NASA Contractor Report 195080

ICASE Report No. 95-31

ICASE

PARALLEL RENDERING

Thomas W. Crockett

Contract No. NAS1-19480
April 1995

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

Parallel Rendering

Thomas W. Crockett

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center

Abstract

In computer graphics, rendering is the process by which an abstract description of a scene is converted to an image. When the scene is complex, or when high-quality images or high frame rates are required, the rendering process becomes computationally demanding. To provide the necessary levels of performance, parallel computing techniques must be brought to bear. Although parallelism has been exploited in computer graphics since the early days of the field, its initial use was primarily in specialized applications. The VLSI revolution of the late 1970's and the advent of scalable parallel computers during the late 1980's changed this situation. Today, parallel hardware is routinely used in graphics workstations, and numerous software-based rendering systems have been developed for general-purpose parallel architectures.

This article provides a broad introduction to the subject of parallel rendering, encompassing both hardware and software systems. The focus is on the underlying concepts and the issues which arise in the design of parallel rendering algorithms and systems. We examine the different types of parallelism and how they can be applied in rendering applications. Concepts from parallel computing, such as data decomposition, task granularity, scalability, and load balancing, are considered in relation to the rendering problem. We also explore concepts from computer graphics, such as coherence and projection, which have a significant impact on the structure of parallel rendering algorithms. Our survey covers a number of practical considerations as well, including the choice of architectural platform, communication and memory requirements, and the problem of image assembly and display. We illustrate the discussion with numerous examples from the parallel rendering literature, representing most of the principal rendering methods currently used in computer graphics.

This work was supported in part by the National Aeronautics and Space Administration under Contract No. NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), M/S 132C, NASA Langley Research Center, Hampton, VA 23681-0001.

E-mail: tom@icase.edu

World Wide Web: http://www.icase.edu/~tom/

Table of Contents

List of Figures

I. Introduction

In computer graphics, *rendering* is the process by which an abstract description of a scene is converted to an image. Figure 1 illustrates the basic problem. For purposes of this discussion, a scene is a collection of geometrically-defined objects in three-dimensional *object space,* with associated lighting and viewing parameters. The rendering operation illuminates the objects and projects them into two-dimensional *image space*, where color intensities of individual pixels are computed to yield a final image.

For complex scenes or high-quality images, the rendering process is computationally intensive, requiring millions or billions of floating-point and integer operations for each image. The need for interactive or real-time response in many applications places additional demands on processing power. The only practical way to obtain the needed computational power is to exploit multiple processing units to speed up the rendering task, a concept which has become known as *parallel rendering*.

A. Historical perspective

The incorporation of parallelism into rendering systems has been an evolutionary process, with its origins in the early days of computer graphics. The pioneering Graphic 1 display system developed at Bell Telephone Laboratories in the early 1960's used its own internal processor to drive the display and handle user interactions, allowing it to operate independently of its mainframe host (1). In 1968, Myer and

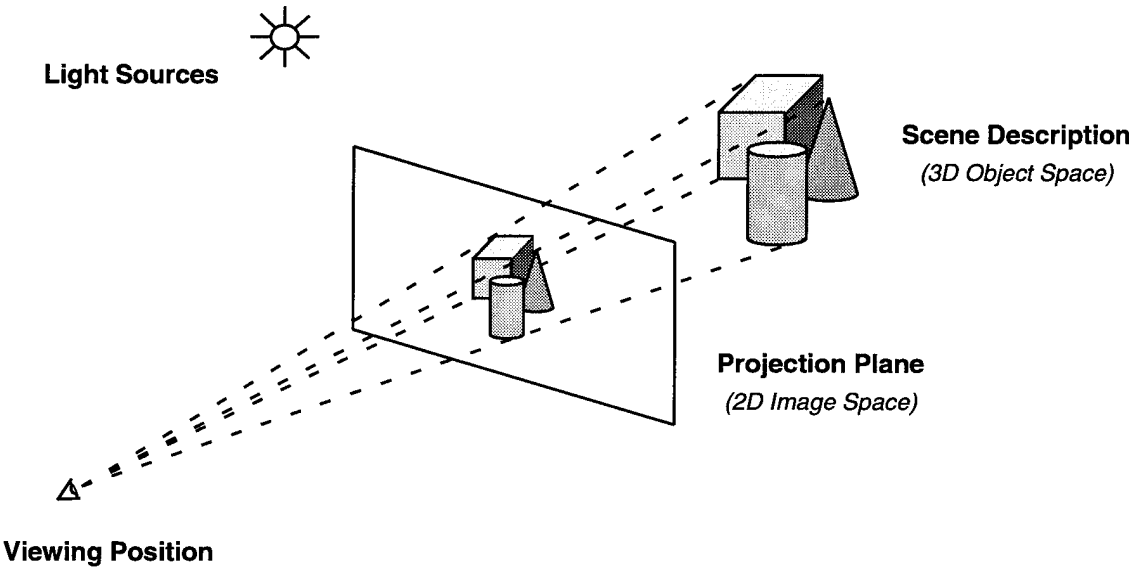

Figure 1. The generic rendering problem. A three-dimensional scene is projected onto an image plane, taking into account the viewing parameters and light sources.

Sutherland (2) examined the allocation of graphics functionality in a multiprocessor configuration composed of a host computer, display processor, and display channel[1]. They discussed the advantages and disadvantages of using shared-memory to communicate between the central processor and the display subsystem, and noted a trend toward increasingly complex display architectures. During this same time period, more sophisticated graphics hardware began to appear, incorporating multiple function units and low-level parallelism in the form of simultaneous logic operations. Sproull and Sutherland's classic "clipping divider" provides a modest example (3). The demands and budgets of real-time flight simulation prompted more ambitious designs, including one by Schumacker *et al.* for the U.S. Air Force (4). That architecture included multiple processors and a variety of specialized function units organized into three distinct rendering subsystems, one for terrain, one for objects, and a third for point source lights.

During the 1970's, real-time flight simulation continued as a primary driver of high-performance graphics systems. By the end of the decade, these systems routinely incorporated modest levels of parallelism (5), but they were highly specialized and very expensive, making them ill-suited for more general rendering tasks.

The VLSI revolution of the late 1970's and early 1980's marked an important turning point in the development of computer graphics architectures. The availability of compact, low-cost processors and high-capacity memory chips made high-performance systems practical for general-purpose use. The relative simplicity of constructing systems by replicating off-the-shelf components encouraged additional experimentation with parallel architectures. Early designs based on this new hardware paradigm included z-buffered scan conversion systems by Fuchs and Johnson (6, 7) and Parke (8), and a pipelined polygon rendering architecture by Clark (9, 10).

During the 1980's, the use of multiple special-purpose hardware units became the standard approach for achieving high rendering rates in graphics accelerators, graphics workstations, and specialized graphics computers. The advent of "massively" parallel computer systems, containing from tens to thousands of generic processing elements, added a new dimension to parallel rendering, promising added flexibility, but raising numerous algorithmic and efficiency issues for software-based parallel renderers.

B. Organization

In the remainder of our discussion, we assume a passing familiarity with the basic principles and

---

[1] The concept of *channels* was common in mainframe systems from the 1960's and 1970's. Channels are essentially specialized co-processors used to offload I/O tasks from the central processing unit.

terminology of computer graphics. Parallel processing concepts are presented at a somewhat more introductory level. We begin our examination of parallel rendering in Section II with a brief overview of the applications to which it has most commonly been applied. Specific application areas will be addressed in more detail in the context of subsequent sections. Section III explores different types of parallelism and how they relate to the rendering problem. Section IV introduces a number of concepts which are central to an understanding of parallel rendering algorithms. Building on this base, Section V considers design and implementation issues for parallel renderers, with an emphasis on architectural considerations and application requirements. Throughout Sections III, IV, and V, we illustrate our discussion with examples from the parallel rendering literature, encompassing both hardware and software systems. Section VI completes our survey of parallel rendering systems with an examination of several parallel hardware architectures as well as radiosity and terrain rendering methods.

II. Applications of Parallel Rendering

Parallel techniques are appropriate whenever rendering performance is an issue. Demanding applications such as real-time simulation, animation, virtual reality, photo-realistic imaging, and scientific visualization all benefit from the use of parallelism to increase rendering performance. Indeed, these applications have been primary motivators in the development of parallel rendering methods. Parallel rendering has been applied to virtually every image generation technique used in computer graphics, including surface and polygon rendering, terrain rendering, volume rendering, ray-tracing, and radiosity. Although the requirements and approaches vary for each of these cases, there are a number of concepts which are important in understanding how parallelism applies to the generic rendering problem. We consider these in Sections III and IV.

III. Parallelism in the Rendering Process

Several different types of parallelism can be applied in the rendering process. These include *functional parallelism*, *data parallelism*, and *temporal parallelism*. Some are more appropriate to specific applications or specific rendering methods, while others have broader applicability. The basic types can also be combined into hybrid systems which exploit multiple forms of parallelism. Each of these options is discussed below.

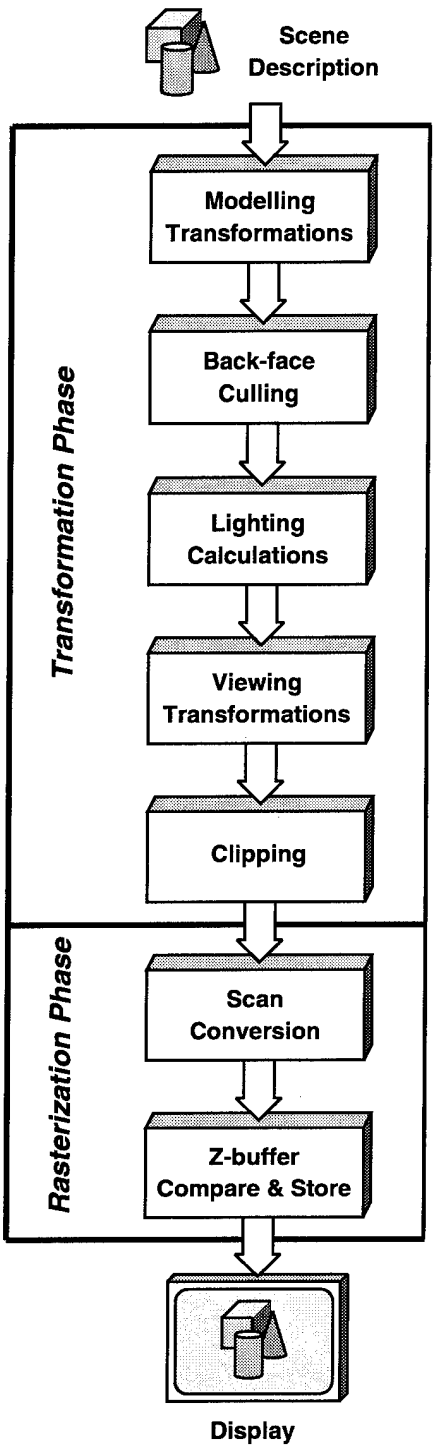

Figure 2. A typical polygon rendering pipeline. The number of function units and their order varies depending on details of the implementation.

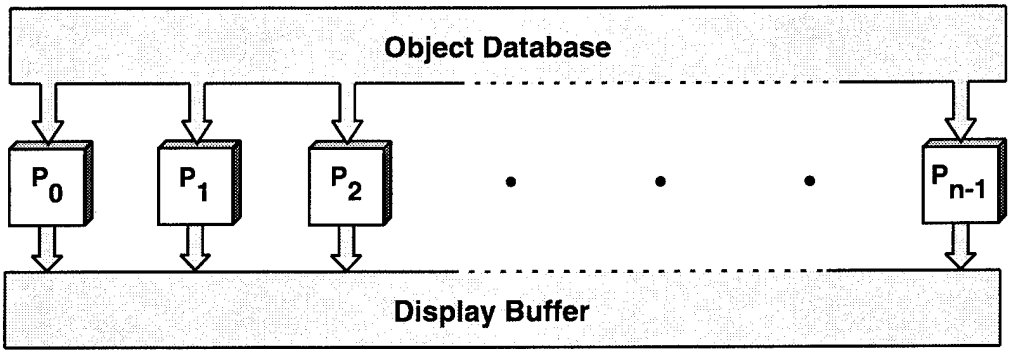

Figure 3. A data-parallel rendering system. Multiple data items are processed simultaneously and the results are merged to create the final image.

A. Functional parallelism

One way to obtain parallelism is to split the rendering process into several distinct functions which can be applied in series to individual data items. If a processing unit is assigned to each function (or group of functions) and a data path is provided from one unit to the next, a rendering *pipeline* is formed (Figure 2). As a processing unit completes work on one data item, it forwards it to the next unit, and receives a new item from its upstream neighbor. Once the pipeline is filled, the degree of parallelism achieved is proportional to the number of functional units.

The functional approach works especially well for polygon and surface rendering applications, where 3D geometric primitives are fed into the beginning of the pipe, and final pixel values are produced at the end. This approach has been mapped very successfully into the special purpose rendering hardware used in a variety of commercial computer graphics workstations produced during the 1980's and 1990's. The archetypal example is Clark's Geometry System (9, 10), which replicated a custom VLSI geometry processor in a 12-stage pipeline to perform transformation and clipping operations in two and three dimensions.

Despite its success, the functional approach has two significant limitations. First, the overall speed of the pipeline is limited by its slowest stage, so functional units must be designed carefully to avoid bottlenecks. More importantly, the available parallelism is limited to the number of stages in the pipeline. To achieve higher levels of performance, an alternate strategy is needed.

B. Data parallelism

Instead of performing a sequence of rendering functions on a single data stream, it may be preferable to

split the data into multiple streams and operate on several items simultaneously by replicating a number of identical rendering units (Figure 3). The parallelism achievable with this approach is not limited by the number of stages in the rendering pipeline, but rather by economic and technical constraints on the number of processing units which can be incorporated into a single system. Of particular importance is the communication network which routes data among the processing units. As we will see in subsequent sections, the characteristics of the communication network have a significant influence on the choice of rendering algorithms, and *vice versa*.

