Exposing Application Graphics to a Dynamic Heterogeneous Network

John Stavrakakis
jstavrakakis@vislab.usyd.edu.au

Zhen-Jock Lau
zhenjock@gmail.com

Nick Lowe
nickl@vislab.usyd.edu.au

Masahiro Takatsuka
masa@vislab.usyd.edu.au

ViSLAB, The School of IT
The University of Sydney

ABSTRACT

With the abundance of high performance personal computers, rendering thousands to millions of polygons per second is an inexpensive task. In recent years, there have been advances in networking technologies that have enabled applications to become distributed over a network and many applications require this functionality. These applications can range from driving a large display, collaborating over an internet, or supporting pervasive environments. Solutions currently exist in providing graphics over a network. However, they are usually targeted to satisfy particular problem domains or are otherwise difficult to adopt as applications require major adjustment. OpenGL ® is a graphics drawing library. Although many applications have made use of this API, few provide direct interaction within networked environments. In this paper we present Lumino, a framework that enables graphics from an existing OpenGL application to become available to a dynamic heterogeneous network. What differentiates Lumino from prior work is that it provides this functionality to existing unmodified applications at a very low level and is capable of supporting flow control, quality and scalability. Moreover, it is targeted at wide adoption and will be released under a Free Software license.

Keywords: Graphics, OpenGL, Network, Remote Rendering, Collaborative Environment, Visualisation, Grid

1 INTRODUCTION

Over the years computer graphics has seen many advances in the area of graphics hardware, software for new functionality in graphics APIs, and algorithms for new rendering techniques in complex modelling. An emerging trend for computer graphics is migration from single to multiple display devices or graphics accelerators. As this trend continues, a network will be needed to support this activity between multiple devices[19].

The OpenGL[17] API is an application programming interface that allows a programmer to access the features of the underlying graphics hardware. It has been accepted as an industry standard under which many applications have been developed. While the needs of local graphics resources have been satisfied, the advent of fast networking technologies has made it feasible to transport graphics over a network. As this became more apparent, applications have begun to utilise proprietary protocols for distributing graphics over a network. This does not extend to existing applications as they would require reimplementation and integration.

WSCG 2005 conference proceedings, ISBN 80-903100-7-9
WSCG'2005, January 31 – February 4, 2005
Plzen, Czech Republic.

Our solution Lumino, addresses the many issues involved with transporting OpenGL graphics over a network. It is able to provide fundamental services including flow control, state management to support dynamically connecting clients and peer-to-peer connectivity. It additionally enables applications making use of the OpenGL API to transparently distribute an OpenGL stream to multiple recipients who are able to dynamically connect and disconnect. As a result, applications need not be reimplemented, a scalable number of clients can share graphics of an application while quality graphics is provided. The parser, code generation, networking, and fundamental algorithms (for state management, encoding, decoding, and network transmission with flow control) are currently implemented in Lumino. Lumino will be made available under GPL[8], as complete OpenGL function coverage remains ongoing work.

The rest of this paper is structured as follows. We first assess existing solutions; outline our approach, present implementation details and their results. Finally, we conclude with a short discussion, and a brief outline of future work.

2 RELATED WORK

In this section we discuss previous work with particular emphasis on GLX [21] and Chromium [11]. These systems provide a foundation for some aspects of Lumino, but differ greatly in global design and application domain. This distinction is clarified in section 3. These systems have a number of features in common

with Lumino, but are not functionally equivalent. Both are freely available, use well-defined non-proprietary protocols for sending rendering command streams over a network, and are designed for applications that use OpenGL documented capabilities of existing proprietary systems.

2.1 GLX

A standard technique for remote display of 3D applications is GLX, the "OpenGL Extension to the X Window System". The X Window System[9] was developed to provide a network transparent user interface with rendering on a remote server. The X protocol is a client-server protocol in which applications are run on the client machine and display requests are sent to a server. This design allows application data and processing to be performed locally with window management and drawing to be handled transparently at the server. It is highly portable because any X client can connect to any X server, regardless of platform or operating system. Moreover, since all communication is handled by the client-side X library (Xlib) and the X server, applications do not have to be network aware.

