# Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs

Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh
Princeton University

## Abstract

We investigate a new hybrid of sort-first and sort-last approach for parallel polygon rendering, using as a target platform a cluster of PCs. Unlike previous methods that statically partition the 3D model and/or the 2D image, our approach performs dynamic, view-dependent and coordinated partitioning of both the 3D model and the 2D image. Using a specific algorithm that follows this approach, we show that it performs better than previous approaches and scales better with both processor count and screen resolution. Overall, our algorithm is able to achieve interactive frame rates with efficiencies of 55.0% to 70.5% during simulations of a system with 64 PCs. While it does have potential disadvantages in client-side processing and in dynamic data management—which also stem from its dynamic, view-dependent nature—these problems are likely to diminish with technology trends in the future.

**Keywords:** Parallel rendering, cluster computing.

## 1 Introduction

The objective of our research is to investigate whether it is possible to construct a *fast* and *inexpensive* parallel rendering system leveraging the aggregate performance of multiple commodity graphics accelerators in PCs connected by a system area network. The motivations for this system architecture are numerous:

- **Lower-cost:** The price-to-performance ratio of commodity graphics hardware far exceeds that of custom-designed, high-end rendering systems, and some inexpensive PC graphics accelerators can already deliver performance competitive with an SGI InfiniteReality while costing orders-of-magnitude less. A system composed only of off-the-shelf commodity components produced for a mass market costs far less than traditional high-end, custom-designed rendering systems.
- **Technology tracking:** The performance of PC graphics accelerators have been improving at a rate far beyond Moore's law for the last several years, and their development cycles are very short (six months) as compared to custom-designed, high-end hardware (one year or more). Accessing hardware components (PC graphics accelerators) only through standard software APIs (OpenGL) makes it easy to replace them on a frequent basis as faster versions become available from *any* hardware vendor.
- **Modularity & flexibility:** Networked systems in which computers communicate only via network protocols, allow PCs to be added and removed from the system easily, and they can even be heterogeneous. Network protocols also allow specific rendering processors to be accessed directly by remote computers attached to the network, and the processors can be used for other computing purposes when not in use for high-performance graphics.
- **Scalable capacity:** The aggregate hardware compute, storage, and bandwidth capacity of a PC cluster grows linearly with increasing numbers of PCs. Since each PC has its own CPU, memory, and AGP bus driving a single graphics accelerator, the scalability of a cluster-based system is far better than a tightly-integrated graphics architecture where multiple graphics pipelines share a bus, memory and I/O subsystem. Also, when using a cross-bar system area network, such as Myrinet [1], the aggregate communications bandwidth of the system scales as more PCs are added to the cluster, while the point-to-point bandwidth for any pair of PCs remains constant.

The main challenge is to develop efficient parallel rendering algorithms that scale well within the processing, storage, and communication characteristics of a PC cluster. As compared to traditional, tightly-integrated parallel computers, the relevant limitations of a PC cluster are that the processors (PCs) do not have fast access to a shared virtual address space, and the bandwidths and latencies of inter-processor communication are significantly inferior. Moreover, commodity graphics accelerators usually do not allow efficient access through standard APIs to intermediate rendering data (e.g., fragments), and thus the design space of practical parallel rendering strategies is severely limited. The challenge is to develop algorithms that partition the workload evenly among PCs, minimize extra work due to parallelization, scale as more PCs are added to the system, and work efficiently within the constraints of commodity components.

Our approach is to partition the rendering computation dynamically into coarse-grained tasks requiring practical inter-process communication bandwidths using a hybrid sort-first and sort-last strategy. We take advantage of the point-to-point nature of a system area network, employing a peer-to-peer sort-last scheme to composite images for tiles constructed with a sort-first screen decomposition. The most important contribution of our work is the partitioning algorithm that we use to simultaneously decompose the 2D screen into tiles and the 3D polygonal model into groups and assign them to PCs to balance the load and minimize overheads. The key idea is that both 2D and 3D partitions are created together dynamically in a view-dependent context for every frame of an interactive visualization session. As a result, our algorithm can create tiles and groups such that the region of the screen covered by any group of 3D polygons is closely correlated with the 2D tile of pixels assigned

to the same PC. In this case, relatively little network bandwidth is required to re-distribute pixels from PCs that have rendered polygons to the PCs whose tiles they overlap.

In this paper, we describe our research efforts aimed at using a cluster of PCs to construct a high-performance polygon rendering system. We propose a hybrid sort-first and sort-last partitioning algorithm that both balances rendering load and requires practical image composition bandwidths for typical screen resolutions. This algorithm is used to drive a prototype system in which server PCs render partial images independently and then composite them with peer-to-peer pixel redistribution. We report the results of simulations aimed at investigating the scalability and feasibility of this approach and evaluating algorithmic trade-offs and performance bottlenecks within such a system.

The paper is organized as follows. The following section reviews previous work in parallel rendering, while Section 3 discusses potential partitioning strategies for our system. Section 4 gives an overview of our approach. The details of our hybrid load balancing algorithm are described in Section 5 followed by a communication overhead analysis in Section 6. Section 7 presents the results and analysis of simulations with our algorithm compared to previous sort-first and sort-last methods. Section 8 discusses limitations and possible extensions of our approach, while Section 9 contains a brief summary and conclusion.

## 2 Previous Work

Previous parallel systems for polygon rendering can be classified in many ways: hardware vs. software, shared memory vs. distributed memory, object decomposition vs. image decomposition, SIMD vs. MIMD, sort-first vs. sort-middle vs. sort-last, or functional vs. data vs. temporal partitioning. See [6, 7, 17, 32] for more on parallel rendering taxonomies.

Hardware-based systems employ custom integrated circuits with fast, dedicated interconnections to achieve high processing and communication bandwidths. Early parallel rendering hardware was developed for z-buffer scan conversion [9, 10, 24] and for geometry processing pipelining [3, 4]. Most current systems are based on a sort-middle architecture and rely upon a fast, global interconnection to distribute primitives from geometry processors to rasterizers. For instance, SGI's InfiniteReality Engine [18] uses a fast Vertex Bus to broadcast screen space vertex information from semi-custom ASIC geometry processors to ASIC rasterization processors. The main drawback of the hardware approach is that the special-purpose processors and interconnects in these systems are custom-designed and therefore very expensive, sometimes costing millions of dollars.

Software-based systems have been implemented on massively parallel architectures, such as Thinking Machines' CM-5 [23], BBN's Butterfly TC2000 [33], Intel's Touchstone Delta system [8], and SGI's Origin 2000 [14, 30]. Unfortunately, tightly-coupled, scalable parallel computers are also expensive, and since they are typically designed or configured for scientific computations and not for graphics, the price-to-performance ratio of pure software rendering is not competitive with current graphics hardware.

Relatively little work has been done on interactive polygon rendering using a cluster of networked PCs [13, 28, 29]. Traditionally, the latency and bandwidths of typical networks have not been adequate for fine-grained parallel rendering algorithms. Rather, most prior cluster-based polygon rendering systems have utilized only inter-frame parallelism [13, 25], rendering separate frames of an image sequence on separate computers in parallel. Other parallel rendering algorithms, such as ray tracing, radiosity [11, 26], and volumetric rendering [12] have been implemented with PC clusters. However, they generally have not achieved fast, interactive frame rates (i.e., thirty frames per second). We are not aware of any prior system that has achieved scalable speedups via intra-frame data parallelism while rendering polygonal models with a network of workstations.

Last year, Samanta et al. [28] described a sort-first parallel rendering system running on a cluster of PCs driving a high-resolution, multi-projector display wall. In that paper, the goal was to balance the load among a fixed number of PCs (eight) each driving a separate projector corresponding to part of a very large and high resolution seamless image. They focused on sort-first algorithms aimed at avoiding large overheads due to redistribution of big blocks of pixels between multiple PCs driving full-screen projectors. The method described in this paper is similar in concept. But, it uses a sort-last image composition scheme to achieve scalable speedups for large clusters of PCs driving a single display.

## 3 Choosing a Partitioning Strategy

The first challenge in implementing a parallel rendering system is to choose a partitioning strategy. Following the taxonomy of Molnar et al. [5, 17], we consider sort-middle, sort-first, and sort-last approaches.

In sort-middle systems, processing of graphics primitives is partitioned equally among geometry processors, while processing of pixels is partitioned among rasterization processors according to overlaps with screen-space tiles. This approach is best suited for tightly-coupled systems that use a fast, global interconnection to send primitives between geometry and rasterization processors based on overlaps with simple and static tilings, such as a regular, rectangular grid. Using a sort-middle approach would be very difficult in a cluster-of-PCs system for two reasons. First, commodity graphics accelerators do not generally provide high-performance access to the results of geometry processing through standard APIs (e.g., "feedback" mode in OpenGL). Second, network communication performance is currently too slow for every primitive to be redistributed between processors on different PCs during every frame.

