EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

A novel control mechanism for distributed stream rendering

J. T. O'Brien[1] and R. S. Kalawsky[1]

[1]Department of Electronic & Electrical Engineering, Loughborough University, United Kingdom

Abstract

This paper describes a new control mechanism for distributed rendering. Control mechanisms have previously been widely used in many fields from autonomous robots to video streaming. Their use in video streaming has allowed quality-of-service, user orientated, transmission of videos across changing transport networks. We show how control mechanisms can be applied to distributed rendering to provide responsive, user orientated visualisations. The control mechanisms are implemented as an extension of Chromium, a stream processing framework for OpenGL. The system should also be applicable to other stream based render systems with a significant benefit. The new system allows distributed rendering to be modularised into specialist units that self organise work load to meet the demands of the user.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Distributed/network graphics

1. Introduction

Scientific visualisation has become a necessity for understanding large data sets generated by scientific instruments such as telescopes, microscopes, particle accelerators, and imaging machines [McC88]. Automated data analysis and reduction can play a role in this process, but understanding is only currently achieved by human interpretation of the visualisation.

Interactive visualisation requires a consistent, low latency representation at a frame rate in excess of 10fps (frames-per-second). Further more, for a user to feel that a system is responding interactively, an overall response time of less than 20ms (between input and screen update) is required [KOC05].

Providing interactive visualisations of very large data sets presents a challenging problem. Whilst desktop graphics have dramatically improved in recent years, few researchers have sufficient local resources for high-fidelity rendering of such large visualisations. One example of a visualisation problem on this scale is the large time-varying Computed Tomography (CT) scans used in the diagnosis of cardiovascular conditions. The resulting multi-gigabyte data sets require specialist volume rendering facilities to achieve interactive rendering [PHK*99]. Advances in remote rendering have sought to provide desktop access to specialist visualisation hardware. However, quality-of-service (QoS) for such access has not been provided.

Existing remote rendering techniques have focused on a client-server paradigm, remotely serving rendered images or scene graph descriptions for display on a local client. Increasingly, this paradigm has become insufficient for describing large geographically distributed visualisations [SB03]. These large visualisation make use of modern distributed rendering techniques to harness geographically distributed resources [BTL*00, Bet00]. Often this is achieved through custom applications targeted to a particular problem domain (an example is Visapult [Bet00]). Alternately transparent distributed rendering can be achieved by operating on the graphical API stream of an application. One such system is Chromium [HHN*02], which dynamically links a replacement OpenGL library to the application during initialisation. Chromium provides a distributed stream rendering framework, where nodes can be configured in arbitrary acyclic directed graphs. However, Chromium is targeted at high-speed LAN cluster systems, lacking a management system to effectively operate in wide area networks.

To take advantage of these improvements we have developed a control framework for distributed stream rendering that automatically adapts the rendering process to changes

in resource capabilities. We analyse distributed rendering as an information distribution problem, showing that modules within a distribution must cooperate to achieve interactive rendering. The framework is designed to adjust the graphical command stream, in this case OpenGL, transmitted to nodes within a distributed network to best match the resources available according to user preferences on frame rate, response time and scene quality. The techniques used should be applicable to any network transport system for graphical APIs, a test system has been developed using Chromium. Chromium's stream processing model makes it an ideal candidate with which to develop adaptive modules for our framework.

2. Background

Research into rendering across computer networks has been largely divided between remote and distributed rendering. Remote rendering has focused on targeting a visualisation to network and client resources, whilst distributed rendering has targeted the efficient distribution of visualisation workload across a collection of homogeneous resources.

2.1. Remote Rendering

We define remote rendering as the process of serving abstract visualisation objects (AVO) to a local client. AVOs can take the form of 2D rendered images or other abstract objects such as volumetric data or polygon meshes. Transportation of 2D rendered images encapsulates remote frame buffering, where a client provides remote terminal access for a user. One of the first examples of this was VizServer [Oha99], which permitted remote visualisation of graphics pipes on SGI machines. The performance of remote frame buffering is largely governed by the speed at which images can be transmitted to the client. The use of image compression [MC00] and video stream compression methods [ESEE99] have been explored as methods to improve delivery of rendered images. To reduce the number of interaction events transmitted back to the server, client based GUI components have been suggested [ESE00].

