Evaluation of Techniques for the Instrumentation and Extension of Proprietary OpenGL Applications

Maik Mory, Mario Pukall, Veit Köppen, Gunter Saake
Otto-von-Guericke University, P.O. Box 4120, D-39016 Magdeburg, Germany
(mory|pukall|vkoeppen|saake)@ovgu.de

ABSTRACT

Instrumentation of interfaces is a popular design pattern in engineering. Academic and industrial projects are already using instrumented OpenGL clients for various purposes. We perceive instrumentation of proprietary OpenGL applications as a basic technology to open up interactive three-dimensional graphics as a potent interoperability platform for heterogeneous simulation software in engineering. Hence, we describe and compare four instrumentation techniques on the MS Windows platform: relink library, dynamic library replacement, virtual display driver, and binary interception. We qualitatively evaluate them for four capabilities: to instrument proprietary simulation software; to instrument a subset only of the OpenGL interface; to instrument multiple interfaces simultaneously; and to chain intermediaries. The relink library technique is powerful, except that it cannot be used with proprietary simulation software. Dynamic library replacement and virtual display drivers potentially support all features, although some features are difficult to implement. The binary interception technique inherently supports all capabilities. We conclude with directions for future research.

Keywords

I.3.4 [Computer Graphics]: Graphics Utilities — Software support, Virtual device interfaces; D.2.12 [Software Engineering]: Interoperability — Distributed objects; J.2 [Physical Sciences and Engineering]: Engineering, Digital Engineering; I.6.8 [Simulation and Modeling]: Visual Simulation

1. INTRODUCTION

The digital revolution has changed the domain of engineering. Product values are determined increasingly by the software they contain. Even more, almost every engineering process is supported by computer simulations today. An important issue in digital working environments is interoperability between heterogeneous software components. We focus on interoperability between interactive simulations. The naive way to couple components is to implement one distinct adapter for each coupling wanted by the stakeholders. If we consider a project's size in number of components or products used, in the long term the naive approach results in the development of $O(n^2)$ adapters. Thus, the naive approach does not scale. Therefore, many projects define an interoperability platform. The interoperability platform captures a common concept, which is shared between components. Couplings that are wanted by the stakeholders are implemented by one adapter to the interoperability platform for each component. Because adapters can ideally be reused for different couplings, the interoperability platform approach results in the development of $O(n)$ adapters. Figure 1 illustrates the two strategies. The design of valuable interoperability platforms has become a critical concern in every bigger project.

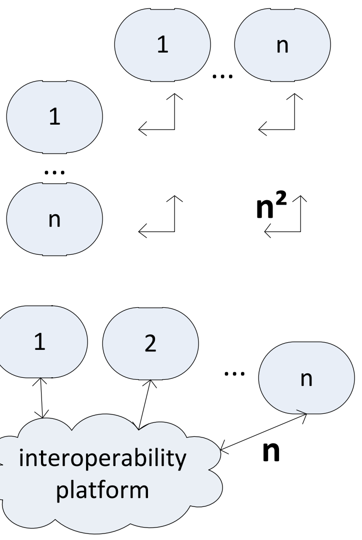


Figure 1: The curse of interoperability: Does a project that uses n applications implement $O(n^2)$ adapters; or is there a common concept, which serves as interoperability platform for $O(n)$ adapters?

The critical point when establishing an interoperability platform is to identify a common concept of candidate components. We want to improve interoperability of simulations in engineering. The question for an interoperability platform in industrial engineering software finds no answer since decades, e.g., [17, 26, 27]. Moreover, we do not want to establish yet another file based interoperability platform. We want to enable qualitative higher levels of interoperability through interactivity: components should exchange three-dimensional data during runtime, so that users are able to achieve information and gain knowledge in real time. Today, interactive visualization of simulation data is an omnipresent feature in simulation software. Often, simulation software uses OpenGL for interactive visualization of simulation data. We believe that the OpenGL industry standard serves very well as a common concept for interoperability of distributed virtual simulations [22].

As a first step towards the envisioned, OpenGL based interoperability platform we have to check, how to plug into the OpenGL pipeline. We focus on the pattern of instrumentation. Instrumentation by now primarily is known in performance analysis and debugging. But once the stream of function calls is tapped, it is only a matter of technical creativity which further applications can be implemented [6]. In the following sections, we describe a concrete application scenario and provide a distilled problem statement. Based upon component based system design, we work out an evaluation scheme with attestable properties. The properties are: capability to work with proprietary simulation software, to instrument interface subsets, to instrument multiple interfaces, and to chain instrumentation software. Then, we examine four instrumentation techniques for these properties. The techniques are relink library, dynamic library replacement, virtual display drivers, and binary interception. Finally, we conclude and discuss future directions of our research.

