An Introduction to Parallel Rendering

Thomas W. Crockett [1]

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton, VA, USA

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. 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 an overview of the parallel rendering field, encompassing both hardware and software systems. The focus is on the underlying concepts and the issues which arise in the design of parallel renderers. We examine the different types of parallelism and how they can be applied in rendering applications. Concepts from parallel computing, such as data decomposition and load balancing, are considered in relation to the rendering problem. Our survey explores 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.

Keywords: parallel rendering, computer graphics, survey

1 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-

[1] 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.

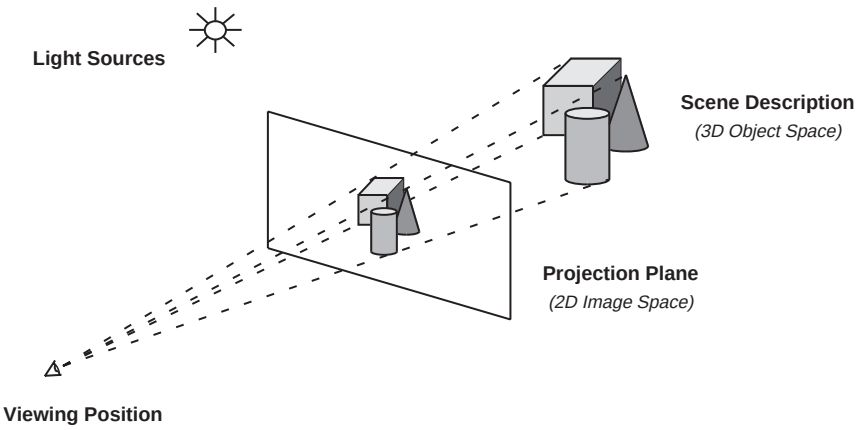

Fig. 1. The generic rendering problem. A three-dimensional scene is projected onto an image plane, taking into account the viewing parameters and light sources.

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 [2].

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*.

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 begin our examination of parallel rendering in section 2 by considering the types of parallelism which are available in computer graphics applications. Section 3 then introduces a number of concepts which are central to an understanding of parallel rendering algorithms. Building on this base, section 4 considers design and implementation issues for parallel renderers, with an emphasis on architectural considerations and application requirements. Sections 2, 3, and 4 are illustrated throughout with examples from the parallel rendering literature. Section 5 completes our survey with an examination of several parallel rendering applications.

[2] For a comprehensive reference to the discipline of computer graphics, see [23].

2 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*. These basic types can also be combined into hybrid systems which exploit multiple forms of parallelism. Each of these options is discussed below.

2.1 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 [10][11], 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.

2.2 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).

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

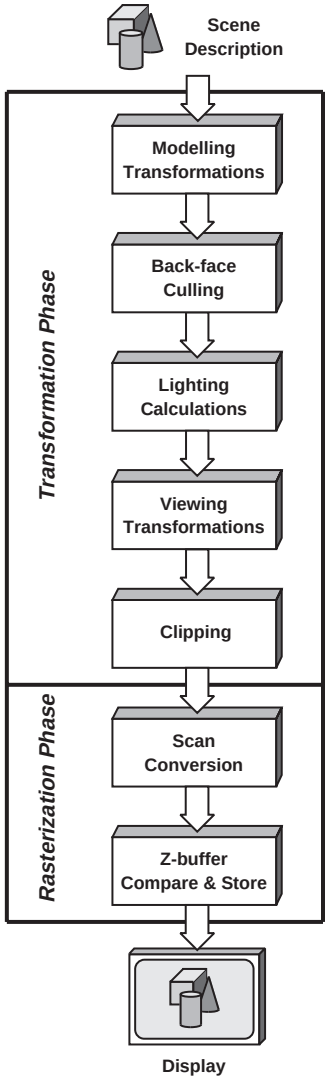

Fig. 2. A typical polygon rendering pipeline. The number of function units and their order varies depending on details of the implementation.