The transmission of more complex AVOs (including polygon meshes and texture data) is often more bandwidth intensive than transmitting a final rendered image of equivalent detail. However if the client is provided with sufficient information about the 3D scene, local 3D scene exploration can be achieved without further network communication. Client-side exploration is only possible given sufficient rendering resources. Transmitting AVOs with a level-of-detail matched to the resource requirements of the client workstation has been suggested as a solution [Mar00]. Restricted client based interaction is also possible using image-based rendering techniques to reconstruct the 3D scene using rendered view points from the server [BTL*00, Bet00]. Any changes in parameters governing the generated AVO, made either by the user or visualisation, require the AVO stored on the client to be updated. These updates need to occur quickly to maintain a feeling of interactivity with the visualisation, which can place high bandwidth requirements on the network.

The application domain of a visualisation can impose restrictions on how AVO serving can be achieved, in particular on the level-of-detail allowed within a visualisation. For example medical domains often demand visually lossless rendering. Such requirements may not be constant within a visualisation. For example, when searching for a region of interest or interactively exploring a visualisation lower quality representations may be tolerated [ESEE99].

The variety of possible AVOs and differing performance requirements has meant that AVO serving frameworks have largely been customised solutions targeted at a particular application. Developments in distributed stream rendering presents an opportunity for transparent AVO adaptive systems.

2.2. Distributed stream rendering

The intensive bandwidth requirements of distributing large visualisations have led many distributed render system to favour distributed scene graph approaches [AGMR02, SWNH03, VBRR02, vdSRG*02]. Such network bandwidth problems are also present across the memory bus within workstations. As a result immediate mode graphical APIs have developed to provide retained mode components for managing textures and vertex meshes [WSND03]. The flexibility of providing distribution at this level has many advantages, not least of which is application transparency, as demonstrated by Chromium. Chromium represented a generalisation of the ideas developed in WireGL [HEHE01], with the goal of aggregating the power of commodity PC graphics cards without imposing a specific rendering topology. As a transportation protocol for OpenGL, WireGL drew comparison to the stream processing model of immediate mode graphics in which finite resources must operate on a continuous sequence of primitives [HHN*02]. Within Chromium, stream processing nodes consist of two parts: transformation and serialisation.

The serialisation portion of each node receives one or more OpenGL streams and output a single OpenGL stream. A serializer could take one of two forms: client or server. A node with no input streams was a client and must produce an OpenGL stream from a standalone application (already serial). A node with incoming edges was a server and must manage multiple incoming streams, performing context switching between them. The transformation portion of each node receives a single stream of OpenGL commands and output zero or more OpenGL streams. OpenGL stream transformations were performed by Stream Processing Units (SPUs). Each node may chain together multiple SPUs, with

each SPU in the chain performing some arbitrary transformation on the incoming stream and outputting the stream. An inheritance model was used for the SPUs through a function lookup table, which allowed child SPUs to implement a subset of OpenGL commands, those not implemented would be processed by the parent SPU. A common parent SPU was the *passthrough* SPU, which passed all of the OpenGL calls on to the next SPU in the chain.

Chromium has no inherent facility to monitor the effects of output from an SPU and no ability to monitor other SPUs in the network. As a result, transformation behaviour remains static within current SPUs. Such a constraint is sufficient within homogeneous cluster facilities, but is unsuitable for guaranteeing services in dynamic resource environments.

