Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Parallel particle rendering: a performance comparison between Chromium and Aura

Tom van der Schaaf[1], Michal Koutek[1,2] and Henri Bal[1]

[1]Faculty of Sciences - Vrije Universiteit,
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
[2]Faculty of Information Technology and Systems - Delft University of Technology

Abstract

In the fields of high performance computing and distributed rendering, there is a great need for a flexible and scalable architecture that supports coupling of parallel simulations to commodity visualization clusters. The most popular architecture that allows such flexibility, called Chromium, is a parallel implementation of OpenGL. It has sufficient performance on applications with static scenes, but in case of more dynamic content this approach often fails. We have developed Aura, a distributed scene graph library, which allows optimized performance for both static and more dynamic scenes.

In this paper we compare the performance of Chromium and Aura. For our performance tests, we have selected a dynamic particle system application, which reveals several issues with the Chromium approach of implementing the OpenGL API. Because our distributed scene graph architecture was designed with a different approach, the test results will show that it performs better on this application.

1. Introduction

The areas of high-performance computing and scientific visualization are going through many technological changes. In both areas, there is a trend from expensive high-end machines (supercomputers and high-speed graphics systems) to more cost-effective solutions (e.g. cluster computers, PC graphics cards, and tiled displays). Today, research projects often use an infrastructure consisting of many different visualization devices and parallel computers, all interconnected. For example, the Dutch VL-e (Virtual Laboratory for e-Science) project makes use of a CAVE, scalable tiled displays, a workbench, Personal Space Stations, several cluster computers and graphics machines, as well as a large number of common Windows and Linux workstations (Figure 1). Within the VL-e project, there is a need for distributed simulation as well as distributed visualization.

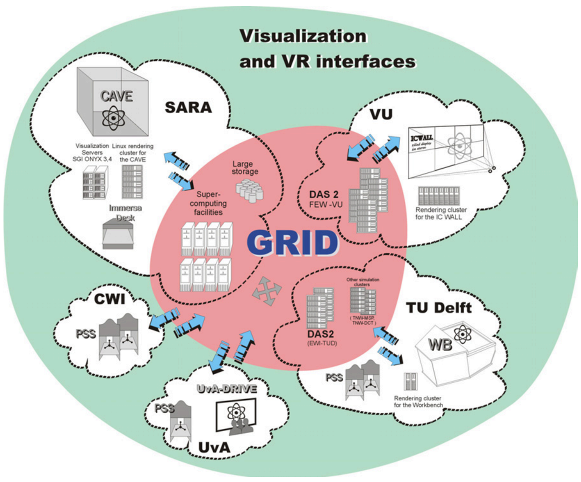
Visualization
and VR interfaces
CAVE
SARA
Visualization Servers SGI ONYX 3,4
Linux rendering cluster for the CAVE
Immersa Desk
Large storage
Super-computing facilities
VU
ICWALL tiled display in stereo
DAS 2 FEW -VU
Rendering cluster for the IC WALL
GRID
CWI
PSS
Other simulation clusters
(TNW-MSP, TNW-DCT)
DAS2 (EWI-TUD)
TU Delft
WB
PSS
Rendering cluster for the Workbench
UvA-DRIVE
PSS
UvA

Figure 1: *The VL-e infrastructure*

In the recent years we have been developing a scene graph library called **Aura**, with the specific purpose of coupling running parallel simulations to graphics clusters that render the simulation data. **Chromium** is a popular parallel implementation of OpenGL, that focuses on similar infrastructures. In this paper, we will evaluate the performance of Aura and Chromium using a parallel application that renders a large set of particles. In this evaluation, Aura has a significantly higher performance in applications with large numbers of moving particles, even if the Chromium program uses OpenGL display lists properly.

The contributions of this paper are:

- We will describe several problems that occur when rendering large particle systems using Chromium: the high CPU-load caused by handling the large amount of OpenGL functions, the large amount of data sent per particle and the lack of server side culling.
- We show how our implementation of Aura differs from the Chromium approach, to prevent these problems.
- We implemented the same particle system application for both the Chromium API (OpenGL) and the Aura API. Several test cases with different configurations for this application show that Aura does not have the problems Chromium has when rendering particles. As a result, the Aura application performs better.