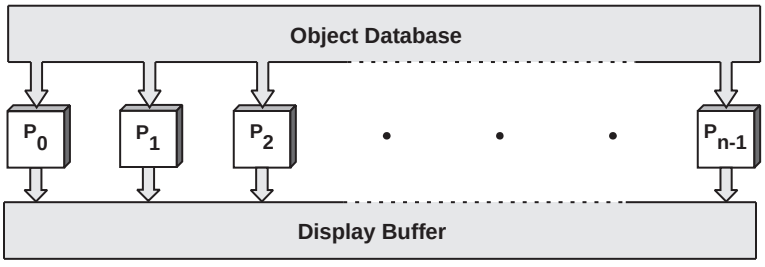

Fig. 3. A data-parallel rendering system. Multiple data items are processed simultaneously and the results are merged to create the final image.

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*.

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.

2.3 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.

2.4 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). This strategy was adopted for Silicon Graphics' RealityEngine [1], which combines 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.

3 Algorithmic concepts

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

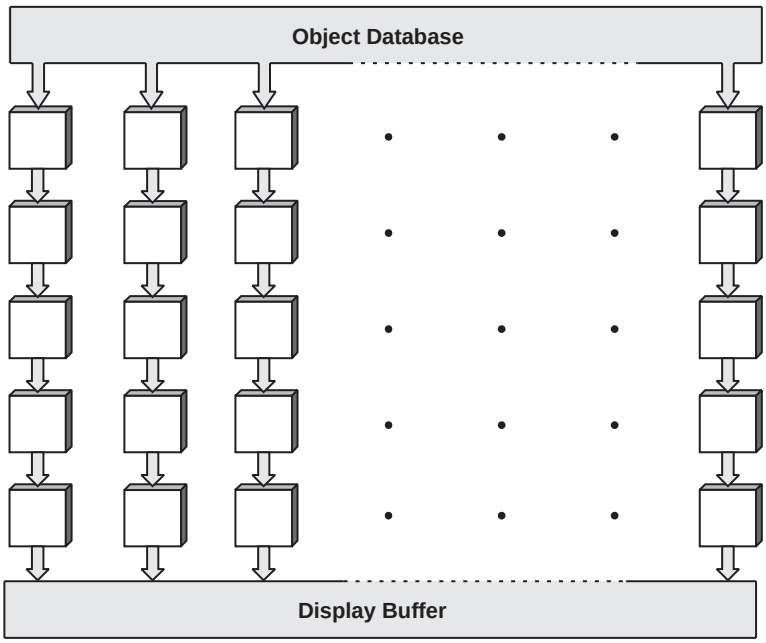

Fig. 4. A hybrid rendering architecture. Functional parallelism and data parallelism are both exploited to achieve higher performance.

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 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.

In other cases the design of effective parallel rendering algorithms can be a challenging task. 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, load balancing) are common to most parallel algorithms, while others (coherence, object-space to image-space mapping) are specific to the rendering problem.

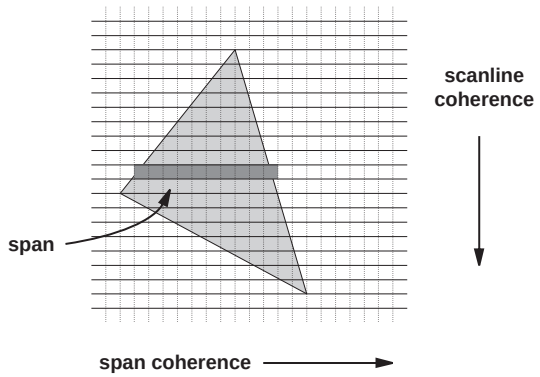

Fig. 5. Spatial coherence in image space. Pixel values tend to be similar from one scanline to the next, and from pixel to pixel within spans. Sequential rendering algorithms exploit this property to reduce computation costs during scan conversion.

3.1 Coherence

In computer graphics, *coherence* refers to the tendency for features which are nearby in space or time to have similar properties [64]. 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 (for predicting workloads) and in image display (by reducing the number of pixels to be transmitted).

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 *span 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 [50][69], and hence is an important consideration in the design of parallel polygon renderers [21].

A related notion in ray-casting renderers 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 [2][48].

3.2 Object-space to image-space mapping

The key to high performance on many parallel architectures is successful exploitation of data locality to minimize remote memory references. In parallel rendering algorithms, we also want to partition the image and object data among the available processors to achieve scalable performance and to accommodate increases in scene complexity and image resolution. Unfortunately, these two goals are in conflict.

To understand the problem, 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 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 the processors. Because of the complexity and dynamic nature of the mapping function, the communication pattern is essentially arbitrary, with each processor sending data to, and receiving data from, a large number of other processors.