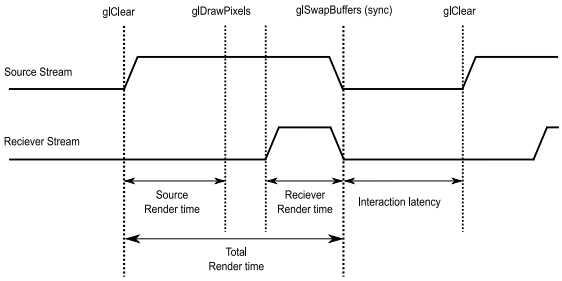

Figure 1: *OpenGL stream timing analysis. Synchronisation between source and receiver is shown as occurring at the end of each frame.*

3. Control Mechanisms

Providing a quality-of-service system for a OpenGL stream has commonalities with application-layer quality-of-service (QoS) control mechanisms developed for video streaming. The objectives of video streaming QoS can be defined as avoiding congestion and maximising video quality in the presence of packet loss [WHZ*01]. These objectives hold true for distributed rendering, but the factors governing quality and throughput are more complex. As well as packet loss (in the presence of lossy UDP based protocols) and network bandwidth, hardware rendering performance can also have an effect on both visualisation quality and frame rate. Another more imposing consideration is that unlike the frames of a video stream, rendered frames are not periodic. Frames are instead triggered by scene changes or display corruption. Issues such as jitter in video streaming, which are readily determined by the expected arrival period of a frame in a video stream, are non-trivial in the case of distributed rendering.

Control schemes to meet the two objectives of congestion avoidance and image quality can be classified as receiver-based, source-based and hybrid control schemes [WHZ*01]. Sections 3.1, 3.2, and 3.3 introduce each of these schemes and present new strategies for their use in distributed rendering. The classification of receiver and source helps move distributed rendering beyond the client-server rendering paradigm. When discussing receiver and source control strategies in the subsequent sections, these entities are considered to be nodes in a directed acyclic graph connected by a common edge. Some control mechanisms may impose certain additional conditions such as the availability of local rendering resources.

3.1. Receiver-based control

A receiver is responsible for maintaining display quality and render performance using only receiver-based mechanisms. In order to avoid confusion between receiver and source based control mechanisms, the receiver is assumed to terminate the stream and act as a display client for the application user.

A receiver has no knowledge of the causes of delays upstream as it has no method for determining the expected arrival time of each frame. The delay could be natural, introduced by the sporadic interactions of the user (interaction latency); or artificial, caused by network or render performance. However, the receiver does know the time taken to render a frame downstream, assuming frame synchronisation is performed at the end of each frame. This is illustrated by the timing diagram in Figure 1, which depicts the source stream acting as a remote framebuffer, locally rendering the OpenGL frame and using glDrawPixels to transmit the frame to the receiver.

Under these assumptions, receiver control mechanisms can take two forms: stream switching and stream filtering. Stream switching is possible when multiple source streams are available. These could include a complete AVO stream, reduced level-of-detail AVO stream, or remote frame buffer; each reducing scene complexity. Each stream requires additional render resources on the server, but when serving multiple clients of varying resource power such mechanisms would be cost effective.

Stream filtering involves performing local filtering on the AVO stream to reduce scene complexity to better match render resources. Possible AVO reduction methods have a processing cost associated with them. OpenGL quality hints provide information to the OpenGL implementation on the relative importance of render speed over quality. As single API calls, OpenGL hints can be easily filtered from the application AVO stream, however the effects of such calls are implementation specific and some OpenGL implementations will ignore them completely [WSND03]. Other methods which reduce scene complexity by altering texture or geometry structures are more costly to perform. The key to stream filtering at this level is an effective model of how elements of the scene effect overall complexity and how reduction methods can be applied. In section 5 we examine a

novel adaptation method, which is cost effective in providing scalable complexity.

In general, receiver control mechanisms work well for situations where poor graphic resources sit on a high speed network. The strategies make inefficient use of networking resources as redundant scene complexity is transmitted to the receiver.

3.2. Source-based control

Within source based control mechanisms, a source node is responsible for ensuring that nodes downstream receive and render the OpenGL scene correctly, maximising both render performance and quality. Assuming that a source node has no other streams feeding it, then source control mechanisms are aware of the total time taken to render a frame, including network delivery, and the current interaction latency (refer to Figure 1). However, the source control mechanism is not explicitly able to distinguish between network and receiver render performance downstream.

