A PARALLEL ARCHITECTURE FOR RAY TRACING

by

Severin Gaudet

B. Sc., University of Victoria, 1978

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

SIMON FRASER UNIVERSITY
August 1985

Approval

Name:	Severin Gaudet
Degree:	Master of Science
Title of Thesis:	A Parallel Architecture for Ray Tracing

Examining Committee:

Chairperson: Dr. Art Liestman

Dr. Thomas Calvert
Senior Supervisor

Dr. Richard Hobson

Dr. Louis Hafer

Roy Hall
Graphics Consultant,
External Examiner (in absentia)

24 May 1985
Date approved

Abstract

Ray-tracing techniques for image rendering have produced some of the most realistic images to date. They are also slow because the process of tracing a ray is computationally intensive and because there are many rays to be traced. However, since computations for each pixel are independent, ray tracing is amenable to parallel processing using image space subdivision. The processor per subdivision approach is unattractive because of either the large memory requirements per processor or the high communication overhead of a shared memory.

We present an architecture that addresses these drawbacks using broadcasting. The architecture is based on. (a) an interconnection of multiple *ray tracing engines* working in parallel; (b) 3 disjoint data sets resulting from the use of a modified hierarchical data structure-based ray tracing algorithm; and (c) 3 broadcast processors each with its own memory module and broadcast bus. Simulation results show substantial rendering time improvements over mini-computer timings.

Dedication

To Sandra

Acknowledgements

I thank the following people:

Dr. Tom Calvert, my senior supervisor, for his insight and guidance throughout the course of my studies.

Dr. Rick Hobson, Dr. Lou Hafer, and Dr. Binay Bhattacharya for their valuable contibutions and the many hours of discussion.

Pradeep Chilka, my research partner, with whom a joint reasearch project became the subject of our respective theses.

This work was supported by a Natural Science and Engineering Research Council Post-Graduate Scholarship.

Table of Contents

List of Figures

Chapter 1

RAY TRACING

The potential of ray tracing techniques to produce realistic images has been extolled by so many that it is on the verge of becoming a cliche. Nevertheless, the images speak for themselves: images which can be virtually indistinguishable from photographs. These realistic images are a product of both good scene descriptions or models which describe the shape and position of objects, and good rendering techniques. We are concerned with the latter. In this chapter we shall discuss what creates the illusion of realism and why ray tracing techniques are capable of exploiting this.

1.1. The Shading Model

As stated above, ray tracing techniques have generated some of the most realistic images to date. To understand what contributes to the realism of a synthetic image, one must first understand the process that occurs naturally in the real world.

It is generally accepted that a colour video camera produces a realistic image. So let us first consider how the camera records a scene onto the phosphors or pixels of a monitor. Imagine that for each pixel on the monitor screen, these is a corresponding sensor on the camera's focal plane behind the lens. The surfaces in the scene visible to the camera reflect or transmit light into the lens and onto the sensors that in turn measure the light and send signals to their respective display pixels. The colour of each pixel is determined by the colour of the corresponding area in the scene. The colour of a surface is determined by the properties of the surface and the light falling on the it; this means we have to know how the light interacts with the surface.

In rendering a scene, it must be possible to model these light interactions in order to simulate the light being reflected or transmitted to the sensors. Examination of the light falling upon an area of the surface allows it to be classified in one of two ways. The first is light coming directly from emitting sources (eg. the sun, an incandescent bulb, a flourescent tube), this type of light is referred to as a direct source. The second type is light being reflected onto the surface from other surfaces; this constitutes an indirect or global source.

Next, examining the surface with which these two sources of light interact, we can distinguish three surface characteristics which influence these interactions. The first of these is the roughness of the surface at the microscopic level. This determines how light falling on the surface is scattered by reflection in all directions and thus how good it is as a diffuse reflector. The second characteristic, the opposite of the first, is the smoothness at the microscopic level that in turn determines the degree to which the surface can be characterized as a mirror, this property results in a specular reflection. Finally the third characteristic determines how well a surface transmits light from a light source from behind.

Combining these characteristics with the types of light sources, a formula can be derived which models the cumulative effect of the six combinations according to the physical laws of optics. This formula is referred to as the shading or illumination model. When rendering an image, we can now model the interaction of light with a surface by applying the shading model to the point being examined. Consequently, it is the completeness of the shading model which determines the degree of realism of a computer generated image. Figure 1-1 shows examples of the same scene with different light interactions being modeled.

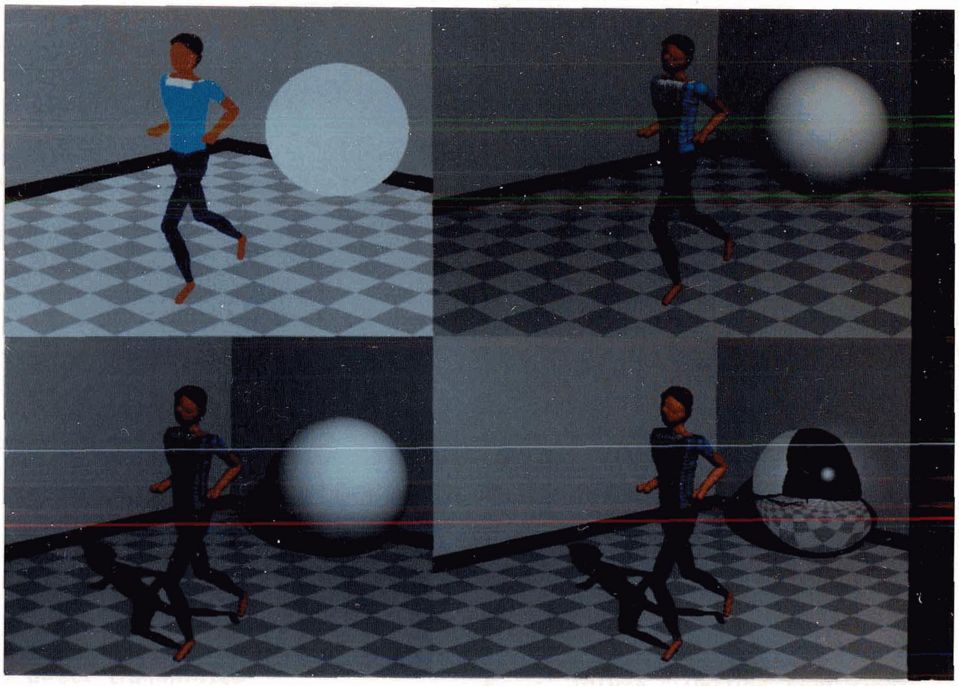

Figure 1-1: Examples of different light interactions

Shading models have become more sophisticated since the early days of computer graphics when diffuse Lambertian shading (direct source diffuse reflection) was used. In a sense, the evolution in the shading model can be compared to the evolution in painting that occurred with the Italian renaissance when the flat two dimensional-like Byzantine technique was surpassed by the vibrant realism of Michelangelo and Raphael with their studies of both light and form.

This evolution toward a better shading model began when Phong [PHON73] proposed a shading model based on empirical observations which included a term for direct source specular reflection and global source diffuse reflection. Blinn [BLIN77], Kay [KAY79], Whitted [WHIT80], and Cook and Torrance [COOK82] have contributed to making shading models more physical and less empirical by defining terms for, among other things, global and direct source transmission, the Fresnel relationship for angle of incidence, and direct source specular reflection. Most of these contributions have been brought together nicely by Hall [HALL83] in his shading model that is illustrated in Figure 1-2.

$$I = k_d \sum_{j=1}^{l} (\overline{N} \cdot \overline{L}) R_d I_j$$

direct diffuse

$$+ k_s \sum_{j=1}^{l} (\overline{N} \cdot \overline{H})^n R_f I_j$$

direct reflected

$$+ k_s \sum_{j=1}^{l} (\overline{N} \cdot \overline{H}')^n T_f I_j$$

direct transmitted

$$+ k_s R_f I_r F_r^{dr}$$

global reflected

$$+ k_s T_f I_t F_t^{dt}$$

global transmitted

$$+ I_a R_d$$

global diffuse

dr	=	distance of reflected ray travel
dt	=	distance of refracted ray travel
F_r	=	trans. per unit length of reflected ray
F_t	=	trans. per unit length of refracted ray
$\overline{H}$	=	unit reflection mirror-direction vector
$\overline{H}'$	=	unit trans. mirror-direction vector
I	=	intensity of point
I_a	=	intensity of global ambient light
I_j	=	intensity of jth direct light source
I_r	=	intensity of reflected ray
I_t	=	intensity of refracted ray
j	=	direct light source index
k_d	=	diffuse reflection coefficient
k_s	=	specular reflection coefficient
l	=	number of direct light sources
$\overline{L}$	=	unit light source vector
n	=	exponent for glossiness
$\overline{N}$	=	unit surface normal vector
R_f	=	Fresnel reflectance curve
R_d	=	diffuse reflectance curve
T_f	=	Fresnel transmission curve

Figure 1-2: The Hall shading model

1.2. Ray Tracing

When rendering an image from a 3-dimensional scene model, the following two functions are executed: a) the visibility of the surfaces is determined with respect to the viewpoint and b) light interaction with the visible surfaces and the production of colour is characterized. Most rendering techniques, such as z-buffering, cannot exploit the complex shading models because they determine visibility by projecting the 3-D modeling space onto the 2-D image plane and thus lose the third dimension necessary for the simulation of the light interactions.

Ray tracing, on the other hand, can exploit the shading models because it determines visibility not on the 2-D image plane but in the 3-D modeling space. The origin of ray tracing is found in ray casting that was proposed by Appel [APPE68] and implemented by Goldstein and Nagle at MAGI [GOLD71] as a visible surface algorithm. However, Whitted's classic algorithm [WHIT80] brought ray casting and a good shading model together in the technique now known as ray tracing.

Going back to the example of the colour video camera, ray tracing simulates its operation in reverse. Instead of than recording the light rays being reflected from the visible surfaces through the lens and onto the sensors, ray tracing sends out rays originating at each sensor on the focal plane (*image plane*) through the lens (*focal point*) into the scene (a model described in 3-D). An *initial ray* for each pixel of the image plane is sent out in this manner. Each ray is then intersected with each object in the scene to find the closest surface that is visible.

Once the nearest intersection point is found, the shading model is used to compute the colour. This involves spawning the following rays from the intersection point:

1. toward each direct light source in the scene (*light rays*) to determine if it is visible to the point and what contribution it makes to the diffuse, specular and transmitted components of the shading model;

2. in the mirror reflection direction (*reflected ray*) to determine the light intensity coming from that direction for calculation of the global source specular component; and

3. in the refracted ray direction (*transmitted ray*) to determine the light intensity from that direction for calculation of the global transmitted component of the shading model.

The algorithm s elegance lies in recursion because once spawned, the reflected and transmitted rays are traced in the same fashion as the initial rays. If these rays intersect other surfaces, the shading model is applied and new rays are spawned until the rays leave the scene or intersect a non-reflecting surface. In this fashion, the intersection tree for each pixel is built up. The intersection tree has at its root the pixel, interior nodes are intersected surfaces and leaves are direct light sources or the exterior of the scene. The branches of the intersection tree are the rays spawned during the tracing of the pixel.

Figure 1-3 follows the tracing of a ray and the resulting intersection tree. An initial ray (ir) strikes object 1 (O1). The shading model is applied at the intersection point and three secondary rays are spawned. Light ray 2 (lr2) is blocked and thus ignored. The reflected ray (rr1) strikes the semi-transparent object 2 (O2). Again, secondary rays are spawned. The reflected ray (rr2) leaves the scene and is ignored. The transmitted ray (tr1) would be traced further.

Once all the rays have been traced for a pixel, the intersection tree will contain all the light source information in the leaf nodes and all the surface characteristic information in the interior nodes. The tree is traversed in a depth-first order to calculate the final pixel colour.

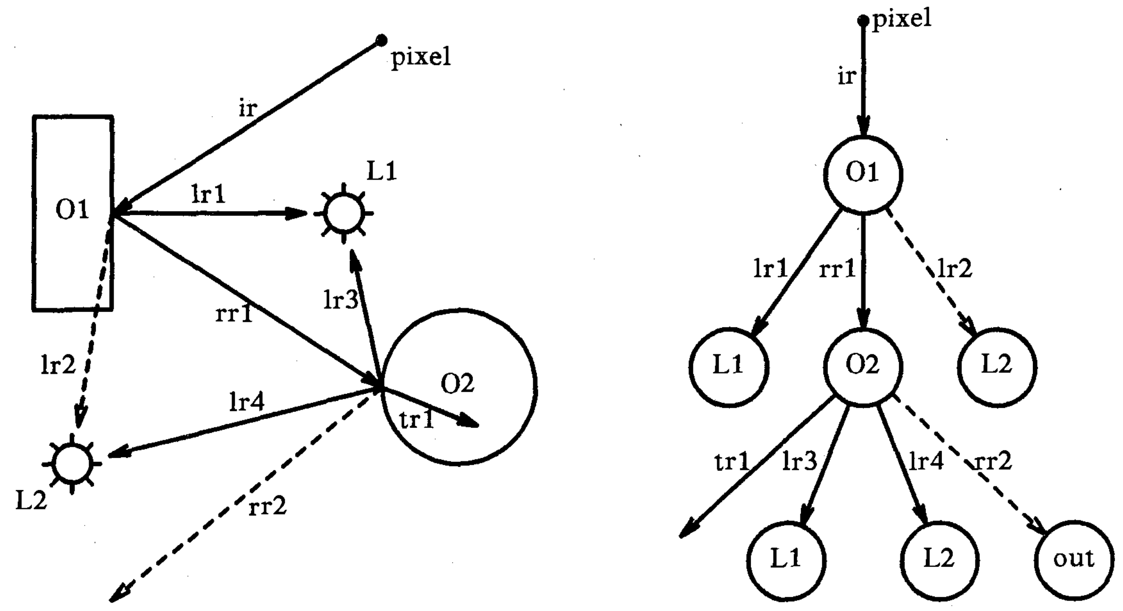

Figure 1-3: Example of the tracing of a pixel and the building of the intersection tree

1.3. An Analysis

As shown, ray tracing is a simple recursive algorithm which exploits a good shading model. However, the obvious advantages of using ray tracing are almost outweighed by its principal disadvantage computational cost. As an illustration of how severe this is, most of the reported times for published images rendered using DEC VAX/780's have been measured in hours.

Why is the algorithm so computationally intensive?

- all computations are executed in floating point.
- extensive use is made of the square root function for vector normalization of rays, normals and dot products.

- complex intersection computations are required for some classes of objects such as fractals and 3-D spline surfaces.

- the number of intersection calculations is large since determination of the closest surface requires that a ray be tested be tested against all objects in the scene.

- the number of rays spawned during the ray tracing process is also large.

To show the sheer number of computations required in ray tracing an image, we shall use an analysis of the complexity of ray tracing similar to that found in [DIPP84]. We shall also use data from the run-time profile of the program used to generate Figure 1-4 on a DEC VAX/750 with a floating point unit. To do this, we make the following assumptions:

- each intersection tree has depth $D = 4$.

- the average number of recursive reflected and transmitted rays spawned per intersection $N = 1.1$ (100% of the intersections will spawn a reflected ray; 10%, a transmitted ray).

- the number of objects in the scene $O = 1093$ (833 spheres and 260 polygons) which corresponds to the scene model used to generate Figure 1-4.

- the number of direct light sources $L = 1$.

- the resolution of the image $R_0 = 512 \times 384 = 196608$ pixels.

- the average intersection calculation time $T_i = 0.000429$ seconds.

- the average ray spawning time $T_s = 0.000710$ seconds.

The resulting calculations are given below:

- total number of rays traced: $R_t = R_0 \bullet \frac{\bullet(N^D-1)}{\bullet(N-1)} (1+L) = 1824915.$

- total number of intersections: $I_t = OR_t \sim 2^9.$

- total time: $T_t = R_t(T_s+T_iO)$ *Approx* 238 hours.

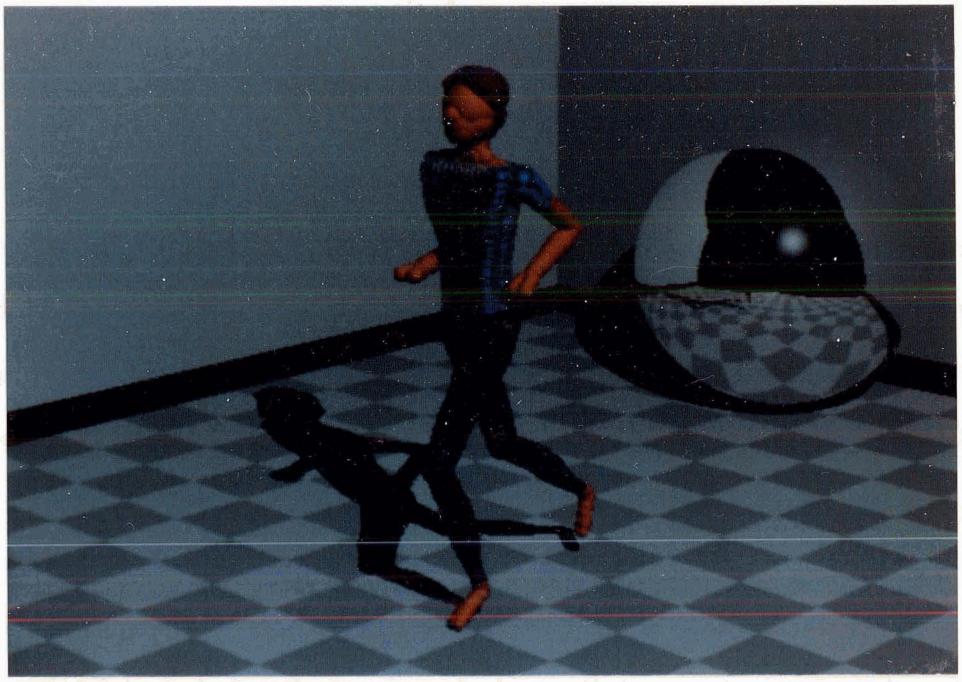

Figure 1-4: Sample scene for analysis

Varying the size of the parameters can significantly increase the number of intersection calculations that must be performed. For example:

- doubling image resolution R_0 to 1024×768 increases I_t by a factor of 4.
- adding 2 more direct light sources to the scene doubles I_t.
- doubling the number of objects in the scene also doubles I_t.

