The Sort-First Rendering Architecture for High-Performance Graphics

Carl Mueller

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Interactive graphics applications have long been challenging graphics system designers by demanding machines that can provide ever increasing polygon rendering performance. Another trend in interactive graphics is the growing use of display devices with pixel counts well beyond what is usually considered "high resolution." If we examine the architectural space of high-performance rendering systems, we discover only one architectural class that promises to deliver high polygon performance *with* very-high-resolution displays and do so in an efficient manner. It is known as "sort-first."

We investigate the sort-first architecture, starting with a comparison to its architectural class mates (sort-middle and sort-last). We find that sort-first has an inherent ability to take advantage of the frame-to-frame coherence found in interactive applications. We examine this ability through simulation with a set of test applications and show how it reduces sort-first's communication needs and therefore its parallel overhead. We also explore the issue of load-balancing with sort-first and introduce a new adaptive algorithm to solve this problem. Additional simulations demonstrate the effectiveness of this algorithm. Finally, we touch on a variety of issues that must be resolved in order to fulfill sort-first's ultimate promise: millions of polygons for zillions of pixels.

1. Introduction

The demands for better interactivity and realism in applications such as vehicle simulation, architectural walkthrough, computer-aided design, and scientific visualization have continually been driving forces for increasing the graphics performance available from high-end graphics systems.

Interactivity implies that the images are drawn in real-time in rapid response to user input. This immediately brings out two requirements from the graphics system: it must be able to draw images at approximately 30 frames per second (real-time), and it must have low latency (rapid response).

Realism implies that the images are rendered from detailed scene descriptions, meaning that the scenes consist of many thousands of graphics primitives. Realism also requires a display system that can show the scenes with a level of detail matching what the eye can see. Providing such detail for a reasonable field of view requires millions of pixels.

There are a variety of display devices on the path toward offering better realism. The proposed HDTV standard aims at nearly two million pixels. CAVE-type immersive displays [5] cover 4 walls of a room with a total of 5 million pixels, a number much smaller than what is desirable. Even head-mounted displays (HMDs), which would seem to require many fewer pixels, already are reaching one million pixels per eye [12] and are expected to go much further. In fact, Kaiser Electro-Optics is working on an ARPA-sponsored project to create an immersive HMD system with 4.6 million pixels per eye [7].

The number of applications which will want to take advantage of such high-resolution display devices will only increase as such devices become more popular. Yet so far, the only way to generate interactive images for these devices requires massive duplication of graphics hardware. Without an efficient solution, use of such devices will be limited to those parties with large acquisition budgets.

2. Parallel Graphics Systems

The task of a graphics computer can be described fairly simply. Given a mathematical model of all the objects in a particular environment, it must compute the visual contribution of each object for each pixel in a given viewing plane. This is a type of sorting problem, a fact recognized by Sutherland, Sproull, and Schumacher in 1974 [18]. For interactive graphics, the task is performed in two major stages: transformation and rasterization. The former converts the model coordinates of each object primitive into screen coordinates, while the later converts the resulting geometry information for each primitive into a set of shaded pixels.

The graphics performance demanded by the aforementioned applications requires parallel processing at both the transformation and rasterization stages of the graphics pipeline. The former is needed to cope with the large number of primitives, while the latter is needed for the large number of display pixels. The choices for how to partition and recombine the parallelized work at the different pipeline stages lead to a taxonomy of different architectures: sort-first, sort-middle, and sort-last [13, 15].

We now briefly examine each of sort-first, sort-middle, and sort-last. In the following descriptions, we consider a framework of an application host computer working with a

UNC Sitterson Hall CB 3175; Chapel Hill, NC 27599-3175
phone: (919) 962-1878; email: mueller@cs.unc.edu

graphics computer subsystem. The latter consists of many parallel processors working to produce the desired images in real time. Initially, the display database is partitioned and distributed among all the processors.

2.1 Sort-first

In sort-first (figure 1), each processor is assigned a portion of the screen to render. First, the processors examine their primitives and classify them according to their positions on the screen. This is an initial transformation step to decide to which processors the primitives actually belong, typically based upon which regions a primitive's bounding box overlaps. During classification, the processors redistribute the primitives such that they all receive all of the primitives that fall in their respective portions of the screen. The results of this redistribution form the initial distribution for the next frame.

Following classification, each processor performs the remaining transformation and rasterization steps for all of its resulting primitives. Finished pixels are sent to one or more frame buffers to be displayed.

2.2 Sort-middle

In sort-middle (figure 2), there is a set of transformation processors and a set of rasterization processors. Physically, the two sets may use the same hardware, but they remain logically separate sets. Each rasterization processor is assigned a portion of the screen. To produce an image, each transformation processor completely transforms its portion of the primitives. The resulting primitive information is again classified by screen location and sent to the correct set of rasterization processors. After rasterization, finished pixels go to the frame buffer(s).

In contrast to sort-first, the original distribution of primitives is maintained on the transformation processors. For each frame, all of the transformed primitives must be routed to the correct set of rasterization processors.

2.3 Sort-last

For sort-last (figure 3), each processor has a complete rendering pipeline and produces an incomplete full-area image by transforming and rasterizing its fraction of the primitives. These partial images are composited together, typically by depth sorting each pixel, in order to yield a complete image for the frame buffer. The composition step requires that pixel information (at least color and depth values) from each processor be sent across a network and sorted along the way to the frame buffer.

Naturally, each architecture has a set of advantages and disadvantages. We outline these briefly here; for a more complete comparison, refer to [15].

2.4 Comparison

Sort-last is a very promising architecture and is discussed in detail in [13] and [14]. It offers excellent scalability in terms of the number of primitives it can handle. However, its pixel budget is limited by the bandwidth available at the composition stage. Using a specialized composition network can help to overcome this problem.

Anti-aliasing is a major problem for sort-last: regardless of the solution chosen, the composition task is non-trivial. Using super-sampling multiplies the amount of pixel bandwidth required, since each sample must be composited. A-buffer approaches introduce new complications to the composition process, since the number of fragments per pixel may vary and become arbitrarily large.

Finally, since visibility is not decided until after the composition stage, sort-last places limitations on the kinds of rendering algorithms which may be used. The choice of algorithms available for rendering transparent polygons becomes limited, for example, and visibility-based culling algorithms are less useful on sort-last.

