OpenGL|D - A Multi-user Single State Architecture for Multiplayer Game Development

Karsten Pedersen
Department of Creative Technology,
Faculty of Science and Technology
Bournemouth University, UK
Email: pedersenk@bournemouth.ac.uk

Wen Tang
Department of Creative Technology,
Faculty of Science and Technology
Bournemouth University, UK
Email: wtang@bournemouth.ac.uk

Christos Gatzidis
Department of Creative Technology,
Faculty of Science and Technology
Bournemouth University, UK
Email: cgatzidis@bournemouth.ac.uk

***Abstract*—Multi-user applications can be complex to develop due to their large or intricate nature. Many of the issues encountered are related to performance and security. These issues are exacerbated when the scale of the application increases. This paper introduces a novel distributed architecture called OpenGL|D (OpenGL Distributed). This technology enables an application to pass through the graphical calls between a Virtual Machine (VM) and the graphics processing unit (GPU) on the native host across a network. This ability allows applications to run inside a virtual machine (VM), whilst still benefiting from hardware accelerated performance from the GPU for the computationally intensive graphical processing. This allows for the development of 3D software requiring no dependencies on specific hardware or technology other than ANSI C and a network stack, demonstrating our approach to platform agnostic development and digital preservation.**

***Index Terms*—Multiplayer Games, Multi-user Applications, Digital Preservation, Portability and Platform Agnostic**

1. Introduction

Online multiplayer games, it can be argued, are one of the most popular entertainment media in recent years. However, the software infrastructure to support these multiplayer games is very large and complex [1]. Issues regarding real-time performance of user interactions and graphics rendering remain challenging even with today's state of the art software technology [2] [3]. Common to multiplayer games are problems associated with server workload latency, scalable communication costs plus real-time localisation and replication of player interaction. Specifically, large-scale games involving tens and thousands of players require a range of solutions to address the problem from design, implementation and evaluation.

The rapid development and evolution of computer architecture often fails to provide an infrastructure in order to ensure that older software can continue to run on recent platforms. These (potentially) standards compliant and well-implemented programs are often still valid for many industry standard applications. Thus, the lack of infrastructure in place to cater for these sometimes mission critical software packages may cause a failure in the uptake of the new platform. Being able to run existing or legacy applications has the benefits of saving costs and reducing the risk of introducing bugs during the development of the replacement software [4]. Currently, VM technology is one of the few ways to do this.

In this paper, we introduce a novel distributed architecture for multiplayer games; OpenGL|D (OpenGL Distributed), which is an evolving attempt at addressing the aforementioned challenging issues. In addition to this, OpenGL|D is also aimed at improving the lifespan of software. In particular, through OpenGL|D, 3D software applications such as Virtual Reality (VR) and Augmented Reality (AR) applications are allowed to be run from inside a virtual machine (VM), whilst still benefiting from hardware accelerated performance from the graphical processing unit (GPU). This is achieved by forwarding out the graphical calls from the virtual environment into a WebGL enabled web browser via websockets.

OpenGL|D can offer more beyond potential success in the area of digital preservation as it can also open up new possibilities for the architecture of multi-user, collaborative tools and gaming software. Of particular interest is the fact that even though the graphics are processed on the GPU of the individual connected client machines, the software itself and the logic contained within is running on a single machine, the server. This means that each client implicitly shares a single application state which completely eliminates the need to synchronise the clients. This not only simplifies the development of multi-user network software, but can also potentially reduce bandwidth [5] [6].

2. Related Work in Client Synchronisation

Existing online multiplayer games utilize a client-server model which not only introduces latency but also a single point of failure to a game. Distributed architectures eliminate these issues but add additional complexity in the synchronisation and robustness of the shared data. The work carried out by Cronin et al [7] introduces an alternative synchronisation mechanism (called Trailing State Synchronisation) which offers a hybrid approach between the traditional client-server model and a distributed approach. It allows clients to share data in a peer to peer manner whilst periodically checking with the central server to confirm their state is correct. Their results appear promising but in the worst case scenario, their system has resulted in multiple inconsistencies and delays due to the rollback mechanism.

We have previously undertaken research work in the similar area of multiplayer synchronisation but with a very different approach to what we propose in this paper [8]. In order to create a protocol which reduces player cheating, we proposed the idea of using a node based approach to lay out shared data in memory. Each of these nodes then had an owner attached and respective permissions. This allowed for a flexible protocol to be built which was potentially trivial to maintain and extend. It also performed efficiently where players could interact with the world and make changes to any object or data they owned whilst also preventing others from modifying unauthorised objects. Thus, this achieves protecting the server and other players from any potential cheating. The technology performed well and, as part of a prototype, was integrated with three existing games an independent games development studio developed for LEGO. The fact that it could easily integrate with existing software, as opposed to software being built from scratch, demonstrated that this approach was very easy to maintain and extend.

