Intercepting OpenGL calls for rendering on 3D display

Examensarbete utfört i Bildkodning

Av

Joel de Vahl

LiTH-ISY-EX–05/3748–SE

December 21, 2005
Abstract

An OpenGL applications usually renders to a single frame. Multi-view or 3D displays on the other hand, needs more images representing different viewing directions on the same scene, but modifying a large number of applications would be unsuitable and problematic. However, intercepting and modifying these calls before they reach the GPU would dramatically decrease the amount of work needed to support a large number of applications on a new type of multi-view or 3D display. This thesis describes different ways on intercepting, enqueueing and replaying these calls to support rendering from different viewpoints. Intercepting with both an own implementation of opengl32.dll and an OpenGL driver is discussed, and enqueueing using classes, function pointers and enumeration of functions is tried. The different techniques are discussed quickly with the focus being a working implementation. This resulting in an fully blown OpenGL interceptor with the ability to enqueue and replay a frame multiple times while modifying parameters such as the projection matrix. This implementation uses an own implementation of opengl32.dll that is placed in the application directory to be loaded before the real one. Enqueueing is performed by enumerating all OpenGL calls, pushing this enumeration value and all call data to a list. Replaying is done by reading the same list and calling the function pointer associated with the enumeration value with the data in the list.
Contents

1 Introduction .............................................. 8
   1.1 Background ............................................. 8
   1.2 Purpose ................................................. 8
   1.3 Scope ................................................... 8
   1.4 Method ................................................. 9
   1.5 Overview .............................................. 9

2 3D displays and interaction .............................. 10
   2.1 Why is there a need for an interceptor? ............... 10
      2.1.1 Other possible applications ...................... 11
   2.2 3D displays ............................................. 11
      2.2.1 Simple stereo displays ............................ 11
      2.2.2 Autostereoscopic displays ....................... 11
      2.2.3 Holoform, more than stereo ...................... 12
      2.2.4 Other types of display ........................... 12
   2.3 Stereo rendering ....................................... 14
   2.4 Desired user settings .................................. 15
   2.5 Conclusions .......................................... 15

3 The Win32 DLL and OpenGL system ...................... 16
   3.1 The Win32 DLL ......................................... 16
   3.2 Implicit and explicit linking .......................... 17
   3.3 The DLL loading sequence .............................. 18
   3.4 The OpenGL chain ..................................... 18
   3.5 Conclusion ............................................. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Intercepting the calls</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Helping headers and structures</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Trojan DLL</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>OpenGL ICD</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>DLL injection</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusions</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Enqueueing and replaying</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Object wrapping</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Function pointers</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>Enumeration</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Conclusions</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Final design and implementation</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>Specification parser</td>
<td>31</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Calculation of parameter positions</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>Tables</td>
<td>32</td>
</tr>
<tr>
<td>6.3</td>
<td>The queue</td>
<td>33</td>
</tr>
<tr>
<td>6.4</td>
<td>Replayers</td>
<td>33</td>
</tr>
<tr>
<td>6.5</td>
<td>Extensions</td>
<td>34</td>
</tr>
<tr>
<td>6.6</td>
<td>User input</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>Call modifications</td>
<td>36</td>
</tr>
<tr>
<td>7.1</td>
<td>Classes</td>
<td>36</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Vector calls</td>
<td>36</td>
</tr>
<tr>
<td>7.2</td>
<td>Vertex arrays</td>
<td>37</td>
</tr>
<tr>
<td>7.3</td>
<td>Modified projection matrices and view ports</td>
<td>37</td>
</tr>
<tr>
<td>7.4</td>
<td>Other special cases</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Discussion</td>
<td>38</td>
</tr>
<tr>
<td>8.1</td>
<td>Implementation</td>
<td>38</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Intercepting</td>
<td>38</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Enqueueing</td>
<td>39</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Replaying</td>
<td>39</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Overall design</td>
<td>39</td>
</tr>
<tr>
<td>8.2</td>
<td>Application support</td>
<td>39</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Working applications</td>
<td>39</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Assumptions and limitations</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Example of an autostereoscopic display. .......................... 12
2.2 Example of a full 3D display. ................................. 13
2.3 Frustums for rendering of a stereo pair. ...................... 14

3.1 The DLL export table. ............................................ 16
3.2 The DLL import table. .......................................... 17
3.3 The Win32 OpenGL chain. ...................................... 19

4.1 Application initialization with Trojan DLL. ................. 23
4.2 Application initialization with ICD. ............................ 24

5.1 UML class diagram of the object wrapping method. ......... 27

6.1 Flow between the tables. ......................................... 33
6.2 Application flow during replaying of the queue. .......... 34
List of Tables

6.1 Example of a function call specification 32
List of Algorithms

1. Explicit loading of a DLL (pseudo code) ........................................ 17
2. Enqueuing using function pointers (pseudo code) .......................... 28
3. Enqueuing using enumeration (pseudo code) ............................... 30
Chapter 1

Introduction

1.1 Background

Regular OpenGL applications render a 3D scene to a single screen space, a frame. When rendering for a multi-view or 3D display, more views or angles of the same scene need to be produced. A lot of OpenGL drivers already support stereo rendering, but autostereoscopic displays need to rely on applications supporting their rendering algorithms. One such autostereoscopic display is the Scanning Slit Display being developed by Setred AB. This project was targeted primarily at supporting Setred’s display, but the work is applicable to all autostereoscopic displays.

The need to provide “per-application support” makes development and adoption of new display technology harder and more time consuming than it would be if 3D cards directly supported multi-view rendering in their drivers.

1.2 Purpose

The purpose of this thesis is to investigate different ways of intercepting OpenGL calls under Microsoft Windows to provide a rendering architecture for 3D displays, and implement a set of these techniques. The interceptor should provide support for basic OpenGL versions and work with OpenGL application that works in a fixed pipeline way, not rendering to textures or doing any “fancy” rendering techniques. It should also be as easy as possible for the user to control.

1.3 Scope

The scope of this report is to discuss three different intercepting, and enqueuing techniques. Selecting one of each type as the most suitable for the purpose and provide a detailed description on the final implementation of the selected techniques.
1.4 Method

This report is formed as an investigation and implementation of a chosen solution. The method used is divided into four parts: study of previous work, implementation and testing of the selected techniques, a final production and presentation of the work.

The study of previous work is intended to widen the perspective and to provide initial solution ideas on the problems presented. The implementation and testing is to provide enough testing on the selected techniques to select the best according to the criteria of the final implementation, and the final implementation is to test the selected techniques thoroughly and show which limitations they imply. The presentation stage is when the production of the report is taking place, aided by notes and documentation taken during the whole process.

1.5 Overview

This thesis is divided into four parts. The first part, Chapter 2, provides background information on 3D displays and the need for an interceptor and/or special driver support. The second part, Chapter 3, 4 and 5, presents the DLL and OpenGL systems in Microsoft Windows, interception methods and methods for enqueuing and replaying OpenGL calls. The third part, Chapter 6 and 7, presents the final design and the modifications made to make calls work when replayed. The last part, 8 and 9, discusses the limitations of the system and different ways to improve performance and compatibility.
Chapter 2

3D displays and interaction

This chapter aims at being an introduction to what a 3D display is and why an interceptor might be needed. Several types of displays are discussed and different rendering algorithms are outlined.

