Hydra: Loosely Coupling the Graphics Pipeline to Facilitate Digital Preservation

Karsten Pedersen

Department of Creative Technology

September 13, 2021

Abstract

It can be argued that software can be seen as a form of art and digital heritage and yet it rarely enjoys the same efforts afforded to it compared to physical counterparts. There are many reasons for this, such as the increasing costs of maintenance or the reducing amount of expertise in the specific aging technology. Maintaining software and ensuring that it continues to work on current hardware and operating systems is known as digital preservation.

There are many ways in which we can attempt to preserve digital software and one of the most effective ones is by using emulation to simulate the obsolete hardware. However, for games and other entertainment media, this technique is not always effective due to a requirement on specific hardware, such as an accelerated GPU in order to reach an acceptable performance for the user. It is often difficult to emulate a GPU and, as such, a different approach often needs to be taken, which reduces the flexibility and portability of the emulation software.

Hydra is a new approach to accessing the native hardware from within an emulated environment which allows for a much simpler emulator to be developed and maintained and yet also offers the potential of accessing other types of hardware without needing to modify the emulation software itself. Hydra is designed to be platform agnostic in that not only is it possible to integrate with existing emulators but also be immediately usable from within guest operating systems, ranging from legacy platforms such as MS-DOS, through to modern platforms such as the PlayStation 4 (Orbis OS, a FreeBSD derivative), through to more exotic platforms such as Plan 9 from Bell Laboratories. It can do this because it does not rely on a complex emulator-specific virtual driver stack. This PhD thesis provides the research undertaken for Hydra, including the motivation behind it, the specific problems it was designed to solve and how it can be implemented in a platform agnostic manner.

Hydra's performance is analysed to ascertain the suitability of the output to cater for, specifically, a wide variety of platforms that it can run on in a satisfactory manner within less powerful or emulated environments. A performance analysis study is conducted to ensure that the technology provides an acceptable solution to accessing preserved titles. This study concluded with results showing that Hydra offers a greater performance than software rendering, especially within emulated environments. A bandwidth comparison between Hydra and VNC was undertaken to ascertain the use of the technology as a streaming medium. The results concluded that under specific conditions, Hydra performed better than VNC by streaming at a higher resolution and consuming less bandwidth. Hydra is also utilised in a number of engineering tasks relating to preservation of software. The experiences of using Hydra in this way are discussed, including any difficulties encountered. Lastly, a conclusion is made and any future work is identified.

Acknowledgements

I would like to thank everyone who helped me through the journey of completing my PhD. In particular I would like to thank Professor Christos Gatzidis for his role as my primary supervisor. His support and guidance throughout has been extremely helpful and encouraging. Secondly I would like to thank my secondary supervisor, Professor Wen Tang who I have learned a lot from both academically and professionally. I would also like to thank all of my collegues at Bournemouth University for offering their help and sharing their experience over the years. I would like to thank my family and especially my partner, Shelly for her interest, care and patience when I would tell her "just another month to go!" over the last 8 years.

Contents

1 Introduction **19**

1.1 Digital Preservation 19

1.2 The Importance of Digital Preservation 20

1.3 Difficulties in Preserving Software 23

1.4 Aims of this Research 27

1.5 Outcomes of this Research 28

1.6 Overview of this Research 30

2 Literature Review **33**

2.1 The State of Digital Preservation 33

2.2 The Roles of Emulation and Virtualisation in Digital Preservation . . 34

2.3 Emulation Performance 38

2.4 Binary Translation 40

2.5 Emulating the GPU 43

2.6 GPU Passthrough 43

2.7 Remote Virtual Graphics Systems and Streaming 45

2.8 Ethical Considerations in Digital Preservation 49

2.9 Cloud Solutions to Digital Preservation 51

2.10 The Digital Preservation of Computer Games 54

2.11 Evaluation 56

2.12 Additional Techniques to Preserve Software 57

2.12.1 API Cloning 58
2.12.2 Compatibility Layer 62
2.12.3 Comparison of the Techniques 64
2.13 Conclusion 73

3 Methodology - GPU Passthrough with Hydra 76

3.1 Introduction to Hydra 76
3.1.1 What is Hydra? 76
3.1.2 Reduction of Dependencies 78
3.1.3 Separation of Application and Hardware 83
3.1.4 The Illusion of Programming Languages 84
3.1.5 Platform Identification 97
3.2 Architecture 102
3.2.1 API Cloning 103
3.2.2 State Management 105
3.2.3 Multi-user State Management 107
3.2.4 Overcoming Issues at a Protocol Level 109
3.2.5 Avoiding Memory Errors 112
3.2.6 Testing Different Memory Strategies 126
3.2.7 Technical Details Behind libstent 133
3.2.8 Object-orientation in C 140
3.2.9 WebSocket Implementation 143
3.3 Summary 148

4 An Alternative Approach to Multi-user Architecture 150

4.1 Introduction 150
4.2 Related Work in Client Synchronisation 153
4.3 Complexities Involved in Client Synchronisation 155
4.4 Inner Workings of Hydra 157

4.4.1 Protocol Overview 159
4.4.2 How Clients Share a Single State 160
4.4.3 Unique Client Specific Rendering 161
4.4.4 Cheat Prevention 163

5 Evaluating the use of Hydra in Multi-user Software 164
5.1 Streaming Comparison Against VNC 164
5.2 Bandwidth Comparison with QuakeWorld 166
5.3 Network Protocol Optimisation Mirrors GPU 168
5.4 Planned Optimisations 169
5.5 Conclusion 170
5.6 Future Work in Multi-user Synchronisation 171
5.7 Summary 172

6 Evaluating the Effectiveness of Hydra 174
6.1 Performance Against Existing Techniques 174
6.1.1 Experiment Design 175
6.1.2 Analysis of Results 181
6.1.3 Conclusion 188
6.2 Interactions with Virtual Machines 189
6.3 Summary 195

7 Conclusion and Future Research 197
7.1 Review of the Goals of this Research 197
7.2 Summary of Research 198
7.3 Conclusion 199
7.4 Future Research 203
7.4.1 Limitations of Hydra 203
7.4.2 An Audio Equivalent to Hydra 205

7.4.3 A Game Engine Built Upon Hydra 206

Bibliography **207**

List of Figures

1.1 *Figure showing the reduction in spending on modernisation of software for the US government (United States Government Accountability Office, 2017).* 22

1.2 *Retro City Rampage screens. Note that the copyright year is 2015 even though it is running on MS-DOS and has a very old fashioned visual look to it (Prescott, 2015 (accessed July 9, 2019)).* 23

2.1 *Diagram showing the isolation of applications each running within their own operating system (Reuben, 2007)* 34

2.2 *The architecture of a virtual frame-buffer-based application (Lok et al., 2002)* 46

2.3 *Diagram showing how an X11 server can abstract the complexities of the underlying system drivers for GUI applications such as XTerm and Mozilla Firefox* 47

2.4 *Screenshot of Citrix WinFrame showing the strong relationship with Windows NT 3.x. Screenshot of a typical Windows NT 4.0 TSE session connected from a Linux terminal running the open-source rdesktop client.* 49

2.5 *The technology stack of PhysX. Note the fairly limited range of hardware and the PC platform limitation to Windows. The use of a standard GPU has greatly increased potential hardware support in recent years* 52
2.6 *Requirements tree for console video games with importance factors (first two levels only) (Guttenbrunner et al., 2008) (Licensed under CC BY 4.0)* 55
2.7 *The technology stack based on using the Google Angle middleware.* . . . 59
2.8 *The technology stack when requiring OpenGL 1.2 compatibility on the HTML5 platform* 60
2.9 *A potential API stack required to support OpenGL on macOS since OpenGL was deprecated in version 10.14* 61
2.10 *A simplified view of responsibilities provided by the RedHat Cygwin compatibility layer* 63
2.11 *A very simple 3D program showing a rotating model running on Linux (Debian GNU/Linux 9)* 65
2.12 *Graph showing the time required to manually port the experiment program to different operating systems. Time is shown as percentage compared to time taken for original implementation.* 67
2.13 *Table showing the executable types supported by Windows 10 on ARM (Microsoft Corporation, 2019)* 68
2.14 *The final result of the port to Plan 9. It required the implementation of a software based 3D renderer (written in ANSI C) which was not only time consuming but also fairly resource intensive (especially in an emulator).* 70
2.15 *Graph showing the relative time saved when using the respective technique compared to manually porting the software* 71

2.16 *Graph showing the relative time saved per platform using the respective technique. Windows NT 4.0 and Plan 9 were already running in an emulator, explaining the less consistent emulation results* 72

3.1 *A view of the dependencies required for a simple game on Linux. Note that a large number of them ultimately depend on device drivers.* . . . 79

3.2 *A comparable view of dependencies required for the same game on Windows. Again, note that a large number of them ultimately depend on device drivers but the rest of the stack is quite different.* 79

3.3 *Diagram showing the large number of C header files required by the standard SDL2 distribution* 80

3.4 *Diagram showing the greatly reduced number of C header files required by SDL2 using a Hydra back end* 81

3.5 *Passthrough within the hypervisor, near-native performance can be achieved using device passthrough (Jones, 2009 (accessed January 2, 2019)).* 84

3.6 *A diagram showing that the Java platform on Android (Dalvik) is made up of many different dependencies to the standard Java reference implementation (Google, 2019 (accessed January 2, 2019)) (licensed under CC BY 2.5).* 86

3.7 *A diagram showing that the bytecode must ultimately end up being processed for a specific platform. Sometimes a JVM provides only a limited choice of supported platforms. Here Windows, Linux and Mac are shown as supported.* 87

3.8 *Diagram showing the interaction between the JIT system and the rest of the Java JVM* 89

3.9 *Graph showing the number of companies working together to maintain the Java platform.* 90

3.10 *Diagram showing the underlying technology and dependencies for Microsoft's .NET framework. 2011 IEEE. (Reprinted, with permission, from Teresa P. Lopes and Yonet Eracar, TPS development using the Microsoft .NET Framework, AUTOTESTCON, 2011 IEEE)* 90
3.11 *A diagram showing the indirection that a managed language needs to go through in order to call into a native or system library* 92
3.12 *A diagram showing the many layers of indirection that a managed language needs to go through in order to call into another managed platform and language* 93
3.13 *Diagram showing that Hydra is a fairly typical client-server application. Note in particular that many different client implementations can be used.* 102
3.14 *Components can easily be swapped with others if they have identical APIs.* 103
3.15 *An example listing of Hydra code which when compiled will display a triangle. Note that it is almost identical to the OpenGL counterpart.* . . 104
3.16 *One of the simplest Hydra applications to display a triangle on the screen.* 105
3.17 *Diagram showing texture data being retained on the GPU but vertex data being transferred across each frame.* 106
3.18 *Diagram showing both texture and vertex data being retained on the GPU. Only the instructions are sent across each frame.* 106
3.19 *Diagram showing both texture and vertex data being retained on the GPU in a similar manner to traditional OpenGL implementations. The difference is that it had been passed through a network / serial rather than simply across the CPU bus.* 107

3.20 *Diagram demonstrating the duplicate state system. Upon a flush to the client, changes needed to bring a state into sync with the current one can be identified.* 109
3.21 *Diagram demonstrating the separation between an environment where a tight loop is possible and one where code paths must complete so that execution can return to the web browser.* 111
3.22 *Diagram demonstrating the type of image data that is uploaded via Hydra in order to cause the least amount of blocking.* 112
3.23 *Diagram demonstrating how memory is accessed and that pointers in C are capable of referring to previously freed memory.* 114
3.24 *Diagram showing that a subsequent freed block of memory could be later allocated as a different type. Note that the pointer is also referring to data within the middle of the block.* 114
3.25 *Diagram showing the scanning of heap memory to ascertain which blocks of memory were no longer referenced.* 115
3.26 *Diagram showing the data making up a fat pointer in comparison to a standard raw pointer. Note that the fat pointer is much larger than the raw equivalent.* 118
3.27 *An example of libstent showing a fairly typical case of a use after free programming error.* 119
3.28 *The resulting error message for the previous erroneous program. Note that not only is the source unit file and line number exposed; but also the type of structure.* 120
3.29 *Code to create and destroy objects showing a subtle memory leak.* 120
3.30 *The resulting warning message for the previous leaking program. Note that the source unit and line number where the memory was allocated is also exposed along with the type.* 121

3.31 *The Tank Museum Gun Game was an arcade kiosk system where guests to the Tank Museum could utilise real weapons to shoot at targets on the screen. The hardware involved requires nothing more than a webcam for tracking and a serial cable for the trigger.* 122
3.32 *Diagram showing the locked memory block. Note that it is still allocated and is taking up memory, however, accessing it will cause the program an immediate abort.* 124
3.33 *Gameplay of Zombie Maths Game. A game for helping to teach children mental arithmetic in a fun and engaging way. This game was presented to the public at the 2017 BU Science Tent with support from the British Science Association and Siemens UK.* 125
3.34 *Screenshot showing the VR Spatial Audio tool. We aim to publish a paper using the results gathered from this tool in the future.* 127
3.35 *Diagram showing the percentage of errors reported by specific technologies. Lower numbers show that the technology failed to expose a specific programming error. Note that although libstent was close, none of the solutions provided 100% coverage. There were always errors missed that other strategies had uncovered.* 129
3.36 *A small sample of tests showing results with libstent enabled. The 100% pass rate shows that if the test contained an artificial memory error, it was correctly recognised and flagged by libstent.* 131
3.37 *A small sample of tests showing results without libstent enabled. These failures all represent major errors in programming code that would not immediately terminate the program and as such could be very hard to diagnose.* 132
3.38 *A simple listing to demonstrate how accessing invalid memory can cause undefined results later on in the program.* 133

3.39 *Diagram demonstrating how type safety is an important factor in keeping data consistent. Note that there is no guarantee that the referred data is aligned to a single type and may be misframed over other fields.* 135
3.40 *Listing of the Allocation structure used internally within libstent. Note that the ptr field is the first one.* 136
3.41 *Diagram demonstrating the approach taken by libstent to provide meta-data on individual allocations.* 138
3.42 *Diagram demonstrating the libstent vector architecture. Note that a similar pattern has been used to create contiguous dynamic arrays.* . 139
3.43 *Listing to code in traditional C++, modern C++ and C showing the subtle differences in object-oriented approaches. It is also interesting to note how much C++ has changed over time compared to ANSI C.* 141
3.44 *Class diagram showing a comparison between the approaches taken to implement an object in C and C++. It could be considered based on this example that the most modern C++ language features makes this task simpler.* 141
3.45 *Diagram showing the PIMPL pattern applied to the previous classes. Note that it is now showing many similarities to the C89 approach.* . 142
3.46 *An overview of the WebSocket protocol implemented as part of Hydra. This specifically allows for the communication with modern web browsers (Oracle Corporation, 2018 (accessed March 3, 2019)).* . 144
3.47 *A diagram showing the low-level layout of a WebSocket frame as specified by RFC 6455 (RFC6455, 2011).* 145
3.48 *Flow diagram of the Hydra WebSocket component* ***libws****. All WebSocket paths ultimately end in raw TCP socket communication.* 146

3.49 *Class diagram showing the design of libws. It has a number of subtle differences compared to that implemented using a conventional OOP language.* 147

4.1 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. 155

4.2 Diagram describing the layers that OpenGL is built upon compared to Hydra. Notice that Hydra has additional layers of abstraction. . . . 158

4.3 Diagram demonstrating a typical yet simplified communication between the client and server components of Hydra in order to upload a texture. 159

4.4 *A multiplayer (across a network) football game where the players are characters from traditional fantasy role playing games* 161

4.5 *Screenshot showing four wave clients connected. They all look similar because they all share the same drawing commands apart from a few subtle differences. For example, the player select menus are unique.* . 162

5.1 Graph comparing the bandwidth requirements between Hydra and VNC with a varying number of objects in the scene. 165

5.2 Graph comparing the bandwidth requirements between Hydra and VNC with an increasing image resolution. 166

5.3 Screenshot of an example Hydra 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. 169

6.1 *Screenshot showing a Windows NT 4.0 VM running Half-Life and a Linux VM running the software renderer* 180

6.2 *Screenshot showing the sample map within Blender and being rendered by the software renderer* 181
6.3 *Graph showing the CPU utilisation of Half-Life using the software renderer and OpenGL accelerated render compared to the internally implemented software renderer. All instances are running at 800x600 resolution on a native machine running Windows Server 2012 R2.* . 182
6.4 *Graph showing Half-Life software renderer against internal software renderer at different screen resolutions running in a fully emulated Windows NT 4.0 environment.* 183
6.5 *Graph showing the CPU requirement of different renderers running at 800x600 resolution on the different test platforms.* 185
6.6 *Graph showing the CPU utilisation required by various resolutions for the accelerated renderer compared to the Hydra renderer running on Windows NT 4.0 operating system on native hardware.* 186
6.7 *Graph showing the CPU requirement of the commercial software renderer vs the Hydra renderer at various screen sizes whilst running in a fully emulated environment.* 187
6.8 *Graph showing the CPU resource utilisation for collision. Note that rendering had been disabled for this test.* 188
6.9 *Dialog showing a limited selection of tweaks that can be performed in order to run old software. Unfortunately there is no guarantee that these compatibility features will exist on future versions of Windows.* 190
6.10 *Screenshot showing the graphical calls being externally diverted from the VM and into a web browser running natively on the host. Note the garbage on the GtkRadiant window because it is never being rendered to or cleared.* 191

6.11 *Diagram showing the flow of data between Hydra running in the emulator and the standalone Hydra rasteriser running natively on the host.* 192

6.12 *Screenshot showing the final result of the external raster solution. Whilst it did require modifications to the GtkRadiant code; it now looks and feels native running in the emulator.* 192

6.13 *Screenshot showing various generations of Steam platform deprecations. Often the games themselves have not been updated; just the Steam launcher which unfortunately is mandatory for DRM and anti-piracy reasons.* 194

6.14 *The resulting dialog when trying to run a recent Qt application, compiled with the latest GCC based MinGW compiler.* 194

7.1 *Diagram from Khronos showing a simple overview of how the OpenGL Safety Critical profile aligns to the commonly used OpenGL profile (Khronos Group, 2003 (accessed February 9, 2015)).* 205

Publications

- ***Karsten Pedersen, Christos Gatzidis, and Barry Northern. 2013. 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 (GAS '13). IEEE Press, 40–43.***

  The content of this paper demonstrates an early attempt at separating the game logic from the underlying platform. Whilst it provided promising results as a networking solution, it required a fair amount of engineering to move the core game logic from one engine to another. However the discovered limitations and new ideas developed at this stage underpinned the later research, particularly within **Chapter 3** where the rationale behind the reduction of dependencies is introduced. Perhaps one of the most critical answers this preliminary investigative research provided was how low-level did we need to go in order to capture the ability to develop in a platform agnostic manner. Ultimately future work had to be done at the graphics API level rather than a high-level game engine.

- ***K. Pedersen, W. Tang and C. Gatzidis, "OpenGLD - A Multi-user Single State Architecture for Multiplayer Game Development," 2017 International Conference on Cyberworlds (CW), Chester, 2017, pp. 198-201.***

This conference paper shows the early stages of investigation into the benefits from separating the core game logic from the rendering pipeline. In the simplest terms it allowed for the removal of the client-side state which in turn removed the requirement to manually synchronise the clients with the server. This greatly reduced complexity and provided a novel approach to multi-player networked games. Much of the work here is covered in **Chapter 3** where the architecture of Hydra is discussed.

- ***Simons, Alain & Pedersen, Karsten & Abdulaziz, Hasan & Melacca, Davide. (2016). Scale Model Games (SMG): An Introduction to a New Type of Game Play. HCI 2016.***

  The research presented in this paper demonstrates the use of Hydra in an embedded environment. In particular by separating the platform specific logic from the more intensive rendering via the use of Hydra, it was possible to run entire software program on less powerful ARMv6 processors and directly interface with the hardware without needing to develop a custom protocol to synchronise between the main rendering PC. This research supported the initial idea that utilising a technology like Hydra could greatly simplify the architecture and development of many multi-user applications. This is one of the areas that this PhD thesis focuses on and underpins the work identified in **Chapter 6.2**. Particularly the topic of sensor synchronisation from an embedded device.

- ***Pedersen, K., Gatzidis, C. and Tang, W., 2018. OpenGL| D-an alternative approach to multi-user architecture. In Transactions on Computational Science XXXII (pp. 57-74). Springer, Berlin, Heidelberg.***

  The research presented in this journal article shows the comparison between standard streaming approaches offered by VNC and the new approach taken by Hydra in terms of bandwidth cost. This experimentation was based on the

results of the work undertaken in **Chapter 4**. The novel multi-user architecture was made possible by separating the core logic from the platform in such a way that the isolated logic could then be designed to interact with multiple clients.

Chapter 1

Introduction

1.1 Digital Preservation

The digital preservation of software is the process of maintaining a codebase and surrounding dependencies so that it remains usable after an extended period of time (Zabolitzky, 2002). This means that as the surrounding software and hardware, such as an operating system or graphics card needs to be replaced, the software can still function as intended. Certainly in the IT industry, nothing can remain in the same working state forever. Hardware not only breaks over time but it becomes impossible to source exact replacements for because the original manufacturer has long since stopped fabrication of that model of hardware. Software needs to evolve and adapt at a relatively similar pace to the hardware in order to stay functioning. A typical way to preserve software is to simply update the code to ensure it still runs on the latest platforms (Grover and Nolan, 2007). This is called porting and is discussed further in **Section 2.12**.

1.2 The Importance of Digital Preservation

There are many reasons as to why we may want to preserve software, ranging from commercial and monetary reasons to historic and even personal nostalgia-related reasons. Keeping business critical software running is extremely important. Any downtime due to newly introduced bugs or training can have a potentially significant impact on income. A common symptom of this need to keep software working can be seen in automatic teller machines (ATMs). Even in 2020, there are many installations of bank ATMs that are still running Microsoft Windows XP, which was released in 2001, making it 19 years old. Microsoft had dropped official support for the product in 2009 and extended support ended in 2014, making it crucial that companies running these systems migrate to a newer solution. It can be particularly damaging to use old technology, especially if it is open to the public or if it is connected to a network (ATMs have both of these characteristics). This is due to the rise in viruses, exploits and vulnerabilities, not just from the software side but now also from the hardware side too, because of Spectre and other vulnerabilities (Kocher et al., 2018). The companies behind the vulnerable hardware will only provide support and protection for their current and future hardware so this suggests that keeping and maintaining older hardware as a means to keep the old software running is simply not feasible without some kind of isolation strategy. This use of antiquated software happens behind the scenes and rarely comes to light. One example from 2015 is the legacy software called DECOR handling weather information communication with pilots (Longeray, 2015 (accessed February 9, 2015)). It was running on Windows 3.1 which was over 20 years old at the time. The reasons cited for the archaic system was that people do not like to do maintenance and the reason for it to fail was that they were starting to lose expertise in dealing with that type of operating system. At that point in time there were only three specialists who could deal with DECOR-related issues. DECOR was planned to be replaced in 2019 but the developers still do not

seem optimistic and the retirement has not yet occurred so it is highly likely that the Windows 3.1 software is still running for many more years.

Until as recently as 2012, New York City used IBM OS/2 to power the swiping fare card system and many of its bank ATMs (Egan, 2019 (accessed July 9, 2019); Bass, 2014 (accessed July 9, 2019); Scott, 2014 (accessed July 9, 2019)) There are instances of these machines running IBM OS/2 released in 1996. This shows the importance of these systems and exemplifies the reluctance to upgrade. Unlike DECOR, where there was apparently no risk to passengers, using a known vulnerable operating system to manage important data and money is potentially quite dangerous.

This reluctance to upgrade operating system is caused by a number of issues. The main one however is the ATM software. If a different operating system is used, this software will need to be ported to work on the new platform and will also need testing performed on it. In the worst case scenario, the software will need to be rewritten. These issues are discussed further in **Section 1.3**. In many cases, the software itself is up-to-date and maintained, however, the platform or features that it requires (such as the operating system) is out of date or obsolete. It is frustrating to have to rewrite a piece of software that is not broken just because the platform features that it utilises and depends upon are no longer available in later operating systems. So, it is often the chain of dependencies that causes a software to have portability issues rather than the software itself.

A report from the US government in 2016 demonstrates that they are still relying on floppy disks and other technology from 50 years ago (United States Government Accountability Office, 2017). Interestingly that whilst they state that using archaic technology gives them a strategic advantage, they must find ways to modernise their technology. The ageing software of particular note is for the Strategic Automated Command and Control System (SACCS). This is their legacy system that coordinates the operational functions of the nation's nuclear forces. This report also presents data (**Figure 1.1**) showing that the amount of funding available for modernising software

is decreasing and that ageing software is likely to remain for the foreseeable future, even for critical roles.

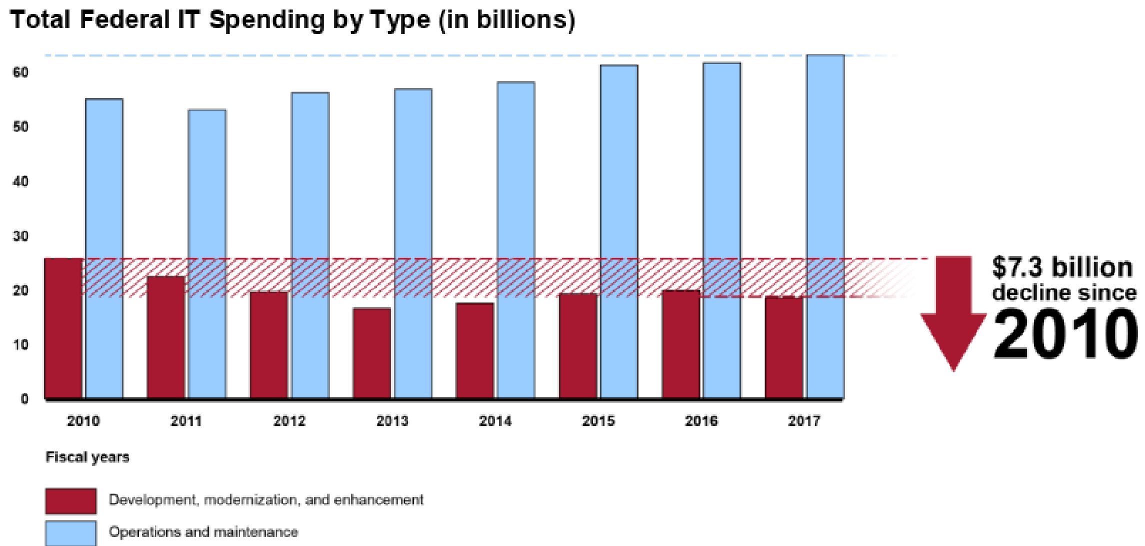

Figure 1.1: *Figure showing the reduction in spending on modernisation of software for the US government (United States Government Accountability Office, 2017).*

Digital preservation is becoming even more popular within the hobbyist sector. It is very common that someone may want to play an old game from their childhood. Perhaps due to nostalgia reasons rather than any technical advantages, older games can still be an enjoyable experience and in most cases can be much cheaper than current commercial titles. Often this software is now abandonware and no longer supported or distributed by the original publisher. This form of retro gaming is very popular within communities, so much so in fact that even modern titles are being released for older platforms, either as a technical demonstration or as a form of widening the target market. For example the title Retro City Rampage (**Figure 1.2**) was originally released for Microsoft Windows 7 but has subsequently been backported to run on Microsoft MS-DOS and IBM PC-DOS (Prescott, 2015 (accessed July 9, 2019)).

Figure 1.2: *Retro City Rampage screens. Note that the copyright year is 2015 even though it is running on MS-DOS and has a very old fashioned visual look to it (Prescott, 2015 (accessed July 9, 2019)).*

Keeping older software working is useful as a form of history and digital archiving, for example, in a very basic sense we can learn what did and did not work well from past products. Even the most modern software available today will have similarities with software originally developed in the 80s. If this information was lost not only would it be a shame but a significant amount of knowledge would need to be re-invented for modern software projects (Lavoie and Dempsey, 2004).

1.3 Difficulties in Preserving Software

Software preservation can be complex to achieve with software because technology is always evolving. Entire software stacks evolve together with hardware, sometimes in a completely transparent manner. However, especially in the case of entertainment software, if left behind, it can be very complicated to pick up the software again and maintain it against the current state of progression.

The most common complexity during the task of maintaining software is centred around software dependencies. A software dependency is often a library that the immediate software relies on. This can be recursive so that dependencies of dependencies are just as crucial to getting the original software working. If just

one of these dependencies becomes unavailable, the chain is broken and either the dependency needs to be maintained and fixed itself or the code relying on it needs to be replaced in order to perform the task itself or utilise a different dependency.

Examples of pure software dependencies could be libraries such as an image loader or a file parser. These can be written in such a way that they require no other dependencies, other than to be able to read data from a file. These are relatively straightforward to maintain.

Slightly more complex dependencies consist of libraries that call into the operating system, such as GUI libraries utilising the operating system's default looks and feel or encryption libraries that utilise the operating system's in-built keychain. These are often straightforward enough to maintain so long as the operating system has good backwards compatibility strategies in place (such as the ones found in commercial UNIX or non-consumer editions of Windows). However, moving between different operating systems often means replacing parts of the code with the respective code for the new target operating system.

If an entire operating system becomes inaccessible, such as RISC OS (for around 10 years before the Raspberry Pi gave it a new lease of life), a developer will need to port their software to an alternative existing platform in order to successfully continue being commercially viable. Other examples of deceased operating systems can include large platform changes, such as Mac OS 9 moving to Mac OS X. Though the operating system retained the same name (and in some respects, look and feel), the migration to a UNIX-based platform caused a large amount of existing code to break, so much so that Apple invested in developing and licensing the Rosetta compatibility layer to help ease the transition. Though not always the case, it is extremely useful for an operating system vendor to maintain good backwards compatibility. Microsoft offers a good example in that software for DOS, which was a popular OS at the time, can usually still work on Windows as modern as Windows 7 almost 20 years later with the in-buit NTVDM (NT Virtual DOS Machine) (Anderson et al., 2013). The

functionality of NTVDM has not come for free however and is a complex piece of software to maintain. Much of the work was undertaken as a partnership between Microsoft and IBM in order to allow OS/2 Warp to execute DOS binaries (which at the time was the most common platform). Potentially, if this code was not already mostly in place, modern Windows would not have this level of backwards compatibility. However, even this software is beginning to show its limits and is only available on 32-bit installs of Windows, greatly limiting its use because Intel hardware has moved on to 64-bit and it is common-place to have a 64-bit install of Windows to take advantage of that. Of course, Microsoft could maintain the software to make it work with newer hardware in a similar way to WOW64 (discussed further in Section 2.4) but there is little incentive to do so, digital preservation at this level appears not to be a priority for them; again, showing that relying on commercial software vendors to provide backwards compatibility is often not a viable strategy.

The most complex dependencies to replace are those that require physical hardware. Examples include OpenGL, which is not simply a 3D drawing library for programming. Instead, it is an API specification to use GPU hardware and drivers implemented by hardware manufacturers. Without the physical hardware, its use is very limited. Another example includes OpenAL which, again, required specific audio hardware to work (unlike OpenAL Soft (StrangeSoft, 2019 (accessed July 9, 2019)), which is more common nowadays that the CPU is fast enough to do all processing on a generic processor). When migrating software between operating systems, the new target often provides these same libraries; however, when the hardware itself becomes obsolete, the software that depends upon it can become very hard to maintain. Instead, the code needs to be largely rewritten to support newer hardware. An example of this could be refactoring a codebase using the 3DFX Glide API to DirectX. This task would require a fair amount of time to complete. For a more modern example, potentially in the future, Apple's Metal API could be replaced by the more cross platform Vulkan. Again, this could require significant

efforts from developers to migrate their code to the new API.

One of the final blocking factors to maintaining software is less technical and more related to human impacts and licensing. If a company is no longer in business, there is very little support in terms of them maintaining one of their old products. So, if a software was to originally use a proprietary sound library (for example) from that company, there would be no way to modify the code and port it to a new platform. If the terms of the license included access to the source code (source code access is still common for proprietary libraries to facilitate better integration with existing systems), in theory the downstream developers could modify the code to run on newer platforms but they in turn would not be able to release that change so others can benefit from it or it could still be a license violation. Instead, this potentially complex work would need to be duplicated by others.

This is one of the main reasons why older titles and game engines are not made open-source. An example; Unreal Engine versions 3 and below are not accessible to the community because they themselves depend on 3rd party middleware that is from companies no longer in trade, making it impossible to ever get the green-light for release and the guarantee that no license agreement is being violated. This is unfortunate because the older versions of Unreal would be perfect for lower powered devices such as mobiles or tablets.

In increasingly frequent cases, a library will have DRM inbuilt into it. This means that if the upstream company ever ceased trading, the license server that the library contacts before it initialises will also be taken offline. For example, if Valve ever went bankrupt, there would be no funds available to pay their developers to go through every game on the Steam software store and modify the code to remove the DRM and license checks. All games, tools, libraries and even game engines such as the Source Engine, purchased through Steam at that point will become unusable. There are currently no laws in place to protect downstream consumers from this, potentially making the research presented in this thesis a much more pressing topic in future.

1.4 Aims of this Research

This research intends to solve a limitation with current emulation techniques and not only allows the guest access to the host GPU but does so in a generic manner that works for any emulated operating system and host. This not only produces a solution to the problem of emulating a GPU but also solves it in a manner which satisfies the constraints set by digital preservation over time.

The specific research question is; by separating the graphical pipeline from the rest of a program utilising 3D graphics, can the portability and future maintenance be improved? Can we go beyond simply improving these attributes and instead look more towards achieving a platform agnostic solution in which the portability is guaranteed in the future?

The prime use of the GPU in this research will be to render 3D graphics for software such as games. However, a large proportion of the research outcome will be directly transferable to other software disciplines. Allowing developers to choose where and when to upgrade their products as opposed to being forced to stay current with their upstream vendors will permit for a much more stable and deterministic product. In turn, they will be able to maintain their software for longer passing this benefit further down to their users. This should benefit the industry as a whole. This research has four objectives:

- **To investigate a platform agnostic approach towards the digital preservation of software**. This includes the viability of long term maintenance and portability. In order for the guest to access the host GPU, software must be written that has virtually no limitations detracting from its portability. This will greatly reduce the amount of work to deploy it on newer or exotic platforms without placing a strain on the developer in order to maintain it.

- **To develop a system that allows access to the host GPU in a generic manner**. This means that using a platform agnostic approach researched in the first objective, a system is to be built that can simply pass commands between the guest and host without adding additional engineering requirements for the user. It must fit in any existing design or development methodology.

- **To evaluate the success of this approach**, not just in terms of performance but also its applicability within the domain of 3D software development. For example, the kinds of software portability issues that 3D graphics passthrough can solve will be investigated.

- **Explore any innovative possibilities in terms of multi-user interaction** which may potentially have been exposed by separating the system logic from the presentation of graphics in such a way that it can be run on separate machines, contributing to now multi-user interaction.

1.5 Outcomes of this Research

The outcome of this research is the design, development and implementation of the OpenGL specification into an ANSI C library called Hydra that can work across machine boundaries, effectively solving one of the last limitations preventing hardware accelerated 3D software and games from running in an emulator. The solution presented in this thesis is more portable and thus suited towards digital preservation than the current state of the art of emulation / GPU passthrough discussed in **Section 2.6** in that it does not need a specific emulator to work, nor does it need additional code written and due to its design, it can work on almost all existing platforms. This includes much earlier operating systems such as RISC OS, DOS, Apple System7, NeXT and Plan 9, to more recent platforms that are in common use today. It can even work on embedded platforms such as Arduino (if it has network capability).

Therefore, any software written using it is very likely to be preserved digitally, even as newer platforms come out and current ones die out.

This research then analyses and compares Hydra against existing solutions with a specific focus on digital preservation. Hydra and the results gathered are evaluated in terms of whether it was successful in achieving its goal of facilitating digital preservation and platform agnostic development.

Finally, the use of Hydra as an underlying platform for multi-player development will be explored. Any benefits in terms of safety, usability and performance will be identified.

The desired goals to be reached within this research are:

- **A conducted literature review** providing insight into the current state of the art in terms of porting software and digital preservation. From this an analysis can be performed looking at any deficiencies and an alternative solution can be proposed and implemented.

- **The implementation of a platform agnostic technology** called Hydra which provides an implementation of OpenGL that is designed to work effectively through the boundaries of an emulator allowing for the separation of graphics from not only the programs logic but also the entire emulated platform.

- **The implementation of an ANSI C safety library** which will serve as a framework for the technical implementation of Hydra whilst greatly reducing potential programming errors that the C language does not protect against.

- **Provide evidence of viability of Hydra** and a demonstration that in cases where hardware acceleration is unavailable (such as within an emulator) the approach from Hydra can provide a feasible solution which provides improvement on the existing technique of software rendering.

- **Provide evidence that multi-user software can benefit from Hydra** and

results that demonstrate that the performance offered by existing streaming solutions such as VNC can be improved upon with the introduction of an intelligent protocol provided by Hydra that maps well to an existing underlying Application Programming Interface (API).

1.6 Overview of this Research

Chapter 2 provides the conducted literature review where a range of issues pertaining to both cross-platform development and digital preservation were explored. This covered a number of use-cases such as businesses keeping their critical software running and hobbiests, attempting to keep their favorite games alive. A number of solutions were identified and evaluated ranging from compatibiliy layers to full system emulation. The literature generally agreed that emulation was the solution most likely to yield best results but there were still a number of inherent issues which would need to be overcome. For example access to hardware specific devices such as the GPU and also the emulation software itself becoming potentially difficult to maintain.

Chapter 3 presents the methodology which explores the technical details behind the primary solution presented by this research. An architecture and reference implementation called Hydra allowing for GPU passthrough in a generic manner and compatible with a number of different emulation software from different vendors. Work undertaken to implement Hydra in a platform agnostic manner was also presented, including a thorough description and rationale of the technological improvements to how native ANSI C can be utilised leading to greatly improved maintainability and portability. Testing relating to memory safety and correctness was also provided.

Chapter 4 explores the inherent complexites in synchronising state between multiple computers. This is an important factor to the success of exposing the

GPU to emulation software in a generic manner. Literature surrounding the area of synchronisation is consulted and analysed in terms of effectiveness and appropriateness to the specific use-case along with initial prototypes. The specific topic of synchronisation is then explored further by looking at any potential advantages that having a network aware graphical renderer can provide in terms of multi-user interaction and cheat prevention.

Chapter 5 provides a comparison of streaming techniques using traditional rasterisation approached such as VNC and the intellegent protocol offered by Hydra. The main focus is on bandwidth consumption at different resolutions and when synchronising many objects. A comparison with common game server approaches (such as QuakeWorld) is also provided. These results help to investigate if the technology behind Hydra can provide any advantages to the manual synchronisation of remote objects through IPC. A number of potential optimisations are discussed based on these results which could help improve Hydra's use for this purpose.

Chapter 6 details the experimentation undertaken to acertain the success of the approach to digital preservation. The first experiment was designed to measure the bandwidth utilisation of the network aware protocol described in the previous chapter compared to the popular VNC streaming solution. The results gathered from this were very important because this underlying network protocol is key in driving the GPU passthrough between VM and native hardware.

A second experiment was undertaken and designed to compare the success of Hydra itself as a medium to run 3D software from within a fully emulated environment. The performance of Hydra in terms of CPU utilisation and frame-rate was compared against that of a custom built software renderer running within the same environment and both a software renderer and hardware accelerated renderer provided by Valve Software's Half-Life.

A final experiment involved a more practical exercise using Hydra and an attempt to directly run an existing software package, the GtkRadiant level editor within a fully

emulated environment. This was intended to encounter and experience any potential complexities that could appear in the future when using Hydra. Real-world software, especially tools featuring complex GUI systems provide a number of useful challenges to overcome.

Chapter 7 provides a review of the research aims and a summary of the research undertaken. A conclusion is then formed based on how the objectives of each aim have been met. Finally the intended future research is discussed including technical improvements that can be made to Hydra to cover support for a wider range of hardware and use-cases.

Chapter 2

Literature Review

2.1 The State of Digital Preservation

In recent years, digital preservation has become a significant issue with many diverse groups and organisations recognising the requirement to preserve documents and software (Doyle et al., 2007). As stated in the well-cited paper by Kuny (1997), in the current climate, digital providers facilitate access of data but do not facilitate preservation. The recent growth of cloud services and products is allowing for a great resource for content but potentially for a relatively limited time (Caplan et al., 2005). The vast majority of websites accessible today are rarely over 5 years old (Chen, 2001). Systems such as The Internet Archive and the Wayback Machine have been developed to partially solve this problem (Murphy et al., 2007). Chen (2001) provide a good overview of the types of information that has been lost by 2001. This includes 50% of the films produced in the 1940s, most TV interviews and even the first e-mail sent in 1964. This data demonstrates a large loss of cultural heritage.

There are methods of evaluating the effectiveness of storing digital information, such as the OAIS standard, but this does not provide representation information about the preservation environment itself (Giaretta, 2008). This means no standards have

been devised to provide guidelines for exactly how data or software is to be preserved.

2.2 The Roles of Emulation and Virtualisation in Digital Preservation

Emulation consists of developing a complex piece of software that not only simulates the required host platform but also the entire hardware, down to the registers and CPU. In many cases this technique is one of the most guaranteed ways to run an older piece of software on a newer or different platform. Emulation is not a new technology, with the term "Virtual Machine" being first used to describe an operating system concept in 1960 (Rosenblum, 2004). There are research papers dating back as early as 1973 (Canon et al., 1980; Facey and Gaines, 1973; Mace et al., 1974) looking into ways to improve emulation performance on the mainframe computers of the time. These papers provide a wealth of information. Much of it is still very relevant today. In the enterprise world emulation allows for benefits in security such as process isolation (Reuben, 2007) (**Figure 2.1**) and also for improved scalability and utilisation of resources (Ray and Schultz, 2009; Christodorescu et al., 2009; Luo et al., 2011).

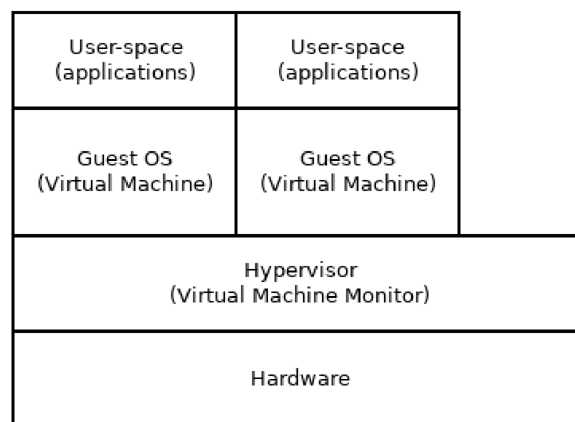

Figure 2.1: *Diagram showing the isolation of applications each running within their own operating system (Reuben, 2007)*

Outside of the enterprise and onto the consumer desktop area, emulation has traditionally been seen as software for hobbyists or for those looking for nostalgia (Rosenthal, 2015). However, this notion is quickly starting to change as it is now being seen as one of the only ways to reliably maintain older software that cannot be ported to newer platforms (Liebetraut et al., 2014). Many emulators attempt to emulate a single hardware specification, such as a games console. This can often be seen as a slightly simpler task because certain assumptions can be made about how the hardware interacts. However, cycle accurate emulators also exist, which provide the best accuracy at the cost of performance and host computer resources (Van Der Hoeven and Van Wijngaarden, 2005).

Emulation strategies are now becoming viable complements to migration (Von Suchodoletz et al., 2010), whereas in the past many were sceptical about using them due to their complexity (Van der Hoeven et al., 2008). Doyle et al. (2007) provide a good example of this in their attempts to develop a digital archive to preserve anthropomorphic data such as human height, weight etc. CAESAR 3D was one of their research outputs. It was a database archive that makes use of an IBM DB2 database and they recognised that this software may no longer be provided in the future by IBM. For this reason, they have invested time into researching the possibilities of maintaining it themselves. They too have arrived at the conclusion that emulation is a good avenue. However, unlike others arriving at similar conclusions (Granger, 2000; Rechert et al., 2016), they recognise that their chosen emulation platform is not necessarily implemented with long term digital preservation in mind and that in future their chosen emulator software itself may need preserving.

Whilst Rechert et al. (2016) did not detail potential issues with emulation platforms, they did describe potential issues with virtualisation platforms in terms of digital preservation. They stated that the close ties to today's computer platforms restrict a virtualised machine's longevity, particularly a virtual machine relying on contemporary virtualised hardware components or hardware that potentially may

not exist or has been obsoleted in the future. This idea is further supported by the work undertaken by Von Suchodoletz (2011) in that they state that the long term availability of emulators remains uncertain and that many emulation tools have only existed for under 10 years, providing very little proof of longevity. In particular, those products that have been available for a long time have gone through a number of changes to the virtualised hardware and actively deprecating older platforms such as Windows 3.x. This greatly diminishes their fitness for purpose as serious vessels for digital preservation.

