A Distributed System for Near-Real-Time Display of Shaded Three-Dimensional Graphics

Steve Sistare and Mark Friedell
Aiken Computation Lab
Harvard University
Cambridge, MA 02138

Abstract

We present a distributed system that provides low-cost, interactive response for three-dimensional graphics applications. The system runs on commonly available hardware, which consists of a set of distributed workstations connected by a medium-bandwidth network. Problems such as heterogeneous processors, time-shared use of the workstations, and non-uniform image complexity are addressed through a load-balancing algorithm that achieves effective parallel speedups, allowing near-real-time response in interactive applications. A scheme for exploiting frame-to-frame coherence, which is characteristic of many interactive applications, further enhances the performance. The images rendered may be displayed at near-real-time rates on an inexpensive, 8-bit frame buffer through the use of a novel, color-quantization scheme that avoids recomputing a color mapping for every frame of an animation sequence. Several applications of the system are described, and a video showing near-real-time interaction with these applications will be presented.

KEYWORDS: Color Quantization, Distributed Rendering, Frame-to-Frame Coherence, Interactive Applications, Load-Balancing Algorithms, Near Real-Time Rendering, Parallel Processing.

1 Introduction

Interactive graphics applications incorporating shaded renderings of three-dimensional scenes are extremely demanding computational tasks. Ordinarily, these applications require dedicated use of expensive, special-purpose, graphics hardware. This is certainly true when real-time frame rates, e.g., 30 frames per second, are required.

In many cases, however, a near-real-time frame rate, e.g., one frame per one to three seconds, is acceptable, and the possibility of using more common computing hardware exists. In [1], Fuchs et al. describe their Binary Space Partition (BSP) algorithm, a clever approach to producing near-real-time renderings of three-dimensional scenes with a general-purpose computer and a high-performance graphics controller. The principle drawback to the BSP algorithm is that it is limited to static scenes, in which only the viewer's position and viewing direction may change.

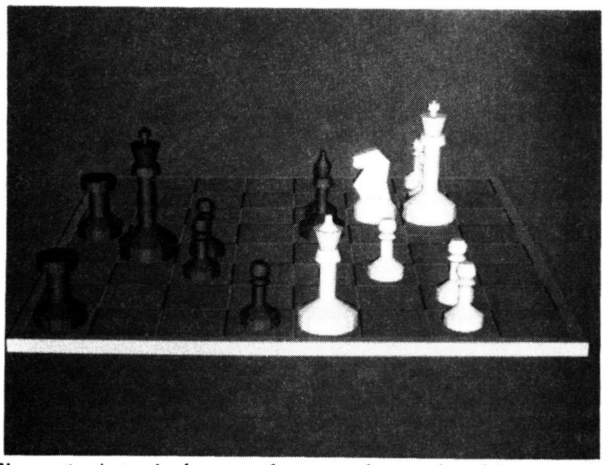
Figure 1: A typical scene that may be rendered in near-real-time. All images were generated in full color and reproduced in black and white.

In this paper, we describe a distributed technique for rendering with frame-to-frame coherence that provides near-real-time frame rates without restricting the scene or viewing parameters, i.e., the viewer's position and viewing direction may change, as may any part of the scene. An example of a scene we can manipulate in near-real-time is shown in Figure 1.

Our technique may be implemented on a collection of general-purpose, time-shared computers and workstations connected by a typical local-area network. One of the machines on the network must have a graphics controller, which could be an inexpensive device with only 8 bits per pixel. An example of appropriate hardware is a collection of Sun-3 workstations, one with a color console, connected by a 10-megabaud ethernet.

We certainly do not advocate purchasing a network of general-purpose computers to support interactive three-dimensional graphics. However, such hardware is increasingly common, and our technique allows that hardware to support demanding graphics applications that would otherwise require additional, specialized, graphics equipment.

2 The Graphics Pipeline

Typical rendering systems, whether implemented in software or hardware, usually operate in a pipelined fashion, with graphics primitives, e.g., polygons, sent through the following stages:

1. Transformation into perspective space
2. Clipping against the viewing frustum
3. Transformation into device coordinates
4. Hidden-surface elimination and shading

Any attempt to apply distributed, parallel processing to the rendering problem must involve breaking up this pipeline. One simple approach to doing so is to assign one processor to each of the above stages. This is the approach taken in early hardware graphics systems, and is motivated by the difficulty of building specialized hardware that can handle more than one of the stages well. Note that, for a specific scene, one processor in the pipeline could be a bottleneck if its job is too complex, and that the degree of parallelism achievable is low, with only four participating processors.

