Distributing Display Lists on a Multicomputer[1]

David Ellsworth, Howard Good, and Brice Tebbs

Department of Computer Science
University of North Carolina, Chapel Hill

Abstract

We have developed techniques for distributing a hierarchical display list from a PHIGS-like library across a multicomputer. By storing a portion of the database at each processor, interprocessor communication is reduced. This reduction promises traversal of the display list at rates supporting rendering speeds of one million polygons/second or more, as we hope to achieve on our new machine, Pixel-Planes 5 (under construction). Our distribution techniques support order-dependent primitives and allow general display list editing.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures] Multiprocessors–MIMD processors; C.2.4 [Computer-Communication Networks] Distributed Systems–Distributed applications; I.3.2 [Computer Graphics] Graphics Systems–Distributed Graphics; I.3.4 [Computer Graphics] Graphics Utilities–Graphics Packages; PHIGS; I.3.5 [Computer Graphics] Computational Geometry and Object Modeling–Hierarchy and geometric transformations.

Additional Key Words and Phrases: display list, structure network, multicomputer, order dependent primitive.

[1]This work was supported by the Defense Advanced Research Projects Agency, DARPA ISTO Order No. 6090, the National Science Foundation, Grant No. DCI-8601152, and the Office of Naval Research, Contract No. N0014-86-K-0680.

1. Introduction

Most of the work in parallelizing graphics systems has concentrated on the rasterization, or back-end, part of the traditional graphics pipeline [Fuchs 77, Parke 80]. Considerably less attention has been given to parallelizing the traversal and transformation, or front-end, part of the pipeline. Increasingly sophisticated methods of distributing front-end calculations over multiple processors will be needed as the desired real-time speeds of graphics machines increase from the current 100-200 thousand triangles/second to 1 million triangles/second and beyond.

Current designs for multiprocessor front-ends include vector processor [Apgar 88], pipeline [Akeley 89], shared memory [Borden 89], and MIMD [Torborg 87] architectures. While these architectures achieve high performance, they all have limitations in their ability to scale to larger numbers of processors. Vector processors with longer vector lengths achieve little speedup on display lists with arbitrary sized elements. Pipelined multiprocessor architectures are difficult to scale because their performance is limited by the slowest stage of the pipeline, and partitioning tasks evenly on pipelines with many processors is very difficult.

A more important limitation is that all of these architectures perform a single-threaded, or serial, traversal of the database. Serial traversal at rates high enough to sustain 1 million polygons per second is difficult, and would today likely require specialized traversal hardware, limiting possible graphics algorithms [Foley 90]. Storing the entire database in a single memory subsystem for a serial traversal would require extremely high bandwidth from the memory to the parallel transformation units. As performance levels increase to 10 million polygons per second, we believe serial traversal will no longer be practical.

One solution to the scalability problem is to use a distributed-memory MIMD, or multicomputer, architecture [Athas 88]. Such a system could perform both traversal and transformation in parallel. The system will scale well as long as required interprocessor communication bandwidth is kept at a reasonable level. This solution does not require specialized hardware, as traversal and transformation can be done on the same general-purpose processors.

1.1 Pixel-Planes 5

Pixel-Planes 5, a high performance graphics system being being built at UNC, is a heterogeneous multicomputer [Fuchs 89]. The Pixel-Planes 5 system consists of a host workstation, nominally 16 Intel i860-based Graphics Processors (GPs), nominally 16 SIMD processor arrays called Renderers, and a frame buffer, all of which are

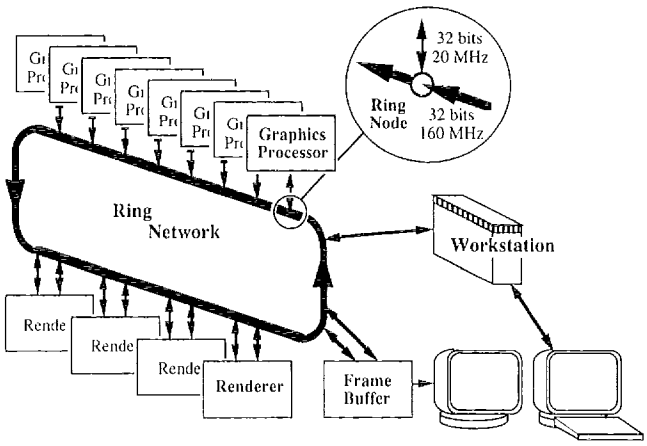

Figure 1. Pixel-Planes 5 Block Diagram

mounted on a high bandwidth (640 MByte/sec) ring network (see Figure 1). The host computer provides standard UNIX system services such as file I/O. Each Graphics Processor is a general-purpose floating-point processor with its own memory. The Renderers perform rasterization by using using their 128x128 array of 1-bit processors and a quadratic expression evaluator tree.