This leads us into the idea that even though perhaps a specific emulator will be unlikely to stand the test of time, it is very likely that a number of new emulators will be available to take its place. Emulators themselves may be difficult to write, with some sources suggesting around 2 man years (Van der Hoeven et al., 2008), however when it comes as a business requirement, this is quite an acceptable cost. Still, in many cases a number of general purpose emulators will already exist, including both hobbyist and commercial. Therefore, it becomes important to be able to evaluate and ultimately choose between them on which one will be most useful to fulfil its task and be able to provide a usable environment to the older software.

Work into measuring the effectiveness of emulators has been explored and undertaken (Guttenbrunner and Rauber, 2012). This is important because it allows for improving the process of what tools we adopt in the present as part of our digital preservation strategy in the future. Potentially in the future we will not be able to act on hindsight and if the strategy fails, it could be costly, both financially and through loss of digital heritage.

Some of the characteristics of an emulated solution stated by Guttenbrunner and Rauber (2012), which can be measured, include:

- Frame rate (average, max, min)
- CPU cycles per second (average, max, min)

- Number of files opened on a certain I/O device
- Number of bytes read from I/O devices
- Number of certain input/output events

The solution presented in this thesis will be measured in ways very similar to this, particularly when it comes to graphical performance and bandwidth compared to existing solutions, again measured in a similar way.

After recognising that emulators can themselves become obsolete Rothenberg (1999) have produced a set of specifications so that new emulators can be produced in the future with suitable compatibility for a specific digital artefact that needs preservation. Similar work was undertaken by Jamraj et al. (2017), providing emulation models to aid with the requirements of emulators capable of supporting legacy software. In this work they tested the success of these models against three emulators. Two of these emulators were originally designed as hobbyist projects with no real safety or correctness guarantee. These were Project64 and Basilisk II for the N64 and Mac OS 9 platforms respectively. The feasibility for these emulators comes from the fact that they were originally open-source and so have been maintained through the years. Contrast this to the emulator used for the Windows Phone, which is a proprietary and closed-source product from Microsoft; being one of the only emulators for the Windows Phone platform, without a potentially hobby or open-source Windows Phone emulator being developed, a void will appear in which we can no longer emulate this platform. Because the Windows Phone operating system is itself proprietary and on a ROM, the legality of developing a Windows Phone emulator with a focus on digital preservation will make it extremely unlikely to come to fruition, regardless of what emulation requirements models are produced, such as those by Jamraj et al. (2017) and Rothenberg (1999).

An emulator has been developed with the sole purpose of digital preservation (van der Hoeven, 2007; Van der Hoeven et al., 2008; Von Suchodoletz and Van der Hoeven,

2009). Dioscuri has been developed in a modular way after the authors had recognised that flexibility was extremely important. The additional flexibility is needed in order to support the majority of features required to provide suitable environments for a vast range of older digital software. However, it has been stated by Morrissey (2010) that open-source is not always the solution for a "free" digital preservation strategy. Software needs to be analysed to examine issues potentially reducing the feasibility for digital preservation. For example, the Dioscuri emulator is open-source but relies on the Java virtual machine. This complex piece of software is therefore the limiting factor when it comes to future maintenance, rather than the emulator itself. Solutions to this are discussed in Section 2.9. Occasionally, software is so complex that even though the source code is available, no-one with the required skill or time is able to maintain the code so that the software will work. This is particularly an issue with emulators because the developer needs a vast technical knowledge of the platform being emulated to make any real progress with maintenance tasks. These skill sets or expertise often disappear along with the ageing platforms, unless there is a suitable amount of documentation (Lee et al., 2002).

2.3 Emulation Performance

In order for an emulator to provide an adequate experience for the user in the context of running gaming titles it needs to allow the software to run at the intended speed. If it runs too slowly, the experience will be reduced. Due to the limitations imposed by virtualisers rather than pure emulators, this section will discuss only entirely software-based emulators rather than virtualisers that require host support from the CPU. In the case of one popular emulation platform QEMU, which can also utilise the host's virtualisation capabilities via KQEMU (Bartholomew, 2006), only the unaccelerated capabilities will be explored. KQEMU is also deprecated now in favour of KVM, a largely Linux specific virtualisation system (Ribiere, 2008), reducing

the number of operating systems that it can support.

With emulation, there is a very obvious trade-off between portability and performance. A running trend with many emulation solutions is utilising the virtualisation capabilities of the host (Mihocka and Shwartsman, 2008), which as discussed before requires a specific host processor and a specific guest environment. Failing this, many solutions then fall back on JITing, which consists of translating instructions being emulated to the host platform. This reduces portability because not only does this translation of architecture instructions needs to be provided for each platform the emulator intends to run on but it also means that the operating system needs to provide the ability to perform this JIT related functionality. Sometimes this requires a significant amount of operating system specific code. Crucially both of these techniques provide large performance improvements and in the case of virtualisation, there is almost zero performance cost. Bellard (2005) states that QEMU is a good compromise between performance and complexity. It provides a 4x loss in performance compared to native execution on integer code. It also provides a 10x loss in performance on floating point code due to lack of access to physical floating point hardware. These figures show relatively little performance loss when we consider that no unique or specialised hardware is required to facilitate this form of emulation.

One important point worth noting is that whilst no specialised hardware is required to facilitate these results, much platform specific code was needed, reducing the portability of the emulator and potentially reducing its value for future use for digital preservation. Whilst QEMU is generally well ported to a large variety of existing platforms, there is still a large amount of complexity and unknowns whilst doing so. This is best demonstrated in a technical paper by Filardo (2007), which details a large amount of the complexities experienced in order to port QEMU to the Plan 9 operating system. Plan 9 was originally presented as the successor to UNIX and was largely written by the same team at AT&T labs. It has many similarities to UNIX but

at the same time provides many radically different designs. In some ways, Microsoft Windows and UNIX have more in common than Plan 9 does with UNIX directly, particularly in the networking area, where Microsoft Winsock is largely based off an older specification of UNIX / POSIX sockets. Many of the largest issues discussed by Filardo (2007) include a lack of virtual memory, making user-mode emulation a complex task and an ANSI C compiler which does not provide the extensions to inline assembly code, making a rewrite of a large proportion of the code necessary as well as disallowing arbitrary jumps, which, again, may require a large refactor of the codebase. Sadly, the port of QEMU to Plan 9 was never completed and development stopped in 2007.

Work undertaken by Ding et al. (2011) builds upon Bellard's work in order to yield better performance results by better utilising a modern processor. What this means is parallelising the code to support not only multiple threads but also multiple cores. Their work (PQEMU) yields between a 1.8x to 3.7x performance increase compared to the base QEMU. These results are impressive and would mean that compared to running software on a native processor, we would only see a 2x speed decrease under the perfect conditions. However, the use of parallelism to achieve this has a reasonably negative impact to portability. These benefits would also only be seen on a modern multi-core device. If in the future we start seeing less powerful devices with single cores such as is common on mobile devices today, there would be no benefit to running PQEMU. In fact, the authors described that there was a potential overhead of 12% as many of the threads are correctly scheduled.

2.4 Binary Translation

Emulation does not necessarily need to provide an entire virtualised computer, instead just the CPU may need emulating. This lighter form of emulation is known as Binary Translation (Zheng and Thompson, 2000) and has many benefits. The main one is

performance; by avoiding having to emulate every component of a computer, ranging from the BIOS to the graphics card, many resource intensive tasks could be avoided. Due to the fact that binary translation is still emulation, there are still performance costs to consider, such as the time required to interpret the processor instructions before executing them. However, because, as mentioned, not all components of the computer need to be emulated, these additional costs can be avoided.

Binary translation also has the potential to integrate better with the host operating system in that rather than running the software in an emulator, the software runs as usual but the code itself is just passed through the binary translator to simulate the instructions. The rest of the application is then able to access system libraries such as the C standard runtime or even GUI toolkits (Wang et al., 2007).

A disadvantage of binary translation compared to full emulation is that the translator itself needs to know about both the guest and the host operating system. It is very complex to provide a generic translator because it is just providing the emulation of the CPU; for the rest of the components it needs to have knowledge of the host system in order to use them.

An effective example of binary translation is demonstrated in Microsoft Windows 7. Many people first assume that because a 64-bit Intel processor is fundamentally the same as a 32-bit processor with 64-bit extensions that 32-bit software can just execute on a 64-bit operating system without any complexities. This is not at all the case. Microsoft has implemented an emulator called WOW64 (Microsoft Developers Network, 2018 (accessed January 4, 2019)) (Windows on Windows 64) that dynamically translates 32-bit binaries to run on a 64-bit processor at runtime. This is what allows the older 32-bit software to work. The same performance costs for binary translation do occur though and, again, a lesser known fact is that a 32-bit Windows application will run slower on a 64-bit installation of Windows than the same software on a 32-bit installation of Windows. This is rather unfortunate because even today many games for modern Windows are released only as 32-bit binaries in order

to maximise compatibility with slightly older 32-bit hardware and their respective installations of Windows.

QEMU is a popular tool to use for dynamic translation (Liu et al., 2015). In particular, QEMU Static can be used to provide just dynamic translation rather than full system emulation. One of the most typical uses for it on UNIX-like operating systems such as Linux or FreeBSD is to create a chroot or jail populated with executables and libraries compiled for a foreign architecture and then instruct the host kernel to pass any binaries with a foreign ABI (Application Binary Interface) through the QEMU Static translator. This then allows the chroot or jail to function almost entirely as if it was populated with native software. Again, this is a very powerful feature because only the userland software is then needed to be translated, the actual kernel of the operating system which potentially does the majority of the work is of the host's native architecture and can run at full speed.

Android and iOS are very popular albeit mobile operating systems which run primarily on ARM processors. This is often due to ARM processors currently providing the best compromise of price and power for mobile hardware. However, this is also one of the reasons as to why they are lacking a large suite of quality software which was originally developed for Intel x86 processors. Binary translation can be used to good effect in this situation by allowing the x86 machine code to be translated to the ARM or ARM 64-bit processor as needed (Shen et al., 2012).

However, as the work undertaken by Penneman et al. (2016) suggests, binary translation still incurs an overhead of up to 5 times slower than virtualised in the worst case and only in the very best case is binary translation still 16% slower. Therefore, for computationally intensive parts of the application, such as the emulation of the graphics processing unit, it is suggested to find an alternative solution. Hydra, the solution presented in this research, forwards the instructions outside of the emulator to be executed on the native machine with minimal overhead (however, there are other approaches discussed in **Section 2.5**).

2.5 Emulating the GPU

There has been a considerable amount of work undertaken in order to effectively emulate the GPU and largely since 2014 it is considered a solved problem commercially (Li et al., 2014; Kazama and Miura, 2014). However, even though it can be emulated, it is very rarely emulated in a performant enough manner for games relating to digital preservation, let alone modern titles. Games often need to render at least 40 frames per second (FPS) in order to provide the user with the best experience (Claypool and Claypool, 2009; Claypool et al., 2006; Bernier, 2001) and emulation of the GPU can rarely achieve this. The complexity arises from the fact that the GPU is architectured in a relatively different way to a traditional CPU (Seiler et al., 2008). This was carried out in order to maximise rendering performance. These architectural differences include better parallel performance using micro threads and allow it to specialise in the task of processing many tasks in parallel, such as rasterising many triangles. However, this also makes it hard to emulate using a standard CPU because the CPU itself does not have such specialisations; as well as the issue of having to emulate each instruction rather than simply execute it. Seiler et al. (2009) have provided work in which they augment a number of standard x86 CPUs with a wide vector processor unit as well as fixed function blocks in order to make the CPU more appropriate for the task of rendering pipeline workloads. If this work can make its way into future CPUs, the task of emulating a GPU will be simplified, however, as it stands, off the shelf CPUs do not provide the ideal capabilities to emulate a GPU.

2.6 GPU Passthrough

The idea behind reduction of emulated components is well explored in many virtualisation platforms, in particular the GPU which deals with many performance critical workloads. Rather than emulate the GPU, the commands are forwarded to

the host operating system, where they can be natively processed on the host's physical GPU (Walters et al., 2014). Walters et al. (2014) have provided work into comparing the performance of this solution for a variety of commercial and open-source offerings. One important aspect of this is the note that due to hypervisor requirements, they were prevented from standardising on a single host or guest operating system. This finding almost exactly mirrors the same issues that other researchers have faced when dealing with emulators or virtualisers insofar that it is currently not yet possible to rely on a single piece of software for all emulation needs and, instead, flexibility is required to effectively utilise emulation as a migration strategy.

This is particularly important because the current state of implementing GPU passthrough is very platform-specific, which is fairly counteractive to this requirement of remaining flexible across platforms. The specific platforms required are also not suitable for digital preservation purposes because there is no evidence that older platforms will be maintained in the future. As a new platform comes out, the previous platform will be deemed legacy and subsequently dropped, destroying any potential there may have been for digital preservation.

But with a compromise in portability, performance is gained. One of the first commercial applications of GPU passthrough was in VMware's commercial offering VMware Fusion where even though it was still early, the results in a benchmark by Dowty and Sugerman (2009) were still very acceptable; between 18-42 frames per second were reported on games that were at the time seen as cutting edge. The game titles included Half-Life 2: Episode 2, Civilization 4 and Max Payne 2. Overhead (though not necessarily final performance) was recorded to be as low as just 2 times that of the native GPU.

Similar work undertaken by Shea and Liu (2013) also noted that once data is transferred from the VM to the host's native GPU there is very little overhead. Whilst context of their research is more about looking into the ways that games could be streamed from cloud infrastructure (a restricted collection of KVM and Xen

virtualisers), these findings are very important because the design of modern graphical APIs today favour the idea that data is retained on the GPU (and thus outside the VM). Their results show a performance of around 40 fps running Doom 3; which could be considered a fairly low performance by today's standards. However these results are slightly dated and also include the overhead of streaming the resultant rasterised image out across a socket. When running directly on the host rather than through a network, much more acceptable performance is likely.

Hong et al. (2014) reported similar results and also demonstrated a lower performance when the network was under the stress of multiple connected clients. Even though the cloud may well be an efficient way to monetise the future maintenance of old titles, it is not providing much of an aid when it comes to providing a single user the ability to run their old titles as effectively as possible. These results were again from the Xen virtualisation solution, which is a further example of the limited number of virtualisation platforms that support GPU passthrough.

2.7 Remote Virtual Graphics Systems and Streaming

Remote Graphics Systems are a common technology in order to facilitate multiple users sharing the same server or mainframe (Halperin et al., 2014). Virtual graphics systems are closely related and provide very powerful functionality and their use could potentially be extended for digital preservation purposes even though this remains fairly unexplored. The ideas presented by Hydra very much stem from the research in this area and by building upon this and also emulation technology, a solution to the GPU passthrough problem can start to emerge.

This architecture could be best described as a layer of indirection between the model and view. Rather than the data of an application being sent directly to the driver and thus the graphical hardware, it is stored within an intermediate layer; often a virtual

frame buffer. From here this data can be requested as needed by a connecting client. This architecture is discussed by Lok et al. (2002) and is demonstrated in **Figure 2.2**.

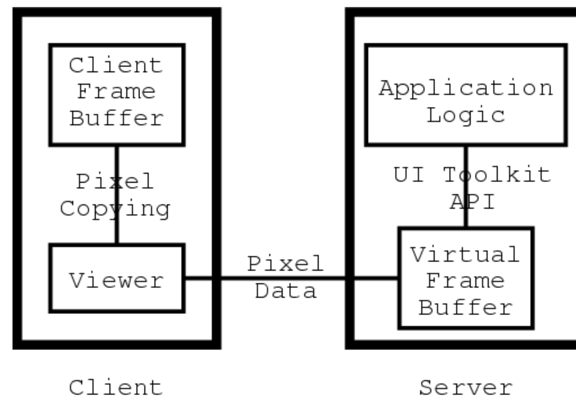

Figure 2.2: *The architecture of a virtual frame-buffer-based application (Lok et al., 2002)*

Due to the fact that the intermediate layer is not tied directly to the graphical hardware, it allows for the ability to have arbitrary screen sizes and, as such, these sizes are not tied to the graphics hardware.

Within the enterprise area, this is useful because it enables thin clients to connect, which can be served by much cheaper hardware (albeit less powerful) in order to save costs. However, this same strength also applies for digital preservation. Even though the thin client is no longer particularly thin; it may even be more powerful than the emulated platform. However, it is potentially a very different environment in terms of processor architecture, available libraries, graphics hardware, etc. that in order to port the software to it directly, many changes may be required. If treated as a thin client, only the system required to render the final output will need to be ported, which in some cases can be implemented with very few lines of code and thus is a much less intensive porting task (Baratto et al., 2005).

The most prominent example of this is using VNC on UNIX-like operating systems. VNC creates a new display server called Xvnc which mimics the X11 Window System server program called X. There are other types of these servers that provide slightly

different functionality. For example:

- **Xvfb** - Frame buffer server. No hardware acceleration
- **Xnest** - Embeddable server that can be run within another X11 session
- **Xephyr** - A more modern implementation of Xnest
- **Xrdp** - A server similar in architecture to Xvnc but for the Microsoft Remote Desktop Protocol (RDP).

Xvnc, in a similar way to Xvfb, can have an arbitrary desktop size specified with the **-geometry** option. This is possible because the server is not tied to the graphics hardware, unlike the standard Xorg server (**Figure 2.3**). This unfortunately is not available on other platforms such as Windows.

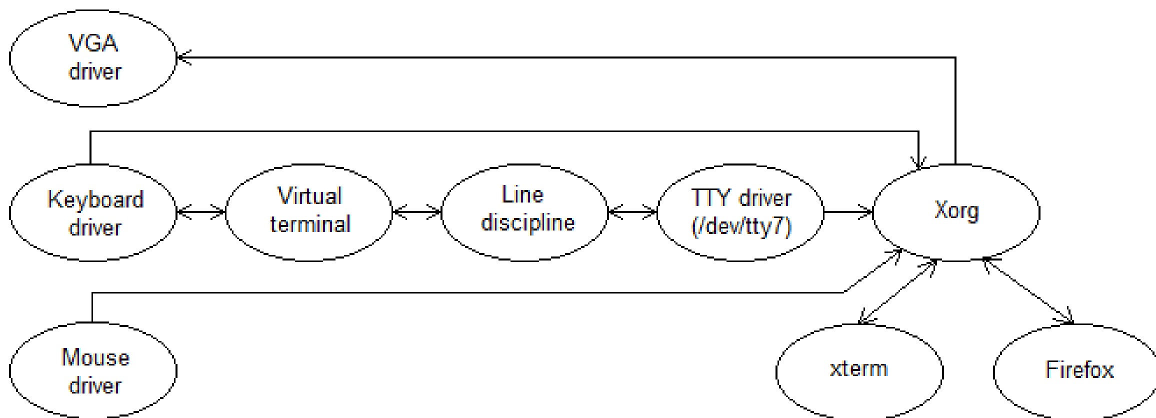

Figure 2.3: *Diagram showing how an X11 server can abstract the complexities of the underlying system drivers for GUI applications such as XTerm and Mozilla Firefox*

The X11 protocol can be run on platforms other than UNIX but this will not implicitly make the native software work on a virtual desktop. This is due to the large difference in architecture. In software written to utilise X11, rather than draw directly, it instead connects to the X11 server over a network socket and using a protocol it instructs the server what to draw, such as buttons, images etc. This protocol is complex and potentially very complicated to port to other platforms. Instead, VNC is used to

generate a rasterised image and sends that to the connecting client. In effect, VNC can only provide arbitrary desktop sizes when using a virtual desktop system such as X11 or Remote Desktop Protocol (RDP) underneath.

Whilst Windows does not provide the native X11 system, it does provide an alternative. RDP, as mentioned works on largely a similar way, using an intelligent but complex protocol. Originally developed by Citrix as a partnership with Microsoft under the product known as WinFrame (**Figure 2.4**), it aimed to add enterprise features into Windows which were lacking compared to commercial UNIX offerings at the time. Microsoft then provided this functionality in later server offerings starting with Windows NT 4.0 Terminal Server Edition. Pedersen and Perry (1998) discuss the actual implementation of RDP in significant detail. The protocol documentation has also been released by Microsoft Corporation (2013 (accessed February 9, 2018)). However, an interesting side effect of RDP is that it allows for the separation of the display buffer from the rest of the operating system in the form of user sessions. This is in fact all that is needed to use any desktop streaming technology. For example, multiple sessions can be started up but RDP never used; instead, a VNC server could be run on each session (using a different port) in order to provide a simpler and more portable protocol to the connecting clients running platforms in which an RDP capable client is not available.

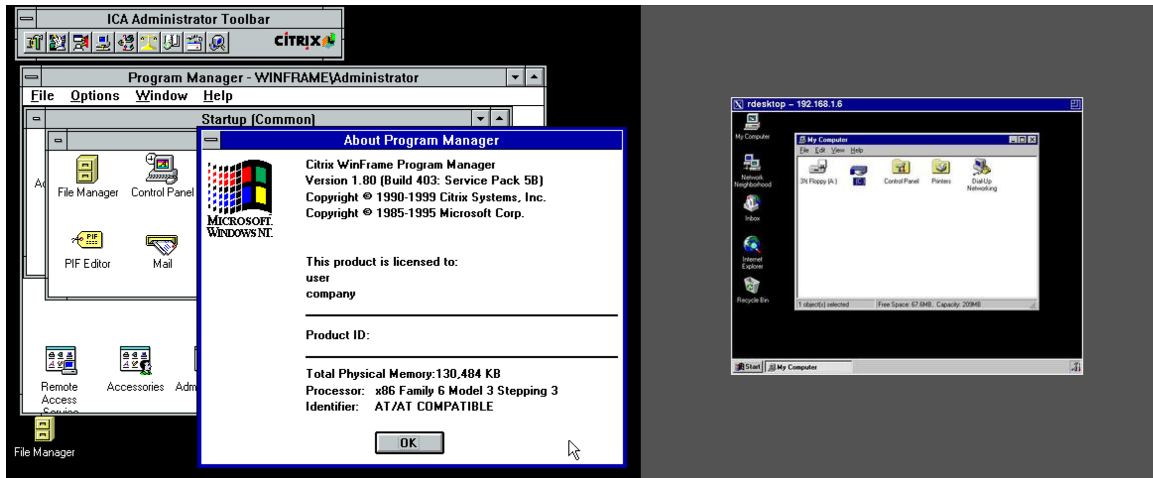

Figure 2.4: *Screenshot of Citrix WinFrame showing the strong relationship with Windows NT 3.x. Screenshot of a typical Windows NT 4.0 TSE session connected from a Linux terminal running the open-source rdesktop client.*

2.8 Ethical Considerations in Digital Preservation

Traditionally emulation has also been seen legally as a grey area (Downing, 2011), largely grouped within the same category as sharing via the Torrent network and other P2P services (Drachen et al., 2011).

The legal issues involved in digital preservation are catching up to the technology albeit at a much slower rate. The work by Muir (2004) is suggesting that a clarification of the legal ethics involving emulation within the UK is required. Wessels et al. (2014) in a more recent paper, citing the work undertaken by Muir (2004), have also demonstrated that a considerable amount of work is still required to facilitate some aspects of using emulation to enable digital preservation. Their main argument of the importance of these legal issues to be solved is in the area of open access. If a publisher or government publishes data to the public, in an obsolete or proprietary format that only specific software can read (with the aid of an emulator), the laws must potentially be modified to allow this process to work and for this data to be accessible by everybody.

If we are looking at digital preservation and ensuring that it is legally sound, one of the most likely routes to take is to tie it in with digital rights management and provide these digital artefacts as a cloud service or grid (Liebetraut et al., 2014; Innocenti et al., 2009). This solution however does not explicitly deal with preserving all resources and may likely end up focusing on non-free or copyrighted resources only. However, two services which are currently free to access and deal with free content and potentially copyrighted content in a non-commercial way (but also potentially include pirated material) is the Wayback Machine and the Internet Archive. Both of these services sometimes offer us the only solution available when it comes to accessing old data in a completely fair way without government or corporate influence or censorship (Thelwall and Vaughan, 2004).

A very interesting paper by Conley et al. (2003) looks at the legality of emulation from a piracy point of view. They go on to discuss the issue that it is perfectly reasonable to buy a Sony PlayStation 2 and use the in-built emulation layer in order to run games developed for the PlayStation 1 and yet running these games on something different such as a newer PlayStation 3 and a third party emulator could be deemed as piracy. This is a demonstration of the grey area and ad-hoc nature of the legal issues presented by digital preservation. However, dealing with these issues is out of the scope for the research presented in this thesis. It is also suggested by van der Hoeven et al. (2010) that legal issues pertaining to digital preservation do not tend to present themselves for a number of reasons primarily relating to the software becoming abandoned. This "abandonware" either has no commercial vendor prepared to protect the copyright or the vendor simply no longer exists due to bankruptcy etc. Possibly if the digital preservation of abandonware becomes a larger market, these companies will reappear and attempt to monetise the software again.

2.9 Cloud Solutions to Digital Preservation

Oltmans et al. (2004) have seen a requirement to reduce dependencies on platform requirements without the need for emulation. An example output that they have devised consists of simplifying the data to be preserved, such as their Preservation Processor converting PDFs into JPEGs, which are an easier technology to work with if the PDF file format ever became unmaintained. This has the general issue related to the fact that with simplicity comes the lower denomination of functionality. For example, PDFs provide searchable text whereas JPEG is a rasterised compressed image format without text searching capabilities. Whilst this is of fairly limited application for games and software preservation in general, it does suggest some merit in reducing complexity of file format, especially if it requires specific software to access. An example possibly more related to games is reducing dependence on proprietary vendor middleware such as audio, rendering or physics. A fairly known example of where a port of a Windows game to Linux was blocked due to proprietary middleware was Epic Software's attempt at porting Unreal Engine 3. The PhysX middleware library (then owned by Ageia) was chosen to calculate physics collisions. It worked by offloading the physics calculations to a dedicated maths co-processor (nowadays the GPU) as shown in **Figure 2.5**. However, it supported only a very limited number of platforms and did not support open platforms like Linux. What further exasperated this situation was that Ageia either did not have the relevant skill set within house to make a Linux port or it was not seen as commercially viable for them. The middleware was also proprietary and the source code was withheld and, as such, Linux developers and Epic were unable to make the required fixes. Later, PhysX was acquired by NVIDIA and subsequently open-sourced, however if this was not done, many games would never be portable to anything other than Windows Vista/7 because of their dependence on this middleware.

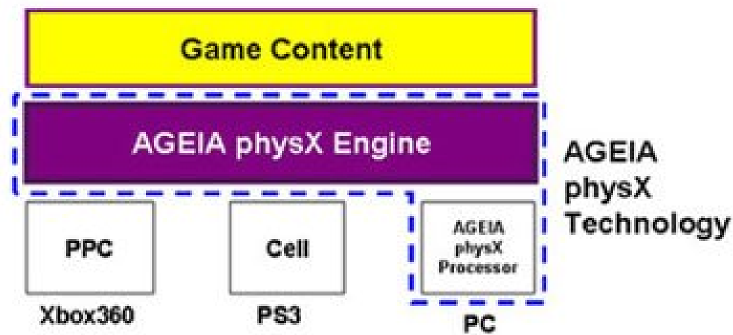

Figure 2.5: *The technology stack of PhysX. Note the fairly limited range of hardware and the PC platform limitation to Windows. The use of a standard GPU has greatly increased potential hardware support in recent years*

Situations such as this demonstrate that middleware or hardware is not always available to the user. This can be due to the operating system and platform they are running. The current trend in solutions to this problem is offloading anything potentially awkward to set up by casual users to a remote service, often ran by a company and monetised. This is currently referred to as The Cloud (Buyya et al., 2008; Li et al., 2013). Emulation is no different and research has been undertaken to see how this may be feasible (Rechert et al., 2010). Rechert et al. (2010) propose an abstract architecture to run a wide range of emulation infrastructure as remote services. The main issues inherent to this is that the emulators still need to be ported in order for them to work. Whilst the number of platforms that the software needs porting to is greatly reduced (because they can simply receive the stream), this is just delaying the inevitable issue that one day in the future, this emulation software will no longer work on existing operating systems and hardware. It will either need porting or a rewrite if the programming language's execution runtime is no longer available.

Another issue is that some platforms will still not have a suitable emulator available to them. For example, SGI IRIX is seemingly a very complex platform to obtain a working emulator for. Either this is due to the requirement on an accelerated GPU

to render the desktop or there is simply not enough commercial value or manpower to create such software. The emulation infrastructure discussed above is still reliant on the underlying (often hobbyist) emulators.

It can already been seen in the Dioscuri emulator that although it was originally designed with lifespan in mind that there are certain platforms which it cannot run on. For example mobile (Android, iOS) or web platforms. This is mostly due to technical and licensing limitations with Java with which it was originally chosen to be implemented with. For this reason, it was decided to develop a remote access system utilising the VNC protocol to avoid this limitation (Genev, 2010). Whilst this makes it suitable for the cloud and remote streaming, this is not seen as a viable solution in the long term because the cloud itself is built on physical hardware and low level software and operating systems. As soon as Java is unable to be run on current configurations, this form of digital preservation will begin to fail. There is no guarantee that servers making up the cloud of the future will be running an operating system or computer architecture capable of supporting a specific version of the Java VM (Blem et al., 2013; Tso et al., 2013).

For a considerable amount of time the server market has been dominated by the Intel x86 processor architecture (Gawer and Cusumano, 2002). However, it appears that other architectures such as ARM are looking to also become popular (Guan and Gu, 2010) in the server space. Whilst ARM processors can also use virtualisation (Varanasi and Heiser, 2011), they cannot virtualise x86 instructions and vice versa. This means that a deeper look into digital preservation techniques might need to take place rather than relying on the "quick and easy" solution of x86 virtualisation. Otherwise, commercial companies might find themselves tied to ageing or insecure hardware in order to keep their legacy application(s) running.

The ecosystem may become more fragmented still if custom architectures start becoming common. Currently, it is the norm for a small number of vendors to provide processors using their own proprietary architecture designs. However, if an

open architecture such as RISC-V appears, it opens up the ability for many different hardware companies to provide their own subtly different architectures (Asanović and Patterson, 2014). At this point, binaries compiled for one processor design may not necessarily run on the next. Potentially the only way to migrate programs between different vendors would be using source code and building a binary as needed. At the moment the trend is to keep code proprietary but this may have to change if there are simply too many subtly different architectures to provide individual binaries for. Of course, code can still be obfuscated to protect intellectual property as a compromise whilst still retaining a high portability and even now this is still a common practice for this reason (Collberg, 2018).

Possibly the biggest issue inherent in relying on Cloud computing for digital preservation, as explored by Ford (2012), is that there is no way to get access to a full snapshot of data required for preservation. Instead, data is streamed or offered in chunks meaning that when the Cloud provider disappears, so do these full snapshots of data. Therefore, streaming technology (previously discussed in **Section 2.7**) looks set to be able to delay the inevitable loss of digital heritage but can not quite prevent it indefinitely.

2.10 The Digital Preservation of Computer Games

Unlike utilising software in order to achieve an output file or converting documents into a format easier to preserve, the complicated with preserving games is that the software itself is what needs to be preserved rather than the functionality or the output file. This is also particularly difficult because not only does the software need to be run to be enjoyable but it also needs to be run in a way that the original characteristics are preserved, such as screen size and frame rate, as discussed in the work undertaken by Pinchbeck et al. (2009), where they describe the specific issues that they encountered in games preservation and these are often agreed upon by a

variety of other sources (Swalwell, 2009; Guttenbrunner et al., 2010). In particular, Guttenbrunner et al. (2008) have provided a clear requirements diagram (**Figure 2.6**) tailored towards video games.

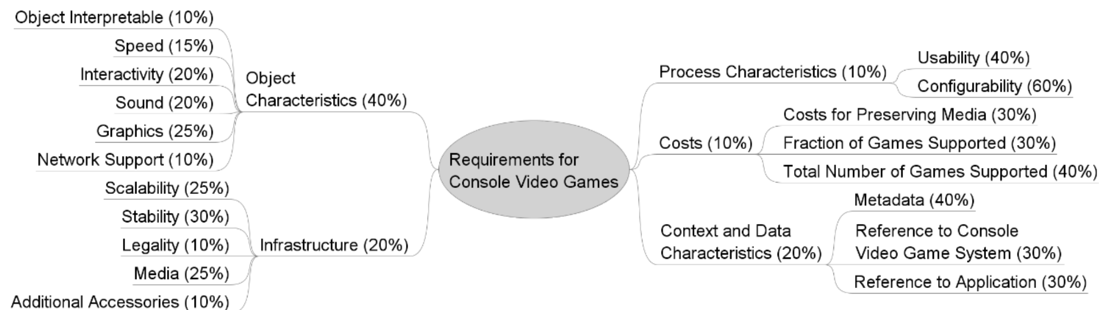

Figure 2.6: *Requirements tree for console video games with importance factors (first two levels only) (Guttenbrunner et al., 2008) (Licensed under CC BY 4.0)*

One interesting characteristic stated by Guttenbrunner's requirements diagram is the Network Support. This requirement is generally missing in older sources but with the current trend of games today becoming online-only, this presents another problem for the preservation; not only must the complexities of emulating the game be overcome but it presents other external dependencies to potentially retired servers (Winget, 2011). Winget (2011) discuss this issue in detail, however it seems there is very little in terms of a solution to this problem. The companies behind these games rarely intend to provide the server software required to host the game, sometimes due to protection of their IP but also as part of their planned obsolescence strategies. This presents a very similar problem to the issues caused by streaming services (discussed further in **Section 2.7**). One solution is to develop a server emulator to provide the required communication back to the required software to authorise it. However this is also deemed outside the scope of this research and presents more of a legal issue which may become more explored in the future by researchers with expertise in this specific area.

2.11 Evaluation

It can be seen in the literature covered in the thesis so far that emulation is seemingly the most reliable solution when it comes to ensuring digital artefacts can remain usable. However, there are a number of issues that need to be understood. The main one is that flexibility must be maintained.

It is not yet possible to rely on the fact that a single platform or product, be it an emulator or virtualiser, is enough to be able to target every platform we may want to support. Any solution must be able to migrate between emulation tools. There have been a number of ways of specifying emulation requirements in order to help facilitate this. Those requirements specifications presented by Rothenberg (1999) and Jamraj et al. (2017) will likely remain relevant much longer than any individual emulation system can be maintained for and these specifications will then facilitate the procurement of a replacement emulator.

What these specifications lack a solution for and what is very rarely discussed in academic literature is what is to occur if there is no alternative emulator that can successfully emulate a platform. For example, it is extremely hard to obtain an emulator capable of correctly providing an environment suitable of sustaining the IRIX operating system. Potentially the reason for this is because the architecture chosen for that platform was MIPS (in particular the 64-bit MIPS R8000) based rather than the ubiquitous x86, where there are a number of emulators available. However, this is not an entirely satisfactory answer because in the context of digital preservation, it is not unfeasible that x86 will be replaced by something different (perhaps even a modern MIPS) and emulators for the platform will diminish in numbers.

The cloud is not a solution to this problem other than perhaps providing a linear performance increase for CPU processing. Any technologies that would fail to facilitate digital preservation on a standard PC, such as a laptop, will also be very

likely to fail if simply hosted on a cloud server instead.

This research indicates that it may become crucial in the future to be able to utilise a potentially small number of available emulators effectively because there is no guarantee that there will be an abundance to choose from. Any work undertaken to ensure that 3D games can be run effectively on them should remain platform agnostic, not just in terms of the guest operating system but also when it comes to the actual emulation platform chosen, because there may very well end up being a small number of alternatives.

Emulating games via the use of GPU passthrough exists and is tested but only on a very limited number of platforms which makes it unsuitable for digital preservation purposes. The current technique used to provide GPU passthrough often involves augmenting the emulator itself to handle specific calls. Subsequently, a small platform specific driver is provided to the limited number of guest operating systems to allow them to connect. Again, they connect in a very specific way to a specific emulated virtual device. This technique appears not to be generic enough to provide the flexibility needed to be able to migrate between different emulators effectively.

2.12 Additional Techniques to Preserve Software

Though the literature commonly directs towards emulation as the most feasible solution for digital preservation, there are a variety of techniques other than emulation that either help facilitate the maintenance of software and porting it to another platform or allow the target software to be run unaltered on alternative platforms. These techniques do not exist for the sole task of digital preservation directly but can likely be used to achieve a similar goal. In this section, these different techniques will be explored and subsequently be evaluated for their effectiveness when used to facilitate the porting of a simple 3D program in **Section 2.12.3**.

2.12.1 API Cloning

API cloning consists of replacing a dependent library with another that mimics the same interface. This allows the original software codebase to simply link to this new library and remain unaltered. API cloning can often be seen as the most "native" way of porting software and can yield satisfactory results in terms of reducing the number of bugs introduced due to new development. This is because the original codebase remains unchanged, so the only part that needs testing is the new cloned API. The typical usage of API cloning is if a target platform lacks a certain library so the cloned API effectively just points towards the target platform's alternative.

Google had decided to clone Oracle's Java standard library API for their own Dalvik system on Android (Ehringer, 2010). It did this in order to allow a large amount of existing Java code to work unaltered on Android. This was particularly important during the platform's infancy when Google needed to make targeting the Android platform as desirable as possible for early adopters. This technique is often known as "creating a drop-in replacement" (Henning, 2009).

Another example of API cloning is a solution to replace Creative Lab's OpenAL. Since a potential target platform may not not have an OpenAL compatible sound card hardware, it would be unable to use the official OpenAL API. Instead an API clone called OpenAL Soft can be used which is an entirely software-based implementation which rather than utilising the hardware specific calls, simply delegates the audio processing to the target platform's available API. In many cases, it will be DirectSound on Microsoft Windows, ALSA (Advanced Linux Sound Architecture) on Linux and OSS (Open Sound System) on FreeBSD.

OpenAL Soft is not the only API clone of OpenAL. The Emscripten C/C++ to ASM.js compiler provides an OpenAL API which uses HTML5's audio system underneath. An application compiled for the web using OpenAL will end up utilising the HTML5 audio as a back-end whilst the API remains identical to OpenAL, allowing

the code to compile and run unchanged.

OpenAL Soft and Emscripten are well utilised and fairly common technologies when porting games with few issues encountered these days whilst using them. However, neither Emscripten or OpenAL Soft are trivial pieces of code and are not something that could have been written within the time frame of most projects. OpenAL Soft can also only use a number of native audio backends (DirectSound, ALSA and OSS). It does not support Android's OpenSL or Plan 9's audio objects.

Another example of an API clone is Regal. Among others, it provides a clone of the OpenGL 2.0 fixed function pipeline. This is for parts of the API that were deprecated or removed in the OpenGL 4.0 Core profile such as *glBegin()*, *glEnable(GL_TEXTURE_2D)* etc. These functions are emulated using the underlying OpenGL 4.0 Core functionality. It works in a very similar way to Google's Angle project but instead of using OpenGL to emulate the functionality, it uses DirectX to provide the OpenGL functionality (this can be seen in **Figure 2.7**). Finally, Emscripten, like OpenAL, also provides an API clone of OpenGL.

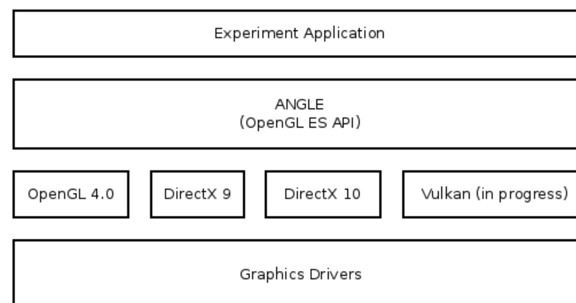

Figure 2.7: *The technology stack based on using the Google Angle middleware.*

Between Angle, Regal and Emscripten, a program could be compiled and ran on a number of platforms with an underlying support for a GPU with minimal changes to the codebase. However, as can be seen in **Figure 2.8**, the technology stack is starting to become at this stage quite complex. This can potentially start to become a source of bugs and other complexities.

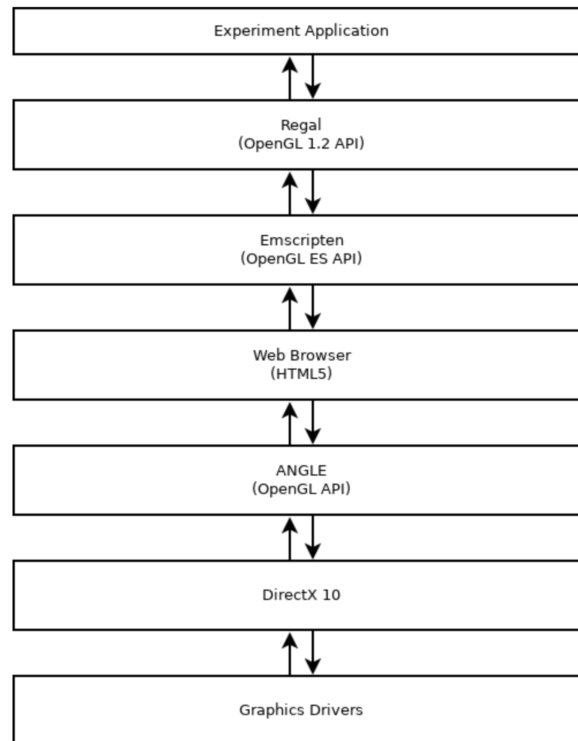

Figure 2.8: *The technology stack when requiring OpenGL 1.2 compatibility on the HTML5 platform*

As with OpenAL Soft, the Regal and Angle codebases are large and must cover a significant amount of conditional / platform specific compilation to be useful. This is not a trivial task. On platforms which are less open and flexible such as the Microsoft WinStore platform, Regal is still potentially limited in use because there is no OpenGL implementation available on that platform. Instead, Microsoft has dictated that only DirectX is to be used by app developers. Likewise, if the underlying platform was Vulkan or Metal, GLVK or Molten would have to be used instead. This large number of dependencies will start to convolute the build system and greatly increase testing requirements.

In recent years, especially with the advent of much lower level platforms such as Metal and Vulkan, it is possible to create more complex stacks of technology to help provide a consistent environment. An example of this can be seen in **Figure 2.9**.

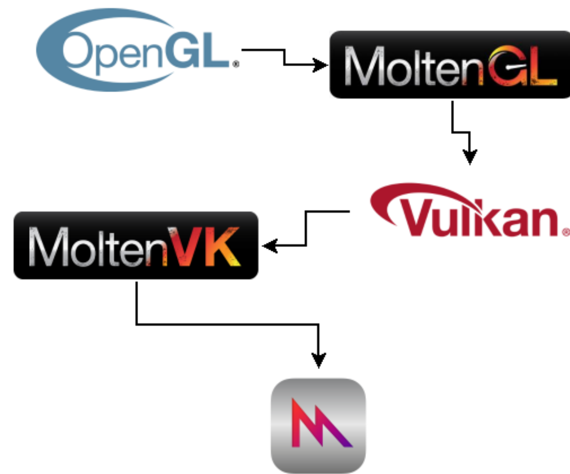

Figure 2.9: *A potential API stack required to support OpenGL on macOS since OpenGL was deprecated in version 10.14*

As a developer it is fairly straightforward to see that this complex hierarchy of dependencies could potentially be quite fragile. None of these compatibility layers are as well tested or supported as the official APIs that they intend to mimic. The build system required to build such a project utilising this architecture could also potentially be very convoluted, which may lead to future maintenance issues even outside the main application code.

There are a number of issues with this approach at maintaining software however, with some of these issues described in a study undertaken by Hsieh et al. (2013), where they attempt to port an existing Java program to Android's clone of Oracle's Java reference API. A number of subtle differences in the API implementation had meant that changes to their code were required to support both platforms. There are also a number of facilities that could potentially be missing, such as the standard Java Swing GUI library. On Android there are no alternative classes because the Android system uses a different UI paradigm and no adapter classes currently exist. Mostly, API cloning appears to be a fairly reliable way to create a compatibility between different platforms. There are potential issues but in many cases these are due to an incomplete implementation and can be overcome in time. For this reason

it was decided that Hydra, the tool developed as part of the research presented in this thesis (discussed further in Chapter 3) would leverage the benefits gained from cloning the OpenGL API to create a drop-in replacement as a graphics library, whilst providing the additional benefits of future maintenance and integration with Virtual Machines.

