macro has been executed, the reverse of this process takes place. The values of the output parameters are pushed onto the stack, followed by the current values of the global output variables. The symbol table and code stream are discarded, and the previous ones are revived. Finally, the values of the global output variables are stored in this original symbol table, while the other values returned from the macro remain on the stack, ready to be used in an expression.

As the previous discussion suggests, after a macro has been expanded, no indication remains in the dataflow diagram that it was ever there at all. This means that the compiler cannot make use of any hierarchical structure that the designer may have included in the Silicon source program. On the other hand, the actual silicon wafer is an inherently homogeneous material, and it is often the case that a hierarchical structure will be imposed more for the benefit of the designer than for the satisfaction of constraints imposed by the implementation. The programming language Silicon derives two benefits from its policy of discarding hierarchy information. First, the

Figure C–9. An **ifYes** operator is denoted by Y; **ifNo** and **Merge** operators are denoted by N and M.

programmer can choose a program structure that suits the application, rather than the compiler. Second, the compiler itself is free to apply optimizations across hierarchical boundaries in ways that may not have been apparent to the programmer.

Control structures are a little more difficult to implement than the simple expressions described above, but not unreasonably so. Implementing the conditional statement requires three dataflow operators that behave somewhat differently from the usual expression operators. These are the **ifYes**, **ifNo**, and **Merge** operators, shown in Figure C–9. The **ifYes** operator has a condition input and a value input. The condition input must have a value of zero or one, corresponding to **False** or **True**. When the **ifYes** operator receives both of its inputs, it either passes the value input along as its output or does nothing, depending on the value of the condition input. Thus, if the condition input is one, meaning **True**, the value is propagated through the **ifYes** operator. If the condition is zero, or **False**, the value is blocked. Notice that this behavior differs from that of the ordinary operators because there is no strict one-to-one correspondence between sets of inputs and sets of outputs. The **ifNo** operator, of course, passes its value input if the condition input is **False**. Finally, the **Merge** operator expects to receive a value on one or the other of its inputs, and it simply propagates this value along to its output.

Consider next the following conditional statement.

```
IF a > b THEN a, b:= a+b, a-b ELSE b:= a
```

Recall that the condition, **a>b** in this case, produces a one-bit value that specifies which of the two clauses should be executed. This bit is routed to the condition inputs of the **ifYes** and **ifNo** operators that screen the inputs to the portions of the dataflow diagram for each clause of the conditional. At the end of the conditional, **Merge** operators rejoin the final results from each clause. The complete dataflow diagram appears in Figure C–10. At first glance, the graph looks rather complicated, but it actually is not. Variable names have been placed next to the corresponding arcs in an attempt to make the diagram more readable. One interesting thing to note is that the assignment statement in the ELSE-clause causes a single value to be named by two variables. Another is that the output of the **ifNo** operator for the variable **b** is never

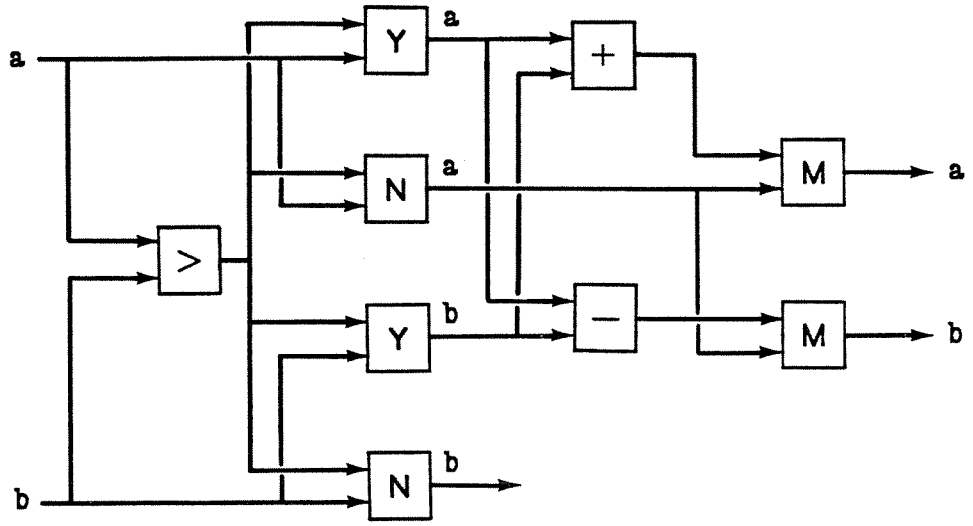


Figure C–10. Dataflow diagram resulting from the Silicon statement `IF a > b THEN a, b:= a+b, a-b ELSE b:= a.`

used, so it will eventually atrophy and disappear from the dataflow diagram.

As one might expect, generating the dataflow diagram for conditional statements requires some careful symbol table manipulation on the part of the interpreter. At the beginning of the conditional, it is necessary to generate a new symbol table for each clause. These must each contain all of the variables from the original symbol table, passed through the appropriate `ifYes` or `ifNo` operator. At the end of the statement, these symbol tables must be combined by passing corresponding entries through **Merge** operators.

It may seem that this scheme would generate a dataflow diagram that is substantially more complicated than necessary, especially if there were many variables not used in the conditional statement. This problem turns out to be relatively minor, as will be shown in a later section. An alternate technique for doing the dataflow analysis avoids the problem by adding an extra pass to determine what variables are used in the conditional statement. This method turns out to have difficulties that make it unsuitable for use here. This will also be discussed later.

The **CASE**-statement implementation is very similar to that of the conditional statement. It does, however, require the additional dataflow operator shown in Figure C–11. The **Case** operator has a single input that is used to determine which branch should be executed. Upon decoding the input value, it generates either a zero or a one on each of its outputs. These outputs are all connected to `ifYes` operators that control the inputs to the various alternatives of the **CASE**-statement. Thus, only those

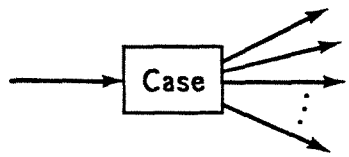


Figure C–11. The **Case** dataflow operator.

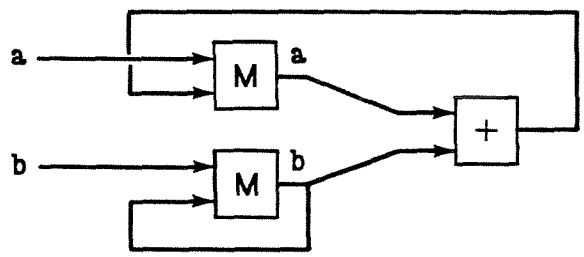


Figure C–12. Dataflow diagram resulting from the statement `DO a:= a+b`.

alternatives receiving a **True** value from the **Case** operator will be executed.

Looping statements use the same **Merge** operator that was described for conditional statements. For loops, however, one input to the **Merge** is the initial value of the loop and the other is the value from the previous iteration. For example, consider the following statement and the corresponding dataflow diagram given in Figure C–12.

```
DO a:= a+b;
```

Notice that the value of the variable **a** will never be completely computed, while the value of **b** never changes. This can be determined from the dataflow diagram by noting that the output of the **Merge** operator for the variable **b** is the same as its input. Again, some rather tricky symbol table manipulation is required to reflect this.

Loops that terminate are quite a bit more complicated than infinite loops, but they still use the operators that were described for conditional statements. Here is a fairly simple example that illustrates the general idea:

```
DO
  { a:= a + b;
    IF a > b THEN BREAK;
    b:= a - b
  }
```

The corresponding dataflow diagram appears in Figure C–13. The effect of the **BREAK**

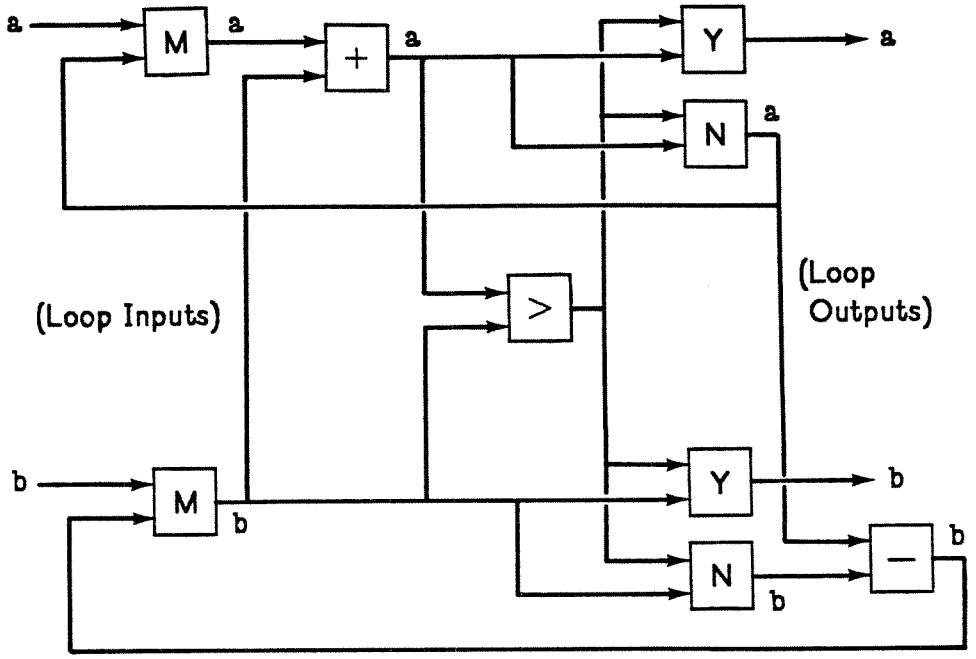


Figure C–13. Dataflow diagram corresponding to a loop that terminates.

statement is to pull all of the values in the symbol table to the outside of the loop. The values of the variables **a** and **b** will therefore be available after the loop when the condition **a>b** has been satisfied. When the condition has been satisfied and the loop terminates, the values are not only permitted to leave the loop, they are also prohibited from cycling around it any further. The mechanism for loop termination thus prepares the loop to be executed again, if it should it be enclosed within an outer loop.

The **WHILE** and **UNTIL** forms of loops are just special cases of the loop outlined above. For these, the loop termination test is either the first or the last statement of the loop body. A complication, however, is the case where a loop has several possible exit points. This would occur if there were more than one **BREAK** statement within a single loop. To implement this case, it is necessary to route the corresponding outputs from each exit point through **Merge** operators. This must be done, of course, for every variable, and it has the effect of joining the various operators that may be the last ones executed before leaving the loop.

Until now, no mention has been made of where the dataflow diagram is rooted. That is, every operator described so far has had inputs that must be supplied before an output could be produced. There is a single exception to this observation. The **Reset** operator, shown in Figure C–14, has no inputs. Instead, it produces its single output when the chip represented by the dataflow diagram is initialized. Recall that most operators produce encapsulated values as outputs. The **Reset** operator produces

Figure C–14. The **Reset** dataflow operator.