This analysis was based on the standard algorithm whereby all rays are intersected with all objects. Fortunately, many modifications have been proposed to the algorithm to increase its performance. These improvements are discussed in the next chapter.

Chapter 2

ALGORITHM IMPROVEMENTS

Whitted [WHIT80] has stated that intersection calculations can account for up to 95% of the rendering time Using the standard recursive algorithm, the work due to intersection calculations is expressed as *number of rays* x *number of objects*. To reduce the time to accomplish a task one can either work faster or one can work more efficiently. Working faster means using faster computers, special-purpose processors or specialized architectures These are issued discussed in the next chapter. Working more efficiently means reducing the number of intersection calculations by either reducing the number of rays spawned or by reducing the number of objects that must be intersected, or both. In this chapter, proposed improvements to the standard algorithm are discussed.

2.1. Reducing Rays

The number of rays spawned during the rendering of an image is dependent on many factors such as the number of pixels to be traced, the number of lights, the amount of empty space in the scene and the density of reflective and transparent surfaces. These factors are outside the control of the renderer. Where the renderer has control over the number of rays is in the process of spawning secondary rays

Adaptive tree depth proposed by Hall [HALL83] is aimed at controlling the depth of a pixel's intersection tree. Before spawning a ray, the maximal contribution that the ray could potentially make to the final pixel value is calculated. If this contribution is below a pre-determined threshold, the ray is not spawned. Hall has shown that even in highly reflective scenes such as a room of mirrors, the average tree depth was 1.71.

Assuming an average tree depth of 1.71 in the analysis discussed in the previous chapter, both the number of rays traced and the intersection time would be reduced by 62%.

2.2. Reducing Objects

Reducing the number of objects with which a ray must be intersected holds the greater potential for increasing performance. Rather than doing a blind search through the entire list of objects, techniques have been proposed to partition the objects or the scene to permit a more efficient search. The objective is to determine the subset of objects which are spatially close to a given ray such that the chances of the ray intersecting any of these objects is greater. In all techniques discussed below, the data organization particular to each is created as a pre-processing step. The time penalty for pre-processing is typical less than 8% of the new image generation time which is, in turn, significantly less than the standard algorithm time.

2.2.1. Bounding Volumes

Objects that require complex intersection calculations, such as is needed for fractal or spline surfaces, can be enclosed in a bounding volume, such as a sphere or a rectangular parallelepiped; this results in a much simpler intersection calculation that will potentially save time. If the ray does not intersect the bounding volume, then there is no need to execute the test with the complex object. Similarly, if one has built an object from a collection of objects, for example, the collection of spheres making up the forearm of the jogger in Figure 1-4 , this logical collection of spatially related objects can also be enclosed within a bounding volume to save on intersection calculations.

The concept of bounding volumes, [CLAR76], [WHIT80], involves enclosing a complex object or a collection of objects as tightly as possible within a volume which is simple to intersect. If a ray is tested for intersection against this volume and fails, the result is that the enclosed object or objects are efficiently eliminated from the intersection calculation.

Figure 2-1 shows a 2-D view of a collection of spheres bounded by a box. Ray *a* intersects the volume and so must be tested against every enclosed sphere; ray *b* fails the intersection test with the volume thus avoiding 12 intersection calculations with the enclosed objects.

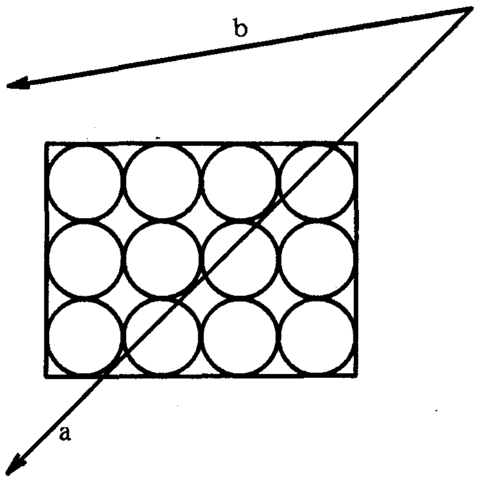

Figure 2-1: Example of a bounding volume

The decision on how to group objects and on which bounding volume to choose is largely in the hands of the user who models the scene. Weghorst et. al. [WEGH84] have done some work on the automatic selection of bounding volumes using the criteria of void area and a total cost of intersection test function. Both of the criteria are ray dependent and thus scene dependent.

At this stage we have a collection of bounding volumes. The next step would be to have a process whereby only bounding volumes lying along a ray's path are tested for intersection.

2.2.2. Hierarchical Data Description

From a collection of bounding volumes, a hierarchical data description, [CLAR76], [WEGH84] can be built using a similar approach as for the definition of bounding volumes. Collections of bounding volumes that are spatially close can be enclosed by a larger bounding volume and so on, until the whole scene is enclosed. The result is a tree where the root node is this volume, the interior nodes are bounding volumes enclosing bounding volumes, and the leaves are bounding volumes enclosing objects Again the choice of volume and the grouping of the volumes are largely defined by the user during the modeling process.

The purpose of the hierarchy is to rapidly eliminate bounding volumes and objects from the intersection calculation. When a ray is spawned, it is assumed to always intersect the root volume. It is tested against the second level bounding volumes. If a volume is intersected, a recursive descent of the hierarchy begins. The saving occurs because a bounding volume is tested for intersection if and only if its parent volume has been intersected by the ray The hierarchy is pruned down to the leaf level. Figure 2-2 shows a 2-D representation of a scene with its corresponding hierarchy.

Weghorst et. al have shown savings of 12% to 55% over the use of bounding volumes only. Our own results have shown that the use of both bounding volumes and a hierarchical data structure decreases rendering times by up to 95% over the standard algorithm.

The efficiency of using bounding volumes with a hierarchical data structure is largely in the hands of the user. The depth of the data tree, the number of children per node, the number of objects per bounding volume are critical to the performance of the algorithm. This dependence may seem to be a liability but it may also be an advantage for the following reason. The performance of any ray tracing algorithm is dependent on the scene model. A user with a good understanding of the use of bounding volumes can thus tailor these volumes for efficiency.

2.2.3. Octree Subdivision

Glassner [GLAS84] has proposed a technique based on octrees for sub-dividing the modeling space into a hierarchical structure of subvolumes. Octrees allow dynamic recursive sub-division of the modeling space until each subvolume or *voxel* satisfies the termination condition. The condition or threshold is designed to to ensure that each voxel represents a uniform amount of work. The measure of work here is the number of objects that are wholly or partially contained in the voxel. The resulting voxel data organization allows the direct identification of the voxels lying along the ray's path.

The recursive sub-division of voxels begins by defining a cube which completely encloses the scene. This cube is the root of the hierarchical subvolume structure. The cube is divided into eight cubes or voxels each of which is tested for the termination condition. If a voxel fails the test, it is in turn subdivided and so on until all voxels have no more than the threshold number of objects. An example of the sub-division is shown in Figure 2-3.

Unlike the hierarchical data description described above, the hierarchy of voxels is in itself unimportant to the rendering process. There is no need to traverse a data tree. Only the

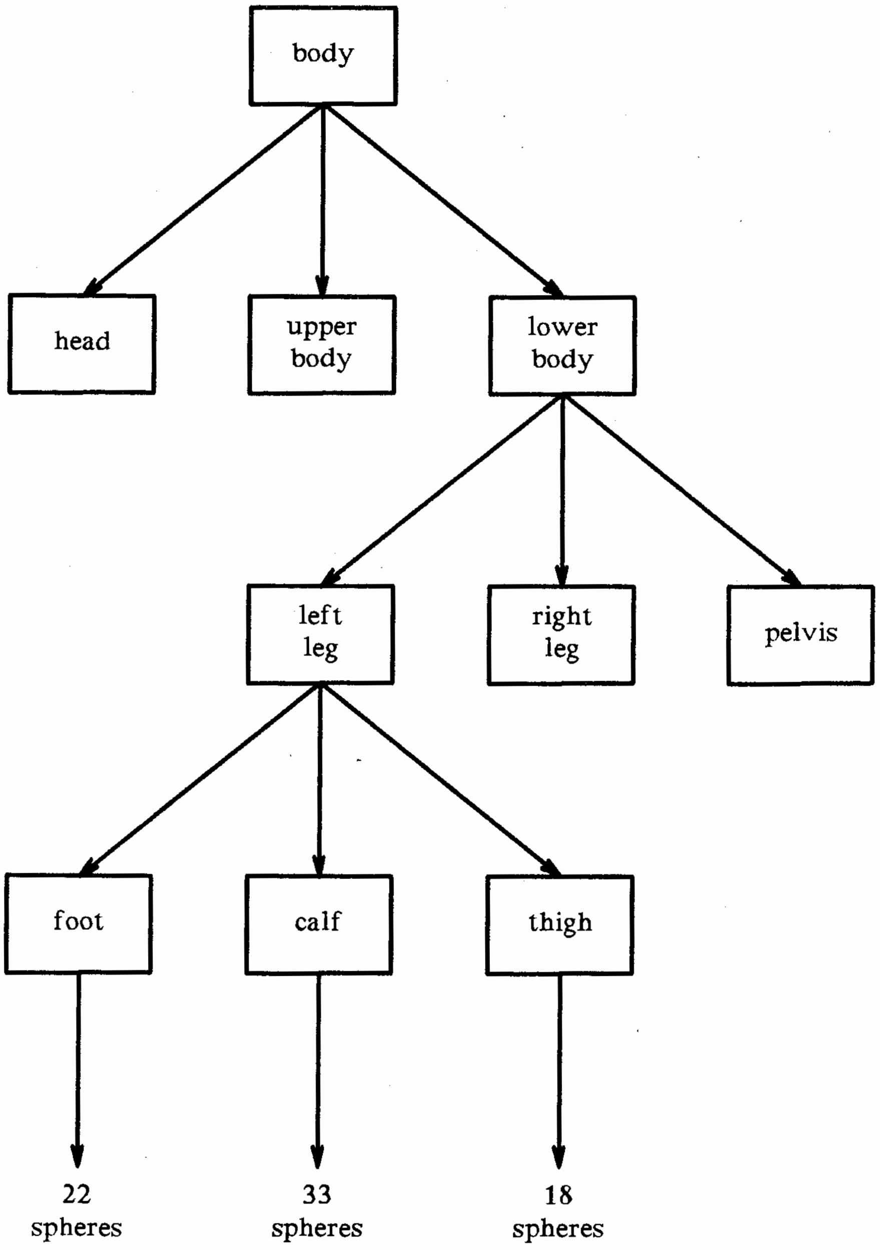

Figure 2-2: Example of a hierarchical data description

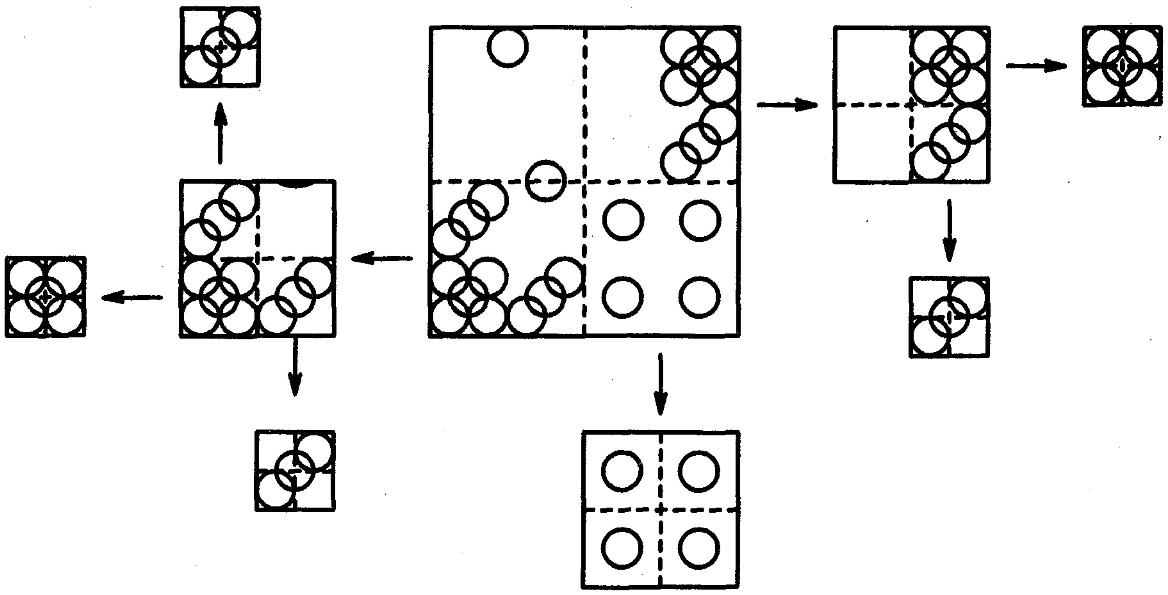

Figure 2-3: Examples of voxel sub-division

leaf voxels are kept along with their associated object lists. Using this structure, Glassner has proposed a method of quickly computing the transfer of a ray from one voxel to another When a ray is spawned, its first voxel intersection is computed. From there, if no intersections are found within the voxel, the next voxel along the ray's path is computed and the intersection test begin with it's children. Voxels are examined in same order that the ray encounters them in the modeling space. If an intersection is found in the current voxel, the ray need not be traced any further.

Published results using this approach have shown decreases in total rendering time of 70% to 90% compared to the standard algorithm.

This approach to eliminating object intersections is straight forward and elegant. It allows one to intersect only those objects associated with the voxels lying along a ray's path. It also gives the ray access to voxels in order of increasing distance, allowing termination of the tracing process if an intersection in found in the current voxel. However, there are potential weaknesses. The first is that the voxel threshold is based on the number of objects as opposed to the computational work required to process the voxel. A complex object could unbalance a voxel. Secondly, an object could span several voxels, necessitating several ray-object intersections for the same ray and object. Again, with complex objects, this could be a significant drawback.

2.2.4. Modeling Space Subvolumes

Another approach to reducing the number of ray-object intersections is modeling space subdivision [ULLN83] and [CLEA83]. Although developed primarily for parallel processor implementation, the technique itself is presented here within the context of a sequential algorithm. The concept is similar to octree subdivision in that the modeling space is divided into subvolumes where each subvolume has a list of objects that it wholly or partially contains. The difference is that the subvolumes are geometrically uniform subdivisions in two or three dimensions and are not recursively subdivided. The process of tracing a ray is similar to the process used with the octree subdivision technique.

Unfortunately in addition to having the same weaknesses as octree subdivision, modeling space subvolumes have an added disadvantage - there is no attempt to balance the workload associated with each subvolume. As mentioned, the algorithm's strength lies in its adaptability to parallel processing and, as such, it is discussed within that context in the next chapter.

2.2.5. Light Rays

The last technique discussed here has more to do with how a light ray is processed than with a more efficient search. The purpose of light rays is to determine if a direct light source is visible to the origin of the ray. If the light ray intersects *any* surface, the direct light source for which the ray was spawned does not contribute to the colour of the point and can be ignored. The search through the object list can then be stopped on finding the first intersection. Since light rays can account for 50% or more of the rays spawned, the potential reduction is significant.

2.3. Discussion

Improvements to the standard algorithm have been presented. Two techniques, adaptive tree depth and light rays, can be incorporated in any algorithm. On the other hand, a choice has to be made between octree subdivision or bounding volumes with hierarchical data structure. Unfortunately, published results do not use the same scene models, resolutions, shading models, performance measurements, and computers, making absolute comparisons difficult. Until someone publishes a good comparative study, the choice of algorithm must be made on different criteria, eg., which one has the least significant weaknesses.

Chapter 3

ARCHITECTURAL PERSPECTIVE

Ray tracing machines can be loosely classified into 3 classes based on the aspect of concurrency they exploit. The *intelligent pixel* machines exploit parallelism by distributing local intelligence to each pixel (or a group of pixels). This is possible since pixel computations are independent of each other. In the *intelligent object* class, processing power is allocated to each object. Thus, for a given ray, each object computes intersections in parallel. The *intelligent volume* machines subdivide 3D modeling space into subregions and allocate processing power to each region, which is now solely responsible for the objects that lie within its own volume.

In this chapter we shall the examine architectures that have been proposed or built specifically for ray tracing. We shall conclude with a discussion of the relative merits and drawbacks of the various architectures proposed.

3.1. Ullner's Machines

Ullner [ULLN83], in his doctoral thesis, proposes three different machine organizations. In the first approach, the intersection computation itself is massively pipelined to provide high throughput. In the second approach, which would fall under the *intelligent object* classification suggested above, each object is processed simultaneously. Finally, in the third approach, objects are separated into disjoint regions, and these regions are processed independently, thus following the *intelligent volume* approach.

3.1.1. The Ray Tracing Peripheral

As observed by Whitted and Rubin [WHIT80, RUBI80], most of the time in a ray tracing algorithm (70-90%) is spent in finding ray surface intersections. Therefore, if these intersection computations could be cast into hardware, one could significantly reduce the running time of the ray tracing algorithm.

Ullner proposed a ray tracing processor which acts as a peripheral to a host computer. The host computer fires rays at the peripheral which in turn returns the closest polygon intersected along with the intersection information. The ray tracing peripheral has its own copy of the scene model which besides reducing the load on the host's memory, also permits the model to be organized in a way that is suitable for intersection computation.

At the topmost level the ray tracing peripheral is organized as a three stage pipeline, see figure 3-1, each of which may be internally pipelined. The first stage fetches successive polygons from a scene model memory and passes their representations to a second stage, which performs the actual intersection. The third stage examines each new intersection and discards all but the the one closest to the origin of the ray. Note that the ray must be intersected against each polygon in the scene model before the closest one can be determined. Since most of the work must be done by the intersection stage, it may internally be pipelined, as shown in figure 3-2, to increase its performance. Applying stepwise refinement we can further internally pipeline each of the stages shown in figure 3-2 until we reach the level of the actual operators implementing the arithmetic.

