Exposing Proprietary Virtual Reality Software to Nontraditional Displays

Maik Mory
Otto-von-Guericke-University
maik.mory@ovgu.de

Steffen Masik
Fraunhofer IFF
steffen.masik@iff.fraunhofer.de

Richard Müller
University of Leipzig
rmueller@wifa.uni-leipzig.de

Veit Köppen
Otto-von-Guericke-University
veit.koeppen@ovgu.de

Abstract

Nontraditional displays just started their triumph. In contrast to traditional displays, which are plane and rectangular, they do not only differ in design and architecture; they also implicate different semantics and pragmatics in the rendering pipeline. We strive for a generic solution that couples legacy applications with nontraditional displays. In this paper, we present an architecture and a respective experiment, which exposes a proprietary virtual reality software to a 360 degree virtual environment. Therefore we introduce a rigorous master-slave design. The proposed architecture requires discussion of the following details: how to access a proprietary application's OpenGL stream; how to transmit the OpenGL stream efficiently in a clustered rendering setup; how to process the OpenGL stream for adaption to nontraditional display semantics; and how to deal with the arising code complexity, withal. Our design decisions are highly interdependent. The presented architecture overcomes limitations, which were implied by client-server design in earlier work. The proposed rigorous master-slave design is totally transparent to the client software, and reduces interdependencies between rendering software and rendering clusters. Thus, it inherently reduces network round trips and promotes the use of scalable multicast. Our architecture is tested in a reproducible experiment, which provides a qualitative proof of concept.

Keywords: Nontraditional Display, OpenGL, Distributed Rendering, Multicast, Interoperability, Generative Programming

1 INTRODUCTION

Nontraditional displays (e.g., CAVEs, powerwalls, domes) just started their triumph. In the 1990s, nontraditional displays were driven by special monolithic rendering hardware. Together with the advent of cheap general and graphics computation power driven by the computer games industry, research shifted towards rendering clusters made from off-the-shelf hardware, during the first decade of our century.

In contrast to traditional displays, which are plane and rectangular, nontraditional displays do not only differ in design and architecture; they implicate different semantics and pragmatics in the rendering pipeline, too. This paper discusses several solutions that were implemented to bring legacy software to nontraditional displays. Our predecessors inherited client-server semantics from the OpenGL specification. We present an implementation that rigorously uses master-slave semantics to enhance scalability of distributed rendering architectures, beside other minor optimizations. Throughout our discussion, we use a practical application scenario that is presented in this section's remainder together with a problem statement. Section 2 provides background on interoperability of distributed virtual reality software. An architecture, which enables node-based OpenGL stream processing with little interference to the communication between a central OpenGL client software and its local OpenGL server hardware, is presented in Sections 3 and 4. That our proposed architecture is feasible is proven with a reproducible experiment in Sections 5 and 6. Finally, we conclude with an outlook on future work.

1.1 Application Scenario

Today, every engineering process is supported by software. Any reasonable engineering software has a visualization component. Note, the visualization component is not necessarily a dominant or permanent element of the user interface; we only require it to be available. We choose Bitmanagement's BSContact [8] as an exemplary sample of generic information and geometry visualization. Basically, BSContact is a generic 3D file viewer for VRML and X3D. Most engineering software is able to export these data formats. Although, for us BSContact's primary functionality is not that relevant. For us it is important, that BSContact uses the most widespread patterns of three dimensional data visualization; it is proprietary; and it renders interactive geometry.

Several nontraditional displays exist in various dimensions and shapes. Most of them have an entertainment background, but some of them are also used for research and industrial applications. One of those systems is the ElbeDom located at the Fraunhofer Institute for Factory Operation and Automation (IFF) in Magdeburg, Germany. The ElbeDom is a large cylindrical virtual environment. It has been designed to satisfy the demand

for immersive virtual reality (VR) and massive multi-user collaboration in the areas of virtual manufacturing and factory planning. A detailed description of the system and a comparison of similar projection systems are provided in [22]. Basically, the ElbeDom is representative hardware for distributed, tiled rendering on curved screens.

