Viargo - A Generic Virtual Reality Interaction Library

Dimitar Valkov* Benjamin Bolte†
Visualization and Computer Graphics (VisCG) Research Group,
University of Münster, Germany

Gerd Bruder‡ Frank Steinicke§
Immersive Media Group (IMG),
University of Würzburg, Germany

ABSTRACT

Traditionally, interaction techniques for virtual reality applications are implemented in a proprietary way on specific target platforms, e. g., requiring specific hardware, physics or rendering libraries, which withholds reusability and portability. Though hardware abstraction layers for numerous devices are provided by multiple virtual reality libraries, they are usually tightly bound to a particular rendering environment.

In this paper we introduce *Viargo* - a generic virtual reality interaction library, which serves as additional software layer that is independent from the application and its linked libraries, i. e., a once developed interaction technique, such as walking with a head-mounted display or multi-touch interaction, can be ported to different hard- or software environments with minimal code adaptation. We describe the underlying concepts and present examples on how to integrate Viargo in different graphics engines, thus extending proprietary graphics libraries with a few lines of code to easy-to-use virtual reality engines.

Keywords: Virtual reality library, interaction metaphors, abstraction layers.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

1 INTRODUCTION

The field of virtual reality (VR) encompasses research on various aspects of semi-immersive or immersive virtual environments (IVEs), including the development of multimodally integrated interaction, computer graphics techniques as well as hardware technology. IVEs are often complex hardware and software systems that require application developers to incorporate knowledge from different domains such as computer science, engineering and psychology, by integrating libraries or frameworks in extensive software engineering projects. Although many applications provide multimodal, i. e., visual, auditory and haptic rendering, the most dominant sensation in IVEs is usually the vision. In almost all VR applications a computer-generated graphical environment is presented to the user by mapping tracked head movements to camera motions in the virtual world. Though such a setup provides the least common denominator of all VR applications, the graphical environments and underlying frameworks differ significantly between and within research groups, being constantly adapted for shifted research requirements, and often replaced due to rapid developments in the computer graphics community. Furthermore, interaction techniques in VEs are often implemented on a prototyping basis for a specific graphical and display environment, tightly coupled to particular hardware configurations and rendering frameworks. They are often based on a research group's locally developed rendering libraries, developer experience and preference. For these reasons, VR applications usually lack portability and reusability, which hinders collaborative work and progress in the field of complex interaction techniques, e. g., making it near-impossible to develop interaction code collaboratively with multiple research groups and VR laboratories.

Obvious effects of this situation can be observed in numerous VR demonstrations. Usually, VR systems are based on immersive or semi-immersive displays and tracking systems combining head tracking and view-dependent rendering with virtual object interaction via various input devices. While such kinds of multimodal user interfaces provide compelling immersive experiences, they often lack state-of-the-art rendering technologies, i. e., the visual appearance of the VE is often antiquated in contrast to current efforts in the game or cinema industry, which does not reflect the perceptual importance of visual stimulation in multimodal input and output environments. Often this can be traced to developers integrating hardware technology and designed interaction concepts in locally developed graphics libraries based directly on *OpenGL* or *DirectX*, or open source graphics engines such as *OGRE*, *OSG* or *IrrLicht*. Many VR libraries and toolkits have been developed on top of these rendering engines, which allow to abstract the hardware interface from the application, but cause significant re-implementation when porting a VR application to another graphics engine. In particular, to our knowledge none of these commercial or free VR toolkits provide sufficient support for all current state-of-the-art rendering systems or game engines such as *Crytek's CryEngine 3*, the *Unreal Engine 3*, *Valve Source Engine* or *Unity 3D*.

In effect, if a VR developer wants to use such a game engine, she is usually forced to implement the plugin or layer between her currently used VR toolkit and the engine, which is often not very stable to changes in the graphics engine. Furthermore, most VR toolkits [3, 9] are mainly fully integrated frameworks providing application programming interfaces (APIs) for definition of VEs and behavior, while hiding the complexity of hardware event handling and visualization processes, i. e., proving difficult and time-consuming for users to register low-level processes with a state-of-the-art graphics engine, or adding support for hardware input or output devices. As a result, migration of VR interaction techniques from one environment to another usually leads to dropping support for many of the implementations, causing transfer of interaction techniques between researchers to be usually limited to general descriptions of the concepts, i. e., being sufficiently simple for easy re-implementation or causing loss of features on the target system.

