Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

# Distributed OpenGL Rendering in Network Bandwidth Constrained Environments

B. Neal[1], P. Hunkin[1] and A. McGregor[1]

[1]The University of Waikato, Hamilton, New Zealand

**Abstract**

*Display walls made from multiple monitors are often used when very high resolution images are required. To utilise a display wall, rendering information must be sent to each computer that the monitors are connect to. The network is often the performance bottleneck for demanding applications, like high performance 3D animations. This paper introduces ClusterGL; a distribution library for OpenGL applications. ClusterGL reduces network traffic by using compression, frame differencing and multi-cast. Existing applications can use ClusterGL without recompilation. Benchmarks show that, for most applications, ClusterGL outperforms other systems that support unmodified OpenGL applications including Chromium and BroadcastGL. The difference is larger for more complex scene geometries and when there are more display machines. For example, when rendering OpenArena, ClusterGL outperforms Chromium by over 300% on the Symphony display wall at The University of Waikato, New Zealand. This display has 20 monitors supported by five computers connected by gigabit Ethernet, with a full resolution of over 35 megapixels. ClusterGL is freely available via Google Code.*

## 1. Introduction

In recent years, clusters of interconnected workstations have become a common solution for powering large composite displays, or "display walls". These displays are typically both physically large (e.g. 10m x 3m) and high resolution (e.g. 30 megapixels). Applications that need this type of resource include scientific data set visualisation, advanced human-computer interaction experiments, and many others.

This type of display is often used with high resolution 3D interactive applications that use accelerated graphics via OpenGL. Ideally the applications would run at a minimum of 60 frames per second to ensure smooth interaction. In this scenario, the typical bottleneck is the speed at which OpenGL commands can be transferred from the computer hosting the application to the display computers. The network is significantly slower than the PCIe graphics bus connection that carries the commands when a single machine is used.

Several systems have been developed to make cluster rendering possible. Some require applications to be modified or recompiled. While this approach has the most scope for performance optimisation, it requires that source code and technical skills are available before a new application can be run on a display wall. On the other hand, the range or applications that can be supported by approaches that do not require the application to be modified are limited by network performance.

This paper describes ClusterGL, a system that transparently distributes rendering over multiple rendering nodes without modification to the application. ClusterGL uses three optimisations to reduce network traffic and improve performance for demanding applications. The system was initially developed to allow a visualisation of network traffic created by the BSOD network monitoring system [Hun09], to be displayed at high resolution on a display wall (see figure 1). Existing approaches did not perform well enough to support this application which creates a large amount of dynamically changing vertex data.

ClusterGL is currently in operation on the Symphony Cluster display wall [Wai11], at The University of Waikato, New Zealand. The wall has 20 22" monitors in a 5x4 arrangement. It has a total resolution of 8400x4200 (approxi-



DOI: 10.2312/EGPGV/EGPGV11/021-029

**Figure 1:** *Network traffic being visualised on the Symphony wall*

mately 35Megapixels). The monitors are driven by five display nodes that are connected to one another via a single gigabit Ethernet switch. The goal for the project was to be able to support BSOD and other applications at the full resolution of the display wall with a frame rate of at least 60 Frames Per Second (FPS), the refresh rate of a typical LCD monitor. Additionally, existing applications should be able to use the wall without modification or recompilation. ClusterGL successfully meets these goals.

## 2. Background and related work

There are other approaches to implementing distributed rendering for display walls. The main approaches are described in the following sections.

### 2.1. Single machine systems

Recent advances in GPU technology such as AMD Eyefinity [AMD09] allow up to six monitors to be connected to a single PCIe graphics card. Up to four of these cards can be used in a single display machine, allowing up to 24 monitors to be connected as one computers display surface. This provides high levels of performance, as there is no network traffic involved.

However (at the time of writing), this approach requires specialised hardware, and will only scale to 24 total monitors. The techniques we describe here, allow scaling beyond the capabilities of a single machine.

### 2.2. 2D only

Some systems are intended primarily for 2D rendering. These include Distributed Multi-head X (XDMX) [MKDF04] and the Scalable Adaptive Graphics Environment (SAGE) [JRJ*06, LBH07]. While there may be some support for 3D acceleration, these systems do not meet the requirements of many 3D applications. Because high performance 3D applications are our area of interest, these approaches are not considered further here.