Pixel-Planes 5 (*Pxpl5*) is suited to a wide variety of graphics algorithms, including volume rendering and CSG [Fuchs 89]. In this paper, however, we concentrate on *Pxpl5* in a traditional display list rendering mode. When running in this mode, the application runs on the workstation host. Since the host is a conventional UNIX workstation, it has significantly less computing power and I/O bandwidth than the GPs. For this reason we maintain the database on the GPs to minimize the amount of data that the host must send to the GPs every frame.

The basic display list rendering process on *Pxpl5* proceeds as follows: the GPs traverse the database, transform the polygons into screen space, and compute coefficients for a set of linear expressions that are used to scan-convert each polygon. These linear expressions are evaluated by the SIMD arrays on the Renderer boards. The Renderer boards then compute the image, each handling different regions of the screen. Finally, the screen sub-images are collected in the frame buffer and displayed.

1.2 PPHIGS

We have implemented a variant of the PHIGS+ standard [van Dam 88] in a graphics library called PPHIGS (Pixel-Planes Hierarchical Interactive Graphics System). Like PHIGS+, the PPHIGS database consists of a network of structures. Each structure contains structure elements that are either graphics *primitives* such as lines and polygons, *attributes* such as colors and matrix operations, or *executes* (calls) to instance other structures. PPHIGS implements a subset of the PHIGS+ collection of attributes and primitives. A sample PPHIGS structure network is given in Figure 2.

PPHIGS, like any hierarchical graphics library, puts several constraints on any database distribution algorithm: First, in PPHIGS, attributes such as colors or matrix operations can be inherited from parent structures. Second, PPHIGS allows the user to perform general editing of the structure network. Finally, PPHIGS has some graphics primitives (particularly 2-D ones) that must be rendered in a particular order.

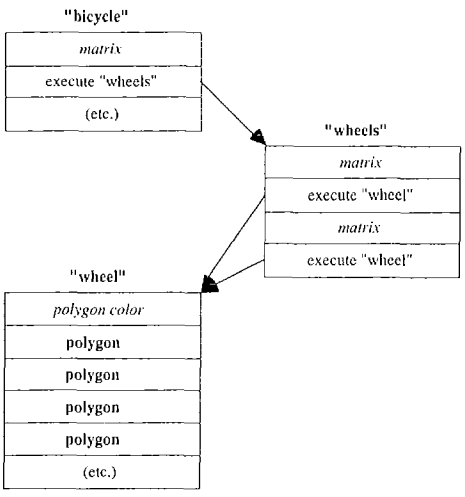

Figure 2. Sample PPHIGS structure network.

1.3 Application Mix

PPHIGS-based applications can be characterized in two ways: 1) database organization and 2) editing requirements. The database organization is the arrangement and number of structures in the network. The editing requirements are characterized by the frequency and types of changes that must be done to the structure network.

We have analyzed applications that run on our current graphics system, Pixel-Planes 4, including molecular graphics, architectural walkthrough, and head-mounted display research. We have found that the majority of the applications have databases with relatively few structures, each structure having a large number of polygons, and that the database editing that is done each frame usually consists of changing only a few transformation matrices. Nearly all of the current applications are interactive applications.

We expect that our application mix for Pixel-Planes 5 will include similar applications, as well as new ones made possible by the increase in performance. When the number of available polygons increases with *Pxpl5*, we expect that the number of structures and transformations will increase more slowly than the number of polygons. This will further increase the number of polygons per structure without significantly increasing the amount of editing that must be done each frame.

The rest of the paper describes techniques we have developed for distributing an application's structure networks evenly on a multicomputer. The techniques presented have been implemented on the Pixel-Planes 5 system simulator and have been tested on several different databases. Although the techniques were developed for Pixel-Planes 5 and PPHIGS, we expect them to be applicable to distributing any type of hierarchical display list on any multicomputer.

2. Display List Distribution

Given the expected application mix for *Pxpl5*, a display list distribution method should have the following goals:

- The processors' loads should be balanced: each processor should take the same amount of time to transform its portion of the display list.
- As the display list and the viewing position is modified, the processors' loads should remain balanced without redistribution of the display list. This allows the time to draw a frame to be consistent, which is important for interactive applications.
- The amount of duplicated work should be minimized, thus allowing the system to be scaled.

To meet these goals, we have investigated several different approaches. The two most promising approaches, distributing by structure and distributing by primitive, are discussed below.

2.1 Distribute by Structure

One method of distributing the display list is to assign instances of structures to processors so that the loads are balanced. If the number of structure instances is less than the number of GPs, the structures can be broken up.

