A load-balancing strategy for sort-first distributed rendering

Frederico Abraham, Waldemar Celes, Renato Cerqueira, João Luiz Campos
Tecgraf - Computer Science Department, PUC-Rio
Rua Marquês de São Vicente 225, 22450-900 Rio de Janeiro, RJ, Brasil
{fabraham,celes,rcerq,joaoluiz}@tecgraf.puc-rio.br

Abstract

In this paper, we present a multi-threaded sort-first distributed rendering system. In order to achieve load balance among the rendering nodes, we propose a new partitioning scheme based on the rendering time of the previous frame. The proposed load-balancing algorithm is very simple to be implemented and works well for both geometry- and rasterization-bound models. We also propose a strategy to assign tiles to rendering nodes that effectively uses the available graphics resources, thus improving rendering performance.

1. Introduction

PC-based clusters have been widely used to achieve real-time rendering of complex scenes, in the place of powerful graphics workstations. While graphics workstations with shared-memory, multi-processing and multiple synchronized graphics pipelines are very costly, clusters of commodity PCs, equipped with high-performance graphics cards, can be built at a reasonable cost. Moreover, the scalability of PC-based clusters is better than that of tightly-integrated graphics workstations [11].

However, by using a cluster-based distributed rendering system, we have to face new issues, such as sharing data among different processors and exchanging rendering information [14]. Cluster-based rendering also brings the challenge of creating partitioning schemes to distribute the load among the rendering nodes and of developing communication mechanisms that scale well within the bandwidth and latency limitations of the underlying network [12].

Over the last years, the rendering performance of PC graphics processors has been increasing at a notable rate. However, despite this high-performance gain, nowadays the flexibility of modern programmable graphics processors has encouraged the development of sophisticated vertex- and pixel-based algorithms for achieving high-quality imagery. Although originally designed to be executed in real time, such algorithms, when applied to complex scenes, still demand graphics power that a single processor may not be able to deliver. Examples of such scenarios include scenes described by a dense and highly tessellated geometry set, sophisticated per-pixel lighting algorithms and volume rendering.

The purpose of our research is to investigate the use of PC-based clusters for improving the rendering performance of such complex scenes, delivering the frame rate usually required by virtual-reality applications. This goal can be achieved by combining the graphics power of a set of PCs equipped with graphics accelerators. Although PC clusters have also been used for supporting high resolution multi-display rendering systems [2, 12, 14, 13], in this paper we focus on the use of a PC cluster for improving rendering performance for a single display. We also limit the discussion to models that fit on the available memory, not dealing with out-of-core scene management.

We have opted for a sort-first architecture, where the screen is partitioned in disjoint tiles that are rendered by the different rendering nodes. The display node is then responsible for receiving the rendered tiles and composing the final image. The main advantage of such architecture is its relatively small communication requirements. On the other hand, it is very susceptible to load imbalance [8]. In order to achieve a good load balance among the rendering nodes, we propose a new partitioning algorithm. Our algorithm is very simple to implement and has a negligible running time, while bringing quite good results. We explore frame-to-frame coherence and estimate each node load based on its previous frame time. Differently from previous proposals, our algorithm deals well with both static and dynamic scenes and with both geometry- and rasterization-bound applications.

For improving the average performance and minimizing fluctuations in frame rate, we also propose a strategy for assigning tiles to rendering nodes in a way that the rendering of consecutive frames is overlapped. As a result, we are able to fully explore the graphics processing power of each node. This strategy tends to increase display lag, the time delay between a user action and its displayed results[1], but experimental results have shown this latency can be kept under acceptable limits.

This paper is organized as follows. In the next section, we describe related distributed-rendering works. Section 3 describes our proposal in detail, presenting a new load-balancing algorithm and the strategy we use to assign tiles to nodes. Section 4 presents experimental results that illustrate the efficiency of the proposed system. Finally, in Section 5, some concluding remarks are drawn.

2. Related Work

Molnar el al. [8] have classified distributed rendering based on where the visibility sort occurs: sort-last, sort-middle, and sort-first. In the sort-last architecture, each rendering node is responsible for rendering part of the scene. The display node is responsible for composing depth images to solve the visibility order. As pointed out by Mueller [9], sort-last offers excellent scalability in terms of the number of primitives it can handle and it is less prone to load imbalance. However, it requires a very high bandwidth to support the transmission of several color and depth buffers. Sort-last is also not able to easily handle scenes with transparent objects. In the sort-middle architecture, geometry processing is separated from rasterization, which makes it possible to redistribute the primitives between these two stages. Currently, this approach can only be implemented using specialized high-end workstations, thus being impractical for us. In the sort-first architecture, the screen is partitioned into non-overlapping tiles (usually with rectangular shapes) and the rendering nodes are responsible for all the rendering computation that affects their respective screen regions. These rendering nodes then send their images to the display node that is responsible for the final image composition.