One way to overcome these drawbacks can be seen in the architecture depicted in Figure 2, in which each stage may have several general-purpose processors dynamically assigned to it. Work flowing into a stage is distributed over the available processors. However, this architecture has an excessive amount of overhead, as interprocessor communication must take place between each of the four stages of the pipeline (assuming a non-shared memory system).

Consideration of the data dependencies leads to a more natural and efficient decomposition of the graphics pipeline. The first three stages in the pipeline operate on an isolated polygon, while stage 4, hidden-surface elimination, must in general consider all polygons. Therefore, the division between stages 3 and 4 is a natural one, while the divisions between stages 1, 2, and 3 are not strictly necessary. Stages 1, 2, and 3, hereafter referred to as the *geometry phase*, may be coalesced, yielding the architecture of Figure 3. Finally, note that there is little or no advantage to having processors simultaneously work on both of these stages for a single frame, as the hidden-surface stage must wait for all data to pass through the geometry stage before it can do any work.

The architecture we use, shown in Figure 4, rectifies this last inefficiency. The Master Processor distributes polygons among the Slave Processors, which pass their polygons through the geometry phase. The output from this phase is sent back to the Master. When all intermediate results have been gathered, the Master redistributes the transformed polygons to the Slaves for hidden-surface elimination, step 4 in the pipeline. Image output from this step is sent to the Image Collector, which writes the image fragments to a display device. In addition to its duties as a system coordinator, the Master also serves as the host for the graphics application.

3 Choice of Hidden-Surface Algorithm

To efficiently decompose the hidden-surface stage for parallel processing, each process must operate on a set of polygons that have no visual interactions with polygons held by other processes. One simple approach, taken in [5] and here, is to partition the image into a grid, clipping polygons by the planes extending from the grid lines. The polygons in each grid cell are given to a separate processor. Virtually any hidden-surface algorithm can then be used to calculate the resulting image for each grid cell.

Transmission of the image fragments to the Collector in a reassonably short time presents a problem. The volume of raw-image data is too large to transfer over a medium-bandwidth network, and encoding algorithms that would compress the image are CPU-intensive. The solution is to choose a hidden-surface algorithm that can generate encoded image data directly. For example, a scan-line algorithm [6,7] could return a list of visible spans – in effect, a run-length encoded format. The Newell-Newell-Sancha algorithm [4] could produce similar output, by returning the extents of the polygon segments that ordinarily would be used for polygon scan conversion. The Newell-Newell-Sancha algorithm was chosen for our implementation for this reason.

4 Data Flow, Synchronization, and Load Balancing

Upon initialization, the Master Processor distributes a copy of the entire scene description to each Slave Processor. The scene is given in the form of a *Structured Display File* (SDF), an acyclic directed graph in which each node contains polygons in a local coordinate system and each edge represents a modelling transformation. By appropriate traversal of the SDF and application of the relevant modelling transformations, all polygons may be mapped into world coordinates. Between frames, the Master updates the scene description by broadcasting to the Slaves the required modifications to the SDF. Usually, this includes only updates to the modelling transformations.

As outlined earlier, each frame is processed by a geometry stage and a hidden-surface stage. During the geometry stage, the Slaves allocate from the Master sequences of numerical polygon identifiers that are defined by the order of SDF traversal. The corresponding polygons are clipped, transformed into image space, sorted into grid cells, and sent to the Master. Note that the Master grants a sequence to a Slave by passing only polygon identifiers, not entire polygon descriptions, over the network.

After all transformed polygons are received by the Master, the hidden-surface phase begins. The Master distributes the polygons in the grid cells to the Slaves,