Two potential problems need to be addressed at this point. In order to keep the pipe full, the polygon parameters used must be accessed in parallel. This is resolved by storing each of the twenty polygon parameters in one of twenty independent memories so that all

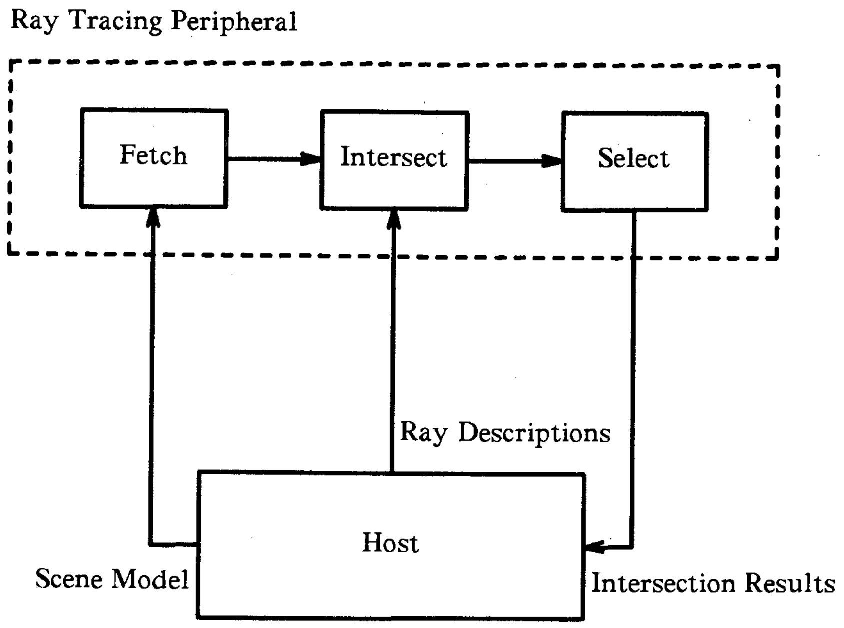

Figure 3-1: The three major pipeline stages in the ray tracing peripheral

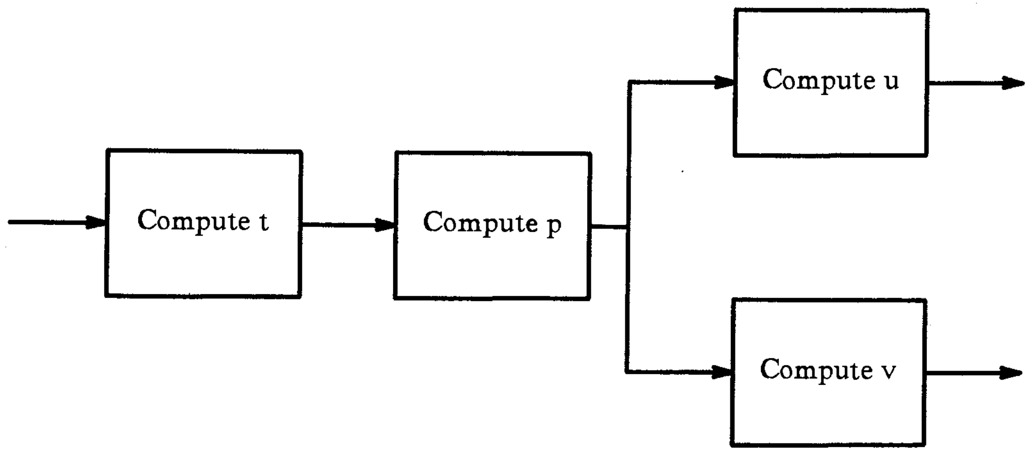

Figure 3-2: Pipeline stages within the Intersection Processor

may be accessed simultaneously. The second point is that an exception, such as in the divide operation, may be generated within the pipe, since the results may be undefined for some values of inputs. To resolve this Ullner associated a validity bit with each intermediate result flowing through the pipe. By convention, operations in the pipeline will always produce a result, but will mark that result to indicate its validity. Although later stages will accept these invalid values as if they were meaningful, the fact that their own results are invalid will be reflected in the validity bit of the output. The last stage in the pipeline takes into account the validity bit in determining the closest intersection.

All of Ullner's machines use floating point number representation which has a far greater dynamic range than fixed point numbers, freeing the user from having to pay much attention to scaling. Analysis of the ray tracing peripheral assumes that all the data operators in the pipeline are implemented using a parallel multiplier manufactured by TRW which is capable of producing a 48 bit product from two 24 bit operands in a maximum of 285 ns. Using the TRW multiplier, and a few "glue chips", a floating point multiplication takes about a third of a microsecond, but the other floating point operations cannot be completed so quickly. Each one of these operations may however be pipelined to operate at the same rate. Thus using this fully pipelined arithmetic the complete peripheral can produce three results every microsecond.

Using the above metric, we could make some estimates for the time required to generate a picture using the ray tracing peripheral. Assuming a scene model consisting of a thousand polygons, it would take a third of a millisecond to intersect a ray with each of these surfaces. In an image with 512 X 512 pixels of resolution, it would take a minute and a half to trace one ray per pixel. Of course, the number of rays increases if shadows are to be modelled and antialiasing is to be performed. Note that the time is linearly dependent on the number of polygons in the scene.

3.1.2. The Ray Tracing Pipeline

The ray tracing peripheral described earlier was not very extensible; it could not be easily enhanced to accomodate a more complex scene. The ray tracing peripheral has a single but fast intersection processor, but the intersection process has to be repeated for each polygon. Consider the other extreme now. If we had a less complex, and therefore slower, intersection processor, we could have many more of these processors working in parallel to achieve similar performance. The obvious advantage would be extensibility. The greater the number of these intersection processing units, which could be implemented as custom VLSI processors, the shorter would be the time for a more complex scene. Ideally, every object in the scene model could be attached to one of these processors typifying the *intelligent object* paradigm.

Based on the above principles, Ullner proposed the ray tracing pipeline which comprised intersection processors strung together to form the pipeline shown in figure 3-3. Each processor stores the description for a single polygon and it passes the description of rays through its input and output ports. On receiving a ray description the processors determine whether that ray intersects its stored polygon, and if so, locates the intersection point. Each ray is represented by a descriptor which has a field for the identity of the closest polygon encountered so far, and another for the *t value* of the polygon. The *t value* is initialized to infinity before entering the pipe. As it flows through the pipeline, each processor, on finding an intersection compares its *t value* with current *t value* in the descriptor field. If it is less, then that processor's polygon must be closer, and hence it swaps the identity of the polygon and the t value before passing it on through the output port to the next processor. Finally, when the ray descriptor leaves the pipeline it contains the identity of the closest polygon and corresponding t value.

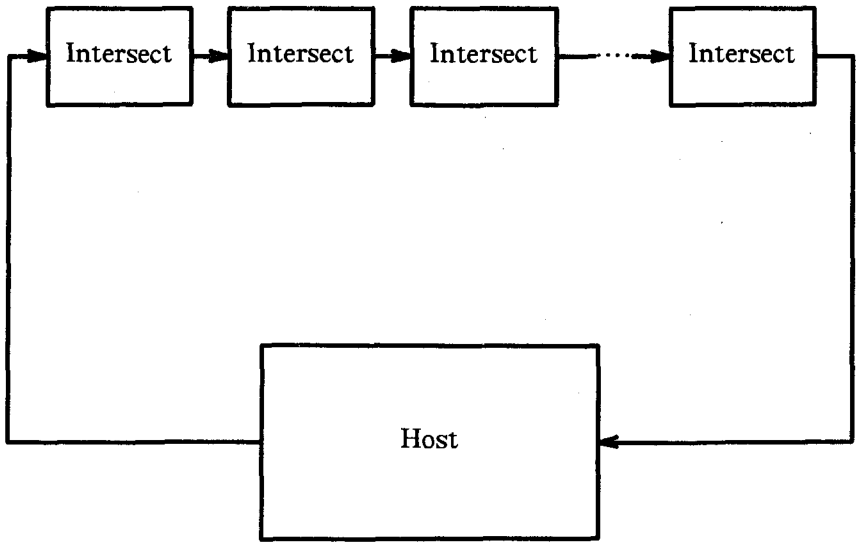

Figure 3-3: The Ray Tracing Pipeline

Since the ray tracing pipeline assumes the availability of low cost custom designed intersection processors, it would not be feasible to devote substantial chip area required to implement parallel multiplication circuitry to match the performance of the TRW multiplier used in the ray tracing peripheral. The alternative is to use a space effective, but slower, shift and add multiplier. Ullner estimates such an multiplier would perform a full 32 bit floating point multiplication in five microseconds, and also shows how other floating point operations can be implemented in the same area and speed.

Based on the above, we can conclude that the ray tracing pipeline can complete a ray tracing computation every five microseconds. Since Ullner estimates, for bit serial communication, the transmission time to be roughly five microseconds, we are still looking at a ray being processed every five microseconds. For a machine with a thousand processors, the latency would be 5 ms., and a 512 X 512 pixel image could be generated in 1.3 seconds assuming one ray per pixel.

3.1.3. The Ray Tracing Array

In the ray tracing array, a three dimensional grid is superimposed on the modelling space to section off the volume into a collection of subvolumes, each one of which has, at least in concept, a dedicated processor typifying the ***intelligent volume*** approach. Each of these processors is responsible for maintaining the surface models in its own subvolume, as well as for computing intersections of these surfaces with the rays passing through the subvolume. With such an arrangement one would expect a 3 dimensional lattice of processors, each connected to its six neighbouring processors. However, the cumbersome nature of wiring entailed by such an organization, acts as a major deterrent. Ullner overcame this problem by organizing the machine as a 2 dimensional array of processors with the third dimension of the partitioning grid simulated within each processor in the array. This structure allows each processor to communicate with its four neigbouring processors, as shown in figure 3-4. Each processor is also assumed to be a general purpose computing element since each processor should now be capable of carrying out shading computations, which in previous architectures were carried out in the host. Each processor also has some special purpose intersection hardware to aid in intersection computation.

The processors communicate with each other through messages. Each processor is responsible for a block of pixels corresponding to its position in the array and has an independent frame buffer used to store the pixel intensities. The different fields of the ray

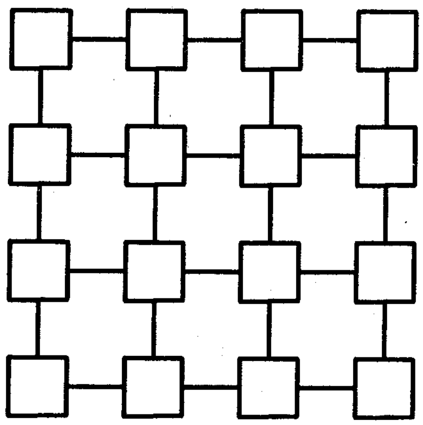

Figure 3-4: Organization of processors in a 16 processor ray tracing array

k	Message type (e.g. vision, shadow, etc.).
(r,c)	Row and column of pixel for this ray.
ro	Origin of this ray.
rd	Direction of this ray.
c	Color contribution of this ray.

Figure 3-5: Fields of a ray message

message are show in figure 3-5 Processors create initial ray messages for pixels that lie within their portion of the frame buffer. The processor then computes the closest subvolume which the ray enters, and then passes the ray message in the direction of the processor responsible for that subvolume. On reaching the destination processor, the ray is tested for intersection against all the objects within the subvolume. If no intersection is found then the processor incrementally computes the next closest subvolume which is handled by one of the four adjacent processors, and sends the ray message in that direction. If an intersection is found, a result message, which contributes to the intensity of its originating pixel, is passed off to the processor responsible for that pixel. Any secondary rays such as reflected, refracted or light rays are passed off to appropriate subvolumes for further intersection tests.

Cleary, et. al. [CLEA83] also proposed a similar processor array for ray tracing. They considered both square arrays and cubic arrays, and found that, in general, square arrays perform better than cubic arrays. A machine based on a 10 x 10 square array is currently under construction at the University of Calgary.

3.2. Dippe's Parallel Architecture

Mark Dippe & John Swensen [DIPP84], proposed an architecture for ray tracing which is quite similar to the ray tracing array proposed by Ullner, thus belonging to the *intelligent volume* family. The major difference between the two is that Dippe's parallel architecture allows for the subdivision of object space to be adaptively controlled, in order to maintain a roughly uniform load amongst the different processors. This turns out to be a serious drawback in Ullner's ray tracing array where no attempt was made to address the issues of uniform load distribution over the subregions. Uneven object distribution amongst different subregions can lead to load disparities between processors, causing computing power to be wasted. Therefore the ability to adaptively redistribute over time is crucial because load distributions are extremely difficult to calculate *a priori*, and hence must be done dynamically during the actual execution of the ray tracing process.

Since the operation of this parallel architecture is very similar to the ray tracing array, we shall concentrate on the dynamic load distribution aspect of this organization. The three dimensional space of the scene to be rendered is divided into several subregions which are initially assigned volumes more or less uniformly, and object descriptions are loaded into the appropriate subregions. As computational loads are determined, the space is redistributed among the subregions to maintain uniformity of load. Unlike the straightforward orthogonal subvolumes in Ullner's architecture, Dippe considered several different shapes for subregions. The choice of a subregion shape is influenced by the following criteria:

1. the complexity of subdividing the problem e.g. intersecting objects or rays with the boundaries.
2. the ability to subdivide space without splitting objects, and
3. the uniformity of the distributed loads attainable with the shape.

A strong candidate based on the abovementioned criteria would be "general cubes", which resemble the familiar cube, except they have relaxed constraints on the planarity of faces and on convexity. General cubes allow the most local control of subregion shape at the cost of slightly higher complexity of boundary testing.

The load information is shared among the neigbouring subregions, and this allows relatively more loaded subregions to reduce load by adjusting their boundaries. The load metric is primarily determined by the product of

1. number of objects and their complexity, and
2. number of rays

Load is transferred by moving corners of a subregion. Once the new position for a corner of a subregion has been determined, object descriptions and other information are redistributed to reflect the new subdivision.

Due to the subdivision, a speedup of the order of $O(S^{2/3})$ is expected by the authors, where S is the number of subdivisions of the object space. The parallel architecture is estimated to be three orders of magnitude faster than the standard algorithm with 125 computers working in parallel.

3.3. The LINKS-1 Multimicrocomputer System

LINKS-1 [NISH81] was an experimental machine which was built and tested at Osaka University in Japan. The system consists of 64 unit computers which are interconnected with a root computer such that a number of unit computers constitute a pipelined computer and such pipelined computers work in parallel, all controlled by the root computer. The number and length of each pipeline can be controlled dynamically, although it is not readily apparent how this dynamic reconfiguration would be useful. On the other hand the organization is general enough to be used for other image creation applications by means of more sophisticated parallel processing schemes which utilize different numbers of pipelines, perhaps with different lengths. Intercomputer program/data transfer is greatly facilitated by the use of a device called the intercomputer memory swapping unit (IMSU). LINKS-1 permits neighbouring unit computers to exchange data/programs using IMSU, and also between each unit computer and the root computer. There also exists a slow serial link between each unit computer and the root computer.

The root computer distributes the programs and data to be executed to the unit computers and the results are collected by the data collector. Each unit computer comprises five units:

1. the Control Unit for data transfer and communication control,
2. the Arithmetic Processing Unit for floating point calculations,
3. the 1Mb Memory Unit,
4. the I/O unit to be used as an outlet for debugging and monitoring,
5. the Intercomputer Memory Swapping Unit (IMSU).

The IMSU has two memory areas which are connected to a pair of control units through a bus exchange switch. Each of the control unit works independently on a memory area, and upon finishing they send a bus exchange signal which connects them to the other memory area.The IMSU is used to exchange program/data both between the root computer and the unit computers and also between two adjacent computers.

3.4. Discussion

Both the ray tracing peripheral and the ray tracing pipeline are, in a way, brute force approaches to the ray tracing problem, since they attempt to intersect every ray with every polygon. As noted in earlier chapters, techniques such as object space subdivision and bounding volumes can be used to significantly minimize the most computationally expensive operation — the ray surface intersections. The ray tracing peripheral, however, can be modified to use object space subdivision. The basic idea here is to superimpose a three-dimensional grid on the object space. The objects are then partitioned into these subvolumes. An extra stage is added to the pipeline which computes the subvolume which the ray intersects and passes the descriptor addresses of the polygons residing in the subvolume onto the next stage. Thus, the subsequent stages only have to compute intersections with a small number of polygons. No such arrangement is possible with the ray tracing pipeline since a separate pipe would be required with each subvolume.

The ray tracing pipeline is ostensibly fast, but on careful observation one quickly realizes that no general purpose host could keep up with it since it is unreasonable to expect a host to generate ray descriptions at this rate and deal with responses in the same time. Of course, one can design a special purpose host, sacrificing the flexibility offered by a general purpose host. It is also impossible for the ray tracing pipeline to process a scene with more objects than the number of processors in the pipeline. Note that this does not pose a problem for the peripheral since in the worst case all that needs to be done is to increase memory size. In case of the ray tracing pipeline, however, it becomes infeasible to increase the number of processors after a certain point.

Ullner's machines assume convex quadrilaterals as the basic modelling primitive. To achieve maximum performance, all intersection processors are dedicated to ray intersections with polygons. In computer graphics, however, it is often advantageous to model with alternative surface representations, such as bicubic patches, splines, quadric surfaces etc. The dedicated intersection processors are incapable of performing these intersections. On one hand, it appears in order to accommodate a variety of modelling surfaces, the intersection processors should be general purpose with fast floating point hadware to boost performance. On the other hand, we could tesselate most modeling surfaces into polygons and continue using dedicated intersection processors. Interestingly enough, there are devices available, such as the Weitek Transformation Engine [WEIT85a], which perform the tesselation functions with great speed.

The ray tracing array is probably the most promising approach of the three machines proposed by Ullner. Its chief drawbacks stem from the straightforward orthogonal subdivision of object space, which can cause immense disparity in object distribution among the subvolumes. Dippe's architecture takes care of this problem by using an adaptive subdivision approach. Also, for some choices of viewing position, not all processors are equally busy.