2. Related Work

Stanford University Computer Graphics Lab have performed much research in the field of parallel graphics [BHH00, HBEH00, HEB*01]. They designed an implementation of the OpenGL API that transparently distributes the rendering over multiple nodes. This library, called Chromium [HHN*02], replaces the normal OpenGL dynamic link library (DLL) on the client by its own library. This new library, instead of executing the OpenGL commands, can wrap the commands in packages and sends them to the appropriate rendering servers. There, the commands are executed as normal. Chromium can allow multiple nodes to perform OpenGL rendering into the same display or window, provided the user properly synchronizes the commands. Unfortunately, the low-level rendering commands often require large amounts of bandwidth, as they are all sent over the network. More specifically, unless a program uses OpenGL display lists, it usually results in suboptimal performance for Chromium. To allow parallel programs to draw to rendering clusters, an open source scene graph library called OpenRM was adapted to work with Chromium [BHPB03]. OpenRM uses display lists whenever possible. However, tests with visualization of complex molecules will show that even when programs use display lists properly, the performance of Chromium can still be suboptimal (see Section 5).

Another similar project exists at Princeton University [CCC*01]. The project investigates several different approaches to parallel rendering. One such approach involves DLL replacement similar to Chromium. Two other approaches focus on synchronizing copies of the same program running in parallel. The last approach is a virtual display driver, which can transparently distribute a single desktop over multiple PCs but is not suited for hardware accelerated 3D rendering.

OpenSG [VBRR02] is a graphics API with a focus on providing a multi-thread safe scene graph that can be extended to clusters. However, the focus of this project is parallel rendering of single node applications and not parallel simulations. Also, the burden placed on the programmer is high, as the API is inherently multi-threaded, which can be difficult to program. Another problem is the need for the programmer to provide separate implementations for all processes taking part in the rendering (for example, the application client and the rendering server).

VR Juggler [BJH*01] is an API developed at Iowa State University, that can be used to create VR-applications in a platform independent way. VR Juggler supports many platforms, ranging from CAVEs running a SGI-Onyx cluster, to common desktop workstations. Applications can also run on clusters by using Net Juggler on top of Chromium. However, there is no direct support for visualization of parallel simulations.

3. Aura

The Vrije Universiteit developed an API called Aura [GSRB01, dSRG*02, dSSKB06] originally intended for multi-platform Virtual Reality. The current focus of this project is on interactive and collaborative visualization of parallel simulations onto a multitude of displays, such as tiled displays, CAVEs and regular desktop workstations. Our approach differs from those at Stanford and Princeton in that we do not attempt to allow arbitrary existing programs to perform parallel rendering. Instead, we focus on designing an API that can be used for writing new high performance parallel graphics applications.

The goal of Aura is to support rendering on high-end platforms (e.g. graphics supercomputers or clusters of PCs with modern graphics cards) as well as low-end platforms (e.g. desktop PCs). Furthermore, it should be possible to connect the system to any number of computing nodes. This makes it possible for complex simulations running on a parallel computer to visualize their data on a (graphics dedicated) remote computer or cluster.

A key issue in the performance of a system that couples simulations to visualization clusters is communication. For many applications, large amounts of data must be shipped from one cluster to another. The system should try to reduce the amount of data sent and should take into account different types of networks between and within the different clusters.

In order to bring structure to the different systems that Aura supports, we distinguish four stages of the visualization pipeline, as shown in Figure 2. The first stage of the pipeline is the source data that requires visualization. The data can originate from a range of sources, such as files, simulations or databases. The second step, filtering and mapping (which we will refer to as mapping), filters the data and converts it to a graphical representation, usually a scene graph. The next stage interprets this graphical representation and performs the actual rendering into an image. In the last stage, the pixels of the image are transmitted to the actual display device.

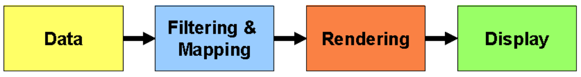

Figure 2: *The visualization pipeline*

Each stage in Figure 2 can be performed on a different physical location and each stage can be performed by multiple devices in parallel. For example:

- The data can be calculated by a parallel simulation.
- The data coming from this simulation can be distributed over multiple mapping nodes.
- The graphical representation of the data can be rendered on a rendering cluster.
- The image from the cluster can be drawn on a tiled display consisting of multiple projectors or flatscreen panels.

