Broadcast GL: An Alternative Method for Distributing OpenGL API Calls to Multiple Rendering Slaves

Tommi Ilmonen
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
Tommi.Ilmonen@tml.hut.fi

Markku Reunanen
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
marq@tml.hut.fi

Petteri Kontio
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
jpkontio@tml.hut.fi

ABSTRACT

This paper describes the use of UDP/IP broadcast for distributing OpenGL API calls. We present an overview of the system and benchmark its performance against other common distribution methods. The use of network broadcasts makes this approach highly scalable. The method was found effective for applications that need to transmit changing vertex arrays or textures frequently.

Keywords
Distributed rendering, OpenGL, Virtual Reality

1 INTRODUCTION

There are numerous situations where one needs to render the same 3D graphics divided to multiple displays in real time. Figure 1 shows a typical example of a virtual reality (VR) environment with multiple video walls.

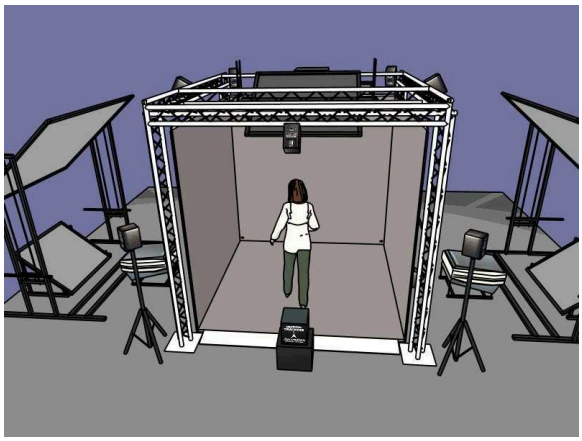

Figure 1. A VR setup with multiple walls

The Journal of WSCG, Vol.13, ISSN 1213–6964
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

Traditionally such situations have been handled by using a single high-performance computer with several graphics outputs. Recently a number of projects have utilized low-cost PC hardware for this purpose — using a cluster of commodity PCs to render all the walls. A similar change from an SGI Onyx2 server to a cluster of commodity PCs was the motivation behind the development of Broadcast GL as well.

2 BACKGROUND

Typically the most efficient way to accomplish high frame rates is to write applications that can be distributed and only send minimal amount of application data to the renderers. In these cases the application copies must produce identical behavior in all situations, which requires the programmer to write the application to support multiple hosts.

This is difficult if the application has a complex internal logic with plenty of user interaction. An alternative method of distributing the application is to spread the graphics API calls (OpenGL, DirectX) to multiple renderers. This is typically rather easy, since a normal 3D application already uses those calls to render its graphics. If these API calls can be distributed effectively to multiple rendering hosts, there is no need to rewrite the application. Since our software uses OpenGL, we are interested in distributing the OpenGL calls (`glVertex3f`, `glBegin`, `glEnd` etc.).

There are already several methods to spread OpenGL calls to multiple renderers. Staadt et al. have written an overview of different methods and analyzed their performance[Sta03a].

- GLX is the standard that is used in most UNIX-based operating systems that support the X windowing system [Wom98a]. GLX-based clustering integrates seamlessly to the windowing environment and it works without additional toolkits. For efficient multi-display rendering the renderer must be parallelized with one rendering thread per display pipe. There are toolkits that manage GLX contexts and set up projections matrices, for example VR Juggler [Jus98a].

- Chromium is a distributed 3D graphics system that uses the OpenGL-API to render graphics on multiple slaves [Hum02a]. Chromium optimizes the network usage by culling primitives before sending them over the network.

- Multi-display systems offered by Hewlett-Packard use a broadcasting method similar to ours. The method is briefly described in [Lef] but no benchmarks or in-depth details are provided. In addition to multi-display systems the architecture has been used in single-display environments to distribute the rendering load between multiple computers.

3 BROADCAST GL

Both GLX and Chromium transmit the rendering commands over a unicast TCP/IP connection. This approach is far from optimal if the same rendering commands need to be spread to multiple slaves. In this case both Chromium and GLX waste network resources by sending the information many times over. An example of such situation is a cluster of PCs rendering multiple walls of a VR installation: all the walls receive almost identical rendering commands, apart from the projection matrices.

Broadcast GL (BGL) solves this problem by using a broadcast technique to transmit the OpenGL API calls. As a result the BGL needs to send the graphics only once and each slave gets a copy of the rendering information.