The Links-1 has a topology that allows work to be distributed by the root computer so that it can be performed independently in parallel, or pipelined from neighbour to neighbour, or some combination of both. This allows a variety of image creation algorithms to be used. But, the connection topology is restricted enough that any situation which demands substantial communication amongst the various unit computers would be almost impractical.

Chapter 4

A 3-TASK RAY TRACING ALGORITHM

In the previous chapters we discussed approaches for improving ray tracing performance by reducing the amount of computation and by increasing the speed of computation. As demonstrated in the modeling space subvolume approach, algorithms can be designed that directly map onto system architectures.

In this chapter we describe our modified ray tracing algorithm which maps directly onto a pipelined parallel processor architecture. To reduce the number of intersection calculations, our algorithm is based on bounding volumes and the hierarchical description of data. This approach also allows the tracing of a ray to be divided into three balanced tasks that map onto the pipeline architecture. In addition, the potential for parallelism lies in image space subdivision, where a pipeline can independently compute the value of a given set of pixels.

4.1. Definition of Terms

The following definitions are for terms used in this and following chapters. Some of the terms are similar to those used in [WEGH84].

contribution factor	factor which determines the contribution made to the pixel by the intensity found at the end of the ray.
data tree	the hierarchical description of the scene; its non-terminal nodes are parent shells and its terminal nodes, leaf shells.
initial ray	a ray originating at the eye and passing through a pixel on the image plane.

leaf shell	a shell which encloses primitives; whose children are primitives.
light	a geometric entity with an associated set of emittance characteristics.
light ray	a ray spawned on intersecting a reflecting surface in the scene; its origin is the intersection point and its direction is toward a specific light.
object	a geometric or procedural entity with an associated set of surface characteristics reflecting and possibly transmitting light.
parent shell	a shell which encloses shells; whose children are shells.
prim processor	performs the ray-primitive intersections.
primitive	an object or a light.
ray	a vector with a specific origin and direction.
reflected ray	a ray spawned on intersecting a reflecting surface in the scene; its origin is the intersection point.
refracted ray	a ray spawned on intersecting a transmitting surface in the scene; its origin is the intersection point.
scene	the uppermost parent shell in the hierarchical description; it has no parent shell.
shade processor	spawns initial and secondary rays; also computes the contribution a ray makes toward the final pixel value.
shell	a bounding volume.
shell processor	performs the ray-shell intersections.
t-value	a parametric value that defines a point on a ray where the ray intersects a surface.

4.2. Overview

Before delving into the details, we present a brief overview of the algorithm. An initial ray is spawned. This ray is tested for intersection against the nodes of the data tree in a recursive depth-first descent. If a parent node is intersected by the ray, all its children are in turn tested; if not, that branch of the tree is ignored. A list of all leaf shells intersected is generated and sorted in order of increasing *t*-value. The next step is to determine the closest primitive intersected. Beginning with the leaf shell closest to the origin of the ray, its child primitives are tested for intersection. If no intersection is found, the child primitives of the next closest leaf shell is tested and so on.

When an intersection is found, secondary rays are spawned. Using the surface characteristics associated with the intersected surface, the contribution each secondary ray makes to the final pixel value is computed and tagged onto the ray. Secondary rays are then processed in the same fashion as the initial ray. When all rays spawned for a pixel have been traced, the pixel value calculation is complete.

4.3. Features

Several features of our algorithm are important to its eventual mapping onto an architecture.

4.3.1. Data Tree

The data tree has two restrictions. The first of these is that all primitives must be enclosed within a leaf shell, either individually or within a collection of other primitives. Secondly, a parent shell can only have shells as children; a leaf shell can only have primitives as children.

4.3.2. Shell Shape

So far we have talked about shells without making any specific reference to the shape of the shells. The shape of the shell is an important issue, as discussed in [WEGH84]. We explored two of the possible alternatives for shells - spheres and orthogonal boxes. Orthogonal boxes have sides parallel to the axes of the modeling space coordinate system. In general, orthogonal boxes serve as better shells than spheres for the following reasons:

- In general, orthogonal boxes have less void area than spheres; they enclose their primitives more tightly. This increases the probability that a ray will intersect an enclosed primitive if it intersects the shell.

- The ray-shell intersection test is faster to compute. Note that if we only needed to know whether a ray hits or misses a shell, then spheres would be better since they require fewer floating point operations. If the exact point of intersection is also desired, then the intersection of a sphere, which requires computation of a square root, is slower.

Table 4-1 shows results that support the argument regarding shell shapes. The total rendering time is tabulated for a sample scene using the two shapes.

SPHERES	4162.01 secs.
ORTHOGONAL BOXES	2727.29 secs.

Figure 4-1: Total time taken for rendering a sample scene using spherical shells and orthogonal box shells

Another possibility is to use randomly oriented boxes, which potentially have less void area than orthogonal boxes. However, more overhead is associated with these boxes. The ray has to be transformed into the coordinate system of the random box and more data (the transformation matrix) must be stored. As we shall see later, in the context of our proposed architecture, the extra computations and the larger size of the shell data set could prove to be costly. Hence, orthogonal boxes represent a compromise between architectural demands and intersection efficiency.

4.3.3. Simplified Shader

The algorithm used a simplified version of the Hall shading model described in Chapter 1. The current algorithm does not trace rays through transparent surfaces. Fresnel reflectance and transmission curves and distance factors are also not implemented. Intensities and reflectance characteristics are represented using RGB triplets (a value for each of the primary colours - red, green and blue). The same RGB triplet is used for both specular and diffuse reflections. Using terms defined in Figure 1-2, our model is as follows:

$$\begin{aligned} I = & \sum_{j=1}^{l} [k_d(\overline{N}\cdot\overline{L}) + k_s(\overline{N}\cdot\overline{H})^n] R_d I_j \\ & + k_s R I_r \\ & + k_d R I_a \end{aligned}$$

Our algorithm and proposed architecture do not limit the complexity of the shading model. The reason for its simplicity has more to do with our emphasis on architecture.

4.3.4. No Intersection Tree

Although useful for describing the concept of ray tracing, intersection trees are not necessary in practice. Secondary rays are spawned to determine the intensities of various sources of illumination. The maximum contribution to the final pixel value that can be made by the intensity of a source of illumination can be computed. This contribution factor is calculated from the intersected surface characteristics and the intersecting ray factor. If a source of illumination does contribute, its intensity is multiplied by the contribution factor and the result added to the pixel value. To keep track of which ray belongs to which pixel, each ray is tagged with the pixel coordinates.

The advantage of this approach [ULLN83] is in removing the memory requirements and computation overhead associated with building and traversing intersection trees. This is especially important in the context of a VLSI processor pipeline.

4.3.5. Adaptive Tree Depth

Computing the contribution factor of a ray before it is traced enables us to use adaptive tree depth. If the factor is below a significant threshold, its contribution can be ignored and thus the ray need not be traced.

4.3.6. Primitives Types

Currently, the types of objects that our algorithm can render is limited to spheres and polygons. Work is currently underway to add fractals to the system. The algorithm is not really limited to those primitives and could easily be expanded to include other geometric or procedural primitives such as cylinders, cones, surfaces of revolution, prisms, and 3-dimensional curved surfaces.

4.3.7. Sorting Leaf Shells

Instead of performing a depth-first descent down to and including enclosed primitives, the algorithm initially tests only as far as the leaf shells. The intersected leaf shells are then sorted in order of increasing t-value (distance from the origin of the ray). In a strategy similar to that described for octree subdivision in chapter 2, the primitives enclosed by the nearest shell are tested for intersection. The closest surface intersected is identified. If such a surface is found, then the search is stopped; otherwise the primitives enclosed in the next closest shell are tested. This process is repeated until either a surface is intersected or no more leaf shells are left, implying that the ray does not intersect any primitive.

Unlike octree subdivision, hierarchical data organization may not produce disjoint leaf shells, i.e., shells whose volumes do not overlap. Fortunately, the above technique can be modified for use with overlapping shells. The t-value of an intersected primitive t_p is checked against the t-value of the next closest leaf shell t_s. If $t_p < t_s$, then the primitive is the closest. Otherwise the primitives in the next leaf shell must be checked.

Figure 4-2 illustrates this point. The two shells enclose exactly one primitive each.

Primitive A belongs to shell A and primitive B to shell B. Shell A is closer than shell B to the origin of the ray, i.e., $t_{shell\_A} < t_{shell\_B}$. Hence, primitive A would be tested for intersection first. Let us assume that the ray does intersect primitive A at t_A. However, as can be readily observed, primitive A is <u>not</u> the closest primitive (t_A is not less than t_{shell_B}). The primitives of shell B have to be tested before the closest surface can be identified. Here, primitive B is the closest primitive, although shell B is farther from the ray's origin than shell A.

This technique permits the identification of the closest primitive intersected without necessarily testing all the primitives in all the intersected leaf shells. Test results from rendering the scene in Figure 1-4 show that, on average, a ray tests the contents of only 80% of the sorted leaf shells.

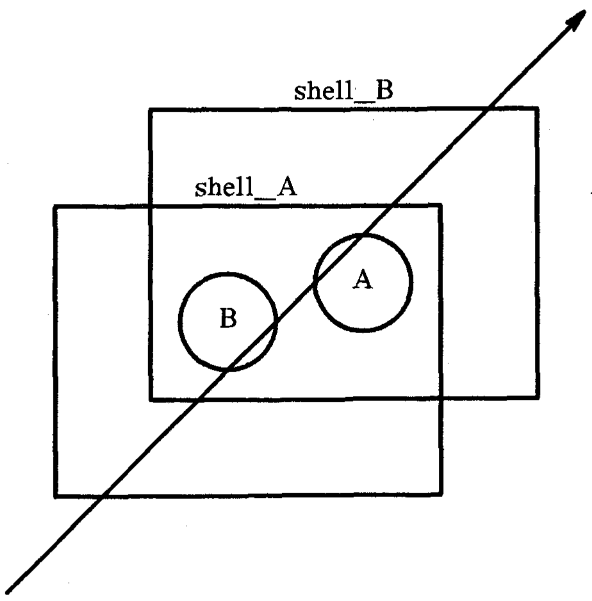

Figure 4-2: A 2-dimensional view of overlapping shells

4.4. The 3 Data Sets

Examining the data required by our algorithm, we can identify three disjoint data sets. This partitioning of the data also corresponds to the partitioning of the tasks described in the next section. The data sets are the shells of the hierarchical data description, the collections of primitives enclosed by the leaf shells and the different surface characteristics found in the scene model.

4.4.1. Shell Data

The basic element of the shell data set is the structure SHELL illustrated in Figure 4-3. The collection of shells making up the hierarchical data description is stored in an array called SHELL_ARRAY illustrated in Figure 4-4. The organization of data in this array retains the tree structure of the data tree. An entry in this array is a linked list of sibling shells, i.e., children of the same parent. The variable *leaf* indicates whether the shell is a leaf or parent shell. For a parent shell, the variable *child_index* is the index to its list of children. For a leaf shell, the variable is an index into the PRIM_ARRAY where the child primitives are stored. By convention, the index to the children of the scene or root shell is 0.

4.4.2. Prim Data

The basic element of the primitive dataset is the structure PRIM illustrated in Figure 4-5. The variable *type* indicates what type of primitive be it a sphere, polygon or whatever. The variable *p* is the union structure through which the geometric description can be accessed. The variable *surface_index* is an index into the SHADE_ARRAY where the surface characteristics associated with the particular primitive are stored. The collection of primitives making up the model description is stored in an array called PRIM_ARRAY illustrated in Figure 4-6. An entry in this array is a linked list of sibling primitives, i.e., children of the same parent.

```
typedef struct  shell   {
        int             leaf;
        int             child_index;
        COORD           max;
        COORD           min;
        struct  shell   *next;
        } SHELL;
```

Figure 4-3: SHELL data structure

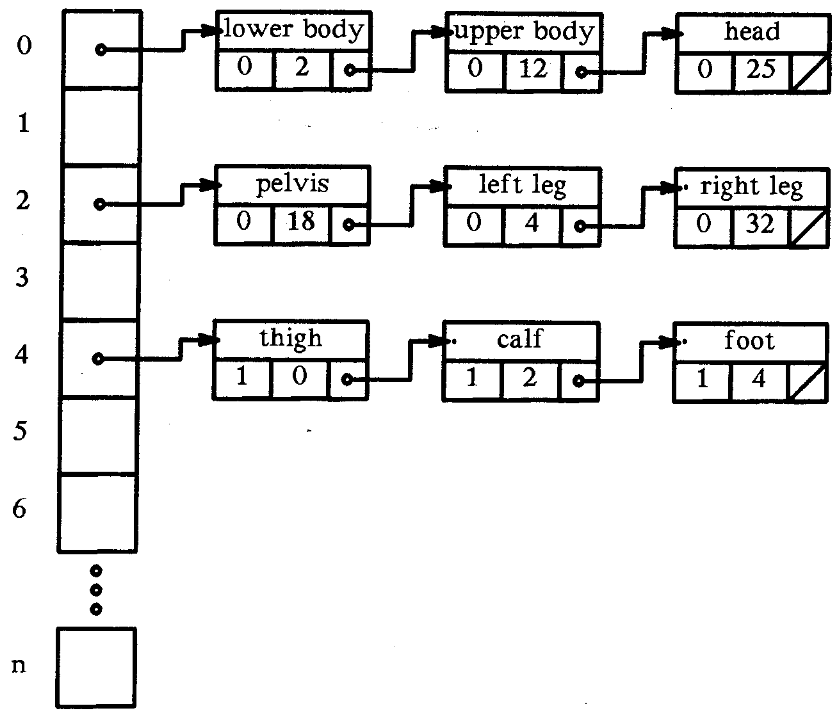

Figure 4-4: Illustration of SHELL_ARRAY

```
typedef struct  prim    {
        int             prim_id;
        int             surface_index;
        int             type;
        PTYPE           p;
        struct  prim    *next;
        } PRIM;
```

Figure 4-5: PRIM data structure

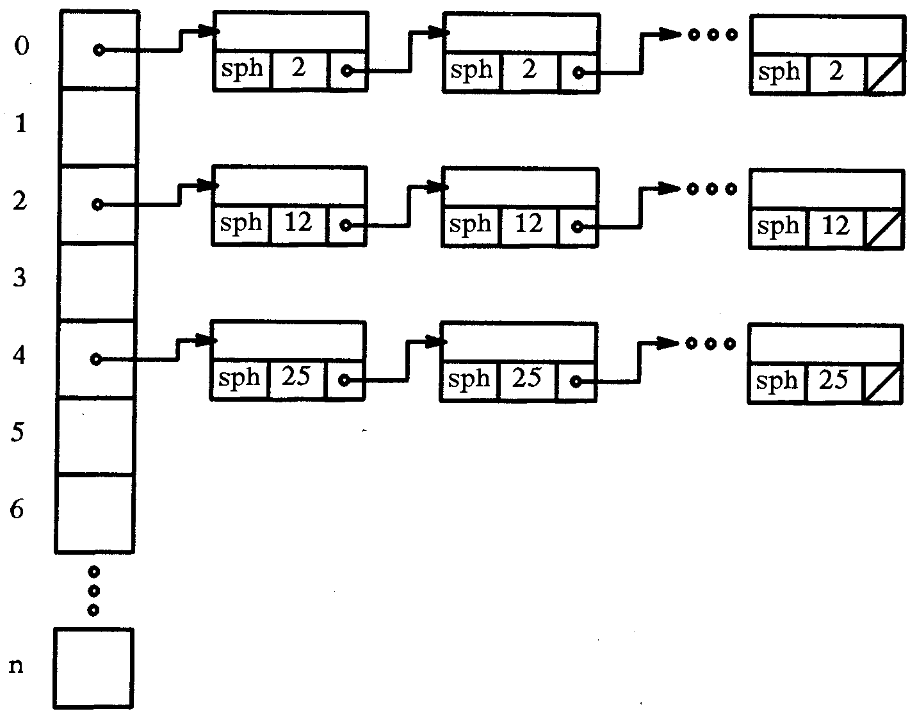

Figure 4-6: Illustration of PRIM_ARRAY

4.4.3. Shade Data

The basic element of the shade data set is the structure SHADE illustrated in Figure 4-7. Unlike the previously described arrays, the array for the shade data set is a simple array of SHADE structures. The variables *reflectance* and *transmittance* are triplets for red, green and blue values. Although the structure is designed for reflectance characteristics, emittance data can also be stored in the same structure by interpreting the *reflectance* variable as an emittance triplet and setting all other variables to 0.

```
typedef struct  shade   {
        float           k_s;
        float           k_d;
        int             n;
        RGB             reflectance;
        RGB             transmittance;
        } SHADE[];
```

Figure 4-7: SHADE data structure

4.5. The 3 Tasks

Our sequential ray tracing algorithm described above can be cleanly divided into the following tasks:

1. The first task spawns all the initial and secondary rays. It also computes the contribution factors that these rays make to the final pixel values.
2. The next task traverses the hierarchical tree with a given ray and makes up a sorted list of all the leaf shells intersected by the ray.
3. The third task intersects primitives contained in the leaf shells to compute the closest intersecting primitive.

In this section we shall outline each task's basic algorithm and the input and output data structures used by each.

4.5.1. The Shade Task

The first task, called ShadeTask, spawns rays for a given set of pixels. For each ray, an output data structure (illustrated in Figure 4-9) is filled and sent to the ShellTask described below. The variable *ray_type* indicates whether the ray is an initial, reflected or light ray. The coordinates of the pixel to which the ray belongs are found in *pixel_index* and the ray's contribution in *factor*.

When a ray returns to the ShadeTask after being traced, the combination of *ray_type* and what it hit, *hit_type*, determines the action to be taken. When a ray leaves the scene or when a light ray is blocked, the ray is ignored. Otherwise, if the ray is a light ray, the product of the intensity and *factor* is added to the pixel; if it is another type of ray, the product of the ambient intensity and *factor* is added to the pixel and new secondary rays are spawned. The algorithm is illustrated in Figure 4-8.

4.5.2. The Shell Task

The second task, called ShellTask, is outlined below in Figure 4-10. Receiving the structure SHADE_TO_SHELL as its input, the this task traverses the SHELL_ARRAY tree with the given ray. When a leaf shell is intersected by the ray, the child index and the *t*-value which defines the point of intersection are stored in the LeafShellList of the output data structure. When the traversal has been completed, the list is sorted on ascending *t*-values.