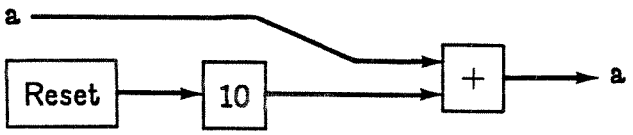


Figure C–15. Dataflow diagram corresponding to the statement **a:= a+10**, illustrating the use of constants.

an empty capsule; that is, there is no associated value. This means, for example, that its result cannot be used as the input of an adder because there is no value to add. On the other hand, passing the **Reset** value through an **ifYes**, **ifNo**, or **Merge** operator is perfectly legitimate, so that the value may be referenced from within loops and conditional statements. The **Reset** value is intended for use only by the compiler and possibly by the compiled layout, and it is therefore inaccessible from the Silicon source program.

The **Reset** value is used primarily for implementing constants. These are operators that produce a predetermined value upon receipt of some other value. This input value will be ignored, so that the use of the **Reset** value is reasonable. The next example illustrates the use of constant operators. Figure C–15 shows the dataflow diagram for the following statement.

```
a:= a + 10
```

Notice how triggering constant operators in this manner insures that the correct number of constant values will be generated. For example, if a constant operator is used within a loop, it will generate only one value per iteration because the **Reset** value cycles through the loop at that rate.

Communications ports provide another example of "valueless values." Recall that consecutive accesses to a communications port must not only be separated in time,

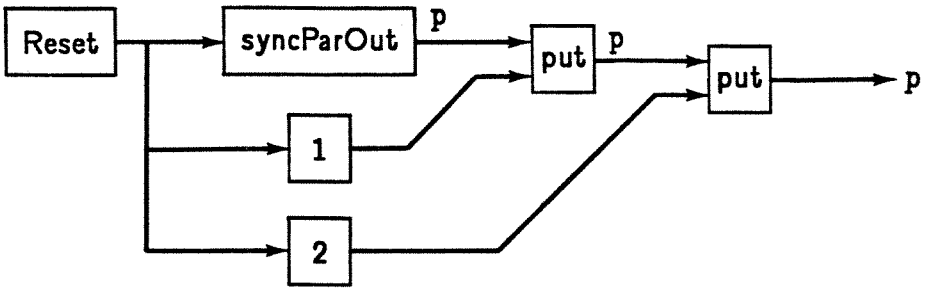


Figure C–16. Fragment of a dataflow diagram illustrating port creation.

they must occur in the same order as specified in the source program. Maintaining this order for, say, additions is no problem because the outputs of an adder will not appear until the inputs are ready. Furthermore, if two additions are independent of each other, the order in which they are performed doesn't really matter. In contrast, while the inputs of two **put** operations to the same port may become available in either order, the operations themselves must be performed as specified in the source program. This problem can be solved with the introduction of port values. Like the **Reset** value, these are values in form only. The **put** operation just mentioned would have two inputs: the value to be put and the port value. Its single output would be the port value, which would not be generated until the numeric value had been sent through the port. The **get** attribute of input ports would thus have a single input for the port value, one output for the value obtained from the port, and one output for the propagated port value.

An example of communications ports might help to clarify the description given above:

```
p:= syncParOut(16,0);
p.put(1);
p.put(2);
```

The dataflow diagram for this program is given in Figure C–16. The **syncParOut** operator creates a port value upon receipt of its input value. Notice that the port value can be assigned to other variables or used within loops and conditionals pretty much as any other value might be. The one restriction is that the compiler must be able to trace every path from a particular port access to the same port definition. This is because the compiler must be able to determine which port is being accessed so that it can run a wire there. Here is a program that violates this restriction:

```
p1:= syncParOut(16,0);
p2:= syncParOut(16,0);
-- The following statement is wrong.
(IF a > 0 THEN p1 ELSE p2).put(10);
```

Figure C–17 shows the corresponding dataflow diagram. The problem is that when the **port** input of the **put** operator is traced backwards to determine the port, the two paths lead to two different ports. If this situation is detected in a program, the

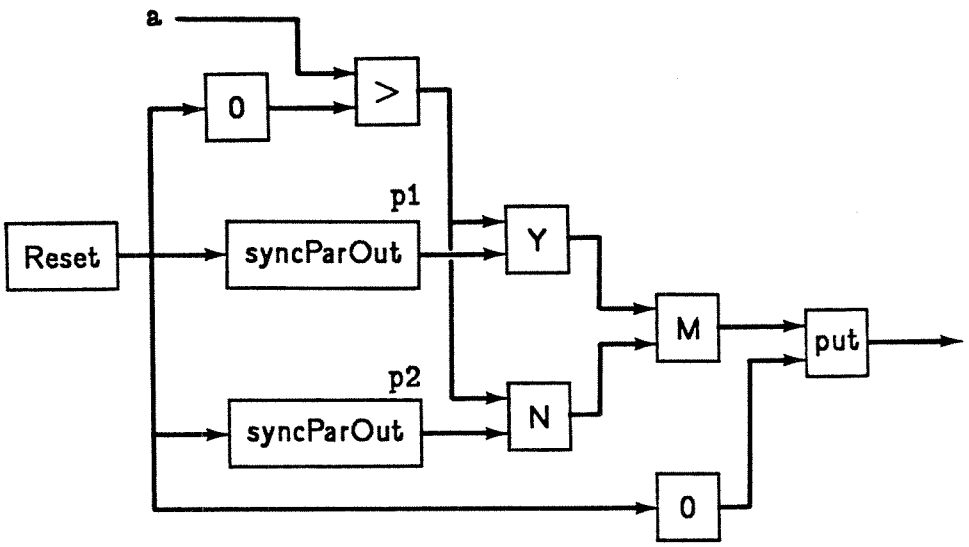


Figure C–17. Dataflow diagram corresponding to an incorrect use of ports. The compiler cannot determine whether the put operation is being applied to output port p1 or output port p2.

compiler will issue an error message.

Once the dataflow analysis is complete, the only handles that the compiler maintains on the dataflow diagram are through the communications port values. That is, variable names have disappeared, and only those operators that are accessible from the final port values will be considered as part of the dataflow diagram. Notice that these are the only operators that can possibly have any effect on the external behavior of the chip. Thus, at practically no cost in terms of compilation time, code that was included solely for the purpose of debugging will be rendered invisible. This code may be arbitrarily complicated, possibly even including definition invocations.

Although not explicitly stated, it is probably clear that concurrency information is readily available in the dataflow diagram. In order to determine whether one operation may be performed at the same time as another, it is necessary only to determine if there is a path in the dataflow diagram from the first operator to the second. The existence of such a path implies that the first operator depends on the output of the second. If no such dependencies exist, and if sufficient hardware is available, then the operations may be performed at the same time.

The process of tracing through the dataflow diagram in order to determine potential concurrencies works even for seemingly more complicated cases. Consider the following program in which two infinite loops cooperate by means of an internal communications port.

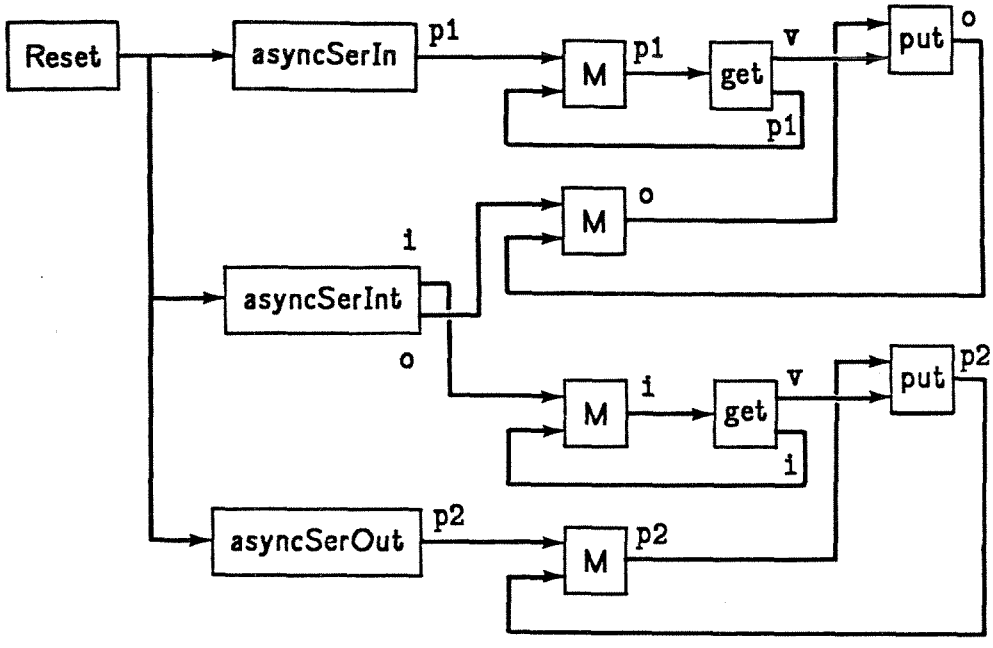


Figure C–18. Dataflow diagram illustrating a wire implemented with internal communication ports.

```
p1:= asyncSerIn(16,0);
p2:= asyncSerOut(16,0);
i,o:= asyncInt(16,0);
DO
  { v:= p1.get;
    o.put(v)
  };
DO
  { v:= i.get;
    p2.put(v)
  }
```

This is a very expensive implementation of a wire, but it does serve to illustrate some interesting points. The corresponding dataflow diagram is shown in Figure C–18. Notice that the invocation of **asyncInt** produces two ports that are independent with respect to the dataflow diagram. This makes it possible for the loops to be independent. Since the loops are independent, that is, no operator in one uses a result produced in the other, they may be executed in parallel. This is not, however, to say that they will run independently. The shared communications port will require some sort of synchronization to take place.

A few things have been left out of the foregoing discussion of the dataflow analysis phase of the Silicon compiler. These are mainly concerned with generating the simplest diagram that can accurately represent a given source program. Such matters are treated in the next section.

C.10 Folding

A dataflow diagram produced by the algorithm described in the preceding section will end up containing many artifacts that were simply not present in the original Silicon source program. For example, values may pass through many operators having to do with implementing loops, even though those values are not generated or even referenced from within a loop in the source code. This unfortunate state of affairs can be corrected by a technique called folding, which attempts to merge nearby operations in the dataflow diagram. In addition to this kind of remedial work, the folding process can evaluate constant expressions at compile time. A simple extension of this action implements Silicon's conditional compilation facility.

Although it may be thought of as a distinct operation, folding can be performed as part of the dataflow analysis. There are several reasons for doing so. First of all, folding reduces the size of the dataflow diagram, so that unneeded parts do not have to be carried through the whole compilation. Second, in order to perform conditional compilation, it is essential that the value of the condition be known early in the compilation process. This is especially true if the condition is being used to terminate a recursive definition. Improper handling of this case could cause the compiler to enter an endless loop. Finally, better error messages can be generated if folding is done as part of the dataflow analysis. This is so because the source code line numbers and variable names are available in the reverse polish code used by the dataflow analysis program, but this information is missing from the completed dataflow diagram.

