OpenGL|D - An Alternative Approach to Multi-user Architecture

Karsten Pedersen, Christos Gatzidis, and Wen Tang

Department of Creative Technology, Faculty of Science and Technology, Bournemouth University, UK

Abstract. Synchronising state between multiple connected clients can be a challenging task. However, the need to carry this out is becoming much greater as a larger number of software packages are becoming collaborative across a network. Online multiplayer games in particular are already extremely popular but the synchronisation methods and architecture have largely remained the same. OpenGL|Distributed, presented here, aims to provide not only an alternative to this architecture allowing for a greatly simplified development pipeline, but also the opportunity for a number of additional features and design patterns. The architecture provided by OpenGL|D is such that no state information needs to be transferred between clients. Instead, the OpenGL API has been utilised as a platform agnostic protocol. This means that graphical calls can be streamed to each client rather than relying on for manual synchronisation of application domain specific data. Initial test results are discussed, including performance evaluation using data from a number of small prototypes developed within a constrained 48-hour timeframe. These results are compared and evaluated against a more traditional approach to network multiplayer by id Software's QuakeWorld client. It should be noted that this article is an extended version of the work we published in the proceedings of the Cyberworlds 2017 conference[1].

1 Introduction

One could argue that online multiplayer games are amongst the most popular entertainment media of the last few years. However, the software infrastructure to support these multiplayer games is very large and complex [2]. Issues regarding real-time performance of user interactions and graphics rendering remain challenging, even with today's state of the art software technology [3] [4]. 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 and implementation to evaluation.

The most popular contemporary game engines such as Unreal Engine 4 [5] and Unity [6] are employing the centralized client/server architecture. Whilst providing efficient state updates via players sending control messages to a central server, multiplayer games developed using this approach present some inherited problems in terms of robustness and scalability. With the increasing complexity of contemporary multiplayer games, the client-server architecture can potentially become a computation and communication bottleneck.

Further to the scalability issue, the centralized design enforces the game developers to rely on infrastructures provided by game engine manufacturers, which can prevent software preservation and reusability, an important topic that has been overlooked until now [7].

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. For example, the recent developments in Windows 10 S [8], to allow only Microsoft Store apps from being run, effectively fail to cater for many previous standard, well-implemented Win32 programs, which are 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 a 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 [9].

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 the graphical calls from the virtual environment into a WebGL enabled web browser via websockets.

VMs today can be seen as one of the few solutions to running old software without needing to port it to a modern platform, and together with OpenGL|D, older 3D software can be guaranteed to run because of their use.

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 [10] [11].

Our contributions in this paper can be summarised to the following:

– A new architecture design and implementation of medium-scale multiplayer games, VR and AR applications.
– A framework to allow games to be developed in a simple intuitive manner without needing to consider the complexity of multiplayer system design
– A platform agnostic approach allowing multiplayer software to be written and executed on any computer platform
– An innovative re-implementation of one of the industry standard graphics APIs, OpenGL, allowing a drop-in replacement to help integration with existing projects

In the following sections we first describe the existing solutions to the synchronisation of multiplayer games. We then provide further details and examples of the complexities involved in client side synchronisation, which illustrate a number of scenarios that developers will be faced with during the development process of multi-user/multiplayer software, and, subsequently, how a new approach will improve upon the ways of traditional architectures. We then describe the design of OpenGL|D in Section 4. Performance evaluation is presented in Section 5. Finally, we point out some future developments for the extended use of OpenGL|D in both research and development projects.

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 [12] 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. The results in this work appear promising but, in the worst case scenario, this system can result in multiple inconsistencies and delays due to the rollback mechanism.

