Enabling Closed-Source Applications for Virtual Reality via OpenGL Intercept-based Techniques

David J. Zielinski[1]
Duke University

Ryan P. McMahan[2]
University of Texas at Dallas

Solaiman Shokur[3]
International Institute of Neuroscience at Natal

Edgard Morya[4]
International Institute of Neuroscience at Natal

Regis Kopper[5]
Duke University

ABSTRACT

Everyday, people use numerous high-quality commercial software packages on desktop systems. Many times, these software packages are not able to access specialized virtual reality (VR) display and input devices, which can enhance interaction and visualization. To address this limitation, we have been using the well-known OpenGL intercept concept to insert middleware at runtime between the application and the graphics card. In this paper, we motivate the use of OpenGL intercept techniques and present three intercept-based techniques that enable closed-source applications to be used with VR systems. To demonstrate the usefulness of these intercept-based techniques, we describe two case studies. In the first case study, we enabled MotionBuilder, a commercial motion capture and animation software, to work with the Oculus Rift, a consumer-level head-mounted display (HMD). In the second case study, we enabled MATLAB, a commercial mathematics and simulation software, to run in the Duke immersive Virtual Environment (DiVE), six-sided CAVE-like system. In both cases, display and interaction are successfully handled by intercept-based techniques.

Keywords: Virtual Reality, OpenGL, MotionBuilder, MATLAB.

Index Terms: H.5.1 [Multimedia Information Systems]: Artificial, augmented, and virtual realities; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement.

1 INTRODUCTION

Virtual reality (VR) experiences often utilize special hardware [9] that increases the fidelity of the interaction and display for the purposes of increased immersion. The software for these experiences must utilize libraries, languages, or software packages that are developed to take advantage of these special display and interaction devices. Asking a user to learn a new application-programming interface (API) to utilize VR technology is non-trivial and is often a barrier preventing the adoption of VR by users of established desktop systems. Ideally, a developer would modify the desktop software code directly. However, if the current desktop software is commercial or proprietary, it is often closed-source, which restricts the addition of code and recompilation. In order to bypass such limitations for potential VR users, we propose the use of OpenGL intercept-based techniques.

[1]email address: djzielin@duke.edu
[2]email address: rymcmaha@utdallas.edu
[3]email address: solaiman.shokur@gmail.com
[4]email address: edmorya@gmail.com
[5]email address: regis.kopper@duke.edu

OpenGL is a standardized API (i.e., list of commands) that a programmer can use to draw graphics on a screen in real-time [8]. At runtime, an OpenGL application loads and accesses the OpenGL driver (a dynamic-linked library under the Microsoft Windows operating system), which contains the full functionality for each of the supported OpenGL commands. If we would like to modify the commands used by a closed-source application, we could create a replacement driver that is loaded by the application, which processes the commands before they are passed on to the true OpenGL driver. This technique is known as an OpenGL intercept, which was initially used in the WireGL [5] and Chromium [4] projects (see section 2 for more details).

In this paper, we present three OpenGL intercept-based techniques that can be used to enable closed-source applications for immersive VR systems. The first technique, called In-and-Out, utilizes the built-in scripting or plugin architecture of a closed-source application to facilitate access to VR input devices. The second technique utilizes predefined OpenGL geometry calls, known as intercept tags [16], which are interpreted as scene information, cues, or function calls, instead of being rendered. The third technique, called Driver-Mediated View, utilizes the VR input data within the intercept driver to manipulate the current view of the host's generated graphics.

To demonstrate the capabilities of these OpenGL intercept-based techniques, we provide two case studies of enabling closed-source applications for VR. The first case study involves enabling Autodesk MotionBuilder, a commercial 3D character animation and motion capture software, to be used with an Oculus Rift. MotionBuilder has been successfully used for brain-computer interface (BCI) experiments to provide visual feedback on a desktop [13]. Using intercept-based techniques, we enabled existing experiments in the Oculus Rift. The second case study involves enabling MathWorks MATLAB, a commercial computational software system and integrated development environment (IDE), for use with the Duke immersive Virtual Environment (DiVE), a six-sided cluster-based CAVE-like system. This case study allowed users to immersively view and interact with robotics-based simulations that they had created in MATLAB [17].