Besides taking full advantage of the network resources this approach also simplifies the programming work, while the application can be completely single-threaded and still take full advantage of the multiple slaves. This is a relevant detail since most application programmers prefer writing non-threaded code. Potentially difficult problems such as thread synchronization and interlocking are avoided.

With the approach chosen in BGL we can implement only a subset of the OpenGL API. In practice the functions that return some data from the OpenGL system are currently only partially implemented. In theory all OpenGL functionality can be implemented, but the implementation of certain calls would be inefficient. The subset that is implemented works by caching a copy of the data in the application machine.

Due to its architecture BGL has strict requirements about the underlying network architecture. First of the network must support UDP multicast or broadcast. In practice this rules out wide-area networks. The network should also be fast and reliable. In practice these limitations imply the use of a cluster in local-area network with a number of computers connected via a switch.

4 IMPLEMENTATION

BGL uses a client-server architecture (following Staadt's taxonomy [Sta03a]). The application functions as a client that broadcasts BGL command byte stream (binary encoded OpenGL API calls) to the rendering servers over a UDP/IP socket. The slaves are independent rendering applications that receive the BGL byte stream and convert it back into OpenGL API calls. As a return channel each slave has a dedicated TCP/IP connection. BGL overview is shown in Figure 2.

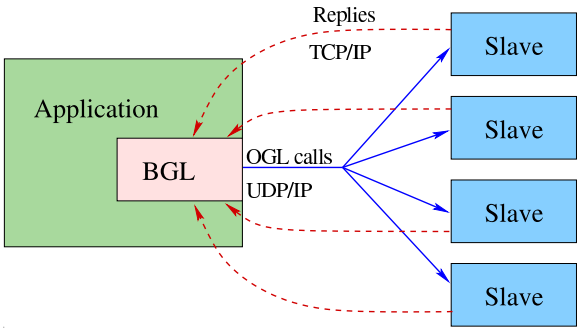

Figure 2. The networking architecture used in BGL. The rendering slaves can be either in the same machine or distributed across the network.

From the application perspective, BGL is little more than an OpenGL implementation, having the network transmission hidden behind the standard OpenGL API. Special BGL calls are used when OpenGL does not define calls that are necessary for applications. Examples of these needs are window handling, buffer swaps and selecting the rendering slave.

The BGL encoder library consists of functions that implement the OpenGL API (`glVertex3f`, `glNormal3f` etc.). The encoding functions store a number of bytes into a local data buffer. The data buffer can contain as much data as the system can fit into a single UDP packet. Once the packet is filled it is sent to the network.

Since OpenGL applications occasionally need to read back variables from the OpenGL implementation, BGL encoder keeps a local copy of some states. In practice this means that the current transformation matrices are kept in the encoder and they can be queried with normal `glGetFloatv`, `glGetDoublev` and `glGetIntegerv` functions.

BGL Specific Functions

The are also special BGL functions, such as OpenGL initialization and buffer swaps, that are needed to control behavior that is outside the basic OpenGL API, but needed by all applications. Below is a list of the BGL-specific functions that are visible to the application programmer.

- `bglInit(const char * address)` — This function initializes the BGL data transmission layer and connects to the slaves using the argument address.
- `bglQuit()` — This function shuts down the slaves and the data transmission layer.
- `bglSwapBuffers()` swaps the OpenGL buffers.
- `bglCreateWindow(int flags)` creates an OpenGL window .
- `bglResizeWindow(int w, int h)` resizes the OpenGL window.
- `bglMoveWindow(int w, int h)` moves the OpenGL window.
- `bglSelectRenderer(int id)` instructs the selected slave(s) to listen to the broadcast.
- `bglDeSelectRenderer(int id)` instructs the selected slave(s) to ignore the broadcast.

A typical way to use the “select” and “deselect” functions is in setting separate transformations for each renderer, for example:

```
// No one is listening now:
bglDeSelectRenderer(-1);
// Slaves with id 1 are listening:
bglSelectRenderer(1);
//Slaves with id 1 and 2 are listening:
bglSelectRenderer(2);
// Translate the geometry in slaves 1 and 2:
glTranslatef(0, 0, 1);
// All slaves are listening again
bglSelectRenderer(-1);
// Now we can render the scene
```

Send & Return Channels

When sending data over a socket we have to choose between UDP/IP and TCP/IP. UDP is a connectionless protocol that does not guarantee that all data that is transmitted gets to target, nor does it guarantee that the data arrives in the correct order. TCP/IP in turn provides a reliable connection, but with higher connection overhead.