2.12.2 Compatibility Layer

Compatibility layers can be thought of as a large collection of API clones in the way that if POSIX / UNIX functionality is required to be used on Windows such as *opendir()*, then by compiling and running the code using a compatibility layer such as Red Hat's Cygwin (Noer, 1998) will redirect the underlying calls to the Window API such as the *FindFirstFile()* family of functions (Racine, 2000). Where compatibility layers differ is that further up the system, less and less functionality needs to be redirected to the native platforms alternative because the underlying functionality is already implemented within the compatibility layer itself. This helps reduce the amount of code required to create the platform compatibility layer (though this is much larger than a single cloned API). This is demonstrated in **Figure 2.10**.

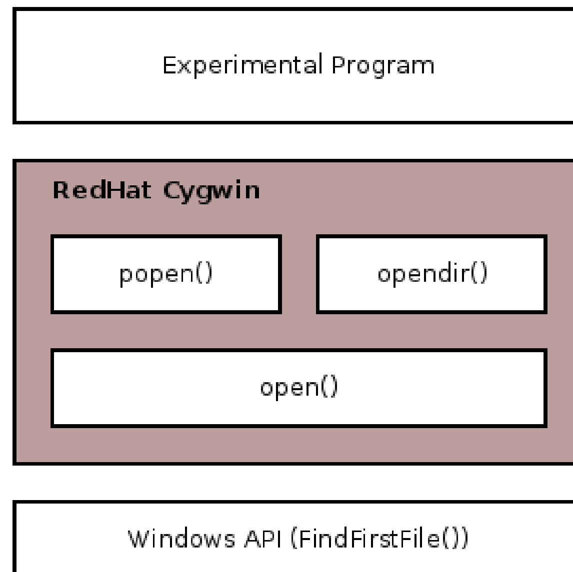

Figure 2.10: *A simplified view of responsibilities provided by the RedHat Cygwin compatibility layer*

Cygwin, as mentioned, is one example but others are available from Microsoft directly. They consist of Microsoft Services for Unix, which again is a POSIX compatibility layer, deprecated after Windows 7. The recent iteration of this technology is Windows Subsystem for Linux (WSL), which does not quite provide a full POSIX environment but is fairly close to that since Linux is a clone of UNIX. Unlike the other two, binaries compiled for Ubuntu Linux actually run directly on WSL rather than having to be re-compiled to take advantage of these unique platforms.

Moving away from the Windows platform, the Wine (Wine Is Not an Emulator) platform compatibility layer is tested on Linux and FreeBSD. This software effectively does the reverse of WSL, in so far that the Microsoft Windows environment is provided to the Linux or FreeBSD host so that executables compiled for Microsoft Windows can call into their dependent APIs such as DirectX, MFC, Winsock, etc.

Using a unique kernel interface *linux(4)*, FreeBSD can run Linux executables in a *chroot(8)* like environment. This is extremely useful in order to get certain closed source gaming titles working on FreeBSD, since it is often overlooked by game developers as a supported platform compared to Linux and Windows. It is useful

to note that FreeBSD is well known to be the foundation for CellOS and Orbis OS; the operating system for the Playstation 3 and 4 respectively. Considering the success of these platforms and the wide variety and number of games, it is fairly disappointing if not surprising that FreeBSD, as the upstream operating system, has virtually zero commercial titles available for it natively.

Finally, on a more exotic operating system, Plan 9 is able to utilise POSIX functionality through the use of the ANSI POSIX Environment (APE) layer. This was developed at Bell Labs with an understanding that even though Plan 9 was meant to be the successor to UNIX, in practice, UNIX had already penetrated so much of the industry that to drop compatibility with it would have been a large disadvantage.

What has been noticed in recent years has been a reduction of the number of compatibility layers. There are a number of reasons for this but with the increasingly perilous environment provided to us by an increasingly connected infrastructure, one of the major concerns is security (Yegulalp, 2016 (accessed July 9, 2019)). Compatibility layers are often poorly maintained and, as such, can provide an increased attack vector. In particular, OpenBSD (a distant relative of FreeBSD focusing on security and code correctness) has removed the Linux compatibility layer from its kernel since version 6.0 citing security concerns. This is particularly important to note for digital preservation because even though compatibility layers exist now, they possibly only extend the ability to run older software for a decade or so, i.e. they do not provide an indefinite solution.

2.12.3 Comparison of the Techniques

In order to evaluate the effectiveness of the described porting techniques, a simple 3D application was developed to serve as a test bed (**Figure 2.11**). The application was developed on Linux (Debian 8) on an Intel Xeon 64-bit processor. This means that Linux x86_64 is the control platform. The application was developed in a naive way in that no real consideration was made with regards to portability. The application has

two dependencies, OpenGL 2.0 and FreeGLUT, because those are two fairly standard technologies used on Linux to create 3D applications.

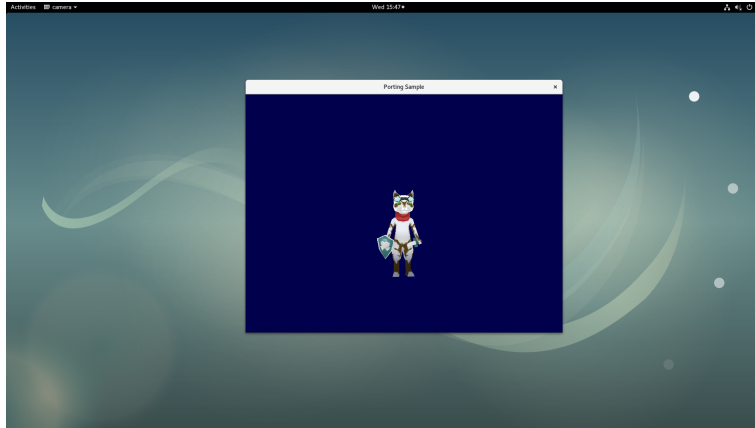

Figure 2.11: *A very simple 3D program showing a rotating model running on Linux (Debian GNU/Linux 9)*

This simple 3D application is first to be ported manually to a number of different platforms. The parts of the software code and dependencies that are not valid for the given platforms will be rewritten and replaced. The time taken to do so (relative to the original development) was recorded.

This application is then to be ported again to those same platforms but, instead, using different porting techniques. The times will again be recorded (relative to the manual port) for the process and compared against the times for the manual port.

The platforms chosen to port the software to consist of the following:

1. **Microsoft Windows 10** - To test porting to a fairly non-POSIX compliant operating system

2. **Microsoft Windows NT 4.0** - To test porting to an older revision of Windows for backwards / forwards compatibility

3. **Microsoft Universal Windows Platform (UWP)** - To test porting to a largely locked down, restrictive and kiosk-type environment

4. **Linux (Debian GNU/Linux 9) ARM** - To test porting to an almost identical environment but entirely different CPU architecture

5. **FreeBSD 10** - To test porting to an alternative POSIX compliant operating system

6. **Plan 9** - To test porting to an older platform using a POSIX compatibility layer

7. **Android** - to test porting to a locked down (similar to UWP) but POSIX compliant operating system

8. **HTML 5** - to test porting to a platform that is widely accepted as "the universal platform".

It should be noted that due to the age of Windows NT 4.0 and Plan 9, both of these platforms will be run within an x86 emulator (QEMU) during the experiment.

Analysis of Manual Porting Process

Figure 2.12 shows the initial time required to port the experiment program from a standard 64-bit Linux to a variety of other platforms. The time taken to complete the process shows a good (albeit rough) estimate of the feasibility, if projected on a much larger task.

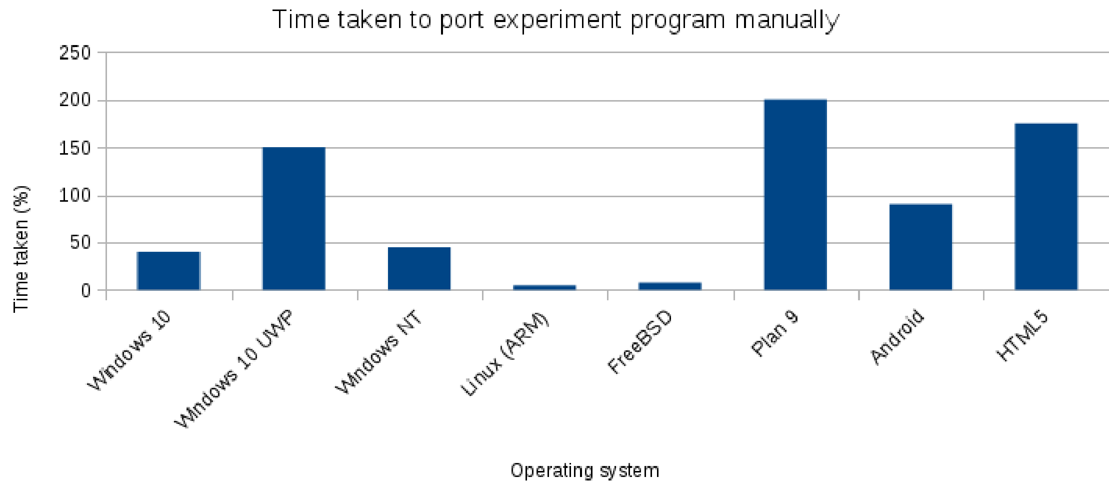

Figure 2.12: *Graph showing the time required to manually port the experiment program to different operating systems. Time is shown as percentage compared to time taken for original implementation.*

These results are interesting in that they show that it is largely trivial to port a high level program such as a game from Intel x86_64 to ARM. Contrary to the complexities of the Android SDK or Windows Store toolchains, the actual process of compiling the same C/C++ code into machine code for either the x86, ARM, MIPS, etc. is very straight forward. A number of sources state that platforms such as Windows RT fail due to the use of an ARM processor rather than x86. However, other sources disagree (Iqbal et al., 2014; Xing et al., 2015) and suggest that it is instead due to the locked down and restrictive nature of the platform with inflexible toolchains and policies. Once a device has been jailbroken and a suitable C++ compiler has been procured, the platform remains very useful.
This is further evidenced by Microsoft's later iteration of Windows on ARM (Windows 10), where not only can executables be compiled to native ARM32 and ARM64 rather than enforce the restrictive Universal Windows App Platform (UWP) but they are also providing initial binary translation to support Intel x86 architecture binaries. This support can be seen in **Figure 2.13** and it demonstrates the less restrictive

approach that Microsoft has taken for Windows 10 on ARM.

App Type	Intel 32-bit	Intel 64-bit	ARM 32-bit	ARM 64-bit
Win32 (native)	Yes	No	N/A	N/A
Desktop Bridge (native)	Yes	No	N/A	N/A
Win32 and Desktop Bridge .NET	Yes	No	N/A	N/A
UWP	Yes	No	Yes	N/A

Figure 2.13: *Table showing the executable types supported by Windows 10 on ARM (Microsoft Corporation, 2019)*

The time required to port the software from Linux to FreeBSD was also only slightly higher. This is because FreeBSD has a very similar userland environment to Linux in terms of the build tools, platform libraries and functionality. Again, this is because both operating systems adhere to the POSIX standard which guarantees some similarities.

Next, Windows 10 and Windows NT 4.0 are largely matched in terms of porting time. Much of the work done was replacing the POSIX related code with the Win32 API equivalent, which in many cases was very similar between one another. This is interesting because even though the timeframe between the release of Windows NT 4.0 and Windows 10 (1996 and 2015 respectively) is relatively large, there is very little change between the two with regards to their underlying APIs.

Windows 10 UWP is a very different matter. This was a complex port in terms of restrictions and limitations put on the programmer. There were language changes between standard C++ and Microsoft C++/cx meaning that much of the core code required replacing. There was little access to the Windows API and the replacement functionality was, instead, provided by a fairly foreign environment. The core difficulty was that OpenGL was disabled in this profile and, instead, a large rewrite of the code to utilise Microsoft Direct X was required. Interestingly, this port took longer to complete than the original program written from scratch and resulted in a larger number of lines.

Similar difficulties were expected with the port of the program to Android, however, due to the fact that OpenGL was allowed and standard C++ could be used the port turned out to be more straightforward than the UWP platform. That said, even though Android is based on the POSIX standard (in this case it is Linux (ARM)), the standard filesystem could not be used due to security sandbox reasons. There was also a heavy focus on Java as the premier programming platform which meant that additional steps were required to call into and utilise C++ (in order for correct code reuse) via the Java Native Interface (JNI).

The manual port to HTML5 was time consuming mostly because the C++ code had to be converted to JavaScript. This meant a rewrite. HTML5 provided the required graphical capabilities (via WebGL) and the filesystem capabilities. There was also minor refactors that had to be undertaken to avoid the design that the web browser does not allow "blocking" in code. In order to wait until an image was loaded, a more asynchronous approach had to be taken which does not work particularly well for games (as they follow a more "real-time system" based architecture).

The port to Plan 9 was the most complicated one (**Figure 2.14**). Whilst it did provide a POSIX layer and a standard C compiler, it did not provide an OpenGL API. This meant that a simple software renderer had to be written from scratch and integrated. Had the experiment program been written to use a software engine in the first place, much of this would have been avoided (at the expense of run-time performance) and the graphics could have been handled in a largely standard way. Whilst the renderer was the most significant complexity, the handling of assets from the filesystem also required a large amount of code to be modified. Even though the standard C library was available and in particular the *fopen()* and *fclose()* functions were present, the location of the files was non-trivial. The reason for this is that Plan 9 uses the idea of namespaces for the filesystem (Pike et al., 1992; Welch, 1994). This is a radical departure from the standard setup seen in POSIX/UNIX-like operating systems and also Windows and is largely due to the distributed nature of the Plan 9

operating system (Presotto et al., 1991). Whilst it is unlikely that in the future we will be using this particular operating system or filesystem, it is certainly possible that our existing filesystems will change in a similar manner to better facilitate modern trends in computing, such as cloud or grid architectures (Shafer et al., 2010).

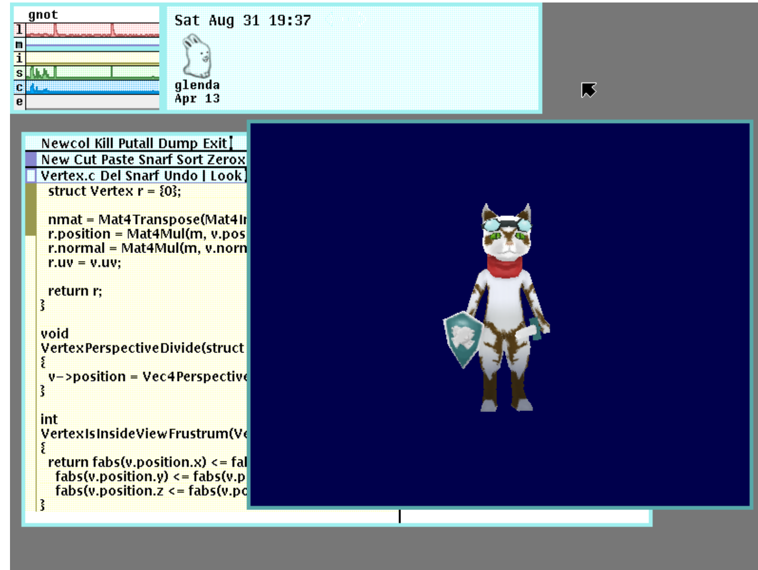

Figure 2.14: *The final result of the port to Plan 9. It required the implementation of a software based 3D renderer (written in ANSI C) which was not only time consuming but also fairly resource intensive (especially in an emulator).*

Alternative ports to these platforms were attempted using a number of existing approaches. **Figure 2.15** shows the percentage of time saved using these techniques. It should be noted that a number of techniques could be used at the same time to potentially port software faster, however, this was not explored in this simple pilot study.

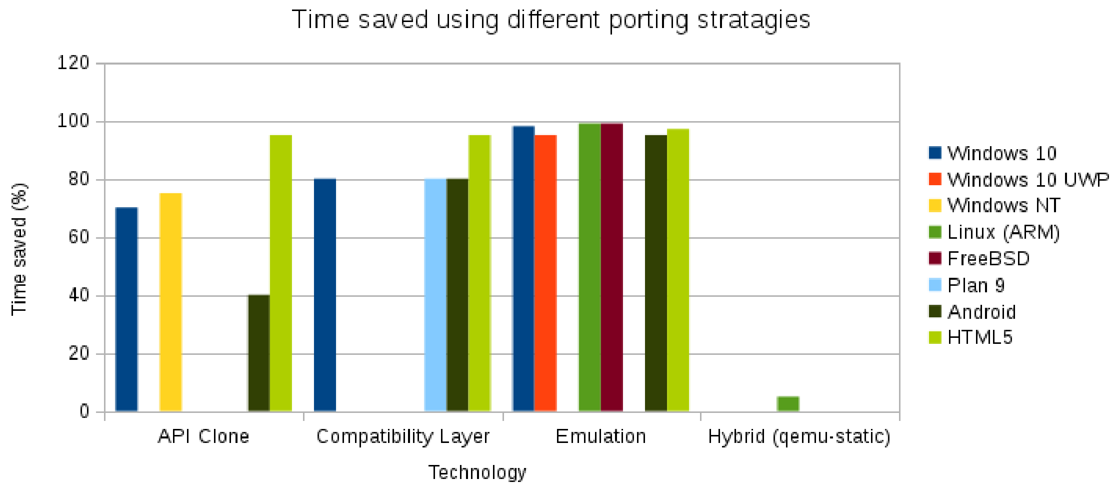

Figure 2.15: *Graph showing the relative time saved when using the respective technique compared to manually porting the software*

Analysis of Porting Techniques

The results show that emulation can be the best performing time saver when it comes to porting software to a new platform. The amount of time saved is consistently above 95%. The only platforms where it has not been useful are the ones which are already old enough to themselves need to run within an x86 emulator.

The API layers and compatibility layers have also shown to be useful but perhaps not quite as consistently as emulation. This is largely due to the amount of work needed to bind the layers to the underlying technologies. Emulation may at first seem like the most amount of work but if written in ANSI C (as many of them are for performance reasons), the same emulation code can be used on almost all platforms (Dolenc et al., 2000; Thompson, 1990).

Critically, it should be noted than many of these techniques were only useful for a restricted number of platforms and use-cases. **Figure 2.16** shows a per-platform overview of the numbers of porting techniques that were viable. Common reasons for specific techniques not working were due to either highly restrictive environments such as Microsoft's UWP, where portability aids were not allowed to run, or much

older environments where these portability layers or tools were not themselves ported to or, finally, in the case of OpenGL, entirely lacking the hardware support.

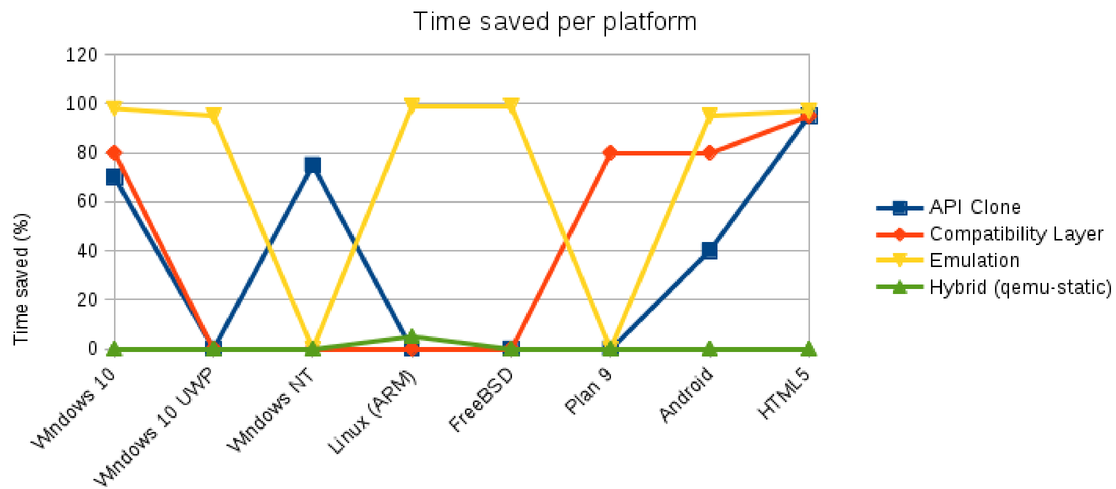

Figure 2.16: *Graph showing the relative time saved per platform using the respective technique. Windows NT 4.0 and Plan 9 were already running in an emulator, explaining the less consistent emulation results*

It can be seen in this brief pilot study of available techniques that emulation scores highly as a way of running software on alternative platforms. It is due to this reason that emulation is currently one of the best ways of running older software; on occasion it is in fact one of the only choices currently available. The focus of this thesis revolves around emulation as the main portability strategy. Emulation as a solution is not without limitations however. Emulation has allowed the software to run but the performance was not acceptable in many use-cases. For example, it had too low a frame-rate for gaming to be possible because all of the rendering was performed by software, rather than utilising the performance of the host's graphics card. Work has been undertaken to attempt to solve this but, again, there are a number of issues presented which this research aims to solve.

2.13 Conclusion

From the conducted literature review, there is clear evidence that there is an interest in preserving software and that a fair amount of work has already been undertaken by both commercial entities as well as hobbyist communities. There are a number of different solutions ranging from compatibility layers to full platform emulation. Many of which were evaluated in the brief study. In many ways the industry has solved, at least temporarily, a number of these issues relating to future maintenance. However there are areas which are still unresolved which this work intends to address. These areas are as follows:

- **Maintenance of software on alternative architectures** poses one of the largest challenges to digital preservation. Currently the ability to virtualise on the popular AMD64 architecture (and to a lesser extent AArch64) has been suggested to at least provide a temporary solution in that these hardware facilities on the processors allow for the efficient forwarding of of instructions to be executed. However for platforms utilising architectures without virtualisation, there has been much less success at solving this. There are a number of platforms and software that simply cannot be run effectively without the original hardware.

- **Maintenance of software on aging architectures** is still a long running problem. Virtualisation on a specific architecture can only work whilst these physical architectures are available. If the industry ever moves on to another type of architecture, the processors capable of such virtualisation will start to become less common and potentially more expensive. The literature does suggest that translation layers will be available, however, these are often unportable, complex to maintain and not readily available. The literature does suggest that full system emulation is a more sustainable solution but there

are issues with performance because the instructions need to be fully emulated rather than simply forwarded through to the processor.

- **Full system emulation** is generally agreed to be too computationally expensive for games and simulation software. This is a specific area that this research intends to address. By leveraging full system emulation for the majority of the program but only forwarding out the expensive graphical rendering, portability and performance can be achieved.

- **Portable programming languages** are often cited as the solution to easily migrate software across to different hardware as the industry evolves. However there is some amount of disagreement here as to the feasibility of this. Particularly now as we are starting to see technologies such as the Java platform age, with it we are seeing deprecations and loss of functionality which almost mirrors the same losses within hardware. Ultimately porting these technologies across to newer hardware is becoming a bottleneck to portability. This research intends to investigate this specific issue of portable programming languages and look into ways that ANSI C could potentially be improved as a conduit for portable software, even compared to many modern alternatives.

- **Accessing native hardware from within emulated environments** has been addressed in much of the research covered. However what seems to remain largely unexplored is how this can be done in a generic manner. For example, one that does not require bespoke code for each host and guest platform being utilised. This is again fairly important because whilst a commercial company might have the resources to achieve this, individuals will not and for a large proportion of digital preservation, much of it is pioneered by these individuals rather than as a companies business direction. This research provided aims to solve this by exploring the use of ethernet as the standard communication medium between the guest and host. In particular, attempt to solve any issues

that typically lead to an alternative approach being used.

Chapter 3

Methodology - GPU Passthrough with Hydra

3.1 Introduction to Hydra

3.1.1 What is Hydra?

Hydra is an approach that intends to solve a number of issues which currently block the portability of software to a wide range of platforms. This approach intends to leverage the existing technique of emulation but allow for a number of critical extensions that currently limit the use of emulators for games.

From a technical viewpoint, discussed further in Section 3.2, Hydra is a tool that allows graphical commands to be streamed outside of the software or game running from within an emulator and executed on the native hardware. This provides very powerful functionality and can solve a number of the issues discussed previously.

For example, by removing the direct reliance on a physical graphics card, a game can be written without requiring dependencies to interface with one. A more specific example; Direct3D on the Windows platform; not only is a specific version of Direct3D

removed from the list but also underlying libraries, such as the Windows API or DXUT (DirectX Utility Toolkit) providing the window and graphical context to render to. This, in turn, removes further dependencies such as those handling keyboard or mouse input from the game window using DirectInput. Ultimately, a whole chain of dependencies can be removed.

With this in place, the game can be developed without specifically targeting a platform. If a new platform appears, migration to it should require very few modifications because no assumptions about an initial platform have even been made by this point.

Most importantly, if a new platforms appears at a much later date, the exact same idea holds. With this reduced set of dependencies and assumptions, migrating to it would barely require any porting. This provides a strong strategy towards digital preservation and also mitigates against risks of platforms that become obsolete.

The next concept behind Hydra is that while an emulator can provide as much of the platform as is needed to support the game, the graphical calls can be passed outside of the virtual machine and consumed by any hardware that can process these graphical calls. Because of this, the binary does not need modification to support another platform. This allows for the preservation of software in which the source is unavailable (either because it has been lost or because a company has withheld it as a trade secret and/or intellectual property, even after its bankruptcy).

The protocol behind Hydra (discussed in Section 3.2) is intelligent, meaning that certain actions can be inferred by the native host rather than processed entirely by the software running within the emulator. This is important because the general consensus of the literature presented in Chapter 2 was that pure emulation is often the only guaranteed way to successfully run many types of software, however, it is also the most expensive in terms of resources. By reducing the stress on the emulated processor by delegating as much of the processing to the native host, it is much more likely that acceptable performance can be reached for a large number of use-cases,

with gaming software being an example.

3.1.2 Reduction of Dependencies

In software development, a dependency is a requirement that needs to be solved in order for a solution to work. The most common example of such is a library or existing bit of code that provides crucial functionality that a program requires to execute. For example, this could be some code that allows a program to access a specific piece of sound hardware.

The complexity arises when that dependency does not exist or cannot exist for various reasons. For example, one machine might provide the sound hardware in a very different way to another machine, thus causing the original piece of code to no longer make sense. The different hardware may require different signals to be sent in order to operate it. Instead, a very different piece of code may be required. Likewise, some systems may not provide audio hardware at all.

In all software such as games, a program may use a multitude of different dependencies in order to interface with the different hardware, such as audio (as mentioned) but also video, disk, keyboard, gamepad, network, fonts, etc. Each one of these dependencies are often in turn dependent on another large number of dependencies which brings the software closer to the hardware. This is demonstrated in **Figure 3.1** and **Figure 3.2**. Dependencies do not always need to directly be code, at deeper levels they become more and more reliant on the constraints of the hardware.

Game			
libglut	libgl	OpenAL	libpng
libglx	libX11	ALSA	zlib
Mesa / GPU Driver		Sound Driver	glibc
Debian Linux			
Intel x86 PC			

Figure 3.1: *A view of the dependencies required for a simple game on Linux. Note that a large number of them ultimately depend on device drivers.*

Game			
SDL		SDL_mixer	SDL_image
Direct3D	WinAPI	DirectSound	libpng
DirectX			zlib
GPU Driver		Sound Driver	msvcr
Windows			
Intel x86 PC			

Figure 3.2: *A comparable view of dependencies required for the same game on Windows. Again, note that a large number of them ultimately depend on device drivers but the rest of the stack is quite different.*

In many ways, each dependency can potentially cause an issue when running software on another hardware or platform. If there is a single break in the dependency chain, and if the software relies on that functionality, it will not be possible to continue unless that chain link can be replaced. **Figure 3.3** demonstrates the large number of dependencies that the popular Simple DirectMedia Layer (SDL), a relatively simple game development API, has. This data was generated using Doxygen (a C and C++ documentation generator) to obtain all the includes that SDL at some point requires

during compilation.

Figure 3.3: *Diagram showing the large number of C header files required by the standard SDL2 distribution*

Replacing dependencies is part of a process called porting. It is often a very important task when migrating software to another platform. It can consist of replacing a dependency with another, providing similar functionality but instead specific to that different platform. Once these dependencies have been replaced, the software can continue on the new platform. Depending on the type of library, sometimes it can run on multiple platforms. This is especially true if the dependent libraries are minimal or themselves portable to multiple platforms.

If a library can be reduced to only require dependencies that are already existing within the project (perhaps for another task), then this is effectively reducing the number of dependencies and is one of the biggest steps towards writing cross platform and platform agnostic code.

In theory, if a game can eliminate all dependencies, then it is effectively platform agnostic and can be therefore be run on any past, present and future platform. Reducing dependencies is one of the main aims of this research. In particular, replacing dependencies in such a way that they are less tightly bound to the platform or removing them entirely.

The reference implementation of Hydra which is used for experimentation in the research has been written in a disciplined manner. For one, it introduces a very limited number of dependencies. This means that very likely it can be utilised on all platforms, ranging from DOS all the way to the present day and even platforms not yet introduced. **Figure 3.4** shows the number of dependencies that SDL requires when using Hydra as a back end. It can be seen that this number has been greatly reduced to the standard C runtime, such as **stdlib.h**, **string**, etc. and POSIX libraries such as **unistd.h** and **sys/socket.h**.

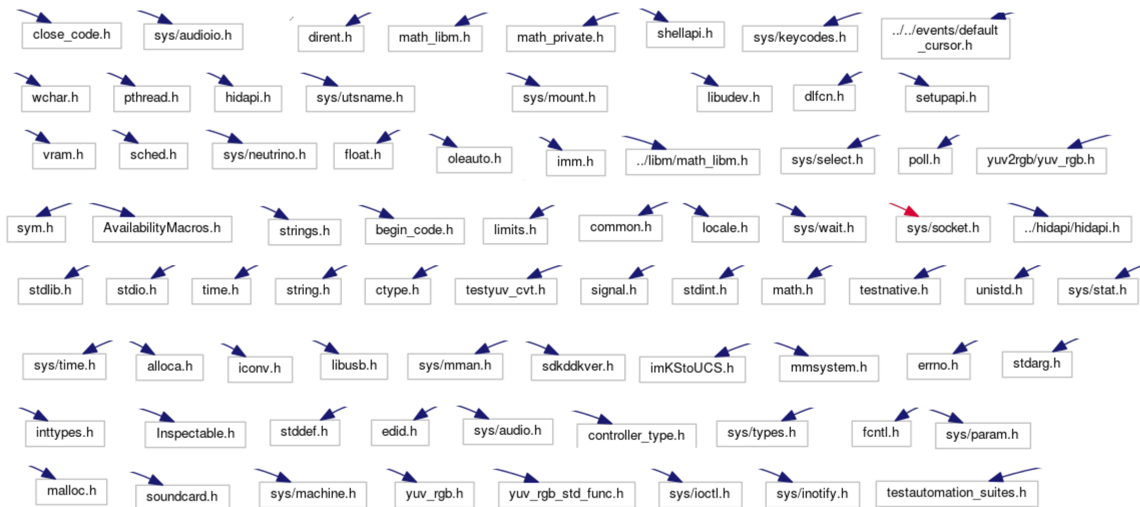

Figure 3.4: *Diagram showing the greatly reduced number of C header files required by SDL2 using a Hydra back end*

The reduced number of dependencies also stems from the fact that, as discussed in **Section 3.1.4**, most programming languages require C at some level in their technology stack even if they themselves are written in another language, adding

additional dependencies. Use of the Java language alone and all the additional VM and platform integration machinery that it requires adds many more external dependencies than SDL2 alone (Hsieh et al., 2013). Hydra is written directly in C and so additional dependencies are minimal. Hydra also needs a way to pass data outside of the virtual machine, this can be done in a variety of ways but a network connection or RS-232 stream are likely to be the most suitable. This is because they are both standardised interfaces which in turn means that emulators are much more likely to support them. They can be seen as the lowest common denominator approach.

A popular alternative is to use a synthetic device specifically suited for this task. This can certainly achieve faster performance because it does not have the overhead of emulating the entire TCP/IP or RS-232 stack but it also requires bespoke drivers to be developed which are typically unique to each emulation platform. One example of this is for the Microsoft Hyper-V virtualisation platform. OpenBSD implemented a driver for the synthetic interrupt controller and paravirtual bus allowing for simple key/value pairs to be sent between guest and host very quickly. This support was first introduced in OpenBSD 6.1 in 2017 (9 years after Hyper-V was first released) which means that any OpenBSD version prior to that will not support this feature. This will cause an issue if you specifically needed to emulate an older version of OpenBSD or if Microsoft changed the synthetic hardware specification, requiring an updated driver. This will potentially be an ever changing goal post and unlikely to be sustainable for digital preservation.

Hydra provides support for many types of communication across a VM boundary and due to the nature of them it is very simple to add more (potentially as loadable plugins), so the original binary can still remain untouched. However the main focus is on the TCP/IP approach because it provides superior performance compared to serial and in some cases is even more common in PC emulators.

3.1.3 Separation of Application and Hardware

Given enough time it can be argued that even the best maintained hardware will develop issues and malfunction. Even without physical moving parts such as hard disk platters, the transistors in hardware eventually fall to fatigue. Hardware over time will need to be replaced. However, it also gets to a point that it is impossible to procure a specific type of hardware part that needs replacing. Moore's law (Schaller, 1997) is a double edged sword in that the constant upgrade cycle of hardware, whilst striving for performance, also means that older designs get left behind; not only for technical reasons but also social and monetary reasons such as planned obsolescence. For digital preservation strategies to work it is very important to reduce the reliance on hardware. Emulation is the key here and as suggested by the literature in **Section 2.11** preferably pure emulation rather than virtualisation because, again, it is very likely that hardware available today, which is able to virtualise current software could also break, deprecate and disappear, leaving no machinery capable of natively executing the virtualised instructions.

However, for entertainment, it is often the case that using the hardware is part of the attraction (Oswal et al., 2016) and it is not desirable to lock the software purely into an emulator like we might for a business or utility program. An example of this is using a specific gamepad and being able to pass the protocol used by the gamepad in and out of the emulated system. This is a very common approach and is seen in many emulators specialising in game consoles or those that need to passthrough generic PCI devices (Jones, 2009 (accessed January 2, 2019)) (**Figure 3.5**). If we use the same logic to look at a graphics card as a bonus rather than a commodity, we will see that players do not just want to display graphics which emulation already provides. Instead, they ask for much higher performance than traditional emulation often can provide. This outlook helps us to avoid falling into the trap of targeting the lowest common denominator, which ends up providing a very poor output for

consumers.

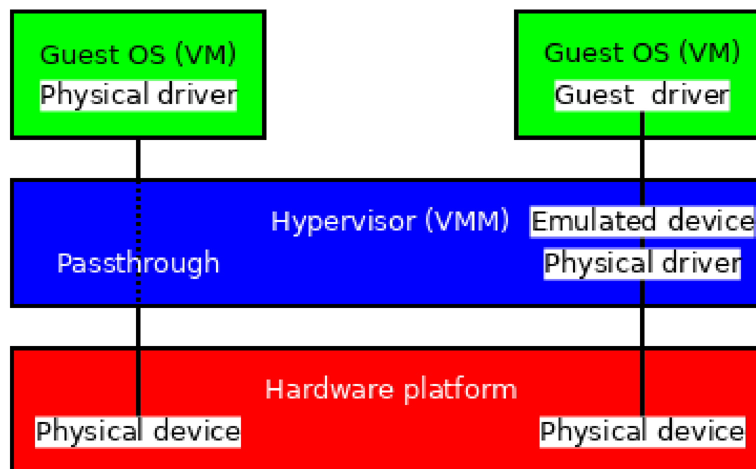

Figure 3.5: *Passthrough within the hypervisor, near-native performance can be achieved using device passthrough (Jones, 2009 (accessed January 2, 2019)).*

The solution to the GPU problem may potentially be very similar to how we solve the gamepad issue. If necessary, we need to be able to pass control of the GPU in and out of the emulator via the use of a protocol.

3.1.4 The Illusion of Programming Languages

There is a vast number of different programming languages available today, each one with potential advantages and disadvantages. For example, Python has been touted to be easy to learn (Lindstrom, 2005), whereas Haskell has been developed with the reduction of side effects in mind (Harris, 2005).

However, when it comes to the portability of a language or how well it caters to different hardware, this is a very different problem to solve and one that tends to remain quite unsolved within the industry.

An example of this can easily be observed in the Java programming language. From inception it was designed to be cross platform and portable (Curtin, 1998; Grønli et al., 2014). Sun Microsystem's original slogan for Java, "Write once, run anywhere" has often been cited within the industry as a very significant benefit and

has potentially been a large contributing factor for Java's popularity (Singhal and Nguyen, 1998; Bishop and Horspool, 2006).

However, there are a number of papers which are critical of Java's cross platform abilities suggesting only varying degrees of success (Bloice et al., 2009; Latif et al., 2016). In particular, porting the Java platform to new operating systems, especially in the mobile or embedded sector has been complicated (Blom et al., 2008). Some projects have even migrated away from their Java implementation to alternatives such as HTML5 citing portability issues; especially with modern platforms such as web browsers (Zbyszyński et al., 2017).

This brings us on to the main issue with looking at programming languages as a means to gauge potential portability. The language ends up having very little bearing on the portability of software and can do very little to help or hinder the software being ported. As the Java documentation(Oracle, 2019 (accessed July 9, 2019)) demonstrates, the language itself is actually quite a trivial component on top of a much larger system. The rest of the system is what dictates the portability of a development platform.

If we contrast this against the Java platform on Android (**Figure 3.6**) it can be seen that although the Java language remains largely the same, there is little else that has remained constant. For example, the GUI library *javax.Swing* is not present, instead, it is replaced by the View System component. Many of the frameworks are missing or replaced with incompatible alternatives. Further down the technology stack, a firm C and C++ underpinning can be seen. This suggests that not only does Java impose many dependencies on a project, but it can also not currently exist without the C (and C++) languages.

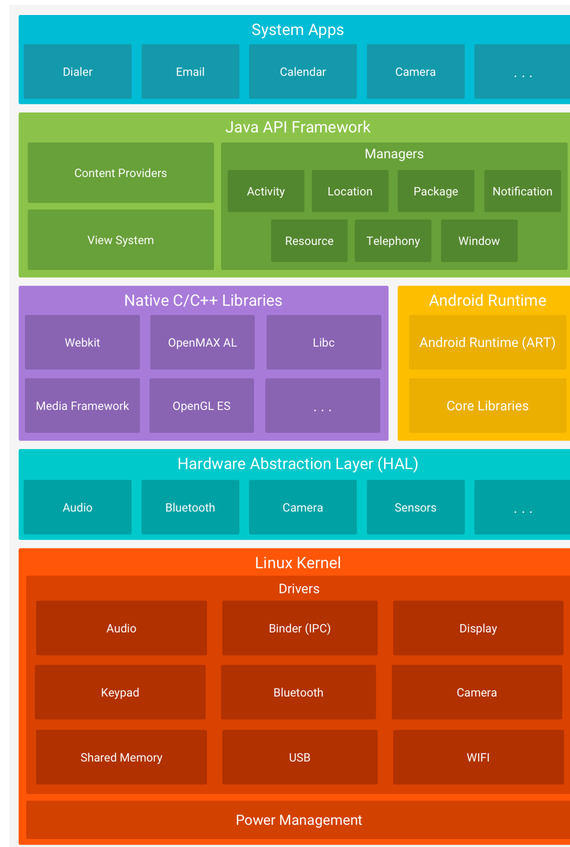

Figure 3.6: *A diagram showing that the Java platform on Android (Dalvik) is made up of many different dependencies to the standard Java reference implementation (Google, 2019 (accessed January 2, 2019)) (licensed under CC BY 2.5).*

As can be seen in **Figure 3.7**, Java as a language in almost all cases exists as bytecode when compiled. This bytecode is a platform agnostic representation of the program's logic. This platform agnostic approach has a benefit in so far that the code does not need to be recompiled to run on a different machine architecture, yielding some benefits. However, modern processors cannot run Java bytecode directly and, instead, this task is delegated to a Virtual Machine known as the JVM. This unfortunately is where the portability of Java effectively ceases.

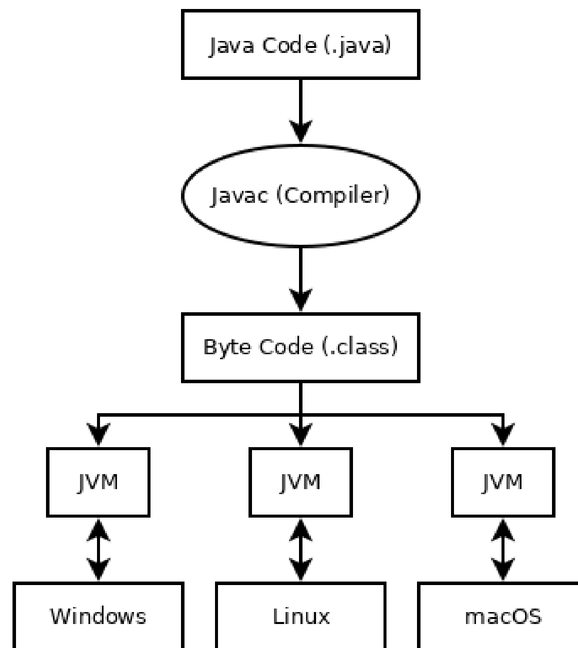

Figure 3.7: *A diagram showing that the bytecode must ultimately end up being processed for a specific platform. Sometimes a JVM provides only a limited choice of supported platforms. Here Windows, Linux and Mac are shown as supported.*

A Virtual Machine is, by definition of its task, unable to be written itself in Java. Usually for performance reasons and also a number of technical reasons this must be written in a native language. Often this is C or C++. This ends up suggesting that Java technology can therefore in no way be more portable than a native language such as C or C++ because it ultimately ends up relying on it anyway.

Where Virtual Machine technology then goes on to end up less portable than a native language is when it needs to perform garbage collection and Just in Time (JIT) compilation for performance.

Garbage collection is the solution that a number of languages have taken to solve the issue of memory management (Detlefs et al., 1993). A basic overview of the garbage collection process is that the heap and stack are scanned for any references that are currently in use. Then, if there is some data on the heap that is no longer referenced anywhere in the heap or stack, it can safely be freed. There are many complexities to garbage collection (Weiser et al., 1989; Henderson et al., 1995) such

as if a reference to memory is stored in a non-standard way (such as XORed) or if a reference has moved half way through the initial scanning pass. However, the biggest issue with garbage collection which affects portability is that it has to either make a number of assumptions about a platform or request this information using a very platform specific manner such as an API (Holden, 2019 (accessed January 2, 2019)). One example is the stack. Not all platforms have a contiguous stack. Some older versions of the PIC platform do not have a stack at all in fact. On some platforms the stack grows upwards and on others it might grow downwards. This makes it quite complicated to ensure that the stack can be scanned for references in a robust manner whilst at the same time remaining platform agnostic. We can achieve some amount of success in ascertaining the top or bottom of the stack in C. For example with the following function:

```
1  char *GcFindStackBottom()
2  {
3    char *tmp = NULL;
4
5    return (char *)&tmp;
6  }
```

At first, this may seem trivial. We are simply creating a new variable on the stack and then returning its location. This is then used to calculate the stack bounds and thus where the garbage collector will need to scan to. Unfortunately, it is more complex than this; for example, depending on the compiler optimisation, the variable might never end up on the stack; instead, it could be retained within a processor register for performance reasons. It is possible to partially mitigate against this issue by using the ***volatile*** key word when declaring the temporary variable, however, many compilers either ignore this qualifier or they use it as a hint and are not guaranteed to put the variable on the stack. The C standard does not dictate that they need to adhere to this key word.

JIT is the process of converting logic obtained from the bytecode into native instructions for the platform which the hardware can execute natively. This by definition can not be platform agnostic and further reduces the portability of Java.

Figure 3.8 shows where JIT is utilised within the Java system.

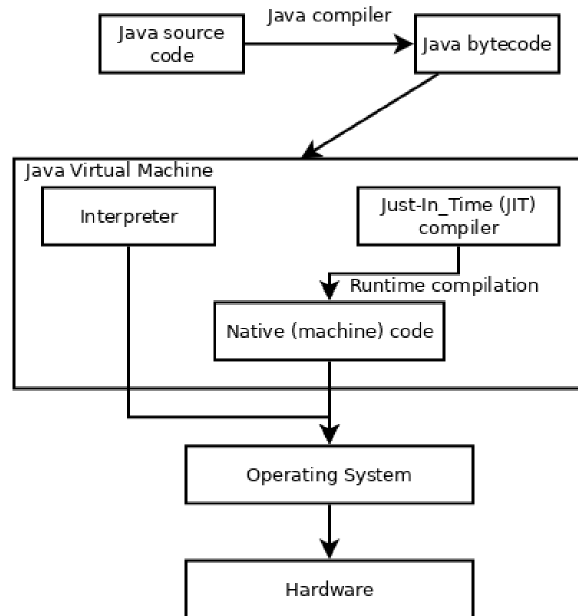

Figure 3.8: *Diagram showing the interaction between the JIT system and the rest of the Java JVM*

As can be seen in the diagram, only the steps above the Java Virtual Machine are platform agnostic. The JVM itself and the lower layers are very platform specific. JIT in particular needs to have knowledge about these underlying layers in order to generate the correct machine code.