One of the main disadvantages of the sort-first architecture is that overlapping primitives must be redundantly rendered by more than one tile. Sort-first scalability is limited mainly because of such overheads. Sort-first is also very susceptible to load imbalance and load balancing is one of the biggest concerns under this architecture [9]. A naïve approach to better balance the load among the rendering nodes would be to increase the number of tiles (and each node would be responsible for rendering more than one tile), but this increases the percentage of overlapping primitives, which is a direct cause of overhead [3, 6, 12].

Mueller [9] has opted for the sort-first architecture for taking advantage of the frame-to-frame coherence found in interactive applications. He has introduced an adaptive load-balancing algorithm based on a hierarchical decomposition of the 3D geometry model, thus making it adequate for geometry-bound models.

Samanta et al. [12] have proposed three different partitioning strategies for sort-first multi-projector rendering systems. Their algorithms are all based on high-level 3D primitives, being also applicable only for geometry-bound models. In order to overcome the limited scalability of sort-first architectures, Samanta et al. [11] have later proposed an algorithm for simultaneously decomposing the 2D screen into tiles and the 3D polygon model into groups, thus building a hybrid sort-first and sort-last rendering system. The proposed partitioning algorithm is also based on the 3D primitive geometry.

Nguyen et al. [10] have presented a toolkit for distributed rendering on commodity clusters, where the partitioning algorithm is based on image layer decomposition. Their strategy requires the construction of an occlusion graph that can be very expensive for scenes with moving objects (their initial proposal only deals with static scenes). Their approach also increases the rendering overload due to the need for splitting primitives in order to decompose the scene. Once the occlusion graph is obtained, it is partitioned into subgraphs, based on estimated rendering times and the pixel-area of each object. The rendering time of each object in the current frame is estimated by its rendering time in the previous frame. In order to improve the average frame rate, they have proposed the use of multi-thread for overlapping the communication of the image layers generated for a frame with the rendering of the next frame. However, they have observed that this improvement in average frame rate may only be feasible at the expense of significantly greater frame latency. They have also mentioned the importance of rasterization-bound applications and suggested the investigation of partitioning algorithms for balancing the depth complexity rather than trying to minimize primitive overlaps.

Wylie et al. [17] have opted for a sort-last architecture to build a scalable distributed rendering system based on PC clusters. By using sort-last, they were able to use relatively simple strategies to partition data for parallel rendering. Their partitioning is done based on the number of triangles, which have demonstrated in practice to have excellent load-balancing characteristics. They have also stated that for a large number of triangles the rasterization imbalance tends to become relatively small.

Different strategies for exchanging rendering information among the nodes have also been presented. Chen et al. [2] have stated that one of the major challenges for high-resolution displays is to develop scalable algorithms to partition and distribute rendering tasks effectively under the available network bandwidth. They have compared three approaches to distribute the data among the rendering nodes. In the first approach, a copy of the application runs on each rendering node; the master handles user inputs and distributes control information. In the second approach, the master is responsible for handling user inputs and distributing the primitives among the rendering nodes. The third approach is only applicable for the use of multi-projectors for high-resolution displays, because the master is responsible for the entire rendering, sending compressed final images for each server to be displayed.

Humphreys et al. [6] have presented Chromium, the successor to WireGL [5], a system for manipulating streams of graphics API commands on a cluster of workstations. One of its major benefits is the transparent support for running existing OpenGL applications on a cluster. Van der Schaaf et al. [13] have compared the use of immediate mode, such as WireGL, with retained mode rendering paradigms for scalable tiled displays. They have demonstrated that immediate mode exposes a scalability problem that they have tried to solve using retained mode. They have experimented two approaches: replicating data on the nodes and broadcasting graphics commands over the network. The replicated-data approach has achieved better performance while the broadcasting approach has eased transparency and input handling. It is worth mentioning that for reducing the network communication, Samanta et al. [12] have also opted for replicating the 3D scene on every node. For transparently managing replicated data, Voß et al. [15] have extended the OpenSG scene graph system.