1.1 Application Scenario

Our scenario takes place in an engineering office environment. The project uses a heterogeneous setup: Microsoft Windows XP, Vista, and 7, which regarding our research question behave quite similar. The reference development platform for the operating systems is Microsoft Visual Studio.

Within the office environment, engineers use interactive simulation software (e.g., Matlab/Simulink[1]) that renders three-dimensional data based upon results of simulations. In the engineering domain simulation software mostly is proprietary or it is developed as a side project with scarce resources. Thus, we consider the simulation software to be proprietary. The visualization component communicates with the OpenGL application programming interface. Note, the visualization component is not necessarily a dominant or permanent element of the user interface; we only require it to be available. So far, the application scenario resembles the state of the art as common in every engineering office.

The engineers in our scenario now bear a challenge, when they want to modify the rendering behavior of the proprietary simulation software. We represent the modification of the simulation software, as we introduce notional analysis software, which profiles access on databases. For real-time presentation of the profile data, OpenGL should be used as pragmatic interoperability platform (see Figure 2). This means, that the analysis software visualizes aggregate data and embeds its visualization into the simulation's visualization. Therefore, the analysis software has to instrument the simulation's database interface, what we do not discuss in detail. Furthermore, it has to instrument the simulation's OpenGL interface. More precisely, within the OpenGL interface the analysis software instruments the `SwapBuffers` function [19], which semantically indicates, that the simulation has done its rendering work and the new window content should appear on the screen. The instrumented variant of the function renders the analysis software's graphics, which are a stream of OpenGL commands, into the simulation's graphics; finally, it invokes the original `SwapBuffers` function to put the merged graphical content onto the screen.

[1]The Mathworks Inc. provides the Matlab/Simulink simulation suite, which is very popular in the engineering domain. Detailed product descriptions are provided at the vendor's website (`http://www.mathworks.com/products`).

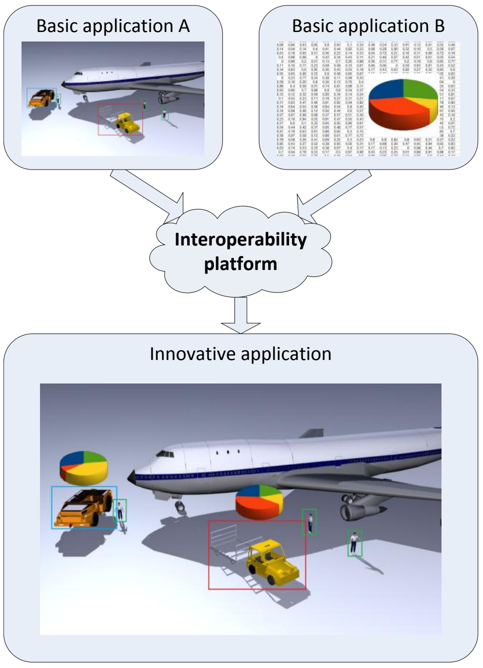


Figure 2: In our application scenario, we assume three-dimensional graphics as interoperability platform. This paper discusses instrumentation of proprietary OpenGL applications as basic technology.

1.2 Problem Statement

Based upon our application scenario with simulation software that should be instrumented, we ask the following questions:

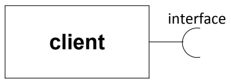
- Which techniques are available for instrumentation of OpenGL?
- Which techniques are applicable to proprietary simulation software?
- Which technique is preferable, if a narrow subset of the OpenGL interface should be instrumented?
- Which technique is preferable, if multiple independent interfaces should be instrumented to implement interleaved behavior?
- Which technique is preferable, if proprietary intermediaries should be chained?

We do not discuss the inner functionality of the instrumentation software. There is already a lot of research about stylized rendering (e.g., [21]), distributed rendering (e.g., [3, 12, 14, 22]), and GPU debugging (e.g., [9, 24]) through instrumented OpenGL. In contrast to the literature that we are aware of we focus on the mechanics, how instrumentation software is attached to simulation software.

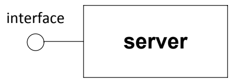
2. BACKGROUND

This section introduces terms and concepts, which are necessary to evaluate techniques for the instrumentation of proprietary OpenGL applications. Therefore, we declare a component and interface oriented notation together with an

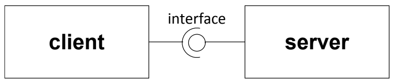
(a) A component.



(b) A client requires an interface.



(c) A server provides an interface.



(d) A client uses a server.

Figure 3: Definition of a simple component diagram notation based on UML 2

interoperability hierarchy. Building on that terminology, we discuss the instrumentation pattern and which properties are necessary, to cover our application scenario. We describe relevant implementation details of OpenGL on the MS Windows platform. Finally, we introduce four instrumentation techniques, which we evaluate in Section 3.