These issues discussed remain unsolved in existing modern implementations of Java and can be evidenced by the fact that the current reference implementation of Java 7 (OpenJDK 7) consists of a very large codebase (Kaczmarek and Kucharski, 2004) involving a number of different languages and in particular 13.7% C++ and 5.6% C code, along with many of the maintenance issues that they provide. Currently a large team of developers employed by a variety of companies within the IT sector are available to maintain such a platform as shown by **Figure 3.9**. However, there is no guarantee that this same resource will be available in the future, effectively halting the feasibility of the Java platform for digital preservation uses.

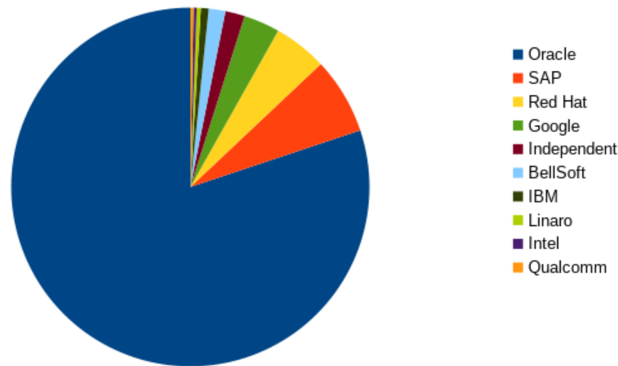

Figure 3.9: *Graph showing the number of companies working together to maintain the Java platform.*

The majority of issues discussed in the portability of Java unfortunately exist in the majority of managed VM solutions. The popular .NET platform based languages such as VB.NET and C# are no exception to this. The technology stack (**Figure 3.10**) looks remarkably similar to Java.

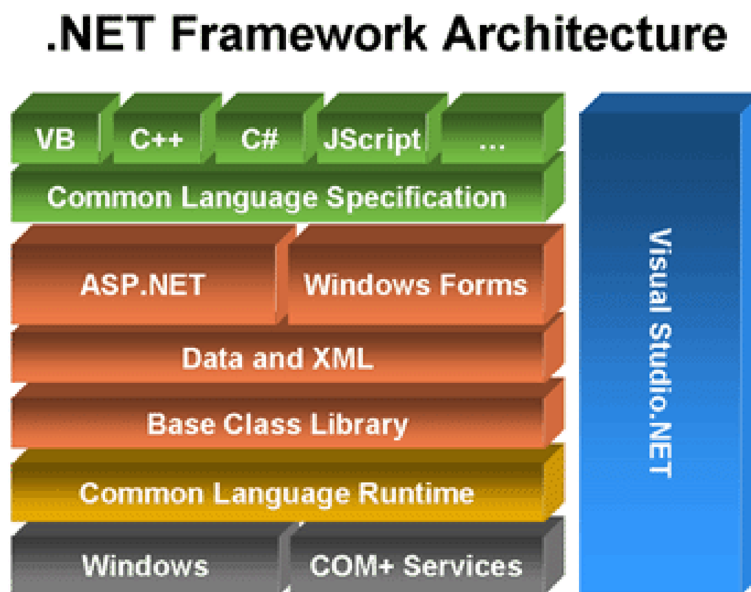

Figure 3.10: *Diagram showing the underlying technology and dependencies for Microsoft's .NET framework. 2011 IEEE. (Reprinted, with permission, from Teresa P. Lopes and Yonet Eracar, TPS development using the Microsoft .NET Framework, AUTOTESTCON, 2011 IEEE)*

It could be said that Microsoft was heavily inspired by Java in their design of .NET and this is often seen as true (Tebbe, 1998), also inciting some criticism from James Gosling, the creator of Java (Gosling, 2002 (accessed January 2, 2019)). However, Java and the JVM itself it can be argued was not an original concept and was in competition with very similar technology called Limbo (Back and Hsieh, 1999; Yurkoski et al., 1998). What is interesting is that both of these technologies were commercial ventures with their design heavily based on Alef, the predecessor to Limbo, developed at Bell Labs in 1992 from the same creators of the C language (Kernighan, 1996). Alef at the time had a much simpler underlying platform and was trivial in comparison to port to other platforms. It could be said that .NET and Java lost this valuable attribute as they gained more and more features whilst competing to win favour with the industry.

A core requirement for most technologies is to be able to control the machine hardware that the software is run on. Interestingly, VM technologies like .NET and Java are unable to do so directly because they are confined to a VM but also their data types do not directly correspond to the native types of the machine currently executing the stack. This is for the portability of the applications running on top of the VM so the developer does not need to take into consideration platform specific requirements. In order for a technology such as this to communicate with the underlying machine, what is known as a binding (Beazley et al., 1996) must be developed. What this process involves is mapping certain data and functions to the native equivalents (written in a native language such as C) and then this can be interfaced with via the underlying native VM (such as the JVM) rather than the managed language directly. It is only because both components are native that they can interact. This can be a time consuming and complex process where a number of errors can be introduced. Bindings often involve two counter parts (especially if object oriented in nature or dealing with arrays) which need to be kept in sync, one written in the language that

the callee target is written in and another written in the language of the caller's language. This is shown in **Figure 3.11**. Many developers can avoid this experience by simply using or purchasing a binding made by a third party, however, these often become unmaintained as the underlying native API that the binding is interfacing with changes (Mayer and Schroeder, 2014). Again, the fact that some developers will not experience the binding creation process themselves first hand can also lead to this false assumption that native languages are unnecessary and that anything can be accomplished inside a managed language like Java or C# which is simply not true (Saipullah et al., 2012).

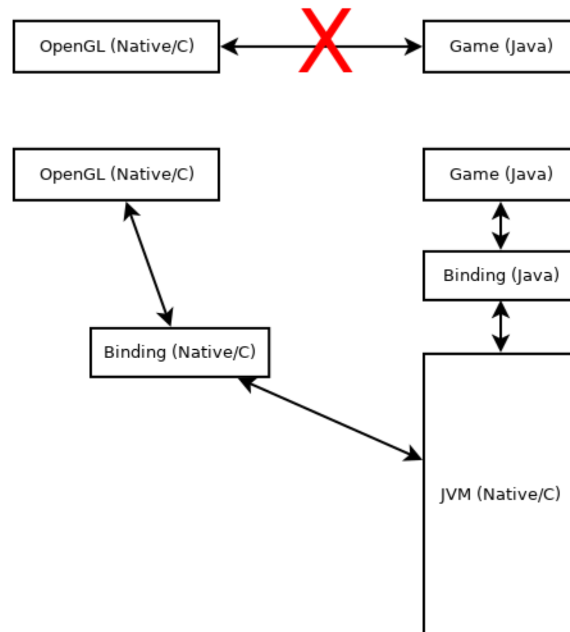

Figure 3.11: *A diagram showing the indirection that a managed language needs to go through in order to call into a native or system library*

An extension to this problem of binding in managed languages is not just in calling functionality in the host operating system or platform but also attempting to access functionality written in different languages. This is often a complex undertaking and can involve a large project using an unsuitable language for large proportions of it simply because the interfacing between different languages is too awkward. The

reason why this is a complex process is because managed languages from different managed platforms are unable to call into other languages directly (this is in fact a relatively rare ability for any language). This is mainly due to the reason mentioned previously, i.e. managed languages do not call into other languages and libraries directly, instead, they call into their underlying VM implementation and the native layers within there perform the important interop with other systems. This ultimately means that if JVM/Java and .NET/C# were to call between each other, not only would they both need to call into their different respective underlying VMs but then an intermediate system would need to be developed to allow the different native VMs to be able to communicate. There would be significant overhead for this to translate all the communicated data into forms that both VM implementations can then understand. This is exemplified in **Figure 3.12**.

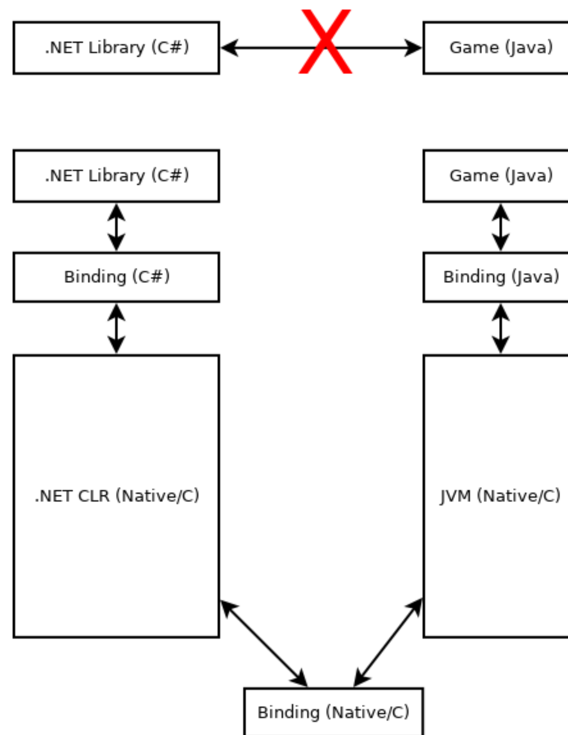

Figure 3.12: *A diagram showing the many layers of indirection that a managed language needs to go through in order to call into another managed platform and language*

This is, to an extent, simplified if one of the languages in question is native (i.e C and JVM/Java) because that can communicate directly with the native VM and avoid much of the complexities encountered in the previous scenario. This approach can remove the need for one layer of bindings at the very least.

Note that this problem is not in the same category as calling between languages using the same underlying managed VM platform. This is a relatively simple task because both languages will mostly output the same bytecode. For example VB.NET and C# both run on top of the CLR (Common Language Runtime) and when compiled generate almost identical IL (Intermediate Language). This means that one could mistakenly assume that because C# can interface with VB.NET or IronPython that it is a good choice for integration with other languages, however serious issues will be encountered when attempting to interface with JVM or interpreted languages such as Java or Python.

This can potentially be seen with the Unity game engine; there is a much smaller selection of AAA middleware available for it compared to say Unreal Engine 4 because being constrained to the .NET platform via Mono, it is complex to integrate with the majority of middleware, which due to the nature of the high performance gaming industry is overwhelmingly dominated by native C++ development. UE4, with a focus on the native C++ language, ensures a much more straight forward integration path (Sweeney, 2006).

Based on the discussion above, in order to maintain our goal of finding a truly cross platform solution, these Virtual Machine based solutions unfortunately all need to be discarded which leaves us with a much smaller pool of potential choices. It has also been discussed that for portability, it is of little use looking at programming languages themselves. Instead, it is better to identify a solution based on underlying technology and dependencies. In this work, languages have been broken down into 3 categories; Managed, Interpreted and Native. It has been seen that the Managed option, due to the requirement of complex unportable Virtual Machine technology is

not a viable solution. The next to discuss is Interpreted languages.

Interpreted languages read instructions in the same way as Managed languages but rather than convert the instructions, either plain text or bytecode, into native machine instructions, they are processed indirectly by ultimately calling into the native binary and, potentially, based on a lookup table of functions to call based on keywords. This has the advantage that JIT compiling is not necessary, so no additional assumptions need to be made about the underlying platforms. However, due to the lack of direct execution of the instruction on the processor, there is a high performance cost.

Popular interpreted languages for games consist of Lua and Kismet 3 (now turned into Blueprint). Both of these can generate bytecode but this is ultimately interpreted rather than via JIT.

As with the Managed languages, the code ultimately needs to run on the hardware. Again, as is typical of Managed languages, this is done by using a native language such as C or C++ to implement the actual interpreter. Therefore, in terms of cross platform portability, very little has been achieved by using an interpreted language. It can never be more portable than the underlying native language it is being interpreted by. In fact, there are even some cases where the native language such as C is better used as an intermediate portable assembly rather than an interpreter (Henderson et al., 1995). This is in fact the strategy used by Cfront, the first C++ compiler written by Bjarne Stroustrup. It generated standard C code from the higher level C++ (Carcerano, 1998). For this reason, it was more known as a transpiler rather than a compiler.

Even though interpreted languages can be more portable than managed languages, the vast majority put restrictions on the C compiler used. The most common is the use of a modern implementation of C such as C99 or C++, which for most purposes poses absolutely no issue, however, when looking at supporting older platforms for digital preservation purposes, these older compilers may not support code which is written to target a newer standard than C89.

Native languages are those that compile directly to machine code which executes directly on the processor. There are relatively few of these languages, however, arguably there are many more compilers for them than there are languages. There is a very large number of vendor agnostic C compilers available and being actively developed today (Henderson et al., 1995). In fact, there are more compilers that work with the C language than any other programming language.

These languages require a compiler to generate the machine code which executes on the hardware. The compilers themselves are often bootstrapped from other platforms (cross compiling). The very early compilers were generally bootstrapped from earlier languages such as Forth or directly in the platform-specific assembly.

The most common native languages used to date are C and C++. One of the main strengths of C was that it was originally developed with portability in mind. It was designed as a portable assembler. Rather than having unique assembly instructions per platform, such as x86 Assembly, SPARC64 Assembly, C would be just high level enough to convert one high level instruction to all the many different hardware instruction set specific versions.

For digital preservation purposes, C has the important advantage that it has a very limited number of subsequent dependencies in order for the binary to function on a machine. Unlike Managed solutions, fewer assumptions need to be made at run-time and a large complex Virtual Machine does not need to be maintained in order for the program to run. For all intents and purposes, it is the lowest common denominator of the technology stack from which most other solutions rely on but it itself does not rely on any of the others, causing many to agree that it is one of the very few solutions to true platform agnostic development.

It could be thought that if we go one step below C and use Assembly directly, we could eliminate more dependencies and become even more portable. However, this is incorrect; there is no single assembly language, it is different for each processor architecture. Therefore, it becomes less portable to write code directly in Assembly.

With that said, C was designed from inception to provide a portable assembler (Kernighan and Ritchie, 2006). C as a language does seem relatively high level compared to many dialects of Assembly but that unfortunately was necessary to abstract the differences between Assembly languages. In the past there has been work in the area to improve the ability for C to act as a portable Assembly (Jones et al., 1999; Henderson et al., 1995; Jones et al., 1993).

C++ is a close relative of C in that it has been derived from it, whereas languages like Java and .NET are languages written from scratch to feel similar to C. C# in particular is not related to C, the name was known to be a partial marketing tactic by Microsoft to encourage uptake by developers and to gain popularity for their .NET platform (Hamilton, 2008 (accessed January 2, 2019)). As discussed previously, this language has many more technical similarities to Java than native C.

Based on this research, it really cannot be underestimated how important the portability benefits provided by C is to cross platform development and digital preservation. Since the implementations provided as part of this research must strive to achieve a result as close to platform agnostic as possible, C has been chosen as the development language. This has unfortunately meant that some additional work has been required, and undertaken to overcome some of the complexities with dealing with a relatively low level language. This work is discussed in more detail in Section 3.2.5.

3.1.5 Platform Identification

Usually the constraints placed upon a project will reduce the number of platforms that the software is intended to run on. Such constraints usually consist of time limitations for release, number of developers to work on the project or man-hours (Weinberg and Schulman, 1974). On occasion there are also technical constraints such as hardware limitations. For example, a game using serial communication will be a poor candidate for porting to a phone because these typically (but not always)

do not provide RS-232 hardware. Games designed to run on more powerful hardware pose a challenge when being ported to computationally weaker devices such as mobile devices. This is because these devices are designed to be utilised as a client where much of the processing is undertaken on the more powerful remote serverJärvinen et al. (2010). Therefore mobile devices often lack the processing power required to undertake a number of intensive tasks, including emulation.

Almost all technical issues can be overcome with enough hours spent on the project (Weinberg and Schulman, 1974). For example, the popular DOSBox software allowing users to play older DOS games over a serial cable has been ported to phones and tablets. It gets past the limitation by channelling the serial commands through TCP/IP to another client across a network, avoiding the requirement of specific hardware. Likewise, id Software's Quake II was seen as quite advanced for its time on the PC, which typically had much higher specifications than available consoles at that time such as the original Sony PlayStation. Lobotomy Software Inc. still ported the game to the N64, Sony PlayStation and even Sega Saturn by replacing the entire renderer with a custom one called SlaveDriver (Linneman, 2018 (accessed March 3, 2019)) (originally intended for the game Powerslave). Not only was the renderer replaced but most of the game levels also had to be remade. Arguably, if too much of the original software is replaced, the port could be better described as a rewrite.

Avoiding the need to rewrite software as per the example above is one of the main goals of this research, for example, Minecraft was originally written in Java which unfortunately has proved to not be as cross-platform as many developers had believed at the time. The technical reasons for this were discussed in Section 3.1.4. For example, it was not possible to get a Java VM running on the Microsoft Windows Store or the Xbox One. Again, the reasons were partly due to technical issues and partly due to artificial restrictions laid down by Microsoft's policy. Therefore, Minecraft was rewritten in C++ rather than .NET with the goal of future platforms

(particularly mobile) in mind.

The main issue comes from the fact that target platforms cannot be completely identified on the day that development begins. In some cases, the target hardware could become deprecated half-way through development. Software teams must be flexible and sometimes portable code should be written, just in case. This way, if a new for example, mobile platform comes out, the software can hopefully be ported quickly (and become an early adopter of that platform) in order to increase the financial gain from the software.

Likewise, the solution provided by this research is based on the idea that no assumptions must be made against which platforms can and cannot be supported, i.e. be it modern, current or past hardware.

Much can be learned by targeting past platforms. In particular, it provides a great test-bed for potential digital preservation strategies. For example, if software works on current platforms but not past platforms, it is not unfeasible based on this knowledge to project that future platforms may also not be supportable. It could be that the software relies too much on a single type of hardware that did not exist in 1980 and probably also will not exist in 2040. Or, it could be that the software requires a specific operating system facility, such as the Windows API that did not exist in DOS and therefore might not exist in a future revision of Windows.

It is unwise to make the assumption that because platforms become more complex they will also become more powerful and featureful. This has not always been the case. For example, MS-DOS was able to host large database software and control commercial hardware for a large number of industry applications, whereas iOS today can barely write a file outside of a very limited sandbox or listen on a port for waiting connections. Of course, the hardware and UNIX-based platform might be far superior to DOS but due to security and potential artificial limitations on the platform based on licensing and Apple's store policy (Blazakis, 2011; Keene, 2011; Sharpe and Arewa, 2006) it is potentially more restrictive and limiting than an operating system that

existed almost 20 years before it. Another example was that DOS could directly access the framebuffer of the graphics card to draw. For performance reasons, this is no longer how modern GPUs work and, as such, direct manipulation is no longer possible in the same way. Instead, shaders are used as a more restrictive (albeit much more performant) workaround.

For the purpose of this research the platforms identified that intend to be supported range from those that existed in 1980, all the way to the current day and ideally into the future. Given that 1980 is around 40 years ago from when this research was conducted, based on that projection, this solution aims to support platforms up until around the year 2060. As a basic requirement, a platform at the very least must support a C compiler (so that it can be programmed in a cross-platform manner) and a way of communicating externally such as RS-232 or TCP/IP (so that a protocol can be channelled through from the hardware) An example list is as follows;

- **DOS (MS-DOS, PC-DOS)** - Represents an older platform lacking a number of features. Most importantly an inbuilt TCP/IP stack which instead will need to be provided by the third party Trumpet Sockets layer. Whilst it had some early GPU hardware acceleration through the 3Dfx Voodoo hardware, it certainly did not support even early OpenGL support. The DJGPP compiler and DOS extender did ultimately support functionality taking advantage of a memory management unit (MMU) through a minimal POSIX layer but again, this was very primitive. Porting to this platform will be useful to test Hydra with some of the earlier C compilers. It will also be useful to see interaction with the networking library on a platform that does not support shared objects.

- **Plan 9** - A useful platform to test in terms of exotic features. Much of this was due to the fact that the developers were given full access to break compatibility with UNIX. Something that many other operating systems were unable to due to likely loss of sales. Whilst Plan 9 does provide POSIX sockets via the APE

(ANSI/Posix Environment) layer, the goal is to utilise the native TCP/IP stack via the dial frontend. A useful challenge to exercise the portability of Hydra.

- **Solaris 9** - An older UNIX platform. Useful to test because very few virtualisation platforms officially support it. It was just old enough to miss commercial interest. It provides a fairly standard POSIX sockets layer but it will be interesting to see if there are any subtle differences from modern Linux / BSD which pose a challenge to the portability of Hydra and need to be overcome.
- **Windows NT 4.0** - An older Windows platform. The socket layer is Winsock, which is ultimately based on an older specification of the BSD sockets. This version of Windows is particularly interesting because the Terminal Services Edition (TSE) provides the earliest implementation of Microsoft Remote Desktop. It will be interesting to run a software renderer through this at high resolution compared to via Hydra. Though this experiment is going to be part of future research.
- **Windows 10** - A modern Windows platform. Part of the interest of this is actually how it will interact with the version built for Windows NT 4.0. Windows has very good backwards compatibility but there are certain areas where this has not been possible to retain such as DirectX (Audio/Video) and OpenGL due to close interaction with the hardware. It would be useful to discover if delegating a number of the graphical requirements to Hydra could reduce pressure on the backwards compatibility. Such as only requiring the sound support module of DirectX. It will also be useful to find out if the Winsock API has had any breaking changes, both in terms of API (Application Programming Interface) or ABI (Application Binary Interface).
- **Android** - An example of a more restrictive platform in terms of sandboxing permissions. Whilst it is ultimately utilising the Linux kernel, there are enough differences to cause concern. It provides a POSIX sockets layer and with a

native C/C++ cross compiler, this can be directly accessed. If particular interest is how the OpenGL state can be managed. By default the context is lost when a program is suspended. With Hydra, this should no longer pose a problem because the state is safely stored on the client machine connecting to it. Potentially a localhost loopback would also be interesting to test so that the web browser running on the local device can connect to the Hydra session running on the same device. This could be useful to reduce pressure on porting components such as the app UI and input system.

3.2 Architecture

The architecture of Hydra follows a fairly typical client-server networked server. As **Figure 3.13** demonstrates, multiple clients can connect to a single server and communicate. In this case, the server is the game or graphical software utilising Hydra in place of the standard OpenGL library as a drop-in replacement, in a similar way to other OpenGL implementations such as OpenGL|SC (Khronos Group, 2003 (accessed February 9, 2015)).

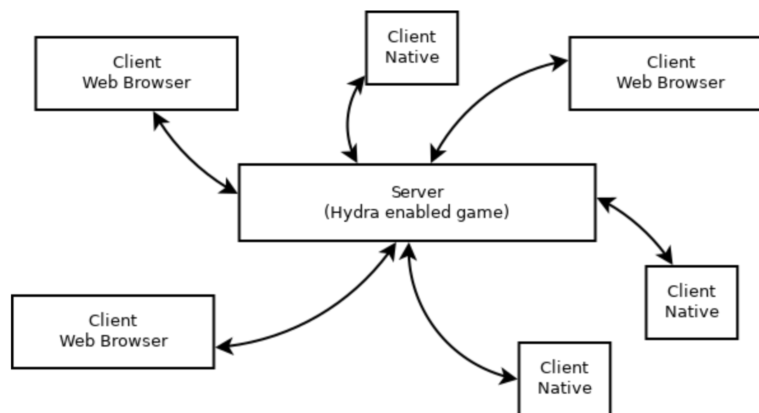

Figure 3.13: *Diagram showing that Hydra is a fairly typical client-server application. Note in particular that many different client implementations can be used.*

3.2.1 API Cloning

The Hydra API directly mimics OpenGL in terms of its design, function calls and usage. This approach is important for a developer because it reduces the amount of work needed on their behalf in order to integrate with an existing OpenGL application. API cloning was discussed in **Section 2.12.1** and it was generally agreed that it can greatly facilitate cross platform development.

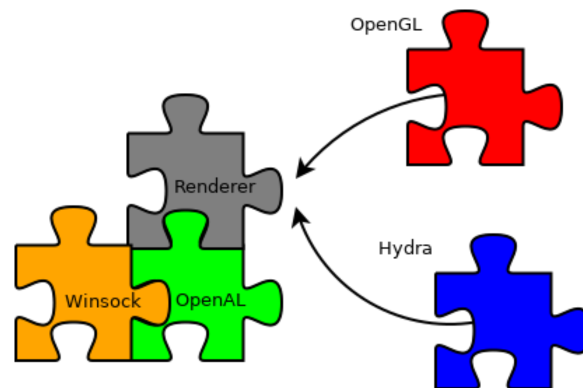

Figure 3.14: *Components can easily be swapped with others if they have identical APIs.*

At compile time, a developer can simply instruct their C compiler to search for the Hydra header files (GL/gl.h) in place of the standard OpenGL ones (GL/gl.h). The code itself will then be able to utilise standard OpenGL commands. At link time, the developer instructs the linker program to include the Hydra library (libhydra.a) in place of the original OpenGL one (libGL.a). This means that as the objects are linked, they will be correctly connected to the corresponding implementations of the same name (**Figure 3.14**). However, it is important to note that the implementations of Hydra and OpenGL differ greatly.

To help demonstrate how powerful this cloned API functionality is; the following code listing (**Figure 3.15**) is compatible with both Hydra and OpenGL and provides the bare minimum required to display a triangle on the screen (**Figure 3.16**).

```
1  #include <GLD/glut.h>
2  #include <GLD/gl.h>
3  
4  GLuint positionBuffer = 0;
5  
6  void init()
7  {
8    const GLfloat vertices[] = {
9      0.0, 1.0, 0.0,
10     1.0, 0.0, 0.0,
11     0.0, -1.0, 0.0
12   };
13 
14   glGenBuffers(1, &positionBuffer);
15   glBindBuffer(GL_ARRAY_BUFFER, positionBuffer);
16   glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
17   glBindBuffer(GL_ARRAY_BUFFER, 0);
18 }
19 
20 void display()
21 {
22   glClearColor(0.1f, 0.0f, 0.1f, 1.0f);
23   glClear(GL_COLOR_BUFFER_BIT);
24 
25   glBindBuffer(GL_ARRAY_BUFFER, positionBuffer);
26   glVertexPointer(3, GL_FLOAT, 0, 0);
27   glBindBuffer(GL_ARRAY_BUFFER, 0);
28   glEnableClientState(GL_VERTEX_ARRAY);
29   glDrawArrays(GL_TRIANGLES, 0, 3);
30   glDisableClientState(GL_VERTEX_ARRAY);
31 
32   glutSwapBuffers();
33 }
34 
35 int main(int argc, char *argv[])
36 {
37   glutInit(&argc, argv);
38   glutInitDisplayMode(GLUT_DOUBLE);
39   glutInitWindowSize(300, 300);
40   glutCreateWindow("Hydra Example");
41   glutDisplayFunc(glutPostRedisplay);
42   glutIdleFunc(idle);
43 
44   init();
45   glutMainLoop();
46 
47   return 0;
48 }
```

Figure 3.15: *An example listing of Hydra code which when compiled will display a triangle. Note that it is almost identical to the OpenGL counterpart.*

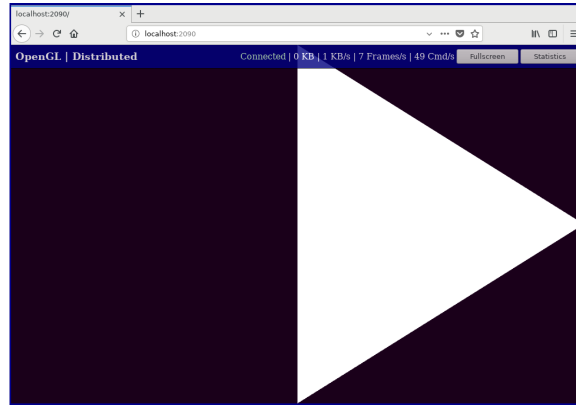

Figure 3.16: *One of the simplest Hydra applications to display a triangle on the screen.*

Those familiar with OpenGL will realise that the code required is identical to the OpenGL counterpart. This means that in most cases, their existing OpenGL code can migrate to Hydra with zero change.

There is no reason as to why the decision between Hydra and OpenGL needs to be made during compilation. Using the well known LD_PRELOAD technique on UNIX-like operating systems or by simply replacing the OpenGL32.dll on Windows with Hydra.dll, the software should simply call into the respective implementation at run time. This functionality could be very important for retrofitting older OpenGL applications with Hydra, even if their original source code is inaccessible.

3.2.2 State Management

As part of the design of Hydra, the application logic is now distributed between the core software, which can be referred to as the model and the simple client, which displays the raw graphics, i.e. the view. This imposes some additional complexity which has required consideration to solve.

The main complexity is that the state of the model must be synchronised with the view. Without this correct synchronisation, the view will simply render the application incorrectly. A very simple example of this is instructing the view which

sampler to use whilst rendering. Without this, a mesh could be drawn with an incorrect texture.

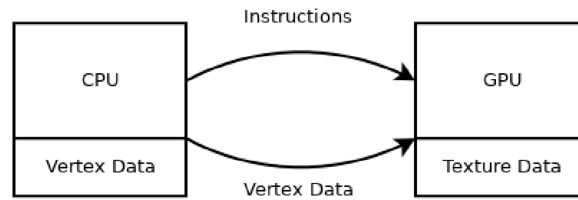

Figure 3.17: *Diagram showing texture data being retained on the GPU but vertex data being transferred across each frame.*

Synchronising state between model and view works largely seamlessly because it almost directly mirrors how modern graphics APIs work. In the past, instructions such as drawing via glBegin, glVertex3f and glEnd used to be sent for each vertex for each shape, each frame. This was relatively expensive because for complex models, the CPU would have to loop through them all, but it would also have to send these instructions and data through the bus between the main memory and the CPU. This often put a bottleneck on the pipeline. This is shown in **Figure 3.17**.

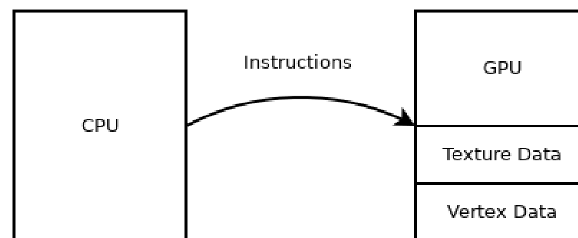

Figure 3.18: *Diagram showing both texture and vertex data being retained on the GPU. Only the instructions are sent across each frame.*

The way that the current APIs and hardware are designed is that the data is retained on the memory on the graphics hardware. This means that the vertex data is uploaded to the GPU once, probably during the loading of a scene. Then, for each frame, the GPU is simply instructed to draw a certain number of vertices from the given vertex data. This means that the data sent between the CPU and GPU is greatly reduced, as demonstrated in **Figure 3.18**.

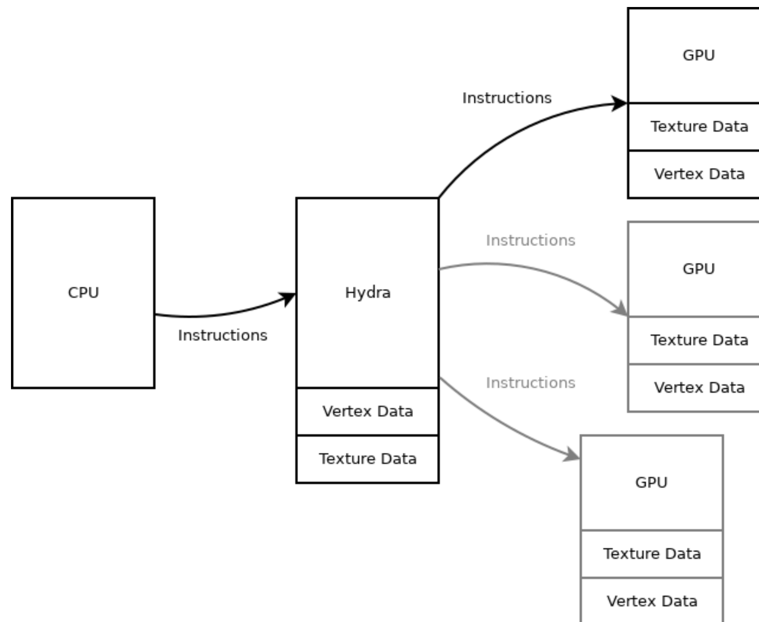

Figure 3.19: *Diagram showing both texture and vertex data being retained on the GPU in a similar manner to traditional OpenGL implementations. The difference is that it had been passed through a network / serial rather than simply across the CPU bus.*

As this is reflected in Hydra, the data is sent from the model through the network once, where it is then stored on the view software. Presumably, unless hardware changes substantially, this will then in turn be stored on the GPU using any graphics API available. This extension to the pipeline is demonstrated in **Figure 3.19**. In future, the hardware will likely change once more and even if GPU technology did end up going back to the old system of sending data per vertex, this would be trivial to decompose from the retained data.

3.2.3 Multi-user State Management

Hydra is designed to be connected to by multiple clients. Not only is this due to the fact that it provides the capability of new potential architectures in multi-user interactive software and multiplayer but also with a view on digital preservation in the future. The use of the precursor of this technology developed in the early stages

of this thesis (called Distributed DeepThought) has been previously explored and has shown strong results in the areas of development and maintainability of multi-user software (Pedersen et al., 2013).

Currently, due to Moore's Law, we have had an implicit expectation that machines will get faster but there is not always that guarantee. With the advent of low power phones, mobile devices and IoT hardware, the industry has already demonstrated that much weaker hardware is entering the market and, in many ways, is replacing the older but more powerful originals (Aroca and Gonçalves, 2012). If this idea is projected further, it might be possible to end up with devices too weak to power a game. In fact, this already exists with the idea of very low powered streaming devices such as the Steam Link boxes and other multimedia specific thin clients (Chen et al., 2018). With the onset of the cloud, potential monetisation of online services and the very effective anti-piracy benefits of streaming, this future of weak devices is becoming increasingly more likely.

Hydra is designed to address this issue; by allowing multiple view devices to connect to a single model, it allows for the expensive generation of graphics to be balanced over a large number of weaker in spec devices.

Due to the fact that Hydra is designed to be connected to by multiple clients, the synchronisation complexities have been increased further. If a state change is requested by the model, it would be wasteful to send the change to each connected client, even if they had no intention to render that part of a scene. Instead, a state differencing system was developed so that only when the final instruction to render has been sent (such as a glFlush) will the model look at the differences in state between the model and specific view and, then, generate a list of instructions to make the client match it. Again, all of the other clients remain unaffected until it is their turn to be processed. This technique is demonstrated in **Figure 3.20**.

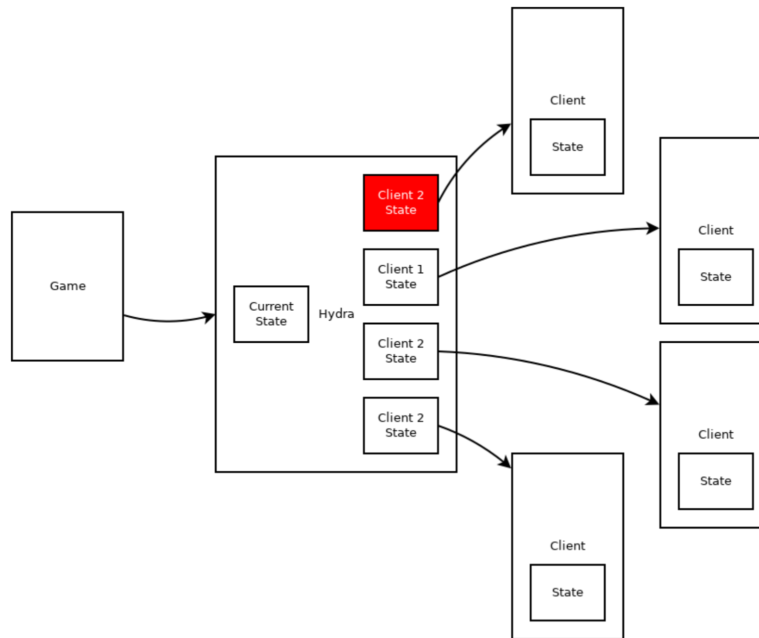

Figure 3.20: *Diagram demonstrating the duplicate state system. Upon a flush to the client, changes needed to bring a state into sync with the current one can be identified.*

3.2.4 Overcoming Issues at a Protocol Level

No matter how well defined a computing standard or API is, the underlying implementations will often deviate from these standards for various reasons (such as for performance or hardware limitations) and thus provide incompatibilities. There are a vast number of implementation details that make working with hardware via OpenGL directly quite frustrating. Typically, a developer will use conditional compilation to ensure that the correct code gets compiled for the correct target. For example, OpenGL prior to 2.1 and OpenGL ES 1.x does not guarantee support for non power of two textures (NPOT), which is still a problem even on some current hardware still in use today (Weber and Quayle, 2018). This is due to a variety of reasons such as vendor optimisation and the standard dictates that it is implementation defined; some implementations do in fact work fine and support these textures. However there is no guarantee that this will not stop working in a future driver update so it cannot be relied upon.

Due to the problems that Hydra aims to solve, this conditional compilation cannot be relied upon because it is impossible to know at build time the platform which is ultimately intended to be used to render the scene. This means that a run-time solution needed to be devised.

There are some advantages to using a protocol to transfer data between the application and the rendering client. One of the advantages is that the data can be modified based on the communicating client. In the case of NPOT textures, the **glTexImage2D** can simply re-sample the image depending on the client platform and OS capabilities. This can allow the developer utilising Hydra to develop standard OpenGL code without even having to know this underlying re-sampling is taking place for cross platform purposes.

Issues caused by platform differences are not confined entirely to OpenGL implementation differences. In particular, there is no reason as to why OpenGL is needed to be used as the client render at all. This is discussed further in Section 7. Some platform differences can be due to security or sandboxing purposes; this might become more and more of an issue in the future as internet services become popular. One of the reasons why the HTML5/WebGL platform was chosen as an initial client test bed is because it is relatively restrictive in the functionalities it provides. One of the most prominent of examples is the asynchronous design with a focus on disallowing long pauses (Chęć and Nowak, 2018). This has caused quite a complex issue for alternative portability solutions such as Emscripten (Jangda et al., 2019). However, for Hydra this has not posed a problem as the code written by the game developer needs no modification; the game will have a main loop that will run continuously without relinquishing execution back to the underlying platform. This holds no effect on the client web browser which will simply act upon messages as they are received. This detachment between model and view solves this issue implicitly (**Figure 3.21**).

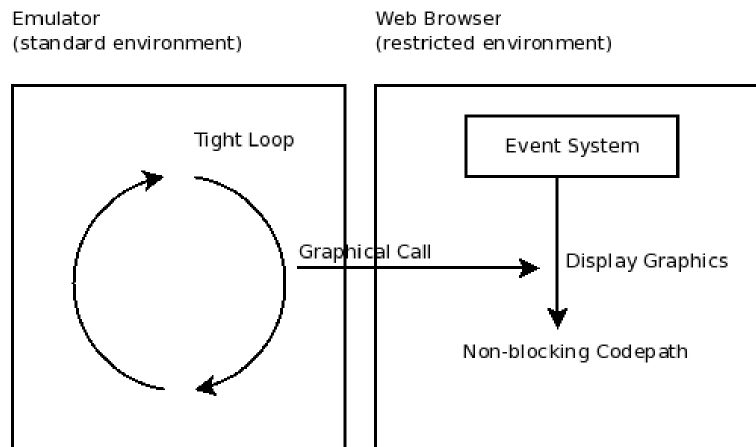

Figure 3.21: *Diagram demonstrating the separation between an environment where a tight loop is possible and one where code paths must complete so that execution can return to the web browser.*

Where work has been undertaken to improve stability is in how the data is transferred between model and view. Due to the limitation of certain web browsers, if too many messages are received the browser main thread will block during processing them and the run-time will freeze or abort the process. The most straight forward solution is to limit the amount of messages and wait for the browser to reply with a request for more work. However, due to knowledge of the OpenGL state machine being known at the protocol level, it has also been possible to modify the data as it is sent. Again, looking at **glTexImage2D**, it was possible to re-sample the image to send a very low fidelity version requiring very little data for the client to render almost immediately. After this, subsequent chunks of the full sized image can be sent per frame until the entire image has been sent and then the client will use that on future renders (**Figure 3.22**). Not only does this delayed loading system reduce spikes in bandwidth utilisation but it also creates a more seamless experience for web browsers, especially on low powered devices. Again, this functionality comes for free for any developer replacing OpenGL with Hydra as a drop in replacement.

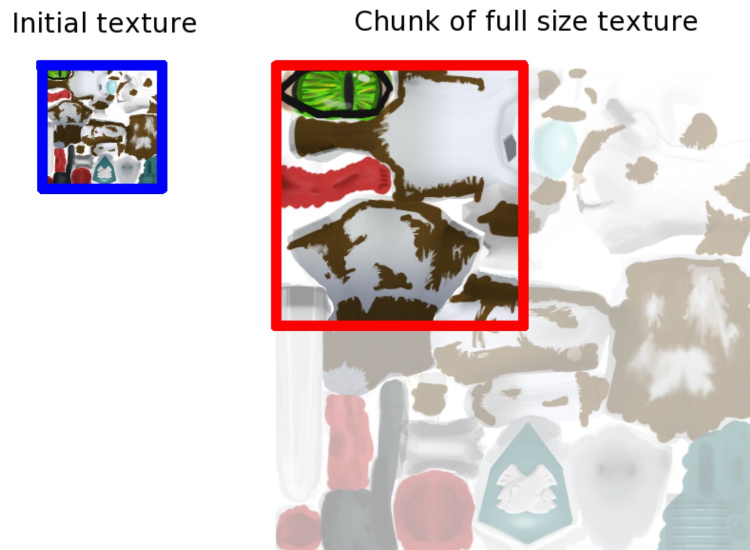

Figure 3.22: *Diagram demonstrating the type of image data that is uploaded via Hydra in order to cause the least amount of blocking.*

3.2.5 Avoiding Memory Errors

Being able to test code for errors on a platform is just as important as being able to develop and run the code in the first place (Lin et al., 2002; Cleraux and Perrin, 2011). Due to the nature of Hydra and the complexity of the problem it intends to solve, there is a very real risk of running into memory problems. Not only is this exasperated by the large number of messages and events it must handle but also the portability requirement of using C; a language with almost no safety guarantees.

The main reasons as to why C is seen as an unsafe language is because many of the tasks are manual and are left up to the programmer. The language makes no assumptions on what the developer intends to do. As mentioned in Section 3.1.4, there is no automatic memory management scheme such as RAII or garbage collection. It all has to be done manually. There are also no facilities for bounds checking when dealing with arrays. In particular, C does not necessarily have a concept of objects, much of it deals with blocks of memory, which happen to contain data resembling multiple instances of a structure.

The first class of errors that can result from using C consists of memory leaks. Due to the manual nature of C, the programmer must specify the absolute lifetime of heap data and when no longer in use, it must be explicitly freed. If the developer forgets to do this, a leak will result. Leaks are not particularly dangerous to the operation of a program; at worst they are quite wasteful. However, if a large number of these leaks occurs, the memory of a system can become depleted and no more usable memory will be available. Best case scenario, the operating system will provide memory from the pagefile and hard drive. However, this will be slow. Worst case scenario, an OS will provide no pagefile functionality, future allocations will return invalid memory and upon access, the program will crash if not properly handled.

Due to the nature of games, and the use of a very fast running main loop; if leaked memory is allocated and lost each frame, this will very quickly add up and memory will be depleted quickly. This is even more crucial in Hydra because a large number of messages and data is dealt with each frame so it is very important that this is correctly handled or the program utilising Hydra will not be able to run for long.

Traditionally these issues can be mostly found using memory testing tools such as Valgrind, however, in practice, it is quite common for both false negative and false positive reports to be given. In particular, it is very hard for such tools to know where in the program the leaked memory is emitted. When dealing with OpenGL, it can sometimes be observed that memory is leaked in the graphics driver layer, such as Mesa, and not in the actual program. Whilst Hydra can alleviate this problem slightly by actually avoiding the video driver layer, similar issues can occur in the sound system or the network layers. The largest issue with relying on tools like Valgrind (on Linux) or IBM Rational Purify (on Windows) is that a developer is starting to become constrained on the platform choice again. This goes against the essence of Hydra.

The second class of errors arising from using an unsafe language is much more damaging to the operation of a program. Memory corruption can occur via the

misuse of pointers. In particular, if memory referred to by a pointer is freed and later that same pointer is accessed, this can cause undefined behaviour and is known as a dangling pointer (**Figure 3.23**).