The output of the ShellTask is a structure similar to the one shown in Figure 4-11.

```
/****************************************************************************
Function: ShadeTask
Purpose : Spawn rays and compute contribution factors according to the
          shading model.
****************************************************************************/

ShadeTask ()

begin
if (light ray)
    begin
    if (self hit)  pixel += light intensity * factor;
    else           ignore ray;
    end
else
    begin
    if (no hit)  ignore ray;
    else
        begin
        pixel += ambient intensity * factor;
        spawn secondary rays and compute contribution;
        end
    end

if (pixel is finished)  spawn initial ray for next pixel;
end
```

Figure 4-8: The ShadeTask algorithm

```
typedef struct  {
        int             ray_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        } SHADE_TO_SHELL;
```

Figure 4-9: Output structure from ShadeTask

```
/*****************************************************************************
Function: ShellTask
Purpose : Produce a list of child indices and t-values (LeafShellList)
          of leaf shells intersected by the ray.
*****************************************************************************/

ShellTask (ix)

begin

/* Let S be the set of all shells pointed to by SHELL_ARRAY[ix] */

for each shell ∈ S
    begin
    if (the ray intersects the shell)
        begin
        if (leaf shell)  LeafShellList ← LeafShellList ∪ {child_index,tvalue};
        else             ShellTask(child_index of shell);
        end
    end

Sort LeafShellList on increasing t-value;
end
```

Figure 4-10: The ShellTask algorithm

```
typedef struct  {
        int             ray_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        LSS             LeafShellList [50];
        int             LeafShellCount;
        } SHELL_TO_PRIM;
```

Figure 4-11: Output structure from ShellTask

4.5.3. The Primitive Task

The third task, which we shall call PrimTask, receives the shell to prim data structure as input. This task executes exactly what has been described in the overlapping shell discussion above. The task proceeds to intersect primitives starting with the primitives enclosed in the closest leaf shell and stops on finding the closest primitive. It then also computes the information needed by the first task, the Shader Task, such as the surface normal at the point of intersection.

The detailed algorithm is show in figure 4-12. Note that in the actual implementation the algorithm treats different types of rays differently. For example, light rays need not find the closest intersection but any intersection will do. On the other hand, for initial and reflected rays, the algorithm goes through all the primitives in the given primitive list.

The output of the PrimTask is a structure similar to the one shown in Figure 4-13. The variables filled by the task when an intersection is found are *surface_index*, *point* that contains the coordinates of the intersection point and the surface normal at that point, and *hit_type* which describes what the ray hit.

```
/*****************************************************************************
Function : PrimTask
Purpose  : To compute the nearest primitive.
Note     : 1. LeafShellList comes from the ShellTask.
           2. Indices in the set LeafShellList are accessed in order i.e.
              we get the element with the least t-value first.
*****************************************************************************/

PrimTask()

begin

for each index ∈ LeafShellList
    begin

    /* Let P be all Primitives pointed to by the current index. *.

    find the nearest_primitive ∈ P;
    if (t-value of nearest_primitive
                 < t-value of next index in LeafShellList)
        begin

        /* we have found the nearest primitive */

        found = TRUE;
        break;
        end
    end

if (found) compute info (intersection point, normal, surface_index);
else       report no hit;
end
```

Figure 4-12: The PrimTask Algorithm

```
typedef struct  {
        int             ray_type;
        int             hit_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        INTER           point;
        int             surface_index;
        } PRIM_TO_SHADE;
```

Figure 4-13: Output structure from PrimTask

Chapter 5

A PARALLEL ARCHITECTURE

PERT — Pipelined Engine for Ray Tracing — is a pipeline of three processors each of which executes one of the three tasks described in the previous chapter. PERT can be used in two different configurations: a) in a *single-PERT* configuration, where each of the 3 processors has access to an independent memory module that stores the appropriate data set, and b) in a *multi-PERT* configuration that consists of an interconnection of n PERTs working in parallel.

In the multi-PERT configuration, the question of access to data sets is not so easily resolved as with a single-PERT configuration. There are two extreme approaches to this question. One is to provide each processor in each PERT with its independent memory module as with the single-PERT configuration. The disadvantage of this approach is in its inefficient use of hardware — individual memories for the same processors in each PERT store exactly the same data set. This results in n duplicate data sets in $3n$ memories. The opposite approach is to have a global memory for each of the data sets that is shared by all the processors. The communication overhead and the memory contention that would result from this approach is dependent on the number of PERTs in the system; the more PERTs, the greater the problem. This is not a desired approach when considering a multi-PERT architecture. What is needed is a way of allowing the PERTs concurrent access to a global shared memory without the communication overhead.

In this chapter, we describe a parallel architecture for ray tracing using PERTs that allows the desired concurrent access to the data sets by cyclically broadcasting these to each appropriate processor. The basic building blocks of this architecture are PERTs, broadcast processors (BP) and bus interface controllers (BIC).

5.1. Broadcasting

Broadcasting provides a means of allowing concurrent access by many processors to a data set. Broadcasting was chosen as a solution for the following reasons:

- it avoids the need to duplicate data sets;
- it avoids the need for large independent memory modules for the PERT thus allowing a PERT to have a reasonably small board size (important in a multi-PERT configuration).
- A one-way data flow on the broadcast bus keeps the communication overhead low and constant for a given data set.
- By keeping the communication overhead constant, we could expect a linear increase in performance with the addition of extra processors.

5.1.1. Description

To illustrate the concept of *broadcasting*, we draw on the following analogy. Assume we have a read-only disk subsystem and think of the output of the read head as a (single line) bus to which several processors are attached as illustrated in figure 5-1. Let us further assume that our hypothetical disk has only one track and the read head is permanently positioned over it. Now, what appears on the bus is a bit-stream of data organized in blocks that is repeated periodically owing to the circular nature of the track containing the bits of information. Each processor has access to any block in the stream, but the access is sequential as opposed to being random. Thus, associated with each block access, is a potential latency delay. We shall herewith refer to such a periodic transmission of data over a bus as *broadcasting*, the bus, which is the broadcast medium as the ***broadcast bus***, and the time taken to cycle through the entire set of data as the ***broadcast cycle time***.

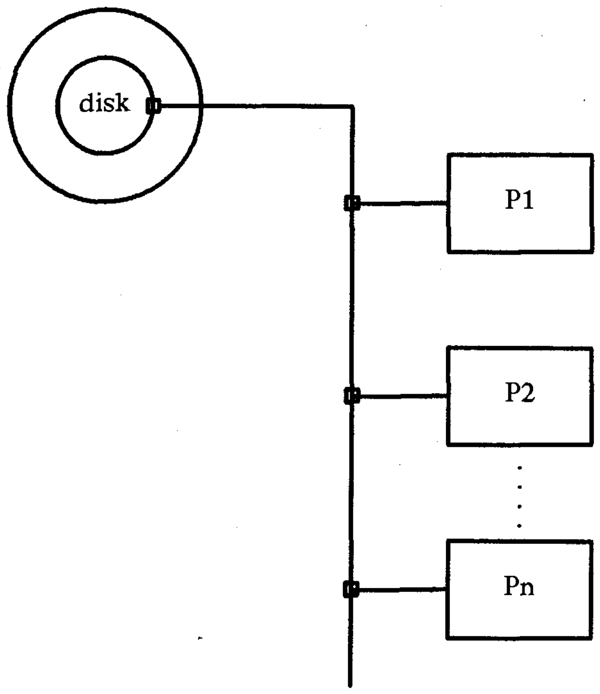

Figure 5-1: Example to illustrate broadcasting

In reality, the processors in our analogy are PERTs, the function of the hypothetical disk is taken over by fast broadcast processors that have access to the global memory and the data being broadcast is organized in packets. The broadcast processors transmit these packets at high speeds over their broadcast busses. Speed is a critical issue here, since the slower the broadcaster, the greater the access latency. Each of the PERTs can now, irrespective of the others, access a packet off the bus as needed, without any contention for memory. The penalty is the access delay owing to the cycle latency.

5.1.2. Adaptive Broadcasting

As data sets get larger, the broadcast cycle time lengthens and the average latency time increases to the point where a processor spends most of its time waiting for data. To reduce the latency time for a given data set, adaptive broadcasting was developed. Adaptive broadcasting is a process whereby processors indicate to the broadcast processor whether a given packet of data is required by any one of them. If no, then the packet is not broadcast. The result is a variable length broadcast cycle and a much reduced average latency time. Adaptive broadcasting will be explained in detail in the following sections.

5.2. The PERT

PERT is a pipeline of three processors connected cyclically as illustrated in figure 5-2. This architecture is a direct map of the ray tracing algorithm described earlier, with the three processors performing the three tasks — the ShellProcessor performing the ShellTask, the PrimProcessor performing the PrimTask and the ShadeProcessor performing the ShadeTask. The organization deviates from the classical Von-Neumann architecture, since three instruction streams are concurrently active on three independent data sets, and hence would be classified as a MIMD organization under Flynn's [FLYN66] taxonomy.

The three processors of PERT are hardware embodiments of the three tasks of the ray tracing algorithm. Since the operation of the ray tracing algorithm has been covered in great detail in chapter 4, and the operation of PERT is identical, it will not be discussed here. Figure 5-3 shows the internal organization of the processors. We shall now briefly discuss the various modules comprising each processor. A more detailed explanation of the design and performance of the PERT is contained in [CHIL85].

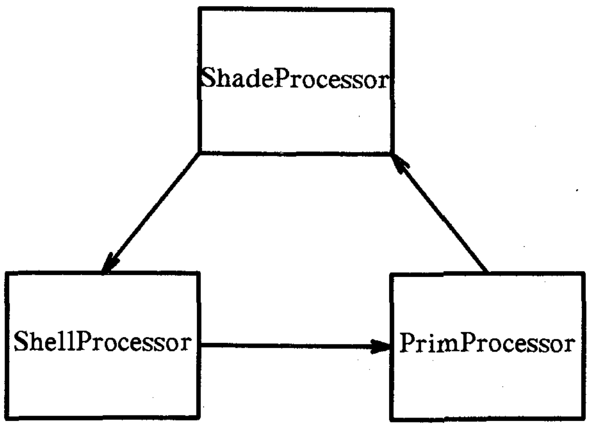

Figure 5-2: Block diagram of PERT

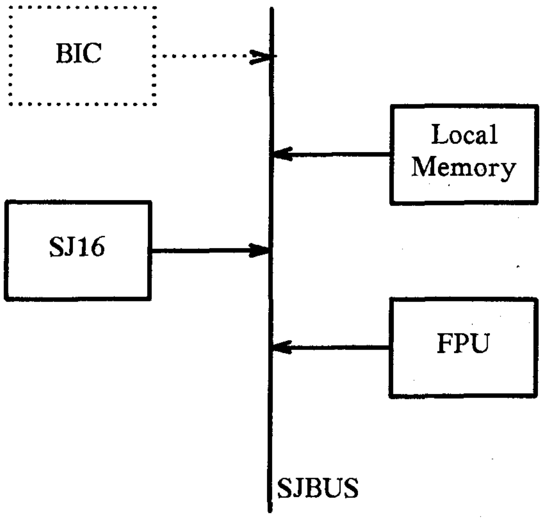

Figure 5-3: Detailed block diagram of a processor

5.2.1. The SJ16

SJ16 is a 16 bit microprocessor intended for use as a hardware building block for multiprocessor systems [HOBS81]. SJ16 is fabricated as a single chip VLSI processor and is currently being tested at Simon Fraser University. It was a natural processor choice: it is microprogrammable and it has excellent hardware features. Since microcode development for the various task algorithms was a key issue in the PERT design, another attractive feature of SJ16 was the microprogramming environment — the Architecture Support Package (ASP). Details of microcode development for SJ16 can be found in [HOBS82].

5.2.2. The Floating Point Unit

The floating point unit (FPU) is capable of fast execution of floating point operations. For simulation purposes, this special function unit was modeled around the Weitek WTL1164/1165 low-latency floating point chip set [WEIT85] capable of executing floating point operations with speeds above 2.78 Mflops. Recalling the voracious appetite of the ray tracing algorithm for floating point computation, one can see that the high throughput of the Weitek chip set makes it a prudent choice.

5.2.3. The Memory Module

The memory module provides independent storage for each of the three processors. In the multi-PERT configuration described here, each processor requires a minimal amount of memory for global variables, stack space, etc. This contrasts with the single-PERT configuration where the memory module would be large enough to hold an entire data set. Reads and writes to the memory can be *streamed*. The memory controller buffers data words and hence after the first access, memory can be accessed sequentially in a single cycle.

The ShadeProcessor is a special case. Its memory module could include the part of the frame buffer corresponding to the image space subdivision assigned to the PERT. Intensity values from each ray can be added to the appropriate pixel in the frame buffer. This minimizes communication overhead once the rendering process is initiated.

5.3. The Broadcast Processor

Basing the broadcasting scheme on the adaptive process requires the broadcasting hardware to decide when to broadcast a packet, to find that packet in memory and to return to an ID list in memory. What is needed is a processor. Figure 5-4 illustrates the configuration of the Broadcast Processor (BP). The main components are the processor itself with an independent memory module and a collection of communication lines, the 32-bit *broadcast bus*, the *sync line* and the *hit line*.

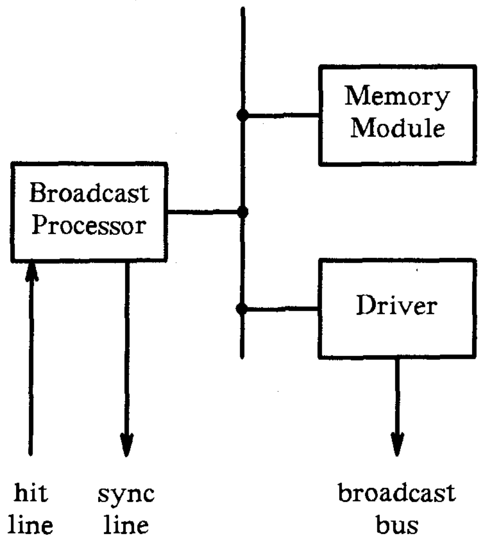

Figure 5-4: Block diagram of the BP

5.3.1. The Memory Module

Similar in function to the memory modules of the single-PERT configuration, the module in the BP stores the complete data set be it the ShadeArray, the ShellArray or the PrimArray as described in Chapter 4. Also, similar to memory modules for all PERTs, the memory controller allows the read operation from the memory to be streamed.

The way a data set is organized and stored is important to the process of adaptive broadcasting. The data is organized in *packets* each with an ID. Contained within a packet are all the data associated with an index into the array — the list of sibling shells or the collection of primitives enclosed by the same volume or the reflectance characteristics of a surface. The ID that identifies the packet is the index into the corresponding array.

Figure 5-5 illustrates how the data sets are stored. The first structure occupying successive memory locations is the ID list. With each ID the size and address of its packet are also stored. The BP broadcasts an ID to determine if any PERTs require the packet. If yes, the BP needs to know where the packet is stored and how large it is. The contents of each packet are also stored in successive memory locations. The reason for storing collections of data in this fashion is to take advantage of the streaming capability of the memory controller.

5.3.2. The Processor

The processor is modeled on the SJ16 but with a 32-bit internal bus. We assumed a 200 nanosecond processor cycle time and a broadcast rate of twice the cycle time — 400 nanoseconds per word. The broadcast rate is considered to be conservative. However, the speed of a BIC's comparator and the reasonable delay required for the response on the hit

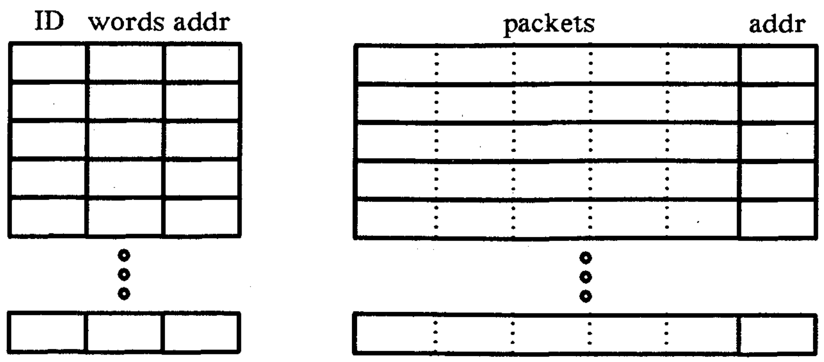

Figure 5-5: Data organization in memory

line are ultimately the limiting factors. There is also the problem of the signal propagation delay associated with a physically long broadcast bus.

The task of the BP is a straight forward one. Once the memory module has been loaded with the data set and the rendering process initiated, the BP executes the algorithm detailed in pseudo-microcode in Figure 5-6.

```
loop0:  set address register to start of ID list
loop1:  fetch next ID from list and broadcast
        no operation
        fetch packet address and store in register
        fetch number of bytes in packet and load count
        if ~hitline, goto check
        start packet stream access
loop2:  fetch word and broadcast
        if ~count, goto loop2
check:  if end of ID list, goto loop0
        goto loop1
```

Figure 5-6: The BP algorithm

5.4. The Bus Interface Controller

The processors within a multi-PERT machine are not connected directly to the broadcast busses, but are connected through a device called the Bus Interface Controller (BIC) illustrated in Figure 5-7. The advantages of using the BIC are:

- it serves as an I/O processor for SJ16 by relieving it of data collection chores;
- it operates in parallel with SJ16 to reduce the total processing time;
- it reduces latency since it has multiple ID-registers and looks for a match with any one ID contained in the registers and because it has a double buffer; and
- it enables adaptive broadcasting to be used because of the hit line feedback to the BP that indicates if a packet is required by the PERT.

The 3 major functional components of the BIC are described below.

5.4.1. ID Registers

When the SJ16 needs to process a particular data packet, it serially searches the ID registers of the BIC for an available register that is then filled with the ID number of the desired packet. Each processor in the PERT uses its registers differently. The ShadeProcessor stores at most one ID per ray processed. The PrimProcessor stores at most 2 IDs at a time in the BIC until it has processed all the packets required for a given ray. The ShellProcessor continually stores IDs in the BIC and may potentially overflow. Therefore, the ShellProcessor needs to be able to stack overflow IDs in its memory until an ID register becomes available.