The simplest case where folding can be applied occurs when every input to a particular operator in the dataflow diagram is a constant. For example, consider the fragment of the dataflow diagram produced by the Silicon expression **5+7**, which is shown in Figure C–19*a*. If the addition operation is performed at compile time, it and the two constants may be replaced by the constant result, as illustrated in Figure C–19*b*. This type of folding is useful primarily in situations like bit subscription, where the actual value must be known at compile time.

Another, somewhat more complicated folding operation detects values that pass through a conditional statement or expression without being modified. The general form of the dataflow diagram is shown in Figure C–20, before and after the folding has taken place. Recall that when the conditional statements or expressions are processed in the dataflow analysis, every variable in the symbol table is somehow passed through this configuration of **ifYes**, **ifNo**, and **Merge** operators. If most of these variables are not referenced within the body of the conditional, many extra nodes in the dataflow diagram will be created. Thus, the folding operation that eliminates these extra ones can effect a substantial savings.

Looping statements cause a similar problem. When a loop is encountered in a Silicon program, the dataflow analysis routine will pass through the loop every variable that exists at the time. For every variable that is not used within the loop, unnecessary nodes will appear in the dataflow diagram. Figure C–21 contains a fragment of a

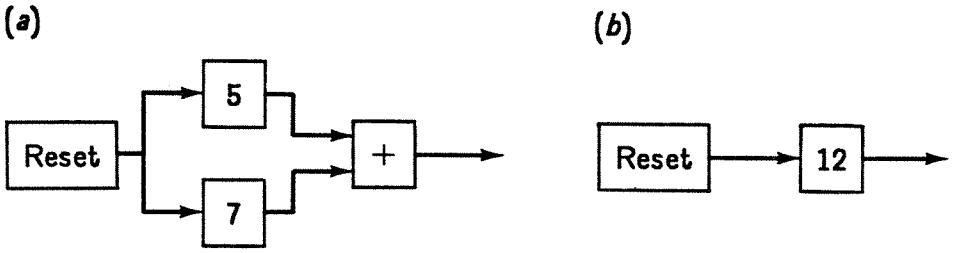


Figure C–19. Dataflow diagram corresponding to the expression 5+7 before and after folding.

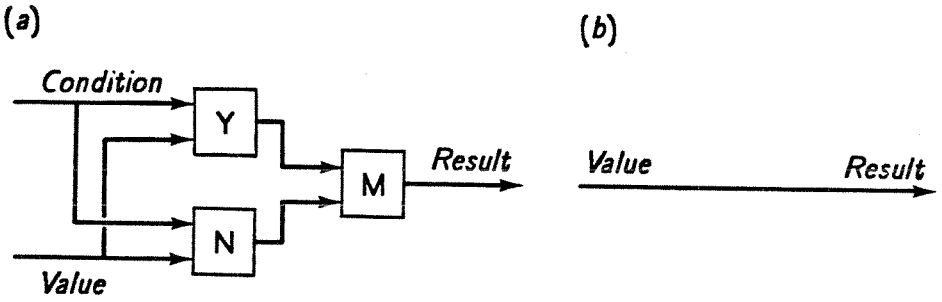


Figure C–20. Dataflow diagram before and after an unnecessary conditional operation has been eliminated by folding.

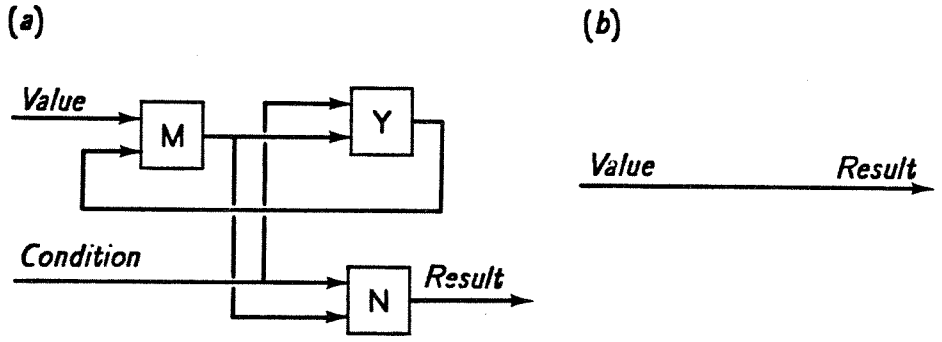


Figure C–21. Fragment of a dataflow diagram shown before and after an unnecessary loop has been folded away.

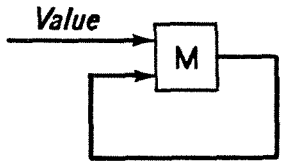


Figure C–22. Even values not referenced within an infinite loop are inaccessible afterwards if folding is not performed properly.

dataflow diagram shown before and after folding has been performed. Here, too, substantial savings will be derived if the body of the loop doesn't reference most of the variables in the program. Beyond this, loop folding can affect the timing of a program. Since a loop may execute an arbitrary number of times, an arbitrary delay may occur before the values are released. If a value is not computed within the loop, there is no reason for this delay, and so the folding operation produces a dataflow diagram that is, in some sense, more correct.

When infinite loops are not folded properly, it is actually possible for an erroneous dataflow diagram to be generated. Before folding, a value that is never referenced in the infinite loop will be trapped inside of it, as shown in Figure C–22. Notice that the loop has no outlet, so if the program would be unable to reference the value after the loop. It is clearly unreasonable for all of the variables to be unavailable after every infinite loop, and the folding operation that breaks invariants out of loops insures that this will not happen. Notice, however, that it is possible for infinite loops to consume values if variables are changed within the loop. In this case, since the loop is infinite, the computation of the values never finishes. Finally, this case provides another reason for performing folding in tandem with the dataflow analysis, since the code following the infinite loop cannot be processed until values not used in the loop have been made to bypass it.

It is sometimes possible to perform more folding if constants defined outside of loops or conditionals are pulled to the inside. The following excerpts of Silicon code, for example, illustrate the basic principle.

```
--  Before folding  --
n:= 10;  a:= 100;
DO
  IF a > n THEN a:= a-n ELSE a:= a-1
UNTIL a < 2*n

--  After folding  --
n:= 10; a:= 100;
DO
  IF a > 10 THEN a:= a-10 ELSE a:= a-1
UNTIL a < 20
```

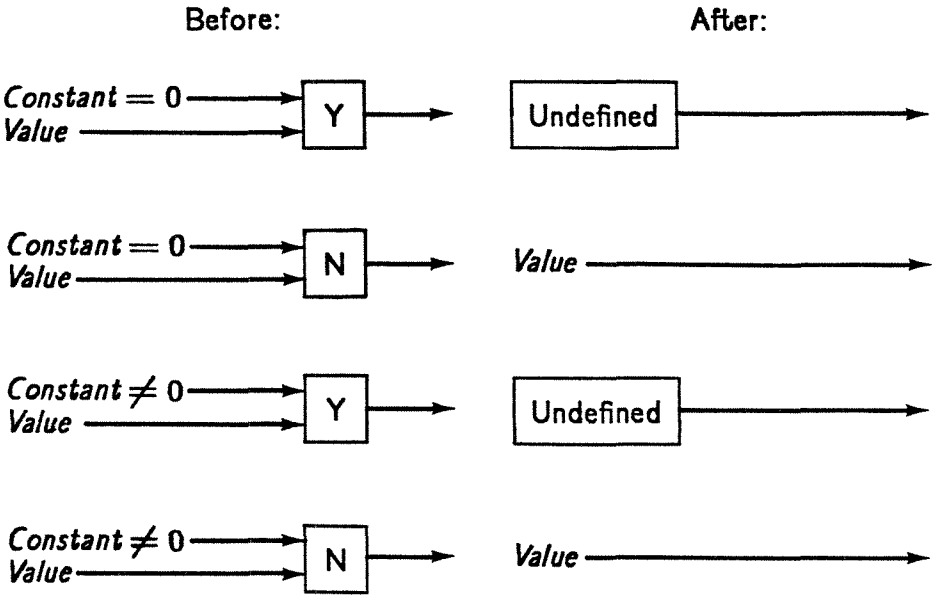


Figure C–23. Conditionals having a constant conditional input may be eliminated by folding.

Of course, the transformation is not performed on the source program, but rather on the underlying dataflow diagram. Notice that the constant could be pulled inside of the loop only because the variable naming it, `n`, was not modified within the loop.

Another type of folding, used to implement conditional compilation, makes use of the pseudo-value `Undefined`. This is a sort of cancerous value in that it is propagated directly to the outputs of almost any operator receiving it as an input. For example, if one or both of the inputs of a `+` operator are `Undefined`, both the sum and the carry outputs will be folded to the undefined value as well. Undefined values are created by the `ifYes` and `ifNo` operators if a constant condition input is of the wrong sense. The four combinations of these operators and constant values, together with their folded representations, are illustrated in Figure C–23. When the condition input of an `ifYes` or `ifNo` operator is known, its output will be folded either to its value input or to `Undefined`. The only operation with any immunity at all to the undefined value is `Merge`. If one of its inputs is `Undefined`, its output will be folded to the value of its other input.

A simple example may help to illustrate how the undefined value is used to provide for conditional compilation. Here is a fragment of a Silicon source program:

```
double:= 1;
b:= IF double THEN 2*a ELSE a;
```

Figure C–24*a* shows the dataflow diagram that corresponds to this program fragment.

