Retained Mode Parallel Rendering for Scalable Tiled Displays

Tom van der Schaaf – Luc Renambot – Desmond Germans – Hans Spoelder – Henri Bal
{tvdscha | renambot | bal}@cs.vu.nl {desmond | hs}@nat.vu.nl

Faculty of Sciences - Vrije Universiteit
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

I Introduction

The current trend in hardware for parallel graphics is to use clusters of off-the-shelf PCs instead of high-end super computers. This trend has emerged since the dramatic change in the price/performance ratio of today's PCs. Using large, high-resolution displays is another trend that is currently emerging. High resolution allows for detailed scientific visualization, and overcomes the limited screen resolution of standard monitors. Large displays also have applications in teaching environments, where multiple people (small groups or full classroom) are looking at a single large screen.

The goal of this project is to investigate the field of parallel rendering in the context of scalable tiled displays. A *renderer* calculates a 2D picture from a 3D model specified by the programmer. A *parallel renderer* uses multiple processors to calculate a single image. The target system for this parallel renderer is a so-called *tiled display*, a high-resolution screen. It consists of several projectors that compose a single, seamless image onto the screen. For this project, a cluster of off-the-shelf PCs with off-the-shelf video hardware does the rendering. These PCs control the projectors, and calculate the image in parallel. This work is based on the Aura API developed by the *vrije Universiteit, Amsterdam*. Aura is designed as a portable 3D graphics retained-mode layer for scientific visualization in virtual reality [8].

The contributions of this paper are:

- It presents a retained-mode API for 3D graphics, with a portable implementation across a variety of VR platforms (several operating systems and hardware environments),
- Two different implementations of the API are presented, with different tradeoffs in scalability and performance,
- It presents a preliminary performance study, with a focus on the network requirements of different benchmark applications.

The rest of the paper is organized as follow. Section II discusses briefly the work that has been done already in the field of rendering and parallel rendering. Next, Section III describes the important issues in parallel rendering for tiled displays. Section IV introduces WireGL, as a possible choice of parallel rendering for tiled displays. Measurements on WireGL exposed a scalability problem, caused by the fact that WireGL is an *immediate mode* renderer. Therefore, two different renderers, called Aura 'Broadcast' and Aura 'Multiple Copies', have been designed using a different paradigm, called *retained mode*. Both are described in Section V. Using several benchmarks, we compare our implementations to WireGL in Section VI, showing that *retained mode* indeed allows for better scalability.

II Related Work

Parallel rendering adds processors to the calculation to achieve higher performances. In 1994, Molnar et al described a sorting classification for parallel rendering [3]. Three different types of sorting are proposed, based on where in the rendering pipeline sorting takes place. The three modes are sort-first, sort-middle and sort-last. However, adding computing power and graphic resources is generally used in two ways. The first option is to distribute all 3D objects over all processors, and then each object is rendered by one processor. The alternative is to split the screen into a number of *tiles* and to let each processor render all objects that have to be displayed onto that tile. This approach usually causes some objects to be rendered by multiple processors, because those objects intersect multiple tiles.

Much work has been devoted to the field of parallel rendering. *The full version of this paper discuss the advantages and drawbacks of several parallel rendering systems, among them:* GLX [14], WireGL [2, 4, 5]

from Stanford University, PGL (Parallel Graphics Library) [25] from NASA Institute for Computer Applications in Science and Engineering (ICASE), Parallel Mesa Library [26] from the University of New-York, the Power Wall [21] and the Infinity Wall [22], the Distributed Graphics Database (DGD) system by Ben Schaeffer of the University of Illinois [10], and finally the different systems proposed by Princeton University [20].

III Parallel Rendering

This section discusses the general issues in parallel rendering. First the hardware is described to form the background of all that follows. Then three important design issues in parallel rendering are discussed.

1 Overview

The application (i.e. the program that uses the parallel renderer) runs on a single computer: the client. A fast network (e.g. Myrinet) connects the client to a cluster of computers: the servers or renderers. The servers are connected to a set of projectors; each server controls one projector. Every projector is responsible for 'drawing' a rectangular region of the screen (a projector-tile). Each server is responsible for assembling the final image of its projector and correcting the image for a seamless integration with its neighbors, and finally sending the image to the projector, as described Figure 1. The rendering of the image is done in parallel on all servers. The software layer correcting the images for a seamless projection is beyond the scope of this paper and consequently not addressed here.