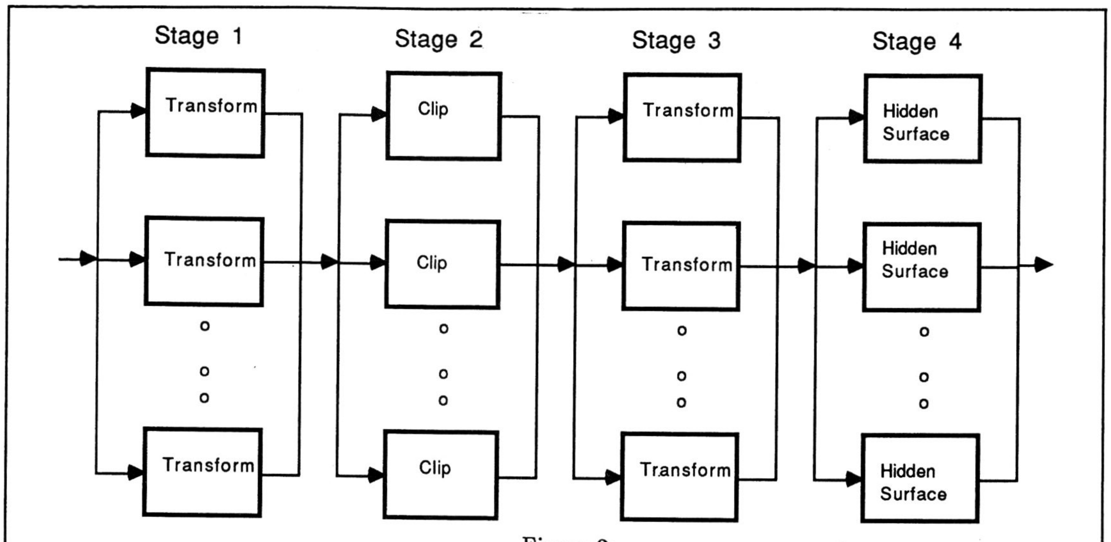

Figure 2:

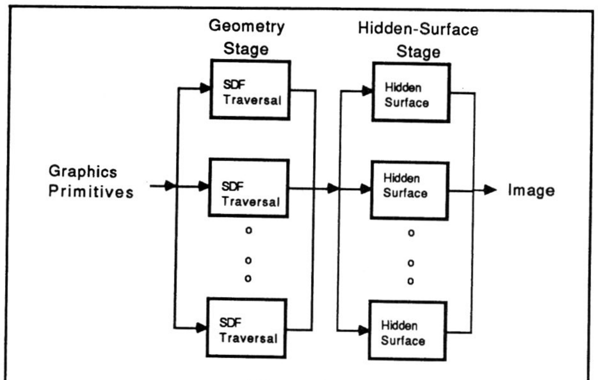

Figure 3:

and the Slaves send the rectangular image segments corresponding to their cells to the Collector, which assembles the complete image in its frame buffer.

4.1 Load Balancing

Load balancing is a critical component of our system, made necessary by the environment in which we are running as well as the nature of the rendering problem. For each phase, the problem is to divide the workload among p Slave Processors such that they all finish at the same time. Given the barrier synchronization between the geometry and hidden-surface stages, we must independently apply a load-balancing solution to each stage.

There are three reasons why statically partitioning the workload into p chunks is an inadequate solution. First, image complexity is not distributed uniformly and changes over time. Varying numbers of differently sized polygons will drift in and out of each region during the course of an animation. Second, the relative speed of each processor may be unknown and in any case is difficult to measure. Third, each processor in our network is time-shared, and response times will vary greatly depending on the multiprogramming level of each processor.

4.2 Load Allocation

To perform load balancing for each rendering stage, the Master maintains a pool of work items, which are allocated by Slaves as they become idle, with no requirement that the work items be of the same size. Fast Slaves will allocate more work items than slow Slaves during the time it takes to exhaust the pool, and the maximum difference in the total work time spent by any two processes will be bounded by the time it takes to process one work item. To see this, consider the system with one item left in the pool and all Slaves busy. The next Slave that goes idle will allocate the remaining item, and a different Slave may go idle immediately after this. Thus, better balancing is achieved with a large number of small work items. However, there is a fixed cost per work allocation, so total overhead increases with the number of work items. Consequently, increasing the number of work items will decrease the variance in execution time but raise the mean.

The smallest possible unit of work in the geometry stage is the polygon, but allocating polygons one at a time would involve too many allocation transactions for scenes with thousands of polygons. Instead, a group of polygons is handed out in each allocation, and the size of the group decreases with each successive allocation until single polygons are handed out in the final allocations. The result is that fewer allocations are necessary, and there is still a good chance that the processes will finish within one polygon-processing time of each other. The largest group of polygons is sized so that an unloaded Slave would be able to finish them and come back for several more allocations before the geometry stage is over. Thus, if the time-sharing load on a Slave goes up, it still has a reasonable chance of finishing at least its first allocation before the stage is over, and any successive allocations it would have received will go to other Slaves.