2 RELATED WORK

One of the first OpenGL intercept projects was WireGL, which used an intercept method to transmit OpenGL commands to one or more pipeservers [5]. WireGL evolved into the better-known Chromium, a software library for distributing streams of graphics calls to a cluster-based display environment [4]. Chromium has since been used to display graphics from closed-source applications in immersive VR systems, such as the Allosphere at the University of California Santa Barbara [3]. TechViz is a commercial product similar to Chromium. It uses an intercept-based approach for viewing the graphics generated by a desktop application on immersive VR systems [14].

In addition to duplicating the intercept OpenGL stream, researchers have also investigated ways to modify the contents of the stream. One example of this is HijackGL [7]. By modifying

the incoming OpenGL stream of commands, HijackGL can create several stylized rendering effects, including pencil sketch, blue print, and cartoonish appearances.

Our contribution is the presentation and application of existing techniques in the context of immersive virtual environments, along with our novel work with intercept tags and driver-mediated viewing.

3 OpenGL Intercept-Based Techniques

Before we review the three intercept-based techniques used in our case studies, we provide a clear explanation of how the underlying OpenGL intercept works in a Microsoft Windows operating system environment.

In Windows, dynamic-linked libraries (DLLs) are used to replace function calls with actual code at runtime, similar to the shared object libraries (SOs) used in Linux-based operating systems. When a DLL is requested by an application, the LoadLibrary function first looks in the application's current directory and then in the directories specified by various environment variables. By placing the intercept DLL in the current directory, we can ensure that it gets loaded before the actual OpenGL DLL.

To create a new intercept, we must first create an Opengl32.dll file that defines and implements all of the expected OpenGL functions (e.g. glColor3f). This list can be quite long, and if we fail to define all the OpenGL functions that the host is expecting, the host application may crash during startup. Next, during an initialization phase, we load the real Opengl32.dll file from within the intercept DLL using the LoadLibrary function, and then use the GetProcAddress function to make a table of the function locations of the real OpenGL functions. If we want to create an identity intercept, we just pass through all the data to the real DLL without making changes. We can also modify the passed in values. For example, by overriding the color values passed in to the glColor3f function, we can make all the objects in the scene blue (see Figure 1). In our implementations, we utilized the open source project glTrace [2] as a starting point for our intercept software.

```
glColor3f(float r, float g, float b)
{
   r=0.0f;
   g=0.0f;
   b=1.0f; //make all colors blue

   real_glColor3f(r,g,b);
}
```

Figure 1: Code example showing intercepted function.

3.1 In-And-Out Technique

The in-and-out technique, while not new, is quite effective at enabling closed source applications for VR. In-and-out uses a network connection to transmit interaction data from the VR devices into the host application and then relies on an OpenGL intercept to transmit the host's graphics out to a VR display (see Figure 2). This technique basically allows a closed-source application to be modified for a VR system without needing access to the host's underlying source. The major limitation of this technique is that the host application must allow for the creation of a network server and for information received by the server to be used as input data. Such functionality is normally provided through an additional plugin, an advanced feature, or a simple programming language offered by the host application. MotionBuilder, MATLAB, and Unity3D [15] are examples of applications that offer such functionality.

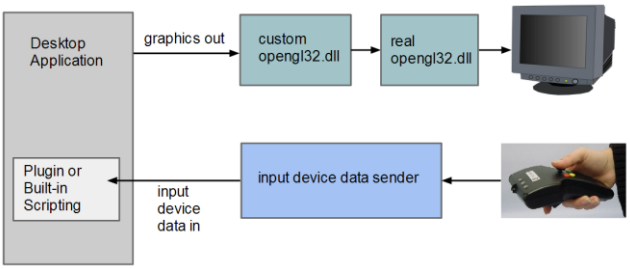

Figure 2: Diagram for In-And-Out technique. On the top, graphics data goes out of the host application. On the bottom, event data comes into the application.

