Hierarchical Graphics Databases in Sort-First

Carl Mueller, UNC-CH Computer Science

ABSTRACT

Sort-first architectures for parallel rendering use the natural frame-to-frame coherence of primitives on the screen to increase performance by grouping primitives that are close together on the screen onto the same processor. This technique reduces processor-to-processor communications overhead. As primitives move on the screen, they may also be moved to other processors. This migration poses many problems for the editing and traversal of hierarchical graphics databases (HGD's) such as PHIGS.

In this paper we describe the major problems associated with implementing an HGD on a sort-first system. We discuss the basics of HGD's and how they extend to parallel graphics systems. We show why bookkeeping for HGD's is a more complex issue for sort-first than for other parallel architectures. We then examine two possible solution branches for this issue and delve into the design choices and implementation issues that arise with either method. The methods differ in the amount of bookkeeping that they do versus the amount of primitive cull-testing that must be performed.

CR Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; I.3.1 [Computer Graphics]: Hardware Architecture; I.3.6 [Computer Graphics]: Methodology and Techniques.

Additional Key Words: computer image generation, parallel computing, MIMD, sort-first, hierarchical graphics database.

1. INTRODUCTION

The hierarchical graphics database (HGD) is a powerful tool for easily expressing and manipulating complex three-dimensional models [6]. It is no surprise that HGD implementations are available for all of today's real-time graphics supercomputers [7, 14, 13]. We focus in this paper on the implementation of HGD's on highly parallel graphics systems, and in particular, on the class of architectures known as "sort-first" [9, 11, 3, 12].

We take a moment to note that the HGD is a type of retained-mode graphics-pipeline API (application-program interface). Immediate-mode APIs such as OpenGL are a popular alternative way of presenting geometry to the graphics pipe, but we note that each type of API has its uses, and in this paper we focus entirely on the retained HGD.

Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. mueller@cs.unc.edu

1.1 HGD Basics

As a matter of introduction and also of definition of terms, let us consider the elements of a basic, generic hierarchical graphics database. Such a database consists of one or more structures. A structure can contain:

- primitives (triangles, polygons, lines, etc.)
- surface parameters or attributes (colors, texture to use, etc.)
- transformation matrices
- executes of other structures (cyclic executes are not allowed)

For various reasons (which may become evident later), a system can deal with structures more easily when the different kinds of structure content are kept segregated (rather than freely interspersed). This does not decrease the generality of the database.

We define the following operations on structures:

- create a new structure, add to it, close it
- delete a structure
- post a structure (add it to the scene)
- unpost a structure (remove it from the scene)
- change an element in a structure to a similar element (i.e., change a matrix, color, execute, etc.)

Finally, a scene consists of:

- a viewing matrix (plus other viewing parameters)
- default surface parameters
- the list of posted structures

Drawing the scene then consists of traversing the posted structures. The scene can in fact be viewed as a single main, root structure to which you can only add or remove execute calls. In this paper, we will use the term "instance" to refer to an instance of a structure that is being executed, whether due to the structure being posted or due to an execute call by some parent structure.

1.2 Parallel Rendering Systems

The implementation of an HGD on a uniprocessor system is straightforward. However, due to the computational requirements of real-time rendering, today's high-performance graphics systems must be highly parallel systems. Such systems use parallelism for both the transformation and rasterization stages of the graphics pipeline [6]. There are various ways of dividing the rendering task among the various processors and recombining of the various processor outputs into a finished scene. According to [11], the possibilities fall into the categories "sort-first", "sort-middle", and "sort-last" (SF, SM, and SL for short).

Both sort-middle and sort-last start with a fixed distribution of the graphics database. In sort-middle, the primitives are transformed by several transformation processors and then routed to several rasterization processors, each assigned to different portions of the screen [7, 5]. In sort-last, the transformation and rasterization processors are paired together such that full-screen images are produced by each node. The individual images are composited together in a final sorting step [9, 10].

Sort-first also pairs together transformation and rasterization processors, but each node is assigned a portion of the screen. It relies upon an early screen-space sort to route primitives to the correct nodes. Unlike sort-middle and sort-last, the database distribution is not fixed; the primitives are constantly migrating.

This property takes advantage of frame-to-frame coherence, reducing communication bandwidth requirements for sort-first. However, it is also what makes implementing HGD's on sort-first particularly difficult.

Sort-first has various issues which affect database design choices. One issue is that SF rendering is ideally done with at least two passes over the database: a sorting pass followed by a rendering pass. Another major SF issue is load-balancing. Adaptive screen-space partitioning is a good load-balancing solution for SF, but it requires estimating the primitive work distribution over the screen [12]. Another initial pass over the database can be added for this purpose. A final SF issue to keep in mind is the method of dealing with off-screen primitives. The chosen bookkeeping method should not allow such primitives to accumulate very unevenly among the graphics processors.

2. HGD'S IN SORT-MIDDLE AND SORT-LAST

With either sort-middle or sort-last, the strategies for dealing with HGD's are similar [4, 9]. We describe these first since the basic ideas apply to all distributed database systems. In later sections we will see how these ideas are extended to work with sort-first.

The main issue is how to distribute the database, and the common solution is as follows: replicate most of the hierarchy except for the primitives, which are distributed. The distribution scheme attempts to give each processor a similar work load.