In this position paper we introduce Viargo [vi:ɪ:go:], an interaction subsytem that allows definition of interaction metaphors between the application layer and low-level tracking or device input, allowing easy transfer of interaction code between different graphics or game engines, VR laboratories and research groups. In contrast to other VR toolkits, Viargo does not provide an API layer to encapsulate the graphical subsystem, but is conceptional self-contained and provides a localizable interface, i. e., allowing to synchronize interaction techniques with the application at a single point.

*e-mail: dimitar.valkov@uni-muenster.de
†benjamin.bolte@uni-muenster.de
‡gerd.bruder@uni-wuerzburg.de
§frank.steinicke@uni-wuerzburg.de

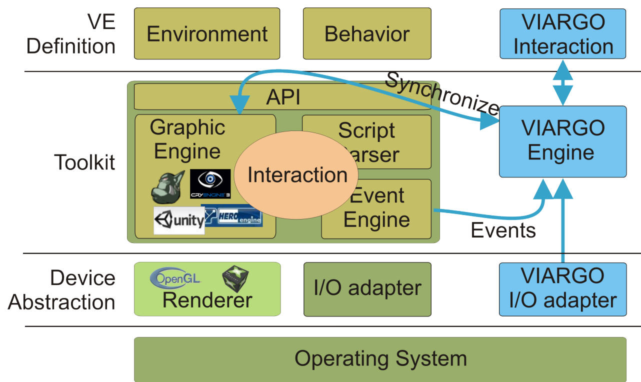

Figure 1: Block diagram of a typical VR framework and the integration of the Viargo system.

2 Related Work

For implementations of interactive systems, in particular VR software systems, it is important to subdivide the system such that modifications to one component can be made with minimal impact on the others. Many VR software systems and VR toolkits have been proposed to support the development of VR applications. These systems usually provide interfaces for specification of VEs or interactions, abstracting hardware and device handling from layout and dynamics of the virtual scene. However, most of these systems do not providing sufficient modularity for rapid integration of the currently most advanced rendering systems, and cannot be easily integrated in existing VR hardware or software environments.

Examples of traditional VR software systems that provide high-level interfaces for developers are *VPL's Body Electric* [1] and *SGI's Open Inventor* [8]. These systems allow users to specify relations between virtual objects and input or output devices in a dataflow diagram editor, but limiting program modularity [3], in particular, not providing a dedicated interface for integrating a modern rendering engine. Various other VR software systems provide rapid prototyping environments for creating interactive computer graphics applications without requiring a strong technical background, but abstracting control of graphics and rendering [11]. Many other VR libraries are focused on specific display technology or applications, making it difficult to share interaction techniques developed for one hardware setup with another [5]. For instance, some libraries are based on specific rendering platforms [14, 4], or focus on particular areas of virtual or augmented reality, such as the Studierstube project [12] or ARToolkit [13].

Integrated software systems, such as *VRJuggler* [3], *DIVERSE* [9], *Vizard* [18] or *Virtools* [16], allow building high-end VR applications and have been designed to overcome the drawbacks of some of the previous systems. For instance, *VRJuggler* establishes a modular architecture for different devices, and provides a graphics-level API to interface with the virtual scene. Some of the other systems allow to choose from multiple wrapped rendering frameworks or provide plugins for different libraries. However, some experience with low-level programming in these environments is required for being able to write a plugin for state-of-the-art graphics rendering or game development environments. Moreover, developed interaction techniques often cannot be shared with other research groups due to incompatible versions or licensing issues.

As discussed above, these developments often lack standardization of VR system components and architecture, but incorporate abstractions of hardware interaction. Some research was conducted on flexible abstraction for interaction metaphors or interface definition, such as *VITAL* [6], *InTml* [7] or *CHASM* [15].

Although many systems are available for creating and developing VR-based applications, due to compatibility and customization issues universities and research institutions tend to write their own VR-based extensions to open source or commercially available graphics or game engines. In the next sections we present an alternative approach for encapsuling VR interaction metaphors in an easily sharable and integrable interaction subsystem.