3.2 Intercept Tag Technique

The intercept tag technique uses predefined OpenGL geometry calls that are added to the scene within the original application and then intercepted and interpreted by the intercept software as scene information, cues, or function calls, instead of being rendered [16]. This requires the ability to directly or indirectly make OpenGL geometry calls within the host application. Because intercept tags are interpreted and not rendered when decoded by the intercept driver, the OpenGL geometry chosen to define a tag must be carefully chosen. In practice, we have chosen geometries that would never appear in our intercepted applications, such as an empty polygon (e.g., a triangle with all three vertices at (0, 0, 0)). We can also vary the specific values of the points in the empty polygon to encode different types of tags (e.g., (1, 1, 1)).

We normally use intercept tags in pairs, like HTML or XML tags, with a start tag and an end tag enclosing a block of OpenGL calls. When the first tag is decoded, it provides additional information about the upcoming OpenGL calls or signifies that they should be specially handled. When the second tag is encountered, it is interpreted as the end of the information or special handling. Like XML, intercept tags can be nested. This requires that the order in which the objects are drawn can be controlled within the host application.

We originally developed intercept tags in an attempt to reduce the latency of interacting with a MATLAB-based robotics simulation that was intercepted and displayed in the DiVE. In our original case, the MATLAB script received the event data and displayed the results of the simulation using the in-and-out technique. However, this meant that the virtual hand interactions happened at the frame rate of the simulation. This low frame rate caused noticeable latency while attempting to manipulate objects in the scene. Thus, the idea of utilizing intercept tags to hand off control of certain latency-critical interactions to the end VR application was developed.

We have identified three potential uses of intercept tags: hand-off techniques, display techniques, and visual enhancements. Hand-off techniques use intercept tags to signal the use of a particular interaction technique, such as a virtual hand technique for manipulations or a slice plane technique for viewing the structure and internal volume of geometries. Display techniques specify how and when parts of the OpenGL scene should be displayed through the concepts of display lists and levels of detail. Finally, intercept tags can be used for visual enhancements, such as interpolated animations and advanced shaders.

3.3 Driver-Mediated View Technique

The in-and-out technique works only in applications that support network connections and manipulation of the view. It also requires extra effort for the target application user, as the network connections must be specified and the input data must be processed. To address the limitations of the in-and-out technique, we propose the driver-mediated view technique. In this technique, we utilize the VR input data within the intercept driver to manipulate the current view of the host's generated graphics (Figure 4). The simplest example of this technique is to implement head tracking inside the OpenGL intercept driver. To accomplish this, we monitor glMatrixMode to determine when we are in model view mode. We then wait for a call to glLoadMatrixd and replace the passed-in model-view matrix with a matrix that we create based on the VR input device tracking data.

In addition to using the driver-mediated view technique for physical view manipulations, such as head tracking, it can also be used for virtual locomotion. For example, a steering technique **Error! Reference source not found.** can be implemented using the data from a 6-DOF wand and updating the model view matrix.

The driver-mediated view technique has some limitations. For example, frustum culling needs to be disabled, as camera views are different between the host application and the final view. Additionally, special intercept code needs to be written to handle certain effects (e.g. shadows) that are natively enabled in the application. An incidental benefit from the driver-mediated view technique is that lower head tracking latency can be achieved, as there is no need for network communication between the VR device and the application.

4 Case Studies

In order to demonstrate the usefulness of OpenGL intercept-based techniques for enabling features for VR, we conducted two case studies in two widely used closed-source desktop applications. In the first case study, we present the use of intercept techniques to enable MotionBuilder to be visualized with an Oculus Rift head-mounted display. The second case study explains our work on enabling MATLAB for the DiVE.

4.1 MotionBuilder

MotionBuilder is an advanced motion capture and animation software by Autodesk. This software is commonly used for motion capture and animation as part of a larger workflow, perhaps ending with the use of the animations in a separate game engine. However, MotionBuilder can also be used as an end-user application to create interactive experiences via its constraints and relations system, python scripting, and a C++ software development kit (SDK). This is especially useful, as it is now possible to use the high-quality real-time inverse kinematics, to modify character animations directly in a simulation. We have been using MotionBuilder for desktop visualizations during brain-computer interface experiments [13]. Here, we describe the process to enable such experiments in the Oculus Rift.