Managing this communication is one of the central issues for parallel renderers, particularly on distributed-memory architectures. To better understand this problem, Molnar *et al.* [50] 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. Additional analysis of the three strategies can be found in [15], and a detailed study of the rarely-used sort-first method is presented in [51]. Examples of sort-middle renderers include [20] and [21], while the sort-last strategy is used in [14][16][28][40].

3.3 Task and data decomposition

Data-parallel rendering algorithms may be distinguished based on the way in which they decompose the problem into individual workloads or tasks. There are two main strategies. 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 be combined to form a final image. In contrast, *image-parallel* algorithms reverse this mapping. Tasks are formed by partitioning the image space, and each task renders those

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, the integration step needed to combine pixel values into a finished image 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 [20][21][53][62]. 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.

3.4 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[3] 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-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 respon-

[3] Although the causes are different, similar imbalances arise in other rendering methods as well.

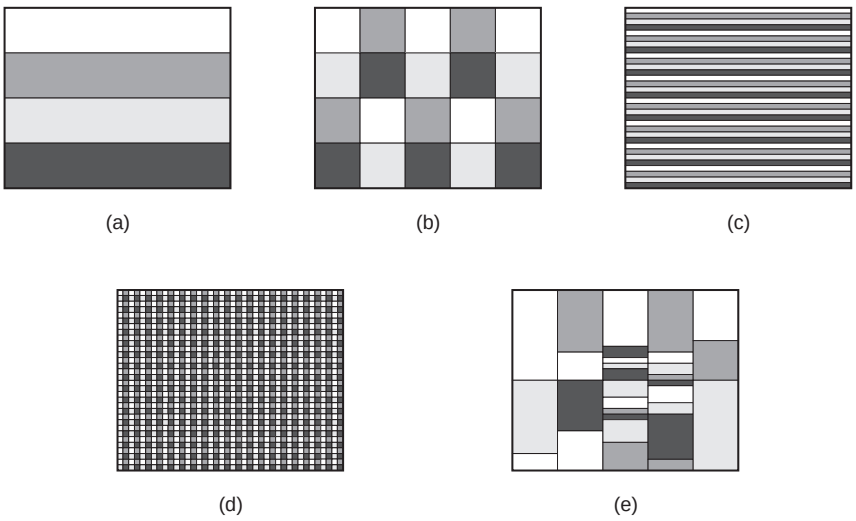

Fig. 6. Image partitioning strategies. Shading indicates the assignment of image regions to four processors. *(a)* Blocks of contiguous scanlines; *(b)* square regions; *(c)* interleaved scanlines; *(d)* pixel interleaving in two dimensions; *(e)* adaptive partitioning (loosely based on [70]).

sible 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.

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 in Section 3.1. Analytical and experimental results [68][69] 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.

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 independent tasks which are 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 finish while other processors are still busy with shorter

tasks. 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 [70]. 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.

4 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.

4.1 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. The 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 and Paragon systems [21][40], Thinking Machines' CM-200 and CM-5 [28][55], and Cray's T3D [16] achieved performance levels which equalled or exceeded those of contemporary high-end graphics workstations.

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 [16][17].

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 [26][46], radiosity renderers [59][60][72], and Pixar's photorealistic NetRenderMan system [57].

4.2 Shared vs. distributed memory

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 with large numbers 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 access to off-processor data may take several orders of magnitude longer to complete than local accesses.

A number of recent systems combine elements of both architectures, using physically distributed memories which are mapped into a global shared address space [13][36][41]. 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.

From an algorithmic standpoint, shared-memory systems provide relatively efficient access to a global address space, which in turn reduces the need to pre-partition major data structures, simplifies processor coordination, and maximizes the range of practical algorithms. To avoid resource contention, good shared-memory algorithms must decompose the problem into tasks which eliminate 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 still must strive for a high degree of locality in their memory reference patterns.

Distributed-memory systems offer improved architectural scalability, but generally incur higher costs for remote memory references. For this class of machines, managing communication is a primary consideration. Parallel renderers must pay special attention to this issue due to the large volume of intermediate data which must be re-mapped from object space to image space. In the absence of specialized 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.

4.3 SIMD vs. MIMD

Because MIMD architectures allow processors to respond to local differences in workload, they would seem to be a good match for the highly variable operation counts and data access patterns which characterize the rendering process (see Section 3.4). 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, encompassing all

of the major rendering methods.