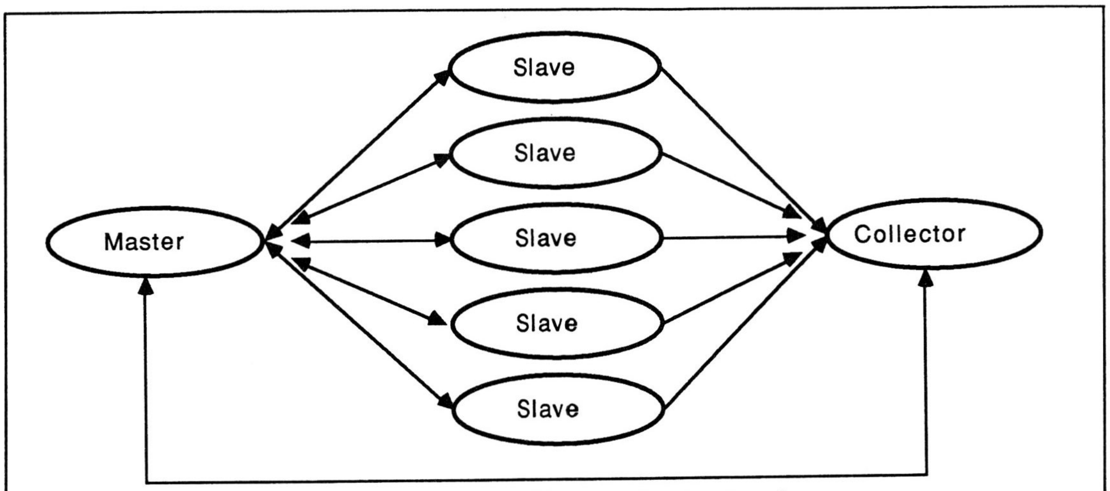

Figure 4: System architecture showing data flow

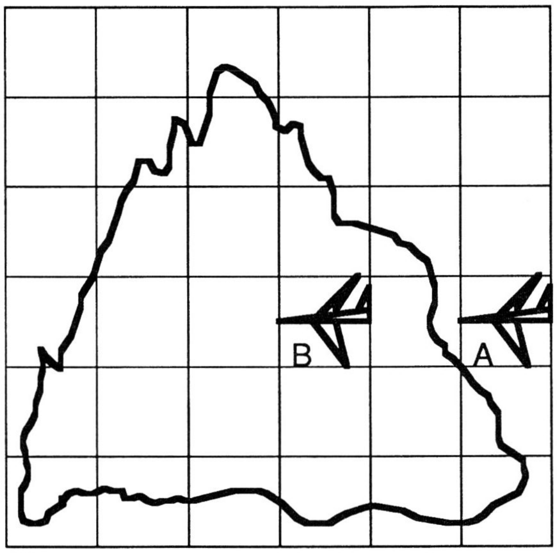

Figure 5:

Work in the hidden-surface stage may be divided as mentioned previously, i.e., by partitioning the image into a grid, with cells of the grid being allocated by Slaves one at a time. We have found empirically that making the grid sufficiently fine so as to produce six to eight cells per Slave Processor yields good load-balancing results. When a grid cell is allocated, the Master transmits to the receiving Slaves all polygons that were output from the geometry stage and lie in the cell.

5 Frame Coherence

Frame-to-frame coherence, or frame coherence for short, is said to hold when parts of an image remain unchanged from one frame in a sequence to the next [3]. The minimum requirement for frame coherence is that the viewer's position and line of sight in the environment must remain constant between frames. In addition, some of the objects in the environment must remain stationary. There are many applications that meet these requirements. An application generating animation sequences in which a foreground object moves inside an unchanging and potentially complex background environment exhibits frame coherence, as does an application in which an object is incrementally built by adding and subtracting single parts. Examples of the former include a viewer watching an airplane fly around a mountain, and a robot watching its mechanical arm sweep out across a factory floor. Examples of the latter include CAD, an incremental scene editor, and some forms of scientific visualization.

When frame coherence is present, there is a potential for computational savings if the system can isolate and recompute only the parts of an image that change. For example, in Figure 5, we have an airplane flying in front of a mountain as the airplane moves from cell A to cell B between frames. To render the next frame, cell A must be recomputed because the airplane is no longer there, and cell B must be recomputed because the airplane has moved into it. Any mountain polygons lying in cells A and B are included in the recomputation. None of the other cells in the image, which may include thousands of polygons, need be recomputed.

Frame coherence can be incorporated into a distributed system with few modifications to the system architecture: The existing grid that is used to divide the scene for the hidden-surface stage is used to isolate image changes between frames. When transformed polygons from the Slaves are sent to the Master at the end of the geometry phase and stored in the appropriate grid cells, the Master must maintain the polygons in the grid for possible use in future frames.

When an object moves - which happens when the graphics application changes a modelling transformation in the SDF that affects the object - the object's polygons are deleted from the Master's grid, and the grid cells that these deleted polygons were in are marked. When the Slaves allocate work during the geometry phase, they allocate only polygons belonging to moving objects. These moving polygons are transformed, sent back to the Master, and added to the appropriate grid cells, which are also marked. In the hidden-surface phase, all marked grid cells are allocated to the Slaves to be re-rendered. A marked cell may contain polygons describing moving objects that have just been deposited in the cell, as well as polygons from stationary objects that were deposited in the cell in earlier frames. Thus, moving and stationary objects will properly intersect and/or obscure each other. Finally, the Slaves send the image fragments resulting from each marked cell to the Collector, and the Collector paints these fragments over the areas on the image that the fragments replace, without repainting static image areas.