Briefly, the ElbeDom's cylindrical screen is 6.5 m high and 16 m in diameter. The 330 m^2 screen is covered by six LDT G2 laser projectors. The projectors operate at 1600×1200 pixels. Thus, the viewer is surrounded by approximately eleven megapixels from -20° below horizon to +30° above horizon and full 360° in horizontal. Six so-called warping engines geometrically adjust and blend the six projector's pictures. Their input is generated by a cluster of six commodity nodes. Throughout this paper we call the six nodes slaves. Among other control and automation nodes, there is a dedicated headed node for the operator. We will call it the master in this paper. The commodity cluster is interconnected via off-the-shelf GBit Ethernet.

1.2 Problem Statement

Our primary goal is to provide interoperability between arbitrary proprietary virtual reality software and arbitrary nontraditional displays. Looking at this paper's application scenario alone, several questions arise. Bitmanagement's BSContact has been developed for traditional displays connected to local graphics hardware. Thus, one has to consider aspects of syntactical, semantical and pragmatical interoperability between the visualization software and the nontraditional display's rendering pipeline. Challenges, how to interface a proprietary software's visualization component and how to handle variant implementations of several hundred OpenGL functions frame our considerations.

2 BACKGROUND

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. Therefore, a component is a definable software artifact. Connections of components are differentiated between tight couplings and loose couplings. Tight coupling exploits interdependencies and relations between components; the connected components are not supposed to be exchanged. A loose coupling minimizes dependencies and relations between the components 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.

Coupling components is 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 [14], which we briefly introduce as follows:

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.

Another aspect about couplings are messaging patterns. Most procedural, object-oriented, and distributed systems follow the client-server pattern. A server component provides an interface. A client component requires 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. Servers or services can be stateful or stateless. If the server is stateless, the effect to a request depends on the request only. If the server is stateful, the reply depends on the request and on the server's state.

The Gang of Four [7] identified an extreme variant, where the request declares the client's interest in a series of replies – the observer pattern. Other names are one-way messaging, publish-subscribe, producer-consumer, or master-slave as we call it in this paper. Master-slave tends to be loosely coupled, because in an ideal implementation the consumer requires no knowledge about identity or number of the producers and vice versa as well, for example.

An interface definition covers a slice 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.

In a system of n components, one may implement $O(n^2)$ adapters for each coupled pair of components. When there is a common concept, which is shared among several interfaces, established protocols and other

interface specifications are reused for multiple components. This we call an interoperability platform. In a system of n components, one implements $O(n)$ adapters between each component and the interoperability platform.

There has been vast work to establish interoperability platforms for VR applications. For example, Schumann [23] uses the high-level architecture (HLA) for syntactical interoperability among distributed simulations; Ošlejšek [21] tries to establish semantic interoperability with a unified scene graph definition. The crucial point in the design of an interoperability platform is the common concept shared between participants. In our observation, there are two types of interoperability platforms: those which declare and impose an artificial common concept; and those which find and exploit an existing common concept. We believe that the latter have better chances to succeed in software evolution. Looking at the abundance of VR software, there is one thing obviously common: OpenGL. The OpenGL specification defines syntax through function signatures together with a finite state machine and it defines semantics through human readable documentation for modules and functions.

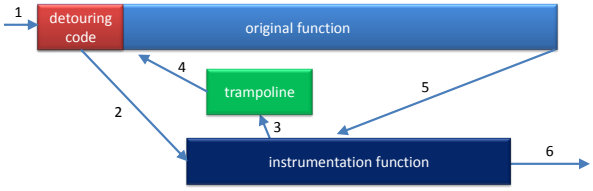
We are not the first ones who exploit the well supported OpenGL industry standard for interoperability. The commercial software products TechViz XL [2] and ICIDO's Capture [1] impressively show the potential. However, because they are closed-source they give little value to our discussion. During our discussion we mostly refer to selected aspects of Chromium [11, 16], Lumino [25], and BroadcastGL [13].

3 EXPOSING A PROPRIETARY APPLICATION'S OPENGL STREAM

In [18], the authors evaluate four techniques, how to intercept a proprietary application's invocations of the OpenGL API. Three out of the four techniques have been used in multi-hosted rendering before. The relink library technique (e.g., MPIglut [15], although it is not a node-based stream processor) cannot be applied to proprietary software. The replace dynamic library technique (e.g., Chromium [11]) is unreliable within MS Windows' dynamic library facility. The virtual device driver technique (e.g., VirtualBox [26]) does not scale for complex application scenarios. We prefer the binary interception technique because it is flexible and robust at once.