When a packet is received into a buffer, the ID register containing the ID is flagged as being available.

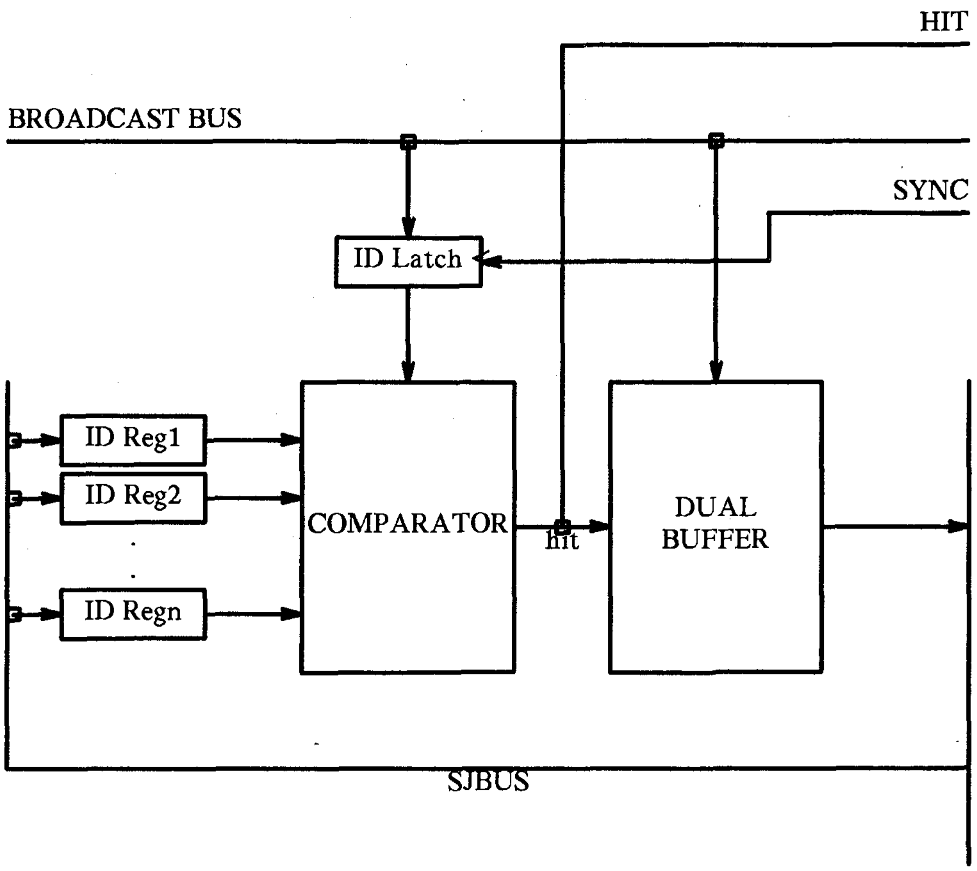

Figure 5-7: The Bus Interface Controller

5.4.2. The Comparator and Hit Line

When an ID is broadcast, the BP sends a pulse on the sync line that causes the word on the bus to be copied into the ID latch. The comparator compares the ID with the contents of all valid ID registers in parallel. If there is a match and a buffer is available, the hit line is set high to flag the BP to send the packet. The hit line also sets the available buffer to start copying the data from the broadcast bus.

5.4.3. The Buffers

The BIC is equipped with a double buffer that is used in two ways. When a desired packet is broadcast, it is copied off the broadcast bus into a buffer. Secondly, the SJ16 uses the buffer as memory when processing the packet. The buffers are FIFOs to allow the SJ16 to read their contents in one cycle. The double buffer allows the BIC to operate in parallel with the SJ16 when the SJ16 is using one of the buffers as memory.

5.5. System Organization

Figure 5-8 shows the system organization of the multi-PERT configuration. The PERTs are each connected to each of the three broadcast busses as illustrated. The control processor shown coordinates the function of the machine. It pre-processes the scene data, downloads the data sets into each BP's memory, downloads the global variables required for spawning rays to each of the ShadeProcessors and initiates the rendering process.

As intensity values for rays and pixels are computed, they must be added to the frame buffer. The obvious approach is to have all the ShadeProcessors communicate these values to the control processor which in turn loads them into the frame buffer. Unfortunately, this is another potential communication bottleneck. One way to minimize this overhead is to have the ShadeProcessors communicate directly with the frame buffer. Memory contention for the frame buffer is avoided by partitioning it and providing each PERT with an independent portion.

The distribution of the workload among PERTs is important to the efficiency of the machine. There are many ways to subdivide the image space among PERTs the best being allocation based on PERT work load. However, if the direct frame buffer approach mentioned above is used, the subdivision must be predetermined since each PERT must be

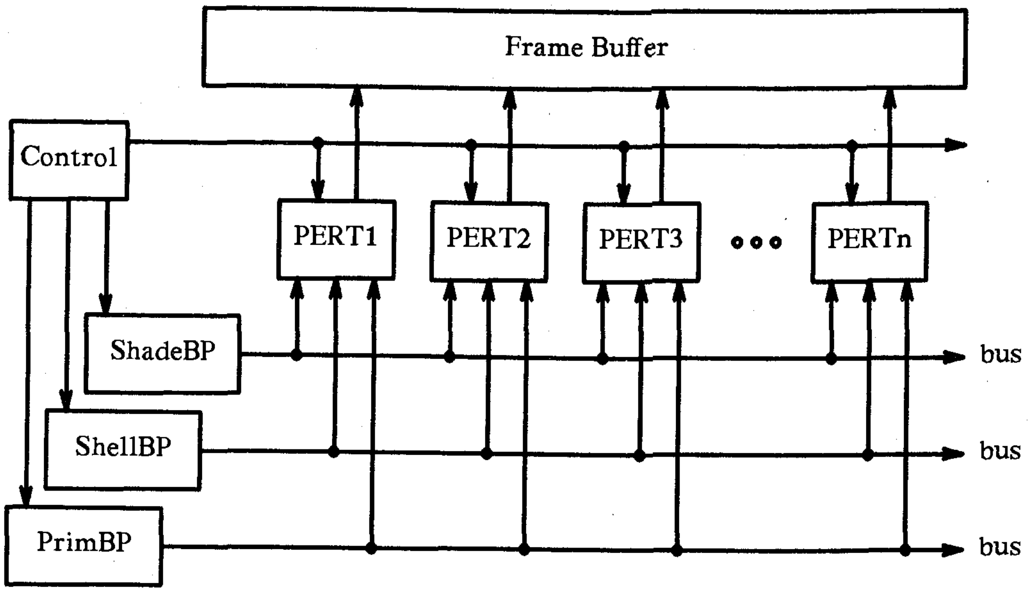

Figure 5-8: The multi-PERT configuration

directly connected to a part of the frame buffer. We have chosen to model the allocation of sets of scanlines to each PERT. For *n* PERTs in a machine, PERT$_i$ is assigned the *i*th scanline from the top of the image plane and then *leapfrogs* its way down the image plane, processing every *n*th scanline thereafter.

Chapter 6

RESULTS AND CONCLUSION

Several *layers* of simulators were designed and implemented to model the key features of the architecture as described in Chapter 5. A series of scenes were created to test different parts of the algorithm. The results are presented in two parts — the evaluation of adaptive broadcasting and the analysis of latency in the multi-PERT configuration. A discussion of the overall architecture and the conclusion close the chapter.

6.1. Simulation

6.1.1. The Simulator

Several simulators were designed and implemented with two objectives in mind — to evaluate the performance of adaptive broadcasting and to predict the performance of a multi-PERT machine. There were three key features of the architecture which had to be modeled accurately to allow for valid results. These were the SJ16 timings, the pipelined configuration of the PERT and the three broadcast processors.

The SJ16 with FPU was simulated in the ASP environment [CHIL85]. The algorithms as described in Chapter 4 were in large part microcoded and the operations timed. The timings were based on a 5 MHz. processor.

The algorithms were also written in C and implemented as separate programs on the VAX. The timings from the ASP environment were coded into these programs such that the timings returned were SJ16 timings. The pipeline was simulated using IPC sockets under UNIX. Times were passed between processors to ensure the proper simulation of the pipeline. In this way, a single-PERT was modeled.

To simulate broadcasting, an event-driven broadcast module was added to each to the programs above. The operation of the BPs was based on the following timings:

- 1000 nanoseconds to broadcast an ID.
- 800 nanoseconds overhead for a packet.
- 400 nanoseconds per packet word.
- 400 nanoseconds to reset the loop.

Lastly, a renderer also written in C and based on the same algorithms but running sequentially was implemented and run on a VAX with a floating point unit. No ASP timings were included in this version This allowed us the compile run time profiles for the algorithms on the VAX while rendering the same scenes as the simulator.

An important aspect of the simulators running on the VAXs was to actually render scenes. Timings for any architecture or algorithm performance are highly dependent on the nature of the scene. One can obtain very good results if the data set is small or if there is a lot of void space in the scene. By rendering the same scenes in the different simulators, we could be sure that we were comparing apples to apples.

6.1.2. Test Scenes

To evaluate adaptive broadcasting, a series of scenes were created and identified with a number of the form *npp*. All these scenes had 64 clusters of randomly placed primitives. The *pp* of the number indicates the number of primitives (spheres) in each cluster. The *n* indicates the order of the complete hierarchical tree be it binary (2), quad (4) or oct (8). For example scene 804 means that there are 4 spheres per cluster and the hierarchical tree is a complete octree. Also, scenes 204, 404 and 804 look the same because they share the same primitive data.

To predict multi-PERT performance, more complex and *realistic* scenes were created. The figures in scenes 23 and 45 are bubble figures generated by the SFU kinematic simulation system [CALV82]. With their large number of primitives, these scenes would demonstrate if adaptive broadcasting could cope with the demands of the multi-PERT machine.

Figure 6-1 details the characteristics of each scene. Pictures of the test scenes are in figures 6-2 to 6-8.

6.2. Broadcast Performance

The first step of the simulation was to evaluate the effectiveness of adaptive broadcasting. All simulations were run on the single-PERT simulator with broadcasting.

	number of shells	number of shell packets	number of primitives	number of primitive packets
scene 204	126	63	256	64
scene 408	84	21	512	64
scene 812	72	9	768	64
scene 816	72	9	1024	64
scene 820	72	9	1280	64
scene 23	211	61	1093	152
scene 45	454	138	2754	319

Figure 6-1: Characteristics of the test scenes

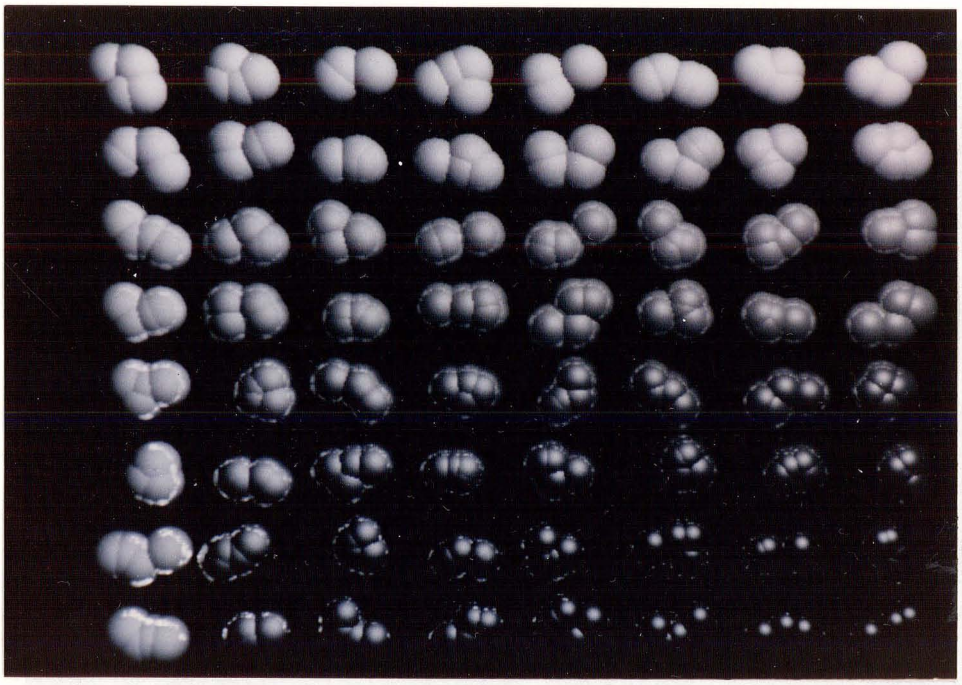

Figure 6-2: 256 primitives (4 per leaf shell)

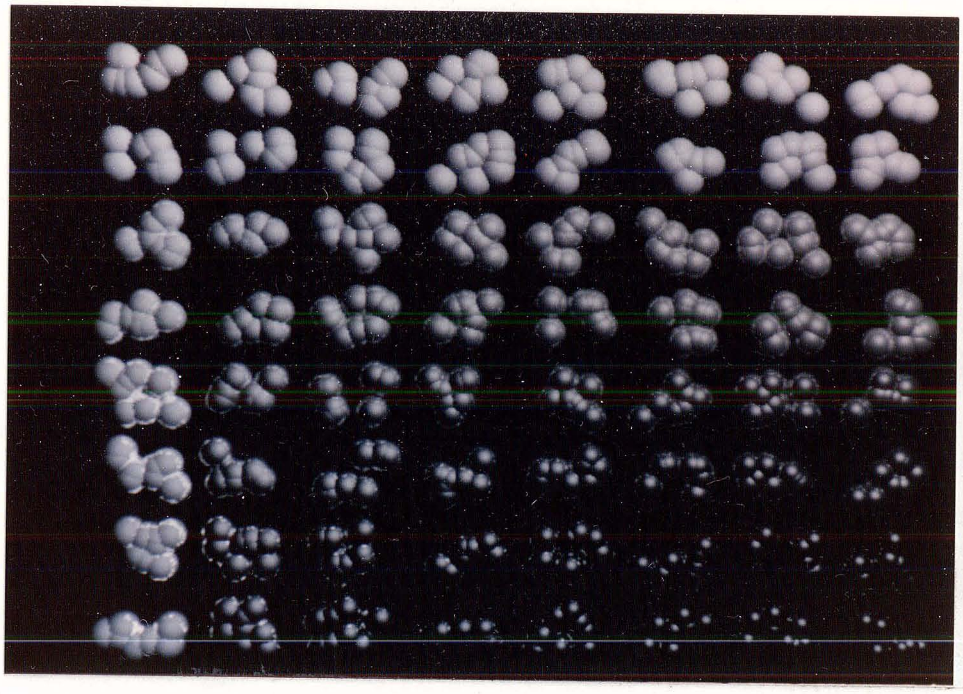

Figure 6-3: 512 primitives (8 per leaf shell)

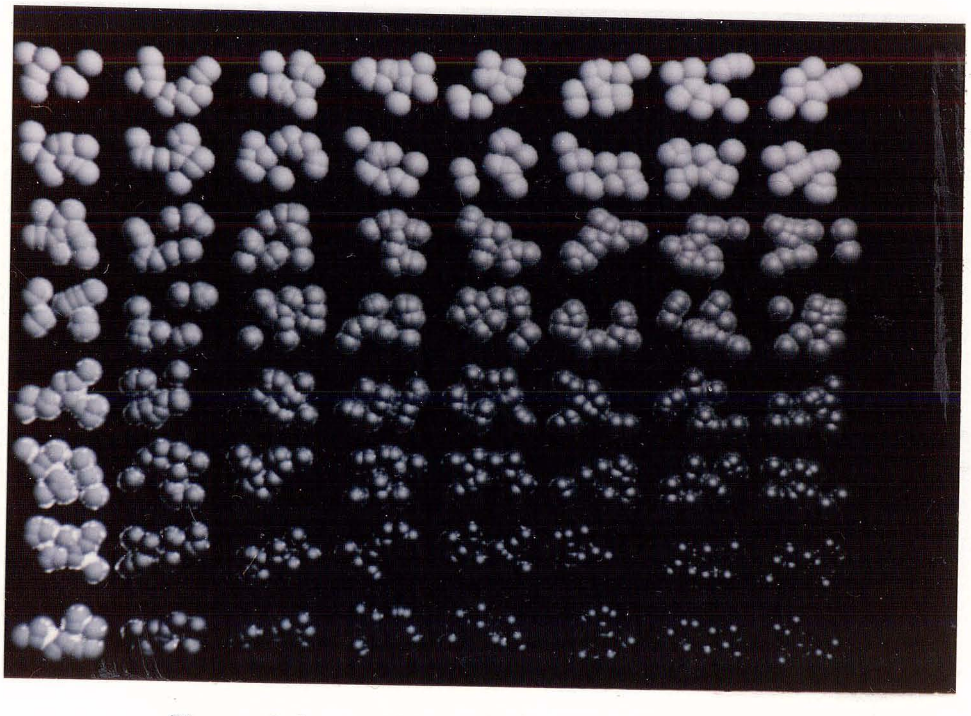

Figure 6-4: 768 primitives (12 per leaf shell)