Within video streaming there are two basic approaches for determining network bandwidth: probe based and model based. A model based approach estimates the network throughput using a suitable mathematical model of the networking protocol, this is commonly used for TCP connections where information on packet loss and round trip times are available [WHZ*01, FF99]. Using a protocol model, source control mechanisms are able to determine the dominant factor in the observed render time and act accordingly. Probe based approaches interrogate the network or receiver to determine the factors creating the observed render time. Third-party applications can be used to profile network connections to determine network performance during initialisation [Mar00]. However, changes in network conditions will not be detected by one time profiling such as this. Within a render stream, information on network performance can be obtained through appropriate use of synchronisation barriers.

When a barrier is established within Chromium, the framework guarantees that all OpenGL commands previously made have been delivered and executed by all clients before proceeding. Therefore, the time required to deliver and execute *glDrawPixels* can be determined by placing a barrier directly after the call. This time represents both the network delivery time and the time to decode and display the pixels.

Given that the source control mechanism can determine the dominant factor between transmission and render time, the control mechanism can employ more powerful strategies than receiver based mechanisms. A source node is free to switch between serving a complete or reduce AVO stream to the client, or act as a remote frame buffer. Acting as remote frame buffer would reduce both network congestion and client render time for complex dynamic scenes.

When networking performance is the dominant factor, the source control mechanism is able to employ compression techniques and level-of-detail strategies to reduce frame size. As well as compression of pixels, geometry compression can also be applied [PBCK05].

Source nodes have no method for determining the render time observed by the receiver, so level-of-detail scaling or compression cannot accurately be applied to the geometry or texture of a visualisation scene. However, when operating as a remote frame buffer it is possible to calculate the decompression time of the receiving node, if time symmetric codecs are used such as JPEG. A source node can exploit this information to model the decompression time on the receiving node. Alternately decompression can be performed in parallel by a separate thread, allowing the receiver to decode the current frame, whilst receiving the next. A synchronisation barrier placed after the transmitted pixels would then allow the source node to determine transmission time for the pixels.

Source-based control mechanisms allow the graphical AVO stream to be adapted in response to changing network conditions. These mechanisms require a complete round trip on every frame to perform the frame synchronisation necessary in determining networking performance. In high latency networking environments this constraint would become a dominating factor in the visualisation performance. A simple solution would be to only measure bandwidth occasionally or when a change in frame rate occurs at the AVO source. However this may further disrupt the user by introducing jitter into the visualisation. A hybrid control mechanism, where both source and receiver nodes co-operate, would allow modules to measure network performance without the use of synchronisation barriers.

3.3. Hybrid control

Hybrid control mechanisms can help ensure that all nodes in a distributed render network have access to detailed information on transfer and render performance. The previous sections have demonstrated how a receiving node, acting as a render client, can observe render performance of the hardware resources present at the node. Similarly, source nodes attached to the application are aware of how quickly an application is generating render frames by measuring the time between the end and start of each frame. By transmitting performance and quality-of-service information through the AVO stream or a separate control stream, all nodes in a distributed render network are given access to this information and are better able to adapt the visualisation to available resources.

Each node in a distributed render system contains at least two pieces of unique information that is not observable from any other node: the time between the end of a frame and the start of the next frame, referred to here as the frame latency;

and the time taken between the start of the frame and the end of the frame, referred to as the render time. By understanding the role of each node in the system, control mechanisms can adapt the AVO stream in response to this timing information. For example, network bandwidth can be determined by subtracting the frame latency and render time observed by the source node from the frame latency observed by a receiver (see Figure 1, without frame synchronisation points).

Source nodes cannot simply request information from receiving nodes downstream through OpenGL calls as this would interrupt the AVO stream with unnecessary synchronisation points. Alternately, the frame latency and render time information can be passed back through the network connections from receiver nodes to source nodes (refer to Figure 3) as a separate asynchronous control loop.