In sort-first systems, screen-space is partitioned into non-overlapping 2D tiles, each of which is rendered independently by a tightly-coupled pair of geometry and rasterization processors (i.e. a PC graphics accelerator, in our case), and the subimages for all 2D tiles are composited (without depth comparisons) to form the final image. The main advantage of sort-first is that its communication requirements are relatively small. Unlike sort-middle, sort-first can utilize retained-mode scene graphs to avoid most data transfer for graphics primitives between processors, as graphics primitives can be replicated and/or sent dynamically between processors as they migrate between tiles [20]. The disadvantage is that extra work must be done to transform graphics primitives, compute an appropriate screen space partition, determine primitive-tile overlaps for each frame, and render graphics primitives redundantly if they overlap multiple tiles. Most significant among these is the extra rendering work, which can be characterized by the *overlap factor* – the ratio of the total rendering work performed over the ideal rendering work required without redundancy. Since overlap factors grow linearly with increasing numbers of processors, the scalability of sort-first systems is limited. In our experience, the efficiency of sort-first algorithms [19] drops below 50% with 16 processors [27].

In sort-last systems, each processor renders a separate image containing a portion of the graphics primitives, and then the resulting images are composited (with depth comparisons) into a single image for display. The main advantage of sort last is its scalability. Since each graphics primitive is rendered by exactly one processor, the overlap factor is always 1.0. The main disadvantage of sort-last is that it usually requires an image composition network with very high bandwidth and processing capabilities to support transmission and composition of overlapping depth images. Also, the sort-last approach usually provides no strict primitive ordering semantics,

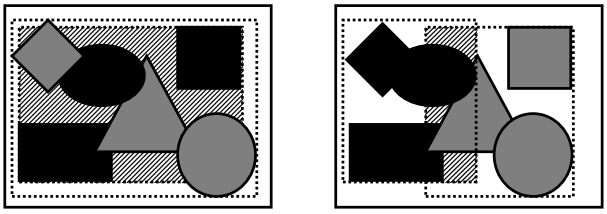
and it incurs latency as subimages must be composited before display.

To summarize, no prior parallel rendering algorithm is both scalable and practical for a PC cluster. At one extreme, sort-first systems require relatively little communication bandwidth. However, they are not scalable, as overlap factors grow linearly with increasing numbers of processors. On the other extreme, sort-last systems are scalable. But, their bandwidth requirements exceed the capabilities of current system area networks. The goal of our work is to develop a parallel rendering algorithm which strikes a practical balance between these trade-offs.

## 4 Overview of Our Approach

Our approach is to use a hybrid parallel rendering algorithm that combines features of both "sort-first" and "sort-last" strategies. We execute a view-dependent algorithm that dynamically partitions both the 2D screen into *tiles* and the 3D polygons into *groups* in order to balance the rendering load among the PCs and to minimize the bandwidths required for compositing tiles with a peer-to-peer redistribution of pixels.

The key idea is to cluster 3D polygons into groups for rendering by each server dynamically based on the overlaps of their projected bounding volumes in screen space. The motivation for this approach is best demonstrated with an example. Consider the arrangement of polygons shown in Figure 1. If polygons are assigned randomly to two groups, as shown in Figure 1(a), the area of overlap between the bounding boxes of the two groups is quite large (shown in hatch pattern). On the other hand, if the polygons are grouped according to their screen space overlaps, as shown in Figure 1(b), the area of intersection between the groups' bounding boxes is much smaller. This difference has critical implications for the bandwidth required for image compositing in a sort-last system.



(a) Random grouping. (b) Spatial grouping.

Figure 1: Polygon groupings affect image composition bandwidths.

We have implemented a view-dependent object-partition strategy motivated by examples such as this one using a sort-first screen-partition algorithm in a *client* PC controlling a parallel rendering system comprising $N$ server PCs and one display PC. Specifically, for every frame of an interactive visualization session, the system proceeds in a three phase pipeline, as shown in Figure 2.

- **Phase 1:** In the first phase, the client executes a partitioning algorithm that simultaneously decomposes the 3D polygonal model into $N$ disjoint groups, assigning each group to a different *server* PC, and partitions the pixels of the 2D screen into non-overlapping tiles, also assigning each tile to a different server PC.

- **Phase 2:** In the second phase, every server $A$ renders the group of the 3D primitives it has been assigned into its local frame buffer. It then reads back from the frame buffer into memory the color and depth values of pixels that reside within any intersection of the group's projected screen-space bounding box and a tile assigned to another server $B$, and it sends them over the system area network to server $B$. Meanwhile, after every server $B$ has completed rendering its own group of polygons, it receives pixels rendered by other servers from the network, and it composites them into its local frame buffer with depth comparisons to form a complete image for its tile. Finally, each server reads back the color values of pixels within its tile and sends them to a *display* PC.

- **Phase 3:** In the third phase, the display PC receives subimages from all servers, and composites them (without depth comparisons) to form a final complete image in its frame buffer for display.

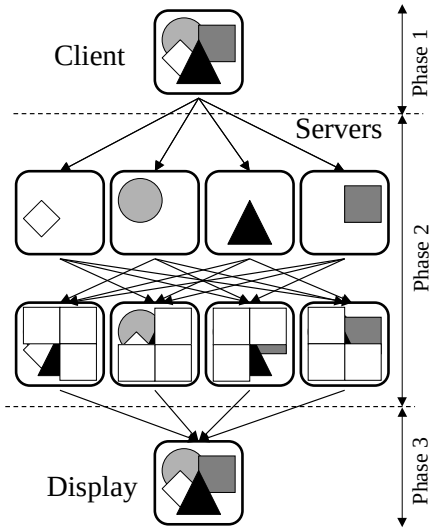



Figure 2: System architecture with client, server and display phases.

This system architecture has two important advantages. First, it uses peer-to-peer communication to redistribute pixels between servers, a strategy that is very well-suited for point-to-point system area networks, which are available for clusters of PCs. The result is lower latencies and higher network utilizations than other more traditional sort-last methods. Second, the aggregate pixel bandwidth received by the display machine is minimal, as every pixel is sent to the display exactly once for each image in the third phase of the pipeline. This feature is important for our system because the bandwidth of network communication into any one PC is limited ($\sim$100 MB/s), and sort-last methods in which the display processor receives pixels with depth values for the whole screen would be impractical for an interactive system.

## 5 Hybrid Partitioning Algorithm

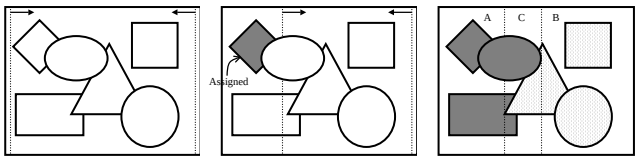
In order to make our system architecture viable, we must develop a partitioning algorithm that constructs tiles of pixels and groups of objects and assigns them to servers for each frame. An effective algorithm should achieve three goals:

- It should balance the work load among the servers.
- It should minimize the overheads due to pixel redistribution.
- It should run fast enough such that the client is not the limiting pipeline stage.

Prior work on dynamic screen-space partitioning has focused on constructing partitions with balanced rendering loads. For instance, Whelan developed a median-cut method in which the screen was partitioned recursively into a number of tiles exactly equal to the number of processors [31]. For each recursive step, a tile was split by a line perpendicular to its longest axis so that the centroids of its overlapping graphics primitives were partitioned most equally. In later work, Mueller developed a mesh-based median-cut method in which primitives were first tallied up according to how their bounding boxes overlapped a fine mesh, and an estimated cost was calculated for each overlapped mesh cell [19]. Then, using this data as a hint, screen space tiles were recursively split along their longest dimensions until the number of regions equaled the number of processors. Whitman used a dynamic scheduling method in which he started with a set of initial tiles and "stole" part of another processor's work dynamically when no initial tiles remain [33]. The stealing is achieved by splitting the remaining tile region on the maximally loaded processor into two vertical strips having the same number of scanlines.

In contrast to this previous work, we not only try to balance the rendering load across the partition, but we also aim to minimize the screen space overlaps of the subimages rendered by different servers. Our method is a recursive binary partition using a two-line sweep.