However, we discovered a number of complexities with the protocol, described in Section 3, so our solution started to become hard to manage. The node ownership system works well for a number of scenarios but transferring ownership (i.e as part of a trade) still felt overly complex. This very fact is what prompted us to look into new ways to reduce the need to synchronise the state entirely and move towards streaming technologies such as the one we propose in this paper.

3. Complexities Involved in Client Synchronisation

Developing a multi user application is a more complicated and expensive process than single user software [9]. The main reason for this is because there are more entry points for the incorrect handling of data. Since there is effectively more than one unit of execution operating at a time, in a similar way to a multi-threaded application, it opens up the possibilities of race conditions and other time dependent bugs. This can cost time and effort to debug.

With the increasingly complex network interactions seen in games today, including all the underlying data that needs to be synchronised, it soon becomes evident that without an effective design, performing this process for similar events could provide a large number of potential entry points for bugs and synchronisation issues. Whilst it is certainly possible to write a game with a large number of synchronisations taking place, it will require a large amount of care from experienced and disciplined programmers. However, with a technology such as OpenGL|Distributed, all of these steps needed to synchronise client states can be avoided.

4. Inner Workings of OpenGL|D

OpenGL|D implements a client/server architecture where rather than have the running 3D program calling the OpenGL API to communicate with the GPU to rasterize a scene on the local machine, it, instead, creates a server for clients to connect to via a web browser. Once connected, the OpenGL calls are translated to a protocol and back on the client to finally be executed by the WebGL equivalents. Technically this creates a partition in the technology stack which is almost entirely independent from the hardware it is run on. This can be seen in Figure 1. From a technical viewpoint this architecture has the benefit that complexity can be encapsulated.

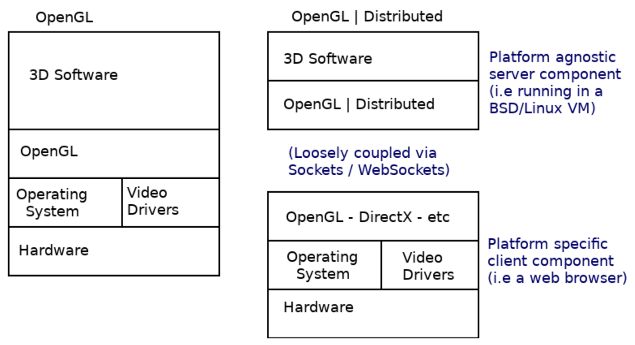

Figure 1. Diagram describing the layers that OpenGL is built upon compared to OpenGL|D. Notice that OpenGL|D has additional layers of abstraction.

From a digital preservation viewpoint, this architecture is useful because the 3D software can be run in a VM running an old operating system as a guest. The host can then run a web browser and simply connect to the server through the virtual machine boundary. However, from a multi-user collaboration viewpoint, the additional benefit is that multiple clients can connect to this server and render out the same scene. This provides the foundation for OpenGL|D's use as a multi-user solution.

4.1. Protocol Overview

The OpenGL|D protocol is fairly straightforward. This is largely due to the fact that it can mimic how the computer and GPU communicate in a largely faithful manner. This also allows for traditional graphics programming optimisations to remain valid. When an OpenGL command is called, the server library encodes the command and data into a smaller message and forwards it onto the client. The client then decodes this message and executes it on the underlying platform, whether that is OpenGL, OpenGL|ES, WebGL or even other graphics APIs such as DirectX. Any necessary response is then sent back to the awaiting server. This is demonstrated in Figure 2.

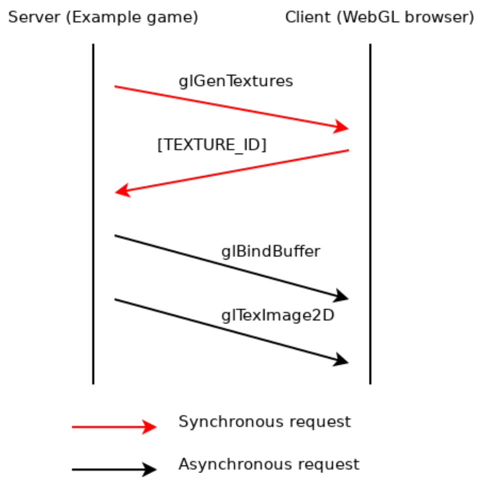

Figure 2. Diagram demonstrating a typical yet simplified communication between the client and server components of OpenGL|D in order to upload a texture.