Staadt et al. [14] have presented a survey of different software platforms for rendering in a multi-display environment to achieve higher resolution and better immersion. They have opted for a master-slave model as the strategy to distribute the data among the nodes. In this approach, the application runs on every cluster node. Execution must be synchronized to ensure consistency among all the application instances. Typically, the master node is responsible for handling user inputs and synchronizing state changes between nodes.

3. Proposed rendering system

We have also opted for a sort-first architecture and for replicating the model in all nodes. The model's consistency is maintained by using the dead reckoning technique as discussed by Ferreira et al. [4]. In order to achieve load balance, we propose a new partitioning scheme based uniquely on the rendering time of the previous frame. We will show that this strategy works well for both geometry- and rasterization-bound applications, while being very easy to implement. We also propose a strategy to assign tiles to nodes that effectively uses the available graphics resources. Similar to Nguyen et al. [10], we use a multi-threaded approach to parallelize operations done on the CPU, the GPU and the network.

3.1. System Overview

Our system is composed by a cluster of PCs with hardware graphics accelerators connected by a local area network, driving a single display. One cluster node (the display node) is connected to the display and coordinates the rendering among the other cluster nodes (the rendering nodes).

Our sort-first architecture can be conceptually described as follows. For each frame, the display node partitions the screen into a set of N non-overlapping rectangular tiles, which are distributed to the N rendering nodes, one tile per node. The strategy used to subdivide the screen tries to balance the load among the rendering nodes. Tile parameters are packed into a frame-tile rendering request, which is composed by the tile dimensions (width and height) and by the modelview and projection matrices, with the frustum adjusted to cover the desired screen area. Application-specific parameters, such as the current virtual time needed to update the scene, can be attached to the frame-rendering request packets. The rendering nodes then receive the request and perform the rendering. At each rendering node, a view-frustum culling algorithm is performed in order to avoid sending to the graphics pipeline primitives that do not contribute to the final tile image. Once rendering is complete, the resulting RGB components of the tile image are sent to the display node. In order to reduce network traffic, the images can be compressed before being transferred (we use the LZF library [7]). Once the display node receives all the frame tiles, it decompresses the tile images, composes the final image and displays it.

Of course, there are several issues that have to be addressed in order to effectively use the available resources, which include the CPU, the GPU and the network infrastructure. A naïve approach would misuse the resource by not fully using their capabilities.

3.2. Use of multiple threads

Because network communication is basically an IO operation, data transfer can happen in parallel with other operations. This parallelism is especially interesting for transferring large amounts of data such as the tile images. Therefore, we use a multi-threaded approach to implement the system on both the display and the rendering nodes. The rendering node's implementation has been divided into two threads: the rendering thread and the sending thread.

The rendering thread is responsible for receiving the display node request, rendering the tile, reading back the frame buffer, and sending a 'ready' message to the display node signaling that the requested rendering task is concluded. The resulting image is then placed in a queue accessible by the sending thread. After that, the rendering thread is ready to process a new display node request. The sending thread is responsible for reading tile images from the queue, compressing them, and issuing send() calls to the display node. The data transfer will only effectively take place when the display node issues recv() calls to the rendering nodes.

The display node's implementation has been divided into three threads: the receiving thread, the composing thread and the control thread. The receiving thread has a communication channel with the sending thread of every rendering node. Through this channel, it receives the compressed tile images and places them into a tile queue, signaling the composing thread that a new tile has been received.

The composing thread is responsible for displaying the final image. It gets each tile from the tile queue, decompresses it and writes it to the local frame-buffer. Once all tiles from the current frame have been received, decompressed and written, the composing thread issues a swap-buffers() call and starts composing the next frame.

The control thread is connected to the rendering thread of every rendering node. Whenever a 'ready' signal of the current frame arrives from any rendering node, this thread notifies the receiving thread that it can start receiving the tile. When all the 'ready' signals related to the current frame have arrived, this thread computes a screen partitioning for the next frame and sends the new frame tile parameters to the rendering nodes. Figure 1 shows the diagram of the message exchange between the nodes and their threads.