Thus each processor has knowledge of the entire hierarchy (and everything that affects the traversal state), but each has only a subset of the primitives. It is desirable for SM and SL to scatter the primitives across the processors such that each processor will likely have a similar number of on-screen primitives to transform.

All of the database operations are relatively straightforward. For traversal, each processor simply traverses the hierarchy and processes the subset of the primitives which it has. Most of the other operations are performed by broadcasting the command to all of the processors, which change their copies of the database appropriately. The only exception is when the command is to create or change a given primitive. In this case, the command should go to only the relevant processor.

Note that with this kind of strategy, only the work related to the primitives is parallelized, while all other kinds of operations are replicated. Some optimizations are possible to reduce the extra work. A common one involves batching primitives together somewhat, then removing redundant state changes and any structures that become completely empty on any given processor [4]. This can be very helpful for applications which either have frequent state changes or which use a large number of small structures. On the negative side, it makes keeping track of the database somewhat more complicated, since all processors will no longer "see" the same hierarchy.

3. HGD'S IN SORT-FIRST

3.1 Introduction

In the following sections, we examine the issues surrounding the implementation of HGD's in sort-first. First we consider the question of how to distribute the actual database among the processors. Next, we start to consider the issues surrounding primitive migration. We set up an example to help illustrate the problems and solutions. We divide the solution space into two branches and examine how each addresses storage, basic operations, traversal, and migration of primitives.

3.1.1 Distribution Method

With SF, we have the choice of using either the same "distribute-by-primitive" method (Fig. 1) as with the other architectures, or the alternative "distribute-by-structure" method (Fig. 2) [4]. In this method, one tries to chop up and distribute the database along structure lines, thus giving each processor only a portion of the overall hierarchy. Such a distribution scheme would appear to better parallelize the traversal load.

However, this scheme makes traversal very complicated as a result of instancing (structure executes). This is because the inherited state (including the matrix transformation) needed for a given structure depends upon all of its parent structures, but those structures may not be present where they are needed (as in Fig. 2, structure C). To eliminate this problem, we can replicate all such state information, but this make the method very similar to the distribute-by-primitive approach.

Given that the hierarchy and state information are replicated, we are left with the choice of how to actually distribute the primitives themselves. Scattering the primitives finely usually

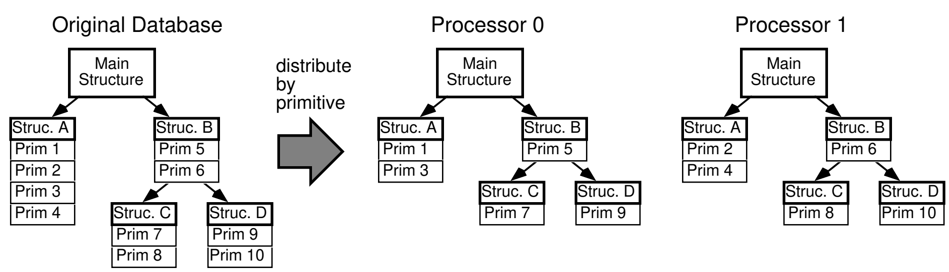

Figure 1. Distribute-by-primitive method

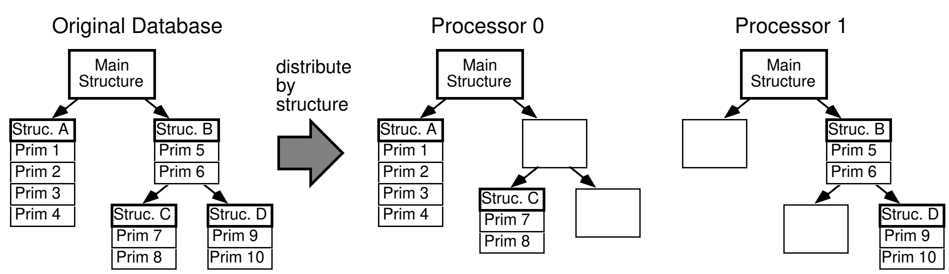

Figure 2. Distribute-by-structure method

guarantees that they will be evenly distributed across the processors. Distributing the primitives in large consecutive clusters or groups provides no guarantee of an even distribution unless one bases the group size on a-priori knowledge about the database size (something most API's don't provide for). Using medium-size groups is perhaps a better choice.

For SM or SL, one should choose a relatively fine primitive scattering since this is a main factor in determining the overall processor load-balance. For SF, the initial primitive distribution determines the load-balance for the sorting pass over the database, but not for the overall rendering. Thus a fine scattering is good for this purpose, but a coarser scattering has the desirable characteristic that it may leave some structures empty. With empty structures, one has the opportunity to skip over them during database traversal, improving the parallel efficiency of the system.

Also with SF, the actual primitive data will tend to cluster together due to the very nature of the architecture. Whether or not this provides a window for greater efficiency is determined by various bookkeeping decisions which are discussed in sections below.

3.1.2 Primitive Migration Issues

With SF, unlike SM or SL, we must deal with primitive migration. This means that a processor must end up having the right set of primitives to render each frame. This set consists of every primitive which has an instance appearing in a screen region for which this processor is responsible. Note that we must manage both the actual primitive and its instances. To migrate primitives, we must find efficient solutions for deciding where the primitives go and dealing with them as they arrive. There are many efficiency criteria: a given processor should not have to store or transform many more primitives than those which fall in its region; the cost of the basic editing and traversal operations should be kept minimal (they should not take much longer than SM or SF require); finally, of course, the amount of bookkeeping information and its management should not weigh too heavily.