The binary interception technique was introduced by Hunt and Brubacher [12] to instrument and extend proprietary software. An injected intermediary manipulates the proprietary software's binary image at runtime. We illustrate the principle in Figure 1. For each function that should be instrumented, the intermediary installs what is referred to as detour. The installation procedure for a

(a) In unmodified software, the client requests the server function (1); then the server function returns a reply (2).



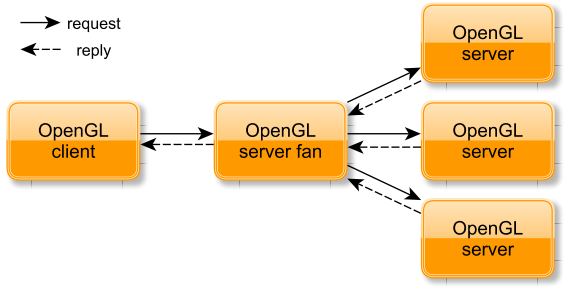
(b) In intercepted software, the detouring code redirects the instruction pointer to an instrumentation function (2). We internally invoke the original function through the trampoline (3,4,5). After processing the request-reply tuple, we return to the client (6).

Figure 1: Binary Interception

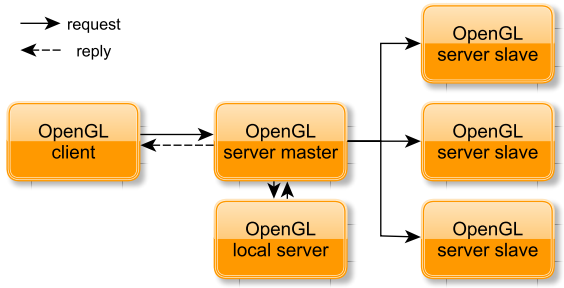
function overwrites the first bytes of the server's function with machine code, which detours the execution path to the intermediary's function. When the OpenGL client invokes an intercepted OpenGL server's function, the overlaid detouring code is executed instead of the original code. In effect, the client invokes the intermediary's function. The installation procedure produces a so-called trampoline function, which keeps the original server's function available. The trampoline contains a backup of the server function's machine code that was overwritten during installation of the detour and additional machine code that repatriates the execution path to the unmodified remainder of the server function's machine code. In effect, invocations of the trampoline delegate calls to the server.

Microsoft Windows' implementation of OpenGL, which is determined to be compatible with OpenGL version 1.1, provides 2400 functions (c.f., Section 6), which divide into 357 core functions, 1671 extension functions, and 372 alias functions. Core functions are provided in the `opengl32.dll`'s symbol table. We install interceptions for every core function during startup time before the application is able to access them. Extension functions are provided on the client's demand on the server through the `wglGetProcAddress` function. We install interceptions for every extension function when it is passed through the `wglGetProcAddress` function for the first time. What is known as alias functions are alias names for core or extension functions. In the data structure that tracks installed interceptions, alias functions are associated with their respective original functions.

In result, the complete OpenGL API is instrumented. Our instrumentation functions first transparently delegate the client's request to the original function through the trampoline. After the original function returned (item 5 in Figure 1b), the request-reply tuple is available to the instrumentation function from a totally transparent observation. In our further discussion, a published sequence of request-reply tuples we call the OpenGL



(a) Traditionally, the client's requests are fanned to the distributed renderer and replies are merged after a full round trip. The display's semantics and pragmatics are opaque to the client.



(b) With our approach, the server master uses a local server to gather replies with minimal latency. Then it publishes the requests with attached replies to the slaves without network round trips. The slaves adapt semantics and pragmatics transparently.

Figure 2: A node-based OpenGL stream processor that multicasts request-reply tuples has looser couplings.

stream. After publication of the new OpenGL stream element, we return to the original OpenGL client using the reply from the original server function. Thus, we have extracted the OpenGL stream from the client-server coupling with minimal interference.

4 FULL MULTICAST SEMANTICS FOR OPENGL STREAM DISTRIBUTION

Commonly, implementations of node-based, distributed OpenGL processing use unicast via TCP/IP for data distribution. We guess that this design decision has two major reasons. Back in the days of the first hype about rendering on commodity clusters (for a survey see Chen et al. [5]), TCP/IP was available, tried-and-tested, and well supported. Further, for those systems it is a basic assumption, that the invocation actually happens on the remote site; and thus, return values and argument alterations have to be propagated back from the distributed OpenGL server to the central client. Figure 2a illustrates this in terms of distributed systems. The OpenGL client's requests are fanned to multiple OpenGL servers. For operations with output, the server fan adapts the client's request for the remote servers and merges the remote servers' replies to a singular reply for the client. Thus, the nontraditional display's semantics and pragmatics are opaque to the client. As an example for opaque semantics, all of our predecessors tried to map the semantics of windowing and camera setup (e.g., the `glViewport` function) from their tiled display setups to the clients' calls.