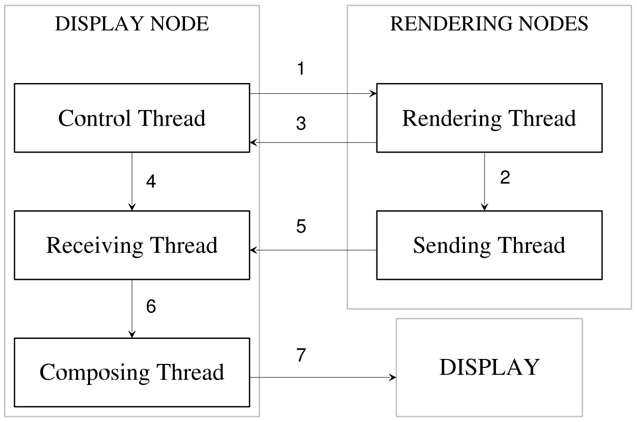


Figure 1. Message exchange between the display and rendering node threads: (1) the display node requests a frame tile rendering; (2) the rendered tile buffer is placed into a queue; (3) the display node's control thread is signaled; (4) the display node's receiving thread is signaled; (5) the frame tile buffer is sent; (6) after the tile is received, the display node's composing thread is signaled; (7) the tile is decompressed and written to the local frame-buffer.

3.3. Load balancing

The display node has to wait for all rendering nodes to conclude their rendering task before composing the final image. Clearly, the slowest rendering node represents the bottleneck of the application. In order to achieve good performance with the sort-first architecture, it is crucial to apply a partitioning scheme that tries to balance the load among the rendering nodes.

An effective partitioning algorithm should be designed to achieve different goals:

- Balancing the load among the rendering nodes;
- Minimizing primitive overlaps to avoid redundant renderings;
- Being general and effective for both geometry- and rasterization-bound models;
- Being efficiently evaluated not to impose an additional burden to the application performance.

These are conflicting goals and an optimal solution to meet all of them may not be feasible. For this reason, researchers focus on finding good heuristic methods [12]. Our heuristics takes advantage of frame-to-frame coherence and tries to balance the load based on the time each node took to render the previous frame. The partitioning is not based on the 3D geometry model; nevertheless, we will show that the resulting percentage of primitive overlaps is within reasonable limits. Its main advantages are that it is rather simple to implement, works well for both geometry- and rasterization-bound applications, and requires a negligible running time. Experiments show that the proposed heuristic algorithm results in good load-balancing among the rendering nodes.

The screen is initially subdivided into a set of N disjoint tiles, where N represents the number of rendering nodes. The algorithm uses the previous frame time to adjust the tile sizes in order to achieve load balancing for the next frame. The overall effort needed for rendering one frame is measured by summing the rendering time of all tiles. For each tile, we consider, at first, that its rendering load is uniformly distributed over the tile region. Therefore, we estimate a rendering cost per pixel within each tile. We then resize the tiles in order to have all tiles with the same amount of load, based on the previous frame. The tiles are resized by moving their boundary edges.

Mueller [9] has pointed out that natural choices to subdivide the screen include horizontal strips, vertical strips, and more rectangular shapes. Square shapes are often the preferred choice because they minimize the total region boundaries, thus minimizing the percentage of overlapping primitives. Let us first consider that the screen is subdivided into horizontal strips, with each strip representing a tile. Our algorithm computes each tile load (in fact, its rendering time) and moves the horizontal edges to achieve load balancing. For a perfect balancing, each tile should have a load equal to the overall frame time over the number of rendering nodes. By computing the load associated to each screen line, we are able to reposition the tile edges. From top to bottom, we sum the load of each line until the desired load for each tile is reached, delimiting the tile boundary edges.

In order to minimize tile boundaries, we first subdivide the screen into horizontal strips and then subdivide each strip into vertical (sub-) strips. The total number of vertical (sub-)strips, considering all horizontal strips, equals the number of rendering nodes. The load of each horizontal strip is then obtained by summing all its vertical (sub-)strips. The algorithm first repositions the horizontal strips then repeats the balancing procedure for the vertical (sub-)strips within each horizontal strip. For partitioning each horizontal strip, the rendering time per pixel of each tile must be considered, since pixels from different horizontal strips of the last frame can be present in the new horizontal strip. Pixel columns and their rendering times must be added until the desired load for each tile is reached, delimiting the tiles' vertical boundary edges. Figure 2 illustrates how we have chosen to arrange the subdivision for different number of tiles.