6 Static Color Quantization

Color quantization generally is required to display color images with a typical, low-cost, 8-bit frame buffer. There are many known color quantization algorithms. However, most algorithms that give good results are arduously slow, and the time to invoke color quantization can be much longer than the time to render the image!

In the context of our system, it is undesirable that color quantization be performed for each frame, as the resulting degradation in frame rate would be intolerable, even if the quantization were done in parallel. Instead, we have developed a method for static color quantization, in which color quantization is performed just once (after the scene description is defined) and the color map thus computed is used for all frames. The color domain to be quantized must include all possible colors that can be generated during the course of the animation. Without prior knowledge of the motions that the object will undergo, the best one can do is assume that every polygon in the scene can take on any orientation, and thereafter generate all possible final shades for each polygon based on the shading model being used.

The number of possible surface orientations, hence shades, is infinite, so the terms in the shading model involving the surface normal must be quantized into a finite number of values. For example, in a simple, faceted shading model, the scalar result of taking $N \cdot L$, where N is the surface normal and L is the light-source direction, must be quantized. The computation involved in generating all shades for all polygons can be reduced by recognizing that in the scene description, many polygons will have the same color. One need evaluate only the shading model for each (quantized) orientation and each polygon color, and weight each shade thus generated by the number of polygons that have the color. Strongly weighted shades will, in general, have less error introduced in the quantization than weakly weighted shades. The weights could be made more accurate by considering the average projected area of each polygon on the screen during the animation.

The shades and their weights are fed into a standard color-quantization algorithm, such as the median-cut algorithm, described in [2], and a color sampling and mapping to this sampling is produced. For fast access, we convert this mapping to a two-dimensional table, organized as follows. One dimension is indexed by polygon color. For this purpose, an integer color index is associated with each color and stored in every polygon. The other dimension is indexed by the quantity $N \cdot L$, which has been quantized. Thus, to determine the pixel value to be scanned into the frame buffer for any polygon in any frame, one just looks it up based on the polygon's normal, N, and color index.

The color map remains valid as long as parameters of the shading equation remain unchanged, such as ambient lighting and light-source direction, and as long as no new colors are added. When any of these factors changes, color quantization must be called again, resulting in a temporary delay in the animation.

The visual results achieved using static color quantization appear to be satisfactory, although at this time we are using only a faceted shading model. The acid test will come when smooth shading is incorporated, as quantization effects will be most apparent in that setting.

7 Implementation

Our experimental system runs on a collection of Sun workstations – one Sun-4 serving as the Master and up to N Sun-3 Slaves. However, the Sun-3's are of different models (and hence speeds), and the Slaves could include any variety of machine types.

The top-level interface to our system consists of a set of C subroutines that may be called to construct and render a scene description. The application programmer need not be concerned with the underlying parallel nature of the system.

When the Master is invoked, it first starts up the Slave Processes and the Collector using the UNIX call **rexec**, which creates a TCP/IP socket between the Master and each new process. The host machines used may be specified at run time. The Master then returns from initialization and executes the application program. Each Slave, upon startup, creates a socket connection to the Collector, and then goes into an endless loop of waiting for messages from the Master and acting upon them, terminating when the Master closes the connection.

The Collector behaves similarly to a Slave, looping and waiting for messages. Its only function is to receive image pieces and pass them to a frame-buffer controller for display. Since many Slaves send image messages to the Collector, it is possible that the Collector's input buffer could fill up, blocking further messages and holding up the Slaves until the buffer has room. To minimize this possibility, the Collector always listens for incoming messages during the hidden-surface stage (i.e., when the Slaves are generating image messages) and immediately copies any received messages into a temporary memory area. The image is not assembled and sent to the frame-buffer controller until the start of the geometry stage in the next frame. The Collector must be capable of putting up the image before the geometry stage is over, requiring a high-bandwidth connection between the Collector and its graphics controller. It is helpful if the controller is capable of drawing vectors, in which case the Collector's de-

