Image Layer Decomposition for Distributed Real-Time Rendering on Clusters

Thu D. Nguyen* and John Zahorjan†
tdnguyen@cs.rutgers.edu – zahorjan@cs.washington.edu

Abstract

We propose a novel work partitioning technique, Image Layer Decomposition (ILD), designed specifically to support distributed real-time rendering on commodity clusters. ILD has several advantages over previous partitioning algorithms for our targeted environment, including its compatibility with the use of hardware graphics accelerators, decoupling of communication bandwidth requirement from scene complexity, and reduced communication bandwidth growth as the system size increases. Furthermore, ILD tries to optimize the rendering of a sequence of frames (of an interactive application) instead of only individual frames. We simulate ILD using traces taken from a VRML viewer. Our results show that ILD can be expected to work well up to moderately sized clusters and to outperform sort-last, a common partitioning approach, because of its smaller communication bandwidth requirement.

1 Introduction

The work in this paper is motivated by our effort to build a distributed rendering system that uses a cluster of workstations/PCs to provide *real-time* rendering performance that is one or two generations ahead of what can be achieved using only a single commodity machine. While the commodity cluster is an attractive platform because of its ubiquity, low cost, and ease of expandability, it presents a number of challenges. In particular, a cluster-based distributed real-time renderer must be structured to: *(i)* use the multiple hardware graphics accelerators in the cluster to increase rendering performance over what is achievable by a sequential renderer that makes use of an accelerator, *(ii)* minimize variance in the frame rate because variance is important to a user's perceived quality of service, and *(iii)* decouple communication bandwidth requirements from the complexity of the scene and the number of nodes used. Controlling communication bandwidth is particularly important because, in a commodity system, the network will likely be a bottleneck for both high frame rate and high image resolution. Thus, to scale with scene complexity, the bulk of the communication required to render a frame must be of rendered pixels, not scene primitives. To scale with system size, the bandwidth requirement must be independent of (or only weakly dependent on) the number of nodes, requiring that the pixels rendered on and transmitted by one node overlap only slightly with the pixels produced on the other nodes.

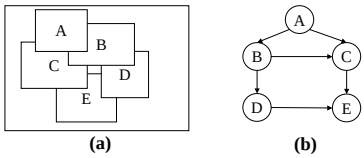

Figure 1. *(a) A particular view of a scene with 5 objects and (b) the corresponding occlusion graph (visibility order).*

To help meet the above challenges, we propose a novel work partitioning technique called *Image Layer Decomposition* (ILD). Each frame, given a set of scene objects to be rendered and a viewpoint, ILD assigns the objects in such a way that sets of objects assigned to different machines are not mutually occlusive. For example, given the visibility ordering of the five objects A, B, C, D, and E shown in Figure 1(a), a legal ILD partitioning for a 2-node system is $\{A, B\}$ and $\{C, D, E\}$. Because all the objects in the first set, $\{A, B\}$, are closer to the viewer than all the objects in the second set, $\{C, D, E\}$, we can compose the final image by simply "layering" the $\{A, B\}$ image on top of the $\{C, D, E\}$ image. In contrast, the decomposition $\{A, C\}$ and $\{B, D, E\}$ is not legal under ILD.

ILD is particularly suitable to our goals because: *(i)* The rendering problem at each node looks exactly the same as if it were an independent rendering application, facilitating

*Department of Computer Science, Rutgers University, 110 Frelinghuysen Rd, Piscataway, NJ 08854.
†Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195.
This work was supported in part by the National Science Foundation under Grant CCR-9704503.

the use of hardware rendering and all attendant optimizations. *(ii)* Using a small amount of preprocessing, we can predict the amount of data that each node must send per frame, allowing us to factor in the transmission time of each node when partitioning the work to minimize load imbalances. *(iii)* By replicating the scene on all nodes, we can avoid the need to communicate polygons, thereby decoupling the bandwidth requirement from scene complexity. *(iv)* Each node needs only send the pixels that it has rendered, such that, in the best case, the bandwidth required is only proportional to the size of the final image and not the size of the NOW.