In 2005, Ilmonen et al. [13] unveiled the potential of non-unicast stream distribution in the context of multi-tile rendering. They discovered, that unicast stream distribution does not scale with the number of tiles, because shared commands that are used by n tiles have to be sent n times. The more tiles a distributed graphics application uses, the more neighboring and blended regions share geometry data. Global state changes are shared between all tiles. Ilmonen et al. describe an experiment, where they use broadcast via UDP/IP for stream distribution and a TCP/IP backchannel to add reliability and congestion control. Their main contribution is an empirical proof, that broad- and multicast OpenGL stream distribution scales with the number of tiles in a centralized application with distributed graphics.

Lorenz et al. [16] implement a modification of Chromium, which uses multicast for parallelizing commands and unicast for serializing commands. In their reasoning, serializing commands are those commands that are unique to each remote server. Serializing commands are different with respect to the distinct peers, because they are adapted on the client's side of the network in the server fan. We argue that they could be parallel calls – and thus be appropriate for multicast distribution – if the adaption stage would be shifted from the server fan to the remote peers.

Ilmonen et al. [13] already shift adaption of the requests to the remote peers. In their discussion, they point out that commands which require merged replies from the distributed server stall the streaming. Neal et al. [20], who advance efficiency in multicast OpenGL stream distribution by applying compression techniques to the distributed stream, observe the same problem. They identify, that commands which have replies effectively are network round trips and hence cause blocking at the client. This leads us to the question: Is it really necessary to aggregate state from the distributed server? Neal et al. as well as Ilmonen et al. still use client-server semantics as it is defined in the OpenGL API specification. As an aside, they borrow a potential solution from prior work, that round trips may be avoided by state management [4].

Chromium [11] and Lumino [25], for example, implement state management. State management emulates the OpenGL state machine as a component of the stream processing framework. Stavrakakis et al. [25] claim that state management is necessary for two reasons: late joiners should be able to retrieve OpenGL machine's state; and operation accumulation can be used for compression of transferred data (i.e., a-priori-aggregated state of

the distributed server). However, state management is expensive. With emulation software, it is tedious to keep track with the original implementations in functionality and in performance. When Chromium introduced state management, they assumed that the client may run on a platform without graphics acceleration. Today, every host has basic graphics hardware acceleration or at least a good software implementation. Hence, we consider the topic of emulated state management obsolete. Even more, we explicitly recommend using the central application's local OpenGL implementation.

This yields a novel architecture for centralized applications with distributed graphics, which is depicted in Figure 2b. Our implementation of the intercepted OpenGL API, which we name Vanadium[1], first invokes the original local OpenGL server with unmodified commands from the client and returns unmodified output to the client. This renders Vanadium transparent in functional behavior to the OpenGL client and in appearance to the user at the host with the central application, as well. After the trampoline function has returned and before the decorator function returns (cf., Figure 1), the decorator publishes the OpenGL stream. In contrast to earlier work, the stream does not only contain opcode and relevant input arguments (i.e., the request), but also every output, like return values and referenced arrays (i.e., the reply). After publishing the stream, any further stream processing is asynchronous. Thus, there are no round trips on the network anymore.

We agree to Stavrakakis et al. [25], that there should be a possibility to join lately to the stream. Nevertheless, late joining is a very infrequent event. Thus, we abnegate the implementation cost and runtime cost of dedicated state management. In the infrequent case of a late join, we are able to retrieve the OpenGL machine state directly from the original driver vendor's implementation through the `glGet` function and other inspection functions; at least unless the client uses no deprecated technique like display lists. The joined slave uses its adaption (see Section 5) to map the master's late state to a consistent slave's state. Then, the stream processing can continue. Regarding stream compression, which is a topic to all distributed OpenGL renderers, because the network bandwidth bottleneck is very dominant, we refer to Neal's work [20] and Lorenz' work [16].