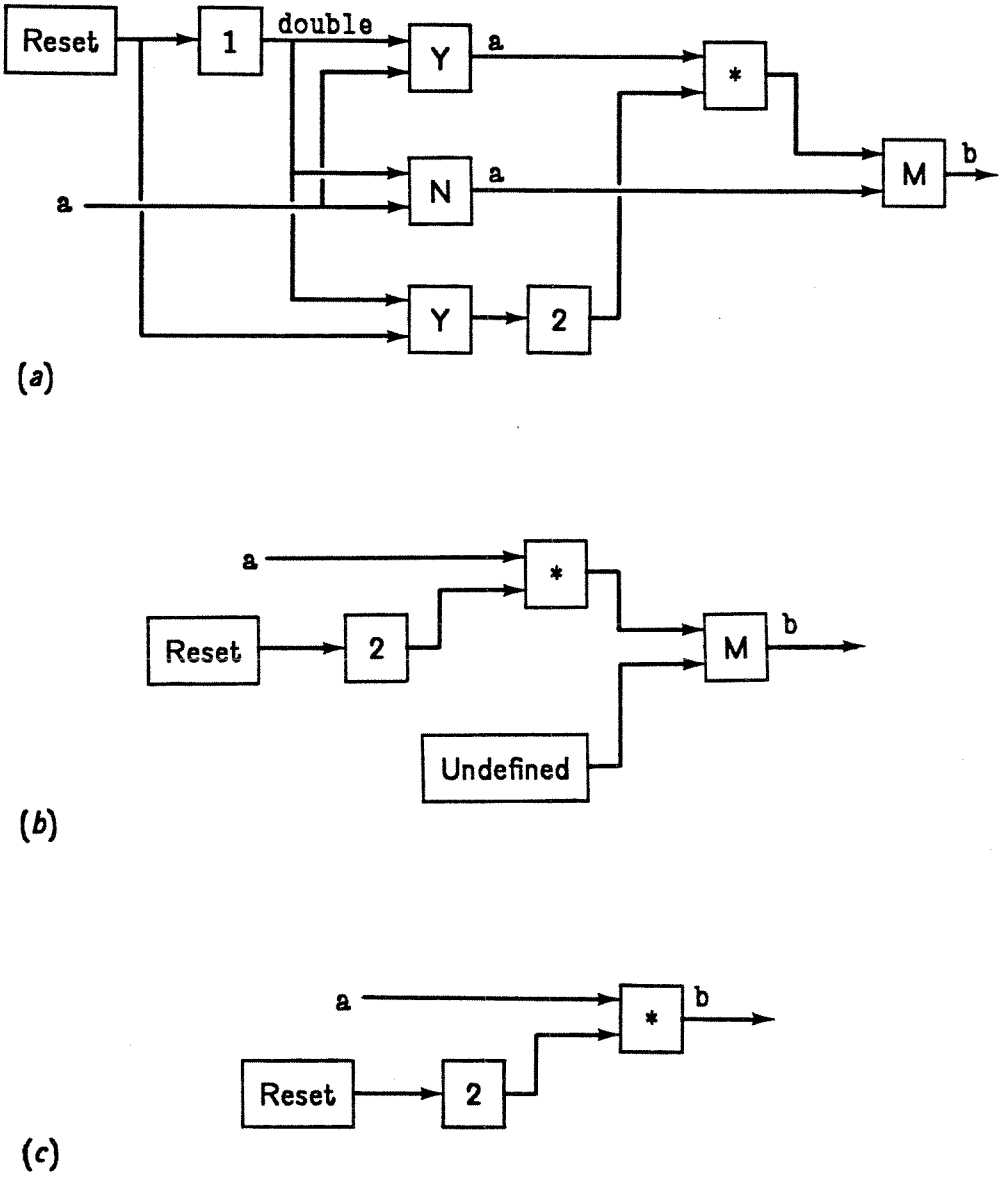


Figure C–24. Folding may be used to achieve a sort of conditional compilation, as illustrated by the dataflow diagram before folding, after folding the conditional operators, and after folding the merge operator.

After the **ifYes** and **ifNo** operators are folded, the dataflow diagram is changed so that it corresponds to Figure C–24*b*. Folding the **Merge** operation gives the final result, shown in Figure C–24*c*.

The compiler uses a similar technique when it expands macros. Upon encountering a macro invocation, the compiler will check the value of **Reset**. If this value is **Undefined**, it will not expand the definition. The **Reset** signal is used because it should have a defined value in every circumstance but this one. By testing the **Reset** signal before expanding the definition, the compiler can handle recursive definitions without going into an infinite loop.

It may seem that much of the effort required for the folding operation goes into making up for the ineptness of other parts of the compiler. If the dataflow analysis generated less spurious junk, there would be less need for folding. Although it is generally true of optimizing compilers that much of their effort is spent solving problems introduced by the compiler itself, this is especially true here. One way to improve this situation would be to introduce an extra pass for performing a source/sink analysis before the dataflow analysis, as is done in typical compilers. With this information the dataflow analysis program could avoid, for example, running values through loops where the value was never used. This scheme almost works here, but not quite. Part of the problem is that not all of the required information is available until the folding of definition expansions has been completed. While the scheme would work most of the time, it still requires a folding operation like the one described here to handle the odd cases. By adding an extra pass for source/sink analysis and modifying the dataflow analysis and folding operations, it would probably be possible to make a cleaner and better implementation. For the current, trial implementation, however, this was deemed unnecessary.

C.11 Size Determination

Part of the compiler's task is to determine the sizes of values used in a Silicon program. This information is needed for the later stages of the compiler, which select the devices used to implement the program. Be performing size determination as part of the dataflow analysis, more meaningful error messages can be generated. Aside from this consideration, however, it might just as well be performed as a separate pass over the dataflow diagram.

Size information is available from several places. The first and most obvious source is constants whose size was specified explicitly. Next, values produced by invoking the **get** attribute of communications ports have a known size that was specified when the port was created. Similarly, values used as inputs of the **put** attribute also have known sizes. Finally, some operators always produce fixed-size results. For example, comparison operators like **<** always produce a one-bit value.

In addition to the explicit sources of size information, most operators impose

some kind of constraint on the sizes of their inputs and outputs. The multiplication operator, for example, requires that its two inputs and single output all be of the same size. The comparison operators expect only that the sizes of the two inputs match. For the bit concatenation operator, the size of the result must be the sum of the sizes of the two inputs.

Size determination may be done, then, by propagating the known sizes through the dataflow diagram while observing the constraints imposed by the operators at the nodes. There are two potential problems here. The first is that there may not be enough information to determine the size of every value. In some cases, like bit subscripts, this is not a problem because the size doesn't matter, but in most situations, an error message is called for. The second problem is that sizes may be inconsistent, as, for example, when a ten-bit value is given as a parameter to the **put** attribute of a sixteen-bit port. Here again, the compiler must issue an error message.

C.12 Functional Simulation

Once the dataflow diagram has been constructed, it is a relatively simple matter to perform a functional simulation. Notice that the simulator is representation independent, as is the dataflow diagram itself. Thus, a single functional simulator could serve for every version of the compiler. It would not entirely replace the low-level, representation specific simulators, however. These would supply timing information as well as provide a double-check on the validity of the compiler itself.

The Silicon language includes a debugging statement that has not yet been described. The statement consists of a text string, enclosed within quotes, and an expression, in the following form:

```
"Number of points: " nPoints;
```

At the appropriate point in the simulation, the text string will be typed out on the user's terminal, along with the value of the expression. Recall that for ordinary compilation, the only handles on the completed dataflow diagram are through the operations on communications ports. For simulations, the debugging statements provide an additional handle. This means that debugging statements will be removed automatically if an ordinary compilation is underway, and furthermore, the expressions used in the debugging statement may be arbitrarily complex.

The actual functional simulation process is fairly simple, proceeding in two indefinitely repeated phases. During the first phase, a new value is computed for each of the operators in the dataflow diagram. During the second phase, each of these new values is made available at the output of the operator. Of course, the undefined value is a perfectly legitimate result at this stage. Debugging statements and accesses to communications ports cause terminal or file I/O to occur.

A slight problem with the simulation technique just described is that it may

require an arbitrary amount of storage along each of the arcs. Such would be the case if certain portions of a loop ran at a rate substantially different from the others. The problem can be solved by forcing all values in a particular loop to circulate at the same speed. A convenient way to accomplish this is to synchronize all of the **Merge** operators for that loop. None would be permitted to produce a result until all were ready to do so.

C.13 Bit Serial Implementation

This section outlines the process used to transform the dataflow diagram representing a Silicon program into a layout that implements it. This process is broken down into three stages. First, the compiler applies transformations to the dataflow diagram in order to make it more suitable for a serial implementation. Next, it analyzes the timing properties of the program to determine where it must insert storage elements. Finally, it selects and places the serial components to be used, and it generates the connections between them.

Bit serial arithmetic has several properties that make it well suited for implementation as an integrated circuit. The individual computing elements require just a fraction of the space used by their fully parallel counterparts. Moreover, the wiring costs are substantially reduced because the number of wires used to represent a value is independent of and much smaller than the actual word size. Also, the communication that does occur tends to be fairly localized.

On the other hand, bit serial circuits may be substantially slower than functionally similar parallel circuits. This is so because, although the area consumption of the computing elements is independent of word size, the computing time is not. There are, however, cases where this is not a great problem. For example, a shift-and-add multiplier looks and performs much the same whether implemented serially or in parallel. If other serial operations can be pipelined together with multiplications, their performance limitations can be minimized. Furthermore, because of their smaller size, more serial devices can fit on a single chip. In some applications, it is possible to enhance the performance by making use of this potential for parallelism.

The convention followed here for bit serial arithmetic uses two wires to represent a single number. One of these, the data wire, carries the actual bits that make up the number. These are passed at the rate of one bit per clock cycle, with the least significant bit appearing first and the most significant bit last; there are no gaps between successive bits of a word. The second wire carries a control signal that is high whenever the data wire represents the bits of a word; it is low between words. Note that there must be a gap of at least one clock cycle between successive words. The data line must be held low between words when the control line is low. The timing diagram in Figure C–25 is an example illustrating how the value fifty would be represented as an eight-bit number.

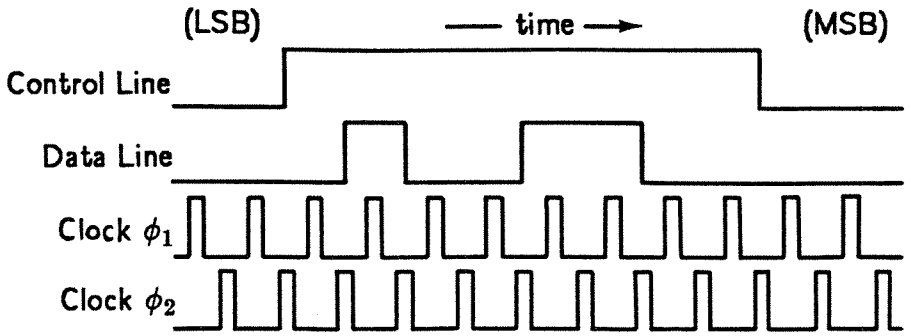


Figure C–25. Timing diagram showing the number fifty represented bit serially as an eight-bit number ($50_{10} = 00110010_2$).

This dual wire representation has two main advantages. The first is that the serial devices need not maintain counters to be able to determine where a word starts and where it ends. Adders, for example, can use the control line to determine when the carry bit should be cleared. The second advantage is that there is no centralized control. Since values announce themselves, global circuitry for synchronizing events is unnecessary. This circumstance, in turn, lessens the need for global communication.

The compiler makes two kinds of modifications to the dataflow diagram in its attempt to improve it for a bit serial implementation. The first modification concerns the use of constants. Recall that constants are represented in the dataflow diagram as nodes with a single input serving as a sort of trigger for the constant valued output. This trigger input is somehow connected to the reset signal in the dataflow diagram, and that is the potential source of the inefficiency. If the constant is used in a deeply nested control structure, many nodes in the dataflow diagram will be required simply to propagate the reset signal.

The constant optimization improves the dataflow diagram by finding suitable substitutes for the reset value used as constant inputs. Constants are most often used as one of the two inputs of a binary operator, and therefore the other input is a convenient replacement for the reset value. Figure C–26 illustrates a dataflow diagram before and after this modification. Note that operators with a constant input will always have at least one non-constant input because if all of the inputs had been constant, the operator would have been folded out at an earlier stage of the compilation.

The method just described for dealing with the constants has other attractive properties besides getting rid of the reset signal. These have to do with the implementation of the serial elements themselves. First, notice that most of the serial devices expect to receive their inputs at the same time. For example, the two inputs of an adder must arrive together. If a constant input to the device is triggered by another input, this constraint will be satisfied without the use of additional circuitry. Second, for many devices, both operands must be of the same size. If this is the case, and the control signal for the non-constant input is made to double as the control signal for the constant input, the implementation of the constant can often be made simpler.