Due to the fact that OpenGL|D is designed to support a large number of connected clients, it is important that no specific operation blocks or waits for a response. This means that work undertaken to handle a client request must cause minimal delay for the other connected clients. In practice this means that the example given in Figure 2, which demonstrates synchronous requests, utilized a command buffer, storing commands until such a time that the current task is complete and commands can be resumed.

The clients themselves retain almost no state other than the OpenGL|D graphics state such as glEnable(), glEnableClientState() etc. This has the benefit of almost no complexity when syncing a new client. Once vertex buffers and textures are uploaded, the newly connected client is ready for future frames. If a potentially complex action occurs , it happens only in one place, the server. Nothing will need to be synced to the clients to handle this event. They will receive their rendering commands as usual and continue. This behavior was demonstrated in a simple multiplayer football game (Figure 3) where players would knock each other away from the ball whilst applying forces or "grabbing" the ball. Typically, this ownership of the ball and the forces applied upon it would be complex to synchronise between clients but, with OpenGL|D, was not required at all.

4.2. Client Specific Rendering / Cheat Prevention

Other than perhaps some of the more basic collaboration software, it is important that even though clients share the same state with OpenGL|D, it is still possible for them to display different outputs, such as a camera view from another position. This functionality is expressed very naturally with OpenGL|D in that whilst the update function

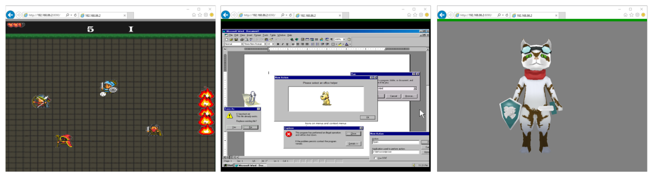

Figure 3. Screenshots of a number of tech demos developed using OpenGL|D. The platforms running these programs included OpenBSD (x86_64) or Plan 9 (ARM / APE Layer) running within VirtualBox. The graphics are then streamed out of the guest VM and into the WebGL enabled Internet Explorer web browser running on the Windows 10 host.

is called just once per frame in OpenGL|D, the display callback is called multiple times for each connected client. This means that during the display function path, it is very easy to query which client ID is the current active one (via gldCurrentClientId()) and then either use the view matrix from its assigned camera to get a unique view port or go down a path of logic that displays the GUI for that client. The whole process could even be described akin to an extension to rendering to a texture, which is a common technique that developers have been using for years. A simple example can be seen in Figure 3 where a player select dialog is shown to a newly connected client without obstructing the view of existing players.

Perhaps one of the more interesting features of using OpenGL|D as a solution for multi-user applications and games is that cheating can be eliminated. The clients themselves are akin to dumb terminals [10] and do no processing themselves. All they do is execute OpenGL commands and respond to key presses or mouse motion commands. This means that any modifications to the client cannot adversely affect the server because all it reads back from the client is a key press.

5. Performance Evaluation

Compared to existing solutions involving manually syncing client state [11] [12], there is virtually no network overhead when using OpenGL|D because, as discussed previously, there is no actual game state to synchronise. However, there certainly is a cost on bandwidth because we are effectively dealing with streaming technology and this means we must send enough data to generate a new image each frame. An additional overhead also needs to be considered when dealing with Websockets (so that the output can be rendered in a web browser). Websockets have a much larger header than standard packets so require more data to be sent across the network. Websockets also do not support UDP technology so TCP is enforced even though, as with other streaming technology, the occasional dropped packet can be easily handled.

That said, compared to other streaming technology such as VNC which deals with rasterized images, OpenGL|D has the potential to be a much faster solution because it uses an intelligent protocol which sends the commands which can generate the output image on the destination hardware rather than send over a pre-rendered image each frame. This can be seen in Figure 4. If there are few models in the scene much less data needs to be transferred through to the client whereas with VNC a map of the rasterized pixels is sent regardless. The bandwidth requirements when using OpenGL|D only start to match that of VNC when dealing with a large number of shapes (almost 10K). This is rarely the case in games due to optimization techniques used to reduce the number of draw calls.