Before traversing a structure, a processor must have that structure's inherited attributes. One way of computing this is to maintain a

skeleton on each processor of the network from the local structure back to the root of the structure network. The skeleton contains only the attribute elements of the ancestor structures. The inherited attributes can be computed by traversing the ancestor structures. This requires some duplication of computation and additional overhead to manage the network skeleton when the network is altered. Another way is to maintain a skeleton of the entire structure network on a single processor (perhaps the host), and then pre-calculate the inherited attributes, sending those to the appropriate processors. This would require considerable processing during editing to keep the inherited attributes on the processors up to date.

A more important problem is that the processor workloads do not remain balanced when objects are clipped or are added or subtracted from the display list. If the load becomes unbalanced, redistribution will be necessary to balance the workloads. This would either hold up the transformation process, thereby affecting the frame rate, or it would require resources dedicated to balancing the workloads.

2.2 Distribute by Primitive

Using this method, the primitives of each structure are divided equally among the processors. This division is done primitive-by-primitive so that successive primitives (polygons, spheres, etc.) are generally placed on different processors. Attributes are sent to all processors to insure that each processor has the correct attribute values during traversal. All processors are given structure execute primitives so each can traverse the structure network.

This method does a better job of insuring that the load remains closely balanced. When a structure is added or removed from the structure network, the loads remain balanced because the individual structure's load is balanced among the processors. When part of the structure is off screen, the load will remain relatively balanced if the clipped primitives have been sent to different processors. This should be the case since successive primitives in a structure are often near each other in space.

Unfortunately, duplicating attributes and structure executes on each processor adds work as compared to a single processor traversal. For many models this is not a serious problem since there are many more primitives than attributes and many primitives in a structure. A second problem is that an imbalance may be created if small structures which aren't perfectly balanced are instanced many times.

For example, consider a cube structure, distributed among 8 processors, that is instanced 100 times. The the distributed structure on the first 6 processors would each contain one face of the cube; the structure on the last two processors would be empty. The 100 instance calls would go to all the processors. The result is that 6 processors would each traverse 100 cube faces while the last 2 processors execute empty structures.

2.3 Our Implementation

We have chosen to implement a variation of the distribute-by-primitive method. In our implementation we have chosen to keep a global copy of the entire database, not just a skeleton, on the host. This simplifies editing, and makes display list inquiry and disk archival faster since no communication with the processors is required. In the rest of Section 2 we describe what we've done to deal with the simple distribute-by-primitive method's shortcomings.

2.3.1 Distributing Attributes

The main problem with the simple distribute-by-primitive approach

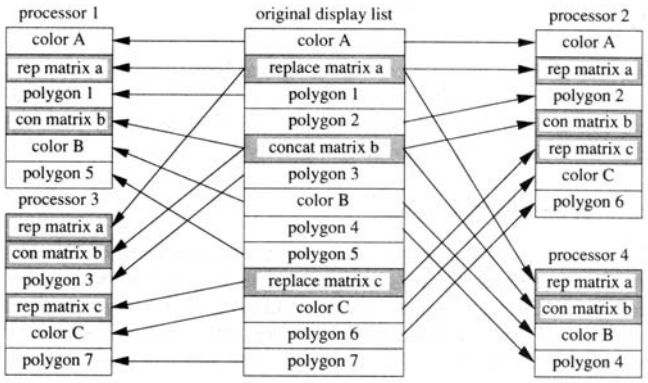

Figure 3. Example of distributing a single structure across four processors.

is that each attribute is duplicated on all the processors. This means that some processor cycles and memory space are wasted on redundant attributes. As systems are scaled up to more processors the problem worsens, because the number of redundant attributes increase at a one-per-processor rate while the number of primitives remains constant. We ameliorate the problem by having the host send each attribute only to those processors on which it is required, i.e. only to those processors that have primitives affected by that attribute (see Figure 3).

In order for the host to know all the current attributes and those that have been sent to each processor, it maintains an *attribute state* for the global display list and for the display list on each processor. Each attribute state records, for a given point in the display list, the colors and transformation matrices that would be active for each type of primitive during traversal of the display list at that point.

The host initially reads in each structure from the application and distributes it element by element. As each primitive is added and assigned to a processor, the attribute state of that processor is checked to make sure that it is current, *i.e.* it matches the global one. If not, the missing attributes are sent as well as the primitive. As each attribute is added, the global attribute state is updated, but the attribute is not immediately sent to any processor.

Some attributes, such as concatenate-mode transformation matrices (which concatenate with rather than replace the current transformation matrix), cannot be distributed in this way because they have a cumulative effect on the structure. These are distributed to all processors as soon as encountered, along with any pending replace-mode matrices.

While this is fairly simple to implement for initially distributing a structure from start to finish, it is more difficult to implement for arbitrary editing of structures, as discussed in Section 3.