Because the data-parallel approach can take advantage of larger numbers of processors, it has been adopted in one form or another by most of the software renderers which have been developed for general-purpose "massively parallel" systems. Data parallelism also lends itself to scalable implementations, allowing the number of processing elements to be varied depending on factors such as scene complexity, image resolution, or desired performance levels.

Two principal classes of data parallelism can be identified in the rendering process. *Object parallelism* refers to operations which are performed independently on the geometric primitives which comprise objects in a scene. These operations constitute the first few stages of the rendering pipeline (Figure 2), including modeling and viewing transformations, lighting computations, and clipping. *Image parallelism* occurs in the later stages of the rendering pipeline, and includes the operations used to compute individual pixel values. Pixel computations vary depending on the rendering method in use, but may include illumination, interpolation, composition, and visibility determination. Collectively we call the object-level stages of the pipeline the *transformation phase;* the image-level stages are grouped together to form the *rasterization phase*.

Potential levels of data parallelism can be quite high. The number of geometric primitives in a scene typically ranges from a few hundred to a few million. The number of pixel values to be computed may range from thousands to hundreds of millions, depending on image resolution, sampling frequency, and depth complexity of the scene. In practice, geometric primitives and pixels are usually processed in groups to take advantage of more efficient algorithms and to reduce communication requirements, but the available parallelism normally exceeds the number of processing elements by a large factor.

To avoid bottlenecks, most data-parallel rendering systems must exploit both object and image parallelism. Obtaining the proper balance between these two phases of the computation is difficult, since the workloads involved at each level are highly dependent on factors such as the scene complexity, average screen area of transformed geometric primitives, sampling ratio, and image resolution. One approach is to define performance targets for each phase and construct the system to meet those goals. This approach is generally preferred when separate hardware will be dedicated to object and image

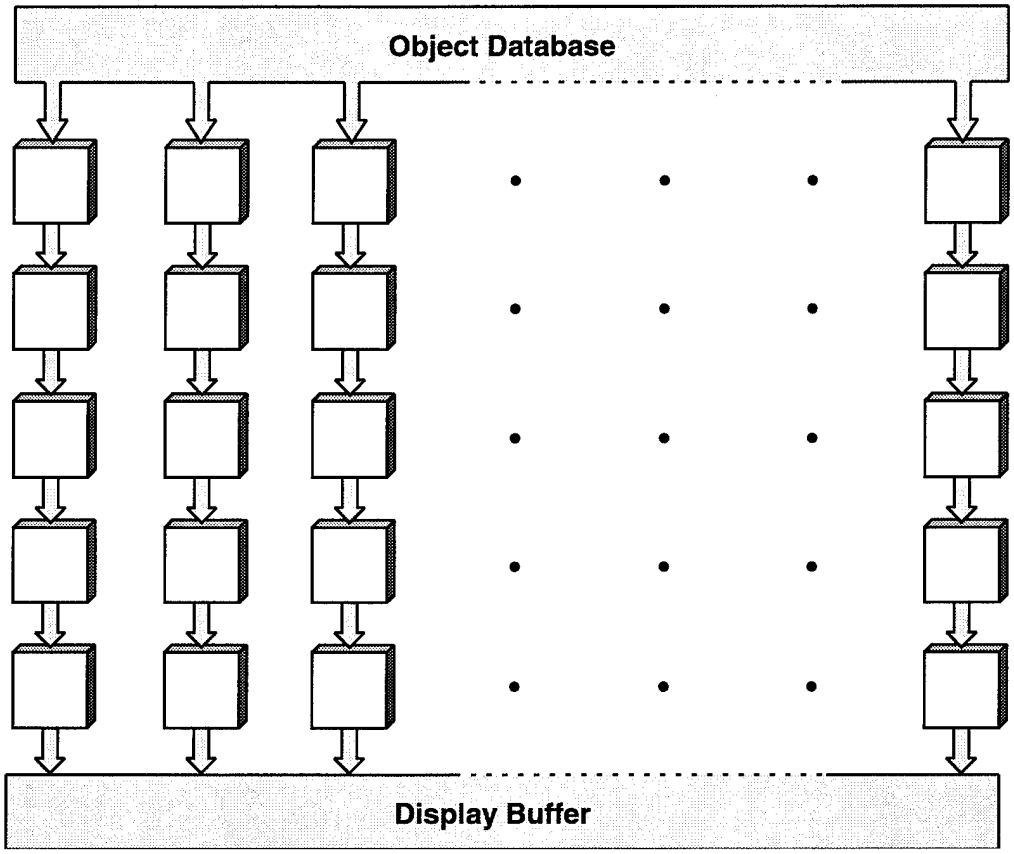

Figure 4. A hybrid rendering architecture. Functional parallelism and data parallelism are both exploited to achieve higher performance.

computations. In systems where the object and image computations are performed using the same processing units, performance targets must be based on the combined workloads. In either case, load balancing is important in assuring efficient utilization of the hardware.

C. Temporal parallelism

In animation applications, where hundreds or thousands of high-quality images must be produced for subsequent playback, the time to render individual frames may not be as important as the overall time required to render all of them. In this case, parallelism may be obtained by decomposing the problem in the time domain. The fundamental unit of work is a complete image, and each processor is assigned a number of frames to render, along with the data needed to produce those frames.

D. Hybrid approaches

It is certainly possible to incorporate multiple forms of parallelism in a single system. For example, the functional- and data-parallel approaches may be combined by replicating all or part of the rendering pipeline (Figure 4). An early example of this approach is the LINKS-1 system (11), which contained 64 identical microcomputers which could be dynamically reconfigured into multiple pipelines of varying depth. A more recent example is Silicon Graphics' RealityEngine (12), which uses multiple transformation and rasterization units in a highly pipelined architecture to achieve rendering rates on the order of one million polygons per second. In similar fashion, temporal parallelism may be combined with the other strategies to produce systems with the potential for extremely high aggregate performance.

IV. Algorithmic Concepts

The design of effective parallel rendering algorithms can be a challenging task. In some cases, existing sequential algorithms have straightforward parallel decompositions. In other cases, new algorithms must be developed from scratch. Whatever their origin, most parallel algorithms introduce overheads which are not present in their sequential counterparts. These overheads may result from some or all of the following:

- communication among tasks or processors
- delays due to uneven workloads
- additional or redundant computations
- increased storage requirements for replicated or auxiliary data structures

To understand how these overheads arise in parallel rendering algorithms, we need to examine several key concepts. Some of these concepts (task and data decomposition, granularity, scalability, and load balancing) are common to most parallel algorithms, while others (coherence and object-space to image-space data mapping) are specific to the rendering problem. Each of these topics is considered in detail in the remainder of this section.

A. Embarrassingly parallel algorithms

Some problems can be parallelized trivially, requiring little or no interprocessor communication, and with no significant computational overheads attributable to the parallel algorithm. Such applications are said to be *embarrassingly parallel*, and efficient operation can be expected on a variety of platforms, ranging from networks of personal computers or graphics workstations up to massively parallel supercomputers. Rendering algorithms which exploit temporal parallelism typically fall into this category.

Rendering methods based on ray-casting (such as ray-tracing and direct volume rendering) also have embarrassingly parallel implementations in certain circumstances. Because pixel values are computed by

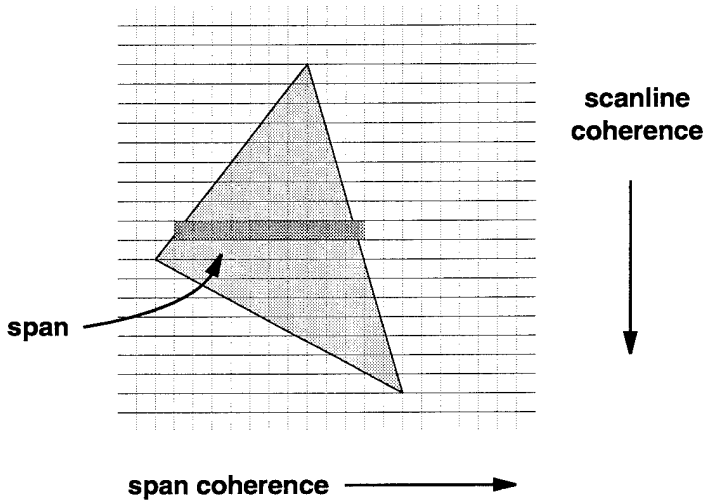

Figure 5. Spatial coherence in image space. Pixel values tend to be similar from one scanline to the next, and from pixel to pixel within spans. Sequential rendering algorithms exploit this property to reduce computation costs during scan conversion.

shooting rays from each pixel into the scene, image-parallel task decompositions are very natural for these problems. If every processor has fast access to the entire object database, then each ray can be processed independently with no interprocessor communication required. This approach is practical for shared-memory architectures, and also performs well on distributed-memory systems when sufficient memory is available to replicate the object database on every processor.

B. Coherence

In computer graphics, *coherence* refers to the tendency for features which are nearby in space or time to have similar properties (13). Many fundamental algorithms in the field rely on coherence in one form or another to reduce computational requirements. Coherence is important to parallel rendering in two ways. First, parallel algorithms which fail to preserve coherence will incur computational overheads which may not be present in equivalent sequential algorithms. Secondly, parallel algorithms may be able to exploit coherence to reduce communication costs or improve load balance.

Several types of coherence are important in parallel rendering. *Frame coherence* is the tendency of objects, and hence resulting pixel values, to move or change shape or color slowly from one image to the next in a related sequence of frames. This property can be used to advantage in load balancing and image display, as we will discuss in subsequent sections.

Scanline coherence refers to the similarity of pixel values from one scanline to the next in the vertical

direction. The corresponding property in the horizontal direction is called s*pan coherence*, which refers to the similarity of nearby pixel values within a scanline (Figure 5). Sequential rasterization algorithms rely on these two forms of *spatial coherence* for efficient interpolation of pixel values between the vertices of geometric primitives. When an image is partitioned to exploit image parallelism, coherence may be lost at partition boundaries, resulting in computational overheads. The probability that a primitive will intersect a boundary depends on the size, shape, and number of image partitions (14, 15), and hence is an important consideration in the design of parallel polygon renderers (16).

A related notion in ray-casting renderers[2] is *data* or *ray coherence*. This is the tendency for rays cast through nearby pixels to intersect the same objects in a scene. Ray coherence has been exploited in conjunction with data-caching schemes to reduce communication loads in parallel volume rendering and ray-tracing algorithms (17, 18).

C. Task and data decomposition

Data-parallel rendering algorithms may be further distinguished based on the way in which the problem is decomposed into individual workloads or tasks. Since work is essentially defined as "operations on data", the choice of task decomposition has a direct impact on data access patterns. On distributed-memory architectures, where remote memory references are usually much more expensive than local memory references, the issues of task decomposition and data distribution are inseparable. Shared-memory systems offer more flexibility, since all processors have equal access to the data. While data locality is still important in achieving good caching performance, the penalties for global memory references tend to be less severe, and static assignment of data to processors is not generally required.

There are two main strategies for task decomposition. In an *object-parallel* approach, tasks are formed by partitioning either the geometric description of the scene or the associated object space. Rendering operations are then applied in parallel to subsets of the geometric data, producing pixel values which must then be integrated into a final image. In contrast, *image-parallel* algorithms reverse this mapping. Tasks are formed by partitioning the image space, and each task renders the geometric primitives which contribute to the pixels which it has been assigned.

The choice of image-parallel versus object-parallel algorithms is not clear-cut. Object-parallel algorithms tend to distribute object computations evenly among processors, but since geometric primitives usually vary in size, rasterization loads may be uneven. Furthermore, primitives assigned to different processors

---

[2] We use the term *ray-casting* to include all rendering methods which project rays from the view point through screen pixels into the scene. This encompasses both traditional ray-tracing algorithms as well as a large class of volume rendering methods.

may map to the same location in the image, requiring the individual contributions to be integrated to produce the final image. With large numbers of processors this integration step can place heavy bandwidth demands on memory busses or communication networks.

Image-parallel algorithms avoid the integration step, but have another problem: portions of a single geometric primitive may map to several different regions in the image space. This requires that primitives, or portions of them, be communicated to multiple processors, and the corresponding loss of spatial coherence results in additional or redundant computations which are not present in equivalent sequential algorithms.

To achieve a better balance among the various overheads, some algorithms adopt a hybrid approach, incorporating features of both object- and image-parallel methods (16, 19, 20, 21). These techniques partition both the object and image spaces, breaking the rendering pipeline in the middle and communicating intermediate results from object rendering tasks to image rendering tasks.

D. Granularity

Related to the concept of task and data decomposition is the notion of *granularity*. Granularity refers to the amount of computation in a basic unit of work. This workload unit may be defined to be an entire task, or it may be some smaller quantum, such as the number of instructions executed between communication events. A computation is *fine-grained* if workload units are small, or *coarse-grained* if they involve substantial processing. Granularity may also refer to data decompositions. A fine-grained decomposition includes one or a few data items in each partition, whereas a coarse-grained decomposition uses larger blocks of data. In a rendering context, a fine-grained task might compute the value of a single pixel, while a coarse-grained task might compute an entire frame in an animation sequence.

Granularity often has a direct bearing on the efficiency of a parallel computation. Fine-grained computations generally incur more overhead for task scheduling and communication, but offer the possibility of more precise load balancing. Coarse-grained computations tend to minimize communication and scheduling overheads, but they are more susceptible to load imbalances and impose tighter limits on the amount of available parallelism.

Granularity considerations are inseparably linked to performance parameters of the target architecture. For example, fine-grained algorithms are not well-suited to systems which have high overheads for task scheduling and communication, such as workstation networks. On the other hand, a coarse-grained algorithm could not be expected to map well onto a SIMD architecture composed of thousands of simple processing elements. A further discussion of SIMD versus MIMD architectures can be found in Section V.B.3.