A scientist who wants to visualize data is mostly interested in the mapping stage of the pipeline, i.e. defining what the data should look like. Preferably, the mapping method should work with any configuration of parallel or sequential rendering, display and simulation stages. The system we present provides scientists with a scene graph API for writing this mapping stage, that hides all the details about the nature of the next stages: rendering and display. At the same time, it provides tools that make it easier to access the data from the previous stage.

When implementing the mapping stage of a visualization, the scientist first needs to access the (possibly distributed) data source. In order to simplify this task, Aura provides several tools such as file loaders and a parser to read simulation output. After the data has been loaded, the scientist applies visualization tools (such as the Visualization Tool Kit [SML99]) to convert the data to its graphical representation in the form of an Aura scene graph and submits it to the system. Whenever the source data changes, the graph is modified accordingly. How this scene graph is processed depends on the specific configuration of the underlying system and is not the scientist's concern.

In the remainder of this paper, we will refer to the nodes that perform the mapping stage of the pipeline as **masters** and nodes that perform the rendering stage as **slaves**.

Due to the hybrid nature of the target platforms of Aura and the many possible levels of parallelism, much flexibility is required in the API. We use a modular approach to facilitate this flexibility. A detailed description of the design of the API can be found in [dSSKB06].

4. Aura and Chromium

The largest bottleneck in performance of parallel rendering is often network bandwidth. Therefore it is useful to make a classification of graphics applications in terms of the amount of data that changes each frame, as it is this data that must be transmitted to the rendering nodes.

- **Static** applications have only a small amount of scene data that changes each frame, typically only the camera position changes. Examples of this category are: rendering a single pre-calculated iso-surface and visualization of static objects like buildings.
- For **dynamic** applications most or all scene data is modified each frame. Examples of this category are: rendering time-varying data sets using iso-surfaces or 3D textures.
- Finally, a large group of applications is **hybrid**, as it has both static and dynamic content. Examples of this category are: molecular dynamics visualization and games like Quake.

OpenGL based approaches such as Chromium [HHN*02] often experience problems, because the API requires that the complete scene is specified each frame, effectively making all applications dynamic. For applications with static scene data, many of these commands will contain redundant information. An important concept in reducing redundancy for Chromium applications is the OpenGL display list. A **display list** is a group of OpenGL commands and arguments that has been stored for subsequent execution and is represented by a unique number. Chromium can make efficient use of this construct by transmitting the command sequence and associated number when the list is constructed. Later, when the application requests for the display list to be rendered, Chromium can send only the number rather than the entire set of commands. Therefore, display lists are essential in reducing the amount of data sent over the network.

Using display lists, Chromium can very efficiently render static applications. Aura, with its scene graph approach, performs equally well on static scenes. Dynamic applications are very hard to parallelize, because of the large amounts of data that must be transmitted. Both Aura and Chromium will not perform well on such applications, although [dSSKB06] gives an example of how Aura can be extended for a particular application to get good performance in that domain. In this paper, we examine a test case of a hybrid nature: a large particle system. This application consists of a large amount of simple geometries (the static data), each moving in a separate direction (the dynamic data).

Even when the programmer of an OpenGL application is properly using display lists, problems can arise when the application is hybrid and has a large part of dynamic data. For example, the program described in Section 5, which calls over 300,000 display lists per frame, performs poorly using Chromium. Each frame, every display list call must be handled separately and a message must be transmitted to the

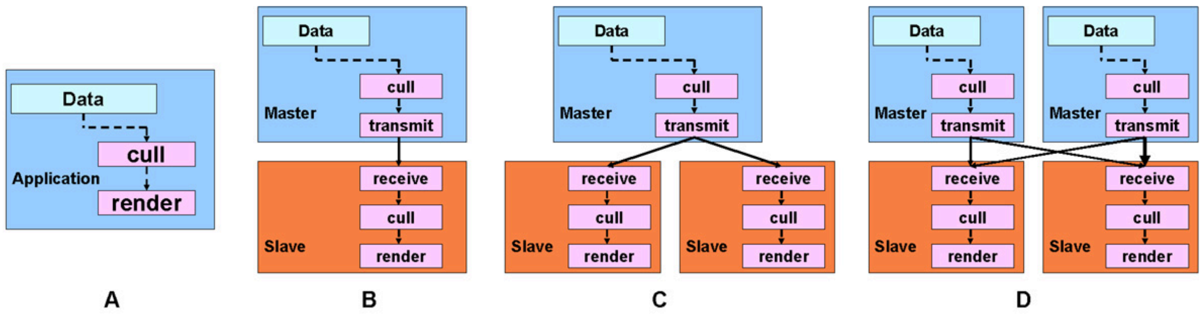

Figure 3: *The configurations used for testing*

nodes responsible for rendering it. Even though these individual messages are small, for a program that uses many simple display lists the overhead is significant. The overhead is caused by both network bandwidth and processing time per function call. Since display list objects normally appear for many frames in a row, it would be more efficient to send messages only in those frames where they appear first or disappear again. The latter cannot be implemented efficiently within the OpenGL API.