In the remainder of the paper, we first discuss related work (Section 2) and describe ILD in detail (Section 3). We then simulate a long-running off-line version of ILD using traces taken from a VRML viewer to evaluate its potential for achieving good load balancing and scalability (Section 4). Our results show that ILD can be expected to work well (up to moderately sized clusters) when given scene objects of reasonable granularity. We also simulate a two-phase greedy on-line ILD partitioning algorithm (Section 5). While not optimal, our results show that this on-line algorithm performs only slightly worse than the off-line algorithm. Furthermore, our results show that the on-line ILD algorithm can be expected to outperform sort-last, a common partitioning approach, because of its smaller communication bandwidth requirement.

Many details about ILD as well as much of the simulation results cannot be presented in this paper because of space constraints. We refer the reader to [10] for a more complete discussion and evaluation of ILD.

2 Related Work

In this section, we discuss previous distributed polygon rendering efforts. We focus on polygon renderers because polygon rendering is the most common technique supported by hardware accelerators and is the method employed in our prototype. The basic task in polygon rendering can be viewed as the sorting of scene primitives from their world coordinates to the screen [13]. To date, most polygon distributed renderers can be categorized into three classes, *sort-first*, *sort-middle*, and *sort-last*, depending on whether the sorting process takes place during the geometric transformation phase, between the geometric transformation phase and the rasterization phase, or after the rasterization phase [6].

In sort-first (e.g., [14, 9]) and sort-middle (e.g., [1, 8]), the geometric transformation and rasterization of a polygon may be performed by different nodes, depending on the specific work assignment of each frame, typically requiring the redistribution of a significant number of primitives. This per-frame redistribution is a fundamental problem in our targeted environment because: *(i)* it requires either recomputing the geometric transformation of primitives that must be redistributed or accessing information that may be hidden inside a hardware graphics pipeline, and *(ii)* its bandwidth requirement directly depends on the complexity of the scene that is being rendered, violating one of our basic goals.

Sort-last (e.g., [7, 5]) corresponds to a data partitioning, where each node is assigned a subset of the polygons in the scene without any restrictions on the position of the polygons. Each frame, once each node has rendered its assigned polygons, the pixels must be sorted, typically using Z-buffering [2]. While compatible with our environment, sort-last is less than ideal because: *(i)* rendering nodes must send their Z-buffers along with the rendered pixels for the composition of the final image, which approximately doubles the required bandwidth, and *(ii)* primitives are typically assigned to renderers without regard to where they map to in screen space, implying that each renderer must typically send the entire image each frame.

3 Image Layer Decomposition (ILD)

ILD is similar to sort-last in that it uses a data partitioning. However, unlike sort-last, each frame, ILD (re)partitions the scene objects into P non-mutually occlusive subsets for a system with P nodes and assign each subset to a node for rendering. Because subsets are non-mutually occlusive, each node generates a coherent image layer; to compose the final image, ILD simply layers these image layers on top of one another according to the visibility order of the subsets (and thus do not need the Z-buffer generated at each node). Furthermore, when computing the partition, ILD attempts to minimize the overlap between distinct image layers in order to minimize the overall communication.

More specifically, each frame, ILD partitions the overall rendering work as follows:

1. Construct an occlusion graph as shown in Figure 1(b), where the vertices correspond to scene objects and a directed edge from A to B means that A may occlude B when viewed from the current viewpoint.

2. Estimate the rendering time and *footprint* of each object, where the footprint of an object is defined as the projection of its bounding volume on the viewport.

3. For a system with P nodes, partition the occlusion graph into P subsets such that: *(a)* if we collapse the occlusion graph so that there's a single vertex per subset as shown in Figure 2, the resulting *partition-occlusion* graph is acyclic, *(b)* the expected rendering plus transmission time of each subset is load-balanced,

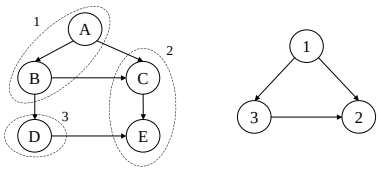

Figure 2. *A legal ILD partitioning and its corresponding partition-occlusion graph.*

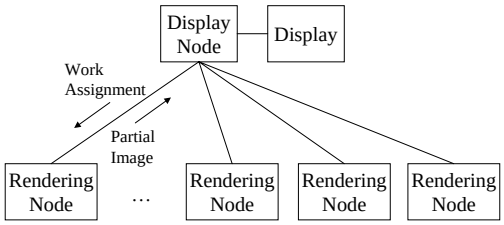

Figure 3. *System architecture.*

and *(c)* the total amount of data (pixels) that must be communicated is minimal.