2.1 Why is there a need for an interceptor?

Today’s 3D accelerators (GPU:s) have support for rendering to a single frame buffer or to both a left and a right buffer, providing support for single display (regular monitor) or stereo display.

In a typical GPU implementation, a single vertex is transformed and the corresponding primitive is rasterized to pixels. A single triangle can in that way only be transformed to one view space to be rastered at one time. This is not in line with what 3D displays require, as a 3D “frame” can be composed of multiple views. To produce these multiple views the rendering commands must be iterated N times (where N is the total number of views) with different model view and/or projection matrices.

Generating the views could be the responsibility of either the driver or the application, but today’s drivers do not support that kind of rendering, and modifying each application to support each 3D displays different rendering algorithm would be too much work. Until GPU:s and drivers provide broader support for 3D displays, there is a need to intercept calls between the application and the driver. This extra layer will record the commands required to render a regular 2D frame and replay it as many times as needed from different angles to create a 3D frame suitable for the target display. Ideally, this would be done by the GPU or the driver, but since current hardware does not expose this functionality, alternative implementations are needed.

Some GPU drivers already support stereo rendering [1], but whether this is done by enqueuing and replaying the whole scene or by drawing each primitive in first the left, and then the right buffer is a closely guarded secret by the GPU vendors. Other interceptors exist, the most well known being Chromium [6], gDEBugger [16] and gIntercept [17], but these are mostly used as either
debuggers or information collectors for OpenGL applications or for networked renderers for large display walls (multiple projectors or screens tiled).

2.1.1 Other possible applications

The Stanford Chromium system, formerly WireGL, [5] is an interceptor system made for networked rendering. Chromium supports multiple render nodes connected to a provider. The nodes are in turn connected to projectors or other types of large screens to provide large video walls. This application has in turn inspired several new ideas on how to use the Chromium architecture to perform geometric transformations on the OpenGL command stream. One of them presents a way to slice scenes to provide better ways to visualize architectural 3D scenes [11].

As mentioned before, interceptors can be used for debugging applications by looking at the calls they make. Applying this to OpenGL, debugging an application can provide information on states, break on certain calls and track memory usage for textures and geometry. The commercial debugger gDEBugger [16] provides an interface for viewing all textures and display lists created by the application, along with breakpoints, logs and statistics.

2.2 3D displays

There are a lot of different 3D display systems available, ranging from under 100 sek to several million sek in price. The range covers everything from simple red/blue glasses to multi projector systems for full wall displays. This section is intended to shed some light on different types of 3D displays.

2.2.1 Simple stereo displays

A stereo display is a display that provides two separate images to the viewer, one for the left and one for the right eye, with the limitation that only those two images are rendered, independent of the number of viewers and their position. The simplest example of that kind of system is red and blue glasses where a red filter for a left eye and a blue for the right eye filters out the two different images from a regular screen or projector.

Other variations on this theme can provide full color for both eyes. One way is to polarize the light in orthogonal directions when passing out of the screen and use polarizing lenses in the glasses. Another is to use a simple LCD shutter for each eye and display the stereo pair sequentially, switching the eye that the image will reach. The last approach requires active glasses that have to be synchronised with the screen, usually by wire or IR.

2.2.2 Autostereoscopic displays

Unlike regular stereo, autostereoscopic displays [3] do not require the user to wear special glasses to create a feeling of depth. Instead, a set of lenses or
parallax barriers is used to transmit different images in different directions. An example of this is shown in Figure 2.1. The left and right images are interlaced and a barrier mask is placed to block the left image from reaching the right eye and vice versa.

This kind of screen puts severe constraints on the user. Depth is only experienced when the eyes are receiving the correct image, which requires the user to hold their head in a fixed position. To enable movement, several screens use a technique called head-tracking. A head-tracking system follows the user's head and/or eyes and updates the direction in which the images are correct to match. This puts a limit on how many simultaneous users the display can have.

2.2.3 Holoform, more than stereo

Holoform displays are another type of 3D display. This type of display requires more than two views of the scene (therefore “more than stereo”) but can provide depth experience for all viewers inside the display’s view cone. Figure 2.2 provides an example of this, where a set of images from the left and right are sent out, and the user can receive 3D experience inside the area where both left and right images are projected.

This type of screen does in general have worse depth quality since there are no “optimized” viewing positions, but provides another very important depth queue: motion parallax. To this point all other types of screen discussed have provided only two views, independent of the user position. Holoform displays allow users to move freely inside the view cone, providing the ability to look around objects to a certain degree. Examples of such displays are the display developed by Setred AB and the 3D TV display developed by Mitsubishi Research [8].

2.2.4 Other types of display

Apart from the types of screen mentioned above, there are a couple of completely different display techniques. One of them is to project an image set onto multiple stacked LCDs, one at a time. Depending on the number of such LCDs, the user can get a feeling of depth as different images have physically different depth.
Figure 2.2: Example of a full 3D display.
Figure 2.3: Frustums for rendering of a stereo pair.

Common to most of the displays described in this chapter is that they all require something more than just the standard D-SUB/DVI output from the graphic card, but the rendering algorithms might vary very much between them.

2.3 Stereo rendering

A stereo pair is two images of the same scene taken with a slight displacement of the camera position (see Figure 2.3). This is usually done from an existing view frustum by selecting an image projection plane with the distance D from the camera and a certain stereo separation. Assuming a right hand coordinate system, x axis to the right, camera positioned at the origin and looking along negative z (standard OpenGL setup), the camera is displaced both left and right rendering one image from each position.

As the camera is moved sideways, the intersections between each frustum and the image projection plane are also translated. To keep the physical image plane the same between these translations, the frustums are sheared. The above mentioned image shows how the frustum is sheared when moving the camera a distance to the left.
2.4 Desired user settings

A 3D display has many possible configuration options that are not needed for a regular screen. As OpenGL has no defined image projection plane and most rendering algorithms for 3D displays require one to be defined, the interceptron might need to have one defined. Different programs may require different coordinate systems, or at least a different scale of geometry, making the image projection plane a good value to be user-controlled in some fashion. As seen in the stereo rendering example in Figure 2.3, the camera is translated sideways. Different eye separation values will give different viewing results, which may also be something the user will want to tweak.

The nVidia stereo driver supports a multitude of controls [2, 1], the most notable being clamping the depth experience both at the front and at the back of the view frustum. This means that objects very near to and very far away from the viewer are projected according to the regular view frustum to the image projection plane. These distances are also something that could be user controlled.

2.5 Conclusions

As more and more types of 3D displays become available, different rendering algorithms must be supported, either by the application or by some other kind of layer. This makes an interceptron a feasible solution to the problem of providing per-application support for each possible rendering algorithm for multi view displays.
Chapter 3

The Win32 DLL and OpenGL system

This chapter provides an overview on how DLLs (Dynamically Linked Libraries) are handled in Microsoft Windows. It also aims to describe how OpenGL calls are handled in Windows and how an OpenGL call reaches the hardware after it is made from an application.