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 [29][33][45][55]. 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.

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

4.4 Communication

For renderers which exploit both image and object parallelism, a high volume of interprocessor communication is inherent in the process (see Section 3.2). 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 [15][20][32][50][53]. 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.

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 [20][21]. 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 [21][40]. With this approach, 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 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.

While helpful in reducing latency, large message buffers can contribute to contention delays when network bandwidth is insufficient [20]. 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.

4.5 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 techniques by partitioning data structures across processors, or inhibiting scalability 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 [17]. In this case, data scalability may be a more important consideration than absolute performance.

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.

4.6 Image 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.

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 [1][24][58]. Highly parallel, multi-ported 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 [25], which uses a 640 MB/s token ring network to interconnect system components, including the pixel renderers and frame buffer. The PixelFlow system [49] 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.

With general-purpose parallel computers, sustaining high frame rates is problematic, since these systems often 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 device. 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 internal 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 [18][19]. In the absence of multi-ported 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 [30] 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.

To avoid the bottlenecks associated with serial I/O interfaces, some general-purpose architectures incorporate multi-ported frame buffers which attach either directly or indirectly to the system's internal communication network [4][67]. Pixels or image segments must then be routed to the appropriate frame buffer ports and the inputs must be synchronized to ensure coherent displays.

5 Examples of Parallel Rendering Systems

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 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 at the end of this article for additional citations.

5.1 Polygon rendering

One of the earliest graphics architectures to exploit large-scale data parallelism was Fuchs and Poulton's classic Pixel-Planes system [24]. 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 Pixel-Planes 5 architecture [25] 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.

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 [49] 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. A 256-bit-wide pipelined interconnect supports the bandwidth-intensive image composition step.

Among commercially-available polygon renderers, Silicon Graphics' RealityEngine series [1] has enjoyed the most success, and is the renderer of choice in a host of demanding applications, including virtual reality, real-time simulation, and scientific visualization. The recently-introduced InfiniteReality system continues this tradition, boosting polygon rendering rates by a factor of five over the second-generation RealityEngine2.

5.2 Volume rendering

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 [35] 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-4 [56], uses a more flexible memory organization to support a general ray-casting model with arbitrary viewing angles, perspective projections, and trilinear interpolation of ray samples.

Knittel and Straßer [37] 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.

Recent developments in algorithms and computer architectures have combined to produce substantial performance increases for software-based volume renderers as well. One of the best examples is Lacroute's image-parallel renderer for shared-memory systems [39], which is capable of interactive frame rates on large datasets (256^3 and above) using commercially-available symmetric multiprocessors. Lacroute employs an optimized shear-warp rendering algorithm [38] which exploits both image-space and object-space coherence and incorporates demand-driven dynamic load balancing.

While the majority of volume rendering algorithms are designed for use with simple rectilinear grids, many scientific and engineering applications rely on more complex discretizations, including curvilinear, unstructured, and multi-block grids. This has prompted the development of several specialized volume renderers. For non-rectilinear and multi-block grids, Challinger developed an image-parallel shared-memory algorithm and tested it on the BBN TC-2000 [7]. While the rendering phase showed good speedups, a sorting step is needed to assign cell faces to image tiles, and this, along with load imbalances, tended to limit performance. More recently, Ma developed a distributed-memory volume renderer for unstructured grids [47], and implemented it on the Intel Paragon. He also noted performance limitations due to load imbalances. Together, these results suggest that additional work is needed to develop scalable volume rendering strategies for complex grids.

5.3 Ray-tracing

Due to its computational expense, its ability to produce realistic images, and its lack of support in commercial graphics architectures, ray-tracing was an early and frequently-addressed topic in parallel rendering. The SIGHT architecture [52] is one example of a system which was designed specifically to support 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.

On general-purpose parallel systems, the majority of the execution time in ray tracing is spent on calculating the intersections of rays with objects in the scene. When the object data can be shared or replicated over the processors, a task distribution based on an image space subdivision will generally be very efficient. When the data size is larger and has to be distributed over the processors, either static or demand-driven task assignment can be used, both of which introduce additional communication and scheduling overheads.