GLX enables OpenGL applications to draw to windows provided by the X Window System. It is comprised of an API, an X protocol extension, and an X server extension. When using GLX for remote rendering, GL commands are encoded by the client-side API and sent to the X server within GLX packets. These commands are decoded by the X server and submitted to the OpenGL driver for rendering on graphics hardware at the server. Importantly, GLX provides a network transparent means of remote rendering to any X server that supports the extension. It also specifies a standard encoding for GL commands.

Figure 1 illustrates GLX. An application that uses the GLX API can send GL render requests to a remote X server that supports the GLX server extension. GL commands are encoded in the GLX packet, which is itself inserted into an X packet. Any number of clients can connect to an X server, but a client will only ever connect to a single X server. GLX is limited because of these characteristics. Rendering is always necessarily server-side and it cannot support GL command streaming to multiple remote displays.

2.2 Chromium

Chromium is a well established and widely used for rendering on display clusters. It is based on another technology called WireGL [10]. One of the major advantages of Chromium is that it enables users to construct a high-performance rendering system, which is capable of large scale complex rendering. Moreover, it can drive a large multi-display system to display high-resolution images. It is possible to create a remote rendering system based on Chromium and video streaming

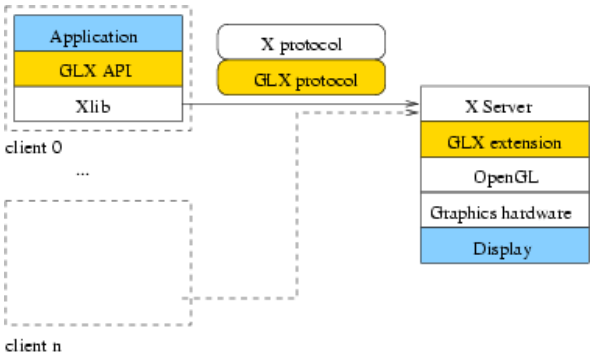

Figure 1: GLX is composed of a client-side API, a protocol for GL command stream encoding, and an X server extension (all components shown in orange). The application resides on the client machine and the display is connected to the server (both indicated in light blue).

or via its reflect SPU. However, this approach would be very inflexible and would not scale well in dynamically supporting multiple clients/rendering tasks.

Figure 2 illustrates Chromium's distribution model. It is a general node-based model for stream processing. A node accepts one or more OpenGL command streams (GL streams) as input and outputs GL streams to one or more other nodes. Each node contains one or more Stream Processing Units (SPUs) that modify the GL stream. Rendering or display can occur at any node in the graph. This depends entirely on whether the SPUs on the node perform rendering or display. One characteristic of Chromium that is not illustrated in the figure is that configuration of the graph is centralized and set at system initialization. This is suitable for dedicated clusters (with fixed, single-purpose resources), but not ideal for grid computing (with heterogeneous resources that are dynamically added and removed, and also available for other services).

Chromium follows the OpenGL Stream Codec (GLS)[6] to encode GL commands. GLS defines methods to encode and decode streams of 8-bit values that describe sequences of GL commands invoked by an application. Chromium, however, employs its own optimized protocol to pack all opcodes into a single byte. Hence, any stream produced becomes specific to Chromium. While this feature enables Chromium to achieve excellent cluster-based rendering performance, the portability of the generated stream becomes a limiting factor for extending Chromium into other OpenGL based projects.

2.3 Commercial remote visualization products

There are a number of commercially available solutions to provide visualization capabilities to remote users, such as SGIs OpenGL Vizserver[18], Teraburst[14] and

Figure 2: Chromium has a flexible configuration system that arranges nodes in a directed acyclic graph (DAG). Each node can take input from, and send output to, multiple nodes. A client node takes GL commands from an application and creates a GL command stream. A server node takes GL commands from another node (and usually renders the stream on local hardware). Because Chromium is usually used for remote display, this diagram shows rendering and display at a server node. However, it is important to note that rendering (and display) can occur at the any node.

