ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

SCHOOL OF SCIENCE
Master degree in Computer Science

LAYING THE PATH TO CONSUMER-LEVEL IMMERSIVE SIMULATION ENVIRONMENT

Thesis supervisor:
Gustavo Marfia

Thesis co-supervisor:
Lorenzo Donatiello

Presented by:
Lorenzo Gasparini

Session II
Academic Year 2019/2020

Abstract

Since the end of the long winter of virtual reality (VR) at the beginning of the 2010 decade, many improvements have been made in terms of hardware technologies and software platforms performances and costs. Many expect such trend will continue, pushing the penetration rate of virtual reality headsets to skyrocket at some point in the future, just as mobile platforms did before. In the meantime, virtual reality is slowly transitioning from a specialized laboratory-only technology, to a consumer electronics appliance, opening interesting opportunities and challenges. In this transition, two interesting research questions amount to how 2D-based content and applications may benefit (or be hurt) by the adoption of 3D-based immersive environments and to how to proficiently support such integration. Acknowledging the relevance of the former, we here consider the latter question, focusing our attention on the diversified family of PC-based simulation tools and platforms. VR-based visualization is, in fact, widely understood and appreciated in the simulation arena, but mainly confined to high performance computing laboratories. Our contribution here aims at characterizing the simulation tools which could benefit from immersive interfaces, along with a general framework and a preliminary implementation which may be put to good use to support their transition from uniquely 2D to blended 2D/3D environments.

Contents

Chapter 1

Introduction

The slow, but steady increase, in terms of popularity and use of consumer-oriented virtual reality-based platforms has been propelled, at the time of writing, by the pandemic wave of 2020. Although the long-term thrust to VR provided by lock-down countermeasures may be uncertain to establish, it is possible to observe that a number of different academic, industrial and political domains are for the first time considering the use of such technology [1].

However, despite the contingency caused by COVID-19, many are questioning or even envisioning a future where 2D and 3D interfaces will both be part of digital entertainment and computing systems to provide better performances and experiences [2, 3, 4, 5, 6, 7, 8]. Clearly, the price drop of hardware components (e.g., Oculus Quest and Rift with prices below 500€ and Oculus GO below 200€) amounts to only one of the factors that may make all of this happen. As prices drop, progressively encouraging a mass-market diffusion of VR headsets, the integration of 2D/3D interface paradigms into software platforms proceeds more slowly, as it is not possible to observe a steep increase in the number of applications adding immersive experiences to their traditional 2D ones. Such resistance may be determined also by the fact that, to the best of our knowledge, no general and simple approach has so far been developed to implement an easy transition from 2D to 2D/3D settings.

Different solutions exist, as we will briefly describe in Chapter 2, but these are either custom or costly and not freely available to everyone. The risk, for the near future, is that hardware will really become cost-effective while a scarcity of software solutions will instead be available. In this work we resume such problem, which has been considered to some extent in the past years in literature. Previous works, however, have mainly concentrated on providing immersive interface support either through the provision of software platform specific add-ons or with the exhibition of dedicated APIs inside existing visualization frameworks [9, 10, 11, 12, 13]. No framework instead exists, to the best of our knowledge, capable of supporting: (a) a wide set of applications to (b) simply select the 2D interface elements which should be exported to a 3D immersive environment, while, (c) not requiring the creation of custom software, specific for the intended task.

We here aim at moving a step forward along the path set by (a), (b) and (c), considering an application domain that has for long experimented and appreciated the opportunities laid by 3D interfaces and virtual reality: scientific computation and simulation platforms [14, 15, 16]. An important body of evidence has demonstrated, in fact, the measurable benefits that can be attained when exploring scientific data using immersive interfaces [17, 18, 19, 20, 21, 22, 23, 24]. Even if difficult to quantify such phenomenon, any time scientists express situations where they discovered some relationship in their data while immersed in virtual reality systems, we can make the case that the interface provided a utility that helped them advance their work [13].

This work contributes to the research path presented so far with the provision of an analysis of the graphical libraries utilized by a few of the most widely used desktop computer simulation platforms, along with a preliminary implementation demonstrating the feasibility of the proposed technical approach.

This work has been presented at the IEEE ACM DS-RT 2020 conference [25].

Chapter 2

State of the art

The use of virtual reality approaches for the visualization of simulation results has been investigated for long now [26]. Scientific visualization, resorting to computer graphics, has been so far used to represent: (a) data sets, which may be the output of numerical simulations, (b) recorded data, or, (c) constructed shapes. Virtual reality has been widely explored to aid in the display of 3D structures providing spatial and depth cues, as it allows a rapid and intuitive exploration of the volume containing the data.

The authors of [27], for example, have analyzed the usefulness of VR in specific task performance with volume datasets, finding that such systems, with a high field of regard and head tracking, improve performance in spatial judgment tasks. More recently VR systems have been assessed in the visualization of complex weather-related information [28]: to this aim, the effectiveness and usability of the Xbox One controller in combination with a VR display proved to be the most effective. Reski and Alissandrakis conducted a user interaction study to investigate the experienced workload and perceived flow of interaction in the exploration of open data [29]: they also identified trends indicating a user preference for virtual representations.

Implementation-wise, virtual reality visualizations have, so far, been run according to one of the three following approaches:

- **Native virtual reality** - Such applications are designed and developed for the VR environment, therefore their user interface is designed in a three-dimensional way in the first place. Native approaches typically provide the most advantages is in terms of the degree of immersion and efficiency they are able to support;

- **Plugins for virtual reality** - The application is updated by the producers with the addition of a plugin that allows the user to interact with the software in VR (e.g. SketchUp or for AutoCAD) [9, 30]. The main disadvantage compared to the native VR approach is that the software is designed for 2D interactions in the first place, therefore, it may pay the price of a design that has not accounted for immersive environments since the beginning. However, since the vendors have access to the source code of their application, they can also extend their code-base to increase the immersion of the virtual reality interface (such changes may be expensive, though, if the application has already been released and has reached a considerable size);

- **Porting in virtual reality** - This is the situation where the application solely provides a 2D interface and its producers do not provide any type of plugin or update to support VR interactions. A solution has been found in the past, intercepting the calls made by the rendering library (e.g. DirectX, OpenGL) and redirecting them to an application that handles them, executing them in a virtual reality environment [31, 11, 12, 13]. Obviously this approach may not be as efficient as the previously described ones, but provide the means to flexibly port any type of application, without requiring an access to its source code [11]. In terms of immersion, the main disadvantage is that it may be lower when compared to the previous approaches. It may not be possible to achieve the same level of interaction, as the actions that may be implemented in a general way are limited to those involving the graphics components (i.e., the calls made by the rendering library), while direct interactions with the original application would require custom, perapplication solutions, which are ill suited in the context of a general framework.

Some companies, such as Techviz or Moreviz, offer porting frameworks based on OpenGL intercept mechanisms: both specialize in the porting of CAD applications [32, 33]. However, both solutions come at a cost, with required price expenditures exceeding those affordable to the consumer market or even, in many cases, to a research lab.

Another experience worth citing is the one of Envelop, a company that recently attempted to introduce a VR-based interface for the visualization and interaction with Microsoft Windows-based applications, its software provided a 3D view of the 2D Windows Desktop. The entrepreneurial initiative ended in 2017 before having any chance of getting mainstream [34].

Compared to the previous experiences described in this Chapter, we place our contribution at the intersection of scientific computing, consumer oriented solutions and virtual reality. With this work we aim at reviving the stream of work started in [11], providing the building blocks necessary to support an immersive view of 3D content produced in different scientific/simulation frameworks. Unlike [32, 33], however, our approach is an open source one. In the following, we describe the path that has been started, according to a vision which aims at providing an easy-to-use and integrable 3D immersive environments for legacy simulation frameworks.

Chapter 3

Implementation

This chapter begins with the description of the use case scenario proposed by us. Then is described the legacy ML2VR project which is the starting point of our solution and finally is described the architecture implemented to achieve the proposed scenario.

3.1 Use Case Scenario

The scenario of reference may be simple represented by Figure 3.1. In essence, we envision a situation where a 3D object produced by a scientific computing or simulation software may be explored and manipulated within an immersive environment.

In terms of visualization, the aim is to provide the standard 3D manipulation functionalities, typically available in immersive environments, such as rotation, scale, translation along the three different axes and zoom [35]. In addition to these, we aim at providing means of toggling on/off or highlighting specific 3D objects or classes of objects. Clearly, maintaining such approach, the possible interactions are those that may be implemented with the rendered graphics.

In a vehicular simulation scenario, for example, it may be possible to highlight (e.g., change the color) specific or groups of automobiles to track their behavior. In the same scenario, it may also be possible to toggle off all the vehicles that are moving, in order to focus on those that are not.

As anticipated, we aim at implementing the depicted scenario for as many as simulation platforms as possible, while not requiring the users of such platforms to write any lines of code. In the following we explain in detail how this may be done.

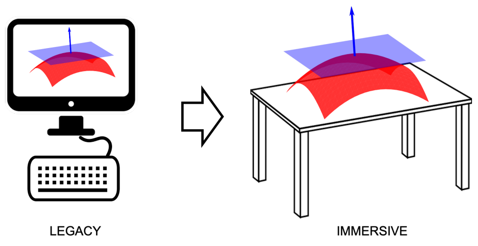


Figure 3.1: Reference scenario.

3.2 Legacy ML2VR

As described in the chapter 2, our aim is to revive the stream of work started in [11] publication. The goal of their project, called ML2VR, was to provide VR interaction capabilities to Matlab.

Since Matlab employs OpenGL as graphic library, the authors created an OpenGL library clone as a middle-ware layer between the real OpenGL library and the application. The role of this library was to redirect the calls made by Matlab to another application that handled the rendering in virtual reality.

Unfortunately the developed application no longer works due to the the lack of availability of most dependencies and to the change of the OpenGL library usage by Matlab. Matlab now uses `glDrawArrays` calls instead of single vertex and color calls and the original application does not intercept them.

For this reason, we started from the code they developed and our first goal was to make their project usable with the latest version of Matlab.

We decided to build our proposal upon this approach, rewriting every part of the software architecture to handle the latest Matlab version (v. R2019b).

3.3 Architecture

We've started by defining an OpenGL library which is injected into the Matlab process. This library intercepts and redirects some of the calls made by the main Matlab process to our immersive application. Then our application executes these OpenGL calls to render them into a virtual environment.

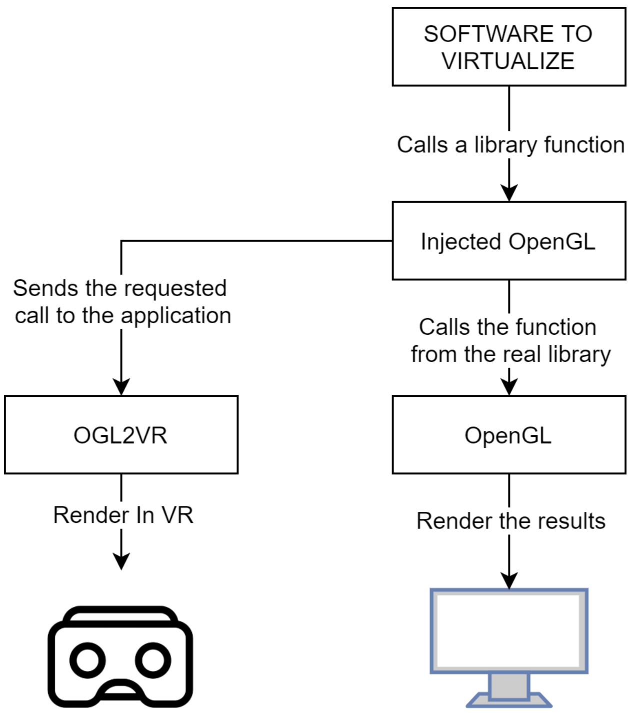


Figure 3.2: Complete architecture.

Figure 3.2 represents the structure of the software architecture. We can individuate two key components:

- **OGL2VR**: the application that handles the information received from the injected library, rendering it into a virtual reality environment.

- **Injected OpenGL**: this is the library that is injected into Matlab. This library redirects the relevant OpenGL calls to the OGL2VR application. To inject the library into the process is used a third application called Injector. This application requires the PID (process ID) of the process and handles the injection of the library in that process.

3.3.1 Sequence Diagrams

This section describes the architectural diagram sequences.