2.1 Modular Systems and Interoperability

Engineers describe system architectures in terms of components and interfaces. Depending on the engineer's domain and preferred method, what we call a component may be called an object, device, service, module, or other too. We focus on the domain of computer science. In our figures, we use a simplified dialect of UML 2's component diagram notation [11], which we present in Figure 3. A component is a definable software artifact. A server component provides an interface. A client component uses an interface. If a client's required interface and a server's provided interface are compatible, they can be connected. Then, the client uses the server.

Connections of components are differentiated between tight couplings and loose couplings. Tight coupling exploits dependencies and relations between the server and the client; the connected components are not supposed to be exchanged. A loose coupling minimizes dependencies and relations between client and server to a well-defined specification of the interface; loosely coupled components tend to be exchangeable. In real life, couplings are not clearly the one or the other. Rather, real couplings distribute in a continuum with ideal loose coupling on one end and with ideal tight coupling on the other end. The distinction is made, whether an instance is more the one or the other.

Loose couplings are the subject of interoperability. Interoperability is a field of active research. The most exhaustive, recent survey we know of was done by Manso et al. [17]. They declare seven levels of interoperability: technical, syntactic, semantic, pragmatic, dynamic, conceptual, and organizational. In our context it is sufficient to stick with a three level hierarchy of interoperability [16], which we briefly introduce as follows:

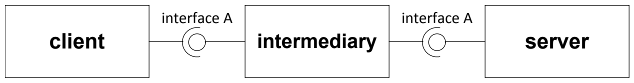


Figure 4: An instrumented connector has an intermediary component in the connection between client and server.

Syntactic interoperability is data exchange with a common set of symbols to which a formal grammar applies.

Semantic interoperability is information exchange with a shared, common vocabulary for interpretation of the syntactic terms.

Pragmatic interoperability is contextual exploitation of applications and services through shared knowledge.

An interface definition covers a subset of the interoperability hierarchy. Most interface definitions in computer science, especially application programming interfaces' documentations focus on syntactic and semantic interoperability. Software developers usually delegate technical interoperability to electrical engineers, who design computer chips and network links. The upper half of the interoperability hierarchy usually is in the responsibility of software project's stakeholders. The specification of OpenGL[2] defines syntax through function signatures together with a finite state machine and it defines semantics through human readable documentation for modules and functions.

2.2 Instrumented Interfaces

Instrumented connectors are a popular design pattern in engineering. For example in the domain of computer science, the Decorator, Proxy, and Composite design patterns [7] are object oriented interpretations of the instrumentation pattern. Given a connected client and server, a component commonly known as intermediary is inserted into the connector. The intermediary is the client's new server and the server's new client. Thus, an intermediary provides and requires the interface that connects the client and the server. Figure 4 depicts an instrumented connector. Instrumentation adds functionality to a given system. Thus, it can serve many purposes: Most instrumented connectors inspect the connector while the system is running; often, just like in our application scenario, connectors are instrumented to modify the behavior of the system [8].

Based upon the property that ideal loosely coupled components are interchangeable, Grechanik et al. [10] state, that an instrumented connector should be non-invasive and idempotent. They define an instrumented connector non-invasive, if it is undetectable in a connector's implementation. In today's complex world of software products with multi project source code bases, embedded scripting languages, and the "code is data" paradigm, Grechanik's definition is hard to decide for any real example. Therefore, we propose a pragmatic definition: An instrumented connector is non-invasive, if it works with proprietary client and server. Note that a free simulation model and script interpreted by proprietary infrastructure still builds up a proprietary software

[2]The Khronos Group provides the specification of OpenGL in a set of documents on their website (`http://www.opengl.org/documentation/specs/`).

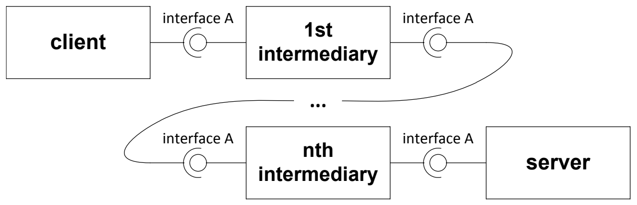


Figure 5: If intermediaries are idempotent, then they can be chained.

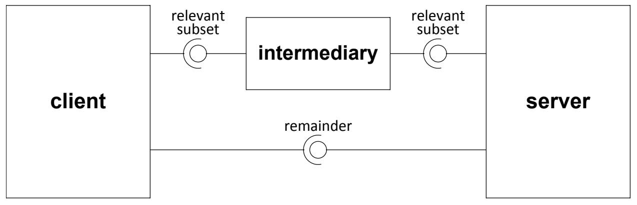


Figure 6: When a subset of the interface is instrumented, the remainder of the interface should be kept available to preserve interoperability between client and server.

product. If intermediaries can be chained because they have equal import and export interfaces and because they are non-invasive, then they are called idempotent [10]. Figure 5 shows chained intermediaries. We stick with Grechanik's definition of idempotency together with our adapted understanding of non-invasiveness. In our opinion, idempotency is the definite criterion whether a component matches the instrumentation pattern.