4.1.1 Brain Computer Interface

In order to provide input to a virtual scene, various tracking systems, game pads, and body gestures can be used. An interesting method of input is known as Brain Computer Interface (BCI). These are methods of real-time decoding of brain activity and can be classified as either invasive (using implanted electrodes) [10] or non-invasive (using sensors placed on top of the head) [13]. We believe that VR can be used to train subjects with reduced mobility to utilize a BCI to control physical devices in a safe environment (e.g., wheelchairs [6], exoskeletons).

4.1.2 Oculus Rift

The Oculus Rift is a low-cost head-mounted display (HMD) [11]. The current development version has a large 110-degree diagonal field of view (FOV). This high FOV is obtained by using special optics, which cause pincushion distortion. In order to remove the distortion, we need to apply a post-processing barrel distortion to the image, which is not currently available in Motion Builder. We can also get head tracking data from the Oculus. We will later discuss how we have integrated this tracking data into Motion Builder.

4.1.3 Rendering

Typically, the intercept code will pass on the function calls to the real DLL. However, several functions need to have additional functionality added. The first step to implement the intercept techniques for rendering is to conduct an analysis of the OpenGL output of the program. For that, we utilized the open-source program glIntercept.

We have found that the analysis of the existing frame format is one of the most important and challenging aspects to intercept-based projects. It is easy to override every instance of a command (e.g. change all colors to blue), but often we only want to override a specific command in a specific location in the frame.

We conducted an analysis of the output of the "parallel view" or side-by-side mode, and we found that MotionBuilder already renders a view for each eye to a texture. This is exactly the setup needed to further apply the post-processing barrel distortion. In order for everything to look correct, the correct projection and model matrices need to be set, and the shaders need to be activated at the appropriate time.

In this side-by-side viewing mode of MotionBuilder, the image does not extend to the top and bottom of the screen. Through analysis of the glViewport function, we determined that this was easily fixed by overriding the values on the sixth and seventh glViewport function calls.

Next, we look at the glLoadMatrixd function. This is the function that is used to specify the projection matrix and also the model view matrix, depending on the value from glMatrixMode. The projection matrix encodes things like the field of view, aspect ratio, and clipping planes. Through analysis of the OpenGL frame, we determined that we needed to override the values that occurred during the first and third viewports, which contain the original rendering of the left and right images, respectively. We use the calculations from the Oculus Rift SDK to compute the projection matrices, and override the passed-in values to glLoadMatrixd.

Finally, we activate the post-processing distortion shaders on the sixth and seventh time glViewport is called. We used the shader code from the Oculus Rift SDK, and modified them slightly, as MotionBuilder writes the left and right images to separate off-screen textures and the Oculus Rift examples are setup to use a single image.

4.1.4 In-And-Out Technique

The Oculus currently has an internal sensor that can provide the head orientation of the user. We can capture and send the data to MotionBuilder and let it control the camera orientation. To achieve this, we created a standalone UDP sender. We then wrote a UDP receiver utilizing the MotionBuilder Plugin SDK. This allowed us to send UDP data into MotionBuilder from an external program. We utilized this custom plugin, along with our standalone program, which obtains tracking data from the Oculus Rift. MotionBuilder wants the Euler angles (0,0,0) to map to a positive X orientation, so we needed to change the order of our axes to Z,Y,-X. We then had to determine the correct order for the Euler angles, which we found to be Z,Y,X. Finally, we utilized the

constraint system of MotionBuilder to have the received data control the camera (see Figure 3).