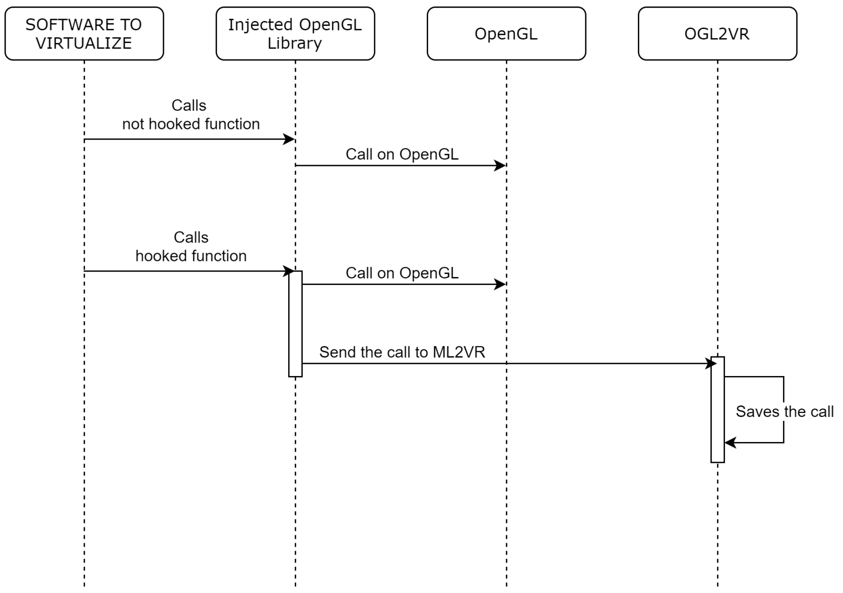


Figure 3.3: OpenGL calls flow.

The figure 3.3 shows the flow of the OpenGL calls through the architecture and can be described as follow:

1. The software to virtualize calls an OpenGL library
2. If the OpenGL call requested by the software is to be redirected it is sent to OGL2VR (our application) and then called on the real library, otherwise it is directly called on the real library.

3. OGL2VR saves each call and render them in VR. Once the call for the end of the frame is received, clear the list and start again.

This process is repeated until the user closes the OGL2VR application. If the software to be virtualized is closed, calls are no longer received but the last frame is still rendered as it is still present in the application.

We now proceed to describe the sequence diagram of the two key components.

OpenGL injected library

We start with the sequence diagram of the OpenGL injected library.

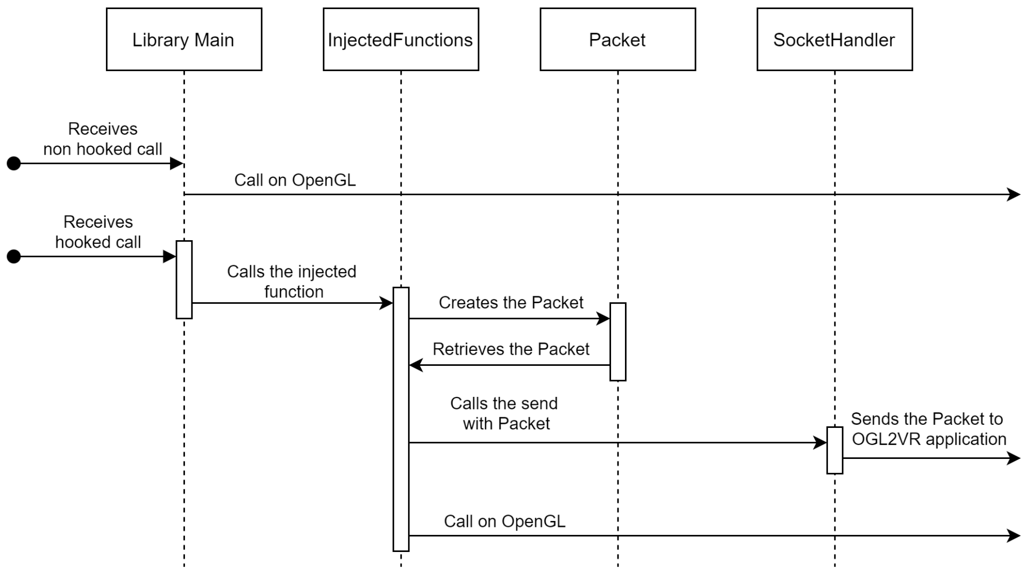


Figure 3.4: Sequence diagram of the Injected library.

The figure 3.4 shows the sequence of actions made by the injected library when the call is received from the main application. The sequence can be described as follow:

1. The main of the library receives the call from the main application
2. If the received OpenGL call is not hooked, it is redirected directly to the real library, otherwise the injected function relating to the received

call is called.

3. The function starts by creating the Packet using the information of the call. After that call the send function of the SocketHandler class which sends the Packet to the OGL2VR application. Finally calls the real function from the library.

4. The function starts by creating the Packet using the call parameters. Next, it calls the send function of the SocketHandler class which sends the packet to the OGL2VR application. Finally call the function from the real library.

OGL2VR

For the OGL2VR application, we have two main flows, one for receiving and parsing calls and one for rendering elements in VR.

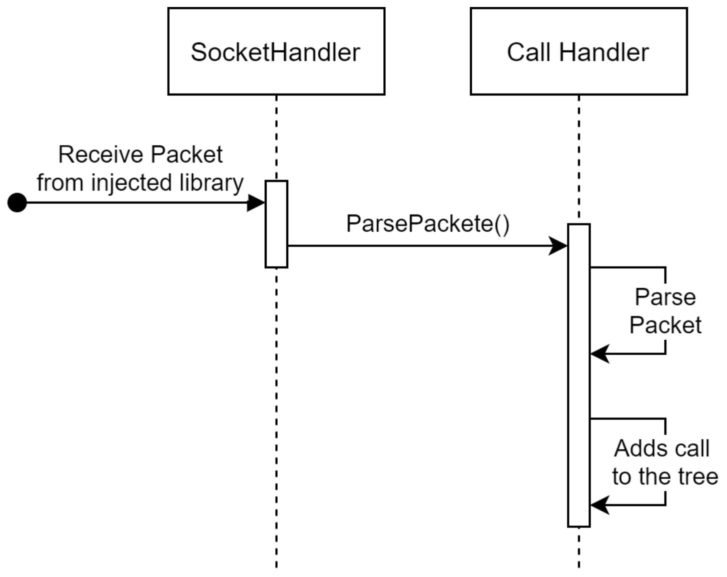


Figure 3.5: Sequence diagram of calls parsing.

The figure 3.5 shows the sequence of actions made by OGL2VR application when the packet is received from the injected library. The sequence can be described as follow:

1. The SocketHandler, the class that manage the communication, receives the packet from the library and calls the ParsePacket function passing it as a parameter.

2. The call handler receives the packet, parse it and add the call to the tree that contains all the calls to render.

The call handler is much more complex and will be described in detail later, but for now it is sufficient to understand how the packet receive works.

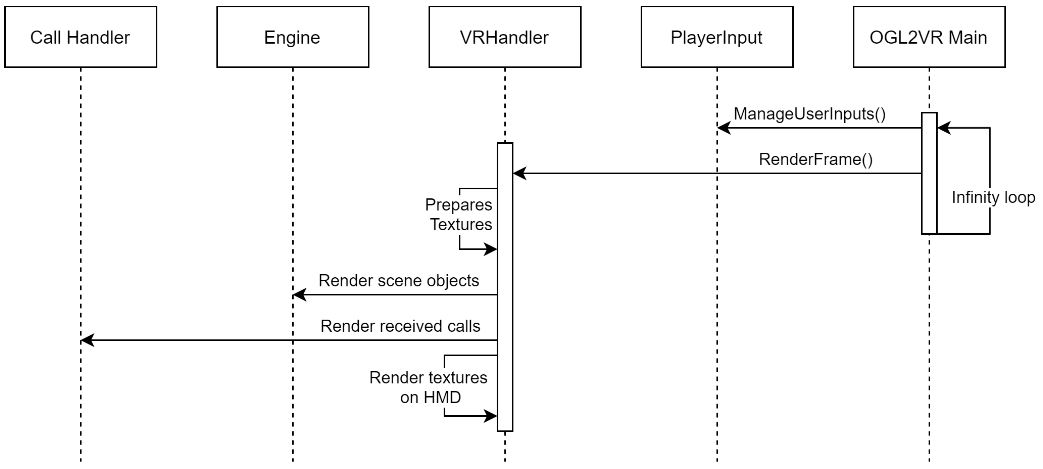


Figure 3.6: Sequence diagram of VR rendering.

The figure 3.6 shows the sequence of actions made by OGL2VR application for rendering all the elements and the calls in VR. The sequence can be described as follow:

1. The main of the OGL2VR application has a display function which handles the rendering of the scene. The function is looped and called several times at second.

2. Each cycle starts by calling the ManageUserInputs function to manage user inputs and apply changes to the rendered element.

3. After handling the inputs, VRHandler's RenderFrame function is called.

4. The RenderFrame function starts by preparing the textures for the render.
5. Then the engine rendering function is called which draws every object of the scene in the textures.
6. After that, each call received from the injected library is executed. In this way the elements to be virtualized are drawn in the textures.
7. Finally the textures are rendered in the head mount display (HMD).

Just like for the call handler, the Engine, VRHandler and PlayerInput blocks are more complex and will be described later. For now they are sufficient to understand the rendering process.

Chapter 4

OpenGL/Injector

This chapter describes the OpenGL library created for the injection and the injector used to inject it.

4.1 OpenGL injected library

At the beginning of the development, the aim of the project was the same as that of the old OGL2VR application. Improve Matlab with the addition of VR interaction features.

The library class architecture used at the time was based on three main classes that managed three different categories of calls: **vertex/color calls**, **drawVertex calls** and **default calls**. Each of the three classes had a unified function `SendValue` inside which was used to send values to the OGL2VR application. An example, the calls **glColor3f**, **glVertex2d** and so on, were classified in the vertex/color category and used the same `SendValue` function as they belonged to the same "category". This method worked well because the injected calls were easily classifiable and the actions performed by the `SendValue` function were the same for each category.

Everything worked well until we decided to switch to supporting any application that used OpenGL as a graphics library. The problem was that the amount of calls to be handled increased exponentially and the "generic"

`SendValue` functions were no longer so “generic”. Many calls, in fact, fell into the **default calls** category and the `SendValue` function used for that category became a big if-then statement (it this call send this, if this call send this and so on).

For this reason we decided to refactor the architecture of the OpenGL library.

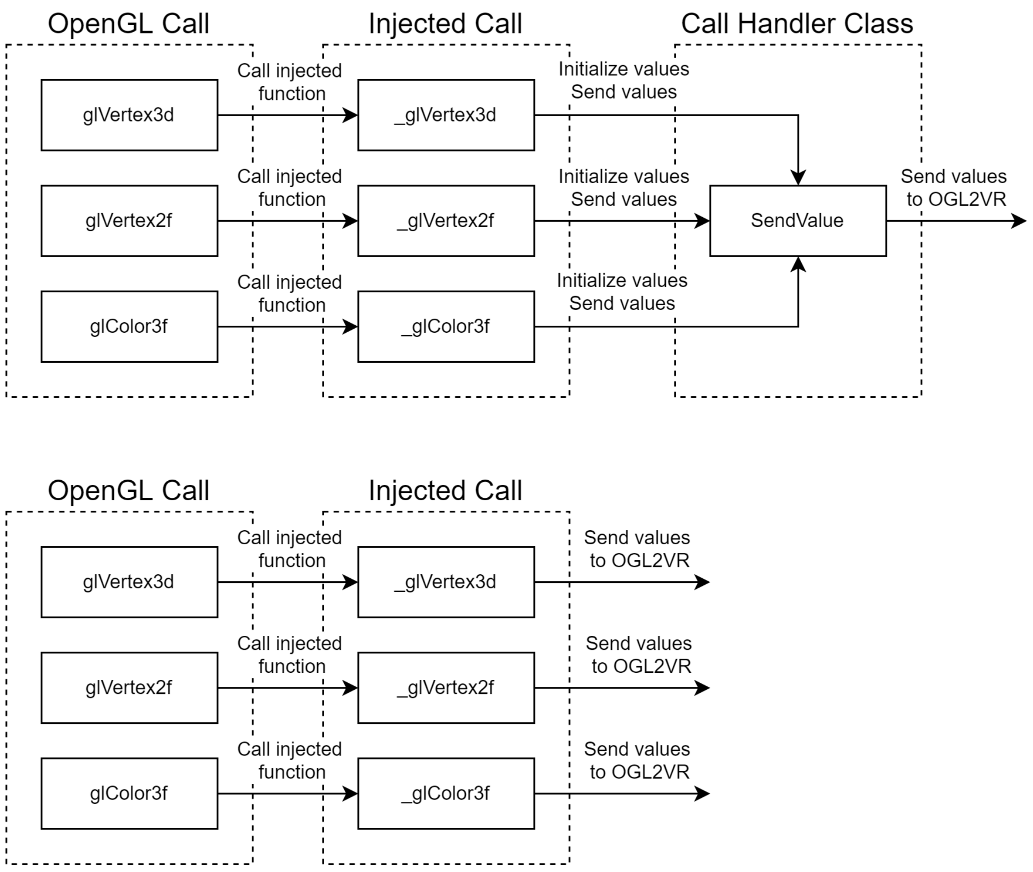


Figure 4.1: Old call management (above) — New call management (below).