5 EXAMPLES FOR DETACHED PROCESSING OF THE OPENGL STREAM

In this section, we clarify the architecture shift that we propose in Section 4. Therefore, we describe the processing chain as we implemented it with Bitmanagement's BSContact as source and VDTC's ElbeDom as sink (cf., Section 1.1 and Figure 3).

Figure 3: VDTC's ElbeDom driven by Vanadium. The transparently distributed application is Bitmanagement's BSContact with a software visualization scene [19].

Please note, that we intercept the whole OpenGL and WGL API. In the decorator functions we first delegate the call to the original function via the trampoline. After the trampoline function returned with the output from the original server, the processing described in this section takes place. After our processing, the decorator function returns the values from the local original server invocation to the original client (cf., Figures 1 and 2).

First of all, we need to handle recursion. As we use binary interception, we get each and every call of the OpenGL API – literally. So, there are the calls actually made by the client; and there are recursive calls by the original server implementation to itself. For example, the `wglDescribePixelFormat` function calls itself; many functions call the `glFlush` function internally. We distinguish client calls and recursive server calls by tracking stack depth with a counter. When there is no active call from the client, the counter rests at minus one. During client's requests the counter is zero. When the server calls itself, the counter is greater than zero. Recursive calls are skipped in stream processing, because they reflect internal behavior of the original local server and thus do not matter. Now, the stream contains every invocation actually made by the client.

[1] Vanadium as used in the presented experiment is provided in the additional material. It will be open-sourced, soon.

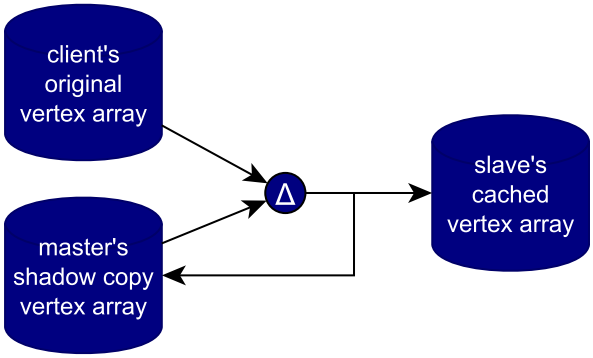


Figure 4: Vertex Array Cache – The master maintains a shadow copy of the client's vertex arrays. The vertex array transmission to the slave is differential.

Looking at the stream of client calls, there is a huge amount of calls that are irrelevant to the remote display. Most commercial OpenGL software use an offscreen rendering technique for auxiliary calculations, like mouse pointer ray collision testing or occlusion culling. In Bitmanagement's BSContact, a temporary viewport is used, that is overdrawn by visible content before the next SwapBuffers invocation. BSContact's hidden viewport is easily determinable, because it is always square (e.g., 100×100 pixels) and smaller than the window's rendering area. We skip all calls that are made to the finally invisible viewport. Then BSContact sets the viewport to the whole visible area and renders the visible content. We passthrough these calls for handling on the nontraditional screen. As can be seen from the invocation log, another viewport is set, that is always 86 pixels in height. Because we could not imagine a mapping from its two-dimensional content to the 360 degree screen of the ElbeDom, we skip the two-dimensional content, too. The considerations in this paragraph are highly interdependent with the client software and thus, have to be reconsidered for every new client.

To reduce bandwidth usage further, we apply a caching technique to the vertex array facility (Figure 4). To achieve this, our decorated `glDrawElements` function and related vertex array draw functions determine the array in CPU's RAM that should be drawn by pointer and by size. The master keeps a shadow copy that resembles the arrays in the slave's cache. We introduce cache management commands in the stream to advice the slave for modifications of its cache. Initially the master's shadow copy and the slave's cache are empty. When there is a new array (i.e., pointer is not in shadow copy mapping), it is added to the shadow copy and transmitted to the slaves. When the client draws a known array (i.e., pointer exists in shadow copy mapping) whose content is unchanged (i.e., client's array equals shadow copy array), the slave is told to use the cached array. When the client draws a known array whose content has changed (i.e., pointer exists in shadow copy mapping and client's array is not equal to shadow copy array), the changed array is transmitted to the slave before usage. At the slave, the modified array replaces the respective array, because obviously the client discarded the old content before. When distributing the arrays and when referring to them in draw commands, the master uses handles that are derived from its shadow copy index. During cache management commands, the slave maintains a mapping between the master's handles and the cache's pointers. After cache synchronization the master emits the draw command. Then the slave uses its master's-handle-to-slave's-pointer mapping to invoke the draw command with valid data. This simple caching technique doubles the client application's memory consumption with respect to its vertex arrays. We do not recommend the use of hashes, because collisions in the index may corrupt the stream fatally [16]. For us, usage of the `memcmp` function worked out by reducing network bandwidth at negligibly increased CPU load.