Each single partition step proceeds as follows: We split the current region along the longest axis. Without loss of generality, let us assume the width is larger than the height of the region. Hence, we shall sweep vertical lines. We begin with two vertical lines, one at each side of the region to be partitioned. The vertical line on the left will always move to the right, and the one on the right will always move left (as shown in Figure 3(a)). The objects assigned as we move the left line will belong to the left group. Similarly, the right line will assign objects for the right group. At every step, the algorithm moves the line associated with the group with the least work in an attempt to maintain a good load balance. The line is moved until it passes a currently unassigned object. In our figure, we move the line on the left until it completely passes the leftmost object on the screen (which is unassigned since the algorithm has just begun executing). This object is assigned to the left group (see Figure 3(b)). This process is repeated until all objects have been assigned, in which case the sweep lines may have crossed and there is a narrow swath (labeled C in Figure 3(c)) where the bounding boxes of the left and right groups overlap. This swath is a region requiring composition of the two images rendered for the objects assigned to the left and right groups. On average, the swath's width is the mean width of a bounding box. The screen partition continues recursively until exactly $N$ tiles and groups are formed, and one is assigned to each of the $N$ servers.




(a) Beginning (b) 1 object assigned (c) All objects assigned

Figure 3: Example execution of the hybrid partition algorithm.

This algorithm has a running time complexity of O($M \log M + MN$), where $M$ is the number of bounding boxes in the scene and $N$ is the number of servers. The $M \log M$ term arises from the stage the $M$ bounding boxes are sorted according to their screen position, and the $MN$ factor comes from the fact that $N - 1$ cuts are made, while O($M$) bounding boxes are considered for each cut.

# 6 Analysis of Communication Overheads

We analyze the average case communication requirements with our algorithm by making the following simplifying assumptions:

- All objects are evenly distributed.
- All objects are squares of equal size (in screen space). The edge of the square is of length $B$.
- There are $P$ total pixels on screen and $N$ servers.
- Each server is assigned a square tile with area $P/N$ and dimension $\sqrt{P}/\sqrt{N}$.

Based on these assumptions, each tile will have a region of width $B/2$ around its perimeter which its server will need to read back from its frame buffer and send to other servers for compositing (see Figure 4). Similarly, every server will receive pixels within approximately the same sized regions from other servers.

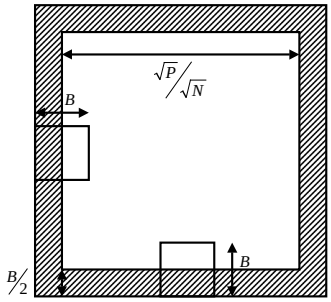



Figure 4: Pixel areas to be sent out.

The number of pixels composited by each server is the sum of the four corner squares and the four rectangles along the perimeter of the box. The four corner squares each have area :

$$\frac{B}{2} \cdot \frac{B}{2} \tag{1}$$

The area of each rectangle on the four sides is :

$$\frac{B}{2} \frac{\sqrt{P}}{\sqrt{N}} \tag{2}$$

Summing the area of these regions, we have:

$$4\frac{B}{2} \frac{\sqrt{P}}{\sqrt{N}} + 4\frac{B}{2} \frac{B}{2} \tag{3}$$

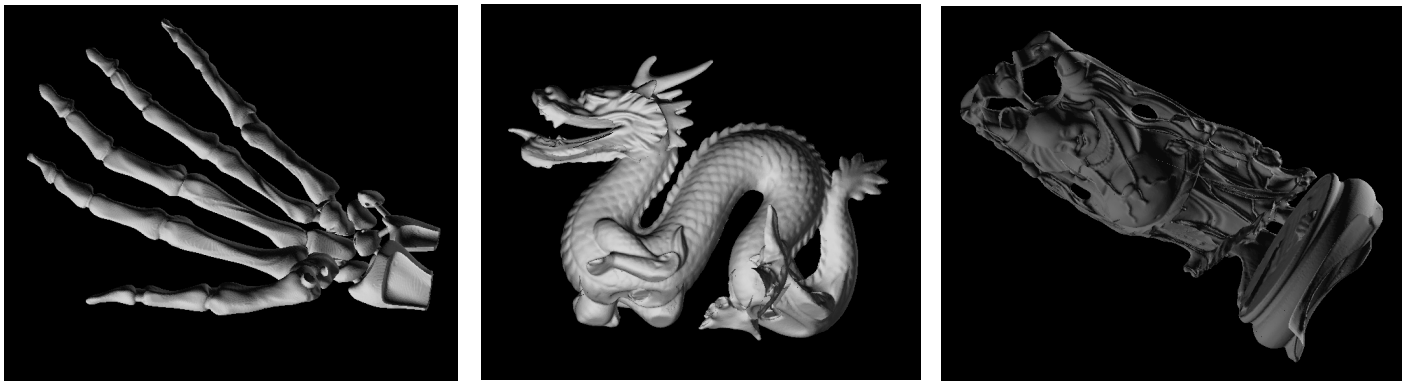
or,

$$2B \frac{\sqrt{P}}{\sqrt{N}} + B^2 \tag{4}$$

which represents the average number of pixels transferred one way from a single server for each frame.

This pixel redistribution overhead compares favorably with previous sort-last methods. For instance, several volume rendering systems employ static partitions which result in an approximate depth complexity of $\sqrt[3]{N}$, since the volume is divided into $\sqrt[3]{N}$ blocks along each of its three axes [15, 21]. If we assume each server is responsible for a tile covering $P/N$ pixels, the total number of pixels composited by each server is:

$$\frac{P}{\sqrt[3]{N^2}} \tag{5}$$



(a) Hand
(643 objects, 654,666 polygons)

(b) Dragon
(1,257 objects, 871,414 polygons)

(c) Buddha
(1,476 objects, 1,087,716 polygons)

Figure 5: Test models.

We can now compare the overheads in the two methods with respect to $N$ and $P$. Increasing $N$ reduces the size of the compositing areas by a factor of $\sqrt[3]{N^2}$ with the previous algorithms, whereas it affects our composite regions by a factor of $\sqrt{N}$. This factor favors previous algorithms.

However, increasing $P$ causes the overheads of image compositing to grow linearly using previous algorithms, while they only increase with $\sqrt{P}$ using our method. Since $P$ is generally quite large (e.g., $1280 \times 960$), this is usually the dominant factor in practice, and thus our algorithm has significantly lower compositing overheads than previous schemes.

# 7 Results

We have implemented several parallel rendering algorithms in C++ under Windows NT and incorporated them into a cluster-based parallel rendering system for interactive visualization of 3D polygonal models in a VRML browser. Our prototype system is comprised of a 550 Mhz Pentium-III client PC, eight 500 Mhz Pentium-III server PCs with 256 MB RAM and nVidia GeForce 256 graphics accelerators, and a 500 MHz Pentium-III display PC with an Intergraph Wildcat graphics accelerator. All ten PCs are connected by a Myrinet system area network and use VMMC-2 communication firmware and software [2]. This implementation allows us to measure various architectural and system parameters for studies of feasibility and scalability.

In this section, we report data collected during a series of simulations of our parallel rendering system. The goals of these simulations are to investigate the algorithmic trade-offs of different partitioning strategies, to identify potential performance issues in our prototype system, to analyze the scalability of our hybrid algorithm, and to assess the feasibility of constructing a parallel rendering system with a cluster of PCs. Unless otherwise specified, the parameters of every simulation were set as follows (these "default" values reflect the measured parameters of our prototype system):

- Number of servers (N) = *64 PCs*
- Number of pixels (P) = *1280x960 pixels*
- Network i/o latency = *20 microseconds*
- Network bandwidth = *100 MB/second*
- Server polygon throughput = *750K polygons/second*
- Server pixel fill throughput = *300 Mpixels/second*
- Server color buffer i/o latency = *50 microseconds*
- Server color buffer read throughput = *20 Mpixels/second*
- Server color buffer write throughput = *10 Mpixels/second*
- Server Z buffer i/o latency = *50 microseconds*
- Server Z buffer read throughput = *20 Mpixels/second*
- Server Z buffer write throughput = *10 Mpixels/second*
- Display color buffer write throughput = *20 Mpixels/second*

During every simulation, we logged statistics generated by the partitioning algorithms running on a 550 MHz Pentium-III PC while rendering a repeatable sequence of frames in an interactive visualization program. Every test was run for three 3D models (shown in Figure 5), each of which was represented as a scene graph in which multiple polygons were grouped at the leaf nodes and treated as atomic objects by the partitioning algorithms (the numbers of polygons and objects are listed under the image of each test model in Figure 5). The objects were formed by clustering polygons whose centroids lie within the same cell of an octree expanded to a user-specified depth (usually 3-5 levels in our cases). In all three models, the input polygons were rather small, as is typical for a high-performance rendering system, and thus the object bounding boxes closely resembled boxes of an octree, and rendering was always "geometry-bound." It was assumed that the 3D model was static and was replicated in the memory of every server, a current limitation of our system that will be discussed in Section 8. For each model, the camera traveled along a preset path of 50 frames that rotated at a fixed distance around the model, looking at it from a random variety of viewing directions while the model approximately covered the full height of the screen.