The figure 4.1 shows the old method used and the new one. The upper image shows the old method in which all the injected calls belonging to the same category were managed by a single class which was in charge of sending the value. The image below shows the new method in which all injected calls autonomously manage the sending of information.

With this change we can get rid of the functions filled with the if-then statement and manage the sending within the injected function itself. Moreover, it is also better for the scaling of the library because if in the future we want to manage new calls we just have to add the injected function and manage the sending of the information inside it.

To improve modularity we have also added a new class to manage the construction of the packet to be sent through the socket. This class, called `Packet`, makes it easier for the user to create, manage and send packets via sockets.

The `Packet` class will be described in detail later, now we will talk about the final architecture of the OpenGL injected library.

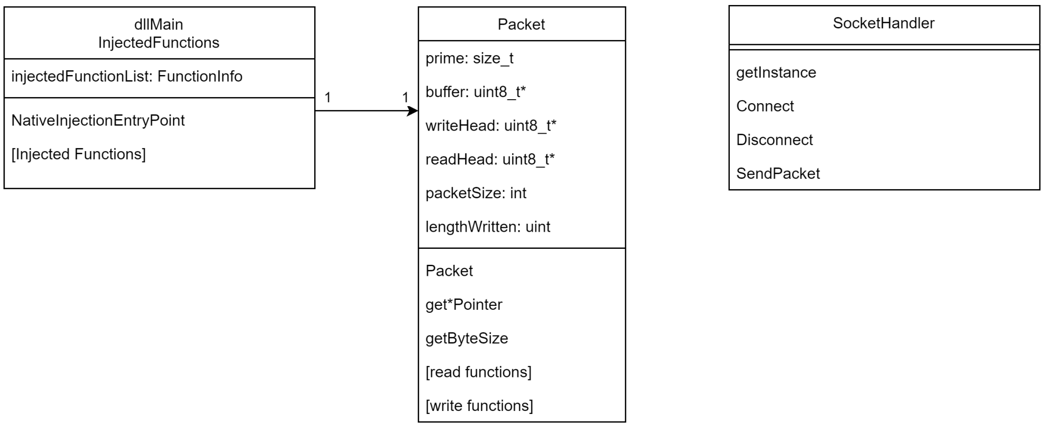


Figure 4.2: OpenGL injected library class diagram.

The figure 4.2 shows the class diagram of the injected library. Three main components can be identified from the image:

- **dllMain/InjectedFunctions** component, which is what manages the hooking of the functions and the execution of them.
- **Packet** component, which is the class that manages the creation of the packet to be sent through the socket.
- **SocketHandler** component, which is the class that manages socket communication with the OGL2VR application.

The interaction between them allows the application to send the intercepted calls to the application for virtualization.

In the next sections the three components will be described more in details.

4.1.1 dllMain/InjectedFunctions

In this section the dllMain/InjectedFunctions component will be described. This component is made of two subcomponents: the **dllMain** component and the **InjectedFunctions** component. These two components are deeply linked, the first contains the hooking of the OpenGL functions, the second contains the functions that are hooked.

Hooking

For the hooking of the functions is used EasyHook library which simplifies the entire injection/hooking process for the user. The library as a function called `LhInstallHook` which requires:

- The entry point of the function to override
- The pointer to the new function
- A pointer to a callback function which is not used by us
- A pointer to the trace of the hook

The pointer to the function to override is obtainable using the `GetProcAddress` function. `GetProcAddress` requires the library containing the function and the name of that function, the output is the pointer to that library function.

The pointer to the new function is simply the pointer to the function that should overwrite the library function.

To make the library scalable and avoid a wall of `LhInstallHook` functions we decided to add an array of `FunctionInfo` elements, called **injectedFunctionList**, to manage information about the hooked functions.`FunctionInfo` is a structure that contains the name of the function to be overwritten and a pointer to the function that should overwrite it.

The array is initialized as follow with the functions currently managed by our application.

```
FunctionInfo injectedFunctionList[..] = {
        { "glBegin", _glBegin },
        { "glEnd", _glEnd },
        { "glClear", _glClear },
    .
    .
    .
        { "glNormal3d", _glNormal3d },
        { "glNormal3f", _glNormal3f },
        { "glDrawArrays", _glDrawArrays }
};
```

The first element is the name of the function, the second element is the pointer to the new function. The example is a part of the complete list, currently the application manages 26 opengl functions.

Thanks to this implementation, the injection process simply cycles each element of the array and calls the `LhInstallHook` function on each of them.

Injected Functions

Each function injected is different from the others but they all follow the same pattern which is:

- Create the packet
- Send the packet using the SocketHandler
- Call the real function

All the packet sent by the injected functions follow a defined structure: the first element of the packet contains the name of the call (e.g. `gl_scale_d`, `gl_begin`, `gl_vertex_3d`) the rest contain the call parameters.

4.1.2 Packet

The Packet class is the one that manages the information to be sent to the OGL2VR application.

The Packet class has two working methods: creating the package from scratch or parsing an existing packet.

Packet from scratch

In the case of creating the packet from scratch, the constructor `Packet(int byteSize)` is used, which allows the user to define the maximum size of the packet.

This constructor initializes the pointers, sets the current size to 0 and allocates an byte array with size byteSize + 4. The size is increased by 4 because the packet follows the structure: `size_of_packet` + `data`. This structure simplifies the reading of the packet.

Through the write functions, the user can add data to the array. These functions write the data passed by the user to the array and update the head pointer and the total size.

Once the user has finished writing data to the array, he can retrieve the pointer to the packet using `getVoidPointer`, `getBytePointer` and `getCharPointer` functions. Each function, before returning the pointer, writes the final size of the packet at the top of the array (the first 4 bytes described above).

Packet parse

In the case of parsing an existing packet, the constructor `Packet(uint8_t* pointer, int size)` is used.

This constructor assigns the pointer passed as a parameter to the head and sets the maximum packet size using the size passed as a parameter. Through the read functions the user can retrieve all the information contained in the packet.

When reading the package, the hash of the package itself is also calculated. This is used by OGL2VR but will be described in detail later.

The writing and reading functions are very simple. They are based on reading/writing the number of bytes of the variable type (e.g. 4 bytes for the integers, 2 bytes for the shorts), except for reading or writing of bytes. In these cases, the size of the bytes to be read/written is retrieved/added first and then the bytes are read/written.

4.1.3 SocketHandler

SocketHandler is the class that manages socket communication with the OGL2VR application. It is based on the singleton design pattern.

The class is very simple. It has three functions which are:

- **Connect**, used to start the connection with the OGL2VR application
- **Disconnet**, used to close the connection
- **SendPacket**, which takes the packet as a parameter and calls the `send` function by passing the socket, the pointer to the packet and the size of the packet as parameters.

4.2 Injector

The injector is very simple. Asks the user the PID of the process and execute the injection of the library on that process. Just like the OpenGL library described above, it uses the EasyHook library to handle the injection.

The code is simple as well, it is composed only of the main function and follow these steps:

1. Start by asking the user for the PID and retrieving it from the console.
2. Then get the DLL to inject (our OpenGL library) that lays on its folder.

3. Finally, call the function for the injection and print the result to the console.

The function used for the injection is `RhInjectedLibrary`, this function requires many parameters such as processId, the path of the library to be injected and others that we do not use.

Chapter 5

OGL2VR

In this chapter, the OGL2VR application is discussed in detail.

We start by showing the overall class architecture of the application.

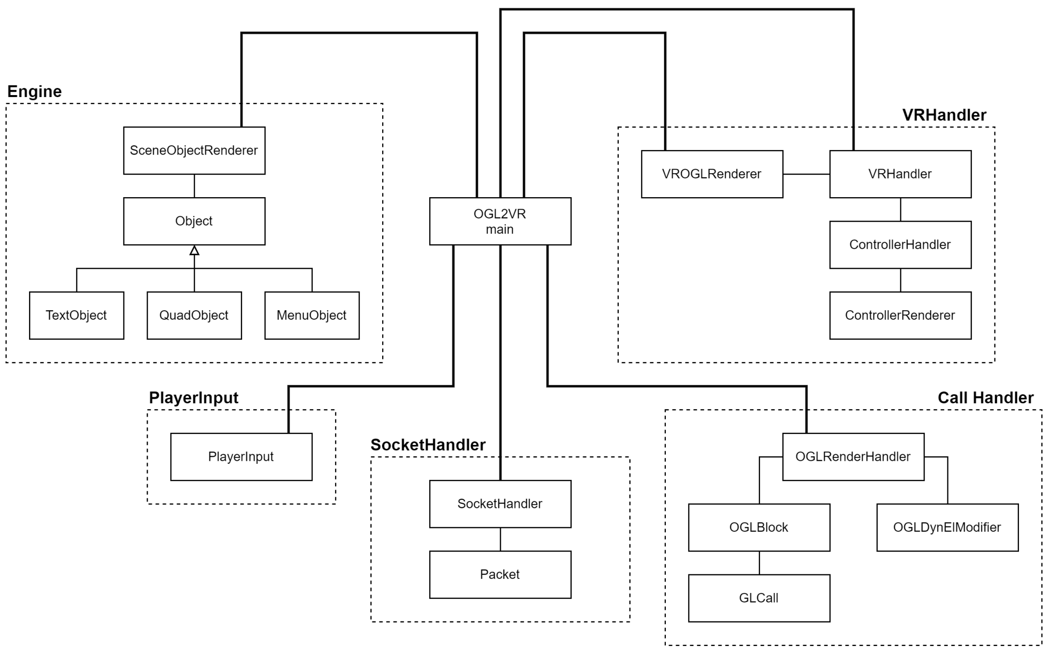


Figure 5.1: OGL2VR class diagram.

The figure 5.1 shows the class architecture of the 5 components seen above in the sequence diagrams [figure 3.5 and 3.6]. The figure does not show the interaction between each class but only the links with application's main.

The interactions will be described later during the detailed description of each component.

The task of the 5 components are:

- **SocketHandler** - management of communication with the injected library
- **Call handler** - management of the calls received from the injected library
- **VRHandler** - management of the rendering in the VR environment and handling of the controllers input (update of the state of the controller inputs)
- **PlayerInput** - management of the inputs received by the controllers
- **Engine** - management of the rendering of the objects. Those created by the user.

Each of the components will now be described in detail.

5.1 SocketHandler

The SocketHandler component consists of two classes: the **SocketHandler** class and the **Packet** class. While the latter is the same as the one described above for the OpenGL library [4.1.2], the former is much more complex.

The figure 5.2 shows the class diagram of the SocketHandler component. The **Packet** class is not described because is already described in the figure 4.2.

The `SocketHandler` class follows the singleton design pattern, contains 4 functions and a constructor: the constructor, `StartServer` and `StopServer` functions are trivial, so we can skip their description.

`SetCallsHandler` is used to set the `OGLRenderHandler` variable which is the handler for the calls received from the library but will be described in detail later.

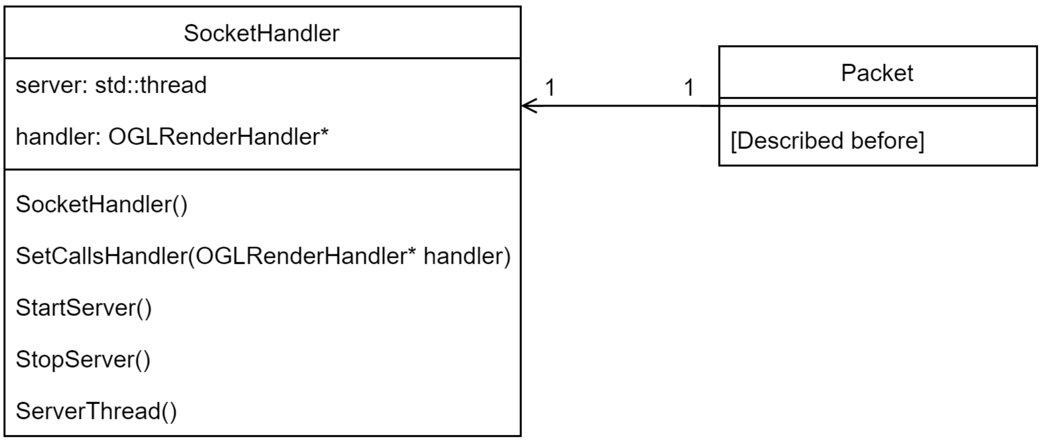


Figure 5.2: SocketHandler component class diagram.