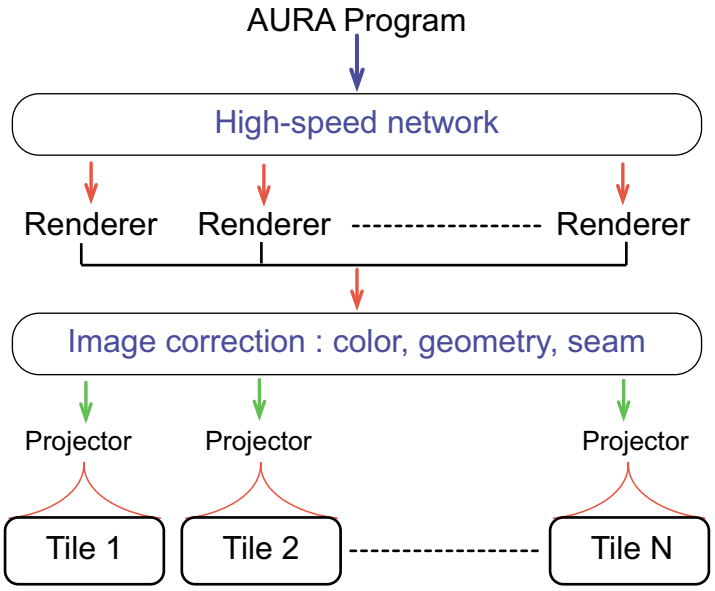

Figure 1, Parallel Rendering for Tiled Display

The following subsections contain the main implementation issues that have to be considered when building a parallel renderer.

2 The Application Programming Interface (API)

The interface of the graphics library is a critical issue. Ideally the interface is easy to use and flexible, allowing the programmer to use every feature of the underlying hardware efficiently. When making a choice for an interface, two options are possible: a new interface, designed specifically for the purpose of a tiled display, or an existing interface. The latter has the advantage that existing applications, for that API, can run on the tiled display without any changes to the source code. In fact, when using dynamic link libraries, no access to source code is necessary. The first option requires rewriting of all applications, to be used for the new API. Writing a new interface is, of course, much work, but allows the interface to be optimized for the target environment.

3 Communication

There are two main ways of communication for a parallel renderer [10]. The first approach (let us call it 'multiple copies') replicates the application to all servers, with some frustum adjustments during rendering (see Figure 2, left). These adjustments are necessary to let every server draw only a sub-frustum to form a correct viewing volume. In order to be able to do interaction (keyboard, mouse, tracker, etc…) with the user and I/O, the

client must broadcast a message that informs the other applications of the event. This way assures that every server performs exactly the same actions. In addition, to prevent temporally misaligned images, the frames must be synchronized.

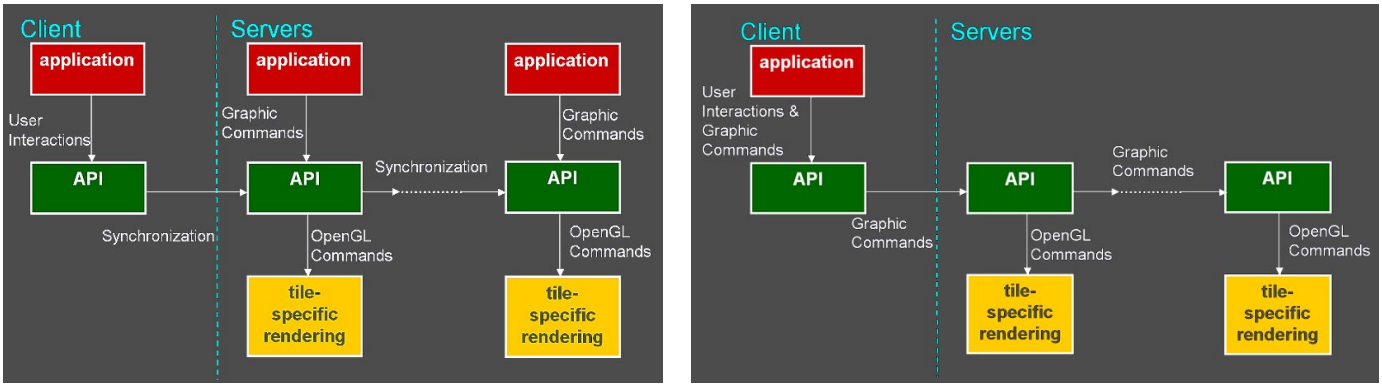

Figure 2, The 'Multiple Copies' approach versus the 'Broadcast' Approach

The other approach ('broadcast') has the application running only on the client. In this case, all graphics commands from the API are broadcast to all servers, instead of executed (see figure 2, right). Again, each server manages a sub-frustum. Synchronization between frames is needed. Although this approach is called broadcast here, it is not mandatory to use broadcast. Some objects could be sent only to those servers, which eventually draws it onto the screen. By clever use of bounding boxes, multicast, and normal point-to-point communication, it is possible to reduce the required bandwidth.