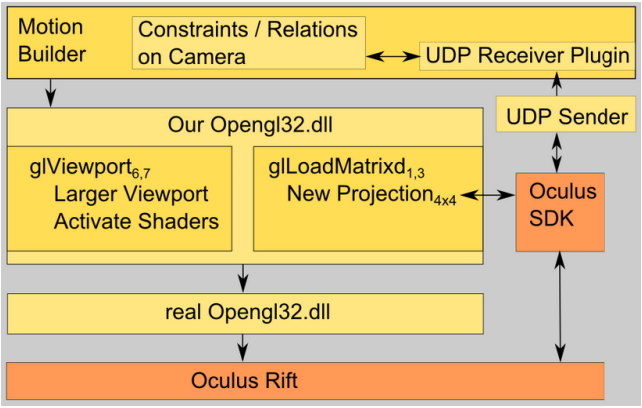

Figure 3: MotionBuilder distortion shader insertion and view control via the In-And-Out technique.

4.1.5 Driver-Mediated View Technique

The in-and-out technique worked, but required extra steps for the MotionBuilder user. We realized that we could implement head tracking inside the OpenGL driver (see Figure 4). When a glLoadMatrixd is called while in model view mode, we query the data from the Oculus Rift, which gives us back an orientation (but not currently a position with the first development kit). We then do an inverse of the passed-in matrix to get the existing camera position. We can then add in our orientation from the Rift. We use this new value to call the real glLoadMatrixd. This allows us to have simple head tracking, without having to modify the MotionBuilder projects.

However, it does lead to some issues. We need to disable frustum culling, since the camera views are now different between MotionBuilder and the final view. This is simple to do, as we only need to change a checkbox in the MotionBuilder profiler screen. Another issue we had was that the "live shadow" effect in MotionBuilder was now no longer working. We will soon discuss how we addressed that issue. Finally, one potential benefit of the driver-mediated technique is that we felt that we achieved lower head tracking latency via this method, which could be critical for reducing simulator sickness.

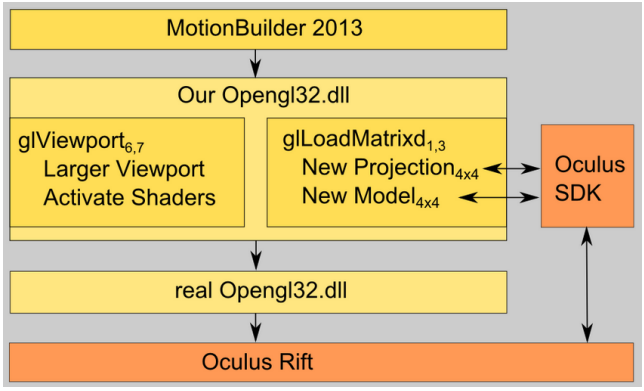

Figure 4: System diagram for MotionBuilder shader insertion and driver-mediated head tracking.

4.1.6 Issues: Clipping Planes, Anti-Aliasing, Shadows

By overriding the projection matrixes and the model-view matrices we introduce several problems. Our first problem is that our clipping plane values are hardcoded into our software. The correct behavior would be to get the values from MotionBuilder. This is important, because in a large environment, we want to move the far clipping plane back, so that not all of our objects get culled and vanish from the scene. In a smaller environment, we want to bring the clipping plane forward, so that we obtain higher accuracy in terms of the depth tests. We can see that we ideally have different clipping plane values in different simulations. So, we can analyze the incoming projection matrix before we discard it. As pictured in Figure 5, we can analyze the values of A and B. Based on the MotionBuilder documentation, we learned that the near clip is B/(A-1) and that the far clipping plane is B/(A+1). We can now feed those values into the projection matrix generation function in the oculus rift SDK.

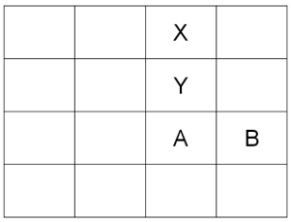

Figure 5: Key positions in the projection matrix. X and Y facilitate anti-aliasing, and A and B denote clipping plane information.

The next issue we addressed was that anti-aliasing was now broken in our system. Anti-aliasing is a technique where multiple copies of the scene are rendered and combined, in order to smooth out the jagged edges on objects. From analyzing the output of MotionBuilder, we discovered that the shift in the camera position was actually not in the model view matrix, but actually in the projection matrix. We determined that the offset of the camera for the multiple renders of the scene was stored in the projection matrix in positions X and Y as illustrated in Figure 5. So once again, we analyze the incoming projection matrix from MotionBuilder and store the values. We then use these values by adding them into the projection matrix that is generated from our calls to the Oculus Rift SDK.