### 2.3. Indirect rendering

Xorg provides indirect network rendering capabilities for OpenGL applications and can be used for a display wall when using Distributed Multi-head X (XDMX) [MKDF04]. This involves an additional layer of indirection between the application and the graphics hardware. This allows commands to be tunnelled over a network but means the application will suffer a speed penalty. Some of the more complex graphics features provided by OpenGL extensions (like most advanced shaders operations) do not operate in an indirect rendered environment so this approach also does not meet our needs.

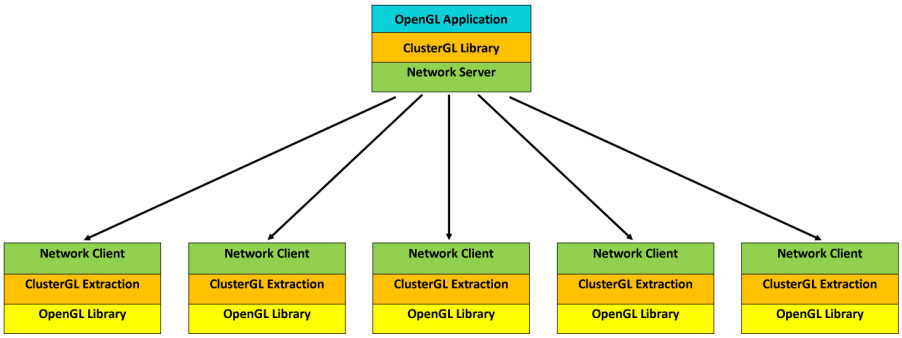


**Figure 2:** *Overview of ClusterGL running on five display nodes*

### 2.4. Frame buffer copying

Sage [JRJ*06, LBH07] supports OpenGL applications in a network environment by copying the rendered OpenGL frame buffer contents over the network. Unlike many of the approaches, its performance does not degrade with high-complexity geometries because geometry information is not passed over the network. However, while this approach is simple and easy to implement, it creates a large quantity of network traffic, particularly when high resolutions are required. On the Symphony display wall this would achieve frame rates of less than 10 FPS depending on the real-time compressibility of the image.

### 2.5. Parallel OpenGL rendering

Equalizer [EMP09] is a widely-used parallel rendering framework for OpenGL applications. It greatly increases rendering efficiency by parallelising rendering code, but requires modification to the original application and does not meet our goals.

### 2.6. OpenGL command stream

WireGL [HEB*01] and its successor Chromium [HHN*02] operate by capturing the OpenGL command stream and passing the commands and their arguments over the network to remote OpenGL renderers. This technique normally requires much less network traffic that frame buffer copying and scales better to high resolutions. For many applications, performance is still limited by network performance, particularly if there are complex geometries and/or many display nodes. This method has become the most widely used solution for distributed OpenGL rendering for unmodified applications.

BroadcastGL [IRK05] also uses this approach but reduces network load by using a reliable broadcast protocol. Instead of using TCP [Pos81] which provides a reliable point-to-point transport mechanism, BroadcastGL uses UDP [Pos80] which can multi-cast a single packet to multiple destinations but does not provide reliable transmission. Using multi-cast reduces network traffic because the data is sent once for all display nodes, rather than once for each display node. ClusterGL also uses multi-cast; more details are given in section 3.3.

The current implementation of BroadcastGL only implements a small subset of the OpenGL API up to v2.1 (390 of the 1228 commands), which limits the applications that can use it. The project is not under active development and the software is not publicly available, however a copy was provided for benchmarking purposes. See section 4.

## 3. ClusterGL

ClusterGL allows Unix applications that use OpenGL to be transparently rendered on a display wall. It was developed and tested on the Debian distribution of GNU/Linux but should work well on most Unix systems. ClusterGL captures the OpenGL command stream in a similar manner to Chromium and BroadcastGL. However, it uses three optimisations to improve performance. These are: frame differencing, described in section 3.1; compression, described in section 3.2; and multi-cast, described in section 3.3.

ClusterGL has two main components: a client library on the machine that is running the application and a renderer which runs on the machines that are connected to the displays. Figure 2 shows the architecture for five display nodes.