3 VIARGO

A common VR application (cf. Figure 1), whenever based on full integrated framework or composed of different libraries, usually contains two main components - a *graphic engine* and an *event system*. In addition, a script or XML parser is often integrated into the system to provide options for simple customization of its properties without the need for recompiling the application. The main task of the event system is to hide the complexity of the hardware components and provide transparent abstraction to the sensor data. This is often done by providing an API layer for accessing the sensor data or by encapsulating this data into generic structures (*events*) which are then broadcasted to the corresponding system components. As mentioned in Section 1, the interaction metaphors in such systems are usually integrated into the application with parts of their code residing in the event system (e. g., filters, event re-processors) and part of it residing in the graphic engine (e. g., camera manipulators, object selectors).

The main goal of Viargo is to provide a self-contained system for specifying interactions, which can be integrated in arbitrary rendering engines with minimum effort. The library provides its own set of hardware abstraction components, and can be combined with a graphic or game engine extending it to a basic VR framework. Nevertheless, the application is still free to acquire the sensor data by its own and then inject it into the Viargo core. Viargo could be driven either by the application's main rendering loop or by separate threads. Furthermore, Viargo provides an internal state system, which contains structures commonly found in the most graphic engines such as the virtual camera's position and orientation. This state system, if synchronized with the rendering engine, is used to exchange the data between Viargo and the application.

3.1 Architecture

A diagram of the framework components of Viargo is shown in Figure 2. The core data transfer unit in Viargo is an *event*. Viargo events are simple specializations of a common abstract base class used to transfer data from the device abstraction units, which acquire the sensor data to the interaction processing units, i. e., the *interaction metaphors*. The *event engine* is used to dispatch this data flow and manage multi-threading or sequencing issues. The result of the data processing could be either a change of the state in the state system, or a generation of new events, which enables multi-level data processing.

3.1.1 Device Abstraction

A characteristic feature of all VR applications is that they use data from different types of sensors, such as tracking or analog input devices. The type of the used sensors as well as the way the data is acquired differs considerably from one setup to another, although the semantics are very similar. For instance, a commercial marker-based tracking system and a custom face recognition program could both be used to determine the user's head position, i. e., provide semantically equivalent data, but the format of the output data (e. g., units, coordinate system) and the way it is fed to the application usually varies significantly. In order to provide unified abstraction of the sensors Viargo uses the, so called, *device units*. Device units are adapters for each sensor or protocol (e. g., VRPN) which translate the acquired data into events and broadcast them to the rest of

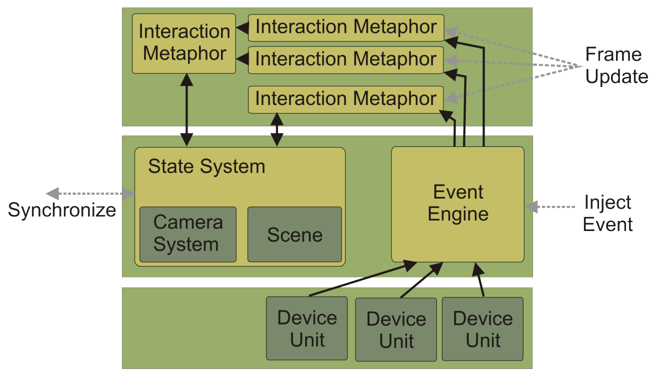

Figure 2: Component diagram of the Viargo system.

the system. A single device unit could be used to provide different types of events, such as changes of accelerometer data, button states, etc. However, different device units could also provide the same type of events, i. e., tracking events from different tracking systems.