3.1 The Win32 DLL

On Windows, the PE (Win32 Portable Executable) defines a standard format for both DLLs and standard Win32 executables (EXE) both on disk and in memory [13, 14]. On disk, the DLL is composed by a set of headers and a section table describing what sections are available. A section can be either a code or data section, but where there is just one type of code section there can be many types of data sections.

<table>
<thead>
<tr>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>... Other fields ...</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Base</td>
</tr>
<tr>
<td>Number of Functions</td>
</tr>
<tr>
<td>Number of Names</td>
</tr>
<tr>
<td>Address of Functions</td>
</tr>
<tr>
<td>Address of Names</td>
</tr>
<tr>
<td>Address of Name Ordinals</td>
</tr>
</tbody>
</table>

Figure 3.1: The DLL export table.
Algorithm 1 Explicit loading of a DLL (pseudo code).

```c
function_pointer glBegin;
dll_handle handle;

handle = LoadLibrary("opengl32.dll");
glBegin = GetProcAddress(handle, "glBegin");

// Use acquired pointer
FreeLibrary(handle);
```

One important part for us is the export table (seen in Figure 3.1). This table contains information on the number of exported symbols, their address inside the DLL and their names. The address of these symbols can then be fetched using the different methods described in Section 3.2. Another important table is the import table (seen in Figure 3.2) which holds information on all DLLs to be loaded implicitly and which symbols to load from them.

### 3.2 Implicit and explicit linking

DLLs can be loaded in two different ways. Implicit (load time) and explicit (run time) linking.

Implicit linking happens when linking to the DLL and its .lib file at compile time, marking certain symbols in the code to be imported from that specific DLL. When the executable that links to the DLL is loaded (this can be either a DLL or a regular EXE), the import table is traversed and all DLLs are loaded into memory. An executable that is implicitly linked against another contains a table of symbols to import. This table contains the name and a dummy pointer to the function to be imported. Until the implicitly linked DLL is loaded, the symbols in this table cannot be used. When the DLL to be loaded is fully in memory, the linker will traverse the import table looking up each symbol from the DLL's export table by name and replacing the dummy pointer with the real address of the function.
Explicit linking, on the other hand, is completely driven by the developer. A library is loaded by calling the LoadLibrary function and released with the FreeLibrary function. When a DLL is loaded, the developer can call GetProcAddress to request a specific symbol. This might look as in Algorithm 1, where opengl32.dll is loaded and an exported symbol is fetched.

3.3 The DLL loading sequence

When loading a DLL, Windows looks in specific places for the requested file. Using the default behaviour on Windows XP, the following directories are checked in order [10]:

1. The directory from which the application loaded.
2. The current directory.
3. The system directory (c:\windows\system32).
4. The 16-bit system directory (c:\windows\system).
5. The Windows directory (c:\windows).
6. The directories in the PATH environment variable.

During loading of a DLL an optional entry point is called, if available. This function, called DllMain [10], has very restricted functionality and should only provide the simplest of setup. Calls like LoadLibrary have to be called at a later point.

3.4 The OpenGL chain

OpenGL support in Windows is provided by the opengl32.dll located in the system32 directory. This DLL exports symbols equal to OpenGL 1.1 with the addition of some WGL extensions for selecting rendering contexts and pixel formats. Most applications using OpenGL link to opengl32.lib which implies implicit loading of opengl32.dll but there are some applications, most notably the Quake series by id Software, that load the DLL explicitly and fetch the required symbols. WGL functions are platform specific (Microsoft Windows) extensions to OpenGL that provide support for selecting pixel format and other frame buffer related functionality.

When OpenGL support for windows was introduced, there were two ways for a hardware vendor to provide support. Either Providing a Mini Client Driver (MCD) or a Installable Client Driver (ICD). Both these drivers are included into the Windows 2D GDI driver package, provided by graphic card vendors to make the operating system utilize the cards capabilities fully. The MCD is a primitive rasterization path that exports a number of calls to Microsoft’s OpenGL implementation, to be used instead of software blitting. As all geometry had already been passed through the geometry transformation pipeline and
Microsoft's implementation could be used as an fall back for features not supported in hardware, the hardware vendor could concentrate on getting OpenGL support up and running while implementing the much more complicated ICD. The ICD is a full blown OpenGL implementation, where the vendor has to implement all OpenGL calls and provide software implementation for anything that the hardware is unable to provide. Since MCD support was removed from Windows 98, providing the much more complicated ICD is the only way to achieve accelerated OpenGL on Microsoft Windows.

The opengl32.dll in turn fetches the hardware vendor implementation from an Installable Client Driver (ICD) installed by the vendor. This ICD DLL is found by looking at the registry key “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\OpenGLDrivers”. Under this key the vendor provides information on which DLL to load as the ICD. The ICD then provides a table with all the OpenGL 1.1 calls. While the ICD is mostly undocumented in the Platform SDK provided by Microsoft, many alternative OpenGL implementations have reverse engineered this functionality. The open source OpenGL implementation Mesa 3D [15] has an ICD implementation which defines the function calls defined in Appendix A. If no ICD is registered, the OpenGL DLL can fall back onto a Mini Client Diver (MCD) or even software driver [12]. The complete OpenGL chain is shown in Figure 3.3.

The SetContext call is probably the most important call here. It returns a pointer to a table where the first element is a DWORD (an Microsoft specified data of size 32 bits) containing the total number of function pointers in the table. The order of these function pointers has also been reverse engineered by Mesa.

3.5 Conclusion

The easiest way to intercept OpenGL calls seems to be to override the DLL loading. This can be done at two points in the OpenGL chain, either replacing
the opengl32.dll or the ICD provided by the hardware vendor. Renaming the
original ICD or opengl32.dll and placing the intercepting ICD or Trojan in their
place is also feasible, but as that would collide with the consistency of the system
it was decided to intercept either by changing the ICD name in the registry or by
placing a Trojan opengl32.dll before the real one in the DLL loading sequence.
Chapter 4

Intercepting the calls

The first step to intercepting a function call is to somehow fetch the call from the sender before it reaches the receiver. Some logic can then process the function arguments and decide what to do. This chapter describes different interception strategies for OpenGL on Windows. The different strategies all come from a single base idea: tricking some part of the system to load a different DLL than usual.

4.1 Helping headers and structures

To make implementation and testing easier, a couple of headers were created to ease the creation and processing of the multitude of OpenGL calls that need to be implemented for an interceptor. These were later created using the specification parser described in Chapter 6. The headers consist of calls to a C macro called PROCESS_NAME that contains the following information, in order:

prefix The prefix that the function call has. D3V for ICD calls, wgl or gl.