Sun Microsystems' VisGrid[13]. However, they all require a proprietary protocol and a communication channel to exchange and control visual information. In contrast, most grid services are based on open source applications. Moreover, many resources are heterogeneous in their nature. Hence, the software infrastructure providing such grid services must allow users and research communities to construct and provide grid resources without being restricted to a particular platform technologies. As many commercial remote rendering solutions are used along with the data grid, computational grid and access grid, they are not a part of grid services. Therefore, users have to access such facilities in a conventional client/server manner and the system is normally configured at the beginning of each session.

Although these commercial and proprietary solutions provide a high and stable performance, many graphics and visualization projects continue to use commodity based approach, such as Chromium, due to the following reasons as described by Samanta et al.[16]:

- **Lower-cost** Price performance ratio of consumer hardware vs. custom architectures

- **Technology tracking** Rate of performance improvement in consumer hardware exceeds that of custom architectures, consumer hardware is also easier to upgrade

- **Modularity & flexibility** Networked systems allow PCs to be added or removed, and enables PCs to be used as other resources not only for rendering

- **Scalable capacity** The aggregation of compute power, storage, and bandwidth capacity in a network of PCs scales easier than that of multiprocessor architectures, which are limited by the complexity of their interconnects.

3 OUR WORK

Current widely used cluster rendering systems are targeted at the provision of a high-performance rendering capability to a local display system. Solutions to remote rendering currently exist. However, they are based on proprietary commercial solutions, and are difficult to integrate with open source grid infrastructures.

We have developed an open source remote distributed rendering system named Lumino. Figure 4 shows the basic system diagram of how Lumino interacts with a client application and remote rendering system. At any given instance, it appears very similar to the Chromium model. However, it is a dynamic system with a fundamentally different processing model also detailed in figure 3.

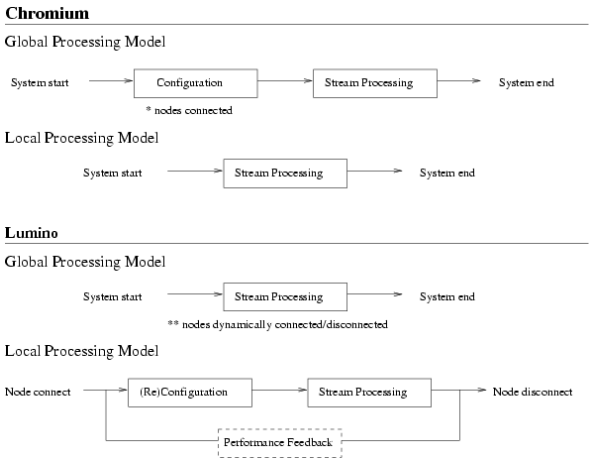

Figure 3: This diagram illustrates the differences between the Chromium and Lumino distributed processing models. Chromium adapts a global processing model composed of an initial configuration stage followed by stream processing on static nodes. Lumino uses a dynamic local reconfiguration model that allows nodes to connect and disconnect at runtime. This removes the need for an initial global configuration stage. Note that node connection and disconnection in this diagram refers to connection to the system (stream processing at each node can involve input and output to/from many other nodes).

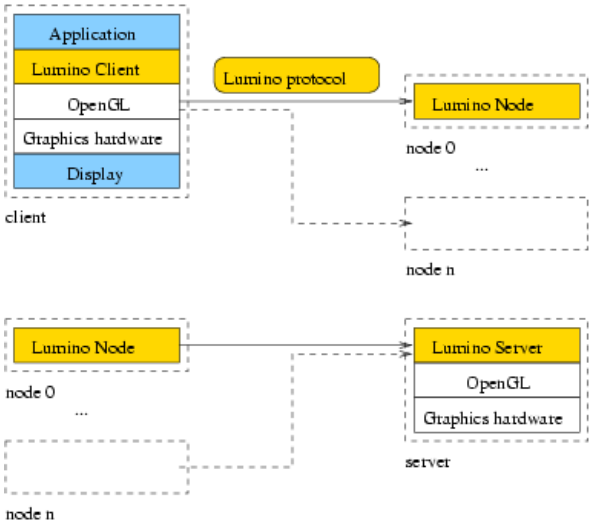

Figure 4: At any given time, a Lumino network configuration looks similar to a Chromium configuration. It has nodes arranged in a graph, in which each node can take input from, and send output to, multiple nodes. A client node takes GL commands from an application and creates a GL command stream. A server node accepts GL commands from another node. It is important to note that this diagram represents a system-wide snapshot and that node connections are dynamic. Also of note is that rendering (and display) can occur at any node.