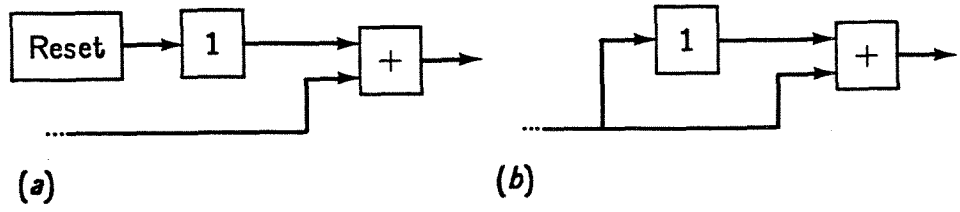


Figure C–26. Constant optimization to reduce the use of **Reset**.

Moreover, the actual wiring cost may be reduced slightly.

The second form of optimization performed by the compiler is known as Common Subexpression Elimination, or CSE. In its simplest form, this optimization simply combines the code for multiple occurrences of the same computation. For example, the following Silicon excerpts show how a program might be transformed by CSE.

```
Before:  b:= (a+1)*(a+1);
After:   tmp:= (a+1); b:= tmp*tmp
```

This is a rather contrived example, but it does serve to illustrate the basic idea. It is possible for entire statements to be merged in a similar manner.

The scope of common subexpression elimination is the entire program. This means that if two portions of the dataflow diagram can be merged, those portions will be found even if the corresponding parts of the source code were widely separated. In addition, since macros have been completely expanded by this point in the compilation process, macro expansions present no barriers to the removal of redundant subexpressions.

The compiler is also capable of reaching inside of conditionals when performing CSE. That is, if there were common subexpressions within two alternatives of a conditional statement, or even within different conditional statements, they would be merged and implemented just once. For example, suppose that some combinations of the alternatives of a **CASE**-statement contained code to increment a program counter. This computation would be implemented just once, and the result would either be passed through the **CASE**-statement or ignored, depending on the value of the switch. The inherently parallel nature of serial systems makes it reasonable to compute values that are simply discarded later. Once a serial device has been created, the interconnection structure is rich enough that it costs no more to compute than to sit idle. If the result of its computation is useful, so much the better.

Eliminating common subexpressions cannot, of course, work miracles. One limitation is that although CSE can detect and merge duplicate conditionals, duplicate loops are completely out of its reach. Another is that it cannot automatically convert macro expansions into something more akin to subroutine calls. In other words, it would be

desirable if complicated macros could be expanded only once and then somehow be linked to each of the places where they were referenced. Such an operation could be performed by creating a separate process for the subroutine and then connecting it to the main process by means of internal communications ports. The compiler should leave simple macros unchanged because the overhead needed to combine them would outweigh any benefits that might be derived. This kind of behavior begins to move from the realm of optimization into the area of system design, and it should probably be left in the hands of the user rather than be fully performed by a compiler.

After the compiler has finished applying transformations in its attempt to tailor the dataflow diagram for serial arithmetic, it must perform a timing analysis of the entire dataflow diagram. With the resulting information, the compiler can determine where storage devices are required for the correct functioning of the circuit. It then inserts the necessary devices into the dataflow diagram immediately before producing the final layout.

Serial devices each require some number of clock cycles or bit times in which to perform their respective operations. This time is the delay introduced by the device, and it may take one of two forms. Fixed delays are those that can be determined at compile time. For example, an adder invariably produces the first bit of its result one clock cycle after it has received the first bit of its inputs. Fixed delays, denoted by f, are the most common in serial systems. The second type of delay cannot be predicted at compile time. It is called a variable delay, denoted by v, because it depends on the environment of a working chip. For example, an asynchronous communication port might not produce a value until some person walks by and pushes a button. Notice that the delay introduced in this way may not even be the same each time the device is activated. Another, less obvious example is the loop. Loops may repeat an indefinite number of times, perhaps governed by a value received from off-chip. Since values cannot escape a loop until it terminates, an arbitrary delay will result.

In addition to their delay characteristics, most serial devices impose constraints on the arrival times of their inputs. An adder, for example, expects to receive both of its inputs at the same time. A concatenation device, on the other hand, requires the second input to immediately follow the first. Another type of constraint arises from loops. Here, the total delay through the body of the loop must be at least as large as the word size of the values used within the loop. If this constraint is violated, the initial values of the loop will collide with the results of the first iteration at the merge operator. Yet another constraint is that the delay through a loop must be the same for all of the values computed or referenced within the loop. Without this constraint, some parts of the loop could run faster than others. This situation, in turn, might require the compilation of unbounded storage, which is, of course, rather difficult to implement.

In order to satisfy the constraints imposed by the network of serial devices, it is necessary to introduce extra delays, or storage devices. These devices accept one input value that will be copied to their single output at some time after it arrives. These

introduced delays, denoted by d, may potentially appear in the dataflow diagram before the inputs of any operator that has at least two operands. No delays are necessary for single input operators because there is nothing to synchronize. For example, Figure C–27 shows the dataflow diagram corresponding to the following Silicon program, including the required delays.

```
i:= asyncSerIn(16,0);
o:= asyncSerOut(16,0);
sum:= 0;
n:= i.get;
WHILE n > 0 DO
 { sum:= sum + i.get;
   n:= n - 1
 };
o.put(sum);
```

Notice that the loop makes use of a variable delay to express the fact that it is executed repetitively. The value of the variable delay is zero for the initial iteration and increases by one for every successive iteration. The length of each iteration is denoted by m.

It is now possible to write a set of equations that describe the constraints. There will be one equation for each device having more than one input, in addition to the equations required to express the loop constraints. Basically, these equations are formed by computing the arrival times of each of the inputs in terms of the f, v, and d values. Appropriate combinations of these are then set equal to each other, perhaps with some constant offset. The equations for the previous example are given in Figure C–28

In theory, it should be possible to find values for the d's in terms of the f's, v's, and m that minimize some cost function. There are some problems with this in practice, however. For one thing, it isn't clear what the cost function should be. Should it measure area utilization, execution time, or something else? What is more, any solution would depend on the v's and would therefore have to be computed symbolically. Even if it could compute such a solution, the compiler would probably have difficulty implementing it.

An alternative is to use a heuristic approach in order to find a solution that may not necessarily be optimal in every case, but that would always work. This is the approach taken here. Basically, the compiler will generate circuitry to insure that devices with more than one operand will receive their inputs at the correct times. It will also generate devices to insure that the constraints imposed on variables used within loops are satisfied.

To begin with, consider the techniques required for aligning the inputs of a two-operand device. Loops and devices with more than two inputs are just extensions of these basic techniques. There are three cases to consider, determined by how much is known about the relative timing of the inputs.

The first case occurs when the difference between then arrival times of the two inputs can be determined at compile time. This would be the case if the expressions

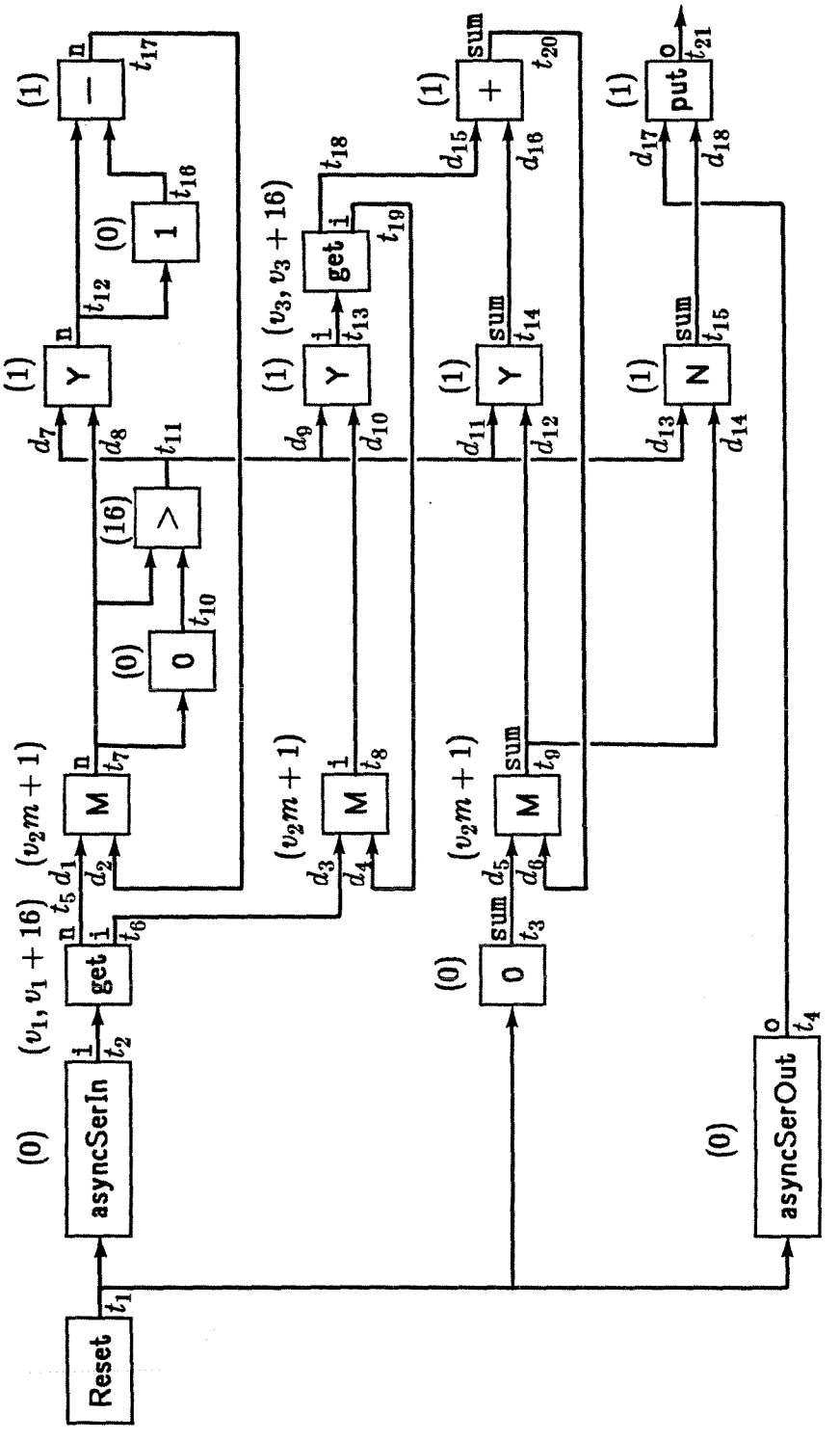


Figure C–27. Dataflow diagram corresponding to a simple summation program.