Because of the way it builds upon traditional graphics pipelines, sort middle is a fairly natural architecture which has resulted in many implementations. Some examples are [1], [3], [6], [9], [11], and [21]. However, sort-middle's requirement that any transformation processor be able to talk to any rasterization processor means that its scalability is limited. Increasing the number of processors geometrically increases the demands on the communications network between them.

In addition, sort-middle faces load-balancing problems when the on-screen distribution of primitives is uneven. This will result in rasterization processors becoming unevenly loaded, and this in turn may degrade system performance unless careful attention is given to this problem. A variety of solutions have been used to address this issue (refer to the references above).

Sort-first is a promising architecture that has until now received little attention. It is the only architecture which inherently takes advantage of frame-to-frame coherence. In an interactive application, the viewpoint changes very little from frame to frame, and thus the on-screen distribution of primitives does not change appreciably. Since primitives in a sort-first system are only transferred when they cross from one processor's screen region to another's, only a fraction of them will have to be communicated each frame. Also, any

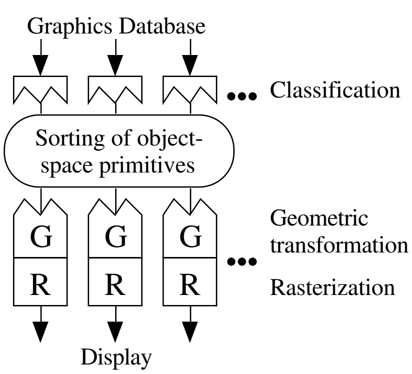


Figure 1. Sort-First Pipeline

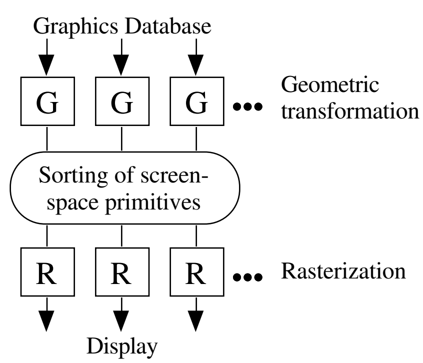


Figure 2. Sort-Middle Pipeline

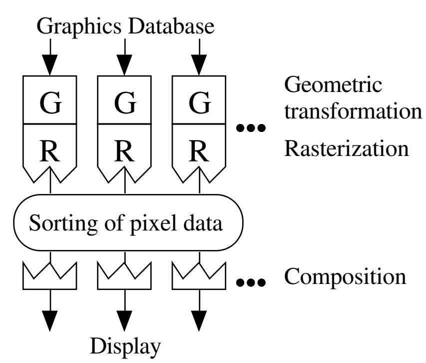


Figure 3. Sort-Last Pipeline

communication that does occur is typically fairly local; usually only "neighboring" processors will need to talk with each other. These facts suggest that it has good scalability in terms of the number of primitives it can handle.

In sort-first, once a processor has the correct set of primitives to render, only that processor is responsible for computing the final image for its portion of the screen. This allows great flexibility in terms of the rendering algorithms which may be used. All the speed-ups which have been developed over time for serial renderers may be applied here.

Since only finished pixels need to be sent to the frame-buffer, sort-first can easily handle very-high-resolution displays. This is the bottleneck for sort-last. Sort-middle also sends only finished pixels to the frame-buffer, but increasing the display resolution requires increasing either the size or number of rasterization processors, either of which causes problems. Thus sort-first is the only architecture of the three that is ready to handle large databases *and* large displays.

However, sort-first is not without its share of problems. Load-balancing is perhaps one of the biggest concerns: because the on-screen distribution of primitives may be highly variable, some thought must go into how the processors are assigned screen regions. Also, managing a set of migrating primitives is a complex task. These and other problems are the focus of this research.

3. Coherence Study

Because sort-first utilizes the coherence of on-screen primitive movement, we performed experiments to analyze this factor and determine what kind of savings might be achieved with actual applications. We wanted to know what fraction of primitives would need to be sent from processor to processor in a sort-first implementation. This testing was done using a simulation with several simplifying assumptions.

The testing involved two phases. The first was to make recordings from actual applications running on UNC's Pixel-Planes 5 graphics system. The resulting recordings contain a series of viewpoint information for each frame rendered while the application was run. The second phase was to take this information and the graphics database archive files and feed them to the simulation program. This program is based upon a framework written by David Ellsworth for his study of sort-middle systems [9]. Code was added to implement a sort-first partitioning and to calculate the resulting primitive traffic.

Various applications were used for the different test cases. "PLB" spins its database on the screen's vertical axis (named after a graphics performance benchmark from [16]). "Vixen" is a HMD-based visualization program that allows one to fly through an arbitrary display database. Finally, "Xfront" is similar except that it is joystick-controlled.

The setup for these tests is as follows:

- The database is simply a list of polygons (no structure).
- The aspect ratio of the screen is square.
- The screen is subdivided into equal-size square regions with one region assigned to each processor.
- The primitives are initially randomly distributed (the first frame's data is ignored for this reason).
- Primitives are redistributed according to the regions their bounding boxes cover.
- If a primitive falls into multiple regions, the processor at the upper-left region is deemed to be "in charge" of it.
- Off-screen primitives remain at the processor where they were last on-screen.

In these tests, the screen resolution is irrelevant; only the number of regions (and thus processors) matter. Several configurations of regions were tested: 4x4, 8x8, and 16x16. The simulation program outputs a series of values per frame representing the percentage of primitives that had to be communicated in that frame. From these figures, we calculate the arithmetic mean, the high value, the standard deviation, and the 95th percentile value.

For PLB, the database is a scanned model of a human head (see plate 1). The model is placed in the center of the screen and spun at 4.5 degrees per iteration around a vertical axis through its center (as in [16]).

PLB head	59,592 polygons, 80 frames		
regions:	4x4	8x8	16x16
mean	4.06 %	8.80 %	18.07 %
high	5.19	10.30	20.80
std-dev	0.54	0.70	1.05
95-%	5.07	9.92	20.06

For Vixen, the test case is a HMD walk-through of a Sitterson Hall's lobby (plate 2). The path starts on the mezzanine, goes down the stairs, and then turns around to look back at the starting point.