2.3.2 Distributing Primitives

Our implementation improves on the simple method of sending a primitive at a time to the most lightly loaded processor by sending clusters of primitives rather than single ones. This has the advantage that attributes affecting a small number of primitives are sent to fewer processors, since the affected primitives will be distributed among fewer processors. Clustering is especially useful for structures where attributes change relatively often, i.e. between every few primitives. The parameter controlling the amount of clustering allowed is based on the number of processors. Note that while clustering allows sending attributes to fewer processors, it also introduces more spatial coherence in each structure on each processor. This could introduce

load imbalances similar to the ones encountered when using the distribute-by-structure method.

We plan to break up large primitives such as long triangle strips into smaller pieces and distribute them among the processors. This would prevent the loads from becoming seriously unbalanced if such an element is deleted or is moved completely off screen.

2.3.3 Primitive Structures

To avoid creating a load imbalance when small structures are instanced multiple times, a graphics system could automatically detect small, often instanced structures and identify them as *primitive structures*. Primitive structures can be created and edited like structures but are distributed like primitives. The entire primitive structure is broadcast to all the processors, but each instance is sent to only one processor. Instances of primitive structures can then be balanced in the same way as pre-defined primitives. Primitive structures can contain any primitives or attributes and can execute other primitive structures. Primitive structures cannot execute ordinary structures, because only a portion of the ordinary structure would be on the same processor as the the primitive structure instance.

We have not implemented automatic detection of primitive structures, as it is difficult to switch between structure types as a structure is edited and the structure size changes. Instead, our system has the user designate which structures should be treated as primitive structures.

2.3.4 Weighting Structure Elements

Implicit in the discussion of balancing structure elements among processors is the relative "weight" of each element. Primitive and attribute weights must be known before an application is run so that databases can be balanced as they are loaded. We have determined the *nominal weight* of each element by calculating the usual processor time required to transform that element. Other criteria, such as memory usage, could be used for other systems. An element's nominal weight is changed only when its usual processing time changes, *e.g.* when its transformation code is changed.

3. Structure Editing

Interactive applications need to perform general editing of structures. All editing tasks must ensure that each attribute is sent to every processor that requires it and should preserve processor load balancing. Distributing attributes correctly during editing is straightforward using the simple distribute by primitive method in which each attribute is distributed to all the processors: the host simply sends the attributes to all processors, and sends primitives to the most lightly loaded processor. Unfortunately, as previously mentioned, the simple method can result in greatly reduced distribution efficiency (see Results).

PPHIGS allows four types of structure editing operations:

modify: Replace structure element with a new one of the same type (not standard PHIGS+).
append: Append element to the end of a structure.
insert: Insert an element at an arbitrary point in a structure.
delete: Delete an element at an arbitrary point in a structure.

The first two tasks are simple and can be performed very quickly. Insert and delete, however, require some analysis of the structure to determine the current and processor attribute states at the point being edited.

3.1 Modify Operation

The **modify** operation is the simplest editing task, requiring just that the new data be sent to the processor(s) that have the old data. Since it is assumed that the old element was distributed correctly, no analysis of the rest of the structure is required. The modify operation is sufficient for many application tasks such as updating transformation matrices and colors, and is the most heavily used in our current applications.

3.2 Append Operation

The **append** operation requires knowing the current global attribute state as well each processor's attribute state at the end of the edited structure. We call the set of these attribute states the total attribute state. Append requires no structure analysis, since the attribute states at the end of each structure are saved with the structure descriptor. Once the attribute states are known, elements can be appended in the same way as when initially distributing a structure (see Section 2.3.1).

3.3 Insert and Delete Operations

The **insertion** and **deletion** operations also require knowing the total attribute state at an arbitrary point in a structure, and can affect distribution of elements both before and after the edit point. Determining the attribute states and distributing attributes correctly for each insert and delete would make each operation expensive. Instead, edits are performed on the host copy of the structure network and then propagated to the processors when editing in a particular region of a structure is completed. Thus, insert and delete editing is done in two steps:

Step 1: A series of insertion and/or deletions is done to the host display list. Insertion of primitives, and deletion of attributes and primitives, are propagated to the processors immediately. Inserted attributes only appear in the host display list.

Step 2: The attribute states are acquired for the first point in the structure affected during step 1. Then the global display list is traversed from that point until the last point affected and attributes are distributed to those processors which require them, as is done when appending to a structure. After the traversal, the attribute state at the last affected point is used to send attributes to the processors that do not have the current attributes.