Finally, we addressed the problem that our shadows were now broken. We carefully analyzed the OpenGL output of MotionBuilder and discovered that during the section of code that dealt with the shadows, a glMultMatrixd call was applied to the model-view matrix instead of glLoadMatrixd, which is used everywhere else in the code. Our solution was to examine calls to glMultMatrixd and see if the parameter being passed in was the original model-view matrix. If they are equal, we substitute in our new model-view matrix. This resolved our issues, and shadows are now working.

4.1.7 Discussion

We are currently working on acquiring a high-speed camera to enable us to have quantitative data about the latency differences between the in-and-out and driver-mediated techniques. Perhaps driver-mediated head tracking could be utilized to benefit other VR applications. This project was deemed successful, as we can now utilize MotionBuilder with the Oculus Rift for BCI experiments (see Figure 6) with no introduced artifacts.

Figure 6: User utilizing Oculus Rift with MotionBuilder.

4.2 MATLAB

MATLAB is a commercial software product made by MathWorks, Inc. It is often utilized by engineers for calculations, modeling, and visualization. Our collaborators were modeling robot path-planning algorithms on their desktop systems utilizing MATLAB. They desired a way to easily use the DiVE, which has a cluster-based architecture (e.g. one computer per screen). We decided to develop software and utilize a preexisting VR software library, Syzygy [12], to provide the features of cluster-based rendering and access to VR input devices. We called our software ML2VR [17].

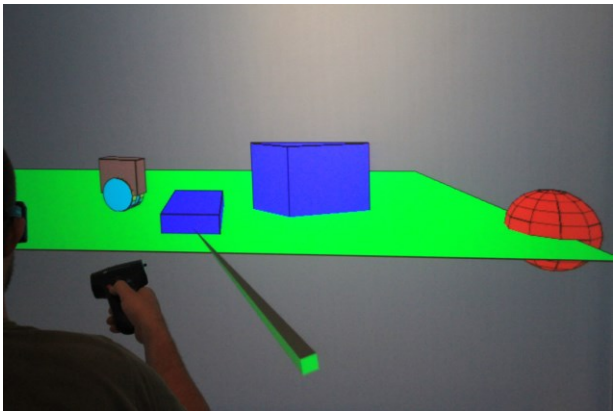

Figure 7: User interacts with MATLAB robot simulation in VR.

4.2.1 Distributed Rendering

We utilized the OpenGL intercept technique and first analyzed the MATLAB frame of OpenGL commands. We discovered the geometry we wanted to capture was located after the matrix mode had been switched into model view. Thus, in our intercept code, we added code to send the results of the most necessary commands (glColor, glVertex, glBegin) over to each Syzygy node. After receiving and storing the data, a ready signal is sent from each node to the "Swap Manager", which is located on the master Syzygy node (see Figure 8). The Swap Manager then utilizes Syzygy's variable distribution method to distribute the ready signal. All nodes thus swap to a new content frame at the same time, by watching the ready signal.

Syzygy automatically updates the head position during each frame. Our content swapping is synchronized among the nodes, yet asynchronous to the main render loop. Thus, we obtained a high frame rate when the user walks around the model, even if the MATLAB content is arriving at a low rate. This is especially useful when dealing with larger scenes or more complicated MATLAB simulations.

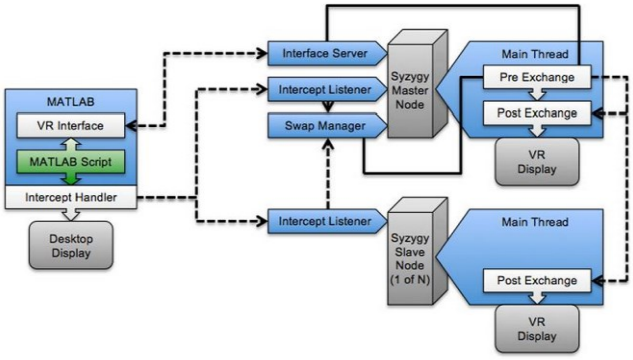

Figure 8: System architecture of ML2VR utilizing In-And-Out.