Interfaces can be composed of interfaces and conversely be part of other interfaces. OpenGL's formal interface definition indicates several decompositions. For example the notorious OpenGL extensions are self-contained subsets of the interface. In our application example in Section 1.1, we outline the instrumentation of a minimal interface subset, which consists of one function. Ideally, an instrumentation technique should be able to put an intermediary into a relevant subset of the interface while it keeps the remainder of the interface available to preserve interoperability between client and server. Figure 6 illustrates our requirement for interface subset instrumentation.

An intermediary not necessarily restricts to a singular, well defined interface. Sometimes, the intermediary should implement behavior that is interleaved between multiple interfaces. In our application scenario in Section 1.1, we want to interleave the behavior of two interfaces: the database application programming interface and the OpenGL application programming interface. If one intermediary instruments more than one connector, we call this multiple interface instrumentation. Figure 7 illustrates an intermediary, which implements cross-cutting functionality on multiple interfaces.

2.3 OpenGL on Microsoft Windows

OpenGL is part of MS Windows' software development environment. The OpenGL application programming interface is provided for all languages of Microsoft Visual Studio (e.g., C/C++, C#, Java). Because .NET and other script languages are built on top of the C/C++ toolchain, we constrain our discussion to the latter.

Developers and compilers access the syntactical interface

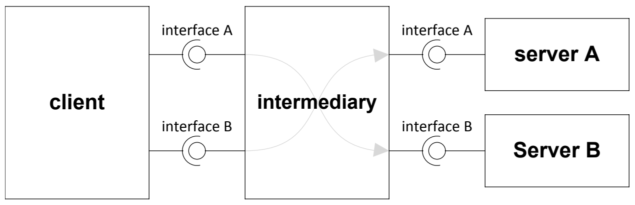


Figure 7: If the behavior of multiple connectors should be interleaved, they have to be instrumented with one intermediary.

definition of OpenGL through the `gl/gl.h` header file. The source code of the simulation software contains requests of OpenGL functions based upon the function definitions from the header file. At the time of this writing the OpenGL specification[3] defines 2269 functions. The windowing system interface to the Microsoft Windows implementation of OpenGL (WGL[4]) adds 131 function definitions. Thus, an OpenGL application on Windows has access to a repository of 2400 functions. Microsoft Windows' OpenGL implementation is determined to be compatible with OpenGL version 1.1. Hence, the 2400 functions divide into three sets: 357 core functions, 1671 extension functions, and 372 alias functions. Core functions are directly accessible C/C++ routines. For extension functions the signature only is declared. A client has to request the procedure's entry point via the `wglGetProcAddress` function [20] before it can invoke the extension function. The server is not obligated to provide all extension functions; it might return the `NULL` value to indicate, that a particular extension function is not supported. Alias functions are identifiers that actually refer to one of the core or extension functions.

Windows establishes several abstraction layers between OpenGL client software and OpenGL server implementations (see Figure 8). During the compilation of the software, the compiler translates source code symbols to library symbols. In the linker step, the library symbols are resolved, so that the final binary executable image contains binary function code from `opengl32.lib` together with the simulation's binary code, which invokes the OpenGL functions. The `opengl32.lib` is a stub that redirects core function calls to the functions exported by the `opengl32.dll` dynamic library's symbol table during runtime. Extension functions are passed through the `wglGetProcAddress` function as it is stubbed by the static library likewise. While the linked functions of the static library are integral part of the simulation software, the dynamic library is searched, loaded and linked by the `LoadLibrary` function [18] during the initialization phase of the simulation's runtime. The `opengl32.dll` dispatches the application's function calls to a matching display driver. The concrete workings of this mechanism are an implementation detail of Windows' WGL facility. In our application scenario it is sufficient to assume that there is one screen that is driven by one graphics hardware with one display driver.

[3]We used the OpenGL specification as published on `http://www.opengl.org/registry/api/gl.spec` revision 12819 with the timestamp 2010-11-03 19:02:01.

[4]We used the WGL specification as published on `http://www.opengl.org/registry/api/wgl.spec` revision 10796 with the timestamp 2010-03-19 17:31:10 and `http://www.opengl.org/registry/api/wglext.spec` revision 12183 with the timestamp 2010-08-06 02:53:05.