Lobby	16,267 polygons, 218 frames		
regions:	4x4	8x8	16x16
mean	2.13 %	4.95 %	11.41 %
high	21.17	45.17	87.44
std-dev	3.38	7.28	15.05
95-%	8.67	20.67	44.60

For Xfront, the model is a terrain database of a section of the Sierra Nevada mountains (plate 3). The model undergoes a series of zooms, rotations, and translations, with an abrupt reset between each sequence.

Sierra	162,690 polygons, 234 frames		
regions:	4x4	8x8	16x16
mean	3.17 %	6.08 %	11.51 %
high	98.07	102.26	107.38
std-dev	7.68	9.53	11.76
95-%	5.04	10.36	20.53

Looking at the results, we can see that increasing the number of regions increases the percentage of primitives that are communicated. This is fairly obvious, since increasing the number of region borders will increase the chance of a primitive crossing them.

The high values are somewhat interesting. For Sierra, the large values resulted from the abrupt transitions in this sequence. These exceeded 100% for two of the cases since primitives which fall into multiple regions may need to be sent more than once.

The percentiles perhaps are of greatest interest. They show that for moderately interactive applications (PLB, Xfront), 95% of the rendered frames require reshuffling of only about 20% of the primitives or less. For more highly interactive applications (Vixen), this figure goes up to about 45% in the worst case. As one may expect, this figure is directly related to the type of motion present in the application and how it affects the scene. A HMD user can create a lot of relative motion simply by rapidly turning his head.

The figures suggest that temporal coherence can provide sort-first with a dramatic savings in the amount of communication it must perform. The amount of savings is related directly to the nature of the application and the number of regions into which the screen is divided. Even for highly dynamic applications, the savings can be large, provided that the number of regions is kept small. As the number of regions increases, the amount of savings decreases in proportion, but remains quite substantial even for fairly large numbers of regions.

Although not demonstrated here, one may note that the frame-to-frame coherence is also proportional to the speed of the rendering system. An interactive system running at a higher rate will have smaller "deltas" between the frames. Thus the faster one can make a sort-first system perform, the more efficient it will be.

4. Off-Screen Primitives

We now examine some important issues of the sort-first architecture, starting with off-screen primitives. Such primitives are an interesting complication for sort-first, since these will not map to any processor's screen region(s). There are several alternatives for deciding what to do with them, each offering important tradeoffs.

One solution is to simply keep off-screen primitives on the processors where they were before they went off screen. This solution could ultimately lead to load-balancing problems or even memory overflow problems if a majority of the primitives go off-screen from the same processor. Since off-screen primitives still require geometric processing, this processor will be overloaded, assuming that the load-balancing algorithm (see below) does not take this into account. It is apparent that off-screen primitives have to be sent away eventually. The questions that come up are where to send them, and when?

As for the former, one could send them to "neighboring" processors, since this might offer a communications advantage; however these processors might then become overloaded themselves unless they also send the primitives away. However, sending off-screen primitives more than once seems counterproductive.

Another solution is to send them to a processor that is underloaded. This approach requires that processors distribute information about their primitive loads. Also, additional logic would be necessary to prevent many overloaded processors from sending their unneeded primitives to the same underloaded one.

An alternative place to send off-screen primitives is to a random processor. This method may be reasonable assuming that processors were fairly evenly loaded to begin with. If processor-load information is distributed, this method could be combined with the above approach by using this information to make it more likely that underloaded processors will receive primitives than overloaded ones.

The ideal solution for where to send off-screen primitives is a system and application dependent issue requiring consideration of many factors. The solution must be developed hand-in-hand with solutions to the load-balancing and database management problems (discussed below).

Aside from where, one must also consider the question of when to send away off-screen primitives. Primitives that are at the edge of the view screen might tend to pop in and out of view frequently, especially when there is some noise in the inputs that determine viewing direction. Thus it seems smart not to immediately send away a primitive that has gone off-screen (since it may be needed again shortly), but rather to keep it around for a few frames before doing so. A least-recently-used scheme would be applicable to decide which primitives to send away.

It should also be mentioned that this same issue arises with on-screen primitives as well. In a sense, any primitives that are outside a particular processor's region may be considered off-screen with respect to that processor. Such primitives may come back into a region soon after leaving it, and depending upon the costs of extra bookkeeping versus extra communication, it might be wise to keep a copy of departing primitives on-hand for a short time.

5. Load Balancing

The choice of strategy for mapping screen regions to processors has a critical impact on the performance of a sort-first system. A simple strategy such as dividing the screen into as many equal rectangles as there are processors and assigning them one-to-one can result in severe load-balancing problems. If the greatest concentration of primitives happens to fall into a single region, the parallel advantage of the system will be lost as the non-busy processors wait for the overloaded one to do its job. There are several approaches one can take to solve the load-balancing problem.

5.1 Static vs. Adaptive Region Assignment

One basic question is whether the assignment should be static or adaptive. If it is static, one must be careful about the assignment. If it is adaptive, then one needs a way to measure the rendering load across the screen, and then a way to divide the screen in a reasonable way. Each method has a set of advantages and disadvantages.

5.2 Static Methods

As its name suggests, static assignment requires a passive load-balancing approach. The general strategy is to divide the screen into more regions than there are processors and assign the regions to the processors in an interlaced fashion. The idea is that if the screen is divided finely enough, each processor will have portions of both the populated and the sparse areas, and thus have nearly equal loads.

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

Figure 4. Static Region Assignment

One problem with this approach is that there is still the possibility that a high concentration of primitives will fall into one region, no matter how small the regions are. However, in practice, this is fairly unlikely to happen; smart culling and level-of-detail control reduce this possibility.

Another obvious drawback is that increasing the number of regions per processor increases the amount of overhead. Still, the simplicity of this approach makes it worthwhile to study. With that in mind, several concerns come up: how should the regions be shaped, how many of them should there be, and how should they be assigned?

The obvious choices for region shapes are horizontal strips, vertical strips, or rectangles. For several reasons, rectangles are the preferred choice. The key is to minimize the total length of region boundaries in order to reduce the chances of primitives crossing these boundaries.

The question of how many regions there should be is not answered so easily, since this is dependent upon scene content and the number of processors. This subject is studied in the experiments described below. The question of region assignment can perhaps be addressed more simply. Unless there are special concerns involved, it is difficult to see why anything more complicated than regular interlacing would be beneficial.