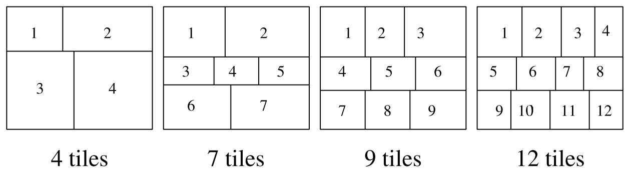


Figure 2. Subdivisions for different tile numbers.

The algorithm takes a negligible running time because its complexity is proportional to the number of tiles. By not being based on primitive counts or any other information derived from the model, it can be directly used for a variety of applications, from geometry-based scene descriptions to volume rendering. Using the past frame to compute the next screen division does not represent a severe problem. Because of frame-to-frame coherence, consecutive frames have similar load distribution, even for dynamic scenes.

On the other hand, the algorithm assumes that the rendering load of a tile is uniformly distributed over the entire tile region, which is a coarse assumption in many cases. In order to overcome this limitation, we try to explore spatial coherence by using the load-balancing algorithm only to guide the tile edge's movements. As soon as a complete frame is rendered, we run the balancing algorithm as described above, but, instead of displacing the tile edges according to the computed balanced subdivision, we use the algorithm's result to apply velocities to the tile edges. The edges then start moving in the direction of a load-balanced screen subdivision. The applied velocities should be chosen in a way to take advantage of spatial frame-to-frame coherence and to avoid severe load imbalances. In our experiments, we have used a velocity that would require 5 frames for the edges to reach the balanced screen subdivision.

The heuristic algorithm reduces the difference in rendering time among the rendering nodes, thus improving the overall rendering performance. But, of course, it is not able to ensure a perfect balance, resulting in frame-rate fluctuation. Watson et al. [16] have showed that fluctuations in frame rate can degrade the performance of interactive tasks, especially when the frame rate is low. Their experiments have demonstrated that for frame rates above 20 Hz, deviations up to 40% do not significantly affect task performance, but for applications where frame time consistency requirements are strict, frame time variations should be kept below 10%.

3.4. Tile-assignment strategy

In order to decrease fluctuations in frame rate and also to maximize the use of the graphics resources of all rendering nodes, we propose a strategy to interchange tile-node assignments.

Let us first consider the absence of a load-balancing algorithm. If the tiles are maintained as originally set (for instance, subdividing the screen in regions with the same area), due to frame-to-frame coherence we expect that the load of each tile remains similar for consecutive frames. We can take advantage of such coherence by interchanging tile-node assignments. As soon as the fastest rendering node concludes its current task, the display node can request the rendering of a tile for the next future frame. In order to improve overall performance, the currently fastest node should be requested for rendering the most demanding tile of the next frame. This most demanding tile is chosen based on the rendering time of the previous frame. When another rendering node concludes its task, it receives a request for the second most demanding tile, and so on. By interchanging tile-node assignments this way, we are able to achieve overall performance close to the one that would be achieved if the tile loads were perfectly balanced, because the rendering nodes will be always busy. Figure 3 shows a diagram which illustrates the tasks performed by each node along the time.

However, for severe load imbalances, this strategy would

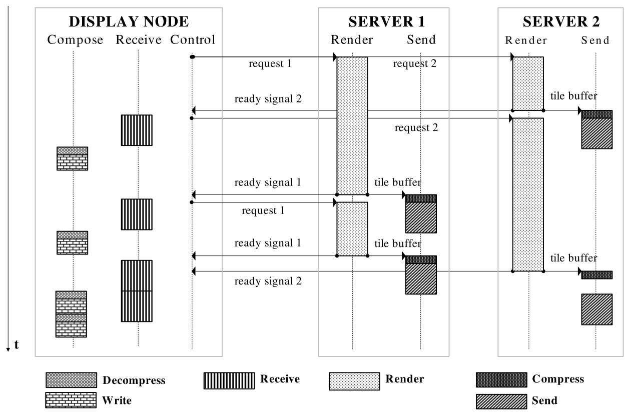


Figure 3. A temporal diagram of the tile assignment technique.

cause prohibitive latency, which would drastically degrade user performance [1]. We then combine both the load-balancing algorithm and this tile-assignment strategy to achieve better performance while keeping the latency within reasonable limits. The load-balancing algorithm avoids severe load imbalances, but still preserves tile-load coherence because the tile's edges move slowly to the balanced position.

In order to keep the latency under control, we do not allow the system to start rendering frame $i + 2$ while frame i is not complete.