In our approach, we choose to transmit scene graph data instead of lower level graphics commands. This allows our implementation to use information from the scene graph itself to reduce communication by completely removing redundant messages. Aura only transmits the data in the graph that actually changed. In case of the particle example, the position data of the particles is stored in a single array. Aura keeps track of which positions have been updated and sends only the dirty positions. The application tested in Section 5 updates every position during each time-step, which means only a single copy of the array is required to put the data on the network. We used several configurations with different numbers of master and slave nodes. Figure 3 shows several of these configurations.

Figure 3A shows the application as it runs sequential on a single node. Figure 3B is a case of remote rendering, where a single master describes the scene, which is transmitted to a single rendering slave. Figure 3C shows multiple slaves rendering the scene data of a single master. Finally, Figure 3D describes the situation in which multiple masters define parts of the scene which is rendered by a group of rendering slaves.

5. Test Cases and Measurements

We performed several case studies to validate Aura and to analyze the performance of the system. This section will describe one of these test cases. The other test cases can be found in [dSSKB06]. All tests were performed on the ICWall [Vri] cluster, consisting of 9 dual Athlon 1.33 Mhz PCs, each equipped with a Geforce 4 Ti4600 videocard. The first set of tests is performed through a regular 100 Mbit Ethernet connection. The second set of tests uses Myrinet [BCF*95] with an effective throughput of over 70 Megabyte per second.

Visualization of an particle systems (e.g. N-body applications) is often a challenging problem. First of all, it usually involves displaying a large amount of graphical objects, which is a heavy burden on the graphics hardware. Furthermore, these graphical objects are highly dynamic, as their positions change with every time-step of the simulation. The latter is challenging for remote or parallel rendering, as it will result in high bandwidth consumption.

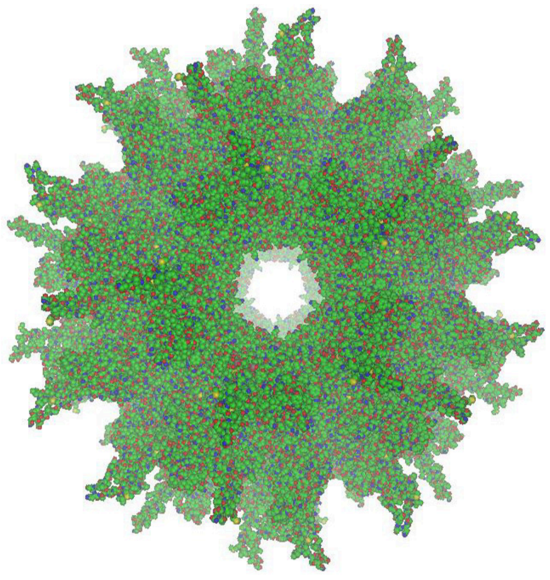

Figure 4: *Bacteriophage Alpha3*

As a case study, we implemented a simple molecular dynamics visualization, that loads a molecule in the

PDB [BWF*00] format. The implementation uses a number of masters that all use the same source code, each loading a different subset of atoms from the molecule. Since we focus on the visualization of the molecule, our simulation performs a random jittering of each atom during each timestep, rather than calculating actual force, acceleration, velocity and position. In the future, we will couple our system to real Molecular Dynamics (MD) simulation packages, such as Gromacs [BvdSvD95], NAMD, or Amber. For this case study we chose a large PDB description of a bacteriophage consisting of over 300,000 atoms (Figure 4). Each atom is represented by a sphere, built up using 60 vertices (all lit with both specular and diffuse light), without any other level-of-detail optimizations.