For networking, we use ØMQ [10], which provides us with superior inter-thread, inter-process, cluster-wide, and world-wide messaging. One command is one message. For each command in OpenGL's API specification, we derive a struct, which contains the opcode, any value arguments, and the return value if applicable. Array arguments are packed into submessages. Because ZeroMQ takes care of message sizes robustly, therewith we significantly reduce any risk associated with wrong array sizes in C/C++. As an optimization, we exploit that call-by-value arguments already are packed in the stack. Thus, we only need to copy a slice from the stack into the corresponding slice of the message buffer struct. ZeroMQ offers various reliable multicast protocols for data distribution. After message transmission through ZeroMQ to the slaves, the commands are dispatched to handler functions based on their opcode. The slave's default handler implementation directly mimics the client node's invocation. Some handler functions are modified to implement adaption of the stream at slave side.

Comparable to the mapping of array pointers the slave implements a mapping of OpenGL names. In OpenGL's terminology, names are numeric identifiers for objects in the OpenGL state machine. For example, the `glGenTextures` function outputs integers to the client, which the client should use to identify and refer textures unambiguously. Usually, equally replayed commands should yield equal names. However, we cannot guarantee that for heterogeneous environments, for late joiners, and when splicing command streams. The mapping mechanism is simple. There are functions that generate (i.e., output) names and there are functions that use (i.e., input) names. The slave maintains an associative array with the master's names as keys and the slave's names as values. Please remember, that the master includes invocations' outputs in the stream. When

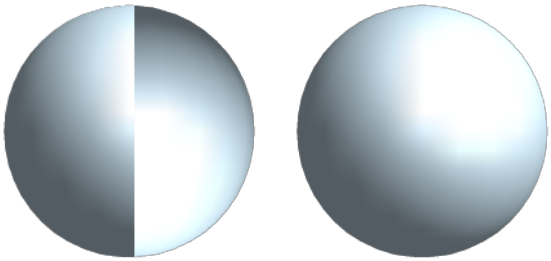


Figure 5: If lighting is calculated relative to the camera and the camera has different poses on different tiles then the light's pose is different on each tile (left). Reposing light sources with respect to the tiles' camera pose differences compensates the visual inconsistency (right).

a command that generates names is called, the slave extracts the master's names from the arrived message. It replays the command, which yields the slave's names. Then the pair is added to the map. When a name is used, the message refers the master's name. The slave decodes the master's name to its local name using the map. We call this mechanism name mapping.

The ElbeDom is an exemplary virtual environment. The cylindrical, surrounding screen has little potential for two-dimensional WIMP semantics. Hence, we do not even try to map the client's WIMP semantics to the system's VE semantics. The filtering stage at the master yields a singular stream with visually relevant content only. Therefore, calls to the `glViewport` function, which refer to the whole visible window area at the master, are mapped to fullscreen rendering at the slaves. This leaves us to adapt the camera pose. The appropriate place in the stream to achieve camera adaption is client-software-specific. With Bitmanagement's BSContact, the most robust solution is to modify invocations of the `glLoadIdentity` function and the `glLoadMatrix` function where the model view matrix is selected. There we premultiply the tile's camera orientation. This gives us the basic adaption to the ElbeDom's 360 degree view. Additionally, the ElbeDom's warping and blending facility specifies asymmetric frusta for each tile. At the `glFrustum` function we discard the client's inputs for a symmetric view into the window and overwrite with the tile's asymmetric frustum configuration. This completes camera adaption to the ElbeDom's interleaved frustum configuration.

The adaption pipeline is completed by a pragmatic adaption of lighting. Like most other OpenGL software, BSContact uses lighting to give three-dimensional impression. BSContact's lighting is designed relative to the camera. Because we rotate the world to adapt the camera position, the lighting is inconsistent between the tiles (cf., Figure 5). Hence, we install a filter on the `glLight` function family, which applies the tile's camera pose to the position and direction lighting parameters. Thus, the lighting is consistent through all of the ElbeDom's six tiles.