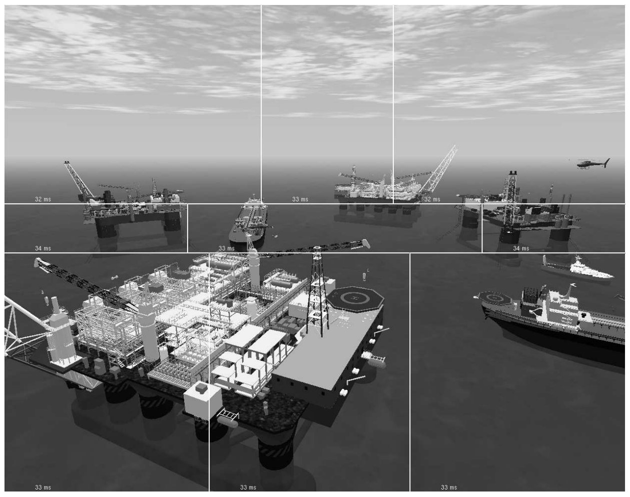
It is worth mentioning that this tile-assignment strategy only works well for homogeneous clusters, since it is expected that a given tile takes approximately the same time on any rendering node. Also, this strategy is not adequate for out-of-core scene management, because each node has to render different regions of the model (which would require bringing another part of the model to memory).

4. Experimental Results

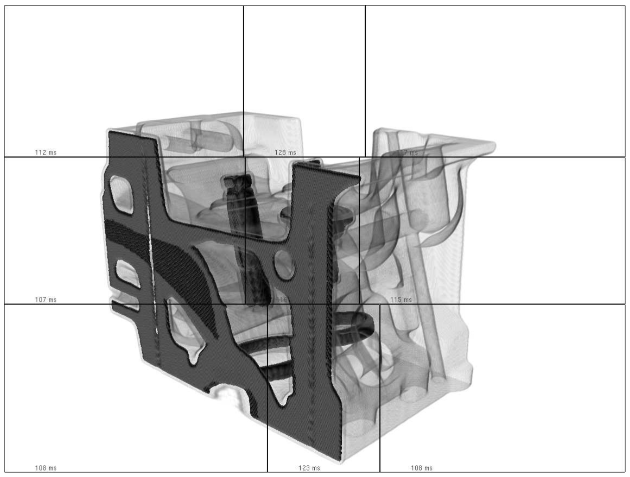
We have tested our system on a cluster composed by 10 PCs, each one with an Intel Pentium 4 1.8 GHz processor, 512 MB of RAM and equipped with a Geforce 4 Ti 4200 graphics card. The 10 PCs were connected by a switched Gigabit Ethernet network, running on Linux operating system. We have implemented the parallel rendering algorithm and the applications using C++ and OpenGL.

The distributed visualization system was tested using two different models, shown in Figure 4:

- Platforms: a geometry-bound model, composed mainly by six black-oil platforms with a big polygon count: the entire scene has 1,884,844 triangles distributed among 11,811 objects.
- VolRender: a rasterization-bound model composed by a volume data representing an engine. The engine data is stored in a 256x256x110 3D texture visualized by 379 quadrilateral slices.



Platforms: geometry-bound model.



VolRender: rasterization-bound model.

Figure 4. Models used during the tests

The Platforms model was visualized through a path that performs a navigation around the set of objects and then zooms in, from t=60s to t=75s, to have a close view of one platform. For the VolRender model, we have applied a set of rotations and translations to the engine, seeing it from different points of view.

In order to avoid limiting the performance due to network traffic, we chose a screen resolution of 600x600 for Platforms and of 800x600 for VolRender. We also turned compression off.

4.1. Load-balancing algorithm

In order to test the proposed load-balancing algorithm, we measure the load imbalance obtained for visualizing both models. As Mueller [9], we measure load-imbalance as the ratio of the maximum processor load over the average load. Mueller [9] considered a load-balance reasonable if the maximum/average load ratio was 1.5 or less. As shown in Figure 5, the proposed algorithm provided a ratio below 1.4 for both models along the entire paths, with an average value below 1.2.