4.2.2 In-And-Out Technique

For ML2VR, we decided to allow the MATLAB programmer to easily access the VR input devices supported by the Syzygy framework. This access was provided via the "Interface Server" and a MATLAB script that we called the "VR Interface". The MATLAB user first initializes the VR Interface (by calling the function in the provided MATLAB script file). Once connected, they can query the VR Interface to determine if any button events (i.e., presses and releases) have occurred. Every event also contains the position and orientation of the wand device when that event occurred. By tracking the state of the wand when they occur, events provide more-accurate interactions than simple polling since the current wand position usually has changed since the button press. We also supported simple polling operations.

4.2.3 Intercept Tags

Our next work involved the development of intercept tags, which are predefined geometry that are interpreted instead of rendered. Our efforts focused on improving the interaction speed of the virtual hand technique. The virtual hand technique essentially consists of three phases: selection of the desired object, manipulation of its position and orientation, and object release. In our MATLAB-based simulation, selection and release of the object functioned well thanks to the event-based input data available from the wand. But, due to the slow frame rate of the simulation, manipulation of the object's position and orientation was noticeably affected by latency.

We have thus used intercept tags to hand off the manipulation of the object from our MATLAB simulation to the DiVE application intercepting the OpenGL calls. Once the simulation determines that an object is selected, a pair of intercept tags is used to enclose the OpenGL geometry of that object (See figure 9). Upon intercepting the first handoff tag, the DiVE application uses a glPushMatrix and an appropriate glMultMatrix command to translate and rotate the upcoming geometry relative to the wand's current position. Once the second hand-off tag is intercepted, the DiVE application uses glPopMatrix to leave the remaining scene geometry unchanged. When the object is released, the simulation removes the intercept tags, ending hand-off manipulation. While the object has been handed off, the MATLAB simulation can still update some properties of the object (e.g. color, size, shape). Thus we can manipulate the object at the frame rate of the VR application/viewer, rather than at the frame rate of the MATLAB simulation.

We conducted a user study to evaluate the differences between the original in-and-out technique, and the new hand-off dragging acceleration. The task for the user study was to place a cube inside a slightly larger wireframe cube. We found that our new technique was significantly faster. It also caused significantly fewer clutches (i.e., the user releasing and picking up the object again). From our analysis of the questionnaires, we found significantly more usability, more presence, and less simulator sickness using the intercept tags. For further details, see [16].

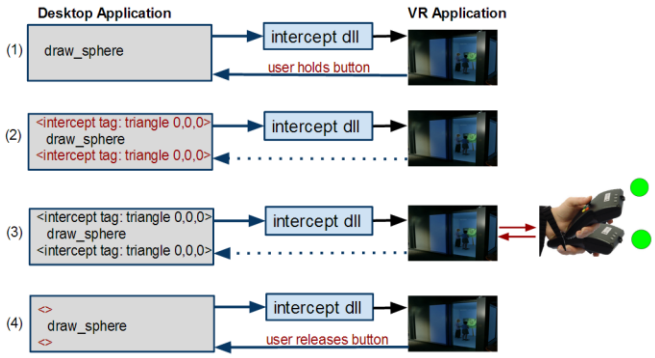

Figure 9: Hand-off technique for virtual hand acceleration. (1) User presses and holds button on the wand. (2) The desktop software (e.g. MATLAB) receives button event and adds intercept tags around object of interest. (3) VR application facilitates low latency manipulation of the object. (4) Release occurs and tags removed.

4.2.4 Discussion

Our work allows the MATLAB user to construct scripts as a loop: query user activity, update the simulation, and then render the resulting scene. This project was deemed a success, as our robotics collaborators can now use the DiVE for their existing and future MATLAB projects (Figure 7).