$$
\begin{aligned}
t_1 &= 0 \\
t_2 &= t_1 \\
t_3 &= t_1 \\
t_4 &= t_1 \\
t_5 &= t_2 + v_1 \\
t_6 &= t_2 + v_1 + 16 \\
t_7 &= t_5 + d_1 + v_2 m + 1 \\
&= t_{17} + d_2 + (v_2 - 1)m + 1 \\
t_8 &= t_6 + d_3 + v_2 m + 1 \\
&= t_{19} + d_4 + (v_2 - 1)m + 1 \\
t_9 &= t_3 + d_5 + v_2 m + 1 \\
&= t_{20} + d_6 + (v_2 - 1)m + 1 \\
t_{10} &= t_7 \\
t_{11} &= t_7 + 16 \\
&= t_{10} + 16 \\
t_{12} &= t_{11} + d_7 + 1 \\
&= t_7 + d_8 + 1
\end{aligned}
\qquad
\begin{aligned}
t_{13} &= t_{11} + d_9 + 1 \\
&= t_8 + d_{10} + 1 \\
t_{14} &= t_{11} + d_{11} + 1 \\
&= t_9 + d_{12} + 1 \\
t_{15} &= t_{11} + d_{13} + 1 \\
&= t_9 + d_{14} + 1 \\
t_{16} &= t_{12} \\
t_{17} &= t_{12} + 1 \\
&= t_{16} + 1 \\
t_{18} &= t_{13} + v_3 \\
t_{19} &= t_{13} + v_3 + 16 \\
t_{20} &= t_{18} + d_{15} + 1 \\
&= t_{14} + d_{16} + 1 \\
t_{21} &= t_4 + d_{17} + v_4 \\
&= t_{15} + d_{18} + v_4
\end{aligned}
$$

Figure C–28. Timing constraints for the dataflow diagram of Figure C–27.

for the two times contained the same variable delay. Figure C–29*a* shows a dataflow diagram that gives rise to the following set of equations.

$$
\begin{aligned}
t_1 &= v_1 + f_1 \\
t_2 &= v_1 + f_2
\end{aligned}
$$

The difference in arrival times is

$$t_1 - t_2 = f_1 - f_2.$$

Notice that the variable delay terms cancel in the subtraction, leaving only fixed delays. Since these are known to the compiler, a numeric answer will result. In this case, a fixed-length delay must be inserted before one of the inputs to the binary device, as shown in Figure C–29*b*. The delay is implemented as a simple shift register with its length chosen to delay the earlier value by the amount required for alignment with the later one.

The second case in the problem of storage insertion for a binary device occurs when one of the inputs is known to follow the other. This happens if all of the variable delay terms present in one of the times are also present in the other, as in the following set of equations.

$$
\begin{aligned}
t_1 &= v_1 + v_2 + f_1 \\
t_2 &= v_1 + f_2
\end{aligned}
$$

The difference here is

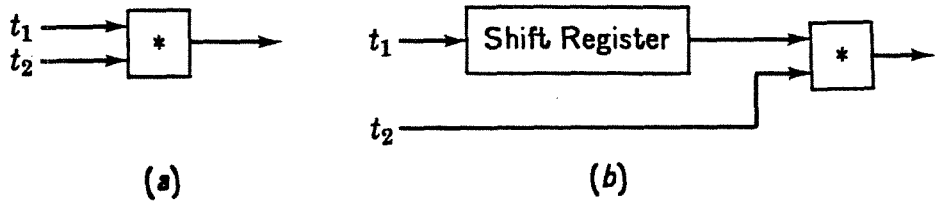


Figure C–29. A dataflow operator whose inputs are ready at comparable times, shown before and after delay insertion.

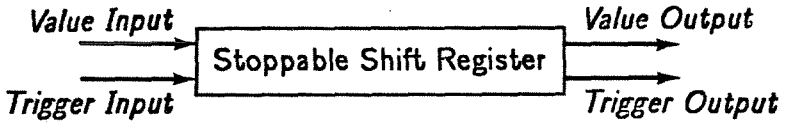


Figure C–30. Representation of a stoppable shift register in a dataflow diagram.

$$t_1 - t_2 = v_2 + (f_1 - f_2).$$

If the value of v_2 is large enough, the first input will arrive after the second. By inserting a fixed delay before the first input, effectively increasing f_1, the compiler can insure that the first input will always arrive after the second. Notice, however, that since v_2 might be arbitrarily large, the second input may be required to wait for an arbitrary length of time. An ordinary shift register cannot implement this operation, because its length would have to vary. Instead, the compiler uses a stoppable shift register, represented as shown in Figure C–30. This device is like a shift register having a length equal to the word size, but once a value has been shifted all the way in, it is stopped and stored until released by a signal on the trigger input. The trigger output signals when a value being held is ready to be released. Using a stoppable shift register, the delays required by the binary device can be implemented as shown in Figure C–31. The length of the standard shift register is chosen to insure that the second value will always be waiting in the stoppable shift register when the first value arrives. Using the control signal of the first input as the trigger for the second input guarantees that they will arrive simultaneously at the binary device.

The third and final case arises when the compiler has no information whatsoever about the ordering of the two arrival times. This would happen if each of the arrival times contained variable delays that were not present in the other.

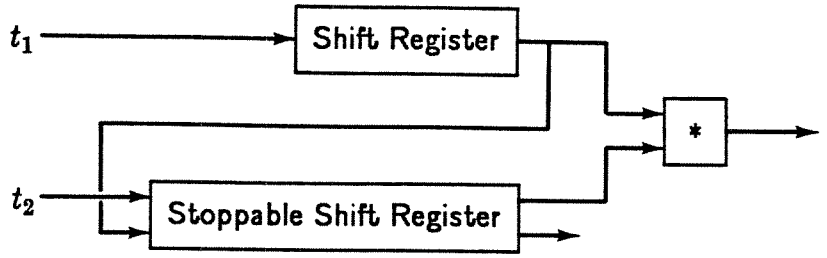


Figure C–31. Delays used to synchronize a binary device when one of the inputs is known to arrive some time later than the other one.

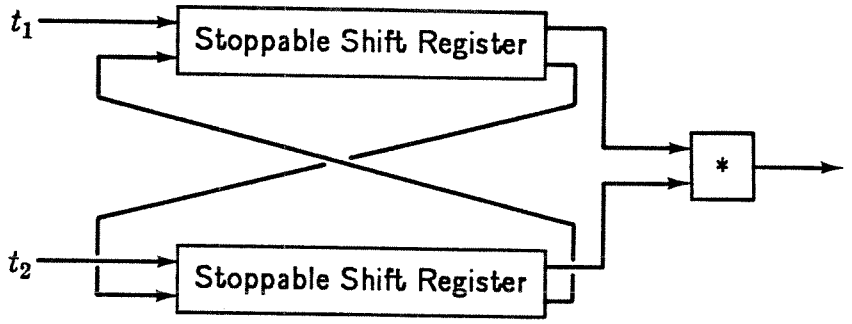


Figure C–32. Dataflow diagram showing synchronization required when the two inputs of a binary device arrive at incomparable times.

$$t_1 = v_1 + f_1$$
$$t_2 = v_2 + f_2$$

The difference in arrival times is

$$t_1 - t_2 = (v_1 - v_2) + (f_1 - f_2).$$

Depending on the relative values of v_1 and v_2, either value might be the first to arrive at the device. Each of the values must, therefore, be prepared to wait for the other, and two cross-coupled stoppable shift registers are required, as illustrated in Figure C–32. The earlier value will thus be held until the later one is available.

Stoppable shift registers are sometimes used where an ordinary shift register might suffice. This would happen if the difference in the arrival times of a device's inputs could be determined at compile time, but was very large. In such a case, it would be more economical to use a stoppable shift register, because its maximum length is limited to the size of the word that it must be able to hold.

As mentioned earlier, synchronizing the inputs of loops or devices with many inputs is basically an extension of the techniques used for binary devices. The inputs

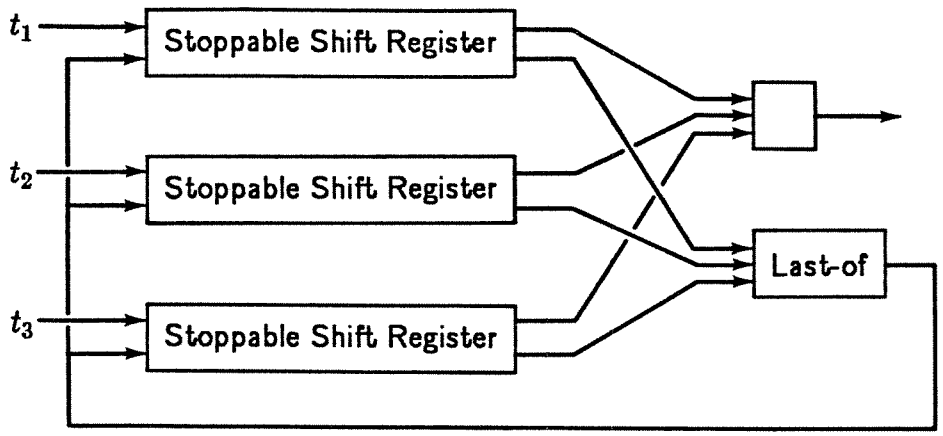


Figure C–33. A last-of device used to synchronize three signals.

can be split up into groups according to their arrival times. The groups would be defined so that the compiler could compute the difference in the arrival times of any pair of values within a group. If it turns out that there are just one or two such groups, synchronization can be performed pretty much as described for binary devices. However, another device is necessary if more than two groups must be synchronized.

The device required for synchronizing three or more mutually incomparable devices is called a *C*-element, or last-of device. It has a single output, which is not asserted until all of its inputs have been asserted. This output will not be deasserted until all of the inputs have been subsequently deasserted. If the compiler cannot determine the order of arrival of the inputs of a device having many operands, all inputs must passed through stoppable shift registers that are triggered by a last-of device. Figure C–33 contains an example showing a last-of device being used to synchronize the three inputs of another device. This notion can also be extended beyond devices having just three inputs, or three input groups.

After storage allocation is complete, the only task remaining for the compiler to generate the actual layout. By this point in the compilation process, nodes in the dataflow diagram correspond directly to physically realizable devices. The compiler selects them from a library of serial devices, tailors them to their electrical environment, places them on the layout, and routes wires to connect them as indicated in the dataflow diagram.