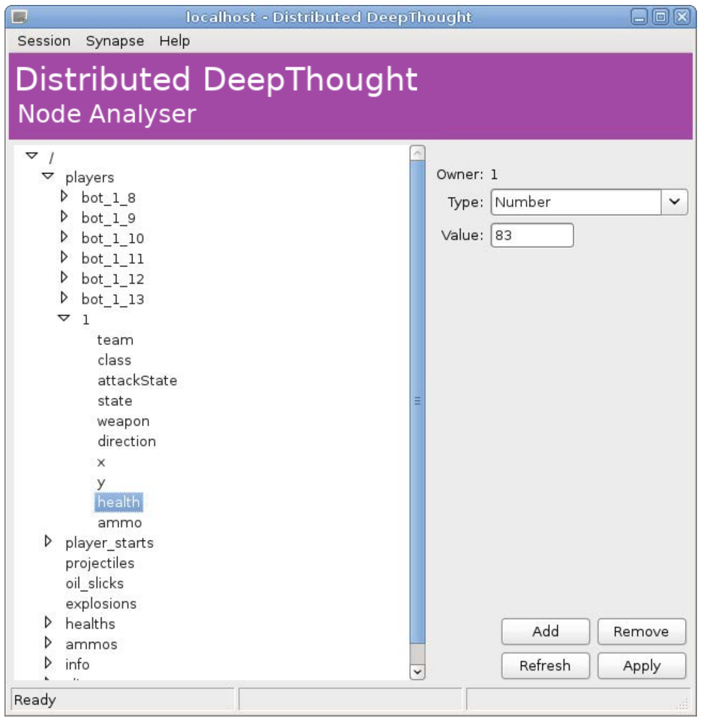
Inconsistencies can manifest into flaws which can be exploited by users to create cheats for a game. By reducing the client side inconsistencies, these flaws can be prevented. However, maintaining consistency also means the restriction on the amount of data that a client can input into the game world or, at the very least, a hybrid design introducing a complex and inflexible protocol for game programmers to work around. Baughman et al [13] proposed a protocol for multiplayer game communication that has anti-cheating guarantees. One particular module of the proposed Lockstep protocol works as a transaction based system which has the guarantee that no host ever receives the state of another host before the game rules permit. This work then improves upon this relatively expensive new protocol with the author's faster Asynchronous Synchronization protocol which relaxes the requirements of Lockstep by decentralizing the game clock. The results have suggested that cheating is effectively eliminated whilst also maintaining a good performance. However, in the examples demonstrated, integrating this technology into a project is non trivial and significant expertise will almost certainly be required.

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 [14]. In order to create a protocol which reduces 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 of 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 [15]. 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.



Owner: 1
Type: Number
Value: 83

Fig. 1. A small internal tool which allowed for the debugging of the Distributed DeepThought node based hierarchy. This tool was invaluable in simulating and testing any potential damage that a malicious user could make.

In a game scenario, for example, if a client opens up a door in the game world, the following steps need to follow:

1. The client notifies the server that they are attempting to open the door
2. The server decides whether they have the correct authorisation to do so
3. The server tells the client that the door is opened
4. The server then notifies all other clients that the door is open
5. The clients change the state in their copy of the game state so that the door is now open

With the increasingly complex network interactions evident in games today, including all the underlying data that need to be synchronised, it soon becomes evident that without an effective design, performing this process for similar events would quickly become unwieldy. This stands true especially if we now add the additional requirement that a new client is connecting and needs to be synchronised to the existing state on the server. The following steps would then be necessary:

1. A client connects to the server and requests a state synchronisation

2. The server needs to scan through its copy of the game state and serialize all the changeable states into a data stream and send to the client
3. The client receives this stream and processes it, updating and adding to its state as necessary
4. The server notifies all other clients that a new client has joined
5. Existing clients update their game state to include this new client

This synchronisation of data, depending on the size of the game world, could become very large and, without a good design, could potentially cause latency issues on other clients whilst the new client is being handled.

The next level of complexity is how clients interact with one another directly. For example, let us assume a scenario where they need to perform a trade of virtual items. Then, the following steps would need to be performed:

1. Client one informs the server they are trading an item with a specified ID with a client of specified ID.
2. Client two informs the server they are trading their item with specified ID with a client of specified ID.
3. Server matches the IDs to create an idea of a trade instance.
4. Server checks that both items are valid and there is no cheating such as memory editing happening (see Section 4.4 for more details)
5. Server accepts the trade and sends success to each client
6. Each client now removes their traded item and creates a new object representing the item they received