The client library is implemented as a shared library and is responsible for capturing the OpenGL commands from an OpenGL application. A user wishing to use ClusterGL sets their `LD_PRELOAD` environment variable so that ClusterGL's implementation of the OpenGL calls are used in preference to the standard system OpenGL library. The ClusterGL implementations of the OpenGL calls serialise their parameters, perform the optimisations noted above and send the call and parameters to the display nodes. Individual op-

timisations can be disabled to match the needs of the application.

Many OpenGL calls include a pointer to an array of data currently stored in RAM. The size of this array must be known to be able to serialise the data for transmission across the network. Some calls provide this value as an additional argument, while others can easily be calculated with pre-existing knowledge of the data layout. However, some calls do not provide enough information to be able to calculate the array size. In these cases, the size of the array is provided in a later call when rendering the geometry using that pointer. ClusterGL caches these calls until the size of the array is known. This allows correct serialisation of the array so it can be passed to the rendering nodes.

The renderers receive the optimised commands from the client library and expand them by reversing the optimisations. They then call the native OpenGL library to execute the instructions and render the image via the local graphics card.

Each renderer is configured to display the part of the virtual display that its monitors show. The Symphony wall is bigger than the maximum size of an OpenGL context (which on the hardware we use is 8192x8192) so simply manipulating the OpenGL viewport will not fully solve the problem– if it is used, a part of the display will be blank because it is outside the OpenGL viewing context. Instead, ClusterGL manipulates the frustum so that only the part of the world that is displayed by that node is in the rendering space. The normal use of the frustum has the centre of the frustum at the centre of the world with equal lengths for the left and right boundaries. This is not the case when the frustum is used to select just a part of the world that may, for example, be entirely to the left of the centre line. This requires manual calculation of the frustum parameters.

Monitor bezel compensation is performed by increasing the size of the virtual world to match the resolution that would be formed if the bezels were replaced with display surface at the same number of dots-per-inch as the monitors. The pixels that are "under" the bezels in this extended image are not rendered. This means that diagonal lines in the geometry appear straight on the display. Some 2D applications (e.g. still images) are easier to view without bezel compensation. ClusterGL allow bezel compensation to be disabled if appropriate.

The time that renderers take to render their part of the display may vary because they have different objects to render. To maintain accurate synchronisation of the images displayed by the each display node, the client library periodically sends synchronisation packets to the renderers (the default is every 20 frames). The client will wait until replies from all synchronisation packets are received before sending new frame data. Our experience over a range of applications, including scientific animations and interactive 3D games, is that no noticeable cases of lost synchronisation occur.

### 3.1. Frame Differentials

Many OpenGL applications produce sequences of frames where the OpenGL commands for each frame are very similar to previous frames but with small differences. For example, successive frames might draw the same object but with small differences in rotation.

ClusterGL optimises repeated sequences by only transmitting the differences between consecutive frames. The frame differential algorithm is shown in figure 3.

```
sameCount = 0

for( i = 0; i < thisFrame.size(); i++ )
    Instruction inst1 = prevFrame[i]
    Instruction inst2 = thisFrame[i]

    if( inst1 != inst2 )
        if( sameCount > 0 ) {
            sendIdentical( sameCount )
            sameCount = 0
        }
        sendCommand( inst1 )
    } else {
        sameCount++
    }

}

if( sameCount > 0 ) {
    sendIdentical( sameCount )
}
```

**Figure 3:** *Pseudo-code outline of the operation of the CLUSTERGL_REPEAT ClusterGL command*

This approach requires that the command stream be buffered on both the client and renderer side. When renderers receive the commands sent by `CLUSTERGL_REPEAT()`, they copy the required number of OpenGL instructions from the previous frame command. The client library ensures that a `CLUSTERGL_REPEAT()` command cannot be invoked before a full set of frame commands has been transmitted.

In the best case (where a scene is static except for an initial transform), an entire frame can be sent with just three commands (see the example in figure 4). In the worst case, no frame differencing occurs and performance is the same as it would be with this optimisation turned off. This is because the CPU and memory cost of maintaining the frame comparison buffers is negligible when compared with the cost of network transmission.