5 Conclusion

In this paper, we have described in detail how to utilize the OpenGL intercept technique to add VR capabilities to popular closed-source applications. We discussed the issues and successes with this technique in two case study examples. In our most recent work, we enabled MotionBuilder to utilize the Oculus Rift, by inserting the proper distortion shaders. We discussed how the frame format can be tricky to analyze, and thus our code to intercept is often custom tailored to specific programs, or even specific modes of the program. During development, we also explored moving the head tracking from the application, to inside the driver, which we call driver-mediated view technique, for potentially lower latency head tracking independent from the host application. Finally, we reviewed our successful work enabling MATLAB for the DiVE, allowing the user to easily access 6-DOF input device data, and enabling accelerated interactions via intercept tags.

References

[1] Bowman, Doug A., Ernst Kruijff, Joseph J. LaViola Jr, and Ivan Poupyrev. 3D user interfaces: theory and practice. Addison-Wesley, 2004.

[2] glTrace. http://hawksoft.com/gltrace/

[3] Höllerer, Tobias, JoAnn Kuchera-Morin, and Xavier Amatriain. "The allosphere: a large-scale immersive surround-view instrument." In Proceedings of the 2007 workshop on Emerging displays technologies: images and beyond: the future of displays and interacton, p. 3. ACM, 2007.

[4] Humphreys, Greg, 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 Transactions on Graphics (TOG), vol. 21, no. 3, pp. 693-702. ACM, 2002.

[5] Humphreys, Greg, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. "WireGL: a scalable graphics system for clusters." In SIGGRAPH, vol. 1, pp. 129-140. 2001.

[6] Leeb, Robert, Doron Friedman, Gernot R. Müller-Putz, Reinhold Scherer, Mel Slater, and Gert Pfurtscheller. "Self-paced (asynchronous) BCI control of a wheelchair in virtual environments: a case study with a tetraplegic." Computational intelligence and neuroscience 2007 (2007).

[7] Mohr, Alex, and Michael Gleicher. "HijackGL: reconstructing from streams for stylized rendering." In Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, pp. 13-ff. ACM, 2002.

[8] Neider, Jackie, Tom Davis, and Mason Woo, eds. OpenGL. Programming guide. Addison-Wesley, 1997.

[9] Ni, Tao, Greg S. Schmidt, Oliver G. Staadt, Mark A. Livingston, Robert Ball, and Richard May. "A survey of large high-resolution display technologies, techniques, and applications." In Virtual Reality Conference, 2006, pp. 223-236. IEEE, 2006.

[10] O'Doherty, Joseph E., Mikhail A. Lebedev, Peter J. Ifft, Katie Z. Zhuang, Solaiman Shokur, Hannes Bleuler, and Miguel AL Nicolelis. "Active tactile exploration using a brain-machine-brain interface." Nature 479, no. 7372 (2011): 228-231.

[11] Oculus Rift. http://oculusvr.com

[12] Schaeffer, Benjamin, and Camille Goudeseune. "Syzygy: native PC cluster VR." In Virtual Reality, 2003. Proceedings. IEEE, pp. 15-22. IEEE, 2003.

[13] Shokur, Solaiman, Joseph E. O'Doherty, Jesse A. Winans, Hannes Bleuler, Mikhail A. Lebedev, and Miguel AL Nicolelis. "Expanding the primate body schema in sensorimotor cortex by virtual touches of an avatar." Proceedings of the National Academy of Sciences 110, no. 37 (2013): 15121-15126.

[14] Techviz. http://www.techviz.net/

[15] Unity3D. http://unity3d.com/

[16] Zielinski, David J., Regis Kopper, Ryan P. McMahan, Wenjie Lu, and Silvia Ferrari. "Intercept tags: enhancing intercept-based systems." In Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology, pp. 263-266. ACM, 2013.

[17] Zielinski, David J., Ryan P. McMahan, Wenjie Lu, and Silvia Ferrari. "ML2VR: providing MATLAB users an easy transition to virtual reality and immersive interactivity." In Virtual Reality (VR), 2013 IEEE, pp. 83-84. IEEE, 2013.