To compare the results with Chromium, we re-implemented the application using the OpenGL API. The program is optimized for use with Chromium by using different display lists for the different atom types present (7 in total). This display list contains the geometry, color and proper scaling for a given atom. Listing 1 shows the main loop of this program. Note that it is possible to optimize this implementation for Chromium by writing dedicated vertex and fragment shader programs. For example, using vertex and fragment shaders to render atoms as point sprites, would likely improve the performance of Chromium on the tests presented in this paper. Unfortunately, these optimization are not possible on our current hardware. In the near future, we will repeat our experiments on more recent hardware and also include shader implementations in our performance comparison. In any case, such low level hand-tuning is not required for Aura programs to have good performance in parallel, reducing application developement time significantly. The source code shown in Listing 2 shows the corresponding loop for Aura.

Listing 1: *Main loop of the OpenGL version of the MD program*

```
for (i = 0; i < nr_of_atoms; i++)
{
    glPushMatrix();
        glTranslatef(atom_position[i]);
        glCallList(display_list[i]);
    glPopMatrix();
}
```

Listing 2: *Main loop of the Aura version of the MD program*

```
for (i = 0; i < nr_of_atoms; i++)
{
    molecule->
        SetParticlePosition(i,
            atom_position[i]);
}
```

We will present several test cases, using different networks. For the first test case, we will run the application with 1 master and different numbers of slave nodes to analyze scalability and bandwidth requirements using Ethernet. The second test case will use multiple masters and multiple slaves, to investigate the effects of the reduced CPU load and the better network utilization that can be achieved when the application is distributed. The third test case will show the results on the faster Myrinet network, to show that Aura is network bound, while Chromium is CPU bound.

As a basis for comparison in all test cases, we measured the sequential performance of the application for the Aura application, the OpenGL application and the OpenGL application running with Chromium on a single node. When multiple slaves are used, the resulting image has a resolution that is higher than the sequential image, proportional with the number of rendering nodes. The scene is rendered such that the load of the rendering nodes is well balanced. For Chromium, the *tilesort* Stream Processing Unit (SPU) is used to distribute the data over the slave nodes. Because Chromium SPUs have many configuration options, for every test run we tried all relevant configurations (in case of tilesort these are the bucket mode and various display list options) and selected the test that gave the best performance.

5.1. Ethernet

In the first test case, we compare the speed of Aura and Chromium when running the visualization part of the application (master) on a single node. The rendering part of the application (slave) is performed on the same node, 1, 2 and 4 remote nodes respectively. Table 1 shows the number of frames per second (fps) for the Aura program and the OpenGL program running on Chromium in this test case. The performance of the OpenGL program using the normal OpenGL library is included to serve as reference.

Configuration	Aura (fps)	Chromium (fps)	OpenGL
single node	1.69	0.9	1.7
1 master 1 slave	1.0	0.15	N/A
1 master 2 slaves	0.8	0.1	N/A
1 master 4 slaves	0.8	0.05	N/A

Table 1: *Comparing framerates on the MD-simulation of Aura and Chromium over ethernet*

Configuration	Aura (MB/s)	Chromium (MB/s)
1 master 1 slave	3.4	3.1
1 master 2 slaves	6.1	3.8
1 master 4 slaves	11.5	4.4

Table 2: *Comparing throughput on the MD-simulation for Aura and Chromium over 100 Mbit Ethernet*

The first row of Table 1 shows that the OpenGL program using Chromium reaches only half the performance of the OpenGL program using the native library. The main cause of this slowdown is the overhead introduced by processing the individual OpenGL calls and communicating them to the server process (Chromium uses two processes even when running on a single machine; these processes run without interference on the dual-core test machine).

When not running on a single node, both Aura and Chromium perform worse than the sequential program. For Aura, the lower performance is caused by the fact that in single threaded mode, slaves have to wait for the next set of position updates. Since the amount of communication increases with the number of slaves, the performance decreases. Chromium is significantly slower than Aura on all runs.

Table 2 shows that Chromium does not come close to filling the available 12.5 MB/s (100 Mbit) bandwidth in any of the runs. This is caused by the fact that for Chromium, the performance this application is CPU bound. Chromium needs to process each OpenGL function call individually: extracting the relevant data, determining where to send it and adding it to the transmission buffer. Because the overhead of this processing is large compared to the work it creates for the slave nodes, this has a negative impact on performance. Listing 1 shows that each atom requires 4 OpenGL calls to be packed and sent over the network. Aura on the other hand, stores the data associated with the atoms in a single large buffer, which is sent as a whole. This means that there is only little overhead per atom while preparing the data for transmission. As stated before, the overhead for Chromium could be reduced by implementing an application specific vertex and fragment shader pair.