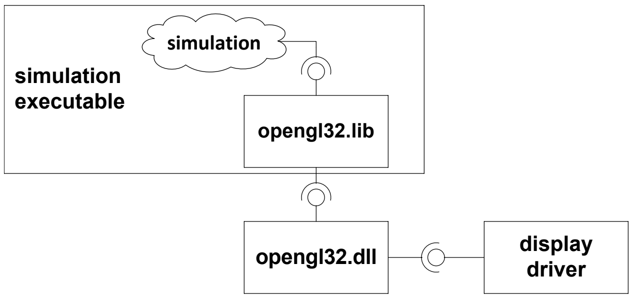


Figure 8: The OpenGL pipeline from a client, which requests visualization routines, to the server, which actually performs the rendering work, has several levels of indirection on Microsoft Windows.

2.4 Instrumentation Techniques

In this section, we describe four techniques that plug an intermediary into a simulation application: relink library, dynamic library replacement, virtual display drivers, and binary interception. We decrease redundancy in the descriptions as we anticipate an implementation detail that is common to each of the four instrumentation techniques. Extension functions are accessed through the `wglGetProcAddress` function. Therefore, it is sufficient to instrument extension functions by on-the-fly substitution of function addresses. One instruments the `wglGetProcAddress` function with an implementation, which returns intermediary's extension functions to the client and stores server's extension functions for later delegation. The `wglGetProcAddress` function is a core function. Thus, in the following it is sufficient to focus on instrumentation of core functions.

An intermediary that uses the **relink library** technique (e.g., [5]) is injected during compile time of the client software. In its simplest form, which gave the technique its name, a developer reconfigures the linker not to link the original `opengl32.lib` but to link the intermediary static library instead. The intermediary provides all symbols that the original library provided. With this simple approach, the intermediary static library cannot directly utilize the original static library, because otherwise there would be a name clash in the linker step. Today, one avoids the name clash issue by a modification of the relink library technique. A developer reconfigures the compiler not to use the original header files but to use intermediary's header files. The intermediary's header files define the OpenGL API for the client's source code but link to distinct symbols for the intermediary's static library instead. The implementation of the intermediary uses original header files, which link to the original static library. Throughout this paper, the shorthand term "relink library" refers to the latter, improved relink library technique.

An intermediary that uses the **dynamic library replacement** technique (e.g., [9, 15, 21, 22, 24], esp. [12]) is injected when the client software attempts to load the `opengl32.dll` dynamic library. The execution environment of the client process is configured, so that the call `LoadLibrary("opengl32")` does not load the original dynamic library but loads the intermediary dynamic library. For this, either the operating system's dynamic library is replaced on the file system; or the intermediary's dynamic library is in a place of `LoadLibrary`'s search sequence [18] prior to the original one. The intermediary library has to keep a reference to the location of the original dynamic library. If the library is replaced on the file system, the intermediary dynamic library loads the original library with the qualified path to the original library's backup. If the library has been put into `LoadLibrary`'s search sequence, the intermediary library loads the original library with the qualified path to the original dynamic library in the operating system installation. The intermediary then uses the symbols from the original dynamic library, which was loaded by qualified path, to invoke the original server.

A **virtual display driver** (e.g., [1, 2, 23], esp. [25]) is injected through the display driver framework of the operating system. The operating system maps displays to device drivers. The virtual display driver mimics a display. Client software that is executed on a virtual display is associated with the underlying virtual display driver. The virtual display driver invokes the original server through a proxy process, which is spawned on the display that is associated with the original display driver.

The technique of **binary interception** was designed by Hunt and Brubacher [13] with the intention to instrument and extend proprietary software. The intermediary manipulates the software's binary image during runtime. For each function that should be instrumented, the intermediary installs a detour. The installation procedure for a function overwrites the first bytes of the server's function with bytecode, which detours the execution path to the intermediary's function. When the client invokes an intercepted server's function, the overlaid detouring code is executed instead of the original code. In effect, the client invokes the intermediary's function. The installation procedure produces a so-called trampoline, which keeps the original server's function available. The trampoline contains a backup of the server function's bytecode that was overwritten during installation of the detour and additional bytecode that repatriates the execution path to the unmodified remainder of the server function's bytecode. In effect, invocations of the trampoline delegate calls to the server.

3. EVALUATION

In this section, we evaluate the four instrumentation techniques, which we introduced in Section 2.4, with the properties that we developed in Section 2.2. Because the instrumentation of extension functions is common to all of the four techniques, we anticipate its evaluation. We assume that the `wglGetProcAddress` core function is instrumented. If a subset of the extension functions should be instrumented, the intermediary substitutes the procedure addresses in the subset; in the remainder, it passes through the original addresses. The instrumentation of extension functions is not related to the instrumentation of interfaces other than OpenGL. Therefore, it has no effect for a technique's ability to instrument multiple interfaces. If a chained intermediary uses an instrumented `wglGetProcAddress` function, it substitutes and uses the procedure addresses to instrumented extension functions, which are provided by the preceding intermediary. Thus, instrumentation of extension functions through procedure address substitution does not break idempotency. Hence, without loss of generality, we focus on the instrumentation of core functions, when we evaluate the four techniques in the following.