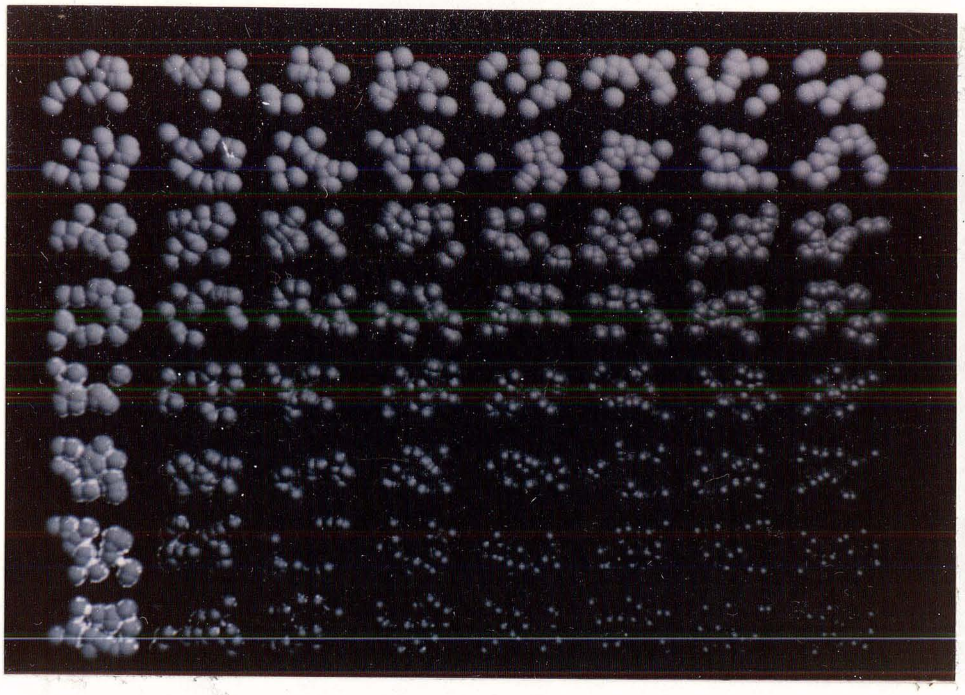

Figure 6-5: 1024 primitives (16 per leaf shell)

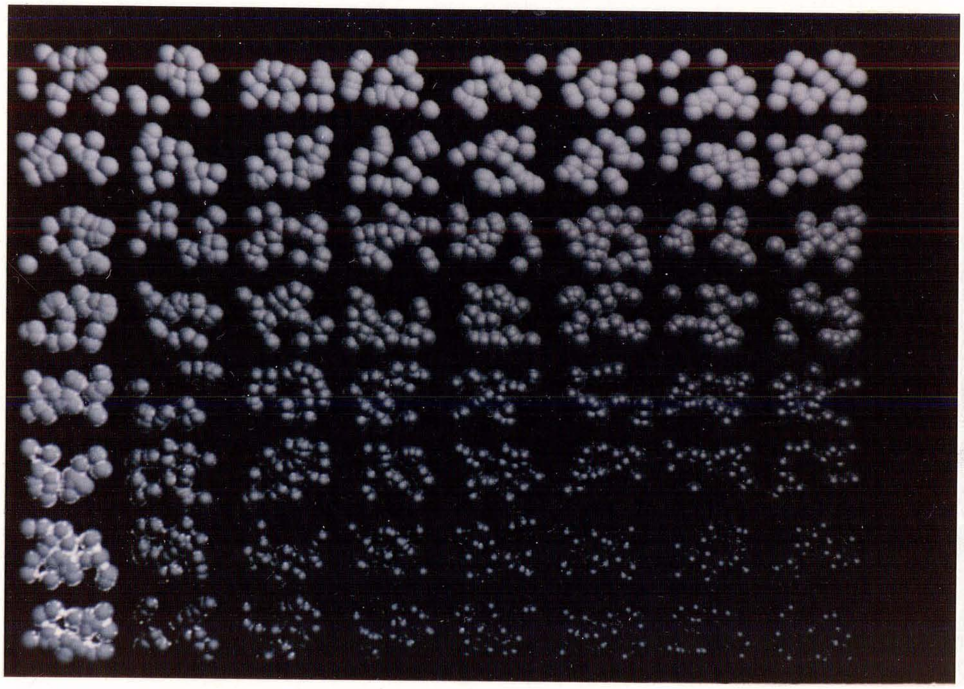

Figure 6-6: 1280 primitives (20 per leaf shell)

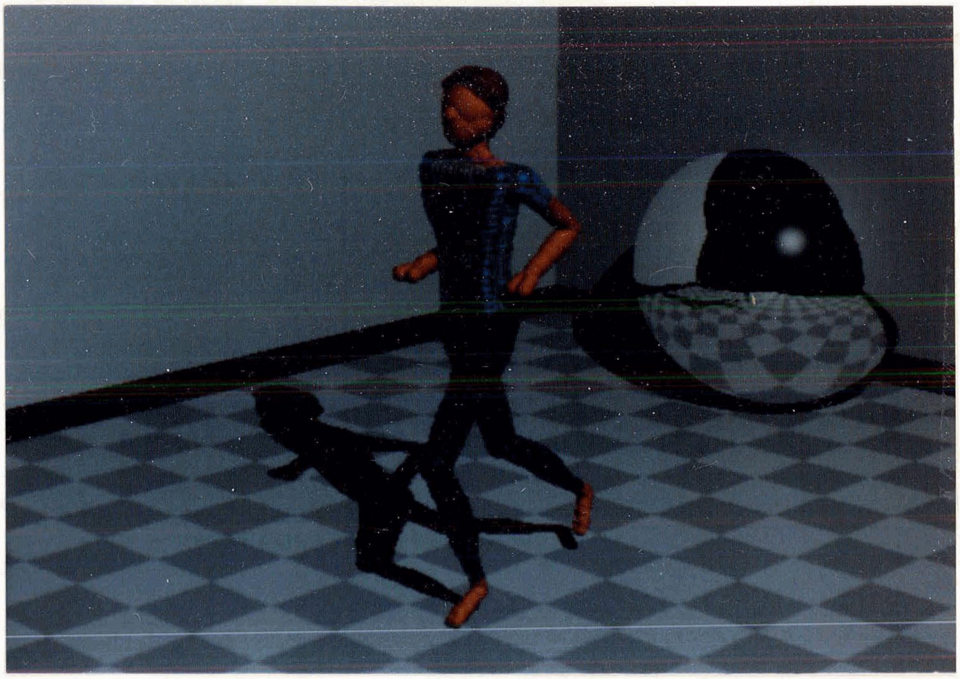

Figure 6-7: 1 jogger with ball

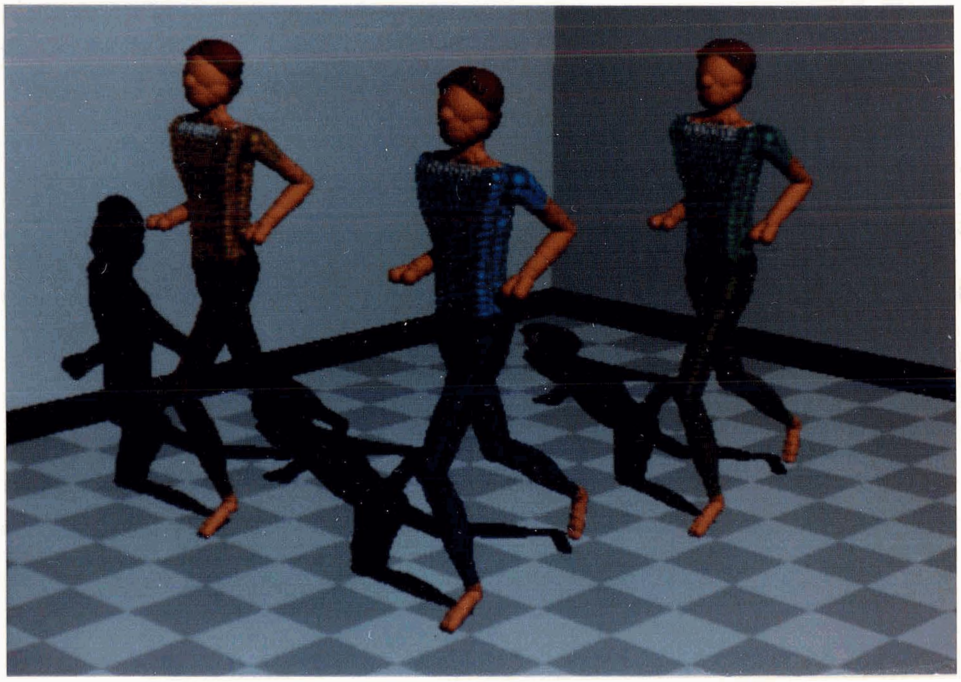

Figure 6-8: The 3 joggers

6.2.1. Adaptive versus Non-adaptive

Adaptive broadcasting was expected to keep latency times low. Figure 6-9 illustrates this point. As data sets get larger, the non-adaptive latency time increases dramatically affecting the overall time taken per ray. On the other hand, the adaptive curve increases slowly. The difference between the adaptive and direct memory curves is the overhead associated with broadcasting.

6.2.2. Type of Hierarchical Structure

The order of the hierarchical tree was found to be important to the efficiency of the ShellProcessor (Figure 6-10). Although binary tree packets are faster to process, more packets have to be retrieved. The saving in processing time is more than negated by the increase in latency time. The opposite holds true for octree packets. The best performance was the medium processing time and the medium latency time of the quad tree packets. Unlike with the single-PERT with direct memory, broadcasting introduces an important latency factor in the evaluation of the *best* tree.

6.2.3. Organization of Shell Data

We also questionned whether the order in which the shell packets were broadcast had any effect on the ShellProcessor performance. Test scenes were rendered using depth-first, breadth-first and random traversal of the hierarchical data structure. As illustrated in Figure 6-11, breadth-first broadcasting showed consistently good results. This was expected since the contents of the packets are processed in breadth-first fashion.

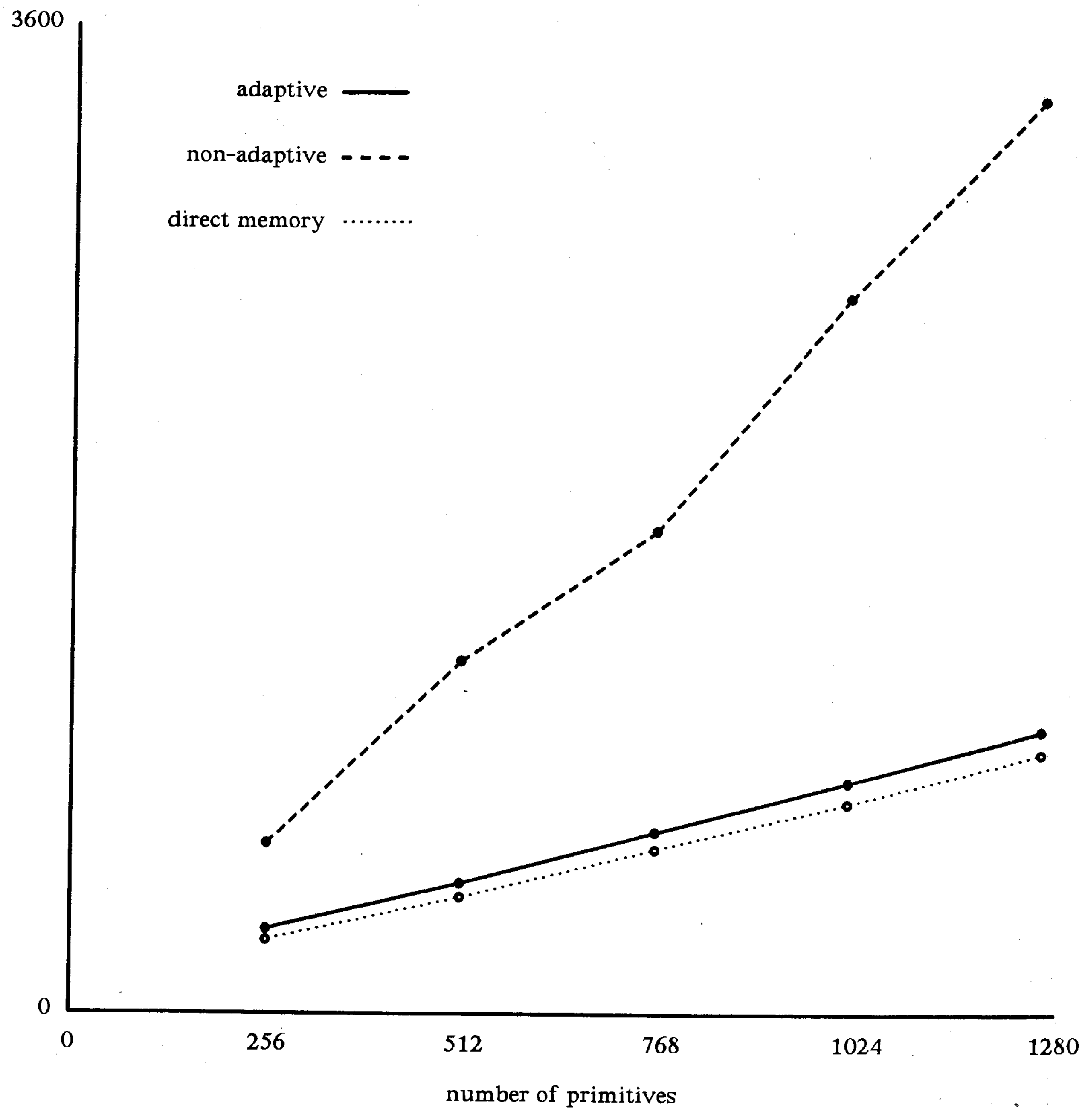

Figure 6-9: Average time per ray (microseconds)
Scenes 404, 408, 412, 416, 420
PrimProcessor

	binary	quad	oct
processing	467.8	523.9	690.5
latency	124.9	5.0	0.5
total	592.7	528.9	691.0

Figure 6-10: Comparison of tree structures
Averages times per ray (microseconds)
Scenes 212, 412, 812
ShellProcessor

	breadth	depth	random
scene 404	5.1	11.4	11.4
scene 408	4.7	11.4	11.3
scene 412	5.0	12.0	12.0
scene 416	5.1	12.1	11.9
scene 420	4.9	11.9	11.7

Figure 6-11: Comparison of broadcast order of tree
Average latency per ray (microseconds)
ShellProcessor

6.3. Multi-PERT Performance

Having shown that adaptive broadcasting is effective in a single-PERT system, we then looked at the effect of going to a multi-PERT configuration. The rendering time for a processor in the PERT can be expressed as:

$$time = (processing/ray + latency/ray + pipelinewait/ray) \times rays$$

. For n PERTs, the number of rays per PERT is approximately $\frac{totalrays}{n}$; $processing/ray$ and $pipelinewait/ray$ remain roughly constant. In the case of non-adaptive broadcasting, the $latency/ray$ also remains constant. However, this is not the case with adaptive broadcast.

As PERTs are added to the configuration, more packets will be broadcast per broadcast cycle. This results in longer cycles and thus longer latency times. We needed a method of determine how $latency/ray$ was affected as a function of n PERTs.

6.3.1. Analysis

To predict the multi-PERT performance, we chose to do an analysis based on the PrimProcessor. The reasons for choosing the PrimProcessor are as follows:

- the primitive data set is usually the largest, resulting in longer broadcast cycles and greater latency.
- the PrimProcessor puts at most 2 IDs in the BIC at any one time.
- the IDs need to be processed in a specific order and not on a first come, first served order.
- there is less chance of overlapping requests between the PrimProcessors.

With the PrimProcessor, the average latency for the first packet potentially accounts for the largest part of the total latency per ray since there is no parallelism until the first packet is retrieved. From simulations, for each scene, the predicted average latency for the first packet can be used to calculate the expected average latency per ray.

The analysis is for the worst case broadcast cycle. This means that we assume all PrimProcessors will put their requests for packets to their respective BICs simultaneously and that there are no overlapping requests. Therefore, in one broadcast cycle, the PrimBroadcaster will broadcast all the IDs and as many packets as there are IDs in the BICs. If, for example, there is an average 1.45 packets broadcast per ray and *n* PERTs, the minimum packets broadcast in cycle would be *n* and the maximum *2n*.

The following terms are used in the analysis:

p	total number of packets broadcast
r	total number of rays processed
n	number of processors
k	number of packets broadcast in a cycle
mct	minimum broadcast cycle time
apt	average broadcast time of a packet
eal	expected average latency per packet
$P(k)$	probability of k packets broadcast
$L(k)$	average latency when k packets broadcast
P_1	probability of a processor requiring 1 packet
P_2	probability of a processor requiring 2 packets
i	number of processors requesting 2 packets

The expected average latency for the first packet in a worst case broadcast cycle is given by:

$$eal = \sum_{k=n}^{2n} P(k)\ L(k)$$

Letting $k = n+i$:

$$eal = \sum_{i=0}^{n} P(n+i)\ L(n+i)$$

where

$$L(n+i) = \frac{mct + (n+i-1)\,apt}{2} + apt$$

$$P(n+i) = \binom{n}{i} P_1^{n-i} P_2^{i}$$

and where

$$P_1 = \frac{2r-p}{r} \quad \text{and} \quad P_2 = \frac{p-r}{r}$$

Using this analysis, we can now predict worst case performance of a multi-PERT adaptive broadcasting architecture.

6.3.2. Results

Graphs illustrating the predicted performance of a multi-PERT machine to render the test scenes 412, 23 and 45 are shown in Figure 6-12 to Figure 6-14. Three lines are plotted on each graph. The dashed line represents performance with non-adaptive broadcasting. This is the *worst* case for broadcasting. The dotted line represents a multi-PERT configuration where each PERT has its own memory modules for direct memory access to data. Each PERT has the complete scene stored in its memory. In a sense, this represents a *best* case but in reality, as stated previously, this is an inefficient way to resolve the data contention problem.

The solid line represents predicted results based the analysis described above for adaptive broadcasting. This is a *worst* case prediction. Realistically, results should be significantly better. The shape of the curve is interesting. Whereas the other 2 curves are linear, this curve approaches the non-adaptive curve as the broadcast cycle becomes increasingly saturated and latency increases. The sharp dip toward the non-adaptive curve represents the sharp increase in latency as the probability of saturation becomes greater than 0.0 ($P(k) > 0.0$ where k = total packets).

6.4. Discussion

These are many advantages to the multi-PERT architecture, the main one being its high degree of parallelism. Not only are there many PERTs working in parallel, the PERT itself is a pipeline and the BIC operates in parallel with the SJ16. Also the PERT's modularity makes it easy to implement in a parallel architecture.

Another significant advantage is the microcodable microprocessor that allows for a more flexible PERT which can be tailored to applications instead of being a fixed hardware solution. The potential for downloading user-defined micro-instructions means that one could add to the ShellProcessor functions for other bounding volume shapes; to the PrimProcessor, functions for other geometric and procedural objects; and to the ShadeProcessor, better shading models. This contrasts with other proposed architectural solutions that are limited to polygons for example.