The wgl calls are separated from regular gl calls since they are platform specific, and are not implemented by the ICD.

ret The return type of the function

name The function name, without the prefix.

args _tn All arguments with type, written as in the function declaration. Eg. (ArgType1 arg1, ArgType2 arg2, ... ArgTypeN argN).

args _n All arguments without type (just the name) written as in a function call. Eg. (arg1, arg2, ... argN).

num The ICD number of the function, only used for determining the order in the ICD table for OpenGL 1.0 and 1.1 functions.
These headers can then be processed by defining the macro, including all suitable headers and then undefining the macro. The reason for these headers was to be able to create some kind of "loop" where a specified operation could be done per call, with all the information about the call provided. For example creating a structure with typed function pointers to all OpenGL 1.1 calls.

Appendix B provides examples on how these headers can be used to perform a multitude of per call operations. Including enumeration, and loading of function pointers.

### 4.2 Trojan DLL

Intercepting with what I call a Trojan DLL is simply putting a new DLL with the same name as the original one (opengl32.dll in our case) in the DLL search path before the one in system32 (see Section 3.3). Note that Trojan in the context does not mean a provider of malicious code, but a OpenGL implementation that foils the application to think it is the real one. To make this work, the DLL must export the exact same symbols as the original and cannot load-time link to the real opengl32.dll since a DLL with that name is already loaded. Since Windows XP, all system files are under the control of Windows File Protection (WFP). The result is that no system files can be overwritten by “mistake” or purpose without turning the protection completely off. This dismisses the approach of renaming the original opengl32.dll while letting the Trojan take its place.

Figure 4.1 describes how the initialization sequence and the handling of OpenGL calls is done when an Trojan DLL Interceptor is loaded. Due to the nature of the Windows DLL loader (described in Chapter 3) , the Trojan DLL cannot load the real OpenGL DLL when inside DllMain; this has to be done when the first call is made to the exported symbols. During this initialization (marked as “Init Interceptor” in the image), the real OpenGL DLL and all exported symbols of that DLL is loaded and stored internally in the interceptor. All calls can now be intercepted and modified.

To implement this, a DLL with all OpenGL 1.1 and WGL functionality needs to be created. The list of symbols to export can be achieved by looking on the list of symbols the opengl32.dll provides (using the dumpbin utility in Microsoft’s Platform SDK). This DLL is then placed in each application’s directory and is in that way loaded as the default OpenGL DLL on systems with the default DLL search path. Once fully loaded some initialization code is run that loads the real DLL using a LoadLibrary call and logic that loads all the real OpenGL function pointers for internal storage in the Trojan. Incoming calls to the Trojan’s exported functions can then be processed before forwarding to the real DLL’s function pointers. The pointers can be loaded and used as shown in Appendix B.

### 4.3 OpenGL ICD

Creating an OpenGL ICD is a variation on the Trojan DLL. This method requires a DLL much like the Trojan, but instead of exporting all OpenGL func-
Figure 4.1: Application initialization with Trojan DLL.
Figure 4.2: Application initialization with ICD.

tions, only the ICD functions for requesting and managing contexts and pixel formats are exported. The real symbols can then be imported when the SetContext function is called, returning pointers to the internal GL functions in the DLL.

Figure 4.2 shows the difference between intercepting with the Trojan DLL and the ICD. Since the ICD interface specifies that DrvSetContext has to be called before calling any OpenGL functions, we can concentrate on loading the real ICD there. The handling of OpenGL calls is not shown, but doesn’t differ from the Trojan version other that modification of the call happens after the OpenGL DLL instead of before.

Building upon the Trojan DLL implementation described above, the DLL needs to implement all OpenGL calls and all ICD calls. Using the same approach as when intercepting wglSwapBuffers, DrvSetContext can be forced to return pointers to the internal OpenGL function calls inside our ICD. This is shown in Appendix B.
4.4 DLL injection

One method of interception, described in [9], is to hook onto DLL load messages for a specific thread and force the loading of our Trojan DLL. This approach is simply another way of connecting the Trojan to the application without the need to copy files. The downside is that applications that should be intercepted have to be launched by some kind of control application that initializes the injection code.

This method was not investigated further, but could be a viable alternative to copying the Trojan DLL to each application folder.

4.5 Conclusions

The Trojan DLL seems to be the most straightforward implementation of an interception mechanism, but might require some user interaction before an application can be intercepted. An ICD on the other hand could work completely without user interaction, but requires some thought on which function pointers are returned and builds upon undocumented functionality in Windows. Both these methods seem viable alternatives for an OpenGL interceptor implementation. DLL injection could also be possible, injecting either the Trojan or the ICD into the host process, but this was not tested due to time limitation and the dependence on another tool to launch the application to intercept.
Chapter 5

Enqueueing and replaying

When the structures for intercepting calls are in place, as described in Chapter 4, the calls can be altered and changed. The aim of this thesis is to enqueue and replay calls, which means that the DLL needs to have some internal structure to hold the calls. This chapter will discuss how this information can be stored and replayed. Three different approaches will be discussed together with their strengths and limitations.

A function call can be seen as a message containing a message ID (the function name) and some message data (the arguments). Some calls require information to be returned which means that the OpenGL commands need to be executed while enqueueing to keep the real driver in the correct state for return values on later calls. Chapter 4 described how the calls could be intercepted, deciding that all available methods require a function pointer to be called with the right convention and number of arguments. This means that the DLL will implement all OpenGL calls and that each call will have a function body that can contain any desirable logic that might be needed to enqueue that call, with the limitation that the return value needs to be the return value of the real OpenGL call in the context that it is called.

The most important features required of the enqueueing techniques are enqueue speed and the possibility to handle different types of OpenGL functions. One major problem is the possibility to handle vector functions, functions that have a pointer as argument and can fetch a fixed or arbitrary number of data from that pointer. A test interceptor was created for each of the techniques to evaluate functionality and performance.

5.1 Object wrapping

This method can be seen as an implementation of the Command design pattern [4], encapsulating a call (or command) into an object to be replayed at will. Each call can be represented as an object with private variables holding the data sent as parameters. Defining an abstract base class GICall which has the pure virtual method execute() and a list of pointer to this class makes looping
through the calls and executing them easy. The real call object then only needs to implement a constructor that takes all the arguments, store the arguments internally and implement execute() to call the real OpenGL call with the stored arguments.

For example, the call void glBegin(GLenum mode) can be represented as an object of type GICall_begin with an internal variable of type GLenum holding the same value as the argument, see Appendix C for a sample C++ implementation of one call.

This method can support different types of functions, for example vector functions, by letting the object constructor store all data in the vector and by providing a pointer to the stored vector when calling the real OpenGL function.

The main problem with this technique is that creating and deleting objects each frame has a huge impact on performance. Although this method is easy to implement, it did not perform well on large data sets (many function calls). This could be solved by pooling the objects inside a specified memory segment.