3.4 Calculating the Attribute States

For step 2 we must acquire the total attribute state at an arbitrary reference point. Saving the attribute states at every point in the structure where editing operations could take place would take far too much memory. Instead, we determine the entire attribute states on the fly by stepping backwards through the structure and examining elements sent to each processor. This requires looking at at least one attribute of each type sent to each processor and could involve stepping back through the entire structure. To reduce the number of steps needed, a limited form of caching can be used. It is possible either to save attribute states every n primitives, or to save a set of attribute states in a cache to advantage of locality of reference during editing.

For step 2 we also must propagate the attributes active at the last referenced edit point to all processors that have at least one primitive affected by the edit. This could involve stepping forward through the rest of the structure to check elements.

3.5 Bounding Insert and Delete Operations

Calculating the attribute states as described above could require checking all the elements in a structure. Instead, we don't explicitly calculate the attribute states, but only determine which processors have received the current attribute state. This involves stepping backwards through the structure as before, but now every time a primitive is encountered, we flag the attribute state for that processor as current, because it was current when the primitive was first distributed.

This method requires us to step back through the structure only until one primitive for each processor is encountered. Using this method and the distribute by primitive algorithm described in Section 2, acquiring the attribute states will require examining $O(n)$ elements, where n is the number of processors. This is because only a certain number of primitives can be sent to one processor before its load becomes too heavy and primitives are sent to the next processor. That number is $c*(maxw/minw)$, where c is the primitive clustering factor and $maxw$ and $minw$ are the maximum and minimum primitive weights. The maximum number of elements that can be examined without finding one on each of n processors is therefore $c*(maxw/minw)*(n-1)$, because at that point $n-1$ processors are maximally loaded and the next primitive would have gone to the nth processor.

This method is complicated by the fact that some attributes, such as a sphere color, only affect certain types of primitives. Encountering a primitive of a different type, such as a polygon, in the structure would not guarantee that the attribute state at that point was current for all primitive types. In order for the previous bound to hold, we distribute attributes as if each attribute affects all succeeding primitives, regardless of their type. This ensures that once any primitive is sent to a processor, the attribute state for that processor is correct for all primitives. This strategy can save large amounts of structure traversal. However, it comes at a cost of sometimes sending unnecessary attributes, adding duplicate work. We expect the number of these unnecessary attributes to be very small.

4. Order-Dependent Primitives

PPHIGS allows certain primitives, such as 2D polygons, to be displayed in the order that they are encountered during the display list traversal. These are called order-dependent primitives, or ODPs. These primitives are displayed in front of the 3D z-buffered primitives to allow for overlays and annotation. A parallel implementation of PPHIGS could synchronize the output of the transformation processors so the rasterization unit receives primitives in the correct order, such as done in [Torborg 87] and [Borden 89]. However, this synchronization would add overhead and require a serial step to the rendering process, which would reduce the degree of parallelization in the system.

We preserve the effect of rendering order by using a variant of the z-buffer algorithm. During the transformation process, priority numbers are assigned to each primitive. As each primitive is rendered, we use a "priority buffer" to determine which primitive should be visible at each pixel. This allows us to get the effect of a "painter's algorithm" by using a z-buffer type approach. To use such an approach, the z-buffer must have enough resolution to hold both the 3D primitives' z values and the order-dependent primitives' priority values. A simple implementation would use one bit of the z-buffer to differentiate between the z values used for 3D and order dependent primitives.

In a single processor system the priority numbers can be assigned by sequentially numbering the ODPs as they are encountered while traversing the display list. Because a single processor in a multicomputer has only a part of the distributed display list, it cannot simply number the primitives since it doesn't know how many primitives are on the other processors We solve this problem by recording, for each primitive, a "delta" value: the difference in priority numbers between that ODP and the previous ODP on the same processor. This delta value is one more than the number of "missing" ODPs (those on other processors) between the current and previous ODPs. We also record, for each structure, the difference in priority numbers between the last ODP and the end of the structure. With this information, a processor can independently assign priority numbers to its portion of the display list (see Figure 4). These delta values are calculated when distributing each structure. After editing an ODP, some of the deltas must be updated; this can be done during the same traversal as when the attributes are distributed to the correct processors (see section 3.3).

5. Display List Rebalancing

Although we expect that our distribution techniques will keep the processors' workloads closely balanced, it is possible for the workloads to become imbalanced. We have been investigating the ways this may occur as well as methods for dynamically re-balancing the load. Since we have not characterized how the workloads become imbalanced, this work is preliminary.

We have found two major causes of workload imbalance: imperfect distribution and invalidated weights. Each requires different techniques for eliminating the imbalances.

5.1 Imperfect Distribution

When distributing each structure across the processors it is almost always impossible to balance the workload perfectly. These slight imbalances, randomly distributed across the processors, could add up to a large imbalance when one structure is instanced several times.