5.3 Adaptive Methods

Adaptive region assignment offers the benefit of keeping the number of regions to a minimum, but at the cost of increased overhead and complexity. Adaptive methods come in a variety of algorithms. We turn our attention toward several previously-developed algorithms that happen to apply here, and we consider their possibilities. From there, we develop an algorithm which appears to combine many of the other algorithms' beneficial ideas.

5.3.1 Roble's Method

Roble describes an algorithm that starts with a standard rectangular decomposition [17]. According to the number of primitives in each region, lightly loaded regions are combined and highly loaded regions are split in half and assigned to the processors freed by the combining. The main problem with this algorithm is that there is no information on how to divide the highly loaded regions, and thus the resulting splits may add little benefit if most of the region's primitives fall on one side of the split.

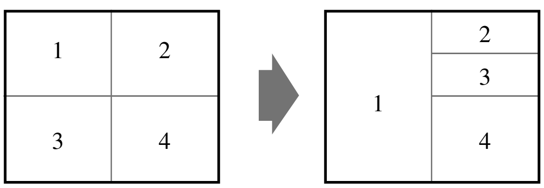


Figure 5. Roble's Method
Regions 1 and 3 are combined; processor 3 helps with original region 2.

5.3.2 Whelan's Method

Whelan proposes an algorithm known as median-cut [19]. Median-cut splits the screen into subregions based upon the distribution of the centroids of each primitive. The cuts recursively divide the longer dimension of the screen until the number of regions equals the number of processors. For large numbers of primitives, the sorting required by this approach makes it too time consuming, and since it only considers the primitive centroids, it is not sensitive to primitive size.

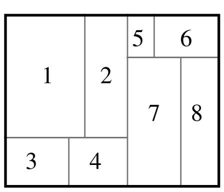


Figure 6. Whelan's Method
The screen is recursively subdivided according to primitive centroids.

5.3.3 Whitman's Method

Another strategy is Whitman's top-down decomposition [20], which starts by tallying up primitives based upon how their bounding boxes overlap a fine mesh. A unit is added to each mesh cell that the bounding box overlaps. After the tallying, adjacent mesh cells are combined and summed hierarchically to form a tree structure. The tree is then traversed top-down by splitting the region with the most primitives in half each time. To compensate for the fact that the resulting regions may still have largely varying numbers of primitives, Whitman subdivides until the number of regions is ten times the number of processors. Dynamic task assignment is used to even out the processor load balance. However, the resulting finer granularity of the regions results in more overhead for this method.

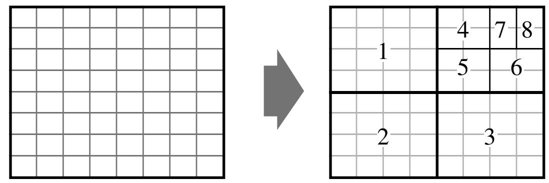


Figure 7. Whitman's Method
Mesh cells are combined hierarchically; the screen is split by traversing the hierarchy.

5.3.4 Combining It All: MAHD

Combining some of the above ideas leads to the algorithm presented here. This adaptive algorithm also uses a fine mesh to tally the primitives. To avoid errors caused by counting large primitives multiple times, the amount tallied to each cell is inversely proportional to the number of cells a primitive covers. Once all the primitives have been counted, the cells are summed into a summed area table [4]. Finally, the screen is divided along cell boundaries using a hierarchical approach similar to that of median-cut. The summed-area table allows a binary search operation to determine the location of each cut. Also, the algorithm allows for using a number of processors that is not a power of two by choosing appropriate split ratios rather than always dividing regions equally. Hereafter, we refer to this algorithm as the mesh-based adaptive hierarchical decomposition algorithm, or MAHD for short.

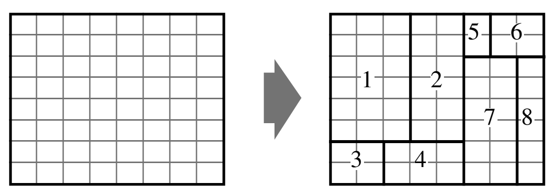


Figure 8. The MAHD Method
Mesh cells are used to accelerate Whelan-like subdivision.

Tallying the primitives inversely to their area is somewhat questionable, since although transformation costs may be equal, larger primitives do have greater rasterization costs. However, empirical results show an improvement using this modification.

5.4 Load-Balancing & Off-Screen Primitives

Off-screen primitives must still be classified in order to determine when they become on-screen again. This processing of off-screen primitives is an additional factor which must be considered in a load-balancing solution.

For static methods, the flexibility in dealing with off-screen primitives can be used to compensate for the lack of flexibility in dealing with on-screen primitives. Processors that have few on-screen primitives may be assigned additional off-screen primitives to deal with. This would require processors to distribute their load-balance information. One must be cautious with this approach, however, since the time saved by the extra load-balancing achieved might be lost by the overhead of sending primitives unnecessarily.

While adaptive methods can utilize the technique above, they also can include off-screen primitive counts in their screen-subdivision calculations. The number of off-screen primitives per processor can be used to adjust the target number of on-screen primitives for each processor to have as a result of the subdivision.

5.5 MAHD Algorithm Details

5.5.1 Parallel Issues

Implementing the MAHD algorithm across a parallel machine adds some issues which must be addressed. Making the sum table requires collecting data from all of the processors about how their primitives are distributed. Fortunately, the amount of data from each processor is likely to be proportional to the size of its region, and in this respect the algorithm scales reasonably. Thus a single processor collects the tally tables from all the others to build the complete sum table. It uses this table to perform the subdivision and then broadcasts the results to all processors.

Another implementation concern is the scheduling of the algorithm steps. If one wanted only to achieve the best load balance, the sequence of steps for a frame would be to first pre-transform the primitives to find out their distribution and the resulting screen subdivision, then sort and render all the primitives. The problems with this approach are 1) that it requires two full passes over all the primitives, and 2) that processors must synchronize with each other between these two passes. These problems mean decreased efficiency and increased latency.

One solution approach for this problem is to reduce the cost of the first pass by sampling the database rather than processing every primitive. This adds questions about how to perform the sampling. A similar approach may be possible if the database has a structured, hierarchical representation [2]. The primitive distribution could be estimated by examining where the structures fall, rather than examining all the primitives within each structure.

Another solution is to eliminate the extra pass by performing both operations at the same time. While transforming the primitives for frame n, the processors keep track of the distribution information in order to compute the subdivision for frame n+1.