5.2 Function pointers

This method is a low level version of the object wrapping technique mentioned above. Instead of having a vector of object pointers, a vector of, for example, unsigned ints is used to store the pointer to the real OpenGL function, the number of unsigned ints the data takes and the data (see Appendix C for enqueue abstraction). As seen, the data is pushed in reverse order, so that replaying can be done without looping back wards through the queue. The call is replayed by fetching the function pointer, pushing the enqueued number of arguments to the stack and calling the function pointer using a low level CALL assembly instruction, see Algorithm 2 for a quick overview using pseudo code and Appendix C for a sample C++ implementation of one call. Note that the call in the example algorithm is a fictive OpenGL call used to illustrate how the technique works. Replaying can be done in a loop, not requiring special replay functions to be called (like the execute method in the object wrapped technique). Although this can look like a good way to handle replaying, it makes handling vector functions quite difficult.
Algorithm 2 Enqueueing using function pointers (pseudo code).

function glFunctionEnqueue(data[N]) do
    enqueue(address_of(glFunctionReplay))
    enqueue(N)
    for i = N..1 do
        enqueue(data[i])
    end
end

function Replay() do
    int offset = 0
    while offset < queue.size() do
        functionAddress = queue[offset]
        offset += 1
        N = queue[offset]
        for i = 1..N do
            system_stack.push(queue[offset + i])
        end
        CALL functionAddress
        offset += N
    end
end
Even though this technique is fast, requiring no extra function calls for replaying, it might not be feasible due to the hard coded function pointer and the fixed amount of data to be passed as arguments on the system stack.

5.3 Enumeration

Enumeration is a variation on both of the previously mentioned enqueuing techniques. As in the function pointer technique, all data is stored in a list of unsigned ints, but instead of storing the function pointer, a unique identifier for each call is stored, and the function arguments are stored after that.

Replaying can be done by having a table of function pointers indexed by the unique identifier pushed to the queue. To be able to handle different types of functions a replay function is defined. This function is written like unsigned int glFunction(unsigned int offset) and takes the offset of the first data in the queue as an argument. The function can then replay using any logic it wants to and returns the offset to one position past the last data, the position of the next function identifier. This way, the enqueuing function can put arbitrary data in the queue, assuming that the replay function can parse it and return the offset of the next function, making way for very specialized functions. The same enqueue abstraction as used in Section 5.2 can be used, but modified so that arguments with size larger than 4 bytes are not enqueued in swapped order. Algorithm 3 shows a pseudo code implementation using this method, using the same fictive OpenGL call as in the function pointer example. See Appendix C for a sample C++ implementation of one call using this method.

From a different point of view, this method is almost exactly the same as the object wrapping method described above. A constructor (enqueue function) allocates memory in the queue for the data to be stored and a execute function (replay function) reads the specified data from the queue and provides it to the real OpenGL function. The main difference is that instead of naming the variables, this technique works completely with the addresses inside the queue, not needing to allocate memory each time a new call is made.

5.4 Conclusions

All the above discussed enqueuing methods work, but with varying flexibility and performance. The enumeration method has enough flexibility (can handle both specialized enqueue and replay functions) and operates in a pre-allocated memory space which removes the most dominant performance bottleneck of the object wrapping method.
**Algorithm 3** Enqueueing using enumeration (pseudo code).

function glFunctionEnqueue(data[N]) do
    enqueue(enumeration(glFunction))

    for i = 1..N do
        enqueue(data[i])
    end

end

data[N]

function glFunctionReplay(offset){
    for i = 0..(N-1) do
        data[i + 1] = queue[offset + i]
    end

    realGl.glFunctionReplay(data)

    return offset + N
end

function Replay() do
    offset = 0

    while offset < queue.size() do
        functionEnumeration = queue[offset];
        functionPointer = get_replay_func(fenum);
        offset += 1

        offset = functionPointer(offset);
    end
end
Chapter 6

Final design and implementation

This chapter describes the final design and implementation created in this thesis, together with utilities to simplify adding new extensions and OpenGL versions. The design is heavily based on the Trojan and ICD examples described in Chapter 4, with a separate DLL for each implementation. These DLLs acts as a front end to the real interceptor, a statically linked library common to both front ends.

6.1 Specification parser

Writing each OpenGL function by hand is not feasible and time effective. To make the interceptor more maintainable and flexible, a small Ruby script was created. This script parses a number of extension files containing information about all OpenGL functions and generates functions for enqueuing and replaying each call. As a base, the Chromium specification file was used. This file was parsed into the internal data structure in the Ruby script and written to file. Parsing is done before compiling the interceptor, providing information about all possible supported calls.

Each function call is specified using the following keywords:

- **name**  Name of the function call, without prefix. E.g. Begin instead of glBegin.
  This line must be the first line of the specified call.

- **prefix** The prefix of the specified call. Either gl, wgl or Drv.

- **return** Return type of the function.

- **param** Specifies an argument to the function. Takes two whitespace separated arguments, the first is the name of the argument and the second is the type which can contain white spaces (e.g. const GLint *). This keyword can be used several times in a function specification, but the names has to be in the same order as in the OpenGL specification.
vector Specifies that a parameter is a vector parameter. Takes two arguments, the first is the name of the parameter (as specified by param) and the second is the number of elements in the vector.

category The version of OpenGL or the name of the extension where this call was introduced.

number The ICD number of the specified call. Set to -1 for all other calls.

type Function classification using the keys specified in Chapter 7.

```c
name Vertex3fv
prefix gl
return void
param v const GLfloat *
vector v 3
category 1.0
number 137
type
```

Table 6.1: Example of a function call specification.

### 6.1.1 Calculation of parameter positions

As shown in Section 5.3, replay functions must calculate sizes of arguments and add those together to find the position of the next argument in the queue. This is done in the parser, generating code that compiles to a constant when using an optimizing C compiler. When enqueuing, all data sizes is rounded up to the next multiple of 8. This is also true for vector calls, but here is the total number of bytes enqueued rounded up to the next multiple of 8.

### 6.2 Tables

The design of the interceptor revolves around a number of tables of function pointers. These tables control the behaviour of the interceptor at a given point, making different interception strategies possible.

**In** The table containing all entry points to the DLL and all OpenGL calls that are available using wglGetProcAddress. Each function in this table forwards the call to the receive table. This extra step is needed to provide the ability to change interception strategies for receiving calls.

**Receive** The table containing function pointers to whatever will happen when a call is received. This can usually be a pointer to the real OpenGL implementation or to an enqueue function. These pointers are interchangeable at runtime.
Replay The table containing function pointers to the replay functions currently in use. This table is used by the replay functions.

Debug The table containing function pointers to the debug functions. These functions use the same calling convention as the replay functions and dump all data enqueued in a frame to file.

Real The table containing function pointers to the real OpenGL implementation.

The application is only aware of the In table, or at least the content of it. This is where all the function calls will arrive to the interceptor. The function from the In table will then call the pointer associated with the same function in the receive table.

6.3 The queue

The enqueue strategy used in the interceptor is the same as the one described in Section 5.3. An array of unsigned ints contains a unique identifier and arbitrary data per function call. This data is enqueued on an individual basis by a function, and replayed by a replay function that mimics the enqueue when reading the data.

6.4 Replayers

A frame is considered done when wglSwapBuffers is called, and the Frame-Control function is called. This function processes messages, selecting which
replayer (or replay back end) to use. Supposing the frame was intercepted (en-queue using the enqueue functions) a replayer can use the queue to perform different rendering algorithms to create the necessary images.