The entire process provides a large number of potential entry points for bugs and synchronisation issues in the above scenarios. For example, let us assume that one of the clients disconnects at around step 4. Scanning the state and fixing failed trade instances could be one possible solution but this alone is a complex task. A suitably complex server could have many of these processes for a wide range of functionality which will all need care whilst implementing. Whilst this can certainly yield an acceptable and secure system, as seen in successful commercial games such as Quake 3, it still requires very experienced and disciplined programming [16]. However, the idea is that 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 having 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 to 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 runs on. This can be seen in Figure 2. From a technical viewpoint this architecture has the benefit that complexity can be encapsulated. For example, results from memory checking tools such as Valgrind[17] can often be affected from details of the lower level layers of an operating system. With OpenGL|D, the boundary is limited to data being sent through a socket and, as such, the complex workings of the graphics driver stack can have no influence on the memory allocated by the program being tested.

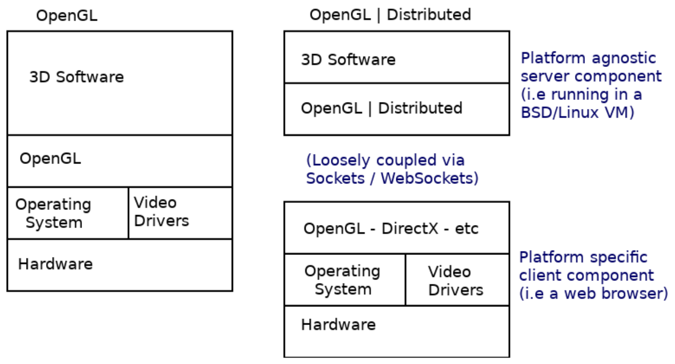


Fig. 2. Diagram describing the layers that OpenGL is built upon compared to OpenGL|D. Notice that OpenGL|D has additional layers of abstraction.

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's CPU 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 3.

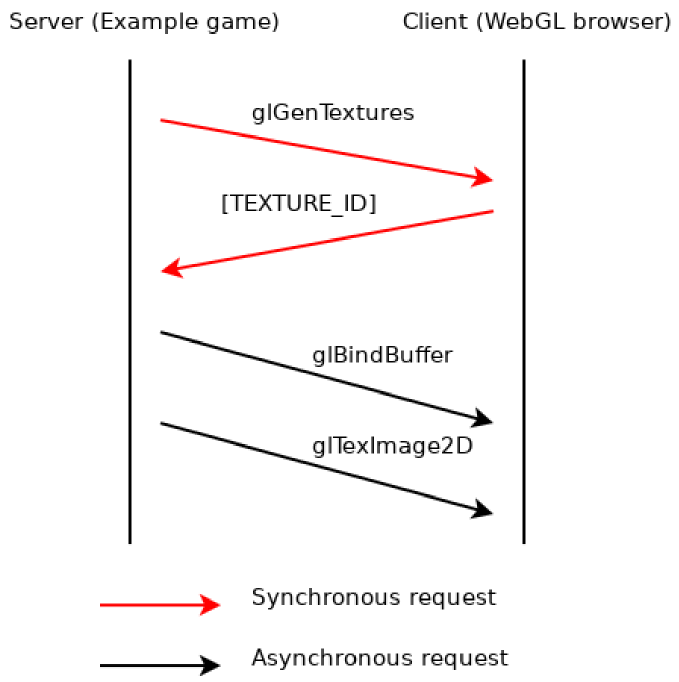


Fig. 3. Diagram demonstrating a typical yet simplified communication between the client and server components of OpenGL|D in order to upload a texture.

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 execution of the server whilst waiting 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 3, which demonstrates synchronous requests, utilizes the OpenGL|D request buffer so that every message for that client after the required synchronous request is stored in a buffer, rather than executed until the dependent request is complete. It then processes the existing request buffer until it is empty or until another synchronous request is required. This works largely well and threads can be avoided which aids with portability. However, this architecture does increase memory usage. This may not be an issue when streaming graphics via OpenGL|D on a server but on a low-powered mobile device this becomes much more important if needing to stream to a large number of clients.