To make it possible to construct an occlusion graph and partition it in real-time, we group the polygons into a relatively small number of aggregate objects. Also, we assume that the scene description is replicated on all nodes before rendering begins so that assigning work only involves the transmission of (a small number of) object IDs.

Figure 3 shows a possible architecture (used in our prototype system) for employing ILD: the node that is interacting with the user is designated the display node and all other nodes are rendering nodes. The display node is responsible for computing the current viewpoint, constructing and partitioning the occlusion graph, and assigning work at the beginning of each frame. It is also responsible for composing and displaying the rendered image at the end of the frame. Each frame, each rendering node receives its work assignment, renders the objects it is assigned, and sends the generated image back to the display node. Clearly, this architecture allows each rendering node to employ hardware rendering since the problem of generating each image layer looks exactly as if it were an independent rendering problem.

In the remainder of this section, we address five essential components of ILD: *(i)* a method for grouping polygons into aggregate objects, *(ii)* a method for constructing the occlusion graph, *(iii)* a method for estimating the rendering time of each object, *(iv)* a method for estimating the footprint of each object, and *(v)* an efficient on-line partitioning algorithm that can achieve good load balancing, taking into account both rendering and communication costs.

3.1 Constructing Aggregate Scene Objects

Because we want our approach to scale well with scene complexity, the per-frame parallelization overhead must be independent of the number of polygons in the scene. To achieve this, we group the polygons into aggregate objects, which then form the units of load assignment. In particular, we use a static octree decomposition of the scene to generate non-interleaving aggregate objects with approximately equal number of primitives.

To compute the static octree decomposition, we first compute a tight rectangular bounding volume of the entire scene. We then divide the bounding volume of the scene into 8 sub-volumes by dividing each dimension in half. If a primitive is in more than one sub-volume, we split it. Finally, we recursively subdivide a sub-volume if: *(i)* it contains more than a threshold percentage of the primitives in the scene (called the *work decomposition threshold*), or *(ii)* any dimension of a tight rectangular bounding volume of the primitives in the sub-volume is more than a threshold percentage of that dimension of the bounding volume of the scene (called the *space decomposition threshold*). The first condition is a heuristic used to limit the amount of work of each leaf octant so that no single leaf octant will dominate the rendering time of a frame (unless it is one of only a few visible octants, in which case, the frame time is likely to be small and so load balancing is not such a concern). The second condition is a heuristic used to limit the size of the footprint of each leaf octant. This is important because it is difficult to limit the overlap in the image layers generated on different nodes if leaf octants can have very large footprints. When applying this heuristic, we use the dimensions of a tight bounding volume instead of the octant itself because, in many cases, polygons contained in an octant only occupy a small portion of that octant.

3.2 Constructing the Occlusion Graph

Given the octree decomposition of a 3D model and a viewpoint, we rely on a well-known geometric property of rectangular volumes to build the corresponding occlusion graph: we can easily compute the faces of a rectangular volume that are visible to the viewer using the coordinates of the viewpoint and of the vertices of the volume [2]. Then, for any octant o, octants that are adjacent to o on faces *not* visible to the viewer must be "behind" o, and therefore, may be occluded by o[1].

More specifically, define the occlusion graph as $G = (V, E)$, where V is the set of leaf octants and E is the set of edges, where $(a, b) \in E$ if and only if a may occlude b from

[1]More accurately, primitives in these neighboring octants may be occluded by primitives in o.

the current viewpoint. To construct G, set V to include all leaf octants and E to null. Then, given the viewpoint, for each leaf octant a, if a neighbor leaf octant b is adjacent to a on a non-visible face, enter (a, b) into E.

3.3 Estimating the Rendering Time of an Object

In order to partition the overall rendering work in a load-balanced manner, we must be able to estimate the rendering loads of scene objects. One possible basis for such an estimation is the number of primitives in each object. Estimation methods based on primitive count can be very inaccurate, however, because the time required to render a set of polygons can vary widely depending on the viewpoint. Thus, we take a different approach that leverages the fact that our targeted application domain involves interactive rendering, such as performed by a VRML viewer.