The `ServerThread` function is the core of the `SocketHandler` class. This function handles socket communication with the library by reading packets received from the library and passing them to the packet parser (the `OGLRenderHandler` described before). This function is executed on a thread to decouple the reception of the packets from the main application.

The figure 5.3 shows the flow chart of the function.

1. Steps [1], [2] and [3] are those for initializing communication. The connection is accepted and the two main buffers are initialized. The byte buffer is the one that contains the raw bytes currently read, the dataRead is the one that contains the final packet sent through the socket and is filled during the loop.

2. After the initialization, a while loop begins and continues until an error occurs [4].

3. In step [5] the buffer is read and saved in the raw byte buffer.

4. Step [6] checks for errors in the reading. If less than 4 bytes are read and there are no previously read bytes, we are in the presence of an error so the communication is closed. Exactly 4 bytes are checked because the packet structure is `size_of_packet` + `data` (as described in section

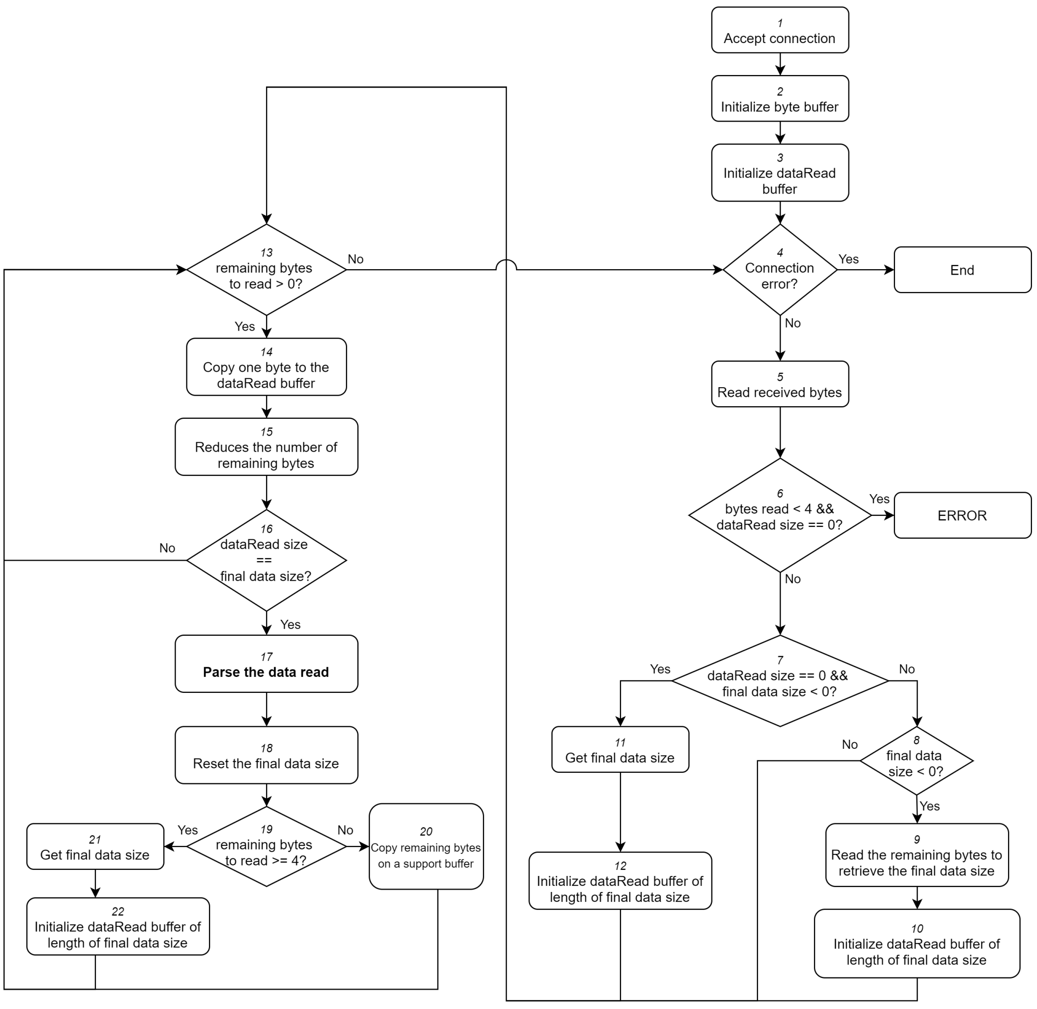

1
Accept connection
2
Initialize byte buffer
3
Initialize dataRead buffer
4
Connection error?
Yes
End
No
5
Read received bytes
6
bytes read < 4 && dataRead size == 0?
Yes
ERROR
No
7
dataRead size == 0 && final data size < 0?
Yes
No
8
final data size < 0?
No
Yes
9
Read the remaining bytes to retrieve the final data size
10
Initialize dataRead buffer of length of final data size
11
Get final data size
12
Initialize dataRead buffer of length of final data size
13
remaining bytes to read > 0?
No
Yes
14
Copy one byte to the dataRead buffer
15
Reduces the number of remaining bytes
16
dataRead size == final data size?
No
Yes
17
Parse the data read
18
Reset the final data size
19
remaining bytes to read >= 4?
Yes
No
20
Copy remaining bytes on a support buffer
21
Get final data size
22
Initialize dataRead buffer of length of final data size

Figure 5.3: ServerThread function flow chart.

[4.1.2]) and `size_of_packet` is represented as an integer, therefore 4 bytes.

5. Step [7] checks if the packet size is unknown (less than 0) and the dataRead buffer is currently empty.

   If true, it means that the last reading ended with the parsing of the packet, therefore the raw bytes read are from a new packet with the first 4 bytes containing the size. In this case is saved the final size of the packet [11] and the dataRead buffer is initialized of the length of final packet size [12].

If false is checked if the final packet size is known.

6. Step [8] checks if the final packet size is known. If true, it means that the last read finished reading a new packet but the bytes read were not enough to get the size, so the raw bytes read contain the remaining bytes to complete the size. The remaining bytes are read and the final size of the packet is saved [9], then, just like before, the dataRead buffer is initialized of the length of final packet size[10].

   If false, it means that we are currently reading a known packet.

7. After getting the packet information, there is a loop [13] that handles reading the raw bytes. This cycle continues until there are no more bytes to read.

8. In step [14] the current byte is read and in step [15] the number of remaining bytes is reduced.

9. Step [16] checks whether the length of the dataRead buffer (the bytes read) is the same as the size of the final packet. If true it means that the packet is completely received.

10. Step [17] parse the received packet by calling the `ParsePacket` function of the `OGLRenderHandler` class. Step [18] resets the size of the final packet and the dataRead buffer.

11. Step [19] checks whether the remaining bytes to be read are equal to or greater than 4. If true, the size of the new packet is saved [21] and the dataRead buffer is initialized as done above [22]. If false, the remaining bytes are saved in a buffer [20] and will be used during the next cycle to obtain the size of the new packet (step [9]).

5.2 Call handler

The call handler component consists of a group of classes that manage the parsing, storage, and rendering of calls received from the injected library.

Just like the injected OpenGL library [4.1], this component has been refactored due to the change in the project aim. Before the refactor took place, the structure used for call handling was similar to that used in the injected library.

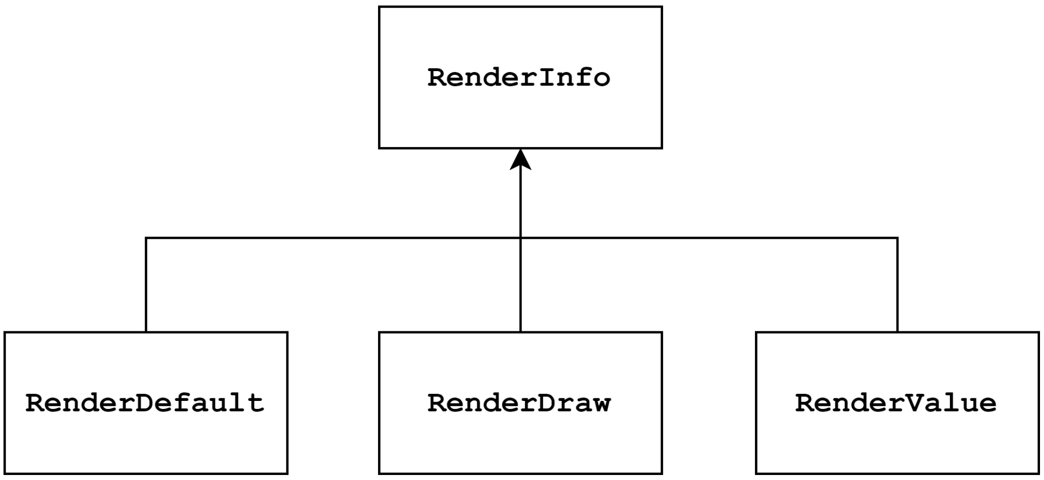


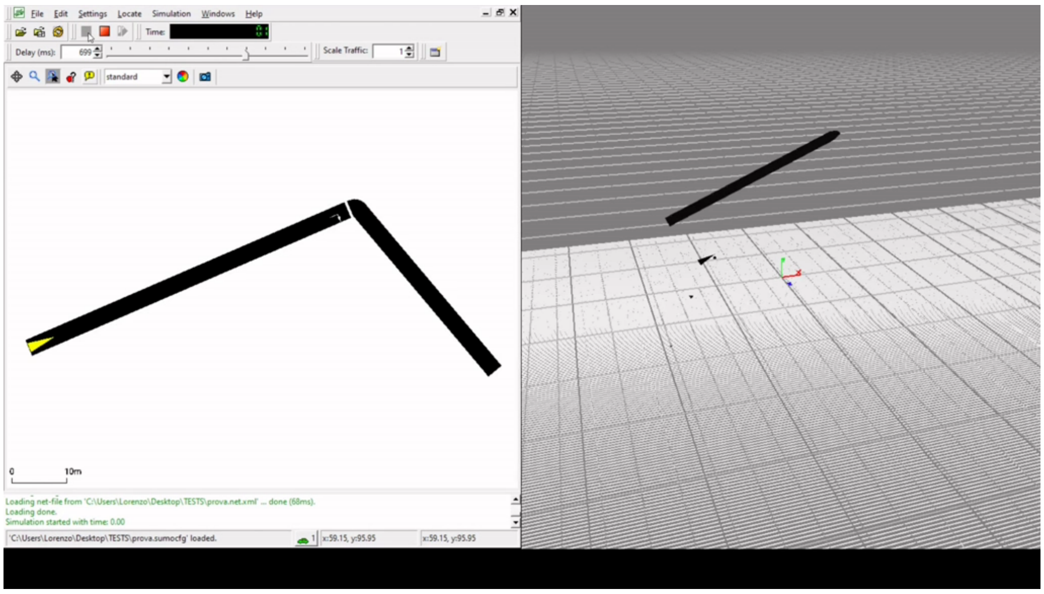
Figure 5.4: OGL2VR old call handler classes.

Figure 5.4 shows the three main classes that handled the calls. Each class handled a different category of calls: `RenderValue` for the **vertex/color calls**, `RenderDraw` for the **drawVertex calls** and the `RenderDefault` for the **default calls**. Each of the three classes was used to store information about received calls.

Within the **SocketHandler** class there were three different functions, one for each category of calls, used for message parsing. The operations performed by the three functions were: retrieve the information of the call within the message, analyze it in a variable with type one of the three classes described above, return the variable. All these variables were then added into a list of call.

The call rendering process was simply scrolling through the list and rendering each item.

This method worked fine for message parsing (up to aim change) but had problems for the rendering. Using a single call list resulted in the rendering of graphic artifacts during frame switching.



```
Loading net-file from 'C:\Users\Lorenzo\Desktop\TESTS\prova.net.xml' ... done (68ms).
Loading done.
Simulation started with time: 0.00
'C:\Users\Lorenzo\Desktop\TESTS\prova.sumocfg' loaded.   1   x:59.15, y:95.95   x:59.15, y:95.95
```

Figure 5.5: Graphic artifact, original figure on the left, rendered on the right.