In the static approach the ray tasks are allocated to the processors that contain the relevant data, and rays are communicated from one processor to another as needed. In the demand-driven approach the ray tasks are delegated to processors on request, which then have to fetch the needed object data, introducing extra communication. The amount of communication can be reduced by caching object data, in effect exploiting coherence in the scene. The static approach can handle arbitrarily large models but balancing workloads among processors is very difficult, since the cost of calculating ray/object intersections and evaluating secondary rays varies depending on the type and distribution of objects within the scene. The demand-driven approach has turned out to be rather efficient even with limited cache sizes [2][27]. With larger caches even complex models can be rendered successfully [63].

A hybrid load-balancing scheme for distributed-memory MIMD architectures was developed independently by Salmon and Goldsmith [62] and Caspary and Scherson [6]. With this approach the object data is organized using a hierarchical spatial subdivision, a well-known technique employed by sequential ray-tracers to reduce the search space for intersection testing. The upper part of the hierarchy is replicated on every processor, while the lower parts (which comprise the bulk of the object data) are distributed among the processors. This results in two distinct types of tasks: one which performs intersection calculations in the upper hierarchy (ray traversal), and another which performs the same calculations for the local data (ray-object intersection). Because the upper-level hierarchy is available everywhere, any processor in the system can perform the initial intersection tests on any ray, effectively decoupling the image-space and object-space partitionings.

A similar hybrid strategy has been adopted for use in stochastic ray tracing

with explicit sampling of the diffuse reflectance [31][61]. In this method data coherence is almost completely lost, severely impacting the performance of caching schemes. However, for this application a different task distribution is needed: non-coherent ray tasks are assigned statically to provide a basic load that is adjusted by demand-driven tasks that execute the coherent ray tasks (mainly the primary rays and the shadow rays).

5.4 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. In radiosity methods, the primary expense arises in generating the geometric *form factors* which are used to compute energy transfers among patches. Hence, parallel implementations have focused on speeding up this portion of the computation.

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* [12] 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) [5][8][22][54][59][60]. 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 [54][59][60]. 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.* [5] used 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 [8] and [22].

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. The effect is minimal when only a few shooting patches are active [3], but becomes more pronounced as the number of processors increases [5]. In order to exploit massive parallelism, a different approach is needed. Varshney and Prins developed a SIMD radiosity renderer for a MasPar MP-1 with 4096 processing elements [65]. As in Çapın'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. All of the other patches in the environment are then scan-converted onto the shooting patch, and form factors are obtained by accumulating the resulting pixel values. Energy transfers are performed in parallel using the results of 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.

5.5 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 methods can be used to render terrain, most of the parallel techniques 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.* [33][34] developed data-parallel terrain rendering techniques for the Princeton Engine, a programmable SIMD system originally designed for real-time processing of digital video [9]. 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 efficiently by decomposing the necessary transformations into a sequence of shear, shear/scale, and transpose operations. Hidden surfaces are eliminated by scanning the transformed data from front-to-back, one horizontal scanline at a time. The pixels in each scanline are processed in parallel. With each pass, a horizon line is updated; only those pixels which lie above the current horizon line will be visible. 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 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 [42] partitioned the input data by horizontal slices and assigned them to processors in interleaved fashion, more recent implementations use rectangular tiles with either interleaved [44] or random [43] assignment. The random strategy 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.

While the two previous examples both exploited data parallelism, other approaches are certainly possible. Wright and Hsieh [71] 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 con-

catenated pipelines, one for voxel processing and one for pixel processing. 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. 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.

6 Summary

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. We have examined many of the general principles and algorithmic approaches which apply to computer graphics rendering on parallel architectures, and surveyed representative implementations in both hardware and software.

As our discussion illustrates, the algorithm or architecture designer is faced with a wide range of implementation strategies and a complex series of tradeoffs. A successful parallel renderer 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 fallen short. Many challenges remain, particularly in the areas of scalability, load balancing, communication, and image assembly. Finding solutions to these problems will motivate further explorations in parallel rendering as computer architectures advance into the teraflops regime.

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. Erik Jansen provided extensive editorial assistance and contributed to the section on ray-tracing. David Banks and John van Rosendale offered valuable comments on an early version of the manuscript.

References

[1] K. Akeley, RealityEngine graphics, in: *Comp. Graphics Proc.*, Ann. Conf. Series (ACM SIGGRAPH, 1993) 109-116.