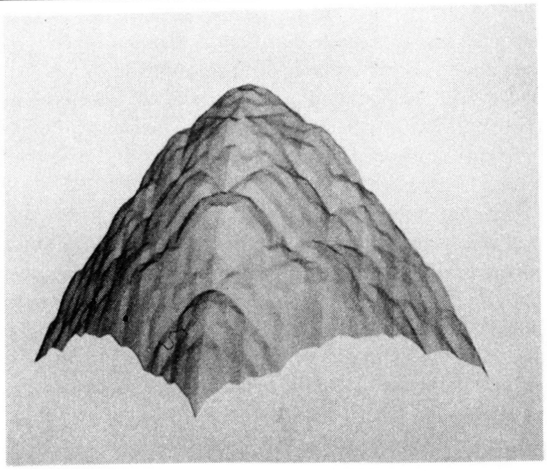
Figure 6: 4096-polygon mountain

coding task will be minimal, as polygon-segment-encoded images generated by the Newell-Newell-Sancha algorithm map directly into vectors. The controller that we use, a Sun-3 with graphics processor board, has this capability. An alternative available on most frame buffers, including generic Sun-3's, is to use the bitblt fill operation to fill out each polygon segment.

Absolutely essential to efficient communication over the ethernet is buffering of both input and output, as the system calls necessary for network communications are very expensive compared to the cost of moving data. As much data as possible must be transferred to and from the operating system in each **read** and **write** system call. Furthermore, it is noteworthy that data transmission rates over the net are limited as much by machine speed as by network bandwidth. One Sun-3, a 1-MIP machine, sending raw data to another Sun-3 over an ethernet achieves a maximum data-transfer rate of less than 200KB per second. If five Sun-3's transmit to one, the result is the same. However, if five (or more) Sun-3's transmit to one Sun-4, a 10-mip machine, the combined data transfer rate is 1MB per second, which is the full bandwidth of the network. The implication for our distributed system is that the fastest host available should serve as the Master, which in our case is a Sun-4.

8 Applications and Results

Several interactive graphics applications have been constructed using our distributed system, including a crude flight simulator and a crystalline-structure editor. The flight simulator was used to gather extensive timing data, shown in the Results section, from which one can evaluate the parallel efficiency of our distributed system. The simulator operates in two modes, which are purposely simplistic in order to generate repeatable and consistent timing data. The viewer can either continuously move around a mountain or remain stationary and watch an airplane move around the mountain. A typical image from this application may be seen in Figure 6.

In the first mode, the entire scene must be recomputed

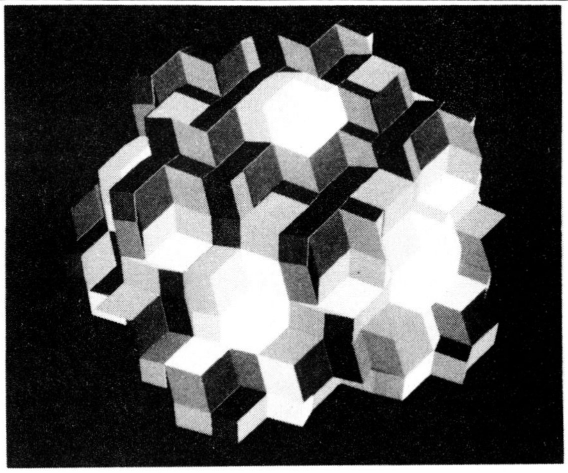
Figure 7: quasi-crystal

in each frame. One can see from the tables in the Results section that frames could be generated in just over two seconds with 1024 polygons in the mountain, which yields a surprisingly good rotational effect. Mountains with 256 polygons positively zip around, having frame-generation times of less than 1 second. In the second mode, frame coherence limits the recomputation necessary, and the airplane flys around the mountain at a rate of one to four frames per second. Thus, a wide variety of applications that interactively generate animation sequences using scenes of moderate complexity can be supported by our distributed system.

The quasi-crystal editor is a scientific-visualization tool that is being used in materials-science research at Harvard. Its purpose is to investigate the structure of the *quasi-crystal*, which is a solid crystalline structure. A typical quasi-crystal may be seen in Figure 7. Various polyhedra, such as tetrahedra and icosahedra, are used to represent structures within the crystal. The function of the editor is to help understand how these polyhedra may fit together exactly, with no space in-between, to form a solid crystalline structure. Supported actions include adding a polyhedron to an existing polyhedral face, deleting an existing polyhedron, and viewing the entire crystal from a different perspective.