3.1 Framework

The OpenGL stream is a sequence of serialised OpenGL commands intercepted by the application. Each command in the stream uses a unique identifier to recognise the OpenGL API call it represents, termed an *opcode*. The Lumino framework consists of several components; each strictly requires an interface to an OpenGL stream. For the OpenGL stream, we adopted the GLX protocol as it can readily interface with existing GLX servers. It is also well-defined for interoperability of future OpenGL functionality and its opcode identifiers. To describe processes among machines in the system, we term them producers or consumers of an OpenGL stream.

3.2 State Management

State management is a component that aims to maintain a representation of an applications OpenGL context, that is, the state of the OpenGL machine. This influences how rendering is done, for example which lights are enabled, what are their attributes etc.

State management is required to keep the state of separate OpenGL machines synchronised. This enables the rendering process for *each* machine to produce the same results that are intended by the application. In addition, state management allows any consumer to dynamically connect and render frames of the application.

The OpenGL API contains commands that influence the state of the OpenGL machine. These commands are captured from the stream and appended to an OpenGL state stream; representing the OpenGL context. The state stream must remain consistent, transparent and incur low computational cost. These are the requirements of state management. In being consistent, the state stream remains in order, thus allowing their execution in a new OpenGL context to result in the same graphics state. It is transparent such that only a GL stream is interfaced. State is arranged in an unknown manner, and when a consumer requires the entire state or a state update, a GL stream is returned. That is, no explicit state information is sent.

To simply append state commands to a data structure would lead to a memory explosion. Our backend implementation uses a data structure that can efficiently manipulate commands such that they are deleted when certain conditions are satisfied. For example, when GL commands operate on matrix transformations that are later reset with the identity matrix, the previous transformation is no longer required. There are many such relationships between OpenGL commands that cause redundancy in similar ways. Results indicate that the costs associated with performing checking and reduction can be implemented efficiently with the right data structure or at best with an entire emulation of the OpenGL state machine.

In providing state management, consumers are able to connect at any time. Our interface strictly remains a GL stream, thus consumers need only understand one protocol rather than deal with synchronisation from out-of-band communication. This idea extends the simplicity of designing a system where only graphics is outputted on the network interface. As many good OpenGL applications make heavy use of server side state; clients on low bandwidth connections are able to download the state environment as an OpenGL stream and then experience remote rendering without the impact of network bandwidth.

3.3 Flow Control

Bottlenecks evident between OpenGL applications and the rendering backend are typically dependent on data transfer rates on AGP/PCI buses, execution rate of the application, and deliverable performance of the graphics card. These bottlenecks also exist across networks, the only difference being that the network changes from local AGP/PCI to LAN/WAN. Bandwidth is a major problem for transferring graphics and it remains a critical goal to many existing solutions to minimise bandwidth usage. Prior approaches taken usually result in optimised proprietary protocols and limited extensibility for widespread use. Lumino supports an interoperable means to distribute graphics over a network. In coping with limitations of underlying network capac-

ities, flow control is available to compensate for low bandwidth consumers.

In heterogeneous environments, performance of different graphics hardware can vary greatly. By selecting OpenGL as the abstraction of graphics resources, we are able to use the API to gain performance feedback of the graphics hardware itself. The Lumino processing model permits each consumer to use out-of-band communication channel for relaying performance feedback information. This allows the consumer to control the bit-rate or frame-rate at which the rendering commands are received. While consumer oriented flow control is desirable, the producer may choose to throttle all consumers to a specific rate. This way, a producer remains in control for coordinating the resources of multiple consumers. This mechanism helps the producer, consumer, and the network in only utilising those resources required.