In BGL the OpenGL data is sent over a UDP/IP socket since UDP offers lightweight broadcast and multicast features. TCP is used as the return channel protocol since return data rates are much lower, meaning that we can use a slower and more reliable connection.

Replies

If the application sends data at an excessive rate to the slaves it can overflow their UDP buffers, i.e. data arrives faster than it can be consumed. To avoid this the BGL requests replies from the slaves at fixed intervals (equal to ”buffer flush” in [Lef]). The slaves then answer that they have received the reply request and once BGL has received all the replies it can continue transmission. For example BGL might send a reply request after sending 16 packets. After transmitting the request, BGL will send a few more packets and then collect the replies from all the slaves. If the replies were collected immediately the renderers would have to empty their buffers before they could receive more data. This asynchronous approach helps us keep a buffer of rendering content in the slaves, resulting in higher performance.

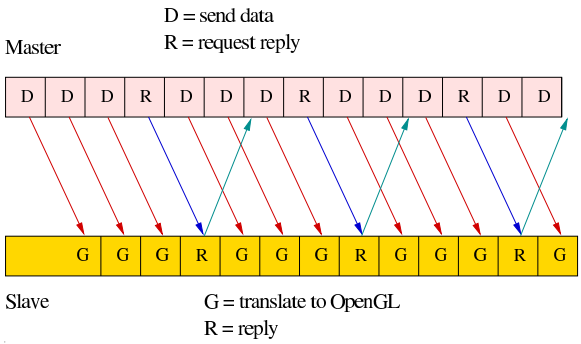

Figure 3. Asynchronous reply mechanism.

The reply system is also used when the application calls functions `glFlush`, `glFinish` or `bglSwapBuffers`. Each of these functions return only after all the slaves have replied. In the case of `bglSwapBuffers` the system first makes sure that all the slaves have done their rendering work and then issues a command to swap buffers.

Scalability

In BGL the data transfers are highly asymmetric. To render one frame the application may send out several megabytes of data, while the renderers' replies use only a fraction of that. The following calculation, which matches the benchmark setup below, gives a real-world example of the asymmetry.

When using UDP packets with 4096 bytes per packet and UPD buffers of 256 kilobytes, BGL application sends reply queries to the renderers at every 19 packets, resulting in 77824 bytes per reply request. Each reply packet uses 4 bytes, thus the downstream traffic takes roughly 19000 times more bandwidth. Since each renderer requires a separate reply connection this ratio is overly optimistic, but even with 1000 renderers the application sends out 19 times more data than it receives. The amount of data sent does not depend on the number of slaves, unless the slaves are controlled individually, as was done in the transformation example above. As long as the used network is reliable, new renderers can be added with minimal performance loss.

Recovering from Transmission Errors

UDP connections are inherently unreliable. The packets can be lost or they may arrive in the wrong order to the recipient. Altough we are using a very reliable network both error cases do occur. Since OpenGL does not tolerate missing commands these errors must be corrected in the transport layer. Both TCP and UDP guarantee the correctness of the transmitted packets, so there's no need to build an additional bit-level error correction mechanism.

BGL uses the TCP return channel to report missing packets. When a renderer receives a packet with unexpected counter value it puts the packet to a store the notifies the master that a packet was missing. The master in turn keeps the latest UDP packets in a ring-buffer and retransmits the missing packet. This error correction is not enough in the cases where a renderer loses multiple packets (including packets with reply commands). To handle these situations the master retransmits packets automatically if the renderers do no reply within a given time interval.

Together these strategies guarantee that the transmission errors are corrected as long as at least some amount of packets reach the renderers. We have tested the system by intentionally losing packets. The error recovery works correctly even when 80 % of all transmitted packets are lost.

Internal Structure

BGL is composed of two parts. The application library (libBGL) implements the OpenGL API and the BGL-specific extra functions. This library contains OpenGL encoding functions and data transport layer. The renderer is a stand-alone application that also includes the transport layer and OpenGL decoding functions.

The OpenGL API has been originally designed to be easily streamable. This makes encoding and decoding the API calls fairly easy. In BGL most OpenGL functions are defined with one-line macros. Writing the encoding and decoding layers took only two days.