Before our distributed system became available, the quasi-crystal application could be termed interactive only for very patient users, since simply adding a new polyhedron could take many seconds as the crystal being built became larger. When linked with our distributed system, the user detects no appreciable delay when a polyhedron is added or deleted, and the actual response time is just fractions of a second. This response is due more to frame coherence than the system's parallel-processing power, as excellent response can be achieved using only one Slave Process. When more Slave Processes are used, the extra Slaves are idle most of the time during polyhedron add/delete operations and waste no CPU cycles. However, when the user wishes to view the crystal from another perspective, all the Slaves are fully utilized, and the new view is generated rapidly. Using 10 Slaves, the

slaves	time (ms)	efficiency	overhead (ms)	speedup	cpu avail
14	1084	0.33	726	4.80	0.95
13	1063	0.36	676	4.98	0.95
12	1069	0.38	662	4.76	0.97
11	1138	0.40	683	4.62	0.96
10	1106	0.45	606	4.70	0.98
9	1216	0.48	633	4.42	0.96
8	1234	0.52	587	4.38	0.98
7	1370	0.55	619	3.97	0.97
6	1463	0.59	606	3.70	0.98
5	1669	0.63	618	3.35	0.96
4	1936	0.64	688	2.75	0.97
3	2313	0.70	700	2.31	0.98
2	3181	0.78	691	1.75	0.86
1	5737	0.70	1715	0.70	0.80

Table 1: polygons = 256

slaves	time (ms)	efficiency	overhead (ms)	speedup	cpu avail
14	2202	0.46	1186	6.68	0.98
13	2218	0.51	1082	6.81	0.98
12	2453	0.50	1221	6.11	0.96
11	2349	0.59	965	6.60	0.93
10	2463	0.59	1012	6.01	0.97
9	2601	0.61	1007	5.66	0.98
8	2787	0.65	986	5.29	0.97
7	2874	0.69	889	4.97	0.99
6	3407	0.72	969	4.40	0.97
5	4187	0.73	1127	3.74	0.96
4	5300	0.76	1285	3.06	0.98
3	7059	0.78	1564	2.36	0.99
2	11216	0.80	2248	1.60	0.98
1	21660	0.84	3422	0.84	0.95

Table 2: polygons = 1024

slaves	time (ms)	efficiency	overhead (ms)	speedup	cpu avail
14	7949	0.48	4096	7.27	0.90
13	6726	0.60	2689	7.98	0.96
12	7332	0.59	2993	7.29	0.96
11	7089	0.66	2426	7.40	0.98
10	7710	0.67	2547	6.90	0.97
9	8062	0.71	2339	6.57	0.97
8	8332	0.76	1968	6.30	0.97
7	9934	0.76	2407	5.46	0.97
6	11514	0.78	2539	4.81	0.98
5	13782	0.82	2457	4.19	0.98
4	19364	0.76	4579	3.12	0.96
3	25411	0.80	4981	2.46	0.97
2	42702	0.67	13983	1.37	0.98
1	57419	0.90	5850	0.90	0.98

Table 3: polygons = 4096

crystal in Figure 7 can be re-displayed in several seconds, which is satisfyingly interactive. The interactive requirements of the quasi-crystal editor are an ideal match to the capabilities of our distributed system, and there are many other applications having these same characteristics.

9 Analysis of Results

Timing results from distributed rendering using load allocation may be found in Tables 1, 2, and 3. The data in these tables was generated from the flight-simulator application, with the viewer moving around a mountain. The mountain contained 256 polygons in Table 1, 1024 polygons in Table 2, and 4096 polygons in Table 3, and all images were computed at a resolution of 640 by 483. The 4096-polygon mountain is shown in Figure 6.

The timing figures shown are averages taken over many frames and represent real elapsed time. The processors for the runs were used in a timesharing environment and were not 100% available to the graphics application. The aggregate CPU availability is indicated for each time measurement.

The entries in the table were derived as follows. Let

P = number of Slaves

T_{S_i} = serial CPU-time on Slave i

A_i = CPU availability of Slave i

T_p = actual parallel time

T'_p = ideal parallel time given 100% CPU availability

T''_p = ideal parallel time considering actual CPU availability

Then

$$T'_p = \left(\sum_{i=1}^{P} 1/T_{S_i}\right)^{-1}$$

$$T''_p = \left(\sum_{i=1}^{P} A_i/T_{S_i}\right)^{-1}$$

$$\text{aggregate CPU availability} = T'_p/T''_p$$

$$\text{efficiency} = T''_p/T_p$$

$$\text{overhead per processor} = T_p - T''_p$$

$$\text{speedup} = \left(\sum_{i=1}^{P} \frac{T_{S_i}}{A_i T_p}\right)/P$$

The quantities and formulae defined above require some explanation, as they are complicated by the presence of heterogeneous, non-dedicated processors. The quantity A_i was determined for each Slave by measuring the real time and CPU time spent during a compute-bound section of code in the distributed system. The ideal parallel time is the inverse of the sum of the rates of work achieved by each Slave Processor. In the formula for T''_p, the rates are scaled by the CPU availability. The definitions of efficiency and overhead follow naturally. Speedup is relative an average processor. Note that these formulae reduce to an easily recognizable form when T_{S_i} is the same for all i, and A_i is 100% for all i.