Given that only primitives (with their intrinsic state) will be migrating, we can see that it is necessary to keep primitives distinct from the other types of database contents. However, depending upon the database semantics, some primitive attributes may be tied into the primitive's position in the database, and thus we must be able to efficiently maintain these ties as primitives migrate. Such attributes may include the current color or the latest transformation matrix. Manipulating the database semantics to limit such ties is one possible solution (for instance, only allow a single current color and matrix per structure), but another possibility is to maintain efficient lookup tables for such information and assign appropriate indices to each primitive.

3.1.3 Database Representation Issues

In order to illustrate various problems associated with database representation and traversal, we set up an example database scenario:

Structure A	Structure B	C (cont'd.)
xform 1	xform	primitive e
execute B	color	primitive f
xform 2	execute C	primitive g
execute B		primitive h
xform 3		primitive i
execute B	Structure C	primitive j
xform 4	primitive a	primitive k
execute B	primitive b	primitive l
	primitive c	primitive m
	primitive d	primitive n

Figure 3. Example database contents

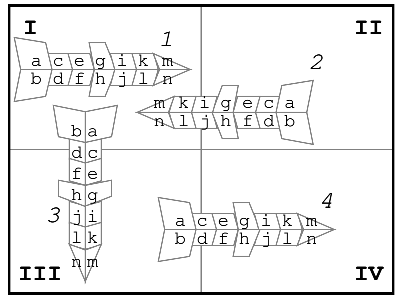

Figure 4. Example database scene

The database consists of three structures: A, B, and C. Structure A is posted. The hierarchical structure of the scene can be viewed two ways:

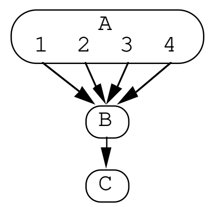

Figure 5. Minimal view

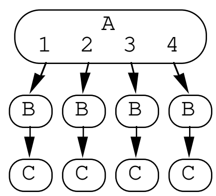

Figure 6. Maximal view

Figure 5 shows a *minimal* view of the database by presenting it as a directed-acyclic graph. This graph is derived directly from Figure 3, with each structure shown as a node and each execute shown as an arrow. It contains the minimal information we need to traverse the database. Figure 6 shows a *maximal* view of the database by presenting it as a tree. This time each execute points to a separate instance of each structure. This view gives us a clearer picture of how much work this database requires to actually render.

Now let us suppose an actual rendering of this scene appears as shown in Figure 4. The screen regions are the quadrants in Roman numerals, while the given instances of C's primitives are numbered in Arabic.

For this scenario, we notice that the processors for regions I and III must receive the entire set of primitives, while those for regions II and IV only need to deal with a subset. The problem we face is to determine for each branch of Figure 6 which set of primitives must be rendered by each processor.

We attack this problem by considering two main solution methods. The methods are based upon which view of the hierarchy is stored and managed:

1. The minimal set ("min-set") containing only defined instances of primitives (as in Fig. 5).
2. The maximal set ("max-set") containing all of the instances of all primitives (Fig. 6).

Each method has an interesting set of advantages and disadvantages.

3.2 Common Issues and Solutions

Before we go into a discussion of either method, we take a moment to examine some more issues that are common to both methods.

With either method, there is the question of what granularity to use for database migration and bookkeeping. That is, should bookkeeping be done for every individual primitive, or should it only be done at the whole structure level? We propose a solution that is somewhere in between, and this is accomplished by clustering primitives into groups or chunks of a given maximum size. The bookkeeping costs (both computation and storage) would therefore be amortized over the primitives in the chunk, this being at the expense of having some primitives be unnecessarily sent or transformed. Finding the chunk size which balances these factors is something we will have to look into, though. An additional benefit of having fixed-size chunks is that it facilitates efficient memory management.

As we mentioned earlier, SF typically performs more than one pass over the database. Also, when a processor receives primitives from other processors, it must be able to efficiently store these primitives into their correct places in the database. Both of these factors suggest that one should build and store information from the first pass over the database to facilitate subsequent database operations. Such information includes:

• Combined transformation matrices. Normally, these are discarded during single-pass traversal. In SF, primitives arriving from other processors will need to be transformed, and thus all the combined matrices must be kept around in such a way that they can be quickly found and used.

• A linear index table of all the structures, with structure numbers that are consistent across the processors. Such a table facilitates the storing of arriving primitives in the correct places. Since all processors have a copy of the hierarchy skeleton, the synchronized table is easy to build.

• Information about which primitive chunks actually need to be rendered on this processor. While this is pretty obvious, what is less obvious is that such information corresponds to a local max-set view of the database. Thus even if only the min-set is being kept around from frame to frame, one must build a certain amount of the max-set for use during a given frame.

If one also builds a linear table of all max-set structures, the database is essentially "flattened out," and any subsequent traversal passes can be performed very easily.

3.3 The Min-Set Method

Keeping track of only the minimal set of database information has a number of advantages. Since this is the same procedure used by SM and SL, it means that both the storage space required and the cost of basic operations will be similar among SF, SM, or SL. However, this method has a significant drawback: for a given traversed instance of a structure, a processor does not know which primitives (if any) it must render.