Our simulation study aims to answer the following performance questions about the proposed hybrid approach:

- Is it feasible for moderately-sized PC clusters?
- Does it scale with increasing numbers of processors?
- How does it compare with a sort-first algorithm?
- How does it compare with a sort-last algorithm?
- What is the impact of increasing display resolution?
- What is the impact of varying object granularity?

## 7.1 Feasibility of Proposed Rendering System

Our first study investigates whether it is practical to build a parallel rendering system with a cluster of PCs using our hybrid partitioning algorithm. We define success in this case to be interactive frame rates (e.g., 30 frames per second) and attractive efficiencies (e.g., $\geq$ 50%) for moderately sized clusters (e.g., 8-64 PCs).

For this investigation, we executed a series of simulation tests with our hybrid algorithm using increasing numbers of server PCs. The timing results for client, server, and display PCs appear for all three test models in the top one-third of Table 1 (all times are in milliseconds). Note that the average time taken (per frame) by a uniprocessor system with the same hardware as the server PCs for the three models is as follows : Hand - 872.9 ms, Dragon - 1161.9 ms and Buddha - 1450.3 ms. Bar charts of the total client, server, and display phases are shown for the Buddha model in Figure 6. During analysis of these results, note that the client, servers, and display execute concurrently in a pipeline, so the slowest of these three stages determines the effective frame time.

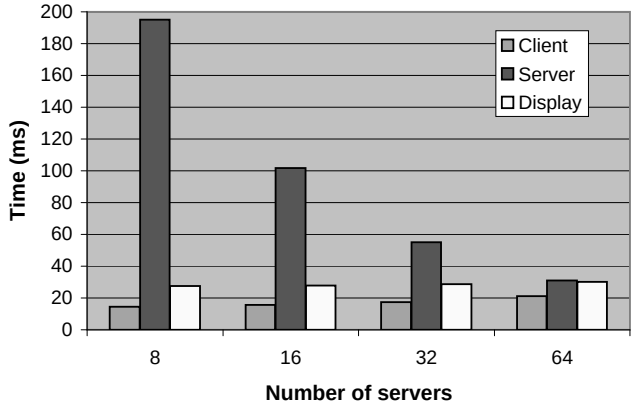



Figure 6: Client, server, and display times for Buddha model.

Examining the bars in Figure 6, we see that the server is usually the limiting factor in these tests (the dark bar in the middle of each triple). The client (gray) was never limiting, and the display (white) was the bottleneck in only two tests (when 64 servers were used to render the simpler 3D models). In those cases, the servers were cumulatively able to deliver images to the display at approximately 40 frames per second. But, due to the limited I/O bandwidth of the display PC, it could only receive around 30 frames per second worth of data (100 MB/second / 4 bytes/pixel * 1280 $\times$ 960 pixels * 0.75 screen coverage). The net result is an interactive frame rate (30 frames per second), but with slight under-utilization of the server PCs. This result points out the need for faster digital display devices or for faster I/O bandwidths in commodity display PCs.

Examining the rightmost column of Table 1, we see that the speedups of the hybrid algorithm remains high for up to at least 64 processors. The efficiency of the system ranged between 55.0% and 70.5% for 64 processors for all three test models.

From these results, we conclude that our hybrid algorithm can provide a practical, low-cost solution for high-performance rendering of static, replicated 3D polygonal models on moderately sized clusters of PCs.

## 7.2 Scalability with Increasing Processors

Our second study investigates the scalability of the hybrid partitioning algorithm as the number of servers ($N$) increases. System speedups for up to 64 servers are shown in Figure 9. There are two concerns to be addressed in this study - scalability of the client and scalability of the servers - since the least scalable pipeline stage dictates overall system scalability.

First, consider client scalability. As stated in Section 5, the hybrid partitioning algorithm grows linearly with $N$. This trend can be seen clearly in the dark black line in Figure 7, which shows client processing times for the hybrid algorithm as a function of the number of servers measured on a 550 MHz Pentium-III PC during tests with the Buddha model. Extrapolating this curve indicates that the client will be able to achieve 30 updates per second for at least up to 150 servers or so. This result is encouraging, as such large clusters are very rare today, and client PC processor speeds are growing more rapidly than cluster sizes.

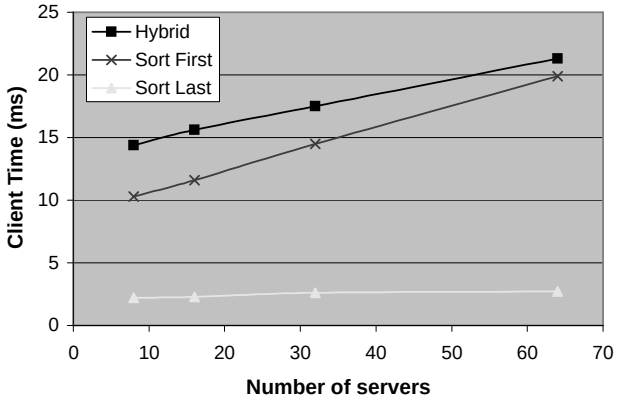



Figure 7: Client times for sort-first, sort-last, and the hybrid algorithms for increasing numbers of servers.

Second, consider server scalability. As stated in Section 6, the pixel redistribution overheads of each server scale with $\frac{1}{\sqrt{N}}$. This result can be seen in Figure 8, which shows breakdowns of server processing times. In the case of the hybrid algorithm (the middle set of bars), the overheads are due primarily to pixel reads and pixel writes during server-to-server pixel redistribution. Although the reduction in overheads is not linear, the simulated speedups of our system are quite good. For 64 processors, the speedups of the hybrid approach are 36.3, 43.7 and 46.5 for Hand, Dragon and Buddha, respectively, corresponding to effective frame times of 32.8 ms, 34.6 ms, and 31.1 ms, respectively (see Figure 9). These speedups compare favorably to previous parallel rendering systems whose frame times are significantly longer.

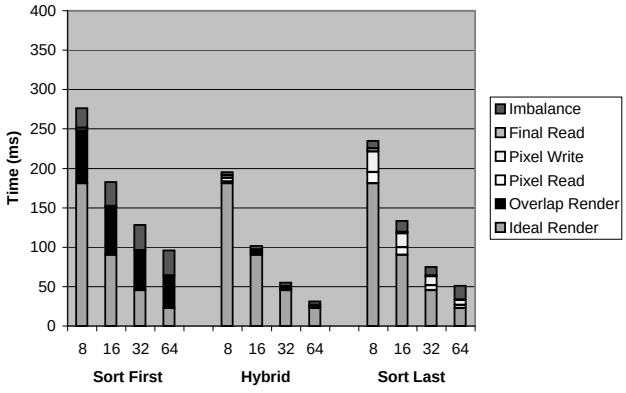
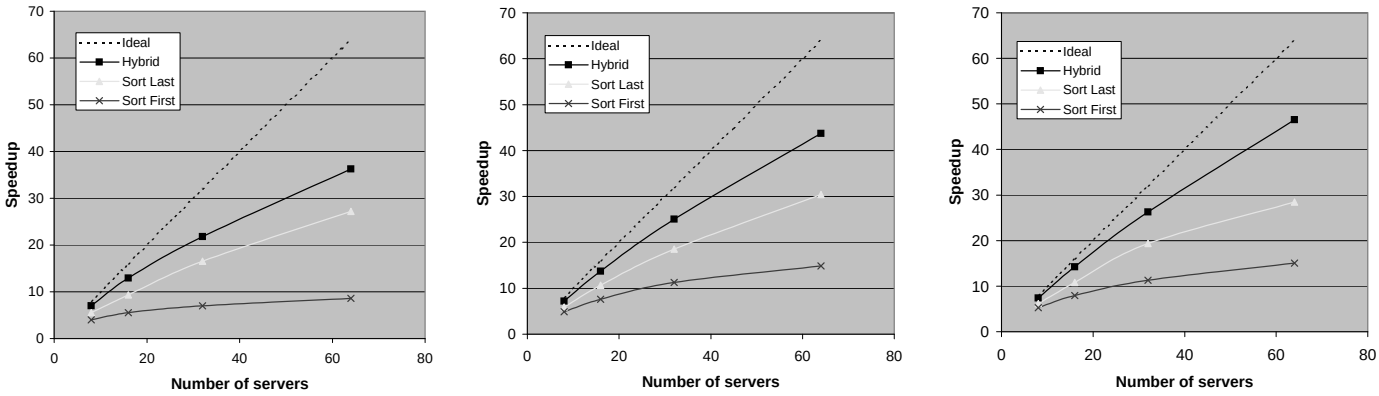



Figure 8: Server times for sort-first, sort-last, and the hybrid algorithms for screen resolution 1280x960. The height of each bar represents the time required for processing in the server.