This technique incurs an overhead of increased CPU usage. However, most application are limited by network bandwidth (which is the primary motivation for this work). In this case, the CPU is not fully utilised and the additional CPU overhead does not significantly change the performance of the application as a whole.

```
sendIdentical( sameCount )
rotate( rot, x, y, z )
sendIdentical( sameCount )
```

**Figure 4:** *Pseudo-code outline of the operation of the ClusterGL REPEAT_ALL command*

### 3.2. Stream Compression

Even after frame differencing had reduced the redundancy in an OpenGL command sequence, it is possible to compress it further using general purpose compression algorithms. There may be common sequences of instructions that are less regular, or at a finer scale, than that required for frame differencing to be effective. Further, native OpenGL applications write textures to the graphics card in an uncompressed format. While the performance cost of compression and decompression is not warranted on a single system, it can be useful when OpenGL commands are sent over a network.

To be useful, the compression algorithm must be able to compress the OpenGL instruction stream and thereby decrease the bandwidth needed to send it to the renderers but not require so much CPU time that the CPU becomes a new bottleneck. ClusterGL currently supports the `libZ` [lGA10] and `LZO` [Obe10] compression algorithms. On our hardware, the LZO algorithm performs very well on most OpenGL command streams, typically achieving $> 70\%$ compression without saturating the CPUs on either the compressing or decompressing nodes.

### 3.3. Multi-cast

Every instruction in an OpenGL command stream is required, and must be reliably sent to each rendering client. TCP [Pos81] can be used to provide a reliable, in order stream of data between two network hosts. However, TCP is a point-to-point protocol, meaning the data can only be transmitted to one rendering client at a time. To send the stream of OpenGL instructions to all rendering clients, multiple TCP connections are needed. This means that the same data is set over the network multiple times and the network capacity is effectively divided by the number of rendering nodes. This effect reduces the performance and limits the scalability of display walls.

ClusterGL uses UDP [Pos80]; a connectionless protocol that does not guarantee the data will arrive at the destination, nor the order in which the packets will arrive. UDP is lightweight and offers broadcast and multi-cast ability. This allows a stream of data to be sent to multiple destinations using a single send command. The data will only be transmitted once over the network. Switches along the path will forward each packet to all destinations.

While UDP permits multi-cast, graphical rendering requires reliable, in sequence delivery of data. Existing reliable multi-cast protocols contain complex processes to add reliability [CGF*01, ARA09, SL96, WVK*01]. These protocols are designed to operate over complex networks with multiple hops. As most display walls have a dedicated, high speed network, the current reliable multi-cast protocols are not optimised for this type of network.

To add reliability to UDP multi-cast packets, ClusterGL adds a 12 byte header onto each packet. The header provides message sequencing and loss detection. While the high-volume OpenGL command stream is sent over UDP, a TCP connection is also used between the client and each renderer. This allows communication in the reverse direction for OpenGL commands that return a result. It also provides a reliable channel for positive or negative acknowledgements of the data that has arrived via UDP.

As the network is dedicated for display wall traffic and only contains a single switch, it is very uncommon for data not to arrive reliably and in order. At the end of each OpenGL frame (which may have been sent via multiple UDP messages) each renderer sends an acknowledgement message (ACK) to notify the ClusterGL client library that all data has arrived successfully. If a renderer receives a packet that is not the next in the sequence, a negative acknowledgement (NACK) is sent to the library. When the library receives a NACK, it retransmits all packets starting from the offset the renderer that transmitted the NACK packet is expecting. Renderers that did not experience loss simply drop the retransmitted packets, while the renderer(s) that were missing the packet(s) are able to catch up. Note that there is no extra network overhead associated with broadcasting to all renderers. Also, the little CPU time used by renderers to receive and skip the unneeded frames is not significant because those renderers must stay in sync with those that did lose the frame; they will have CPU to spare.

A double buffering scheme is used to allow overlap between the network transmission and the application generating the next frame. That is, while one frame is begin transmitted on the network, the client library returns to the application allowing it to generate the next frame.

## 4. Benchmarks

ClusterGL was benchmarked against Chromium and BroadcastGL on the Symphony display wall described in the introduction. The OpenGL distribution platforms are:

1. BroadcastGL: The most recent version of BroadcastGL was obtained directly from the authors.
2. Chromium: Version 1.7 with the default settings.
3. ClusterGL: Version 0.9 with the default settings (all optimisation features and bezel correction enabled).