The data transport layer is more demanding for the programmer. Finding the most effective way to use network resources took more time than implementation of decoding library. This part is also more easily broken by networking anomalies that may not have been present when the system was first tested.

5 BENCHMARKS

BGL was benchmarked against Chromium and a GLX-based graphics distribution mechanism. The OpenGL distribution platforms are detailed below:

1. GLX-based threaded renderer: This system uses a separate rendering thread for each X11 display, thus rendering two windows per thread. This system is similar to the VR Juggler OpenGL application framework [Jus98a]. Based on informal tests, our GLX-distribution system has performance characteristics similar to the VR Juggler implementation.
2. Chromium: We used Chromium version 1.7. Chromium's tilesort SPU was used for the graphics distribution and the render SPU for viewing the graphics. The tilesort SPU culls polygon faces before sending rendering commands to the network, thus decreasing the network load.
3. BGL: The application was linked with the BGL encoder library and a small projection management library. We used a normal broadcast address `10.0.0.255:10001` to deliver the broadcast from the application to the renderers.

All the described methods were tested in the following three test cases:

1. Display of a real-world architectural model, rendered with display lists. This benchmark represents a typical static model, for example a background scene in computer games. A part of the scene is shown in Figure 4.

2. Display of a real-world architectural model, rendered without display lists, i.e. in immediate mode. This benchmark represents volatile data sets — for example objects under deformation cannot be compiled into display lists.

3. Texture streaming. This benchmark represents a case where texture animation is made by streaming a new (sub)texture into the hardware at each frame. Such approach is commonly used when a video stream is embedded into OpenGL graphics. In our test the size of the RGB texture was 320 by 240 pixels (225 kB). A screenshot of this test is in Figure 5.

Figure 4. A screenshot of the architectural scene used in tests 1 and 2. The scene has 96733 triangles. All lighting is done with texture maps.

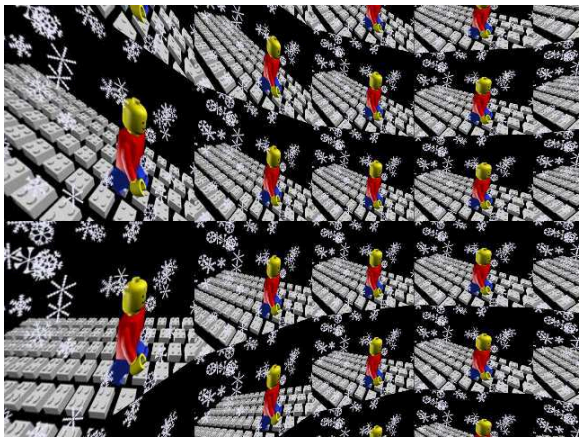

Figure 5. A screenshot of the video player test software.

Tests 2 and 3 are bandwidth-intensive, while test 1 stresses the graphics pipeline. During the tests we measured the following metrics:

1. Frame rate (frames per second, fps)
2. Network traffic in the application computer (megabytes per second)
3. Application computer load (percentage of CPU resources used)

In the test the scene was rendered on four rendering computers. Each computer displayed two separate OpenGL windows, representing the left and right eye views. The window size was 1024 x 1024 pixels. Each window had a different projection matrix, matching a typical four-wall Cave setup similar to Figure 1. Additional tests were run on an SGI Onyx2 system and a single desktop PC. The SGI rendered the graphics into four stereo windows resulting in a render load equal to the PC cluster tests. These tests were ran to compare the performance of the PC cluster to the retiring system. The stand-alone PC in turn rendered the graphics into two windows, providing an estimate of the highest achievable frame rate.

The test setup was composed of five Linux-based computers — an application PC and four rendering machines. Each computer had a 2.8 GHz Intel P4 CPU, an integrated gigabit Ethernet controller and an NVidia FX5900 graphics card. The PCs were running Linux kernels from the series 2.4 and 2.6. The SGI-based system was an Onyx2 with two IR2 pipelines and eight 200 MHz R10000 CPUs. The test results have been collected to Tables 1–3.

Test	Architecture	GLX	Chromium	BGL
1	1 GB	2.8	37	19
	100 MB	1.62	46	15
	PC/Local	16	-	-
	SGI/Local	1.3	-	-
2	1 GB	0.87	2.4	4.2
	100 MB	0.10	0.32	0.82
	PC/Local	24	-	-
	SGI/Local	1.8	-	-
3	1 GB	7.5	10	105
	100 MB	3.1	1.8	24
	PC/Local	150	-	-
	SGI/Local	20	-	-

Table 1. Frame rates for three tests in gigabit and 100 Megabit networks (frames per second).