The **relink library** technique requires access to the client's code base. Thus, it is not possible to instrument proprietary clients with this technique. For the instrumentation of an

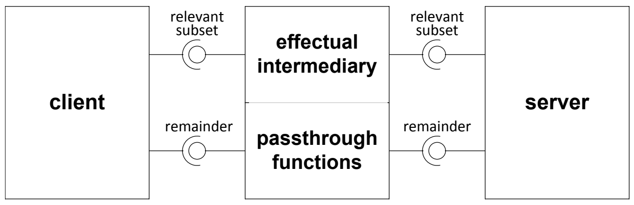


Figure 9: Dynamic library replacement and virtual display drivers do not inherently support instrumentation of subset interfaces. They need passthrough functions as workaround.

interface subset, the injected header file redefines the relevant symbols only. Functions that should not be instrumented are duplicated from the original OpenGL header file, which puts the original server's core functions into effect. Thus, it is easy to instrument a subset. If multiple injected header files point their symbols to the same software artifact, they instrument multiple interfaces with one monolithic intermediary. Thus, the relink library technique allows for instrumentation of multiple interfaces. An intermediary's implementation generally uses the original OpenGL header file to delegate calls to the original server. If one intermediary's header file is injected into another intermediary's implementation, then the latter one delegates its calls to the first one. Thus, the relink library technique is idempotent.

Dynamic library replacement does not require access to the client's code base. Hence, it is suited for proprietary simulation software. The intermediary has to provide all functions of the OpenGL API to preserve interoperability with the client. For core functions that should not be instrumented, the intermediary has to export symbols for passthrough functions. Figure 9 illustrates how passthrough functions are used to pad the interface. Thus, instrumentation of an interface's subset is possible. The interface provided by a dynamic library is well defined. Therefore, instrumentation of multiple interfaces with dynamic library replacement requires injecting distinct dynamic libraries for each of the multiple interfaces. These separate dynamic libraries, which are loaded into the process, frame a distributed system as depicted in Figure 10, which produces its own additional interoperability issues. Thus, we notice that instrumentation of multiple interfaces through dynamic library replacement is possible but elaborate and error prone. The chaining of proprietary intermediaries is limited by the way, how the library loader works. First, the intermediary only can be applied if the client does not reference the OpenGL dynamic library through a fully qualified path. Then, the intermediary references the original server through its fully qualified path. If we apply more than one intermediary, then one of them comes into effect first. This prioritized intermediary directly loads the original server through the fully qualified path, which effectively skips the other intermediaries. Figure 11 illustrates how one intermediary disables other intermediaries through a short circuit to the original server. Thus, dynamic library replacement is not idempotent.

Virtual display drivers do not require access to the client's code base. Thus, virtual display drivers work with proprietary OpenGL clients and are non-invasive. If a virtual display driver instruments a subset of the interface, it still has to provide all other functions of the interface, because otherwise interoperability with the client would break. Therefore,

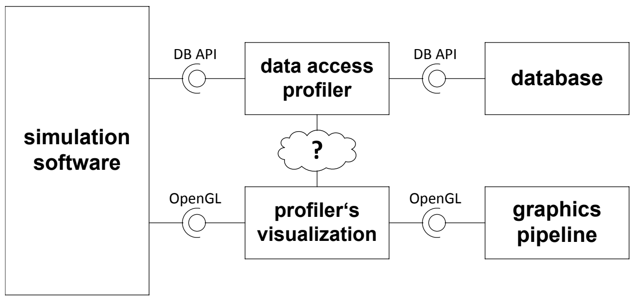


Figure 10: Dynamic library replacement and virtual display drivers do not inherently support instrumentation of multiple interfaces. They need distributed intermediaries as workaround, which produces additional interoperability issues.

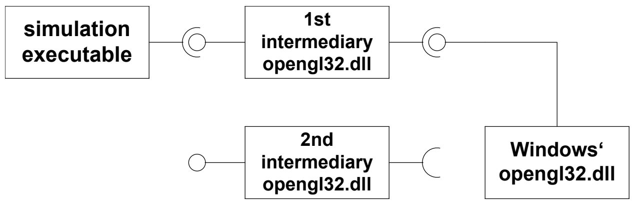


Figure 11: If dynamic library replacement is not done carefully, chained intermediaries break with a short circuit to the original dynamic library.

it has to implement passthrough functions for the remainder of the core interface as depicted in Figure 9. Thus, interface subset instrumentation is not inherently supported but possible. Whether virtual or not, a display driver provides display functionality. If multiple interfaces should be instrumented, each function that is not related to display drivers has to be handled by another intermediary than the virtual display driver. Comparable to multiple interface instrumentation with dynamic library replacement, this introduces a distributed system of intermediaries, which produces additional interoperability issues (see Figure 10). Thus, virtual display drivers are not suited for multiple interface instrumentation. An intermediary virtual display driver spawns a proxy client application on another server's display. Whether the server is the original server or a chained intermediary virtual display driver makes no difference. Thus, virtual display drivers are idempotent.