Besides being CPU-bound, another problem is the amount of data required to send the four OpenGL calls. For Chromium we measured that it sends around 70 bytes per atom. The only data that change between frames are the atom positions, consisting of only 12 bytes (3 floats of 4 bytes each). Therefore, Chromium sends 58 bytes of redundant information per atom, whereas Aura has no such overhead, as it sends the minimum of 12 bytes per atom.

Configuration	Aura (fps)	Chromium (fps)	OpenGL
single node	1.69	0.9	1.7
1 master 1 slave	1.0	0.15	N/A
2 masters 2 slaves	0.9	0.19	N/A
4 masters 4 slaves	1.0	0.21	N/A

Table 3: *Comparing parallel MD-simulation of Aura and Chromium over ethernet*

To better utilize the network capacity and reduce CPU load, we performed a second test case, in which the MD-simulation is distributed over multiple nodes. Instead of using one master to run the application, we increase the number of masters at the same rate as the number of slaves. Table 3 shows the results of this test case. Again we see a significant difference between Aura and Chromium. In this test case, the performance Chromium does improve, because with the increasing number of masters, the CPU load is reduced.

5.2. Myrinet

From tests with a faster network (see Table 4) we can conclude that Aura is mainly bound by network bandwidth, as it gains significantly from the higher throughput of Myrinet. Chromium does not gain as much, since it suffers from the overhead for handling OpenGL function calls and is mainly processor bound. Unfortunately, a bug in the Chromium code (responsible for communication over Myrinet) prevented us from running tests with more than 3 nodes. However, the results from the few measurements that did succeed confirm that Chromium does not benefit much from a faster network.

Configuration	Aura (fps)	Chromium (fps)	OpenGL
single node	1.69	0.90	1.7
1 master 1 slave	1.46	0.20	N/A
1 master 2 slaves	1.80	0.13	N/A
1 master 4 slaves	2.43	-	N/A
4 masters 4 slaves	3.21	-	N/A

Table 4: *Comparing parallel MD-simulation of Aura and Chromium over Myrinet*

The performance of Aura over Myrinet shows a speedup when compared to the sequential OpenGL program. Due to the faster network, performance is not limited as much by bandwidth and speedups can be gained. An important factor for this speedup is culling on the render nodes. Particles that aren't visible need not be sent to the graphics card for rendering. Therefore, performing culling on the molecule can increase rendering performance significantly. Another approach to culling, would be to perform culling on the application node. This method of calculating where on the screen an atom will end up, has the advantage that we know which slave will have to render the atom before sending its data updates, as this information can be used to prevent unnecessary communication.

All the Chromium test results we described for this application, were performed with the bucketing mode 'broadcast'. Bucketing mode broadcast means that Chromium will always transmit all data to all slaves and performs no culling whatsoever. We tried all other configurations, but performance was always worse than 'broadcast'. The most promising of these other configurations is the bucketing mode 'regular grid', which assumes a simple grid alignment of the slave tiles and sends the display list commands only to those slaves that actually render it. However, in all our tests, the cost of calculating which slave will render a particular atom was more than the cost of simply broadcasting it to all slaves.

We expect that with higher numbers of slaves than tested here, the performance of 'regular grid' approach will overtake the performance of 'broadcast'.

In Aura, it would be possible to improve the frame rate of the dynamic PDB to up to a measured 6 frames per second by running the rendering and the scene graph updating stages in different threads. This way the frame rate can be close to that of the static application while the atom movements will be applied on a background copy of the scene graph. However, our focus in these measurements is on the update rate of the molecule and not on the interactivity of the application.

6. Conclusions and future work

We described a performance evaluation of Chromium and Aura on an application with high amounts of independently moving particles. Several test cases reveal problems within the Chromium approach for this type of application:

- Overhead of handling OpenGL function calls becomes a bottleneck for the application, making the application strongly CPU-bound for Chromium . Aura has an entirely different approach, dealing only with scene graph updates. This approach has much less processing overhead, making the application less CPU-bound. It would be possible to reduce the overhead for Chromium by utilizing low-level features of modern graphics cards. This does require a significant amount of hand tuning, something that is not necessary when writing Aura applications.
- Because of the redundancy within the OpenGL API, the amount of data transmitted per atom is more than five times larger than for Aura.
- If (display list) culling is enabled, Chromium performs this on the master, which leads to extra processing overhead per atom. For an application that is already CPU bound, this mostly hampers performance. Aura performs slave-side culling. This allows significant reduction of particles being sent to the graphics hardware and can even result in speed-ups when using faster networks.