3.1.2 Interaction Metaphors

Interaction metaphors are implemented as event processing units encapsulated as derivations of a simple abstract class. They receive events from the core system and process them to change some of the parameters in the state system, which are then transferred to the rendering engine by the time of synchronization. Alternatively, a metaphor could be used as a device unit, to feed back events to the Viargo core [1] The data processing can be realized at three different stages: (i) as soon as an event appears, (ii) in the main synchronization thread or (iii) after all events have been processed, providing great flexibility for designing interactions. An example code listing of a Viargo metaphor is shown in Listing 1.

```
1
2  class MyInteractionMetaphor : public Metaphor {
3
4  public:
5      // Every metaphor instance has a unique name
6      MyInteractionMetaphor(string name)
7      :Metaphor(name)
8      {
9      }
10
11     // [asynchronous]
12     // react on incoming event when received
13     bool onEventReceived(Event* e)
14     {
15         if (typeid(*e) == typeid(TrackintEvent))
16         {
17             // pre-process the event
18             // [...]
19             // ... and enqueue it for later
                   evaluation
20             return true;
21         }
22         else
23         {
24             // ignore all other events
25             return false;
26         }
27     }
28
29     // [synchronous]
30     // process all enqueued events
31     void onEvent(Event* e)
32     {
33         if (typeid(*e) == typeid(TrackintEvent))
34         {
35             TrackintEvent* te = (TrackintEvent*)e;
36             vec3f trackedPos = te->position();
37             // do something with the data
38             // [...]
39
40             // ...and manipulate the 'main' camera
41             Viargo.camera("main").setPosition(...);
42         }
43
44         else if(...) // process other events
45         {
46         }
47     }
48
49     // handle frame updates
50     void update(float timeSinceLastUpdate)
51     {
52         // do something
53     }
54 };
```

Listing 1: Example Viargo metaphor.

[1] This could be particularly useful in multi-touch applications, where the raw touch events are usually only used to detect predefined high-level gestures, which are then mapped to manipulations.

The function `onEventReceived(...)` is called asynchronously at the time the event is received in the event engine. The function returns a boolean value, which decides if the event should be saved in the metaphors internal event queue for later evaluation in the main synchronization thread, or if it have to be discarded (since it is already processed or not needed by the metaphor). In the main synchronization thread all events previously saved in the event queues are consequently passed through the `onEvent(...)` function for evaluation. The event engine takes care in this case, that the events are evaluated in the same temporal sequence, as they are received by the different metaphors. Finally, after all events in the event queues of all metaphors are evaluated their `update(...)` functions are called, providing the metaphors with an option to perform frame specific actions. For instance, a metaphor may only save the most actual tracking position during the execution of the `onEvent(...)` function and perform the actual camera manipulations only once per frame, in the `update(...)` function.

In addition the metaphors have access to the entire Viargo core, giving them the opportunity to change the configuration on the fly, e. g., adding/removing interaction metaphors, or enabling/disabling device units.

3.1.3 State System and Synchronization

Viargo can be thought of as a VR framework with a null rendering system, which we call *state system*. The state system consists of abstractions of components found in all rendering engines, e. g., cameras and scene representations. This architecture enables metaphors to change the state of the abstract components instead of changing the state of the graphic engine. The transfer of the abstract component's state to its real counterpart is then made at a single place, usually just before a new frame is rendered.

Camera Abstraction

An example how a simple synchronization of a single camera could be implemented in arbitrary graphics engine with open rendering loop is shown in Listing 2.

```
// ... rendering loop

vec3f position, target, up;
float left, right, bottom, top, zNear, zFar;

Viargo.update();
Viargo.camera("main").getLookAt(position, target,
    up);
Viargo.camera("main").getFrustum(left, right,
    bottom, top, zNear, zFar);

applicationCamera.setLookAt(...);
applicationCamera.setFrustum(...);

// render all ...
```

Listing 2: Synchronization of Viargo's main camera with the application.

In the example code, the event processing is executed with a call to `Viargo.update()`, and then the application's camera is overwritten with Viargo's main camera representation. This method is useful if the interaction techniques are only implemented in Viargo, but leads to overriding interaction techniques implemented in the application itself. Furthermore, multiple cameras may be needed, which have to be synchronized. An example showing two-sided synchronization of multiple cameras is shown in Listing 3.

```
//.. rendering loop

vec3f position, target, up;
float left, right, bottom, top, zNear, zFar;

// synchronize Viargo with application
while(Viargo.cameraSystem().hasNext()) {
    viargo::Camera& vrgCam =
        Viargo.cameraSystem().next();
    // find corresponding
    // application camera
    appCam.getLookAt(position, target, up);
    appCam.getFrustum(left, right, bottom, top,
        zNear, zFar);

    vrgCam.setLookAt(...);
    vrgCam.setFrustum(...);
}

// process all events
Viargo.update();

// synchronize application with Viargo
while(Viargo.cameraSystem().hasNext()) {
    viargo::Camera& vrgCam =
        Viargo.cameraSystem().next();
    // find corresponding
    // application camera
    vrgCam.getLookAt(position, target, up);
    vrgCam.getFrustum(left, right, bottom, top,
        zNear, zFar);

    appCam.setLookAt(...);
    appCam.setFrustum(...);
}

// render all ...
```

Listing 3: Two-sided synchronization of multiple cameras.