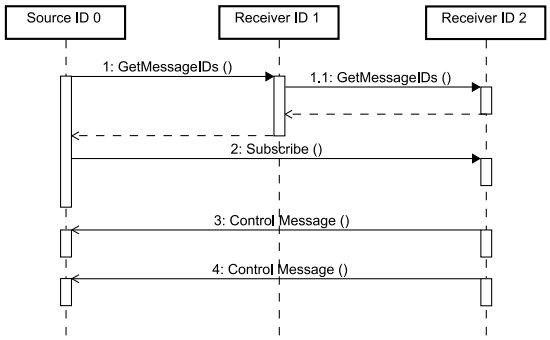

Figure 2: *Sequence diagram for subscribing to a message in the control framework. The command* GetMessageIDs *can be used to obtain a list of supported messages at each node, before subscribing to messages.*

4. Architecture

The development of a distributed render system capable of hybrid control requires an efficient feedback loop to relay information to relevant nodes in a render system. It is important that message passing occurs asynchronously to the render stream to avoid introducing round trip delays into the rendering stream. Care must also be taken to avoid overwhelming the network and nodes with messages. We propose a simple publish-subscribe event system to achieve this, which is discussed in the following section. A number of modules have been constructed as SPUs to demonstrate the effectiveness of the system. These are not designed to be exhaustive as different visualisation techniques will require specific modules.

4.1. Publish-Subscribe Control Messages

A control message framework was instrumented for Chromium to facilitate experiments with hybrid control mechanisms. The framework is designed around a publish-subscribe system to reduce network load; network traffic is reduced by only transmitting messages which source modules have registered an interest in. A typical message exchange to subscribe and receive a message is shown in Figure 2. Each node is assigned a unique ID by the configuration server (Chromium mothership) during initialisation, the ID is used to specify the receiving SPU to subscribe to, and allow source SPUs to distinguish the origin of a control message. A message consists of a unique ID representing the message type, the ID of the source and destination node, the length of the message, and the message payload. It is up to the receiving SPU to unpack and interpret the message payload. Nodes are allocated ID's in ascending order from the first source node to the final receiver node. The destination ID of a message represents the lowest ID of the nodes subscribed to that message type. Such that, the destination ID is effectively operating as a time to live for the message, rather than a list of node IDs.

A message handling routine runs as a separate thread on each node in the distributed system to dispatch received messages to subscribed SPUs. Messages are passed to each SPU through a callback function, which is specified when the SPU initiates a subscription. If the destination ID of a received message is less than the value of the node's ID, the message is relayed to all source nodes attached to the node.

Using this new system, source nodes are able to adjust the AVO stream in response to rendering and network performance observed by receiving nodes. However nodes not attached to the visualisation application are still unaware of how quickly each frame must be delivered to the user; and none of the nodes have any indication of the relative importance of visualisation quality and speed within the distributed render system. A quality-of-service extension was added to the OpenGL API to achieve this. As well as supporting the *Subscribe* and *GetMessageIDs* functions (refer to Figure 2), the following two functions are proposed:

- glRequestedUpdateRateQoS(rate)
- glCurrentInteractionRateQoS(rate)

In order to minimise the impact on network performance, the parameter for each function is sent as a single byte, representing quantised floating point value between 0.0 - 1.0. The parameter is used to weight the target frame rate against a user configured maximum. The *RequestedUpdateRate* informs SPUs down stream of the most important factor within the current scene: speed or quality. The assumption is made that quality and speed form a linear continuum of possible rendering options. A parameter of 0.0 would suggest that quality is the most important factor; conversely a parameter of 1.0 would suggest that render speed is of most importance.

This approach is similar to that of glHint [WSND03], but differs in that a fine granularity of rendering performance between GL_NICEST and GL_FASTEST is made explicit. The approach is based around the interaction model used for determining level-of-detail rendering within VTK [SML98].

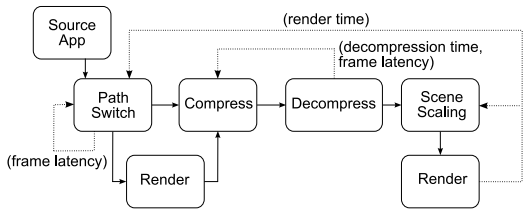

Figure 3: *Distributed stream rendering architecture. Each box is a separate module or SPU within the render network. Dashed lines represent control messages, solid lines represent the OpenGL stream.*