![Diagram](image.png)

**Figure 6.2: Application flow during replaying of the queue.**

The interceptor implements a multitude of different renderers for both testing and rendering of images to 3D display.

**RedrawClear** Simple dummy redraw function that just clears the queue and returns.

**RedrawTiled** Redraw function that loops through the queue 16 times, rendering the scene to a 4x4 grid.

**RedrawRedBlueStereo** Renders red and blue images of the scene, with skewed frustums, creating an image suitable for red/blue stereo glasses.

**RedrawCoreRender** Renders scene for holeform 3D display.

### 6.5 Extensions

As opengl32.dll only provides functions for OpenGL 1.1, all other functions are provided to the application using the wglGetProcAddress function. A call to `glGetString(GL_EXTENSIONS)` will return a list of extensions supported by the driver. To force the application to only use the extensions the interceptor implements, an array of strings is created by the specification parser. This list is compared with the string from the driver, returning only the union of the two. This way the application will only recognize extensions implemented by both
the driver and the interceptor. When an application tries to fetch a function pointer, the interceptor intercepts the call and returns a function pointer to the In table. The real pointer is then stored in the real table instead.

New extensions are added by adding a new file with the extensions name (e.g. the extension named GL_EXT_framebuffer_objects becomes GL_EXT_framebuffer_objects.txt). This file is parsed using the specification parser and implementations for all extension functions are created.

6.6 User input

The input system is designed around Windows hooks to plant a system wide keyboard hook. This is due to full screen application (mainly games) taking full control of the system and not letting other applications be visible. It also provides a way to always fetch certain keyboard keys without being forced to use an graphical user interface. The keys are forwarded into the interceptor and is either processed at once or stored as a message that is read before replay. A method called FrameControl is called from all calls that swap OpenGL buffers which will trigger the control, queue clear and message processing before swapping the buffers.

Currently, there are controls for turning the interceptor on/off, switching to different replays, setting the distance of the image projection plane and dumping the queue to text file for debugging.
Chapter 7

Call modifications

Not all calls are suitable for enqueuing and replaying, and some replayers might need to substitute some calls with special calls during replay. This chapter describes the different classes the functions were partitioned into and how they act.

7.1 Classes

Calls were categorized to create different logic depending on the call type. The following list describes call modifiers that can be put in the type field in the function specification files. No modifiers means that the call should be intercepted.

Get  A function with the purpose of fetching data from OpenGL. Does not need to be queued since no real caller exists during replay and no state changes in OpenGL are made. Calls in this class are forwarded to the real table and do not have any enqueuer or replayers (the pointer in the enqueue table is equal to the pointer in the real table).

Special  This function has a special implementation, for example wglSwapBuffers, wglGetProcAdress and glGetString. Used for calls that need special care. The pointer in the enqueue table points to glNameSpecial.

None  No text in the type field. This call should be intercepted and enqueued.

7.1.1 Vector calls

A vector call is a call that takes a pointer to a memory segment and uses that during execution of the function. Since we are unaware of whether this data has changed during the application’s execution or not, we cannot assume it is correct when replaying. To solve this, all vector arguments have to be defined in the specification along with their length (4 GLfloats for example).
Enqueues and replayers are then generated by the specification parser. The enqueuer copies the memory segment to the queue and the replayer calls the real OpenGL call with a pointer to the enqueued data, skipping the size of the data when returning. For calls with a variable but limited amount of data, the upper limit is used.

### 7.2 Vertex arrays

One special class of functions is the vertex arrays functions. This class can be divided into pointer and draw calls, the pointer calls providing information on where to read the arrays and how (stride, type etc.) while the draw calls do the actual drawing. According to the OpenGL specification [7], the memory is read during the draw call, transferring geometry to the driver and hardware.

The memory location provided by the pointer calls is not of a known size until a draw call is made, making direct caching of this data inside the queue unfeasible. Instead a draw call could be exploded into regular immediate mode calls (glVertex, glColor etc.). This is described in the OpenGL specification and is what we used here.

Each pointer call is stored as an internal state to be used later. When a draw call arrives, the arrays are looped through in the specified order and the corresponding immediate mode calls are enqueued. When replaying, these calls are treated just like any regular calls.

### 7.3 Modified projection matrices and view ports

As applications can modify the projection matrix at any time during the frame, the replayers will need a way to ensure that its modified matrices are used even after the modification. To achieve this, the glMatrixMode call is modified to store if the application has switched to GL_PROJECTION and back, and glBegin is modified to add the replayers projection matrix before rendering any geometry.

Some replayers might want to change the view port to render to just a part of the screen, which means that glViewport has to be modified during replay to scale and translate the view port changes according to the replayers needs.

### 7.4 Other special cases

Some calls that can be considered as vector calls take a pointer to a memory location together with row and column stride and the number of data to read. Examples of this kind of function are the glMap calls.
Chapter 8

Discussion

The interception techniques and enqueuing methods described in this thesis should be applicable to most cases where intercepting of function calls. With very little overhead, the function is caught, processed, and forwarded to a suitable receiver or the real receiver. This chapter discusses how the implementation of the selected techniques performs and what might be improved.

8.1 Implementation

8.1.1 Intercepting

The interception techniques used where the Trojan DLL and the ICD. Both these methods work very well. The Trojan is probably the most straightforward implementation possible, just exporting and forwarding selected symbols, while the ICD is a bit more tricky due to more function pointers to keep track of. DLL injection was not investigated further, but might remain a feasible alternative to both implemented techniques.

As for a comparison between the implemented techniques, the Trojan is (as said above) by far the easiest way to implement an interceptor on Windows. The only information needed is the function declarations, which is given by the specification and the specification parser, and a way to handle various calls to get new OpenGL functions beyond 1.1 (wglGetProcAddress). Tables can then easily be set up to provide desired interception functionality. This technique can be extended to the ICD, but that requires some thought on how pointers are provided and handled between the ICD, the application and the OpenGL DLL. Performance wise, they are both on par since they both work on the same set on function pointers all through on different stages in the OpenGL chain.

As a DLL cannot load another DLL explicitly inside DllMain, both the ICD and the Trojan needs to have some mechanism to load its function pointers before the first call is processed. For now, this is done by checking the 'IsInitialized' flag each call, and calling Init() if this flag is false. As this only is true for the first call, a lot of unnecessary checks are done. The DllMain entry in the MSDN
8.1.2 Enqueueing

The queue implemented uses a std::vector<unsigned int>, which stores a 32bit chunk of data in each queue position. This size was used both since it is the registry size of 32bit CPUs and the size of a function pointer on that platform. The vector is initialized to a fixed number of elements, but is expanded each time it is filled. This way, it will reach a level where no new allocations is needed, and the queue will operate in a fixed memory space.

Raw enqueue performance is not the primary goal of this project since it is targeted at at least a couple of replays, where the fill rate and GPU bandwidth will be the bottlenecks. On the other hand, the enqueue performance hit can be measured using the Clear redrawer that just clears the queue. On a regular Quake2 map the performance drops to about 70% of the original performance using this replayer. This for a scene that takes up 81 KB in the queue.