The **binary interception** technique does not require access to the client's code base. Therefore, it can be used with proprietary simulation software. The instrumentation of an interface's subset is easy. One installs detours for every function that should be instrumented. The other functions, which should not be instrumented, remain untouched. A detour can be installed for every function that is provided by any server. One monolithic intermediary can install detours into an arbitrary interface inside the running process. Thus, the instrumentation of multiple interfaces is easy, too. In Section 2.4 we describe instrumentation through binary interception for the first intermediary. We show idempotency, as we assume, that there are already detours installed. The installation of yet another detour exactly overwrites the old detouring code with new detouring code. The new trampoline contains the detouring code of the previous intermediary

technique	proprietary client	interface subset	multiple interfaces	idempotency
Relink Library	−−	++	++	++
Dynamic Library Replacement	++	+	−	−
Virtual Display Driver	++	+	−	++
Binary Interception	++	++	++	++

Table 1: The instrumentation techniques' strengths and weaknesses in a shorthand comparison. The symbols are used as follows: (++) the technique inherently supports the feature; (+) the technique needs a workaround to support the feature; (−) the technique basically supports the feature with severe restrictions; (−−) the feature is inherently not supported by the technique.

in its bytecode backup. In effect, a client's call gets detoured to the last installed intermediary. An intermediary's trampoline function invokes the detouring code of the intermediary that was installed previously. This pattern continues until the execution path arrives at the first installed intermediary. The first installed intermediary's trampoline contains the original server's function. Thus, binary interception is idempotent.

We recapitulate our results in Table 1. The relink library technique does not work with proprietary simulation software as OpenGL client. With dynamic library replacement it is hard to instrument multiple interfaces. Moreover, it is nearly impossible to achieve idempotency for proprietary intermediaries with dynamic library replacement. Virtual display drivers do not support instrumentation of interfaces that are not related to display functionality. Binary interception fulfills all declared requirements.

4. CONCLUSION AND OUTLOOK

In this paper, we define an application scenario, which we believe resembles the near future of pragmatically interoperable, distributed interactive simulations. Instrumentation of the OpenGL API is a basic technology in this application scenario. We identify requirements, which an instrumentation technique should fulfill to be useful in a proprietary software environment and to scale for various purposes. We describe four techniques for instrumentation of OpenGL: relink library, dynamic library replacement, virtual device drivers and binary interception. We argue that binary interception is the best available technology.

Today, binary interception is primarily used in malicious software and for game cheating. The experiments and products for instrumented OpenGL that we know of use the other three techniques that we outlined. The shortcomings of these three techniques, as documented in Section 3, restrict existing products to niches. We argue that binary interception is useful for the instrumentation of OpenGL. The next step is to provide an experiment, where binary interception is used for a complete computer graphics application with useful behavior in the intermediary. The following paragraphs outline our preliminary plans in doing so.

Based upon the research presented in this paper, we will implement a prototypical software using the Generative Programming Paradigm [4]. The OpenGL API's formal specification[5] will be input to a generative build process. Therefore, it will be easy to generate new versions of the software when new OpenGL versions are released. We already encapsulated Microsoft Research's C-based Detours library[6] into typesafe C++ classes. This keeps implementation efforts for additional detours low. The low implementation effort is accompanied by the fact, that debugging the intermediary is hassle-free. An intermediary's runtime performance is dominated by its inner functionality. Our preliminary experience shows, that the cardinal bottleneck is bandwidth consumption when the OpenGL command and data stream is transmitted between processes.

On our way to pragmatical interoperability, as exemplified in the application scenario, the main research question is interoperability between OpenGL streams from different applications. In preliminary experiments we instrumented applications from the NeHe OpenGL tutorials[7], the KUKA.Sim[8] robotics simulation suite, and Bitmanagement's BS Contact[9] generic VR-platform. With these applications, which use traditional, geometry-based rendering techniques, we adjusted camera parameters and excavated background graphics. We are curious, how it will work out with shader-based rendering techniques. We will discuss interoperability of OpenGL command streams in our future work. Then, we will be able to show, whether OpenGL opens up interactive three-dimensional graphics as pragmatic interoperability platform.

Acknowledgements

This work is supported by the German Ministry of Education and Science (BMBF), within the ViERforES-II project No. 01IM10002B. Mario Pukall is funded by German Research Foundation (DFG), project number SA 465/31-2.

5. REFERENCES

[1] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 277–290, New York, NY, USA, 2005. ACM.

[2] Emanuela Boutin-Boila. TechViz XL, March 2010.