In an interactive application, the viewpoint typically does not change significantly from frame to frame because the user is navigating through the scene in real-time. This implies that the rendering time of each object this frame will be about equal to the rendering time during the previous frame. (Exceptions to this include abrupt jumps to predefined viewpoints, objects coming into or going out of visibility, and crossing levels-of-detail thresholds.) For example, Figure 4(c) plots the rendering time of an object as the viewer "walks" from the viewpoint shown in Figure 4(a) to that shown in Figure 4(b). Note that while the rendering times of the object at (a) and (b) are quite different, they are (almost always) very similar in adjacent frames. Based on this observation, we use the measured rendering time of an object in the last frame as the predictor of its rendering time in the current frame[2].

3.4 Estimating the Footprint of an Object

Estimating objects' footprints in the final image of a frame is important for two reasons. First, the transmission time of an image layer may comprise a substantial portion of the load on a rendering node and so must be taken into account by the work assignment algorithm. We can estimate this required transmission time if we know (approximately) the aggregate footprint of the objects assigned to each node and the achievable bandwidth. Second, at the end of each frame, all rendering nodes send their image layers to the display node. Typically, this many-to-one communication must be performed sequentially since we assume that the display node has only one network connection. This serialization implies that ILD must strive to assign the aggregate scene objects in a way that partitions the final 2D image among the rendering nodes, or at least minimizes the overlap of the pixels they render, if it is to scale with system size. This critical optimization is only possible if we can estimate objects' footprints.

[2]The 4 points of observable discontinuity in Figure 4(c) correspond to the crossing of levels-of-detail levels – the object model has 5 levels of detail.

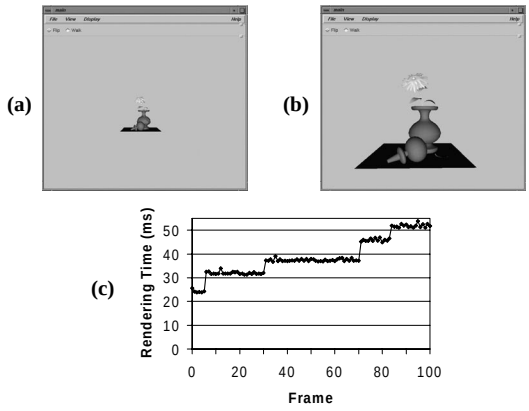

Figure 4. *(a) A VRML object viewed from far away, (b) the same object viewed close up, and (c) the rendering time of the object as the viewer "walks" from viewpoint (a) to viewpoint (b). This measurement was taken using a slightly modified version of VRweb [11] running on a 180 MHz R5000 SGI O2.*

The most accurate way to compute the footprint of each object is to determine exactly the set of pixels it paints. This approach implies parallelization overhead that is proportional to the scene complexity, though, and so is too expensive for our purposes. Instead, we use a coarser, scene-independent approach, as follows. We divide the viewport into a grid of $W \times H$ cells, where each cell corresponds to a block of pixels. (In our test cases, we use a 10x8 grid to represent a 640x480 pixel viewport.) Each frame, we calculate the projection of a tight rectangular bounding volume of the polygons in each leaf octant on the 2D grid. The footprint of an octant is then estimated as the set of grid cells that the projection overlaps.

3.5 A Greedy On-line Partitioning Algorithm

Before describing our on-line algorithm, we first consider an essential characteristic of the expected system architecture: the sequential many-to-one communication that must occur at the end of each frame (as noted in Section 3.4). To avoid the resulting idle time, we need to overlap the communication phase of a frame with the rendering of the next frame. This overlapping is supported by all hardware accelerators (that we know of) via double buffering.

Overlapping of communication and rendering means that, at the beginning of each frame, each node is already loaded with the time required to transmit the image layer

it generated for the last frame. The total (expected) load on each node for each frame is this transmission time plus the rendering times of objects assigned to it in this frame. In addition, the minimum frame time is determined by the larger of the maximum load on any node and the sum of the transmission times for the last frame. Thus, computing an ILD partitioning corresponds to a dual optimization problem: we want to minimize the load imbalance and to minimize the total transmission time.

More specifically, for a frame f, let $Partition_{P,f} = \{S_{0,f}, S_{1,f}, ..., S_{P-1,f}\}$ be a P-way partition of the occlusion graph for a system with P nodes. We define f's frame time T_f as the greater of: *(i)* the maximum expected load assigned to any one node, and *(ii)* the sum of the expected transmission time for transmitting the rendered image layers for frame $f-1$. We define the cost of a partition, $Cost(Partition_{P,f})$, as the greater of: *(i)* T_f, and *(ii)* the sum of the expected transmission time for transmitting the image layers for frame f. We include the latter component in our cost metric, as opposed to just using T_f, so that we do not compromise $f+1$'s frame time when optimizing f's frame time.