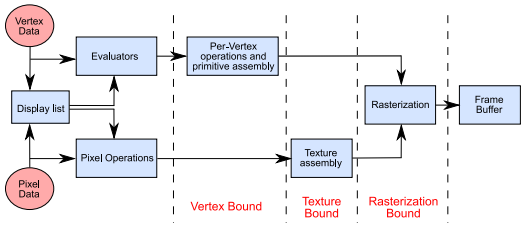

Figure 4: *A simplified non-programmable OpenGL pipeline showing possible bottlenecks in the pipeline, adapted from [WSND03].*

The *CurrentInteractionRate* acts as a supplement to the *RequestedUpdateRate*, and represents the quantised time interval between render frames (interaction latency). The value allows SPUs to decide the current context of the OpenGL stream. To illustrate this, consider an adaptive compression SPU, which adapts image compression to available bandwidth, a *RequestedUpdateRate* of 0.8 would suggest that speed is of most importance over quality overall. The SPU can then use the *CurrentInteractionRate* to decide if speed is still an important factor in the current context. This makes the assumption that during periods of slow interaction a long response time for a higher quality frame is preferable.

The proposed QoS extension allows receiver nodes to better optimise AVO rendering based on the requirements and context of the source. The messages received from nodes operating down stream, indicating the performance of rendering hardware and network characteristics, allowing source nodes to reduce AVO complexity before it is transmitted down stream. Distributed rendering systems which are self-organising can now be designed. These systems automatically use both client rendering hardware and server hardware to provide the best possible interaction.

4.2. Processing modules

Within our controlled rendering system, collections of SPUs are used to deliver an adaptive render system. Each SPU performs some transformation on the incoming stream based on input stimulus from control messages and the observed state of stream. In this way SPUs in the control system have a close relationship to that of neurons within a neural network and exhibit a basic form of learning by altering their behaviour over time. For our experiments we have developed three SPUs: path-switch, compression and scene scaling (refer to Figure 3).

The path-switch module allows the stream to be redirected to new modules when existing down stream modules have been exhausted. For example, when rendering a large retained display list, the path-switch module will initially start by transmitting OpenGL calls straight to the receiver. The path-switch module switches the stream to perform local rendering instead if the receiver becomes incapable of rendering the frame. This places a higher demand on network bandwidth (rather than on the receiver's render performance) and is achieved by affecting the function dispatch table within the SPU. The new QoS hints within the OpenGL stream are used to determine when this action is appropriate. Depending on user preferences, a low quality but responsive scene may be preferable over a high-latency network, in this instance a watchdog routine is placed in the path-switch module to provide high-quality rendered images during idle periods.

Compression modules perform compression of polygon or pixel data. A compression module observes QoS commands within the stream and timing information from the control channel to calculate available bandwidth and target compression level. A JPEG based module has been developed for this purpose. Image compression was chosen over video compression like MPEG, as this provides the ability to quickly switch compression methods (i.e. switching between lossless JPEG-LS and lossy JPEG compression during idle and interactive periods), and can be encapsulated within a WireGL stream.

Scene scaling modules dynamically adapt the level-of-detail of parts of the render stream. Such modules are very specific to rendering techniques used within the visualisation. Bottlenecks in the OpenGL pipeline predominantly occur in three places, as shown in Figure 4. A stream which is vertex bound can be improved by performing mesh decimation on the vertex mesh supplied for the scene; streams which are texture or rasterization bound can be improved by reducing texture and scene resolution respectively. However, the dominant process in the pipeline must first be determined, typically this is performed by repeated experimentation [Spi03]. For automatic stream adaptation to be possible a more formal process must be taken to determining bottlenecks. A scene scaling module has been developed to target direct volume rendering applications. These are predominantly texture or pixel bound, by scaling the image resolution and texture size render time can be reduced. Resolution scaling is achieved by reducing the rendering viewport of the

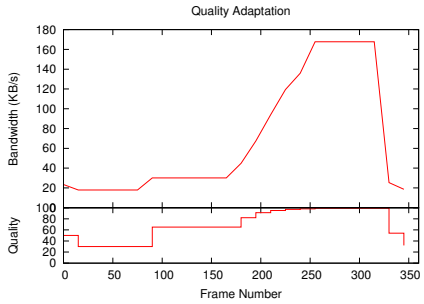

Figure 5: *Compression quality adaptation in response to changing network conditions.*