The disadvantage of this solution is that the current frame's subdivision is based upon "old" data. However, because of the expected temporal coherence of frames, we expect the old data to be good enough for this purpose. We investigate this assumption in the experiment presented below.

5.5.2 Mesh Size

As mentioned, the MAHD algorithm uses a "fine mesh" in order to calculate the primitive load across the screen and also as a quantum basis for making screen subdivisions. The question then arises of how fine this mesh should be: smaller cells allow a more precise measurement, but increasing the number of cells increases the algorithm overhead. The question of mesh size is related to the question of how many regions per processor are necessary for the static algorithm. These questions are investigated in the experiment below.

5.5.3 Overhead Costs

The MAHD algorithm adds two basic operations to the processing cycle: the tallying of the primitives and the computation of the screen subdivisions. Additional costs include the communication of the tallies and of the results from the subdivision calculation, plus the increased computational costs of classifying primitives among oddly shaped regions (versus equal-sized regions). Altogether, the increased computational and communication overheads due to the MAHD algorithm are fairly small. Still, these factors need to be taken into account when comparing MAHD with other load-balancing strategies.

6 Load-Balance Study

6.1 Goals

A number of simulations were done in order to evaluate both static region assignment and adaptive region assignment using the MAHD algorithm. The main issue for the static method is how the number of regions per processor affects the load-balance and the system overhead. For the adaptive method, we are concerned with the mesh cell size and the use of the last frame's primitive distribution data versus the current frame's. Finally, we examine how the methods compare and how they scale with respect to the number of processors.

With respect to load-balance, two questions arise. First, how does one measure load-balance? Since a frame in a parallel graphics system is normally not displayed until all processors have finished their work, we will measure load-balance as the ratio of the maximum processor's load over the average load. Next, what is a good load-balance? Since load-balancing is only one interwoven factor towards achieving good performance, we cannot answer this question easily. Somewhat arbitrarily, we will consider a load-balance reasonable if the maximum/average load ratio is 1.5 or less.

6.2 Procedure

For these experiments, the simulator system used above to evaluate coherence was suitably modified. The set of assumptions that were made remains the same as in the coherence tests, with one exception. Off-screen primitives are sent away randomly after remaining off-screen for 3 frames. This makes a difference only for the Lobby case.

For these tests, the data that we are interested in is mainly the distribution of primitives across processors. For each recorded frame, we compute the minimum, maximum, and average number of primitive fragments per processor. We also compute the standard deviation of this figure. The frame-by-frame results are then averaged, and the resulting values are shown in the figures as MIN, MAX, AVG, and ST-DEV. Also, in order to consider the overhead associated with a method, we show the primitive traffic (the total number of primitives that need to be communicated) and the total number of primitives (which varies due to overlap) for each frame. These are labeled COM and TOT, respectively.

The following tests were performed for each application:

Static algorithm:

Processors	Region configuration
16	4x4 - 40x40 (1-100 regions/proc.)
64	8x8 - 64x64 (1-64 regions/proc.)

Adaptive algorithm:

Processors	Mesh configuration
16	16x16, 32x32, 64x64
64	16x16, 32x32, 64x64

Each run of the adaptive algorithm was performed twice, once using the previous frame's distribution data to determine the subdivisions, and once using the current frame's data. The tests are labeled PF and CF, respectively.

6.3 Results

Refer to graphs 1a-9a and 1b-9b found after the references.

6.4 Discussion

For the PLB and Sierra static cases, we can see that 9-25 regions/processor are required to achieve a maximum/average load ratio of 1.5 or less. This value varies according to the number of processors being used. The Lobby run is somewhat of a special case, since during much of the fly-through, the on-screen scenery is very simple. Its good load-balance with only 4 regions/processor is mainly due to the random distribution of off-screen polygons.

Because the number of regions per processor has a direct bearing on the size of each region and thus on the overlap factor, we naturally expect that increasing the number increases the overhead. As the (b) graphs show, both the amount of communication required as well as the number of primitive fragments that must be processed increase in direct relation to the number of regions into which the screen is divided. The increase is actually directly proportional to the total length of cuts made across the screen. Doubling the cut length (by increasing the number of regions per processor, say, from 4 to 16) approximately doubles the amount of communication as well as the number of additional primitive fragments in the system.

For the adaptive cases, we can see that the mesh-cell-size parameter has a significant effect on load-balance. Each halving of the mesh-cell dimensions results in nearly a halving of the primitive distribution standard deviation. The change in mesh size does not significantly alter the primitive communication overhead. Rather, increasing the number of mesh cells increases the algorithm overhead and its associated communication.

The graphs also reveal that using the previous frame's data results in no significant performance degredation. A small savings in the number of primitives to be communicated is shown by using the current frame's distribution data. One must remember that the applications tested here were fairly simple in nature. Further testing with different types of applications is desirable.

How do the methods compare? The static method requires 9 to 25 regions per processor to achieve the same load balance that the adaptive method can achieve with one region per processor. Thus the static method needs 3 to 5 times the communications bandwidth to take care of the additional primitive-shuffling overhead, plus additional processing capability to account for the increased number of primitive fragments. The tradeoff for this is that the static method has no overhead for any primitive distribution measurements or screen subdivision procedures. Additionally, the classification algorithm is simpler and the rasterization stage does not have to deal with varying-size regions. Finally, the processor-to-frame-buffer mapping is fixed for the static algorithm. The simplicity of the static approach may be worthwhile for a low-end system, where a small number of processors (and therefore regions) would not incur too much overhead. For a high-performance system, the adaptive algorithm appears to be the most reasonable choice.

We now consider scalability. Both methods have increased overhead as the number of processors increases, again due to the problem of dividing the screen into more regions. With regard to the static method, the graphs show that PLB does fine with 16 processors, while 64 is too many; the average number of primitive fragments doubles by the time a reasonable load-balance is reached. On the other hand, Sierra, a database nearly 3 times larger, does fine with up to 64 processors. Sierra does have smaller primitives (reducing its overlap-factor overhead), but this is typical of larger databases.

As for the adaptive method, both cases are fine up to 64 processors. Since overhead is due largely to the overall number of regions, we can look at the Sierra static test cases and suggest that the adaptive method can handle much larger numbers of processors, perhaps several hundred, before overhead becomes too large of a problem.