Figure 5.5 shows an artifact created by using the list. In the picture on the right some parts of the original image are missing and there is a grid at the bottom that should not be rendered. This behavior was due to rendering when receiving the list. During the reception, the call list was not received completely and therefore not all the calls were present in it. Rendering an incomplete list resulted in a partial rendering of the original figure and some artifacts.

To solve this problem we have decided not to render the current unstable frame but to stay one frame behind the original application. The implementation is based on two lists, one containing the complete call list of the last frame and another containing the call list of the current frame of the original application. Once the frame is received, the lists are swapped.

Another improvement was the transition from using lists to using trees for saving calls.

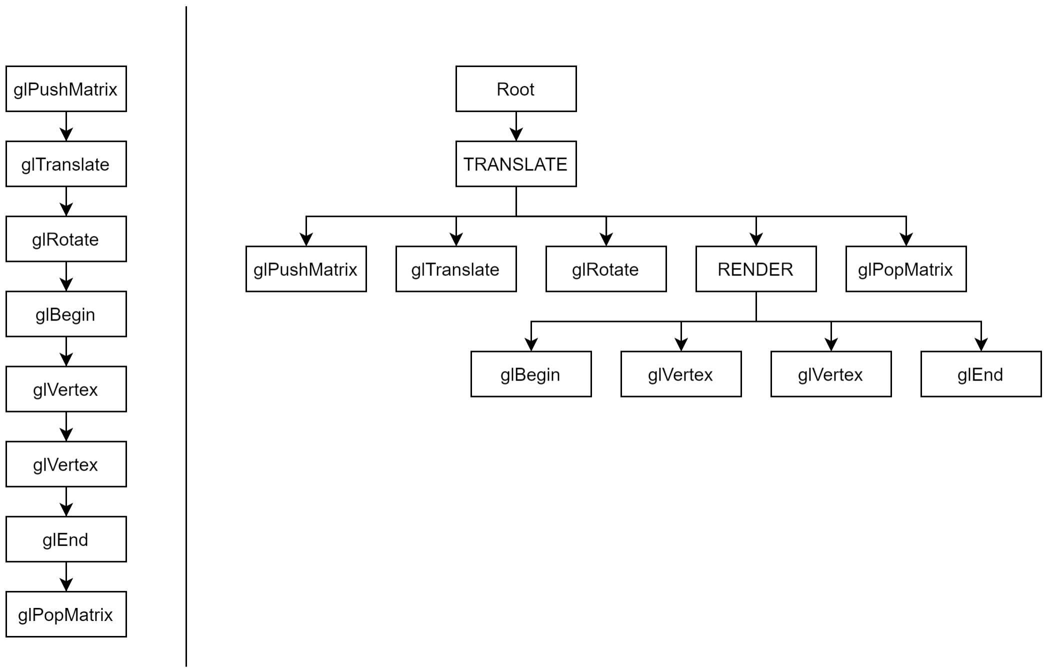


Figure 5.6: Transition from the call list to the call tree.

The figure 5.6 shows the transition from the call list to the call tree. On the left we have the old method used where each element in the list was an OpenGL call. This method was good enough for rendering calls but did not allow the user to easily interact with it: the user only saw a long list of calls and in case he wanted to disable rendering of an element he had to disable a large number of elements of the list.

On the right we have the new approach where calls are logically grouped into tree nodes. The list of calls shown in the figure is the same but we can immediately see that the tree is much more intuitive than the list. The internal nodes contain the grouping type, the leaves contain the call. In the example we have two internal nodes whose types are `TRANSLATE` and `RENDER`, the first indicates a call group that changes the position/rotation or scale of the object, the second indicates the call group which renders the object in the scene.

There are three types of nodes: `TRANSFORM`, `RENDER` and `CALL`. The first two have already been described above, the `CALL` nodes are the one containing the final call and are the leaf of the tree. Thanks to this approach, if the user wants to disable the rendering of an element, he can simply disable the relative `RENDER` node.

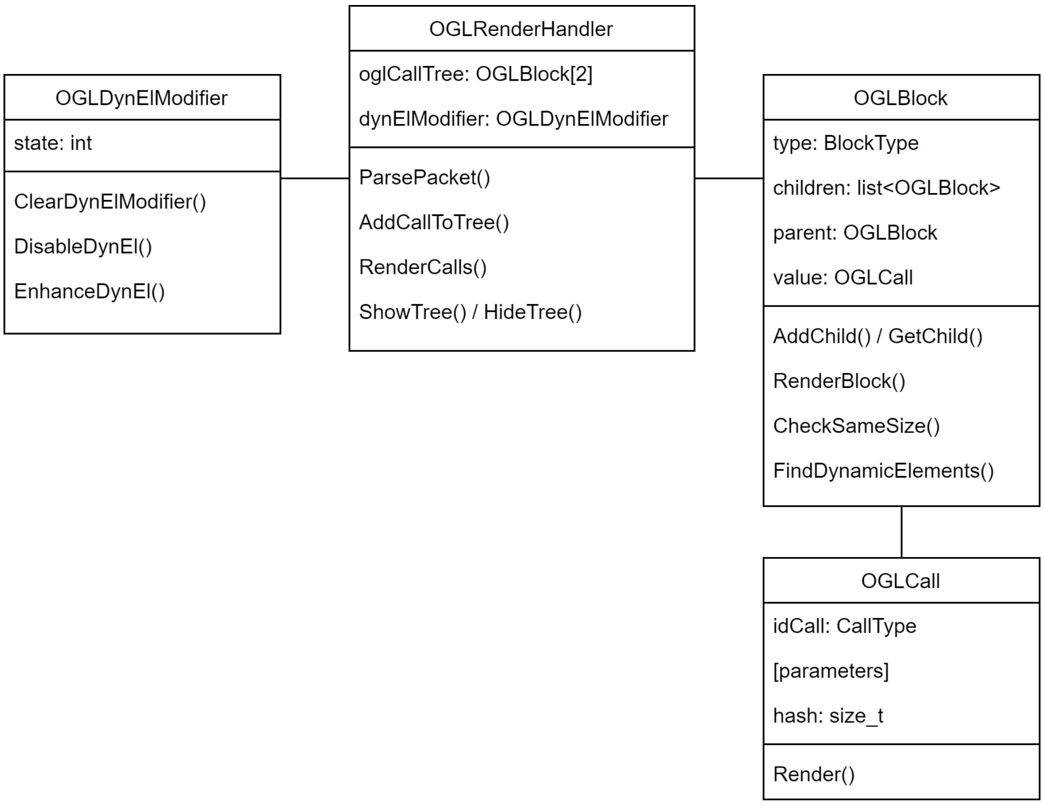


Figure 5.7: Call Handler Component Class Diagram.

The figure 5.7 shows the final class diagram of the call handler component. The scheme consists of 4 classes:

- **OGLRenderHandler**: core of the component, it manages the packets received from the library by analyzing them and adding them to the tree. It also manages the exchange and rendering of trees
- **OGLBlock**: this is the class used for tree nodes. It has the information

about the node type and, in the case discussed above, contains the OpenGL call to be made

- **OGLCall**: is the class that represent the OpenGL call
- **OGLDynElModifier**: this class is used to store changes to be made on dynamic elements. It will be described in more detail later.

The main tasks of the call handler component are two: the management of **dynamic elements** and the **parse** and **rendering** of received calls.

5.2.1 Dynamic elements

Dynamic elements are types of rendered elements that respect certain properties. The adjective dynamic was decided by us and identifies their tendency to vary.

Dynamic elements are those elements that vary from frame to frame. Basically it is those elements that are not static in the original application. For example, the 3D Cartesian chart on Matlab is a static element while the plot of the function is the dynamic element. We can find a better example on the SUMO simulator, during the simulation, the track is the static element while the car moving there, is the dynamic element.

The peculiar thing is that the OpenGL calls of static elements remain the same while those of dynamic elements vary from frame to frame.

Identifying these elements can be useful in various contexts, for example we can decide to enhance or disable the rendering of these elements or we can lighten the communication with the injected library by removing the sending of the static elements.

In our application the user can disable or improve the dynamic elements. Through a menu it is possible to select which modification to apply. The selected modification is saved inside the `dynElModifier` variable and is propagated within the tree. During the rendering phase, the node itself, if dynamic, applies these changes.

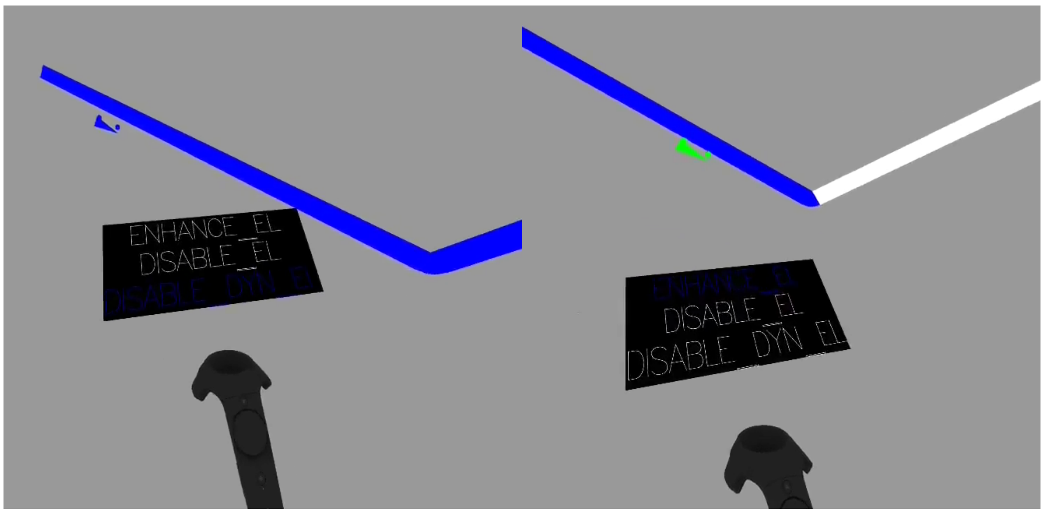


Figure 5.8: Dynamic elements, without modifier (left), with modifier (right).

The image 5.8 shows an example of dynamic element editing. On the left we have the rendered element without any modification (the car is still blue), on the right we applied the enhance modifier to the element (the enhancement modifier changes color to yellow).

5.2.2 Packet parse / Render calls

We will now describe the most important part of the call handler component, the parsing of the received packet and the building of the call tree.

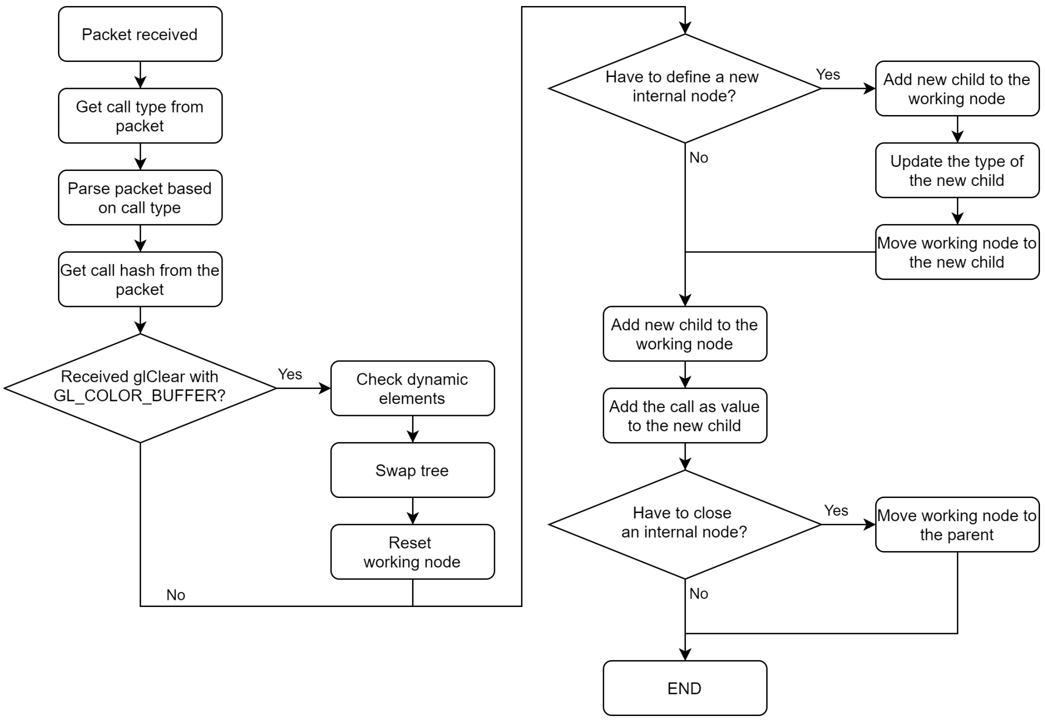
The figure 5.9 shows the flow chart of the parsing packet process. The chart can be divided into two sections, the left one where the received packet is parsed, the right where the parsed call is added to the tree.