As an addendum, we note that bandwidth requirements of the display PC are not affected by increasing the number of servers. Exactly one screen-full of pixels arrives at the display PC for each image, no matter how finely the image is split into tiles. Latencies of pixel I/O operations do impact display scalability, but they are significant only for very large numbers of servers.

| Parrallel Algorithm | Test Model | N | Client | | | Server | | | | | | | Display | Speedup Factor |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Obj Xform | Compute Partition | Total | Ideal Render | Overlap Render | Pixel Read | Pixel Write | Final Read | Wait for Imbalance | Total | Final Write | |
| Hybrid | Hand | 8 | 2.1 | 5.6 | 7.7 | 108.8 | - | 2.6 | 4.9 | 3.7 | 4.9 | 124.9 | 29.7 | 7.0 |
| | | 16 | 2.1 | 6.2 | 8.3 | 54.4 | - | 1.9 | 3.4 | 1.9 | 5.8 | 67.4 | 30.1 | 12.9 |
| | | 32 | 2.1 | 7.2 | 9.3 | 27.2 | - | 1.8 | 3.1 | 1.0 | 6.8 | 39.9 | 30.8 | 21.8 |
| | | 64 | 2.1 | 9.6 | 11.7 | 13.6 | - | 1.5 | 2.3 | 0.5 | 6.1 | 24.0 | 32.8 | 36.3 |
| | Dragon | 8 | 4.1 | 10.6 | 14.7 | 144.9 | - | 2.5 | 4.7 | 4.0 | 4.8 | 160.9 | 32.0 | 7.2 |
| | | 16 | 4.1 | 11.7 | 15.8 | 72.4 | - | 2.1 | 3.6 | 2.0 | 4.4 | 84.5 | 32.3 | 13.7 |
| | | 32 | 4.1 | 13.1 | 17.2 | 36.2 | - | 1.8 | 3.1 | 1.0 | 4.2 | 46.3 | 33.0 | 25.0 |
| | | 64 | 4.1 | 16.5 | 20.6 | 18.1 | - | 1.5 | 2.4 | 0.5 | 4.0 | 26.5 | 34.6 | 43.7 |
| | Buddha | 8 | 4.8 | 9.6 | 14.4 | 180.8 | - | 2.4 | 4.4 | 3.4 | 4.0 | 195.0 | 27.6 | 7.4 |
| | | 16 | 4.8 | 10.8 | 15.6 | 90.4 | - | 1.9 | 3.3 | 1.7 | 4.3 | 101.6 | 27.9 | 14.2 |
| | | 32 | 4.8 | 12.7 | 17.5 | 45.2 | - | 1.7 | 2.9 | 0.9 | 4.3 | 55.0 | 28.6 | 26.3 |
| | | 64 | 4.8 | 16.5 | 21.3 | 22.6 | - | 1.4 | 2.2 | 0.5 | 4.4 | 31.1 | 30.2 | 46.5 |
| Sort-First | Hand | 8 | 2.1 | 3.2 | 5.3 | 108.8 | 66.6 | - | - | 2.7 | 37.9 | 216.0 | 30.6 | 4.0 |
| | | 16 | 2.1 | 4.0 | 6.1 | 54.4 | 59.5 | - | - | 1.3 | 42.9 | 158.1 | 31.0 | 5.5 |
| | | 32 | 2.1 | 5.4 | 7.5 | 27.2 | 50.1 | - | - | 0.6 | 46.1 | 124.0 | 31.8 | 7.0 |
| | | 64 | 2.1 | 8.4 | 10.5 | 13.6 | 42.1 | - | - | 0.3 | 45.0 | 101.0 | 33.4 | 8.0 |
| | Dragon | 8 | 4.1 | 4.8 | 8.9 | 144.9 | 60.7 | - | - | 3.4 | 28.2 | 237.2 | 32.9 | 4.9 |
| | | 16 | 4.1 | 6.1 | 10.2 | 72.4 | 49.7 | - | - | 1.6 | 29.0 | 152.7 | 33.3 | 7.6 |
| | | 32 | 4.1 | 8.6 | 12.7 | 36.2 | 39.4 | - | - | 0.8 | 26.2 | 102.6 | 34.1 | 11.3 |
| | | 64 | 4.1 | 13.5 | 17.6 | 18.1 | 32.5 | - | - | 0.4 | 27.0 | 78.0 | 35.6 | 14.9 |
| | Buddha | 8 | 4.8 | 5.5 | 10.3 | 180.8 | 67.2 | - | - | 3.1 | 25.0 | 276.1 | 28.4 | 5.2 |
| | | 16 | 4.8 | 6.8 | 11.6 | 90.4 | 60.0 | - | - | 1.5 | 30.6 | 182.5 | 28.8 | 7.9 |
| | | 32 | 4.8 | 9.7 | 14.5 | 45.2 | 50.2 | - | - | 0.7 | 32.2 | 128.3 | 29.6 | 11.3 |
| | | 64 | 4.8 | 15.1 | 19.9 | 22.6 | 41.0 | - | - | 0.4 | 32.0 | 96.0 | 31.2 | 15.1 |
| Sort-Last | Hand | 8 | 2.1 | 0.1 | 2.2 | 108.8 | - | 14.1 | 26.2 | 3.9 | 4.6 | 157.6 | 31.5 | 5.5 |
| | | 16 | 2.1 | 0.2 | 2.3 | 54.4 | - | 9.2 | 16.8 | 1.8 | 11.2 | 93.4 | 29.5 | 9.3 |
| | | 32 | 2.1 | 0.5 | 2.6 | 27.2 | - | 5.2 | 9.5 | 0.9 | 9.9 | 52.7 | 29.2 | 16.5 |
| | | 64 | 2.1 | 0.6 | 2.7 | 13.6 | - | 3.1 | 5.5 | 0.4 | 9.4 | 32.0 | 28.8 | 27.2 |
| | Dragon | 8 | 4.1 | 0.1 | 4.2 | 144.9 | - | 14.0 | 26.0 | 4.4 | 6.7 | 196.0 | 34.8 | 5.9 |
| | | 16 | 4.1 | 0.2 | 4.3 | 72.4 | - | 9.3 | 17.1 | 2.0 | 7.9 | 108.7 | 32.4 | 10.7 |
| | | 32 | 4.1 | 0.5 | 4.6 | 36.2 | - | 5.6 | 10.1 | 1.0 | 9.6 | 62.5 | 31.6 | 18.5 |
| | | 64 | 4.1 | 0.6 | 4.7 | 18.1 | - | 3.2 | 5.7 | 0.4 | 10.7 | 38.1 | 30.8 | 30.4 |
| | Buddha | 8 | 4.8 | 0.1 | 4.9 | 180.8 | - | 14.2 | 26.1 | 4.0 | 9.7 | 234.8 | 31.8 | 6.2 |
| | | 16 | 4.8 | 0.2 | 5.0 | 90.4 | - | 9.5 | 17.5 | 1.9 | 14.2 | 133.5 | 31.2 | 10.8 |
| | | 32 | 4.8 | 0.5 | 5.3 | 45.2 | - | 6.3 | 11.5 | 0.9 | 10.8 | 74.7 | 29.8 | 19.4 |
| | | 64 | 4.8 | 0.6 | 5.4 | 22.6 | - | 3.8 | 6.7 | 0.4 | 17.3 | 50.8 | 29.2 | 28.5 |

Table 1: Timing statistics gathered during simulations on test models with sort-first, sort-last, and the hybrid parallel rendering algorithms. All times are in milliseconds.




(a) Hand (b) Dragon (c) Buddha

Figure 9: Speedup curves with increasing numbers of processors.

### 7.3 Comparison to Sort-First Algorithms

In our third study, we compare our hybrid partitioning algorithm to a sort-first approach previously described in the literature. Although we have implemented two sort-first methods, Mueller's MAHD [19] and Samanta's KD-Split [28], we compare simulation results for only the MAHD algorithm in this paper. The comparison for KD-Split is similar.