There are many application issues that determine the limits of how well the architecture scales. Any screen-subdivision architecture will suffer from increased overhead as the number of subdivisions increases. The amount of extra overhead is determined partly by the overlap factor and, in the case of sort-first, partly by the database dynamics.

To get around the scalability limitations caused by screen subdivision, one may consider a hybrid architecture with an extra level of parallelism. On the top level, the machine would be sort-first. For each region we replace the single processor with a parallel graphics system. Sort-last would be ideal for this purpose, as it does not have the screen subdivision problems that affect the other alternatives. Thus we introduce another architecture space to explore.

7. Graphics Database Management

Another set of major issues for sort-first is what form the display database should take, and how will it be managed? The assumption so far has been that the display database is a simple list of primitives, only modified by a viewing transformation matrix. While this may be adequate for some classes of applications, perhaps the majority of applications require support for object-based operations (i.e., manipulations of groups of primitives). Since most graphics systems provide this support through a hierarchical display database with instancing [10], we consider how to implement this solution on a sort-first system.

Aside from the distribution of the data structures, we also need to look at how the run-time operations on the data will be implemented. These operations include editing the database, traversing it for display, culling it to a particular view, and paging it to and from disk (for very large databases).

Because sort-first requires dynamic distribution of the display database, implementation of a hierarchical database requires one to take a different kind of approach than those offered for sort-middle or sort-last systems [8, 13]. If one attempts to dynamically distribute the entire hierarchy with the primitives, one rapidly runs into a bookkeeping nightmare, especially when one starts thinking about instancing.

Various approaches have been examined, and the one that seems to offer the most potential involves a separation of hierarchy structure from the primitives themselves. The hierarchy structure may be kept statically (i.e., non-migrating) on one or more processors, while the primitives themselves are free to migrate as usual. A system of tags is used to bind the primitives to the appropriate points in the hierarchy. The overhead of the primitive tags may be reduced by using tags for groups of related primitives rather than for each individual primitive.

This is only a start to the database management solution. There are many issues left to resolve, and resolving some of these requires further investigation into the nature of the applications' requirements of the graphics system. For instance, designing efficient support for database editing requires knowledge about the frequency of the various database editing operations.

8. Conclusion

Sort-first offers a powerful promise for interactive graphics applications. It is the only architecture that can deliver millions of rendered pixels for thousands of primitives in real-time without requiring phenomenal communication bandwidths or excessive duplication of hardware. In addition, it features reasonable scaling characteristics.

Yet for sort-first to follow through on these promises, many issues remain to be resolved. Issues that are simple on other architectures introduce complex new twists for sort-first. This research has begun the process of uncovering these issues and finding solutions for them.

Acknowledgements

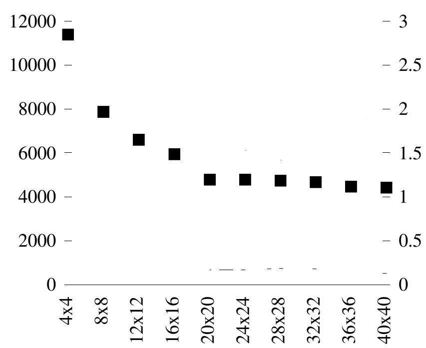
The author would like to thank Anselmo Lastra and David Ellsworth for their helpful suggestions towards this research and for reading drafts of this document. This work was supported by the Link Foundation, ARPA (ISTO Order No. A410), and NSF (Grant No. MIP-9306208).

References

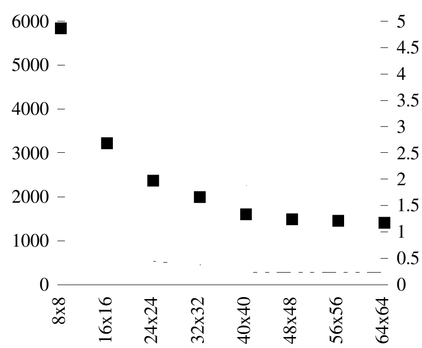
1. Akeley, Kurt. RealityEngine Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 109-116.
2. Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Commun. ACM 19*, 10 (October 1976), pp. 547-554.
3. Crockett, Thomas and Tobias Orloff. "A Parallel Rendering Algorithm for MIMD Architectures," Proceedings of the 1993 Parallel Rendering Symposium (San Jose, California, October 25-26, 1993), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1993, pp. 35-42.
4. Crow, Frank. Summed-Area Tables for Texture Mapping. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 23-27, 1984). In Computer Graphics 18, 3 (July 1984), pp. 207-212.
5. Cruz-Neira, Carolina, Daniel Sandin, and Thomas DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 135-142.
6. Deering, Michael and Scott Nelson. Leo: A System for Cost Effective 3D Shaded Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 101-108.
7. DeFoe, Douglas, Kaiser Electro-Optics, Inc., WWW URL http://esto.sysplan.com/ESTO/Displays/HMD-TDS/Factsheets/Immersion.html.
8. Ellsworth, David, Howard Good, and Brice Tebbs. "Distributing Display Lists on a Multicomputer," Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, Utah, March 25-28, 1990), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1990, pp. 147-155.
9. Ellsworth, David. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 33-40.
10. Foley, James, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*, 2nd Ed., Addison-Wesley, Reading, Mass., 1990.
11. Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steven Molnar, Greg Turk, Brice Tebbs, Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Proceedings of SIGGRAPH '89 (Boston, Massachusetts, July 31-August 4, 1989). In Computer Graphics 23, 3 (1989), pp. 79-88.
12. LaCroix, Michel and James Melzer. Helmet-Mounted Displays for Flight Simulators. *Proceedings of the 1994 Image VII Conference*, June 1994, pp. 34-40.
13. Molnar, Steven. *Image-Composition Architectures for Real-Time Image Generation*. Ph.D. dissertation, TR-91-046, University of North Carolina at Chapel Hill, 1991.
14. Molnar, Steven, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992) In Computer Graphics 26, 2 (1992), pp. 231-240.
15. Molnar, Steven, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 23-32.
16. National Computer Graphics Association Picture-Level Benchmark, *GPC Quarterly Report* 2, 4 (1992).
17. Roble, Douglas. A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube. *Proceedings of Pixim '88*, Paris, France, October 1988, pp. 177-192.
18. Sutherland, Ivan, Robert Sproull, and Robert Schumacker. "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys* 6, 1 (March 1974), pp. 1-55.
19. Whelan, Daniel. *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, Ph.D. dissertation, California Institute of Technology, 1985.
20. Whitman, Scott. *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Wellesley, Massachusetts, 1992.
21. Whitman, Scott. Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 41-48.