6 HANDLING CODE COMPLEXITY

The functionality that we describe in this paper handles several hundred functions from the OpenGL API. Moreover, we talk about variant functionality. During analysis of the application's OpenGL stream, we need a variant that logs the OpenGL stream to disk. The sender is a variant that implements serialization. The receivers have to implement deserialization. Processing of the OpenGL streams yields building blocks (i.e., filters, adapters), that ideally should be individually and independently reusable with a broad variety of virtual reality software and virtual environments. Nevertheless, we expect that processing nodes need to be tailored with respect to functional and non-functional behavior (cf., Siegmund et. al. [24]) as soon as more than one application will be supported. At a first glance, this requires a codebase of several thousand repetitive functions. At the second glance, we see two core problems: repetitiveness and variability.

Under the bottom line, we deal with a component oriented system, where the components stem from a software system family. Such a family includes a number of systems that are similar enough from an architectural point of view to be assembled from a common set of components. We achieve development efficiency through Generative Programming (GP) [6]. The main goal of GP is to generate a partial or an entire software system automatically from such implementation components. The requirements for the desired result, i.e. the generate, are defined in a domain specific language (DSL). A DSL is a specialized and problem-oriented language for domain experts to specify concrete members of a system family. This specification is processed by a generator, which automatically builds the system by combining components according to configuration knowledge. The Generative Domain Model (GDM) in Fig. 6 illustrates the concept of this paradigm establishing a relationship between the basic terms of GP. It consists of the problem space, the solution space, and the configuration knowledge mapping both spaces. The problem space includes domain specific concepts and features to specify requirements by means of one or more DSL(s). The solution space offers elementary and reusable implementation components corresponding to the system family architecture. The configuration knowledge comprises illegal feature combinations, default settings, construction rules, and optimizations as well as related information. In order to instantiate this theoretical concept, we perform a technology projection. Therefore, we identify concrete techniques for the elements of the GDM.

In particular, we enhanced Microsoft Visual C++ at Visual Studio 2010's prebuild stage. First, our Python

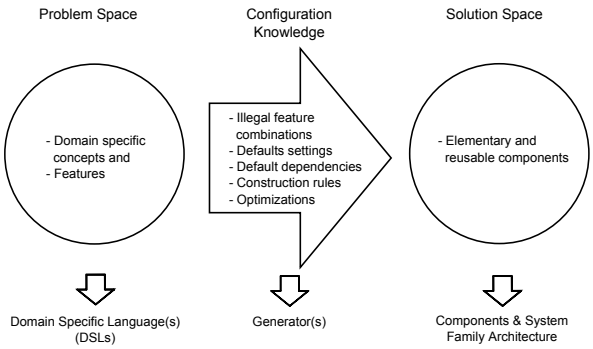


Figure 6: Generative Domain Model [6]

script parses the OpenGL API's formal specification. At the time of our experiment, the OpenGL specification (revision 12819) defines 2269 functions. Microsoft Windows' OpenGL windowing system (WGL) (revision 10796) adds 131 function definitions. Thus, an OpenGL application on Windows has access to a repository of 2400 functions. Secondly, within our Python-based domain specific language, a feature configuration is modeled. Both models together are applied to a template engine, which generates C++ source code. Then, the prebuild step, or code generation step respectively, terminates and the Visual C++ tool chain builds the executable software artifacts. Applying the means of the generative paradigm, repetitive development tasks are automated and the high variability of the virtual reality domain is made manageable.

7 CONCLUSION AND FUTURE WORK

We presented an architecture for node-based OpenGL stream processing. It is highly interoperable with proprietary and legacy software, because we use a robust technique for function interception, and especially because we adapt from OpenGL's client-server pattern to a master-slave pattern, which is more feasible for stream processing. The adaption is as transparent as possible to the client. Thereby, the proposed architecture removes any interdependencies between rendering software and rendering clusters. We show that there is no source code access to the application required. The reduced interdependency promotes to use scalable multicast, and removes network roundtrips. The architecture is tested by a reproducible experiment, which we comprehensively described in Section 5.

For sake of clarity, we focused on the combination of one virtual reality software with one nontraditional display. We want to generalize our approach of course. At the time of this writing, we are experimenting with other rendering software, and we are experimenting with other nontraditional displays, which have different pragmatics than the ElbeDom. One open question we are researching is how to deal with frustum culling and occlusion culling in legacy software. These techniques are highly optimized towards traditional displays. With BSContact, we could switch them off through its ActiveX interface. If culling was not disengageable, further investigation would be necessary. In the long term we are curious, if fully programmable pipelines may yield new rendering pipeline semantics, will our approach scale?