[2] D. Badouel, K. Bouatouch, and T. Priol, Distributing Data and Control for Ray Tracing in Parallel, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 69-77.

[3] D. R. Baum and J. M. Winget, Real time radiosity through parallel processing and hardware acceleration, in: *Proc. 1990 Symp. on Interactive 3D Graphics*, *Comp. Graphics* **24**(2) (ACM SIGGRAPH, 1990) 67-75.

[4] R. E. Benner, Parallel graphics algorithms on a 1024-processor hypercube, in: *Proc. 4th Conf. on Hypercubes, Concurrent Computers, and Applications*, Vol. I (Monterey, CA, 1989) 133-140.

[5] T. K. Çapın, C. Aykanat, and B. Özgüç, Progressive refinement radiosity on ring-connected multicomputers, in: *Proc. 1993 Parallel Rendering Symp.* (ACM Press, 1993) 71-76.

[6] E. Caspary and I. D. Scherson, A self-balanced parallel ray-tracing algorithm, in: P. M. Dew, R. A. Earnshaw, and T. R. Heywood, eds., *Parallel Processing for Computer Vision and Display* (Addison-Wesley, 1989) 408-419.

[7] J. Challinger, Scalable parallel volume raycasting for nonrectilinear computational grids, in: *Proc. 1993 Parallel Rendering Symp.* (ACM Press, 1993) 81-88.

[8] A. G. Chalmers and D. J. Paddon, Parallel processing of progressive refinement radiosity methods, in: *Photorealistic Rendering in Computer Graphics: Proc. 2nd Eurographics Workshop on Rendering* (Springer-Verlag, 1991) 149-159.

[9] D. Chin, J. Passe, F. Bernard, H. Taylor, and S. Knight, The Princeton Engine: A Real-Time Video System Simulator, *IEEE Trans. Consumer Electronics* **34**(2) (1988) 285-297.

[10] J. Clark, A VLSI Geometry Processor for Graphics, *Computer* **13**(7) (1980) 59-68.

[11] J. Clark, The Geometry Engine: A VLSI Geometry System for Graphics, *Comp. Graphics* **16**(3) (1982) 127-133.

[12] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, A Progressive Refinement Approach to Fast Radiosity Image Generation, *Comp. Graphics* **22**(4) (1988) 75-84.

[13] *Convex Exemplar System Overview* (Convex Computer Corporation, Richardson, TX, 1994).

[14] M. Cox and P. Hanrahan, A Distributed Snooping Algorithm for Pixel Merging, *IEEE Parallel and Dist. Tech.* **2**(2) (1994) 30-36.

[15] M. B. Cox, Algorithms for parallel rendering, Ph.D. dissertation, Department of Computer Science, Princeton University, 1995.

[16] *Cray Animation Theater* (Cray Research, Inc., Eagan, MN, 1994).

[17] T. W. Crockett, Design considerations for parallel graphics libraries, in: *Proc. Intel Supercomputer Users Group*, Annual North America Users Conf. (San Diego, 1994) 3-14.

[18] T. W. Crockett, Parallel rendering, Rep. 95-31 (NASA CR-195080), Institute for Computer Applications in Science and Engineering, Hampton, VA, 1995.

[19] T. W. Crockett, Beyond the renderer: software architecture for parallel graphics and visualization, in: A. Chalmers and E. Jansen, eds., *Proc. 1st Eurographics Workshop on Parallel Graphics and Visualisation* (αlpha Books, Bristol, UK, 1996) 1-15.

[20] T. W. Crockett and T. Orloff, Parallel Polygon Rendering for Message-Passing Architectures, *IEEE Parallel and Dist. Tech.* **2**(2), (1994) 17-28.

[21] D. Ellsworth, A New Algorithm for Interactive Graphics on Multicomputers, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 33-40.

[22] M. Feda and W. Purgathofer, Progressive refinement radiosity on a transputer network, in: *Photorealistic Rendering in Computer Graphics: Proc. 2nd Eurographics Workshop on Rendering* (Springer-Verlag, 1991) 139-148.

[23] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Second Edition in C (Addison-Wesley, 1996).

[24] H. Fuchs and J. Poulton, Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine, *VLSI Design* **Q3** (1981) 20-28.

[25] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories, *Comp. Graphics* **23**(3) (1989) 79-88.