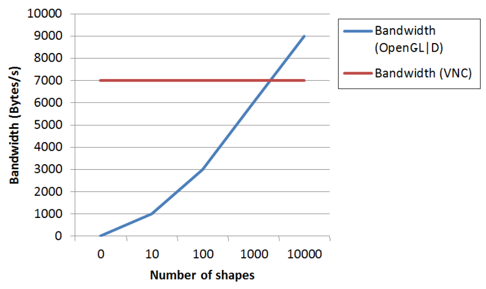

Figure 4. Graph comparing the bandwidth requirements between OpenGL|D and VNC with a varying number of objects in the scene.

In general, network synchronisation via OpenGL|D will have the best performance compared to other solutions when only dealing with a small number of OpenGL draw calls and a large complex game state. Such examples could potentially include software with complex inventory systems that need to be interacted with via simple GUI systems in the client. It will also perform better than most rasterized streaming solutions at higher resolutions. OpenGL|D does not need to send through each pixel to the client, the clients do the actual rasterization, therefore there is no additional costs to bandwidth using OpenGL|D at higher resolutions. This is demonstrated in Figure 5.

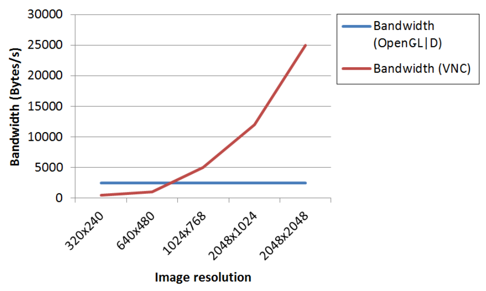

Figure 5. Graph comparing the bandwidth requirements between OpenGL|D and VNC with an increasing image resolution.

Network synchronisation via OpenGL|D will compare worse against other solutions when dealing with simple states to share (such as just synchronising projectiles and player positions) or large complex game worlds with many objects to render. Such examples could include real-time strategy (RTS) games or open world shooters.

6. Conclusion

The process of developing a multi-user project can be greatly simplified by using OpenGL|D. Not only is the developer released from the error-prone task of manually synchronising objects within the game but also new development architectures are made available. Rather than build up hierarchies of objects in a manner ready to be serialized and shared, the development process can invest a greater focus on the logic to carry out tasks in a natural manner. A reduced number of callbacks and rules needs to be applied because the logic is effectively developed in exactly the same way as a single user experience.

References

[1] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers, "A novel approach to the detection of cheating in multiplayer online games," in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, July 2007, pp. 97–106.

[2] D. Wu, Z. Xue, and J. He, "icloudaccess: Cost-effective streaming of video games from the cloud with low latency," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 8, pp. 1405–1416, Aug 2014.

[3] T. Karachristos, D. Apostolatos, and D. Metafas, "A real-time streaming games-on-demand system," in *Proceedings of the 3rd International Conference on Digital Interactive Media in Entertainment and Arts*, ser. DIMEA '08. New York, NY, USA: ACM, 2008, pp. 51–56. [Online]. Available: http://doi.acm.org/10.1145/1413634.1413648

[4] K. Bassin and P. Santhanam, "Managing the maintenance of ported, outsourced, and legacy software via orthogonal defect classification," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 726–734.

[5] J. D. Pellegrino and C. Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," in *Proceedings of the 2nd workshop on Network and system support for games*. ACM, 2003, pp. 52–59.

[6] A. I. Wang, M. Jarrett, and E. Sorteberg, "Experiences from implementing a mobile multiplayer real-time game for wireless networks with high latency," *Int. J. Comput. Games Technol.*, vol. 2009, pp. 6:1–6:14, Jan. 2009.

[7] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin, "An efficient synchronization mechanism for mirrored game architectures," in *Proceedings of the 1st workshop on Network and system support for games*. ACM, 2002, pp. 67–73.

[8] K. Pedersen, C. Gatzidis, and B. Northern, "Distributed deepthought: Synchronising complex network multi-player games in a scalable and flexible manner," in *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, ser. GAS '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 40–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=2662593.2662601

[9] S. R. James and B. D. Gillam, "Network multiplayer game," Oct. 12 1999, uS Patent 5,964,660.

[10] D. C. Bulterman and R. Van Liere, "Multimedia synchronization and unix," in *International Workshop on Network and Operating System Support for Digital Audio and Video*. Springer, 1991, pp. 105–119.

[11] J. Smed, T. Kaukoranta, and H. Hakonen, "A review on networking and multiplayer computer games," *Turku Centre for Computer Science*, 2002.

[12] J. Smed, T. Kaukoranta, and H. Hakonen, "Aspects of networking in multiplayer computer games," *The Electronic Library*, vol. 20, no. 2, pp. 87–97, 2002.