One important example of not blocking communication between clients is when the server handles a new client connection. The new client is updated with a snapshot of the entire current OpenGL state. Even though the state driven architecture of OpenGL works well here, there is still potentially a considerable amount of data to be sent, including textures, buffer objects, etc. However, a similar system to the one described previously is utilized. Whilst the client is being synchronised, new messages are stored in a buffer and processed when ready, whereas other clients remain unaffected (unless we run into bandwidth limitations). See Section 5.4 for an overview of planned optimization techniques.

4.2 How Clients Share a Single State

As described in the previous section, clients connect to a server and simply receive rendering commands whilst sending back key presses or mouse motion events. This means that 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 (as described earlier in the paper), such as opening a door or a trade, 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 4) where players would knock each other away from the ball whilst applying forces or "grabbing" the ball. Typically, this ownership of the ball would be complex to synchronise between clients but, with OpenGL|D, this was not required at all. Applying forces between players can also be complex due to position snapshots often lagging behind in traditional synchronisation approaches. Again, with OpenGL|D, this complexity could be avoided.

4.3 Unique Client Specific Rendering

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. For example, in a 3D game, the clients would likely require a view of the game world from different camera angles, have different information on their heads up display (HUD) and perhaps even have GUI elements displayed just for them. This functionality is expressed very naturally with OpenGL|D in that whilst the update function 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 5, where a player selection dialog is shown to a newly connected client without obstructing the view of existing players.

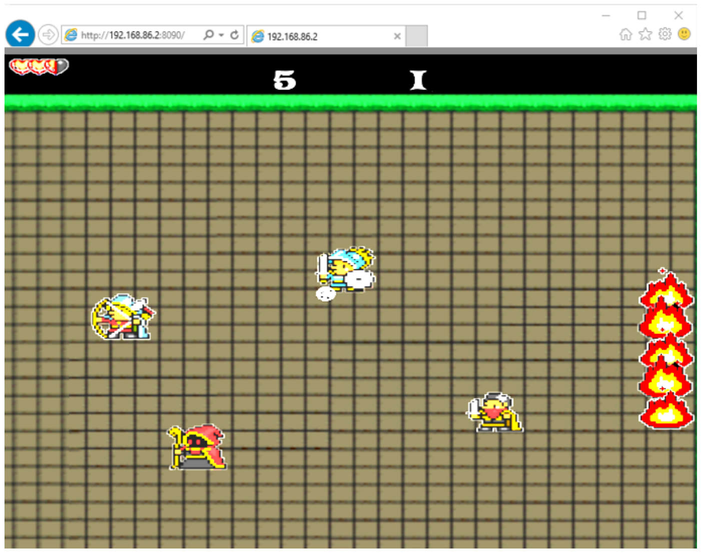


Fig. 4. Screenshot of Fantasy Football; a basic prototype multi-player football game demonstrating fast action and complex interaction between knocking other players away and "ownership" of the ball.