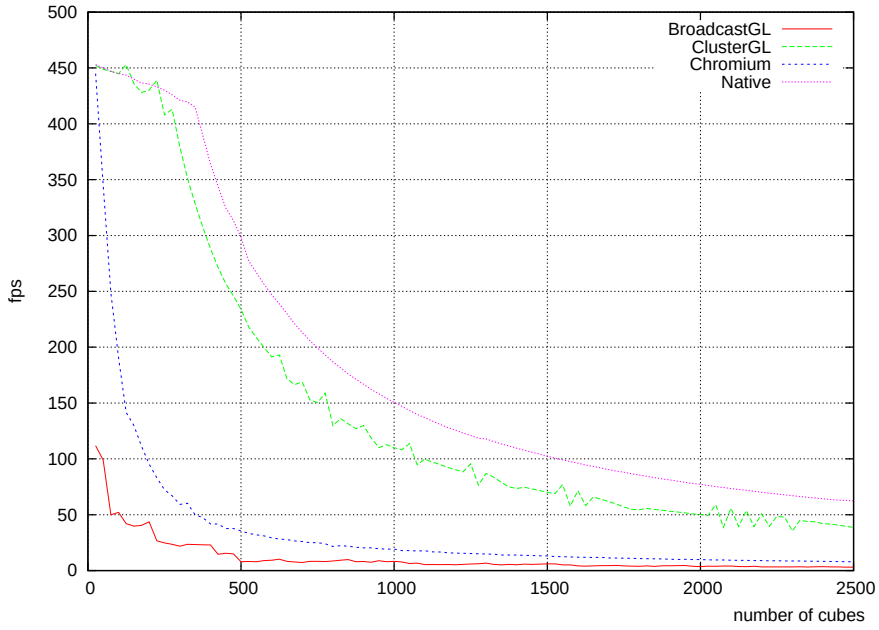


**Figure 5:** *Display list stress test results*

### 4.1. Cube Test

The first test is a display list stress test. It involves rendering a number of static cubes, each with different colours. This test is based on the NeHe Tutorial 12 [Mol03]). The test generates a large number of OpenGL instructions and consequently a high network load. The test approximates a static model, for example a background scene in computer applications or a large, complex scientific model.

Figure 5 shows the results for rendering between 25 and 2,500 cubes. In addition to the frame rate achieved by the three systems, the performance of OpenGL on a single system is shown, rendering at 1680x4440 to a single vertical slice of our wall. This gives a best case reference.

With this application, ClusterGL benefits from frame differencing and compression. Its performance follows the same trend as the best case (running the application locally) achieving in the range of close to 100% (for small numbers of cubes) to about 62% for 2500 cubes.

Chromium also performs well with small numbers of cubes but it's performance drops more quickly than ClusterGL. The biggest discrepancy is at 225 cubes (ClusterGL is approximately 100% of the best case while Chromium is approximately 20%).

While BroadcastGL produces less network traffic than Chromium, it does not perform as well as Chromium (it achieves between 5% and 25% of the best case). We believe this is probably because of inefficiencies in the design of the BroadcastGL reliable multi-cast protocol. With a larger number of rendering nodes (as might be the case with a very large display wall) BroadcastGL might outperform Chromium as its savings in network traffic become more dominant.

### 4.2. OpenArena

The cube test is a simple stress test. While it is easy to analyse it does not necessarily represent real applications. As a cross check, we compared the performance of ClusterGL and Chromium when running OpenArena [Ope05] on the Symphony display wall. OpenArena is a free and open source first-person shooter game. It is heavily based on the open source release of Quake III Arena. It uses a wide range of OpenGL commands (testing OpenGL coverage) as well as producing a lot of data in most frames. We only tested ClusterGL and Chromium in this case because BroadcastGL does not implement enough of the OpenGL API to run OpenArena successfully.

To test each system, the "anholt" time demo was run using each system. This demo is used as a standard benchmark by the Xorg development team (and many others) to test the speed of OpenGL implementations. [anh05, FE09, SJNC09, JBG*10] The demo generates a sequence of 840 frames as quickly as the hardware will allow.