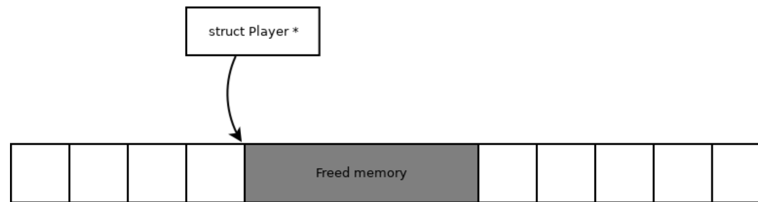

Figure 3.23: *Diagram demonstrating how memory is accessed and that pointers in C are capable of referring to previously freed memory.*

Further complications can occur if subsequent allocations of memory reuse the same data originally pointed to by that recently invalidated pointer; not only can the data referred to be of a completely different type but it may also not even point to the first element of a structure, meaning that any reading or writing that could occur after that point is likely to be reading from unreliable memory (**Figure 3.24**).

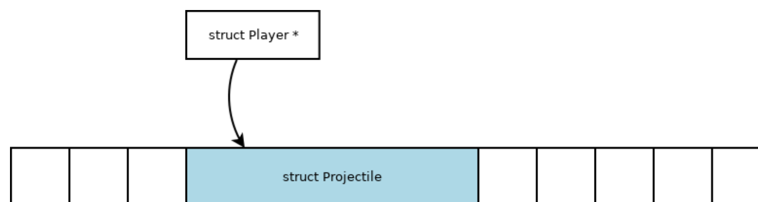

Figure 3.24: *Diagram showing that a subsequent freed block of memory could be later allocated as a different type. Note that the pointer is also referring to data within the middle of the block.*

In many cases, the program can continue but there are no guarantees that a crash will not occur. This non-deterministic nature of this class of error makes it extremely difficult to debug, especially given that the program will not necessarily crash at the same point that the error has occurred. The error causing the crash is usually just a symptom of memory corruption occurring elsewhere in the program.

In order to facilitate safer development with the C language a number of technologies

were developed and trialled. The first technology was an attempt at a portable garbage collector (libgc). Identified in Section 3.1.4, it was seen that garbage collectors were non-portable because they made assumptions about the platform; however, by making some compromises it was useful to see how far a garbage collection scheme could go before portability became constrained.

The portable garbage collector developed worked by simply scanning through the memory allocated, starting from a root block and seeing if any data values could represent pointer values to the first byte of existing valid blocks of memory. This block was then taken out of a remove list and the scan would continue. At the end of the scan, any blocks still remaining in the remove list were deemed unreachable and were freed. This idea is demonstrated in **Figure 3.25**. It worked in a serial nature, not taking advantage of parallelisation or multi-threading to increase performance in order to maintain simplicity and flexibility as a research prototype.

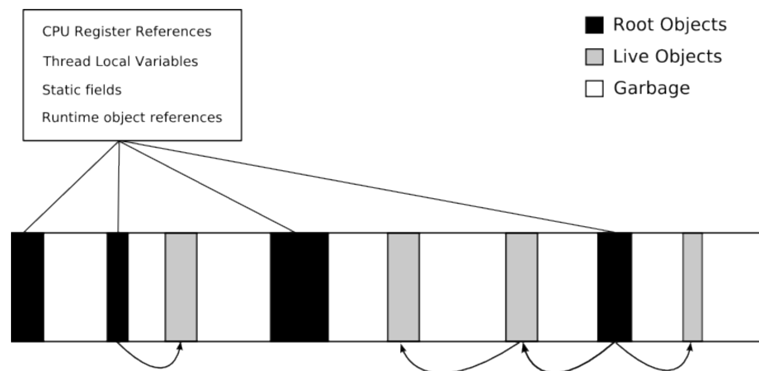

Figure 3.25: *Diagram showing the scanning of heap memory to ascertain which blocks of memory were no longer referenced.*

It was observed that heap memory was fairly easy and portable to scan byte by byte looking for active references also in the heap. This was a naive implementation but did not need to stop the world to do an entire scan, instead smaller re-entrant scans were performed to avoid large pauses. The performance was not up to the same quality as the main reference, Boehm's GC.

The main challenge arose when it came to scanning the stack memory. Attempts

were made to ascertain details of the stack using standard C. The theory is that if a variable is created on the stack early on in the program, its address could be found and tagged as the beginning of the stack. When a collect is made, a similar technique could be carried out to find the bottom of the stack and then from the top to the bottom the stack could be scanned for active references in the same way as the heap. A minor complexity here is that variables in C are not simply put in stack memory. Due to optimisation performed by the compiler, some variables may be kept in registers and so their address cannot be obtained. This complexity can be overcome fairly easily by marking the variable with the volatile keyword. The problem with this solution is that not all C compilers can support the volatile keyword in all standard configurations.

The most significant issue with this research garbage collector is that even though the top of the stack and bottom of the stack were obtained, an assumption is made; the stack is contiguous on the majority of platforms but not all of them (Feeley and Dubé, 2003). Because of this, the ability to scan the stack remains unsolved.

It is possible to avoid the need to scan the stack with small changes of architecture such as collecting the garbage at the highest possible level of the application (such as towards the end of the game loop) and avoiding variables on the stack up to that point, as well as pooling as much memory as possible within Hydra itself to overcome many of the performance issues with such a primitive collector. This allows the majority of the game to utilise the garbage collector but this solution is not necessarily trivial to implement for all application domains, detracting from the goal of platform agnostic development, so other methods of C safety were researched. Later methods focused on avoiding the need to have run-time checking within the release binary and, instead, a tool that could be used during development but then stripped out of the release build, reducing any overhead and portability issues. It was noted that Unreal Engine 4 uses a garbage collector. In order to track pointers they use a number of C++ functionality such as inheritance and RAII. RAII is especially important because it binds the life

cycle of a resource to the object on the stack (Schäling, 2011). Whilst this allows for knowing which objects to scan for active memory, it still appears redundant because RAII provides a very effective (although more complex) form of automatic memory management.

The second memory protection strategy (libstent) explored using a more mechanical approach to handling large amounts of memory. It can be likened most to the smart pointers in C++ since Technical Report 1 (TR1) (Austern, 2005). The smart pointer class; shared_ptr<T> works by utilising the unique feature of C++ called Resource Acquisition Is Initialisation (RAII) and binds object memory to the lifespan of the object with clear hooks on when the object is created and when the object is destroyed. In the language, these are called the constructor and the destructor.

With a shared pointer in particular, it resides on the stack and wraps data (as a pointer) which resides on the heap. When the smart pointer has its destructor called due to the variable going out of scope, either naturally or via the stack unwinding process due to an exception, it will in turn free the heap memory it is pointing to. Due to its shared nature, multiple shared pointers can point at a single block of memory and only when the last one goes out of scope will the data be deleted. This is carried out internally via reference counting.

There is a counterpart to shared pointers called weak_ptr<T>. These weak pointers work in the same way as shared pointers but do not increment the reference counter. When the original shared pointer goes out of scope, the data is destroyed. Most importantly however is that the weak pointer is not simply left pointing at invalid memory (dangling) but is put into an expired state. This means that due to erroneous programming, if an expired weak pointer is accessed, a clean crash can be expected at the source of the problem. The debugger can then output the stack trace and the developer can fix the issue.

This functionality is extremely desirable but does not exist in C. For example, C provides no RAII functionality, making concepts such as smart pointers more complex

to implement. The GNU C Compiler provides a non-standard extension to call a destructor when an object goes out of scope (GNU Compiler Collection, 2019 (accessed April 3, 2019)) but due to portability issues with other compilers, this was not an option here.

The approach taken by libstent was to classify the automatic memory cleanup as out of scope for the project and, instead, focused on the weak pointer functionality.

The notion of "Fat Pointers" was used, which meant that rather than using a raw pointer, a struct on the stack was used which not only wrapped the raw pointer but also provided a unique ID. This unique ID was made up of the current time in seconds and an additional number variable called **instance**, which would increment with each allocation that second. This structure can be seen in **Figure 3.26**.

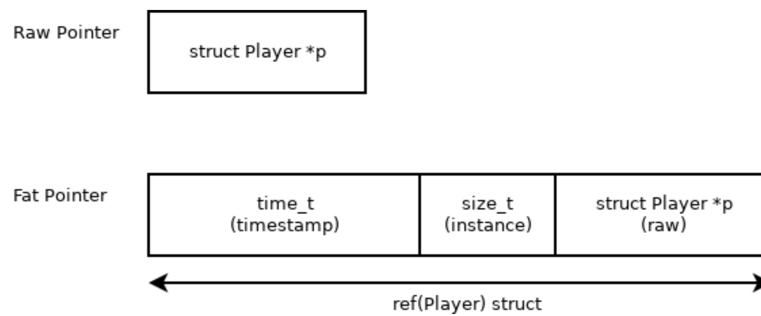

Figure 3.26: *Diagram showing the data making up a fat pointer in comparison to a standard raw pointer. Note that the fat pointer is much larger than the raw equivalent.*

In a similar way to C++, a raw pointer cannot be extracted from the weak pointer directly. Instead, a member function called **lock** must be called which returns the raw pointer. This is mirrored in this C implementation by requiring the **get** function. However, internally both systems differ. In libstent, the **get** function goes through a list of all allocated memory using **stent_alloc** and checks if there is one with a matching time, instance and raw pointer variable. If there is not one with those matching identifiers, an abort is called. A sample of code utilising libstent can be seen in **Figure 3.27**.

```c
#include <stent.h>
#include <stdio.h>

struct Bomb
{
  int fuse;
};

void BombTick(ref(Bomb) b)
{
  if(_(b)->fuse < 1)
  {
    printf("Boom!\n");
    sfree(b);
  }

  _(b)->fuse--; // Use after free
}

int main()
{
  ref(Bomb) bomb = NULL;

  bomb = salloc(Bomb);
  _(bomb)->fuse = 2;

  BombTick(bomb); // Fine
  BombTick(bomb); // Fine
  BombTick(bomb); // Crash

  return 0
}
```

Figure 3.27: *An example of libstent showing a fairly typical case of a use after free programming error.*

The programming error in this case is fairly easy to find. The issue is that the fuse timer is decremented after the object has potentially been destroyed. However, in a larger project involving a large number of nested functions, it can sometimes be more difficult to know exactly which code path frees a specific piece of memory. For this reason, libstent provides a useful error message as soon as this type of code is executed. The respective error message for this code can be seen in **Figure 3.28**.

```
Warning: Debug memory allocator enabled
Boom!
Error: Pointer to memory [Bomb] no longer valid [bomb.c:17]
Abort trap (core dumped)
```

Figure 3.28: *The resulting error message for the previous erroneous program. Note that not only is the source unit file and line number exposed; but also the type of structure.*

Another fairly common error made is when dealing with collections of objects. For example, **Figure 3.29** demonstrates some code which intends to create multiple objects and then simply destroy them.

```c
#include <stent.h>
#include <stdio.h>

struct Bomb
{
  int fuse;
};

int main()
{
  int bi = 0;
  vector(ref(Bomb)) bombs = NULL;

  bombs = vector_new(ref(Bomb));

  for(bi = 0; bi < 10; bi++)
  {
    ref(Bomb) b = salloc(Bomb);
    vector_push_back(bombs, b);
  }

  for(bi = 0; bi < vector_size(bombs); bi++)
  {
    sfree(vector_at(bombs, bi));
    vector_erase(bombs, bi, 1);
  }

  vector_delete(bombs);

  return 0;
}
```

Figure 3.29: *Code to create and destroy objects showing a subtle memory leak.*

This mistake is a more subtle one. As the destroyed object is removed from the

collection, the next object is shuffled down whereas the current index is incremented. This in turn means that every second object is skipped. The debug message returned from libstent (**Figure 3.30**) gives a clear indication of the problem.

```
1 Warning: Debug memory allocator enabled
2 Warning: Allocated memory [bombs.c:18] persisted after exit [Bomb]
3 Warning: Allocated memory [bombs.c:18] persisted after exit [Bomb]
4 Warning: Allocated memory [bombs.c:18] persisted after exit [Bomb]
5 Warning: Allocated memory [bombs.c:18] persisted after exit [Bomb]
6 Warning: Allocated memory [bombs.c:18] persisted after exit [Bomb]
```

Figure 3.30: *The resulting warning message for the previous leaking program. Note that the source unit and line number where the memory was allocated is also exposed along with the type.*

An abort was used for a variety of reasons rather than an exception. The main reason against an exception is that they do not exist in standard C and whilst they could be emulated using a complex system of **setjmp** and **longjmp** this idea would be flawed. The nature of these errors is not something that can or should be handled under exceptional circumstances. They are caused entirely by programmer error and by aborting the program at the debugging stage, the programmer can fix the underlying problem. They are more akin to assertions to help develop correct code rather than to be used as error handlers in the release build.

libstent was used in a number of commercial and research development projects including the server for a VR architectural simulation called QuickVR, a training simulator for an AI development competition called Formula Pi as well as an analytics library for Unreal Engine 4, as part of a University research platform initiative called BU Games. It was also used for the development of the Tank Museum Gun Game (TMGG), a game for the Dorset Tank Museum in Bovington which uses an innovative system involving a standard off-the-shelf webcam and a real-world decommissioned rifle to ascertain where on the screen a player is attempting to aim (**Figure 3.31**). This yielded very good results and due to the nature of it being an arcade kiosk, it

had to run from morning to evening, every day of the week. This meant that memory leaks, corruption and other programming issues throughout the day would become a considerable problem. The Tank Museum is currently running three of these arcade cabinets (each with a different gun from varying eras). The analytics show that each cabinet receives approximately 50 plays a day and there have been no reports of crashes (a common symptom of memory problems).

Figure 3.31: *The Tank Museum Gun Game was an arcade kiosk system where guests to the Tank Museum could utilise real weapons to shoot at targets on the screen. The hardware involved requires nothing more than a webcam for tracking and a serial cable for the trigger.*

The use of libstent in these projects was evaluated and it has performed well; it maintained almost identical performance of the unharnessed alternative, even when dealing with a large number of requests. However, there were some issues with the technology. The first one was that it was difficult to define fat pointers in the same way as the raw pointer counterpart. The full implementation of the fat pointer had to be known before hand, which made the opaque pointer pattern (Cacho et al., 2006)

that makes C a powerful cross platform language much harder to exploit. This could be overcome but was a little unwieldy and verbose; also contributing to long compile times. With the relatively smaller projects, this was not an issue but it became a hindrance on more moderate codebases. The non-standard foreign syntax required by libstent also meant that future maintenance of any code would be difficult. The second issue was that in some situations, especially when dealing with many small allocations, the approach taken by libstent would consume considerably more memory.
The third memory protection strategy took a slightly different approach. Rather than any checks or protection happening at run-time in the release build, all the checks should instead be carried out during the development / debug stages. This effectively eliminated any overhead in the final build, whilst also ensuring that the C generated was also 100% valid ANSI C, which would not hinder portability or maintenance.
This strategy ultimately took advantage of the Memory Management Unit (MMU) on a processor. With this hardware, more knowledge on the memory could be known and controlled. This meant that there was some compromise on portability on the wide range of test platforms identified. One of the older operating systems used (MS-DOS) had access to the MMU using the DJGPP compiler (based on GNU C Compiler) utilising the **mprotect** system call. Later, this same call was standardised into the POSIX standard (Issue 4, Version 2) (Walli, 1995) and thus is available on any standards-conforming operating system. For non-conforming systems such as Microsoft Windows, there is often a similarly achieving alternative via **VirtualProtect**.
However, importantly, this requirement on an MMU remains optional as a development requirement, not a release requirement. It is only used as a debugging aid during development. Specifically, this is important if targeting an embedded platform with no physical MMU. If there was a hard dependency on an MMU it would pose significant porting issues such as those encountered by Filardo (2007) when attempting to port QEMU to Plan 9. By keeping it as an optional dependency,

a release build can still be compiled for that platform but without the additional debug memory checking.

This strategy was implemented in a debugging library called libplank. The underlying process of how it works was that memory was allocated and when no longer required, rather than freeing it, it would use **mprotect** to lock the memory and render it inaccessible. If this memory was used again, either by a dangling pointer or overflow of an array, as soon as the memory was read or written to, it would cause an abort. This process is explained in **Figure 3.32**. This control of the memory by the MMU was extremely powerful.

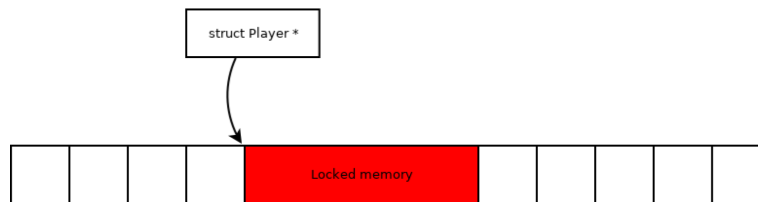

Figure 3.32: *Diagram showing the locked memory block. Note that it is still allocated and is taking up memory, however, accessing it will cause the program an immediate abort.*

Once the program was tested and no aborts were found due to memory errors, libplank would simply be disabled and the memory would be allocated and freed as usual. This is particularly important for release builds because libplank effectively causes a memory leak on purpose due to the fact that it never frees memory, it only locks it (instead) to prevent its use in subsequent allocations.

libplank has been successfully used on a number of other University-related medium scale projects including Zombie Maths Game (ZMG) (**Figure 3.33**), a 3D game used to help children practice their maths skills and mental arithmetic. It is modelled 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. Not only did the debug library yield a high success rate on detecting the source of dangling pointers causing any crashes but it also exhibited no additional

overhead in the release build because it simply was not present and had been stripped out.

Figure 3.33: *Gameplay of Zombie Maths Game. A game for helping to teach children mental arithmetic in a fun and engaging way. This game was presented to the public at the 2017 BU Science Tent with support from the British Science Association and Siemens UK.*

The libplank memory strategy showed promising results in all but one area; portability. It was deemed that this strategy would be suitable for the development of Hydra on current and existing workstations but there is simply no way of knowing what hardware and operating system configurations will be present in the future. There is no reason to suspect that the expensive, high power consuming development workstations of today, needed to run heavyweight software suites such as Unreal Engine 4's Editor and the Unity Editor will exist in the same way in the years to come.

Evidence from the past helps confirm this notion when looking at the old DOS series of platforms such as MS-DOS or IBM PC-DOS and their access to the MMU, even though they were still very primitive. It is particularly interesting to compare this to the much newer operating system Plan 9 (a successor to UNIX), where there was

no functionality for virtual memory via the MMU. Whilst UNIX had the ability to lock virtual memory via the POSIX **mprotect** call it was decided that this was not actually in line with the goals of AT&T's future operating systems. So whilst DOS was much more primitive than Plan 9; in this one specific area Plan 9 could almost be seen as a regression.

Conjecture suggests that with the move to cloud services, if development tools of the future mean that we design, develop and test much of the software in a web browser, then this platform does not provide virtual memory for sandbox and security purposes and potentially finding these memory issues present will rely on the need of live testing using device specific hardware, which will make the development process awkward and unfeasible.

Realising that there may be no alternative, the ideas presented by the libstent architecture were deemed most suitable for the development of Hydra. Many of the issues with using fat pointers were resolved using a number of patterns making heavy use of MACROs in the preprocessing stages (this is discussed further in Section 3.2.7).

3.2.6 Testing Different Memory Strategies

All three memory strategies were measured in terms of effectiveness at detecting memory issues. They were added to the codebase of a third program using Hydra at three separate periods during its development. The program consisted of a VR training tool to help users learn musical intervals using spatial audio (**Figure 3.34**).

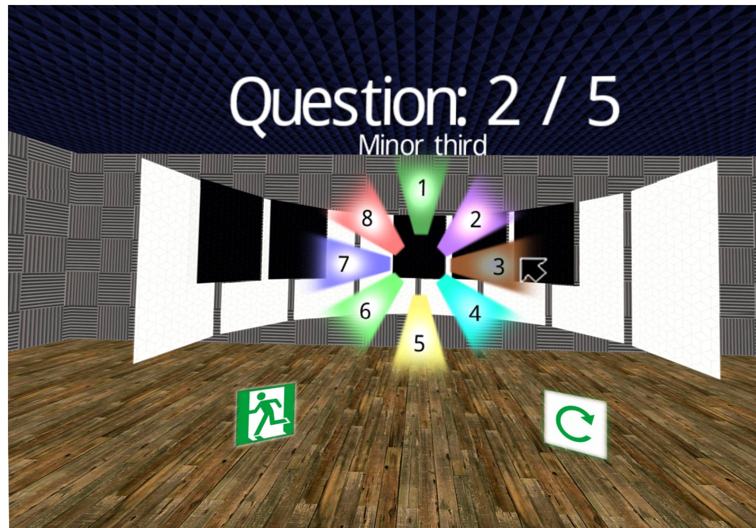

Figure 3.34: *Screenshot showing the VR Spatial Audio tool. We aim to publish a paper using the results gathered from this tool in the future.*

Even though this was a VR program utilising a headset rather than conventional 3D program, it still provided a useful experimentation platform for these different memory strategies. It also had the added benefit that it was designed to be very linear in terms of use so it would reduce the risk of faulty code paths being skipped. This is hoped to serve as a more deterministic testing platform and help reproducability of memory errors.

The aim of this experiment was to find out which memory strategy provided the largest coverage of reported errors from those detected. It was prepared with the following objectives:

- **Augment an existing program with a number of different memory strategies**, including libstent, libplank and libgc. This will allow for testing to discover which is the most appropriate in terms of integration but also which can report the highest number of errors.
- **Identify a range of possible errors within the codebase** using all techniques, including Valgrind and VTune and record them. This will then serve as the set of tests that need to pass by each strategy. This is particularly

useful because the different strategies expose errors that others did not.

- **Measure the percentage of these passing tests from each strategy**. This will give an overview of how likely a memory strategy can detect errors.

The chosen platform for this experiment was Debian 9 on the AMD64 architecture. This choice was not only due to the requirements of VR hardware and drivers for this specific project but also by the range of memory testing tools. Valgrind in particular does not support Windows and was seen as a valuable addition to the set of tests. It is fairly typical that different platforms can expose different memory errors in a program and in future it will be useful to perform a similar test on a number of alternatives. However for this test, limiting to just one platform does constrain the number of potential variables for the test which should result in more reliable results. With the memory strategies requiring manual instrumentation of the code in place, the program was run through to completion five times for each of the different strategies.

The number of detected issues, including those from conventional tools such as Valgrind and Intel Inspector (VTune) are shown in **Figure 3.35**.

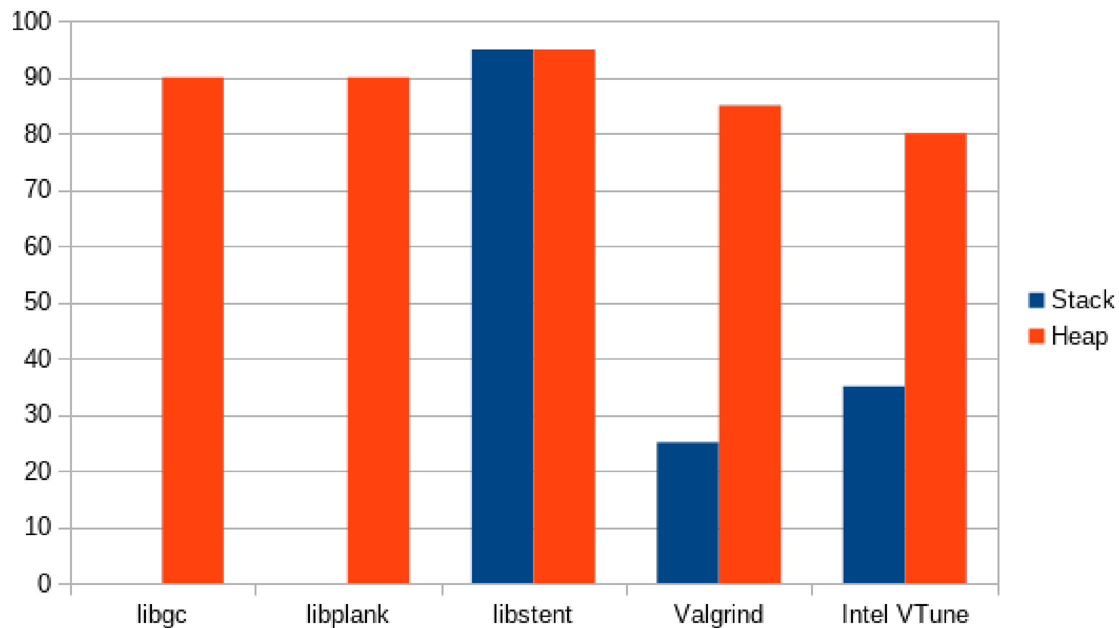

Figure 3.35: *Diagram showing the percentage of errors reported by specific technologies. Lower numbers show that the technology failed to expose a specific programming error. Note that although libstent was close, none of the solutions provided 100% coverage. There were always errors missed that other strategies had uncovered.*

The results show a positive outcome from many of the memory safety strategies. In particular, libstent has showed the highest effectiveness; even beating Valgrind in some aspects. It should be noted here that the discrepancy between Valgrind and libstent is due to the checking of stack memory; its main focus is heap memory and offers only experimental facilities for checking stack memory. The garbage collector approach with libgc and the memory locking approach with libplank also lacked effective stack checking functionality so also reported a number of false negatives in these tests. The development fat pointer approach with libstent yielded much closer results to Intel Inspector which does offer fairly good stack checking. That said, it is not perfect, the Intel Inspector tool in the past has reported a false negative with certain stack problems so, as always, all of these tools can only report errors, they cannot report the absence of any errors. What is particularly noteworthy with libstent is that it

has yielded comparable results to the commercial tool by Intel whilst also remaining completely platform agnostic and thus is able to run on constrained platforms such as MS-DOS, Plan 9 and HTML5. That said, all three of the memory testing strategies have performed well. It was noted that the majority of errors have come from the use of third party libraries. This is because these libraries are not using the memory testing instrumentation and, as such, this is where issues have been able to arise. It should also be noted that libstent has a very restrictive approach to using the stack which in some cases may be too limited to use within a number of very specific embedded projects. This restrictive nature was necessary to provide the required safety.

In conclusion, the objectives have been met and even though no memory strategy achieved 100% success in terms of detecting memory errors, libstent was identified to be the closest to achieving this. However part of this success is due the very restrictive access to the stack which does have some impact on integration and performance. It should also be noted here that the 100% goal could arguably not represent the total number of errors in the program. It is very possible that all of these tools combined had missed a number of errors or that the defective code-path was simply not executed during the testing phase. The introduction of static analysing tools could be useful here. As discussed, further testing on alternative platforms could also help to expose additional errors. However these results have still been very useful in identifying libstent as a useful memory strategy for Hydra.

The listing in **Figure 3.36** below showing tests with and without libstent further demonstrates why an error checking tool has been crucial to the project. Without this error checking library, many unintentional programming errors would make their way into the codebase and slow down development time attempting to find and eliminate them. With a more strict approach to error checking, these errors can be eliminated shortly after they have been added and much more progress could be made. Most importantly however is that improving the safety of the C programming language

whilst allowing it to retain its strengths in terms of portability is a key part to writing platform agnostic software. So far libstent fills in this missing piece.

```
*** Running ctest with STENT_ENABLE ***

Test project /home/kpedersen/Projects/stent/build
     Start  1: ref
 1/17 Test #1: ref .............................. Passed 0.01 sec
     Start  2: ref_copy
 2/17 Test #2: ref_copy ......................... Passed 0.01 sec
     Start  3: copy
 3/17 Test #3: copy ............................. Passed 0.01 sec
     Start  4: zero_initialized
 4/17 Test #4: zero_initialized ................. Passed 0.01 sec
     Start  5: void_cast
 5/17 Test #5: void_cast ........................ Passed 0.01 sec
     Start  6: vector
 6/17 Test #6: vector ........................... Passed 0.01 sec
     Start  7: fstream
 7/17 Test #7: fstream .......................... Passed 0.05 sec
     Start  8: dangling_ref
 8/17 Test #8: dangling_ref ..................... Passed 0.02 sec
     Start  9: dangling_ref_copy
 9/17 Test #9: dangling_ref_copy ................ Passed 0.02 sec
     Start 10: invalid_cast
10/17 Test #10: invalid_cast ..................... Passed 0.02 sec
     Start 11: invalid_void_cast
11/17 Test #11: invalid_void_cast ................ Passed 0.02 sec
     Start 12: use_null
12/17 Test #12: use_null ......................... Passed 0.02 sec
     Start 13: release_null
13/17 Test #13: release_null ..................... Passed 0.02 sec
     Start 14: leak
14/17 Test #14: leak ............................. Passed 0.02 sec
     Start 15: vector_oob
15/17 Test #15: vector_oob ....................... Passed 0.01 sec
     Start 16: vector_use_null
16/17 Test #16: vector_use_null .................. Passed 0.01 sec
     Start 17: vector_dangling_ref
17/17 Test #17: vector_dangling_ref .............. Passed 0.01 sec

100% tests passed, 0 tests failed out of 17
```

Figure 3.36: *A small sample of tests showing results with libstent enabled. The 100% pass rate shows that if the test contained an artificial memory error, it was correctly recognised and flagged by libstent.*

```
*** Running ctest with STENT_DISABLE ***

Total Test time (real) = 0.30 sec
Test project /home/kpedersen/Projects/stent/build
      Start  1: ref
 1/17 Test  #1: ref .............................. Passed 0.01 sec
      Start  2: ref_copy
 2/17 Test  #2: ref_copy ......................... Passed 0.01 sec
      Start  3: copy
 3/17 Test  #3: copy ............................. Passed 0.01 sec
      Start  4: zero_initialized
 4/17 Test  #4: zero_initialized ................. Passed 0.01 sec
      Start  5: void_cast
 5/17 Test  #5: void_cast ........................ Passed 0.01 sec
      Start  6: vector
 6/17 Test  #6: vector ........................... Passed 0.01 sec
      Start  7: fstream
 7/17 Test  #7: fstream .......................... Passed 0.02 sec
      Start  8: dangling_ref
 8/17 Test  #8: dangling_ref .....................***Failed 0.01 sec
      Start  9: dangling_ref_copy
 9/17 Test  #9: dangling_ref_copy ................***Failed 0.01 sec
      Start 10: invalid_cast
10/17 Test #10: invalid_cast .....................***Failed 0.01 sec
      Start 11: invalid_void_cast
11/17 Test #11: invalid_void_cast ................***Failed 0.01 sec
      Start 12: use_null
12/17 Test #12: use_null ......................... Passed 0.01 sec
      Start 13: release_null
13/17 Test #13: release_null .....................***Failed 0.01 sec
      Start 14: leak
14/17 Test #14: leak .............................***Failed 0.01 sec
      Start 15: vector_oob
15/17 Test #15: vector_oob ....................... Passed 0.01 sec
      Start 16: vector_use_null
16/17 Test #16: vector_use_null .................. Passed 0.02 sec
      Start 17: vector_dangling_ref
17/17 Test #17: vector_dangling_ref ..............***Stalled -.-- sec

58% tests passed, 7 tests failed out of 17
```

Figure 3.37: *A small sample of tests showing results without libstent enabled. These failures all represent major errors in programming code that would not immediately terminate the program and as such could be very hard to diagnose.*

Contrasting this to the results of the same tests but without enabling libstent, shown in **Figure 3.37**. These results show that a number of errors did not get flagged and the program ran to completion. Programming errors such as these can have significant negative effect on the operation of the program and yet be very difficult to discover because they don't cause a crash at the immediate time that they occur.

This is well demonstrated by test #17. This stalled the program in an infinite loop rather than either crashing or completing. This is because after memory was deleted, it was subsequently accessed. Had libstent be enabled at this point, it would immediate terminate with an error message (as in **Figure 3.36**). However instead invalid memory was accessed which happened to be zeroed out on the OpenBSD platform. **Figure 3.38** is a simplified listing of the type of code that was negatively impacted after the initial use after free error. If the list's size was anything but 0, it would not have resulted in an infinite loop.

```
void po2_resize(struct list *list /* freed */, size_t size)
{
  size_t s = 0;

  s = list->size; // Invalid memory access

  while(1)
  {
    if(s >= size)
    {
      break;
    }

    s = s * 2;
  }

  // ...
}
```

Figure 3.38: *A simple listing to demonstrate how accessing invalid memory can cause undefined results later on in the program.*

3.2.7 Technical Details Behind libstent

As discussed, ANSI C is one of the most crucial components to cross platform development. Unfortunately it was not designed with safety in mind and due to the nature of the specific problem of digital preservation it is not a viable option to extend the compiler in non-standard ways in order to add this safety (Kowshik et al., 2002). At the same time, it is very important for the uptake and maintenance of Hydra that the safety checking is not invasive or awkward to work with.

The early design of libstent required a “fat pointer” to be defined in every compilation unit of the project, making it hard to take advantage of forward declaring and private implementations. This was resolved by adding an additional layer of indirection so that a standard pointer is used to point to the fat pointer which finally points to the allocation. This may sound inefficient which certainly is the case. Some measurements show an overhead as high as 10% for tasks involving non-contiguous data sets such as traversing linked lists. However, with the introduction of selectively activating stent in debug mode, this has become much less of an issue. For example the overhead is only present during debugging and then once libstent is disabled within a release build there is zero overhead. Due to the fact that raw pointers are used, there is even less overhead than the C++ smart pointer approach to memory correctness.
One area that had to be resolved for the thin-to-fat-pointer approach was keeping type safety. Type safety is important to avoid data being used incorrectly. For example, as **Figure 3.39** demonstrates, if data containing information relating to a player is being used by code that is expecting weapon related data, there is going to be a number of issues. Especially if data types do not match; such as writing numeric values into text variables.

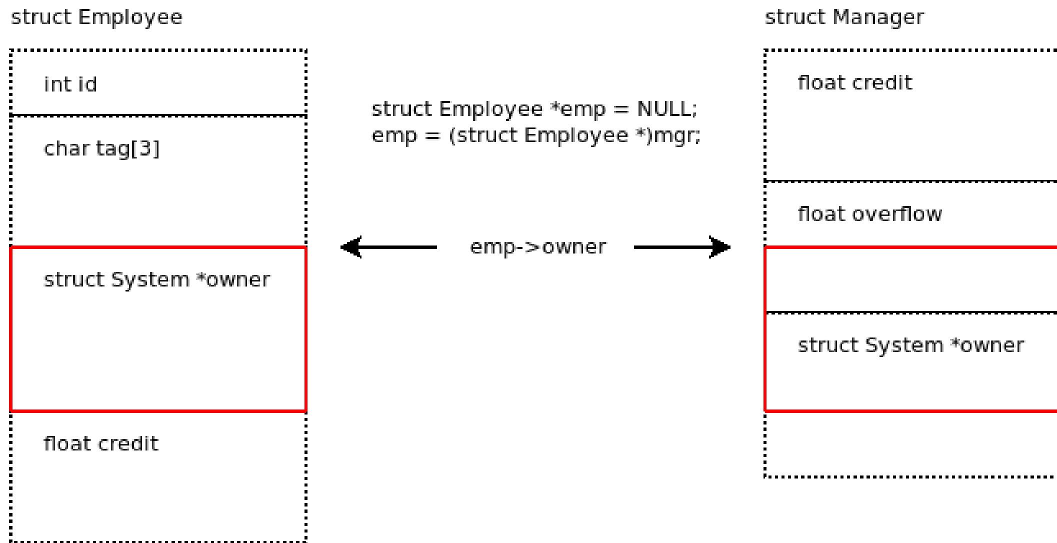

Figure 3.39: *Diagram demonstrating how type safety is an important factor in keeping data consistent. Note that there is no guarantee that the referred data is aligned to a single type and may be misframed over other fields.*

The issue arises that in order for libstent to keep an internal record of an allocation, it needs to have a reference to the allocated memory. This, as shown in **Figure 3.40**, is stored as a *void ** (or a *char ** on pre ANSI C compilers). This means information has been lost regarding its type. In the old system, a specific fat pointer was defined to point to a specific type but that is no longer possible in the same way when going from the thin pointer.

```c
/***********************************************************************
 * Allocation
 *
 * Structure containing information about individual allocations. The ptr
 * must be the first element to allow the additional indirection of the
 * type-safe reference to work. The memory pointed to by ptr may get freed
 * but the Allocation memory itself persists throughout the lifespan of
 * the program.
 ***********************************************************************/
struct Allocation
{
  void *ptr;       /* Pointer to the native C memory block */
  int expired;     /* Track whether allocation has been freed */
  const char *type; /* Specified type for run-time type identification */
};
```

Figure 3.40: *Listing of the Allocation structure used internally within libstent. Note that the ptr field is the first one.*

Instead, a different approach was taken. Rather than using a standard pointer to the allocation struct (such as *struct Allocation *player* or *struct Player *player*), an additional level of indirection was added. So instead a pointer to a Player pointer to the same allocation was used (so now a *struct Player **player*). The most important part to note is that within the allocation structure, the *void *ptr* field is the first one. This means that the location pointed to it will be used when player is dereferences (either via ***player* or *player[0][0]*). It will work in exactly the same was as if the allocation had a *struct Player *ptr* as the first field.

Accessing memory in this way is completely safe and correct when it comes to the ANSI C standard but it is of course very prone to error so needed to be implemented carefully. Not only has a fairly substantial test suite been developed for libstent but most importantly the user of libstent is completely protected from the complexities such as this. Their code can be much more straightforward and verifiable as a result.

The next challenge arose due to the fact that in order for errors such as use-after-free or memory leaks to be reported, the allocation must be recorded through the lifetime of the program. This meant that once memory is allocated, it must not be released so that it can be flagged as deleted and so that it doesn't become reused (potentially

as a different data type). Even on modern systems this can cause an exhaustion of memory. What is interesting is that this is not typically caused by using too much memory (because memory is a fairly inexpensive resource) but instead by running out of the number of possible allocations. This is because each allocation has some amount of overhead and the registry is also limited in size specifically on 16-bit and 32-bit platforms. Even though Physical Address Extension (PAE) in modern kernels allow 32-bit operating systems to allocate more than 4GB in total, the maximum per process is still limited to around 4GB and the registry, again, is limited to a maximum number of entries. If software allocates and deallocates a lot of memory when libstent is enabled; it will fairly quickly run into issues. 64-bit platforms are not completely immune to this problem either, most operating systems, especially those with a focus on servers have specific limits on how much memory certain processes can allocate.

To overcome this problem, it was decided to build upon the extra level of indirection to the "fat pointer" and allow the system to delete the allocated memory whilst leaving the fat pointer as the history of what has been allocated, freed and potentially a dangling pointer. Even though this greatly reduced memory usage, it still had the potential to cause memory exhaustion due to the maximum number of allocations on a 32-bit system. So as a further improvement rather than allocating individual fat pointers, larger chunks would be used (defaulting to enough space for 2048 fat pointers). This can be seen in **Figure 3.41** and meant that even after 2048 allocations and frees there was no memory consumed other than one single allocation with a size of *2048 * sizeof(struct Allocation)* which was required to act as the history for the old allocations.

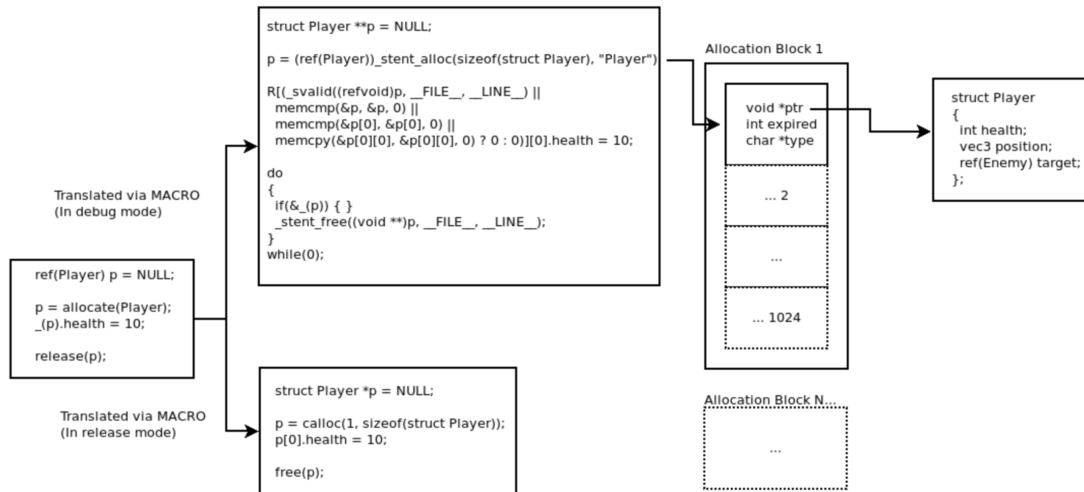

Figure 3.41: *Diagram demonstrating the approach taken by libstent to provide meta-data on individual allocations.*

In practice, and even though libstent would be disabled for a release build anyway; it was extremely unlikely that memory would be exhausted in a typical and correctly functioning program. Even when the 16-bit Windows 3.1 was being used as a test-bed for Hydra, there were no issues encountered which were caused by memory exhaustion due to libstent's debug memory allocation mechanism. The most typical symptom of memory exhaustion is simply that future calls to memory allocation routines such as *malloc()* or *calloc()* return *NULL*. A programmer should check for this and can often make the assumption that the program is in the out of memory state.

Dealing with dynamic arrays of memory in C is another potentially error prone task. To minimise the risk of memory errors, libstent also provides functionality akin to *std::vector* in C++. Again, in order to enable type-safety, a similar design pattern is used (shown in **Figure 3.42**).

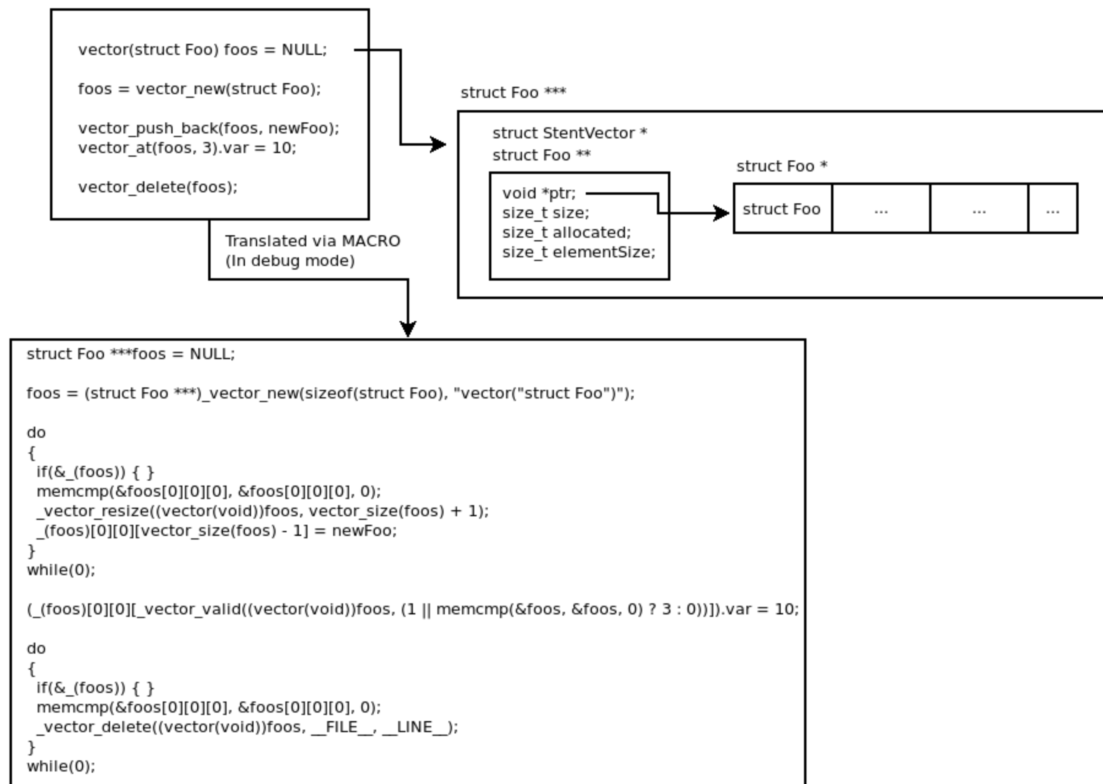

Figure 3.42: *Diagram demonstrating the libstent vector architecture. Note that a similar pattern has been used to create contiguous dynamic arrays.*

By ensuring that the pointer to the raw memory is the first field within the **StentVector** structure, a reference to it can be given an additional level of indirection and the data can be accessed in not only a type-safe manner but also with additional bounds checking. In particular the code listing above will provide the following error.

```
Error: Index [index=3] out of bounds [size=1] in main.c:6
Abort trap (core dumped)
```

It can be seen within the generated C code from the previous diagrams that there are a number of seemingly unused functions or branches. These are simply intended to provide an additional number of safety checks within debug mode. For example they test the assertion that the provided pointer is indeed that of a stent vector (it provides the correct level of indirection) and that it is not an L-value so that in rare cases,

the MACRO can refer to the argument multiple times without potentially calling a function or executing an expression multiple times.