E. Scalability

Scalability of a parallel system refers to the ability to provide additional capacity by increasing the number of processing elements. Two distinct types of scalability are important in parallel rendering. *Performance scalability* is the ability to achieve higher levels of performance on a fixed-size problem. *Data scalability* is the ability to accommodate larger problem sizes, *e.g.*, more complex scenes or higher image resolutions.

Scalability considerations apply to both hardware architectures and software rendering algorithms. Either may have bottlenecks which limit the performance levels which can be achieved or the problem sizes which can be addressed. An important consideration in designing a parallel renderer is to ensure that the architecture and algorithms will scale to the levels desired.

While traditional shared-memory systems offer the potential for low-overhead parallel rendering, their performance scalability is limited by contention on the busses or switch networks which connect processors to memory. Adding processors does not increase the memory bandwidth, so at some point the paths to memory become saturated and performance stalls. For this reason, most parallel architectures which are intended to scale to hundreds or thousands of processing elements employ a distributed-memory model, in which each processor is tightly coupled to a local memory. The combined processor/memory elements are then interconnected by a relatively scalable network or switch. The advantage is that processing power and aggregate local memory bandwidth scale linearly with the number of hardware units in the system. The disadvantage is that references to non-local data may be several orders of magnitude slower than references to local data.

A number of recent systems combine elements of both architectures, using physically distributed memories which are mapped into a global shared address space (22, 23, 24). The shared address space permits the use of concise shared-memory programming paradigms, and is amenable to hardware support for remote memory references. The result is that communication overheads can be significantly lower than those found in traditional message-passing systems, allowing algorithms with fine-grained communication requirements to scale to larger numbers of processors.

F. Load balancing

In any parallel computing system, effective processor utilization depends on distributing the workload evenly across the system. In parallel rendering, there are many factors which make this goal difficult to achieve. Consider a data-parallel polygon renderer which attempts to balance workloads by distributing geometric primitives evenly among all of the processors. First, polygons may have varying numbers of vertices, resulting in differing operation counts for illumination and transformation operations. If back-

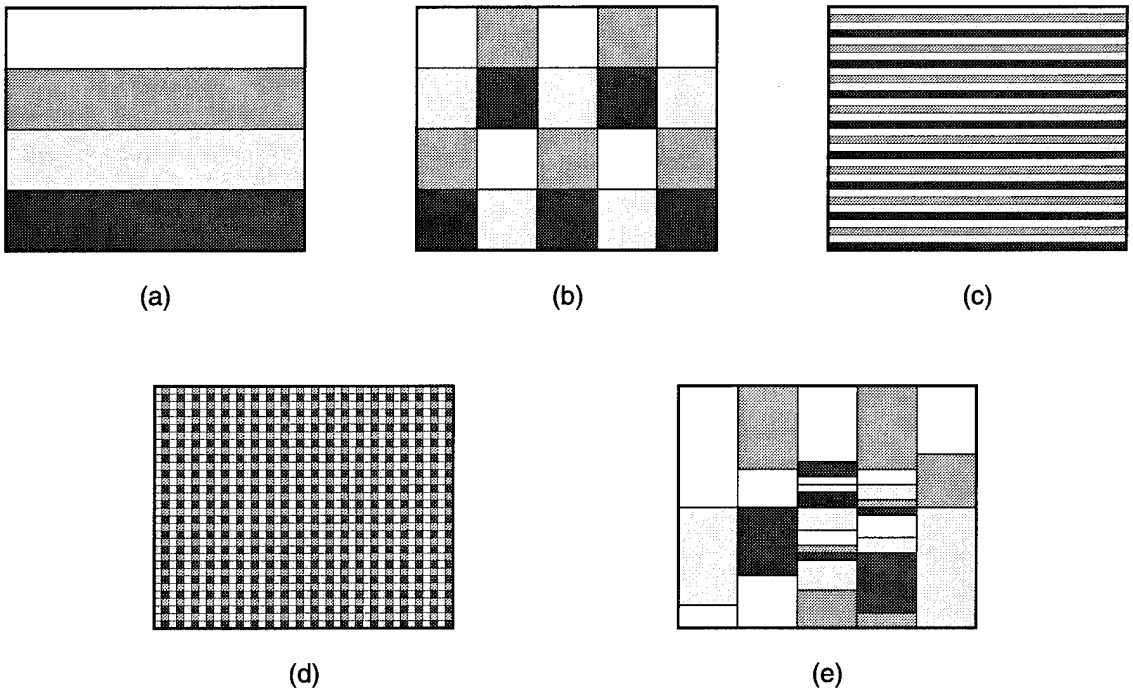

Figure 6. Image partitioning strategies. Shading indicates the assignment of image regions to four processors. *(a)* Blocks of contiguous scanlines; *(b)* square regions; *(c)* interleaved scanlines; *(d)* pixel interleaving in two dimensions; *(e)* adaptive partitioning (loosely based on Ref. 26).

face culling is enabled, different processors may discard different numbers of polygons, and the subsequent clipping step may introduce further variations. The sizes of the transformed screen primitives will also vary, resulting in differing operation counts in the rasterization routines. Depending on the method being used, hidden surface elimination will also produce variations in the number of polygons to be rasterized or the number of pixels to be stored in the frame buffer.

While this list may seem intimidating, we observe that if the number of input primitives is large (as it usually is) and the primitives are randomly assigned to processors, the workload variations described above will tend to even out. Unfortunately, a much more serious source of load imbalance arises due to another factor: in real scenes, the distribution of primitives in image space is not uniform, but tends to cluster in areas of detail. Thus processors responsible for rasterizing dense regions of the image will have significantly more work to do than other processors which may end up with nothing more than background pixels. To make matters worse, the mapping from object space to image space is view dependent, which means the distribution of primitives in the image is subject to change from one frame to the next, especially in interactive applications.

1. *Static schemes*

Strategies for dealing with this image-space load imbalance may be classified as either *static* or *dynamic*. Static load balancing techniques rely on a fixed data partitioning to distribute local variations across large numbers of processors. Figure 6 shows several different image partitioning strategies with different load balancing characteristics. Large blocks of contiguous pixels (Figure 6a) usually result in poor load balancing, while fine-grained partitioning schemes (Figure 6c,d) distribute the load better. However, fine-grained schemes are subject to computational overheads due to loss of spatial coherence, as discussed above. Analytical and experimental results (15, 25) indicate that square regions (Figure 6b) minimize the loss of coherence since they have the smallest perimeter-to-area ratio of any rectangular subdivision scheme.

2. *Dynamic schemes*

Dynamic load-balancing schemes try to improve on static techniques by providing more flexibility in assigning workloads to processors. There are two principal strategies. The *demand-driven* approach decomposes the problem into a large number of independent tasks, which are then assigned to processors one-at-a-time or in small groups. When a processor completes one task, it receives another, and the process continues until all of the tasks are complete. If tasks exhibit large variations in run time, the most expensive ones must be started early so that they will have time to complete while other processors are still busy with shorter tasks. Failure to observe this rule results in poor load balancing as processors become idle waiting for long tasks to complete. Run time estimates for tasks are usually computed heuristically in a pre-processing step, which introduces a computational overhead. The alternative is to use large numbers of fine-grained tasks in order to minimize potential variations, but this approach suffers increased overheads due to loss of coherence and more frequent task assignment operations.

The alternate *adaptive* strategy tries to minimize pre-processing overheads by deferring task partitioning decisions until one or more processors becomes idle, at which time the remaining workloads of busy processors are split and reassigned to idle processors. The result is that data partitioning is not predetermined, but instead adapts to the computational load (Figure 6e). A good example is Whitman's image-parallel polygon renderer for the BBN TC2000 (26). Whitman's renderer initially partitions the image space into a relatively small number of coarse-grained tasks, which are then assigned to processors using the demand driven model. When a processor becomes idle and no more tasks are available from the initial pool, it searches for the processor with the largest remaining workload and "steals" half of its work. The principal overheads in the adaptive approach arise in maintaining and retrieving non-local status information, partitioning tasks, and migrating data.

While dynamic schemes offer the potential for more precise load balancing than static schemes, they are

successful only when the improvements in processor utilization exceed the overhead costs. For this reason, dynamic schemes are easiest to implement on architectures which provide low-latency access to shared memory. In message-passing systems, the high cost of remote memory references makes dynamic task assignment, data migration, and maintenance of global status information more expensive, especially for fine-grained tasks. Ellsworth (16) attempted to overcome this limitation by employing an *inter-frame* load balancing scheme on Intel's Touchstone Delta system. Rather than trying to balance the load within a single frame of an image sequence (the *intra-frame* approach), his renderer uses the workload distribution from one frame to reassign image regions for the next frame. This strategy assumes that the distribution of geometric primitives will be similar in consecutive images, an example of frame coherence. The advantage of this approach is that load balancing is performed at a higher level of granularity, with less overhead. Nonetheless, Ellsworth's experiments indicated that this technique was only partially successful, encountering scalability problems in obtaining global workload information for large numbers of processors.

3. *Load balancing for ray-casting renderers*

Although the above discussion is set in the context of polygon rendering algorithms, similar considerations apply for other rendering techniques. In ray-cast volume rendering, for example, the viewing angle, distribution of features within the volume, and optimizations such as early ray termination all contribute to workload imbalances. Nieh and Levoy use a demand-driven scheme in an image-parallel volume renderer for the Stanford DASH Multiprocessor (27). Their strategy uses an initial coarse-grained static partitioning of the image, with dynamic reassignment of sub-tasks based on a finer-grained second-level partitioning.

In ray-tracing, the majority of the execution time is used to compute the intersections of rays with objects in the scene. Load imbalances arise because the cost of calculating ray/object intersections and evaluating secondary rays varies depending on the type and distribution of objects within the scene. Caspary and Scherson (28) and Salmon and Goldsmith (20) independently developed an innovative load balancing scheme for ray-tracing on distributed-memory MIMD architectures. The method begins by organizing the object data as a hierarchical tree of bounding volumes, a well-known technique employed by sequential ray-tracers to reduce the search space for intersection testing. The basic idea in the parallel implementation is to cut the tree at particular locations to produce a two-level object-space partitioning (Figure 7). The upper portion of the tree (and its associated object data, which tends to be small) is replicated on every processor, while the subtrees below the cuts (which comprise the bulk of the data) are distributed among the processors. Two distinct types of tasks are spawned on each processor, one which performs intersection calculations in the upper tree, and another which performs the same calculations for local subtrees. Because the upper-level tree is available everywhere, any processor in the system can

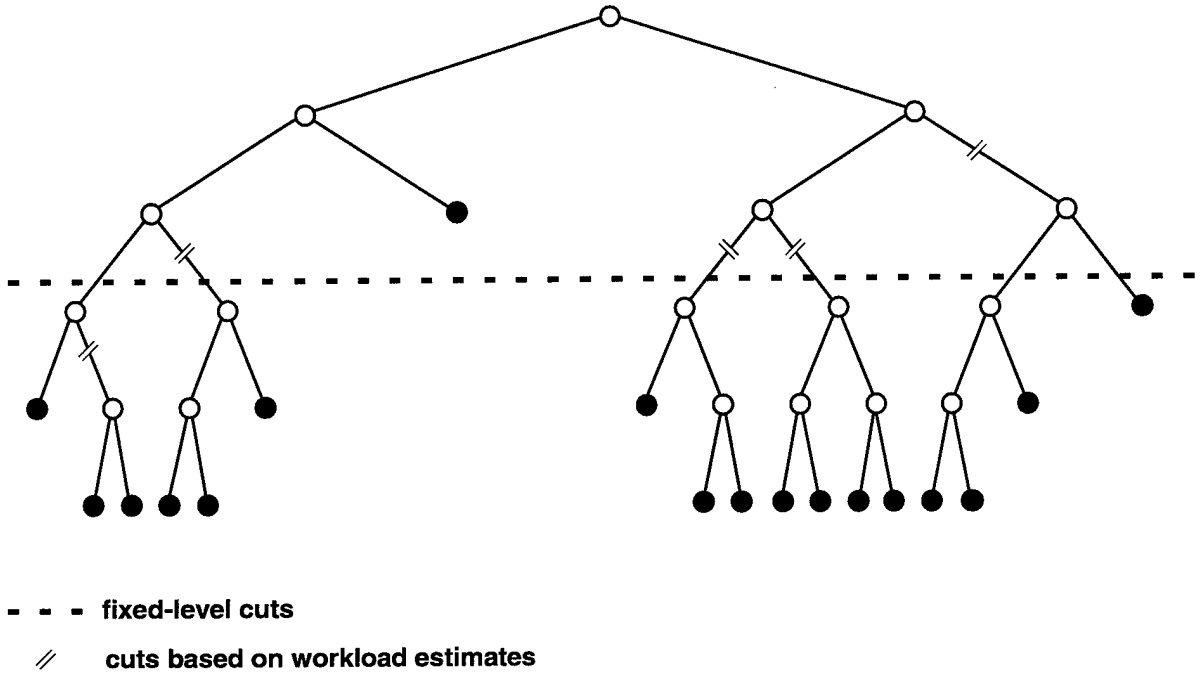

Figure 7. Hierarchical tree of bounding volumes. Subtrees are pruned out and distributed among processors. The upper portion of the tree is replicated on every processor. Cuts may be made at a fixed level in the tree (Ref. 28) or at varying levels based on estimates of the subtree workloads (Ref. 20).

perform the initial intersection tests on any ray, effectively decoupling the image-space and object-space partitionings.

The upper-level task operates on one ray at a time, checking it for intersections against the volume extents in the upper portion of the tree. When an intersection test descends to the level of a cut, a ray message is sent to the subtree task on the appropriate processor, which completes the intersection calculations for that subtree and returns the result to the processor which originated the request. Rather than waiting for the result to come back, the upper-level task tries to stay busy by processing additional rays.