In the test, ClusterGL averaged 102.5 FPS compared with Chromium's average of 32.6 FPS. Chromium has less variation in the frame rate. We believe this is because the amount of data it needs to send varies less. ClusterGL's optimisations


**Figure 6:** *OpenArena running under ClusterGL*

vary in their effectiveness but always allow it to outperform Chromium. For more than 80% of the test the difference in frame rates is greater than three times. For a game, this is a significant difference. 98% of the time, ClusterGL exceeds 60 FPS where as Chromium never achieves this rate.

ClusterGL produces a maximum frame rate of 500 frames per second when the player is spawning. At this time, the game is displaying a static menu with less variation in the background.

The anholt test described above uses an "indoor" map. OpenArena is particularly stressful when rendering outdoor maps. This is because the majority of geometry data needs to be transmitted every frame as the player is often able to see from one end of the map to the other. We also tested the two systems with the "osago2" outdoor map. Chromium averaged 8 FPS compared with 25 FPS for ClusterGL. While ClusterGL is closer to the target frame rate of 60 FPS neither achieve it in this case.

### 4.3. Impact of individual optimisations

The effectiveness of various combinations of optimisation techniques were benchmarked on the OpenArena timedemo and compared to Chromium. As a reference, they were also compared to a single machine running in full screen mode at 1680x4440 and no remote renderers (i.e. not using ClusterGL or Chromium). The results are shown in table 1.

These results are indicative only. The relative improvement of each technique is dependent on the application; some applications may benefit more than others for a given technique.

## 5. Discussion

In our testing, ClusterGL's optimisations reduced network traffic and significantly increase frame rates for all applications tested. Performance declined as the geometry became more complex but following the same trend as OpenGL on a single machine. The primary source of this decrease is the CPU and GPU load required to generate and render the more complex scenes.

We observed that ClusterGL traffic tends to be more "bursty" than Chromium traffic. In the worst case, ClusterGL sends each OpenGL instruction individually. This always happen on the first frame in a repeated sequence. In the example of the Display List application, ClusterGL transmitted the initial burst of commands in the first frame, causing a bandwidth spike. After that, throughput fell as `CLUSTERGL_REPEAT` instructions take effect.

As Chromium has not had any significant updates since

| *Technique* | *Average FPS* |
|---|---|
| None | 22.5 |
| Compression | 59.9 |
| Deltas | 27.6 |
| Compression + Deltas | 66.7 |
| Multicast + Deltas | 68.6 |
| Compression + Multicast | 83.6 |
| All techniques | 102.5 |
| Chromium | 32.6 |
| Single Machine | 155.7 |