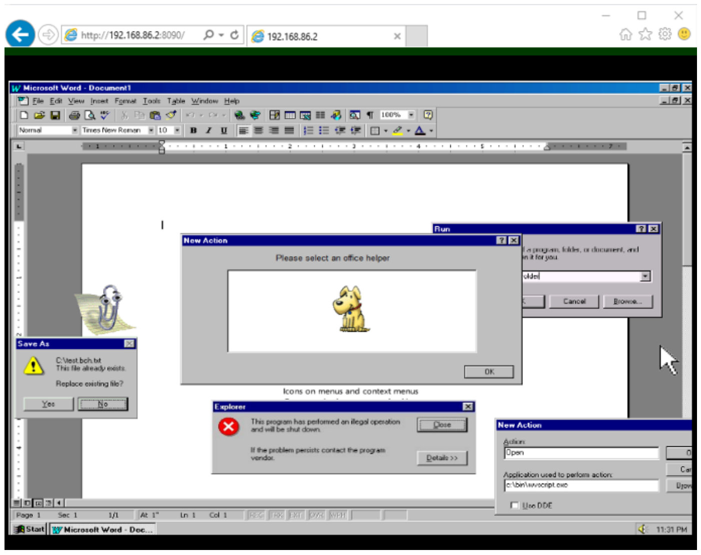
4.4 Cheat Prevention

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 [18] and do no processing themselves. All they do is executing OpenGL commands and responding 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. The types of cheats this avoids include memory editors which can, among other things, freeze memory locations so data such as health cannot be decremented when a player is hurt. Other cheats involve the modification of the client and, if dealing with native C/C++ programs, entire functions dealing with player health could be patched out and replaced with null operations (NOPS) to, again, avoid the decreasing of values such as health. This is even more likely if a client is written in an interpreted language (such as Javascript) or JIT bytecode (i.e. JVM or .NET) since even if this is obfuscated, it is still relatively easy to patch or completely decompile these programs compared to native machine code.

5 Results and Discussion

5.1 Performance Evaluation

Compared to existing solutions involving manually syncing the client state [19] [20], there is virtually no network overhead when using OpenGL|D because, as dis-


http://192.168.86.2:8090/
192.168.86.2
Microsoft Word - Document1
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
10
100%
Run
Cancel
Browse...
New Action
OK
Save As
Yes
No
Icons on menus and context menus
Explorer
Close
Details >>
Action:
Open
Application used to perform action:
c:\bin\wvscript.exe
Use DDE
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1
Start
Microsoft Word - Doc...
11:31 PM

Please select an office helper

C:\test.bch.txt
This file already exists.
Replace existing file?

This program has performed an illegal operation and will be shut down.
If the problem persists contact the program vendor.

Fig. 5. Screenshot of Cloud Office 95; a basic prototype multi-player game developed during a games jam. One player has the character select menu open whereas it is hidden for other clients, demonstrating client specific rendering paths.

cussed 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 6. 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.

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 in-

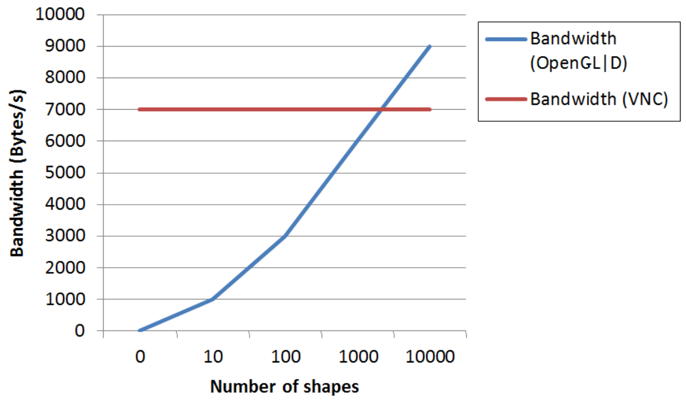


Fig. 6. Graph comparing the bandwidth requirements between OpenGL|D and VNC with a varying number of objects in the scene.

teracted 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 7.

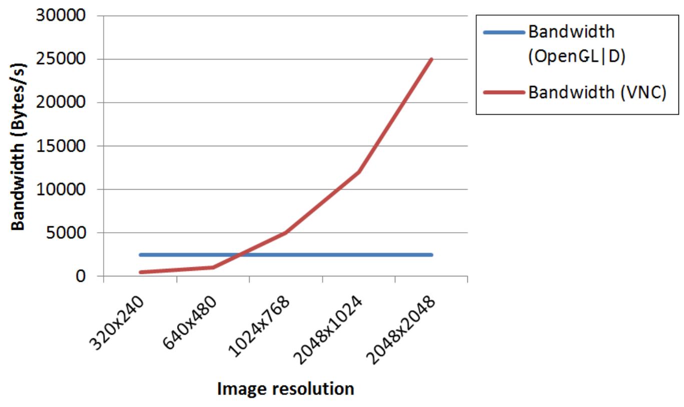


Fig. 7. Graph comparing the bandwidth requirements between OpenGL|D and VNC with an increasing image resolution.

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.