In this case all cameras specified in Viargo receive the parameters of their counterparts in the application. Then all received events are executed and finally the parameters of the cameras of Viargo are communicated back to the application. In order to simplify the mapping between the cameras of Viargo and the application, a property set is integrated in each camera abstraction. A property set consists of name-value pairs in which different variable types, such as pointers, names, IDs, etc., can be saved.

Scene Abstraction

The second abstraction currently implemented in Viargo's state system is a general purpose scene graph. All objects relevant for interaction can be added to this scene graph with their relative position, orientation and properties. In addition, each object has its own property set, which allows saving arbitrary parameters needed to be manipulated by the interaction techniques, such as materials, colors, object type, etc. For instance, consider an interaction technique, which allows to change the position of a light source. In this case the application's light sources could be registered to the Viargo's scene interface with a type property identifying it as light. During the system initialization a new Viargo scene object is created with an unique name as shown in the Listing 4.

```
//.. initialization
SceneNode* node = new SceneNode("myLight");
node->setRelativePos(...);
node->setRelativeOri(...);

Viargo.scene().root().addObject(node);

// [...]
```

Listing 4: Registering an object to the Viargo scene abstraction.

Furthmore, certain proprties can be assigned to each scene objects, for example, via `node->setProperty("ObjectType", ...);` or `node->setProperty("Color", ...);`

In the rendering method, this object is synchronized with the applications light source on each frame as shown in Listing 5.

```

//.. rendering loop
SceneNode& light = Viargo.scene().find("myLight");

appLight.setPosition(light.getRelativePos());
appLight.setOrientation(light.getRelativeOri());

// render all
```

Listing 5: Updating object of the Viargo scene abstraction.

Certain parameters can be modified according to the specification in the interaction metaphor, e. g., `appLight.setColor(light.getProperty("Color"));`.

Alternatively, a dedicated (recursive) function can be used to synchronize the entire scene abstraction with the application's scene. The geometry of the registered objects in the scene graph is described as a tree of hierarchical, axis-aligned bounding boxes. The scene abstraction provides options for searching single or multiple objects by an application or metaphor specific predicate and to apply a manipulator function to all objects in the scene.

3.1.4 Implementation

We have designed the Viargo API to be as simple as possible, still providing great functionality and reduce the amount of code that has to be written in order to integrate it in the target application. The library is implemented in C++ with source code and build system (CMake) being platform independent. The main component of the API is the core engine represented by the `Viargo` singleton class. The core engine can be initialized with an XML file, which

(a)

(b)

Figure 3: Operators using the example applications developed with Viargo within the scope of the (a) AVIGLE project and the (b) LOCUI project.

contains descriptions of all the cameras and their initial parameters, as well as the device components, which have to be loaded and the initially registered metaphors. All metaphors are implemented as a derivation from the abstract class `viargo::Metaphor`, and have to be registered with a unique name to the core engine. This allows to retrieve them at any point in the code. In the same way, all device components have to be derived from the base class `viargo::Device` which provides them access to the event engine. Again, all device components have to be registered to the core engine with a unique name. Finally, the events are specializations of the base class `viargo::Event` and can be differentiated using run-time type identification (RTTI).

In effect, the code additions that have to be made to the targeted rendering engine are a simple call to Viargo's initialization function with the path of the initialization XML file and a synchronization code akin to the code presented in Listings 2, 3 and 4. This is also the only code that has to be adapted if migrating from one rendering engine to another. Further interaction techniques can be developed entirely within the scope of the Viargo system, and Viargo device components can be implemented for additional hardware.