Caspary and Scherson's method differs from Salmon and Goldsmith's primarily in the way in which load balancing is achieved. Salmon and Goldsmith adopt a static approach, partitioning the image space among the processors and allocating subtrees to processors based on workload estimates derived from a pre-processing step. Caspary and Scherson use a simpler random allocation strategy for subtrees, relying

instead on a demand-driven assignment of rays to the upper-level tasks.[3] In both cases, the location of the cuts is an important consideration. If cuts are too high in the tree, the number of subtrees will be small, and load balance will be poor. If cuts are too low in the tree, much of the object data will have to be replicated, limiting the size of scenes which can be accommodated.

Additional load balancing strategies for parallel ray-tracing are described in Badouel *et al.* (18).

G. Object-space to image-space mapping

Since distributed-memory systems have the potential to scale to higher performance levels, and since they are the current architecture of choice for parallel supercomputers, there is considerable interest in rendering algorithms which are suitable for this environment. The key to high performance on these systems is exploiting data locality to minimize remote memory references. At the same time, we want to partition the image and object data among the processors to achieve both performance scalability and data scalability. Unfortunately, these two goals are in conflict in parallel rendering algorithms.

To understand this conflict, we observe that, geometrically, rendering is a mapping from three-dimensional object space to two-dimensional image space (Figure 1). This mapping is not fixed, but instead depends on the modeling transformations and viewing parameters in use when a scene is rendered. If we assume that both the object and image data structures are partitioned among the processors, then at some point in the rendering pipeline data must be communicated among processors to satisfy the mapping from object space to image space. Because of the complexity and dynamic nature of the mapping function, the rendering algorithm perceives the communication pattern to be essentially arbitrary, with each processor sending data to, and receiving data from, a large number of other processors.

1. Sorting classifications

Managing this object-space to image-space communication is one of the central issues for parallel rendering algorithms on distributed-memory systems. To better understand this problem, Molnar *et al.* (14) developed a taxonomy of parallel rendering algorithms based on the point in the rendering pipeline at which the object-space to image-space mapping occurs. They classify algorithms as either *sort-first, sort-middle*, or *sort-last*, depending on whether the communication step occurs at the beginning, middle, or end of the rendering pipeline. Their analysis of the computation and communication costs of each approach concludes that none of them is inherently superior in all circumstances.

---

[3] Salmon and Goldsmith suggest a similar demand-driven strategy, but their emphasis is on the static subtree assignment technique.

Sort-first

Because sort-first algorithms perform object-space to image-space mapping early in the rendering process, they require an initial pre-processing step to assign primitives to the appropriate processors. This pre-processing step adds computation or communication overheads which are not present in sort-middle and sort-last methods. Sort-first is also subject to load imbalances due to uneven distribution of primitives within the image. On the other hand, sort-first has lower communication requirements than the other approaches when object primitives are tessellated into larger numbers of smaller polygons, or when image sampling ratios are high. This is because the data generated by these operations has already been mapped to the appropriate rasterization processor, and does not have to be relocated for subsequent processing. Sort-first can also take advantage of frame coherence, making it potentially attractive in animation applications.

Sort-middle

Sort-middle algorithms are straightforward since the communication step occurs at a natural break in the rendering pipeline, between the transformation and rasterization phases. If tessellation is used, communication costs can be high due to the large number of display primitives which are generated. Sort-middle also incurs overheads for splitting primitives at image boundaries (loss of spatial coherence). Like sort-first, sort-middle is susceptible to image-space load imbalances, but the impact is not as severe because more work is performed before the data is mapped into image space.

Sort-last

Sort-last algorithms are less sensitive to the distribution of primitives within the image, since most of the computations are performed using the initial object-space mapping of primitives to processors. However, communication is performed at the pixel or sub-pixel level, implying that bandwidth requirements are very high. Nonetheless, sort-last has been used in several commercial rendering systems (29, 30), and is an active area of current research (31, 32).

V. Design and Implementation Issues

As the above discussion suggests, the design space for parallel rendering algorithms is large and replete with trade-offs. How these trade-offs are resolved depends on a variety of factors, including application requirements and characteristics of the target architecture. In the following sections, we examine some of the issues which must be considered.

A. Hardware versus software systems

Perhaps the most fundamental distinction between parallel rendering designs is that of hardware-based

versus software-based systems. Hardware systems, ranging from specialized graphics computers to graphics workstations and add-on graphics accelerator boards, all employ dedicated circuitry to speed up the rendering task. In the simplest case, the graphics hardware may consist of a single microprocessor coupled to a video memory system. In other cases, custom integrated circuits directly implement highly parallel rendering pipelines in hardware. As a rule, the higher the target performance levels, the more specialized and the more parallel the hardware becomes.

The dedicated-hardware approach has been very successful, although commercial systems to date have been designed primarily for polygon rendering. Furthermore, the specialization which contributes to the high performance and cost-effectiveness of dedicated hardware also tends to limit its flexibility. Specialized lighting models, high-resolution imaging, and sophisticated rendering methods such as ray-tracing and radiosity must be implemented largely in software, with a corresponding degradation in performance.

One way to boost the performance of software-based renderers is to implement them on general-purpose parallel platforms, such as scalable parallel supercomputers or networks of workstations. On these systems, the processors are not specifically optimized for graphical operations, and communication networks often have bandwidth limitations and software overheads which are not found in hardware-based rendering systems. The challenge is to develop algorithms which can cope successfully with these overheads in order to realize the performance potential of the underlying hardware. Some recent examples indicate that this challenge can be met. Polygon renderers developed for Intel's Touchstone Delta system (16), Thinking Machines' CM-200 and CM-5 (33), and Cray's T3D (30) have achieved performance levels that equal or exceed those obtained on contemporary high-end graphics workstations such as Silicon Graphics' RealityEngine (12).

Software-based renderers are of interest on massively parallel architectures for another reason: massive data. The datasets produced by large-scale scientific applications can easily be hundreds of megabytes in size, and time-dependent simulations may produce this much data for hundreds or thousands of time-steps. Visualization techniques are imperative in exploring and understanding datasets of this size, but the sheer volume of data may make the use of detached graphics systems impractical or impossible. The alternative is to exploit the parallelism of the supercomputer to perform the visualization and rendering computations in place, eliminating the need to move the data. This has motivated recent work on software-based rendering systems which can be embedded in parallel applications to produce live visual output at run time (30, 34).

Networks of workstations and personal computers provide another type of platform which can be used by software-based parallel renderers. These systems are inexpensive and ubiquitous, and their processing

power and memory capacities are increasing dramatically. However, they tend to be connected by low-bandwidth networks, and suffer from high communication latencies due to operating system overheads and costly network protocols. For these reasons, they are best used in modest numbers for large granularity computations where high frame rates are not an overriding consideration. They are also well-suited for embarrassingly parallel applications which replicate the object database or exploit temporal parallelism to render entire frames locally. Examples of network-based systems include volume renderers (35, 36), radiosity renderers (37, 38), and Pixar's photorealistic NetRenderMan system (39).

Hardware-based renderers have a distinct price-performance advantage over software-based systems which run on massively parallel supercomputers. For similar levels of rendering performance, massively parallel systems cost ten to one hundred times more than specialized graphics workstations. This is partly due to the much larger component counts (including larger memories) in the massively parallel systems, and partly due to the lower levels of performance which are achieved in a general-purpose system relative to one that is specifically designed to perform graphics operations.

Specialized graphics hardware retains its price-performance advantage over networks of conventional workstations as well. One reason for this is the expense of components other than the processor, such as power supplies, backplanes, and network interfaces, which must be replicated in each workstation. More importantly, the higher communication costs associated with network-based systems have significant performance implications, giving specialized systems the edge for many applications.

B. Architectural considerations

The architecture of the target system, including the memory organization and programming paradigm, has a major impact on the design of software renderers. We now turn our attention to these issues.

1. Vector processing

Vectorization is a simple form of pipelining which can be viewed algorithmically as a data-parallel operation over individual elements of regular arrays. While vectorization has been used primarily in high-performance computer systems to speed up floating-point operations in numerical applications, it has also been applied to graphics at both the architectural and algorithmic level. Systems developed by Ardent (40) and Stellar (41) in the late 1980's coupled graphics display systems to floating-point vector processors. The vector units were used for object-level computations on geometric primitives as well as for general-purpose computation, while rasterization was performed using special-purpose hardware.

To take advantage of vectorization, standard rendering algorithms and data structures must be redesigned to perform identical operations on long sequences of contiguous data elements. This ensures that pipeline

startup costs will be effectively amortized, and facilitates the high-speed memory accesses needed to keep the pipeline running at full speed. Unfortunately, these requirements are sometimes at odds with the data irregularities which are encountered in the rendering process (see Section IV.F).

Perhaps because of these difficulties, the literature contains relatively few examples of vectorized renderers. Dyer and Whitman report on their experiences in vectorizing a z-buffered polygon renderer in (42). While certain operations (surface normal calculations, edge and span interpolation, and shading) vectorized well, others (clipping, edge extraction, sorting, and anti-aliasing) did not. In some cases, the overhead required to set up a vector operation exceeded the benefits. Overall performance of their vectorized implementation on a Convex C-1 was less than a factor of 2 better than an optimized scalar renderer.

Plunkett and Bailey (43) report somewhat better results with a vectorized ray-tracer for the CDC Cyber 205. Speedup factors of 10–30 were achieved for the computationally intensive ray/surface intersection calculations. Overall performance was approximately a factor of 6 better than a purely scalar implementation. While the vector algorithm performs many more arithmetic operations than its scalar counterpart, the higher speeds of the vector operations more than make up the difference. However, this performance comes with a price: the vector intersection computations require additional memory in proportion to the vector length, which in this case is 500. Another example of vectorization in a ray-tracing application can be found in (44).

2. *Shared vs. distributed memory*

As we noted in Sections IV.E and IV.F, shared-memory systems provide relatively efficient access to a global address space. This simple system model reduces the need to pre-partition major data structures, simplifies processor coordination, and maximizes the range of practical algorithms. The chief disadvantage is limited architectural scalability, which results in high memory latencies and contention for shared resources as the number of processors increases. To minimize these problems, good shared-memory algorithms must decompose the problem into tasks which avoid memory hot spots and keep critical sections and synchronization operations to a minimum. Since most shared-memory systems are augmented with processor caches and/or local memories, algorithms intended for these platforms must also be structured to achieve good locality in their memory reference patterns.

Distributed-memory systems offer improved architectural scalability, but often with higher costs for remote memory references. For this class of machines, managing communication is a primary consideration. Since the rendering process tends to generate large volumes of intermediate data which must be dynamically mapped from object space to image space, parallel renderers must pay special attention to this issue. In the absence of special hardware support, global operations and synchronization

may be particularly expensive, and the higher cost of data migration may favor static assignment of tasks and data.

3. *SIMD vs. MIMD*

In 1966, Flynn (45) proposed a taxonomy of computer architectures based on the number of instruction and data streams in the system. General-purpose parallel architectures fall into one of two categories, Single Instruction Multiple Data (SIMD), or Multiple Instruction Multiple Data (MIMD). In a pure SIMD architecture, every processor executes the same instruction at every clock cycle, in lock step. Conditionals are implemented by setting local mask bits which disable individual processors while some set of instructions is executed. Systems in this class typically provide large numbers of simple processors and instruction-level support for moving data on- and off-processor through the interconnection network. Examples of commercial SIMD systems include Thinking Machines' CM-2 and CM-200 and MasPar's MP-1 and MP-2.

By contrast, each processor in a MIMD architecture executes its own instruction stream, independently of every other processor. Processors are free to take divergent paths through a program, or even to execute completely different programs. Synchronization operations must be accomplished explicitly under software control. Recent systems in this class include the Intel Paragon, nCUBE 3, Thinking Machines CM-5, IBM SP2, and Cray Research T3D, among others.

Because they allow processors to respond to local differences in workload, MIMD architectures would appear to be a good match for the highly variable operation counts and data access patterns which characterize the rendering process (see Section IV.F). Furthermore, the MIMD environment lends itself to demand-driven and adaptive load balancing schemes, where processors work independently on relatively coarse-grained tasks. Numerous MIMD renderers have been implemented, on a variety of hardware platforms. They encompass all of the major rendering methods, including polygon rendering (16, 19, 26), volume rendering (17, 21, 27, 35, 46, 47), terrain rendering (48), ray-tracing (18, 20, 49, 50), and radiosity (51, 52, 53, 54).

Despite the apparent mismatch between the variability of the rendering process and the tight synchronization of SIMD architectures, a number of parallel renderers have demonstrated good performance on SIMD systems (33, 55, 56, 57). There are several reasons for this. First of all, the flexibility of MIMD systems imposes a burden on applications and operating systems, which must be able to cope with the arrival of data from remote sources at unpredictable intervals and in arbitrary order. This often results in complex communication and buffering protocols, particularly on distributed-memory message-passing systems. The lock-step operation of SIMD systems virtually eliminates these software overheads, resulting in communication costs which are much closer to the actual hardware speeds.