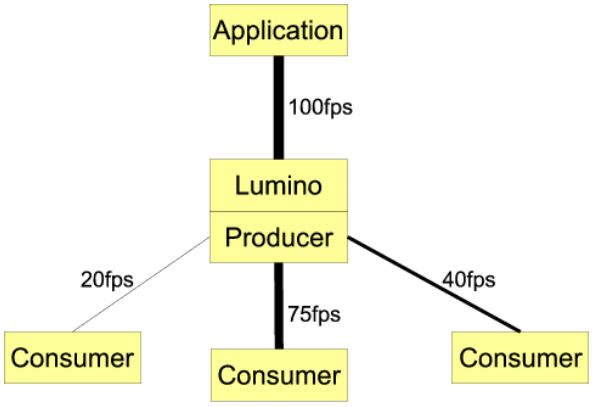

Figure 5: In this figure the application is capable of generating 100 frames per second. Lumino producer is able to negotiate rate control parameters among several consumers to satisfy any of bandwidth, processing or privilege requirements of all concerned.

3.4 Scalability

Scalability of network based graphics solutions can be defined as being resolution scalable, data scalable, consumer scalable or hardware scalable[7]. There are many limitations of existing solutions to provide remote rendering to a scalable number of consumers[20]. This is typically the case as they create centralised environments that put greater load on the server which would otherwise be uniformly distributed on the network. Our initial design sought TCP as a first solution in bringing graphics over a network. TCP offers many benefits to simplify implementation, but lacks point-to-point scalability.

Many P2P (peer-to-peer) solutions [1, 2, 3] allow consumers of a data stream, to become producers of that data stream to other parties. Lumino has applied the idea on sharing data stream, the only difference being that the stream is GL. This is achievable by being able to dynamically monitor and export the OpenGL context on-demand, enabling greater scalability as any consumer can now become a producer. The participation of new consumers is no longer burdening the single original producer; but rather, one of multiple producers as demonstrated in figure 6. By distributing the load among all parties involved, not only does scalability improve, but the requirements of the networks are eased which would otherwise be centralised and cause "hot-spots".

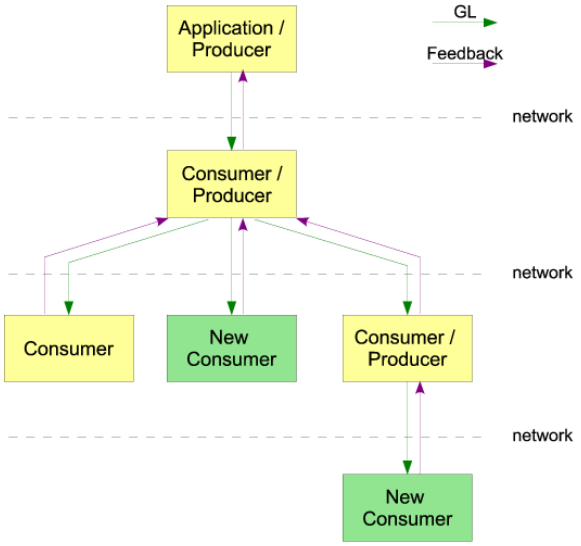

Figure 6: An application may be runnable on a device such as a laptop but is not capable of supporting more than one client due to processing limitations. To host this application for potentially hundreds of consumers, a more resourceful machine can be used to host the GL stream as a proxy to consumers. This alleviates any burden on the device to deliver processing and network resources. Furthermore, the GL stream can extend to other networks that can present the gateway to receiving the stream.

There remains a lingering issue of latency. For how long a chain of consumers will it be infeasible? This primarily depends on the underlying network hardware, but more importantly the amount of data generated by the application. The latency limitations are explored in the following section.

4 RESULTS

4.1 Processing overhead

Applications that run with Lumino are subject to overhead from encoding, state management and network I/O. We use three metrics to measure the performance of applications under Lumino, they are: frame rate, indicating how many frames per second can be rendered, execution rate, being the number of GL commands generated by the application, and byte rate, being the num-

ber of bytes generated after encoding. In this paper, we present experiments using two applications. The XScreenSaver[22] package contains various OpenGL programs that are used as screen savers. These are very simple with respect to demands and coverage of OpenGL. A more complicated and demanding application is the game application Tuxracer[4]. The machines used in testing are illustrated in table 1. Figures 7 and 8 describe the frame rate performance of Molecule and Tuxracer.