3.2 Examples

We use the Viargo library within the scope of our research projects with open source rendering engines, such as OGRE, IrrLicht and OSG as well as with state-of-the-art game engines, such as *Crytek's CryEngine 3* or *Unity 3D*. In all cases the integration of the library within the engine was straightforward as explained above.

In the following, we describe two example applications as illustration of the diversity of use cases and setups supported by Viargo.

3.2.1 Multi-Touching Stereoscopic Objects

This test application has been developed in the scope of the AVIGLE project [17]. The goal of this project is to explore novel approaches for remote sensing based on a swarm of Miniature Unmanned Aerial Vehicles (MUAVs). These MUAVs are equipped with different sensing and network technologies. At the end of the pipeline the currently available sensor data from the MUAVs and their status are displayed to a human operator, while new data continuously arrives. The user interacts with this visualization, for instance, to change the viewpoint in the VE. In addition, she can define new positions for each MUAV moving their visual representation in the VE, as shown in Figure 3 (a). Since MUAVs within a swarm usually fly at different altitudes, stereoscopic visualization is essential to provide additional depth cues.

For the initial implementation of the user interface we have used a passive stereoscopic, multi-touch enabled projection wall and tracked the user's head position with an optical tracking system. The VE was initially implemented with OGRE and later on changed to the IrrLicht game engine. We use the Viargo framework to provide position and orientation for stereoscopic rendering based on head-tracking events. Furthermore, the viewpoint and the virtual MUAVs were manipulated via multi-touch interaction techniques developed entirely within the Viargo framework.

3.2.2 Locomotion in IVEs

The second test application implements the so called *jumper metaphor*. The application is developed in the scope of the LOCUI (Locomotion User Interfaces) [10] project. The goal of this project is to investigate the benefits and limitations of real walking within IVEs. The jumper metaphor combines natural direct walking with magical locomotion through large-scale IVEs. The key characteristic of the jumper metaphor is that it supports real walking for short distances, such that the user can walk around objects, or use small head movements to explore the environment while perceiving motion parallax and occlusion effects similar to the real world. To travel over large distances, the metaphor predicts the planned travel destination by monitoring the user's viewing direction. The user could initiate a virtual jump, which starts a smooth viewpoint animation that transfers her to the corresponding target position (cf. Figure 3 (b)).

This application was implemented for IVEs based on head-mounted displays (HMDs). The VE was created and visualized with Crytek's CryEngine 3. An optical tracking system was used for tracking the user's head position in combination with a gyroscope (InterSense's Wireless InertiaCube3) for tracking the users head rotations.

4 Initial Evaluation and Feedback

Since one year we are using Viargo in the context of research projects, but also student seminars. In the scope of student projects, several students have used Viargo for developing their applications using a variety of hardware systems (e. g., CAVEs, HMDs, multi-touch walls, hand-held devices, Microsoft Kinects, AR drones). To receive feedback of different users, we encouraged the students to inform the seminar advisor if they had questions or comments about the Viargo library. In addition, we performed informal interviews and questionnaires with students, who have worked at least

3 months with Viargo. The major observations concerning the demands in terms of ease-of-use, flexibility and performance, as described in [2], as well as problems with the use of Viargo are discussed in the following.

Students were able to modify existing interaction metaphors and writing simple custom interaction metaphors very efficiently. For instance, object selection and manipulation (i. e., translation, rotation and scaling) metaphors were implemented as well as camera manipulation metaphors such as zooming could be implemented within 2 hours. Furthermore, students stated that Viargo provides an easy interface to the hardware, i. e., instead of directly processing the low-level data in different formats from various hardware devices, events containing preprocessed data allow students to use various hardware without knowing the underlying low-level data format.

Processing abstract events showed another advantage in terms of flexibility, as students were able to easily exchange their interaction metaphors independently of the used hardware. For instance, an interaction metaphor for mapping a skeletal posture onto a virtual avatar was transferred from tracking using active LED markers to tracking using the Microsoft Kinect by adjusting only the coordinate space differences. Moreover, students were able to easily extent the Viargo library to support new hardware devices using either the VRPN protocol, which is already supported by Viargo or writing a custom device using Viargo classes for network communication. In addition to the flexibility in terms of hardware support, the integration of Viargo into current state-of-the-art rendering systems such as Crytek's CryEngine 3.0 was achieved by the students within less than 30 minutes. Even their aforementioned initial metaphors could be easily transferred to work with these rendering systems.