The main advantage of the 'multiple copies' approach lies in the area of performance. Since there is hardly any communication necessary (only synchronization) it runs very fast. However, there is much redundancy; all servers perform the same actions. Also the scene graph (or all scene data if there is no graph) is replicated on all servers. Multiple copies approach might be the ideal solution, but it has two main drawbacks. The first is the fact that I/O can be a big problem. If the application requires data from a file or database, all servers require that information and all must read the file or database. It is clear that multiple accesses to the same database cause a scalability problem. A second problem is that this approach is not transparent. The application must use functions provided by the API for handling user interactions (mouse, keyboard, trackers) so that these can be transferred to the servers. Also, for other functions that require synchronization, like generating random numbers, the programmer must use functions provided by the API instead of his own. Lack of transparency, albeit very little, makes this approach unsuitable for existing APIs. 'Broadcast' has the advantage that all cluster-related issues can be hidden in the implementation of the interface. This complete transparency enables the implementation of an existing API. Furthermore, I/O is straightforward for the 'Broadcast' approach. Only one application is running; only one application reads files or databases. To the outside world, the application now behaves like an ordinary application on a single node. The disadvantage is performance. Because this approach requires graphics commands to be sent over the network, it potentially eats up much bandwidth. If the application is complex enough, available bandwidth becomes the limiting factor.

Besides these two general approaches, there is also a lot to be developed on a lower level. Especially the broadcast approach requires a carefully designed communication protocol that minimizes communication overhead. Several strategies can be used for this, such as:

- Fast compression and decompression of packets to send them in a smaller size.
- Collapsing multiple packets into one, to prevent latency problems.
- Preventing the sending of data to servers that do not need that data for displaying their part of the scene.
- Choosing between function shipping and data shipping.

4 Implementation

Implementation issues are described in the full paper.

IV Experiments

This section compares the parallel renderers described Aura 'multiple copies' and Aura 'broadcast' to WireGL. First, it describes the programs used for testing. Then it discusses the results for some communication benchmark tests and the results for test with rendering.

1 The Benchmark Programs

The comparisons between Aura and WireGL in this section are based on tests with three different demo programs. All three programs are original (optimized) OpenGL programs. For usage with WireGL, these programs remained unmodified. For Aura, the test programs were all almost completely rewritten. The output of the Aura programs is, however, identical to the output of their respective OpenGL/WireGL versions. The three demos are:

- *Fire*: The fire program is a *Glut* demo. It displays a grass plain with a number of textured trees. In the center of this field, a 'fire' burns. The fire consists of a large number of triangles that fly through the air according to a complex formula.
- *Atlantis*: It displays two whales and a dolphin swimming in circles in front of the camera. In the center of the screen, a number of sharks are swimming. Like the *fire* demo, *Atlantis* is an immediate-mode program. The objects (fire or sharks) are updated every frame.
- *Cube*: It displays a large (≥ 1000) number of cubes that rotate as a whole in front of the camera. The Aura version of this program is completely retained mode. Because every cube has a different color and material it uses just as many different states as there are cubes. This proved to be a great performance hazard for the sequential Aura program.

Each benchmark can be scaled to increase scene complexity. We tested each one in two configurations: *light* and *heavy*. *Fire* and *Atlantis* have been chosen for their immediate mode nature. They should prove that both Aura versions can handle immediate mode programs and that they do not perform much worse than WireGL. In addition, *Fire* serves as an example of a worst-case program for WireGL. Due to the unfortunate implementation of *Fire*, WireGL consumes high amounts of bandwidth. *Cube* serves as an example retained mode program. It is used to show that both Aura implementations perform much better than WireGL on this type of program. The results of the testing are in the next two subsections.

2 Communication Analysis

The full paper presents a communication analysis for the described benchmarks, focusing on bandwidth requirements and scalability issues. This study is performed on a large-scale Myrinet PC cluster, without rendering.

3 Rendering Analysis

The final paper will contain full results with rendering on a small-scale cluster of PCs, cluster designed dedicated for parallel rendering: high-performance dual-cpu nodes with NVIDIA GeForce3 graphics card.

4 Discussion

The full paper presents a detailed discussion on the presented rendered ('multiple copies' and 'broadcast') versus the well-known WireGL system. The analysis focuses on issues such as performance, scalability, required latency and bandwidth, and finally ease of programming.