8.1.3 Replaying

Aside from the obvious workarounds for projection matrices and view port fixes, the replayers are working fine. The replay overhead is almost none, since it consists of just a loop with two queue reads and a function call to the replay function in it. The replay function then does another call and fetches a total of N variables from the queue, where N is the number of argument the specified function takes.

8.1.4 Overall design

The most visible drawback of the interceptor is the performance decrease when using vertex arrays. Although it seems like a bottleneck for optimization, that task might be harder than it looks like at first glance (see the OpenGL specification [7]). Each element in the vertex array, color array, texture array etc. is exploded into a separate call, adding both call and queue header overhead.

8.2 Application support

8.2.1 Working applications

As the purpose of this thesis states, the interceptor should work with as many standard rendering applications as possible. In it current state, it works on a whole range of applications. From simple triangle test applications to full games like Quake 2 and 3, and applications like SketchUp.
8.2.2 Assumptions and limitations

The current implementation assumes that the application does not use shaders for multi pass and render to textures. It is also assumed that the model view and projection matrices are provided to the shader using the built in OpenGL matrices, and not using uniforms or other non standard constructs. The applications must also swap buffers using wglSwapBuffers, since that is the only function that will trigger redraw and a queue clear.

Another problem can occur when an application removes an object (texture, display list etc.). All creation and deletion of objects are ignored (not enqueued and replayed), which could cause problems if the application creates and deletes objects during a frame and not just at the start and end of the program. This problem could be avoided by having a remove list that stores all objects to be removed and removes them after all replays are done.
Chapter 9

Future work

This chapter contains thoughts on things that can be researched and implemented in the interceptor.

9.1 Networked rendering

As holoform displays can require more than 50 different viewing angles per frame to create an acceptable image, rendering performance might be a bottleneck on reaching real time performance. The interceptor could propagate the render queue to other computers on the network, the same way Chromium does, making them render the same frame from different angles in parallel.

9.2 Call aliases

As OpenGL evolved, many functions that were originally part of an extension were included into the specification. Many of these functions are the same functions with another name, taking the same parameters and have the same specified functionality. The specification could contain information about this, allowing the parser to skip generating enqueuer and replayers for the renamed calls, forwarding all calls to just one enqueuer. This would reduce total code size and might provide additional optimization for networked rendering since fewer calls would have to be enumerated which in turn would help compressing data.

9.3 Advanced function classes

The current implementation provides automatic parsing of vector class functions that have all data sequentially in memory. There are other types of functions that pass a dynamic amount of memory, and the specification parser could be extended to parse these. One example is the glMap* family of functions that take a row and column stride.
9.4 Multiple configurations

As the ICD is connected to all OpenGL applications on the system, some kind of selective interception should be implemented. The user should be able to select which applications to intercept, and maybe even set some configuration options depending on how that application behaves. This could probably be made by fetching information on the current process and reading configuration parameters from a config file.

9.5 Pluggable render back ends

Render back ends (Tiled, RedBlue and CoreRender) are implemented as separate functions inside the interceptor. To make the interceptor more flexible, these back ends should be implemented as separate DLLs using a defined interface.

9.6 Ports and other API:s

The techniques discussed and implemented on OpenGL for Windows could be adapted to work with both Direct3D on Windows and OpenGL on other platforms. Most operating systems provide some sort of dynamically linked libraries which can be intercepted using a trojan library. The specification parser could then be modified to generate interception code for nearly any API.

9.7 OpenGL extension

As more and more 3D displays becomes available on the market, it might not be suitable to for each display vendor provide an interceptor with the required rendering back end. To avoid a multitude of different interceptors, a standard interface could be defined, letting the vendor to hook onto the OpenGL driver and controlling rendering of the different view ports. All the interception and enqueuing could then be performed inside the GPU vendors drivers (which is probably already done in nVidiias stereo drivers).

To bypass some of the limitations of the interceptor implemented in this thesis, an OpenGL extension that controls the enqueuing could be exposed, allowing the application to somewhat control which parts of the rendering to be enqueued and which not.
Bibliography


Appendix A

ICD Interface

BOOL DrvCopyContext(HGLRC hGlrcSrc,
                     HGLRC hGlrcDst,
                     UINT mask)
HGLRC DrvCreateContext(HDC hDc)
BOOL DrvDeleteContext(HGLRC hGlrc)
HGLRC DrvCreateLayerContext(HDC hDc,
                             int iLayerPlane)
CDCallTable * DrvSetContext(HDC hDc,
                             HGLRC hGlrc,
                             void * callback)
void DrvReleaseContext(HGLRC hGlrc)
BOOL DrvShareLists(HGLRC hGlrc1,
                   HGLRC hGlrc2)
BOOL DrvDescribeLayerPlane(HDC hDc,
                           int iPixelFormat,
                           int iLayerPlane,
                           UINT nBytes,
                           LLPAYERPLANEDESCRIPTOR pIp)
int DrvSetLayerPaletteEntries(HDC hDc,
                              int iLayerPlane,
                              int iStart,
                              int cEntries,
                              CONST COLORREF * pcr)
int DrvGetLayerPaletteEntries(HDC hDc,
                              int iLayerPlane,
                              int iStart,
                              int cEntries,
                              COLORREF * pcr)
BOOL DrvRealizeLayerPalette(HDC hDc,
                        int iLayerPlane,
                        BOOL bRealize)
BOOL DrvSwapLayerBuffers(HDC hDc,
                     UINT fuPlanes)
int DrvDescribePixelFormat(HDC hDc,
int iPixelFormat,
UINT nBytes,
LPPIXELFORMATDESCRIPTOR ppfD)
PROC DrvGetProcAddress(LPCSTR lpstrProc)
int DrvSetPixelFormat(HDC hDC,
int iPixelFormat)
BOL DrvSwapBuffers(HDC hDC)
BOL DrvValidateVersion(DWORD version)
Appendix B

Intercepting

B.1 Using the helper headers

typedef struct GLImplementation{
  #define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
  ret (__stdcall* prefix##name##) args_tn;
  #include "gltable-1.0.h"
  #include "gltable-1.1.h"
  #include "wgltable.h"
  #undef PROCESS_NAME
} GLImplementation;

GLImplementation realGl;

void LoadPointers(){
  #define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
  *(*(FUNCTION*) &realGl.#prefix##name##) = \
    (FUNCTION)ProcAddress(glHandle, #prefix #name);
  #include "gltable-1.0.h"
  #include "gltable-1.1.h"
  #include "wgltable.h"
  #undef PROCESS_NAME
}