The flow chart steps can be described as follow:

1. First of all the packet is received and the call type is retrieved (e.g. of call type could be `gl_begin`, `gl_vertex_3d`).

   After that, the packet is parsed and saved in a variable of type `OGLCall`, the hash of the packet is also stored inside the variable itself.

   The parse process varies from call to call, but the pattern is always the



Packet received
Get call type from packet
Parse packet based on call type
Get call hash from the packet
Received glClear with GL_COLOR_BUFFER?
Yes
No
Check dynamic elements
Swap tree
Reset working node
Have to define a new internal node?
Yes
No
Add new child to the working node
Update the type of the new child
Move working node to the new child
Add new child to the working node
Add the call as value to the new child
Have to close an internal node?
Yes
No
Move working node to the parent
END

Figure 5.9: Packet parse flow chart.

same. The parameters are retrieved and saved inside the variable. The hash stored is used later to identify the presence of dynamic elements.

2. Once the call variable has been created and initialized, is checked if the call is a `glClear` and if the cleared buffer is `GL_COLOR`. If so, it means that the original application is moving to the next frame (`GL_BUFFER` is the buffer that contains the rendered pixels). If true, the swap process starts.

   The swap process begins by finding the dynamic elements. The identification process is very simple, first of all is checked if the two trees contain the same number of elements. If so, proceed otherwise the check stops as elements have been added or removed and this change would ruin the next check.

   Then a one-to-one check is done by passing each node of the trees and checking the hash of each node. If the node is internal, the function continues recursively on the children, if the node is a leaf, and therefore a call, the hashes are checked. If the hashes are different, the new tree node, siblings, and all their children are marked as dynamic.

Once done with the identification of dynamic elements, the trees swap and the working node is restored. The working node is the current node where new calls are added.

3. After the swap part, add the call to the tree begins. First of all is checked if the call parsed require a new internal node to be created (e.g. `glPushMatrix` starts a new `TRANSFORM` node, `glBegin` starts a new `RENDER` node). If this is the case, a new child is added to the working node, the type of the new child is set according to the call (ex. In the example above we have `TRANSFORM` or `RENDER` as type) and then the working node is moved to the new child.

4. Then the call is added to the tree. A new child is added to the working node, its type is set as **CALL** and is given the parsed call as a value.

5. Finally, mirroring the previous check, it is checked whether the parsed call closes an internal node. (e.g. `glPopMatrix` closes a `TRANSFORM` node, `glEnd` closes a `RENDER` node). If this is the case, the working node is moved to the parent.

5.3 VRHandler

This section describes the VRHandler component which is the one that handles VR rendering and controller updating. The rendering process will not be described in technical detail because it would be difficult and not worth it.

The component uses an external library (**OpenVR**) for rendering in VR. Even when using the library, the steps involved in rendering are still very difficult. For this reason we have created this wrapper in order to simplify rendering in VR.

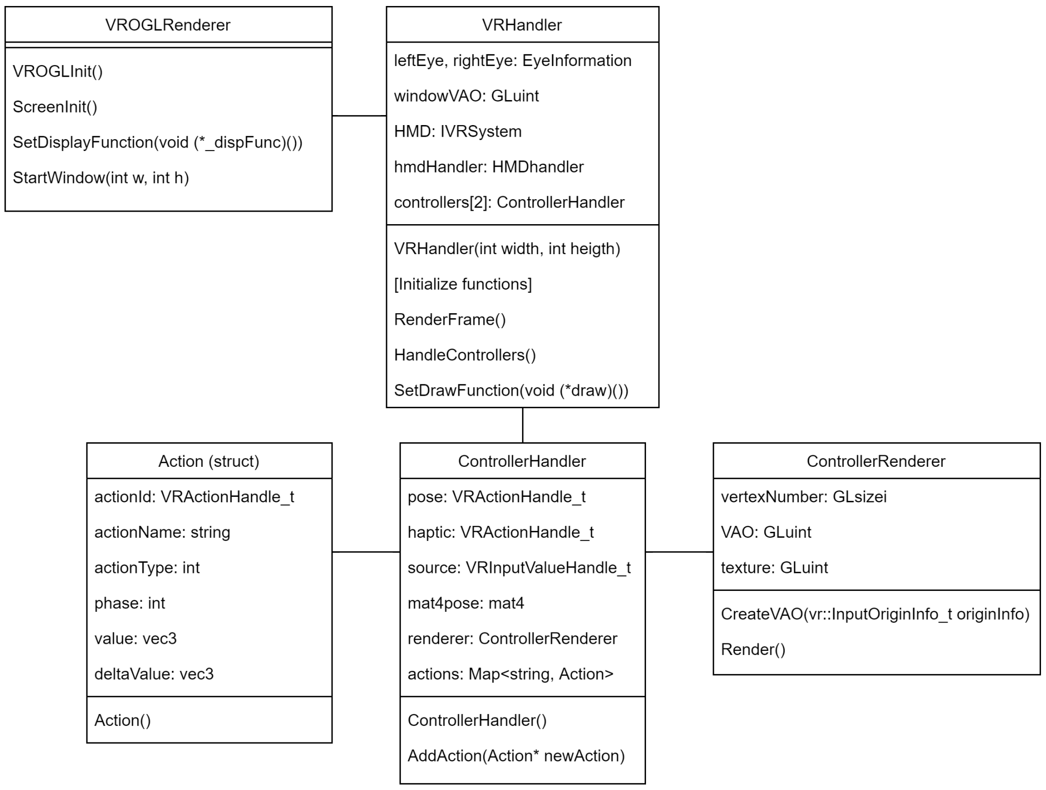


Figure 5.10: VRHandler class diagram.

The figure 5.10 shows the class diagram of the VRHandler component. As we can see from the picture there are four classes: **VROGLRenderer**, **VRHandler**, **ControllerHandler** and **ControllerRenderer**.

VROGLRenderer

This class contains the functions used to initialize the OpenGL library used by OGL2VR application to render in virtual reality. To improve modularity, the display function is not defined within the class but is linked by the user using the **SetDisplayFunction** function. This function takes the pointer to the function as input and passes it to the function **glutDisplayFunc**.

The display function is the one that manages the rendering of the frames and is called every time the frame needs to be updated.

VRHandler

This class is the core of the component. It manages all the information on the VR headset and controllers, manages their position and has the function to render elements in virtual reality. Using bfRenderFrame and **Handle-Controllers** functions the user can update the rendered frame on the HMD and update controller information such as position, pressed buttons, and so on.

ControllerHandler

This class handles the controllers information. Inside it is saved the information on the position/rotation of the controller and the list of possible actions. The latter is a list that contains information about each action connected to the controller. Through the AddAction function the user can add new actions to the controller.

How the controllers and actions works will be described in more detail later.

ControllerRenderer

This class manages the rendering of a controller on the scene. It is very simple and has only two functions, one to create the VAO used for rendering and one to render the controller.

5.3.1 VR rendering

In this section it will be described how the scene is rendered in virtual reality.

First of all we begin to describe how the VRHandler class interacts with the OGL2VR main. The figure 5.11 shows this interaction. As we explained

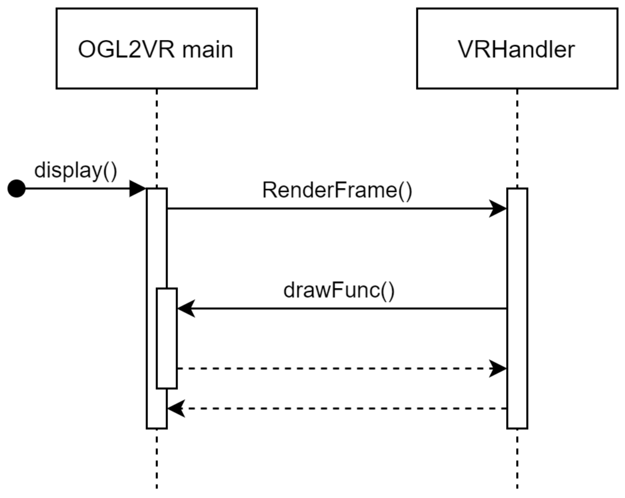


Figure 5.11: Render sequence diagram.

before, the display function (the one called each frame) is inside the main and is the one that manages the rendering of the application. The display function, each frame, calls the **VRHandler**'s render function to render the scene on virtual reality.

Once called, **VRHandler**'s render function calls **OGL2VR**'s `drawFunc` function used to render the scene. This function is not inside the **VRHandler** class for modularity reasons. The VR library should only handle the steps required for VR rendering, while the scene rendering itself should be user-defined.

In fact, inside the **VRHandler** class there is the `SetDrawFunction` function which requires the pointer of a function as input and is used to set the scene rendering function that is subsequently called during rendering in virtual reality

Before explaining the steps followed by the RenderFrame function to render in virtual reality it is better to explain how rendering in virtual reality works.

Figure 5.12 shows the VR rendering procedure. The main difference between standard rendering and VR rendering is that you need to render two images instead of just one. The two images are generated starting from the

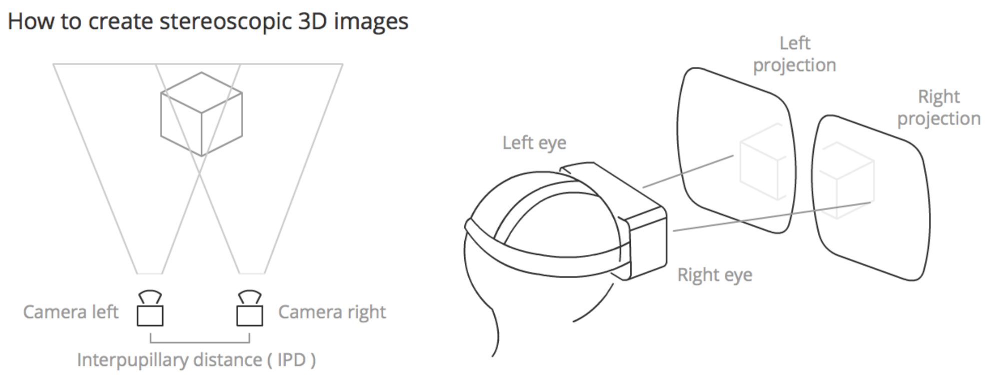


Figure 5.12: VR rendering.

same initial scene but from two different points that correspond to the position of the eyes. Once generated, some graphic changes are applied and sent to the VR headset displays.

We can now describe how the `RenderFrame` function works.

The figure 5.13 shows the steps taken by the `RenderFrame` function to render the scene in virtual reality.

1. First of all, start creating the two textures. For this it calls the `EyeTextureCreation` function which manages the generation of textures.

2. The steps taken by `EyeTextureCreation` are: prepare the rendering on the texture and perform the rendering by calling the `RenderScene` function (the function requires as a parameter which eye is being rendered). The creation and rendering of the texture is done twice, once for each eye. This leads to the creation of the two final textures.

3. The `RenderScene` function applies a translation to the camera based on the eye passed as a parameter and renders the scene by calling the OGL2VR render function pointed to by `drawFunc`.