Database: PLB Head

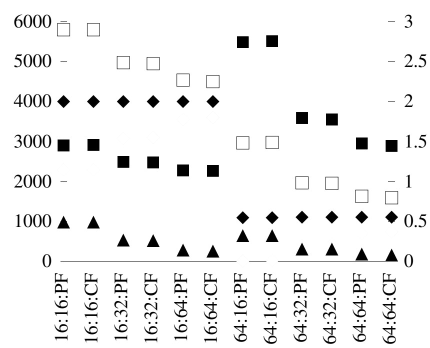
1a. Static Load-Balancing: 16 Processors



2a. Static Load-Balancing: 64 Processors

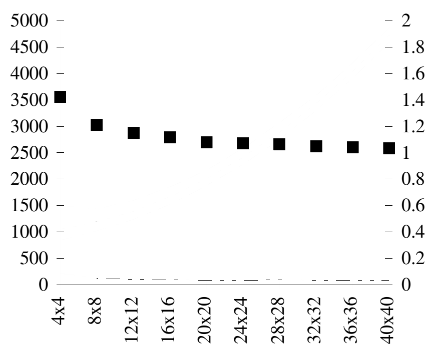


3a. Adaptive Load Balancing

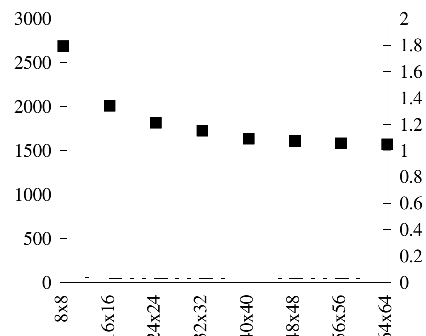


Database: Lobby

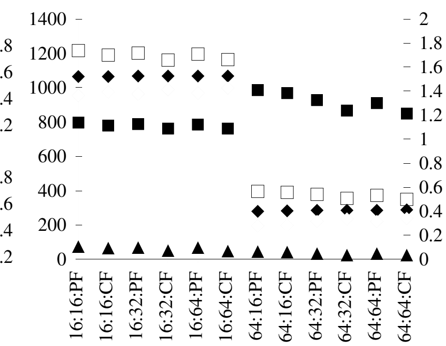
4a. Static Load-Balancing, 16 Processors



5a. Static Load-Balancing: 64 Processors

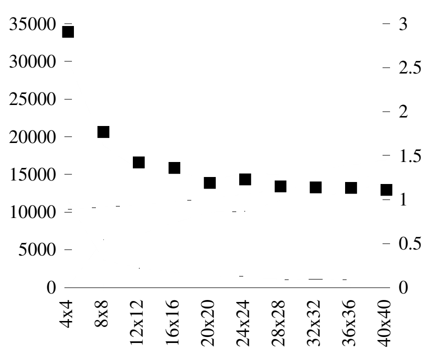


6a. Adaptive Load Balancing

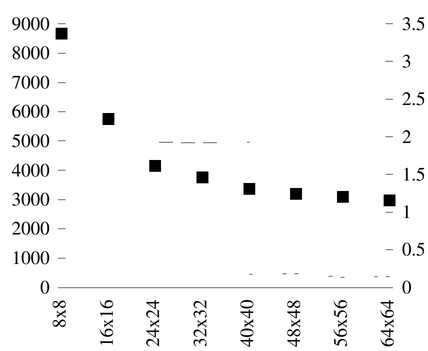


Database: Sierra

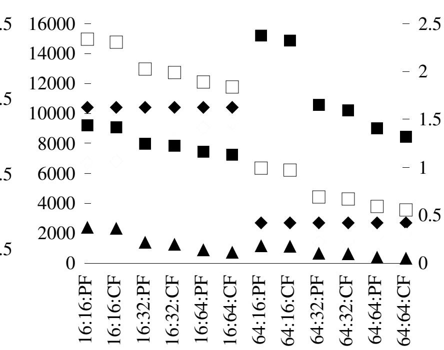
7a. Static Load-Balancing: 16 Processors

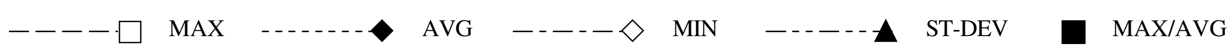


8a. Static Load-Balancing: 64 Processors



9a. Adaptive Load-Balancing





Left side scale = number of primitives Right side scale = MAX/AVG ratio

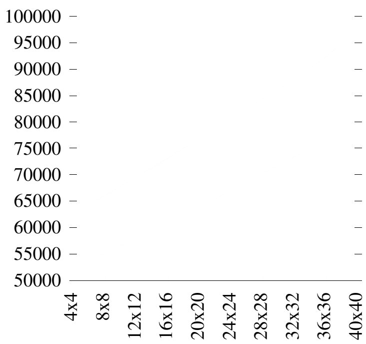
Static L.B. legend = number of regions (yielding 1, 4, 9, 16, 25, 36, 49, 64, 81, or 100 regions per processor)

Adaptive L.B. legend = number of processors **:** mesh dimension **:** use Previous Frame's or Current Frame's distribution data

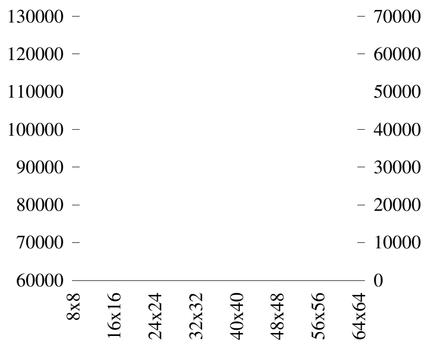
Graphs 1a - 9a. The graphs above show various statistics averaged over the frames for each of the test runs. The statistics are the maximum (MAX), average (AVG), and minimum (MIN) numbers of primitive fragments per processor, the standard deviation (ST-DEV) of these values, and the MAX/AVG ratio, a figure which provides an indication of the success of the load-balancing method.