The problem is then to find a partitioning $Partition_{P,f,best}$ such that:

- *Correctness Criterion:* there does not exist $o_w, o_x \in S_{i,f}$ and $o_y, o_z \in S_{j,f}$, $i \neq j$, where there is a directed path in the occlusion graph G from o_w to o_y and a directed path from o_z to o_x, and
- *Optimization Criterion:* $Cost(Partition_{P,f,best}) \leq Cost(Partition_{P,f})$ for all partitions $Partition_{P,f}$ that satisfy the correctness criterion.

The following is a two-phase greedy algorithm that, while not optimal, works well in practice (see Section 5). The basic idea is to start from the root of the occlusion graph and construct a P-way partition sequentially, greedily keeping the expected load of each subset as close to the average load as possible. Then, we make a second pass through the partition to make corrections in the cases where the greedy heuristic either overloaded or underloaded individual subsets.

For each frame f, we start phase 1 by placing all scene objects into a set of unallocated objects, U. Then, we construct each subset $S_{i,f}$ of the partition by sequentially moving one or more objects from U to $S_{i,f}$. To meet the correctness criterion, we maintain a *frontier* of U, defined as the set of objects in U that cannot be occluded by any other object in U. Then, when choosing an object from U to allocate to $S_{i,f}$, we limit the choice to those in the frontier. This ensures that objects in $S_{i,f}$ can never occlude those in $S_{j,f}$ if $i > j$. To (try to) meet the optimization criterion, we use two heuristics: (1) as we move objects from U to $S_{i,f}$, we always choose the object (in the frontier) that would increase $S_{i,f}$'s footprint size by the least amount, and (2) we stop allocating to $S_{i,f}$ when adding any object from the frontier of U to $S_{i,f}$ would make its expected load exceed X times the average load, where X is typically slightly greater than 1.0 (we use 1.1 in our test cases).

Given the partition $Partition_{P,f,1}$ produced in phase 1, we make a second pass if $Cost(Partition_{P,f,1})$ is determined by the expected load (the first component of T_f) of some node. We simplify the optimization problem in this phase by assuming a complete ordering of the scene objects as given by the order in which they were "chosen" in phase 1 (as opposed to the original partial order given by the occlusion graph). We search for a better load-balanced partition as follows: suppose we know the optimal value B, where, for all $Partition_{P,f}$ consistent with the complete ordering, $Cost(Partition_{P,f}) \leq B$. Then, the obvious first-to-last greedy algorithm that fits as much as it can into each subset subject to the bound B finds an optimal partition. To find B, we perform a binary search in the interval $[Cost_{best,f}, Cost(Partition_{P,f,1})]$, where $Cost_{best,f}$ is defined as the frame time if we can achieve perfect load balancing.

4 Evaluating ILD's Potential

We now turn to evaluating ILD's performance. We perform this analysis in two steps. In this section, we first evaluate the impact of the restrictions underlying the general ILD approach, using an off-line ILD implementation that is based on simulated annealing [4]. Then, in the next section, we evaluate our on-line algorithm to determine how well the particular heuristic we have chosen for grouping aggregate objects into subsets of a partition (while conforming to the ILD restrictions) works.

To assess ILD's potential performance, we measure its ability to: *(i)* achieve good load balancing, and *(ii)* limit the growth in bandwidth as the number of nodes in the system grows. We evaluate ILD by comparing its achieved frame times (for a number of traces taken from a VRML viewer) to an ideal case scenario defined as the greater of: *(i)* a perfectly load-balanced partition or *(ii)* the transmission time of the footprint of the entire scene from the last frame if it were rendered on a single node. The latter represents the case when, even if there was zero overlap between the footprints of distinct subsets in the work partition, the transmission time would dominate.