[26] C. Giertsen and J. Petersen, Parallel Volume Rendering on a Network of Workstations, *IEEE Comp. Graphics and Apps.* **13**(6) (1993) 16-23.

[27] S. A. Green and D. J. Paddon, A Highly Flexible Multiprocessor Solution for Ray Tracing, *Visual Computer* **6**(2) (1990) 62-73.

[28] C. D. Hansen, M. Krogh, and W. White, Massively parallel visualization: parallel rendering, in: *Proc. 7th SIAM Conf. Parallel Proc. for Sci. Comp.* (SIAM, 1995) 790-795.

[29] W. M. Hsu, Segmented ray casting for data parallel volume rendering, in: *Proc. 1993 Parallel Rendering Symp.* (ACM Press, 1993) 7-14.

[30] J. P. Hughes, HIPPI, in: *Proc. 17th Conf. Local Comp. Networks* (IEEE CS Press, 1992) 346-354.

[31] F. W. Jansen and A. Chalmers, Realism in real time?, in: *Proc. 4th Eurographics Workshop on Rendering* (1993) 1-20.

[32] D. W. Jensen and D. A. Reed, A Performance Analysis Exemplar: Parallel Ray Tracing, *Concurrency: Practice and Experience* **4**(2) (1992) 119-141.

[33] J. Kaba, J. Matey, G. Stoll, H. Taylor, and P. Hanrahan, Interactive terrain rendering and volume visualization on the Princeton Engine, in: *Proc. Visualization '92* (IEEE CS Press, 1992) 349-355.

[34] J. Kaba and J. Peters, A pyramid-based approach to interactive terrain visualization, in: *Proc. 1993 Parallel Rendering Symp.* (ACM Press, 1993) 67-70.

[35] A. Kaufman and R. Bakalash, Memory and Processing Architecture for 3D Voxel-Based Imagery, *IEEE Comp. Graphics and Apps.* **8**(6) (1988) 10-23.

[36] R. E. Kessler and J. L. Schwarzmeier, Cray T3D: a new dimension for Cray Research, in: *Digest of Papers, COMPCON Spring '93* (IEEE CS Press, 1993) 176-182.

[37] G. Knittel and W. Straßer, A compact volume rendering accelerator, in: *Proc. 1994 Symp. on Volume Visualization* (ACM SIGGRAPH, 1994) 67-74.

[38] P. Lacroute and M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, in: *Comp. Graphics Proc.*, Ann. Conf. Series (ACM SIGGRAPH, 1994) 451-458.

[39] P. Lacroute, Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization, *IEEE Trans. Visualization and Comp. Graphics* **2**(3) (1996) 218-231.

[40] T.-Y. Lee, C. S. Raghavendra, and J. N. Nicholas, Image Composition Schemes for Sort-Last Polygon Rendering on 2-D Mesh Architectures, *IEEE Trans. Visualization and Comp. Graphics* **2**(3) (1996) 202-217.

[41] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, The Stanford Dash Multiprocessor, *Computer* **25**(3) (1992) 63-79.

[42] P. P. Li and D. W. Curkendall, Parallel three dimensional perspective rendering, in: *Proc. 2nd European Workshop on Parallel Comp.* (1992) 320-331.

[43] P. Li, D. Curkendall, W. Duquette, and H. Henry, Interactive scientific visualization on massively parallel processors, in: *CSCC Update: The Newsletter of the Concurrent Supercomputing Consortium* **13**(7) (Caltech CCSF, Pasadena, CA, 1994) 4-6.

[44] P. P. Li, W. H. Duquette, and D. W. Curkendall, RIVA: A Versatile Parallel Rendering System for Interactive Scientific Visualization, *IEEE Trans. Visualization and Comp. Graphics* **2**(3) (1996) 186-201.

[45] T. T. Y. Lin and M. Slater, Stochastic Ray Tracing Using SIMD Processor Arrays, *Visual Computer* **7**(4) (1991) 187-199.

[46] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, Parallel Volume Rendering Using Binary-Swap Compositing, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 59-68.

[47] K.-L. Ma, Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures, in: *Proc. 1995 Parallel Rendering Symp.* (ACM Press, 1995) 23-30.

[48] P. Mackerras and B. Corrie, Exploiting Data Coherence to Improve Parallel Volume Rendering, *IEEE Parallel and Dist. Tech.* **2**(2) (1994) 8-16.