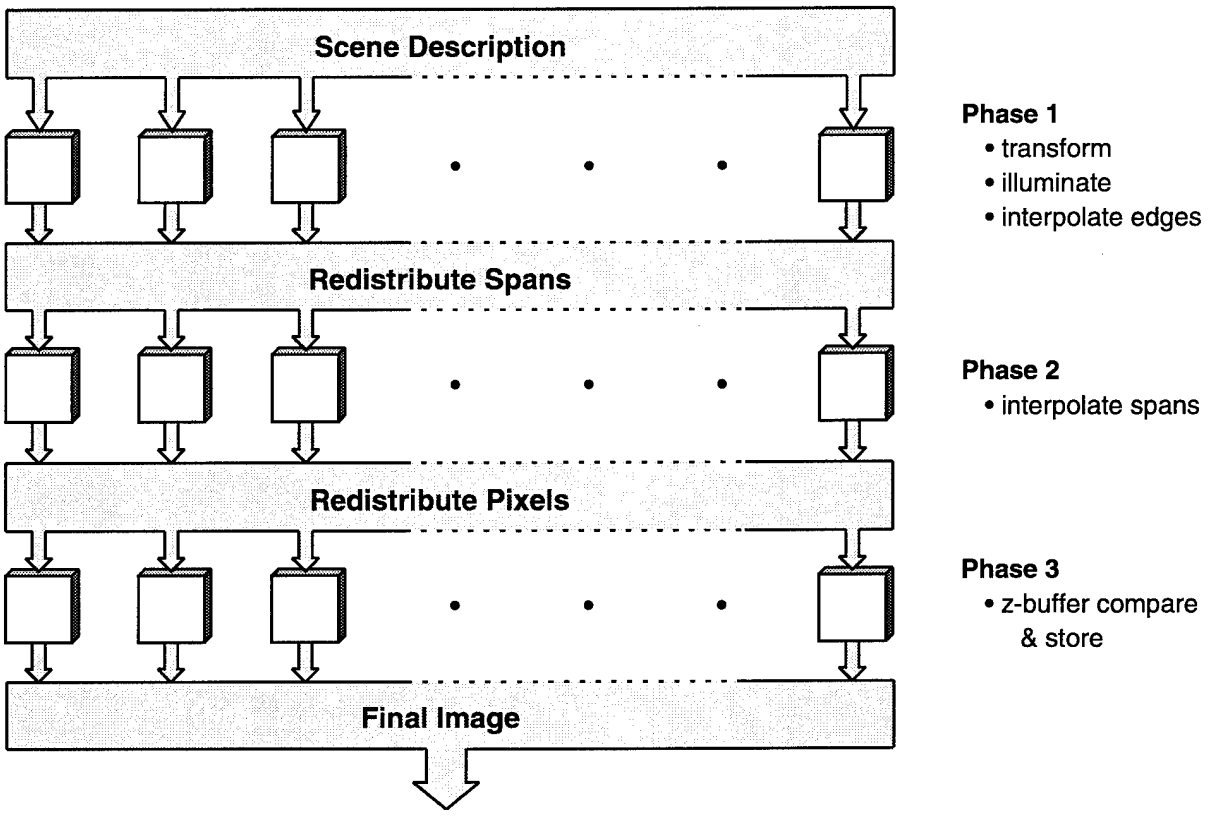

Figure 8. A three-phase rendering pipeline with two data redistribution steps. The extra communication step provides better load balancing and allows a SIMD implementation to operate on uniform data structures within each phase. (Based on Ref. 33.)

Secondly, it is often possible to structure algorithms as several distinct phases, each of which operates on a uniform data type. The rendering pipeline maps naturally onto this structure, and the regularity of the data structures within each phase leads to uniform operations, providing a good fit with the SIMD programming paradigm.

Finally, SIMD architectures usually contains thousands of simple processing elements. Because of their sheer numbers, good performance can often be achieved even though processors may not be fully utilized.

A data-parallel polygon renderer developed by Ortega, Hansen, and Ahrens for the CM-200 and CM-5[4] illustrates these principles (33). Instead of the single remapping step used by most algorithms (see

[4]Although the CM-5 is a MIMD system, it has a number of hardware and software features which allow it to support SIMD-style programs efficiently.

Section IV.G), their algorithm breaks the rendering pipeline into three phases. Figure 8 shows a simplified schematic of the basic approach. The first phase transforms and illuminates polygons, interpolating in the vertical direction to produce spans (Figure 5). The spans are then reassigned to processors in order to level the load prior to the horizontal scan-conversion phase, and the resulting pixels are reassigned once more during the final z-buffering phase. This multi-phase approach provides uniform operations within each phase, and efficient communication reduces the impact of the extra remapping step. The algorithm also takes advantage of low-overhead global summations to evaluate processor workloads at each iteration within the scan-conversion phase, an operation which would be prohibitively expensive on most large MIMD systems. The workload information is used to adaptively repartition span data when the imbalance becomes large enough to justify the expense. Despite these efforts, large disparities in polygon size degrade performance, and the algorithm works best for scenes composed of large numbers of small polygons, where variations in rasterization time are more tightly bounded.

SIMD architectures have also been used extensively for volume rendering. Hsu (56) developed an object-parallel volume renderer which employs a three-phase algorithm to regularize the data structures. His approach requires a single communication step for mapping partial ray segments to their image-space destinations for final compositing. Other researchers have adopted image-parallel approaches, holding the image data fixed and communicating object data instead (58, 59, 60).

C. Communication

For renderers which exploit both image and object parallelism, a high volume of interprocessor communication is inherent in the process (see Section IV.G). Managing this communication is a central issue in renderer design, and the choice of algorithm can have a significant impact on the timing, volume, and patterns of communication (14, 19, 21, 61). There are three main factors which need to be considered: *latency, bandwidth,* and *contention.* Latency is the time required to set up a communication operation, irrespective of the amount of data to be transmitted. Bandwidth is simply the amount of data which can be communicated over a channel per unit time. If a renderer tries to inject more data into a network than the network can absorb, delays will result and performance will suffer. Contention occurs when multiple processors are trying to route data through the same segment of the network simultaneously and there is insufficient bandwidth to support the aggregate demand.

The time for one processor to send data to another can be expressed by the following simple formula,

$$t_{comm} = t_{latency} + t_{transfer} + t_{delay}$$

where the total communication time, t_{comm}, is the sum of the latency ($t_{latency}$), data transfer time ($t_{transfer}$), and contention delay (t_{delay}). The transfer time is simply the volume of data to be sent divided by the channel

bandwidth. Latency can be better understood as the sum of three components,

$$t_{latency} = t_{send} + t_{route} + t_{recv}$$

where t_{send} is the time to initiate a transfer, t_{route} is the latency through the network, and t_{recv} is the time to receive the data at the other end.

The values of these variables differ widely depending on the system in use. Hardware latencies for sending, receiving, and routing messages are in the sub-microsecond range on many systems. However, software layers can boost these times considerably—measured send and receive latencies on message-passing systems often exceed the hardware times by a few orders of magnitude. Bandwidths exhibit similar variations, ranging from hundreds of kilobytes/second on workstation networks up to several gigabytes/second in dedicated graphics hardware. While latencies and bandwidths can usually be determined with reasonable precision, contention delays are more difficult to characterize, since they depend on dynamic traffic patterns which tend to be scene- and view-dependent.

A number of algorithmic techniques have been developed for coping with communication overheads in parallel renderers. A simple way to reduce latency is to accumulate short messages into large buffers before sending them, thereby amortizing the cost over many data items. Unfortunately, this technique does not scale well for the common case of object- to image-space sorting, since the communication pattern is generally many-to-many (16, 19). This implies that the number of messages generated per processor is $O(p)$, where p is the number of processors in the system. Assuming a fixed scene and image resolution and a p-way partitioning of the object and image data, the number of data items per processor is proportional to $1/p$, and the number of data items per message decreases as $1/p^2$. Hence overheads due to latency increase linearly with the number of processors and amortization of these overheads becomes increasingly ineffective.

One solution is to reduce the algorithmic complexity of the communication by using a multi-step delivery scheme, as proposed by Ellsworth (16). With this method, the processors are divided into approximately $\sqrt{p}$ groups, each containing roughly $\sqrt{p}$ processors. Data items intended for any of the processors within a remote group are accumulated in a buffer and transmitted together as a single large message to a forwarding processor within the destination group. The forwarding processor copies the incoming data items in into a second set of buffers on the basis of their final destinations, merging them with contributions from each of the other groups. The sorted buffers are then routed to their final destinations within the local group. Figure 9 illustrates this process. The net effect is that the number of messages generated per processor is reduced to $O\left(\sqrt{p}\right)$ and message lengths decline more slowly (proportional to $1/p^{3/2}$ rather than $1/p^2$), allowing latencies to be amortized more effectively. The algorithm does require the bulk of the data to be examined, copied, and transmitted a second time, so the benefits are only

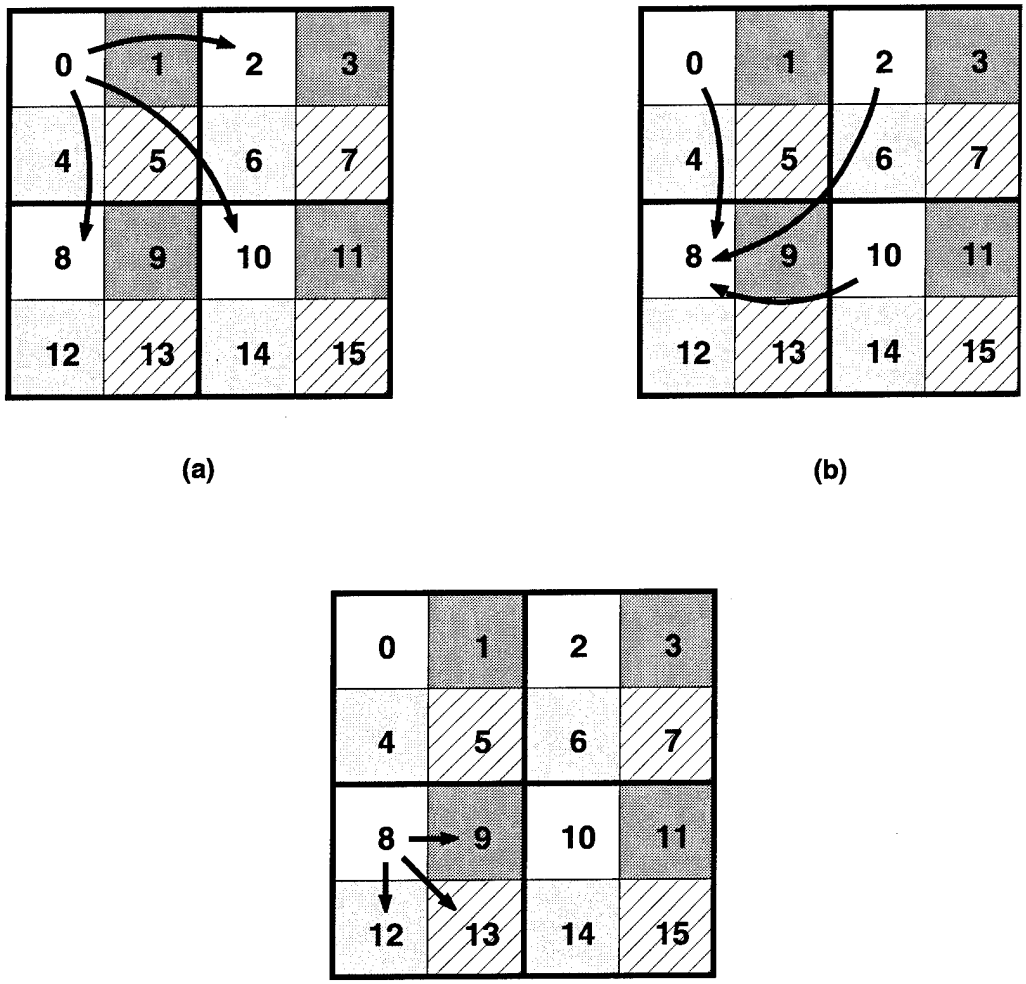

Figure 9. Two-step data redistribution. The image is partitioned into square regions which are assigned to 16 processors. Shading indicates sets of communicating processors for the first step. Data originating on processors 0, 2, or 10 which is destined for processors 9, 12, or 13 will pass through processor 8 first. *(a)* Step 1: Destinations of processor 0. *(b)* Step 1: Sources for processor 8. *(c)* Step 2: Final destinations of processor 8. (Based on Ref. 16.)

realized when latency is sufficiently high. Nonetheless, Ellsworth found the technique to be effective when rendering small datasets with large numbers of processors.

While helpful in reducing latency, large message buffers can contribute to contention delays when network bandwidth is insufficient, as Crockett and Orloff discovered in their experiments on an Intel iPSC/860 (19). The problem arises when a large volume of data is injected into the network within a short period of time. If the traffic fails to clear rapidly enough, processors must wait for data to arrive,

and performance suffers. The problem is most pronounced when workloads are evenly balanced, since processors tend to be communicating at about the same time. By using a series of intermediate-sized messages and asynchronous communication protocols, the load on the network can be spread out over time, and data transfer can be overlapped with useful computation.

D. Memory constraints

Memory consumption is another issue which must be considered when designing parallel renderers. Rendering is a memory-intensive application, especially with complex scenes and high-resolution images. As a baseline, a full-screen (1280 x 1024), full-color (24 bits/pixel), z-buffered image requires on the order of 10 MB of memory for the image data structures alone. The addition of features such as transparency and antialiasing can push memory demands into the hundreds of megabytes, a regime in which parallel systems or high-end graphics workstations are mandatory.

The structure of a parallel renderer can have a major impact on memory requirements, either facilitating memory-intensive rendering by providing data scalability (Section IV.E), or exacerbating the problem by requiring replicated or auxiliary data structures. Sort-middle polygon rendering is one example of an approach which exhibits good data scalability, since object and image data structures can be partitioned uniformly among the processing elements. The cost of image memory in these systems is essentially fixed. By contrast, some sort-last algorithms require the entire image memory to be replicated on every processor, increasing the cost in direct proportion to the number of processing elements in the system.

The issue of memory consumption involves many tradeoffs, and system designers must balance application requirements, performance goals, and system cost. For example, replicating object data in an image-parallel renderer can reduce or eliminate overheads for interprocessor communication, a strategy which may work well for rendering moderately complex scenes in low-bandwidth, high-latency environments, such as workstation networks. On the other hand, rendering algorithms which are embedded in memory-intensive applications must be careful to limit their own resource requirements to avoid undue interference with the application (34). In this case, data scalability may be a more important consideration than absolute performance.