Serial devices in the compiler's library must adhere to some conventions. Power, ground, and two clock signals are accessed in a standard way on the metal layer. The control and data inputs and outputs are also standardized. Figure C–34 outlines a device having a single input and a single output. A device having more than one input or output would be formed from more than one copy of this basic layout, stacked vertically one atop the next. It is also necessary for the internals of the serial device to be defined in coordinates that are relative to the global metal wires. This convention allows the distance between the wires to be adjusted to fit the largest device. Another

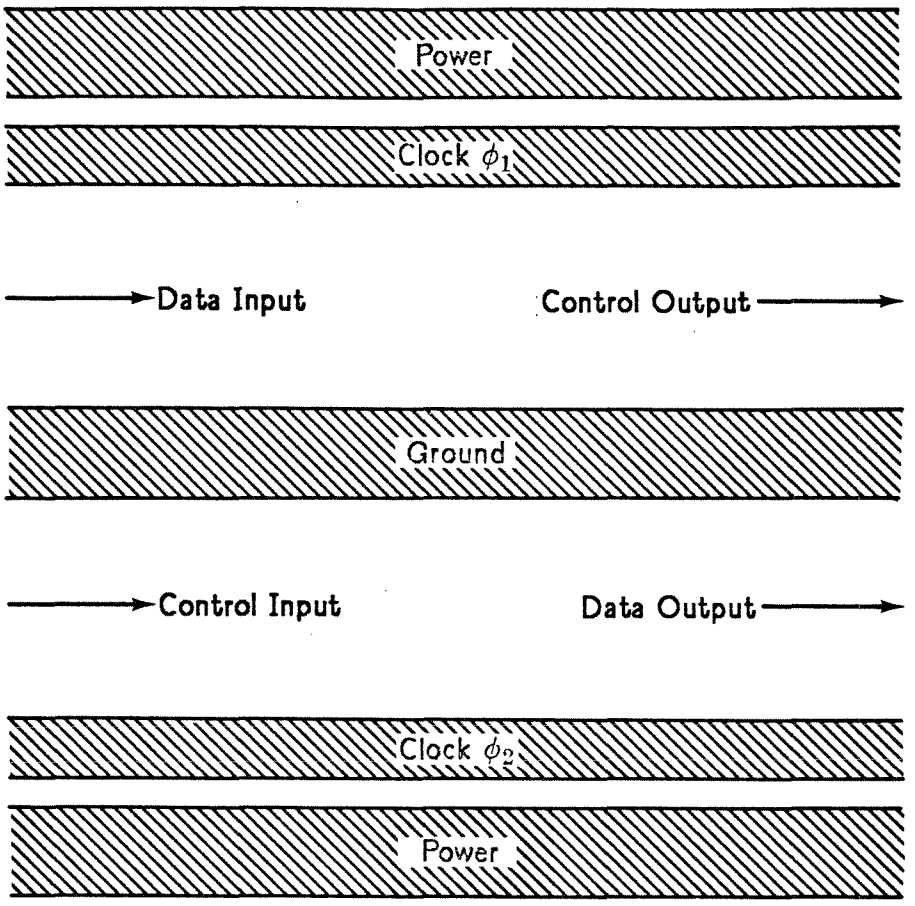


Figure C–34. Overall layout scheme for a device having a single input and a single output.

reason for this is that the compiler may find it necessary to widen the power or ground lines; it might even split them so that it can route wires through the center of a cell. Therefore, communication crossing power or ground lines in a single cell must be implemented on either the polysilicon layer or the diffusion layer using coordinates relative to different wires on each end.

The overall layout of the chip includes many copies of the power and clock busses, described above, running horizontally across the chip. The serial devices hang on these wires as required, and the spaces between them are used as wiring channels for connecting them. Vertical wires run in polysilicon between neighboring cells; horizontal wires run in metal either between cells or through them. This latter type of routing is accomplished by splitting power and ground lines, as mentioned earlier. Finally, the pads surround the whole affair and are connected to the rest of the circuit with the routed wires.

One possible placement operation, which determines where on the buses serial

devices should be hung, is simplified somewhat by the fact that it need not be done perfectly. If it isn't exactly right, the wire routing algorithm will still be able to connect the devices. Moreover, since only two wires are needed to represent a value, the cost of any extra wiring is not prohibitive.

The placement algorithm makes use of the timing information that was computed earlier during the storage allocation stage of compilation. By setting all of the variable delays to zero, the compiler can determine the earliest time at which each value might be available. This time can then be used as an approximation to the horizontal position of a device, so that devices used early in the program tend to appear to the left of the layout, and those active later appear to the right. Similarly, device outputs will be somewhere to the left of the inputs that reference them. The vertical position should be chosen in an attempt to reduce wiring by maintaining the continuity of values. That is, a device should be vertically placed in an attempt to keep down the distance to the devices producing its inputs.

A reasonable placement can be accomplished by controlling primarily the order in which the devices are placed. That is, the compiler traces through the dataflow diagram to find devices used for computing a single value, and it lays them out near each other in the order that they appear in the diagram. Devices used to compute the next value are placed beneath the first row, and so on. This method works well for single-input devices, but is less than perfect for others. It does seem to be adequate, however, and the wire routing algorithm can always take up the slack.

Pad placement is not very difficult. Recall that the pads are spread around the periphery of the circuit. The power and ground pads are at the upper left and lower right corners. The two clock pads are at the other two corners, and the reset pad is somewhere along the top. The pads implementing communications ports appear along the left edge if they are input ports, or along the right edge if they are output ports. If there isn't enough room on these two edges, the top and bottom can be used. Since the ports are connected to the rest of the circuit by only a few wires, their placement is not extremely critical.

The actual wire routing algorithm can be fairly simple because it is permitted to create all the space that it may need. The output of the placement phase consists of a tightly packed layout and a wiring list. The router goes through the list, one net at a time. It determines the most direct path for each net and shoves the serial devices aside to make room for the wire. Other more sophisticated schemes are available, but they do not seem really necessary.

Finally, although not mentioned earlier, there is almost unlimited room for ingenuity in the code generation process. It is often possible to replace two devices with a single one that is perhaps more complicated but occupies less space than the original two. This is the case, for example, when many bit-fields of a single value are extracted using the bit subscription operation. Also, clever selection of the control inputs of constants and other devices may result in some simplification. Each one represents individually only a small improvement, but the result is cumulative. Tricks

of this nature do not change the overall structure of the compiler, however, and useful though they may be, they were deemed unsuitable for inclusion in the first version of the Silicon compiler.

C.14 Status of the Serial Implementation

When the Silicon compiler project was undertaken, it was thought to be a relatively simple task that would require a detour of no more than a few months. It turned out to be substantially more involved than anticipated, not because any part of the implementation was especially difficult, but rather because there were so many individual parts. The implementation of the compiler was carried through two versions to the point where every phase has at least some tentative behavior. The current version will accept simple Silicon programs and will produce a layout that is essentially complete, except for pads and the actual serial devices themselves.

The newest version of the Silicon compiler consists of six passes. The first pass parses the Silicon source program to produce a symbol table and a series of polish strings. Pass two transforms the polish strings into a dataflow diagram. Pass three folds constant subexpressions and propagates sizes throughout the dataflow diagram. The fourth pass examines the dataflow diagram and performs an analysis to determine where storage should be inserted. Pass five generates a placement for the serial devices; and lastly, pass six routes wires though the devices and generates the final layout in the form of a CIF file [SPRO80].

The second version of the Silicon compiler was streamlined in many ways to simplify the resulting implementation. The primary technique for doing so was to substitute several simple passes for the few complex passes that were used in the initial approach. For example, the folding and size determination operations have been separated from the task of generating the dataflow diagram. The code for producing the dataflow diagram must still be capable of determining when values are not used in loops, but it need no longer be cluttered with the code for folding and size determination. On the other hand, the code for these operations no longer has access to the variable names and line numbers present in the polish string form of the source program. In order to be able to generate intelligible error messages, this information had to be included in the dataflow diagram. It was easy for the dataflow analysis pass to add this information, and moreover, having the line numbers and variable names in the dataflow diagram turned out to be useful when debugging the compiler itself.

Delay insertion in the second version of the compiler differs somewhat from the description given earlier. The goal was to attempt to generate reasonable code, but if that wasn't possible, to generate correct code at all times. There is still considerable room for improvement, however. The new delay insertion algorithm begins by performing a constant optimization, just as the old one did. Recall that the old algorithm treated loops by forcing all of the inputs of a loop to arrive at known

relative times. The new algorithm begins by assuming that none of the loop inputs arrive at times that are comparable to any other. It then determines what constraints are actually required within the body of the loop, and finally, after processing the inside of the loop, it inserts the required synchronization hardware. The advantage of postponing the insertion of extra hardware is that, in theory at least, only those signals that actually require synchronization will be synchronized. Furthermore, if values entering the loop for the second or succeeding iteration also require synchronization, the hardware may be shared between the two tasks. Similarly, if the looped values do not require synchronization, they need not endure the extra delay, since the hardware will be placed outside of the loop.

One aspect of the compiler that was not carried directly from the old version to the new one was common subexpression elimination (CSE). This feature was omitted, not because it was deemed undesirable, but rather because it was thought to be inessential to a demonstration compiler. In fact, some aspects of CSE survived. For example, when delays are being inserted, two shift registers passing the same value will be merged. Source level optimizations are still available to the programmer, although admittedly, it would be better to spend programmer effort on more creative aspects of the design.

The revised placement algorithm was designed primarily to splatter the serial devices down in two dimensions and not particularly to optimize overall connection lengths or any other such parameter. The basic idea is to place every device immediately to the right of the device producing its inputs. Since the desired location may already be occupied, the new device will be moved up or down in order to fit.

As for other aspects of the compiler, the routing algorithm used is pretty minimal. It consists of two phases. The first phase allocates vertical wiring space between adjacent serial devices, pushing them to the right, if necessary, to create space. The second phase adds the horizontal portions of the connection path, pushing the serial devices up and down to make room. As might be expected, this simple approach can produce some rather anomalous results, but the intent was to demonstrate that the compiler could make use of a wire router. Other researchers are tackling the problem of producing wire routers that are capable of greater efficiency.

Although it is still incomplete, the serial version of the Silicon compiler is capable of processing some simple examples. For example, it has processed the following Silicon program.

```
in, out:= asyncSerIn(10), asyncSerOut(10);
n:= in.get;
sum:= 0;
WHILE n > 0 DO n, sum:= n-1, sum+in.get;
out.put(sum)
```

The resulting layout is reproduced in Figure C–35. As mentioned earlier, the serial devices themselves have not yet been designed, so they are represented on the layout as large empty boxes. Although it is clearly not a production design, the layout illustrates the feasibility of compiling Silicon into silicon.

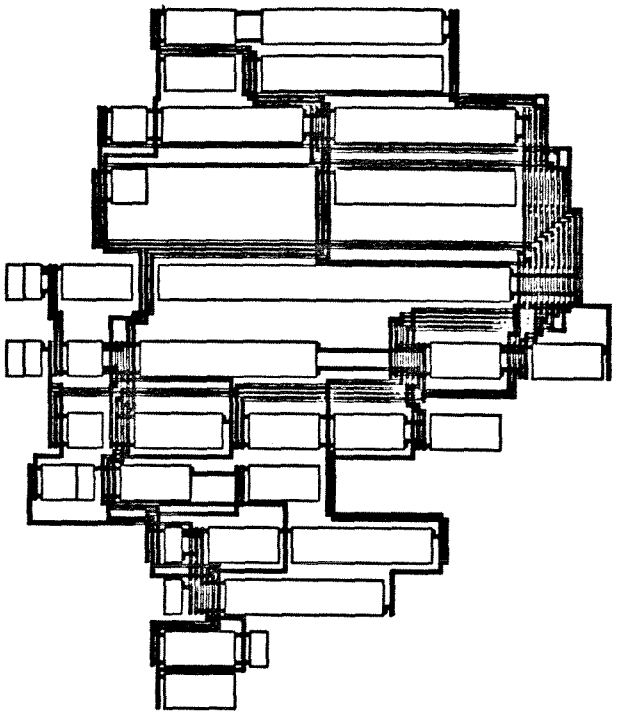


Figure C–35. Sample layout generated by the serial version of the Silicon compiler.