5.2 Bandwidth Comparison with QuakeWorld

Whilst id's Quake is now regarded as a fairly antiquated game and certainly no longer cutting edge in any way, there have been a large number of improvements to its codebase (largely due to its open-source nature) such as FTE QuakeWorld which is still in active development[21]. The QuakeWorld client in particular still, int our view, provides an adequate test bed for comparison with OpenGL|D. What makes the QuakeWorld client very convenient to test against is that it employs an older implementation of OpenGL and so is fairly straightforward to port to using OpenGL|D. Whilst OpenGL|D does not yet provide a full covarage of an OpenGL API, the majority of functionality is there and, most importantly, the data is still being sent across the network so will still provide valid (albeit early) test results. One important limitation is that in our implementation, the different clients only see the same image rather than a unique image from their player's viewport. However, given the way that OpenGL|D works, this was deemed satisfactory and would not alter results in any way. Further work is certainly planned in this area.

Our initial tests agree with the work carried out by Cordeiro et al[22] and Abdelkhalek et al[23][24] and show that the QuakeWorld client has a generally low bandwidth requirement of 2-3KB for both incoming and outgoing traffic. This is with the official maximum of 32 players. However, this number does occasionally spike when an interesting event happens, such as a player death or teleportation. This suggests that the additional synchronisation messages required for such an event are in place and sent through the network so that the clients can keep up to date with the world state. In OpenGL|D it was predicted that these spikes would never exist. In the tests performed, whilst our prediction remained true, the base bandwidth required was consistently higher at around 6-7KB. Again, this points to OpenGL|D's scalability being most effective in intricate and complex state updates rather than synchronising a large number of clients.

To demonstrate this view, a small modification (written in QuakeC) was made to the client to artificially produce a need for a large number of state changes. What this modification provided was the creation of a constant trade based system so that on each frame a virtual item was passed around the clients until one of the clients matched a set criteria. The additional bandwidth required for the locking and synchronisation involved in the trade of these items did start to increase. If around 50 of these trades happened at once, the bandwidth required matched that of the OpenGL|D client, whereas with the same trade mechanism, the OpenGL|D client showed no increase in bandwidth. Although rather artificial in nature, this very basic experiment demonstrates that for certain tasks, the synchronisation system provided by OpenGL|D can potentially scale in a more favorable way compared to traditional approaches.

5.3 Network Protocol Optimization Mirrors GPU

The overhead discussed previously can be greatly reduced using a variety of techniques. Most of these are techniques that are also evident in standard OpenGL software. In general, reducing the amount of data being sent to and from the graphics card translates almost exactly to reducing the amount of data being sent to and from the client and the server. A basic example is reducing the number of draw calls by batching mesh data together into vertex buffer objects (VBOs) and vertex array objects (VAOs). Grouping mesh data together based on material and texture can also avoid the need for binding a texture sampler between each mesh or changing light data.

Generally, once mesh and texture data has been uploaded to the client and the client state has been prepared, the only calls that need to be made are updating the model view matrix and initiating the drawing of a number of triangles (via glDrawArrays()). This means that much less data is sent through the network compared to other streaming technologies such as VNC. This is almost comparable to the manual synchronisation system found in existing games (yet retaining all the benefits of having a single program state).

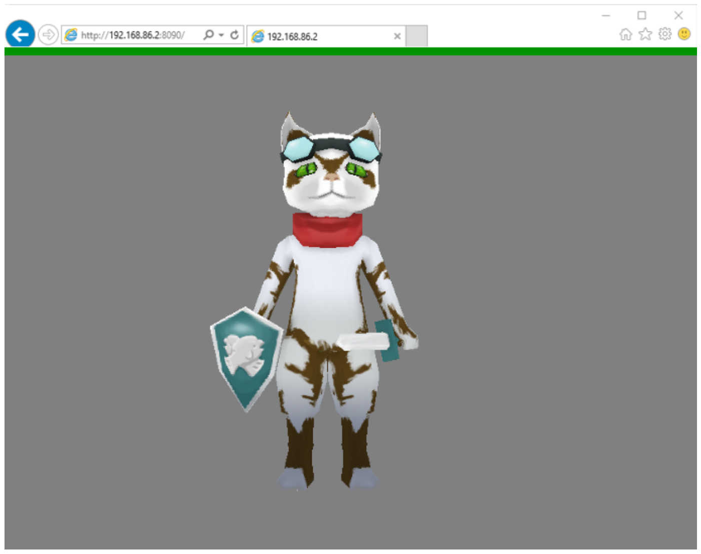


Fig. 8. Screenshot of an example OpenGL|D output. An application with this rotating 3D model takes less than 10 bytes each frame. Even with maximum compression, VNC takes over 20 times that in bandwidth for a similar (but lower quality, lossy compressed) image.