As another example, message-passing renderers can often achieve performance improvements by aggregating data items into large buffers before sending them, as discussed in Section V.C. With fixed-length buffers and direct communication, the total space needed for message buffers increases in proportion to p^2, where p is the number of processors in the system. But as we observed in the previous section, the average amount of data to be communicated from each processor to every other processor decreases by the same factor, so a more intelligent strategy would scale the size of individual buffers by $1/p^2$, thereby holding the system-wide buffer space to a constant. However, latency amortization may

dictate that buffer sizes should not be allowed to drop below some efficiency threshold, so beyond a certain number of processors, the buffer space would begin to grow again. If Ellsworth's two-step sending method (16) is used instead, the total number of buffers needed in the system is reduced to $p^{3/2}$, allowing this cross-over point to be deferred to larger system sizes.

Some renderers operate in distinct phases, requiring each phase to complete before the next phase begins. This implies that intermediate results produced by each phase must be stored, rather than being passed along for immediate consumption. The amount of intermediate storage needed for each phase depends on the particular data items being produced, but in general is a function of the scene complexity. For complex scenes the memory overheads may be substantial, but they do exhibit data scalability, assuming the object data is partitioned initially.

E. Image assembly and display

High-performance rendering systems produce prodigious quantities of output in the form of an image stream. For full-screen, full-color animation (1280 x 1024 resolution, 24 bits/pixel, 30 frames/sec), a display bandwidth of 120 MB/s is required. Since most parallel renderers either partition or replicate the image space, the challenge is to combine pixel values from multiple sources at high frame rates. Failure to do so will create a bottleneck at the display stage of the rendering pipeline, limiting the amount of parallelism which can be effectively utilized.

1. Hardware solutions

The display problem is best addressed at the architectural level, and hardware rendering systems have adopted several different techniques. One approach is to integrate the frame buffer memory directly with the pixel-generation processors (12, 62, 63). Highly parallel, multiported busses or other specialized hardware mechanisms are then used to interface the distributed frame buffer to the video generation subsystem. Alternatively, the rasterization engines and frame buffer may be distinct entities, with pixel data being communicated from one to the other via a high-speed communication channel. One example is the Pixel-Planes 5 system (64), which uses a 640 MB/s token ring network to interconnect system components, including the pixel renderers and frame buffer. The PixelFlow system (31) pushes transfer rates a step further, using a pipelined image composition network with an effective interstage bandwidth in excess of 4 GB/s. The frame buffer resides at the terminus of the pipeline, acting as a sink for the final composited pixel values.

2. *Considerations for general-purpose systems*

Sustaining high frame rates with general-purpose parallel computers is problematic, since these systems typically lack specialized features for image integration and display. There are two principal issues, assembling finished images from distributed components, and moving them out of the system and onto a display. The bandwidth of the interprocessor communication network is an important consideration for the image assembly phase, since high frame rates cannot be sustained unless image components can be retrieved rapidly from individual processor memories. Several current systems, including the Intel Paragon and Cray T3D, provide networks with transfer rates in excess of 100 MB/s, which is more than adequate for interactive graphics. The challenge on these systems is to orchestrate the image retrieval and assembly process so that the desired frame rates can be achieved. In the absence of multiported frame buffers, the image stream must be serialized, perhaps with some ordering imposed, and forwarded to an external device interface.

Assuming that the internal image assembly rate is satisfactory, the next bottleneck is the I/O interface to the display. The typical configuration on current systems uses a HIPPI interface (65) attached to an external frame buffer device. While many of the existing implementations fail to sustain the 100 MB/s transfer rate of the HIPPI specification, the technology is improving, and either HIPPI or emerging technologies such as ATM (66) are likely to provide sufficient external bandwidth in the near future.

3. *Algorithmic approaches*

Most software-based parallel renderers either partition the image memory across processors, or else replicate it everywhere. In the first scenario, the image partitions must be assembled to produce a complete image. This may be done either internally on the parallel machine, or externally in the display system. *Internal assembly* implies that memory for a complete image must be allocated somewhere in the system, a requirement which is potentially at odds with the desire for data scalability. *External assembly* can occur in any of several places, including a host or front-end computer, an addressable external frame buffer, or on secondary storage.

For renderers which replicate the image memory on every processor, generating the finished product usually requires the individual contributions to be z-buffered or composited with a sort-last communication phase. As we noted in Section IV.G.1, this works best on architectures with high-bandwidth internal networks. The issue of memory allocation for the final result is moot, since renderers which adopt this strategy have already incurred this cost many times over (once on each processor).

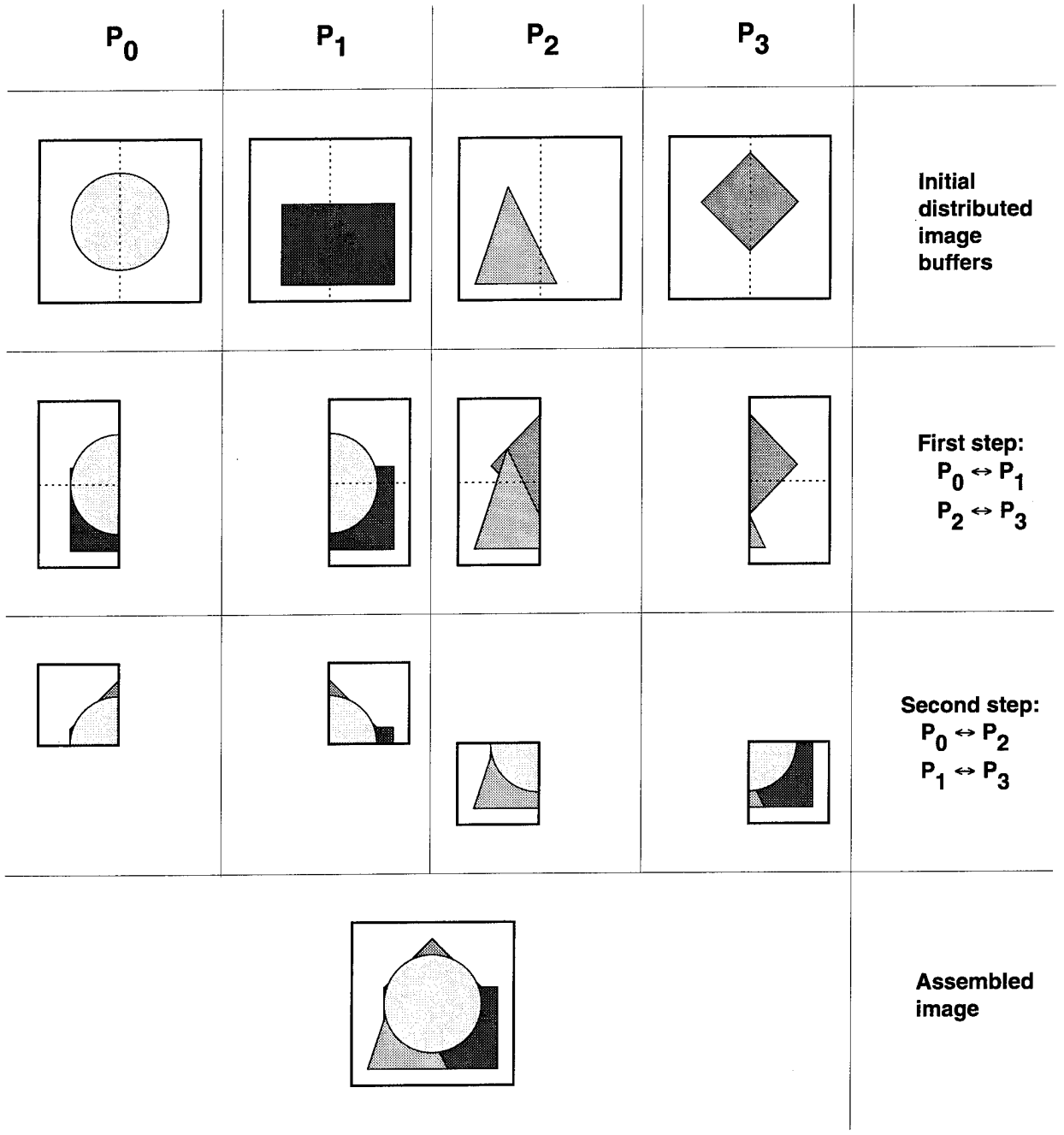

Figure 10. Binary-swap image compositing. At each step, the image is partitioned and every processor is responsible for compositing two image segments. (Based on Ref. 35.)

Several algorithmic strategies are available for image assembly and composition. The naive approach is to have a designated processor or host accept the contributions from all of the other processors, performing the appropriate z-buffering or compositing operations for each contribution. While this may be acceptable with small numbers of processors, it results in poor utilization and does not scale well, since

the receiving processor is a serial bottleneck. An obvious improvement is to merge image components in a tree-structured fashion, combining contributions at each level in the tree. This results in somewhat better utilization of both processors and the communication network, and runs in $O(\log p)$ time. Ma *et al.* (34) observed that even with the tree-merging approach, processors are under-utilized, and those at higher levels of the tree tend to have disproportionately high workloads. They devised an alternate scheme for compositing images which also runs in logarithmic time, but keeps most or all of the processors busy at every stage. The key idea in their *binary-swap* compositing method is to split the image at each step, with pairs of processors operating on different subimages. At the end of the process, the image is partitioned among all of the processors, requiring a final image assembly step to retrieve all of the pieces. Figure 10 illustrates this procedure with four processors.

The pipelined compositing strategy used in the PixelFlow system (31) can also be implemented in software. Silva and Kaufman (67) adopt this approach in a distributed-memory volume renderer for the Intel iPSC/860 and Paragon systems. In order to improve processor utilization, several frames of animation are active in the system simultaneously (an example of temporal parallelism), with processors alternating between rendering and compositing tasks. A potential difficulty with pipelined image composition is high end-to-end latency as the system scales up and the length of the pipeline increases. Applications which require rapid response times, such as virtual reality and real-time simulation, may prefer to use a logarithmic image assembly method.

4. *Remote image display*

The utility of directly-attached frame buffers in conjunction with large-scale parallel systems is limited, since users are often located at geographically remote sites. This has prompted a number of researchers to examine the potential for transmitting images across local-area and wide-area networks. One example is the display system used in the PGL graphics library (34). PGL partitions its image memory, assigning scanlines to processors in interleaved fashion (Figure 6c). Image assembly occurs externally on the receiving workstation, eliminating the need for a complete image buffer to be allocated within the parallel system. To reduce the volume of output to a manageable level, each processor compresses its local scanlines by determining which pixels have changed since the previous frame, and then run-length encoding the differences. The resulting contributions from each processor are merged into large packets which are sent across the network to a remote workstation for decompression, image assembly, and display. While straightforward, this technique has several advantages, including exploitation of both temporal and spatial image coherence, lossless encoding, embarrassingly parallel image compression, and rapid sequential decompression. Although performance depends heavily on factors such as network traffic and image resolution and content, this technique can provide up to a few frames per second across Ethernet (68) and FDDI (69) networks.

VI. Examples of Parallel Rendering Systems

As we noted in Section I.A, virtually all current graphics systems incorporate parallelism in one form or another. We have illustrated the preceding discussion with a number of examples. In this section, we round out our survey by examining some additional representative systems, running the gamut from specialized graphics computers to software-based terrain and radiosity renderers. Our coverage is by no means complete—many more examples can be found in the literature. Readers are encouraged to explore the references and the suggested readings at the end of this article for more information.

A. Polygon rendering and multi-purpose architectures

One of the earliest graphics architectures to exploit large-scale data parallelism was Fuchs and Poulton's classic Pixel-Planes system (62). Pixel-Planes parallelized the rasterization and z-buffering stages of the polygon rendering pipeline by augmenting each pixel with a simple bit-serial processor which was capable of computing color and depth values from the plane equations which described each polygon. The pixel array operated in SIMD fashion, taking as input a serial stream of transformed screen-space polygons generated by a conventional front-end processor. While Pixel-Planes provided massive image parallelism, it suffered from poor processor utilization, since only those processors which fell within the bounds of a polygon were active at any given time. The serial front-end processor also proved to be a bottleneck as rasterization performance and scene complexities increased in subsequent generations of the architecture.

The Pixel-Planes 5 architecture (64) rectifies these deficiencies. Instead of a single large array of image processors, it incorporates several smaller ones which can be dynamically reassigned to screen regions in demand-driven fashion. The serial front-end is replaced by a collection of general purpose transformation processors which operate in MIMD mode. The transformation and rasterization units are connected by a high-speed ring network, allowing data to flow in both directions. In addition to improved load balance and higher performance, the flexibility of the architecture allows it to be applied to a broader range of applications, including volume rendering and radiosity techniques. These architectural improvements are not without cost, however. The dynamic assignment of rasterization units to screen space requires the front-end processors to sort geometric primitives by screen region before initiating the rasterization phase. This implies both a computational overhead and a memory penalty for storing the sorted primitives.

Pixel-Planes 5 is a classic example of a sort-middle architecture, with global communication occurring at the break between the transformation and rasterization phases. By contrast, the newer PixelFlow design (31) implements a sort-last architecture, in which each processing node incorporates a full graphics pipeline. Object parallelism is achieved by distributing primitives across the nodes, while pixel parallelism is provided by a Pixel-Planes-style SIMD rasterizer on each node. The sort-last strategy

necessitates a bandwidth-intensive image composition step to integrate the partial images from each rasterizer, but this is accomplished using unidirectional nearest-neighbor communication in a 256-bit-wide pipelined interconnect (see Section V.E.1).