Let us examine the traversal problem by considering Figure 4. We see that instance 1 of structure C falls completely in region I. Instances 2 and 3 fall partially in this region, while instance 4 falls completely outside of this region.

The processor for region I must have all of the primitives in C because it will need them to draw instance 1. The processor also has the entire hierarchy information and thus it knows about all the other instances. How will this processor know that it only needs to render instance 1, parts of 2 and 3, and not worry at all about instance 4?

3.3.1 Min-Set Implementation

In order to efficiently decide which instances of a structure each processor must render, one may implement bounding volumes around each structure [2]. This allows a given processor to quickly test whether or not an instance of a structure falls in the processor's region (cull-test). This is done by first checking whether or not the bounding volume intersects the region and only acting on the enclosed primitives if it does.

This will help deal with most cases. For this particular example, the processor for region I can use this information to quickly decide that it need not worry about instance 4 of structure C, since it falls completely outside of its region. It still must deal with instances 2 and 3, however.

There are many different choices for what kind of bounding volumes to use. These including spheres, axis-aligned boxes, oriented boxes, and multiple slab intersection [1]. These are listed in order of increasing complexity and tightness of fit, which together represent the major tradeoff associated with the choice of which to use. Tighter fitting volumes will help to decrease the number of primitives that a given processor must deal with, and they will also help to decrease the amount of primitive communication. However, since bounding-volume tests must be performed for every instance of every structure for each frame, the tests should be as simple and fast as possible.

Given that one has bounding volumes around each structure, one could consider building a hierarchy of bounding volumes and using various techniques to further speed up the rendering by reducing the amount of traversal [2]. However, this can only be readily applied to the max-set, not to the min-set.

Since structures can have arbitrarily large numbers of primitives, it really is necessary to be able to place bounding volumes around portions of structures. As discussed above, the bounding volumes could be applied to chunks of primitives. Ideally, the primitives in each chunk will be near each other, but this is difficult to enforce. The grouping of primitives into chunks can be done in a number of different ways. One can group the primitives as they are received: every N primitives within the structure forms a new chunk. Alternatively, it could be left up to the application program to determine the chunking, or the system itself could implement one or more of the various existing partitioning algorithms.

Aside from bounding-volume creation, one must also worry about the effects of editing on bounding volumes. If primitives within a structure are changed, the bounding volume may need to grow or shrink.

3.3.2 Primitive Migration / Communication

We now examine the problem of determining how primitives will be handed off from processor to processor. We will continue to use the chunk idea from before; thus rather than keeping track of and sending primitives around individually, a processor would only worry about managing and sending whole chunks.

There are two solution methods for the problem at hand:

1. Push method: the chunks from the database are assigned to the various processors, and each processor is responsible for making certain that its chunks are sent to the right places.

2. Pull method: a processor will, upon seeing that it needs a given chunk, request that it be sent from some known location.

3.3.2.1 Min-Set Push Communication

Again, each method has advantages and disadvantages. For the push method, we must decide which processors will be responsible for which chunks. A processor must make sure that the chunks for which it is responsible are located on the correct set of processors. This bucket-sorting process is done by checking the bounding volumes for each chunk against the screen subdivisions. In order to know which processors must be sent a given chunk, the sending processor must store for each chunk a bit vector indicating which processors have a copy.

We will use the terminology of calling a processor a "master" of a given chunk if it is responsible for the distribution of that chunk, while other processors which have a copy of the chunk solely for rendering purposes are referred to as "slaves".

Assuming that the assignment of chunks to masters is fixed, there will typically be two copies of the primitives present at all times: the fixed master set (where the chunks were assigned) and the dynamic slave set (where the chunks actually appear). If the assignment of mastership is allowed to be dynamic, this can be reduced to a single copy. However, without active load-balancing, the actual distribution of this copy may vary widely from processor to processor. Due to structure instancing, it is not clear that load-balancing the rendered primitives (the max-set) does anything to balance the min-set. Fixed mastership solves this problem since the master distribution can be assigned evenly while the database is being created. Having a good master load balance means that the initial SF pass (or two) over the database will be well balanced among the processors.

There are other issues between fixed and dynamic mastership. The latter provides greater possibility that on a given processor structures will become empty, giving the opportunity to skip over such structures during traversal (section 3.5). But dynamic mastership exhibits the off-screen primitive problem, where master primitive chunks accumulate at processors rendering the screen edges and thus require redistribution. Fixed mastership avoids this problem.

During traversal, a processor will treat "master chunks" and "slave chunks" differently. For master chunks, the processor must execute the bucket-sorting algorithm: it determines the set of regions in which the chunk appears based upon bounding-volume overlap. The master must then compare this set with the existing set of slaves and finally send the chunk to any new slaves.

For the slave chunks, the processor needs only to perform a simple cull test to see whether or not the chunk appears in its region and proceeds accordingly. However, slave chunks introduce a new challenge: one must decide when to delete them. One can only delete them when no instance of the chunk appears in this processor's region. Since one cannot know this until after a complete traversal, one can collect this information during one frame and decide whether or not to delete the chunk during the next frame.