3.2.8 Object-orientation in C

ANSI C does not specify any specific programming paradigm such as procedural, functional or object-oriented. However there is good evidence that an object-oriented programming (OOP) approach can be easier to understand and maintain, leading to their popularity within the industry (Kölling and Rosenberg, 1996). Kölling and Rosenberg (1996), make a specific note that C++ is a more popular language than C because it has OOP concepts built into it; such as classes, inheritance, etc. However even in 2019, the TIOBE index (TIOBE Group, 2019 (accessed July 9, 2019)) for ranking the popularity of programming languages has demonstrated that C is still at a higher rank than C++ (with 3x a higher score). The suggested reason for this is due to the popularity of C as an embedded language, including its very high portability. However, in practice it is often the case that object oriented code can also be implemented in a very effective way simply by using C in a suitably disciplined manner. For one example, this is demonstrated well in the popular open-source GUI library GTK+. Written entirely in C, it provides almost all functionality in an object-oriented manner (Wright, 2000).
As a very simple demonstration; the following listing (**Figure 3.43**) shows how C can be used in an object-oriented way compared to traditional C++ and modern C++.

```
Player *player = NULL; // c++03
std::shared_ptr<Player> player; // c++11
struct Player *player = NULL; /* c89 */

player = new Player(); // c++03
player = std::make_shared<Player>(); // c++11
player = PlayerCreate(); /* c89 */

player->jump(); // c++03
player->jump(); // c++11
PlayerJump(player); /* c89 */

delete player; // c++03
/* RAII: implicitly deleted */ // c++11
PlayerDestroy(player); // c89
```

Figure 3.43: *Listing to code in traditional C++, modern C++ and C showing the subtle differences in object-oriented approaches. It is also interesting to note how much C++ has changed over time compared to ANSI C.*

A very powerful feature of designing software in an object oriented manner is there are a number of consistent was of documented the architecture. One of the more popular ways is via a class diagram. **Figure 3.44** demonstrates how the objects involved in the previous code example might be structured.

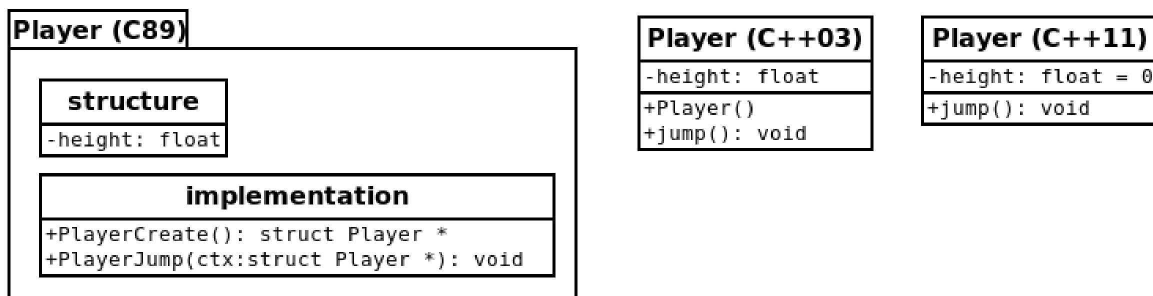

Figure 3.44: *Class diagram showing a comparison between the approaches taken to implement an object in C and C++. It could be considered based on this example that the most modern C++ language features makes this task simpler.*

One area where C potentially out performs C++ when it comes to object-oriented programming is information hiding. Whilst it is true that C++ provides this feature as part of the language (Pokkunuri, 1989), there are some issues inherent with

it. For example, whilst fields can be marked as private or protected within an object, the entire class definition still needs to be visible (within a header file) to the rest of the program. Contrasting this to a typical opaque pointer in C where only the implementations of the functions (not forward declarations) need knowledge of the internals of a structure. Therefore they can be defined within the same compilation unit as the structure definition and can operate on the data referenced by the pointer. The internal details of the structure can be hidden from everything else. This can potentially provide a more mechanical way of hiding complex implementation details whilst providing a number of other benefits such as faster compilation because complex structure implementation can be ignored for a larger proportion of the codebase. The solution to this in C++ is the PIMPL (Pointer to IMPLementation) pattern (Lischner, 2009). However not only does it have some (albeit, somewhat trivial) amount of performance overhead but it also requires a fair amount of boilerplate and repetitive code to be written for each object. This is shown in **Figure 3.45**.

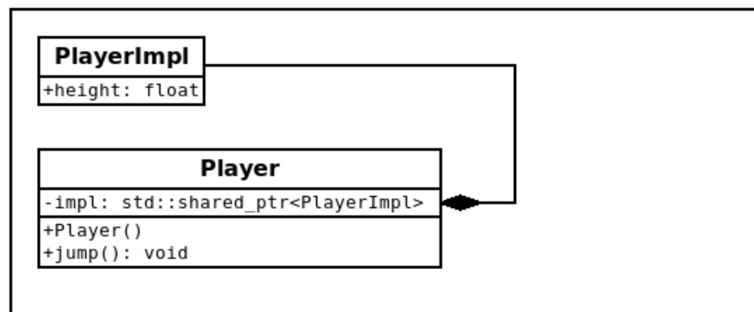

Figure 3.45: *Diagram showing the PIMPL pattern applied to the previous classes. Note that it is now showing many similarities to the C89 approach.*

In practice, the increased verbosity of the codebase is not the primary issue involved in the use of the PIMPL pattern, it also damages a number of other C++ object-oriented mechanisms, such as the ability to use weak pointers (to avoid cyclic references), factory classes and inheritance. However it is still one of the most popular ways to design C++ software due to the extra level of information hiding. This provides a

fairly good justification for using C to leverage this functionality in a more natural way.

3.2.9 WebSocket Implementation

During the development of the client (which was intended to connect to Hydra to perform the native rendering) a choice of technology was required. Due to the client remaining as simple as possible, portability of the client was not much of a concern compared to Hydra itself so after a small survey of the current environment it was deemed that using a HTML 5 web browser was a suitable choice. It also provided an opportunity to perform some additional testing and validation of the solution because not only is a web browser a fairly restrictive and sandboxed environment but also has a fairly different asynchronous and single-threaded design and architecture to most other platforms. In order to maintain a platform agnostic approach and minimise unnecessary dependencies, whilst at the same time allowing for communication with a modern web browser via WebSockets, a bespoke WebSocket library was needed. Alternative implementations were evaluated but they were either written using very large platforms (such as the Java Runtime Environment or .NET Framework) which rely on platform specific VMs to function. Those that were written using C or C++ often took advantage of the very latest language features such as Boost Asio (an asynchronous framework which could potentially become difficult to maintain in the future) or *C++17 futures* which greatly limits backwards compatibility. This would unfortunately restrict the support to the very latest compilers which are simply not present on older platforms (or potentially early stage research compilers in the future). The WebSocket protocol was relatively complex to develop and consisted of a mostly standard HTTP server, where clients could connect and request an upgrade of protocol. This would involve a handshake and the transferring of protocol data. After this, packets needed to be sent in such a way that it was suitable for browsers to understand. A general overview can be seen in **Figure 3.46**.

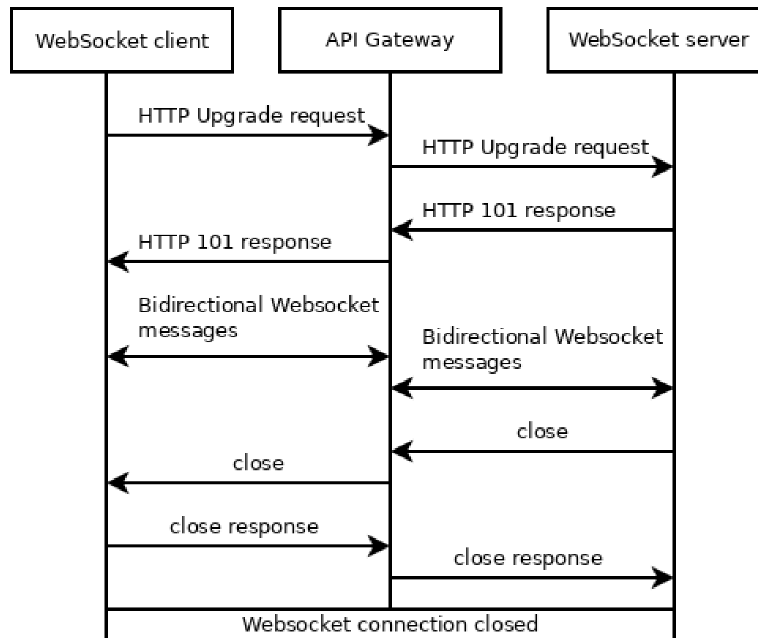

Figure 3.46: *An overview of the WebSocket protocol implemented as part of Hydra. This specifically allows for the communication with modern web browsers (Oracle Corporation, 2018 (accessed March 3, 2019)).*

This task was mostly technical in nature but provided a very useful experience and insight into just how much work was involved to add an additional method of communication from Hydra. What was most interesting to note was that so far, it has been the most complex in terms of protocol. The implementation using raw sockets and the RS-232 serial were fairly straightforward in comparison. Overall, however, in terms of the actual code required for the implementation of a new protocol, it has shown that the design of Hydra is very scalable. Possibly, in future, with the addition of LibreSSL or OpenSSL on supported platforms, encryption can be added as a further exercise. This could be useful in case unencrypted WebSocket communication ever becomes deprecated (Berners-Lee, 2015 (accessed April 3, 2019)).

The decision to use WebSockets was not due to the current ubiquitous nature of web browsers but in fact originated from its relative complexity. It was noted that if Hydra could be successful, even when built upon a fairly restrictive platform such as HTML5, it would demonstrate that so long as a subset of a platform is available, at

the very least, a proof of concept could be developed. Due to the volatile nature of web browsers, it is fairly evident that they are are not going to be a long term solution to digital preservation, however they demonstrate a good example of appropriate content viewers of this current era. Had Hydra been developed in the late 90's, a client developed in Microsoft Visual Basic 6 would have been equally appropriate as choice of technology for the simple viewing client.

WebSocket itself is a socket stream connection, in that it requires a connection to be established (unlike the connectionless UDP), however each message sent is preceded by additional information. A typical WebSocket packet has a header that contains the data as shown in **Figure 3.47**.

Bit	+0..7	+8..15	+16..23	+24..31
0	FIN Op	Mask Length	Extended length (0-8 bytes) ...	
32	...			
64	...		Masking key (0-4 bytes) ...	
96	...		Payload ...	
...	...			

Figure 3.47: *A diagram showing the low-level layout of a WebSocket frame as specified by RFC 6455 (RFC6455, 2011).*

Each time a message is received, this additional data needed to be processed to that the packet could be handled correctly. Some of the important flags and fields are detailed below:

- **FIN** - Instructs whether the packet is complete or is to be followed up by a continuation packet.
- **Opcode** - Notates the type of packet such as a message or a ping.
- **Length** - Holds the length of the following message. If larger than the allocated section (value greater than 126) then its size grows into the Extended length sections.

- **Masking key** - The payload is encoded with this key to ensure that unique data is sent (thus avoiding issues with caching proxies).

It was discovered early on during development that although time consuming to research and implement, WebSocket did not add much complexity above the work required to build up and process the packet headers. In particular there was a maximum packet size, however this was extremely large and caused no issue with the traffic produced by Hydra, even when uncompressed. The main requirement was that a robust TCP implementation was available. This is demonstrated in **Figure 3.48**. Of particular note, each path of the call graph was terminated by a call to a standard TCP function; showing the strong relationship between the two technologies.

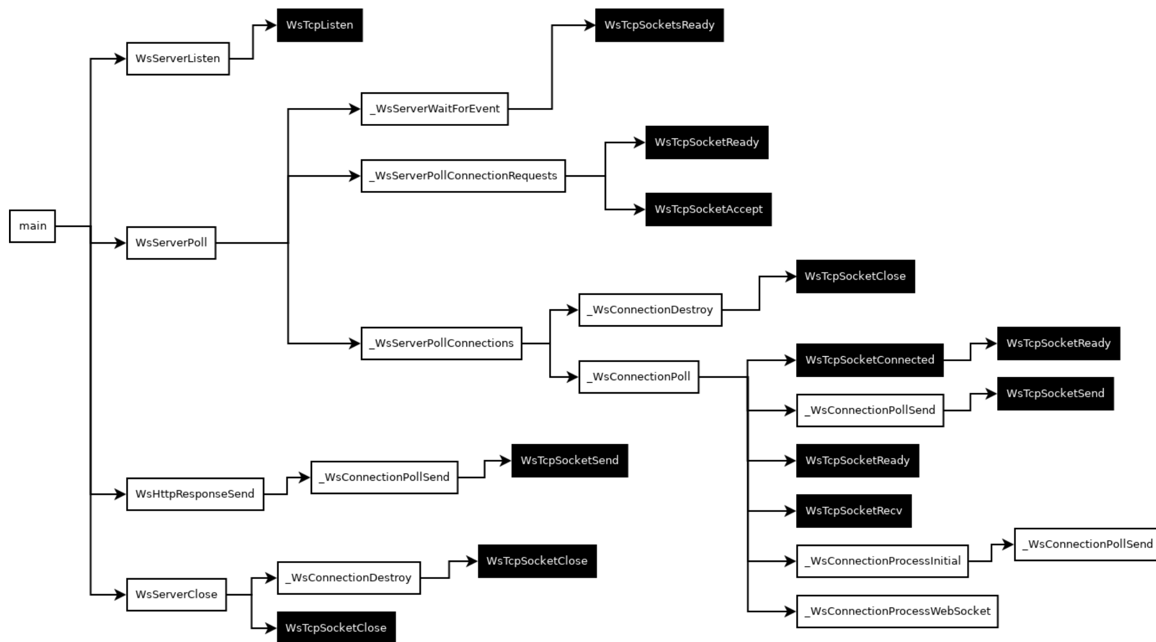

Figure 3.48: *Flow diagram of the Hydra WebSocket component **libws**. All WebSocket paths ultimately end in raw TCP socket communication.*

As discussed in 3.2.8, the web socket library was developed in an object oriented manner. The class diagram shown in **Figure 3.49** demonstrates an attempt to map a number of concepts together including:

- **ANSI C -** a language with no built in OOP features.

- **libstent -** a complex library which extends the core C language.
- **WebSockets -** an implementation of RFC 6455.

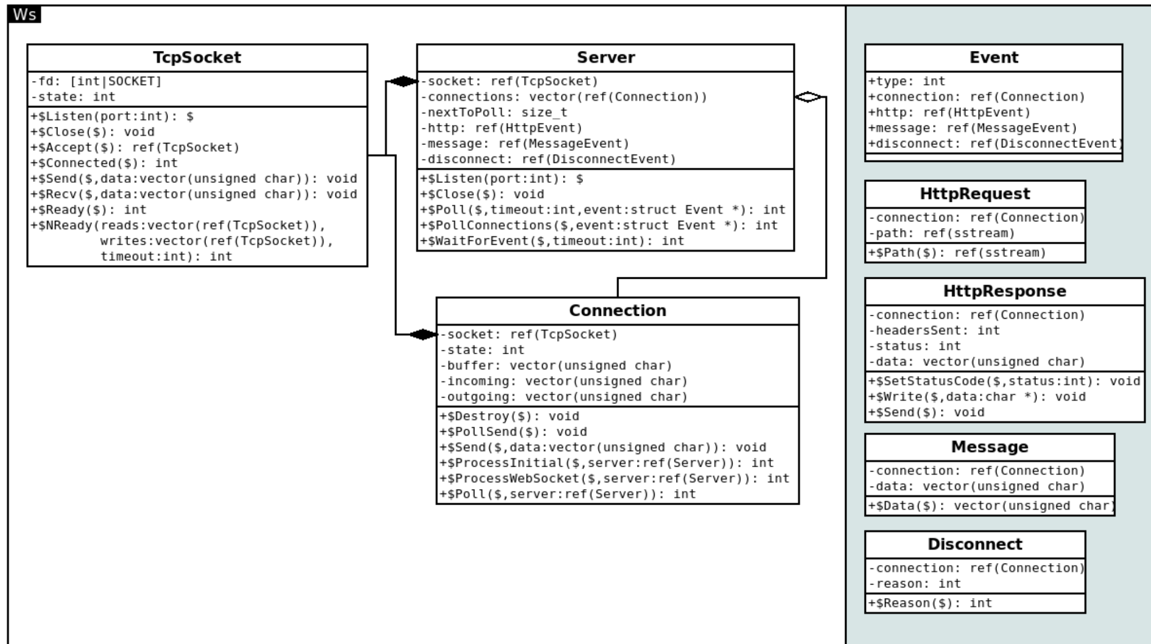

Figure 3.49: *Class diagram showing the design of libws. It has a number of subtle differences compared to that implemented using a conventional OOP language.*

Whilst libws is only a very small part of Hydra, this diagram provides a useful example of some of the considerations and changes needed in order to be represented in a class diagram intended for an object-oriented language. Of particular note, the **$** characters represent a shorthand for the full function name, as well as providing the context structure pointer. This was required to avoid a lot of repetitive information being included which would reduce the clarity of the diagram. Another area of note is that objects could contain either public or private variables; not a mixture of both. Again, this is less of a limitation than one might first assume and one that the C++ PIMPL idiom also dictates. Finally the **ref()** attributes simply refer to a pointer that is managed for safety. Even though this library was designed to be reusable for a number of projects the managed pointers are used almost exclusively because

libstent can be disabled in projects that do not need to utilise the extra safety related machinery.

3.3 Summary

This section has covered many details relating to the design and implementation of Hydra. It has looked into the benefits of reducing dependencies and how that interacts with future portability. Likewise the separation of these dependencies or the underlying platform from the core application logic has also been explored, including it's role in increasing flexibility when it comes to maintaining software. The chosen programming language itself is important, not so much due to the language itself but due to the binding with the underlying platform which can itself begin to pose a problem when it comes to portability and maintenance. Java in particular was used for many examples but the same can be true of many programming technologies. A number of platforms have been identified that could pose a technical challenge to port to. These are all examples of where an approach utilising Hydra could be beneficial. A more technical discussion of the underlying Hydra API was also provided, showing how the API cloning technique can provide a good benefit when integrating the system with existing software. Likewise even new greenfield projects can benefit from developer familiarity with the existing OpenGL API. Some additional work on leveraging this familiarity was also discussed as part of overcoming issues at a protocol level. For example the handling of deprecated OpenGL functionality and limitations such as non power-of-two sized textures.

The underlying approach to synchronisation of client state was described, including how it maps well to the retained mode approach utilised in OpenGL. The additional requirement of multi-user state management was not necessary for digital preservation with Hydra, however it opens up a number of interesting oportunities for multi-user applications. For this reason support was provided and the state duplication system

was discussed.

An important section covering a unique approach to memory management (and a range of other errors) was covered via the use of the Stent framework. This was not only useful in order to overcome a number of technical challenges when dealing with the lower-level C language but also provided a number of improvements in terms of future maintenance of the existing software written with Hydra. With this in place, it is envisioned that simulation software written directly using C is more viable with this in place. In many ways this viability was exercised with the implementation of the underlying WebSockets layer, used to connect a client web browser to the program utilising Hydra. This WebSocket implementation was also discussed.

Chapter 4

An Alternative Approach to Multi-user Architecture

4.1 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 (Laurens et al., 2007). Issues regarding real-time performance of user interactions and graphics rendering remain challenging, even with today's state of the art software technology (Wu et al., 2014; Karachristos et al., 2008). 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 (Carnall, 2016; Glazer and Madhav, 2015) and Unity (Stagner, 2013) are employing the centralised client/server architecture (Färber, 2004). 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 centralised design enforces the game developers to rely on infrastructures provided by game engine manufacturers, which can prevent software preservation and re-usability, an important topic that has been overlooked until now (Matthews et al., 2010).

The rapid development and evolution of computer architecture often fails to provide the infrastructure required in order to ensure that older software can continue to run on recent platforms. The reasons for this were investigated and discussed in Chapter 2.

One of the outputs of this thesis is to introduce a novel distributed architecture for multiplayer games; Hydra, which is an evolving attempt at addressing the aforementioned challenging issues. In addition to this, Hydra is also aimed at improving the lifespan of software. In particular, through Hydra, 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 Hydra, older 3D software can be guaranteed to run because of their use.

Hydra 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 (Pellegrino and Dovrolis, 2003; Wang et al., 2009).

The contributions provided by Hydra 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 some of the existing solutions to the synchronisation of multiplayer games are identified. Then, some of the complexities involved in client side synchronisation will be analysed, 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. The design of Hydra and how it relates to a multiplayer deployment will be discussed in Section 4.4. Performance evaluation is presented in Section 5. Finally, some future developments for the extended use of Hydra in both research and development projects will be covered.

4.2 Related Work in Client Synchronisation

Existing online multiplayer games utilise 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. (2002) 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 and Levine (2001) 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 Synchronisation protocol which relaxes the requirements of Lockstep by decentralising 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 appears to be required.

Research work has been previously undertaken in the similar area of multiplayer synchronisation but with a very different approach to what was proposed in an earlier paper (Pedersen et al., 2013). In order to create a protocol which reduces cheating, the proposed idea was to use 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. The implementation (Distributed DeepThought) 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 in a number of areas including bandwidth consumption and ease of implementation. As part of a prototype, the synchronisation system was integrated with two existing games developed for LEGO from Amuzo (an independent games development studio). These were LEGO Ninjago and LEGO Hero Factory: Brain Attack The fact that the system could be retro-fitted and integrated with the existing software, as opposed to the software being re-engineered from scratch or by requiring a large refactor, helped to demonstrate that this approach was very easy to maintain and extend. Later, two separate software titles were developed using the same technology. These were Sumo Penguin and a BT Sports game prediction service.

However, a number of complexities with the protocol were discovered. These are described in Section 4.3, so the 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 the need to look into new ways to reduce the need to synchronise the state entirely and move towards streaming technologies, such as the one used in the final design of Hydra.

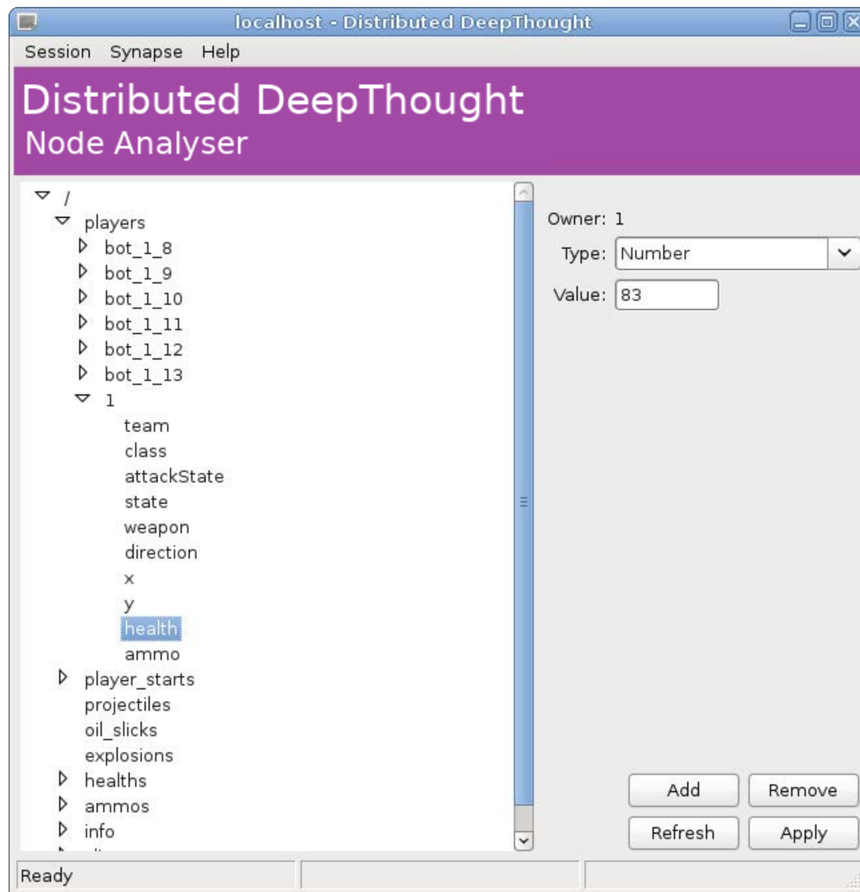

Figure 4.1: A small internal tool which allowed for the debugging of the Distributed DeepThought node based hierarchy. This tool was invaluable in simulating and testing any potential damage that a malicious user could make.

4.3 Complexities Involved in Client Synchronisation

Developing a multi user application is a more complicated and expensive process than single user software (James and Gillam, 1999). 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.

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 needs 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 an additional requirement is subsequently added and 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 serialise 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 4.4.4 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 (Sanglard, 2012). However, the idea is that with a technology such as Hydra, all of these steps needed to synchronise client states can be avoided.

4.4 Inner Workings of Hydra

Hydra implements a client/server architecture where rather than having the running 3D program calling the OpenGL API to communicate with the GPU to rasterise 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 4.2. From a technical viewpoint this architecture has the benefit that complexity can be encapsulated. For example, results from memory checking tools such as Valgrind (Valgrind Memory Debugger, 2017) can often be affected from details of the lower level layers of an operating system. With Hydra, 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.

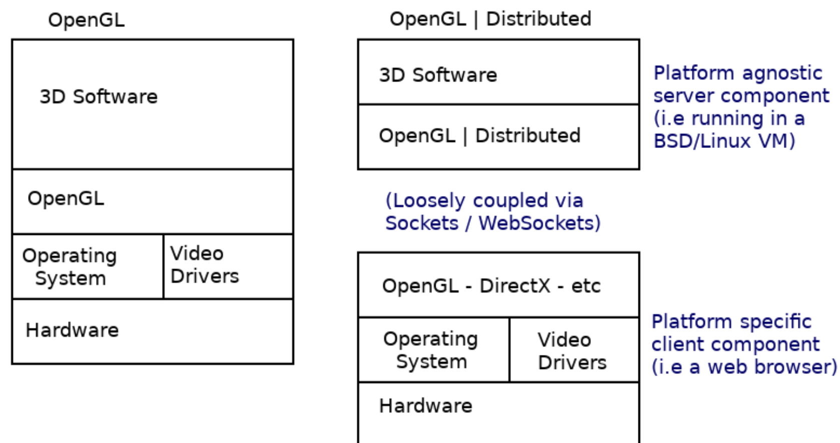

Figure 4.2: Diagram describing the layers that OpenGL is built upon compared to Hydra. Notice that Hydra has additional layers of abstraction.

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 Hydra's use as a multi-user solution.

4.4.1 Protocol Overview

The Hydra 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 4.3.

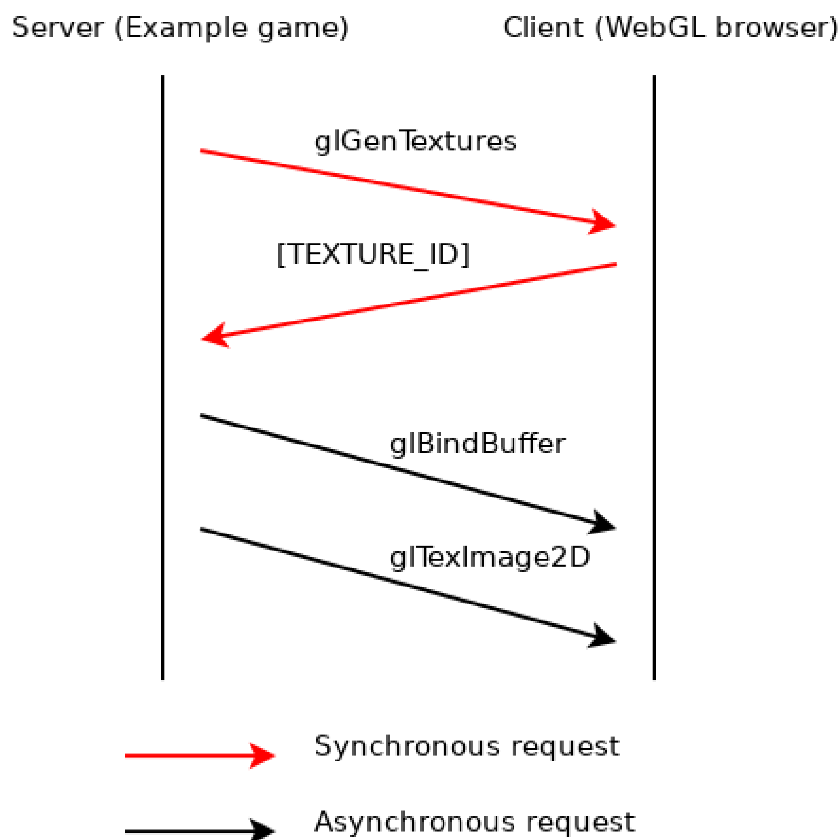

Figure 4.3: Diagram demonstrating a typical yet simplified communication between the client and server components of Hydra in order to upload a texture.

Due to the fact that Hydra 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 4.3, which demonstrates synchronous requests, utilises the Hydra 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 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 Hydra 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.

Unlike when used for digital preservation purposes; there are a number of additional requirements that need to be resolved within Hydra. One important example is the increased importance of not blocking communication between clients 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 utilised. 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 optimisation techniques.

4.4.2 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 Hydra 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 thesis), 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 behaviour was demonstrated in a simple multiplayer football game created as a simple prototype (**Figure 4.4**) 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 Hydra, 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 Hydra, this complexity could be avoided.

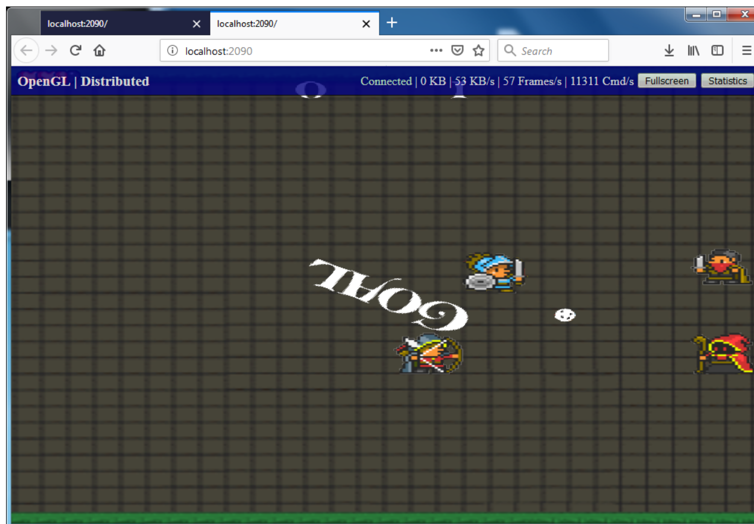

Figure 4.4: *A multiplayer (across a network) football game where the players are characters from traditional fantasy role playing games*

4.4.3 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 Hydra, 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 Hydra in that whilst the update function is called just once per frame in Hydra, 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 4.5**, where a player selection dialog is shown to a newly connected client without obstructing the view of existing players.

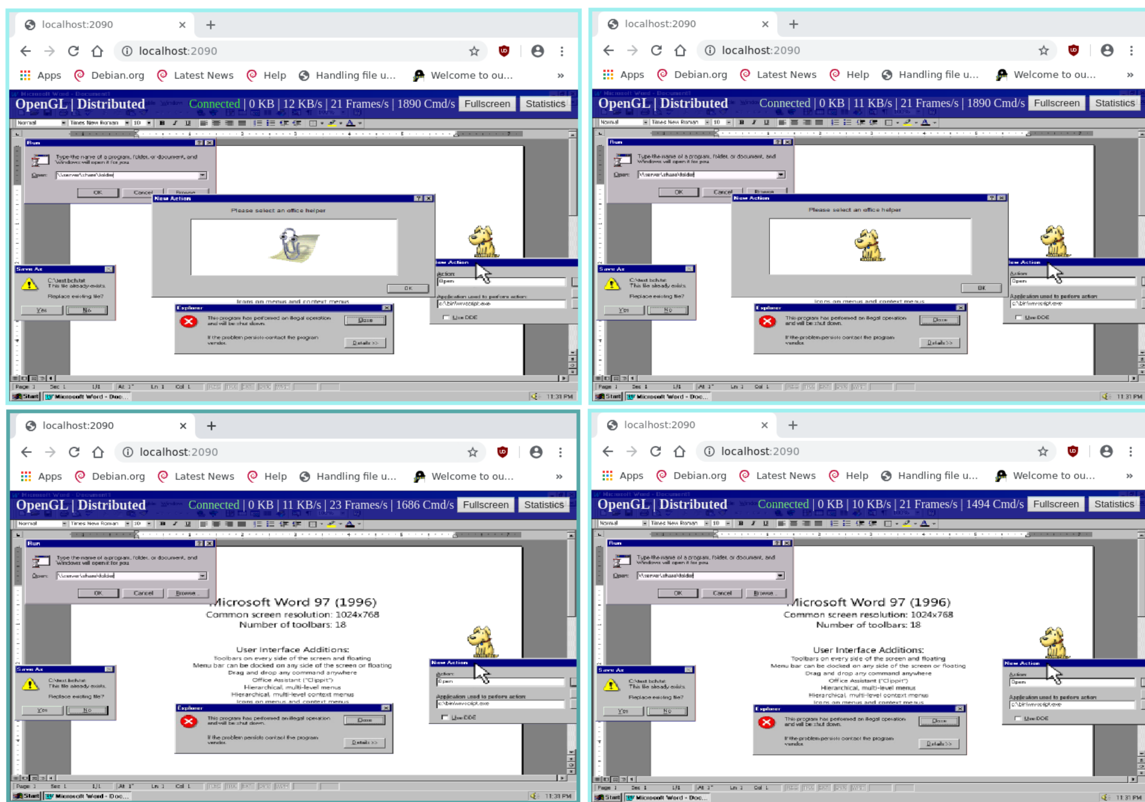

Figure 4.5: *Screenshot showing four wave clients connected. They all look similar because they all share the same drawing commands apart from a few subtle differences. For example, the player select menus are unique.*

4.4.4 Cheat Prevention

One of the more interesting features of using Hydra as a solution for multi-user applications and games is that cheating can be eliminated. The clients themselves are akin to dumb terminals (Bulterman and Van Liere, 1991) 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.

Chapter 5

Evaluating the use of Hydra in Multi-user Software

This chapter aims to evaluate the effectiveness of Hydra as a medium to transmit graphics across a network as an alternative to the popular streaming technology VNC (Virtual Network Computing). It aims to investigate and compare both the performance and the bandwidth usage when dealing with streaming a 3D scene.

This chapter also compares the bandwidth requirements of Hydra with QuakeWorld to ascertain its feasibility as a technology for multiplayer games.

This chapter then discusses any potential limiting factors and optimisations that could help improve its usage for these kinds of high performance streaming uses.

5.1 Streaming Comparison Against VNC

Compared to existing solutions involving manually syncing the client state (Kaukoranta et al., 2002; Smed et al., 2002), there is virtually no network overhead when using Hydra 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 rasterised images, Hydra has the potential to be a much faster solution because it uses an intelligent protocol which sends the commands that 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 5.1. 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 rasterised pixels is sent regardless. The bandwidth requirements when using Hydra 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 optimisation techniques used to reduce the number of draw calls.

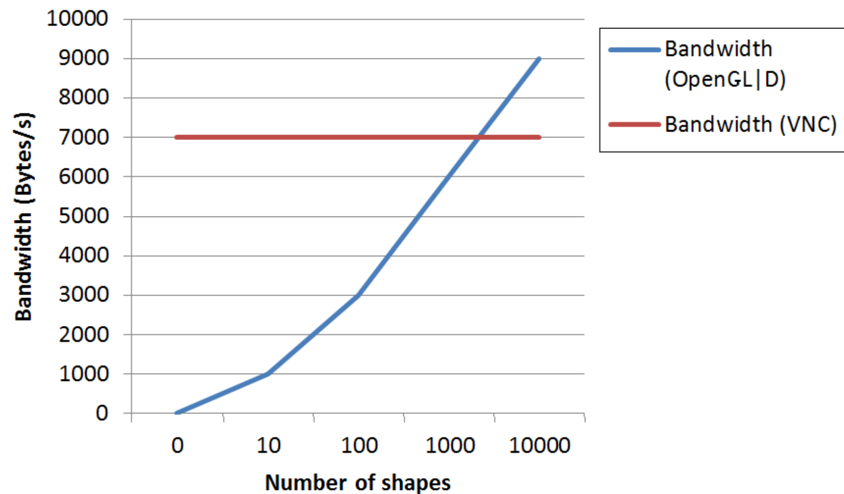

Figure 5.1: Graph comparing the bandwidth requirements between Hydra and VNC with a varying number of objects in the scene.

In general, network synchronisation via Hydra 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 rasterised streaming solutions at higher resolutions. Hydra does not need to send through each pixel to the client, the clients do the actual rasterisation, therefore, there are no additional costs to bandwidth using Hydra at higher resolutions. This is demonstrated in Figure 5.2.

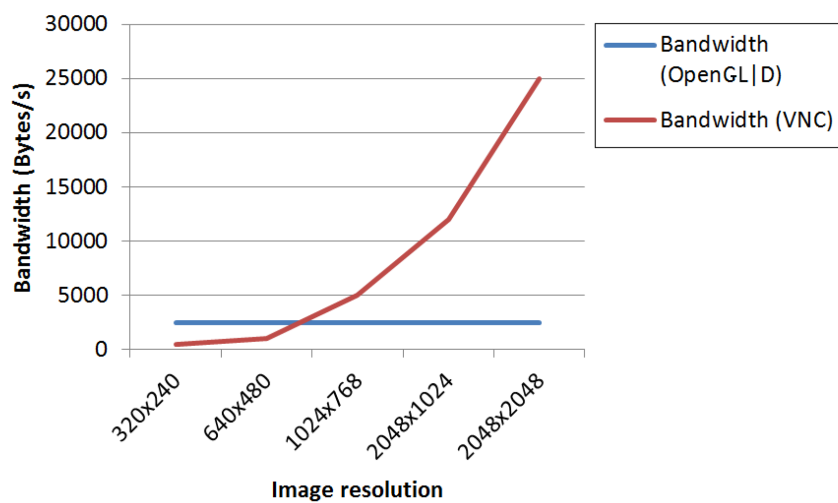

Figure 5.2: Graph comparing the bandwidth requirements between Hydra and VNC with an increasing image resolution.

Network synchronisation via Hydra 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 (id Software, 2017). The QuakeWorld client in particular still, provides an adequate test bed for comparison with Hydra. What makes the QuakeWorld client very convenient to test against is that it employs a constrained subset of OpenGL called miniGL (Hilgart, 2006) and so is fairly straightforward to port to using Hydra. Whilst Hydra does not yet provide full coverage of the most recent OpenGL API, the majority of functionality required for miniGL 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 Hydra works, this was deemed satisfactory and would not alter results in any way. Further work is certainly planned in this area.

The initial tests agree with the work carried out by Cordeiro et al. (2007), Abdelkhalek et al. (2003); Abdelkhalek and Bilas (2004) 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 Hydra 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 Hydra'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 Hydra client, whereas with the same trade mechanism, the Hydra 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 Hydra can potentially scale in a more favourable way compared to traditional approaches.

5.3 Network Protocol Optimisation 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 comparable to the manual synchronisation system found in existing games (yet retaining all the benefits of having a single program state).

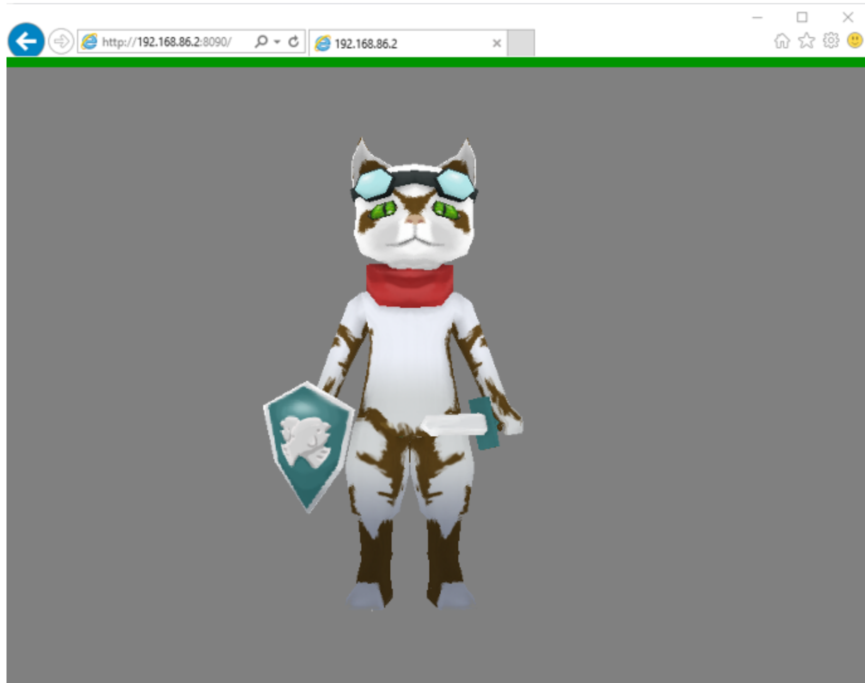

Figure 5.3: Screenshot of an example Hydra output. An application with this rotating 3D model takes less than 10 bytes each frame. Even with maximum compression, VNC takes over 20 times that in bandwidth for a similar (but lower quality, lossy compressed) image.

5.4 Planned Optimisations

There are a number of optimisation techniques we plan to introduce to Hydra 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 sub optimal.

Finally, based on the experience that UDP is likely to be a more optimal solution than the existing TCP based system (Xylomenos and Polyzos, 1999) and the realisation that in the main draw routines, unreliable packet transmission can be handled effectively, this faster but less reliable protocol is a feasible optimisation. Only in transferring permanent state changes or uploading data objects is reliable transmission (either via TCP or a reliable UDP scheme) desired. However, the 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 Hydra viewer is planned, where the use of UDP can be explored in a more thorough fashion. This is particularly important because digital preservation is the highest priority for Hydra. Improvements to performance can only be made if they do not conflict with the overall portability of the solution.

5.5 Conclusion

The results from this chapter suggest that Hydra provides a strong solution to a number of use-cases. In particular at higher resolutions, the sheer volume of data that would require to be transferred each frame becomes difficult to process if using a traditional rasterisation technique provided by VNC. The intelligent protocol provided by Hydra alleviates this by simply sending the instructions which are typically smaller packets. Then the client can build up the much larger and detailed image dynamically on their end. The bandwidth comparison between VNC and Hydra

has demonstrated this to be possible.

Where Hydra is possibly less suited to streaming than rasterisation approaches like VNC is where there are a lot of objects without any potential batching of instructions. At low resolutions, the overhead of the individual instructions being sent end up outweighing the bandwidth required for the VNC framebuffer. Unfortunately at very high resolutions and with many objects, neither solution provides an ideal result. However, with the ability for Hydra to batch data on the client, directly mirroring the OpenGL API and the retaining of data on the GPU, then this bandwidth requirement is greatly reduced and Hydra becomes a very feasible solution when VNC may not be appropriate.

In the comparison with QuakeWorld, it can be seen that in general Hydra is more expensive in terms of bandwidth. However in more complex situations involving the synchronisation of a lot of state, there becomes a point where Hydra provides a better result. This suggests that whilst Hydra might not be an ideal solution for an underlying transport mechanism for a fast paced action game, for simulations or those with complex multi-user requirements, it could be very suitable.

5.6 Future Work in Multi-user Synchronisation

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 analytical 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.

A future multi-user project involving Hydra is to expand upon ZMG, the game to help

learn and practice maths in such a way that multiple players can join along with a class instructor. This will allow for a greater interaction between the students and the instructor. It also enables the ability for students to compete and for the instructor to dynamically alter the gameplay as they deem necessary. 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 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, Hydra is very likely to simplify this process by virtue of each client implicitly sharing the same game state.

5.7 Summary

The process of developing a multi-user project can be greatly simplified by using Hydra. 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 serialised 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 Hydra against VNC has been valuable. This is because in terms of portability, other solutions such as NoMachine's NX server (NoMachine, 2017) or

GameStream (NVIDIA, 2017), NVIDIA's commercial streaming technology, are not available on all but the most common platforms. This would greatly limit their ability to be used for the facilitation of 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 Hydra and VNC however. Yet as the results suggest, there are a number of performance concerns with VNC, especially at higher resolutions which many users now expect. Hydra provides a potentially strong alternative to VNC for a number of use-cases and the results in this chapter support this.