[49] S. Molnar, J. Eyles, and J. Poulton, PixelFlow: High-Speed Rendering Using Image Composition, *Comp. Graphics* **26**(2) (1992) 231-240.

[50] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, A Sorting Classification of Parallel Rendering, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 23-32.

[51] C. Mueller, The sort-first rendering architecture for high-performance graphics, in: *Proc. 1995 Symp. Interactive 3D Graphics* (ACM SIGGRAPH, 1995) 75-84.

[52] T. Naruse, M. Yoshida, T. Takahashi, and S. Naito, SIGHT—A Dedicated Computer Graphics Machine, *Comp. Graphics Forum* **6**(4) (1987) 327-334.

[53] U. Neumann, Communication Costs for Parallel Volume-Rendering Algorithms, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 49-58.

[54] A. Ng and M. Slater, A Multiprocessor Implementation of Radiosity, *Computer Graphics Forum* **12**(5) (1993) 329-342.

[55] F. A. Ortega, C. D. Hansen, and J. P. Ahrens, Fast data parallel polygon rendering, in: *Proc. Supercomputing '93* (IEEE CS Press, 1993) 709-718.

[56] H. Pfister and A. Kaufman, Cube-4—a scalable architecture for real-time volume rendering, in: *Proc. 1996 Symp. on Volume Visualization* (ACM SIGGRAPH, 1996) 47-54.

[57] *PhotoRealistic RenderMan Toolkit v3.5 Reference Manual* (Pixar, Richmond, CA, 1994).

[58] M. Potmesil and E. M. Hoffert, The Pixel Machine: A Parallel Image Computer, *Comp. Graphics* **23**(3) (1989) 69-78.

[59] C. Puech, F. Sillion, and C. Vedel, Improving Interaction with Radiosity-Based Lighting Simulation Programs, *Computer Graphics* **24**(2) (1990) 51-57.

[60] R. J. Recker, D. W. George, and D. P. Greenberg, Acceleration Techniques for Progressive Refinement Radiosity, *Comp. Graphics* **24**(2) (1990) 59-66.

[61] E. Reinhard and F. W. Jansen, Hybrid scheduling for efficient ray tracing of complex images, in: M. Chen, P. Townsend, and J. A. Vince, eds., *Proc. Intl. Workshop on High Performance Computing for Computer Graphics and Visualisation* (Springer-Verlag, 1996) 78-87.

[62] J. Salmon and J. Goldsmith, A hypercube ray-tracer, in: G. C. Fox, ed., *Proc. 3rd Conf. Hypercube Concurrent Comps. and Apps.*, Vol. II, Applications (ACM Press, 1988) 1194-1206.

[63] J. P. Singh, A. Gupta, and M. Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, *Computer* **27**(7) (1994) 45-55.

[64] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A Characterization of Ten Hidden-Surface Algorithms, *Comp. Surveys* **6**(1) (1974) 1-55.

[65] A. Varshney and J. F. Prins, An environment-projection approach to radiosity for mesh-connected computers, in: *Proc. 3rd Eurographics Workshop on Rendering* (Springer-Verlag, 1992) 271-281.

[66] R. J. Vetter, ATM Concepts, Architectures, and Protocols, *Comm. ACM* **38**(2) (1995) 30-38.

[67] B. Wei, G. Stoll, D. W. Clark, E. W. Felten, and K. Li, Synchronization for a multi-port frame buffer on a mesh-connected multicomputer, *Proc. 1995 Parallel Rendering Symp.* (ACM Press, 1995) 81-88.

[68] D. S. Whelan, Animac: a multiprocessor architecture for real-time computer animation, Ph.D. dissertation, California Institute of Technology, 1985.

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

[70] S. Whitman, Dynamic Load Balancing for Parallel Polygon Rendering, *IEEE Comp. Graphics and Apps.* **14**(4) (1994) 41-48.

[71] J. R. Wright and J. C. L. Hsieh, A voxel-based forward projection algorithm for rendering surface and volumetric data, in: *Proc. Visualization '92* (IEEE CS Press, 1992) 340-348.

[72] D. Zareski, B. Wade, P. Hubbard, and P. Shirley, Efficient parallel global illumination using density estimation, in: *Proc. 1995 Parallel Rendering Symp.* (ACM Press, 1995) 47-54.