To detect the extent of imperfect distribution, each processor traverses its portion of the display list and adds up the nominal weights of its structure elements. This computation could be done as part of a normal traversal. The host uses these sums to detect an imperfect distribution. To fix the imbalance, the host moves randomly picked primitives from heavily loaded to lightly loaded processors using procedures based on the editing sequences described above.

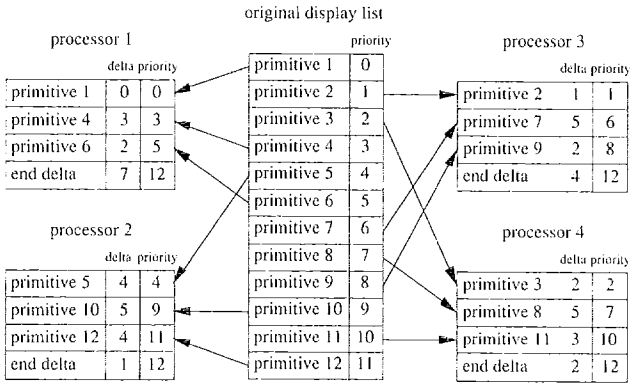

Figure 4. Example of assigning priorities to a structure of ODPs. A primitive's priority number is determined by adding its delta to the previous primitive's priority.

5.2 Invalidated Weights

The second cause of imbalance is the fact that the nominal weights of the structure elements (the ones used for the distribution) are not always equal to the actual weight of the elements. For example, this can happen when a polygon is backface culled, is off screen, or is clipped to the viewing volume. Because these weight changes have spatial coherence, in general there should only be large imbalances when the database is distributed in a spatially coherent fashion. While this is usually avoided in the primitive by primitive distribution scheme, it can occur in some cases One example of this is when a 16x16 quadrilateral mesh is distributed across 16 processors: each processor will have a strip of the mesh.

This cause of imbalance can be distinguished from imperfect distribution when the imperfect distribution test described above indicates that the database is distributed correctly according to the nominal weights. To correct the imbalance, the host must request that the processors traverse their display lists and find primitives with actual weights different from the nominal weights. The overloaded processors find primitives with actual weights larger than nominal weights, and lightly loaded processors find primitives with actual weights less than the nominal weights. Then, the host exchanges primitives on different processors that have the same nominal weights but that have actual weights that balance the processor load.

Both redistribution methods should only be used when the imbalance is significant (say >10% of the total transformation time) and remains for several frames. If this is not done, then the rebalancing algorithm will make fairly expensive adjustments every frame to correct the normal slight imbalance. For the same reason, and also to avoid a 'hiccup' when the database is rebalanced, a limited amount of the total imbalance should be corrected each frame. The amount to redistribute per frame should be chosen so that it can be done within the current frame time.

6. Results

We have simulated the two distribute-by-primitive algorithms: the simple one described in section 2.2 and the more complex one described in section 2.3. We have calculated their distribution efficiency and load balancing for four hierarchical databases. The distribution efficiency is the percentage of non-redundant work performed by the processors, assuming perfect load balancing. The processor utilization percentages show the quality of the load balancing. The formulas used are:

$$\text{processor utilization} = \frac{\sum_{\text{processors}} \text{processor time}}{\text{number processors} \cdot \underset{\text{processors}}{\text{MAX}} \text{ processor time}}$$

$$\text{distribution efficiency} = \frac{\text{single processor time}}{\sum_{\text{processors}} \text{processor time}}$$

The overall speedup is given by the number of processors multiplied by both percentages. The databases are illustrated in figures 6-8.

The space station database is the simplest case, namely a large number of primitives with no hierarchy. The building lobby is similar, but 55% of the primitives are off-screen and thus have invalidated weights, as the time to transform each such polygon is a fraction of the time indicated by its nominal weight. However, the processor utilization remains high because our method distributes the on-screen polygons evenly across the processors. The flock consists of 144 "boids" (bird-oids) [Reynolds 87] in flight about the Old Well. The boids are defined by a primitive structure containing 5 polygons and 4 colors. A separate flock structure instances the primitive structure 144 times, with a different transformation matrix each time. The dramatic increase in distribution efficiency for the complex algorithm reflects both the use of primitive structures and the breakdown of the simple attribute distribution method. Finally, the human figures are two structures with deep hierarchies (8 levels of nesting), which are balanced well, but which require a considerable amount of redundant work among the processors. The poor distribution efficiency occurs because the individual structures contain few polygons: extra work results from flushing the attribute state to all the processors before each new structure is instanced. We are exploring ways to efficiently analyze paths through the hierarchy so that only necessary attributes are propagated at each structure instance.