In our implementation of the MAHD algorithm, the client PC first transforms the objects into 2D and marks a 2D grid-based data structure with an estimate of the rendering load associated with each cell in this grid. Next, the algorithm partitions this grid data structure into tiles with balanced rendering load by recursively partitioning the grid structure along the longest dimension at every step of the recursion. The algorithm terminates when the number of tiles is equal to the number of servers. Each tile is then assigned to a server PC, which renders an image containing all polygons in any object that at least partially overlaps the extent of its tile. The servers then read the resulting pixels back from their color buffers and send them to the display PC so that they can be composited into a complete image for display. The client processing times for this method grow with $MN$ for $M$ objects and $N$ tiles, just like the hybrid algorithm. The main overheads incurred by the sort-first algorithm are due to client processing as the screen is partitioned, redundant server rendering of objects overlapping multiple tiles, and server-to-server pixel redistribution.

Timing results measured during tests with both the hybrid and sort-first methods appear for all three test models in Table 1, and more detailed breakdowns of the server processing times are shown for the tests with the Buddha model in Figure 8. From these breakdowns of Figure 8, we can visually compare the overheads of different algorithms. In particular, the dark bands ("Overlap Render") in the bars for sort-first (the leftmost set of bars) represent overheads due to redundant rendering of objects overlapping multiple tiles. Note how these bands become a larger percentage of the total server time as the overlap factor grows with increasing numbers of processors. The hybrid algorithms incur no such overheads due to overlaps, as every object is rendered exactly once by a server. As a result, the server times simulated with the hybrid algorithm scale better than with sort-first.

Speedup curves for the hybrid and sort-first algorithms can be compared directly in Figure 9. For 64 processors, the speedups of the hybrid approach are 36.3, 43.7 and 46.5 for Hand, Dragon and Buddha respectively, whereas the sort-first approach speedups are 8.0, 14.9, and 15.1. Overall, the efficiency of the hybrid algorithm is generally around 3 to 4 times better than the sort-first algorithm in these tests.

### 7.4 Comparison to a Sort-Last Algorithm

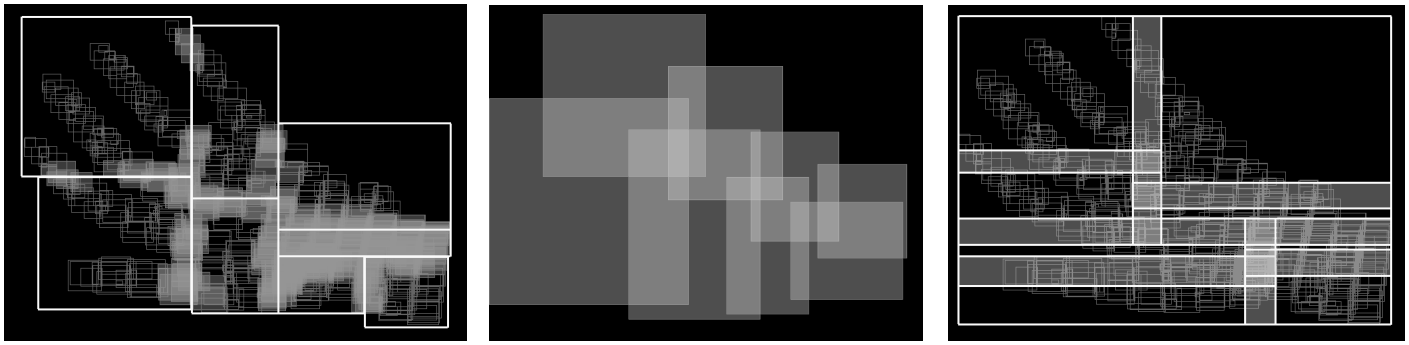
In our fourth study, we compare our hybrid partitioning algorithm to a sort-last approach. For the purposes of this experiment, we have implemented a polygon rendering version of a sort-last algorithm motivated by Neumann's "Object Partition with Block Distribution" algorithm [22]. This algorithm is used for comparison because it requires the least composition bandwidth, provides the best balance, and fits into our peer-to-peer image composition system better than any other sort-last algorithm we are aware of.

In our implementation of Neumann's algorithm, groups of objects are partitioned among servers statically using an algorithm that is a direct 3D extension of the 2D method described in Section 5. Then, during each frame of an interactive visualization session, every server renders its assigned group of objects into its own frame buffer. Then, the servers redistribute rendered pixels according to a static assignment of an interleaved fine-grained grid of tiles. Specifically, after rendering is complete, every server reads the pixels in the bounding box of rendered objects from both its color buffer and its Z-buffer and sends them to the server PC to whom the tile has been assigned. Upon receiving pixels, the servers composite them into their frame buffers. Finally, after all tiles have been fully composited, the servers read the resulting pixels back from their frame buffers and send them to the display PC so that they can be composited into a complete image for display.

Simulation results for this sort-last algorithm are shown in the bottom one-third of Table 1. The key comparison points are client processing times and server overheads.

Clearly, the client processing times for the sort-last algorithm are significantly less than for the hybrid and sort-first methods (see Figure 7). The only work that must be done by the sort-last client is to transform $N$ 3D bounding boxes into the 2D screen space so that "active" tiles can be marked, and server-to-server pixel transfers can be avoided for the others. This simple processing could easily be done directly on the servers, or possibly on the display PC. So, in effect, the hybrid and sort-first algorithms require one more PC's processing power than the sort-last algorithm - an advantage of pure sort-last. However, the impact of this difference is rather small for large numbers of servers.

On the server side, there is a significant difference between the sort-last and hybrid algorithms in pixel redistribution costs (see the columns labeled "Pixel Read" and "Pixel Write" in Table 1). The sort-last algorithm incurs more overheads, as it must perform image composition transfers and processing for larger numbers of pixels (see Section 6). The difference can be seen in our simulation results as the white regions in the rightmost set of bars in Figure 8. In the sort-last case, the average number of pixels transferred between



(a) Sort-first (b) Sort-last (c) Hybrid

Figure 10: Visualizations of server overheads with different algorithms. In (a) highlighted object bounding boxes span multiple tiles and must be rendered redundantly. In (b) and (c) brighter pixel intensities represent more image composition overheads.

servers during each frame is related to the scene depth complexity (D) multipled by the resolution of the screen (P), where D is approximately $N^{1/3}$ and the total overhead is $\frac{P}{\sqrt[3]{N^2}}$ [15, 21]. In contrast, for the hybrid algorithm, the number of pixels transferred is related to $2B\frac{\sqrt{P}}{\sqrt{N}} + B^2$, where B is the average size of an object's bounding box (see Section 6). Empirically, the pixel redistribution overheads of the hybrid algorithm are much smaller than for sort-last in all of our simulations.

The difference between the pixel redistribution costs can be seen very clearly in the visualizations of Figures 10(b) and Figure 10(c), which show pixels transferred across the network for the sort-last and hybrid algorithms respectively, shaded in transparent white. Just by looking at "the amount of white" in these images, one gets an intuitive feel for how much pixel redistribution overhead is incurred by the system.

Overall, the efficiency of the hybrid algorithm is approximately 33% to 53% better than the sort-last algorithm in all of our tests.

### 7.5 Impact of Increasing Screen Resolution

In our fifth study, we analyze the effect of increasing screen resolution on each of the partitioning algorithms discussed so far: hybrid, sort-first, and sort-last. There are two issues to consider – display times (for all algorithms) and server times (for hybrid and sort-last).

First, consider display times. As previously discussed, the display PC can be the limiting factor if it is not able to receive data from the network or write it into its frame buffer quickly enough to keep up with the client and servers. Of course, this problem gets worse with increasing screen resolution. For example, increasing the resolution from 1280×960 to 2560×1920, the final display time will increase by a factor of 4. However, this concern may go away in a few years, as PC network and AGP bandwidths improve at a rate higher than screen resolution.

Second, consider server times. Here, there are fundamental differences between sort-first, sort-last, and our hybrid algorithm. For sort-first, the dominant overhead (overlap factor) is completely independent of screen resolution (until the rendering becomes rasterization bound). As a result, sort-first is probably appropriate for very high-resolution screens, such as multi-projector display walls [28].