We believe Chromium is a good solution for many applications (especially if they are static) and the only solution if an existing OpenGL application must be parallelized. However, our measurements show that there exist applications that do not fit the Chromium approach very well. For such applications, an alternative approach, such as our approach with Aura, can yield faster performance with less programming work involved.

Our future work will include:

- A case study using a real parallel MD-simulation, which will be more computation intensive than the simple test case presented in this paper. For this case study we will also use newer hardware with support for vertex and fragment shader programs, to get fairer performance results for Chromium.
- Perform experiments with different types of visualization applications, such as: parallel volume rendering, parallel terrain rendering and visualization of time varying isosurfaces.

7. Acknowledgements

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&;W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ).

References

[BCF*95] BODEN N. J., COHEN D., FELDERMAN R. E., KULAWIK A. E., SEITZ C. L., SEIZOVIC J. N., SU W.-K.: Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro 15*, 1 (1995), 29–36.

[BHH00] BUCK I., HUMPHREYS G., HANRAHAN P.: Tracking graphics state for networked rendering. In *Proceedings of the 2000 Eurographics Workshop on Graphics Hardware* (Aug. 21–22 2000), Spencer S. N., (Ed.), ACM Press, pp. 87–96.

[BHPB03] BETHEL E. W., HUMPHREYS G., PAUL B., BREDERSON J. D.: Sort-first distributed memory parallel visualization and rendering. In *IEEE Symposium on Parallel and Large-DataVisualization and Graphics (PVG)* (Oct. 20–21 2003), IEEE Computer Society Press.

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proc. IEEE Virtual Reality '01,* (2001), pp. 89–96.

[BvdSvD95] BERENDSEN H., VAN DER SPOEL D., VAN DRUNEN R.: GROMACS: A message-passing parallel molecular dynamics implementation. *Comp. Phys. Comm. 91* (1995), 43–56.

[BWF*00] BERMAN H. M., WESTBROOK J., FENG Z., GILLILAND G., BHAT T. N., WEISSIG H., SHINDYALOV I. N., BOURNE P. E.: The protein data bank. *Nucleic Acids Research 28*, 1 (2000), 235–242.

[CCC*01] CHEN Y., CHEN H., CLARK D., LIU Z., WALLACE G., LI K.: Software Environments for Cluster-Based Display Systems. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)* (May 2001).

[GSRB01] GERMANS D., SPOELDER H. J., RENAMBOT L., BAL H. E.: VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality. In *5th Immersive Projection Technology Workshop* (May 2001).

[HBEH00] HUMPHREYS G., BUCK I., ELDRIDGE M., HANRAHAN P.: Distributed rendering for scalable displays. In *SC2000: High Performance Networking and Computing. Dallas, TX, USA* (2000), ACM Press and IEEE Computer Society Press.

[HEB*01] HUMPHREYS G., ELDRIDGE M., BUCK I., STOLL G., EVERETT M., HANRAHAN P.: WireGL: A scalable graphics system for clusters. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 129–140.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream processing framework for interactive rendering on clusters. In *SIGGRAPH 2002, Computer Graphics Proceedings* (2002), Annual Conference Series, ACM Press / ACM SIGGRAPH.

[dSRG*02] VAN DER SCHAAF T., RENAMBOT L., GERMANS D., SPOELDER H., BAL H.: Retained mode parallel rendering for scalable tiled displays. In *6th Immersive Projection Technology Workshop* (Mar. 2002).

[dSSKB06] VAN DER SCHAAF T., SMIT F., KOUTEK M., BAL H.: Aura: a flexible distributed scene graph API for visualization of parallel simulations. *under preparation* (2006).

[SML99] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: *The Visualization Toolkit*, 2nd ed. Prentice Hall PTR, 1999. http://public.kitware.com/VTK.

[VBRR02] VOSS G., BEHR J., REINERS D., ROTH M.: A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002* (2002), ACM Press, pp. 33–37.

[Vri] VRIJE UNIVERSITEIT, AMSTERDAM: ICWall Tiled Display. http://www.nat.vu.nl/~tvdscha/icwall/.