4. Once the two textures are generated, they are rendered on the PC screen. This is done because the scene is not only visible through the

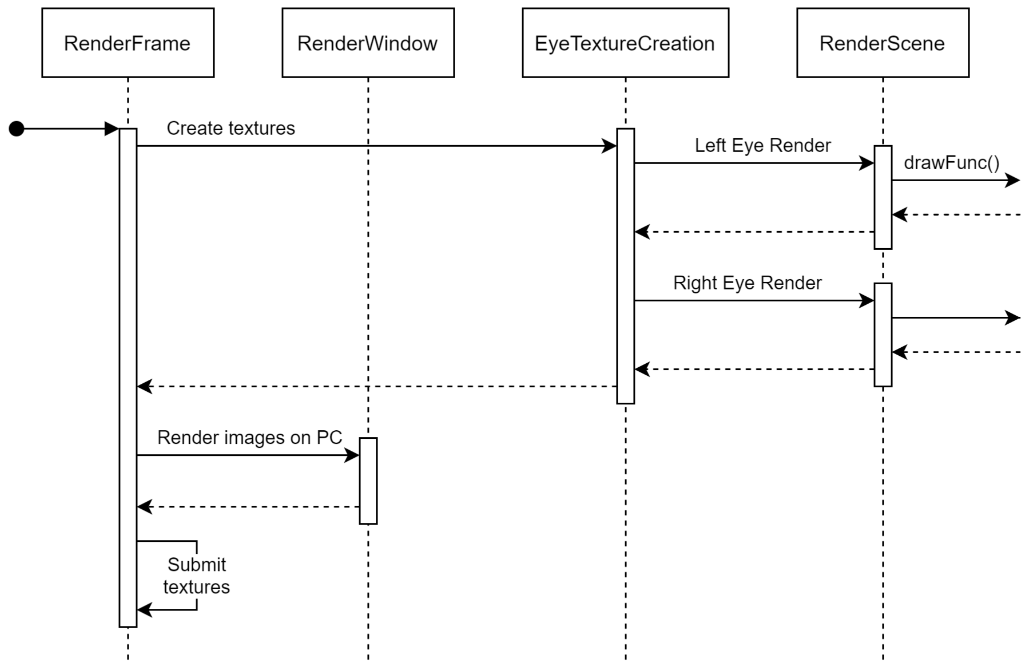


Figure 5.13: RenderFrame sequence diagram.

VR headset but also through the screen. Obviously the scene rendered on the PC has the classic VR scene format (screen split in two).

5. After the rendering on the screen the textures are finally passed to the VR headset.

5.3.2 Controller management

In this section is described how the controllers are handled in our application.

The OpenVR library we use uses a very basic approach to handle controller actions. All actions and controller bindings for each action are encoded in two json files.

```
{
  "name": "/actions/main/in/leftTrigger",
  "type": "boolean"
}
```

In this example the /actions/main/in/leftTrigger action is defined as a boolean type.

```
{
   "inputs": {
      "click": {
         "output": "/actions/main/in/leftTrigger"
      }
   },
   "mode": "button",
   "path": "/user/hand/left/input/trigger"
}
```

In this other example the action /actions/main/in/leftTrigger is bound to the input identified as /user/hand/left/input/trigger of the controller.

Using the OpenVR library, to get action information you need to follow these steps: call a library function to update action status and retrieve, using the identification string (e.g. /actions/main/in/leftTrigger) the status of the action.

This method is simple but requires the user to perform many functions to retrieve a single input. Furthermore, all inputs can only be retrieved at a specific point in the iteration making information retrieval very complex.

For these reasons we decided to create the ControllerHandler class to manage controllers, which stores the actions of its controller and their state. The updating of the actions is automatic and the user can easily retrieve the status of the different inputs without the need to implement anything.

The controller actions are saved in the actions variable of type Map<string, Action>. The action identification string is used as the key, and an Action type variable is used as the value. Through the AddAction function, the user can add other actions to the controller by passing an action type variable as a parameter. An action can be initialized by passing the identification string and an integer indicating whether the action is digital (0) or analog (1).

The Action structure is the one that contains the current information of the action. The structure contains the name of the action, the current phase of the button (`KEY_NOT_PRESS`, `KEY_DOWN`, `KEY_PRESS`, `KEY_UP`) and the value of the button in the case of analog actions (e.g. trackpad position).

Another advantage of this method is that if you are evaluating an action, you can immediately access information about its controller. Because the list of actions is contained within the controller class. In the original method, the actions were disconnected from the controllers. So, if information about the controller of the action being evaluated was needed, it had to be retrieved and processed independently.

5.4 Engine

In graphic engines such as Unreal or Unity there is the definition of `object` . An object is a graphic element that has its own shape that is rendered and has the possibility of interacting with the other elements in the scene or with the user himself. These objects, when working directly with graphics libraries, do not exist, as they are elements defined by the graphics engine itself

Since we needed to generate elements that could be interacted by the user, we decided to create a basic engine. Currently our engine only manages two-dimensional elements, as they are the only ones needed in our application.

The figure 5.14 shows the class diagram of the `engine`. We can divide the diagram into two main areas: the **object manager** and **objects**.

Objects

The **objects** area is composed of the `Object` class and its subclasses. The `Object` class contains information about the object such as the position, rotation or size of the bounding box (used for user interaction) and the Render, Destroy and CheckRay functions respectively used for Rendering, Destruction and control of interaction.

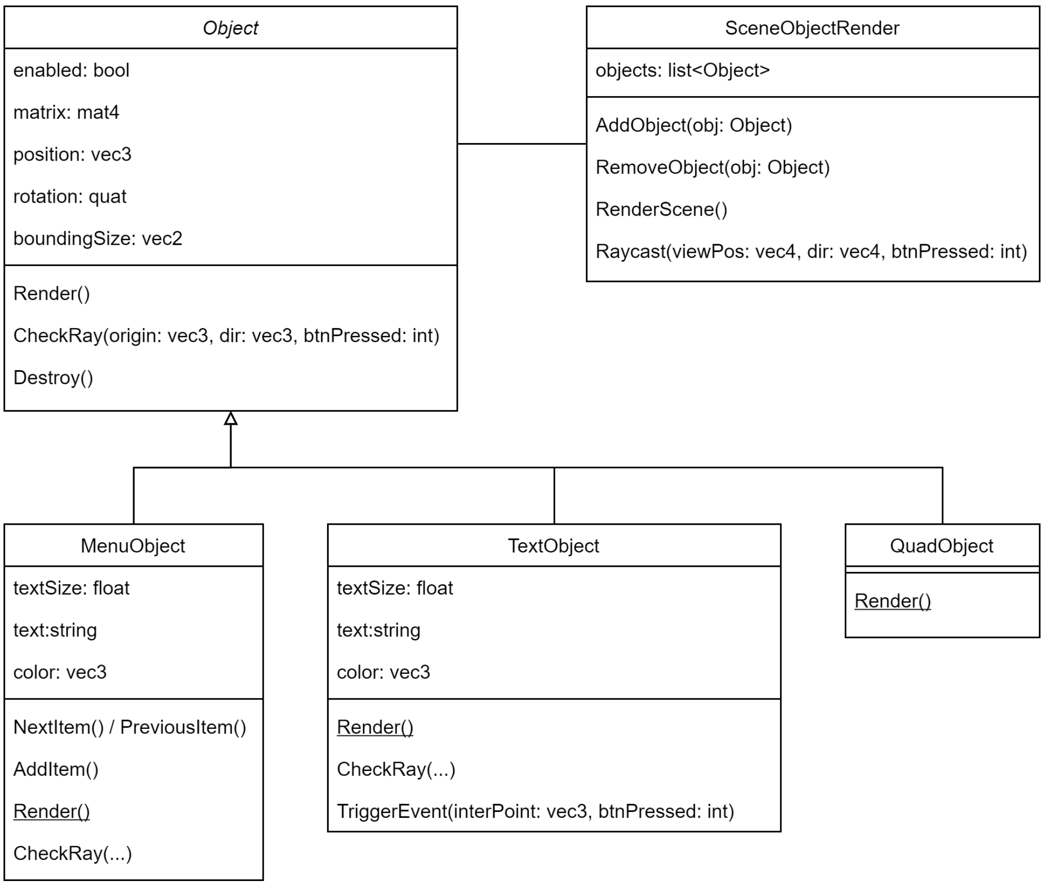


Figure 5.14: Engine class diagram.

The object subclasses override some of these functions and extend the main class with other feature:

- The `TextObject` class is the one from which each node of the tree derives and is used for its construction. The object rendered is a quad with text inside of it. The `TriggerEvent` function is called when the `CheckRay` function identifies an intersection from the ray projected by the user and the object (interaction between the user and the object).
- The MenuObject class allows the user to create menus. It is used to create menus on controllers to allow the user to navigate between possible actions. It has some functions used to add new items to the

menu or to navigate them.

An object (both of the parent class and of the subclasses), when defined, is automatically added to the list of objects of the `SceneObjectRender` class and remains there until it is destroyed by the `Destroy` function.

Object manager

All objects in the scene are managed by the `SceneObjectRender` class. This class follows the singleton pattern so its instance is accessible from anywhere in the application.

The class has the `AddObject` and `RemoveObject` functions used to manage the list of objects defined within it. Through these functions the defined objects can be added or removed from the list of objects in the scene.

The `objects` list contains all the objects currently present on the scene. All the objects inside are visible on the scene and are interactable.

The `RenderScene` function is used to render the objects within the list. The function is very simple, it just loops through each object in the list and calls its render function. The `Raycast` function is used to check if the user is interacting with any object. Its operation is similar to that of `RenderScene`, it goes through each object in the list and calls its `CheckRay` function.

5.5 PlayerInput

This section describes how the **PlayerInput** component works. The **PlayerInput** component consists of a single class called `PlayerInput` that manages navigation through menus and the activation of application functionality.

The `PlayerInput` class has access to controllers and through this it manages user interaction with menus and features. User inputs are taken by checking the action list of each controller, based on them the class show/disable menu and enable/disable functionality

There are two main menus, one for each controller: the **transform** menu and the **tree/dyn** menu.

Transform menu

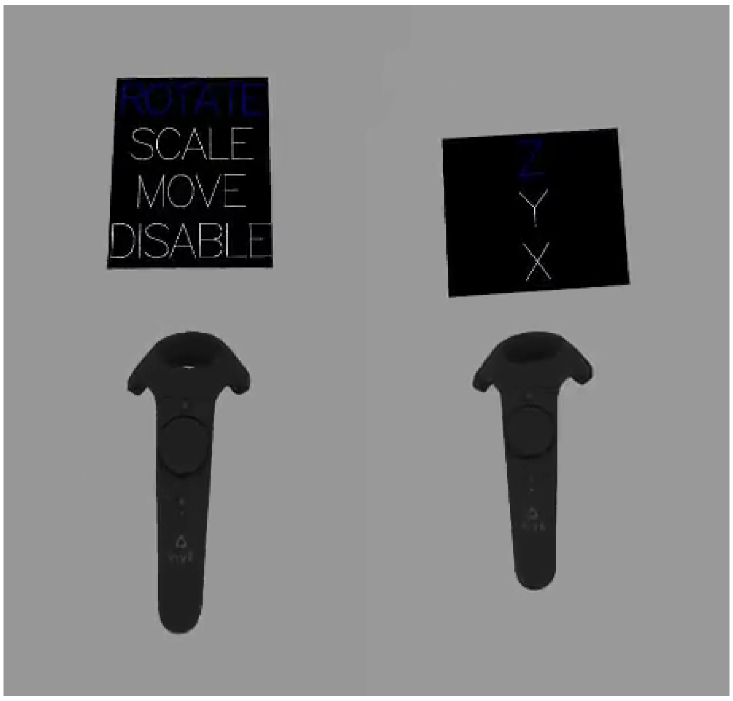


Figure 5.15: Transform menu.

The figure 5.15 shows the transform menu (left on the picture), this menu is used to apply transformations to the rendered object. These transformations can be translating, moving, or scaling. Using the controller trackpad the user can navigate between functions, when one is selected the user can apply it using the controller trigger:

- **Rotation**: pressing the left trigger the object takes the rotation of the current controller. By holding the trigger and turning the controller you can apply a rotation to the object.
- **Translation**: Pressing the left trigger and moving the controller moves the object based on where the controller is moved from its starting po-

sition. If we hold down the trigger and move the controller to the left of the initial position the object will move to the left, if we move upwards the object will move upwards. The speed of movement depends on the distance from the controller to the starting position. By pressing the right trigger, the position of the object is reset to the origin.

- **Scaling**: We have two options for scaling: uniform scaling or single axis scaling. Uniform scaling allows the user to increase the size of the object by pressing the right trigger or reduce the size by pressing the left trigger. For scaling on a single axis, the user must first enter, using the trackpad, the menu displayed on the right in the image, then select on which axis to scale. Scaling is then performed as before, right trigger to increase, left trigger to reduce.