In contrast, the sort-last and hybrid algorithms are definitely impacted by higher screen resolutions. It is important to note, however, that the pixel redistribution overheads of the sort-last algorithm grow linearly with $P$, while the overheads of our hybrid algorithm grow only with $\sqrt{P}$. Essentially, sort last must composite areas, while the hybrid algorithm composites perimeters. This is a significant difference, which can be seen very clearly when comparing Figures 8 and 11, which show breakdowns of server times for display resolutions 1280x960 and 2560x1920. The significantly taller white areas in the bars for sort-last indicate larger pixel redistribution overheads that thwart speedups for high resolution screens. By comparison, pixel redistribution times are around 5 times less for 2560x1920 images with the hybrid algorithm.

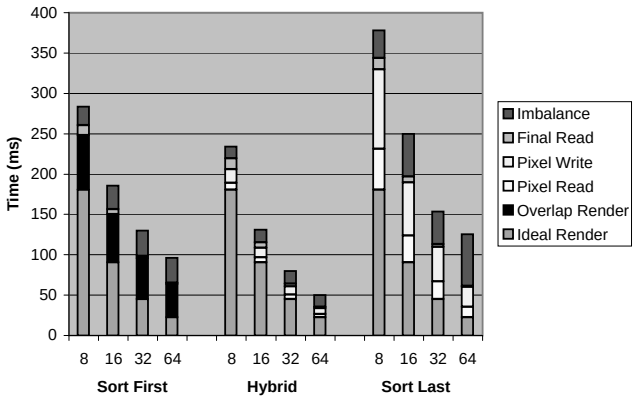



Figure 11: Server times for sort-first, sort-last, and the hybrid algorithms at screen resolution 2560x1920. The height of each bar represents the time required for processing in the server.

### 7.6 Impact of Increasing Object Granularity

In our final study, we investigate the effects of different object granularities on partitioning algorithms. Figure 12 and 13 show the client and server times measured with three different object granularities (399, 1,476, and 6,521 objects, respectively) for the Buddha model – each case has the same number of polygons, just separated into different numbers of objects. In this case, we can trade-off increases in client processing for better efficiencies in servers with the hybrid and sort-first algorithms. The sort-last algorithm is largely unaffected.

First, consider the impact of increasing the number of objects to be processed by the client. Judging from the curves in Figure 12, we see that the client times of all algorithms increase linearly with object granularity. This result is due to a combination of linearly increasing object transformations and linearly increasing partition processing times. For 6,521 objects, the client processing time for the hybrid method clearly becomes the bottleneck.

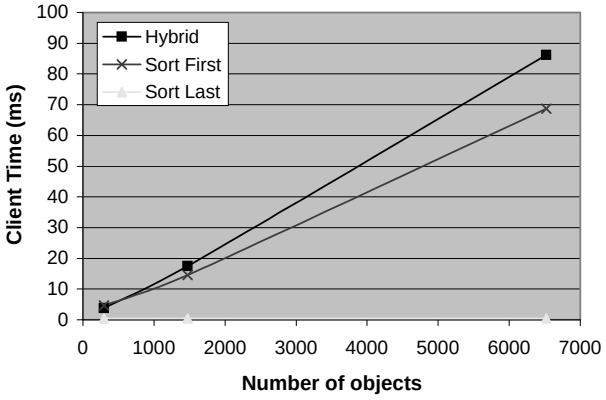



Figure 12: Plot of client processing times for different granularities of objects in the Buddha model.

Second, consider the impact on the servers. We see in Figure 13 that the relative overheads for the hybrid and sort-first algorithms decrease significantly for finer object granularities. Essentially, as object bounding boxes get smaller, the overlaps get smaller, as predicted by Molnar et al. [16]. In the case of the hybrid algorithm, the effect is to make the region on the boundary of each tile to be composited with neighbors "thinner." This effect can be seen clearly in the images of Figure 14, which show visualizations of pixel redistribution costs (more white means more overhead) for three different object granularities – the "swaths" requiring image composition get thinner with shrinking bounding boxes.

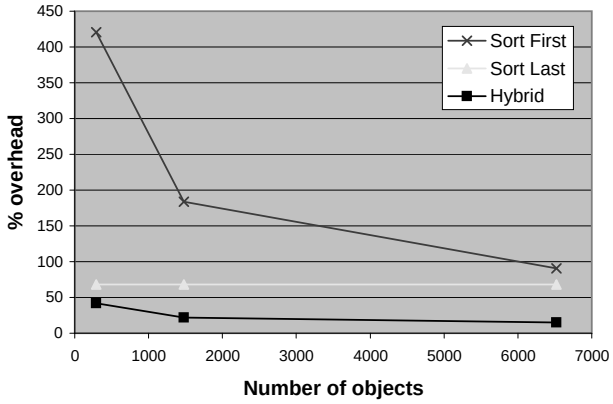



Figure 13: Plot of server overheads as a percentage of ideal rendering time for different granularities of objects in the Buddha model.

In the limit, as the object granularity of increases, and bounding box widths approach zero, the overheads of hybrid and sort-first algorithms tend toward zero. Hence, faster clients can significantly improve server efficiencies by partitioning at finer object granularities. This is a significant difference from the sort-last algorithm, which is unaffected by object granularity. Developing and analyzing the effects of hierarchical and/or parallel client algorithms is an interesting topic for future study.

## 8 Discussion

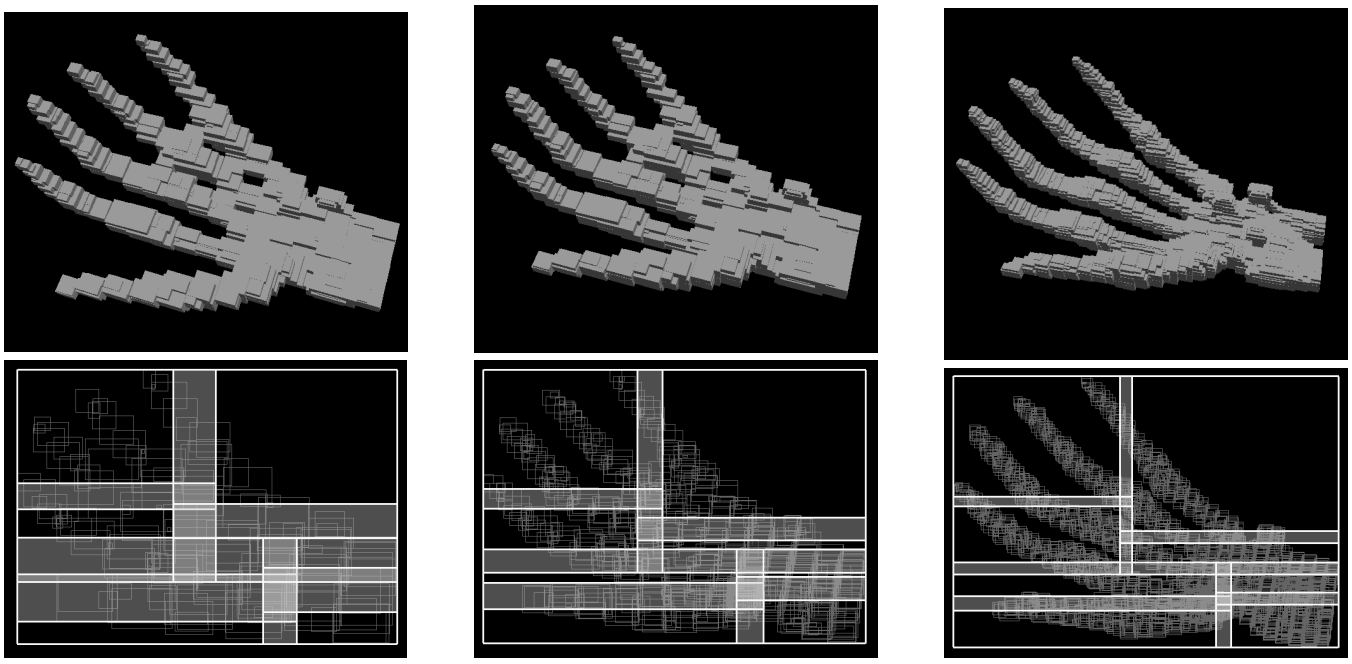
The novel idea presented in this paper is that of dynamic, view-dependent partitioning of both the 3D model space and the 2D screen space for parallel polygon rendering on a cluster. The key result is that this approach both performs better and scales better with processor count than the best known earlier approaches, at least in the cluster environment. Compared to pure sort-first approaches, this approach greatly reduces the problems that arise due to objects overlapping multiple screen partitions. Compared to sort-last approaches with one-time, view-independent partitioning of the model, it greatly reduces the bandwidth required for pixel transfer. The approach has further advantages in situations where applications not only render the model from different viewpoints but also zoom in on certain sub-areas and use greater detail in those areas (which would make static partitions perform poorly since the area zoomed in on would be assigned to only a small number of processors in that case).