3.3.2.2 Min-Set Pull Communication

With the pull method, the job of making sure that a processor receives the right set of primitives is left to the receivers (as opposed to the senders). A processor will transform a bounding volume and see if it must render a given chunk. If it does not have the chunk, then it will request it from a known location (the chunk's "master" processor). To establish known locations, however, requires again that there be two distributions of primitives: the fixed, known (master) distribution, and the dynamic (slave) distribution.

Unlike push communication, the pull method does not require master processors to store and manipulate the bit-vector indicating which slave processors have a copy of a given chunk. A larger advantage of the pull method is that the processors need only perform the simple cull test and not the full bucket-sorting algorithm for each chunk.

The disadvantages of the pull method are two-fold. One is the need to have request messages. This adds a significant number of messages to the rendering process compared to the push method. Using buffering to reduce the number of messages sent will likely increase rendering latency. The request messages also require processors to interrupt their normal work in order to respond. Finally, the pull method also requires that all processors transform and test all chunks' bounding volumes for visibility (as opposed to only testing the present chunks). No portion of the hierarchy can be skipped over, since all chunks are always potentially present until tested otherwise.

Load-balancing considerations are similar to that of the push method, except that the only additional work required of masters is the processing of request messages and the sending of chunks as a result.

Considering the two methods, it appears that the push method is the better choice. Since it does not involve any primitive request messages, it has lower communication bandwidth and latency. The push method has much lower computational requirements and it also offers more potential for optimizations.

3.3.2.3 Bookkeeping for Arriving Primitives

With either method, there are some additional problems which must be dealt with, both concerning arriving primitives. When a processor receives a chunk of primitives, it must be able to store the primitives efficiently in the correct place in the hierarchy, and it must efficiently transform and render them, using the correct state information (e.g., transformation matrix).

The problem of efficiently storing the arriving primitive chunks has already been addressed. However, there is still the problem of rendering the new chunks. Part of the problem is that the receiving processor must have the correct transformation matrices available for ready use. This has also been addressed.

What has not been covered is the fact that the receiving processor will not necessarily know which max-set instances should be rendered for the new min-set chunks. Processors will need to be able to identify all the max-set transformation states that correspond to a given min-set structure. Such lists can be constructed by linking together corresponding max-set information structures. When a primitive chunk is sent, it can include information about which was the first instance where the chunk appeared in the destination processor's region. That processor can start from this point in the list and then perform cull tests for this chunk as it is transformed according to the rest of the instances in the list.

3.4 The Max-Set Method

With the minimal-set method, a processor must always transform and cull every instance of every chunk it has. With the maximal-set method, we reduce this work by choosing instead to keep track of each primitive chunk instance separately. This information can be distributed among the processors rather than requiring each to compute it for every frame. With the max-set method, bounding volumes are not a necessary component. However, they can be retained with primitive chunks to provide the advantages mentioned earlier.

The max-set method has its downside, however. It must generate and store bookkeeping data for all the primitive chunks within the max-set. The problem is that the max-set can change significantly as the hierarchy is modified, and thus the basic database operations become more complex.

Let us consider the max-set method with respect to the example shown earlier. Each processor now keeps track of a structure similar to Figure 6 (perhaps in addition to the structure in Figure 5). However, on a given processor, what is present in the structure instances marked "C" will vary from instance to instance. Consider the processor for region I. For instance 1, it will have data for all of the primitives in the original structure; for instances 2 and 3, it will have data for only portions of the primitives; and for instance 4, it will contain no primitives at all.

Now consider modifying the original database. If one wants to change a primitive in structure C, there are four instances which are affected. If one wants to modify the execute in B (perhaps to execute some other structure not shown), then all the information about the primitives within the four instances in C becomes invalid. If we imagine that structure A is a substructure itself and if it is executed an additional time, then the entire A-B-C subtree must be replicated to maintain the maximal set.

3.4.1 Max-Set Implementation

To keep track of all the primitive chunk instances, we could consider a simple replication strategy whereby we generate the max-set by simply making copies of the original data for every structure instance that comes into existence. However, this method has major drawbacks. Should any part of a given structure change, one must find and change every instanced copy. Thus each edit becomes a linear-time (or worse) operation rather than constant. Also, whenever a chunk instance is sent from one processor to another, the entire chunk must always be sent, since there is no notion of the primitive data vs. an instance pointer.

A much better way of building the max-set would be to first start with the min-set, then make a tree of pointers that corresponds to the max-set. Each max-set instance would contain only pointers to the original data within the min-set. Given the much greater feasibility of this pointer method over the replication method, the pointer method is the only one we will explore further.

Thus with the pointer method, we maintain two appearances of the database. In the min-set copy we store the exact same information as for the min-set method. For a given processor, this information includes (at least) primitive chunks for which any instance appears in that processor's region. The max-set copy of the database contains an entire skeleton of the actual max-set. Within each max-set structure, there will be pointers to only the chunks which appear in this processor's region for that instance. In addition, each max-set structure will point to its original min-set version. This will allow one to quickly find the non-primitive information for a given structure. Finally, the max-set can also be used to store certain information generated during traversal (namely, combined transformation matrices). One difference between the min-set and max-set skeletons in that the min-set uses "soft-links" for its execute calls (the calls are by structure ID), whereas the max-set would use "hard-links" (memory pointers).