All results are from the Pixel-Planes 5 software simulator, where each processor, SIMD rasterizer, or device is simulated with as an separate Unix process. While the simulator accurately simulates the effect of C code running on the host and Graphics Processors, it does not give timing information about the transformation process for the i860-based GPs. All element transformation values for Pixel-Planes 5 are estimated based on performance of the Pixel-Planes 4 system.

7. Conclusions

As front-end computing power requirements continue to increase, multicomputer graphic systems will become more common since the architecture can be expanded without requiring higher memory system performance or extensive communication. We have developed an initial solution to distributing hierarchical display lists across a multicomputer that handles many databases with reasonable efficiency and load balancing. While we feel that our solution will be effective for most interactive situations, there is still more work to be done, particularly in finding an algorithm that satisfies our distribution goals without requiring a complicated editing procedure.

8. Acknowledgements

We wish to thank the Pixel-Planes project principle investigators Henry Fuchs and John Poulton for their many helpful comments, and our colleagues on the Pixel-Planes 5 software team, Michael Bajura, Andrew Bell, Jonathan Leech, Ulrich Neumann, John Rhoades, and Greg Turk, for helping design and implement the Pxpl5 simulator. We would like to thank the members of the Walkthrough project [NSF Grant #CCR-8609588], Frederick P. Brooks Jr.-principal investigator, John Airey, Randy Brown, Penny Rheingans, and Dana Smith for the lobby database. We would also like to thank to Don E. Eyles for the space station database, Andy Skinner for the flock simulator database, and Young Harvill of VPL Research for the human figures database which he made using Paracomp's Swivel 3D modeling package.

Database	Hierarchy Depth/ Structure Count	Average Polygons per Structure	Polygon/ Attribute Ratio	Distribution Efficiency				Processor Utilization (complex distribution algorithm)		Overall Speedup (complex distribution algorithm)	
				Simple Algorithm		Complex Algorithm					
				4 GPs	16 GPs	4 GPs	16 GPs	4 GPs	16 GPs	4 GPs	16 GPs
space station	1/1	3388	21.6	98.3%	92.2%	99.5%	98.8%	99.9%	99.1%	3.97	15.66
building lobby	1/1	3923	7.8	93.1%	73.0%	98.3%	96.2%	98.3%	93.4%	3.86	14.37
flock and old well	2/147	10.1	1.8	67.9%	29.8%	98.4%	94.8%	99.3%	96.4%	3.91	14.62
two human figures	9/91	72.1	36.1	83.1%	49.7%	85.8%	57.8%	85.8%	90.1%	2.94	8.33

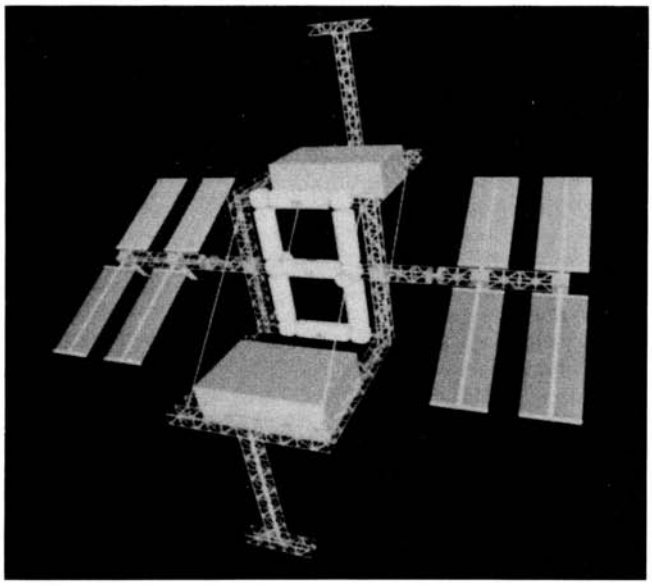

Figure 4. Space station database.

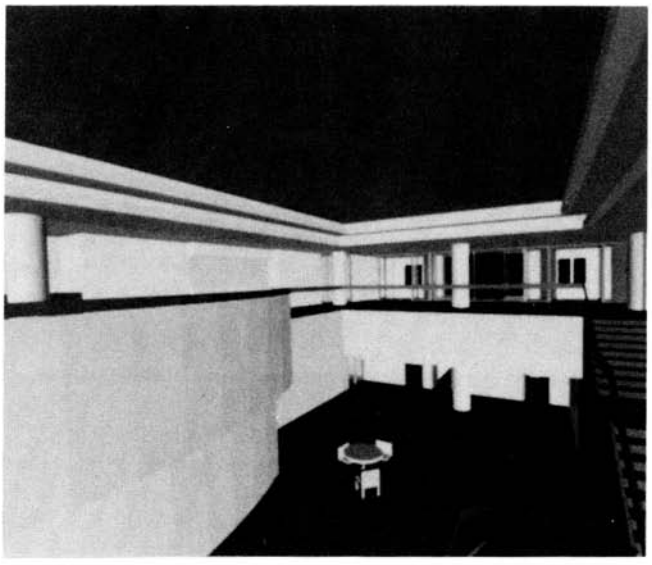

Figure 5. Building lobby database.