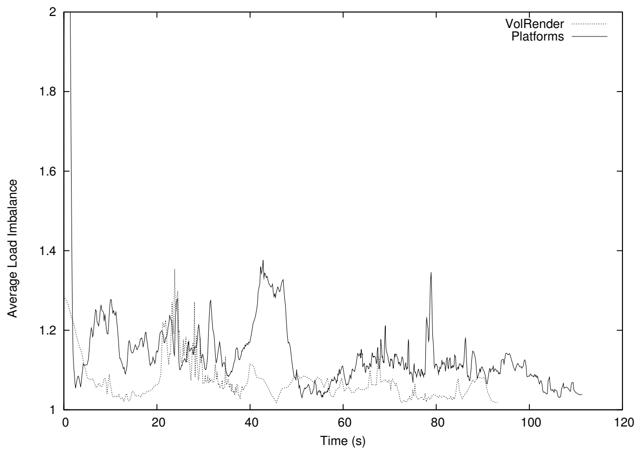


Figure 5. Load imbalances for both applications.

We then tested the system's scalability. Sort-first scalability is limited mainly due to the percentage of overlapping primitives that has to be redundantly rendered by more than one tile [3, 6, 9, 11, 12]. In order to analyze the system's scalability, we measure, for the geometry-bound model, primitive overlaps, subdividing the scene into different number of tiles (4, 16, 36, and 64 tiles). As in Molnar et al. [8], we define overlap factor as the average number of tiles a primitive overlaps. As shown in Figure 6, this factor is less than 2 along the path even for 64 tiles, except during the interval where we have a close view of a platform. Note, however, that the total number of primitives needed to visualize a close view of an object is far smaller than the total number of primitives that compose the entire scene, due to the applied frustum culling technique.

4.2. Tile-assignment strategy

For both models, we compare the frame time achieved using the cluster against the frame time using a single local machine. Figure 7 illustrate the gain for both Platforms

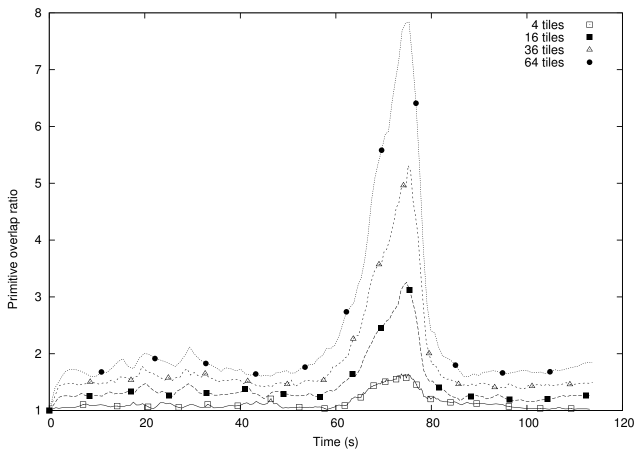


Figure 6. Primitive overlap factor for Platforms.

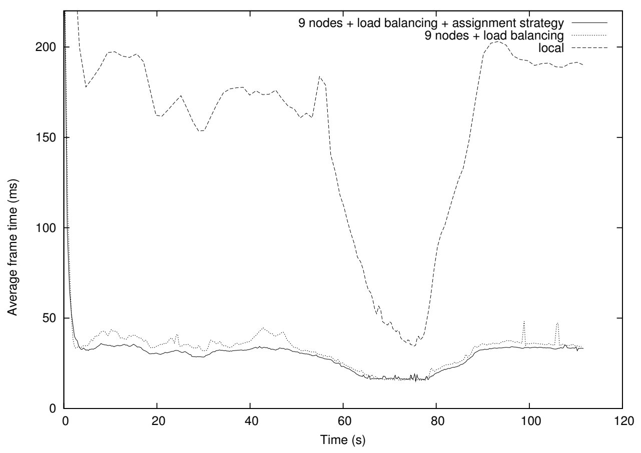
and VolRender models, respectively. While using the cluster, we also compare the performance achieved by using the tile-assignment strategy against the performance using uniquely the load-balancing algorithm. As we can note, the tile-assignment strategy provides a gain in performance and decreases frame-time fluctuations.

We have also measured the latency introduced by using the tile-assignment strategy. For both models, the additional latency, when compared to the use of the load-balancing algorithm in isolation, is negligible (average values of 2.6 ms for Platforms and of 4.8 ms for VolRender, with standard deviations of 2.4 ms and 7.6 ms, respectively).