We also compare ILD's performance to a variant of sort-last distributed rendering, which, as explained in Section 2, may also be appropriate to our targeted environment. In particular, we simulate a sort-last implementation that is similar to ILD, using the octree aggregation of primitives. At the beginning of each frame, we also partition the aggregate ob-

Model	# of Prims	Source
Aztec City	63933	ORC, Inc., http://www.ocnus.com
Chamber Hall	34537 22492	Lightscape Technologies, http://www.lightscape.com
Coronary Left Lung	19335 66876	UW Structural Informatics Group, http://sig.biostr.washington.edu
CS Building	56052	Dept. of CS, Millersville U, http://cs.millersv.edu/cs373.dir/vrml.dir

Table 1. *VRML models used for evaluating ILD's performance.*

Model	# Obj	Max % Work	Avg % Work	Avg RT on 1 Proc (ms)
Aztec City	190	8.4	0.6	766
Chamber	253	3.2	0.5	725
Hall	323	3.2	0.5	1413
Coronary	120	7.8	1.0	337
Left Lung	253	4.1	0.5	1278
CS Building	344	5.0	0.5	374

Table 2. *Model characteristics under an octree decomposition with thresholds of 20% (work decomposition) and 20% (space decomposition). RT is the rendering time for one frame.*

jects in a way that minimizes the overlap between subsets' footprints as well as minimizes the load-imbalance (by simulated annealing). Compared to ILD, this variant of sort-last has an advantage in that it does not have to obey visibility constraints, allowing greater flexibility to either improve load balance or reduce the overall transmission time. On the other hand, sort-last has the disadvantage that each cell takes longer to transmit than under ILD since the depth information must be sent for the composition of the final image.

4.1 Experimental Environment

As test cases, we traced paths through a number of VRML scenes using a version of the public domain VR-web viewer [11] modified to aggregate scene primitives into larger objects using our octree decomposition. We took two sets of traces, one where the work decomposition threshold and space decomposition threshold were 30% and 30% respectively, and one where the thresholds were 20% and 20%. In each trace, we recorded the dimensions of the leaf octants, and, for each frame, the position of the viewpoint and the rendering time and footprint of each leaf octant. We used a 10×8 2D grid to calculate footprints over a 640×480 pixel viewport. Traces were taken on a 180 MHz R5000 SGI O2.

Table 1 lists the VRML scenes that we use for our evaluation, the number of polygons in each scene, and the scenes' origins. Traces contain between 68 to 138 frames. Table 2 gives the number of leaf octants, the maximum percentage of primitives in any one leaf octant, the average percentage of primitives in a leaf octant, and the average frame time when rendered on a single machine for the 20/20 decomposition. Our results show that the 30/30 partitioning is not sufficiently fine for reasonable load-balancing and so we do not discuss this case further in this paper. We refer the reader to our project's web site[3] to see what our test scenes look like and the paths that we traced through them.

[3]http://www.cs.washington.edu/research/ddddrraw/

4.2 Simulation Parameters

Our simulation model has two parameters, the system size, P, and the cost for transmitting a grid cell, C_T.

We consider systems with P ranging from 4 to 16, realistic sizes for the application domain that we are most interested in. In addition, if ILD can achieve reasonable speedup, 16 nodes should be sufficient for achieving 10 fps or better for all of our VRML models.

We consider $C_T = 0, 75\mu s, 150\mu s, 300\mu s$. We consider $C_T = 0$ to assess ILD's load balancing performance, given the limitations of using (potentially large) aggregate objects and the necessity of obeying the occlusion ordering of the occlusion graph. We consider $C_T = 150\mu s$ because this is approximately the time required to send a cell in a 10×8 grid layered over a 640×480 32-bit RGBA-pixel image on a LAN with 100 MB/s bandwidth, a rate that is nearly achievable on current Gb/s LANs [12, 3]. We consider $C_T = 300\mu s$ for the case where either the image is twice as large or the network is only half as fast (50 MB/s). Finally, we consider $C_T = 75\mu s$ for the near future case where the LAN bandwidth doubles.

For sort-last, we assume a 32-bit depth buffer so that the cell transmission cost for sort-last is twice that for ILD.

4.3 Results

Figure 5 shows: *(i)* two representative graphs of the average normalized frame times under ILD and Sort-last, where the frame times achieved by ILD and Sort-last are normalized against the ideal frame times, and *(ii)* a representative graph of the average frame time speedups under ILD. Based on these results (and others in [10]), we make the following observations.