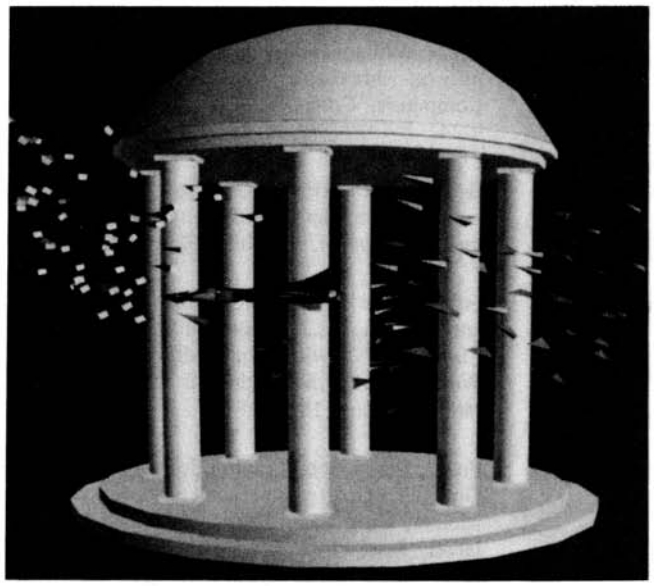

Figure 6. Flock and old well database.

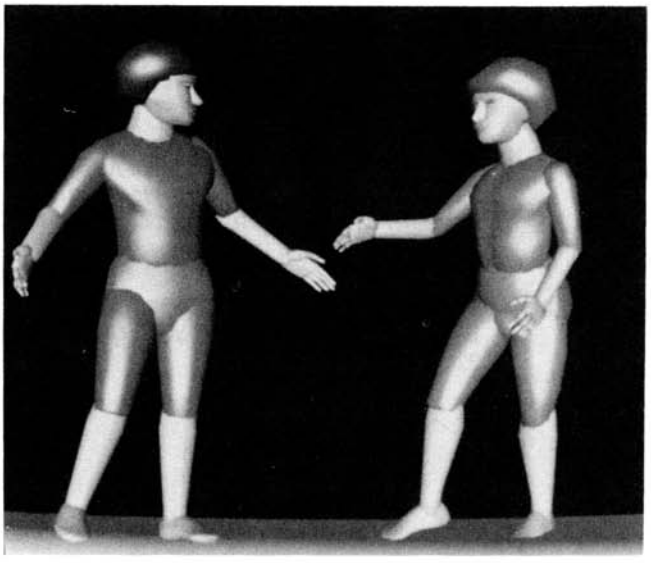

Figure 7. Human figures database.

9. References

[Akeley 89] Akeley, Kurt, "The Silicon Graphics 4D/240GTX Superworkstation", *IEEE Computer Graphics and Applications*, 9(7), July 1989, pp 71-83.

[Apgar 88] Apgar, Brian, Bret Bersack and Abraham Mammen, "A Display System for the Stellar™ Graphics Supercomputer Model GS1000™", *Computer Graphics*, 22(4), (Proceedings of SIGGRAPH '88), pp 255-262.

[Athas 88] Athas, William and Charles Seitz, "Multicomputers: Message Passing Concurrent Computers," *Computer* 21(8), August 1988, pp 9-24.

[Borden 89] Borden, Bruce, "Graphics Processing on a Graphics Supercomputer", *IEEE Computer Graphics and Applications*, 9(7), July 1989, pp 56-62.

[Foley 90] Foley, James, Andries van Dam, Steven Feiner, and John Hughes, *Fundamentals of Interactive Computer Graphics*, 2nd Edition, Addison Wesley, Reading, Massachusetts, 1990 (in preparation), sections 18.5 and 18.6.

[Fuchs 89] Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics*, 23(3), (Proceedings of SIGGRAPH '89), pp 79-88.

[Reynolds 87] Reynolds, Craig, "Flocks, Herds, and Schools: A Distributed Behavior Model", *Computer Graphics*, 21(4), (Proceedings of SIGGRAPH '87), pp 25-34.

[Torborg 87] Torborg, John, "A Parallel Processor Architecture for Graphics Arithmetic Operations", *Computer Graphics*, 21(4), (Proceedings of SIGGRAPH '87), pp 197-204.

[Van Dam 88] Van Dam, Andries, ed. "PHIGS+ Functional Description Revision 3.0", *Computer Graphics*, 22(3), July 1988, pp 125-218.