Test	Network	GLX	Chromium	BGL
1	1 GB	25	4	0.48
	100 MB	0.38	5.3	0.47
2	1 GB	95	87	55
	100 MB	12	12	12
3	1 GB	7.5	46	29
	100 MB	6.4	11	9

Table 2. Network traffic (Megabytes transmitted per second).

Test	Network	GLX	Chromium	BGL
1	1 GB	44	20	2
	100 MB	6	20	2
2	1 GB	70	45	35
	100 MB	10	10	1.5
3	1 GB	6	24	18
	100 MB	6	7.5	5

Table 3. Application computer CPU load.

The CPU load of the application is split into user-space load and kernel-space load. The CPU loads were measured with "top" -program that is part of standard Unix command set. This measurement is complicated by the fact that the definition of CPU load is not an obvious measure on modern hyper-threading CPU's. In this case we took the CPU idle time from "top" and calculated the application load from it. The idle time represents how much time the CPU has left to run other applications. These load values are shown in table 3.

While the above benchmarks measure run-time performance there are other aspects that are important for the application programmer as well. A summary of these aspects has been collected to Table 4.

System	GLX	Chromium	BGL
Network scalability	Poor	Moderate	High
OpenGL compliance	Good	Moderate	Moderate
Ease of programming	Poor*	Good	Good

Table 4. Qualitative differences between different approaches
* Requires threaded rendering into multiple GLX contexts.

It is worth noting that the test setup differs from Staadt's. We are running a single centralized application with distributed graphics, while Staadt's tests also included distributed applications [Sta03a].

In addition to the system benchmarks we ran a small-scale scalability test. Test 2 (immediate mode rendering of the architectural model) was run on one to four rendering computers. Tests 1 and 3 were discarded because they were too dependent on pure rendering or network speed and would not have given meaningful results about scalability. The graph shown in Figure 6 displays the frame rates obtained in this test.

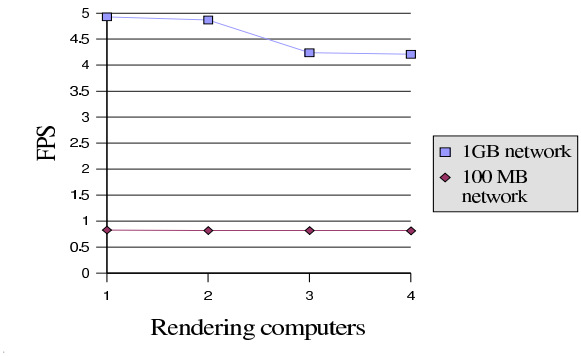

Figure 6. The effect of added rendering computers on the frame rate.

Analysis of the Benchmarks

The benchmarks above show that BGL, in many test cases, outperforms both the standard GLX-based graphics distribution and Chromium. In these tests Chromium could often use its culling algorithms to lower the network traffic. If one thinks about the usage in a fully immersive six-wall Cave, this culling cannot eventually do more than ensure that the same data is not sent from the application to the renderers more than once. Since there are two windows with nearly identical views for each wall, the vertex data will be sent twice unless the software can recognize the overlap.

In test 1 Chromium did extremely well and surpassed even the local GLX rendering. This is apparently due to its heavy culling methods that could discard even complete display lists. BGL proved its scalability by providing approximately the same frame rate as the single PC.

BGL was clearly the fastest system in tests 2 and 3, delivering higher frame rates with lower CPU load and lower network stress. In test 3 both Chromium and GLX were forced to send the texture eight times to the renderers, resulting in roughly eight times more data traffic per frame.

The 100 Megabit Ethernet was easily saturated by all systems. Surprisingly, none of the systems could saturate the gigabit Ethernet in any of the test cases. It seems that the computers have trouble moving data over the network at such high rates. Also it seems that in the renderer computers the OpenGL usage has negative effect on the networking performance, probably because both require bus resources that are mutually exclusive.

The performance of the GLX-based distribution was in most cases disappointing. Especially, one would expect that GLX-distribution would run well with display lists, but this was not the case. This problem

might be caused by networking issues or problems within NVidia's GLX implementation. We have experienced similar performace problems when using VR Juggler in our test configuration. When run locally the GLX code worked fine, both in the SGI tests and in the stand-alone PC test.