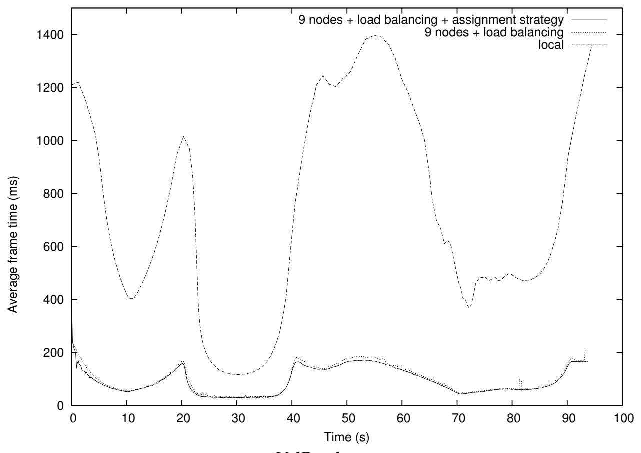
5. Conclusion

We have presented a sort-first distributed rendering system based on a PC cluster implemented using a multithreaded approach to parallelize operations done on the CPU, the GPU and the network. Experiments have demonstrated that the system provides a significant gain in rendering throughput.

We have proposed a new load-balancing algorithm for the sort-first distributed rendering system. The proposed algorithm has shown to provide good results for both geometry- and rasterization-bound models, while being very simple to implement. Combined with the load-balancing algorithm, we have also proposed the use of a new tile-assignment strategy to fully explore the graphics processing power of each node, thus improving rendering performance.



Platforms



VolRender

Figure 7. Average frame time for both applications.

6. Acknowledgements

The first author was supported by CNPq. Tecgraf is a laboratory in PUC-Rio mainly funded by PETROBRAS. We thank Marcio Pereira de Araujo and Sergio Alvares R. de S. Maffra for their help in the scene setups.

References

[1] R. S. Allison, L. R. Harris, M. Jenkin, U. Jasiobedzka, and J. E. Zacher. Tolerance of Temporal Delay in Virtual Environments. In *IEEE Virtual Reality International Conference*, pages 247–254, 2001.

[2] H. Chen, Y. Chen, A. Finkelstein, T. Funkhouser, K. Li, Z. Liu, R. Samanta, and G. Wallace. Data Distribution Strategies for High-Resolution Displays. *Computers & Graphics*, 25(5):811–818, Oct. 2001.

[3] M. Cox and N. Bhandari. Architectural Implications of Hardware-accelerated Bucket Rendering on the PC. In *Proceedings of the Eurographics workshop on Graphics hardware*, pages 25–34, 1997.

[4] A. Ferreira, R. Cerqueira, W. Celes, and M. Gattass. Multiple Display Viewing Architecture for Virtual Environments over Heterogeneous Networks. In *Proceedings of SIBGRAPI'99*, pages 83–92, Campinas, Brazil, 1999. SBC.

[5] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of the annual conference on Computer graphics and interactive techniques*, pages 129–140, 2001.

[6] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *Proceedings of the annual conference on Computer graphics and interactive techniques*, pages 693–702, 2002.

[7] M. Lehmann. Liblzf Data Compression Library. http://www.goof.com/pcg/marc/liblzf.html.

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

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

[10] T. D. Nguyen, C. Peery, and J. Zahorjan. DDDDRRaW: A Prototype Toolkit for Distributed Real-Time Rendering on Commodity Clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2001.

[11] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In *Proceedings of the Eurographics workshop on Graphics hardware*, pages 97–108, 2000.

[12] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. In *Proceedings of the Eurographics workshop on Graphics hardware*, pages 107–116, 1999.

[13] T. van der Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal. Retained Mode Parallel Rendering for Scalable Tiled Displays. In *Proceedings of the Immersive Projection Technology Workshop*, Orlando, Florida, Mar. 2002.

[14] O. G. Staadt, J. Walker, C. Nuber, and B. Hamann. A Survey and Performance Analysis of Software Platforms for Interactive Cluster-based Multi-screen Rendering. In *Proceedings of the workshop on Virtual environments*, pages 261–270, 2003.

[15] G. Voß, J. Behr, D. Reiners, and M. Roth. A Multi-Thread Safe Foundation for Scene Graphs and its Extension to Clusters. In *Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37, 2002.

[16] B. Watson, V. Spaulding, N. Walker, and W. Ribarsky. Evaluation of the Effects of Frame Time Variation on VR Task Performance. In *Proceedings of the Virtual Reality Annual International Symposium*, page 38, 1997.

[17] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable Rendering on PC Clusters. *IEEE Computer Graphics & Applications*, 21(4):62–70, 2001.