Given objects of sufficiently fine granularity such that no single object contains too much work, ILD can achieve

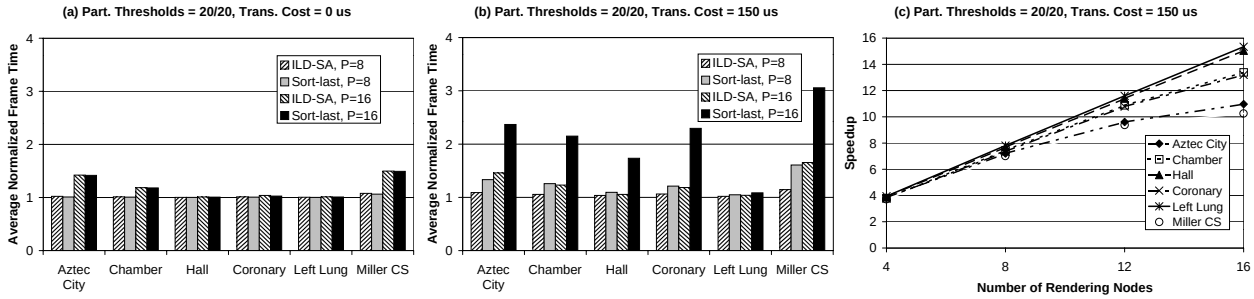

Figure 5. *(a), (b) Average normalized frame times for ILD and Sort-last for $C_T = 0\mu s$ and $C_T = 150\mu s$. (c) Average frame time speedups under ILD for $C_T = 150\mu s$.*

virtually as good load-balancing as any unconstrained partitioning algorithm. Our results show that when $C_T = 0$, ILD is always within 6% of sort-last; most often, ILD is within 2% of sort-last. This means that the visibility constraints that ILD must obey does not significantly limit its ability to load-balance. Rather, load balancing depends on not having any object whose rendering time is large enough to dominate the rendering time of a frame: the average frame times under ILD for Hall, Coronary, and Left Lung are within 5% of the ideal case. The average frame time for Chamber is within 5% of the ideal case for $P \leq 12$ but degrades to close to 20% from the ideal case for $P = 16$. Similarly, the average frame times for Aztec City and CS Building are within 6% of the ideal case for $P \leq 8$ but degrade to close to 40% from the ideal case when P grows beyond 8. Performance degrades for Aztec City, Chamber, and CS Building for large P because several large objects start to dominate the frame time. For example, Figure 6(a) plots the frame times under ILD and the ideal case scenario for Chamber when $C_T = 0$ and $P = 16$. Notice that for frames 21–35 and 41–61, ILD is significantly worse than the ideal case. This is because, in each of these frames, many objects are out of sight and so are culled. Of the remaining objects, at least one is large enough to dominate the frame time and so prevents good load balancing. For example, in frame 32, a typical frame, one of the aggregate object's rendering time is 52 ms and so determines the minimum frame time. Note, however, that because we are targeting specific frame rates (e.g., 30 fps), we do not need arbitrarily fine-grained objects.

Distributed rendering using ILD can be expected to achieve good speedup and to scale well to moderately sized clusters. Two factors affect the ability of a distributed renderer to achieve good speedup and scalability: load balance and communication time. We have shown that ILD achieves good load balancing. Figure 5(c) shows that, with a transmission cost of 150μs, on average, ILD can achieve speedups of 13.2–15.3 for Chamber, Hall, Coronary, and Left Lung on 16 nodes. Performance is less scalable for Aztec City and CS Building, topping out at 11 and 10, respectively. Both rising transmission cost and large objects contribute to this limitation on performance.

As can be expected, our results show that ILD scales inversely with transmission cost, performing better when $C_T = 75\mu s$ and worse when $C_T = 300\mu s$.

5 On-line Performance

In Section 4, we examined how the restrictions imposed by the ILD framework affect the quality of achievable partitions in the best case. In this section, we look at the performance of a practical implementation of ILD: we simulate our two-phase greedy ILD algorithm using the same set of traces and simulation parameters. Figure 6(b) and (c) show two representative graphs, one of the average normalized frame time for ILD based on simulated annealing (ILD-SA), ILD based on our greedy algorithm (ILD-OL), and Sort-last and one of the average frame time speedups under ILD-OL.