3.4.2 Max-Set Operations

All operations are initially performed upon the min-set copy, and then the system must eventually update the max-set copy. As before, we retain the idea of master and slave processors for primitive chunks which overlap multiple regions. The master processor is responsible for making sure that the chunk goes to all the necessary slave processors. To extend this idea to the max-set method, we make the master processors responsible for properly distributing all the chunk instances. Thus the master must consider not only which processors have a copy of a given min-set primitive chunk, but also which processors have a copy of the max-set instance pointers for that chunk.

Now let us consider how to perform the basic operations on the max-set database. The creation of new structures is straightforward; it only affects the min-set. The deletion of a structure is similar: you should only delete non-instanced structures. Posting a structure requires the construction of a max-set tree, while unposting a structure requires deletion of the same. The expense of posting structures can be minimized by deferring the real work until scene traversal time. Similarly, unposting can be done by putting the tree in a garbage-collection list, to be cleaned up after traversal and rendering. Editing parameters and matrices within the hierarchy is straightforward. Editing primitives can also be straightforward assuming one uses the "fixed" master system mentioned earlier (meaning the primitive will be easy to find). Finally, changing an execute is potentially complex, but again we minimize the work by delaying it.

3.4.3 Max-Set Traversal

All of the complex work is delayed until scene traversal time. During the first pass, a processor must simultaneously traverse both the min-set and the max-set, making sure that the max-set appropriately parallels the min-set. When there are pieces of the min-set that do not exist in the max-set (due to a new post or execute), the processor creates them on the spot. When the processor comes across pieces in the max-set that have become obsolete due to database edits, these are added to the garbage-collection.

When a new instance of a structure is to be created in the max-set, a processor places within it pointers to all of its master chunks for that structure. This guarantees a complete yet disjoint initial distribution for the max-set structure instance.

During traversal, a processor considers each chunk pointer in the max-set structure. If it points to a master chunk, then this processor must see that this chunk instance (and its corresponding min-set data) is properly distributed to all necessary slaves. If the chunk pointer is directed at a slave chunk, then the processor only decides if it still needs to keep the pointer. If it does not need it anymore (due to the associated instance no longer falling in this processor's region), then it can delete it and possibly also the associated slave chunk data. To know about this possibility, reference counts can be used to keep track of how many max-set instances are pointing to the min-set data.

The bookkeeping just discussed is greatly simplified if fixed masters are used. If dynamic mastership is used, then one must be careful to migrate all the information associated with a given primitive chunk: which processors have copies of the actual chunk, and which processors have a copy of each instance. Not transferring this information together leads to greater complications.

3.4.4 Max-Set Primitive Migration

We now turn some additional attention to the problem of transmitting primitives, since new complications arise due to the fact that we have to worry about both min-set primitive chunks and their max-set instances.

When a processor sees that it must send a chunk instance, it will go ahead and place a message regarding that instance into the appropriate outgoing buffers. The processor must also then check to see if the receiving processor has a copy of the original chunk data. If not, this is also queued to be sent.

The difficulty in this process is finding an efficient way for the master processor to pass along the instance pointer information. The pointer must refer to the original chunk by an ID (since memory pointers are not valid across processors). The receiver must have a quick way of looking up the ID to see where in its memory the chunk is found. One solution is to place the chunk IDs in a hash table. Keeping a chunk index table for each min-set structure may also be practical.

Thus when a processor receives a chunk of primitive data, it just places it in the appropriate structure in the min-set. When a processor receives a chunk pointer, it places the pointer in the appropriate max-set structure and links it to its min-set data. It then (or eventually) renders that max-set chunk.

3.5 Improving Efficiency

For various reasons, one may find in a distributed graphics database structures which contain no primitives. One reason is when the number of primitives in that structure is smaller than the number of processors times the number of primitives per chunk. Applications often have large numbers of small structures (in addition to smaller numbers of larger structures), and thus locally empty structures are likely. In addition, with dynamic mastership, SF allows the possibility of primitives to migrate out from structures and leave them locally empty on some processors.

Given the presence of many empty structures, it pays to optimize traversal by not traversing into empty portions of the hierarchy. Bounding volume hierarchies (see section 3.2.1), applicable to the max-set method, provide this possibility for locally off-screen portions of the hierarchy. It does not help with empty on-screen structures.

A more general method is to simply keep a flag for each structure to indicate the presence of any primitives. For the min-set, this is about all you can do. During traversal, one can skip empty structures, though backtracking may be required if non-empty structures are found further down the tree.

For the max-set, the flags can be made hierarchical, indicating the presence of primitives within entire branches of the structure hierarchy. One difficulty with this method is that any primitive chunks which arrive in previously "empty" portions of the hierarchy require backtracking to compute the relevant transformations to use. Another small problem is the fact that if all the processors are not performing the same traversal, it is not as easy to generate synchronized structure indices. One can overcome this problem either by always doing light traversal (to compute structure numbers), or by keeping track of the number of sub-structures under any given structure.

4. EXPERIMENTAL STUDY

In order to evaluate the effect of chunk-based primitive bookkeeping and to help us understand performance differences between the min-set and max-set bookkeeping strategies, we have conducted some experiments. These experiments are based upon statistics gathered from a software-based sort-first implementation. We will describe the implementation, the input data, and the test cases run. In section 4.2 we discuss the results.