Machine	Specification
kat	P4 2.4GHz, 512MB RAM NVIDIA GeForce FX 5200/PCI/SSE2
duck	Celeron 2.2GHz, 256MB RAM ATI Radeon IGP 330
grudge	Dual P4 2.8GHz, 512MB RAM NVIDIA Quadro4 750 XGL/AGP/SSE2
calvin	P4 2.6GHz, 512MB RAM NVIDIA GeForce4 Ti 4600/AGP/SSE2

Table 1: Each machine has a minimum 100Mbps ethernet connection, and a Linux OpenGL driver with hardware acceleration.

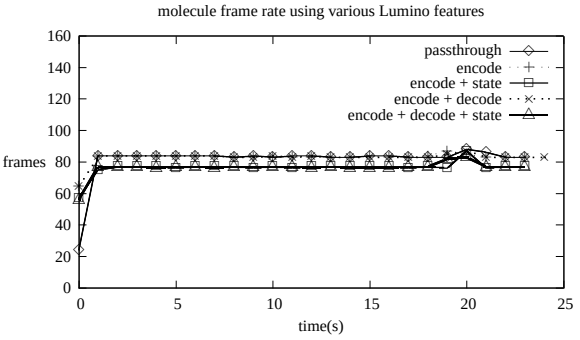

Figure 7: The Molecule application generates a very small amount of data, as a result the impact of encoding, state management and decoding is less noticeable when compared with its original performance.

4.2 Scalability Testing

While Lumino is able to support consumer scalability, the latency effects of intermittent processing between producers and consumers can affect display synchronisation. We devised an experiment that investigates the latency effects as the number of indirect consumers increases. In our efforts to demonstrate heterogeneous networking, we used two LAN environments: the School of IT (SIT), and ViSLAB shown in figure 10. In this experiment the networks and machines all differed other than running Linux (namely Ubuntu[5]) with 32-bit graphics drivers. We ran applications on a PC (kat) on the SIT LAN; the Lumino stream was then transmitted to a PC in ViSLAB (grudge), then to another ViSLAB PC (calvin), then to a laptop (duck) in

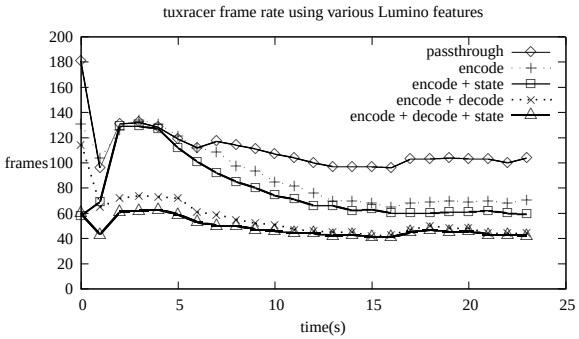

Figure 8: The Tuxracer application, is constantly submitting data to OpenGL. As a result the cost involved with encoding the OpenGL commands becomes a much larger issue than that of state management. It is important to note that a typical producer will be hosting the application by only performing encoding and state management. The lines with decoding indicate that decoding an already encoded OpenGL stream is marginally small. Consumers receive an OpenGL stream that is already encoded, thus the processing overhead is much smaller for them.

SIT, and finally to the original SIT PC kat. Figure 9 depicts this arrangement.

Figure 9: The Lumino stream begins at kat, it is forwarded to grudge, calvin, duck and finally again to kat. With the initial and resulting stream simultaneously rendered at kat we are able to observe the effects of latency over a 100Mbps network.

As the communication of Lumino remained TCP, the round-trip-times of each hop were measured with ping of maximum IPv4 sized packets shown in figure 2.

ping	rtt (ms)
kat->grudge	32.338
grudge->calvin	11.801
calvin->duck	31.399
duck->kat	35.989
total	111.527

Table 2: Average round-trip-time between machines detailed in table 1.