The approach is not without disadvantages, however, and these too stem from the dynamic, view-dependent nature of computing partitions. The major disadvantages are in the following two areas.

First, a separate client machine computes partitions for every frame, so the speed of partitioning on the client can limit the achieved frame rate. This tradeoff is particularly visible in our approach, since unlike in other approaches making objects smaller continues to help the parallel rendering rate, asymptotically resulting in near-zero overheads. However, making objects smaller makes partitioning more expensive and hence makes it difficult for the client to keep up with the servers. A possible solution is to use a parallel client-side engine for partitioning. Another is to have the client maintain a hierarchical view of the objects in the model and subdivide objects adaptively as it computes its partitions, as opposed to the current method of fixing the sizes of objects up front. These methods remain topics for future investigation.

Second, with a dynamic, view-dependent approach, the 3D data rendered by a given server may change from frame to frame. In the absence of a shared address space, it can be both difficult and overhead-consuming to manage the partitioning or distribution of the 3D model data among processors and the dynamic replication of data on demand. We have therefore chosen to replicate the entire 3D model on all processing nodes for now, to eliminate the data management problem and focus on the partitioning issues. However, this clearly restricts our current implementation to rendering static models and limits the size of the models that can be rendered. Since dynamic data distribution and replication will introduce runtime overheads, to build a truly scalable approach we must examine methods to perform this data management efficiently. We plan to do this in future work, examining system-level approaches like software implementation of a shared memory programming model or memory servers on clusters, as well as application-level approaches like customized "shared memory" or conservative replication of data at partition boundaries based on the partition construction. It is worth noting, of course, that an approach that uses static partitioning of the 3D model does not suffer from this replication problem, so it is likely that for situations in which the 3-D model is very large and the screen is small (so that sort-last bandwidth is not a major problem), a static sort-last approach that avoids the data management overheads may work best as long as there is enough bandwidth to sustain the desired frame rate.

We expect that limitations of our approach should diminish with time and its performance and scalability should improve further, given technology trends. In particular, both client processor speed and bandwidth tend to grow much faster than display resolution. The former helps the client partition at finer granularities, while the latter helps reduce the overheads of pixel redistribution by alleviating the dynamic data management problem. We therefore believe that this type of dynamic, view-dependent approach is likely to be a very good one for parallel polygon rendering on commodity clusters (and potentially other parallel machines), as long as the data management problems can be solved satisfactorily.



(a) 293 objects (b) 1,476 objects (c) 6,521 objects

Figure 14: Visualizations of tile overlaps in partitions created with our hybrid algorithm using different object granularities. The top row of images shows 3D object bounding boxes. The bottom row shows tiles with overlap regions shaded. Larger overlap regions cause higher server pixel redistribution overheads.

## 9 Summary and Conclusion

This paper presents a hybrid parallel rendering algorithm that combines features of both "sort-first" and "sort-last" strategies for a PC-cluster architecture. The algorithm dynamically partitions both the 2D screen into *tiles* and the 3D polygons into *groups* in a view-dependent manner per-frame, to balance the rendering load among the PCs and to minimize the bandwidths required for compositing tiles with a peer-to-peer redistribution of pixels.

During simulations of our working prototype system with varying system parameters, we find that our hybrid algorithm outperforms sort-first and sort-last algorithms in almost all tests, including ones with larger numbers of processors, higher screen resolutions, and finer object granularities. Overall, it is able to achieve interactive frame rates with efficiencies of 55.0% to 70.5% during simulations of a system with 64 PCs. We believe this approach provides a practical, low-cost solution for high-performance rendering of static, replicated 3D polygonal models on moderately sized clusters of PCs.

Topics for futher study include development of algorithms for dynamic database management, support for fast download of images from a network into a frame buffer, and methods to partition graphics primitives among servers with finer granularity using either parallel clients, hierarchical algorithms, or faster algorithms.

## Acknowledgements

This project is part of the Princeton Scalable Display Wall project which is supported in part by Department of Energy under grant ANI-9906704 and grant DE-FC02-99ER25387, by Intel Research Council and Intel Technology 2000 equipment grant, and by National Science Foundation under grant CDA-9624099 and grant EIA-9975011. Thomas Funkhouser is also supported by an Alfred P. Sloan Fellowship.

We would like to thank Jiannan Zheng who discussed with us the hybrid algorithm, and Yuqun Chen and Xiang Yu who designed and implemented the VMMC-II communication mechanisms for the PC cluster. We also would like to thank Greg Turk and the Stanford Computer Graphics Laboratory for sharing with us the models used in our experiments.

## References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] Y. Chen, A. Bilas, S. Damianakis, C. Dubnicki, and K. Li. Utlb: A mechanism for translations on network interface. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operaing Systems*, pages 193–204, 1998.

[3] J. Clark. A VLSI geometry processor for graphics. *Computer (IEEE)*, 13:59–62, 64, 66–68, July 1980.

[4] James H. Clark. The geometry engine: A VLSI geometry system for graphics. *Computer Graphics*, 16(3):127–133, July 1982.

[5] Michael Cox. *Algorithms for Parallel Rendering.* PhD thesis, Department of Computer Science, Princeton University, 1995.

[6] Thomas W. Crockett. Parallel rendering. Technical Report Technical Report TR95-31, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1995.

[7] T.W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23:819–843, 1997.

[8] David Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics and Applications*, 14(4):33–40, 1994.

[9] H. Fuchs and B. Johnson. An expandable multiprocessor architecture for video graphics. In *Proceedings of the 6th Annual ACM-IEEE Symposium on Computer Architecture*, April 1979.

[10] Henry Fuchs. Distributing a visible surface algorithm over multiple processors. In *Proceedings of the 1977 ACM Annual Conference, Seattle, WA*, pages 449–451, October 1977.

[11] Thomas A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. In *Computer Graphics (SIGGRAPH 96)*, 1996.

[12] C. Grietsen and J. Petersen. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, 1993.

[13] Jeremy Hubbell. Network rendering. In *Autodesk University Sourcebook*, volume 2, pages 443–453. Miller Freeman, 1996.

[14] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. In *SIGGRAPH 98 Conference Proceedings*, pages 141–150. ACM SIGGRAPH, July 1998.

[15] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[16] S. Molnar. Image-composition architectures for real-time image generation. Ph.d. thesis, University of North Carolina at Chapel Hill, 1991.

[17] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[18] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. Infinitereality: A real-time graphics system. In *Proceedings of Computer Graphics (SIGGRAPH 97)*, pages 293–303, 1997.

[19] Carl Mueller. The sort-first rendering architecture for high-performance graphics. In *ACM SIGGRAPH Computer Graphics (Special Issue on 1995 Symposium on Interactive 3-D Graphics)*, 1995.

[20] Carl Mueller. Hierarchical graphics databases in sort-first. In *Proceedings of the IEEE Symposium on Parallel rendering*, pages 49–57, 1997.

[21] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 97–104. ACM, November 1993.

[22] Ulrich Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, 1994.

[23] Frank A. Ortega, Charles D. Hansen, and James P. Ahrens. Fast data parallel polygon rendering. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 709–718. IEEE Computer Society Press, 1993.

[24] F. I. Parke. Simulation and expected performance analysis of multiple processor z-buffer systems. In *Proceedings of Computer Graphics (SIGGRAPH 80)*, pages 48–56, 1980.

[25] Pixar. *PhotoRealistic RenderMan Toolkit*. 1998.

[26] R.J. Recker, D.W. George, and D.P. Greenberg. Acceleration techniques of progressive refinement radiosity. In *Computer Graphics (Proceedings of the 1990 Symposium on Interactive 3D Graphics)*, pages 59–66, 1990.

[27] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of pcs. In *SIGGRAPH 2000 Technical sketches*, August 2000.

[28] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH '99. Proceedings 1999 Eurographics/SIGGRAPH workshop on Graphics hardware, Aug. 8–9, 1999, Los Angeles, CA*, pages 107–116. ACM Press, 1999.

[29] Bengt-Olaf Schneider. Parallel rendering on pc workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDTA98)*, Las Vegas, NV, 1998.

[30] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, July 1994.

[31] Daniel S. Whelan. Animac: A multiprocessor architecture for real-time computer animation. Ph.d. thesis, California Institute of Technology, 1985.

[32] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett Publishers, 20 Park Plaza, Boston, MA 02116, March 1992.

[33] Scott Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, 1994.