4.1 Experimental Setup

We have written a parallel sort-first software-based implementation. PVM is used to provide multiprocessing capability [8]. A host process spawns processes for each graphics processor plus a framebuffer. The system is capable of using either the min-set or the max-set method of database bookkeeping. One can vary the maximum number of primitives per chunk in addition to several other parameters.

The system places a 3D bounding box around each primitive chunk. During database traversal, these bounding boxes are transformed into screen space, and their bounding rectangles are computed. These data are used to perform both view-frustum culling and bucket-sorting of the primitive chunks.

The test database is a model of a Ford Bronco. The Bronco consists of 466 structures and 74,647 triangles. For the test runs, the Bronco was instanced three times, with the instances distributed around a circle. In the animation, the camera revolves around the circle while viewing approximately one half of the circle's area. The color plate section shows representative frames from the animation.

Test runs were made using various parameters. The number of primitives per chunk included: 1, 5, 12, 26. The number of processors included: 4, 8, 12, 16, 20. Both the min-set and max-set bookkeeping methods were tested.

Due to concerns about the bounding box volume of the primitive chunks formed, different strategies for forming the primitive chunks were tried before selecting one. The first strategy (called "unsorted") was to take every N primitives and make them a chunk, where N is the selected chunk size. The second strategy ("partially sorted") was to presort the primitives in the input file by performing an octree-based sort over each structure's primitives, then proceed as with the first strategy. The third strategy ("sorted") was to read in the primitives in each structure and form the chunks by performing a relatively simple 3D spatial-subdivision algorithm. Chart 1 details the results of these different strategies.

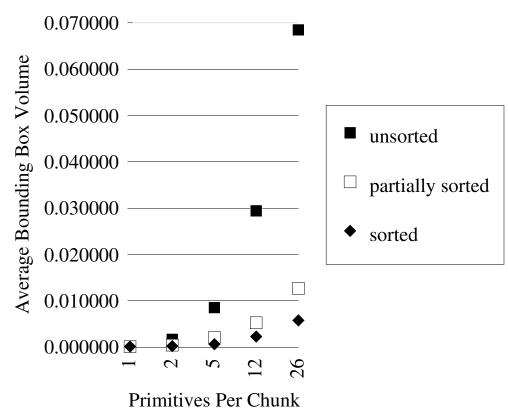

Chart 1. Chunking strategy results

The sorted method was chosen for use with the test runs. As the sorting is only done at database load time, this is a reasonable choice for a model with static structures.

In the tests, for each frame drawn, various statistics concerning the number of operations done on each processor are written to a number of trace files. For a given set of test conditions, the statistics across the processors and across the frames in the animation are averaged together. This method provides a broad view of each test case.

4.2 Experimental Results

First we examine some of the results concerning the chunking strategy. Chart 2 shows us the relationship between the chunk size and the amount of communication that SF performs. The chunk size has a moderate effect upon the amount of communication. The number of processors has a much greater

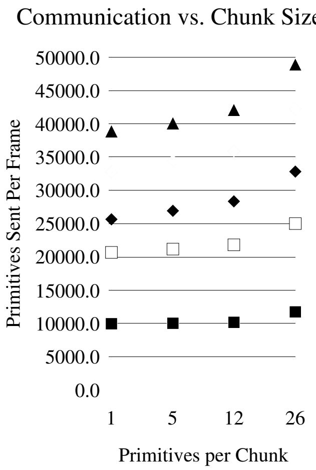

Chart 2.

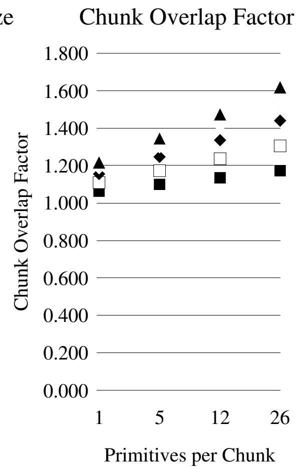

Chart 3.

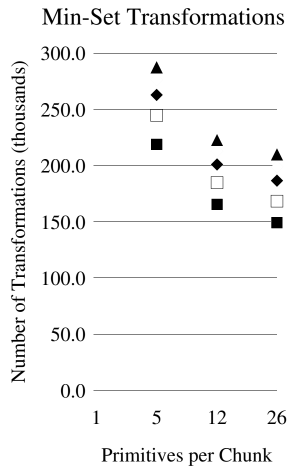

Chart 4.

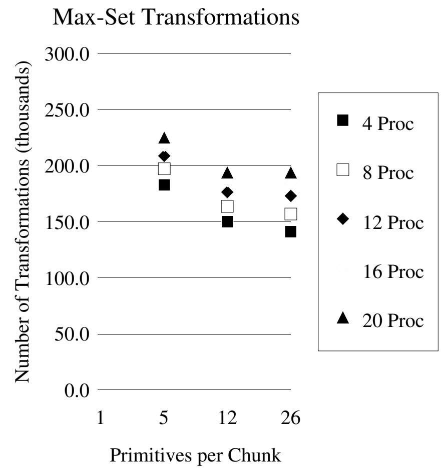

Chart 5.

impact upon this statistic than the chunk size, though as the number of processors increases, chunk size has greater impact.