Chapter 6

Evaluating the Effectiveness of Hydra

The main focus of Hydra is to facilitate digital preservation and previously in the thesis, only the network performance has been discussed and compared to the existing VNC technology (Chapter 5). This chapter will analyse the effectiveness of Hydra in areas directly relating to digital preservation.

With a large proportion of Hydra implemented to support the OpenGL 2.1 specification, and thus a number of different software packages, it was deemed ready for experimentation. Experiments were broken down into two parts, each testing a different aspect of Hydra. This was done due to the recognition that the preservation of software is more complex than simply getting it to run on current hardware found today but, instead, by measuring the ability for a legacy developer to maintain the software on today's platform and seeing how much of their workflow has needed to change.

6.1 Performance Against Existing Techniques

The first of a series of experiments is focused on primarily performance concerns. Specifically for games, performance is critical for the success of Hydra as a solution for digital preservation because without a playable frame rate, there is not much in

the sense of a playable game. The goal of this experiment is to find out if software rendering is a solution for gaming titles in different types of operating environments and also to ascertain whether the passthrough technique employed by Hydra is comparable to the current industry standard solution of virtualised GPU passthrough. As a note, Hydra also aims to support non-game related software for which the performance is less critical because real-time frame-rates are not strictly necessary in modelling tools (to name but one example).

6.1.1 Experiment Design

A number of different types of software were tested in specific configurations in order to test the limits of the run-time environment. These limits are required to evaluate to what extent the current state of digital preservation of the software is potentially lacking and to measure in what ways Hydra has improved upon them or has encountered issues. The limits pertaining to the successful execution of the software in this case consist of the following:

- **Frame-rate** - An important factor, especially in games because the software doesn't just need to work but it needs to provide a suitable experience for the user. If the frame-rate is too low, stuttering behaviour can be observed which breaks emersion. This specific data will be collected using the inbuilt debugging tools from the Half-Life engine and likewise from the custom software renderer and Hydra which both support reporting the framerate as part of their development requirements to help facilitate debugging performance.
- **CPU usage** - Whilst strongly linked to the frame-rate, the CPU usage is an important factor to record separately. This is because if the rendering system retains a high framerate but requires the majority of the CPU processing time, this will likely detract from other game or simulation features such as collision which will also restrict the suitibility for use. This data was recorded using

the Sysinternals Proces Monitor tool on the Windows 2012 platform and on Windows NT 4.0 using the older Resource Kits for Windows NT. On Linux, this was recorded via a script piping output from *uptime* and *top* to a file during execution.

frame rate. Along with the frame rate, the CPU utilisation will also be recorded to help find the stress on the hardware, even if the desired frame rate or limiter has been reached.

The primary configuration is the operating environment; a number of different operating systems are required to test the availability of the software including those from the past (when these operating systems were common). While some of these operating systems are still in use today, some are regarded as quite exotic or have some idiosyncrasies leading to difficulties in porting to them. This diversity of these environments was necessary to ensure that the complexities and edge cases of running old software in emulated or virtualised environments were correctly encountered. The test platforms and justifications are as follows:

- **Windows NT 4.0 [native | virtualised | emulated]** - This platform was chosen because it has a large catalogue of software available for it, allowing for testing against early commercial titles. Unlike MS-DOS, many of the software titles were starting to make use of a GPU so not only offered a software renderer fallback but also a GPU accelerated configuration. It can be run natively on old hardware (IBM ThinkPad T23), virtualised in a limited selection of hypervisors and emulated by QEMU.

- **Raspbian (Linux) [native | emulated]** - This platform was chosen because it offers an alternative CPU architecture to the other tests (ARM aarch64) and provides a good catalogue of open-source software to test on it. It can be run natively on current hardware (Raspberry Pi 3), can be fully emulated by QEMU (qemu-system-aarch64) but as of yet there are no virtualisers capable

of supporting this platform due to their commercial focus on primarily current x86_64 architectures.

- **Windows Server 2012 R2 [native | virtualised]** - This platform was chosen because it has a good backwards compatibility strategy to run older titles. It can be run natively on current hardware (HP Z420 Workstation), virtualised in a large selection of hypervisors but is simply too heavy to be fully emulated by QEMU without any form of virtualisation. This is seen as the pilot environment because this version of the Windows kernel is potentially the most common platform in use today for entertainment and media (Valve Corporation, 2018 (accessed April 3, 2018)).

Note that all platforms include native execution, virtualised execution and fully emulated execution where possible. This is to help ensure that any performance issues can be attributed to the display method, rather than other activities such as physics. The Raspbian platform was unable to be virtualised and the Windows Server 2012 R2 platform was unable to be emulated in this experiment for the reasons stated.

The second configuration related to how the graphics were rendered. This allowed for a number of different ways to ascertain exactly how the issues running the software had manifested themselves. By using any solution available to us, and by getting the software in the very least to run, any potential compromises can be identified. These configuration variables include:

- **Accelerated 3D Rendering [320x240 | 800x600 | 1024x768]** - Where possible, the use of the GPU will be utilised so that maximum performance can be achieved. This is so that it can be compared with the other forms of rendering and their relative performance evaluated. Many of the chosen operating environments will not support this feature however, so, this configuration will

also act as a good evidence of how 3D hardware is not accessible for many preserved entertainment titles.

- **Hydra Passthrough Rendering [320x240 | 800x600 | 1024x768]** - Rendering will be performed on the host machine outside any virtual or emulated environment and the performance data will be measured against the other forms of rendering. This data should provide the results needed to measure the success of Hydra.

- **3D Software Rendering [320x240 | 800x600 | 1024x768]** - For the software that supports this ability; by measuring the performance of rendering without any accelerated hardware, this can act as a control to find out the specific results of a worst case scenario.

Note that the different types of rendering are performed at three resolutions, 320x240, 800x600 and 1024x768. This is so that the increased cost for the resolution increase can be measured and the scalability of the different rendering techniques can be evaluated.

In order to test the different rendering techniques, a number of different software packages are used. These packages consist of an older commercial gaming title, and two simpler 3D games developed as part of this research to utilise various technologies. These software packages and their justifications are described below.

- **Half-Life - Valve Software [Software | Accelerated]** - In order to help produce robust results Half-Life was chosen due to the fact that it has both a software renderer and an OpenGL hardware accelerated renderer. This allows for the normalisation of results spent on unrelated engine activities such as collision or even audio. Half-Life was released in the time frame where accelerated 3D graphics cards were not necessarily available to all consumers and as such the software renderer fallback was required. Another reason as to

why Half-Life has this functionality is because it was based on id Software's Quake 1 (id Tech 2) (Gamespot, 2011 (accessed April 3, 2014)) (not the Quake III Arena engine (id Tech 3) being released at around the same time or even Quake II (a more recent version of id Tech 2) released earlier). This software renderer also has intense optimisation due to the lower spec machines it needed to be able to run on effectively. Much of it was even written in low-level x86 assembly (Sanglard, 2017) for additional performance, which will provide very interesting results. The software can be seen running in **Figure 6.1**.

- **3D Software Renderer - Internal Prototype [Software]** - As part of this experiment a very simple 3D software renderer was developed to help compare and verify findings from Half-Life. It was written entirely using a modern C compiler and using parallelisation (-fopenmp) and full optimisation flags (-O3). However it did not utilise the lower level x86 assembly employed by the Quake 1 based renderer and as such it should provide an interesting comparison. Unlike the closed source nature of Half-Life, this software can be modified to test various aspects of the rendering process such as enabling back face culling or rendering a single triangle which, again, can help to ascertain the most expensive parts of the rendering. This software is also interesting in that regardless of resolution, the fragments rendered will only be in a 320x240 environment. Not only does this help maintain the performance but it also allows for testing that the majority of processing power is spent on the rendering of 3D polygons rather than simply drawing the fragments to the screen. **Figure 6.1** demonstrates the output quality from this software renderer.

- **Zombie Maths Game - Internal Prototype [Passthrough | Accelerated]** - Another internal prototype but of a much larger scale than the previous one and written in C++. This prototype was originally written to help facilitate the learning of maths in younger children and originally written utilising OpenGL.

This was an ideal platform to test the capabilities of Hydra as an OpenGL drop in replacement. This process was already discussed in **Section 3.2.1**. This application will be tested to provide results for both accelerated rendering and the Hydra passthrough rendering. As with the other prototype, certain aspects of it can be disabled, such as the physics and certain graphical effects.

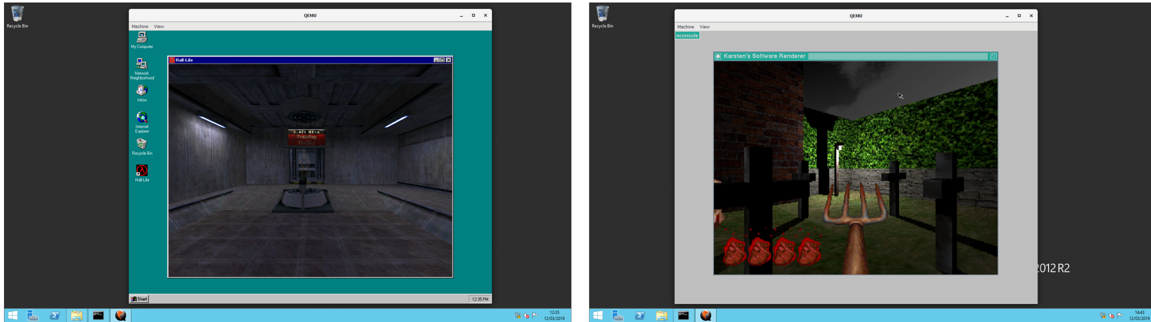

Figure 6.1: *Screenshot showing a Windows NT 4.0 VM running Half-Life and a Linux VM running the software renderer*

In order to ensure a fair test, the same map was used for each renderer. Based on a number of smaller pilot studies, it was decided best to use a relatively low number of triangles (262) due to the computationally expensive nature of running 3D renderers within an emulated environment. The map consists of a low number of simple 3D shapes, positioned in a way that all of them could be in view. The final map used can be seen in **Figure 6.2**.

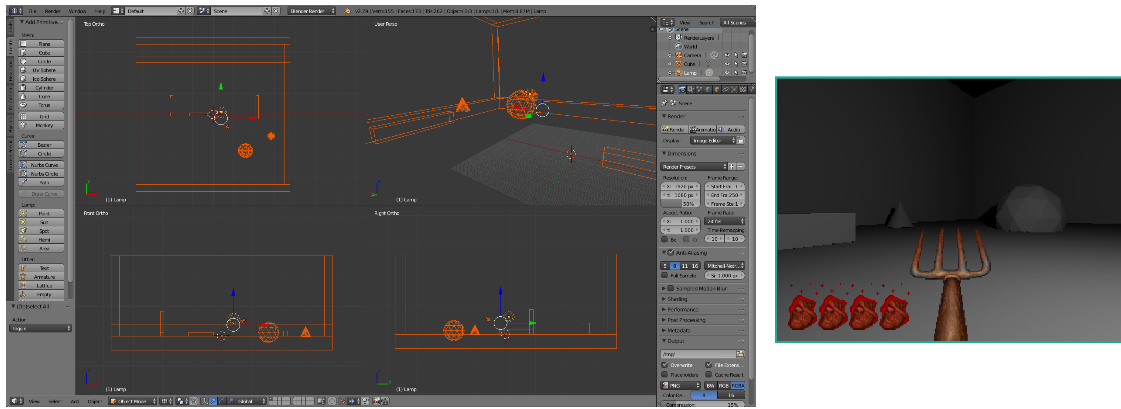

Figure 6.2: *Screenshot showing the sample map within Blender and being rendered by the software renderer*

6.1.2 Analysis of Results

After the experimentation on the configurations above were carried out, the results were gathered and analysed. The first is a coverage on the types of performance which can be expected with software rendering versus acceleration. **Figure 6.3** shows the performance data for rendering on native hardware.

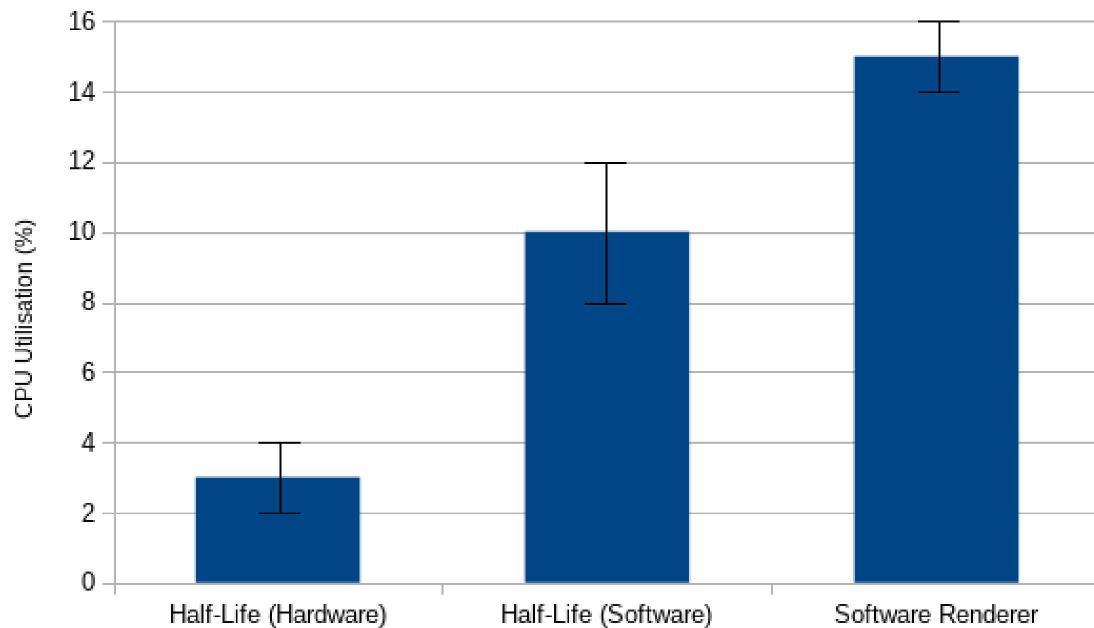

Figure 6.3: *Graph showing the CPU utilisation of Half-Life using the software renderer and OpenGL accelerated render compared to the internally implemented software renderer. All instances are running at 800x600 resolution on a native machine running Windows Server 2012 R2.*

The data suggests that whilst the frame rate of 60 frames per second has been reached on all configurations (and thus appropriate for games) the execution cost in terms of CPU utilisation of both software renderers is much higher than the GPU acceleration configuration. This is expected because much of the workload on the CPU has simply been passed off to the GPU. The GPU load is largely irrelevant although the GL performance analyser reports it is low due to the fact that the hardware is specialised for this type of calculation. Half-Life is also a much older title so does not pose much of a challenge for modern GPU hardware.

What is more interesting is the performance comparison between the two software renderers. Half-Life reports a much lower CPU requirement even though it is not only using an older compiler (Valve Corporation, 2011 (accessed April 3, 2019)) with less optimisation work but it is also processing many more fragments than the internal software renderer (which as mentioned previously only really addresses

320x240 fragments). This is likely due to the low level x86 assembly used and the many years of commercial optimisation which our internal software renderer cannot compete with. These optimisations do not necessarily translate well to emulated environments where less assumptions can be made.

One area of note is that the CPU utilisation of both Half-Life (Hardware) and the custom software renderer was fairly consistent, occasionally fluctuating with a difference of 2%. However for the Half-Life software renderer, these fluctuations were often greater with a 4% difference. Whilst this difference is still fairly subtle, it could be a symptom of a number of optimisation strategies in place such as the binary space partitioning going between boundaries. Whereas the custom software renderer is fairly basic in comparison and does not take advantage of this.

Figure 6.4 shows the performance data for rendering on emulated hardware. Note that this comparison does not include the accelerated Half-Life renderer because it is simply not available in the environment.

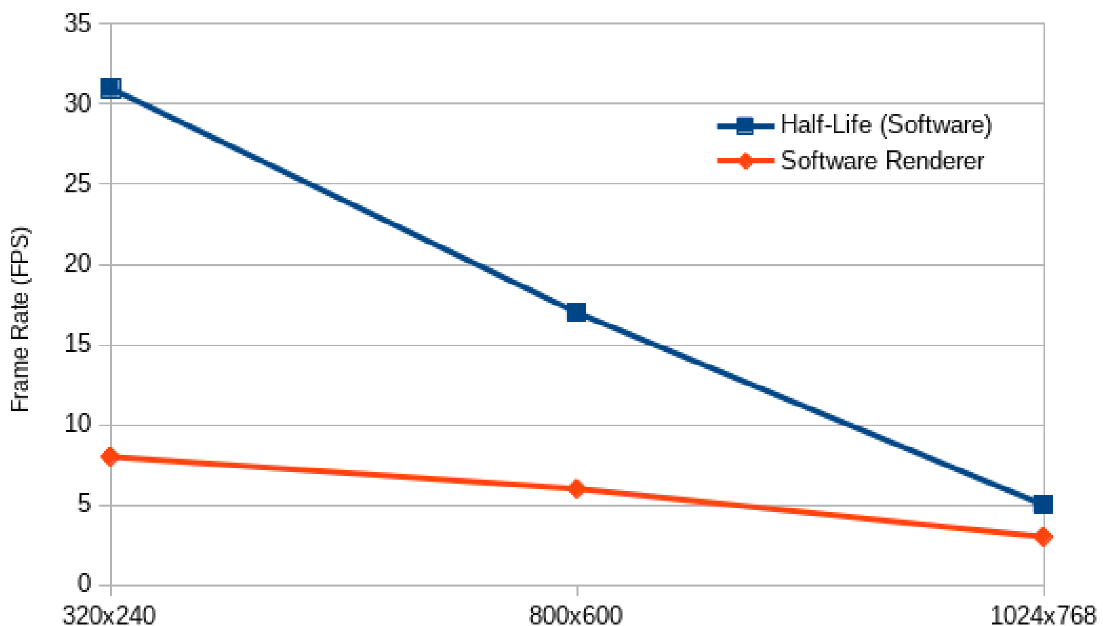

Figure 6.4: *Graph showing Half-Life software renderer against internal software renderer at different screen resolutions running in a fully emulated Windows NT 4.0 environment.*

The performance data for software renderers in an emulated environment is quite disappointing. Neither programs run at a playable speed from the offset whilst both are using the maximum CPU. What is interesting however is that both seem to be rendering at a similar frame rate at the higher resolution, showing that many of the optimisation techniques on the commercial software renderer are less effective. It should be noted that the compiler optimisations and parallelisation for the internal software renderer were also not available on the compilers available to Windows NT 4.0 and, as such, many of these optimisations were also not effective. In particular, the emulated environment only provides one emulated processor so much of the parallelisation code would be ineffective anyway. The results also show that as the resolution increases, the frame rate drops rapidly demonstrating that software rendering is not scalable to the types of screen resolutions that are expected today (Valve Corporation, 2018 (accessed April 3, 2018)). Again, the internal software renderer does not change too much with resolution because the fragments are locked to 320x240. This also demonstrates that the cost is in generating the fragments rather than drawing the pixels on screen. This data shows that for pure emulation, software 3D rendering is not a viable solution.

These results help us to effectively discard the idea of using software renderers within emulated environments. The speeds are simply not feasible. Moving away from software rendering as a solution, the next series of results show the performance data of hardware virtualised acceleration and the Hydra renderer compared to the software renderers on a native platform. This can be seen in **Figure 6.5**.

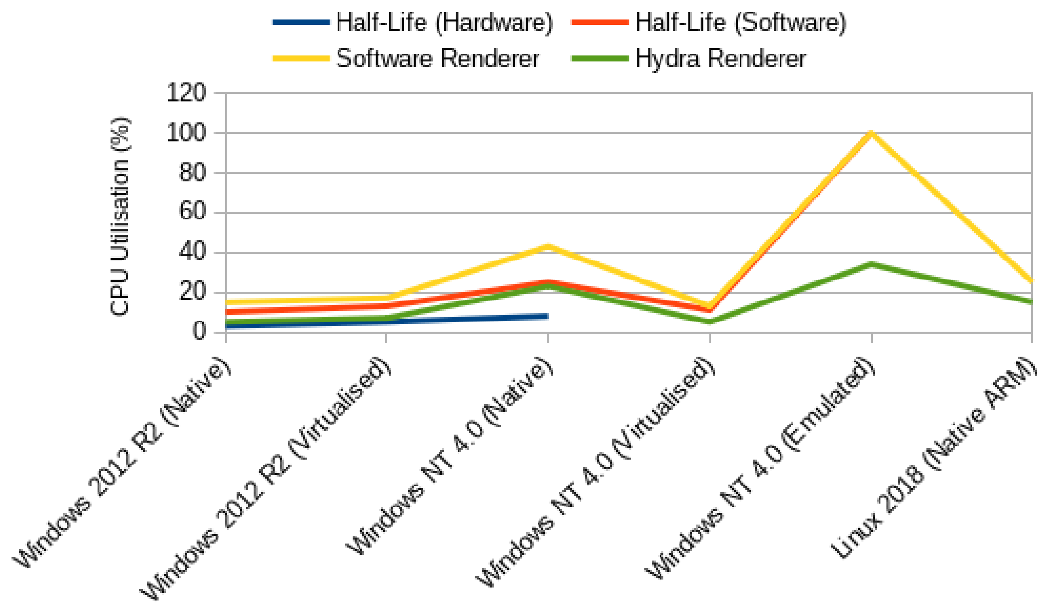

Figure 6.5: *Graph showing the CPU requirement of different renderers running at 800x600 resolution on the different test platforms.*

The results show a wide range of different performances. An interesting note is that the 3D renderer running on the Raspberry Pi 3 has a lower CPU utilisation than the native Windows NT 4.0. Presumably, this is due to the fact that although the older Intel (Pentium III) is running at a higher clock speed, the use of parallelisation in the ARM Cortex-A53 hardware (4 cores) appears to make quite a difference. Even though they are all running at playable speeds the CPU utilisation of the Hydra and virtualised renderer are among the lowest. There is however a large gap between them showing that the virtualised GPU solution is the fastest.

As explored previously, the increase of screen resolution had a large impact on the performance of the software renderers. **Figure 6.6** shows the performance data of the two faster rendering solutions at increasing resolutions.

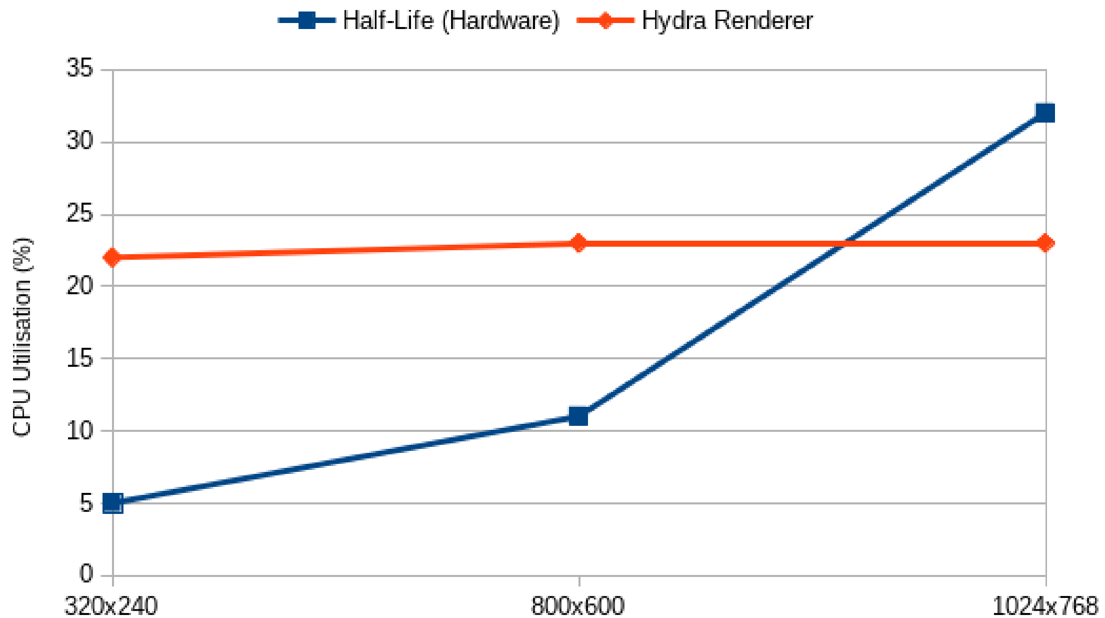

Figure 6.6: *Graph showing the CPU utilisation required by various resolutions for the accelerated renderer compared to the Hydra renderer running on Windows NT 4.0 operating system on native hardware.*

As these results demonstrate, the increase in resolution has virtually no effect on the Hydra based renderer, whereas the native renderer has a rapid increase of utilisation as the resolution increases. Such a large utilisation increase in the accelerated renderer was fairly unexpected because both rendering strategies involve offloading the processing away from the CPU and onto a different device. This in both cases is the GPU (in the case of Hydra, it passes it out of the virtualised environment completely) and, as such, the CPU was predicted to be idle with a low utilisation for much of the time when the GPU is processing more data. One potential reason is that the older OpenGL implementation for Windows NT 4.0 uses a fixed function pipeline with potentially an immediate mode drawing strategy which will keep the processor busy. Regardless, these results are very positive for Hydra, especially for resolutions higher than 800x600 in which the performance of Hydra surpasses the native rendering in terms of performance. In environments where a virtualised GPU is not available such as the vast majority of PC emulators, this is a strong use-case

for Hydra. This can certainly be seen in **Figure 6.7**, where the Hydra renderer is compared with the software renderers all running in a fully emulated environment where no accelerated GPU is available.

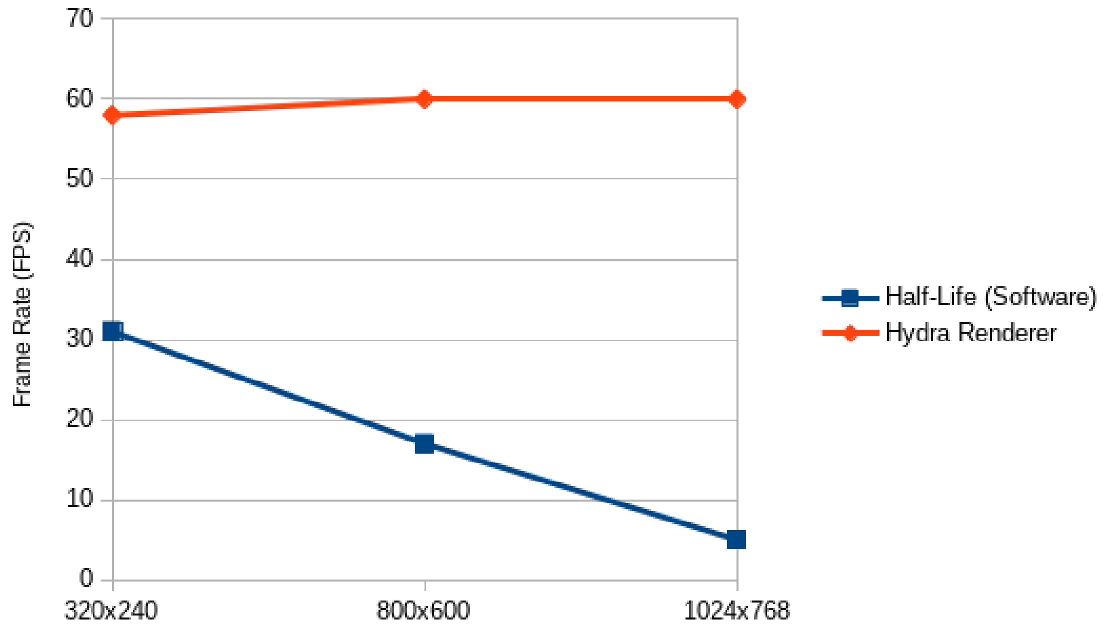

Figure 6.7: *Graph showing the CPU requirement of the commercial software renderer vs the Hydra renderer at various screen sizes whilst running in a fully emulated environment.*

Compared with the commercial software renderer in Half-Life, the rendering approach by Hydra is much more scalable. Whilst exhibiting a higher CPU utilisation than native or virtualised execution, it allows for increasing resolutions without any major drop in frame rate or increase in CPU utilisation. A result that first appeared as an anomaly but ended up being reproducible is that when running in the lowest resolution (320x240), Hydra ran slightly slower than the others. Whilst still at very playable speeds, it appeared to be an odd result. An initial explanation could be that the reduced time required to render a frame on the resulting host via Hydra meant that more messages were sent across the socket to the emulated environment where they are relatively expensive to process. This additional stress could be taking resources away from other tasks and thus contributing to the slightly decreased frame

rate. Additional investigation may be required here.

One reason as to why Hydra running in an emulated environment is predictably slower than native execution is because the game itself such as collision and physics is taking its toll in the emulated environment. If this is turned off, the results in **Figure 6.8** demonstrate the difference in performance.

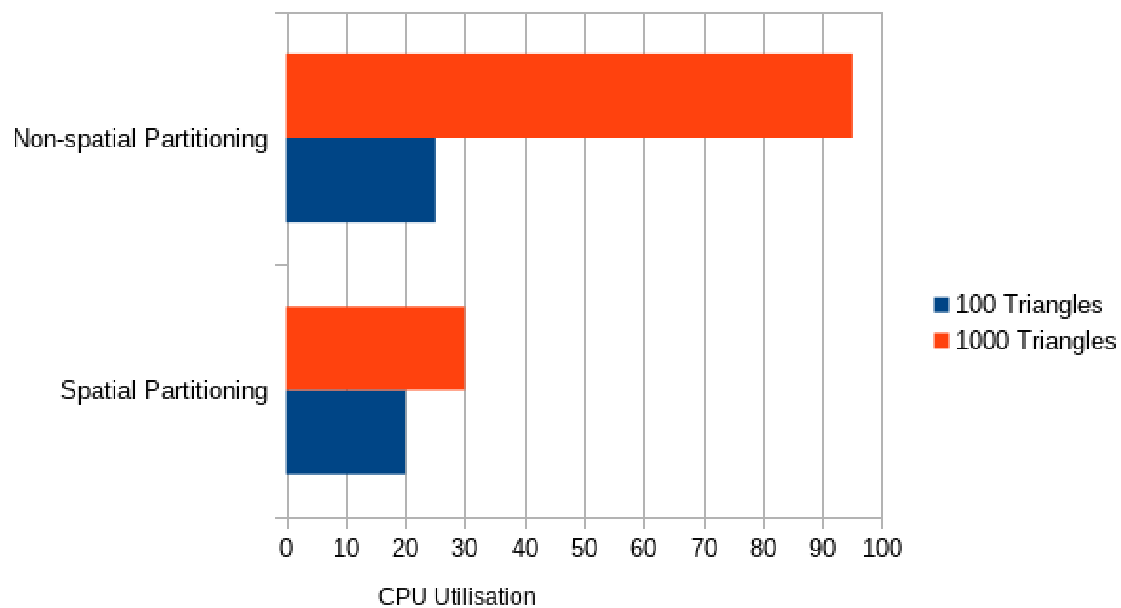

Figure 6.8: *Graph showing the CPU resource utilisation for collision. Note that rendering had been disabled for this test.*

This allowed for a small increase in performance but also shows that the collision and physics calculations are far less resource-intensive compared to the rendering.

6.1.3 Conclusion

The results provided by this experiment provide evidence that a renderer based upon Hydra has been the only one feasible that can provide a playable frame rate on all of the test platforms. Whilst the performance of Hydra is consistently lower than that of the virtualised GPU approach; the latter was not available for a large proportion of the platforms tested. This high compatibility of Hydra is one of its key strengths.

6.2 Interactions with Virtual Machines

If Hydra was to provide an adequate solution for digital preservation, it was deemed important that it can work along-side emulation because currently this is seen as the most successful solution.

Specific focus was given to GtkRadiant, a 3D level editor for the Quake III Engine as a case study because it was suitably complex in terms of user-interface, input and was starting to show signs of maintenance failure on more recent Windows platforms. In particular graphical compatibility mode had to be enabled in order to run it (Shown in **Figure 6.9**). It was also important to experiment with the functionality of Hydra along with tools rather than games because in many cases, tools interface with more of the operating system (such as filesystem, external utilities, etc). Due to this larger coverage of integration with the underlying operating system, tools can also be considerably more difficult to port to other platforms and so provides an even more useful application for Hydra.

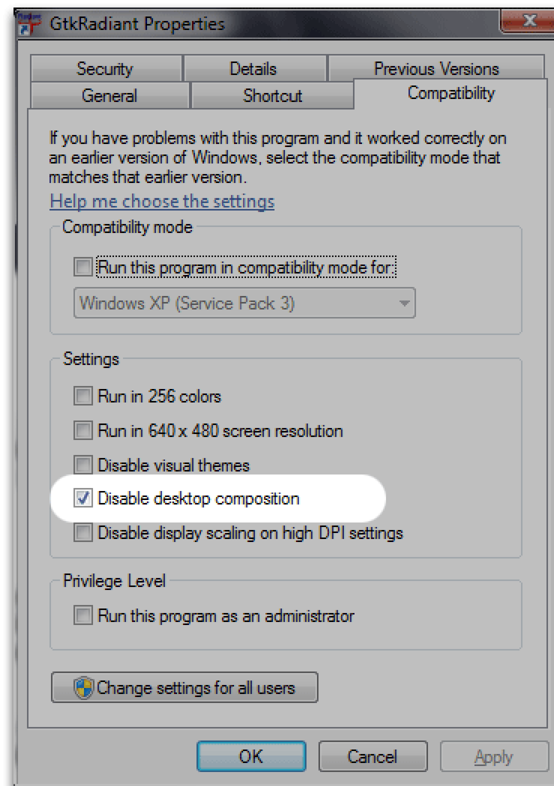

Figure 6.9: *Dialog showing a limited selection of tweaks that can be performed in order to run old software. Unfortunately there is no guarantee that these compatibility features will exist on future versions of Windows.*

Running this tool within an emulator is also initially not possible because it has a strict requirement on OpenGL 2.1 which the Windows software drivers do not provide. As discussed previously, unlike virtualisation, emulators do not provide accelerated 3D graphics drivers either and so it seems that this program cannot run without additional facilities.

The program was instead run using Hydra as a drop-in replacement to the default OpenGL implementation. The emulators network ports were subsequently forwarded to the host and the web browser was pointed towards the correct address (in this case it was localhost because the emulator was running on the same host as the client web browser). The setup worked and the graphical output of the tool could be seen on the host (**Figure 6.10**).

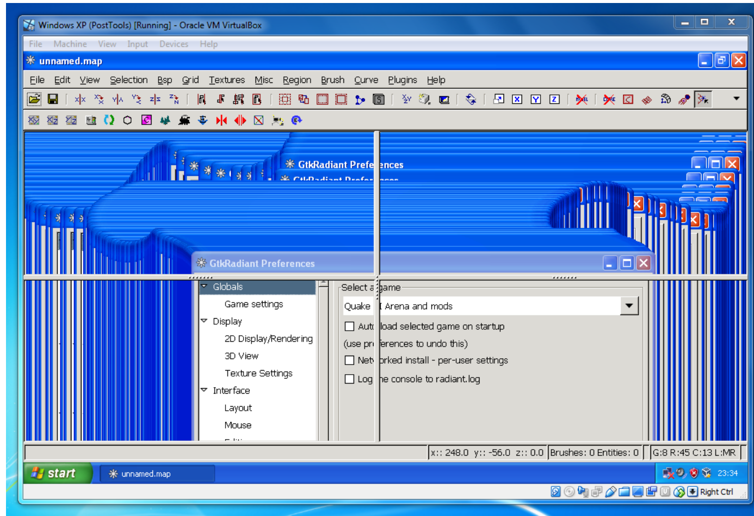

Figure 6.10: *Screenshot showing the graphical calls being externally diverted from the VM and into a web browser running natively on the host. Note the garbage on the GtkRadiant window because it is never being rendered to or cleared.*

This was a fairly positive result; the graphics had been preserved and the tool was responsive. However there were some side effects which can be seen in the screen shot. The tool within the emulator shows a black (or corrupted) screen because the OpenGL is no longer rendering to the buffer or clearing it. Usually this would not be a problem because it can be ignored and the user interfaces with the main graphical output in the client browser. Unfortunately the user is required to click and drag on the tools GUI window itself because this is what handles the user input. These cannot be forwarded to the client web browser in any meaningful way without modifying the application source code. The design of Hydra is such that it only deals with graphical output. This meant that it was rather awkward to use the tool itself in this state. However, in this case, as an interim solution, Hydra was modified in such a way that rather than outputting to the client web browser, it would output to a raster buffer and forwarded back into the emulator where it was displayed in place of the original unused surface. An overview of this solution can be seen in **Figure 6.11**.

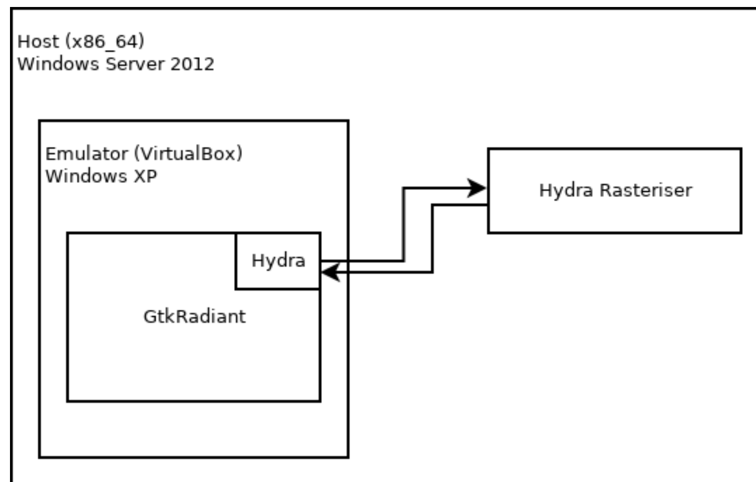

Figure 6.11: *Diagram showing the flow of data between Hydra running in the emulator and the standalone Hydra rasteriser running natively on the host.*

This specific solution was fairly ad-hoc but shows the power and flexibility of Hydra as a design when it comes to digitally preserving fairly complex 3D software, including many elements of their usability. Most importantly, it makes it possible to use whereas before, it was simply not able to run within the emulator. The final output can be seen in **Figure 6.12**.

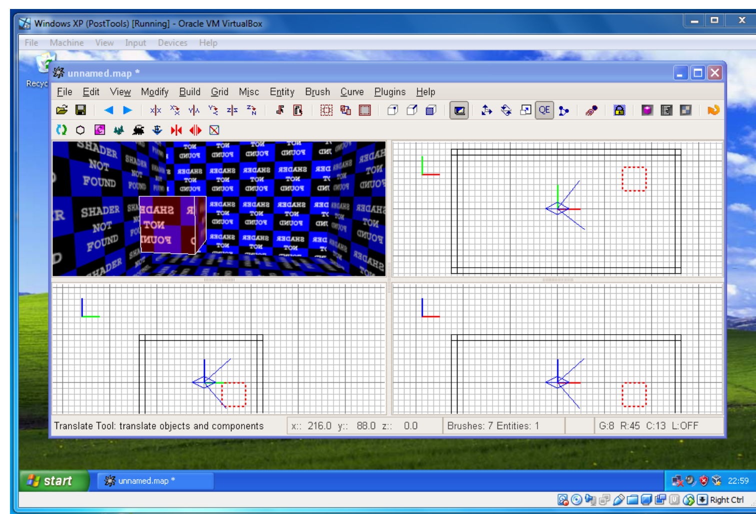

Figure 6.12: *Screenshot showing the final result of the external raster solution. Whilst it did require modifications to the GtkRadiant code; it now looks and feels native running in the emulator.*

This mechanism isn't necessarily limited to backwards compatibility, it can also provide a solution to run more modern applications on a host machine. For example if a specific application requires the latest version of an operating system to function and yet the hardware required to support that modern operating system is not available, it is possible to run that operating system in a Virtual Machine. Due to the nature of this setup, it is very likely that the virtualiser or emulator was originally developed without support for the (as of then, unreleased) modern operating system running as a guest. This in turn means that virtual passthrough GPU drivers would potentially be unavailable. Hydra provides a suitable solution to this problem by simply avoiding the need for platform specific drivers. This scenario can occur in various forms; especially with the growing trend in rolling release updates and cloud services. For example Valve's Steam digital store service can no longer be run on Windows XP hosts. This is fairly crucial because this acts as a gateway to the purchased software which can still function well on the older Windows release. In fact this software was designed for and originally purchased with the intention to run on this specific release of Windows. When attempts to access the purchased software by running Steam, it results in the error message shown in **Figure 6.13** and is in essence, a fairly artificial restriction. If a Windows XP era machine cannot be upgraded to the latest Windows 10 due to the ageing hardware, a virtual machine along with Hydra could provide a solution. This is particularly important because this whole issue has arisen as evidence that companies such as Valve do not currently have an invested interest in digital preservation or the lifespan of software. And yet, games themselves are very good candidates for software in which preservation attempts are often made.

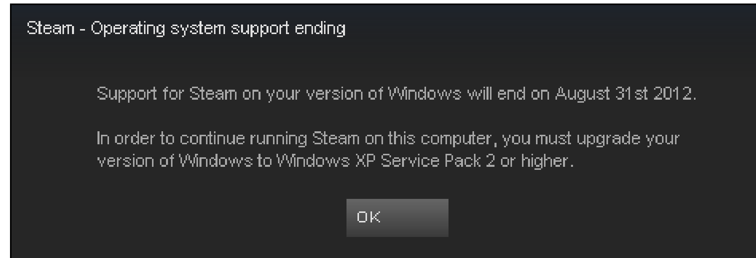

Figure 6.13: *Screenshot showing various generations of Steam platform deprecations. Often the games themselves have not been updated; just the Steam launcher which unfortunately is mandatory for DRM and anti-piracy reasons.*

Another more subtle example of this is that the runtimes for a newer version of software require newer functionality in the operating system kernel. This would result in the error seen in **Figure 6.14**.

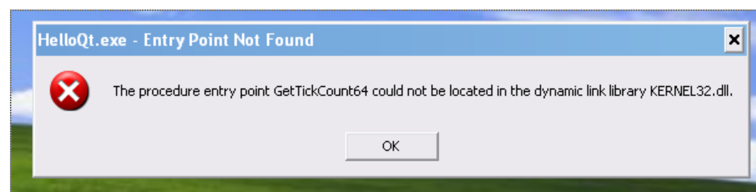

Figure 6.14: *The resulting dialog when trying to run a recent Qt application, compiled with the latest GCC based MinGW compiler.*

If upgrading the operating system is not possible, due to hardware constraints or compatibility with the other software packages, Hydra will help facilitate the ability to run the incorrectly functioning software in an emulator until the rest of the components are updated.

6.3 Summary

The results provided in this chapter give clear indication that there exists an issue in terms of rendering performance within emulators and that the technology behind Hydra has provided an acceptable solution. Starting with the initial results that suggest the GPU hardware reduces stress on the CPU, even in non-emulated environments. This is expected because the hardware is more suited for the purpose. Later results then show that at higher resolutions, software renderers start to become infeasible in terms of over CPU utilisation leading to decreasing frame rates. This is ultimately the case even with high levels of platform specific optimisation as provided by the Half-Life software renderer. There are still a number of platforms that do not support hardware acceleration, particularly when dealing with emulated environments of older platforms. This is demonstrated in the results that Half-Life with the hardware renderer only runs on a small fraction of the platforms tested. The results also demonstrate that emulated platforms in particular do not run software renderers well, even at very low resolutions because each instruction is fully emulated providing a large amount of overhead.

When compared with Hydra, the Half-Life hardware renderer is generally favourable on the platforms it supports, however even at the lowest resolutions, the Half-Life software renderer is considerably slower than Hydra. In all tests, the frame-rate and CPU utilisation of Hydra remained stable, showing that it can also provide a good experience, even at larger screen sizes on a much wider selection of platforms than are supported by the direct use of the hardware renderer.

The results demonstrate that the Hydra renderer can relieve the stress on the CPU by avoiding emulated software rendering. This frees the CPU resources for other simulation or game specific tasks such as the collision. In the experiment conducted it was shown that whilst collision can become very expensive, in typical use-cases and some amount of optimisation (spatial partitioning), the performance is entirely

acceptable for a viable user experience.

This chapter also demonstrated how Hydra can be used directly with a virtual machine software (Virtual Box) to provide a rendering service that can be sent back within the machine. This flexibility was used to maintain an older development tool which otherwise had issues running on more modern platforms.

Chapter 7

Conclusion and Future Research

7.1 Review of the Goals of this Research

The aim of this research was to develop a solution which can solve many of the complexities inherent in the digital preservation of 3D content such as games. The implementation of a generic graphical passthrough mechanism could help facilitate platform agnostic development and provide a long term solution to maintaining future software titles.