C.15 Interactive Implementations

Interactive implementations of the Silicon compiler would share some of the better features of custom design and automatic layout generation. On the one hand, more of the designer's experience and global knowledge of the problem could be used to determine a solution. On the other hand, the compiler's attention to detail could help produce a design free from errors.

Interactive compilation might take several forms. In what is perhaps the simplest, the compiler would generate a trial layout from a Silicon program. The designer, upon examining this layout, might see a more efficient implementation of parts of the program. The compiler would make those changes only if they preserved the function as specified in the original Silicon program.

Another possibility along these lines would permit the designer to participate in the optimization of the dataflow diagrams. He would have a catalog of transformations that could be applied to the diagram. The compiler would insure that each transformation did indeed preserve the function of the program, and that it was correctly applied. This scheme partitions the design task in a way that is more suitable for each party involved. The human designer is very adept at making the higher level decisions, while the mechanical compiler is better suited to carrying them out without error.

A further possibility for interaction would permit the designer to specify differing

representations within a single chip. For example, a particular time-critical part of the chip might be implemented in parallel. Another part might have a lot of inherent concurrency and would be better if implemented serially. The compiler would handle the transduction between the two representations.

Perhaps the most interesting scheme would allow the designer to interfere at any stage of the compilation process. While the compiler would be able to produce a complete layout without assistance, the incorporation of hints from the designer would probably result in a better final product. Moreover, this process might be repeated several times, enabling the designer to try out many different ideas.

What makes all of this possible is using the dataflow diagram as the internal representation of a program. Dataflow diagrams contain very little information that is not present in the original program. That is, they make few assumptions about a program that might restrict its later implementation. Chip assemblers do not share this property, because they start out with a complete specification of the exact devices to be used, together with their interconnection properties. The dataflow diagram, on the other hand, can be a very flexible and malleable representation of a program.

References

[AMD77] Advanced Micro Devices, Inc.: "Am25S10 Four-Bit Shifter With Three-State Outputs," Data Sheet, 1977.

[ANDE75] George A. Anderson and E. Douglas Jensen: "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," *ACM Computing Surveys*, **7**(4):197–213, December 1975.

[APPE68] A. Appel: "Some Techniques for Shading Machine Renderings of Solids," *Proceedings of the AFIPS 1968 Spring Joint Computer Conference*, The Thompson Book Company, Washington, D.C., 1968, pp. 37–45.

[ARMS77] Phillip N. Armstrong: "An Investigation of Sorting and Self-Sorting Memory," *Final Technical Report on Smart Memory Structures*, California Institute of Technology, 1977.

[ATHE81] Peter R. Atherton: "A Method of Interactive Visualization of CAD Surface Models on a Color Video Display," SIGGRAPH'81 proceedings, published as *Computer Graphics*, **15**(3):279–287, August 1981.

[BLIN77] James F. Blinn: "Models of Light Reflection for Computer Synthesized Pictures," SIGGRAPH'77 proceedings, published as *Computer Graphics*, **11**(2):192–198, Summer 1977.

[BLIN78] James F. Blinn: "Simulation of Wrinkled Surfaces," SIGGRAPH'78 proceedings, published as *Computer Graphics*, **12**(3):286–292, August 1978.

[BRES65] J. E. Bresenham: "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, **4**(1):25–30, 1965.

[CATM74] Edwin Catmull: "A Subdivision Algorithm for Computer Display of Curved Surfaces," Ph.D. Thesis, Computer Science Department, University of Utah, UTEC-CSc-74-133, December 1974.

[CLAR80] James H. Clark: "A VLSI Geometry Processor for Graphics," *IEEE Computer*, **12**(7):59–68, July 1980.

[CLAR82] James H. Clark: "The Geometry Engine: A VLSI Geometry System for Graphics," SIGGRAPH'82 proceedings, published as *Computer Graphics*, **16**(3):127–133, July 1982.

[CRAY80] Cray Research, Inc.: *Cray-1S Series Hardware Reference Manual*, Mendota Heights, Minnesota, 1980.

[CROW77] Franklin C. Crow: "The Aliasing Problem in Computer-Generated Shaded Images," *Communications of the ACM*, **20**(11):799–805, November 1977.

[DAHL74] Germund Dahlquist and Åke Björck: *Numerical Methods*, translated by Ned Anderson, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[DEC72] Digital Equipment Corporation: *PDP-8/I & PDP-8/L Small Computer Handbook*, Revised Edition, 1972.

[DENN70] Peter J. Denning: "Virtual Memory," *ACM Computing Surveys*, **2**(3): 153–189, September 1970.

[EICH77] E. B. Eichelberger and T. W. Williams: "A Logic Design System for LSI Testability," *Proceedings of the 14th Annual Design Automation Conference*, Jointly sponsored by IEEE and ACM, June 1977, pp. 462–468.

[EVAN81] Evans & Sutherland Computer Corporation: *PS300 User's Manual*, Salt Lake City, Utah, 1981.

[FLOR63] Ivan Flores: *The Logic of Computer Arithmetic*, Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[FLYN66] Michael J. Flynn: "Very High-Speed Computing Systems," *Proceedings of the IEEE*, **54**(12):1901–1909, December 1966.

[FOLE82] James D. Foley and Andries Van Dam: *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1982.

[FRAN80] Wm. Randolf Franklin: "A Linear Time Exact Hidden Surface Algorithm," SIGGRAPH'80 proceedings, published as *Computer Graphics*, **14**(3):117–123, July 1980.

[FUCH77] Henry Fuchs: "Distributing a Visible Surface Algorithm Over Multiple Processors," *Proceedings 1977 ACM National Conference*, pp. 449–450.

[FUCH79] Henry Fuchs and Brian W. Johnson: "An Expandable Multiprocessor Architecture for Video Graphics," Proceedings of the 6th Annual Symposium on Computer Architecture, published as *SIGARCH Newsletter*, **7**(6):58–67, April 1979.

[FUCH81] Henry Fuchs and John Poulton: "Pixel-planes: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*, **2**(3):20–28, Third Quarter 1981.

[GOLD71] Robert A. Goldstein and Roger Nagel: "3-D Visual Simulation," *Simulation*, **16**(1):25–31, January 1971.

[GOUR71] Henri Gouraud: "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, **C–20**(6):623–628, June 1971.

[HALV82] Jack Halverson: Private communication, 1982.

[KAHN77] Gilles Kahn and David B. MacQueen: "Coroutines and Networks of Parallel Processes," *Information Processing 77*, North-Holland Publishing Company, Amsterdam, 1977, pp. 993–998.

[KAJI82] James T. Kajiya: "Ray Tracing Parametric Patches," SIGGRAPH'82 proceedings, published as *Computer Graphics*, **16**(3):245–254, July 1982.

[KNUT79] Donald E. Knuth: *TEX and METAFONT, New Directions in Typesetting*, Digital Press, Bedford, Massachusetts, 1979.

[KUNG80] H. T. Kung and Charles E. Leiserson: "Algorithms for VLSI Processor Arrays," published as Section 8.3 of Mead and Conway [MEAD80].

[LYON80] Richard F. Lyon: "Signal Processing with VLSI—Notes for a Constantly Changing Talk," Xerox Palo Alto Research Center, November, 1980.

[MEAD80] Carver Mead and Lynn Conway: *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

[MOTO82] Motorola, Inc.: *MC68000 16-Bit Microprocessor User's Manual*, Third Edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[NEUM79] William M. Neuman and Robert F. Sproull: ***Fundamentals of Interactive Computer Graphics***, Second Edition, McGraw-Hill, New York, 1979.

[PARK80] Frederic I. Parke: "Simulation and Expected Performance Analysis of Multiple Processor *Z*-Buffer Systems," SIGGRAPH'80 proceedings, published as ***Computer Graphics***, **14**(3):48–56, July 1980.

[PHON75] Bui-Tuong Phong: "Illumination for Computer Generated Pictures," ***Communications of the ACM***, **18**(6):311–317, June 1975.

[REDD78] D. R. Reddy and Steven Rubin: "Representation of Three-Dimensional Objects," Department of Computer Science, Carnegie-Mellon University, CMU-CS-78-113, April 1978.

[ROBI81] John Robinson: "State of the Art in Real Time Image Generation," included with the SIGGRAPH'81 course notes for the ***State-of-the-Art in Image Synthesis*** Seminar, August 1981.

[ROTH80] Scott D. Roth: "Ray Casting as a Method for Solid Modeling," General Motors Research Laboratories, Warren, Michigan, Research Publication GMR–3466, October 1980.

[RUBI80] Steven M. Rubin and Turner Whitted: "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," SIGGRAPH'80 proceedings, published as ***Computer Graphics***, **14**(3):110–116, July 1980.

[RUSS78] Richard M. Russell: "The CRAY-1 Computer System," ***Communications of the ACM***, **21**(1):63–72, January 1978.

[SCHU69] R. A. Schumacher, B. Brand, M. Gilliland, and W. Sharp: "Study for Applying Computer-generated Images to Visual Simulation," ***U.S. Air Force Human Resources Laboratory Technical Report***, AFHRL-TR-69-14, September 1969.

[SEIT80] Charles L. Seitz: "System Timing," published as Chapter 7 of Mead and Conway [MEAD80]

[SPRO68] Robert F. Sproull and Ivan E. Sutherland: "A Clipping Divider," ***Proceedings of the AFIPS Fall Joint Computer Conference***, The Thompson Book Company, Washington, D.C., 1968, pp. 765–775.

[SPRO80] Robert F. Sproull and Richard F. Lyon: "The Caltech Intermediate Form for LSI Layout Description," published as Section 4.5 of Mead and Conway [MEAD80].

[SUTH74a] Ivan E. Sutherland and Gary W. Hodgman: "Reentrant Polygon Clipping," *Communications of the ACM*, **17**(1):32–42, January 1974.

[SUTH74b] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker: "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, **6**(1):1–55, March 1974.

[SUTH75] Ivan E. Sutherland: "A System of Polygon Sorting by Dissection," U.S. Patent No. 3,889,102, June 10, 1975, Assigned to Evans & Sutherland Computer Corporation.

[SUTH77] Ivan E. Sutherland and Carver A. Mead: "Microelectronics and Computer Science," *Scientific American*, **237**(3):210–228, September 1977.

[TRW78] TRW LSI Products: "MPY-24HJ 24 × 24 Bit Parallel Multiplier," Preliminary Data Sheet, September 1978.

[WARN69] John E. Warnock: "A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures," Ph.D. Thesis, Computer Science Department, University of Utah, TR 4-15, 1969, NTIS AD-753 671.

[WATK70] Gary S. Watkins: *A Real-Time Visible Surface Algorithm*, Ph.D. Thesis, Computer Science Department, University of Utah, UTEC-CSc-70-101, June 1970, NTIS AD-762 004.

[WHIT80] Turner Whitted: "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, **23**(6):343–349, June 1980.

[WHIT82] John Whitney, Jr. as quoted by Ware Myers: "Supercomputer Is Applied to Graphics, Film Production," *IEEE Computer Graphics and Applications*, **2**(9)107–109, November 1982.

[WIRT71] Niklaus Wirth: "The Programming Language PASCAL," *Acta Informatica*, **1**(1):35–63, 1971.