**Table 1:** *The effect of different ClusterGL optimisations on the OpenArena timedemo benchmark*

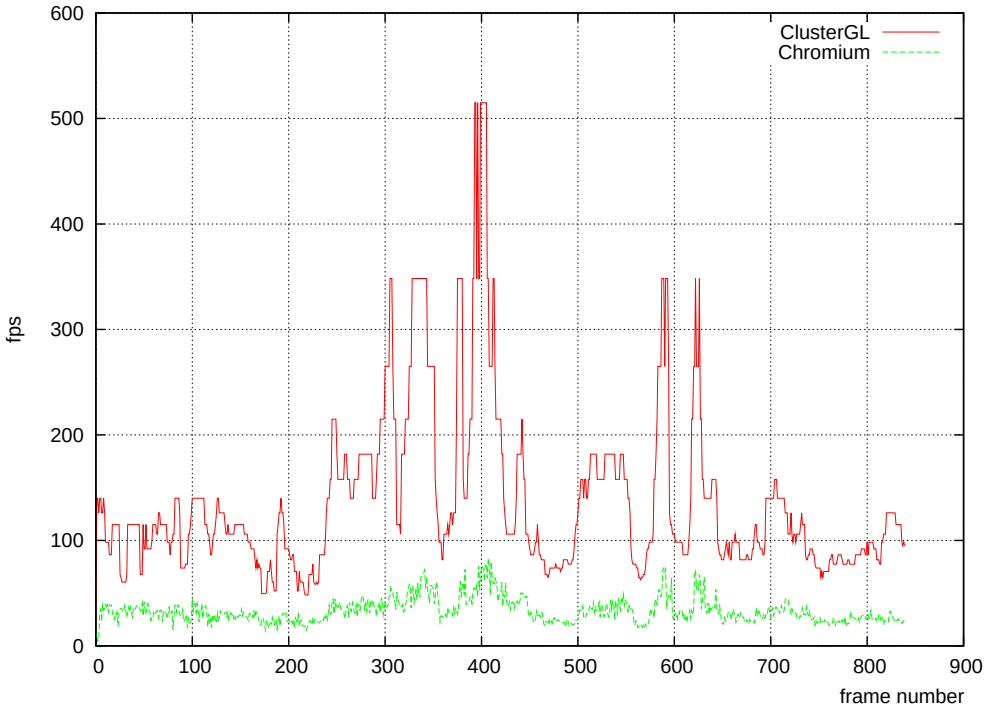


**Figure 7:** *OpenArena results*

2005, it only contains support for OpenGL version 2.0 or earlier. Applications that use newer features are not able to be run or do not render correctly. OpenGL 2.1 contains new methods–for example newer versions of pixel and vertex shaders. These features are widely used in modern applications. ClusterGL supports most OpenGL 2.1 calls.

## 6. Future Work

Currently, ClusterGL supports 92% of the OpenGL API. This subset supports all the applications that we have run to date. While we would like to achieve 100% coverage, this is difficult to achieve because not all calls are well documented. Most of the remaining calls are vendor extensions to OpenGL or rarely used legacy support and software will normally adapt to run without them.

It is possible to improve the frame differencing algorithm that ClusterGL uses. In particular, streams of instructions that contain instruction re-ordering will currently defeat the optimiser. Techniques to identify more cases where differencing can be used should further improve ClusterGL's performance.

Another bottleneck in the performance of ClusterGL is the time spent blocking, waiting for return data from each renderer. For example, a poorly written OpenGL application may request the maximum view port dimensions every frame. This results in a network round trip, while ClusterGL waits for the returned value, before carrying on running the application. This could be improved by using a cache of recently used objects on the client and returning the information from the cache. Static values, such as the maximum view port dimensions, could be cached during initialisation, while dynamic values, such as lighting parameters, could be cached as they are sent in calls. Our initial experimentation indicates that this can result in significant gains; up to three times the frame rate in one case.

## 7. Conclusion

ClusterGL is a new system to allow unmodified OpenGL applications to use a multiple monitor display wall. ClusterGL includes optimisations that reduce network traffic and increase frame rates. In tests to date, it achieves significant performance improvements over other approaches that support unmodified applications. The system is generic enough to be suitable for a range of applications and hardware. ClusterGL is available via Google Code `http://code.google.com/p/clustergl2`.

## References


[AMD09] AMD: Eyefinity, Sept. 2009. http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/AMD-EYEFINITY-TECHNOLOGY/Pages/eyefinity.aspx *Accessed: 3 Mar 2011.* 2

[anh05] ANHOLT: anholt timedemo, Apr. 2005. http://dri.freedesktop.org/wiki/Benchmarking *Accessed: 3 Mar 2011.* 6

[ARA09] ADAMSON B., ROCA V., ASAEDA H.: RFC5740: NACK-Oriented Reliable Multicast (NORM) transport protocol, Nov. 2009. http://tools.ietf.org/html/rfc5740 *Accessed: 3 Mar 2011.* 5

[CGF*01] CROWCROFT J., GEMMELL J., FARINACCI D., LIN S., LESHCHINER D LUBY M., MONTGOMERY T., RIZZO L., TWEEDLY A., BHASKAR N., EDMONSTONE R., SUMANASEKERA R., VICISAN L.: RFC3208: PGM reliable transport protocol specification, Dec. 2001. http://tools.ietf.org/html/rfc3208 *Accessed: 3 Mar 2011.* 5

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 436–452. DOI 10.1109/TVCG.2008.104. 3

[FE09] FECHTELER P., EISERT P.: Depth map enhanced macroblock partitioning for H.264 video coding of computer graphics content. In *Proceedings of the 16th IEEE international conference on image processing* (Piscataway, NJ, USA, Nov. 2009), ICIP'09, IEEE Press, pp. 3405–3408. DOI 10.1109/ICIP.2009.5413851, issn 1522-4880. 6

[HEB*01] HUMPHREYS G., ELDRIDGE M., BUCK I., STOLL G., EVERETT M., HANRAHAN P.: WireGL: A scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 129–140. DOI 10.1145/383259.383272. 3