To understand the data presented, we must examine the potential sources of overhead that contribute to the calculated speedups and efficiencies. Overhead springs from four sources: time to partition the scene for the hidden-surface stage, communication time, allocation-request time, and barrier-synchronization time. The number of partitions for all runs was fixed, so partitioning time is independent of the number of Slaves. Communication time also will be a constant for a given number of polygons. Allocation-request time is the time required by the Master to respond to an allocation request and will remain constant as long as Slaves do not queue up waiting for service. Of course, for the initial request for work in each phase, queueing is unavoidable, as all Slaves want work at the same time. Thereafter, Slaves will finish their pieces of work at different intervals and thus spread out their additional requests. However, if the number of Slaves is large enough, the Master will not be able to keep up with them, they will queue up, and the overhead for allocation requests will rise. Finally, barrier-synchronization time is a function of the performance of the load-balancing algorithm.

Within each of Tables 1, 2, and 3, we see that the overhead per Slave is large for small numbers of Slaves, and it decreases to a relatively stable value as the number of Slaves increases. This is because sources of overhead which are independent of number of Slaves are distributed over an increasing number of Slaves, until contribution of these overhead sources per Slave becomes negligible. The overhead that remains stems largely from waiting time at barrier-synchronization points. The fact that this time is constant per Slave makes perfect sense, as all Slaves will have to wait for the last Slave to finish, and this finishing time will be one half of the average time to process a piece of work.

The efficiencies go down as the number of Slaves increases, because the overhead per Slave levels off, while the useful work done per Slave declines. In addition, we see that efficiency goes up with polygons for a given number of Slaves, as the amount of useful work gets larger with respect to the constant sources of overhead. In general, the efficiencies are low in comparison to many tightly coupled multiprocessor systems. However, the overhead that causes this low efficiency is composed largely of waiting time, which burns no CPU cycles. Thus, in a time-sharing environment, the cycles available during this waiting time may be put to good use by other processes on the hosts, and low efficiency does not necessarily imply waste in the context of the entire distributed system.

Tests done during moderately loaded system conditions give further evidence in favor of our approach to load balancing. The average frame times are of course higher than those for unloaded Slaves, but the significant result is that the individual frame times usually are within 10% of each other. Rarely, a frame may take several seconds longer than average, because a Slave Process does not get attention from its CPU. This is an unfortunate characteristic of UNIX that we must live with.

10 Conclusion

We presented a distributed system that gives interactive response to a wide variety of interactive applications requiring near-real-time shaded display of three-dimensional scenes. The system's hardware requirements are met easily by a network of ordinary, low-cost, personal workstations and an inexpensive frame buffer. The rendering technique allows arbitrary changes in the scene between frames and detects frame coherence when it exists. Applications that generate scenes with several hundred to several thousand polygons are well supported using approximately 10 processors, which can render these scenes in 1 to 3 seconds per frame. Larger scenes using more processors may still be rendered at sufficiently interactive rates. When degrees of frame coherence hold, frame generation times may be decreased greatly, giving instantaneous response to applications such as the quasi-crystal editor. The distributed-system approach to rendering will not become obsolete as technology improves; rather, the performance of a distributed system will only increase with advances in the speed of its components.

References

[1] Fuchs, H., G.D. Abram, E.D.Grant, "Near Real-Time Shaded Display of Rigid Objects", *Computer Graphics*, 17(3), July 1983, pp. 65-72.

[2] Heckbert, P., "Color Image Quantization for Frame Buffer Display", *Computer Graphics*, 16(3), July 1982, pp. 297-307.

[3] Hubschman, H. and S. Zucker, "Frame-to-Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World", *Transactions on Graphics*, 1(2), 1982, pp. 129-162.

[4] Newell, M.E., R. G. Newell, and T.L. Sancha, "A New Approach to the Shaded Picture Problem", *Proc. ACM Nat. Conf.* 1972, pp. 443.

[5] Parke, F. "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems", *Computer Graphics*, 14(3), July 1980, pp. 48-56.

[6] Watkins, G.S., *A Real-Time Visible Surface Algorithm*, Univ. Utah Computer Sci. Dept., UTEC-CSc-70-101, June 1970, NTIS AD-762 004.

[7] Wylie, C., G.W Romney, D.C. Evans, and A.C. Erdahl, *Halftone Perspective Drawings by Computer*, FJCC 1967, Thompson Books, Washington, D.C., pp. 49-58.