Based on these results (and others in [10]), we conclude that all observations made for the simulated annealing version of ILD hold for our greedy implementation as well. In addition, our results show that *the on-line ILD algorithm outperforms the idealized version of sort-last for transmission costs that are typical in our targeted environment.*

6 Conclusions

Increasingly, important interactive 3D applications such as virtual reality viewers and CAD systems are migrating to the commodity platform because of its low cost and rapidly increasing performance. At the same time, there is a demand for virtually unbounded rendering performance because of the desire for increasingly more accurate representations of the real world. Thus, we are motivated to build a distributed rendering system that exploits the resources of a cluster of workstations/PCs to make available real-time rendering performance that is one or two generations ahead of

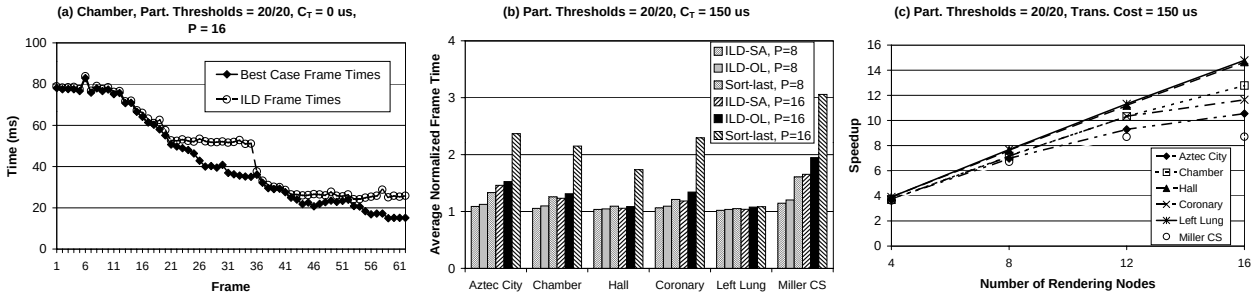

Figure 6. *(a) Frame-by-frame frame times for Chamber for $C_T = 0\mu s$ and $P = 16$. (b) Average normalized frame times for $C_T = 150\mu s$. (c) Average frame time speedups under ILD-OL for $C_T = 150\mu s$.*

what can be achieved using only a single commodity machine.

In this paper, we have addressed a critical problem in building such a system: how to partition the work to minimize the required communication bandwidth and allow the use of hardware accelerated rendering. We propose a novel partitioning technique called ILD and show that it can be expected to outperform sort-last, a commonly used partitioning technique, in our targeted environment. We give one possible approach to aggregate the many primitives of a large scene into aggregate objects. We show how to efficiently construct the needed occlusion graph given the aggregate objects and a viewpoint and give a heuristic-based on-line ILD partitioning algorithm. Finally, we show that ILD has the potential for good speedup and scalability on moderate size clusters and that our on-line algorithm can achieve much of this performance.

Acknowledgement

We thank Tracy Kimbrel and Jayram Thathachar for suggesting the partitioning algorithm that we are using in phase 2 of our on-line ILD implementation. We thank Ruth Anderson for her contribution to the discussions that culmanated in this paper. We thank Kevin Hinshaw and the UW Structural Informatics Group for providing two of our test VRML scenes. We thank ORC, Inc., Lightscape Technologies, and the Dept. of CS at Millersville for posting interesting VRML scenes on their web sites.

References

[1] T. W. Crockett and T. Orloff. A MIMD Rendering Algorithm for Distributed Memory Architectures. In T. Crockett, C. Hansen, and S. Whitman, editors, *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 35–42. ACM, Nov. 1993. Color plates on page 108.

[2] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[3] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *Proceedings of the 1999 USENIX Technical Conference (Freenix Track)*, June 1999.

[4] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[5] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3), Sept. 1996.

[6] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[7] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH '92*, pages 231–240, July 1992.

[8] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of SIGGRAPH '97*, pages 293–302, Aug. 1997.

[9] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 75–82, Apr. 1995.

[10] T. D. Nguyen and J. Zahorjan. Image Layer Decomposition for Distributed Rendering on NOWs. Technical Report UW-CSE-99-10-01, University of Washington, Department of CS&E, Oct. 1999.

[11] M. Pichler, G. Orasche, K. Andrews, E. Grossman, and M. McCahill. VRweb: A Multi-System VRML Viewer. In *Proceedings of The 1st Annual Symposium on the Virtual Reality Modeling Language*, Dec. 1995.

[12] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High-Performance Networking on Myrinet. In *Proceedings of the IPPS/SPDP '98 Workshop PC-NOW*, pages 472–485, Apr. 1998.

[13] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden Surface Algorithms. *ACM Computing Survey*, 6(1):1–55, Mar. 1974.

[14] S. Whitman. A Task Adaptive Parallel Graphics Renderer. In *Proceedings of the Parallel Rendering Symposium*, Oct. 1993.