AT&T's Pixel Machine (63) combines pipelined parallelism with data parallelism in a programmable MIMD architecture. The system includes one or two 9-stage pipelines for object-level processing and an array of up to 64 pixel processors for image-level operations. Like Pixel-Planes, the frame buffer is integrated with the pixel processors, but in the Pixel Machine each processor is responsible for multiple pixels, distributed in a two-dimensional interleaved fashion. Each processing element is independently programmable and capable of floating-point operations, resulting in an architecture which is adaptable to a variety of rendering and image-processing tasks. As with Pixel-Planes, the limited parallelism provided by the front-end pipelines has proven to be a bottleneck when rendering small primitives.

B. Volume rendering and ray-tracing architectures

Graphics architectures have also been developed specifically for volume rendering and ray-tracing applications. In volume rendering, one of the keys to performance is providing high-bandwidth, conflict-free access to the volume data. This has prompted the development of specialized volume memory structures which allow simultaneous access to multiple data values. Kaufman and Bakalash's Cube system (70) introduced an innovative 3D voxel buffer which facilitates parallel access to cubes of volumetric data. A linear array of simple SIMD comparators simultaneously evaluates a complete shaft or "beam" of voxels oriented along any of the three principal axes (x, y, or z). The output of the comparator network is a single voxel chosen on the basis of transparency, color, or depth values. By iterating through the other two dimensions, the complete volume can be scanned at interactive rates. The most recent version of the Cube architecture, Cube-3 (71), supports a more general ray-casting model, and incorporates additional parallel and pipelined hardware to support arbitrary viewing angles, perspective projections, and trilinear interpolation of ray samples.

Knittel and Straßer (72) adopt a somewhat different approach with a VLSI-based volume rendering architecture intended for desktop implementation. Memory is organized into eight banks in order to provide parallel access to the sets of neighboring voxels which are needed for trilinear interpolation and gradient computations at sample points along rays. The basic design consists of a volume memory plus four specialized VLSI function units arranged in a pipeline. One function unit performs ray-casting and computes sample points along each ray, generating addresses into the volume memory. A second unit accepts the eight data values in the neighborhood of each sample and performs trilinear interpolation and gradient computations. A third unit computes color intensities for each sample point using a Phong illumination model, while the fourth unit composites the samples along each ray to produce a final pixel

value. To obtain higher performance, the entire pipeline can be replicated, with subvolumes of the data being stored in each volume memory.

The SIGHT architecture (73) was designed specifically to support image-parallel ray-tracing. The image space is partitioned across processors, with each processor responsible for tracing those rays which emanate from its local pixels. Interprocessor communication is largely avoided by replicating the object database in each processor's memory. An additional level of parallelism is achieved through the use of multiple floating-point arithmetic units in each processing element to speed up the ray intersection calculations.

C. Radiosity renderers

Radiosity methods produce exceptionally realistic illumination of enclosed spaces by computing the transfer of light energy among all of the surfaces in the environment. Strictly speaking, radiosity is an illumination technique, rather than a complete rendering method. However, radiosity methods are among the most computationally-intensive procedures in computer graphics, making them an obvious candidate for parallel processing.

Because the quality of a radiosity solution depends in part on the resolution used to compute energy transfers, the polygons which describe objects are typically subdivided into small patches. Energy transfers between patches are computed using geometric constructions known as *form factors*. In the basic radiosity method, form factors must be computed from every patch in the environment to every other patch. Because of this quadratic complexity, form factor computations constitute the primary expense in radiosity methods. Hence, parallel implementations have focused on speeding up the generation of form factors.

Although radiosity solutions can be computed directly by solving the system of equations which describes the energy transfers between surfaces, all of the form factors must be generated first, resulting in lengthy solution times which preclude interactive use. For this reason, an alternate iterative approach known as *progressive refinement* (74) has become popular. In this technique, the patch with the highest energy level at each iteration is selected as the *shooting patch*, and energy is transferred from it to other patches in the environment. This process repeats until the maximum level of untransmitted energy drops below some specified threshold. In this way, an initial approximation of the global illumination can be computed relatively quickly, with subsequent refinements resulting in incremental improvements to the image quality.

Many of the parallel radiosity methods described in the literature attempt to speed up the progressive refinement process by computing energy transfers from several shooting patches in parallel (i.e., several

iterations are performed simultaneously) (37, 38, 51, 52, 53, 54). Because the time to complete an iteration can vary considerably depending on the geometric relationships between patches, load imbalance can seriously degrade overall performance. Several implementations compensate for this using a demand-driven strategy in which multiple worker processes independently compute form factors for different shooting patches (37, 38, 54). With this strategy, the complete patch database is usually replicated on every processor, and a separate master process picks shooting patches and completes the energy transfers using vectors of form factors generated by the workers. This approach has several drawbacks, including a lack of data scalability for complex scenes and the tendency for the master process to become a bottleneck as the number of workers increases.

The alternative is to distribute the patch database and radiosity computations across all of the processors. This strategy necessitates global communication in order to compute form factors and complete the energy transfers from shooting patches. Çapın *et al.* (53) use a simple ring network, circulating patch data and local results from processor to processor in pipelined fashion to obtain global solutions. Because performance is limited at each step of the computation by the slowest processor, load imbalances can have a profound effect on overall performance. By ensuring that patches belonging to the same object are scattered across processors, variations in workload due to spatial locality are minimized, and a rough static load balance is maintained. Additional examples of radiosity renderers which use distributed databases can be found in (51) and (52).

The strategy of processing multiple shooting patches in parallel perturbs the order of execution found in the sequential version of the progressive refinement algorithm, and this can lead to slower convergence, partially offsetting the benefits of parallel execution. While this effect is minimal when only a few shooting patches are active (75), it becomes more pronounced as the number of processors increases and the order of shooting patch selection deviates further from the optimum (53). In order to exploit massive parallelism, a different approach is needed.

In contrast to the previous examples, which all target MIMD systems with modest numbers of processors, Varshney and Prins describe a SIMD radiosity renderer implemented on a MasPar MP-1 with 4096 SIMD processing elements (76). As in Çapın *et al.*'s algorithm, patches are distributed uniformly among the processors. At each iteration, a global reduction operation is used to find the shooting patch with the highest energy, thus maintaining the convergence properties of the sequential algorithm. Once the shooting patch is selected, all of the other patches in the environment are projected onto the shooting patch's *single-plane* (38), where they are scan-converted and z-buffered to determine visibility from the shooting patch. Form factors are obtained by accumulating contributions from the single-plane "pixels", and energy transfers are performed in parallel for each patch using the results from the form factor computations. While this algorithm is able to exploit the massive parallelism of its target architecture,

load imbalances in the scan conversion phase are found to be significant, and further static or dynamic load balancing measures appear to be in order.

D. Terrain rendering

In terrain rendering, the problem is to generate a plausible representation of a real or imaginary landscape as viewed from some point on or above the surface. Typically the viewpoint will change over time, often under interactive control, and in some applications additional objects such as vegetation, buildings, or vehicles must be included in the scene. Terrain rendering techniques have been widely applied in areas such as flight simulation, scientific data analysis and exploration, and the creation of virtual landscapes for entertainment or artistic purposes. The need for high-quality images, high frame rates, rapid response to changes in viewpoint, and the ability to navigate through large datasets has stimulated the development of parallel terrain rendering techniques.

Although a variety of techniques can be used to render terrain, most of the parallel methods described in the literature begin with an aerial or satellite image of an actual planetary surface. This image is registered with a separate elevation dataset of the same region, typically represented by a two-dimensional grid with an associated height field. The problem, then, is to assign an elevation value to pixels in the input image and project them onto a display with hidden surfaces eliminated. This technique is known as *forward projection*, in contrast to ray-casting methods which begin at the eye point and project rays through display pixels into the scene. With the forward projection approach, care must be taken to account for the mismatch between input and output image projections, filling in gaps in the output image and compositing input pixels which map to the same location in screen space.

Kaba *et al.* (57, 77) have developed data-parallel terrain rendering techniques for the Princeton Engine, a programmable SIMD system originally developed for real-time processing of digital video (78). Their methods utilize an object-parallel task decomposition, distributing the input image and elevation datasets among the processors by assigning complete columns of pixels to processors. Before projecting the data onto the display, it must be rotated and scaled to account for the viewing direction and altitude. This is accomplished by decomposing the necessary transformations into a sequence of simple shear and shear/scale operations. To avoid costly interprocessor communication, horizontal shears (along pixel rows) are decomposed into a transpose plus a vertical shear (which requires only local memory references due to the column-wise data decomposition). The image transpose is performed efficiently using the Princeton Engine's specialized output sequencer and image feedback channel. Hidden surface elimination is accomplished by scanning the transformed data from front-to-back, one horizontal scanline at a time. As each scanline is processed, a horizon line is updated; only those pixels which lie above the current horizon line will be visible. The column-oriented image partitioning assures that each horizontal

scan can be performed as a data-parallel operation. The system is capable of rendering terrain fly-overs at 30 frames/sec using 512 x 512 resolution and 8-bit color, or 15 frames/sec with 24-bit color.

At the Jet Propulsion Laboratory, Li and Curkendall (48) have developed techniques for rendering planetary surfaces using a variety of large-scale distributed-memory architectures, including Intel's iPSC/860, Delta, and Paragon systems, and Cray's T3D. Like Kaba, they use surface images registered with elevation data, and project object-space pixels into screen space. While their initial methods partitioned the input data by horizontal slices and assigned them to processors in interleaved fashion, more recent implementations (79) decompose the data into square regions which are randomly assigned to processors. The random assignment provides a measure of stochastic load balancing, reducing sensitivity to hot spots in the data which may occur when the view zooms in on small terrain regions.

For hidden surface elimination, Li and Curkendall use a standard z-buffer technique, based on the distance from the view point to individual terrain pixels. The output image memory is replicated on every processor, with each processor projecting its local terrain pixels into its local output buffer. This necessitates a sort-last image composition phase, which is performed using a logarithmic merge similar to Ma *et al.*'s binary-swap method (see Section V.E.3). A final image assembly step is required to retrieve completed sub-images from each processor and route them to secondary storage or an external display. JPL's parallel terrain renderers have been used to produce renowned fly-overs of Mars and Venus using data from NASA's planetary probes. Some of the datasets involved are quite large (in excess of a gigabyte), making large-scale parallel systems particularly attractive for this application.

While the two previous examples both exploited data parallelism, other approaches are certainly possible. Wright and Hsieh (80) describe a pipelined terrain rendering algorithm which has been implemented in hardware. As in the other examples, a forward projection technique is used to map from object to image space, but the surface data and objects in the scene are represented as specialized volume elements (voxels). The architecture consists of two pipelines, one for voxel processing and one for pixel processing. The output of the voxel pipeline feeds the pixel pipeline, so conceptually the system can be viewed as one long pipeline. The voxel pipeline scans through the database, generating columns of voxels which are illuminated, transformed into viewing coordinates, and rasterized into pixels. The pixel pipeline projects pixels from polar viewing coordinates into screen space, performs haze, translucency, and z-buffering calculations, and normalizes pixel intensities. A variety of techniques are applied at different levels in the pipeline to reduce temporal and spatial aliasing. Objects in motion relative to the terrain are rendered using additional passes through the pipeline. The hardware implementation is capable of rendering 10 frames/sec at 384 x 384 resolution, a speedup of more than three orders of magnitude over a software-based sequential implementation.

VII. Summary

As the above discussion illustrates, parallel processing techniques have been applied to virtually every computationally-intensive task in computer graphics. Architectural platforms range from simple co-processors to specialized VLSI circuitry to general-purpose parallel supercomputers. At every step, the algorithm or architecture designer is faced with a wide range of implementation strategies and a complex series of tradeoffs. A successful parallel rendering design must take into account application requirements, architectural parameters, and algorithmic characteristics. As the rapidly growing performance of rendering systems indicates, there have been numerous successes, but these are balanced by other attempts which have stumbled. Many challenges remain, and the discipline of parallel rendering is likely to be an active one for years to come.

Acknowledgments

The author would like to thank Tony Apodaca, Chuck Hansen, Scott Whitman, Craig Wittenbrink, Sam Uselton, and the staff of NASA Langley's Technical Library for their assistance in researching this article. David Banks and John van Rosendale provided valuable feedback on a draft of the manuscript.

References

1. Ninke, W. H. Graphic 1 — A Remote Graphical Display Console System. *1965 Fall Joint Computer Conference*, AFIPS Conference Proceedings, Vol. 27, Part 1, 1965, 839-846.

2. Myer, T. H., and Sutherland, I. E. On the Design of Display Processors. *Communications of the ACM*, Vol. 11, No. 6, June 1968, 410-414.

3. Sproull, R. F., and Sutherland, I. E. A Clipping Divider. *1968 Fall Joint Computer Conference*, AFIPS Conference Proceedings, Vol. 33, Part 1, December 1968, 765-775.

4. Schumacker, R., Brand, B., Gilliland, M., and Sharp, W. *Study for Applying Computer-Generated Images to Visual Simulation*. Report AFHRL-TR-69-14, Air Force Human Resources Laboratory, September 1969.

5. Schacter, B. J., ed. *Computer Image Generation*. Wiley-Interscience, 1983.

6. Fuchs, H. Distributing A Visible Surface Algorithm Over Multiple Processors. *Proceedings ACM National Conference*, October 1977, 449-451.

7. Fuchs, H., and Johnson, B. W. An Expandable Multiprocessor Architecture for Video Graphics. *Proceedings of the 6th Annual ACM-IEEE Symposium on Computer Architecture*, April 1979, 58-67.

8. Parke, F. I. Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems. *Computer Graphics*, Vol. 14, No. 3, July 1980, 48-56.

9. Clark, J. A VLSI Geometry Processor for Graphics. *Computer*, Vol. 13, No. 7, July 1980, 59-68.

10. Clark, J. The Geometry Engine: A VLSI Geometry System for Graphics. *Computer Graphics,* Vol. 16, No. 3, July 1982, 127-133.

11. Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K. LINKS-1: A Parallel Pipelined Multicomputer System for Image Creation. *Proceedings 10th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, 1983, 387-394.