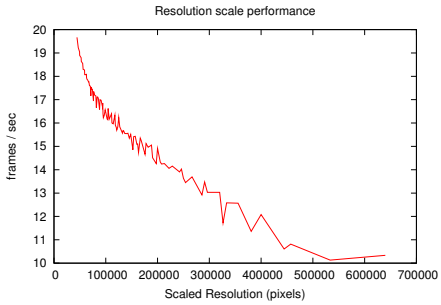

Figure 6: *The effect of resolution scaling on frame rate using an NVIDIA GeForce 7600 graphics card.*

application and quickly rescaling the rendered image using texture hardware. A linear convolution filter is used to reduce the largest dimension of each 3D texture by a power of 2 (application behaviour is not effected as OpenGL texture coordinates are normalised to a unit cube).

5. Results

A direct volume rendering application was developed to evaluate the performance of the adaptive render system. The application produces a shaded 3D volumetric effect by compositing a set of view plane aligned polygons that are texture mapped through two 3D textures of size (256,256,512). This is an expensive process and results in a frame rate of 10.3fps at a default resolution of 800x800 on an NVIDIA GeForce 7600 256MB desktop graphics card. Figures 6 and 7 shows the result of independently varying resolution and texture size using our scene scale SPU.

Small reductions in resolution size offer little improvement to the frame rate. This is partly due to the overhead involved in rescaling the rendered image to full resolution. However, examination of the texture scale graph shows that the application is initially texture bound, as small reductions in texture size result in an improved frame rate. Once the resolution is reduced further, by over a half, the frame rate begins to improve greatly and continues to improve with reduced resolution. Rather than attempt to model the rendering process, the scene scale module deduces the dominant rendering factor by observing the change in frame rate due to the previous action taken by the module, alternating between reducing texture and resolution size to converge on an optimum configuration. During less active periods, indicated by *CurrentInteractionRate,* the path-switch SPU transmits full resolution images from the server.

If the scene scale SPU cannot obtain a suitable frame rate, the stream switch SPU detects this and switches to source based rendering. At which point the adaptive JPEG compression module begins to converge on a compression quality which matches the available bandwidth (refer to Figure 5).

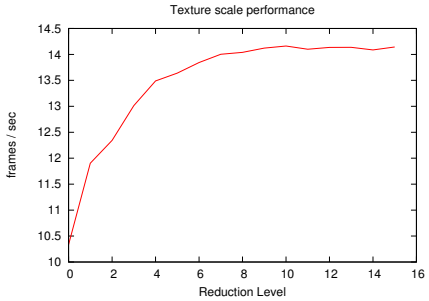

Figure 7: *The effect of texture scaling on frame rate using an NVIDIA GeForce 7600 graphics card. Each reduction level reduces the largest texture dimension by a power of two.*

6. Conclusions and Further Work

Providing quality-of-service for distributed stream rendering represents an information distribution problem: nodes in the network understand local resources, but are unaware of how actions on that stream effect other nodes. We have addressed this problem using a classic feedback control model, and shown how this can be used to deliver adapted graphical scenes for large visualisations.

Our novel use of feedback control loops and quality-of-service messages moves remote rendering beyond a client-server paradigm and towards a distributed streaming paradigm of source and receiver. Specialised modules (providing compression, path-switching and volume rendering reduction) are now able to co-operate to target distributed remote rendering networks to a user needs.

These networks facilitate the connection of multiple clients to a rendering stream, each presented with a tailored environment to both user and resource requirements. Where sufficient client resources are present to render a visualisation scene, local interaction loops can now be used to provide very responsive user interaction. We are currently investigating how these local interaction loops can be applied to our adaptive render system.

References

[AGMR02] ALLARD J., GOURANTON V., MELIN E., RAFFIN B.: Parallelizing pre-rendering computations on a net juggler pc cluster. *Immersive Projection Technology Symposium* (march 2002).