B.2 Forward table

GLImplementation receiveGl;

extern "C"{
  #define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
  ret (prefix##name## args_tn);
BOOL wglSwapBuffersInterceptor(HDC hDc) {
    // Do stuff
    return realG1.wglSwapBuffers(hDc);
}

void SetUpForwardTables() {
    #define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
    receiveG1.#name = realG1.#name;
    #include "gltable-1.0.h"
    #include "gltable-1.1.h"
    #include "wgltable.h"
    #undef PROCESS_NAME

    receiveG1.wglSwapBuffers = wglSwapBuffersInterceptor;
}

B.3 Loading ICD pointers

typedef struct ICDCallTable{
    DWORD numCalls;
    PROC table[336];
} ICDCallTable;

ICDCallTable* APIENTRY DrvSetContextSpecial(HDC hDc,
HGLRC hGlrc,
void *callback){
static ICDCallTable* icdTable = NULL;

if(!icdTable){
    // Get the GL calltable from the real ICD
    icdTable = realG1.DrvSetContext(hDc, hGlrc, callback);

    // Fetch all calls to our calltable
    #define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
    *((FUNCTION*) &realG1.#name) = \
    (FUNCTION) icdTable->table[#num];
}
#include "gltable-1.0.h"
#include "gltable-1.1.h"
#undef PROCESS_NAME

// Rewrite calls for interception
#define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
icdTable->table[##num##] = (PROC)##prefix##name##;
#include "gltable-1.0.h"
#include "gltable-1.1.h"
#undef PROCESS_NAME

SetUpForwardTables() SetupTables();
}

return icdTable;
}

void SetUpForwardTables(){
#define PROCESS_NAME(prefix, ret, name, args_tn, args_n, num) \
receiveGl.##prefix##name = realGl.##prefix##name##;
#include "gltable-1.0.h"
#include "gltable-1.1.h"
#include "icdtable.h"
#undef PROCESS_NAME

receiveGl.DrvSetContext = DrvSetContextIntercept;
}
Appendix C

Enqueueing and Replaying

C.1 Enqueue function

template<class T>
inline void enqueue(const T t){
  if(sizeof(T) == 1){
    const unsigned char* c =
      reinterpret_cast<const unsigned char*>(t);
    queue.push_back(static_cast<unsigned int>(c[0]));
  } else if(sizeof(T) == 2){
    const unsigned short int* i =
      reinterpret_cast<const unsigned short int*>(t);
    queue.push_back(static_cast<unsigned int>(i[0]));
  } else if(sizeof(T) == 4){
    const unsigned int* i =
      reinterpret_cast<const unsigned int*>(t);
    queue.push_back(i[0]);
  } else if(sizeof(T) == 8){
    const unsigned int* p =
      reinterpret_cast<const unsigned int*>(t);
    queue.push_back(p[1]);
    queue.push_back(p[0]);
  } else{
    assert(false);
  }
}

C.2 Object wrapping

class GlCall{
public:
   GICall() { }
   virtual ~GICall() { };
   virtual void execute() = 0;
};

class GICall_gIBegin : public GICall{
private:
   GLenum mode;
public:
   GICall_gIBegin(GLenum _mode) : mode(_mode) { };
   ~GICall_gIBegin() { };
   void execute(){
      realGl.gIBegin(mode);
   };
};

void gIBeginEnqueue(GLenum mode){
   queue.push_back(new GICall_gIBegin(mode));
   realGl.gIBegin(mode);
}

void Replay(){
   for(size_t i = 0; i < queue.size(); i++)
      queue[i]->execute();
}

C.3 Function pointers

void gIBeginEnqueue(GLenum mode){
   enqueue(reinterpret_cast<unsigned int>)(realGl.gIBegin));
   enqueue((static_cast<unsigned int>((sizeof(mode)<4?sizeof(mode)) + 0)) >> 2);
   enqueue(mode);
   return realGl.gIBegin(mode);
}

Replay(){
   unsigned int offset = 0;
   unsigned int faddr;
   unsigned int numArgs;
   unsigned int ac;
   while(offset < queue.size()){
      faddr = queue[offset];
      offset++;
      }
numArgs = queue[offset];
offset++;
ac = numArgs;

_asm{
    push eax;
push ebx;
mov ebx, esp;
}

while(ac > 0){
    arg = queue[offset];
    _asm{
        mov eax, arg;
push eax;
    }
    offset++;
    ac--;
}

_asm{
    mov eax, faddr;
call eax;
mov esp, ebx;
pop ebx;
pop eax;
}

} }

C.4 Enumerated

typedef unsigned int (*REPLAY_FUNCTION)(unsigned int);

void glBeginEnqueue(GLenum mode){
enqueue(G1FuncEnum_g1Begin);
enqueue(mode);
return realGl.glBegin(mode);
}

unsigned int glBeginReplay(unsigned int offset){
    realGl.glBegin( *reinterpret_cast<GLenum *>(&(queue[offset + 0 ])) );
    return offset + 0 + ((sizeof(GLenum)<4?4:sizeof(GLenum)) >> 2);
}

void Replay(){
unsigned int offset = 0;
unsigned int fenum;
unsigned int faddr;
while(offset < queue.size()){
    fenum = queue[offset];
    faddr = *replayPointers[fenum];
    offset++;
    offset = reinterpret_cast<REPLAY_FUNCTION>(faddr)(offset);
}
Appendix D

Modified call

```c
void APISENTRY glMap2fEnqueue(GLenum target,
    GLfloat u1,
    GLfloat u2,
    GLint ustride,
    GLint uorder,
    GLfloat v1,
    GLfloat v2,
    GLint vstride,
    GLint vorder,
    const GLfloat * points){
    unsigned int numElems = GetNumMap2Elements(target);
    enqueue(GLFuncEnum_glMap2f);
    enqueue(numElems);
    enqueue(target);
    enqueue(u1);
    enqueue(u2);
    enqueue(uorder);
    enqueue(v1);
    enqueue(v2);
    enqueue(vorder);
    for(unsigned int j = 0; j < static_cast<unsigned int>(vorder); j++)
        for(unsigned int i = 0; i < static_cast<unsigned int>(uorder); i++)
            for(unsigned int e = 0; e < numElems; e++)
                enqueue(points[e + i*ustride + j*vstride]);
    return realGl.glMap2f(target, u1, u2, ustride, uorder,
        v1, v2, vstride, vorder, points);
}

unsigned int glMap2fReplaySpecial(unsigned int offset){
    unsigned int numElems = queue[offset];
    offset++;
    GLint uorder = reinterpret_cast<GLint *>(&queue[ ... ]));
    GLint vorder = reinterpret_cast<GLint *>(&queue[ ... ]));
```
realGl.glMap2f(
    *reinterpret_cast<GLenum*>(&queue[ ... ]),
    *reinterpret_cast<GLfloat*>(&queue[ ... ]),
    *reinterpret_cast<GLfloat*>(&queue[ ... ]),
    *reinterpret_cast<GLuint*>(&queue[ ... ]),
    *reinterpret_cast<GLuint*>(&queue[ ... ]),
    *reinterpret_cast<GLuint*>(&queue[ ... ]),
    *reinterpret_cast<GLfloat*>(&queue[ ... ]),
    *reinterpret_cast<GLfloat*>(&queue[ ... ]),
    reinterpret_cast<const GLfloat*>(&queue[ ... ])
);
return ... ;
}
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.