12. Akeley, K. RealityEngine Graphics. *Computer Graphics Proceedings, Annual Conference Series, 1993*, ACM SIGGRAPH, July 1993, 109-116.

13. Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys*, Vol. 6, No. 1, March 1974, 1-55.

14. Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, 23-32.

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

16. Ellsworth, D. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, 33-40.

17. Mackerras, P., and Corrie, B. Exploiting Data Coherence to Improve Parallel Volume Rendering. *IEEE Parallel and Distributed Technology*, Vol. 2, No. 2, Summer 1994, 8-16.

18. Badouel, D., Bouatouch, K., and Priol, T. Distributing Data and Control for Ray Tracing in Parallel. *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, 69-77.

19. Crockett, T. W., and Orloff, T. Parallel Polygon Rendering for Message-Passing Architectures. *IEEE Parallel and Distributed Technology*, Vol. 2, No. 2, Summer 1994, 17-28.

20. Salmon, J., and Goldsmith, J. A Hypercube Ray-tracer. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Vol. II, *Applications*, G. C. Fox, ed., ACM Press, January 1988, 1194-1206.

21. Neumann, U. Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, 49-58.

22. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S. The Stanford Dash Multiprocessor. *Computer*, Vol. 25, No. 3, March 1992, 63-79.

23. Kessler, R. E., and Schwarzmeier, J. L. Cray T3D: A New Dimension for Cray Research. *Digest of Papers, COMPCON Spring '93*, IEEE Computer Society Press, February 1993, 176-182.

24. Convex Computer Corporation. *Convex Exemplar System Overview*. 1994.

25. Whelan, D. S. *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*. Ph.D. dissertation, California Institute of Technology, 1985.

26. Whitman, S. Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, 41-48.

27. Nieh, J., and Levoy, M. Volume Rendering on Scalable Shared-Memory MIMD Architectures. *Proceedings of the 1992 Workshop on Volume Visualization*, ACM Press, October 1992, 17-24.

28. Caspary, E., and Scherson, I. D. A Self-Balanced Parallel Ray-Tracing Algorithm. In *Parallel Processing for Computer Vision and Display*, Addison-Wesley, 1989, 408-419.

29. Evans and Sutherland Computer Corporation. *Freedom Series Technical Report*. 1992.

30. Cray Research, Inc. *Cray Animation Theater*. 1994.

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

32. Cox, M., and Hanrahan, P. A Distributed Snooping Algorithm for Pixel Merging. *IEEE Parallel and Distributed Technology*, Vol. 2, No. 2, Summer 1994, 30-36.

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

34. Crockett, T. W. Design Considerations for Parallel Graphics Libraries. ICASE Report No. 94-49 (NASA CR 194935), Institute for Computer Applications in Science and Engineering, Hampton, Virginia, June 1994.

35. Ma, K.-L., Painter, J. S., Hansen, C. D., and Krogh, M. F. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications,* Vol. 14, No. 4, July 1994, 59-68.

36. Giertsen, C., and Petersen, J. Parallel Volume Rendering on a Network of Workstations. *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, November 1993, 16-23.

37. Puech, C., Sillion, F., and Vedel, C. Improving Interaction with Radiosity-based Lighting Simulation Programs. *Computer Graphics*, Vol. 24, No. 2 (*Proceedings of the 1990 Symposium on Interactive 3D Graphics*), March 1990, 51-57.

38. Recker, R. J., George, D. W., and Greenberg, D. P. Acceleration Techniques for Progressive Refinement Radiosity. *Computer Graphics*, Vol. 24, No. 2 (*Proceedings of the 1990 Symposium on Interactive 3D Graphics*), March 1990, 59-66.

39. Pixar. *PhotoRealistic RenderMan Toolkit v3.5 Reference Manual*. 1994.

40. Diede, T., Hagenmaier, C., Miranker, G., Rubinstein, J., and Worley, W. The Titan Graphics Supercomputer Architecture. *Computer*, Vol. 21, No. 9, September 1988, 13-30.

41. Apgar, B., Bersack, B., and Mammem, A. A Display System for the Stellar Graphics Supercomputer Model GS1000. *Computer Graphics*, Vol. 22, No. 4, August 1988, 255-262.

42. Dyer, S., and Whitman, S. A Vectorized Scan-Line Z-Buffer Rendering Algorithm. *IEEE Computer Graphics and Applications*, Vol. 7, No. 7, July 1987, 34-45.

43. Plunkett, D. J., and Bailey, M. J. The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed. *IEEE Computer Graphics and Applications*, Vol. 5, No. 8, August 1985, 52-60.

44. Max, N. L. Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset. *Computer Graphics*, Vol. 15, No. 3, August 1981, 317-324.

45. Flynn, M. J. Very High-Speed Computing Systems. *Proceedings of the IEEE*, Vol. 54, 1966, 1901-1909.

46. Challinger, J. Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 81-88.

47. Camahort, E., and Chakravarty, I. Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 89-96.

48. Li, P. P., and Curkendall, D. W. Parallel Three Dimensional Perspective Rendering. *Proceedings of the Second European Workshop on Parallel Computing*, March 1992, 320-331.

49. Green, S. A., and Paddon, D. J. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer*, Vol. 6, No. 2, March 1990, 62-73.

50. Lefer, W. An Efficient Parallel Ray Tracing Scheme for Distributed Memory Parallel Computers. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 77-80.

51. Feda, M. and Purgathofer, W. Progressive Refinement Radiosity on a Transputer Network. *Photorealistic Rendering in Computer Graphics: Proceedings of the Second Eurographics Workshop on Rendering*, Springer-Verlag, May 1991, 139-148.

52. Chalmers, A. G., and Paddon, D. J. Parallel Processing of Progressive Refinement Radiosity Methods. *Photorealistic Rendering in Computer Graphics: Proceedings of the Second Eurographics Workshop on Rendering*, Springer-Verlag, May 1991, 149-159.

53. Çapın, T. K., Aykanat, C., and Özgüç, B. Progressive Refinement Radiosity on Ring-Connected Multicomputers. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 71-76.

54. Ng, A., and Slater, M. A Multiprocessor Implementation of Radiosity. *Computer Graphics Forum*, Vol. 12, No. 5, December 1993, 329-342.

55. Lin, T. T. Y., and Slater, M. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, Vol. 7, No. 4, 1991, 187-199.

56. Hsu, W. M. Segmented Ray Casting for Data Parallel Volume Rendering. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 7-14.

57. Kaba, J., Matey, J., Stoll, G., Taylor, H., and Hanrahan, P. Interactive Terrain Rendering and Volume Visualization on the Princeton Engine. *Proceedings Visualization '92*, IEEE Computer Society Press, October 1992, 349-355.

58. Vézina, G., Fletcher, P. A., and Robertson, P. K. Volume Rendering on the MasPar MP-1. *Proceedings 1992 Workshop on Volume Visualization*, ACM Press, October 1992, 3-8.

59. Schröder, P. and Salem, J. B. Fast Rotation of Volume Data on Data Parallel Architectures. *Proceedings Visualization '91*, IEEE Computer Society Press, October 1991, 50-57.

60. Schröder, P., and Stoll, G. Data Parallel Volume Rendering as Line Drawing. *Proceedings 1992 Workshop on Volume Visualization*, ACM Press, October 1992, 25-31.

61. Jensen, D. W., and Reed, D. A. A Performance Analysis Exemplar: Parallel Ray Tracing. *Concurrency: Practice and Experience*, Vol. 4, No. 2, April 1992, 119-141.

62. Fuchs, H., and Poulton, J. Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine. *VLSI Design*, Third Quarter 1981, 20-28.

63. Potmesil, M., and Hoffert, E. M. The Pixel Machine: A Parallel Image Computer. *Computer Graphics*, Vol. 23, No. 3, July 1989, 69-78.

64. Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics*, Vol. 23, No. 3, July 1989, 79-88.

65. Hughes, J. P. HIPPI. *Proceedings 17th Conference on Local Computer Networks*, IEEE Computer Society Press, September 1992, 346-354.

66. Vetter, R. J. ATM Concepts, Architectures, and Protocols. *Communications of the ACM*, Vol. 38, No. 2, February 1995, 30-38.

67. Silva, C. T., and Kaufman, A. E. Parallel Performance Measures for Volume Ray Casting. *Proceedings Visualization '94*, IEEE Computer Society Press, October 1994, 196-203.

68. Shoch, J. F., Dalal, Y. K., and Redell, D. D. Evolution of the Ethernet Local Computer Network. *Computer*, Vol. 15, No. 8, August 1982, 10-27.

69. Ross, F. E. An Overview of FDDI: The Fiber Distributed Data Interface. *IEEE Journal on Selected Areas in Communications*, Vol. 7, No. 7, September 1989, 1043-1051.

70. Kaufman, A., and Bakalash, R. Memory and Processing Architecture for 3D Voxel-Based Imagery. *IEEE Computer Graphics and Applications*, Vol. 8, No. 6, November 1988, 10-23.

71. Pfister, H., Kaufman, A., and Chiueh, T. Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization. *Proceedings 1994 Symposium on Volume Visualization*, ACM SIGGRAPH, October 1994, 75-82.

72. Knittel, G., and Straßer, W. A Compact Volume Rendering Accelerator. *Proceedings 1994 Symposium on Volume Visualization*, ACM SIGGRAPH, October 1994, 67-74.

73. Naruse, T., Yoshida, M., Takahashi, T., and Naito, S. SIGHT – A Dedicated Computer Graphics Machine. *Computer Graphics Forum*, Vol. 6, No. 4, December 1987, 327-334.

74. Cohen, M. F., Chen, S. E., Wallace, J. R., and Greenberg, D. P. A Progressive Refinement Approach to Fast Radiosity Image Generation. *Computer Graphics*, Vol. 22, No. 4, August 1988, 75-84.

75. Baum, D. R., and Winget, J. M. Real Time Radiosity Through Parallel Processing and Hardware Acceleration. *Computer Graphics*, Vol. 24, No. 2 (*Proceedings 1990 Symposium on Interactive 3D Graphics*), March 1990, 67-75.

76. Varshney, A., and Prins, J. F. An Environment-Projection Approach to Radiosity for Mesh-Connected Computers. *Proceedings of the Third Eurographics Workshop on Rendering*, Springer-Verlag, May 1992, 271-281.

77. Kaba, J., and Peters, J. A Pyramid-based Approach to Interactive Terrain Visualization. *Proceedings 1993 Parallel Rendering Symposium*, ACM Press, October 1993, 67-70.

78. Chin, D., Passe, J., Bernard, F., Taylor, H., and Knight, S. The Princeton Engine: A Real-Time Video System Simulator. *IEEE Transactions on Consumer Electronics*, Vol. 34, No. 2, May 1988, 285-297.

79. Li, P., Curkendall, D., Duquette, W., and Henry, H. Interactive Scientific Visualization on Massively Parallel Processors. *CSCC Update*: *The Newsletter of the Concurrent Supercomputing Consortium*, Vol. 13, No. 7, Caltech CCSF, Pasadena, California, August 1994, 4-6.

80. Wright, J. R., and Hsieh, J. C. L. A Voxel-Based, Forward Projection Algorithm for Rendering Surface and Volumetric Data. *Proceedings Visualization '92*, IEEE Computer Society Press, October 1992, 340-348.

Further Reading

Dew, P. M., Earnshaw, R. A., and Heywood, T. R., eds. *Parallel Processing for Computer Vision and Display*. Addison-Wesley, 1989.

Green, S. *Parallel Processing for Computer Graphics*. MIT Press, September 1991.

Hu, M.-C., and Foley, J. D. Parallel Processing Approaches to Hidden-Surface Removal in Image Space. *Computers and Graphics*, Vol. 9, No. 3, 1985, 303-317.

Kaplan, M., and Greenberg, D. P. Parallel Processing Techniques for Hidden Surface Removal. *Computer Graphics*, Vol. 13, No. 2, August 1979, 300-307.

Kaufman, A., Bakalash, R., Cohen, D., and Yagel, R. A Survey of Architectures for Volume Rendering. *IEEE Engineering in Medicine and Biology*, Vol. 9, No. 4, December 1990, 18-23.

Lerner, E. J. Fast Graphics Use Parallel Techniques. *IEEE Spectrum*, Vol. 18, No. 3, March 1981, 34-38.

Molnar, S., and Fuchs, H. Advanced Raster Graphics Architecture. In *Computer Graphics: Principles and Practice*, 2nd ed., J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, Addison-Wesley, 1990, 855-922.

Singh, J. P., Gupta, A., and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. *Computer*, Vol. 27, No. 7, July 1994, 45-55.

Theoharis, T. *Algorithms for Parallel Polygon Rendering*. Lecture Notes in Computer Science, Vol. 373, G. Goos and J. Hartmanis, eds., Springer-Verlag, 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
April 1995

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
PARALLEL RENDERING

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Thomas W. Crockett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 95-31

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-195080
ICASE Report No. 95-31

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report
To appear in The Encyclopedia of Computer Science and Technology

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This article provides a broad introduction to the subject of parallel rendering, encompassing both hardware and software systems. The focus is on the underlying concepts and the issues which arise in the design of parallel rendering algorithms and systems. We examine the different types of parallelism and how they can be applied in rendering applications. Concepts from parallel computing, such as data decomposition, task granularity, scalability, and load balancing, are considered in relation to the rendering problem. We also explore concepts from computer graphics, such as coherence and projection, which have a significant impact on the structure of parallel rendering algorithms. Our survey covers a number of practical considerations as well, including the choice of architectural platform, communication and memory requirements, and the problem of image assembly and display. We illustrate the discussion with numerous examples from the parallel rendering literature, representing most of the principal rendering methods currently used in computer graphics.

14. SUBJECT TERMS
parallel rendering; computer graphics; survey

15. NUMBER OF PAGES
52

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		