With our proposed rigorous master-slave design, the nodes of an OpenGL stream processor are coupled more loosely than in competing frameworks. Therefore, our architecture supports development and recombination of OpenGL stream processor nodes. For example, we would be glad to see an effect processing engine (e.g., Haringer and Beckaus [9], or Brennecke et al. [3]) applied to our OpenGL stream processing approach. In summary, we see great potential for innovative use cases in entertainment and engineering.

ACKNOWLEDGEMENTS

We are thankful to Fraunhofer IFF's Virtual Development and Training Center (VDTC), which provided its ElbeDom for our experiments.

REFERENCES

[1] IC:IDO – The Visual Decision Company. `http://www.icido.de/PDF/ICIDO_Broschuere_A4.pdf`, Stuttgart, November 2006. In German.

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

[3] Angela Brennecke, Christian Panzer, and Stefan Schlechtweg. vSLRcam – Taking Pictures in Virtual Environments. In *WSCG (Journal Papers)*, pages 9–16, 2008.

[4] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, pages 87–95, New York, NY, USA, 2000. ACM.

[5] 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.

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming Methods, Tools and Applications*. Addison-Wesley, 2000.

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

[8] Bitmanagement Software GmbH. BS Contact 8.0. `http://www.bitmanagement.de`.

[9] Matthias Haringer and Steffi Beckhaus. Dynamic Visual Effects for Virtual Environments. In *WSCG (Full Papers)*, pages 49–56, 2010.

[10] Pieter Hintjens. *ØMQ – The Guide*. iMatix Corporation, `http://zguide.zeromq.org/`.

[11] 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.

[12] 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.

[13] Tommi Ilmonen, Markku Reunanen, and Petteri Kontio. Broadcast GL: An Alternative Method for Distributing OpenGL API Calls to Multiple Rendering Slaves. In *WSCG (Journal Papers)*, pages 65–72, 2005.

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

[15] Orion Sky Lawlor, Matthew Page, and Jon Genetti. MPIglut: Powerwall Programming Made Easier. In *WSCG (Journal Papers)*, pages 137–144, 2008.

[16] Mario Lorenz, Guido Brunnett, and Marcel Heinz. Driving Tiled Displays with an Extended Chromium System Based on Stream Cached Multicast Communication. *Parallel Computing*, 33(6):438 – 466, 2007. Parallel Graphics and Visualization.

[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] Maik Mory, Mario Pukall, Veit Köppen, and Gunter Saake. Evaluation of Techniques for the Instrumentation and Extension of Proprietary OpenGL Applications. In *2nd International ACM/GI Workshop on Digital Engineering (IWDE)*, pages 50–57, Magdeburg, Germany, 2011.

[19] Richard Müller, Pascal Kovacs, Jan Schilbach, and Ulrich Eisenecker. Generative Software Visualizaion: Automatic Generation of User-Specific Visualisations. In *2nd International ACM/GI Workshop on Digital Engineering (IWDE)*, pages 45–49, Magdeburg, Germany, 2011.

[20] Braden Neal, Paul Hunkin, and Antony McGregor. Distributed OpenGL Rendering in Network Bandwidth Constrained Environments. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *EGPGV*, pages 21–29. Eurographics Association, 2011.

[21] Radek Ošlejšek. Virtual Scene as a Software Component. In *WSCG (Posters)*, pages 33–36, 2008.

[22] Wolfram Schoor, Steffen Masik, Marc Hofmann, Rüdiger Mecke, and Gerhard Müller. ElbeDom: 360 Degree Full Immersive Laser Projection System. In *Virtual Environments 2007 - IPTEGVE 2007 - Short Papers and Posters*, pages 15–20, 2007.

[23] Marco Schumann. *Architektur und Applikation verteilter, VR-basierter Trainingssysteme*. Dissertation, Otto-von-Guericke-University Magdeburg, Magdeburg, November 2009. In German.

[24] Norbert Siegmund, Martin Kuhlemann, Sven Apel, and Mario Pukall. Optimizing Non-functional Properties of Software Product Lines by means of Refactorings. In *Proceedings of Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 115–122, 2010.

[25] 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.

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