5.4 Planned Optimisations

There are a number of optimization techniques we plan to introduce to OpenGL|D as and when required. The first priority is likely to be in the initial synchronisation of the OpenGL state. In preliminary tests on mobile devices, we deemed it too expensive to compress every message before it is sent. However, as with most streaming technology the payload size sent for each frame is quite small anyway thus the effectiveness of most compression schemes is greatly reduced for this task. However, since the initial synchronisation of the client is likely to be much larger and can be delivered as a contiguous block of data, compression is likely to yield more positive results, making this a worthwhile avenue to explore for decreasing the initial load time.

The next priority is likely to lie in the sending of buffer objects and textures. Not only can these large blocks of data be compressed, but in the case of 2D textures, lossless encoding (such as with PNG) should produce even better results. Research into 3D image compression schemes will need to be undertaken in case sending an array of PNG images is suboptimal.

Finally, we realize that UDP is likely to be a more optimal solution than our existing TCP based system [25] and in the main draw routines, unreliable packet transmission can be handled effectively therefore, this faster but less reliable protocol is a feasible optimization. Only in transferring permanent state changes or uploading data objects is reliable transmission (either via TCP or a reliable UDP scheme) desired. However, our current priority is to support the transmission of data to a HTML5 web browser via WebSockets, which not only produce a larger overhead to raw sockets but also restrict our protocol to TCP based technology. In the future, if the web browser environment proves to be too volatile or too restrictive, a standalone OpenGL|D viewer is planned where the use of UDP can be explored in a more thorough fashion.

6 Summary of Discussion

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 now 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. Arguably, this new flexibility in design also allows for greater support for logical distribution on clusters. This is because without needing to focus on the synchronisation of hierarchies between computers, this additional time can be spent solving the problems provided by traditional clustering complexities. The task of comparing OpenGL|D against VNC has been valuable. This is because in terms of portability, other solutions such as NoMachine's NX server[26] or GameStream[27], NVIDIA's commercial streaming technology, are not available on all but the most common platforms. This would greatly limit their ability to be used to facilitate digital preservation and perform on older platforms such as DOS or Plan 9 and newer platforms such as Tizen or Jolla/Sailfish. All of these platforms are supported by OpenGL|D and VNC however.

7 Conclusion and Future Work

Allowing users to share a single state provides some interesting avenues for analytical data. For example, most software will record an event when a specific action occurs. This happens in isolation from other users. However, if the same state is shared, it should be possible to obtain analytics data for choices the users have made at that exact second alongside one another. We can then compare the choices made knowing that all users have experienced exactly the same stimulus, distractions and context at the time the event triggered. This should ensure a more robust correlation between analytical results.

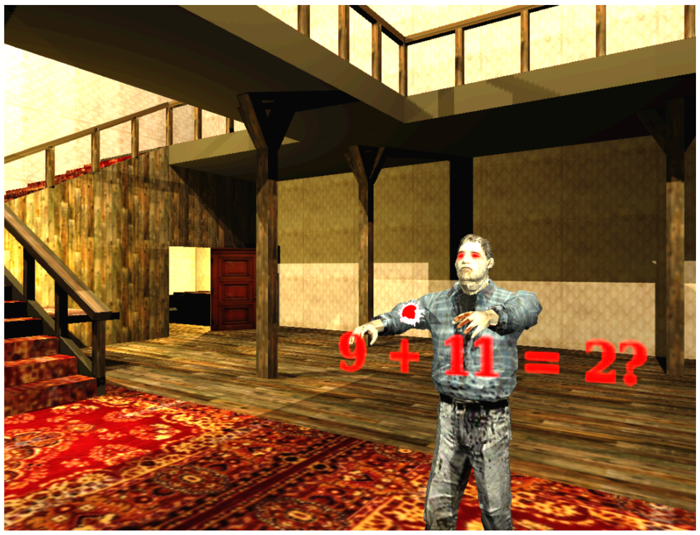


Fig. 9. Screenshot of the Zombie Maths Game. This early prototype has been trialled by a large number of participants and the results, which have been uploaded to our internal analytics server platform, have shown promise in terms of engagement and improvement for all age ranges.