We modified gltext, to render a string representing the time in milliseconds. The round trip time difference for this application is measured by taking a snapshot of both renders. Shown in figure 11.

Our theoretical round trip time was measured at approximately 111ms. This did not explain the latency

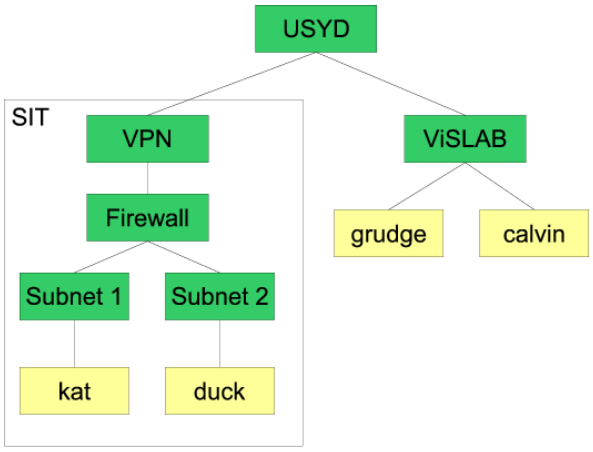

Figure 10: The organisation of the SIT and ViSLAB networks are bridged by a much larger USYD network, each uses different technologies including both Gbps and Mbps. We present these results knowing that the communication between all machines is limited to 100Mbps.

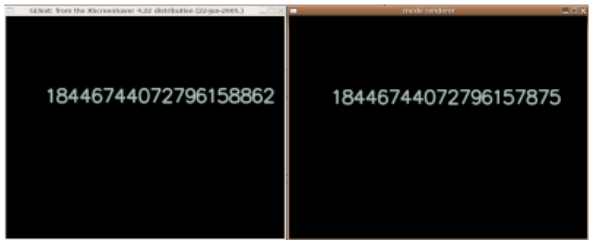

Figure 11: The left render is performed locally, whilst the right has been transmitted through 4 hops before rendering. The numbers from both indicate the current time from the source; the difference represents the latency which is approximately 987ms.

presented here. The factors that introduced latency include:

- Firewall processing - for the interests of SIT, packets are delayed at the firewall for security.
- TCP latency and processing - under TCP, processing of packets must continue to the application level, considering the burden on each machine there will be delay. In each machine the GL stream is received, the data is moved up through to the application level, and then pushed back down the stack for forwarding.
- Higher data throughput - the amount of data is proportional to the latency cost as a result of both transmission and additional processing.
- Incurred cost of state management has mostly affected the less powerful machines, namely the laptop duck. State management remains a backend modular component, as such other approaches are yet to be evaluated.

In exploring a more demanding application, we found Tuxracer provides a more network intensive latency test under the same conditions, shown in figure 12.

Figure 12: As before, the left indicates the local render, while the right has been transmitted through 4 hops before rendering. Using the rendered timer, the difference can be observed to be about 480ms; on average we found it to be approximately 940ms.

As Tuxracer incurs a similar latency cost with greater bandwidth requirement, we can conclude that Lumino is scalable, but remains bound to the underlying network technology.

To improve on the scalability, we are able to implement a multicast based replacement over TCP in a similar manner to Broadcast GL[12]. However, support for multicast networks are limited over an internet which would otherwise require reservation of IP addresses. In consideration of the user, this is not a requirement but an option.

5 CONCLUSION AND FUTURE WORK

We have presented Lumino, a framework that enables graphics from an existing OpenGL application to become available to a dynamic heterogeneous network.

Solutions available to providing graphics over a network are able to offer flexibility in their own domain. However, they exist with proprietary interfaces or have limited applicability in the broad resolution of network based graphics.

We have defined a unique and novel problem statement that Lumino addresses. Lumino is built from the aspects of existing solutions, the implementation of peer-to-peer scalability, flow control and dynamic connectivity are new features founded by Lumino.

Lumino already maintains large application support via the OpenGL API. It is complementary to existing technologies like that of Chromium and VNC. Moreover, it is established as a working product and model.

Its future development envisages innovative uses for transporting graphics over a network. As such, there are many research challenges that await Lumino, one of them describing a realtime system.