Adaptive broadcasting allows concurrent access to data while avoiding the communication bottleneck of a shared memory. It has been shown to be better than non-adaptive broadcast up to the point where the broadcast cycle becomes saturated, i.e., when all packets are being broadcast in a cycle.

There are two main disadvantages with this architecture. As data sets get larger, latency increases and performance decreases. However, the PERT's capability to handle

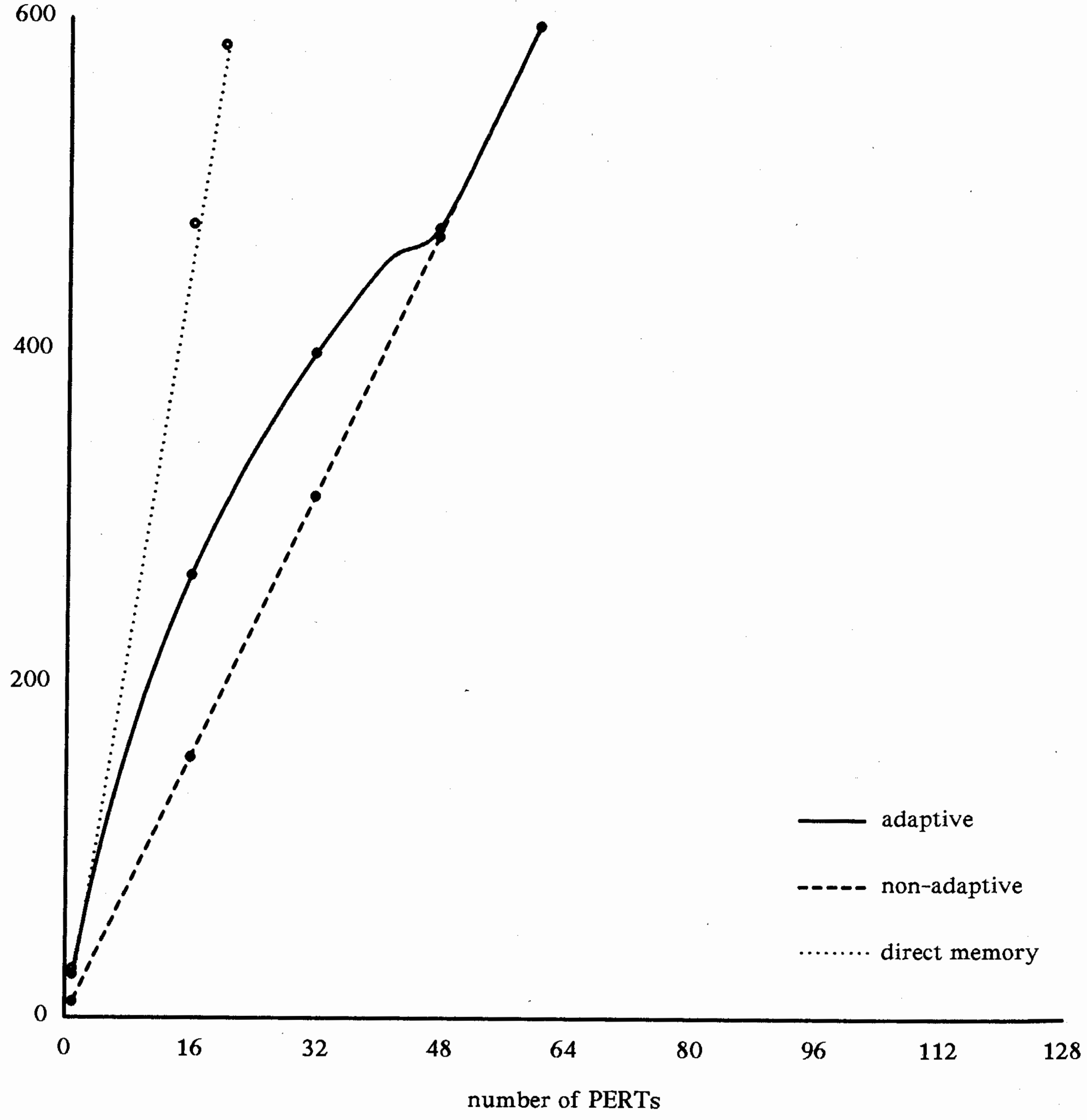

Figure 6-12: Times faster than VAX/750
based on expected average latency
Scene 412

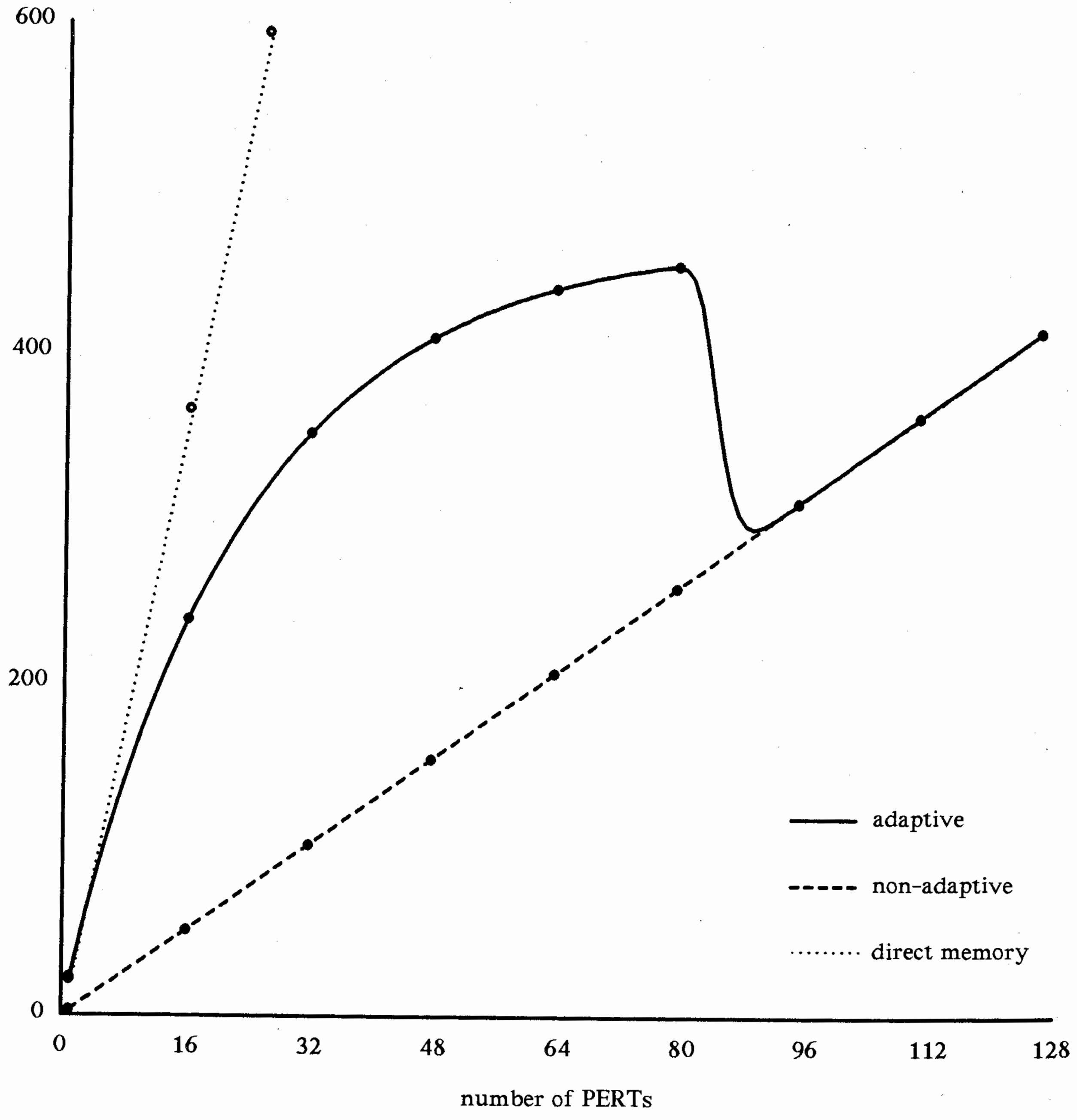

Figure 6-13: Times faster than VAX/750 based on expected average latency Scene 23

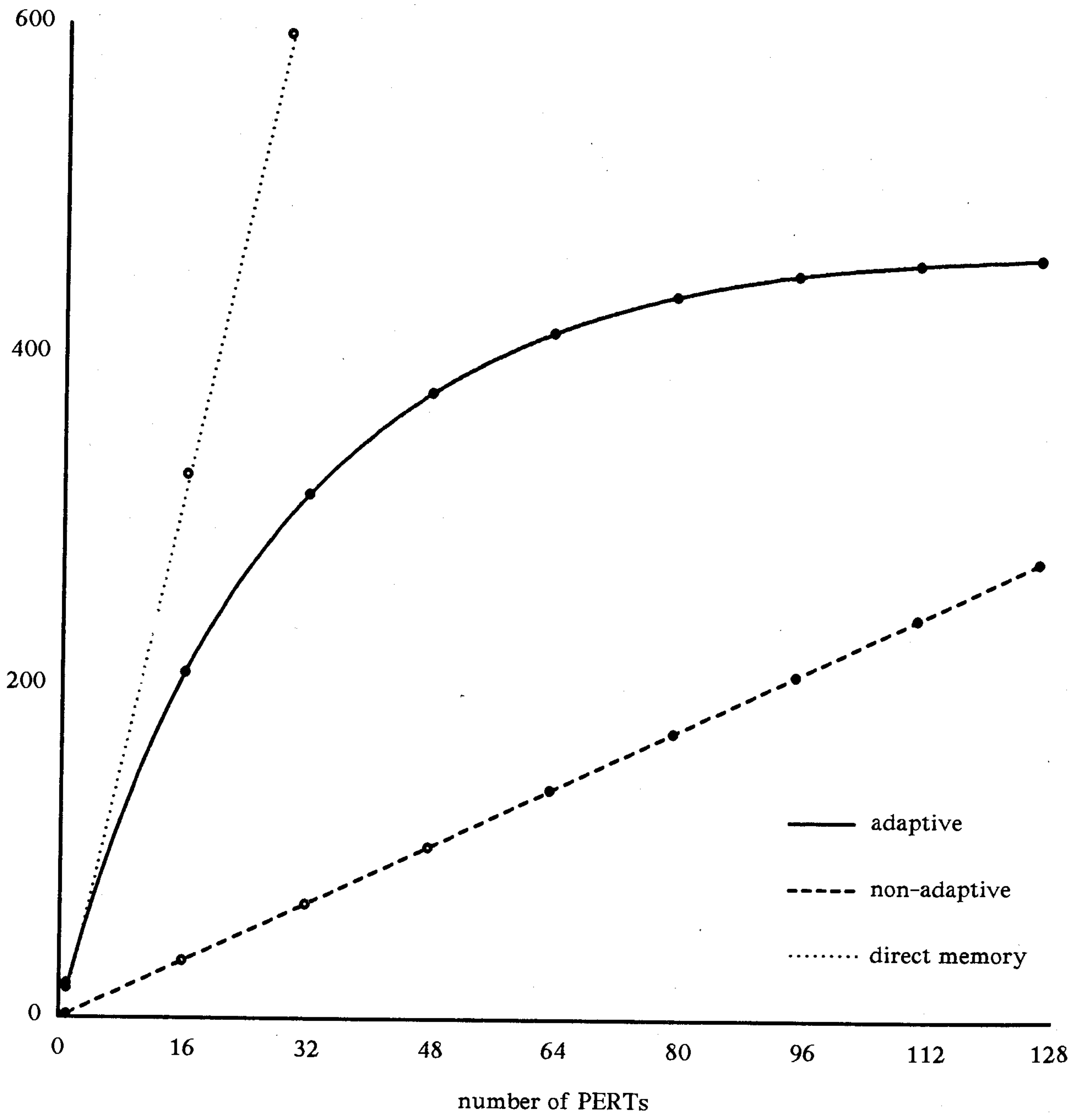

Figure 6-14: Times faster than VAX/750 based on expected average latency Scene 45

complex surfaces means that scenes which now consist of many polygons or spheres could potentially be reduced to fewer but more complex primitives. Therefore the impact of large data sets could be reduced.

The other main disadvantage is harder to evaluate. This is the *goodness* of the hierarchical data structure; the balancing of the tree depth and order versus the number and type of primitives enclosed in leaf shells. This issue requires further study. However, at the present time, the design of the hierarchical data structure is left completely in the hands of the user.

It is obvious from our results that as the data sets get larger or the number of PERTs increases, latency becomes significant. But even with latency, we have obtained 2 orders of magnitude with only 8 PERTs. Yet two other issues should be explored. With a high latency wait and the parallelism of the BIC, the SJ16 may be able to do floating point arithmetic in firmware without any serious performance degradation. This would make the PERT even more attractive in both size and cost. The other issue is whether or not broadcasting is needed for a small number of PERTs, i.e., determining at what point memory contention for a shared memory becomes equal to the latency for adaptive broadcasting. Unfortunately, latency is dependent on the data set and on the number of PERTs making generalizations of improvements difficult.

6.5. Conclusion

We have presented in these chapters the features of a multiprocessor architecture designed to improve the performance of ray tracing image generation techniques. The architecture was the result of a three-step process. The first step was the modification of the standard ray tracing algorithm. The key features of this modification are: a) the hierarchical data structure, b) the partitioning of the ray tracing algorithm into three independent tasks, and c) the partitioning of the data into three independent data sets. These have resulted in significant improvements over the standard algorithm.

The second step was the mapping of the algorithm into the PERT. The key features of the PERT are: a) the microprogrammable VLSI processors, b) the special floating point units, and c) the 3 processor pipeline. A single PERT has been shown to run 20 times faster than a VAX/750 [CHIL85].

The third step was the interconnection of a set of PERTs into a parallel architecture. The key features of the multi-PERT machine are: a) adaptive broadcasting, b) 3 broadcast busses, and c) BICs for parallel input. Predicted performance for complex scenes show improvements of 2 orders of magnitude over the VAX for an 8 PERT configuration.

Although there are many issues associated with both the algorithms and the hardware modeled which remain to be studied, our results for the proposed multi-PERT architecture are very promising.

References

[APPE68] Appel, A.
Some techniques for shading machine renderings of solids.
In *AFIPS Spring Joint Conference*, pages 37-45. AFIPS, 1968.

[BLIN77] Blinn, J.F.
Models of light reflection for computer synthesized pictures.
In *Siggraph'77 Conference Proceedings*, pages 192-198. ACM, San Jose, California, 1977.

[CALV82] Calvert, T.W., Chapman, J., and Patla, A.
Aspects of the Kinematic Simulation of Human Movement.
IEEE Computer Graphics and Applications 2(9):41-49, November, 1982.

[CHIL85] Chilka, P.
PERT: A Pipelined Engine for Ray Tracing Graphics.
Master's thesis, Simon Fraser University, May, 1985.

[CLAR76] Clark, J.H.
Hierarchical geometric models for visible surface algorithms.
Communications ACM 19(10):547-554, October, 1976.

[CLEA83] Cleary, J.G., Wyvill, B., Birtwistle, G. M., and Vatti, R.
Multiprocessor ray tracing.
Technical Report 83/128/17, University of Calgary, October, 1983.

[COOK82] Cook, R.L., and Torrance, K.E.
A reflection model for computer graphics.
ACM Transactions on Graphics 1(1):7-24, January, 1982.

[DIPP84] Dippe, M., and Swensen, J.
An adaptive subdivision algorithm and parallel architecture for realistic image synthesis.
In ACM (editor), *SIGGRAPH'84 Conference Proceedings*, pages 149-157. ACM, New York, 1984.

[FLYN66] Flynn, M.J.
Very high-speed computing systems.
In *Proceedings of the IEEE*, pages 1901-1909. 1966.

[GLAS84] Glassner, A.S.
Space subdivision for fast ray tracing.
IEEE Computer Graphics and Applications 4(10):15-22, October, 1984.

[GOLD71] Goldstein, R.A., and Nagel, R.
3-D visual simulation.
In *SIMULATION*, pages 25-31. January, 1971.

[HALL83] Hall, R.A., and Greenberg, D.P.
A testbed for realistic image synthesis.
IEEE Computer Graphics and Applications 3(10):10-20, November, 1983.

[HOBS81] Hobson, Richard.
Structured Machine Design: An Ongoing Experiment.
In *Proceedings of the 8th Symposium on Computer Architecture*, pages 37-55. SIGARCH, Minneapolis, May, 1981.

[HOBS82] Hobson, Richard.
SAMjr Microprogamming guide, Version 2.1.
1983.

[KAY79] Kay, D.S.
Transparency, refraction, and ray-tracing for computer synthesized images.
Master's thesis, Cornell University, January, 1979.

[NISH81] Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K.
LINKS-1: A parallel pipelined multimicrocomputer system for image creation.
In ACM (editor), *Proceedings of the 10th Symposium on Computer Architecture*, pages 387-394. ACM, New York, 1981

[PHON73] Phong B-T.
Illumination model for computer generated pictures.
Communications ACM 18(6), June, 1975.

[RUBI80] Rubin, S.M., and Whitted, T.
A 3-dimensional representation for fast rendering of complex scenes.
In ACM (editor), *SIGGRAPH'80 Conference Proceedings*, pages 110-116. ACM, New York, 1980.

[ULLN83] Ullner, M.K.
Parallel machines for computer graphics.
PhD thesis, California Institute of Technology, 1983.

[WEGH84] Weghorst, H., Hooper, G., and Greenberg, D.P.
Improved computational methods for ray tracing.
ACM Transactions on Graphics 3(1):52-69, January, 1984.

[WEIT85a] Weitek Solids Modeling Engine.
Weitek Corporation Product Literature. 1985.

[WEIT85] WTL1164/1165 Low-Latency 64-bit IEEE Floating Point Multiplier/ALU. Weitek Corporation Product Literature. 1985.

[WHIT80] Whitted, T.
An improved illumination model for shaded display.
Communications ACM 23(6):343-349, June, 1980.