A future multi-user project involving OpenGL|D is a game to help learn and practice maths (Figure 9). It is modeled after traditional light gun games such as House of the Dead or Time Crisis. Instead of aiming and pulling a trigger, a series of correct answers from the players will clear the enemies. With this in place, event data such as a user encountering a certain task and subsequently interacting and performing with it can be obtained and compared with the other players logged in at that time, dynamically changing the rules of the game. The main aim of this game is to encourage users to practice their maths by allowing them to play together and which will result in repeat plays and thus hopefully increase the lifespan of the game itself. From a technical viewpoint, synchronising the enemies with maths questions on each client would potentially be non-trivial, however, OpenGL|D is very likely to simplify this process by virtue of each client implicitly sharing the same game state.

References

1. K. Pedersen, C. gatzidis, and W. Tang, in *OpenGL|D - A Multi-user Single State Architecture for Multiplayer Game Development. International Conference on Cyberworlds 2017. Chester, UK*, Sep 2017.
2. 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.
3. 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.
4. 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
5. J. Färber, “Traffic modelling for fast action network games,” *Multimedia Tools and Applications*, vol. 23, no. 1, pp. 31–46, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:MTAP.0000026840.45588.64
6. A. R. Stagner, *Unity multiplayer games.* Packt Publishing Ltd, 2013.
7. B. Matthews, A. Shaon, J. Bicarregui, and C. Jones, “A framework for software preservation,” *International Journal of Digital Curation*, vol. 5, no. 1, pp. 91–105, 2010.
8. Microsoft, “Introducing windows 10 s,” https://www.microsoft.com/en-us/windows/windows-10-s, 2017, [Online; accessed 20-January-2017].
9. 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.
10. 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.
11. 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.
12. 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.
13. N. E. Baughman and B. N. Levine, “Cheat-proof playout for centralized and distributed online games,” in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 2001, pp. 104–113.
14. 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
15. S. R. James and B. D. Gillam, “Network multiplayer game,” Oct. 12 1999, uS Patent 5,964,660.

16. F. Sanglard, "Fabien sanglard's website," http://fabiensanglard.net/quake3/network.php, 2012, [Online; accessed 20-January-2017].
17. V. Developers, "Valgrind memory debugger," http://valgrind.org, 2017, [Online; accessed 20-January-2017].
18. 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.
19. J. Smed, T. Kaukoranta, and H. Hakonen, "A review on networking and multiplayer computer games," *Turku Centre for Computer Science*, 2002.
20. 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.
21. id Software, "Fte quake world," https://sourceforge.net/p/fteqw/code/HEAD/tree/trunk, 2017, [Online; accessed 20-October-2017].
22. D. Cordeiro, A. Goldman, and D. da Silva, in *Euro-Par 2007 Parallel Processing, 13th International Euro-Par Conference, Rennes, France, 2007.* Springer, 2007.
23. A. Abdelkhalek, A. Bilas, and A. Moshovos, "Behavior and performance of interactive multi-player game servers," *Cluster Computing*, vol. 6, no. 4, pp. 355–366, Oct 2003. [Online]. Available: https://doi.org/10.1023/A:1025718026938
24. A. Abdelkhalek and A. Bilas, "Parallelization and performance of interactive multiplayer game servers," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 72–.
25. G. Xylomenos and G. C. Polyzos, "Tcp and udp performance over a wireless lan," in *INFOCOM '99. Eighteenth Annual Join Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, Mar 1999, pp. 439–446 vol.2.
26. NoMachine, "Nomachine nx server," http://www.nomachine.com, 2017, [Online; accessed 20-January-2017].
27. NVIDIA, "Nvidia gamestream: Play pc games on nvidia shield," http://www.nvidia.co.uk/shield/games/gamestream, 2017, [Online; accessed 20-January-2017].