[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. *ACM Trans. Graph. 21*, 3 (July 2002), 693–702. DOI 10.1145/566654.566639. 3

[Hun09] HUNKIN P.: The BSOD Network Visulisation Tool, June 2009. http://research.wand.net.nz/software/visualisation.php *Accessed: 3 Mar 2011.* 1

[IRK05] ILMONEN T., REUNANEN M., KONTIO P.: Broadcast GL: An alternative method for distributing OpenGL API calls to multiple rendering slaves. In *WSCG: Journal of The Winter School of Computer Graphics, volume 13, Plzen, Czech Republic* (2005), Science Press, pp. 65–72. http://dblp.uni-trier.de/db/conf/wscg/wscg2005.html#IlmonenRK05 *Accessed: 3 Mar 2011.* 3

[JBG*10] JURGELIONIS A., BELLOTTI F., GLORIA A. D., LAULAJAINEN J.-P., FECHTELER P., EISERT P., DAVID H.: Testing cross-platform streaming of video games over wired and wireless LANs. In *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops* (Washington, DC, USA, 2010), WAINA '10, IEEE Computer Society, pp. 1053–1058. 6

[JRJ*06] JEONG B., RENAMBOT L., JAGODIC R., SINGH R., AGUILERA J., JOHNSON A., LEIGH J.: High-Performance dynamic graphics streaming for scalable adaptive graphics environment. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 108. DOI 10.1145/1188455.1188568. 2, 3

[LBH07] LORENZ M., BRUNNETT G., HEINZ M.: Driving tiled displays with an extended chromium system based on stream cached multicast communication. *Parallel Computing 33*, 6 (2007), 438 – 466. Parallel Graphics and Visualization DOI 10.1016/j.parco.2007.02.016 http://www.sciencedirect.com/science/article/B6V12-4N7RW3T-1/2/404a2ead18e70a0f106b7bcb26269fe8 *Accessed: 3 Mar 2011.* 2, 3

[lGA10] LOUP GAILLY J., ADLER M.: A massively spiffy yet delicately unobtrusive compression library, Apr. 2010. http://www.zlib.net/ *Accessed: 3 Mar 2011.* 5

[MKDF04] MARTIN K. E., DAWES D. H., FAITH R. E.: Distributed multihead X project, June 2004. http://dmx.sourceforge.net/*Accessed: 3 Mar 2011.* 2

[Mol03] MOLOFEE J.: The NeHe OpenGL tutorial, Sept. 2003. http://nehe.gamedev.net. 6

[Obe10] OBERHUMER M. F. X. J.: LZO version 2.0.4, Oct. 2010. http://www.oberhumer.com/opensource/lzo/ *Accessed: 3 Mar 2011.* 5

[Ope05] OPENARENA:, Apr. 2005. http://www.openarena.ws *Accessed: 3 Mar 2011.* 6

[Pos80] POSTEL J.: RFC768: User Datagram Protocol, Aug. 1980. http://tools.ietf.org/html/rfc768 *Accessed: 3 Mar 2011.* 3, 5

[Pos81] POSTEL J.: RFC793: Transmission Control Protocol, Sept. 1981. http://tools.ietf.org/html/rfc793 *Accessed: 3 Mar 2011.* 3, 5

[SJNC09] SHI S., JEON W. J., NAHRSTEDT K., CAMPBELL R. H.: Real-Time remote rendering of 3d video for mobile devices. In *Proceedings of the seventeen ACM international conference on Multimedia* (New York, NY, USA, 2009), MM '09, ACM, pp. 391–400. 6

[SL96] SANJOY P., LIN J.: RMTP: A Reliable Multicast Transport Protocol. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation.* (San Francisco, CA , USA, Mar. 1996), IEEE. 5

[Wai11] WAIKATO UNIVERSITY: The symphony cluster, 2011. http://symphony.waikato.ac.nz/ *Accessed: 3 Mar 2011.* 1

[WVK*01] WHETTEN B., VICISANO L., KERMODE R., HANDLEY M., FLOYD S., LUBY M.: RFC3048: Reliable multicast transport building blocks for one-to-many bulk-data transfer, Jan. 2001. http://tools.ietf.org/html/rfc3048 *Accessed: 3 Mar 2011.* 5