When we compare the performance of the network rendering against rendering the same graphics locally we can see that with display lists (test 1) the local rendering is in fact slower. In immediate mode (test 2) the local rendering is significantly faster while the video streaming (test 3) application is 50 % faster when ran locally.

The BGL-based PC-cluster outperforms our old SGI-system in all the tests. While this information is not particularly surprising, it created significant confidence to the new platform. The scalability of the system is good (Figure 6). In gigabit network the frame rate dropped only 15 % when the number of renderers was changed from 1 to 4. In the fully saturated 100 MB network the number of renderers made no difference.

6 DISCUSSION

As the BGL implementation matures, it allows for several interesting applications. Because of the scalability of the approach, large rendering clusters can be built without significantly increasing the load of the the application computer. The broadcast graphics can be viewed across the network in different visualization devices, such as an ordinary monitor, head-mounted display or a Cave, whereas for the application code the final output device bears very little importance. The method somewhat resembles the traditional radio and TV broadcasting and could be even used for similar purposes in the form of a "3D television". Large-scale broadcasting for various bandwidths cannot be handled by a single computer, thus creating a need for a proxy or other middleware solution.

The current BGL implementation can store the OpenGL command stream to a file. This feature was created mostly as a debugging aid, but it could also be used as a 3D video format. The resulting files can readily be compressed with ordinary tools such as gzip and even further with more advanced techniques such as texture compression.

In its current state, BGL features only a bare-bone OpenGL implementation. Full OpenGL compliance is in practice difficult to achieve, mainly because the OpenGL state is distributed over a cluster of nodes. Frequent state queries from the nodes is also likely to cause performance loss due to the stalling of the rendering stream.

At the moment one badly behaving renderer can stall the whole cluster. This clearly means that the synchronization should be studied further. We suspect that once the network latency and synchronization are handled better, the overall throughput of the system will increase considerably. The symmetry of the rendering computers is vital to good performance since the slowest node effectively dictates the overall frame rate.

The tests that were conducted did not incorporate genlocking or any synchronization to display updates. This choice was intentional because we wanted to measure the maximum throughput possible with each of the systems. In practise such constraints are often present and slow down the frame rate. For example a double-buffered 100 Hz sychronized display typically limits the steady frame rates to 100, 50, 25 FPS and so on. Hardware genlocking should not affect the frame rate but a software-based approach such as SoftGenLock [All03a] does because it introduces additional system load.

7 CONCLUSIONS

We have presented and evaluated an alternative method to distribute graphics API calls to multiple rendering computers. By using the broadcast/multicast networking we have managed to ensure the same graphics data is not sent more than once across the network, regardless of the number of renderers. The current BGL implementation is far from perfect and we will continue to improve it.

In the light of the benchmark results it seems obvious that none of the OpenGL distribution systems is in all cases better than the others. Rather, the best choice depends on the application, computers used and the network characteristics. Obviously the best use cases for BGL are data-intensive applications that require good scalability to multiple displays. Furthermore, the simple single-thread application logic allows for easy adaptation of existing desktop OpenGL software.

References

[All03a] Allard, J., Gouranton, V., Lamarque, G., Melin, E., Raffin, B. Softgenlock: Active Stereo and Genlock for PC Cluster. in Proceedings of the Joint IPT/EGVE'03 Workshop, Zurich, Switzerland, May 2003.

[Hum02a] 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 ACM

Transactions on Graphics (TOG) , Proceedings of the 29th annual conference on Computer graphics and interactive techniques, Volume 21, Issue 3, 2002.

[Jus98a] Just, C., Bierbaum, A., Baker, A., and Cruz-Neira, C. VR Juggler: A Framework for Virtual Reality Development. 2nd Immersive Projection Technology Workshop (IPT98), Ames, Iowa, May 1998.

[Lef] Lefebre, K. An Exploration of the Architecture Behind HP's New Immersive Visualization Solutions. Hewlett-Packard Company.

[Sta03a] Staadt, O.G., Walker, J., Nuber, C., Hamann, B. A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. in Proceedings of the workshop on Virtual environments 2003.

[Wom98a] Womack, P., Leech, J. (eds.). OpenGL Graphics with the X Window System. Version 1.3, October 19, 1998.