As well as the investigation of existing methods of maintaining and porting software, a research objective was to produce a practical technology that can be utilised by other developers in order to help future-proof their own products.

This technology was to be built using a very disciplined platform agnostic development methodology pertaining to portability. In particular exploring the use of a very strict subset of ANSI C in order to develop the software to a fairly high quality whilst at the same time making zero compromise to the cross-platform support. This solution was to be subsequently evaluated in terms of performance and applicability to 3D software development.

Additional functionality, especially relating to multi-user synchronisation (such as

multi-player games) was to be explored and any findings to be documented and evaluated.

7.2 Summary of Research

A number of different porting techniques were examined in Chapter 2.12.3 with a general consensus that emulation had the potential to be the most successful for a vast number of use cases. The emulation of games was investigated, showing limited access to the GPU was one of the largest constraints to emulating 3D gaming titles. In an attempt to solve this problem, a new solution called Hydra was developed to allow access to the GPU in a very portable and flexible manner. This solution was developed with an emphasis on maintainability and portability and, as such, a number of architectural decisions had to be made, including an in-depth analysis of language portability which in many ways went against trends within the industry. The implementation was discussed in Chapter 3.2.

Hydra provided not only a new alternative to help facilitate digital preservation but also a framework for similar technologies relating to different fields of computer hardware interoperability. Potentially this solution is not just limited to the GPU but to any modern hardware devices on the host machine, such as audio or USB connectivity.

The performance of Hydra in an emulator compared with platform specific GPU passthrough and software rendering was analysed in Chapter 6. Here the results demonstrated were very promising.

The multi-user aspects of Hydra were also documented in Chapter 4, including its use within a number of multi-player tech demos and games. A number of publications were also written.

Hydra was measured against common streaming techniques and software rendering, where it yielded results where, in the correct circumstances, could be used within a

production context in order to replace streaming technologies in a wide number of areas, not just entertainment and games.

7.3 Conclusion

Based on the outcome of Hydra, it is certain that the digital preservation of 3D software can be achieved by forwarding platform specific functionality from inside an emulator as a protocol to the host through a medium such as a network interface. In many cases, the loose coupling of the application logic and the renderer has shown to potentially be one of the very few feasible solutions to running older 3D titles on modern platforms.

This new method of improving the portability of software has so far been experimented by testing against existing platforms and back-porting to legacy platforms. It had been decided that this was the best approach to evaluating the effectiveness of Hydra as a solution. However, interestingly, it is very difficult to say for certain, what kinds of challenges to portability the future technological environment may bring. For example, making a 3D game work effectively today within an emulator running MS-DOS may end up proving to be a very challenge compared to running that same game on a computer 50 years from now.

However, based on the output of this thesis and the trends so far discovered by connecting the older platforms to the modern host, there is evidence to strongly suggest that reducing the number of dependencies (including language dependencies) is likely to be an important factor in the future portability of many software projects. Particularly working within the constraints of ANSI C as a conduit to portability.

Along with using C, it is fairly difficult to perform a viable usability study to measure the ease of use of this new technology. This is because unlike products such as Unity, the powerful platform agnostic functionality comes from using a low level library and due to its nature could be deemed more complex to use than a full fledged game

engine. However, due to the successful approach of cloning the existing OpenGL API rather than designing a new one from scratch, this comparison with an engine becomes irrelevant because the engine could potentially run on top of Hydra. If a 3D graphics developer was also to use both OpenGL or Hydra directly, they would notice very few (if any) differences between the two technologies in terms of use.

It is often suggested that focusing too much on portability is detrimental to a project within the industry (Hook, 2005). This is one of the areas where a technology such as Hydra has demonstrated that it allows a developer to make a compromise. For example it allows them to fall back to emulation which greatly simplifies the development workload whilst at the same time allowing the software to run on a large number of platforms. At the same time it also offers them the ability to access the native host's capabilities if strictly dictated by the project's requirements.

Along with the contributions highlighted in Chapter 4 that this research has provided in the area of multi-user synchronisation, Hydra has also provided a new novel approach to preserving 3D software. By leveraging the idea of streaming instructions, which in itself is not new, but in a specific way that allows for access to the host from within any generic emulation technology. The result is that software and games that would otherwise be unplayable via other solutions can be made to function and with an acceptable performance. The implementation process of Hydra itself also provides a number of contributions, including an analysis of the most important aspects of development which contribute towards portability and most importantly, those that detract from it. A number of novel solutions with regards to memory correctness and technical usability were also designed which allow for the safe use of ANSI C. These are not specific to the implementation of Hydra and can be utilised by any project.

By referring the goals outlined in **Section 1.5** the output of this research can successfully be validated.

- **A literature review has been conducted** and a number of different approaches currently in use have been identified, including their potential

limitations. From this it has been discovered that there are still a number of unsolved areas relating to keeping software alive. This research has led to the contribution of a new approach called Hydra which can solve a number of these issues, particularly by providing a solution to graphical passthrough that can work in all emulators, including those with less man-power maintaining them or for exotic alternative architectures.

- **The implementation of a platform agnostic technology** was undertaken and the output; Hydra successfully provides the ability to pass graphical information between the emulator and host environment whilst at the same time providing an almost identical API to that of OpenGL. This contribution is in the form of a library that can in many ways serve as a drop-in replacement to OpenGL and yet allow a project to instantly benefit from its attributes of being able to stream the graphical calls through a virtual machine boundary.

- **A safety framework for ANSI C was developed** that not only provided the fundamental building blocks for Hydra allowing for an effective implementation but it also paves the way for more greenfield development projects to utilise the often unmatched portability benefits of the C programming language. The libstent safety framework can notify the developer of a larger proportion of memory errors compared to existing tools without reducing flexibility of the language. This contribution is in the form of a library that can be integrated with any project which will then alert the developer to a range of memory errors in a deterministic manner as soon as they occur.

- **Evidence showing the viability of Hydra has been presented** and compared against existing solutions. The results show a greater performance of graphics rendering from software running within an emulated environment using Hydra compared to software rendering approaches. The results provided also show that Hydra can cater for platforms which traditional GPU passthrough

cannot whilst still retaining competitive performance. The contribution provided is the idea that GPU passthrough does not necessarily need to be implemented by the emulator vendor but instead by the individual software packages running within it. This gives greater flexibility and also reduces reliance on a specific emulation platform.

- **Evidence demonstrating that multi-user software can benefit from Hydra has been provided** which compares existing VNC streaming approaches with that of Hydra. In particular when streaming at high resolutions, the intelligent protocol approach provided by Hydra can greatly reduce the required bandwidth. Likewise when compared with QuakeWorld in a scenario requiring many intricate state synchronisations, Hydra can also serve as an alternative by reducing two-way communication between the client and server, and instead streaming the final graphics output. The contribution in this area is the demonstration that streaming with an intelligent protocol can reduce the bandwidth requirements for a number of use-cases and yet can remain suitably generic so that the same protocol can be used for a wide range of software rather than bespoke for individual software packages, such as Microsoft's Remote Desktop.

The direct use of Hydra in Section 6.2 to specifically address the issue of GtkRadiant not being able to execute correctly within the emulated environment is also a useful demonstration that the presented research has a direct practical use. This is a good example of face validity in that even during the relatively early stages of development, Hydra has provided not only measurable results but also provided a solution to a problem that did not have a viable answer, other than perhaps a rewrite of the relevant software. It is hoped that this face validity will encourage others to utilise this software for their own projects in an attempt to reduce maintenance burden and keep more software alive for longer.

7.4 Future Research

7.4.1 Limitations of Hydra

Due to the nature of Hydra, there are a number of issues that could potentially be exposed by a number of specific use cases. Firstly, the protocol of Hydra attempts to reduce the number of messages between the guest and host, however, in certain circumstances, mostly common in very old gaming titles when the immediate mode of OpenGL was common, there can be a very large number of commands because every vertex to draw was specified per frame. This style of code is not particularly fast on graphics hardware, which was the reason why it has been deprecated. However, the inefficiencies are greatly exaggerated by the addition of a network protocol. It seems that older titles render far fewer polygons than modern titles, however, if a large number of polygons was to be rendered using this form of architecture, performance would be greatly impacted. This issue was not explored further because immediate mode rendering is generally seen as deprecated and is therefore quite a rare use-case within modern titles, especially those rendering larger numbers of polygons.

Whilst Hydra itself, as an isolated component has proven to be useful and provides a solution to a complex problem in the area of platform agnostic development and future proofing software. However, this problem itself is a relatively small part of a potentially much larger problem. If Hydra was used but then bound to much larger software platforms, such as Java and complex platform specific build systems, the flexibility and portability of Hydra's design is quickly diminished. Whilst some of the benefits of a network-aware graphical protocol are still very useful for a number of use cases, it will no longer provide the project with instant portability.

If the use of Hydra is coupled with the strict subset of ANSI C, identified in Chapter 3.2 then this specific issue could be resolved. However by enforcing this type of development could severely impact development workflow and prevent the uptake of

Hydra in general.

Another potential limitation is that Hydra does not benefit from the latest breakthroughs in graphics technology. Using the OpenGL architecture there are a number of limitations inherent in this approach. For example, the optimisations and flexibility of newer modern standards such as Vulkan cannot be exploited. With the introduction of threads much higher performance can be achieved; this however is not feasible with the state machine driven approach of OpenGL. This limitation does not affect the use of Hydra for digital preservation and the design of Hydra could even be extended to include technologies such as Vulkan over time as it is needed.

To this point we can assume that graphics cards are becoming more flexible; this is fine because it means that Hydra can utilise them by providing a corresponding client, such as the reference WebGL client developed during this research. However, if graphics cards ever became less flexible or no longer provide a programmable pipeline, this could cause an issue because they could lack the functionality to provide full OpenGL 2.x compliance. This was not explored because it is unlikely at this point, however, in reality we do not know what is going to happen in the future. It could also limit Hydra's usage to more niche areas such as the embedded or safety sector, where limited graphical support is common. A specific example of this is the OpenGL|SC (OpenGL Safety Critical) profile (**Figure 7.1**) that involves the removal of much of the OpenGL functionality to reduce complexity and dependencies in order to help facilitate testing and verification.

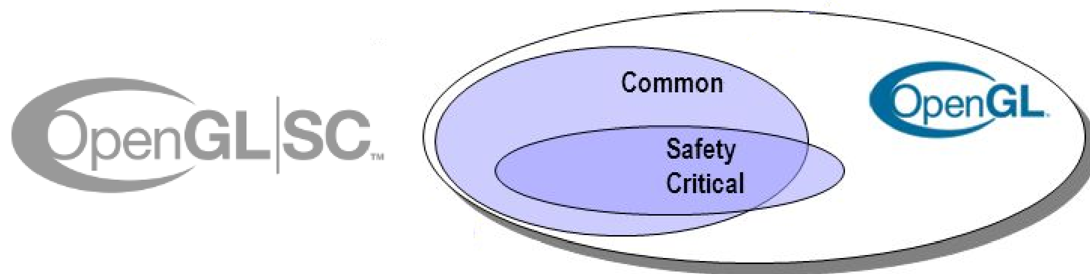

Figure 7.1: *Diagram from Khronos showing a simple overview of how the OpenGL Safety Critical profile aligns to the commonly used OpenGL profile (Khronos Group, 2003 (accessed February 9, 2015)).*

7.4.2 An Audio Equivalent to Hydra

So far, the sole focus of Hydra has been to passthrough graphical functionality to the native host. However, it is entirely possible to achieve a similar effect with other systems, such as USB, file systems and audio, to name a few.

Audio in particular could be a very useful addition to Hydra because it is used extensively in games to provide immersion. It is extremely rare that a game is released without audio.

Whilst it is true that most emulators provide audio in terms of hardware they emulate there is still potential benefit in explicitly streaming this data to the host. For example, audio will successfully play from within the emulator whilst the graphics themselves are rendered on the host. However if the native host is not the same machine running the emulator, this is unlikely to be heard. This is the likely outcome if Hydra is being used for multi-player functionality or a cloud streaming service.

Another example of where explicit audio passthrough could be useful is to avoid the lowest common denominator approach to emulation. For Example, it is highly unlikely that most emulation software will support surround sound to multiple speakers. However if the audio data is streamed to the host in a similar fashion to Hydra, i.e using an API such as OpenAL, then the host could utilise the audio in

a much more integrated way. This is because it would have more knowledge about the host system; even if the client rendering the graphics and playing the audio is implemented on top of HTML 5, as is the case with the reference implementation presented in this research.

7.4.3 A Game Engine Built Upon Hydra

For the purposes of this thesis, there has not been much additional exploration of the potentially new areas of research that a network aware graphical protocol has exposed. In particular, by developing a game engine specifically targeting Hydra as the graphical API, instead of standards such as OpenGL, DirectX or even Vulkan, it might be possible to facilitate an extensible and safe multi-user platform within the very core of the engine itself.

For example, many existing game engines provide an API to access multiple cameras or different input devices, however, a game engine built upon Hydra could take this concept further by providing multiple cameras but, instead of rendering to different render textures, they could render to different clients, instantly allowing players to see from different view points. Likewise, instead of reading input from a keyboard or gamepad, input could be read from keyboards and gamepads attached to different clients, allowing much better contextual information of the messages passed so that they can be handled in potentially new and innovative ways with minimal complexity. With just these two ideas, the majority of games developed with split screen local multi-player in mind could be ported to network multi-player with a reduced number of changes to the source code being required.

Bibliography

Abdelkhalek, A. and Bilas, A., 2004. Parallelization and performance of interactive multiplayer game servers. *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 72–.

Abdelkhalek, A., Bilas, A. and Moshovos, A., 2003. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6 (4), 355–366. URL https://doi.org/10.1023/A:1025718026938.

Anderson, P. K. et al., 2013. Virtualisation for local cloud computing: Dwu data centre. *Contemporary PNG Studies*, 18, 15.

Aroca, R. V. and Gonçalves, L. M. G., 2012. Towards green data centers: A comparison of x86 and arm architectures power efficiency. *Journal of Parallel and Distributed Computing*, 72 (12), 1770–1780.

Asanović, K. and Patterson, D. A., 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*.

Austern, M., 2005. Draft technical report on c++ library extensions. *ISO/IEC DTR*, 19768.

Back, G. and Hsieh, W., 1999. Drawing the red line in java. *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, IEEE, 116–121.

Baratto, R. A., Kim, L. N. and Nieh, J., 2005. Thinc: a virtual display architecture for thin-client computing. *ACM SIGOPS Operating Systems Review*, ACM, volume 39, 277–290.

Bartholomew, D., 2006. Qemu: a multihost, multitarget emulator. *Linux Journal*, 2006 (145), 3.

Bass, D., 2014 (accessed July 9, 2019). Six things you need to know about atms and the windows xp-ocalypse. URL `https://www.bloomberg.com/news/articles/2014-04-03/six-things-you-need-to-know-about-atms-and-the-windows-xp-ocalypse`.

Baughman, N. E. and Levine, B. N., 2001. Cheat-proof playout for centralized and distributed online games. *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, IEEE, volume 1, 104–113.

Beazley, D. M. et al., 1996. Swig: An easy to use tool for integrating scripting languages with c and c++. *Tcl/Tk Workshop*, 43.

Bellard, F., 2005. Qemu, a fast and portable dynamic translator. *USENIX Annual Technical Conference, FREENIX Track*, volume 41, 46.

Berners-Lee, T., 2015 (accessed April 3, 2019). Web security - tls everywhere, not https: Uris. URL `https://www.w3.org/DesignIssues/Security-NotTheS.html`.

Bernier, Y. W., 2001. Latency compensating methods in client/server in-game protocol design and optimization. *Game Developers Conference*, volume 98033.

Bishop, J. and Horspool, N., 2006. Cross-platform development: Software that lasts. *Computer*, 39 (10), 26–35.

Blazakis, D., 2011. The apple sandbox. *Arlington, VA, January*.

Blem, E., Menon, J. and Sankaralingam, K., 2013. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, IEEE, 1–12.

Bloice, M. D., Wotawa, F. and Holzinger, A., 2009. Java's alternatives and the limitations of java when writing cross-platform applications for mobile devices in the medical domain. *Proceedings of the ITI 2009 31st international conference on information technology interfaces*, IEEE, 47–54.

Blom, S., Book, M., Gruhn, V., Hrushchak, R. and Köhler, A., 2008. Write once, run anywhere a survey of mobile runtime environments. *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, 132–137.

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

Buyya, R., Yeo, C. S. and Venugopal, S., 2008. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. *2008 10th IEEE international conference on high performance computing and communications*, Ieee, 5–13.

Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T. and Lucena, C., 2006. Composing design patterns: a scalability study of aspect-oriented programming. *Proceedings of the 5th international conference on Aspect-oriented software development*, ACM, 109–121.

Canon, M. D., Fritz, D. H., Howard, J. H., Howell, T. D., Mitoma, M. F. and Rodriquez-Rosell, J., 1980. A virtual machine emulator for performance evaluation. *Commun. ACM*, 23 (2), 71–80. URL http://doi.acm.org/10.1145/358818.358821.

Caplan, P., Halbert, M., Horton, R. and Surface, T., 2005. Is digital preservation an oxymoron? *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '05)*, 159–159.

Carcerano, C. J., 1998. Processing object oriented code and virtual function code. US Patent 5,764,991.

Carnall, B., 2016. *Unreal Engine 4. X By Example*. Packt Publishing Ltd.

Chęć, D. and Nowak, Z., 2018. The performance analysis of web applications based on virtual dom and reactive user interfaces. *KKIO Software Engineering Conference*, Springer, 119–134.

Chen, D., Jarrett, P. B. and Chatham, H. L., 2018. Managing metadata in cloud driven, thin client video applications. US Patent App. 15/359,869.

Chen, S.-S., 2001. The paradox of digital preservation. *Computer*, 34 (3), 24–28.

Christodorescu, M., Sailer, R., Schales, D. L., Sgandurra, D. and Zamboni, D., 2009. Cloud security is not (just) virtualization security: a short paper. *Proceedings of the 2009 ACM workshop on Cloud computing security*, ACM, 97–102.

Claypool, M. and Claypool, K., 2009. Perspectives, frame rates and resolutions: it's all in the game. *Proceedings of the 4th International Conference on Foundations of Digital Games*, ACM, 42–49.

Claypool, M., Claypool, K. and Damaa, F., 2006. The effects of frame rate and resolution on users playing first person shooter games. *Multimedia Computing and Networking 2006*, International Society for Optics and Photonics, volume 6071, 607101.

Cleraux, C. and Perrin, B., 2011. Method and system for debugging of software on target devices. US Patent App. 12/696,381.

Collberg, C., 2018. Code obfuscation: Why is this still a thing? *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, New York, NY, USA: ACM, CODASPY '18, 173–174. URL `http://doi.acm.org/10.1145/3176258.3176342`.

Conley, J., Andros, E., Chinai, P. and Lipkowitz, E., 2003. Use of a game over: emulation and the video game industry, a white paper. *Nw. J. Tech. & Intell. Prop.*, 2, 261.

Cordeiro, D., Goldman, A. and da Silva, D., 2007. *Euro-Par 2007 Parallel Processing, 13th International Euro-Par Conference, Rennes, France, 2007*, Springer.

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

Curtin, M., 1998. Write once, run anywhere: Why it matters. *Technical Article. http://java. sun. com/features/1998/01/wo*.

Detlefs, D., Dosser, A. and Zorn, B., 1993. *Memory allocation costs in large C and C++ programs*. Department of Computer Science, University of Colorado.

Ding, J.-H., Chang, P.-C., Hsu, W.-C. and Chung, Y.-C., 2011. Pqemu: A parallel system emulator based on qemu. *Parallel and distributed systems (icpads), 2011 ieee 17th international conference on*, IEEE, 276–283.

Dolenc, A., Lemmke, A., Keppel, D. and Reilly, G., 2000. Notes on writing portable programs in c. *CS&E University of Washington pardo@ cs. washington. edu*, 1595.

Downing, S., 2011. Retro gaming subculture and the social construction of a piracy ethic. *International Journal of Cyber Criminology*, 5 (1).

Dowty, M. and Sugerman, J., 2009. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43 (3), 73–82.

Doyle, J., Viktor, H. L. and Paquet, E., 2007. Long term digital preservation - an end user's perspective. *2007 2nd International Conference on Digital Information Management*, volume 1, 146–151.

Drachen, A., Bauer, K. and Veitch, R. W. D., 2011. Distribution of digital games via bittorrent. *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, New York, NY, USA: ACM, MindTrek '11, 233–240. URL http://doi.acm.org/10.1145/2181037.2181077.

Egan, A., 2019 (accessed July 9, 2019). Why new york's subway still uses os/2. URL https://tedium.co/2019/06/13/nyc-subway-os2-history.

Ehringer, D., 2010. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4 (8).

Facey, P. V. and Gaines, B. R., 1973. Real-time system design under an emulator embedded in a high-level language. *Proc. BCS DATAFAIR*, 73, 285.

Färber, J., 2004. Traffic modelling for fast action network games. *Multimedia Tools and Applications*, 23 (1), 31–46. URL http://dx.doi.org/10.1023/B:MTAP.0000026840.45588.64.

Feeley, M. and Dubé, D., 2003. Picbit: A scheme system for the pic microcontroller. *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, Citeseer, 7–15.

Filardo, N. W., 2007. Porting qemu to plan 9: Qemu internals and port strategy.

Ford, B., 2012. Icebergs in the clouds: The other risks of cloud computing. *2012 HotCloud'12 Conference*, USENIX. URL https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/ford.

Gamespot, 2011 (accessed April 3, 2014). The final hours of half-life. URL http://uk.gamespot.com/features/halflife_final/part22.html.

Gawer, A. and Cusumano, M. A., 2002. *Platform leadership: How Intel, Microsoft, and Cisco drive industry innovation*, volume 5. Harvard Business School Press Boston, MA.

Genev, E., 2010. *Vnc-interface for java x86-emulator dioscuri*. Ph.D. thesis, University of Freiburg.

Giaretta, D., 2008. The caspar approach to digital preservation. *International Journal of Digital Curation*, 2 (1).

Glazer, J. and Madhav, S., 2015. *Multiplayer game programming: architecting networked games*. Addison-Wesley Professional.

GNU Compiler Collection, 2019 (accessed April 3, 2019). Common variable attributes. URL `https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html`.

Google, 2019 (accessed January 2, 2019). Android platform architecture. URL `https://developer.android.com/guide/platform`.

Gosling, J., 2002 (accessed January 2, 2019). Why microsoft's c# isn't. URL `https://www.cnet.com/news/why-microsofts-c-isnt`.

Granger, S., 2000. Emulation as a digital preservation strategy. 6.

Grønli, T.-M., Hansen, J., Ghinea, G. and Younas, M., 2014. Mobile application platform heterogeneity: Android vs windows phone vs ios vs firefox os. *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, IEEE, 635–641.

Grover, R. and Nolan, J., 2007. Method of porting software. US Patent 7,185,344.

Guan, M. and Gu, M., 2010. Design and implementation of an embedded web server based on arm. *2010 IEEE International Conference on Software Engineering and Service Sciences*, IEEE, 612–615.

Guttenbrunner, M., Becker, C. and Rauber, A., 2010. Keeping the game alive: Evaluating strategies for the preservation of console video games. *International Journal of Digital Curation*, 5 (1), 64–90.

Guttenbrunner, M., Kehrberg, C., Rauber, A. and Becker, C., 2008. Evaluating strategies for the preservation of console video games. *iPRES*, Citeseer.

Guttenbrunner, M. and Rauber, A., 2012. A measurement framework for evaluating emulators for digital preservation. *ACM Trans. Inf. Syst.*, 30 (2), 14:1–14:28. URL http://doi.acm.org/10.1145/2180868.2180876.

Halperin, Y., Chamcham, J., Leroy, C. M., CHEONG, G. I., ECCLESTON, M. and Feng, J., 2014. Extending server-based desktop virtual machine architecture to client machines. US Patent 8,640,126.

Hamilton, N., 2008 (accessed January 2, 2019). The a-z of programming languages: C#. URL https://www.computerworld.com.au/article/261958/a-z_programming_languages_c_.

Harris, T., 2005. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58 (3), 325–343.

Henderson, F., Conway, T. and Somogyi, Z., 1995. Compiling logic programs to c using gnu c as a portable assembler. *Proceedings of the ILPS*, volume 95, 1–15.

Henning, M., 2009. Api design matters. *Commun. ACM*, 52 (5), 46–56. URL http://doi.acm.org/10.1145/1506409.1506424.

Hilgart, M., 2006. Step-through debugging of glsl shaders. *School of Computer Science, DePaul University, Chicago, USA*, 2.

van der Hoeven, J., 2007. Dioscuri: emulator for digital preservation. *International Preservation News*, (43), 23.

Van der Hoeven, J., Lohman, B. and Verdegem, R., 2008. Emulation for digital preservation in practice: The results. *International journal of digital curation*, 2 (2).

van der Hoeven, J. R., Dindorf, M. and Sepetjan, S., 2010. Legal aspects of emulation. *iPRES*, Citeseer.

Holden, D., 2019 (accessed January 2, 2019). Cello garbage collection. URL `http://libcello.org/learn/garbage-collection`.

Hong, H.-J., Fan-Chiang, T.-Y., Lee, C.-R., Chen, K.-T., Huang, C.-Y. and Hsu, C.-H., 2014. Gpu consolidation for cloud games: Are we there yet? *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*, IEEE Press, 3.

Hook, B., 2005. *Write portable code: an introduction to developing software for multiple platforms*. No Starch Press.

Hsieh, G., Paruchuri, D., Steward, C., Nwafor, E. and Gadam, D., 2013. Lessons learned: Porting java applications to android. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 1.

Innocenti, P., Ross, S., Maceviciute, E., Wilson, T., Ludwig, J. and Pempe, W., 2009. Assessing digital preservation frameworks: The approach of the shaman project. *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, New York, NY, USA: ACM, MEDES '09, 61:412–61:416. URL `http://doi.acm.org/10.1145/1643823.1643899`.

Iqbal, A., Al Obaidli, H., Marrington, A. and Jones, A., 2014. Windows surface rt tablet forensics. *Digital Investigation*, 11, S87–S93.

James, S. R. and Gillam, B. D., 1999. Network multiplayer game. US Patent 5,964,660.

Jamraj, D., Huang, S., Neelakanta, P. and Alhalabi, B., 2017. Applications of an emulation model towards the preservation of modern computing systems. *2017 Annual IEEE International Systems Conference (SysCon)*, 1–8.

Jangda, A., Powers, B., Guha, A. and Berger, E., 2019. Mind the gap: Analyzing the performance of webassembly vs. native code. *arXiv preprint arXiv:1901.09056*.

Järvinen, K., Kolesnikov, V., Sadeghi, A.-R. and Schneider, T., 2010. Embedded sfe: Offloading server and network using hardware tokens. *International Conference on Financial Cryptography and Data Security*, Springer, 207–221.

Jones, M., 2009 (accessed January 2, 2019). Linux virtualization and pci passthrough. URL `https://developer.ibm.com/tutorials/l-pci-passthrough`.

Jones, S. P., Hall, C., Hammond, K., Partain, W. and Wadler, P., 1993. The glasgow haskell compiler: a technical overview. *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93.

Jones, S. P., Ramsey, N. and Reig, F., 1999. C-—: A portable assembly language that supports garbage collection. *International Conference on Principles and Practice of Declarative Programming*, Springer, 1–28.

Kaczmarek, J. and Kucharski, M., 2004. Size and effort estimation for applications written in java. *Information and Software Technology*, 46 (9), 589–601.

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

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

Kazama, T. and Miura, V. O. S., 2014. Adaptive load balancing in software emulation of gpu hardware. US Patent App. 13/631,803.

Keene, T., 2011. The apple barrier: an open source interface to the iphone. *EVA*.

Kernighan, B. W., 1996. A descent into limbo. *Online White-Paper. Lucent Technologies*, 12.

Kernighan, B. W. and Ritchie, D. M., 2006. *The C programming language*.

Khronos Group, 2003 (accessed February 9, 2015). Bringing 3d gaming to cell phones. Khronos Group.

Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y., 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*.

Kölling, M. and Rosenberg, J., 1996. An object-oriented program development environment for the first programming course. *ACM SIGCSE Bulletin*, 28 (1), 83–87.

Kowshik, S., Dhurjati, D. and Adve, V., 2002. Ensuring code safety without runtime checks for real-time control systems. *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 288–297.

Kuny, T., 1997. A Digital Dark Ages? Challenges in the Preservation of Electronic Information. 12.

Latif, M., Lakhrissi, Y., Nfaoui, E. H. and Es-Sbai, N., 2016. Cross platform approach for mobile application development: A survey. *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, IEEE, 1–5.

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

Lavoie, B. and Dempsey, L., 2004. Thirteen ways of looking at... digital preservation. *D-Lib magazine*, 10 (7/8).

Lee, K.-H., Slattery, O., Lu, R., Tang, X. and McCrary, V., 2002. The state of the art and practice in digital preservation. *Journal of research of the National institute of standards and technology*, 107 (1), 93.

Li, J., Li, C., Chen, G. and Tong, X., 2014. Vgpu: a real time gpu emulator. US Patent 8,711,159.

Li, Z., Zhang, H., O'Brien, L., Cai, R. and Flint, S., 2013. On evaluating commercial cloud services: A systematic review. *Journal of Systems and Software*, 86 (9), 2371–2393.

Liebetraut, T., Rechert, K., Valizada, I., Meier, K. and Suchodoletz, D. V., 2014. Emulation-as-a-service - the past in the cloud. *2014 IEEE 7th International Conference on Cloud Computing*, 906–913.

Lin, J.-H., Geiger, R., Wang, A., Wanchoo, S., Chan, A. and Smith, R., 2002. Method for permitting debugging and testing of software on a mobile communication device in a secure environment. US Patent App. 09/745,061.

Lindstrom, G., 2005. Programming with python. *IT professional*, (5), 10–16.

Linneman, J., 2018 (accessed March 3, 2019). Powerslave exhumed - the brilliant legacy of lobotomy software. URL `https://www.eurogamer.net/articles/digitalfoundry-2018-powerslave-exhumed-the-brilliant-legacy-of-lobotomy-software`.

Lischner, R., 2009. Smart pointers. *Exploring C++ The Programmer's Introduction to C++*, 567–580.

Liu, I., Wu, I. and Shann, J. J., 2015. Instruction emulation and os supports of a hybrid binary translator for x86 instruction set architecture. *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 1070–1077.

Lok, S., Feiner, S. K., Chiong, W. M. and Hirsch, Y. J., 2002. A graphical user interface toolkit approach to thin-client computing. *Proceedings of the 11th international conference on World Wide Web*, ACM, 718–725.

Longeray, P., 2015 (accessed February 9, 2015). Windows 3.1 is still alive, and it just killed a french airport. URL `https://news.vice.com/en_us/article/7xakd9/windows-31-is-still-alive-and-it-just-killed-a-french-airport`.

Luo, S., Lin, Z., Chen, X., Yang, Z. and Chen, J., 2011. Virtualization security for cloud computing service. *Cloud and Service Computing (CSC), 2011 International Conference on*, IEEE, 174–179.

Mace, D., Crowe, T. and Jones, J., 1974. The use of minicomputers in higher education in the united kingdom. *International Journal of Mathematical Educational in Science and Technology*, 5 (3-4), 543–548.

Matthews, B., Shaon, A., Bicarregui, J. and Jones, C., 2010. A framework for software preservation. *International Journal of Digital Curation*, 5 (1), 91–105.

Mayer, P. and Schroeder, A., 2014. Automated multi-language artifact binding and rename refactoring between java and dsls used by java frameworks. *European Conference on Object-Oriented Programming*, Springer, 437–462.

Microsoft Corporation, 2013 (accessed February 9, 2018). Remote desktop protocol: Basic connectivity and graphics remoting. Microsoft Corporation. URL https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr/5073f4ed-1e93-45e1-b039-6e30c385867c.

Microsoft Corporation, 2019. Windows 10 on arm. Microsoft Corporation. URL https://docs.microsoft.com/en-us/windows/arm.

Microsoft Developers Network, 2018 (accessed January 4, 2019). Wow64 implementation details. URL https://docs.microsoft.com/en-us/windows/desktop/winprog64/wow64-implementation-details.

Mihocka, D. and Shwartsman, S., 2008. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35, Beijing*, 32.

Morrissey, S., 2010. The economy of free and open source software in the preservation of digital artefacts. *Library Hi Tech*, 28 (2), 211–223.

Muir, A., 2004. Digital preservation: awareness, responsibility and rights issues. *Journal of Information Science*, 30 (1), 73–92.

Murphy, J., Hashim, N. H. and O'Connor, P., 2007. Take me back: validating the wayback machine. *Journal of Computer-Mediated Communication*, 13 (1), 60–75.

Noer, G. J., 1998. Cygwin32: A free win32 porting layer for unix applications. *2nd Usenix Windows NT Symposium*, 3–4.

NoMachine, 2017. Nomachine nx server. URL http://www.nomachine.com, [Online; accessed 20-January-2017].

NVIDIA, 2017. Nvidia gamestream: Play pc games on nvidia shield.

URL http://www.nvidia.co.uk/shield/games/gamestream, [Online; accessed 20-January-2017].

Oltmans, E., van Diessan, R. J. and van Wijngaarden, H., 2004. Preservation functionality in a digital archive. *Proceedings of the 2004 Joint ACM/IEEE Conference on Digital Libraries, 2004.*, 279–286.

Oracle, 2019 (accessed July 9, 2019). Java platform standard edition 8 documentation. URL https://docs.oracle.com/javase/8/docs/index.html.

Oracle Corporation, 2018 (accessed March 3, 2019). Websocket protocol overview. URL https://docs.oracle.com/cd/E55956_01/doc.11123/user_guide/content/general_websocket.html.

Oswal, S., Thanvi, R. K. and Thakur, N., 2016. Emulating everything. *International journal of Nursing Didactics*, 6 (01), 23–27.

Pedersen, B. J. and Perry, W. K., 1998. Method for supporting an extensible and dynamically bindable protocol stack in a distrubited process system. US Patent 5,826,027.

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

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

Penneman, N., Kudinskas, D., Rawsthorne, A., De Sutter, B. and De Bosschere, K., 2016. Evaluation of dynamic binary translation techniques for full system virtualisation on armv7-a. *Journal of Systems Architecture*, 65, 30–45.

Pike, R., Presotto, D., Thompson, K., Trickey, H. and Winterbottom, P., 1992. The use of name spaces in plan 9. *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, ACM, 1–5.

Pinchbeck, D., Anderson, D., Delve, J., Alemu, G., Ciuffreda, A. and Lange, A., 2009. Emulation as a strategy for the preservation of games: the keep project. *DiGRA 2009-Breaking New Ground: Innovation in Games, Play, Practice and Theory*.

Pokkunuri, B. P., 1989. Object oriented programming. *ACM Sigplan Notices*, 24 (11), 96–101.

Prescott, S., 2015 (accessed July 9, 2019). Retro city rampage for dos is out now, and windows 3.1 is next. URL `https://www.pcgamer.com/uk/retro-city-rampage-for-dos-is-out-now-and-windows-31-is-next`.

Presotto, D., Pike, R., Thompson, K. and Trickey, H., 1991. Plan 9, a distributed system. *Proc. of the Spring*, 43–50.

Racine, J., 2000. The cygwin tools: a gnu toolkit for windows. *Journal of Applied Econometrics*, 15 (3), 331–341.

Ray, E. and Schultz, E., 2009. Virtualization security. *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ACM, 42.

Rechert, K., Falcao, P. and Ensom, T., 2016. Introduction to an emulation-based preservation strategy for software-based artworks.

Rechert, K., von Suchodoletz, D. and Welte, R., 2010. Emulation based services in digital preservation. *Proceedings of the 10th annual joint conference on Digital libraries*, ACM, 365–368.

Reuben, J. S., 2007. A survey on virtual machine security. *Helsinki University of Technology*, 2 (36).

RFC6455, I., 2011. The websocket protocol.

Ribiere, A., 2008. Using virtualization to improve durability and portability of industrial applications. *2008 6th IEEE International Conference on Industrial Informatics*, 1545–1550.

Rosenblum, M., 2004. The reincarnation of virtual machines. *Queue*, 2 (5), 34.

Rosenthal, D. S., 2015. Emulation & virtualization as preservation strategies. *Andrew W. Mellon Foundation*.

Rothenberg, J., 1999. *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*. Council on Library & Information Resources.

Saipullah, K. M. b., Anuar, A., binti Ismail, N. A. and Soo, Y., 2012. Real-time video processing using native programming on android platform. *2012 IEEE 8th International Colloquium on Signal Processing and its Applications*, IEEE, 276–281.

Sanglard, F., 2012. Fabien sanglard's website. URL `http://fabiensanglard.net/quake3/network.php`, [Online; accessed 20-January-2017].

Sanglard, F., 2017. *Game Engine Black Book*. USA: CreateSpace Independent Publishing Platform, 1st edition.

Schäling, B., 2011. *The boost C++ libraries*. Boris Schäling.

Schaller, R. R., 1997. Moore's law: past, present and future. *IEEE spectrum*, 34 (6), 52–59.

Scott, G. F., 2014 (accessed July 9, 2019). Your bank machine probably runs windows xp, and that's about to become a problem. URL `https://www.canadianbusiness.com/technology-news/95-percent-of-atms-still-run-windows-xp`.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R. et al., 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27 (3), 18.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Dubey, P., Junkins, S., Lake, A., Cavin, R., Espasa, R., Grochowski, E. et al., 2009. Larrabee: A many-core x86 architecture for visual computing. *IEEE micro*, 29 (1), 10–21.

Shafer, J., Rixner, S. and Cox, A. L., 2010. The hadoop distributed filesystem: Balancing portability and performance. *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, IEEE, 122–133.

Sharpe, N. F. and Arewa, O. B., 2006. Is apple playing fair-navigating the ipod fairplay drm controversy. *Nw. J. Tech. & Intell. Prop.*, 5, 332.

Shea, R. and Liu, J., 2013. On gpu pass-through performance for cloud gaming: Experiments and analysis. *Proceedings of Annual Workshop on Network and Systems Support for Games*, IEEE Press, 1–6.

Shen, B.-Y., Chen, J.-Y., Hsu, W.-C. and Yang, W., 2012. Llbt: An llvm-based static binary translator. *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, New York, NY, USA: ACM, CASES '12, 51–60. URL `http://doi.acm.org/10.1145/2380403.2380419`.

Singhal, S. and Nguyen, B., 1998. The java factor. *Communications of the ACM*, 41 (6), 34–38.

Smed, J., Kaukoranta, T. and Hakonen, H., 2002. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20 (2), 87–97.

id Software, 2017. Fte quake world. URL `https://sourceforge.net/p/fteqw/code/HEAD/tree/trunk`, [Online; accessed 20-October-2017].

Stagner, A. R., 2013. *Unity multiplayer games*. Packt Publishing Ltd.

StrangeSoft, 2019 (accessed July 9, 2019). Openal soft. URL `https://kcat.strangesoft.net/openal.html`.

Swalwell, M., 2009. Towards the preservation of local computer game software: Challenges, strategies, reflections. *Convergence*, 15 (3), 263–279.

Sweeney, T., 2006. The next mainstream programming language: a game developer's perspective. *ACM SIGPLAN Notices*, 41 (1), 269–269.

Tebbe, M., 1998. Microsoft has embraced and extended java: Did this squeeze the life out? *InfoWorld*, 20 (11), 94–94.

Thelwall, M. and Vaughan, L., 2004. A fair history of the web? examining country balance in the internet archive. *Library & information science research*, 26 (2), 162–176.

Thompson, K., 1990. A new c compiler. *Proceedings of the Summer 1990 UKUUG Conf*, 41–51.

TIOBE Group, 2019 (accessed July 9, 2019). Tiobe index for ranking the popularity of programming languages. URL `https://www.tiobe.com/tiobe-index`.

Tso, F. P., White, D. R., Jouet, S., Singer, J. and Pezaros, D. P., 2013. The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. *Distributed*

Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on, IEEE, 108–112.

United States Government Accountability Office, 2017. Federal agencies need to address aging legacy systems. URL https://www.gao.gov/assets/680/677454.pdf.

Valgrind Memory Debugger, 2017. Valgrind memory debugger. URL http://valgrind.org, [Online; accessed 20-January-2017].

Valve Corporation, 2011 (accessed April 3, 2019). Half-life sdk. URL https://github.com/ValveSoftware/halflife.

Valve Corporation, 2018 (accessed April 3, 2018). Steam hardware software survey. URL https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.

Van Der Hoeven, J. and Van Wijngaarden, H., 2005. Modular emulation as a long-term preservation strategy for digital objects. *5th International Web Archiving Workshop (IWAW05)*.

Varanasi, P. and Heiser, G., 2011. Hardware-supported virtualization on arm. *Proceedings of the Second Asia-Pacific Workshop on Systems*, ACM, 11.

Von Suchodoletz, D., 2011. A future emulation and automation research agenda. *Automation in Digital Preservation*, 10291.

Von Suchodoletz, D. and Van der Hoeven, J., 2009. Emulation: From digital artefact to remotely rendered environments. *International Journal of Digital Curation*, 4 (3), 146–155.

Von Suchodoletz, D., Rechert, K., Schröder, J., van der Hoeven, J. and Bibliotheek, K., 2010. Seven steps for reliable emulation strategies solved problems and open issues. *iPRES 2010*, 373.

Walli, S. R., 1995. The posix family of standards. *StandardView*, 3 (1), 11–17. URL `http://doi.acm.org/10.1145/210308.210315`.

Walters, J. P., Younge, A. J., Kang, D. I., Yao, K. T., Kang, M., Crago, S. P. and Fox, G. C., 2014. Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications. *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, IEEE, 636–643.

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

Wang, C., Hu, S., Kim, H.-s., Nair, S. R., Breternitz, M., Ying, Z. and Wu, Y., 2007. Stardbt: an efficient multi-platform dynamic binary translation system. *Asia-Pacific Conference on Advances in Computer Systems Architecture*, Springer, 4–15.

Weber, M. and Quayle, P., 2018. Post-processing effects on mobile devices. *GPU Pro 360 Guide to Mobile Devices*, AK Peters/CRC Press, 57–72.

Weinberg, G. M. and Schulman, E. L., 1974. Goals and performance in computer programming. *Human factors*, 16 (1), 70–77.

Weiser, M., Demers, A. and Hauser, C., 1989. The portable common runtime approach to interoperability. *ACM SIGOPS Operating Systems Review*, ACM, volume 23, 114–122.

Welch, B., 1994. A comparison of three distributed file system architectures: Vnode, sprite, and plan 9. *Computing Systems*, 7 (2), 175–199.

Wessels, B., Finn, R. L., Linde, P., Mazzetti, P., Nativi, S., Riley, S., Smallwood, R., Taylor, M. J., Tsoukala, V., Wadhwa, K. et al., 2014. Issues in the development of open access to research data. *Prometheus*, 32 (1), 49–66.

Winget, M. A., 2011. Videogame preservation and massively multiplayer online role-playing games: A review of the literature. *Journal of the American Society for Information Science and Technology*, 62 (10), 1869–1883.

Wright, P., 2000. *Beginning GTK+/Gnome Programming*. Wrox Press Ltd.

Wu, D., Xue, Z. and He, J., 2014. icloudaccess: Cost-effective streaming of video games from the cloud with low latency. *IEEE Transactions on Circuits and Systems for Video Technology*, 24 (8), 1405–1416.

Xing, L., Bai, X., Li, T., Wang, X., Chen, K., Liao, X., Hu, S.-M. and Han, X., 2015. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 31–43.

Xylomenos, G. and Polyzos, G. C., 1999. Tcp and udp performance over a wireless lan. *INFOCOM '99. Eighteenth Annual Join Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, 439–446 vol.2.

Yegulalp, S., 2016 (accessed July 9, 2019). Openbsd 6.0 tightens security by losing linux compatibility. URL `https://www.infoworld.com/article/3099038/openbsd-60-tightens-security-by-losing-linux-compatibility.html`.

Yurkoski, C., Rau, L. and Ellis, B., 1998. Using inferno™ to execute java™ on small devices. *Languages, Compilers, and Tools for Embedded Systems*, Springer, 108–118.

Zabolitzky, J. G., 2002. Preserving software: Why and how. *Iterations: An Interdisciplinary Journal of Software History*, 1 (13), 1–8.

Zbyszyński, M., Grierson, M., Fedden, L. and Yee-King, M., 2017. Write once run anywhere revisited: machine learning and audio tools in the browser with c++ and emscripten.

Zheng, C. and Thompson, C., 2000. Pa-risc to ia-64: transparent execution, no recompilation. *Computer*, 33 (3), 47–52.