Database: PLB Head

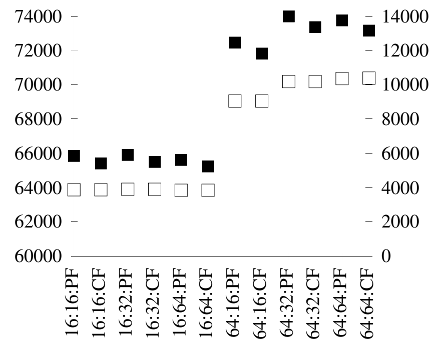
1b. Static Load-Balancing: 16 Processors



2b. Static Load-Balancing: 64 Processors

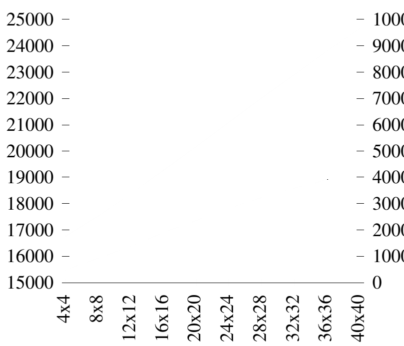


3b. Adaptive Load-Balancing

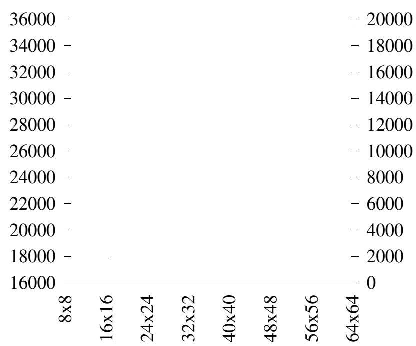


Database: Lobby

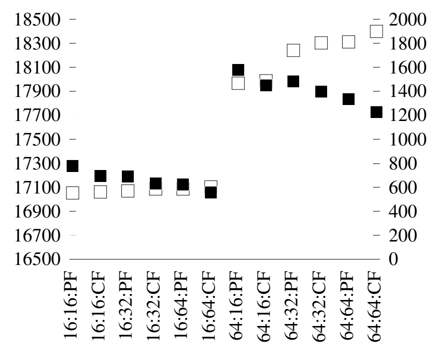
4b. Static Load Balancing, 16 Processors



5b. Static Load-Balancing: 64 Processors

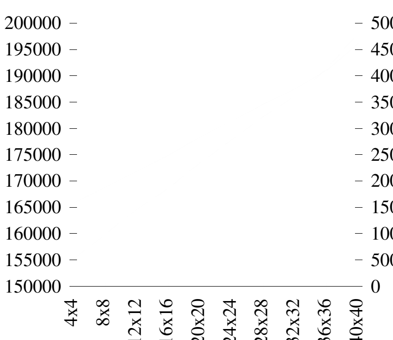


6b. Adaptive Load Balancing

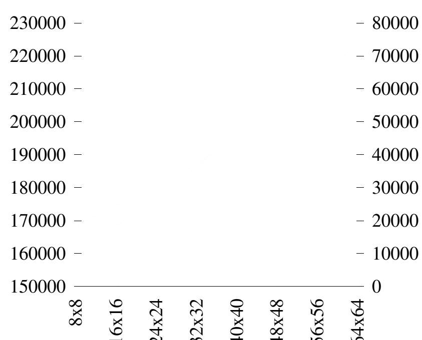


Database: Sierra

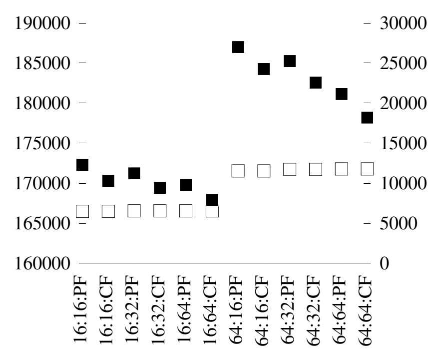
7b. Static Load-Balancing: 16 Processors



8b. Static Load-Balancing: 64 Processors



9b. Adaptive Load-Balancing



TOT COM

Left side scale = Total number of primitives Right side scale = Number of primitives communicated

Static L.B. legend = number of regions (yielding 1, 4, 9, 16, 25, 36, 49, 64, 81, or 100 regions per processor)

Adaptive L.B. legend = number of processors **:** mesh dimension **:** use Previous Frame's or Current Frame's distribution data

Graphs 1b - 9b. The graphs above show various statistics averaged over the frames for each of the test runs. The statistics are the total number of primitives (TOT) in the system (which increases as primitives overlap more regions) and the total number of primitives that must be communicated (COM) each frame for proper sorting.

Plate 1. The model for the "PLB Head" test case (59,592 polygons). As in the National Computer Graphics Association Graphics Performance Committee's Program Level Benchmark, the model rotates 360 degrees about a vertical axis in 4.5 degree increments. The model is courtesy of the IBM AWD Graphics Lab, Austin, Texas.

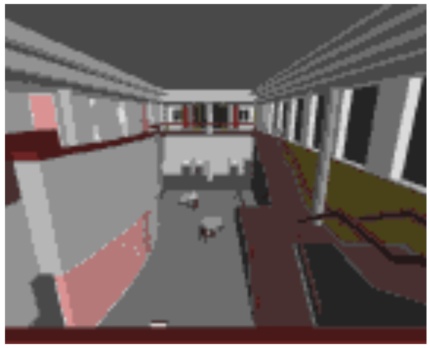


Plate 2. The lobby of UNC's Sitterson Hall (16,267 polygons). The viewer turns toward the right and proceeds down the steps towards the far table, then turns around to face the starting point. The model is courtesy of the UNC Building Walkthrough project, F. P. Brooks, Principal Investigator.

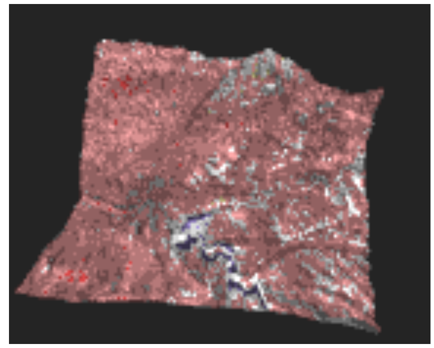


Plate 3. A section of the Sierra Nevada mountains (162,690 polygons). The model undergoes a series of zooms, rotations, and translations, with a reset between each sequence. The model is courtesy of H. Towles, Sun Microsystems.