V Conclusion

This paper focuses on rendering for large tiled displays using an off-the-shelf cluster of PCs. The Stanford University parallel renderer WireGL is the leading renderer in the field. It has the advantage of compatibility with all existing OpenGL programs. However, its scalability is poor. The two Aura renderers presented in this paper try to solve this scalability problem, by using retained mode graphics instead of immediate mode graphics. Aura 'Multiple Copies' uses simple replication of the application and synchronization to achieve the best performance. Aura 'Broadcast' broadcasts scene data to co-operating PCs to achieve complete transparency.
Tests regarding the communication requirements of WireGL and both Aura versions show that Aura 'Multiple Copies' has by far the fastest communication and the best scalability. Aura 'Broadcast', a more flexible and transparent renderer than Aura 'Multiple Copies', performs almost as good on real retained mode benchmark

tests. On the more immediate mode oriented tests, it performs slightly worse than WireGL. Some optimizations to Aura 'Broadcast' can improve its performance.

It has never been the purpose of this paper to show that Aura is 'better' than WireGL. It tries to show that there are alternatives. For OpenGL programs, WireGL is the way to go. However, if a new 3D-visualization program is designed to run on a parallel renderer, WireGL might not be the obvious choice. If performance is an issue, Aura (or any other immediate mode renderer) might be a better choice. Aura also has a simpler, more natural interface. The choice between Aura 'Broadcast' and Aura 'Multiple Copies' also requires some thought. If the program is to run on a cluster only and does not read large files or databases from a remote location, Aura 'Multiple Copies' is the best option. Otherwise, Aura 'Broadcast' performs better.

References

[1] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107-116, August 1999.

[2] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.

[3] Michael Cox, Steven Molnar, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Algorithms*, pages 23-32, July 1994.

[4] Greg Humphreys and Pat Hanrahan. A distributed graphics system for large tiled displays. *IEEE Visualization '99*, pages 215-224, October 1999.

[5] Ian Buck, Greg Humphreys, Matthew Elldridge, and Pat Hanrahan. Distributed Rendering for scalable Displays. *SC2000: High Performance Networking and Computing,* November 2000.

[6] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. *SIGGRAPH 2000.*

[7] Thomas W. Crockett. Parallel Rendering. *NASA/ICASE Report,* April 1995.

[8] Desmond Germans, Hans J.W. Spoelder, Luc Renambot and Henri E. Bal. VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality. *IPT/EGVE 2001*, May 2001.

[9] Rudrajit Samanta, Thomas Funkhouser and Kai Li. Parallel rendering with K-Way replication. *IEEE 2001 Symposium on Parallel Graphics*, October 2001.

[10] Ben Schaeffer. A Software System for inexpensive VR via Graphics Clusters. 2000. http://www.isl.uiuc.edu/ClusteredVR/paper/DGDoverview.htm.

[11] Mason Woo, OpenGL ARB, Jackie Neider, Tom Davis, Dave Shreiner. *The OpenGL Programming Guide, Third Edition*. Addison-Wesley, 1999.

[12] E. Angel. *Interactive Computer Graphics with OpenGL, Second Edition*. Addison-Wesley, 2000.

[13] Mark Segal and Kurt Akeley. The OpenGL Graphics Interface: A Specification. 1993.

[14] Phil Karlton. OpenGL Graphics with the X Window System (Version 1.2). Silicon Graphics. 1997. http://reality.sgi.com/mjk_asd/glxspec/glxspec.html.

[15] Steve Cunningham. *Notes for a Computer Graphics Programming Course*. In development for future publication. California State University Stanislaus. 2001. www.cs.csustan.edu/~rsc/CS3600F00/FrontStuff.pdf.

[16] Andrew S. Tanenbaum. *Computer Networks, Third Edition.* Prentice-Hall, 1996.

[17] M.J. Quinn. *Parallel Computing - Theory and Practice, second edition*. McGraw-Hill, 1994.

[18] Foley, van Dam, Feiner and Hughes. *Computer Graphics – Principles and Practice, Second Edition*. Addison-Wesley, 1990.

[19] Homan Igehy, Gordon Stoll and Pat Hanrahan. The design of a Parallel Graphics Interface. *Proceedings of SIGGRAPH '98*, pages 141-150, July 1998.

[20] Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace and Kai Li. Software Environments for Cluster-based Display Systems. *In FirstIEEE/ACM International Symposium on Cluster Computing and the Grid,* May2001.

[21] University of Minnesota. Power Wall. http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[22] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. Dawe and M. Brown. The ImmersaDesk and Infinity Wall projection-based virtual reality displays. In *Computer Graphics*, May 1997.

[23] Carl Mueller. The sort-first rendering architecture for high-performance graphics. *1995 Symposium on Interactive 3D Graphics*, pages 75-84, 1995.

[24] Thomas W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23:819-843, 1997.

[25] Thomas W. Crockett. PGM: A Parallel Graphics Library for Distributed Memory Applications. http://www.icase.edu/reports/interim/29/Abstract.html.

[26] Tulika Mitra and Tzi-cker Chiueh. *Implementation and Evaluation of the Parallel Mesa Library*. Technical Report, State University of New York at Stony Brook, 1998.