[5]The Khronos Group provides a formal specification of OpenGL in their registry at `http://www.opengl.org/registry/`.

[6]Detours' project page is available at `http://research.microsoft.com/projects/detours/`.

[7]The NeHe tutorials are available at `http://nehe.gamedev.net`.

[8]KUKA.Sim Viewer is available at `http://www.kuka-robotics.com/en/products/software/kuka_sim/`.

[9]BS Contact is available at `http://www.bitmanagement.com/`.

[3] Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, and Kai Li. Software Environments For Cluster-Based Display Systems. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 202–210, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[5] K. Doerr and F. Kuester. CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):320 –332, March 2011.

[6] Craig Dunwoody. *The OpenGL Stream Codec – A Specification*. Silicon Graphics, 1996. `http://www.opengl.org/documentation/specs/gls/glsspec.txt` Accessed 2011-08-16.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[8] Michael M. Gorlick and Rami R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pages 23–34, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[9] Graphic Remedy. gDEBugger. Version 5.8, December 2010.

[10] Mark Grechanik, Don Batory, and Dewayne E. Perry. Integrating and Reusing GUI-Driven Applications. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 1–16, London, UK, UK, 2002. Springer-Verlag.

[11] Object Management Group. OMG Unified Modeling Language^{TM}(OMG UML), Superstructure Version 2.1.2, 2007.

[12] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 43:1–43:10, New York, NY, USA, 2008. ACM.

[13] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, WINSYM '99, page 14, Berkeley, CA, USA, 1999. USENIX Association.

[14] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[15] Gi Jung and Soon Jung. A Streaming Engine for PC-Based 3D Network Games onto Heterogeneous Mobile Platforms. In Zhigeng Pan, Ruth Aylett, Holger Diener, Xiaogang Jin, Stefan Göbel, and Li Li, editors, *Technologies for E-Learning and Digital Entertainment*, volume 3942 of *Lecture Notes in Computer Science*, pages 797–800. Springer Berlin / Heidelberg, 2006.

[16] Veit Köppen and Gunter Saake. Einsatz von Virtueller Realität im Prozessmanagement. *Industrie Management*, 2:49–53, 2010.

[17] Miguel-Ángel Manso, Monica Wachowicz, and Miguel-Ángel Bernabé. Towards an Integrated Model of Interoperability for Spatial Data Infrastructures. *Transactions in GIS*, 13(1):43–67, 2009.

[18] Microsoft Corporation. *Microsoft Developer Network Online Documentation – LoadLibrary Function (Windows)*, July 2011. `http://msdn.microsoft.com/en-us/library/ms684175.aspx` Accessed 2011-08-10.

[19] Microsoft Corporation. *Microsoft Developer Network Online Documentation – SwapBuffers Function (Windows)*, March 2011. `http://msdn.microsoft.com/en-us/library/dd369060.aspx` Accessed 2011-08-19.

[20] Microsoft Corporation. *Microsoft Developer Network Online Documentation – wglGetProcAddress Function (Windows)*, March 2011. `http://msdn.microsoft.com/en-us/library/dd374386.aspx` Accessed 2011-08-17.

[21] Alex Mohr and Michael Gleicher. HijackGL: Reconstructing from Streams for Stylized Rendering. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '02, pages 13–20 and 154, New York, NY, USA, 2002. ACM.

[22] John Stavrakakis, Masahiro Takatsuka, Zhen-Jock Lau, and Nick Lowe. Exposing Application Graphics to a Dynamic Heterogeneous Network. In *Proceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, WSCG '2006, pages 71–78, 2006.

[23] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. The 22 Megapixel Laptop. In *Proceedings of the 2007 Workshop on Emerging Displays Technologies: Images and Beyond: the Future of Displays and Interacton*, EDT '07, New York, NY, USA, 2007. ACM.

[24] Magnus Strengert, Thomas Klein, and Thomas Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 81–88, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[25] VirtualBox Service Desk. #475 – 3D acceleration support for VBox guests. `http://www.virtualbox.org/ticket/475` Accessed 2011-10-10.

[26] Stephan Vornholt, Michael Stoye, Ingolf Geist, Veit Köppen, and Gunter Saake. Datenmodell zur flexiblen Verwaltung von Datenaustauschprozessen in der virtuellen Produktentwicklung. In Roland Kasper u.a., editor, *10. Magdeburger Maschinenbau-Tage 27.-29.09.2011*, Magdeburg, September 2011. Otto-von-Guericke-Universität Magdeburg.

[27] Levent Yilmaz. Using Meta-Level Ontology Relations to Measure Conceptual Alignment and Interoperability of Simulation Models. In *Proceedings of the 39th Conference on Winter Simulation*, WSC '07, pages 1090–1099, Piscataway, NJ, USA, 2007. IEEE Press.