The majority of problems or inconveniences that occured was caused by difficulties of the students with compiler settings or conceptual ideas. For example, some students placed the code of new metaphors in the Viargo project, which leads to a necessary rebuild of both the Viargo library and the application project itself. Furthermore, the integration of the Viargo library into Crytek's CryEngine 3.0 requires to change the default project compilation settings to enable run-time type information support. Some students had also difficulties with the network connection itself, e. g., firewall setting issues or conceptual issues concerning data formats for data exchange, but less with the implementation in Viargo itself. However, most of these issues were easily fixed by providing the students with instructions to avoid such pitfalls.

5 Conclusions and Future Work

In this position paper we introduced Viargo which serves as additional software layer that is independent from the application and its linked libraries. We explained the architecture and discussed its benefits. We described how interactions can be implemented and easily ported to different graphics engines.

Although, the Viargo library is already used in different VR-based applications, it is still under development and not very well documented. In the future, we will further evaluate and test the library in different use-cases in order to identify drawbacks and limitations.

Acknowledgements

The work was supported by grants from the Deutsche Forschungsgemeinschaft (DFG) in the scope of the iMUTS and LOCUI projects. We would like to thank all students for their thorough feedback and evaluation.

References

[1] Y. Adachi, T. Kumano, and K. Ogino. Intermediate Representation for Stiff Virtual Objects. In *Proceedings of IEEE VRAIS*, pages 195–203, 1995.

[2] A. Bierbaum and C. Just. Software Tools for Virtual Reality Application Development. In *Course Notes for SIGGRAPH 98 Course 14, Applied Virtual Reality*, 1998.

[3] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE Proceedings of VR2001*, pages 89–96, 2001.

[4] R. Blach, J. Landauer, A. Roesch, and A. Simon. A Flexible Prototyping Tool for 3D Real-Time User Interaction. *ACM Proceedings of Eurographics Workshop of Virtual Environments (VE'98)*, pages 195–203, 1998.

[5] C. Cruz-Neira. *Virtual Reality based on Multiple Projection Screens: The CAVE and Its Applications to Computational Science and Engineering*. PhD thesis, University of Illinois at Chicago, 1995.

[6] M. Csisinko and H. Kaufmann. Vital - the virtual environment interaction technique abstraction layer. In *Proceedings of the IEEE Virtual Reality 2010 Workshop: Software Engineering and Architectures for Realtime Interactive Systems*, pages 77–86. Shaker Verlag, 2010. Vortrag: IEEE Virtual Reality 2010, Boston, USA; 2010-02-00.

[7] P. Figueroa, W. F. Bischof, P. Boulanger, H. J. Hoover, and R. Taylor. Intml: A dataflow oriented development system for virtual reality applications. *Presence: Teleoper. Virtual Environ.*, 17:492–511, October 2008.

[8] O. I. A. Group. *Open Inventor C++ Reference Manual*. Addison-Wesley, 2005.

[9] J. Kelso, L. Arsenault, S. Satterfield, and R. Kriz. DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments. In *Proceedings of Virtual Reality 2002 Conference*, pages 183–190. IEEE, 2002.

[10] locui.uni muenster.de. Locui - virtual locomotion user interfaces. 2011.

[11] R. Pausch, T. Burnette, A. C. Capehar, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality. In *IEEE Computer Graphics and Applications*, pages 195–203, 1995.

[12] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavari, M. Encarnação, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. In *PRESENCE - Teleoperators and Virtual Environments 11(1)*, pages 32–45, 2002.

[13] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled Simulation in Virtual Reality with the MR Toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, 1993.

[14] H. Tramberend. AVANGO: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality '99*, 1999.

[15] C. Wingrave and D. Bowman. Tiered developer-centric representations for 3d interfaces: Concept-oriented design in chasm. In *IEEE Virtual Reality Conference VR '08*, pages 193–200, 2008.

[16] www.3dviavirtools.com. 3dvia virtools vr library. 2011.

[17] www.avigle.de. Avigle - avionic digital service platform. 2011.

[18] www.worldviz.com/products/vizard. Vizard vr toolkit - rapid prototyping for novices. 2011.