[Bet00] BETHEL W.: Visualization viewpoints: Visualization dot com. *IEEE Computer Graphics and Applications 20*, 3 (2000), 17–20.

[BTL*00] BETHEL W., TIERNEY B., LEE J., GUNTER D., LAU S.: Using high-speed wans and network data caches to enable remote and distributed visualization. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2000), IEEE Computer Society.

[ESE00] ENGEL K., SOMMER O., ERTL T.: A framework for interactive hardware accelerated remote 3d-visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym'00* (2000), pp. 167–177.

[ESEE99] ENGEL K., SOMMER O., ERNST C., ERTL T.: Remote 3D visualization using image-streaming techniques. In *Proceedings of the International Symposium on Intelligent Multimedia and Distance Education* (1999).

[FF99] FLOYD S., FALL K.: Promoting the use of end-to-end congestion control in the internet. *Networking, IEEE/ACM Transactions on 7*, 4 (1999), 458–472.

[HEHE01] HUMPHREYS G., EVERETT M., HANRAHAN P., ELDRIGDE M.: Wiregl: A scalable graphics system for clusters. In *International Conference on Computer Graphics* (2001), pp. 129 – 140.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P., KLOSOWSKI J.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702.

[KOC05] KALAWSKY R., O'BRIEN J., COVENEY P.: Improving scientists' interaction with complex computational–visualization environments based on a distributed grid infrastructure. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences 363*, 1833 (2005), 1867–1884.

[Mar00] MARTIN I. M.: Arte - an adpative rendering and transmission environment for 3d graphics. In *Proceedings of the eighth ACM international conference on Multimedia* (New York, NY, USA, 2000), ACM Press, pp. 413–415.

[MC00] MA K., CAMP D.: High performance visualization of time-varying volume data over a wide-area network. *Proceedings of Supercomputing 2000 Conference* (2000).

[McC88] MCCORMICK B.: Visualization in scientific computing. *ACM SIGBIO Newsletter 10*, 1 (1988), 15–21.

[Oha99] OHAZAMA C.: *OpenGL Vizserver White Paper*. Tech. rep., Silicon Graphics Inc., 1999.

[PBCK05] PURNOMO B., BILODEAU J., COHEN J., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM Press New York, NY, USA, pp. 53–61.

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The volumepro real-time ray-casting system. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), ACM Press/Addison-Wesley Publishing Co., pp. 251–260.

[SB03] SHALF J., BETHEL E. W.: How the grid will affect the architecture of future visualization systems. *IEEE Computer Graphics and Applications 23*, 2 (May/June 2003), 6–9.

[SML98] SCHROEDER W., MARTIN K., LORENSEN B.: *The visualization toolkit*. Prentice Hall PTR Upper Saddle River, NJ, 1998.

[Spi03] SPITZER J.: Opengl performance tuning. *NVIDIA Corporation, GameDevelopers Conference* (2003).

[SWNH03] STAADT O. G., WALKER J., NUBER C., HAMANN B.: A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. *IPT/EGVE 2003.Seventh Immersive Projection Technology Workshop.Ninth Eurographics Workshop on Virtual Environments* (2003), 70,. Conference Paper; 20; C2004-09-6110B-03301; English; ID: INSPEC (EBSCO).

[VBRR02] VOSS G., BEHR J., REINERS D., ROTH M.: A multi-thread safe foundation for scene graphs and its extension to clusters. *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (2002), 33–37.

[vdSRG*02] VAN DER SCHAAF T., RENAMBOT L., GERMANS D., SPOELDER H., BAL H.: Retained mode parallel rendering for scalable tiled displays. *Proc. 6th Ann. Immersive Projection Technology (IPT) Symposium, Orlando, Florida* (2002).

[WHZ*01] WU D., HOU Y., ZHU W., ZHANG Y., PEHA J.: Streaming video over the internet: approaches and directions. *Circuits and Systems for Video Technology, IEEE Transactions on 11*, 3 (2001), 282–300.

[WSND03] WOO M., SHREINER D., NEIDER J., DAVIS T.: *OpenGL Programming Guide: the official guide to learning OpenGL, Version 1.4*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.