Tree menu / Dynamic elements menu

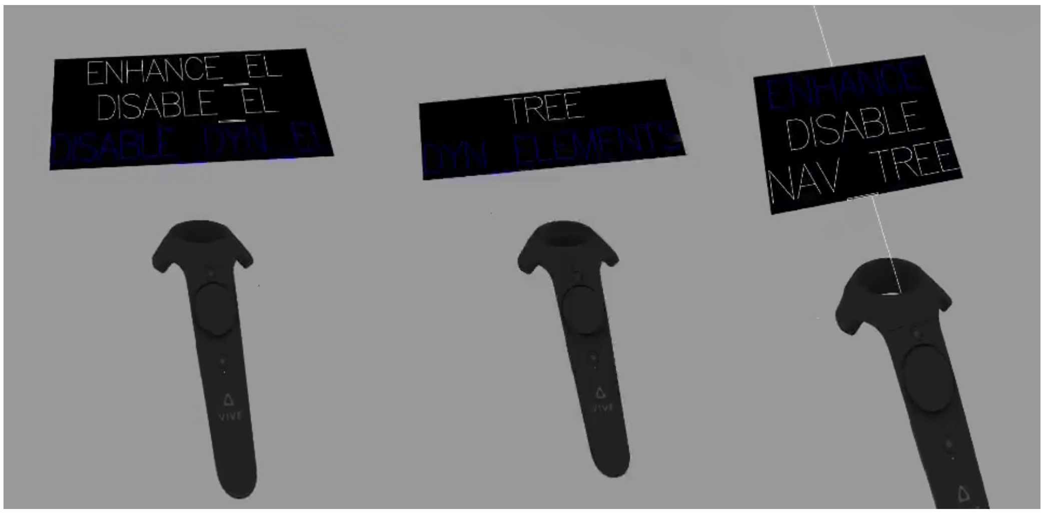


Figure 5.16: Tree/Dynamic elements menu.

The figure 5.16 shows the main menu (in the center of the image) through which it is possible to switch to the submenu for managing the tree (on the right in the image) or to the submenu for managing the modifier of the dynamic elements (left in the image). Navigating through menu options and menus is done using the trackpad.

Through the dynamic element modifier management menu (on the left in the image), it is possible to choose which modifier to apply to the dynamic elements. To select one you just need to scroll over it. Once selected, the modifier is activated immediately.

Through the tree management menu (on the right in the image) the user can interact directly with the opengl call tree and apply changes to the calls themselves. By entering the tree management submenu, the call tree is made visible and a pointer on the controller is activated (the pointer is visible in the right image). The user, through the menu, can decide what happens by interacting with the tree. The interaction with the tree is very simple, simply have to point a node with the pointer and press the trigger of the controller used. The action that is activated on that node depends on the one selected in the menu. Actions can be:

- Enhance: usable only on render nodes, it applies a yellow color to the rendered element to make it easier to identify which object in the scene corresponds to that node or to make it stand out.

- Disable: disables the selected node and all children. Useful when the injected library sends calls that ruins the final rendering (e.g. calls to render toolbars). Through this feature the user can disable these calls.

- Nav_tree: this option allow the user to nagivate the tree. By default is not shown all the tree because it would be to big for a good user interaction. For this reason is rendered only one node father and his children. Using the trackpad the user can scroll the children list and by selecting one of the nodes with this option activa can navigate though the tree. If a child is selected it goes down the tree while if the father is selected it goes up the tree.

  this option allows the user to navigate the tree. By default the whole tree is not shown because it would be too big for good user interaction.

For this reason, only one parent node and its children are rendered. Using the trackpad the user can scroll through the list of children and by selecting one of the nodes (with this option active) he can navigate the tree. If a child is selected it goes down the tree while if it is selected the father goes up the tree.

Chapter 6

Future developments

Since this was a very extensive work in which we started to create an infrastructure from scratch, there are various points that can be extended to improve the quality of the application even more. During the development, these points were not extended as they are not strictly necessary to achieve the set goal, for this reason it was decided to reach the final goal first.

The possible additions/changes to be made to the application are:

- **AI for dynamic element recognition**: Enhance identification of dynamic elements by using an AI to compare call trees and mark which calls are dynamic. By achieving high reliability it is also possible to modify how the injected library sends calls. It would be possible to control the dynamic elements directly within the injected library and send only the dynamic calls. Obviously on the OGL2VR side an algorithm is needed to add or update these calls in the call tree.

- **Server improvement**: Currently the server can only handle one connection, after which it is closed. The two possible improvements are: avoiding server shutdown after client disconnection and allowing more connections. The first is a trivial implementation, the second is much more complex. Call management needs to be extended using not just two trees but 2n trees where n is the number of clients. Each client will have its own pair of trees (ready for rendering / loading) and the server, upon receiving the call, will have to add the received call to the tree of the client that sent it.

- **Refactor player input**: It's not a real addition, the problem with player input class is that it is currently very cumbersome and it would be useful to refactor to make the class more modular.
- **Injection from OGL2VR**: Currently the source application must be run and injected before the OGL2VR run. It might be useful to allow interaction with the source application from virtual reality, allowing the user to run a pre-injected application and interact with it in virtual reality.

Chapter 7

Conclusions

This work wants to reconnect to a stream of works that have been published in the past and which have had the merit of indicating a pathway for the provision of 3D immersive experiences, also for all those applications which are not designed VR-ready. Virtual reality, however, has been moving through different stages of maturity and interest, also in recent years. At times, in fact, it has given the impression of being almost good to go for the consumer market, but so far it has always backtracked to the role of a technology with a great potential, but not yet mainstream. This may be one of the reason why the approaches, such as the ones that have inspired this work, have at some point gone cold.

Regardless of the (un)certain future of VR, our contribution wants to respond to the needs of a niche of users that have always demonstrated their interest towards immersive technologies: scientific computing and simulation research professionals. The proposed approach may hence, at once, serve an interested group of users, while fostering the development of a set of technologies which may in the near future bloom also in the general consumer market.

This work represents a first step in the direction of the final goal. Future works will require the completion of the implementation and a thorough experimentation, which may also include performance evaluation and human-computer interaction approaches, with the 3D immersive scientific computing and simulation environments supported within the proposed framework.

Bibliography

[1] G.-Z. Yang, B. J. Nelson, R. R. Murphy, H. Choset, H. Christensen, S. H. Collins, P. Dario, K. Goldberg, K. Ikuta, N. Jacobstein, *et al.*, "Combating covid-19—the role of robotics in managing public health and infectious diseases," 2020.

[2] K. Risden, M. P. Czerwinski, T. Munzner, and D. B. Cook, "An initial examination of ease of use for 2d and 3d information visualizations of web content," *International Journal of Human-Computer Studies*, vol. 53, no. 5, pp. 695–714, 2000.

[3] A. G. Sutcliffe and K. D. Kaur, "Evaluating the usability of virtual reality user interfaces," *Behaviour & Information Technology*, vol. 19, no. 6, pp. 415–426, 2000.

[4] J. J. LaViola Jr, "Bringing vr and spatial 3d interaction to the masses through video games," *IEEE Computer Graphics and Applications*, vol. 28, no. 5, pp. 10–15, 2008.

[5] W. Cellary and K. Walczak, *Interactive 3D multimedia content: models for creation, management, search and presentation.* Springer, 2012.

[6] D. A. Bowman, R. P. McMahan, and E. D. Ragan, "Questioning naturalism in 3d user interfaces," *Communications of the ACM*, vol. 55, no. 9, pp. 78–88, 2012.

[7] A. Cockburn and B. McKenzie, "3d or not 3d? evaluating the effect of the third dimension in a document management system," in *Proceed-*

ings of the SIGCHI conference on Human factors in computing systems, pp. 434–441, 2001.

[8] R. Alkemade, F. J. Verbeek, and S. G. Lukosch, "On the efficiency of a vr hand gesture-based interface for 3d object manipulations in conceptual design," *International Journal of Human–Computer Interaction*, vol. 33, no. 11, pp. 882–901, 2017.

[9] "Vr software for virtual reality design — autodesk."

[10] M. Mine, A. Yoganandan, and D. Coffey, "Making vr work: building a real-world immersive modeling application in the virtual world," in *Proceedings of the 2nd ACM symposium on Spatial user interaction*, pp. 80–89, 2014.

[11] D. J. Zielinski, R. Kopper, R. P. McMahan, W. Lu, and S. Ferrari, "Intercept tags: enhancing intercept-based systems," in *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, pp. 263–266, 2013.

[12] D. J. Zielinski, R. P. McMahan, S. Shokur, E. Morya, and R. Kopper, "Enabling closed-source applications for virtual reality via opengl intercept-based techniques," in *2014 IEEE 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 59–64, IEEE, 2014.

[13] P. O'Leary, S. Jhaveri, A. Chaudhary, W. Sherman, K. Martin, D. Lonie, E. Whiting, J. Money, and S. McKenzie, "Enhancements to vtk enabling scientific visualization in immersive environments," in *2017 IEEE Virtual Reality (VR)*, pp. 186–194, 2017.

[14] C. Shaw, M. Green, J. Liang, and Y. Sun, "Decoupled simulation in virtual reality with the mr toolkit," *ACM Transactions on Information Systems (TOIS)*, vol. 11, no. 3, pp. 287–317, 1993.

[15] C. J. Turner, W. Hutabarat, J. Oyekan, and A. Tiwari, “Discrete event simulation and virtual reality use in industry: new opportunities and future trends,” *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 6, pp. 882–894, 2016.

[16] I. J. Akpan, M. Shanker, and R. Razavi, “Improving the success of simulation projects using 3d visualization and virtual reality,” *Journal of the Operational Research Society*, pp. 1–27, 2019.

[17] F. P. Brooks Jr, M. Ouh-Young, J. J. Batter, and P. Jerome Kilpatrick, “Project gropehaptic displays for scientific visualization,” *ACM SIGGraph computer graphics*, vol. 24, no. 4, pp. 177–185, 1990.

[18] R. Brady, J. Pixton, G. Baxter, P. Moran, C. S. Potter, B. Carragher, and A. Belmont, “Crumbs: a virtual environment tracking tool for biological imaging,” in *Proceedings 1995 Biomedical Visualization*, pp. 18–25, IEEE, 1995.

[19] K. Gruchalla, “Immersive well-path editing: investigating the added value of immersion,” in *IEEE Virtual Reality 2004*, pp. 157–164, IEEE, 2004.

[20] A. Forsberg, M. Katzourin, K. Wharton, M. Slater, *et al.*, “A comparative study of desktop, fishtank, and cave systems for the exploration of volume rendered confocal data sets,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 3, pp. 551–563, 2008.

[21] X. Yang and Q. Chen, “Virtual reality tools for internet-based robotic teleoperation,” in *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 236–239, IEEE, 2004.

[22] D. Kranzlmuller, H. Rosmanith, P. Heinzlreiter, and M. Polak, “Interactive virtual reality on the grid,” in *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 152–158, IEEE, 2004.

[23] L. T. De Paolis, M. Pulimeno, and G. Aloisio, “The simulation of a billiard game using a haptic interface,” in *11th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT’07)*, pp. 64–67, IEEE, 2007.

[24] Y. Peng, Y. Ma, Y. Wang, and J. Shan, “The application of interactive dynamic virtual surgical simulation visualization method,” *Multimedia Tools and Applications*, vol. 76, no. 23, pp. 25197–25214, 2017.

[25] L. Donatiello, L. Gasparini, and G. Marfia, “Laying the path to consumer-level immersive simulation environments,” in *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT) (DS-RT’20)*, (Prague, Czech Republic), Sept. 2020.

[26] S. Bryson, “Virtual reality in scientific visualization,” *Communications of the ACM*, vol. 39, no. 5, pp. 62–71, 1996.

[27] B. Laha, D. A. Bowman, and J. J. Socha, “Effects of vr system fidelity on analyzing isosurface visualization of volume datasets,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 4, pp. 513–522, 2014.

[28] B. J. Andersen, A. T. Davis, G. Weber, and B. C. Wünsche, “Immersion or diversion: Does virtual reality make data visualisation more effective?,” in *2019 International Conference on Electronics, Information, and Communication (ICEIC)*, pp. 1–7, IEEE, 2019.

[29] N. Reski and A. Alissandrakis, “Open data exploration in virtual reality: a comparative study of input technology,” *Virtual Reality*, vol. 24, no. 1, pp. 1–22, 2020.

[30] “3d design software — sketchup.”

[31] G. Marino, D. Vercelli, F. Tecchia, P. S. Gasparello, and M. Bergamasco, “Description and performance analysis of a distributed rendering

architecture for virtual environments," in *17th International Conference on Artificial Reality and Telexistence (ICAT 2007)*, pp. 234–241, IEEE, 2007.

[32] "Techviz website."

[33] "Moreviz website."

[34] "Envelop shutdown."

[35] L. Yu, P. Svetachov, P. Isenberg, M. H. Everts, and T. Isenberg, "Fi3d: Direct-touch interaction for the exploration of 3d scientific visualization spaces," *IEEE transactions on visualization and computer graphics*, vol. 16, no. 6, pp. 1613–1622, 2010.