In Chart 3, we show the average chunk overlap factor for the different test cases. The chunk overlap is the average number of regions each chunk overlaps. Similar to the primitive overlap factor (see [11]), this value has a significant impact upon the parallel efficiency of the system. This is because whenever a chunk's bounding rectangle overlaps multiple regions, all the primitives in the chunk must be transformed for each region. This source of inefficiency increases as the regions become smaller and as the chunks become larger. Note that this chunk overlap inefficiency affects only a part of the graphics pipeline: the unnecessarily transformed primitives are culled from further processing. Chart 3 shows us the fairly direct relationship between the chunk overlap factor and the ratio of the chunk bounding box size vs. the average region size. As this ratio increases, chunk overlap steadily becomes a source of inefficiency we must contend with.

In order to compare the min-set and max-set bookkeeping methods, we examine the total number of transformations that occur in each. This includes transformations of the master chunks, the slave chunks (including newly received ones), and of all the primitives in the chunks which overlapped any processor's region. We count one transformation for each chunk or primitive. These numbers are shown for the min-set method in chart 4, and for the max-set method in chart 5. We have omitted the numbers for the chunk size of one since it is not practical to place a bounding volume around a single primitive (the numbers are disproportionately large).

The total number of transformation has an interesting relationship to chunk size. Larger chunks mean that there are fewer of them to transform. However, as mentioned before, larger chunks increase the overlap inefficiency. Comparing the results on charts 4 and 5 show us that the max-set method has much less transformation work to do than the min-set method. The source of this difference comes from the number of slave chunk bounding boxes that are transformed. This figure is three times higher for the min-set method, which is what we expect, given that there are three instances of the Bronco being executed.

5. CONCLUSION

It should be evident that devising a distributed hierarchical graphic database for sort-first is much more difficult than for sort-middle or sort-last. The complications arise due to structure instancing and primitive migration. A pair of solutions have been proposed to address these problems. Each solution branch has its set of tradeoffs in addition to a variety of smaller issues to decide.

For the min-set solution, we keep track of only the original primitive data. This simplifies the bookkeeping, but the problem of knowing which primitives to render for a given instance arises. We have dealt with this problem through the use of bounding volumes around primitive chunks.

With the max-set solution, we keep track of both primitive data and all primitive instances. This reduces the number of primitive chunks that are cull-tested against the local viewing frustums, but at the expense of more complex bookkeeping.

Which of these methods is a better choice depends to a large extent upon how much applications make use of structure instancing. The additional overhead of the max-set method does not appear to be very significant, and this method is the better choice for applications with moderate or large amounts of instancing. For applications that make little or no use of structure instancing, the min-set method is the obvious choice.

Further exploration of different primitive bookkeeping methods is worthwhile. The question of static versus dynamic mastership also remains open, with the latter offering opportunities for traversal optimization while introducing new complexities. In addition, we are interested in examining various optimizations to reduce the inefficiency caused by the chunk overlap factor.

Exploration is currently continuing into these issues. In addition, a more detailed analysis of the overhead from the various bookkeeping strategies is being conducted. Finally, we are examining how these bookkeeping issues interact with other system issues, especially SF load-balancing. It is hoped that from this research a real-time implementation will be written to run on hardware such as the upcoming PixelFlow machine from Hewlett Packard and UNC [10, see also http://www.cs.unc.edu/~pxfl/arpa4_95].

6. ACKNOWLEDGMENTS

I would like to thank Anselmo Lastra and Jonathan Cohen for helping me with this paper. Thanks go to Division and Viewpoint for providing the Bronco model, and to Jonathan for helping to convert it. This research was supported in part by the Defense Advanced Research Projects Agency, ISTO Order No. A410, and the National Science Foundation, Grant No. MIP-9306208.

REFERENCES

[1] Arvo, James and David Kirk, "A Survey of Ray Tracing Acceleration Techniques," Chapter 6 in *An Introduction to*

Ray Tracing, edited by Andrew S. Glassner, Academic Press, New York, 1989.

[2] Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM 19*, 10 (October 1976), pp. 547-554.

[3] Cox, Michael. *Algorithms for Parallel Rendering*, Ph.D. dissertation, Princeton University, 1995.

[4] 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.

[5] Ellsworth, David. "A Multicomputer Polygon Rendering Algorithm for Interactive Applications," 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. 43-48.

[6] Foley, James, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*, 2nd Ed., Addison-Wesley, Reading, Mass., 1990.

[7] 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.

[8] Geist, Al, Adam Beguelin, Jack Dongarra, Robert Mancheck, Weicheng Jiang, and Vaidy Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, Mass., 1994 (also http://www.netlib.org/pvm3/book/pvm-book.html).

[9] Molnar, Steven. *Image-Composition Architectures for Real-Time Image Generation*. Ph.D. dissertation, TR-91-046, University of North Carolina at Chapel Hill, 1991.

[10] 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.

[11] 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.

[12] Mueller, Carl. "The Sort-First Architecture for High-Performance Graphics," Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, April 9-12, 1995), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1995, pp. 75-84.

[13] Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24-29, 1994). In Computer Graphics Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, New York, 1994, pp. 381-394.

[14] Strauss, Paul and Rikk Carey. An Object-Oriented 3D Graphics Toolkit. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In Computer Graphics Proceedings, Annual Conference Series, 1992, ACM SIGGRAPH, New York, 1992, pp. 341-350.