5.1 Future Work

There are many possible outcomes that Lumino is able to offer. In understanding that existing solutions pro-

vide focused objectives for application domains we are hoping to use Lumino to bridge such technologies together

- Transparent collaboration - applications are usually written without concern for network interoperability. As the software becomes a useful tool, VNC is used to view and control the application remotely.
- Lumino currently provides a one-to-many arrangement from single producer to multiple consumers. We are in the process of implementing a many-to-one and many-to-many arrangement, such that one consumer can have multiple producers for a single OpenGL context. We can interpret the scenario as "compositing" OpenGL streams. This permits geometry from one application to mix with geometry from another creating a hybrid rendered image. It presents an interesting gateway to a new breed of network enabled graphics applications.
- As the amount of geometry used in the render generates greater bandwidth, Lumino is not data scalable. The contrasting solutions are remote frame buffer such as VNC[15] where it is not resolution scalable. We aim to find a method by which we can gradually alter the bandwidth usage from sending geometry to sending images. This slider like control is desirable for consumers on heterogeneous networks.

REFERENCES

[1] BitTorrent. http://www.bittorrent.com/, 2005.

[2] eMule. http://emule.org/, 2005.

[3] The FreeNet Project. http://freenet.sourceforge.net/, 2005.

[4] Tux racer. http://tuxracer.sourceforge.net/, 2005.

[5] Ubuntu - linux for human beings. http://www.ubuntulinux.org/, 2005.

[6] Craig Dunwoody. The opengl® stream codec: A specification. Technical report, Silicon Graphics, Inc., October 1996.

[7] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: a fully scalable graphics architecture. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 443–454, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[8] Free Software Foundation. Gnu general public license. http://www.fsf.org/copyleft/gpl.html.

[9] X.Org Foundation. About the X Window System. http://www.x.org/X11.html, 2005.

[10] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable diaplays. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page Article No. 30, Washington, DC, USA, 2000. IEEE Computer Society, IEEE Computer Society.

[11] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, July 2002.

[12] Tommi Ilmonen, Markku Reunanen, and Petteri Kontio. Broadcast gl: An alternative method for distributing opengl api calls to multiple rendering slaves. In *WSCG'2005: The Journal of WSCG*, volume 13, Plzen, Czech Republic, 2005. Science Press.

[13] Sun Microsystems. Sun fire^{TM} visual grid system architecture - building scalable and flexible high-end visualization systems. Technical white paper, Sun Microsystems, November 2003.

[14] TeraBurst Networks. Immersive visualization theater connectivity over the wide area network. http://www.teraburst.com/technology/wavs.pdf, October 2002. http://www.teraburst.com/technology/wavs.pdf.

[15] RealVNC Ltd. RealVNC: the original open-source cross-platform remote control solution. http://www.realvnc.com, 2005.

[16] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, New York, NY, USA, 2000. ACM Press.

[17] Mark Segal, Kurt Akeley, Chris Frazier, Jon Leech, and Pat Brown. The opengl® graphics system: A specification. Technical report, Silicon Graphics, Inc., October 2004.

[18] Silicon Graphic, Inc. Opengl® vizserver 3.1: Appliation-transparent remote interactive visualization and collaboration. White paper, Silicon Graphic, Inc., Mountain View, CA, USA, April 2003.

[19] John Stavrakakis, Nick Lowe, Masahiro Takatsuka, and Zhen-Jock Lau. An On-demand Streaming Protocol for 3D Graphics on the Grid. APAC 05, 2005.

[20] John Stavrakakis, Masahiro Takatsuka, Nick Lowe, and Zhen-Jock Lau. Lumino: Platform independent remote distributed rendering framework. http://wiki.vislab.usyd.edu.au/moinwiki/Lumino, 2005.

[21] Paula Womack and Jon Leech. Opengl® graphics with the x window system®: Version 1.3. Technical report, Silicon Graphics, Inc., October 1998.

[22] Jamie Zawinski. XScreenSaver: A screen saver and locker for the X Window System. http://www.jwz.org/xscreensaver/, 2005.